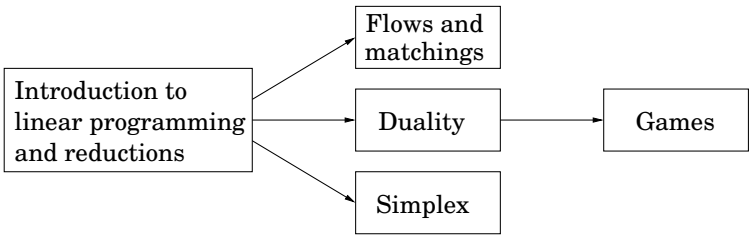
Chapter 7

Linear programming and reductions

Many of the problems for which we want algorithms are *optimization* tasks: the *shortest* path, the *cheapest* spanning tree, the *longest* increasing subsequence, and so on. In such cases, we seek a solution that (1) satisfies certain constraints (for instance, the path must use edges of the graph and lead from s to t, the tree must touch all nodes, the subsequence must be increasing); and (2) is the best possible, with respect to some well-defined criterion, among all solutions that satisfy these constraints.

Linear programming describes a broad class of optimization tasks in which both the constraints and the optimization criterion are *linear functions*. It turns out an enormous number of problems can be expressed in this way.

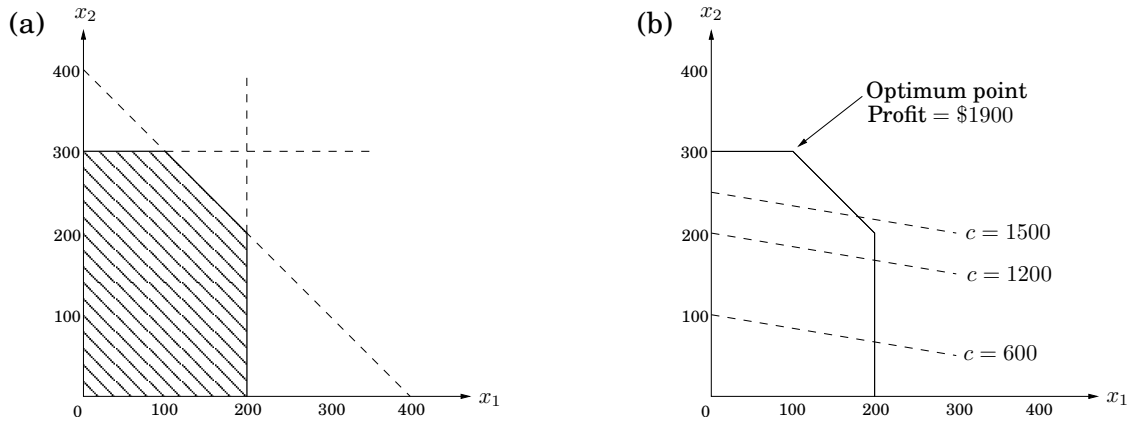
Given the vastness of its topic, this chapter is divided into several parts, which can be read separately subject to the following dependencies.



7.1 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.

Figure 7.1 (a) The feasible region for a linear program. (b) Contour lines of the objective function: $x_1 + 6x_2 = c$ for different values of the profit c.



7.1.1 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? Let's say it makes x_1 boxes of Pyramide per day, at a profit of $\$1$ each, and x_2 boxes of Nuit, at a more substantial profit of $\$6$ apiece; x_1 and x_2 are unknown values that we wish to determine. But this is not all; there are also some constraints on x_1 and x_2 that must be accommodated (besides the obvious one, $x_1, x_2 \geq 0$). First, the daily demand for these exclusive chocolates is limited to at most 200 boxes of Pyramide and 300 boxes of Nuit. Also, the current workforce can produce a total of at most 400 boxes of chocolate per day. What are the optimal levels of production?

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

$$\begin{array}{ll} \text{Objective function} & \max x_1 + 6x_2 \\ \text{Constraints} & x_1 \leq 200 \\ & x_2 \leq 300 \\ & x_1 + x_2 \leq 400 \\ & x_1, x_2 \geq 0 \end{array}$$

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, shown in Figure 7.1.

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 $-1/6$ and is shown in Figure 7.1 for selected values of c. 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, as shown in the figure. If the slope of the profit line were different, then its last contact with the polygon could be an entire edge rather than a single vertex. In this case, the optimum solution would not be unique, but there would certainly be an optimum vertex.

