CS 521: Linear Programming **Rutgers: Fall 2022**

Lecture 1

September 15, 2022

Instructor: Sepehr Assadi *Scribe: Sepehr Assadi*

Topics of this Lecture

1 Introduction to Linear Programming

1.1 Optimization Problems

Optimization problems are everywhere: How to design the best schedule of classes? How to go from point A to point B in the quickest possible way? How to transfer information in a network at the highest rate possible? How to fit the largest number of packages in a delivery truck? These, and numerous examples alike, are all optimization problems.

For our purpose, we can model optimization problems formally as follows.

Definition 1. Let Ω be a **domain** and $f : \Omega \to \mathbb{R}$ be an **objective function** that assign value to the elements of this domain. Moreover, let $C \subseteq \Omega$ denote a subset of the domain identified by the **constraints** of the problem. The goal is now to solve the following problem:

$$\max_{x \in \Omega} \quad f(x) \qquad \text{subject to} \qquad x \in C.$$

In words, Definition 1 simply defines optimization problems as finding an element of the domain Ω that also belongs to the "feasible" subset C and maximizes the value of $f(x)$. Notice that it is also possible for an optimization problem to involve *minimizing* the objective function instead; however, since minimizing $f(x)$ and maximizing $-f(x)$ are equivalent, there is no need for a further definition.

Attempting to solve optimization problems at this level of generality is virtually hopeless. For instance,

- Consider the following family of optimization problems $\{\Omega_n, f_n : \Omega_n \to \mathbb{R}, C_n\}_{n \in \mathbb{N}}$: for any integer $n \in \mathbb{N}$, we take Ω_n to be the set of all n-states Turing machines, C_n to be the subset of n-states

Turing machines that eventually halt on the input 0, and $f_n(x)$ for each Turing machine $x \in \Omega_n$ be the function that counts the number of 1's output by x on input 0 (we set $f(x) = 0$ if x outputs any symbol other than 1).

Now, maximizing $f_n(x)$ subject to $x \in C_n$ is equivalent to the infamous *busy beaver* problem which is a well-known undecidable problem. Thus, it is impossible to design an algorithm for solving this optimization problem in any finite time.

- Another, perhaps "scarier" example, is the following optimization problem on only four variables:

$$\min_{x=(x_1,x_2,x_3,x_4)\in\mathbb{N}^4} \quad |x_1^{x_4} + x_2^{x_4} - x_3^{x_4}| \qquad \text{subject to} \qquad x_1, x_2, x_3 \geqslant 1 \text{ and } x_4 \geqslant 3.$$

Checking whether the optimal solution to this optimization problem has value 0 or not is equivalent to *Fermat's Last Theorem*!

Given these, it is clear that we need to impose certain restriction on our optimization problems before we can hope to study them further.

1.2 Linear Optimization (a.k.a. Linear Programming)

This course is about a particularly important family of optimization problems: *Linear Programs (LPs)*[1]. In a linear program, the domain Ω is $\mathbb{R}^n$, the objective function is always a linear function $c^T \cdot x$ (or equivalently $\langle c\,,\, x\rangle$), and set C is identified by a series of linear constraints of the form $\langle a\,,\, x\rangle \geqslant b$. Formally,

Definition 2. In a linear program, we have an n-dimensional vector of **variables** $x \in \mathbb{R}^n$, an n-dimensional **objective function** $c \in \mathbb{R}^n$, and **constraints** given via a $m \times n$ matrix $A \in \mathbb{R}^{m\times n}$ and m-dimensional vector $b \in \mathbb{R}^m$. The goal is now to solve the following problem:

$$\max_{x\in\mathbb{R}^n} \quad c^T \cdot x \qquad \text{subject to} \qquad A \cdot x \geqslant b.$$

A simple example of a linear program is the following:

$$\begin{aligned} \max_{(x_1,x_2,x_3)\in\mathbb{R}^3} \quad & x_1 + 2x_2 + 3x_3 \\ \text{subject to} \quad & x_1 + x_2 + x_3 \leqslant 3 \\ & x_2 + x_3 \leqslant 2 \\ & x_1, x_2, x_3 \geqslant 0. \end{aligned}$$

The optimal solution to this LP is $(x_1, x_2, x_3) = (1, 0, 2)$ which achieves the value of 7 (later in the course, we will see a simple way *verifying* this is indeed an optimal solution; for now, this is left as an exercise for the reader).

So, why we would like to study linear programming? For one thing, it has many applications:

(a) We can *solve* numerous "day-to-day life" problems, ranging from simple scheduling to complicated scientific computing tasks, by modeling them via LPs and then solving them using many currently available LP solvers.

(b) We can use LPs as a tool toward *designing* "generic" algorithms for various problems, for instance, in combinatorial optimization.

