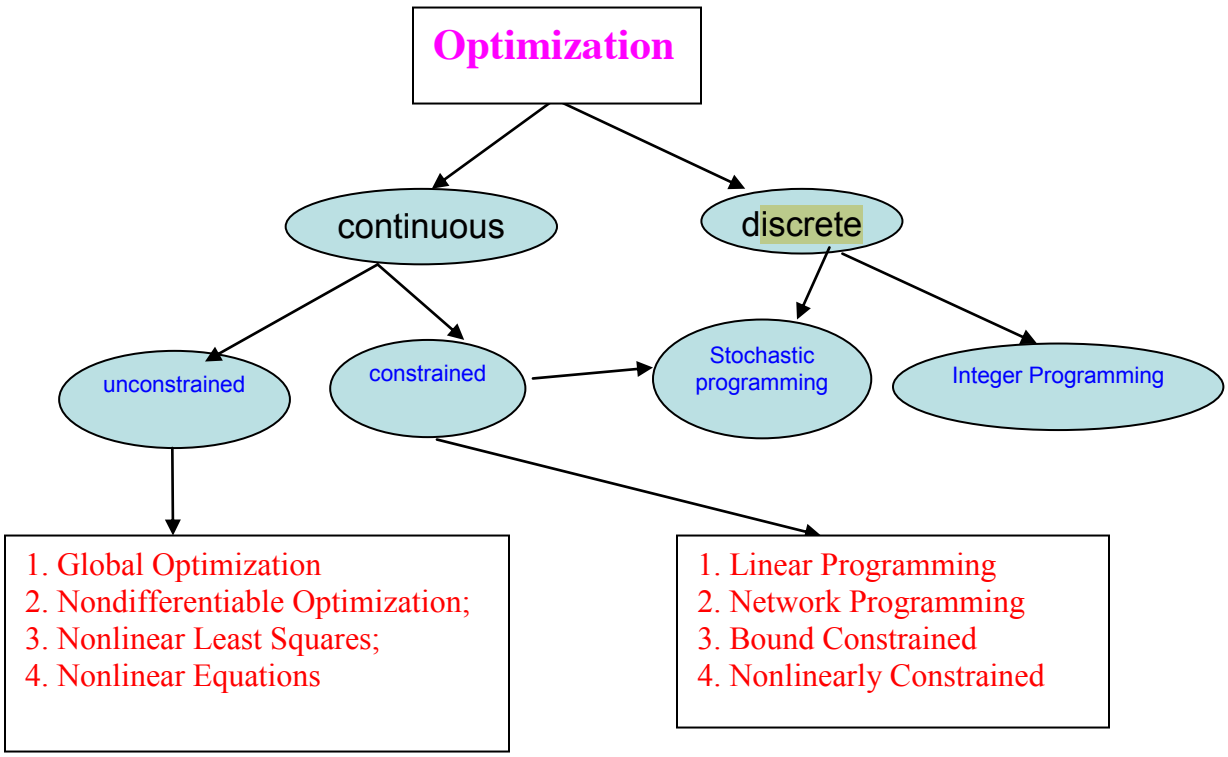
SYS 5130 – by Dr. Hua

Table of Contents

Chapter 1 Introduction of Optimization

Look at the following optimization tree:



1.1 What is optimization?

Optimization problem consists of three basic components:

目标函数

- **Objective function** which we want to minimize or maximize.

For instance, in a manufacturing process, we might want to maximize the profit or minimize the cost. In fitting experimental data to a user-defined model, we might minimize the total deviation of observed data from predictions based on the model. In designing an automobile panel, we might want to maximize the strength.

Almost all optimization problems have a single objective function. The two interesting exceptions are:

1. No objective function. In some cases (for example, design of integrated circuit layouts), the goal is to find a set of variables that satisfies the constraints of the model. The user does not particularly want to optimize anything so there is no reason to define an objective function. This type of problems is usually called a feasibility problem.
2. Multiple objective functions. Often, the user would actually like to optimize a number of different objectives at once. For instance, in the panel design problem, it would be nice to minimize weight and maximize strength simultaneously. Usually, the different objectives are not compatible; the variables that optimize one objective may be far from optimal for the others. In practice, problems with multiple objectives are reformulated as single-objective problems by either forming a weighted combination of the different objectives or else replacing some of the objectives by constraints.

变量

- **Variables** which affect the value of the objective function.

In the manufacturing problem, the variables might include the amounts of different resources used or the time spent on each activity. In fitting-the-data problem, the unknowns are the parameters that define the model. In the panel design problem, the variables used define the shape and dimensions of the panel.

These are essential. If there are no variables, we cannot define the objective function and the constraints.

约束

- **Constraints** that allow variables to take on certain values but exclude others.

For the manufacturing problem, it does not make sense to spend a negative amount of time on any activity, so we constrain all the "time" variables to be non-negative. In the panel design problem, we would probably want to limit the weight of the product and to constrain its shape.

Constraints are not essential. In fact, the field of unconstrained optimization is a large and important one for which a lot of algorithms and software are available.
Some problems really do have constraints. For example, any variable denoting the "number of objects" in a system can only be useful if it is less than the number of elementary particles in the known universe! In practice though, answers that make good sense in terms of the underlying physical or economic problem can often be obtained without putting constraints on the variables.

Optimization means to **find values of the variables that minimize or maximize the objective function while satisfying the constraints.**

离散

1.2 Discrete Optimization

1.2.1 Integer Programming 整数规划

In many applications, the solution of an optimization problem makes sense only if certain of the unknowns are integers. Integer linear programming problems have the general form

$$\min\left\{c^T x : Ax = b, x \geq 0\right\},$$

where x is the variable of integer vector, c is the cost vector, A is the constraint matrix. In mixed-integer linear programs, some components of x are allowed to be real.

Integer programming problems, such as the fixed-charge network flow problem and the famous traveling salesman problem, are often expressed in terms of binary variables. The fixed-charge network problem modifies the minimum-cost network flow paradigm by adding an adjust term to the cost. In other words, there is a fixed overhead cost for using the arc at all. In the traveling salesman problem, we need to find a tour of a number of cities that are connected by directed arcs, so that each city is visited once and the time required to complete the tour is minimized.

1.2.2 Stochastic Programming 随机规划

For many actual problems, the problem data cannot be known accurately for a variety of reasons. The first reason is due to simple measurement error. The second and more fundamental reason is that some data represent information about the future (e.g., product demand or price for a future time period) and simply cannot be known with certainty.

1.3 Continuous Optimization

In which all the variables are allowed to take values from subintervals of the real line.

1.3.1 Unconstrained optimization

The unconstrained optimization problem is central to the development of optimization software. Constrained optimization algorithms are often extensions of unconstrained algorithms, while nonlinear least squares and nonlinear equation algorithms tend to be specializations.

In the unconstrained optimization problem,

局部优化

- Local optimization: seek a local minimizer (or a local maximizer) of a real-valued function, f(x), where x is a vector of real variables. In other words, we seek a vector, x*, such that
  1. (Minimizer) $f(x^*) \leq f(x)$ for all x near x*; or
  2. (Maximizer) $f(x^*) \geq f(x)$ for all x near x*.

  Here in 1., f(x*) is called *minima*, and in 2., f(x*) is called *maxima*.

全局优化
- Global optimization: algorithms try to find an x* that minimizes (or maximizes) f(x) over all possible vectors x.

For many applications, local minima (or maxima) are good enough, particularly when the user can draw on his/her own experience and provide a good starting point for the algorithm.

Many methods can be used to solve this kind of problem, for example:

- **Basic Newton's method**: Objective Function should be'mooth

Newton's method gives rise to a wide and important class of algorithms that require computation of the gradient vector 梯度向量

$$\nabla f(x) = (\partial_1 f(x), \ldots, \partial_n f(x))^T$$

and the Hessian matrix,

$$\nabla^2 f(x) = (\partial_i \partial_j f(x)).$$

Newton's method forms a model of the objective function around the current iterate. The model function is defined by

$$g_k(s) = f(x_k) + \nabla f(x_k)^T s + \frac{1}{2} s^T \nabla^2 f(x_k) s.$$

In the basic Newton method, the next iterate is obtained from the minimizer of g_k, we denote it by s_k. The iteration is defined by

$$x_{k+1} = x_k + s_k.$$

Newton's method can be implemented by using **GAUSS** programming language, which is a high level matrix programming language specializing in commands, functions, and procedures for data analysis and statistical applications.

- **Nonlinear conjugate gradient methods**

Which are motivated by the success of the linear conjugate gradient method in minimizing quadratic functions with positive definite Hessians. They use search directions that combine the negative gradient direction with another direction, chosen so that the search will take place along a direction not previously explored by the algorithm. At least, this property holds for the quadratic case, for which the minimizer is found exactly within just n iterations. For nonlinear problems, performance is problematic, but these methods do have the advantage that they require only gradient evaluations and do not use much storage.

- **Nonlinear Simplex method**

Which requires neither gradient nor Hessian evaluations. Instead, it performs a pattern search based only on function values. Because it makes little use of information about f, it typically requires a great number of iterations to find a solution that is even in the ballpark. It can be useful when f is non-smooth or when derivatives are impossible to find, but it is unfortunately often used when one of the algorithms above would be more appropriate.

The simplex method generates a sequence of feasible iterates by repeatedly moving from one vertex of the feasible set to an adjacent vertex with a lower value of the objective function. When it is not possible to find an adjoining vertex with a lower value of the objective function, the current vertex must be optimal, and termination occurs.

After its discovery by Dantzig in the 1940s, the simplex method was unrivaled, until the late 1980s, for its utility in solving practical linear programming problems. Although never observed on practical problems, the poor worst-case behavior of the algorithm---the number of iterations may be exponential in the number of unknowns---led to an ongoing search for algorithms with better computational complexity. This search continued until the late 1970s, when the first polynomial-time algorithm (Khachiyan's ellipsoid method) appeared.

1.3.2 Constrained optimization

1. linear programming 线性规划

The basic problem of linear programming is to minimize a linear objective function of continuous real variables, subject to linear constraints. For purposes of describing and analyzing algorithms, the problem is often stated in the standard form

$$\min\left\{c^T x : Ax = b, x \geq 0\right\},$$

where x is the vector of variables, c is the cost vector, A is the constraint matrix.

2. Network programming

Network problems come from applications that can be represented as the flow of a commodity in a network. The resulting programs can be linear or non-linear

For example, assume that you are the manager of a company that has different production lines in different locations. The goods produced by your company (in these different locations) are shipped to the distribution centers. You want to minimize the cost of shipping your product to the different distribution centers while meeting the demand of the customers. Remember, your company produces items or units that cannot be broken

down into fractions (cars for example); i.e., some of the decision variables representing your shipping problem must be integer.

Network problems cover a large number of applications. For example:

- **Transportation Problem.** We have a commodity that can be produced in different locations and needs to be shipped to different distribution centers. Given the cost of shipping a unit of commodity between each two points, the capacity of each production center, and the demand at each distribution center, find the minimal cost shipping plan.
- **Assignment Problem.** This is a special case of transportation problem. There are x individuals that need to be assigned to x different tasks one to one. Given the cost that each individual charges for performing each of the jobs, find a minimal cost assignment.
- **Maximum Value Flow.** Given a directed network of roads that connects two cities and the capacities of these roads, find the maximum number of units (cars) that can be routed from one city to another.
- **Shortest Path Problem.** Given a directed network and the length of each arc in this network, find a shortest between two given nodes.
- **Minimum Cost Flow Problem.** Given a directed network with upper and lower capacities on each of its arcs, and given a set of external flows (positive or negative) that need to be routed through this network, find the minimal cost routing of the given flows through this network. Here, the cost per unit of flow on each arc is assumed to be known.

1.4 Global optimization

Global optimization means to find global maximum or minimum.
Three most popular methods are:

Simulated annealing,
genetic algorithms, and
smoothing/continuation methods.

They are based on analogies to natural processes where more or less global optima are reached. A more mathematically motivated technique is:

Branch and bound.

Generally, branch and bound methods (in particular, interval methods, dc methods, and mixed integer methods based on piecewise linear approximations) are more reliable since, to the extent they work (which depends on the difficulty of the problem), they have built in guarantees; however, they require more or less detailed access to global information about the problem.
On the other hand, stochastic or heuristic methods are generally easier to program, depend only on black box function (and sometimes gradient) routines, and therefore can never be sure of not having missed the global optimizer.

Chapter 2 Fourier-Motzkin Elimination

- Invented by Fourier (1827)
- Similar to Gaussian elimination (1800)

2.1 Fourier-Motzkin theorem:

1. Take all pairs of inequalities with opposite sign coefficients of x1, and for each generate a new valid inequality that eliminates x1;
2. Take all inequalities from the original set which do not depend on x1;
3. Repeat the procedure until find solutions for the last variable.

Example. Consider the following system

$3x + 4y \geq 16$ (1)
$4x + 7y \leq 56$ (2)
$4x - 7y \leq 20$ (3)
$2x - 3y \geq -9$ (4)

Let's see how to solve.

(i) Express all inequalities as upper or lower bounds on x.

$x \geq 16/3 - 4y/3$ (5)
$x \leq 14 - 7y/4$ (6)
$x \leq 5 + 7y/4$ (7)
$x \geq -9/2 + 3y/2$ (8)

For any y, if there is an x that satisfies all inequalities, then every lower bound on x must be less than or equal to every upper bound on x.

(ii) Generate a new system of inequalities from each pair (upper, lower) bounds.

$5 + 7y/4 \geq 16/3 - 4y/3$
$5 + 7y/4 \geq -9/2 + 3y/2$
$14 - 7y/4 \geq 16/3 - 4y/3$
$14 - 7y/4 \geq -9/2 + 3y/2$

(iii) Simplify:

$y \geq 4/37$
$y \geq -38$
$y \leq 104/5$
$y \leq 74/13$

=>

$max(4/37,-38) \leq y \leq min(104/5, 74/13)$

=>

$4/37 \leq y \leq 74/13$

(iv) We can now express solutions in closed form as follows:

$$4/37 \le y \le 74/13$$
$$max(16/3 - 4y/3, -9/2 + 3y/2) \le x \le min(5 + 7y/4, 14 - 7y/4)$$

Remark. If you eliminate y at first, and follow similar procedures, then you will get another solution. Are theese two solutions equivalent?

Iterative algorithm

Two systems equivalent means:
A can imply B, B can imply A

Iteration steps:

- Obtain reduced system by projecting out a variable
- Termination: no variables left.

If the original system has solutions, so does the reduced system. How about the converse? The answer is MAYBE.

Question: Construct a system such that this system has no integer solution, but its reduced system has integer solutions.

Definition: Two systems are said to be equivalent if they have the same solutions.

Question: How to prove that the two systems are equivalent?

2.2 Infeasible System

A system is infeasible if we can derive a contradiction from the system.

Example: consider the system

$$x1 \ge 0,$$
$$x2 \ge 0,$$
$$x1 + x2 \le -2.$$

Eliminating x1 gives

$$x2 \le -2 \quad \text{and} \quad -x2 \le 0.$$

and subsequently eliminating x2 gives

$$0 \le -2,$$

which is a contradiction. Therefore the system is infeasible.

Z = set of integer

Chapter 3 Integer Programming

An integer programming problem in which all variables are required to be integer is called a *pure integer programming problem*. If some variables are restricted to be integer and some are not then the problem is a *mixed integer programming problem*. The case where the integer variables are restricted to be 0 or 1 comes up surprising often. Such problems are called *pure (mixed) 0-1 programming problems* or *pure (mixed) binary integer programming problems*.

To solve these programming, there are two common approaches. Historically, the first method developed was based on cutting planes (adding constraints to force integrality). In the last twenty years or so, however, the most effective technique has been based on dividing the problem into a number of smaller problems in a method called *branch and bound*. Recently (the last ten years or so), cutting planes have made a resurgence in the form of facets and polyhedral characterizations. All these approaches involve solving a series of *linear* programs.

3.1 Formulation

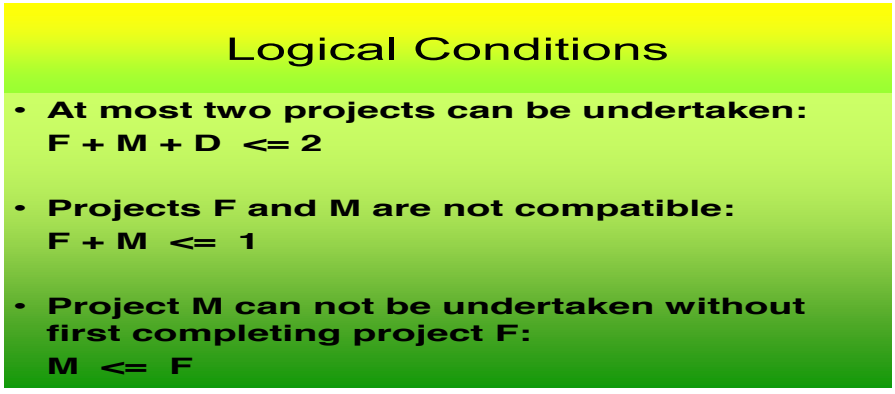
To model a situation, we follow the following approach:

1. Introducing **variables**
2. Establishing all the **constraints**
3. Constructing the **objective**.

A useful tip when formulating Programming is to express the variables, constraints and objective in words before attempting to express them in mathematics.

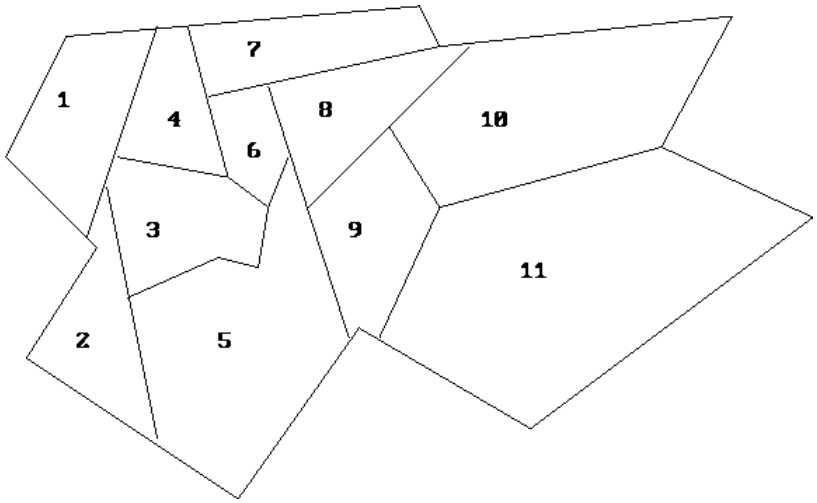
To get constraints, we translate verbal descriptions into an equivalent mathematical descriptions.

Suppose we have three projects: F, M, D. The value 1 means selecting the project, and 0 means not selecting.



3.2 Practical questions

Example 1. A city is reviewing the location of its fire stations. The city is made up of a number of regions, as illustrated in the following Figure. A fire station can be placed in any region. It is able to handle the fires for both its region and any adjacent region (any other region which share part of borders with it). The objective is to minimize the number of fire stations used. should be the function



Solution: We can create one variable x_j for each neighborhood *j*. This variable will be 1 if we place a station in the neighborhood, and will be 0 otherwise. This leads to the following formulation

Objective Function

$$\text{Minimize } x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10} + x_{11}$$

$$\begin{aligned}
\text{subject to } & x_1 + x_2 + x_3 + x_4 \geq 1 \\
& x_1 + x_2 + x_3 + x_5 \geq 1 \\
& x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \geq 1 \\
& x_1 + x_3 + x_4 + x_6 + x_7 \geq 1 \\
& x_2 + x_3 + x_5 + x_6 + x_8 + x_9 \geq 1 \\
& x_3 + x_4 + x_5 + x_6 + x_7 + x_8 \geq 1 \\
& x_4 + x_6 + x_7 + x_8 \geq 1 \\
& x_5 + x_6 + x_7 + x_8 + x_9 + x_{10} \geq 1 \\
& x_5 + x_8 + x_9 + x_{10} + x_{11} \geq 1 \\
& x_8 + x_9 + x_{10} + x_{11} \geq 1 \\
& x_9 + x_{10} + x_{11} \geq 1
\end{aligned}$$

give all possible constraint

$$x_j \in \{0,1\} \quad j = 1, \ldots, 11$$

The first constraint states that there must be a station either in neighborhood 1 or in some adjacent neighborhood. The next constraint is for neighborhood 2 and so on. Notice that the constraint coefficient a_{ij} is 1 if neighborhood *i* is adjacent to neighborhood *j* or if *i=j*

and 0 otherwise. The *j*th column of the constraint matrix represents the set of neighborhoods that can be served by a fire station in neighborhood *j*. We are asked to find a set of such subsets *j* that *covers* the set of all neighborhoods in the sense that every neighborhood appears in the service subset associated with *at least* one fire station.

