PRACTICAL OPTIMIZATION: A GENTLE INTRODUCTION

John W. Chinneck
Systems and Computer Engineering
Carleton University
Ottawa, Canada

Available online at
www.sce.carleton.ca/faculty/chinneck/po.html
This version dated June 23, 2015.

Table of Contents

Chapter 14: Heuristics for Discrete Search: Genetic Algorithms and Simulated Annealing. Some problems are just too big for branch and bound, in which case you must abandon the guarantee of finding the optimum solution and instead opt for heuristic methods which can only guarantee to do fairly well most of the time. Genetic Algorithms and Simulated Annealing are two popular heuristic methods for use on very large problems.

Chapter 15: Dynamic Programming. This optimization technique builds towards a solution by first solving a small part of the whole problem, then gradually incrementing the size in a series of stages until the whole problem is solved. Efficiency results from combining the local solution for a stage with the optimum found for a previous stage. We look at the simplest deterministic discrete cases.

Chapter 16: Introduction to Nonlinear Programming (NLP). NLP is a lot harder than linear programming. We start by looking at the reasons for this. Next we look at the simplest method for solving the simplest type of NLP: unconstrained problems that consist only of a nonlinear objective function. The method of steepest ascent/descent is described.

Chapter 17: Pattern Search for Unconstrained NLP. What do you do if you don't have access to gradient information? In that case you can use pattern search techniques (also known as derivative-free, direct search, or black box methods). We look at the classic Hooke and Jeeves pattern search method.

Chapter 18: Constrained Nonlinear Programming. Now that we have some idea of how to solve unconstrained NLPs, how do we deal with constrained NLPs? The first idea is to turn them into unconstrained NLPs of course! This is done by using penalty and barrier methods which replace or modify the original objective function in ways that make feasible points attractive in the resulting unconstrained problem.

Chapter 19: Handling Equality Constraints in NLP. Equality constraints are the hardest to handle in nonlinear programming. We look at two ways of dealing with them: (i) the method of Lagrange, and (ii) the Generalized Reduced Gradient (GRG) method. And we take a look at making linear approximations to nonlinear functions because we need that for the GRG method.

Chapter 20: Function and Region Shapes, the Karush-Kuhn-Tucker (KKT) Conditions, and Quadratic Programming. Function and region shapes (convex, nonconvex, etc.) are important in understanding how nonlinear solvers work. The KKT conditions are very useful in deciding whether a solution point is really a local optimum in a nonlinear program, and form the basis for some methods of solving quadratic programs. Last revision: April 1, 2015.

See the online algorithm animations linked from www.sce.carleton.ca/faculty/chinneck/po.html.

Chapter 1: Introduction

Practical optimization is the art and science of allocating scarce resources to the best possible effect. Optimization techniques are called into play every day in questions of industrial planning, resource allocation, scheduling, decision-making, etc. For example, how does a global petroleum refiner decide where to buy crude oil, where to ship it for processing, what products to convert it to, where to sell those products, and at what prices? A maximum-profit optimization model is used to solve this problem. How does an airline know how to route its planes and schedule its crews at minimum cost while meeting constraints on airplane flight hours between maintenance and maximum flight time for crews? A minimum-cost optimization model is used.

Many of the large scale optimization techniques in general use today can trace their origins to methods developed during World War II to deal with the massive logistical issues raised by huge armies having millions of men and machines. Any techniques that promised to improve the effectiveness of the war effort were desperately needed, especially in the face of limited numbers of people, machines, and supplies. What is the optimum allocation of gasoline supplies among competing campaigns? What is the best search and bombing pattern for anti-submarine patrols?

The fundamentals of the first practical, large-scale optimization technique, the simplex method, were developed during the war. The simplex method was perfected shortly after the war when the first electronic computers were becoming available. In fact, the early history of computing is closely intertwined with the history of practical optimization. In the early years, the vast majority of all calculation on electronic computers was devoted to optimization via the simplex method!

The entire history of large-scale practical optimization is very short. In fact, the inventor of the simplex method, George Dantzig, died only very recently in 2005. He told an amusing story at the 1997 International Symposium on Mathematical Programming. A graduate student of optimization, on hearing that he was present at the meeting, rushed up to him and pumped his hand excitedly, exclaiming “Professor Dantzig! I’m so glad to meet you! I thought you were dead!”

The field itself is an exciting hotbed of innovation, and a terrific field for researchers. Spectacular breakthroughs are still happening, like the development of interior point methods for linear programming in the late 1980s. More recently, ideas abound as researchers from mathematical backgrounds trade thoughts with researchers from a computer science tradition. An important example is the emergence of constraint programming as a powerful paradigm for optimization. One of the largest companies in the field of practical optimization, Ilog, is also one of the newest, and has achieved its position by combining the best of the mathematical techniques with the best of the constraint programming techniques from computer science.

New optimization techniques are arriving daily, often stimulated by fascinating insights from other fields. Genetic algorithms, for example, use an analogy to chromosome encoding and natural selection to “evolve” good optimization solutions. Optimization techniques play a role in training artificial neural networks used in artificial intelligence research for pattern recognition. “Swarm intelligence” research looks at using independent software agents to collectively solve various optimization problems. Perhaps you have ideas of your own: maybe you can be the next George Dantzig….

Today, optimization methods are used everywhere in business, industry, government, engineering, and computer science because optimization problems arise just as regularly in these fields as they did during World War II. An edge in maximizing profits or minimizing costs can often mean the difference between success and failure in business. In academia, the teaching of optimization takes place in several different departments and under different course names. You will find optimization taught in university departments of engineering, computer science, business, mathematics, and economics; and in courses going under the names of optimization, mathematical programming, operations research, industrial engineering, and algorithms.

Here are a few more examples of optimization problems:

- How should the transistors and other devices be laid out in a new computer chip so that the layout takes up the least area?
- What is the smallest number of warehouses, and where should they be located so that the maximum travel time from any retail sales outlet to the closest warehouse is less than 6 hours?
- How should telephone calls be routed between two cities to permit the maximum number of simultaneous calls?

The process of optimization is shown schematically in Figure 1.1. You typically begin with a real problem, full of details and complexities, some relevant and some not. From this you extract the essential elements to create a model, and choose an algorithm or solution technique to apply to it. In practical problems, the computer will carry out the necessary calculations.

There is an unavoidable loss of realism as you move down the diagram, from *real world problem* to *algorithm, model, or solution technique*, and finally to *computer implementation*. An important part of successful applied optimization is skill in determining what is and what is not important during this process of abstraction. For example, suppose that production efficiency increases slightly nonlinearly as production volume increases on your assembly line. Is the nonlinearity significant, or is it accurate enough to use a much simpler linear approximation? And where do you get the “profit per item produced” that your model requires? In reality it is an estimate produced by the accounting department who consider many factors in arriving at the number, including the pay rates for workers mandated by the last union contract, the estimated prices of raw materials and energy, and the estimated sales based on forecasts provided by the marketing department. The trick is to learn, through experience, how to match messy real world situations with the correct optimization algorithms, and how to combine algorithms to create new solvers as the situation may demand.

The arrows in Figure 1.1 indicate the normal process of the optimization cycle. Moving from the *real world problem* to the *algorithm, model, or solution technique* is known as **analysis.** It is here that the main work of abstracting away irrelevant details and focusing on the important elements takes place. In many cases, the process of analysis is extremely valuable by itself, even without carrying out any subsequent optimization. Just laying bare the essential elements of a problem is often extremely valuable to the client, and leads to insights into how to approach the problem. Many of the skills needed here are in courses titled *systems analysis* in software engineering or computer science programs. Skills needed include the ability to interact with people to find information, and expertise with tools for expressing your understanding of the situation in a manner that is clear, complete and easily understood by nontechnical folks. I highly recommend a course in systems analysis for anyone who will be doing practical optimization.

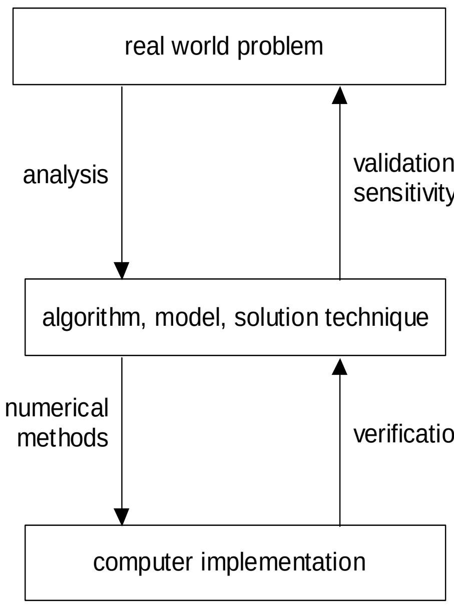


Figure 1.1: The optimization process.

Moving from the *algorithm, model, or solution technique* to the *computer implementation* is generally the province of **numerical methods** (and other computer science techniques). This covers issues such as calculation accuracy when using digital computers, efficient implementations of matrix inversion techniques, and the like. An in-depth knowledge is not normally necessary, but some familiarity is helpful when trying to adjust the control parameters of the solvers that you may be using.

Moving from *computer implementation* back to the *algorithm, model, or solution technique* is called **verification**. The main idea is to make sure that the computer implementation is actually carrying out the algorithm as it is supposed to. Again, this will not be of great concern for users of well-tested commercial optimizers.

You will be greatly concerned with **validation and sensitivity analysis**, the process of moving between the *algorithm, model, or solution technique* and the *real world problem*. It is here that the loop is completed and you finally compare the results you are obtaining with the real situation. Are the results appropriate? Do they make sense? Does the model need to be modified, or another solution technique chosen? If so, then the loop begins again.

Validation is the process of making sure that the model or solution technique is appropriate for the real situation. **Sensitivity analysis** looks at the effect of the specific data on the results. For example, consider again how the "profit per item produced" is estimated. Suppose that you are not very confident that the estimate is accurate. Sensitivity analysis asks how sensitive the final results are to variations in data such as the profit per item produced. It may show, for example, that the final solution changes very little even with a large variation in profit per item produced. In this case you heave a sigh of relief. On the other hand if it turns out that the results change dramatically when the estimated profit per item produced changes very slightly, then you have cause for worry, especially if there are millions of dollars depending on the outcome of your analysis! In this case you will need to run a number of scenarios showing how things will turn out at various values of profit per item produced in order to arrive at a final recommendation.

This book concentrates on the top two boxes in Figure 1.1, and on the processes of moving between them. You will be exposed to some of the details of the algorithms and their computer implementations so that you have enough understanding to deal with unexpected outcomes. For success in practical optimization, you will need three things:

- a broad knowledge of the classical generic optimization problems,
- a knowledge of the available solution algorithms and their implementations in commercially available solvers,
- practice in the process of analyzing and solving problems (as in Figure 1.1).

A good approach to acquiring these essential elements is to:

- use this book to learn about the solution algorithms, and for examples of problem formulations,
- obtain practice in the optimization process through homework examples and examinations which consist entirely of applied problems, i.e. purely theoretical exercises of the mechanics of the solution algorithms are to be avoided.

Of course, textbook word problems are vastly simpler than the complexities, irrelevancies, and political difficulties that arise in real situations, but they do provide some practice in analysis and model formulation.

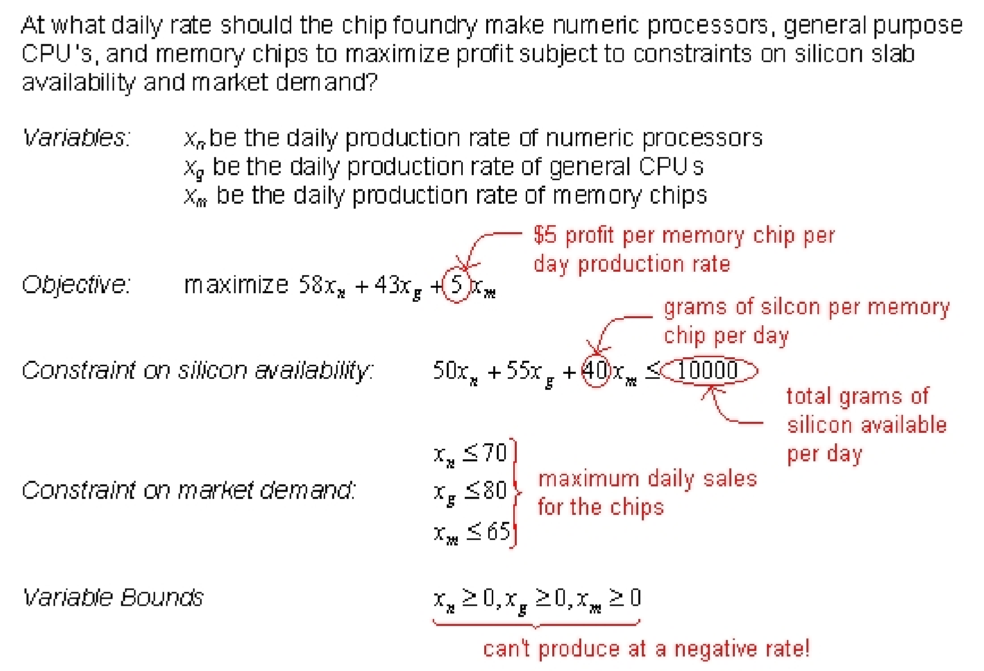
This book is designed as a one-term introduction to the most important topics in applied optimization. It is impossible to cover all of optimization in a one-term course: there are entire yearlong courses on each of the individual topics that we will cover! This book provides a fairly broad survey at a medium depth. There are numerous topics that you should be aware of, but which cannot be covered in an introductory book like this one; brief sketches of these topics appear throughout the book.

The main goals of a course using this book should be to equip students to:

1. recognize problems that can be tackled using the tools of applied optimization,
2. formulate optimization problems correctly and appropriately,
3. solve optimization problems, primarily by selecting and applying the correct solvers, but also possibly by writing special software or hiring experts.

These abilities will be an excellent addition to your skills toolkit, and especially useful as the world becomes more complex and computer-centric. Note that a course in optimization or operations research is required in most MBA courses, in business, industrial engineering (and many other engineering programs), and economics. Such courses are usually a recommended option in computer science as well. These courses are included in all those programs precisely because the material finds so many practical applications.

A brief overview of the main topics covered in the book follows.



At what daily rate should the chip foundry make numeric processors, general purpose CPU's, and memory chips to maximize profit subject to constraints on silicon slab availability and market demand?

Variables: x_n be the daily production rate of numeric processors
x_g be the daily production rate of general CPUs
x_m be the daily production rate of memory chips

Objective: maximize $58x_n + 43x_g + 5x_m$

Constraint on silicon availability: $50x_n + 55x_g + 40x_m \leq 10000$

Constraint on market demand: $x_n \leq 70$, $x_g \leq 80$, $x_m \leq 65$

Variable Bounds $\underbrace{x_n \geq 0, x_g \geq 0, x_m \geq 0}$

$5 profit per memory chip per day production rate
grams of silcon per memory chip per day
total grams of silicon available per day
maximum daily sales for the chips
can't produce at a negative rate!

Figure 1.2: A simple linear program.

Linear programming is the most widely used of the major techniques for constrained optimization. “Programming” in this sense does not imply computer programming – it is an old word for “planning”. In linear programming, all of the underlying models of the real-world processes are linear, hence you can think of “linear programming” as “planning using linear models”.

While linear models (e.g. each refrigerator produced consumes exactly 5 kWh of electricity and 60 kilograms of steel) might seem too simple to model many real world problems, they actually have a vast range of applications. Many production problems are adequately modeled with linear relations, and many routing problems are linear, for example. The largest practical constrained optimization problems that are regularly solved are linear programs. Airline scheduling problems may have hundreds of thousands of constraints and a million variables, for example.

A portion of a typical linear programming problem is shown in Figure 1.2. Note that all of the relationships, the single objective function and the multiple constraints, are all linear relationships. The problem is solved by determining the values of the variables that maximizes the profit relationship in this case.

Breakthroughs in the solution of linear programs include the introduction of the simplex method in 1947, and Karmarkar's interior point method in 1984. Research on improved methods continues to this day.

Networks are natural models of many real situations. A network model consists of nodes and connecting lines called arcs, as sketched in Figure 1.3. The nodes in Figure 1.3 represent cities in the eastern part of Canada, and the lines between the nodes may variously represent roads or railways or telephone lines, depending on the application.

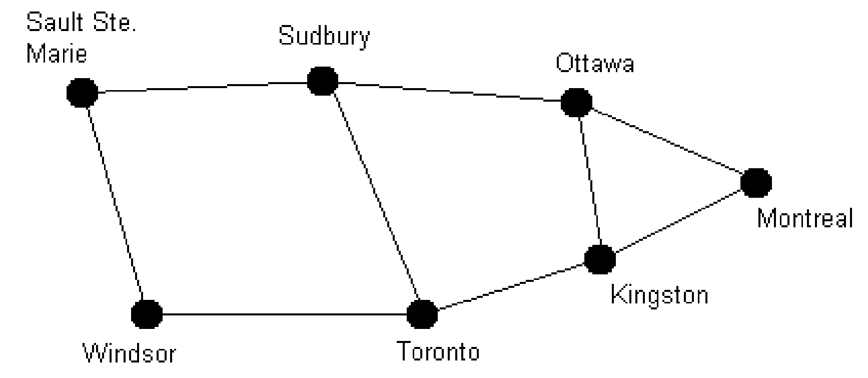


Figure 1.3: A simple network diagram.

Examples of some questions that might be posed, given the network model in Figure 1.3:

- Label each of the arcs with the distance between the cities that it connects. What is the shortest route between Sault Ste. Marie and Montreal? This may be a simple problem to solve in Figure 1.3, but now imagine solving it on a network diagram of a million-transistor layout: brute force enumeration of all of the possible routes simply won't work! This is the classic *shortest route problem*. If the arcs are labeled with travel times, then it is the fastest-route problem.
- Let the arcs represent telephone lines, and label each arc with the maximum capacity of the phone line in terms of number of simultaneous calls. Now how should you route calls from Windsor to Ottawa so as to maximize the number of simultaneous calls that can be handled? Note that calls can go via several routes at once: Windsor-Toronto-Kingston-Ottawa and Windsor-Sault Ste. Marie-Sudbury-Ottawa and Windsor-Toronto-Sudbury-Ottawa, etc. This is the *maximum flow problem*. The related *minimum-cut* problem can be used to answer

questions about where larger-capacity telephone lines should be constructed so that more calls can flow simultaneously between two cities.

- Label the arcs with the cost of constructing a high-speed fiber-optic computer line between cities that it connects. Which of the fiber-optic lines should be constructed so that all of the cities can be connected together at minimum total cost? This is the *minimum spanning tree problem*.

There is a huge array of other problems that can be tackled by network formulations and solution methods.

We will also take a brief look at the PERT methodology. This is a network-based technique for managing complex projects consisting of many activities, some of which have precedence relationships, and some of which can run in parallel. Typical questions include: What is the shortest time in which the project can be completed? How do limited resources affect the completion time? PERT techniques have many applications beyond project management, for example in computer chip design where there are questions of signal propagation time through a network of transistors.

In linear programming, all of the variables are real-valued, that is they can take on fractional values such as 3.7 or 19.331. In **integer programming** the variables are restricted to taking on integer values, or perhaps only binary (0 or 1) values. Many problems are integer, for example:

- Given the members of a swim team and knowledge of their typical times for various strokes, how do you assign the members of the team to the legs of a multi-stroke relay to achieve the fastest total time? This is a binary problem: you can either *assign* someone to a leg of the relay or not assign them; you can't assign half of each of two people!
- How do you *schedule* professors, rooms, courses and students so that there are no conflicts (or a minimum number of conflicts)? This is again a binary problem: you can't assign a professor to be in two places at once.
- How do you determine the number of rail cars to assign to various cargoes? This is an integer problem: each rail car either goes or stays.

In practical problems it is generally impossible to enumerate every possible solution and just take the best of the lot. Integer programming problems experience "combinatorial explosion" in which the number of solutions grows extremely rapidly as the number of elements in the problem increases because of the many, many possible ways of combining the elements. For example it is very easy to construct relatively small integer programming problems for which the number of possible different solutions is greater than the number of atoms in the known universe. That is *way* too many to enumerate. Adding to the difficulty, it is rarely possible to apply real-valued techniques such as linear programming and then simply round the results to the nearest integer.

We will look at the branch and bound technique for structuring the solution to slow down the combinatorial explosion, and at genetic algorithms for cases in which the combinatorial explosion defeats even branch and bound.

Dynamic programming is used when making a sequence of interrelated decisions. It is used in *equipment replacement* problems for example, where you need to keep a functioning heart monitor in place over a number of years, and which to plan the sequence of repairs and replacements that minimize the total cost of the horizon under consideration.

Nonlinear programming is identical to linear programming, except that the relationships can have a nonlinear form. This makes the problem *much* harder than linear programming, for reasons that we will explore in detail later. However nonlinearity is inescapable in many real-world problems. Nonlinear relationships may, for example, be required to accurately model the losses in radio signal strength with distance as you optimize a microwave communication system. We will explore some of the simpler techniques of nonlinear programming, those that are most often implemented in commercial solvers. Fortunately, while there is a huge and rapidly growing research literature on the mathematics of nonlinear programming, only a relatively small number of techniques are actually available in commercial solvers.

These are the main topics covered in this introductory textbook, but there is a wealth of other techniques available. I hope I've whetted your appetite.

Chapter 2: Introduction to Linear Programming

You may recall *unconstrained optimization* from your high school years: the idea is to find the highest point (or perhaps the lowest point) on an *objective function* (see Figure 2.1). For optimization to be required, there must be more than one solution available. In Figure 2.1, any point on the function is a solution, and because the single variable is real-valued, there are an infinite number of solutions. Some kind of optimization process is then required in order to choose the very *best* solution from among those available. What is meant by *best* depends on the problem at hand: it might mean the solution that provides the most profit, or that consumes the least of some limited resource, e.g. area in computer chip design, or fuel in delivery route design.

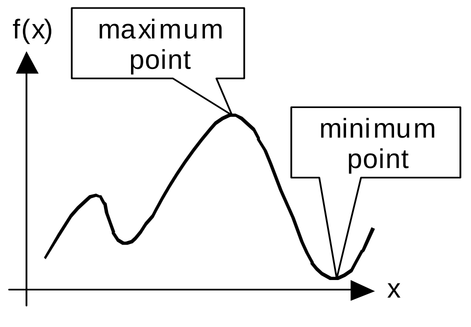


Figure 2.1: Simple unconstrained optimization.

Linear programming (LP) is the most commonly applied form of *constrained optimization*. Constrained optimization is much harder than unconstrained optimization: you still have to find the best point of the function, but now you also have to respect various constraints while doing so. For example, you must guarantee that the optimum point does not have a value above or below a prespecified limit when substituted into a given constraint function. The constraints usually relate to limited resources. The simple methods you used in high school to find peaks and valleys won't work anymore: now the best solution (the *optimum point*) may not occur at the top of a peak or at the bottom of a valley. The best solution might occur half way up a peak when a constraint prohibits movement farther up.

The main elements of any constrained optimization problem are:

- **Variables** (also called *decision variables*). The values of the variables are not known when you start the problem. The variables usually represent things that you can adjust or control, for example the rate at which to manufacture items. The goal is to find values of the variables that provide the best value of the objective function.
- **Objective function.** This is a mathematical expression that combines the variables to express your goal. It may represent profit, for example. You will be required to either maximize or minimize the objective function.
- **Constraints.** These are mathematical expressions that combine the variables to express limits on the possible solutions. For example, they may express the idea that the number of workers available to operate a particular machine is limited, or that only a certain amount of steel is available per day.

- **Variable bounds.** Only rarely are the variables in an optimization problem permitted to take on any value from minus infinity to plus infinity. Instead, the variables usually have bounds. For example, zero and 1000 might bound the production rate of widgets on a particular machine.

In *linear programming (LP)*, all of the mathematical expressions for the objective function and the constraints are linear. The *programming* in linear programming is an archaic use of the word “programming” to mean “planning”. So you might think of linear programming as “planning with linear models”. You might imagine that the restriction to linear models severely limits your ability to model real-world problems, but this isn’t so. An amazing range of problems can be modeled using linear programming, everything from airline scheduling to least-cost petroleum processing and distribution. LP is very widely used. For example, IBM estimated that in 1970, 25% of all scientific computation was devoted to linear programming.

Linear programming is by far the most widely used method of constrained optimization. The largest optimization problems in the world are LPs having millions of variables and hundreds of thousands of constraints. With recent advances in both solution algorithms and computer power, these large problems can be solved in practical amounts of time.

Of course, there are also many problems for which LP is *not* appropriate, and part of the job for this textbook is to help you decide when to use LP and the other techniques covered here, and when *not* to use them.

A Prototype Example: The Acme Bicycle Company

It’s time now to introduce you to a small example that we will be visiting numerous times throughout the book: the Acme Bicycle Company (ABC). The name follows the time-honored tradition in optimization and operations research texts of inventing bogus companies such as the “Wyndor Glass Company” (which, oddly, makes glass windows and doors), or the “Nori and Leets” Iron and Steel Company.

The Acme Bicycle Company produces two kinds of bicycles by hand: mountain bikes and street racers. Acme wishes to determine the rate at which each type of bicycle should be produced in order to maximize the profits on the sales of the bicycles. Acme assumes that it can sell all of the bicycles produced.

The physical data on the production process is available from the company engineer. A different team produces each kind of bicycle, and each team has a different maximum production rate: 2 mountain bikes per day and 3 racers per day, respectively. Producing a bicycle of either type requires the same amount of time on the metal finishing machine (a production bottleneck), and this machine can process at most a total of 4 bicycles per day, of either type. The company accountant estimates that mountain bikes are currently generating a profit of around $15 per bicycle, and racers a profit of around $10 per bicycle.

This problem is small enough to solve without using LP, just the straightforward application of common sense. To maximize profit, start by producing the maximum number possible of the higher-profit mountain bikes and use any leftover production capacity to produce racers for additional profit. This would mean a production rate of 2 mountain bikes per day, which is the limit of the mountain bike team, yet leaves spare capacity on the metal finishing machine. This remaining capacity can be used to produce 2 racers per day, which is below the capacity of the racer production team. The total profit would then be 2×\$15+2×\$10=\$50 per day.

We will be formulating and solving the Acme problem as a linear program, but there is an important lesson here: the results returned by a mathematical program should always be compared to the results predicted by common sense. If the two are in conflict, investigate. You will discover either a modeling or data error, or will learn more about the underlying process, thereby sharpening your intuition. The LP solution of the Acme problem had better turn up a daily profit of at least \$50!

The first step in formulating the ABC problem as a linear program is to identify the variables. These are the items whose values you can set or otherwise control. The Acme variables are the production rates of mountain bikes (call this x_1) and racers (call this x_2). Note any bounds on the variables:

- variable nonnegativity: $x_1 \geq 0$, $x_2 \geq 0$

Using these variables, next write the objective function:

- maximize daily profit: maximize $Z=15x_1+10x_2$ (in \$ per day)

It's a convention to represent the value of the objective function by Z.

Use the variables to write the constraints as well:

- mountain bike production limit: $x_1 \leq 2$ (in bikes per day)
- racer production limit: $x_2 \leq 3$ (in bikes per day)
- metal finishing machine production limit: $x_1+x_2 \leq 4$ (in bikes per day)

The first two constraints are normally considered variable bounds, but we will treat them as general constraints for now.

Note that it is customary to write LP constraints with all of the variables on the left hand side of the relationship, and the constant value on the right hand side (*rhs*). As in all LPs, all of the relationships (constraints and objective) are linear.

On a superficial level, you now have all that you need to apply a linear programming solver: a set of linear constraints (≥ type, ≤ type, or = type) and a linear objective, and some variable bounds. LP solvers are not hard to find: several are available for free via the World Wide Web, and an LP solver is even included in the Microsoft Excel spreadsheet software for PCs. For this reason, many people with only very limited understanding of LP are formulating and solving them. The difficulty arises when unexpected results are returned: then a deeper understanding of LP is essential.

Cornerpoints are Important

Because there are only two variables in the Acme Bicycle Company formulation, the problem can be sketched on the plane, as shown in Figure 2.2. The limiting value of each of the constraints is shown as a line. Each constraint eliminates part of the plane. For example, the vertical line labeled "x_1=2" is the limiting value of the inequality $x_1 \leq 2$ and all points to the right of the line violate the constraint (i.e. are *infeasible*). The areas eliminated by the constraints are shaded. The unshaded area represents points that are not eliminated by any constraint, and is called the *feasible region*. Points in the feasible region (which includes the bordering lines) satisfy *all* of the constraints.

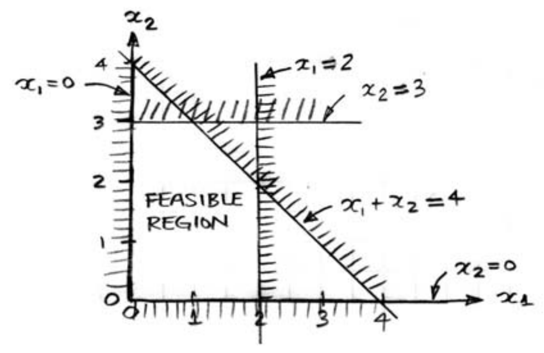


Figure 2.2: The feasible region for the Acme Bicycle Company problem.

The linear programming problem in Figure 2.2 is to find the point in the feasible region that gives the largest value of the objective function. One (silly) way to do this is to randomly choose feasible points and to calculate the value of the objective function at those points, keeping the point that gives the best value of the objective. Because there are an infinite number of points in the feasible region, this is not very effective! There is no guarantee that the best point will be found, or even that an objective function value that is close to the best possible value will be found. We need a more efficient way of searching the feasible region.

We can develop a more efficient search technique based on a couple of simple observations. First, let's plot points in the 2-dimensional $x_1 \times x_2$ plane that have the same value of the objective function. As shown in Figure 2.3, points having the same value of Z (value of the objective function) form a line. This is easy to understand if we replace Z by the specific value that we want to plot, e.g. Z=$15x_1+10x_2$ becomes the line $15x_1+10x_2=20$ plotted in Figure 2.3.

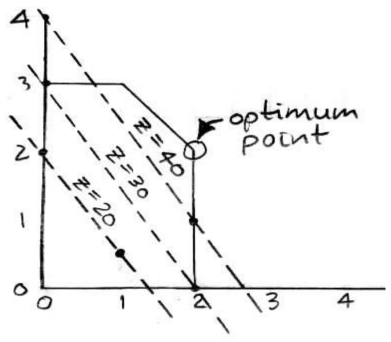


Figure 2.3: Some constant-profit lines for the ABC problem.

Figure 2.3 also shows that all of the constant-profit lines are parallel. This is because all of the constant-profit line equations differ only by the selected value of Z. If you were to calculate the slope of any constant-profit line, the Z constant disappears; hence the slopes of all of the constant-profit lines are the same. For the Acme Bicycle Company, all of the constant profit lines have the same slope, given by $dx_2/dx_1 = -15/10 = -1.5$.

The third important observation is that the value of Z is higher for the constant-profit lines towards the upper right in Figure 2.3. We will revisit this property in more detail when we cover nonlinear programming, but for now accept that this is a property of linear functions because they have a constant *gradient*.

Now we can view the linear programming problem in a different way. Picture the objective function as a constant-profit line that is floating upwards from the lower left to the upper right in Figure 2.3, increasing in value as it floats higher. Now the linear programming question is this: what is the last point in the feasible region that the objective function passes through as it floats up to infinity? From Figure 2.3 we see that the last point is (2,2) with Z=50. This is the solution to the LP: the feasible point that has the best value of the objective function! Another analogy is to imagine the objective function sinking from infinity in the upper right, decreasing in value until it first bumps into the feasible region. What is the first feasible point that it bumps into (which will give the best value of the objective function)? It is of course (2,2) with Z=50.

Here is the final and most important observation. Because lines define the feasible region, all of its external edges (or *faces*) are flat linear surfaces that join together at *cornerpoints*. Again imagine the objective function as a line sinking from infinity: where will it first bump into this feasible region defined by flat faces and cornerpoints? As you can see by inspection, the linear objective function will always first bump into the feasible region *at a cornerpoint!* This is because a cornerpoint "sticks out farthest" in the direction of the sinking objective function line, hence first contact will be at a cornerpoint, and this will define the optimum point.

In some cases, the objective function has exactly the same slope as a face of the feasible region and first contact is between the objective function and this face, as in Figure 2.4. This means that *all* of the points on that face have the same value of the objective function, and all are optimum: i.e. there are *multiple optima*. Notice, though, that if a face has first contact, then the cornerpoints of the face also have first contact. The important idea is that first contact between the objective function and the feasible region always involves at least one cornerpoint. Hence, *an optimum solution to the LP is always at a cornerpoint.*

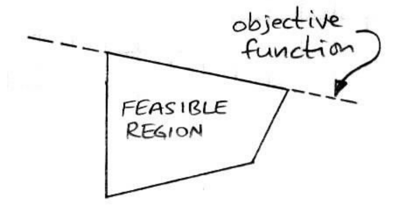


Figure 2.4: The slope of the objective function exactly matches the slope of the face of the feasible region.

This observation drastically simplifies the search for the optimum point: we need to look only at the relatively small number of cornerpoints of the feasible region instead of randomly sampling the infinite number of points on the interior of the feasible region.

This fact underlies the *simplex method* of linear programming, which we shall begin to address in the next chapter. For now just observe in Fig. 2.2 that there are only five *feasible cornerpoints* that need to be visited to find an optimum solution to the Acme Bicycle Company LP.

It is also easy to identify when there are multiple optima just by looking at the feasible cornerpoints. In two dimensions, when two feasible cornerpoints have the same optimum value of the objective function, then all of the points on the line segment joining the two cornerpoints have the same optimum value. It's worth knowing this because one of the intermediate optimum points may be preferable to the cornerpoints for nonquantifiable reasons. It is possible to have three or more feasible cornerpoints with the same optimum value of the objective function in a three dimensional problem. Imagine, for example, that the feasible region is defined by a three-dimensional tetrahedron and that the slope of the objective function plane is exactly equal to the slope of one of the faces of the tetrahedron. Now all three feasible cornerpoints of the triangular face of the tetrahedron, and all of the points on the face of the tetrahedron, will have the same optimum value of the objective function.

The Underlying Assumptions in Linear Programming

The inescapable underlying assumption that is made in modeling the real world via linear programming is that a linear model is suitable. Models constructed solely from linear relationships have certain limitations. The most obvious is that some real-world phenomena are poorly modeled by lines. Nonlinear relationships such as curves or step-functions may be needed instead. If such nonlinear or discontinuous relationships are not adequately approximated by linear relationships, then you must use a technique other than linear programming.

The two linear properties of *additivity* and *proportionality* preclude curves or step-functions. The additivity property prohibits cross-product terms, e.g. $5x_1x_2$, which might represent interaction effects between the variables. For example, Acme may discover that ordering materials for both bicycles together from the same supplier lowers costs, but this effect cannot be modelled using a linear relationship.

The proportionality property requires that the value of each term in the linear function is strictly proportional to the value of the variable in the term. For example, the objective function cost of using a certain variable is always directly proportional to the level of use of the variable: it is not possible to include a start-up cost. In the Acme model, the proportionality assumption is violated if, for example, the production efficiency improves significantly as the rate of production increases.

Linear programming assumes that the variables are real-valued, meaning that they can take on fractional values. In the Acme problem we are trying to determine the *rate* of production, which can take on fractional values, e.g. produce mountain bicycles at the rate of 1.5 per day. Fractional values are not suitable in some problems, such as determining the number of people to staff a set of restaurants or the number of ships to purchase. Where at least one variable is restricted to taking on an integer value, then you must use the methods of *integer programming*, which are covered in a later chapter. For now, note that it is *not* acceptable to treat integer problems as linear problems and then just round the results to the closest integer. This may yield an infeasible solution. The true optimum in an integer programming problem can be very far away from the approximate solution obtained by linear programming.

A weakness common to all of mathematical programming is the assumption that the input data are perfectly accurate. You are assuming that the objective function coefficients (profit per bicycle for Acme), the constraint coefficients, and the constraint right hand sides (e.g. maximum team daily production of mountain bikes) are all correct. In the real world, these numbers are seldom known with accuracy. For example, how does Acme really know how much profit it makes per bicycle of either type? In large companies such a number is generally produced by the Accounting department, which uses data about average amount of material used in each bicycle, average price paid for the materials, average worker wages, yearly depreciation estimates on machines, average selling prices, etc., to *estimate* the "profit per mountain bike sold". The emphasis is on "estimate".

So how useful is the optimum result produced by the mathematical program if the input data is of poor quality? It can be extremely useful, but you have to be careful. First and foremost, don't treat the output result as if it is "the" answer: you might arrive at quite different results just by using slightly different estimates of the input parameters. For example, will you get a different result if the profit per mountain bike is estimated at $14 instead of the present $15? This is where *sensitivity analysis* is applied: using various tests to determine how sensitive the optimum result is to small changes in the values of the input parameters. It turns out that Acme should still make two each of the racers and mountain bikes per day even if the profit per mountain bike is $1 lower than estimated, but of course the total rate of profit generation will be lower. If knowing the total profit generation rate is crucial to Acme, then it is worthwhile to analyze various scenarios of profit per bicycle.

Formulation Practice: the Smalltime Mutual Funds Company

Formulating LPs well takes practice. In a classroom situation you will often know in advance that you are formulating a linear program. In contrast, in the real world you normally *don't* know the type of the problem when you begin studying it, and that makes formulation much more difficult. The only way to improve your skills in formulation is practice, practice, practice. So here's another example. Try to formulate it before looking at the solution as give below.

You are the investments manager for the Smalltime Mutual Funds Company, and are trying to determine how to invest a pool of $14 million released by cashing out some of the stock investments. Table 2.1 summarizes the information that you have about a set of five possible investments.

To be as conservative as possible, you assume that in the event of a loss by the investment, you lose all of your money. This is a fairly serious assumption, since most mutual fund investments are likely to lose some but not all of their value. On the other hand, you also assume that if there is not a loss, then the investment will grow by the growth rate shown.

For policy reasons, there are limits on how you can invest the money. You must allocate at least 35% of the total funds available to the balanced and bond investments. Of all the money put into equity, special equity and foreign investments, at least half must be in the equity investment. Finally, the expected lost capital must be less than 10%. Of course, your overall objective is to maximize the return on the original pool of money.

This is a textbook problem, so the data is stated much more succinctly and clearly than in real world problems, which are plagued by misleading, hidden, and spurious information. Still, extracting an LP formulation from even a textbook word problem can be harder than it seems. You can test yourself by trying to answer the questions posed in the next few paragraphs before reading the answers.

type of investment	annual growth rate	probability of loss
equity	0.15	0.18
special equity	0.21	0.31
balanced	0.11	0.09
foreign	0.19	0.19
bond	0.08	0.03

Table 2.1: Investments available to the Smalltime Mutual Funds Company.

The first thing to do in an LP formulation is to identify the decision variables. Ask yourself what it is that you can control in this problem, what quantities do you need to find values for? *What are the decision variables?*

The most straightforward formulation of this problem chooses variables representing the amount of money put into each investment:

- x_1: millions of dollars put into equity investment,
- x_2: millions of dollars put into special equity investment,
- x_3: millions of dollars put into balanced investment,
- x_4: millions of dollars put into foreign investment,
- x_5: millions of dollars put into bond investment.

It is also possible to formulate this problem using variables representing the *fraction* of the total money to be put into each kind of investment. It is an awkward approach, but the results are the same in the end.

Now that you have selected variables, the second question is: *what is the objective function?* You are to “maximize the return on the money invested”, but what does this mean? Since some money is gained from interest on the investments and some money is lost, let’s say the net return is:

(expected growth from good investments) *minus* (expected losses from bad investments).

If you are not sure if this is what the managers at Smalltime mean by “maximizing the return on money invested”, then *make sure you ask!* Effective mathematical programming is not just about number-crunching, it’s about crunching the *right* numbers.

Assuming we have the correct idea about the objective, let’s now write it out in terms of the decision variables:

maximize $Z = 0.15(1-.18)x_1 + 0.21(1-.31)x_2 + 0.11(1-.09)x_3 + 0.19(1-.19)x_4 +$
$0.08(1-.03)x_5 - 0.18x_1 - 0.31x_2 - 0.09x_3 - 0.19x_4 - 0.03x_5$

The pattern is simple: the first five terms represent the income due to annual growth on the investments that do not lose money, and the second five terms represent the capital losses on the investments that lose money (remember that we assume you also get no interest on losing investments).

Now we add the constraints. Scan the problem description. *Can you identify all of the constraints?* There are five:

1. limit on proportion of total funds put into balanced and bond investments,
2. limit on proportion of funds in the equity, special equity, and foreign investments that goes into equity funds,
3. limit on expected capital losses,

And the two most frequently forgotten by students:

4. limit on total funds available,
5. nonnegativity constraints on the variables.

Now we can write out these constraints.

Limit on proportion of total funds put into balanced and bond investments:

$$(x_3+x_5)/14 \geq 0.35 \Rightarrow x_3+x_5 \geq 4.9$$

Limit on proportion of funds in the equity, special equity, and foreign investments that goes into equity funds:

$$x_1/(x_1+x_2+x_4) \geq 0.5 \Rightarrow -0.5x_1 + 0.5x_2 + 0.5x_4 \leq 0$$

Limit on expected capital losses. We will interpret this to mean (expected capital loss)/(total capital invested), so:

$$(0.18x_1 + 0.31x_2 + 0.09x_3 + 0.19x_4 + 0.03x_5)/(x_1 + x_2 + x_3 + x_4 + x_5) \leq 0.1$$
$$\Downarrow$$
$$0.08x_1 + 0.21x_2 - 0.01x_3 + 0.09x_4 - 0.07x_5 \leq 0$$

Limit on total funds available (remember that the variables are in units of millions of dollars):

$$(x_1 + x_2 + x_3 + x_4 + x_5) \leq 14$$

Nonnegativity of the variables:

$$x_1, x_2, x_3, x_4, x_5 \geq 0$$

Are the assumptions inherent in any LP model appropriate for *this* model? The additivity and proportionality assumptions are likely correct here. Strictly speaking, the restriction to real numbers does not hold because you can't subdivide a penny, but when dealing with very large numbers, this rounding to the nearest penny is negligible. The worst assumption here is that the parameters are known for certain. Both the annual growth rate and the probability of loss are educated guesses at best. Since $14 million depends on this decision, you should very carefully examine how changes in those numbers affect your solution. You will need to do some *sensitivity analysis*, a topic addressed later in this book.

Another unrealistic assumption is that you lose all of your capital if the mutual fund loses value. There is an old saying that "the map is not the territory", or to paraphrase for applied optimization "the model is not the real world". You must be aware of the losses in accuracy inherent in the assumptions that you make during modeling. Always check any assumptions with the client to make sure they are appropriate for the task at hand.

The Standard Form LP

Linear programs can have objective functions that are to be maximized or minimized, constraints that are of three types (≤, ≥, =), and variables that have upper and lower bounds. An important subset of the possible LPs is the **standard form LP**. A standard form LP has these characteristics:

- the objective function must be maximized,
- all constraints are ≤ type,
- all constraint right hand sides are nonnegative,
- all variables are restricted to nonnegativity.

A standard form LP is the simplest form of linear program, so we will begin our study of how to solve LPs using them. The most significant property of a standard form LP is that the origin (all variables set to zero) is always a feasible cornerpoint. This is because all standard form LPs have the kind of shape illustrated in Figure 2.5. Knowing this initial feasible cornerpoint greatly simplifies the search for the optimum. After studying how to solve standard form LPs, we will return to the problem of optimizing LPs that are not in standard form.

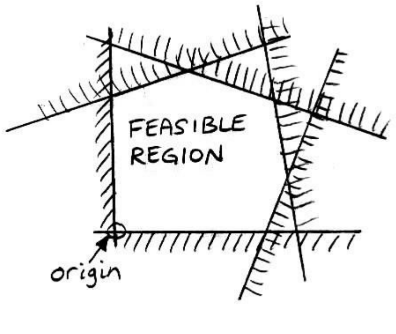


Figure 2.5: The origin is always a feasible cornerpoint in a standard form LP.

In an algebraic representation, a standard form LP having *m* functional constraints and *n* variables looks like this:

- **Objective function:** maximize $Z = c_1x_1 + c_2x_2 + \ldots + c_nx_n$
  where the c_j, the coefficients in the objective function, represent the increase or decrease in Z, the objective function value, per unit increase in x_j. For the Acme Bicycle Company, Z is the daily profit, and the c_i are the contributions to profit made by the mountain bikes (c_1) and the racers (c_2).
- m **functional constraints**, so called because they take a functional form:

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_{1n} \le b_1 \\ &a_{12}x_1 + a_{22}x_2 + \ldots + a_{2n}x_{2n} \le b_2 \\ &\qquad\qquad\vdots \\ &a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \le b_\mathrm{m} \end{aligned}$$

  where the b_i are the resource limits, and the a_{ij} are the coefficients of the functional constraint equations, expressing the usage resource i consumed by activity j. For the Acme Bicycle Company, the b_i are limits on mountain bike production (b_1), on racer production (b_2) and on the metal finishing machine (b_3).
- n **nonnegativity constraints**: $x_1 \ge 0$, $x_2 \ge 0$, …, $x_\mathrm{n} \ge 0$.

Don’t forget to explicitly include the nonnegativity constraints when writing out a problem formulation. You don’t want to allow negative values for the variables accidentally: in the Acme example, this would mean that you could perhaps make money by disassembling bicycles and selling the materials back to the suppliers! Fortunately, most commercial LP solvers will assume nonnegativity if you don’t mention it, but while you are learning the subject, show that you have considered the variable bounds by explicitly writing them out. There are some formulations in which negative variables are allowed, for example when the variable represents change from the current level, as in the level of water in a reservoir.

In Practice

It is easy to get started using linear programming on real problems. Modern LP solvers are generally coupled with a user-friendly front-end which permits easy input of the model and browsing of the results. The solvers use a variety of input formats, so choose a solver that includes an input format that suits the way you work:

- A straight algebraic representation of the problem, with each constraint written out explicitly, such as in this book.
- A spreadsheet representation, generally with columns for the variables and rows for the constraints.
- An algebraic language that allows use of summations and indices to write the model very compactly. One statement in the algebraic language may generate numerous individual constraints for submission to the solver.
- A proprietary input format.

Algebraic modeling languages are the best developed input format, and are most used in practice. The ability to use summations and indices means that large industrial-scale models can be written in a concise form that is easier to debug. Many of the languages permit direct connection to databases, so the details of very large models can be easily changed without altering the overall form of the problem. For example, you may have a very stable overall model of your worldwide petroleum refining system, but need to change prices and demand information on a regular basis. This is straightforward to do with an algebraic modeling language.

Algebraic modeling languages also act as generic front-ends to solvers. In other words, you can write the model in the modeling language, and then choose from among the solvers that are attached to the modeling system. You might want a different solver because it uses a different algorithm, or because your model has gradually grown so large that a more powerful solver is required. With modeling languages, these kinds of changes are not a problem.

Web Resources

The following web page has links to web pages where you can find free student-edition linear programming solvers, on-line solvers, and other helpful resources:

http://www.sce.carleton.ca/faculty/chinneck/StudentOR.html

Chapter 3: Towards the Simplex Method for Efficient Solution of Linear Programs

The simplex method, invented by George Dantzig in 1947, is the basic workhorse for solving linear programs, even today. While there have been many refinements to the method, especially to take advantage of computer implementations, the essential elements are the same as they were when the method was invented. During the Second World War, calculations were carried out on manual hand calculators by rooms full of clerks. Fortunately today, these calculations are done much more rapidly and accurately by digital computers.

The simplex method has a poor theoretical efficiency, but actually performs extremely well in practice. It is the method of choice for a broad range of small to large problems, but the newer *interior-point methods* are preferred for extremely large problems. The simplex method also has the advantage of making sensitivity analysis easier.

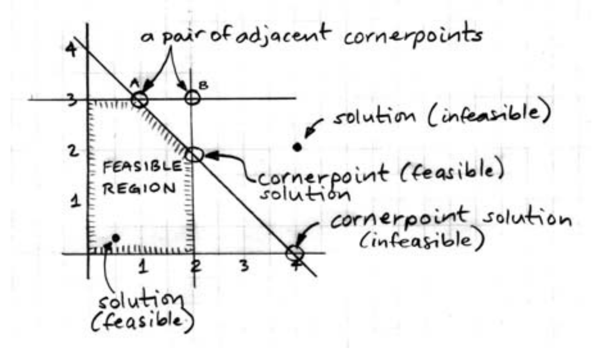
The goal in this chapter is to give you an understanding of the basic mechanics of the method, and to show you why it works. This will equip you to deal with the inevitable unexpected results when you are solving linear programs in practice later in your career.

Some Definitions

Here is some fundamental vocabulary for talking about linear programs when they are expressed in a graphical manner (refer to Figure 3.1 for examples):

- **solution:** any point in the variable space. Could be feasible (satisfies all constraints), or infeasible (violates at least one constraint).

- **cornerpoint solution:** anywhere two or more constraints intersect. Again, such a point might be feasible or infeasible.



a pair of adjacent cornerpoints
A
B
solution (infeasible)
cornerpoint (feasible) solution
cornerpoint solution (infeasible)
FEASIBLE REGION
solution (feasible)
4
3
2
1
1
2
3
4

Figure 3.1: Vocabulary for graphical representations of LPs.

- **feasible cornerpoint solution:** a cornerpoint solution that is feasible. As we will see later, the simplex method is *very* interested in feasible cornerpoint solutions.

- **adjacent cornerpoint solutions:** two cornerpoint solutions that are connected by a single constraint line segment are adjacent cornerpoint solutions. Again, there are no assumptions about feasibility.

Key Properties of Linear Programs

Here are the three key properties of linear programs that drive the design of the simplex method:

1. **The optimum point is always at a feasible cornerpoint.** As we saw previously (see Figure 2.3), this is a by-product of the fact that all of the constraints and the objective function are linear. Remember that there can also be multiple optima, but this will happen only when at least two of the optima are adjacent cornerpoint feasible solutions.
2. **If a cornerpoint feasible solution has an objective function value that is better than or equal to all adjacent cornerpoint feasible solutions, then it is optimal.** Again, this is a by-product of modeling with lines.
3. **There are a finite number of cornerpoint feasible solutions.** This means that any method which concentrates on looking only at cornerpoint feasible solutions, as the simplex method does, will eventually terminate. Good news!

Think about point 1 for a moment: is it possible to have 3 or more cornerpoints having the same optimum value? Sure, but you have to go to higher dimensions (i.e. problems with more variables). Imagine a problem in 3 dimensions in which the feasible region resembles a cube. Now imagine that the “slope” of the objective function exactly matches the “slope” of one of the faces of the cube. In this case, there will be 4 cornerpoints all having the exact same optimum value of the objective function!

These properties have tremendous practical impact. Property 1 means that you only need to look at cornerpoints, rather than at the infinite number of points in the feasible region, which is a tremendous boost in efficiency. Property 2 means that you can easily recognize when you have found the optimum point, so you don’t have to look at all of the feasible cornerpoints, another boost in efficiency. Finally, property 3 means that the method is guaranteed to terminate, so you *will* get an answer.

We now have enough information to provide a bird’s-eye view of the simplex method. It has two main phases:

1. **Phase 1 (start-up):** find *any* cornerpoint feasible solution. The reason why we start our study of linear programming with standard form LPs is that the origin (the (0,0,…,0) point) is *always* a cornerpoint feasible solution in a standard form LP, so phase 1 is simple. Phase 1 is more complicated for non-standard form LPs, and requires a special method, which we will cover later.
2. **Phase 2 (iterate):** repeatedly move to a better adjacent cornerpoint feasible solution until no further better adjacent cornerpoint feasible solutions can be found. This final cornerpoint feasible solution defines the optimum point. Note that there could be other adjacent cornerpoint feasible solutions with the *same* optimum value.

As shown in Figure 3.2, the sequence of feasible cornerpoints visited in the Acme Bicycle Company problem is as follows:

1. *Phase 1:* choose the origin (0,0). The value of the objective function at that point is zero, expressed as $Z(0,0)=0$.
2. *Iteration 1:* move from (0,0) to (2,0). $Z(2,0)=30$, an improvement.
3. *Iteration 2:* move from (2,0) to (2,2). $Z(2,2)=50$, an improvement.
4. *Stop:* the only feasible cornerpoint not yet visited that is adjacent to (2,2) is at (1,3), and $Z(1,3)=45$. So (2,2) with $Z=50$ is better than both of the adjacent feasible cornerpoints: $Z(2,0)=30$ and $Z(1,3)=45$, so (2,2) is the optimum point, and the best achievable value of the objective function is 50.

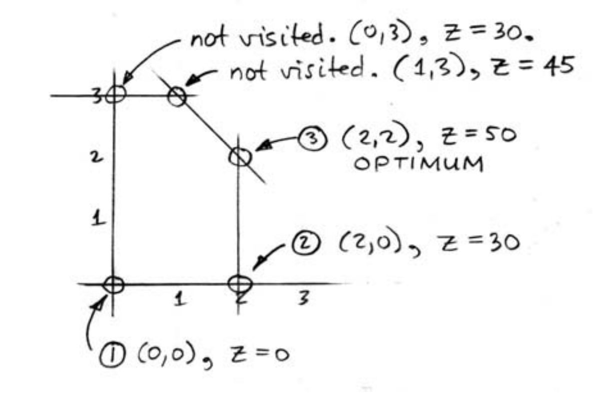


Figure 3.2: Visiting cornerpoints in the Acme Bicycle Company problem.

This means that the Acme Bicycle Company should produce mountain bikes at the rate of 2 per day and racers at the rate of 2 per day to achieve a maximum profit rate of $50 per day.

As we will see later, the simplex method does not actually need to visit all of the feasible cornerpoints adjacent to the optimum point. There is a simple way to recognize that there are no better adjacent feasible cornerpoints left to visit.

Finding Cornerpoints Algebraically

Graphical representations of linear programming problems are only used for teaching the principles; real problems have hundreds, thousands, even millions of variables. For problems at that scale, we need an algebraic way to find the cornerpoints. For standard form LPs, the answer lies in converting the inequalities to equations, and then solving for the intersection of a subset of the equations. There are well-developed methods for finding the intersection of linear equations, such as Gaussian elimination.

Note that we must solve for the intersection of a *subset* of the equations because in the usual case not all of the equations derived from the original inequalities can hold simultaneously. So, in addition to converting the inequalities to equations, we need a way to keep track of which of the equalities are currently selected, or *active*. The resolution of both difficulties lies in the addition of *slack variables* to the inequalities to convert them to equations. For example:

$x_1 \leq 2$ becomes $x_1 + s_1 = 2$ when the nonnegative slack variable s_1 is added.

Slack variables are so named because they "take up the slack" between the left hand side of the equation (in this case just x_1) and the right hand side value. The nonnegativity of the slack variable is essential to this role.

Converting the Acme Bicycle Company to equality format in this way yields the LP shown in Figure 3.3. The problem, originally in two dimensions (x_1 and x_2), is now a problem in five dimensions (x_1, x_2, s_1, s_2, s_3). Note that the slack variables take on positive values only when the constraint that they appear in is *not* active.

	original variables		slack variables			
$Z =$	$15x_1$	$+ 10x_2$				
	x_1		$+ s_1$			$= 2$
		x_2		$+ s_2$		$= 3$
	x_1	$+ x_2$			$+ s_3$	$= 4$

$$x_1, x_2, s_1, s_2, s_3 \geq 0$$

Figure 3.3: Equality version of the Acme Bicycle Company problem.

There is some special terminology when working with the algebraic, equation-converted version of the original LP model, as follows:

- **augmented solution:** the values for all of the variables are given, including both the original variables and the slacks. For example, at the optimum solution to the Acme Bicycle Company problem, the augmented solution is $(x_1,x_2,s_1,s_2,s_3) = (2,2,0,1,0)$.
- **basic solution:** an augmented *cornerpoint* solution (could be feasible or infeasible). In the ABC problem, (2,3,0,0,-1) is a basic solution that happens to be infeasible.
- **basic feasible solution:** an augmented cornerpoint *feasible* solution. In the ABC model, (0,3,2,0,1) is a basic feasible solution.

As you might imagine, the simplex method is mostly concerned with basic feasible solutions.

Setting the Values of the Variables

Somehow we need to set the values of the variables, and this should be done in such a way that we arrive at a feasible cornerpoint, or basic feasible solution. Consider the ABC problem in which we have 5 variables after conversion to equation format, but only 3 constraints. This must mean that we can set the value of 2 of the variables arbitrarily, and then calculate the values of the other 3 using the equations.

The number of variables whose values can be set arbitrarily is known as the *number of degrees of freedom (df)* of a problem. In general,

df = (number of variables in equation format) – (number of independent equations)

As we will see later, the simplex method will automatically set the values of *df* of the variables, and then solve for the values of the others. In fact, simplex will set those *df* variables to a value of zero. Why zero? Because this implies that the corresponding constraint is *active*, i.e. right on it's limiting value and actively preventing the solution process from venturing into the infeasible zone. Let's see if this is true for the ABC problem. Considering the equation form of the problem as in Figure 3.3:

- x_1=0 implies that you are on the limiting value of the $x_1 \geq 0$ constraint.
- x_2=0 implies that you are on the limiting value of the $x_2 \geq 0$ constraint.
- s_1=0 implies that you are on the line x1=2.
- s_2=0 implies that you on the line x_2=3.
- s_3=0 implies that you are on the line x_1+x_2=4.

Because *df*=2 in the ABC company, that means that simplex will set the values of two of the 5 variables to zero. In other words, this will make two of the constraints active, and it will thereby define a cornerpoint where these two constraints cross. Figure 3.4 shows that the selection of which two variables are set to zero can define a feasible, or perhaps an infeasible cornerpoint.

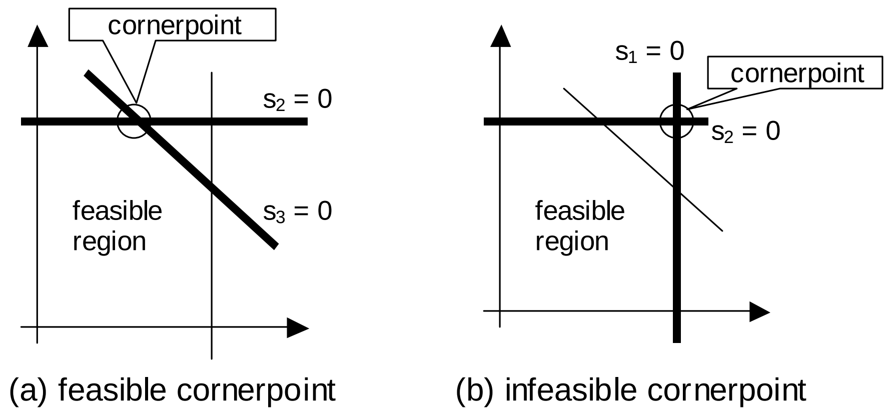


Figure 3.4: Setting variables to zero defines a cornerpoint.

