COMPSCI 311: Introduction to Algorithms **Spring 2022**

Homework 5

Released 4/7/2022 Due 4/20/2022 11:59pm in Gradescope

Name: Brendan Cadogan Collaborators: Ben Tufano, Nare Courchesne, Anthony Ureña
List collaborators on your submission. You are allowed to have at most 4 collaborators. Also list any sources of help (including online sources) other than the textbook and course staff.

If you are asked to give an algorithm, please provide: (a) the pseudocode or precise description in words of the algorithm, (b) an explanation of the intuition for the algorithm, (c) a proof of correctness, (d) the running time of your algorithm and (e) justification for your running time analysis.

Submissions. Please submit a PDF file. You may submit a scanned handwritten document, but a typed submission is preferred. Please assign pages to questions in Gradescope.

1. **(20 points) One Flow, One Cut**
a) Our flow algorithms had some unspecified augmenting path choices, so they might produce different results. Give an efficient algorithm that determines whether a flow network can have just one maximum flow.

A. First we run Ford-Fulkerson to get the a graph with augmenting paths and a max flow. Next we run DFS over the augmenting graph for all components, and if we get a cycle, we know there is more than one max flow, otherwise, there is just one max flow.

B. We knew it had something to do with cycles, but we couldn't figure out how to tie it and the flow network together until we remembered the augmenting graph.

C. So we know that if a graph has more than one max flow, it has to have a cycle. The problem is that a network flow by definition is directed with no edges going into the source node. This means that possible cyles may not be detected by DFS. If we instead take the augmenting flow graph, we now have a graph that has edges going in both ways. Now, let's say we have a network that has more than one max flow. That means, in the augmenting graph, there is at least one cycle that our DFS will find. This is because if there is more than one max flow, there is at least one cycle that has edges on at least one side pointing towards the sink, and the other side with edges pointing to the source. Having a cycle like this means that we could take some flow x from one side and push it to the other side of the cycle ans still maintain the max flow. Let's take a graph with a cycle. It either has all back edges, in which case there is one max flow, or it has both back edges and forward edges. If it has all back edges, there is no way to redirect the flow to have a new max flow. But if it has both back and forward edges in a way that allows DFS to find a cycle, it means we can reroute some value of flow and have a different max flow.

D. The runtime is $O(mnC_{max})$.

E. The runtime is $O(mnC_{max})$ because Ford-Fulkerson is $O(mnC_{max})$, and we don't know the dimensions of the flow network so we can choose any variant we want. The DFS to find a cycle has a run time of $O(m+n)$ which is dominated by $O(mnC_{max})$, so our final runtime is $O(mnC_{max})$.

b)Give an efficient algorithm that takes a flow network and determines if it has just one minimum cut.

A. First we run a version of ford-fulkerson to find the min cut. Next we take one of the edges in the min cut, and increase it's capacity by one. We then run ford-fulkerson again, and if the capacity of the new min cut is the same, we know it has more than one min cut, otherwise, it has just one min cut.

B. We knew we needed to find a way to eliminate the first min cut we found to find if there was a second, but we thought we had to delete an edge. The problem with that is that it can mess up the flow network as a whole, so we decided on increasing a min cut's capacity and see if it remains the min cut.

C. The proof for this is fairly simple. Let's say we have ran ford-folkerson and have found a min cut with capacity x. If we increase the capacity of one of the edges in that min cut, it's new capacity will be x+1. Now, if we run ford-folkerson again, if the new min cut is still x, we know that there is a different min cut because we added one to the capacity of the old one so it can't be found again with a capacity of x. Instead, if we find a min cut with a capacity of x+1, we know that we only have one min cut because if we had at least two, Ford-Folkerson would have found a different min cut with capacity x.

D. The run time is $O(mnC_{max})$

E. The run time is $O(mnC_{max})$ because that is the run time of ford-fulkerson, and all our algorithm does is run ford-fulkerson twice and does a couple constant time operations like comparing the capacity of the first min cut found and that of the second. We could also use other variations of ford-fulkerson because we don't know anything about the network given, so the original ford-fulkerson works just fine. If we knew the typical dimensions of the graph, we might want to choose a different variation to suit the network better.

2. **(20 points) Even Numbers**
You have a matrix $A[1..m][1..n]$ of positive reals, with the property that each row and each column has an even integer sum. Give an efficient algorithm that rounds each matrix element up or down (your choice) to the next even number, keeping the original sum on every row and column, or report this is not possible.

