Chapter 7: Linear Programming in Practice

Because linear programming is so remarkably useful in practice, it has been the subject of ongoing research since its invention over 50 years ago. There have been some very interesting and valuable developments in that time. If you will be working with linear programming in practice, then you should be aware of some of the developments that are briefly surveyed in this chapter.

Alternative Algorithms for Solving Linear Programs

The simple version of the simplex method that you have seen so far, and that is presented in most introductory textbooks, is not actually used in most commercial LP solvers. Commercial LP solvers use a variation that is much more efficient when implemented on a digital computer. In fact, there are several common variations of the simplex method, as well as methods that are not based on simplex at all. Some of the most important are described below. The important issue is to know when to apply each algorithm. If speed of solution is an issue, then it may pay to search out the special solution algorithm that applies in your case.

The Revised Simplex Method

This is the workhorse algorithm implemented in most commercial simplex-based LP solvers. It is the same familiar simplex method that you know, with a few key differences that make it more efficient for computer solution. The main improvements over the basic simplex method are described next.

Updating the objective function row. When the tableau is put into proper form, the objective function coefficients of the basic variables should be zero. Knowing this, there is no point in actually carrying out those calculations. The revised simplex method calculates only the objective function row coefficients for the nonbasic variables. It needs these to choose the entering basic variable.

Finding the leaving basic variable. The leaving basic variable is found by using the minimum ratio test in which each ratio is calculated as the right hand side value divided by the coefficient in the pivot column (entering basic variable column). This means that we don't really need to know the coefficients in the other columns. Hence the revised simplex method calculates only the tableau coefficients in the entering basic variable column. Further, the minimum ratio test is not carried out for rows whose pivot column entry is negative or zero (the result is known beforehand to be "no limit"). Hence the revised simplex method calculates the right hand side value only for rows whose pivot column entry is greater than or equal to zero.

Basis update. The essential facts about any tableau are contained in a matrix called the "basis inverse". The basis matrix is constructed by including only those columns from the original tableau that correspond to the current set of basic variables. This will be a square matrix. If you wanted to calculate the values of the basic variables from scratch, you would need to invert this matrix as part of the process. Hence the basis inverse is an implicit part of any tableau. We don't have to do an entire basis matrix inversion at each tableau though because the simplex method provides us with a way of updating the basis inverse as we move from tableau to tableau. The revised simplex method uses very efficient updating methods to update a representation of the basis inverse (called an LU factorization) at each iteration.

Efficient storage. Finally, the revised simplex method uses efficient sparse-matrix storage schemes. A "sparse matrix" is filled mostly with zeros. If you were to write out the tableau for most industrial-scale linear programs, you would find that they are sparse; more than 90% of the entries are zero. This is because each constraint normally only involves a few of the variables from among the many defining the problem, and each variable usually appears in only a few constraints and/or the objective function. Instead of storing a huge matrix that is mostly zeros, sparse-matrix techniques instead store the address of the cell of the matrix, along with the value in the matrix cell. For example, if the matrix has the value 75.4 in the 98th row and 506th column, a sparse matrix representation might be (98, 506, 75.4). Even more efficient representations are used in many LP codes. Because only the nonzero elements of the matrix are stored, sparse matrix schemes are hugely more space-efficient.

Other than these implementation differences, the revised simplex method is basically the same as the simple simplex method that you have learned. However, those implementation differences can be important in practice. For example, knowing which sparse matrix representation is used to store the matrix can help you to quickly estimate the memory requirements for a particular LP model so you will know whether the problem can be solved on the machine that you are using.

The Dual Simplex Method

Every linear programming model has a related mirror-image representation called the *dual*. Without going into details, the dual is constructed by turning the LP matrix sideways: the objective function row becomes the right hand sides column, the right hand sides column becomes the objective function row, and the constraints are read along the original columns instead of along the rows. There are also rules governing the conversion of the variable bounds and the constraint types ($\leq$, $\geq$, $=$). It turns out that if you solve the dual form of an LP model, you can also recover the solution to the original problem (called the *primal*)!

There are many other interesting relationships between the primal and dual models. For example, as you proceed through the primal solution, every feasible cornerpoint solution is dual-infeasible. The reverse is also true: every dual-feasible cornerpoint traversed while solving the dual problem is primal-infeasible. The only point that is both primal-feasible and dual-feasible is the optimum point.