There are two final items of terminology at this point:

- **nonbasic variable:** a variable currently set to zero by the simplex method.
- **basic variable:** a variable that is *not* currently set to zero by the simplex method. Basic variables normally have nonzero values (positive in the case of standard form LPs), but they can also be zero in special circumstances. This means that you can't reliably tell which variables are zero and which are nonzero just by looking at their current values.
- **a basis:** as simplex proceeds, the variables are always assigned to the basic set or the nonbasic set. The current assignment of variables is called the basis.

As we develop the simplex method, we will be working intensively with this idea of variables being set to zero and thereby activating constraints. You should memorize the following mantra, and chant it to yourself or work it into casual conversations at every opportunity:

Nonbasic, variable set to zero, corresponding constraint is active.

The simplex method is all about deciding what the current basis is, i.e. which variables are currently basic and which are currently nonbasic. In fact, if you were exceedingly lucky, you could solve LPs directly just by guessing which variables should be basic and which nonbasic. This then defines the active set of constraints, so you can solve for the intersection point of the active constraints to find the optimum point and the value of the objective function at that point. But there are some pitfalls if you guess incorrectly! Figure 3.4 (b) shows how you could select a basis that defines an infeasible cornerpoint. Figure 3.5 shows how you might even select a basis that does not define a cornerpoint at all.

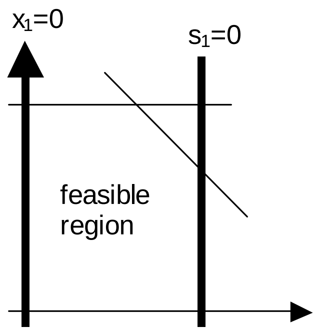


Figure 3.5: A basis that does <u>not</u> define a cornerpoint.

Moving to a Better Basic Feasible Solution

Given that phase 1 provides us with a basic feasible solution to start from, the simplex method then just needs to move to a better basic feasible solution, and to continue doing this until the stopping conditions (no better adjacent cornerpoint feasible solution) are met. As we've seen above, choosing the next basis cannot be done at random: you might choose an infeasible basis, a worse basis, or a basis that does not define a cornerpoint at all.

It turns out that if you are at a basic feasible solution (feasible cornerpoint), then the easiest basic feasible solution to find next will be adjacent. This is partly due to the following property of adjacent cornerpoints:

- in two adjacent cornerpoints, the nonbasic sets will be identical, except for one member
- in two adjacent cornerpoints, the basic sets will be identical, except for one member.

For example, consider the two adjacent cornerpoints labeled in Figure 3.1 (these are the same cornerpoints shown in Figure 3.4):

- point A: nonbasic set = $\{s_2, s_3\}$, basic set = $\{x_1, x_2, s_1\}$
- point B: nonbasic set = $\{s_1, s_2\}$, basic set = $\{x_1, x_2, s_3\}$

This seems to imply that we can move from one cornerpoint to the next simply by swapping a pair of variables between the basic and the nonbasic sets. Unfortunately, this condition is necessary, but *not* sufficient for adjacency. For example, the pair of cornerpoints at (0,4) and at (4,0) also has this property, but those cornerpoints are not adjacent. So there are three conditions that must be taken care of while moving from one cornerpoint to the next:

- the cornerpoints must be adjacent,
- the cornerpoints must be feasible,
- the new cornerpoint must have a better value of the objective function than the current cornerpoint.

The simplex method has an impressively clever yet simple way of making sure that all three conditions are satisfied. There are two steps:

1. Determine which nonbasic variable (remember! nonbasic variables are set to zero) will increase the objective function value most swiftly if allowed to take on a positive value. Move this variable from the nonbasic set to the basic set. This is called the *entering basic variable* because it is entering the basic set.
2. Allow the entering basic variable to increase only until one of the basic variables is forced to a value of zero. Move the variable that is forced to zero from the basic set to the nonbasic set, where it will retain its value of zero. This is called the *leaving basic variable* because it is leaving the basic set.

How do we determine the variable that most swiftly increases the value of the objective function in step 1? At the origin, this is simply done by looking at the objective function. For the ABC problem, $Z = 15x_1 + 10x_2$, so it is immediately obvious that x_1 provides the swiftest rate of increase in Z, because it increases Z by 15 per unit increase in x_1, where x_2 increases Z by only 10 per unit increase in x_2. So if we are currently at the basic feasible solution at the origin point (0,0), then x_1 is chosen as the entering basic variable. At other basic feasible solutions we will work with rewritten versions of the objective function, but the basic principle is the same.

This first step does two things. First, moving a variable out of the nonbasic set inactivates one of the equations (remember! nonbasic – set to zero – constraint active). Second, because that variable is going to the basic set, we know that it is now allowed to increase in value, so this gives us a direction in which to move. Figure 3.6 shows the effect of starting at the origin and choosing x_1 as the entering basic variable. The dashed line indicates the position of the x_1=0 constraint, which has just been released.

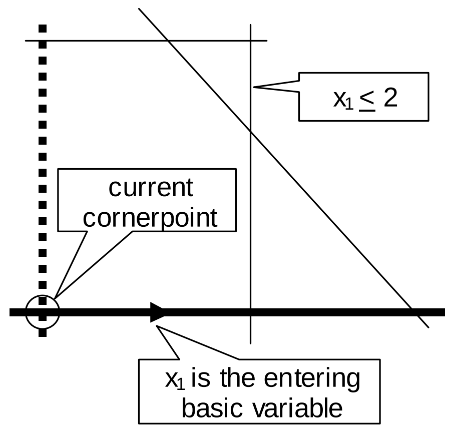


Figure 3.6: Choosing x_1 as the entering basic variable.

Step 2 then tells us when to *stop* increasing the value of the entering basic variable. We can see in Figure 3.6 that the correct place to stop increasing x_1 is when we bump into the limiting value of the $x_1 \leq 2$ constraint, otherwise we will move out of the feasible region. But how do we algebraically detect that we have reached the point at which x_1=2, i.e. the point at which the $x_1 \leq 2$ constraint becomes active? The answer is that the variable associated with the equation form of $x_1 \leq 2$ is driven to zero before any other variable.

Look at Figure 3.6 again. At the origin, the situation is this:

- basic variables: s_1, s_2, s_3
- nonbasic variable: x_2
- entering basic variable: x_1

Let us consider the effect of increasing the entering basic variable value on each of the constraint equations, as in the Table 3.1. Basic variables are underlined in the table. Note that there is exactly one basic variable in each of the constraint equations. Its value at the origin is shown.

basic variable	**constraint equation**	**bound on increase in x_1**
$\underline{s_1} = 2$	$x_1+\underline{s_1} = 2$	$x_1 \leq 2$
$\underline{s_2} = 3$	$x_2+\underline{s_2} = 3$	no limit
$\underline{s_3} = 4$	$x_1+x_2+\underline{s_3} = 4$	$x_1 \leq 4$

Table 3.1: Finding the leaving basic variable.

Considering the first equation in Table 3.1, you can see that as x_1 increases, s_1 decreases to keep the equation satisfied. Because s_1 must be nonnegative, x_1 can only increase until s_1 becomes zero. s_1 reaches a value of zero when x_1 reaches a value of 2. Because s_1 is the only basic variable in that equation, it is the only one whose value you have to worry about. Similarly, s_3 is the only basic variable in the third constraint equation, so it is the only variable that can compensate for an increase in the value of x_1 by decreasing its own value (x_2 is nonbasic, and remember! nonbasic variables are set to zero). s_3 reaches a value of zero when x_1 reaches a value of 4. Because x_1 does not even appear in the second constraint equation, that constraint places no limit on the increase in x_1.

Table 3.1 shows that it is the first constraint equation (based on $x_1 \leq 2$) that most limits the increase in x_1. When the basic variable in that equation (s_1) is driven to zero, the constraint associated with s_1 ($x_1 \leq 2$) becomes active. As you can see in Figure 3.6, this is just the constraint to activate to prevent straying out of the feasible region. Hence s_1 is chosen as the leaving basic variable.

At the new cornerpoint defined by the intersection of the limiting values of the two constraints $x_2 \geq 0$ and $x_1 \leq 2$, i.e. the point (2,0), the basis is this:

- basic variables: x_1, s_2, s_3
- nonbasic variable: x_2, s_1

Compared to the basis at the origin point, this represents a swap of x_1 and s_1 between the basic and nonbasic sets.

Finding the leaving basic variable via the method demonstrated in Table 3.1, as required in step 2, depends on having exactly one basic variable per constraint equation, and the coefficient of the basic variable being exactly 1. As we will see later, the simplex method makes sure that these conditions are met. Once this condition is met, we need only to look at the value of the following ratio to find the limiting value placed on the entering basic variable by a particular constraint equation:

$$\frac{\text{right hand side value of constraint equation}}{\text{coefficient of entering basic variable in the equation}}$$

Because we are always looking for the most limiting bound on the entering basic variable, we need the *smallest* value of this ratio, so the process of finding the leaving basic variable in this manner is called the *minimum ratio test.* The coefficients of the entering basic variable will not always have a value of 1 as they do in Table 3.1: that's an artifact of this simple example.

There are two special cases of the minimum ratio test, as illustrated in Figure 3.7:

- **if the coefficient of the entering basic variable is zero:** this means that the constraint in question does not intersect with the still-active constraints represented by the remaining nonbasic variables, so it places no limit on the increase in the entering basic variable.
- **if the coefficient of the entering basic variable is negative:** this means that the constraint in question does intersect with the still-active constraints, but the direction of increase of the entering basic variable is *away* from the intersection point. Hence the constraint in question places no limit on the increase in the entering basic variable.

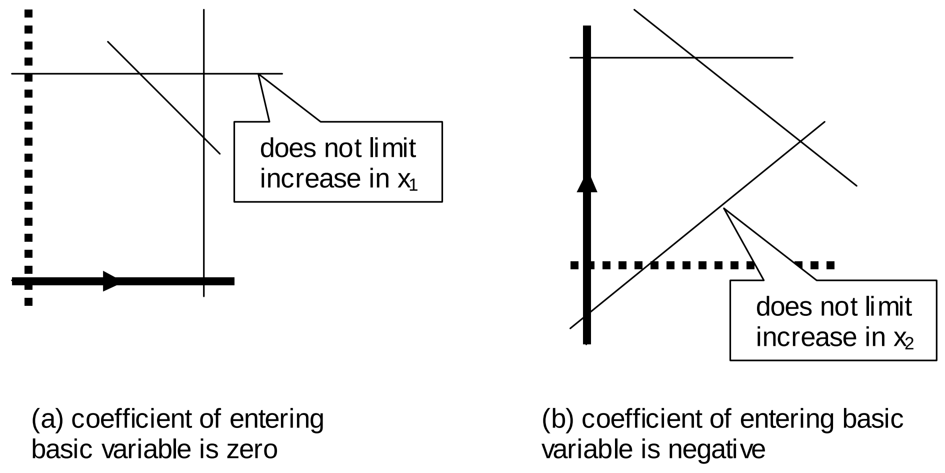

does not limit increase in x_1
does not limit increase in x_2
(a) coefficient of entering basic variable is zero
(b) coefficient of entering basic variable is negative

Figure 3.7: Why zero or negative coefficients of the entering basic variable do not limit its increase.

Finding the New Basic Feasible Solution

Now that the new basis is known, how do you actually find the point associated with the next basic feasible solution? A possible, but inefficient, method is to set the nonbasic variables to zero and then solve the remaining $m \times m$ system of linear equations, perhaps by Gaussian elimination.

A much more efficient method is to *update* the current set of equations using just a little bit of Gaussian elimination. This step can put the equations back into the format needed for applying both the test for selecting the entering basic variable (objective function must be rewritten), and the minimum ratio test for selecting the leaving basic variable (constraint equations must have exactly one basic variable each, and the coefficient of that basic variable must be 1).

We will hear more about this in the next chapter, which introduces the simplex tableau, which is a way of formalizing the steps in the simplex method.

A related question is this: how do we know when to stop iterating? This happens when we are unable to find an entering basic variable. In other words, in the updated version of the objective function, there is *no* nonbasic variable that, if allowed to become positive, would increase the value of Z. This means that we don't actually need to visit the next cornerpoint and see that the value of the objective function has worsened: we will already know that there is no improving direction to go in! Details follow in the next chapter.

Chapter 4: The Mechanics of the Simplex Method

The simplex method is a remarkably simple and elegant algorithmic engine for solving linear programs. In this chapter we will examine the internal mechanics of the simplex method as formalized in the *simplex tableau*, a table representation of the basis at any cornerpoint. The tableau contains all the information that is needed to decide on the exchange of variables that drives the movement between cornerpoints as the simplex method advances. Using the tableau, you can solve small LPs by hand, though this is tedious and error-prone. More importantly, the tableau calculations are similar to the calculations carried out by computer implementations of the simplex method. You need a good grasp of the tableau calculations to understand how the computer does its work on large LPs, how the computer calculations can go wrong, and why certain modifications to the basic method permit faster and larger solutions by computer.

Let's review what we know so far about the simplex method by presenting a high level view of the algorithm, as in Figure 4.1.

Phase 1: Find an initial cornerpoint feasible solution (basic feasible solution). If none is found, then the model is infeasible, so exit.

Note: If dealing with a standard form LP, the origin is always a basic feasible solution, so you can always start there. If it is not a standard form LP, then you must use a special phase 1 procedure that we will study later.

Phase 2: Iterate until the stopping conditions are met.

.2.1 *Are we optimal yet?* Look at the current version of the objective function to see if an entering basic variable is available. If none is available, then exit with the current basic feasible solution as the optimum solution.

.2.2 *Select entering basic variable*: choose the nonbasic variable that gives the fastest rate of increase in the objective function value.

2.3 *Select the leaving basic variable* by applying the Minimum Ratio Test.

2.4 *Update the equations* to reflect the new basic feasible solution.

2.5 Go to Step 2.1.

Figure 4.1: Overview of the simplex method.

The Simplex Tableau

The Acme Bicycle Company problem is a standard form LP, so we know that the origin is a basic feasible solution (feasible cornerpoint). Here is the simplex tableau for the basic feasible solution for ABC at the origin:

basic variable	eqn. no.	Z	x_1	x_2	s_1	s_2	s_3	RHS	MRT
Z	0	1	-15	-10	0	0	0	0	*never*
s_1	1	0	1	0	1	0	0	2	
s_2	2	0	0	1	0	1	0	3	
s_3	3	0	1	1	0	0	1	4	

Tableau 4.1: ABC at the origin.

The tableau is just a way of writing the equation versions of the objective function and the constraints in a nice aligned manner. The second column, titled *eqn. no.*, gives a label to each of the equations. Equation 0 is the objective function, and equations 1-3 are the three constraint equations. The *RHS* column is the right hand side of the equation, where the constant goes. For example, equation 3 could be written out in full as:

$$0Z + 1x_1 + 1x_2 + 0s_1 + 0s_2 + 1s_3 = 4$$

or more simply as:

$$x_1 + x_2 + s_3 = 4$$

which you will recognize as the equality version of the original constraint on the metal finishing machine production limit.

The objective function is also rewritten to align with the constraints. The original version was $Z = 15x_1 + 10x_2$, but we have moved all of the variables to the left hand side to create this equivalent version: $Z - 15x_1 - 10x_2 = 0$.

The *MRT* column is for the minimum ratio test. We will fill this column in as we begin the simplex calculations. The *basic variable* column shows the single basic variable that occurs in each equation.

The most important thing about Tableau 4.1 is that it is in *proper form* (remember, *standard form* is a type of LP, but *proper form* is the way that a tableau is written). A tableau in proper form has these characteristics:

- exactly 1 basic variable per equation.
- the coefficient of the basic variable is always exactly +1, and the coefficients above and below the basic variable in the same column are all 0.
- Z is treated as the basic variable for the objective function row (equation 0).

A big advantage of proper form is that you can always read the current solution directly from the tableau. This is because there is exactly one basic variable per row, and the coefficient of that variable is +1. All the other variables in the row are nonbasic (set to zero, remember?), so the value of any variable is just given by the value shown in the right hand side column. For example, in Tableau 4.1, the value of s_3 is 4. What's the value of x_1? Well, x_1 does not appear in the list of basic variables, which means it must be nonbasic, and nonbasic variables are always zero, hence x_1 is zero in this basic feasible solution. What's the value of the objective function in this tableau? Because Z is the basic variable for equation 0, we can read it's current value from the RHS column: its value is 0.

Now we can use the tableau to keep track of our calculations as we go through the steps of phase 2 of the simplex method as described in Figure 4.1.

Step 2.1: Are we optimal yet?

We are optimal if no entering basic variable is available. Recall that the entering basic variable chosen is the one that gives the fastest rate of increase in the objective function value. In the original version of the objective function, $Z = 15x_1 + 10x_2$, the obvious entering basic variable is x_1. In the tableau, however, we have moved all of the variables to the left hand side of the equation, to obtain the form $Z – 15x_1 – 10x_2 = 0$. Because of this the rule is a little bit different: *choose the variable in the objective function row that has the most negative value as the entering basic variable*.

In this step, then, just check to see if there are *any* negative coefficients in the objective function row. There are two potential entering basic variables in the tableau at this point, so we are *not* optimal yet: the simplex iterations must continue.

Of course, we are only selecting from among currently nonbasic variables at this point: these are the only ones that could enter the basis. A nice feature of proper form is that *only* nonbasic variables can have a nonzero value in the objective function row, so you never have to worry about accidentally selecting a basic variable (which is already in the basic set) as the entering basic variable. Proper form enforces this by requiring that each basic variable have all zeroes in its column, except for the row in which it is the basic variable.

Step 2.2: Select the entering basic variable

As described above, when using the simplex tableau, the entering basic variable is the nonbasic variable in the objective function row (equation 0) that has the most negative coefficient. In Tableau 4.1, this is x_1 whose objective function coefficient of –15 is more negative than the objective function coefficient of –10 for x_2. The column for the entering basic variable is called the *pivot column*. The pivot column is highlighted for ease of reference in Tableau 4.2.

Recall that the entering basic variable, which is currently nonbasic and therefore set to zero, will now be allowed to take on a positive value.

Step 2.3: Select the leaving basic variable

The minimum ratio test is used to determine the leaving basic variable. Recall that this test determines which constraint most limits the increase in the value of the entering basic variable. The most limiting constraint is the one whose basic variable is driven to zero first as the entering basic variable increases in value.

For the minimum ratio test, look only at the entries in the pivot column (the column for the entering basic variable) for the constraint rows, and calculate:

(RHS)/(coefficient of entering basic variable)

There are two special cases:

- if the coefficient of the entering basic variable is zero: enter *no limit* in the minimum ratio test column
- if the coefficient of the entering basic variable is negative: enter *no limit* in the minimum ratio test column.

The minimum ratio test is *never* applied to the objective function row. Why? The objective function row is not a constraint, so it can never limit the increase in the value of the entering basic variable. The objective function always goes along for the ride, just measuring the value of the objective at any point that it is given.

As usual in the minimum ratio test, the leaving basic variable is associated with the row that has the minimum value of the ratio test. This row is called the *pivot row* and is highlighted for ease of reference in Tableau 4.2.

At this point, we have determined that we are not optimal, and have selected an entering basic variable and a leaving basic variable. These can now be swapped so that we can move to the next basic feasible solution (feasible cornerpoint). The calculations so far are summarized in Tableau 4.2.

basic variable	eqn. no.	Z	x_1	x_2	s_1	s_2	s_3	RHS	MRT
Z	0	1	-15	-10	0	0	0	0	*never*
s_1	1	0	1	0	1	0	0	2	2/1 = 2
s_2	2	0	0	1	0	1	0	3	*no limit*
s_3	3	0	1	1	0	0	1	4	4/1 = 4

Tableau 4.2: Updated ABC at the origin.

Step 2.4: Update the tableau

Now that the entering and leaving basic variables are known, we can move to the new and better basic feasible solution. This is easily done by putting the tableau back into proper form. It is not in proper form now because the exchange of the entering and leaving basic variables has not yet been done, so for example, s_1 still shows as a basic variable when it should now be nonbasic, and the column for x_1, the new basic variable does not have zeros everywhere except in its own row.

Here are the steps for putting the tableau back into proper form:

Step 2.4.1: In the column entitled *basic variable*, replace the leaving basic variable listed for the pivot row by the entering basic variable. In Tableau 4.2 the basic variable listed for equation 1 is s_1, but in Tableau 4.3 it is x_1.

Step 2.4.2: The tableau element where the pivot row and the pivot column intersect is known as the *pivot element*. This will be the coefficient associated with the new basic variable for this row, so if it is not already +1, then we must divide all of the elements in the pivot row by the pivot element to obtain a +1 in the pivot element position. Luckily, we already have a +1 in the pivot element position in Tableau 4.2.

Step 2.4.3: Now we need to eliminate all of the coefficients in the pivot column except the pivot element. This is done by simple Gaussian operations. To remove the pivot column element in some row k for example, proceed as follows:

$$\text{(new row } k) = (\text{row } k) - (\text{pivot column coefficient in row } k)\times(\text{pivot row})$$

Remember that the pivot row has been updated by now so that the pivot element is +1.

For example, to clean out the -15 that appears in the objective function row,

- new Z coefficient= $1 - (-15)\times 0 = 1$
- new x_1 coefficient = $-15 - (-15)\times 1 = 0$
- new x_2 coefficient = $-10 - (-15)\times 0 = -10$
- new s_1 coefficient = $0 - (-15)\times 1 = 15$
- new s_2 coefficient = $0 - (-15)\times 0 = 0$
- new s_2 coefficient = $0 - (-15)\times 0 = 0$
- new RHS = $0 - (-15)\times 2 = 30$

Always follow the procedure given above to make sure that there is never any change in the Z column, because Z will always be the basic variable for the objective function, so you must always maintain a +1 in the equation 0 row under the Z column.

Tableau 4.3 shows the tableau after it has been put into proper form by eliminating all of the elements in the pivot row except the +1 for the pivot element. This now summarizes the second basic feasible solution (or cornerpoint, if thinking graphically).

basic variable	eqn. no.	Z	x_1	x_2	s_1	s_2	s_3	RHS	MRT
Z	0	1	0	-10	15	0	0	30	*never*
x_1	1	0	1	0	1	0	0	2	
s_2	2	0	0	1	0	1	0	3	
s_3	3	0	0	1	-1	0	1	2	

Tableau 4.3: The second basic feasible solution.

As before, the values of the basic variables at the second feasible solution can be read directly from Tableau 4.3. For example, the value of x_1 is now 2. Recalling that nonbasic variables are always zero, the new basic feasible solution is (x_1,x_2,s_1,s_2,s_3) = (2,0,0,3,2). The value of Z can also be read directly from the tableau because it is the basic variable for equation 0: its value now is 30, which is an improvement over the value at the initial basic feasible solution.

In putting the tableau into proper form, we have reached the second basic feasible solution. This process of moving from one basic feasible solution to the next is called *pivoting*, and so we have just completed one pivot.

Completing the ABC solution

We can see from Tableau 4.3 that have not yet reached the optimal solution because there are still negative coefficients available in the objective function row, equation 0. So the solution process continues for another iteration, as follows:

- **Step 2.2:** x_2 is the entering basic variable (see Tableau 4.4).
- **Step 2.3:** the minimum ratio test shows that s_3 is the leaving basic variable (see Tableau 4.4).
- **Step 2.4:** Tableau 4.5 shows the revised tableau in proper form.

basic variable	eqn. no.	Z	x_1	x_2	s_1	s_2	s_3	RHS	MRT
Z	0	1	0	-10	15	0	0	30	*never*
x_1	1	0	1	0	1	0	0	2	no limit
s_2	2	0	0	1	0	1	0	3	3/1 = 3
s_3	3	0	0	1	-1	0	1	2	2/1 = 2

Tableau 4.4: The second iteration.

basic variable	eqn. no.	Z	x_1	x_2	s_1	s_2	s_3	RHS	MRT
Z	0	1	0	0	5	0	10	50	*never*
x_1	1	0	1	0	1	0	0	2	
s_2	2	0	0	0	1	1	-1	1	
x_2	3	0	0	1	-1	0	1	2	

Tableau 4.5: The optimal tableau.

Applying Step 2.1 shows that we have reached the optimum solution in Tableau 4.5: there are no negative variable coefficients in the objective function. This means that there is no improving direction in which to move. We can now read the optimum solution from the tableau (remembering that any variables not shown in the *basic variables* column must be nonbasic and hence zero): $(x_1,x_2,s_1,s_2,s_3) = (2,2,0,1,0)$. The optimum value of the objective function is the value of Z shown in the tableau: 50.

The optimum solution shows that the Acme Bicycle Company should produce 2 mountain bikes per day and 2 racing bicycles per day for a total daily profit of $50 per day.

Special Cases in Tableau Manipulation

There are several unusual cases that arise as you progress from tableau to tableau while carrying out the simplex method. Each of these special cases has an interesting interpretation.

Tie for the Entering Basic Variable

Suppose that there are two or more nonbasic variables that have the same most negative value in the objective function row. For example, suppose that the objective function for the Acme Bicycle Company problem was actually $Z = 15x_1 + 15x_2$ (i.e. $Z - 15x_1 - 15x_2 = 0$). What is the interpretation? This means that both x_1 and x_2, if allowed to become basic and increase in value, would both increase Z at the same rate. See Figure 4.2 for an example of a cornerpoint at which the rate of increase would be the same for two entering basic variables.

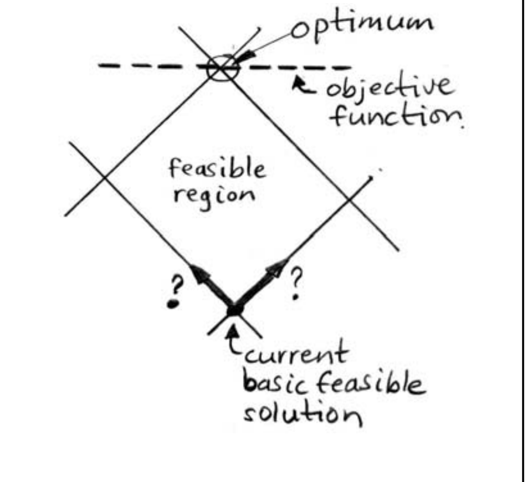


Figure 4.2: A tie for the entering basic variable.

So how to handle this situation? Simple: choose the entering basic variable arbitrarily from among those tied at the most negative value, because there is no good way to determine in advance which selection will reach the optimum solution in the smallest number of pivots. Figure 4.3 shows an example in which two variables are tied, but for which many more pivots are required to reach the optimum via one choice (clockwise) than via the other choice (counterclockwise).

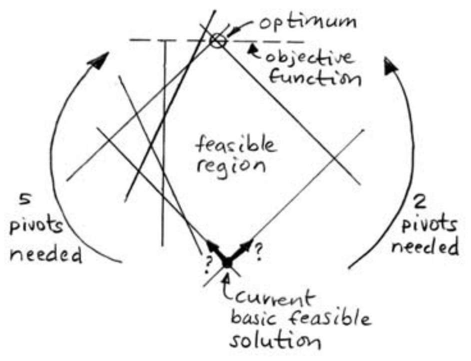


Figure 4.3: One entering basic variable mav be better than another.

In fact, the choice of the "most negative objective function coefficient" is only a *heuristic* method (rule of thumb) for choosing the entering basic variable. It usually provides a good entering basic variable, but not always. On the other hand it is very quick. In commercial solvers for very large LPs, the solver may not even calculate the objective function coefficients for all of the variables. Instead it might only calculate them for say 1000 of the variables in a 500,000 variable problem, and take the most negative from among those 1000. Next time around it might calculate the objective function coefficients for the next 1000 variables and choose the entering basic variable only from among that 1000. The only time it might calculate the objective function coefficients for all 500,000 variables is at the optimum, when it must check that there are no negative coefficients left at all.

Tie for the Leaving Basic Variable

Recall the mantra: "nonbasic, variable set to zero, corresponding constraint is active". This should remind you that the choosing the leaving basic variable via the minimum ratio test is the same as finding out which constraint you first bump into as the entering basic variable increases in value. So what does it mean when there is a tie during the minimum ratio test? As Figure 4.4 shows, it must mean that there are two constraints (with their corresponding basic variables) that are tied as the first constraints you bump into. Choosing the basic variable associated with constraint A *or* the basic variable associated with constraint B as the leaving basic variable will define the same new cornerpoint. Technically speaking, the basic feasible solutions will be different, because each will define a different partitioning of the variables into basic and nonbasic sets.

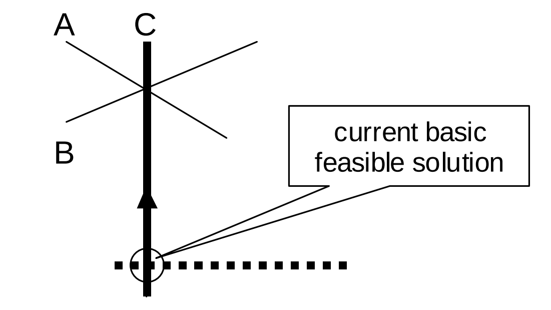


Figure 4.4: A tie for the leaving basic variable.

So how do we choose between the two possibilities for the leaving basic variable? Again, simply choose arbitrarily. The variable that is *not* chosen as the leaving basic variable will remain basic, but will have a calculated value of zero. In contrast, the variable chosen as the leaving basic variable will of course be forced to zero by simplex. Both variables must be zero simultaneously because both constraints are active at that point; it's just that simplex only needs one of them to define the basic feasible solution.

When there is a tie for the leaving basic variable as we have described, the basic feasible solution defines what is known as a *degenerate solution*. Degenerate solutions can lead to an infinite loop of solutions that traps the simplex solution method; this is known as *cycling*. This does not happen in 2-dimensional problems, so it is difficult to show a simple diagram. But here is the general idea of how cycling happens, if it could happen in two dimensions (refer to Figure 4.4): first define the basic feasible solution using constraints A and C, then pivot to a new basic feasible solution defined by constraints B and C, then pivot back to the basic solution defined by constraints A and C, then continue around this loop infinitely.

Cycling is actually fairly rare in real problems, and most good-quality commercial linear programming solvers have special code to detect and exit from cycling behavior. One of the most often used anti-cycling routines for network LPs was in fact invented by former Carleton University professor William Cunningham.

Now suppose that all of the minimum ratio tests are tied at "no limit": what does this mean? The interpretation is straightforward: no constraint puts a limit on the increase in the value of the entering basic variable! Because of this, there is then no limit on the increase in the value of the objective function. Your result might then show that you can make an infinite amount of profit for the Acme Bicycle Company, for example. Sounds great, but it usually means that you forgot a constraint. You may have assumed, for example, that you can sell whatever quantity of widgets you manufacture, or that a key ingredient is available in infinite quantities. Problems of this type are called *unbounded*, and have an *unbounded solution*. Figure 4.5 gives an example of an unbounded problem.

An unbounded problem is recognized while pivoting when all of the entries in the pivot column (the entering basic variable column) are zero or negative, and hence place no limit on the increase in the value of the entering basic variable.

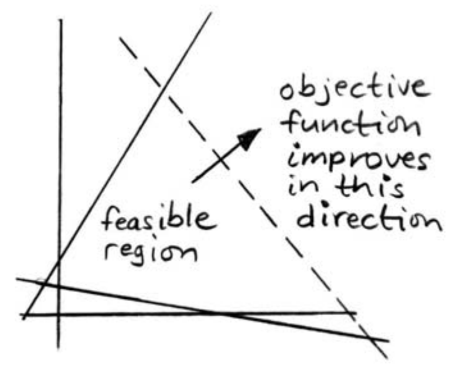


Figure 4.5: An unbounded linear program.

At the Optimum, the Coefficients of Some Nonbasic Variables are Zero in the Objective Function Row

You expect that the objective function coefficients of the *basic* variables will be zero because putting the tableau into proper form requires that those coefficients be zeroed. But the coefficients of the *nonbasic* variables are normally nonzero, and in fact are normally positive at the optimum (if any are negative, you are not yet at the optimum). So what does it mean if a nonbasic variable has a coefficient of zero in the objective function at the optimum? It means that choosing this variable as the entering basic variable increases *Z* at the rate of zero, i.e. that choosing this variable as the entering basic variable has *no effect* on *Z*!

But choosing such a variable as the entering basic variable means that you will pivot to a *different* basic feasible solution. And the resulting new basic feasible solution will have the *same* value of *Z.* Two different basic feasible solutions, same value of *Z*? That means that there must be *multiple optimum solutions*. You will have the same value of the objective function at both of the two basic feasible solutions *and* at any point between those two solutions. Remember that there may in fact be more than two basic feasible solutions with the same value of Z.

You may wish to find the other basic feasible solutions. Sometimes one solution is preferred over another for non-quantifiable reasons, so you would like to see a selection of solutions that give the optimum value. To see the other optima, choose one of the nonbasic variables whose objective function coefficient is zero as the entering basic variable, and pivot to another basic feasible solution as you normally would.

Chapter 5: Solving General Linear Programs

So far we have concentrated on linear programs that are in *standard form*, which have a maximization objective, all constraints of ≤ type, all of the right hand side constants are greater than or equal to zero, and all of the variables are restricted to nonnegative values. As we saw in Chapter 2 (see Figure 2.5), a standard form LP has one big advantage: the origin [the point at (0,0,0,…)] is always a feasible cornerpoint, so the simplex method can always start there. The simplex method then happily proceeds from cornerpoint to better cornerpoint until it recognizes optimality. But the simplex method is in trouble if it can't find that initial cornerpoint to start at.

All forms of LPs arise in practice, not just standard form LPs. You may have minimization objectives, constraints of the ≥ or = type, variables which can take negative values, and right hand side constants which are negative. How can we deal with these forms? That is the subject of this chapter.

Minimization

There are two simple ways to deal with a minimization objective function. The easiest and most popular method is to simply multiply the minimization objective function by -1, and then to *maximize* the resulting function. For example, suppose your objective function is:

$$\text{minimize } Z = 12x_1 + 5x_2 - 7x_3$$

then convert to a maximization:

$$\text{maximize } (-Z) = -12x_1 - 5x_2 + 7x_3$$

As a reminder that the original objective function was a minimization, put a –1 in the Z column of the simplex tableau. After solving the linear program, recover the solution as follows:

- multiply the optimum objective function value by –1 to recover the minimum value of the original minimization objective function.
- the values of the variables as taken from the tableau will be correct.

The second and less popular alternative to dealing with minimization objectives is to keep the minimization objective function in place, unaltered, but to change the rules of tableau manipulation, as follows:

- new test for optimality: are all of the coefficients in the objective function row *nonpositive*? If yes, then stop iterating: you've reached the optimal solution.
- new choice for entering basic variable: choose the *most positive* coefficient in the objective function row.

For most students, it is easiest to use the first option because the rules of tableau manipulation remain unchanged. This is the method that we will assume from now on.

Equality Constraints

Suppose that the Acme Bicycle Company problem included an equality constraint, as below:

maximize $Z = 15x_1 + 10x_2$

subject to: $x_1 \leq 2$

$x_2 \leq 3$

$x_1 + x_2 = 4$ ← *note change to equality constraint!*

As you can see in Figure 5.1, the origin is no longer a feasible cornerpoint at which to start the simplex method. There doesn't seem to be any easy way to find an initial basic feasible solution. How do we get the simplex method started?

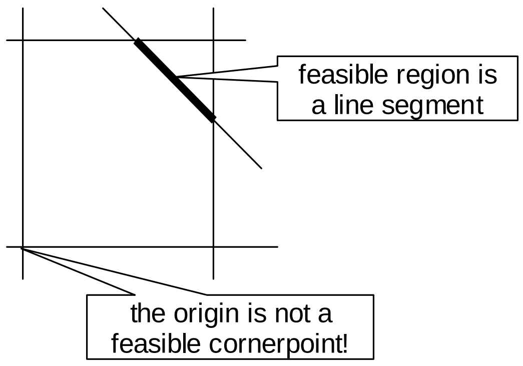

feasible region is a line segment
the origin is not a feasible cornerpoint!

Figure 5.1: An equality constraint eliminates the origin as a feasible cornerpoint.

When converting to equality format, we will add one slack variable for the first constraint, and one slack variable for the second constraint, but will not need a slack variable for the third constraint because it is already in equality format. And therein lies the difficulty! The third constraint will involve only x_1 and x_2, and we may not be able to directly see how to set those two values such that all of the constraints are simultaneously satisfied. You can see how this problem is even more acute in LPs having many constraints.

The answer lies in adding a dimension to the mathematical representation of the original problem by adding a nonnegative *artificial variable* to any equality constraints in the model, in this case to the third constraint. While an artificial variable may *look* like a slack variable, it is treated much differently.

After converting the problem to equality format and adding the artificial variable (a_1), the initial set of constraint equations looks like this:

$$\begin{array}{lcccccl} (1) & x_1 & & +s_1 & & & = 2 \\ (2) & & x_2 & & +s_2 & & = 3 \\ (3) & x_1 & +x_2 & & & +a_1 & = 4 \end{array}$$

If a_1 were a slack variable, we would be able to use the origin as the initial feasible cornerpoint to get started, giving an initial basic feasible solution of $(x_1,x_2,s_1,s_2,a_1) = (0,0,2,3,4)$. But here's the difficulty: a_1 cannot be nonzero in the final solution, or the original 3rd constraint is violated! Somehow we need to force a_1 to zero to make sure that the 3rd constraint is satisfied.

The Phase 1 LP

The solution to this dilemma is to solve *two* LPs. This is where the phase 1 that we have ignored until now finally comes in. The sole purpose of the phase 1 LP is to obtain a basic feasible solution for the original problem, as follows:

- **Phase 1:** solve an LP whose objective is to minimize the value of any artificial variables in the model, in this case a_1. If you can drive all of the artificial variables to zero, then you will be at a feasible cornerpoint of the original problem.
- **Phase 2:** starting at the feasible cornerpoint found during phase 1, switch over to the original objective function and continue iterating until the optimum point is found.

The phase 1 objective function is simply to minimize the sum of the artificial variables. In general this is expressed as:

$$\text{minimize } W = a_1 + a_2 + a_3 + \ldots$$

You can think of W as the sum of the constraint violations because each artificial variable, in a manner similar to slack variables, takes up the slack between the constraint left hand side and the right hand side. But the left hand side should be equal to the right hand side in an equality constraint, so the artificial variable shows the amount by which the constraint is violated. Because we are minimizing W, after we multiply by –1 and move the variables to the left hand side, the phase 1 objective function looks like this:

$$\text{maximize } -W + a_1 + a_2 + a_3 + \ldots = 0$$

In setting up the tableau, the original (or phase 2) objective function is also retained. We use the phase 1 objective during phase 1, but also update the phase 2 objective at the same time so that it is ready to go when phase 1 ends. On the other hand, after phase 1 ends successfully we ignore the phase 1 objective function and the artificial variables thereafter because they are no longer useful.

Here is one important thing to remember: the first phase 1 tableau is never in proper form. This is because the artificial variables will be the initial basic variables for the constraints they appear in, and yet they will appear twice in the column: once in their constraint row, and once in the objective function row. They should appear only once, with a +1 coefficient, in their constraint row. Hence some initial work is required to put the tableau into proper form so that the iterations can start.

Let's go through the iterations for the version of the Acme Bicycle Company problem in which the third constraint is an equality. The initial tableau is as follows:

basic variable	eqn. no.	W	Z	x_1	x_2	s_1	s_2	a_1	RHS	MRT
W	ph1	-1	0	0	0	0	0	1	0	*never*
Z	ph2	0	1	-15	-10	0	0	0	0	*never*
s_1	1	0	0	1	0	1	0	0	2	
s_2	2	0	0	0	1	0	1	0	3	
a_1	3	0	0	1	1	0	0	1	4	

As you can see, this tableau has two objective functions: W for phase 1, which seeks to minimize the sum of the artificial variables (only a_1 in this case), and Z for phase 2, the original objective. Looking at the a_1 column, you can see that this tableau is not in proper form: we need to eliminate the coefficient in the phase 1 (W) objective function row. To do this, subtract each row containing an artificial variable from the phase 1 objective function row. In the tableau above, subtract equation 3 from equation *ph1*. The tableau in proper form is shown below.

basic variable	eqn. no.	W	Z	x_1	x_2	s_1	s_2	a_1	RHS	MRT
W	ph1	-1	0	-1	-1	0	0	0	-4	*never*
Z	ph2	0	1	-15	-10	0	0	0	0	*never*
s_1	1	0	0	1	0	1	0	0	2	
s_2	2	0	0	0	1	0	1	0	3	
a_1	3	0	0	1	1	0	0	1	4	

Notice that the RHS for the phase 1 objective function now reads "-4". But don't forget that we multiplied the original phase 1 minimization objective function by –1 so that we could operate on it as a maximization objective function: the –1 in the W column reminds us of this. So the true value of W at this point is 4, meaning there are 4 units of constraint violation. The goal of phase 1 is to reduce the sum of the constraint violations to zero. During phase 1, we are using the phase 1 objective function, and as the tableau above shows, both x_1 and x_2 are tied for the entering basic variable with objective function coefficients of –1. Let us arbitrarily choose x_2 as the entering basic variable. The calculations are shown below.

basic variable	eqn. no.	W	Z	x_1	x_2	s_1	s_2	a_1	RHS	MRT
W	ph1	-1	0	-1	-1	0	0	0	-4	*never*
Z	ph2	0	1	-15	-10	0	0	0	0	*never*
s_1	1	0	0	1	0	1	0	0	2	no limit
s_2	2	0	0	0	1	0	1	0	3	3/1=3
a_1	3	0	0	1	1	0	0	1	4	4/1=4

basic variable	eqn. no.	W	Z	x_1	x_2	s_1	s_2	a_1	RHS	MRT
W	ph1	-1	0	-1	0	0	1	0	-1	*never*
Z	ph2	0	1	-15	0	0	10	0	30	*never*
s_1	1	0	0	1	0	1	0	0	2	2/1=2
x_2	2	0	0	0	1	0	1	0	3	no limit
a_1	3	0	0	1	0	0	-1	1	1	1/1=1

The phase 1 objective function value, the sum of the constraint violations, has been reduced to 1 in the tableau above. Phase 1 is not yet complete though: there is still a negative coefficient in the phase 1 objective function row. Note that the phase 2 objective has also been updated to reflect its value at this new (infeasible) cornerpoint.

basic variable	eqn. no.	W	Z	x_1	x_2	s_1	s_2	a_1	RHS	MRT
W	ph1	-1	0	0	0	0	0	1	0	*never*
Z	ph2	0	1	0	0	0	-5	15	45	*never*
s_1	1	0	0	0	0	1	1	-1	1	
x_2	2	0	0	0	1	0	1	0	3	
x_1	3	0	0	1	0	0	-1	1	1	

The tableau above has at last managed to force the phase 1 objective function to zero. Phase 1 is now complete: no constraints are violated, and we are at a feasible cornerpoint for the original problem. Now it is time to discard the phase 1 objective function, the *W* column, and all of the artificial variables (just a_1 in this case), which results in the tableau below. Note that the phase 2 objective function, which has been carried through all of the calculations so far, is updated, in proper form, and ready to go. And because there is a negative coefficient in the phase 2 objective function row, the iterations must continue.

basic variable	eqn. no.	Z	x_1	x_2	s_1	s_2	RHS	MRT
Z	ph2	1	0	0	0	-5	45	*never*
s_1	1	0	0	0	1	1	1	1/1=1
x_2	2	0	0	1	0	1	3	3/1=3
x_1	3	0	1	0	0	-1	1	no limit

basic variable	eqn. no.	Z	x_1	x_2	s_1	s_2	RHS	MRT
Z	ph2	1	0	0	5	0	50	*never*
s_2	1	0	0	0	1	1	1	
x_2	2	0	0	1	-1	0	2	
x_1	3	0	1	0	1	0	2	

After this final phase 2 iteration, the solution is complete: there are no negative coefficients remaining in the phase 2 objective function. The final solution is (x_1,x_2,s_1,s_2)

= (2,2,0,1) with $Z = 50$. As before, Acme should produce mountain bikes at the rate of 2 per day and racers at the rate of 2 per day to achieve a profit rate of $50 per day.

Because of the added dimensions due to the slack and the artificial variables, it is not strictly possible to draw a picture illustrating what happens during a two-phase solution. But we can project what is happening onto a 2-dimensional, x_1-x_2 plane, as in Figure 5.2. The figure shows that the phase 1 solution still starts at the origin, which is *feasible* as far as the phase 1 formulation is concerned, because of the added artificial variable, but which is *infeasible* for the phase 2 formulation. The phase 1 procedure moves through other cornerpoints that are phase-1-feasible but phase-2-infeasible until it reaches a cornerpoint that is feasible for *both* phase 1 and phase 2. It is at this point that the phase 1 formulation is discarded in favor of the phase 2 formulation. The phase 2 solution then iterates from there to problem optimality.

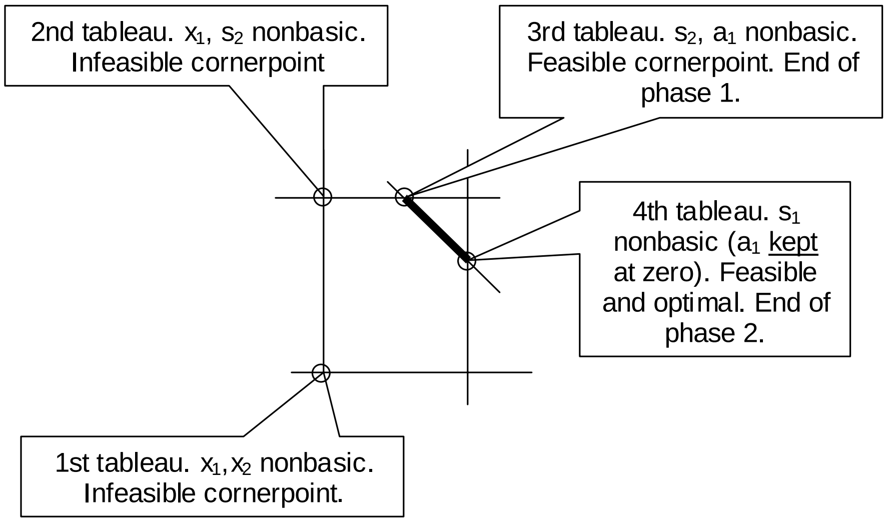

2nd tableau. x_1, s_2 nonbasic. Infeasible cornerpoint
3rd tableau. s_2, a_1 nonbasic. Feasible cornerpoint. End of phase 1.
4th tableau. s_1 nonbasic (a_1 kept at zero). Feasible and optimal. End of phase 2.
1st tableau. x_1,x_2 nonbasic. Infeasible cornerpoint.

Figure 5.2: Projecting the cornerpoints onto the x1-x2 plane.

It's no surprise to see that we arrive at the same solution obtained when we first solved the Acme Bicycle Company problem in Chapter 4. The problems are identical, except that the feasible region for this version of the problem is much smaller. The important point is that the two feasible regions overlap on (2,2), which is the optimal point.

Recognizing Infeasible Linear Programs

It is not difficult to construct linear programs for which there is no feasible solution. The infeasibility can be as simple as two parallel and oppositely-oriented constraints (e.g. $x_1 \leq 8$ and $x_1 \geq 10$), or it could be caused by a number of constraints interacting in such a way that the feasible region is completely eliminated. Fortunately, infeasibility is simple to recognize: if the phase 1 LP terminates (all of the phase 1 objective function coefficients are nonnegative), and yet W is still positive, then not all of the constraint violations have been eliminated. This means that the LP is infeasible. One or more of the artificial variables could not be forced to zero, and for feasibility, *all* of the artificial variables must be forced to zero.

The solution procedure terminates at this point without going on to the phase 2 solution. Now you need to analyze your linear program to determine *why* it is infeasible. It may be that you have correctly expressed the constraints, so there really is no solution to the problem. But it's much more likely that you have made an error in writing the constraints, such as expressing a ≤ constraint as a ≥ constraint, or added a superfluous, contradicting constraint.

Fortunately, there are tools that can assist in your analysis. Most modern commercial solvers incorporate special routines to find an *Irreducible Infeasible Set (IIS)* of constraints. An IIS is usually a very small subset of the constraints in the model and has a special property: the IIS is infeasible, but taking away one or more of the constraints in the IIS makes it feasible. This means that every member of the IIS contributes directly to the infeasibility. Instead of laboriously working through the thousands of constraints in a model, an IIS brings the problematic constraints directly to the modeler's attention. Human insight is then needed to determine which of the constraints in the IIS to correct.

Most of the routines for finding IISs and analyzing infeasible linear programs in general were developed by, ahem, your humble author. I would very much enjoy going into great detail on this *fascinating* topic, but alas, time and space are limited in this introductory exposition…

The Big-M Method

The Big-M method is an alternative to the two-phase method that we have described above. It is presented in many textbooks, but is not used in commercial solvers because it causes numerical problems in the computer calculations. The main idea in the Big-M method is to combine the phase 1 and phase 2 objective functions into a single objective function. This is done by writing the original objective function, and then adding the artificial variables to it. Each artificial variable appears in the objective function with a large penalty coefficient: this is the "Big-M", which stands for "big multiplier". For example, the Big-M version of the objective function for the problem solved above is:

$$\text{maximize } Z = 15x_1 + 10x_2 - \mathrm{M}a_1$$

As you can see, if M is a large positive number then a straightforward solution of the problem will drive a_1 to zero because it has such a reducing effect on Z, which simplex is trying to maximize. When a_1 is zero, the solution will be feasible for the original problem.

Greater-than-or-Equal Constraints

Constraints of the $\geq$ type with a positive right hand side constant are not allowed in a standard form LP, again because such constraints eliminate the origin as a feasible cornerpoint, similar to the way in which equality constraints eliminate the origin. You can't simply multiply through by –1 because this then gives a negative right hand side. The problem with negative right hand sides is that they imply a negative value for a basic variable in the tableau, which is not possible if the variables are restricted to nonnegativity.

The solution to this problem is to convert the $\geq$ constraint to an equality constraint by including a *surplus variable*. A surplus variable is exactly the same as a slack variable, except that it expresses the surplus of the left hand side over the right hand side of the constraint. For example:

$$3x_1 + 5x_2 \geq 20 \ \Rightarrow\ 3x_1 + 5x_2 - s_1 = 20.$$

A surplus variable cannot be used as the basic variable for the constraint it appears in because its coefficient is –1, and we need +1 for the coefficients of basic variables. On the other hand, the constraint is now an equality, so we can treat it exactly as we treat any other equality constraint: add an artificial variable and use a phase 1 procedure! In the end, our example constraint would look like this:

$$3x_1 + 5x_2 - s_1 + a_1 = 20$$

So there will be *two* variables added to every $\geq$ constraint: one surplus variable (with a coefficient of –1) and one artificial variable (with a coefficient of +1). Any artificial variables added to $\geq$ constraints will be driven to zero during phase 1.

Negative Right Hand Side

As described above, a negative right hand side constant presents a problem because it implies that the basic variable for the constraint must take on a negative value, which is not possible for nonnegative variables. The solution to this problem is simple: just multiply by –1 to make the right hand side positive, and then treat the resulting $\leq$, $\geq$, or = as appropriate.

How Many Variables?

The reformulation of the problem for solution via the simplex method does add a number of variables. Let's suppose we have a problem which has *n* original variables, *l* less-than-or-equal constraints, *g* greater-than-or-equal constraints, and *e* equality constraints. The reformulated model will then have:

- *n* original variables,
- *l* slack variables,
- *g* surplus variables, and

- $g + e$ artificial variables.

All of the g + e artificial variables will be eliminated at once during the same phase 1 procedure using a phase 1 objective of minimizing $W = \sum$(artificial variables).

The speed of the simplex solution depends mostly on the number of constraints in the model, not the number of variables, so the blow-up in the number of variables is not important. Solution speed depends on the number of constraints because these define the cornerpoints, and it is the number of cornerpoints that must be traversed that determines how long the solution takes.

An Example Conversion

Here is another non-standard-form linear program that is vaguely related to the Acme Bicycle Company problem:

minimize	$Z = 15x_1 + 10x_2$	$\leftarrow$ minimization instead of maximization
	$x_1 \leq 2$	
	$x_2 = 3$	$\leftarrow$ equality instead of $\leq$
	$x_1 + x_2 \geq 4$	$\leftarrow$ $\geq$ instead of $\leq$

When converted for solution in the simplex tableau, this becomes:

maximize	$(-W) + a_1 + a_2 = 0$	phase 1 objective function
maximize	$(-Z) + 15x_1 + 10x_2 = 0$	phase 2 objective function
	$x_1 + s_1 = 2$	
	$x_2 + a_1 = 3$	
	$x_1 + x_2 - s_2 + a_2 = 4$	

The initial basic variables will be s_1 for the first constraint, a_1 for the second constraint, and a_2 for the third constraint. These equations can be transported directly to the tableau, which must then be put into proper form (it will not be in proper form because the two artificial variables, both basic, will appear in the phase 1 objective function row as well as in their respective constraints).

Variables that can be Negative

In some problems, the most natural formulation of the problem allows some variables to possibly take on negative values. For example, the variable might express the difference between the production level now and the production level in the last fiscal quarter, so it will take on a negative value if the production level falls. There are two cases to consider, as described below.

There is a Lower Bound on the Negative Value

You may know, for example, that the production level cannot drop more than 25 units from its present value, that is that $x_1 \geq -25$, or in general that $x_j \geq L_j$, where L_j is a negative number. In this case we can use a change of variables: $x_j = x_j' + L_j$. Now we replace x_j by $x_j' + L_j$ everywhere in the problem and solve the resulting LP for the value of x_j'. Afterwards, the value of x_j is recovered from the variable relationship: $x_j = x_j' + L_j$.

For example, consider the following linear program in which one of the variables can be negative:

$$\begin{array}{ccccc}
Z = 15x_1 - 5x_2 & & Z = 15(x_1'-25) - 5x_2 & & Z = 15\,x_1' - 5x_2 - 375 \\
2x_1 \leq 10 & & 2(x_1'-25) \leq 10 & & 2x_1' \leq 60 \\
4x_2 \leq 25 & \Rightarrow & 4x_2 \leq 25 & \Rightarrow & 4x_2 \leq 25 \\
3x_1 - 2x_2 \leq 8 & & 3(x_1'-25) - 2x_2 \leq 8 & & 3x_1' - 2x_2 \leq 83 \\
\boldsymbol{x_1 \geq -25} & & (x_1'-25) \geq -25 & & x_1' \geq 0 \\
x_2 \geq 0 & & x_2 \geq 0 & & x_2 \geq 0
\end{array}$$

As you can see, the final converted form of the problem is an ordinary linear program with nonnegative variables that we already know how to solve! After solving, the value of x_1 is found from the original change of variables relationship: $x_1 = x_1' - 25$. By the way, the constant in the converted objective function is easy to handle: it just appears on the right hand side in the initial tableau and is carried along. The final value of Z is given correctly in the final tableau and is not affected by the change of variables.

Unrestricted Variables

Variables can also take on negative values when they are totally unrestricted; when their value may be anything between minus infinity and plus infinity. Unrestricted variables must be handled in a different way: each unrestricted variable x_j is replaced by the difference of two new nonnegative variables: $x_j = x_j^+ - x_j^-$. Even though x_j^+ and x_j^- are nonnegative, x_j can take on any value between minus and plus infinity by choosing appropriate values for x_j^+ and x_j^-.

As it happens, the simplex method will automatically set one or the other of x_j^+ or x_j^- to zero, or it will set both to zero, but it will never happen that both are positive simultaneously. Why is this? Notice that when you make the change of variables, replacing x_j by $x_j^+ - x_j^-$, you get two tableau columns that are identical, except that one is the negative of the other. This property is maintained throughout the solution. Recall that when a tableau is in proper form, the basic variable has a +1 coefficient, and the rest of the column is filled with zeros. If the x_j^+ and x_j^- columns are negative copies of each other, then it is not possible for both to be basic at the same time: the +1 entry in one column should be mirrored by a –1 entry in the other column, and this is not possible when the tableau is in proper form.

This means that reading the value of x_j after the revised problem is solved is straightforward:

- if $x_j^+ > 0$ then $x_j = x_j^+$,
- if $x_j^- > 0$ then $x_j = - x_j^-$,
- if $x_j^+ = x_j^- = 0$ then $x_j = 0$.

You might think that if all of the variables in the problem are unrestricted, then there must be twice as many variables in the revised problem, but this is not so. In this case we just use the *same* x_j^- in *all* of the change of variable expressions. Upon solution, x_j^- is the absolute value of the most negative variable. The various x_j^+ then express the amount by which each x_j exceeds that most negative value in a manner similar to the case of lower-bounded negative variables.

In Practice

In practice, you don't have to worry about any of the details of converting a linear program into a form that can be solved by the simplex method. The solver software carries out any necessary conversions automatically. Your responsibility is making sure that the mathematical model correctly represents reality, or more accurately, makes a relevant approximation of reality!

Some of the conversions mentioned in this chapter are not in fact needed at all in some LP solvers. For example, many solvers implicitly assume that all variables (and row constraints) are upper and lower bounded. Whether the bounds are positive or negative does not matter: all bounds are handled in a similar manner.

Chapter 6: Sensitivity Analysis

Suppose that you have just completed a linear programming solution which will have a major impact on your company, such as determining how much to increase the overall production capacity, and are about to present the results to the board of directors. How confident are you in the results? How much will the results change if your basic data (e.g. profit per item produced, or availability of a component) is slightly wrong? Will that have a minor impact on your results? Will it give a completely different outcome, or change the outcome only slightly?

These are the kinds of questions addressed by sensitivity analysis. Formally, the question is this: is my optimum solution (both the values of the variables and the value of the objective function) sensitive to a small change in one of the original problem coefficients (e.g. coefficients of the variables in the objective function or constraints, or the right hand side constants in the constraints)? If Z or the x_i change when an original coefficient is changed, then we say that the LP is *sensitive*. We could ask, for example, if the Acme Bicycle Company solution is sensitive to a reduction in the availability of the metal finishing machine from 4 hours per day to only 3 (i.e. a change in the third constraint from $x_1+x_2 \leq 4$ to $x_1+x_2 \leq 3$).

This sort of examination of the impact of the input data on the output results is crucial. The procedures and algorithms of mathematical programming are important, but the problems that really bedevil you in practice are usually associated with data: getting it at all, and getting *accurate* data. Some data, necessary for your mathematical model, is inherently uncertain. Consider profit per item, for example, which is approximated from estimates of the fluctuating costs of raw materials, expected sales volumes, labour costs, etc. What you want to know from sensitivity analysis is which data has a significant impact on the results: then you can concentrate on getting accurate data for those items, or at least running through several scenarios with various values of the crucial data in place to get an idea of the range of possible outcomes.

There are several ways to approach sensitivity analysis. If your model is small enough to solve quite quickly, you can use a *brute force* approach: simply change the initial data and solve the model again to see what results you get. You can do this as many times as needed. At the opposite extreme, if your model is very large and takes a long time to solve, you can apply the formal methods of *classical sensitivity analysis*. The classical methods rely on the relationship between the initial tableau and any later tableau (in particular the optimum tableau) to quickly update the optimum solution when changes are made to the coefficients of the original tableau.

Between these two extremes is *computer-based ranging*. This is simple information about how much certain coefficients can change before the current optimum solution is fundamentally changed. Most commercial LP solvers provide such information. In this introductory book, we will concentrate on this form of sensitivity analysis.

