# Linear Programming and Reductions

## 1 Linear Programming

Many of the problems we want to solve by algorithms are *optimization* problems: Find the shortest path, the cheapest spanning tree, the most efficient encoding, the longest increasing subsequence. In an optimization problem we seek a solution which (a) satisfies certain constraints (the path must use edges of the graph and lead from $s$ to $t$, the tree must spann all nodes, the subsequence must be increasing); and (b) is the best possible, with respect to some well-defined criterion, among those that do.

*Linear programming* (or *LP)* is a very general class of optimization problems. In an LP problem we want to find values for certain variables that (a) satisfy a set of linear equations and/or inequalities, and (b) among those values we want to choose the one that maximizes a given linear objective function.

### An example

A company produces three products, and wishes to decide the level of production of each so as to maximize profits. Let $x_1$ be the amount of Product 1 produced in a month, $x_2$ that of Product 2 and $x_3$ of Product 3. Each unit of Product 1 brings to the company a profit of 100, each unit of Product 2 a profit of 600, and each unit of Product 3 a profit of 1400. But not any combination of $x_1, x_2, x_3$ is a possible production plan; there are certain constraints on $x_1$, $x_2$, and $x_3$ (besides the obvious one, $x_1, x_2, x_3 \geq 0$). First, $x_1$ cannot be more than 200, and $x_2$ more than 300 —presumably because of supply limitations. Also, the sum of the three must be, because of labor constraints, at most 400. Finally, it turns out that Products 2 and 3 use the same piece of equipment, with Product 3 three times as much, and hence we have another constraint $x_2 + 3x_3 \leq 600$. What are the best possible levels of production?

We represent the situation by a *linear program*, as follows:

$$\begin{aligned} \max 100x_1 + 600x_2 + 1400x_3 & \\ x_1 &\leq 200 \\ x_2 &\leq 300 \\ x_1 + x_2 &\leq 400 \\ x_2 + 3x_3 &\leq 600 \\ x_1, x_2, x_3 &\geq 0 \end{aligned}$$

The set of all *feasible* solutions of this linear program (that is, all vectors in 3-d space that satisfy all constraints) is precisely the polyhedron shown in Figure 1.

We wish to maximize the objective function $100x_1 + 600x_2 + 400x_3$ over all points of this polyhedron. This means that we want to find the plane that is parallel to the one with equation $100x_1 + 600x_2 + 400x_3 = 0$, touches the polyhedron, and is as far towards the positive orthant as possible. Obviously, the optimum solution will be a vertex of the polyhedron (or the optimum solution may not be unique, but even in this case there will be an optimum vertex). Two other possibilities with linear programming are that (a) the optimum solution may be infinity, that is, the constraints are so loose that we can increase the objective at will, or (b) that there may be no feasible solution — the constraints are too tight.

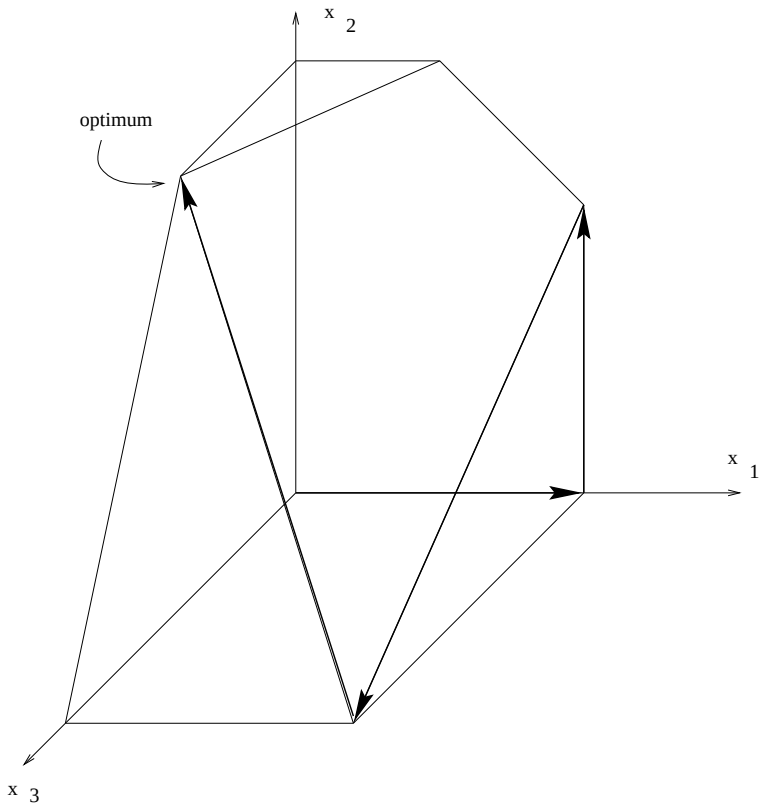


Figure 1: The feasible region

Linear programs can be solved by the *simplex method*, an algorithm devised by George Dantzig in 1947, and which we shall explain in detail soon. The simplex method starts from a vertex (in this case perhaps the vertex $(0, 0, 0)$) and repeatedly looks for a vertex that is adjacent (it is joined to the current vertex by an edge of the polyhedron), and has better objective value. You can think of it as a kind of *hill-climbing* in the vertices of the polytope. When a vertex is found that has no better neighbor, simplex stops and declares this vertex to be the optimum. For example, in the figure one of the possible paths followed by the simplex algorithm is shown.

## Reductions

Often, the most clever way to solve a problem is by *reducing it to an already solved one*. A *reduction* from problem **A** to problem **B** is, informally, an algorithm that solves problem **A** by using any algorithm that solves **B**. That is, we transform any given instance $x$ or **A** to an equivalent instance $y$ of **B**, and then use the algorithm for **B** to solve $y$ (see Figure **??**).

We have seen reductions: In the chapter on dynamic programming, we solved the longest common subsequence problem essentially by reducing it to the problem of finding long paths in an appropriately constructed dag (and then we reduced that problem, in turn, to finding shortest paths in a dag with edge weights $-1$).

The reason why linear programming is a very powerful algorithmic technique is that an astounding variety of problems can be reduced to it; in this chapter we shall see many examples.

There are many different variants of linear programming:

1. a constraint may be an equation or an inequality;
2. we may wish to maximize or minimize the objective function;
3. the variables may or may not be required to be nonnegative.

