CS 520: Notes on Uninformed Search

16:198:520
Instructor: Wes Cowan

1 General Graph Search

As a general problem solving algorithm, we apply search when we can formulate a problem in terms of the following:

- State Space: descriptions of the available information at various points in a problem.
- Action Space: the actions that can be taken to move the problem from one state to another.
- Initial State: the initial state of information at the start of the problem.
- Restricted States: states that are not allowed, based on problem constraints.
- Goal State(s): the state or states that need to be reached for the problem to be solved.

The point of search algorithms is to produce a solution in the form of a sequence of *feasible* actions capable of moving the problem from the initial state to the goal state(s). Search algorithms can be thought of as systematically 'going through all the possible combinations'.

The following 'tree search' algorithm returns whether a path from start to goal was found, and a chain of pointers indicating the path.

```
def TreeSeach(initial state, goal states, restricted states):
    fringe = DataStructure( { initial state } )
    prev[ initial state ] = null

    while fringe is not empty:
        current state = fringe.remove()
        if current state is goal:
            return "Success!", current state, prev
        for each child of current state:
            if child not in restricted states:
                fringe.add( child )
                prev[ child ] = current_state

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

The function returns whether a path was found, the goal state that was discovered (important if there are multiple feasible goal states), and a set of pointers **prev** that allows you to reconstruct the path from start to the goal. Tree search is particularly suited for situations where 'loops' are impossible - you can't take actions to revisit states. If you can revisit in this way, the search algorithm can get caught in loops, perpetually re-loading the same states onto the fringe. In this case, we can augment tree search to 'graph search', by including a 'closed set' of states already visited.

```
def GraphSeach(initial state, goal states, restricted states):
    fringe = DataStructure( { initial state } )
    closed set = {}
    prev[ initial state ] = null

    while fringe is not empty:
        current state = fringe.remove()
        if current state is goal:
            return "Success!", current state, prev
        for each child of current state:
            if child not in restricted states and child not in closed set:
                fringe.add( child )
                prev[ child ] = current state
        closed set.add( current state )

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

This whole process can be visualized in some sense by imagining the search tree starting at **initial state** and then growing outwards - the order of growth outwards being determined by the order at which nodes come off the fringe. Note: if the state space is finite, then if there is a path from **start state** to a goal state, this search algorithm will find one (though not necessarily a specific one). *Why are we guaranteed that this algorithm will find a solution in this case if one exists?*

It is worth pausing to consider here the problem of *complexity*. We can measure the time complexity of these algorithms in terms of the number of states visited to add children to the fringe. In the worst case, this will be **All Possible States**. Can we bound this? This is usually quantified in the following way: let b be the *branching factor* of the search space, the maximum number of children reachable from any state. Expanding the initial state results in at most b children. Expanding each of these children results in at most b^2 children. k steps out will produce at most b^k children in total. If D is the maximum number of steps that can be taken from the start state, this results in $1 + b + b^2 + \ldots + b^{D-1} = O(b^D)$ nodes visited/processed during the course of the algorithm.

At this point, the discussion has been completely general - note, in particular, the use of the generic **DataStructure** type in the pseudo-code above. The particular structure of the search tree, and the properties of the search, are defined almost entirely by the specific choice of data structure, and the resulting order the nodes are processed in.

1.1 Depth-First Search

To implement a depth-first search, the only thing required is to take **fringe = Stack({root})**. 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 root, extending a path out essentially as far as it can go (loading each step of the path into the stack), until the search reaches a node on that path that has no children (or, in the case of graph search, 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.

It is worth noting here that the classic treatment of DFS (particularly as seen in class, and in the book) is typically given in a recursive formulation, exploring, in turn, each of the children of the currently explored node. What this

recursive formulation accomplishes is essentially utilizing the function stack of your hardware as a substitute for the explicitly specified fringe structure.

1.2 Breadth First Search

To implement a breadth-first search (BFS), the only thing required is to take **fringe = Queue({root})**. 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 **root** (i.e., **root** itself). Popping this off and processing it, the fringe will then contain everything at distance 1 from **root**. 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 node is removed from the fringe represents a shortest possible path to that node.* Again this can be seen inductively or by contradiction - if there is a path from **root** to **v**, 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 tree that results from BFS has the property that every resulting path is *minimal*, connecting every node to the root in a minimal number of steps. It is possible that there are multiple paths of the same minimal length - in a case like this, there is no way to guarantee which of these multiple paths BFS will discover, but you are guaranteed that the path that it *does* discover will be of minimal length.

2 Dijkstra's Algorithm or Uniform Cost Search (UFCS)

We can extend the previous discussion, with a bit of extra bookkeeping, to the case that the actions have unequal costs. The classic example might be nodes as cities and action costs/weights as the distance or cost of traveling between them. Another example might be nodes as steps or reactants or products in some kind of chemical process, and actions having the cost of performing certain reactions. In a case like this, the 'minimal' path may not simply have the fewest steps, because those few steps may cost arbitrarily much to traverse, while 'longer' paths have less total cost. In general, we introduce the weight function w such that if the action taking you from node v to node u has cost given by $w(v, u)$. What then is the minimal total cost of traveling from x to y? Enter Dijkstra, or Uniform Cost Search

In general, DFS can be thought of as growing the search tree by picking one direction / branch, growing it as far as possible, then backtracking to the next available branch to grow, and repeating until the whole tree is grown / populated. BFS can be similarly thought of as first growing all the branches of length 1, then all the branches of length 2, then 3, etc, until the whole tree is grown / populated.

