ICS 311 #21: Linear Programming

Outline

1. Introduction to Linear Programming
2. Formulating Problems as Linear Programs
3. Foundations in Gaussian Elimination
4. The Simplex Method

Objectives

- Be aware of the range of problems to which linear programming can be applied.
- Understand the Simplex algorithm just enough to understand the format of linear equations used and what is done with them.
- Be able to write a simple linear program for a problem.

Readings

If you have a background in Gaussian Elimination and read and understand Sections 29.0-29.3 of CLRS, up through the description of Simplex (you need not read the proofs that follow), these objectives will be met. (The material of CLRS Sections 29.4-29.5 is excellent, but we don't need to see all the proofs concerning Simplex to use it.)

If you don't have a background in Gaussian Elimination, then reading and understanding Section 28.1 of CLRS would provide it. However, Section 28.1 provides more detail than is needed to get the gist of Gaussian Elimination and the Simplex. I found Sedgewick's (1984), Chapter 5 presentation of Gaussian Elimination to be clear and sufficient. I also found his presentation of Linear Programming in Chapter 38 useful for its clear narrative around an example.

For a full study of linear programming I recommend this reading sequence:

- **Chapter 5 of Sedgewick (1984) on Gaussian Elimination**
- **Chapter 38 of Sedgewick (1984) on Linear Programming**
- **Sections 29.0 through the first half of 29.3 of CLRS**

If you don't have time for the full reading:

- **Read the following web notes** (which summarize the main points from Sedgewick and some material from CLRS 29.0-29.3).
- Then **read 29.0, 29.1 and 29.2 of CLRS before class** (quiz questions and class problems are drawn from those sections).

Screencasts

No Screencasts were made for this topic.

Introduction to Linear Programming

The following brief conceptual overview of Linear Programming and its roots in Gaussian Elimination is based largely on Chapters 5 and 38 of: Robert Sedgewick (1983). Algorithms. Reading, MA: Addison-Wesley. First Edition (available on Internet), with some comments from CLRS Chapter 29.

Mathematical programming is the process of modeling a problem as a set of mathematical equations. (The "programming" is in mathematics, not computer code.)

Linear programming is mathematical programming where the equations are *linear equations* in a set of variables that model a problem, and include:

- a set of **constraints** on the values of the variables (each constraint being expressed as a linear equation), and
- an **objective function** or linear function of these variables that is to be maximized subject to these constraints.

A large and diverse set of problems can be expressed as linear programs and solved. Examples include:

- ***Scheduling tasks,*** such as in business, construction or manufacturing, for example, scheduling flight crews for an airline.
- ***Flows in a network,*** including flows of multiple types of substances or commodities subject to various constraints (example to be given).
- ***Maximizing an outcome*** given a set of constrained resources, such as deciding where to drill for oil for maximum expected payoff.

Simplex Algorithm

- A well established algorithm (actually, family of algorithms) for solving linear programming problems.
- Available in many computer packages.
- Usually not the most efficient way to solve a problem (many of the algorithms we have studied are more efficient for their specialized problem), but is often easy to "program".
- Well studied, but analyzing its asymptotic complexity is still an active area of research, over 50 years after its invention!
- Examples have been given requiring exponential time, but Simplex has been repeatedly shown to have good performance in practice on real problems.

Examples

We begin with examples of problems for which we already have more efficient algorithms. The point of revisiting them here with less efficient linear programming solutions is to show you how linear programming works in terms of familiar problems; and also to reinforce the recurring theme that problems can be solved with different algorithms if you change problem representation.

Linear Program for Single-Pair Shortest Paths

The Bellman-Ford algorithm for single-source shortest paths uses the `Relax` procedure to find a distance $v.d$, where for every edge $(u, v) \in E$, $v.d \leq u.d + w(u, v)$ (since `Relax` changes $v.d$ precisely when this is not true). Also, $s.d$ for the source vertex s is always 0.

We can translate these observations directly into a linear program for the **single-pair shortest-path** problem from s to t. We will use notation d_v instead of $v.d$ to be consistent with typical linear programming notation:

Maximize: d_t
Subject to:

$d_v \leq d_u + w(u, v), \ \forall (u, v) \in E$
$s.d = 0.$

The expression $d_v \leq d_u + w(u, v)$ is a constraint based on what `Relax` guarantees, and $s.d = 0$ by definition (assuming no negative weight cycles).

But why are we <u>maximizing</u> d_t when we seek <u>shortest</u> paths?

- If we minimized d_t, then there would be a trivial solution where $d_v = 0, \forall v \in V$.
- The minimization that finds shortest paths is actually implicit in the first constraint.
  Each d_v will be given the maximum value that is yet ≤ the *smallest* $d_u + w(u, v)$.

(Compare to the fact that we needed to find <u>longest</u> paths when determining the shortest time in which a set of jobs could finish in the parallel scheduling problem given in class.)

The extension to **single-source all-destinations** is straightforward: maximize the <u>sum</u> of the destination distances.

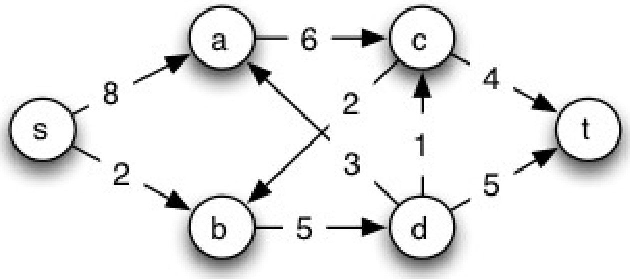
The custom algorithms for single-source and indeed all-pairs shortest paths will be more efficient than solving these problems with linear programming, but it is very easy to write these linear programs, and this example (and the next) illustrates how linear programming works in terms of a familiar example.

