A Course in Graduate Algorithms **Lecture 16**

Linear Programming

Designed by Prof. Bo Waggoner for the University of Colorado-Boulder *Updated: 2023*

Linear programming is a general technique for solving a very large variety of problems. A linear program is an optimization problem involving finding a vector $x \in \mathbb{R}^n$ that maximizes a linear function, subject to a set of linear constraints. If we can write our problem down as a linear program, then we can use a general-purpse LP algorithm to solve it. This lecture is only a brief pragmatic introduction to linear programming, and there is much more (such as duality theory) that we will not cover here.

Objectives:

- Know the definition and three key components of a linear program.
- Be able to write a given problem down as a linear program.
- Know the definition of integer linear programming and the difference.

1 Linear Programs

1.1 Example

Suppose we have a set of resources we can use to produce different kinds of goods:

- Our mining operations produce 100 units of wood and 50 units of stone per day.
- We can produce a house using 5 units of wood and 10 units of stone.
- We can produce a wagon using 4 units of wood and 1 unit of stone.
- We make a profit of 10 per house and 7 per wagon.
- Our goal is to decide how many houses and how many wagons to produce per day[1] to maximize profit per day.

We can write this down as an optimization problem with three key components: *variables*, an *objective*, and *constraints*.

- The **variables** are x_1, the amount of houses per day, and x_2, the amount of wagons per day, that we will decide to produce.
- The **objective** is to maximize profit per day. Based on the above, profit per day is $10x_1 + 7x_2$.
- The **constraints** are that we only have so much wood and stone available per day. Specifically, we must satisfy $5x_1 + 4x_2 \leq 100$, because each house uses 5 units of wood and each wagon 4. Similarly, $10x_1 + x_2 \leq 50$. Finally, note that we can't produce "negative" houses or wagons, so $x_1 \geq 0$ and $x_2 \geq 0$.

We can write this optimization problem as follows:

$$\begin{aligned} & \max_{x_1, x_2} && 10x_1 + 7x_2 \\ \text{s.t.} \quad & && 5x_1 + 4x_2 \leq 100 \\ & && 10x_1 + x_2 \leq 50 \\ & && x_1, x_2 \geq 0 \end{aligned}$$

[1] This production is on average, i.e. it's okay to produce at a non-integer rate such as 1.5 houses per day.

This is an example of a *linear program (LP)*. We will see that there are general-purpose algorithms to solve any optimization problem of this form. So we do not need to design a specific algorithm for our production problem. We just need to write it as an LP, and then we can use a solver.

1.2 Definitions

A **linear function** on $\mathbb{R}^n$ is one that can be written as $f(x) = w \cdot x$ for some vector $w \in \mathbb{R}^n$, i.e. a function of the form $f(x) = w_1 x_1 + \cdots + w_n x_n$.

A **linear program** is an optimization problem with the following three components:

- n **variables**, which we will usually represent as $x \in \mathbb{R}^n$.
- m **constraints**, where each constraint has one of three possible forms: $f(x) \leq \beta$, or $f(x) \geq \beta$, or $f(x) = \beta$. In all cases, f is a linear function and $\beta \in \mathbb{R}$.
- An **objective**, which is one of two possible forms: $\max g(x)$ or $\min g(x)$. In both cases, $g(x)$ is a linear function.

The optimization problem is to choose values for the variables, that satisfy all the constraints, in order to optimize the objective (maximize or minimize, as the case may be).

1.3 Feasible and optimal solutions

A solution x is **feasible** if it satisfies all of the constraints; we call all such x the **feasible region**. If the feasible set is empty, we call the linear program **infeasible**. In this case, the constraints ultimately contradict each other; it is not possible to satisfy all of them at once. Here is an example of an infeasible linear program:

$$\begin{aligned} \max_x \quad & x_1 + x_2 \\ \text{s.t.} \quad & x_1 \geq 10 \\ & x_2 \geq 10 \\ & x_1 + x_2 \leq 15. \end{aligned}$$

A solution x is **optimal** for a maximization objective $\max g(x)$ if (1) x is feasible, and (2) for any feasible solution x', we have $g(x) \geq g(x')$. Optimality for a minimization problem is analogous.

For any linear program, there are three mutually exclusive possibilities:

1. The program is infeasible.
2. The program is feasible and has at least one optimal solution.
3. The program is feasible but has no optimal solution.

