CSCI 270 Homework 7

Qilin Ye

April 29, 2023

1. Removing Cycles from Flow

Solution. The pseudocode goes as follows.

Algorithm 1: Removing cycles from flow

Inputs: directed graph $G = (V, E)$, capacities $\{c_e\}_{e \in E}$, source s, sink t, and flow f.
Notation: $\nu(f)$ denotes the value of flow f; $f(e)$ denotes flow passing of f through edge e.
Run DFS on $V \backslash \{s, t\}$ (or V) to look for cycles.
while there exists a cycle P with positive flow **do**
    Compute the bottleneck $b(f, P) \leftarrow \min_{e \in P} f(e)$.
    $f(e) \leftarrow f(e) - b(f, P)$ **for** each $e \in P$.
    Run DFS on $V \backslash \{s, t\}$ (or V).
Output: $f' = f$.

Before proving correctness, some lemmas:

LEMMA 1 The number of iterations the **while** loop is executed is upper bounded by m.

Proof. Every time the loop is executed, the flow on (at least) one edge in the cycle P is set to 0. There are m edges, after at most m iterations (ignoring the fact that edges incident to s or t will never be part of a cycle), no more cycle exists. END OF PROOF OF LEMMA 1

LEMMA 2 Each iteration of the **while** loop takes (weakly) polynomial time in $m = |E|$ and $n = |V|$.

Proof. The operations involved (comparing minimum and subtraction) take linear time with respect to the number of bits of the largest flow value on an edge, which is further bounded by the number of bits of $\nu(f)$.

Finding the bottleneck and updating edge values both take $|P| = \mathcal{O}(m)$ iterations. This multiplied with the complexity of the arithmetic operations is still (weakly) polynomial, say $\mathcal{O}(Cm)$. DFS on finding cycle takes $\mathcal{O}(m + n)$. END OF PROOF OF LEMMA 2

LEMMA 3 The output f' is a flow.

Proof. Since $f' \leqslant f$ it is clear that f' obeys the capacity condition. For conservation condition, we note that after each iteration, only vertices associated with the path P have been affected. For any such v, both the amount in and out decrease by $b(f, P)$, so by induction, $f^{\text{out}}(v) = f^{\text{in}}(v)$ for all v after any number of iterations. END OF PROOF OF LEMMA 3

LEMMA 4 The DFS of finding a cycle takes $\mathcal{O}(m + n)$ time.

Proof. The runtime and basic idea of DFS is mostly covered in 104/170, so we will outline the process of finding a cycle:

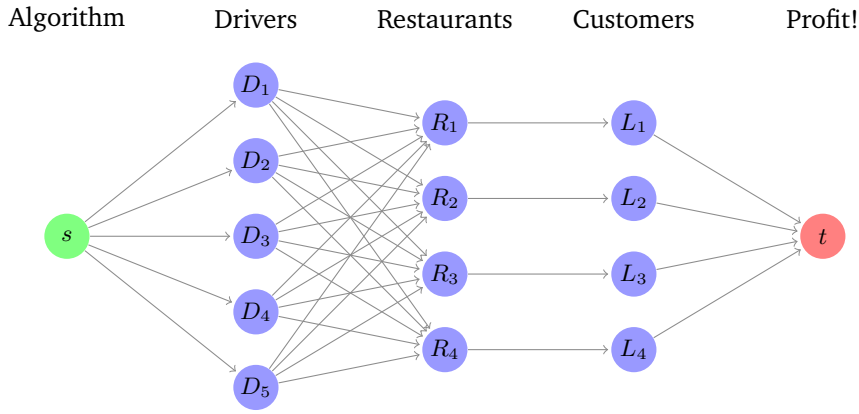
- Iterate over all nodes and keep an array recording the status (visited or not visited) of each node.
- Run DFS any node.
  - Mark starting node as visited. Iterate over all adjacent nodes of the current node.
  - If not visited, recurse and run DFS on this node.
  - Otherwise, if visited, backtrack all its ancestors, output the cycle (along with the edges), and return the cycle.
- If DFS no longer detects unvisited nodes, this means we have exhausted all nodes in the current connected component of the graph. If there are still unvisited nodes in the entire graph, pick any, and repeat DFS.
- Repeat, until all nodes in the graph have been visited. If the algorithm reaches this point, then there is no cycle!

Correctness proof. The algorithm terminates by lemma 1, and f' is a flow by lemma 3. By construction $f' \leqslant f$ on all edges. Since s is a source, it only contains edges pointing out, so no cycle can possibly contain s, meaning $\nu(f)$ remains invariant throughout the algorithm. Hence $\nu(f') = \nu(f)$. Finally, the algorithm terminates when no positive flow cycle exists anymore, so f' is acyclic.
Runtime. By lemma 1 we have at most m iterations, and by lemma 2 each iteration takes $\mathcal{O}(m + (m+n)) = \mathcal{O}(m+n)$. Therefore the total runtime is $\mathcal{O}(m(m+n))$, assuming constant time arithmetic operation.