We next point out that all of these variations *can be reduced to one another!*

1. To turn an inequality constraint, say $\sum_{i=1}^{n} a_i x_i \leq b$, into an equation, we introduce a new variable $s$ (called the *slack variable* for this inequality and required to be nonnegative), and rewrite the inequality as $\sum_{i=1}^{n} a_i x_i + s = b, s \geq 0$. Any combination of values that satisfies the new constraints also satisfies the original one, and vice-versa. Similarly, any inequality $\sum_i a_i x_i \geq b$ is rewritten as $\sum_i a_i x_i - s = b, s \geq 0$; $s$ is now called a *surplus* variable.

   Naturally, it is easy to go in the other direction, and replace an equation by two inequalities: $\sum_{i=1}^{n} a_i x_i = b$ can be written $\sum_{i=1}^{n} a_i x_i \leq b, \sum_{i=1}^{n} a_i x_i \geq b$.
2. An variable $x$ that is unrestricted in sign can be replaced by two nonnegative variables as follows: We introduce two nonnegative variables, $x^+, x^- \geq 0$, and replace $x$, wherever it occurs in the constraints or the objective function, by $x^+ - x^-$. This way, $x$ can take on any real value by appropriately adjusting the new variables.
3. Finally, to turn a maximization problem into a minimization one, or vice-versa, we just multiply the coefficients of the objective function by $-1$.

For example, the linear program with the three products can be rewritten thus:

$$\begin{aligned} \min -100x_1 - 600x_2 - 1400x_3 & \\ x_1 + s_1 &= 200 \\ x_2 + s_2 &= 300 \\ x_1 + x_2 + s_3 &= 400 \\ x_2 + 3x_3 + s_4 &= 600 \\ x_1, x_2, x_3, s_1, s_2, s_3, s_4 &\geq 0 \end{aligned}$$

This form of a linear program, seeking to minimize a linear objective function with equational constraints and nonnegative variables, is called the *standard form*. It is the form solved by the simplex algorithm.

## Production planning

We have the demand estimates for our product for all months of 2005, $d_i : i = 1, \ldots, 12$, and they are very uneven, ranging from 440 to 920. We currently have 30 employees, each of which produce 20 units of the product each month at a salary of 2,000; we have no stock of the product. How can we handle such fluctuations in demand? Three ways:

- overtime —but this is expensive since it costs $80\%$ more than regular production, and has limitations, as workers can only work $30\%$ overtime.

- hire and fire workers —but hiring costs 320, and firing costs 400 per worker.
- store the surplus production —but this costs 8 per item per month. And we must end the year without any items stored.

This rather involved problem can be formulated and solved as a linear program. As in all such reductions, a crucial first step is defining the variables:

- Let $w_i$ be the number of workers we have the $i$th month —we have $w_0 = 60$.
- Let $x_i$ be the production for month $i$.
- $o_i$ is the number of items produced by overtime in month $i$.
- $h_i$ and $f_i$ is the number of workers hired/fired in the beginning of month $i$.
- $s_i$ is the amount of product stored after the end of month $i$.

There is one variable of each of these six kinds for each time period $i = 1, \ldots, 12$. We now must write the constraints:

- $x_i = 20w_i + o_i$ (one constraint for each $i = 1, \ldots, 12$) —the amount produced is the one produced by regular production, plus by overtime.
- $w_i = w_{i-1} + h_i - f_i, w_i \geq 0$, one for each $i$ —the changing number of workers; we know that $w_0 = 30$.
- $s_i = s_{i-1} + x_i - d_i \geq 0$ —the amount stored in the end of this month is what we started with, plus the production, minus the demand. By $d_i$ we denote the known demand at month $i$; and by requirement, $s + 0 = s_{12} = 0$.
- $o_i \leq 6w_i$ —only 30% overtime.

Finally, what is the objective function? It is

$$\min 2000 \sum_{i=1}^{12} w_i + 400 \sum_{i=1}^{12} f_i + 320 \sum_{i=1}^{12} h_i + 8 \sum_{i=1}^{12} s_i + 180 \sum_{i=1}^{12} o_i,$$

a linear function of the variables. Solving this linear program by simplex will give us the optimum business strategy of the company.

## A Communication Network Problem

We have a network whose lines have the bandwidth shown in the Figure. We wish to establish three connections: One between A and B (connection 1), one between B and C (connection 2), and one between A and C (connection 3). We must give each connection at least 2 units of bandwidth, but possibly more. The connection from A to B pays \$3 per unit of bandwidth, from B to C pays \$2, and from A to C pays \$4. Notice that each connection can be routed in two ways (the long and the short path), or by a combination (for example, two units of bandwidth via the short route, and one via the long route). How do we route these calls to maximize the network's revenue?

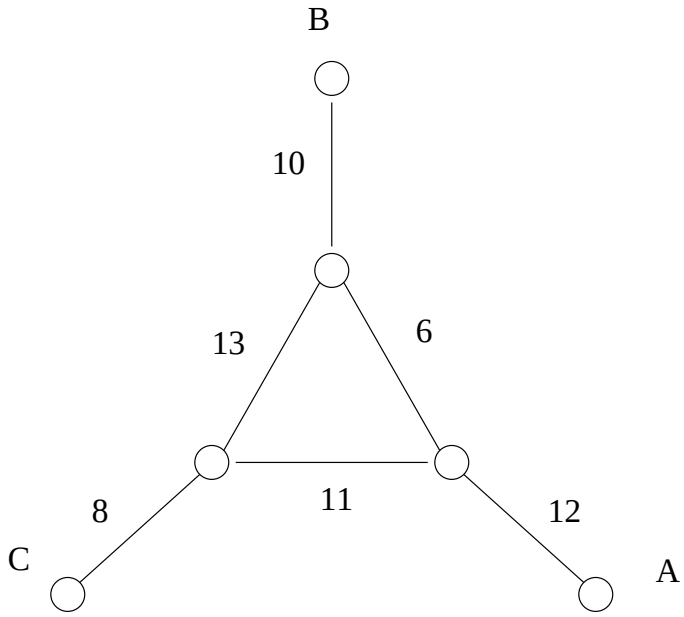


Figure 2: A communication network

This is also a linear program. We have variables for each connection and each path (long or short); for example $x_1$ is the short path bandwidth allocated to connection 1, and $x_2'$ the long path for connection 2. We demand that (1) no edge bandwidth is exceeded, and (2) each call gets a bandwidth of 2.