A. There are two main ways to solve this problem. The first is that it is very similar to coloring squares from hw2, so we can either make a similar greedy algorithm to solve it, or just prove that we can reduce it. The other way is through a network flow, which is very similar to the above method. I'll do the greedy way without reducing the hw2 problem even though it has a slightly higher run time than a network flow solution.
For the greedy solution we want to round down every number and take the old sums and new sums of the columns and rows. We then will choose square by square which squares to round up. This means that when we round up a square, it increases the sum of both the row and column by 2, except if the original number was an even integer like 2, because in this problem, 2 rounds up and down to 2. We want to choose which squares get rounded up by which columns and rows have the biggest needs for round ups. We will sort columns and rows by $\frac{sumneeded-currentsum}{2}$ which represents how many round ups we need in that row or column. We then choose the column and row with the most need, and round up that number as long as it hasn't been rounded up before and as long as if rounded up it doesn't stay the same, i.e it wasn't originally an even integer. We then subtract 1 from the that column and row's demand, and then resort the lists so we always have the highest demand column and row at the front. We continue until we either have a working solution, or until it is impossible to round up more because either the rows or columns don't want any more round ups, but the other one does.

B. We were thinking of a network flow solution, but then we realized this was similiar to coloring squares on HW2 so we decided to do that. We also solved the network solution afterwards due to knowing the greedy solution, which is very similiar except your squares are nodes and the edge weights are 1 for round up, 0 for round down.

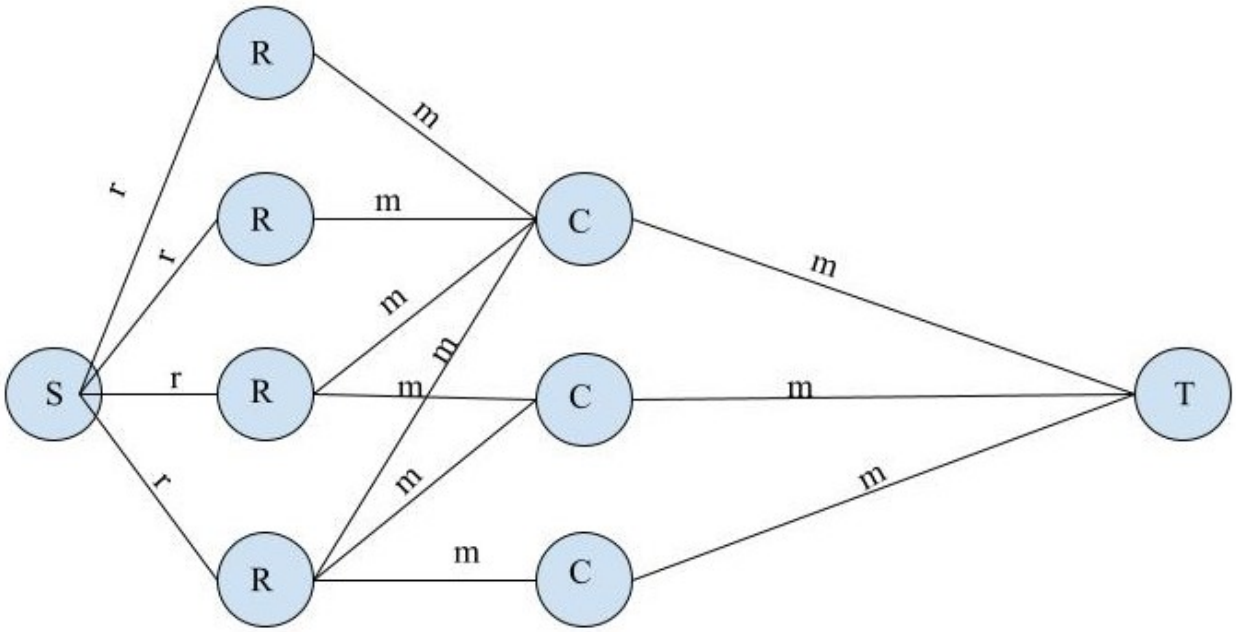
C. Just like HW2, we can use an exchange argument to prove this. Let's call our algorithm A and a different but optimal algorithm O, and an algorithm closer to ours O'. Lets define two squares i and j in which one was rounded up, i and the other one was not, j. Let's define an inversion as when i had a lower desire for rounded up numbers than j, but j was rounded up and not i. Now let's say both i and j are in a sub two by two square of our bigger square. If we move the rounded up from i to j, we are moving it to a row and column that desires more round ups. There are 4 possible amounts of round ups the rows and columns can desire. That is, 1,0,1,0 1,1,1,1 2,1,2,1 or 2,2,2,2. Our new solution O', is still works for these amounts, and is actually the same as O for some of them and therefore is also an optimal solution because it also works.