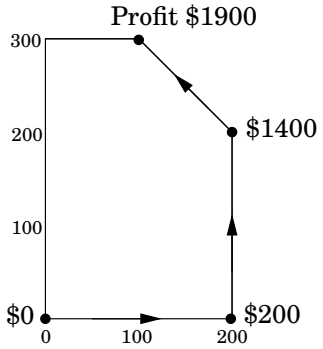
It is a general rule of linear programs that the optimum is achieved at a vertex of the feasible region. The only exceptions are cases in which there is no optimum; this can happen in two ways:

1. The linear program is *infeasible*; that is, the constraints are so tight that it is impossible to satisfy all of them. For instance,
$$x \leq 1, \quad x \geq 2.$$

2. The constraints are so loose that the feasible region is *unbounded*, and it is possible to achieve arbitrarily high objective values. For instance,
$$\max \; x_1 + x_2$$
$$x_1, x_2 \geq 0$$

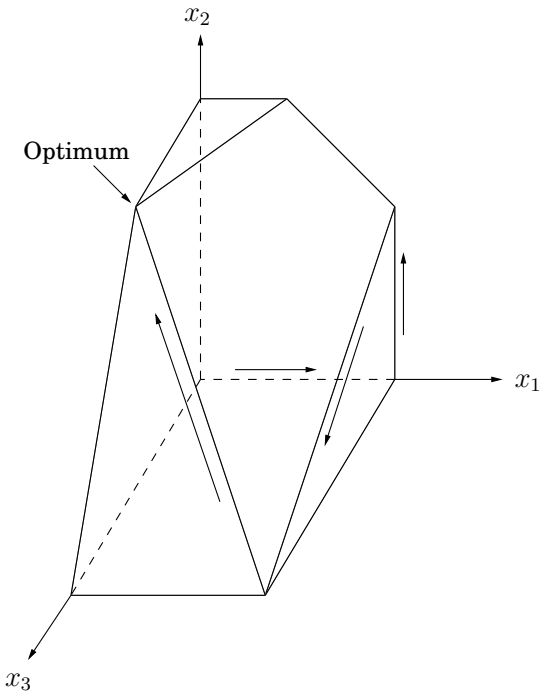
Solving linear programs

Linear programs (LPs) can be solved by the *simplex method*, devised by George Dantzig in 1947. We shall explain it in more detail in Section 7.6, but briefly, this algorithm starts at a vertex, in our case perhaps $(0, 0)$, 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. Here's a possible trajectory.



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.

Figure 7.2 The feasible polyhedron for a three-variable linear program.



More products

Encouraged by consumer demand, 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 $x_2 + 3x_3 \leq 600$. What are the best possible levels of production?

Here is the updated linear program.

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

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 (Figure 7.2). Looking at the figure, can you decipher which inequality corresponds to each face of the 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? As before, it would move from vertex to vertex, along edges of the polyhedron, increasing profit steadily. A possible trajectory is shown in Figure 7.2, corresponding to the following sequence of vertices and profits:

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

Finally, upon reaching a vertex with no better neighbor, it would stop and declare this to be the optimal point. Once again by basic geometry, if all the vertex's neighbors lie on one side of the profit-plane, then so must the entire polyhedron.

A magic trick called duality

Here is why you should believe that $(0, 300, 100)$, with a total profit of $\$3100$, is the optimum: Look back at the linear program. Add the second inequality to the third, and add to them the fourth multiplied by 4. The result is the inequality $x_1 + 6x_2 + 13x_3 \leq 3100$.

Do you see? This inequality says that no feasible solution (values x_1, x_2, x_3 satisfying the constraints) can possibly have a profit greater than 3100. So we must indeed have found the optimum! The only question is, where did we get these mysterious multipliers $(0, 1, 1, 4)$ for the four inequalities?

In Section 7.4 we'll see that it is always possible to come up with such multipliers by solving another LP! Except that (it gets even better) we do not even need to solve this other LP, because it is in fact so intimately connected to the original one—it is called the *dual*—that solving the original LP solves the dual as well! But we are getting far ahead of our story.

What if we add a fourth line of chocolates, or hundreds more of them? Then the problem becomes high-dimensional, and hard to visualize. Simplex continues to work in this general setting, although we can no longer rely upon simple geometric intuitions for its description and justification. We will study the full-fledged simplex algorithm in Section 7.6.

In the meantime, we can rest assured in the knowledge that there are many professional, industrial-strength packages that implement simplex and take care of all the tricky details like numeric precision. In a typical application, the main task is therefore to correctly express the problem as a linear program. The package then takes care of the rest.

With this in mind, let's look at a high-dimensional application.

7.1.2 Example: production planning

This time, our 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: $d_1, d_2, \ldots, d_{12}$. As feared, they are very uneven, ranging from 440 to 920.

Here's a quick snapshot of the company. We currently have 30 employees, each of whom makes 20 carpets per month and gets a monthly salary of $\$2{,}000$. 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.

This rather involved problem can be formulated and solved as a linear program!

A crucial first step is defining the variables.

$$\begin{aligned} w_i &= \text{number of workers during } i\text{th month; } w_0 = 30. \\ x_i &= \text{number of carpets made during } i\text{th month.} \\ o_i &= \text{number of carpets made by overtime in month } i. \\ h_i, f_i &= \text{number of workers hired and fired, respectively, at beginning of month } i. \\ s_i &= \text{number of carpets stored at end of month } i\text{; } s_0 = 0. \end{aligned}$$

All in all, there are 72 variables (74 if you count w_0 and s_0).

