Linear Programming I

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.

Objectives of this lecture

In this lecture, we will cover

- 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:

- Maximum bipartite matching
- Maximum 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**. 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. There are also commercial software packages charging tens of thousands of dollars per license for solving linear programs (okay they also solve more general problems too, but linear programming is the basis for most of them)! 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.

2 Definition of Linear Programming

Formally, a linear programming problem is specified as follows.

Definition: Linear program

Given:

- n **real-valued** *variables* $x_1, \ldots, x_n$.
- A linear *objective function.* e.g., $2x_1 + 3x_2 + x_3$.
- m *linear inequalities* in these variables (equalities are OK too).

  e.g., $3x_1 + 4x_2 \leq 6$, or $0 \leq x_1 \leq 3$, etc.

Goal:

- Find values for the x_i's that satisfy the constraints and maximize or minimize the objective function.

Remark: No strict inequalities

Linear programs **can not** contain strict inequalities, e.g., $x_1 < 3$ or $x_2 > 5$. Why? Suppose we wrote maximize x_1 such that $x_1 < 3$, then there does not exist an optimal solution.

Remark: Variables are real-valued

This is super important to remember!! Linear program variables take on real values, i.e., the variables can not be guaranteed to take integer values!

An LP may also come without an objective function, in which case we simply wish to find any value for the x_i's that satisfy all of the constraints. Such an LP is called a "feasibility problem". You can think of this as a special case of a linear program where the objective value is just a constant (e.g., zero). We can write either minimization problems or maximization problems.

A set of x_i's that satisfies all of the constraints is called a *feasible solution.* Not all linear programs have a solution; it may be impossible to find any x_i's that satisfy the constraints. Alternatively, it may be the case that there is no optimal objective value because there exists feasible solutions of arbitrarily high value. We can classify all LPs this way into three categories.

Definition: Classification of LPs

Every LP falls into one of three categories:

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

3 Modeling problems as Linear Programs

3.1 An Operations Research Problem

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 constraints. Let's go through these questions for this problem.

Variables: x_1, x_2, x_3, x_4, where x_i denotes the number of cars at plant i.
Objective: maximize $x_1 + x_2 + x_3 + x_4$.
Constraints:

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

Note that we are not guaranteed the solution 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.

3.2 Modeling Maximum Flow

We can model the max flow problem as a linear program too. In fact, when we wrote down the definitions of the max flow problem, like the capacity and conservation constraints, and defined the net s-t flow, we pretty much did write down a linear program already!

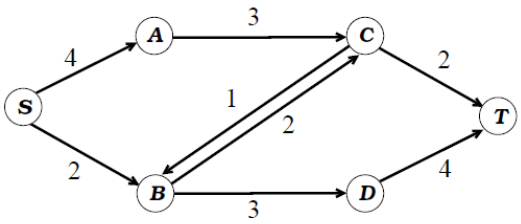
Variables: Set up a variable f_{uv} for each edge (u, v), representing $f(u, v)$

Objective: maximize $\sum_{u \in V} f_{su} - \sum_{u \in V} f_{us}$. (the net s-t flow)

Constraints:

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

For instance, consider this example:



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

Capacity	**Conservation**
$0 \leq f_{sa} \leq 4$	$f_{sa} = f_{ac}$
$0 \leq f_{ac} \leq 3$	$f_{sb} + f_{cb} = f_{bc} + f_{bd}$
$0 \leq f_{ct} \leq 2$	$f_{ac} + f_{bc} = f_{cb} + f_{ct}$
$0 \leq f_{sb} \leq 2$	$f_{bd} = f_{dt}$
$0 \leq f_{bd} \leq 3$	
$0 \leq f_{cb} \leq 1$	
$0 \leq f_{bc} \leq 2$	
$0 \leq f_{dt} \leq 4$	

3.3 Modeling Minimum-cost Max Flow

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

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

Approach #1: Maximize flow first One way to do this is to first solve for the maximum flow f, ignoring costs. Then, set up a new linear program and add the constraint that flow must equal f, i.e.