$$\begin{aligned}
\max 3x_1 + 3x_1' + 2x_2 + 2x_2' + 4x_3 + 4x_3' & \\
x_1 + x_1' + x_2 + x_2' &\leq 10 \\
x_1 + x_1' + x_3 + x_3' &\leq 12 \\
x_2 + x_2' + x_3 + x_3' &\leq 8 \\
x_1 + x_2' + x_3' &\leq 6 \\
x_1' + x_2 + x_3' &\leq 13 \\
x_1' + x_2' + x_3 &\leq 11 \\
x_1 + x_1' &\geq 2 \\
x_2 + x_2' &\geq 2 \\
x_3 + x_3' &\geq 2 \\
x_1, x_1' \ldots, x_3' &\geq 0
\end{aligned}$$

The solution, obtained via simplex in a few microseconds, is the following: $x_1 = 0, x_1' = 7, x_2 = x_2' = 1.5, x_3 = .5, x_3' = 4.5$.

The reader may have expected integer values for the bandwidth; actually, the optimum here is the fractional solution indicated. This is not a problem in this example, but it would be so in production scheduling: There may not be a way to hire 4.5 workers. There is a tension in LP between the ease of obtaining fractional solutions and the desirability of integer ones. As we shall see, finding the optimum integer solution of an LP is a very hard problem, called *integer linear programming*.

Question: Suppose that we removed the constraints stating that each call should receive at least two units. Would the optimum change?

## 2 Simplex: A Rough Outline

We are given a linear program in standard form (minimization with equality constraints and nonnegative variables), such as the one equivalent to our original three-products example, reproduced below:

$$\begin{aligned} \min -100x_1 - 600x_2 - 1400x_3 & \\ x_1 + s_1 &= 200 \\ x_2 + s_2 &= 300 \\ x_1 + x_2 + s_3 &= 400 \\ x_2 + 3x_3 + s_4 &= 600 \\ x_1, x_2, x_3, s_1, s_2, s_3, s_4 &\geq 0 \end{aligned}$$

How should we go about solving it? Notice first that, in contrast to all other optimization problems we have seen so far in this book (shortest path, minimum spanning tree, etc.), this is a *continuous* problem, as its solutions (values of the six variables) live in a continuous space (in particular, the polyhedron of Figure **??**. Is there a way to make this problem finite? For example, in relation to the three-products example we argued that, in order to find the optimum, we need only consider the (finitely many) *vertices* of the polyhedron. But a vertex is a concept from geometry; if we are given a system of equations as above, what are its "vertices"?

Suppose that we are given a linear program in standard form with $m$ equations in $n$ variables (constrained to be nonnegative), where $m < n$. We can choose $m$ of the $n$ variables and set the remaining $n - m$ variables to zero. This gives us a system of $m$ equations with $m$ unknowns; as we know, such systems have in general a unique solution. If this system has a solution, and this solution happens to be nonnegative, then we call it a *vertex* of the linear program.

The term *vertex* refers exactly to the vertices of the polyhedron in Figure **??**. For example, in the linear program above we can set $s_2, s_3, s_4$ to zero, and solve the resulting system of 4 equations with 4 unknowns to obtain $x_1 = 100, x_2 = 300, x_3 = 200, s_1 = 100$ — a vertex, since all values are nonnegative. (This vertex corresponds to the vertex $(100, 300, 200)$ in Figure **??**.) Similarly, by setting $x_1, x_2, x_3$ to zero we obtain the vertex $s_1 = 200, s_2 = 300, s_3 = 400, s_4 = 600$ (the $(0, 0, 0)$ vertex in the figure). And so on.

What makes vertices important is the following property, which we state without proof (besides the sound geometric intuition from Figure **??**:
*If a linear program has an optimum (that is, if it is not infeasible, and the objective function is not unbounded) then the optimum is obtained at a vertex.*

Therefore, linear programming is a finite problem: We need only look at each vertex — more precisely, at each combination of $m$ columns, concentrating on those that have nonnegative solutions. In other words, the property above is a reduction from linear programming to solving systems of linear equations!

This motivates us to pause for a moment and ponder that problem.

## Gaussian Elimination

We are given a system of $n$ linear equations with $n$ unknowns, say

$$\begin{aligned} x_1 - 2x_3 &= 2 \\ x_2 + x_3 &= 3 \\ x_1 + x_2 - x_4 &= 4 \\ x_2 + 3x_3 + x_4 &= 5 \end{aligned}$$

with $n = 4$. We know from highschool a systematic way for solving such systems of equations. This method is based on the following obvious property of such systems:
*We can obtain an equivalent system of linear equations if we replace an equation by the result of adding to that equation a multiple of another equation.*
For example, we can multiply the first equation by $-1$ and add it to the third to obtain the new equation $2x_3 - x_4 = 2$; we can now replace the third equation by the new one, obtaining the equivalent system

$$\begin{aligned} x_1 + x_2 - 2x_3 &= 2 \\ x_2 + x_3 &= 3 \\ x_2 + 2x_3 - x_4 &= 2 \\ x_2 + 3x_3 + x_4 &= 5 \end{aligned}$$

Our choice of the two equations and the multiplier was clever in the following sense: It *eliminated* the variable $x_1$ from the third equation. Since the second and fourth equations did not involve $x_1$ to begin with, now only the first equation contains $x_1$. In other words, ignoring the first equation, we have a system of *three* equations with *three* unknowns: We decreased $n$ by one! If we solve this smaller system, it is easy to substitute the values for $x_1, x_2, x_3$ back to the first equation and solve for $x_1$.

This suggests an algorithm — once more due to Gauss (see Figure **??**).

The running time of the algorithm is obviously $T(n) = T(n-1) + n^2 = O(n^3)$.

## Boxed Subsection: Determinants

Going back to LP and the simplex algorithm, we noticed that the concept of vertex entails a reduction from LP to solving systems of linear equations. Unfortunately the reduction is not efficient, as there are $\binom{n}{m}$ combinations to check — an exponential number.