We now write the constraints. First, all variables must be nonnegative:

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

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

$$x_i = 20w_i + o_i$$

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

$$o_i \leq 6w_i.$$

Finally, what is the objective function? It is to minimize the total cost:

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

a linear function of the variables. Solving this linear program by simplex should take less than a second and will give us the optimum business strategy for our company.

Well, almost. 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*.

7.1.3 Example: optimum bandwidth allocation

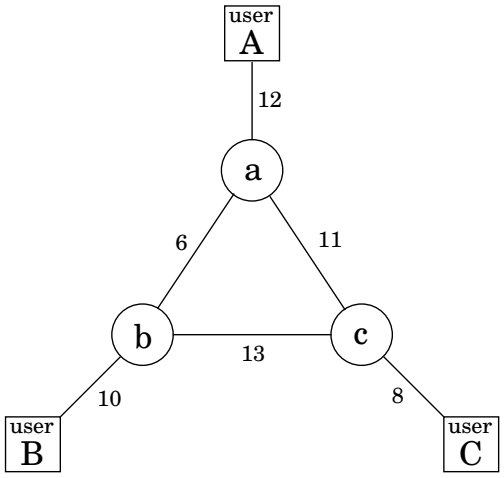
Next we turn to a miniaturized version of the kind of problem a network service provider might face.

Suppose we are managing a network whose lines have the bandwidths shown in Figure 7.3, and we need to establish three connections: between users A and B, between B and C, and between A and C. Each connection requires at least two units of bandwidth, but can be assigned more. Connection A–B pays \$3 per unit of bandwidth, and connections B–C and A–C pay \$2 and \$4, respectively.

Each connection can be routed in two ways, a long path and a short path, or by a combination: for instance, two units of bandwidth via the short route, one via the long route. How do we route these connections to maximize our network's revenue?

This is a linear program. We have variables for each connection and each path (long or short); for example, x_{AB} is the short-path bandwidth allocated to the connection between A and B, and x'_{AB} the long-path bandwidth for this same connection. We demand that no edge's bandwidth is exceeded and that each connection gets a bandwidth of at least 2 units.

Figure 7.3 A communications network between three users A, B, and C. Bandwidths are shown.



$$
\begin{aligned}
\max \quad & 3x_{AB} + 3x'_{AB} + 2x_{BC} + 2x'_{BC} + 4x_{AC} + 4x'_{AC} \\
& x_{AB} + x'_{AB} + x_{BC} + x'_{BC} \le 10 \qquad \text{[edge } (b, B)\text{]} \\
& x_{AB} + x'_{AB} + x_{AC} + x'_{AC} \le 12 \qquad \text{[edge } (a, A)\text{]} \\
& x_{BC} + x'_{BC} + x_{AC} + x'_{AC} \le 8 \qquad \text{[edge } (c, C)\text{]} \\
& x_{AB} + x'_{BC} + x'_{AC} \le 6 \qquad \text{[edge } (a, b)\text{]} \\
& x'_{AB} + x_{BC} + x'_{AC} \le 13 \qquad \text{[edge } (b, c)\text{]} \\
& x'_{AB} + x'_{BC} + x_{AC} \le 11 \qquad \text{[edge } (a, c)\text{]} \\
& x_{AB} + x'_{AB} \ge 2 \\
& x_{BC} + x'_{BC} \ge 2 \\
& x_{AC} + x'_{AC} \ge 2 \\
& x_{AB}, x'_{AB}, x_{BC}, x'_{BC}, x_{AC}, x'_{AC} \ge 0
\end{aligned}
$$

Even a tiny example like this one is hard to solve on one's own (try it!), and yet the optimal solution is obtained instantaneously via simplex:

$$x_{AB} = 0,\ x'_{AB} = 7,\ x_{BC} = x'_{BC} = 1.5,\ x_{AC} = 0.5,\ x'_{AC} = 4.5.$$

This solution is not integral, but in the present application we don't need it to be, and thus no rounding is required. Looking back at the original network, we see that every edge except a–c is used at full capacity.

One cautionary observation: our LP has one variable for every possible path between the users. In a larger network, there could easily be exponentially many such paths, and therefore

this particular way of translating the network problem into an LP will not scale well. We will see a cleverer and more scalable formulation in Section 7.2.

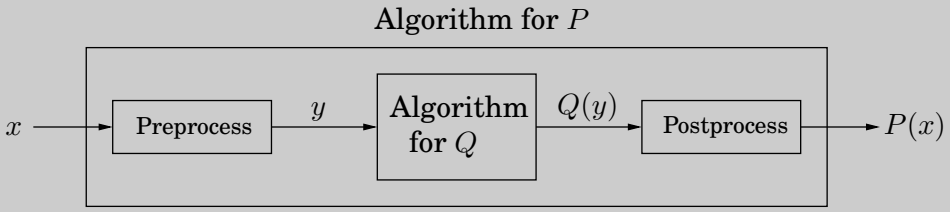
Here's a parting question for you to consider. Suppose we removed the constraint that each connection should receive at least two units of bandwidth. Would the optimum change?

