Homework 7

Winston (Hanting) Zhang

Friday, November 4, by 23:00

General Instructions. The following assignment is meant to be challenging, and we anticipate that it will take most of you at least 10–15 hours to complete, so please allow yourself plenty of time to work on it. We highly recommend reading the entire assignment right away — you never know when inspiration will strike. Please provide a formal mathematical proof for all your claims, and present runtime guarantees for your algorithms using asymptotic (big-$O/\Omega/\Theta$) notation, unless stated otherwise. You may assume that all basic arithmetic operations (multiplication, subtraction, division, comparison, etc.) take constant time.

Collaboration. Please carefully check the collaboration policy on the course website. When in doubt, ask an instructor.

Consulting outside sources. Please carefully check the policy regarding outside sources on the course website. Again, when in doubt, ask an instructor.

Submission. Homework submission will be through the Gradescope system. Instructions and links have been provided through the course website and Piazza. The only accepted format is PDF. Starting with this homework, only typed solutions will be accepted, and we highly recommend producing your solutions using LaTeX (the text markup language we are also using for this assignment).

Recommended practice problems (do not hand in): KT Problems 7.6, 7.7, 7.9, 7.12, 7.13, 7.16, 7.19, 7.22, 7.23, 7.25, 7.27, 7.35

Problem 1. [15]

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., $\{e \in E\} f_e > 0$, 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 $f'_e \leq f_e$ 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 Remove cycles from flow.

```
procedure FIND-CYCLE-AT(G, f, s)
  let q ← {s} (queue with one element)
  let visited ← {} (empty set)
  let prev ← {} (empty map)
  while not q.empty do
    let v ← q.pop
    visited[v] = true
    for each outgoing neighbour w of v do
      if w = s then
        let C ← BACKTRACK(prev, w)
        return C
      if f_(v,w) ≠ 0 and not visited[w] then
        q.push(w)
        prev[w] ← v
  return null

procedure REMOVE-CYCLE-AT(G, f, C)
  let e* ← edge with minimum cost in C.edges
  for e in C.edges do
    let f_e ← f_e − f_e*
  return f

procedure REMOVE-CYCLES(G, f)
  for v in V do
    while let C ← FIND-CYCLES-AT(G, f, v) ≠ null do
      let f ← REMOVE-CYCLE-AT(G, f, C)
  return f
```

Definition 1. *Let f, f' be flows on G. Define $f \leq f'$ if $f_e \leq f'_e$ for all $e \in E$.*

Proposition 1. *Let f be a valid flow on G. Let $E_f = \{e \in E : f_e > 0\}$ be the edge set of f.* FIND-CYCLES-AT*(G, s) outputs a cycle in G containing the vertex s (if it exists) in $O(E_f)$ time complexity.*

Proof. This is a rare *proof by piazza post*. Kempe states in a post: "If you want to detect cycles, you need to outline it at a level of detail that a good 104 student could implement it from your description."

A good 104 student should recognize that FIND-CYCLES-AT(G, f, s) is a BFS procedure. The condition for termination is hitting the start node s again in our search, at which point we backtrack through a map of previous pointers to construct a cycle and return it.

We point to Kempe's comment again and omit a full proof of the follow claims. The cost of searching with BFS is bounded by the edges that must be exploed. Note the **key** condition that we only explore edges which have **postive** flow. Thus BFS runs in $O(E_f)$ and not $O(E)$. When the termination condition is triggered, the cost of constructing individual cycles is bounded by E_f, since we can backtrack at most E_f times. Hence the total runtime complexity is $O(E_f) + O(E_f) = O(E_f)$, as desired. □

Proposition 2. *Let f be a valid flow on G. Given $v \in V$, let $f' =$ REMOVE-CYCLE-AT(G, f, C). Then*

1. *f' is a valid flow on G;*
2. *$f' \leq f$;*
3. *f' has equal value to f, i.e. $\nu(f') = \nu(f)$;*
4. *REMOVE-CYCLES-AT(G, f, v) runs in $O(E_f)$.*

Proof. **Correctness:** We analyze the loop that modifies f into f'. Consider each node w in C. Since C is a cycle, we know that w has exactly two edges e_{in} and e_{out} going in and out. When we loop through all edges in C in our algorithm, we subtract f'_{e^*} from e_{in} and e_{out}, setting $f'^{\text{in}}(w) = f^{\text{in}}(w) - f'_{e^*}$ and $f'^{\text{out}}(w) = f^{\text{out}}(w) - f'_{e^*}$. By assumption, f is a valid flow! Thus $f^{\text{in}}(w) = f^{\text{out}}(w)$, and hence we may conclude that $f'^{\text{in}}(w) = f'^{\text{out}}(w)$. This hold for all vertices w which we modify within C, showing that f' conserves the in-out flow at each vertex.

