CS 170 — Efficient Algorithms and Intractable Problems
Spring 2017 — Prasad Raghavendra and Sanjam Garg

Homework 7

Instructions: You are welcome to form small groups (up to 4 people total) to work through the homework, but you **must** write up all solutions by yourself. List your study partners for homework on the first page, or "none" if you had no partners.

If using LaTeX (which we recommend), you may use the homework template linked on this Piazza post to get started.

Begin each problem on a new page. Clearly label where each problem and subproblem begin. The problems must be submitted in order (all of P1 must be before P2, etc). For questions asking you to give an algorithm, respond in what we will refer to as the *four-part format* for algorithms: main idea, pseudocode, proof of correctness, and running time analysis.

Read the Homework FAQ Piazza post on Piazza before doing the homework for more explanation on the four-part format and other clarifications for our homework expectations.

No late homeworks will be accepted. No exceptions. This is not out of a desire to be harsh, but rather out of fairness to all students in this large course. Out of a total of approximately 12 homework assignments, the lowest two scores will be dropped.

Special Questions:

- *Shortcut questions*: Short questions are usually easy questions that give you opportunities to practice basic materials. However, we understand that some of you are very familiar with the topics and do not want to spend too much time on easy questions. Therefore, we design shortcut questions for this purpose. A shortcut question usually has multiple parts that build upon each other and are ordered by their difficulty level. You can work on those in order or start from wherever you like. However you only need to submit the last part you are able to solve. For example, if a question has 5 parts (a, b, c, d, e), you are confident about part e, you should submit part e without any of the previous four parts. If you are confident about d but not sure about e, you should submit d for grading purposes. Please clearly indicate in your submission which part you are submitting.

- *Redemption questions*: It is important that you carefully read the posted solutions, even for problems you got right. To encourage this, you have the option of submitting a redemption file, a few paragraphs in which you explain, for each problem you choose to cover, what you did wrong and what the right idea was in your own words (not cutting and pasting from the solution!), and appending it to your homework. For example, suppose that as you review your solutions to HW1, you realize you had misunderstood question 3 and answered it incorrectly. You would write down what you just learned, and then submit it in your HW2 assignment the following week. Because these are mainly for your benefit, feel free to format them however is most useful for you.

- *Extra credit questions*: We might have some extra credit questions in the homework for people who really enjoy the materials. However, please note that you should do the extra credit problems only if you really enjoy working on these problems and want an extra challenge. It is likely not the most efficient manner in which to maximize your score.

Due Tuesday, April 4, at 11:59am

1. (★ level) Linear Programming Fundamentals

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)

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

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

Solution: No, this is not a linear program because of the quadratic term x^2.

(b)

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

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

Solution: No, this LP is unbounded. y can be arbitrarily large, so the objective function can be as well.

(c)

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

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

Solution: Yes. Note that since x is capped at 9 (by the second constraint), the first constraint puts an upper limit on y.

(d)

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

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

Solution: No, this LP is infeasible (there are no solutions that would satisfy the constraints). Adding the first and second constraint, we get $2x+2y \le 18$. This tells us $x+y \le 9$, which is incompatible with the constraint $x+y \ge 10$.

(e)

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

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

Solution: Yes, the feasible region is unbounded, but there is still an optimum because the feasible region is only unbounded in the positive y direction, for which the objective function is decreasing.

2. (★★★ level) Matches for tutoring

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: The idea here is to extend bipartite matching to capture the additional constraints. Construct the usual setup with edges between nodes in U and nodes in V, such that an edge (u,v) exists if u,v are compatible with each other. Add as well the source s and sink t. Give the U to V (tutor to tutee) edges capacities of 1 as usual. The edges from s to nodes in U will be set to each tutor's c_i to limit how his or her number of pairings. Similarly, edges from nodes in V to t will all have capacity 1 so that no tutee gets more than one tutor. Now run the max flow algorithm; each saturated edge (u,v) represents a returned pairing.

3. (★★★★ level) Criminal capture

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: Add a source node s, with an edge from s to each vertex in C. Add a sink node t, with an edge from each vertex in P to t. Make each edge from the original graph have capacity one, and each new edge have capacity ∞. Now find a minimum (s,t)-cut $(S, V-S)$, and place a roadblock on each edge that crosses the cut in the $s \to t$ direction, i.e., on each edge $(u,v) \in E$ such that $u \in S$ and $v \notin S$. We can find the minimum cut by computing the max flow (via Ford-Fulkerson) and then using the max-flow min-cut theorem.

Interpretation: we want to find a cut across paths the thief can take from their starting location in C to the exit points in P, and set up a roadblock across each edge in the cut. The min-cut guarantees we use the smallest possible number of roadblocks.

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

Solution: Ford-Fulkerson's running time is $\Theta(f \cdot |E|)$, where f is the value of the maximum flow. In this case, the value of the max flow must be at most $|E|$, so we get a $\Theta(|E|^2)$ running time.

(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[1]. 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.

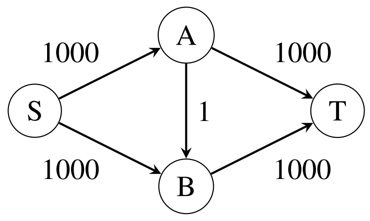
Solution: This is like before, but now we are blocking vertices instead of edges. We can split each vertex u_i into two vertices, say $u_{i,in}$ and $u_{i,out}$, with a directed edge going from $u_{i,in}$ to $u_{i,out}$. All edges that were originally going into u_i will go into $u_{i,in}$ and all outgoing edges will come out of $u_{i,out}$. As before, we add a source node that has edges to all of C and a sink that has edges from all of P.

