NETWORK OPTIMIZATION

Megan Bryant
mrbryant@email.wm.edu

Department of Mathematics, The College of William and Mary, P.O. Box 8795, Williamsburg, VA 23187

Executive Summary

Homework 7.
Due date: April 2nd, at the start of class
Collaborators:

1

(Quidditch Elimination) Consider the problem of determining whether your favorite professional league quidditch team has been eliminated from the championship at some point in the middle of the season. The only rules of professional quidditch that you need to know are that there are 3 points awarded for each game. If the team has the highest score and captures the Golden Snitch, then 3 points go to the winner, and 0 go to the loser; if the team has the highest score but doesnt capture the Golden Snitch, it gets 2 points and the other team gets 1 point. Another way to view this is that the team with the highest score gets 2 points, and the team that captures the Golden Snitch gets 1 point. The following are the current standings in the season

		Remaining Games			
Team	Points	SL	GR	RA	HU
Slytherin	22	-	1	1	1
Gryffindor	21	1	-	2	1
Ravenclaw	17	1	2	1	1
Hufflepuff	15	1	1	1	-

Set up a maximum flow problem and use it to argue that Hufflepuff has not yet been eliminated by the professional scoring rules. For full credit on this part, you must (1) explain how your maximum flow problem models the problem of determining an outcome such that Hufflepuff is not eliminated (in other words, you should fully explain why Hufflepuff is not eliminated if and only if <some condition on the maximum flow>), and (2) solve the flow problem, and from it derive the outcomes of the remaining games that show a scenario in which Hufflepuff can still win the championship.

We know that if Hufflepuff were to win all 3 of its remaining games and wins all 3 points in each game, then they can achieve a maximum of 9 additional points for a maximum total of 24 points. They will win (or tie) the championship if and only if no other team scores more total points then they do. Therefore, the maximum number of points that each of the remaining teams can win is as follows.

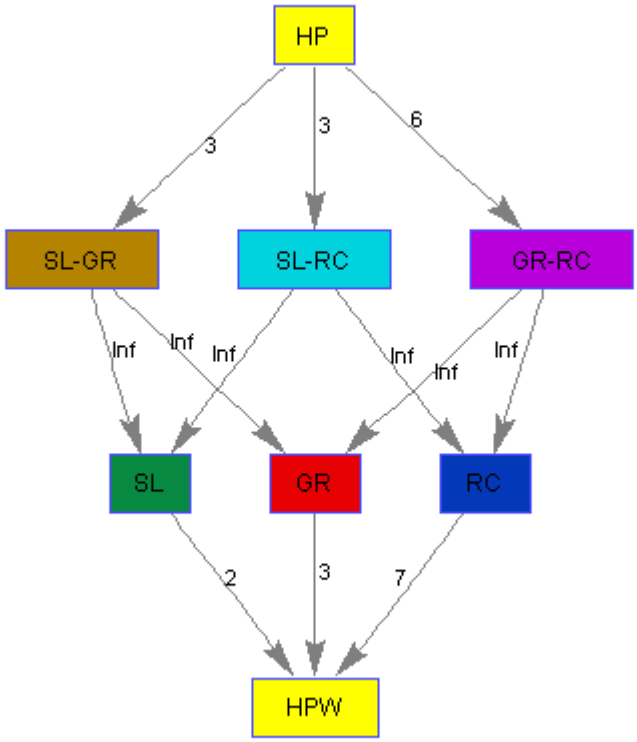
House	Max Points Allowed
Slytherin	$24 - 22 = 2$
Gryffindor	$24 - 21 = 3$
Ravenclaw	$24 - 17 = 6$

Now, we know from the graph that we have the following remaining matchups (not including Hufflepuff games, which we have assumed that they won) and the following points up for grabs per pair.

