Chapter 7. Linear programming and reductions

An introduction to linear programming

In a linear programming problem we are given *a set of variables*, and we want to *assign real values to them* so as to

(1) satisfy a set of linear equations and/or linear inequalities involving these variables, and

(2) maximize or minimize a given *linear objective function*.

Example: profit maximization

A boutique chocolatier has two products:

- its flagship assortment of triangular chocolates, called **Pyramide**,
- and the more decadent and deluxe **Pyramide Nuit**.

How much of each should it produce to maximize profits?

- Every box of Pyramide has a a profit of \$1.
- Every box of Nuit has a profit of \$6.
- The daily demand is limited to at most 200 boxes of Pyramide and 300 boxes of Nuit.
- The current workforce can produce a total of at most 400 boxes of chocolate per day.

LP formulation

Objective function	$\max\ x_1 + 6x_2$
Constraints	$x_1 \leq 200$
	$x_2 \leq 300$
	$x_1 + x_2 \leq 400$
	$x_1, x_2 \geq 0$

A linear equation in x_1 and x_2 defines *a line in the two-dimensional (2D) plane*, and a linear inequality designates *a half-space*, the region on one side of the line.

Thus the set of all **feasible solutions** of this linear program, that is, *the points* (x_1, x_2) *which satisfy all constraints*, is the intersection of five half-spaces.

It is a convex polygon.

The optimal solution

We want to find the point in this polygon at which the objective function – the profit – is maximized.

The points with a profit of c dollars lie on the line $x_1 + 6x_2 = c$, which has a **slope** of $-1/6$.

As c increases, this "profit line" moves parallel to itself, up and to the right. Since the goal is to maximize c, we must move the line as far up as possible, while still touching the feasible region.

The optimum solution will be *the very last feasible point* that the profit line sees and must therefore be a vertex of the polygon.

The optimal solution (cont'd)

It is a general rule of linear programs that *the optimum is achieved at a vertex of the feasible region*.

The only exceptions are cases in which there is no optimum; this can happen in two ways:

1. The linear program is **infeasible**; that is, the constraints are so tight that it is impossible to satisfy all of them. For instance,

$$x \leq 1, \qquad x \geq 2.$$

2. The constraints are so loose that the feasible region is **unbounded**, and it is possible to achieve arbitrarily high objective values. For instance,

$$\begin{array}{c} \max\ x_1 + x_2 \\ x_1, x_2 \geq 0 \end{array}$$

Solving linear programs

Linear programs (LPs) can be solved by *the simplex method*, devised by George Dantzig in 1947.

This algorithm starts at a vertex, and repeatedly looks for an *adjacent vertex* (connected by an edge of the feasible region) of better objective value.

In this way it does *hill-climbing* on the vertices of the polygon, walking from neighbor to neighbor so as to steadily increase profit along the way.

Upon reaching a vertex that has no better neighbor, simplex declares it to be optimal and halts.

Why does this local test imply global optimality? By simple geometry – think of the profit line passing through this vertex. Since all the vertex's neighbors lie below the line, the rest of the feasible polygon must also lie below this line.

More products

The chocolatier decides to introduce a third and even more exclusive line of chocolates, called **Pyramide Luxe**. One box of these will bring in a profit of \$13.

Let x_1, x_2, x_3 denote the number of boxes of each chocolate produced daily, with x_3 referring to Luxe.

The old constraints on x_1 and x_2 persist, although the labor restriction now extends to x_3 as well: the sum of all three variables can be at most 400.

What's more, it turns out that Nuit and Luxe require the same packaging machinery, except that Luxe uses it *three times as much*, which imposes another constraint $x_2 + 3x_3 \leq 600$.

LP

$$\begin{array}{c} \max\ x_1 + 6x_2 + 13x_3 \\ x_1 \leq 200 \\ x_2 \leq 300 \\ x_1 + x_2 + x_3 \leq 400 \\ x_2 + 3x_3 \leq 600 \\ x_1, x_2, x_3 \geq 0 \end{array}$$

The space of solutions is now *three-dimensional*.
Each linear equation defines a 3D plane, and each inequality a half-space on one side of the plane. The feasible region is an intersection of seven half-spaces, *a polyhedron*.

A profit of c corresponds to the plane $x_1 + 6x_2 + 13x_3 = c$. As c increases, this profit-plane moves parallel to itself, further and further into the positive **orthant** until it no longer touches the feasible region.
The point of final contact is the optimal vertex: $(0, 300, 100)$, with total profit \$3100.

How would the simplex algorithm behave on this modified problem? A possible trajectory:

$$\underset{\$0}{(0,0,0)} \rightarrow \underset{\$200}{(200,0,0)} \rightarrow \underset{\$1400}{(200,200,0)} \rightarrow \underset{\$2800}{(200,0,200)} \rightarrow \underset{\$3100}{(0,300,100)}$$

A magic trick called duality

Here is why you should believe that $(0, 300, 100)$, with a total profit of \$3100, is the optimum:

Recall

$$\begin{array}{c} \max\ x_1 + 6x_2 + 13x_3 \\ x_1 \leq 200 \\ x_2 \leq 300 \\ x_1 + x_2 + x_3 \leq 400 \\ x_2 + 3x_3 \leq 600 \\ x_1, x_2, x_3 \geq 0 \end{array}$$

Add the second inequality to the third, and add to them the fourth multiplied by 4.