[1] The term 'programming' here has nothing to do with computer programming and instead is used as somewhat an equivalent of scheduling.

(c) We can use LPs as an *analytical* tool for proving theorems (particularly in combinatorics) by exploiting strong properties of LPs such as duality theorems (that we will learn about later in the course).

In the next section, we give concrete examples of each of these applications.

But there is also another high important reason behind studying LPs: we can solve them *efficiently* both in theory and practice!

This course is primarily geared toward this last reason from a theoretical point of view—fundamentals of linear programming and how to solve them efficiently in *theory*—as well as parts (b) and (c) of applications above (this is *not* to say that applications of type (a) are unimportant – on the contrary, they certainly are but need a dedicated course of their own and thus are out of the scope of this course).

2 Applications of Linear Programming

Let us make the above applications more concrete via several examples.

2.1 An Application of Type (a): Creating a "Balanced" Diet

The following is a textbook example of applications of LPs in solving "day-to-day life" problems. Suppose we are interested in creating a balanced diet in terms of only its contents in vitamins A,B, C, its fiber, and its protein. We are further provided by the dietary guidelines on how much of each of these our meal should consists of, as well as a list of foods together with their vitamins/fiber/protein contents. On top of this, we are given the price for each of these foods. This is basically a table of the following form:

	Required amount	Food 1 (e.g., carrot)	Food 2 (e.g., pizza)	$\cdots$	Food n
Vitamin A	r_A	a_1	a_2		a_n
Vitamin B	r_B	b_1	b_2		b_n
Vitamin C	r_C	c_1	c_2		c_n
Fiber	r_F	f_1	f_2		f_n
Protein	r_P	p_1	p_2		p_n
Price	–	$price_1$	$price_2$		$price_n$

Table 1: Of course, this table is supposed to be filled with actual numbers but this course is not about dietary guidelines and I had no idea what numbers to put here; so, we will leave it at that and instead write them as the above parameters.

The goal is now to find a diet as a combination of these foods that satisfy all the required amounts of vitamins, fiber, and protein, and has the minimum price possible.

We can write this as an LP as follows:

- For $i \in [n]$, let x_i be a variable denoting how much of food i we will include;
- For $i \in [n]$, let a_i, b_i, c_i, f_i, p_i, denote vitamins A,B,C, fiber, and protein content of food i, respectively;
- For $i \in [n]$, let $price_i$ denote the price we have to pay for each unit of food i;
- Finally, let r_A, r_B, r_C, r_F, r_P denote the required amount of vitamins A,B,C, fiber, and protein.

Note that the last three lines are *not* variables, rather, numbers that are (supposedly) in Table 1

We can now write the following LP for solving our problem.

$$\begin{aligned}
\min_{x=(x_1,\ldots,x_n)\in\mathbb{R}^n} \quad & \sum_{i=1}^{n} price_i \cdot x_i \\
\text{subject to} \quad & \\
& \sum_{i=1}^{n} a_i \cdot x_i \geqslant r_A \quad \sum_{i=1}^{n} b_i \cdot x_i \geqslant r_B, \quad \sum_{i=1}^{n} c_i \cdot x_i \geqslant r_C, \quad \sum_{i=1}^{n} f_i \cdot x_i \geqslant r_F, \quad \sum_{i=1}^{n} p_i \cdot x_i \geqslant r_P \\
& x_i \geqslant 0 \qquad \forall i \in [n].
\end{aligned}$$

(We again emphasize that in the above LP, the only variables are $x_1, \ldots, x_n$; remaining parameters are all numbers that we can fill up using Table 1 assuming we had the actual numbers.)

We will leave verifying this LP actually solves the original problem to the reader as an easy exercise. At this point, we can simply give this LP to a standard LP solver and obtain the solution to our problem.

2.2 An Application of Type (b): The Bipartite Matching Problem

We consider yet another textbook example of LPs, this time for solving algorithmic problems. In the *bipartite matching* problem, we are given a bipartite graph $G = (L \sqcup R, E)$ with bipartition L and R of vertices. A *matching* M in G is any collection of edges that do not share any vertices (hence, they 'match' vertices in L to vertices in R so that each participating vertex in the matching is matched to precisely one other vertex). In the bipartite matching problem, the goal is to find a maximum matching, namely, a one with the largest number of edges.

Let us define an *integral* 0/1 variable $y_e \in \{0, 1\}$ for each edge $e \in E$ (we emphasize that these variables are *not* in $\mathbb{R}^n$, but rather we forced them to be only 0 or 1). Think of $y_e = 1$ as showing that we pick the edge e in our maximum matching and $y_e = 0$ as we do not.

