CSCI270 Homework 7

Jacob Ma

November 8, 2022

Problem 1

When we defined flows in class, we did not rule out that there might be cycles in the flow. Having cycles in your flow seems kind of pointless at best, and a nuisance at worst. Here, you are going to prove a formal version of this intuition, by designing an algorithm that proves that you do not need cycles to get good flows. To be precise, when we say that a flow f "contains cycles", what we mean is that the set of edges with positive flow, i.e., $\{e \in E \mid f_e > 0\}$, contains a cycle.

Your algorithm will be given a graph $G = (V, E)$ with non-negative edge capacities c_e, a source s, sink t, and a valid flow f. Also, there will be no edge into s or out of t; otherwise, the definition of the value of a flow is a bit strange. It is supposed to run in polynomial time (not pseudo-polynomial) and output a new flow f' with $f'_e \leqslant f_e$ for all edges e (so you cannot add flow to any edges), of the same value $\nu(f') = \nu(f)$, and such that f' is acyclic.

Give and analyze a polynomial-time algorithm for finding such an f'.

Algorithm 1: Simplify cyclic flow into acyclic flow

```
Input: Graph $G = (V, E)$, flow $f$, source $s$, sink $t$
Output: New acyclic flow $f'$
1 Detect cycles by applying DFS on $G' = (V\backslash\{s,t\}, f)$, store as $C$ if exists ;
2 while exists cycle $C$ do
3     Initialize bottleneck $(C, f_b)$, $f_b = \infty$ ;
4     for each edges $e$ in $C$ do
5         $f_b = \min_{e \in C} f(e)$, update bottleneck $(C, f_b)$ ;
6     for each edge $e$ in $C$ do
7         $f(e) - = f_b$ ;
8     Perform DFS on $G' = (V\backslash\{s,t\}, f)$, update $C$ if cycle exists ;
9 return $f' = f$
```

Here are some clarification on the algorithm above:

Notice: How we use a DFS traversal to find cycles?

> *Detect Cycle Using DFS.* In a complete DFS implementation, in a certain iteration from *visited* node v, while visiting all its neighbors, if it has a adjacent node u which has already been detected but not finished, there exists a cycle. If all nodes in the graph have been finished, directly return and no cycle exist. If the adjacent

node is not visited, continue DFS on the adjacent node.
All nodes (including the path) within the cycle could be retrieved by iterating the parents of node u, just like how we find path in DFS.
A complete and nice DFS would take $O(m+n)$. □

The correctness proof will be conducted by proving these few lemmas:

(1) The program terminates, and the while loop takes at most $m = |E|$ iterations.

(2) The output f' is a flow.

(3) $\nu(f') = \nu(f)$

(4) The output f' is acyclic.

Lemma 0.0.1

The program terminates, and the while loop takes at most $m = |E|$ iterations.

Proof of Lemma. The only while loop in the program is on line 2, the program terminates when there is no cycle C exist in $G = (V \backslash \{s,t\}, f)$.

We claim that the while loop will take at most $m = |E|$ iterations. Because inside each iteration, at least one edge, bottleneck(C, f_e), will be deducted to 0 since $f_e - f_e = 0$. So in each iteration, at least one edge will be eliminated, thus there will be at most m iterations before we erase every single edge in the net flow (There could not be any cycle if we have erased all the edge).

Thus, the program terminates and the while loop takes at most m iterations. □

Lemma 0.0.2

The output f' is a flow.

Proof of Lemma. We can prove f' is a flow by firstly indicating all $f_e \leqslant c_e$ where c_e indicts capacity for e, and then show $f'^{\text{in}} = f'^{\text{out}}$ for every node u.

Firstly, since through out the algorithm, we are deducting f_b from f_e for some edge e, and $f_e \geqslant f_b$ since f_b is the flow of the bottleneck. We know that there will be no negative flow, and also all new flow $f'_e \leqslant f_e \leqslant c_e$, thus satisfying the capacity condition.
Secondly, for every node u, it could only be changed through the deduction by f_b. However, in every cycle C, there will be one edge into u and one edge out of u, both of them will got deducted. Thus,

$$\begin{aligned} f^{\text{in}}(u) &= f^{\text{out}}(u) \\ f^{\text{in}}(u) - f_b &= f^{\text{out}}(u) - f_b \\ f'^{\text{in}}(u) &= f'^{\text{out}}(u) \end{aligned}$$

The equality relationship will maintain in each iteration, thus satisfying $f'^{\text{in}}(u) = f'^{\text{out}}(u)$ for all u after all iterations. □

i

Lemma 0.0.3

$$\nu(f') = \nu(f)$$

Proof of Lemma. This is trivially true, since that according to lemma 2, the output f' is still a flow, and the algorithm did not perform any operations on any flow out of s and into t. And also s and t could not be involved in *ANY* cycles, since there all the edges into t and out of s are mono-direction. Thus, $\nu(f) = \sum_{e \text{ out of } s} f^{\text{out}}(e) = \sum_{e \text{ out of } s} f'^{\text{out}}(e) = \nu(f')$. □

Lemma 0.0.4

The output f' is acyclic.

