lMin Spaning Trees

1.1
Suppose in a minimum spanning tree of graph G has a path between vertices s and t which is not the minimax path of G.

=> between s and t there exist a path which has a lower weight
=> replace the path between s and t in the original MST will further lower the net weight of the whole MST
=> which contradicts of definition of MST

Q.E.D.

2.1
Suppose the most expensive edge e = (v, w) \in C belongs to a MST of G, and the circle consists of edges between v, x, y, z … w

=> break the MST at edge (v, w), by definition of MST, this will return two MSTs A and B
=> C is a circle, and before cutting off the edge (v, w), the circle is in no way to be fully connected (definition of MST).
=> Could find a edge on the circle (a, b) \in C other than (v, w) to connect A and B
=> & edge (v, w) has the highest weight in circle and in graph G all edge weight are distinct
=> edge (a, b) has a lower weight than (v, w)
=> contradicts with definition of MST

Q.E.D

2.2
Suppose edge (v, w) does not belong to any MST of G

=> MST can be split up to subset S and V - S where S is neither empty set nor V
=> MST must connects S and V - S through an edge but this edge can not be (v, w)
=> (v, w) has the lowest weight connecting S and V - S, and all edges in G has distinct weight.
=> connect S and V - S through (v, w) will yield a lower cost
=> contradicts with definition of MST

Q.E.D.

2.3.1
Prove e = (v, w) does not belong to a MST of G **if** v and w can be joined by a path consisting entirely of edges that are cheaper than e.

=> Suppose v and w can be joined by a number of edges [e1, e2, e3 … en], where e1, e2, … en all have weight lower than e
=> [e , [e1, e2, … en]] makes a circle C

=> from 2.3 we know the heaviest edge in a circle cannot belong to any MST of G
Q.E.D

2.3.2
Prove v and w can be joined by a path consisting entirely of edges that are cheaper than e = (v, w), if e does not belong to a MST of G.

=> e does not belong to a MST of G
=> from 2.1 it must NOT be the edge connecting S and V - S where S and V - S covers all possible combinations in G
=> connecting any S and V - S has best connecting edge other than e, i.e. can find e' connecting S and V - S with lower cost than e
=> S can be any set of nodes, which concludes on any MST, every edge must have lower weight than e
=> by the definition of MST, v and w must be connected by some path
=> while previously we proved that every path on MST must have lower cost than e
Q.E.D

3.
Suppose that in the adjacency matrix, A(i,j)=1 indicates that candidate i prefers candidate j.

Objective:
We need to find a vertex in this graph, where its corresponding row are all 0s and its column (except self) are all 1's, which makes it a "perfect president".

Observations:
1) Either there exists such a vertex, or it doesn't exist at all: If there are two "perfect presidents", according to the definition, they have to prefer themselves only, such that they can't prefer each other, which makes it a contradiction.
2) If we have observed that candidate A doesn't prefer candidate B, we can immediately rule out B, according to the definition.
3) A(i,i) = 1, which we can skip over while traversing the matrix.

Algorithm:
We start by traversing the first row (skip over self). If we ever encounter A(1,j) = 0, it rules out vertex j from the "perfect candidate".
a) If we have not encountered 1 in the entire row, we then check if all numbers in the first column is 1. If yes, then the first candidate is the "perfect candidate", otherwise there is no such "perfect candidate". The algorithm exits.
b) If A(1, 2...k-1)=0 but A(1,k)=1, it suggests that vertices 1…(k-1) are ruled out from "perfect candidates", because the first candidate did not prefer them. It can be observed that we have eliminated k-1 vertices in k steps. Then we start from A(k, k), traverse its row like we did in the first row. If we reached the end of the row, start from the beginning of the row again, to make sure this row is all zeros.

Time complexity:

1) Because every step we eliminate one candidate, searching for a row with all zeros takes at most O(2n).
2) Checking if a row consists of all ones takes at most O(n), which only occurs once.

Total time complexity: O(2n) + O(n) = O(n)

4.(a)

step1: take u0 as the source, perform DIJKSTRA (using min-heap as the min-priority queue). Store the d of each vertex in S

step2: change all the edge to the opposite direction, take the u0 as a source, perform DIJKSTRA algorithm.(using min-heap as the min-priority queue). Store the d of each vertex in S'.

step3: Iterate all the vertexes pairs, calculate the d(u,v) = u.d(in S') + v.d(in S)

Proof of correctness:

Lema: the shortest path between any two vertex (t,v) passing through u0 equals to the summary of d(t, u0) + d(u0, v).