Thinking about this idea of using the ordering / data structure to decide which branches of the search tree are worth growing and exploring - if I am starting at **x**, and interested in pursuing the shortest path to **y**, Dijkstra operates on the principle of *extending the shortest path **anywhere** you have seen so far.* In particular, we take the fringe to be a **PriorityQueue**, where when we place a node **v** on the fringe, we assign it priority based on the *total distance from **root** to **v*** discovered so far. Tracking the distance requires a little extra bookkeeping, but leads to a structurally identical algorithm in the following way:

```
def GenerateGraphSearchTree(G, root):
    for v in V:
        dist[ v ] = infinity
        processed[ v ] = false
        prev[ v ] = null

    dist[ root ] = 0
    fringe = PriorityQueue( { (root, dist[ root ] ) } )
    prev[ root ] = root

    while fringe is not empty:
        (v, d) = fringe.remove()
        if v is not processed:
            for each child u of v:
                if d + w(v,u) < dist[ u ]:
                    dist[ u ] = d + w(v,u)
                    fringe.add_or_update_key( u, dist[ u ] )
                    prev[ u ] = v
            processed[ v ] = true

    return prev
```

In this case, at any given point in time, the front of the priority queue represents the shortest path to any as-yet-unprocessed state. We pop that off, extend that shortest path to any reachable children - and if we, in doing so, discover a new shorter path to an unprocessed state (going through **v** to **u**), we update our distance measurement and potentially re-order nodes in the priority queue. The result is again a search tree, as in the previous two cases, but structured specifically so that any recoverable path is minimal total cost from the root.

The 'proof of correctness' of Dijkstra's algorithm essentially boils down to the following point: the first time a node comes off the priority queue represents the discovery of a minimal path to that node. *Why is this true?*

Now priority queues have slightly more overhead than regular queues (Queue Classics, if you will), especially for insertion and potentially reordering if the key needs to be updated. As a result, Dijkstra tends to have a slightly higher computational overhead, but the specifics of this can depend on the specific implementation of the priority queue. But note that, every state will be added to the priority queue at most once (though potentially reordered/reinserted), and every child of every state will be visited (in the worst case) - hence we get roughly the same cost as the previous searches, scaled appropriately for the additional overhead.

Note: An important, and as yet unstated, assumption in the previous analysis is that all the edge weights are strictly non-negative - that is, traversing any edge incurs at least some (potentially zero) cost. Why is this important? What assumptions about the structure of the graph would make Dijkstra's algorithm work in a general case?

3 Notes on the Complexity of Search

In the analysis above, it is argued that the time complexity of BFS and DFS are essentially both exponential in the branching factor and depth of the problem.

Exponential time complexity is not great, but in general it *cannot be avoided* in the worst case, because the size of the search space will always be exponential in terms of the branching factor and distance, *and the goal may very well be in the last place you look, if it can be reached at all.*

However, an interesting distinction arises when we look at the space complexity. The space costs of BFS and DFS are essentially the maximum number of states the fringe must be storing at any given point in time. In the case of BFS, the fringe must store entire levels at a go, so the worst case, the fringe will balloon to holding something on the order of $O(b^D)$ vertices. When running DFS however, every time you process or expand a vertices' children, you are adding at most b vertices to the fringe. This occurs potentially every step along the longest path DFS explores, i.e., DFS incurs a memory cost of at worst $O(D * b)$ - linear in terms of the length of the path.

This is a tremendous savings in terms of implementation - DFS is very frequently the search algorithm of choice on account of the much smaller memory footprint. The cost of this is of course the fact that optimality is potentially sacrificed, that the path returned may not be the shortest possible path.

One last note of interest: in some cases, the goal state may be explicitly identified (for instance, if you want to travel from **LA** to **NYC**). In other cases, the goal state may be less explicit (in a game of chess, you know when you have reached a state of checkmate, but you do not necessarily know in advance *which* state of checkmate you will or want to reach).

In the case that the goal state is well defined, we could consider running a *reverse* breadth-first search backwards from the goal to try to discover a path to the start state. In fact, we could consider running this and the original BFS in parallel with each other. One fringe proceeds level by level through the search space starting at s, the other fringe proceeds level by level through the reverse-search-space starting from g. The search terminates when the fringes intersect. One could imagine that if you are trying to find a path from LA to NYC, if you start in LA and a partner starts in NYC and you both meet in Denver, each of you has half the total path necessary, and you can combine what you've found to yield the whole path from start to finish. Similar to BFS, it can be shown that the resulting path (with uniform action costs) will be minimal.

But what is the complexity of this search? If there are D steps from start to goal, a regular BFS would take roughly b^D search steps. However, if two searches proceed from both directions and meet in the middle, each will have only taken rough $D/2$ steps, or $b^{D/2}$ search steps, for a total time complexity (and space complexity) of roughly $2b^{D/2}$, or $O\left(b^{D/2}\right)$. In terms of overall cost, this is still exponential, but we have essentially taken the square root of the complexity, a dramatic speedup. Instead of visiting 10,000 state for instance, you might be looking only at 200 states in total. Bidirectional BFS can lead to a lot of space and time improvements for problems of reasonable size, while maintaining the optimality properties of BFS. For especially large problems, however, the space and time explosion is still considerable.