CS 170: Efficient Algorithms and Intractable Problems

Spring 2017

●

Homework 7

Due on April 4 at 12:00pm

●

Solutions by

Michael Fan

26697596

In collaboration with

none

Problem 1: Linear Programming Fundamentals

★ Level

For each of the following optimization problems, is there a finite optimal solution that can be found by an LP solver? If the answer is no, identify the reason why (i.e. unbounded, infeasible, or not a linear program).

(a)

$$\max\ 5x + 3y$$

$$\begin{aligned} x^2 + y &\leq 45 \\ x &\leq 7 \\ x, y &\geq 0 \end{aligned}$$

Solution One of the constraints is quadratic so this is not a valid linear program.

(b)

$$\max\ 8x + \frac{1}{13}y + z$$

$$\begin{aligned} x &\leq 5 \\ 4x + 3z &\leq 9 \\ x, y, z &\geq 0 \end{aligned}$$

Solution y can just increase infinitely, increasing the objective function indefinitely this is unbounded.

(c)

$$\max\ 3x + 3y$$

$$\begin{aligned} 5x - 2y &\geq 0 \\ 2x &\leq 18 \\ x, y &\geq 0 \end{aligned}$$

Solution Valid linear program. x is bounded, and consequently so is y.

(d)

$$\max\ 2x + 2y$$

$$\begin{aligned} x + 2y &\leq 12 \\ x &\leq 6 \\ x + y &\geq 10 \\ x, y &\geq 0 \end{aligned}$$

Solution Infeasible. $6 \geq x \geq 10 - y$ implies that $y \geq 4$. However, for this to be true, y must equal $4, 5$, or 6 for the first constraint to hold true. In none of these situations does there exist a valid x value which satisfies constraint 3.

(e)

$$\max\ 2x - 3y$$

$$\begin{aligned} 3x + 5y &\geq 13 \\ x &\leq 4 \\ x, y &\geq 0 \end{aligned}$$

Solution It is unbounded. While the range of values for x is quite clear, there are an infinite possible number of values for y. While logic would tell us to make y as small as possible, the simple LP solver would not be able to find the next vertex.

Problem 2: Matches for tutoring

★★★ Level

A tutoring service has contracted you to work on pairing tutors with tutees. You are given a set of tutors U and a set of tutees V. Everyone has filled out some questionnaires, so you know which tutors are compatible with which tutees (i.e. able to tutor the right subject). Additionally, each tutor i has given a limit c_i on how many tutees they want to work with. Each tutee only gets one tutor. Describe an efficient algorithm for assigning tutors to tutees, such that as many tutees receive tutoring as possible. Give only the main idea.

Solution This is solved through bipartite matching. Create a source node and a sink node. Represent tutees as one set of nodes, and tutors as another set. There exist edges from the one source to all the i tutors, each edge with corresponding capacity c_i. Then, each tutor node has an edge to the set of tutees that they are compatible with, each of these edges having capacity 1. Then, each tutee has an edge to the sink node. Then, run the max flow algorithm. This will maximize the amount of tutees who "get" one flow (are assigned to a tutor) while also preventing tutors from being assigned too many students.

Problem 3: Criminal capture

★★★★ Level

The Central Intelligence Agency has tasked you with preventing a criminal from fleeing the country. Roads and cities are represented as an unweighted directed graph $G = (V, E)$. We're also given a set $C \subseteq V$ of possible current locations of the criminal, and a set $P \subseteq V$ of all airports out of the country. You want to set up roadblocks on a subset of the roads to prevent the criminal from escaping (i.e. reaching an airport).

Clearly you could just put a roadblock at all the roads to each airport, but there might be a better way: for example, if the criminal is known to be at a particular intersection, you could just block all roads coming out of it. You may assume that roadblocks can be set up instantaneously.

Give an efficient algorithm to find a way to stop the criminal using the least number of roadblocks.

(a) Describe the main idea of your algorithm (no proof or pseudocode necessary).

