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

My solution to part 5 is

Part 3 Complexity Analysis:

* O(n) to build a visited array.
* O(1) to initialise the algorithm.
* O(n) maximum to visit each vertex.  
  Arbitrary O(1) variable updates are herein ignored as they’re inconsequential yet numerous.
  + O(m) to step through the connected vertices to find an unvisited vertex. Note that (m <= n-1) even for a complete graph where all but one element has been visited, but is usually less than that.

So DFS is of O(nm +n), where m <= , which is O(nm).