CS 170: Efficient Algorithms and Intractable Problems

Spring 2017

●

Homework 7

Due on April 4 at 12:00pm

●

Solutions by

Ninh DO

25949105

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
YES. Can be NO if we consider this problem non-LP as the first constraint is not linear. However, we can always modify the LP solver to solve this particular non-LP.

(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
NO. Unbounded y

(c)

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

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

Solution
YES

(d)

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

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

Solution
NO. Infeasible: $x + y$ is both less than or equal 9 and greater than or equal 10.

(e)

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

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

Solution
YES

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

Represent the problem as a directed graph in which vertices are tutors and tutees and edges direct from tutors to tutees if they are compatible. We can see that there are only edges from the tutor set U to the tutee set V, but no edge within U or V. This is a bipartile matching problem.

Set up a vertex S directing to all tutors u_i with edges of capacities c_i, and set up a vertex T to which all tutees v_i direct with edges of capacities 1. The directed edges from tutors to tutees are assigned capacities 1. Our algorithm is to run the maximum-flow algorithm from S to T for the above graph. To the end, the flow from U to V goes through the edges that match tutors to tutee satisfying the problem requirements.

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
We set up a directed graph for the purpose of running maximum-flow algorithm: vertices are all cities and directed edges of capacity 1 are all roads. We consider two sets C or P for a bipartile matching problem.
Set up a vertex S directing to all criminals with edges of very large capacities, say infinity, and set up a vertex T to which all out-of-the-country airports direct with edges of very large capacities, say infinity. The directed edges corresponding to roads are assigned capacities 1 as above.
Our algorithm is to run the maximum-flow algorithm from S to T for the above setting. To the end, the min-cut between C and P is the collection of roads on which we should put blocks.

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

Solution
The general runtime of maximum-flow problem is $O(\text{number of edges} * \text{maximum flow})$. The number of edges in our setting is $|C| + |P| + |E|$, the maxinum flow is $O(|E|)$. So the runtime is $O((|C| + |P| + |E|) * |E|)$. But $|C|, |P|$ are $O(|E|)$ since we do not consider isolated criminals or airports, so the total runtime can be said $O(|E|^2)$.

(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.

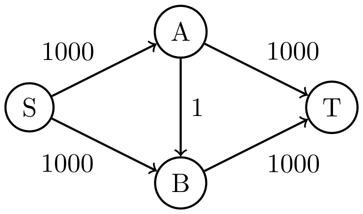
We formulate this as a graph problem where V is now the set of airports, and the (directed) edges E represent flights from one airport to another. We still know a set $C \subseteq V$ of possible current locations of the criminal, and $P \subseteq V$ the set of all airports with flights leaving the country.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

Solution
We duplicate every airport and set up a directed edge of capacity 1 between each pair of the identical airports.
The capacities of edge representing roads are 1. The capacities of edges from S and edges to T are infinity.
We run the maximum-flow algorithm from S to T for this setting, the min-cut going through the pair of the identical airports either in C or P give us the collection of airports that we should shut down.
The maximum flow is the number of vertices at most, so the runtime is $O((|C| + |P| + |E|) * |V|)$ or $O(|E| * |V|)$

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
Given the above graph, if we alternate between the paths $S - A - B - T$ and $S - B - A - T$, we will respectively decrease the capacities of SA, BT and SB, AT by 1 each time. The sum of two edges is 2000 so it needs 2000 iterations.

(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
Pseudocode:

```
for all u in V:
        flow(u) = 0
        prev(u) = none
flow(s) = \infty
H = makequeue(V)       # (H is max heap using flow-values as keys)
while H is not empty:
        u = deletemax(H)
        for all edge (u,v) in E:
                if flow(v) < min(flow(u), c(u,v)):
                        flow(v) = min(flow(u), c(u,v))
# c(u,v) is capacity of edge (u,v)
                        prev(v) = u
                        H.increasekey(v)
```

Justification:
We use induction. If v is one of the neighbors of source s, $flow(v) = min(flow(s), c(s, v)) = c(s, v)$ is exactly the fattest path from s to v since each path has only one edge. That is, the algorithm is correct up to at least one level away from s. We assume that the algorithm is correct up to k levels away from s, which is $flow(u)$ is the fattest path from s to u. We prove that the algorithm is correct up to the $k + 1$ levels away from s, that is the neighbors of u: when u is popped, the 'for' loop examines all the neighbors of u and it will update the fasttest path up to that neighbor if it finds any alternative edge

Solution (cont.)

having the capacity larger than the previous capacity. This is done by the 'if' statement.
Since the algorithm examines all vertices, so it definitely reaches t and update the fattest path from s to t along the way.

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

Solution
This comes straightforward from the min-cut property. Consider the graph G after we run the maximum-flow algorithm on it, let's look at a min-cut on this new G, the maximum flow is the sum of individual flows through edges on this min-cut, and the number of edges on this min-cut is equal to the number of paths from s to t. The number of edges on this min-cut never exceeds $|E|$, so is the number of paths. Therefore, the above property holds.

(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
Let F_t be the maximum flow NOT obtained after t iterations (i.e. the remaining of the maximum flow), we have:

$$F_{t+1} \leq F_t - \frac{F_t}{|E|} = F_t \left(1 - \frac{1}{|E|}\right)$$

$$F_t \leq F_0 \left(1 - \frac{1}{|E|}\right)^t < F_0 \left(e^{-\frac{1}{|E|}}\right)^t = F * e^{-\frac{t}{|E|}}$$

where $\frac{F_t}{|E|}$ is at least the amount of flow reduced after each iteration.
At $t = |E| \log F$, F_t is strictly less than $F * e^{-\ln F} = 1$, or in other words the algorithm will terminate in at most $O(|E| \log F)$ iterations.

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
Main idea:
Put all canisters in a min heap H with keys ordered by height or radius, then pop one by one out. The latter eats the former if it is strictly larger. If they are equal, put them in a queue to wait to be eaten. When a larger canister comes out, it will eat one in the queue in order. Say, if $a < b < c$, then a popped out first, b then popped out and eats a, c then popped out and eats b. If $a = b = c < d < e < f$, then a, b, c popped out and form a queue, larger d then popped out and eats a, x-larger e then popped out and eats b, xx-larger f then popped out and eats c. By this way, we can maximize the number of eaten canisters and resting canisters.
Runtime:
$O(n \log n)$ to put all canisters into a min heap, $O(n)$ to pop them all. In general, the runtime is $O(n \log n)$

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

Solution
Main idea:
Step 1: Duplicate the set of canisters. That is, we work with two sets of the same canisters at the same time, say sets U and V.
Step 2: Run two nested loops to construct a directed graph with edges of capacity 1 pointing from bigger canisters in U to smaller canisters in V. The canister at edge tail must be strictly larger than, and thereby being able to eat, the canister at edge head in terms of height and radius.
Step 3: Set up the source S pointing to each canisters in U with edges of capacity 1.
Step 4: Set up the sink T to which each canisters in V points with edges of capacity 1.
Step 5: Run maximum-flow alrorithm for the setting above. At the cut between two sets, we have pairs of eating and eaten canisters, say (4,1), (2,3), (3,4), etc.
Step 6: Topologically sort these pairs, we end up with the food chain where the eating order start from the end. Say $2 \to 3 \to 4 \to 1$ means: 4 eats 1, then 3 eats 4, then 2 eats 3.
Runtime:
$O(n^2)$ to made egdes, $O(n^3)$ to run maximum-flow algorithm, $O(n)$ to topologically sort the canisters out. In general, the runtime is $O(n^3)$