2. Food Delivery Optimization

Solution. We visualize the problem by converting it into a max-flow problem.

- We define five layers of nodes. Notation-wise, we call them the source layer, the driver layer, the restaurant layer, the customer layer, and the revenue layer, respectively. Let $T(a, b)$ denote the precise time required to travel from a to b. Having obtained a directed graph, we now assign capacities to each edge, transforming it into an integer flow network.
- Set the capacity of all edges of form (s, D_i) to 1.
- Set the capacity of all edges of form (L_i, t) to 1.
- For each i, set the capacity of (R_i, L_i) to 1.
- For each pair (i, j), set the capacity of (D_j, R_i) to
  - 1 if the $T(D_j, R_i) \leqslant t_i - T(R_i, L_i) - 1$;
  - 0 otherwise.
- Run Ford-Fulkerson and obtain flow f. Return 10 times the value of the flow [the profit], as well as pairs (j, i) such that $f((D_j, R_i)) = 1$ [the specific driver assignment].



In a nutshell, we represent an assignment (driver j, customer i) by the path $s - D_j - R_i - L_i - t$ with flow 1. Below is why.

Explanation. Before even attempting to assign drivers to customers, we need to rule out the obviously infeasible ones, namely, the assignments (driver j, customer i) that lead to a delayed delivery. This happens if and only if $T(D_j, R_i) + 1 + T(R_i, L_i) > t_i$, i.e., driver j cannot deliver food to customer i on time because of long distance needed to travel. By setting such edges to have capacity 0 we force our max flow to avoid such edges. All other edges in this directed graph are reasonable, so we set their capacity to 1 and below explains why.

Since the question also dictates that one driver can be assigned at most one order, for each D_j, there can be at most one nonzero flow leaving D_j. Further, nonzero flow leaving D_j can only happen if driver D_j is assigned. These are done by setting all edges (D_j, R_i) to have capacity 1 and limiting the flow into D_j to also be 1 by setting the capacity of (s, D_j), the only edge ending at D_j, to be 1. If so, driver j is assigned customer i if and only if (i) driver j is assigned to *some* customer, corresponding to $f((s, D_j)) = 1$, and (ii) *that* customer is customer i, corresponding to $f((D_j, R_i)) = 1$ and $f((D_j, R_\ell)) = 0$ for all $\ell \neq i$, as driver j needs to drive to restaurant i.

The rest is straightforward. Since for each R_i, the flow in is at most 1, we simply set all outward edges, namely (R_i, L_i), to have capacities 1, and similarly for edges (L_i, t).

If we decide to serve customer i by assigning some driver j, then $f((R_i, L_i))$ and $f((L_i, t))$ are 1, and we obtain an $s - t$ flow of 1, corresponding to earning $\$10$. If we decide to not serve a customer, these two edges are 0. Therefore, total revenue $= 10\nu(f)$.

Pseudocode.

Algorithm 2: Delivery driver assignment

Inputs: drivers $\{D_j\}_{j=1}^m$; customer locations $\{L_i\}_{i=1}^n$, patience $\{t_i\}$, and restaurants $\{R_i\}_{i=1}^n$; & Google Maps[??], a mapping of (location × location) $\to \mathbb{R}$.

Initialization. 1-indiced matrices $A = M_{m\times n}(0)$, $P = M_{m\times n}(0)$, and $D = M_{n\times 1}(0)$. (Acronyms for Ans, Pickup, & Delivery, respectively.)

Initialization. Source node s, sink note t. Weighted edge set $E = \{(s, D_j)\} \cup \{(D_j, R_i)\} \cup \{(R_i, L_i)\} \cup \{(L_i, t)\}$ with weights/capacities 0. Vertex set $\{s\} \cup \{D_j\} \cup \{R_i\} \cup \{L_i\} \cup \{t\}$. Directed graph $G = (V, E)$.

for $1 \leqslant j \leqslant m, 1 \leqslant i \leqslant n$ **do**

  $P[j][i] \leftarrow$ Google Maps(D_j, R_i).

  $D[i][1] \leftarrow$ Google Maps(R_i, L_i).

  Weight/capacity of $(s, D_j) \leftarrow 1$.

  Weights/capacities of $(R_i, L_i), (L_i, t) \leftarrow 1$.

  if $P[j][i] \leqslant t_i - 1 - D[i][1]$: weight/capacity of $(D_j, R_i) \leftarrow 1$.

Run Ford-Fulkerson on G with source s, sink t. Let f be the resulting flow.

Output: $10 \cdot \nu(f)$ as **revenue**.

Output: $\{(j, i) : f((D_j, R_i)) = 1\}$ as **assignment**.

Correctness proof follows from the explanation above as well as that of the Ford-Fulkerson algorithm.

