15-451/651: Design & Analysis of Algorithms March 7, 2017
Lecture #14: Linear Programming I last changed: March 14, 2017

In this lecture we describe a very general problem called *linear programming* that can be used to express a wide variety of different kinds of problems. We can use algorithms for linear programming to solve the max-flow problem, solve the min-cost max-flow problem, find minimax-optimal strategies in games, and many other things. We will primarily discuss the setting and how to code up various problems as linear programs (LPs). At the end, we will briefly describe some of the algorithms for solving LPs. Specific topics include:

- The definition of linear programming and simple examples.
- Using linear programming to solve max flow and min-cost max flow.
- Using linear programming to solve for minimax-optimal strategies in games.
- Linear programs in standard form.

1 Introduction

In recent lectures we have looked at the following problems:

— Bipartite maximum matching: given a bipartite graph, find the largest set of edges with no endpoints in common.

— Network flow (more general than bipartite matching).

— Min-Cost Max-flow (even more general than plain max flow).

Today, we'll look at something even more general that we can solve algorithmically: **linear programming**. (Except we won't necessarily be able to get integer solutions, even when the specification of the problem is integral).

Linear Programming is important because it is so expressive: many, *many* problems can be coded up as linear programs (LPs). This especially includes problems of allocating resources and business supply-chain applications. In business schools and Operations Research departments there are entire courses devoted to linear programming. Today we will mostly say what they are and give examples of encoding problems as LPs. We will only say a tiny bit about algorithms for solving them.

Before defining the problem, let's motivate it with an example:

Example: There are 168 hours in a week. Say we want to allocate our time between classes and studying (S), fun activities and going to parties (P), and everything else (E) (eating, sleeping, taking showers, etc). Suppose that to survive we need to spend at least 56 hours on E (8 hours/day). To maintain sanity we need $P + E \geq 70$. To pass our courses, we need $S \geq 60$, but more if don't sleep enough or spend too much time partying: $2S + E - 3P \geq 150$. (E.g., if don't go to parties at all then this isn't a problem, but if we spend more time on P then need to sleep more or study more).

Q1: Can we do this? Formally, is there a *feasible* solution?

A: Yes. For instance, one feasible solution is: $S = 80, P = 20, E = 68$.

Q2: Suppose our notion of happiness is expressed by $2P + E$. What is a feasible solution such that this is maximized? The formula "$2P + E$" is called an *objective function.*

The above is an example of a *linear program.* What makes it linear is that all our constraints are linear inequalities in our variables. E.g., $2S + E - 3P \geq 150$. In addition, our objective function is also linear. We're not allowed things like requiring $SE \geq 100$, since this wouldn't be a linear inequality.

2 Definition of Linear Programming

More formally, a linear programming problem is specified as follows.

Given:

- n variables $x_1, \ldots, x_n$.
- m linear inequalities in these variables (equalities OK too).

  E.g., $3x_1 + 4x_2 \leq 6$, $0 \leq x_1 \leq 3$, etc.
- We may also have a linear objective function. E.g., $2x_1 + 3x_2 + x_3$.

Goal:

- Find values for the x_i's that satisfy the constraints and maximize the objective. (In the "feasibility problem" there is no objective function: we just want to satisfy the constraints.)

An LP with an objective falls into three categories:

- **Infeasible** (there is no point satisfying the constraints)
- **Feasible and Bounded** (there is a feasible point of maximum objective function value)
- **Feasible and Unbounded** (there is a feasible point of arbitrarily large objective function value)

An algorithm for LP should classify the input LP into one of these categories, and find the optimum feasible point when the LP is feasible and bounded.

For instance, let's write out our time allocation problem this way.

Variables: S, P, E.

Objective: maximize $2P + E$, subject to

Constraints:

$$\begin{aligned} S + P + E &= 168 \\ E &\geq 56 \\ S &\geq 60 \\ 2S + E - 3P &\geq 150 \\ P + E &\geq 70 \\ P &\geq 0 \qquad \text{(can't spend negative time partying)} \end{aligned}$$

3 Modeling problems as Linear Programs

Here is a typical Operations-Research kind of problem (stolen from Mike Trick's course notes): Suppose you have 4 production plants for making cars. Each works a little differently in terms of labor needed, materials, and pollution produced per car:

	labor	materials	pollution
plant 1	2	3	15
plant 2	3	4	10
plant 3	4	5	9
plant 4	5	6	7

Suppose we need to produce at least 400 cars at plant 3 according to a labor agreement. We have 3300 hours of labor and 4000 units of material available. We are allowed to produce 12000 units of pollution, and we want to maximize the number of cars produced. How can we model this?

To model a problem like this, it helps to ask the following three questions in order: (1) what are the variables, (2) what is our objective in terms of these variables, and (3) what are the constaints. Let's go through these questions for this problem.

