Part 6: Graph Representation

Firstly, how the graph is iterated through would have to be changed. Both depth first and breadth first traversal would now iterate through the two-dimensional adjacency matrix either row-by-row or column-by-column.
In the depth-first traversal case, once an edge was found, it would store this in a list, stack or array for checking purposes (if it wasn't already travelled to), just like the current implementation of stack/list checking in my first 3 functions. Once this was added to the data structure, the iterator would then start iterating horizontally or vertically depending on the previous iteration. This is illustrated in the diagram below (assuming initially incrementing vertically).

	1	2	3	4	5	6
1		x				
2	x			x		
3				x		x
4		x	x			
5						
6			x			

The change to the code would be somewhat simple. Now instead of using a while loop and iterating via 'ptr->next_edge', the iteration would now be done via a simple incrementor until an edge was encountered (note: in figure 2 a for loop could also be used).

```
while (array_size < graph_size) {
    // If checking vertically
    while (graph[i][j] != 1) {
        i++;
    }
    ...
    // If checking horizontally
    while (graph[i][j] != 1) {
        j++;
    }
    ...
    add_to_stack(graph[i][j]); // or queue in BFS case
}
```

Figure 2: Pseudocode for iterating through an Adjacency Matrix

A breadth first search instead would iterate through an entire row or column, and after iterate through every edge (as shown in the diagram below).

	1	2	3	4	5	6
1		x				
2	x			x		
3				x		x
4		x	x			
5						
6			x			

For each valid edge not yet visited, the algorithm would then move to that row/column and proceed to iterate through all the edges. Iteration would also be like that in DFS, using a while loop to increment the indexes. This process would repeat until all edges were travelled to.

Part 7: Design of Algorithms

Part 3 uses a recursive DFS algorithm, unlike part 1. In this case, I found it easier to implement a recursive function as it was easier to find a simple path that can backtrack when reaching a 'dead end'. In this case the algorithm works by recursively stepping into the first valid edge. This vertex is checked against a stack of vertexes that have already been travelled to. As a new vertex is reached, the vertex is put into the stack. This continues until the destination is reached. This process repeats until the destination is reached, upon which the recursive call ends. Once the destination is reached, the path is copied into a separate stack to be printed later.

Part 4 also uses a recursive DFS algorithm. The algorithm is very similar to part 3, except that instead of terminating when one path has been found, the algorithm continues. Every time the destination is reached, the current path is printed, and the last vertex is popped from the list.

Part 5 is very similar to part 4, but instead, when the destination is reached, the algorithm calculates the total distance of the path. If the total distance of the path is shorter than the last shortest distance, then the path is saved into a new stack. After the recursive function calls are complete, the function prints the shortest path.

Time Complexity Analysis:

Part 3's DFS algorithm is recursive (unlike part 1). This function means that the average case is O(n^2). This is because the algorithm checks against the stack every iteration – resulting in a O(n^2) time complexity.

Part 4's algorithm also uses a recursive DFS search. However, this function ends in the average case O(n!) as it finds all possible paths – every combination of n possible. In the event every path must be found, this results in n! calls, and as a result worst case is n!.

Part 5 uses the same recursive DFS algorithm as part 4, and thus also has a worst case of O(n!). It also calculates the stack separately, but the O(n) is already in the order of O(n!).