Linear Program for Max Flow

Next we show how to model a max-flow problem with linear programming. Instead of writing $f(s,a)$ to indicate the flow over edge (s,a) (for example), we follow the conventions of the linear programming literature and write f_{sa}. (Sedgewick uses X_{AB}.) CLRS present a more general template for any flow network, whereas here we look at a specific example:

Maximize: $f_{sa} + f_{sb}$
Subject to:

$f_{sa} \leq 8$ $\quad f_{sb} \leq 2$
$f_{ac} \leq 6$ $\quad f_{da} \leq 3$
$f_{bd} \leq 5$ $\quad f_{cb} \leq 2$
$f_{ct} \leq 4$ $\quad f_{dt} \leq 5$
$f_{sa} + f_{da} = f_{ac}$ $\quad f_{sb} + f_{cb} = f_{bd}$
$f_{dt} + f_{da} + f_{dc} = f_{bd}$ $\quad f_{cb} + f_{ct} =$
f_{ac}
$f_{sa}, \ f_{sb}, \ f_{ac}, \ f_{cb}, \ f_{ct}, \ f_{bd}, \ f_{da}, \ f_{dt} \ \geq \ 0.$



The expression to be maximized,

$f_{sa} + f_{sb}$

is the flow over the edges coming out of the source, and hence will be the flow of the entire network. If the linear program maximizes this, then we have found the max flow. (If there are edges incoming to s we can subtract these in the expression to be maximized.)

These inequalities capture edge capacities:

$f_{sa} \le 8;\ \ f_{sb} \le 2;\ \ f_{ac} \le 6;\ \ f_{da} \le 3;\ \ f_{bd} \le 5;\ \ f_{cb} \le 2;\ \ f_{ct} \le 4;\ \ f_{dt} \le 5.$

These equalities capture the conservation of flow at vertices (I've reordered the last two to show that it's flow in = flow out):

$f_{sa} + f_{da} = f_{ac}$ (flow through a)
$f_{sb} + f_{cb} = f_{bd}$ (flow through b)
$f_{ac} = f_{cb} + f_{ct}$ (flow through c)
$f_{bd} = f_{dt} + f_{da}$ (flow through d)

The final eight inequalities (written in one line for brevity) express the constraint that all flows must be positive:

$f_{sa},\ f_{sb},\ f_{ac},\ f_{cb},\ f_{ct},\ f_{bd},\ f_{da},\ f_{dt}\ \ge\ 0.$

The Simplex algorithm (discussed later and in the readings), when given a suitable form of these equations (see section 29.1 CLRS), will return an assignment of values to variables $f_{sa}, \ldots f_{dt}$ that maximizes the expression $f_{sa} + f_{sb}$ and hence flow.

The Edmonds-Karp flow algorithm is more efficient than the Simplex algorithm for solving this version of the max-flow problem. However, Edmonds-Karp is difficult to modify for problem variations such as multiple commodities or dealing with cost-benefit tradeoffs. These additional constraints are easy to add to a linear program.

In general, if a problem can be expressed as a linear program it may be quicker from a development standpoint to do that rather than to invent a custom algorithm for it. Linear programming covers a large variety of problems.

The point here is to introduce linear programming with a familiar example, and to illustrate its generality, but this also provides another example of "problem reduction", a concept that will be at the core of the final topic of this course on Complexity Theory & NP-Completeness.

Gaussian Elimination

The Simplex algorithm works in a manner similar to (derived from) Gaussian Elimination for solving a set of linear equations.

Invented by Chinese mathematicians a few thousand years ago, and in Europe by Newton and revised by Gauss, Gaussian elimination is a two part method for solving a system of linear equations.

As a simple example, suppose we have the following system:

$$\begin{aligned} &\boldsymbol{x + 3y - 4z = 8} \\ &\boldsymbol{x + y - 2z = 2} \\ &\boldsymbol{-x - 2y + 5z = -1} \end{aligned}$$

The goal is to find values of *x*, *y*, and *z* that satisfy these equations. (Recall that there may be zero, one, or an infinite number of solutions, and you need as many equations as variables to have a unique solution.)

If we think of the variables as subscripted as shown on the left, then we can rewrite the system of equations as a matrix equation using the subscripts as column indices as shown on the right:

$$\begin{aligned} x_1 + 3x_2 - 4x_3 &= 8 \\ x_1 + x_2 - 2x_3 &= 2 \\ -x_1 - 2x_2 + 5x_3 &= -1 \end{aligned}$$

$$\begin{pmatrix} 1 & 3 & -4 \\ 1 & 1 & -2 \\ -1 & -2 & 5 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 2 \\ -1 \end{pmatrix}.$$

The following operations can be done on systems of linear equations such as the above. (Later, in the section on linear programing, we'll drop the parentheses and put everything in one matrix. Then, the operations below will be operations on rows and columns of the matrix.)

- ***Interchanging equations:*** Since the order in which we write equations does not matter, we can reorder the rows.
- ***Renaming variables:*** Swapping entire columns with each other. If we swap columns *i* and *j*, what was formerly x_i becomes x_j and vice-versa. This is OK because variable names are arbitrary.
- ***Multiplying equations by a constant:*** Accomplished by multiplying all numbers in a row by that constant.
- ***Adding two equations and replacing one by the sum:*** Since the two sides of an equation are equal, we can add them to the two sides of another equation without affecting equality.