One final observation on the state of the art in sensitivity analysis: you are typically limited to analyzing the impact of changing only *one* coefficient at a time. There are a few accepted techniques for changing several coefficients at once: the 100% rule, and parametric programming. The 100% rule is typically limited to changing only a few coefficients at once, with tight limits on how much they can change, and parametric programming changes all of the coefficients in ratio. What is really needed is a method that allows *all* of the coefficients to vary independently. Recent research by your humble author and graduate student Khaled Ramadan provides such a method: it allows all of the coefficients to be specified as intervals (e.g. the availability of the metal finishing machine is between 3 and 5 hours per day). Unfortunately this method is not yet implemented in commercial LP solvers.

Simple Computer-Based Sensitivity Analysis

Most commercial LP solvers return at least the following information:

- The objective function coefficients for the original variables at the optimum, called the *reduced costs*.
- The objective function coefficients for the slack and surplus variables at the optimum, called the *shadow prices* or *dual prices*.
- The ranges of the original objective function coefficients of the original variables for which the current basis remains optimal.
- The ranges of the right-hand-side constants for the constraints for which the current basis remains optimal.

What is missing from this list is any mention of the constraint coefficients. The real weakness of the simple computer-based sensitivity analysis is that it does not deal with changes to constraint coefficients.

Consider the solution output returned by LINDO solver for the Acme Bicycle Company problem, for example:

```
LP OPTIMUM FOUND AT STEP      2

       OBJECTIVE FUNCTION VALUE
       1)     50.000000

 VARIABLE        VALUE          REDUCED COST
       X1         2.000000           .000000
       X2         2.000000           .000000

      ROW   SLACK OR SURPLUS     DUAL PRICES
       2)          .000000          5.000000
       3)         1.000000           .000000
       4)          .000000         10.000000
```

Note that LINDO refers to the objective function as "row (1)", and the constraints as rows (2)–(4). LINDO also provides the simple ranging information that can be used for sensitivity analysis:

```
RANGES IN WHICH THE BASIS IS UNCHANGED:

                           OBJ COEFFICIENT RANGES
VARIABLE         CURRENT        ALLOWABLE        ALLOWABLE
                   COEF          INCREASE         DECREASE
      X1       15.000000         INFINITY         5.000000
      X2       10.000000         5.000000        10.000000

                           RIGHTHAND SIDE RANGES
     ROW         CURRENT        ALLOWABLE        ALLOWABLE
                    RHS          INCREASE         DECREASE
       2        2.000000         2.000000         1.000000
       3        3.000000         INFINITY         1.000000
       4        4.000000         1.000000         2.000000
```

There are several things to observe about this output data. First, note that the reduced costs for x_1 and x_2 are both zero in the top table: why? Well, the reduced costs are the objective function coefficients of the original variables, and since both original variables (x_1 and x_2) are basic at the optimum, their objective function coefficients must be zero when the tableau is put into proper form. This is always true: either the variable is zero (nonbasic), or the reduced cost or dual price is zero. You can see that the pattern holds for the slack and surplus variables too. The dual prices for rows (2) and (4) are nonzero at the optimum because they correspond to the two active constraints at the optimum, hence their slack variables are nonbasic (value is zero), so the dual prices can be nonzero. When *both* the variable and the associated reduced cost or dual price are zero, then you have either degeneracy if the variable is basic, or multiple optima if the variable is nonbasic, just as we would expect from the tableau.

In the ranging information, the "allowable increase" and "allowable decrease" refer to the maximum *changes* from the "current coefficient" or "current right hand side" which will keep the optimum solution at the same basis. Remember that the basis is the division of the variables into the basic and nonbasic sets. Now because the nonbasic variables identify the constraints that are active, this amounts to saying the following: if you do not change a coefficient more than the amounts in the "allowable increase" or "allowable decrease", then the optimum will still be at the intersection of the same constraints. But note this carefully: *even if the basis does not change, the point in space [(x_1,x_2) for Acme] may change*! This also means that the optimum value of the objective function may change too. Figure 6.1 shows how this can happen.

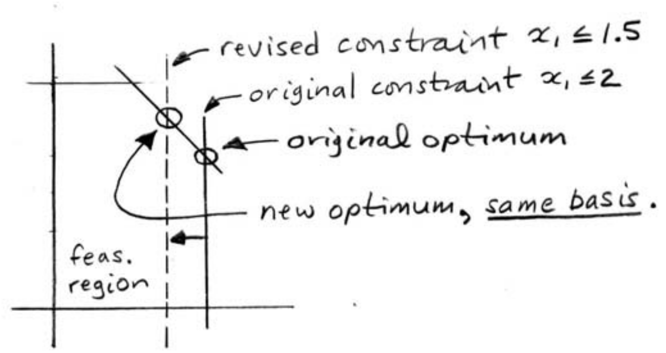

revised constraint $x_1 \le 1.5$
original constraint $x_1 \le 2$
original optimum
new optimum, same basis.
feas. region

Figure 6.1: The basis remains the same, but the point in space is different.

So what is the advantage of staying at the same optimum basis if both the point and the objective function value might change? The advantage lies in the fact that you can still use the solution that you already have to determine what the new point and Z will be after the coefficients have changed, as we will see below. You do not need to re-solve the problem from scratch, which can be a huge advantage for very large problems.

Now let's look at changing the various coefficients as permitted in simple computer-based ranging.

Changing Objective Function Coefficients

Here is the first important observation about changing the coefficients of the objective function: this does not affect the feasible region! For this reason, the optimum point that you found during the original solution of the problem will remain a feasible cornerpoint. The worst that can happen is that the original optimum cornerpoint will no longer be optimum after the objective function is changed. In that case, you can probably restart the solution at the original optimum cornerpoint and continue iterating until you reach the new optimum. The simple ranging analysis will tell you whether you have to do this or not.

Figure 6.2 shows that the effect of changing a coefficient of the objective function is to tilt it. What the allowable range tells you is essentially the maximum tilt in any direction before the optimum moves to a different cornerpoint. For example, looking at the LINDO ranging data above, the objective function coefficient of x_2 can increase from 10 to as much as 15 before the optimum moves to a different basis. The objective function coefficient of x_2 can also decrease from 10 to as little as zero before the optimum moves to a different basis.

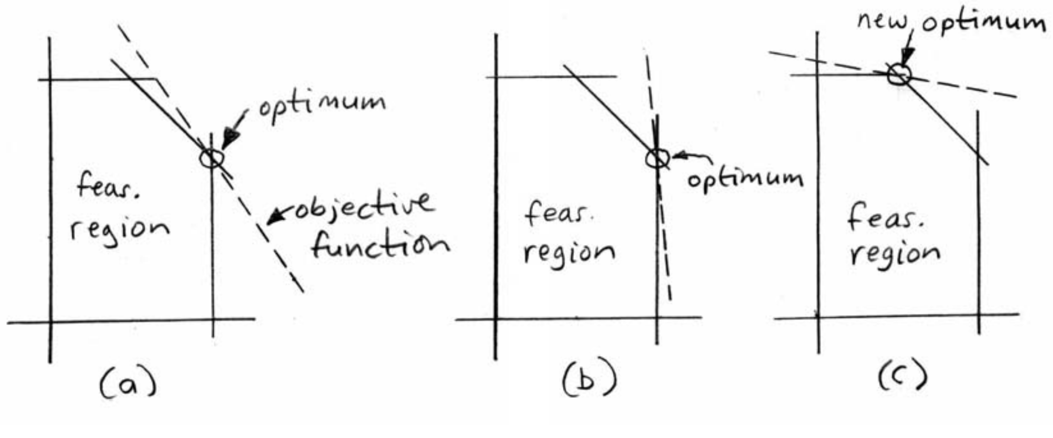


Figure 6.2: Changing a coefficient in the objective function tilts it. (a) original objective function. (b) a tilt which does not change the optimum basis. (c) a tilt which changes the optimum basis.

If you are investigating a change to an objective function coefficient, just compare the change to the ranging table. If the change is more than allowed, then you have no option but to re-solve the problem to find the new optimum cornerpoint. Most commercial solvers allow you to re-start from any basis, so if you have previously saved the optimum basis, then restart there since the new optimum is likely to be only a few pivots away. Or you can restart the model from scratch in the worst case.

But what if the proposed change to the objective function coefficient is within the allowable range? Then how do you determine the new values of Z and the variables? This is straightforward:

- The new optimum variable values will be the same as in the original solution because the constraints are not changed by a modification to the objective function, and we are at the same basis, so we will be at the same place in space.
- You can calculate the new value of the objective function easily: just substitute the old values of the variables into the new objective function. Alternatively, since only one objective function coefficient at a time is changed, you can calculate the change in the objective function value by calculating the difference due to the changed coefficient.

For example, is the Acme Bicycle Company solution sensitive if the objective function coefficient of x_1 is changed from 15 to 25? As you can see from the tables above, the allowable increase for the objective function coefficient for x_1 is infinity, so the basis will remain the same. The optimum solution will still be $(x_1,x_2)=(2,2)$, but the new Z will be given by $25x_1+10x_2 = 25\times2+10\times2 = 70$. You could also calculate the new Z by looking only at the changed coefficient: the change in Z will be $50+2\times$[(new coefficient)–(old coefficient)] $= 50+2\times(25–15) = 70$.

What about changing the objective function coefficient of x_2 from 10 to 25? As you can see from the tables, this is a change of 15 units, which is greater than the allowable increase of 5. This means that the basis will change, so the only way to find out what the variable values and Z will be at the new optimum is to restart the solution and continue iterating until you reach the new optimum point.

Changing a Right Hand Side Constant

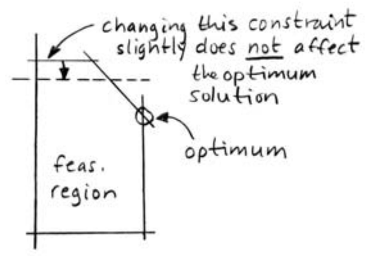

changing this constraint slightly does not affect the optimum solution
optimum
feas. region

Figure 6.3: Changing a right hand side may not affect the optimum at all.

Right hand side constants normally represent a limitation on a resource, and are likely to change in practice as business conditions change. The allowable increase and allowable decrease listed in the ranging tables again show by how much a right hand side can change before the basis changes. Look again at Figure 6.1 to see how the variable values and Z might change at the optimum even when the basis does not change. On the other hand, these things might *not* change if a change is made to a constraint that is not active at the current optimum. Why? Because the change may not affect the optimum point at all, as shown in Figure

6.3.

Figure 6.4, a picture of our favourite Acme Bicycle Company problem, shows more clearly why there is a limited range in which the right hand side can change before the basis changes. Note that a change to the right hand side is the same as a parallel shift of the constraint. In Figure 6.4, the basis remains the same (intersection of 1st and 3rd constraints) while the right hand side of the 1st constraint is changed, until it has been shifted enough that a different constraint becomes active, so the basis changes. As the right hand side of the 1st constraint is gradually increased, the basis does not change until the right hand side reaches a value of 4 (the 1st constraint is now $x_1 \leq 4$). If the right hand side moves beyond 4, then the nonnegativity constraint on x_2 becomes active, and this defines a different basis (3rd constraint and x_2 nonnegativity active). If the right hand side is gradually decreased, then the basis does not change until the right hand side reaches a value of 1 (1st constraint is now $x_1 \leq 1$). Beyond here, the 2nd constraint ($x_2 \leq 3$) becomes active, and the basis is defined by the intersection of the 2nd and 3rd constraints.

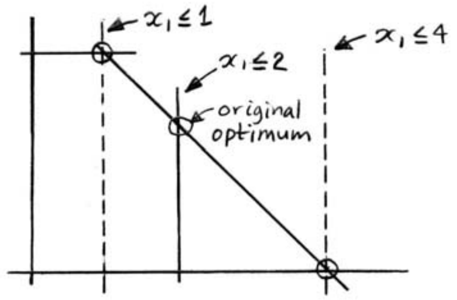


Figure 6.4: Why the basis changes outside the range.

Again, you can see in Figure 6.4 how the Z and variable values may change even though the basis is the same, as long as we propose changes to the coefficients that are within the allowable ranges.

Here is the overall procedure for examining proposed changes to the right hand sides of constraints. First check whether the proposed change is within the allowable range of changes for the right hand side of the constraint. If it is not within the allowable range, then you must re-start the solution process and iterate to the new optimum basis and solution. If the proposed change is within the allowable range, then the values of Z and the variables at the optimum are recovered as follows:

- To calculate the new value of Z:
    - If it is a maximization problem: $Z_{new} = Z_{original}$+(dual price)×Δb, where Δb represents the change in the right hand side value (b_{new})–($b_{original}$).
    - If it is a minimization problem: $Z_{new} = Z_{original}$–(dual price)×Δb.
- Getting the values of the variables is harder. If the solver allows you to recover the "basis inverse" matrix, denoted by $\mathbf{B}^{-1}$, then you can calculate the matrix product $\mathbf{B}^{-1}\mathbf{b}_{new}$, where $\mathbf{b}_{new}$ denotes the new right hand side vector. This will give the values of the basic variables. Or, knowing the basis, you can solve the matrix obtained by setting the nonbasic variables to zero.

This method of finding the new Z works because the dual price is the objective function coefficient for the slack variable associated with the changed constraint. It thus gives the change in Z per unit change in the slack variable for the constraint, and the change in the slack variable is identical to any change in the right hand side.

As an example, let us consider whether the Acme Bicycle Company solution is sensitive to a change in the availability of the metal finishing machine from 4 hours per day to 3.5 hours per day (this is a change to the right hand side of the 3rd constraint, row 4 in the LINDO tables). The tables show that this decrease of ½ unit in the right hand side of the 3rd constraint is within the allowable range. Hence the basis will remain unchanged. The objective function value in this maximization problem will change as follows: $Z_{new} = 50+10\times(3.5–4) = 45$. We will not recover the new values of the variables. So the solution is sensitive to the proposed change in the availability of the metal finishing machine.

Is the Acme Bicycle Company solution sensitive to a change in the right hand side of the 2nd constraint from $x_2 \le 3$ to $x_2 \le 4$? From the tables, this proposed change is within the allowable range, and the constraint dual price is zero (constraint is not active), so no, the solution is not sensitive to this change: the Z and variable values will be unaffected.

Buying Extra Resources

As mentioned above, the right hand side often represents the amount of a given resource that is available. Given that you are currently limited to a maximum amount b of some resource, how much would you willingly pay for an extra unit of b if you could get it? This question often comes up when you observe that any possible improvement in your objective function value is being limited by the availability of some resource. For example, suppose that the available pool of labour is limiting your increase in profits: how much would you pay per unit of extra labour (e.g. by paying overtime, or by hiring temporary workers)? If extra labour can be had cheaply, then you may be able to improve your optimum Z considerably, but if extra labour is expensive, then paying for it may in fact worsen your optimum Z. The idea is to determine the upper limit on how much to pay before you have a worsening effect on the objective function value.

Suppose that the Acme Bicycle Company could rent time on another metal finishing machine for $8 per day: would this be a good idea? Here is how the reasoning goes:

- Acme currently pays nothing per hour for the metal finishing machine because they own it.
- The dual price of the metal finishing constraint is $10 from the LINDO solution. This means that each unit increase in the right hand side of the metal finishing machine constraint (3rd constraint) increases Z by $10.
- Hence, we could pay up to $10 for an extra hour of metal finishing machine capacity and still increase our overall profit.

So, yes, we would happily pay $8 for another hour's worth of use of a metal finishing machine.

The general rule is that you would pay up to (price you are paying now for the resource)+(dual price) for each extra unit of a resource. Note, though, that this analysis only holds if the amount of extra resource that you are planning to buy remains within the allowable range as shown in the tables. Why? Because buying extra resources is the same as changing a right hand side coefficient. If you buy too much of the extra resource, then the basis changes and you are no longer sure if the purchase at the given price is a good idea.

For example, consider buying extra units of capacity for making mountain bikes (i.e. a change to the right hand side of the 1st constraint). In the LINDO tables above, we see that the dual price of the 1st constraint (row 2 in the table) is \$5 per unit. Thus if offered the chance to buy extra units at a price of \$3 per unit, Acme would accept. But how many units should they buy? The ranging tables show that the maximum increase in the right hand side of the 1st constraint is just two units before the basis changes. You can see why in Figure 6.4.

Suppose that Acme was offered extra units of racer production at a bargain rate of \$1 per unit. Should they accept? This is equivalent to a change in the right hand side of the 2nd constraint, and the tables above show that the dual price of the 2nd constraint is 0. Hence Acme should *not* buy any extra units of this resource. Look again at Figure 6.3 to see why. The 2nd constraint is not limiting production now, so a slight increase or decrease in its right hand side value has no impact on the optimum solution. There is no point in buying any extra racer production capacity.

A Note on Terminology

As I've defined it here, a model is "sensitive" if a proposed change to an original coefficient causes a change to the optimum objective function value or variable values. These are the outcomes that materially affect the use of the solution in practice. However, to some authors, a model is "sensitive" only if the basis will be changed by the proposed change to a coefficient. As we have seen, the objective function value and the variable values can both change even when the basis does *not* change.

So beware of miscommunication when talking with others about the sensitivity of a linear program!

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 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.

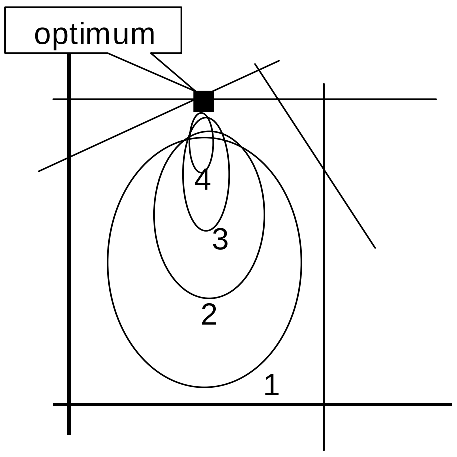


Figure 7.1: Illustration of Khatchian's ellipsoid method.

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 $3.8\times10^2 + 0.000012\times10^2$. 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 often face 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.

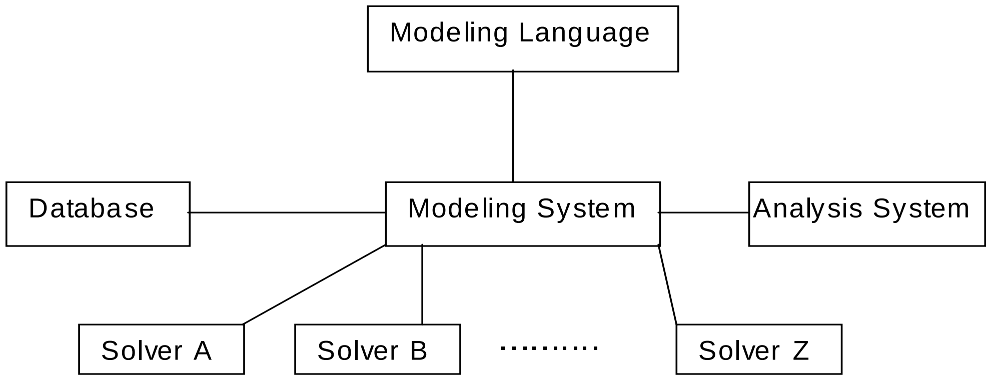


Figure 7.2: Schematic view of a typical modeling system.

Chapter 8: An Introduction to Networks

Network models are an extremely important category of mathematical program that have numerous practical applications. Part of their appeal is the direct and intuitive mapping between the real world, the network diagram, and the underlying solution algorithms. Many network problems can be solved via linear programming, and in fact, special extremely fast variants of linear programming can be applied. The largest mathematical programs that are regularly solved in practice, e.g. airline crew scheduling problems, are usually network problems.

For many types of network problems there are also specialized non-LP solution algorithms. We will first look at some of the classic non-LP solution methods, and later return to the idea of solving networks via LP.

Basic Definitions

Network models are created from two major building blocks: *arcs* (sometimes called *edges*), which are connecting lines, and *nodes*, which are the connecting points for the arcs. A *graph* is a structure that is built by interconnecting nodes and arcs. A *directed graph* (often called a *digraph*) is a graph in which the arcs have specified directions, as shown by arrowheads. Finally, a *network* is a graph (or more commonly a digraph) in which the arcs have an associated *flow*. Some example diagrams are given in Figure 8.1.

Here are some simple examples of networks:

nodes	**arcs**	**flow**
cities	highways	vehicles
call switching centers	telephone lines	telephone calls
pipe junctions	pipes	water

There are some further definitions associated with graphs and networks:

chain: a sequence of arcs connecting two nodes *i* and *j*. For example, in Figure 8.1(a), we might connect nodes A and E via the chains ABCE or ADCE.

path: a sequence of directed arcs connecting two nodes. In this case, you must respect the arc directions. For example, in Figure 8.1(b), a path from A to E might be ABDE, but the chain ABCE is *not* a path because it traverses arc BC in the wrong direction.

cycle: a chain that connects a node to itself without any retracing. For example, in Figure 8.1(a), ABCEDA is a cycle, but ABCDECBA is *not* a cycle because of the double traversal of arcs AB and BC.

connected graph (or connected network): has just one part. In other words you can reach any node in the graph or network via a chain from any other node. It is sometimes important to know whether a graph is connected and there are efficient computer algorithms for checking this.

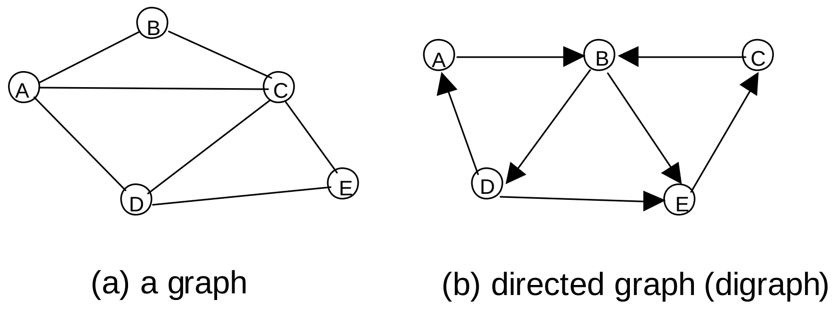


Figure 8.1: Graphs and directed graphs.

tree: a connected graph having no cycles. Some examples are shown in Figure 8.2(a).

spanning tree: normally a tree selected from among the arcs in a graph or network so that all of the nodes in the tree are connected. See Figure 8.2(b). Spanning trees have interesting applications in services layout, for example, finding a way to lay out the computer cable connecting all of the buildings on a campus (nodes) by selecting from among the possible inter-building connections (arcs).

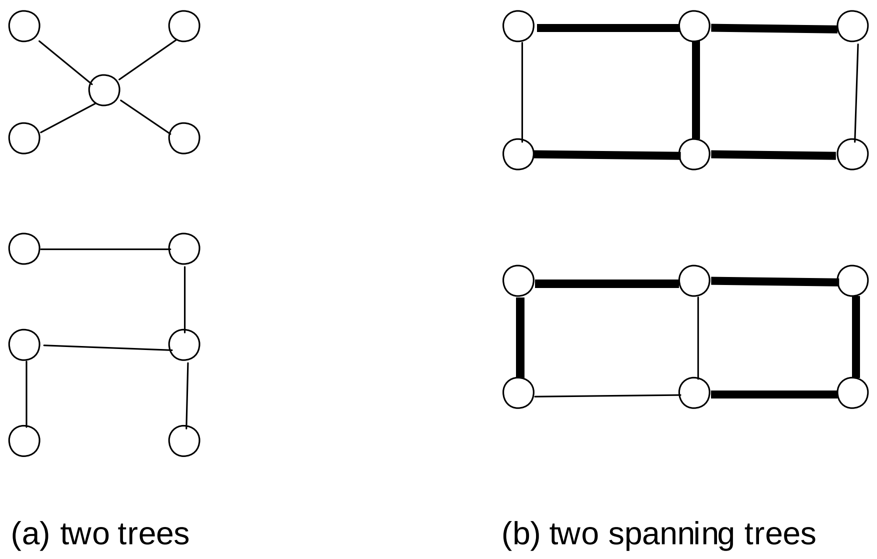


Figure 2: Examples of trees and spanning trees.

flow capacity: an upper (and sometimes lower) limit on the amount of flow in an arc in a network. For example, the maximum flow rate of water in a pipe, or the maximum simultaneous number of calls on a telephone connection.

source (or source node): a node which introduces flow into a network. This happens at the boundary between the network under study and the external world.

sink (or sink node): a node which removes flow from a network. This happens at the boundary between the network under study and the external world.

The Shortest Route Problem

Here is a technical statement of the shortest route problem: given a graph in which each arc is labeled with the distance between the two nodes that it connects, what is the shortest route between some node *i* and some other node *j* in the graph? For example, consider the graph in Figure 8.3: what is the shortest route between node A and node H? What is the length of the shortest route?

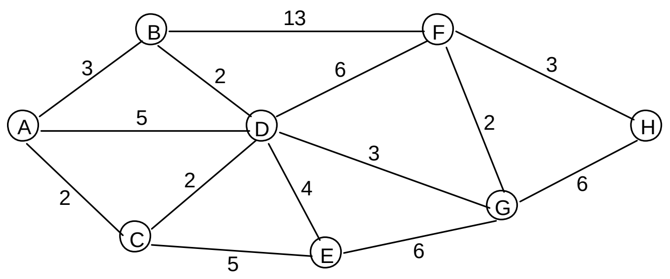


Figure 8.3: Find the shortest route from A to H.

In a graph this small, it is possible to solve the problem by simple inspection: try this for the graph in Figure 8.3. However, if the graph is quite large, a more organized approach is needed. Brute force enumeration of all of the possible routes is impractical in large graphs because the number of possible routes is combinatorially explosive. Try tracing out all of the routes between nodes A and H in Figure 8.3 for example: there are quite a few!

Note that "shortest route problem" is also the name applied when the arc labels are not distances, but perhaps travel time (quickest route problem) or travel cost (cheapest route problem). The solution methods are identical in all cases.

We will look first at one of the most popular methods for finding the shortest route in a graph or network: *Dijkstra's algorithm.* It is an iterative method. At the *n*th iteration you will find the *n*th closest node to the start node (the *origin*), and the shortest route to that node. You stop iterating when the destination node is reached, even if you have not visited all of the nodes in the network.

All of the nodes in the network are initially "unsolved". A node is "solved" at each iteration when the shortest distance and route to that node is found. The length of the shortest path from the origin to a node is the "distance": a node is solved when the distance is determined. By definition (and common sense), the distance from the origin node to the origin node is zero. Arcs which may form part of the shortest route are gradually added to the "arc set" as the method proceeds; the arc set is initially empty. Note that the method ventures up some blind alleys as it proceeds: not all of the arcs that are added to the arc set will feature in the final shortest route!

Here is a statement of Dijkstra's algorithm:

0. To initialize:
    a. the origin is the only solved node.
    b. the distance to the origin is 0.
    c. the arc set is empty.
1. Find all the unsolved nodes that are directly connected by a single arc to any solved node. For each unsolved node, calculate all of the "candidate distances" (there may be several of these for one unsolved node because it may directly connect to more than one solved node):
    a. choose an arc connecting the unsolved node directly to a solved node.
    b. the "candidate distance" is (distance to solved node) + (length of arc directly connecting the solved and the unsolved node)
2. Choose the <u>smallest</u> candidate distance:
    a. add the corresponding unsolved node to the solved set.
    b. distance to the newly solved node is the candidate distance.
    c. add the corresponding arc to the arc set.
3. If the newly solved node is <u>not</u> the destination node, then go to step 1. Else, stop and recover the solution:
    a. length of shortest route from origin to destination is the distance to the destination node.
    b. shortest route is found by tracing backwards from the destination node to the origin (or the reverse) using the arcs in the arc set. It is usually easiest to trace backwards because each node is reached from exactly one other node, but may have outward arcs to several other nodes.

You can keep track of the progress of the algorithm by constructing a table with headings such as iteration number, unsolved node connected to solved node, candidate distances and arcs, selected node, distance and arc. However, for relatively small problems it is easier to keep track by making notations directly on the diagram. We will use the following conventions for tracking the algorithm on a diagram:

- mark the solved nodes in boldface and label them with the distance,
- mark the arcs in the arc set in boldface.

Let us follow the progress of Dijkstra's algorithm as we apply it to the network shown in Figure 8.3. We will omit the initial diagram in which only node A, the origin, is solved, and so shown in boldface and labeled with the distance 0. At this point, there are 3 unsolved nodes to consider (B, C, and D) because they are directly connected to the only solved node (A). The candidate distances are AB (0+3=3), AC (0+2=2), and AD (0+5=5). The smallest candidate distance is AC, so C is now a solved node with a distance of 2 as shown in Figure 8.4.

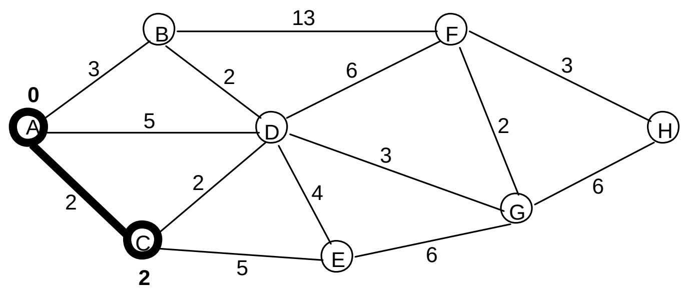


Figure 8.4: First closest node to A is C.

Now there are 3 unsolved nodes to consider (B, D, and E). Candidate distances are AB (0+3=3), AD (0+5=5), CD (2+2=4), and CE (2+5=7). The smallest candidate distance is AB at 3, so B is now a solved node, as shown in Figure 8.5.

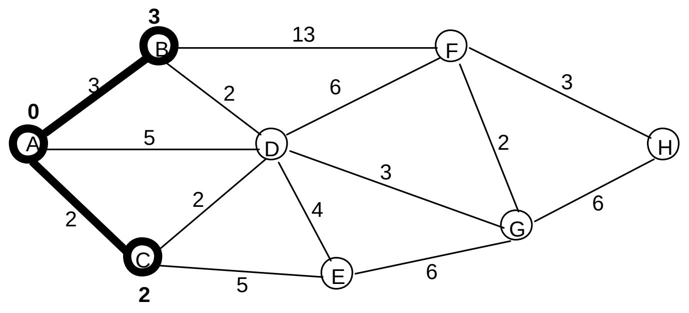


Figure 8.5: Second closest node to A is B.

There are again 3 unsolved nodes to consider (F, D, and E). Candidate distances are BF (3+13=16), BD (3+2=5), AD (0+5=5), CD (2+2=4), and CE (2+5=7). The smallest candidate distance is CD at 4, so D is now a solved node, as shown in Figure 8.6.

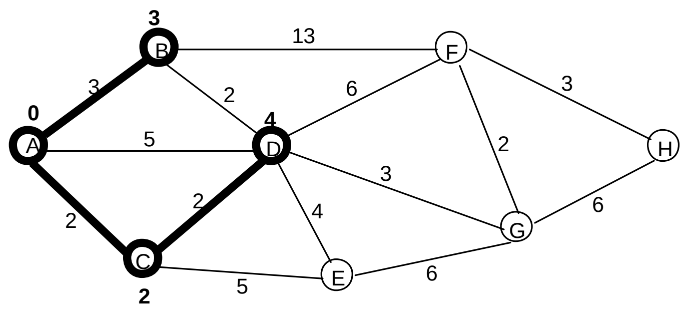


Figure 8.6: Third closest node to A is D.

Now the unsolved nodes to consider are F, G, and E. Candidate distances are BF (3+13=16), DF (4+6=10), DG (4+3=7), DE (4+4=8), and CE (2+5=7). The smallest candidate distance is 7 with a

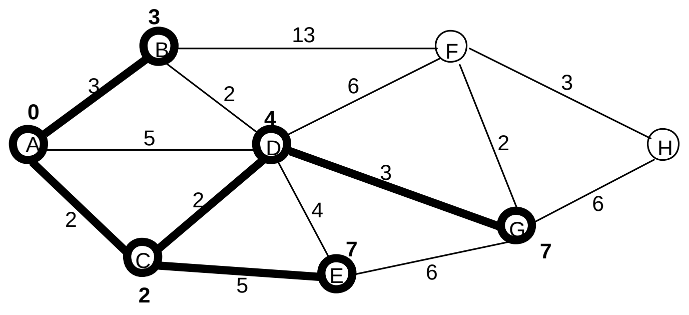


Figure 8.7: 4th and 5th closest nodes to A are E and G (tied).

tie between DG and CE. We can choose either candidate arbitrarily and update the diagram accordingly, however we know that the candidate not chosen this time will definitely be chosen in the next iteration, so we might as well update both candidates at once, as shown in Figure 8.7.

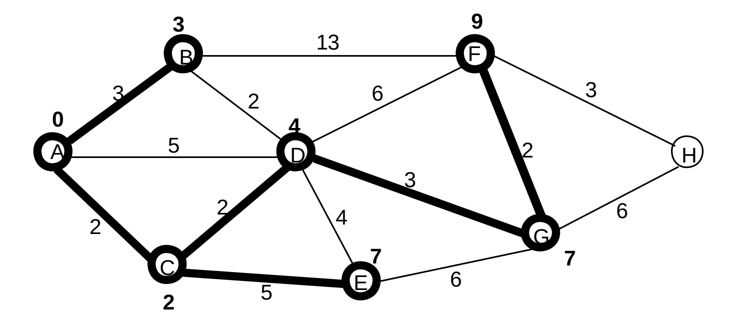


Figure 8.8: 6th closest node to A is F.

Now there are only two unsolved nodes to consider: F and H. The candidate distances are BF (3+13=16), DF (4+6=10), GF (7+2=9), and GH (7+6=13). The smallest candidate distance is 9, so F is now a solved node, as shown in Figure 8.8.

Finally there is only one remaining unsolved node: H. The candidate distances are FH (9+3=12) and GH (7+6=13). So H becomes a solved node with a distance of 12, as shown in Figure 8.9. Since H is the destination node, the iterations stop. We visited every node in the network in this small example, but this will not always happen.

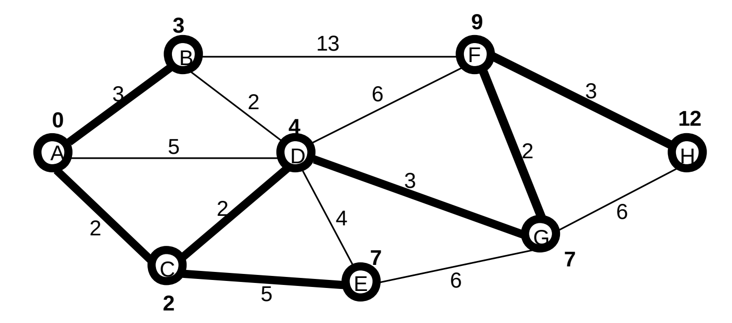


Figure 8.9: 7th closest node to A is H. Finished!

At this point, we know that the length of the shortest route from A to H is 12 units (kilometers perhaps). But how do we determine the route? Simply trace backwards from H to A, and then reverse the route that you find in this manner. In Figure 8.9, the reverse route is HFGDCA, and when given in the forward order this is ACDGFH. The reason for finding the route in this manner is that working backwards avoids the blind alleys that exist in the forward direction.

Note that you may sometimes find a tie in the routing to an intermediate node. For example, if arc BD had a length of 1, then there would be two routes from A to D with length 4: ACD as selected in the diagram, and ABD. It doesn't matter which one you use. In fact, you can mark both choices on the diagram as you proceed, which will remind you of the alternate routes.

What makes Dijkstra's algorithm efficient is that it calculates the lengths of only a very small subset of all of the possible routes through the graph. Once a node has been solved, the shortest path to the node from the origin is known. All paths that follow on from that node then have the benefit of the shortest path to that point. The method is actually a specific implementation of a dynamic programming algorithm (covered later).

It is easy to modify Dijkstra's algorithm for use on directed graphs. You simply need to respect the arc directions when selecting candidate connections between solved and unsolved nodes. Think of it as finding the shortest route through a system of one-way streets.

The Minimum Spanning Tree Problem

The technical statement of the minimum spanning tree problem is simple: given a graph in which the arcs are labeled with the distances between the nodes that they connect, find a spanning tree which has the minimum total length. Recall that a spanning tree connects all of the nodes in the graph, and has no cycles.

As for the shortest route problem, the arc labels could as well be related to time or cost. There are many examples of applications of the minimum spanning tree problem:

- Find the least cost set of roadways among the possible set of roadways to connect a set of locations.

- Find the shortest total length of sewer to lay among the buildings in a planned subdivision, given the set of possible inter-building sewer routes.

- Find the minimum total length of telephone cable to connect all of the offices in a building, given the possible routings of cable between offices.

The algorithm for solving this problem is probably the simplest that you will ever learn in mathematical programming:

0. Initialize: all nodes are unsolved, no arcs are in the arc set.

1. Select any node arbitrarily and label it as solved. Connect this node to its nearest neighbour: label the neighbour solved and add the connecting arc to the arc set.

2. Identify the unsolved node that is closest to a solved node. Label this node as solved and add the connecting arc to the arc set.

3. If all nodes are solved, then stop. Else go to Step 2.

Upon termination of the algorithm, the minimum spanning tree is given by the arcs in the arc set, and the length of the minimum spanning tree is found by summing the lengths of the arcs in the arc set.

In the case of a tie for the next solved node, choose arbitrarily. You will obtain the same final optimum solution, but it indicates that there are multiple optima. You will need to go back and enumerate them all if you need them.

There is again a simple diagrammatic convention for keeping track of the steps of the algorithm: show the solved nodes and the arcs in the arc set in boldface as you proceed. Let us apply this convention while solving the minimum spanning tree problem for the graph in Figure 8.3 that we already used for the shortest route problem. Let us begin arbitrarily at node E, and label the nodes with the order in which they are solved, while keeping track of all of the notations on a copy of the graph in Figure 8.10.

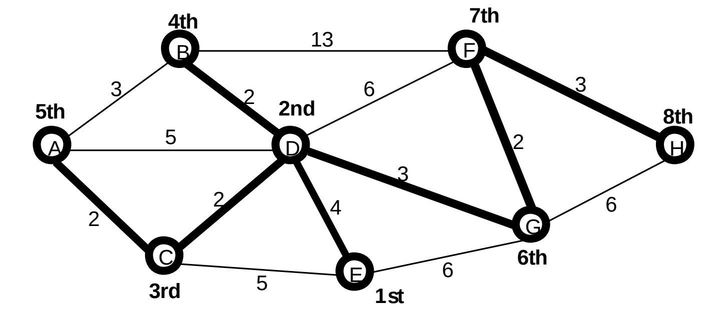


Figure 8.10: Finding the minimum spanning tree.

The closest node to node E is node D at a distance of 4 units, so node D is the 2nd solved node.

Candidate unsolved nodes are now B (BD=2), A (AD=5), C (CD=2, CE=5), F (FD=6), and G (GD=3, GE=6). The smallest connecting length is 2 (CD or BD), so we choose CD arbitrarily. Node C is the 3rd solved node.

Nodes E, D, and C are now solved. Candidate unsolved nodes are A (AC=2, AD=5), B (BD=2), F (FD=6), and G (GD=3, GE=6). The smallest connecting length is 2 (AC or BD), so we choose BD arbitrarily. Node B is the 4th solved node.

Nodes E, D, C and B are now solved. Candidate unsolved nodes are A (AB=3, AD=5, AC=2), F (FB=13, FD=6) and G (GD=3, GE=6). The smallest connecting length is 2 (AC), so this is chosen. Node A is the 5th solved node.

Nodes E, D, C, B and A are now solved. Candidate unsolved nodes are F (FB=13, FD=6), and G (GD=3, GE=6). The smallest connecting length is 3 (GD), so node G is the 6th solved node.

Nodes E, D, C, B, A and G are now solved. Candidate unsolved nodes are F (FB=13, FD=6, FG=2), and H (HG=6). The smallest connecting length is 2 (FG), so F is the 7th solved node.

Finally, all nodes are solved except node H. The candidate connecting lengths are HF=3 and HG=6. HF is chosen, so H is the 8th and last solved node.

Now the solution can be recovered. The minimum spanning tree itself is *all* of the arcs in the arc set (i.e. all of the arcs shown in bold in Figure 8.10). Note how this differs from the shortest route solution in which not all of the arcs in the arc set are part of the final solution. The total length of the minimum spanning tree is found by summing the lengths of all of the arcs in the arc set: ED + DC + DB + CA + DG + GF + FH = 4 + 2 + 2 + 2 + 3 + 2 + 3 = 18. The total length of the minimum spanning tree is 18 units.

Note that you will get the identical total of 18 units no matter which node you choose as the initial node (try it yourself). There is more than one spanning tree that gives this same result, however. We know this because of the arbitrary choice we made in solving the 3rd node.

The solution method for the minimum spanning tree problem is an example of a *greedy algorithm*. A greedy algorithm does whatever is best at the current step, without ever considering what the impact might be on the overall problem. This is usually a bad idea in optimization because it leads to a solution that is less than optimal overall. However, just choosing the closest unsolved nodes leads to an overall optimum solution in this special case.

The algorithm for a *maximum* spanning tree is obvious: simply choose the *longest* solved-to-unsolved node connection at each step. You might want a maximum spanning tree in a case where profit is involved, for example, choosing television cable routings to connect a set of locations. The extension to directed graphs is also straightforward: you can only select from among arcs that connect in the direction that you are working (either away from the initial node, or towards the initial node).

Chapter 9: Maximum Flow and the Minimum Cut

A common question about networks is "what is the maximum flow rate between a given node and some other node in the network"? For example, traffic engineers may want to know the maximum flow rate of vehicles from the downtown car park to the freeway on-ramp because this will influence their decisions on whether to widen the roadways. Another example might be the maximum number of simultaneous telephone calls between two cities via the various land-lines, satellites, and microwave towers operated by a telephone company.

An infinite flow rate is impossible because the individual roads or telephone links have limited capacities to carry flow. On the other hand, there are usually multiple ways to drive between the downtown car park and the freeway on-ramp, or to route calls between two cities. Finding the maximum flow involves looking at *all* of the possible routes of flow between the two end-points in question. When the system is mapped as a network, the arcs represent channels of flow with limited capacities. To find the maximum flow, assign flow to each arc in the network such that the total simultaneous flow between the two end-point nodes is as large as possible.

A further wrinkle is that the flow capacity on an arc might differ according to the direction. For example, a particular road might have two lanes in the A to B direction, but only one lane in the B to A direction. Or it might be simply a one-way street, with no flow in the B to A direction at all. In Figure 9.1, the labels on the arcs indicate the flow capacities in both directions. For example, the label near node A on the arc A-B indicates the flow capacity in the A-to-B direction, while the label near node B on the arc A-B indicates the flow capacity in the B-to-A direction.

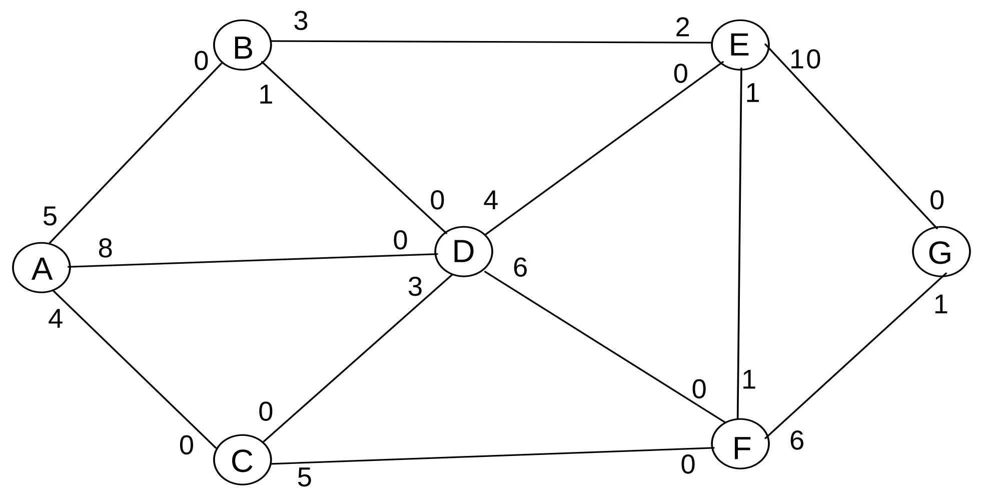


Figure 9.1: Find the maximum flow from A to G.

Let's imagine that the flows in Figure 9.1 are in units of vehicles per minute. Here are some examples of routes on which flow could travel from node A to node G:

- 4 vehicles per minute along the route A-D-E-G. This is the maximum flow on this route because of the bottleneck on arc D-E.
- 3 vehicles per minute along the route A-B-E-G. The bottleneck here is the arc B-E. Note, though, that with the simultaneous flow on route A-D-E-G, the total flow on arc E-G is now 7 vehicles per minute.
- 4 vehicles per minute along the route A-C-F-G. The bottleneck on this route is arc A-C.

This set of flows gives a total flow of 4 + 3 + 4 = 11 vehicles per minute from A to G. But it is not the *maximum* flow attainable from A to G. There remain unused routes for carrying flow between the two end nodes. We need an organized method of tabulating the routes and flows, and this is provided by the Ford and Fulkerson method. As we will see later, the maximum flow problem can be solved by linear programming, but the Ford and Fulkerson method is simple and even faster than linear programming when implemented on a computer. Ford and Fulkerson first published their method in the *Canadian Journal of Mathematics* in 1956 – it is a real classic paper, very often referenced to this day.

The main idea is careful bookkeeping of the flows assigned to different routes from the origin node to the destination node. The steps in the method are:

1. Find *any* path from the origin node to the destination node that has a strictly positive flow capacity remaining. If there are no more such paths, exit.

2. Determine *f*, the maximum flow along this path, which will be equal to the smallest flow capacity on any arc in the path (the bottleneck arc).

3. Subtract *f* from the remaining flow capacity in the forward direction for each arc in the path. Add *f* to the remaining flow capacity in the backwards direction for each arc in the path.

4. Go to Step 1.

On termination, the sum of the flows along the paths found during Step 1 gives the maximum total flow between the origin and destination nodes.

Let's try the Ford and Fulkerson method on the network in Figure 9.1. The results are shown in Figures 9.1 through 9.6. Figures 9.2 through 9.4 show the three flow paths suggested earlier, and Figures 9.5 and 9.6 show two more flow paths that can be added before we are unable to find a path that can support a strictly positive flow *f*. Note the bookkeeping on the flow capacities as the solution progresses, and how it becomes more and more difficult to find a path having positive flow capacity.

The algorithm terminates after the last path is found in Figure 9.6. No more strictly positive flow paths can be found between A and G. This is obvious since all paths must pass through the set of arcs B-E, D-E, F-E, and F-G, and these arcs have all had their flow capacities in the forward direction reduced to zero.

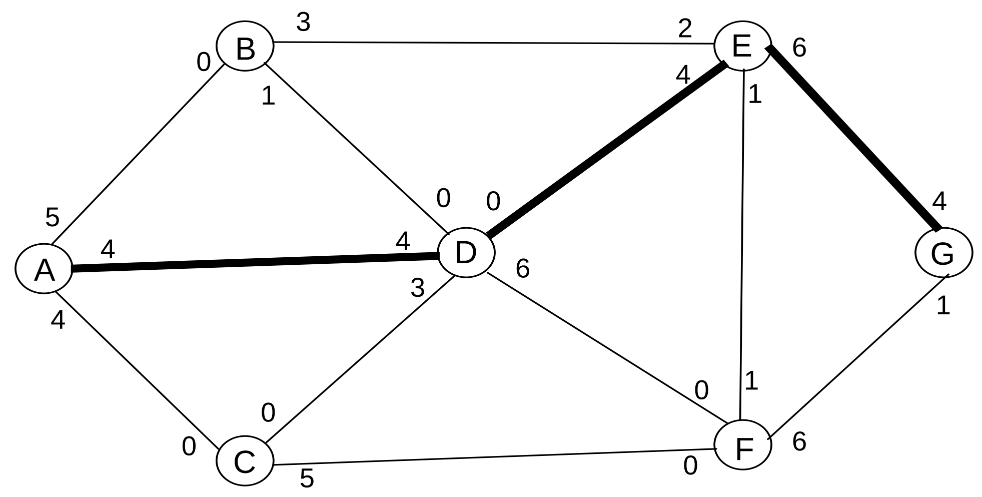


Figure 9.2: 4 vehicles per minute along A-D-E-G.

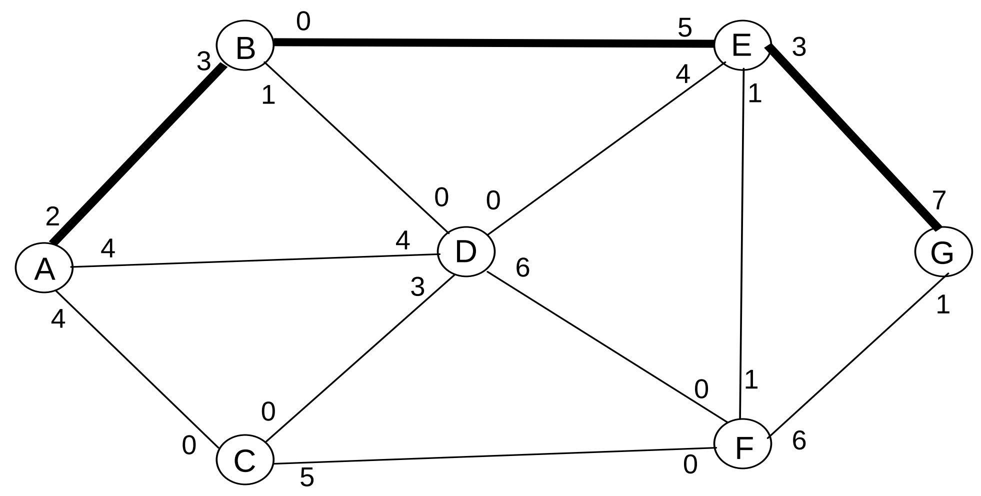


Figure 9.3: 3 vehicles per minute along A-B-E-G.

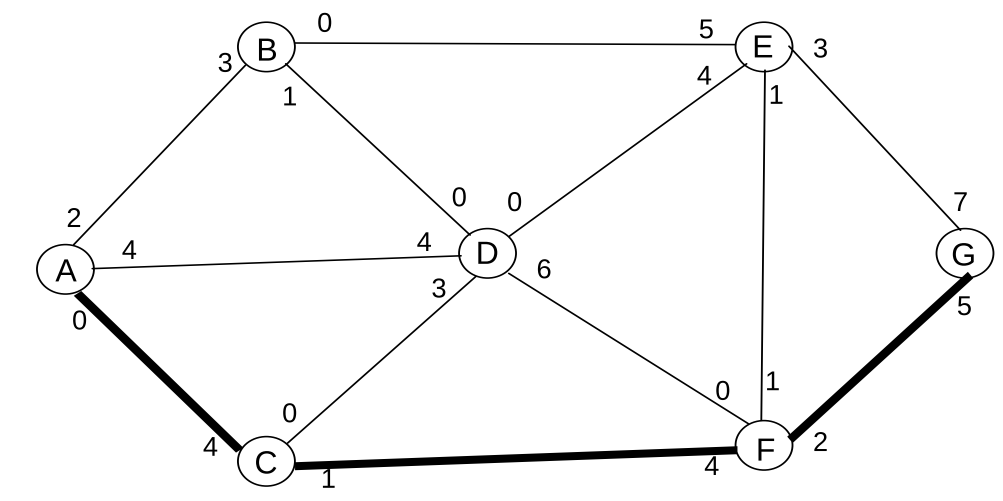


Figure 9.4: 4 vehicles per minute along A-C-F-G.

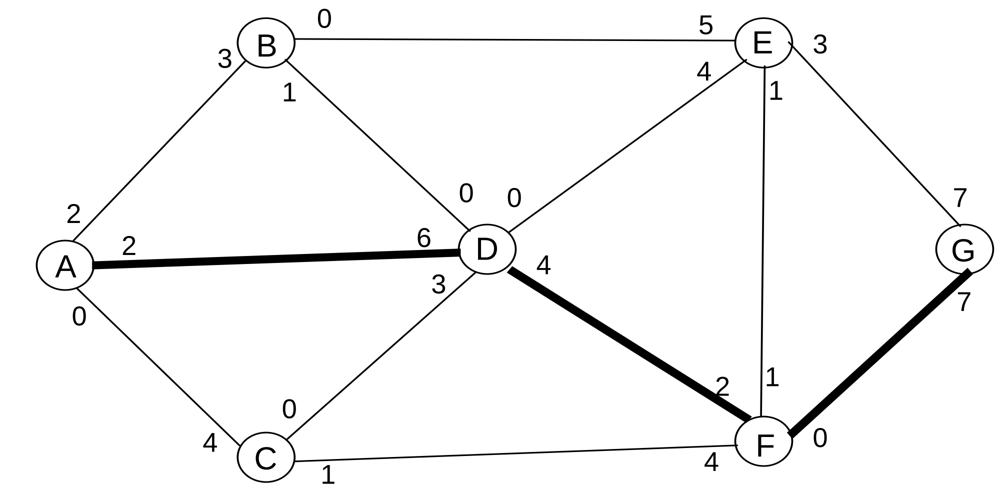


Figure 9.5: 2 vehicles per minute along A-D-F-G.

When the algorithm terminates, the maximum total simultaneous flow of vehicles from A to G is given by summing the flows on the 5 paths we selected: 4 + 3 + 4 + 2 + 1 = 14 vehicles per minute.

But what is the actual *pattern* of flows that gives this optimum? How much flow should go on each arc, and in which direction? This is found by looking at the difference between the initial flow capacity and the final flow capacity: a positive difference indicates a flow in the associated

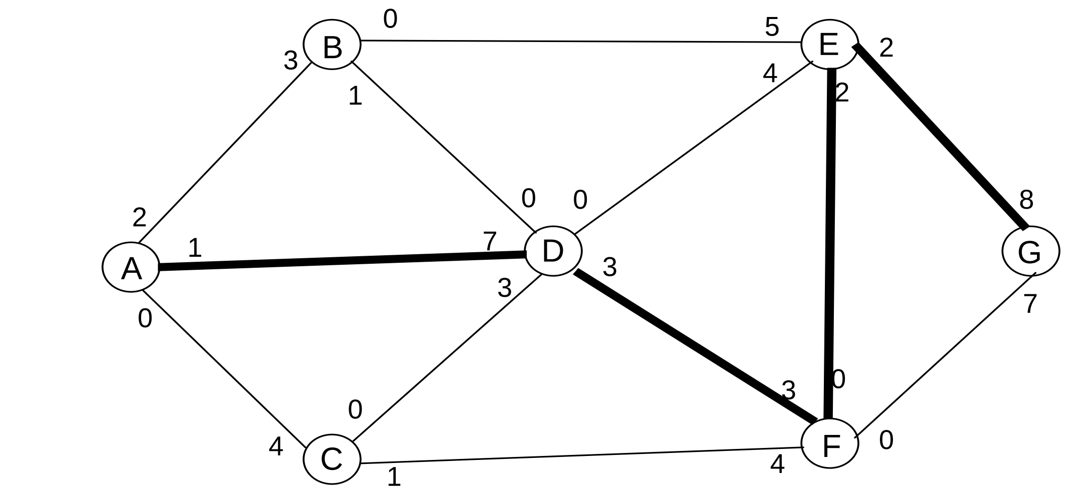


Figure 9.6: 1 vehicle per minute along A-D-F-E-G.

direction (a negative difference is ignored). The pattern of flows – and their directions – which gives the maximum simultaneous flow of vehicles per minute, is shown in Figure 9.7. The arc labels in Figure 9.7 show the amount of flow in each arc. Note that the principle of flow conservation at a node is respected. For example, the flows entering node F total 7 vehicles per minute, as do the flows leaving node F.

You probably noticed that it becomes harder and harder to find a strictly positive flow path as the algorithm progresses and all the easy-to-spot paths are used up. You might think this would be a

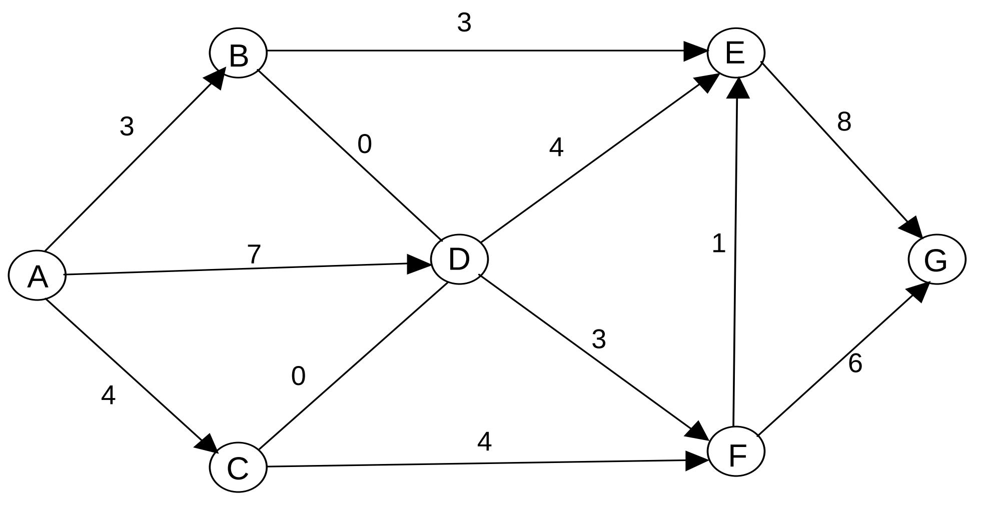


Figure 9.7: Pattern of flows yielding the maximum flow.

problem in a computer implementation of the method, but it turns out that simple depth-first and breadth-first searches are quite efficient for finding positive flow paths.

Students often ask why the Ford and Fulkerson algorithm bothers to update the flow capacities in the backwards directions on the arcs. This is because the backwards capacities that are added are a bookkeeping convention to indicate flow that can be undone if needed. This did not happen in our example, but Figure 9.8 shows a simple example in which the backwards capacities are used in reaching a larger total flow. As you see, after the first path is chosen, the only way for the second path to route more flow from A to B is by *undoing* the flow placed on the vertical arc by the first path. The resulting flow pattern in (d) shows that the vertical arc is not used at all in the final solution.

The maximum flow problem is intimately related to the minimum cut problem. A *cut* is any set of directed arcs containing at least one arc in every path from the origin node to the destination node. In other words, if the arcs in the cut are removed, then flow from the origin to the destination is completely cut off. The *cut value* is the sum of the flow capacities in the origin-to-destination direction over all of the arcs in the cut. The minimum cut problem is to find the cut that has the minimum cut value over all possible cuts in the network. Some possible cuts are illustrated in Figure 9.9; each cut is labeled with the cut value.

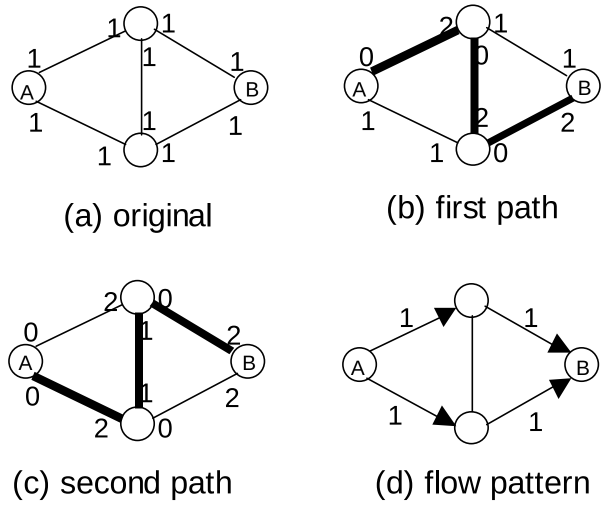


Figure 9.8: Why backwards flow capacities are updated.

