100

HOMEWORK #02

Advanced Topics in Algorithms, Spring Semester, 2013-2014

Question 1

In order to make the proof, I'm going to divide it into the same lemmas as we did in class, but without "discovering" at the end that we need to define the special edges – this will be done in advance.

Logically, for the algorithm we would like to define and work with the **basic length function** ℓ (where $u_f(e)$ denotes the residual capacity of the edge e with respect to the flow f):

$$\ell(e) = \begin{cases} 1 & u_f(e) < 3\Delta \\ 0 & \text{otherwise} \end{cases}$$

In the second step, we will consider **special edges**. An edge (v, w) is said to be special iff:

- It's symmetric edge has a zero length (as assigned in the first step) - $\ell(w, v) = 0$
  - This means that $u_f(w, v) \geq 3\Delta$
- It connects nodes of the same level (as assigned in the first step) - $d_\ell(v) = d_\ell(w)$
- It has a (residual[1]) capacity that when increased by Δ would be at least 3Δ - $r(v, w) \geq 2\Delta$

So, we can define this as the **modified length function** $\bar{\ell}$:

$$\bar{\ell}(e) = \begin{cases} 1 & u_f(e) < 3\Delta \text{ and } e \text{ is } \textbf{not special} \\ 0 & \text{otherwise} \end{cases}$$

The second iteration will update the length of all the special edges to 0, which **would make them all admissible**. <u>This is guaranteed not to change any of the existing level computations</u> (since if the previous length was 1 while $d_\ell(w) = d_\ell(v)$, it means that no shortest path to t used (v, w) in it), meaning that $\boldsymbol{d_\ell = d_{\bar{\ell}}}$.

Using the above length computation, we are going to prove the lemmas we saw in class. We'll begin by making a proof similar to the one we made for Dinic's algorithm; that after adding each flow on the admissible subgraph, **while still in the same phase (<u>Δ remains fixed</u>)**, the distance of s from t **with respect to d_ℓ** increases.

Lemma 1

Let R be a residual graph and L_{ad} be its admissible subgraph, with the basic length function ℓ and a distance function d_ℓ. **After augmenting a flow f' on L_{ad}**, let R' be the updated residual graph, L'_{ad} be the updated admissible subgraph, with the updated basic length function ℓ' and distance function $d_{\ell'}$. Then, we claim that

$$\forall (v, w) \in R', \qquad d_\ell(v) \leq d_\ell(w) + \ell'(v, w)$$

Corollary 1

Basically, we claim that for every residual edge in R', the distance difference between its two nodes couldn't have been smaller in R. It would be more clearly written as

[1] In the slides, this was written as $u(v, w) \geq 2\Delta$. However since we are referring to the residual capacity and not the initial capacity, I chose to denote this as $r(v, w)$

$$d_\ell(v) - d_\ell(w) \le \underbrace{\ell'(v,w)}_{0 \text{ or } 1} \le d_{\ell'}(v) - d_{\ell'}(w)$$

Specifically, if we think of a path from s to t in L'_{ad} passing through $v_1, v_2, \ldots, v_k$ (for simplicity, let's denote $v_0 = s$ and $v_{k+1} = t$), then:

$$d_\ell(s) = d_\ell(v_0) \le d_\ell(v_1) + \ell'(v_0, v_1) \le \left(d_\ell(v_2) + \ell'(v_1, v_2)\right) + \ell'(v_0, v_1) \le \cdots \le \underbrace{d_\ell(v_{k+1})}_{0} + \sum_{i=0}^{k} \ell'(v_i, v_{i+1})$$

$$= d_{\ell'}(v_{k+1}) + \sum_{i=0}^{k} \ell'(v_i, v_{i+1}) = d_{\ell'}(s)$$

Proof of Lemma 1

Let's analyze the type of edges $(v, w) \in R'$ the using ℓ (the previous distance function):

- If $d_\ell(v) < d_\ell(w)$ ((v, w) is a "backwards" arc) then the claim trivially holds since we add the non-negative value $\ell'(v, w)$ to the right side of the equation, preserving the inequality
- The same claim holds for the case when $d_\ell(v) = d_\ell(w)$ ((v, w) connects nodes inside the same level)
- The only remaining case is when $d_\ell(v) > d_\ell(w)$
  - By the definition of our distance function, it means that $\ell(v, w) = 1$
  - So, the only problem would be if $\ell'(v, w) = 0$. Can this happen?
    - No! In order for this to happen, the capacity of (v, w) should have increased
      - We are talking about iteration inside a phase, so Δ remains fixed!
      - The flow increase should have been from less than 3Δ ($\Rightarrow \ell(v, w) = 1$) to at-least 3Δ ($\Rightarrow \ell'(v, w) = 0$)
    - But, this means that flow was augmented on the opposite edge (w, v), yet (w, v) is not admissible so no flow could have been increased on it, since we computed an augmenting flow in L_{ad}!