The result is the inequality

$$x_1 + 6x_2 + 13x_3 \leq 3100.$$

Example: production planning

The company makes *handwoven carpets*, a product for which the demand is extremely seasonal.

Our analyst has just obtained demand estimates for all months of the next calendar year: $d_1, d_2, \ldots, d_{12}$, ranging from 440 to 920.
We currently have 30 employees, each of whom makes 20 carpets per month and gets a monthly salary of \$2000. We have no initial surplus of carpets.

How can we handle *the fluctuations in demand*? There are three ways:

1. **Overtime**, but this is expensive since overtime pay is 80% more than regular pay. Also, workers can put in at most 30% overtime.
2. **Hiring** and **firing**, but these cost \$320 and \$400, respectively, per worker.
3. **Storing surplus production**, but this costs \$8 per carpet per month. We currently have no stored carpets on hand, and we must end the year without any carpets stored.

LP formulation

w_i	=	number of workers during ith month; $w_0 = 30$.
x_i	=	number of carpets made during ith month.
o_i	=	number of carpets made by overtime in month i.
h_i, f_i	=	number of workers hired and fired, respectively, at beginning of month i.
s_i	=	number of carpets stored at end of month i; $s_0 = 0$.

First, all variables must be nonnegative:

$$w_i, x_i, o_i, h_i, f_i, s_i \geq 0, i = 1, \ldots, 12.$$

The total number of carpets made per month consists of regular production plus overtime:

$$x_i = 20w_i + o_i$$

(one constraint for each $i = 1, \ldots, 12$).

LP formulation (cont'd)

The number of workers can potentially change at the start of each month:

$$w_i = w_{i-1} + h_i - f_i.$$

The number of carpets stored at the end of each month is what we started with, plus the number we made, minus the demand for the month:

$$s_i = s_{i-1} + x_i - d_i.$$

And overtime is limited:

$$o_i \leq 6w_i.$$

The objective function is to minimize the total cost:

$$\min \quad 2000 \sum_i w_i + 320 \sum_i h_i + 400 \sum_i f_i + 8 \sum_i s_i + 180 \sum_i o_i.$$

Integer linear programming

The optimum solution might turn out to be *fractional*; for instance, it might involve *hiring 10.6 workers in the month of March.*

This number would have to be **rounded** to either 10 or 11 in order to make sense, and the overall cost would then increase correspondingly.
In the present example, most of the variables take on fairly large (double-digit) values, and thus rounding is unlikely to affect things too much.

There are other LPs, however, in which rounding decisions have to be made very carefully in order to end up with an integer solution of reasonable quality.

In general, there is a tension in linear programming between *the ease of obtaining fractional solutions* and *the desirability of integer ones*.

As we shall see in Chapter 8, finding the optimum integer solution of an LP is an important but very hard problem, called **integer linear programming**.

Reductions

We want to solve Problem P.

We already have an algorithm that solves Problem Q.

If any subroutine for Q can also be used to solve P, we say P **reduces to** Q.
Often, P is solvable by *a single call to Q's subroutine*, which means any instance x of P can be transformed into an instance y of Q such that $P(x)$ can be deduced from $Q(y)$.

Variants of linear programming

A general linear program has many degrees of freedom:

1. It can be either a maximization or a minimization problem.
2. Its constraints can be equations and/or inequalities.
3. The variables are often restricted to be nonnegative, but they can also be unrestricted in sign.

We will now show that these various LP options can all be *reduced* to one another via simple transformations.

Variants of linear programming (cont'd)

1. To turn a maximization problem into a minimization (or vice versa), just multiply the coefficients of the objective function by -1.

2a. To turn an inequality constraint like $\sum_{i=1}^{n} a_i x_i \leq b$ into an equation, introduce a new variable s and use

$$\sum_{i=1}^{n} a_i x_i + s = b$$
$$s \geq 0.$$

This s is called the **slack variable** for the inequality.

2b. To change an equality constraint into inequalities is easy: rewrite $ax = b$ as the equivalent pair of constraints $ax \leq b$ and $ax \geq b$.

3. Finally, to deal with a variable x that is unrestricted in sign, do the following:
   - Introduce two nonnegative variables, $x^+, x^- \geq 0$.
   - Replace x, wherever it occurs in the constraints or the objective function, by $x^+ - x^-$.

Standard form

Therefore, we can reduce any LP (maximization or minimization, with both inequalities and equations, and with both nonnegative and unrestricted variables) into an LP of a much more constrained kind that we call the **standard form**:

- the variables are all nonnegative,
- the constraints are all equations,
- and the objective function is to be minimized.

$$\begin{array}{c} \max\ x_1 + 6x_2 \\ x_1 \leq 200 \\ x_2 \leq 300 \\ x_1 + x_2 \leq 400 \\ x_1, x_2, x_3 \geq 0 \end{array} \quad \Longrightarrow \quad \begin{array}{c} \min\ -x_1 - 6x_2 \\ x_1 + s_1 = 200 \\ x_2 + s_2 = 300 \\ x_1 + x_2 + s_3 = 400 \\ x_1, x_2, s_1, s_2, s_3 \geq 0 \end{array}$$

Flows in networks

Shipping oil

We have a network of pipelines along which oil can be sent. The *goal* is to ship as much oil as possible from the **source** s and the **sink** t.