In the last case, what happens is that we can find feasible solutions with arbitrarily high objective value (for a maximization problem, or arbitrarily low for minimization). Such a linear program is called **unbounded**. Here is an example of an unbounded linear program:

$$\begin{aligned} \max_x \quad & x_1 \\ \text{s.t.} \quad & x_1 \geq 10. \end{aligned}$$

When we have an optimal solution x^*, intuitively, we have pushed x^* as far as possible in the direction of the objective, up to the limits imposed by the constraints. If a constraint holds at x^* with exact equality, we say that constraint is **binding**. Otherwise, we sometimes say the constraint is *slack*, and the amount of slack is the difference between the left and right sides of the inequality.

Exercise 1. Given one example each of a linear program that is (a) infeasible, (b) feasible and with an optimal solution, (c) feasible and unbounded.

2 Standard form

A linear program is in **standard form** if (1) it is a maximization problem, (2) all of the variables are constrained to be nonnegative, and (3) except for nonnegativity, all constraints are less-than-or-equal constraints. We can write such a program as follows:

$$\begin{aligned} & \max_x && c_1x_1 + \cdots c_nx_n \\ \text{s.t.} \quad & A_{11}x_1 + \cdots + A_{1n}x_n && \le b_1 \\ & \ddots && \le \vdots \\ & A_{m1}x_1 + \cdots + A_{mn}x_n && \le b_m \\ & x_1 && \ge 0 \\ & \ddots && \ge \vdots \\ & x_n && \ge 0. \end{aligned}$$

Here, of course, some of the c_i and the A_{ij} parameters may be zero. As hinted by the subscripts, **we can write an LP in standard form more succinctly as**

$$\begin{aligned} \max_x \quad & c \cdot x \\ \text{s.t.} \quad & Ax \le b \\ & x \ge \vec{0}. \end{aligned}$$

Here $x \in \mathbb{R}^n$, $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ and $A \in \mathbb{R}^{m \times n}$.

Proposition 1. *Any linear program can be converted into standard form such that the feasible set and optimal solutions (if any) are the same as the original.*

Proof. First, if the objective is a minimzation $\min c \cdot x$, we can convert it into a maximization $\max -c \cdot x$. A solution x minimizes $c \cdot x$ if and only if it maximizes $-c \cdot x$. So, now we have a maximization problem.

Second, we can take any variable x_i that is not constrained to be nonnegative, and create two nonnegative variables, y and z, satisfying $x_i = y - z$. In the objective and all constraints, we replace the term c_ix_i with $c_iy + (-c_i)z$. Observe that for any value of the original variable x_i, there are values $y, z \ge 0$ such that $x_i = y - z$, so that the objective value is the same and all new constraints are satisfied if and only if the old constraints were.

Finally, we can convert all constraints to $\le$ constraints. We first replace equality constraints $a \cdot x = \beta$ with a pair of inequality constraints $a \cdot x \le \beta$ and $a \cdot x \ge \beta$. Then, for any $\ge$ constraint $a \cdot x \ge \beta$, we can simply replace it with $(-a) \cdot x \le -\beta$. □

Exercise 2. Convert this LP into standard form:

$$\begin{aligned} \min_x \quad & 3x_1 - 2x_2 \\ \text{s.t.} \quad & x_1 + x_2 = 10 \\ & x_1 \ge 0. \end{aligned}$$

Exercise 3. Sometimes we are given inequalities with variables on both sides, such as $x_1 \ge x_2$. How can such constraints also be converted to standard form?

3 Geometry and Algorithms

Our main focus for linear programming will be on how to solve our problems by writing them as LPs. Then, we can apply standard algorithms or packages for solving the LPs. We won't go into much detail on how those algorithms work, but we will overview the basics now.

3.1 Geometry

Constraints. Consider a constraint such as $a \cdot x \leq \beta$. The set of variables satisfying this constraint is called a *halfspace*, because the hyperplace $\{x \ : \ a \cdot x = \beta\}$ cuts $\mathbb{R}^n$ into two pieces, and the variables satisfying the constraint consist of everything on one side of the hyerplane.

An LP's feasible region consists of the x that satisfy a set of constraints. The feasible region is therefore an intersection of a finite number of halfspaces. Such a set is called a *polyhedron*. For example, in this linear program:

$$\begin{aligned} \max_x \quad & x_1 + \cdots + x_n & \\ \text{s.t.} \quad & x_i \geq 0 & \text{(for all } i\text{)} \\ & x_i \leq 1 & \text{(for all } i\text{),} \end{aligned}$$

the feasible region is the hypercube in $\mathbb{R}^n$.