Proof of Lemma. Showed in Lemma 3, no cycle will involve s and t.
And if there still exist any cycle in $G' = (V, f')$, the program will not terminate and continue erasing cycles. The program also is shown to be terminated in lemma 1. The lemma follows the contradiction. □

Conclusion: Combining all of the lemma above, the algorithm will perform correctly and outputs a valid correct output f'.

Runtime Anaylsis. Let's consider the runtime of this algorithm. On line 1, based on the reasoning given in the *Notice* part below the algorithm, the DFS detecting cycle part would take $\mathcal{O}(m+n)$.
Based on lemma 1, the while loop will take at most m iterations, and in each iteration, two two for loops each takes $|C| \leqslant |E| = m$ iterations.
In side each iteration on line 5 and line 7, it take constant time for arithmetic operation including comparison, assignment, and subtraction. Thus, these two for loop each takes $\mathcal{O}(m)$ runtime.
However, both the constant initialization on line 5 and two $\mathcal{O}(m)$ for loops are dominated by the DFS search, which takes $\mathcal{O}(m+n)$, based on the reasoning given below the algorithm.
Thus, through at most m iterations in while, each takes $\mathcal{O}(m+n)$, the overall runtime is $\mathcal{O}(m(m+n)) = \mathcal{O}\left(m^2+mn\right)$. □

Problem 2

Imagine that you are starting a food delivery startup with the motto "We know you hate when you must wait. Herewith we state that it's not great when we are late, so it's our fate to not get paid.[a]" The business model is as follows: you have m delivery drivers. Customers place food orders from restaurants of their choosing, and state a deadline of how much maximum they are willing to wait for their food. They pay the restaurant for the food, and possibly you for delivery. If your company accepts an order from a customer, you promise to deliver it within the specified time limit; otherwise, the (flat) delivery fee of \$10 is waived.[b]
Your high-level goal is now to select as many orders to accept as possible (and thus to make as much money

as possible), but subject to not being late on any of them.

To be more precise, you are choosing from among n potential customers. For each customer i, you are told where they live (let's call that location L_i), and where the restaurant is that they are ordering from (let's call that R_i). You are also told how many minutes they are willing to wait; this is a number $t_i > 0$. In addition, you know the locations of your m delivery drivers (let's call the location of the j-th driver D_j).

Each delivery driver can only be assigned at most one order (even though some orders may be close together, or close to where the driver lives). If a driver is in charge of an order, they first drive from their location to the restaurant, where they pick up the food. You can assume that picking up food always takes exactly one minute. From the restaurant, they then drive to the customer's house. You have a route planning software (think Google Maps) into which you or your program can enter any pair of locations (restaurants, customer locations, driver locations) and get a precise (and always completely accurate) answer of how long it will take to drive from the first location to the second. Remember that for each order that you serve on time, you get \$10, and for any other order (which you decline or for which you are late), you get nothing.

Give and analyze a polynomial-time algorithm for maximizing the profit that your company makes. Your algorithm should output both the profit and the assignment of drivers to orders that gives that profit.

[a]There wasn't enough budget to hire someone to come up with a good marketing slogan.

[b]The customer still pays for the food, but since that money goes directly to the restaurant, it doesn't help you.

