SE 2C03 - Assignment 4 solution suggestions

4.1.30 Note that a graph has Eulerian cycle(s) if and only if starting from a node we can explore all edges (one at a time) by entering and then leaving a node through previously unexplored edges (if you get stuck at a node, then the graph hasn't such a cycle); if you end up at the starting node with all edges transversed, then the graph has such a cycle. You can easily see how to modify DFS to continue its exploration whenever there is an unvisited edge, even when the node you end up has already been visited (so here you mark *edges*, not *nodes*). Since you transverse each edge at most once, your algorithm is linear ($O(m)$).

4.1.31 Since the vertices are distinct, every graph has a different subset of all possible $\binom{n}{2} = \frac{n(n-1)}{2}$ edges; there are exactly $2^{\binom{n}{2}}$ such subsets.

4.1.32 We maintain a boolean array of all the nodes, initialized to all 0's. At every node v, we use this array to keep track of its neighbours (every time we see that a node has already been marked as a neighbour of v we count it as a parallel edge). After we are done, we do another pass over the adjacency list of v to reset all the marked array elements to 0, and then we move to the next node (at the end, you may need to divide your count by 2 if you have counted an edge for both its endpoints). Note that for every edge in the adjacency list, we access one element of the array twice. Therefore, the number of array accesses is proportional to the number of edges, and the number of total accesses (array+lists) is linear.

4.1.38 Follow the hint and use a stack to simulate the recursion tree of recursive DFS, together with the `marked[]` array.

4.2.31 A straight-forward answer would be to run a DFS(v) in G to discover all nodes in G that are reachable starting from v, and then a DFS(v) on the reverse graph G^R to discover all vertices that can reach v in G (and, therefore, can be reached from v if we reverse all edges). Then take the intersection of the two sets. Repeat for each node, to get the quadratic algorithm.

4.2.41 On the web site for the course, you can find a proof of the fact that no odd-length directed cycle exists iff G is bipartite. A simple algorithm then is to run BFS, painting each level of nodes discovered blue or red in turn. If there is an edge between two same colour nodes that closes a cycle, then this cycle is odd-length.

4.2.42 If there are two (or more) nodes of outdegree 0 then one of them is not reachable from the other. Then it is easy to see that in a topological ordering, this unique node would be the last node, and there is a path from any node to this one following edges in a left-to-right direction. Therefore, all we need to do is to check all nodes and count the number of nodes with outdegree 0 (in linear time).

4.2.43 Following the hint, first calculate the kernel graph (in linear time), and then run 4.2.42 on it.

4.3.4 Not true (G=tree with two edges with the same weight).

4.3.8 The path will have to cross the cut with an edge cheaper than the edge crossing it in the given MST, a contradiction.

4.3.20 It is true. If there is an edge from v to outside its component that is cheaper than the vertex connecting v to its component subtree, then Kruskal would have chosen this one first, since it wouldn't close a cycle.

4.3.32 If W is the minimum edge weight in G, then set the weight for all edges in S to $W-1$; then Kruskal would pick first all of S (since there is no cycle in S) before starting picking from the other edges. To see that the algorithm is correct, imagine that the components of S are collapsed to single nodes. Then Kruskal's algorithm proof of correctness proves that we will pick the cheapest cost edges from the rest of G.

4.4.22 Replace each node v by two nodes v^+, v^-, and edges $(v^+, v^-), (v^-, v^+)$, each with weight $weight(v)$. Then connect all edges incoming to v to v^+ and all edges outgoing from v to v^-.

4.4.25 Add new vertices s and t, and edges $(s, v),\ \forall v \in S$ and $(v, t),\ \forall v \in T$, all with weight 0. Then running Dijkstra's algorithm on the new graph to find the shortest path from s to t, will give the shortest path from S to T.

4.4.33 Create a graph where every grid cell is a node with weight equal to the cell value, and has outgoing edges towards its neighbouring cells (no weights on edges). Then you need to find the shortest path from node $(0, 0)$ to node $N-1, N-1)$, which can be done using exercise 4.4.22. For the second part, create edges that are only going to the right and the bottom neighbours of a cell.

4.4.40 Let P_{uv} be the path connecting u and v in the MST, and let b be the length of its longest edge. Suppose that there is a different path $P'_{uv} \neq P_{uv}$ from u to v with its longest edge shorter than b (therefore, all its edges are shorter than b). Then we have found a cycle $P'_{uv} + P_{uv}$ whose longest edge is the one of length b in the MST, a contradiction of exercise 4.3.8.

4.4.47 If an edge (u, v) is relaxed in the Vth pass, then the shortest path from s to v has at least V edges, i.e., it has at least $V+1$ nodes, therefore it contains a cycle. This cycle has to be a negative cycle, and it is easy to output it following the `edgeTo[]` entries (since they 'define' the shortest paths).