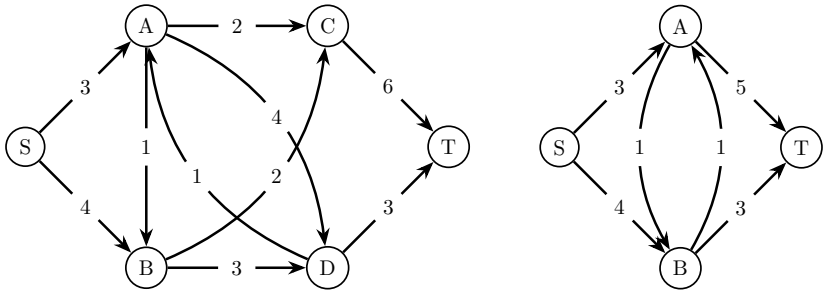
CSCI-570 Homework 4

April 6, 2020

1 Compute Max-Flow And Min-Cut

Solution: Figure 1 has max-flow 7. One feasible flow is $S \to A \to C \to T$ with flow 2, $S \to A \to B \to C \to T$ with flow 1, $S \to B \to D \to T$ with flow 3 and $S \to B \to C \to T$ with flow 1. All min-cuts are $\{S\}||\{A, B, C, D, T\}$, $\{S, A, B, D\}||\{C, T\}$. Figure 2 has max-flow 7. A feasible flow is $S \to A \to T$ with flow 3, $S \to B \to T$ with flow 3 and $S \to B \to A \to T$ with flow 1. All min-cuts are $\{S, B\}||\{A, T\}$, $\{S\}||\{A, B, T\}$.



Rubric: Each subproblem has 5 points. 2 points for correct max-flow value, 1 point for any feasible max flow and 2 points for the two min cuts.

2 Escape From the Building

In this problem, we need to decide whether there is a feasible plan for all the persons in a building to escape when they meet some emergency issues. More specifically, a building is described as an n by n grid and the position of p persons are represented as the integer points $(x_1, y_1), \ldots, (x_p, y_p)$ in the building. Note that to ensure safety, we don't allow any intersection between the paths of any two person. Therefore, your task is to decide whether there exist p vertex-disjoint paths from their starting points to any p different points on

the boundary of the grid. Give an algorithm polynomial in n and prove the correctness of it.

Solution: We construct the network as follows. First, we split each point (i, j) in the grid into two points (i^{in}, j^{in}) and (i^{out}, j^{out}) and add an edge with capacity 1 from (i^{in}, j^{in}) to (i^{out}, j^{out}). In addition, for all (i^{out}, j^{out}), we add directed edges to its adjacent in-node, namely $(i + 1^{in}, j^{in})$, $(i^{in}, j + 1^{in})$, $(i - 1^{in}, j^{in})$ and $(i^{in}, j - 1^{in})$ if they are well defined already. Then, We add a source node S and add an directed edge from S to each start in-node (x_i^{in}, y_i^{in}) with capacity 1, $i \in [p]$ and finally, we add a sink node T and connect all the boundary out-node (i^{out}, j^{out}) to it with a directed edge of capacity 1. Our algorithm is to first compute an integer max-flow of this graph by using Edmund-Karp algorithm. According to our construction, the value of the max-flow is no more than p.

Claim: There exist p disjoint paths if and only if the above network has a max-flow of value p.

Proof: If there exists a max-flow of value p, we can construct p disjoint paths according this flow. Specifically, for each (x_i, y_i), there must be one unit flow from S to (x_i^{in}, y_i^{in}) the finally it will reach the sink, which means it reaches one of the boundary point. This one unit flow therefore indicates a path from (x_i, y_i) to one of the boundary point. Also, these paths will never intersect. If so, there would be two units coming into one in-node. As the corresponding out-node is of capacity 1, it is not feasible.

On the other hand, if the value of the max-flow is not p, we claim that there does not exist such p disjoint paths because if there exists p disjoint paths, we can construct a flow of value p by following these paths. Concretely, for a path $(x_i, y_i) \to (x_i + 1, y_i) \to \cdots \to (x^*, y^*)$ (The last one is the boundary point), we have a one unit flow $S \to (x_i^{in}, x_i^{out}) \to (x_i^{out}, y_i^{out}) \to (x_i + 1^{in}, y_i^{in}) \to (x_i^{out}, y_i^{out}) \to \cdots \to (x^{*^{in}}, y^{*^{out}})$. for each $i \in [p]$. This is a feasible flow as the p paths are disjoint. Therefore, we show the correctness of our algorithm and the runtime is polynomial in n as the complexity of generating graph and translating the solution and Edmond Karp algorithm are all polynomial in n.