■

Now we will prove a lemma similar to the previous, but this time we will want the distance to strictly increase (it can't be equal anymore) if the flow was blocking.

Lemma 2

Let R be a residual graph and L_{ad} be its admissible subgraph, with the <u>modified length function $\bar{\ell}$</u> and a distance function d_ℓ. **After augmenting a <u>blocking</u> flow f' on L_{ad}**, let R' be the updated residual graph, L'_{ad} be the updated admissible subgraph, with the updated basic length function ℓ', the <u>updated modified length function $\bar{\ell}'$</u> and distance function $d_{\ell'}$. Then, we claim that there exists **a specific arc (v, w) along the shortest path from s to t** (according lot ℓ) so that

$$d_\ell(v) \lneqq d_\ell(w) + \bar{\ell}'(v, w)$$

Corollary 2

Specifically, if we think of a path from s to t in L'_{ad} passing through $v_1, v_2, \ldots, v_k$ (for simplicity, let's denote $v_0 = s$ and $v_{k+1} = t$), where the lemma holds for (v_{j+1}, v_{j+2}), then by repeating the computation from corollary 1 we would simply get the same result without an equality:

$$d_\ell(s) = d_\ell(v_0) \leq \underbrace{\cdots}_{\substack{\text{Applying}\\ \text{lemma 1}\\ j \text{ times}}} \leq d_\ell(v_{j+1}) + \sum_{i=0}^{j} \ell'(v_i, v_{i+1}) \underset{\text{lemma 2}}{\leq} \underbrace{d_\ell(v_{j+2}) + \overline{\ell}'(v_{j+1}, v_{j+2})}_{\leq \ell'(v_{j+1}, v_{j+2})} + \sum_{i=0}^{j} \ell'(v_i, v_{i+1})$$

$$\leq d_\ell(v_{j+2}) + \sum_{i=0}^{j+1} \ell'(v_i, v_{i+1}) \leq \underbrace{\cdots}_{\substack{\text{Applying lemma 1}\\ k-j+1 \text{ times}}} \leq d_{\ell'}(v_{k+1}) + \sum_{i=0}^{k} \ell'(v_i, v_{i+1}) = d_{\ell'}(s)$$

Proof of Lemma 2

Since we augmented a flow which was blocking, it means that every path from s to t was saturated. So on our current residual path (after we augmented the blocking flow), we must have some edge (v, w) which was not admissible previously (because if all the edges in the current path were admissible before, it means that they all had a capacity of at least 2Δ, but then none of them could have been saturated by a flow of at most Δ!). Having an edge which was not admissible implies that $d_\ell(v) < d_\ell(w) + \overline{\ell}(v, w)$! For that edge, we would like to show the inequality:

20/20

$$d_\ell(v) < d_\ell(w) + \overline{\ell}'(v, w)$$

Let's analyze the possible cases for (v, w):

- Like before, if (v, w) was a "backwards" arc ($d_\ell(v) < d_\ell(w)$) then the claim trivially holds since we add the non-negative value $\ell'(v, w)$ to the right side of the equation, preserving the inequality
- The case where (v, w) was a "forwards" arc ($d_\ell(v) > d_\ell(w)$) does not exist, because <u>since all sizes are integers</u>, it means that $\overline{\ell}(v, w)$ must be at least 2 in order to change the direction of the inequality and this is not possible!

$$\begin{cases} d_\ell(v) > d_\ell(w) & \Rightarrow \quad d_\ell(v) \geq d_\ell(w) + 1 \\ d_\ell(v) < d_\ell(w) + \overline{\ell}(v, w) & \Rightarrow \quad d_\ell(v) \leq d_\ell(w) + \overline{\ell}(w, v) - 1 \end{cases}$$
$$\Rightarrow d_\ell(w) + 1 \leq d_\ell(w) + \overline{\ell}(w, v) - 1$$
$$\Rightarrow 2 \leq \overline{\ell}(w, v)$$

- The only remaining case is where (v, w) was an **arc going inside the same level** ($d_\ell(v) = d_\ell(w)$), implying that $\overline{\ell}(v, w) = 1$ in order for the original inequality ($d_\ell(v) < d_\ell(w) + \overline{\ell}(v, w)$) to hold
  - So, we will only have a problem if $\overline{\ell}'(v, w) = 0$. Can this happen?
    - The first option is that (v, w) became special, but this can't happen:
      - If (v, w) was not special ($\overline{\ell}(v, w) = 1$), even though $d_\ell(v) = d_\ell(w)$, and if also $\overline{\ell}(v, w) = 1$, it implies that either $u_f(w, v) < 3\Delta$ (while $u_f(v, w) < 3\Delta$) or $u_f(v, w) < 2\Delta$
      - Since increasing the residual capacity of one of them would force us to reduce it on the other (since we augment flow on the other edge), it means that we can't satisfy

both of these conditions together – meaning (v, w) can't be special under the updated flow f'