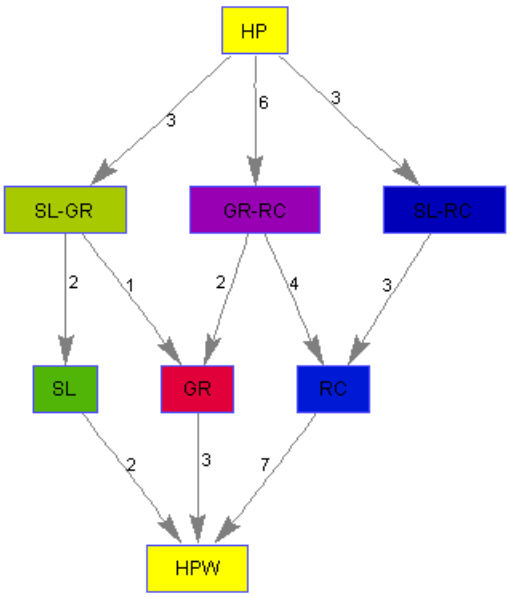
Matchup	Number of Games	Total Points Available
Slytherin-Gryffindor	1	3
Slytherin-Ravenclaw	1	3
Gryffindor-Ravenclaw	2	6

So, we will let $g^* = 12$ be the total number of points left to be claimed among the other teams. An integer flow value of g^* thus gives us a scenario in which no other team gets more wins than our team. This also means that a scenario in which our team wins gives us a flow value of g^*.

We will solve this max flow problem in Matlab using the Push-Relabel algorithm. The flow graph was created using an assignment matrix which was then made sparse for use in our algorithm. The biograph below represents our input to the algorithm.



This graph was then processed using Matlab's Max Flow algorithm. The maximum flow is thus determined to be $g^* = 12$. Based on our formulation, we know that his means that a scenario exists in which Hufflepuff is not eliminated from the competition. The flow along the graph represents the total points that each team must win from each matchup. Below is the resulting maximum flow represented as again as a biograph.



This maximum flow graph tells us that there exists one scenario in which the outcomes of the remaining games results in Hufflepuff not being eliminated from the competition. In fact, since the flows on each of the outgoing arcs from the house nodes are exactly what each house needs to achieve 24 total points, we see that the only circumstance in which Hufflepuff is not eliminated is actually a tie between the houses. The following table details a possible scenario outlined by the max flow graph for conditions under which Hufflepuff remains competitive (note: since Gryffindor and Ravenclaw had two games, there are multiple scenarios pictured in the flow graph).

Matchup	# Games	Points	Outcome
SL-GR	1	Slytherin: 2 Gryffindor: 1	Highest Points Snitch
GR-RC	2	Ravenclaw: 2 Gryffindor: 1 Ravenclaw: 2 Gryffindor: 1	Highest Points Snitch Highest Points Snitch
SL-RC	1	Ravenclaw: 3 Gryffindor: 0	Highest Points + Snitch

Thus, we can confirm that there does in fact exist a scenario in which house Hufflepuff is not eliminated from the Quidditch championship. Included below is a list of the Matlab codes that were used to produce these results.

```
EDU>> S = sparse(A)
EDU>> [U,V,W] = graphmaxflow(S, 1, 8);
EDU>> BG= view(biograph(S,data,'ShowWeights','on'));
EDU>> data = [' HP '; 'SL-GR';'SL-RC';'GR-RC'; ' SL '; ' GR '; ' RC '; ' HPW '];
EDU>> BF= view(biograph(V,data,'ShowWeights','on'));
```

2

A plumber is trying to decide which tools to purchase to maximize his income on the jobs he can do with them. There are five different types of jobs, and the table shows the profit he expects to make per week for each job type. The table also shows the tools he would need to purchase in order to be able to do the different jobs. He will spread the cost of the tools over a year, and for each tool i, $c(i)$ gives the cost divided by 52, i.e., it is the cost of the tool per week.

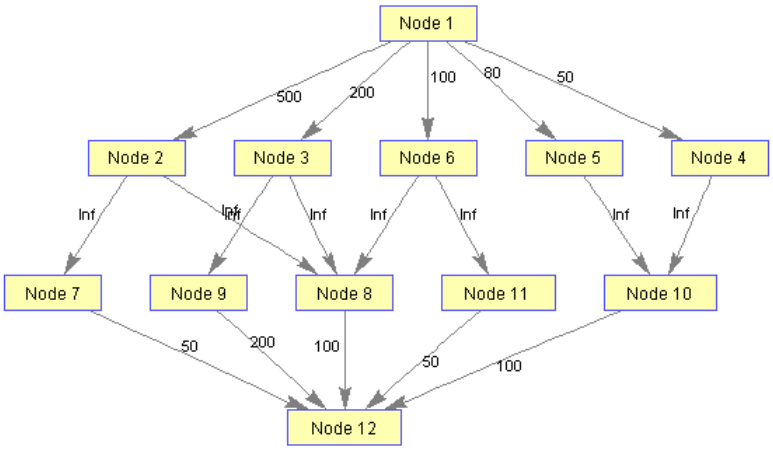
Jobs\Tools	I_1	I_2	I_3	I_4	I_5	p_j
J_1	1	1	0	0	0	500
J_2	0	1	1	0	0	200
J_3	0	0	0	1	0	50
J_4	0	0	0	1	0	80
J_5	0	1	0	0	1	100
cost (c_i)	50	100	200	100	50	

Help the plumber decide which tools to buy and which jobs to accept, in order to maximize his expected weekly profit (income minus tool cost).

