A2 W2016

1. [Covered in notes]
We define low[v] as min(start[v], min(start[w] where uw is a back edge where u is a descendant of v or u=v)). We run DFS on the graph and compute start[v] and low[v] starting from the leaves of the tree. For each leaf, we compute low[v] as the minimum of start[w] of all edges vw. For non-leaves, we compute low[v] as the minimum of the low value of its children as well as the start time for all back edges involving v.

Cut vertices: For a non-root vertex v in a DFS tree, v is a cut vertex if and only if there is a subtree below v with no edges going to an ancestor of v. Hence for each non-root vertex v, v is a cut vertex if and only if low[u] < start[v] for all children u of v. The root vertex is a cut vertex if and only if it has at least 2 children.

Cut edges: A back edge is never a cut edge. A tree edge uv is an edge if and only if without using edge uv, there are no edges from the v-subtree to a strict ancestor of v. Hence an edge uv is a cut edge if and only if it is a tree edge and low[v] > start[u].

Runtime: It takes O(m+n) time to run DFS, and O(m+n) time to iterate the vertices and compute the comparisons for cut vertices and edges.

2. We can model this as a graph problem, where each vertex is a permutation of the numbers, and there is an edge uv if u can be formed from v with one reversal. Finding the smallest number of required reversals between two permutations is equivalent to finding the shortest path between the two corresponding vertices in the graph.

There are n! vertices in the graph. Each permutation has 1+2+..+n-1 = 1/2(n^2-n) edges. Hence there are n!(n^2-n)/4 edges in the graph, by the handshaking lemma.

BFS takes O(|V| + |E|) = O(n!*n^2) time. There are ~n^2 states of distance 1 from the start, n^4 states of distance 2 … n^2k states of distance k from the starting state. In BFS, we only visit these states, so we have

$$\sum_{i=1}^{k}(n^2)^k = \frac{n^{2k+2}}{n^2-1} - \frac{n^2}{n^2-1} \in O(n^{2k})$$

Hence the runtime of our algorithm is O(min(n^2k, n!*n^2)).

3. [A2 problem]
We use a modified BFS algorithm to return the number of shortest s-t paths. We initialize an array numpaths, where numpaths[v] is the number of shortest paths from s to v. We initialize this array to 0, except numpaths[s] = 1. We also initialize an array mindist, where mindist[v] is the minimum path length from v to s. In each iteration of the BFS, we update numpaths as follows. Let v be the current vertex we are traversing in the BFS. For each of its neighbours u, we:

    If u has not been visited, we set numpaths[u] = numpaths[v], and mindist[u] = mindist[v] + 1.

Otherwise, if mindist[u] = mindist[v] + 1, we update numpaths[u] to numpaths[u] + numpaths[v].

At the end of the BFS, we return numpaths[t].

Correctness: We prove correctness of this algorithm by induction on k, the length of the shortest path from s to any vertex t.

Base case: k=0. Then s=t, and there is one shortest path from s to itself. We initialize numpaths[s] = 1, so we return the correct solution.

Inductive step: Assume the algorithm computes the correct number of shortest paths for all paths of length k. Every path of length k+1 from s to t must pass through a neighbor of t with distance k from s. By the inductive hypothesis, our algorithm computes the correct number of k-length shortest paths. Hence our algorithm computes the correct number of k+1-length shortest paths.

4. [A2 problem]
We model the problem as follows: given an undirected graph (each vertex is an intersection, and each edge is a two-way street), convert it into a strongly connected directed graph (each vertex is an intersection, each edge is a one-way street, and each vertex is reachable from every other vertex in the graph). If the graph is not connected, it is obviously not possible, so we assume the graph is connected.

Our algorithm operates as follows. We first check if G contains any cut edges using the algorithm specified in the tutorials. If G contains a cut edge, we return that it is not possible to convert. Otherwise, we convert it into a strongly connected directed graph as follows.

From above, we have a DFS tree of G, rooted at some vertex s. We set the direction of all tree edges away from s (pointing down the tree). We set the direction of all non-tree edges towards s (up the tree, ie back edges). There are no cross edges, since we did DFS on an undirected graph, so all edges incident to a vertex were explored. We then return this new directed graph.

Correctness: If there is a cut edge uv in the graph, if we assign it a direction u->v, then u must be unreachable from v. We now show that our algorithm for generating the directed graph creates a strongly connected one. We have that G is strongly connected if and only if every vertex v is reachable from s (the root), and s is reachable from every vertex v. Every vertex is reachable from the root by construction. For every tree edge uv, there exists a directed path from v to u, and then, the general claim follows by induction. Since uv is not a cut edge, then low[v] <= start[u]. This means that there exists a back edge from some vertex b in the subtree of v to u or some ancestor a of u. Then the path v -> b -> a -> u is a directed path from v to u.

Runtime: Our algorithm for finding cut edges takes O(n + m) time. The algorithm for converting the undirected graph into a strongly connected directed graph runs DFS once, which takes O(n + m) time. Setting the direction of each edge takes O(m) time. Hence our algorithm runs in O(n + m) time.