Algorithm 2: Food Delivery Driver Assignment

```
Input: Drivers $\{D_j\}_{j=1}^m$, Customer Locations $\{L_i\}_{i=1}^n$, Restaurants $\{r_i\}_{i=1}^n$, Allowed Wait Time $\{t_i\}_{i=1}^n$
Output: Driver-Restaurant Assignment & Maximum Revenue
1  Initialize dummy source $s$, and dummy sink $t$ ;
2  Construct a graph $G = \{E, V\}$, where $E = \{(s, D_j)\} \cup \{(D_j, R_i)\} \cup \{(R_j, L_i)\} \cup \{(L_i, t)\}$, and
   $V = \{\{s\} \cup \{D_j\}_{j=1}^m \cup \{L_i\}_{i=1}^n \cup \{L_i\}_{i=1}^n \cup \{t\}\}$, all edges initialize with both flow and capacity = 0;
3  Initialize DR[m][n], RL[n] with value 0 ;
4  for $1 \leqslant i \leqslant n$ do
5      RL[i] = GoogleMap($R_i, L_i$) $\Longrightarrow$ Time used starting form $R_i$ to $L_i$;
6      Set Capacity of $(R_i, L_i) = 1$ ;
7      Set Capacity of $(L_i, t) = 1$ ;
8      for $1 \leqslant j \leqslant m$ do
9          DR[j][i] = GoogleMap($D_j, R_i$);
10         Set Capacity of $(s, D_j) = 1$ ;
11         if $DR[j][i] + 1 + RL[i] \leqslant t_i$ then
12             Set Capacity of $(D_j, R_i) = 1$ ;
13 Run Ford-Fulkerson on $G = \{E, V\}$, find maximum $s - t$ flow $f$ starting from $s$ to $t$ ;
14 return $10 \cdot \nu(f)$;
   Output: Driver-Restaruant Assignment could be found in $G$ where $f_{D_j, R_i} = 1$
```

We are trying to apply Ford-Fulkerson after constructing a valid graph G, which looks something like below:

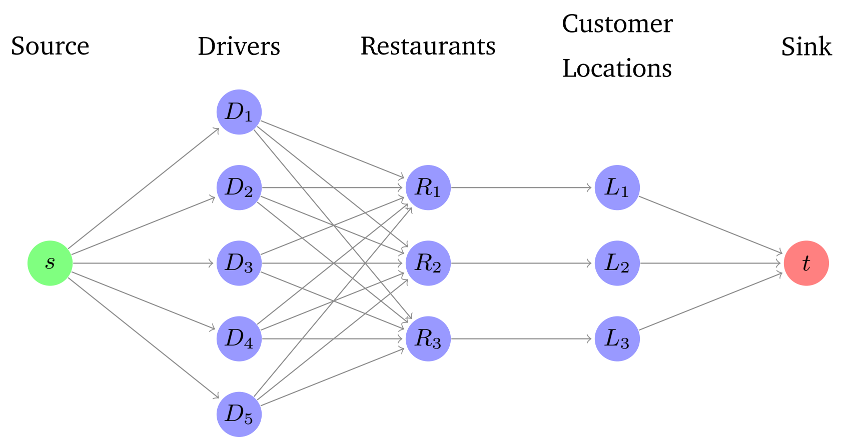

Figure 1: Sample Constructed $G = (V, E)$ with 5 drivers and 3 customers

Graph drawing and LaTeX packages referenced from Qilin Ye to draw this sample graph

In order to prove the correctness of the algorithm, it suffices for us to prove the following lemmas:

(1) The algorithm terminates.

(2) Each driver is assigned to at most one customer/restaurant, and each customer/restaurant is mapped by at most one driver.

(3) Applying Ford-Fulkerson to the constructed graph $G = (V, E)$ gives a valid and best assignment.

(4) The $10 \cdot \nu(f)$ of the max $s - t$ flow is the maximum revenue.

Lemma 0.0.5

The algorithm terminate.

Proof of Lemma. Trivially true, no while loop except Ford-Fulkerson, and Ford-Fulkerson terminates. □

Lemma 0.0.6

Each driver is assigned to at most one customer/restaurant, and each customer/restaurant is mapped by at most one driver.

Proof of Lemma. Firstly, note that the restaurants in the constructed algorithm above does not really stands for distance restaurant locations, it is constructed alongside with Customer Locations with the same size j. And there is a *bijective* relationship between $\{R_i\}_{i=1}^{n}$ and $\{L_i\}_{i=1}^{n}$. So we can pre-calculate the time needed from the restaurant location to the customer location $RL[i]$ beforehand on line 9 even before introducing the concept of drivers.

Secondly, note that during the construction of G, we could also rule out some impossible assignments by *NOT* giving any capacity to the edge $e = (D_j R_i)$ where $DR[j][i] + 1 + RL[i] > t_i$. This means that even if we assign these drivers to the costumers, they can not send the means within the possible tolerated time t_i, thus not making any profit. (In this problem, if a driver *CAN NOT* send any meal in time, it's the same between assigning him to any customer and no customer since there is no profit anyway) And also note that if a capacity of some edge e is 0, it is equally to the situation that no such edge exists in Ford-Fulkerson.

