Algorithms & Analysis Dijkstra Report - s3723825

My Implementation of Dijkstra's Algorithm is stock standard, the pseudocode I followed was easily translatable into Java. I originally tried to implement the algorithm using a Priority Queue but found that accessing elements when creating paths was harder to wrap my head around. Using a HashMap I was easily able to model the a(a,0) structure seen in Dijkstra's Algorithm tables and found it easier to access. To further aid my implementation I used a custom type called a CoordDist, which is the (a,0) part of the Structure, it hold a previous Coordinate in the shortest path to its Coordinate which can be used to loop though the HashMap until the distance is 0. I had one helper method to return a coordinates list of neighbours, there was a tossup of efficiency here as I didn't know if it would be more efficient to loop through unvisited Coordinates and check if those coordinates were adjacent or create adjacent Coordinate and use contains to see if they are in the list. From what I know contains loops through the array, and I would have to use contains 4 times, so I opted out of this strategy. My strategy can be further optimised by doing some additional checks if 4 coordinates have been found (3 on edges and 2 in corners), my strategy also gets more efficient as coordinates are removed. Task A was very easy to implement.

I'm not sure if Task B was supposed to be as easy as it was, I switched the number 1 to Coordinate.getTerrainCost(), that was the extent of my thinking here, since I never switched to the node representation as there are a lot of examples of matrices used with Dijkstra on the internet to find notes on when I got lost.

Task C was also easy, using two for loops (nested destCells in originCells) I added each possible path to an array and then just looped over the arrays to find the shortest distance. This implementation can still be seen in my code but it also has the code for task D inside its loops.

Task D was a little harder, I originally thought that chaining dijkstra's algorithm would work (this is ultimately the solution) but I ran into road block when I was told that waypoints may not be received in order, my mind quickly went to trying to come up with a way to generate the different orders that waypoints can be travelled to in. To do this I used Heaps Algorithm which was simple to implement (just took me awhile to find notes on) waypoints add O(w!) efficiency to the program so I tried to make efficiency changes anywhere I could. One of those efficiencies was only generating the shortest paths once for each origin cell and waypoint which I wasn't originally doing (O (1) access with hash maps trumps using Dijkstra again). Now I had every possible path that could be taken (built with waypoints, origins and destinations) I could add their paths to the paths list and get their distances. As a quick aside, for the waypoint task I think Dijkstra is probably the most efficient way (that I have learnt) to implement this problem

In the code possibly the biggest optimisation that could be made is to remove the paths list (the only reason its still here is so I have something else to talk about) instead of saving all the paths and finding the shortest one I could just save one path and replace it when it longer than the new path being created, this is more of a space issue than a time issue (since all the comparisons would still happen). However in an actual program I feel like it would still be important to keep all the shortest paths for every combination especially in emergency dispatch and similar programs.

I also didn't touch any of the other code because I didn't se the point, everything is easily done in DijkstraPathFInder (that is unless I missed something).