Objectives. Focusing on maximization objectives, a linear objective is represented by a vector $c \in \mathbb{R}^n$ such that the objective value is $c \cdot x$. The set of points with a given objective value, for example 2, consist of a hyperplane $c \cdot x = 2$. Solving the linear program consists of finding the "farthest" hyperplane that still intersects the feasible set.

Figure 1 illustrates these concepts with the following linear program:

$$\begin{aligned} \max_x \quad & 2x_1 + x_2 \\ \text{s.t.} \quad & x_1 + 2x_2 \leq 5 \\ & x_1 - 2x_2 \leq 4 \\ & 5x_1 + x_2 \geq 1. \end{aligned} \tag{1}$$

Exercise 4. At an optimal solution to LP 1, see Figure 1, which constraints are binding and which are slack?

3.2 Simplex-like algorithms

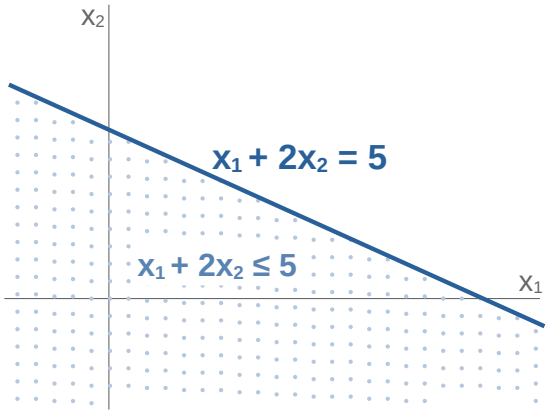
In the example of Figure 1, we saw that the optimal solution occurred at a vertex of the feasible polytope. In general, it is possible that one entire "side" of the feasible set is optimal, if the objective is perpendicular to the constraint. However, the following useful fact holds (we state it without proof here):

Fact 1. *If a linear program in standard form is feasible and not unbounded, i.e. has an optimal solution, then in particular there is at least one optimal solution that is a vertex of the feasible polyhedron.*

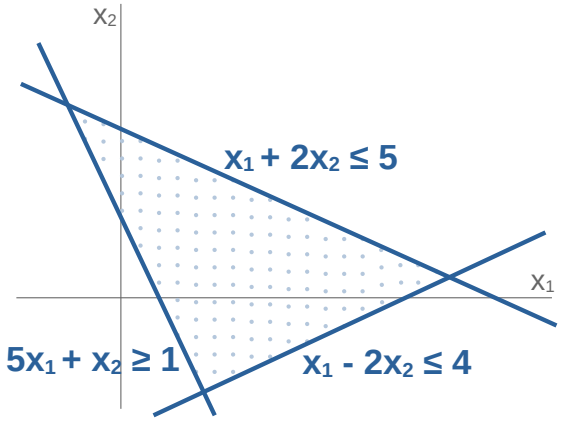
This suggests that we can solve LPs by only looking at the vertices. One way to think of a vertex is as a place where a certain subset of constraints all bind, i.e. all hold with equality. So one algorithm is a brute-force approach that considers each possible subset of constraints, checks if their intersection exists and is feasible, and returns the resulting vertex with highest objective value. The following fact (also stated without proof) enables a much more efficient approach.

Fact 2. *Given a linear program in standard form that is feasible and not unbounded, and given a vertex x of the feasible region that is not optimal, there is a "neighboring" vertex that has strictly higher objective value. By neighboring, we mean exchanging one constraint out of the binding set and one constraint into it.*

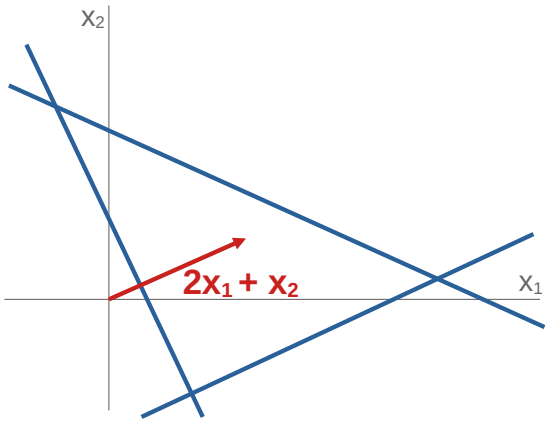
Figure 1: An illustration of LP 1.