The simplex algorithm is a clever way of exploring this exponential space of vertices in a greedy fashion. It is based on another important property of vertices. Let us call two vertices *neighbors* if the sets of variables of one can be obtained from the other by adding one variable and deleting another. Obviously, each vertex has at most $m(n-m)$ neighbors.
*If in a linear program that has an optimum a vertex has the property that no neighbor has a better objective function, then it is the optimum vertex.*
This fact suggests a greedy algorithm for LP:

*procedure simplex*($LP$)
`Input:  A linear program` $LP$ `in standard form with` $m$ `equations in` $n$ `nonnegative var`

**Figure 2.1** Gaussian elimination.

*procedure gauss*$(E, X)$
Input: A system $E = \{e_1, \ldots, e_n\}$ of linear equations with $n$ unknowns $X = \{x_1, \ldots, x_n\}$:
$e_1 : a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1$; *dots*; $e_n : a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n$
Output: A solution of the system, if one exists

```
if all coefficients a_ij are zero:
  if all b_j's are zero return ''any real numbers x_1,...,x_n''
  else return ''no solution''
else choose a nonzero coefficient a_pq
for i = 1,...,n:
  if i ≠ p:       for j = 1,...,n:
    set a_ij = a_ij − a_pj/a_pq
    set b_j = b_j − b_p/a_pq
call gauss(E − {e_p}, X − {x_q}) to obtain solution x_i : i = 1,...,n, i ≠ q
set x_q = (b_p − Σ i ≠ q a_pj x_j)/a_pq
```

```
let v be any vertex of LP
while there is a neighbor v' of v
(that is, a set of m columns that differs from v by one
and yields a system of n equations that has a nonnegative solution)
with better objective:
    set v = v'
```

For example, in the LP above we would start with the vertex $v = (0, 0, 0)$ and continue with the vertices ???

There are several problems with simplex as presented here:

- How do we find the starting vertex? In our example we had the obvious starting point $(0, 0, 0)$, but in a general LP? It turns out this can be reduced to LP: Add new artificial variables $a_1, \ldots, a_m \geq 0$, and add $a_i$ to the $i$th equation. Now we have a starting vertex, namely the one with $a_i = b_i$ for all $i$ and all other variables zero. Change the objective to $\sum_{i=1}^{m} a_i$, and solve this LP by simplex. If the optimum objective is positive, this means that the original LP has no feasible solution. If it is zero, we have our starting vertex!

- **Degeneracy:** Two different subsets of $m$ variables may give rise to the same vertex (see Figure[**?**]). This is a serious problem, as it can result in simplex cycling, or returning a suboptimal vertex. It can be taken care of by *perturbation:* Change each $b_i$ by a tiny ransom amount to $b_i + \epsilon_i$.

- **Unboundedness:** If we find out that a set of $m$ columns yields an answer "any $m$ real numbers," this may be evidence that the LP is unbounded, that is, its objective function can be made arbitrarily small.

- **Time complexity:** As presented, simplex runs in time $O(Lm^4n)$, where $L$ is the length of the longest objective-decreasing path of vertices. As it turns out, there are specialized examples where $L$ is exponential in $m$ and $n$! But the $m^4n$ factor is also highly unappetizing!

  This $m^4n$ factor can be improved to $mn$, making simplex a practical algorithm. One improvement is that, once we have solved a system of equations, we can solve another system that differs in one column in time $O(mn)$ instead of $O(m^3)$. An extension of the same method — by treating the objective function as an extra equation — makes it simple to determine which variable to omit and which to add in order to get to a vertex with a better objective without trying all possible $O(mn)$ combinations of variables.

# 3 Network Flows

Suppose that we are given the network of the Figure (top), where the numbers indicate capacities, that is, the amount of flow that can go through the edge in unit time. We wish to find the maximum amount of flow that can go through this network, from $S$ to $T$.

This is yet another problem that can be reduced to linear programming. We have a nonnegative variable for each edge, representing the flow through this edge. These variables are denoted $f_{SA}, f_{SB}, \ldots$ We have two kinds of constraints: Capacity constraints such as $f_{SA} \leq 5$ (a total of 9 such constraints, one for each edge), and flow conservation constraints such as $f_{AD}+f_{BD} = f_{DC}+f_{DT}$, stating that the amount of flow going into $D$ is the same as the amount of flow leaving $D$. (a total of 4 such constraints in our example, one for each node except $S$ and $T$)). We wish to maximize $f_{SA} + f_{SB}$, the amount of flow that leaves $S$, subject to these constraints. It is easy to see that this linear program is equivalent to the max-flow problem. The simplex mathod would correctly solve it.

But in the case of max-flow, it is very instructive to "simulate" the simplex method, to see what effect its various iterations would have on the given network. Simplex would start with the all-zero flow, and would try to improve it. How can it find an improvement in the flow? Answer: It finds a path from $S$ to $T$ (say, by depth-first search), and moves flow along this path of total value equal to the *minimum* capacity of an edge on the path (it can obviously do no better). This is the first iteration of simplex (see the bottom of Figure 3).

How would simplex continue? It would look for another path from $S$ to $T$. Since this time we already partially (or totally) use some of the edges, we should do depth-first search on the edges that have some *residual capacity*, above and beyond the flow they already carry. Thus, the edge $CT$ would be ignored, as if it were not there. The depth-first search would now find the path $S-A-D-T$, and augment the flow by two more units, as shown in the top of Figure 4.

Next, simplex would again try to find a path from $S$ to $T$. The path is now $S-A-B-D-T$ (the edges $C-T$ and $A-D$ are full are are therefore ignored), and we augment the flow as shown in the bottom of Figure 4.

Next the algorithm would again try to find a path. But since edges $A-D$, $C-T$, and $S-B$ are full, they must be ignored, and therefore depth-first search would fail to find a path, after marking the nodes $S, A, C$ as reachable from $S$. *Simplex then returns the flow shown, of value 6, as maximum.*

How can we be sure that it is the maximum? Notice that these reachable nodes define a *cut* (a set of nodes containing $S$ but not $T$), and the *capacity* of this cut (the sum of the capacities

of the edges going out of this set) is 6, the same as the max-flow value. (It must be the same, since this flow passes through this cut.) The existence of this cut establishes that the flow is optimum!

