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

Midterm Exam #2 Solutions

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. Suppose towards a contradiction that there is a cycle C that contains the edge (u, v). This means that there is a path P from u to v that does not use the edge (u, v) in this cycle. Since this path starts at $u \in S$ and ends in $v \in V \setminus S$, there should be an edge e in this path that goes from S to $V \setminus S$. This means that the cut $(S, V \setminus S)$ has at least one cut edge, 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. Suppose towards a contradiction that there is some MST T that contains the edge f. Consider the graph $T - f$ which contains two connected components S_1 and S_2 (we removed an edge from a tree). Since f is the heaviest edge of some cycle in G, there is an edge e that also connects C_1 and C_2 and $w_e < w_f$. Now consider the graph $T - f + e$ which we claim is a tree: it has $n - 1$ edges and it is connected because we connected S_1 and S_2 via the edge e again. So $T - f + e$ is a spanning tree with weight strictly less than T, contradicting the fact that T was a MST. So no MST of G can 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. The idea behind the algorithm is the following: consider a topological ordering of G; informally speaking, we claim that an edge e is a bottleneck if no other edges of G "go over" this edge in the ordering. The above algorithm then tries to find all such edges and the proof of correctness formalizes our claim.

We give the following algorithm for finding all bottleneck edges in G:

- Run topological sort on G and let $order(v)$ denote the order of vertex v in the topological ordering.
- Define $reach(v)$ to be the maximum value of $order(w)$ taken over all edges (u, w) for vertices u with $order(u) < order(v)$. To compute $reach(v)$:
  - Iterate over vertices of G in the ordering, for each vertex v_i at the i-th position of the ordering (i.e., $order(v_i) = i$), let $reach(v_i) = \max\{reach(v_{i-1}, order(z)\}$ for $z \in N(v_i)$.
- For any vertex v, if v only has a single outgoing edge e, and $reach(v) \leq order(v)$, then output e as a bottleneck edge in G.

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

Solution. The proof of correctness is based on the following claim.

Claim: An edge $e \in N^+(v)$ (here $N^+(v)$ denotes the outgoing edges of v) is a bottleneck edge if and only if out-degree of v is one and there is no edge (u, w) in G with $order(u) < order(v) < order(w)$.

Proof of the claim:

- If $e \in N^+(v)$ is a bottleneck edge in G then out-degree of v is one and there is no edge (u, w) with $order(u) < order(v) < order(w)$.

  Suppose this is not true. Then either out-degree of v is more than one or there is an edge (u, w) with $order(u) < order(v) < order(w)$.

  If out-degree of v is more than one, then there is a path starting from s to v that takes this other edge, and then from its endpoint goes to t (here, we use the fact that s is the only source and so all vertices are reachable from s and t is the only sink so all vertices reach t). This means there is an s-t path that does not use the edge e, a contradiction with e being bottleneck.

  If there is an edge (u, w) with $order(u) < order(v) < order(w)$, then there is a s-t path that goes from s to u, takes the edge (u, w) and then goes from w to t; this path also does not use the edge e contradicting e being a bottleneck edge.

- If $e \in N^+(v)$, out-degree of v is one and there is no edge (u, w) with $order(u) < order(v) < order(w)$, then e is a bottleneck edge.

  Suppose towards a contradiction that e is not a bottleneck edge. So there is a s-t path P that does not use the edge e. Since out-degree of v is just one, the path P cannot go to the vertex v either. But then it means that path P goes to a vertex u with $order(u) < order(v)$ and then take an edge (u, w) to go to a vertex w with $order(w) > order(v)$. This means there is an edge (u, w) with $order(u) < order(v) < order(w)$, a contradiction.

end of the proof of the claim.

We now analyze the algorithm. Firstly, it can be seen that for each vertex v,

$$reach(v) = \max \{order(w) \mid \text{there is an edge } (u, w) \text{ such that } order(u) < order(v)\}.$$

The algorithm outputs the set of edges e where e is the only outgoing edge of some vertex v and $reach(v) \leq order(v)$; by definition of $reach(v)$, this happens if and only if there is no edge (u, w) in the graph G with $order(u) < order(v) < order(w)$. The correctness of the algorithm now follows from the above claim.

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

Solution. Topological ordering takes $O(n + m)$ time. For each vertex computing $reach(v)$ takes $O(deg(v))$ time and so this step also takes $O(n + m)$ time. The final step takes $O(n)$. Hence total runtime is $O(n + m)$.

