Discrete Optimization

Optimization problems can be divided naturally into two categories:

- those with continuous variables
- those with discrete variables (often called combinatorial optimization)

For continuous problems you have a function $f(x_1, x_2, \ldots, x_n)$ which you want to maximize or minimize. The function may be differentiable or not. There may be constraints or not.

For discrete optimization problems we are looking for the maximum or minimum from a finite set which often consists of integers; in fact, in some cases the solution can only take on the value zero or one.

The algorithms for solving these two types of problems are typically vastly different.

Some Standard Techniques for Solving Discrete Optimization Problems

- Linear Programming
- Quadratic Programming
- Nonlinear Programming
- Dynamic Programming
- Simulated Annealing
- Branch and Bound
- Genetic Algorithms
- etc.

We will concentrate on problems where the function to maximize is linear in its unknowns and any constraints are also linear.

Some Examples of Discrete Optimization Problems

Knapsack Problem Suppose a hitchhiker has to fill up his knapsack by selecting from among various possible objects those which will give him maximum comfort. We have n objects each of which has a measure of some quantity such as comfort or value assigned, p_i and a weight w_i. Clearly we want to choose objects such that the sum of their weights are less than the given capacity but which maximizes the comfort. If we denote variables x_i, $i = 1, n$ as

$$x_i = \begin{cases} 1 & \text{if object } i \text{ is selected} \\ 0 & \text{otherwise} \end{cases}$$

then we want to maximize

$$p_1x_1 + p_2x_2 + \cdots + p_nx_n$$

subject to the constraint

$$w_1x_1 + w_2x_2 + \cdots + w_nx_n \leq W$$

where W is the maximum carrying weight of the knapsack.

There are many applications of the Knapsack Problem. For example, suppose we only have an amount W of file storage to store the results of n calculations. Each calculation requires an amount of space w_i and $\sum_{i=1}^{n} w_i > W$. Each calculation has a cost p_i to recompute the results. We want to choose the calculations to store so that we minimize the amount of time to recompute the results, or equivalently maximize the amount of compute time for the stored files.

Traveling Salesman Problem A company is sending one of its salesmen to clients in certain major cities. The distance between the cities is given. We want to find the path starting and ending in a particular location which minimizes the distance traveled while visiting each city only once.

Scheduling Problem Suppose we have a schedule of flights for a particular aircraft type and we want to design weekly schedules for the crews. Each day a crew must be assigned a duty period consisting of a set of one or more linking flights satisfying numerous constraints such as a limit on the total flying time, minimum rest requirements between flights, etc. Then the daily schedules must be put together to satisfy further constraints such as overnight rests, returning to starting point, etc. The objective is to minimize the amount paid to crews

which is a function of flying time, length of duty periods, guaranteed number of hours, etc.

Assignment Problem Suppose there are n people available to do n jobs. For each person there is a cost associated to do a particular job. Assign each person to a job in such a way that the total cost is minimized.

Linear Programming

For a discrete linear optimization problem we are typically given a finite set S and a vector $\vec{c}$ where we want to find $\vec{x}$ to

maximize (or minimize) $\quad c_1x_1 + c_2x_2 + \cdots + c_nx_n$

subject to some linear constraints

$$a_{11}x_1 + a_{12}x_2 + \cdots a_{1n}x_n \leq (\geq, =)b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots a_{2n}x_n \leq (\geq, =)b_2$$

$$\vdots$$

$$a_{m1}x_1 + a_{m2}x_2 + \cdots a_{mn}x_n \leq (\geq, =)b_m$$

To see the types of problem we can solve via linear programming, we will first look at some examples.

I. The Diet Problem Suppose a "starving" graduate student makes a list of his favorite foods and wants to determine a diet which would give him the necessary energy, protein and calcium per day while minimizing his costs. His favorite foods are:

Food	Serving size	energy	protein	calcium	price per serving
oatmeal	28 g	110	4	2	$0.75
chicken	100 g	205	32	12	$ 2.40
eggs	2 large	160	13	54	$0.50
whole milk	8 oz	160	8	285	$0.50
cherry pie	170 g	420	4	22	$2.50
pork & beans	260 g	260	14	80	$0.25

His minimum daily requirements are energy - 2000 kcal, protein - 55 g, calcium - 800 mg. The problem is that the cheapest way to eat would be to consume 10 servings of pork & beans but not many people can maintain an exclusive diet of pork & beans for too long. So the student realizes that he has to put some constraints on the problem because he isn't willing to eat 10 servings of one item in a day. Here are the maximum number of servings of each he is willing to eat.

Oatmeal ≤ 4 servings

Chicken ≤ 3 servings

Eggs≤ 2 servings

Whole milk ≤ 8 servings

Cherry pie ≤ 2 servings

Pork & beans ≤ 2 servings

Even with only 6 foods and the given constraints on the maximum number of servings per day, there are a lot of combinations that are possible. Let's formulate the problem mathematically.

Let the number of servings of each food be designated by $x_1, x_2, \ldots, x_6$ in the order given in the table. Then our constraints on the number of servings of each are

$$0 \leq x_1 \leq 4, \quad 0 \leq x_2 \leq 3, \quad \text{etc.}$$

Our requirement that the servings yield at least 2000 calories gives

$$110x_1 + 205x_2 + 160x_3 + 160x_4 + 420x_5 + 260x_6 \geq 2000;$$

our requirement for the daily protein is

$$4x_1 + 32x_2 + 13x_3 + 8x_4 + 4x_5 + 14x_6 \geq 55;$$

and the requirement for calcium is

$$2x_1 + 12x_2 + 54x_3 + 285x_4 + 22x_5 + 80x_6 \geq 800$$

Finally our last condition is that we want to minimize the cost of our daily food so our complete problem is given as

$$\text{minimize}\Big[.75x_1 + 2.4x_2 + .5x_3 + .5x_4 + 2.5x_5 + .25x_6\Big]$$

subject to the constraints

$$110x_1 + 205x_2 + 160x_3 + 160x_4 + 420x_5 + 260x_6 \geq 2000$$

$$4x_1 + 32x_2 + 13x_3 + 8x_4 + 4x_5 + 14x_6 \geq 55$$

$$2x_1 + 12x_2 + 54x_3 + 285x_4 + 22x_5 + 80x_6 \geq 800$$

$$0 \leq x_1 \leq 4, \quad 0 \leq x_2 \leq 3, \quad 0 \leq x_3 \leq 2,$$

$$0 \leq x_4 \leq 8, \quad 0 \leq x_5 \leq 2, \quad 0 \leq x_6 \leq 2$$

So we have a linear function of our integer unknowns $x_1, x_2, \ldots, x_6$ which we want to minimize and we have 9 linear inequality constraints.

This is an example of a Linear Programming problem.

II. The Transportation Problem A manufacturer has two plants, one located in Salt Lake City and another in Denver. There are three distributing warehouses, one in Los Angeles, one in Chicago and the third in New York City. The Salt Lake City plant can supply 120 tons of the product per week whereas Denver can supply 140 tons per week. The LA warehouse needs 100 tons weekly to meet demands, the Chicago warehouse needs 60 tons weekly and the NYC warehouse needs 80 tons weekly. There are shipping costs associated with transporting the product from the plant to the distributing warehouses; this is indicated below in cost per ton of the product

	Los Angeles	Chicago	NYC
from Salt Lake City	\$5	\$7	\$9
from Denver	\$6	\$7	\$10

To formulate this mathematically we let P_1 denote the Salt Lake City plant and P_2 the Denver plant; let W_i, $i = 1, 3$ denote the warehouses in LA, Chicago, and NYC, respectively. Let x_{ij} denote the tonnage shipped from plant P_i to warehouse W_j and c_{ij} the cost per ton in shipping from plant P_i to warehouse W_j.

For the Salt Lake City plant

$$x_{11} + x_{12} + x_{13} \leq 120$$

while for the Denver plant

$$x_{21} + x_{22} + x_{23} \leq 140$$

The demand for the LA warehouse requires

$$x_{11} + x_{21} \geq 100$$

and for Chicago we have

$$x_{12} + x_{22} \geq 60$$

and for NYC we have

$$x_{13} + x_{23} \geq 80$$

Our goal is to minimize the cost of shipping the product while meeting the demands of each warehouse. We summarize the problem as

$$\text{minimize}\Big[\sum_{i=1}^{2}\sum_{j=1}^{3} c_{ij}x_{ij}\Big]$$

where $c_{11} = 5$, $c_{12} = 7$, $c_{13} = 9$, $c_{21} = 6$, $c_{22} = 7$, $c_{23} = 10$

subject to the constraints

$$x_{11} + x_{12} + x_{13} \leq 120 \quad x_{21} + x_{22} + x_{23} \leq 140 \quad \text{(supplies)}$$

$$x_{11} + x_{21} \geq 100 \quad x_{12} + x_{22} \geq 60 \quad x_{13} + x_{23} \geq 80 \quad \text{(demands)}$$

$$x_{ij} \geq 0, i = 1, 2; j = 1, 2, 3$$

Once again we have to minimize a linear function of our integer variables subject to a set of linear constraints.

III. Blending Problem A manufacturer of artificial sweetener blends 14 kg saccharin and 18 kg dextrose to prepare two new products: SWEET and LO-SUGAR. Each kg of SWEET contains 0.4 kg dextrose and 0.2 kg saccharin while each kg of LO-SUGAR contains 0.3 kg of dextrose and 0.4 kg of saccharin. If the profit on each kg of SWEET is 20 cents and on each kg of LO-SUGAR is 30 cents, how many kilograms of each product should be made to maximize the profit.

Let x_1 denote the number of kg of SWEET and x_2 the number of LO-SUGAR produced. Our problem can be posed mathematically as

$$\text{maximize}\Big[.2x_1 + .3x_2\Big]$$

$$0.4x_1 + 0.3x_2 \leq 18 \quad \text{(dextrose supply constraint)}$$

$$0.2x_1 + 0.4x_2 \leq 14 \quad \text{(saccharin supply constraint)}$$

$$x_i \geq 0, i = 1, 2$$

In general, a Linear Programming (LP) problem is to maximize (or minimize) a linear function of our unknowns subject to linear equality and/or inequality constraints.

General Linear Programming Problem: Find values $x_1, x_2, \ldots, x_n$ which

$$\text{maximize or minimize} \quad z = c_1x_1 + c_2x_2 + \cdots + c_nx_n$$

subject to the conditions

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq (\geq)(=)b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq (\geq)(=)b_2$$

$$\vdots$$

$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq (\geq)(=)b_m$$

The function z is called the objective function and the restrictions/conditions are called constraints.

To standardize our discussion, we will say that a Linear Programming problem is in standard form if we are $maximizing\ z$, our variables $x_i \geq 0$ and all other constraints are $\leq$. Note that different authors may choose the standard form as minimizing z.

If our problem is to minimize a function, we simply maximize the negative of it; that is, if we seek $\vec{x}$ to

$$\text{minimize} \quad z = c_1x_1 + c_2x_2 + \cdots + c_nx_n$$

then we change it to

$$\text{maximize} \quad z = -(c_1x_1 + c_2x_2 + \cdots + c_nx_n)$$

If we have a constraint that is $\geq$ we simply multiply the inequality by -1 to change the sign of the inequality.

Standard Form of Linear Programming Problem: Find values $x_1, x_2, \ldots, x_n$ which

$$\text{maximize} \quad z = c_1x_1 + c_2x_2 + \cdots + c_nx_n$$

subject to the conditions

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2$$

$$\vdots$$

$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m$$

$$x_j \geq 0, \quad j = 1, 2, \ldots, n$$

We can also rewrite our standard form in terms of matrices and vectors. We have the following.

Matrix Form of Linear Programming Problem: Given $\vec{c}$, find a vector $\vec{x} = (x_1, x_2, \ldots, x_n)^T$ which

$$\text{maximizes} \quad z = \vec{c}^T \vec{x}$$

subject to the constraints

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \leq \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

$$x_j \geq 0, \quad j = 1, 2, \ldots, n$$

Any vector $\vec{x}$ satisfying our constraints is called a feasible solution to our LP program.

The specific vector $\vec{x}$ which is a feasible solution and which maximizes the ob-

jective function z is called an optimal solution.

Example Consider the LP problem of finding $\vec{x} \in \mathbb{R}^2$ such that $\vec{x}$ maximizes

$$z = \begin{pmatrix} 120 & 100 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

subject to

$$\begin{pmatrix} 2 & 2 \\ 5 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq \begin{pmatrix} 8 \\ 15 \end{pmatrix}$$

$$\begin{pmatrix} x \\ y \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Determine if $(1, 2)^T$ and $(2, 2)^T$ are feasible solutions.

All we need to do is to verify that they satisfy the constraints. Clearly each component is non-negative and

$$\begin{pmatrix} 2 & 2 \\ 5 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 6 \\ 11 \end{pmatrix} \leq \begin{pmatrix} 8 \\ 15 \end{pmatrix}$$

so $(1,2)^T$ is a feasible solution. Also

$$\begin{pmatrix} 2 & 2 \\ 5 & 3 \end{pmatrix} \begin{pmatrix} 2 \\ 2 \end{pmatrix} = \begin{pmatrix} 8 \\ 16 \end{pmatrix} \not\leq \begin{pmatrix} 8 \\ 15 \end{pmatrix}$$

So $(2,2)^T$ is not a feasible solution.

For a linear programming problem with two unknowns we can solve it by geometrical reasoning. Before we look at a method to solve a general Linear Programming problem we look at this special case so that we can gain some intuition about the problems.

Linear Programming Problem with Two Unknowns

Suppose we want to maximize a function of two unknowns subject to a set of constraints. Then this problem can be solved by geometrical reasoning. To do this, we graph the constraints (as equalities) and see that the constraints define a polyhedron and all feasible solutions lie inside the polyhedron.

Example Sketch the region of feasible solutions for the LP problem in standard form:

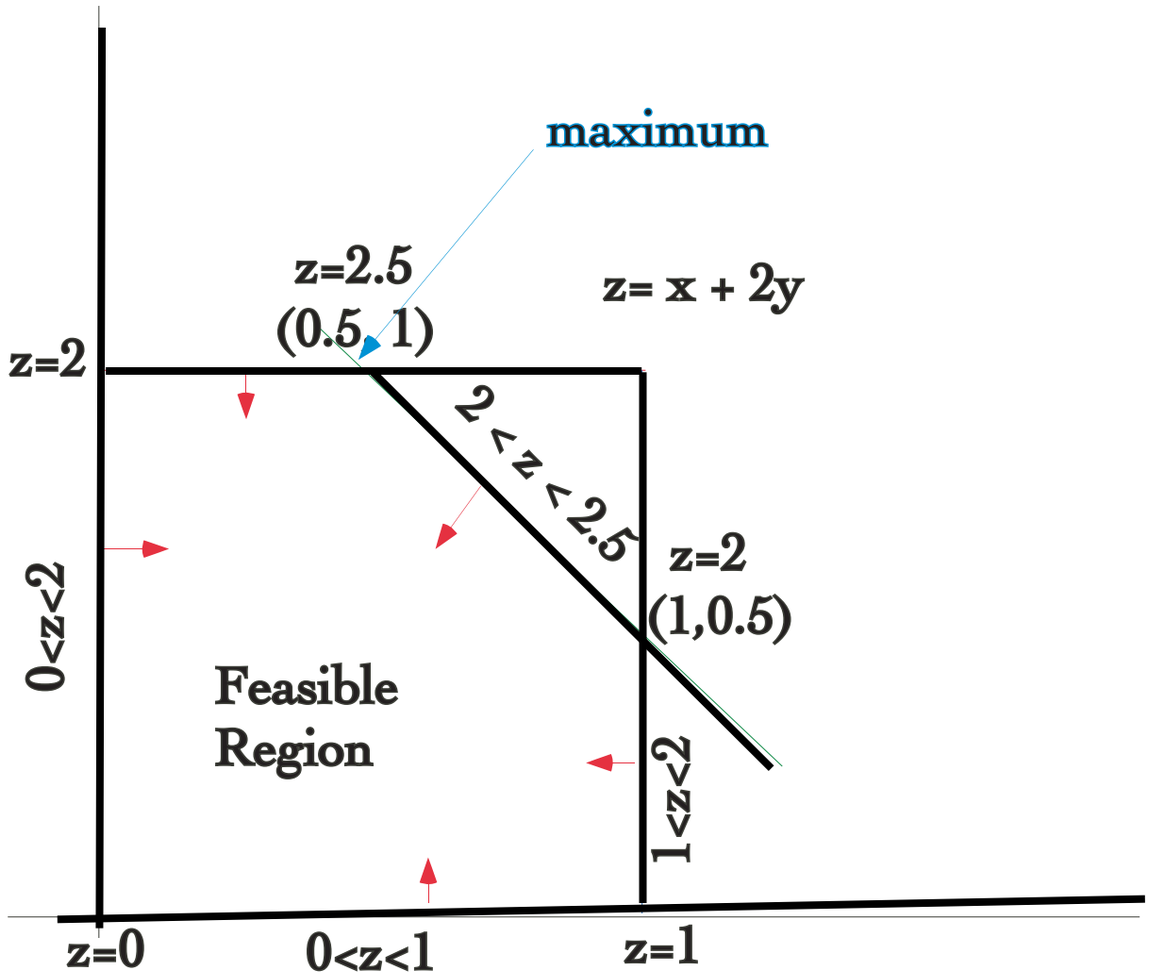
$$\max(x + 2y)$$

subject to

$$x \leq 1, y \leq 1, \qquad x + y \leq 1.5, \qquad x \geq 0, \qquad y \geq 0$$

and use graphical reasoning to find the optimal solution.