The Strategy

Gaussian elimination is a systematic way of applying these operations to make the value of one variable obvious (*forward elimination*), and then substituting this value back into the other equations to expose their values (*backward substitution*).

Forward Elimination (Triangulation)

Forward elimination turns the matrix into a triangular matrix, where there is only one variable in the last equation, only that variable plus one more in the next equation up, etc.

For example, replace the second equation by the difference between the first two:

Before: $$\begin{pmatrix} 1 & 3 & -4 \\ 1 & 1 & -2 \\ -1 & -2 & 5 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 2 \\ -1 \end{pmatrix}.$$

After: $$\begin{pmatrix} 1 & 3 & -4 \\ 0 & 2 & -2 \\ -1 & -2 & 5 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 6 \\ -1 \end{pmatrix}.$$

One term has gone to 0: this means x_1 has been eliminated from the second equation. Let's eliminate x_1 from the third equation by replacing the third by the sum of the first and the third:

$$\begin{pmatrix} 1 & 3 & -4 \\ 0 & 2 & -2 \\ 0 & 1 & 1 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 6 \\ 7 \end{pmatrix}.$$

Now if we replace the third equation by the difference between the second and twice the third, we can eliminate x_2 from the third row, leaving a *triangular* matrix. Writing the result as equations:

$x_1 + 3x_2 - 4x_3 = 8$

$$\begin{aligned} 2x_2 - 2x_3 &= 6 \\ -4x_3 &= -8 \end{aligned}$$

At the completion of the forward elimination phase, the equations are easy to solve.

Backward Substitution Phase

It is easy to determine from the third equation that $x_3 = 2$. Substituting that into the second equation, we can derive x_2:

$$\begin{aligned} 2x_2 - 4 &= 6 \\ x_2 &= 5 \end{aligned}$$

Substituting this and $x_3 = 2$ into the equation above (rewritten below) solves for x_1:

$$\begin{aligned} x_1 + 3x_2 - 4x_3 &= 8 \\ x_1 + 15 - 8 &= 8 \\ x_1 &= 1 \end{aligned}$$

The Algorithm

In general we can solve systems of linear equations as written on the left by converting them into matrices as written on the right:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2N}x_N &= b_2, \\ \\ a_{N1}x_1 + a_{N2}x_2 + \cdots + a_{NN}x_N &= b_N. \end{aligned} \qquad \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \vdots & & & \\ a_{N1} & a_{N2} & \dots & a_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix}$$

or $\boldsymbol{Ax} = \boldsymbol{b}$ in matrix equation form. It is convenient to represent this entire system in one N x (N+1) matrix consisting of $\boldsymbol{A}$ and the last column for $\boldsymbol{b}$:

$$\begin{array}{ccccc} a_{11} & a_{12} & \dots & a_{1N} & b_1 \\ a_{21} & a_{22} & \dots & a_{2N} & b_2 \\ & & \dots & & \\ a_{N1} & a_{N2} & \dots & a_{NN} & b_N \end{array}$$

Basic Algorithm for Gaussian Elimination

We can eliminate

- the *first variable* from *all but the first equation* by adding an appropriate multiple of the first equation to each of the second through Nth equations (the multiple will be different for each equation);
- the *second variable* from *all but the first two equations* by adding an appropriate multiple of the second equation to the third through Nth equations;
- and so on ...

In general, the algorithm for forward elimination eliminates the ith variable in the jth equation by multiplying the ith equation by a_{ji} / a_{ii} and subtracting it from the jth equation, for $i+1 \leq j \leq N$.

We use a_{ji} / a_{ii} because $(a_{ji} / a_{ii}) * a_{ii} = a_{ji}$, so when we subtract row i from row j we get $a_{ji} - a_{ji} = 0$ in cell j,i.

The essential idea can be expressed in this pseudocode fragment (translated from Sedgewick's Pascal):

```
for i = 1 to N do
    for j = i + 1 to N do
        for k = N + 1 downto i do
            a[j,k] = a[j,k] - a[i,k] * a[j,i] / a[i,i]
```

There are three nested loops. *Trivial Question: How do the loops grow with N? What's the complexity?*

Elimination Elaborated

This code is too simple: In an actual implementation, various issues must be dealt with, including:

- If $a_{ii} = 0$, we cannot divide by 0. Need to swap rows to make a_{ii} non-zero in the outer loop. If this is not possible, there is no unique solution.
- If a_{ii} is very small, the scaling factor a_{ji} / a_{ii} could get very large, leading to rounding error in floating point representations used in computers. This is solved by always choosing the row in i+1 to N with the largest absolute value.

The process of elimination is also called **pivoting**, a concept that shows up in the application to linear programming.

Sedgewick presents an improved version as a Pascal procedure. If you want to understand the algorithm at this level of detail you should read CLRS 28.1.

Linear Programming

Linear programs are systems of linear equations, but with the additional twists that

- The constraint equations may include inequalities.
- There is also a linear expression, the objective function, to be maximized.

These two are related:

- The constraints being inequalities means there is often no unique solution to the system of constraints.
- Maximizing the objective function helps us choose from among the infinite possible solutions.

