Trailblazer Notes

Dawson Zhou

Trailblazer can be split up into three distinct parts:

1) shortest path search with Dijkstra's algorithm
2) shortest path search with A* search
3) maze creation with Kruskal's algorithm

While the first two parts are very closely related to one another, the third part is completely separate, and it makes sense to focus on that only once you have the first two parts working.

Dijkstra's Algorithm

The pseudo-code provided in the assignment handout is really your friend here. Make sure you really understand what each step is doing. Look at the graph in the slides from Lecture 25 (starting at slide 66), and trace through the pseudo-code using that specific example. This will really help you solidify your understanding of what's going on, as well as the data structures needed to make the algorithm work.

Dijkstra's algorithm is all about making our best guesses that we can about the shortest paths to various cells in the world, starting with the beginning cell. Along the way, we need to track certain information for each cell: have we visited this cell yet? Do we know for certain if we've found the shortest path to it? If not, what's our best guess for the shortest path here?

In order to answer those questions, we'll need to store information that's associated with each cell. As it turns out, each cell can also be described by its location (i.e. row and column). You may be tempted to change the `Loc` struct by adding data members – *you should avoid changing Loc*. Stylistically, it doesn't make sense for a location structure to be packed with other context-specific information. Instead, if you want to package all of that data together, you should define your own struct. For example, let's define a `Node`:

```
struct Node {
    // data associated with a cell that matters to Dijkstra
};
```

If you're reading the graphs chapter in the course reader, take note: you probably don't want to use the `Graph` type provided by our libraries. Instantiating a `Graph` object would require defining `Node` and `Arc` types that are dynamically allocated connected to each other via pointers, and so on. It ends up creating an over-complicated solution. In Trailblazer, the graph is stored *implicitly*. If you're looking at the node located at (2, 3), you don't need to need to explicitly store which nodes are its neighbors. You know that they're the eight nodes located at (1, 2), (1, 3), (1, 4), (2, 2), ... etc. Look to your Boggle code for this part!

Since you'll be dealing with (row, col) pairs a lot, if you decide to define an additional structure to hold information associated with cells, you'll need a data structure to store all of that information. Ideally, you want to be able to look up that information given the row and column. This lends itself to either a map or a grid – think about which might make more sense, or be easier to code up.

An important thing to make sure you understand is how the priority queue plays its role. At any point in time, the queue stores some number of cell locations (if you've been told that you should store vectors of locations in the priority queue, you have been misinformed!). The cells currently inside the queue represent ones that you've seen during your outward exploration from the starting point. Let's suppose that one of those locations is called X. Since you've visited X, you have a guess for the shortest path from the start to X. However, you're not certain yet that it's actually the shortest path, because there might be another way to reach X which is even shorter.

The only time you know that you've found the shortest path to a location is when you pull it out of the priority queue. At that point, you know for a fact that there's no shorter path to X; if there were, you would have seen it earlier. That property is given to us by the fact that the priority queue always returns the element with the lowest weight. In this case, we're using path costs as our weight, so the shortest paths come out first.

- When we enqueue a location into the priority queue, we mark it **yellow**. We've visited it, but we don't know if we've found the best path there quite yet.
- When we dequeue a location from the priority queue, we mark it **green**. It's the cheapest path in the queue, so we know we've found the best path there.

When we're exploring new nodes, we have to consider a few different possible scenarios. Suppose we just pulled location X from the priority queue. That means we know the cheapest path from start to X. Now, consider X's neighbor Y. We know that Y is either:

1) **gray**, which means we've never visited Y before. In that case, we now have a path from start to Y: the path from start to X, followed by a hop from X to Y.
2) **yellow**, which means we've already seen Y and made a guess about the best path from start to Y. In that case, we want to see if we can beat that best path by going through X instead.
3) **green**, which means we've already found the shortest path to Y, and there's not going to be a shorter path. In that case, we can do absolutely nothing.

This means that if Y is gray or yellow, we should calculate the cost of going from start to Y by going first from start to X, then hopping from X to Y. Our nifty Node structure from before should be able to remember the cost of going from start to X, and our cost function will tell us the cost of hopping from X to Y. If we sum these together, we have the cost of routing our path to Y through X.

