Dijkstra's Algorithm

Dijkstra's Algorithm

- We saw that breadth first search let us find shortest paths in unweighted graphs
- BFS stored nodes that needed to be explored in a queue, which explored them in “fewest edges to the start” order
- For weighted graphs, a node that is 3 edges away from the start might have a lower weight path to it than a neighbor of start
- We need a data structure that lets us get the “lowest cost to start” of all explorable nodes quickly
- A heap based priority queue works, if we add one more method

Dijkstra's algorithm:

- This is the same template that we saw for other search algorithms!

```
PQ = priority queue keyed on "length of shortest path to start"
PQ.add(start)
start.visited = true

while !PQ.empty:
    n = PQ.removeMin()
    if n == end: reconstruct path and return

    for unvisited neighbor of n:
        add neighbor to the PQ
        neighbor.visited = true
```

Uh oh!

- Consider a graph search starting at node A
- A has edges of length 4 to B, and length 1 to C, and 3 to D
- C has an edge of length 1 to B
- When we start, we add B,C,D to the queue
- C is visited first since it's “closest”
- But, we can actually tell that B is distance 2 from A if we go through C! But we don't re-add it to the PQ since we marked it as visited!

New operation needed!

- When we're looking at a node n's neighbors we may find nodes which are marked as visited that we can reach more quickly through n than any previous path
- We need to update the priority queue to increase their priority
- This is called an “decrease key” operation
- In a binary heap, we can do this by calling percolateUp() starting from the entry we care about
- But we need to know which index that value is at...

Solution: Extra Data structure

- In addition to the array which stores the binary heap, we also store a map which tells us where to find each node in the heap array
- Any swaps we perform must update this map in addition to the array
- Using a hashmap, all map lookups are expected to be O(1), so adding that data structure doesn't change our runtime in terms of big O
- Percolate up is worst case O(lg N) if we were to decrease a leaf node's priority number to be the minimum
- The average case will probably be much less for the same reason as heap's add method

Bonus AI Algorithm

- When I took an AI class it was ~50% “graph search” since so many “AI” scenarios can be represented via graphs
- The phrase has now lost all useful meaning, but we'll still look at a “classical AI algorithm” here!

Problems with BFS + Dijkstra's algorithm

- Both BFS and Dijkstra's algorithm explore nodes closes to the source first, and ripple out from there
- Neither of them takes the target into account when choosing which neighbor to explore next
- We can modify Dijkstra's algorithm slightly to take the target into account for some scenarios and we end up with a significantly improved algorithm!
- The algorith is called `A*` pronounced “A star” where the star means “optimal” ... you can't do better than it!

Heuristic Functions

- A* requires extra data compared to the algorithms we've seen so far, called a “heuristic function”
- h(n) is an estimate of the distance from n to the target
- A* requires that h be an **under*estimate of the true distance, which is actually pretty convenient for a lot of problems
- As an example, if we have a graph representing streets and addresses, the a reasonable heuristic for how far something is from the target is the “crow flies distance” between them
- This heuristic ignores that we have to drive on streets to get between them, so it will be an underestimate (driving will always be at LEAST as far ar our estimate), and it's easy/cheap to compute

A*

- A* is the same as dijkstra's algorithm except that we change the comparator we use for our priority queue
- We choose to explore the node with the “lowest cost estimated path” to the target
- In other words, we order our frontier by “f(n) + h(n)" where f(n) is the length of the path from the source to n, which we know exactly, and h(n) is the estimated length of the rest of the path
- This combines information which we know with certainty (how far from s to n) with an estimate of what we don't know
- Compared to BFS, A* looks more like a teardrop where we expand out in all directions, but “steer” towards the target

Properties of A*

- A* is an “optimal” algorithm, which means a couple of things in this context
- First, it's guaranteed to find the lowest cost path from source to target (ie, it's correct)
- Second, if you tried to come up with a better algorithm, it would have to look at all the nodes that A* does in order to guarantee that it finds the shortest path. So, no matter how clever you get, you can't really do “better”
- So, if you're able to come up with a reasonable heuristic, you should defintitely use A* !!