In fact, these points capture our motivations, in many cases, for using linear programming for real-world problems! There are many ways to act (i.e., many solutions), but we want to know which one is the best (i.e., maximized objective function). The constraints model a set of possible solutions, and the objective function helps us pick one that maximizes something we care about. Linear programming is a *general* way to approach any such situation that can be modeled with linear equations.

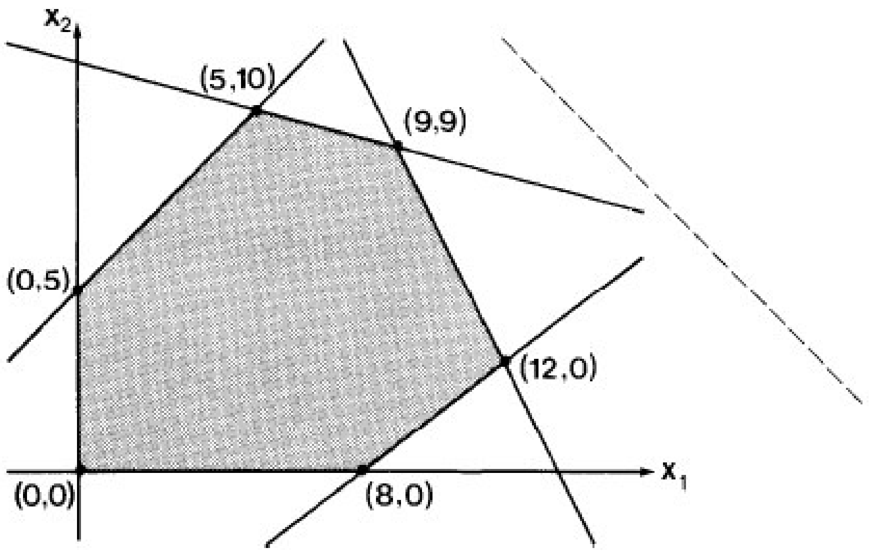
Example

For example, a simple linear program in two variables might look like this:

$$-x_1 + x_2 \leq 5$$
$$x_1 + 4x_2 \leq 45$$

$$2x_1 + x_2 \le 27$$
$$3x_1 - 4x_2 \le 24$$

$$x_1, x_2 \ge 0$$



Geometric Interpretation

We can graph this example as shown:

Each inequality divides the plane into one half in which a solution cannot lie and one in which it can.

For example, $x_1 \ge 0$ excludes solutions to the left of the x_2 axis, and $-x_1 + x_2 \le 5$ means solutions must lie below and to the right of the line $-x_1 + x_2 = 5$, shown between (0,5) and (5,10).

Simplex

Solutions must lie within the feasible region defined by the intersection of the regions defined by the constraint equations. That region is called the **simplex**. In the above example, each equation defines a half plane and the simplex, shaded grey in the above figure, is the intersection of these half planes. In systems with more variables the regions are in higher dimensions that are harder to visualize.

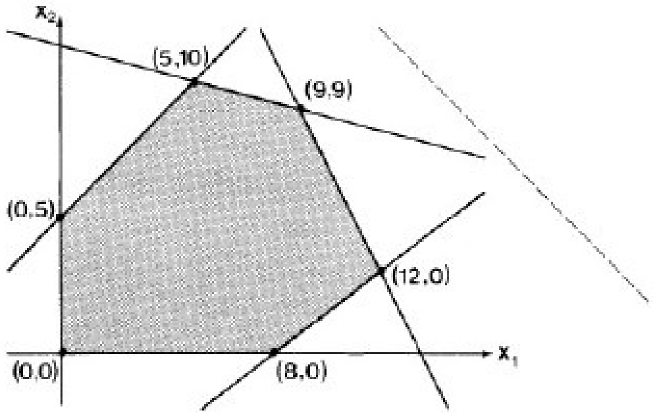
The simplex is a **convex region:** for any two points in the region, all points on a line segment between them are also in the region. Convexness can be used to show an important fact:

Fundamental Theorem

The objective function is always maximized at one of the vertices of the simplex.

Think of the objective function (here, $x_1 + x_2$, the dotted line) as a line of known slope but unknown position. Imagine the line being slid towards the simplex from infinity. If there is a solution, it will first touch the simplex at one of the vertices (one solution) or coincide with an edge (many solutions) that includes a vertex.

Where would this line touch the simplex?



Simplex Algorithm

<u>The algorithm does not actually slide a line</u>. Rather, this geometric interpretation tells us that the algorithm need only need search for a solution at the vertices of the convex simplex.

<u>The simplex method systematically searches the vertices</u>, moving to new vertices on which the objective function is no less, and is usually greater than the value for the previous vertex. An example will be given below. See CLRS 29.3 for details of the algorithm.

Other Issues Exposed by the Geometric Interpretation

- **Linearity is important**: if either the objective function or the simplex were curved, it would be much harder to tell where they overlap optimally.
- If the intersection of the half-planes is empty, the linear program is **infeasible**.
- A constraint is **redundant** if the simplex defined by the other constraints lies entirely within its half-plane. This is not a problem but the code must handle these situations.
- The simplex may be **unbounded**. As a result, the solution may be ill-defined, or even if it is well defined an algorithm may have difficulty with the unbounded portion.

Multiple Dimensions

The geometric interpretation extends to more variables = dimensions.

In three dimensions,

- The simplex is a convex 3-dimenstional solid defined by the intersection of half-spaces defined by constraints expressing planes rather than lines.
- The objective function is a plane and Simplex finds the maximal value for which it intersects a vertex of the 3-dimensional solid.

