CS 520: Notes on (Global) Search

16:198:520
Instructor: Wes Cowan

1 General Search

We are frequently interested in generating solutions to the following problem: given a problem specified by some kind of *state*, and available actions we can take, we want to perform a sequence of actions to produce some specified state or outcome. This is the situation in the wolf/goat/cabbage problem, Rubik's cubes, one player games, etc. In these instances we have a specification of the following:

- State Space: a way of describing the 'state' or information defining an instance of a problem.
- Action/Transition Set: the actions that can be taken, turning one instance of a problem into another.
- Initial State: the initial specification of the problem to be solved (initial state of the Rubik's cube, for instance).
- Goal State(s): state or states that indicate successful completion of the problem.
- Restricted States: states that are not feasible or allowed.

In an automated fashion, we want to generate a sequence of allowed actions capable of transitioning the initial state of the problem to one of the goal states. Search algorithms represent a systematic approach to looking at 'all possible combinations'.

If there are no 'loops' in the search space, no sequences of actions that can revisit states (for instance mixing chemicals that can only move the reaction forward), **Tree Search** suffices.

```
def tree_search(initial_state, goal_states, restricted_states):
    fringe = DataStructure( { initial_state } )
    prev[ null ] = null

    while fringe is not empty:
        current_state = fringe.remove()

        if current_state is a goal_state:
            return "Success!", current_state, prev
        else:
            for each child of current_state:
                if child is not restricted:
                    fringe.add( child )
                    prev[ child ] = current_state

    return "No Solution", null, null
```

If a solution path is discovered, this function returns the goal state reached, and a set of pointers **prev** such that the path from start to goal may be reconstructed. If the algorithm exhausts all children and children of children etc, such that there are no more nodes to explore, then there is no solution. Note - this requires a finite total state space to be guaranteed. If there is an infinite state space, there is no guarantee that tree search will terminate when there is no solution.

If there are loops in the search space, ways of revisiting nodes, then tree search is insufficient - it can get trapped loading and reloading the same states into the fringe over and over again. In this case, we need additional bookkeeping to guarantee that the search algorithm will continue to explore new areas of the search space - a **closed set** that tracks states that have already been expanded and processed.

```
def graph_search(initial_state, goal_states, restricted_states):
    fringe = DataStructure( { initial_state } )
    prev[ null ] = null
    closed_set = {}

    while fringe is not empty:
        current_state = fringe.remove()

        if current_state not in closed_set:
            if current_state is a goal_state:
                return "Success!", current_state, prev
            else:
                for each child of current_state:
                    if child is not restricted:
                        fringe.add( child )
                        prev[ child ] = current_state

            closed_set.add( current_state )
    return "No Solution", null, null
```

In the case of a finite state space, why do tree-search and graph-search guarantee that a solution will be found if one exists?

This whole process can be visualized in some sense by imagining a search tree starting at **initial state** and then growing outwards - the order of growth outwards being determined by the order at which vertices come off the fringe.

It's worth observing here that the search tree - the connections between all the states and their children, is not something that exists in memory. These are frequently constructed dynamically, through the exploration of the actions. They might also be enormous when considered in their entirety. As such we cannot guarantee in advance anything about the structure of the search tree, for instance that it be balanced. Without prior knowledge of the situation, blind exploration is the only solution we have available.

It's worth considering at this point the complexity of these algorithms. The **prev, fringe, and closed_set** structures should be chosen to be as efficient as possible in their operations. The main constraint in terms of the time complexity is the number of states / number of nodes in the search tree that these algorithms visit and pass through the fringe.

This is typically quantified in the following way: let b be the *branching factor*, or maximum number of valid children any state may have. After expanding the initial state, there will be at most b children under consideration. Expanding these, in the second generation, there will be at most b^2 children. Both algorithms, in the worst case (if there is no solution to find) will need to visit every node. If, in the finite state space case, there are at most D generations away from the initial state, then in total these algorithms will process $1 + b + b^2 + b^3 + ... + b^D = O(b^{D+1})$ many nodes. We see that in terms of the time complexity of these algorithms, they have exponential complexity in terms of the branching factor.

The main space constraint is the size of the fringe, or the number of unexplored states stored in memory at any one time. The size of the fringe, and the order that states are explored, depend on precisely how the current_state is chosen and removed from the fringe. This gives rise to the two major algorithms to consider, **Depth First Search** and **Bredth First Search**.

