# Discrete Mathematical Modeling

Math 381 Course Notes
University of Washington

A. Greenbaum
Winter Quarter, 2006

©R. J. LeVeque

**Acknowledgments**: These notes are based on many quarters of Math 381 at the University of Washington. The original notes were compiled by Randy LeVeque in Applied Math at UW. Tim Chartier and Anne Greenbaum contributed significantly to the current version in 2001-2002. Jim Burke contributed a portion of the chapter on linear programming. We thank them for their efforts. Since this document is still evolving, we appreciate the input of our readers in improving the text and its content.

# Contents


**1 Introduction to Modeling** **1**

1.1 Lifeboats and life vests . . . . . 1

1.2 The goals of 381 . . . . . 3

**2 Graph Theory and Network Flows** **5**

2.1 Graphs and networks . . . . . 5

2.2 The traveling salesman problem . . . . . 6

2.2.1 Brute force search . . . . . 6

2.2.2 Counting the possibilities . . . . . 6

2.2.3 Enumerating all possibilities . . . . . 7

2.2.4 Branch and bound algorithms . . . . . 7

2.3 Complexity theory . . . . . 9

2.4 Shortest path problems . . . . . 9

2.4.1 Word ladders . . . . . 9

2.4.2 Enumerating all paths and a branch and bound algorithm . . . . . 10

2.4.3 Dijkstra's algorithm for shortest paths . . . . . 10

2.5 Minimum spanning trees . . . . . 11

2.5.1 Kruskal's algorithm . . . . . 11

2.5.2 Prim's algorithm . . . . . 12

2.5.3 Minimum spanning trees and the Euclidean TSP . . . . . 12

2.6 Eulerian closed chains . . . . . 13

2.7 $\mathcal{P}$, $\mathcal{NP}$, and NP-complete problems . . . . . 15

**3 Stochastic Processes** **17**

3.1 Basic Statistics . . . . . 17

3.2 Probability Distributions and Random Variables . . . . . 18

3.2.1 Expected Value . . . . . 20

3.3 The Central Limit Theorem . . . . . 20

3.4 Buffon's Needle . . . . . 21

**4 Monte Carlo Methods** **23**

4.1 A fish tank modeling problem . . . . . 23

4.2 Monte Carlo simulation . . . . . 24

4.2.1 Comparison of strategies . . . . . 27

4.3 A hybrid strategy . . . . . 28

4.4 Statistical analysis . . . . . 30

4.5 Fish tank simulation code . . . . . 31

4.6 Poisson processes . . . . . 33

4.7 Queuing theory . . . . . 37

**5 Markov Chains and Related Models** **43**
5.1 A Markov chain model of a queuing problem . . . 43
5.2 Review of eigenvalues and eigenvectors . . . 46
5.3 Convergence to steady state . . . 47
5.4 Other examples of Markov Chains . . . 49
5.4.1 Breakdown of machines . . . 49
5.4.2 Work schedules . . . 51
5.4.3 The fish tank inventory problem . . . 52
5.4.4 Card shuffling . . . 53
5.5 General properties of transition matrices . . . 54
5.6 Ecological models and Leslie Matrices . . . 56
5.6.1 Harvesting strategies . . . 58

**6 Linear Programming** **59**
6.1 Introduction . . . 59
6.1.1 What is optimization? . . . 59
6.1.2 An Example . . . 59
6.1.3 LPs in Standard Form . . . 63
6.2 A pig farming model . . . 66
6.2.1 Adding more constraints . . . 68
6.2.2 Solving the problem by enumeration . . . 68
6.2.3 Adding a third decision variable . . . 70
6.2.4 The simplex algorithm . . . 70
6.3 Complexity and the simplex algorithm . . . 73

**7 Integer Programming** **75**
7.1 The lifeboat problem . . . 75
7.2 Cargo plane loading . . . 77
7.3 Branch and bound algorithm . . . 78
7.4 Traveling salesman problem . . . 80
7.4.1 NP-completeness of integer programming . . . 81
7.5 IP in Recreational Mathematics . . . 81
7.5.1 An Example Puzzle . . . 82
7.5.2 Formulation of the IP . . . 82
7.6 Recasting an IP as a 0-1 LP . . . 83

# Chapter 1

# Introduction to Modeling

## 1.1 Lifeboats and life vests

Recently there was an article in the Seattle Times about Washington State Ferries and the fact that they do not carry enough lifeboats for all passengers. They do carry enough life vests to make up the difference. It would be better to carry more lifeboats since your chances of surviving are better in a lifeboat than in cold water. However, lifeboats take up much more space and so there is a tradeoff — the capacity of the ferries would be greatly reduced if they had to carry enough lifeboats for everyone.

Suppose the XYZ Boat-builders have come to us with the following problem. They are designing a new ferry boat and want to determine the number of lifeboats and life vests to equip it with. They give us the following information:

- Each vest holds 1 person and requires 0.05 $m^3$ of space to store.
- Each boat holds 20 people and requires 2.1 $m^3$ of space to store.
- We must have capacity for 1000 people in one or the other.
- The total space to be devoted to this equipment is at most 85 $m^3$.

How many of each should they install?

One might go through the following steps in determining a solution:

1. First see if it's possible to accommodate everyone in lifeboats, which would be best. This would require $(1000/20) \times 2.1 = 105m^3$ of space, so the answer is no, we must use some combination.

2. Make sure there's enough space for life vests for everyone, otherwise the problem is impossible: $1000 \times .05 = 50m^3$, so it would be possible to provide only life vests and have space left over. But for the best solution we'd like to use more of the available space and provide as many boats as possible.

3. Figure out how many boats and vests are used if we *exactly* use all the available space and provide *exactly* the necessary capacity. We can set this up as a mathematical problem. There are two unknowns

$$\begin{aligned} x_1 &= \text{number of vests} \\ x_2 &= \text{number of boats} \end{aligned}$$

and two equations to be satisfied, one saying that the total capacity must be 1000 people and the other saying that the total volume must be $85m^3$. This is a linear system of equations:

$$\begin{aligned} C_1x_1 + C_2x_2 &= C \\ V_1x_1 + V_2x_2 &= V \end{aligned} \tag{1.1}$$

where

$$\begin{aligned} C_1 &= 1 \quad \text{(capacity of each vest)} \\ C_2 &= 20 \quad \text{(capacity of each boat)} \\ C &= 1000 \quad \text{(total capacity needed)} \\ V_1 &= 0.05 \quad \text{(volume of each vest)} \\ V_2 &= 2.1 \quad \text{(volume of each boat)} \\ V &= 85 \quad \text{(total volume available)} \end{aligned}$$

4. Solve the mathematical problem. Equation (1.1) is just a $2 \times 2$ linear system so it's easy to solve, obtaining

$$\begin{aligned} x_1 &= 363.6364 \quad \text{(number of vests)} \\ x_2 &= 31.8182 \quad \text{(number of boats)} \end{aligned}$$

5. Interpret the mathematical solution in terms of the real problem. Clearly we can't actually provide fractional vests or boats, so we'll have to fix up these numbers. The only way to do this and keep the volume from increasing is to reduce the number of boats to 31 and increase the number of vests to keep the capacity at 1000. Since $31 \times 20 = 620$ this means we need 380 vests. The total volume required is $84.1m^3$.

6. Present the answer to the XYZ Company. At which point they say: "You must be crazy. Everybody knows that the lifeboats are arranged on symmetric racks on the two sides of the boat, and so there must be an even number of lifeboats". Fix up the solution using this new information. There must be 30 boats and 400 vests.

7. Reconsider the solution to see if we've missed anything. Notice that the original mathematical solution had $x_2$ almost equal to 32. This suggests that if we had just a little more space ($85.2m^3$), we could increase the number of boats to 32 instead of 30, a substantial improvement. Suggest this to XYZ. They respond: "Well sure, the $85m^3$figure was just an approximation. We can easily find another $0.2m^3$if it means we can fit in two more boats.

This simple example illustrates some of the issues involved in mathematical modeling:

- The problem is usually not presented as a standard mathematical problem, but must be worked into this form.

- There may be many different ways to model the same problem mathematically. In the above example we reduced it to a linear system. But for more complicated problems of this same type this would not be possible and we would have to deal directly with the fact that we are given *inequality constraints* (e.g., the total volume must be *no more than* $85m^3$) rather than equalities. The problem is better set up as a *linear programming problem*, a type of discrete optimization problem in which we try to maximize the number of boats subject to two sets of constraints (an upper bound on volume and a lower bound on capacity). Actually it is an *integer programming problem* since we also have the constraint that the solution must be a pair of integers. There are standard methods for solving these types of problems.

- Once the "mathematical problem" has been solved (e.g., the linear system above), we need to interpret the answer to this problem in terms of the original problem. Does it make sense?
- Often the information presented is not complete or necessarily accurate. Here, for example, we weren't told to begin with the information that the number of boats had to be even, or that the available volume was only approximate.
- Often the problem is far too hard to model in detail and solve exactly. This example was quite straightforward in this regard, but typically one must make decisions about what aspects to consider and which can be safely ignored (or must be ignored to get a solvable problem). Often there is no "right answer", though some answers are much better than others!

## 1.2 The goals of 381

The goals of Math 381 are the following:

- Learn about the modeling process. How does one take an ill-defined "real world problem" and reduce it to a mathematical problem that can be solved by standard techniques? What issues are involved in dealing with real data and complex problems, where we can often only hope to model some small piece of the total problem?
- In order to have any chance of making a model that reduces to a "standard mathematical problem", we must learn what some of these standard problems are. For example, in the above problem we recognized early on that we could set it up as a linear system of equations, and we knew that this would be a good thing to do since there are standard methods for solving such systems (such as Gaussian elimination). We need some more tools in our tool bag in order to do modeling. We will see a variety of standard problems and how each can arise in many different contexts. These will be introduced in the course of looking at a number of modeling problems.
- Math 381 concentrates on *discrete modeling*, which means the mathematical problems typically involve a discrete set of values or objects (e.g. the life vests and lifeboats). Moreover there are typically a discrete set of possibilities to choose between. Trying out all of the possibilities to see which is best might be one possible approach, but is typically very inefficient. We want to use mathematical techniques to make the job easier.

  For example, it is clear that the solution to the problem above must consist of at most 1000 vests (since this would accommodate everyone) and at most 40 lifeboats (since this would use all the available space). One could try all possible combinations and find the right mix by trial and error, but as mathematicians we recognize that it is much easier to simply solve a $2 \times 2$ linear system.

  There are many standard mathematical problems and techniques which are useful in discrete modeling. Some of these are:

  - Linear programming and integer programming,
  - Combinatorics and combinatorial optimization,
  - Graph theory,
  - Network flows and network optimization,
  - Dynamic programming,
  - Queuing theory,
  - Stochastic processes and Markov chains,
  - Monte Carlo simulation,
  - Difference equations and recurrence relations,

  - Discrete dynamical systems,
  - Game theory,
  - Decision analysis,
  - Time series analysis and forecasting.

  Many modeling problems are better solved with continuous models, in which case different mathematical techniques are used, for example *differential equations*. Such models are studied in AMath 383.

- The main emphasis of this course will be on the modeling process, taking a problem and setting it up as a mathematical problem, and interpreting the results of the model. Along the way we also will learn something about techniques for solving some of the standard problems listed above, e.g., solving a linear programming problem by the "Simplex Method". This is necessary in order to solve the modeling problem, but the study of methods for solving these problems is not the main emphasis of this course.

  There are other courses which concentrate on understanding the mathematical aspects of these problems and the methods which have been developed to solve them. Some of these are:

  - Math 461: Combinatorics
  - Math 462: Graph Theory
  - Math 407: Linear optimization
  - Math 408: Nonlinear optimization
  - Math 409: Discrete optimization
  - Math 491, 492: Stochastic processes
  - CSE 373: Data structures and algorithms
  - CSE 417: Algorithms and Complexity
  - Industrial Engineering 324,5,6: Operations Research

  This course will provide an introduction to many of these topics which might motivate you to take some of these other courses and pursue the mathematics in more depth.

  There are also other related courses in other departments, including Industrial Engineering and Management Science. See the description of the Operations Research Option in the ACMS B.S. Degree Program for a list of some of these:
  `http://www.ms.washington.edu/acms`

- Another goal of the course is to give you experience in working together with others on a mathematical problem. Discussion will be encouraged in class and collaboration is encouraged outside of class. The projects must be done in groups. This is the way most modeling is done in the real world: by teams of mathematicians, or more commonly by a mathematician working together with scientists, engineers, or specialists in another discipline.

- The project also must be written up in an appropriate manner. This is an important part of any real-world modeling project and an important skill to master. During the quarter you also will read some papers written by others where modeling problems are presented and do some shorter writing assignments.

# Chapter 2

# Graph Theory and Network Flows

## 2.1 Graphs and networks

Imagine you are planning a trip around the United States based on a map showing distances between various cities and points of interest. The map has a labeled point for each city and lines connecting nearby cities are labeled by the distance between the two cities. One might ask a question like "How should we drive if we want to visit every city on the map?" This is an example of a *network flow* problem from *graph theory*. More generally, a *graph* is a collection of *vertices* or *nodes* which are connected by *edges*.

A graph is *connected* if there is a way to get from any vertex to any other vertex by following edges. Figure 2.1 shows a few types of (connected) graphs that arise in many different applications:

- A directed graph (or *digraph*) has arrows on edges indicating what direction one can move between nodes.
- A *network* has weights on the edges, often indicating the cost of moving from one vertex to the neighbor, e.g. the distance or driving time in the map example. A network could also be directed, for example if some routes on the map are one-way streets.
- A *tree* is a connected graph which contains no *circuits*. A family tree is one familiar example.

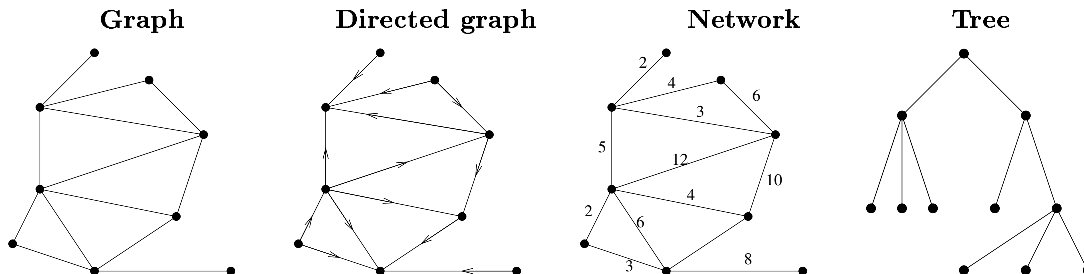



Figure 2.1: Some common graphs.

The vertices and edges of a graph might represent any number of different things, depending on the application. For example, the vertices in a digraph might represent legal positions in a game, with an arrow from Vertex A to Vertex B if you can go from Position A to Position B by a single legal move. Or the vertices could represent individuals in a hierarchical organization, with an edge from A to B if Person A reports to Person B. These and other examples are discussed in [Rob76].

## 2.2 The traveling salesman problem

A famous network flow problem is the *traveling salesman problem* (or TSP for short). This name comes from one particular application, but the same mathematical problem arises in many contexts.

GENERAL FORMULATION OF TSP: Given a network with $N + 1$ vertices, having an edge between every pair of vertices, what is the shortest path through the network which starts at a particular vertex, visits every other node exactly once, and returns to the original vertex?

Studying the TSP gives a good introduction to the mathematical theory of combinatorics and complexity of algorithms. Consider the TSP for a salesman who wants to start in Seattle, visit 4 other cities, and end up back in Seattle. What is the optimal order in which to visit the other cities?

### 2.2.1 Brute force search

One way to solve this problem is to simply *enumerate* all the possible routes, determine the total length of each, and choose the shortest. This is called the "brute force approach" since we aren't using any sophisticated ideas to do an intelligent search for the best route, we're just doing the obvious thing. It may still take some thought, however, to figure out how to systematically enumerate and check each possibility without missing any routes.

### 2.2.2 Counting the possibilities

The first question is: how many possible routes are there? This will give us some guidance when designing an algorithm to check them all, and also will tell us something about how long it will take to do this brute force search. Counting questions like this are one aspect of the mathematical field of *combinatorics*.

To do a count for this 4-city TSP, note that we must choose one city to visit first and there are 4 possibilities. Once we've chosen that city, we must choose which city to visit second and there are only 3 possibilities since this must be a different city. Then we have to choose the city to visit third and there are only 2 unvisited cities left. Finally the fourth city is determined since there is only one city left. This is illustrated as a *decision tree* in Figure 2.2, where the four cities are labeled A, B, C, D. Starting from the city of origin O, we have 4 choices, then 3 choices, then 2 and 1. The total number of distinct routes is

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24.$$

Similarly, for the TSP with $N$ cities (actually $N + 1$ cities including Seattle), the number of possible routes we must check is

$$N! = N \cdot (N - 1) \cdot (N - 2) \cdots 3 \cdot 2 \cdot 1.$$

Note that what we have counted is the number of ways one can *permute* the $N$ cities. In general for any set of $N$ distinct objects there are $N!$ possible orderings, or *permutations* of the $N$ objects.

With only 4 cities we could quickly compute the traveling salesman's path length for all 24 permutations by hand and determine the shortest route. But unfortunately $N!$ grows very rapidly with $N$, e.g.,

$$\begin{aligned} 5! &= 120 \\ 10! &= 3,628,800 \\ 20! &= 2,432,902,008,176,640,000 \\ 50! &\approx 3 \times 10^{64}. \end{aligned}$$

For a TSP with 10 cities we would need to check about 3.6 million possibilities, still fairly easy on a computer. To get an idea of how long this would take, consider a 500 MHz PC. This is a computer which performs 500 million "instructions" or "cycles" every second. Here "instruction" means a low-level machine operation and it typically takes many cycles to perform one arithmetic operation such

as an addition or multiplication. These are called floating point operations since scientific notation is usually used to represent a large range of values on the computer. For computations of the sort mathematicians, engineers and scientists do, a useful measure of a computer's speed is the number of *floating point operations per second*, called flops. Machine speed is often measured in *megaflops* (millions of floating point operations per second). A 500 MHz machine might do about 40 megaflops. Supercomputer speeds are often measured in gigaflops (billions of flops) or teraflops (trillions of flops).

Returning to the traveling salesman problem, note that to compute the length of a single route requires about $N$ additions, adding up the length between each pair of cities on the route, as well as more work to keep track of which route to examine next, which route is shortest, etc. Let's not worry too much about the exact cost, but suppose that we are working on a machine that can examine ten million routes per second. Then solving the 10-city TSP would take about 1/3 of a second. Not bad.

But already with 20 cities it would be difficult to find the answer using this brute force approach. We now have about $2.4 \times 10^{18}$ routes to examine, and at a rate of 10 million per second this will take $2.4 \times 10^{11}$ seconds, which is about 7,715 years. Even if we move up to a supercomputer that is a thousand times faster, it would take almost 8 years. For a TSP with 50 cities there is no conceivable computer that could find the solution in our lifetimes, using the brute force approach.

Many simple problems suffer from this kind of *combinatorial explosion* of possible solutions as the problem size increases, which makes it impossible to simply enumerate and check every possibility. Nonetheless such problems are solved daily, using more sophisticated mathematical techniques.

### 2.2.3 Enumerating all possibilities

For an N-city TSP with $N$ sufficiently small, we could solve by enumeration. This requires some algorithm for systematically checking every possibility. Figure 2.2 shows one way to think about enumerating all the possible routes for the 4-city TSP. In the rightmost column is an enumeration of all 24 routes in the *lexicographic order*, so called because this is the order that would be obtained by putting the 24 strings representing different routes into alphabetical order. Note that the $3! = 6$ routes starting with A come first, then the 6 routes starting with B, and so on. Within each set, the routes are grouped by which city comes second, ordered from "lowest" to "highest" in lexicographic ordering.

### 2.2.4 Branch and bound algorithms

Instead of evaluating the cost of every path appearing in the tree of Figure 2.2, one might be able to rule out some paths early if their cost exceeds that of the best path discovered so far. Consider, for example, the network in Figure 2.3.

One might first try a *greedy* algorithm that takes the shortest edge to a new city at each step. This leads to the path O-ADCB-O, which has a total cost of 15. After discovering this, any path in Figure 2.2 that uses edge BD (or, equivalently, DB) can be eliminated, since the cost of that edge alone is 16. By continuing to check the other paths that start with O-A, one finds that the path O-ABCD-O has a cost of only 14. Examining next the paths that begin with O-D, and discarding any partial paths whose cost exceeds 14, one finds that O-DABC-O costs only 11, and this enables one to eliminate even more possibilities. Paths beginning with O-C and O-B still must be checked, but as soon as the cost of a partial path exceeds 11, that branch of the tree can be eliminated from further consideration. This type of algorithm is sometimes called a *branch and bound* algorithm. The key idea is to obtain some sort of *bound* on how good the shortest path within a given subtree could possibly be and, if a less costly path has already been identified, to *prune* away those branches of the tree, thereby reducing the required computation.

If we are unlucky then we might not be able to prune much of the tree and in the worst-case scenario we have to examine the entire tree anyway, which is no improvement in running time. However in practice a good branch and bound algorithm will typically be *much* faster than pure enumeration, and problems are solved this way which would otherwise be intractable.

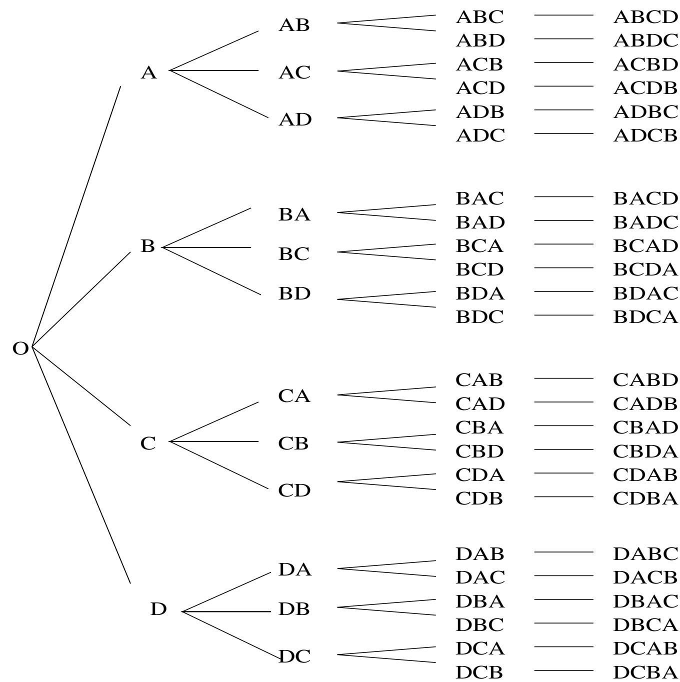



Figure 2.2: Decision tree for enumerating all possible routes in a traveling salesman problem starting from City O and visiting four other cities A, B, C, and D. The right column shows an enumeration of all permutations of the 4 cities, in lexicographic order.

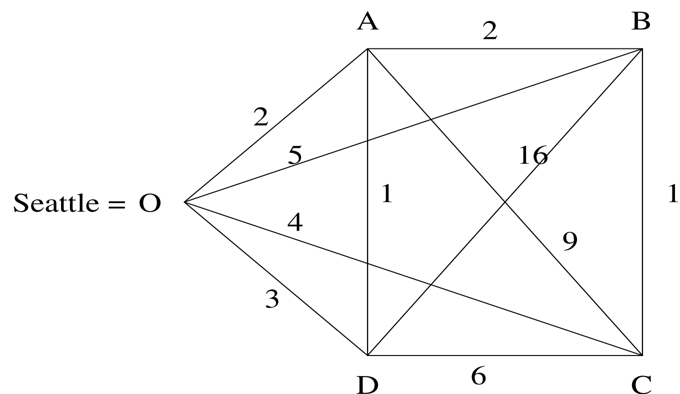



Figure 2.3: A traveling salesman network starting from Seattle and visiting four other cities A, B, C, and D.

Many other algorithms have been developed for the traveling salesman problem and for various special cases of this problem. An introduction to many of these can be found in [LLAS90].

## 2.3 Complexity theory

For a particular mathematical problem, say the traveling salesman problem, there is often a parameter $N$ which measures the "size" of the problem, e.g., the number of vertices in the network. If we have an algorithm for solving the problem, then we are interested in knowing how much work the algorithm requires as a function of $N$. In particular we want to know how fast this grows as $N$ increases. For some algorithms the amount of work required (or time required on a computer) is *linear* in $N$, meaning it is roughly $cN$ where $c$ is some constant. If we double the size of the problem then the amount of work required roughly doubles. We say that such an algorithm requires $O(N)$ work. The "big Oh" stands for "order" — it takes order $N$ work and we're usually not too concerned about exactly what the size of the constant $c$ is.

For some algorithms the work required increases much faster with $N$. An $O(N^2)$ algorithm requires work that is proportional to $N^2$, which means that doubling $N$ increases the work required by a factor of 4. If an algorithm requires $O(N^p)$ work for some $p$ then it is said to be a *polynomial time* algorithm. The larger $p$ is, the more rapidly the work grows with $N$. Here are some examples of polynomial time algorithms:

- Finding the largest of $N$ values. Checking the value of each item and keeping track of the largest value seen takes $O(N)$ operations.
- Solving a system of $N$ linear equations in $N$ unknowns using the Gaussian elimination algorithm takes $O(N^3)$ operations.
- If the linear system is upper triangular, then it only takes $O(N^2)$ operations using "back substitution".
- Dijkstra's algorithm for finding the shortest path between two vertices in a network (see below) requires $O(N^2)$ work in general.

For most problems we will discuss with polynomial time algorithms, the value of $p$ is typically fairly small, like 1, 2, or 3 in the examples above. Such problems can generally be solved fairly easily on a computer unless $N$ is extremely large.

There are other problems which are essentially much harder, however, and which apparently require an amount of work which grows *exponentially* with the size of $N$, like $O(2^N)$ or like $O(N!)$. The traveling salesman problem is in this category. While many algorithms have been developed which work very well in practice, there is no algorithm known which is guaranteed to solve all TSP's in less than exponential time. (See Section 2.7 for more about this.)

## 2.4 Shortest path problems

A class of problems that arise in many applications are network flow problems in which we wish to find the shortest path between two given vertices in a network. The problem of finding the shortest route on a map is one obvious example, but many other problems can be put into this form even if they don't involve "distance" in the usual sense.

### 2.4.1 Word ladders

A word ladder is a sequence of valid English words, each of which is identical to the previous word in all but one letter. It's fun to try to construct word ladders to go from some word to a very different word with as few intermediate words as possible. For example, we can make FLOUR into BREAD as follows:

```
FLOUR
FLOOR
FLOOD
BLOOD
BROOD
BROAD
BREAD
```

Given two English words with the same number of letters, how could we determine the shortest word ladder between them? (Assuming there is one, or determine that none exists.) This can be set up as a network flow problem.

Suppose we are considering $N$-letter words, then we could in principle construct a graph where the vertices consist of all $N$-letter words in the English language. We draw an edge from one vertex to another if those two words are simple mutations of each other in which only a single letter has been changed. A word ladder then corresponds to a path through this graph from one word to the other. In fact this graph (with nearly 6000 vertices) has been constructed for $N = 5$ by Don Knuth and is available on the web (see the pointer from the class webpage). Word ladders and algorithms for manipulating these graphs are discussed at length in [Knu93], which describes a general platform for combinatorial computing. Finding the shortest word ladder is equivalent to the shortest-path problem through a network if we assign every edge in the graph the weight 1.

Given a network and two specific vertices, how do we find the shortest path between them? For small networks this is easy to do by trial and error, particularly if the graph is drawn in such a way that the physical distance between the vertices roughly corresponds to the weight of the edge, as is the case for a map. But the weights may have nothing to do with physical distance and at any rate if there are thousands or millions of vertices and edges it may be impossible to even draw the graph.

### 2.4.2 Enumerating all paths and a branch and bound algorithm

One approach to solving a shortest-path problem would be to enumerate all possible paths between the start and end vertices, compute the length of each path, and choose the shortest. To enumerate all paths, we could build a "decision tree" similar to what was illustrated in Figure 2.2 for the TSP. The problem with this approach is that for a graph with $N$ vertices, there may be very many possible paths to choose between. How many? In the worst case, there would be an edge connecting each vertex with each of the $N - 1$ other vertices. In this case we could visit any or all of the other vertices on the way between the start and finish, and the number of possible paths would be exponential in $N$, just as for the TSP.

