CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2019**

Homework #5 Solutions

December 8, 2019

Problem 1. Describe and analyze an algorithm that finds the *second smallest spanning tree* of a given undirected connected (weighted) graph G, that is, the spanning tree of G with smallest total weight except for the minimum spanning tree. You may assume that all edge weights are *distinct*s in the graph (recall that in this case the minimum spanning tree would be unique). **(20 points)**

Hint: First prove that the second smallest spanning tree of G is different from the MST of G in only one edge. Then use this to design your algorithm by starting from an MST of G and checking which edge should be changed to obtain the second smallest spanning tree.

Solution. *Proving the first part of the hint:*

Following the hint, we first prove that the second-smallest spanning tree (let us call it SSST for the second smallest spanning tree) and the minimum spanning tree (MST) differ in exactly one edge, namely, the SSST is obtained by removing one edge from the *unique* MST of G (since the edge weights are unique) and replacing it with some other edge. More formally:

> Let T be the unique MST of G and T' be any SSST of G. Then, there exists an edge $e \in T$ and another edge $e' \in G - T$ such that $T' = T - \{e\} \cup \{e'\}$.

Proof: Suppose by contradiction that T' is a SSST of G but it differs from T in *at least* two edges e_1, e_2 (i.e., $e_1, e_2 \in T$ are both missing from T' and instead two other edges are in T – note that T' can differ from T in even more number of edges). Consider the graph $T'' = T' \cup \{e_1\} - \{e''\}$ where e'' is the edge with maximum weight in the unique cycle obtained by adding e_1 to T'. We first claim that $e'' \neq e_1$: this is because otherwise e_1 would be the maximum weight edges of a cycle and hence cannot belong to the MST T (recall that we proved earlier in the course that there is an MST that does not contain the maximum weight edge of a cycle and since in this case, the MST is unique, e_1 cannot be the maximum weight edge of any cycle). Moreover, T'' is a *spanning tree* as it it has $n-1$ edges and is connected (we removed e'' from a cycle in $T' \cup \{e_1\}$ and hence the graph remains connected). Finally, $T'' \neq T$ also since $e_2 \in T$ but $e_2 \notin T''$ and hence T'' is not an MST (by uniqueness of MST when edge weights are unique). As such, T'' is a spanning tree of G which is *not* T and its weights is smaller than weight of T', so T'' should be a SSST instead of T', a contradiction. This proves the statement above.

Algorithm: We now use the hint to design an algorithm for this problem.

1. Find the MST T of G (say by using Kruskal's or Prim's algorithm).
2. For any $e \in T$:
   (a) Remove e from T to obtain two connected component S and $V - S$ of T.
   (b) Let e' be the edge with minimum weight other among the cut edges $(S, V - S)$ in $G - e$ (if e is the unique cut edge of $(S, V - S)$ and thus there is no e', go to the next edge of the for-loop right away).
   (c) Let $T'_e = T - \{e\} \cup \{e'\}$.
3. Return the spanning tree among the (at most) $n-1$ spanning trees T'_e for $e \in T$ with minimum weight as the answer.

Proof of Correctness: We first have that every T'_e is a spanning tree in G and no $T'_e = T$: the former is because we are removing edge e from T which disconnects the tree but then add an edge in the corresponding cut to make it connected again (and also since T'_e has $n-1$ edges), and the latter is because T'_e is different from T in at least the edge e.

Secondly, by the claim proved in the first part, there exists an edge $e \in T$ and $e' \in G - T$, where $T' = T - \{e\} \cup \{e'\}$ is a SSST of G. Consider that edge e and the spanning tree T'_e created in the algorithm for that edge. We claim that T'_e has the minimum weight among all spanning trees that are different from T in edge e. This is because any such spanning tree needs to have their difference edge from the cut $(S, V-S)$ in $G - e$ (otherwise it will not be connected), and T'_e is picking the minimum weight edge in that cut. This implies that $T'_e = T'$, namely, a SSST of G. By picking the minimum weight choice of T'_e for all $e \in T$, we are ensured to pick this tree then, finalizing the proof.