Each pipeline has a *maximum capacity* it can handle, and there are no opportunities for storing oil en route.

Maximizing flow

The networks consist of a directed graph $G = (V, E)$; two special nodes $s, t \in V$, which are, respectively, a **source** and **sink** of G; and **capacities** $c_e > 0$ on the edges.

We would like to send as much oil as possible from s to t without exceeding the capacities of any of the edges.
A particular shipping scheme is called a **flow** and consists of a variable f_e for each edge e of the network, satisfying the following two properties:

1. It doesn't violate edge capacities: $0 \leq f_e \leq c_e$ for all $e \in E$.
2. For all nodes u except s and t, the amount of folw entering u equals the amount leaving

$$\sum_{(w,v) \in E} f_{wu} = \sum_{(u,z) \in E} f_{uz}$$

In other words, flow is *conserved*.

Maximizing flow (cont'd)

The **size** of a flow is the total quantity sent from s to t and, by the *conservation principle*, is equal to the quantity leaving s:

$$\texttt{size}(f) = \sum_{(s,u) \in E} f_{su}.$$

In short, our goal is to assign values to $\{f_e \mid e \in E\}$ that will satisfy a set of linear constraints and maximize a linear objective function.
But this is a linear program! The maximum-flow problem reduces to linear programming.

A closer look at the algorithm

All we know so far of the simplex algorithm is the vague *geometric intuition* that it keeps making local moves on the surface of a convex feasible region, successively improving the objective function until it finally reaches the optimal solution.

The behavior of simplex has an *elementary interpretation:*

Start with zero flow.

Repeat: choose an appropriate path from s to t, and increase flow along the edges of this path as much as possible.

A closer look at the algorithm (cont'd)

There is just one complication.

What if we choose a path that blocks all other paths?

Simplex gets around this problem by also allowing paths to *cancel existing flow*.

To summarize, in each iteration simplex looks for an s-t path whose edges (u, v) can be of two types:

1. (u, v) is in the original network, and is not yet *at full capacity*.
2. The reverse edge (v, u) is in the original network, and there is some flow along it.

If the current flow is f, then in the first case, edge (u, v) can handle up to $c_{uv} - f_{uv}$ additional units of flow, and in the second case, up to f_{vu} additional units (canceling all or part of the existing flow on (v, u)).

A closer look at the algorithm (cont'd)

These flow-increasing opportunities can be captured in a **residual network** $G^f = (V, E^f)$, which has exactly the two types of edges listed, with residual capacities c^f:

$$\begin{cases} c_{uv} - f_{uv} & \text{if } (u, v) \in E \text{ and } f_{uv} < c_{uv} \\ f_{vu} & \text{if } (v, u) \in E \text{ and } f_{vu} > 0. \end{cases}$$

Thus we can equivalently think of simplex as choosing an s-t path in the residual network.

By simulating the behavior of simplex, we get a *direct algorithm* for solving max-flow.

It proceeds in iterations, each time explicitly constructing G^f, finding a suitable s-t path in G^f by using, say, a linear-time breadth-first search, and halting if there is no longer any such path along which flow can be increased.

Cuts

A truly remarkable fact:

not only does simplex correctly compute a maximum flow, but it also generates a short proof of the optimality of this flow!

An (s, t)-**cut** partitions the vertices into two *disjoint* groups L and R such that $s \in L$ and $t \in R$. Its **capacity** is the total capacity of the edges from L to R, and as argued previously, is an *upper bound* on *any* flow:

Pick any flow f and any (s, t)-cut (L, R). Then $\texttt{size}(f) \leq \texttt{capacity}(L, R)$.

A certificate of optimality

Theorem (Max-flow min-cut)

The size of the maximum flow in a network equals the capacity of the smallest (s, t)-cut.

Proof.

Suppose f is the final flow when the algorithm terminates.
We know that node t is no longer reachable from s in the residual network G^f.
Let L be the nodes that are reachable from s in G^f, and let $R = V \setminus L$ be the rest of the nodes. We claim that

$$\texttt{size}(f) = \texttt{capacity}(L, R)$$

To see this, observe that by the way L is defined, any edge going from L to R must be at full capacity (in the current flow f), and any edge from R to L must have zero flow.
Therefore the net flow across (L, R) is exactly the capacity of the cut. □

Efficiency

Each iteration of our maximum-flow algorithm is efficient, requiring $O(|E|)$ time if a DFS or BFS is used to find an s-t path.

But how many iterations are there?

Suppose all edges in the original network have integer capacities $\leq C$.
Then on each iteration of the algorithm, the flow is always an integer and increases by an integer amount. Therefore, since the maximum flow is at most $C|E|$, the number of iterations is at most this much.

If paths are chosen in a sensible manner – in particular, by using a BFS, which finds the path with the fewest edges – then the number of iterations is at most $O(|V| \cdot |E|)$, no matter what the capacities are.

This latter bound gives an overall running time of $O(|V| \cdot |E|^2)$ for maximum flow.

Bipartite matching

The problem

A graph with nodes on the left representing boys and nodes on the right representing girls.

There is an edge between a boy and girl if they like each other.
Is it possible to choose couples so that everyone has exactly one partner, and it is someone they like?

In graph-theoretic jargon, is there a **perfect matching**?

