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

Midterm Exam #2

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

Name: Letao Zhang *NetID: 186004459*

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: Letao Zhang

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. We prove that in the original graph $G = (V, E)$, there is a path from s to t. Since G is acyclic, we never visit a vertex twice. This implies that by adding the new edge(u, v), there will be no cycle in G. Since T is a tree, this subgraph will have one cycle containing the edge f. Recall that a spanning tree of a connected subgraph G is any subgraph of G which is a tree – a tree itself is a connected subgraph which has no cycle. An undirected graph is acyclic if a DFS yields no back edges. Since back edges are those edges(u, v) connecting a vertex u to an ancestor v in a depth-first tree, so no back edges means there are only tree edges, so there is no cycle.

(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 not part of T. Consider the subgraph $T + f$ obtained by adding the edge f to T. Since T is a tree, this subgraph will have one cycle containing the edge f. Let e be the smallest weight of this cycle and note that by the promise of the problem, $w_f < w_e$. Now consider the subgraph $T + f - e$. This subgraph has $n-1$ edges and is connected, as $T + f$ was connected and we removed an edge from the cycle of this subgraph (which does not change connectivity). As such, $T + f - e$ is a spanning tree. But since $w_f < w_e$, the weight of this spanning tree is strictly larger than the weight of T, a contradiction with T being an MST of G that does contain the 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. We simply can use Depth-First Search(DFS) to solve this problem. Cause we need to go through all the nodes in order to find out whether every path from s to t passes through this edge e.

Initialize an arraymark[1 : n] with 'FALSE'. Run the following recursive algorithm ons, i.e., return $DFS(s)$: $DFS(v)$: 1) If mark[v] = 'TRUE' terminate. 2) Otherwise, set mark[v] ='TRUE' and for every neighbor $u \in N(v)$ (this is the list of neighbors ofvthat we have direct access to in the adjacency list representation): •Run $DFS(u)$. Then we return all vertices v with mark[v] = 'TRUE' as the answer, i.e., as vertices that are in thesame connected component of s. 3) Adding extra flag, if the path passes through the edge e returns true, otherwise returns false.

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

Solution. We need to prove that every vertex connected to s will be output and no other vertex is also output by this algorithm. The second part is straight forward because we are only visiting s,neighbors of s, their neighbors and so on and so forth, so for any vertex marked, there is a path fromstothat vertex in the graph.The other part is also easy: consider any unmarked vertex u. We prove that u is not in the connected component of s. Since u is unmarked, none of the neighbors v of u can be marked: otherwise, when we visited v for the first time, we would have marked u also at some point later (because we runDFS(u) for thatvertex when visitingv, i.e., in DFS(v)). We can then continue like this to all neighbors of v (since they arenot marked either) and say that all neighbors of v needs to be unmarked also. We expand this until we findall of the vertices that are connected touand we know that they are all unmarked. But this means thatswas not any of those vertices sincesis actually marked; henceucannot be part of the connected componentofssince there is no path from s to u.

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

Solution. We are finding all bottleneck edges while performing the DFS. We visit each vertex v at most once (after that it is marked and we do not spend more time in visiting the vertex (think of memoization)) and it takes $O(|N(v)|)$ time to process this vertex. Hence, the total runtime is at most $c\sum_{v\in V}|N(v)|$ for some constant c: since the total degree of vertices is proportional to the number of edges, the runtime of this algorithm is $O(n+m)$ as desired.

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. The set of streets with maximum total cost that should be destroyed by the mayor, is same as minimum total cost of non-destroyed streets. Therefore, the maximum total cost of destroying should be the total cost of all streets minus the total cost of non-destroyed streets. Let us construct a weighted undirected graph G= (V,E) as follows. The vertex set V is the set of all houses in Crazy City. The edge set E is the set of all bidirectional streets in Crazy City. The weight of any edge e from u to v is the cost of maintaining the street from u to v. We find the minimum spanning tree T of G with Kruskal's algorithm.

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

Solution. To make the Crazy City connected we need to pick at least a spanning tree of the street to go from any house to another one following these streets. Among all spanning trees, MST will have the minimum cost by definition, thus proving the correctness of the algorithm.

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

Solution. We create the graph G= (V,E) in $O(n+m)$ time and it has vertices and edges. The runtime of Kruskal's algorithm on this graph is $O(mlogm)$. Since $m \geq n-1$ (otherwise, we can never make the city connected), we have that the runtime is $O(mlogm)$.

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 simply use Dijkstra's algorithm to solve this problem. 1) Let mark [1:n] = FALSE and s be the designated source vertex. 2) Let $d[1 : n] = +\infty$ and set d[s] = 0. 3) Set mark[s] =TRUE and let S(initially) be the set of edges incident on s and assign a value value(e) =d[s]+w_e to each of these edges. 4) While S is non-empty: (a) Let e=(u, v) be the minimum value edge in S and remove e from S. (b) If mark[v] =TRUE ignore this edge and go to the next iteration of the while-loop. (c) Otherwise, set mark[v] =TRUE, d[v] =value(e), and insert all edges e' incident on v to S with $value(e') = d[v] + w'_e$. 5) Return d.

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

Solution. The proof of correctness of Dijkstra's algorithm is quite similar to that of BFS and Prim's algorithm. We prove by induction that in every iteration of the while-loop in the algorithm. The base case of the induction, namely for iteration 0 of the while-loop (i.e., before we even start the while-loop) is true sincesis the closest vertex to s(satisfying part one) and d[s] =dist(s,s) = 0 satisfying part two. Now suppose this is true for some iteration i of the while-loop and we prove it for iteration i+ 1. Let e= (u,v) be the edge removed from S in this iteration. If mark[v] =TRUE we simply ignore this edgeand hence the set C remains the same after this step and by induction hypothesis, we minimize the weight of the shortest path from s to t in the resulting graph $G + f$.

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

Solution. Pick exactly one edge f from L to add to G takes k times. Then we implement the set S with a min-heap: this allows us to implement Dijkstra's algorithm in $O(n + m \log m)$ time as well. Thus, the total runtime of this problem should be $O(k + n + m \log m)$ 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. *Algorithm.* We can simply use topological ordering to solve this problem. The algorithmis simply as follows: we pick a vertex with in-degree equal to 0 and place it in the beginning of the ordering; remove all its outgoing edges from the graph, and repeat. Also if at any point we can no longer find a vertex of in-degree 0, we return that if the number of different shortest paths from s to t is an odd number or an even one.

Proof of Correctness. Suppose first that $|O| = n$: we prove O is a topological ordering of G in this case: (1) 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. (2) 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. (3) This implies that u should have joined O before v, thus appearing before v in the ordering. This implies that if $|O| = n$, O would be a topological ordering of G.

*Runtime Analysis.*The runtime of this algorithm isO(n+m): The first and third line before the for-looptakesO(n) and the second line takesO(n+m) (we go over each edge only once). Also, the while-loop takesat mostO(n+m) time because we consider each vertex at most once in the while-loop and when consideringeach vertex, we go over its out-degree; since sum of the out-degrees isO(m), we obtain theO(n+m) bound.

Extra Workspace