Rubric: 5 points for giving correct directed graph with capacity. 4 points for the correct claim. 3 points for the proof for each direction. (If direction and Only-if direction) (Running FF here is correct as it is polynomial time in this case.)

3 Install Software to Your New Computer

Suppose that you have just bought a new computer and you want to install software on that. Specifically, two companies, which you can think of like Microsoft and Apple, are trying to sell their own copy of n different products, like Operation System. Spread Sheet, Web Browser. For each product i, $i \in \{1, 2, \ldots, n\}$, we have

- the price $p_i \geq 0$ that Microsoft charges and the price $p_i' \geq 0$ that Apple charges.

- the quality $q_i \geq 0$ of Microsoft version and the quality $q'_i \geq 0$ of Apple version.

For example, Apple may provide a better Web Browser Safari, but Microsoft a better Word Processor. You want to assemble your favorite computer by installing exactly one copy of each of the n products, e.g. you want to buy one operating system, one Web Browser, one Word Processor, etc. However, you don't want to spend too much money on that. Therefore, your goal is to maximize the quality minus total price.
However, as you may know, the products of different companies may not be compatible. More concretely, for each product pair (i, j), we will suffer a penalty $\tau_{ij} \geq 0$ if we install product i of Microsoft and product j of Apple. Note that τ_{ij} may not be equal to τ_{ji} just because Apple's Safari does not work well on Microsoft Windows doesn't mean that Microsoft's Edge does not work well in Mac-OS. We assume that products are always compatible internally, which means that there is no penalty for installing two products from the same company. All pairwise penalties will be subtracted from the total quality of the system.

Your task is then to give a polynomial-time algorithm for computing which product i to purchase from which of the two companies (Apple and Microsoft) for all $i \in \{1, 2, \ldots, n\}$, to maximize the total system quality (including the penalties) minus the total price. Prove the correctness of your algorithm. (Hint: You may model this problem as a min-cut problem by constructing your graph appropriately.)

Solution: In this problem, we actually need to separate the objects into two parts A (Microsoft) and B (Apple) so that we can maximize the following:

$$\max \sum_{i \in A} q_i + \sum_{j \in B} q'_j - \sum_{i \in A, j \in B} \tau_{ij} - \sum_{i \in A} p_i - \sum_{j \in B} p'_j.$$

With a little calculation, we can transform the above objective function as follows:

$$\begin{aligned}
&\sum_{i \in A} q_i + \sum_{j \in B} q'_j - \sum_{i \in A, j \in B} \tau_{ij} - \sum_{i \in A} p_i - \sum_{j \in B} p'_j \\
&= \sum_{i \in [n]} (q_i + q'_i) - \left(\sum_{j \in B} q_j + \sum_{i \in A} q'_i + \sum_{i \in A, j \in B} \tau_{ij} + \sum_{i \in A} p_i + \sum_{j \in B} p'_j \right).
\end{aligned}$$

Note that $\sum_{i \in [n]} (q_i + q'_i)$ is a constant in this problem. Therefore, to maximize the original objective, we are to minimize the following:

$$\min \left(\sum_{j \in B} q_j + \sum_{i \in A} q'_i + \sum_{i \in A, j \in B} \tau_{ij} + \sum_{i \in A} p_i + \sum_{j \in B} p'_j \right).$$

Then we can model this problem as a min-cut problem by constructing the graph as follows. First, we add a source node s and a sink node t. For each object j, we add one node A_j. We add an edge from S to A_j with capacity p'_i+q_i and add an edge from each A_j to T with capacity $p_j + q'_j$. For the correlation between objects, we add an edge from A_i to A_j with capacity τ_{ij}.

Claim: The above minimization problem has an optimal value v if and only if the above constructed network has a max-flow (min-cut) of value v.