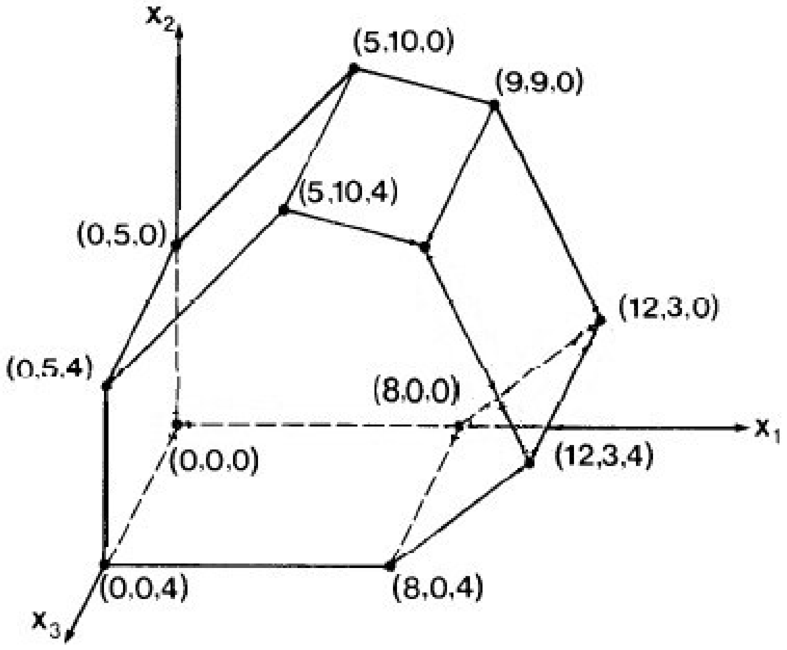
In *n* dimensions,

- (n-1)-dimensional hyperplanes are intersected to define an n-dimensional simplex.
- The objective function is an n-1 dimensional hyper-plane, and again Simplex finds the maximal value at which it intersects a vertex of the n-dimensional region.

The anomalous situations get much harder to detect in advance as dimensions increase, so it is important to handle them well in the code.

As an example, add the inequalities $x_3 \leq 4$ and $x_3 \geq 0$ to our previous example. The simplex becomes a 3-D solid:

$$\begin{aligned} -x_1 + x_2 &\leq 5 \\ x_1 + 4x_2 &\leq 45 \\ 2x_1 + x_2 &\leq 27 \\ 3x_1 - 4x_2 &\leq 24 \\ x_3 &\leq 4 \\ \\ x_1, x_2, x_3 &\geq 0 \end{aligned}$$



If the objective function is defined to be $x_1 + x_2 + x_3$, this is a plane perpendicular to the line $x_1 = x_2 = x_3$. Imagine this plane being brought from infinity to the origin: *where would it hit the simplex?*

Again, the algorithm we discuss below does not actually move planes from infinity; this is just a way of visualizing the fact that an optimal solution must lie on *some* vertex of the n-dimensional simplex, so we need only search these vertices.

The Simplex Method

Now we see how pivoting from Gaussian elimination is used. Pivoting is analogous to moving between the vertices of the simplex, starting at the origin. First, we need to prepare the data ...

Standard Form

(Note: Sedgewick does not distinguish between standard and slack forms; this discussion is based on CLRS section 29.1, to which the reader is referred for details.)

When equations are written to model a problem in a natural way, they may have various features that are not suitable for input to the Simplex Method. We begin by conversion into **standard form**:

> Given n **real numbers** $c_1, c_2, \dots c_n$ *(coefficients on objective function),*
> m **real numbers** $b_1, b_2, \dots b_m$ *(constants on right hand side of equations),*
> and mn **real numbers** a_{ij} for $i = 1, 2 \dots m$ and $j = 1, 2, \dots n$ *(coefficients on variables in equations),*
> **find real numbers** $x_1, x_2, \dots x_n$ *(the variables)*
>
> **that maximize:** $\Sigma_{j=1,n}\, c_j\, x_j$ *(the objective function)*
>
> **subject to:** $\Sigma_{j=1,n}\, a_{ij}\, x_j \le b_j$ **for** $i = 1, 2, \dots m$ *(regular constraints)*
>
> **and** $x_j \ge 0$, **for** $j = 1, 2, \dots n$ *(nonnegativity constraints)*

The following conversions may be needed to convert a linear program into standard form (see CLRS for details and justification):

1. If the objective function is to be minimized rather than maximized, negate the objective function (i.e., negate its coefficients).
2. Replace each variable x that does not have a nonnegativity constraint with $x'-x''$, and introduce the constraints $x' \ge 0$ and $x'' \ge 0$.
3. Convert equality constraints of form $f(x_1, x_2, \dots x_n) = b$ into two inequality constraints $f(x_1, x_2, \dots x_n) \le b$ and $f(x_1, x_2, \dots x_n) \ge b$.
4. Convert $\ge$ constraints (except the nonnegativity constraints) into $\le$ constraints by multiplying the constraints by -1.