The set of all possible paths could be searched more efficiently using a branch and bound algorithm, similar to that described for the TSP. However, for this problem there turn out to be much more efficient algorithms which are guaranteed to give the answer with relatively little work.

### 2.4.3 Dijkstra's algorithm for shortest paths

*Dijkstra's algorithm* is a systematic procedure for finding the shortest path between any two vertices. Actually this algorithm solves the problem by embedding it into what appears to be a harder problem, that of finding the shortest route to all other vertices from a given vertex. The set of all shortest routes is built up in $N$ steps, by finding the closest node in Step 1, the next closest node in Step 2, and so on. By the end we know the best route to every other node and in particular to the destination node.

Each step requires examining the distance from the node that was "solved" in the previous step to all unsolved nodes which can be reached from this node. This requires examining at most $N - 1$ edges, so the work required is $O(N)$ in each of the $N$ steps, and hence the total work is $O(N^2)$.

To implement it on a computer for a large network requires some suitable *data structure* for storing the network with information about which vertices are connected to one another and the weight of each edge. One way to represent a network or directed network on a computer is with a matrix. The rows

and columns correspond to the nodes of the network and the $(i, j)$ entry of the matrix is the weight on the edge from node $i$ to node $j$ (or, if there is no edge from $i$ to $j$, that entry can be set to $\infty$). A matrix representation of the network in Figure 2.3 is:

| | $O$ | $A$ | $B$ | $C$ | $D$ |
|---|---|---|---|---|---|
| $O$ | 0 | 2 | 5 | 4 | 3 |
| $A$ | 2 | 0 | 2 | 9 | 1 |
| $B$ | 5 | 2 | 0 | 1 | 16 |
| $C$ | 4 | 9 | 1 | 0 | 6 |
| $D$ | 3 | 1 | 16 | 6 | 0 |

We'll go through some simple examples of Dijkstra's algorithm in class. It is described in many books, e.g., [HL95] and [Rob84]. A writing assignment will be to produce your own description of this algorithm.

## 2.5 Minimum spanning trees

Another type of problem that arises in many applications is to connect a collection of vertices at the lowest possible cost. Examples include highway construction and telephone line installation: the vertices represent cities and the roads or phone lines correspond to edges constructed so that there is some path between every pair of vertices. In graph theoretic terms, this is accomplished via a *minimal cost spanning tree*.

Recall that a *tree* is a graph that is connected and has no circuits. If a group of vertices is connected by edges that do contain a circuit, then it is not hard to see that one (or more) of the edges can be removed without destroying the connectedness of the graph. This is illustrated in Figure 2.4. Since each edge has a nonnegative cost associated with it, clearly the least expensive way to connect the vertices will be via a graph with no circuits (i.e., a spanning tree).

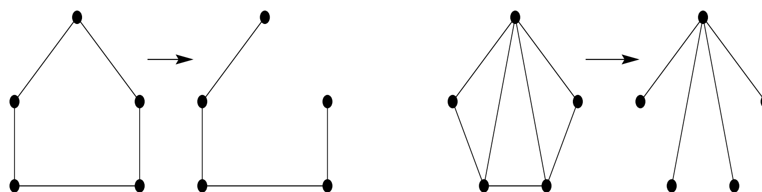


Figure 2.4: Graphs with circuits and corresponding spanning trees

In a spanning tree, the number of edges is always one less than the number of vertices: $e = N - 1$. Each graph in Figure 2.4 has $N = 5$ vertices, and the two trees have $e = 4$ edges. There are two well-known algorithms for finding a minimal cost spanning tree in a connected network: Kruskal's algorithm and Prim's algorithm.

### 2.5.1 Kruskal's algorithm

Kruskal's algorithm begins by sorting the edges of the graph in order of increasing cost. For the network shown in Figure 2.5 this order is: $\{AB\}(8)$, $\{CD\}(10)$, $\{CE\}(12)$, $\{DE\}(22)$, $\{BC\}(53)$, $\{BE\}(54)$, $\{AD\}(63)$, and $\{AE\}(70)$. Each edge is then considered in turn and, if it does not form a circuit with edges already in the good set, it is added to the good set. This is continued until the good set has $N-1$ edges. Since the good set has no circuits, it is a tree. If all edges have been examined and the good set still does not have $N - 1$ edges, it means that the original graph must not have been connected, and this fact can be reported to the user. The first edges added to the tree from Figure 2.5 are $\{AB\}$,

$\{CD\}$, and $\{CE\}$. Edge $\{DE\}$ is then examined, but it is found to form a circuit with edges already in the tree, so this edge is skipped and edge $\{BC\}$ is added to the tree. Now the tree has $N - 1 = 4$ edges, so the algorithm is finished. The edges of the minimum spanning tree are: $\{AB\}$, $\{CD\}$, $\{CE\}$, and $\{BC\}$.

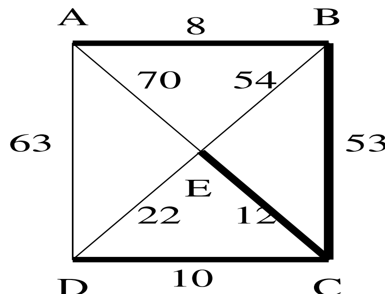



Figure 2.5: Minimum spanning tree

### 2.5.2 Prim's algorithm

Prim's algorithm for finding a minimal cost spanning tree avoids the initial sorting phase of Kruskal's algorithm. The two algorithms generate the same result, however, when the minimal cost spanning tree is unique.

Prim's algorithm begins by choosing any vertex from the graph and including it in the tree $T$. For the graph of Figure 2.5, we could choose, for instance, vertex $D$. It then finds that edge in the graph joining a vertex in $T$ to one not in $T$ which has the lowest cost: $\{CD\}$ in our example. This step is repeated until the tree has $N - 1$ edges or until there are no more edges joining vertices of $T$ to vertices not in $T$. In the latter case, it reports that the graph is not connected. In our example problem, the next edge to be added is $\{CE\}$. We then search for the lowest cost edge joining $C$, $D$, or $E$ to one of the remaining nodes, and add edge $\{BC\}$ to the tree. Finally the lowest cost edge between either $B$, $C$, $D$, or $E$ and $A$ is added, namely, $\{AB\}$.

*Exercise:* Try using Prim's algorithm with different starting vertices. Convince yourself that it generates the same minimal cost spanning tree (or one with the same cost, if there is more than one minimal cost spanning tree), no matter where you start. Do you see why Kruskal's algorithm and Prim's algorithm generate the same result?

Both Kruskal's algorithm and Prim's algorithm are polynomial time algorithms. The best implementations of these algorithms run in $O(E \log_2 N)$ time, where $N$ is the number of vertices and $E$ the number of edges in the original graph. The number of edges in a graph is at most $\binom{N}{2} = \frac{N(N-1)}{2}$, so the work required is $O(N^2 \log_2 N)$.

### 2.5.3 Minimum spanning trees and the Euclidean TSP

Often the weights on the edges of a network represent actual distances between cities or other destination points. In this case, the weight or cost of going from vertex $A$ to vertex $B$ is the same as that of going from $B$ to $A$, and the costs also satisfy the *triangle inequality*:

$$\text{cost}(AB) + \text{cost}(BC) \geq \text{cost}(AC).$$

When the weights on a network for the traveling salesman problem represent actual distances between points in a plane, the problem is called a *Euclidean TSP*.

While there is no known polynomial time algorithm for solving the Euclidean TSP, there is an easy way to find a route whose cost is within a factor of two of the optimal: Construct a minimum spanning tree. Starting at the origin node, move along edges of the minimum spanning tree until you can go no further. Then backtrack along these edges until you reach a point from which you can traverse a new edge to visit a new node or until all of the nodes have been visited and you can return to the origin. This is illustrated in Figure 2.6, where the minimum spanning tree is marked and the suggested path is ODOABCBAO. The total cost of this path is twice the cost of the minimum spanning tree. Another possible path is OABCBAODO, which also traverses each edge of the minimum spanning tree twice.

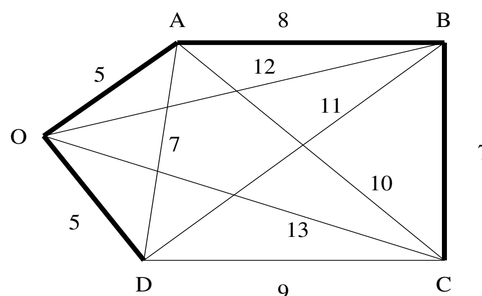



Figure 2.6: Euclidean TSP network with minimum spanning tree. Path OD-O-ABC-BA-O costs twice as much as the minimum spanning tree, and tour ODABCO is within a factor of two of optimal.

The path constructed in this way is not a legitimate tour for the TSP, since it visits some nodes more than once. But because the weights satisfy the triangle inequality, any path between nodes that visits other nodes along the way that have already been visited can be replaced by a direct route between the given nodes, and the cost will be no greater. Thus one arrives at a tour whose cost is at most twice that of the minimum spanning tree. For the example in Figure 2.6, this tour is ODABCO and has a total cost of 40. Had we chosen the other path, OABCBAODO, it would have been replaced by the tour OABCDO, which has a total cost of 34 (and turns out to be the optimal tour for this problem).

Since the minimal cost spanning tree is the lowest cost set of edges that connects all of the vertices, any traveling salesman tour that visits all of the vertices must cost at least as much as the minimal cost spanning tree: 25 for our example. This gives a *lower bound* on the cost of the optimal tour, while the tour derived above, whose cost is within a factor of two of this value, provides an *upper bound.* This information can be used within a branch and bound algorithm to hone in on the actual optimal tour.

## 2.6 Eulerian closed chains

Still another type of graph thoeretic problem that arises in many applications is that of finding an *Eulerian closed chain*: A path that starts at a given node, traverses each *edge* of the graph exactly once, and returns to the starting point. This problem might arise, for example, in scheduling snowplows or street sweepers. If the edges of the graph represent streets, then the object is to plow or sweep each street once but, if possible, not to backtrack over streets that have already been serviced.

This type of problem is also popular in many mathematical games. In fact, it is said that graph theory was invented by Euler in the process of settling the famous Königsberg bridge problem. The situation is pictured in Figure 2.7. There are seven bridges and the question is whether one can start at a particular point, cross each bridge exactly once, and return to the starting point. If you try it for a while, you can probably convince yourself that the answer is "No".

To see this in a more formal way, let the land masses in the problem be represented by nodes and the bridges by edges in a *multigraph* (a graph that can have more than one edge between a pair of nodes). This multigraph is shown in Figure 2.8. Note that if we start at a particular node in the multigraph and travel along one edge to a different node, then we must leave that node along a different edge. If there is only one edge through the node then this will not be possible. Moreover, if there are three or five or

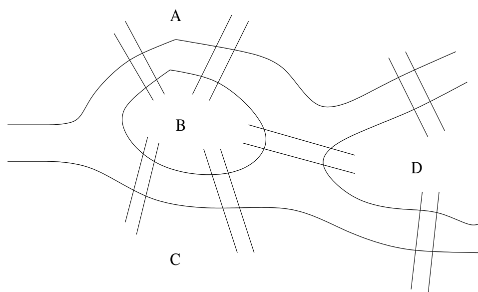



Figure 2.7: The bridges of Königsberg

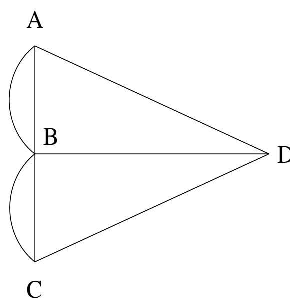



Figure 2.8: Multigraph representing the bridges of Königsberg

any odd number of edges through that node, then in order to traverse all of these edges, we will have to enter then leave the node until there is only one unused edge remaining. But at that point we are stuck, because we must enter the node in order to use that edge, but then we cannot leave. The same holds for the starting node. If there is only one edge through it, then we can leave but never return, and if there are an odd number of edges through it, then we can leave and return a certain number of times but eventually we will be at the starting node with one unused edge remaining to traverse.

The *degree* of a vertex is defined as the number of edges through that vertex. Using the above sort of reasoning Euler proved the following:

**Theorem (Euler[1736]).** A multigraph $G$ has an Eulerian closed chain if and only if it is connected up to isolated vertices and every vertex of $G$ has even degree.

The condition that the graph be connected up to isolated vertices just means that there is no edge that cannot be reached at all.

There is no Eulerian closed chain in Figure 2.8 since, for example, vertex $D$ has degree 3. (In fact, all of the vertices in this multigraph have odd degree.)

A good way to establish the sufficiency of Euler's theorem is by describing an algorithm for computing an Eulerian closed chain in an even connected multigraph. Consider, for example, the multigraph in Figure 2.9. Start at any vertex and travel to other vertices along unused edges until you can go no further. For example, we might start at $A$, go to $B$, $C$, $D$, back to $B$, and then to $A$. For each of the vertices visited we use an even number of edges — one to enter and one to leave — each time we visit the vertex, so that the number of unused edges through each vertex remains even. Moreover, once we leave the initial vertex, it is the only one with an odd number of unused edges through it; for every other vertex, if we enter we can also leave. Therefore our path can end only back at the original vertex. At this point, we may not have traversed all of the edges of the multigraph, so go to any vertex that has some unused edges through it: vertex $C$, for example, and repeat. Take one of the unused edges to a new vertex and again continue until you can go no further. For example, we might go from $C$ to $E$ to $F$ to $G$ to $E$ to $C$. This path can be combined with our previous one to form the chain:

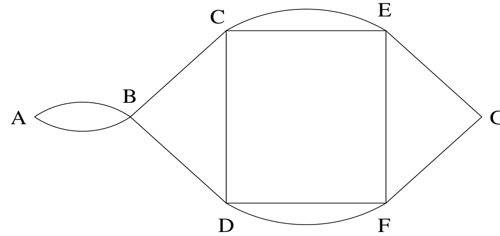



Figure 2.9: Multigraph with an Eulerian closed chain

$ABC - EFGEC - DBA$. There are still two unused edges in the multigraph, so we repeat by going from $D$ to $F$ and back to $D$. Combining this with the previous chain gives an Eulerian closed chain: $ABCEFGECD - FD - BA$.

The amount of work required to find an Eulerian closed chain is proportional to the number of edges in the multigraph.

## 2.7 $\mathcal{P}$, $\mathcal{NP}$, and NP-complete problems

We have seen that some problems, such as the shortest path problem, the minimum spanning tree problem, and the Eulerian closed chain problem, can be solved in a number of steps that is polynomial in the size of the problem. Such problems are in the class $\mathcal{P}$, the set of all problems that can be solved in P-time.

Some problems, such as the traveling salesman problem, *might* lie in the class $\mathcal{P}$ but probably don't. If so, nobody has yet found the P-time algorithm. This problem does lie in a larger class called NP, however. NP stands for *nondeterministic polynomial*, and the exact definition is a bit complicated. The basic idea is that for a problem in this class it is at least possible to check one possible solution in polynomial time, *e.g.*, for the TSP it is possible to compute the length of any one possible path with only $O(N)$ operations. For such problems there is at least a chance one could find a P-time algorithm to find the best path, if we could search efficiently enough.

A problem which is not in $\mathcal{NP}$ is more clearly hopeless. For example, suppose we consider a variant of the TSP in which we specify a number $B$ and we wish to find *all* tours on a given set of $N$ cities which have total length $B$ or less. Then it is possible to construct examples in which there are exponentially many paths in the solution set, so even writing down the solution if we knew it would take exponentially long and clearly there's no hope of a P-time algorithm to solve the problem.

Note that any problem in P is also in $\mathcal{NP}$, so $\mathcal{P}$ is a subset of $\mathcal{NP}$. Is it a proper subset? In other words, are there problems in $\mathcal{NP}$ which are not in $\mathcal{P}$? Or does $\mathcal{P} = \mathcal{NP}$? This is one of the great unsolved problems of theoretical computer science.

Some progress was made toward this problem when Karp [Kar72] showed that many hard problems lie in a set of problems within $\mathcal{NP}$ called the *NP-complete problems*, which have the following property: if any one of them can be solved in P-time, then *every* problem in $\mathcal{NP}$ can be solved in P-time and so in fact $\mathcal{P} = \mathcal{NP}$. The TSP is NP-complete, and so are many other famous and easy-to-state problems, such as the Hamiltonian circuit problem and the integer programming problem. For more discussion of such problems, see for example [GJ79], [Sip97].

# Chapter 3

# Stochastic Processes

So far we have studied deterministic processes, *i.e.* situations where the outcome is completely determined by our decisions. In real life however, there are many situations which are out of our control or where things happen with a certain probability. These situations are called *stochastic processes* or *random processes* or *probabilistic processes*. For example, the amount of time we have to wait for the next bus to arrive is a stochastic process. Also, the number of salmon returning to spawn in the UW Hatchery each year, the number of games won by the Husky volleyball team, and the daily Dow Jones Industrial Average are other examples.

A thorough understanding of statistical modeling is beyond the scope of this course. Still, your modeling capabilities can be enhanced by a brief introduction to some fundamental concepts from probability and statistics including the mean, median and variance of a probability distribution, random variables and the Central Limit Theorem.

## 3.1 Basic Statistics

In general terms, a *statistic* is a number that captures important information about a data set. If we capture the right information, we can use the statistic to predict future outcomes. The primary example of a statistic is the average or mean. For example, if we learn that bus number 43 comes every 10 minutes on average during rush hour, then it helps us predict what time we will arrive at our destination.

For the data set $X = \{x_i : i = 1, 2, \ldots, n\}$, the *mean* of $X$, denoted $\mu$ or $E(X)$, is defined as

$$\mu = \frac{\sum_{i=1}^{n} x_i}{n}.$$

To find the *median*, order the values $x_i$ in ascending (or descending order). If $n$ is odd, then the median equals $x_m$ where $m = \frac{n+1}{2}$. Otherwise, the median equals the average of $x_m$ and $x_{m+1}$ where $m = \frac{n}{2}$.

To analyze the behavior of averages, consider the following data set. Let $X$ be the number of minutes we wait for the bus on the first 51 days of class.

| | | | | | |
|---|---|---|---|---|---|
| 3.759 | 1.939 | 6.273 | 7.165 | 1.146 | 5.973 |
| 0.099 | 9.048 | 6.991 | 5.113 | 6.649 | |
| 4.199 | 5.692 | 3.972 | 7.764 | 3.654 | |
| 7.537 | 6.318 | 4.136 | 4.893 | 1.400 | |
| 7.939 | 2.344 | 6.552 | 1.859 | 5.668 | |
| 9.200 | 5.488 | 8.376 | 7.006 | 8.230 | |
| 8.447 | 9.316 | 3.716 | 9.827 | 6.739 | |
| 3.678 | 3.352 | 4.253 | 8.066 | 9.994 | |
| 6.208 | 6.555 | 5.947 | 7.036 | 9.616 | |

```
7.313    3.919    5.657    4.850    0.589
```

To aid in calculating the median, the following sorted list is also included.

```
0.099    3.716    5.488    6.555    7.939    9.994
0.589    3.759    5.657    6.649    8.066
1.146    3.919    5.668    6.739    8.230
1.400    3.972    5.692    6.991    8.376
1.859    4.136    5.947    7.006    8.447
1.939    4.199    5.973    7.036    9.048
2.344    4.253    6.208    7.165    9.200
3.352    4.850    6.273    7.313    9.316
3.654    4.893    6.318    7.537    9.616
3.678    5.113    6.552    7.764    9.827
```

The mean and median of $X$ equal 5.715 and 5.973, respectively. Consider changing the largest value of $X$ from to 9.994 to 100. Now, the mean equals 7.483. Such a change in the mean is expected given the large change in a value of the corresponding data set. On the other hand, the median remains unchanged. Our change to $X$ altered only the value of the largest element. Therefore, the value of the middle element, which equals the median, remained unchanged.

Suppose instead, we had changed the largest value of $X$ from 0.9994 to another random number between 0 and 1, say 0.5484. Now, the mean and median equal 0.5626 and 0.5947, respectively. In such a case, the mean still changes, albeit by a small amount. Take a moment and reflect on why the median changes in this case. Larger data sets (say, of size 1,000 or 10,000) exhibit similar behavior although the change in the mean is seen in less significant digits. (The median also changed to 0.5819, can you see why? Check your understanding of these important concepts by verifying for yourself that this should occur.)

Another important statistic is the *variance* of the data set. The variance gives us information about how close the data is centered around the mean $\mu$. The variance of the data set $X = \{x_i : i = 1, 2, \ldots, n\}$ is defined as [1]

$$\mathrm{Var}(X) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu)^2. \tag{3.1}$$

The variance for the original data set above is 6.284. This tells us we would probably never have to wait for the bus for more than 12 minutes. On the other hand, replacing 9.994 by 100 we get a variance of 180.

## 3.2 Probability Distributions and Random Variables

In order to model stochastic processes, we first need a mathematical notation that can describe a wide variety of situations. We will in introduce the concepts of a probability distribution and a random variable through examples.

Let $Y$ denote any one of the six possible outcomes that can be observed when a fair die is rolled. Since a fair die is equally likely to roll any of the values from 1 to 6, there is a $1/6^{\text{th}}$ chance of rolling a given number. We will write this as $P(Y = y) = \frac{1}{6}$ for $y = 1, 2, \ldots, 6$. Equivalently, we are defining a *probability distribution* function $p(y) = 1/6$ for $y = 1, 2, \ldots, 6$. Such a probability distribution can be written as

| $y$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $p(y)$ | 1/6 | 1/6 | 1/6 | 1/6 | 1/6 | 1/6 |

[1] Some authors defined the sample variance to be $\mathrm{Var}(X) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$ instead.

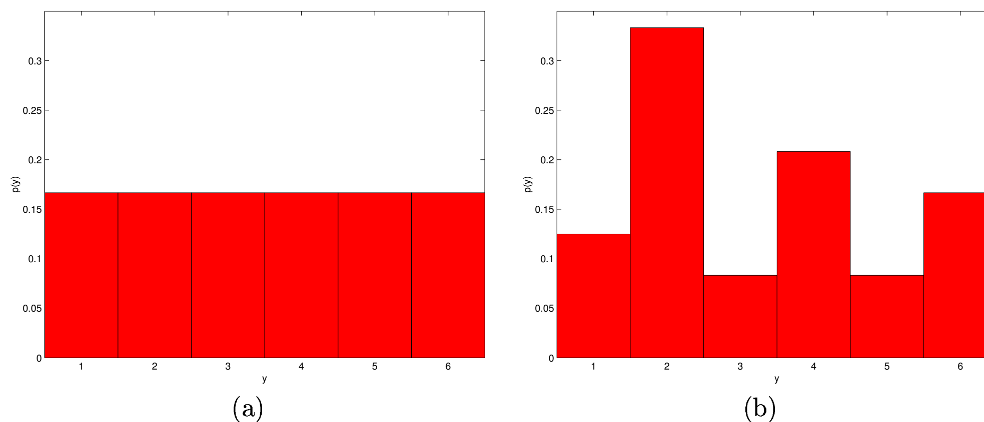



Figure 3.1: Probability histograms (a) for a uniform distribution and (b) a nonuniform distribution.

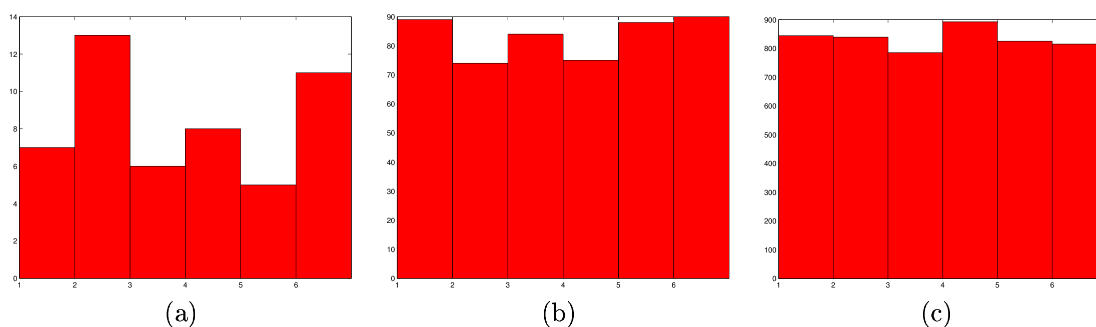



Figure 3.2: Histograms for tossing a fair die 50, 500 and 5000 times.

In this example, $Y$ is a *random variable*; it is a variable that we cannot solve for explicitly but instead one that takes on a different fixed value each time we run a trial by rolling the die. Random variables are very useful in probability because we can manipulate them just like ordinary variables in equations. For example, one could write $P(2 < Y \leq 5)$ to mean the probability that a roll is among the numbers $\{3, 4, 5\}$. A random variable $Y$ is said to be *discrete* if it can assume only a finite or countably infinite number of distinct values. Every discrete random variable comes with an associated probability distribution function, $P(Y = y) = p(y)$.

An experiment where every value is equally likely to occur is said to have a *uniform distribution* as its underlying probability distribution. Figure 3.1 (a) depicts a probability histogram for a uniform distribution. In contrast, Figure 3.1 (b) is an example of a probability histogram for a nonuniform distribution.

Another way to recognize a uniform distribution is by noting the probability distribution has the form

$$p(y) = \frac{1}{\text{total \# outcomes}}.$$

Therefore, this type of probability is closely related to the study of counting methods known as *combinatorics.*

Clearly, a probability histogram for $Y$ should exhibit such a uniform distribution. Yet, each roll of the die is independent from the previous rolls. In other words, the die has no memory of its last roll. As any gambler will attest, the uniform distribution does not imply that a streak of 4's is impossible.

To see this graphically, histograms of 50, 500, and 5000 such experiments are depicted in Figure

3.2 (a), (b) and (c), respectively. As the number of experiments increases, the histograms converge toward the corresponding uniform distribution. Note that even 10,000 experiments will not produce a histogram that exactly replicates the distribution in Figure 3.1 (a). Seen from the perspective of a gambler, indeed, guessing the value of a single roll of a fair die is mere chance even with knowledge of a previous outcome. Yet, from the perspective of a casino, the underlying distribution can place the odds heavily in the "house's" favor over many such experiments.

There are many additional probability distributions. The modeler's challenge is to choose the one that most closely approximates the data.

### 3.2.1 Expected Value

The *expected value* $E$ of a discrete random variable $Y$ with probability function $p(y)$ is defined to be

$$E(Y) = \sum_y yp(y).$$

Recall that the probability function for the rolling of a fair die is $p(y) = 1/6$ for $y = 1, 2, \ldots, 6$. Therefore, $E(Y)$ equals $\sum_{i=1}^{6} i/6 = 3.5$.

## 3.3 The Central Limit Theorem

Let $Y_1, Y_2, \ldots, Y_n$ be independent and identically distributed random variables. Then, $\overline{Y} = (1/n)(Y_1 + Y_2 + \ldots + Y_n)$ is the random variable measuring the mean over $n$ trials. What can be said about the distribution of $\overline{Y}$? The Central Limit Theorem answers this question.

**Theorem 3.3.1 (Central Limit Theorem)** *Let $Y_1, Y_2, \ldots, Y_n$ be independent and identically distributed random variables. Then the sample average $\overline{Y}$ tends to a normal distribution as the sample size tends to infinity. The mean of this distribution will be the mean of the population and the variance of this distribution will be the variance of the population divided by the sample size. (This all holds providing the mean and variance of the population is finite and there is random sampling).*

Take a moment and consider the power of this theorem. The theorem allows *any* population. In fact, two populations may have widely differing underlying distributions such as the uniform and nonuniform probability distributions in Figure 3.1. Yet, if multiple, independent experiments occur for either population, the resulting distribution of the sample average tends toward a normal distribution as sample size increases.

Let $Y$ and $X$ denote any one of the six possible outcomes available from tossing a fair and weighted die, respectively. The probability function $p(y) = 1/6$ for $y = 1, 2, \ldots, 6$. The probability distribution for $X$ is:

| $x$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $p(x)$ | 1/8 | 1/3 | 1/12 | 5/24 | 1/12 | 1/6 |

Figure 3.1 (a) and (b) are the probability histograms for $Y$ and $X$, respectively. Suppose we throw the fair die 50 times and record the sample average of the 50 outcomes. Repeating this 1000 times and plotting the results produces the histogram in Figure 3.3 (a). Repeating such an experiment with the weighted die produces the histogram in Figure 3.3 (b). Both histograms reflect a normal distribution. Should this be expected? Indeed, it is by the powerful Central Limit Theorem. If this is not readily apparent, take a moment and compare our experiments to Theorem 3.3.1.