Now we must add capacities to enforce that the only edges crossing the cut will be ones from $u_{i,in}$ to $u_{i,out}$ for various i. Thus we will set each of these edges to capacity 1, and all other capacities to ∞.

The running time analysis is the same as before, except that our flow is limited by the number of vertices, not edges, so we have $\Theta(|V||E|)$ running time.

4. (★★★★★ level) Another max flow algorithm

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:

Suppose the sequence of updates alternates path $S-A-B-T$ with path $S-B-A-T$. With each update we send a single unit of flow from s to t. Hence, we will need 2000 updates to achieve the optimal flow.

(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.

[1] Assume flights are frequent enough that we can ignore departure/arrival times.

Solution:

We make the following changes to Dijkstra's algorithm:

- `dist`(v) represents the capacity of the fattest path from s to v.
- The update function becomes

```
procedure update$((u,v) \in E)$
  dist$(v) = \max\{$dist$(v), \min\{$dist$(u), c(u,v)\}\}$
```

- We now use a max-heap instead of a min-heap.
- The initialization is `dist`$(s) = \infty$, and `dist`$(v) = 0$ for all $v \neq s$.

We can argue the correctness of this algorithm by showing that when a vertex v is removed from the priority queue its distance value is the correct capacity of the fattest path to the vertex. We do this by induction on the vertices in order of their removal from the priority queue. Suppose this is not the case and the capacity of the fattest path to v is not that given by the last update to `dist`(v) performed by edge (u,v). Then, the fattest path to v must use another edge $e = (w,z)$ out of the set of current nodes. But the fattest path to z through (w,z) must be thinner than the current value `dist`(z), which is less than `dist`(v). Hence, any path through (w,z) must yield a thinner path for v. Hence, our algorithm is correct.

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

Solution:

Given a maximum flow, consider the following algorithm which decomposes it into the sum of flow along $|E|$ paths: let e be the edge carrying the least non-zero amount of flow. Remove all flow through e by picking any path from s to t through e. This is always possible, as all other edges in the network are carrying an equal or greater flow. Repeat this procedure on the new flow obtained until all flow has been eliminated. At every iteration we make the flow through one edge to be 0. Hence, this decomposition can yield at most $|E|$ paths.

(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: At every iteration the fattest path strategy picks the path which can carry the most flow from s to t. Because the optimum flow can be decomposed in $|E|$ paths, by c), at least one of these paths must carry at least $\frac{F}{|E|}$ units of flow. Hence, the greedily picked path must achieve at least $\frac{F}{|E|}$ units of flow. But this is true for any flow through the network, as the argument c) always applies. Hence, if we let F_t denote the flow remaining after the t^{th} iteration, then $F_{t+1} < F_t(1 - \frac{1}{|E|})$. Hence:

$$F_t < F\left(1 - \frac{1}{|E|}\right)^t$$

showing that $|E| \log F$ iterations suffice, as required.

5. (★★★★★ level) Cannibal canisters

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 $\forall i, h_i = r_i$, and all heights are distinct. How would you solve the problem? Describe only the main idea and runtime, no need for pseudocode or proof.

Solution: We can sort all the canisters via increasing heights in $O(n \log n)$ time. From the assumption, if c_i has a smaller height than c_j, then c_j can eat c_i. Besides the shortest canister, in increasing height, have each canister eat the canister on its left, in that order. This will result in $n-1$ eatings, and we can do this in overall $O(n \log n)$ time.

(b) Solve the question fully, without the assumption that $h_i = r_i$. Again, you only need to provide main idea and running time.

Hint: Think about bipartite matching. This won't necessarily be similar to part (a).

Solution:

First, observe that the runtime bound is quadratic. So this means we could consider every single pair (c_i, c_j) as a preprocessing step, and our overall runtime should still be ok.

Construct G, which we will feed into a max-flow solver, as follows:

- For each canister c_i, create two vertices, v_{i1} and v_{i2}. This takes $O(n)$ time.
- For every pair of canisters c_i, c_j, if c_i can eat c_j, then create a directed edge of weight 1 (v_{i1}, v_{j2}). This takes $O(n^2)$ time to iterate through all possible i, j.
- Create a dummy source node s that has a directed edge of weight 1 to all v_{i1}. There are n such vertices, so this takes $O(n)$ time.
- Create a dummy sink node t that has a directed edge of weight 1 from all v_{i2} to t. There are n vertices, so this takes $O(n)$ time.

We should observe that in G, all potential paths are of this form:

$$s \rightarrow v_{i1} \rightarrow v_{j2} \rightarrow t$$

So we know that any path has a total path length of 3. Therefore, after feeding G into our black-box max-flow solver, we obtain the max flow, f. The maximum number of eatings is exactly $\frac{f}{3}$. If we use Ford-Fulkerson, we can compute the max flow in $O(|E| \cdot f)$ time. We know that the f cannot possibly be greater than $3(n-1)$, because at best, every canister except 1 will eat a canister. So since $O(|E|) = O(n^2)$, this is overall $O(n^3)$. All steps involved in the creation of the graph is quadratic at worst, and thus our bottleneck is running max-flow.

Now, we need to postprocess our output so that we can know in what order we should do the eating:

From the max flow graph, consider all vertices and edges that contributed to the max flow. Create a new graph G', the condensed version, as follows:

- For each v_{i1} in the original graph G touching a nonzero edge, create vertex v_i.
- For each v_{i2} in the original graph G touching a nonzero edge, create vertex v_i, if it has not already been created.
- For each nonzero edge (v_{i1}, v_{j2}), create a directed edge (v_i, v_j).

Then this graph should be a DAG, because we know no circular eating can happen. An optimal eating order can be obtained by iterating in reverse topological order, and having the current node execute its eating (if it eats). Then any canister c_i that will eat something will do so before c_i itself gets eaten, which is what we want.