> **Solution** Group all the possible current locations C into a supernode and label it as your source node. Group the set P of airports out of the country into another supernode which will be the sink. All of the edges connecting the individual locations V will have capacity 1. Then, run the MIN-CUT algorithm. This will find the minimum number of edges that should be cut to disconnect the source from the sink which are the current locations from the airports out of the country.

(b) Analyze the asymptotic running time of your algorithm, in terms of $|C|, |P|, |V|, and |E|$.

> **Solution** The min cut algorithm (Ford-Fulkerson) is $O(Ef)$ time where f is the maximum possible flow throughout the graph. The worst case for f is that there is one vertex in C and in P, and all the other V-2 vertices are reachable from C and can reach P, meaning the worst case value for f is approx V. Thus, the worst case runtime is $O(EV)$.

(c) Unfortunately we were too slow in implementing your roadblocks. The criminal has made it to an airport and is flying from city to city within the country. Your only option now is to shut down entire airports. Naturally, you want to minimize the number of airports you must shut down.

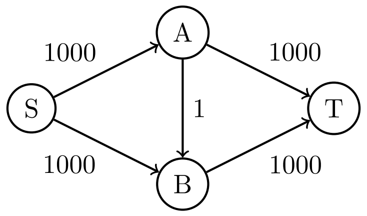
We formulate this as a graph problem where V is now the set of airports, and the (directed) edges E represent flights from one airport to another. We still know a set $C \subseteq V$ of possible current locations of the criminal, and $P \subseteq V$ the set of all airports with flights leaving the country. Give an efficient algorithm to find the minimal set of airports (vertices) to block that will prevent the criminal from reaching P.

> **Solution** Replace each vertex in the graph with a vertex-edge-vertex setup. This introduces a new 'edge' in which cutting that edge is analogous to removing the vertex. Then, to ensure that this doesn't get confused with the min cut of the 'real' edges, or have our min cut algorithm return some combination of 'real' edges and the 'artificial' edges, set all the real edges to capacity infinity. Then, it's just the same MIN-CUT process as part (a).

Problem 4: Another max flow algorithm

★★★★★ Level

Consider the following simple network with edge capacities as shown.



(a) Show that, if the Ford-Fulkerson algorithm is run on this graph, a careless choice of updates might cause it to take 2000 iterations. Imagine if the capacities were a million instead of 2000!

Solution Ford fulkerson pushes one unit of flow through the graph at a time. Thus, the top path would be updated 1000 times, and thee bottom path would be updated 1000 more times, for 2000 total updates.

(b) We will now find a strategy for choosing paths under which the algorithm is guaranteed to terminate in a reasonable number of iterations.

Consider an arbitrary directed network $(G = (V, E), s, t, \{c_e\})$ in which we want to find the maximum flow. Assume for simplicity that all edge capacities are at least 1, and define the capacity of an $s - t$ path to be the smallest capacity of its constituent edges. The fattest path from s to t is the path with the most capacity.

Show how to modify Dijkstra's algorithm to compute the fattest $s - t$ path in a graph. The full four-part algorithm response is not needed, but provide a convincing justification that your modification finds this path.

Solution Instead of labeling the nodes you discover with the length of the path that it took to get there, label the nodes you discover with the capacity of the path (smallest capacity of constituent edges) that was taken to reach that node. For example, when you have some node A with value X, and it has an edge to node B with capacity Y, label B with either X or Y depending on which is smaller. Additionally, instead of putting the nodes in a priority queue and exploring/removing the one with the minimum value, explore/remove the ones with the maximum value.

This essentially "reconstructs" all possible paths s-t by only passing on the value of the minimum edge, while treating all other edges as length 0. Then, with a simple modification in selection from the priority queue, Dijsktra's will find the longest such path (the largest capacity) rather than the smallest one.

(c) Show that the maximum flow in G is the sum of individual flows along at most $|E|$ paths from s to t.

Solution First, prove the max number of paths in a flow is $|E|$ by using the proven correctness of the min-cut/max-flow algorithm. One cannot 'cut' through more edges than actually exists in the graph, so we do prove that the maximal number of 'cuts' or max-flow paths is $|E|$. Next, we use a proof by induction to show that the max flow is the sum of individual flows.