LP formulation

This matchmaking game can be reduced to the maximum-flow problem, and thereby to linear programming!

- Create a new source node s with outgoing edges to all the boys;
- a new sink node t, with incoming edges from all the girls;
- and direct all the edges in the original bipartite graph from boy to girl.
- Finally, give every edge a capacity of 1.

Then there is a perfect matching if and only if this network has a flow whose *size equals the number of couples*.

The maximum-flow problem has the following property: if all edge capacities are integers, then the optimal flow found by our algorithm is *integral*.
We can see this directly from the algorithm, which in such cases would increment the flow by an integer amount on each iteration.

Duality

Recall:

$$\begin{aligned} \max\ & x_1 + 6x_2 \\ & x_1 \leq 200 \\ & x_2 \leq 300 \\ & x_1 + x_2 \leq 400 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

Simplex declares the optimum solution to be $(x_1, x_2) = (100; 300)$, with objective value 1900.

Can this answer be checked somehow?

We take the first inequality and add it to six times the second inequality:

$$x_1 + 6x_2 \leq 2000.$$

Multiplying the three inequalities by 0, 5, and 1, respectively, and adding them up yields

$$x_1 + 6x_2 \leq 1900.$$

Let's investigate the issue by describing what we expect of these three multipliers, call them y_1, y_2, y_3.

Multiplier	Inequality				
y_1	x_1			$\leq$	200
y_2			x_2	$\leq$	300
y_3	x_1	$+$	x_2	$\leq$	400

These y_i's must be nonnegative, for otherwise they are unqualified to multiply inequalities.

After the multiplication and addition steps, we get the bound:

$$(y_1 + y_3)x_1 + (y_2 + y_3)x_2 \leq 200y_1 + 300y_2 + 400y_3.$$

We want the left-hand side to look like our objective function $x_1 + 6x_2$ so that the right-hand side is an upper bound on the optimum solution.

$$x_1 + 6x_2 \le 200y_1 + 300y_2 + 400y_3 \quad \text{if} \quad \left\{ \begin{array}{c} y_1, y_2, y_3 \ge 0 \\ y_1 + y_3 \ge 1 \\ y_2 + y_3 \ge 6 \end{array} \right\}$$

We can easily find y's that satisfy the inequalities on the right by simply making them large enough, for example $(y_1, y_2, y_3) = (5, 3, 6)$.
But these particular multipliers would tell us that the optimum solution of the LP is at most

$$200 \cdot 5 + 300 \cdot 3 + 400 \cdot 6 = 4300,$$

a bound that is far too loose to be of interest.

What we want is a bound that is as tight as possible, so we should minimize $200y_1 + 300y_2 + 400y_3$ subject to the preceding inequalities. And this is a *new linear program*!

The dual program

$$\begin{array}{c} \min \ 200y_1 + 300y_2 + 400y_3 \\ y_1 + y_3 \ge 1 \\ y_2 + y_3 \ge 6 \\ y_1, y_2, y_3 \ge 0 \end{array}$$

By design, any feasible value of this *dual* LP is an upper bound on the original *primal* LP.

So if we somehow find a pair of primal and dual feasible values that are equal, then they must both be optimal. Here is just such a pair:

Primal: $(x_1, x_2) = (100, 300)$; **Dual:** $(y_1, y_2, y_3) = (0, 5, 1)$.

They both have value 1900, and therefore they certify each other's optimality.

A generic primal LP in matrix-vector form and its dual

Primal LP:

$$\begin{array}{c} \max \mathbf{c}^T \mathbf{x} \\ \mathbf{A}\mathbf{x} \le \mathbf{b} \\ \mathbf{x} \ge 0 \end{array}$$

Dual LP:

$$\begin{array}{c} \min \mathbf{y}^T \mathbf{b} \\ \mathbf{y}^T \mathbf{A} \ge \mathbf{c}^T \\ \mathbf{y} \ge 0 \end{array}$$

Theorem (**Duality**)

If a linear program has a bounded optimum, then so does its dual, and the two optimum values coincide.

Zero-sum games

Rock-paper-scissors game

Payoff matrix G**:** the Row player vs. the Column player

	r	p	s
r	0	-1	1
p	1	0	-1
s	-1	1	0

Mixed strategy

We play this game *repeatedly*.

If Row always makes the same move, Column will quickly catch on and will always play the countermove, winning every time.

Therefore Row should *mix things up*: we can model this by allowing Row to have a mixed strategy, in which on each turn she plays r with probability x_1, p with probability x_2, and s with probability x_3.
This strategy is specified by the vector $\mathbf{x} = (x_1, x_2, x_3)$, positive numbers that add up to 1.

Similarly, Column's mixed strategy is some $\mathbf{y} = (y_1, y_2, y_3)$.

On any given round of the game, there is an $x_i y_j$ chance that Row and Column will play the ith and jth moves, respectively. Therefore the *expected (average) payoff* is

$$\sum_{i,j} G_{ij} \cdot \text{Prob[Row plays } i\text{, Column plays } j] = \sum_{i,j} G_{ij} x_i y_j.$$

Row wants to *maximize* this, while Column wants to *minimize* it.

Completely random strategy

Suppose Row plays the "*completely random*" strategy $x = (1/3, 1/3, 1/3)$.