One optimal solution to this is $x_3 = x_8 = x_9 = 1$ and the rest equal to 0.

Example 2. Suppose we wish to invest $14,000. We have identified four investment opportunities. Investment 1 requires an investment of $5,000 and has a present value (a time-discounted value) of $8,000; investment 2 requires $7,000 and has a value of $11,000; investment 3 requires $4,000 and has a value of $6,000; and investment 4 requires $3,000 and has a value of $4,000. Into which investments should we place our money so as to maximize our total present value?

Solution: Our first step is to decide on our variables. This can be much more difficult in integer programming because there are very clever ways to use integrality restrictions. In this case, we will use a 0-1 variable xj for each investment. If xj = 1, then we will make investment *j*. If it is 0, we will not make the investment. This leads to the 0-1 programming problem:

$$\begin{array}{ll} \text{Maximize} & 8x_1 + 11x_2 + 6x_3 + 4x_4 \\ \text{subject to} & 5x_1 + 7x_2 + 4x_3 + 3x_4 \leq 14 \\ & x_j \in \{0,1\}\ j = 1, \ldots 4. \end{array}$$

方程已约分1000

Now, ignoring integrality constraints, the optimal linear programming solution is

x1= x2 = 1, x3 = 0.5, x4 = 0 x must be a integer

for a value of $22,000. Unfortunately, this solution is not integral. If you round x3=1, then you have a contradiction; if you round x3=0, then you will get a feasible solution with a value of $19,000. However, this value is not best, since we will get a value of $21,000 when x1=0, x2=x3=x4=1.

3.3 Branch and bound

Only discuss 2 branches

Problem Max 15x1+24x2

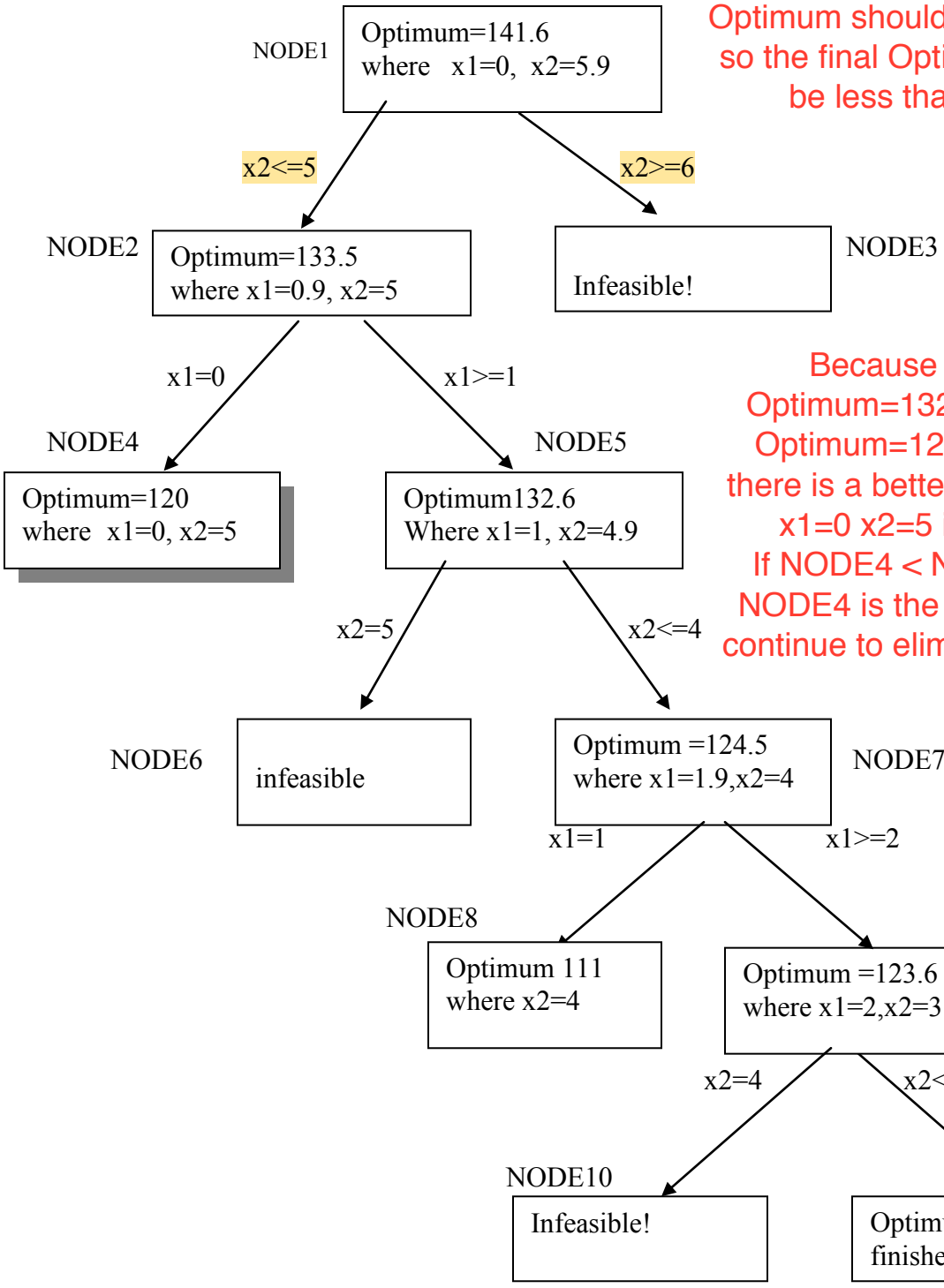
s.t. 18x1+10x2<=90

20x1+20x2<=118

x1, x2>=0 integer

1) Use Fourier elimination at every node:

If x1=0.1 x2=5.9 then pick up x1 or x2 to discuss
!!! Do not combine x1 and x2 (will have 4 situation)
This method only have 2 branches.



Optimum should be a integer, so the final Optimum should be less than 141.6

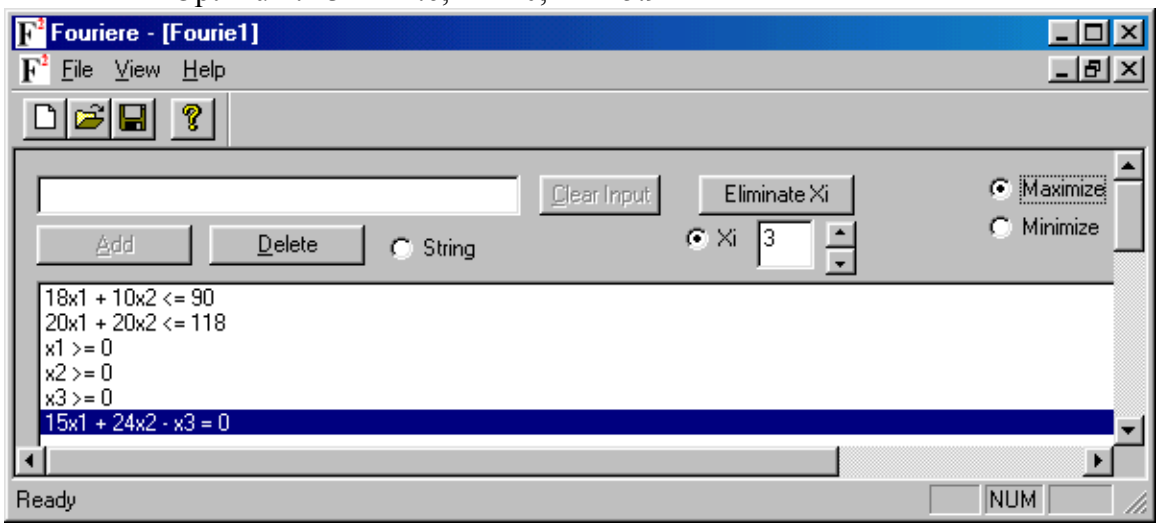
Because NODE5 Optimum=132.6 > NODE4 Optimum=120, so maybe there is a better solution than x1=0 x2=5 in NODE5.
If NODE4 < NODE5, then NODE4 is the solution, else continue to eliminate NODE5.

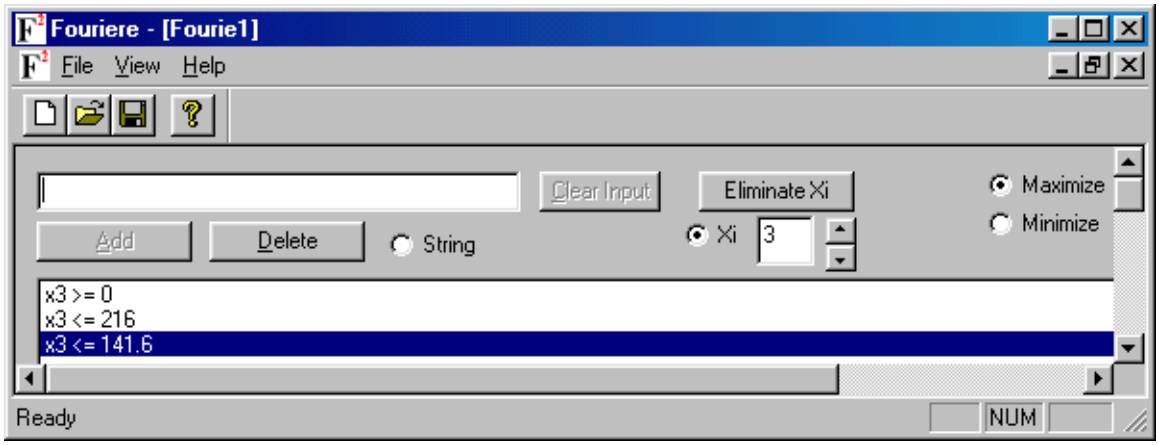
NODE11 115.5 < NODE4 120, so finish.

Inputs and optimum values for all the nodes above:

NODE1:

Max x3 = 15x1+24x2
s.t. 18x1+10x2 <= 90
20x1+20x2 <= 118
x3, x1, x2 >= 0
Optimum: x3 =141.6, x1 = 0, x2 = 5.9





NODE2:

Max x3 = 15x1+24x2
s.t. 18x1+10x2 <= 90
20x1+20x2 <= 118
x3, x1, x2 >= 0
x2 <= 5
Optimum: x3 = 133.5, x1 = 0.9, x2 = 5

NODE3:

Max x3 = 15x1+24x2
s.t. 18x1+10x2 <= 90
20x1+20x2 <= 118
x3, x1, x2 >= 0
x2 >= 6

infeasible!

NODE4:

Max x3 = 15x1+24x2
s.t. 18x1+10x2 <= 90
20x1+20x2 <= 118
x3, x1, x2 >= 0
x2 <= 5
x1 = 0

Optimum: x3 = 120, x1 = 0, x2 = 5

NODE5:

Max x3 = 15x1+24x2
s.t. 18x1+10x2 <= 90
20x1+20x2 <= 118
x3, x1, x2 >= 0
x2 <= 5
x1 >= 1

Optimum: x3 = 132.6, x1 = 1, x2 = 4.9

NODE6:

Max x3 = 15x1+24x2
s.t. 18x1+10x2 <= 90
20x1+20x2 <= 118
x3, x1, x2 >= 0
x2 <= 5
x1 >= 1
x2 = 5

infeasible!

NODE7:

Max x3 = 15x1+24x2
s.t. 18x1+10x2 <= 90
20x1+20x2 <= 118
x3, x1, x2 >= 0
x2 <= 5
x1 >= 1
x2 <= 4

Optimum: x3 = 124.5, x1 = 1.9, x2 = 4

NODE8:

Max x3 = 15x1+24x2
s.t. 18x1+10x2 <= 90
20x1+20x2 <= 118
x3, x1, x2 >= 0
x2 <= 5
x1 >= 1
x2 <= 4
x1 = 1

Optimum: x3 = 111, x1 = 1, x2 = 4

NODE9:

Max x3 = 15x1+24x2
s.t. 18x1+10x2 <= 90
20x1+20x2 <= 118
x3, x1, x2 >= 0
x2 <= 5
x1 > = 1
x2 <= 4
x1 >= 2

Optimum: x3 = 123.6, x1 = 2, x2 = 3.9

NODE10:

Max x3 = 15x1+24x2
s.t. 18x1+10x2 <= 90
20x1+20x2 <= 118
x3, x1, x2 >= 0
x2 <= 5
x1 >= 1
x2 <= 4
x1 >= 2
x2 = 4

infeasible!

NODE11:

Max x3 = 15x1+24x2
s.t. 18x1+10x2 <= 90
20x1+20x2 <= 118
x3, x1, x2 >= 0
x2 <= 5
x1 >= 1
x2 <= 4
x1 >= 2
x2 <= 3

Optimum: x3 = 115.5, x1 = 2.9, x2 = 3

FINISHED!

3.4 Cutting Plane Techniques 平面切割

There is an alternative to branch and bound called **cutting planes** which can also be used to solve integer programs. The fundamental idea behind cutting planes is to add constraints to a linear program until the optimal basic feasible solution takes on integer values. Of course, we have to be careful which constraints we add: we would not want to change the problem by adding the constraints. We will add a special type of constraint called a **cut.** A cut relative to a current fractional solution satisfies the following criteria:

- Every feasible integer solution is feasible for the cut, and
- The current fractional solution is not feasible for the cut.

keep good point
remove bad point

There are two ways to generate cuts. The first, called Gomory cuts, generates cuts from any linear programming tableau. This has the advantage of ``solving" any problem.

1) Gomory Cuts:

Consider the following integer program:

$$\begin{array}{lrcrcl} \text{Maximize} & 7x_1 & + & 9x_2 & & \\ \text{subject to} & -x_1 & + & 3x_2 & \leq & 6 \\ & 7x_1 & + & x_2 & \leq & 35 \\ & x_1, & & x_2 & \geq & 0 \text{ integer.} \end{array}$$

To find Cutting planes, we change the system as follows:

Chang original system to equation

$$7x_1 + 9x_2 - z = 0$$
$$-x_1 + 3x_2 + s_1 = 6$$
$$7x_1 + x_2 + s_2 = 35$$
$$x_1, x_2, s_1, s_2 \geq 0, \text{int } eger.$$

Then we get the following optimal:

Isolate X1 get 2 equations,
Isolate X2 get 2 equations;
get three equations contain S1,S2

通常是-z, because z = ax + by, ax + by - z = 0

Variable	x_1	x_2	s_1	s_2	$-z$	RHS
x_2	0	1	7/22	1/22	0	7/2
x_1	1	0	-1/22	3/22	0	9/2
$-z$	0	0	28/11	15/11	1	63

Get 3 equations:
X2+7/22*S1+1/22*S2=7/2
X1-1/22*S1+3/22*S2=9/2
28/11*S1+15/11*S2-Z=63

Let's look at the first constraint:

$$x_2 + 7/22s_1 + 1/22s_2 = 7/2$$

We can manipulate this to put all of the integer parts on the left side, and all the fractional parts on the right to get:

$$x_2 - 3 = 1/2 - 7/22s_1 - 1/22s_2$$

Now, note that the left hand side consists only of integers, so the right hand side must add up to an integer. The right hand side can only be

$$0, -1, -2, \ldots$$

Therefore, we have derived the following constraint:

$$1/2 - 7/22s_1 - 1/22s_2 \leq 0.$$

This constraint is satisfied by every feasible integer solution to our original problem. Hence it is a cut. We can now add this constraint to the linear program.
We can also generate a cut from the other constraint. Here we have to be careful to get the signs right:

$$\begin{aligned} x_1 - 1/22s_1 + 3/22s_2 &= 9/2 \\ x_1 + (-1 + 21/22)s_1 + 3/22s_2 &= 4 + 1/2 \\ x_1 - s_1 - 4 &= 1/2 - 21/22s_1 - 3/22s_2 \end{aligned}$$

gives the constraint

$$1/2 - 21/22s_1 - 3/22s_2 \leq 0.$$

In general, let [b] be defined as the largest integer less than or equal to b.
If we have a constraint

$$x_k + \sum a_i x_i = b$$

with b not an integer, we can write each

$$a_i = \lfloor a_i \rfloor + a_i' \qquad \text{eg: } [3.8]=3,\ [3.1]=3$$
$$[-3.0]=-3,\ [-3.8]=-4$$

for some

$$0 \le a_i' < 1$$

and

b=[b]+b'

for some 0<=b'<1. Using the same steps we get:

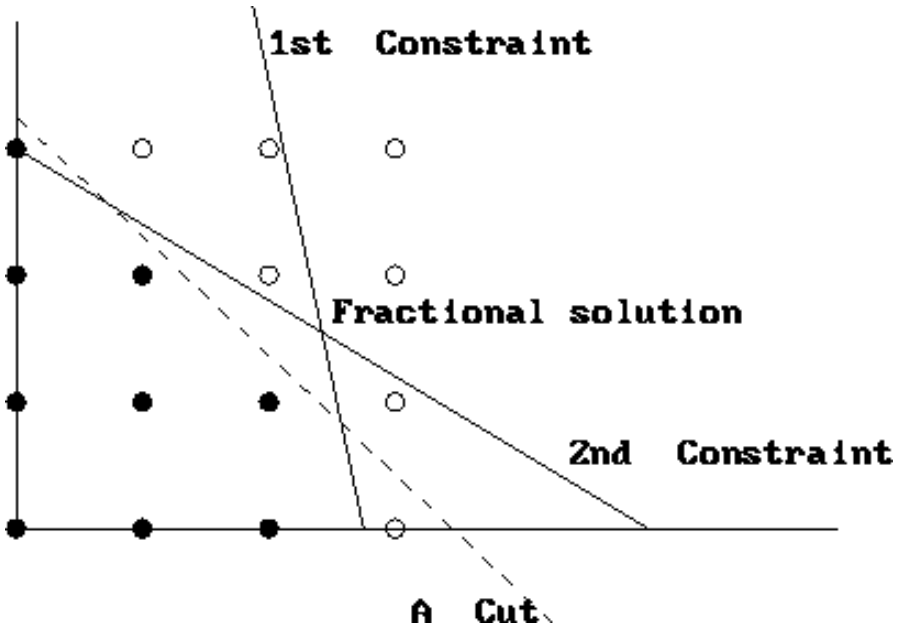
$$x_k + \sum \lfloor a_i \rfloor x_i - \lfloor b \rfloor = b' - \sum a_i' x_i$$

to get the cut

$$b' - \sum a_i' x_i \le 0.$$

This cut can then be added to the linear program and the problem resolved.