The first thing we do is to plot the lines $x = 1$, $y = 1$, $x + y = 1.5$ and $x = 0$, $y = 0$.



maximize z subject to constraints

x>=0, y>=0, x<= 1, y<=1, x+y <=1.5

The region containing all feasible solutions is the polyhedron bounded by our lines. Here z is a linear function of x and y; its maximum must occur on the

boundary of the polyhedron.

Along $y = 0$, z increases from 0 to 1 at (1,0).

Along $x = 1$, z increases from 1 to 2 at (1,0.5).

Along $x = 0$, z increases from 0 at the origin to 2 at (0,1).

Along $y = 1$, z increases from 2 at (0,1) to 2.5 at (0.5,1).

Along the line $x + y = 1.5$, z increases from 2 at (1,.5) to 2.5 at (.5,1).

We see that the maximum occurs at the point $(0.5, 1)$ which satisfies all constraints.

Example Consider the LP problem

$$\text{maximize} \qquad z = 2x + 3y$$

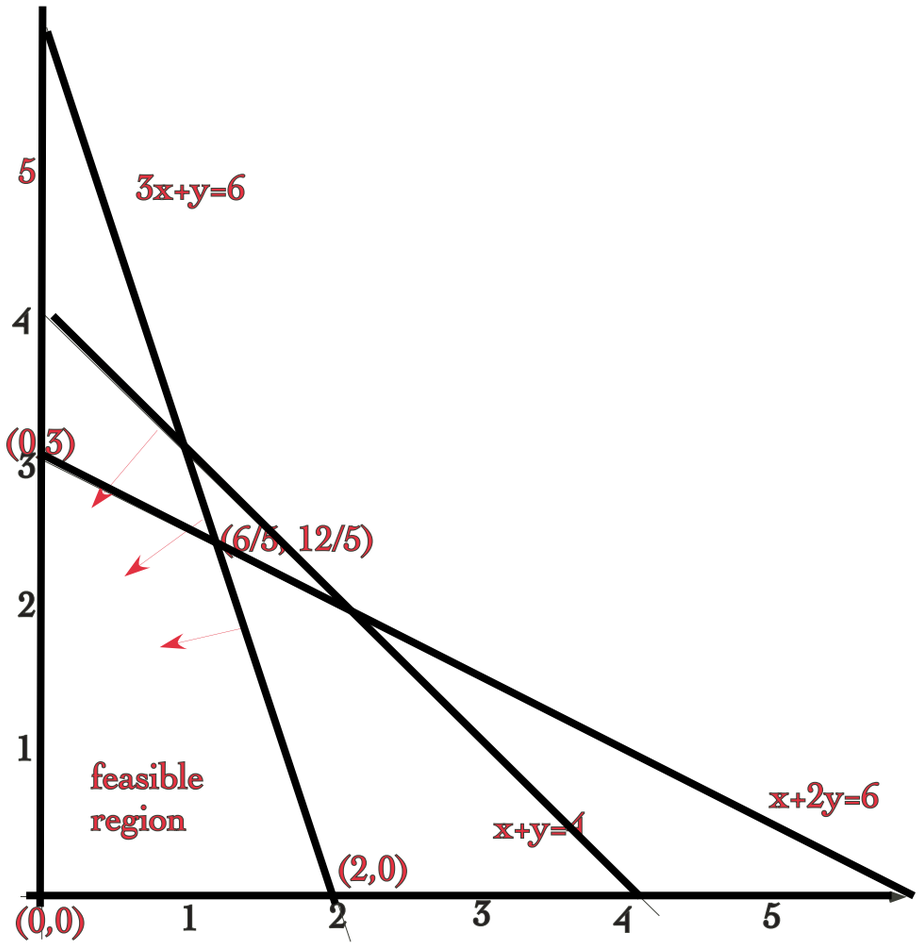
subject to the constraints

$$3x + y \leq 6; \quad x + y \leq 4, \quad x + 2y \leq 6$$

$$x, y \geq 0$$

We plot the three equations and indicate the polyhedron containing the feasible solutions.

Graphically we see that the vertices of the polyhedron are (0,0), (2,0), (1.2,2.4), and (0,3).



Vertices: (0,0), (2,0), (0,3) (6/5,12/5)

If we evaluate z along the edge of the polyhedron $y = 0$ we see that at (0,0) $z = 0$; at (2,0) $z = 4$ so z increases from 0 to 4 along this edge.

Along $x = 0$, we see that z increases from 0 at the origin to $z = 9$ at (0,3).

Along the line $3x + y = 6$, z increases from 4 at (2,0) to 9.6 at (1.2,2.4).

Along the line $x + 2y = 6$, z increases from 9 at (0,3) to 9.6 at (1.2,2.4).

Thus the optimal solution is at (1.2, 2.4) with a value $z = 9.6$.

Example Sketch the region of feasible solutions for the LP problem in standard form:

$$\text{maximize} \quad (2x + 5y)$$

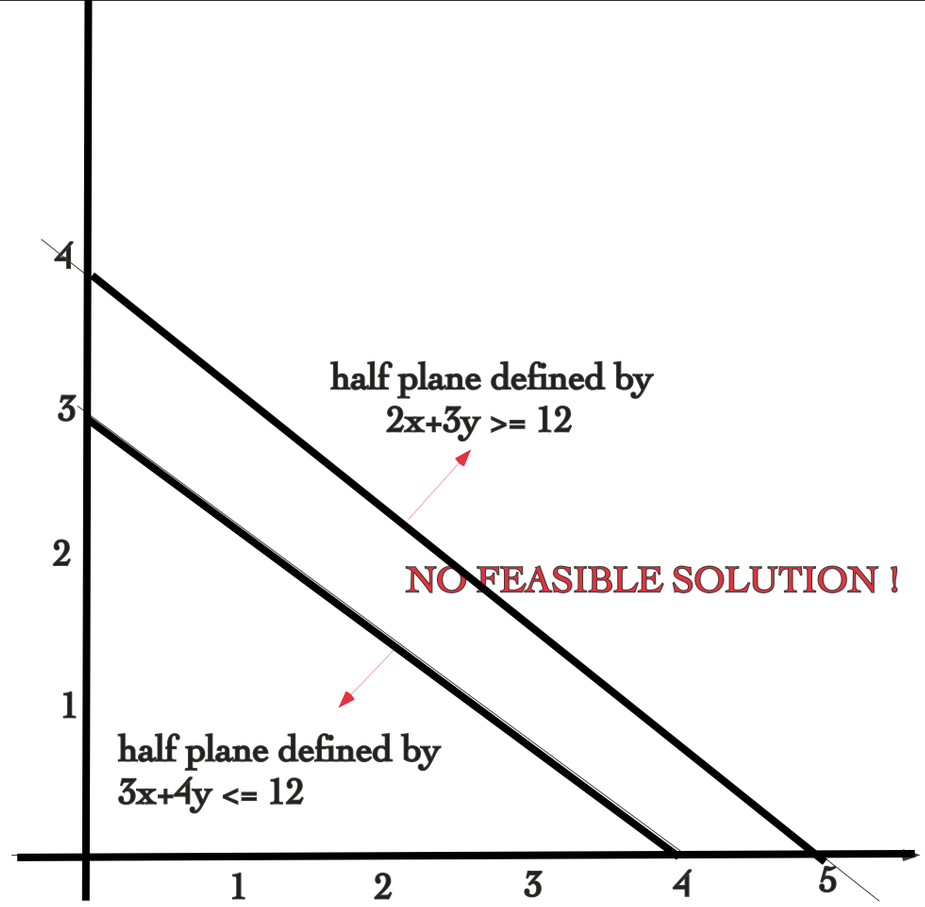
$$\text{subject to} \quad -2x - 3y \leq -12, \quad 3x + 4y \leq 12$$

$$x \geq 0, y \geq 0$$

and use graphical reasoning to find the optimal solution.

In this case the constraints are conflicting because we require $-2x - 3y \leq -12$ or equivalently $2x + 3y \geq 12$ but we also require $3x + 4y \leq 12$.

Thus there is no feasible solution to this LP problem. This is illustrated in the figure.



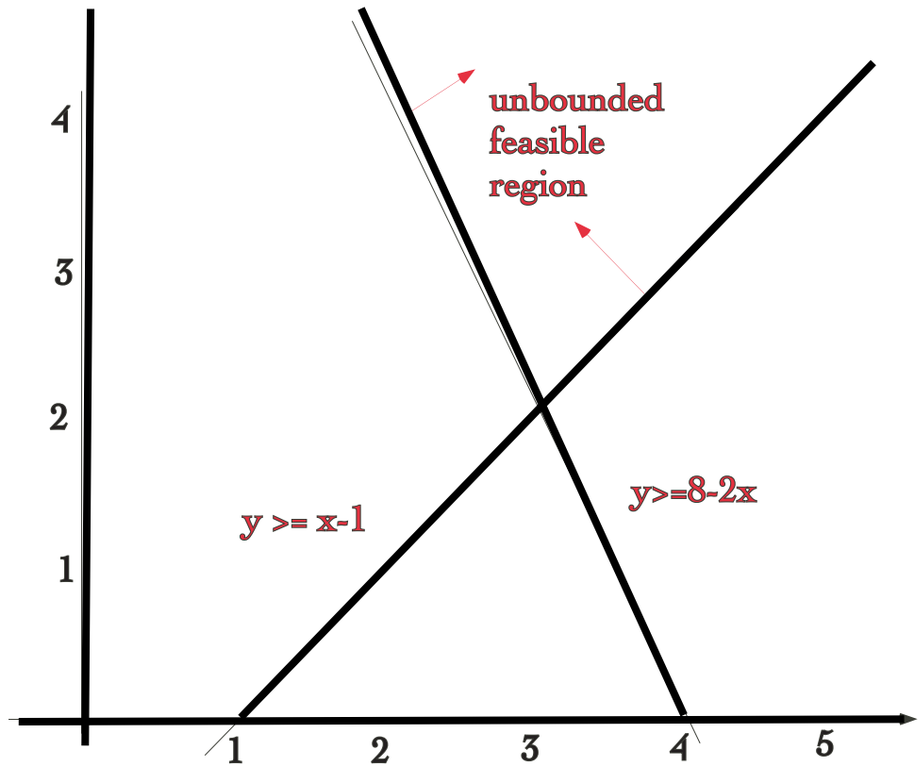
Not all linear programming problems have a solution!

Can we have an unbounded feasible set?

You can easily imagine a scenario where we are trying to maximize some quantity and the constraints yield an unbounded set. In this case there is no optimal solution. For example,

$$\text{maximize} \quad (x + 4y)$$

subject to $\quad x - y \leq 1, \quad 2x - y \leq 8, \quad x_1, x_2 \geq 0$



What if we changed the previous problem to minimizing $x + 4y$ instead of maximizing it? We have

$$\text{minimize} \quad (x + 4y)$$

$$\text{subject to} \quad x - y \leq 1, \quad 2x - y \leq 8, \quad x_1, x_2 \geq 0$$

We still have an unbounded feasible region but we want to minimize $x + 4y$ so

we take the only vertex of the region $x = 3, y = 2$ and the minimum value is 3+8=11. Any other value in the unbounded region produces a larger value.

If we have a solution is it guaranteed to be unique?

Consider the example

$$\text{maximize} \quad (3x_1 + 2x_2)$$

$$\text{subject to} \quad x_1 + x_2 \leq 5, \quad 3x_1 + 2x_2 \leq 12, \quad x_1, x_2 \geq 0$$

We want to maximize $(3x_1 + 2x_2)$ but from the constraint we see that the largest it can be is 12.

Is there more than one point which satisfies this? Yes.

Consider points on the line $3x_1 + 2x_2 = 12$ which satisfy $x_1 + x_2 \leq 5$. For example, (2,3) and (5/2,9/4); both points lie on the line $3x_1 + 2x_2 = 12$ and sum to ≤ 5; thus both are optimal solutions to this LP problem.

Thus we see that even if we have a bounded feasible solution set, we may not

have a unique optimal solution.

Exercise For each of the following problems, transform it into a LP problem and determine the solution graphically.

1. Michigan Polar Products makes downhill and crosscountry skis. A pair of downhill skis requires 2 man-hours for cutting, 1 man-hour for shaping and 3 man-hours for finishing while a pair of crosscountry skis requires 2 man-hours for cutting, 2 man-hours for shaping and 1 man-hour for finishing. Each day the company has available 140 man-hours for cutting, 120 man-hours for shaping and 150 man-hours for finishing. How many pairs of each type of ski should the company manufacture each day in order to maximize profit if a pair of downhill skis yields a profit of $10 and a pair of cross-country skis yields a profit of $8?

2. A dietitian wants to design a breakfast menu for certain hospital patients. The menu is to include two items A and B. Suppose that each ounce of A provides 2 units of vitamin C and 2 units of iron and each ounce of B provides 1 unit of vitamin C and 2 units of iron. Suppose the cost of A is 4 cents per ounce and the cost of B is 3 cents per ounce. If the breakfast menu must provide at least 8 units of vitamin C and 10 units of iron, how many ounces of each item should be provided in order to meet the iron and vitamin C requirements for the least cost? What will this breakfast cost?

Last time we saw that for a LP problem with two unknowns which has a bounded feasible set, then we can find an optimal solution graphically. Now we want to determine an approach for solving a general LP problem.

We have seen that for a LP problem where the constraints form a bounded feasible set, then the constraint equations form the boundary of a polyhedron which contains all feasible solutions to our problem.

Theorem. The set of all feasible solutions to a linear programming problem is a convex polyhedron.

Recall that a set S is convex if for any pair of points $x, y \in S$ then the line connecting x and y is in S. So, for example, a circle, a square, a triangle, a polygon are all convex sets. A star-shaped region or an L-shaped region are not convex.

In the examples where we had a bounded feasible set of solutions, the optimal

solution was at a vertex of the polyhedron, i.e., at an extreme point.

What we claim is that is that this is always true.

> For any bounded feasible linear programming problem there exists an optimal solution which corresponds to an extreme point (i.e., a vertex) of the polyhedron containing all feasible solutions.

This result is the basis of the standard algorithm for LP (the Simplex Method) because it moves from one vertex of the polyhedron to another while making sure the objective function never decreases (assuming we are maximizing a quantity).