1.1 Depth-First Search

Depth First Search is defined by exploring branches of the search space to their entirety, as deeply as possible, before backtracking to explore other branches. To implement a depth-first search, the only thing required is to take **fringe = Stack({initial_state})**. In this case, the *most recent nodes* put on the stack are popped off and processed first. This has the effect of starting from the initial state, extending a path out essentially as far as it can go, until the search reaches a node on that path that has no children it hasn't seen before. At that point, it essentially **backtracks** to the most recent step in the path with unexplored children, then explores those paths as deeply as it can go, etc. The search concludes when every visited node as no children that are unvisited - thus the full connected component is discovered. This ordered exploration is accomplished entirely via the stack structure.

At this point, we can ask - how big does the stack get at any point in time? What is the space complexity of DFS? At every step away from the initial state, DFS adds at most b children from the current state onto the stack. If the deepest possible path if D steps or generations long, this gives a bound of at max $O(D * b)$ children loaded into the stack at any one time. We see therefore that the space complexity of DFS is linear in terms of the branching factor, and the depth of the search tree.

1.2 Breadth First Search

To implement a Breadth-First Search (BFS), the only thing required is to take **fringe = Queue({initial state})**. In this case, the *oldest nodes currently on the queue* are removed and processed first. This creates a sort of 'level-by-level' effect: at the start, the fringe contains only things at distance 0 from **initial state** (i.e., **initial state** itself). Popping this off and processing it, the fringe will then contain everything at distance 1 from **initial state**. Popping these off (dealing with the oldest nodes first) and keeping track of visited nodes, at some point the fringe will contain everything at distance 2, then everything at distance 3, etc, proceeding until everything in the component is discovered.

The effect of this level-by-level approach is that *the first time a goal state is removed from the fringe represents the shortest possible path to any goal state.* This can be seen inductively or by contradiction - if there is a path from **initial state** to a goal state, there is a *minimal* path, and if there is a minimal path, this level-by-level approach ensures that no longer path can be seen before the minimal path is seen.

As a result of this, the search that results from BFS has the property that every returned solution is *minimal*, connecting every goal to the initial state in a minimal number of steps. It is possible that there are multiple paths of the same minimal length in the original graph - in a case like this, there is no way to guarantee which of these multiple paths BFS will discover (or discover first), but you are guaranteed that the path that it *does* discover will be of minimal length.

1.3 Comparisons

It's worth summarizing some important differences between BFS and DFS:

- DFS is *not complete* if the state space is infinite - a solution is not guaranteed to be found, even if one exists.
- BFS is complete, even if the state space is infinite - if there is a solution, there is a minimal solution, and BFS will find it.
- Both algorithms suffer from exponential time complexity - but this is impossible to avoid as in the worst case, all possible paths and states must be explored to verify there is no solution.
- DFS has a smaller (linear) memory footprint, but that comes at the cost of not guaranteeing how good the returned solution is.
- BFS has a larger (exponential) memory footprint, but comes with the guarantee any solution will be minimal.

1.4 Fun Alternatives

There are many related algorithms and variants, but two deserve mention in particular:

- **ID-DFS, Iterated Deepening DFS** - The linear space complexity of DFS is very attractive, especially for systems with limited resources. Could that be preserved, while at the same time guaranteeing minimality of the solution in the spirit of BFS? If you are willing to increase the time spent (though not by much), the answer is yes. Consider running DFS on the search space, but only to a fixed max depth of size 1. If a goal is found, return it, otherwise repeat DFS increasing the max allowed depth to size 2. Including a max depth limit is simply a matter of some additional bookkeeping. In this way, increasing the max depth by one every time, every path up to a given depth is explored before moving on to allow deeper paths. Which means that for any goal that is discovered, all shorter paths failed to yield goal states, and this goal must be minimal.

  As DFS is being run at every step, this maintains the linear space complexity. However there is a lot of redundancy, revisiting states and re-running paths multiple times. This adds to the overall time complexity - but not by much. Again, suppose that the maximum possible depth was D steps away from the initial state. In the first run of DFS to depth 1, at most $1 + b = O(b)$ nodes will be visited. In the next run, to depth 2, at most $1+b+b^2 = O(b^2)$ nodes will be visited, revisiting some from the initial DFS. In general, going to depth k will visit $O(b^k)$ nodes. In total, running ID-DFS out to depth D will take $O(b) + O(b^2) + O(b^3) + \ldots + O(b^D) = O(b^{D+1})$ node visits. We see that while there is a lot of redundancy it doesn't increase the overall expontial time costs.

