COSC 240, Spring 2020: Problem Set #4

Assigned: Thursday, 3/26.
Due: Tuesday 4/14, submit electronically by the beginning of class.
Lectures Covered: 18 to 23
Academic Integrity: You can work with other people in the class but you must write up your own answers in your own words. You can also use the textbook and talk to the professor. You may not use any other resources (e.g., material found online) or talk to people outside the class about these problems. See the syllabus for details on the academic integrity policy for problem sets.

Problems

1. Rewrite the pseudocode for the BFS algorithm studied in class (and presented in the textbook) to work for an adjacency matrix representation of the graph instead of an adjacency list representation.

2. Analyze the time complexity of your answer from the previous problem. (To receive credit, you must not only give the time complexity for this algorithm, but also have a clear and correct argument for why this complexity is correct.)

3. Let (u, v) be the edge in a weighted undirected graph with the minimum weight. Prove that there exists at least one MST of this graph that includes (u, v).

4. In class, we studied Prim's MST algorithm. This algorithm stores graph nodes in a priority queue. We analyzed the time complexity under the assumption that this queue was implemented with an efficient Fibonacci heap. This problem asks you to analyze the time complexity that results if the priority queue used by the algorithm is implemented less efficiently.

   In more detail, assume you implement with priority queue with a sorted linked list. With this implementation, to insert an element into the queue, you traverse the list from left to right (i.e., in increasing order) to find the first element with a key that is greater than or equal to the new element, and insert the new element before it in the list. If you change the key value of an element in the queue, you remove it from the underlying list, then re-insert it using the above insert procedure. Assume you can test for membership in $O(1)$ steps by keeping an auxiliary array with a bit for each node that indicates whether or not it is in the list.

   Analyze the time complexity of Prim's MST algorithm under the assumption that it uses this list-based priority queue implementation. Your analysis should touch on all the ways that the algorithm uses the queue (see the analysis of Prim's from the textbook to make sure you are not missing any way that the algorithm implicitly uses the queue).

5. Draw a connected undirected weighted graph G that contains an MST T and pair of nodes (u, v) such that the shortest path between u and v in T has a larger weight than the shortest path between u and v in G. To receive credit you must identify the MST T and pair (u, v) in G for which this property holds.

6. Assume $p = \langle u_1, u_2, ..., u_i \rangle$ is a shortest path from u_1 to u_i in some weighted graph G that contains only positive weights. Prove that p cannot contain a cycle.

   (Hint: start by assuming for the sake of contradiction that p *does* contain a cycle.)

7. Describe a flow network G and flow f such that G has 5 nodes and every edge in G_f has residual capacity 5. To receive credit you must draw both the flow network (with flow values) and its corresponding residual network (with residual capacities).

8. This question has two parts, both of which concern the time complexity analysis of the Ford-Fulkerson algorithm (with integer capacities) that we studied in class.

   (a) In the time complexity analysis, I argued that each iteration of the main *while* loop must increase the value of the flow f by at least 1. Why is this?

   (b) We proved that the algorithm we studied requires $O(|f^*| \cdot |E|)$ time, where f^* is a maximum flow in the network. Consider the following "optimization." Starting with integer capacities, we reduce each capacity by a factor of $1/k$. That is, we replace each $c(u, v)$ with $c(u, v)/k$. We then calculate the max flow g^* on this reduced-capacity graph. It is clear to see that $|g^*| = |f^*|/k$, so to get the final answer we multiply all resulting flow values by k.

   At first glance, it might seem like this optimized version of the algorithm will run in $O(|g^*| \cdot |E|) = O(|f^*|/k \cdot |E|)$ time—potentially a big improvement for large k. But this first glance analysis is wrong. Explain why this is wrong and provide a more accurate time complexity upper bound.