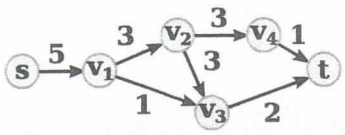
- The second option is that $u_{f'}(v, w) \geq 3\Delta$, but this also can't happen:
  - For this to happen, we must have pushed flow on (w, v) in the augmented flow
  - But we pushed at most Δ flow[2], meaning that we must have had $u_f(v, w) \geq 2\Delta$
  - But, this means that (v, w) should have been special, which is a contradiction to the fact that we had $\bar{\ell}(v, w) = 1$!

■

Question 2

There are several assumptions that I made when solving this question about details that were not explicitly mentioned:

- t never participates in a discharge operation (because it doesn't make sense to do any operations on it based on its excess – we do want it to be active and there is no sense in moving flow away from it (pulling or pushing) to make it inactive)
- t starts as "blocked" – this helps in some of the explanations, and it does not matter since it never participates in a discharge operation
- t has no outgoing edges (in a similar fashion to the fact that s doesn't have incoming edges)
- t is the last element in the topological ordering
  - This can be accomplished only once t has no outgoing edges
  - If t doesn't turn out last in the ordering, it can be moved to the end because it shouldn't affect the correctness of the ordering
  - Finally, I make this assumption to simplify some of the explanations of **part b**. It has no effect whatsoever on the actual algorithm performance
- Finally, <u>there is a slight correction that needs to be done in the algorithm</u>. As it is described currently, if we advance the incoming edge pointer after every pull operation, we may "run out" of incoming edges before we finish to reduce the excess, as shown in the following graph:



Let (v_2, v_4) precede (v_2, v_3) in the outgoing list of v_2, and let it be that $T = s, v_1, v_2, v_3, v_4, t$.
It is easy to see that v_1 would have to pull twice on (s, v_1) due to the discharge order: v_1 (block! **pull** $(\boldsymbol{s}, \boldsymbol{v_1})$), v_2, v_3, v_4 (block! pull (v_2, v_4)), v_2 (blocked, pull (v_1, v_2)), v_1 (**pull** $(\boldsymbol{s}, \boldsymbol{v_1})$), ...
The correct version should have been "3) if v is blocked, we perform a pull on the current incoming arc (v, u) if possible. **Only if not possible,** advance the incoming arc pointer". However, in order for this to

[2] Now that I think about it, when we route the flow of Δ inside each strongly connected component, we could have created a flow of 2Δ on each arc (that's why the limitation for special arcs was at least 2Δ). But, if this is true, then we should have changed our limit from 3Δ to 4Δ for this proof to work. So, something here feels missing...

work without any penalties in running time, the list of incoming vertices that we maintain for each vertex should contain only edges which have some flow on them, and support insertion and deletion in $O(1)$ time!

- This can be done by having a doubly linked-list of the relevant incoming edges, where each edge also holds a pointer to its node in the list
- Insertion of elements which are not already in the list will be to the end of the list

Part a

We'll begin by understanding when we block a vertex; during the discharge operation, we perform 3 operations:

1. (While we are not blocked,) **Push** as much of the excess flow of v as possible, to the next nodes
2. (If we are not blocked,) **Block** if we can't push anymore flow out and the node is active (= has excess)
3. (If we are blocked,) **Pull** - Send back the excess to the previous nodes

Now, let's start analyzing this:

Lemma 1.1: At the moment of marking a vertex v as blocked, there must be a saturated edge on every path from v to t.

Proof 1.1: This definition of step 2 of the discharge step is that a node is <u>marked as blocked only when all of its outgoing edges are saturated!</u> So, at least initially, when the node is blocked, indeed all paths from v to t (or any other place) are indeed blocked. ■

To prove the invariant, we must make sure that whenever one of its outgoing edges is de-saturated (meaning some flow is reduced from it), the invariant still holds. This will be the logic behind our proof of the claim:

Claim 1.2: Let w be any blocked edge. Therefore, every path from w to t contains a saturated edge.

Proof 1.2: By induction on k, the length of the path from w to t (The path is $w = v_0, v_1, \dots, v_k = t$)

<u>Base:</u> For $k = 1$, we are discussing the case where the path consists simply of one edge - (w, t). When w was blocked, then by lemma 1.1 we know that (w, t) was saturated. Additionally, we never need to do a discharge step on t, and so there is no step in the algorithm that would do a pull operation on (w, t)

<u>Induction Hypothesis:</u> We assume that for every path from w to t of (integer) length up to k, the path contains a saturated edge.