Note that convergence is slower for highly screwed distributions. That is, larger sample sizes are needed for a histogram of sample averages to reflect the underlying normal distribution of such an experiment. We will see such behavior in the convergence of histograms for different experiments modeled with Monte Carlo methods in the next chapter.

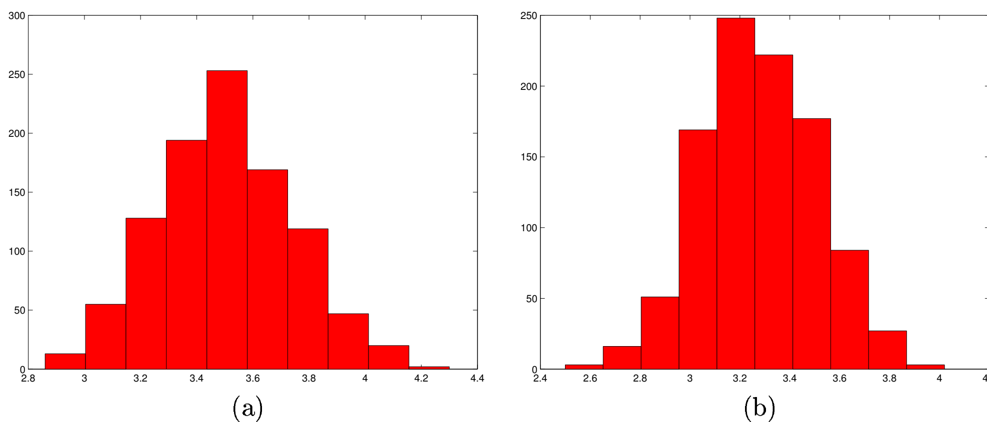
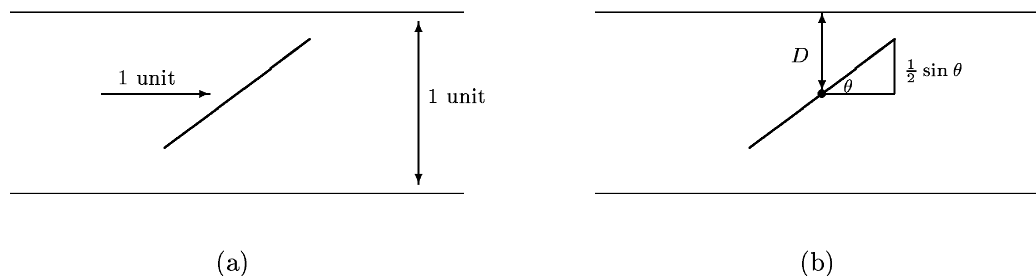



Figure 3.3: Histograms taken from sample averages (a) for a uniform distribution and (b) a nonuniform distribution.




(a) (b)

Figure 3.4: Buffon's Needle experiment; (a) depicts the experiment where a needle of unit length is randomly dropped between two lines of unit length apart. In (b), $D$ denotes the distance between the needle's midpoint and the closest line; $\theta$ is the angle of the needle to the horizontal.

## 3.4 Buffon's Needle

In 1777, Comte de Buffon described an experiment that relates the real number $\pi$ to a random event. Specifically, a straight needle of unit length is dropped at random onto a plane ruled with straight lines of unit length apart as depicted in Figure 3.4 (a). What is the probability $p$ that the needle will intersect one of the lines?

Before the time of digital computers, laptops and calculators, the term "computer" referred to a job title. Parallel computing meant calculating with a large number of mathematicians. Comte de Buffon performed the experiment repeatedly to determine $p$. Such hand computation limited the feasibility of mathematical experiments before the advent of the digital computer.

Comte de Buffon also derived the value of $p$ mathematically. Let $D$ denote the distance between the needle's midpoint and the closest line, and let $\theta$ be the angle of the needle to the horizontal. (Refer to Figure 3.4 (b).) Here both $D$ and $\theta$ are random variables. The needle crosses a line if

$$D \leq \frac{1}{2} \sin \theta.$$

Consider the plot of $\frac{1}{2} \sin \theta$ as seen in Figure 3.5. The probability of the needle intersecting a line is the ratio of the area of the shaded region to the area of the rectangle. This is the continuous version

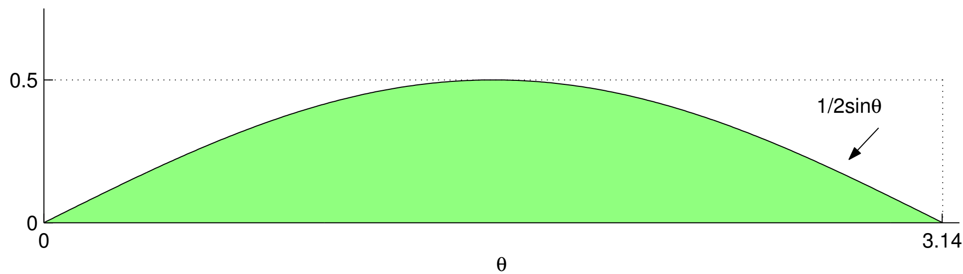



Figure 3.5: Plot of $\frac{1}{2}\sin\theta$ aids in analysis of Buffon's Needle.

of the uniform distribution. The area of the rectangle is clearly $\pi/2$. The area of the shaded region is

$$\int_0^\pi \frac{1}{2}\sin\theta \, d\theta = 1,$$

which implies that the probability of a hit is

$$p = \frac{1}{\left(\frac{\pi}{2}\right)} = \frac{2}{\pi}.$$

Hence, an estimate of $\pi$ is

$$2\left(\frac{\text{total number of drops}}{\text{total number of hits}}\right).$$

Just imagine, the next time that you drop a pencil, you have completed the first step in a Monte Carlo experiment in estimating $\pi$. Yet, it takes heroic patience before an accurate estimate emerges. We will see in class that Buffon's Needle experiment indeed produces an estimate of $\pi$ but is slow to converge.

# Chapter 4

# Monte Carlo Methods

Monte Carlo methods model physical phenomenon through the repeated use of random numbers. It can seem surprising that meaningful insights can be made from such results. Yet, Monte Carlo methods are used effectively in a wide range of problems including physics, mechanics, and economics.

Buffon's Needle is a classic example of Monte Carlo methods and the ability of random numbers to generate meaningful results. Note that no essential link exists between such methods and a computer. Yet, computers enormously enhance the methods' effectiveness. Monte Carlo methods are essentially carefully designed games of chance that enable research of interesting phenomenon.

While Monte Carlo methods have been used for centuries, important advancements in their use as a research tool occurred during World War II with mathematicians including Neumann and Ulam and the beginnings of the modern computer. Today, Monte Carlo methods are used in a wide variety of fields to model the physical world.

With our knowledge of statistics and ideas regarding the use of random numbers in simulation, we are ready to examine problems that we can model and study with Monte Carlo methods.

## 4.1 A fish tank modeling problem

The XYZ Tropical Fish Store has come to us for advice on the following problem. They carry 150 gallon fish tanks which are quite large and take up a lot of shelf space in the store. They observe that on average they only sell about 1 each week, so they don't want to have too many in stock at any one time. On the other hand, these are a high-profit item and if they don't have one in stock when a customer comes in, the customer will go elsewhere. Moreover, it takes 5 days for a new one to be delivered by the distributor after they place an order.

They are trying to decide on the best strategy for ordering the 150 gallon tanks. They are considering these two in particular:

1. **Strategy 1:** Order a new tank each time one is sold. Then they will never have more than one in stock at a time, so they won't waste much shelf space, but they may often be caught without any.

2. **Strategy 2:** Order a new tank once a week, arriving 5 days later. Since they sell one a week on average it may be good to have one arrive each week on average. There's probably less chance of being caught without one in stock, but if there aren't any customers for several weeks in a row they will accumulate.

The question is: which strategy is better, or is there another that is better yet?

Think about some of the questions that might naturally arise if you were going to try to determine an answer. What questions should you ask the store owners before you even try to model it? What

other information would you need to know before you could hope to evaluate the strategies? Thinking about this should help to suggest what is important in a mathematical model.

Here are a few examples:

- How much is the profit on one of these fish tanks, relative to other items in the store that would have to be eliminated in order to stock more fish tanks? (This is one thing we would need to know in order to make any sort of quantitative comparison of different strategies.)
- What is the cost of having additional tanks in stock? What is the cost of losing a customer by not having a tank available? (Besides the lost sale itself, there might be the cost of losing that customers business in the future, if they decide to buy all their fish and supplies from the same place they buy the tank.)
- What do you mean by "we sell one a week on average"? Is there one person in town who comes in every Wednesday at noon and always buys one tank? Or are there 52 people in town who always give a tank as a Christmas present, so there are lots of sales in the month of December and none in other months? Either scenario would lead to "one a week on average" over the year, but would lead to very different ordering strategies.

This is a classic example of an *inventory problem.* Variants of this problem are constantly being solved by businesses, which often have to decide on an ordering strategy for many different products simultaneously, even thousands for a grocery store, for example.

Here we consider the two simple strategies mentioned above, based on some simplified assumptions of customer behavior. In particular, we'll assume the sales are uniform over the year. We'll also assume that "an average of once a week" means that each day there is a $1/7$ chance that a customer will come. This isn't completely realistic: surely there's some chance that two customers will appear in the same day. Later we'll consider this further. Note in making these assumptions that we follow a primary rule of model building:

> *It is best to start with a simple model and refine it incrementally. Trying to start with a completely realistic model is usually hopeless.*

## 4.2 Monte Carlo simulation

The approach we will consider at this stage is a *simulation* of how the competing strategies might work. This means we write a computer program which simulates several weeks of how the store's inventory fluctuates assuming one strategy or the other, and see which pays off better. Of course we don't know exactly when customers arrive, but we can model the arrival of customers at random times by generating random numbers on the computer and using these values to indicate whether a customer arrives on each day or not. For example, in MATLAB, the command

```
rand(1)
```

generates 1 random number which is uniformly distributed between 0 and 1. If this number is less than $1/7$ we can say that a customer comes, if it is greater than $1/7$ we can say that a customer does not come.

On the next page is a simple MATLAB program which simulates the strategy of ordering a new fish tank only when the one in stock is sold. In this case there is always at most one tank in stock, which simplifies the program. The program prints out information about what happens each day and keeps track of statistics over the course of the simulation.

The parameters have been changed here to make it possible to see more action in a short simulation. The probability of a customer each day is $1/3$ and it only takes 2 days for a new tank to be delivered after ordering. The program has been simplified by leaving out some commands to keep track of totals and display results.

A more complete program which also allows comparison of different strategies can be found in Section 4.5, and downloaded from the class webpage.

```
a = 1/3;   % probability of a customer each day
days_for_delivery = 2;   % days from order to delivery of new tanks

stock = 1;   % number of tanks in stock
deliv = -1;  % number of days until delivery of tank on order
             % -1 means none on order

total_cust = 0;
total_sold = 0;
total_lost = 0;

for week = 1:3
  for weekday = 1:7
    sold = 0;
    lost = 0;
    if deliv == 0
           stock = stock+1;   % a new tank is delivered
           end
    if deliv >= 0
           deliv = deliv-1;   % days till next delivery
           end

    random_num = rand(1);     % generate random number
    if random_num < a         % use this number to tell if a customer arrived
       customers = 1;
      else
       customers = 0;
      end

    if customers==1
       if stock>0             % we have a tank to sell the customer
          sold = sold+1;
          stock = stock-1;
          if deliv < 0
             deliv = days_for_delivery;  % we sold a tank and now order
                                         % another one.
             end
        else
          lost = lost+1;      % we didn't have a tank and lost a customer
        end
      end % if customers==1

    % keep track of total statistics:
    total_cust = total_cust + customers;
    total_sold = total_sold + sold;
    total_lost = total_lost + lost;

    disp([week weekday stock customers sold lost]);   % display results for
                                                      % this day
    end % loop on weekday
  end % loop on week
% output total statistics now....
```

Figure 4.1: First pass at a MATLAB program for Monte Carlo simulation of the fish tank modeling problem.

Here are the results of one sample simulation:

| week | day | stock | cust | sold | lost |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 2 | 1 | 0 | 0 | 0 |
| 1 | 3 | 1 | 0 | 0 | 0 |
| 1 | 4 | 0 | 1 | 1 | 0 |
| 1 | 5 | 0 | 0 | 0 | 0 |
| 1 | 6 | 0 | 0 | 0 | 0 |
| 1 | 7 | 0 | 1 | 1 | 0 |
| 2 | 1 | 0 | 1 | 0 | 1 |
| 2 | 2 | 0 | 0 | 0 | 0 |
| 2 | 3 | 0 | 1 | 1 | 0 |
| 2 | 4 | 0 | 0 | 0 | 0 |
| 2 | 5 | 0 | 1 | 0 | 1 |
| 2 | 6 | 1 | 0 | 0 | 0 |
| 2 | 7 | 1 | 0 | 0 | 0 |
| 3 | 1 | 0 | 1 | 1 | 0 |
| 3 | 2 | 0 | 1 | 0 | 1 |
| 3 | 3 | 0 | 0 | 0 | 0 |
| 3 | 4 | 1 | 0 | 0 | 0 |
| 3 | 5 | 1 | 0 | 0 | 0 |
| 3 | 6 | 1 | 0 | 0 | 0 |
| 3 | 7 | 1 | 0 | 0 | 0 |

total over entire simulation:

| customers | sold | lost |
|---|---|---|
| 7 | 4 | 3 |

It's hard to come to any conclusion from this one short simulation. The fact that we could only serve 4 of the 7 customers was due to the particular pattern in which they arrived. If they had appeared differently, the numbers could have been very different.

To get a better feel for how this strategy works in the long run, we can run the simulation for many more weeks. For example we might run it for 104 weeks (2 years) rather than only 3 weeks. We might also run the simulation several times over to see if we get similar results even though the exact pattern of customer arrivals will be different in each case. Here are the results of 10 different simulations, each for 104 weeks:

| customers | sold | lost | fraction served |
|---|---|---|---|
| 241 | 147 | 94 | 0.6100 |
| 223 | 135 | 88 | 0.6054 |
| 255 | 149 | 106 | 0.5843 |
| 266 | 158 | 108 | 0.5940 |
| 237 | 144 | 93 | 0.6076 |
| 235 | 144 | 91 | 0.6128 |
| 255 | 148 | 107 | 0.5804 |
| 250 | 147 | 103 | 0.5880 |
| 227 | 138 | 89 | 0.6079 |
| 249 | 152 | 97 | 0.6104 |

Notice that we expect roughly $104 \times 7/3 = 243$ customers to appear over this time period (since we're using the rate of 1 every 3 days). In all of the above results we have roughly this number. Much

more interesting is the fact that in each simulation we end up being able to serve about 60% of these customers. This might be one number we want to use in comparing different strategies.

**Markov chains.** By doing the above simulations we came to the conclusion that with this strategy and particular set of parameters, we could serve about 60% of the customers who come looking for a 150 gallon tank. It turns out that we can predict this number using mathematics without doing any simulations, if we use the theory of Markov chains. We will study this later in the quarter.

### 4.2.1 Comparison of strategies

Section 4.5 contains a more complex program for this Monte Carlo simulation which allows comparison of different strategies. The variable `order_when_out` is a flag which determines whether to order a new tank whenever we run out. The variable `fixed_delivery` determines whether there is a fixed schedule of deliveries, once every so many days. Strategy 1 corresponds to

```
order_when_out = 1;
fixed_delivery = 0;   % no standing order
```

while Strategy 2 uses

```
order_when_out = 0;
fixed_delivery = 7;   % arrivals every 7th day
```

Percentage of customers served is one important number to compare. More important to the business, however, might be the expected profit over time with different strategies. This depends not only on how many customers we serve or lose, but also on the cost of any overstock. For Strategy 1 tested above, there is never more than 1 tank in stock and so no overstock. Strategy 2 can easily lead to overstock.

Here are the results of a few 104-week simulations with each strategy. We also compute the "profit" for each simulation and the average profit over 100 simulations with each strategy. This profit is based on the assumed profit per sale, cost of losing a customer, and cost per night of overstock as illustrated in the program.

```
Strategy 1:

customers  sold      lost   fraction_served    overstock  end_stock    profit
   106      63        43         0.594             0          1         830.00
   100      61        39         0.610             0          1         830.00
   104      58        46         0.558             0          1         700.00
   104      60        44         0.577             0          0         760.00
    94      55        39         0.585             0          1         710.00
    90      59        31         0.656             0          0         870.00
etc...

   108      58        50         0.537             0          1         660.00
   101      59        42         0.584             0          0         760.00
    94      54        40         0.574             0          0         680.00
    99      61        38         0.616             0          1         840.00
    98      55        43         0.561             0          1         670.00

average_profit = 787.9000
```

```
Strategy 2:

customers  sold      lost   fraction_served   overstock  end_stock   profit
    93      92         1          0.989          4795        13       -567.50
    99      98         1          0.990          1802         7       1049.00
    98      95         3          0.969          2716        10        512.00
    96      96         0          1.000          3690         9         75.00
    99      98         1          0.990          3098         7        401.00
    87      87         0          1.000          6811        18      -1665.50
   106     101         5          0.953          3450         4        245.00
etc...

    94      94         0          1.000          2867        11        446.50
   126     103        23          0.817           493         2       1583.50
   106     101         5          0.953          2023         4        958.50
   131     104        27          0.794           709         1       1455.50
   109     104         5          0.954           637         1       1711.50
    91      88         3          0.967          6135        17      -1337.50
    91      90         1          0.989          4165        15       -292.50
   101     101         0          1.000          3899         4         70.50
   101     100         1          0.990          3755         5        112.50

average_profit = 342.7050
```

Note some interesting features of these simulations:

- The percentage of customers served is generally larger with Strategy 2, usually above 90%, while with Strategy 1 the percentage is around 60%. However, the average profit is larger with Strategy 1. This is because Strategy 2 generally leads to an overstock. Customers are rarely lost but there's a large cost associated with this overstock which reduces profit.

- The results for Strategy 1 have much smaller variance than for Strategy 2. The profit for each individual simulation is typically not too far from the average. For Strategy 2 the values are all over the place. Some simulations give much higher profits than Strategy 1 while many simulations resulted in negative profits (a net loss). This is seen more strikingly by plotting histograms of the profits over each set of 100 simulations, as shown in Figure 4.2. If the business is lucky, profits may be higher with Strategy 2, but there is a great risk of being unlucky.

- Note one problem with Strategy 2 as implemented so far: There is no mechanism to stop automatic delivery if the stock goes too high. Since customers arrive at the same average rate as new tanks, a high stock will tend to stay high. Figure 4.3 shows the stock of tanks as a function of day in the course of one particular unlucky simulation. By the end of two years the stock has grown to 20 tanks.

## 4.3 A hybrid strategy

A better ordering strategy might be to have a standing order for a tank to arrive every $N$ days, where $N$ is larger than 7, and then to also order a new tank whenever we run out. In the program $N$ is denoted by `fixed_delivery`. Figure 4.4 shows histograms of the performance of this strategy for two different values of $N$, $N = 9$ and $N = 14$.

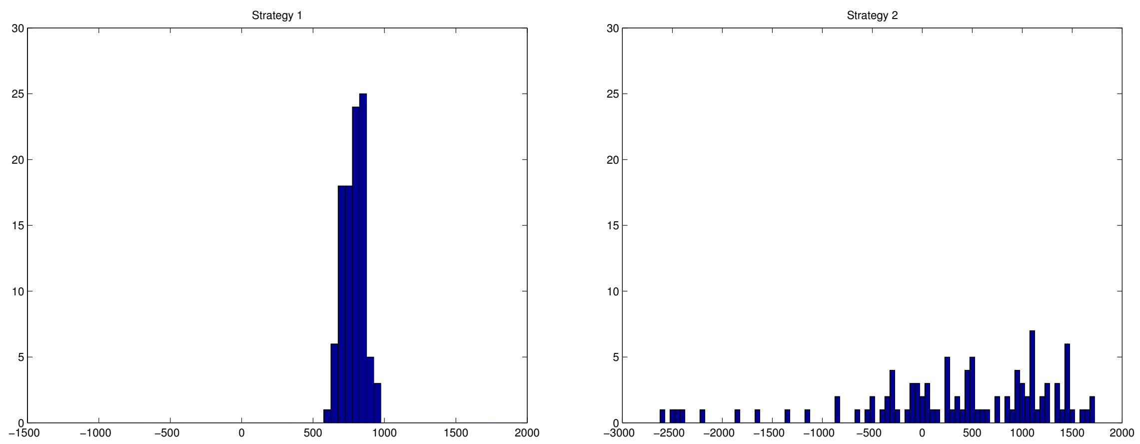



Figure 4.2: Histograms of profit observed over 100 simulations with two different strategies.

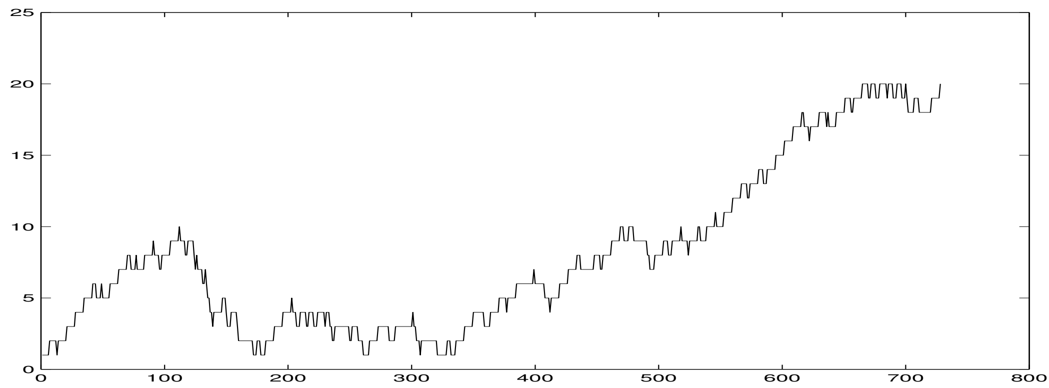



Figure 4.3: History of number of tanks in stock each day for one particular simulation with Strategy 2.

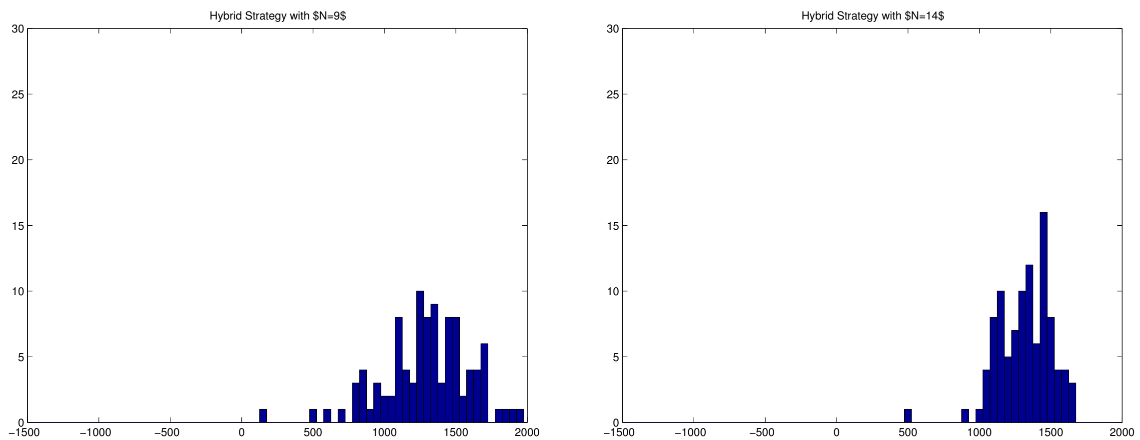



Figure 4.4: Histograms of profit observed over 100 simulations with hybrid strategies.

## 4.4 Statistical analysis

In comparing the results of simulations with different strategies, we need to be able to reach sensible conclusions from a large quantity of "experimental data". Typically we would also like to be able to quantify how confident we are in the conclusions we reach, how sensitive these conclusions are to the parameters we chose in the model, etc. This is the realm of *statistics*, and sensible analysis of Monte Carlo simulations generally requires a good knowledge of this field.

One statistic we have already looked at is the *average profit* observed for each strategy. If one strategy gives a larger average profit over many simulations than another strategy, it *might* be the better strategy. On the other hand we have also observed that some strategies give results with much greater *variance* between simulations than other strategies. In practice the owners of the store don't really care about the average behavior over hundreds of simulations, they care about what will happen in the one real experience they will have with whatever strategy they adopt. The point of doing many simulations is to get some experience with what *range* of different outcomes they might reasonably expect to observe in practice. (Assuming of course that the model is good enough to match reality in some sense.)

Looking at the histograms of Figure 4.2, we see that with Strategy 1 we might be able to predict fairly confidently that the profit which will actually be observed (assuming the model is good enough ...) will lie somewhere between 600 and 1000. With Strategy 2, we can only predict that it will probably lie somewhere between -2000 and 2000. Even if the average observed with Strategy 2 were higher than the average profit observed with Strategy 1, the strategy with the smaller variance might be preferable in practice because there would be less risk involved.

If we do $N$ simulations and obtain $N$ different values $y_1, y_2, \ldots, y_N$ for the observed profit, then the average (sample mean) is defined as

$$\bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i.$$

The *sample variance* is

$$s^2 = \frac{1}{N-1}\sum_{i=1}^{N}(y_i - \bar{y})^2,$$

and the *sample standard deviation* is

$$s = \sqrt{s^2}.$$

We can view the $N$ values observed as $N$ observations of a random variable chosen from some distribution. Typically we don't know what this distribution is, and the point of doing a Monte Carlo simulation is to gain some information about the distribution. With a large enough sample, the sample mean should approximate the mean of the true distribution and the sample variance should approximate its variance.

How big a sample is "large enough"? That depends on the distribution. Comparing the histograms of Figure 4.2, for example, we see that 100 trials gives more information about the distribution of results in Strategy 1 than with Strategy 2. To get a better idea of the underlying distributions, we might repeat these experiments with many more simulations. Figure 4.5 shows histograms of the profits observed when 5000 simulations are performed with each strategy. Here we get a much better sense of the distributions.

For Strategy 1 we merely confirm what we already guessed from the 100 simulations of Figure 4.2. The distribution has a fairly "Gaussian" shape with almost all results lying between 600 and 1000. In fact it can be shown that the distribution of observed profits should be approximately normally distributed (Gaussian). This follows from the *Central Limit Theorem* as we will be better able to understand after studying the Markov Chain interpretation of this model.

The sample mean from this particular set of runs is about 783 and the standard deviation is about 83. For a normally distributed random variable, it can be shown that the value observed will be farther than twice the standard deviation away from the mean only about 5% of the time. So if the profit with

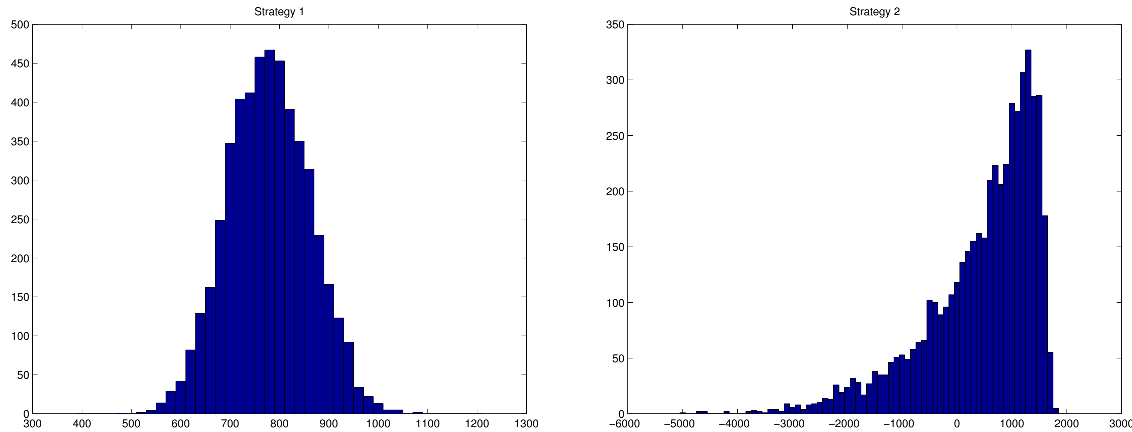



Figure 4.5: Histograms of profit observed over 5000 simulations with two different strategies.