If Column plays r, then the average payoff will be

$$\frac{1}{3} \cdot 0 + \frac{1}{3} \cdot 1 + \frac{1}{3} \cdot -1 = 0.$$

This is also true if Column plays p, or s.

And since the payoff of any mixed strategy (y_1, y_2, y_3) is just a *weighted average* of the individual payoffs for playing r, p, and s, *it must also be zero*.

$$\sum_{i,j} G_{ij} x_i y_j = \sum_{i,j} G_{ij} \cdot \frac{1}{3} y_j = \sum_j y_j \left(\sum_i \frac{1}{3} G_{ij} \right) = \sum_j y_j \cdot 0 = 0.$$

Completely random strategy (cont'd)

Thus by playing the "completely random" strategy, Row *forces an expected payoff of zero*, no matter what Column does.
This means that Column cannot hope for a negative expected payoff (remember that he wants the payoff to be as small as possible).

Symmetrically, if Column plays the completely random strategy, he also forces an expected payoff of zero, and thus Row cannot hope for a positive expected payoff.

The best each player can do is to play completely randomly, with an expected payoff of zero.

Two scenarios

1. First Row *announces* her strategy, and then Column picks his.
2. First Column announces his strategy, and then Row chooses hers.

We've seen that the average payoff is the same (zero) in either case if both parties play optimally.
But this might well be due to the high level of symmetry in rock-paper-scissors.

In general games, we'd expect the first option to *favor Column*, since he knows Row's strategy and can fully exploit it while choosing his own.

Likewise, we'd expect the second option to *favor Row*.

Amazingly, this is not the case: if both play optimally, then it doesn't hurt a player to announce his or her strategy in advance!

What's more, this remarkable property is a consequence of and in fact – equivalent to – *linear programming duality*.

Presidential election

There are two candidates for office, and the moves they make correspond to campaign issues on which they can focus (the initials stand for **economy**, **society**, **morality**, and **tax cut**).

The payoff entries of G are millions of votes lost by Column.

	m	t
e	3	-1
s	-2	1

Pure strategy

Suppose Row announces that she will play the mixed strategy $x = (1/2, 1/2)$. What should Column do?

Move m will incur an expected loss of $1/2$, while t will incur an expected loss of 0. The best response of Column is therefore the *pure strategy* $\mathbf{y} = (0, 1)$.

More generally, once Row's strategy $\mathbf{x} = (x_1, x_2)$ is fixed, there is always a pure strategy that is optimal for Column: either move m, with payoff $3x_1 - 2x_2$, or t, with payoff $-x_1 + x_2$, whichever is smaller.

Therefore, if Row is forced to announce $\mathbf{x}$ before Column plays, she knows that his best response will achieve an expected payoff of $\min\{3x_1 - 2x_2, -x_1 + x_2\}$. She should choose $\mathbf{x}$ defensively to *maximize her payoff against this best response*:

Pick (x_1, x_2) that maximizes $\min\{3x_1 - 2x_2, -x_1 + x_2\}$.

LP formulation

$$z = \min\{3x_1 - 2x_2, -x_1 + x_2\} \quad \text{is equivalent to} \quad \begin{array}{c} \max\ z \\ z \leq 3x_1 - 2x_2 \\ z \leq -x_1 + x_2 \end{array}$$

Row needs to chooses x_1 and x_2 to maximize this z:

$$\begin{array}{c} \max\ z \\ -3x_1 + 2x_2 + z \leq 0 \\ x_1 - x_2 + z \leq 0 \\ x_1 + x_2 = 1 \\ x_1, x_2 \geq 0 \end{array}$$

LP formulation (cont'd)

Symmetrically, if Column has to announce his strategy first, his best bet is to choose the mixed strategy $\mathbf{y}$ that minimizes his loss under Row's best response:

$$\text{Pick } (y_1, y_2) \text{ that minimizes } \max\{3y_1 - y_2, -2y_1 + y_2\}.$$

In LP form:

$$\begin{gathered} \min\ w \\ -3y_1 + y_2 + w \geq 0 \\ 2y_1 - y_2 + w \geq 0 \\ y_1 + y_2 = 1 \\ y_1, y_2 \geq 0 \end{gathered}$$

These two LPs are dual to each other (see Figure 7.11)! Hence, they have the same optimum, call it V.

Value of the game

By solving an LP, Row (the maximizer) can determine a strategy for herself that guarantees an expected outcome of at least V no matter what Column does.

And by solving the *dual* LP, Column (the minimizer) can guarantee an expected outcome of at most V , no matter what Row does.

It follows that this is the *uniquely defined optimal play*: a priori it wasn't even certain that such a play existed.

V is known as the value of the game.

In our example, it is $1/7$ and is realized when Row plays her optimum mixed strategy $(3/7, 4/7)$ and Column plays his optimum mixed strategy $(2/7, 5/7)$.

Min-max theorem of games

Theorem

$$\max_{\mathbf{x}} \min_{\mathbf{y}} \sum_{i,j} G_{ij} x_i y_j \quad = \quad \min_{\mathbf{y}} \max_{\mathbf{x}} \sum_{i,j} G_{ij} x_i y_j$$

This is surprising, because the left-hand side, in which Row has to announce her strategy first, should presumably be better for Column than the right-hand side, in which he has to go first.

Duality equalizes the two.