There is a complication that we have swept under the rug so far: When we do depth-first search looking for a path, we use not only the edges that are not completely full, but we must also traverse *in the opposite direction* all edges that already have some non-zero flow. This would have the effect of canceling some flow; canceling may be necessary to achieve optimality, see Figure 5. In this figure the only way to augment the current flow is via the path $S-B-A-T$, which traverses the edge $A-B$ in the reverse direction (a legal traversal, since $A-B$ is carrying non-zero flow).

To summarize: The max-flow problem can be easily reduced to linear programming and solved by simplex. But it is more interesting to understand what simplex would do by following its iterations directly on the network (as we see in the Problems, such simulation actually leads to a polynomial algorithm for max flow). It repeatedly finds a path from $S$ to $T$ along edges that are not yet full (have some non-zero *residual capacity*), and also along any reverse edges with non-zero flow. If an $S-T$ path is found, we augment the flow along this path, and repeat. When a path cannot be found, the set of nodes reachable from $S$ defines a cut of capacity equal to the max-flow. Thus, *the value of the maximum flow is always equal to the capacity of the minimum cut*. This is the important *max-flow min-cut theorem*. One direction (that max-flow$\leq$min-cut) is easy (think about it: how can *any* cut be smaller than *any* flow?). The other direction is proved by the algorithm just described.

# 4 Duality

As it turns out, the max-flow min-cut theorem is a special case of a more general property of linear programs called *duality*. Duality means that a maximization problem and a minimization problem have the property that any feasible solution of the min problem is greater than or equal any feasible solution of the max problem (see the Figure). Furthermore, and more importantly, *they have the same optimum value*.

Consider the network shown in the Figure below, and the corresponding max-flow problem. We know that it can be written as a linear program as follows:

$$
\begin{array}{lccccccc}
\max f_{SA} & +f_{SB} & & & & & & \\
f_{SA} & & & & & & & \leq 3 \\
 & f_{SB} & & & & & & \leq 2 \\
 & & f_{AB} & & & & & \leq 1 \\
 & & & & f_{AT} & & & \leq 1 \\
 & & & & & & f_{BT} & \leq 3 \\
f_{SA} & & -f_{AB} & & -f_{AT} & & & = 0 \\
 & f_{SA} & +f_{AB} & & & & -f_{BT} & = 0 \\
 & & & & & & & f \geq 0P
\end{array}
$$

Consider now the following linear program:

$$
\begin{array}{lllllllll}
\min 3y_{SA} & +2y_{SB} & +y_{AB} & +y_{AT} & +3y_{BT} & & & & \\
y_{SA} & & & & & +u_A & & \geq 1 \\
& y_{SB} & & & & & +u_B & \geq 1 \\
& & y_{AB} & & & -u_A & +u_B & \geq 0 \\
& & & y_{AT} & & -u_A & & \geq 0 \\
& & & & y_{BT} & & -u_B & \geq 0 \\
& & & & & & & y \geq 0D
\end{array}
$$

This LP describes the min-cut problem! To see why, suppose that the $u_A$ variable is meant to be 1 if $A$ is in the same set of the cut as $S$, and 0 otherwise, and similarly for $B$ (naturally, by the definition of a cut, $S$ will always be with $S$ in the cut, and $T$ will never be with $S$). Each of the $y$ variables is to be 1 if the corresponding edge contributes to the cut capacity, and 0 otherwise. Then the constraints make sure that these variables behave exactly as they should. For example, the second constraint states that *if $A$ is not with $S$, then $SA$ must be added to the cut.* The third one states that *if $A$ is with $S$ and $B$ is not* (this is the only case in which the sum $-u_A + u_B$ becomes $-1$), *then $AB$ must contribute to the cut.* And so on. Although the $y$ and $u$'s are free to take values larger than one, they will be "slammed" by the minimization down to 1 or 0.

Let us now observe that these two programs have strikingly symmetric, *dual*, structure. Each variable of $P$ corresponds to a constraint of $D$, and vice-versa. Equality constraints correpond to unrestricted variables (the $u$'s), and inequality constraints to restricted variables. Minimization becomes maximization. The matrices are transpose of one another, and the roles of right-hand side and objective function are interchanged.

Such LP's are called *dual* to each other. It is mechanical, given an LP, to form its dual: *Transpose the matrix, convert maximization to minimization and vice-versa, interchange the roles of the right-hand side and the objective function, and introduce a nonnegative variable for each inequality, and an unrestricted one for each equality.*

By the max-flow min-cut theorem, the two LP's $P$ and $D$ above have the same optimum. *In fact, this is true for general dual LP's!* This is the *duality theorem*, which can be stated as follows (we shall not prove it; the best proof comes from the simplex algorithm, very much as the max-flow min-cut theorem comes from the max-flow algorithm):

*If an LP has a bounded optimum, then so does its dual, and the two optimal values coincide.*

# 5 Matching

It is often useful to *compose* reductions. That is, we can reduce a problem A to B, and B to C, and since C we know how to solve, we end up solving A. A good example is the matching problem.

Suppose that the *bipartite* graph shown in Figure 6 records the compatibility relation between four boys and four girls. We seek a maximum matching, that is, a set of edges that is as large as possible, and in which no two edges share a node. For example, in the figure below there is a *complete* matching (a matching that involves all nodes).

To reduce this problem to max-flow we do this: We create a new source and a new sink, connect the source with all boys and all girls with the sinks, and direct all edges of the original bipartite graph from the boys to the girls. All edges have capacity one. It is easy to see that the maximum flow in this network corresponds to the maximum matching.

Well, the situation is slightly more complicated than was stated above: What is easy to see is that the optimum *integer-valued* flow corresponds to the optimum matching. We would be at a loss interpreting as a matching a flow that ships, for example, .7 units along the edge Al-Eve! Fortunately, what the algorithm in the previous section establishes is that *if the capacities are integers, then the maximum flow is integer.* This is because we only deal with integers throughout the algorithm. Hence *integrality comes for free in the max-flow problem*.

Unfortunately, max-flow is about the only problem for which integrality comes for free. It is a very difficult problem to find the optimum solution (or *any* solution) of a general linear program with the additional constraint that (some or all of) the variables be integers. To see why, notice that the NP-complete *satisfiability* problem can be reduced to *integer linear programming* as this problem is called: The clause $(x_1 \vee \overline{x_2} \vee x_3)$ can be represented by the constraints

$$x_1 + (1 - x_2) + x_3 \geq 1, \quad , 0 \geq x_i \geq 1, \quad x_i \text{integers}.$$

Repeating for all clauses of a given Boolean formula, we get an integer linear program in which finding any feasible solution is equivalent to solving the original instance of satisfiability!

