**Part 6:**

If the graph was represented as an adjacency matrix rather than an adjacency list, stepping through the list for a depth-first search would be a case of stepping though the rows of the matrix looking for the next un-visited vertex, and adding it to the stack. (In the recursive case the function is re-called with the start id of the unvisited vertex.) Algorithmically, this isn’t dissimilar, but on a code level, it would probably mean dealing with less pointers, variables, and a less confusing storage structure, as you can simply iterate through the elements of each row, if the matrix structure was set up in such a simple manner. In the breadth-first case, each row would be iterated though once, and unvisited vertexes enqueued. While not as clever perhaps as the adjacency list, it is conceptually simpler in my opinion, and has a neater code solution as you don’t need to keep track of the next edge *and* its id, just where the counter is. In later parts, the values in the adjacency matrix could be weighted, allowing shortest path search to also be completed easily.

**Part 7:**

My solution to *part 3* is a slight modification on the depth-first search used in Part 1, with an addition of a cumulative weight counter and a test to see if the destination node has been reached.

The algorithm for both parts 1 and 3 takes a note of all the visited nodes in a (0, 1) array, the length of the number of vertices. A stack is kept containing the vertices planned to be visited, the last vertex added to be the next visited. The source is pushed onto the stack, and the algorithm begins, popping the top element, effectively “visiting” it, and looks through each edge, pushing the next un-visited vertex onto the stack and “visiting” that, until all the vertices have been visited or the destination is reached. Complexity analysis can be found after the other part analyses.

My solution to *part 4* is a recursive Depth-First Search approach, in that it keeps track of which vertexes are in the current path on a stack, then recursively calls itself on each unvisited connected edge (kept track of in an array) to the current vertex. Each time the destination is found, the algorithm pops the current node from the path, marks it as unvisited again, and runs the next edge down until all the connected edges have been tried. If the desination is not found, the path is ignored. Complexity Analysis can be found below.

My solution to *part 5* is a modified Dikjstra’s variant which uses a queue to store the order of vertices to check distances from. The total distance to each node and the node shortest node to previous are stored in arrays which are updated if the current distance is less than the stored distance, or if the stored distance is “infinity” (or -1 explicitly). The path is traced with a stack, each node of ‘prev’ is followed until the source node is found.

**Complexity Analysis:** Note that O(1) costs are ignored due to them being numerous and mostly inconsequential.

**Part 3 Complexity Analysis:**

* O(n) to build a visited array.
* O(n) maximum to visit each vertex.
  + O(m) to step through the connected vertices to find an unvisited vertex.

On inspection it appears that DFS is O(nm) however we know that each edge is only traversed once, so the complexity must be O(n + m), where n = number of vertices and m = number of edges.

**Part 4 Complexity Analysis:**

* O(n) to build visited array
* O(n!) to permute all paths
* O(m) to print all elements in a path, where m is the number of elements in the stack.

By inspection we know that generating all paths for n nodes must be O(n!) as there are n! possible paths. Generating all paths is a non-polynomial problem, and as such, is not a useful solution to real-world problems.

**Part 5 Complexity Analysis:**

* O(n) to build dist and prev arrays
* O(n2 + m) maximum to check and update all distances
* O(p) to build and print the path on the stack.

So this implementation should be about O(n2 + n + m + p) to run. Not efficient in this case, but it should be functional.