Next, we are substracting flow, so clearly $f' \leq f$. Furthermore, since f'_{e^*} is the minimum flow in C, we can guarantee that all modified edges are still non-negative. Thus this shows that f' is valid and $f' \leq f$.

By assumption, G has two distingushed nodes s and t which have no incoming and outgoing edges, respectively. Since s has no incoming edges, it cannot be a part of any cycle. Thus REMOVE-CYCLES-AT(G, f, C) does not modify the flow out of s – this exactly implies $\nu(f') = \nu(f)$.

These are exactly the first three conditions that completes the correctness proof.

Termination and runtime: We are simply looping with a finite number of iterations, REMOVE-CYCLE-AT(G, f, C) terminates. Note that our cycle length is bounded by E_f, so in fact we have a runtime of $O(E_f)$. □

Proposition 3. *Let f be a valid flow on G and $f' =$ REMOVE-CYCLES(G, f). Then*

1. *f' is a valid flow on G;*
2. *$f' \leq f$;*
3. *f' has equal value to f, i.e. $\nu(f') = \nu(f)$;*
4. *f' is acyclic.*

Proof. Index the vertices $v_1, \ldots, c_n$. We follow the execution of the loop and define a sequence of modifications to the original flow f as follows. Start with $f_0 = f$, then at each iteration i of the loop, we modify f_{i-1} an undetermined k_i times with the while loop and produce a new flows $f_{i,j}$. Denote the resulting flow at the end of iteration i by $f_i = f_{i,k_i}$. Note that we end with $f_n = f_{n,k_n} = f'$ (which we output). We claim that for every $1 \leq i \leq n$, f_i is a valid flow such that $f_i \leq f$, $\nu(f_i) = \nu(f)$ and f_i does not contain any cycles containing v_j for $j \leq i$.

We proceed by induction on i. The base case $i = 0$ is trivial: $f_0 = f$ is obviously a valid flow, $f \leq f$, $\nu(f) = \nu(f)$, and f does not contain any cycles with vertices that do not exist.

Now assume for the sake of induction that f_{i-1} is a valid flow such that $f_{i-1} \leq f$, $\nu(f_{i-1} = \nu(f)$, and f_{i-1} does not contain any cycles containing v_j for $j \leq i-1$. We want to show that f_i is a valid flow such that $f_i \leq f$, $\nu(f_i = \nu(f)$, and f_i does not contain any cycles containing v_j for $j \leq i$.

Indeed, the body loops while v still has a cycle, starting with f_{i-1} and then at each iteration setting $f_{i,j} =$ REMOVE-CYCLES-AT$(G, f_{i,j-1})$. By Proposition 2, we can conclude that the chain $f_{i-1} = f_{i,0}, f_{i,1}, \ldots, f_{i,k_i} = f_i$ is a chain of valid flows such that $f_{i,0} \leq f_{i,1} \leq \ldots \leq f_{i,k_i}$ and $\nu(f_{i,0}) = \nu(f_{i,1}) = \ldots = \nu(f_{i,k_i})$. From parts 4 of Proposition 2, we may also conclude that the while loop only terminates with f_i when f_i does not contain any cycles containing v_i. Thus, applying out induction hypothesis, we know f_i does not contain any cycles containing v_j for $j \leq i-1$ plus v_i itself. This completes the induction.

Thus $f_n = f'$ is a valid flow such that $f' \leq f$, $\nu(f') = \nu(f)$ and f' does not contain any cycles, as desired.

Termination and runtime: This is more tricky. Ignore the outer for loop for now and only consider the effects of the inner while loop. Every time we execute that loop, we guarantee removing one cycle by setting the flow in one edge to 0. Hence, we can at maximum only run REMOVE-CYCLE-AT(G, f, C) $|E| = m$ times before the flow in all edges are set to zero, and FIND-CYCLE-AT(G, f, s) can no longer run. (Because it runs in $O(E_f)$, and that goes to 0 when f has no flow.) Thus the inner while loop executes at most m times. By Proposition 2, the runtime of REMOVE-CYCLE-AT(G, f, C) is $O(E_f)$, which is bounded by $O(E) = O(m)$. Thus the total runtime is $O(m \cdot m) = O(m^2)$. Clearly this terminates. □

Problem 2. [15]

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.[1]" 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.[2] 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.