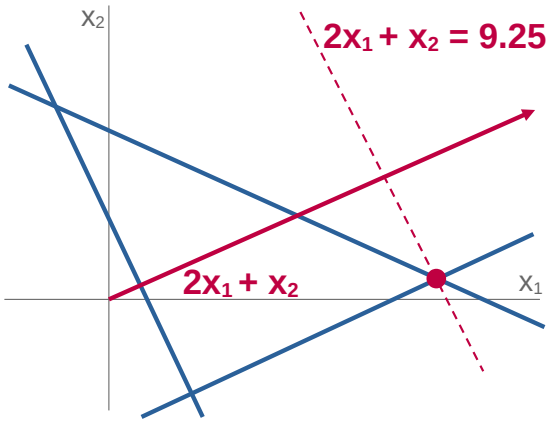
(a) The line $2_x1 + x_2 = 5$ and the halfspace of points x satisfying $2x_1 + x_2 \leq 5$.



(b) A polyhedron, i.e. intersection of three halfspaces.



(c) The objective function $\max 2x_1 + x_2$, represented as the vector $(2, 1)$.



(d) The optimal solution to LP 1, $x_1 = 4.5$, $x_2 = 0.25$, with an objective value of 9.25.

Using Fact 2, we obtain the **Simplex Algorithm**:

1. Check that the problem is feasible.
2. Find an initial feasible vertex.
3. Check if the current vertex is optimal.
4. If not, move to a neighboring vertex with strictly higher objective function.
5. Repeat steps 3-4 until optimality is reached or we find the problem is unbounded.

Steps 1 and 2 are not trivial and involve solving an easier linear program as a subroutine. We may find the problem is unbounded if we search for a "neighboring" vertex to step to, and find essentially that we can step "infinitely far".

Complexity. Many variants of the simplex algorithm have been studied in depth since its introduction in about 1950. First, note that the feasible polyhedron may have exponentially many vertices in n and m. For example, the hypercube in $\mathbb{R}^n$ is described by $2n$ constraints but has 2^n vertices. So an algorithm that visits vertices sequentially might run in exponential time.

And in general, the simplex algorithm could indeed visit exponentially many vertices. Its worst-case runtime is not polynomial. However, in practice, it is often found to be very efficient. A linear program generally needs to be carefully constructed in a very adversarial way for simplex to take exponential time. Whether simplex is the best choice can depend on the structure of the problem.

3.3 Interior-point algorithms

The simplex method worked by exploring vertices on the boundary of the feasible region. Interior-point methods do the opposite: they move candidate solutions "through" the interior of the space. We will describe one of the simplest and oldest methods: the **Ellipsoid Algorithm**. An ellipsoid is a sphere whose axes have been stretched; in other words, a set of the form $\{x : \sum_i \alpha_i x_i^2 \leq 1\}$ for some coefficients $\{\alpha_i\}$.

1. Find an initial large ellipsoid that contains the feasible region.
2. Check if the center $\hat{x}$ of the ellipsoid is feasible.
3. If it is feasible, add a new constraint $c \cdot x \geq c \cdot \hat{x}$. Cut the ellipsoid in half with this constraint and define a new ellipsoid that covers the new, smaller feasible region.
4. If it is not feasible, find a constraint that $\hat{x}$ violates. Cut the ellipsoid in half with this constraint and define a new ellipsoid that covers the new, smaller feasible region.
5. Repeat steps 2-4 until the ellipsoid is so small that it essentially contains just one point; return that.

As with the simplex method, the initial step of bounding feasible solutions is not trivial. It is also clear that the final step, termination, involves a number of details. The broad point, however, is that at each stage, the ellipsoid shrinks in size significantly, which implies that the method halts after polynomially many steps.

Complexity. The ellipsoid method runs in polynomial time. This proves that linear programming is solvable in polynomial time, something not known to be true until the ellipsoid method in 1979. It was originally interesting only theoretically, as the simplex algorithm was found to generally be faster in practice. Other interior-point methods besides the ellipsoid method have become competitive in practice. Whether they are the best depends on the problem structure.

Separation oracles. It has been observed that the ellipsoid method can run efficiently even in some cases where there are exponentially many constraints. (For example, there may be a large family of constraints that we could theoretically enumerate, but would take a long time to write down.) All that the ellipsoid needs is an "oracle" that, given $\hat{x}$, decides whether it is feasible and, if not, returns one constraint that it violates. Thanks to this observation, one can sometimes solve problems that are apparently too large in polynomial time by showing that they have such a "separation oracle".