Strategy 1 is essentially normally distributed then we expect that the profit observed should lie between $783 - 2 \cdot 83 = 617$ and $783 + 2 \cdot 83 = 949$ about 95% of the time.

For Strategy 2 we see in Figure 4.5 that the distribution of profits is certainly not a normal distribution. Note that the profit would not be expected to exceed about $104 \cdot 20 = 2080$ even with a perfect strategy (why not?), which partly accounts for the sharp cut-off at the higher end. The fact that a very large overstock can accumulate accounts for the wide spread to the left. The average profit observed is 439 and the standard deviation is 1020. In 26% of the simulations the observed profit was negative.

There are many other statistical questions that one might ask about the data obtained from these simulations, and various ways that the model might be improved. We will discuss some of these in class.

## 4.5 Fish tank simulation code

```
% set parameters:
nsim = 100;          % number of different simulations to do
nweeks = 104;        % number of weeks in each simulation
ndays = 7*nweeks;    % number of days in each simulation

a = 1/7;    % probability of a customer each day

days_for_delivery = 5;    % days from order to delivery of new tanks
order_when_out = 1;       % = 1  ==>  order a new tank when stock==0
                          % = 0  ==>  don't order when out of tanks

fixed_delivery =  12;      % >0   ==>  standing order for a new tank
                          %           every so many days

% profits and losses:
saleprofit = 20;          % profit from selling one tank
lostloss = 10;            % loss from losing a customer
overstockloss = .50;      % cost of each tank overstock per night

% initialize:
profit = zeros(nsim,1);

% print column headings:
```

```
disp('customers  sold      lost   fraction_served   overstock  end_stock   profit')


for sim=1:nsim
  % initialize:
  random_nums = rand(ndays,1);   % array of random numbers to use each day
  total_cust = 0;
  total_sold = 0;
  total_lost = 0;
  stock = 1;   % number of tanks in stock
  deliv = -1;  % number of days until delivery of tank on order
               % -1 means none on order
  overstock = 0;  % increment every night by number of excess tanks in stock


  % main loop for a single simulation:

  day = 0;
  for week = 1:nweeks
    for weekday = 1:7
      day = day+1;              % day in the simulation
      sold = 0;
      lost = 0;
      if deliv == 0
             stock = stock+1;   % a new tank is delivered
                        % at the beginning of the day
             end
      if deliv >= 0
             deliv = deliv-1;   % days till next delivery
             end


      if (mod(7*week + day, fixed_delivery) == 0)
           % A new tank is delivered every so many days regardless of stock
           stock = stock+1;
           end


      % use random number to decide how many customers arrived
      % Here assume 0 or 1:
      if random_nums(day) < a
         customers = 1;
        else
         customers = 0;
        end


      if customers==1
         if stock>0             % we have a tank to sell the customer
            sold = sold+1;
            stock = stock-1;
          else
            lost = lost+1;      % we didn't have a tank and lost a customer
          end
        end

      if (order_when_out & stock==0 & deliv < 0)
```

```
          % none in stock and none on order
   deliv = days_for_delivery;  % order another
   end

      if stock > 1
         overstock = overstock + (stock - 1);
 end

      % keep track of total statistics:
      total_cust = total_cust + customers;
      total_sold = total_sold + sold;
      total_lost = total_lost + lost;
      stock_record(day) = stock;  % keep track of stock on each day

      end  % loop on day
    end  % loop on week




  fraction_served = total_sold / total_cust;

  profit(sim) = total_sold*saleprofit ...
               - total_lost*lostloss - overstock*overstockloss;

  % output total statistics:
  disp(sprintf('%6.0f  %6.0f  %8.0f  %12.3f  %11.0f  %8.0f  %12.2f', ...
      total_cust, total_sold, total_lost, fraction_served, overstock, stock, ...
      profit(sim)))

  end % loop on sim

% compute and print average profit over all simulations:
average_profit = sum(profit) / nsim
disp(' ')
disp(['  average profit     = ' sprintf('%10.2f', average_profit)])

% standard deviation:
variance = sum((profit - average_profit).^2) / (nsim-1);
standard_deviation = sqrt(variance);
disp(['  standard deviation = ' sprintf('%10.2f', standard_deviation)])
disp(' ')

% plot histogram of profits over all simulations:
hist(profit,-1500:50:2000)
axis([-1500 2000 0 30])
```

## 4.6 Poisson processes

In the Monte Carlo simulation of the fish tank inventory problem, we assumed that each day there was a probability $a$ of a customer coming in. The MATLAB code contained the following lines:

```
random_num = rand(1);     % generate random number
if random_num < a         % use this number to tell if a customer arrived
   customers = 1;
```

```
else
 customers = 0;
end
```

This assumes that each day there is either 0 or 1 customers. This is not very realistic, since some days we would expect 2 or even more to appear if they come at random times. Moreover, we might want to simulate a situation in which there are typically many customers each day, instead of only one customer every few days.

More generally let $a$ represent the average number of customers per day. If $a$ is very small then we can interpret $a$ as the probability that a customer comes on any given day, and ignore the possibility of more than one customer. This isn't very realistic if $a$ is somewhat larger. If $a > 1$ then it is impossible to interpret $a$ as a probability, and clearly unrealistic to assume there will be either 0 or 1 customer.

It would be better if we could predict the probabilities $p_k$ that exactly $k$ customers will appear in a given day and then use something like:

```
random_num = rand(1);     % generate random number
if random_num < p0
     customers = 0;
  else if random_num < p0+p1
     customers = 1;
  else if random_num < p0+p1+p2
     customers = 2;
  else
     customers = 3;  % assume negligible chance of more
  end
  end
  end
```

Here we have assumed that $a$ is small enough that the chance of more than 3 customers is negligible, *i.e.*, $p_0 + p_1 + p_2$ is very close to 1.

One way to estimate these probabilities would be to do a Monte Carlo simulation. We could split a day up into $N$ much smaller pieces (hours, minutes, seconds or even smaller) and assume that in each small piece there is really no chance that more than one customer will appear, and that the probability of one customer appearing is $p = a/N$. We could simulate many days and see what fraction of the days there are $k$ customers for each $k$. (Note that if $N$ is large enough then $p = a/N$ will be very small even if $a > 1$, so it is valid to view this as a probability.)

Here are the results of a 10 trials with $N = 20$ and $a = 1$. (Generated using `poisson.m` from the webpage.) Each string has 20 bits, with 0 representing "no customer during this interval" and 1 representing "a customer during this interval". The first "day" two customers arrived, the second day only one, etc.

```
00000010001000000000
0000010000000000000
0001010000000000000
0000000000000000000
0000000000000000100
0000000000000000000
0000000000000000000
0000000100000000010
0011000000000000000
1010000000010000000
```

In this short simulation, 30% of the days no customer arrived, 20% had 1 customer, 40% had 2 customers, and there was 1 day (10%) when 3 customers arrived. If we ran another 10-day simulation we'd probably get different results. To estimate the probabilities more reliably we should do a simulation with many more days. Below are the results of simulations with 10,000 days. This was repeated 3 times and we get slightly different results each time.

| | Run 1 | Run 2 | Run 3 | Poisson |
|---|---|---|---|---|
| 0 customers | 0.3686 | 0.3580 | 0.3606 | 0.3679 |
| 1 customer | 0.3722 | 0.3717 | 0.3767 | 0.3679 |
| 2 customers | 0.1837 | 0.1911 | 0.1883 | 0.1839 |
| 3 customers | 0.0575 | 0.0646 | 0.0596 | 0.0613 |
| 4 customers | 0 | 0 | 0.0122 | 0.0153 |
| 5 customers | 0.0151 | 0.0120 | 0.0021 | 0.0031 |
| 6 customers | 0.0025 | 0.0024 | 0.0004 | 0.0005 |
| 7 customers | 0.0004 | 0.0002 | 0.0001 | 0.0001 |

Note that by running the simulation 3 times we get some reassurance that the value of 10,000 is large enough to give values that don't change much when we rerun the simulation. If we had run it for only 100 days each we would see quite a bit of variation from run to run and would know that we don't have reliable values. Here it seems fairly clear that $p_0 \approx 0.36$, $p_1 \approx 0.37$, $p_2 \approx 0.18$, etc.

**Poisson distribution.** In fact we can compute these values without doing a simulation. It can be shown mathematically that probabilities follow a *Poisson distribution*, and that for any value of $a$, the average number of customers per day, the probability of observing exactly $k$ customers in a single day is

$$p_k = \frac{a^k e^{-a}}{k!}.$$

In particular,

$$\begin{aligned} p_0 &= e^{-a} \\ p_1 &= ae^{-a} \\ p_2 &= \frac{1}{2}a^2e^{-a} \\ p_3 &= \frac{1}{6}a^3e^{-a} \end{aligned}$$

The values for the case $a = 1$ are shown in the last column of the table above, and they compare well with the values obtained from the three 10,000-day simulations.

Note that

$$\sum_{k=1}^{\infty} p_k = e^{-a}\left(1 + a + \frac{a^2}{2!} + \frac{a^3}{3!} + \cdots\right) = e^{-a}e^{a} = 1.$$

The probabilities all add up to one as we should expect.

The formulas for the Poisson distribution can be derived by using a combinatorial argument based on the probability of the event occurring in each small time period, and the number of ways it is possible to have 0, 1, 2, ... events occurring over the entire day. For this to be valid we must assume that customers appear randomly with probability $a/N$ in any small increment of $1/N$ day, without regard to the past history. The arrival of customers is then said to be a *Poisson process*. The sequence of tests to see whether there is a customer in each interval also called a series of *Bernoulli trials*.

To see how these values arise, first consider a simpler case where we look at a sequence of only 3 Bernoulli trials, with a probability $p$ of "success" in each trial (denoted by 1) and probability $1-p$ of "failure" (denoted by 0). There are $2^3 = 8$ possible strings that can arise from these 3 independent trials. These are listed below along with the probability of observing each:

$$\begin{array}{ll} \texttt{000} & (1-p)^3 \\ \texttt{001} & (1-p)^2 p \\ \texttt{010} & (1-p)^2 p \\ \texttt{011} & (1-p)p^2 \\ \texttt{100} & (1-p)^2 p \\ \texttt{101} & (1-p)p^2 \\ \texttt{110} & (1-p)p^2 \\ \texttt{111} & p^3 \end{array} \tag{4.1}$$

Note that there are 3 ways to observe 1 customer during the entire day, and 3 ways to observe 2 customers, but only 1 way to observe 0 or 3 customers. Adding up the probabilities we find the following values for $p_k^{(3)}$, the probability of observing $k$ customers when $N = 3$:

$$\begin{array}{ll} \text{0 customers} & (1-p)^3 = p_0^{(3)} \\ \text{1 customers} & 3(1-p)^2 p = p_1^{(3)} \\ \text{2 customers} & 3(1-p)p^2 = p_2^{(3)} \\ \text{3 customers} & p^3 = p_3^{(3)} \end{array} \tag{4.2}$$

Note that these probabilities sum to 1. We can see this easily using the fact that

$$(x+y)^3 = x^3 + 3x^2y + 3xy^2 + y^3$$

with $x = 1-p$ and $y = p$. In fact we can use this type of *binomial expansion* to easily compute the probabilities $p_k^{(N)}$ when $N$ is larger without enumerating all $2^N$ possible strings and counting things up. We have

$$(x+y)^N = x^N + \binom{N}{1} x^{N-1}y + \binom{N}{2} x^{N-2}y^2 + \cdots + \binom{N}{N-1} xy^{N-1} + y^N. \tag{4.3}$$

The values $\binom{N}{k}$ ("$N$ choose $k$") are often called the *binomial coefficients*. The probabilities we seek are

$$\begin{aligned} p_0^{(N)} &= (1-p)^N \\ p_1^{(N)} &= \binom{N}{1}(1-p)^{N-1}p = N(1-p)^{N-1}p \\ p_2^{(N)} &= \binom{N}{2}(1-p)^{N-2}p^2 = \frac{1}{2}(N^2-N)(1-p)^{N-2}p^2 \end{aligned} \tag{4.4}$$

etc.

This is called the *binomial distribution* for the total number of "successes" in $N$ Bernoulli trials.

To relate this to the Poisson distribution, recall that we want to determine the probability of observing $k$ customers in the case where $N$ is very large and $p = a/N$, where $a$ is a fixed value representing the average number of customers per day. The probability $p_0$ is given by

$$p_0 = \lim_{N\to\infty} p_0^{(N)} = \lim_{N\to\infty} \left(1 - \frac{a}{N}\right)^N = e^{-a}. \tag{4.5}$$

Similarly,

$$\begin{aligned} p_1 &= \lim_{N\to\infty} N \left(1 - \frac{a}{N}\right)^{N-1} \left(\frac{a}{N}\right) \\ &= a \lim_{N\to\infty} \left(1 - \frac{a}{N}\right)^{N-1} \\ &= ae^{-a}. \end{aligned} \tag{4.6}$$

## 4.7 Queuing theory

As another example of Monte Carlo simulation, we will look at an example from queuing theory. Suppose customers arrive at a service center or cashier and have to line up to wait for service. Queuing theory is the study of how such lines are expected to behave, and the comparison of different strategies. For example, if there are multiple servers we might want to compare the strategy of having independent lines for each (as is common in grocery stores) or have a single line with the next customer going to the first available server (as is common in banks and airport check-in counters). Which approach minimizes the average time a customer has to wait, for example?

There is a large literature on this subject, see for example [HL95]. Here we will look at the simplest example of a single server and discuss how Monte Carlo simulation might be used to study the expected length of the line.

The MATLAB code in Figure 4.6 shows a simple simulation of a queue. Customer arrivals are modeled with a Poisson process with average rate $a$ per minute. They must wait in the queue until the single server is free. The amount of time required to serve them is also random, with average rate of $b$ people per minute. In this simulation this is also assumed to be Poisson process with probability of roughly $b/N$ that they will be finished in any small subunit of time $1/N$. (This may not be very realistic. A better model might be that there is some minimal service time plus a random length of time after that, or some other distribution of service times.)

One could model this by splitting each minute up into $N$ subunits and generating a random number to determine whether a new customer arrives, and a second random number to determine whether the customer currently being served finishes.

Instead, this simulation works by first generating a set of arrival times and service times for each customer, and then computing the total time that each customer must wait on the basis of these times. The times are generated using an *exponential distribution*:

$$\text{Probability that inter-arrival time is larger than } T = e^{-aT}.$$

This can be shown to be the proper distribution of service times for a Poisson distribution in the following manner. Suppose we split up the time since the last arrival into units $T, 2T, 3T, \ldots$ rather than minutes. Then in each unit the expected number of arrivals is $aT$. According to the Poisson distribution, the probability of exactly 0 arrivals in the first of these periods is thus $e^{-aT}$, but this is exactly the probability that the next arrival time is more than $T$ units later.

To generate random numbers from an exponential distribution with parameter $a$, we can first generate a random number $r$ uniformly distributed in the interval $[0, 1]$ (using `rand` in MATLAB, for example) and then compute

$$-\frac{1}{a}\log(r).$$

Note that `log` is the natural logarithm command in MATLAB.

Figure 4.7 shows the results of three simulations using the parameters $a = 1$ and $b = 1.5$, for 1000 customers each. Two plots are shown for each simulation: on the left is the length of the queue as a function of time, and on the right is a histogram of the total time spent by each customer (including both the time in line and the time being served). While the 3 simulations show variation in details, some similarities are observed. Note that although the server can handle 50% more customers than are arriving on average, the queue length can grow quite long due to the randomness of arrivals. In all three simulations the queue length is more than 10 at times. Also, looking at the histogram we see that there are some customers who have to wait as long as 10 or 12 minutes in each simulation, though the average total time is more like 2 minutes.

Figure 4.8 shows another set of simulations in which the parameter $b = 1.1$ is used instead, in which case the average service time $1/b \approx 0.91$ is even closer to the average interarrival time. Again the details vary between simulations, but it is clear that now the queues will typically grow to be of length greater than 20 at times and the average waiting time is much longer. There are now typically some customers who have to wait more than 20 minutes.

From comparing these sets of results it should be clear that such simulations can be effectively used to predict how badly queues may behave if the proper parameters are estimated. One can also use Markov chain ideas to study the expected behavior of such systems and determine expected values for quantities such as the total waiting time. In fact it can be shown that for arbitrary parameters $a < b$, the expected total waiting time is $1/(b - a)$. Note that this is consistent with what is observed in these simulations.

We have only looked at the simplest situation of a single queue and a single server. This model is called an M/M/1 queue. The first M indicates that the arrival times are exponentially distributed (or Markovian), the second M indicates that the service times are also Markovian, and the 1 indicates there is a single server. More complicated models would be required to model a bank or super market with multiple lines and servers, for example.

Various modifications of this model are possible. For example, we could assume a maximum queue length beyond which customers go away rather than joining the queue. In this case there are a finite set of states possible (different queue lengths) and it is possible to derive a Markov chain model of transitions, as we will look at later.

```
% Simple queuing theory simulation,  M/M/1 queue
% Single server, single queue:

a = 1;  % average number of arrivals per minute
b = 1.5;  % average number of people served per minute

ncust = 1000;
% Notation:
%  at = arrival time of a person joining the queue
%  st = service time once they reach the front
%  ft = finish time after waiting and being served.
%

% initialize arrays:
at = zeros(ncust,1);
ft = zeros(ncust,1);

% Generate random arrival times assuming Poisson process:
r = rand(ncust,1);
iat = -1/a * log(r);   % Generate inter-arrival times, exponential distribution
at(1) = iat(1);        % Arrival time of first customer

for i=2:ncust
   at(i) = at(i-1) + iat(i);   % arrival times of other customers
   end

% Generate random service times for each customer:
r = rand(ncust,1);
st = -1/b * log(r);   % service times for each


% Compute time each customer finishes:
ft(1) = at(1)+st(1);  % finish time for first customer
for i=2:ncust
   % compute finish time for each customer as the larger of
   %   arrival time plus service time (if no wait)
   %   finish time of previous customer plus service time (if wait)
   ft(i) = max(at(i)+st(i),  ft(i-1)+st(i));
   end

total_time = ft - at;           % total time spent by each customer
wait_time  = total_time - st;   % time spent waiting before being served

ave_service_time = sum(st)/ncust
ave_wait_time = sum(wait_time)/ncust
ave_total_time = sum(total_time)/ncust

% Plot histogram of waiting times:
hist(total_time,0:.5:20)
```

Figure 4.6: Queuing theory simulation. The code shown here is simplified and doesn't show the computation of the queue length which is plotted here. See the Handouts webpage for the full m-file.

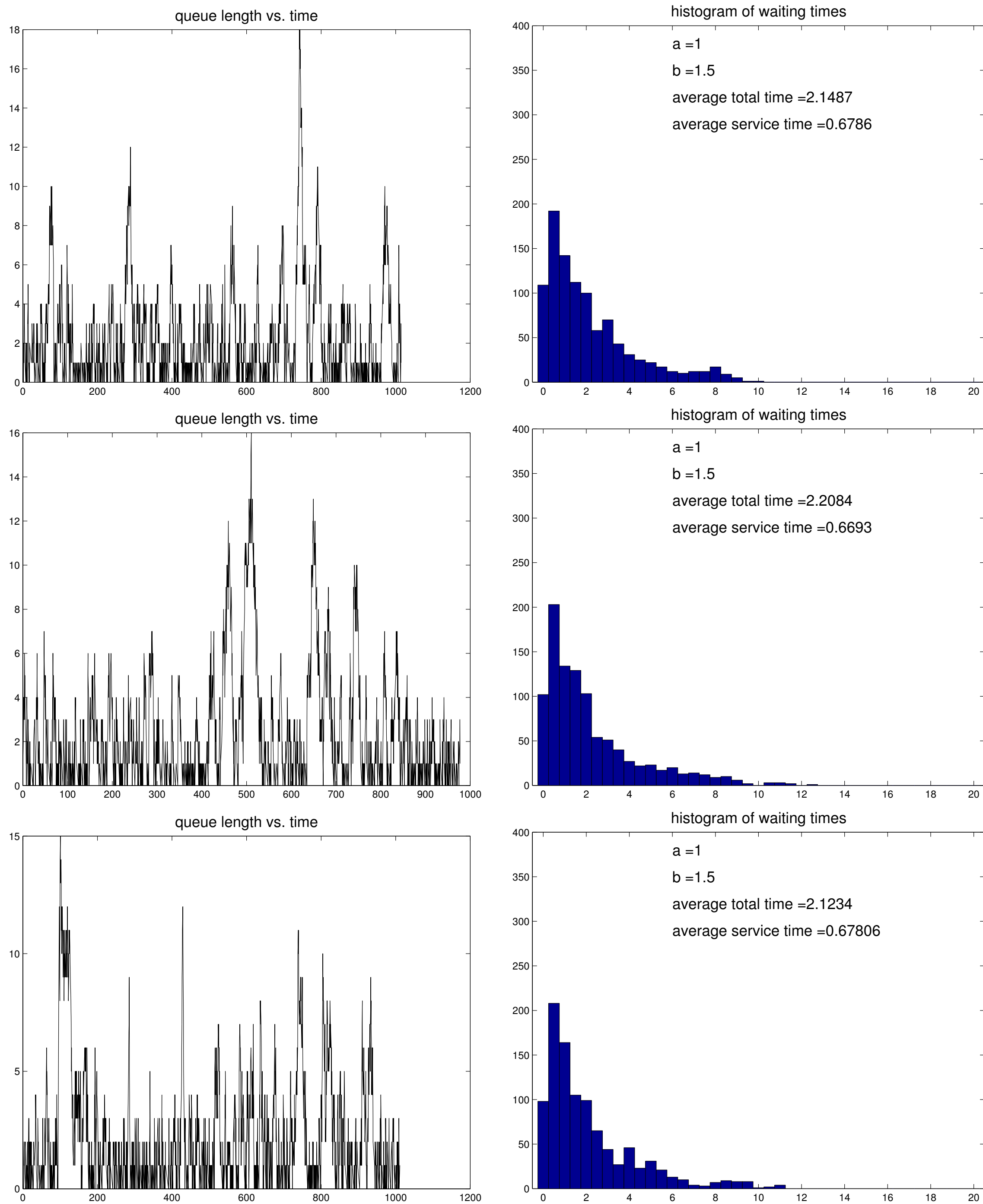



Figure 4.7:

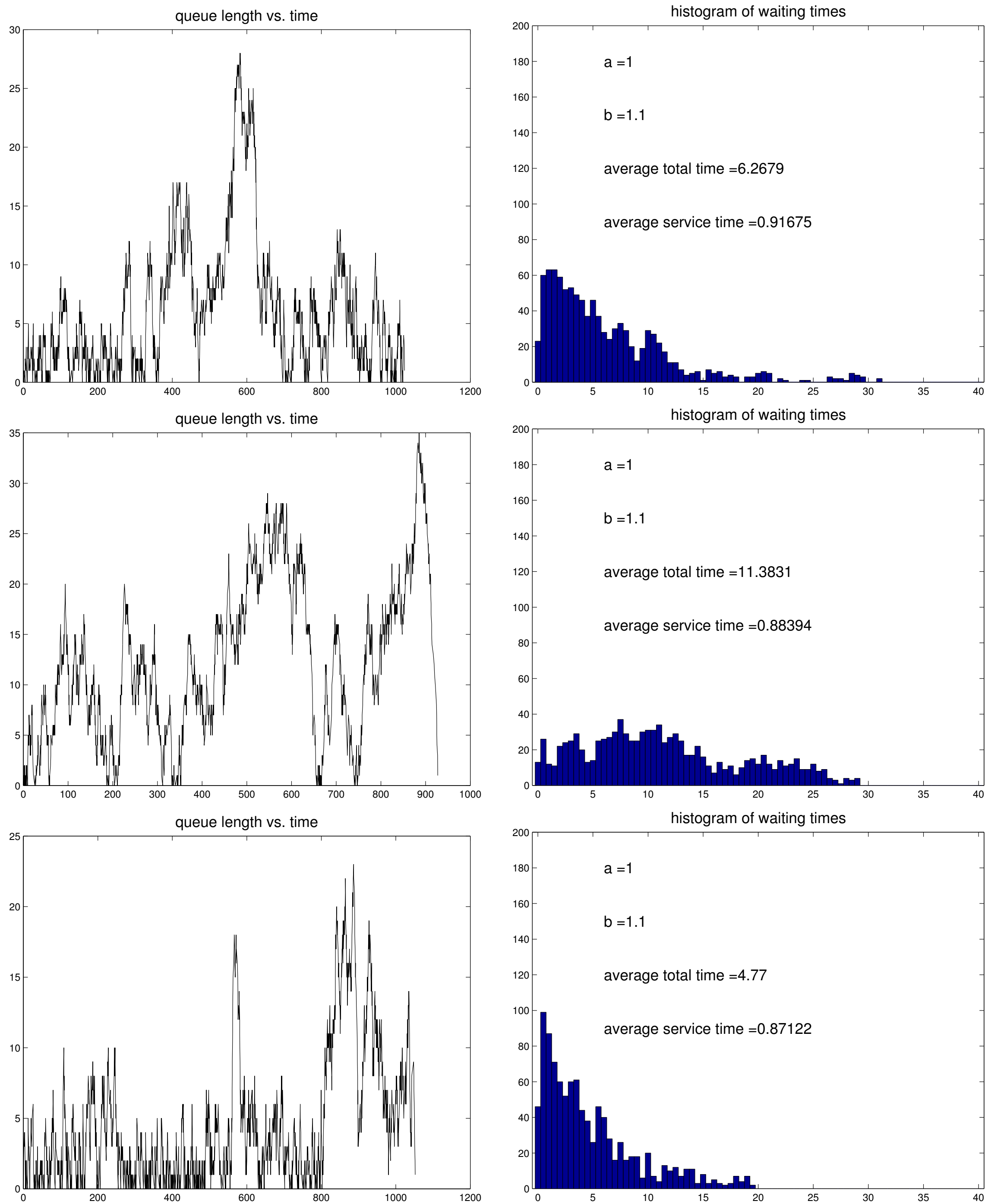



Figure 4.8:

# Chapter 5

# Markov Chains and Related Models

## 5.1 A Markov chain model of a queuing problem

To introduce the theory of Markov chains, let's consider a simple queuing theory model in which there is a single server and there are only three "states" which the queue can be in at any given time:

| | |
|---|---|
| State 1: | No customers are present |
| State 2: | A customer is being served |
| State 3: | A customer is being served and another is waiting. |

Assume that the queue is never longer: if a customer arrives when the queue is in State 3 then that customer disappears.

Suppose that arrivals and service are Poisson processes, and over a given short time interval there is a probability $p$ of a new customer arriving and a probability $q$ that any customer currently being served will finish. If the time interval is very short then these are very small and we can assume there is negligible chance of more than one customer arriving or finishing, and also negligible chance of both one customer finishing and another arriving in the same period.

We could perform a Monte Carlo simulation of this queue by keeping track of what state it is in, and each time period either staying in this state or moving to a different state depending on the value of some random number:

```
random_num = rand(1);
if random_num < p
    # a new customer arrives:
    if state<3
       state = state + 1;
       end
    end
if random_num > 1-q
    # a customer finishes:
    if state>1
       state = state - 1;
       end
    end
```

Note that we've done the tests in such a way that if $p$ and $q$ are both very small then we can't have both happen.

Figure 5.1 shows the *transition diagram* between the three states. For example, if we are in State 2, then with probability $p$ we move to State 3 (if a customer arrives), with probability $q$ we move to State 1 (if a customer finishes), and with probability $1-p-q$ we stay in State 2 (if neither happens).

