Part 6: Graph Representation

Altering the representation of the graph from an adjacency list to an adjacency matrix would force the manner in which the edges are visited and the vertices are traversed to be changed.

As my implementation of depth-first traversal does not use recursion, instead using an explicit stack, the way the adjacent vertices are reached changes from iterating over each edge in the singly-linked list that is pointed to by the current vertex, to iterating through the 2D array representation of the adjacency matrix. This means that there would have to be a for loop instead of the while loop that currently iterates over connected edges to the current vertex. This for loop would iterate either over the columns or the rows of the adjacency matrix until an edge is reached, storing the vertices and marking them as visited whenever the value indexed in the adjacency matrix indicates an edge between the two vertices. The iteration over the matrix alternates between column and row after each edge is found, until the number of visited vertices is equal to the total number of vertices in the graph, as we may assume the graph is connected (the same as the current implementation).
Unlike in depth-first, breadth-first traversal would not alternate the direction of its iteration, instead iterating over an entire row or column before iterating over all the edges. Similar to the changes to depth-first traversal, the while loop used to iterate over each adjacent vertex to the current vertex would have to be changed to a for loop that iterates over each index, dequeuing visited vertices until the queue is empty, which is when the program terminates.

Part 7: Design of Algorithms

Parts 3, 4 and 5 all make use of modified versions of a basic recursive helper function used to find all possible simple paths between a source and destination vertex. This is because part 3 requires any simple path, which is not possible by using the DFS algorithm I used in part 1 as backtracking becomes an issue with calculating unique paths and their cumulative distances, part 4 iterates over each possible simple path along with print statements for each not yet discovered vertex visited in a path and part 5 uses the algorithm for part 4 whilst storing current minimum total path distances and paths to find the shortest possible path by distance. All three paths make use of an array representation of a stack, unlike parts 1 and 2, which use singly-linked lists to represent a stack and queue respectively.

In part 3, for every recursive call, the current vertex is pushed to the stack. The recursive helper function takes a start and end vertex while updating the start vertex for every call until the start and end vertices are the same, which is when the elements of the stack are printed as the first unique path from the source to the destination vertex. When the start vertex is not equal to the end vertex in a function call, the adjacent vertices are iterated over by continually changing the current edge to the next pointed to by the current vertex until an undiscovered vertex is found, which then becomes the current vertex. This continues until the last vertex of the first path is found, at which point the is_first flag is set to false and the recursive function no longer prints the paths.
For part 4, a slightly simpler version of the part 3 recursive helper function is used, as there is no need for checking if the current path is the first path. Using the same algorithm for traversing the graph, part 4 prints each vertex of each unique path. Part 5, again, uses the same algorithm to determine each possible simple path, however for every path it finds, the current total distance, found by cumulatively adding the weights of the edges traversed, of the path is compared to the current minimum distance, and the current minimum is set to the current distance should it be lower. Additionally, if the current path has a lower total distance, the current path is stored in an array, which is used in shortest_path, after the recursive helper function has finished all of its calls, to print the path of minimum length.

Bonus Mark: Complexity Analysis

Part 3:

Despite only needing the first permutation of the unique vertices for its output, part 3 makes use of the algorithm used in part 4 that determines every possible permutation of the n vertices in the graph, therefore resulting in n! calls of the recursive helper function. This means that determining a simple path between an input source and destination vertex, provided the two vertices are different, would be O(n!).

Part 4:

As with part 3, the same basic algorithm calculates every simple path using an adjusted recursive depth-first traversal. Additionally, the only other operation in this part would be printing each vertex in the current path which would be O(1), meaning the algorithm as a whole should have the same worst case of O(n!).

Part 5:

As with the two previous parts, the underlying algorithm that calculates every possible simple path is used here, and the additional operations of appending distances to an array and updating minimum total distances would be O(1), meaning the algorithm for part 5 would also have a worst case of O(n!).