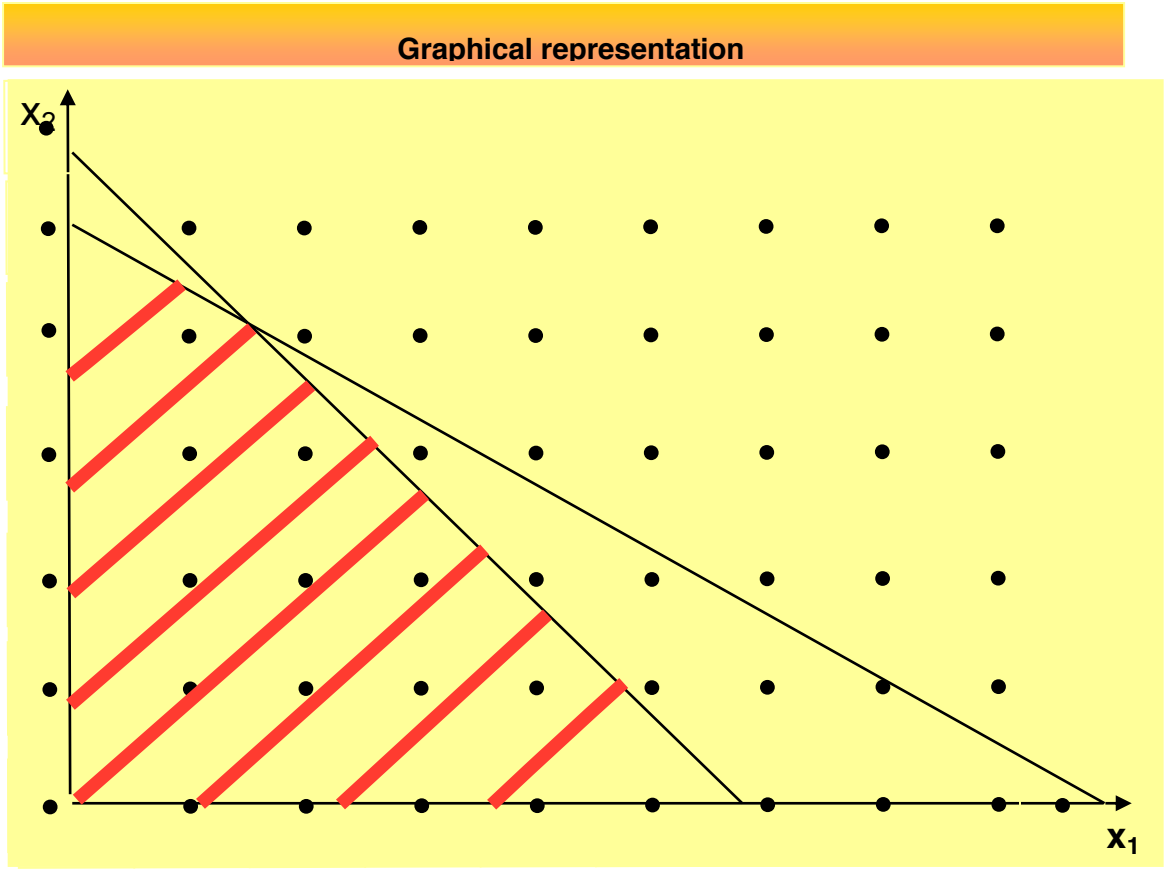
2) The second approach is to use the structure of the problem to generate very good cuts. The approach needs a problem-by-problem analysis, but can provide very efficient solution techniques.

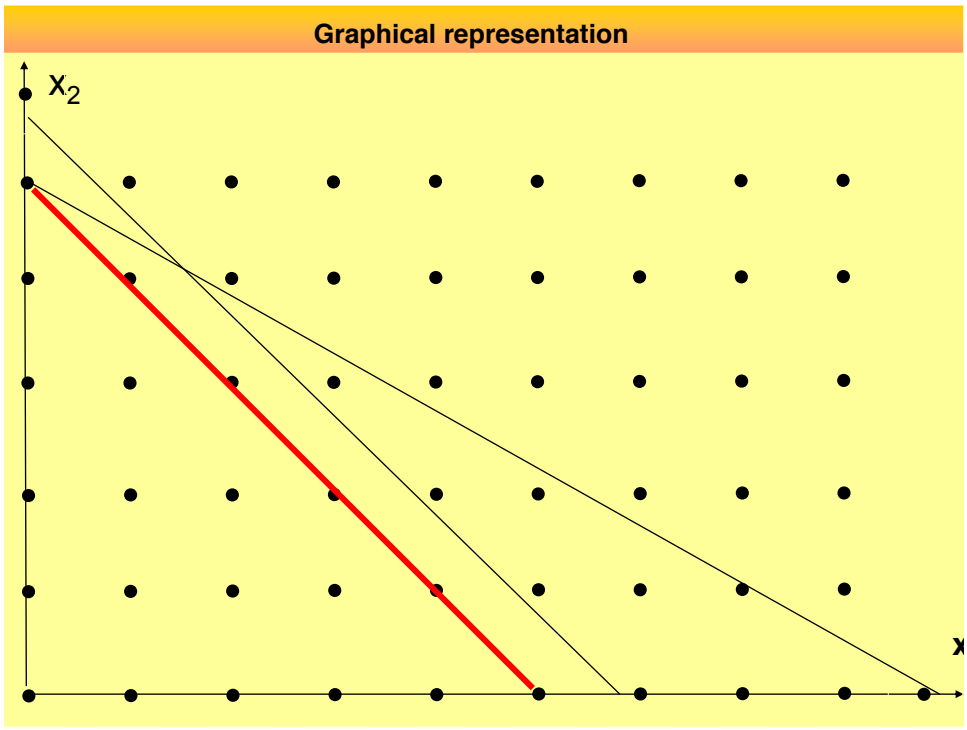


Cutting Planes for 0,1 Programs

Max x3 = 5 x1 + 8 x2
s.t.

10x1 + 10x2 <= 59
5x1 + 9 x2 <= 45
x1 >= 0 integer
x2 >= 0 integer





Elimination of Integrality Requirements

Max x3 = 5 x1 + 8 x2

s.t.

10x1 + 10x2 <= 59

5x1 + 9 x2 <= 45

10x1 + 10x2 <= 50 (or x1 + x2 <= 5)

x1 >= 0 integer

x2 >= 0 integer

Elimination of Integrality Requirements

$$\max \quad x_3 = 5\,x_1 + 8\,x_2$$

$$\text{s.t.}$$

$$x_1 + x_2 \leq 5$$

$$x_1 \geq 0$$

$$x_2 \geq 0$$

Chapter 4 Network

Network optimization is a special type of linear programming model. Network models have three main advantages over linear programming:

traffic network, social network

- They can be solved very quickly. This allows network models to be used in many applications (such as real-time decision making) for which linear programming would be inappropriate.
- Have integer solutions.
- They are intuitive. Network models provide a language that is much more intuitive than linear and integer programming.

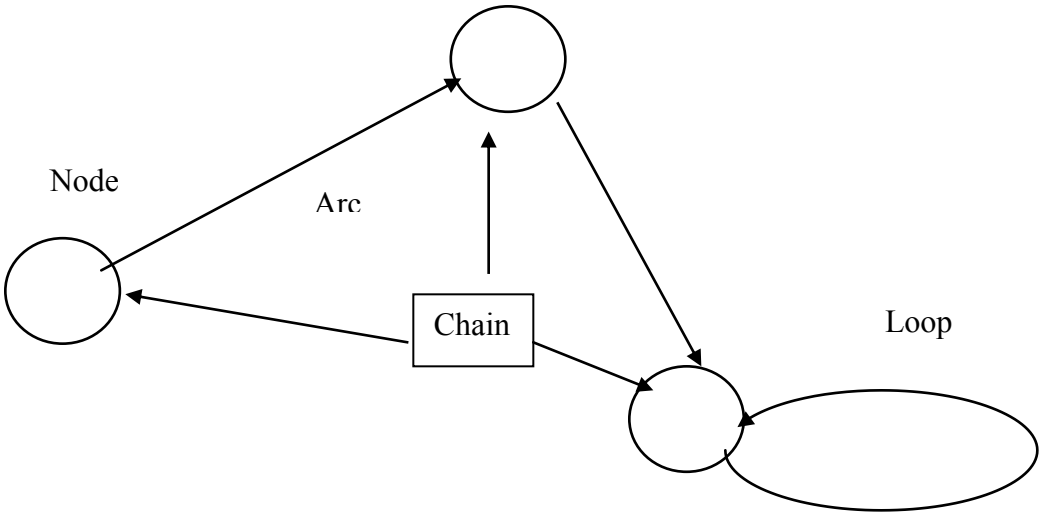
Of course these advantages come with a drawback: network models cannot formulate the wide range of models that linear and integer programs can.

4.1 Introduction

Networks are made up of Nodes connected by Arcs (also known as Links, or Routes or Edges).

- Node represents a location (city, warehouse etc.), which is identified by a unique number.
- Arc: a process, or a road, or a link between two nodes.
- Directed arc: Arc with direction.
- Directed network: All the arcs in the network are directed.
- Edge: Arc without direction.
- Undirected network: All the arcs in the network are undirected. (But arcs here may have direction pointed to both ends).
- Graph: a collection of nodes.
- Loop: an arc with identical origin and destination.
- Capacity: limiting the amount of flow that it can process.
- Cost: fixed cost (constructing the arc, etc.), variable cost. The cost can represent a distance, a penalty, etc., can be negative such as revenues, utinities.
- Multiplier: can be a physical loss, a logical multiplication.
- Source: a node with no inward arcs.
- Sink: a node with no outward arcs.
- Path: a series of nodes connected by directed arcs.
- Adjacent nodes: two nodes are connected by an arc.
- Chain: a series connected nodes.
- Flow: vehicles, telephone calls, water, etc.

Cost can be positive, negative and 0



There are four classic Network Problems:

- Minimisation (the tree spanning problem);
- Shortest Route;
- Critical path method (CPM);
- Maximal flow.

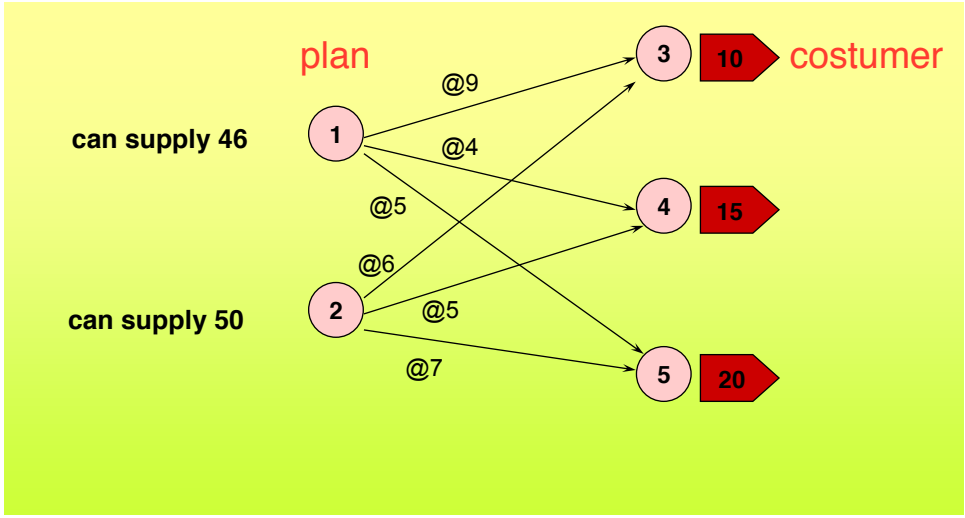
4.2 Network flow with fixed cost

A company has two plants producing motors that must be shipped to three customers. Plants 1, and 2 can produce 46, and 50 motors per month, respectively. Production start-up costs are $150K and $260K respectively (which are not incurred if the plant is closed). Customers 1, 2 and 3 need to receive 10, 15 and 20 motors per month, respectively. The shipping cost from each plant to the respective customers follows:

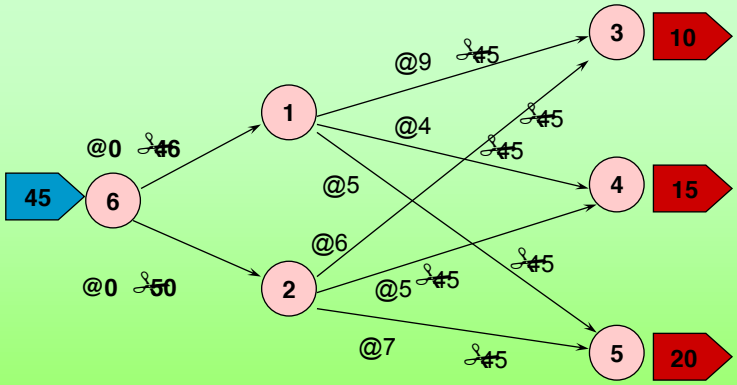
每一辆motor的shipping cost是9 units

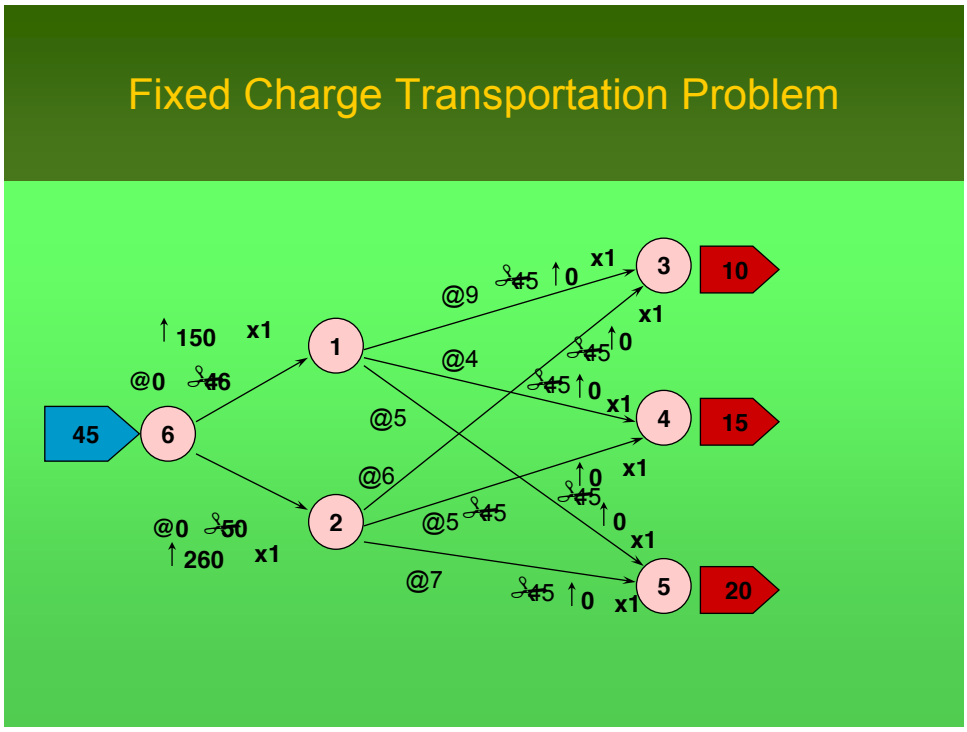
	Customer 1	Customer 2	Customer 3
Plant 1	9	4	5
Plant 2	6	5	7

Graphic View of the Transportation Problem

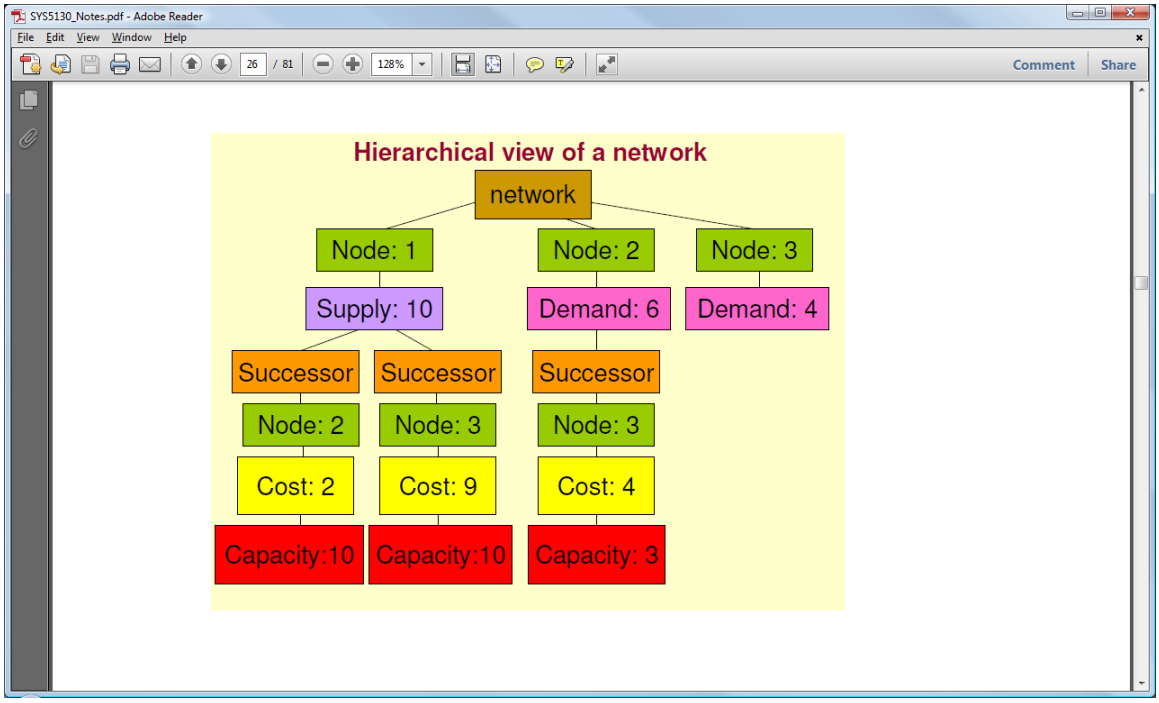


Graphic View of the Transportation Problem





4.3 Hierarchy



Different relationships in a hierarchy

Network

Node 1: <---- origin
supply: 10
succeeding arc: Node: 2 <---- destination
Cost 2
Capacity 10
succeeding arc: Node: 3 <---- destination
Cost 9
Capacity 10
Node 2: <---- origin
demand: 6
succeeding arc: Node: 3 <---- destination
Cost 4
Capacity 3
Node 3: <---- origin
demand: 4

4.4 Economic systems simulation

An **economic system** is the combination of the various agencies, entities (or even sectors) that provide the economic structure that guides the social community.

An economic system is composed of people, institutions, rules, and relationships. For example, the convention of property, the institution of government, or the employee-employer relationship. Examples of contemporary economic systems include capitalist systems, socialist systems, and mixed economies.

Example. Exchange rate

	USD	CAD	EURO	RMB
USD	1	1.0251	1.4126	0.1566
CAD	0.9752	1	1.3968	0.1549
EURO	0.7079	0.7159	1	0.1109
RMB	6.3855	6.4578	9.0202	1

Suppose you have 1000CAD and 10000RMB, how much USD we can get with arc capacity 99999.

4.5 The tree spanning problem

A tree is a connected graph without cycles. A *spanning tree* of a graph is just a sub-graph that contains all the vertices and is a tree. A graph may have many spanning trees. The standard application is to a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect them up with each other;

and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. It should be a spanning tree, since if a network isn't a tree you can always remove some edges and save money.

Conditions for Minimum Spanning Tree:

- A tree with Minimum total length
- Collects all the nodes in the graph
- No cycles

Prim's Algorithm

- Step 1

  Pick any vertex as a starting vertex. (Call it S). Mark it with any given colour, say red.

- Step 2

  Find the nearest neighbour of S (call it P_1). Mark both P_1 and the edge SP_1 red.

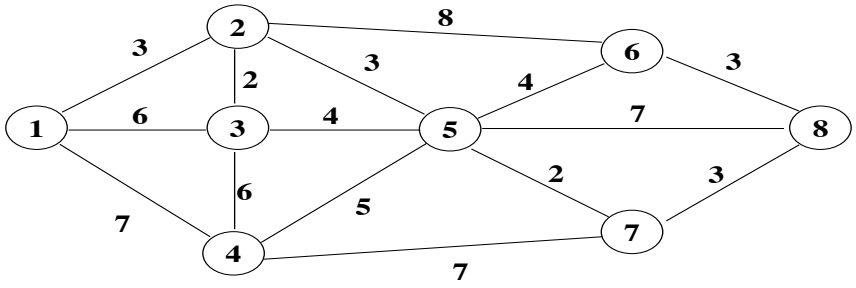
- Step 3

  Find the nearest uncoloured neighbour to the red subgraph (i.e., the closest vertex to any red vertex), without creating a cycle. Mark it and the edge connecting the vertex to the red subgraph in red.