<u>Induction Step:</u> We would like to prove that for every path from w to t of length $k + 1$, the path contains a saturated edge. To do so, we'll divide the path in to two parts – the edge (w, v_1) and the path from v_1 to t (which is of length k). Now, let's examine the possible cases:

- The first case is when v_1 is a vertex that was not blocked yet. In that case, (w, v_1) couldn't have participated in a pull operation (by the definition of the pull operation) after it was saturated when w was blocked (lemma 1.1). Therefore, (w, v_1) must have remained saturated since no other step in the algorithm decreases the flow!

- The second case is when v_1 is a blocked vertex. In that case, we can use the induction hypothesis to conclude that the path contains a saturated edge (somewhere between v_1 and t) ■

Part b

For this claim, we would need to use the topological sorting of the graph. Intuitively, the algorithm behaves in the following way:

- Start at s and go forward (according to the topological ordering T), pushing flow as much as you can
  - Once we reach t (which is at the end of the ordering), it means that we already passed through all the other nodes and pushed as much flow as we can forward
- Now, start going back from t to s (or forwards if we consider the order T^R), where at each step you solve the excess in the current node
  - You do so by discharging the current node (which should become blocked)
  - As part of the discharge step, you pull back excesses to the previous nodes
    - If these nodes are blocked, we will continue pulling back the excess as we go back to s
- Now, for the nodes that pulled some excess back without being blocked themselves, we will repeat the process of pushing flow forward from s to t (and then back) to try and get the flow forward until it's "stuck" (until it saturates some edges)

We'll denote $s = v_{T_0}, v_{T_1}, \dots, v_{T_n} = t$ as the topological ordering of the graph ($v_{T_i} = v_{T^R_{n-i}}$)

Lemma 2.1: The last non-blocked vertex v that was discharged on T (before heading to discharge vertices in T^R) must have been blocked by this discharge operation

Proof 2.1: Let's assume it was not blocked, which means one of several cases:

- It was not active
  - <u>This can't be the case</u> – we only apply discharging on vertices which are active (after marking them when pushing flow into them)
- It is now inactive, meaning it managed to push out all of its excesses
  - But, pushing means having a non-blocked vertex w so that (v, w) is an edge in the graph
  - Applying a push like that means that we should have marked w (since it's ought to be active after this push operation)
  - w comes after v in the topological ordering, meaning that by the definition of the algorithm, it should have been discharged after v if we follow the ordering T (and before heading to discharge blocked vertices according to T^R). <u>This is a contradiction!</u>

Since we achieved a contradiction to our assumption, we proved the original claim! ■

Claim 2.2: <u>As long as there are unblocked vertices</u>, Every $n - 1$ discharge operations will block a vertex

Proof 2.2: Assume there are k blocked vertices after discharging the last unblocked vertex (which now became charged). The maximal amount of discharges that we can apply on blocked vertices (in T^R) is $n - k - 1$ (-1 because we don't apply discharge operations on s). After that, we would need at most $k - 1$ more discharge

operations on unblocked vertices (we have only k more vertices in T) before discharging the last non-blocked vertex in T (which should now become discharged according to lemma 2.1). In total, we have at most $(n-k-1)+(k-1)=n-2$ discharge operations between blocking nodes; so after at most $n-1$ discharge operations, we will block a node.

Part C

Lemma 3.1: When we discharge a blocked node, it becomes inactive

Proof 3.1: Implied directly be the definition of the discharge operation – as long as it has some excess, we reduce the incoming flow by applying pull operations on the incoming nodes. The only potential pitfall is that we need to make sure that we won't "run out" of incoming edges before we finish pulling out all the excess. This is guaranteed by the fix that we made in the algorithm – we have a list of incoming edges with flow on them. When the discharge operation stops, we either have no more excess, or the list is empty (which implies that there is no incoming flow at all! So the vertex is also inactive in this case!) ■

Lemma 3.2: Once all vertices are blocked, and we are not in the middle of any discharge operation, let k be the smallest index (in T^R) of an active vertex. Then no node that comes before $v_{T_k^R}$ it in T^R will be discharged again.

Proof 3.2: Once all nodes are blocked, and we are not in the middle of any discharge operation, the only reason for discharging a vertex is that it will become active. But, in order to make $v_{T_j^R}$ active, we must discharge some other vertex $v_{T_i^R}$ with $i<j$! So by induction (omitted since it's obvious and long...), the claim holds. ■

Lemma 3.3: Once all vertices are blocked, and we are not in the middle of any discharge operation, let k be the smallest index (in T^R) of an active vertex. Then after discharging $v_{T_k^R}$, the index (according to T^R) of the first active vertex in T^R monotonically increases (would be at least $k+1$).