Example Let's return to our first example of finding $\vec{x} \in \mathbb{R}^2$ such that $\vec{x}$ maximizes

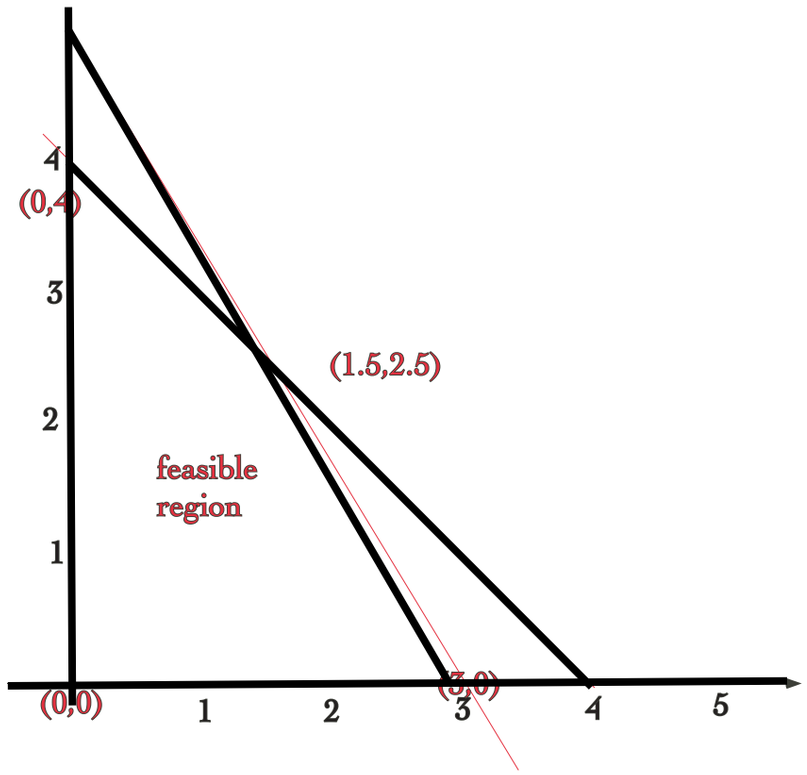
$$z = \begin{pmatrix} 120 & 100 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

subject to

$$\begin{pmatrix} 2 & 2 \\ 5 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq \begin{pmatrix} 8 \\ 15 \end{pmatrix}$$

$$\begin{pmatrix} x \\ y \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

The region is sketched below.



We see that the vertices of the polyhedron are

$$(0,0), \quad (0,4), \quad (3,0), \quad (\frac{3}{2},\frac{5}{2})$$

so all we have to do is evaluate z at each point.

We have at (0,0) $z = 0$.

At (0,4) $z = 400$.

At (3,0) $z = 360$.

At $(\frac{3}{2},\frac{5}{2})$, $z = 180 + 250 = 430$.

So the optimal solution is at $(\frac{3}{2},\frac{5}{2})$ which is the result we found graphically.

Slack Variables

We now want to begin to look at an approach for solving a general LP problem, i.e., one with more than two unknowns.

In general, inequality constraints are much more difficult to handle than equality constraints. So what we want to do is determine a way to convert our problem with inequality constraints to one with equality constraints. Of course this comes with a price; in our case, it is the introduction of additional variables.

Example Let's return to our first example again where we seek $\vec{x} \in I\!R^2$ such that $\vec{x}$ maximizes

$$z = \begin{pmatrix} 120 & 100 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

subject to

$$2x + 2y \leq 8, \qquad 5x + 3y \leq 15, \qquad x, y \geq 0$$

To convert these two constraints to equality constraints we introduce two new

variables $u \geq 0$ and $v \geq 0$ where

$$2x + 2y + u = 8 \qquad 5x + 3y + v = 15.$$

The new variables u, v are called slack variables because they "take up the slack" between the left and right sides of the constraint.

In matrix form our new problem becomes to find $\vec{x} = (x, y, u, v)$ such that $\vec{x}$ maximizes

$$z = \begin{pmatrix} 120 & 100 & 0 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ u \\ v \end{pmatrix}$$

subject to

$$\begin{pmatrix} 2 & 2 & 1 & 0 \\ 5 & 3 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ u \\ v \end{pmatrix} = \begin{pmatrix} 8 \\ 15 \end{pmatrix}, \qquad \begin{pmatrix} x \\ y \\ u \\ v \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Now if (x, y, u, v) is a solution to this new LP problem then (x, y) satisfies the original LP problem. To see this, we have $u = 8 - 2x - 2y$ and $v = 15 - 5x - 3y$. Because we assume $u \geq 0$ this says $8 - 2x - 2y \geq 0$ which implies $2x + 2y \leq 8$; similarly $v \geq 0$ implies $5x + 3y \leq 15$. Clearly these are our original constraints.

Conversely, if (x, y) satisfies the original LP problem then $2x + 2y \leq 8$ and $5x + 3y \leq 15$. If u, v are defined as $u = 8 - 2x - 2y$ and $v = 15 - 5x - 3y$, then $u \geq 0$ because $8 - 2x - 2y \geq 0$; similarly $v \geq 0$ because $15 - 5x - 3y \geq 0$. Thus (x, y, u, v) satisfies the new LP problem with equality constraints.

If our original constraint was, e.g, $2x + 2y \geq 8$ then we want $2x + 2y - 8 \geq 0$ and $u \geq 0$ so we would take $u = 2x + 2y - 8$ or equivalently $2x + 2y - u = 8$.

If we have m inequality constraints (other than the ones which say the variables are non-negative) then we simply add m slack variables, one for each constraint. In general, our problem with m constraints becomes

Linear Programming Problem with Slack Variables : Assume we originally had n variables and m inequality constraints. After introducing slack variables $x_{n+1}, \ldots, x_{n+m}$ we seek values $x_1, x_2, \ldots, x_n, x_{n+1}, \ldots, x_{n+m}$ to

$$\text{maximize} \quad z = c_1 x_1 + c_2 x_2 + \cdots + c_n x_n$$

subject to the conditions

$$a_{11} x_1 + a_{12} x_2 + \cdots + a_{1n} x_n + x_{n+1} = b_1$$

$$a_{21} x_1 + a_{22} x_2 + \cdots + a_{2n} x_n + x_{n+2} = b_2$$

$$\vdots$$

$$a_{m1} x_1 + a_{m2} x_2 + \cdots + a_{mn} x_n + x_{n+m} = b_m$$

$$x_j \geq 0, \quad j = 1, 2, \ldots, n+m$$

Solving the LP with slack variables is equivalent to solving it in Standard form.

Now we need to figure our how to solve a LP problem in this form, i.e., with equality constraints (except for the constraints that the components of $\vec{x}$ are non-negative).

First, we rewrite our LP problem using matrix notation.

Matrix form of LP Problem Using Slack Variables Let $\vec{c} = (c_1, c_2, \cdots, c_n, 0, 0, \ldots, 0) \in \mathbb{R}^s$ where $s = n + m$. Then we seek $\vec{x} \in \mathbb{R}^s$ to

$(\dagger)$ maximize $z = \vec{c}^T \vec{x}$

subject to the equality constraints $A\vec{x} = \vec{b}$ and $\vec{x} \geq \vec{0}$

Here A is the $m \times s$ matrix with $m \leq s$ given by

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} & 1 & 0 & \cdots & \cdots & 0 \\ a_{21} & a_{22} & \cdots & a_{2n} & 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots & & & & & \\ a_{m1} & a_{m2} & \cdots & a_{mn} & 0 & \cdots & \cdots & 0 & 1 \end{pmatrix}$$

We assume that the rank of A is m.

Because we assume that the rank of A is m we know that A has m linearly independent columns and m linearly independent rows. If we choose m linearly independent columns of A then they form a basis for $\mathbb{R}^m$.

Now the constraints (other than $\vec{x} \geq 0$) form a linear system of equations where A is an $m \times s$ (with $s = m + n$) matrix; because $s > m$ this is an under-determined system, i.e., we have more unknowns than equations and thus expect multiple solutions. Because we have assumed that A has rank m the constraint equations are not redundant.

Let S denote the set of solutions to this under-determined linear system where $x_i \geq 0$ for all i.

Example Consider the under-determined system $A\vec{x} = \vec{b}$ where

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & -1 & -1 & 0 & -1 & -1 \\ 1 & 2 & 2 & 1 & 1 & 1 \end{pmatrix}$$

A has rank 3 and so its null space has dimension 3 (because 6-3=3). To solve this in the usual way, we have 3 degrees of freedom, i.e., we can set 3 of the components of $\vec{x}$ and use the system to solve for the remaining 3. Of course there are an infinite number of choices for these variables.

If we solved this system by row-reducing it, we would be lead to the equivalent system

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & -1 & -1 & 0 & -1 & -1 \\ 1 & 2 & 2 & 1 & 1 & 1 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & -1 & 1 \\ 0 & 2 & 1 & 1 & 0 & 1 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & -1 & -1 & 0 & -1 & -1 \\ 0 & 0 & 1 & -1 & 2 & 1 \end{pmatrix}$$

which says that we can set the last three variables arbitrarily and we can get the 3 linearly independent solutions to our problem. However, this requires more work than we are willing to do because we don't need the most general solution.

Because we want $x_i \geq 0$ for all i, let's pick 3 of the components of $\vec{x}$, set them to zero and see the result.

If we set $x_2 = x_3 = x_5 = 0$ then we get the system

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & -1 & -1 & 0 & -1 & -1 \\ 1 & 2 & 2 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ 0 \\ 0 \\ x_4 \\ 0 \\ x_5 \\ x_6 \end{pmatrix} = \vec{b}$$

or equivalently we have the reduced system

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_4 \\ x_6 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Because the coefficient matrix here is invertible, we have the solution $x_1 = b_1$, $x_6 = -b_3$ and $x_1 + x_4 + x_6 = b_3$ implies $x_4 = b_3 - b_1 + b_3$ and a solution to the under-determined system is

$$\vec{x} = (b_1, 0, 0, b_3 - b_1 + b_3, 0, -b_3)^T$$

Does this always work? If we choose $x_1 = x_3 = x_5 = 0$ then we get the reduced

system

$$\begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & -1 \\ 2 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_2 \\ x_4 \\ x_6 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

but clearly in this system the coefficient matrix is singular so we don't have a unique solution; that is, we aren't free to choose any 3 components arbitrarily. This is because in our original matrix the 6th column plus the 4th column yields the 2nd column so they are linearly dependent.

Definition: A basic solution to the under-determined system $A\vec{x} = \vec{b}$ where A is $m \times s$ with rank m is a solution where we have set $s - m$ components of the solution to 0. The basic variables in the basic solution are those which are not set to 0.

Definition: A basic feasible solution to our LP problem with slack variables is a basic solution to $A\vec{x} = \vec{b}$ which is also a feasible solution to our LP problem.

Theorem For our LP problem $(\dagger)$ we have that every basic feasible solution is an extreme point and every extreme point is a basic feasible solution. Moreover there are a finite number of basic feasible solutions to $(\dagger)$.

So this theorem tells us that all we need to do is find every basic feasible solution to $(\dagger)$ and then evaluate z there and choose the largest.

Example Let's return to our first example again where we seek $\vec{x} \in I\!R^2$ such that $\vec{x}$ maximizes

$$z = \begin{pmatrix} 120 & 100 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

subject to

$$2x_1 + 2x_2 \leq 8, \qquad 5x_1 + 3x_2 \leq 15$$

$$x_1, x_2 \geq 0$$

We converted this using slack variables x_3, x_4 to get the under-determined system

$$\begin{pmatrix} 2 & 2 & 1 & 0 \\ 5 & 3 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 8 \\ 15 \end{pmatrix}, \qquad \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Because $\vec{x} \in I\!R^4$ we have 6 possible combinations for setting two components to zero

$$x_1 = x_2 = 0, \quad x_1 = x_3 = 0, \quad x_1 = x_4 = 0,$$

$$x_2 = x_3 = 0, \quad x_2 = x_4 = 0, \quad x_3 = x_4 = 0$$

For each combination we need to determine if it is a basic feasible solution or not; if it is, then we evaluate z there and choose our maximum. Here we show the details for 2 choices and the rest are summarized in the table below.

For $x_1 = x_2 = 0$ we have the system

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 8 \\ 15 \end{pmatrix} \implies x_3 = 8, x_4 = 15$$

Because the system is solvable and $x_3, x_4 \geq 0$ this is a feasible solution so we evaluate z there to get 0.

For $x_2 = x_3 = 0$ we have the system

$$\begin{pmatrix} 2 & 0 \\ 5 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_4 \end{pmatrix} = \begin{pmatrix} 8 \\ 15 \end{pmatrix} \implies x_1 = 4, x_4 = 15 - 5(4) = -5 < 0$$

Therefore this is NOT a feasible solution because x_4 violates the non-negativity constraint so we discard it.

For each basic solution we have

x_1	x_2	x_3	x_4	feasible or not	z
0	0	8	15	feasible	0
0	4	0	3	feasible	400
0	5	-2	0	not feasible	-
4	0	0	-5	not feasible	-
3	0	2	0	feasible	360
1.5	2.5	0	0	feasible	430

As we can see from the table, we get the maximum of $z = 430$ at $(1.5, 2.5)$ which is the result we found by geometrical arguments.

In the next example, we take a linear programming problem from its initial formulation to its solution (by hand). In addition, we compare our result with the solution we can obtain via graphical arguments because there are only 2 unknowns.

Example Consider the LP problem

$$\text{maximize} \qquad z = 2x_1 + 3x_2$$

subject to the constraints

$$3x_1 + x_2 \leq 6; \quad x_1 + x_2 \leq 4, \quad x_1 + 2x_2 \leq 6$$

$$x_1, x_2 \geq 0$$

We have 3 constraints so we need to add a slack variable for each. Our inequality constraints become

$$3x_1 + x_2 + x_3 = 6; \quad x_1 + x_2 + x_4 = 4, \quad x_1 + 2x_2 + x_5 = 6$$

$$x_i \geq 0, 1, 2, \ldots, 6$$

and our under-determined system has the coefficient matrix which is 3×5 with a rank of 3. Therefore it has a two dimensional nullspace.

$$A = \begin{pmatrix} 3 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 2 & 0 & 0 & 1 \end{pmatrix}$$

This means we can set 2 variables to zero. We have 5 variables so we have 10 possibilities.

If we set $x_1 = x_2 = 0$ then we have the system

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 6 \\ 4 \\ 6 \end{pmatrix} \implies x_3 = 6, x_4 = 4, x_6 = 6$$

so it is a basic feasible solution.

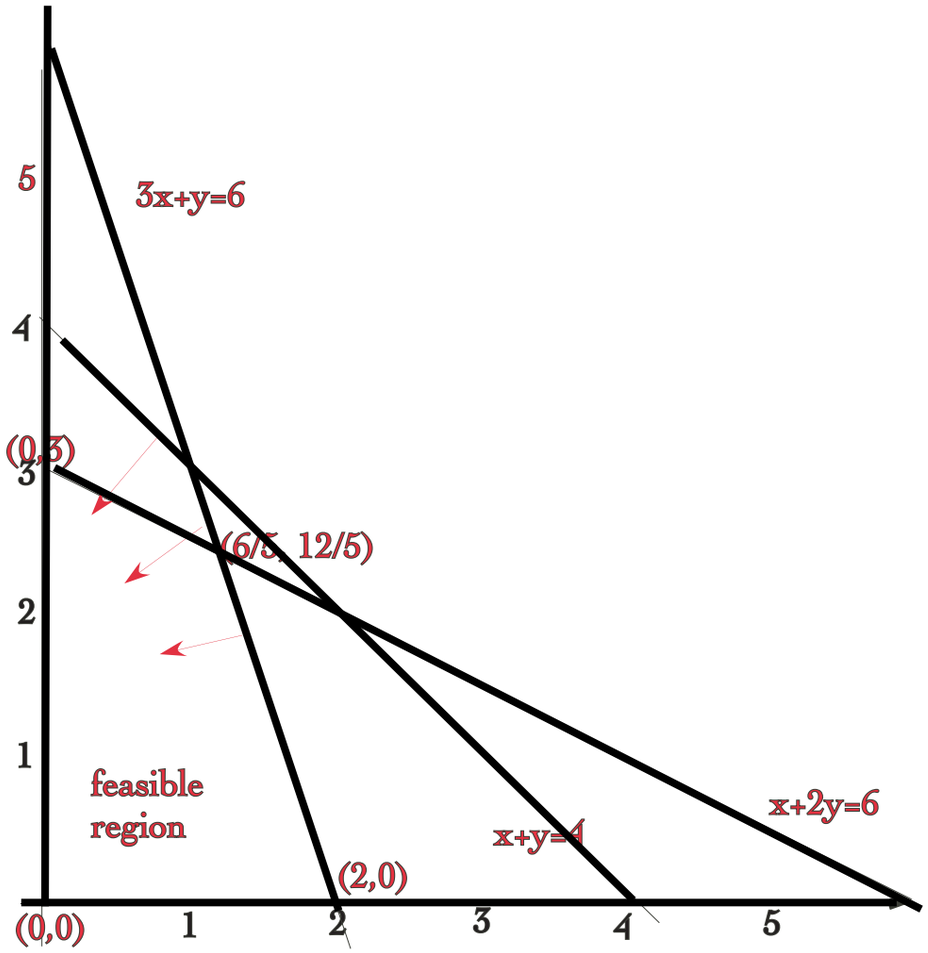
If we set $x_1 = x_3 = 0$ then we have the system

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_2 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 6 \\ 4 \\ 6 \end{pmatrix} \implies x_2 = 6, x_4 = -2, x_6 = -6$$

so it is NOT a basic feasible solution since the constraints $x_i \geq 0$ are not satisfied.