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 algorithm is as follows:

- Create a graph $G = (V, E)$ where V is the set of n vertices and E is the set of bidirectional streets. Let weight of each edge e bee equal to the cost c_e.
- Compute an MST T of the graph G using Kruskal's or Prim's algorithm.
- Output all edges of G that are not in T, i.e., $G - T$ as the solution.

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

Solution. Consider the graph G. To destroy a subset of streets while only ensuring that there is still a way for every house to reach mayor's house following the remaining streets, is equivalent to destroying as many edges in G as possible while keeping G connected (note that the only way for all houses to reach the Mayor's house is that the remaining streets are connected; in other words, all houses reaching mayor's house is equivalent to all houses reaching each other as well). This means that the remaining streets should form a spanning tree.

Next, note that *maximizing* the cost of destroyed streets is equivalent to *minimizing* the cost of *not* destroyed streets. Thus the goal is to simply pick a spanning tree with minimum cost. This is exactly the MST problem, hence the correctness of the algorithm.

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

Solution. Creating the graph G takes $O(n+m)$ time. Running Kruskal's or Prim's algorithm on this graph also takes $O(m \log m)$ time. Also, since the graph is connected to begin with, we have $m \geq n-1$ and thus $O(n+m) = O(m) = O(m \log m)$. So the total runtime is $O(m \log m)$.

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. The algorithm is as follows:

- Run Dijkstra's algorithm from s to compute $dist(s, v)$ for all vertices $v \in V$.
- Run Dijkstra's algorithm from t to compute $dist(v, t)$ for all vertices $v \in V$ (note that since G is undirected, $dist(v, t) = dist(t, v)$).
- For each edge $f = (u, v) \in L$, compute

$$value(f) = \min\{dist(s, u) + w_f + dist(v, t), dist(s, v) + w_f + dist(u, t)\};$$

  Return the edge f with minimum $value(f)$.

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

Solution. First note that if adding none of the edges $f \in L$ can reduce $dist(s,t)$, then returning any edge in L is a valid solution (because $dist(s,t)$ always remain the same). So, in the following, we focus on the case that at least adding one of the edges in L reduces $dist(s,t)$.

Consider adding any edge f to the graph. If f reduces the shortest path between s and t, then it means in the graph $G+f$, there is a shortest s-t path P that contains the edge e. This path will be either a shortest path (in G) from s to u, take the edges $f=(u,v)$, and then a shortest path from v to t, or it will a shortest path from s to v, take the edge (v,u), and then a shortest path from u to t. So the weight of this path will be $\min\{dist(s,u)+w_f+dist(v,t), dist(s,v)+w_f+dist(u,t)\}$, the same as $value(f)$ computed in the algorithm. As such, returning the edge with minimum *value* results in finding the edge that leads to the shortest possible s-t path.

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

Solution. The first two lines take $O(n+m\log m)$. The next for-loop takes $O(k)$ time, so the total runtime is $O(k+n+m\log m)$.

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:* The algorithm is as follows:

- Run BFS starting from s and let $d[v]$ for each $v \in V$ denote the distance of s to v computed by BFS.
- Create the following *directed* graph H with the same vertices V: For any edge $(u, v) \in E$, if $d[v] = d[u] + 1$, add the *directed* edge (u, v) to H. Note that H is a DAG (a topological ordering is to write the vertices in the increasing order of $d[v]$-values).
- Run the algorithm from Problem 3 of homework 3 on H to find the total number of paths from s to t in the DAG H mod 2 and return the same answer.

Proof of correctness: Any shortest s-t path P will appear in H because for every consecutive set of vertices u, v in P, $d[v] = d[u] + 1$ and thus the edge (u, v) is added to H. Conversely, any path Q from s to t in H will have length $d[t]$ because each edge of the path increases the d-values by 1 and so to reach t, the length of the path should be exactly $d[t]$.

This means the number of s-t shortest paths in G is equal to the total number of s-t paths in H. The proof now follows from the correctness of the algorithm of homework 3 used as a subroutine.

Runtime: Running BFS takes $O(n + m)$ time. Creating H takes another $O(n + m)$ time. Finally, running the subroutine on H takes $O(n + m)$ time (as proven in homework 3), so the total runtime is $O(n + m)$ as desired.

Extra Workspace