Algorithm 2 Maximize profit from deliveries.

procedure MAXIMIZE-PROFITS(L, R, D, t)
  let $V \leftarrow \{s, t\} \cup \{v_j : 1 \leq j \leq m\} \cup \{w_i : 1 \leq i \leq n\}$
  let $E \leftarrow \{\}$ (adjacency matrix of edges)
  let $c \leftarrow \{\}$ (empty map of capacities)
  for $1 \leq j \leq m$ **do**
    E.insert(s, v_j)
    $c[(s, v_j)] \leftarrow 1$
  for $1 \leq j \leq m$ **do**
    for $1 \leq i \leq n$ **do**
      if dist(D_j, R_i) + dist$(R_i, L_i) + 1 \leq t_i$ **then**
        E.insert(v_j, w_i)
        $c[(v_j, w_i)] \leftarrow 1$
  for $1 \leq i \leq n$ **do**
    E.insert(w_i, t)
    $c[(w_i, t)] \leftarrow 1$
  let $f \leftarrow$ FORD-FULKERSON$((V, E), c)$
  return $10 \cdot \nu(f)$, matchings in f

Proposition 4. MAXIMIZE-PROFITS(L, R, D, t) *returns the matching which gives the maximial profit.*

Proof. **Correctness:** The idea is to build a graph which represents this delivery problem and then run Ford-Fulkerson to obtain the "maximum" flow that we can achieve. Indeed, consider the same setup as we have from the maximum bipartite graph matching problem. We label each driver j with the vertex v_j and each customer i with the vertex w_i.

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

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

Then we introduce the source s and the sink t. This $V = \{s, t\} \cup \{v_j : 1 \leq j \leq m\} \cup \{w_i : 1 \leq i \leq n\}$ – this is the first step of our algorithm.

Now we need to assign the edges. First off, we connect every driver to s and every customer to t. Next, we need to add all possible matchings of drivers to customers as edges. Naturally, we iterate over each driver and each custom in a double for loop. However, we are constained by the fact that customer i will only pay us if driver j arrives before t_i. Driver j must first drive from D_j to R_i, then wait exactly 1 minute for food, then drive from R_i to L_i. If it is the case that this is within the allotted time, i.e. $\text{dist}(D_j, R_i) + \text{dist}(R_i, L_i) + 1 \leq t_i$, then we must consider that as a possible matching, and hence add (v_j, w_i) into E with capacity 1. Otherwise, it is no possible for driver j to deliver to customer i, so we can ignore the edge. This decision is exactly reflected is the algorithm.

After constructing V and E, simply run Ford-Fulkerson on $G = (V, E)$ with capacities c to obtain some flow f. Given f, we simply take all the edges between $e_{ji} = (v_j, w_i)$ such that $f_{e_{ji}} = 1$ to recreate the desired matching M. We claim that this construction finds the best possible matching of deliveries. Indeed, assume for the sake of contradiction that we failed and there was a better set of matchings M'. This matching does not violate any of the time constraints t_i, so we know $M' \subseteq E$. But that would imply there is a flow $f' \geq f$, contradicting the correctness of Ford-Fulkerson! Thus f must be maximal, and our algorithm outputs $10 \cdot \nu(f)$ and M.

Termination and runtime: We already know that Ford-Fulkerson terminates. Other than that, our algorithm is not recursive and simply executes finite loops, which must terminate. Initializing V, E, and c costs $O(m + n)$, $O(|V|^2) = O((m+n)^2)$, and $O(1)$, respectively. The three loops that run have m, mn, and n iterations, respectively. Each have bodies which are simple assignments that take $O(1)$ time. Thus the total runtime of the loops are $O(m + mn + n) = O(mn)$. The final call to Ford-Fulkerson costs $O(F \cdot (m + n))$, where F is the total capacities of edges out of s. There are m edges out of s all with capacity 1, so we have $F = m$ and runtime $O(m(n + m))$. Thus our **total** runtime is $O((m + n)^2) + O(mn) + O(m(m + n)) = O((m + n)^2)$[1]. □

Problem 0.

Chocolate Problem: 1 chocolate bar

Problem 7.51 from the textbook.

[1] Actually, I must digress, this runtime is naive in that we represent E with an adjacency matrix. Because E is sparse and its structure is known (bipartite between v_j and w_i), it would be much for efficient to encode the edges with an adjacency list. The problem is that Ford-Fulkerson assumes $O(1)$ edge access, which can only be guaranteed with an adjacency matrix.