If Y was gray, then this path is our best bet, because we haven't found any other paths to Y yet. If Y was yellow, then we already had a best guess for the shortest path from start to Y. In that case, we should compare our path which routes through X and see if it's even shorter. If it is, we need to update our best guess to involve X, and adjust the weight of the Y location in the priority queue accordingly.

At some point, if we dequeue the end node and color it green, it means we've found the absolute shortest path from start to end. Then, we can quit searching and return an answer. The expected answer should be a path represented by a Vector of locations. In order to actually form that Vector using the algorithm just described, we'll need to store a bit more additional information for each node.

As of now, each node remembers the total cost of the shortest path from start to that node. However, to reconstruct the path itself, we need each node to know which location preceded it in the path. Luckily, that's just something else we can store for each node! Suppose we just came from location X and we're looking at its neighbor, location Y. Suppose further that we discover that the path from start to Y via X is the shortest path we've ever found to Y. In that case, we would update Y's node to remember the cost of said shortest path. On top of that, we should also indicate that to get to Y along that path, we should find our way to X and hop from X to Y.

X in turn remembers whichever location precedes it in the shortest path from start to X, and so on. By following these links back until we reach the start location, we can reconstruct the shortest path from start to Y.

As a hint, we've used the term "parent pointers" to refer to the preceding cell. However, you don't need to store them as actual pointers; every node just needs to remember the (row, col) location of the preceding node. Your life will be a lot easier if you avoid pointer manipulation here.

A* Search

Although A* search provides a much smarter searching algorithm, it barely requires any change at all to Dijkstra's algorithm. The idea with A* is that we can try exploring paths that are roughly in the general direction of the end location. The way we specify that is by changing the order of the cells dequeued from the priority queue, since we continue exploring from those cells.

You'll notice in the A* pseudo-code that the only place in which it differs from Dijkstra's algorithm is when you're adding something to the priority queue or when you're adjusting the weight of something already inside the priority queue. In both of these cases, instead of making the weight of a path its total cost, we set it to be the total cost plus an estimate of how much longer it might take to reach the end (given by the heuristic function).

In summary: make sure the heuristic is only used to calculate priority queue weights.

Kruskal's Algorithm

Suppose we have an open world with no walls. From every cell, we can visit each of its four neighbors in the north, south, east, and west directions. Technically, this is a maze. A really bad one, but a maze nonetheless. We can construct a more interesting maze by introducing walls in between adjacent cells, making it impossible to travel directly between the two cells.

Kruskal's algorithm works by treating each cell in the world as a node in a graph and including as few edges as possible while still ensuring that every node can reach every other node. The graph produced, sometimes referred to as a *minimal spanning tree*, can be converted into a maze using the above description. An edge allows traveling between the two nodes directly, representing open space in the maze. If an edge isn't present, the maze has a wall there which prevents crossing from one cell to the other.

To generate the minimal spanning tree, we start with every location sitting in isolation (i.e. no edges). Then, we consider every edge in some random order. Every time we see an edge, we can ask if the two nodes on either side are already connected (potentially indirectly). If they are, then we should ignore this edge, since it's redundant. If they aren't, then the edge should be used. From that point on, those two nodes should be considered connected.

If we repeat this process until every node is connected, then we have a graph where any node can reach any other node, but not a single edge is redundant. That represents a maze, and we're done.

The trickiest thing about Kruskal's algorithm is representing connectivity between nodes. We can think of groups of nodes which are connected to each other as *clusters*. Suppose X belongs to the cluster consisting of X, Y, and Z, and B belongs to the cluster consisting of A, B, and C. Consider an edge which could connect X and B together. Since X and B are not in the same cluster, the edge is not redundant and therefore should be included. By including it, we decide to merge those clusters into one. Now, X is connected to Y and Z as before, but also A, B, and C. On top of that, even though Z was not directly involved in the selection of that particular edge, Z should now know that it's connected to A, B, and C, since it can now reach them via the edge between X and B.

It's up to you how you represent clusters, but keep in mind what you want to do with them:

- You should be able to find the cluster that X is associated with. Likewise for B.
- You should be able to ask if X's cluster and B's cluster are the same or different.
- You should be able to merge the two clusters so that every node in the larger combined cluster knows that it is part of a new family.

Good luck on the assignment! As always, feel free to email Keith or myself with any questions.