Uninformed Search - Chap 3 of Russell/Norvig

Prof. Andrew Shallue – CS 338

September 3, 2024

1 Setup

The focus in this chapter is on problem-solving agents, i.e. agents that can maximize their performance measure through formulating a goal, then discovering a sequence of actions that enables the agent to reach that goal.

For example, suppose I have a map of Illinois. I want to drive from Bloomington to St. Louis. To reach my goal, I can develop a list of roads and turns that would allow me to reach my destination. Then when I execute that plan, I can be confident that my behavior is rational.

For this to work, our environment needs to have specific properties. The environment needs to be:

- Observable, so the agent knows the current state and its connections. The environment need not be fully observable; we can pursue uninformed search if we only know the current state. The closer the environment gets to fully observable, the more efficient our search can be.
- Discrete, so that there is a finite number of paths to check. If the environment is continuous, the techniques for search become calculus-based rather than discrete-math based.
- Deterministic, so that between when a goal is formulated and executed, nothing surprising happens and each action has only one pre-determined outcome.

Theorem 1. *Under the above assumptions, the solution to any problem is a finite, fixed sequence of actions.*

For search to be applied, we need to model our problem so that it fits the assumptions above. Such a model is formulated through the process of abstracting away some details so that other details come to the foreground. Always remember, all models are lies, but some are more useful than others. A good model is one that is easier to solve than the real-world problem, and for which the solution can be extrapolated to the real-world problem.

The ingredients needed for a search model are:

1. initial state
2. given a particular state, the set of potential actions
3. given a set of potential actions, the set of potential new states that can be reached from the previous state
4. a goal test, so that we know when the goal is reached
5. a path cost function that assigns a numeric cost to each path.

   The optimal solution is the path that leads from the intial state to a goal state, with the lowest total path cost among all possible paths.

2 Surprising example of search

The most obvious example of searching is any sort of route-finding problem, where a graph can be superimposed onto a map.

A non-obvious example of search is solving the 8-puzzle. This puzzle consists of a 3-by-3 board with numbered tiles 1 through 8 and a blank space. A numbered tile adjacent to the blank space can slide into the blank space. The goal is to reach the following configuration:

	1	2
3	4	5
6	7	8

The initial state could be anything, let's take this as an example:

7	2	4
5		6
8	3	1

Notice that this initial state has four successor states. The 2 could slide to the middle, leaving the blank in the upper center. Alternatively, I could slide the 5, 6, or 3 into the center. If I slide the 2 into the middle, that new state has two successor states, because either the 7 or the 4 could slide into the space which was created.

If we continue with this thought experiment, we can model the entire state space as a tree. The initial state is the root. Its successor states are the states at the next level. We don't want our search tree to have any loops, so if a successor state would be repeated we ignore that edge. Leaf states are then the states with no new successors, and a solution to the search problem exists if the goal state is somewhere in the tree.

3 Comparing search algorithms

The performance measure for the agent is to minimize the path cost between the intitial state and the goal state. To solve this problem the agent will perform a search, which

will be implemented by some search algorithm. But which search algorithm should we choose? It depends, specifically on the following factors:

- Completeness: the algorithm is guaranteed to find a solution if it exists,
- Optimality: the algorithm is guaranteed to find and identify the optimal solution,
- Time complexity: the amount of time required for the algorithm to run, usually expressed in big-Oh notation
- Space comlpexity: the amount of memory required for the algorithm to run.

4 General description of search algorithms

Given the setup above, one can give pseudocode for search algorithms on both trees and graphs.

If we know the state space is a tree (i.e. no loops), we can do the following:

1. Tree-Search(problem) returns **solution** or **failure**
2. Frontier initializes via initial state of the problem
3. Loop do
4. If the frontier is empty then return **failure**
5. Choose a leaf node and remove from the frontier
6. If node is the goal then return **solution**
7. Expand node, adding its neighbors to the frontier

Note the use of a data structure called the frontier. In theory it could be anything, but in practice we will usually use either a FIFO queue or a LIFO queue.

To generalize this algorithm to graphs, a key consideration is how to avoid loops, which would cause the algorithm to run forever. The answer is to use an explored set data structure which remembers which nodes have already been explored. This could be implemented as simply a list, or perhaps a hash table if searching a list would be too expensive.

