CS 512: Comments on Graph Search

16:198:512
Instructor: Wes Cowan

1 General Graph Search

In general terms, the generic graph search algorithm looks like the following:

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

    fringe = DataStructure( { root } )
    prev[ root ] = root

    while fringe is not empty:
        v = fringe.remove()
        if v is not processed:
            for each child u of v:
                if u is not processed and prev[ u ] = null:
                    fringe.add( u )
                    prev[ u ] = v
            processed[ v ] = true

    return prev
```

This function returns a set of pointers **prev** such that if I want to find the path from **root** to some vertex **v**, I simply get the pointer **prev**[**v**], then track the previous pointer to that, and the previous pointer to that, until I reach a point where **prev** return **root**, and this reconstructs the path from **root** to **v**. In this way, the path from **root** to any vertex may be reconstructed; the whole set of pointers may be thought of as defining the search tree, a directed acyclic graph from **root** to every other vertex. This whole process can be visualized in some sense by imagining the search tree starting at **root** and then growing outwards - the order of growth outwards being determined by the order at which vertices come off the fringe. Note: if there is a path from **root** to a given vertex **v**, this search algorithm will find one (though not necessarily a specific one). *Why are we guaranteed that this algorithm will discover all vertices connected to* ***root****? Think by induction, or by contradiction.*

One thing to note at this point: what is the complexity of constructing the search tree? Because of the bookkeeping of the **prev** and **processed** structures, no vertex enters the fringe more than once. Note further, the loop does not exit until every vertex in the fringe has been processed and left. This yields, in total, a maximum of $|V|$ add and remove operations. Further, for each vertex, we loop over the children of that vertex - the total number of children looped over is precisely the total number of (directed) edges. There is, at worst, a constant amount of work done for each of these children, yielding a total search complexity of

$$O\left(|V| + |E|\right).$$

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 vertex 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.

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.

Additionally, when studying DFS, significant attention was paid to the notion of pre/post numbering, and the structure that these numbers indicates in the underlying graph. Where would pre/post numbering be done in this non-recursive formulation?

1.2 Breadth First Search

To implement a depth-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 vertex is removed from the fringe represents the shortest possible path to that vertex.* 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 vertex to the root 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, but you are guaranteed that the path that it *does* discover will be of minimal length.

2 Dijkstra's Algorithm

We can extend the previous discussion, with a bit of extra bookkeeping, to the case that the weights have unequal costs. The classic example might be vertices as cities and edge weights as the distance or cost of traveling between them. Another example might be vertices as steps or reactants or products in some kind of chemical process, and

edge weights as 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 (v, u) is an edge in G, $w(v, u)$ is the cost of traversing that edge. What then is the minimal total cost of traveling from x to y through G? Enter Dijkstra.

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 vertex **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 vertex. We pop that off, extend that shortest path to any reachable vertices - and if we, in doing so, discover a new shorter path to an unprocessed vertex (going through **v** to **u**), we update our distance measurement and potentially re-order vertices 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 vertex comes off the priority queue represents the discovery of a minimal path to that vertex. *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 again, every vertex will be added to the priority queue at most once (though potentially reordered/reinserted), and every child of every vertex will be visited - 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 complexity of BFS and DFS are essentially both linear in the size of the problem. This is both true, and potentially misleading - complexity often depends on the precise way that you assess the problem you are operating on.

Consider characterizing a graph in the following way: the **branching factor** b of a graph is the maximal number of children any vertex has. Given a start vertex s and a goal vertex g, let's suppose that the shortest path between them is d steps, and the longest path at all in the graph is at worst m steps. Characterizing the graph in this way, there are *at worst* b vertices at distance 1 from s, b^2 vertices at distance 2 from s, etc, b^k vertices at distance k from s. Note, the *total number of vertices* at distance at *most* k from s is $O(b^{k+1})$ - *why?*

In the case of both BFS and DFS, the amount of time the algorithm takes to run is essentially the number of vertices it needs to visit before it discovers the target. In the **worst** case, in both search algorithms, the goal is the **last** vertex searched / last vertex to come off the fringe. In this case, every vertex in the graph must be visited at least once - this gives a time complexity that is **exponential** in the path length, $O(b^{d+1})$ for BFS and $O(b^{m+1})$ for DFS.

Exponential time complexity is not great, but in general it *cannot be avoided* in the worst case, because the size of the graph 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.*

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 vertices 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 $O(mb)$ - 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 vertex may be explicitly identified (for instance, if you want to travel from **LA** to **NYC**). In other cases, the goal vertex 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 vertex 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 vertex. In fact, we could consider running this and the original BFS in parallel with each other. One fringe proceeds level by level through G starting at s, the other fringe proceeds

level by level through G_R 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.

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 vertices for instance, you might be looking only at 200 vertices 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.