One terribly inefficient way to find the minimum cut is to simply try all possible cuts and select the smallest. However the number of possible cuts is extremely large, and it is impossible to enumerate all possible cuts in a larger network, even with extremely fast computers and a lot of computing time.

A better approach is to make use of the *max-flow / min-cut theorem*: for any network having a single origin node and a single destination node, the maximum possible flow from origin to destination equals the minimum cut value for all cuts in the network.

This may seem surprising at first, but makes sense when you consider that the maximum flow through a series of linked pipes equals the maximum flow in the smallest pipe in the series, i.e. the flow is limited by the bottleneck pipe, as illustrated in Figure 9.10. The minimum cut is just

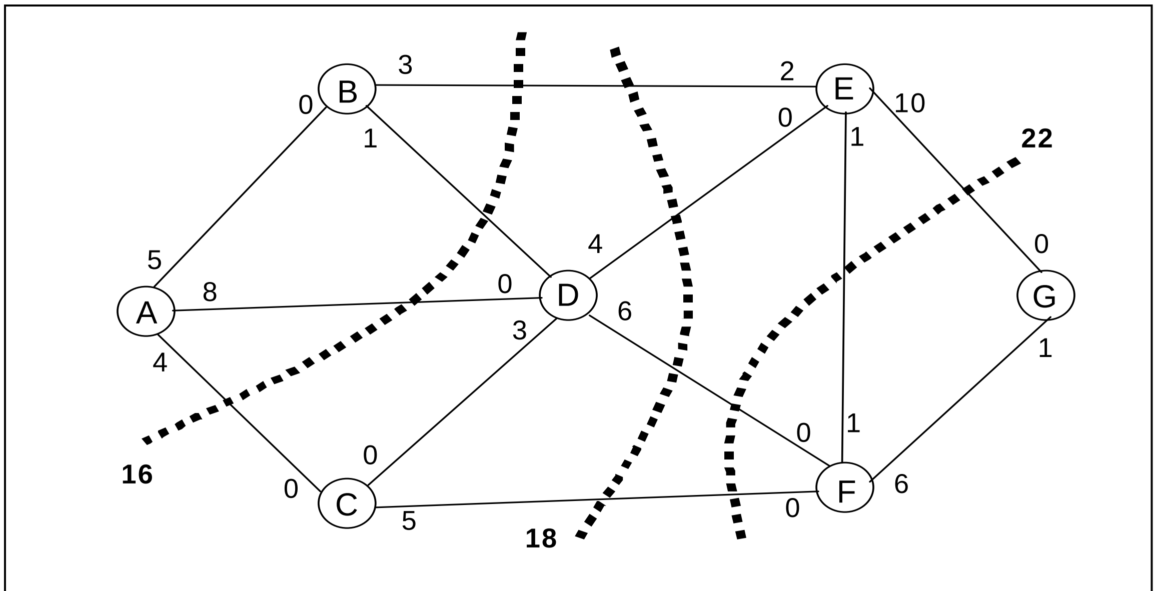


Figure 9.9: Some possible cuts.

a kind of *distributed* bottleneck; i.e. a bottleneck for a whole network as opposed to a simple bottleneck for a series of pipes.

The max-flow / min-cut theorem means that we can determine the minimum cut value using the Ford and Fulkerson maximum flow algorithm. But the real question is *where* the minimum cut is located. If our traffic engineers determine that the maximum flow rate of vehicles from the car park to the freeway on-ramp of 14 vehicles per minute is too small to handle peak rush hour traffic, then we are going to want to expand the roadways. But which ones? There is no point in expanding roadways that are carrying less than their maximum capacity. The only roadways to expand are those that are part of the bottleneck, i.e. the minimum cut.

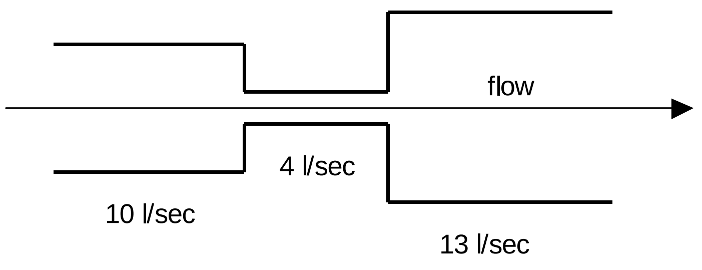


Figure 9.10: The maximum flow through this series of pipes is 4 l/sec.

The minimum cut is actually simple to find after the Ford and Fulkerson algorithm has been completed. Simply mark the arcs that are carrying a flow equal to their maximum flow capacity and look for a cut that consists *only* of marked arcs and no other arcs. It often happens that not all of the marked arcs are used in the cut, and it may also happen that there are multiple cuts that can be made in this way. The results for our vehicle flow example are shown in Figure 9.11. Not surprisingly, the cut is the same set of arcs whose flow capacities were forced to zero by the Ford and Fulkerson algorithm, hence forcing the termination: B-E, D-E, F-E, and F-G. The cut value in the forward (origin-to-destination) direction is 14 for these arcs, the same as the maximum flow. These four arcs are the bottleneck for the network. These are the roads that the traffic engineers should consider widening to increase the flow from the car park to the freeway on-ramp.

Keep in mind, though, that you may not get one unit of increase in the maximum flow for every unit of added capacity in an arc from the minimum cut. This is because the increased flow through that arc may cause another bottleneck to come into play upstream or downstream from that arc.

There are many applications that make use of the minimum cut, including finding the bottlenecks in traffic applications as we have shown here, or in telecommunications networks, production lines, etc. It's interesting to note that the algorithms for finding the max flow and the min cut do not require that the associated labels be flow capacities. Suppose the flow capacity labels actually represented costs, e.g. costs of building dams over various tributaries of a river system. Then the min cut would actually show the minimum cost of completely damming the water flow in the river network. In a military application during the Vietnam war, the infamous Ho Chi Minh trail, actually a network of trails, was modeled as a network, and estimated "costs" were assigned to cutting the trail at various points. The min cut then showed the set of trail segments that should be attacked in an attempt to cut the flow of enemy men and supplies at the least "cost".

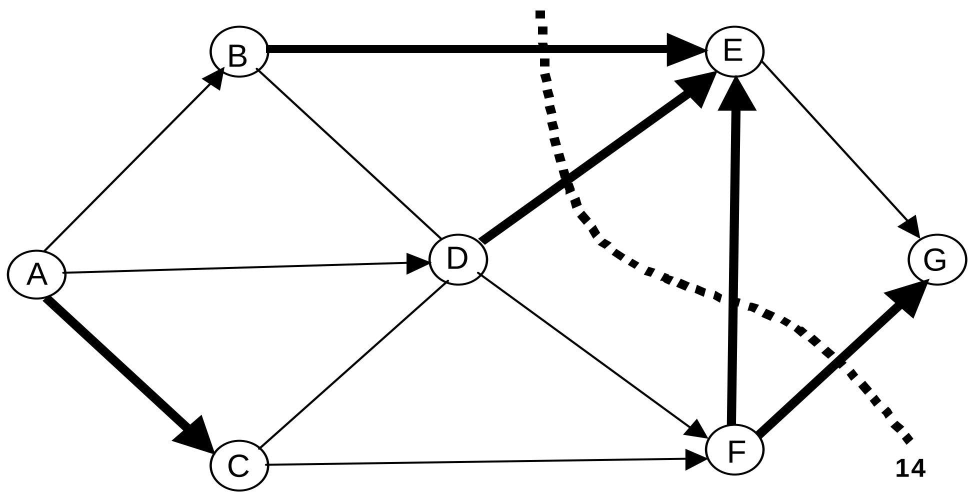


Figure 9.11: Finding the minimum cut.

Chapter 10: Network Flow Programming

Linear programming, that amazingly useful technique, is about to resurface: many network problems are actually just special forms of linear programs! This includes, for example:

- the transportation problem,
- the transshipment problem,
- the assignment problem,
- the maximum flow and minimum cut problem,
- the minimum spanning tree problem,
- the shortest route problem,
- etc.

We will see in this chapter how these problems can be cast as linear programs, and how the solutions to the original problems can be recovered. We will also see that there are specialized algorithms that can solve network linear programs many times faster than if they are solved by the general-purpose simplex method. Formulating and solving network problems via linear programming is called *network flow programming.*

Any network flow problem can be cast as a *minimum-cost network flow program*. A min-cost network flow program has the following characteristics.

Variables. The unknown flows in the arcs, the x_i, are the variables.

Flow conservation at the nodes. The total flow into a node equals the total flow out of a node, as shown in Figure 10.1(a) for example. It makes things easier later if we follow the convention of writing the flow conservation equation at a node as:

$$\sum_{\text{outflows}} x_j - \sum_{\text{inflows}} x_j = 0$$

Source and sink nodes. Some nodes are connections to the environment surrounding the network. At these entryway nodes, there may be a net gain of flow into the network (source node), or a net loss of flow out of the network (sink node). To emphasize that flow conservation still holds at source and sink nodes, a dashed "phantom" arc can be shown on the network diagram. The phantom arc will be an inflow for a source node, and an outflow for a sink node. See the examples in Figures 10.1(b) and 10.1(c).

Now we use a similar convention for writing the flow conservation equation at the source or sink node:

$$\sum_{\text{outflows}} x_j - \sum_{\text{inflows}} x_j = b_i$$

When written this way, b_i is a positive constant for a source node, and a negative constant for a sink node. The magnitude of b_i is the amount of flow in the phantom source or sink arc. Note also that the relationship may not be an equality relationship: inequalities are common for sources and sinks. For example, the flow of water exiting a supply network must be at least 100 liters per minute, or the flow of oil entering a refinery network must not exceed 10,000 barrels per day.

When all of the flow conservation equations (or inequalities) are written following the *outflows – inflows* convention, it is easy to see what type of node is associated with each relationship by looking at the value of the right hand side constant. The constant will be zero for a simple flow conserving node, positive for a source node, and negative for a sink node.

Bounds on the arc flows. There may be upper and lower bounds on the flows in the arcs (i.e. the variables in the model. $x_j \geq b_j$ is a lower bound on an arc flow, and $x_j \leq b_j$ is an upper bound on an arc flow. For example, the maximum flow of water through a particular pipe is limited because of the pipe diameter and interior roughness, so an upper bound is applied: $x_j \leq 25$ *l/sec.*

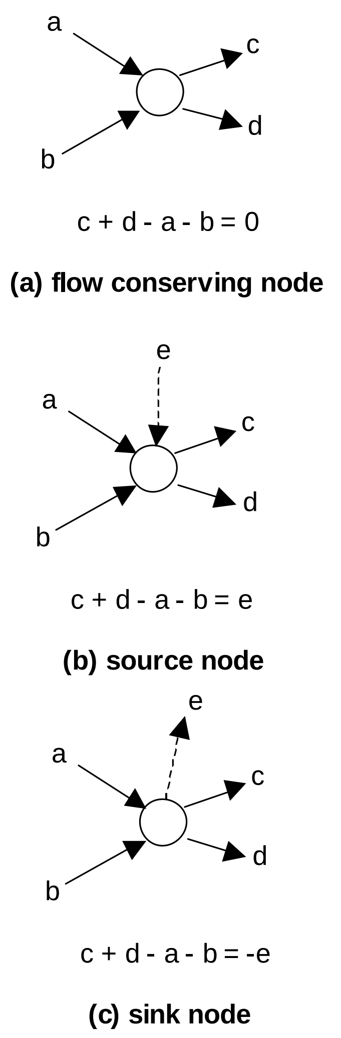


Figure 10.1: Flow conservation at nodes.

Upper bounds are easy to understand, but why might there be a nonzero lower bound? This might represent the minimum required production rate at a factory (e.g. 250 vehicles per day required at the output arc of an automobile plant) or a minimum flow rate through steam piping to prevent condensation.

The default arc flow bounds are a lower bound of zero and no upper bound. It is not unusual for almost all of the arcs in a network model to have the default bounds, with only a few arcs having specified upper or lower bounds, typically those at the "edges" of the network, representing, say, upper limits on the rates of raw materials flowing into the network and lower limits on the production the rate at the main outflow from the network.

Some modeling systems will allow a negative lower bound on an arc flow. My strong advice is *don't do this!* The meaning of a negative flow is a *backwards* flow in the arc. This immediately destroys the best feature of a network model, the intuitive understanding of the system that you gain by looking at the network diagram. You expect the flows to follow the directions indicated by the arrowheads, but the arrowheads may lie if you permit negative flows. There are other ways to accommodate two-way flow, if necessary, such as pairing oppositely oriented arcs between two nodes.

Cost per unit of flow. There is a cost per unit of flow, c_j, associated with each arc. In many network models, the cost per unit of flow is zero for most of the arcs, with costs being typically associated with arcs at the "edges" of the network. The default value of c_j is zero.

Objective function. In a minimum cost network flow problem, the objective is to find the values of the variables (the x_j) that minimize the total cost of the flows over the network:

$$\text{minimize} \sum_{arcs} c_j x_j$$

Of course, the solution must respect all of the constraints: flow conservation at the nodes, and the upper and lower flow bounds on the arcs.

As we will see in the remainder of this chapter, an astonishing array of network problems can be cast as minimum cost network flow programs.

From Network Diagram to Linear Program

A huge attraction of network models is the immediate intuitive understanding provided by the diagram. In fact, given a properly labeled diagram, the conversion to a minimum cost network flow linear program is automatic. There are some commercial modeling systems that support this direct conversion.

There are three parameters associated with each arc: the lower flow bound, the upper flow bound, and the cost per unit of flow. The arc labeling convention that we will use shows a triple of numbers in square brackets, [*l*, *u*, *c*]. *l* is the lower bound on the flow in the arc, with a default value of zero if not explicitly specified; *u* is the upper bound on the flow in the arc, with a default value of infinity if not explicitly specified; *c* is the cost per unit of flow in the arc, with a default value of zero if not explicitly specified. For example, an arc having a lower flow bound of zero, and upper flow bound of 25, and a cost per unit of flow of $6 would be labeled [0, 25, 6].

Source and sink node behavior is controlled by the label on the phantom arc associated with the node. If the upper and lower flow bounds on the phantom arc are identical, then the node relationship is an equation, but if the upper and flow bounds on the arc differ, then the node relationship is an inequality.

Consider the network diagram in Figure 10.2 for example. The phantom arcs on the 3 source and sink nodes are fully labeled. Node A is a source of up to 12 units of flow at a cost of $5 per unit of flow. Node C is a sink of up to 4 units of flow at an *income* of $6 per unit of flow – the negative cost per unit of flow means income. Node D is a sink of exactly 8 units of flow, but with no cost or income associated with that flow. The remaining arcs are also labeled following the convention. Note that arc 4 has a positive lower bound.

Given the fully labeled diagram, writing the associated linear program is a mechanical process: write the minimum cost objective function, then the node conservation equations, the arc flow bounds, and the nonnegativity constraints. The linear program constraints associated with Figure 10.2 are:

node A:	$x_1 + x_2 + x_3 \le 12$
node B:	$x_4 - x_1 = 0$
node C lower bound:	$x_5 - x_2 \ge -4$
node C upper bound:	$x_5 - x_2 \le 0$
node D:	$-x_3 - x_4 - x_5 = -8$
flow bound arc 2:	$x_2 \le 6$
flow bound arc 3:	$x_3 \le 3$
flow bound arc 4:	$x_4 \ge 4$
nonnegativity:	$x_1, x_2, x_3, x_4, x_5 \ge 0$

The minimum cost objective function can be written as:

$$\text{minimize } 5A - 6C + 2.5x_3 + 3.7x_4 + 0.5x_5$$

where A and C represent the nonnegative flows in the phantom arcs associated with nodes A and C. But those two variables are not needed because the phantom arc flows can be rewritten in terms of the other flows incident on (i.e. touching) the associated nodes, as follows:

$$A = x_1 + x_2 + x_3 \text{ and } C = x_2 - x_5$$

These relationships are substituted into the objective function to remove the variables A and C from the model entirely. The final version of the objective function is then:

$$\text{minimize } 5x_1 - x_2 + 7.5x_3 + 3.7x_4 + 6.5x_5$$

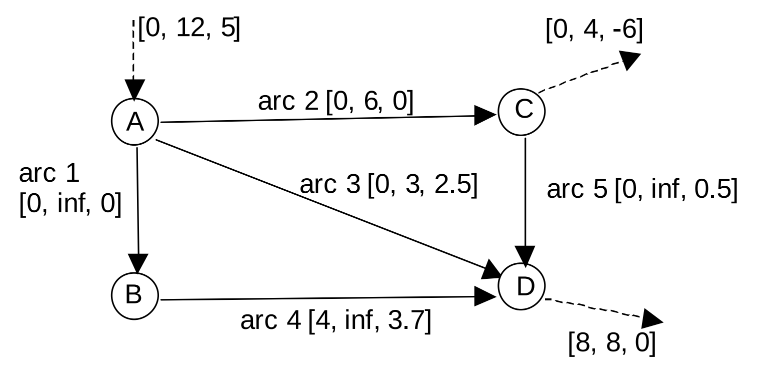


Figure 10.2: A fully labeled network diagram.

As we have seen, the network diagram contains all the information needed to derive an associated linear programming model via a straightforward mechanical writing of the constraints and objective function. For this reason, we will assume from here onward that a properly labeled network diagram *is* the formulation.

The linear programming version of the network model has some very interesting properties. Look at the left hand sides of all of the constraints. What do you notice? All of the coefficients there are either 0, +1, or –1. This is because all of the node relationships are simple summations of flows, and the remainder of the constraints are simple bounds. But this fact has some very important consequences.

The most important consequence is that all of the pivot steps during the linear programming solution consist of simple addition and subtraction: there is no need for multiplication. Now this doesn't seem very dramatic, but at the very least it eliminates floating-point computer operations in favor of much faster arithmetic operations.

But it gets better: those simple addition/subtraction operations can actually be replaced by *logical* operations in a different solution algorithm known as the *network simplex method*, which can be hundreds of times faster than the ordinary simplex method when applied to a network problem. In fact, the solution times for networks are so much faster than regular linear programs that some sophisticated linear programming solvers will scan the LP for network portions. If it finds that parts of the LP are in network form, it has a way of solving those portions separately using the fast network simplex method, and then tying the solution back into the rest of the linear program in an iterative manner. Overall solution time is reduced in this way.

The second important consequence is this: if all of the constraint right hand side values are integers (as in our example), and if all of the pivot operations are simple additions and subtractions, then we can guarantee that the solution values of the variables at the optimum will also all be integers. This is known as the *unimodularity* property. This is extremely useful in solving certain types of integer programming problems, such as assignment problems. Most integer programming problems must be solved using much slower solution algorithms, so it is very fortunate that a fast technique such as linear programming can be used on some problems.

The Transportation Problem

The transportation problem is simple in form, but surprisingly useful in practice. It consists of a set of sources of some product (e.g. factories producing canned vegetables), which are directly linked to sinks of the product (e.g. markets in various cities which buy the canned vegetables). Each link has an associated cost per unit of flow (e.g. cost per delivered truckload in this case).

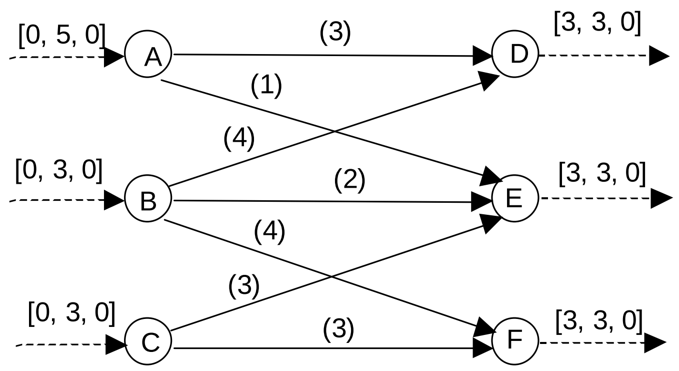


Figure 10.3: A simple transportation model. The transportation arcs are labeled with the cost per unit of flow.

Consider the example in Figure 10.3, which has three factories (A, B, and C) shipping to three markets (D, E, and F). The "transportation arcs" are the arcs which directly connect the sources (factories) to the sinks (markets); these are labeled only with the cost per unit of flow because the lower flow bounds are all at the default of zero, and the upper flow bounds are all at the default of infinity.

The question in this case is: how many truckloads per day should be produced at each factory and shipped to each sink to meet the market demands at minimum total cost? After solution, the flows in the transportation arcs will be known; hence the number of truckloads to ship from each factory to each market will be known.

It's always good to give a network model of this type a simple "idiot test" at first glance. In Figure 10.3 we see that the factories can produce up to a total of 11 units of flow while the markets demand exactly nine units of flow. This model passes the idiot test: there is sufficient supply to meet the demand. Of course, the model may fail for other reasons, e.g. the demand at a particular market cannot be met from the supply available to it. The linear programming solver will detect any of these problems, and appropriate infeasibility analysis routines can be brought into play.

The Assignment Problem

The assignment problem is a classic that also appears in the integer programming literature. In the usual form of the problem, you need to assign a set of people to a set of tasks. Each person takes a certain number of minutes to do a certain task, or cannot do a particular task at all, and each person can be assigned to exactly one task. How should the people be assigned to the tasks to minimize the total time taken to do all of the tasks?

The data for an assignment problem is often collected in a table, as shown below for example. The number in each cell indicates the number of minutes required for a particular person to do a particular task. The notation "n.a." in a cell indicates that the associated person cannot do the task associated with the cell. It's not obvious how to assign the people to the tasks by simple inspection of the table. For example, you may try looking at each task and simply choosing the best person for that task. But as you can see, person A is the best for tasks 2 and 4 and second-best for task 3. How should the tie be broken? Other ad hoc procedures also soon run into trouble. A more organized approach is needed.

	task 1	**task 2**	**task 3**	**task 4**
person A	n.a.	9	7	13
person B	16	13	8	n.a.
person C	10	n.a.	6	15
person D	11	n.a.	13	17

Surprisingly, the assignment problem can be cast as a transportation problem! Each person is modeled as a source node which introduces exactly one unit of flow into the network, and each task is modeled as a sink node which removes exactly one unit of flow from the network, as shown in Figure 10.4. Each arc has the default upper and lower flow bounds, but the cost per unit of flow is set equal to the number of minutes for the person to do the job. To avoid diagram clutter, each arc is labeled only with the cost per unit of flow.

After the solution of the resulting network linear program, the flows in the arcs (i.e. the values of the variables in the linear program) will be known. The flow in any arc will be exactly zero or exactly one. Why? Isn't it possible to send fractional units of flow and still satisfy all of the source and sink relationships? It's because (i) the unimodularity property restricts the arc flows to integer values because all of the node equations have integer constants, and (ii) the sources and sink nodes in the model all have inflows or outflows of exactly one unit of flow. Given this, the optimal set of assignments is shown by the arcs that have a positive flow. Each positive-flow arc indicates a person-to-task assignment that should be made. The objective function value gives the minimum total time associated with this assignment.

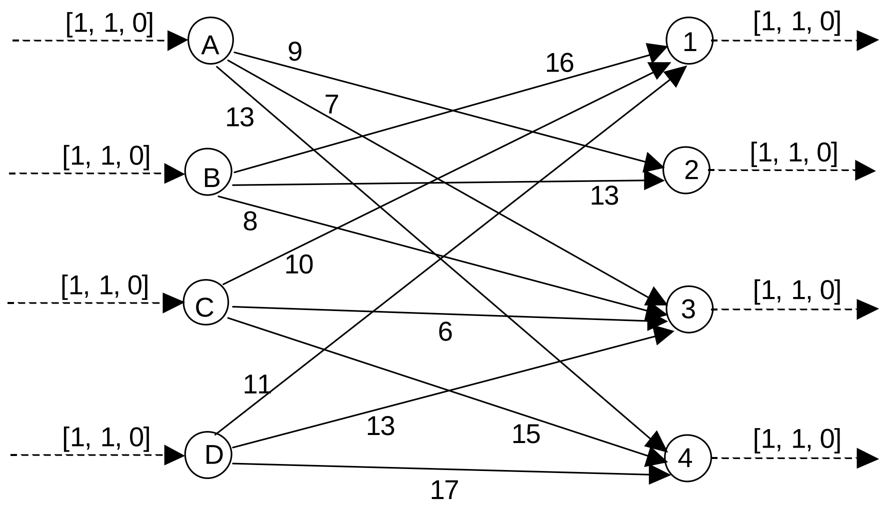


Figure 10.4: The assignment problem cast as a transportation network.

A straightforward variation of the assignment problem is the case in which there are more people than jobs. This is easy to handle simply by making each person the source of *up to* one unit of flow, by labeling the phantom arc associated with each person as [0,1,0].

The Transshipment Problem

This is a variation on the transportation problem in which shipping via intermediate nodes is allowed. In other words, not all of the nodes in the model are sources or sinks; some are simple flow conserving nodes. In addition, the sources and sinks may also transship flow. As in the transportation model, the nodes are assumed to have the default flow bounds. A simple, partially labeled transshipment network is illustrated in Figure 10.5. Node B is a source node that also acts as a transshipment node, node G is a sink node that also acts as a transshipment node, nodes D and E are pure transshipment nodes.

The Shortest Route Problem

Believe it or not, the shortest route problem, previously solved via Dijkstra's algorithm, can also be cast as a minimum cost network flow program, and therefore solved by extremely fast network flow programming codes. It's just a matter of properly labeling the nodes and arcs and interpreting the LP solution. Here are the main points of the problem setup:

- Create the network diagram,
- Label each arc with a lower flow bound of zero, an upper flow bound of infinity, and a cost per unit flow equal to the length of the arc. For example, an arc with a length of 12 kilometers would be labelled [0, ∞, 12].
- Make the origin node a source of exactly one unit of flow, with no cost per unit of flow. The label on the phantom arc should be [1,1,0].
- Make the destination node a sink of exactly one unit of flow, with no cost per unit of flow. The label on the phantom arc should be [1,1,0].

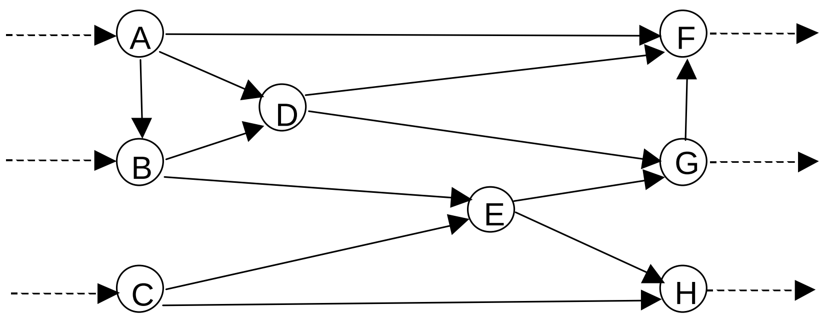


Figure 10.5: An unlabeled transshipment network.

The LP solution will assign flow to some arcs in the network, but not to other arcs. Arcs having a positive flow will be on the shortest route. In fact, if an arc has a positive flow, the flow value will be exactly one unit. Why? Because of the unimodularity property discussed earlier! It's a network flow value in which all of the node equations have integer right hand side constants, hence the network flows will all be integer. But the integers can't be any bigger than 1 because of the limitations placed on the single source node and the single sink node, so all arc flows will be either 0 or 1. And because of the minimum cost objective function, the LP will choose the "cheapest" (i.e. shortest) route to connect the input flow at the origin source node to the output flow at the destination sink node.

So there's no need to write or find an implementation of Dijkstra's algorithm to solve a shortest route problem when you have a linear programming solver at hand.

The Shortest Route Tree Problem

This is related to the shortest route problem, except that there is a single origin node, and many (if not all) of the other nodes in the network are destination nodes. The problem is to find the shortest route from the origin node to every one of the destination nodes. Of course, setting up and solving one shortest route problem for every destination node in the network can do this, but there is a simpler method that solves *all* of the shortest route problems simultaneously.

Again, it's just a matter of properly labeling the nodes and arcs and interpreting the LP solution. Let's assume that there are *n* destination nodes. Here are the main points of the problem setup:

- Create the network diagram,
- Label each arc with a lower flow bound of zero, an upper flow bound of infinity, and a cost per unit flow equal to the length of the arc. For example, an arc with a length of 12 kilometers would be labelled [0, ∞, 12].
- Make the origin node a source of exactly *n* units of flow, with no cost per unit of flow. The label on the phantom arc should be [*n*, *n*, 0].
- Make each destination node a sink of exactly one unit of flow, with no cost per unit of flow. The label on the phantom arc for each destination node should be [1,1,0].

Just as in the shortest route problem, the unimodularity property will permit only integer amounts of flow in an arc. Some arcs will have no flow, and some will have a flow equal to some positive integer less than or equal to *n*, the maximum amount of flow introduced at the origin node. If you mark the arcs that have a positive flow, then the marked arcs will form a tree on the diagram. The shortest route from the origin to each destination node is actually discovered backwards: trace the route from the destination node back to the origin via marked nodes. Because the marked arcs form a tree, there is only one shortest route for each destination node. The routes to several destination nodes may share an arc, which is why the flow in some arcs may be greater than one.

The Maximum Flow and Minimum Cut Problem

This problem too can be solved by proper formulation as a minimum cost network flow model. The main points of the problem setup:

- Create the network diagram,
- Label each arc with a lower flow bound of zero, the upper flow bound associated with the arc, and a cost per unit flow of zero. For example, an arc with an upper flow bound of 25 litres per minute would be labelled [0, 25, 0].
- Make the origin node a possible source of a very large volume of flow, with no cost per unit of flow. The label on the phantom arc should be [0, M, 0], where M represents a very large number.
- Make each destination node a possible sink of a very large volume of flow, with a cost per unit of flow of -1. The label on the phantom arc for the destination node should be [0, M, -1], where M again represents a very large number.

Now what is the effect of this strange labeling? Consider the destination node: it has a cost per unit of flow of –1. Since the objective function will automatically attempt to minimize costs, it will try to get as much flow as possible out via the destination node in order to drive the cost function as negative as possible. This is just like attaching a giant vacuum cleaner to the destination node, which attempts to suck as much flow as it can out of the network. At the other end, the origin node will certainly permit a very large volume of flow to enter the network. However, what limits the total flow is the upper flow bounds on the arc, just as it did in the Ford and Fulkerson algorithm we looked at earlier.

Thus, the minimum cost network flow solution will provide the following information:

- The flow out of the network at the destination node is the maximum total flow through the network.
- The flows in the arcs show a flow pattern that will provide this maximum flow. As usual, there may be other flow patterns that provide the same total flow.

The minimum cut is found in exactly the same way as it was after solving the maximum flow problem via the Ford and Fulkerson algorithm: mark the arcs that are full to capacity, then find a cut that uses only marked arcs.

A final question: what value should the M in the origin and destination nodes be set at? It should be a value that is at least as big as the maximum flow, but as small as possible to avoid any numerical problems in the computer solution. An easy way to find a suitable value is to take the cut-value for any random cut in the network diagram, because we know by the max-flow/min-cut theorem that the cut value will be greater than or equal to the maximum flow in the network.

Generalized Networks

In a *generalized network*, the arcs have *gain factors*. This is a number that multiplies the flow entering an arc to yield the flow leaving the arc. If all of the gain factors on all of the arcs are equal to 1, then this is a normal network. Using gain factors allows you to model a wider range of phenomena. For example, a gain factor of 0.9 might apply to a leaky pipe in a water network that loses 10% of its flow. Can you think of a reason for having a gain factor that is greater than 1? Obviously this represents an increase in the amount of flow leaving the arc as compared to the amount of flow entering the arc.

I asked this question to a graduate class some years back. They thought for a while and finally a student ventured an answer: "Suppose the flow represented chickens being transported to market, and meanwhile the chickens were laying eggs and they were hatching, thereby creating more chickens…". Creative answer. More realistically, a gain factor greater than 1 might be applied when the flow represents the value of an item, the arc represents the transportation of the item between two locations, and the item is worth more on arrival in the destination location.

Gain factors of zero are not allowed. Negative gain factors are allowed, but are not a good idea because they have the effect of reversing the flow at the head end of the arc, which destroys the intuitive meaning associated with the network diagram itself.

There are specialized fast networks for solving networks with side constraints.

Networks with Side Constraints

Sometimes the model is almost completely a network, but there are additional constraints that simply cannot be expressed as network relationships. For example, you may need to add a constraint like $13x_1 - 0.5x_2 + 12x_3 \geq 10$ to your network model. It's easy to see that this is not a network constraint in this case because the coefficients are not all +1 or –1. In a case like this you have a network with side constraints.

Networks with side constraints can of course be solved as ordinary general linear programs, but this sacrifices the solution speed that can be achieved for network models. Fortunately, there are specialized solution methods for networks with side constraints, which are quite fast. In fact, smart solvers will scan a model to see if it has network portions, and if it does, it may apply such specialized solvers. Some of the specialized solution algorithms work by solving the network portions by the fast network solution algorithms and then stitching the network portion together with the non-network portion. This is done numerous times in an iterative manner until the solutions for the two portions converge.

Processing Networks

Processing networks are extremely useful for modeling engineering systems, such as flows through factories. A processing network has regular nodes (constrained only by the usual conservation of flow), and at least one *processing node* in which the flows in the incident arcs are further constrained to have fixed proportions of flows relative to each other. For example, a processing node representing the energy flows in a large industrial boiler is shown in Figure 10.6. The numbers in the figure represent the fixed proportions of the energy flows. As you can see, the boiler is 80% efficient.

Processing nodes are shown as small squares on a network diagram to distinguish them from regular nodes, which are shown as small circles. If there are *k* incident arcs, then a processing node is represented by *k-1* ratio equations. For example, the processing node in Figure 10.6 is completely represented by the following 3 ratio equations:

$$a/b = 1/0.01 \rightarrow 0.01a - b = 0$$

$$a/c = 1/0.21 \rightarrow 0.21a - c = 0$$

$$a/d = 1/0.8 \rightarrow 0.8a - d = 0$$

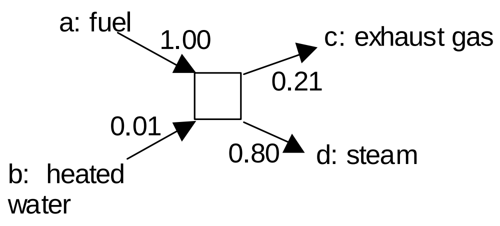


Figure 10.6: Processing node representation of an industrial boiler.

A complete model of a processing network is assembled by writing appropriate ratio equations for each processing node along with the flow conservation equations for the regular nodes. Then the usual flow bounds and minimum cost network flow objective function completes the model.

As you can see, a processing network is nothing more than a network with side constraints added because of the ratio equations generated by the processing nodes. The fast solution algorithms for networks with side constraints can then be used.

In a *flow-conserving processing network* the sum of the inflow proportions equals the sum of the outflow proportions. For example, in Figure 10.6, the inflow proportions sum to 1.01, as do the outflow proportions. This guarantees flow will be conserved in the processing node. However, there are many cases where flow conservation will not hold, especially when the units differ across the various flows incident on a processing node, as in an assembly model as shown in Figure 10.7. Here, the inflow proportions sum to 6, while the outflow proportions sum to 1.

However, this still simply defines a network model with side constraints, and so it can be solved by the usual methods.

As you have seen in this chapter, network models are incredibly versatile modeling tools with an appealing intuitive mapping to the system under study.

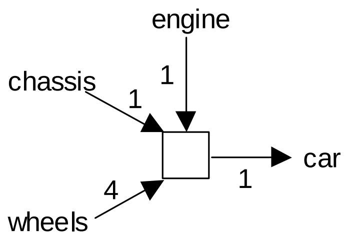


Figure 10.7: Processing node representation of an automobile assembly process

Chapter 11: PERT for Project Planning and Scheduling

PERT, the Project Evaluation and Review Technique, is a network-based aid for planning and scheduling the many interrelated tasks in a large and complex project. It was developed during the design and construction of the Polaris submarine in the USA in the 1950s, which was one of the most complex tasks ever attempted at the time. Nowadays PERT techniques are routinely used in any large project such as software development, building construction, etc. Supporting software such as Microsoft Project, among others, is readily available. It may seem odd that PERT appears in a book on optimization, but it is frequently necessary to optimize time and resource constrained systems, and the basic ideas of PERT help to organize such an optimization.

PERT uses a network representation to capture the precedence or parallel relationships among the tasks in the project. As an example of a precedence relationship, the frame of a house must first be constructed before the roof can go on. On the other hand, some activities can happen in parallel: the electrical system can be installed by one crew at the same time as the plumbing system is installed by a second crew.

The PERT formalism has these elements and rules:

- Directed arcs represent *activities*, each of which has a specified *duration*. This is the "activity on arc" formalism; there is also a less-common "activity on node" formalism. Note that activities are considered to be uninterruptible once started.
- Nodes are *events* or points in time.
- The activities (arcs) leaving a node cannot begin until all of the activities (arcs) entering a node are completed. This is how precedence is shown. You can also think of the node as enforcing a rendezvous: no-one can leave until everyone has arrived.
- There is a single starting node which has only outflow arcs, and a single ending node that has only inflow arcs.
- There are no cycles in the network. You can see the difficulty here. If an outflow activity cannot begin until all of the inflow activities have been completed, a cycle means that the system can never get started!

Consider the example given in Figure 11.1. Perhaps the pouring of the concrete foundation (activity A-B), happens at the same time as the pre-assembly of the roof trusses (activity A-D). However, the finalization of the roof (activity D-E), cannot begin until both A-D and B-D (assembly of the house frame), are done. Of course B-D cannot start until the concrete foundation has been poured (A-B). All of this precedence and parallelism information is neatly captured in the PERT diagram.

There are two major questions about any project:

- What is the shortest time for completion of the project?

- Which activities *must* be completed on time in order for the project to finish in the shortest possible time? These activities constitute the *critical path* through the PERT diagram.

The process of finding the critical path answers the first question as well as the second. Of course we need to know how long each individual activity will take in order to answer these questions. This is why the arcs in Figure 11.1 are labelled with numbers: the numbers show the amount of time that each activity is expected to take (in days, let's say).

The critical path is of great interest to project managers. The activities on the critical path are the ones which absolutely must be done on time in order for the whole project to complete on time. If any of the activities on the critical path are late, then the entire project will finish late! For this reason, the critical path activities receive the greatest attention from management. The non-critical activities have some leeway to be late without affecting the overall project completion time.

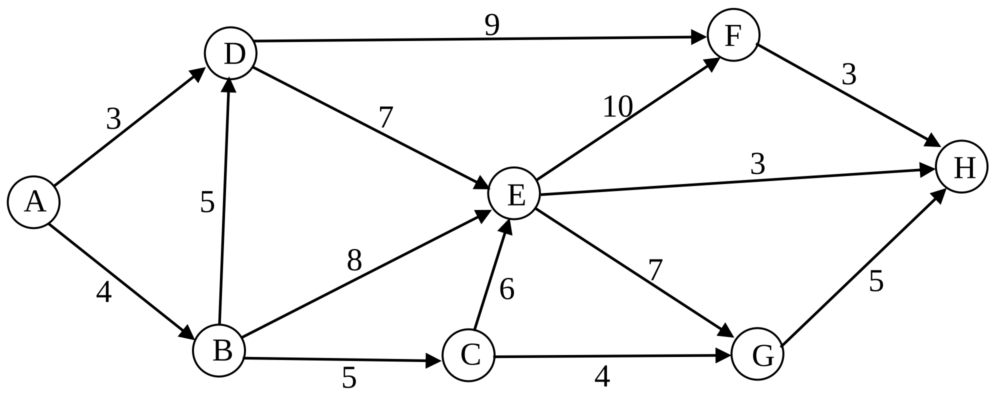


Figure 11.1: An example of a PERT diagram.

The following steps find the critical path and calculate other useful information about the project.

Step 1. Make a forward pass through the diagram, calculating the *earliest time* (T_E) for each event (node). In other words, what is the earliest time at which all of the activities entering a node will have finished? To find T_E, look at all of the activities which enter a node. T_E is the <u>latest</u> of the arrival times for entering arcs, i.e. T_E = max [(T_E of node at tail of arc) + (arc duration)] over all of the entering arcs. By definition, T_E of the starting node is zero.

Step 2. Make a backward pass through the diagram, calculating the *latest time* (T_L) for each event (node). In other words, what is the latest time that the outflow activities can begin without causing a late arrival at the next node for one of those activities? To find T_L, look at all of the activities which exit a node. T_L is the <u>earliest</u> of the leaving times for the exiting arcs, i.e. T_L = min [(T_L of node at head of arc) − (arc duration)] over all of the exiting arcs. By definition, the T_L of the ending node equals its T_E.

Step 3. Calculate the *node slack time* (S_N) for each node (event). This is the amount of time by which an event could be adjusted later than its T_E without causing problems downstream. $S_N = T_L - T_E$ for each node.

Step 4. Calculate the *total arc slack time* (S_A) for each arc (activity). This is the amount of time by which an activity could be adjusted later than the T_E of the node at its tail without causing problems later. S_A = (T_L of node at arc head) − (T_E of node at arc tail) − (arc duration).

Step 5. The critical path connects the nodes at which $S_N = 0$ via the arcs at which $S_A = 0$.

It should be no surprise that the critical path connects the nodes and arcs which have no slack. If there is slack, then the activity does not need to be done on time, which is exactly the opposite definition of the critical path!

As an example, let’s find the critical path for the PERT diagram in Figure 11.1. Note that there is an implied order in which the T_E’s can be calculated in Step 1. For example, the T_E of node D cannot be found until the T_E of node B is known. The starting node in Figure 11.1 is node A, and by definition the T_E of the starting node is 0. To calculate T_E at a node, we need to know the T_E of the node at the tail of every entering arc, so we can next only calculate the T_E of node B. This is simple since there is only one inflow arc, from node A, so T_E(B) = T_E(A) + (duration of A-B) = 0 + 4 = 4. The complete set of T_E calculations follows:

T_E(A)	= Starting node	= 0
T_E(B)	= T_E(A)+(duration A-B)	= 0+4 = 4
T_E(D)	= max{T_E(A)+(duration A-D), T_E(B)+(duration B-D)}	= max{0+3, 4+5} = 9
T_E(C)	= T_E(B)+(duration B-C)	= 4+5 = 9
T_E(E)	= max{T_E(D)+(duration D-E), T_E(B)+(duration B-E), T_E(C)+(duration C-E)}	= max{9+7, 4+8, 9+6} = 16
T_E(F)	= max{T_E(D)+(duration D-F), T_E(E)+(duration E-F)}	= max{9+9, 16+10} = 26
T_E(G)	= max{T_E(E)+(duration E-G), T_E(C)+(duration C-G)}	= max{16+7, 9+4} = 23
T_E(H)	= max{T_E(F)+(duration F-H), T_E(E)+(duration E-H), T_E(G)+(duration G-H)}	= max{26+3, 16+3, 23+5} = 29

The shortest time in which the project can be completed is now known: it is the same as the T_E of the ending node, node H, i.e. 29 days. But we still need to complete the remaining 4 steps of the algorithm to positively identify the critical path.

The backwards pass in Step 2 begins with the ending node H. By definition, the T_L of the ending node is equal to its T_E so T_E(H) = 29. This makes sense: otherwise the whole project would be pushed later! As for the forward pass, there is an implied order in which the node T_L values can be found, determined by the outflow arcs for which T_L is known. The complete set of calculations follows.

T_L(H)	= ending node = T_E(H)	= 29

$T_L(F)$	$= T_L(H)-(\text{duration F-H})$	$= 29-3 = 26$
$T_L(G)$	$= T_L(H)-(\text{duration G-H})$	$= 29-5 = 24$
$T_L(E)$	$= \min\{T_L(F)-(\text{duration E-F}), T_L(H)-(\text{duration E-H}), T_L(G)-(\text{duration E-G})\}$	$= \min\{26-10, 29-3, 24-7\} = 16$
$T_L(C)$	$= \min\{T_L(E)-(\text{duration C-E}), T_L(G)-(\text{duration C-G})\}$	$= \min\{16-6, 24-4\} = 10$
$T_L(D)$	$= \min\{T_L(F)-(\text{duration D-F}), T_L(E)-(\text{duration D-E})\}$	$= \min\{26-9, 16-7\} = 9$
$T_L(B)$	$= \min\{T_L(D)-(\text{duration B-D}), T_L(E)-(\text{duration B-E}), T_L(C)-(\text{duration B-C})\}$	$= \min\{9-5, 16-8, 10-5\} = 4$
$T_L(A)$	$= \min\{T_L(D)-(\text{duration A-D}), T_L(B)-(\text{duration A-B})\}$	$= \min\{9-3, 4-4\} = 0$

It's no surprise that T_L of the starting node is 0. If it wasn't it would mean that we could start the whole project late and yet still finish on time!

In Step 3 we find the *node slack time* (S_N) for each node (event) as shown below:

Node	A	B	C	D	E	F	G	H
S_N	0	0	1	0	0	0	1	0

This small PERT diagram is quite tight: only two of the nodes have nonzero slack. Larger diagrams with many parallel activities often have much more slack in the nodes.

In Step 4 we find *total arc slack time* (S_A) for each arc (activity) as shown below.

Arc	AB	AD	BC	BD	BE	CE	CG	DE	DF	EF	EG	EH	FH	GH
S_A	0	6	1	0	4	1	11	0	8	0	1	10	0	1

The arcs have quite a bit more slack time than the nodes in this small example. Later we will see how this slack can be put to good use in adjusting resource demands.

Finally, in Step 5, we find the critical path by linking the nodes having no slack via the arcs having no slack. Figure 11.2 shows the critical path for our example PERT diagram. The nodes and arcs having no slack are shown in boldface. If you've been watching closely, you might have noticed that the critical path through the PERT diagram is actually the *longest* path through the network.

Sometimes a situation arises in which one activity must precede two *different* events. How can this happen when a single arc can terminate only at a single event node? The solution lies in the use of *dummy arcs* which have a duration of zero. Dummy arcs are normally shown as dashed lines, as in the diagram fragment in Figure 11.3, in which activity A-B is the immediate predecessor of both event C and event D.

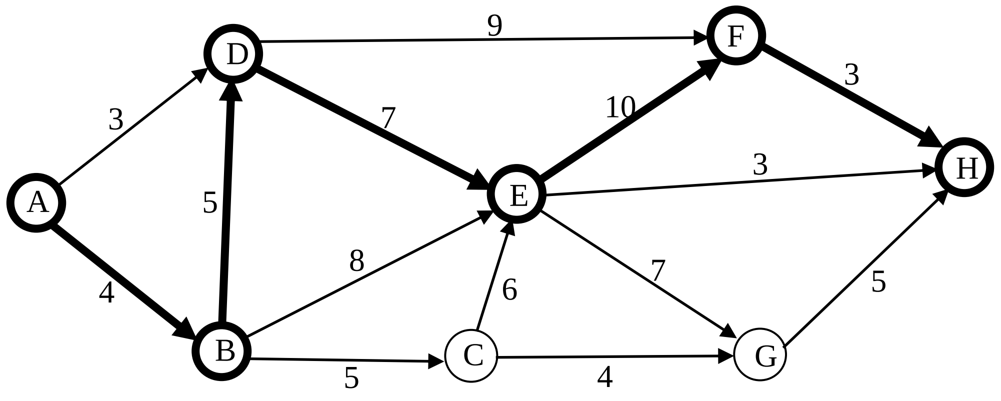


Figure 11.2: The critical path.

The determination of the critical path and hence of the duration of the entire project obviously depends very heavily on accurately assessing the duration of each individual activity. How is this done in practice? There are two main approaches: direct estimates, and the three-estimate method.

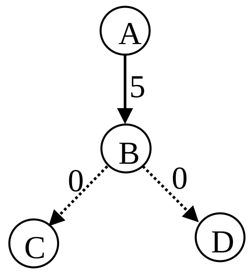


Figure 11.3: Dummy arcs.

In direct estimation, a single number is stated directly, perhaps based on long experience with similar projects. In the three-estimate approach, 3 estimates with specific properties are used in a weighted average. *m* is the most likely value, obtained in a manner similar to the direct estimate. *a* is an optimistic estimate, i.e. the time needed if everything goes just right (the weather is good, building materials arrive on time, crew is on time). Finally *b* is the pessimistic estimate, i.e. the time needed if everything goes wrong (it's raining, materials are late, crew books off sick, etc.). Given these three estimates, the final duration is set as $(a+4m+b)/6$.

Probabilistic PERT

Estimating is an inexact art, so we expect that our initial duration estimates have some error in them. What we would really like to know is how much this error is going to affect our estimate of the total project duration. Fortunately, with a few assumptions and very little extra work we can make some judgments about the likely amount of variation in the total project time. To do this we start with the three-estimate approach to estimating the activity durations. Next we make the following assumptions:

- The activity durations fit a Beta distribution.
- The range from *a* to *b* in the three-estimate approach covers 6 standard deviations.
- The activity durations are statistically independent.
- The critical path now means the path that has the longest *expected value* of total project time.
- The overall project duration has a normal distribution.

Given these assumptions, the expected value of each activity duration is given in exactly the same way as for the three-estimate approach: $(a+4m+b)/6$. The variance of each activity duration in this model is $[(b-a)/6]^2$.

Now the expected value of the total project duration is the sum of the expected activity durations along the critical path, which is found in the usual way. Finally the payoff: the variance of the total project duration is the sum of the variances of the activity durations for the activities in the critical path. This is very useful information for the managers: now they have some idea of how much the total project time might vary.

Resource Leveling

The method shown previously for determining the minimum amount of time to complete a project assumes that you have all the resources that you need. For example, it assumes that you have several crews to carry out activities simultaneously where the activities run in parallel in the PERT diagram. But suppose two activities should be done at the same time, but both require the use of the bulldozer, and you only have one bulldozer? If this happens the project may take longer to complete because the two activities must now be done in sequence, instead of in parallel. I say *may* take longer, because there may be slack in the two arcs which allows them to be shifted apart in time so that they no longer compete for the bulldozer, without lengthening the overall project time. As you see, the time and resources needed to complete a project interact in fundamental ways.

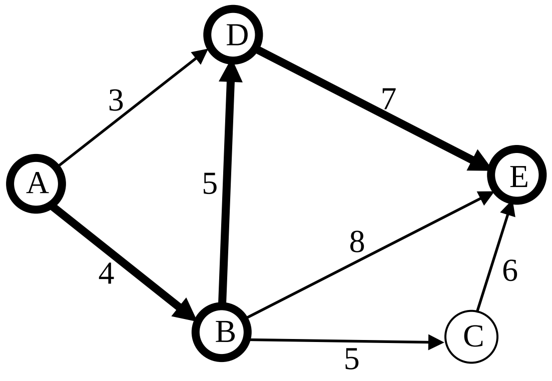


Figure 11.4: A small part of Figure 11.1.

We will explore this concept of using the slack time in the activities to move resource demands around. To keep things manageable, we will use a small piece of the PERT diagram from Figure 11.1, as shown in Figure 11.4.

There are actually two kinds of slack associated with an arc, e.g. arc B-C in Figure 11.4. The *total arc slack*, as we have seen before, is given in this case by $T_L(C) - T_E(B)$ – (duration B-C) = 10 – 4 – 5 = 1. The *free arc slack* is given in this case by $T_E(C) - T_E(B)$ – (duration B-C) = 9 – 4 – 5 = 0.

The total arc slack is the maximum amount of slack time available. If you start an activity later than the T_E of the node at its tail, by an amount equal to the total arc slack, this pushes the node at the head of the arc to its latest time. This has consequences downstream: activities later than the head node event time may now not be able to use their total slack. It's always OK to use up the free arc slack because this has no downstream consequences. However if you wish to push an activity later by any amount of time between the free slack and the total slack, you must be careful that there are no negative downstream consequences. In other words, you can schedule actvity B-C to begin at any time between $T_E(B)$ and $T_E(B)$ + (free slack B-C) without fear. However if you want to schedule activity B-C to start sometime between $T_E(B)$ + (free slack B-C) and $T_E(B)$ + (total slack B-C), then there may be downstream consequences because node C will be pushed later than $T_E(C)$.

The table below summarizes the total and free slack for all of the non-critical activities. Why do we list only the non-critical activities? Because the slack, both total and free, is always zero for the critical activities, by definition.

Activity	*Total slack*	*Free slack*
A-D	6	6
B-E	4	4
B-C	1	0
C-E	1	1

Now suppose that each activity, including the critical activities, must be staffed by certain numbers of employees, as shown in the following table:

Activity	*Employees needed*
A-B	3
B-D	2
D-E	3
A-D	4
B-E	3
B-C	2
C-E	2

Now we will try a few simple scenarios to see the effect that using the slack time in different ways has on the total number of employees needed at any point in the project. Remember that

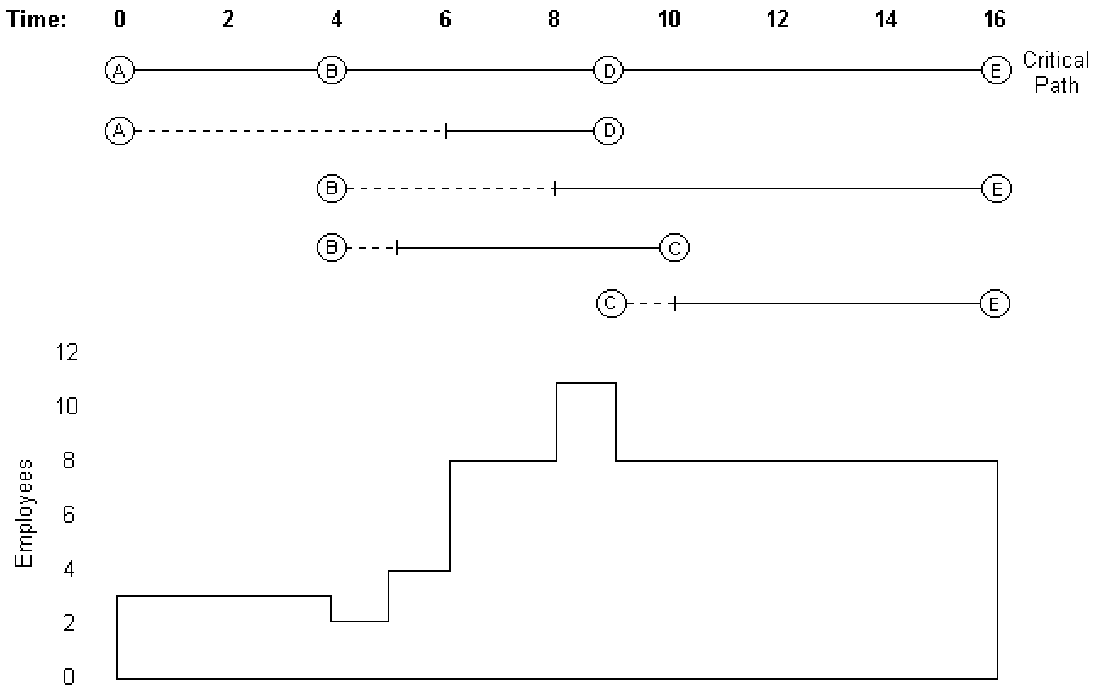


Figure 11.5: Every activity scheduled as *late* as possible.

each activity can be scheduled individually within the parameters of its free and total slack. However, to keep things simple, we will try these two scenarios: all activities scheduled as early as possible, and all activities scheduled as late as possible.

The scenario in which all activities are scheduled as late as possible is shown in Figure 11.5. The top part of Figure 11.5 is a Gantt chart, which shows the timing of each activity as a solid line. For example, activity A-D starts at time 6 and runs to time 9. The dashed extensions on the A-D line indicate the range of other possible starting times for A-D. The length of the dotted section is the amount of total slack available to the activity. There is no slack for the critical path activities.

The Gantt chart shows when each activity takes place, and we know how many employees are needed for each activity, so we can calculate the number of employees needed every day. For example, between time 8 and time 9 there are four activities taking place simultaneously: activity B-D (2 employees), activity A-D (4 employees), activity B-E (3 employees) and activity B-C (2 employees) for a total of 11 employees, the peak demand over the whole 16 day period. The demand for employees is charted in the bottom part of Figure 11.5.

The resource demands in Figure 11.5 are very uneven. Few employees are needed over the first 6 days, but then the demand shoots up quickly to a peak of 11 employees at time 8 to 9 before settling back to a need for 8 employees. This is a real problem if the company is operating with 8-person crews. For the first several days, most of the employees have nothing to do except drink coffee, while at time 8-9 the company will need to hire extra people or to pay overtime. For reasons such as these, managers usually want to level out the demand for resources, and this is referred to as *resource leveling*. What we'd like to do in Figure 11.5 is to somehow move some of the resource demand from the later part of the project to the earlier part of the project. This can be done by rescheduling some of the activities which have slack.

Figure 11.6 shows a second scenario, this time with all of the activities scheduled as early as possible. This scenario eliminates the peak of demand for employees that was seen in Figure 11.5. In fact, this is a schedule which can be accomplished without overtime using an 8-employee crew, so it is much preferred by management.

Note that many other scheduling scenarios are possible, using any combination of the possible starting times for the non-critical activities. For example, in Figure 11.6 we could shift activity C-E one day later in order to remove part of the 3-day period that requires 8 employees. Now it might be cheaper to run the project with a 7-employee crew and pay two days of overtime. Note that you must be careful with events such as C which have node slack. As you see in both Gantt charts, the ending C in activity B-C is offset from the starting C in activity C-E, indicating the possible ending and starting times of the two activities. However, if activity B-C uses its total slack, then C is pushed to its latest time, which means that activity C-E cannot start any earlier than that.