Runtime Analysis: Finding MST takes $O(m \log m)$ time. We then spend $O(n)$ time to go over each edge of T and for each edge we spend $O(m)$ time to go over all cut edges of $(S, V-S)$ in $G-e$ (by going over all edges in $G-e$ and considering the ones that belong to this cut edges), hence taking $O(nm)$ time. Thus, the total runtime of the algorithm is $O(m \log m + nm) = O(nm)$ (since $m \leq n^2$ and hence $\log m \leq 2 \log n = O(n)$).

Problem 2. You are given a weighted graph $G(V, E)$ (directed or undirected) with positive weight $w_e > 0$ on each edge $e \in E$ and two designated vertices $s, t \in V$. The goal in this problem is to find a s-t path P where the maximum weight edge of P is minimized. In other words, we define the weight of a s-t path P to be $w_{MAX}(P) = \max_{e \in P} w_e$ and our goal is to find a s-t path P with minimum $w_{MAX}(P)$ (recall that in the shortest path problem, we define weight of a path to be sum (instead of max) of the weights of edges).

Design and analyze an $O(n + m \log m)$ time algorithm for finding such a path in a given graph. **(20 points)**

Hint: In this rare instance, it may be easier to modify Dijkstra's algorithm directly, instead of doing a reduction (although the latter option is also completely possible) – just remember to prove the correctness of your modified algorithm from scratch. Another option would be to instead use the DFS algorithm combined with a clever binary search approach.

Solution (**Solution 1**)**.** The first solution is based on modifying Dijkstra's algorithm (and act more as an exercise in the proof of Dijkstra's algorithm itself).

Algorithm: The algorithm is a simple modification of Dijkstra's algorithm that turns it even more similar to the Prim's algorithm:

1. Let $mark[1:n] = FALSE$ and s be the designated source vertex.
2. Let $d[1:n] = +\infty$ and set $d[s] = 0$. Also let $parent[1:n] = NULL$.
3. Set $mark[s] = TRUE$ and let S (initially) be the set of edges incident on s and assign a value $value(e) = \max(d[s], w_e)$ 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 $value(e') = \max(d[v], w'_e)$.
5. If $mark[t] = FALSE$, return 'no s-t path'; otherwise compute the path

$$t, \quad parent[t], \quad parent[parent[t]], \quad parent[parent[parent[t]]], \quad \ldots, \quad s$$

using the parent-pointers, reverse the path, and output it as the answer (the w_{MAX} weight of this path is also the weight of the maximum weight edge in this path).

Some implementation details: (1) Similar to Dijkstra's and Prim's algorithm, we should implement the set S with a min-heap to ensure the fast running time. (2) Computing the parent-paths in the last steep can be done in a while-loop by following the parents until we hit a vertex with NULL parent which can only be s.

Proof of Correctness: We give a similar proof as Dijkstra's algorithm. Let us define $dist_{MAX}(s, v)$ to be the weight of the minimum weight path from s to v according to weights w_{MAX}, i.e.,

$$dist_{MAX}(s, v) = \min_{P} \ w_{MAX}(P),$$

where P ranges over all s-v paths. In the following proof, whenever we say 'shortest path', we mean the path with minimum w_{MAX} weight.

We prove by induction that in every iteration of the while-loop in the algorithm, the set C of all marked vertices (i.e., $C = \{v \mid mark[v] = TRUE\}$) has the following two properties:

- For any $u \in C$ and $v \in V - C$, $dist_{MAX}(s, u) \leq dist_{MAX}(s, v)$;
- For every $v \in C$, $d[v] = dist_{MAX}(s, v)$, i.e., distances are computed correctly.

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 since s is 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 edge and hence the set C remains the same after this step and by induction hypothesis, we have the above two properties. Now suppose $mark[v] = FALSE$. In this case:

1. Every edge among the cut edges of $(C, V - C)$ belong to the set S at this point (we have not removed any of those cut edges yet as otherwise C would have contained the other endpoint of that edge as well and we included all those edges when we marked their first endpoint and hence added them to C).
2. We are setting $d[v] = value(e) = \max(d[u], w_e) = \max(dist_{MAX}(s, u), w_e)$ (by induction hypothesis for u) where e is the edge with minimum $value(e)$ in S and hence minimum $value(e)$ among cut edges of $(C, V - C)$.
3. Since $dist_{MAX}(s, w) \geq dist_{MAX}(s, u)$ for all $w \in V - C$ by induction hypothesis, we know that the shortest path from s to v does not visit any of the vertices in $V - C$ (otherwise the edge leading to that vertex will have a smaller value than the edge e leading to v). Hence, by setting $d[v] = \max(dist_{MAX}(s, u), w_e)$ where edge e minimizes the right hand side of this equation, we will have that $d[v] = dist_{MAX}(s, v)$. This proves the second part of the induction hypothesis. For the first part also, notice that $value(e)$ is minimized and hence v is the "closest" vertex to s in $V - C$ and hence after adding v to C, $dist_{MAX}(s, w) \geq dist_{MAX}(s, v)$ for all $w \in V - C$. This proves the second part of the induction hypothesis.

This concludes the proof of correctness of the algorithm as at the end of the last iteration, by induction hypothesis (second part), $d[v] = dist_{MAX}(s, v)$ for all $v \in V$. This ensures that $d[t] = dist_{MAX}(s, t)$ as desired. Moreover, as proven earlier in the course (say for BFS), by following the parents in this fashion we can find the s-t path as well with w_{MAX} equal to $d[t]$.

Runtime Analysis: Using min-heaps, the runtime of our algorithm is the same as Dijkstra's and Prim's algorithms and is $O(n + m \log m)$ time.

Solution (**Solution 2**)**.** We now give another solution, this time by a reduction to single-source shortest path directly without modifying the Dijkstra's algorithm.

Algorithm: (a reduction to the single-source shortest path problem)

1. Create a new weight w'_e for each edge $e \in E$ by setting $w'_e = (n+1)^{w_e}$ where w_e is the 'old' weight of the edge e.
2. Run Dijkstra's shortest path algorithm on G with this new weights w' and return the s-t path found as the answer.

Proof of Correctness: As any other reduction, the proof of correctness boils down to showing that for every graph G, the s-t path P with minimum $w_{MAX}(P)$ under the edge weights w is exactly the same as the shortest path under the edge weights w'.

Consider any two s-t paths P_1 and P_2. We prove that:

1. If $w_{MAX}(P_1) < w_{MAX}(P_2)$ then $w'(P_1) < w'(P_2)$ (under weights w' where $w'(P_1) = \sum_{e \in P_1} w'_e$):
   If $w_{MAX}(P_1) = w_1$ and $w_{MAX}(P_2) = w_2$, then
   $$w'(P_1) = \sum_{e \in P_1} w'_e = \sum_{e \in P_1} (n+1)^{w_e} \leq (n-1) \cdot (n+1)^{w_1},$$
   because $w_e \leq w_1$ for every $e \in P_1$ and there can at most $(n-1)$ edges in the path P_1. On the other hand, $w'(P_2) \geq (n+1)^{w_2}$ and since $w_2 > w_1$, we have $w'(P_2) > w'(P_1)$.
2. If $w'(P_1) < w'(P_2)$ then $w_{MAX}(P_1) \leq w_{MAX}(P_2)$:
   If $w'(P_1) < w'(P_2)$, then the maximum weight edge in P_1 according to w_1 cannot be larger than the maximum weight edge according to w_2 by the argument in the above part (as otherwise $w'(P_1) > w'(P_2)$, a contradiction). But then this implies that $w_{MAX}(P_1) \leq w_{MAX}(P_2)$ as desired.

The above two parts then ensure that the minimum w_{MAX}-weight path in G under weights w is the same as actual shortest path in G according to weights w', concluding the proof.

Runtime Analysis: Constructing the new weights takes $O(n+m)$ time and running Dijkstra's algorithm on the new graph takes $O(n + m \log m)$ time, resulting in an $O(n + m \log m)$ time algorithm overall.

Solution (Solution 3). Finally, we give yet another algorithm for this problem using the last part of the hint by using the DFS algorithm combined with a clever binary search approach.