Runtime. By our construction, the sum of capacities of edges leaving s is $|D_j| = m$. The total number of edges is upper bounded by $m + mn + 2n = \mathcal{O}(mn)$, where we count the number of maximum edges between each "layer." Ford-Fulkerson therefore terminates in $\mathcal{O}(m \cdot mn = m^2 n)$. Our **for** loop does constant work in each iteration with a total of $\mathcal{O}(mn)$ iterations, which is dominated by $\mathcal{O}(m^2 n)$. Hence the total runtime is $\mathcal{O}(m^2 n)$.

3. Two Degrees of Chocolate

Solution. Consider a bipartite matching problem between the n actresses and the n actors, with the existence of an edge indicated by whether the two have appeared in the same movie. **We claim that player P_0 wins the game if and only if there does *not* exist a perfect matching.**

We first show $\Rightarrow$. If there exists a perfect matching $\{(x_i, y_{\sigma(i)}\}_{i=1}^n$, then player P_1's strategy is to simply name actor $y_{\sigma(i)}$ every time P_0 names some actress x_i. After a round, P_0 either loses because they fail to find an actress not yet mentioned who has co-starred with the actor $y_{\sigma(i)}$ picked by P_1 in the latest round, or P_0 can, and the game continues. But after n rounds, there are no actresses left, and P_0 is forced to lose!

The converse is significantly more involved. Let G be a maximum matching. By assumption G is not perfect, so there exists some actress x_0 unmatched to any actor. We claim P_0's winning strategy is by naming x_0 first. Intuitively, the idea is that, after P_0 has picked x_0, there are two possibilities:

- x_0 has never co-starred with any actor, so P_0 instantly wins, or
- x_0 has co-starred with some actors. P_1 picks one such y_0. The fact that (x_0, y_0) is not an edge in G and G is maximal implies y_0 must be paired with some other actress x_1 in G. Hence P_0 picks x_1, and the game continues.

More formally, we prove the following induction statement, the base case of which is listed in the bullet points above.

> LEMMA. If P_0 and P_1 have collected generated the sequence of actresses and actors $x_0, y_{\sigma(0)}, ..., x_{k-1}, y_{\sigma(k-1)}, x_k$, then (i) either P_1 is unable to append an actor to the sequence, resulting in P_0 winning, or (ii) P_1 appends an actor $y_{\sigma(k)}$, but there exists a not-yet-mentioned x_{k+1} such that $(x_{k+1}, y_{\sigma(k)}) \in G$.

Proof of lemma (i.e. inductive step). It suffices to show that if P_1 is able to follow up with an actor $y_{\sigma(k)}$, then $y_{\sigma(k)}$ must appear in G. By induction hypothesis, $x_1, ..., x_k$ are all actresses in the maximal matching G, and they are matched to $y_{\sigma(0)}, ..., y_{\sigma(k-1)}$. Call this collection of k pairs S.
If the claim were not true, then there exists $y_{\sigma(k)}$ not in G but is nevertheless connected to x_k. We are now able to augment S and obtain a even larger matching than G:

$$|\underbrace{\{(x_0, y_{\sigma(0)}), \cdots, (x_k, y_{\sigma(k_0)})\}}_{(k+1) \text{ pairs}} \cup (G \backslash S)| > |\underbrace{\{(x_1, y_{\sigma(0)}), \cdots, (x_k, y_{\sigma(k-1)})\}}_{k \text{ pairs}} \cup (G \backslash S)|,$$

contradiction! END OF PROOF OF LEMMA

Back to the problem. If the game terminates at round $j < n$, then straight from the lemma this implies P_1 is forced into case (i) at round j, thereby losing the game. If the game terminates at round n[1], then it means $x_1, ..., x_{n-1}$ are all in G. And if $y_{\sigma(0)}, ..., y_{\sigma_{(n-2)}}$ are all in G and x_0 is not, then the last available actor, $y_{\sigma(n-1)}$, must also not be in G. But this violates the lemma. Therefore, if there does not exist a perfect matching between the actresses and the actors, the game *must* end before the n^{th} round starts. And as we have shown, P_0 wins.

As for algorithm — I believe we both know what it is now. I will append it for the sake of completeness, though.

Algorithm 3: Two degrees of chocolate

1 **Inputs**: actresses $\{F_i\}_{i=1}^n$, actors $\{M_i\}_{i=1}^n$, & adjacency info/edges $\{(F_i, M_j)\}$ such that F_i and M_j have appeared in the same movie.
2 **Initialization**. Source s, sink t. Vertex set $G = \{s\} \cup \{F_i\} \cup \{M_i\} \cup \{t\}$.
3 **Initialization**. Edge set $E = \{(s, F_i)\} \cup \{(F_i, M_j)\} \cup \{(M_j, t)\}$. Weighted directed graph $G = (V, E)$.
4 **Initialization**. **for** edge $e \in E$: set weight to 1 if adjacent to s or t; ∞ else (i.e. if of form (F_i, M_j)).
5 Compute $\nu(f)$ where f is the output of Ford-Fulkerson on G with source s, sink t.
6 **Output**: P_0 wins if $\nu(f) < n$; P_1 wins otherwise. □

[1] Of course the game must end by round n... C'mon, this is a chocolate problem. Surely I don't need to prove that? :)