Continuing in this fashion we get the results given in the table. There are four feasible solutions and we see that the optimal one occurs at $(1.2, 2.4)$.

x_1	x_2	x_3	x_4	x_5	feasible or not	z
0	0	6	4	6	feasible	0
0	6	0	-2	-6	not feasible	-
0	4	2	0	-2	not feasible	-
0	3	3	1	0	feasible	9
2	0	0	2	4	feasible	4
4	0	-6	0	2	not feasible	-
6	0	-6	2	0	not feasible	-
1	3	0	0	-1	not feasible	-
1.2	2.4	0	0.4	0	feasible	9.6
2	2	-2	0	0	not feasible	



Vertices: (0,0), (2,0), (0,3) (6/5,12/5)

Graphically we see that the vertices are (0,0), (2,0), (1.2,2.4), and (0,3) which are the 4 feasible basic solutions that we found above.

Implementing this approach to the Linear Programming Problem

We have seen one approach for solving the LP problem after we have converted it to a problem with equality constraints by introducing slack variables.

This may not be the most efficient way to solve a LP problem, but implementation is straightforward so we look at it here. However, it will be effective when we have a relatively small number of constraints.

The steps in our approach are as follows:

1. To initialize, we input our $m \times s$ constraint matrix A, the right hand side vector $\vec{b}$ and the vector $\vec{c}$ such that we want to maximize $z = \vec{c}^T \vec{x}$. Initialize z.
2. Determine the number of variables we set to zero.
3. Loop over all possible combinations of variables to set to zero.

- Remove columns of A corresponding to the variables we set to zero to form an $m \times m$ matrix $\widetilde{A}$.
- Solve $\widetilde{A}\vec{y} = \vec{b}$ for the variables which are not set to zero.
- Determine if the solution is feasible or not.
- If it is feasible, evaluate z there. If this value of z is larger than current value, save solution and z value.

So if we use the backslash command to solve the linear system, all we really need to write is a routine which forms the reduced matrix $\widetilde{A}$ from A.

```
function a_reduced = delcols(a,i)
% Delete specified columns of the matrix a.
% Particular columns to delete are in integer array i where
% it is assumed these are in order
%
[m,s]=size(a);

loc=1;
```

```
first = 0;
for j = 1:s

   if j == i(loc)

       loc=loc+1;
       if loc > size(i)
           if j < s
             a_reduced = [ a_reduced, a(:,j+1:s) ]
           end
           break
       end
   else


       if ( first == 0 )
           a_reduced = [ a(:,j) ]
           first = 1;
       else
```

```
            a_reduced = [ a_reduced, a(:,j) ]
        end
    end
end
end
```

However, there is a function in Matlab which makes this process a lot easier.

Suppose we want to eliminate columns 3, 7, 9 from a 5×10 matrix. Then we set

```
i=[3, 7,9];
j=[1:10];
```

```
k=setdiff(j,i)
```

Then the result is

```
 1     2     4     5     6     8    10
```

Thus all we have to set is

```
a_reduced = a(:,k);
```

to get our reduced matrix. It's a lot simpler when you know the right command!

Let's return to one of our initial "real world" problems and solve it using the approach of finding all feasible solutions. Recall that the problem was summarized as follows:

$$\text{minimize}\Big[\sum_{i=1}^{2}\sum_{j=1}^{3} c_{ij}x_{ij}\Big]$$

where $c_{11} = 5$, $c_{12} = 7$, $c_{13} = 9$, $c_{21} = 6$, $c_{22} = 7$, $c_{23} = 10$

subject to the constraints

$$x_{11} + x_{12} + x_{13} \leq 120 \quad x_{21} + x_{22} + x_{23} \leq 140 \quad \text{(supplies)}$$

$$x_{11} + x_{21} \geq 100 \quad x_{12} + x_{22} \geq 60 \quad x_{13} + x_{23} \geq 80 \quad \text{(demands)}$$

$$x_{ij} \geq 0, i = 1, 2; j = 1, 2, 3$$

If we wrote our code to compute all possible feasible solutions then it really

doesn't matter whether we are maximizing or minimizing z. The first thing we do is introduce slack variables so that we have equality constraints.

We renumber our unknowns as $x_1, x_2, \cdots, x_6$ and introduce 5 slack variables $x_7, \cdots, x_{11}$.

We want to minimize

$$z = 5x_1 + 7x_2 + 9x_3 + 6x_4 + 7x_5 + 10x_6$$

subject to the constraints

$$x_1 + x_2 + x_3 + x_7 = 120 \quad x_4 + x_5 + x_6 + x_8 = 140 \quad \text{(supplies)}$$

$$x_1 + x_4 - x_9 = 100 \quad x_2 + x_4 - x_{10} = 60 \quad x_3 + x_6 - x_{11} = 80 \quad \text{(demands)}$$

$$x_i \geq 0, \quad i = 1, 2, \ldots, 11$$

Note that for each solution we obtain we have to solve a 5×5 system of equations.

Notice that for the inequalities which were originally $\geq$, we subtract the slack variable. This is because, e.g.,

$$x_9 = x_1 + x_4 - 100, \quad x_9 \geq 0 \implies x_1 + x_4 \geq 100$$

Now for our code to calculate all basic feasible solutions we need to input A, $\vec{b}$ and $\vec{c}$. We have

$$A = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 \end{pmatrix}$$

$$\vec{c} = (5, 7, 9, 6, 7, 10, 0, 0, 0, 0, 0)^T \qquad \vec{b} = (120, 140, 100, 60, 80)^T$$

We present the results in the table below where we only give the feasible solutions, i.e., where $x_i \geq 0$ for all i.

x_1	x_2	x_3	x_4	x_5	x_6	Cost
0	60	60	120	0	20	\$1880
0	40	80	100	40	0	\$1880
100	0	20	0	80	60	\$1840
40	0	80	80	60	0	\$1820
40	0	80	60	80	0	\$1840
40	0	80	80	60	0	\$1820
100	20	0	0	60	80	\$1860
60	60	0	60	0	80	\$1880
100	20	0	0	40	80	\$1720

So our conclusion is that

- Salt Lake City (whose maximum supply is 120 tons) should ship 100 tons to LA, 20 tons to Chicago, and none to NYC
- Denver (whose maximum supply is 140 tons) should ship none to LA, 40 tons to Chicago and 80 tons to NYC
- The transportation costs for this minimum configuration is \$1720 .

Exercise For each LP problem, introduce slack variables and change the inequality constraints to equality constraints.

1. We want to maximize $4x_1 + 12x_2 + 8x_3$ subject to the constraints

$$3x_1 + 2x_2 - 6x_3 \leq 20, \qquad x_1 + 6x_2 \geq 10, \qquad x_i \geq 0, i = 1, 2, 3$$

2. The "diet problem" from Lecture 1

$$\text{minimize}\Big[.75x_1 + 2.4x_2 + .5x_3 + .5x_4 + 2.5x_5 + .25x_6\Big]$$

subject to the constraints

$$110x_1 + 205x_2 + 160x_3 + 160x_4 + 420x_5 + 260x_6 \geq 2000$$

$$4x_1 + 32x_2 + 13x_3 + 8x_4 + 4x_5 + 14x_6 \geq 55$$

$$2x_1 + 12x_2 + 54x_3 + 285x_4 + 22x_5 + 80x_6 \geq 800$$

$$0 \leq x_1 \leq 4, \quad 0 \leq x_2 \leq 3, \quad 0 \leq x_3 \leq 2,$$

$$0 \leq x_4 \leq 8, \quad 0 \leq x_5 \leq 2, \quad 0 \leq x_6 \leq 2$$

BRUTE FORCE Linear programming algorithm

The steps in our approach are as follows:

1. To initialize, we input our $m \times s$ constraint matrix A, the right hand side vector $\vec{b}$ and the vector $\vec{c}$ such that we want to maximize $z = \vec{c}^T \vec{x}$. Initialize z.

2. Determine the number of variables we set to zero.

3. Loop over all possible combinations of variables to set to zero.

   - Remove columns of A corresponding to the variables we set to zero to form an $m \times m$ matrix $\widetilde{A}$.
   - Solve $\widetilde{A}\vec{y} = \vec{b}$ for the variables which are not set to zero.
   - Determine if the solution is feasible or not.
   - If it is feasible, evaluate z there. If this value of z is larger than current value, save solution and z value.

The Simplex Method

Our goal now is to develop an algorithm for solving the LP problem when our constraints have been converted to equality constraints which does not involve solving so many linear systems as we did in the last approach.

As we saw in our previous examples, we could solve the LP problem by

(i) finding all basic solutions by solving $m \times m$ linear systems;

(ii) discard the ones which are not feasible;

(iii) evaluate the function to maximize at each feasible basic solution and pick the maximum.

However, for each basic solution we have to solve a linear $m \times m$ system with a different coefficient matrix each time; this can be expensive if we have many constraints, i.e., m is large.

What should our strategy be to reduce the amount of work?

- One reduction in work that we would like to accomplish is to only find feasible solutions because a solution to our linear system which doesn't satisfy the constraints (or the matrix is not invertible) is discarded.
- If we determine a new feasible solution and when we evaluate z there, if it is less than the current maximum, then this is discarded and our work is wasted. So what we would like to do is to start at a feasible solution and then move to another feasible solution which has a larger value of z than the current one.

Of course we also need a way to determine when we have found an optimal solution.

Let's look again at our table for one of the previous examples where we only include the feasible solutions in the table.

x_1	x_2	x_3	x_4	feasible or not	z
0	0	8	15	feasible	0
0	4	0	3	feasible	400
3	0	2	0	feasible	-
1.5	2.5	0	0	feasible	430

The first point has basic variables (i.e., the ones which aren't 0) x_3, x_4

The second point has basic variables x_2, x_4

The third point has basic variables x_1, x_3

The fourth point has basic variables x_1, x_2

Note that

- the first and second points have the basic variable x_4 in common
- the first and third points have the basic variable x_3 in common
- the first and fourth points have none in common
- the second and third points have none in common
- the second and fourth points have the basic variable x_2 in common

- the third and fourth points have the basic variable x_1 in common

If two basic feasible solutions have a variable in common then we call them adjacent.

In our example above, all points are adjacent to each other except (i) one and four and (ii) two and three.

The Simplex Method was developed by George Dantzig in 1947 as a more efficient approach to solving the LP problem than finding all basic feasible solutions. The idea is that it starts with a basic feasible solution and then moves to an adjacent point in such a way that the function to maximize increases.

What do we need to implement the Simplex Method?

(1.) an initial basic feasible solution

(2.) a way to determine an adjacent basic feasible solution for which z is larger than the value at the current basic feasible solution;

(3.) a way to determine if a given basic feasible solution is an optimal solution; i.e., to terminate our search.

Determining the Initial Basic Feasible Solution

In general, finding an initial basic feasible solution can be a major difficulty. However, we will make an assumption which makes it easy.

We assume that each component of $\vec{b}$ satisfies $b_i \geq 0$. If it is not, an alternate approach must be used.

There are $m+n = s$ components in $\vec{x}$; m slack variables (one for each constraint) plus the n original variables. To get an initial feasible solution we set the original (nonslack) variables to 0. When we do this our system is reduced to the $m \times m$ system with the coefficient matrix the identity

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ & & & \ddots & \\ 0 & 0 & \cdots & 0 & 1 \end{pmatrix} \begin{pmatrix} x_{n+1} \\ x_{n+2} \\ \vdots \\ x_{n+m} \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

and thus

$$x_1 = x_2 = \cdots = x_n = 0 \qquad x_{n+1} = b_1, \quad x_{n+2} = b_2, \quad \cdots \quad x_{n+m} = b_m$$

Because we have assumed that $b_i \geq 0$ then this is a basic feasible solution.

Example Let's return to our original example of finding $\vec{x} \in I\!R^2$ such that $\vec{x}$ maximizes

$$z = \begin{pmatrix} 120 & 100 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

subject to

$$2x_1 + 2x_2 \leq 8, \qquad 5x_1 + 3x_2 \leq 15, \qquad x_1, x_2 \geq 0$$

We converted this using slack variables x_3, x_4 to get the under-determined system

$$\begin{pmatrix} 2 & 2 & 1 & 0 \\ 5 & 3 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 8 \\ 15 \end{pmatrix}, \qquad \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

We have that each $b_i > 0$ so our approach works. We set $x_1 = x_2 = 0$ and then

$x_3 = 8$ and $x_4 = 15$. Our initial basic feasible solution is

$$(0, 0, 8, 15)$$

Often one summarizes the results in a Tableau. To do this we write the objective as

$$z - 120x_1 - 100x_2 = 0$$

and include it as a variable in the table. We have our initial tableau as

	x_1	x_2	x_3	x_4	z	
x_3	2	2	1	0	0	8
x_4	5	3	0	1	0	15
	-120	-100	0	0	1	0

The variables on the left are the basic variables for the point, i.e., the ones we have not set to zero so we are solving for them. All rows except the last represent the constraints so the values under x_1 through x_4 are just the entries in our under-determined system with the last column giving the right hand side. The last row represents the objective $z - 120x_1 - 100x_2$.

Notice that the row corresponding to the basic variable x_3 has a coefficient $+1$ in the x_3 column and the other entry in that column is 0; this means that x_3 is set to the right most column which here is 8. In the second row we have the basic variable x_4 with its coefficient as $+1$, no other nonzero entries in that column and it is set to 15.

We say that this system is in canonical form with respect to the variables x_3, x_4 because each basic variable must (i) appear in exactly one equation with a coefficient of $+1$ and (ii) other entries in that column are 0. Then its value is set to the entry in the right most column.

The idea is that at each step of the process we will have an updated tableau.

How do we move to an adjacent point so that our function increases?

Let's look at this in terms of our example. At our initial point $(0, 0, 8, 15)$ we have that $z = 120x_1 + 100x_2 + 0x_3 + 0x_4 = 0$ so we can increase it if we move to a point where either x_1 or x_2 are > 0 because both of our coefficients in the expression for z are > 0.

In general we increase any of the nonbasic variables which has a positive coefficient.

So in our example how do we choose whether to increase x_1 or x_2? There are different strategies; for example, we could (i) choose the first one with a negative coefficient (because we wrote $z - 120x_1 - 100x_2$ in the table) or (ii) choose the one with the largest negative coefficient in the objective row.

We will choose the second criteria so that for our example the most negative coefficient is -120 so we increase x_1.

The variable to be increased is called the entering variable because in the next iteration it will become a basic variable, i.e., enter the set of basic variables.

We always have to satisfy the linear system $A\vec{x} = \vec{b}$ so if we increase one variable then we must decrease another one. So in our case if we increase x_1, keep x_2 at 0 then we have

$$2x_1 + 2x_2 + x_3 = 8 \implies x_3 = 8 - 2x_1$$

$$5x_1 + 3x_2 + x_4 = 15 \implies x_4 = 15 - 5x_1$$

Thus an increase in x_1 decreases both x_3 and x_4.

Are we free to increase x_1 as much as we want?

Of course the answer is no because we require $x_3, x_4 \geq 0$. So we really need

$$x_3 = 8 - 2x_1 \geq 0 \implies x_1 \leq 4 \quad \text{and} \quad x_4 = 15 - 5x_1 \geq 0 \implies x_1 \leq 3$$

Thus we can't increase x_1 by more than 3 in our example. If $x_1 = 3$ then $x_3 = 2$ and $x_4 = 0$. Thus our next basic feasible solution is

$$(3, 0, 2, 0)$$

and it is adjacent to the original one $(0, 0, 8, 15)$ because it still has one basic variable (x_3) in common. When we evaluate z we see that it is now 360 which is a great improvement over 0.

Note that the fact that we can increase x_1 by no more than 3 can be seen from the tableau. For the row corresponding to x_3 we see that the coefficient of x_1 is 2 and the right hand side term is 8 so we have 8/2=4; for the row corresponding

to x_4 we see that the coefficient of x_1 is 5 and the right hand side term is 15 so we have 15/5=3.

The fourth variable x_4 is now zero so it is no longer a basic variable; we call it a departing variable because it has departed the set of basic variables. The column of the entering variable is called the pivotal column and the row labeled with the departing variable is called the pivotal row. The entry in the pivotal column and pivotal row is the pivot.