Algorithm:

1. Sort the edges in increasing order of their weights in an array $M[1:m]$.
2. Let $a = 1$ and $b = m$. Do the following binary search steps (lines 3,4,5):
3. If $a \geq b$, goto the last line, otherwise let $Mid = (a+b)/2$.
4. Create a graph G' from G by only picking edges in $M[1:Mid]$ in G'.
5. Run DFS on G' to see whether s can reach t or not:
   (a) If YES, set $b = Mid$ and go back to Line (3).
   (b) If NO, set $a = Mid$ and go back to Line (3).
6. Create a graph G' from G by only picking edges in $M[1:a]$ in G', run DFS on G', and return the s-t path found in this DFS as the answer to the problem.

Proof of Correctness: Let P be a minimum w_{MAX}-weight path from s to t in G and e be the edge with maximum weight w_e on this path (so $w_{MAX}(P) = w_e$). The idea behind the algorithm is to find w_e.

In each binary search step, if $w_e \leq w_{M[Mid]}$, then the path P is present in G' and hence the DFS returns YES. On the other hand, if $w_e > w_{M[mid]}$, then there is no s-t path in G' as otherwise that path will have a smaller w_{MAX} value, contradicting the optimality of P; this means that in this case the DFS returns NO.

As such, in the binary search approach, we always recurse to the "correct" side of $M[a : b]$, until $a = b$ and $e = M[a]$ (we can prove the last part more formally by doing and induction exactly the same way as the proof of binary search; we omit the details here as the proof is already done for binary search – and this is the third solution to the same problem!).

Runtime Analysis: Sorting the edges takes $O(m \log m)$ time (say by merge sort). Each binary search step also reduces the length of 'potential' edges (i.e., $M[a : b]$) by a factor of two and hence we only have $O(\log m)$ binary search steps. Finally, each binary search step takes $O(n + m)$ time for running DFS, hence the total runtime is $O((n + m) \log m)$.

Problem 3. You are given a directed graph $G(V, E)$ with two designated vertices $s, t \in V$, and some of the vertices in G are colored *blue*. Design and analyze an $O(n + m \log m)$ time algorithm that finds a path from s to t that uses the *minimum* number of blue vertices (the path can use any number of non-blue vertices).

(15 points)

Hint: Unlike the previous problem, the easiest solution here is a simple graph reduction.

Solution. *Algorithm: (a reduction to the single-source shortest path problem)*

1. Turn G into a weighted graph by assigning the following weights w_e to every edge e of G: any edge that *enters* a blue vertex is assigned a weight of $(n + 1)$ and any other edge is assigned a weight of 1.
2. Run Dijkstra's shortest path algorithm to find the shortest path from s to t in this weighted graph and return this path as the answer.

Proof of Correctness: As any other reduction, the proof of correctness boils down to showing that for every graph G, the path with minimum number of blue vertices (the answer to the first problem) is the same as the shortest path in the new weighted graph (the answer to the second problem.

Consider any s-t path P in the original graph and let b denote the number of blue vertices and r denote the number of non-blue vertices in the path $P - \{s\}$. As edges going into any blue vertex has weight $(n + 1)$ and remaining edges have weight 1, the weight of this path in the newly weighted graph then would be $b \cdot (n+1) + r < (b+1) \cdot (n+1)$ as r is certainly $< n$. This implies that for every integer k, the weight of any s-t path with k blue vertices is always smaller than weight of any path with $(k + 1)$ (or larger) number of blue vertices. As such, finding the path with minimum weight ensures that we find the path with minimum number of blue vertices as well, finalizing the proof.

Runtime analysis: Adding the weights to the graph takes $O(n + m)$ time to go over every edge and running Dijkstra's algorithm takes another $O(n + m \log m)$ time, resulting in an $O(n + m \log m)$ time algorithm.

Problem 4. In the class, we studied the single-source shortest path problem and saw Bellman-Ford and Dijkstra's algorithm for solving this problem. A related problem is the *all-pairs* shortest problem where the goal is to, given a directed weighted graph, compute the shortest path distances from every vertex to every other vertex, i.e., output an $n \times n$ matrix D where $D_{ij} = dist(v_i, v_j)$, namely, the weight of the shortest path from v_i to v_j. Our goal in this question is to design an $O(n^3)$ time algorithm for this problem.