Common questions related to resource leveling include: Can I complete the project on time with a limited amount of resource (e.g. 8 employees)? What is the minimum amount of resource needed to complete this project on time? Given that I have only a set amount of resource, what is the minimum amount of time in which the project can be completed? Note that this last question takes the resource limit as fixed, and instead allows the project to take more time to complete. As an example, consider a project in which there is exactly one unit of resource (e.g. one employee) and each activity requires 1 employee. Now how long will it take the complete the project? The answer is simple: since there is no possibility of doing any of the activities in

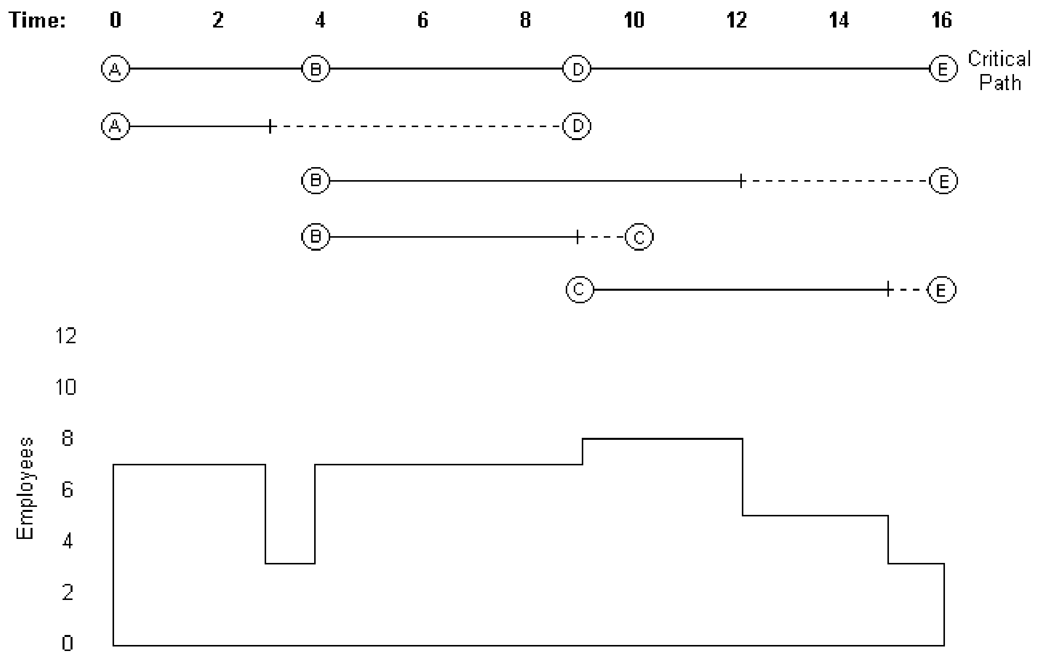


Figure 11.6: Every activity scheduled as *early* as possible.

parallel, the time needed to complete the project is the simple sum of *all* of the activity durations!

We have considered a simple case in which there is only a single type of resource (employees). In more complex projects there can be numerous resource types, e.g. bulldozers, graders, cement mixers, electricians, plumbers, dump trucks, front-end loaders, excavators, etc. Now you will have numerous interrelated resource leveling problems since a single activity will likely involve some subset of several of these resources.

Leveling resources to minimize the maximum amount of resource needed at any point in time is a very difficult combinatorially explosive problem, so it is solved via heuristics. Every commercial vendor of project planning software uses different proprietary heuristics, which means that the same resource leveling problem submitted to several different software packages will probably give you several different solutions. One heuristic I am aware of was developed by Pierre Paulin for the Ph.D. thesis in the Electronics Department here at Carleton University. His problem involved implementing a series of instructions on a chip which had a limited number of devices such as adders, buffers etc., so it was a classic multi-resource leveling problem. He developed the "force-directed" method which first works out the average rate of resource usage, and then conceptually adds "springs" to pull the valleys in the resource diagram up towards the average level and pull down the peaks. You then seek a kind of force balance between the “spring constants” which is achieved when the peaks and valleys are smoothed out as much as possible.

Time-Cost Tradeoffs

In real applications it is sometimes possible to reduce the amount of time that an individual activity takes by paying more, e.g. for overtime, for faster computers, for courier delivery, etc. If the project takes longer to complete than you have available, then you will have to spend some money to speed things up. But which activities should you speed up? Obviously those on the critical path, but if you speed up one activity on the critical path, the critical path itself may change to some other set of activities. And which activity on the critical path should you speed up, since some may be more expensive than others? And how much should you speed it up?

This is a complicated problem. You really can’t solve it by looking at the activities one at a time – you need to consider them all simultaneously. Fortunately there is a nifty optimization formulation which addresses this issue. Here we will show a linear programming formulation, but the extension to nonlinear programming should be obvious.

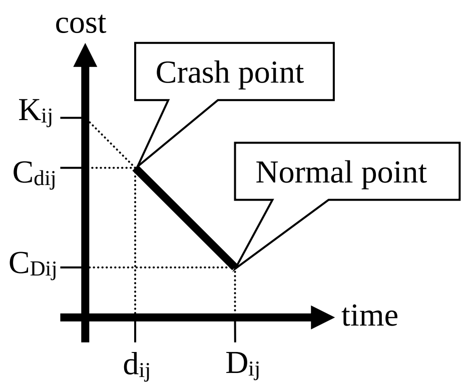


Figure 11.7: Time-cost tradeoff for activity ij.

The fundamental element in the linear programming formulation is a simple model showing how time and money trade off for each activity, as shown in Figure 11.7. In Figure 11.7, the “normal point” shows the length of time (D_{ij}, large D for large duration) and cost (C_{Dij}) when activity *ij* is run in the normal manner. The “crash point” shows the length of time (d_{ij}, small d for small duration) and cost (C_{dij}) when the activity is “crashed”, or sped up to the maximum extent possible. Note that the cost for the activity goes up as the duration goes down – money is exchanged for time. We assume that any amount of speedup between D_{ij} and d_{ij} is possible. K_{ij} is just the value on the cost axis reached by extending the time-cost tradeoff line for the activity; it makes the formulation a little simpler.

Now every activity can have a duration somewhere d_{ij} and D_{ij}. The unknown duration for activity ij is represented by the variable x_{ij}. We also define the constant $C_{ij} = (C_{dij} - C_{Dij})/(D_{ij} - d_{ij})$, which is just the negative of the slope of the time-cost tradeoff line. C_{ij} allows us to compactly express the cost of any intermediate value on the time-cost tradeoff line as $K_{ij} - C_{ij}x_{ij}$, where $d_{ij} \leq x_{ij} \leq D_{ij}$.

The objective is to minimize the total costs of speeding up activities while meeting a specified deadline, i.e. to minimize $\sum_{ij}(K_{ij} - C_{ij}x_{ij})$. However we can always drop constants which appear in objective functions, so it can be rewritten as *minimize* $\sum_{ij}(-C_{ij}x_{ij})$, which is the same as *maximize* $\sum_{ij}C_{ij}x_{ij}$.

We also need to capture the precedence relationships in the PERT diagram. This seems difficult because we don’t know the event times, let alone the durations of the activities. We define the

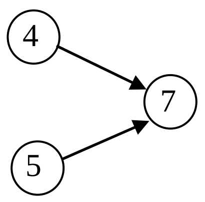


Figure 11.8: Capturing precedence.

variables y_k to represent the unknown event times. Now how do we capture the precedence relationships? The main idea is to make sure that no activity leaving an event node begins before all of the activities entering the event node have terminated. For example, consider Figure 11.8. How do we make sure that y_7, the event time for node 7, is later than the latest arrival time of activities 4-7 and 5-7? The behaviour we want is $y_7 = \max\{y_4+x_{47}, y_5+x_{57}\}$. This is easily achieved by using two inequalities: $y_4+x_{47} \leq y_7$ and $y_5+x_{57} \leq y_7$.

Now we are almost ready to set up the entire linear program. For the starting event, define $y_1 = 0$ and for the ending event, define $y_n \leq T$. Note that T, the project deadline, should be less than the shortest project time if every activity is run in normal time, otherwise the solution is obvious: just run every activity in normal time.

Now we can summarize the entire formulation:

maximize $\sum_{ij} C_{ij} x_{ij}$

subject to: $d_{ij} \leq x_{ij} \leq D_{ij}$ for all activities (arcs)

$y_i + x_{ij} - y_j \leq 0$ for all events (nodes)

$y_1 = 0$

$y_n \leq T$

$x_{ij}, y_k \geq 0$

The minimum cost for speeding up the project is not given directly by the optimum value of the objective function, but it is easy to calculate it once the x_{ij} values are known.

As you can see, the same general formulation can be used when the time-cost tradeoff curve is nonlinear, with the exception that the objective function will not be linear. I have used a nonlinear version of this formulation in a transistor-sizing optimization. The idea is to keep the signal propagation time below a certain limit. The propagation time is given by the critical path through the transistor network. However the propagation time through an individual transistor depends on the sizes of the transistor and its neighbours. “Cost” in this instance is transistor size, where larger transistors are faster. The idea is to make sure that the signal propagates in a time below the stated limit while keeping the total transistor area as small as possible. The objective function in this case was nonlinear, as were the constraints defining the x_{ij}.

Chapter 12: Integer/Discrete Programming via Branch and Bound

Thus far we have been dealing with models in which the variables can take on real values, for example a solution value of 7.3 is perfectly fine. But the variables in some models are restricted to taking only integer or discrete values. You can assign 6 or 7 people to a team, for example, but not 6.3 people; or you can choose to make a transistor from silicon dioxide or gallium arsenide, but not some mixture. Binary variables are a subset of integer/discrete variables that are restricted to 0/1 values. Binary variables are usually associated with yes/no decisions, e.g. to undertake a project or not.

You will often encounter integer models that look like they can be solved by methods suitable for real-valued variables. For example, it is common to find models in which the objective function and constraints are linear relations defined on integer variables. This looks exactly like a standard linear program, except that the variables cannot take on real values. There is an overwhelming temptation to just solve the problem by standard linear programming and then to round any non-integer variable values to the closest integer value. Don't do this! The following simple example (due to Hillier and Lieberman) shows how misleading this can be.

Maximize Z = $x_1 + 5x_2$
Subject to: $x_1 + 10x_2 \leq 20$
$x_1 \leq 2$
$x_1, x_2 \geq 0$ and integer.

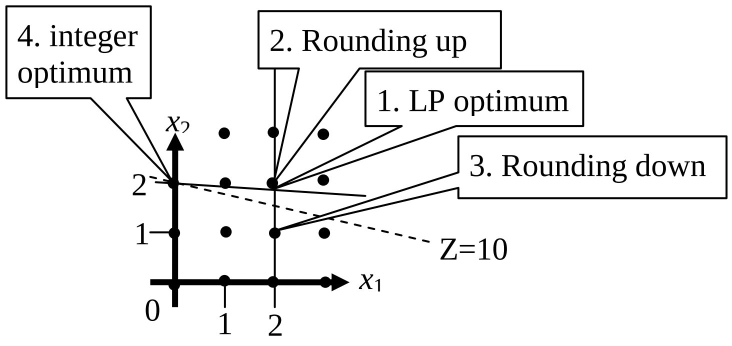


Figure 12.1: Why rounding doesn't work.

The linear programming solution to this problem yields Z=11 at (2, 1.8). The first instinct is to round x_2 to the nearest integer value, i.e. x_2=2. Unfortunately (2, 2) is infeasible by the first constraint. The natural inclination is then to try rounding x_2 in the opposite direction, i.e. x_2=1. This yields a feasible point (2, 1) with Z=7. However, this is not the optimum point! The integer-feasible optimum point is at (0, 2) where Z=10. Figure 12.1 shows a sketch of this problem. The dots in Figure 12.1 show points at which both x_1 and x_2 have integer values simultaneously.

Notice that the integer optimum point is far away from the LP optimum. Notice also how you can't get to the integer optimum (0,2) by rounding the LP optimum (2, 1.8). The moral of the story is that you simply can't use real-valued solution methods to optimize integer-valued models. Completely different methods are needed.

The first thing you might think of for solving integer-valued problems is to simply enumerate all of the possible solutions and then to choose the best one. This will actually work for very small problems, but it very rapidly becomes unworkable for even small- to medium-size problems, let alone industrial-scale problems. For example, consider a binary 0/1 problem that has 20 variables. This will have 2^{20}=1,048,576 solutions to enumerate, which might be possible to do via computer. Now consider a slightly larger binary problem having 100 variables. Now there are $2^{100}=1.268\times10^{30}$ solutions to enumerate, which is likely impossible, even for a very fast computer. The combinatorial explosion is even worse for general integer variables that can take on even more values than just the two possibilities for binary variables. It is easy to construct problems in which the number of solutions is greater than the total number of atoms in the universe! Enumeration just won't work for most real-world problems – we need a better way of tackling the combinatorial explosion.

The *branch and bound* method is the basic workhorse technique for solving integer and discrete programming problems. The method is based on the observation that the enumeration of integer solutions has a tree structure. For example, consider the complete enumeration of a model having one general integer variable x_1, and two binary variables x_2 and x_3, whose ranges are $1\leq x_1\leq3$, $0\leq x_2\leq1$, and $0\leq x_3\leq1$. Figure 12.2 shows the complete enumeration of all of the solutions for these variables, even those which might be infeasible due to other constraints on the model.

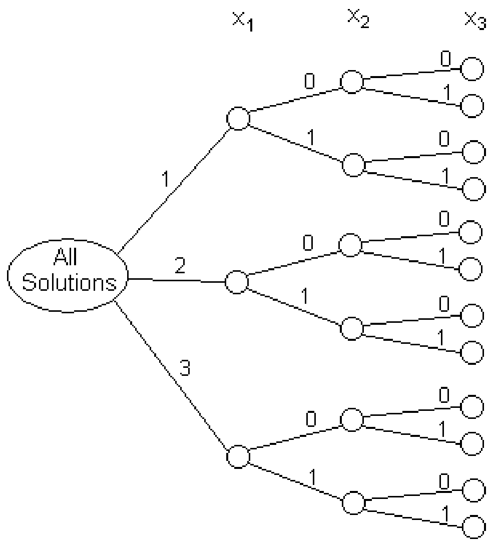


Figure 12.2: A full enumeration tree.

The structure in Figure 12.2 looks like a tree lying on its side with the *root* (or root node) on the left, labeled "all solutions", and the *leaves* (or leaf nodes) on the right. The leaf nodes represent the actual enumerated solutions, so there are 12 of them: (3 possible values of x_1) × (2 possible values of x_2) × (2 possible values of x_3). For example, the node at the upper right represents the solution in which x_1=1, x_2=0, and x_3=0. The other nodes can be thought of as representing sets of possible solutions. For example, the root node represents all solutions that can be generated by growing the tree. Another intermediate node, e.g. the first node directly to the right of the root node, represents another subset of all of the possible solutions, in this case, all of the solutions in which x_1=2 and the other two variables can take on any of their possible values. For any two directly connected nodes in the tree, the *parent* node is the one closer to the root, and the *child* node is the one closer to the leaves.

Now the main idea in branch and bound is to <u>avoid</u> growing the whole tree as much as possible, because the entire tree is just too big in any real problem. Instead branch and bound grows the tree in stages, and grows only the most promising nodes at any stage. It determines which node is the most promising by estimating a bound on the best value of the objective function that can be obtained by growing that node to later stages. The name of the method comes from the *branching* that happens when a node is selected for further growth and the next generation of children of that node is created. The *bounding* comes in when the bound on the best value attained by growing a node is estimated. We hope that in the end we will have grown only a very small fraction of the full enumeration tree.

Another important aspect of the method is *pruning*, in which you cut off and permanently discard nodes when you can show that it, or any its descendents, will never be either feasible or optimal. The name derives from gardening, in which pruning means to clip off branches on a tree, exactly what we will do in this case. Pruning is one of the most important aspects of branch and bound since it is precisely what prevents the search tree from growing too much.

To describe branch and bound in detail, we first need to introduce some terminology:

- *Node*: any partial or complete solution. For example, a node that is two levels down in a 5-variable problem might represent the partial solution 3-17-?-?-?, in which the first variable has a value of 3 and the second variable has a value of 17. The values of the last three variables are not yet set.
- *Leaf (leaf node)*: a complete solution in which all of the variable values are known.
- *Bud (bud node)*: a partial solution, either feasible or infeasible. Think of it as a node that might yet grow further, just as on a real tree.
- *Bounding function*: the method of estimating the <u>best</u> value of the objective function obtainable by growing a bud node further. Only bud nodes have associated bounding function values. Leaf nodes have objective function values, which are actual values and not estimates. It is important that the bounding function be an <u>optimistic</u> estimator. In other words, if you are minimizing, it must underestimate the actual best achievable objective function value; if maximizing it must overestimate the best achievable objective function value. You want it to be as accurate an estimator as possible so that the resulting branch and bound tree is as small as possible, but it <u>must</u> err in the optimistic direction. The bounding function is the real magic in branch and bound. It takes ingenuity sometimes to find a good bounding function, but the payoff in increased efficiency is tremendous. Every problem is different.
- *Branching, growing, or expanding a node*: the process of creating the child nodes for a bud node. One child node is created for each possible value of the next variable. For example, if the next variable is binary, there will be one child node associated with the value zero and one child node associated with the value one.
- *Incumbent*: the best complete feasible solution found so far. There may not be an incumbent when the solution process begins. In that case, the first complete feasible solution found during the solution process becomes the first incumbent.

Branch and bound is a very general framework. To completely specify how the process is to proceed, you also need to define policies concerning selection of the next node, selection of the next variable, how to prune, and when to stop. We'll discuss these next.

At any intermediate point in the algorithm, we have the current version of the branch and bound tree, which consists of bud nodes labeled with their bounding function values and other nodes labeled in various ways that we will see later. The *node selection policy* governs how to choose the next bud node for expansion. There are three popular policies for node selection:

- *Best-first* or *global-best node selection:* choose the bud node that has the best value of the bounding function anywhere on the branch and bound tree. If we are minimizing, this means choosing the bud node with the smallest value of the bounding function; if maximizing choose the bud node with the largest value of the bounding function.
- *Depth-first:* choose only from among the set of bud nodes just created. Choose the bud node with the best value of the bounding function. Depth-first node selection takes you one step deeper into the branch and bound tree at each iteration, so it reaches the leaf nodes quickly. This is one way of achieving an early incumbent solution. If you cannot proceed any deeper into the tree, back up one level and choose another child node from that level.
- *Breadth-first:* expand bud nodes in the same order in which they were created.

Similarly, once a bud node has been chosen for expansion, how do we choose the next variable to use in creating the child nodes of the bud node? The *variable selection policy* governs this choice. There are few standard policies for variable selection. The variables are often selected just in their natural order, though a good variable selection policy can improve efficiency greatly.

We also need to establish policies and rules for pruning bud nodes. As mentioned above, there are two main reasons to prune a bud node: you can show that no descendent will be feasible, or you can show that no descendent will be optimal.

The method for showing that no descendent will be optimal is standard: if the bud node bounding function value is worse than the objective function value for the incumbent, then the bud node can be pruned. This is because the bounding function is an optimistic estimator. Suppose you are maximizing, and the incumbent solution has an objective function value of 87. If the optimistic bounding function overestimate for the bud node has a value of only 79, then you know that no descendent of the bud node will ever exceed 79, let alone 87, and so none of the descendents can ever be optimal. So the bud node is pruned. The same reasoning applies in reverse if minimizing.

Methods for showing that no descendent can ever be feasible vary with the specific problem. In problems that include standard arithmetic constraints, it is sometimes easy to detect this condition. For example, consider a partial solution $(x_1,x_2,x_3,x_4)=(1,1,?,?)$ in an all-binary problem which has the constraint $-10x_1-5x_2+6x_3+4x_4\geq 0$. Now that x_1 and x_2 are both set to 1, there are no possible settings of x_3 and x_4 which will satisfy the constraint. Hence we can deduce that all of the descendents of this bud node will be infeasible, so the node is pruned.

There is one other case in which the expansion of a bud node can be halted: when the best possible value of the objective function obtainable by expansion can be seen directly. This is known as *fathoming* a node. This is sometimes a by-product of the bounding function calculations. Bounding functions often work by solving a simpler problem that is created by ignoring some of the constraints on the real problem. Sometimes the solution to this simpler problem actually does satisfy all of the constraints on the original problem, hence it is the best possible solution for the original problem, and is obtained without expanding the bud node any further. All that is necessary at this point is to compare this solution to the incumbent: if it is better than the incumbent, then it replaces it, otherwise the node can be pruned.

Finally, we need a *terminating rule* to tell us when to stop expanding the branch and bound tree. To guarantee that we have reached optimality, we stop when the incumbent solution's objective function value is better than or equal to the bounding function value associated with all of the bud nodes. This means that none of the bud nodes could possibly develop into a better solution than the complete feasible solution we already have in hand, so there is no point in expanding the tree any further. Of course, according to our policies for pruning, all bud nodes in this condition will already have been pruned, so this terminating rule amounts to saying that we stop when there are no more bud nodes left to consider for further growth! This also proves that the incumbent solution is optimum.

That's a lot of theory, so it's time for an example. Let's revisit the person-task assignment problem. Recall, though, that this problem can be cast in a network-flow format and solved quickly by linear programming, so you would never actually solve it via branch and bound in real life. However it's an easy problem to understand, so we will re-use it to demonstrate a branch and bound solution.

We have four persons, A through D, to assign to four tasks, 1 through 4. The table below shows the number of minutes it takes for each person to do each task. Each person can do exactly one task, and all tasks must have an assigned person. The objective is to minimize the total minutes taken. How many possible assignments of persons to tasks are there? There are 4!=24. In general, when there are *n* persons and *n* tasks there are *n*! possible assignments.

		task			
		1	**2**	**3**	**4**
person	**A**	9	5	4	5
	B	4	3	5	6
	C	3	1	3	2
	D	2	4	2	6

The formal branch and bound formulation follows. Any complete formulation must address all of these items.

- *Meaning of a node in the branch and bound tree*: a partial or complete assignment of persons to tasks. For example, a complete assignment ACBD represents the assignment of person A to task 1, person C to task 2, etc.
- *Node selection policy:* global best value of the bounding function.
- *Variable selection policy:* choose the next task in the natural order 1 to 4.
- *Bounding function:* for unassigned tasks, choose the best unassigned person, even if that person is chosen more than once. This is a relaxation of the original problem in which each person can do exactly one task.
- *Terminating rule:* when the incumbent solution objective function value is better than or equal to the bounding function values associated with all of the bud nodes.

- *Fathoming:* the "solution" generated by the bounding function is feasible if every task is assigned to a different person.

As an example of the calculation of the bounding function, let's look at the first-level node associated with assigning person A to task 1. The set of solutions represented by this node is A???. The bounding function value is calculated as follows:

- Actual value associated with assigning A to task 1: 9.
- Best unassigned person for task 2 is C, value: 1.
- Best unassigned person for task 3 is D, value: 2.
- Best unassigned person for task 4 is C, value: 2.
- The bounding function "solution" is ACDC, with total cost = 9+1+2+2=14. At this point we know that the very best objective function value that we might find at a leaf node descended from A??? is 14. Since person C is repeated, this is not a feasible solution, so the node is not fathomed. Note that person A is actually assigned, so we will not see A repeated in the bounding function calculations at this node or at any of the descendent nodes.

In practice, this bounding function amounts to crossing out the tasks (columns) that have assigned people, and the people (rows) that have assigned tasks, and then choosing the best number in each of the remaining columns, even if a particular person (row) is chosen more than once.

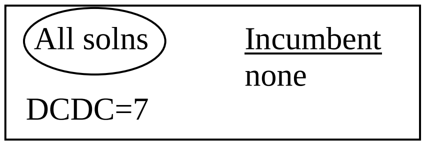


Figure 12.3: Stage 1: the root node.

Now we are ready to develop the branch and bound tree. First we create the root node. It's always worth finding the "solution" generated by the bounding function at the root node, since there is a chance that if you are fantastically lucky this "solution" will fathom the entire tree! Figure 12.3 shows the root node. In all of the figures, each node is labeled with the "solution" generated by the bounding function, and the bounding function value. Pruned nodes are indicated by dashed borders, and feasible nodes are indicated by bold borders. Pruned feasible nodes have both features.

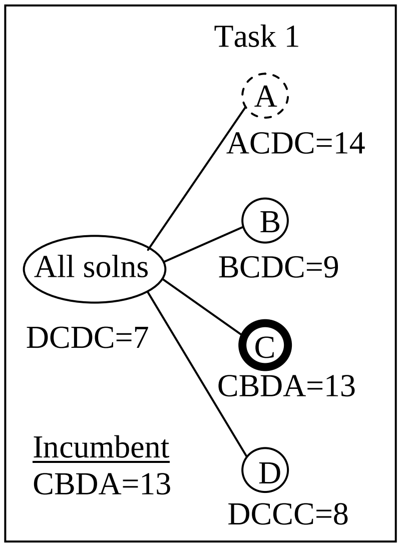


Figure 12.4: Stage 2.

Figure 12.4 shows the next level of the tree, generated from the root node by enumerating the possible persons who can do task 1. Node C??? is fathomed, providing our first feasible solution, and hence the first incumbent solution, CBDA=13. This allows us to prune node A??? whose bounding function value is 14.

There are two bud nodes in Figure 12.4 that show promise of improving on the incumbent solution: B??? with a bounding function value of 9 and D??? with a bounding function value of 8. Since we are using the global-best node selection policy, we choose node D??? for further expansion, which results in Figure 12.5. Note that the child nodes of D??? are labeled A, B, and C, but not D. Why? Because person D is already assigned, and so can't be assigned again in descendent nodes. There are no new feasible solutions, so the incumbent solution is unchanged, and none of the new nodes can be pruned by comparison with the incumbent, or fathomed. So we choose the global best value of the bounding function between nodes B??? with value 9, DA?? with value 12, DB?? with value 10 and DC?? with value 12. The best of these is node B???, which is expanded in Figure 12.6.

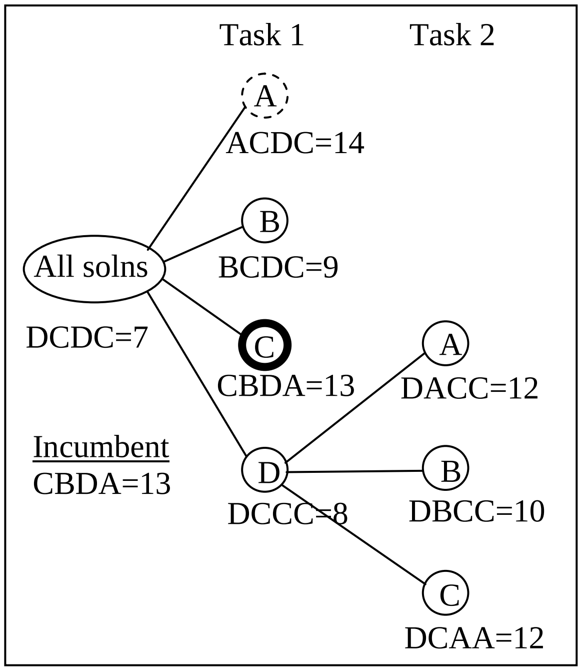


Figure 12.5: Stage 3.

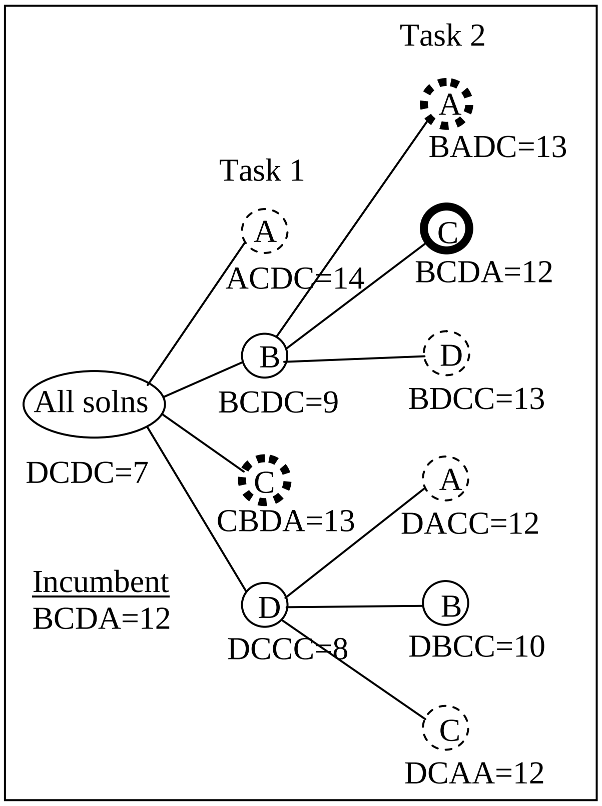


Figure 12.6: Stage 4.

Figure 12.6 shows that there are two new fathomed nodes, BA?? and BC??. The feasible bounding function "solution" associated with BC?? has a lower value than the current incumbent, and so replaces it and prunes the old incumbent. Node BA??, even though feasible, is pruned by comparison with the new incumbent. Nodes DA?? and DC?? are also pruned by comparison with the incumbent (these could be kept for potential later exploration if we wanted to find all possible optimum solutions instead of just one). There remains only a single bud node which has the possibility of producing a better solution than the incumbent: node DB??. This node is expanded to produce Figure 12.7.

Figure 12.7 shows the expansion of DB?? to produce two child nodes. Both of these are feasible since once the first 3 persons are chosen, the only unassigned person is assigned to the final task, producing a feasible solution. The newly produced solution DBAC has a better objective function value than the incumbent, and so replaces it and prunes the old incumbent. The other newly created solution, DBCA, is pruned by comparison with the new incumbent. At this point there are no bud nodes remaining for possible expansion, so we stop. The final solution is given by the current incumbent: DBAC with a minimum total time of 11 minutes for the assignment.

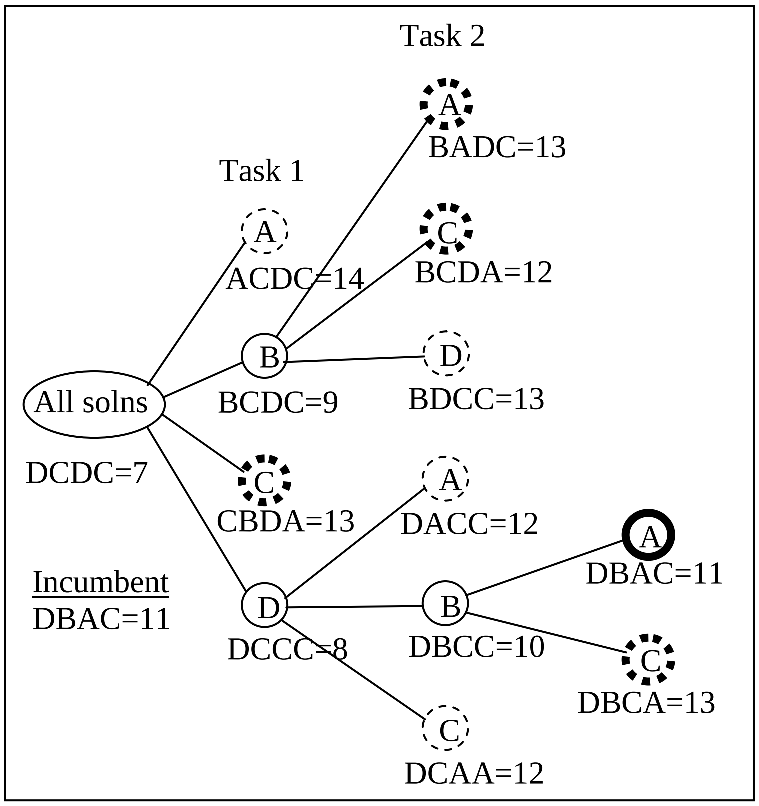


Figure 12.7: Stage 5. The tree is complete.

Now how much work did we do? We evaluated 13 nodes, including the root node. This is about half of the work of a full enumeration of the 24 possible solutions, quite a good speed-up. However, branch and bound solutions for large problems should do a much smaller fraction of the work, more like a tenth of a percent or less.

Some final notes on branch and bound methodology. First, how do we deal with ties for the next node to choose for expansion? You can simply choose arbitrarily: if you choose wrongly, the branch and bound method will eventually bring you back to the unchosen node. A general rule of thumb is to first choose the node that is farthest away from the root, since it is more likely to be closer to a feasible solution.

Second, suppose the incumbent solution objective function value is tied with the bounding function value at some bud nodes. If we are only interested in a single solution to the problem, then the bud nodes can be pruned because the best that they can do is to eventually grow into solutions equal to the one we already have on hand, and even that is not very likely. On the other hand, if we are interested in finding <u>all</u> of the optimal solutions, then those bud nodes are kept until a strictly better incumbent solution prunes them. At the end, if there are bud nodes that have the same bounding function as the optimum solution, then they are explored: each will either produce a different solution with the same value of the objective function, or will be pruned. This allows us to generate all of the equivalent optimum solutions. In Figure 12.6 we pruned two bud nodes that had bounding function values equal to the incumbent objective function value because we were interested in finding just one single optimum solution.

More on Bounding Functions

A good bounding function is really what makes branch and bound work. Sometimes it takes a bit of ingenuity to find a good one. Let's consider a bounding function that we can use in a completely different problem. In the well-known traveling salesman problem we are given a graph whose arcs are labeled with distances. The goal is to find a *tour* that visits each node in the graph exactly once and returns to the original node, and has the shortest total length. This is the optimum route for the traveling salesman's as he visits each node (city) to sell his wares.

In this case a node in the branch and bound tree represents a partial tour, and it is the bounding function's job to estimate the length of the shortest tour that might result from continuing this tour. Let's suppose the partial tour is as shown in boldface in Figure 12.8. How can we estimate the length of the shortest complete tour that incorporates the partial tour shown in Figure 12.8?

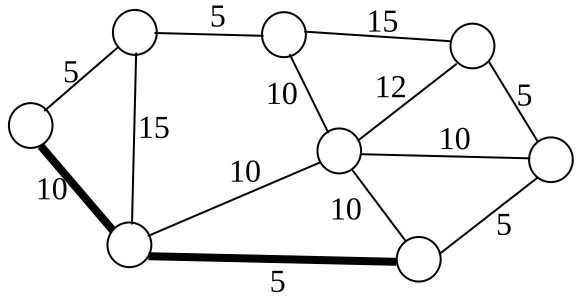


Figure 12.8: A partial tour in a traveling salesman problem.

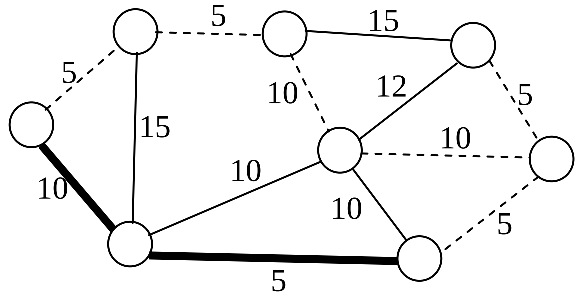


Figure 12.9: Bounding function: partial tour plus minimum spanning tree.

As usual the bounding function will solve a simpler problem that violates some of the constraints on the original problem. One clever bounding function solves a minimum spanning tree problem over the unvisited nodes and the two end nodes on the partial tour, as shown in Figure 12.9 (the minimum spanning tree arcs are dashed). The bounding function value associated with Figure 12.9 is given by (length of partial tour) + (length of minimum spanning tree) = (10+5) + (5+5+10+10+5+5) = 55. The "solution" generated by the bounding function is not feasible in this case (some cities must be visited more than once if this route is used), but it is a good underestimate of the shortest route. It also has the advantage that it is easy to recognize a feasible solution if one is generated by the bounding function, so fathoming is easy.

The traveling salesman problem is one of those interesting problems that are easy to state but hard to solve. It has been the subject of a great deal of work over many years since it has many practical applications (e.g. the routing of the welding head wielded by a robot on an automobile assembly line). There are now some reasonably good heuristic methods for fairly large problems.

Keys to Success

There are several keys to using branch and bound successfully.

Have a good bounding function. It must be optimistic, but as close to feasibility as possible. The more exact it is, the smaller the resulting search tree will be. There are often numerous different ways to construct a bounding function for a particular problem. Be creative!

Get a good incumbent early. This is tremendously useful in pruning because many buds will never be expanded if their bounding function value is worse than the objective function value of the incumbent. Consider using a heuristic to generate an incumbent solution even before beginning the branch and bound process. As an example, consider the following heuristic for generating an initial incumbent in the person-task assignment problem. First choose the smallest number anywhere in the table, then remove the associated task and person from the table. Repeat this process until all tasks and people have been assigned. You could also use depth-first node selection until the first incumbent is found, then switch to another node selection policy.

Find ways to identify nodes that have no feasible descendents. This prevents nodes from being expanded.

Order constraints and variables so that the most restrictive are tested first. The general idea is to encourage early failure of nodes on the tree. The closer to the root that a node is pruned, the more tree is cut off. If one variable has a very restricted range or is involved in a very restrictive constraint, generate child nodes based on that variable first.

Sub-optimizing in Very Large Problems

Branch and bound slows down the combinatorial explosion in integer programs, but it doesn't stop it altogether. There are many problems in which even the standard branch and bound tree is too large. In cases like this we can sacrifice the guarantee of optimality that is provided by branch and bound in favour of getting a reasonable answer quickly, or within the memory limitations of our computer. There are three main heuristics, all based on branch and bound.

Stopping with a guarantee of closeness to optimality. Choose an acceptable distance to optimality, e.g. 5%. Now you simply halt the branch and bound process when the incumbent solution is within 5% of the best bud node bounding function value. The incumbent solution is definitely within 5% of the optimum, and it could be much closer, even optimum. It just takes a lot more branch and bound nodes to prove it.

For example, global optimization of nonlinear functions can be done by a branch and bound procedure which subdivides the search space as it goes. A recent Ph.D. student of mine used this approach. We discovered that his procedure very often reached the optimum solution in the first 5 seconds of computing time, but it then took another 24 hours of computing time to prove that it was really the optimum! Stopping within some stated fraction of the optimum can reduce calculation time a great deal, sometimes without much effect on the final solution.

Beam search. This technique is especially useful if computer memory is limited. Set an upper limit on the number of bud nodes that will be maintained in memory, e.g. 1000. If this limit is reached, rank order all of the bud nodes based on their bounding function values and keep only the best 1000. Of course the guarantee of optimality is lost (one of the discarded nodes may have been the one which ultimately lead to the optimum solution), but the search can run in the limited memory space available.

Depth-first search to first incumbent. If time is limited, depth-first search is preferred since it is the most likely to reach a feasible incumbent solution first. If time runs out, at least one feasible solution is available even though the branch and bound solution is halted prematurely. As we will see later, depth-first search is especially preferred if linear programming is being used in the bounding function since each LP solution is quite similar to the last one, so advanced starts can be used.

Branch and bound methods can be customized to handle special situations. We will address some of these in the next chapter.

Chapter 13: Binary and Mixed-Integer Programming

The general branch and bound approach described in the previous chapter can be customized for special situations. This chapter addresses two special situations:

- when all of the variables are binary (known as "Binary Integer Programming" or BIP),
- when some or all of the variables are integer-valued and the objective function and all of the constraints are linear (known as "Mixed Integer Programming", MIP, or "Mixed Integer Linear Programming", MILP).

Binary Integer Programming

In binary problems, each variable can only take on the value of 0 or 1. This may represent the selection or rejection of an option, the turning on or off of switches, a yes/no answer, or many other situations. We will study a specialized branch and bound algorithm for solving BIPs, known as Balas Additive Algorithm. It requires that the problem be put into a standard form:

- The objective function has the form $\text{minimize } Z = \sum_{j=1}^{n} c_j x_j$
- The m constraints are all inequalities of the form $\sum a_{ij} x_j \geq b_i \text{ for } i = 1,2,..m$
- All of the x_j where j=1,2,…n are binary variables (can only have a value of 0 or 1).
- All objective function coefficients are non-negative.
- The variables are ordered according to their objective function coefficients so that $0 \leq c_1 \leq c_2 \leq \ldots \leq c_n$.

This may seem like a restrictive set of conditions, but many problems are easy to convert to this form. For example, negative objective function coefficients are handled by a change of variables in which x_j is replaced by ($1-x_j$'). It is also easy to reorder the variables. Constraint right hand sides can be negative, so ≤ constraints are easily converted to ≥ form by multiplying through by -1. The biggest restriction is that Balas Additive Algorithm does not handle equality constraints.

The keys to how Balas Algorithm works lies in its special structure:

- The objective function sense is minimization, and all of the coefficients are nonnegative, so we would prefer to set all of the variables to zero to give the smallest value of Z.
- If we cannot set all of the variables to 0 without violating one or more constraints, then we prefer to set the variable that has the smallest index to 1. This is because the variables are ordered so that those earlier in the list increase Z by the smallest amount.

These features also affect the rest of the branch and bound procedures. Branching is simple: each variable can only take on one of two values: 0 or 1. Bounding is the interesting part. Balas algorithm does not perform any kind of look-ahead to try to complete the solution or a simplified version of it. Instead the bounding function just looks at the cost of the next cheapest solution that *might* provide a feasible solution. There are two cases, as described next.

If the current variable is set to 1, i.e. $x_N = 1$, then the algorithm assumes that this *might* provide a feasible solution, so the value of the bound is $\sum_{j=1}^{N} c_j x_j$. Because of the variable ordering, this is the cheapest thing that can happen. Note that x_N ("*x* now") is not the same as x_n (the last *x* in the list).

If the current variable is set to 0, i.e. $x_N = 0$, then things are a little different. Recall that we need to calculate a bounding function value only for nodes that are currently infeasible. In this case, one of the ≥ constraints is not yet satisfied, i.e. the left hand side is less than the right hand constant. But if we set the current variable to zero, the left hand side value of the violated constraint(s) will not change. Therefore we must set at least one more variable to 1, and the cheapest one available is x_{N+1}, so the bounding function value is $\sum_{j=1}^{N} c_j x_j + c_{N+1}$. As before, the algorithm assumes that this cheapest setting *might* provide a feasible solution, so it proceeds.

It is easy to determine whether the solution proposed by the bounding function is feasible: just assume that all of the variables past x_N (when $x_N = 1$) or past x_{N+1} (when $x_N = 0$) take on the value of zero and check all of the constraints. If all of the constraints are satisfied at the solution proposed by the bounding function, then the node is fathomed, since it is not possible to get a lower value of the objective function at any node that descends from this one (the bounding function has made sure of that). All that remains is to compare the solution value at the node to the value of the incumbent, and to replace the incumbent if the current node has a lower value.

Infeasibility pruning is also worthwhile in this algorithm since it is easy to determine when a bud node can never develop into a feasible solution, no matter how the remaining variables are set. This is done by examining each constraint one by one. For each constraint, calculate the largest possible left hand side value, given the decisions made so far, as follows: (*left hand side value for variables set so far*) + (*maximum left hand side value for variables not yet set*). The second term is obtained by assuming that a variable will be set to 0 if its coefficient is negative and 1 if its coefficient is positive. If the largest possible left hand side value is still less than the right hand side constant, then the constraint can never be satisfied, no matter how the remaining variables are set, so this node can be eliminated as "impossible". For example, consider the constraint $-4x_1 - 5x_2 + 2x_3 + 2x_4 - 3x_5 \geq 1$. Suppose that both x_1 and x_2 have already been set to 1, while the remaining variables have not yet been set. The largest possible left hand side results if x_3 and x_4 are set to 1 while x_5 is set to 0, giving a left hand side value of (–9) + (4) = –5, which is not ≥ 1, hence the partial solution (1,1,?,?,?) cannot ever yield a feasible solution, so the node is fathomed as "impossible".

Balas Additive Algorithm uses depth-first node selection. It is of course possible to replace this by another node selection strategy, such as best-first, but if you do so then you are no longer following Balas' algorithm, strictly speaking. If your problem formulation states that you are using the Balas algorithm, then you must use depth-first node selection.

Consider the following example:

Minimize $Z = 3x_1 + 5x_2 + 6x_3 + 9x_4 + 10x_5 + 10x_6$

Subject to: (1) $-2x_1 + 6x_2 - 3x_3 + 4x_4 + x_5 - 2x_6 \geq 2$

(2) $-5x_1 - 3x_2 + x_3 + 3x_4 - 2x_5 + x_6 \geq -2$

(3) $5x_1 - x_2 + 4x_3 - 2x_4 + 2x_5 - x_6 \geq 3$

and x_j binary for $j=1,2\ldots6$

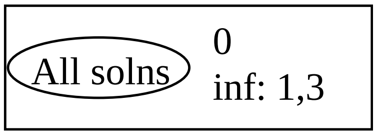


Figure 13.1: Root node.

Note that the variables are already ordered as required by Balas' algorithm. There are $2^6 = 64$ possible solutions if they are all enumerated, but we expect that Balas' algorithm will generate far fewer full and partial solutions. The branch and bound tree develops as shown in the following diagrams. For each new node, check these things in order:

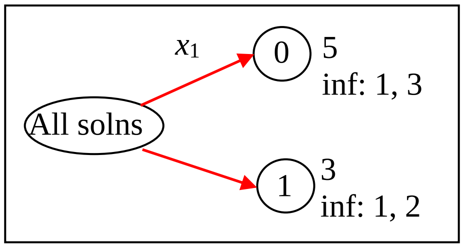


Figure 13.2: First branching.

(i) If it is a 1-node, check the feasibility by evaluating all of the constraints. Note that a 0-node is always infeasible for the same reasons as its parent node since the left hand side value of the constraint will not have changed. If the 1-node is feasible, then the node is fathomed.

(ii) For both 0-nodes and 1-nodes, check whether the decisions made thus far make it impossible to satisfy all constraints at any descendent node. If yes, then prune the node.

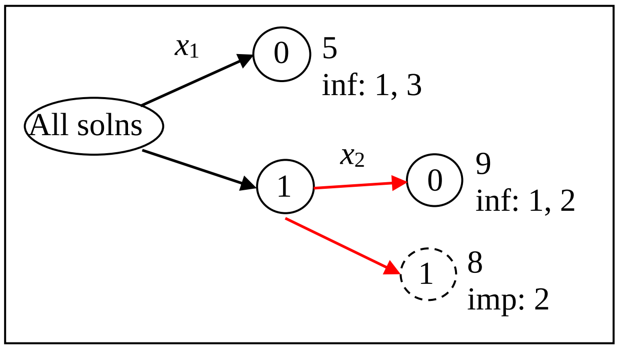


Figure 13.3: Second branching.

(iii) If it is a 0-node, check whether the bounding function solution is feasible by evaluating it in all of the constraints. If it is feasible then the node is fathomed with the bounding function solution.

Figure 13.1 shows the root node, which assumes that all variables can be set to 0, the cheapest solution. This gives an objective function value of 0 and is infeasible by constraints 1 and 3. The bounding function solution is also infeasible, so the root node must be expanded.

From here on, each node is labelled with the bounding function value, and an indication of status for the current solution that the node represents (note that this is not the same as the bounding function solution for zero nodes). *Inf* indicates that the node is currently infeasible and *Imp* indicates that the node has been found to be impossible to satisfy; each of these notations is followed by an indication of which constraints are causing infeasibility or impossibility.

Figure 13.2 shows the branching on x_1, the first variable. Notice that if x_1 is set to 1, then the bounding function assumes that we will not have to set any more variables to 1, hence the bounding function value is 3, since $c_1 = 3$. However, if x_1 is set to 0, then at least one more

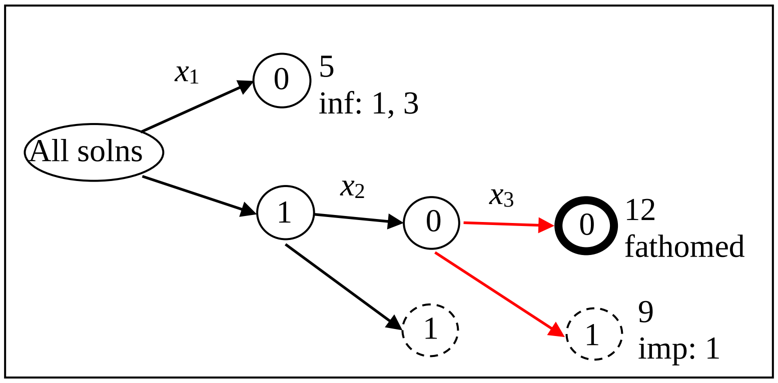


Figure 13.4: Third branching.

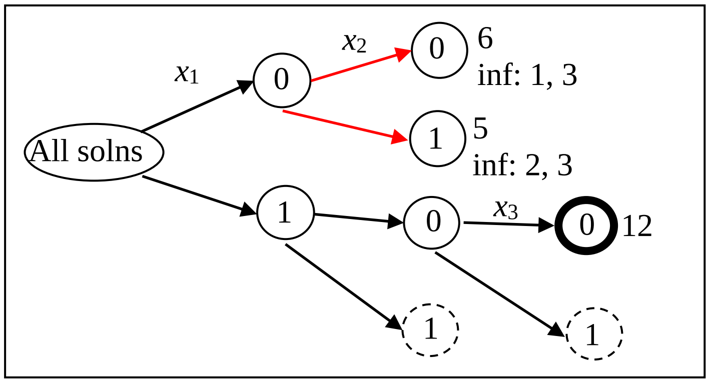


Figure 13.5: Fourth branching.

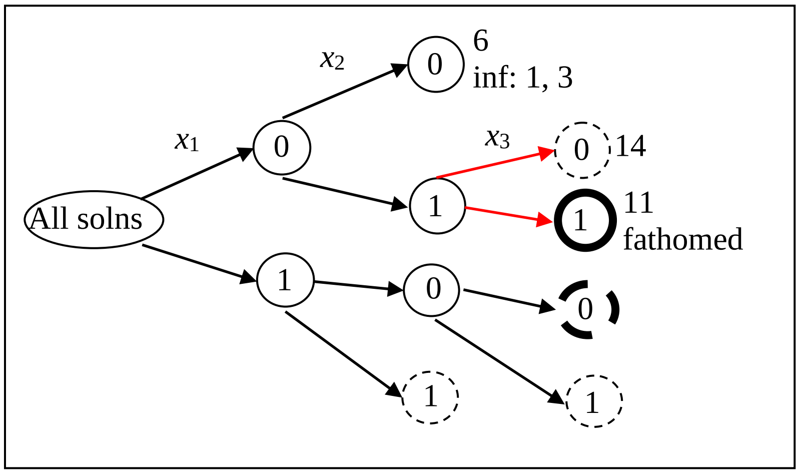


Figure 13.6: Fifth branching.

variable must be set to 1, and the bounding function assumes that the cheapest variable, x_2, the next variable in the ordered list, is the one that will be set to 1, hence the bounding function value is 5 because $c_2 = 5$. The bounding function does not yield a feasible solution at either node.

There are still live bud nodes on the developing branch and bound tree, so we choose the next node to expand, via the depth-first rule. Here we have just the two nodes to choose from, and since this is a minimization problem, we choose the node having the smallest value of the bounding function, and expand it using the next variable in the list, x_2. This is shown in Figure 13.3. Note that node (1,1,?,?,?,?) is impossible by constraint 2. Once x_1 and x_2 are set to 1, no matter how the remaining variables are set, the left hand side will never be greater than –2

Now we select the next node for expansion. Since we are using depth-first node selection, we are left only a single bud node to choose: (1,0,?,?,?,?), with a bounding function value of 9. Best-first node selection would have chosen node (0,?,?,?,?,?) because it has a lower bounding function value of 5.

The expansion of node (1,0,?,?,?,?) is shown in Figure 13.4. Node (1,0,0,?,?,?) is fathomed and found to be feasible when using the bounding function solution (1,0,0,1,0,0), with an objective function value of 12. This is our first incumbent solution. Node (1,0,1,?,?,?) is found to be impossible by constraint 1.

At this point, both of the nodes just created are not eligible for further expansion, so we back up the tree, looking for a level at which one of the nodes is unexplored. Figure 13.5 shows that the next branching occurs at the node (0,?,?,?,?,?).

Both child nodes are infeasible, but not impossible. The node having the lowest bounding function value is chosen for further branching, as shown in Figure 13.6.

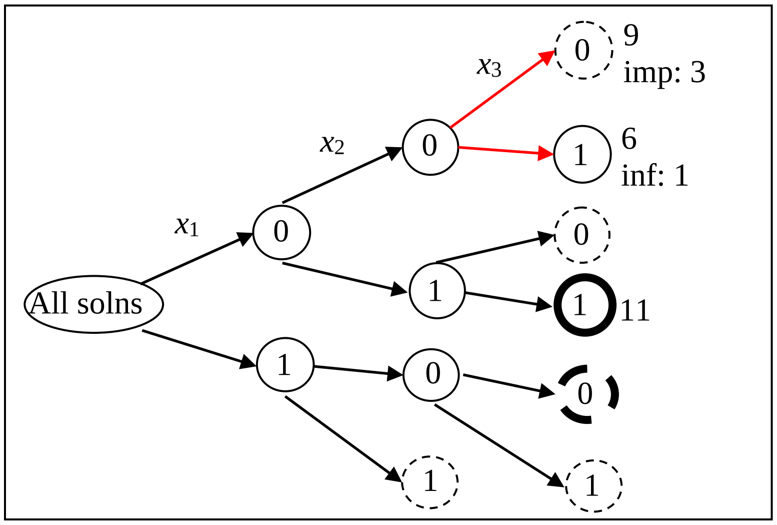


Figure 13.7: Sixth branching.

The new zero node created has a bounding function value of 14, worse than the incumbent, so it is pruned immediately, without bothering to check feasibility or impossibility. The new one node is fathomed, and the associated solution (0,1,1,0,0,0) has an objective function value of 11. This is lower than the previous incumbent, so it becomes the new incumbent, and the node associated with the previous incumbent is pruned. There is still a live node that has a promising value of the bounding function: (0,0,?,?,?,?), so this node is expanded next, as shown in Figure 13.7.

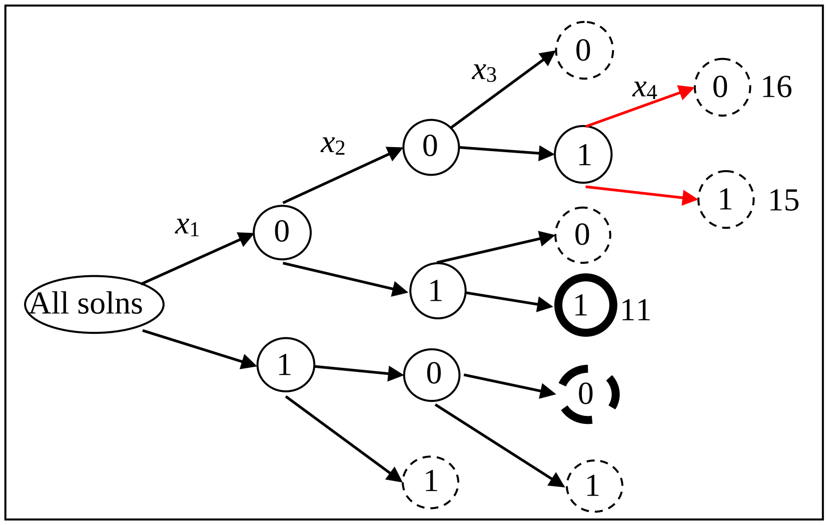


Figure 13.8: Seventh branching.

There is again just one node to expand: (0,0,1,?,?,?), as shown in Figure 13.8. Both child nodes have bounding function values that are worse than the objective function value of the incumbent, and so are pruned.

There are no more bud nodes, so the branch and bound solution is complete and the incumbent solution is the optimum: (0,1,1,0,0,0) with an objective function value of 11.

Note that we examined 15 nodes in total, including the root node. So we did 15/64 = 23% of the work as compared to a full enumeration. This is a good reduction of effort, but we should expect to see much more dramatic speed-ups on larger problems where early pruning cuts off branches that could have had numerous descendants.

Balas' algorithm is just one way of dealing with binary problems. More general methods can also be used, such as the techniques for mixed-integer programming that we will explore next.

Mixed-Integer Linear Programming

A mixed-integer programming (MIP) problem results when some of the variables in your model are real-valued (can take on fractional values) and some of the variables are integer-valued. The model is therefore “mixed”. When the objective function and constraints are all linear in form, then it is a mixed-integer linear program (MILP). In common parlance, MIP is often taken to mean MILP, though mixed-integer nonlinear programs (MINLP) also occur, and are much harder to solve. As you will see later, MILP techniques are effective not only for mixed problems, but also for pure-integer problems, pure-binary problems, or in fact any combination of real-, integer-, and binary-valued variables.

Mixed-integer programs often arise in the context of what would otherwise seem to be a linear program. However, as we saw in the previous chapter, it simply doesn’t work to treat the integer variable as real, solve the LP, then round the integer variable to the nearest integer value. Let’s take a look at how integer variables arise in an LP context.

Either/Or Constraints

Either/or constraints arise when we have a choice between two constraints, only one of which has to hold. For example, the metal finishing machine production limit in the Acme Bicycle Company linear programming model is $x_1 + x_2 \le 4$. Suppose we have a choice between using that original metal finishing machine, or a second one that has the associated constraint $x_1 + 1.5x_2 \le 6$. We can use either the original machine or the second one, *but not both*. How do we model this situation?

An important clue lies in observing what happens if you add a large positive number (call it M) to the right hand side of a ≤ constraint, e.g. $x_1 + x_2 \le 4 + M$. This now says $x_1 + x_2 \le$ “very big number”, so any values of x_1 and x_2 will satisfy this constraint. In other words, the constraint is eliminated. So what we want in our either/or example is the following:

either	$x_1 + x_2 \le 4$	← only this constraint holds
	$x_1 + 1.5x_2 \le 6 + M$	
or	$x_1 + x_2 \le 4 + M$	
	$x_1 + 1.5x_2 \le 6$	← only this constraint holds

We can achieve an equivalent effect by introducing a single binary variable (call it y), and using it in two constraints, both of which are included in the model, as follows:

(1) $x_1 + x_2 \le 4 + My$

(2) $x_1 + 1.5x_2 \le 6 + M(1-y)$

Now if $y = 0$ then only constraint (1) holds, and if $y = 1$ then only constraint (2) holds, exactly the kind of either/or behaviour we wanted. The downside, of course, is that a linear program has been converted to a mixed-integer program that is harder to solve.

k out of N Constraints Must Hold

This is a generalization of the either/or situation described above. For example, we may want any 3 out of 5 constraints to hold. This is handled by introducing *N* binary variables, $y_1 ... y_N$, one for each constraint, as follows:

$$f_1(\boldsymbol{x}) \leq b_1 + My_1$$

$$\ldots$$

$$f_N(\boldsymbol{x}) \leq b_N + My_N$$

and including the following additional constraint:

$$\sum_{i=1}^{N} y_i = N - k$$

This final constraint works as follows: since we want *k* constraints to hold, there must be N-*k* constraints that *don't* hold, so this constraint insures that N-*k* of the binary variables take the value 1 so that associated M values are turned on, thereby eliminating the constraint.

Functions Having N Discrete Values

Sometimes you have a resource that is available in only certain discrete sizes. For example, the metal finishing machine may be an equality that has 3 settings: $x_1 + x_2 = 4$ or 6 or 8. This can be handled by introducing one binary variable for each of the right hand side values. Where there are N discrete right hand sides, the model becomes:

$$f(x) = \sum_{i=1}^{N} b_i y_i \text{ and } \sum_{i=1}^{N} y_i = 1$$

This assures that exactly one of the right hand side values is chosen. In the metal finishing machine example, the model would be:

$$x_1 + x_2 = 4y_1 + 6y_2 + 8y_3$$

$$y_1 + y_2 + y_3 = 1$$

and y_1, y_2, y_3 binary.

Fixed Charges and Set-up Costs

Fixed charges or set-up costs are incurred when there is some kind of fixed initial cost associated with the use of *any* amount of a variable, even a tiny amount. For example, if you wish to use any amount at all of a new type of metal finishing for the ABC Company, then you incur a one-time set-up cost for buying and installing the required new equipment. Fixed charges and set-up costs occur frequently in practice, so it is important to be able to model them.

Mathematically speaking, a set-up charge is modelled as follows:

$$f(x_j) = \begin{cases} 0 & \text{if } x_j = 0 \\ K + c_j x_j & \text{if } x_j > 0 \end{cases}$$

where K is the fixed charge. This says that there are no charges at all if the resource represented by x_j is not used, but if it is used, then we incur both the fixed charge K and the usual charges associated with the use of x_j, represented by $c_j x_j$.