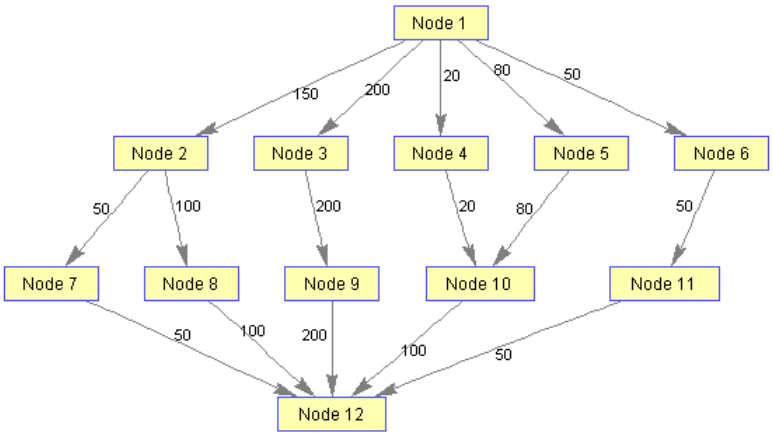
This is an example of the Project selection problem, which can be modeled as a max flow-min cut. First, we must transform the data that we are given into a data structure readable as input for the algorithm. Using the typical PSP architecture, we will nodes out of each job and each tool. Then, we must add a source node, s, and arcs (s, j) such that $j \in J$, the set of jobs with arc capacity p_j. Now, we will add arcs (i, j) such that $i \in J$ and $j \in T$ and tool j is required for the completion of job i with arc capacity ∞. Finally, we must add a sink node t and arcs (i, t) such that $i \in T$ with capacities c_i.

Solving the maximum flow-minimum cut problem on this graph will give us an S-T cut such that the nodes in set S are the nodes that are profitable and the nodes in set T are the nodes that are not profitable. The capacity of the cut will be equal to the cost of purchasing the tools required to complete the jobs in the set S (these are, incidentally, all of the tool nodes in S). To find the expected weekly income, we must then calculate the profit of all of the jobs in the set S and subtract the cost of the tools in the set (the capacity of the cut).

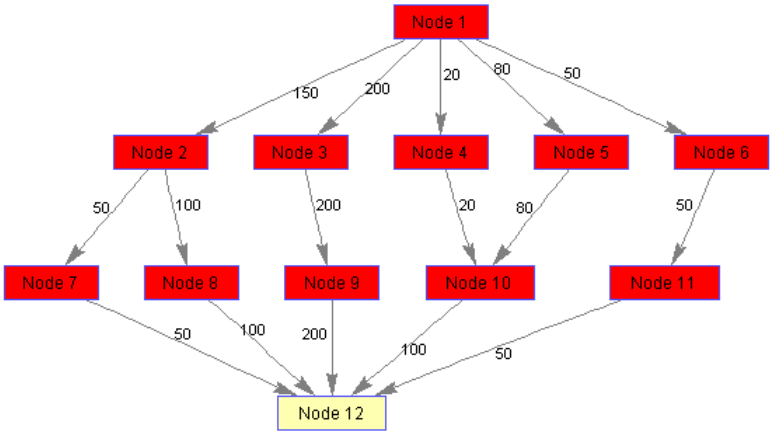
We can begin by representing the flow graph describe above as an adjacency matrix where a_{ij} is the capacity u_{ij} of arc (i, j). We can then use Matlab to produce a biograph of the initial flow.



We can then utilize Matlab to solve the Maximum Flow problem (and thus the Project Selection problem). The resulting maximum flow biograph is below.



While visualizing the graph in this way is helpful, it is only minimally so, since we are interested in the S-T cut and the capacity of the cut. Thus, we can intruct Matlab to highlight the nodes in S and thus make the cut more clear.