- Step 4

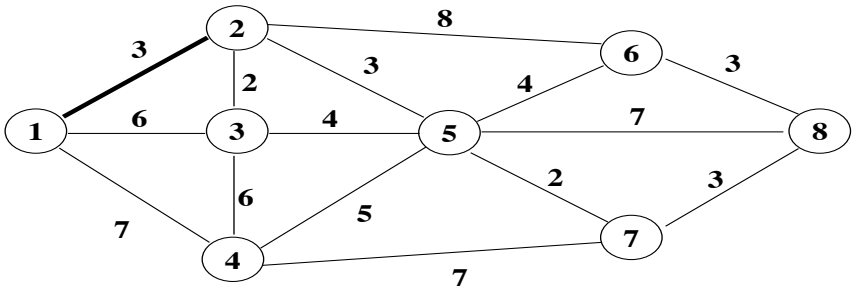
  Repeat Step 3 until all vertices are marked red. The red subgraph is a minimum spanning tree.

(NOTE: Two or more edges may have the same cost, so when there is a choice by two or more vertices that is exactly the same, then one will be chosen, and an MST will still result)



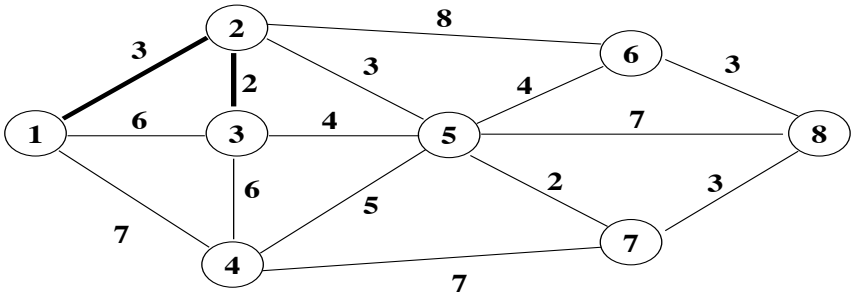
- What is the minimum length necessary to span this network?

Iteration 1:



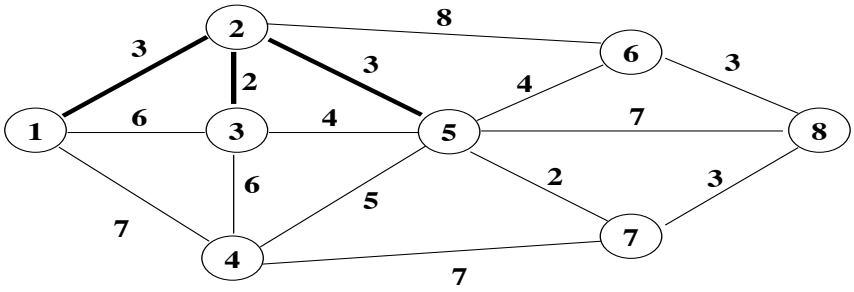
We start from Node 1 (can start from any node), pick the shortest branch and mark it.

Iteration 2:



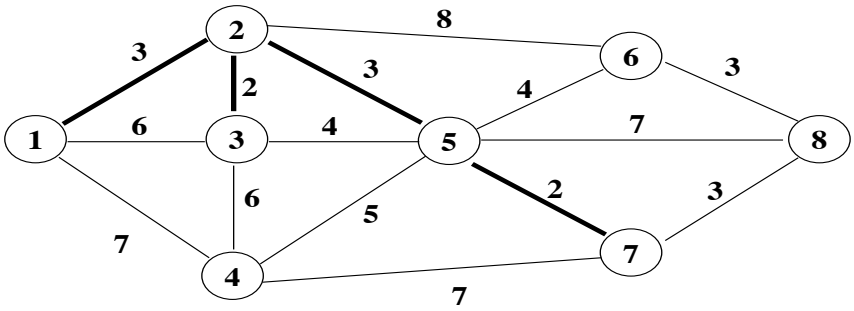
Node 3 is the next nearest node to 2.

Iteration 3:

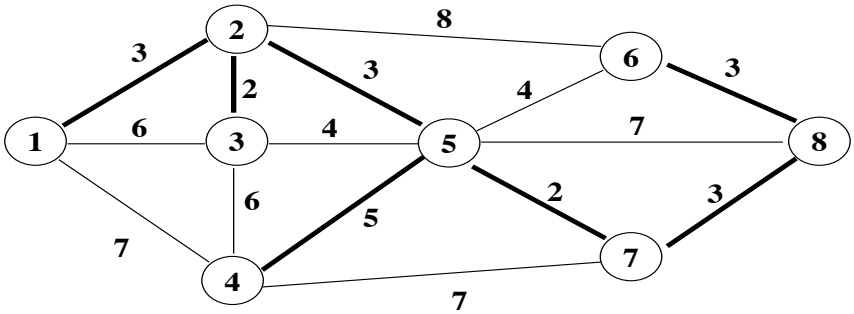


The next nearest node can be joined from any node in the current iteration.

Iteration 4:



Final Solution



4.6 The Shortest Path Problem

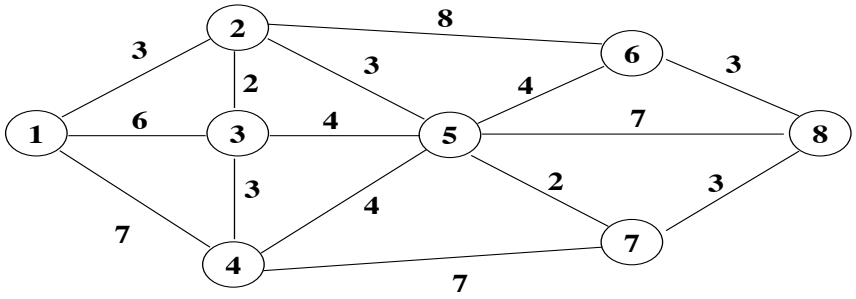
In a network, at any given time, a message may take a certain amount of time to traverse each line (due to congestion effects, switching delays, and so on). This time can vary greatly minute by minute and telecommunication companies spend a lot of time and money tracking these delays and communicating these delays throughout the system. Assuming a centralized switcher knows these delays, there remains the problem of routing a message so as to minimize the delays.

- The objective of the shortest path problem is to find the shortest distance through a network from start to finish.
- It is not necessary to include all nodes in the route.

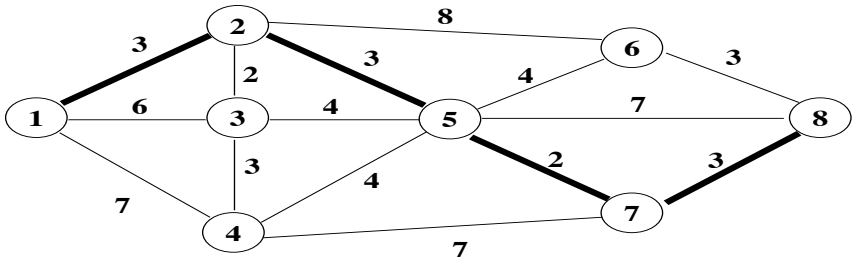
Strategy:

- Identify two sets of nodes -
    1. The Permanent Set and
    2. The Adjacent Set
- Start with the starting node as the permanent set and call it the *Origin*.
- Designate the adjacent set to be all nodes not in the permanent set which can be reached via one link from <u>any</u> node in the permanent set.
- Identify the node in the adjacent set with the shortest distance from the <u>origin</u>.
- Store connecting arcs to this node and delete any other arcs from this node into the permanent set.
- Add the node to the permanent set
- Identify the new adjacent set
- Continue until all nodes are in the permanent set.

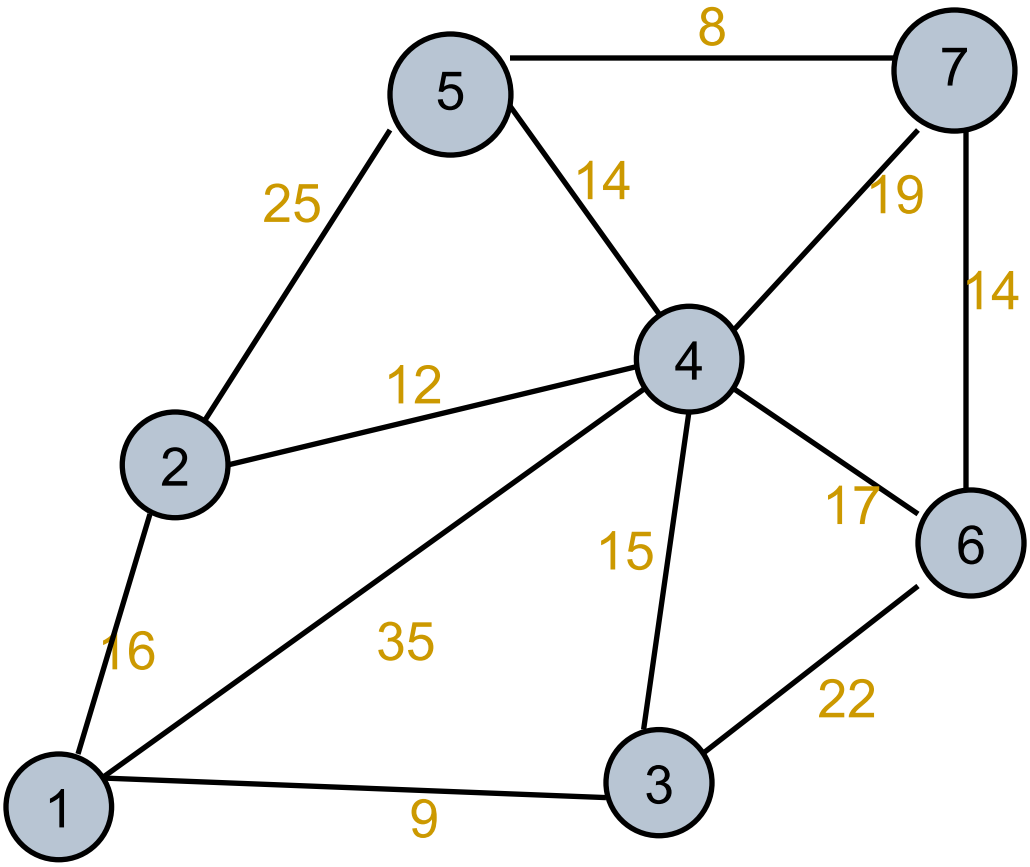
Question: Shortest route from 1 to 8

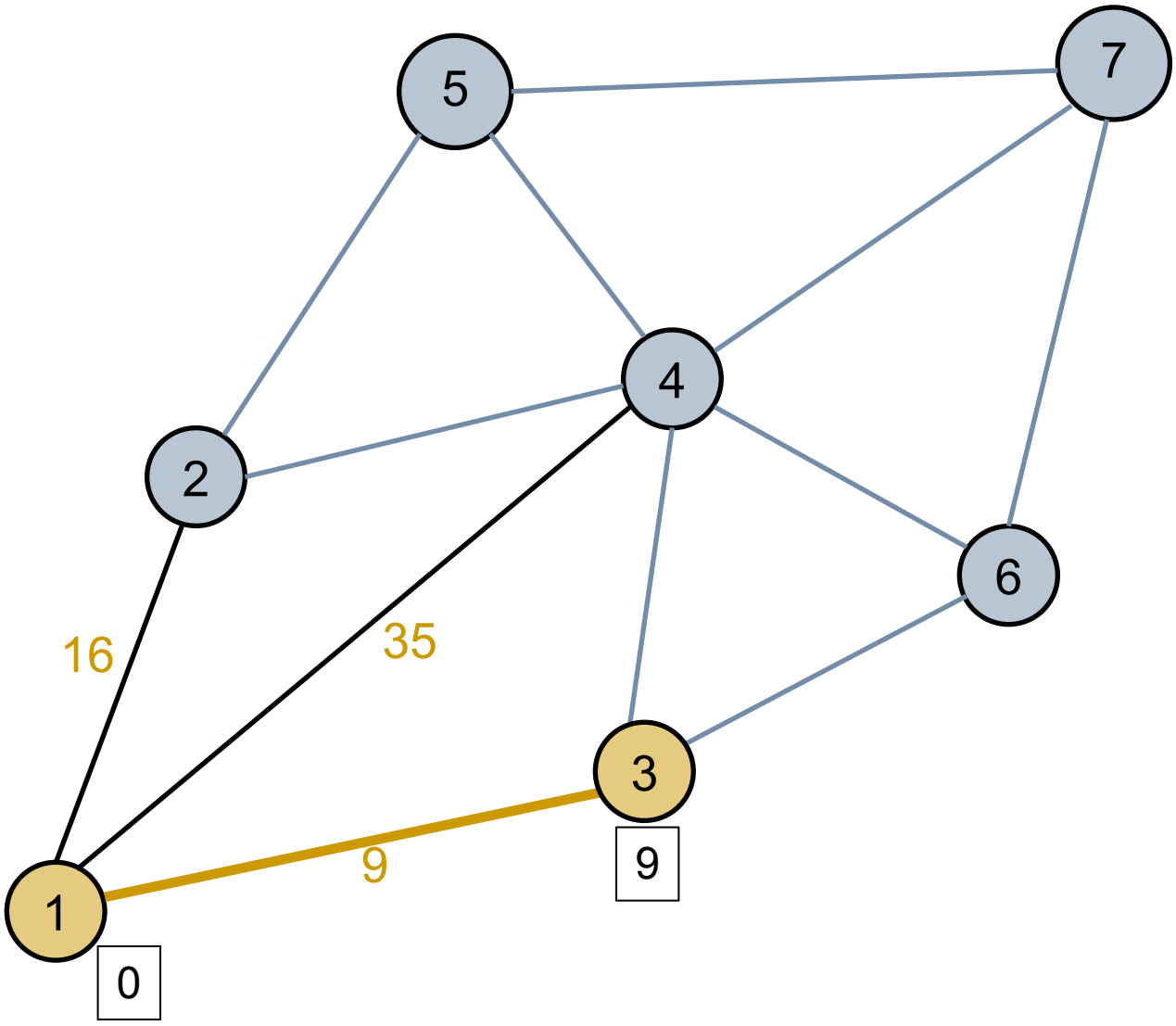


Answer:

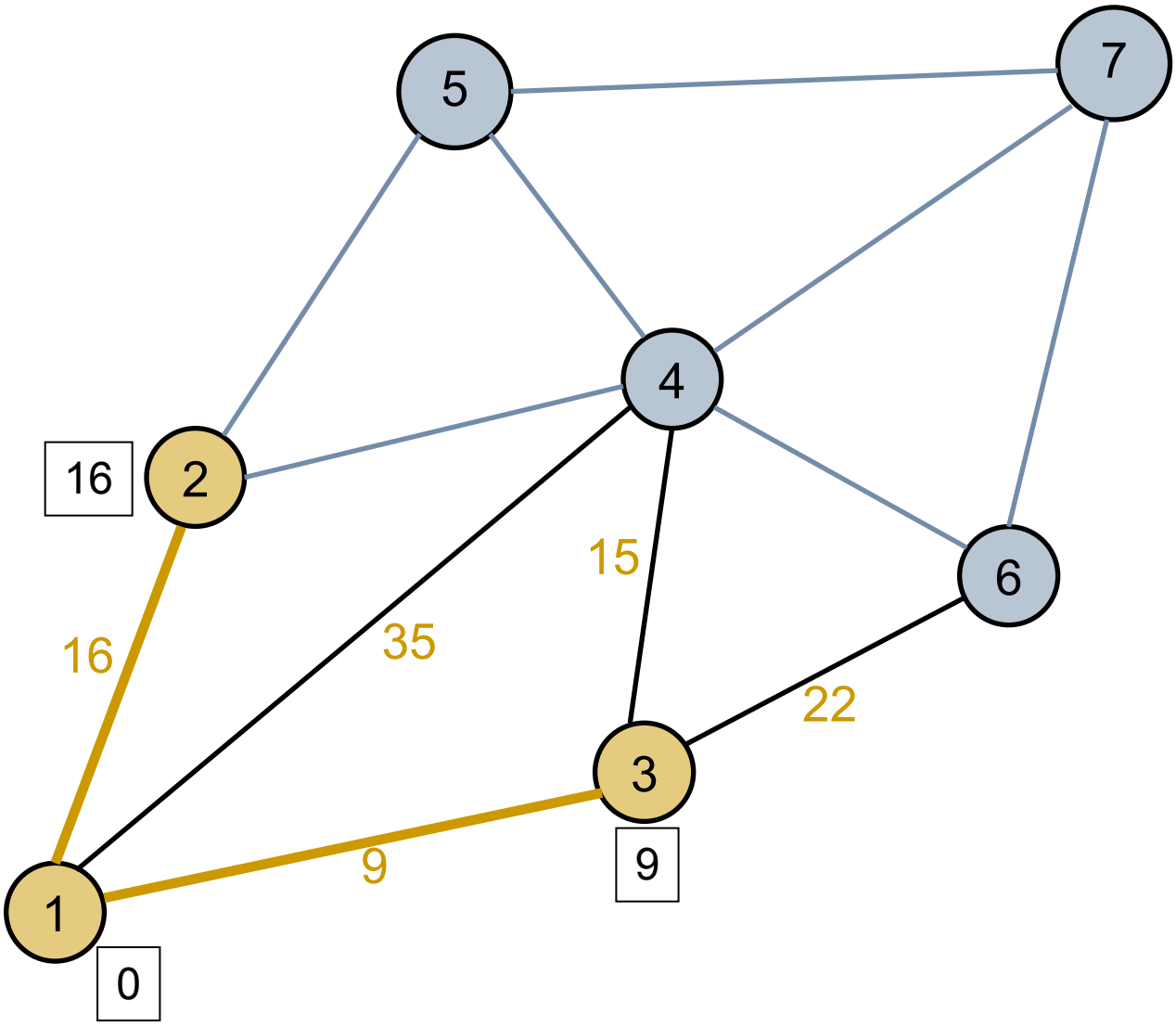


Example. The shipping company manager wants to determine the best routes (in terms of the minimum travel time) for the trucks to take to reach their estinations.