The objective function must also be modified. It becomes:

$$\text{minimize } Z = f(x_j) + (\text{rest of objective function})$$

Note the minimization: set-up costs are only interesting in the cost-minimization context. If it is cost-maximization (a strange concept…) then we would of course *always* incur the set-up cost by insuring that every resource was always used.

The final model introduces a binary variable y that determines whether or not the set-up charge is incurred:

$$\text{Minimize } Z = [Ky + c_j x_j] + (\text{rest of objective function})$$

$$\begin{aligned} \text{subject to:} \quad & x_j - My \le 0 \\ & \text{other constraints} \\ & y \text{ binary} \end{aligned}$$

This behaves as follows. If $x_j > 0$, then the first constraint insures that $y = 1$, so that the fixed charge in the objective function is properly applied. However, if $x_j = 0$, then y could be 0 or 1: the first constraint is not restrictive enough in this sense. We want y to be zero in this case so that the set-up cost is not unfairly applied, and something in the model actually does influence y to be zero. Can you see what it is? It is the minimization objective. Given the free choice in this case between incurring a set-up cost or not, the minimization objective will obviously choose not to. Hence we have exactly the behaviour that we wish to model. Once again, though, we have converted a linear program to a mixed-integer linear program.

Dakin's Algorithm for Solving Mixed-Integer Linear Programs

Now that we've seen how integer or binary variables can enter linear programs, we need a method for solving the resulting mixed-integer problems. Because of the integer or binary variables, we will need to use some kind of branch and bound approach. Dakin's algorithm is a branch and bound method that uses an interesting bounding function: it simply ignores the integer restrictions and solves the model as though all of the variables were real-valued! This *LP-relaxation* provides an excellent bound on the best objective function value obtainable, and sometimes results in a feasible solution when all of the integer variables actually get integer values in the LP solution.

The second aspect of Dakin's algorithm is the branching. As a node is expanded, two child nodes are created in which new variable bounds are added to the problem. We first identify the candidate variables: those integer variables that did not get integer values in the LP-relaxation solution associated with the node. One of these candidate variables is chosen for branching. Let's consider candidate variable x_j that has a non-integer value that is between the next smaller integer k and the next larger integer k+1. The branching then creates two child nodes:

- the parent node LP plus the new variable bound $x_j \le k$
- the parent node LP plus the new variable bound $x_j \ge k+1$

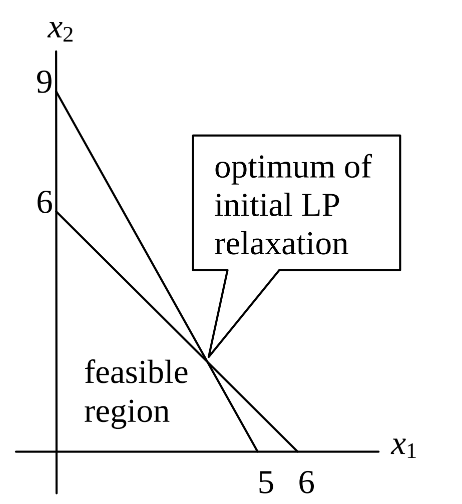


Figure 13.9: Initial LP relaxation.

For example, consider some candidate variable x_j that has the value 5.761. One child node will consist of the parent LP plus the new variable bound $x_j \leq 5$ and the other child node will consist of the parent LP plus the new variable bound $x_j \geq 6$. These new nodes force x_j away from its current non-integer value. There is no guarantee that it will get an integer value in the next LP-relaxation, however (though this often happens).

Fathoming is simple. If the LP-relaxation at a node assigns integer values to all integer variables, then the solution is integer-feasible, and is the best that can be attained by further expansion of that node. The solution value is then compared to the incumbent and replaces the incumbent if it is better. If the LP-relaxation is LP-infeasible, then the node and all of its descendents are infeasible, and it can be pruned.

Node selection is normally depth-first. This is because a child node is exactly the same as the parent node, except for one changed variable bound. This means that the LP basis from the parent node LP-relaxation solution can be used as a hot start for the child node LP-relaxation. Of course the parent node solution will definitely be infeasible in the child node, but a dual simplex restart can be used and will very quickly iterate to a new optimum solution. This is much more efficient than, say, best-first node selection, which would require restarting LPs from scratch at most nodes.

Let's look at a small example that appears in a textbook by Winston (*Mathematical Programming*, 1991, p. 489):

Maximize Z = $8x_1 + 5x_2$
s.t. $x_1 + x_2 \leq 6$
$9x_1 + 5x_2 \leq 45$
x_1, x_2 are integer and nonnegative.

A graph of this problem is shown in Figure 13.9. The initial LP relaxation is created when the original model, shown above, is simply treated as an LP and solved. The LP-optimum occurs at (3.75, 2.25) with Z = 41.25. However this is not integer-optimum since both integer variables have taken on fractional values, so we must develop a branch and bound tree following Dakin's algorithm.

This branch and bound tree develops as shown in Figure 13.10. Small sketches of the feasible region for some of the LP-relaxations are also shown on the diagram. Note how the new variable bounds gradually "square off" the LP-relaxation feasible regions until solutions are found in which both of the integer variables indeed have integer values.

Each node in Figure 13.10 represents the solution of an LP-relaxation. Numbers indicate the order of the solutions. An incumbent solution of (3,3) with Z = 39 is obtained early, at the

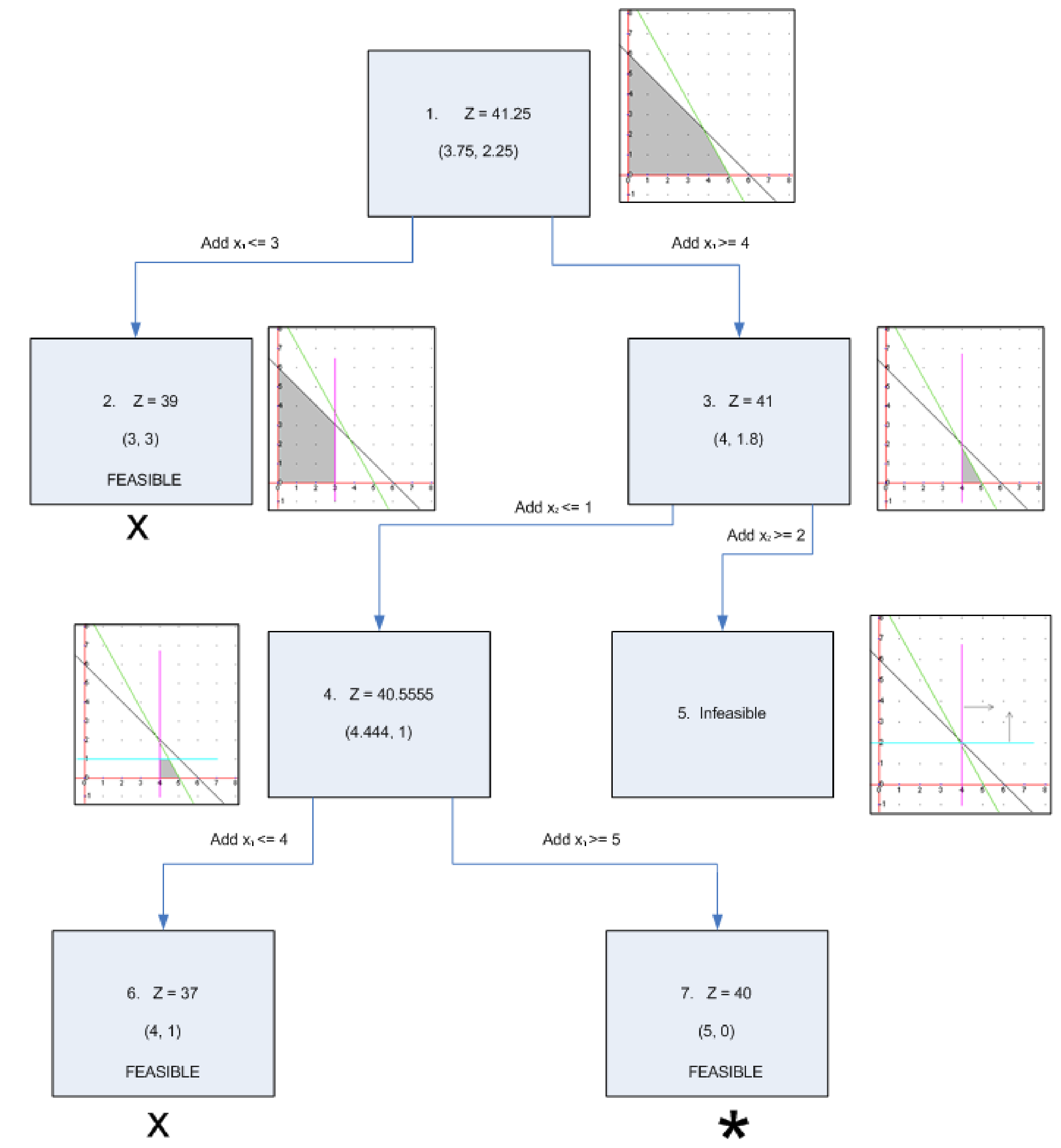


Figure 13.10: Branch and bound tree for Dakin's algorithm.

second LP solution. However there are other promising nodes with better bounding function values, so tree development continues. A better incumbent is eventually found: node 7 with a solution of (5,0) and Z = 40. At this point, all other nodes are pruned and the solution process halts. Note how far the final solution of (5,0) is from the initial LP-relaxation solution of (3.75,2.25): you can't get to the final solution by rounding!

You will not need to set up the MILP branch and bound tree manually in practice. Most commercial LP solvers will accept integer or binary restrictions on variables as part of their input. They then take care of setting up the branch and bound tree automatically. As you can imagine though, MILP solutions generally take a *lot* longer than identical LP solutions!

A number of further improvements on the basic algorithm are used in commercial MILP solvers:

- Which candidate variable is selected for branching can have a big impact, so there are more advanced algorithms for making this decision.
- You don't normally solve the LP relaxation for both child nodes after branching. You normally just solve the up-branch (in which the lower bound was increased), or just the down-branch, or you use a more sophisticated algorithm for choosing the branching direction. Depth-first search then continues without solving the LP-relaxation for the sibling node. If needed, the search will backtrack to the sibling node eventually.
- A variety of sophisticated algorithms are available for choosing the next node to expand when the depth-first search must backtrack.
- The root node is subjected to intensive analysis before branch and bound begins in the hopes of greatly reducing the subsequent amount of work.
- Many other advanced techniques are also available, like probing algorithms that explore the region around promising solutions. MILP is an active area of research!

Chapter 14: Heuristics for Discrete Search: Genetic Algorithms and Simulated Annealing

The branch and bound algorithms that we have studied thus far have one very nice property: they guarantee that the optimum solution will be found. But branch and bound also has one fatal flaw: it is combinatorially explosive, and hence will take excessive time (and possibly computer memory) for problems that are larger than medium scale. Further, discrete problems of large scale are very common in practice, e.g. scheduling (shift workers, exams, airline flights, etc.). But these problems still need to be solved, so we have to give up on finding the optimum solution and instead concentrate on finding a pretty good solution within the limits of time and computer memory available.

This means that we need to employ *heuristic* methods. A heuristic is a method that is not guaranteed to find the optimum, but usually gives a very good solution, though it cannot guarantee to do even that every time. Heuristics are "quick and dirty" methods, generally relatively fast and relatively good. We have actually studied a couple of heuristic methods already in Chapter 12: beam search, and stopping branch and bound with a guarantee of closeness to optimality. Here is a rough guide to when to use various discrete search methods:

Problem Size	**Methods**
small	Enumeration
medium	Branch and bound Dynamic programming A* search
large	Branch and bound variants: • Beam search • Guarantee of closeness to optimality Problem-specific heuristics Controlled random search: • Genetic algorithms • Simulated annealing • Tabu search Pure random search

In the rest of this chapter we will look at two popular heuristic methods that are applicable to a very wide range of practical problems.

Genetic Algorithms

These are fascinating algorithms. The name derives from the way in which they loosely mimic the process of evolution of organisms, where a problem solution stands in for the organism's genetic string. Features include a survival of the fittest mechanism in which potential solutions in a population are pitted against each other, as well as recombination of solutions in a mating process and random variations. The incredible part is that this heuristic can "evolve" better and better solutions without any deep understanding of the problem itself! Genetic algorithms can be applied to any problem that has these two characteristics: (i) a solution can be expressed as a string, and (ii) a value representing the worth of the string can be calculated.

Genetic algorithms have a couple of important advantages. They are simple to program and they work directly with complete solutions: unlike branch and bound, there is no need for estimates or for bounding functions.

As an example, let's look again at a variation of the person-job assignment problem. Let me stress that in practice the best way to solve this problem is actually by the exact and fast assignment problem linear program. However this is an easy-to-understand problem that we have worked with before, so we will see how it can be solved via a genetic algorithm. In this example we are assigning salespeople to regions, and the table below shows the expected number of units sold if a salesperson is assigned to a region.

		Region			
		1	**2**	**3**	**4**
Salesperson	**A**	20	37	15	28
	B	25	24	18	29
	C	18	30	14	24
	D	21	33	16	20
	E	23	31	19	23

Our objective is to maximize the number of units sold. Further, since there are only 4 regions to cover, we must assign just 4 of the 5 salespeople (each salesperson can handle only one region). Which of the 4 salespeople should be chosen, and how should they be assigned to the regions to maximize the total number of units sold?

Let's first check that a genetic algorithm can be applied to this problem. Can a solution be expressed as a string? Yes: a solution such as CDAB can represent the assignment of salesperson C to region 1, salesperson D to region 2, salesperson A to region 3 and salesperson B to region 4. Can a value be assigned to a string to represent its value? Yes: simply add up the expect units sold for the solution; for example the value associated with string CDAB would be 18 + 33 + 15 + 29 = 95.

Now we can use this example to explore a very basic genetic algorithm approach to solving this problem. At all times we will have a *population* consisting of numerous solution strings. Each string is analogous to a genetic string of chromosomes. The solutions will compete with each other in a survival of the fittest contest where their chances of survival are proportional to the relative "goodness" of their solution string value. Parts of surviving strings are then combined in various ways through a process similar to male-female reproduction to create a population of new child strings. Some of these may be randomly changed as happens in real life through e.g. bombardment via cosmic rays. Now we have a new population, and the process repeats. Amazingly, after this cycle repeats a number of times, there are usually much better solutions in the current population than in the original. Note however that the process is not entirely random: good solutions have a better chance of survival, and a better chance or reproduction, and reproduction tends to combine parts of stronger solutions into even better ones. Good characteristics tend to persist in the population and to combine in useful ways.

There are three main *operators* in a basic genetic algorithm: reproduction, crossover, and mutation. We will examine each of these in turn. First, however, it is necessary to establish an initial population of solutions. The simplest (but probably not the best) way to create an initial population is generate it randomly. We will discuss better ways later. The size of the population (i.e. how many solutions there should be) is also an important parameter: it must be large enough that it can support sufficient genetic variation, but not so large that calculations take an inordinate amount of time. In practice, the population size is often determined by experimentation.

The Reproduction Operator

The reproduction is equivalent to the "survival of the fittest" contest. It determines not only which solutions survive, but how many copies of each of the survivors to make. This will be important later during the crossover operation. The probability of survival of a solution is proportional to its solution value; also known as its *fitness* (the function that assigns values to solution strings is also known as the *fitness function*).

As an example, consider a population of 4 solution strings from our small salesperson assignment problem, and the relative fitness of each string:

String	Fitness (solution value)	Fitness as % of total
CDAB	95	95/373 = 25.5%
BADC	102	102/373 = 27.3%
BCDA	99	99/373 = 26.5%
CBAD	77	77/373 = 20.7%
fitness total	*373*	*373/373 = 100.0%*

The first 3 solutions are relatively evenly matched, though the fourth solution is a bit weaker. How will we decide which ones survive? Conceptually, we construct a virtual weighted roulette wheel, as shown in Figure 14.1, where the weight of any solution is proportional to the "fitness as % of total" shown in the table above. "Spinning the wheel" by generating a random number selects a solution string to reproduce a copy of itself into a new intermediate population known as the *mating pool* for reasons that will be clear soon. If we chose a population of size *n*, then the wheel is spun *n* times to create a mating pool of size n. In our small example since the population size is 4, then the wheel is spun 4 times.

In reality we "spin the roulette wheel" by generating a uniformly distributed random number between 0 and 100. The solution is then selected based on the cumulative sum of the fitness relative weights. For the example in the table and in Figure 14.1, we spin the wheel and select as follows:

- If the random number is between 0 and 25.5, then select CDAB,
- If the random number is between 25.6 and 25.5+27.3=52.8, then select BADC,
- If the random number is between 52.9 and 25.5+27.3+26.5=79.3, then select BCDA,
- If the random number is between 79.4 and 100.0, then select CBAD.

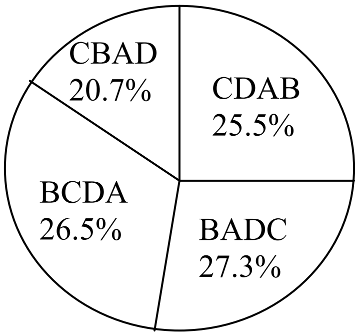


Figure 14.1: A virtual roulette wheel.

Note that it is entirely possible for one of the solutions to be selected more than once, and for some solutions not be chosen at all. In general it is most likely that the stronger (most fit) solutions will be chosen (i.e. survive) most often, and that the weaker (most unfit) solutions will not be chosen (i.e. die). However, due to the random nature of the process, it is also possible for a weak solution to be chosen multiple times and for a strong solution to die, but this is unlikely.

After the reproduction operation, we have an intermediate population known as the mating pool that is ready to mix and mingle, akin to the process of mating and reproducing children that share some of the genetic material of each parent. This is the function of the *crossover* operator.

The Crossover Operator

During crossover, two parent solution strings from the mating pool combine to create two new child solution strings. This happens as follows:

1. Randomly select two parent strings from the mating pool.
2. Randomly select a *crossover point* in the solution string. This is the point between any two positions in the solution string.
3. Swap the ends of the two parent strings, from the crossover point to the end of the string, to create two new child strings.

parent strings *child strings*

XXXX|XX → XXXXOO
OOOO|OO OOOOXX
↑
crossover point

Figure 14.2: Illustration of crossover.

This process is illustrated in Figure 14.2, where X and O represent values in the two solution strings. In our example we might see a crossover such as:

BC|DA → BCAD
CB|AD CBDA

There are numerous variations on the basic crossover operator, for example randomly choosing *two* crossover points and swapping the string contents between those two crossover points.

Of course, it is entirely possible that crossover will produce infeasible children, as for example:

CDA|B	→	CDAC
BAD|C		BADB

In this case, both children are infeasible because they both contain repeated salespeople, and each salesperson can handle just one region.

How are we to handle the problem of infeasible child strings? The best way is to use a different variant of crossover that does not allow infeasible children to be created at all: we will describe one such variant (*partially-matched crossover)* later. If infeasible children are relatively infrequent, they can be handled by simply rejecting the infeasible child and applying the crossover operator again. Finally, if there is no better crossover operator and infeasibility is relatively frequent then you can accept the infeasible child, but penalize its fitness. In our example, we could adjust the fitness downwards, e.g. by 10 points for every repeated salesperson in a solution string (or by a squared factor, or many other ways).

The new population is now almost ready. There is one last operator to apply.

The Mutation Operator

The mutation operator is used to randomly alter the values of some of the positions in some of the strings based on a parameter that determines the level of mutation. One common choice is a 1 in 1000 chance of mutation. This can be implemented as follows. For each position in each string, generate a random integer between 1 and 1000. If this number is 1, then the position is chosen for mutation, and is randomly switched to any other possible value. In our example, the second position in the string CBAD might be chosen for mutation and might randomly switched from a value of B to a value of E. This is an improvement: CBAD has a fitness of 77, while CEAD has a fitness of 84.

Of course it is just as possible that the mutation could worsen the fitness function or even generate an infeasible solution. Given this downside, why do we bother with mutation at all? There is a very good reason. For a clue take a look at the set of solutions that comprised the original population in our example (see table on page 3). What do you notice about that set of solutions?

Salesperson E is not present in *any* of the solutions in that initial population! And there is no way that salesperson E will be introduced by either the reproduction or crossover operators. The *only* way that salesperson E might appear in a solution is via mutation. Now we see the motivation behind mutation: to sample the solution space widely. So where reproduction and crossover try to concentrate the solutions that we already have into better solutions, mutation works instead to sample the solution space and to broaden the search.

Mutation is a vital part of the solution process, and the mutation rate can have a big impact on the quality of the final solution. It is even possible (though vastly more inefficient) to solve problems using *only* the mutation operator.

Overview of the Basic Genetic Algorithm Process

Now that we've seen the basic genetic algorithm operators, we can put the whole process together. Here are the essential steps:

0. Design the algorithm: choose the population size *n* and mutation rate; choose the operators and the stopping conditions (more on stopping conditions later).
1. Randomly generate an initial population (more on generating the initial population later) and calculate the fitness value for each string. Set the incumbent solution as the solution with the best value of the fitness function in the initial population.
2. Apply the reproduction operator to the current population to generate a mating pool of size *n*.
3. Apply the crossover operator to the strings in the mating pool to generate a tentative new population of size *n*.
4. Apply the mutation operator to the tentative new population to create the final new population. Calculate the fitness values of the solution strings in the new population and update the incumbent solution if there is a better solution in this population.
5. If the stopping conditions are met, then exit with the incumbent solution as the final solution. Otherwise go to Step 2.

As you can see, this process generates a series of populations, each of size *n*. Unlike the other optimization algorithms we have looked at that keep track of a single developing solution, a genetic algorithm keeps track of *n* solutions simultaneously. Some of these are good solutions and others are poor, but the diversity of the population turns out to be important in generating good new solutions. In fact, some genetic algorithm implementations suffer from *premature convergence*, which happens when one solution is so strong that it takes over the whole population, often by being almost the only solution to pass through the "survival of the fittest" test in the reproduction operator. This is not a good outcome since the later generations all become very similar with very little chance for useful new variations to arise.

Stopping Conditions

Evolution of creatures is obviously an ongoing process, so how do we decide when to stop the artificial evolutionary process in a genetic algorithm? This can be done in several ways, depending on the problem. The most obvious way is simply to stop after a prespecified number of populations have been created (each population is called a *generation*). But perhaps it would be better to stop when there is very little change between generations, indicating that the evolutionary process has reached a plateau.

It is not a good idea to stop when the incumbent solution has not changed for several generations, since this does not really measure the amount of ferment going on in the current population. To capture this, the genetic algorithm is sometimes stopped when the average population solution value has not changed for several generations. However even this measure does not always represent the amount of change going on in the current population. This is perhaps better represented by a surprising measure: stop when the *worst* solution string fitness in the population has not changed for several generations. It is the worst solution value that usually changes the most between generations; when it settles down it is usually true that the whole population has settled down so that more useful new solutions are unlikely to arise.

A typical solution trajectory is shown in Figure 14.3. Note how the average population fitness varies up and down but generally trends upward. A plot of the worst solution value would have a similar trajectory, but likely with a lot more variation between generations. A similar plot of the incumbent solution value, shown in Figure 14.4, tends to have longer and longer periods of stability, but always improves (since by definition the incumbent solution is the best solution seen so far).

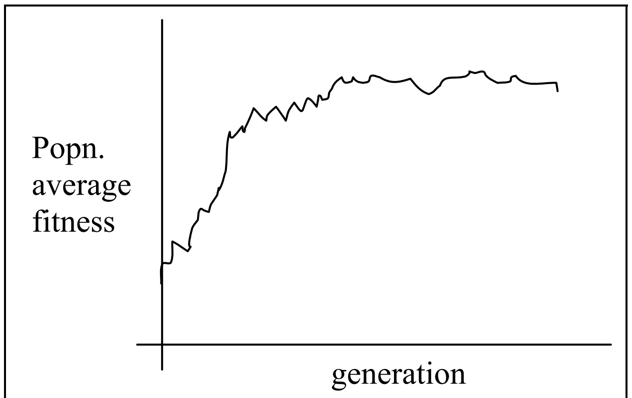


Figure 14.3: Typical solution trajectory.

There are many variations of genetic algorithms. One variant that tends to smooth the solution trajectory is as follows: set the final new population by looking at the last population and the newly-generated population together (hence there will be 2*n* solutions). Select the *n* best solutions from this population and designate this as the final new population. The main difficulty with this approach is that some relatively poor solutions that could have developed into very good solutions later on are eliminated early.

Alternative Operators

Genetic algorithms are under constant development and new operators for special situations are constantly being developed. We describe two here as representative examples of other operators that could take the place of the crossover operators.

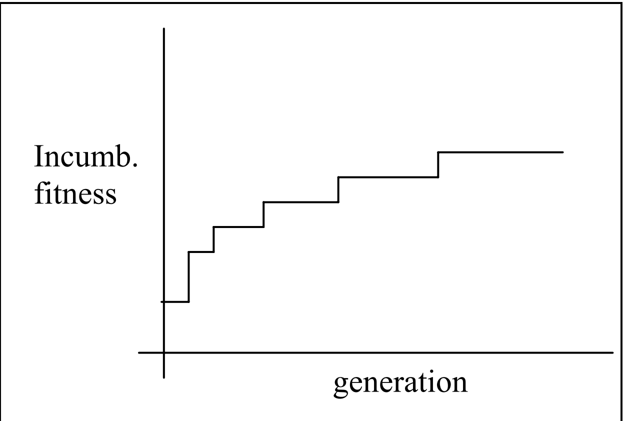


Figure 14.4: Typical incumbent fitness trajectory.

In the *inversion operator*, two inversion sites are randomly chosen on a single solution string. The order of the elements in the substring between the two inversion sites is then reversed. For example, ABC|DEF|GH → ABCFEDGH. You can see how this operator would prevent the creation of infeasible child solutions for the salesperson assignment problem because duplicate salespeople can never result from the inversion.

The *partially matched crossover operator* is similar to ordinary crossover with two crossover points. The difference is that special steps are taken to make sure that no duplication occurs in the resulting child solution strings. Consider the following example:

IHD|EFG|ACBJ
HGA|BCJ|IEDF

Normal crossover would produce two child solution strings that contain duplication outside the crossover zone, as shown in bold:

IHD|BCJ|A**CBJ**
H**G**A|EFG|**I**E**D**F**

Now partially-matched crossover uses the correspondence within the crossover zone to fix up the duplication by switching the values of the duplicated elements that are outside the crossover zone. The crossover zone contains |EFG| in the top string and |BCJ| in the bottom string, and the fix-up rules are derived directly from the correspondence between the elements in those two crossover zone substrings: E to B, F to C, G to J. To fix the duplication outside the crossover in the new top string, proceed this way: if there is a duplicated B replace it with E, if there is a duplicated C replace it with an F, and if there is a duplicated J replace it with a G. To fix the new bottom string, use the reverse rules: if there is a duplicated E replace it with a B, if there is duplicated F replace it with a C, and if there is duplicated G replace it with a J. The fixed strings then are:

IHD|BCJ|AFEG
HJA|EFG|IBDC

As you can see, there is now no duplication in the child solution strings. However, note also that existing substrings (such as ACBJ in the top string) are now also broken up. This may affect the quality of the child solution strings.

Which operators should you chose for your particular application? This depends on the application of course (e.g. whether duplication of elements in a string is allowed), but can often be decided only by some experimentation. Some applications for which genetic algorithms have been used with great success include VLSI circuit layout, scheduling, machine learning, optimizing communication link sizes, etc.

Pointers to Success with Genetic Algorithms

How well a genetic algorithm does depends partly on where it starts: i.e. the quality of the initial population. A randomly-generated initial population is usually of fairly low quality; the genetic algorithm will do much better if provided with a relatively high quality initial population. But the initial population must also include a certain amount of diversity. How might we generate a good quality initial population for the salesperson assignment problem?

One way is to as follows: (i) randomly select a salesperson and randomly assign that salesperson to a region, (ii) select the best unassigned salesperson-region combination and make that assignment, (iii) continue with step (ii) until sufficient salespeople have been assigned. This procedure will give you a semi-random but reasonably good solution, and can be repeated until you have sufficient solutions for the initial population. It also gives you some diversity. Here's another example: (i) randomly choose a region and assign the best salesperson for that region,

(ii) randomly choose an unassigned region and assign the best salesperson for that region, (iii) continue with step (ii) until there are no more regions needing a salesperson.

With some ingenuity, you can usually find a way to generate semi-random solutions that are relatively good. The genetic algorithm then has a head start. I have used this approach in devising a method to assign exam proctors to examinations at Carleton University. Interestingly, the average population fitness for the very first population generated this way was higher than the average population fitness for the final population generated by a genetic algorithm started at an entirely random population (though the genetic algorithm had improved the random initial population considerably).

The second pointer to success is to make sure that your operators are properly chosen. Using a poorly-chosen operator can slow the process considerably.

Finally, make sure that the values of the other control parameters (such as the population size and the mutation rate) are well-chosen. Though you can find rules of thumb for setting these values, sometimes you can only determine the best values by experimentation.

Simulated Annealing

Simulated annealing is another popular heuristic for both discrete and continuous problems. It was developed before genetic algorithms, and has gradually been superceded by them for many applications, though it is still much used. It is based on an analogy to the heat-treatment of metals (known as annealing). When metals are carefully annealed, usually by precise control of the cooling process, certain very desirable properties such as hardness or flexibility can be obtained.

In optimization by simulated annealing, when the “temperature” parameter in the heuristic is high, a great deal of random movement in the solution is tolerated, and as the “temperature” parameter is lowered, less and less random movement is allowed, until the solution settles into a final “frozen” state. This allows the algorithm to sample the solution space widely when the “temperature” is high, and then gradually move towards simple steepest ascent/descent as the “temperature” cools. The effect is to allow the solution to move out of local optima during the high temperature phase of the operation.

Here is an outline of this simple algorithm for the case of minimization of a cost function:

0. Start-up. Find an initial solution S, possibly by generating it randomly. Choose an initial (high) temperature $T > 0$. Choose a value for r, the rate of cooling parameter.
1. Choose a random neighbour of S and call it S'.
2. Calculate the difference in costs: $\Delta = \text{cost}(S') - \text{cost}(S)$.
3. Decide whether to accept the new solution or not: if $\Delta \leq 0$ (S' is better than S, or the same as S), then set $S = S'$, else (S' is worse than S) set $S = S'$ with probability $e^{-\Delta/T}$.
4. If the stopping conditions are met, then exit with S as the final solution, else reduce the temperature by setting $T = rT$, and go to Step 1.

A simple stopping condition is when *S* is “frozen”, i.e. has not changed value for several iterations.

The really interesting feature of a simulated annealing algorithm is how it will accept a worsening move with a certain probability. This probability declines as *T* declines; by analogy the randomness in the movements decrease as the temperature falls. When *T* is small enough the algorithm accepts only improving moves. This blending of random and purposeful search is surprisingly effective and has found many practical applications including layout of integrated circuits, routing and location problems, graph problems, etc. However running times can be long.

While the inspiration is simulated annealing, the more apt analogy for me is to a fly trying to find a way out of a container. Initially when it has a lot of energy it buzzes around wildly, but later when it tires it makes random moves less and less often and gradually settles into walking towards its goal. This is a blend of exploring widely and following up on promising paths. And flies are quite good at getting out of containers!

Chapter 15: Dynamic Programming

Dynamic programming is a general approach to making a sequence of interrelated decisions in an optimum way. While we can describe the general characteristics, the details depend on the application at hand. Most fundamentally, the method is *recursive*, like a computer routine that calls itself, adding information to a stack each time, until certain stopping conditions are met. Once stopped, the solution is unraveled by removing information from the stack in the proper sequence.

Here is an overview of the method:

1. Define a small part of the whole problem and find an optimum solution to this small part.
2. Enlarge this small part slightly and find the optimum solution to the new problem using the previously found optimum solution.
3. Continue with Step 2 until you have enlarged sufficiently that the current problem encompasses the original problem. When this problem is solved, the stopping conditions will have been met.
4. Track back the solution to the whole problem from the optimum solutions to the small problems solved along the way.

While this sounds new, you in fact already know how to solve a problem by dynamic programming: Dijkstra's shortest route algorithm is classic dynamic programming! The "small part of the problem" at each stage is simply to determine the next closest node to the origin. You enlarge this problem slightly at each stage by appending all of the unsolved nodes and arcs that are directly attached to a solved node. The stopping conditions are met when the next closest node is the destination node. Now you recover the solution by tracing back along the arcs in the arc set from destination node back to the origin.

There are several important characteristics of dynamic programming, as described next.

The problem can be divided into stages. In the shortest route problem, each stage constitutes a new problem to be solved in order to find the next closest node to the origin. In some dynamic programming applications, the stages are related to time, hence the name *dynamic* programming. These are often dynamic control problems, and for reasons of efficiency, the stages are often solved backwards in time, i.e. from a point in the future back towards the present. This is because the paths that lead from the present state to the future goal state are always just a subset of all of the paths leading forward from the current state. Hence it is more efficient to work backwards along this subset of paths. We will be dealing with static examples in which the direction is immaterial, but for form's sake we will also work backwards so that it will not be a surprise when you need to do so for a time-related dynamic control problem. This is known as *backwards recursion.*

Each stage has a number of states. Most generally, this is the information that you need to solve the small problem at a stage. For the shortest route problem, the state is the set of solved nodes, the arcs in the arc set, and the unsolved arcs and nodes directly connected to solved nodes.

The decision at a stage updates the state at the stage into the state for the next stage. In the shortest route problem the decision is the arc to add to the arc set and the corresponding unsolved node that is added to the solved set. This obviously affects the sets of solved and unsolved nodes and arcs and the arcs in the arc set. Hence the decision updates the state for the next stage.

Given the current state, the optimal decision for the remaining stages is independent of decisions made in previous states. This is the fundamental dynamic programming principle of optimality. It means that it is okay to break the problem into smaller pieces and solve them independently. For example, in the shortest route problem, we only care about the total distance from the origin to a solved node; we don't care about the actual route from the origin to that solved node. Decisions made thereafter use only the distance information, without regard to the actual route (i.e. the previous decisions).

There is a recursive relationship between the value of decision at a stage and the value of the optimum decisions at previous stages. In other words, the optimum decision at this stage *uses* the previously found optima. In a recursive relationship, a function appears on both sides of the equation. In words, the shortest route recursive relationship looks like this:

$$\begin{aligned}&(\text{length of shortest route from origin to node } i)\\ &\qquad = \min_{i,j}\{(\text{length of shortest route from origin to solved node } j)\\ &\qquad + (\text{length of arc from solved node } j \text{ to unsolved node } i)\}\end{aligned}$$

Note how the "length of shortest route" appears on both sides of the equation: it is recursive. All dynamic programming recursive relationships show the optimum function on both sides of the equation: the new optimum is always derived from the old optimum along with some local value. Note that the relationship need not be addition as it is here. It could be anything: multiplication or some other abstract relationship.

To formulate a dynamic programming solution, you must answer the following questions:

- What are the stages in the solution?
- How is the state defined at a stage?
- What kind of decision must you make at a stage?
- How does the decision update the state for the next stage?
- What is the recursive value relationship between the optimum decision at a stage and a previous optimum decision?

Students are expected to write out these items as part of their problem formulation.

It's time for an example to clarify all of this theory. For some reason, dynamic programming seems to be one of the less intuitive optimization methods and students seem to learn best by being shown several examples, hence that is what we will do next.

An Equipment Replacement Problem

A bicycle courier must obviously have a bicycle at all times that he can use for his deliveries; he can't be without one. Hence he wants to determine the cheapest way to buy and maintain bicycles over the 5-year timeframe that he is currently contemplating working as a bicycle courier before finishing his graduate program and moving on to other career options. He has collected some information about costs. A new Acme bicycle costs \$500. Maintenance costs vary with the age of the bike: \$30 in the first year, \$40 in the second year, and \$60 in the third year. Given the amount of use they get, bicycles will last a maximum of three years before they are sold. He figures he can get \$400 if he sells a bicycle after one year of use, \$300 after two years of use, and \$250 after three years of use. He has no bicycle at the moment. When should he buy bicycles and how long should he keep them to minimize his total costs over the 5-year timeframe?

Let's formulate this as a dynamic programming problem by answering the required questions. The most important items are the stages and the states. See if you can figure out what they might be before looking below. How might we divide this problem into small stages that can be built upon to reach the final answer?

- *Stages*: The problem at time *t* is to find the minimum cost policy from time *t* to time 5. Hence we are subdividing the problem by time in this case. And we will be doing a backwards recursion, so the first problem we solve will be trivial: what is the best thing to do from time 5 until time 5? This is just the same as when we start a shortest route solution by labeling the origin with 0, indicating that the distance from the origin to itself is zero. The next stage will solve time 4 to time 5, the next time 3 to time 5, then time 2 to time 5, then time 1 to time 5, and finally we will have enlarged the problem enough to solve the original problem of what to do from time 0 (i.e. now) until time 5.
- *State at a stage*: The state at a stage is fairly simple in this case: just how many years remain until time 5.
- *Decision at a stage*: Since we are solving the problem anew at each stage, the courier has no bicycle now and must buy a bicycle to get started. The decision then, is how long to keep the bicycle he purchases.
- *Decision update to the state*: Given that he buys a bicycle at the start of the current stage, and makes a decision to keep it for a certain number of years, then the state (how many years until time 5) is updated accordingly. For example if the state is currently 4 (i.e. we are at time 1 and hence it is 4 years until time 5), and the decision is to buy a bicycle now and keep it for 3 years, then the state at the next decision point will be 4-3=1, i.e. we will now be at time 4 with just one year to go until time 5.
- *Recursive value relationship*: First, let's define a few functions and variables:
    - $g(t)$: minimum net cost from time t until time 5, given that a new bicycle is purchased at time t.
    - c_{tx}: net cost of buying a bicycle at time t and operating it until time x, including the initial purchase cost, the maintenance costs, and the salvage value when it is sold.

- The recursive value relationship is then: $g(t) = \min_x \{c_{tx} + g(x)\}$ for t=0,1,2,3,4.

Note how $g(\cdot)$ appears on both sides of the recursive relationship. The optimum from time t to time 5 depends on two things: the value of the current decision c_{tx}, and the value of a previously found optimum $g(x)$.

Note that c_{tx} depends only on how long we keep the bicycle (1-3 years only), so we can work out its possible values in advance to save a little calculation later. Each calculation takes into account the initial purchase price, the maintenance costs, and the salvage value:

- Keep bicycle 1 year: $c_{01} = c_{12} = c_{23} = c_{34} = c_{45}$ = 500 + 30 – 400 = \$130
- Keep bicycle 2 years: $c_{02} = c_{13} = c_{24} = c_{35}$ = 500 + (30 + 40) – 300 = \$270
- Keep bicycle 3 years: $c_{03} = c_{14} = c_{25}$ = 500 + (30 + 40 + 60) – 250 = \$380

Before starting the solution, it is interesting to do some simple analysis to see which length of ownership is the most economical. Is it generally better to keep a bicycle for 1, 2, or for 3 years? The average annual cost of ownership is: for 1 year \$130/1=\$130, for 2 years \$270/2=\$135, for 3 years \$380/3=\$126.67. So it is cheapest to keep each bicycle for 3 years, but if that is not possible, then for just 1 year. Given that we have a 5 year timeframe, we should expect to see some combination of 3 year and 1 year ownerships. Now back to dynamic programming to find the optimum solution…

We also define $g(5) = 0$. No costs are incurred after the timeframe ends. Given that we are using a backwards recursion, we start at time 5 and work backwards. The optimum choice (lowest cost) at each stage is highlighted in bold.

- **$g(5) = 0$**. [This is the trivial first stage.]
- **$g(4) = c_{45} + g(5) = 130 + 0 = 130$**. [Also a trivial stage since there is only one possible decision.]
- $g(3)$ = minimum over: [finally a real decision]
    - **$c_{34} + g(4) = 130 + 130 = 260$**
    - $c_{35} + g(5) = 270 + 0 = 270$
- $g(2)$ = minimum over:
    - $c_{23} + g(3) = 130 + 260 = 390$
    - $c_{24} + g(4) = 270 + 130 = 400$
    - **$c_{25} + g(5) = 380 + 0 = 380$**
- $g(1)$ = minimum over:
    - **$c_{12} + g(2) = 130 + 380 = 510$ [tie]**
    - $c_{13} + g(3) = 270 + 260 = 530$
    - **$c_{14} + g(4) = 380 + 130 = 510$ [tie]**
- $g(0)$ = minimum over:
    - **$c_{01} + g(1) = 130 + 510 = 640$ [tie]**
    - $c_{02} + g(2) = 270 + 380 = 650$
    - **$c_{03} + g(3) = 380 + 260 = 640$ [tie]**

At this point we have hit the stopping conditions: the small problems have been expanded until the entire original problem has been solved. At this point we know that the cheapest policy has a total cost of $640. What we *don't* yet know is how to achieve this result. Here is where we have to unravel the information on the "stack" by tracing back through the solution from time 0 to time 5. We know that we have to buy a bicycle at time 0, but there is tie for the best length of time to keep it: either 1 or 3 years. Let's follow the 1-year ownership first: that leads us to year 1, where we buy another bicycle and again to a tie for the best length of time to keep it, again either 1 year or three years. Following the one year ownership again takes us to year 2, where we buy another bicycle, and this time keep it for 3 years until time 5. So at least one solution that gives a minimum total cost of ownership of $640 is to buy bicycles at times 0, 1, and 2.

Following up the various ties in the solution, we find that there are 3 different solutions that give the same minimum total cost of ownership: buy at 0, 1, 2; buy at 0, 1 and 4; and buy at 0, 3, and 4. This is not surprising given that our initial cursory analysis showed that keeping a bicycle for 3 years is most economical, followed by keeping bicycles for 1 year. The 3 equivalent solutions are made of all the possible combinations of 3-year and 1-year ownership patterns within a 5-year timeframe.

This problem doesn't seem at all like a shortest route problem does it? However it is surprisingly easy to transform an equipment replacement problem into a shortest route problem. Just construct a network in which the nodes represent points in time (time 0, time 1, etc.) and the connecting arcs are labeled with the cost of keeping a bicycle for the appropriate amount of time. The equivalent network is shown in Figure 15.1: just find the shortest route from time 0 to time 5! In this very special case, linear programming could actually be applied to this network formulation. But not all dynamic programming problems can be turned into linear programs so easily.

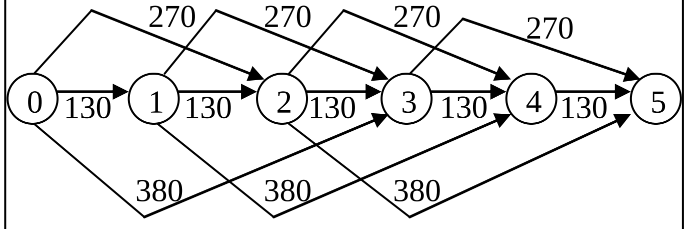


Figure 15.1: Converting the equipment replacement problem to a shortest route problem.

A Knapsack Problem

Knapsack problems typically involve an upper limit on some sort of capacity (e.g. the weight capacity that can be carried in a knapsack), and a set of discrete items that can be chosen, each of which has a value, but also a cost in terms of the capacity (e.g. it has some weight). The goal is to find the selection of items that has the greatest total value while still respecting the limit on the capacity. The classic textbook example is a hiker filling a knapsack with items to take on a hike, each with a certain value and a given weight, but the formulation applies to many other applications such as selecting discrete cargo items for a ship or airplane, or even which weapons to load into a nuclear submarine for an extended cruise.

Let's start with our hiker and her knapsack. She is trying to determine how to choose among several items that she can pack, and how many of each item to take. She can carry up to 10 kg in her knapsack. To keep it simple, let's consider just 3 possible items, whose weights and values are summarized in the table below.

item	**weight (kg)**	**value**
1. food pack	3	7
2. large bottle of water	4	8
3. tent	6	11

She is going to a scout meeting with her troop, so it is possible that she can take more than one of each item on behalf of the troop. How many of each item should she take to maximize the total value while not exceeding her 10 kg carrying capacity?

Before even starting, let's try to guess the solution for this tiny problem, based on the value/weight ratios for the items. These are 7/3=2.33 for food packs, 8/4=2 for bottles of water, and 11/6=1.83 for tents. Hence we would prefer to take as many food backs as possible since they have the greatest value/weight ratio. It's a pretty good bet that the final solution will include some food packs, but it's hard to determine exactly what the final solution will be. We'll have to set up and solve the dynamic program to find out.

But first, note that this problem can be restated as:

Maximize $7x_1 + 8x_2 + 11_{x3}$
subject to $3x_1 + 4x_2 + 6x_3 \leq 10$
where $x_1, x_2, x_3 \geq 0$ and integer.

Do you know a way to solve this problem without using dynamic programming? It can be solved as an integer program via branch and bound in this case. As we will see in the next example though, there are many dynamic programming problems that cannot be turned into more familiar forms such as linear programs (as for the equipment replacement problem) or integer programs solved by branch and bound.

Let's formulate this knapsack problem for a dynamic programming solution.

- *Stages*: The natural breakdown in this case is by item. And since we are practicing backwards recursions, we will work in the order tents, bottles of water, food packs. We will first consider just tents, then bottles of water and tents, then food packs, bottles of water and tents.
- *State at a stage*: The state at a stage is the amount of carrying capacity remaining in the knapsack.
- *Decision at a stage*: The hiker must decide how many copies to take of the item being considered at the current stage.
- *Decision update to the state*: The number of copies of the item taken reduces the carrying capacity for future stages in an obvious way.
- *Recursive value relationship*: First, let's define a few functions and variables:
    - t is the current stage (1, 2, or 3, indicating food packs, bottles of water, or tents)

- d_t is the carrying capacity remaining when you reach stage *t*.
- x_t is the decision, i.e. the number of copies of item *t* that the hiker decides to take
- v_t is the value of one copy of item *t*
- $f_t(d_t)$ is the maximum value obtainable when you enter stage *t* with a remaining carrying capacity of d_t, considering only stages *t*, *t*+1, … 3.
- The recursive value relationship is: $f_t(d_t) = \max_{x_t}\{v_t x_t + f_{t+1}(d_t - w_t x_t)\}$ where $0 \le x_t \le d_t/w_t$ and integer.

Note how $f(\cdot)$ appears on both sides of the recursive relationship. The optimum for stage *t* to stage 3 depends on two things: the value of the current decision at stage *t* (i.e. $v_t x_t$), and the value of the previously found optimum $f_{t+1}(\cdot)$. Note also that $f_{t+1}(\cdot)$ is calculated with a remaining carrying capacity of $d_t - w_t x_t$ meaning that the weight of the items taken at stage *t* has reduced the carrying capacity d_t with which you entered stage *t*.

A difference from the equipment replacement problem is immediately apparent: you can enter a stage in different states. When you are working with stage 3 by itself (the first stage to be calculated), you have no idea what the state might be by the time stages 1 and 2 are considered. Hence for each stage we have to take into account all possible states. This means we need to use a table representation instead of the simple list we used for equipment replacement. Here goes:

We start with stage 3, the tents. You will see that in many dynamic programming problems, the first and last stages considered are simpler than the intermediate stages. That is the case here, where we are considering *only* tents at this time. So the recursive value relationship reduces to $f_3(d_3) = \max_{x_3}\{11x_3\}$ where $0 \le x_3 \le 1$. The hiker can take just 0 or 1 copies of the tent since it weighs 6 kg and her carrying capacity is just 10 kg.

The fully expanded table for stage 3 looks like this, where the best choice in each row is shown in bold:

d_3	$x_3 = 0$	$x_3 = 1$	$f_3(d_3)$
0	**0**	-	0
1	**0**	-	0
2	**0**	-	0
3	**0**	-	0
4	**0**	-	0
5	**0**	-	0
6	0	**11**	11
7	0	**11**	11
8	0	**11**	11
9	0	**11**	11
10	0	**11**	11

The first column indicates the possible remaining carrying capacity when we finally reach stage 3: it could be anything between 0 and 10. The two centre columns show the value of making the decision indicated at the top of the column: note that "-" indicates that the particular decision is not possible. For example, look at the d_3=2 line in the table. If you have only 2 kg of carrying capacity left, then you obviously can't take a 6 kg tent, hence the entry in the x_3 =1 column is "-". The rightmost column just summarizes the best possible thing to do in each row (i.e. the largest value you can attain given your remaining carrying capacity upon entry.

For stage 3, the most important breakpoints are those associated with the weight of a tent (6 kg). If you have less than 6 kg carrying capacity remaining, then you obviously can't take a tent; 6 kg or more left and you can. For this reason, the table for stage 3 can be compressed quite a bit:

d_3	$x_3 = 0$	$x_3 = 1$	$f_3(d_3)$
0-5	0	-	0
6-10	0	11	11

Stage 2 will encompass items 2 *and* 3, and hence will make reference to the table for stage 3. The recursive relationship is now $f_2(d_2) = \max_{x_2}\{8x_2 + f_3(d_2 - 4x_2)\}$ where $0 \leq x_2 \leq 2$ (since, at a weight of 4 kg per bottle of water, we can take at most 2 bottles). The corresponding table is:

d_2	$x_2 = 0$	$x_2 = 1$	$x_2 = 2$	$f_2(d_2)$	$d_3=d_2-4x_2$
0	**0**	-	-	0	0
1	**0**	-	-	0	1
2	**0**	-	-	0	2
3	**0**	-	-	0	3
4	0	**8**	-	8	0
5	0	**8**	-	8	1
6	**11**	8	-	11	6
7	**11**	8	-	11	7
8	11	8	**16**	16	0
9	11	8	**16**	16	1
10	11	**19**	16	19	6

This table may be a little counterintuitive. Let's look at cell for row d_2=10 and column x_2=1, which has a value of 19. How did it get that value? d_2=10 means that we enter stage 2 and 3 (which is what is represented in this table) with 10 kg of carrying capacity remaining, and x =1 means that the hiker has decided to take one copy of item 2 (i.e. a bottle of water). This has a value of 8 and a weight of 4 kg, leaving 6 kg of carrying capacity for stage 3. Now enter the table for Stage 3 on the d_3=6 line, and we see right away that the best thing to do has a value of 11. Hence the total value is 8 (for a bottle of water) plus 11 (for a tent), for a total of 19. The other cells are calculated in the same way, following the recursive relationship.

Stage 1 is again simpler, but for an interesting reason. Now that we have expanded the problem enough to encompass the entire original problem, we know *exactly* what d_1, the carrying capacity as we start the problem, is: it is 10 kg! Hence the table for stage 1 (which encompasses stages 2 and 3 as well), has just a single row for d_1=10, as shown below. The recursive relationship is also slightly simplified because d_1 is replaced by a numeric value: $f_1(10) = \max_{x_1}\{7x_1 + f_2(10 - 3x_1)\}$.

d_1	$x_1 = 0$	$x_1 = 1$	$x_1 = 2$	$x_1 = 3$	f_1	d_2=10-3x_1
10	19	18	**22**	21	22	4

The values in this last table are calculated using only the information about item 1 *and the table for stage 2*. We don't need the table for stage 3 at this point, because stage 2 already incorporates stage 3. As an example, consider the value in the x_1=1 column. The value of 18 is obtained (just as in the recursive relationship) from 1 copy of item 1 (value 7), which has a weight of 3 kg, leaving 7 kg carrying capacity, plus the best thing to do in the stage 2 table when there is 7 kg of carrying capacity left, and that has a value of 11, for a total of 18.

Now at this point we can see that the maximum value combination of items that the hiker can carry has a value of 22, but we still need to unravel the actual solution. To do this we trace back through the tables from stage 1 to stage 3. In stage 1, the best value comes from setting x_1=2, leaving a carrying capacity of 4 kg for the next stage (as summarized in the rightmost column). Entering the table for stage 2 on the line for d_2=4, the best value is for x_2=1, leaving a carrying capacity of 0 as summarized in the rightmost column. Entering the table for stage 3 on the line for d_3=0, the best value is for x_3=0. Hence the solution associated with the maximum value of 22 is to take 2 food packs (item 1), plus one large bottle of water (item 2), and no tents (item 3).

There are a few things to notice about this example. First, the number of rows in the tables can become quite large if there are many states. The number of columns can also become large if there are many possible decisions, so dynamic programming can be crippled by combinatorial explosion for large problems. Second, each cell calculation involves only the item considered at the current stage and the optimum results from just the previous table (since that table encapsulates all the data from all of the previous tables). This is helpful for efficiency. Finally, we need all of the tables to recover the actual solution, and not just its value.

A More General Example: Simultaneous Failure

A data storage company maintains banks of disk drives in three separate locations for security reasons, with probabilities of failure that depend on the locations. For additional security, the company can also provide backup disk drives in each location. There are two backup disk drives available, and the company wishes to determine where to install them to minimize the overall probability that all three locations will fail simultaneously. The estimated probabilities of failure are summarized in the following table.

		Location		
		A	**B**	**C**
backup	**0**	0.20	0.30	0.40
drives	**1**	0.10	0.20	0.25
assigned	**2**	0.05	0.10	0.15

For example, if no backup drives are assigned, then the overall probability of simultaneous failure of all three locations is 0.20×0.30×0.40 = 0.024.

This example is still discrete and deterministic, but it is different in that the recursive relationship that will result is not additive, but multiplicative. The objective is to distribute the 2 available backup disk drives so that the overall probability of failure is minimized. This can be formulated for a dynamic programming solution. Again there is no implied direction in this problem, but we will choose a backwards recursion.

- *Stages*: The natural breakdown in this case is by location. For a backwards recursion we will first consider location C, then locations B and C, then locations A and B and C.
- *State at a stage*: The state at a stage is the number of backup disk drives left to distribute.
- *Decision at a stage*: How many backup disk drives should be assigned to this location? The number could be anything between 0 and 2, depending on the state.
- *Decision update to the state*: The number of backup disk drives assigned to a location reduces the number of backup disk drives available for assignment.
- *Recursive value relationship*: First, let's define a few functions and variables:
    - t is the current stage (A, B, or C, indicating the location)
    - d_t is the number of backup disk drives remaining for assignment when we enter stage t.
    - x_t is the decision, i.e. the number of backup disk drives assigned to this location
    - $p_t(x_t)$ is the probability of failure at stage t given that x_t backup disk drives are assigned to this location
    - $f_t(d_t)$ is the minimum overall probability of failure for stages t+1,…3 obtainable when you enter stage t with d_t backup disk drives available for allocation
    - The recursive value relationship is: $f_t(d_t) = \min_{x_t}\{p_t(x_t) \times f_{t+1}(d_t - x_t)\}$ where $0 \le x_t \le 2$.

As usual, the first table (for location C) is relatively simple. The recursive value relationship is just $f_C(d_C) = \min_{x_C}\{p_C(x_C)\}$ because we not have to worry about stages beyond stage C. The resulting table is:

d_C	x_C=0	x_C=1	x_C=2	$f_C(d_C)$
0	**0.4**	-	-	0.4
1	0.4	**0.25**	-	0.25
2	0.4	0.25	0.15	**0.15**

The second stage (covering locations B and C) has the recursive relationship $f_B(d_B) = \min_{x_B}\{p_B(x_B) \times f_C(d_B - x_B)\}$. As you can see, the minimum function $f(\cdot)$ appears on both sides of the relationship as we expect. The stage B table is:

d_B	x_B=0	x_B=1	x_B=2	$f_B(d_B)$	d_C=d_B−x_B
0	**0.12**	-	-	0.12	0
1	**0.075**	0.080	-	0.075	1
2	0.045	0.050	**0.040**	0.040	0

Let's make sure we understand this table. Consider the row in which d_B=1 and the column in which x_B=0, which has a cell value of 0.075. Where does that value come from? Well, if we have one backup disk drive left to allocate (i.e. d_B=1) and we decide not to allocate it to location B (i.e. x_B=0), then the probability of failure of location B is 0.30, but we still have one backup disk drive left to allocate, so now we enter the stage C table on the d_C=1 line, and the best decision in that line has a value of 0.25, so our recursive relationship for stage B gives a lowest probability of failure under these conditions of 0.30×0.25=0.075 for stage B and C. This is the value in that cell. See if you can see why the other cells have the values they do.

Now we come to the last stage, stage A, which will encompass stages B and C as well and will then encompass the entire original problem, our signal to stop. The recursive relationship for stage A is $f_A(d_A) = \min_{x_A}\{p_A(x_A) \times f_B(d_A - x_A)\}$. The stage A table is again somewhat simpler than the others since we know the number of backup disk drives available when we start the problem: it is exactly 2. We don't have to worry about intermediate numbers as we do for stages B and C.

d_A	x_A=0	x_A=1	x_A=2	$f_A(d_A)$	$d_B=d_A-x_A$
2	0.0080	0.0075	**0.0060**	0.0060	0

Let's review again where the numbers in this table come from. Let's consider the optimum solution in the x_A=2 column, which has a value of 0.0060. That results from the stage A value of assigning 2 backup disk drives to stage A (i.e. 0.05) multiplied by the value of the best action given that you go forward to the next table with 0 backup disk drives left to distribute: the d_B=0 row in the stage B table has a best value of 0.12, so the final result is 0.05×0.12=0.0060. Note that we don't have to look at the stage C table at all to make this calculation. That's because the stage B table already incorporates all the information from the stage C table. This is a general property of dynamic programming: you only need to look at the current stage and one other optimum solution. There is not very much savings from this property in this tiny example, but there can be a huge savings when you have a large problem with many stages.

So at this point we know that the optimum solution has a minimum probability of failure of 0.0060, but we still need to unwind the recursion to find the actual assignment of backup drives to locations that achieves this result (pretty simple in this example). It goes like this:

- From the stage A table we see that of the 2 backup drives available, 2 should be assigned to stage A.
- So we enter the stage B table with 0 backup drives, and the optimum decision in this case is to assign 0 backup drives to stage B.
- So we enter the stage C table with 0 backup drives, and the optimum decision in this case is to assign 0 backup drives to stage C.

This example shows some of the generality of dynamic programming. The recursive relationship is multiplicative in this case, so there is no possibility of some kind of conversion to linear programming. Plus it also shows the value of redundancy in designing failsafe systems!

Final Comments on Dynamic Programming

We have dealt thus far with the simplest possible case for dynamic programming: discrete and deterministic problems in which the direction of solution really didn't matter. Let's discuss each of these cases.

First, there are problems which *must* be done by backwards recursion. For example, consider the employee scheduling problem in which we must choose the number of employees working in each month over the next year. For each month we have forecasts of the number needed to do the work. However skilled workers are hard to hire, so layoffs are to be avoided since they incur a cost and we may not get the workers back. On the other hand, excess employees are costly. The problem is to determine how many employees to hire and lay off every month to minimize overall costs. It is hugely more efficient to solve this problem working backwards in time since we know the numbers of employees needed each month. If we work forwards in time, then we have to keep track of a huge number of possible staffing levels that *might* be needed at future times instead of the small number of known staffing levels.

Continuous models (e.g. water level behind a dam) can be handled by finding meaningful break points in the continuum (e.g. empty, minimum level for water supply, minimum level for hydroelectricity generation, full). Probabilistic versions of dynamic programming are also possible; the goal is then to minimize or maximize the expected value of the recursive value relationship.