Now the speed of the simplex method on a particular problem is mainly governed by how many iterations must be performed before the optimum is found, i.e. how many cornerpoints must be visited. The number of cornerpoints that must be visited is related to how many cornerpoints exist in the model: when some model *A* has more cornerpoints that some model *B*, then you naturally expect that the solution of model *A* will usually take longer than the solution of model *B*. The number of cornerpoints generally increases with the number of constraints in the model, because the constraints define the cornerpoints. Hence a model having more constraints than another most often takes longer to solve because the extra constraints define more cornerpoints.

Now let’s take a look at a model that has many more constraints than variables. The dual of the model (taking the matrix sideways) will have many *fewer* constraints than the original model. All else being equal, solving the dual model will take much less time than solving the original primal model, and yet the solution to the original model can be recovered from the dual solution! Solving the dual then permits a massive speed-up in solution time for very little effort in cases where the number of constraints is greater than the number of variables.

The *dual simplex method* provides a way of using the dual representation while operating on the primal model so that the dual representation is never explicitly formed. Some solver manufacturers report that the dual simplex method outperforms the primal simplex method on a large majority of their test cases.

Solving Network Linear Programs

We will have more to say about this in the next chapter. For now, it is sufficient to know that there is a common and very useful special case of linear programming known as networks, or network flow programming. The main idea here is that the network is constructed by connecting various nodes via arcs, which transport flow. Subject to limits on the arc flow capacities and demands for flow at various nodes, the goal is to find the set of arc flows that solves the problem at least cost.

The specialized network simplex method is up to 100-200 times faster than the general simplex method applied to the same problem. Many solvers now have routines that first scan your LP to see whether it has any embedded network substructures. If it finds an embedded network, then it is able to speed up the overall solution by applying the network simplex method to the network substructure and patching this partial solution together with the rest of the solution. There are a number of other algorithms for solving network LPs, such as the auction algorithm, though the network simplex method is the best known and most used.

There are several different classes of network models, as we shall see in the next chapter, including transportation, assignment, and transshipment models. There are efficient special algorithms for these problems, including the Northwest Corner method, Vogel’s method, the Hungarian method, etc. The maximum-flow/minimum-cut network model can also be solved by numerous special-purpose algorithms including the original Ford and Fulkerson method.

There are also several classes of models that are partway between network models and general LPs. Processing networks, for example, can be viewed as networks with general side constraints that fix the proportions of the flows at a node. Generalized networks are another class of model in which the flow out of an arc is a fixed multiple (including a negative multiple!) of the flow into the arc. Both of these classes have specialized solution algorithms that are many times faster than applying a general revised simplex algorithm.

Special-purpose, very fast algorithms abound in network programming.

Interior Point Methods

Interior point methods were a very surprising development when they arrived in 1979. For the first 30 years of linear programming, all the methods for solving general LPs were based on the simplex method: the important idea was to proceed along the exterior of the feasible region, moving from cornerpoint to cornerpoint. Then along came the notion of moving through the *interior* of the feasible region, and not visiting cornerpoints at all until, perhaps, the final solution.

Khatchian (there are various spellings of this Russian name) introduced the first interior point method in 1979. It is called the ellipsoid method and it works by fitting multidimensional ellipsoids into the feasible region. The axes of the ellipsoid determine the direction for further movement, and the ellipsoids become gradually smaller and smaller as they move farther and farther into the optimum cornerpoint of the feasible region. See Figure 7.1 for an illustration. Khatchian's method was quite slow in practice, much slower than the simplex method, but it was a theoretical breakthrough.

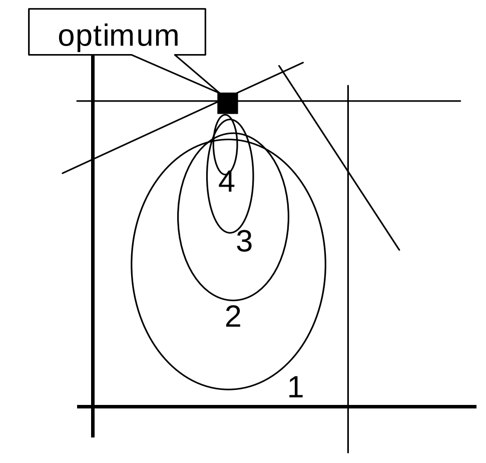

Figure 7.1: Illustration of Khatchian's ellipsoid method.

Karmarkar's method appeared in 1984 and was a major development that was reported in newspapers worldwide. Karmarkar's method uses transformations from projective geometry to determine the direction for movement through the interior of the feasible region. It is a very efficient method for very large LPs, but the revised simplex method is faster for small to medium LPs. The reason for this is that each iteration of Karmarkar's method is very costly in terms of computing time, but a complete solution generally requires only a small number of iterations. This trade-off pays off for large LPs, but for smaller LPs, the very quick iterations of the revised simplex method generally arrive at the solution first.