Proof 3.3: Trivially implied from the previous lemmas; discharging $v_{T_R^k}$ makes it inactive (lemma 3.1), and can't activate any nodes that with an index smaller than k in T^R (lemma 3.2) ■

Lemma 3.4: Once all the nodes are blocked, and we are not in the middle of any discharge operation, we need at most $n-1$ discharge operations for the algorithm to terminate

Proof 3.4: Implied directly from lemma 3.3. Let $v_{T_k^R}$ be the first node (in the order T^R) which is active (we know that $k \geq 1$ because $v_{T_0^R}=t$ and it's irrelevant). Each discharge operation increases k, and k can't increase to more than $n-1$ ($v_{T_n^R}=s$ and it's irrelevant). So we can have at most $n-1$ discharge operations. ■

Lemma 3.5: Once we advance from an edge (v,w), in the incoming list of w, to the next edge, we won't work again on (v,w) from discharging w

Proof 3.5: Implied by the (modified) algorithm definition – we advance only when we can't pull any more flow from it, meaning that it has no flow on it! When it entered the list, it was because some flow was pushed on it when discharging v, and by definition of the discharge step we never push flow twice on the same edge. So, it

entered after the last push on it, and it leaves when there is no flow on it (and no more flow can be added to put it back in the list). ■

Lemma 3.6: The algorithm requires at most $O(n^2)$ discharge operations

Proof 3.6: We block an additional node after at most every $n-1$ discharge operations (claim 2.2), so we can spend at most $(n-1)\cdot(n-1) = O(n^2)$ discharge operations on blocking edges. Afterwards, we spend at most $n-1$ more discharge operations until the algorithm terminates (lemma 3.4). In total, we had $O(n^2)$ discharge operations. ■

Claim 3.5: The algorithm terminates in $O(n^2)$ time

Proof: We can discharge a non-blocked vertex v for at most $outcount(v)$ time, where $outcount(v)$ is the number of outgoing edges from v. This is because the discharge operations for non-blocked vertices do at most $O(1)$ work for every outgoing edge <u>without repeating edges!</u> So, the total amount of time spent on discharging vertices while they are non-blocked is $O(|E|) \leq O(n^2)$

Now, the question is how much time do we spend on discharging vertices while they are blocked? Well, a similar logic applies here. Ignoring for a moment the case when we pull flow on an edge without removing it from the outgoing list, we would have spent $incount(v)$ time on working on discharging blocked edges, requiring $O(|E|) \leq O(n^2)$ work time in total. So, we just need to show that we don't spend more than $O(n^2)$ time on discharging blocked vertices without removing the incoming edge. Every discharge operation on a blocked vertex, which does not remove incoming edges, costs $O(1)$ work – so since we have $O(n^2)$ discharge operations, this also doesn't cost more than $O(n^2)$ time.

The last remaining part is to figure out how much does it cost to mark active nodes for working, according to the topological order T or T^R. Apparently, we don't need any fancy data structure for this – we can simply mark each node with two Boolean flags (**blocked** and **requires-work**), and traverse T and T^R in order. If we follow lemma 2.1, then every time we finish traversing T (while there are nodes to block), we block a node. So, we can have at most $O(n)$ traversals over T and T^R – costing $O(n \cdot 2n) = O(n^2)$ time. Finally, when all nodes are blocked, we showed that we need simply one traversal over T^R to terminate – this costs $O(n)$ time. ■

Question 3

Part a

We actually saw this in class, in the proof of the optimality criteria 2 (f is an MCC $\Leftarrow\Rightarrow$ there exists a potential function π so that $c^{\pi}(e) \geq 0$ for every residual arc e). Pick an arbitrary vertex t and let $\delta(v,t)$ be the length of the shortest path from v to t in the residual of f (where the length of an edge is simply its cost). Set $\pi(v) = \delta(v,t)$. Note that by the definition of the cost function ($c(v,w) = -c(w,v)$), we know that $\delta(v,t) = -\delta(t,v)$.

Correctness: First, we can see these distances are well defined; since we have a minimal cost circulation, then by optimality criteria 1 we know that no cycle of negative cost exists (meaning that there are no cycles of negative length by our definition). Now, let's assume $c^{\pi}(v,w) < 0$ for some residual edge (v,w):

$$\pi(w) - \pi(v) + c(v,w) < 0$$

$$\delta(w,t) + c(v,w) < \delta(v,t)$$

However, this can't be correct! This is because of the definition of the shortest path:

$$\delta(v,t) = \min_{\substack{w \in V \\ (v,w) \in E}} \{c(v,w) + \delta(w,t)\}$$
$$\Rightarrow \forall w \in V \text{ so that } (v,w) \in E, \qquad \delta(v,t) \leq c(v,w) + \delta(w,t)$$

So, we proved the $c^{\pi}(v,w)$ cannot be negative $\Rightarrow c^{\pi}(v,w) > 0$

Running time: We need to find the shortest paths from each node in the residual graph to some node t. We have two algorithms which are capable of handling this problem – the Bellman-Ford algorithm (which handles negative cycles) which runs in $O(|V||E|)$, and Dijkstra's algorithm which runs in $O(|E| + |V|\log|V|)$ (and does not handle negative cycles).

