Lecture 18

Linear Programming

18.1 Overview

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 At the end, we will briefly describe some of the algorithms for solving linear programming problems. 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.
- Algorithms for linear programming.

18.2 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. We're only going to have time for 1 lecture. So, we will just have time to 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.

18.3 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.)

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

Variables: S, P, E.

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

Constraints:

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

18.4 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.

18.5 Modeling Network Flow

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

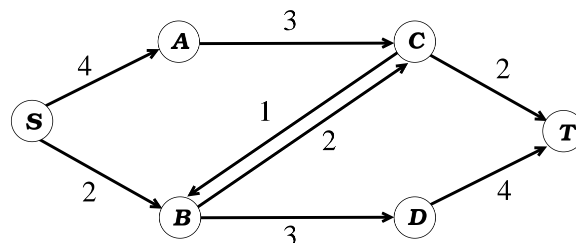
Variables: Set up one variable x_{uv} for each edge (u, v). Let's just represent the positive flow since it will be a little easier with fewer constraints.

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

Constraints:

- For all edges (u, v), $0 \leq x_{uv} \leq c(u, v)$. (capacity constraints)
- For all $v \notin \{s, t\}$, $\sum_u x_{uv} = \sum_u x_{vu}$. (flow conservation)

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



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

$0 \leq x_{sa} \leq 4$, $0 \leq x_{ac} \leq 3$, etc.

$x_{sa} = x_{ac}$, $x_{sb} + x_{cb} = x_{bc} + x_{bd}$, $x_{ac} + x_{bc} = x_{cb} + x_{ct}$, $x_{bd} = x_{dt}$.

How about min cost max flow? 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

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

where $w(u, v)$ is the cost of edge (u, v). 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).

18.6 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 $p_i \geq 0$ 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 $\sum_i p_i m_{ij} \geq v$.

18.7 Algorithms for Linear Programming

How can we solve linear programs? The standard algorithm for solving LPs is the Simplex Algorithm, developed in the 1940s. It's *not* guaranteed to run in polynomial time, and you *can* come up with bad examples for it, but in general the algorithm runs pretty fast. Only much later in 1980 was it shown that linear programming could be done in polynomial time by something called the Ellipsoid Algorithm (but it tends to be fairly slow in practice). Later on, a faster polynomial-time algorithm called Karmarkar's Algorithm was developed, which is competitive with Simplex. There are many commercial LP packages, for instance LINDO, CPLEX, Solver (in Excel) and others.

We won't have time to describe any of these algorithms in detail. Instead, we will just give some intuition and the high-level idea of how they work by viewing linear programming as a geometrical problem.

Think of an n-dimensional space with one coordinate per variable. A solution is a point in this space. An inequality, like $x_1 + x_2 \leq 6$ is saying that we need the solution to be on a specified side of a certain hyperplane. The feasible region is the convex region in space defined by these constraints. Then we want to find the feasible point that is farthest in the "objective" direction.

Let's go back to our first example with S, P, and E. To make this easier to draw, we can use our first constraint that $S + P + E = 168$ to replace S with $168 - P - E$. This means we can just draw in 2 dimensions, P and E. See Figure 18.1.

We can see from the figure that for the objective of maximizing P, the optimum happens at $E = 56, P = 26$. For the objective of maximizing $2P + E$, the optimum happens at $E = 88.5, P = 19.5$.

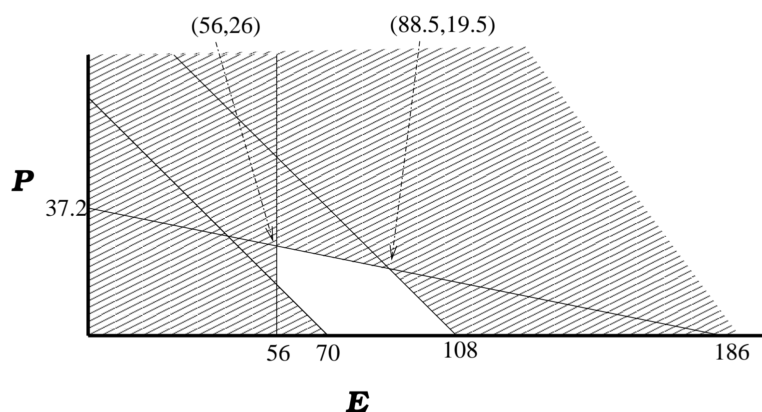


Figure 18.1: Feasible region for our time-planning problem. The constraints are: $E \geq 56$; $P+E \geq 70$; $P \geq 0$; $S \geq 60$ which means $168 - P - E \geq 60$ or $P+E \leq 108$; and finally $2S - 3P + E \geq 150$ which means $2(168 - P - E) - 3P + E \geq 150$ or $5P + E \leq 186$.

We can use this geometric view to motivate the algorithms.

The Simplex Algorithm: The earliest and most common algorithm in use is called the Simplex method. The idea is to start at some "corner" of the feasible region (to make this easier, we can add in so-called "slack variables" that will drop out when we do our optimization). Then we repeatedly do the following step: look at all neighboring corners of our current position and go to the best one (the one for which the objective function is greatest) if it is better than our current position. Stop when we get to a corner where no neighbor has a higher objective value than we currently have. The key fact here is that (a) since the objective is *linear*, the optimal solution will be at a corner (or maybe multiple corners). Furthermore, (b) there are no local maxima: if you're *not* optimal, then some neighbor of you must have a strictly larger objective value than you have. That's because the feasible region is *convex*. So, the Simplex method is guaranteed to halt at the best solution. The problem is that it is possible for there to be an exponential number of corners and it is possible for Simplex to take an exponential number of steps to converge. But, in practice this usually works well.

The Ellipsoid Algorithm: The Ellipsoid Algorithm was invented by Khachiyan in 1980 in Russia.

This algorithms solves just the "feasibility problem," but you can then do binary search with the objective function to solve the optimization problem. The idea is to start with a big ellipse (called an ellipsoid in higher dimensions) that we can be sure contains the feasible region. Then, try the center of the ellipse to see if it violates any constraints. If not, you're done. If it does, then look at some constraint violated. So we know the solution (if any) is contained in the remaining at-most-half-ellipse. Now, find a new smaller ellipse that contains that half of our initial ellipse. We then repeat with the new smaller ellipse. One can show that in each step, you can always create a new smaller ellipse whose volume is smaller, by at least a $(1 - 1/n)$ factor, than the original ellipse. So, every n steps, the volume has dropped by about a factor of $1/e$. One can then show that if you ever get *too* small a volume, as a function of the number of bits used in the coefficients of the constraints, then that means there is no solution after all.

One nice thing about the Ellipsoid Algorithm is you just need to tell if the current solution violates any constraints or not, and if so, to produce one. You don't need to explicitly write them all down. There are some problems that you can write as a linear program with an exponential number of constraints if you had to write them down explicitly, but where there is an fast algorithm to determine if a proposed solution violates any constraints and if so to produce one. For these kinds of problems, the Ellipsoid Algorithm is a good one.

Karmarkar's Algorithm: Karmarkar's Algorithms sort of has aspects of both. It works with feasible points but doesn't go from corner to corner. Instead it moves inside the interior of the feasible region. It was one of first of a whole class of so-called "interior-point" methods.

The development of better and better algorithms is a big ongoing area of research. In practice, for all of these algorithms, you get a lot of mileage by using good data structures to speed up the time needed for making each decision.