So this procedure seemed clear with our small example. We determined the most negative entry in the objective row and took the corresponding variable to be our new basic variable, i.e., our entering variable. Because we want to increase z, we then used the constraints to determine the largest value that the new basic variable could have; this resulted in one of the current basic variables being set to zero; i.e., it became the departing variable. Once we had the largest value for our new variable, we were able to compute the value of the other basic variable, in our case x_3. We then had our new feasible solution which was constructed to have a larger value of z then the previous adjacent solution. In summary, we moved from one vertex of the polyhedron containing all feasible solutions to an

adjacent node which was guaranteed to have a larger z value.

Now we want to systematize this procedure. To do this, let's think of the tableau as containing our under-determined system and we want to perform elementary row operations (as we do in Gauss elimination) to make the entry in the row and column of the entering variable become $+1$ and to make sure all other entries in that column have been zeroed out.

Let's look at our new tableau corresponding to this iteration. We want the coefficient of our entering basic variable x_1 to be $+1$ because it is our new basic variable and we want to make sure the previous basic variable x_3 also maintains a $+1$ in the correct entry.

The departing variable will be x_4 so we use this row for the entering variable x_1. We have $5x_1 + 3x_2 + x_4 = 15$ so the elementary row operation to make the coefficient of x_1 to be $+1$ is to multiply this row by $1/5$. We have $x_1 + \frac{3}{5}x_2 + \frac{1}{5}x_4 = 3$.

Now we need to make the other entry in that column to be 0 so we need to take

a multiple of the new row and add to the first. We multiply our new second row by -2 and add to the first to get $\frac{4}{5}x_2 + x_3 - \frac{2}{5}x_4 = 2$. The coefficient of the basic variable x_3 will remain $+1$ because the entry in that column in the other row is 0.

Thus we have that $x_3 = 2$ and $x_1 = 3$ with $x_2 = x_4 = 0$. If we evaluate z at this point we get 360.

Finally we update the objective row. What we are essentially doing is introducing a 0 into the column corresponding to the entering variable x_1. To this end we multiple our new second row by 120 and add to the objective row to get

$$0 \cdot x_1 + \left[120(\frac{3}{5}) - 100\right] x_2 + 0 \cdot x_3 + \frac{120}{5} x_4 = 0 \cdot x_1 - 28x_2 + 0 \cdot x_3 + 24x_4 = 3 * 120$$

Thus the tableau for the second step is as follows.

	x_1	x_2	x_3	x_4	z	
x_3	0	$\frac{4}{5}$	1	$-\frac{2}{5}$	0	2
x_1	1	$\frac{3}{5}$	0	$\frac{1}{5}$	0	3
	0	-28	0	24	1	360

The process that we used to get from the first tableau is called pivoting.

Return again to our initial tableau where we mark the departing variable with "$\leftarrow$" and the entering variable with "$\downarrow$". We locate and mark the entry at the intersection of the pivotal row and pivotal column, i.e., the position where we have an entering and departing variable.

	$\downarrow$					
	x_1	x_2	x_3	x_4	z	
x_3	2	2	1	0	0	8
$\leftarrow x_4$	5	3	0	1	0	15
	-120	-100	0	0	1	0

Now our pivotal element is 5 so divide that row by 5. This is the new row in our second tableau. Now we have to make our pivotal column all 0's except for the 1 in the pivotal element. So in our case we multiply by -2 and add to the first to get our new first row and then multiply by 120 and add to the last.

In general, the steps to go from one tableau to the next are:

- decide on the entering variable based on some criteria such as the one with the most negative entry in the objective row;
- locate the departing variable to give the pivotal row; locate the entering variable to give the pivot column; then mark the pivotal element;
- if the pivotal element is k, divide row by k

- add multiples of the pivotal row (which now has 1 in the pivot) to zero out other entries in that column

Now let's take our second tableau and do this to get the third.

First the entering variable is x_2 because it has the most negative entry in the objective row. We flag the departing variable which is x_3.

	$\downarrow$				
	x_1	x_2	x_3	x_4	z	
$\leftarrow x_3$	0	$\frac{4}{5}$	1	$-\frac{2}{5}$	0	2
x_1	1	$\frac{3}{5}$	0	$\frac{1}{5}$	0	3
	0	-28	0	24	1	360

To get a $+1$ in the pivot element we multiply the first row by $5/4$ to get

		$\downarrow$				
	x_1	x_2	x_3	x_4	z	
$\leftarrow x_3$	0	1	$\frac{5}{4}$	$-\frac{1}{2}$	0	$\frac{5}{2}$
x_1	1	$\frac{3}{5}$	0	$\frac{1}{5}$	0	3
	0	-28	0	24	1	360

Now we use that new row to zero out the other entries in the column. We modify the second row and the objective row to get our third tableau

	x_1	x_2	x_3	x_4	z	
x_2	0	1	$\frac{5}{4}$	$-\frac{1}{2}$	0	$\frac{5}{2}$
x_1	1	0	$-\frac{3}{4}$	$\frac{1}{2}$	0	$\frac{3}{2}$
	0	0	35	10	1	420

You will recognize this solution $(\frac{3}{2}, \frac{5}{2}, 0, 0)$ as the optimal solution that we found before.

How do we know if we have found an optimal solution?

If the objective row of a tableau has zero entries in the columns labeled by the basic variables and no negative entries in the columns labeled by nonbasic variables, then the solution represented by the tableau is optimal.

We now take an example of a LP problem through all the steps of the Simplex Method.

Example Use the Simplex Method to find an optimal solution to the following LP problem.

$$\text{maximize} \quad 2x_1 + 3x_2 + 3x_3$$

subject to

$$3x_1 + 2x_2 \leq 60 \quad -x_1 + x_2 + 4x_3 \leq 10 \quad 2x_1 - 2x_2 + 5x_3 \leq 50$$

$$x_1, x_2, x_3 \leq 0$$

Our first step is to add slack variables so we will have equality constraints. We have 3 constraints so we need to add 3 additional variables x_4, x_5, x_6. The conditions become

$$3x_1 + 2x_2 + x_4 = 60 \quad -x_1 + x_2 + 4x_3 + x_5 = 10 \quad 2x_1 - 2x_2 + 5x_3 + x_6 = 50$$

In matrix form we seek $\vec{x} = (x_1, x_2, \ldots, x_6)$ which

$$\text{maximizes} \quad (2, 3, 3, 0, 0, 0)^T \vec{x}$$

subject to

$$A\vec{x} = \vec{b} \qquad \vec{x} \geq \vec{0}$$

where

$$A = \begin{pmatrix} 3 & 2 & 0 & 1 & 0 & 0 \\ -1 & 1 & 4 & 0 & 1 & 0 \\ 2 & -2 & 5 & 0 & 0 & 1 \end{pmatrix} \qquad \vec{b} = \begin{pmatrix} 60 \\ 10 \\ 50 \end{pmatrix}$$

Because $b_i > 0$ for $i = 1, 2, 3$ then we can start with the initial feasible solution where we set $x_1 = x_2 = x_3 = 0$. We write our objective as $z - 2x_1 - 3x_2 - 3x_3 = 0$. Our initial tableau is the following

	x_1	x_2	x_3	x_4	x_5	x_6	z	
x_4	3	2	0	1	0	0	0	60
x_5	-1	1	4	0	1	0	0	10
x_6	2	-2	5	0	0	1	0	50
	-2	-3	-3	0	0	0	1	0

So our initial feasible solution is $(0, 0, 0, 60, 10, 50)$ and z evaluated there is zero.

Now we decide which is the entering basic variable. The entry that is most negative is either x_2 or x_3; both are -3. We choose x_2 since it occurs first in the table.

Now if x_2 is the entering variable then we need to determine which variable x_4, x_5 or x_6 is the departing variable. Once we determine this, then we have the pivot element and we can proceed to make that element's entry $+1$ and zero out the other entries in the column.

Now we want to make x_2 as large as possible so this dictates the departing variable. We take x_5 as the departing variable because if $x_1 = x_3 = 0$ then $x_4 = 60 - 2x_2 \geq 0 \implies x_2 \leq 30$; $x_5 = 10 - x_2 \geq 0 \implies x_2 \leq 10$ and $x_6 = 50 + 2x_2 \geq 0$. So the largest x_2 can be is 10. This can be seen from the table without performing these calculations where we take as the departing variable the one corresponding to the row which has the smallest $positive$ ratio of the right hand side and the coefficient of x_2 in that column.

Our pivot element is in row and column 2. This entry is already $+1$ so all we need to do is zero out the (1,2) and (3,2) entries. We multiple the second row by -2 and add to the first; then we multiply by $+2$ and add to the third. The objective row is modified by multiplying row 2 by 3 and adding. We have the next tableau as follows.

	x_1	x_2	x_3	x_4	x_5	x_6	z	
x_4	5	0	-8	1	-2	0	0	40
x_2	-1	1	4	0	1	0	0	10
x_6	0	0	13	0	2	1	0	70
	-5	0	9	0	3	0	1	30

Our new feasible solution is (0, 10, 0, 40, 0, 70) and the value of z there is 30.

This is not an optimal solution because we still have a negative coefficient in our objective row. The coefficient of x_1 is negative in the objective row (the only one) so it is our new entering basic variable.

So do we take x_4 or x_6 as the departing variable now? We take x_4 because 40/5=8 for the x_4 row but for the x_6 row the entry in the x_1 column is 0.

Our pivot element is the (1,1) entry which is 5 so we divide that row by 5. Now we make the (2,1) entry 0 by add the new first row with the second. The entry in the (3,1) position is already 0 so we don't have to do anything. The objective row is modified by multiplying the new first row by 5 and adding.

	x_1	x_2	x_3	x_4	x_5	x_6	z	
x_1	1	0	-$\frac{8}{5}$	$\frac{1}{5}$	-$\frac{2}{5}$	0	0	8
x_2	0	1	$\frac{12}{5}$	$\frac{1}{5}$	$\frac{3}{5}$	0	0	18
x_6	0	0	13	0	2	1	0	70
	0	0	1	1	1	0	1	70

As we can see from the tableau we have reached an optimal solution because all coefficients in the objective row are non-negative.

We can see that an optimal solution to the original problem is at $\vec{x} = (8, 18, 0)$

whose z value is 70.

We now look at an example where the objective function if unbounded.

Example

$$\text{maximize} \quad 2x_2 + x_3$$

subject to

$$x_1 + x_2 - 2x_3 \leq 7 \quad -3x_1 + x_2 + 2x_3 \leq 3, \quad x_i \geq 0$$

We add our slack variables x_4, x_5 to get the equality constraints

$$x_1 + x_2 - 2x_3 + x_4 = 7, \quad -3x_1 + x_2 + 2x_3 + x_5 = 3, \quad x_i \geq 0, i = 1, \ldots, 5$$

Because $b_i \geq 0$ we can begin with the basic feasible solution $(0, 0, 0, 7, 3)$ with a z value of 0.

In the following we give the sequence of tableaus indicating the pivot element in red.

	x_1	x_2	x_3	x_4	x_5	z	
x_4	1	1	-2	1	0	0	7
x_5	-3	1	2	0	1	0	3
	0	-2	-1	0	0	1	0
x_4	4	0	-4	1	-1	0	4
x_2	-3	1	2	0	1	0	3
	-6	0	3	0	2	0	6
x_1	1	0	-1	$\frac{1}{4}$	$-\frac{1}{4}$	0	1
x_2	0	1	-1	$\frac{3}{4}$	$\frac{1}{4}$	0	6
	0	0	-3	$\frac{3}{2}$	$\frac{1}{2}$	1	12

Now we still have a negative coefficient in the objective function for x_3 so for the next step we would typically take this for the entering variable. However, if we try to determine the departing variable we see that the objective function is unbounded below because $x_2 = 6 + x_3 \geq 0$ and $x_1 = 1 + x_3 \geq 0$ puts no restriction on x_3. This is reflected in the fact that the third column has all negative entries.

Will the Simplex Method always find an optimal solution if we have a bounded feasible set?

The answer is "no". There are some pathological examples where the method cycles back to the same basic solution which is suboptimal. This can be fixed by making

Bland's Rule. The Simplex Method will not cycle if we choose the entering and departing variables by the smallest subscript rule.

Exercise Manually go through the steps of the Simplex Method for the following LP problem. Write out the tableau at each step and mark the pivot element. Remember that we solved this problem graphically and got the optimal solution (0.5,1) with a value of 2.5.

Maximize $x_1 + 2x_2$

subject to the constraints

$$x_1 + x_2 \leq 1.5, \quad x_1 \leq 1, \quad x_2 \leq 1, \quad x_1, x_2 \geq 0$$

Quadratic Programming

A Quadratic Programming problem is a specialized optimization problem where the unknown appears quadratically in the object function but the constraints are linear.

A general Quadratic Programming problem is given a vector $\vec{c}$ and a symmetric matrix Q, find a vector $\vec{x}$ which

$$\text{maximizes} \quad z = \frac{1}{2}\vec{x}^T Q\vec{x} + \vec{c}^T\vec{x}$$

subject to the constraints of the form

$$A\vec{x} \leq \vec{b}$$

and/or

$$B\vec{x} = \vec{d}$$

Note that if Q is the zero matrix then the problem reduces to our Linear Pro-

gramming problem.

There are specialized techniques for solving this type of problem but we don't have time to go into them.

Dynamic Programming

We now look at a technique called Dynamic Programming which can be used to solve some discrete linear optimization problems such as the Knapsack Problem or Traveling Salesman Problem . Basically we break our problem into smaller subproblems to solve and then use these to solve the original problem.

The general idea behind Dynamic Programming is to iteratively find the optimal solution to a small part of the entire problem. Using the previous solution, then enlarge the problem slightly and find the new optimum solution. We continue enlarging until we have solved the entire problem. Then we trace back to find the optimal solution.

Dynamic Programming Solution to the Coin Changing Problem

To try and understand how Dynamic Programming works we first look at a simple problem which we have encountered before.

We consider the Coin Changing Problem where we want to make change for n cents using coins of denominations $d_1, d_2, \ldots, d_k$.

We first look at some properties of the problem which allow us to use Dynamic Programming .

First assume that we have found the optimal solution to the Coin Changing problem for n cents (by enumeration or a greedy approach, etc.) which uses, for example, the coins

$$d_1 \quad d_1 \quad d_3 \quad d_2 \quad d_4 \quad d_5 \quad d_2$$

Now suppose we split the coins randomly into two stacks

Stack 1 totaling b cents: $\quad d_1 \quad d_1 \quad d_3 \quad d_2$

Stack 2 totaling $n - b$ cents : $\quad d_4 \quad d_5 \quad d_2$

We claim that Stack 1 must be an optimal way of making change for b cents and Stack 2 is an optimal way of making change for $n - b$ cents. Why?

To see this assume that there is another way to make change for b cents than the way given in Stack 1. Then we could replace the coins in Stack 1 with this choice of coins and we would have a better solution than the original one. But we assumed the original one was optimal so we have a contradiction.

We say that the Coin Changing Problem exhibits optimal substructure.

The next property that the Coin Changing Problem exhibits is that we can recursively define the value of the optimal solution.

Let $C(n)$ denote the minimum number of coins of denominations d_i, $i = 1, \ldots, k$ needed to make change for n cents. For example, we know that $C(6) = 2$, $C(21) = 3$, etc using standard coins.

In the optimal solution to making change for n cents there must exist some first coin d_i where $d_i \leq n$. Furthermore, we have shown above that the remaining coins in the optimal solution must themselves be the optimal solution to making change for $n - d_i$ cents.

Thus if d_i is the first coin in the optimal solution then

$$C(n) = 1 + C(n - d_i)$$

that is, the optimal solution is one d_i coin plus the number $C(n - d_i)$ of coins to optimally make change for $n - d_i$ cents.

However, we don't know which coin d_i is the first coin in the optimal solution to making change for n cents, just that there is one. However, we can check all k such possibilities where $d_i \leq n$ and the value of the optimal solution must correspond to the minimum value of $1 + C(n - d_i)$.

Thus we need a recursion formula which exhibits this as well as works when we are making change for 0 cents. We have

$$C(n) = \begin{cases} 0 & \text{if } n = 0 \\ \min\limits_{i \text{ for } d_i \leq n} \left[1 + C(n - d_i)\right] & \text{if } n > 0 \end{cases}$$

For a Dynamic Programming approach to this problem, we will compute the value of the optimal solution in a bottom-up manner. To this end, we will compute $C(p)$ for $0 \leq p \leq n$ so that $C(p)$ contains the correct minimum number of coins needed to make change for p cents. We will also let $S(p)$ denote the index of the first coin in an optimal solution to making change for p cents. After we do this, then we will demonstrate how to construct the optimal solution from this information.