To ensure that the edges we pick form a matching, we need to make sure we do not pick more than one edge per vertex. We can write this as the following constraint:

$$\forall u \in L: \quad \sum_{e \ni u} y_e \leqslant 1 \qquad \text{and} \qquad \forall v \in R: \quad \sum_{e \ni v} y_e \leqslant 1.$$

Finally, our task of maximizing the number of edges in the matching becomes

$$\max \sum_{e \in E} y_e.$$

Putting these together, we get the following optimization problem:

$$\begin{aligned}
\max_{y} \quad & \sum_{e \in E} y_e \\
\text{subject to} \quad & \forall u \in L: \quad \sum_{e \ni u} y_e \leqslant 1 \\
& \forall v \in R: \quad \sum_{e \ni v} y_e \leqslant 1 \\
& \forall e \in E: \quad y_e \in \{0, 1\}.
\end{aligned}$$

The above optimization problem is *not* an LP (even though it may very much look like it). The problem is with the very last constraint of $y_e \in \{0, 1\}$ that forces the program to only use integer values for the variables. Such a program is instead called an *Integer Linear Program (ILP)*. Unfortunately, unlike LPs, ILPs are much harder to solve or reason about. For one thing, solving general ILPs is an NP-hard problem (you can prove this easily; try!).

There is a general recipe in these situations: what if we *relax* the integrality constraint to obtain the following LP instead?

$$\begin{aligned} \max_{x \in \mathbb{R}^E} \quad & \sum_{e \in E} x_e \\ \text{subject to} \quad & \forall u \in L: \quad \sum_{e \ni u} x_e \leqslant 1 \\ & \forall v \in R: \quad \sum_{e \ni v} x_e \leqslant 1 \\ & \forall e \in E: \quad 0 \leqslant x_e \leqslant 1. \end{aligned}$$

Such an LP is typically called the *LP relaxation* of the original ILP (or directly the original problem). Of course, now it is no longer clear that solving this LP can tell us much about the original problem. Of course, the optimal value of this LP is always an upper bound on the optimal value of the original ILP simply because any solution to that ILP is a feasible solution here as well. But can it be that this relaxation made the problem so different that these values now differ drastically from each other?

The answer to this question *for general LP relaxations and ILPs* can be *Yes*; we will see examples of this soon in the course. However, we are now going to prove that for the specific case of the bipartite matching problem, this is not the case.

Proposition 3. *Any feasible fractional solution $x \in \mathbb{R}^E$ to the LP for bipartite matching can be* rounded *to a feasible integral solution $y \in \{0,1\}^E$ without decreasing the objective value.*

Before getting to the proof, notice a direct corollary of this proposition. We can now solve the bipartite matching LP relaxation to find an optimal solution x. We know that the objective value of x is at least as large as the optimal solution of the ILP. By Proposition 3, we can then round x to an integral solution y of the same value, thus implying that y is also optimal for the ILP. This then gives us a maximum matching of the original graph and solves the problem. Later in the course, we prove a stronger version of Proposition 3 that shows that in fact many "natural" LP solvers when given the bipartite matching LP relaxation, directly find an integral solution (eliminating the need for this rounding step).

Proof of Proposition 3. Fix the feasible solution $x \in \mathbb{R}^E$. If x is already integral, we are done by taking $y = x$. Otherwise, perform the following step first.

Step one: cycle canceling. Find a cycle in the support of x, denoted by $\text{supp}(x) := \{e \in E : x_e > 0\}$, that contains a fractional value. If no such cycle is found go to the next step, otherwise let C be the cycle and let $e \in C$ be the edge with the smallest fractional value x_e in the cycle. Let $\delta = x_e$. Now update x as follows. Let $C = e_1, e_2, e_3, \ldots, e_\ell$ denote the edges of the cycle ordered such that $e_1 = e$ is the picked edge with $x_e = \delta$. Update

$$x_{e_1} \leftarrow x_{e_1} - \delta, \quad x_{e_2} \leftarrow x_{e_2} + \delta, \quad x_{e_3} \leftarrow x_{e_3} - \delta, \ldots,$$

namely, alternately decrease and increase x-values of edges by δ.

Since we are in a bipartite graph, the length of C must be even (recall that bipartite graphs are precisely those that have no odd cycles). This ensures that (i) the value of $\sum_{e \ni w} x_e$ remains the same for every vertex $w \in L \cup R$ in the graph since we increased one edge of w and decreased another one by the same value; and (ii) value of $\sum_{e \in E} x_e$ remains the same by the same reasoning. Moreover, support of x is now at least one edge smaller because the original edge e we started with now has $x_e = 0$.

Thus, through this transformation, we only *shrink* the support of x without decreasing the objective value. As the support is finite, this process has to end eventually and at that point, we can no longer have any cycles inside it.

Step two: rounding forests. Now that there are no cycles left in the support of x, we have that $\text{supp}(x)$ is a forest (namely, a graph without cycles). Any part of this support that is integral has to be disjoint from the rest because integral parts can only form a matching and thus vertices have degree exactly one (they form a collection of vertex-disjoint edges).