1. What are the variables? x_1, x_2, x_3, x_4, where x_i denotes the number of cars at plant i.
2. What is our objective? maximize $x_1 + x_2 + x_3 + x_4$.
3. What are the constraints?

$$\begin{aligned} x_i &\geq 0 \quad \text{(for all } i\text{)} \\ x_3 &\geq 400 \\ 2x_1 + 3x_2 + 4x_3 + 5x_4 &\leq 3300 \\ 3x_1 + 4x_2 + 5x_3 + 6x_4 &\leq 4000 \\ 15x_1 + 10x_2 + 9x_3 + 7x_4 &\leq 12000 \end{aligned}$$

Note that we are not guaranteed the solution produced by linear programming will be integral. For problems where the numbers we are solving for are large (like here), it is usually not a very big deal because you can just round them down to get an almost-optimal solution. However, we will see problems later where it *is* a very big deal.

4 Modeling Network Flow

We can model the max flow problem as a linear program too.

Variables: Set up one variable f_{uv} for each edge (u, v). Let's just represent the positive flow since it will be a little easier with fewer constraints.

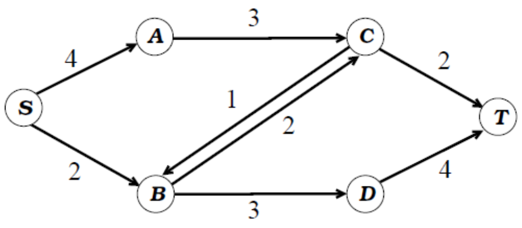
Objective: Maximize $\sum_u f_{ut} - \sum_u f_{tu}$. (maximize the flow into t minus any flow out of t)

Constraints:

– For all edges (u, v), $0 \leq f_{uv} \leq c(u, v)$. (capacity constraints)

– For all $v \notin \{s,t\}$, $\sum_u f_{uv} = \sum_u f_{vu}$. (flow conservation)

For instance, consider the example from the network-flow lecture:



In this case, our LP is: maximize $f_{ct} + f_{dt}$ subject to the constraints:

$$0 \le f_{sa} \le 4,\ 0 \le f_{ac} \le 3, \text{ etc.}$$
$$f_{sa} = f_{ac},\ f_{sb} + f_{cb} = f_{bc} + f_{bd},\ f_{ac} + f_{bc} = f_{cb} + f_{ct},\ f_{bd} = f_{dt}.$$

How about min cost max flow? In min-cost max flow, each edge (u,v) has both a capacity $c(u,v)$ and a cost $w(u,v)$. The goal is to find out of all possible maximum s-t flows the one of least total cost, where the cost of a flow f is defined as

$$\sum_{(u,v)\in E} w(u,v) f_{uv}.$$

We can do this in two different ways. One way is to first solve for the maximum flow f, ignoring costs. Then, add a *constraint* that flow must equal f, and subject to that constraint (plus the original capacity and flow conservation constraints), minimize the linear cost function $\sum_{(u,v)\in E} w(u,v) f_{uv}$. Alternatively, you can solve this all in one step by adding an edge of infinite capacity and very negative cost from t to s, and then just minimizing cost (which will automatically maximize flow).

5 2-Player Zero-Sum Games

Suppose we are given a 2-player zero-sum game with n rows and n columns, and we want to compute a minimax optimal strategy. For instance, perhaps a game like this (say payoffs are for the row player):

20	−10	5
5	10	−10
−5	0	10

Let's see how we can use linear programming to solve this game. Informally, we want the variables to be the things we want to figure out, which in this case are the probabilities to put on our different choices $p_1, \ldots, p_n$. These have to form a legal probability distribution, and we can describe this using linear inequalities: namely, $p_1 + \ldots + p_n = 1$ and $p_i \ge 0$ for all i.

Our goal is to maximize the worst case (minimum), over all columns our opponent can play, of our expected gain. This is a little confusing because we are maximizing a minimum. However, we can use a trick: we will add one new variable v (representing the minimum), put in *constraints* that our expected gain has to be at least v for every column, and then define our objective to be to maximize v. Putting this all together we have (assume our input is given as an array m where m_{ij} represents the payoff to the row player when the row player plays i and the column player plays j):

Variables: $p_1, \ldots, p_n$ and v.

Objective: Maximize v.

Constraints:

- $p_i \geq 0$ for all i, and $\sum_i p_i = 1$. (the p_i form a probability distribution)
- for all columns j, we have $\sum_i p_i m_{ij} \geq v$.

6 Matchings

Given an undirected graph $G = (V, E)$, a *matching* is a set M of edges of G such that no two edges of M share a vertex. Typically we're interested in finding the matching of largest cardinality. We've already seen how to use maximum flow to solve the matching problem in bipartite graphs. It's instructive to see how LP can be used for the matching problem.