AT&T, Karmarkar's employer, originally sold an implementation of his method in a package with a specialized computer at a price of about US$1 million. Many of the essential implementation details were also not released so that the method could not be copied. Seen as a bit of an affront by the rest of the optimization community, this spurred a great deal of research into interior point methods with the result that much improved interior point algorithms are widely available today. Many of these more recent interior point algorithms borrow ideas from nonlinear programming and are known as “barrier methods”.

Interior point methods have great practical value in solving large LPs, but are also of special interest to researchers because they prove for the first time, theoretically, that LPs can be solved quickly (in what is known as “polynomial-time”). Oddly, the simplex method, so successful in practice, has poor theoretical performance. It is possible, for example, to construct a special pathological LP in which the simplex method has to visit every cornerpoint before arriving at the optimum.

In the original basic form, most interior point algorithms approach, but never actually reach the optimum cornerpoint. To function correctly, the current solution point must always remain strictly in the interior region. This prevents the use of most sensitivity analysis features, which depend on arriving at an optimum cornerpoint. For this reason, many solvers have a “cross-over” feature in which, near the optimum, the solution method switches over to the simplex method for the last few iterations to the optimum cornerpoint. Then the sensitivity information is available.

If you will be solving extremely large LPs, then you need to know a bit about interior point solution methods. In practice, interior point methods are most likely to be the best choice for large models, but some versions of the simplex method (especially the dual simplex method) may prove faster in some cases.

Other LP Solution Algorithms

There is a long list of other solution algorithms for general linear programs, for example, the *out-of-kilter method*. There are variants for *upper-bounded variables* and *decomposition methods* that tear large LPs into smaller pieces, solve these independently and stitch the results together, with a larger iterative process to balance the overall solution between the pieces.

Column-generation techniques provide ways of identifying the entering basic variable without calculating all of the objective function coefficients of the nonbasic variables. The columns are then generated only as needed. This can be a valuable time-saver for problems having many variables (e.g. millions).

There is an astonishing gamut of inventive methods and algorithms, with more arriving daily. The solution of linear programs continues as a lively area of research after all these years. Perhaps you can contribute a breakthrough of your own, becoming as famous as Dantzig or Karmarkar!

Advanced Techniques in Linear Programming

There are numerous advanced techniques included in any commercial-quality LP solver. You should be generally aware of these techniques, though the details vary from solver to solver.

Starting Points

Many solvers include the ability to start the solution from any user-chosen point. You may, for example, have reason to believe that the solution is very close to a particular point, perhaps the solution from the last time you ran the LP, maybe in the last business quarter. The solver then starts a phase 1 solution at that point, even if it is not a cornerpoint, moves swiftly to a phase 1 cornerpoint, and continues iterating from there to the optimum solution. A well-chosen initial point can greatly speed the overall solution.

Crash Starts

A *crash start* is a heuristic method to quickly find a point or basis that is likely to be near-feasible or near-optimal. These heuristics are highly individual, but all have the objective of generating an initial point that is closer to optimality than the usual method of simply setting all of the original variables to zero. As described above, the solver is then able to start iterating from this point immediately.

Advanced Starts: Warm and Hot Starts

If the solution has been stopped for some reason, then the current basis and other information can be stored. When the solution is restarted, this stored information provides a *hot start* which permits the solver to simply pick up where it left off and continue to the optimum, rather than having to go through the entire set of iterations to reach the current point again.

If there have been some minor changes to the LP since the solution process was stopped (could have stopped at the optimum), then you may be able to use a *warm start* in which the previous solution and basis are initially used to restart. This usually means that you can arrive at the new optimum point in only a few iterations. Warm starts are extremely useful if you are repeatedly solving different versions of the same LP, each version of which is only very slightly different from the last. This is very useful in mixed-integer linear programming, and also in the algorithms for analyzing infeasibility in linear programs.

Depending on the changes that have been made, an expert can select which algorithm to use when the problem is restarted. For example, if a constraint has been added that renders the current point infeasible, it will still be dual-feasible, so that the dual simplex method can be used to resume the solution process. By the same token, adding a new variable will mean that the current solution is still primal-feasible, but it will usually be dual-infeasible, so the primal simplex method should be used to restart the solution process.