Reductions

Sometimes a computational task is sufficiently general that any subroutine for it can also be used to solve a variety of other tasks, which at first glance might seem unrelated. For instance, we saw in Chapter 6 how an algorithm for finding the longest path in a dag can, surprisingly, also be used for finding longest increasing subsequences. We describe this phenomenon by saying that the longest increasing subsequence problem *reduces to* the longest path problem in a dag. In turn, the longest path in a dag reduces to the shortest path in a dag; here's how a subroutine for the latter can be used to solve the former:

function LONGEST PATH(G)
  negate all edge weights of G
  return SHORTEST PATH(G)

Let's step back and take a slightly more formal view of reductions. If any subroutine for task 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 $Q(y)$:



(Do you see that the reduction from $P =$ LONGEST PATH to $Q =$ SHORTEST PATH follows this schema?) If the pre- and postprocessing procedures are efficiently computable then this creates an efficient algorithm for P out of *any* efficient algorithm for Q!

Reductions enhance the power of algorithms: Once we have an algorithm for problem Q (which could be shortest path, for example) we can use it to solve other problems. In fact, most of the computational tasks we study in this book are considered core computer science problems precisely because they arise in so many different applications, which is another way of saying that many problems reduce to them. This is especially true of linear programming.

7.1.4 Variants of linear programming

As evidenced in our examples, 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. Here's how.

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

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

This s is called the *slack variable* for the inequality. As justification, observe that a vector $(x_1, \ldots, x_n)$ satisfies the original inequality constraint if and only if there is some $s \geq 0$ for which it satisfies the new equality constraint.

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 $x^+ - x^-$.

   This way, x can take on any real value by appropriately adjusting the new variables. More precisely, any feasible solution to the original LP involving x can be mapped to a feasible solution of the new LP involving x^+, x^-, and vice versa.

By applying these transformations 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*, in which the variables are all nonnegative, the constraints are all equations, and the objective function is to be minimized.

For example, our first linear program gets rewritten thus:

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

The original was also in a useful form: maximize an objective subject to certain inequalities. Any LP can likewise be recast in this way, using the reductions given earlier.

Matrix-vector notation

A linear function like $x_1 + 6x_2$ can be written as the dot product of two vectors

$$\mathbf{c} = \begin{pmatrix} 1 \\ 6 \end{pmatrix} \text{ and } \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix},$$

denoted $\mathbf{c} \cdot \mathbf{x}$ or $\mathbf{c}^T\mathbf{x}$. Similarly, linear constraints can be compiled into matrix-vector form:

$$\begin{array}{rcl} x_1 & \leq & 200 \\ x_2 & \leq & 300 \\ x_1 + x_2 & \leq & 400 \end{array} \quad \Longrightarrow \quad \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}}_{\mathbf{A}} \quad \underbrace{\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}}_{\mathbf{x}} \quad \leq \quad \underbrace{\begin{pmatrix} 200 \\ 300 \\ 400 \end{pmatrix}}_{\mathbf{b}}.$$

Here each row of matrix $\mathbf{A}$ corresponds to one constraint: its dot product with $\mathbf{x}$ is at most the value in the corresponding row of $\mathbf{b}$. In other words, if the rows of $\mathbf{A}$ are the vectors $\mathbf{a}_1, \ldots, \mathbf{a}_m$, then the statement $\mathbf{Ax} \leq \mathbf{b}$ is equivalent to

$$\mathbf{a}_i \cdot \mathbf{x} \leq b_i \text{ for all } i = 1, \ldots, m.$$

With these notational conveniences, a generic LP can be expressed simply as

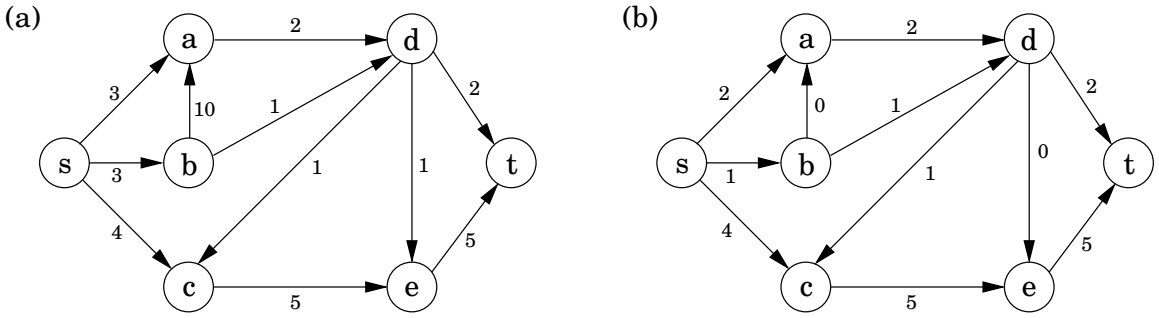
$$\begin{aligned} \max\ & \mathbf{c}^T\mathbf{x} \\ & \mathbf{Ax} \leq \mathbf{b} \\ & \mathbf{x} \geq 0. \end{aligned}$$