We see that set $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$ and $T = \{12\}$ and the capacity of the minimum cut is 500. We know that since all of the tool nodes belong to the set with the source that in order to maximize our profits, we should purchase all of the tools and complete all of the jobs. In a PSP, the cost of purchasing the tools is the value of the minimum cut. In order to find the maximum expected weekly profit, we should calculate the weekly profit of all jobs in the set S and subtract the cost of all of the tools in the set S (the capacity of the minimum cut). Therefore, we have the following

$$\begin{aligned}\mathbb{E}(\text{Income}) &= \sum_{i \in S} p_i - \sum_{j \in S} c_j \\ &= 930 - 500 \\ &430\end{aligned}$$

If the plumber follows our purchase and work schedule, he can expect a weekly profit of 430. We should note, however, that this job schedule has him completeing job 2, which results in a net income of 0 per week (see graph). Depending on the preferences of the plumber, it may be wise to forgoe completing this job (and thus not purchasing tool 3. His expected income would remain 430, however he would have to do less work to achieve this and could devote that time to other pursuits, such as recruiting new job contracts.

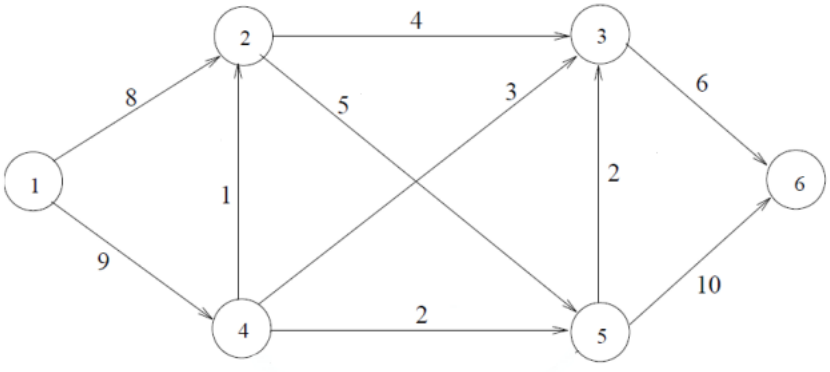
The following Matlab code was used to produce the results.

```
EDU>> S = sparse(A)
EDU>> [U,V,W] = graphmaxflow(S, 1,12);
EDU>> BG= view(biograph(S,data,'ShowWeights','on'));
EDU>> BF= view(biograph(V,[],'ShowWeights','on'));
```

3

(Push-Relabel) Solve Problem 1(a) and (b) from the previous homework using the Push-Relabel algorithm (starting from scratch, i.e., not starting from the flow drawn in the previous homework). You are free to do this by programming the push-relabel algorithm, or by executing the algorithm by hand. If you do the latter, youll need quite a large number of iterations. It is up to you if you want to keep going until the end, or if you want to stop half-way. Feel free also to hand in only a brief summary of how you executed the algorithm, rather than drawings of many residual graphs.

1(a) *The following figure gives an input to the maximum flow problem, and a feasible flow that has been computed by the Push-Relabel (flow is given in box).*



In order to solve this quickly and accurately, we will utilize Matlab to solve using Push-Relabel. The exact code for this function is given below.

Matlab Code:

```
function [flowval cut R F] = max_flow(A,u,v,varargin)
[trans check full2sparse] = get_matlab_bgl_options(varargin{:});
if full2sparse && ~issparse(A), A = sparse(A); end

options = struct('algname', 'push_relabel','fix_diag',1);
options = merge_options(options, varargin{:});

if options.fix_diag, A = A - diag(diag(A)); end
if check, check_matlab_bgl(A,struct('noneg',1,'nodiag',1)); end

if ~trans, A = A'; end
n = size(A,1);

if nargout == 2
[flowval cut] = max_flow_mex(A,u,v,lower(options.algname));
elseif nargout >= 3
[flowval cut ri rj rv] = max_flow_mex(A,u,v,lower(options.algname));
R = sparse(ri,rj,rv,n,n);
if ~trans
R = R';
end
else
flowval = max_flow_mex(A,u,v,lower(options.algname));
end

if nargout >= 4
F = A - R;
end
```