# 6 Games

We can represent various situations of conflict in life in terms of *matrix games*. For example, the game shown below is the *rock-paper-scissors* game. The Row player chooses a row strategy, the Column player chooses a column strategy, and then Column pays to Row the value at the intersection (if it is nagetive, Row ends up paying Column).

$$\begin{array}{c} \\ r \\ p \\ s \end{array} \begin{array}{c} \begin{array}{ccc} r & p & s \end{array} \\ \begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{pmatrix} \end{array}$$

Games do not necessarily have to be symmetric (that is, Row and Column have the same strategies, or, in terms of matrices, $A = -A^T$). For example, in the following fictitious *Presidential Election* game the strategies may be the issues on which a candidate for office may focus (the initials stand for "economy," "society," "morality," and "tax-cut") and the entries are the millions of votes lost by Column.

$$\begin{array}{c} \\ e \\ s \end{array} \begin{array}{c} \begin{array}{cc} m & t \end{array} \\ \begin{pmatrix} 3 & -1 \\ -2 & 1 \end{pmatrix} \end{array}$$

We want to explore how the two players may play "optimally" these games. It is not clear what this means. For example, in the first game there is no such thing as an optimal "pure" strategy — it very much depends on what your opponent does; similarly in the second game. But suppose that you play this repeatedly. Then it makes sense to *randomize*. That is, consider a game given by an $m \times n$ matrix $G_{ij}$; define a *mixed strategy* for the row player to be a vector $(x_1, \ldots, x_m)$, such that $x_i \geq 0$, and $\sum_{i=1}^{m} x_i = 1$. Intuitively, $x_i$ is the probability with which Row plays strategy $i$. Similarly, a mixed strategy for Column is a vector $(y_1, \ldots, y_n)$, such that $y_j \geq 0$, and $\sum_{j=1}^{n} y_j = 1$.

Suppose that, in the Presidential Election game, Row decides to play the mixed strategy $(.5, .5)$. What should Column do? The answer is easy: If the $x_i$'s are given, there is a *pure strategy* (that is, a mixed strategy with all $y_j$'s zero except for one) that is optimal. It is found by comparing the $n$ numbers $\sum_{i=1}^{m} G_{ij}x_i$, for $j = 1, \ldots, n$ (in the Presidential Election game, Column would compare $.5$ with $0$, and of course choose the smallest —remember, the entries denote what Column pays). That is, *if* Column knew Row's mixed strategy, s/he would end up paying the smallest among the $n$ outcomes $\sum_{i=1}^{m} G_{ij}x_i$, for $j = 1, \ldots, n$. On the other hand, Row will seek the mixed strategy that *maximizes this minimum;* that is,

$$\max_x \min_j \sum_{i=1}^{m} G_{ij}x_i.$$

This maximum would be the best possible *guarantee* about an expected outcome that Row can have by choosing a mixed strategy. Let us call this guarantee $z$; what Row is trying to do is solve the following LP:

$$\begin{array}{llll} \max z & & & \\ -z & -3x_1 & +x_2 & \leq 0 \\ -z & +2x_1 & -x_2 & \leq 0 \\ & x_1 & +x_2 & = 1 \end{array}$$

Symmetrically, it is easy to see that Column would solve the following LP:

$$\begin{array}{llll} \min w & & & \\ w & -3y_1 & +2y_2 & \geq 0 \\ w & +y_1 & -y_2 & \geq 0 \\ & y_1 & +y_2 & = 1 \end{array}$$

The crucial observation now is that *these LP's are dual to each other*, and hence have the same optimum, call it $V$.

Let us summarize: By solving an LP, Row can guarantee an expected income of at least $V$, and by solving the dual LP, Column can guarantee an expected loss of at most the same value. It follows that this is the uniquely defined optimal play (it was not *a priori* certain that such a play exists). $V$ is called *the value of the game*. In this case, the optimum mixed strategy for Row is $(3/7, 4/7)$, and for Column $(2/7, 5/7)$, with a value of $1/7$ for the Row player.

The existence of mixed strategies that are optimal for both players and achieve the same value is a fundamental result in Game Theory called *the min-max theorem*. It can be written in equations as follows:

$$\max_x \min_y \sum x_i y_j G_{ij} = \min_y \max_x \sum x_i y_j G_{ij}.$$

It is surprising, because the left-hand side, in which Column optimizes last, and therefore has presumably an advantage, should be intuitively smaller than the righ-hand side, in which Column decides first. Duality equalizes the two, as it does in max-flow min-cut.

# 7 Approximate Separation

An interesting last application: Suppose that we have two sets of points in the plane, the *black points* $(x_i, y_i) : i = 1, \ldots, m$ and the *white points* $(x_i, y_i) : i = m+1, \ldots, m+n$. We wish

to separate them by a straight line $ax + by = c$, so that for all black points $ax + by \leq c$, and for all white points $ax + by \geq c$. In general, this would be impossible. Still, we may want to separate them by a line that minimizes the sum of the "displacement errors" (distance from the boundery) over all misclassified points. Here is the LP that achieves this:

$$\begin{array}{rc} \min e_1 & +e_2 + \ldots + e_m + e_{m+1} + \ldots + e_{m+n} \\ e_1 \geq & ax_1 + by_1 - c \\ e_2 \geq & ax_2 + by_2 - c \\ \vdots & \\ e_m \geq & ax_m + by_m - c \\ e_{m+1} \geq & c - ax_{m+1} - by_{m+1} \\ \vdots & \\ e_{m+n} \geq & c - ax_{m+n} - by_{m+n} \\ & e_i \geq 0 \end{array}$$

# 8 Circuit Evaluation

We have seen many interesting and diverse applications of linear programming. In some sense, this one is the *ultimate* application: Suppose that we are given a *Boolean circui*, that is, a DAG of gates, each of which is either an input gate (indegree zero, and has a value T or F), or an OR gate (indegree two), or an AND gate (indegree two), or a NOT gate (indegree one). One of them is designated as the output gate. We wish to tell if this circuit evaluates (following the laws of Boolean values bottom-up) to T. This is known as *the circuit value problem.*

There is a very simple and automatic way of translating the circuit value problem into an LP: For each gate $g$ we have a variable $x_g$. For all gates we gave $0 \geq x_g \leq 1$. If $g$ is a T input gate, we have the equation $x_g = 1$; if it is F, $x_g = 0$. If it is an OR gate, say of the gates $h$ and $h'$, then we have the inequality $x_g \leq x_h + x_{h'}$. If it is an AND gate of $h$ and $h'$, we have the inequalities $x_g \leq x_h$, $x_g \leq x_{h'}$ (notice the difference). For a NOT gate we say $x_g = 1 - x_h$. Finally, we want to $\max x_o$, where $o$ is the output gate. It is easy to see that the optimum value of $x_o$ will be $1$ if the circuit value if T, and $0$ if it is F.