7.2 Flows in networks

7.2.1 Shipping oil

Figure 7.4(a) shows a directed graph representing a network of pipelines along which oil can be sent. The goal is to ship as much oil as possible from the *source* s to the *sink* t. Each pipeline has a maximum *capacity* it can handle, and there are no opportunities for storing oil

Figure 7.4 (a) A network with edge capacities. (b) A *flow* in the network.



en route. Figure 7.4(b) shows a possible *flow* from s to t, which ships 7 units in all. Is this the best that can be done?

7.2.2 Maximizing flow

The networks we are dealing with 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 flow entering u equals the amount leaving u:
$$\sum_{(w,u)\in E} f_{wu} = \sum_{(u,z)\in E} f_{uz}.$$
In other words, flow is *conserved*.

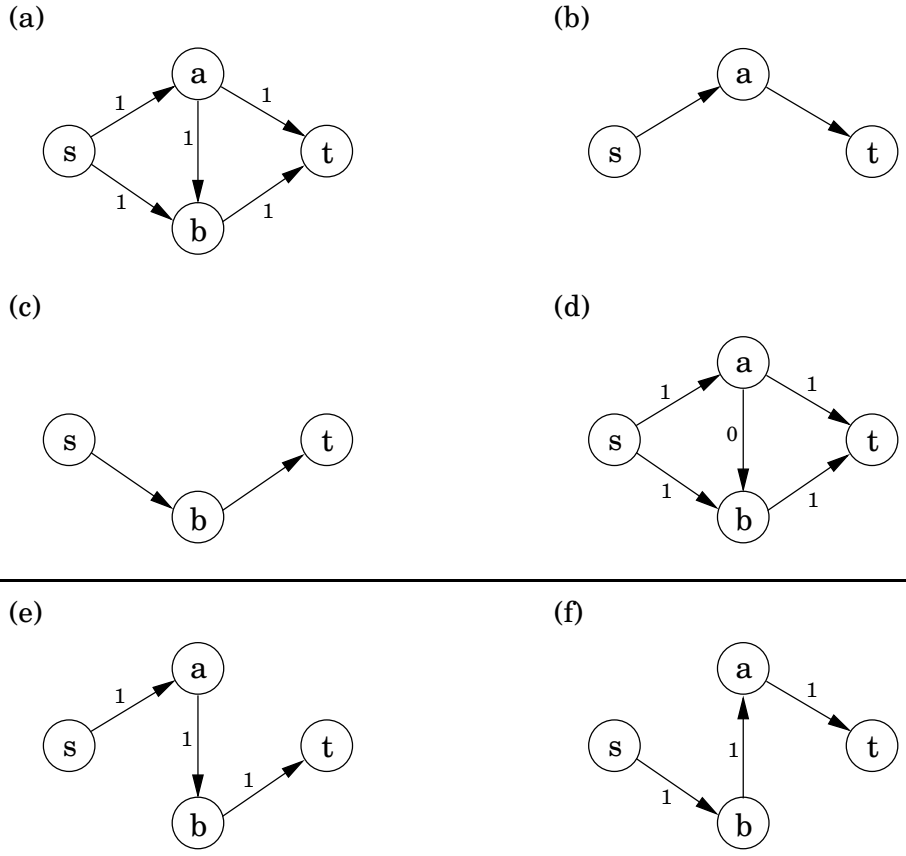
The *size* of a flow is the total quantity sent from s to t and, by the conservation principle, is equal to the quantity leaving s:

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

In short, our goal is to assign values to $\{f_e : e \in E\}$ 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*.

For example, for the network of Figure 7.4 the LP has 11 variables, one per edge. It seeks to maximize $f_{sa} + f_{sb} + f_{sc}$ subject to a total of 27 constraints: 11 for nonnegativity (such as $f_{sa} \geq 0$), 11 for capacity (such as $f_{sa} \leq 3$), and 5 for flow conservation (one for each node of the graph other than s and t, such as $f_{sc} + f_{dc} = f_{ce}$). Simplex would take no time at all to correctly solve the problem and to confirm that, in our example, a flow of 7 is in fact optimal.

Figure 7.5 An illustration of the max-flow algorithm. (a) A toy network. (b) The first path chosen. (c) The second path chosen. (d) The final flow. (e) We could have chosen this path first. (f) In which case, we would have to allow this second path.