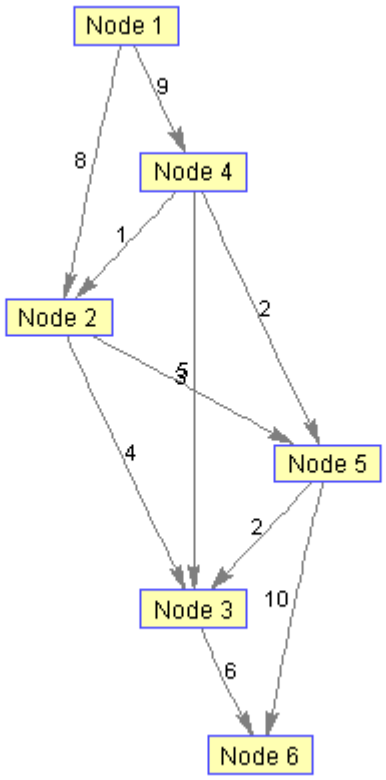
This code requires a sparse matrix an input. However, we will begin by creating an adjacency matrix representing the graph with $a_{ij} = u_{ij}$ where u_{ij} is the capacity of arc (i, j).

$$\begin{pmatrix} 0 & 8 & 0 & 9 & 0 & 0 \\ 0 & 0 & 4 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6 \\ 0 & 1 & 3 & 0 & 2 & 0 \\ 0 & 0 & 2 & 0 & 0 & 10 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

We are now ready to transform this matrix into a sparse matrix so that it is ready for the Push-Relabel algorithm. This is accomplished rather simply using the 'sparse' command in Matlab. The resulting transformed matrix is given below.

(1,2)	8
(4,2)	1
(2,3)	4
(4,3)	3
(5,3)	2
(1,4)	9
(2,5)	5
(4,5)	2
(3,6)	6
(5,6)	10

We can confirm that this is the same flow graph as before by generating a biograph from the sparse matrix using Matlab.

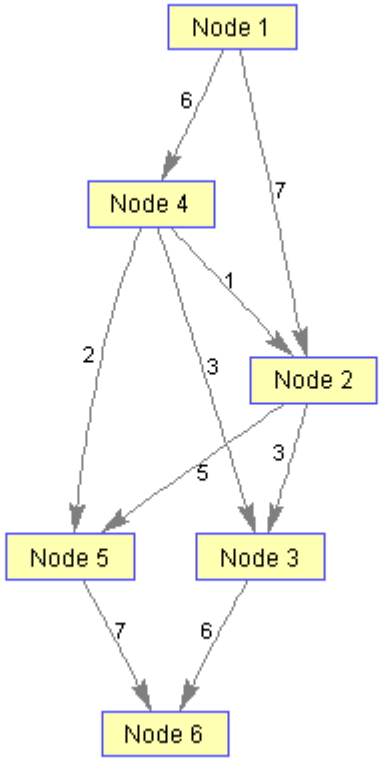


Now, we are ready to execute our Push-Relabel code. We receive the following output, which details not only the max flow, but also the flow on each arc in the optimal flow graph and the members of the set S.

```
13
val =

(1,2) 7.0000
(4,2) 1.0000
(2,3) 3.0000
(4,3) 3.0000
(1,4) 6.0000
(2,5) 5.0000
(4,5) 2.0000
(3,6) 6.0000
(5,6) 7.0000
true true true true false false
```

Therefore we have a maximum flow of 13. This concurs with our original solution in Homework 7. To visualize this data, we can generate a biograph of the optimal flow using Matlab.

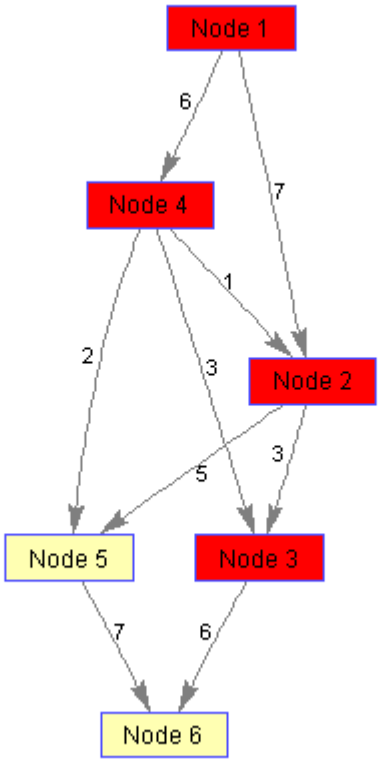


The following Matlab codes were run to solve the Max Flow using the Push-Relabel algorithm.

