CIS 320: Algorithms November 3, 2021

Lecture 17

Lecturer: Aaron Roth *Scribe: Aaron Roth*

Linear Programming

One of the most powerful optimization paradigms that have worst-case efficient solvers are so-called "Linear Programs".

A linear program is defined by a set of d variables to optimize over, a linear function of those variables to optimize, and constraints on how we can set the variables, specified as linear inequalities.

Definition 1 *A linear program is an optimization problem defined over n non-negative decision variables $x_1, \ldots, x_n \geq 0$, a linear objective function, and d linear constraints. It takes the form:*

$$\textit{Maximize} \sum_{i=1}^{n} c_i x_i$$

such that for each constraint $j \in [d]$:

$$\sum_{i=1}^{n} a_{i,j} x_i \leq b_j$$

We start by observing that linear programs are somewhat more general than they appear at first glance. For example, because the constants $a_{i.j}, b_j$ can be negative, they can also encode inequality constraints in the other direction: If we want to express the inequality $\sum_{i=1}^{n} a_{i,j} x_i \geq b_j$, we can write it as $\sum_{i=1}^{n} -a_{i,j} x_i \leq -b_j$ instead. Similarly, if we want to minimize some objective rather than maximize it, we can express that by multiplying the coefficients c_i by -1. Finally, we can express equality constraints as pairs of inequality constraints: $\sum_{i=1}^{n} a_{i,j} = b_j$ can be represented by two constraints, $\sum_{i=1}^{n} a_{i,j} \leq b_j$ and $\sum_{i=1}^{n} a_{i,j} \geq b_j$

Lots of things can be represented as linear programs. Classically, linear programs were used to express production problems. For example:

> A lumber company can produce either pallets or high quality lumber. It cannot produce more than 200 units (thousand board feet) of lumber per day, which maxes out usage of their kiln, and it cannot produce more than 600 pallets per day. Its main saw can process at most 400 logs per day. 1 unit of lumber requires 1.4 logs, and one pallet requires 0.25 logs. High quality logs used for lumber cost \$200 per log, and low quality logs used for pallets cost \$4 per log. Processing lumber costs \$200 per unit, and processing pallets costs \$5. A unit of lumber sells for \$490 per unit, and a pallet sells for \$9. How many pallets and units of lumber should the lumber company produce?

We can directly represent this as a linear program Say that x_L represents the units of lumber to produce, x_P represents the number of pallets, y_H represents the number of high quality logs purchased, and y_L represents the number of low quality logs. Then the problem is to solve:

$$\text{Maximize } 290 \cdot x_L + 4 \cdot x_P - 200 y_H - 4 \cdot y_L$$

such that:

$$x_L \leq 200 \quad x_P \leq 600 \quad 1.4 \cdot x_L \leq y_H \quad 0.25 x_P \leq y_L \quad y_L + y_H \leq 400$$

But observe that we can also write the max-flow problem as a linear program. Suppose we have a flow network $C = (V, E)$ with costs c_e (recall without loss of generality we assume there are no incoming edges to e and no outgoing edges from t). The max flow problem can be expressed as:

$$\text{Maximize} \sum_{e \text{ out of } s} f(e)$$

such that:

$$\text{For every } e \in E : f(e) \leq c_e \quad \text{and} \quad \text{for every } v \notin \{s,t\} : \sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$$

We could also write down linear programs for the bipartite matching, min-cut, minimum spanning tree, and other problems we've studied in this course (although sometimes some cleverness would be needed to argue that they have integer optimal solutions).

It turns out that we can solve linear programs efficiently and in time polynomial in the number of variables and constraints! The story is a little complicated — some of the most efficient algorithms in practice (Simplex) are not polynomial time in the worst case, and some of the polynomial time algorithms (Ellipsoid) are not efficient in practice. In this lecture, we'll show how to use the polynomial weights algorithm we derived last lecture to give approximate solutions to linear programs. A benefit of this approach is that we never need to enumerate all of the constraints, we only need to find violated constraints when they exist. So this lets us efficiently approximate the solutions to linear programs with exponentially many constraints, so long as we can efficiently identify violated constraints given a candidate solution.

To do so, we'll first convert a linear program into a linear feasibility problem, which is just a linear program without the objective

Definition 2 *A linear feasibility problem is defined over n non-negative decision variables $x_1, \ldots, x_n \geq 0$ and d linear constraints. It is the problem of finding values for the x_i such that for each constraint $j \in [d]$:*

$$\sum_{i=1}^{n} a_{i,j} x_i \leq b_j$$

We first observe that if a linear program has a solution x with optimal objective value OPT, then we can write it as a linear feasibility problem simply by adding the constraint that the objective take its optimal value:

$$\sum_{i=1}^{n} -c_i x_i \leq -\text{OPT}$$

Of course we don't know OPT, but if we had the ability to solve linear feasibility problems, then we could find it via binary search. So from here on out, we'll focus on solving linear feasibility problems.