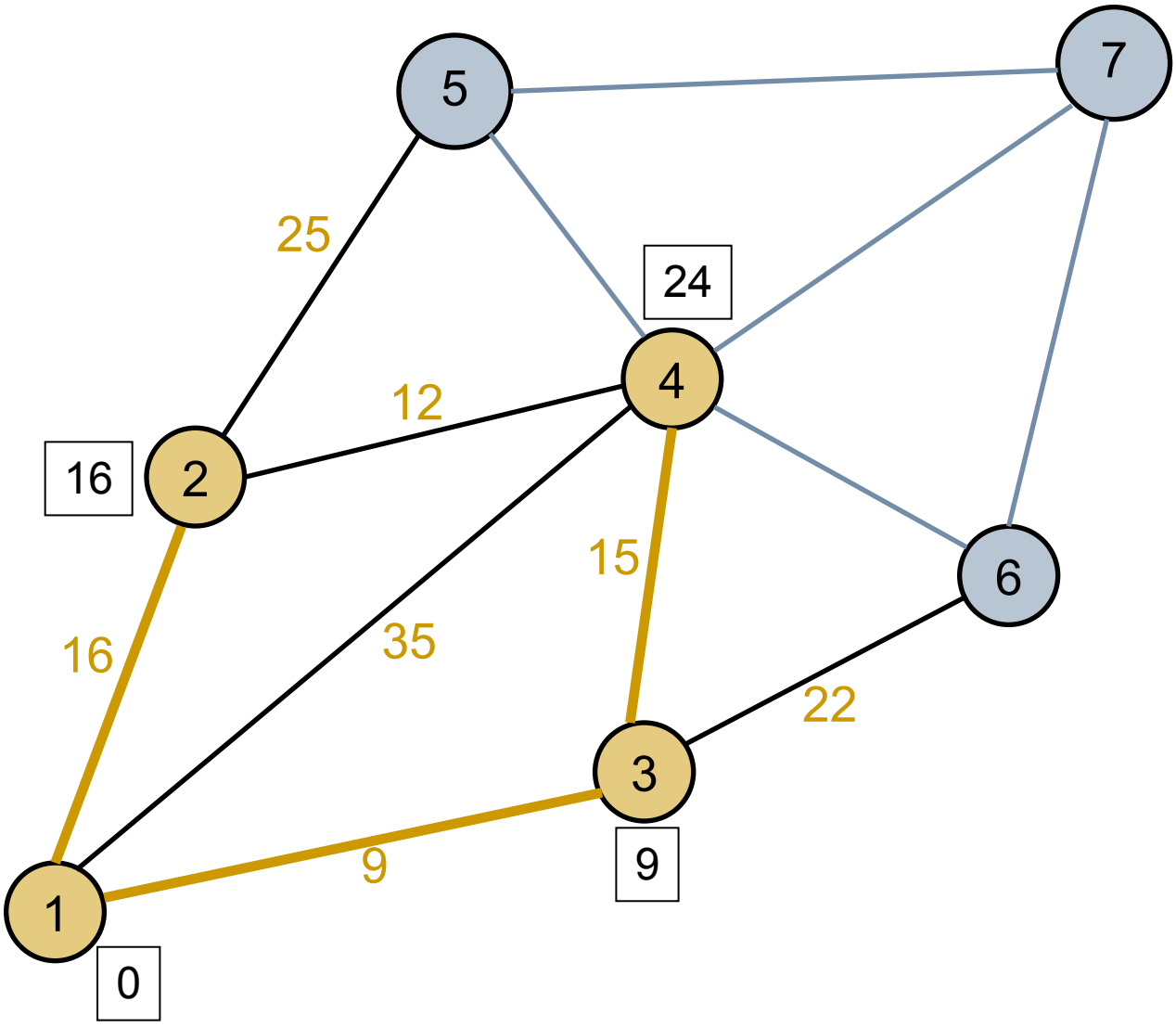

8
5
7
25
14
19
14
4
12
2
17
6
15
16
35
22
3
1
9



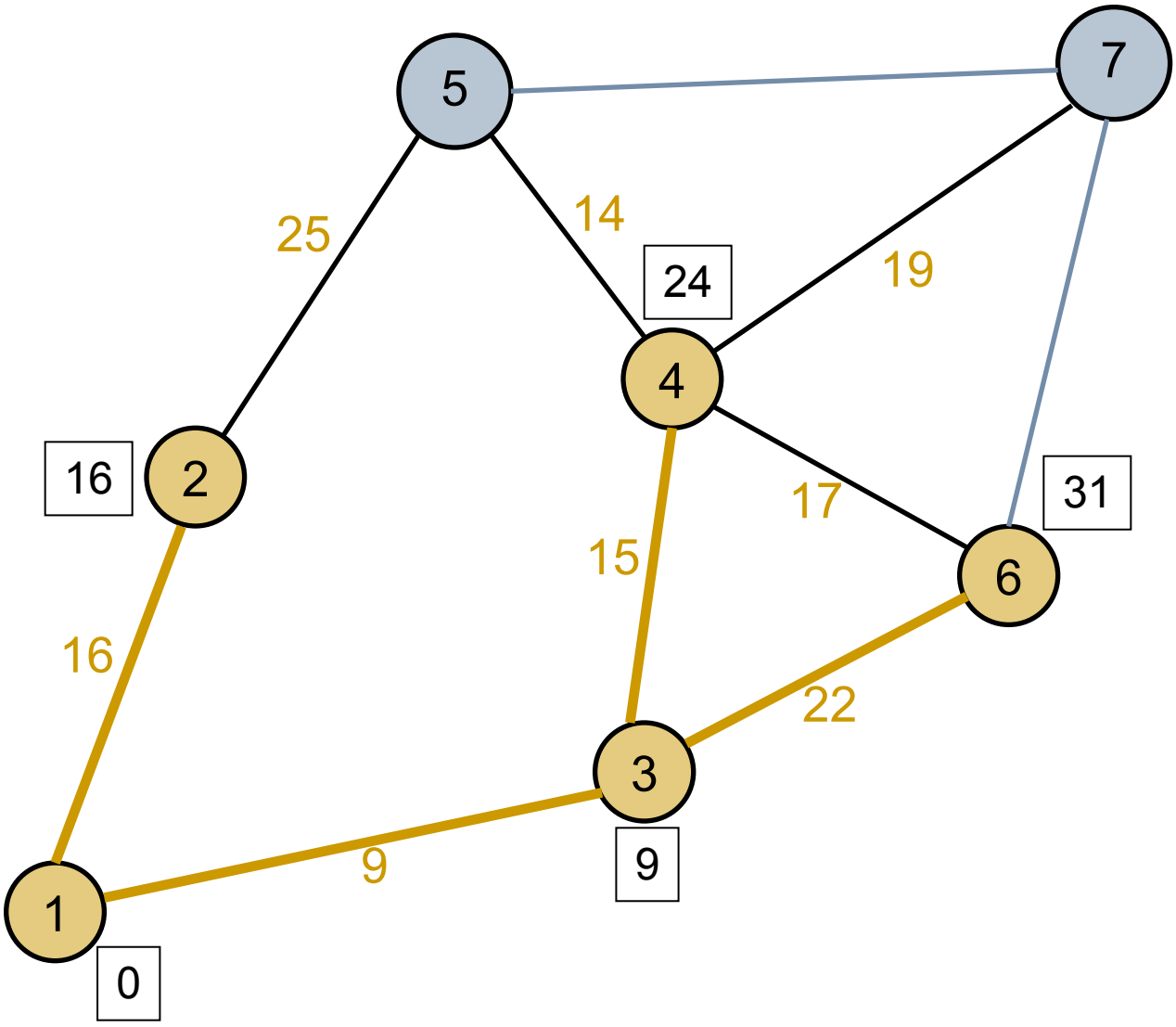
Permanent Set	Arc	Distance
{1}	1-2	16
	1-4	35
	1-3	9



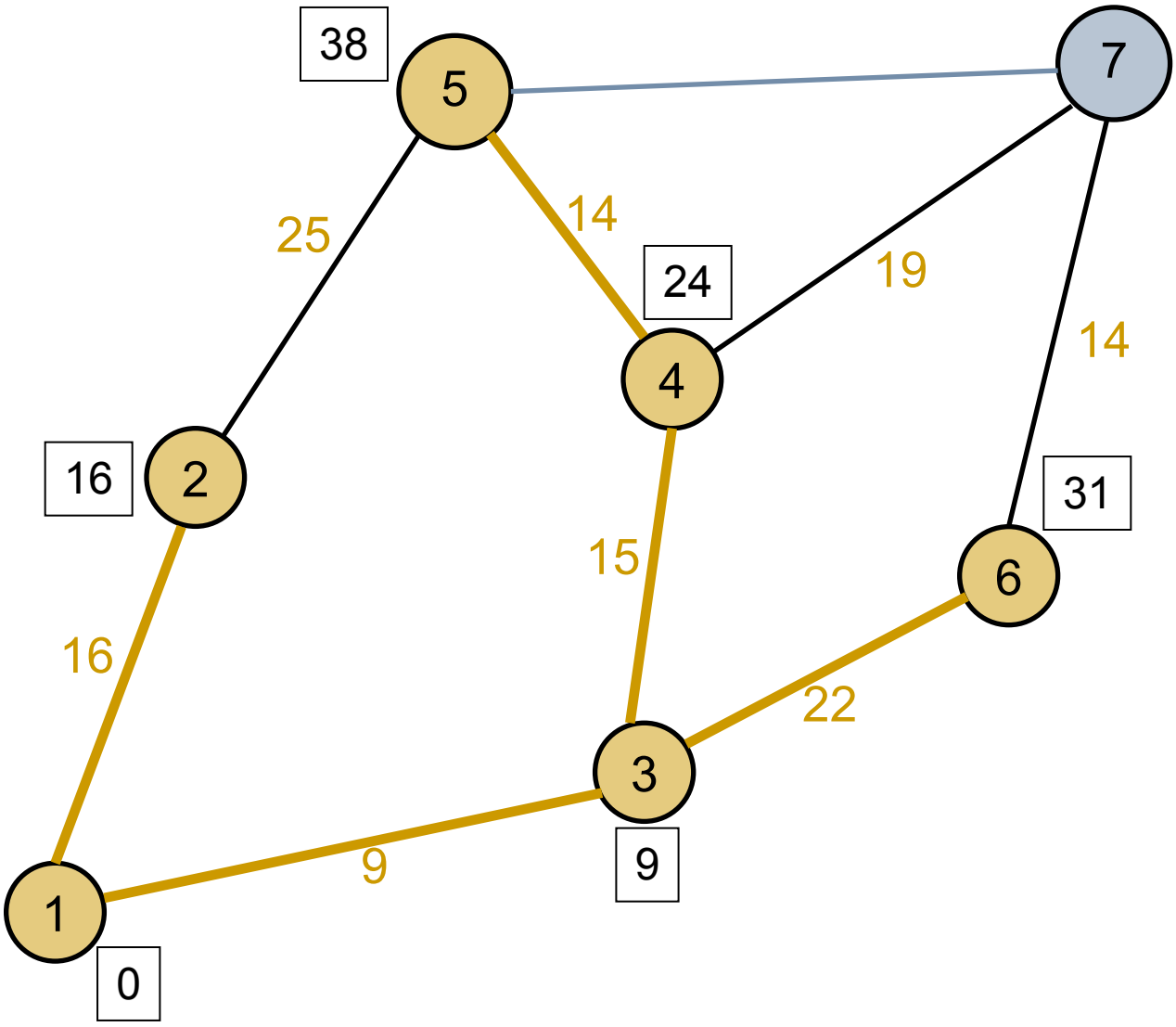
Permanent Set	Arc	Distance
{1, 3}	1-2	16
	1-4	35
	3-4	24
	3-6	31



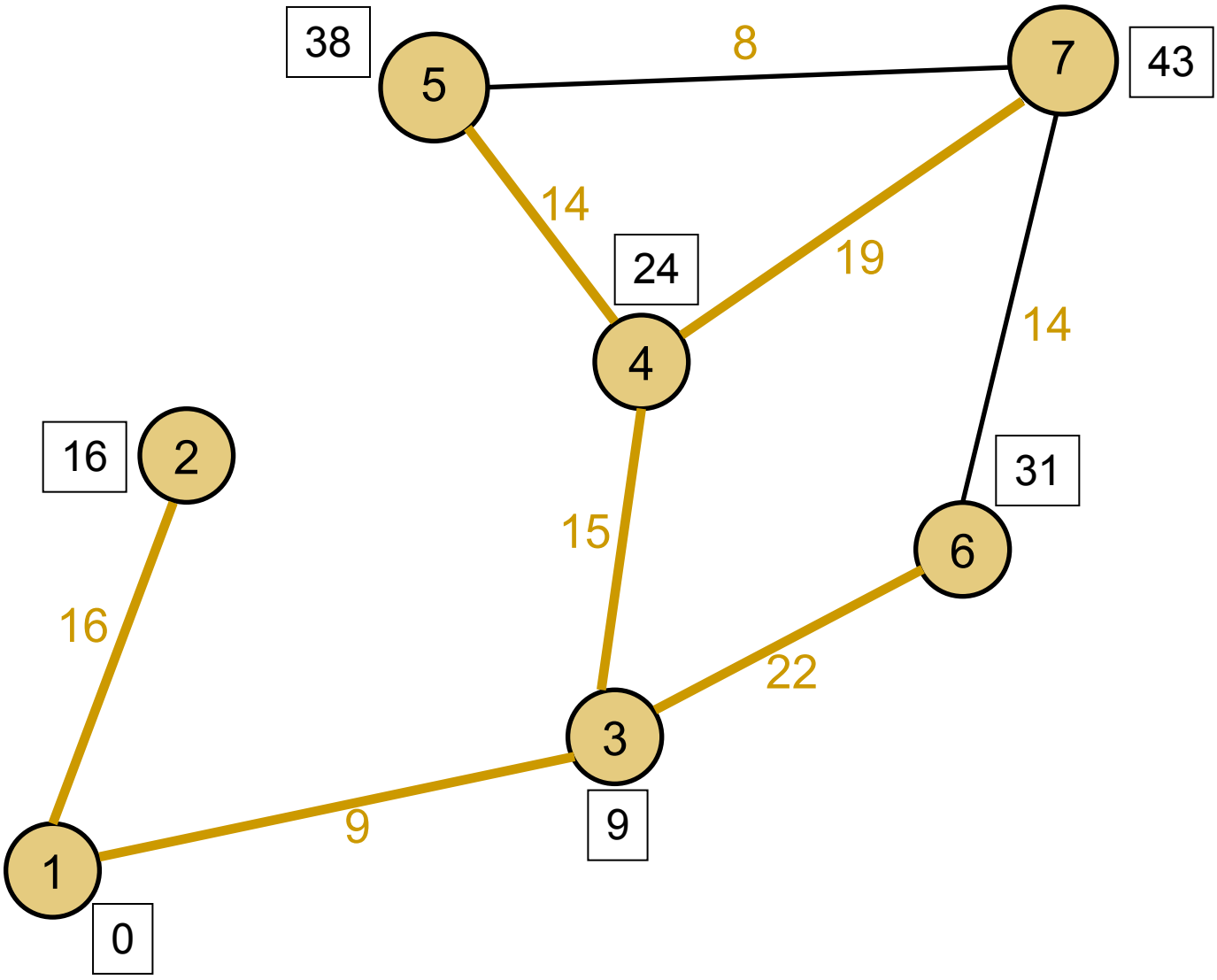
Permanent Set	Arc	Distance
{1, 2, 3}	1-4	35
	2-4	28
	2-5	41
	3-4	24
	3-6	31



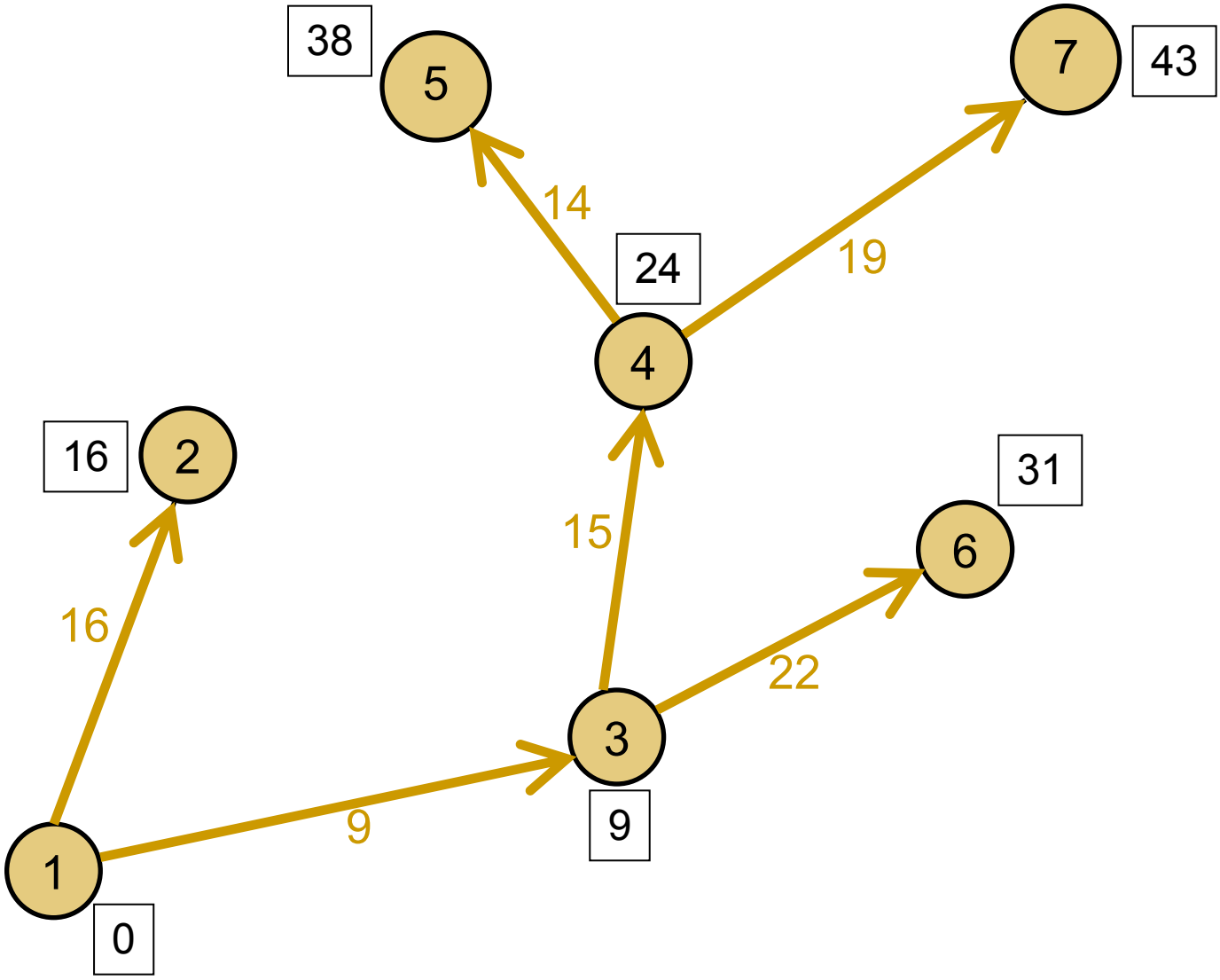
Permanent Set	Arc	Distance
{1, 2, 3, 4}	2-5	41
	3-6	31
	4-5	38
	4-7	43
	4-6	41



Permanent Set	Arc	Distance
{1, 2, 3, 4, 6}	2-5	41
	4-5	38
	4-7	43
	6-7	45



Permanent Set	Arc	Distance
{1, 2, 3, 4, 5, 6}	4-7	43
	6-7	45
	5-7	46



Node 1 to:	Path	Total Distance
Node 2	1-2	16
Node 3	1-3	9
Node 4	1-3-4	24
Node 5	1-3-4-5	38
Node 6	1-3-6	31
Node 7	1-3-4-7	43

4.7 Neural Networks

Neural Networks (NNs) also known as Artificial Neural Networks (ANNs), Connectionist Models, and Parallel Distributed Processing (PDP) Models .

- Characterized by
  - A large number of very simple, neuron-like processing elements called units, or nodes
  - A large number of weighted, directed connections between pairs of units
  - Weights may be positive or negative real values
  - Local processing in that each unit computes a function based on the outputs of a limited number of other units in the network
  - If there are n inputs (x1,…,xn) to a unit, then the unit's output, or activation is defined by

    $$a = g((w1 * x1) + (w2 * x2) + ... + (wn * xn)),$$

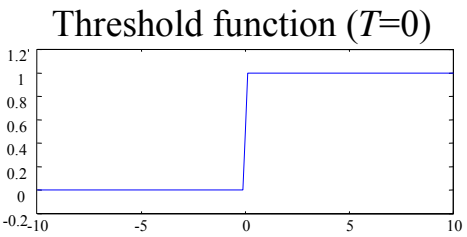
    where wi are weights.
- Some Common Definitions of What a Unit Computes
  - Step (also called Linear Threshold Unit (LTU))

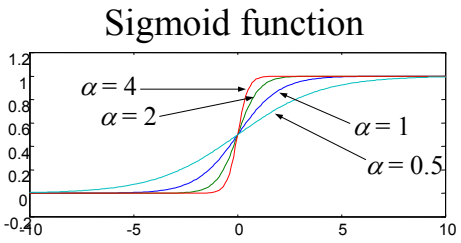
    $$a = \begin{cases} 1, if w_1 x_1 + \ldots + w_n x_n \geq t \\ 0, otherwise \end{cases}$$

  - Sigmoid

    $$a = 1/(1 + e^{w_1 x_1 + \ldots + w_n x_n})$$

Activation function





Why Neural Nets?