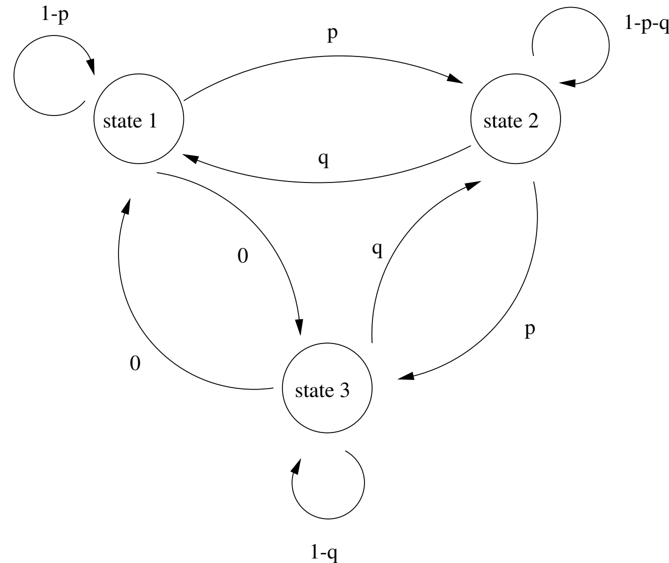



Figure 5.1: Transition diagram for a simple queuing theory model with three states.

If we do a single simulation then we will move around between the three states. We might be interested in determining what percentage of the time we are in each of the three states over the long term. We could determine this for many different simulations and collect some statistics. If we do this with $p = 0.05$ and $q = 0.08$, for example, we will observe that about 50% of the time we are in state 1, about 31% in State 2, and about 19% in State 3.

Another way to get these number is to run lots of simulations simultaneously and keep track of what percentage of the simulations are in each state at each time period. After a while we might expect to see the same percentages as reported above (why??).

Here's what we observe if we do 10,000 simulations, all starting in State 1, and print out how many of the simulations are in each of the three states at each time:

```
    time period     State 1    State 2    State 3

              0       10000          0          0
              1        9519        481          0
              2        9073        891         36
              3        8685       1240         75
              4        8379       1491        130
              5        8096       1702        202
              6        7812       1914        274
etc...
             95        4949       3134       1917
             96        4913       3166       1921
             97        4920       3161       1919
             98        4958       3148       1894
             99        4982       3072       1946
            100        4995       3038       1967
etc...
```

Initially all simulations were in State 1. After one time period a customer had arrived in 481 of the simulations, or 4.81% of the simulations, about what we'd expect since $p = 0.05$. Since it's impossible for 2 customers to arrive in a single period, there are still no simulations in State 3 after one period.

After 2 periods, however, it is possible to be in State 3, and in fact 36 of the simulations reached this state. How does this agree with the number you should expect?

Notice that after many time periods, for $n$ around 100, we appear to have reached a sort of "steady state", with roughly 50% of the simulations in State 1, 31% in State 2, and 19% in State 3. Of course each individual simulation keeps moving around between states, but the number in each stays roughly constant. Note that these are the same percentages quoted above.

What we now want to explore is whether we can determine these values *without* having to do thousands of simulations. In fact we can by using the theory of Markov chains. Let $v_k^{(n)}$ be the fraction of simulations which are in State $k$ after $n$ steps (for $k = 1,\ 2,\ 3$), and let $v^{(n)}$ be the vector with these three components. If all simulations start in State 1 then we have

$$v^{(0)} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}. \tag{5.1}$$

What fractions do we expect for $v^{(1)}$? Since each simulation moves to State 2 with probability $p = 0.05$, or stays in State 1 with probability $1 - p = 0.95$, we expect

$$v^{(1)} = \begin{bmatrix} 0.95 \\ 0.05 \\ 0 \end{bmatrix}.$$

Now consider the general case. Suppose we know $v^{(n)}$ and we want to determine $v^{(n+1)}$. From the transition diagram of Figure 5.1 we expect

$$\begin{aligned} v_1^{(n+1)} &= (1-p)v_1^{(n)} + qv_2^{(n)} + 0 \cdot v_3^{(n)} \\ v_2^{(n+1)} &= pv_1^{(n)} + (1-p-q)v_2^{(n)} + qv_3^{(n)} \\ v_3^{(n+1)} &= 0 \cdot v_1^{(n)} + pv_2^{(n)} + (1-q)v_3^{(n)}. \end{aligned} \tag{5.2}$$

We can write this in matrix-vector notation as

$$v^{(n+1)} = Av^{(n)}, \tag{5.3}$$

where the *transition matrix* $A$ is

$$A = \begin{bmatrix} 1-p & q & 0 \\ p & 1-p-q & q \\ 0 & p & 1-q \end{bmatrix} = \begin{bmatrix} 0.95 & .08 & 0 \\ 0.05 & 0.87 & 0.08 \\ 0 & 0.05 & 0.92 \end{bmatrix}. \tag{5.4}$$

The right-most matrix above is for the specific values $p = 0.05$ and $q = 0.08$. From the relation (5.3) we can compute

$$v^{(1)} = \begin{bmatrix} .95 \\ .05 \\ 0 \end{bmatrix}, \quad v^{(2)} = \begin{bmatrix} .9065 \\ .0910 \\ .0025 \end{bmatrix}, \quad v^{(3)} = \begin{bmatrix} .8685 \\ .1247 \\ .0069 \end{bmatrix}, \quad \text{etc.} \tag{5.5}$$

Note that these values agree reasonably well with what is seen in the simulation after 1, 2, and 3 steps. To predict the percentages seen later, we need only multiply repeatedly by the matrix $A$. Alternatively, note that $v^{(2)} = A^2 v^{(0)}$ and in general $v^{(n)} = A^n v^{(0)}$. We can compute

$$v^{(99)} = A^{99} v^{(0)} = \begin{bmatrix} 0.496525 \\ 0.309994 \\ 0.193481 \end{bmatrix}, \qquad v^{(100)} = A^{100} v^{(0)} = \begin{bmatrix} 0.496498 \\ 0.309999 \\ 0.193502 \end{bmatrix}.$$

Note that again the values in the simulation are close to these values. Also notice that these two values are very close to each other. Just as we conjectured from the simulation, we have reached a steady

state in which the percentages do not change very much from one time period to the next. If we kept applying $A$ we would find little change in future steps, even after thousands of steps, *e.g.*,

$$v^{(1000)} = A^{1000}v^{(0)} = \begin{bmatrix} 0.4961240310077407 \\ 0.3100775193798381 \\ 0.1937984496123990 \end{bmatrix}, \qquad v^{(10000)} = A^{10000}v^{(0)} = \begin{bmatrix} 0.4961240310076444 \\ 0.3100775193797780 \\ 0.1937984496123613 \end{bmatrix}.$$

As $n$ increases, the vectors $v^{(n)}$ are converging to a special vector $\hat{v}$ which has the property that

$$A\hat{v} = \hat{v}, \tag{5.6}$$

so that a step leaves the percentages unchanged. Note that the relation (5.6) means that this vector is an *eigenvector* of the matrix $A$, with eigenvalue $\lambda = 1$.

## 5.2 Review of eigenvalues and eigenvectors

Let $A$ be a square $m \times m$ matrix with real elements. Recall from linear algebra that a scalar value $\lambda$ is an eigenvalue of the matrix $A$ if there is a nonzero vector $r$ (the corresponding eigenvector) for which

$$Ar = \lambda r. \tag{5.7}$$

If $r$ is an eigenvector, then so is $\alpha r$ for any scalar value $\alpha$. The set of all eigenvectors corresponding to a given eigenvalue forms a linear subspace of $\mathbb{R}^m$.

In general $A$ has $m$ eigenvalues $\lambda_1, \ldots, \lambda_m$, which can be determined by finding the roots of a polynomial of degree $m$ in $\lambda$. This arises from the fact that if (5.7) holds, then

$$(A - \lambda I)r = 0$$

where $I$ is the $m \times m$ identity matrix. Since $r$ is a nonzero null-vector of this matrix, it must be singular. Hence its determinant is zero,

$$\det(A - \lambda I) = 0.$$

Computing the determinant gives the polynomial.

If the eigenvalues are *distinct* then the eigenvectors $r_i$ (corresponding to the different $\lambda_i$) will be linearly independent and so we can form a matrix $R$ with these vectors as columns which will be a nonsingular matrix. Then $Ar_i = \lambda_i r_i$ leads to

$$AR = R\Lambda \tag{5.8}$$

where $\Lambda$ is the diagonal matrix with the values $\lambda_i$ on the diagonal. (Note that the order is important! In general $AR \neq RA$ and $R\Lambda \neq \Lambda R$ since matrix multiplication is not commutative.)

Since $R$ is nonsingular, it has an inverse $R^{-1}$ and we can rewrite (5.8) as

$$A = R\Lambda R^{-1}. \tag{5.9}$$

(We might be able to do all this even if the eigenvalues are not distinct, but this case is a bit more subtle.)

One use of the decomposition (5.9) is to compute powers of the matrix $A$. Note that

$$\begin{aligned} A^2 &= (R\Lambda R^{-1})(R\Lambda R^{-1}) \\ &= R\Lambda(R^{-1}R)\Lambda R^{-1} \\ &= R\Lambda\Lambda R^{-1} \\ &= R(\Lambda^2)R^{-1} \end{aligned} \tag{5.10}$$

and similarly,

$$A^n = R\Lambda^n R^{-1} \tag{5.11}$$

for any $n$. This is easy to work with because $\Lambda^n$ is just the diagonal matrix with the values $\lambda_i^n$ on the diagonal.

In MATLAB it is easy to compute eigenvalues and eigenvectors. For example, for the matrix $A$ of (5.4), we can find the eigenvalues by doing:

```
>> p = .05; q = .08;

>> A = [ 1-p    q    0;     ...
          p   1-p-q  q;     ...
          0     p   1-q];

>> eig(A)

ans =

    1.0000
    0.8068
    0.9332
```

If we also want to find the eigenvectors, we can do

```
>> [R,Lambda] = eig(A)

R =

   -0.8050    0.4550    0.7741
   -0.5031   -0.8146   -0.1621
   -0.3144    0.3597   -0.6120


Lambda =

    1.0000         0         0
         0    0.8068         0
         0         0    0.9332
```

This returns the matrices $R$ and $\Lambda$.

## 5.3 Convergence to steady state

Note that one of the eigenvalues of $A$ computed above is equal to 1 and the others are both smaller than 1 in magnitude. As we will see, this means the Markov process converges to a steady state. Recall that

$$\begin{aligned} &|\lambda^n| \to 0 \quad \text{as} \quad n \to \infty \quad \text{if } |\lambda| < 1, \\ &|\lambda^n| = 1 \quad \text{for all } n \qquad \text{if } |\lambda| = 1, \\ &|\lambda^n| \to \infty \quad \text{as} \quad n \to \infty \quad \text{if } |\lambda| > 1, \end{aligned} \tag{5.12}$$

So as $n \to \infty$, the matrix $A^n$ approaches

```
>> R * [1 0 0; 0 0 0; 0 0 0] * inv(R)

ans =

    0.4961    0.4961    0.4961
    0.3101    0.3101    0.3101
    0.1938    0.1938    0.1938
```

as we can easily confirm, e.g.,

```
>> A^100

ans =

    0.4965    0.4960    0.4954
    0.3100    0.3101    0.3102
    0.1935    0.1939    0.1944
```

Now recall that when we multiply a matrix $A$ by a vector $v$, we are forming a new vector which is a linear combination of the columns of the matrix. The elements of $v$ give the weights applied to each column. If $a_j$ is the $j$'th column of $A$ and $v_j$ is the $j$'th element of $v$, then the vector $Av$ is

$$Av = a_1 v_1 + a_2 v_2 + a_3 v_3.$$

In particular, if $v^{(0)} = [1\ 0\ 0]'$, then $Av^{(0)}$ is just the first column of $A$. Similarly, $A^{100}v^{(0)}$ is the first column of the matrix $A^{100}$.

Note that for the example we have been considering, all three columns of $A^{100}$ are nearly the same, and for larger $n$ the three columns of $A^n$ would be even more similar. This has an important consequence: Suppose we started with different "initial conditions" $v^{(0)}$ instead of $v^{(0)} = [1\ 0\ 0]'$, which corresponded to all simulations being in State 1 to start. For example we might start with $v^{(0)} = [0.5\ 0.3\ 0.2]'$ if half the simulations start in State 1, 30% in State 2, and 20% in State 3. Then what would distribution of states would we see in the long run?

After $n$ steps the expected distribution is given by $A^n v^{(0)}$. After 100 steps,

```
>> v100 = A^100 * [.5 .3 .2]'

v100 =

    0.4961
    0.3101
    0.1938
```

The same steady state as before. Since all three columns of $A^{100}$ are essentially the same, and the elements of $v^{(0)}$ add up to 1, we expect $A^{100}v^{(0)}$ to be essentially the same for any choice of $v^{(0)}$.

Even though the *steady state* behavior is the same no matter how we start, the *transient behavior* over the first few days will be quite different depending on how we start. For example, starting with $v^{(0)} = [0.5\ 0.3\ 0.2]'$ instead of (5.1) gives

$$v^{(1)} = \begin{bmatrix} .499 \\ .302 \\ .199 \end{bmatrix}, \quad v^{(2)} = \begin{bmatrix} .4982 \\ .3036 \\ .1982 \end{bmatrix}, \quad v^{(3)} = \begin{bmatrix} .4976 \\ .3049 \\ .1975 \end{bmatrix}, \quad \text{etc.} \tag{5.13}$$

instead of (5.5).

Another way to look at the behavior of $v$ as we multiply repeatedly by $A$ is to decompose the initial vector $v^{(0)}$ as a linear combination of the eigenvectors of $A$:

$$v^{(0)} = c_1 r_1 + c_2 r_2 + c_3 r_3. \tag{5.14}$$

Since the vectors $r_1$, $r_2$ and $r_3$ are linearly independent, this can always be done for any $v^{(0)}$ and the solution $c = [c_1\ c_2\ c_3]'$ is unique. In fact the vector $c$ is the solution of the linear system

$$Rc = v^{(0)},$$

since this is exactly what (5.14) states, writing the matrix-vector multiply as a linear combination of the columns.

Now consider what happens if we multiply $v^{(0)}$ from (5.14) by the matrix $A$:

$$\begin{aligned} v^{(1)} = Av^{(0)} &= c_1 A r_1 + c_2 A r_2 + c_3 A r_3 \\ &= c_1 \lambda_1 r_1 + c_2 \lambda_2 r_2 + c_3 \lambda_3 r_3. \end{aligned} \tag{5.15}$$

Multiplying by $A$ again gives

$$\begin{aligned} v^{(2)} = Av^{(1)} &= c_1 \lambda_1 A r_1 + c_2 \lambda_2 A r_2 + c_3 \lambda_3 A r_3 \\ &= c_1 (\lambda_1)^2 r_1 + c_2 (\lambda_2)^2 r_2 + c_3 (\lambda_3)^2 r_3. \end{aligned} \tag{5.16}$$

In general we have

$$v^{(n)} = A^n v^{(0)} = c_1 (\lambda_1)^n r_1 + c_2 (\lambda_2)^n r_2 + c_3 (\lambda_3)^n r_3. \tag{5.17}$$

Again recall from (5.12), that as $n$ gets large, $(\lambda_j)^n \to 0$ for any eigenvalue that is smaller than one in magnitude. For this particular problem $\lambda_1 = 1$ while the others are smaller, and so

$$v^{(n)} \to c_1 r_1$$

for large $n$. This means the steady state is a multiple of $r_1$, which is again an eigenvector of $A$, as we already knew.

Note that the smaller $|\lambda|$ is, the faster $\lambda^n$ goes to zero. So from (5.17) we can see how quickly we expect to approximately reach a steady state. This depends also on how large the values $c_j$ are, which depends on the particular initial data chosen. If $v^{(0)}$ happens to be a steady state already, a multiple of $r_1$, then $c_2 = c_3 = 0$.

## 5.4 Other examples of Markov Chains

### 5.4.1 Breakdown of machines

A certain office machine has three states: Working, Temporarily Broken, and Permanently Broken. Each day it is in one of these states. The transition diagram between the states is shown in Figure 5.2. If the machine is working on Day 1, what is the probability that it is still functional (*i.e.*, not permanently broken) a year from now? (In this example we will consider at a single machine and talk about the probability that this machine is in each state. Alternatively we could consider a pool of many machines and consider what fraction of the machines are in each state on a given day.)

We can answer this question by computing the 365'th power of the transition matrix:

```
>> A = [.90  .6  0;  ...
       .095  .4  0;  ...
       .005   0  1];
```

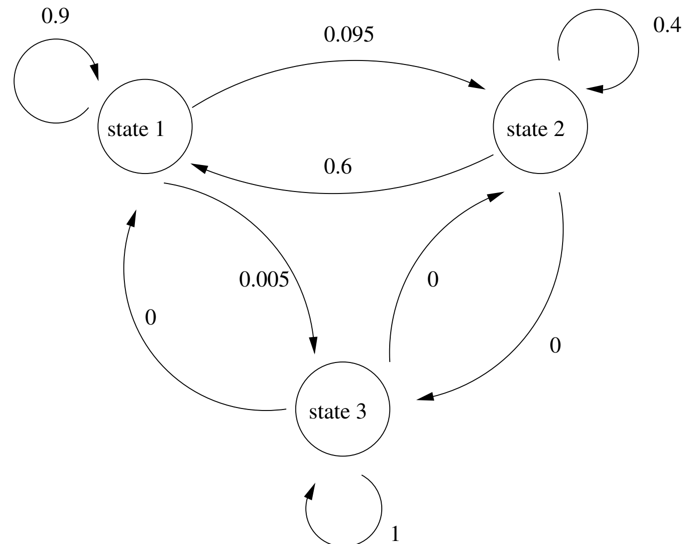



Figure 5.2: Transition diagram for the modeling machine breakdown. The three states are: State 1: Working, State 2: Temporarily Broken, and State 3: Permanently Broken.

```
>> A^365

ans =

    0.1779    0.1792         0
    0.0284    0.0286         0
    0.7937    0.7922    1.0000
```

The first column of this matrix gives the probability of being in each state on Day 366 (with our assumption that it is working on Day 1, so $v^{(0)} = [1\ 0\ 0]$). We see that the probability that the machine is permanently broken in 0.7937 and so there is about a 21% chance that the machine is still functional.

Note that the probabilities would be about the same if we'd started in the Temporarily Broken state on Day 1 (by looking at the second column of this matrix). Why does this make sense? Why is the third column so different?

How about after 5 years, or 1825 days? We find

```
>> A^1825

ans =

    0.0003    0.0003         0
    0.0001    0.0001         0
    0.9996    0.9996    1.0000
```

so by this time the machine is almost surely dead whatever state we started in.

For this Markov chain the steady state solution is the vector $[0\ 0\ 1]'$, as we can see by computing the eigenvalues and eigenvectors:

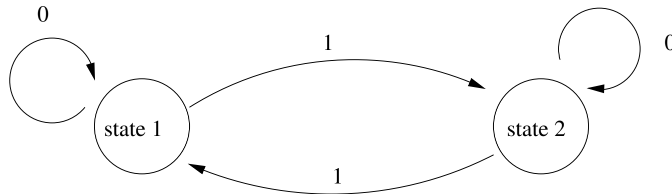
```
>> [R, Lambda] = eig(A)
```

```
R =

         0    0.7096   -0.6496
         0   -0.7045   -0.1036
    1.0000   -0.0051    0.7532


Lambda =

    1.0000         0         0
         0    0.3043         0
         0         0    0.9957
```

State 3 (Permanently Broken) is called an *absorbing state*. No matter what state we start in, we will eventually end up in State 3 with probability 1, and once in State 3 we can never leave.

### 5.4.2 Work schedules

Suppose the XYZ Factory is operating every day and has a work schedule where each employee works every other day. Say that an employee is in State 1 on work days and in State 2 on off days. Then the transition diagram is simply




since with probability 1 the employee switches states each day. In this case the transition matrix is

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

If 70% of the employees are working the first day then we have

$$v^{(0)} = \begin{bmatrix} 0.7 \\ 0.3 \end{bmatrix}, \quad v^{(1)} = \begin{bmatrix} 0.3 \\ 0.7 \end{bmatrix}, \quad v^{(2)} = \begin{bmatrix} 0.7 \\ 0.3 \end{bmatrix}, \quad v^{(3)} = \begin{bmatrix} 0.3 \\ 0.7 \end{bmatrix},$$

and so on. In this case we do not reach a single steady state, but we do have a very regular pattern of cycling back and forth between two states. This can again be understood by looking at the eigenvalues and eigenvectors of the matrix:

```
>> A = [0  1;  ...
        1  0];
```

```
>> [R, Lambda] = eig(A)

R =

    0.7071    0.7071
   -0.7071    0.7071

Lambda =

    -1     0
     0     1
```

Both eigenvalues have magnitude 1. If we decompose $v^{(0)}$ in terms of the eigenvectors,

$$v^{(0)} = c_1 r_1 + c_2 r_2$$

then we have

$$v^{(n)} = c_1(-1)^n r_1 + c_2 r_2.$$

The first terms flips sign every day and accounts for the oscillation.

In the special case where $v^{(0)}$ is a multiple of $r_2$, we would expect $c_1 = 0$ and in this case the solution should be truly steady. Since the elements of $v$ must sum to 1, the only multiple of $r_2$ we can consider is

$$v^{(0)} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}.$$

This makes sense: if the factory starts with 50% of employees working on the first day then 50% will be working every day.

### 5.4.3 The fish tank inventory problem

The Monte Carlo simulation shown in Figure 4.1 gave results indicating that with this particular ordering strategy and set of parameters, we could serve about 60% of the customers who come looking for a 150 gallon fish tank. We can derive this using a Markov chain. For the situation modeled in this program the store can be assumed to always be in one of six states each day:

1. A customer arrives and there's a tank in stock,
2. A customer arrives, there's none in stock but one due to arrive the next day,
3. A customer arrives, there's none in stock but one due to arrive in two days.
4. No customer arrives, but there's a tank in stock,
5. No customer arrives, there's none in stock but one due to arrive the next day,
6. No customer arrives, there's none in stock but one due to arrive in two days.

Working out the probabilities of moving from each state to each of the others, we find the transition matrix

$$A = \begin{bmatrix} 0 & p & 0 & p & p & 0 \\ 0 & 0 & p & 0 & 0 & p \\ p & 0 & 0 & 0 & 0 & 0 \\ 0 & 1-p & 0 & 1-p & 1-p & 0 \\ 0 & 0 & 1-p & 0 & 0 & 1-p \\ 1-p & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{5.18}$$

where $p = 1/3$ is the probability of a customer arriving on any given day. This matrix has three non-zero eigenvalues including $\lambda = 1$. The eigenvector corresponding to $\lambda = 1$, when normalized so the elements sum to 1, is found to be

$$\hat{v} = \frac{1}{15}\begin{bmatrix} 3 \\ 1 \\ 1 \\ 6 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 0.2000 \\ 0.0667 \\ 0.0667 \\ 0.4000 \\ 0.1333 \\ 0.1333 \end{bmatrix}$$

The components of this vector are the expected fraction of days that the store is in each of the six possible states, in the long term steady state. From this we can see that the fraction of time a customer arrives and is served can be computed from the components of $\hat{v}$ as

$$\text{fraction served} = \hat{v}_1/(\hat{v}_1 + \hat{v}_2 + \hat{v}_3) = \frac{3}{3+1+1} = 0.6.$$

Exactly 60%, as observed in the Monte Carlo simulation.

### 5.4.4 Card shuffling

Another sort of application of Markov chains is to model a *riffle shuffle* of a deck of cards. This application received a great deal of attention some years back when it was used to determine how many shuffles are necessary to randomize a deck of cards [Kol90]. The answer turns out to be 7.

The model was developed by Gilbert[Gil55], Shannon and by Reeds [Ree81], and it was used by Bayer and Diaconis [BD92] to answer the question about randomizing an initially sorted deck of cards. The riffle shuffle is modeled as follows: A deck of $n$ cards is divided into two heaps, with the probability of a heap containing $k$ cards being given by a binomial distribution, $\binom{n}{k}/2^n$. The heaps are then riffled together into a single pile, with the probability of dropping the next card from a particular heap being proportional to the number of cards in that heap.

This is generally considered to be a reasonable model of the way humans perform a riffle shuffle, although this has been debated. With this model, the process of card shuffling is represented as a Markov chain. Given the current ordering of the cards, one can write down the probability of achieving any other ordering after the next shuffle. Unfortunately, with a deck of 52 cards there are 52! possible orderings, and this would seem to render the problem intractable from a computational point of view.

The great contribution of Bayer and Diaconis was to notice that the number of "states" of this system could be drastically reduced. Given only the number of *rising sequences* in the current ordering, one could determine the probability of any given number of rising sequences after the next shuffle. A *rising sequence* is defined as a maximal sequence of consecutively numbered cards found (perhaps interspersed with other cards) during one pass through the deck. For example, the ordering 146253 has three rising sequences: 123, 45, and 6. With this observation the number of states of the system is reduced from $n!$ to $n$, and for a deck of $n = 52$ cards it now becomes an easy problem to solve numerically.

An excellent description of the problem and solution from more of a linear algebra point of view can be found in Jónsson and Trefethen [JT98], where they also supply a MATLAB code that forms the probability transition matrix $P$ and a related matrix $A = P - P^\infty$ whose $k$th power gives the difference between the state of the system after $k$ shuffles and that after infinitely many shuffles. A version of their code is available on the class web page. The entries of the probability transition matrix are

$$P_{ji} = 2^{-n}\binom{n+1}{2i-j}\frac{\alpha_j}{\alpha_i},$$

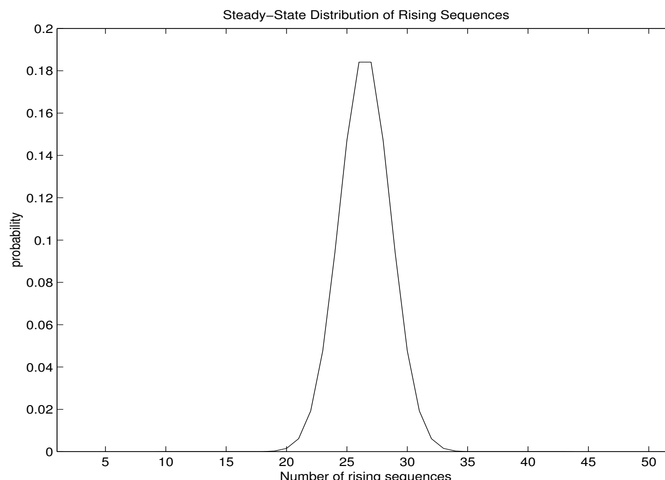



Figure 5.3: Steady-state distribution of number of rising sequences

where $\alpha_j$ denotes the number of permutations of $\{1, \ldots, n\}$ that have exactly $j$ rising sequences. The $\alpha_j$'s are known as *Eulerian numbers*, and they satisfy the recurrence:

$$A_{r,k} = kA_{r-1,k} + (r-k+1)A_{r-1,k-1},$$

where $A_{11} = 1$, $A_{1k} = 0$ for $k \neq 1$, and $\alpha_j = A_{nj}$. See [Slo00] for more details on this sequence.

The eigenvalues of the probability transition matrix are known to be $2^{-j}$, $j = 0, 1, \ldots, n-1$, and the eigenvector corresponding to eigenvalue 1 is known as well:

$$v = (\alpha_1, \ldots, \alpha_n)^T / n!$$

The entries of this vector are the probabilities of each possible number of rising sequences after the deck has been shuffled infinitely many times. The vector $v$ is plotted in Figure 5.3. It can be determined from the entries of $v$ that, for example, with probability .9999 a well-shuffled deck has between 19 and 34 rising sequences.

This begins to give us a clue as to why the system behaves as it does. If the deck is initially sorted in increasing order — so that there is one rising sequence and the initial state vector is $s_0 = (1, 0, \ldots, 0)^T$ — then after one shuffle there will be at most two rising sequences. Do you see why? When you take a deck with two rising sequences, cut it in two and riffle the two portions together, the resulting ordering can have at most four rising sequences, etc. After $j$ shuffles there are at most $2^j$ rising sequences, so that for $j \leq 4$ it is easy to determine that the ordering is not random. Try the experiment. Shuffle a deck of cards four times and count the number of rising sequences. It will be no more than 16. Then shuffle several more times (at least three more times) and with very high probability you will count at least 19 rising sequences.

Figure 5.4 shows a plot of the difference between the state vector after $k$ shuffles and that after infinitely many shuffles: $\frac{1}{2}\sum_{i=1}^{n} |v_i - (P^k s_0)_i|$. This is called the ***total variation*** norm and is the measure used by Bayer and Diaconis. The deck is deemed to be sufficiently well shuffled when this difference drops below .5. Notice that this happens after $k = 7$ shuffles.