The simplex algorithm

General description

```
let v be any vertex of the feasible region
while there is a neighbor v' of v with better objective value:
    set v = v'
```

Say there are n variables, $x_1, \ldots, x_n$.

Any setting of the x_i's can be represented by an n-tuple of real numbers and plotted in *n-dimensional space*.

A linear equation involving the x_i's defines a *hyperplane* in this same space $\mathbb{R}^n$, and the corresponding linear inequality defines a *half-space*, all points that are either precisely on the hyperplane or lie on one particular side of it.

Finally, the *feasible region* of the linear program is specified by a set of inequalities and is therefore the intersection of the corresponding half-spaces, a *convex polyhedron*.

Vertices and neighbors in n-dimensional space

Definition

Each vertex is the unique point at which some subset of hyperplanes meet.

Pick a subset of the inequalities. If there is a unique point that satisfies them with equality, and this point happens to be feasible, then it is a vertex.

Each vertex is specified by a set of n inequalities.

Definition

Two vertices are neighbors if they have $n - 1$ defining inequalities in common.

The algorithm

On each iteration, simplex has two tasks:

1. Check whether the current vertex is optimal (and if so, halt).
2. Determine where to move next.

As we will see, both tasks are easy if the vertex happens to be at the origin. And if the vertex is elsewhere, we will transform the coordinate system to move it to the origin!

The convenience for the origin

Suppose we have some generic LP:

$$\begin{aligned} \max \mathbf{c}^T \mathbf{x} \\ \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ \mathbf{x} \geq 0 \end{aligned}$$

where $\mathbf{x}$ is the vector of variables, $\mathbf{x} = (x_1, \ldots, x_n)$.

Suppose the origin is feasible. Then it is certainly a vertex, since it is the unique point at which the n inequalities

$$\{x_1 \geq 0, \ldots, x_n \geq 0\}$$

are *tight*.

Task 1 in the origin

Lemma
The origin is optimal if and only if all $c_i \leq 0$.

Proof.
If all $c_i \leq 0$, then considering the constraints $\mathbf{x} \geq 0$, we can't hope for a better objective value.

Conversely, if some $c_i > 0$, then the origin is not optimal, since we can increase the objective function *by raising x_i*. □

Task 2 in the origin

We can move by increasing some x_i for which $c_i > 0$.

How much can we increase it?

Until we hit some other constraint. That is, we release the tight constraint $x_i \geq 0$ and increase x_i until some other inequality, previously loose, now becomes tight.

At that point, we again have exactly n tight inequalities, so we are at a *new* vertex.

What if our current vertex $\mathbf{u}$ is elsewhere?

The trick is to transform $\mathbf{u}$ into the origin, by shifting the coordinate system from the usual $(x_1, \ldots, x_n)$ to the "local view" from $\mathbf{u}$.

These local coordinates consist of (appropriately scaled) distances $y_1, \ldots, y_n$ to the n hyperplanes (inequalities) that define and enclose u.

Specifically, if one of these enclosing inequalities is $\mathbf{a}_i \cdot \mathbf{x} \leq b_i$, then the distance from a point x to that particular "wall" is

$$y_i = b_i - \mathbf{a}_i \cdot \mathbf{x}.$$

The n equations of this type, one per wall, define the y_i's as linear functions of the x_i's, and this relationship can be inverted to express the x_i's as a linear function of the y_i's.

Rewriting the LP

Thus we can rewrite the entire LP in terms of the y's.

This doesn't fundamentally change it (for instance, the optimal value stays the same), but expresses it in a different coordinate frame.

The revised "local" LP has the following three properties:

1. It includes the inequalities $\mathbf{y} \geq 0$, which are simply the transformed versions of the inequalities defining $\mathbf{u}$.
2. $\mathbf{u}$ itself is the origin in $\mathbf{y}$-space.
3. The cost function becomes $\max c_{\mathbf{u}} + \tilde{\mathbf{c}}^T \mathbf{y}$, where $c_{\mathbf{u}}$ is the value of the objective function at $\mathbf{u}$ and $\tilde{\mathbf{c}}$ is a transformed cost vector.

The starting vertex

In a general LP, the origin might not be feasible and thus not a vertex at all.

However, it turns out that finding a starting vertex can be reduced to an LP and solved by simplex!

Start with any linear program in *standard form*:

$$\min \mathbf{c}^T\mathbf{x} \text{ such that } \mathbf{A}\mathbf{x} = \mathbf{b} \text{ and } \mathbf{x} \geq 0.$$

We first make sure that the right-hand sides of the equations are all nonnegative: if $b_i < 0$, just multiply both sides of the ith equation by -1.

Then we create a new LP as follows:

- Create m new artificial variables $z_1, \ldots, z_m \geq 0$, where m is the number of equations.
- Add z_i to the left-hand side of the ith equation.
- Let the objective, to be *minimized*, be $z_1 + z_2 + \cdots + z_m$.

The starting vertex (cont'd)

For this new LP, it's easy to come up with a starting vertex, namely, the one with $z_i = b_i$ for all i and all other variables zero.

Therefore we can solve it by simplex, to obtain the optimum solution.