Luckily for us, by optimality criteria 1 (and as we explained in the correctness proof), we don't have any negative length cycles since we have an MCC – so we can use Dijkstra's algorithm for this procedure, which would make the overall running time as $O(|E| + |V|\log|V|)$.

Part b

If we would have had f, then by optimality criteria 2 (f is an MCC $\Longleftrightarrow$ there exists a potential function π so that $c^{\pi}(e) \geq 0$ for every residual arc e) we would have been able to state that it is the MCC. So, the problem we are trying to solve is how to find f efficiently.

By using the optimality criteria, we know that every edge which has a negative reduced cost (with respect to π) cannot be residual in f – so we can begin with saturating these edges. Afterwards, we would like to ship **all** the flow all the excess vertices (A) to the deficit vertices (D), but without modifying the edges we saturated (meaning that we shouldn't add any flow on the opposite edges); we know that there is a feasible circulation that saturates these edges (f does this!).

In order to ship the flow, we must understand the different types of edges (v,w) in G:

- Edges so that $c^{\pi}(v,w) < 0$ – these are edges that we saturate, so they cannot participate in the computation of pushing flow from A to D (simply because they are saturated)
- Edges so that $c^{\pi}(v,w) > 0$ – pushing flow from A to D on these edges mean that we create a residual capacity on (w,v) where $c^{\pi}(w,v) < 0$ while we wanted (w,v) to be saturated! So, these edges also cannot participate in the pushing of flow from A to D
- Edges so that $c^{\pi}(v,w) = 0 (= c^{\pi}(w,v)$ – These are the only edges that will be in the graph in which we'd like to push flow from A to D

So, the most trivial solution would now be to compute a minimum cost flow between A and D (using the same construction we did for the refine step (with adding s to A and t to D)), while including only edges so that $c^{\pi}(e) = 0$. This will yield the correct solution, because:

- The resulting flow value would be correct because:
  - The optimal flow value cannot be more than the sum of all the excesses (we saw that for the refine step)
  - The optimal flow value can reach the sum of all the excesses (f does that!)
- The resulting overall cost would be correct because:
  - The resulting flow would minimize the cost over all the "free" edges we have (i.e. edges that we don't already know their flow (and also their cost in the final circulation)), and we know that the by adding the costs of the "fixed" saturated flows, we are following an optimal circulation f

20/20

The overall complexity is $O\left(|E| + T_{\text{max-flow}}(G)\right)$.

Question 4

To save some space, maximal-flow of minimal-cost would be abbreviated as MFMC.

Part a

First, let's begin by explaining why this does algorithm find a maximal flow (regardless of the cost):

- The algorithm finds augmenting paths in the residual network, sets a flow on it based on the minimal capacity of any edge on the path, and adds the path to the flow
- The algorithm will only halt once there are no more augmenting paths
- **This follows exactly the definition of the Ford-Fulkerson method**, which is known to be correct (in finding the maximal flow) and to terminate for networks of integer capacities

Now, let's see why the cost is indeed minimal. In order to do this, we'll need to use the following lemma:

Lemma 1.1: The MFMC in a graph without any negative costs, does not have any flow cycles (i.e. a cycle in G where on all the edges there is a positive flow).

Proof 1.1: Given any such cycle, let f_0 be the minimal flow value on any of its edges. By reducing f_0 from the flow value on all of its edges, we preserving the flow constraints, not harming the overall flow from s to t, and at the same time decreasing the overall cost! Since this is a contradiction to the minimal-cost of the flow, we conclude that there can't be any flow cycles! ■

Lemma 1.2: The residual graph of an MFMC, where none of the edges in G had a negative cost, can't have any negative cost cycles

Proof 1.2: Implied directly from lemma 1.1 – the existence of a negative cost cycle in the residual network implies the existence of a positive flow cycle in the flow (remember, the residual network includes only edges that have flow on them in G)! ■

Now, for a short comment regarding "non-negative costs" as it was mentioned in the question; as we learned in class, in the maximal-flow of minimal-cost problem, $c(v,w) = -c(v,w)$. So, the only way for the constraint in

the question to be followed, is that for every edge (v, w) in G, if $c(v, w) > 0$ then (w, v) is not a part of G. To distinguish the edges, I'll give them names:

- **Original** edges are edges that were present in G
- **Residual** edges are any of the edges that exist in the residual graph (having a positive capacity)
- **Virtual** edges are residual edges that haven't existed in G (i.e. they are not one of the original edges)

Lemma 1.3: Let f be a maximal flow in G, a graph without any negative costs on the original edges. Then:
f is MFMC $\Leftrightarrow$ there can't be any cycles of negative cost in the residual graph