4 Applications and Examples

Profit-maximizing production. We can first generalize the example from the introduction. We are producing n kinds of products per day using m kinds of resources. A unit of product i requires A_{ij} amount of resource j. We obtain b_j amount of resource j per day. We can sell a unit of product i for a profit of c_i. Our goal is to decide how much of each product to produce, x, to maximize profit:

Profit-Maximizing Production

$$\begin{aligned} \max_x \quad & c \cdot x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq \vec{0}. \end{aligned}$$

This is an example of a *packing* problem: an LP in standard form where all of the parameters A, b, c are nonnegative.

Min-cost procurement. Via a relationship called *duality*, which we will not cover, this problem is closely tied to the previous one. There are again n kinds of products and m kinds of resources. But now, we are buying the resources, and the price we pay for a unit of resource j will be y_j. Now our variables are y instead of x. Recall that there are b_j units of resource j per day, so our total payment will be $b \cdot y$, and we want to minimize this payment. But we have to pay enough. Specifically, we know that the firm can make a profit of c_i by combining A_{1i} units of resource 1, ..., A_{mi} units of resource m. So we must be sure that the total payment we make for that set of resources is at least c_i. Some thought reveals that this yields a constraint of $A^\intercal y \geq c$, where $A^\intercal$ is the transpose of the matrix A from the Profit-Maximizing Production problem.

Min-Cost Procurement

$$\begin{aligned} \min_y \quad & b \cdot y \\ \text{s.t.} \quad & A^\intercal y \geq c \\ & y \geq \vec{0}. \end{aligned}$$

This is an example of a *covering* problem: a minimization LP with only nonnegativity and $\geq$ constraints where all of the parameters $A^\intercal, b, c$ are nonnegative.

Packing and covering problems turn out to be related to each other via *LP duality*, a deep and useful theory that unfortunately this lecture does not have time to explore. But we will mention that the above two LPs – Profit-Maximizing Production and Min-Cost Procurement – are *duals* of each other, so their optimal objective values are actually equal to each other and there are a number of other relationships between their feasible and optimal solutions.

4.1 Zero-sum games

Consider the problem of finding an optimal strategy in a finite two-player zero-sum game. Alice and Bob have finite strategy spaces. Alice's utility is u_{ij} if she plays her ith action while Bob plays his jth.

We want to find a probability distribution x for Alice that solves

$$\max_x \min_j \sum_i x_i u_{ij}.$$

First, we need to write down the constraints on the solution variables: a probability distribution is a nonnegative vector that sums to one.

$$\sum_i x_i = 1$$
$$x \geq 0.$$

Now, however, we need to translate the objective into a linear program. A nice trick is to translate an expression of the form $\min_j f(j) = \alpha$ into a statement of the form "for all j, $f(j) \geq \alpha$." If we are trying to maximize the variable α, then it will be set to the minimum $f(j)$, as required.

Optimal Strategy in Zero-Sum Game

$$\begin{aligned} & \max_{x,\alpha} \quad \alpha \\ \text{s.t.} \quad & \sum_i x_i u_{ij} \geq \alpha && \forall j \\ & \sum_i x_i = 1 \\ & x \geq \vec{0}. \end{aligned}$$

5 Integer-Linear Programming

Often, we have a problem that is written as a linear program, plus an extra constraint that some of the variables must be integers. This is called an **Integer-Linear Program (ILP)**.

For example, consider maximum matching. We have an undirected graph $G = (V, E)$. We create a variable x_{uv} for each edge $\{u, v\} \in E$. The variable is one if we include the edge in our matching and zero otherwise. Therefore, the optimization problem is:

Maximum Matching

$$\begin{aligned} & \max_x \quad \sum_{\{u,v\} \in E} x_{uv} && (2) \\ \text{s.t.} \quad & \sum_v x_{u,v} \leq 1 && \forall u \in V \\ & x_{uv} \in \{0, 1\} && \forall \{u, v\} \in E. \end{aligned}$$

The last constraint says that every x_{uv} must either be zero or one. Another way to phrase it is as follows, where $\mathbb{Z}$ is the set of integers.

$$\begin{aligned} x_{uv} &\leq 1 && \forall \{u, v\} \in E \\ x_{uv} &\geq 0 && \forall \{u, v\} \in E \\ x_{uv} &\in \mathbb{Z} && \forall \{u, v\} \in E. \end{aligned}$$

If we delete the final constraint, that x_{uv} be integral, we obtain a linear program. This is called taking the **LP relaxation** of an ILP: we "relax" the constraints on the variables to allow them to be real numbers. However, it would not solve maximum matching, as shown by the example of the triangle graph (Exercise 5).