Scaling

LP solvers run on digital computers. Anyone doing mathematical calculations on a digital computer needs to be aware that only a finite number of bits are used to represent real-valued numbers (most commonly 64 bits in linear programming implementations). By necessity, this introduces some granularity into the representation of real numbers, which can cause calculation difficulties. Real numbers are represented in two parts using scientific notation: the number itself (or *mantissa*), and the corresponding power of ten (or *exponent*) by which it is multiplied. If a similar scheme were applied to the familiar base-10 number system using two digits for the mantissa and one digit for the exponent, some example number representations might be 3.8×10^{2} or 1.2×10^{-3}.

When we need to add two numbers in the computer, the first step is to align the exponents so that the addition can take place (i.e. convert both numbers to the same exponent, usually the larger one). But look what happens when we try to do this with our hypothetical limited-digit example numbers above: the addition becomes 3.8×10^{2} + 0.000012×10^{2}. Because 0.000012×10^{2} is not representable with only 2 digits for the mantissa, it becomes 0.0×10^{2}, i.e. zero, so the addition does not actually take place. The same problem arises when numbers are represented in binary form.

Similar difficulties plague most numerical operations on digital computers. As we have seen above, the difficulties are particularly acute when the numbers are of very different size. Numerical problems with LP solutions crop up most often when the coefficients in the model are of widely differing size. If possible, this should be avoided during the formulation phase, perhaps by choosing units so that the coefficients are similar in size (use liters instead of milliliters for example).

Modern LP solvers normally take some automatic steps to avoid the problem of coefficients of widely differing size. This process is called *scaling*, and involves, for example, dividing a row or column by a factor to reduce the range of the scales in the coefficients. The scaling process is normally invisible to the user: scaling is carried out, the model is solved, and in a post-processing step the scaling is reversed so that the solution is correctly reported. Some solvers may produce warning messages along the lines of “model is badly scaled”.

Analyzing Infeasibility

What do you do when the solver reports that the model is infeasible? How do you pore through the many constraints (perhaps hundreds of thousands) to find the difficulty? Finding an *Irreducible Infeasible Subset (IIS)* of constraints can be extremely helpful in a case like this. An IIS is a small set of constraints from among the many making up the model and has this property: the IIS itself is infeasible, but removing any one constraint from the IIS renders the remainder of the constraints in the set feasible. Thus every constraint in the IIS contributes to the infeasibility in some way: they all have to be there for infeasibility to be present.

Routines to find IISs are now included in most commercial LP solvers. Finding an IIS helps in debugging the model by focusing attention on a small subset of the constraints in the model. It still requires human insight to examine the IIS to determine what is wrong. Perhaps a greater-than constraint has accidentally been reversed. Perhaps a parameter has been entered incorrectly.

There are a number of combinations of the base IIS-finding routines. Two common selections provided as options in the LP solvers (e.g. Ilog's CPLEX solver) will permit either fast isolation of an IIS, or a slower algorithm can be used to find a better IIS (where a better IIS is one having few row constraints because these are usually easier for the human to interpret).

Presolving

Presolving is reasoning that is done about the model prior to solution with the aim of simplifying it so that a much smaller model is actually presented to the solver. This usually speeds the overall solution time. For example, consider an LP which includes the following three constraints: $x_1 \geq 8$, $x_2 \geq 7$, $x_1+x_2 \leq 15$. It is obvious in this case that the only solution to these three constraints is to set $x_1=8$ and $x_2=7$. Having done this, there is no longer any point in including the constraint $x_1+x_2 \leq 15$ in the model at all. Also, the fixed values of x_1 and x_2 can now be propagated through the rest of the model, resulting in other simplifications. For example, another constraint such as $x_1+x_{27} \geq 96$ can now be reduced to $x_{27} \geq 88$.

In some cases, the presolver may be able to detect infeasibility prior to the LP solution, as in the following example: $x_1 \geq 0$, $x_2 \geq 4$, $x_1+x_2 \leq 3$. The simple bound substitution methods usually employed in the presolver detect infeasibility in this case. Because of the long chain of reductions that it may have gone through, the presolver may not be able to give an accurate report as to the cause of the infeasibility. In that case it may be better to turn the presolver off, then submit the LP to a solver that is able to identify an Irreducible Infeasible Subsystem (IIS). The IIS will be more useful in finding the cause of the infeasibility.