Proof 1.3 $\Rightarrow$: Assume there is such a circulation. We can extract from it a cycle of negative cost, so let c_0 be the minimal capacity value on any of its edges. By adding c_0 flow on all the edges, we preserving the flow constraints, not harming the overall flow from s to t (because the excess flow in each node stays the same), and at the same time decreasing the overall cost! Since this is a contradiction to the minimal-cost of the flow, we conclude that there can't be any negative cost cycles! ■

Proof 1.3 $\Leftarrow$: Assume f is not an MFMC and let f^* be an MFMC so that $Cost(f^*) < Cost(f)$. Let's take a look at $f^* - f$, which is a 0-valued flow from s to t in the residual graph of f (meaning that there is some flow, but the overall excess of t doesn't change), with a negative cost (since everything is linear, we conclude that $Cost(f^* - f) = Cost(f^*) - Cost(f) < 0$). By the flow constraints, the only way to add flow on some edges without changing the excess of any node, is by adding a circulation. But this means we have a circulation of negative cost, which is a contradiction! ■

So, finally we can see what we need to prove – that our algorithm never creates any negative cycles in the residual graph; doing so will prove that it also reaches the minimal cost, out of all the maximal flows!

Proof: By induction, on k – the number of paths that were augmented from the residual graph until now.

<u>Base:</u> For $k = 0$, the residual graph is simply G, and G can't contain negative cost cycles (because it doesn't even have any edge of negative cost...)

<u>Induction Hypothesis:</u> We assume that after j augmented paths, at no stage there were any negative cost cycles

<u>Induction Step:</u> We would like to prove that after augmenting the $j + 1$ path, we still don't have any negative cost cycles. So, let's assume that after augmenting a path $\Gamma = \{e_0, e_1, \dots, e_p\}$ (e_i are edges), a negative cost cycle was created. Since the cycle haven't existed before we augmented the flow, we can conclude that the cycle uses the residual edges that were create by some sub-path $e_\alpha, \dots, e_\beta$ of Γ. By denoting the symmetric edge of e as $\bar{e}$, we can denote the cycle as $\{\bar{e}_\beta, \bar{e}_{\beta-1} \dots, \bar{e}_\alpha, e_\gamma, e_{\gamma+1}, \dots, e_\delta\}$.

First of all, we know that the following inequality holds:

$$(*) \quad \sum_{i=\alpha}^{\beta} c(e_i) \leq \sum_{i=\gamma}^{\delta} c(e_i)$$

This is because it's easy to see that otherwise it means that we haven't picked the path with the minimal cost! (In that case, the path $\Gamma_2 = \{e_0, \dots, e_{\alpha-1}, e_\gamma, \dots, u_\delta, e_{\beta+1}, \dots, e_p\}$ would have had a lower cost than the cost of Γ!)

Additionally, due to the negative cost of the cycle, we can see that:

$$\sum_{i=\alpha}^{\beta} c(\bar{e}_i) + \sum_{i=\gamma}^{\delta} c(e_i) < 0$$

$$\sum_{i=\gamma}^{\delta} c(e_i) < \sum_{i=\alpha}^{\beta} -c(\bar{e}_i) = \sum_{i=\alpha}^{\beta} +c(e_i)$$

However, this is a contradiction to (*)! So, there can't be any negative cost cycles! ■

Part b

Note – I'm using the classic time bounds on Dinic's algorithm (without the special bounds for integer capacities).

Implementation #1

We are going to maintain something similar to the implementation we saw for Dinic using a dynamic tree. The main difference is that instead of having L as the tree resulting from a BFS from s, we should have L as the tree created by selecting for each node v the edge to the next node on the path to t of minimal cost (we may select multiple edges if several paths exist). Other than that, we can continue normally – find a blocking flow in our version of L, augment it, recomputed L, and repeat.

Constructing L requires Belamn-Ford (from s), requiring $O(nm)$ work. In the analysis we did for Dinic, we showed that other than that, we do a work of $O(\log n)$ on each edge on each forest L, and that the distance increases after that by at least 1. The maximal distance between s and t by our definition is nC, and so the overall cost is $O\big(nC \cdot (m \log n + mn)\big) = O(n^2 mC)$

20/20

Implementation #2

Do the straight-forward implementation – use Dijkstra's algorithm to find the "shortest" path each time (where length = cost), and add it. We can do this since (as we proved), the residual network does not contain any negative cycles at any time, and the cost of Dijkstra's algorithm is $O(m + n \log n)$. Since the maximal flow is nU, and since each flow has a size of at least 1, the running time for this algorithm is $O\big(nU \cdot (m + n \log n)\big)$

reduced costs are (c - Sk + Sv)

Question 5

If I understand the question correctly, then when we speak of an undirected <u>forest</u>, we mean a collection undirected <u>acyclic connectivity components</u>. This means that whenever we delete some edge, it has to be the case that we are splitting one connectivity component in to two different components (because if a different connecting path exists between the two vertices, then together with this edge they would have formed a cycle).