- The autonomous local processing of each individual unit combines with similar simple behavior of many other units to produce interesting, complex, global behavior
- Intelligent behavior is an "emergent" phenomenon
- Solving problems using a processing model that is similar to the brain may lead to solutions to complex information processing problems that would be difficult to achieve using traditional symbolic approaches in AI .

Advantages

- Parallel processing
- Distributed representations
- Online (i.e., incremental) algorithm
- Simple computations
- Robust with respect to noisy data and node failure

Disadvantages

- Slow training
- Poor interpretability
- Network topology layouts ad hoc
- Hard to debug
- May converge to a local, not global, minimum of error

Neurobiology Constraints on Human Information Processing

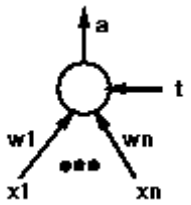
- Number of neurons: 10^{11}
- Number of connections: 10^4 per neuron
- Neuron death rate: 10^5 per day
- Neuron birth rate: 0
- Connection birth rate: very slow
- Performance: about 10^2 msec, or about 100 sequential neuron firings for "many" tasks

Neural network systems can be used:

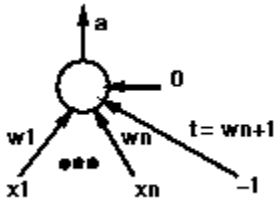
- Where we can't formulate an algorithmic solution.
- Where we can get lots of examples of the behavior we require.
- Where we need to pick out the structure from existing data.

4.7.1 Perceptrons

- Simplest "interesting" class of neural networks
- "1 layer" network -- i.e., one input layer and one output layer. In most basic form, output layer consists of just one unit.
- Linear threshold unit (LTU) used at output layer node(s)

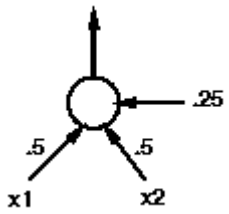


- Threshold associated with LTUs can be considered as another weight. That is, output of a unit is 1 if *w1*x1 + w2*x2 + ... + wn*xn >= t* where *t* is a threshold value. But this is algebraically equivalent to *w1*x1 + w2*x2 + ... + wn*xn + t*(-1) >= 0*. So, in an implementation, consider each LTU as having an extra input which has a constant input value of -1 and the arc's weight is *t*. Now each unit has a fixed threshold value of 0, and *t* is an extra weight called the **bias**.

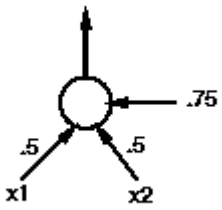


So, from here on learning the weights in a neural net will mean learning the weights and the threshold values.

- Examples
  - OR function
    Two inputs, both binary (0 or 1), and output binary (since output unit is an LTU):



  - AND function



Learning in Neural Nets

- Specifying numbers of units in each layer and connectivity between units, so the only unknown is the set of weights associated with the connections
- Supervised learning of weights from a set of training examples, given at I/O pairs. I.e., an example is a list of values defining the values of the input units in a given network. The output of the network is a list of values output by the output units

- **Perceptron Learning Rule**
  In a Perceptron, we define the `update-weight` function in the learning algorithm above by the formula:

$$w_i = w_i + \Delta(w_i)$$

  where

$$\Delta(w_i) = \alpha * (T - O)\ x_i$$

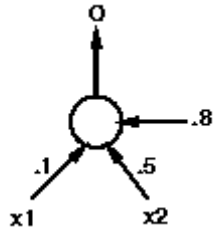
x_i is the input associated with the *i*th input unit, α is a constant between 0 and 1 called the learning rate, T---desired output, O---actual output.

Notes about this update formula:

- Also called the Delta Rule
- "Local" learning rule in that only local information in the network is needed to update a weight
- Performs gradient descent in "weight space" in that if there are *n* weights in the network, this rule will be used to iteratively adjust all of the weights so that at each iteration (training example) the error is decreasing (more correctly, the error is monotonically non-increasing)
- Correct output (T = O) causes no change in a weight
- $x_i = 0$ causes no change in weight
- Does not depend on w_i
- If T=1 and O=0, then increase the weight so that hopefully next time the result will exceed the threshold at the output unit and cause the output O to be 1
- If T=0 and O=1, then decrease the weight so that hopefully next time the result will be below the threshold and cause the output to be 0.

Example: Learning OR in a Perceptron

- Given initial network defined as:



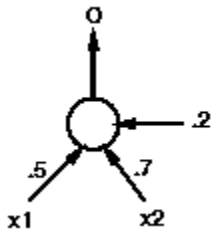
- Let the learning rate parameter be $\alpha = 0.2$

- Let the threshold be specified as a third weight, `w3` with constant input value `x3 = -1`

The result of executing the learning algorithm for 3 epochs:

x1	x2	T	O	Δ(w1)	w1	Δ(w2)	w2	Δ(w3)	w3 (=t)
-	-	-	-	-	.1	-	.5	-	.8
0	0	0	0	0	.1	0	.5	0	.8
0	1	1	0	0	.1	.2	.7	-.2	.6
1	0	1	0	.2	.3	0	.7	-.2	.4
1	1	1	1	0	.3	0	.7	0	.4
0	0	0	0	0	.3	0	.7	0	.4
0	1	1	1	0	.3	0	.7	0	.4
1	0	1	0	.2	.5	0	.7	-.2	.2
1	1	1	1	0	.5	0	.7	0	.2
0	0	0	0	0	.5	0	.7	0	.2
0	1	1	1	0	.5	0	.7	0	.2
1	0	1	1	0	.5	0	.7	0	.2
1	1	1	1	0	.5	0	.7	0	.2

So, the final learned network is:



Linear Separability

- Perceptron output (assuming a single output unit) determined by the separating hyper-plane defined by

  `w1*x1 + w2*x2 + ... + wn*xn = t`

- Example
  - The weights in the initial network for the OR function given above defines a separating line

```
(.1 * x1) + (.5 * x2) - .8 = 0.
```

Rewriting, this is equivalent to the line defined by

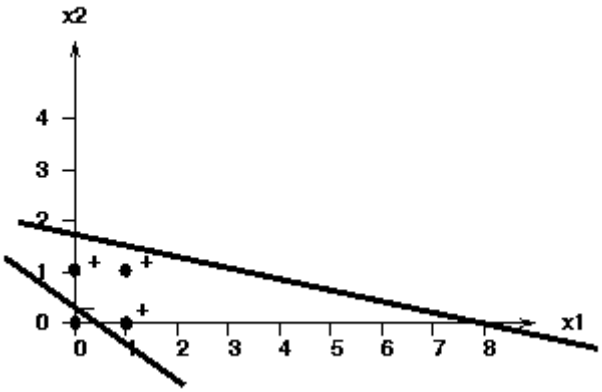
```
x2 = (-.2 * x1) + 1.6,
```

that is a line with slope -.2 and x2-intercept 1.6.

- The weights in the final network for the OR function define a separating line

```
.5 * x1) + (.7 * x2) - .2 = 0, or
x2 = (-.7 * x1) + .3
```

- The initial and final separating lines can be shown graphically in terms of the two-dimensional space of possible inputs as follows.



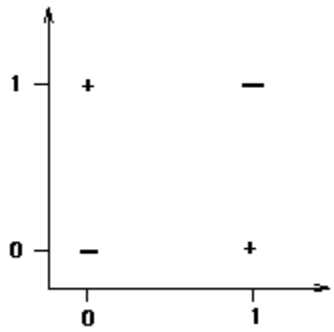
- In general, the goal of learning in a Perceptron is to adjust the separating line by modifying the weights until all of the examples with target value 1 are on one side of the line, and all of the examples with target value 0 are on the other side of the separating line, where the "line" is in general a hyper-plane in an *n*-dimensional space, and *n* is the number of input units.

XOR - A Function that Can Not be Learned by a Perceptron

- The Exclusive OR function can be shown graphically as follows, where + corresponds to an output of 1, and - corresponds to a desired output of 0.

x1	x2	x1 XOR x2
F	F	F

T	F	T
F	T	T
T	T	F

- 

- There does not exist a line that can separate the two classes. Hence, XOR is not a linearly-separable function, and cannot be learned by a Perceptron.

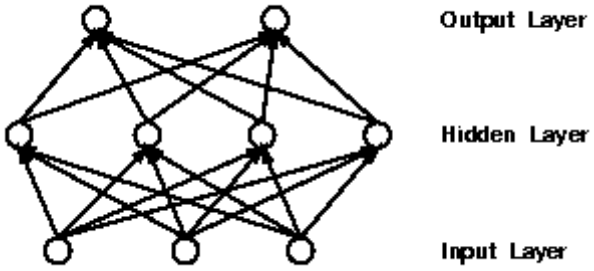
Perceptron Convergence Theorem

- If a set of examples are learnable (i.e., 100% correct classification), the Perceptron Learning Rule will find the necessary weights (Minksy and Papert, 1988)
    - In a finite number of steps
    - Independent of the initial weights
    - The Perceptron Learning Rule does gradient descent search in weight space, so this theorem says that if a solution exists, gradient descent is guaranteed to find an optimal (i.e., 100% correct classification) solution for any 1-layer neural network

4.7.2 Beyond Perceptrons

- Perceptrons are too weak because they can only learn linearly-separable functions
- Define more complex neural networks in order to enhance their functionality
- Multi-layer, feedforward networks generalize 1-layer networks (i.e., Perceptrons) to *n*-layer networks as follows:

- Partition units into n+1 "layers," such that layer 0 contains the input units, layers 1, ..., n-1 are the hidden layers, and layer n contains the output units
- Each unit in layer k, k=0, ..., n-1, is connected to *all* of the units in layer k+1
- Connectivity means bottom-up connections only, with no cycles, hence the name "feedforward" nets
- Programmer defines the number of hidden layers and the number of units in each layer (input, hidden, and output)
- Example of a 2-layer feedforward network:

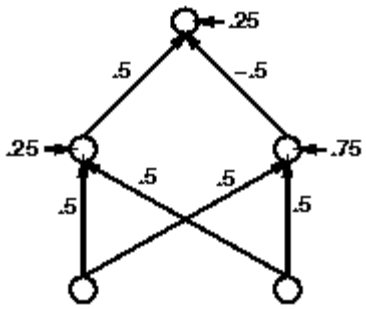


- 2-layer feedforward neural nets (i.e., nets with one hidden layer) with an LTU at each hidden and output layer unit, can compute functions associated with classification regions that are convex regions in the space of possible input values. That is, each unit in the hidden layer acts like a Perceptron, learning a separating line. Together, all of the hidden units' separating lines are combined by the output unit(s) to classify an example by intersecting all of the half-planes defined by the separating lines.
- Recurrent networks are multi-layer networks in which cyclic (i.e., feedback) connections are allowed.

Computing XOR

Using a 2-Layer Feedforward Network

The following network computes XOR. Notice that the left hidden unit effectively computes OR and the right hidden unit computes AND. Then the output unit outputs 1 if the OR output is 1 and the AND output is 0.



Gradient descent search in weight space

Given a network, there are a fixed number of connections with associated weights. Say there are *n* weights, then each configuration of weights that defines an instance of the network is a vector, W, of length *n*. W can be considered to be a point in an *n*-dimensional weight space, where each axis is associated with one of the connections in the network. Given a training set of *m* examples, each network defined by the vector W has an associated error, *E*, indicating the total error on all of the training data. We will define the error *E* as:

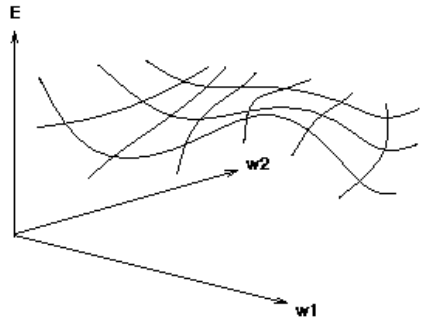
$$E = E1 + E2 + ... + Em$$

where *Ei* is the mean squared error (MSE) of the network on the *i*th training example, computed as follows:

$$Ei = [(T1 - O1)^2 + (T2 - O2)^2 + ... + (Tn - On)]/2$$

where *Tj* is the target value and *Oj* is the network output value for the *i*th example, and there are *n* output units in the network.

Now consider the *n*+1 dimensional space where *n* dimensions are the weights, and the last dimension is the error, *E*. The error is non-negative and defines a surface in this weight space as shown below:



So, the goal is to search for the point in weight space with (global) minimum mean squared error *E*

- To simplify the search, we'll use gradient descent to locally move in weight space, at each iteration, so as to change the weights so that the error decreases the fastest, and halt when we're at a (local) minimum in *E*. The gradient is a vector that indicates the steepest rate of increase in a function, so we'll change the weights in the opposite direction, which decreases *E* the fastest. That is, compute

$$Grad\_E = [dE/dw1, dE/dw2, ..., dE/dwn]$$

and then change the ith weight by

$$\Delta(wi) = - \alpha * dE/dwi$$

- Computing derivatives requires an activation function that is continuous, differentiable, non-decreasing and easily computed. Consequently, we can't use the LTE function. Instead, we'll use the sigmoid function at each unit (except the input units, of course).

Neural networks are being used:

- Investment analysis:

Attempt to predict the movement of stocks currencies etc., from previous data. There, they are replacing earlier simpler linear models.

- Signature analysis:

As a mechanism for comparing signatures made (e.g. in a bank) with those stored. This is one of the first large-scale applications of neural

networks in the USA, and is also one of the first to use a neural network chip.

- Process control:

Most processes cannot be determined as computable algorithms.

- In monitoring:
  - By monitoring vibration levels and sound, early warning of engine problems can be given.
  - British Rail has also been testing a similar application monitoring diesel engines.
- In marketing:

Improve marketing mailshots. One technique is to run a test mailshot, and look at the pattern of returns from this. The idea is to find a predictive mapping from the data known about the clients to how they have responded. This mapping is then used to direct further mailshots.

Some particular Applications

- Speech recognition (Waibel, 1989)
  Converts sound to text
- Character recognition (Le Cun et al., 1989)
- Face Recognition (Mitchell)
- ALVINN (Pomerleau, 1988)
  - Control vehicle steering direction so as to follow road by staying in the middle of its lane.
  - Input color image is preprocessed to obtain a reduced resolution image

    ALVINN learned on the fly as the vehicle traveled, initially "observing" a human driver and later observing its own driving. The current steering position for each input image is saved. Then computes other views of the road by performing various perspective transformations (shift, rotate, and fill in missing pixels) so as to simulate what the vehicle would be seeing if its position and orientation on the road was *not* correct. For each of these synthesized views of the road, a "correct" steering direction is approximated. The real and the synthesized images are then used for training the network.

Initially, a human driver controls the vehicle for about 5 minutes while the network learns weights starting from initial random weights.

4.7.3 Matlab and Neural Network

The Matlab Neural Network Toolbox (NNT) is an all-purpose neural network environment. We will see how to build a custom feed-forward network starting from scratch (i.e. a `blank' neural network object).

Introduction

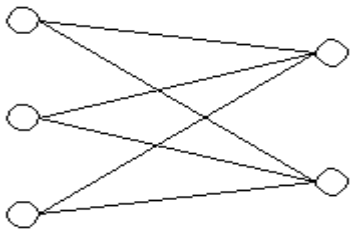
Matlab's Neural Network Toolbox (NNT) is powerful, yet at times completely incomprehensible. All Matlab commands here assume the existence of a NNT network object named `net'. To construct such an object from scratch, type

```
>> net = network;
```

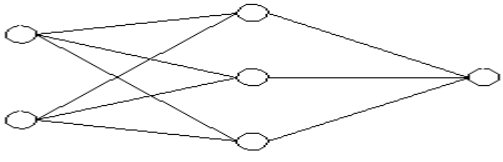
which gives you a `blank' network, i.e. without any properties.

Network Layers

The term `layer' in the neural network sense means, for example, different things to different people. A layer is defined as a layer of neurons, with the exception of the input layer. A one-layer network:



The next one would be a two-layer network:



Each layer has a number of properties, the most important being the transfer functions of the neurons in that layer, and the function that defines the net input of each neuron given its weights and the output of the previous layer.

Constructing Layers

Assume we have an empty network object named `net' in our workspace. Let's start with defining the properties of the input layer. Let's set the input layer to 1:

```
>> net.numInputs = 1;
```

Now we should define the number of neurons in the input layer. This should of course be equal to the dimension of your data set. The appropriate property to set is net.inputs{i}.size, where i is the index of the input layers. So to make a network which has 2 dimensional points as inputs, type:

```
>> net.inputs{1}.size = 2;
```

This defines (for now) the input layer.
The next properties to set are net.numLayers, which not surprisingly sets the total number of layers in the network, and net.layers{i}.size, which sets the number of neurons in the ith layer. To build our example network, we define 2 extra layers (a hidden layer with 3 neurons and an output layer with 1 neuron), using:

```
>> net.numLayers = 2;
>> net.layers{1}.size = 3;
>> net.layers{2}.size = 1;
```

Connecting Layers

Now it's time to define which layers are connected. First, define to which layer the inputs are connected by setting net.inputConnect(i) to 1 for the appropriate layer i (usually the first, so i = 1).

```
>> net.inputConnect(1) = 1; // =1 means true
```

The connections between the rest of the layers are defined a connectivity matrix called net.layerConnect, which can have either 0 or 1 as element entries. If element (i,j) is 1, then the outputs of layer j are connected to the inputs of layer i.

```
>> net.layerConnect(2, 1) = 1;
```

We also have to define which layer is the output layer by setting net.outputConnect(i) to 1 for the appropriate layer i.

```
>> net.outputConnect(2) = 1;
```

Finally, if we have a supervised training set, (training set, which includes the input data for the NN to "see" and the known target data for NN to learn to output. For stock price predictions, for example, the training set and target data would naturally be historical stock prices. A vector of 100 consecutive historical stock prices, for instance, can constitute training data and with the 101st stock price as a target datum.)

We also have to define which layers are connected to the target values. (Usually, this will be the output layer.) This is done by setting net.targetConnect(i) to 1 for the appropriate layer i.

```
>> net.targetConnect(2) = 1;
```

Setting Transfer Functions

Each layer has its own transfer function which is set through the net.layers{i}.transferFcn property. So to make the first layer use sigmoid transfer functions, and the second layer linear transfer functions, use

```
>> net.layers{1}.transferFcn = 'logsig';
>> net.layers{2}.transferFcn = 'purelin';
```

Weights and Biases

Now, define which layers have biases by setting the elements of net.biasConnect to either 0 or 1, where net.biasConnect(i) = 1 means layer i has biases attached to it.
To attach biases to each layer in our example network, we'd use

```
>> net.biasConnect = [ 1 ; 1];
```

Now we should decide on an initialization procedure for the weights and biases.

```
>> net = init(net);
```

to reset all weights and biases according to your choices.
The first thing to do is to set net.initFcn. The value 'initlay' is the way to go. This let's each layer of weights and biases use their own initialization routine to initialise.

```
>> net.initFcn = 'initlay';
```

Exactly which function this is should of course be specified as well. This is done through the property net.layers{i}.initFcn for each layer. The two most practical options here are Nguyen-Widrow initialisation ('initnw', type 'help initnw' for details), or 'initwb', which let's you choose the initialisation for each set of weights and biases separately.
When using 'initnw' you only have to set

```
>> net.layers{i}.initFcn = 'initnw';
```

for each layer i and you're done.
When using 'initwb', you have to specify the initialisation routine for each set of weights and biases separately. The most common option here is 'rands', which sets all weights or biases to a random number between -1 and 1. First, use

```
>> net.layers{i}.initFcn = 'initwb';
```

for each layer i. Next, define the initialisation for the input weights,

```
>> net.inputWeights{1,1}.initFcn = 'rands';
```

and for each set of biases

```
>> net.biases{i}.initFcn = 'rands';
```

and weight matrices

```
>> net.layerWeights{i,j}.initFcn = 'rands';
```

where net.layerWeights{i,j} denotes the weights from layer j to layer i.