## 5.5 General properties of transition matrices

Transition matrices are special types of matrices which have some general properties which we summarize here:

- Every element in a transition matrix must lie between 0 and 1 since it is a probability.

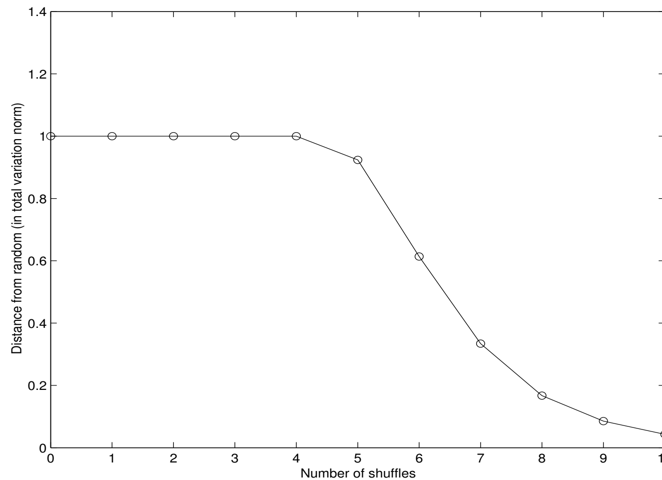



Figure 5.4: Convergence toward the steady-state in total variation norm

- Each column of a transition matrix sums to 1. This is because the elements in the $j$'th column represent the probabilities of moving from State $j$ to each of the other states. Since we must move *somewhere*, all of these probabilities must add up to 1.

- On the other hand the rows do not need to sum to 1. (Note: In many books transition matrices are defined differently, with $a_{ij}$ being the probability of moving from State $i$ to State $j$ instead of being the probability of moving from State $j$ to State $i$ as we have used. In this case the transition matrix is the transpose of ours, and the rows sum to 1 instead of the columns.)

- If $w$ is the row vector with all components equal to 1, then $wA = w$. This is just a restatement of the fact that all columns must sum to 1. But this also shows that $w$ is a left eigenvector of $A$ with eigenvalue $\lambda = 1$. This proves that $\lambda = 1$ is always an eigenvalue of any transition matrix. (Recall that in general the left eigenvectors of $A$, row vectors $\ell$ for which $\ell A = \lambda \ell$, are different from the right eigenvectors, but the eigenvalues are the same. In fact the left eigenvectors $\ell$ are the rows of the matrix $R^{-1}$, the inverse of the matrix of right eigenvectors.)

- It can also be shown that all eigenvalues of a transition matrix satisfy $|\lambda| \leq 1$, so some components of an eigen-expansion such as (5.14) decay while others have constant magnitude, but none can grow exponentially.

- If $\lambda_j = 1$ then the eigenvector $r_j$ can be used to find a potential steady state of the system. Since the elements of $v$ must sum to 1, we must normalize $r_j$ to have this sum. Recall that any scalar multiple $c_j r_j$ is also an eigenvector, so we choose $c_j = 1/\sum_i r_{ij}$ where $r_{ij}$ are the elements of the vector $r_j$. For example, if we use Matlab and obtain an eigenvector such as $r_1 = [-0.8050 \ -0.5031 \ -0.3144]'$, we can compute the normalized eigenvector as

$$\hat{v} = \frac{r_1}{-0.8050 - 0.5031 - 0.3144} = \begin{bmatrix} 0.4961 \\ 0.3101 \\ 0.1938 \end{bmatrix}$$

  This is the expected steady state.

- Note that normalizing an eigenvector gives a reasonable result only if all the elements of the eigenvector have the same sign. What can you hypothesize about eigenvectors of $A$ corresponding to eigenvalues $\lambda_j$ with $|\lambda_j| < 1$?

## 5.6 Ecological models and Leslie Matrices

Similar matrix techniques can be used in other situations that are not exactly "Markov chains". As an example, suppose we want to study the population of a certain species of bird. These birds are born in the spring and live at most 3 years, and we will keep track of the population just before breeding. There will be 3 types of birds: Age 0 (born the previous spring), Age 1, and Age 2. Let $v_0^{(n)}$, $v_1^{(n)}$ and $v_2^{(n)}$ represent the number of females in each age class just before breeding in Year $n$. To model how the population changes we need to know:

- Survival rates. Suppose 20% of Age 0 birds survive to the next spring, and 50% of Age 1 birds survive to become Age 2.
- Fecundity rates. Suppose females that are 1 year old produce a clutch of a certain size, of which 3 females are expected to survive to the next breeding season. Females that are 2 years old lay more eggs and suppose 6 females are expected to survive to the next spring.

Then we have the following model:

$$\begin{aligned} v_0^{(n+1)} &= 3v_1^{(n)} + 6v_2^{(n)} \\ v_1^{(n+1)} &= 0.2v_0^{(n)} \\ v_2^{(n+1)} &= 0.5v_1^{(n)} \end{aligned} \tag{5.19}$$

In matrix-vector form:

$$v^{(n+1)} = \begin{bmatrix} 0 & 3 & 6 \\ 0.2 & 0 & 0 \\ 0 & 0.5 & 0 \end{bmatrix} \begin{bmatrix} v_0^{(n)} \\ v_1^{(n)} \\ v_2^{(n)} \end{bmatrix}. \tag{5.20}$$

Note that in this case the columns of the matrix do not sum to 1, and some of the entries do not represent probabilities.

However, we can predict the future population by iterating with this matrix just as we did using transition matrices for a Markov chain, and the eigenvalues and eigenvectors will give useful information about what we expect to see in the long run. This form of matrix, based on survival rates and fecundities, is called a *Leslie matrix*. (See [Jef78, SK87] for more discussion.)

Suppose we start with a population of 3000 females, 1000 of each age. Then we find

$$v^{(0)} = \begin{bmatrix} 1000 \\ 1000 \\ 1000 \end{bmatrix}, \quad v^{(1)} = \begin{bmatrix} 9000 \\ 200 \\ 500 \end{bmatrix}, \quad v^{(2)} = \begin{bmatrix} 3600 \\ 1800 \\ 100 \end{bmatrix}, \quad v^{(3)} = \begin{bmatrix} 6000 \\ 720 \\ 900 \end{bmatrix},$$

and so on. If we plot the population in each age group as a function of Year, we see what is shown in Figure 5.5(a). Clearly the population is growing exponentially. Figure 5.5(b) shows the fraction of the population which is in each age class as a function of year. This settles down to a steady state.

After 20 years the population is $v^{(20)} = [20833\ 3873\ 1797]'$ while the percentage in each age class is $[0.7861\ 0.1461\ 0.0678]'$. We can compute that between the last two years the total population grew by a factor of 1.0760.

Since $v^{(20)} = A^{20}v^{(0)}$, we can predict all this from the eigenvalues and eigenvectors of $A$. We find

```
A =

         0    3.0000    6.0000
    0.2000         0         0
         0    0.5000         0
```

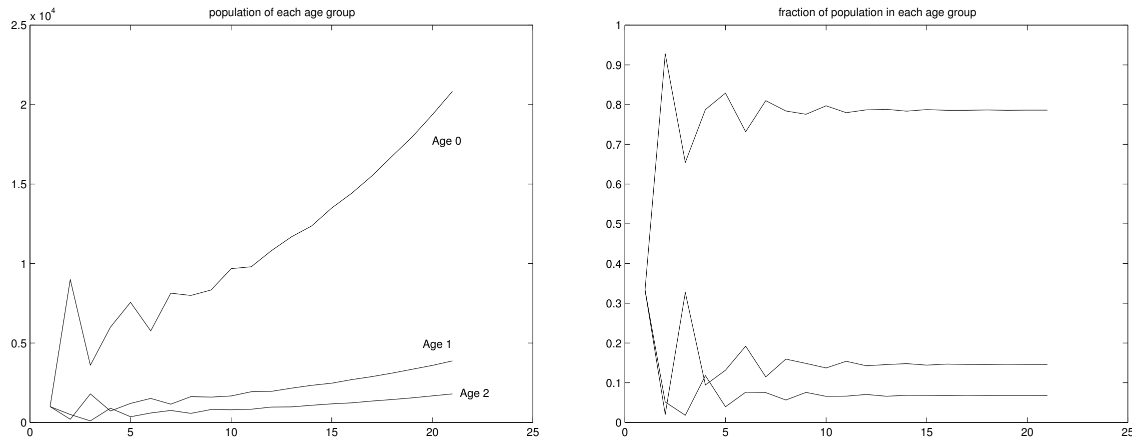



Figure 5.5: Bird population with $a_{32} = 0.5$ and initial data $v^{(0)} = [1000\ 1000\ 1000]'$.

```
>> [R,Lambda] = eig(A)

R =

   0.9796             -0.6990 - 0.6459i  -0.6990 + 0.6459i
   0.1821              0.0149 + 0.2545i   0.0149 - 0.2545i
   0.0846              0.1110 - 0.1297i   0.1110 + 0.1297i


Lambda =

   1.0759                   0                  0
        0             -0.5380 + 0.5179i        0
        0                   0            -0.5380 - 0.5179i

>> R(:,1)/sum(R(:,1))

ans =

    0.7860
    0.1461
    0.0679
```

We see that $\lambda_1 = 1.0759$ is greater than 1 and accurately predicts the growth rate of the total population. The other two eigenvalues are complex and have magnitude 0.7468, so the corresponding components in the initial vector $v^{(0)}$ die out. This *transient behavior* which results from the fact that the initial population distribution (equal number of each age) is not the steady state distribution. The first eigenvector $r_1$, normalized to have sum 1, gives the fraction expected in each age class in the steady state. If we had started with a population that was distributed this way, then we would see smooth behavior from the beginning with no transient. (See the example in the next section.)

### 5.6.1 Harvesting strategies

We can use this model to investigate questions such as the following. Suppose we want to harvest a certain fraction of this population, either to control the exponential growth or as a food source (or both). How much should we harvest to control the growth without causing extinction?

Harvesting will decrease the survival rates. We might like to decrease the rates to a point where the dominant eigenvalue is very close to 1 in value. If we decrease the rate too much, then all eigenvalues will be less than 1 and the population size will decrease exponentially to extinction.

For example, suppose we harvest an additional 20% of the 1-year olds, so the survival rate $a_{32}$ falls from 0.5 to 0.3. Then we find that the largest eigenvalue of $A$ is 0.9830, giving the graphs shown in Figure 5.6.

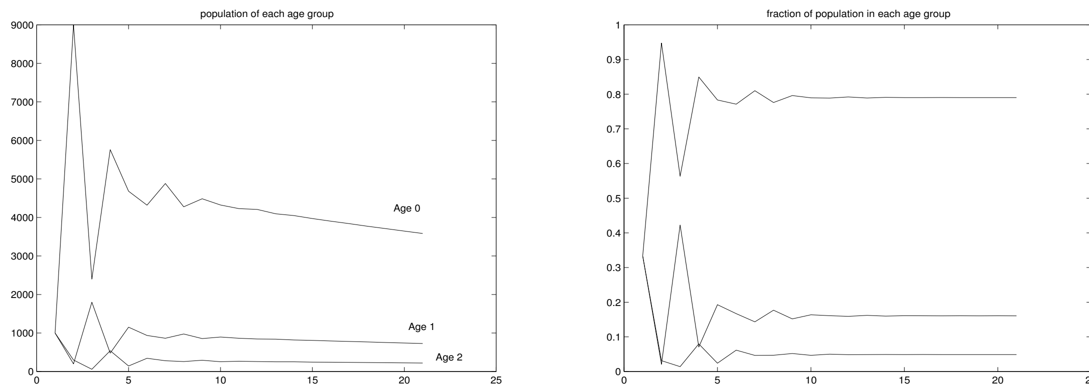



Figure 5.6: Bird population with $a_{32} = 0.3$ and initial data $v^{(0)} = [1000\ 1000\ 1000]'$.

Of course it might be more reasonable to assume the population starts out with a steady-state distribution of ages. For the new matrix the relevant scaled eigenvector is $r_1 = [.7902\ 0.1608\ 0.0491]'$ and for a population of 3000 birds this suggests we should take $v^{(0)} = 3000r_1 = [2371\ 482\ 147]'$. Then we obtain the boring graphs shown in Figure 5.7.

Note that we now have exponential decay of the population and eventual extinction. However, over the 20-year span shown here the total population is predicted to decrease from 3000 to $3000(0.983)^{20} = 2129$ and probably our model is not even accurate out this far. (Assuming it's accurate at all!) To judge this, one would have to investigate what assumptions the model is based on and how far in the future these assumptions will be valid.

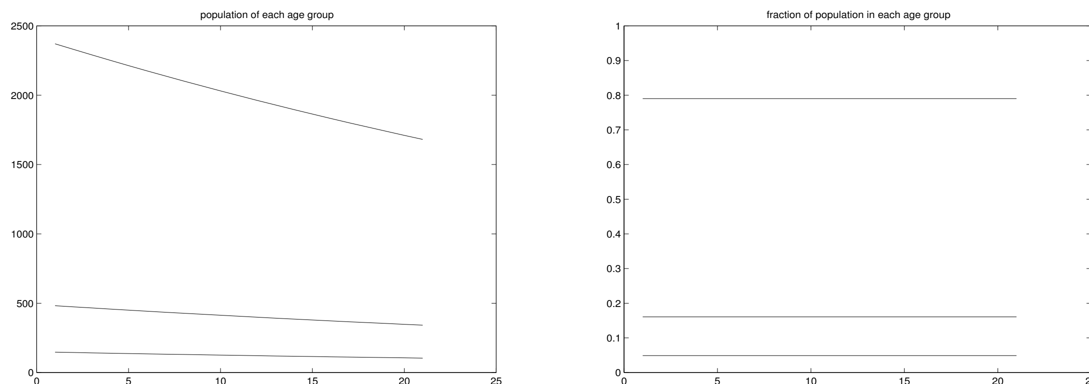



Figure 5.7: Bird population with $a_{32} = 0.3$ and initial data having the expected steady-state distribution of ages.

# Chapter 6

# Linear Programming

## 6.1 Introduction

Section 6.1 of the notes is adapted from an introduction to linear programming written by Professor J. Burke for Math 407. Some of the discussion in the original notes which introduces duality theory and sensitivity analysis has been eliminated. Burke's original notes can be found on the webpage

`http://www.math.washington.edu/~burke/crs/407/HTML/notes.html`

Other introductions to linear programming can be found in many references, including [HL95] and [Chv83].

### 6.1.1 What is optimization?

A mathematical optimization problem is one in which some function is either maximized or minimized relative to a given set of alternatives. The function to be minimized or maximized is called the *objective function* and the set of alternatives is called the feasible region (or constraint region). In this course, the feasible region is always taken to be a subset of $\mathbb{R}^n$ (real $n$-dimensional space) and the objective function is a function from $\mathbb{R}^n$ to $\mathbb{R}$.

We further restrict the class of optimization problems that we consider to linear programming problems (or LPs). An LP is an optimization problem over $\mathbb{R}^n$ wherein the objective function is a linear function, that is, the objective has the form

$$c_1x_1 + c_2x_2 + \cdots + c_nx_n$$

for some $c_i \in \mathbb{R}$ $i = 1, \ldots, n$, and the feasible region is the set of solutions to a finite number of linear inequality and equality constraints, of the form

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \leq b_i \qquad i = 1, \ldots, s$$

and

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n = b_i \qquad i = s+1, \ldots, m.$$

Linear programming is an extremely powerful tool for addressing a wide range of applied optimization problems. A short list of application areas is resource allocation, production scheduling, warehousing, layout, transportation scheduling, facility location, flight crew scheduling, parameter estimation, ....

### 6.1.2 An Example

To illustrate some of the basic features of LP, we begin with a simple two-dimensional example. In modeling this example, we will review the four basic steps in the development of an LP model:

1. Determine and label the *decision variables*.

2. Determine the objective and use the decision variables to write an expression for the *objective function*.

3. Determine the *explicit constraints* and write a functional expression for each of them.

4. Determine the *implicit constraints*.

PLASTIC CUP FACTORY

A local family-owned plastic cup manufacturer wants to optimize their production mix in order to maximize their profit. They produce personalized beer mugs and champaign glasses. The profit on a case of beer mugs is \$25 while the profit on a case of champaign glasses is \$20. The cups are manufactured with a machine called a plastic extruder which feeds on plastic resins. Each case of beer mugs requires 20 lbs. of plastic resins to produce while champaign glasses require 12 lbs. per case. The daily supply of plastic resins is limited to at most 1800 pounds. About 15 cases of either product can be produced per hour. At the moment the family wants to limit their work day to 8 hours.

We will model the problem of maximizing the profit for this company as an LP. The first step in our modeling process is to determine the *decision variables*. These are the variables that represent the quantifiable decisions that must be made in order to determine the daily production schedule. That is, we need to specify those quantities whose values completely determine a production schedule and its associated profit. In order to determine these quantities, one can ask the question "If I were the plant manager for this factory, what must I know in order to implement a production schedule?" The best way to determine the decision variables is to put oneself in the shoes of the decision maker and then ask the question "What do I need to know in order to make this thing work?" In the case of the plastic cup factory, everything is determined once it is known how many cases of beer mugs and champaign glasses are to be produced each day.

*Decision Variables*:

$B = \#$ of cases of beer mugs to be produced daily.

$C = \#$ of cases of champaign glasses to be produced daily.

You will soon discover that the most difficult part of any modeling problem is the determination of decision variables. Once these variables are correctly determined then the remainder of the modeling process usually goes smoothly.

After specifying the decision variables, one can now specify the problem objective. That is, one can write an expression for the objective function.

*Objective Function*:

Maximize profit where profit $= 25B + 20C$

The next step in the modeling process is to express the feasible region as the solution set of a finite collection of linear inequality and equality constraints. We separate this process into two steps:

1. determine the explicit constraints, and

2. determine the implicit constraints.

The explicit constraints are those that are explicitly given in the problem statement. In the problem under consideration, there are explicit constraints on the amount of resin and the number of work hours that are available on a daily basis.

*Explicit Constraints*:

resin constraint: $20B + 12C \leq 1800$

work hours constraint: $\frac{1}{15}B + \frac{1}{15}C \leq 8$.

This problem also has other constraints called implicit constraints. These are constraints that are not explicitly given in the problem statement but are present nonetheless. Typically these constraints are associated with "natural" or "common sense" restrictions on the decision variable. In the cup factory problem it is clear that one cannot have negative cases of beer mugs and champaign glasses. That is, both $B$ and $C$ must be non-negative quantities.

*Implicit Constraints*:

$$0 \leq B, \qquad 0 \leq C.$$

The entire model for the cup factory problem can now be succinctly stated as

$$\begin{array}{rl} \mathcal{P} \;:\; & \max 25B + 20C \\ & \text{subject to } \begin{array}{rcl} 20B + 12C & \leq & 1800 \\ \frac{1}{15}B + \frac{1}{15}C & \leq & 8 \\ 0 & \leq & B, C \end{array} \end{array}$$

Since this problem is two dimensional it is possible to provide a graphical solution. The first step toward a graphical solution is to graph the feasible region. To do this, first graph
the line associated with each of the linear inequality constraints. Then determine on which side of each of these lines the feasible region must lie (don't forget the implicit constraints!). Once the correct side is determined it is helpful to put little arrows on the line to remind yourself of the correct side. Then shade in the resulting feasible region.

The next step is to draw in the vector representing the gradient of the objective function at the origin. Since the objective function has the form

$$f(x_1, x_2) = c_1x_1 + c_2x_2,$$

the gradient of $f$ is the same at every point in $\mathbb{R}^2$;

$$\nabla f(x_1, x_2) = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}.$$

Recall from calculus that the gradient always points in the direction of increasing function values. Moreover, since the gradient is constant on the whole space, the level sets of $f$ associated with different function values are given by the lines perpendicular to the gradient. Consequently, to obtain the location of the point at which the objective is maximized we simply set a ruler perpendicular to the gradient and then move the ruler in the direction of the gradient until we reach the last point (or points) at which the line determined by the ruler intersects the feasible region. In the case of the cup factory problem this gives the solution to the LP as $\begin{bmatrix} B \\ C \end{bmatrix} = \begin{bmatrix} 45 \\ 75 \end{bmatrix}$

We now recap the steps followed in the solution procedure given above:

**Step 1:** Graph each of the linear constraints and indicate on which side of the constraint the feasible region must lie. Don't forget the implicit constraints!

**Step 2:** Shade in the feasible region.

**Step 3:** Draw the gradient vector of the objective function.

**Step 4:** Place a straightedge perpendicular to the gradient vector and move the straightedge either in the direction of the gradient vector for maximization, or in the opposite direction of the gradient vector for minimization to the last point for which the straightedge intersects the feasible region. The set of points of intersection between the straightedge and the feasible region is the set of solutions to the LP.

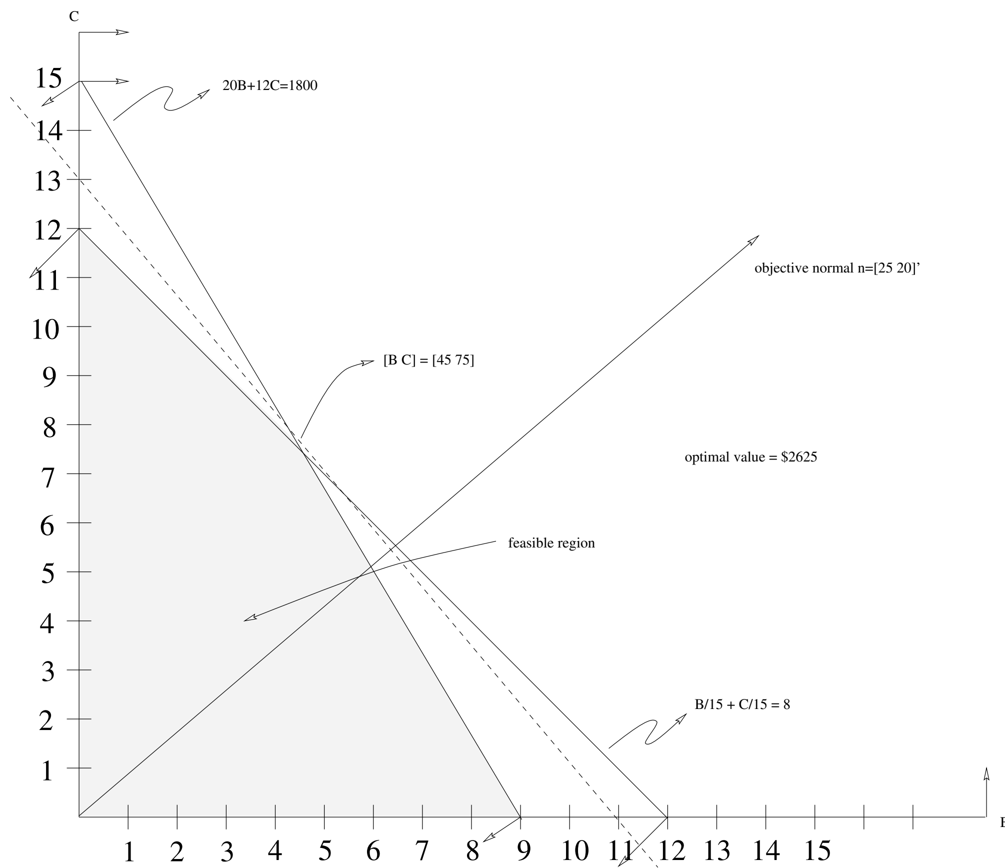



Figure 6.1:

The solution procedure described above for two dimensional problems reveals a great deal about the geometric structure of LPs that remains true in $n$ dimensions. We will explore this geometric structure more fully as the course evolves. But for the moment, we continue to study this 2 dimensional LP to see what else can be revealed about the structure of this problem.

Before leaving this section, we make a final comment on the modeling process described above. We emphasize that there is not one and only one way to model the Cup Factory problem, or any problem for that matter. In particular, there are many ways to choose the decision variables for this problem. Clearly, it is sufficient for the shop manager to know how many hours each days should be devoted to the manufacture of beer mugs and how many hours to champaign glasses. From this information everything else can be determined. For example, the number of cases of beer mugs that get produced is 15 times the number of hours devoted to the production of beer mugs. Therefore, as you can see there are many ways to model a given problem. But in the end, they should all point to the same optimal process.

### 6.1.3 LPs in Standard Form

Recall that a linear program is a problem of maximization or minimization of a linear function subject to a finite number of linear inequality and equality constraints. This general definition leads to an enormous variety of possible formulations. In this section we propose one fixed formulation for the purposes of developing an algorithmic solution procedure. We then show that every LP can be recast in this form. We say that an LP is in *standard form* if it has the form

$$\begin{array}{llll} \mathcal{P}: & \text{maximize} & c_1x_1 + c_2x_2 + \cdots + c_nx_n & \\ & \text{subject to} & a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \quad \leq & b_i \text{ for } i = 1, 2, \ldots, m \\ & & 0 \leq x_j \text{ for } j = 1, 2, \ldots, n\ . & \end{array}$$

Using matrix notation, we can rewrite this LP as

$$\begin{array}{lll} \mathcal{P}: & \text{maximize} & c^Tx \\ & \text{subject to} & Ax \leq b \\ & & 0 \leq x\ , \end{array}$$

where the inequalities $Ax \leq b$ and $0 \leq x$ are to be interpreted componentwise.

#### Transformation to Standard Form

Every LP can be transformed to an LP in standard form. This process usually requires a transformation of variables and occasionally the addition of new variables. In this section we provide a step-by-step procedure for transforming any LP to one in standard form.

*minimization $\rightarrow$ maximization*

To transform a minimization problem to a maximization problem just multiply the objective function by $-1$.

*linear inequalities*

If an LP has an inequality constraint of the form

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \geq b_i.$$

This inequality can be transformed to one in standard form by multiplying the inequality through by $-1$ to get

$$-a_{i1}x_1 - a_{i2}x_2 - \cdots - a_{in}x_n \leq -b_i.$$

*linear equation*

The linear equation

$$a_{i1}x_1 + \cdots + a_{in}x_n = b_i$$

can be written as two linear inequalities

$$a_{i1}x_1 + \cdots + a_{in}x_n \le b_i$$

and

$$a_{i1}x_1 + \cdots + a_{in}x_n \ge b_i.$$

The second of these inequalities can be transformed to standard form by multiplying through by $-1$.

*variables with lower bounds*

If a variable $x_i$ has lower bound $l_i$ which is not zero ($l_i \le x_i$), one obtains a non-negative variable $w_i$ with the substitution

$$x_i = w_i + l_i.$$

In this case, the bound $l_i \le x_i$ is equivalent to the bound $0 \le w_i$.

*variables with upper bounds*

If a variable $x_i$ has an upper bound $u_i$ ($x_i \le u_i$) one obtains a non-negative variable $w_i$ with the substitution

$$x_i = u_i - w_i.$$

In this case, the bound $x_i \le u_i$ is equivalent to the bound $0 \le w_i$.

*variables with interval bounds*

An interval bound of the form $l_i \le x_i \le u_i$ can be transformed into one non-negativity constraint and one linear inequality constraint in standard form by making the substitution

$$x_i = w_i + l_i.$$

In this case, the bounds $l_i \le x_i \le u_i$ are equivalent to the constraints

$$0 \le w_i \qquad \text{and} \qquad w_i \le u_i - l_i.$$

*free variables*

Sometimes a variable is given without any bounds. Such variables are called free variables. To obtain standard form every free variable must be replaced by the difference of two non-negative variables. That is, if $x_i$ is free, then we get

$$x_i = u_i - v_i$$

with $0 \le u_i$ and $0 \le v_i$.

