Part 6: Graph Representation

For depth-first traversal with an adjacency list, connected vertices from the current vertex are pushed to the stack. This is done using iteration that steps through the linked list with the line `curredge = curredge->next_edge` until a NULL curredge is reached. An element is then popped from the stack, and if it is unvisited, it is set to visited and printed. This process is repeated by iteration until the stack is empty and thus all vertices have been visited.

If an adjacency matrix representation was provided, the only difference would be in the method of adding the connected vertices from the current vertex to the stack. If for example, a 2D array was provided, we would iterate through the row for the current vertex (note that id is also the index for vertex's row). For any '1' value found in this row (i.e. there is an edge), the corresponding connected vertex (id = index of column) is pushed to the stack.

For breadth-first traversal with an adjacency list, if the current vertex is not visited, it is set to visited and printed. Then, all connected vertices from the current vertex are checked if they are visited. This is done by iterating through the linked list of edges from the current vertex until a NULL edge is reached. If they are unvisited, then they are set as visited, enqueued and printed. An id is dequeued and the breadth search is then conducted on the new id. This process repeats until the queue is empty and thus all vertices have been visited.

If an adjacency matrix representation was provided, such as a 2D array, changes like those made to the depth-first traversal would need to be made. Rather than iterating through the linked list of edges from the current vertex, we would iterate through the row for the current vertex in the 2D array. Again, for any '1' value found in this row, the corresponding vertex would be enqueued, set as visited and printed.

Part 7: Design of Algorithms

Part 3: Finding a detailed path | To find a detailed path, depth-first traversal is used. In general, if the vertex is unvisited, it is set to visited, then the distance of the edge (0 for first vertex) is added to the total distance, and the vertex is printed along with the total distance (the cumulative distance). If the destination is reached, the loop is broken, otherwise, iteration paired with the line `curredge = curredge->next_edge` is used to push all connected vertices first to a temporary stack, and then to the main stack so that they are in the correct order. If the vertex is visited however, we must set `curredge = curredge->next_edge` so that the weight of the curredge matches the weight of the edge leading to the id to be popped next. An id is popped from the stack and the traversal continues using this new id as the current vertex. This process repeats until the destination is reached, or the stack is empty, however as the graph provided is fully connected, the stack should not become empty since the destination should be found before this occurs.

Part 4: Finding all paths | To find all paths, depth-first traversal is used. If the vertex is unvisited, it is set to visited and pushed to the stack. If the destination is reached, the stack is printed and the destination is popped from the stack. Otherwise, if the destination isn't reached, the depth-first search is recursively called passing the stack, visited array and current vertex. After the recursive call is complete, the current vertex is set to unvisited. This is so that further paths including this vertex can be found. The next edge from the current vertex is then found. If there are no more connected vertices to the current vertex, the current vertex is popped from the stack and the while loop is broken. If this is a recursive call, the recursive call ends, otherwise, it is the end of the depth-first search. If, however, there is another connected vertex, then the whole process of checking if it is visited etc. is repeated until the stack is empty or the loop is broken in the non-recursive call.

Part 5: Finding the shortest path | To find the shortest path, depth-first traversal is used. First, the short_path_search function, a function based off the part 4 depth-first traversal function, is called. This function uses a similar algorithm to the part 4 function; however, it also involves the use of a current distances stack and a total distances queue. Furthermore, upon discovery, the simple paths are not printed. The first call to this function in part 5 stores all the distances of all the simple paths in the distances queue. The shortest distance is then found from the distances queue by dequeueing until it is empty, each time checking if it is a shorter distance, and then increasing the count (to be used like an index, later). The short_path_search function is then called again, this time however, the shortest path's location (the count from finding the shortest distance) is passed. The function thus iterates through all the simple paths until the shortest path is reached (when the count equals the shortest path's location). This path is then printed, along with its total distance.

The main data structures used in all three of these parts are stacks, for storing the path (or current path), and the current distances (parts 5), a queue, for storing total distances (parts 5), and arrays, for storing the visited Boolean values. Furthermore, the edges from each of the vertices are stored as linked lists.