Finally it's important to understand that dynamic programming, though tedious to carry out by hand, is actually quite efficient compared to a brute force listing of all possible combinations to find the best one. For example, if you are finding your way through a graph which has 5 possible nodes to go to at each of 6 stages (and is fully connected between each stage), then there are 5^5=3125 possible paths if they are all enumerated, and each of these requires 5 addition operations for 15,625 total operations. However an analysis of a Dijkstra's algorithm dynamic programming solution shows that it takes just 105 operations. Hence the dynamic programming solution requires just 105/15625 ≈ 0.007 of the work!

Chapter 16: Introduction to Nonlinear Programming

A nonlinear program (NLP) is similar to a linear program in that it is composed of an objective function, general constraints, and variable bounds. The difference is that a nonlinear program includes at least one nonlinear function, which could be the objective function, or some or all of the constraints. Many real systems are inherently nonlinear, e.g. modelling the drop in signal power with distance from a transmitting antenna, so it is important that optimization algorithms be able to handle them.

The problem is that nonlinear models are inherently *much* more difficult to optimize. There are twelve main reasons for this, as described below.

Reason 1: It's hard to distinguish a local optimum from a global optimum. Numerical methods for solving nonlinear programs have limited information about the problem, typically only information about the current point (and stored information about past points that have been visited). The usual available information is (i) the point $\boldsymbol{x}$ itself, (ii) the value of the objective function at $\boldsymbol{x}$, (iii) the values of the constraint functions at $\boldsymbol{x}$, (iv) the gradient at $\boldsymbol{x}$ (the derivative in a 1-variable problem), and (v) the Hessian matrix (i.e. the second derivative in a 1-variable problem). This is enough information to recognize when you are at a local maximum or minimum, but there is no way of knowing whether there exists a different and better local maximum, or even how to proceed towards it. This also means that there is no way to easily determine where the global optimum is.

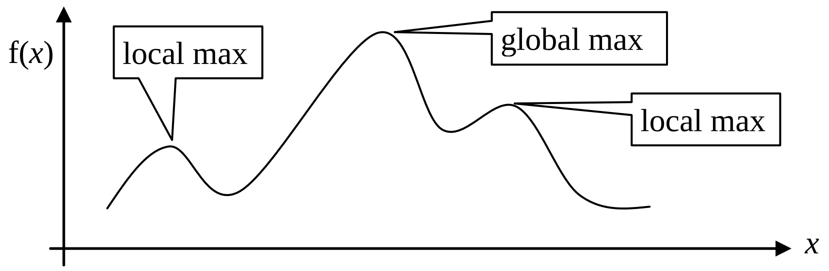


Figure 16.1: Local vs. global maxima for a 1-variable problem.

These ideas are illustrated for a 1-variable unconstrained problem in Fig. 16.1. Formally, a *local optimum* $\boldsymbol{x}^*$ is a feasible point that has a better value than any other feasible point in a small neighbourhood around it, say within some small ball with radius ε. The *global optimum*, on the other hand, is the point with the best value of the objective function anywhere in the feasible region. Note that the global optimum will also be a local optimum.

Reason 2: Optima are not restricted to extreme points. There are a limited number of places to look for the optimum in a linear program: we need only check the extreme points, or corner points, of the feasible region polytope. This isn't so for nonlinear programs: an optimum (local or global) could be anywhere: at an extreme point, along an edge of the feasible region, or in the interior of the feasible region! This is illustrated in Fig. 16.2, which requires a little explanation.

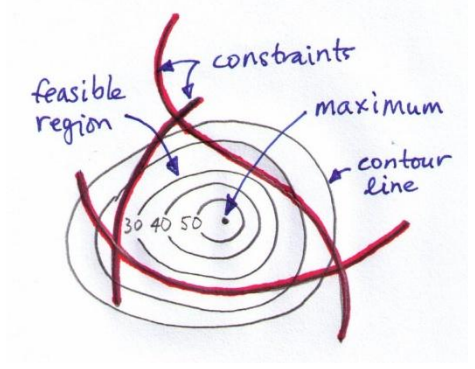


Figure 16.2: Optimum is at an interior point.

We actually need n+1 dimensions to illustrate an n-dimensional function. The added dimension is for the function value itself, i.e. f($\boldsymbol{x}$). You can see this in Fig. 16.1, which requires two dimensions to show a 1-dimension function: the horizontal axis is for the variable x and the vertical axis is for the function value f(x). However we will frequently want to show 2-dimension functions to illustrate ideas in nonlinear programming. So how do we show the 3rd dimension for the function value? One easy way is to use *contour lines* (also known as *level sets*). These are lines that connect a set of points that all have the same value of the function. You may be familiar with this concept from hiking maps which use them to show the topography.

Fig. 16.2 shows the contour lines for an objective function whose maximum point is in the interior of the feasible region, far from the extreme points or the edges. Three of the contour lines are labelled with the value of the objective function for all of the points on that particular line. In this case the optimum is a local maximum, as shown by the fact that the labels on the contour lines increase as we near the "centre" of the concentric ellipses.

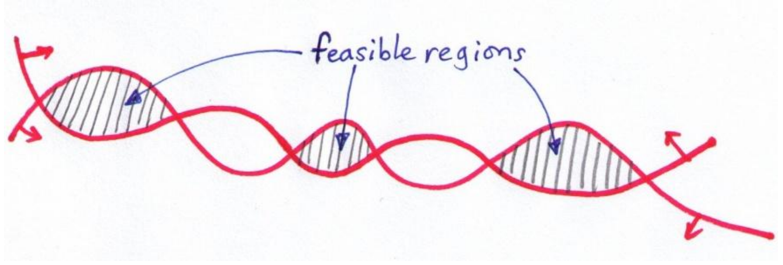


Figure 16.3: Multiple disconnected feasible regions.

Reason 3: There may be multiple disconnected feasible regions. Because of the way in which nonlinear constraints can twist and curve, there may be multiple different feasible regions. So even if you are able to find the optimum within a particular feasible region, how do you know that there isn't some other disconnected (also called *discontiguous*) feasible region that you haven't found and explored? This is illustrated in Fig 16.3 in which the small arrows indicate the feasible sides of the nonlinear inequalities.

Reason 4: Different starting points may lead to different final solutions. This is related to the way in which nonlinear solvers work. One very common algorithm chooses a direction for search (e.g. the direction of steepest descent for minimization), and then finds the best value of the objective function in that direction; the process then repeats until there is no improvement in the value of the objective function. Methods like this will typically descend to the bottom of whatever valley the starting point happens to be in. Because there may be multiple different valleys, starting at some other point may return a different final solution point and objective function value. The difficulty is compounded worse when the problem is constrained: not only are there multiple local optima, but these may be in discontiguous feasible regions! One option is to simply try restarting the solver from many different initial points, but this can be quite time-consuming.

Reason 5: It may be difficult to find a feasible starting point. The phase 1 routine in linear programming will either find a point that satisfies all of the constraints (and hence is feasible), or it will accurately determine that no feasible points exist anywhere. You have no such guarantees in NLP. Most NLP phase 1 procedures try to minimize some measure of the total infeasibility, e.g. the sum of squared constraint violations. For example, if the constraint is $c(\boldsymbol{x}) \leq 12$ and at the current point $c(\boldsymbol{x}) = 15$, then the constraint violation is 3 and the squared violation is 9. If you can find a point at which the sum of the squared constraint violations is zero, then you have reached the global optimum and found a feasible point. However this is in itself a nonlinear programming problem to solve, and also faces the difficulty that solvers are not able to distinguish local and global optima. Solvers get stuck in local optima in which the sum of the constraint violations is not zero. Again, you can try restarting the phase 1 procedure from multiple different starting points, but if you never find a feasible point you are still left unsure as to whether the model is actually infeasible, or whether it is feasible but you simply didn't start your solver in the right place.

Reason 6: It is difficult to satisfy equality constraints (and to keep them satisfied). In linear programming, the phase 1 routine finds a solution that satisfies all of the constraints (including the equalities), and thereafter these are never violated. In NLP, finding a solution that satisfies curving and twisting equations is difficult in and of itself, but even if a solution is found at some point, the equality may again be violated when the algorithm tries to move to another point that has a better value of the objective function.

Reason 7: There is no definite determination of the outcome. In linear programming, the solver (generally) has only a couple of outcomes: (i) the model is feasible and there is a globally optimum solution point, (ii) the model is feasible but unbounded, or (iii) the model is infeasible. These are very definite statements about the status of the outcome (neglecting various error-related outcome possibilities). Things are much less certain for NLP. The solver may report an optimum, but at best it can check certain conditions to guarantee that the point is a local optimum: it is not able to say whether the point is a global optimum. It may also continue improving the value of the objective function for a long time, but will not be able to say whether the model is unbounded. And as discussed in Reason 5, you are never sure whether the model is actually infeasible if no feasible solution is found.

Reason 8: There is a huge body of very complex mathematical theory and numerous solution algorithms. This is not surprising, given the fact that nonlinear functions have a much wider range of behaviours and characteristics than linear functions. But it does make it hard to know how to solve the problem at hand. How do you decide which algorithm to apply, and can you follow the complex steps correctly?

The reasons why NLP is so much harder than LP given above are mainly based on the theory of nonlinear functions. But there are even more reasons why NLP is hard! These are based on more practical considerations.

Reason 9: It is difficult to determine whether the conditions to apply a particular solver are met. Many solution algorithms require that the functions in the model have particular characteristic, which could relate to their algebraic structure (e.g. must be quadratic, or polynomial, etc.), or their shape (e.g. convexity and concavity, i.e. whether they curve up or down everywhere). The problem is that it is often difficult in practice to determine whether these conditions are satisfied or not. How would you know whether a constraint is concave in the region of interest? Or whether the constraints taken together form a convex set (more on this later)? There are very few tools for assessing models prior to selecting the solution algorithm and associated NLP solver. This is a ripe area for research.

Reason 10: Different algorithms and solvers arrive at different solutions (and outcomes) for the same formulation. Almost all NLP algorithms proceed in a step-wise fashion, iteratively improving the initial point until certain stopping conditions are met. But each algorithm will likely have a completely different trajectory of points from the initial point to the final point that is output as the solution. This means that different algorithms may terminate at different local optima, and hence will return different solutions, even for the exact same formulation and initial point. Further, some algorithms may terminate successfully at local optima while others move off in different directions and are unable to even find a feasible point, so the outcomes are completely different. Even worse, different implementations of the *same* algorithm (i.e. different solvers) may also give different results due to different choices of parameters and internal heuristics!

Reason 11: Different but equivalent formulations of the model given to the same solver may produce different solutions and outcomes. Here are two mathematically equivalent formulations of a constraint function: (i) $x^2 + y^2 \leq 25$, or (ii) $a = x^2$, $b= y^2$, $a + b \leq 25$. Solvers may treat these formulations differently however, perhaps finding the second version easier to handle by first dealing with a simple linear inequality in a and b, and then later solving for the values of x and y. Thus the same solver may have a different trajectory of points for different formulations of the same model, and hence arrive at different final solutions and outcomes. This is a marked contrast to LP, where equivalent but different formulations of the same model will normally have the same outcome (though the sequence of intermediate corner point solutions will be different).

Reason 12: Using the available nonlinear solvers can be complex. Most solvers have a large number of user-settable parameters which control details of how the solver will operate. It's hard to know in advance how these parameter settings will affect the solution. What type of line search should be used? What tolerances should be used? What is the largest acceptable basis inverse condition number that should be accepted? Etc. There may be 50 such parameters. Fortunately these are normally set at good default values, but adjusting them takes expert knowledge.

But there is some good news! Some things that were difficult in the past can now be done relatively easily:

- **Derivatives.** It used to be difficult to get function derivatives, which are often required by solvers. There were two approaches: numerical approximation by finite differences, or requiring the modeller to provide code for the derivative in addition to the code for the original function. This latter approach was often problematic when the derivative code and the function code did not match up (due to typos etc.). In that case the solver would not operate correctly. Fortunately the more recent technique of automatic differentiation is largely available. This analyzes the function expression to determine the correct expression for the derivative.

- **Input formats.** At one point, each solver had its own particular input formats to describe the model, or else required that the user attach a subroutine (written in a computer language like Fortran or C) that described the model. Thus if you found that one solver was not effective on the problem at hand it was a tedious and error-prone job to recode the model into a different format so you could try a different solver. Nowadays, however, most solvers can be linked to some standard optimization modelling languages, such as AMPL, GAMS, etc. The means that the model can be written just once, and then multiple different solvers can be tried.

And there is other good news as well:

Relatively few commercially implemented algorithms. Though the academic literature describes a vast range of solution methods, the number of algorithms that have actually been implemented in readily available commercial (or even free open-source) solvers is much smaller. Hence for practical purposes, there are relatively few algorithms that you need to understand in depth. Typical commercially implemented algorithms include the generalized reduced gradient method, quadratic programming and successive quadratic approximation, and barrier methods. As time permits we will spend some time looking at these algorithms (after laying out some groundwork).

Modelling languages and systems. In addition to eliminating the input format problem as described above, modelling languages have evolved into larger systems that include additional features like the ability to perform a pre-solve step that simplifies the model and tightens the bounds, perhaps identifying errors, automatic differentiation, reporting and programming features, etc. The language systems also allow the modeller to link to databases so that very large models can be constructed in a convenient way.

Model analysis tools. There are now a few tools that analyze nonlinear models, and that can be applied either before solution to help decide what type of algorithm might be best, or after solution, e.g. to determine the reasons for an unexpected outcome. MProbe (http://www.sce.carleton.ca/faculty/chinneck/mprobe.html) is one such tool: it empirically tests functions to estimate their shapes: Everywhere convex? Everywhere concave? Both? Almost linear? Etc. It also looks at the effectiveness of the constraints in terms of eliminating feasible solutions and has a number of other useful tools and analyses.

Solving nonlinear programs is difficult, but not impossible: the available solvers are improving rapidly for one thing. But there are a few things that you can do to increase the probability of a successful solution:

- **Always use a modelling language.** This will allow you to easily switch between solvers which may use different algorithms. This will give you a better chance of finding a solution if your first choice solver fails. It also gives you access to the other tools in the associated modelling system such as the nonlinear presolver, automatic differentiation, etc.

- **Look for a simpler formulation.** Sometimes a nonlinear function can even be reduced to a linear form and solved by an LP by simple reformulation. Look at Reason 11 above, for example.

- **Know the characteristics of your model before choosing a solution algorithm.** For example, is it composed entirely of quadratic functions? If so, then there are specialized and highly effective algorithms for solution. Can it be well-approximated by piece-wise linear functions? Then it can be reduced to a linear program. Perhaps you can apply software like MProbe to determine the shapes of the constraint and objective functions; this information will help you decide which solution algorithm to apply.

- **Provide a good starting point.** The closer your initial starting point is to the eventual optimum point, the better chance that the solver will successfully find it. But how do you know where the optimum might be? You may have solved a very similar model before, so you can use the previous solution as your new starting point. This is actually pretty common, e.g. where a company solves the same problem day after day, with just minor changes to the inputs (e.g. changes to prices, or to availability of material). You may be able to guesstimate a good initial point based on a heuristic analysis using a simpler model that is easy to solve (even an LP). Or you may just do some simple analysis. There is recent work on effective methods for improving any selected starting point before passing it to the solver (search on "Constraint consensus": yes I wrote those papers). As the old joke goes, "the best way to solve an NLP is to start at the optimum". At this point you may not see the humour in that statement, but at the end of this course you will find it hilarious!

- **Put reasonable bounds on all variables.** This narrows the search space so the solver does not spend a lot of time exploring useless regions of the variable space.

- **Make the best use of the solver parameter settings.** Many of the default settings for the parameters in a given solver have been carefully tuned to give the best performance over a broad range of model types, so it's usually best to leave them alone. But sometimes there are parameters that can greatly improve the solver performance for your particular model, especially if it has special characteristics, however it takes some expertise to make effective parameter adjustments.

Local Optimization of Nonlinear Programs

Most readily available software for nonlinear programming deals only with local optimization of nonlinear programs. In fact, when most people say "nonlinear programming" they usually mean local optimization; they will normally say "global optimization" when they have that meaning in mind. In any case, most global optimization techniques use local optimization routines in their operation, so we will start by looking at local optimization. And as usual we'll start with the simplest possible cases and work our way upward from there.

One-Dimensional Unconstrained Local Nonlinear Optimization

The simplest possible case of nonlinear optimization has only a nonlinear objective function and no constraints (hence "unconstrained") and involves just one variable. For example, you need to find the maximum value over the one-dimensional unconstrained function sketched in Fig. 16.1. This looks easy in Fig. 16.1, but recall that a computer method does not have a nice sketch to look at: it only has individual points and the associated objective function values that it calculates. What it sees after the first point it tests looks something like Fig. 16.4. The question is this: *where* should we evaluate the objective function? An algorithm does this in an organized manner that leads to a local optimum. It uses information revealed by the various test points to determine where to test next.

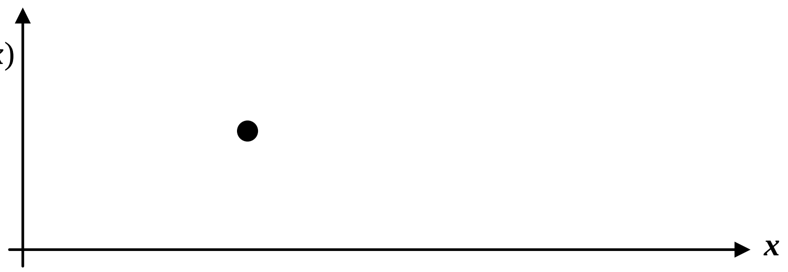


Figure 16.4: The function as seen by a computer after a single test point.

For a differentiable function, the information available at test point x is (i) the value of the function $f(x)$, (ii) the derivative of the function $\frac{df(x)}{dx}$, and (iii) the second derivative of the function $\frac{d^2f(x)}{dx^2}$.

From elementary calculus, recall that a local maximum point x^* of an unconstrained function has two properties: (i) the derivative is zero, i.e. $\frac{df(x^*)}{dx} = 0$, meaning that we are at a level spot on the function, and (ii) the second derivative is negative, i.e. $\frac{d^2f(x^*)}{dx^2} < 0$, meaning that the slope tends downwards in both directions from this spot. For minimization, the second condition changes to $\frac{d^2f(x^*)}{dx^2} > 0$.

So the fundamental question is how to search for a local optimum point x^* that meets these two conditions in an efficient manner. In computer implementations this is usually done via a *numerical* search method (it's numerical because it relies on generating and working with numbers, e.g. function values, as opposed to an *algebraic* method that works with mathematical symbols). There are numerous numerical methods for one-dimensional local optimization: bisection search, golden section search, Fibonacci search, etc. We'll take a quick look at bisection search as an example of the genre.

Bisection Search for a Local Maximum of a One-Dimension Nonlinear Objective Function

The bisection method applies when there is a single continuous variable (dimension) in a differentiable nonlinear function (it must be differentiable because we need to use the derivative). It starts with some initial range of the single variable that is known to encompass a local maximum, and then iteratively halves this range, always guaranteeing that the remaining range includes a local maximum. How does it know that the range encompasses a local maximum? It uses the derivatives at the endpoints of the range: there is at least one local maximum in the range (and there may be several) if the left endpoint has a positive derivative (the tangent line is "going uphill"), and the right endpoint has a negative derivative (the tangent line is "going downhill"): see Fig. 16.5. This is the condition that we make sure to maintain at all times.

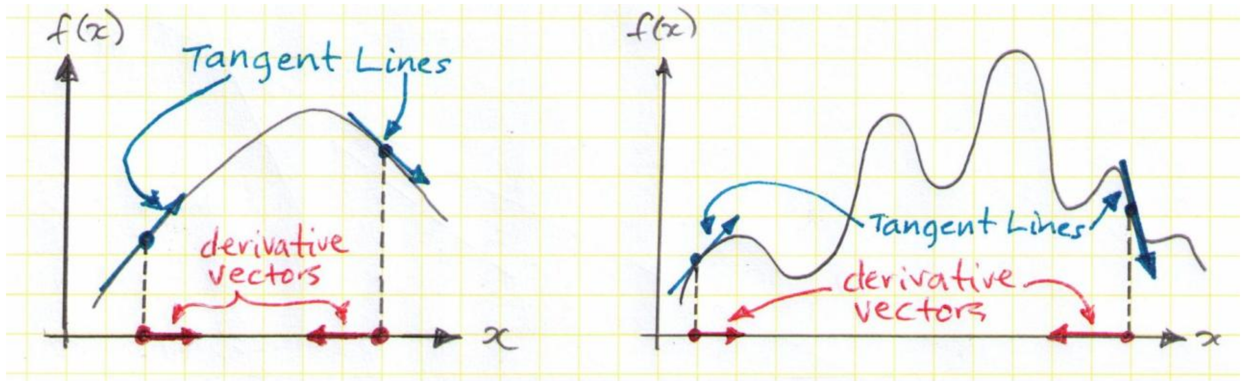


Figure 16.5: The endpoints of the range bracket at least one local maximum

But how do we find the initial range? There are many ways to do an initial coarse search to find two endpoints that satisfy the conditions. For example you could start with a random point: if its derivative is positive, then you have your initial left endpoint, and if its derivative is negative then you have your initial right endpoint (and in the unlikely case that its derivative is zero then you are at a *critical point* which could be a local maximum, a local minimum or a *saddle point*, more about this later). Once you have one endpoint then you could take a step in the specified direction (if you have a left endpoint then the step will increase the variable value; if you have a right endpoint then the step will decrease the variable value). If you don't find a suitable endpoint of the opposite type after one step, then perhaps double the stepsize and move again. Eventually you should end up with a pair of endpoints that satisfy the derivative conditions, so you know there is a local maximum point between them. Now the bisection algorithm can begin. The steps are summarized in Algorithm 16.1. Note that the output solution x^* is guaranteed to be within ε of the true optimum.

Note that the selection of an error tolerance, typically represented by the epsilon symbol ε, is a feature of most numerical algorithms, including both linear and nonlinear programming. It's a side effect of using binary representations of real numbers: they can't be represented exactly with a limited number of bits, so instead of checking whether two numbers are exactly equal, we have to check whether they are approximately equal, i.e. within ε of each other.

I've omitted some details in Algorithm 16.1 such as the case when $f(x')$ has a derivative that is equal to zero (or within ε of zero). We will also return later to the question of how to calculate the derivatives that are needed in the bisection algorithm. For now it's enough to know that they can be estimated by numerical techniques – or calculated exactly using more modern methods.

The main lesson is that there are techniques for finding an unconstrained local optimum in one dimension. This is also known as a *line search* since we are searching along a single line.

Start-up:

- Choose an acceptable error tolerance ε
- Find an initial left endpoint x_L such that $\frac{df}{dx}(x_L) > 0$
- Find an initial right endpoint x_R such that $\frac{df}{dx}(x_R) < 0$ and $x_R > x_L$

Iterate:

- Bisect the range: $x' = (x_L + x_R)/2$
- If $\frac{df}{dx}(x') > 0$ then set $x_L = x'$, else if $\frac{df}{dx}(x') < 0$ then set $x_R = x'$.
- If $(x_R - x_L) \leq 2\varepsilon$ then exit with solution $x^* = (x_L + x_R)/2$, else repeat iteration.

Algorithm 16.1: Bisection Search

Multi-dimension Unconstrained Local Nonlinear Optimization

As you will see throughout these notes, many methods for nonlinear optimization build upon simpler methods. We will look at a very common method for multi-dimension unconstrained local nonlinear optimization that builds directly upon the line search method for the one-dimension case. This is known as the *method of steepest ascent* for maximization and the *method of steepest descent* for minimization, and is also sometimes called *hill-climbing*.

We will look at the case of maximizing $f(\boldsymbol{x})$ over all possible values of $\boldsymbol{x}$, where $\boldsymbol{x}$ is a vector of multiple variables (and hence shown in lowercase boldface). As for the one-dimension case, there may be multiple optima, but we will concentrate on finding a single local maximum. The main idea for maximization is simple: at the current point, find the steepest uphill direction, then conduct a line search in that direction to find a local maximum; update the point to the local maximum; repeat this process until the stopping conditions are met. This reduces the multi-dimension problem to a series of single-dimension problems that we already know how to solve using a technique like bisection search.

The first question is: how do you find the steepest uphill direction? In a one-variable problem this is simple: it is the direction given by the sign of the first derivative of the function. In Fig. 16.5 the tangent arrows on the functions show an upward slope at x_L and a downward slope at x_R. But note that the direction specified by the derivative occurs in the variable space: this is a one-dimensional problem, so we can only increase or decrease the single variable. The search direction is either right (increase the variable) or left (decrease the variable). Recall that sketches of the function value include an extra dimension which shows the value of the function. In Fig. 16.5 the search direction is simply along the x axis.

For multi-dimension problems, the direction of steepest ascent is given by the *gradient vector* of the function, which is the multi-dimension analog of the single-dimension derivative. The gradient vector is defined as follows:

$$\nabla f(\boldsymbol{x}) = \left[\frac{\partial f(\boldsymbol{x})}{\partial x_1}, \frac{\partial f(\boldsymbol{x})}{\partial x_2}, \frac{\partial f(\boldsymbol{x})}{\partial x_3}, \dots \frac{\partial f(\boldsymbol{x})}{\partial x_n}\right]$$

i.e. the vector of first partial derivatives at the point ***x*** where there are *n* variables. The length of the gradient vector is related to the rate of increase of the function in the gradient direction.
The second question is: how do we identify a local maximum point? In the single dimension case this is done by finding a point at which the first derivative is zero (which identifies a critical point that is either a local maximum, a local minimum, or a saddle point), and at which the second derivative is negative (indicating that the function tends downwards in both the forward and backward directions. These two conditions guarantee a local maximum. For a local minimum, the second condition reverses: the second derivative must be positive.

With the gradient as the multi-dimension analog of the derivative, the first condition in the multi-dimension case is simply that $\nabla f(\boldsymbol{x}) = \mathbf{0}$, i.e. that all partial derivatives are zero. This again identifies a critical point. But the multi-dimensional analog of the second derivative is more complicated. It is called the *Hessian* matrix, and is defined as follows. Where:

$$h_{ij} = \frac{\partial^2 f(\boldsymbol{x})}{\partial_i \partial_j}$$

Then the Hessian matrix is defined as the square matrix of second partial derivatives:

$$\boldsymbol{H} = \begin{bmatrix} h_{11}\, h_{12} \dots h_{1n} \\ h_{21}\, h_{22} \dots h_{2n} \\ \vdots \quad \vdots \quad\quad \vdots \\ h_{n1}\, h_{n2} \cdots h_{nn} \end{bmatrix}$$

Using the Hessian is also complicated. First you construct subsets of ***H*** where ***H***$_i$ indicates the subset created by taking the first *i* rows and columns of ***H***. Then you calculate the determinant of each of the *n* subsets created this way (where the enclosing vertical lines indicate the determinant operator):

$$H_1 = |h_{11}|,\ H_2 = \begin{vmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{vmatrix}, \cdots$$

Note that the H_i are not matrices, just scalar numbers given by the determinants. Now you determine the category of the Hessian matrix from the sign patterns of the determinants of the H_i, as follows:

- ***H*** is *positive definite* at ***x*** if $H_i > 0$ for i=1…n
- ***H*** is *negative definite* at ***x*** if $H_1 < 0$ and the remaining H_i alternate in sign.
- ***H*** can be *positive semi-definite* or *negative semi-definite* if some of the values that are supposed to be nonzero turn out to be zero.

So if the gradient vector is entirely zero at some point ***x*** (i.e. $\nabla f(\boldsymbol{x}) = \mathbf{0}$), then the category of the Hessian matrix at ***x*** determines whether ***x*** is a maximum, minimum or saddle point:

- ***x*** is a local maximum if $\nabla f(\boldsymbol{x}) = \mathbf{0}$ and H is negative definite.
- ***x*** is a local minimum if $\nabla f(\boldsymbol{x}) = \mathbf{0}$ and H is positive definite.

- $\boldsymbol{x}$ is a saddle point if $\nabla f(\boldsymbol{x}) = \boldsymbol{0}$ and H is neither positive semi-definite nor negative semi-definite.

While we are on the topic, what is a saddle point anyway? In two dimensions it's a critical point ($\nabla f(\boldsymbol{x}) = \boldsymbol{0}$) which is simultaneously a local maximum in some hyperplane and a local minimum in another; this makes it look like a saddle. See Fig. 16.6: the red point is a local minimum in the left-to-right direction and simultaneously a local maximum in the front-to-back direction.

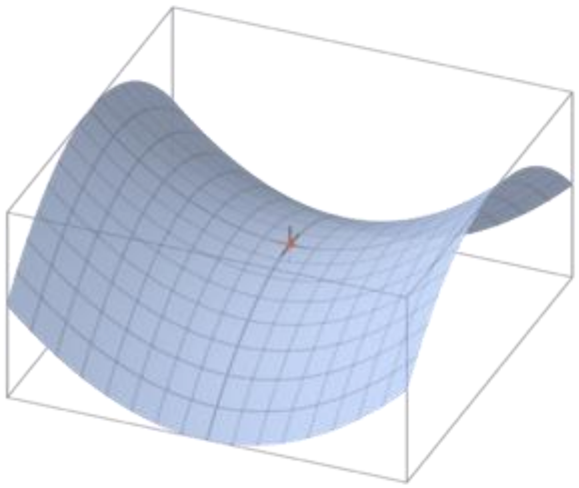


Figure 16.6: Saddle point for the function $f(x)=x^2-y^2$ (image from Wikipedia).

OK, now we can identify a local optimum point if we happen to be at one, but how do we go about actually finding a local optimum point? As described earlier the steepest ascent concept is simple: at the current point, find the steepest uphill direction, then conduct a line search in that direction to find a local maximum; reset your point to the local maximum; repeat this process until the stopping conditions are met. But there are a few important wrinkles in the details. Algorithm 16.2 summarizes the method.

Strictly speaking, Algorithm 16.2 only finds a critical point since it stops in Step 2 when the elements of the gradient vector are all small enough. However the Hessian of the output point can be checked to see whether it really is a local maximum. It usually is because the method is always searching in an uphill direction.

Start-up:

- Choose an acceptable error tolerance ε.
- Choose a starting point $\boldsymbol{x}'$.

Iterate:

1. Calculate $\nabla f(\boldsymbol{x}')$.
2. If $|\nabla f(\boldsymbol{x}')| \le \boldsymbol{\varepsilon}$ then exit with $\boldsymbol{x}'$ as the solution.
3. Set $\boldsymbol{x}'' = \boldsymbol{x}' + t\nabla f(\boldsymbol{x}')$, where $t \ge 0$.
4. Use a line search to find t^* such that $f(\boldsymbol{x}'')$ is a local maximum *Note:* $\boldsymbol{x}'$ and $\nabla f(\boldsymbol{x}')$ are fixed. Only t varies during the line search. Use t^* to calculate $\boldsymbol{x}''$ using the expression in Step 3.
5. Set $\boldsymbol{x}' = \boldsymbol{x}''$.
6. Go to Step 1.

Algorithm 16.2: Steepest ascent algorithm to find an unconstrained local maximum point.

Steps 3 and 4 set up the one-dimensional line search in the gradient direction. $\boldsymbol{x}'$ and $\nabla f(\boldsymbol{x}')$ are fixed during the line search and only t is varied. As Step 3 shows, varying t just gives different values of $\boldsymbol{x}''$ in the gradient direction. t expresses how far to move in the gradient direction and

$\boldsymbol{x}''$ is the actual point in n-space at that distance along the gradient vector from $\boldsymbol{x}'$. The output point corresponds to a local maximum in the gradient direction.

Consider this example: maximize $f(\boldsymbol{x}) = -x_1^2 - 4x_2^2 + 8x_1 + 16x_2$. The gradient for this function at some point $\boldsymbol{x}'$ is given by $\nabla f(\boldsymbol{x}')=[-2x_1'+8, -8x_2'+16]$. So points along the gradient direction from $\boldsymbol{x}'$ are given by $\boldsymbol{x}'' = \boldsymbol{x}'+t\nabla f(\boldsymbol{x}')$, which amounts to $\boldsymbol{x}'' = (x_1', x_2') + t[-2x_1'+8, -8x_2'+16]$ in which x_1' and x_2' are fixed and t is the only variable. The setup and first iteration in solving this problem might look like this:

Start-up: choose ε=0.000001 and $\boldsymbol{x}'$=(0,0). Two notes here. First, tolerances in the range of 10^{-6} are fairly standard in real solvers. Second, while the origin is often used as the initial point, it can be a poor choice, e.g. due to divide by zero errors. There are much better heuristics for choosing the initial point: see the note on page 6 about providing a good starting point for the solver.

The objective function value at (0,0) is 0. The first iteration proceeds as follows:

1. $\nabla f(0,0) = [-2(0)+8, -8(0)+16] = [8,16]$.

2. $[|8|,|16|] > [10^{-6}, 10^{-6}]$ so continue.

3. $\boldsymbol{x}'' = (0,0) + t[8,16] = (8t,16t)$.

4. The one-dimensional line search operates by varying t in the function $f(\boldsymbol{x}'')$, or equivalently, on $f(t) = -(8t)^2 - 4(16t)^2 + 8(8t) + 16(16t)$, obtained by substituting the elements of $\boldsymbol{x}''$ in Step 3 into the original function to create a one-dimensional function of t. Let's assume that we use a bisection search and that it returns t^*=0.147. Using the expression in Step 3, this means that $\boldsymbol{x}''$ = (0,0) + 0.147[8,16] = (1.18, 2.35). The objective function has improved: f(1.18, 2.35) = 23.529.

5. Now we would go to the next iteration, starting at the point (1.18, 2.35). The gradient at this point is [5.6471, -2.8235] so we perform a one-dimensional search along $\boldsymbol{x}''$ = (1.18, 2.35) + t[5.6471, -2.8235], yielding an optimum solution of t^*=0.312. Hence the new point is $\boldsymbol{x}''$ = (1.18, 2.35) + 0.312[5.6471, -2.8235] = (2.942, 1.469) where the objective function has again improved: f(2.942, 1.469) = 29.751.

The solution gradually moves towards the optimum point at f(4,2) = 32. It actually stops a little short of (4,2) because of the solution tolerances, but the final point is very close. As is typical of steepest ascent/descent methods, the solution trajectory zig-zags towards the optimum point, taking smaller and smaller steps, and making smaller and smaller improvements in the objective function value as it proceeds. A typical steepest descent solution trajectory for a different problem is shown in Fig. 16.7.

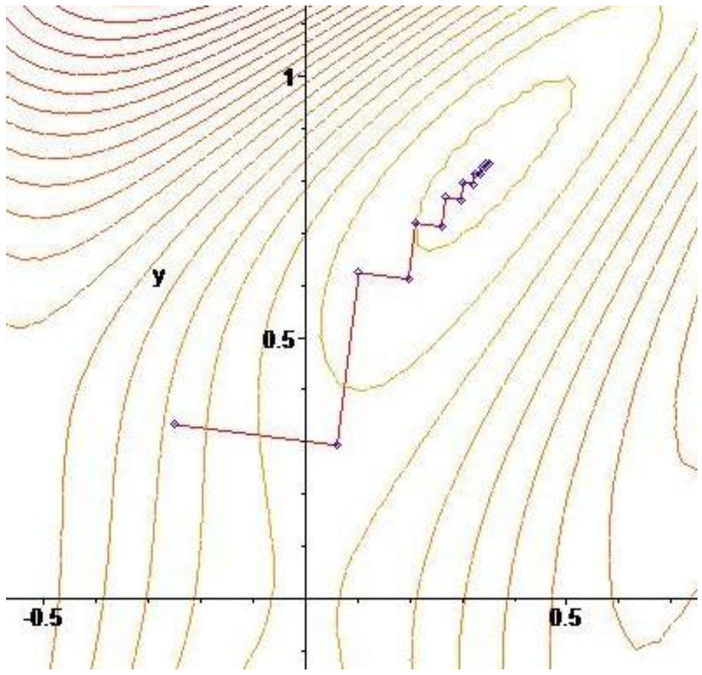


Figure 16.7: Typical zig-zagging steepest ascent/descent solution trajectory (image from Wikipedia).

Obtaining Derivative and Gradient Information

There are several ways to obtain the derivative and gradient information that the steepest/ascent (and many other solution methods) rely on. As described above, years ago the only way to get any information about either the objective function or its derivatives was to supply a subroutine which provided code for these functions. So the user had to analytically derive the derivative function and code it correctly in a programming language such as Fortran or C. When mistakes were made in this process, the derivative function was not correctly matched with the objective function and the entire solution process would go awry.

To avoid this problem, users could also usually ask for the derivatives and gradients to be estimated by *finite difference* methods. Methods such as this evaluate the objective function at the point in question and again at a different nearby point. The derivative is then estimated by dividing the differences in the function values by the distance between the points. In the single variable case there are three variations: *forward differencing* or using a point that is slightly larger than the current point, *backward differencing* or using a point that is slightly smaller than the current point, and *central differencing* or using one point slightly smaller than the current point and one point slightly larger than the current point. Solvers could use the estimated derivatives and gradients directly, or could use them as a check on the values returned by the derivative subroutines, raising an error if the two results differed by a significant amount.

Nowadays there are better alternatives. Most of the well-known modelling languages provide automatic differentiation. If the model is expressed in one of these languages, then the associated system can provide the derivative or gradient for a function at any point. It does this by parsing the expression of the function so that the derivatives and gradient expressions are discovered and hence calculated accurately. Alternatively, symbolic mathematics systems such as Mathematica or Maple can provide the analytic expression for the derivative, which then can be coded into the program.

Chapter 17: Pattern Search for Unconstrained NLP

Sometimes it is inconvenient or not possible to know the first or second derivatives of the objective function in an unconstrained nonlinear programming problem. This can happen, for example, when the function value at some input point $\boldsymbol{x}$ is actually calculated by a simulation of a system, e.g. the simulation returns a measure of the flight characteristics of a new aircraft that is being designed.

In this case we can use heuristic *pattern search* methods which need only the ability to return the value of $f(\boldsymbol{x})$ for some input point $\boldsymbol{x}$. For this reason they are also known as *derivative-free*, *direct search*, or *black box* optimization methods. Of course pattern search methods can also be applied when the objective function is differentiable, but in that case we are ignoring the useful information in the derivatives and second derivatives. So pattern search methods are typically applied only when the derivatives are not available.

Quite a number of pattern search methods have been developed over the years. As an example of the genre, we will look at one of the original methods, known as *Hooke and Jeeves* after the original authors.

Hooke and Jeeves Pattern Search

This method was originally published in 1961. It involves two types of *moves*:

- *Exploratory search*. This is a very local search that looks for an improving direction in which to move. In some senses it is a crude search for the gradient direction.

- *Pattern move*. This is a larger search in the improving direction. Larger and larger moves are made as long as the improvement continues.

We'll look at each of these two types of moves separately before assembling them into the complete algorithm.

Exploratory Search

The main idea is to find *some* improving direction (not necessarily *the best* improving direction, which would be the gradient or anti-gradient direction). This is done by perturbing the current point by small amounts in each of the variable directions and observing whether the objective function value improves or worsens.

First we define the sizes of the perturbation steps that we will take in each dimension by setting up the perturbation vector $\boldsymbol{P}_0 = (\Delta x_1, \Delta x_2, \Delta x_3, \ldots \Delta x_n)$. Note that the perturbation step sizes do not all have to be equal, but in general they are all relatively small. Given some current point $\boldsymbol{x}^{(0)}$ and its associated objective function value $f(\boldsymbol{x}^{(0)})$ we can now perform an exploratory search around it as follows:

1. $\boldsymbol{x}^{(1)} \leftarrow \boldsymbol{x}^{(0)}$, i.e. copy $\boldsymbol{x}^{(0)}$ into $\boldsymbol{x}^{(1)}$. Initialize $f_{\text{best}} \leftarrow f(\boldsymbol{x}^{(0)})$.
2. For each variable x_j in turn:
    a. $\boldsymbol{x}^{(1)} \leftarrow \left(x_1^{(1)}, x_2^{(1)}, x_3^{(1)}, \dots, x_j^{(1)} + \Delta x_j, \dots, x_n^{(1)}\right)$, i.e. bump the jth variable *up* by its perturbation value.
    b. If $f(\boldsymbol{x}^{(1)})$ improves over current value of f_{best}: retain the perturbation, update $f_{\text{best}} \leftarrow f(\boldsymbol{x}^{(1)})$ and go to Step 2.a for the next variable.
    c. [$f(\boldsymbol{x}^{(1)})$ worsens or stays the same for the upwards perturbation]: $\boldsymbol{x}^{(1)} \leftarrow \left(x_1^{(1)}, x_2^{(1)}, x_3^{(1)}, \dots, x_j^{(1)} - \Delta x_j, \dots, x_n^{(1)}\right)$, , i.e. bump the variable *down* by its perturbation value.
    d. If $f\left(\boldsymbol{x}^{(1)}\right)$ improves over current value of f_{best}: retain the perturbation and update $f_{\text{best}} \leftarrow f(\boldsymbol{x}^{(1)})$. Otherwise discard the perturbation: $x_j^{(1)} \leftarrow x_j^{(0)}$.
    e. Go to Step 2.a for the next variable.
3. The improving direction is given by the vector $\boldsymbol{x}^{(1)} - \boldsymbol{x}^{(0)}$. If $\boldsymbol{x}^{(1)} = \boldsymbol{x}^{(0)}$ then the exploratory search has failed.

There are a few things to notice about the exploratory search algorithm:

- If the upward perturbation for a variable is successful, then the downward perturbation for that variable is not even attempted.
- The downward perturbation for a variable is tried only if the upward perturbation for that variable fails.
- It's possible that both the upward and the downward perturbations fail for a particular variable, in which case its value is not changed.
- The method does not try all possible combinations of upward and downward perturbations of the variables (there would be 2^n such combinations: too many to try). It is simply one pass through the list of variables. In the worst case where the upward perturbation fails for every variable, then it will try $2n$ combinations of perturbations, but it normally tries fewer than that. In the best case where every upward perturbation succeeds, it will try just n perturbations.

If *any* of the attempted perturbations succeeds in improving the value of the objective function, then the exploratory search has succeeded and the improving direction is given by the vector between the initial point $\boldsymbol{x}^{(0)}$ and the final point $\boldsymbol{x}^{(1)}$ output by the exploratory search. Now we can use this improving direction in the pattern move. If all of the perturbations fail to find an improved value of the objective function, then the exploratory search has failed: we'll see later what to do in that case.

Figure 17.1 gives some examples of exploratory searches.

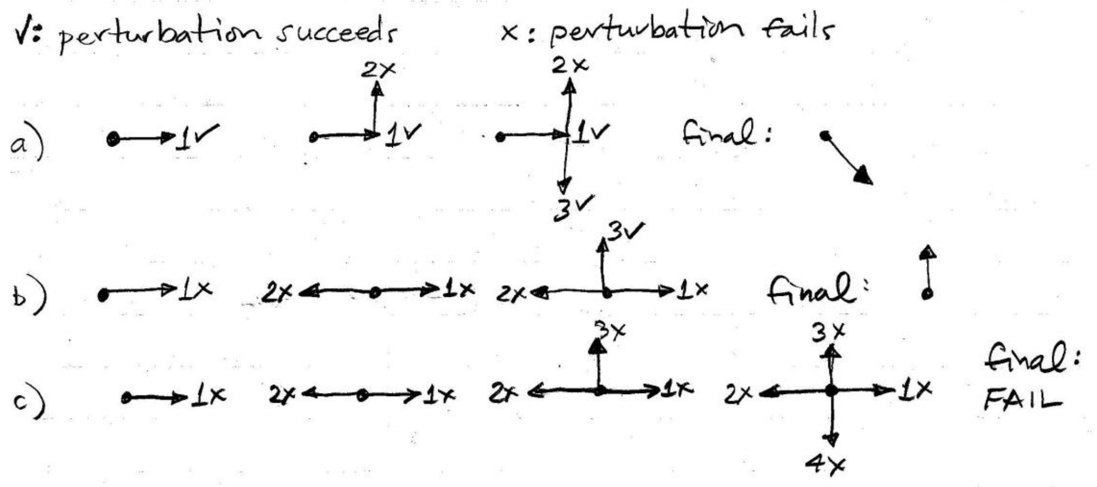


Figure 17.1: Examples of exploratory searches.

Pattern Move

All that the pattern move requires is two points: the current point $\boldsymbol{x}^{(0)}$ and some other point $\boldsymbol{x}^{(1)}$ that has a better value of the objective function. This gives the pattern move an improving direction to move in. A new point $\boldsymbol{x}^{(2)}$ is generated by moving from $\boldsymbol{x}^{(0)}$ through $\boldsymbol{x}^{(1)}$ as follows:

$$\boldsymbol{x}^{(2)} = \boldsymbol{x}^{(0)} + a\left[\boldsymbol{x}^{(1)} - \boldsymbol{x}^{(0)}\right]$$

where a is a positive *acceleration factor* that just multiplies the length of the improving direction vector given by $\boldsymbol{x}^{(1)} - \boldsymbol{x}^{(0)}$. A common choice is a=2, in which case the equation reduces to:

$$\boldsymbol{x}^{(2)} = 2\boldsymbol{x}^{(1)} - \boldsymbol{x}^{(0)}$$

Complete Algorithm

The complete algorithm requires 4 inputs in addition to the objective function to be optimized:

- A starting point $\boldsymbol{x}^{(0)}$,
- The value of the acceleration factor a,
- The initial perturbation vector $\boldsymbol{P}_0$,
- The perturbation tolerance vector $\boldsymbol{T} = (t_1, t_2, t_3, \dots, t_n)$. As we will see later in the complete algorithm, this gives the smallest possible perturbation that can be considered for each variable, and is used to halt the algorithm.

The complete algorithm has 3 main parts: initialization, the start/restart routine, and the pattern move routine.

Initialization:

- choose the values for the starting point $\boldsymbol{x}^{(0)}$, acceleration factor a, perturbation vector $\boldsymbol{P}_0$, and perturbation tolerance vector $\boldsymbol{T}$. Initialize the current perturbation vector: $\boldsymbol{P} \leftarrow \boldsymbol{P}_0$.

Start/Restart Routine:

- Use an exploratory search around $\boldsymbol{x}^{(0)}$ to find an improved point $\boldsymbol{x}^{(1)}$ that has a better value of the objective function.
- IF the exploratory search fails (i.e. $\boldsymbol{x}^{(1)}$ does not give a better value of the objective function than $\boldsymbol{x}^{(0)}$) then:
    - Reset all of the perturbations to ½ their current size, i.e. $\boldsymbol{P} \leftarrow \boldsymbol{P}/2$.
    - If any member of $\boldsymbol{P}$ is now smaller than its corresponding perturbation tolerance in $\boldsymbol{T}$, then exit with $\boldsymbol{x}^{(0)}$ as the solution. Else go to *Start/Restart*.
- ELSE [$\boldsymbol{x}^{(1)}$ gives a better value of the objective function than $\boldsymbol{x}^{(0)}$, so we have an improving direction]:
    - Reset the perturbation vector to its original values: $\boldsymbol{P} \leftarrow \boldsymbol{P}_0$.
    - Go to *Pattern Move*.

Pattern Move:

- Obtain tentative $\boldsymbol{x}^{(2)}$ by a pattern move from $\boldsymbol{x}^{(0)}$ through $\boldsymbol{x}^{(1)}$.
- Obtain final $\boldsymbol{x}^{(2)}$ by an exploratory search around tentative $\boldsymbol{x}^{(2)}$.
- IF $f(\boldsymbol{x}^{(2)})$ is worse than $f(\boldsymbol{x}^{(1)})$ then:
    - Update points: $\boldsymbol{x}^{(0)} \leftarrow \boldsymbol{x}^{(1)}$. [$\boldsymbol{x}^{(1)}$ is the best point seen so far]
    - Go to *Start/Restart*.
- ELSE [$f(\boldsymbol{x}^{(2)})$ is better than or equal to $f(\boldsymbol{x}^{(1)})$]:
    - Update points: $\boldsymbol{x}^{(0)} \leftarrow \boldsymbol{x}^{(1)}$ and $\boldsymbol{x}^{(1)} \leftarrow \boldsymbol{x}^{(2)}$.
    - Go to *Pattern Move*.

Note that pattern moves are repeated as long as they are successful, and usually become longer and longer. But as soon as a pattern move fails by producing an $f(\boldsymbol{x}^{(2)})$ that is worse than the previous best value of the objective function $f(\boldsymbol{x}^{(1)})$, then the pattern move is retracted and we go back to an exploratory search around the best point seen so far.

Note also that a pattern move first obtains a tentative $\boldsymbol{x}^{(2)}$ and then finalizes $\boldsymbol{x}^{(2)}$ by an exploratory search. This helps the search to "curve", as we will see in the upcoming example. Lastly note how the algorithm stops: an exploratory search around the best point seen so far fails at all sizes of perturbation, even the smallest as specified by the perturbation tolerances in $\boldsymbol{T}$. This means that the algorithm cannot find an improving direction away from the best point, hence it must be the optimum point. This test can be fooled though, as we will see later.

Example:

Minimize $f(\boldsymbol{x}) = 3x_1^2 + x_2^2 - 12x_1 - 8x_2$

Initialize:

- Initial point: $\boldsymbol{x}^{(0)}$ = (1, 1). $f(\boldsymbol{x}^{(0)})$ = -16.
- Acceleration factor a = 2.
- Perturbation vector $\boldsymbol{P}_0$ = (0.5, 0.5).
- Perturbation tolerance vector $\boldsymbol{T}$ = (0.1, 0.1).
- $\boldsymbol{P} \leftarrow \boldsymbol{P}_0$.

  Note that these are not very good choices for $\boldsymbol{P}_0$ and $\boldsymbol{T}$. They are chosen in this case just so that the algorithm terminates after a small number of steps. The elements in $\boldsymbol{T}$ would normally be much smaller.

Start/Restart:

- $f_{\text{best}} = f(\boldsymbol{x}^{(0)})$ = -16.
- Try $\boldsymbol{x}^{(1)}$ = (1.5, 1). $f(\boldsymbol{x}^{(1)})$ = -18.25, so keep the perturbation and update f_{best} = -18.25.
- Try $\boldsymbol{x}^{(1)}$ = (1.5, 1.5). $f(\boldsymbol{x}^{(1)})$ = -21, so keep the perturbation and update f_{best} = -21.

  The steps in the exploratory search are shown in this first Start/Restart, but are omitted from here forward.

Pattern Move from $\boldsymbol{x}^{(0)}$ = (1, 1) through $\boldsymbol{x}^{(1)}$ = (1.5, 1.5):

- Tentative $\boldsymbol{x}^{(2)}$ = 2(1.5, 1.5) – (1, 1) = (2, 2). f(2, 2) = -24.
- Final $\boldsymbol{x}^{(2)}$ after exploratory search around tentative $\boldsymbol{x}^{(2)}$ is (2.0, 2.5). $f(\boldsymbol{x}^{(2)})$ = -25.75 is better than $f(\boldsymbol{x}^{(1)})$ = -21 so the move is accepted.
- Update points: $\boldsymbol{x}^{(0)} \leftarrow \boldsymbol{x}^{(1)}$ = (1.5, 1.5) and $\boldsymbol{x}^{(1)} \leftarrow \boldsymbol{x}^{(2)}$ = (2.0, 2.5).

Pattern Move from $\boldsymbol{x}^{(0)}$ = (1.5, 1.5) through $\boldsymbol{x}^{(1)}$ = (2.0, 2.5):

- Tentative $\boldsymbol{x}^{(2)}$ = 2(2.0, 2.5) – (1.5, 1.5) = (2.5, 3.5). f(2.5, 3.5) = -27.
- Final $\boldsymbol{x}^{(2)}$ after exploratory search around tentative $\boldsymbol{x}^{(2)}$ is (2.0, 4.0). $f(\boldsymbol{x}^{(2)})$ = -28 is better than $f(\boldsymbol{x}^{(1)})$ = -25,75 so the move is accepted.
- Update points: $\boldsymbol{x}^{(0)} \leftarrow \boldsymbol{x}^{(1)}$ = (2.0, 2.5) and $\boldsymbol{x}^{(1)} \leftarrow \boldsymbol{x}^{(2)}$ = (2.0, 4.0).

Pattern Move from $\boldsymbol{x}^{(0)}$ = (2.0, 2.5) through $\boldsymbol{x}^{(1)}$ = (2.0, 4.0):

- Tentative $\boldsymbol{x}^{(2)}$ = 2(2.0, 4.0) – (2.0, 2.5) = (2.0, 5.5). f(2.0, 5.5) = -25.75.
- Final $\boldsymbol{x}^{(2)}$ after exploratory search around tentative $\boldsymbol{x}^{(2)}$ is (2.0, 5.0). $f(\boldsymbol{x}^{(2)})$ = -27 is worse than $f(\boldsymbol{x}^{(1)})$ = -28 so the move is rejected.
- Update points: $\boldsymbol{x}^{(0)} \leftarrow \boldsymbol{x}^{(1)}$ = (2.0, 4.0).

Start/Restart:

- Exploratory search around $\boldsymbol{x}^{(0)}$ = (2.0, 4.0) fails at all levels of perturbation size.
- Exit with solution $\boldsymbol{x}^{(0)}$ = (2.0, 4.0) and f($\boldsymbol{x}^{(0)}$) = -28.

The table below shows the sequence of points in the solution to our sample problem, and these are plotted in Figure 17.2.

a: $f(1, 1) = -16$	e: $f(2.5, 3.5) = -27$
b: $f(1.5, 1.5) = -21$	f: $f(2, 4) = -28$ [*eventual* s*olution*]
c: $f(2, 2) = -24$	g: $f(2, 5.5) = -25.75$
d: $f(2, 2.5) = -25.75$	h: $f(2,5) = -27$

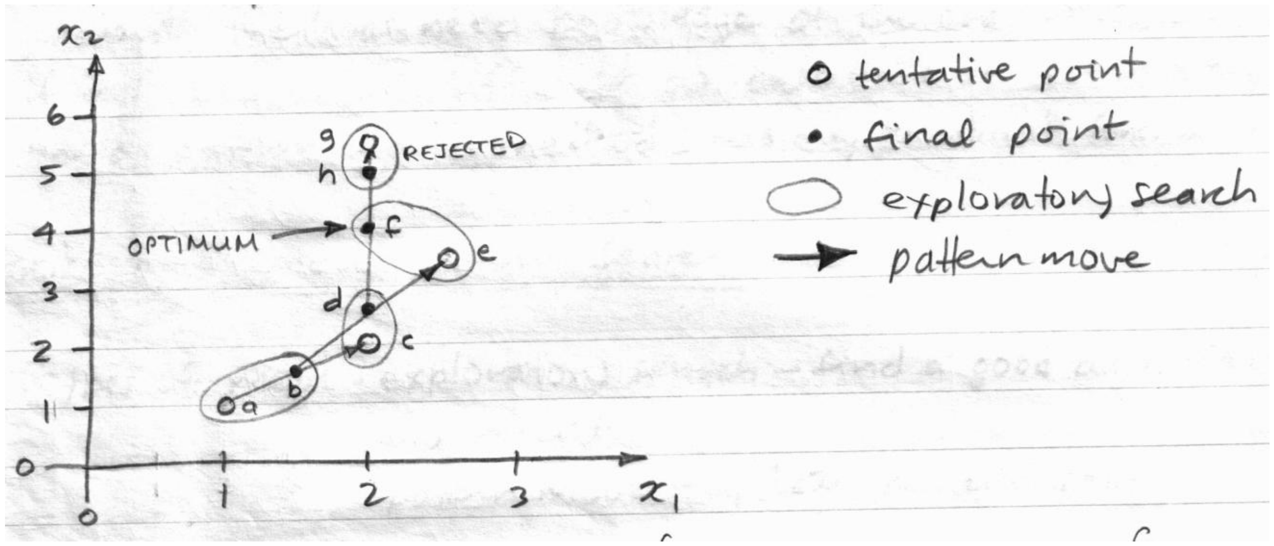


Figure 17.2: Plot of the example.

Note how the exploratory search around each tentative $\boldsymbol{x}^{(2)}$ allows the search to "curve" around corners.

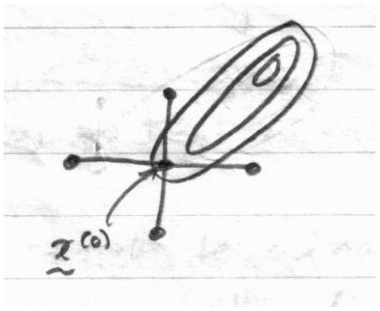


Figure 3: Failure of the exploratory search.

As mentioned earlier, the stopping conditions in this heuristic method can be fooled. This happens when the exploratory search "steps over" a feature, and is therefore unable to find an improving direction even when one exists. This can happen as shown in Figure 17.3, in which the contour lines taper to an optimum point in the small circle at the upper right. However, all of the individual exploratory search steps around $\boldsymbol{x}^{(0)}$ fail, even though an improving direction is available. The simple Hooke and Jeeves exploratory search is unable to find that improving direction, but remember that it is a heuristic after all. However it is easy to improve and extend the basic method. For example, if the exploratory search fails, it could be augmented by testing a few random search directions from $\boldsymbol{x}^{(0)}$.

Other Pattern Search Methods

There are a variety of other pattern search methods. One of the best known is the *Nelder-Mead method*, sometimes (confusingly) called the *downhill simplex method*. This method maintains a polytope of n+1 points in an n-dimensional problem. In two dimensions this polytope is a triangle. The objective function value is evaluated at each of the points, and hence some improving directions can be identified. Movements are then made by "flipping" the triangle to move the worst point, expanding or shrinking the triangle etc. An excellent animation of the Nelder-Mead algorithm can be seen on Wikipedia at http://en.wikipedia.org/wiki/File:Nelder_Mead2.gif.

The Matlab Global Optimization Toolbox includes (as of 2014) 3 pattern search methods: generalized pattern search, generating set search, and mesh adaptive search. The optimization package for the free Matlab-like Octave system includes a Nelder-Mead implementation.

An interesting sub-field of pattern search research is determining how to find a good solution for an unconstrained problem with the smallest number of function evaluations. This is important when each function evaluation is expensive, either in time or actual money. It could be that finding $f(\boldsymbol{x})$ for some value of $\boldsymbol{x}$ requires a long-running simulation (expensive in terms of time), or it requires the construction and testing of a physical prototype (expensive in terms of both time and money). Some of work along these lines can be found under the search term "efficient global optimization".

Chapter 18: Constrained Nonlinear Programming

Unconstrained nonlinear programming is hard enough, but adding constraints makes it even more difficult. Any point in an unconstrained problem is feasible (though probably not optimal), but in constrained NLP a random point may not even be feasible because it violates one or more constraints! Look back at Reasons 3, 5, and 6 of Chapter 16 for specifics on how adding constraints complicates a nonlinear programming problem.

A recurring theme in NLP is trying to convert a difficult problem into something we already know how to solve. We saw this with gradient search where we turned the multi-dimensional unconstrained problem into a series of one-dimension unconstrained problems that we already knew how to solve. It's no different for handling constraints: the first thing we'll try is converting constrained problems to unconstrained problems that we already know how to solve. We'll need *penalty* and *barrier* methods for this.