To illustrate the ideas given above, we put the following LP into standard form.

$$\begin{array}{lrcrcrcrcr}
\text{minimize} & 3x_1 & - & x_2 & & & & & & \\
\text{subject to} & -x_1 & + & 6x_2 & - & x_3 & + & x_4 & \ge & -3 \\
 & & & 7x_2 & & & + & x_4 & = & 5 \\
 & & & & & x_3 & + & x_4 & \le & 2
\end{array}$$

$$-1 \le x_2, x_3 \le 5, -2 \le x_4 \le 2.$$

First, we rewrite the objective as

$$\text{maximize } -3x_1 + x_2.$$

Next we replace the first inequality constraint by the constraint

$$x_1 - 6x_2 + x_3 - x_4 \leq 3.$$

The equality constraint is replaced by the two inequality constraints

$$\begin{array}{rcrcr} 7x_2 & + & x_4 & \leq & 5 \\ -7x_2 & - & x_4 & \leq & -5. \end{array}$$

Observe that the variable $x_1$ is free, so we replace it by

$$x_1 = y_1 - y_2 \text{ with } 0 \leq y_1, 0 \leq y_2.$$

The variable $x_2$ has a non-zero lower bound so we replace it by

$$x_2 = y_3 - 1 \text{ with } 0 \leq y_3.$$

The variable $x_3$ is bounded above, so we replace it by

$$x_3 = 5 - y_4 \text{ with } 0 \leq y_4.$$

The variable $x_4$ is bounded below and above so we replace it by

$$x_4 = y_5 - 2 \text{ with } 0 \leq y_5 \text{ and } y_5 \leq 4.$$

Putting it all together we get the following LP in standard form:

$$\begin{array}{lrcrcrcrcrcr} \text{maximize} & -3y_1 & + & 3y_2 & + & y_3 & & & & & & \\ \text{subject to} & y_1 & - & y_2 & - & 6y_3 & - & y_4 & - & y_5 & \leq & -10 \\ & & & & & 7y_3 & & & + & y_5 & \leq & 14 \\ & & & & - & 7y_3 & & & - & y_5 & \leq & -14 \\ & & & & & & - & y_4 & + & y_5 & \leq & -1 \\ & & & & & & & & & y_5 & \leq & 4 \end{array}$$

$$0 \leq y_1, y_2, y_3, y_4, y_5.$$

## 6.2 A pig farming model

Suppose a pig farmer can choose between two different kinds of feed for his pigs: corn or alfalfa. He can feed them all of one or the other or some mixture. The pigs must get at least 200 units of carbohydrates and at least 180 units of protein in their daily diet. The farmer knows that:

- corn has 100 units/kg of carbohydrate and 30 units/kg of protein
- alfalfa has 40 units/kg of carbohydrate and 60 units/kg of protein.

Here are a few questions the farmer might be faced with in choosing a good diet for the pigs:

1. If the farmer uses only corn, how many kg/day does he need for each pig?
2. If the farmer uses only alfalfa, how many kg/day does he need for each pig?
3. Suppose corn and alfalfa each cost 30 cents / kg. How expensive would each of the above diets be?
4. With these prices, is there some mixture of the two grains that would satisfy the dietary requirements and be cheaper?
5. What is the best mix if the price of alfalfa goes up to $1/kg? What if it goes up only to 60 cents / kg?

The problem of choosing the mix which minimizes cost can be set up as a linear programming problem

$$\begin{aligned} \text{minimize} \quad & c_1x_1 + c_2x_2 \\ \text{subject to} \quad & \\ & 100x_1 + 40x_2 \geq 200 \\ & 30x_1 + 60x_2 \geq 180 \\ & x_1,\ x_2 \geq 0. \end{aligned} \tag{6.1}$$

where $x_1$ and $x_2$ are the quantity of corn and alfalfa to use (in kilograms) and $c_1$ and $c_2$ are the cost per kilogram of each. Multiplying the objective function and constraints by $-1$ would allow us to put this in the standard form discussed above.

Linear programming problems can be solved in MATLAB, which requires a slightly different standard form, since it solves a minimization problem rather than a maximization problem:

```
>> help linprog

 LINPROG    Linear programming.
    X=LINPROG(f,A,b) solves the linear programming problem:

            min f'*x    subject to:   A*x <= b
             x

    X=LINPROG(f,A,b,Aeq,beq) solves the problem above while additionally
    satisfying the equality constraints Aeq*x = beq.

    X=LINPROG(f,A,b,Aeq,beq,LB,UB) defines a set of lower and upper
    bounds on the design variables, X, so that the solution is in
    the range LB <= X <= UB.  Use empty matrices for LB and UB
    if no bounds exist. Set LB(i) = -Inf if X(i) is unbounded below;
```

```
set UB(i) = Inf if X(i) is unbounded above.

X=LINPROG(f,A,b,Aeq,beq,LB,UB,X0) sets the starting point to X0.  This
option is only available with the active-set algorithm.  The default
interior point algorithm will ignore any non-empty starting point.

X=LINPROG(f,A,b,Aeq,Beq,LB,UB,X0,OPTIONS) minimizes with the default
optimization parameters replaced by values in the structure OPTIONS, an
argument created with the OPTIMSET function.  See OPTIMSET for details.
Use options are Display, Diagnostics, TolFun, LargeScale, MaxIter.
Currently, only 'final' and 'off' are valid values for the parameter
Display when LargeScale is 'off' ('iter' is valid when LargeScale is 'on').

[X,FVAL]=LINPROG(f,A,b) returns the value of the objective function at X:
FVAL = f'*X.

[X,FVAL,EXITFLAG] = LINPROG(f,A,b) returns EXITFLAG that
describes the exit condition of LINPROG.
If EXITFLAG is:
   > 0 then LINPROG converged with a solution X.
   0   then LINPROG reached the maximum number of iterations without converging.
   < 0 then the problem was infeasible or LINPROG failed.

[X,FVAL,EXITFLAG,OUTPUT] = LINPROG(f,A,b) returns a structure
OUTPUT with the number of iterations taken in OUTPUT.iterations, the type
of algorithm used in OUTPUT.algorithm, the number of conjugate gradient
iterations (if used) in OUTPUT.cgiterations.

[X,FVAL,EXITFLAG,OUTPUT,LAMBDA]=LINPROG(f,A,b) returns the set of
Lagrangian multipliers LAMBDA, at the solution: LAMBDA.ineqlin for the
linear inequalities A, LAMBDA.eqlin for the linear equalities Aeq,
LAMBDA.lower for LB, and LAMBDA.upper for UB.

NOTE: the LargeScale (the default) version of LINPROG uses a primal-dual
      method. Both the primal problem and the dual problem must be feasible
      for convergence. Infeasibility messages of either the primal or dual,
      or both, are given as appropriate.  The primal problem in standard
      form is
           min f'*x such that A*x = b, x >= 0.
      The dual problem is
           max b'*y such that A'*y + s = f, s >= 0.
```

To solve the optimization problem with costs $c_1 = c_2 = 30$, we can do the following:

```
>> f = [30;30];
>> A = -[100 40; 30 60]
>> b = -[200;180];
>> vlb = [0;0];
>> linprog(f,A,b,[],[],vlb)
Optimization terminated successfully.

ans =
    1.0000
    2.5000
```

This solution is illustrated graphically in Figure 6.2(a). If the cost of alfalfa goes up to \$1.00/kg:

```
>> f=[30;100];
>> linprog(f,A,b,[],[],vlb)
Optimization terminated successfully.

ans =
    6.0000
         0
```

This solution is illustrated graphically in Figure 6.2(b). In class we will study the graphical interpretation of these and other related problems.

### 6.2.1 Adding more constraints

Suppose we add some new constraints to the problem. For example, the farmer might want to keep the pigs from getting too fat by requiring that the total daily diet should be at most 5 kg, adding the constraint $x_1 + x_2 \leq 5$. In addition, suppose pigs should eat at most 3 kg of any particular grain, so that $x_1 \leq 3$ and $x_2 \leq 3$. This problem can be solved as follows:

```
>> A = -[100 40; 30 60; -1 -1];
>> b = -[200; 180; -5];
>> vlb = [0; 0];
>> vub = [3; 3];
>> linprog(f,A,b,[],[],vlb,vub)
Optimization terminated successfully.

ans =
    3.0000
    1.5000
```

This solution is illustrated graphically in Figure 6.3.

### 6.2.2 Solving the problem by enumeration

Each constraint corresponds to a straight line, so the feasible set is a convex polygon. The objective function we are minimizing is also linear, so the solution is at one of the corners of the feasible set. Corners correspond to points where two constraints are satisfied simultaneously, and can be found by solving a linear system of two equations (from these two constraints) in two unknowns $x_1$ and $x_2$. If we knew *which* two constraints were **binding** at the solution, we could easily solve the problem. Unfortunately we don't generally know this in advance.

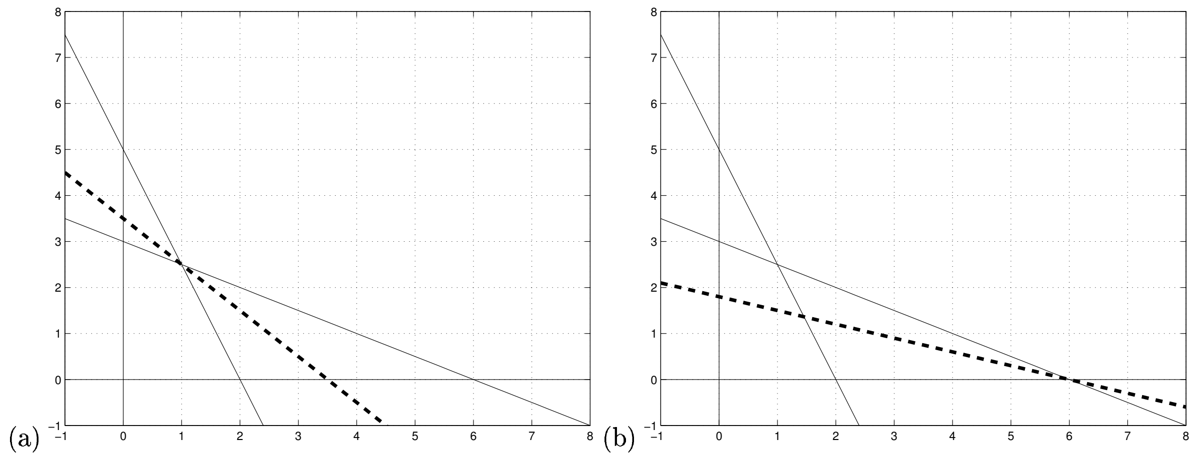



Figure 6.2: Graphical solution of the pig-farming problems in the $x_1$-$x_2$ plane. In each case the solid lines show constraints and the dashed line is the iso-cost line passing through the optimal solution, which is one of the corners of the feasible region. Shade in the feasible region in each figure!

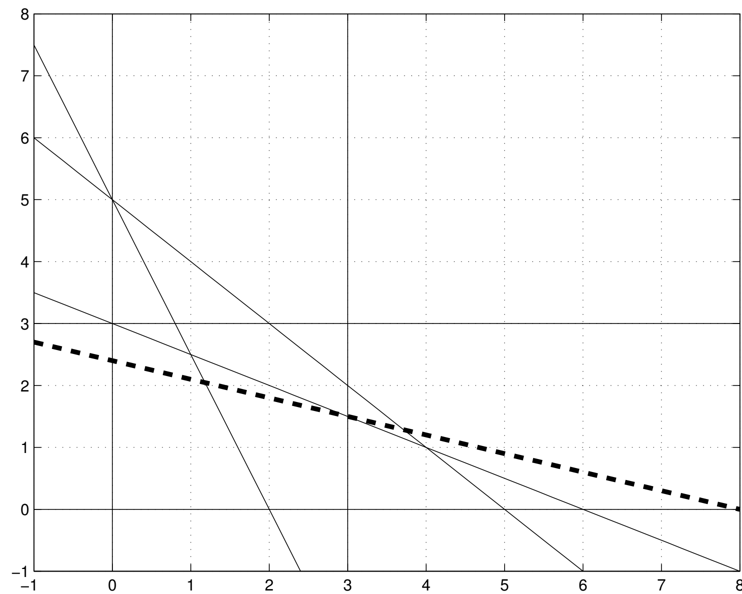



Figure 6.3: Graphical solution of the pig-farming problem with more constraints. The solid lines show constraints and the dashed line is the iso-cost line passing through the optimal solution, which is one of the corners of the feasible region. Shade in the feasible region!

One approach to solving the problem would be to consider all possible corners by solving all possible sets of 2 equations selected from the various constraints. For each choice of 2 equations we could:

- Solve the linear system for these two constraints,
- Check to see if the resulting solution is feasible (it might violate other constraints!),
- If it is feasible, compute and record the value of the objective function at this point.

After doing this for all choices of two constraints, the best value of the objective function found gives the solution.

The problem with this, for larger problems anyway, is that there is a combinatorial explosion of possible choices. If there are $m$ constraints and two decision variables, then there are $\binom{m}{2} = \frac{1}{2}m(m-1) = O(m^2)$ possible choices.

### 6.2.3 Adding a third decision variable

If there are more decision variables, then the problem is even worse. For example, if we consider 3 grains instead of only 2, then we would have three variables $x_1$, $x_2$, and $x_3$ to consider. In this case the pictures are harder to draw. Each constraint corresponds to a plane in 3-space and the feasible set is a convex body with planar sides. Changing the value of the objective function sweeps out another set of planes, and again the optimal solution occurs at a corner of the feasible set.

These corners are points where 3 of the constraints are simultaneously satisfied, so we can find all possible corners by considering all choices of 3 constraints from the set of $m$ constraints. There are now $\binom{m}{3} = O(m^3)$ ways to do this.

### 6.2.4 The simplex algorithm

Practical linear programming problems often require solving for hundreds of variables with thousands of constraints. In this case it is impossible to enumerate and check all possible "corners" of the feasible set, which is now an object in a linear space with dimension in the hundreds.

The *simplex algorithm* is a method that has proved exceedingly effective at finding solutions in much less time (though, like branch and bound algorithms, in the worst case it can still have exponential running time). The basic idea is to start by identifying one corner of the feasible region and then systematically move to "adjacent" corners until the optimal value of the objective function is obtained.

To gain a basic understanding of the simplex method, consider the following LP

$$\begin{aligned} \max \quad & 2x_1 - 3x_2 \\ \text{subject to} \quad & 2x_1 + x_2 \le 12 \\ & x_1 - 3x_2 \le 5 \\ & 0 \le x_1, x_2 \end{aligned} \tag{6.2}$$

The corresponding feasible region is depicted in Figure 6.4.

The simplex method uses a series of elementary matrix operations. Therefore, the system of linear equalities in LP (6.2) must be reformulated into a system of linear equalities constraints. The introduction of *slack variables* facilitates such a conversion of the constraints.

For instance, the inequality, $2x_1+x_2 \le 12$, becomes $2x_1+x_2+x_3 = 12$ with the addition of the slack variable $x_3 \ge 0$. Loosely speaking, $x_3$ makes up the *slack* in the inequality constraint $2x_1 + x_2 \le 12$. Take a moment and verify that the equality and inequality constraints are equivalent. Such verification is an important part of mathematical modeling.

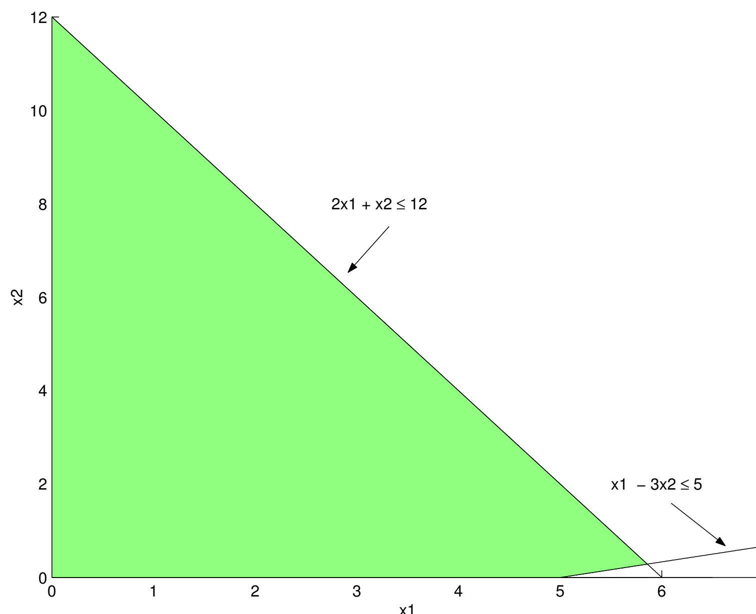



Figure 6.4: Feasible region for LP.

The constraint $x_1 - 3x_2 \leq 5$ still remains to be converted into a linear equality with a slack variable. In an attempt to reduce decision variables, one may be tempted to use $x_3$ again as a slack variable, which would form the linear equality $x_1 - 3x_2 + x_3 \leq 5$. Note that our linear system (6.2) does not imply that $2x_1 + x_2 - 12 = x_1 - 3x_2 - 5$, or loosely speaking that there are equal amounts of "slack" in each constraint. Forcing the same slack variable in each constraint changes the problem we are solving! Take a moment and consider the implications of using $x_3$ for each constraint in (6.2)!

Therefore, another slack variable $x_4 \geq 0$ is introduced into the second constraint, which becomes $x_1 - 3x_2 + x_4 = 5$. In general, each inequality constraint should use a unique slack variable.

Therefore, LP (6.2) becomes

$$\begin{aligned} \max \quad & z = 2x_1 - 3x_2 \\ \text{subject to} \quad & 2x_1 + x_2 + x_3 = 12 \\ & x_1 - 3x_2 + x_4 = 5 \\ & 0 \leq x_i, \quad i = 1, 2, 3, 4 \end{aligned} \tag{6.3}$$

Notice, $z$ equals the optimum of the objective function.

Having obtained a system of linear equality constraints, our system is prepared for matrix computation. Traditionally, the simplex method forms *tableaux*, which are equivalent to matrices. The initial simplex *tableau* for LP (6.3) is

| $z$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | RHS | |
|---|---|---|---|---|---|---|
| 1 | -2 | 3 | 0 | 0 | 0 | $\longleftarrow (z - 2x_1 + 3x_3 = 0)$ |
| 0 | 2 | 1 | 1 | 0 | 12 | $\longleftarrow (2x_1 + x_2 + x_3 = 12)$ |
| 0 | 1 | -3 | 0 | 1 | 5 | $\longleftarrow (x_1 - 3x_2 + x_4 = 5)$ |

Solutions can be read directly from the tableau. Since the column associated with variable $x_1$ contains more than 1 nonzero value, $x_1 = 0$. In contrast, the column associated with $x_3$ contains only one nonzero value, which implies that $x_3 = 12/1 = 12$. Therefore, this tableau leads to the solution

$x_1 = x_2 = 0, x_3 = 12, x_4 = 5$ and $z = 0$. Considering more details of the method requires that we establish additional terminology regarding a simplex tableau.

| $z$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | RHS | |
|---|---|---|---|---|---|---|
| 1 | -2 | 3 | 0 | 0 | 0 | ⟵ indicators |
| 0 | 2 | 1 | 1 | 0 | 12 | |
| 0 | 1 | -3 | 0 | 1 | 5 | |

*Basic variables* are nonzero variables in the tableau and correspond to a corner point of the feasible region. In the current tableau, $x_3$ and $x_4$ are basic variables associated with the corner point (0,0) (since $x_1 = 0$ and $x_2 = 0$). The basic idea of the simplex algorithm is to change which variables in the system are basic variables in order to improve the value of $z$. The numbers in shaded boxes in the top row of the tableau are called *indicators*. Let $\alpha_i$ denote the indicator in the column associated with variable $x_i$. Therefore, the algorithmic steps for the simplex algorithm as applied to the simplex tableau are as follows:

1. *Check the indicators for possible improvement* – If all the indicators are nonnegative, then the solution is optimal else go to Step 2.

2. *Check for unboundedness* – If $\alpha_i < 0$ and no positive element in the column associated with variable $x_i$ exists, then the problem has an unbounded objective value. Otherwise, finite improvement in the objective function is possible; go to Step 3.

3. *Select the entering variable* – The most negative indicator identifies the entering variable. That is, select $\alpha_i$ such that $\alpha_i < 0$, and $\alpha_i$ is the most negative indicator. The entering variable is $x_i$ and will be made nonzero. The column associated with $x_i$ is called the *pivot column.*

4. *Select the departing variable* – This step determines which basic variable will become zero. Find a quotient for each row by dividing each number from the right side of the tableau by the corresponding number from the pivot column. The *pivot row* is the row corresponding to the smallest quotient. The *pivot element* is the element in the pivot row and pivot column.

5. *Pivot creating a new tableau* - Use elementary row operations on the old tableau so the column associated with $x_i$ contains a zero in every entry except for a 1 in the pivot position. Go to Step 1.

Applying these steps to our tableau, we find the following:

| $z$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | RHS | Quotients |
|---|---|---|---|---|---|---|
| 1 | -2 | 3 | 0 | 0 | 0 | |
| 0 | 2 | 1 | 1 | 0 | 12 | $12/2 = 6$ |
| 0 | 1 | -3 | 0 | 1 | 5 | $5/1 = 1 \Rightarrow$ pivot row |
| | ↑ pivot column | | | | | |

Verify these results. Note that $x_1$ is the entering variable and $x_4$, the departing variable. For $x_1$ to become nonzero, we apply elementary row calculations to form the following tableau:

| $z$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | RHS |
|---|---|---|---|---|---|
| 1 | 0 | -3 | 0 | 2 | 10 |
| 0 | 0 | 7 | 1 | -2 | 2 |
| 0 | 1 | -3 | 0 | 1 | 5 |

This tableau leads to the solution $x_1 = 5, x_2 = 0, x_3 = 2, x_4 = 0$ and $z = 10$. The basic variables are indeed $x_1$ and $x_3$. Refer to Figure 6.4 to verify that our solution is a corner point of the feasible solution. Moreover, the current tableau represents an increase in the optimal value! Yet, the indicators identify that possible improvement still exists by selecting $x_2$ as an entering variable. The step of selecting the departing variable is left to the reader. Elementary row operations produce the following tableau:

| $z$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | RHS |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 3/7 | 8/7 | 76/7 |
| 0 | 0 | 1 | 1/7 | -2/7 | 2/7 |
| 0 | 1 | 0 | 3/7 | 1/7 | 41/7 |

This tableau leads to the solution $x_1 = \frac{41}{7}, x_2 = \frac{2}{7}, x_3 = x_4 = 0$ and $z = \frac{76}{7}$. The indicators are all positive, which represents that the optimal value $z$ cannot be improved. Therefore, the solution has been found.

Implementation of the simplex algorithm requires a great deal of linear algebra and bookkeeping. Imagine performing elementary row operations on a simplex tableau with thousands of decision variables. Fortunately, software packages exist that are aimed specifically at solving linear programming and related problems. One popular package is LINDO, which is available on some campus computers. The MATLAB function `linprog` implements the simplex algorithm. Program packages such as LINDO and MATLAB demonstrate the need for effective computation in the field of optimization.

## 6.3 Complexity and the simplex algorithm

The simplex algorithm was invented by George Dantzig in 1947 [Dan63]. Klee (from UW) and Minty [KM72] proved that the simplex algorithm is exponential in the worst case, but is very efficient in practice. Khachian [Kha79] invented the "ellipsoid" algorithm for finding solutions to linear programming problems in 1979 that he showed runs in polynomial time. Another more efficient algorithm due to Karmarkar[Kar84] uses an interior-point method instead of walking around the boundary. This algorithm appeared in 1984. In 2001, Spielman and Teng [ST02] introduced a new complexity class they call polynomial smoothed complexity and they showed that the simplex algorithm is the classic example in this class. Their work attempts to identify algorithms which are exponential in the worst case but which work efficiently in practice like the simplex algorithm.

# Chapter 7

# Integer Programming

## 7.1 The lifeboat problem

Many linear programming problems arising from real problems have the additional complication that the desired solution must have integer values. For example, in the lifeboat problem considered in Chapter 1, we must determine how best to equip a ferry with lifeboats and life vests, maximizing the number of lifeboats subject to certain constraints. The solution to the resulting linear programming problem will typically require some fractional number of lifeboats and life vests, which is not possible. Instead we must find the best integer solution.

Our first guess at how to solve such a problem might be to simply round off the values that come from the LP solution. This usually does not work, however. For example, for the values considered in Chapter 1, the LP solution was

$$\begin{aligned} x_1 &= 363.6364 \quad \text{(number of vests)} \\ x_2 &= 31.8182 \quad \text{(number of boats)} \end{aligned}$$

Trying to round this to $x_1 = 364$, $x_2 = 32$ would give an infeasible solution (requiring too much volume). The best integer solution was found to be $x_1 = 381$, $x_2 = 31$. We had to reduce $x_2$ to 31 and then the number of vests $x_1$ went up substantially.

Figure 7.1 shows the graphical solution to this problem with a different set of parameters:

$$\begin{aligned} C_1 &= 1 \quad \text{(capacity of each vest)} \\ C_2 &= 2 \quad \text{(capacity of each boat)} \\ C &= 11 \quad \text{(total capacity needed)} \\ V_1 &= 1 \quad \text{(volume of each vest)} \\ V_2 &= 3.1 \quad \text{(volume of each boat)} \\ V &= 15 \quad \text{(total volume available)} \end{aligned}$$

The dashed line $x_2 = 3.636$ shows the maximum value of the objective function $x_2$ obtained at the LP solution $x_1 = 3.727$, $x_2 = 3.636$. The feasible integer points are indicated by the dark dots, and we see that the best integer solution is $x_1 = 8$, $x_2 = 2$. In this case we can't even use the approach of Chapter 1, of reducing the $x_2$ to the nearest integer and increasing $x_1$ as needed.

For most IP (integer programming) problems, there is no direct way to find the solution. Recall that for the LP problem we know the optimum occurs at a corner of the feasible set, and the corners can be found by solving an appropriate linear system of equations corresponding to the binding constraints. For the IP problem typically none of the inequality constraints are exactly binding at the solution.

In simple cases one might be able to enumerate all integer points in the feasible set and check each, but for problems with many variables and with a large range of possible integer values for each, this is usually out of the question.

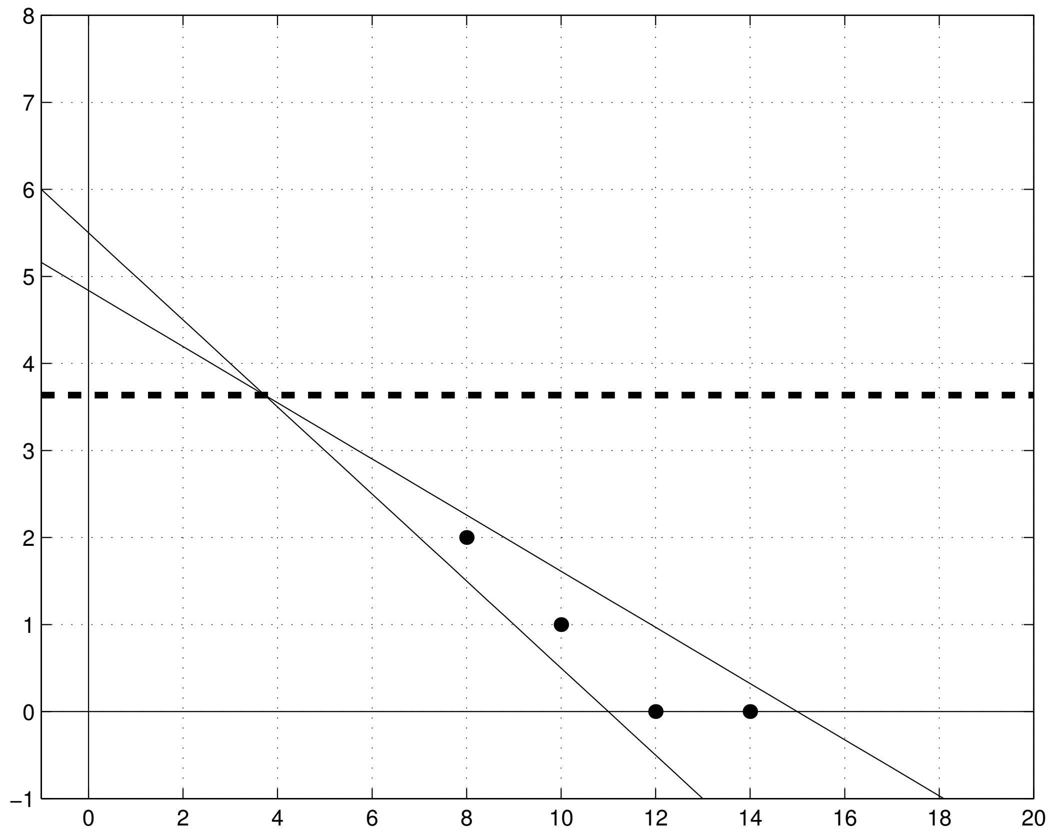



Figure 7.1: Graphical solution of the lifeboat problem in the $x_1$-$x_2$ plane. The solid lines show constraints and the dashed line is the objective function contour line passing through the optimal solution, which is one of the corners of the feasible region. When integer constraints are added, only the points marked by dots are feasible.