Our example above is already in standard form, except that some of the coefficients a_{ij} are equal to 1 and are not written out, and we have not written terms with 0 coefficents. Making all a_{ij} explicit, we would write:

$$
\begin{aligned}
-1x_1 + 1x_2 + 0x_3 &\le 5\\
1x_1 + 4x_2 + 0x_3 &\le 45\\
2x_1 + 1x_2 + 0x_3 &\le 27\\
3x_1 - 4x_2 + 0x_3 &\le 24\\
0x_1 + 0x_2 + 1x_3 &\le 4\\
\\
x_1, x_2, x_3 &\ge 0
\end{aligned}
$$

Slack Form

The Simplex Method is based on methods (akin to Gaussian elimination) for solving systems of linear equations that require that we work with equalities rather than inequalities (except for the constraints that the variables are non-negative).

We can convert standard form into slack form by introducing **slack variables**, one for each inequality, that "take up the slack" allowed by the inequality. (These will be allowed to range as needed to do so.)

For example, instead of $x_1 + 4x_2 \le 45$, we can write $x_1 + 4x_2 + y = 45$, where y can range over the values needed to "take up the slack" between inequality and equality.

Applying this idea to the 3-D example above, and using a different y_i for each equation, we can model that example with:

> Maximize $x_1 + x_2 + x_3$ subject to the constraints:
>
> $$\begin{aligned} -1x_1 + 1x_2 + 0x_3 + y_1 &= 5 \\ 1x_1 + 4x_2 + 0x_3 + y_2 &= 45 \\ 2x_1 + 1x_2 + 0x_3 + y_3 &= 27 \\ 3x_1 - 4x_2 + 0x_3 + y_4 &= 24 \\ 0x_1 + 0x_2 + 1x_3 + y_5 &= 4 \end{aligned}$$
>
> $$x_1, x_2, x_3, y_1, y_2, y_3, y_4, y_5 \ge 0$$

There are m equations in n variables, including up to m slack variables (one for each inequality). (Note: in using n and m, I am following CLRS. Sedgewick uses M for number of variables and N for number of equations.)

- We assume that $n > m$ (more variables than equations), so there are many solutions possible. (In our example above, $n = 8$ and $m = 5$.)
- We assume that the origin ((0, 0, 0) in this example) is a point on the simplex, so we can use it as a starting point for the search for the best solution, which must lie on some vertex. (The assumption that the origin is a solution can be eliminated if needed.)

We can now write the slack-form system of equations (e.g., above) as a matrix (e.g., shown below), where the 0th row contains the negated coefficients of the objective function. Sedgewick describes how this negation directs the procedure to select the correct rows and columns for pivoting), and the (n+1)th column has the numbers on the right hand side of the equation.

```
-1.00  -1.00  -1.00   0.00   0.00   0.00   0.00   0.00   0.00
-1.00   1.00   0.00   1.00   0.00   0.00   0.00   0.00   5.00
 1.00   4.00   0.00   0.00   1.00   0.00   0.00   0.00  45.00
 2.00   1.00   0.00   0.00   0.00   1.00   0.00   0.00  27.00
 3.00  -4.00   0.00   0.00   0.00   0.00   1.00   0.00  24.00
 0.00   0.00   1.00   0.00   0.00   0.00   0.00   1.00   4.00
```

We want to perform pivot operations, using the same row and column manipulations as for Gaussian elimination.

- Instead of trying to make a triangular matrix we are trying to get each column corresponding to the non-slack variables x_1, x_2, and x_3 to have exactly one "1" in it and all the rest "0"s.
- This is because the variables with one "1" in it and all the rest "0"s are the **basis variables**: their values give the solution if we set all other variables to 0.
- Then we will be able to read off the values of the variables in the (n+1)th or rightmost column. The value of variable x_i will be found in row i column n+1, or at $a_{i,\,n+1}$.
- We don't care what the values of the slack variables y_i are (they just move the solution around in the

feasible inequality areas).

As we proceed, the upper right cell will have the current value of the objective function. We always want to increase this. The question is what strategy to take.

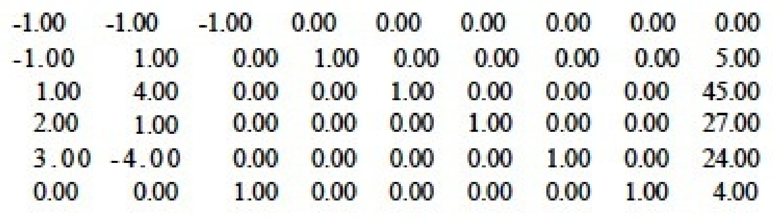
The most popular strategy is **greatest increment**:

- Choose the *column q* with the smallest value in row 0 (the largest absolute value). The objective function will increase if we use any column with a negative entry in row 0.
- Choose the *row p* from among those with positive values in the chosen column that has the smallest value when divided into the (n+1)th element in the same row. (Sedgewick discusses how this guarantees that the objective function increases and also that we stay in the simplex.)
- In the case of ties, choose the row that will result in the column of lowest index being removed from the basis (this policy prevents cycling).

An alternative strategy is **steepest descent** (actually ascent!): evaluate the alternatives and choose the column that increases the objective function the most.

Example

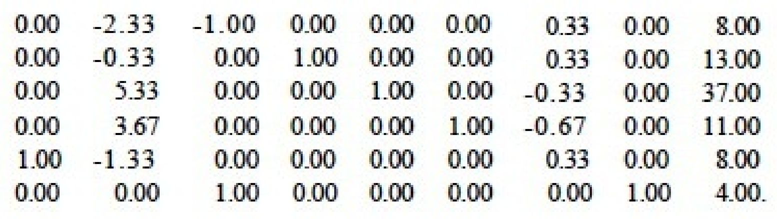
We'll solve the example given above and copied below. Keep in mind that row indices start at 0, but column indices start at 1. (See Sedgewick for discussion of issues concerning staying in the simplex, detecting unbounded simplexes, and avoiding circularity; and then CLRS if you want details and proofs.)