Another interesting property which is implied by the fact we have an undirected forest is that it has only $O(n)$ edges; this is because a tree with m edges connects $m + 1$ vertices, so we cannot have more than $n - 1$ edges

in our setup. This implies that since we are dealing with k operations where $k \gg n$, then we are going to have $k_{lookup} \gg n$ connectivity test operations and only $O(n)$ deletions. So, we'll attempt optimizing the lookup operations, while willing to pay a bit more for deletion operations.

The first thing we'll need is a slight modification on the structure of dynamic trees so that for each node we'll be able to get the number of nodes "under it" (i.e. nodes that their path to the root passes through the given node). This should trivially work as long as we update it in rotations, since we only perform additions/deletions of links when the node is the root of a tree – so only one node needs to be updated.

In addition, each node of the tree will contain an integer field called **SetNumber**, which should be the same for all the nodes under the same tree. We will keep the largest allocated SetNumber as a global value, to allow allocating new unused values in $O(1)$ time (we'll increment it whenever we allocate a new value).

Now, here is how the structure works:

- $connected(v, w)$: Simply compare the SetNumber of v and w (return True if same, False otherwise) <u>without doing anything to the dynamic tree</u> (not even splaying them) since technically we are not using the dynamic tree, just the nodes directly. This always takes $O(1)$ time
- $delete(v, w)$: Apply the delete (unlink) operation of the dynamic tree to delete the link and create two smaller trees in $O(\log n)$ amortized time. **<u>Pick the smaller tree</u>** (possible due to our modification to include sizes), **allocate a new SetNumber** and **assign it to all the nodes of that tree**

Now, we'd like to prove that we won't spend more than $O(n \cdot \log n)$ time on updating SetNumber values (which is the time already spent by the deletions using the dynamic tree). So let's prove this formally:

Lemma: In a forest with n nodes (and at most $n - 1$ edges), no deletion series would cause more than $n \cdot \log_2(n)$ SetNumber updates.

Proof: In induction on n

<u>Base:</u> For $n = 1$, this is trivially correct (there can't be any edge deletions, and the bound is indeed 0). For $n = 2$, we can have at most one deletion, and $1 \leq 2$, so the claim also holds.

<u>Induction Hypothesis:</u> We assume that in every forest with at most j nodes, no deletion series would cause more than $j \cdot \log_2(j)$ SetNumber updates

<u>Induction Step:</u> We would like to prove that in every forest with at most $j + 1$ nodes, no deletion series would cause more than $(j + 1) \cdot \log_2(j + 1)$ SetNumber updates.

- Let $T_1, T_2, \dots, T_t$ be the trees in the forest, and W.L.O.G let's assume that we are performing a deletion in T_1, yielding two new tree - T_{t+1} and T_{t+2} with n_{t+1} and n_{t+2} nodes respectively where $n_{t+2} \leq n_{t+1}$
  - We know that $n_{t+2} \leq (j + 1)/2$ (since $2n_{t+2} \leq n_{t+2} + n_{t+1} \leq j + 1$)
- We will divide the trees into two forests - $F_1 = \{T_2, \dots, T_t, T_{t+1}\}$ and $F_2 = \{T_{t+2}\}$
  - F_1 has $j + 1 - n_{t+2}$ nodes
  - F_2 has n_{t+2} nodes

- Now, let's analyze the overall amount of SetNumber updates:
  - From the deletion that created T_{t+1} and T_{t+2}, we got n_{t+2} SetNumber updates
  - From doing deletions in F_1, by the induction hypothesis, we will do at most $(j+1-n_{t+2}) \cdot \log_2(j+1-n_{t+2})$ updates
  - In a similar fashion, for F_2 do at most $(n_{t+2}) \cdot \log_2(n_{t+2})$ updates
  - Summing it all, we get:

$$
\begin{aligned}
&(n_{t+2}) + \big((j+1-n_{t+2}) \cdot \log_2(j+1-n_{t+2})\big) + \big((n_{t+2}) \cdot \log_2(n_{t+2})\big) \\
&\qquad \le (n_{t+2}) + \big((j+1-n_{t+2}) \cdot \log_2(j+1)\big) + \left((n_{t+2}) \cdot \log_2\left(\frac{j+1}{2}\right)\right) \\
&\qquad = (n_{t+2}) + \big((j+1-n_{t+2}) \cdot \log_2(j+1)\big) + \big(n_{t+2} \cdot (\log_2(j+1) - 1)\big) \\
&\qquad = \big((j+1-n_{t+2}) \cdot \log_2(j+1)\big) + (n_{t+2} \cdot \log_2(j+1)) = (j+1) \cdot \log_2(j+1)
\end{aligned}
$$

■

So, the maximal overall time for k operations is $O(k + n \log n)$

20/20