Part 6: Graph Representation with Adjacency Matrix

If the adjacency list was replaced by an unweighted adjacency matrix for parts 1 & 2 then it's safe to assume the matrix would be filled with '1's to indicate an edge and '0's to indicate no edges to the other vertex. Neither of my DFS and BFS algorithms would change very much if it was given an adjacency matrix instead as they would still need a 'while' loop to iterate through all the nodes while the respective explicit stack and queue are not empty. Perhaps the only chance required would be the style the edges are accessed within the first 'while' loop. Instead of using another 'while' loop to look at each edge extending off each vertex structure linked with pointers. We can simply index into the matrix using the index of the value of the item at the top of the stack for DFS (part 1) and the front of the queue for BFS (part 2). After obtaining information on the vertex we are on, we can use a 'for' loop to iterate through the row of our (current) vertex to find all the edges. At each edge, we perform similar operations as compared to the adjacency list. Pushing the first unvisited vertex's index onto the stack and then 'breaking' the loop for DFS and enqueueing any unvisited vertices in the row of the matrix and then printing them for BFS. As we are using a 'for' loop, there is no need to signal to the loop to go to the next edge using pointers, it will just iterate through until it either gets stopped at a 'break' or reaches the $(v-1)^{th}$ vertex, where 'v' is the number of vertices. This makes it different from the adjacency list and easier.

Part 7:

I implemented the almost the exact same DFS algorithm of part 1 for part 3. I created an explicit stack, a visit list size of 'v' (number of vertices) and total distance integer. I defaulted all the values of the visited list to false (0), pushed the source vertex index on the stack before entering a 'while' loop to iterate through the vertices while the explicit stack is not empty. For every iteration, I pop the stack and check to see if it has been visited. If it has, I do nothing, if it hasn't, I mark it as visited and proceed to print its name and total distance to that vertex so far. I will then proceed to look at its edges using another while loop guarded by the fact there is still an edge to look at since it is an adjacency list. If the vertex the edge leads to has not been visited by me yet I will push its index onto the stack, add the edge's weight to the total distance since we've just traversed it and break the inner 'while' loop to avoid an infinite loop. If the edge leads to a vertex we have already visited, we'll just keep looking at the next ones until we find an unvisited one or run of edges on this vertex. Once this inner 'while' loop terminates we go back to looking at what's on the top of the stack until every vertex has been visited and the stack has been backtracked all the way and popped empty because of it.

Part 4 works in a similar way as it also uses DFS. Instead of a 'while' loop, I call upon a recursive function to go into the depths of each unvisited vertices after doing the same things I've done for part 3 on before the 'while' loop. However, I created a partner stack passed into the recursive function to keep track of each unique simple path to tell the print function what to print. After entering the recursive function by passing the source id too into it, I append to the stack from the back (to make sure it prints out in the right direction), mark the current vertex as visited and start looking at its edges like part 3. Instead of adding to the explicit stack like in part 3, I call the recursive function again. This would send the stack in to be modified, if the recursion was not started again in the child functions, the function would either look at the next edges or backtrack and mark the current vertex as unvisited and reassign a new current vertex from the top of the stack to pave way for the next possible path. At any given point in the recursion into the children functions, if the current vertex is the destination node, the stack is immediately passed into the print function to print we skip looking at the edges to the popping the stack part (changing paths).

Part 5 takes the part 4 and adds a distance factor into it. I included another function 'compare' and an array to store the indexes of the shortest path along with 2 distance variables, one for the shortest path and one for the current path. The shortest path will be initialised to INTMAX at the start to make sure the first path gets assigned as the shortest. The recursive function would then call the compare function to check if the current path is the shorter one using brute force and assign the indexes into the shortest path array if it is. The recursive function assigns a pointer to the traversed edge labelling it as 'previous' before traversing it. This would allow for the total distance for the current path to be deducted correctly when the child function returns '1' through 'backtracking'. This prevents any unnecessary deductions in distance from void returns if function did not 'backtrack'. As this is using the brute force method, the shortest path and its distance would only be confirmed once all the paths have been checked and the recursive function ends. The print function then prints the path and its distance.

Bonus Question: Complexity Analysis

Let v be number the vertices and e be the number of edges, e = v^2 when all vertices are connected to one another

Part 3: O(v+e)

First while loop looks at each vertex basically since each one of them must have been visited at least once, inner loop looks at each edge of each individual vertex, making it v plus e.

Part 4: O(v!e)

The longest possible path a path can be is to travel through all the edges except one to get to its neighbour vertex, which would bring the worst case of ONE path to be (e-1), which is just e. There are v factorial number of ways to get from one point to another. V factorial number of ways with each path performing at e at worst. Hence complexity of v factorial times e.

Part 5: O((v!)^2 e)

V factorial squared times E

The compare function gets called v factorial minus one times as every other path after the first gets compared. Since the compare function is called whilst paths of v! are still being produced, it is essentially searching each path the number of times equivalent to the number of existing paths. This produces a complexity of v factorial times e times another v factorial.