7.2.3 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. Once we have studied it in more detail (Section 7.6), we will be in a position to understand exactly how it handles flow LPs, which is useful as a source of inspiration for designing *direct* max-flow algorithms.

It turns out that in fact 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.

Figure 7.5(a)–(d) shows a small example in which simplex halts after two iterations. The final flow has size 2, which is easily seen to be optimal.

There is just one complication. What if we had initially chosen a different path, the one in Figure 7.5(e)? This gives only one unit of flow and yet seems to block all other paths. Simplex gets around this problem by also allowing paths to *cancel existing flow*. In this particular case, it would subsequently choose the path of Figure 7.5(f). Edge (b, a) of this path isn't in the original network and has the effect of canceling flow previously assigned to edge (a, b).

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, upto f_{vu} additional units (canceling all or part of the existing flow on (v, u)). 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.

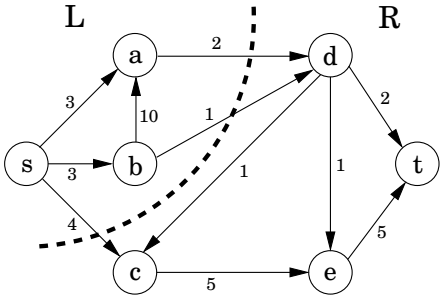
By simulating the behavior of simplex, we get a direct algorithm for solving max-flow. It proceeds in iterations, each time explicitly constructing G^f, finding a suitable $s - t$ path in G^f by using, say, a linear-time breadth-first search, and halting if there is no longer any such path along which flow can be increased.

Figure 7.6 illustrates the algorithm on our oil example.

7.2.4 A certificate of optimality

Now for 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!

Let's see an example of what this means. Partition the nodes of the oil network (Figure 7.4) into two groups, $L = \{s, a, b\}$ and $R = \{c, d, e, t\}$:



Any oil transmitted must pass from L to R. Therefore, no flow can possibly exceed the total capacity of the edges from L to R, which is 7. But this means that the flow we found earlier, of size 7, must be optimal!

More generally, an (s,t)-*cut* partitions the vertices into two disjoint groups L and R such that s is in L and t is 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:

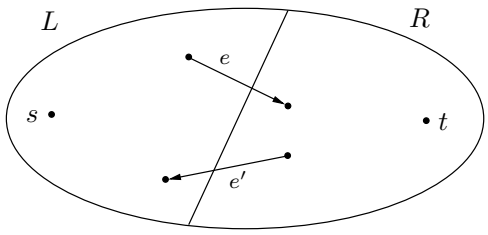
Pick any flow f and any (s,t)-cut (L,R). Then size(f) $\leq$ capacity(L,R).

Some cuts are large and give loose upper bounds—cut $(\{s,b,c\},\{a,d,e,t\})$ has a capacity of 19. But there is also a cut of capacity 7, which is effectively a *certificate of optimality* of the maximum flow. This isn't just a lucky property of our oil network; such a cut *always* exists.

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

Moreover, our algorithm automatically finds this cut as a by-product!

Let's see why this is true. 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 $R = V - L$ be the rest of the nodes. Then (L,R) is a cut in the graph G:



We claim that

$$\text{size}(f) \;=\; \text{capacity}(L,R).$$

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. (So, in the figure, $f_e = c_e$ and $f_{e'} = 0$.) Therefore the net flow across (L,R) is exactly the capacity of the cut.

7.2.5 Efficiency

Each iteration of our maximum-flow algorithm is efficient, requiring $O(|E|)$ time if a depth-first or breadth-first search 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 an inductive argument shows that 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 $C|E|$ (why?), it follows that the number of iterations is at most this much. But this is hardly a reassuring bound: what if C is in the millions?

We examine this issue further in Exercise 7.31. It turns out that it is indeed possible to construct bad examples in which the number of iterations is proportional to C, *if* $s-t$ paths are not carefully chosen. However, if paths are chosen in a sensible manner—in particular, by

using a breadth-first search, 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.

Figure 7.6 The max-flow algorithm applied to the network of Figure 7.4. At each iteration, the current flow is shown on the left and the residual network on the right. The paths chosen are shown in bold.