Proof: If there is a cut of value v in the network, then actually this cut contains two parts of nodes. One part contains S and the other part contains T. Then, we just assign the objects corresponding to the nodes belonging to S to A and assign the objects corresponding to the nodes belonging to T to B. For the cut value, if node i is in A and node j is in B, we will cut the edges from S to A_j, A_i to T and A_i to A_j. These edges are of total capacity $p'_j + q_j + p_i + q'_i + \tau_{ij}$, which is exactly the corresponding term in the above objective function. Therefore, we can achieve v by assigning n objects to either A or B.

On the other hand, if the optimal value of the minimization problem is v, then there must be an assignment of the n objects. Then we just cut the network into two parts. One part includes S and objects in A and the other part includes T and objects in B. It is similar to the former direction to show that the cut value here equals v. Based on the above observations, we prove the claim.

Therefore, our algorithm is to first construct the above network, use Edmund-Karp (as here we require polynomial algorithm) to obtain a max-flow of that graph, traverse along the max-flow to obtain a corresponding min-cut and assign the objects belonging to S to Microsoft and the others to Apple. The construction of graph, Edmond-Karp algorithm, traversing and assigning can all be done in polynomial time. The correctness is proved by the above analysis.

Rubric: 3 points for giving correct transformation of the original problem. 3 points for giving the correct directed graph with capacity. 3 points for the correct claim. 3 points for the proof for each direction. (If direction and Only-if direction) (Note: if the students mention Ford-Fulkerson Algorithm with pseudo-polynomial time, give a 2 points deduction.)

4 Jumping Frogs

Somewhere near the Algorithmville, a number of frogs are standing on a number of lotus leaves. As they are social animals (and yes, they are never infected by coronavirus!), the frogs would like to gather together, all on the same lotus leaf. The frogs do not want to get wet, so they have to use their limited jump distance d to get together by jumping from piece to piece. However, these lotus leaves just started to grow, they will get damaged further by the force needed to jump to another leaf. Fortunately, the frogs are real experts on leaves, and know exactly how many times a frog can jump off each leaf before it sinks and become unavailable. Landing on leaves does not damage it. You have to help the frogs find a leaf where they can meet.

In this question, we will get the position of N lotus leaves. For each $i \in [N]$, we know its position (x_i, y_i), the number of frogs n_i on that leaf and the number of jumps m_i before it sinks. The distance between two leaves (x_i, y_i) and (x_j, y_j) is defined as $|x_i - x_j| + |y_i - y_j|$. Design an polynomial algorithm to determine whether each lotus leaf can hold all frogs for a party. The output is an array with length N, containing yes/no solution.

Solution: The trick in this question is the limit of "departure" on each leaf. To model that, we split each node to be "departures" and "arrivals". The exact graph is described as follows:

- We have a source node s.
- We build N arrival nodes a_i. s is linked to every a_i with capacity n_i.
- We build N departure nodes d_i. Each a_i is linked to d_i with capacity m_i.
- For a pair of leaves (i, j), we link d_i to a_j and d_j to a_i, with capacity ∞, if $|x_i - x_j| + |y_i - y_j| \leq d$, $i \neq j$.

Then we run the Edmond-Karp (polynomial time algorithm) algorithm with source node s to sink node a_i to determine that whether the max-flow value equals to $\sum_{i=1}^{N} n_i$. If so, we know that the leaf i is a valid spot. We repeat the max-flow algorithm for all $i \in [N]$, then we can output the yes/no solution for each leaf.

Claim: The max flow value of the graph with source s and sink a_i is $\sum_{i=1}^{N} n_i$ if and only if there exist a way that all frogs gather together.

Proof: (flow value $\rightarrow$ gathering plan) To give a feasible plan from a feasible flow with value $\sum_{i=1}^{N} n_i$, we consider each unit flow as a frog. Surely for each leaf i, there are n_i frog on it, which in the graph means that n_i units flow are sent to it in the feasible flow of value $\sum_{i=1}^{N} n_i$. Then consider leaf i as sink and other leaf j, $j \neq i$, how do we let the n_j frogs arrive at leaf i? For each

$j \neq i$, we follow the flow from a_j to a_i, which we assume as the following k_j paths: $a_j \to d_j \to a_{j_{\ell,1}} \to d_{j_{\ell,1}} \to \cdots \to a_{j_{\ell,k_1} \to d_{j_{\ell,k_1}} \to a_i}$ with $f_{\ell,j}$ units flow for $\ell \in [k_j]$. Then we just let $f_{\ell,j}$ frogs jump from leaf j to $j_{\ell,1}$, to $j_{\ell,2}$ till i for all $j \neq i$. This is a feasible plan as according to the construction of the graph, no more than m_j units of flow can leave from a_j, which means no more than m_j frogs can leave from leaf j. It also makes sure that the frogs gather together at leaf i as the value of the flow is the total number of frogs. So every frog "flows" to leaf i.