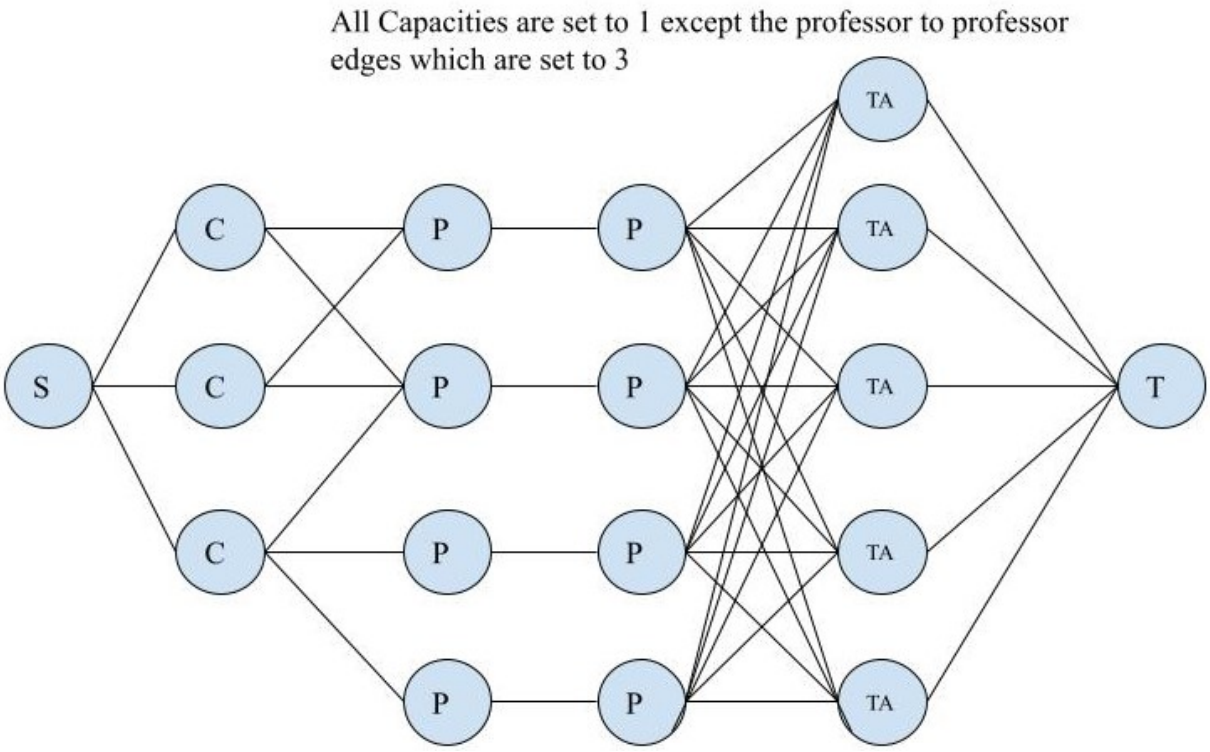
D. The run time is $O(n^2 logn)$.

E. For every row, we need to sort all columns, which is $O(nlogn)$ rounding the row and updating counters is $O(n)$. We do this in every row, for $O(n^2 logn)$ total. Also, at the beginning we round all the numbers down and take the sum of the rows and columns which is $O(n^2)$, but it is dominated by $O(n^2 logn)$ so we can ignore it. The network flow solution, which I did not discuss, would be $O(mnC_{max})$, and since the edges all have a capacity of 1, C_{max} is not that high.

