Part 6

Assuming we will be processing neighbouring towns in order of their index (ascending order where the 0^{th} index will always be processed first) and the adjacency matrix will be in a form of an array.

For the DFS, I will still use a stack and a 'visited' indicator to go through all the vertices; the difference will be how I am traversing through the graph. Given a source_id, I will print it, put it in the stack and mark it as visited. Then I will loop through the adjacency matrix for that particular source_id starting from the start of the array to the last. In the loop I will be checking if the vertex I am looking at is a neighbour, if that is true then I will check if I have visited that vertex. If that vertex is a neighbour and has not been visited, then I will print it, add it to the back of the stack and mark it as visited. The recently added element in the stack will be processed right after being added. If there are no more unvisited neighbour for the current vertex in the last stack, pop the element out of the stack and process the next element (element added just before the popped element).

For the BFS, I will still use a queue and a 'visited' indicator. It will be similar to how I will tackle the DFS with the difference being when I find an unvisited neighbour. After finding the unvisited neighbour, I will add it to the back of the queue but I will still process the current vertex until all of its neighbours are visited; then I process the next element in the queue (most recently added element in the queue after the prior element).

Part 7

I used the same approach for part 3 as I did for part 1 (using a stack). The difference being that I will record the cumulative distance and the traversing will be stopped once the destination_id has been found.

For part 4, I used a recursive function that works much like a stack. It will do a depth-first search algorithm to traverse through the graph (same as part 1). When the destination is reached, we print the simple detailed path and move back to where we just were and search for alternate routes. Once all edges of the current vertex is exhausted (meaning all possible routes from that point onwards have been traversed), we move back and process the city prior. We repeat that until we return to the initial city.

For part 5, I used the same approach as part 4, although I kept track of more variables such as the current path's distance and the distance from the city prior. Once the destination is reached, I compare its distance to any other successful paths and store the path with the shortest distance to my "path" structure. After all the paths have been traversed, print out the shortest path and its distance.