Training Functions & Parameters

The difference between train and adapt:

One of the more counterintuitive aspects is the distinction between train and adapt. Both functions are used for training a neural network, and most of the time both can be used for the same network.
What then is the difference between the two? The most important one has to do with incremental training (updating the weights after the presentation of each single training sample) versus batch training (updating the weights after each presenting the complete data set).
When using adapt, both incremental and batch training can be used. Which one is actually used depends on the format of your training set. If it consists of two matrices of input and target vectors, like

```
>> P = [ 0.3 0.2 0.54 0.6 ; 1.2 2.0 1.4 1.5]

P =

    0.3000    0.2000    0.5400    0.6000
    1.2000    2.0000    1.4000    1.5000

>> T = [ 0 1 1 0 ]

T =

     0     1     1     0
```

the network will be updated using batch training. (In this case, we have 4 samples of 2 dimensional input vectors, and 4 corresponding 1D target vectors).
If the training set is given in the form of a cell array,

```
>> P = {[0.3 ; 1.2] [0.2 ; 2.0] [0.54 ; 1.4] [0.6 ; 1.5]}

P =

    [2x1 double]    [2x1 double]    [2x1 double]    [2x1 double]

>> T = { [0] [1] [1] [0] }
```

T =

[0] [1] [1] [0]

then incremental training will be used.
When using train on the other hand, only batch training will be used, regardless of the format of the data (you can use both).
The big plus of train is that it gives you a lot more choice in training functions (gradient descent, gradient descent w/ momentum, Levenberg-Marquardt, etc.) which are implemented very efficiently. So when you don't have a good reason for doing incremental training, train is probably your best choice. (And it usually saves you setting some parameters).

Performance Functions

The two most common options here are the Mean Absolute Error (mae) and the Mean Squared Error (mse). The mae is usually used in networks for classification, while the mse is most commonly seen in function approximation networks.
The performance function is set with the net.performFcn property, for instance:
>> net.performFcn = 'mse';

Train Parameters

If you are going to train your network using train, the last step is defining net.trainFcn, and setting the appropriate parameters in net.trainParam. Which parameters are present depends on your choice for the training function.
So if you for example want to train your network using a Gradient Descent w/ Momentum algorithm, you'd set

>> net.trainFcn = 'traingdm';

and then set the parameters

>> net.trainParam.lr = 0.1;
>> net.trainParam.mc = 0.9;

to the desired values. (In this case, lr is the learning rate, and mc the momentum term.)

Two other useful parameters are net.trainParam.epochs, which is the maximum number of times the complete data set may be used for training, and net.trainParam.show, which is the time between status reports of the training function. For example,

```
>> net.trainParam.epochs = 1000;
>> net.trainParam.show = 100;
```

Adapt Parameters

The same general scheme is also used in setting adapt parameters. First, set net.adaptFcn to the desired adaptation function. We'll use adaptwb (from 'adapt weights and biases'), which allows for a separate update algorithm for each layer. Again, check the Matlab documentation for a complete overview of possible update algorithms.

```
>> net.adaptFcn = 'adaptwb';
```

Next, since we're using adaptwb, we'll have to set the learning function for all weights and biases:

```
>> net.inputWeights{1,1}.learnFcn = 'learnp';
>> net.biases{1}.learnFcn = 'learnp';
```

where in this example we've used learnp, the Perceptron learning rule. (Type 'help learnp', etc.).

Finally, a useful parameter is net.adaptParam.passes, which is the maximum number of times the complete training set may be used for updating the network:

```
>> net.adaptParam.passes = 10;
```

Chapter 5 Stochastic Programming

5.1 Introduction

All of the model formulations that we have encountered so far have assumed that the data for the given problem are known accurately. However, for many actual problems, the problem data cannot be known accurately for a variety of reasons:

- Due to simple measurement error
- The second and more fundamental reason is that some data represent information about the future (e.g., product demand or price for a future time period) and simply cannot be known with certainty.

We will discuss a few ways of taking this uncertainty into account and, specifically, illustrate how stochastic programming can be used to make some optimal decisions.

5.2 Recourse

The fundamental idea behind stochastic linear programming is the concept of recourse. Recourse is the ability to take corrective action after a random event has taken place. For simplicity, we consider the case that the data for the first period are known with certainty and some data for the future periods are stochastic, that is, random.

Example

You are in charge of a local gas company. When you buy gas, you typically deliver some to your customers right away and put the rest in storage. When you sell gas, you take it either from storage or from newly-arrived supplies. Hence, your decision variables are

1) How much gas to purchase and deliver,
2) How much gas to purchase and store, and
3) How much gas to take from storage and deliver to customers.

Your decision will depend on the price of gas both now and in future time periods, the storage cost, the size of your storage facility, and the demand in each period. You will decide these variables for each time period considered in the problem.
More than likely, the future will not be precisely as you have planned; you don't know for sure what the price or demand will be in future periods though you can make good guesses.

For example, if you deliver gas to your customers for heating purposes, the demand for gas and its purchase price will be strongly dependent on the weather. Predicting the weather is rarely an exact science; therefore, not taking this uncertainty into account may invalidate the results from your model. Your ``optimal'' decision for one set of data may not be optimal for the actual situation.

Scenarios

Assumptions: Suppose that we are experiencing a normal winter and that the next winter can be one of three scenarios: normal, cold, or very cold. To formulate this problem as a stochastic linear program, we must first characterize the uncertainty in the model. **The most common method is to formulate scenarios and assign a probability to each scenario.** Each of these scenarios has different data as shown in the following table:

Scenario	Probability	Gas Cost ($)	Demand (units)
Normal	1/3	5.0	100
Cold	1/3	6.0	150
Very Cold	1/3	7.5	180

Both the demand for gas and its cost increase as the weather becomes colder. The storage cost is constant, say, 1 unit of gas is $1/per year. If we solve the linear program for each scenario separately, we arrive at three purchase/storage strategies:

Normal - Normal

Year	Purchase to Use	Purchase to Store	Storage	Cost
1	100	0	0	500
2	100	0	0	500

Total Cost = $1000

Normal - Cold

Year	Purchase to Use	Purchase to Store	Storage	Cost
1	100	0	0	500
2	150	0	0	900

Total Cost = $1400

Normal - Very Cold

Year	Purchase to Use	Purchase to Store	Storage	Cost
1	100	180	180	1580
2	0	0	0	0

Total Cost = $1580

We do not know which of the three scenarios will actually occur next year, but we would like our current purchasing decision to put is in the best position to minimize our

expected cost. Bear in mind that by the time we make our second purchasing decision, we will know which of the three scenarios has actually happened.

5.3 Formulating a Stochastic Linear Program

Stochastic programs seek to minimize the cost of the first-period decision plus the expected cost of the second-period recourse decision.

$$\begin{aligned} \text{Min} \quad & c^T x + E_\omega Q(x,\omega) \\ \text{Subject to} \quad & Ax = b \\ & x >= 0 \end{aligned}$$

where

$$\begin{aligned} & Q(x,\omega) = \text{Min } d(\omega)^T y(\omega) \\ \text{Subject to} \quad & T(\omega)x + W(\omega)y(\omega) = h(\omega) \end{aligned}$$

- x: the first-period decision
- $\mathbf{c^T x}$**:** the first-period direct costs
- ω: random event
- $Q(x,\omega)$: recourse cost
- $E_\omega Q(x,\omega)$: the expected recourse cost
- $y(\omega)$: decision for each random scenario ω

The first linear program minimizes the first-period direct costs, $\mathbf{c^T x}$ plus the expected recourse cost, $Q(x,\omega)$ over all of the possible scenarios while meeting the first-period constraints, $\mathbf{Ax = b}$.
The recourse cost **Q** depends both on **x,** the first-period decision, and on the random event, ω. The second LP describes how to choose $y(\omega)$ (a different decision for each random scenario ω). It minimizes the cost $\mathbf{d^T y}$ subject to some recourse function,

$$\mathbf{Tx + Wy = h.}$$

This constraint can be thought of as requiring some action to correct the system after the random event occurs. In our example, this constraint would require the purchase of enough gas to supplement the original amount on hand in order to meet the demand.
One important thing to notice in stochastic programs is that the first-period decision, **x,** is independent of which second-period scenario actually occurs. **This is called the nonanticipativity property.** The future is uncertain and so today's decision cannot take advantage of knowledge of the future.

5.4 Deterministic Equivalent

The formulation above looks a lot messier than the deterministic LP formulation that we discuss elsewhere. However, we can express this problem in a deterministic for by introducing a different second-period variable **y** for each scenario. This formulation is called the deterministic equivalent.

$$\text{Min} \quad c^T x + \sum_{i=1}^{N} p_i d_i^T y_i$$

$$\text{Subject to} \quad Ax = b$$
$$T_i x + W_i y_i = h_i \ , \ i=1,\ldots, N$$
$$x \geq 0$$
$$y_i \geq 0$$

where **N** is the number of scenarios and p_i is the probability of the scenario's occurrence. For our three-scenario problem, we have N=3.

Notice that the nonanticipativity constraint is met. There is only one first-period decision, **x,** whereas there are **N** second-period decisions, one for each scenario. The first-period decision cannot anticipate one scenario over another and must be feasible for each scenario. That is, **Ax = b** and $\mathbf{T_i\, x + W_i\, y_i = h_i}$ for **i = 1,...,N** Because we solve for all the decisions, **x** and $\mathbf{y_i}$ simultaneously, we are choosing **x** to be (in some sense) optimal over all the scenarios.
Another feature of the deterministic equivalent is worth noting. Because the **T** and **W** matrices are repeated for every scenario in the model, the size of the problem increases linearly with the number of scenarios. Since the structure of the matrices remains the same and because the constraint matrix has a special shape, solution algorithms can take advantage of these properties. Taking uncertainty into account leads to more robust solutions but also requires more computational effort to obtain the solution.
Comparisons with Other Formulations
Because stochastic programs require more data and computation to solve, most people have opted for simpler solution strategies. One method requires the solution of the problem for each scenario. The solutions to these problems are then examined to find where the solutions are similar and where they are different. Based on this information, subjective decisions can be made to decide the best strategy.

5.5 Comparing with Expected-Value Formulation

A more quantifiable approach is to solve the original LP where all the random data have been replaced with their expected values. Hopefully in this approach we will do all right on average. For our example then, we consider the (expected value) problem data to be

Year	**Gas cost ($)**	**Demand**
1	5.0	100
2	6.167 =(5+6+7.5)/3	143.33 =(100+150+180)/3

Cost1 = $(100+143.33)*5 + 143.33 = $1360.00
Cost2 = 100*5 + 143.33*6.167 = $1384

So the optimal result is: $1360

Year	Purchase to Use	Purchase to Store	Storage	Cost
1	100	143.33	143.33	1360
2	0	0	0	0

Let's compute what happens in each scenario if we implement the expected value solution:

Scenario	Recourse Action	Recourse Cost	Total Cost
Normal	Store 43.33 excess @ $1 per unit	43.33	1403.33
Cold	Buy 6.67 units @ $6 per unit	40	1400
Very Cold	Buy 36.67 units @ $7.5 per unit	275	1635

The expected total cost over all scenarios is (1403.33+1400+1635)/3=1479.44

Stochastic Programming Solution

For our three-scenario problem, we have

$$
\begin{array}{lccccccccc}
\text{Min} & c^T x & + & p_1 d_1^T y_1 & + & p_2 d_2^T y_2 & + & p_3 d_3^T y_3 & & \\
\text{s. t.} & Ax & & & & & & & = & b \\
 & T_1 x & + & W_1 y_1 & & & & & = & h_1 \\
 & T_2 x & + & & & W_2 y_2 & & & = & h_2 \\
 & T_3 x & + & & & & & W_3 y_3 & = & h_3 \\
 & x, y_i & \geq & 0 & & & & & &
\end{array}
$$

Forming and solving the stochastic linear program gives the following solution:

Year	Purchase to Use	Purchase to Store	Storage	Cost
1 Normal	100	100	100	1100
2 Normal	0	0	0	0
2 Cold	50	0	0	50*6 = 300
2 Very Cold	80	0	0	80*7.5 = 600

Average Cost = 1100+300/3+600/3= $1400

Similarly we can compute the costs for the stochastic programming solution for each scenario:

Scenario	Recourse Action	Recourse Cost	Total Cost
Normal	None	0	1100
Cold	Buy 50 units @ $6 per unit	300	1400

Very Cold	Buy 80 units @ $7.5 per unit	600	1700

The expected total cost over all scenarios is (1100+1400+1700)/3=1400

The difference in these average costs ($79.44) is the value of the stochastic solution over the expected-value solution.

	Stochastic	Expected Value
Normal→Normal	1000	1100
Normal→Cold	1400	1400
Normal→Very Cold	1580	1700

By solving each scenario alone, one assumes perfect information about the future to obtain a minimum cost. The stochastic solution is minimizing over a number of scenarios and, as a result, sacrifices the minimum cost for each scenario in order to obtain a robust solution over all the scenarios.

Conclusion

Randomness in problem data poses a serious challenge for solving many linear programming problems. The solutions obtained are optimal for the specific problem but may not be optimal for the situation that actually occurs. Being able to take this randomness into account is critical for many problems where the essence of the problem is dealing with the randomness in some optimal way. Stochastic programming enables the modeller to create a solution that is optimal over a set of scenarios.

Chapter 6 Global Optimization

Nonlinear optimization methods have been studied for hundreds of years, and a huge amount of literature discussed this subject in fields such as operations research, numerical analysis, and statistical computing. So far no single method is the best for all nonlinear optimization. The method chosen to solve a problem should depend on the characteristics of the problem. For functions with continuous second derivatives, there are three general types of algorithms: stabilized Newton and Gauss-Newton algorithms, various Levenberg-Marquardt and trust-region algorithms, and various quasi-Newton algorithms. For most practical purposes those methods have been found to be effective. However, all of the above methods search local optima.

A desired method should be the one that can give global minimum and deal with stochastic traffic. There are a variety of approaches to find global optimization. A simple way is to run any of the local optimization methods from numerous random starting points, or one can apply more complicated

methods designed for global optimization such as simulated annealing (SA) or genetic algorithms.

6.1 Simulated annealing

Simulated annealing (SA) is a Monte Carlo approach for minimizing multivariate functions. Its major advantage over other methods is its ability to avoid becoming trapped at local minima. Since we apply this algorithm to optimize parameters of feedback control laws, it is necessary to present the basic conception.

SA was originally proposed by Metropolis in 1953. It is motivated by an analogy to annealing in solids. When a solid is heated past melting point and then is cooled, the structural properties of the solid depend on the rate of cooling. If the temperature of the liquid is lowered slowly enough, large crystals will be formed. The algorithm simulates the cooling process by gradually lowering the temperature of the system until it converges to a steady, frozen state. In 1982, Kirkpatrick et al took the idea of the algorithm and applied it to search for feasible solutions and converge to an optimal solution.

Sketch of the method

Starting from an initial point, taking a step, the algorithm evaluates the cost function. While minimizing a function, it accepts any downhill step and the process iterates from this new point. It also may accept an uphill step with the probability based on the Metropolis criteria. As a result, it can jump out of local optima. With the optimization process proceeding, the algorithm closes to the global optimum. Since there are very few assumptions made regarding the cost function to be optimized, the algorithm is very robust with respect to non-quadratic surfaces.

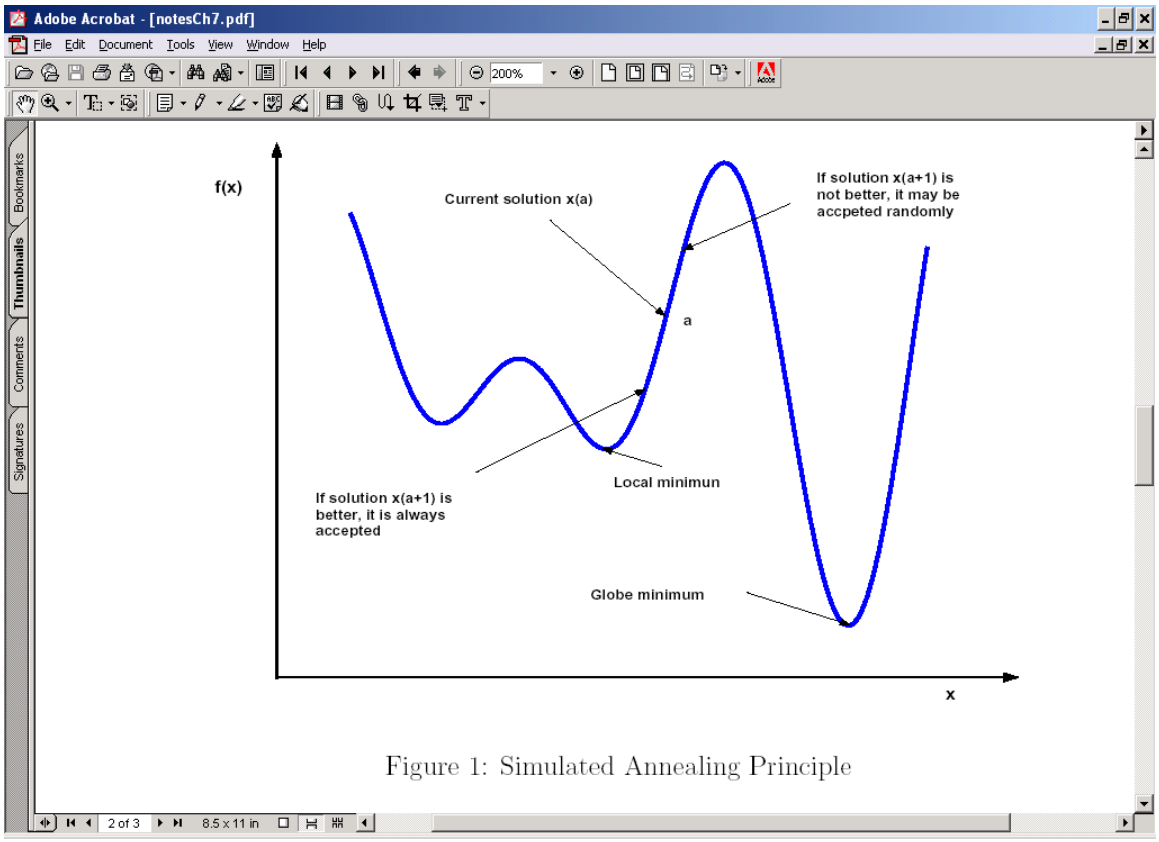


Figure 1: Simulated Annealing Principle

Detailed description

Let atoms or molecule(s) adopt a *state* (configuration or conformation), call it State-i, than has an energy E_i. Boltzmann showed that the number of particles in State-i, N_i, divided by the number of particles in State-j, N_j, is given by the expression

$$\frac{N_i}{N_j} = e^{-(E_i - E_j)/KT}$$

where k is the **Boltzmann's constant** and T is the absolute temperature in Kelvin,

$$k = 1.3806503 \times 10^{-23} \ (m^2 \ kg \ s^{-2} \ K^{-1}).$$

Metropolis Monte Carlo simulation:

If the system is randomly changed from an old state with energy E_o to a new state with energy E_n, the following rules are used to determine if this new state is accepted or rejected. For convenience, we define

Boltzmann ratio of the number of particles in new state (Br) = $e^{(Eo-En)/KT}$

- If $E_n \leq E_o$, the energy of the system dropped and the new state is accepted. It replaces the old state and E_o is set to E_n.
- If $E_n > E_o$, the energy of the system rises and the Boltzmann ratio is compared to a random number, R, between 0.0 and 1.0.
- If $R \leq Br$, the new state is accepted; replacing the old state and E_o.
- If $R > Br$, the new state is rejected and the system stays in the old state.

Metropolis and coworkers proved that the Boltzmann average of a physical property is just the mean value of this property taken over all sampled states (taking the value from the old state again if the new state is rejected), *as long as the simulation is run long enough to properly sample all energetically reasonable states of the system*. If a simulation is able to sample all reasonable states of the system, it is known as *ergodic*, and this has definite ramifications for Simulated Annealing.