(a) Design an $O(n^2)$ time algorithm that takes as input a weighted directed graph G and any arbitrary vertex v in G, and constructs a new directed graph $G'(V', E')$ with weighted edges such that $V' =$

$V \setminus \{v\}$, and the shortest path distances between any two vertices in G' is equal to the shortest path distances between them in G. **(10 points)**

Solution. *Algorithm:* Obtain $G'(V', E')$ from G by removing the vertex v and all its incident edges from G. Additionally, for any two vertices $u_1, u_2 \in V'$, where there is an edge $e_1 = (u_1, v)$ and an edge $e_2 = (v, u_2)$ in G, add a new edge $e = (u_1, u_2)$ to G' with $w_e = w_{e_1} + w_{e_2}$. Note that this process may create parallel edges in G' (namely, two or more edges with the same direction between the same vertices); however, we can remove the parallel edges by picking the edge with the minimum weight among the edges with the same end-points.

Proof of correctness. Consider any two vertices s and t in $V \setminus \{v\}$; we argue that any shortest path P between s and t in G can be translated into a shortest path P' between s and t in G' with $w(P) = w(P')$ and vice versa. This implies that the shortest-path distance from s to t is equal in G and G'.

Suppose $P = u_1, u_2, \ldots, u_k$ is a shortest path between s and t in G ($u_1 = s, u_k = t$); create P' by traversing the path P and whenever it uses the vertex v, i.e., $u_i = v$, connect u_{i-1} to u_{i+1} in P' directly using the edge $e = (u_{i-1}, u_{i+1})$. By our choice of the weight on the edge e, it is clear that $w(P) = w(P')$. Note that we can simply translate back any shortest path P' in G' between s and t using the same method to a path P in G, hence proving the claim.

Running time analysis. Computing G' from G requires creating a new copy of G which takes $O(n+m) = O(n^2)$ time (as $m \leq n^2$) and considering all (u_1, u_2) pairs which again can be done in $O(n^2)$ time; hence the total running time is $O(n^2)$.

(b) Now *assume* we have already computed all-pairs shortest path distances in G'. Describe an $O(n^2)$ time algorithm to compute the shortest-path distances in the original graph G from v (the vertex chosen in part (a)) to every other vertex in V, and from every other vertex in V to v. **(10 points)**

Solution. *Algorithm:* For any vertex u, we compute the shortest-path distance of u to v in G as follows. Let $u_1, \ldots, u_k$ be the set of all vertices that has an *outgoing* edge to v in G. We let shortest-path distance of u to v as the minimum value of $dist(u, u_i) + w_{e_i}$ for $i \in \{1, \ldots, k\}$, where $dist(u, u_i)$ is the shortest-path distance of u to u_i in G' (which we assumed is already computed) and $e_i = (u_i, v)$.

We can compute the shortest-path distance of v to every other vertex in G similarly by considering the vertices that has an *incoming* edge from v in G.

Proof of correctness. Consider a vertex u and any arbitrary shortest path P of u to v in G. Suppose $P = u, \ldots, u', v$, where u' is a vertex that has an outgoing edge to v in G. We argue that the sub-path of P, from u to u' has the shortest-path distance from u to u'. This is true, since otherwise we could have switched this sub-path in P to obtain an even shorter path from u to v. It implies that the shortest path P of u to v always consists of a sub-path from u to one of u_i's for $i \in \{1, \ldots, k\}$ (vertices with an edge to v), where this sub-path has the shortest-path distance from u to u_i, and an edge from u_i to v. By taking the minimum value among weight of all such paths in the algorithm, we can therefore find the shortest-path distance from u to v.

The proof for shortest-path distances from v to every other vertex is exactly the same by symmetry by considering vertices that have an incoming edge from v.

Running time analysis. The vertex v can have at most $O(n)$ incoming (or outgoing) edges and hence for any vertex u, we need to take the minimum between at most $O(n)$ values, implying that the total running time is $O(n^2)$ as there are n vertices in total.

(c) Combine your solutions for parts (a) and (b) to design a *recursive* all-pairs shortest path algorithm that runs in $O(n^3)$ time. **(5 points)**

Solution. *Algorithm:* Pick an arbitrary vertex v from G and run the algorithm in part (a) to obtain G'. Recursively solve the problem on G' – the base case where the graph only has a single vertex is trivial. Use the algorithm in part (b) to obtain the shortest-path distances from v to every other vertex u and vice versa.

