CS:3330 Exam 2, Spring 2018

Tuesday, April 3rd 2018, 6:30 pm to 8:30 pm

1. These problems are on graph representation and basic graph algorithms. The algorithms you design for this problem can be described in pseudocode or in plain English.

   (a) For any vertex v in an undirected graph $G = (V, E)$, define a binary function $H(v)$ as follows: $H(v)$ is set to 1 if there is a vertex with degree at least 10, that is within two hops of v; otherwise $H(v)$ is set to 0.

   Describe an algorithm that computes the entire array of $H(v)$ values for all vertices $v \in V$ in $O(n + m)$ time, assuming that the graph is given in adjacency list format. Here and in the rest of the exam assume that $n = |V|$ and $m = |E|$.

   (b) Now modify the definition of $H(v)$ as follows: $H(v)$ is set to 1 if there is a vertex with degree at least 10, that is within 20 hops of v; otherwise $H(v)$ is set to 0.

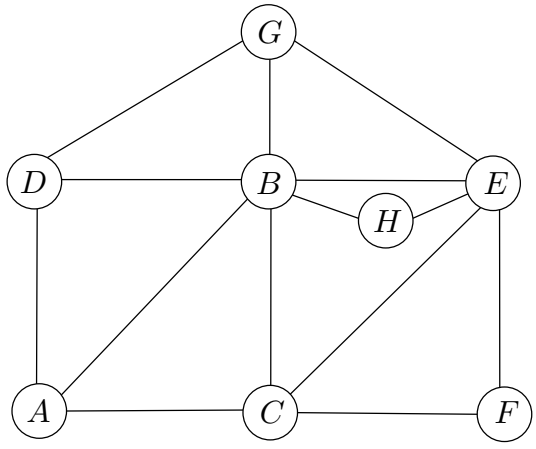
   Describe an algorithm that computes the entire array of $H(v)$ values for all vertices $v \in V$ in $O(n + m)$ time, assuming that the graph is given in adjacency list format.

2. These problems are on DFS and BFS.

   (a) Describe an algorithm running in $O(n+m)$ time, which, given an undirected graph $G = (V, E)$ and a particular edge $e = \{u, v\}$ in it, determines whether G has a cycle containing e.
   Notes: (i) Your answer can be in pseudocode or it can be in plain English; in either case a reader should be able to clearly follow the steps of your algorithm. (ii) If you use an algorithm we have already discussed (e.g., DFS or BFS), you need not write down the pseudocode for that algorithm, you can simply write something like "call DFS(G)." (iii) You don't have to present an argument or proof that your algorithm runs in linear-time or it is correct.

   (b) The Breadth-first Search (BFS) algorithm (see the pseudocode on Page 106) uses the `queue` data structure. The `queue` data structure has the property that whenever we pull out an element from it, we get the element that was inserted earliest into the data structure. Let us now replace the `queue` data structure in BFS by a data structure that we will call `wierdQueue`, which has the property that instead of returning the earliest element, it always returns the second earliest element. For example, if we insert elements A, B, C, and D into a `wierdQueue` data structure in that order and then pull out an element, we will get B out of the data structure (not A). Of course, if a `wierdQueue` data structure contains just one element, then that is what is returned when we pull an element from the data structure.

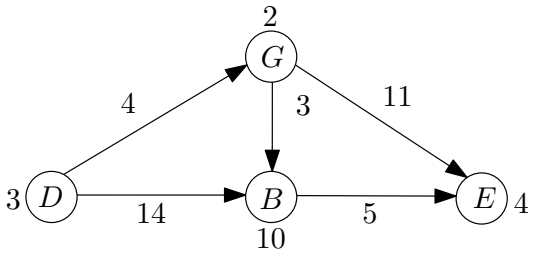
   Show the execution of this modified BFS algorithm on the following graph, using vertex A as the source. Specifically, show the contents of the `wierdQueue` data structure at the start of *each* iteration and show the BFS tree that is produced. You can assume that whenever neighbors of a vertex are considered, they are considered in alphabetical order.



