CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2021**

Midterm Exam #2

Due: Tuesday, April 13, 9:00am EST

Name: *Ryan Coslove* *NetID:* *rmc326*

Instructions

1. Do not forget to write your name and NetID above, and to sign Rutgers honor pledge below.
2. The exam contains 4 problems worth 100 points in total *plus* one extra credit problem worth 10 points.
3. This is a take-home exam. You have exactly 24 hours to finish the exam.
4. The exam should be done **individually** and you are not allowed to discuss these questions with anyone else. This includes asking any questions or clarifications regarding the exam from other students or posting them publicly on Piazza (any inquiry should be posted privately on Piazza). You may however consult all the materials used in this course (video lectures, notes, textbook, etc.) while writing your solution, but **no other resources are allowed**.
5. Remember that you can leave a problem (or parts of it) entirely blank and receive 25% of the grade for that problem (or part). However, this should not discourage you from attempting a problem if you think you know how to approach it as you will receive partial credit more than 25% if you are on the right track. But keep in mind that if you simply do not know the answer, writing a very wrong answer may lead to 0% credit.

   The only **exception** to this rule is the extra credit problem: you do not get any credit for leaving the extra credit problem blank, and it is harder to get partial credit on that problem.
6. **You should always prove the correctness of your algorithm and analyze its runtime.** Also, as a general rule, avoid using complicated pseudo-code and instead explain your algorithm in English.
7. You may use any algorithm presented in the class or homeworks as a building block for your solutions.

Rutgers honor pledge:

On my honor, I have neither received nor given any unauthorized assistance on this examination.

Signature: Ryan Coslove

Problem. #	Points	Score
1	25	
2	25	
3	25	
4	25	
5	+10	
Total	100 + 10	

Problem 1.

(a) Suppose $G = (V, E)$ is any undirected graph and $(S, V - S)$ is a cut with zero cut edges in G. Prove that if we pick two arbitrary vertices $u \in S$ and $v \in V \setminus S$, and add a new edge (u, v), in the resulting graph, there is no cycle that contains the edge (u, v). **(12.5 points)**

Solution.

Given G has zero cut edges, we can add a cut edge (u, v) as $u \exists S$ and $v \exists V - S$ to G, creating $G + e$. $G + e$ will not have a cycle. If there is a cycle, it means there is another path between endpoints u and v of e that does not use e. That path belongs to G then. At least one edge of this path should cross the cut $(S, V - S)$. So there's already a cut edge in $(S, V - S)$, a contradiction.

(b) Suppose $G = (V, E)$ is an undirected graph with weight w_e on each edge e. Prove that if the weight of some edge f is *strictly larger* than weight of all other edges in *some cycle* in G, then *no* minimum spanning tree (MST) of G contains the edge f.

(Note that to prove this statement, it is *not* enough to say that some specific algorithm for MST never picks this edge f; you have to prove *no* MST of G can contain this edge). **(12.5 points)**

Solution.

Fix any arbitrary MST T of G and suppose by contradiction that f is a part of T. Consider the subgraph $T - f$ obtained by removing the edge f from T. Since T is a tree, this subgraph will not have a cycle containing edge f. Let e be the heaviest weight of this cycle and $w_f > w_e$.

Noe consider the subgraph $T - f + e$. This subgraph has $n - 1$ edges and is connected, as $T - f$ was connected and we added an edge to the cycle of this subgraph (which doesn't change connectivity). As such, $T - f + e$ is a spanning tree. But since $w_f > w_e$, weight of the spanning tree is strictly larger than the weight of T, a contradiction with T being a MST of G that contains edge f.

Problem 2. We are given a directed acyclic graph $G = (V, E)$ with a unique source s and a unique sink t. We say that an edge e in G is a *bottleneck edge* if *every* path from s to t passes through this edge e. Design and analyze an algorithm for finding *all* bottleneck edges in G in $O(n + m)$ time.

(a) *Algorithm (or graph reduction)*: **(10 points)**

Solution.

