CMSC 27200 - Winter 2021 — Andrew Drucker, Lorenzo Orecchia

due tuesday, february 2 at 12 noon via gradescope

H3.

- The assignment is due at Gradescope on Tuesday, February 2 at 12 noon.
- You can either type your homework using LaTex or scan your handwritten work. We will provide a LaTex template for each homework. If you writing by hand, please fill in the solutions in this template, inserting additional sheets as necessary. This will facilitate the grading.
- You are permitted to discuss the problems with up to 2 other students in the class (per problem); however, *you must write up your own solutions, in your own words*. Do not submit anything you cannot explain. If you do collaborate with any of the other students on any problem, please do list all your collaborators in your submission for each problem.
- Similarly, please list any other source you have used for each problem, including other textbooks or websites.
- *Show your work.* Answers without justification will be given little credit.

PROBLEM 1 (25 POINTS) *Give a polynomial-time algorithm that solves the following problem. Give a clear proof that your algorithm is correct and runs in polynomial time.*

Input: *a simple (no repeated edges or self-loops), connected, undirected graph $G = (V, E)$ with a positive edge-weight function $w : E \to \mathbb{N}^+$, with all edge weights distinct;*

Output: *a spanning tree $T \subseteq E$ of G, of second-smallest total weight.*

More precisely: T should have minimum weight among all spanning trees of G that are not *Minimum Spanning Trees for (G, w). If there are two or more such trees, you are free to output any one of them, and you don't need to decide whether such a tie can or does occur.*

Solution: Your solution goes here.

Lewis Arnsten, Collaborator: Brian McManus
Professor Drucker granted me a 24-hour extension for this assignment

Problem 1:

The goal of this algorithm is to find and modify the spanning tree of smallest total weight by removing its edge of largest weight and adding a new edge that minimizes the absolute value of the difference between the edge weight lost and the edge weight added.

Step-by-step algorithm: Find the minimum spanning tree using Kruskal's algorithm. Find an edge not already in the minimum spanning tree and add it to the tree. Then find the edge of maximum weight, excluding the edge that was just added. Remove this edge and compute the absolute value of the difference between the weight of the removed edge and the weight of the newly added edge. Save the new tree if this is the lowest computed weight difference so far. Repeat these steps for every edge not in the minimum spanning tree and return the spanning tree with the smallest weight difference.

T = minimum spanning tree with Kruskal's algorithm
lowest _weight = maximum weight of edge in *T* (arbitrary)
For all edges *x* not in *T*:
    Add *x* to *T*, creating cycle *C*
    Find edge *y* in *C*, where *y* has maximum weight and *y* != *x*
    Remove edge *y* from *C*, creating tree *T′*
    Compute *W* = |*w(y)* - *w(x)*|, where *w(x)* = weight of edge *x*
    If *W* < lowest _weight:
        lowest _weight = *W*
        Final_Tree = *T′*

Return Final_Tree //spanning tree with smallest weight difference

Proving correctness: The spanning tree of second-smallest total weight differs from the minimum spanning tree by one edge. Thus, by replacing an edge in the minimum spanning tree with an edge not in the minimum spanning tree such that the weight difference is as small as possible we achieve the spanning tree of second-smallest total weight.

Polynomial time: This algorithm consists of two parts, Kruskal's algorithm for finding the minimum spanning tree and a loop for finding the spanning tree of second-smallest total weight. In both cases the time complexity is O(E^2). Thus, this algorithm is O(E^2).

Problem 2:

The bottleneck rate of a path P is found using the minimum bandwidth of any edge it contains. The best achievable bottleneck rate for a pair u, v is the path P in G with the maximum bottleneck rate. If we set the weights of the edges of G to the negative of their bandwidth when finding the minimum spanning tree, the edge with minimum bandwidth will become the edge with maximum weight and vice versa. Thus, calculating the minimum spanning tree under this condition will produce a spanning tree such that, for every pair of nodes, the path P between them achieves the maximum bottleneck rate (highest possible minimum bandwidth for edges in P).

Algorithm:
We can find a tree T in G such that for every pair of nodes u, v, the unique u-v path P in T attains the best achievable bottleneck rate. To do so, calculate the minimum spanning tree with each edge weight set equal to the negative of its bandwidth. This can be done with Kruskal or Prim's algorithm.