$$\sum_{u \in V} f_{su} - \sum_{u \in V} f_{us} = f$$

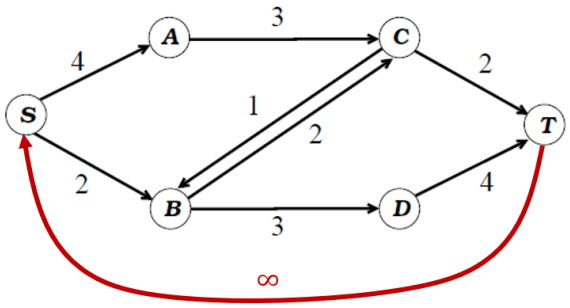
(plus the original capacity and flow conservation constraints), then set the objective to minimize the cost

$$\text{minimize} \sum_{(u,v)\in E} \$(u,v) f_{uv}$$

Note that we have used a minimization objective function rather than a maximization this time. This is allowed. If you want to stick strictly to maximization problems, then you can maximize the negative of the cost instead, as this will give you the same solution.

Approach #2: Reduce to "Minimum-cost circulation" The reason that min-cost max-flow is annoying to write as an LP is that it kind of has two objectives, to maximize flow then to minimize cost, and we can't directly encode two objectives into an LP, hence why we needed to do something like the above. An alternative is to formulate a slightly different version of the problem that does not have two objectives, and reduce min-cost max-flow to that.

To do so, lets forget about "producing" flow at s and "consuming" it at t and imagine that instead we take all of the flow into t and "circulate" it back to s, forming a big cycle of flow instead.



A *circulation* is the same thing as a flow, except that we now require that s and t also satisfy the conservation constraint. In this problem, there is no such concept as "net-flow" anymore, because the net flow should always be zero, so we can't directly ask about maximum flow. However, if we have costs on the edges, then we can write an LP for a *minimum-cost circulation*, which finds a circulation of minimum possible cost.

Variables: Set up a variable f_{uv} for each edge (u,v), representing $f(u,v)$

Objective: minimize $\sum_{(u,v)\in E} \$(u,v) f_{uv}$

Constraints:

- For all edges (u,v), $0 \le f_{uv} \le c(u,v)$. (capacity constraints)
- For **all** v, $\sum_{u\in V} f_{uv} = \sum_{u\in V} f_{vu}$. (flow conservation, **includes s and t!**)

To reduce minimum-cost max flow to a minimum-cost circulation problem, we draw the graph as above where we connect an infinite-capacity edge from t to s, and then give that edge a very negative cost. By giving (t,s) a very negative cost, the solution is encouraged to send as much

flow as possible along (t, s), which we can observe is equivalent to maximizing the s-t flow! There are other variants of the minimum-cost circulation problem but we won't cover those.

3.4 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):

	column player		
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. 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. Putting this all together we have:

Variables: $p_1, \ldots, p_n$ and v.
Objective: Maximize v.
Constraints:

- $p_i \geq 0$ for all $1 \leq i \leq n$,
- $\sum_{i=1}^{n} p_i = 1$. (the p_i form a probability distribution)
- $\sum_{i=1}^{n} p_i m_{ij} \geq v$ for all columns $1 \leq j \leq m$

4 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.

Definition: Standard Form

A linear program (LP) over n variables $x_1, \ldots, x_n$, with m constraints written in the following form is said to be in *Standard Form*:

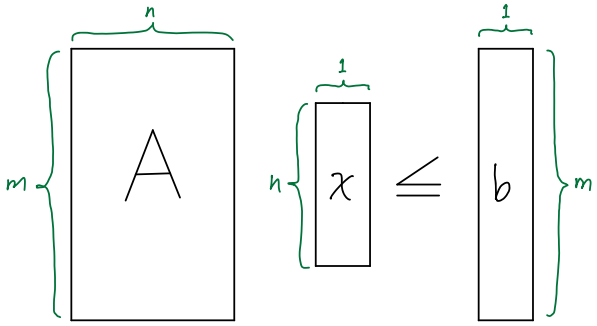
$$\begin{aligned} &\text{maximize } c_1 x_1 + \ldots + c_n x_n, \\ &\text{subject to } a_{11} x_1 + \ldots + a_{1n} x_n \leq b_1, \\ &\qquad\qquad a_{11} x_1 + \ldots + a_{1n} x_n \leq b_2, \\ &\qquad\qquad \vdots \\ &\qquad\qquad a_{m1} x_1 + \ldots + a_{mn} x_n \leq b_m, \\ &\qquad\qquad x_i \geq 0 \text{ for all } i. \end{aligned}$$