(gathering plan $\to$ flow value) For each gathering plan, we can interpret it in the similar way above. Consider a plan that all frogs gather to leaf i. Frogs on leaf j have the following k_j different paths: $a_j \to d_j \to a_{j_{\ell,1}} \to d_{j_{\ell,1}} \to \cdots \to a_{j_{\ell,k_1} \to d_{j_{\ell,k_1}} \to a_i}$ with $f_{\ell,j}$ frogs flow for $\ell \in [k_j]$, then we just send the corresponding value of flow along the path. This will not violate the constraints of the graph as leaf j can only handle m_j leaves. In addition, the flow value is $\sum_{i=1}^{N} n_i$ as the plan is feasible, which means that for each node i, we are sending n_i units of flow.

Rubric: 5 points for giving correct directed graph with capacity. 4 points for the correct claim. 3 points for the proof for each direction. (If direction and Only-if direction) (Note: if the students mention Ford-Fulkerson Algorithm with pseudo-polynomial time, give a 2 points deduction.)

5 Preparing for the Exams

My friend Leo wants to have a emergency plan for his final exams on University of Southern Algorithmville. He has N subjects to prepare for, and for each subject, his score is determined only by the time he spend on learning. It's not surprising that Leo found out he actually spent **zero** time on preparing before.

At least he knows when he can start learning all of these subjects. For each subject i there is a start time s_i when he can get all materials ready to

start learning. And there is also a ending time e_i for each subject, when his learning materials expire and he can't learn anymore. We know that s_i and e_i are integers, and Leo can only dedicate to a single subject within each time phase.

In the University of Southern Algorithmville (USA), a student's total grade is the minimum grade among all subjects. Leo wants you to help him find out the best outcome. Given N subjects and their time intervals (s_i, e_i), design an algorithm to find out the maximum time possible for the least prepared subject. (Hint: It's not enough to use the network flow algorithm alone to determine the answer.)

Solution: Consider that we want to learn for k-unit time for each subject. The problem now become determining the feasibility of learning for k time. We can build the following graph G_k:

- s is the source node.
- (The layer of subject) $p_i, i \in [N]$ are the nodes for each subject. We link $s \to p_i$ for every p_i with a link **with capacity** k.
- (The layer of time intervals) We **mix** all starting points and ending points together to form a set $\{s_i, e_i | i \in [N]\}$ and **de-duplicate**. Finally we **sort** the set to get $M+1$ time points $\tau_1 < \tau_2 < \ldots < \tau_{M+1}$. For example, if we have $\{[1,5],[2,6],[2,4]\}$, after sorting, we have $\tau_1 = 1, \tau_2 = 2, \tau_3 = 4, \tau_4 = 5, \tau_5 = 6$.
- Further, we build the nodes t_j for each interval $[\tau_j, \tau_{j+1}]$, we link an edge from p_i to t_j if and only if the learning time of p_i contains t_j, which means $s_i \leq \tau_j$ and $e_i \geq \tau_{j+1}$. The capacity of the link is $\tau_{j+1} - \tau_j$.
- (Sink node) We build a sink node t and link all t_j to t with capacity $\tau_{j+1} - \tau_j$.

With the graph G_k, we run any maximum-flow algorithm from s to t.

Claim: There is feasible to study for k time if and only if the max-flow value equals $k \times N$.

Proof: (schedule of value k exists $\to$ flow value equals Nk) We convert the schedule explicitly to a flow on G_k:

- We assign flow value k for each edge $s \to p_i$. Because we use all flow capacity from $s \to p_j$, our flow is a maximum flow with value Nk.
- The schedule for each subject i can be represented as the union of a set of intervals with length 1: $\{[t_{i,1}, t_{i,1}+1], [t_{i,2}, t_{i,2}+1], \ldots, [t_{i,k}, t_{i,k}+1]\}$. Let the flow value for each edge $p_i \to t_j$ to be f_{ij}, and the value should be the length of schedule i in the interval j. Formally, it's written as $f_{ij} = |\{t_{i,k} | t_{i,k} \geq \tau_j \text{ and } t_{i,k}+1 \leq \tau_{j+1}\}|$.
- We assign flow value $\sum_i f_{ij}$ from each t_j to the sink node. We observe that $\sum_i f_{ij} \leq \tau_{j+1} - \tau_j$ because the time schedule of each subject should not intersect with each other.