MatLab Code

```
EDU>> [U,V,W] = graphmaxflow(S, 1, 6);
EDU>> M= view(biograph(V,[],'ShowWeights','on'));
EDU>> h= view(biograph(S,[],'ShowWeights','on'));
EDU>> set(h.Nodes(K(1,:)),'Color',[1 0 0])
```

1(b) *Use the Push-Relabel algorithm to produce a cut of minimum capacity.* Using Matlab, we can visualize the minimum cut using a biograph and using a bi-color scheme. We see that nodes 1,2,3, and 4 belong to one set and nodes 5 and 6 to the other. The capacity of the minimum cut is 13, which is the same as the value of the max flow.



4

(Optional Question) Consider the Ford-Fulkerson augmenting path algorithm for the maximum flow problem, and assume that in every iteration the shortest (in terms of number of arcs) augmenting path in the residual graph ($G(x)$) is selected.

a.) *Show that the length of the shortest path in the residual graph does not decrease from one iteration to the next.*

Now, in order to show that the length of the shortest path in the residual graph does not decrease from one iteration to the next, we must begin by showing that the shortest path Ford-Fulkerson algorithm maintains valid distance labels at each step. A distance label is considered *valid* when it satisfies the following conditions

$$d(t) = 0$$
$$d(i) \leq d(j) + 1 \forall (i,j) \in G(x)$$

We will prove by induction that the shortest augmenting Ford-Fulkerson algorithm maintains valid distance labels at each step. We know that, by design, the algorithm initially constructs valid distance labels. We therefore inductively assume that the distance labels satisfy validity conditions prior to any operation. Now, we need to check whether these conditions remain valid after an augment operation and after a relabel operation.

We know that it is possible that augmenting flow on arc (i,j) might remove this arc from $G(x)$. If this is the case, then the distance labels for this arc are not affected. However, if the augmentation of arc (i,j)

creates an additional arc (j, i) with $r_{ij} > 0$, then an additional condition for validity is created, $d(j) \leq d(i)+1$. The distance labels must satisfy this inequality to remain valid. But, if this is the case than the arc (i, j) must have been considered admissible, meaning that it satisfied the condition that $d(i) = d(j) + 1$. Thus, the distance labels remain valid throughout the augmentation operation.

The relabel operation modifies $d(i)$, which means that in order to remain valid, both the incoming and outgoing arcs at this node must also remain valid with respect to the new distance label, $d^*(i)$. Recall that the relabel operation is performed only when there is no admissible arc at node i. This means that there is no arc $(i, j) \in A(i)$ which satisfies the condition $d(i) = d(j) + 1$ and $r_{ij} > 0$. However, the arc $d(i)$ was valid before relabeling, which means that it satisfied the validity condition $d(i) \leq d(j) + 1$. Therefore, we can conclude that since the relabel process must take place, $d(i) \leq d(j) + 1$ for all arcs $(i, j) \in A$ with $r_{ij} > 0$. The newly relabeled node must then have a distance label $d^* = \min\{d(j) + 1 : (i, j) \in A(i) \text{ and } r_{ij} > 0\}$. Note that the validity conditions are thus preserved by the relabeling operation and that each relabel strictly increases the value of $d(i)$.

Thus the distance from any node i to the sink node t is monotonically nondecreasing over all augmentations.

b.) *Show that the length of the shortest path in the residual graph must increase at least every m iterations, where m is the number of arcs in the graph.*

In part **a.)**, we showed that the augmenting process does not decrease or increase the distance label and only the relabel operation increases the distance label. Now, we know that the algorithm is maintaining an arc list $A(i)$ which contains all the arcs emanating from node i and that each node i has a *current arc* which is the next candidate for admissibility testing. If the algorithm finds that the current arc is inadmissible, it simply removes it from the arc list and designates the next arc as admissible. This process will continue until either the algorithm finds an admissible arc or the arc list is empty. Now that $A(i) = \emptyset$, we can say that there are no admissible arcs since an arc (i, j) that is inadmissible in a previous iteration will remain so until $d(i)$ increases. Therefore, we know that if the arc list is empty, we will need to perform a relabel operation, which we have previously shown strictly increases the value of $d(i)$. Now, because the maximum possible value of the cardinality of the arc list is m, we know that the length of the shortest path in the residual graph, $d(i)$, must increase at least every m iterations.