In other words, the objective is maximization, the constraints are all $f(x) \leq b$ for a constant b and the variables are required to be non-negative.

Another way to write this in a much more compact form is to package the c and b values into vectors and the a values into a matrix so we can write:

$$\begin{aligned} &\text{maximize } c^T x \\ &\text{subject to } Ax \leq b \\ &\qquad\qquad x \geq 0 \end{aligned}$$

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. Note that in standard form, the non-negativity constraints $x_i \geq 0$ **are not** counted towards the number of constraints m. The objective function to be maximized is represented by the $n \times 1$ matrix (vector) c.



Standard form turns out to be convenient and useful, especially when we will discuss duality later. For it to be useful, we should convince ourselves that we can actually always use it!

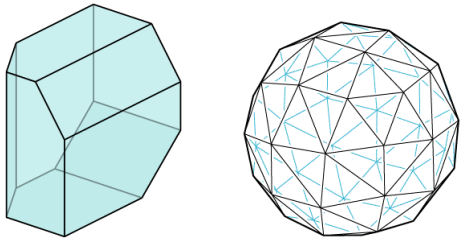
Claim: Any LP can be written in standard form

Given an LP not in standard form, we can write an equivalent LP in standard form.

- **Converting min to max** Suppose we have a minimization LP. Then we can negate the objective function and make it a maximization problem instead.
- **Handling equality constraints** Suppose we have linear equalities in our LP.
  - We can replace LHS = RHS with two inequalities: LHS ≤ RHS and LHS ≥ RHS
  - Then we negate the second inequality to get -LSH ≤ -RHS. Now we have only ≤ inequalities
- **Handling unbounded variables** If we have a variable x_i which can take on any real value, we can replace it by two variables, x_i^+ and x_i^-, and substitute x_i with $x_i^+ - x_i^-$ everywhere. The LP is equivalent, but we can constrain $x_i^+ \geq 0$ and $x_i^- \geq 0$.

4.1 Polytopes, Convexity, and Vertices

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:

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

$\text{Conv}(a,b)$ 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 $\text{Conv}(a,b) \subseteq S$. 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.

5 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.

We can use this geometric view to motivate the algorithms.

5.1 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.

5.2 The Ellipsoid Algorithm

The Ellipsoid Algorithm was invented by Khachiyan in 1980 in Russia. This algorithms solves just the "feasibility problem," i.e., whether or not there exists a feasible solution to the LP, but such an algorithm can always be used to find the optimal objective (see exercises).

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.

5.3 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.

Exercises: Linear Programming Fundamentals

Problem 1. The simplex algorithm starts at a vertex and then iteratively moves to neighboring vertices with better objective value until it arrives at the maximum. How does it actually find the starting point, though? This requires finding a feasible point of the LP, but that is just as hard of a problem!

Let's try a bootstrapping approach: We can write a second LP such that:

1. It has a trivial feasible solution that can be used as a starting point
2. The solution to this LP must be some feasible point of the original LP (it doesn't have to be the maximum, just any feasible point!)

Describe an LP that has these properties. We can solve this LP using simplex and then use the result it finds as the starting point for our actual LP!

Hint: Introduce a new variable s and add it to each constraint. How does this make finding a feasible point easier? What should the objective function for the new LP be?

Problem 2. Suppose you have an algorithm that can tell you whether an LP is feasible, but does not give you the optimal objective value (e.g., Ellipsoid). Describe a simple method for determining the optimal objective value using the feasibility algorithm as a black box.