Instead of being interested in the Boltzmann average of a system, Kirkpatrick and coworkers were interested in finding the most stable state of a system. They modeled their algorithm after the process of cooling glass, known as *annealing*. This process heats the atoms used to make glass to a very high temperature so that the atoms are relatively free to move around. The temperature is very slowly decreased and the atoms start settling into favorable arrangements. As the temperature continues to decrease, the mobility of the atoms is further reduced and they eventually settle into the most favorable arrangement by the time room temperature is reached. The Metropolis Monte Carlo algorithm is well suited for this simulation since only energetically feasible states will be sampled at any given temperature. The Simulated Annealing algorithm is therefore a Metropolis Monte Carlo simulation that starts at a high temperature. The temperature is slowly reduced so that the search space becomes smaller for the Metropolis simulation, and when the temperature is low enough the system will hopefully have settled into the most favorable state.

Simulated Annealing can also be used to search for the optimum solution of the problems by properly determing the initial (high) and final (low)

effective temperatures which are used in place of kT in the acceptance checking, and deciding what constitutes a Monte Carlo step.
The initial and final effective temperatures for a given problem can be determined from the acceptance probability. In general, if the initial Monte Carlo simulation allows an energy increase of dE_i with a probability of P_i, the initial effective temperature is

$$kT_i = -dE_i/\ln(P_i)$$

A similar expression can be used to determine the final effective temperature.
It is important to realize that Simulated Annealing is strictly a minimization procedure.

An outline of a Simulated Annealing run is as follows.

1. Randomly generate a solution array for the problem. This becomes the old solution and its cost, E_o, is determined. This solution and its cost become the best-to-date values.
2. Set the effective temperature (kT) to the initial value.
3. Run a Metropolis Monte carlo simulation at this temperature by repeating the following steps.
    - Generate a new solution by replacing an object that is used in the old solution with one that is not.
    - Calculate the cost of this new solution, E_n.
    - Determine if this new solution is kept or rejected.
        (i) If $E_n \leq E_o$, the new solution is accepted. It replaces the old solution and E_o. If this cost is less than the best-to-date value, it also becomes the best-to-date solution and cost.
        (ii) If $E_n > E_o$, the Boltzmann acceptance probability is calculated. If this number is greater than a random number between 0.0 and 1.0, the new solution is accepted and it replaces the old one and E_o. If the acceptance probability is less than the random number, the new solution is rejected and the old solution stays the same.
4. Reduce the effective temperature slightly. If the new effective temperature is greater than or equal to the final effective temperature, return to Step 3. Otherwise the run is finished and the best-to-date solution and cost is reported.

How to reduce temperature

The final point to discuss is how the temperature is reduced. This reduction is known as the *cooling schedule*, and though many schedules are possible, only two are generally used in practice. These are known as the *linear* and

proportional cooling schedules. In a linear cooling schedule, a new effective temperature is generated by subtracting a constant from the old one.

$$T_{new} = T_{old} - C_{lin}$$

If a proportional cooling schedule is used, the new effective temperature is generated by multiplying the old value by a constant that is less than 1.0.

$$T_{new} = T_{old} \times C_{prop}$$

These constants are easily determined by choosing the number of temperature changes, N, needed to get from the initial effective temperature, T_i, to the final effective temperature, T_f. These constants then become

$$C_{lin} = [T_i - T_f] / N$$

$$C_{prop} = (T_f / T_i)^{1/N}$$

If the number of Metropolis Monte Carlo steps is large enough to ensure that this simulation is ergodic at each temperature, the results should be independent of the initial solution, seed to the random number generator, initial and final effective temperatures, and cooling schedule. In practice the number of steps has to be held to a computationally reasonable level, and so there can be some dependence of the final result on these parameters.

6.2 Genetic algorithms

Genetic algorithms make use of analogies to biological evolution by allowing mutations and crossing over between candidates for good local optima in the hope to derive even better ones. At each stage, a whole population of configurations are stored. Mutations have a similar effect as random steps in simulated annealing, and the equivalent of lowering of the temperature is a rule for more stringent selection of surviving or mating individuals. The ability to leave regions of attraction to local minimizers is, however, drastically enhanced by crossing over. This is an advantage if, with high probability, the crossing rules produce offspring of similar or even better fitness (objective function value); if not, it is a severe disatvantage. Therefore the efficiency of a genetic algorithm (compared with simulated annealing type methods) depends in a crucial way on the proper selection of crossing rules. The effect of interchanging coordinates is beneficial mainly

when these coordinates have a nearly independent influence on the fitness, whereas if their influence is highly correlated (such as for functions with deep and narrow valleys not parallel to the coordinate axes), genetic algorithms have much more difficulties. Thus, unlike simulated annealing, successful tuning of genetic algorithms requires a considerable amount of insight into the nature of the problem at hand.

Both simulated annealing methods and genetic algorithms are, in their simpler forms, easy to understand and easy to implement, features that invite potential users of optimization methods to experiment with their own versions. The methods often work, if only slowly, and lacking better alternatives, they are very useful tools for applications, where the primary interest is to find (near-)solutions now, even when the reliability is uncertain and only subglobal optima are reached.

6.3 Smoothing methods

Which are based on the intuition that, in nature, macroscopic features are usually an average effect of microscopic details; averaging smoothes out the details in such a way as to reveal the global picture. A huge valley seen from far away has a well-defined and simple shape; only by looking more closely, the many local minima are visible, more and more at smaller and smaller scales. The hope is that by smoothing a rugged potential energy surface, most or all local minima disappear, and the remaining major features of the surface only show a single minimizer. By adding more and more details, the approximations made by the smoothing are undone, and finally one ends up at the global minimizer of the original surface. While it is quite possible for such a method to miss the global minimum (so that full reliability cannot be guaranteed, and is not achieved in the tests reported by their authors), a proper implementation of this idea at least gives very good local minima with a fraction of the function evaluations needed for the blind annealing and genetic methods. A conceptually attractive smoothing technique is the diffusion equation method, where the original objective function V(x) is smeared out by artificial diffusion. The solution V(x,t) of the diffusion equation V_{xx}(x,t)=V_t(x,t) with initial condition V(x,0)=V(x) gets smoother and smoother as t gets larger; for large enough t, it is even

unimodal. Thus V(x,t) can be minimized by local methods when t is sufficiently large, and using the minimizer at a given t as a starting point for a local optimization at a smaller t, a sequence of local minimizers of V(x,t) for successively smaller t is obtained until finally, with t=0, a minimizer of the original potential is reached. It should be possible to increase the reliability of the method by using a limited amount of global search in each optimization stage. The diffusion equation can be solved by convolution with a Gaussian kernel and hence is explicitly available, e.g., when V(x) is an expolynomial, i.e., a linear combination of arbitrary products of powers of components x_i and exponentials exp(c^T x). For other partially separable functions, one has to use numerical integration, making function evaluation more expensive. For functions lacking partial separability, the method cannot be used in practice. Again, there is no theoretical work on conditions that would ensure convergence to the global minimum.

While the above techniques are motivated by nature it is important to remember that processes in nature need not be the most efficient ones; at best it can be assumed to be efficient given the conditions under which they have to operate (namely an uncertain and changing environment that is potentially hazardous to those operating in it). Indeed, much of our present technology has vastly surpassed natural efficiency by unnatural means, and it would be surprising if it were different in global optimization.
Even assuming that nature solves truly global optimization problems (a disputable assumption), simple lower estimates for the number of elementary steps -- roughly corresponding to function evaluations -- available to natural processes to converge are (in chemistry and in biology) in the range of 10^15 or even more. Thus to be successful on the computers of today or the near future, we must find methods that are much faster, exploring the configuration space in a planned, intelligent way, not with a blind combination of chance and necessity. And the challenge is to devise methods that can be analyzed sufficiently well to guarantee reliablility and success.

6.4 Branch and Bound methods

----Mixed Integer Programming

Used to solve global optimization problems with piecewise linear objective and constraints, using special ordered sets. Those with nonlinear constraints can be discretized to obtain approximating piecewise linear constraints.

----Constraint Satisfaction Techniques
----DC-Methods
(DC= difference of convex functions)
----Interval Methods
----Stochastic Methods

Branch and bound methods are most interesting in this respect since they lead to lower bounds on the objective function. They are the only current methods that allow an assessment of the quality of the local minima obtained, and combined with computationally verifiable sufficient conditions for global minima , they may allow one to actually prove global optimality of the best local optimizers obtained, and thus may remove the aura of ambiguity inherent in all `global' calculations of the past. In the worst case, branch and bound methods take an exponential amount of work; but many problems arising in practice are far from worst case. Many of the heuristic techniques used currently for searching the conformational space of global optimization problems can be adapted to or combined with the branch and bound approach to take advantage of the structural insights of specific applications. Branch and bound methods will ultimately allow one not only to calculate the global minimum reliably, but also to find all local minima and saddle points within a certain margin of the global minimum. (Saddle points describe transition states in molecular modeling.) This is essentially the problem of finding all zeros of the nonlinear system V'(x)=0 that satisfy a constraint on the signature of the Hessian and on the value of V. Such problems can already be handled in moderate dimensions by branch and bound methods, combined with techniques from interval analysis. It should be possible to combine these techniques with underestimation techniques provided by dc-methods.

Chapter 7 Dynamic Programming

Problem Statement

> Dynamic Programming is an approach developed to solve sequential, or multi-stage, decision problems; hence, the name "dynamic" programming. But, as we shall see, this approach is equally applicable for decision problems where sequential property is induced solely for computational convenience.

Unlike other branches of mathematical programming, one cannot talk about an algorithm that can solve all dynamic programming problems. For example, George Dantzig's Simplex Method can solve all linear programming problems. Dynamic programming, like branch and bound approach, is a way of decomposing certain hard to solve problems into equivalent formats that are more amenable to solution. Basically, what dynamic programming approach does is that it solves a multi-variable problem by solving a series of single variable problems. This is achieved by tandem projection onto the space of each of the variables. In other words, we project first onto a subset of the variables, then onto a subset of these, and so on.

The essence of dynamic programming is Richard Bellman's **Principle of Optimality**. This principle, even without rigorously defining the terms, is intuitive:

> An optimal policy has the property that whatever the initial state and the initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

This is a self-evident principle in the sense that a proof by contradiction is immediate. Rutherford Aris restates the principle in more colloquial terms:

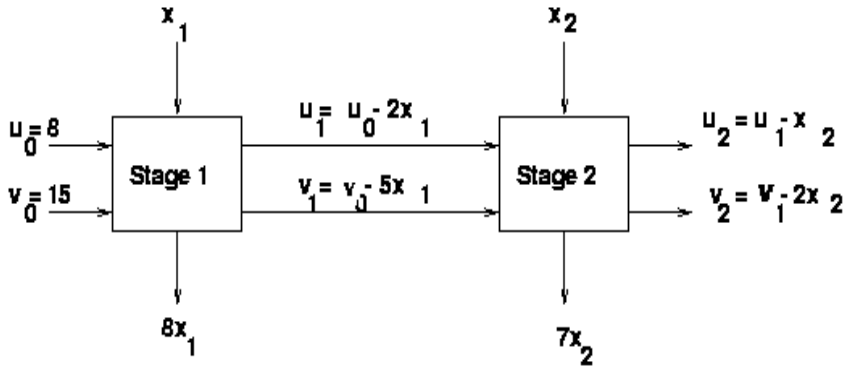
> If you don't do the best with what you have happened to have got, you will never do the best with what you should have had.

We use the following example to illustrate *deterministic* dynamic programming approach.

Consider the following integer programming problem taken from Taha:

$$\begin{aligned} &\text{MAX } 8x_1 + 7x_2, \\ &\text{SUBJECT TO:} \\ &2x_1 + x_2 <= 8, \\ &5x_1 + 2x_2 <= 15, \\ &x_1, x_2 >= 0, \text{ and integer.} \end{aligned}$$

If we interpret x_1 as the value of activity 1 and x_2 as the value of the activity 2, then we can define each stage as the process of deciding on the level of each activity. One can interpret the right hand side values of the constraints as the resource availabilities. At each stage we decide on the level of an activity, thus consuming certain amount from each of the resources. This decision is constrained by the available amounts of each resource at the beginning of the stage, and unused amounts of the resources are left to be used by the activities in the "downstream" stages. Thus, knowing "available amounts of each of the resources at the beginning of a stage" is sufficient to make optimal decisions at the current and all of the remaining stages of the process. Since we have two constraints in the above problem, we need to define state of the process by a vector of two elements. Let $u_{(t-1)}$ be the available amount of first resource and $v_{(t-1)}$ be the available amount of second resource just before we make the decision x_t. Schematically, this multi-stage decision process can be shown as:



There is no inherent sequential character in this integer programming problem: I could have equally defined the first stage as the stage at which decision is made concerning the value of x_2, and the second stage for x_1.

Now we use forward recursion to solve the problem. We start with the first stage:

Initial state vector at the beginning of Stage 1 is (8, 15). Observing this, we need to assign a value for x_1. The only constraint we have is that the state vector at the end of this stage is to be non-negative, i.e. $u_1 = 8 - 2x_1 >= 0$ and $v_1 = 15 - 5x_1 >= 0$. Thus, the possible values x_1 can assume are 0, 1, 2, and 3. These values and the resulting return (i.e. the contribution to the objective function value, i.e. $8x_1$) from this stage (and all the "previous" stages; but this being the first stage, there are no previous stages) are summarized in the following table:

(u_0, v_0)	x_1	(u_1, v_1)	**RETURN**
(8, 15)	0	(8, 15)	0
	1	(6, 10)	8
	2	(4, 5)	16
	3	(2, 0)	24

Now we proceed forward to Stage 2.

For each value of the state vector at the beginning of Stage 2, the maximum value that can be assigned to x_2, such that the ending state vector to be non-negative, is given in the following table:

(u_1, v_1)	x_2	**RETURN**
(8, 15)	7	49
(6, 10)	5	43
(4, 5)	2	30
(2, 0)	0	24

The optimal solution is $x^* = (0, 7)$ with an objective function value of 49.

Here we were working on deterministic problems; given the initial stage and the decision we make at a certain stage, the resulting state of the systems were known with certainty. This type of problems is amenable to deterministic dynamic programming. When the resulting state is not known with certainty but its

realization is governed by a probability distribution, we have a stochastic dynamic programming problem.

Appendix A: User Requirement Analysis

User requirements analysis provides precise descriptions of the content, functionality and quality demanded by prospective users. For the identification of user needs the user perspective must be assumed and result in:

A.1 Functional requirements

Specification of functional requirements. The goals users want to reach and the tasks they intend to perform with the new product or service must be determined, including information needed, modes of access, transactions, modifications to this information.

The distinction between tasks and activities is crucial. Activities describe user procedures (also called user action), i.e. command sequences for performing tasks. Tasks describe the goals of the user with as little reference as possible to the detailed technical realisation of the information product. It is often misleading to transfer action sequences using existing paper based information products to electronic information products. Electronic information products may provide new methods for browsing, selecting and visualising data which induce very different activities of the user.

Understanding the tasks involves abstraction of why the user performs certain activities, what his constraints and preferences are, and how the user would make trade-offs between different products.

A.2 Non-functional requirements

Specification of non-functional requirements includes the description of user characteristics such as prior knowledge and experiences, special needs of elderly and handicapped persons, subjective preferences, and the description of the environment, in which the product or service will be used. For electronic information products and services new business models, legal issues, intellectual property rights, security and privacy requirements be an issue. Further non-functional requirements must be derived from cost constraints.

A.3 Methods

Informal methods such as observation, interview, document analysis, focus group analysis, checklists or questionnaires can be used for the elicitation of user requirements. Different requirements analysis methods can be applied in parallel to complement each others in order to yield more effective results.

Appendix B. Data Mining and Modeling

Data mining, *the extraction of hidden predictive information from large databases*, is a powerful new technology with great potential to help companies focus on the most important information in their data warehouses.

B.1 The Scope of Data Mining

Given databases of sufficient size and quality, data mining technology can generate new business opportunities by providing these capabilities:

- **Automated prediction of trends and behaviors**. Questions that traditionally required extensive hands-on analysis can now be answered directly from the data — quickly. A typical example of a predictive problem is targeted marketing. Data mining uses data on past promotional mailings to identify the targets most likely to maximize return on investment in future mailings.

- **Automated discovery of previously unknown patterns**. Data mining tools sweep through databases and identify previously hidden patterns in one step. An example of pattern discovery is the analysis of retail sales data to identify seemingly unrelated products that are often purchased together.

B.2 Techniques

The most commonly used techniques in data mining are:

- **Artificial neural networks**: Non-linear predictive models that learn through training and resemble biological neural networks in structure.

- **Decision trees**: Tree-shaped structures that represent sets of decisions. These decisions generate rules for the classification of a dataset. Specific decision tree methods include Classification and Regression Trees (CART) and Chi Square Automatic Interaction Detection (CHAID).

- **Genetic algorithms**: Optimization techniques that use processes such as genetic combination, mutation, and natural selection in a design based on the concepts of evolution.

- **Nearest neighbor method**: A technique that classifies each record in a dataset based on a combination of the classes of the k record(s) most similar to it in a historical dataset (where k ³ 1). Sometimes called the k-nearest neighbor technique.

- **Rule induction**: The extraction of useful if-then rules from data based on statistical significance.

Lab

Visual Studio.NET + C#

Network

Fourier

References

J.M.Thizy, Lecture Notes, Sys5130.
http://www.ece.northwestern.edu/OTC/
http://www-fp.mcs.anl.gov/otc/Guide/OptWeb/index.html
http://www.cs.sandia.gov/opt/survey/
http://solon.cma.univie.ac.at/~neum/glopt/my_view.html
http://benli.bcc.bilkent.edu.tr/~omer/research/dynprog.html
http://www.cs.cornell.edu/courses/cs612/ 2001sp/lectures/lecture06.pdf
http://mat.gsia.cmu.edu/orclass/integer/integer.html

http://www.netsoc.tcd.ie/~notoole/CollegeFiles/
Management%20Science/Networks1.ppt
Michael A. Trick, Network Optimization (1996)
L. Smith, An Introduction to Neural Networks
(http://www.cs.stir.ac.uk/~lss/NNIntro/InvSlides.html)
Portegie, Introduction to the Matlab Neural Network Toolbox 3.0
http://www.cs.wisc.edu/~dyer/cs540/notes/nn.html
http://gear.kku.ac.th/~nawapak/NeuralNet
http://www.cs.oberlin.edu/classes/dragn/lectures/7apr3.html (spanning tee)
http://solon.cma.univie.ac.at/~neum/glopt/my_view.html
http://mat.gsia.cmu.edu/orclass/integer/integer.html
http://www.netsoc.tcd.ie/~notoole/CollegeFiles/
Management%20Science/Networks1.ppt
Michael A. Trick, Network Optimization (1996)
L. Smith, An Introduction to Neural Networks
(http://www.cs.stir.ac.uk/~lss/NNIntro/InvSlides.html)
Portegie, Introduction to the Matlab Neural Network Toolbox 3.0
Ivan Galkin, U. MASS Lowell, Crash Introduction to Artificial Neural Networks,
http://ulcar.uml.edu/iag/CS/Intro-to-ANN.html
S. Kirkpatrick, C.D. Gelatt and M.P. Vecchi, *Science* 220 (1983) 671-680.
N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth. A.H. Teller and E. Teller, "Equation of State Calculations by Fast Computing Machines," *J. Chem. Phys.* 21 (1953) 1087-1092.
http://members.aol.com/btluke/featur05.htm
http://csep1.phy.ornl.gov/CSEP/MO/NODE28.html
http://www.maths.abdn.ac.uk/~igc/tch/mx3503/notes/node116.html
http://www.ie.bilkent.edu.tr/~omer/research/dynprog.html
http://www.thearling.com/text/dmwhite/dmwhite.htm
http://www.vnet5.org/pub/approach/ura.html