In general, solving integer-linear programs is NP-hard. One can formulate NP-complete problems such as 3-SAT as ILPs. Therefore, we do not have efficient algorithms to solve ILPs in polynomial time. For some problems, like maximum matching, we can use other polynomial-time algorithms rather than ILP solvers. For other problems, like 3-SAT, no known polynomial-time algorithm exists. For such problems, often, we can get a pretty good approximation or even solve it exactly using linear programming techniques. For example, one approach is to first solve the LP relaxation, then "round" the solution so that the variables are integers again. However, this is only a heuristic and is not guaranteed to optimally solve the ILP.

Exercise 5. Consider the matching ILP, Program 2, on the triangle graph (i.e. complete graph on three vertices). What is the size of a maximum matching in this graph, i.e. the solution to the ILP?

Now consider the LP relaxation. Find a solution whose value is strictly higher than the value of the ILP.

Hint: the solution should assign numbers strictly between 0 *and* 1 *to at least some of the variables.*

You may recall that we can apply max-flow algorithms to solve maximum matching on *bipartite* graphs. In this case, the Integrality Theorem implied that an optimal solution would indeed be integral, i.e. send flow either zero or one on each edge, i.e. cause each edge to either be matched ($x_{uv} = 1$) or not ($x_{uv} = 0$). So in the case of bipartite graphs, the value of the ILP and its LP relaxation are the same. However, Exercise 5 proves that this is not true on general graphs.

5.1 ILP example: scheduling

Integer-linear programs are extremely useful in practice. Many scheduling problems, for example, need integer solutions: a worker should either be scheduled for a shift or not, an airplane must either take one route or another; etc. Often, in such problems, exact optimality is not required. So although they may be NP-hard, it suffices to use an LP approach to efficiently obtain a reasonably good solution.

Problem setup. Suppose you are in charge of a chess tournament with k players. You will schedule a series of rounds where some of the players are paired up in each round. Each player should play each other player exactly twice over the course of the tournament. A player can have a "bye" in a round and not play anyone, for example, if there is an odd number of players.

Variables and constraints. To formalize the problem, let $x_{ijt} = 1$ if player i plays j in round t, and $x_{ijt} = 0$ otherwise. Then we have the following constraints:

$$\sum_{j,t} x_{ijt} = 2 \qquad \forall i \quad (i \text{ plays } j \text{ twice})$$

$$\sum_{j} x_{ijt} \leq 1 \qquad \forall i, t \quad (i \text{ plays at most one game/round})$$

$$x_{ijt} = x_{jit} \qquad \forall i, j, t \quad (\text{if } i \text{ plays } j, \text{ then } j \text{ plays } i)$$

$$x_{ijt} \in \{0, 1\} \qquad \forall i, j, t \quad (\text{a game is either played or not}).$$

But that is not all. Some of the players have notified you of conflicts. They will not be able to play in certain rounds. Suppose we have a set C of the conflicts (i, t).

$$\sum_{j} x_{ijt} = 0 \qquad \forall i, t \in C.$$

Oh, and another thing. Players don't like to play each other twice in a row in back-to-back rounds.

$$x_{ijt} + x_{ij(t+1)} \leq 1 \qquad \forall i, j, t.$$

Objective. If the goal is just to find a feasible schedule, then we can actually put anything in as our constraint, even $\vec{0}$, and ask a solver to find a solution. But suppose that you would like to minimize the number of rounds needed to run the tournament. How can we capture that with a linear objective? We can create binary variables $\{y_t\}$ that are one if round t is needed and zero otherwise. This gives extra constraints

$$y_t \in \{0, 1\} \qquad \forall t.$$

How can we tell if round t is needed? With k players, the total number of games in a given round is at most $2k$, because we count a game once for both players. So we can add the constraint

$$\sum_{i,j} x_{ijt} \leq 2ky_t \qquad \forall t.$$

Now, there are two cases: if there are no games at all in round t, then we can set $y_t = 0$. If there are any games at all, we must set $y_t = 1$.

Finally, our objective is

$$\min_{x,y} \sum_t y_t.$$

Takeaways. LPs and ILPs are useful because they allow us to express the idiosyncratic constraints of our particular problem, such as no re-playing on back-to-back days. We throw all of these constraints together and invoke the solver. Of course, if the problem comes back infeasible, we may have to change our approach.