Let F be the fractional part of the support. Pick any arbitrary leaf-nodes in the forest F (namely, a vertex of degree one). Let v be this leaf-node u be its parent, and $e = (u, v)$ be the edge between them. Let $x_e = 1$ and reduce the x-value of all other edges incident on u to become 0. Firstly, since v was a leaf-node and thus had no other edges, this transformation does not make x infeasible. Secondly, the total value of x incident on vertex u previously was at most 1 (to ensure feasibility of the original x) and by this change it has become exactly 1. Thus, we did not reduce the value of x at all. Finally, all edges incident on u have integral values and thus are removed from F. We can continue this process and at every step we obtain a feasible solution of the same value or higher and eventually transform the entire F into an integral solution.

Let y be this final solution which is now integral and has value as large as x. This concludes the proof. □

Remark. In general, rounding a fractional solution to an integral one does not necessarily need to be "lossless". Finding the best rounding scheme (and even the best LP relaxation of a problem) is one of the main topics studied in *approximation algorithms*, and we will revisit this throughout the course. In general, the gap between the optimal ILP value and its corresponding LP relaxation is referred to as the **integrality gap**.

2.3 An Application of Type (c): Bipartite Matching vs Vertex Cover

Illustrating the analytical applications of LPs and in particular using duality theorems require further knowledge of LPs beyond what we discussed so far. Thus our hands are rather tied for demonstrating these applications. Instead, we are going to show a type of result we can prove using these techniques without actually proving the result using LPs and instead prove it using more combinatorial means. Later in the course, we will see how this result and various other follow from a general approach in linear programming.

Let $G = (L \sqcup R, E)$ be a bipartite graph. We already defined what a matching M is in G. Let $\mu(G)$ denote the size of the maximum matching in G. A vertex cover in G is a set of vertices S that *cover* all the edges, i.e., any edge has at least one endpoint in S. Let $\tau(G)$ denote the size of the minimum vertex cover in G. What is the connection between $\mu(G)$ and $\tau(G)$?

It is easy to see that $\mu(G) \leqslant \tau(G)$ always. Consider the edges of any maximum matching M. Any vertex cover of G needs to cover these edges but since these edges are vertex-disjoint, the vertex cover needs to pick at least one vertex for each edge of M, implying that $\mu(G) \leqslant \tau(G)$. The question now is how much larger can $\tau(G)$ get? The answer in bipartite is nothing at all. This is often referred to as the *König's theorem*.

Proposition 4. *In any bipartite graph, we have $\mu(G) = \tau(G)$.*

Later in the course, we prove this result using LP duality, by showing that (i) fractional matching LP and fractional vertex cover LP are dual to each other, and (ii) applying the strong duality of LPs (and also using Proposition 3 to relate fractional matchings to integral ones) to conclude the proof[2].

We now give a direct combinatorial proof of Proposition 4. To do that, we need the following result referred to as the *extended Hall's theorem* (this can be proven using a simple argument from the original Hall's theorem, or as an immediate corollary of the *Tutte-Berge formula* for the size of maximum matchings).

Fact 5. *Let $G = (L \sqcup R, E)$ be any bipartite graph with $|L| = |R| = n$. Then,*

$$\max_{A \subseteq L \text{ or } A \subseteq R} (|A| - |N(A)|) = n - \mu(G),$$

[2]If none of this makes sense, do not worry; we will get to these later.

where $N(A)$ denotes the set of neighbors of A.

We now prove Proposition 4.

Proof of Proposition 4. We only need to show that $\mu(G) \geqslant \tau(G)$ as we established the other direction easily already. Let A^* be the set in G that *witnesses* the maximum in Fact 5, i.e.,

$$A^* := \operatorname{argmax} (|A| - |N(A)|) .$$

By symmetry, we assume A^* is in L. Let S be $N(A^*) \cup (L \setminus A^*)$. We claim that S is a vertex cover of G. The only edges that may not be covered by S are the ones going from A^* to $R \setminus N(A^*)$. But there can be no such edge in the graph because $N(A^*)$ contains all the neighbors of A^*. Thus, S is a valid vertex cover of G.

We now have that

$$|S| = |N(A^*)| + |L \setminus A^*| = |N(A^*)| + n - |A^*| = n - (|A^*| - |N(A^*)|) = n - (n - \mu(G)) = \mu(G),$$

where in the second to last equality we used the fact that A^* is a witness in Fact 5. As $\tau(G) \leqslant |S|$ (because it measures the minimum vertex cover size and S is some vertex cover), we have $\tau(G) \leqslant \mu(G)$, concluding the proof. □

This concludes our first lecture on linear programming. From the next week, we jump into the fundamental theory of LPs as well as some of their theoretical applications along the way.