Example Suppose we want to make change for 16 cents using quarters (d_1), dimes (d_2), nickels (d_3) and pennies (d_4). From our previous work with a greedy approach we know that the optimal solution is

0 quarters, 1 dimes, 1 nickel and 1 penny

First we want to compute $C(p)$ for $0 \leq p \leq 16$ where $C(p)$ contains the correct minimum number of coins needed to make change for p cents.

We set $C(0) = 0$. Then for $p = 1, 2, \ldots, 16$ we have the following where we have given the justification in some cases; the others are analogous.

For $p = 1$ we have $C(1) = 1$, $S(1) = d_4$ because d_4 is the only choice ≤ 1

For $p = 2$ we have $C(2) = 2$, $S(2) = d_4$ because d_4 is the only choice ≤ 2

For $p = 3$ we have $C(3) = 3$, $S(3) = d_4$ because d_4 is the only choice ≤ 3

For $p = 4$ we have $C(4) = 4$, $S(4) = d_4$ because d_4 is the only choice ≤ 4

For $p = 5$ we have $C(5) = 1$, $S(5) = d_3$ because now we have two coins which are ≤ 5;

$$d_4 \leq 5 \implies 1 + C(5 - 1) = 1 + C(4) = 5$$

and

$$d_3 \leq 5 \implies 1 + C(5 - 5) = 1 + C(0) = 1 < 5 \quad \text{so we take } d_3$$

For $p = 6$ we have $C(6) = 2$, $S(6) = d_3$

For $p = 7$ we have $C(7) = 3$, $S(7) = d_3$

For $p = 8$ we have $C(8) = 4$, $S(8) = d_3$

For $p = 9$ we have $C(9) = 5$, $S(9) = d_3$

For $p = 10$ we have $C(10) = 1$, $S(10) = d_2$ because now we have three coins which are ≤ 10;

$$d_4 \leq 10 \implies 1 + C(9) = 6, \quad d_3 \leq 10 \implies 1 + C(5) = 2$$

and

$$d_2 \leq 10 \implies 1 + C(0) = 1 < 2 < 6 \quad \text{so we take } d_2$$

For $p = 11$ we have $C(11) = 2$, $S(11) = d_2$

For $p = 12$ we have $C(12) = 3$, $S(11) = d_2$

For $p = 13$ we have $C(13) = 4$, $S(11) = d_2$

For $p = 14$ we have $C(14) = 5$, $S(11) = d_2$

For $p = 15$ we have $C(15) = 2$, $S(11) = d_2$

For $p = 16$ we have $C(16) = 3$, $S(16) = d_2$

Now we want to see how to construct the optimal solution (i.e., give the list of coins needed) from our computed information.

Our first coin is $S(16)$ which is d_2 a dime.

Now we take the subproblem of finding the optimal number of coins to make change for $n - d_2$ cents; thus we find $S(n - d_2)$ which in our case is $S(16 - 10) = S(6) = d_3$ so our second coin is a nickel.

Now we take the next subproblem of making change for $n - d_2 - d_3$ cents so we need $S(n - d_2 - d_3) = S(1) = d_4$ which is a penny. We are done because $n - d_2 - d_3 - d_4 = 0$ so our solution is

0 quarters, 1 dimes, 1 nickel and 1 penny

The above procedure correctly outputs an optimal set of coins for making change for 16 cents.

How would we implement a code for using Dynamic Programming to calculate the change problem?

Input: n the cents required to make change for; the k denominations $d(i)$, $i = 1, \ldots, k$

Output: The minimum number of coins required to make change for n cents given in $C(n)$ and a list of the coins used

Set $C(0) = 0$

For $p = 1, n$

set `min` $\leftarrow \infty$

for $i = 1 : 4$

if $d(i) \leq p$ then

```
    if 1 + C(n − d(i)) < min then
        min = 1 + C(n − d(i))
        coin = i
    end if
end if
C(p) =min
        S(p) = coin
% print out coins used
while n > 0
        Print S(n)
        n ← n − d(S(n))
```

Characteristics of a problem that can be solved using Dynamic Programming

- the problem can be divided into stages;
- each stage has one or more states;
- you can make a decision at each stage;
- the decision you make affects the state for the next stage;
- there is a recursive relationship between the value of the decision at the stage and the previously found optimal solution.

In our Coin Changing Problem the stages are coins added; the states are the feasible coins which can be added; the decision to be made is which coin to add; the effect on the next stage is the reduction in the amount of cents you need to make change for; and finally the recursive relationship is $1 + C(n - d(i))$ at the ith stage.

Example Suppose a corporation is willing to invest \$5M into its three plants in order to increase revenue. Each plant submits up to 3 proposals giving the cost and expected revenue. Use Dynamic Programming to determine how the

corporation should invest its money to maximize revenue if the proposal information is given in the table below. The values in the table are given in millions. Assume that a proposal is completely funded which implies the amount to each plant is an integer number (in millions of dollars). Assume further that only 0 or 1 proposal per plant is funded; we have included the first proposal as a no cost-no return proposal which essentially means that if proposal # 1 is funded that plant gets no money; this will allow us to handle the case where a plant gets no funding. Assume that the corporation will invest a total of $5 M.

	Plant	Cost	Revenue	Plant	Cost	Revenue	Plant	Cost	Revenue
Proposal									
# 1	1	0	0	2	0	0	3	0	0
# 2	1	1	5	2	2	8	3	1	4
# 3	1	2	6	2	3	9	3	-	-
# 4	1	-	-	2	4	12	3	-	-

We will let the stages be the allocation to each plant; assume that stage #1 is the allocation to Plant 1, stage # 2 is the allocation to Plant 2, etc. We set this ordering arbitrarily.

Each stage is divided into states. A state encompasses the information required to go from one stage to the next. Let x_i, $i = 1, 2, 3$ denote the number of millions allocated at stage $\#i$. In this case the states for stages 1, 2, and 3 are

$\{0, 1, 2, 3, 4, 5\}$ the amount of money spent on Plant # 1

$\{0, 1, 2, 3, 4, 5\}$ the amount of money spent on Plant # 1 and Plant # 2

$\{5\}$ the amount of money spent on the three plants

Associated with each state is the revenue which is used to make our decision. At the third and last stage it is easy to know how much to allocate to Plant # 3 because it is just 5M minus the amount spent on Plants #1 and # 2. However, for the other two stages we must use the revenue to make a decision.

For the first stage (i.e., determining how much to allocate to Plant # 1 we have the following possibilities. For example, if the plant is allocated 1M, then only Proposal #2 can be awarded with a revenue of 5M. If $\geq$ 2M is allocated than all that can be done is fund Proposal # 3 for a return of 6M.

Investment	Proposal No.	Revenue
0M	1	0M
1M	2	5M
2M	3	6M
3M	3	6M
4M	3	6M
5M	3	6M

For the second stage of the computations, we must find the optimal solution for both Plant # 1 and # 2. Recall that x_2 denotes the amount allocated to the two plants.

The results are presented in the table below but let's look at how we get the values for say $x_2 = 4$, i.e., we are allocating a total of 4M to Plants # 1 and # 2.

When Proposal # 1 is funded for Plant # 2 the cost is 0M and the revenue is 0M; then Plant # 1 has 4M so Proposal # 3 is funded for a revenue of 6M so total revenue is 0M + 6M = 6M for this option.

When Proposal # 2 is funded for Plant # 2 the cost is 2M and the revenue is 8M; then Plant # 1 has 2M so Proposal # 3 is funded for a revenue of 6M so total revenue is 8M + 6M = 14M for this option.

When Proposal # 3 is funded for Plant # 2 the cost is 3M and the revenue is 9M; then Plant # 1 has 1M so Proposal # 2 is funded for a revenue of 5M so total revenue is 9M + 5M = 14M for this option.

When Proposal # 4 is funded for Plant #2 the cost is 4M and the revenue is 12M; then Plant # 1 has 0M so Proposal # 1 is funded for a revenue of 0M so total revenue is 0M + 12M = 12M for this option.

This says that when 4M is allocated to the first two plants the optimal is to fund Proposal # 3 for Plant # 1 and Proposal # 2 for Plant # 2 OR fund Proposal # 2 for Plant # 1 and Proposal # 3 for Plant # 2; both yield a revenue of 14M

We summarize the results for allocations of 0M to 5M for Plants # 1 and # 2 below.

Investment	Optimal Proposal Plant # 2	Optimal Proposal Plant # 1	Total Revenue
0M	1	1	0M
1M	1	2	5M
2M	2	1	8M
3M	2	1	13M
4M	2 or 3	3 or 2	14M
5M	4	2	17M

Now at the final stage we know that we can just allocate 5M minus the amount allocated to Plants 1 and 2. Because Plant # 3 only has 1 proposal at a cost of 1M we need to compare the revenue for the last two cases in the table above.

If we allocate 4M to Plants #1 and # 2 then we have 1M left for Plant #3 for a total revenue of 14M + 4M = 18M. If we allocate 0M to Plant # 3 and 5M to the first two plants then the revenue is 17M.

Therefore the maximum revenue is attained when we fund Proposal # 2 for Plant # 3 and either of the two configurations in the table above for Plants # 1& 2.

Dynamic programming: Needleman-Wunsch sequence alignment

Scoring matrix

	-	A	C	G	T
-	-5	-5	-5	-5	-5
A	-5	10	-1	-3	-4
C	-5	-1	7	-5	-3
G	-5	-3	-5	9	0
T	-5	-4	-3	0	8

Sequence 1 = CGAGACGT
Sequence 2 = AGACTAGTTAC

Construction of the Table (d=gap penalty)

```
for i=0 to length(A)
  F(i,0) ← d*i
for j=0 to length(B)
  F(0,j) ← d*j
for i=1 to length(A)
  for j=1 to length(B)
  {
    Match ← F(i-1,j-1) + S(A_i, B_j)
    Delete ← F(i-1, j) + d
    Insert ← F(i, j-1) + d
    F(i,j) ← max(Match, Insert, Delete)
  }
```

		C	G	A	G	A	C	G	T
	0	-5	-10	-15	-20	-25	-30	-35
A	-5	9	4	-1	-6	-11	-16	-21
G	-10	4	19	14	9	4	-1	-6
A	-15	-1	14	14	13	16	11	6
C	-20	-6	9	14	10	11	16	20
T	-25	-11	4	9	24	19	14	15
A	-30	-16	-1	13	19	19	28	23
G	-35	-21	-6	8	14	16	23	37
T	-40	-26	-11	3	9	11	18	32
T	-45	-31	-16	-2	13	8	13	27
A	-50	-36	-21	-7	8	20	15	22
C								

Backtracking

```
AlignmentA ← ""
AlignmentB ← ""
i ← length(A)
j ← length(B)
while (i > 0 or j > 0)
{
  if (i > 0 and j > 0 and F(i,j) == F(i-1,j-1) + S(A_i, B_j))
  {
    AlignmentA ← A_i + AlignmentA
    AlignmentB ← B_j + AlignmentB
    i ← i - 1
    j ← j - 1
  }
  else if (i > 0 and F(i,j) == F(i-1,j) + d)
  {
    AlignmentA ← A_i + AlignmentA
    AlignmentB ← "-" + AlignmentB
    i ← i - 1
  }
  else (j > 0 and F(i,j) == F(i,j-1) + d)
  {
    AlignmentA ← "-" + AlignmentA
    AlignmentB ← B_j + AlignmentB
    j ← j - 1
  }
}
```

AGACTA-GTTAC
CGA-GACG-T--

	-	C	G	A	G	A	C	G	T
-	0	-5	-10	-15	-20	-25	-30	-35	-40
A	-5	-1	-6	0	-5	-10	-15	-20	-25
G	-10	-6	8	3	9	4	-1	-6	-11
A	-15	-11	3	18	13	19	14	9	4
C	-20	-8	-2	13	13	14	26	21	16
T	-25	-13	-7	8	13	9	21	26	30
A	-30	-18	-12	3	8	23	18	21	25
G	-35	-23	-9	-2	12	18	18	27	22
T	-40	-28	-14	-7	7	13	15	22	36
T	-45	-33	-19	-12	2	8	10	17	31
A	-50	-38	-24	-9	-3	12	7	12	26
C	-55	-43	-29	-14	-8	7	19	14	21

backtracking: (9 12) (9 11) (9 10) (8 9) (8 8) (7 7) (6 7) (5 6) (4 5) (4 4) (3 3) (2 2)

Simulated Annealing

Simulated Annealing is a technique for finding a global extremum which has its basis in the physics of materials. It can be viewed as a generalization of Monte Carlo.

The term "annealing" is used in metallurgy. If a solid material is heated with sufficiently high temperatures past its melting point and then cooled down to a solid state, the structural properties of the cooled solid depend on the rate of cooling. Slow cooling leads to strong, large crystals while fast cooling results in imperfections.

At the end of the liquid to solid transition the crystalline structure has the lowest energy. The metal "finds" this state if the temperature is lowered sufficiently slowly and the original heating was high enough.

At a finite temperature, the system has a small probability to be in a highly

excited state. This prevents it from becoming trapped in a local minimum.

Simulated annealing simulates the slow cooling process to find the global minimum.

In a minimization approach we are tempted to immediately go downhill as far as we can go. However, this often leads to a local, but not necessarily a global, minimum. Natures own minimization algorithm is based on quite a different procedure. The so-called Boltzmann probability distribution,

$$\text{prob}(E) = e^{E/kT}$$

expresses the idea that a system in thermal equilibrium at temperature T has its energy probabilistically distributed among all different energy states E. Even at low temperature, there is a chance, albeit very small, of a system being in a high energy state. Therefore, there is a corresponding chance for the system to get out of a local energy minimum in favor of finding a better, more global, one. The quantity k (Boltzmanns constant) is a constant of nature that relates temperature to energy. In other words, the system sometimes goes uphill as well as downhill; but the lower the temperature, the less likely is any significant uphill excursion.

So our goal is to describe an algorithm which emulates this natural process.

General steps:

Step 1: Initialize Start with a random initial placement. Initialize a very high "temperature".

Step 2: Move – Perturb the placement through a defined move.

Step 3: Calculate score – calculate the change in the score due to the move made.

Step 4: Choose – Depending on the change in score, accept or reject the move. The probability of acceptance depending on the current "temperature".

Step 5: Update and repeat – Update the temperature value by lowering the temperature. Go back to Step 2.

The process is done until a "Freezing Point" is reached.

Whether we accept or reject the move is usually based on the following criteria.

Assume we start at a given state S_0 and compute the energy (or cost) $E(S_0)$. Then we choose a slightly different state S_1 and compute its energy $E(S_1)$.

If $E(S_1) < E(S_0)$ then we accept the new state S_1.

If $E(S_1) \geq E(S_0)$ then we compute the transition probability

$$P = e^{(E(S_0)-E(S_1))/T}$$

Next we compute a random number p uniformly distributed in [0,1]. If $p < P$ switch to the new state S_1; otherwise remain in state S_0.

We will apply the method to both the Knapsack and the Traveling Salesman Problem.

Branch and Bound Algorithms

When you studied Graph Theory you talked about the Depth-First Search of a tree and mentioned the Breadth-First approach. The Branch and Bound approach is based on a Breadth-First approach using pruning.

The basic idea is to try to make a decision if at a node the remainder of that portion of the tree can be pruned, i.e., we don't have to search that part.

Branch and Bound algorithms can be viewed as using a Divide and Conquer strategy.

We have seen that a Brute Force approach to many combinatorial optimization problems is to enumerate all possibilities. In the Branch and Bound approach there is a systematic enumeration of all candidate solutions but large subsets of fruitless candidates are discarded at once, by using upper and lower estimated bounds of the quantity being optimized and comparing these to the current best

solution. Thus it is typically an improvement over complete enumeration. Of course, the method relies on the ability to estimate upper and lower bounds.