Assume the contradiction, there exists a path with the restriction, while the path from t to u0 p1, is not the shortest path, or the path from u0 to v p2 is not the shortest path. Since if the path is the shortest, its weight is smaller or equals than any other path and the d(t, u0) + d(u0, v) <= any(w(p1) + w(p2)). There exists the contradiction. QED.

From the Lemma above, our task is to finding the shortest path of (t, u0) and (u0, v) for every vertex. If we change the direction of edges, the shortest path between (u, v) is equivalent to shortest path between(v, u). Therefore we change the direction of edges and use the DIJKSTRA to find the shortest path between each pair. The correctness of DIJKSTRA'S does not need to prove any more. Q.E.D

Time complexity:

if we implement the min-priority queue by using the min-heap, running DIJKSTRA once costs O((E+V)lgV) time and we run twice therefore T = O((E+V)lgV)

Space complexity:

the min-priority queue uses up O(V) space to store the vertex and its d;
In order to store shortest distance of each pair, we need O(V^2) space
in summary, space = O(V^2)

(b) Algorithm:

Step 1: Reduce weighted-graph to unweighted-graph by introducing new pseudo-nodes: For each edge e with weight w, add w-1 new pseudo-nodes to this edge.

Step 2: Do depth-first-search on the new graph from any given node, which solves the single-source shortest path algorithm.

Proof of Correctness:

According to CLRS pg 599, BFS algorithm correctly computes the shortest path from a single vertex.

Time complexity:

1) Construction of the new graph: Suppose the graph is represented by adjacency lists, constructing the graph (also the space complexity of the new graph) takes at most c*(|V|+|E|) operations.
2) Doing BFS on the new graph: The new graph has at most c*|V| nodes, and c*|E| edges. According to CLRS, the BFS takes at most c*|V|+c*|E| operations.

Total time complexity: c*(|V|+|E|) + c*|V|+c*|E| = O(|V|+|E|)

Space complexity:

1) Original graph if using adjacency list: |V|+|E|
2) Space of new graph if using adjacency list: at most c*(|V|+|E|)
3) Space of queue in BFS: at most c*|V|

Total space complexity: |V|+|E| + c*(|V|+|E|) + c*|V| = O(|V|+|E|)

5. A query establishes an ordering between two nodes, i.e. (u,v)'s existence can be interpreted as and u > v. Since the graph is complete, there exists an ordering between every pair of nodes, in other words, we have a total ordering on the graph. We can simply run a comparison-based sort to output a topological sort. The "max" element will be a source node with out-going edges to all other nodes. Similarly, the "min" element will be a sink node. Since query is effectively a comparison operator, we can use any sorting algorithm such as heap sort, quick sort etc.There is a tight Ω（nlogn） lower bound on performing a topological sort in this model.

Proof of correctness:
As stating above, since the graph is strongly connected, the query can be established as a comparing. Therefore, the vertex has ordering. The correctness of sorting algorithm does not need to be proved.

Time complexity:
Ω（nlogn）

6.(a)
We analyze the amortized cost of the merge operation with accounting method. First, we make the following definitions:

1) If a node is its parent's right heavy node, we abbreviate it as right heavy node.
2) A path in a skewed heap can have at most log(n) light nodes.

For nodes on the right edge of an skewed heap, we have the following observation:
If a node is a right-heavy node, then after the merge and swap operation, it is replaced by a right-light node. This is because its weight only increases after merging, but the tree rooted at its sibling remains the same. However, a right-light node may not become a right-heavy node after merging.

Now, given two skewed heaps S1 and S2, we assume l1 and h1 depict the number of right-light nodes and right-heavy nodes on the right edge of the heap S1, l2 and h2 for S2 likewise.
At every step, we assign an amortized cost as follows:

1) If after swapping, a right heavy node is spawned, such an operation have an amortized cost of $2. One dollar is for the operation itself, and another dollar deposited to swap the heavy-right node back.
2) If a right heavy node on the right edge is eliminated after swapping, the cost is $0.
3) Otherwise, the amortized cost is $1.

In the worst case, which we suppose every swap from a light-node generates a heavy-node, it takes an amortized cost of 2(l1+l2). Note that we have stated that a path in a skewed heap can have at most log(n) light nodes, we can further infer that the merge operation have an amortized cost of O(logn).

(b)
Insert: Treating the inserted node as a skewed heap with only 1 element, the insert operation is essentially merge, taking also O(logn).

Delete-Min: Remove the min element at the root produces two new skewed heaps. Merge the two skewed heaps takes O(logn).