(Flow value equals $Nk \rightarrow$ schedule exists with value k) We only need to look through the link from subject p_i to time interval t_j. We note that if the flow value on the edge is f_{ij}, then we only need to assign f_{ij} time to p_j within that time interval because the capacity of the outer link from t_j have the capacity $\tau_{j+1} - \tau_j$, for every time interval j we have $\sum_i f_{ij} \leq \tau_{j+1} - \tau_j$, which means the plan is feasible. In addition, it is a schedule with value k as all the subjects have a total time k.

Therefore, we can run binary search between $[1, \lfloor(\tau_{M+1} - \tau_1)/N\rfloor]$ to determine the best feasible answer k. For the chosen k, we use Edmond-Karp algorithm to compute whether there exists a max-flow of value Nk, which is in polynomial time.

Rubric: 6 points for giving correct directed graph with capacity. 3 points for the correct claim. 3 points for the proof for each direction. (If direction and Only-if direction) (Note: if the students mention Ford-Fulkerson Algorithm with pseudo-polynomial time, also give full credit to this solution. Even if the number of nodes are pseudo-polynomial (which means the reduction is pseudo-polynomial), give full credits.)

Remark: It is my fault not to clearly point out that the algorithm should be polynomial. To make sure that in your second mid-term, when you want to use NF to solve problems, always make sure that you need to make both your reduction and your algorithm polynomial if it requires you to do so.

6 Help Kumiko!

Kumiko is a member of a high school music group and she is in charge of counting the monthly community fee for each members. This work should be easy but the members are trying to make Kumiko's life harder. They never pay the fee in an integer amount! The following form is what Kumiko has recorded.

	Jan	Feb	Mar	Apr	Total Sum
Reina	13.12	17.04	28.92	16.92	76
Mizore	18.80	20.98	22.95	13.27	76
Shuuichi	0.08	0.98	0.13	107.81	109
Total Sum	32	39	52	138	261

Fortunately, Kumiko finds that the total sum of each month and of each member are integers. To make the table cleaner, she wants to round all data in the table into integers without changing any column or row sum. Therefore, no member is paying more in the table and the amount of money in each month keeps the same! Specifically, each fractional number can be rounded either up or down. For example, a good rounding for the data above would be as follows.

Your goal is to show that Kumiko can ALWAYS do the above process. More specifically, consider there are m members, n months and a data matrix $\{M_{ij}\}_{(i,j)=(1,1)}^{(m,n)}$. The sum of each row and column is integer. You want to round each entry M_{ij} to either $\lceil M_{ij} \rceil$ or $\lfloor M_{ij} \rfloor$ without changing the sum of

	Jan	Feb	Mar	Apr	Total Sum
Reina	13	17	29	17	76
Mizore	19	21	23	13	76
Shuuichi	0	1	0	108	109
Total Sum	32	39	52	138	261

each row and column. Give an polynomial time algorithm to obtain such rounding and show the correctness of it. (Hint: (1). You can check if this kind of rounding is possible by checking whether some flow is feasible. Then show that this flow always exists. (2). You can first consider the case when $M_{ij} \in [0, 1]$ and then generalize it.)

Solution: Suppose a_{ij} is the entry in the i^{th} row and the j^{th} column in the table. We first consider the case when $a_{ij} \in [0, 1]$. We construct the graph as follows. The graph (network) contains one source node S and one sink node T. For every column (month) j, we create a node C_j and for every row (member) i, we create a node R_i. We add an edge from C_j to R_i with capacity 1 for all i and j. In addition, we add an edge from S to each C_j with capacity $S_j = \sum_i a_{ij}$, which is an integer and add an edge from each R_i with capacity $T_i = \sum_j a_{ij}$, which is also an integer.

It is clear to see that the value of the max flow is at most $\sum_{i,j} a_{ij}$.

Claim: If there is an integral max flow of value $\sum_{i,j} a_{ij}$, then Kumiko can have a legal rounding and vice versa.