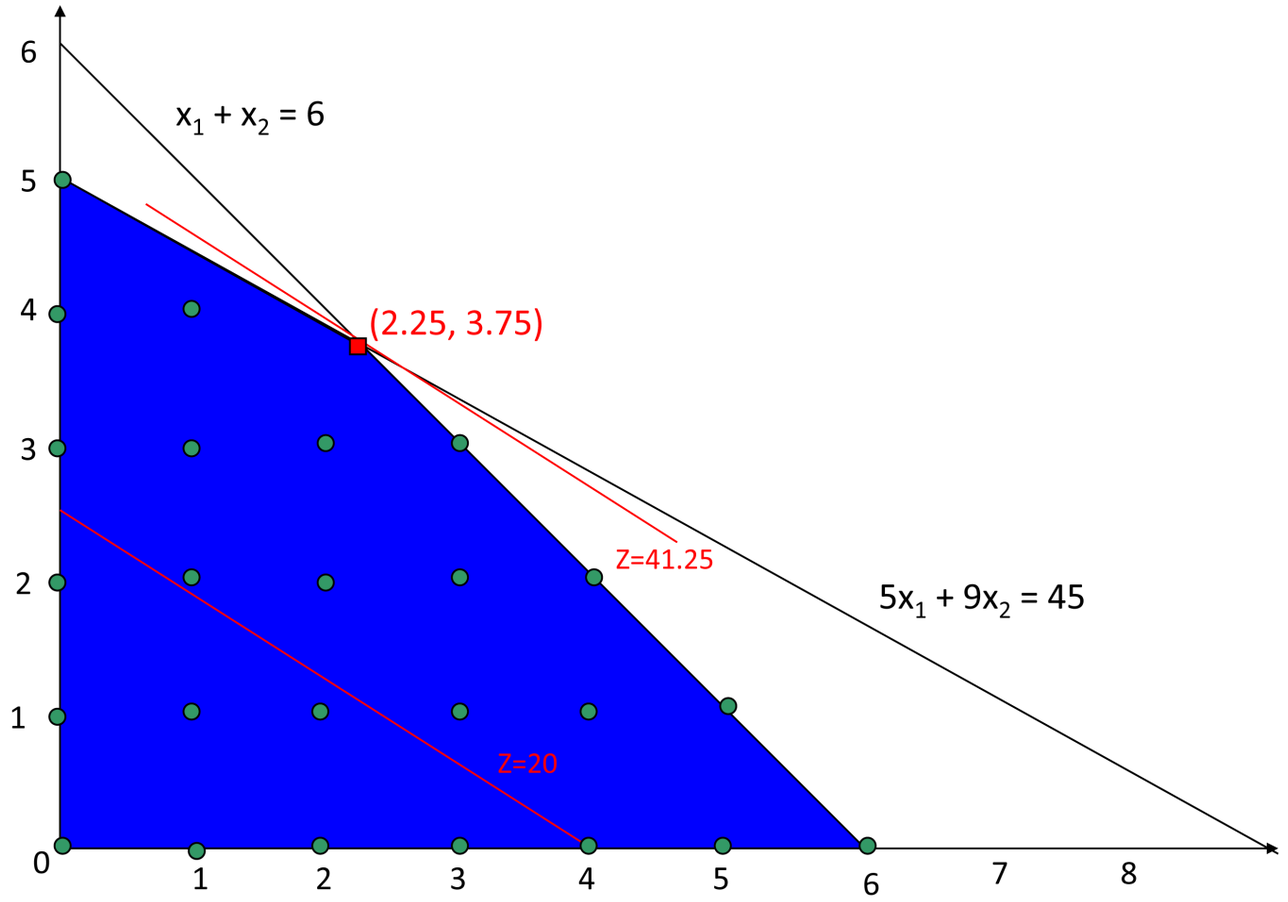
Example Consider the integer programming problem to

$$\text{maximize} \quad z = 5x_1 + 8x_2$$

subject to

$$x_1 + x_2 \leq 6 \quad 5x + 9x_2 \leq 45$$

where $x_1, x_2 \geq 0$ and are integers.



We could solve this by enumerating all possible solutions in our feasible region, i.e.,

$$\{(0,0),(1,0),\cdots(6,0),(0,1),(1,1),\cdots,(5,1),\cdots,(0,4),(1,4),(0,5)\}$$

and then evaluating z there. However, we want to use a Branch and Bound

approach which requires less work than complete enumeration.

Now if we don't require that x_1, x_2 are integers we can find the solution via Linear Programming by checking the vertices of the polyhedron which are (0,0), (6,0), (0,5) and (2.25, 3.75). When we do this, we find that the vertex (2.25,3.75) gives the maximum value of z which is 41.25. So we know that this is an upper bound for the solution to our integer problem. We want to use this to break the problem into two subproblems, i.e., branch.

We will divide the problem into two subproblems using x_2 where we take $x_2 \leq 3$ and $x_2 \geq 4$. So we have the following integer programming problems:

$$S_1: \qquad \text{maximize} \quad z = 5x_1 + 8x_2$$

subject to

$$x_1 + x_2 \leq 6 \quad 5x + 9x_2 \leq 45$$

where $x_1, x_2 \geq 0$, $x_2 \leq 3$ and x_1, x_2 are integers.

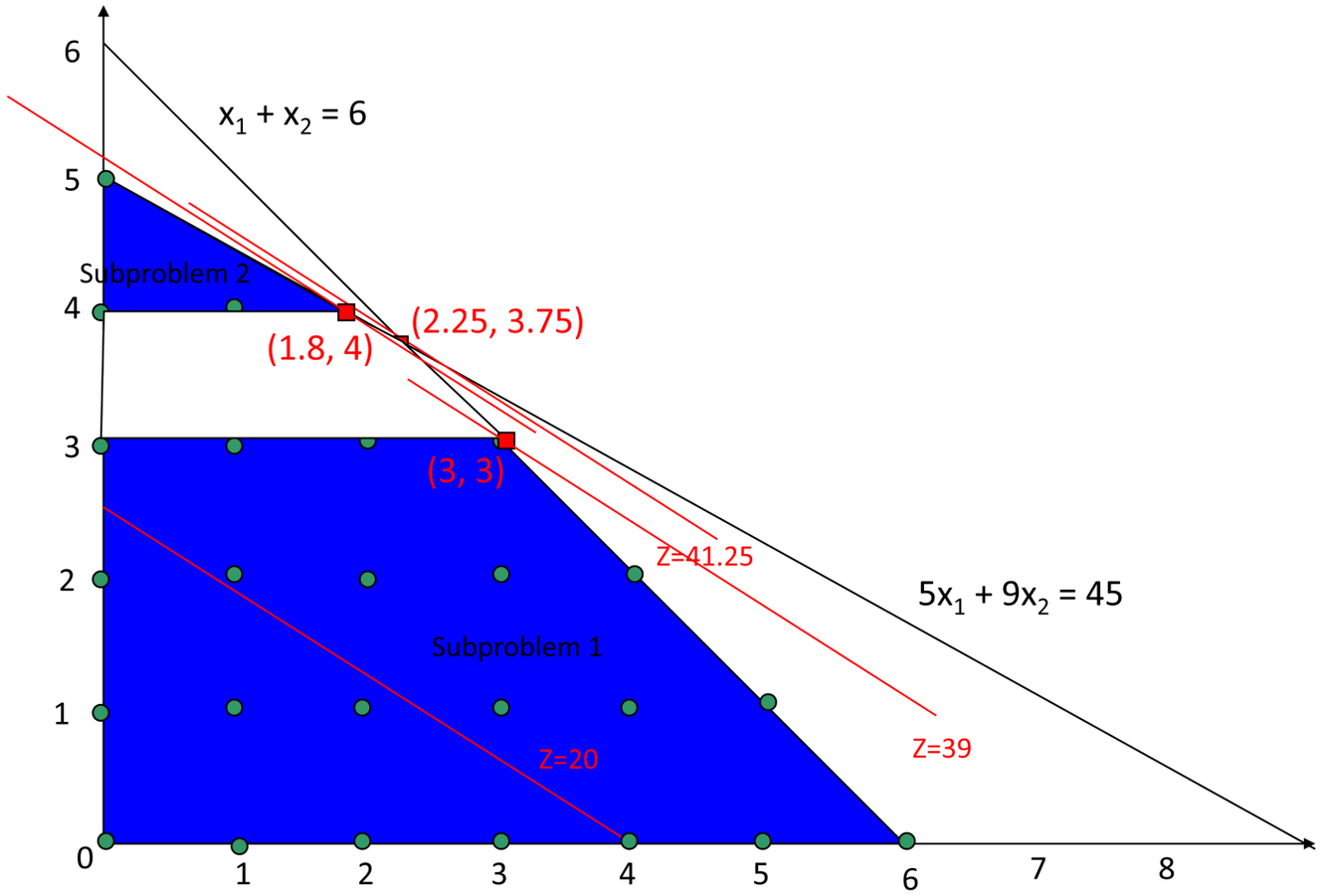
$$S_2: \qquad \text{maximize} \quad z = 5x_1 + 8x_2$$

subject to

$$x_1 + x_2 \leq 6 \quad 5x + 9x_2 \leq 45$$

where $x_1, x_2 \geq 0$, $x_2 \geq 4$ and x_1, x_2 are integers.

This is shown graphically in the next figure. Note that (2.25,3.75) is no longer in either problem.



So we have started with the problem which allowed all values of x_1, x_2 (integer and not) and found that the maximum occurred at (2.25,3.75) with $z = 41.25$. We then branched into two child nodes, one for S_1 where $x_2 \leq 3$ and for S_2 where $x_2 \geq 4$. Now for S_1 the LP approach has us evaluate z at $(0, 0)$, (6,0),

(0,3) and (3,3) and we find the maximum value at (3,3) where $z = 39$. Now (3,3) satisfies the integer constraint and is the optimal solution so there is no need to further divide S_1 so we simply prune the tree at this point. This is the best integer solution stored so far so it becomes our lower bound.

Now for S_2 the maximum is not where x_1, x_2 are integers so we have to subdivide this problem further. We don't have an integer solution from S_2 so we don't update our current value of $z = 39$.

Now we subdivide S_2 into two problems S_3 and S_4. We use x_1 where we have the problems

$$S_3 : \qquad \text{maximize} \quad z = 5x_1 + 8x_2$$

subject to

$$x_1 + x_2 \leq 6 \quad 5x + 9x_2 \leq 45$$

where $x_1, x_2 \geq 0$, $x_1 \leq 1$ and x_1, x_2 are integers.

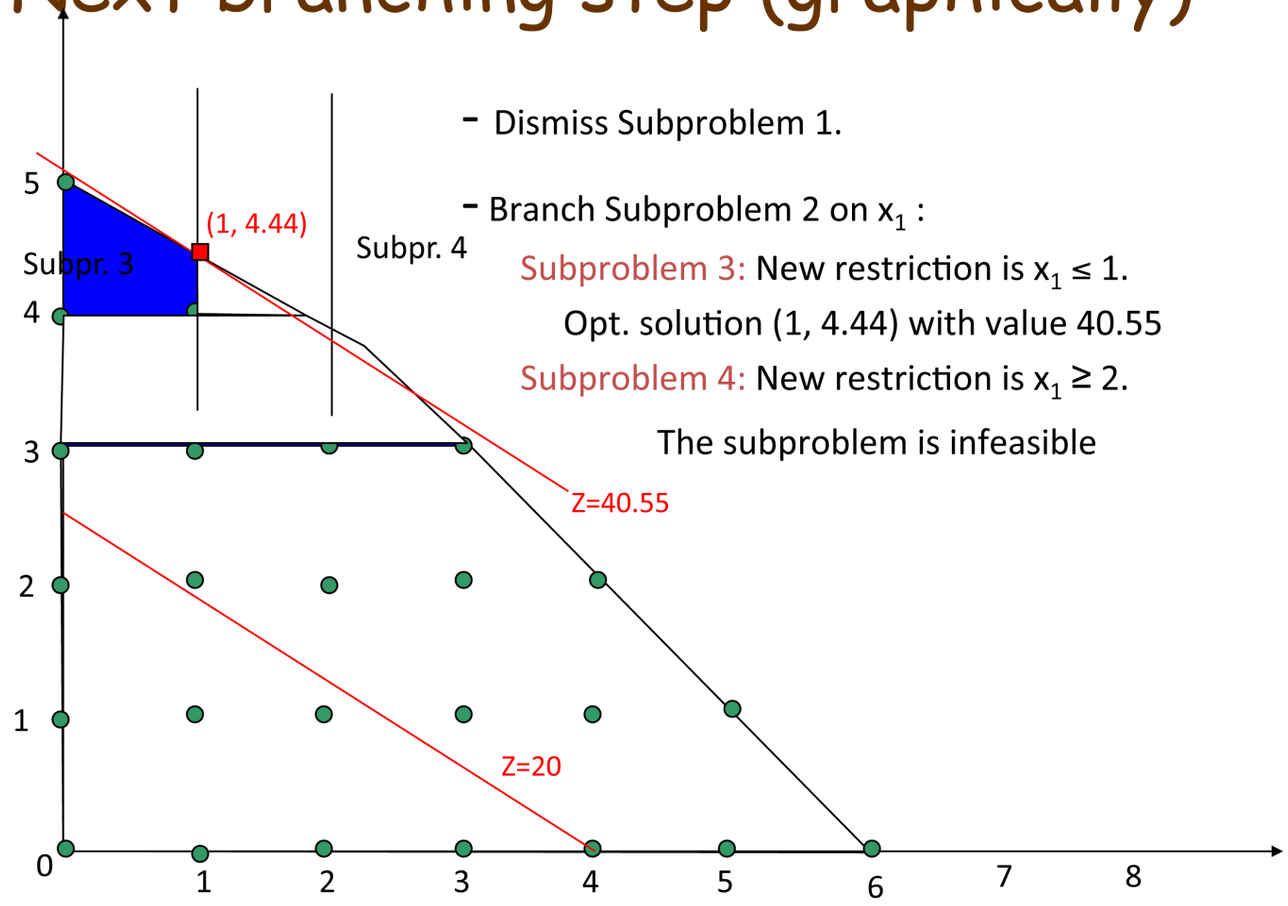
$$S_4 : \qquad \text{maximize} \quad z = 5x_1 + 8x_2$$

subject to

$$x_1 + x_2 \leq 6 \quad 5x + 9x_2 \leq 45$$

where $x_1, x_2 \geq 0$, $x_1 \geq 2$ and x_1, x_2 are integers.

Next branching step (graphically)



- Dismiss Subproblem 1.
- Branch Subproblem 2 on x_1 :

  Subproblem 3: New restriction is $x_1 \leq 1$.

  Opt. solution (1, 4.44) with value 40.55

  Subproblem 4: New restriction is $x_1 \geq 2$.

  The subproblem is infeasible

For S_3 we have the maximum occurs at (1,4.44) where $z = 40.55$ which is larger

than our stored best integer solution of 39 so we can't dismiss the remainder of this tree.

Now for S_4 we see that there are actually no integer values in this problem so the subproblem is not feasible.

Lastly we must subdivide problem S_3 into two problems S_5 and S_6. We choose x_2 and break into the two problems

$$S_5 : \qquad \text{maximize} \quad z = 5x_1 + 8x_2$$

subject to

$$x_1 + x_2 \leq 6 \quad 5x + 9x_2 \leq 45$$

where $x_1, x_2 \geq 0$, $x_2 \leq 4$ and x_1, x_2 are integers.