Let f(u) be the number of ways once can trade from node u to destination vertex. We want to use topological sorting to order the vertices $u_1, ..., u_n$ such that any edge can only be directed from a vertex with a lower index to a vertex with a higher index. We are looking to find outgoing edges with in-degree value 0 that lead to value t

- Let D[1:t] be an array initilized to 0 and O be an empty linked-list for the output ordering. Set a second array C[1:n] that will contain all existing topoligcal sorts. Set a third array E[1:n] that will contain all existing edges.
- Go over all vertices v and for any $u \exists N(v)$, increase D[u] by one. (At the end of this step, D[v] denotes the in-degree of v for all $v \exists V$).
- Insert every vertex v with $D[v] = 0$ into a queue Q.
- While Q is not empty:
  (a) Let v be the first vertex of Q and dequeue (remove) this vertex from Q.
  (b) Add v to the end of the linked-list O.
  (c) For $u \exists N(v)$: Reduce D[u] by one. If D[u] = 0 insert u to Q
- If $|O| < t$, output as not a DAG path from s to t; otherwise output O as a topological ordering path from s to t, place this sorted linked list into C[ ], and increase count by +1.
- Also for every edge in the sort (u, v), place that edge into E[ ], increase count for that edge by +1.
- Run the while loop again, check if sorted path exists in C. If not, add it to C. Also check if edge exists in E. If not, add it to E. Once C[i] = C[i+1], aka no more distinct topological sorts, check count.
- If count for C = count for any edge in E, we have a bottleneck edge that exists. Return all edges in E[ ] whose count is equal to the number of paths in G.

(b) *Proof of Correctness*: **(10 points)**

Solution.

Suppose first that $|O| = n$: we prove O is a topological ordering from s to t in G in this case:

- Consider any edge (u, v) in G. For the vertex v to be added to O, it should first join the queue Q; forthis to happen, we should have D[v] = 0 at some point.
- For vertex v to have $D[v] = 0$, we should have 'removed' edge (u, v) first: in other words, as long as u is not dequeued, and so we reduce D[v] by one, $D[v] > 0$ and so v will not join Q. (Here, we are also using the fact that at the beginning D[v] was equal to in-degree of v and we only reduce D[v] by one whenever we add one of the in-neighbors of v to Q).
- This implies that u should have joined O before v, thus appearing before v in the ordering. This implies that if $|O| = n$ (and hence all vertices appear in the ordering), O would be a topological ordering of s to t in G.
- Also, we are checking for duplicate sorted paths as we iterate. If we get a duplicate path, we have found the number of available paths from s to t. We have kept count of each path. We also are checking for duplicate edges in those paths, keeping a count for each individual edge (u, v). If any edge has the same count as the number of paths, that means that edge exists in all paths and therefore is a bottleneck edge.
- E.g. If there are 4 paths in G making count = 4, and the edge (3,4) exists in all 4 paths making the edge's count = 4, then we know edge (3,4) is a bottleneck edge.

(c) *Runtime Analysis*: **(5 points)**

Solution. The first and third line before the for-loop takes $O(n)$ and the second line takes $O(n + m)$ (we go over each edge only once). Also, the while-loop takes at most $O(n + m)$ time because we consider each vertex at most once in the while-loop and when considering each vertex, we go over its out-degree; since sum of the out-degrees is $O(m)$, we obtain the $O(n + m)$ bound.

Problem 3. Crazy City consists of n houses and m bidirectional streets connecting these houses together, and there is always at least one way to go from any house to another one following these streets. For every street e in this city, the cost of maintaining this street is some positive integer c_e. The mayor of Crazy City has come up with a brilliant cost saving plan: destroy(!) as many as the streets possible to maximize the cost of destroyed streets (so we no longer have to pay for their maintenance) while only ensuring that there is still a way for every house to reach mayor's house following the remaining streets.

Design an $O(m \log m)$ time algorithm that outputs the set of streets with *maximum total cost* that should be destroyed by the mayor.

(a) *Algorithm (or graph reduction)*: **(10 points)**

Solution.

We will use Prim's algorithm for finding an MST, also using amin-heap.

The algorithm:

(a) Let $mark[1:n] = FALSE$ and s be the Mayor's house in the graph.

(b) Let $F = \emptyset$ be the maintained forest and set $mark[s] = TRUE$

(c) Let H be a min-heap data structure: Call $H.preprocess$ and $H.add(e)$ for every edge e incident on s.

(d) While $H.size > 0$:

- Let $e = \{u, v\} = H.extract - min$
- If $mark[u] = mark[v] = TRUE$ ignore this edge and go to the next iteration of the while-loop.
- Otherwise, without loss of generality, assume $mark[v] = FALSE$ (if $mark[u] = FALSE$ simply switch the name of u, v below).
- Set $mark[v] = TRUE$, call $H.add(e)$ all edges e incident on v to S, and add $\{u, v\}$ to F.

