CS490 Quiz 1

NAME: ______________________

STUDENT NO: ______________________

SIGNATURE: ____________________________

This is the written part of Quiz 1. The quiz is closed book; in particular, no notes, calculators and cell phones are allowed.

Not all questions are of the same difficulty, so be sure to pick up the easy points first, before tackling hard problems. The quiz is divided into two parts:

Part 1: [15]

Answers in part 1 should be succinct. Erroneous “extra” comments may decrease your score.

Part 2: [30]

When designing algorithms in part 2, the use of pseudocode is greatly encouraged! Feel free to quote algorithms we described in class, but make sure you describe how the algorithm is used. (e.g. to quote Dijkstra, you should specify your source node and where the results are stored, say an array. Then, go on to explain what you do with those results.)

NOTE:

Since graph datastructures affect the run time of many graphs, it is important to specify which one you’re using (adjacency list, adjacency matrix, edge list). Assume you have access to all three datastructures on all the questions.

Part 1 - Short Answers [5 points each]:

1. Let P = (5, 3, 1, 2, 5, 2, 1), a permutation of the sorted 7-tuple (1, 1, 2, 2, 3, 5, 5). Give the next permutation after P, if all the permutations are ordered lexicographically from (1, 1, 2, 2, 3, 5, 5) to (5, 5, 3, 2, 2, 1, 1). No explanation is required.

   {3}: (5, 3, 1, 5, *, *, *)
   {2}: (*, *, *, *, 1, 2, 2)

2. Suppose we have an undirected, non-negative integer weighted graph G, and two nodes u, v, in G. Further assume that we have found the maximum flow from u to v using the Ford-Fulkerson algorithm. If we now add one edge of capacity C (also a non-negative integer), how many additional augmenting paths would we need to find, in the worst case, to update the flow? Explain briefly.

   {3}: Each augmenting paths will add at least one unit of flow to the whole graph.
   {2}: Since the flow can increase by at most C after adding that new edge, we need at most C augmenting paths.

3. Given an arbitrary undirected, simple (no self edges, duplicate edges) graph as an adjacency list, determine in O(|V|) whether or not the graph is a tree (connected, undirected, acyclic graph). Show that your algorithm has the desired complexity.

   {1}: Run DFS from arbitrary start point.
   {2}: If you see a back edge (i.e. white-grey with coloring, or found a "seen" children with no coloring), then graph isn't a tree
   {2}: We can break if we see more than |V| edges. So, "truncated DFS" runs in O(|V|) time. (Use adjacency list).

Part 2 [10 points each]:

1. Consider a connected, positively weighted, undirected graph, we define the “length” between two nodes as the (weighted) length of the shortest path connecting them. The diameter of a graph is then the longest such length over all pair of nodes. Since the graph is connected, the diameter won’t be infinity. In the questions below, |V| = number of nodes in the graph.

a. [5] Give an O(|V|^3) algorithm that will find the diameter of a graph.
b. [5] Give an O(|V|) algorithm that will find the diameter of an unweighted tree. (recall that a tree is a connected acyclic undirected graph)

For part a, we can find all the shortest paths, and find the maximum one.
e.g.:

{2} Get all shortest paths.
{3} With Floyd-Warshall, that takes O(|V|^3) time. An extra O(|V|^2) time to search the shortest paths between all pairs will give the answer.

For part b, we can use BFS and / or DFS

Using BFS:

{3} Choose arbitrary start point, *s*. Run a BFS to find the furthest node, say *u*.
{2} Run a second BFS from *u*, and the distance to the furthest node is the diameter. (Prove it!)

Using DFS:

{3} Run DFS from arbitrary start point. At each node keep track of the length of it’s longest leaf (LLF). Do this by taking the maximum LLF of its children + 1.
{2} Calculate what’s the longest path that go through each node, by adding up the two distinct children with the largest LLF’s. (If there isn’t enough children, use 0).

Note: In both cases, DFS/BFS runs in O(|V| + |M|) = O(|V|) since M = V-1 for a tree. (Use adjacency list).

[10] We're given an undirected, positively weighted graph and four nodes in the graph: a, b, c, d. Person X would like to walk from node a to b, and person Y would like to walk from node c to d. Both X and Y would like to take some shortest path between their start and end points. Design an algorithm that will determine if it is possible for X and Y to meet each other during their travels. (i.e. for X and Y to travel through a common node, while both are taking a shortest path).

Let sp(a, b) denote the shortest path from a to b. Let | • | denote the length of a path.

{6} Recognize that ***u*** is on some shortest path from ***s*** to ***t*** iff:
| sp(***s***,***t***) | = | sp(***s***, ***u***) | + | sp(***u***, ***t***) | … (*)

{2} Get shortest paths from a, c, to all nodes, and get shortest paths from all nodes to b, c. Do this with Floyd-Warshall, four Belman-Ford, or four Dijkstra.

{2} Check all nodes (including a, b, c, d) for property (*).

[10] Consider a directed, connected graph in which the nodes are weighted (positive or negative), while the edges aren't. The weight of a path is now defined as the sum of the weights of the nodes that it uses. Given two nodes, A START NODE AND AN END NODE, design an algorithm that will determine the LENGTH OF THE shortest path between them. If there are "negative cycles", the algorithm should detect it. For full marks, the algorithm should run in O(|V||E|) (|V| = number of nodes, |E| = number of edges).

Algorithms with longer run times will receive partial marks.

{6} Any correct algorithm.
{1} Runs in polynomial time
{1} Runs in O(|V|^6)
{2} Runs in O(|V| |E|)

Here's one algorithm that gets perfect 10. Assume a start and end node is given.

Use adjacency list. Take graph, and for each edge from a -> b, assign it the weight of b. Now run Bellman-Ford with the start node as your source (no Dijkstra due to negative weights). Finally, add the weight of the start node to the length of the shortest path Bellman-Ford gives from start to end.

Bellman-Ford is O(|V||E|) as desired.