c.) *Deduce that the total number of augmentations must be $O(mn)$, where n is the number of nodes.*

Let us begin by supposing that the algorithm relabels any node k times. Since we have previously shown that relabeling strictly increases the value of $d(i)$, we are supposing that the algorithm increases the value of $d(i)$ k times. Now, we know that when the algorithm saturates an arc, it reduces its residual capacity to 0. 0. Suppose that augmentation saturates an admissible arc (i, j). Since (i, j) is admissible, we know that

$$d(i) = d(j) + 1.$$

In order to saturate this same arc, the algorithm must move some of the flow off of the arc and send it back along the arc (j, i). This would mean that arc (j, i) was admissible ad the new distance labels must also satisfy the validity condition

$$d^*(j) = d^*(i) + 1.$$

Thus, we see that

$$d^{**}(i) = d^{**}(j) + 1 \geq d^*(j) + 1 = d^*(i) + 2 \geq d(i) + 2.$$

We know that these inequalities must hold because we proved in part **a.)** that the algorithm maintains valid distances at each step. We can use similar methodology to prove that $d^{**}(j) \geq d(j) + 2$. Therefore, between any two consecutive saturations of the arc (i, j), both $d(i)$ and $d(j)$ must be increased by at least 2. This implies that the algorithm can saturate any one arc at most $\frac{k}{2}$ times. Therefore, the total number of saturations would be $\frac{km}{2}$.

Now, in order to show that the total number of augmentations is $O(nm)$, we must next show that each distance label increases at most n times. We know from part **a.)** that each relabel operation at node i

increases the value of $d(i)$ by at least one. Now, after the algorithm relabels any node i n times, we see that $d(i) \geq n$. Therefore, the algorithm will never relabel i again as it cannot be a part of a partially admissible path, that is a path from s to i consisting solely of admissible arcs since for any node k in the partially admissible path $d(k) < d(s) < n$. We can thus conclude that the algorithm relabels any one node at most n times. Coupling this with our previous results, we can conclude that the algorithm saturates at most $\frac{nm}{2}$ arcs (since n is the largest possible k). Furthermore, since each augmentation saturates at least one arc, we deduce that the total number of augmentations is $O(nm)$.

d.) *Deduce that the total running time of the algorithm is* $O(nm^2)$. In order to determine the total running time, we must determine the total running time for the different operations. Now, we have previously shown that the total number of augmentations is $O(nm)$.

This is because the distance label of a node can never increase during augmentation and thus it will never be reexamined during the same augmentation. Now, we know that the natural approach for detecting the shortest path is to perform a breath first search in $G(x)$. By maintaining the list of nodes to be examined as a FIFO queue, then we find the shortest paths by examining the nodes in FIFO order. In each iteration, BFS either finds an admissible arc or it does not. If it does find an admissible arc, then the a new node is marked and added to the list of nodes. If it does not find an admissible arc, then it deletes a marked node from the list. Therefore, since the algorithm marks any node at least once and executes its search for admissible arcs at most $2n$ times. Furthermore, the BFS requires that the arcs in the list $A(i)$ be examine at most once. Therefore, the BFS requires $O(m + n) = O(m)$ time since $m >> n$. Thus, the total running time for augmentations is $O(nm) \times O(m) = O(nm^2)$.

Now, we know that each time we retreat from a node on the partially admissible path we must perform a relabel operation. This implies that the total number of retreat operations is $O(n^2)$ since during one augmentation we can relabel each node at most n times. We also know that each advance operation must add an arc to the partially admissible path while each retreat node must delete an arc from said path. Since the path can have length at most n (the number of nodes), the number of advance operations required is thus the number of retreat operations plus the number of augmentation operations: $O(n^2 + nm^2)$.

Therefore, we can conclude that the total running time of the algorithm is $O(n^2 + nm^2) = O(nm^2)$ since $n^2 << m^2$.

However, we know In part a.) we proved that the minimum distance from each node i to the sink is monotonically nondecreasing over all augmentations, which implies that the total running time per augmentation is $O(n)$, a better bound on the computation time than $O(m)$. Therefore, we can conclude that the total running time is $O(n^2 + n^2m) = O(n^2m)$ since $n^2 << n^2m$.