-1.00	-1.00	-1.00	0.00	0.00	0.00	0.00	0.00	0.00
-1.00	1.00	0.00	1.00	0.00	0.00	0.00	0.00	5.00
1.00	4.00	0.00	0.00	1.00	0.00	0.00	0.00	45.00
2.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	27.00
3.00	-4.00	0.00	0.00	0.00	0.00	1.00	0.00	24.00
0.00	0.00	1.00	0.00	0.00	0.00	0.00	1.00	4.00

There are three columns with the smallest value (-1) in row 0; we choose to operate on the lowest indexed column 1. Dividing the last number by the positive values in this column, 45/1 = 45 (row 2), 27/2 = 13.5 (row 3) and 24/3 = 8 (row 4), so we choose to pivot on row 4, as this has the smallest result.

Pivot for row p= 4 and column q = 1 by adding an appropriate multiple of the fourth row to each of the other rows to make the first column 0 except for a 1 in row 4):

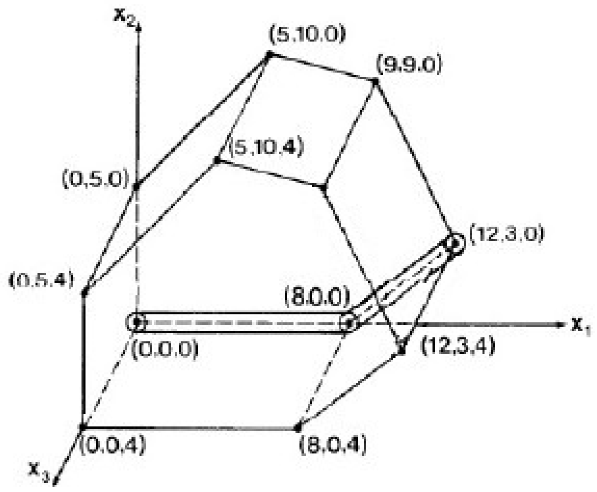


0.00	-2.33	-1.00	0.00	0.00	0.00	0.33	0.00	8.00
0.00	-0.33	0.00	1.00	0.00	0.00	0.33	0.00	13.00
0.00	5.33	0.00	0.00	1.00	0.00	-0.33	0.00	37.00
0.00	3.67	0.00	0.00	0.00	1.00	-0.67	0.00	11.00
1.00	-1.33	0.00	0.00	0.00	0.00	0.33	0.00	8.00
0.00	0.00	1.00	0.00	0.00	0.00	0.00	1.00	4.00.

After that pivot, only x_1 is a basis variable. Setting the others to 0, we have moved to vertex (8,0,0) on the simplex (see figure), and the objective function has value 8.00 (upper right corner of matrix above).

Now, column 2 has the smallest value. Rows 2 and 3 are candidates: for row 2, 37/5.33 = 6.94; and for row 3, 11/3.67 = 2.99. We choose row 3. Pivoting on row p = 3 and column q = 2:

0.00	0.00	-1.00	0.00	0.00	0.64	-0.09	0.00	15.00
0.00	0.00	0.00	1.00	0.00	0.09	0.27	0.00	14.00
0.00	0.00	0.00	0.00	1.00	-1.45	0.64	0.00	21.00
0.00	1.00	0.00	0.00	0.00	0.27	-0.18	0.00	3.00
1.00	0.00	0.00	0.00	0.00	0.36	0.09	0.00	12.00
0.00	0.00	1.00	0.00	0.00	0.00	0.00	1.00	4.00



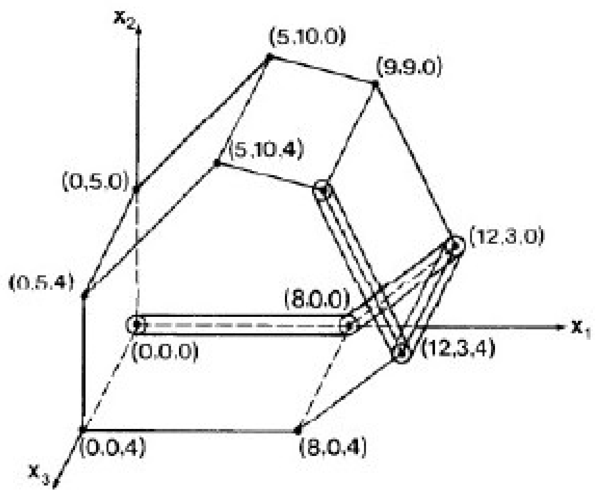
After that pivot, x_1 and x_2 are basis variables, with values 12 and 3 respectively, so we are at vertex (12,3,0). The objective function has value 15.00. The figure to the right shows how we are moving through the space.

Now pivot on column q = 3 (it has -1 in row 0) and row p = 5 (it has the only positive value in column 3).

0.00	0.00	0.00	0.00	0.00	0.64	-0.09	1.00	19.00
0.00	0.00	0.00	1.00	0.00	0.09	0.27	0.00	14.00
0.00	0.00	0.00	0.00	1.00	-1.45	0.64	0.00	21.00
0.00	1.00	0.00	0.00	0.00	0.27	-0.18	0.00	3.00
1.00	0.00	0.00	0.00	0.00	0.36	0.09	0.00	12.00
0.00	0.00	1.00	0.00	0.00	0.00	0.00	1.00	4.00.

