Jung Yoon (jey283)

Take Home Exam #1

1.

a. By Master's Theorem:
   a = 6
   b = 4
   f(n) = n^2 = n^d
   d = 2
   k = log base 4 (6)

   Then, a ? b^d
   6 < 4^2
   so T1(n) = Θ(n^2).

b. By Master's Theorem:
   a = 4
   b = 2
   f(n) = n^2 = n^d
   d = 2
   k = log base 2 (4)

   Then, a ? b^d
   4 = 2^2
   so T2(n) = Θ(n^2 * log (n)).

c. By recursion tree since Master's Theorem can't be applied when b cannot be expressed as a constant:

   First, note that each level will be n, n/n^1/2, n/n^¼, etc. until n gets small enough. This means that n^(½^k) where k is the number of layers/iterations, giving us:
   n^(½^k) = 2^(log n/2^k).

   We can cancel out the 2 on the right because it's a small enough value for n and then we get:
   (log n/2^k) = 1 such that log (n) = 2^k.

   By simple algebra we can see that
   log (log (n)) = log (2^k)
   log (log (n)) = k * log (2) = which is about k since we can omit the small log (2).

   Thus, T3(n) = Θ(log (log (n)).

d. Order of Complexity (fastest to slowest):
T1, T2, T3

2.

Strategy:

We can use a variation of Kruskal's Algorithm to find the maximum spanning tree such that we negate the weights and use Kruskal's Algorithm to find the minimum spanning tree, thus leaving us with the maximum spanning tree at the end. The steps would be as follows:

- start with T = empty set.
- Negate weights for each edge
    - ex: edge (u,v) that originally had weight 15 would become -15
- Consider edges in ascending order of cost
- Delete edge e from T unless doing so would disconnect T.
- Continue until termination and negate weights again to get the maximum spanning tree on the original graph with the original weights.

Proof of correctness:

First, let's note that that Kruskal's Algorithm works perfectly fine for negative weights since the order by ascending weight will not be affected and also since the algorithm always takes the minimum weight not in the set T and in the remaining edges in ascending order that meet the requirements.

Then, let's consider edges in ascending order of weight after we have negated each edge. This is just a proof for Kruskal's Algorithm for minimum spanning tree:

Case 1: If adding e to T creates a cycle, disregard e according to cycle property.
Case 2: Otherwise, insert e = (u,v) into T according to cut property where S = set of nodes in u's connected component.

End of proof for Kruskal's Algorithm for minimum spanning tree.
This shows that the tweak to the original algorithm does not violate the conditions required by the theorems/properties.

Then, let's negate the edges again to get the original weights and we are left with the original graph and the maximum spanning tree.

End of proof.

3.

Proof:
Given everything stated in the question itself, we can proof by observing each case.

→ Case 1: W = W* + 1

For such a case, there would be only one edge in the graph G with a weight of 11 and the rest would have to have weights of 10.

This would mean that W > W* still holds and that there is in fact a spanning tree of the graph G where the weight is equal to W - 1 since that particular graph would be the minimum spanning tree, a spanning tree in its own right.

→ Case 2: W - W* > 1
For such a case, there would be more than one edge in the graph G with a weight of 11 with the rest being weights of 10's.

This is to say that W = W* + 1 + i where i is the number of edges with weights of 11 - 1. We'll call this number E. To restate this, E = #edges_with_weight_of_11 - 1 where E >= 1.

Since we know that E >=1, we know that there is a spanning tree of G that would maintain that its weight is W-1 since all we'd have to do is use E edges with weights of 11's and fill the rest of the spanning tree with 10's to get the correct weight.

End of proof.

4.
Let's say that the 5 strongly connected components arranged as so:

SCC1 → SCC 2 → SCC3 → SCC4 → SCC5

Then, by adding a directed edge from SCC5 to SCC1, we are left with one SCC since now, the entire graph is strongly connected.

5.
Proof:
Let's assume that BFS and DFS trees always start at the same node s making up layer 1. Then, implementing BFS, we fill up layer 2 with all nodes connected to node s. Implementing DFS, we don't pay mind to the number of layers and explore as many nodes as possible from node s recursively and backtrack when you are left with a leaf. This process continues until there are no more nodes left to visit.

Given this, since BFS finds paths using the fewest number of layers since it fills up layers first as much as possible unlike DFS that "selfishly" explores all nodes at the cost of depth, the BFS depth must be at least equal or smaller than the DFS depth.

Thus, DFS trees have a depth greater or equal to BFS trees.

End of proof.