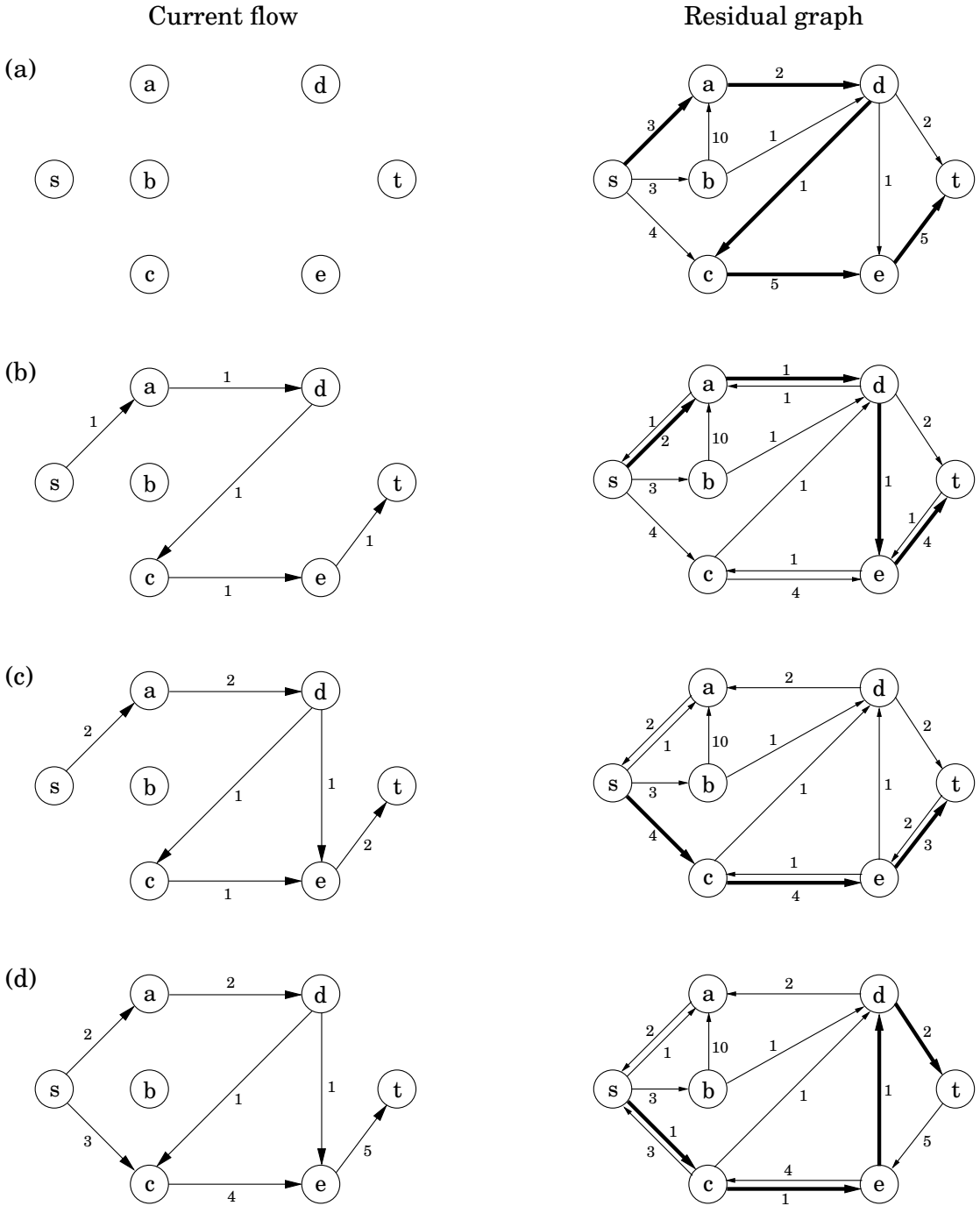


Figure 7.6 *Continued*

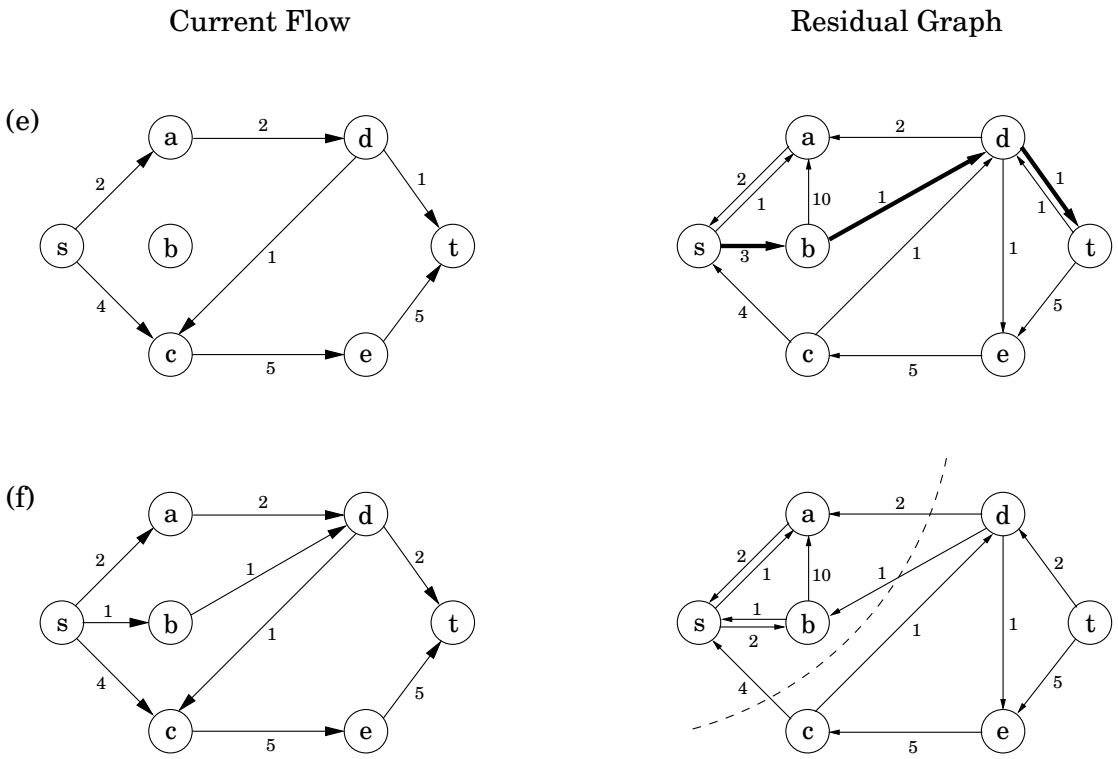
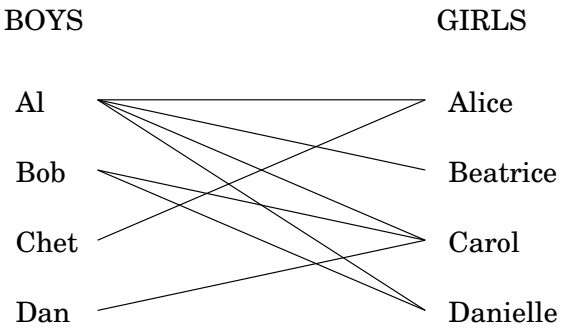


Figure 7.7 An edge between two people means they like each other. Is it possible to pair everyone up happily?



7.3 Bipartite matching

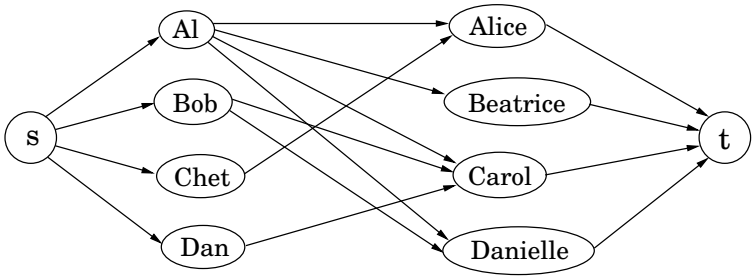
Figure 7.7 shows a graph with four nodes on the left representing boys and four nodes on the right representing girls.[1] There is an edge between a boy and girl if they like each other (for instance, Al likes all the girls). Is it possible to choose couples so that everyone has exactly one partner, and it is someone they like? In graph-theoretic jargon, is there a *perfect matching*?

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 (Figure 7.8). 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. Can you find such a flow in the example?

Actually, the situation is slightly more complicated than just stated: what is easy to see is that the optimum *integer-valued* flow corresponds to the optimum matching. We would be at a bit of a loss interpreting a flow that ships 0.7 units along the edge Al–Carol, for instance!

[1]This kind of graph, in which the nodes can be partitioned into two groups such that all edges are *between* the groups, is called *bipartite*.

Figure 7.8 A matchmaking network. Each edge has a capacity of one.



Fortunately, 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.

Hence integrality comes for free in the maximum-flow problem. Unfortunately, this is the exception rather than the rule: as we will see in Chapter 8, it is a very difficult problem to find the optimum solution (or for that matter, *any* solution) of a general linear program, if we also demand that the variables be integers.

7.4 Duality

We have seen that in networks, flows are smaller than cuts, but the maximum flow and minimum cut exactly coincide and each is therefore a certificate of the other's optimality. Remarkable as this phenomenon is, we now generalize it from maximum flow to *any* problem that can be solved by linear programming! It turns out that every linear maximization problem has a *dual* minimization problem, and they relate to each other in much the same way as flows and cuts.

To understand what duality is about, recall our introductory LP with the two types of chocolate:

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

Simplex declares the optimum solution to be $(x_1, x_2) = (100, 300)$, with objective value 1900. Can this answer be checked somehow? Let's see: suppose we take the first inequality and add it to six times the second inequality. We get

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

This is interesting, because it tells us that it is impossible to achieve a profit of more than 2000. Can we add together some other combination of the LP constraints and bring this upper bound even closer to 1900? After a little experimentation, we find that multiplying the three inequalities by 0, 5, and 1, respectively, and adding them up yields

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

So 1900 must indeed be the best possible value! The multipliers $(0, 5, 1)$ magically constitute a *certificate of optimality*! It is remarkable that such a certificate exists for this LP—and even if we knew there were one, how would we systematically go about finding it?