3. **(20 points) Course Management**
A university needs to plan its courses for next semester. It has m courses, with enrollment given by array $C[1..m]$, and r rooms, with capacities $R[1..r]$; Since times are not set yet, each room can be assigned at most one course. Each course needs one professor and one TA; professors have a load of at most three courses. There are p professors, each with a list of courses that they can teach, and t TAs, who can work in any course, but are limited to one. Each professor has specified a list of five TAs that they are willing to work with. Give an efficient algorithm that produces a complete course assignment, or report this is not possible. State your complexity in terms of the given parameters.

A. To do this, we will construct two flow graphs, one for the rooms and courses, and a second for courses, professors, and TAs. There may be a way to do this with one flow graph, but in the runtime analysis I will talk about why it doesn't matter that much. To construct the room and courses network flow graph, we will start with a source node and r nodes, one node for each room. There will be one edge connecting the room nodes to the source node, the capacity for each being the amount of students that can fit in that room. We then will add m course nodes, one for each course. These nodes will be connected by edges to the room nodes with the edge weights being the course capacity. Edges will only be drawn from room nodes to course nodes if that room can fit that course. Next we will draw one edge from each course node to the sink node. The edge weights for these edges will be the course capacity. Next we will run one of the ford-Fulkerson algorithms. I will discuss which one in the runtime analysis, If the max flow is equal to the sum of the course capacity, we have enough rooms and we can now assign professors and TAs. If not, we return that it is impossible. Next we will construct the course, professor, TA flow graph. We will start with a source node with an edge with a capacity of 1 drawn to m course nodes. We then add p professor nodes, with edges with capacities of 1 drawn from the courses to the professors willing to teach them. We then duplicate the professor nodes and draw edges between each professor to their duplicate with a capacity of three. This is so no professor teaches more than 3 courses. We then add t TA nodes, and draw an edge from each professor to the TA nodes that are on their preference list. Finally, we add our sink node and draw an edge with a capacity of one from each TA to the sink node. We then run one of the ford-Fulkerson algorithms, and if the max flow is equal to the amount of courses, we know it is possible, otherwise, it is not possible.





B. Our intuition was that we could turn this to a network flow problem if we found a way to construct the graph.

C. Because we proved ford-Fulkerson in class, we only need to prove that these two flow networks will give us our desired results if we run ford-fulkerson. First we will prove the rooms and course graph. We can do this by contradiction. Let's say we have a possible list of rooms and courses, but our algorithm said it was impossible. This would mean that the maximum flow would be either greater or less than the sums of the course capacity. However, we know it can't be more because the edges from the course nodes to the sink node is a bottleneck with a capacity of the sum of the course capacity. Furthermore, we know it can't be less because by definition of our graph construction, if it is less than there is no possible solution because that means a course didn't get a room. This proves our contradiction wrong, so we know that after running ford-fulkerson on our graph, if the max flow is equal to the sum of course capacities, we have enough rooms.

Now we can use a similar proof to prove our second graph. Let's assume that we have a possible list of courses, professors and TAs, but that our flow chart didn't have a max flow equal to the number of courses. We know that the max flow can't be more than the number of courses because we have a bottleneck from the course to the course nodes that has a capacity equal to the number of courses. But we also know that our graph won't give a max flow less than the number of courses because by definition of our graph construction, if there is a valid grouping of TAs, profs, and course, our graph will find it. This means that if there is a valid answer, our graph will find it. Also, having two nodes for each professor and edges of capacity 3 prohibit professors from teaching more than three edges.

D. The runtime is $O(mnC_{max})$.

E. First we want to determine the runtime for constructing our two flow networks, then we want to determine what variation of Ford-Fulkerson we want to use. Our graphs have 2m+r+2p+t+4 nodes and r+rm+2m+cp+p+5p+t edges, which will take O(mn) to construct. Next we determin how much time ford-Fulkerson takes, which depends on the varient we use. We don't know how many rooms, courses, profs, or TAs we'll have, but we know that there will probably be a small number of them. That means that we should probably use the original Ford-Fulkerson, which has a run time of $O(mnC_{max})$ where C_{max} = the

max possible capacity which for the first graph will be the sum of course capacities, and for the second graph will be the number of courses. Since $O(mnC_{max})$ dominates $O(mn)$, the run time will just be $O(mnC_{max})$.