Proof: First, if there is a feasible rounding method of the original problem, suppose the rounded entry in the i^{th} row and the j^{th} column is $b_{ij} \in \{0, 1\}$. Then, we just send unit b_{ij} flow from C_j to R_i. We send unit $\sum_i a_{ij} = \sum_i b_{ij}$ from S to C_j and send unit $\sum_j a_{ij} = \sum_j b_{ij}$ from R_i to T for all i and j. We can easily check that it is a feasible and max flow.

On the other hand, if there is a max-flow of value $\sum_{i,j} a_{ij}$, according to the construction of the graph, all the edges from S to C_j and from R_i to T are saturated. Then we round a_{ij} to the number of unit flow sent from C_j to R_i. In this way, the sum of the i^{th} row is the outgoing flow from the node R_i, which is the original sum and it is similar for the columns. Therefore, the existence of that flow is equivalent to the existence of possible rounding.

When it comes to general cases, instead of setting the capacity from C_j to R_i to be 1, intuitively we should set it to be $[\lfloor a_{ij} \rfloor, \lceil a_{ij} \rceil]$, which is with lower bound capacity $\lfloor a_{ij} \rfloor$ and upper bound capacity $\lceil a_{ij} \rceil$. The equivalence between the existence of feasible rounding and the existence of a flow of value $\sum_{i,j} a_{ij}$ is similar to the above case. To solve this max-flow problem with lower bound constraints, we can modify it a little bit by setting the capacity of the edge from S to C_j to be $\sum_i a_{ij} - \lfloor a_{ij} \rfloor$ for all j and setting the capacity of the edge from R_i to T by $\sum_j a_{ij} - \lfloor a_{ij} \rfloor$ for all i. It is an equivalent problem as the flow on edge C_j to R_i can only come from s to C_j. Then it is the same problem as the above case. Then, we can just set the capacity of the edges between C_j and R_i. Therefore, we can solve the above mentioned max-flow problem and if the value is $\sum_{i,j} a_{ij}$, we obtain a feasible rounding for each entry (which is the

amount of flow from C_j to R_i for a_{ij}) in polynomial time by using Edmond-Karp algorithm.

Finally, to show that a max-flow of value $\sum_{i,j} a_{ij}$ always exists, we first show that there exists a real-valued flow that achieves that one. This is trivial because by just putting a_{ij} amount of flow from C_j to R_i, we achieve the value $\sum_{i,j} a_{ij}$. Then according to Edmond Karp algorithm, we can find an integral max-flow of the same value, which proves that such rounding is always possible.

Rubric: 3 points for giving correct directed graph with capacity. 3 points for the correct claim. 3 points for the proof for each direction. (If direction and Only-if direction) 3 points for showing that there ALWAYS exists a max flow of value $\sum_{i,j} a_{ij}$. (Note: if the students mention Ford-Fulkerson Algorithm with pseudo-polynomial time, give a 2 points deduction. In addition, if students create a graph that contains pseudo polynomially many nodes, reduce 1 points in the construction of the graph.)

7 Edges that Increase Max-Flow

Given a graph $G = (V, E)$, the source-sink pair (s, t) and capacity of edges $\{c_e \geq 0 \mid e \in E\}$, design a polynomial-time algorithm to find a set of edges S, such that for every edge $e \in S$, increasing c_e will lead to an increase of max-flow value between s and t. Prove the correctness of your algorithm.

Solution: Finding the new given existing one is equivalent to finding a path on the residual network. So we break down the solution into the following steps:

1) Run Edmund-Karp algorithm to get the maximum flow of G. Subtract the flow from the network to get G_{res}.

2) In G_{res}, s is now disconnected with t. Otherwise, according to Edmond-Karp algorithm, we can increase the flow value by following this path. Therefore we can compute the set of connected vertices starting from s and t respectively by BFS or DFS. We name the set of vertices that corresponds to s and t to be V_s, V_t.

3) For all edges $e = (u, v) \in E$, we check that whether $u \in V_s$ and $v \in V_t$. If so, increasing the flow on e will yield a path on the residual network, resulting in the increase of max-flow value between s and t.

The runtime is polynomial as we are running Edmond-Karp. BFS, DFS and checking each edge are also within poly time.

Rubric: 5 points for each step. If students use the algorithm that is trying to increase each edge with capacity 1 and run Edmond-Karp algorithm, reduce 2 points as it is still polynomial in the size the problem but uses far more time than this one.