Proving correctness: Proof by contradiction
Assume there is a pair u, v with a path P' that has a bottleneck rate higher than the u-v path P in our minimum spanning tree. Say $x = (a,b)$ is an edge of minimum bandwidth on P. By assumption, x has the smallest bandwidth of any edge in P or P'. We can use the edges in both P and P', other than x, to move from a to b (call this path P''). This means that there is a cycle C in which x has the minimum bandwidth. This is a contradiction, as x belongs to the minimum spanning tree but has a lower bottleneck rate than P'' (x has the maximum cost in C).

Polynomial time: Since we are using Kruskal's algorithm directly, with only a change in the weights used, the algorithm is O(n^2) (the same as Kruskal's).

PROBLEM 3 (25 POINTS) *Let $G(V, E)$ be an undirected and **unweighted** graph with n nodes. Let $T_1, T_2, \ldots, T_k$ be $k = n - 1$ distinct spanning trees of G. Devise a polynomial-time algorithm that finds a spanning tree $T = (V, E_T)$ in G that contains at least one edge from each T_i. (Prove correctness and polynomial runtime.)*

Definition: two trees (or for that matter any graphs) on a set of nodes are **distinct**, if they differ in at least one edge.

Solution: Your solution goes here.

Problem 3:

The goal of this algorithm is to build a new spanning tree $T = (V, E_t)$ in G using edges form each tree T_k in $T_1,....,T_k$.

Algorithm:
Initialize a list of nodes S that are in our spanning tree T containing a starting node s_1
Initialize a list of nodes N that are not in our spanning tree T
Initialize a spanning tree with s_1

For each spanning tree T_k in $T_1,....,T_k$:
    For each edge in T_k:
        If the edge has one node in S and the other in N:
            Add this edge to our spanning tree T
            Remove the end of the edge from N and add it to S

Proving correctness: S and N form a partition of G, meaning a spanning tree T_k in $T_1,....,T_k$ connects any two points x in S and y in N through a path P containing an edge with a vertex in S and another in N. Thus, there exists an edge in T_k that connects some s in S to some n in N. This algorithm will never create cycles because it ensures (using S and N) that every new edge contains one node that has not yet been connected to the tree. This algorithm will thus produce a spanning tree with n-1 nodes containing an edge from each T_k, as we choose one edge from each T_k to build our spanning tree.

Polynomial time: The outer loop will run for k iterations as it loops through $T_1,....,T_k$. The inner loop will also run for k iterations as there could be k edges in T_k, and it could take a maximum of k tries to check if each edge has a node in N and S. Thus, this algorithm is O(k^3).

PROBLEM 4 (25 POINTS) *Let $G = (V, E, \{w_e\}_{e \in E})$ be an undirected graph with positive edge weights w_e indicating the lengths of the edges. Devise a polynomial-time algorithm that, given a vertex $i \in V$, computes the length of the shortest cycle containing vertex i in G. Give a proof of correctness and a running-time analysis for your algorithm. For full credit, your algorithm should run in time $O(|V|^2)$.*

Problem 4:

To calculate the shortest cycle containing a vertex *i* in an undirected graph *G,* we will first find the set of vertices *V* connected to *i* by an edge. For each of the vertices v_i in *V*, we will use Dijkstra's algorithm to calculate the shortest path from v_i to *i*, excluding the edge between those two points. For each of these paths, the total distance of the cycle is calculated by adding the length of the excluded edge. This total distance is stored in a set *D.* The minimum of these distances, or the minimum of *D*, is the length of the shortest cycle containing *i* in *G*.

Algorithm:
initialize $V = \{v_1,.....,v_n\}$ where *n* is the number of vertices connected to vertex *i* by an edge
Initialize D = {}

For every vertex in *V:*

l = length of edge (v_i, *i*)
G′ = create a copy of *G* without edge (v_i, *i*)
d = run Dijkstra's algorithm to find the shortest distance from v_i to *i* in G′
Add value (d + l) to set D

Return minimum of set D

Proving correctness: This algorithm finds the shortest path from each of the vertices surrounding the starting vertex *i* to *i*. Since it has been proved that Dijkstra's algorithm calculates the shortest path between two vertices, we know that we have found the shortest paths from each of the vertices surrounding the starting vertex *i* to *i*. By adding the edges between our start and end points we have all the possible shortest cycles containing vertex *i*. Thus, by taking the shortest of these cycles we ensure that we will return the shortest cycle containing *i*.

Runtime analysis: The time complexity for this algorithm is determined by the number of edges connected to vertex *i*. In a graph *G* with n nodes, this algorithm has a worst possible runtime of (n-1)*(runtime of Dijkstra's algorithm) so we know it is at least O(V^3).