For each edge $\{u, v\} \in E$ we create a variable $e_{\{u,v\}}$. And we assert the following inequalities:

$$0 \leq e_{\{u,v\}} \leq 1 \qquad \forall \{u, v\} \in E \tag{1}$$

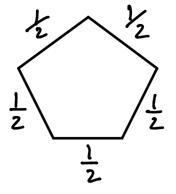
$$\sum_{v:\{u,v\}\in E} e_{\{u,v\}} \leq 1 \qquad \forall u \in V \tag{2}$$

The objective function is:

$$\max \sum_{\{u,v\}\in E} e_{\{u,v\}}$$

It's easy to see that any matching in G can be used to give a solution to this LP (set $e_{\{u,v\}} = 1$ for edges in the matching, and zero otherwise). In other wods, if Z_G is the maximum objective function value of this LP on a graph G, and M_G is the size of the maximum matching in G then $Z_G \geq M_G$. Another way to phrase this is that the integral solutions are a subset of the space being considered by the LP.

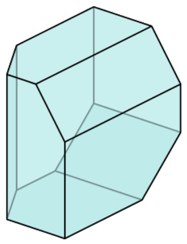
But the converse is not necessarily true, because the solution to the LP might not be integral. It turns out that when G is bipartite $Z_G = M_G$, but equality does not hold in general graphs. For example, consider the following graph of five vertices and five edges:

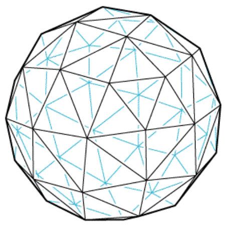


The maximum objective function value $Z_G = 2.5$ (as shown here) but the maximum matching is of size $M_G = 2$. The simplex algorithm for linear programming has a property that guarantees that for bipartite graphs it will find the maximum matching (integral), and for general graphs it will find a solution for which all the edges are integers or half-integers. The next section develops this in more detail.

7 Polytopes, Vertices, and the Simplex Algorithm

It's sometimes helpful to visualize linear programs geometrically. The set of points that satisfy a linear inequality is all the points on, or on one-side of a "plane" in $\mathbb{R}^d$. This set of points is called a *half-space.* The points satisfying all the inequalities of an LP is therefore the intersection of a finite number of half-spaces. Such a set of points is called a *convex polytope.* (Figures from Wiktionary, and Geometry Junkyard.





Given two points a and b in $\mathbb{R}^d$, we can define the *convex combination* of a and b as follows:

$$\mathrm{Conv}(a, b) = \{\alpha a + (1 - \alpha)b \mid 0 \le \alpha \le 1\}$$

$\mathrm{Conv}(a, b)$ is simply the all the points on the straight line in $\mathbb{R}^d$ between points a and b.

A set of points S in $\mathbb{R}^d$ is *convex* iff for all $a, b \in S$ we have that $\mathrm{Conv}(a, b) \subseteq S$. Note that the intersection of two convex sets is convex, because if a and b are in both of the convex sets, then the convex combination is also in both of the sets. It follows that the polytope of an LP is a convex set.

Polytopes have many other interesting properties. They can be decomposed as a collection of interrelated *facets* of dimensions varying from 0 to d. (In three dimensions these are vertices, edges, faces, and volumes.) We're not going to discuss that rich theory here except to talk about the *vertices* of a polytope, that is, the facets of dimension 0.

A point q is a *vertex* of a polytope P if the following hold:

- $q \in P$
- For any $v \in \mathbb{R}^d$ with $v \neq 0$ then at least one of $q + v$ or $q - v$ is not in P.

Put another way, if you're at a point for which there exist two opposite directions such that you can move in these directions and still stay inside the polytope, then you're *not* at a vertex.

Vertices are important because any LP has an optimum solution that is on a vertex. The intuition for this is that if you're at a non-vertex and you can move in directions v and $-v$, then moving in at least one of these directions will not cause the objective function to decrease. So you move in that direction as far as you can go. When you stop you must be entering a facet of a lower dimension. This process is then repeated until you reach a vertex.

Furthermore, the simplex algorithm always moves from vertex to vertex, and therefore the solution it finds is a vertex of the convex polytope. We'll describe the simplex algorithm in the next lecture.