Base case: There is one path from s to t with flow f_1. Obviously, the maximum flow is the flow of this individual path - f_1.

Induction hypothesis: The maximum flow is the sum of flows across any number of paths k.

Solution (cont.)

Induction step: Show this for $k + 1$ paths. The original total flow $f_1 + f_2 + ... + f_k$ used to be the max flow by the induction hypothesis, but it is no longer the maximum flow with the addition of the new path. The max flow can be increased by adding the flow of the newly added path, to reach $f_1 + f_2 + ... + f_k + f_{k+1}$.

(d) Now show that if we always increase flow along the fattest path in the residual graph, then the Ford-Fulkerson algorithm will terminate in at most $O(|E| \log F)$ iterations, where F is the size of the maximum flow. (Hint: It might help to recall the proof for the greedy set cover algorithm in Section 5.4.)

In fact, an even simpler rule—finding a path in the residual graph using breadth-first search—guarantees that at most $O(|V| \cdot |E|)$ iterations will be needed.

Solution Consider timestep t. We know that the remaining optimal flow at time t (call this g_t) is equal to the original optimal flow (f_{opt}) minus the optimal flow in the remaining graph (f_i). At the next timestep, at least $f_i * \frac{1}{|E|}$ of the flow passes through because the optimal flow remaining must be distributed across the $|E|$ remaining edges. Thus, the relation is $f_{i+1} \leq f_i - \frac{f_i}{|E|}$ since this relation is the same for t timesteps, $f_t \leq f_0 * (1 - \frac{1}{|E|})^t$ in which f_0 is the optimal remaining flow at timestep 0 (max flow). This is the exact same relation as the Set Cover greedy algorithm, so we just substitute the variables in the Set Cover approximate bound to get a bound of $O(|E| logF)$ for the “amount” of flow (which is also the number of iterations, since each iteration pushes 1 flow through) that this policy takes.

Problem 5: Cannibal canisters

★★★★★ Level

You see n canisters, each with their own height h_i and radius r_i (they are perfectly cylindrical). A canister can eat another canister if it has a smaller height and radius. The eaten canister will be instantly consumed, and the eating canister will be tired for the rest of the day, unable to eat anymore. Design an algorithm to make as many canisters as possible get eaten (so they don't try to eat you!) by the end of the day. Your solution should give the optimal order that canisters should eat each other, and the runtime should be $O(n^3)$.

As an example, if there are three canisters with height and radius:

$$h_1 = r_1 = 1.5$$
$$h_2 = r_2 = 2.5$$
$$h_3 = r_3 = 3.5$$

Then your output should be "c_2 eats c_1, c_3 eats c_2" or similar.

(a) For this part, assume that $\forall i, h_i = r_i$. How would you solve the problem? Describe only the main idea and runtime, no need for pseudocode or proof.

> **Solution** Sort the canisters by height in $O(nlogn)$ time. Then, iterate through the canisters, having each canister be eaten by the very next canister that is taller than it. We collapse duplicate-height canisters together and assign a number of that height of canister left over. Thus, we can keep track of the number of the canister that should be the eater/eaten next, and means we only need to iterate through all canisters once. The runtime is thus $O(n(logn + 1)) = O(nlogn)$.

(b) Solve the question fully, without the assumption that $h_i = r_i$. Hint: think about bipartite matching.

> **Solution** Create duplicates of each can such that one copy is the eaters and the other is the eaten. The eaters will iterate through the the eaten in order to construct edges with capacity 1 between eaters and the cans that they can eat. This takes n^2 time. Then, have source and sink dummy nodes, in which the source connects to the eater-set and the eaten-set connects to the sinks (all dummy connections have capacity 1). Then, run the max flow algorithm - the edges corresponding to the min-cut are the edges which represent c_1 eating c_2, etc. This max flow algorithm takes $O(|E|f)$ time, in which the largest value of $|E|$ is n^2 and the largest value of f is n (there are n capacity-1 connections to the sink node). Thus, the total runtime is the initial construction + the max flow, which comes out to $O((n^2) * (n) + n^2)$ which is dominated by $O(n^3)$.