- **BD-DFS, Bi-Directional BFS** - In some cases, we can not only specify what child states result from a given state, we may also be able to determine what parents could've given rise to a given child. In these cases the actions are 'reversible'; a good example would be 'backwards' rotations of Rubik's cubes. In these situations, an alternative to BFS preseves minimality, but actually has significant time and space savings.

  If the initial state and goal state are specified uniquely, we can run two BFS searches, one starting at the initial state and working forward, the other starting at the goal state and working backwards, alternating back and forth between them. The search terminates with the discovery of a node that is in both fringes - representing a state reachable from the initial state, that can also lead to the goal state. The minimality of each 'half' BFS solution guarantees the combined solution will also be minimal.

If the path from initial state to goal state is of length D, the point of intersection will be discovered at $D/2$ levels out from the initial state and the goal state. The forwards BFS will have a time complexity of $O(b^{D/2})$ and space complexity of $O(b^{D/2})$ - with similar complexities for the backwards BFS. Which means that the total costs will be $O(b^{D/2}) + O(b^{D/2}) = O(2 * b^{D/2} = O(b^{D/2})$ for space, and similarly $O(b^{D/2})$ for time. While this is still technically exponential in complexity, we've reduced costs over all by a square root - if the initial search would've needed $10,000$ state visits, bi-directional BFS could accomplish it in 200. This represents a considerable savings.

2 Informed Alternatives

As seen previously, the behavior of the search algorithm is determined almost entirely by how the command **current state = fringe.remove()** is executed - how the next state to explore is decided. Removing from one side of the fringe or the other (otherwise blindly) creates BFS, and DFS.

Are there more informed choices that could be made? In the framework provided so far, no. But suppose that we could assign to each state on the fringe a value or a priority, $f(\text{state})$, and then take the fringe to be a **priority queue**, ordering the nodes to explore based on their value of f. Obviously the utility of this depends largely on what f is, but a couple of options immediately present themselves. Note, generalizing away from cost as 'step counts' as in the previous sections, we can include weighted steps or actions between states, with variable costs (such as variable distances between cities, etc). It is convenient to define the following quantities for any state n:

- $g(n)$: this is taken to be the minimal cost to reach n from the initial state. Note, this can be computed in an online fashion such that when n is being processed in a search, the value of $g(n)$ can be known, from the value of g for its parents, and the costs of reaching n from its parents.
- $h(n)$: this is taken to be an *estimate* of the remaining cost to reach a goal node from n. It is necessarily an estimate, since it is based on regions of the search space not yet visited. We will address where these *heuristics* come from separately.

We have the following options for more 'informed' searches.

- **Uniform Cost Search (UFCS)** - In this case, define the priority $f(\text{state}) = g(\text{state})$. UFCS proceeds by exploring the state with current minimal cost, or expanding the current shortest path seen. Proceeding in this way, the first goal state found is guaranteed to have minimal total cost. *Why?* This is a generalization of BFS to the variable weight case.
- **Greedy Best First Search (GBFS)** - Taking the priority to be $f(\text{state}) = h(\text{state})$ prioritizes processing the state next that we *currently believe is closest to the goal*, hence the name 'greedy best first search'. This generalizes the spirit of DFS, where longer paths are explored first in the hopes that we might be closer to the goal. Note, as with DFS, there is no guarantee of minimality, or even completeness. And much will depend on the quality of the estimate h and what guarantees (if any) can be made.
- A^* - In this case, the priority of a given state is taken to be $f(\text{state}) = g(\text{state}) + h(\text{state})$. That is, the priority of any state is *the total estimated cost to travel from start to goal,* ***through*** *the specified state.* With some weak guarantees on the heuristic function h (such as always underestimating the true remaining cost), it can

be shown that not only is A^* an optimal algorithm in terms of returning the path of minimal cost, but it will also have many nice properties such as *exploring the minimal number of nodes possible to discover the solution.*

Note: An important, and mostly unstated, assumption in the previous analysis is that all the costs to move between states are strictly non-negative - that is, traversing between any two states incurs at least some (potentially zero) cost. Why is this important?