The pseudocode then becomes:

1. Graph-Search(problem) returns **solution** or **failure**
2. Frontier initializes via initial state of the problem
3. Explored set initialized to empty set
4. Loop do

5. If the frontier is empty then return **failure**

6. Choose a leaf node and remove from the frontier

7. If node is the goal then return **solution**

8. Add node to explored set.

9. Expand node, adding its neighbors to the frontier, but only if the neighbor is not in frontier and not in explored

One problem with these pseudocode algorithms: we get a big efficiency gain if we check for whether the goal is reached when we expand the node, rather than checking when we pull a node off of the frontier. This minor change will on average cut off an entire level of the search, and the last level is as large as the entire search up to that point.

5 Uninformed search

By uninformed search we mean that the environment is only locally known to the agent. Earlier we called this a partially observable environment, in contrast to a fully observable environment. The agent needs to know its current location, the successor states, and whether or not it has reached the goal. A surprising amount is known about how to solve search problems with such limited information.

5.1 Breadth-first search

Key point: use a FIFO queue as the data structure for the frontier. This way, if N is the current node, its successors will be placed at the back of the queue, so that we finish the current level of the search tree before moving on to the next level.

Advantages: breadth-first search is complete, and is optimal if all actions have the same cost. If all actions don't have the same cost, one might need to extend the search to find the optimal solution.

Disadvantage: space complexity is definitely a problem when searching on a tree, and may or may not be a problem when searching on a graph.

To be more precise, if the branching factor of the tree is b, then at depth d there are b^d nodes at that level, and potentially all need to be stored in the frontier at once. So in this case the space complexity is $O(b^d)$, i.e. exponential. The time complexity is

$$1 + b + b^2 + b^3 + b^4 + \cdots + b^d = O(b^d) \ .$$

To repeat a point made above, this is assuming the solution is at level d, and we are catching it when we are expanding nodes from level $d-1$. If we check for the solution when we pull nodes off the queue, then we are in the process of expanding nodes from level d and storing all the nodes from level $d+1$, so the time and space complexity becomes $O(b^{d+1})$.

5.2 Uniform-cost search

Key point: use a priority queue whose key value is the path cost function, and expand the node that has the least path cost. If using a priority queue, this will be the node on front of the queue.

Advantage: this algorithm is optimal, meaning the first solution found will be the optimal solution.

Disadvantage: can get stuck in an infinite loop if there is a pair of adjacent zero-cost actions.

Implementation notes:

- We cannot use the optimization where the target check is made during expansion, because the first path found to the target might not be the optimal path. For uniform-cost search we must check target when the node comes off the queue, as in the pseudocode above.

- We must differentiate between the explored set and the frontier. For standard BFS one can add to the explored set when a new node is found during expansion, effectively merging the explored set and the frontier. For uniform we need to differentiate: the explored set are the nodes which have been expanded, and thus nodes for which the optimal path has been found. The frontier involves nodes for which a cheaper path may still be found. Again, use the pseudocode above.

- When generating a neighbor, check to see if it is already in the frontier, and if so update it if the new path is cheaper than the old path. This means the priority queue needs to also be searchable and updateable in constant time. I am not aware of a data structure which is both a queue and a hash table; my solution was to keep two data structures and update both.

5.3 Depth-first search

Key point: use a LIFO queue so that it is the deepest node that is expanded first. Alternatively, one can use recursion.

Advantage: better space complexity than breadth-first search. We can delete a node once it has been expanded, so with branching factor b and depth d we have a space complexity of $O(b \cdot d)$ rather than $O(b^d)$.

Disadvantage: not complete, because one could get stuck exploring a path that doesn't contain a solution. Not optimal, without searching the entire space.

One can partially fix the completeness issue by doing depth-limited search, i.e. depth-first search up to depth k. Then it will finish, and will succeed if the solution is at most depth k. Sometimes we know a good bound on k, sometimes we can estimate it, and sometimes we can just try and see if it works.

5.4 Bi-directional search

Another way to do uniformed search is to search from the start and the goal at the same time. Of course, you can't literally do them at the same time, the idea is to trade actions: do an action in the search from the start, then do an action from the search from the goal, and go back and forth until the two searches overlap.

Advantage: in the general case could cut complexity from $O(b^d)$ to $O(b^{d/2})$ which is huge. Disadvantage: it might not be possible to do an uninformed search starting at the goal, and the space requirements are even larger than standard breadth-first search.