It's important not to confuse these two. *Penalty methods* are used to find a feasible point (i.e. a point that satisfies all of the constraints simultaneously), and work by applying a penalty related to some measure of the amount of infeasibility. In other words they try to reduce the infeasibility to zero. *Barrier methods*, on the other hand, are used to prevent the infeasibility measure from growing, and are typically used when we already have a feasible point (at least relative to the inequality constraints). Think of it this way: penalty methods drive you into a feasible region, and barrier methods prevent you from leaving it (roughly speaking).

Penalty Methods to Find a Feasible Point

Penalty methods are normally used as a kind of nonlinear phase 1 whose goal is simply to find a feasible point, any feasible point. Bear in mind that the feasible regions may be disconnected, so a successful run of a penalty method will simply find a point in just one of the possibly many feasible regions in the NLP.

Penalty methods often ignore the objective function $f(\mathbf{x})$ entirely by replacing it with a new objective function that focuses entirely on violations of the constraints (both regular constraints and variable bounds). A definition of a *constraint violation* looks like this:

Constraint type	*Violation*
$g_i(\mathbf{x}) \leq \text{b}$	$max(0, g_i(\mathbf{x}) - \text{b})$
$g_i(\mathbf{x}) \geq \text{b}$	$max\ (0, \text{b} - g_i(\mathbf{x}))$
$g_i(\mathbf{x}) = \text{b}$	$\| g_i(\mathbf{x}) - \text{b}\|$
Variable bound: $x_j \leq \text{b}$	$max\ (0, x_j - \text{b})$
Variable bound: $x_j \geq \text{b}$	$max\ (0, \text{b} - x_j)$
Variable bound: $x_j = \text{b}$	$\|x_j - \text{b}\|$

Now if there are I constraints $\text{f}_i(\mathbf{x})$ for i=1...I and J variables x_j for j=1...J, and we define the violation as $v(g_i(\mathbf{x}))$ for some constraint i, and as $v(x_j)$ for the bound on some variable x_j, then some common penalty functions $p(\mathbf{x})$ are:

- Minimize the sum of the constraint violations:

$$Minimize\ p(\boldsymbol{x}) = \sum_{i=1}^{I} v\big(g_i(\boldsymbol{x})\big) + \sum\nolimits_{j=1}^{J} v(x_j)$$

- Minimize the sum of the squared constraint violations:

$$Minimize\ p(\boldsymbol{x}) = \sum_{i=1}^{I} \big[v\big(g_i(\boldsymbol{x})\big)\big]^2 + \sum\nolimits_{j=1}^{J} \big[v(x_j)\big]^2$$

Now we solve an <u>unconstrained</u> problem that has only the selected penalty function as the objective function. However to evaluate this unconstrained objective function value at a point we still have to inspect the constraint functions at that point to determine their contribution to the objective function if they are violated. The minimum value for either version of the penalty function is zero, and this is achieved only at a feasible point (where all constraints are satisfied).

It is common to add a nonnegative "strength" constant multiplier μ to the penalty function, so that it becomes, for example $Minimize\ \mu p(\boldsymbol{x}) = \mu \sum_{i=1}^{I} \big[v\big(g_i(\boldsymbol{x})\big)\big]^2 + \ \mu \sum_{j=1}^{J} \big[v(x_j)\big]^2$. It is also possible to combine the original objective function with the penalty function so that we seek both feasibility and optimality simultaneously. For example, if the original objective function $f(\boldsymbol{x})$ seeks to maximize, then the combined objective function is: *maximize* $F(\boldsymbol{x}) = f(\boldsymbol{x}) - \mu p(\boldsymbol{x})$. As you can see, any constraint violations worsen the value of $f(\boldsymbol{x})$ achieved at any particular point. The problem with the combined objective function is that it can lead you to a point that has a locally optimum value of $F(\boldsymbol{x})$, but which is not actually a feasible point. This happens where the value of $f(\boldsymbol{x})$ is especially attractive, but it is just a little infeasible, so the penalty does not sufficiently outweigh the attractiveness of the point. *Exact penalty methods* try to handle this problem by adjusting the μ multiplier so that any optimum point of F(x) is also an optimum point for the original constrained problem.

A natural question is how to solve the unconstrained model consisting of either just the penalty function $p(\boldsymbol{x})$ or the combined objective function $F(\boldsymbol{x})$ since they have incorporate discontinuous violation measures like $max(0, g_i(\boldsymbol{x}) - \mathrm{b})$ that are not differentiable. Of course pattern search methods can be used, but it would be nice to be able to use the more efficient gradient methods. As it happens, minimizing the squared version of the penalty function is differentiable: just use the right version of the violation function for the current point.

Barrier Methods

Barrier methods work differently for inequality and equality constraints. For inequality constraints (and variable bounds) they require a starting point that <u>oversatisfies</u> the constraint, e.g. if the constraint is $g(\boldsymbol{x}) \leq 10$, then a point at which $g(\boldsymbol{x}) = 9$ oversatisfies the constraint. The effect of the barrier method is then to prevent the point from moving onto the limiting value of the constraint (i.e. moving to $g(\boldsymbol{x}) = 10$ in this example). It does this by applying an increasing repulsive effect as the point moves nearer and nearer to the limiting value of the inequality. Think of it as an electric fence to keep sheep inside a field (which is the feasible region) and that pushes on the sheep harder and harder as they approach the fence. This is why they are named "barrier methods": they provide a barrier to prevent escape from the feasible region.

One common form of the barrier function for a constraint of the form $g_i(\boldsymbol{x}) \leq b_i$ is

$$B(\boldsymbol{x}, r) = \frac{r}{b_i - g_i(\boldsymbol{x})}$$

where r is a nonnegative parameter that adjusts the strength of the barrier "resistance". You can see that this function works well as long as $g_i(\boldsymbol{x}) \leq b_i$, but if $g_i(\boldsymbol{x})=b_i$ then the function will blow up, and if $g_i(\boldsymbol{x}) \geq b_i$ it will give the wrong sign. Further, as $g_i(\boldsymbol{x})$ approaches b_i the size of the barrier function increases rapidly, until the blow-up at $g_i(\boldsymbol{x})=b_i$. This form works equally well for $\geq$ inequalities: just multiply the inequality by -1 and treat as $\leq$ type.

Simple variable bounds are just a special case of inequality constraints. The standard nonnegativity constraint of the form $x_j \geq 0$ has this barrier form: $B(\boldsymbol{x}, r) = \frac{r}{x_j}$.

But how do we find an initial point that oversatisfies the inequality constraints and variable bounds? You can use a penalty method for that.

Barrier methods have a different effect for equality constraints. Since it's very hard to find an initial point that satisfies a set of inequalities, barrier methods instead try to draw the initial point closer to exactly satisfying the equality constraint. A common form of the barrier function for an equality constraint $g_i(\boldsymbol{x})=b_i$ is:

$$B(\boldsymbol{x}, r) = \frac{\left(b_i - g_i(x)\right)^2}{\sqrt{r}}$$

Here, the effect is to reduce the size of the barrier function as $g_i(\boldsymbol{x})$ approaches b_i. So instead of pushing the point away as for an inequality constraint, the barrier function draws the point toward an equality constraint. Note that this effect increases in strength as r decreases in size.

Barrier methods are similar to penalty methods in that they convert the original constrained problem into an unconstrained problem, usually by combining the original objective function with a barrier function that stands in for the constraints. Then we can use the methods that we already have for unconstrained problems to solve this new version of the original model. The standard way to do this follows next.

Sequential Unconstrained Minimization (Maximization) Technique: SUMT

The *Sequential Unconstrained Minimization Technique* (or *Maximization* if that is what you are doing) is the usual way in which constrained problems are converted to an unconstrained form and solved that way. It makes use of barrier methods, and may use penalty methods as well to find a suitable initial point that oversatisfies the inequality constraints.

The main idea is to solve a *sequence* of unconstrained problems that differ only in the value of the resistance factor *r*. This factor will gradually be reduced, which has the effect of weakening the repulsive effect of the barrier function for the inequalities and strengthening its attractive effect for the equality constraints. Why do this? At first we want to make sure that the current point does not stray out of the feasible region by crossing over any of the inequality constraints or variable bounds, so we start with a high repulsive force (i.e. a larger value of *r*). But on the other hand it is possible that the optimum solution is exactly on the limiting value of an inequality constraint, so we gradually reduce the repulsive effect of the inequalities to see if the solution point will move towards the inequality boundaries. At the same time, reducing the value of *r* increases the attractive effect of the equality constraints because we want to make sure that they are satisfied in the end. The rate of reduction in the value of *r* is controlled by a parameter θ which has a value between 0 and 1.

The steps of the SUMT algorithm are as follows:

0. Choose values for the parameters: (i) initial value of *r*, (ii) θ to control the rate of reduction of r, and (iii) ε, the acceptable error tolerance. Let *k* represent the overall iteration number, initially set at 0.
1. Find a suitable initial point $\boldsymbol{x}_0$ that oversatisfies the inequality constraints and variable bounds and is as close as possible to satisfying the equality constraints. Could use a penalty method for this.
2. Construct the barrier function $B(\boldsymbol{x},r)$ having one term for each constraint and variable bound. Combine this with the original objective function, e.g. minimize $f(\boldsymbol{x})$, to create the unconstrained combined objective function, e.g. *minimize* $F(\boldsymbol{x}) = f(\boldsymbol{x}) + B(\boldsymbol{x},r)$.
3. Using the starting point $\boldsymbol{x}_k$, find the optimum point $\boldsymbol{x}_{k+1}$ for $F(\boldsymbol{x})$ using a technique for unconstrained NLP, e.g. the gradient method.
4. Check the stopping conditions: if $|\boldsymbol{x}_{k+1} - \boldsymbol{x}_k| \leq \boldsymbol{\varepsilon}$ then exit with $\boldsymbol{x}_{k+1}$ as the solution.
5. Reduce the value of *r*: $r \leftarrow r\theta$. Increment *k*: $k \leftarrow k+1$. Go to Step 3.

Let's look at an example reformulation for solution via SUMT. Here is the original constrained NLP model:

$$maximize\ f(\boldsymbol{x}) = x_1^4 - 3x_1x_2^3 + x_3^2 - 8$$

$$subject\ to: x_1 + x_2^2 \leq 18$$

$$x_3^2 + x_1 = 25$$

$$x_1 \geq 0, x_2 \geq 0, x_3\ unrestricted$$

The resulting combined objective will be to *maximize* $F(\boldsymbol{x},r) = f(\boldsymbol{x}) - B(\boldsymbol{x},r)$. Note that we subtract the barrier function in this case so it acts against the required maximization. The barrier function has terms for the inequality constraint, the equality constraint, and the two nonnegative variable bounds (note that x_3 is unrestricted), as follows:

$$B(\boldsymbol{x},r) = \frac{r}{18 - x_1 - x_2^2} + \frac{(25 - x_3^2 - x_1)^2}{\sqrt{r}} + \frac{r}{x_1} + \frac{r}{x_2}$$

So the resulting <u>unconstrained</u> combined objective function to use in the SUMT algorithm is:

$$maximize\ F(\boldsymbol{x},r) = x_1^4 - 3x_1x_2^3 + x_3^2 - 8 - \left[\frac{r}{18 - x_1 - x_2^2} + \frac{(25 - x_3^2 - x_1)^2}{\sqrt{r}} + \frac{r}{x_1} + \frac{r}{x_2}\right]$$

Now let's look at a complete solution for the following model:

$$maximize\ f(x) = -x_1^2 - 4x_2^2 + 8x_1 + 16x_2$$
$$subject\ to\colon x_1 + x_2 \leq 5$$
$$x_1 \leq 3$$
$$x_1 \geq 0, x_2 \geq 0$$

Note that the objective function is identical to the unconstrained model that we solved on page 12 of Chapter 16. As we found there, the optimum solution when there are no constraints is *f*(4,2) = 32. But this solution is infeasible for the constrained version of the problem above: the point (4,2) violates the first two constraints. So now let's solve the constrained version of the problem using the SUMT method.

Step 0: Set the stopping error tolerance at ε=0.01, which is fairly large, but we choose it so that the calculations do not run through too many iterations in this example. We'll also choose the initial value of the repulsion factor as *r*=1.0, and the modification factor for *r* as θ=0.1, both typical values.

Step 1: We need an initial point $\boldsymbol{x}_0$ that oversatisfies the inequality constraints (i.e. satisfies them, but not right at their limiting values), and that is as close as possible to satisfying the equality constraints exactly. Sometimes it is easy to choose an initial point by inspection, e.g. in this case $\boldsymbol{x}_0$ = (1,1) will do. Otherwise you may need to run a penalty method on the constraints.

Step 2: The combined objective function is:

$$maximize\ F(\boldsymbol{x},r) = -x_1^2 - 4x_2^2 + 8x_1 + 16x_2 - \left[\frac{r}{5 - x_1 - x_2} + \frac{r}{3 - x_1} + \frac{r}{x_1} + \frac{r}{x_2}\right]$$

Steps 3-5: Now we solve the unconstrained NLP specified in Step 2 using a gradient search algorithm. The results of several of the steps are summarized in the following table:

Iteration	*r*	*Start point*	*Gradient search solution*	*f(**x**)*	*F(**x**,r)*
1	1.0	$\boldsymbol{x}_0$=(1.00, 1.00)	$\boldsymbol{x}_1$=(2.31, 1.85)	29.054	25.44
2	0.1	$\boldsymbol{x}_1$=(2.31, 1.85)	$\boldsymbol{x}_2$= (2.75, 1.90)	30.398	29.62
...					
6	0.00001		$\boldsymbol{x}_6$= (~3.00, 1.97)	30.996	30.99
7	0.000001	$\boldsymbol{x}_6$= (~3.00, 1.97)	$\boldsymbol{x}_7$= (~3.00, 1.97)	30.996	30.99

As we see in the table, the gradient search finds different solutions as the strength parameter *r* is reduced. Eventually it terminates when the gradient search solutions are sufficiently similar in two successive solutions, in this case at around (3.00, 1.97) because of our relatively large stopping tolerance. This is close to the true optimum solution of (3.00, 2.00) that we would reach with a different choice of the stopping tolerance. Note that this is at the intersection of the two constraints $x_1 + x_2 \leq 5$ and $x_1 \leq 3$. These are the two constraints that are violated when we optimize only the objective function without considering the constraints, as on page 12 of Chapter 16.

Note how much work this solution requires. We solve 7 unconstrained versions of the problem (at different values of r). Each of those unconstrained gradient solutions requires the solution of multiple one-dimensional optimization problems. Figure 18.1 shows the location of the first few points in the SUMT solution trajectory. You can see how the trajectory of the solution points will lead to the intersection of the two constraints.

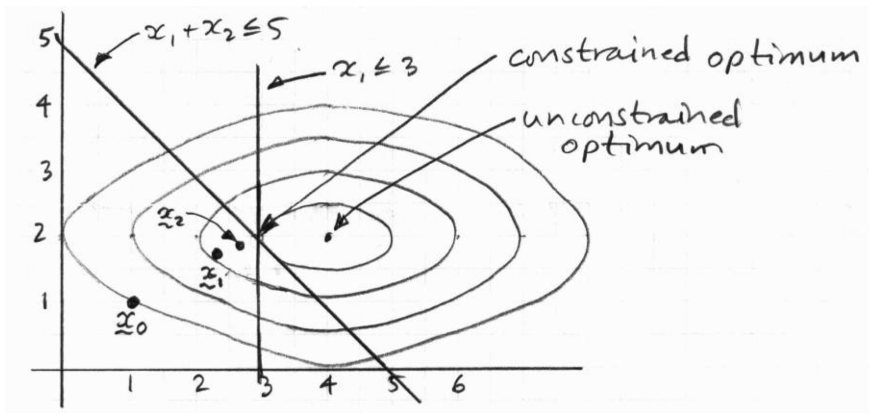


Figure 18.1: Some of the gradient solution end points in the overall SUMT solution.

Dealing with constraints in this way inevitably introduces some distortions, so it is not always possible to get a clean solution as we do in this small example. There are better ways to deal with constraints, as we'll see in the next few chapters.

Remember also that the SUMT method still only returns a local optimum. If you need to find a global optimum to a constrained NLP, then the only thing that you can do at this stage is to try a multistart approach and then take the best solution that you get as your estimated global optimum solution. But multistart is harder with a basic SUMT approach because of the special requirements of the initial $\boldsymbol{x}_0$ point, so you will likely have to couple it with a penalty method in a phase 1 procedure that will find a suitable $\boldsymbol{x}_0$.

Chapter 19: Handling Equality Constraints in NLP

Equality constraints are the most difficult to handle in nonlinear programming: how do you generate a point that lies exactly on a curved surface in a high dimensional space? Even worse, how to you find a point that lies exactly on *multiple* curved surfaces *simultaneously* in a high dimensional space? There are three main approaches:

- Analytic substitution of variables to eliminate equality constraints.
- Lagrange Multipliers
- The Generalized Reduced Gradient (GRG) method.

We'll concern ourselves with problems that have already been analytically reduced, so we'll only cover the second and third methods in the list above. As an aside, note that a significant amount of model simplification is commonly done by the presolve function of modern modelling systems such as AMPL, GAMS, etc.

Lagrange Multipliers

The method of Lagrange multipliers is applicable when the (i) the objective function is differentiable, and (ii) the constraints are all of the equality type. It relies on the following insight: if you have a single equality constraint, then you are at a local optimum point for the model when the gradient of the constraint at that point is *parallel* to the gradient of the objective function at that same point.

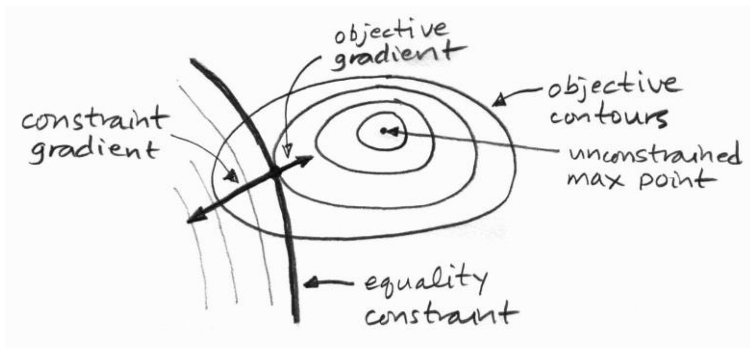


Figure 19.1: Parallel gradients in the method of Lagrange.

Fig. 19.1 illustrates this idea for a maximization objective. At the constrained optimum point on the equality constraint, the objective gradient points up the hill in the general direction of the unconstrained maximum point. At the same point, the gradient of the equality constraint points in exactly the opposite direction. Remember that an equality constraint is just a function like any other, hence you can also draw contour lines for it, and I have sketched a few in faintly. An equality constraint just fixes the function at a particular value, i.e. all solutions *must* be on one particular contour line. Recall also that gradient vectors always point in the direction of the maximum rate of *increase* of the function value. In Figure 19.1 it happens that the direction of increase in the LHS of the equality constraint is down and to the left in the diagram, so the gradient of the equality constraint points in that direction. In the diagram, the gradient of the objective function and the gradient of the equality constraint are parallel (or collinear). It happens that they have opposite directions, but that doesn't matter. If the direction of increase of the equality function had been up and to the right in the diagram, then the two gradients would be on top of each other, with both pointing in the same direction.

As you can see in the diagram, if you move along the equality constraint even slightly you arrive at a worse value of the objective function. And at this new point, the objective and constraint gradients will no longer be parallel. The two are only "in balance" at a local optimum point. Think of the objective function as providing gravity in its gradient direction and the solution point as a marble rolling along a track provided by the equality constraint: the marble will come to rest at a local optimum point where the gradients are in balance. If the gradients aren't in balance, then the marble will roll some more.

Note that the two gradients at the local optimum can be in the same or opposite directions, as long as they are collinear, and can have different lengths. The relationship between the two gradients is expressed by a factor called the *Lagrange multiplier* (λ), which adjusts for direction (has positive or negative sign) and length.

When there are multiple equality constraints, a local optimum is found at a point where the objective function gradient is balanced by a linear combination of the gradients of the equality constraints. Suppose there are *m* independent equality constraints in *n* variables, where $m<n$ (otherwise the problem would have a single solution if $m=n$, or be overconstrained and have no solution if $m>n$):

$$\begin{gathered}\max or \min f(\boldsymbol{x}) \\ s.t.\; g_1(\boldsymbol{x}) = b_1 \\ g_2(\boldsymbol{x}) = b_2 \\ \vdots \\ g_m(\boldsymbol{x}) = b_m\end{gathered}$$

The first step is to convert the entire problem to the *Lagrange function* which incorporates the constraints:

$$h(\boldsymbol{x},\boldsymbol{\lambda}) = f(\boldsymbol{x}) - \sum_{i=1}^{m} \lambda_i [g_i(\boldsymbol{x}) - b_i]$$

where λ_i are the Lagrange multipliers, one for each constraint. Now we find the critical points of the Lagrange function by setting the partial derivatives to zero and solving simultaneously. The set of partial derivatives with respect first to the original variables in $\boldsymbol{x}$, and next with respect to the Lagrange multipliers $\boldsymbol{\lambda}$ is:

$$for\ j = 1 \dots n:\ \frac{\delta h(\boldsymbol{x},\boldsymbol{\lambda})}{\delta x_j} = \frac{\delta f(\boldsymbol{x})}{\delta x_j} - \sum\nolimits_{i=1}^{m} \lambda_i \frac{\delta g_i(\boldsymbol{x})}{\delta x_j} = 0 \quad (1)$$

$$for\ i = 1 \dots m:\ \frac{\delta h(\boldsymbol{x},\boldsymbol{\lambda})}{\delta \lambda_i} = -g_i(\boldsymbol{x}) + b_i = 0 \quad (2) \qquad (2)$$

Now here's the interesting thing: Equation (1) expresses the desired relationship that the gradient of the objective function is parallel to a linear combination of the gradients of the constraints, where the λ_i are weighting factors related to the relative length and direction (given by the sign) of the equality constraint gradients. This is the condition we need to satisfy for local optimality. BUT we must also make sure to satisfy the original equality constraints, and this is exactly what Equation (2) expresses! So, if some point $(\boldsymbol{x},\boldsymbol{\lambda})$ is a critical point for the Lagrange function $h(\boldsymbol{x},\boldsymbol{\lambda})$, then it is a *feasible* critical point for the original model. But just a critical point: it could be a local maximum or a local minimum or even a saddle point for the original problem: you need to do some additional testing to determine which.

The three main steps in the method of Lagrange are:

1. Convert the model to the Lagrange function form.
2. Set all of the partial derivatives to zero and solve the resulting set of equations. Note that this can be a very difficult problem all by itself: the equations are likely to be nonlinear. You must apply a technique for solving sets of nonlinear equations, e.g. the Newton-Raphson method.
3. Test any critical points that are found to see if they are local maxima or minima for the original model. This may *also* be difficult. Recall that you can't simply check the Hessian of the objective function because this is a constrained problem, so you likely are not at an unconstrained local optimum of the objective function (which can be positively identified by the Hessian).

So there are several practical difficulties with the method of Lagrange: (i) you may not be able to solve the set of nonlinear equations in Step 2 because your numerical solution method fails, or because there is no solution, and (ii) deciding the optimality status of any point(s) output by Step 2.

Let's take a look at a simple example. I'm making it easy by using a quadratic objective function (so it's partial derivatives are linear) and a linear constraint. This makes Step 2 easy: all of the equations are linear. You won't normally be so lucky in real life. Here's the model:

$$maximize\ f(\boldsymbol{x}) = -2x_1^2 - x_2^2 + x_1x_2 + 8x_1 + 3x_2$$
$$s.t.\ 3x_1 + x_2 = 10$$

So the Lagrange function is:

$$h(\boldsymbol{x},\lambda) = -2x_1^2 - x_2^2 + x_1x_2 + 8x_1 + 3x_2 - \lambda(3x_1 + x_2 - 10)$$

The resulting partial derivative equations are:

$$\frac{\delta h(\boldsymbol{x},\lambda)}{\delta x_1} = -4x_1 + x_2 + 8 - 3\lambda = 0$$
$$\frac{\delta h(\boldsymbol{x},\lambda)}{\delta x_2} = -2x_2 + x_1 + 3 - \lambda = 0$$
$$\frac{\delta h(\boldsymbol{x},\lambda)}{\delta \lambda} = -3x_1 - x_2 + 10 = 0$$

So we have three equations in three unknowns to solve, and luckily they are linear, so it is straightforward to find the solution: $(x_1, x_2, \lambda) = (2.464, 2.607, 0.250)$. So we have found a point that is critical for $h(\boldsymbol{x},\lambda)$ and is both feasible and critical for the original constrained model. But how to test whether it is a local optimum for the original constrained model? In some cases it's obvious: if the objective function is concave and there is only a single Lagrange point returned in Step 2, then it must be a local maximum for the original constrained problem. In other cases you will need to find all of the Lagrange points and then choose the one that gives the largest value of $f(\boldsymbol{x})$ if maximizing, and the smallest value of $f(\boldsymbol{x})$ if minimizing.

Given all the difficulties associated with the method of Lagrange, why do we bother studying it? There are several reasons:

- It is a classical method that still finds frequent use because it is a good match for certain classes of problems.
- It actually applies to inequality constrained problems too: you just need to convert the inequalities to equality constraints by the addition of suitable slack and surplus variables.
- It is the basis of the Karush-Kuhn-Tucker conditions, which are vital for nonlinear programming. This is a set of conditions allowing for the identification of critical points in general nonlinear programming models, as we will see later.
- Finally, there is this theorem: if there is no $\boldsymbol{\lambda}$ at a point $\boldsymbol{x}$ such that $\nabla f(\boldsymbol{x}) - \sum_{i=1}^{m} \lambda_i \nabla g_i(\boldsymbol{x}) = 0$ then $\boldsymbol{x}$ cannot be a local optimum point. This means that if a point is proposed as a being a local optimum you can sometimes use this rule to prove that it is not.

The Generalized Reduced Gradient (GRG) Method

The main idea in the Generalized Reduced Gradient (GRG) method is that under some conditions it is possible to incorporate equality constraints directly into the objective function, thereby reducing the dimensionality of the model and converting it to an unconstrained form (and we know how to solve unconstrained problems: the gradient method comes to mind). Hence the problem has *reduced* dimensionality and can be solved by the *gradient* method. It is easiest to incorporate equality constraints into the objective function if they are linear. Here's a simple example:

$$\begin{array}{ll} min & f(\boldsymbol{x}) = x_1^2 + x_2^2 + x_3^2 + x_4^2 \\ s.t. & g_1(\boldsymbol{x}) = 2x_1 + x_2 + 2x_3 + x_4 - 8 = 0 \\ & g_2(\boldsymbol{x}) = x_1 - x_2 + 4x_3 + x_4 + 2 = 0 \end{array}$$

Notice that the objective function is nonlinear while the two constraints are linear. We can use those two linear constraint equations to solve for two variables in terms of the other two:

$$x_2 = -0.5x_1 + x_3 + 5$$

$$x_4 = -1.5x_1 - 3x_3 + 3$$

Now we can substitute these two equations directly into the objective function to yield an equivalent unconstrained problem in only two dimensions (x_1 and x_3) instead of the original four:

$$minimize\ f(x_1, x_4) = x_1^2 + [-0.5x_1 + x_3 + 5]^2 + x_3^2 + [-1.5x_1 - 3x_3 + 3]^2$$

After we solve this unconstrained problem in x_1 and x_3 we just recover the values of x_2 and x_4 from the equations above.

But what do we do if the constraints are nonlinear? In that case we make linear approximations.

Making a Linear Approximation to a Nonlinear Function at a Given Point

Linear approximation is a standard and much-used technique in nonlinear programming. The linear approximation to some nonlinear function $g(\boldsymbol{x})$ is given by a truncated Taylor's Series expansion around the given point $\bar{\boldsymbol{x}}$:

$$g(\boldsymbol{x}) \approx g(\bar{\boldsymbol{x}}) + \nabla g(\bar{\boldsymbol{x}})^T(\boldsymbol{x} - \bar{\boldsymbol{x}})$$

Notice that the highlighted items $g(\bar{\boldsymbol{x}}), \nabla g(\bar{\boldsymbol{x}})^T$ and $\bar{\boldsymbol{x}}$ are fixed constants at the given point $\bar{\boldsymbol{x}}$. The accuracy of the approximation weakens as you move farther away from $\bar{\boldsymbol{x}}$ where the gradient of the function was evaluated. If the curve is highly nonlinear then the accuracy of the approximation degrades even faster.

Here is an example. Let's linearly approximate a function $g(\boldsymbol{x}) = x_1^2 + x_2 + 4x_3 + 4x_4 - 4$ around some point $\bar{\boldsymbol{x}}$. In expanded form, this is:

$$g(\boldsymbol{x}) \approx \bar{x}_1^2 + \bar{x}_2 + 4\bar{x}_3 + 4\bar{x}_4 - 4 + [2\bar{x}_1 \;\; 1 \;\; 4 \;\; 4]\begin{bmatrix} x_1 - \bar{x}_1 \\ x_2 - \bar{x}_2 \\ x_3 - \bar{x}_3 \\ x_4 - \bar{x}_4 \end{bmatrix}$$

simplifies to $g(\boldsymbol{x}) \approx (2\bar{x}_1)x_1 + x_2 + 4x_3 + 4x_4 - (\bar{x}_1^2 + 4)$

So at a particular point, say $\bar{\boldsymbol{x}} = (2,3,2,5)$, the linearization reduces to:

$$g(\boldsymbol{x}) \approx 4x_1 + x_2 + 4x_3 + 4x_4 - 8 = 0$$

Now suppose we want to find the value of the function at some nearby point (3,4,3,6). This particular version of the linearized function, we find that $g(\boldsymbol{x}) \approx 44$ where the exact value, using the original nonlinear function, is $g(\boldsymbol{x}) = 45$.

Steps in the Generalized Reduced Gradient Method

With that background, we can now take a look at the steps in the GRG algorithm. The main idea is to solve a *sequence* of subproblems, each using a new linearized approximation of the nonlinear constraints at the current point, and absorbing them into the objective function as per the algorithm.

0. Choose an initial point $\boldsymbol{x}$, tolerances and other parameter settings.
1. Linearly approximate the nonlinear constraints around the current point $\boldsymbol{x}$ and solve the resulting reduced unconstrained problem via a gradient method to find a new solution point $\boldsymbol{x}'$.
2. If $|\boldsymbol{x} - \boldsymbol{x}'|$ is small enough, then exit with $\boldsymbol{x}'$ as the solution.

3. $\boldsymbol{x} \leftarrow \boldsymbol{x}'$. Go to Step 1.

There are a few important algorithm details to discuss that must be clarified.

How do you numerically rewrite the linear (or linearly approximated) equality constraints to isolate a subset of the variables?

We don't want to have to do manual algebra to rewrite the linear equations in a form that isolates a subset of variables: we want this to be done algorithmically. This is actually easy to do using a technique borrowed from the simplex method for linear programming. The steps are as follows:

- Divide the variables into *basic* and *nonbasic* sets. It's the basic variables that will be substituted out of the objective function.
- Recall that there is 1 basic variable per constraint.
- Perform Gaussian elimination to get a +1 coefficient for the basic variable for each row, with zeros above and below it, just like putting the simplex tableau into proper form.
- This isolates the basic variables and allows them to be substituted into the objective function. Again we don't need to do this algebraically, this is taken care of algorithmically.

How do you solve the unconstrained subproblem?

Just like in the name of the method, it's usually some form of gradient method. But, technically, there's nothing to stop you from using, say, a pattern search method.

What if you don't have a feasible starting point?

In this case, you can set up and solve a phase 1 problem whose goal is to find a feasible point. It works like this:

- Choose a starting point. Of course a point that is as close as possible to feasible is best.
- Check which constraints are violated at this point. Satisfied constraints remain as phase 1 constraints. Violated constraints form the phase 1 objective, which may be to minimize the sum of the constraint violations, for example.
- Solve this problem via the GRG method.

When phase 1 terminates at a feasible point, switch to the usual full GRG for phase 2, which then retains feasibility.

How does GRG handle inequality constraints?

The first part is to convert all inequality constraints into equality constraints by the addition of suitable slack and surplus variables. The second part is a little harder. As the algorithm proceeds, we must decide which inequalities are active (i.e. whose slack or surplus variables are equal to zero). There are heuristic *active set strategies* for making this decision, based on e.g. how close a constraint is to being satisfied at the current point.

How are variable bounds handled?

These are respected at all times. This isn't hard to do: you simply make sure not to exceed any variable bounds during the gradient method line searches.

There are a number of commercial implementations of the GRG method, including codes by the unsurprising names of GRG, GRG2 and LSGRG (for "large-scale" GRG), many by Lasdon and Waren. One of their GRG codes is actually embedded in Microsoft Excel as the nonlinear solver.

Linearization Schemes in Two Commercial Codes

Linear approximation is frequently a component of nonlinear solvers. It's interesting to see how it is used in completely different ways in two commercial NLP solvers.

LSGRG is (as might be expected) an implementation of the GRG algorithm for large-scale problems. It is a *feasible-path* method, meaning that it requires a feasible point (hence may employ a phase 1 to find a suitable starting point), and maintains feasibility as it proceeds. Technically it stays *close* to feasibility since it uses approximations to the nonlinear constraints which may violate true feasibility by a small amount. It works as described previously, by choosing an active set of constraints at each iteration, which are linearized and incorporated into the objective function. The subproblem is solved via a gradient method, but feasibility is retained by checking each new point. The cycle of choosing constraints, linearizing, solving, and checking is repeated until the stopping conditions are met.

MINOS stands for "Modular In-core Nonlinear Optimization System" and was quite a popular solver for many years, though it has since been superseded. It was extensively used for solving linear programming problems, as well as nonlinear problems, because of its excellent embedded LP solver. The LP solver is much used in its NLP solution approach, which had four stages:

- *Phase 1*: Linearize the nonlinear constraints at the current point and seek a feasible solution to the resulting LP. Since the nonlinear constraints curve, the linear approximations to them vary quite a bit depending on the linearization point, so there is no guarantee that the linearized version of the problem will have a feasible LP solution, even if there is a feasible solution for the nonlinear model. So MINOS incorporated a heuristic for improving the chances of finding a feasible solution for the LP: it would loosen the constraints a certain amount (by adjusting the right hand side constant to make it easier to solve) and try again. It would try this up to five times before giving up and declaring the model to be infeasible.
- *Phase 2*: Find the optimum point for LP established during Phase 1.
- *Phase 3*: Choose the active constraints at the current point, linearize them, and incorporate into the GRG objective function.

- *Phase 4*: Solve the GRG problem. If the stopping conditions are met, then exit with the current ***x*** as the solution, otherwise go to Phase 1.

So both solvers use the GRG algorithm, and hence must use linearization. But they do so in somewhat different ways. It's interesting to see how the basic ingredients of an algorithm can be combined to produced different solvers.

Chapter 20: Function and Region Shapes, the Karush-Kuhn-Tucker (KKT) Conditions, and Quadratic Programming

Function and Region Shapes

As we saw in Chapter 16, nonlinear programming is much harder than linear programming because the functions can take many different shapes: the objective function may have many hills and valleys, and the constraints can interact in ways that give disconnected feasible regions. There are some fundamental definitions for nonlinear function and region shapes; we'll look at these next.

First, definitions that apply to an individual function, whether it is an objective function or a constraint. Take any two points and connect them via an interpolating line. A function is convex if every point on every interpolating line has an interpolated value that is greater than or equal to the function value at the same point. A function is concave if every point on every interpolating line has an interpolated value that is less than or equal to the function value at the same point. These ideas are illustrated in Figure 20.1. Note that a linear function is both convex and concave.

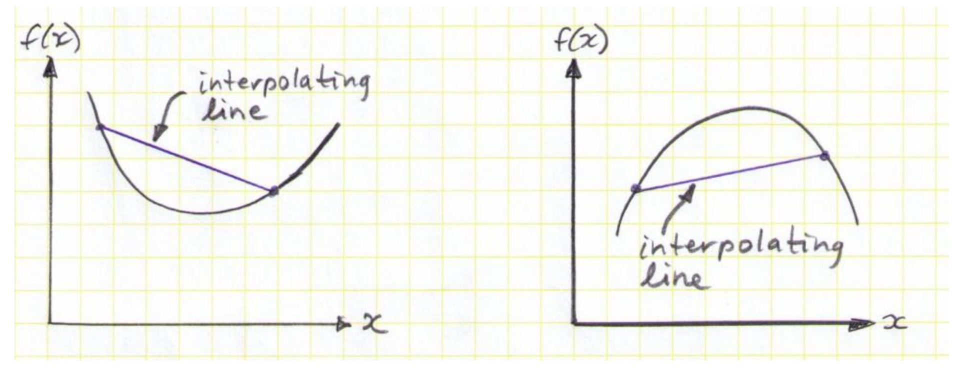


Figure 20.2: A convex function (left) and a concave function (right).

As you can see from the figure, a convex function will have a single local minimum and a concave function will have a single local maximum. This concept extends into multiple dimensions as well, so it corresponds directly with the ideas of positive definiteness of the Hessian matrix (convex function), and negative definiteness (concave function).

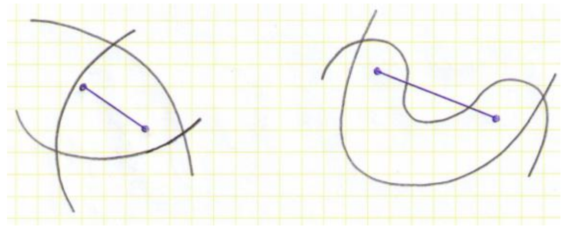


Figure 10.2: A convex set of points (left), and a nonconvex set (right).

We also need definitions that apply to the spaces defined by interacting sets of constraints. The most important concept is a *convex set of points*, defined as follows: all points on any line connecting any two points in the convex set also lie in the convex set. The opposite of a convex set of points is called a *nonconvex set* in which some of the points on the line segment lie outside the set of points. These ideas are illustrated in Fig. 20.2. In a convex set, every point can "see" every other point, but in a nonconvex set there are places where a given point cannot see every other point in the set. This can seriously affect how well a solver works. For example, a steepest descent gradient method may try to move from the current point in the gradient direction, but encounter a constraint which prevents further improvement, even though it could reach a better feasible point if it could only "get around the corner" in the nonconvex set. This doesn't happen when the constraints form a convex set.

Knowledge of the shapes of the functions and the point sets created by the constraints is very useful in selecting the solution algorithm to use. We'll have more to say on this later, but for now observe that when minimizing a convex objective function subject to a convex set of constraints, there is exactly one local minimum, and there is exactly one local maximum when maximizing a concave function subject to a convex set of constraints. In both cases, this single local optimum is by definition also the global optimum.

It's difficult to analytically determine the shapes of functions beyond quadratics (which have a Hessian vector filled with constants), and there are very few software tools for this task either. One approach is to try many random line segments and look at the differences between the interpolated values of the function at the end points and the actual function value at some intermediate points. This is the approach taken by the MProbe software (http://www.sce.carleton.ca/faculty/chinneck/mprobe.html) mentioned in Chapter 16. Another approach is to try to break down a complex function into elemental parts for which the convexity/concavity is known or easily found. There are rules governing the shape of a function composed of convex or concave parts (e.g. the sum of convex functions is itself a convex function), so you can often reason out the shape of the complete function. This is the approach taken by the Dr. AMPL software (http://www.gerad.ca/~orban/drampl/) which operates on nonlinear models expressed in the AMPL modelling language. The final approach is the opposite: building functions from elemental parts in such a way that the resulting function is guaranteed to be convex. This is the approach taken in the CVX software (http://cvxr.com/cvx/).

The Karush-Kuhn-Tucker (KKT) conditions

Thus far we have a small set of methods to use for solving constrained or unconstrained nonlinear programming problems to find local optima. In practice these will always be implemented in computer software which will read a model in some form, calculate for a while, and eventually produce some output. But there is wide variety of possible outputs, some of which are hard to interpret. What does it mean if the solver says that it has made no progress for 25 iterations? Does that mean that it really is at a local optimum? Or does it mean that it is not looking in a promising area?

The best possible output message is one that definitely declares that a local optimum has been found. But how does the solver know this? Typically it is because it has checked the Karush-Kuhn-Tucker (KKT) Conditions. The KKT conditions are used to test a point to determine whether or not it is a critical point in a constrained nonlinear program. They don't actually determine whether the point is a local *optimum*, just that it is a *critical point* (could be a local maximum, a local minimum, or a saddle point). However since the solver is working towards a particular objective (maximization or minimization), it's a good bet that a point that satisfies the KKT conditions is a local optimum of the type sought by the solver.

The KKT conditions recognize that there are two different possibilities for a local optimum point:

A. *No constraints are active at the local optimum point.* In other words there is a local "hill" maximum (or "valley" minimum) in the objective function that is not near the limiting value of any constraint. In a case like this, the gradient will be zero ($\nabla f(\boldsymbol{x}) = \mathbf{0}$) at the local optimum point, and in fact the Hessian can be used to determine whether it is a local maximum, local minimum, or saddle point.

B. *At least one constraint is active at the local optimum point.* In this case some constraint is preventing the search method from improving the value of the objective function. At a point like this, the gradient of the objective function is *not* zero (i.e. $\nabla f(\boldsymbol{x}) \neq \mathbf{0}$) and the Hessian *cannot* be used to determine the optimality status and type of optimum.

The genius of the KKT conditions is that they handle *both* possibilities simultaneously. We'll get started with a simple observation.

Observation: In case B, we know how to use the method of Lagrange for equality constraints, so that will work if there are only equality constraints, but how do we deal with inequalities? This is actually pretty simple: given a point to test, you can determine whether an inequality is active (i.e. the point lies right on the limiting value of the inequality), in which case you can just treat it as if it were an equality constraint. If the inequality is inactive (i.e. it is oversatisfied), then you can just ignore it. To ignore an inactive inequality, just set its Lagrange multiplier to zero (i.e. $\lambda = 0$).

To make use of this observation, we proceed as follows:

- Always write the constraint equations so that the RHS constant is zero. For example a constraint such as $x_1^2 + 3x_2 \leq 5$ becomes $g(\boldsymbol{x}) = x_1^2 + 3x_2 - 5 \leq 0$.
- Now active constraints have value equal to zero, e.g. $g(\boldsymbol{x}) = 0$.
- Inactive constraints have a function value that is *not* equal to zero, e.g. $g(\boldsymbol{x}) = -4$ in the example above. But you can set the associated $\lambda = 0$ because the inequality is inactive.
- Therefore we always have $\lambda_i g_i(\boldsymbol{x}) = 0$ for *i*=1...*m* where there are *m* constraints. This covers both the active and the inactive constraints in one tidy package. Either $g_i(\boldsymbol{x}) = 0$ because the constraint is active, or $\lambda_i = 0$ because the inequality is inactive.

This is the *orthogonality condition*, expressed in matrix form as $\boldsymbol{\lambda}^T \boldsymbol{g}(\boldsymbol{x}) = \mathbf{0}$.

Let's now see if we can use these three elements to identify whether a given point is a critical point in an inequality-constrained NLP: (i) the Lagrange gradient condition, (ii) the orthogonality condition, and (iii) the original inequality constraints. We'll add equality constraints to the mix later. Let's express those three conditions for a system of *m* inequalities in *n* variables:

- *Lagrange gradient condition*: $\frac{\partial h(\boldsymbol{x},\boldsymbol{\lambda})}{\partial x_j} = 0$ for *j*=1...*n*.
- *Orthogonality condition*: $\lambda_i g_i(\boldsymbol{x}) = 0$ for *i*=1...*m*.
- *Original inequalities*: $g_i(\boldsymbol{x}) \leq 0$ for *i*=1...*m*.

At this point, recall that the Lagrange multiplier λ_i can be positive or negative for an equality constraint. This is because you cannot move off an equality constraint anyway, so it doesn't matter whether the increasing direction of the constraint gradient is in the same direction as the objective gradient (gives a positive multiplier), or in the opposite direction (gives a negative multiplier).

But the sign of λ_i does make a difference for active inequality constraints. But why? It's like this: just because the point you are testing makes the inequality active, it doesn't mean that the inequality is preventing you from improving the value of the objective function: it may be that moving towards the interior of the constraint will improve the objective function value. Let's imagine a situation in which you are standing right by a fence (the constraint) on the side of a hill (the objective function), and you would like to get to the top of the hill. The fence constraint is active because you are right by it. There are two possibilities: (*i*) The fence keeps you on the uphill side, in which case there is nothing to prevent you walking right to the top of the hill. In other words, you're at a point where the constraint is active, but it isn't preventing you from improving the value of the objective function. (*ii*) The fence constraint keeps you on the downhill side of the hill, in which case it is both active and *is* preventing you from going higher on the hill. We need to distinguish these two cases to determine whether a given point is a local optimum. In case (*i*) you are *not* at a local optimum, but in case (*ii*) you are. The *sign* of the Lagrange multiplier tells us whether we are in case (*i*) or case (*ii*).

How does this work? First, we require that all inequality constraints be put in the less-than-or-equal form, i.e. $g_i(\boldsymbol{x}) \leq 0$ for i=1...m. As always, the gradient of a function points in the increasing direction. So if a constraint of this form is active, then moving in its gradient direction is prevented: the constraint prevents $g_i(\boldsymbol{x})$ from getting any bigger. Now suppose we wish to maximize the objective function: this means we want to move in the objective function gradient direction. Hence if you are at some point and attempting to move in the objective gradient direction, but an inequality is active, and it's gradient is in the same direction as the objective gradient, then the Lagrange multiplier will be positive, and you will be in case (*ii*) above, i.e. at a local optimum. This concept is sketched in Fig. 20.3.

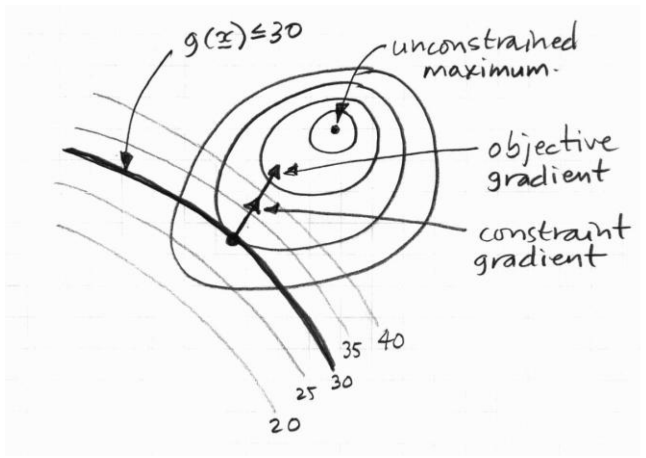


Figure 20.3: Lagrange multiplier nonnegativity condition.

Some gradient lines for the constraint are lightly sketched in Fig. 20.3, along with the value of the function at each of the lines. As you can see, moving in the constraint gradient direction is prevented by the constraint: it disallows values of $g(\boldsymbol{x})$ greater than 30. But it happens that the objective gradient is in the same direction: if we treat the inequality as an equality we will find that the Lagrange multiplier has a positive value.

If the gradient of the constraint pointed in the opposite direction to that shown in Fig. 20.3, moving down and to the left would be the prevented direction of movement, so we could move up the objective hill to the unconstrained maximum point. In this case, treating the inequality as an equality yields a negative Lagrange multiplier.

So we see that an inequality constraint is only tight and "holding" the point when the Lagrange multiplier is positive (for an inequality of the form $g_i(\boldsymbol{x}) \leq 0$). This brings us to the *Lagrange multiplier nonnegativity condition* for inequality constraints (recall that the multiplier can be zero if the inequality is inactive).

The nonnegativity condition works as long as the inequality constraints satisfy a *constraint qualification*: the gradients of the constraints must be linearly independent at the point that is being tested.

Now we can summarize all four of the KKT conditions, as follows:

For a nonlinear program of the form:

$$\text{Max } f(\boldsymbol{x}) \text{ subject to } g_i(\boldsymbol{x}) \leq 0 \text{ for } i = 1 \dots m$$

Where the Lagrangian function is:

$$h(\boldsymbol{x}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) - \sum_{i=1}^{m} \lambda_i \, g_i(\boldsymbol{x})$$

The KKT conditions are:

$$\begin{gathered}
\text{parallel gradients conditions:} \quad \frac{\partial h(\boldsymbol{x}, \boldsymbol{\lambda})}{\partial x_j} = 0 \text{ for } j = 1 \dots n \\
\text{orthogonality conditions:} \quad \lambda_i g_i(\boldsymbol{x}) = 0 \text{ for } i = 1 \dots m \\
\text{satisfaction of original constraints:} \quad g_i(\boldsymbol{x}) \leq 0 \text{ for } i = 1 \dots m \\
\text{Lagrange multiplier nonnegativity:} \quad \lambda_i \geq 0 \text{ for } i = 1 \dots m
\end{gathered}$$

But what if equality constraints are also included in the model? This is easily handled: now we don't care what sign the Lagrange multiplier is because you can't move off an equality constraint in either the gradient or the anti-gradient direction, so just remove the requirement that $\lambda_i \geq 0$ for each equality constraint. The orthogonality constraint is also not needed for an equality constraint because we already have $g_i(\boldsymbol{x}) = 0$, so the orthogonality condition is always satisfied.

What if you are dealing with a minimization instead of maximization? The easy route is to just multiply the objective function by -1 to convert it to a maximization and then apply the conditions listed above. As an alternative, you can keep the minimization objective and replace the Lagrange multiplier nonnegativity conditions by *nonpositivity* conditions.

Example: You have the following NLP model to solve:

$$\begin{gathered}
\text{maximize } f(\boldsymbol{x}) = 10x_1^2 - x_2^2 + x_1 x_2 \\
\text{subject to: } x_1 - x_2^2 \geq 5 \\
x_1, x_2 \text{ unrestricted}
\end{gathered}$$

Your solver has returned the point (9,2), and you want to use the KKT conditions to determine whether this is likely a local maximum point.

The first thing to do is to convert the constraint to the required form, in this case:

$$-x_1 + x_2^2 + 5 \leq 0$$

So the Lagrange function is $h(\boldsymbol{x}, \lambda) = 10x_1^2 - x_2^2 + x_1x_2 - \lambda[-x_1 + x_2^2 + 5]$.

So the KKT conditions for this model are:

Parallel gradients: $\frac{\partial h(\boldsymbol{x},\boldsymbol{\lambda})}{\partial x_1} = 20x_1 + x_2 + \lambda = 0$ and $\frac{\partial h(\boldsymbol{x},\boldsymbol{\lambda})}{\partial x_2} = -2x_2 + x_1 - 2\lambda x_2 = 0$

Orthogonality: $\lambda(-x_1 + x_2^2 + 5) = 0$

Constraint satisfaction: $-x_1 + x_2^2 + 5 \leq 0$

Multiplier nonnegativity: $\lambda \geq 0$

Now let's check these conditions at the point (9,2) returned by the solver. We see immediately that the first parallel gradient condition is only satisfied if $\lambda < 0$, but this contradicts the multiplier nonnegativity condition. Thus we must conclude that the point is *not* a local maximum point. Note though that the constraint is tight at (9,2).

It is sometimes difficult to determine whether the conditions can be satisfied at a given point. The easy part is checking the original constraints to determine whether they are tight (in which case the Lagrange multiplier values must be found) or oversatisfied (in which case we know that the multipliers are zero because of the orthogonality condition). But then, just as for the method of Lagrange, you may need to solve a simultaneous set of possibly nonlinear equations, and that can be difficult. Still, the KKT conditions can be extremely helpful in assessing potential local optima.

Recall that satisfying the KKT conditions only indicates that the point is a critical point for the constrained NLP: there is no general guarantee that it is a maximum or minimum point. However we *can* guarantee that it is the global optimum point in two special cases, provided we know something about the shape of the objective function and the shape of the feasible region:

- *KKT point is a global maximum.* This is guaranteed if the objective function is globally concave and the constrained region is convex.
- *KKT point is a global minimum*. This is guaranteed if the objective function is globally convex and the constrained region is convex.

This happens because there is exactly one KKT point in each case, and it must be of the given type because that is the only type of critical point in the model.

Even better, they are used as the basis for quadratic programming, which allows a quadratic objective function subject to linear constraints, and for quadratically-constrained quadratic programming. This is because the derivatives of the Lagrange function (i.e. the parallel gradients conditions) result in linear equations for quadratic functions. Hence we can essentially set up a linear set of equations to solve a quadratic program. More on this below.

An aside: The KKT conditions were known as the Kuhn-Tucker conditions for many years, following the publication of a paper by Kuhn and Tucker describing them in 1951. Many years later it was noticed that Karush had actually developed the same conditions much earlier in his master's thesis in 1939. But Karush did not publish any papers from his master's work, and so it was unnoticed for decades. The moral of the story for graduate students: publish your work!

Quadratic Programming

A quadratic program (QP) has a quadratic objective function (e.g. $Z = 12x_1^2 - 4x_1x_2 + 5x_3$) subject to linear constraints. Note that the quadratic nature of the objective function means that it can include squared terms, or terms that are the product of two variables, or linear terms. The simplest quadratic program is usually represented as follows:

$$\text{maximize } Z = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{Q}\boldsymbol{x} + \boldsymbol{c}^T\boldsymbol{x}$$
$$\text{subject to } \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$$
$$\boldsymbol{x} \geq \boldsymbol{0}$$

A is the usual *m*×*n* matrix for the linear constraints. ***Q*** is a symmetric *n*×*n* matrix. The ***Q*** matrix makes it easy to represent the second-order terms, which could involve any combination of two variables or the squaring of a single variable. Note that it is entirely possible to formulate the quadratic program without the ½ coefficient and with a ***Q*** matrix that is not symmetric: these just simplify some of the notation.

The ½ constant is required because of the way we construct ***Q***. In our example objective function $Z = 12x_1^2 - 4x_1x_2 + 5x_3$, we set $\boldsymbol{Q} = \begin{bmatrix} 24 & -4 & 0 \\ -4 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$, constructed as follows:

- For squared terms like cx_i^2, write 2*c* in the *ii* position of ***Q***. So the $12x_1^2$ term in our example objective function appears as a 24 in the 1-1 position in ***Q***.
- For quadratic terms involving a pair of variables like cx_ix_j, write *c* in both the *ij* position and the *ji* position in ***Q***. So the $-4x_1x_2$ term in our example objective function appears as two terms: −4 in the 1-2 position and −4 in the 2-1 position in ***Q***.
- Purely linear terms (like the $5x_3$ in our example) do not appear in ***Q*** at all: they appear in the standard linear part of the objective function: $\boldsymbol{c}^T\boldsymbol{x}$.

As mentioned above, the Lagrange conditions will yield linear functions since they are the derivatives of a quadratic function, so we have an *almost* completely linear system to solve to find a solution to this nonlinear problem. We also know the linear constraints form a convex set. Finally, we know that if we are minimizing a convex objective or maximizing a concave objective, subject to a convex set of constraints, then a point satisfying the KKT conditions must be the global optimum point.

So let's set up the KKT conditions for our minimization problem. First we define the Lagrange function a little differently. We use different symbols for the Lagrange multipliers for the *m* row constraints (λ_i, i=1...m) and for the *n* nonnegativity constraints (u_j, j=1...n). And we also need to convert the constraints to the $g_i(\boldsymbol{x}) \leq 0$ form needed for the KKT conditions: $\boldsymbol{A}_i(\boldsymbol{x})$ is the *i*th row of the $\boldsymbol{A}$ matrix, so the *i*th row constraint has the form $\boldsymbol{A}_i(\boldsymbol{x}) - b_i \leq 0$, and the *j*th nonnegativity constraint has the form $-x_j \leq 0$. With those adjustments, the Lagrange function can be written as:

$$h(\boldsymbol{x}, \boldsymbol{\lambda}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{Q}\boldsymbol{x} + \boldsymbol{c}^T\boldsymbol{x} - \sum_{i=1}^{m} \lambda_i\,[\boldsymbol{A}_i(\boldsymbol{x}) - b_i] - \sum_{j=1}^{n} u_j[-x_j]$$

Now we can list the KKT conditions:

$$\text{parallel gradients conditions: } \boldsymbol{Q}\boldsymbol{x} + \boldsymbol{c}^T - \boldsymbol{\lambda}^T\boldsymbol{A} + \boldsymbol{u}^T = 0$$

$$\text{orthogonality conditions: } \begin{cases} \lambda_i[\boldsymbol{A}_i(\boldsymbol{x}) - b_i] = 0 \text{ for } i = 1 \ldots m \\ u_j x_j = 0 \text{ for } j = 1 \ldots n \end{cases}$$

$$\text{satisfaction of original constraints: } \begin{cases} \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ \boldsymbol{x} \geq \boldsymbol{0} \end{cases}$$

$$\text{Lagrange multiplier nonegativity: } \begin{cases} \boldsymbol{\lambda} \geq \boldsymbol{0} \\ \boldsymbol{u} \geq \boldsymbol{0} \end{cases}$$

As you can see, all of the conditions are linear *except* the orthogonality conditions, so it's *almost* linear, temptingly close to a possible solution via phase 1 linear programming where the variables are $\boldsymbol{x}$, $\boldsymbol{\lambda}$, and $\boldsymbol{u}$. Fortunately, there is a simple workaround to deal with the nonlinear constraints: adjust the LP pivoting rules so that we always satisfy the orthogonality conditions. We just make our pivots in a way that makes sure that (*i*) the slack variable for $\boldsymbol{A}_i(\boldsymbol{x}) \leq b_i$ is not basic at the same time as λ_i, and (*ii*) x_j is not basic at the same time as u_i. So we end up using something very much like linear programming to solve quadratic programs. In fact, most commercial LP solvers include the ability to solve quadratic programs since much of the main linear algebra machinery can be put to this purpose as well. However this LP-like approach does not apply to all QP forms: see below.

As is usual for the KKT conditions, things are a little easier when there are equality constraints: the corresponding nonnegativity constraint on the Lagrange multiplier is omitted, as is the orthogonality condition. So if row constraint *i* is an equality, just omit $\lambda_i \geq 0$ and omit $\lambda_i\boldsymbol{A}_i(\boldsymbol{x}) = 0$.

It is straightforward to check the shape of the quadratic objective function to determine whether it is concave, convex, etc. This is because the Hessian matrix (of second partial derivatives) is just equal to ½$\boldsymbol{Q}$, which consists entirely of constants in a quadratic model. If it is negative definite or negative semidefinite, then the objective function is concave and so it is guaranteed that any point satisfying the KKT conditions is a global maximum point. This is the form of QP that can be solved by commercial LP solvers.

However things can be more complicated if ½$\boldsymbol{Q}$ is not negative definite or negative semidefinite. One approach is to make small modifications to $\boldsymbol{Q}$ which make it negative definite so that the solution method described above can still be applied. But that doesn't always work, in which case you are back to solving a general nonlinear programming problem, albeit with some special structure. Since QPs that have only equality constraints are simpler (no orthogonality constraints, no nonnegativity constraints on the row Lagrange multipliers) one common approach is *active set methods*. Just as for GRG these methods try to decide which inequality constraints should be active at a given point, and then treats those as equalities, creating a version of the original QP consisting solely of equality constraints that is consequently easier to solve. The difficulty arises in deciding which inequalities are active at a particular point.

In general, very good solvers are now commercially available for quadratic programs of all types, including quadratically constrained quadratic programs.