7.1 Vertices of the Matching Polytope

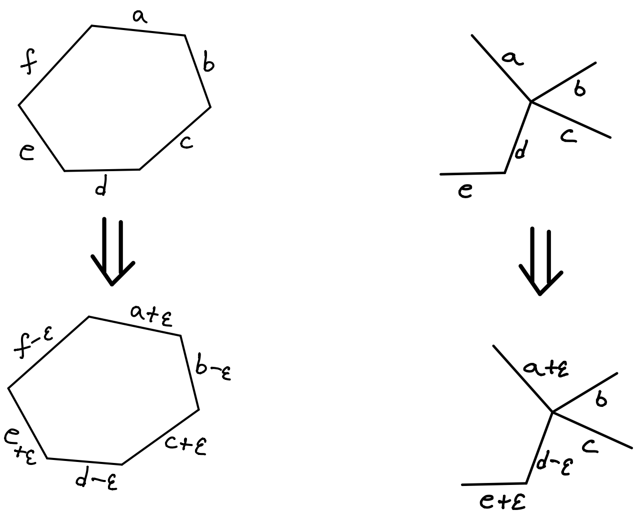
Given a graph $G = (V, E)$, denote by $\text{MP}_G \subseteq \mathbb{R}^{|E|}$ the polytope of the matching LP given by inequalities (1-2).

Theorem 1 *Let G be a bipartite graph, and MP_G be its matching polytope. If q is a vertex of MP_G, then q is an integer point in $\mathbb{R}^d$.*

Proof: We will prove the contrapositive. Suppose q is a point in MP_G that is non-integral. Then we will show that it is not a vertex of MP_G.

To avoid confusion, we'll use the term "node" for the vertices of the graph G. So we have a point q in MP_G some of whose variables are strictly between zero and one. Consider the subgraph of G (called G') consisting only of those edges whose values are non-integral (strictly between 0 and 1).

There are two cases. Supppose G' has a cycle. Because G' is bipartite, the cycle is of even length. Each edge of the cycle has a variable whose value is strictly between 0 and 1. Let ϵ be a number which is so small that when added to or subtracted from any one of the variables on the cycle, its value remains in the closed interval $[0, 1]$. The left side of the figure below illustrates this case.



We can now alternately add and subtract ϵ from the values around the cycle. This preserves the sum of the variables at every node of G'. Also notice that we could have swapped $+$ with $-$ in every case with the same result. This gives us a vector v such that the point $q + v$ and the point $q - v$ are both in the polytope.

The second and final case is illustrated in the figure above on the right. Suppose there is no cycle in G'. Then G' is a tree. We find a path from a leaf to another leaf. We can apply the same technique of alternately adding and subtracting ϵ along this path. One difference is that the sum total of the variables impinging on the starting and ending nodes do change. But since the leaf nodes of G' have only one variable strictly between zero and one impinging on them, we can increase it or decrease it without violating the constraint for that node.

And the result is again a direction v such that we can move in direction v and direction $-v$ and stay inside the polytope. ■

Notice how the proof crucially depends on there not being an *odd* cycle in G. However, the following theorem holds in the non-bipartite case.

Theorem 2 *Let G be a graph, and MP_G be its matching polytope. If q is a vertex of MP_G, then every component of the vector q is in the set $\{0, \frac{1}{2}, 1\}$.*

A similar but somewhat more complicated argument can be used to prove this.

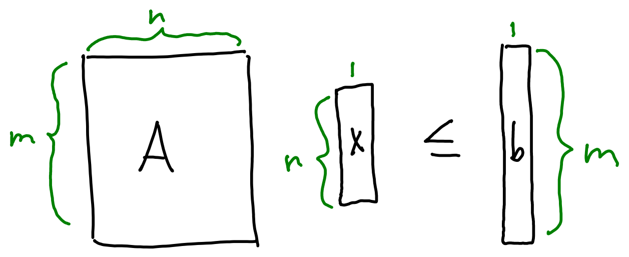
8 Standard Linear Programming Terminology and Notation

It's going to be useful as we discuss algorithms for LP in future lectures, as well as for the concept of duality, to introduce some terminology and notation for linear programs.

A linear program (LP) written in the following form is said to be in *Standard Form*:

$$\begin{aligned} \text{maximize } & c^T x \\ \text{subject to } & Ax \le b \\ & x \ge 0 \end{aligned}$$

Here there are n non-negative variables $x_1, x_1, \ldots, x_n$, and m linear constraints encapsulated in the $m \times n$ matrix A and the $m \times 1$ matrix (vector) b. The objective function to be maximized is represented by the $n \times 1$ matrix (vector) c.



Any LP can be expressed in standard form. Example 1: if we are given an LP with some linear equalities, we can split each equality into two inequalities. Example 2: if we need a variable x_i to be allowed to be positive or negative, we replace it by the difference between two variables x_i' and x_i''. Let $x_i = x_i' - x_i''$, and eliminate all occurrences of x in the LP by this substitution.

An LP is *feasible* if there exists a point x satisfying the constraints, and *infeasible* otherwise.

An LP is *unbounded* if $\forall B\ \exists x$ such that x is feasible and $c^T x > B$. Otherwise it is *bounded*.

An LP has an *optimal solution* iff it is feasible and bounded.

The job of an LP solver is to classify a given LP into these categories, and if it is feasible and bounded, it should return a point with the optimum value of the objective function. Of course, this optimal value may take on fractional values.