Proof of Correctness: As this is a recursive algorithm, we prove its correctness by induction. Our induction hypothesis is that the algorithm is correct. The base case, where $n = 1$ is trivially true as there is no distances to be computed in that case. Now suppose the algorithm is true for all graphs with n vertices and we prove it for graphs with $n+1$ vertices. By part (a), the shortest-path distances for $V \setminus \{v\}$ in G' and G are equal. Since G' now has one less vertex, we can apply induction hypothesis and say that the distances in G' are computed correctly in the recursive call, which by the previous lines, means we have the correct distances for $V - \{v\}$ in G. By part (b), we can obtain the shortest-path distances for v also, proving that all-pairs shortest-path distances are computed correctly.

Runtime Analysis. The running time of algorithm is $T(n) = T(n-1) + O(n^2)$, where $O(n^2)$ term captures the time required for implementing the algorithms in part (a) and (b). By substitution, we have $T(n) = O(n^3)$ yielding the running time of $O(n^3)$ for the algorithm.

Problem 5. You are given an *undirected* graph $G(V, E)$, with three vertices u, v, and w. Design an $O(m+n)$ time algorithm to determine whether or not G contains a (simple) *path* from u to w that passes through v. You do *not* need to return the path. Note that by definition of a path, no vertex can be visited more than once in the path so it is *not* enough to check whether u has a path to v and v has a path to w. **(20 points)**

Hint: While it may not look like it, this is a maximum flow problem. Create a flow network from G by making all edges in G bidirected and adding a new source vertex s that is connected to u and w. Assign an appropriate capacity to the *vertices* in this graph, and identify an appropriate target vertex t. Finally, find a maximum flow using the Ford-Fulkerson algorithm and argue that this only takes $O(m+n)$ time.

Solution. The general idea of this algorithm is to simply find two vertex disjoint-paths P_1 and P_2 from u, w to v, reverse the order of P_2 to get a path from v to w, and 'stitch' the two paths together to find a path $P_1 \circ P_2$ from u to w that passes v – because P_1, P_2 are vertex disjoint, the path $P_1 \circ P_2$ is indeed a path. We now show how to do this using network flows.

Algorithm (a reduction to the network flow problem):

Create the *network* graph $G'(V', E')$ from G with new source and sink vertices as follows:

1. For every vertex $z \in V$, we add *two* new vertices to G' called z^{in} and z^{out}. We connect z^{in} to z^{out} in G' by a new directed edge. The capacity of these edges is 1.
2. For every edge $\{x, y\} \in E$, we add two new edges (x^{out}, y^{in}) and (x^{out}, y^{in}). The capacity of these edges is also 1 (but it can also be $+\infty$).
3. We connect s to u and w by two directed edges of capacity 1 each. We also connect v to t with an edge of capacity 2.

We then compute the maximum s-t flow in this network G' and if the flow value is 2 we return there exists such a path and otherwise we output no such path.

Proof of Correctness: First note that since the total capacity of neighbors of s is 2, the maximum s-t flow can be at most 2 in G'. Moreover, since only one unit of flow can go from any z^{in} to z^{out} (and z^{in} can only send flow to z^{out}), any feasible flow function can only pass one unit of flow through each vertex. Hence any flow function f, defines a set of *vertex disjoint* paths from s to t in G'. If we have a flow function f that sends 2 units of flow from s to t, we get two vertex disjoint paths $P_1 : u, \ldots, v(= t)$ and $P_2 : w, \ldots, v(= t)$. Therefore, we can find a simple path $P : u, \ldots, v, \ldots, w$ in G, using the undirected edges in P_1 and by *reversing* P_2. Similarly, if there is such a path in G, we can define a flow function f that sends two units of flow from s to v in G', by defining P_1 as the $u - v$ part of the path, and P_2 as the reverse of the $v - w$ part.

This implies that there is a u, w path in G that passes v if and only if the maximum s-t flow in the network G' is 2 and vice versa, proving the correctness of the algorithm.

Runtime Analysis: We can construct the network in $O(n + m)$ time by traversing G. The running time of the Ford-Fulkerson algorithm on a flow network with maximum flow F is also $O(mF)$. Since F is bounded by 2 in the instance create above, the total running time is $O(n + m)$.