4. **(20 points) Food Distribution**
A nationwide retailer of canned food has m production plants, each for a different type of food cans, with a current stock of $S[1..m]$ cans. They need to distribute these to n retail stores across the country; the stores want some variety, a minimum $L[1..n][1..m]$ and maximum $M[1..n][1..m]$ of each type of food can per store. For efficient shelf use, each store wants a specific total of cans, given in array $T[1..n]$. You have a map of the directed road network that links plants, stores, and a number p of intermediate warehouses, with processing capacity limited to $C[1..p]$ cans. Cans must pass through exactly one warehouse from plant to any store. Each warehouse processes one specified can type. Each store orders each can type from one specified warehouse. Produce a distribution plan for the cans (how many cans are shipped where), or report that this is impossible.
Produce a distribution plan for the cans (how many cans are shipped where), or report that this is impossible.

A. This problem is very similiar to problem three, but there is one problem, we are give a directed road network that we cannot change, so it may not be a strongly connected component. There is however a simple solution, we learned how to find SCCs in lecture 5, so we can find what nodes are connected, and then construct our flow network! I would also like to mention that I firgured out a way to do this with one iteration of Ford-Fulkerson by setting the min first, but my method still works and doesn't change the big O, so I don't see a reason to retype this.
First we will run DFS to determine SCCs like we did in lecture 5. We have to run DFS over each component otherwise we might not reach all nodes. We will put the nodes that are in each SCC in a list and record what SCCs can reach other SCCs. Next we will construct our flow network, but check if a node can reach the others before we draw edges. First, we will have a source node with an edge drawn to m production plants. The edge capacity will be the current stock for that production plant. Next we have our warehouse nodes. To decide which edges are drawn, we need to look at our SCCs and which cans are supposed to go to which warehouse. Before drawing an edge from a processing plant to a warehouse, we'll look to see if that type of can is processed there. Then we'll look to see if our processing plant is in the same SCC, or if its SCC can reach the SCC of the warehouse. If it can, we draw an edge from the plant to warehouse with an edge capacity of the processing capacity of that warehouse. After this, we need to draw n nodes for the retail stores. Edges will be drawn between the stores and the warehouses that each store has specified for a certain can type. The edge weight will be the minimum amount of cans for that type that the store wants.Finally we will draw an edge from each store to the sink with a capacity of the total cans required for that store. We run Ford-Fulkerson, and if the max capacity is equal to the sum of T, we can return true, if it is less than the sum of minimums, we can return false, and if neither, we need to change the edge weights so we can get up to the max required for each store and still maintain the minimum. To do this, we set the capacity of the store to sink edges to $T - T_{infirstiteration}$. Then we set the warehouse to store edges to $max - min$. Next we need to set the processing plants to warehouse edges to $C - C_{infirstiteration}$. Finally, we set the source to processing plant edges to $S - S_{infirstiteration}$, and run Ford-Folkerson again. We then add of the max flow of the first and second graph, and if they are equal to the sum of T, we return true, if not, we return false.

B. Our intuition was that we needed to construct a flow network, which would be easy if it weren't for the fact that we are given a road network. We then remembered SCCs from earlier in the semester which we decided to use for determining what is connected and not.

C. To prove this, we need to prove that our SCC method works so that we only construct possible flow networks, and we need to prove that our flow network gives us the right answer. The SCC method works because by finding all SCCs and which SCCs have nodes leading into other SCCs, we know what nodes can reach what other nodes. Next, our flow network works because its max flow can't ever be more than the sum of t because the edges from the stores to the sink are a bottleneck, and if it's less than the sum of t, then we haven't met our quota. The edge weights are correct between the first and second iteration of Ford-Fulkerson because we need to run it again only when we have not reached the T for all cans, and we only add more cans for the cans we haven't used yet.

D. The run time is $O(mn^2)$ or $O(m^2n)$ depending on what variant we use.

E. First we look at the run time for determining SCCs. This would be $O(m+n)$ because in lecture 5 we discussed how finding SCCs took a DFS source. Next we look at our network constructions. The first network has $m+n+p+2$ nodes and $m+mp+np+n$ edges. This takes $O(mn)$ to construct. To change the edge capacities from the first to second network, it takes $O(m)$. Since we don't know the amount of cans we'll need, we can use any version of Ford-Fulkerson such as Dinitz or Edmonds-Karp, $O(mn^2)$ and $O(m^2n)$ respectively. Since these dominate $O(mn)$ and $O(m)$ the run time is $O(mn^2)$ or $O(m^2n)$ depending on what variant we use.

Attempt to make all of your algorithms efficient, and provide all elements listed in the instructions.