(e) Return G' = G - F

(b) *Proof of Correctness*: **(10 points)**

Solution.

The proof of correctness for Prim's algorithm is done in lecture 9. The additional information needed to prove that we are returning the answer we are looking for is the last step of returning G'. We have our initial graph of G, containing all roads pre-demolition. We want all the houses to have a path to the mayor's house, so we make his house starting vertex s. We then use prim's algorithm to find the MST that finds the fewest roads possible necessary for all houses to reach the mayor's house, which also finds the lowest cost possible to maintain the streets c_e. This MST is represented as F. So to find the maximum total cost that should be destroyed by the mayor, we take the full graph G and subtract F, so we have all the roads that don't exist in the MST that give us the lowest cost. G' leaves us with all the roads that should be destroyed and thus the meximum total cost.

(c) *Runtime Analysis*: **(5 points)**

Solution. The runtime of this algorithm is now $O(\log m)$ by each iteration of the while-loop to extract the minimum and $O(deg(v) * \log m)$ for each vertex v to insert all edges incident to v in the heap (note that size of the min-heap never gets more than $O(m)$). It would take $O(1)$ time to output G'. This means that the total runtime is $O(m \log m)$ as desired.

Problem 4. We are given a weighted undirected graph $G = (V, E)$ with positive weight w_e over each edge e, and two vertices $s, t \in V$. Additionally, we are given a list L of k *new* edges (not in G), where each edge $f \in L$ has some weight w_f. Our goal is to pick *exactly one* edge f from L to add to G such that we minimize the weight of the shortest path from s to t in the resulting graph $G + f$.

Design an $O(k + n + m \log m)$ time algorithm that determines which edge from L should be added to G.

(a) *Algorithm (or graph reduction)*: **(10 points)**

Solution.

We will use Dijkstra's algorithm:

(a) Let $mark[1:n] = FALSE$ and s be the designated source vertex.

(b) Let $d[1:n] = +\infty$ and set $d[s] = 0$

(c) Let H be a min-heap data structure: Call $H.preproccess$ and $H.add(e)$ for every edge e incident on s and assign a value $value(e) - d[s] + w_e$ to each of these edges.

(d) While $H.size > 0$:

- Let $e = (u, v) = H.extract - min$
- If $mark[v] = TRUE$ ignore this edge and go to the next iteration of the while-loop.
- Otherwise, set $marl[v] = TRUE$, call $H.add(e)$, and insert all edges e' incident on v to S with $value(e') = d[v] + w'_e$.

(e) Run a for loop through L, iterating through each k edge, searching for the smallest w_f in L.

(f) Return d and f.

(b) *Proof of Correctness*: **(10 points)**

Solution.

The proof of correctness for Dijkstra's algorithm is provided in Lecture 10. We are looking for the smallest weight of the shortest path from s to t in the resulting graph $G + f$. We know returning d successfully gives us the shortest path from s to t and therefore will return the minimal weight w_e of G. By running a for-loop comparing all of the w_f values in L, we can find the smallest (minimal) value we can return f, the edge we will add to G. So returning d and f will result in us returning $G + f$ which will have the shortest path and minimal weight $w_{min} = w_e + w_f$.

(c) *Runtime Analysis*: **(5 points)**

Solution.

We implemented the set S with a min-heap: this allows us to use Dijkstra's algorithm in $O(n+m \log m)$ time. Running the for-loop through L will run at $O(k)$ time. Finally, we have a total run time of $O(k + n + m \log m)$ as desired.

Problem 5. [**Extra credit**] You are given an undirected graph $G = (V, E)$ and two vertices s and t in G. Design and analyze an algorithm that in $O(n + m)$ time, decides if the number of *different shortest paths* from s to t is an *odd* number or an *even* one. **(+10 points)**

Hint: This problem is both related to Problem 3 of your homework 3 and also very different: here, you are looking for the number of *shortest* paths (not arbitrary paths) and in an undirected graph (not in a DAG). You should however still feel free to use a graph reduction to that problem if you see a proper way.

Solution. Solution to Problem 5, the extra credit problem, goes here.

Extra Workspace