Part 6: Graph Representation

The difference in representation would really only effect the part of each function where we iterate through neighbouring nodes. With an adjacency list, we have to follow a chain of pointers to edges, starting from `graph->vertices[node_id]->first_edge`, until we see a `NULL` pointer indicating the end of the adjacency list for that node. With an adjacency matrix, we would instead have to iterate through the corresponding row of a matrix within `graph` looking for non-zero entries, which would occur at the ids of adjacent nodes. These differences hold for both BFS and DFS.

Part 7: Design of Algorithms (and Complexity Analysis)

Finding a detailed path My algorithm traverses the graph starting from the source node using a modified recursive depth-first search. When a node is visited, its id is marked in a boolean array. Before exploring each unvisited neighbouring node, the algorithm adds the node id and the weight of the edge used to get there to a pair of stacks. If this exploration leads to the destination, the path will be maintained and the search will finish. If it does not lead to the destination, the node and its distance will be removes from the stacks and the search will continue.
Once the top-level exploration finishes, the first node is at the bottom of the node stack. The stacks are reversed (into another pair of stacks) and then the node labels are printed along with their cumulative distances from the starting node (as the ids and distances are removed in order from the other stacks).

Complexity analysis The analysis is like depth-first search. In the worst case the algorithm must create the visited array, visit every node once and inspect every edge twice (once from either end), and print the resulting path (which may contains all of the nodes). As a result, the time complexity of this algorithm is

$$O(n) + O(n) + 2O(m) + O(n) = O(n + m)$$

where n is the number of nodes and m is the number of edges.

Finding all paths My algorithm traverses the graph starting from the source node using a modified recursive depth-first search. Unlike a normal depth-first search, this algorithm may visit each node multiple times. To traverse the graph, it recursively explores all neighbouring nodes that are not currently in the path. This is checked by looking up node ids in a boolean array. Before exploring a node, we set its entry in this array to `true` and add its id to a path stack. When we finish an exploration, we reset its entry in the array to `false` and remove it from the path stack.
Whenever we encounter the destination node, we print the current contents of the path stack and then cut the exploration. To print a path, we must be careful not to ruin the stack (so that the remainder of the traversal is not effected). So, we empty the path stack into a second stack, and we then empty this second stack back into the original stack while printing the labels of the nodes on the path.

Complexity analysis The algorithm enumerates and prints all paths between the start node and the destination node. The worst case for this algorithm is a complete graph, in which there is an edge between every pair of nodes. In a complete graph of n nodes, the algorithm will look at about $n!$ paths[1]. For each path, which may be up to n nodes in length, we must print the nodes on this path. Thus, an upper bound on the time complexity of this algorithm is

$$O(n \times n!)$$

A better upper bound could be achieved by placing a tighter bound on the number of paths the algorithm will see and the number and lengths of the paths it will print. Additionally, if we had some

bound on the number of edges connected to each node (the graph's 'branching factor'), we could put a much tighter bound on the runtime.

Finding the Shortest Path (by enumerating all paths) This algorithm is very similar to the algorithm for Part 4. It maintains an additional path (node id stack) representing the current minimum-length path from the source to the destination. It also maintains the distance of this path, and the distance of the current path under exploration, by adding and subtracting edge weights as we add and remove nodes from the path.
Whenever the destination is found, instead of printing the path we instead compare its distance with the current minimum-distance path. If we have found a shorter path we copy the contents of the current path into a new minimum-distance path. At the end of the algorithm, we print the minimum-distance path and its distance (in a similar manner to printing each path in Part 4).

Complexity analysis The algorithm's analysis is the same as for part 4, except that path printing has been replaced by a potential path copying. Copying is also an $O(n)$ operation, and in the worst case we will find a new minimum path (and need to copy it) every time we encounter the destination. Therefore, the result will be the same.

Finding the Shortest Path (by Dijkstra's Algorithm) This algorithm is based on Dijkstra's Algorithm for finding the shortest path to each node from a single source. The algorithm maintains an array of distances for the shortest path to each node. It also maintains an array of previous node ids to later reconstruct these paths. Finally, it maintains a boolean array recording whether each node has been 'expanded'.
The algorithm proceeds by 'expanding' nodes until the destination is expanded. To expand a node, we update the distances and previous ids of each of its neighbours (if the path through the expanded node results in a shorter path). The algorithm chooses the next node to expand by looking through the distance array and the expanded array to find the not-yet-expanded node with the smallest distance. Once destination has expanded, the traversal ends. The path is then reconstructed by following the trail of ids in the previous array. This gives us the path in reverse, so the algorithm puts these ids into a stack, and them prints the labels for these ids as they are removed from this stack. The total distance is given by the destination's entry in the distance array.

Complexity analysis Like Dijsktra's algorithm, this algorithm may calculate the next-nearest node once for every node, and may inspect each edge twice (once from each side). Using an unsorted array (indexed by node id) as the underlying data structure, the next-nearest check takes $O(n)$ and an update takes $O(1)$, where n is the number of nodes. Thus, the algorithm has a time complexity of

$$n \times O(n) + 2m \times O(1) = O(n^2 + m)$$

where n is the number of nodes and m is the number of edges.

[1] How many paths will the algorithm see? In a complete graph, there is one simple path of length 2 between the start node and the destination node. There are $n-2$ paths of length 3. There are $(n-2)(n-3)$ paths of length 4. The number of paths continues in this manner all the way up to $(n-2)!$ paths of length n. That's not yet counting all of paths the algorithm will encounter that end in already-visited nodes: Such paths will not have to be printed, but will still be generated by the algorithm. Suffice to say it's a complicated analysis. In the mean time, an upper bound of $O(n!)$ is a good place to start.