General Analysis of Linear Programs

In developing a large and complex linear program, you are often faced unanticipated questions in trying to explain your results. It is usually very difficult to get the kind of information that you need from the solver output. Fortunately, there are a couple of computer tools available for general probing of linear programs. The ANALYZE program [http://www.cudenver.edu/~hgreenbe/imps/analyze.html] allows you to examine your model and solution in numerous different ways to find insights. MProbe [http://www.sce.carleton.ca/faculty/chinneck/mprobe.html] offers a different set of probes for all forms of mathematical program, including linear programs.

Both programs are for expert users. Sooner or later you may need answers that you can't get from the solvers directly. Take a look at ANALYZE and MProbe then.

Modeling Systems and Languages

Modeling systems and languages have been a real advance in the ease of use of mathematical programming. Modeling languages make it simple to express very large and complex models in a way that is easy to understand. Prior to their development, each solver, whether linear or nonlinear, had a different input format, each of which was arcane and complex. Many LP solvers relied on the ancient MPS format for input, a format that is suitable only for machine input, not for human understanding. Many nonlinear solvers required that the nonlinear functions be written as a subroutine in a language such as Fortran or C and then compiled and attached to the solver, a process with its own complications.

Modeling languages, on the other hand, allow the user to easily specify a mathematical program of any type: linear, nonlinear, mixed-integer, etc. The models are normally written in a very straightforward manner, quite similar to how it might be written in a textbook. For example, the Acme Bicycle Company model appears as follows in the AMPL language:

```
var x1;
var x2;
maximize profit: 15 * x1 + 10 * x2;
subject to mtn_bike_prodn: x1 <= 2;
subject to racer_prodn: x2 <= 3;
subject to metal_finishing: x1 + x2 <= 4;
```

In addition, you can index over sets, so that very large models can be written very compactly. A longer objective function might be written as follows, for example:

```
maximize profit: sum {j in P} c[j] * X[j];
```

where the set P is defined elsewhere in a data file. This allows the analyst to concentrate on the correct form of the model without being overwhelmed by the mass of detail that results when the sets are expanded. The associated modeling system takes care of expanding the model before submitting it to the solver, and takes care of receiving the results and making them available in a user-friendly manner.

Modeling systems also normally permit the attachment of numerous different solvers. Hence if you have been using a simplex-based LP solver, but the problem has grown in size so that you wish to try an interior-point method, then it is as simple as including a statement that directs the system to connect to the new solver. This is a godsend in solving nonlinear problems where the results are apt to differ significantly depending on the solver that you use, meaning that you may want to try several different solvers. This avoids the nightmare of writing your complex nonlinear problem in several different input formats.

Modeling systems may also have a number of other useful features. Many have a built-in presolver to simplify models before they are solved. They may have language extensions permitting the programming of loops so that many variations of a base model can be set up and solved easily. Database connectivity is normally provided, along with report-writing capabilities, etc. Figure 7.2 gives a schematic view of a typical modeling system.

If you will be doing serious optimization work, then the use of a modeling language is essential. There are numerous good ones on the market such as AMPL [http://www.ampl.com], GAMS [http://www.gams.com], MPL [http://www.maximal-usa.com], AIMMS [http://www.aims.com], etc. Most have a low-cost student edition that is bundled with student versions of a few commercial solvers. Why not get started with one now? The languages scale to full commercial size, so what you learn now will be directly usable in your professional life.

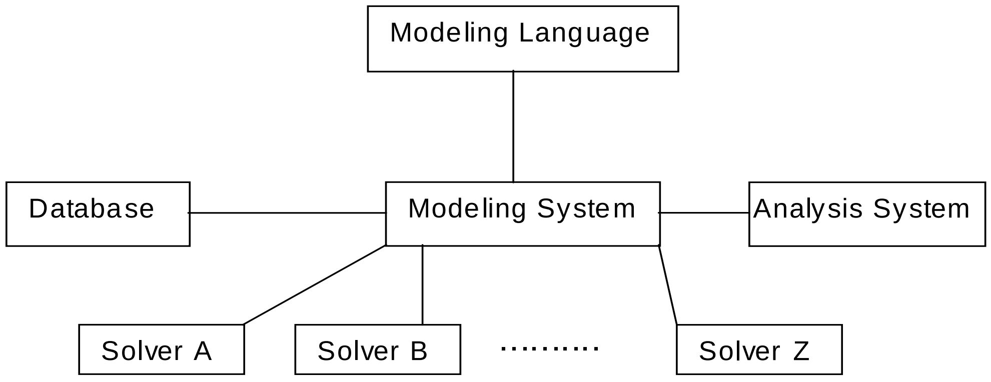

Figure 7.2: Schematic view of a typical modeling system.