There are two cases:

1. If the optimum value of $z_1 + \cdots + z_m$ is zero, then all z_i's obtained by simplex are zero, and hence from the optimum vertex of the new LP we get a starting feasible vertex of the original LP, *just by ignoring the z_i's*.
2. If the optimum objective turns out to be positive: We tried to minimize the sum of the z_i's, but simplex decided that it cannot be zero. But this means that the original linear program is *infeasible*: it needs some nonzero z_i's to become feasible.

Degeneracy

A vertex is degenerate if it is the intersection of more than n faces of the polyhedron, say $n+1$.
Algebraically, it means that if we choose any one of n sets of $n+1$ inequalities and solve the corresponding system of three linear equations in n unknowns, we'll get the *same* solution in all $n+1$ cases.
This is a *serious problem*: simplex may return a *suboptimal* degenerate vertex simply because all its neighbors are identical to it and thus have no better objective.

And if we modify simplex so that it detects degeneracy and continues to hop from vertex to vertex despite lack of any improvement in the cost, it may end up looping forever.

Degeneracy (cont'd)

One way to fix this is by a *perturbation*:

change each b_i by a tiny random amount to $b_i \pm \varepsilon_i$.

This doesn't change the essence of the LP since the ε_i's are tiny, but it has the effect of differentiating between the solutions of the linear systems.

Unboundedness

In some cases an LP is *unbounded*, in that its objective function can be made arbitrarily large (or small, if it's a minimization problem).

If this is the case, simplex will discover it: in exploring the neighborhood of a vertex, it will notice that taking out an inequality and adding another leads to an underdetermined system of equations that has an infinity of solutions.

And in fact (this is an easy test) the space of solutions contains a whole line across which the objective can become larger and larger, all the way to ∞.

In this case simplex halts and complains.

The running time of simplex

What is the running time of simplex, for a generic linear program:

$$\max \mathbf{c}^T\mathbf{x} \text{ such that } \mathbb{A}\mathbf{x} \leq 0 \text{ and } \mathbf{x} \geq 0.$$

where there are n variables and $\mathbf{A}$ contains m inequality constraints?

It is an **iterative** algorithm that proceeds from vertex to vertex.

Let $\mathbf{u}$ be the current vertex, i.e., *unique point at which n inequality constraints are satisfied with equality*. Each of its neighbors shares $n-1$ of these inequalities, so $\mathbf{u}$ can have at most $n \cdot m$ neighbors.

A *naive* way for an iteration: (1) check each potential neighbor to see whether it really is a vertex of the polyhedron, (2) determine its cost.

(2) is easy. (1) involves: *solve a system of n equations* and check whether the result is feasible.

By Gaussian elimination this takes $O(n^3)$ time, giving total $O(mn^4)$ per iteration.

Solving linear systems of equations

Let

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} \qquad \text{and} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Goal: To find $\mathbf{x} = [x_1, x_2, \ldots, x_n]^\mathsf{T}$ such that

$$\mathbf{A}\mathbf{x} = \mathbf{b}.$$

Measure cost by number of multiplicative operations.

Solving upper triangular systems

- Here $a_{i,j} = 0$ for $i > j$, and so

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ 0 & a_{2,2} & a_{2,3} & \cdots & a_{2,n} \\ 0 & 0 & a_{3,3} & \cdots & a_{3,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{n,n} \end{bmatrix}$$

- Back-substitution algorithm:

$$x_i = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=i+1}^{n} a_{i,j} x_j \right) \qquad (1 \le i \le n)$$

- Cost? The ith step uses $n - i$ multiplications and 1 division, and so cost is

$$\sum_{i=1}^{n} (n - i + 1) = \sum_{i=1}^{n} i = \tfrac{1}{2} n(n+1) = \Theta(n^2)$$

Gaussian elimination: reduce to triangular form

- Main idea: Introduce zeros in each column:
  - 1st column: rows $2, \ldots, n$
  - 2nd column: rows $3, \ldots, n$
  - $\vdots$
  - $(n-1)$st column: row n
- How?
  - 1st column: For each row $j = 2, 3, 4, \ldots, n$: subtract $m\times$ 1st row from jth row, where $m = a_{1,j}/a_{1,1}$.
  - 2nd column: For each row $j = 3, 4, \ldots, n$: subtract $m\times$ 2nd row from jth row, where $m = a_{2,j}/a_{2,2}$
  - $\vdots$
  - $(n-1)$st column: For row $j = n$: subtract $m\times$ $(n-1)$st row from nth row, where $m = a_{n,j}/a_{n,n}$
- Cost? $\Theta(n^3)$.

Partial pivoting

- Problem: What if some diagonal element is zero?

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

  Division by zero!
- What about

$$\begin{bmatrix} \varepsilon & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1+\varepsilon \\ 1 \end{bmatrix}.$$

  where ε is small?
- Solution: Partial pivoting—at stage i, swap rows i and r, where $|a_{r,i}|$ is maximal.
- Cost is still $\Theta(n^3)$.

Can we do better than $\Theta(n^3)$?

- The solution to $\mathbf{A}\mathbf{x} = \mathbf{b}$ is given by $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, where $\mathbf{A}^{-1}$ is the inverse of $\mathbf{A}$, which satisfies

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{I} = \mathbf{A}\mathbf{A}^{-1},$$

  and $\mathbf{I}$ is the *identity matrix* (diagonal elements are 1, off-diagonal elements are 0).
- If we know $\mathbf{A}^{-1}$, then can compute $\mathbf{x}$ in $\Theta(n^2)$ operations.
- How hard to compute $\mathbf{A}^{-1}$?