Some IP problems have a sufficiently special form that the solution is easily found. For example, certain *transportation problems*, such as in the example we considered of canneries and warehouses, have the property that the LP solution can be shown to be integer-valued. For other classes of IP problems there are special techniques which are effective.

For many problems a *branch and bound* approach must be used, in which linear programming problems are used to obtain bounds. This is illustrated in the next section for a relatively simple class of problems called *zero-one integer programming*, since each decision variable is restricted to take only the values 0 or 1. Many practical problems can be put in this form. Often each value represents a yes-no decision that must be made.

## 7.2 Cargo plane loading

Suppose you have a cargo plane with 1000 cubic meters of space available, and you can choose from the following sets of cargos to load the plane:

| Cargo | Volume | Profit |
|---|---|---|
| 1 | 410 | 200 |
| 2 | 600 | 60 |
| 3 | 220 | 20 |
| 4 | 450 | 40 |
| 5 | 330 | 30 |

For each cargo the volume and profit expected from shipping this cargo are listed. There are two possible scenarios:

- We are allowed to take any amount of each cargo up to the total volume, with the profit adjusted accordingly. (E.g., taking half the cargo yields half the profit.)
- We must take all of the cargo or none of it.

The second scenario is more realistic and gives a zero-one integer programming problem. The first scenario gives a linear programming problem with continuous variables, and is easier to solve, using the simplex method for example. It will be useful to consider problems of the first type in the process of solving the integer programming problem, as discussed in the next section.

Let $V_j$ be the volume of the $j$'th cargo, $P_j$ the profit, and $V = 1000$ the total volume available in the cargo plane. Introduce the decision variable $x_j$ to represent the fraction of the $j$'th cargo that we decide to take. In the first scenario above, $x_j$ can take any value between 0 and 1, while for the actual problem each $x_j$ must be either 0 or 1.

The integer programming problem then has the form

$$\text{maximize} \quad \sum_j P_j x_j \tag{7.1}$$

subject to

$$\sum_j V_j x_j \leq V \tag{7.2}$$

$$0 \leq x_j \leq 1 \tag{7.3}$$

$$x_j \quad \text{integer} \tag{7.4}$$

If we drop constraint (7.4) then we have the linear programming problem corresponding to the first scenario again.

For this small problem we can easily solve the integer programming problem by simply enumerating all possible choices of cargos. If we have $k$ cargos to choose from, then for each cargo the value $x_j$ can take one of two values and so there are $2^k$ possible combinations of cargo. In this case $k = 5$ and there

are 32 possibilities to consider. Not all will be feasible since the volume constraint may be exceeded. Here is the enumeration of all 32 possibilities, with the profit arising from each (which is set to zero for infeasible combinations):

```
 cargo choice      profit
1  2  3  4  5

0  0  0  0  0        0
0  0  0  0  1        30
0  0  0  1  0        40
0  0  0  1  1        70
0  0  1  0  0        20
0  0  1  0  1        50
0  0  1  1  0        60
0  0  1  1  1        90
0  1  0  0  0        60
0  1  0  0  1        90
0  1  0  1  0        0
0  1  0  1  1        0
0  1  1  0  0        80
0  1  1  0  1        0
0  1  1  1  0        0
0  1  1  1  1        0
1  0  0  0  0        200
1  0  0  0  1        230
1  0  0  1  0        240
1  0  0  1  1        0
1  0  1  0  0        220
1  0  1  0  1        250
1  0  1  1  0        0
1  0  1  1  1        0
1  1  0  0  0        0
1  1  0  0  1        0
1  1  0  1  0        0
1  1  0  1  1        0
1  1  1  0  0        0
1  1  1  0  1        0
1  1  1  1  0        0
1  1  1  1  1        0
```

The best combination is `10101`, meaning we should take Cargos 1, 3, and 5. For a larger problem it might not be possible to enumerate and check all possibilities. Moreover the real problem might be more complicated, for example a shipping company might have many planes and different sets of cargos and want to choose the optimal way to split them up between planes.

## 7.3 Branch and bound algorithm

Integer programming problems are often solved using branch and bound algorithms. This is easily illustrated for the cargo plane problem since this is a particularly simple form of integer programming, a 0-1 integer program in which each decision variable can take only the values 0 or 1. This suggests building up a binary tree by considering each cargo in turn. Part of such a tree is shown in Figure 7.2.

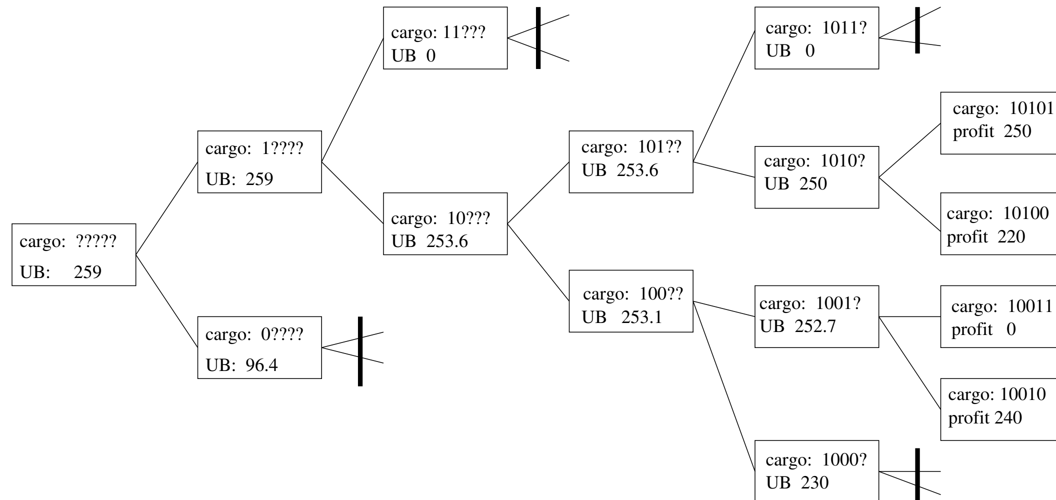



Figure 7.2: Illustration of the branch and bound algorithm for solving the cargo-loading problem. In each box the values of some 0-1 integer values are specified and then the linear programming problem is solved with the other variables allowed to take any value between 0 and 1. This gives an upper bound UB on the best that could be obtained with this partial cargo. If UB is 0 then this choice is infeasible.

The root is to the left, where the cargo is represented as `?????` meaning all five decision variables are unspecified. From here we branch to `1????` and `0????`, in which we either take or do not take Cargo 1. Each of these has two branches depending on whether we take or do not take Cargo 2. Continuing in this way we could build up a full tree whose leaves would be the 32 possible choices of cargos.

The goal of the branch and bound algorithm is to avoid having to build up the entire tree by obtaining a *bound* at each branching point for the best possible solution we could find down that branch of the tree. In this case we want an upper bound on the best profit we could possibly find with any selection that contains the pattern of zeroes and ones already specified, along with any possible choice of zeroes and ones for the cargos which are still denoted by `?`. It is exactly these selections which lie further down the tree on this branch.

We want to obtain this bound by doing something simpler than enumerating all possibilities down this branch and computing each profit — this is what we want to avoid. The essential idea is to instead solve a *linear programming* problem in which the variables still denoted by `?` are allowed to take any value between 0 and 1. This is more easily solved than the integer programming problem. The solution to this problem may involve non-integer values of some $x_j$, but whatever profit it determines (perhaps with these partial cargos) is an *upper bound* on the profit that could be obtained with the integer constraints imposed. Make sure you understand why!

For example, at the root node `?????`, we obtain an upper bound on the profit of any selection of cargos by solving the linear programming problem (7.1) with the constraints (7.2) and (7.3) only, dropping all integer constraints. The solution is $x_1 = 1$, $x_2 = 0.9833$ and $x_3 = x_4 = x_5 = 0$, with a profit of 259. This isn't a valid solution to the integer problem, but any valid selection must have a profit that is no larger than this.

At the node `1????` we solve the linear programming problem (7.1) with the constraints (7.2) and (7.3) and also the constraint $x_1 = 1$, by changing the bounds on $x_1$ to $1 \leq x_1 \leq 1$. (Or simply reduce the number of decision variables and modify the objective function appropriately.) The solution is the same as found at the root node, since the solution there had $x_1 = 1$.

At the node `0????` we solve the linear programming problem (7.1) with the constraints (7.2) and (7.3) and also the constraint $x_1 = 0$. This gives an upper bound on the profit of only 96.36 if Cargo 1 is not taken. Again this requires taking fractional cargos and any integer selection with $x_1 = 0$ will be even worse.

It looks more promising to take Cargo 1 than to not take it, and so we explore down this branch first. We next try `11???`, taking Cargo 1 and Cargo 2 and solving a linear programming problem for the other variables, but we find this is infeasible since Cargo 1 and 2 together require too much volume. At this point we can prune off this branch of the tree since every selection which lies down path must be infeasible.

Continuing this way, we eventually work down to a leaf and find an actual integer selection with an associated profit. From this point on we can also prune off any branches for which the upper bound on the profit is smaller than the actual profit which can be obtained by a valid selection. So, for example, all of the tree lying beyond the node `0????` can be pruned off without further work since the upper bound of 96.36 is smaller than leaf values we have already found.

## 7.4 Traveling salesman problem

We now return to the Traveling Salesman Problem (TSP) of Chapter 2, and see how this can be set up as an integer programming problem. Consider a graph with $N$ vertices labeled 1 through N, and let $d_{ij}$ be the "distance" from $i$ to $j$. Recall that we wish to find the path through all nodes which minimizes the total distance traveled. To set this up as an IP problem, let $x_{ij}$ be 1 if the edge from Node $i$ to Node $j$ is used and 0 otherwise. For example, with $N = 5$ the path $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow 1$ would be encoded by setting

$$x_{12} = x_{23} = x_{35} = x_{54} = x_{41} = 1$$

and the other 15 $x_{ij}$ values to 0. (For a graph with $N$ nodes there will be $N(N-1)$ decision variables.)

Now we wish to

$$\text{minimize} \sum_{i=1}^{N} \sum_{j=1}^{N} d_{ij} x_{ij} \tag{7.5}$$

subject to various constraints. One constraint is that we must use exactly one edge out of each node. This leads to

$$\sum_{j=1}^{N} x_{ij} = 1. \tag{7.6}$$

for each $i = 1,\ 2,\ \ldots,\ N$. Also, we must use exactly one edge into each node, so

$$\sum_{i=1}^{N} x_{ij} = 1. \tag{7.7}$$

for each $j = 1,\ 2,\ \ldots,\ N$.

This is not enough, for the constraints would allow things like

$$x_{12} = x_{23} = x_{31} = 1 \qquad \text{and} \qquad x_{45} = x_{54} = 1 \tag{7.8}$$

with all the other $x_{ij} = 0$. This is not a single TSP tour, but rather two disconnected tours. To rule out the possibility of *subtours*, we must add other constraints. This can be done in various ways. One possibility is to introduce a new set of decision variables $u_1,\ \ldots,\ u_N$ which do not appear in the objective function but which have the $(N-1)^2$ constraints

$$u_i - u_j + N x_{ij} \leq N - 1, \qquad \text{for } i, j = 2,\ 3,\ \ldots,\ N. \tag{7.9}$$

Note that $u_1$ does not appear in these constraints.

The idea is that if we have a set of $x_{ij}$ which correspond to a valid TSP tour, then it should be possible to choose some values $u_i$ so that these constraints are satisfied, so that all such tours are still feasible. On the other hand for a set of $x_{ij}$ that contain subtours, such as (7.8), it should be impossible to find a set of $u_i$ satisfying (7.9) and hence such sets will no longer be feasible.

To see why it is impossible to satisfy these constraints if there are subtours, consider (7.8), for example. The second subtour defined by $x_{45} = x_{54} = 1$ does not involve Node 1. The constraints (7.9) for $i = 4,\ j = 5$ and for $i = 5,\ j = 4$ yield

$$\begin{aligned} u_4 - u_5 + 5 &\leq 4 \\ u_5 - u_4 + 5 &\leq 4 \end{aligned}$$

Adding these together gives

$$10 \leq 8$$

which cannot be satisfied no matter what values we assign to $u_4$ and $u_5$. So we cannot possibly satisfy *both* of the constraints above simultaneously. A similar argument works more generally: If there are subtours then there is always a subtour that doesn't involve Node 1 and writing down the constraints (7.9) for each $(i, j)$ in this subtour gives a set of equations that, when added together, leads to cancellation of all the $u_i$ values and an inequality that can never be satisfied.

On the other hand, we wish to show that if we consider a valid TSP tour (with no subtours) then we can find an assignment of $u_i$ values that satisfies all these constraints. The basic idea is to set $u_1 = 0$ and then follow the tour starting at Node 1, setting the $u_i$ variables according to the order in which we visit the nodes. This is best illustrated with an example:

$$1 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 1$$

yields

$$u_1 = 0, \quad u_2 = 1, \quad u_5 = 2, \quad u_4 = 3, \quad u_3 = 4.$$

We can show that this satisfies all the constraints in (7.9).

First, consider the pairs $(i, j)$ for which $x_{ij} = 0$. In this case (7.9) reads

$$u_i - u_j \leq N - 1$$

This will certainly be true since all the values we assigned to $u$ variables lie between 0 and $N - 1$.

Next, consider the pairs $(i, j)$ for which $x_{ij} = 1$. This means we use the edge from Node $i$ to $j$ in our tour, Node $j$ is visited immediately after Node $i$, and hence $u_j = u_i + 1$. So we can compute that

$$u_i - u_j + N x_{ij} = -1 + N = N - 1$$

and hence the inequality in (7.9) is satisfied (with equality in this case).

### 7.4.1 NP-completeness of integer programming

We just showed that it is possible to reformulate any traveling salesman problem as an integer programming problem. This can be used to prove that the general integer programming problem is NP-complete. For if there were a polynomial-time algorithm to solve the general integer programming problem, then we could use it to solve the TSP, and hence have a polynomial-time algorithm for the TSP.

## 7.5 IP in Recreational Mathematics

Integer Programming also applies to problems in recreational math such as the challenging brain teasers found in puzzle books available at bookstores or on the World Wide Web at sites such as `rec.puzzles`. Whether tinkering with such problems in your mind or applying IP, these problems offer fun challenges.

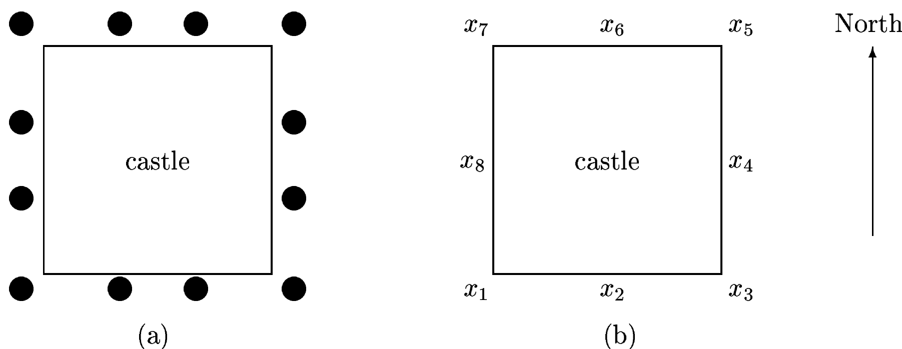



Figure 7.3: The well-guarded castle puzzle involves 12 guards posted around a square castle. (a) Each guard is depicted with a solid dot. Note that 1 soldier is posted at each corner of the castle and 2 soldiers on each wall. (b) The associated IP can be formulated with 8 decision variables.

### 7.5.1 An Example Puzzle

To illustrate IP's in recreational math, we begin with a simple brain teaser based on a puzzle from the book, The Moscow Puzzles, by B. A. Kordemsky. [Kor92]

> THE WELL-GUARDED CASTLE
>
> There is unrest in the land, and King Husky declares that the castle be guarded on every side. Guards will be posted along a wall or at a corner of the square castle. When posted on a corner, a soldier can watch 2 walls. Other soldiers are posted along the face of a wall and watch only 1 side of the castle. The king orders 12 of his bravest soldiers to guard the stone fortress and initially plans to post 2 guards on each wall and 1 soldier on each corner as depicted in Figure 7.5.1 (a). Aid King Husky in securing the kingdom by finding an arrangement of the soldiers such that there are 5 soldiers watching each wall.

Since the problem serves as a brain teaser, you may be able to derive an answer by inspection or trial and error. Imagine a much larger problem with thousands of soldiers. In such cases, the process needed to find a solution could easily move beyond the scope of mental calculation. Therefore, this puzzle serves as an illustrative example of formulating such problems as integer programming problems.

### 7.5.2 Formulation of the IP

The solution process begins by determining the decision variables. As in linear programming, the best way to determine these variables is to put oneself in the shoes of the decision maker and ask the question "What do I need to know in order to fully determine the solution?" In the case of the well-guarded castle, we must know the number of soldiers positioned along each wall and at each corner. Hence, we define 8 decision variables $x_i$ for $i = 1, 2, \ldots, 8$, where $x_i$ equals the number of soldiers posted at position $i$. The positions are numbered counterclockwise beginning at the southwest corner as seen in Figure 7.5.1 (b).

Recall, the king declared that 5 soldiers watch each wall. Again, a soldier on a corner can watch 2 walls while a soldier along a wall watches only 1. This leads to the following constraints:

$$\begin{array}{ll} \text{southern wall constraint:} & x_1 + x_2 + x_3 = 5 \\ \text{eastern wall constraint:} & x_3 + x_4 + x_5 = 5 \\ \text{northern wall constraint:} & x_5 + x_6 + x_7 = 5 \\ \text{western wall constraint:} & x_7 + x_8 + x_1 = 5 \end{array}$$

What is the puzzle's objective? That is, what is a suitable objective function? As you may have noticed, there are several solutions to this puzzle. For instance, the guards can be positioned such that $x_1 = x_5 = 4$ and $x_2 = x_4 = x_6 = x_8 = 1$, or by symmetry, $x_3 = x_7 = 4$ and $x_2 = x_4 = x_6 = x_8 = 1$. Objective functions to maximize $x_1 + x_5$ or $x_3 + x_7$ could produce these answers, respectively. It is also possible to reduce the number of guards on the corners such that $x_1 = x_5 = x_3 = x_7 = 2$, and $x_2 = x_4 = x_6 = x_8 = 1$. Take a moment and consider what objective function could produce this answer. As we see, the choice of an objective function determines which of such solutions is optimal! It should be noted that in some cases, if you desire a particular solution, then additional constraints may be needed.

Suppose King Husky wants the total number of soldiers watching the corners to be at a minimum. Then, our objective function becomes: $\min x_1 + x_3 + x_5 + x_7$.

One constraint remains. The king issues 12 guards, which implies $\sum_{i=1}^{8} x_i = 12$.

The entire model for the well-guarded castle puzzle can now be succinctly stated as

$$
\begin{aligned}
P: \ & \min x_1 + x_3 + x_5 + x_7 \\
& \text{subject to} \quad
\begin{aligned}
x_1 + x_2 + x_3 &= 5 \\
x_3 + x_4 + x_5 &= 5 \\
x_5 + x_6 + x_7 &= 5 \\
x_7 + x_8 + x_1 &= 5 \\
\textstyle\sum_{i=1}^{8} x_i &= 12 \\
0 \le x_i \le 5, \quad & i = 1, 2, \ldots, 8
\end{aligned}
\end{aligned}
$$

In class, we will look at other puzzles and pose the problems as IP problems. The interested reader is encouraged to visit the web site, "Puzzles: Integer Programming in Recreational Mathematics" (`http://www.chlond.demon.co.uk/academic/puzzles.html`) to find additional puzzles.

## 7.6 Recasting an IP as a 0-1 LP

In the last section, $P$ posed the well-guarded castle puzzle as an IP. Yet, posing an IP does not imply that it is solved. King Husky (like an executive of a company who hires a consulting firm to model a phenomenon) demands an answer from a model that he can trust. Presenting King Husky with a properly posed IP and no solution might result in one's being thrown over that mighty wall!

Solving $P$ as an LP allows variables in the solution to contain nonzero decimal values. Such a solution *relaxes* the constraint that the decision variables are integers. Hence, an IP that does not enforce integral solutions is called a *relaxed LP*. Imagine presenting King Husky with the solution that 1.1 soldiers be posted on the southern wall. Even a cruel leader would recognize the foolishness of such an action. Still, solving an IP as a relaxed LP can yield intuition on the final solution. Yet, deciphering the optimal or even near optimal integer solution from a real numbered solution is often a difficult, if not impossible, task.

A 0-1 LP (also called a 0-1 IP) is a linear programming problem where decision variables can be considered as boolean variables attaining only the values 0 or 1. Moreover, every IP can be recast as a 0-1 LP. Techniques such as branch and bound can be applied to solve a 0-1 LP. Our solution can then be recast from the 0-1 LP back to the integer solution of the corresponding IP.

Assume $Q$ is an IP, and $x_i$ is an integer decision variable of $Q$. We introduce the decision variables

$d_{i1}, d_{i2}, \ldots d_{ik}$ and let

$$\begin{aligned} x_i &= \sum_{j=1}^{k} 2^{j-1} d_{ij} \\ &= d_{i1} + 2d_{i2} + \ldots + 2^{k-1} d_{ik}, \\ &\text{where } d_{ij} = 0 \text{ or } 1, j = 1, 2, \ldots, k. \end{aligned} \tag{7.10}$$

For $P$, the decision variables are bounded below by 0 and above by 5. Therefore, it is suitable to let $x_1 = d_{11} + 2d_{12} + 4d_{13}$. Note that if $d_{11} = d_{13} = 1$ and $d_{12} = 0$ then $x_1 = 3$. Similarly, every possible value of $x_1$ can be determined by a suitable linear combination of $d_{11}, d_{12}$, and $d_{13}$. Note that the decision variables $d_{ij}$ can also be viewed as the binary representation of the associated value of $x_i$.

Completing similar substitutions for each decision variable in $P$, the corresponding 0-1 LP becomes:

$$\begin{aligned} \min \quad & d_{11} + 2d_{12} + 4d_{13} + d_{31} + 2d_{32} + 4d_{33} + d_{51} + 2d_{52} + 4d_{73} + d_{71} + 2d_{72} + 4d_{73} \\ \text{subject to} \quad & \textstyle\sum_{i=1}^{8} \sum_{j=1}^{3} 2^{j-1} d_{ij} = 12 \\ & d_{11} + 2d_{12} + 4d_{13} + d_{21} + 2d_{22} + 4d_{23} + d_{31} + 2d_{32} + 4d_{33} \quad = 5 \\ & d_{31} + 2d_{32} + 4d_{33} + d_{41} + 2d_{42} + 4d_{43} + d_{51} + 2d_{52} + 4d_{53} \quad = 5 \\ & d_{51} + 2d_{52} + 4d_{53} + d_{61} + 2d_{62} + 4d_{63} + d_{71} + 2d_{72} + 4d_{73} \quad = 5 \\ & d_{71} + 2d_{72} + 4d_{73} + d_{81} + 2d_{82} + 4d_{83} + d_{11} + 2d_{12} + 4d_{13} \quad = 5 \\ & 0 \leq d_{ij} \leq 1, i = 1, 2, \ldots, 8, j = 1, 2, 3. \end{aligned}$$

Such a problem can be solved using a branch and bound algorithm or linear programming software.

What if our problem did not have an upper bound on the decision variables? In such cases, the IP is recast to a 0-1 LP with a large bound on each decision variable. Then, the bound is incremented by a fixed value (say, between 10 and 50) until the solution no longer changes at which point, the solution is found.

Integer programming can model a diverse range of problems. In this chapter, you learned techniques and some of the difficulties of solving IP problems.

# Bibliography


[BD92] D. Bayer and P. Diaconis. Trailing the dovetail shuffle to its lair. *Ann. Appl. Prob.*, 2:294–313, 1992.

[Chv83] Vasek Chvatal. *Linear Programming.* W. H. Freeman, New York, 1983. T57.74.C54.

[DA93] A. Dolan and J. Aldous. *Networks and Algorithms, An Introductory Approach.* John Wiley & Sons, Chichester, 1993.

[Dan63] George Dantzig. *Linear programming and extensions.* Princeton University Press, Princeton, NJ, 1963.

[Dyk84] D. P. Dykstra. *Mathematical Programming for Natural Resource Management.* McGraw-Hill, 1984.

[Fel81] J. Felsenstein. Evolutionary trees from dna sequences: a maximum likelihood approach. *J. Mol. Evol.*, 17:368–376, 1981.

[Gil55] E. Gilbert. Bell labs technical report. ., 1955.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A guide to the Theory of NP-Completeness.* W. H. Freeman, San Francisco, 1979.

[HL95] F. S. Hillier and G. J. Lieberman. *Introduction to operations research.* McGraw-Hill, New York, 1995.

[Jef78] J. N. R. Jeffers. *An Introduction to Systems Analysis: with Ecological Applications.* University Park Press, London, 1978.

[JT98] G. Jónsson and L. N. Trefethen. A numerical analyst looks at the 'cutoff phenomenon' in card shuffling and other markov chains. In *Numerical Analysis 1997, D. F. Griffiths, D. J. Higham, and G. A. Watson, eds.*, pages 150–178. Addison Wesley Longman, Ltd., 1998.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations, Proc. Sympos. IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y.*, pages 85–103. Plenum, 1972.

[Kar84] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[Kha79] L. Khachian. A polynomial algorithm in linear programming. *Soviet Math. Dokl.*, 20:191–194, 1979.

[KM72] Victor Klee and G. Minty. How good is the simplex algorithm. In *Inequalities III*, pages 159–172. Academic Press, 1972.

[Knu93] D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing.* Addison-Wesley, New York, 1993.

[Kol90] G. Kolata. In card shuffling, 7 is winning number. In *New York Times*, page C1, 1990.

[Kor92] B. Kordemsky. *The Moscow Puzzles.* Dover, reprint edition, 1992.

[LLAS90] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. *The Traveling salesman problem : a guided tour of combinatorial optimization.* Wiley-Interscience, Chichester, 1990.

[Mak73] D. P. Maki. *Mathematical models and applications, with emphasis on the social, life, and management sciences.* Prentice-Hall, Englewood Cliffs, 1973.

[Min78] E. Minieka. *Optimization algorithms for networks and graphs.* Marcel Dekker, New York, 1978.

[Ree81] J. Reeds. Theory of shuffling. *unpublished manuscript*, 1981.

[Rob76] Fred S. Roberts. *Discrete mathematical models, with applications to social, biological, and environmental problems.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

[Rob84] Fred S. Roberts. *Applied combinatorics.* Prentice-Hall, Englewood Cliffs, N.J., 1984.

[Ros90] S. M. Ross. *A Course in Simulation.* Macmillan Publishing Company, New York, 1990.

[SCCH97] G. C. Smith, C. L. Cheeseman, and R. S. Clifton-Hadley. Modelling the control of bovine tuberculosis in badgers in england: culling and the release of lactating females. *J. Applied Ecology*, 34:1375–1386, 1997.

[Sip97] M. Sipser. *Introduction to the theory of computation.* PWS Pub. Co., Boston, 1997.

[SK87] G. L. Swartzman and S. P. Kaluzny. *Ecological Simulation Primer.* MacMillan Publishing, 1987.

[Ski90] S. S. Skiena. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica.* Addison-Wesley, Redwood City, CA, 1990.

[Slo00] Neil Sloane. The on-line encyclopedia of integer sequences. In *A008292.* http://www.reseach.att.com/ njas/sequences, 2000.

[ST02] Daniel Spielman and Shang-Hua Teng. Smoothed analysis of algorithms. In *Proceedings of the International Congress of Mathematicians*, volume I, pages 597–606, 2002.

[UMA95] UMAP Journal. *MCM, The First Ten Years, special edition of the UMAP Journal.* COMAP, 1995.

[ZP97] Z. Zhou and W. Pan. Analysis of the viability of a giant panda population. *J. Applied Ecology*, 34:363–374, 1997.