Now all three x_i are in the basis, and we are at vertex (12,3,4).

But we are not done: there is still a negative value in row 0 (at column 7), so we know that we can still increase the objective function. I leave it to you to do the math to verify that row 2 will be selected. Pivoting on row p = 2 and column q = 7, we get::



0.00	0.00	0.00	0.00	0.14	0.43	0.00	1.00	22.00
0.00	0.00	0.00	1.00	-0.43	0.71	0.00	0.00	5.00
0.00	0.00	0.00	0.00	1.57	-2.29	1.00	0.00	33.00
0.00	1.00	0.00	0.00	0.29	-0.14	0.00	0.00	9.00
1.00	0.00	0.00	0.00	-0.14	0.57	0.00	0.00	9.00
0.00	0.00	1.00	0.00	0.00	0.00	0.00	1.00	4.00

Now row 0 has no negative values, and the columns for the three variables of interest are in the basis (all 0 except one 1 in each). We can read off the solution: x_1 = 9, x_2 = 9, and x_3 = 4, with optimum value 22.

Sedgewick's Code

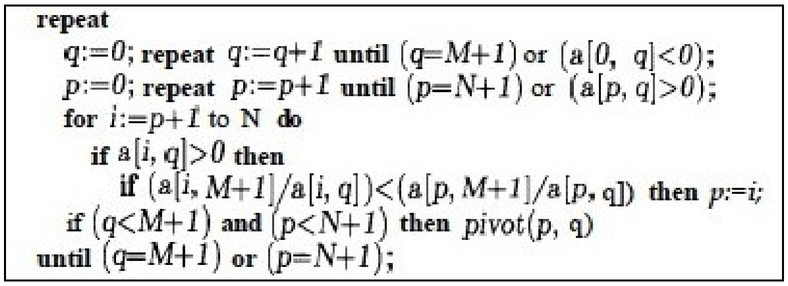
(Here I briefly explain Sedgewick's Pascal code, but if you want to understand the algorithm in detail I recommend going to CLRS for a more current treatment in pseudocode you are familiar with.)

Keep in mind that for Sedgewick there are N equations in M variables.

The main procedure finds values of p and q and calls `pivot`, repeating until the optimum is reached (q=M+1) or the simplex is found to be unbounded (p=N+1).

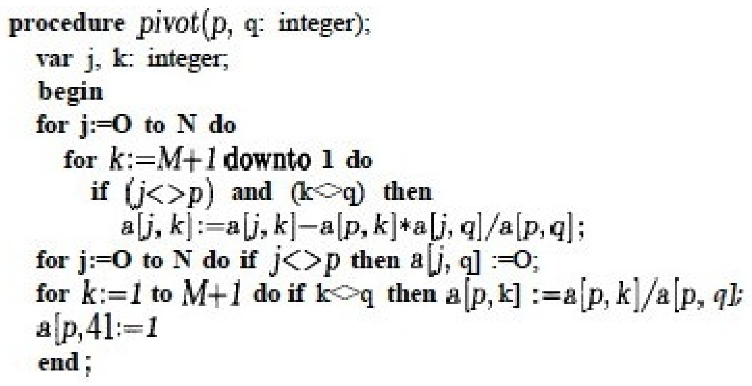
- The first line finds q by finding the first negative value in the 0th row.

- The second line finds the first positive value in the qth column.
- The `for` loop finds the best row p for pivoting by searching for the smallest ratio with the value in M+1).
- If the conditions for continuation are met, `pivot` is called.



```
repeat
  q:=0; repeat q:=q+1 until (q=M+1) or (a[0, q]<0);
  p:=0; repeat p:=p+1 until (p=N+1) or (a[p, q]>0);
  for i:=p+1 to N do
    if a[i, q]>0 then
      if (a[i, M+1]/a[i, q])<(a[p, M+1]/a[p, q]) then p:=i;
  if (q<M+1) and (p<N+1) then pivot(p, q)
until (q=M+1) or (p=N+1);
```

The `pivot` procedure has similarities to Gaussian elimination. (The `for` loops below correspond to the two innermost `for` loops of Gaussian elimination, and the outer `for` loop of Gauss corresponds to the `repeat` loop in the main procedure above):



```
procedure pivot(p, q: integer);
  var j, k: integer;
  begin
  for j:=0 to N do
    for k:=M+1 downto 1 do
      if (j<>p) and (k<>q) then
        a[j, k]:=a[j, k]-a[p, k]*a[j, q]/a[p, q];
  for j:=0 to N do if j<>p then a[j, q] :=0;
  for k:=1 to M+1 do if k<>q then a[p,k] :=a[p, k]/a[p, q];
  a[p,4]:=1
  end;
```

The innermost line is where one row is scaled and subtracted from another. Other details are discussed in Sedgewick's chapter, including the need to implement cycle avoidance and test whether the matrix has a feasible basis (absent from the code above).

What's Next

At this point, I highy recommend reading CLRS Sections 29.0 (the introduction to the chapter) through the middle of 29.3 (where the Simplex algorithm is introduced: as a "consumer" of the algorithm you don't need to read the proofs that follow in the rest of the section).

Dan Suthers
Last modified: Mon May 4 16:49:25 HST 2015
Images are from Sedgewick (1983). Algorithms. Reading, MA: Addison-Wesley. First Edition.