First let us recall the final guarantee we derived last lecture for using the polynomial weights algorithm for online linear optimization:

Theorem 3 *For any sequence of losses $\ell^t \in [-R_2/2, R_2/2]^N$, the polynomial weights algorithm can be used to play vectors $w^t \in B_N(R_1)$ and obtain:*

$$\frac{1}{T}\sum_{t=1}^{T} \langle w^t, \ell^t \rangle \leq \min_{w^* \in B_N(R_1)} \frac{1}{T}\sum_{t=1}^{T} \langle w^*, \ell^t \rangle + 2R_1 R_2 \sqrt{\frac{\ln(N)}{T}}$$

Our goal is to leverage this theorem to solve linear programs. Our plan will be to run the polynomial weights algorithm, which maintains a vector w^t that we will treat as a candidate solution x to our linear feasibility problem. At every round, we will check whether it (approximately) satisfies all of the constraints. If it does, we're done, and we'll return the solution $x = w^t$. Otherwise, we'll run the polynomial weights algorithm for another round, by feeding it a loss vector ℓ^{t+1} defined by one of the constraints that is violated. The algorithm is as follows:

Algorithm 1 Solve($\{a,b\}_{j=1}^d, R_1, R_2$)

Initialize the polynomial weights algorithm, parameterized to produce vectors $w \in B_d(R_1)$ and receive losses in $[-R_2/2, R_2/2]$.
Let $t = 1$, and $w^1 \in \mathbb{R}^n$ be the vector representing the state of the PW algorithm.
while There exists a constraint j^t such that $\sum_{i=1}^n a_{i,j^t} w_i^t \geq b_j + \alpha$ **do**
  Run the PW algorithm for another iterate using loss function $\ell^t \in \mathbb{R}^n$ defined so that $\ell_i^t = a_{i,j^t}$.
  Let $t \leftarrow t+1$ and w^t be the updated state of the PW algorithm.
end while
Output $x = w^t$.

Note that we need to pass to this algorithm an upper bound R_1 on the scale of a feasible solution, and an upper bound R_2 on the quantities $a_{i,j}$, but we can do this — we'll quickly work out how to do it for the max flow problem at the end. It will not exactly solve feasibility problems, but rather will return α-approximate solutions:

Definition 4 *Given a linear feasibility problem $\{a,b\}_{j=1}^d$, x is an α-feasible solution if for all constraints j, we have:*

$$\sum_{i=1}^n x_i a_{i,j} \leq b_j + \alpha$$

The analysis turns out to be very direct and simple (which is to say, we already did most of the work when we analyzed the polynomial weights algorithm):

Theorem 5 *Let $\{a,b\}_{j=1}^d$ be a linear feasibility problem that has a feasible solution $x^* \in B_n(R_1)$, and such that $\max |a_{i,j}| \leq R_2/2$. Then Solve($\{a,b\}_{j=1}^d, R_1, R_2$) returns an α-feasible solution after at most*

$$T \leq \frac{4R_1^2 R_2^2 \ln(n)}{\alpha^2}$$

many iterations.

Proof First, observe that by construction, if the algorithm returns a solution $x = w^t$ for some t, it is an α-feasible solution, so it only remains to argue that the algorithm halts and returns something after at most T iterations. By assumption, there exists $x^* \in B_n(R_1)$ such that for every constraint j, $\sum_{i=1}^n x_i^* a_{i,j} \leq b_j$. We will consider the polynomial weights algorithm regret to x_i^* and derive a contradiction if the algorithm has not returned a solution after T iterations.

By construction, the loss function ℓ^t is defined so that at every round, $\ell^t = a_{j^t}$ for some constraint j^t. On the one hand, we know that $\sum_{i=1}^n x_i^* \ell_i^t \leq b_{j^t}$ by the feasibility of x^*. On the other hand, we know that by construction, $\sum_{i=1}^n w_i^t \ell_i^t \geq b_{j^t} + \alpha$ by definition of the algorithm. Hence the regret of the polynomial weights algorithm is at least:

$$\frac{1}{T}\sum_{t=1}^T \langle w^t, \ell^t \rangle - \frac{1}{T}\sum_{t=1}^T \langle x^*, \ell^t \rangle \geq \frac{1}{T}\sum_{t=1}^T (b_{j^t} + \alpha - b_{j^t}) \geq \alpha$$

On the other hand, the regret bound of the polynomial weights algorithm implies:

$$\alpha \leq \frac{1}{T}\sum_{t=1}^T \langle w^t, \ell^t \rangle - \frac{1}{T}\sum_{t=1}^T \langle x^*, \ell^t \rangle \leq 2R_1 R_2 \sqrt{\frac{\ln(n)}{T}}$$

Hence it must be that:

$$2R_1 R_2 \sqrt{\frac{\ln(n)}{T}} \geq \alpha$$

Solving for T gives the theorem. ■

Lets briefly return to the max-flow problem. How can we bound R_1 the norm of a feasible solution? We can upper bound this by observing that a feasible flow in the worst case saturates the capacity of every single edge, and so we have $R_1 \leq C \equiv \sum_{e \in E} c_e$. Note that this is the same quantity C that we used to bound the running time of the specialized algorithm we derived for max-flow. By inspection, we have that $\max_{i,j} |a_{i.j}| = 1$, and so we can take $R_2 = 2$.