Can we do better than $\Theta(n^3)$?

- Let $M(n)$ be the cost of multiplying two $n \times n$ matrices.
- Let $I(n)$ be the cost of computing the inverse of an $n \times n$ matrix.
- Claim: $M(n) \le I(3n)$, and so $M(n) = O(I(n))$. Indeed, let

$$\mathbf{D} = \begin{bmatrix} \mathbf{I}_n & \mathbf{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_n & \mathbf{B} \\ \mathbf{0} & \mathbf{0} & \mathbf{I}_n \end{bmatrix}$$

  Then

$$\mathbf{D}^{-1} = \begin{bmatrix} \mathbf{I}_n & -\mathbf{A} & \mathbf{A}\mathbf{B} \\ \mathbf{0} & \mathbf{I}_n & -\mathbf{B} \\ \mathbf{0} & \mathbf{0} & \mathbf{I}_n \end{bmatrix}$$

Can we do better than $\Theta(n^3)$?

- What about the opposite direction? Is $I(n) = O(M(n))$?
- The answer is yes! Involves some tricky matrix algebra. You can read the details if you're interested.
- So $I(n) = \Theta(M(n))$.
- Since we can do $n \times n$ matrix multiplication in $\Theta(n^{\log_2 7}) \doteq \Theta(n^{2.81})$ operations, this means that the same is true for computing matrix inverse.
- Hence the same is true for solving linear systems.

Why not use the faster algorithm?

- Overhead: Hidden constants in Θ-notation.
- Hard to implement.
- Algorithm is unstable.

The running time of simplex (cont'd)

A much better way: the mn^4 can be improved to mn.

Recall the *local view* from vertex $\mathbf{u}$. The per-iteration overhead of rewriting the LP in terms of the current local coordinates is just $O((m+n)n)$; since the local view changes only slightly between iterations, in just *one of its defining inequalities*.

Next, to select the *best* neighbor, we recall that the (local view of) the objective function is of the form

$$\max c_{\mathbf{u}} + \tilde{\mathbf{c}} \cdot \mathbf{y}$$

where $c_{\mathbf{u}}$ is the value of the objective function at $\mathbf{u}$.

This immediately identifies a promising direction to move: we pick any $\tilde{c}_i > 0$ (if there is none, then the current vertex is optimal and simplex halts).

Since the rest of the LP has now been rewritten in terms of the $\mathbf{y}$-coordinates, it is easy to determine how much y_i can be increased before some other inequality is violated. (And if we can increase y_i indefinitely, we know the LP is unbounded.)

The running time of simplex (cont'd)

How many iterations could there be?

At most

$$\binom{m+n}{n},$$

i.e., the number of vertices.

It is **exponential** in n. And in fact, there are examples of LPs for which simplex does indeed take an exponential number of iterations.

Simplex is an exponential-time algorithm.

However, such exponential examples do not occur in practice, and it is this fact that makes simplex so valuable and so widely used.

LP in polynomial time

- Simplex for an n-variable LP with m constraints takes time proportional to
$$\binom{m+n}{n} \cdot (m+n)n,$$
which is exponential in n.
- Does this mean that LP requires exponential time? No!
  - Leonid Khachian [1979]: *ellipsoid algorithm* solves any LP in polynomial time.
  - Narenda Karmakar [1984]: *interior point algorithm* also solves any any LP in polynomial time, but with better empirical results than Khachian
- But simplex is still used in practice!

Postscript: circuit evaluation

An ultimate application

We are given a Boolean circuit, that is, a dag of gates of the following types.

- **Input** gates have indegree zero, with value `true` or `false`.
- **AND** gates and OR gates have indegree 2.
- **NOT** gates have indegree 1.

In addition, one of the gates is designated as the **output**.

The CIRCUIT VALUE problem is the following: when the laws of Boolean logic are applied to the gates in topological order, does the output evaluate to `true`?

LP formulation

Create a variable x_g for each gate g, with constraints $0 \leq x_g \leq 1$.

Add additional constraints for each type of gate:

- `true` gate: $x_g = 1$.
- `false` gate: $x_g = 0$.
- OR gate with inputs h and h': $x_g \geq x_h$, $x_g \geq x_{h'}$, $x_g \leq x_h + x_{h'}$.
- AND gate with inputs h and h': $x_g \leq x_h$, $x_g \leq x_{h'}$, $x_g \geq x_h + x_{h'} - 1$.
- NOT gate with input h: $x_g = 1 - x_h$.

These constraints force all the gates to take on exactly the right values – 0 for `false`, and 1 for `true`. We don't need to maximize or minimize anything, and we can read the answer off from the variable x_o corresponding to the **output** gate.

The generality

The CIRCUIT VALUE problem is in a sense the most general problem solvable in polynomial time!
After all, any algorithm will eventually run on a computer, and the computer is ultimately a Boolean combinational circuit implemented on a chip.

If the algorithm runs in polynomial time, it can be rendered as a Boolean circuit consisting of polynomially many copies of the computer's circuit, one per unit of time, with the values of the gates in one layer used to compute the values for the next.
Hence, the fact that CIRCUIT VALUE reduces to linear programming means that *all problems that can be solved in polynomial time do!*