CS 180 Homework 3

Prithvi Kannan

405110096

1. Exercise 10 Page 110

We perform a modified breadth-first search from vertex v to get the distance of each node n to v. If the path $n - v$ has layer numbers that increase by one for each node along the path, then $n - v$ is the shortest (aka most direct) path from n to v. In addition to normal BFS, we store a count at each node which keeps track of the number of shortest paths to that node in the original graph.

To prove we are counting the number of shortest $v - w$, we apply induction. For nodes in the first layer, there exists only one path, so $count(n) = 1$ if n is part of L_1. Consider a node w in layer L_i. The shortest path from $v - w$ is composed of a path from v to a node x in L_{i-1} and a path of length 1 from $x - w$. Therefore, $count(n)$ is the sum of all nodes in layer L_{i-1} with an edge to w.

The BFS from vertex v takes time $O(m + n)$. Then we compute count for each of the nodes, which will take at most the degree of the node. Using a degree-centric approach, the total sum of degrees in the graph is $O(m)$, so the runtime of the algorithm is $O(m + n)$.

2. Exercise 6 on page 108

We will prove that G must be a tree by contradiction. Assume G is not a tree, meaning there is an arbitrary edge e, from n_i to n_j for $i \neq j$, that is not contained in the BFS/DFS tree T. Assume that n_i is n_j's parent in the tree T.

In the BFS tree, since n_i and n_j's distance from the root u can differ by at most one. However, in the DFS tree, if n_i is n_j's parent, then the distance $u - n_j$ must be one more than $u - n_i$. Therefore, the all edges in G will exist in both the BFS and DFS tree, so by contradiction, G must be a tree.

3. Exercise 7 on page 108

Let G be a graph on n nodes, where n is an even number. If every node of G has degree at least n/2, then G is connected.

We will prove this is true by indirect proof. Assume there exists arbitrary node a and b in G for which there exists no path, meaning G is not connected.

As required, the degree of a and b is at least $\frac{n}{2}$. Since we assume that a and b are not connected, then all the nodes connected to a cannot also be connected to b (if there was a node x that was connected to a and b, then there would be a path $a - x - b$ and the graph G would be connected). Therefore we have $\frac{n}{2}+1$ nodes in the component containing node a and $\frac{n}{2}+1$ nodes in the component containing node b, since a and b have degree of at least $\frac{n}{2}$. The $+1$ comes from the fact that we must also add node a or b into the count. This would mean we have at least $2 * (\frac{n}{2} + 1) = n + 2$ nodes in the graph, but since there are only n nodes in the graph, this is a contradiction.

For this reason, G must be connected if each vertex has degree of at least $\frac{n}{2}$.

4. Exercise 3 on page 189

The goal is to minimize the number of trucks, n. A valid solution assigns boxes to trucks such that no truck is carrying more that weight W and the order of the boxes is preserved.

The greedy algorithm is as follows:

```
Assume we have a queue of packages
While truck t's available capacity is less than W
    If possible
        Add the next wi package from queue
    Else
        Send truck t off
        Increment n
```

We now show by induction that the greedy algorithm will always be ahead of any other algorithm, defined as having boxes $b_1, b_2, \ldots, b_i$ in t trucks while another algorithm may have $b_1, b_2, \ldots, b_j$ in the same t trucks for $i \geq j$. For the base case, $t = 1$, both greedy and non-greedy algorithms will fit as many boxes as possible into one truck. Now we assume greedy is ahead for $t = k$, meaning greedy has fit $i \geq j$ boxes, and prove that greedy is still ahead for $t = k + 1$. In truck $k + 1$, the greedy algorithm will pack $b_{i+1}, b_{i+2}, \ldots, b_{i'}$ and the non-greedy algorithm will pack $b_{j+1}, b_{j+2}, \ldots, b_{i'}$. Since $i \geq j$, the greedy solution may still have space for more, making it the optimal solution.

5. Exercise 6 on page 191

```
Calculate each contestants combined bike and run time
Sort in decreasing order and send out contestants
```

Consider the optimal solution and assume our algorithm produces a different result. The optimal solution will have candidate a and b such that the order is $\ldots, a, b, \ldots$ and $b_a + r_a < b_b + r_b$. If we switch a and b, then b will finish the race earlier than they did before swapping. Candidate a will now get out when b used to get out, but since a's combined bike and run time is less than b's, a will finish before b's old finish time. So, switching a and b in this situation reduces the total time since a and b both finish earlier than they did before swapping. Performing these swaps will only lower the overall race time, and eventually end up at the solution no faster than that produced by the algorithm, meaning the algorithm gives the optimal solution.

This algorithm runs in $O(n)$ to calculate the combined bike and run time, and then $O(nlogn)$ to sort in decreasing order. This gives an overall runtime of $O(nlogn)$.

6. (a) Can you design an algorithm that finds the longest path in a directed graph (DG)? (you can use an edge at most once)? If yes, describe the algorithm and analyze its time complexity.

This problem is NP-hard and cannot be computed in polynomial time. There is a brute force solution to this question which analyzes all possible paths within the graph and finds the longest path.

(b) Can you design an algorithm that finds the longest path in a directed acyclic graph (DAG)? (you can use an edge at most once)? If yes, describe the algorithm and analyze its time complexity.

```
Run a topological sort algorithm on the graph
Keep track of the longest distances to each vertex
Looping through each vertex v in topological order
    Loop through all vertices u connected to v
        If the distance of v is less than dist u+1
            u+1 is new dist of v
```

The topological sort takes time complexity $O(m + n)$. The algorithm will go through each vertex in the topological sort and examine all adjacent vertices. The total number of adjacent vertices in a graph is $O(m)$, so this analysis runs in $O(m + n)$.