This is a rather straight-forward reduction to LP, from a problem that may not seem very interesting or hard at first. However, the circuit value problem is in some sense the most general problem solvable in polynomial time! Here is a justification of this statement: After all, a polynomial time algorithm runs on a computer, and the computer is ultimately a Boolean combinational circuit implemented on a chip. Since the algorithm runs in polynomial time, it can be rendered as a circuit consisting of polynomially many superpositions of the computer's circuit. Hence, the fact that circuit value problem reduces to LP means that *all polynomially solvable problems do!*

In our next topic, *Complexity and NP-completeness*, we shall see that a class that contains many hard problems reduces, much the same way, to *integer programming*.

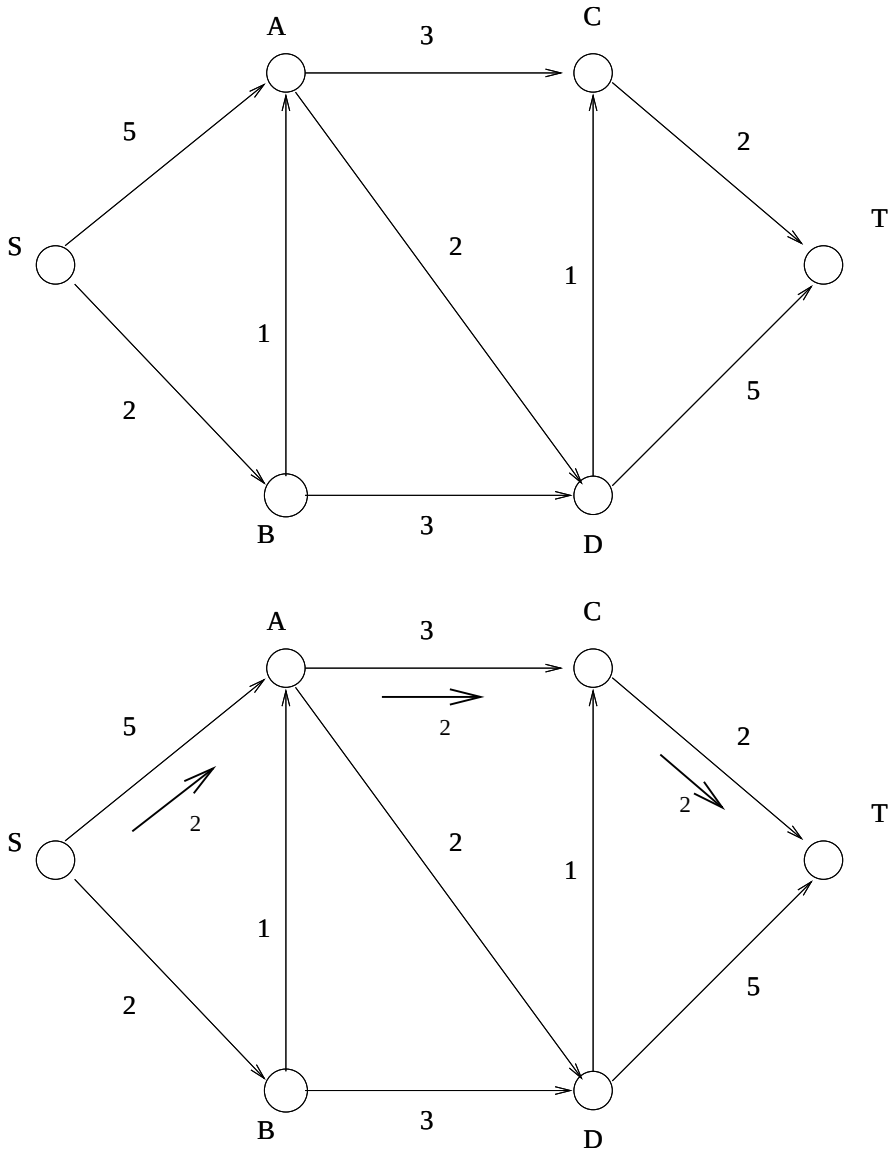


Figure 3: Max flow

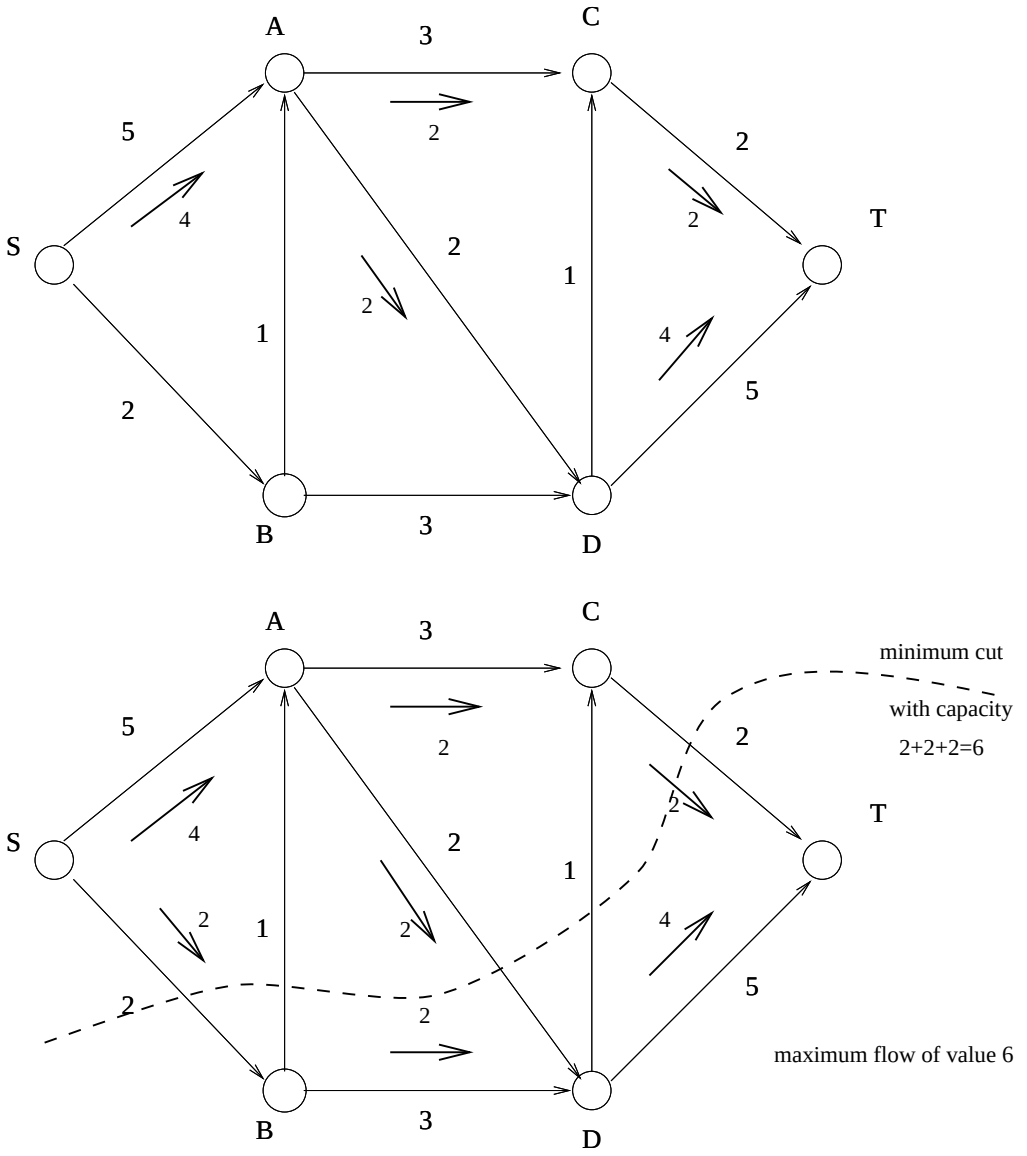


Figure 4: Max flow (continued)

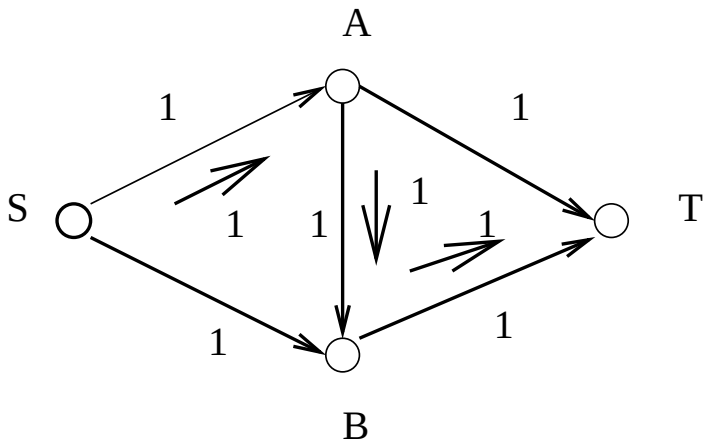


Figure 5: Flows may have to be canceled

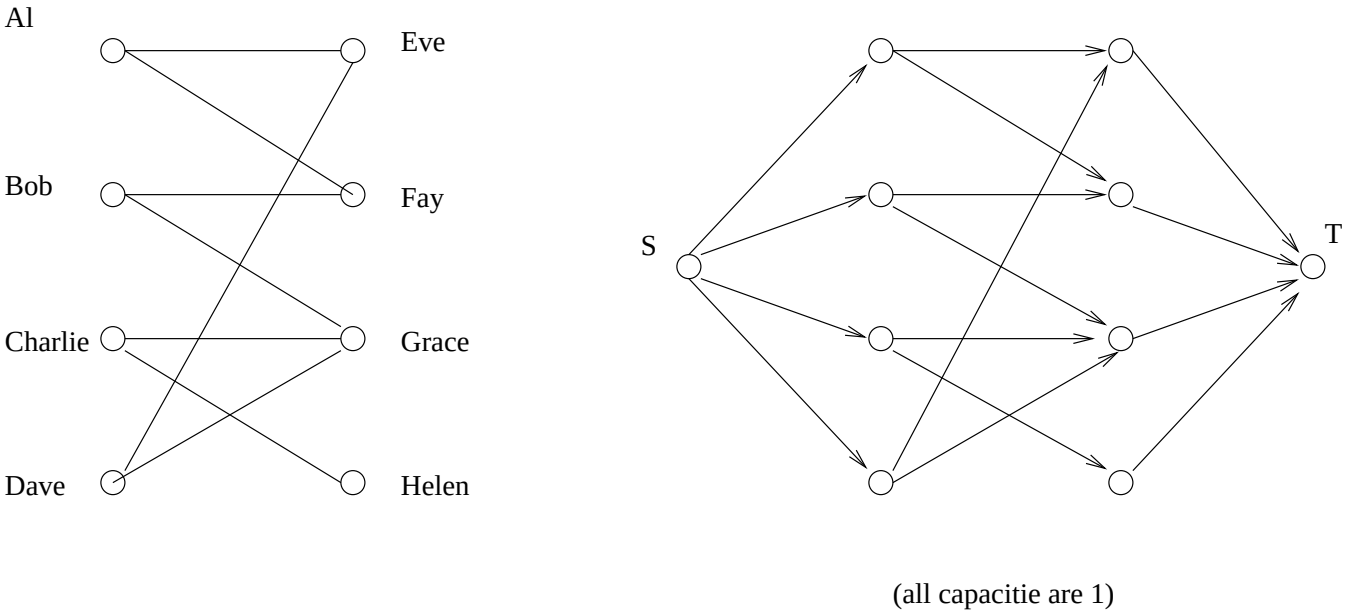


Figure 6: Reduction from matching to max-flow