3. This problem is on Dijkstra's algorithm and the Bellman-Ford algorithm.

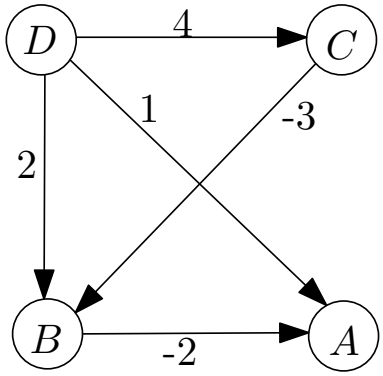
   (a) A natural generalization of the Shortest Path problem is obtained by allowing vertices to have costs, in addition to edges having lengths. So each edge $\{u, v\}$ has a *length* $\ell(u, v)$ and each vertex v has a cost $c(v)$. We can now define the *cost* of a path as the sum of lengths of edges in the path *plus* the costs of all vertices on the path.

   For example, in the directed graph shown below, there are three paths $D \to G \to E$, $D \to B \to E$, and $D \to G \to B \to E$ from source vertex D to vertex E. Path $D \to G \to E$ has cost $3 + 4 + 2 + 11 + 4 = 24$, path $D \to B \to E$ has cost $3 + 14 + 10 + 5 + 4 = 36$ and path $D \to G \to B \to E$ has cost $3+4+2+3+10+5+4 = 31$ and therefore the path $D \to G \to E$ is the cheapest path from D to E.



   Modify Dijkstra's algorithm (see Page 110) to solve the Single Source Shortest Path problem in this more general setting. You may assume that all edge lengths and all vertex costs are non-negative. Your answer should be in the form of pseudocode obtained by modifying the pseudocode for Dijkstra's algorithm on Page 110.

   (b) Consider the following directed, edge-weighted graph with some of the edges having negative edge weights. (i) Using vertex D as the source, hand execute the Bellman-Ford algorithm (Page 118) on this graph and show the values of the `dist` array after each iteration of the outer loop. Process the edges in the inner loop in alphabetical order, i.e., in the order (B, A), (C, B), (D, A), (D, B), (D, C). (ii) If the edges are processed in a different order then it is possible to compute the `dist` values correctly in just one iteration of the outer loop. Write down this order of the edges.



4. This problem is on Greedy algorithms.

   (a) A *dominating set* in a graph $G = (V, E)$ is a subset $D \subseteq V$ of vertices such that every vertex in V is either in D or has a neighbor in D. A well known optimization problem is the *minimum dominating set* problem in which we are given a graph and asked to find a dominating set with *fewest* vertices. For the graph in Problem 2, the vertex set $\{B, F\}$ is a dominating set because every one of the 8 vertices in this graph is either in the set $\{B, F\}$ or has a neighbor in this set. Furthermore, this is a minimum dominating set because there is no dominating set of size 1.

There is a simple, natural *greedy* algorithm for this problem. To describe it, we need some terminology. Any vertex in the dominating set is called a *dominator* and a vertex that is not a dominator, but has a neighbor who is a dominator is said to be *dominated*.

> **Greedy Algorithm:** repeatedly pick a vertex v and add it to the solution if v dominates the most, not-yet-dominated vertices. (Note: a vertex dominates itself and ties are broken arbitrarily.)

The intuition behind this algorithm is that if each vertex in the dominating set dominates lots of other vertices, we might be able to construct a small dominating set. As you might expect, this greedy algorithm does not always return a minimum dominating set. Construct a counterexample to the correctness of this algorithm, i.e., present an input graph for which the greedy algorithm returns a dominating set whose size is strictly larger than the size of the minimum dominating set. For your example, clearly specify (i) the dominating set returned by the greedy algorithm and (ii) a minimum dominating set.

(b) In the "Practice Problem Set" on greedy algorithms, we discussed a problem called *Weighted Interval Scheduling* problem. Here is yet another attempt at a greedy algorithm for this problem.

Define the *degree* of an interval I as the number of other intervals that I overlaps.

> **Greedy Algorithm:** repeatedly pick an interval I with maximum value of the ratio $\frac{w(I)}{degree(I)}$ and then delete I and all intervals that overlap I,

The intuition for this algorithm is that we want to pick an interval with a high weight that rules out only a small number of other intervals.

Either construct a counterexample to show that this greedy algorithm does not always produce an optimal solution to the Weighted Interval Scheduling problem or show that this greedy algorithm is correct by presenting a proof of correctness, similar to the one we saw for the Interval Scheduling problem.