Thirdly, we set the capacity of every edge $e = (s, D_j) = 1$, thus for each D_j, it has at most $f^{\text{in}} = 1$, so it will have at most $f^{\text{out}} = 1$. Note that Ford-Wilkerson will returns a integer flow if we input integer capacities (which we actually did since all possible capacity are 0 and 1), so the possible f^{out} for any D_j will be either 0 or 1. So for each node D_j, there is at most one possible edge out of it could have flow = 1, so each driver will map to exactly one restaurants.
On the other way, consider the restaurants, since they only have one $e = (R_i, L_i)$ as out edge, and its capacity is 1, Ford-Fulkerson will give a valid flow f, making the largest $f^{\text{in}} = f^{\text{out}} =$ Capacity $= 1$. Thus, only at most one driver will be mapped to each restaurants.

Lastly, since $\{R_i\}_{i=1}^n$ and $\{L_i\}_{i=1}^n$ are bijective, combining the second and third point, it also suffies to show that each driver is assigned to at most one customer, and each customer is mapped by at most one driver. □

Lemma 0.0.7

Applying Ford-Fulkerson to the constructed graph $G = (V, E)$ gives a valid and best assignment.

Proof of Lemma. Combining with the lemma, we know we have constructed a graph where all possible assignment have a $s - D_j - R_i - L_i - t$ path with all edges capacity = 1. And we have also shown in the last lemma that each driver is assigned to at most one customer, and one customer is assigned by at most one driver. Thus, by inputting such graph G, Ford-Fulkerson will produce a valid max $s - t$ flow, which is constructed by multiple $s - D_j - R_i - L_i - t$ with $f = 1$. Since this is a max $s - t$ flow and each path has at most $f = 1$, we will have the maximum number of edges, indicating we have the maximum number of possible assignments.
Thus, applying Ford-Fulkerson to the constructed G gives a valid and best assignment.

Note: This also shows that Driver-Restaruant/Customer Assignment could be found in G where $f_{D_j, R_i} = 1$. □

Lemma 0.0.8

The $10 \cdot \nu(f)$ of the max $s - t$ flow is the maximum revenue.

Proof of Lemma. We have shown in the previous lemma that Ford-Fulkerson produces a best assignment, and each $s - D_j - R_i - L_i - t$ with $f = 1$ is an assignment. Thus, the number of such edges = $f^{\text{out}}(s) = \nu(s) =$ the number of maximum possible completed deliveries. Since each delivery makes \$10, the $1t0 \cdot \nu(f)$ of the max $s - t$ flow is the maximum revenue. □

Final Step: Applying all of the lemma above, we know that our algorithm will return the maximum profit and best possible assignment.

Runtime Analysis. In our constructed graph G, the sum of edges could be calculated:

$$\begin{aligned}\sum_{e\in G} c(e) &= \sum_{e\in\{s,D_j\}} 1 + \sum_{e\in\{D_j,R_i\},f_e=1} 1 + \sum_{e\in\{R_i,L_i\}} 1 + \sum_{e\in\{L_i,t\}} 1 \\ &\leqslant m + mn + n + n \\ &= m + 2n + mn\end{aligned}$$

Thus, the sum of edges is $\mathcal{O}(m + 2n + mn) = \mathcal{O}(mn)$.
In our algorithm, line 1 takes constant time. Line 2 constructs all nodes and edges takes $\mathcal{O}m + mn = \mathcal{O}mn$. Line 3 takes $\mathcal{O}(mn)$
From line 4 to line 12, we are iterating through all $\mathcal{O}(m + 2n + mn) = \mathcal{O}(mn)$ edges. On each edges, we are all performing constant time operations, either set capacity, calculating time using Google Map, or run comparison. Thus, the whole block from 4 to 12 runs in $\mathcal{O}(mn)$.
Now consider the runtime of Ford-Fulkerson, we have showed that the maximum possible edges is $\mathcal{O}(mn)$, and $C = \min\left(\sum_{e \text{ out of } s} c_e, \sum_{e \text{ into } t} c_e\right) = \min(|D_j|, |L_i|) = \min(m, n)$. Thus, the runtime of Ford-Fulkerson would be $\mathcal{O}(mn \cdot C) = \mathcal{O}(mn \cdot \min(m, n))$.
Since $\mathcal{O}(mn \cdot \min(m, n))$ dominates $\mathcal{O}(mn)$, the overall runtime is $\mathcal{O}(mn \cdot \min(m, n))$. $\square$