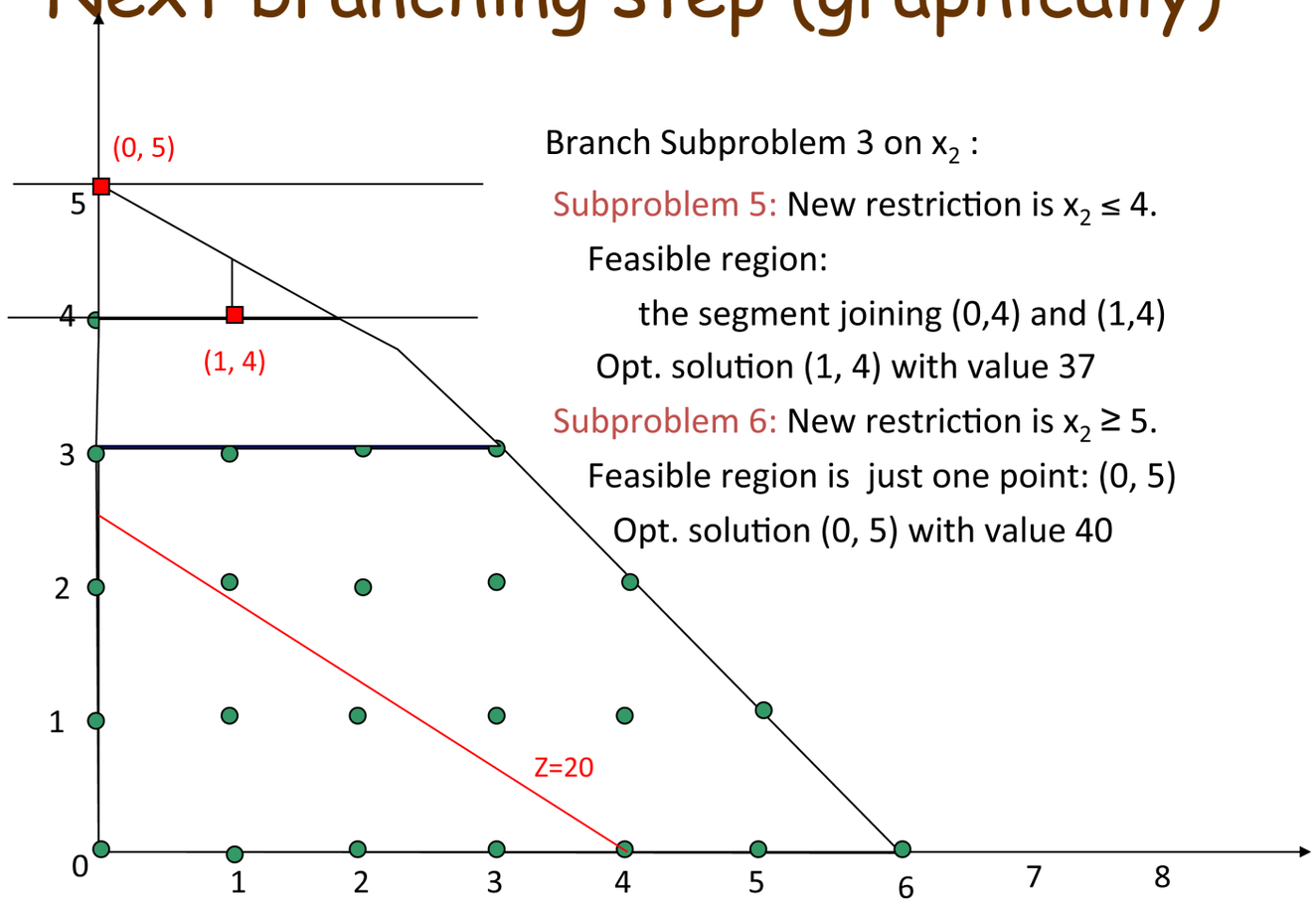
$$S_6 : \qquad \text{maximize} \quad z = 5x_1 + 8x_2$$

subject to

$$x_1 + x_2 \leq 6 \quad 5x + 9x_2 \leq 45$$

where $x_1, x_2 \geq 0$, $x_2 \geq 5$ and x_1, x_2 are integers.

Next branching step (graphically)



Branch Subproblem 3 on x_2 :

Subproblem 5: New restriction is $x_2 \leq 4$.

Feasible region:

the segment joining (0,4) and (1,4)

Opt. solution (1, 4) with value 37

Subproblem 6: New restriction is $x_2 \geq 5$.

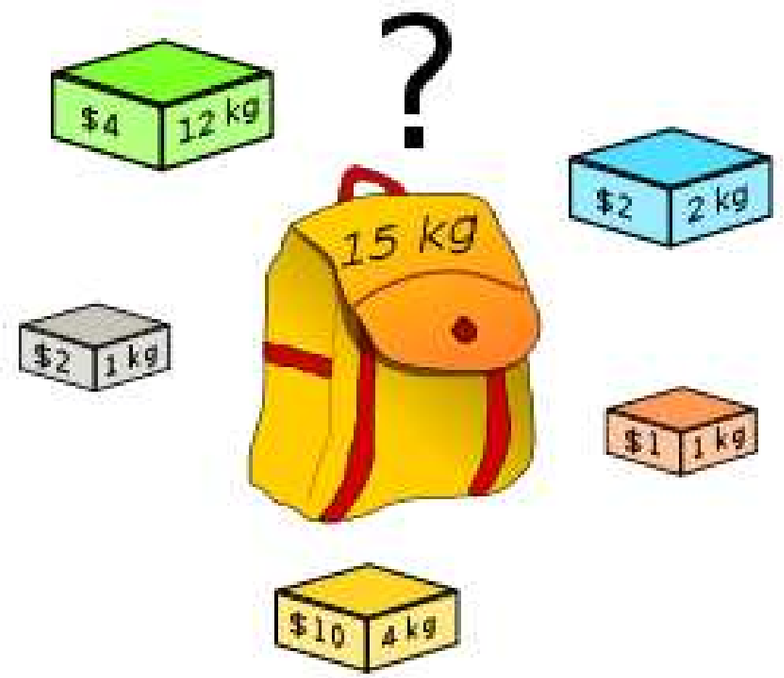
Feasible region is just one point: (0, 5)

Opt. solution (0, 5) with value 40

For S_5 we have the optimal solution is at (1,4) (only feasible points are (0,4) and (1,4)) so we have a z value of 37 which is less than our stored value so it is not optimal.

For S_6 we have only one point (0,5) and its z value is 40 which is larger than our stored value so it is optimal.

The Knapsack Problem



The standard example of a Knapsack Problem is to pack a knapsack which can contain a prescribed weight with objects such that the comfort or value (or some other attribute) of the objects is maximized.

However, there are other examples of problems which are essentially Knapsack

problems. For example, we could have n projects. We assume the jth project, has a cost of c_j and a value of v_j . Each project can either be done or not, i.e., it is not possible to do only a fraction of a project. There is also a budget b available to fund the projects. The Knapsack Problem is to choose a subset of the projects to maximize the sum of the total values while not exceeding the budget constraint.

These problems can be formulated as a discrete optimization problem. We will refer to the following problem as the 0/1 Knapsack Problem because the components of the unknown $\vec{x}$ can only be zero or one.

Let

$$x_i = \begin{cases} 1 & \text{if the object is used} \\ 0 & \text{if the object is not used} \end{cases}$$

then if each object has a weight w_i and a value v_i then we want to

$$\text{maximize} \quad \sum_{i=1}^{n} x_i v_i$$

subject to the constraint

$$\sum_{i=1}^{n} w_i x_i \leq b$$

where b is the weight the knapsack can hold.

Note how this problem compares to a standard LP problem. Here the unknown $\vec{x}$ has components which are binary, i.e., they are either 0 or 1 as opposed to the LP problem where we assume the components are non-negative.

In addition, we typically only have a single constraint in the Knapsack Problem compared with the LP problem where we can have a large number of constraints.

Note also that we are restricting the problem so that we only use an item once; e.g., we can't pack more than one of the same item. There are several modifications to this 0/1 Knapsack Problem ; for example:

- Multiple-Choice Knapsack Problem where we have k classes of items N_i and we must take at least one item from each class.

- Multiple Knapsack Problem where we have n items and m knapsacks with capacities W_i.
- Fractional Knapsack problem where we relax the restriction that $x_i \in \{0, 1\}$ but instead require $0 \leq x_i \leq 1$ for all i.

Fractional Knapsack Problem

The Fractional Knapsack Problem is illustrated in the following example (Ref-Computer Programming weblog).

Ted Thief has just broken into Fort Knox! He sees himself in a room with n piles of gold dust. Because the each pile has a different purity, each pile also has a different value v_i and a different weight w_i. Ted has a knapsack that can only hold W kilograms.

Given n, v_i, w_i and W calculate which piles Ted should completely put into his knapsack and which he should put only a fraction of.

We can use a Greedy algorithm to solve the Fractional Knapsack Problem. It can be described as follows. Here we assume that the sum of all the weights are $\geq$ the capacity of the knapsack.

Input: n items with weights w_i and values v_i; capacity W

Output: an optimal solution $\vec{x}$ where $0 \leq x_i \leq 1$.

Let W_{cur} denote the current weight of the knapsack and V_{cur} the current value in the knapsack.

Preprocessing step We first sort the items in a non-increasing order based on v_i/w_i, i.e.,

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$$

Step 0 Initialize. Set $x_i = 0$, $i = 1, \ldots, n$; set $W_{\text{cur}} = 0$; set V_{cur}= 0

for $j = 1, 2, \ldots, n$

Step 1 If $w_j \leq W - W_{\text{cur}}$
then $x_j = 1$
else set $x_j = (W - W_{\text{cur}})/w_j$

Step 2 Set $W_{\text{cur}} = W_{\text{cur}} + w_j x_j$. If $W_{\text{cur}} \geq W$ terminate

Let's use the Greedy approach to solve our "thief" problem with

$$n = 5, \quad \vec{w} = (1, 12, 2, 4, 1)^T, \quad \vec{v} = (2, 4, 2, 10, 1)^T, \quad W = 15$$

Our first step is to sort the items based upon v_i/w_i. We order the items by

$$\frac{10}{4} \geq \frac{2}{1} \geq \frac{2}{2} \geq \frac{1}{1} \geq \frac{4}{12}$$

so we have the ordered values

$$\vec{w} = (4, 1, 2, 1, 12)^T, \quad \vec{v} = (10, 2, 2, 1, 4)^T$$

Set $W_{\text{cur}} = 0$, $V_{\text{cur}} = 0$

- For $j = 1$ we have $w_1 = 4 < 15 - 0$ so we set $x_1 = 1$ and $W_{\text{cur}} = 0 + 4 = 4 < 15$
- For $j = 2$ we have $w_2 = 1 < 15 - 4$ so we set $x_2 = 1$ and $W_{\text{cur}} = 4 + 1 = 5$
- For $j = 3$ we have $w_3 = 2 < 15 - 5$ so we set $x_3 = 1$ and $W_{\text{cur}} = 5 + 2 = 7$
- For $j = 4$ we have $w_4 = 1 < 15 - 7$ so we set $x_4 = 1$ and $W_{\text{cur}} = 7 + 1 = 8$

- For $j = 5$ we have $w_5 = 12 > 15 - 8$ so we set $x_5 = (W - W_{\text{cur}})/12 = 7/12$ and $W_{\text{cur}} = 8 + (7/12)(12) = 15$. We terminate because $W_{\text{cur}} = W$.

So the thief should take all of the piles except the ones that weighs 12 kg and he should only take approximately 58% of that one.

Remarks:

1. The Greedy Algorithm fills the knapsack completely because we have assumed that

$$\sum_{i=1}^{n} w_i \geq W \, .$$

2. If $\vec{x}$ is the solution to the Greedy Algorithm and if y is any other feasible solution then

$$\sum_i y_i v_i \leq \sum_i x_i v_i$$

The Greedy Algorithm described above gives an optimal solution for the Fractional Knapsack Problem

The 0/1 Knapsack Problem

We now return to the Knapsack Problem where the components of the unknown $\vec{x}$ can only be zero or one. We first look at a Brute Force approach and then move to a Branch and Bound approach.

Brute Force Approach for the 0/1 Knapsack Problem

One approach to solving a Knapsack problem is the Brute Force approach of enumerating all possible combinations; it is often called complete enumeration.

Example Use the approach of enumerating all possible solutions to the following knapsack problem.

$$\text{minimize} \quad z = -10x_1 - 7x_2 - 3x_3 - x_4$$

subject to the constraints

$$5x_1 + 6x_2 + 4x_3 + 6x_4 \leq 11$$

and

$$x_1, x_2, x_3, x_4 \in \{0, 1\}$$

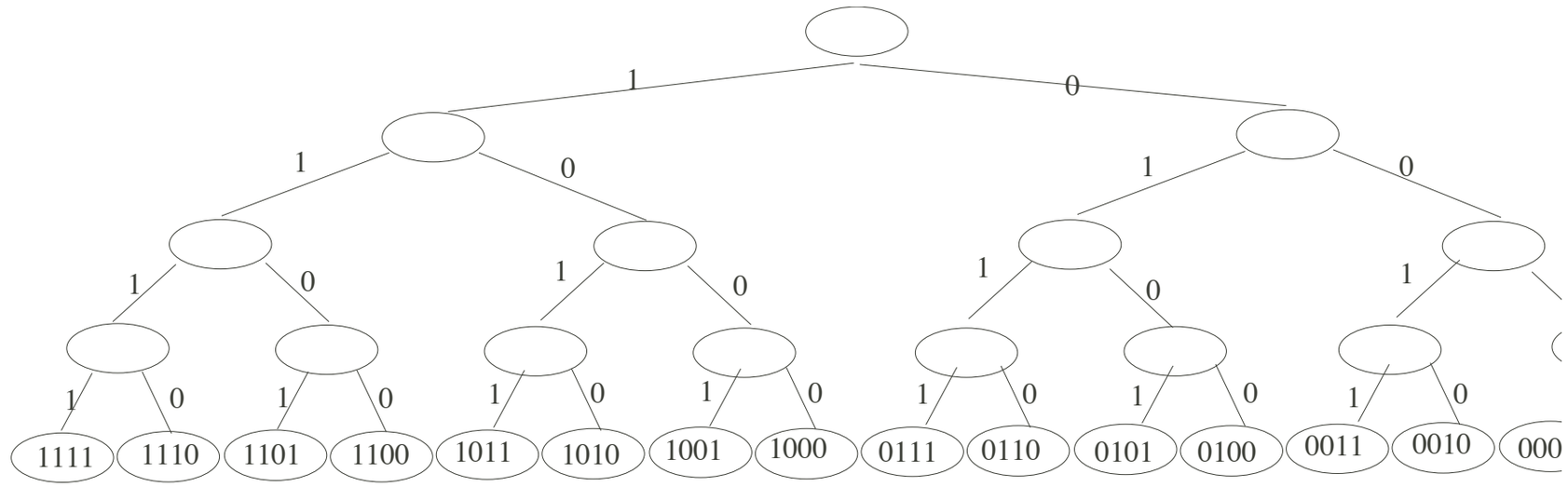
Each of the components of $\vec{x}$ can be 0 or 1 so we have a total of $2^4 = 16$ possible solutions. These are enumerated in the table below. For a shorthand notation we have indicated, e.g., 0010 to denote $x_1 = x_2 = 0$, $x_3 = 1$ and $x_4 = 0$. If the constraint is not satisfied then these are marked in blue.

$\vec{x}$	z	constraint	$\vec{x}$	z	constraint
0000	0	0	1000	-10	5
0001	-1	6	1001	-11	11
0010	-3	4	1010	-13	9
0011	-4	10	1011	-14	15
0100	-7	6	1100	-17	11
0101	-8	12	1101	-18	17
0110	-10	10	1110	-20	15
0111	-11	16	1111	-21	21

From the table we can see that there are 9 feasible solutions and the minimum occurs when $x_1 = 1$, $x_2 = 1$, $x_3 = x_4 = 0$.

To visualize how we could write a code to solve this problem we can think of generating a tree. The root node is the first variable and its two child node correspond to its values of 1 or 0. Then each of these child nodes have two choices for x_2, etc. In this approach we can view it as a graph and perform a depth-first search, for example, to visit every node.

For our example the tree is illustrated below.



One thing to note is that if you look at the leaves of the tree from the right to left then if each leaf is viewed as a binary number then the leaf to the left is found by adding one to the number of the preceding node (using binary addition).

Because we have 2^n possibilities, it can get computationally expensive to enumerate all possible choices and determine whether each is feasible.

We now want to look at a Branch and Bound algorithm which reduces the number of leaves that we have to visit so it is an improvement over the Complete Enumeration approach.

Branch and Bound Approach

When you studied Graph Theory you talked about the Depth-First Search of a tree and mentioned the Breadth-First approach. The Branch and Bound approach is based on a Breadth-First approach using pruning.

The basic idea is to try to make a decision if at a node the remainder of that portion of the tree can be pruned, i.e., we don't have to search that part.

Suppose we are at the first node and we partition the solution set into subsets S_1 and S_2 by branching with respect to the value chosen for x_1; assume each solution in S_1 has $x_1 = 1$ and each solution in S_2 to have $x_1 = 0$.

Each subset S_i, $i = 1, 2$ is a smaller Knapsack Problem . Now we divide each subset S_i, $i = 1, 2$ into two subsets based on whether x_2 is zero or one.

Suppose that we have an effective method to compute a lower bound for any

feasible solution that has the variables up to the point fixed as either zero or one. If our lower bound is not smaller (assuming we are maximizing) than our current best solution that we have found so far, then we do not have to visit any of the child nodes of the current node. The whole branch of the tree can be pruned and we expect that we have reduced our search.

Of course, this relies on having an effective method to compute a lower bound for any feasible solution for which the first k components set as zero or one.

How can we find a lower bound on the outcome of the Knapsack Problem at the kth step?

We can get an upper bound on the outcome of the knapsack problem at the kth level by using our Greedy Algorithm for the Fractional Knapsack Problem . We obtain our lower bound by rounding down the fractional variable, if its exists; if it does not exist then we set the lower bound to the upper bound.

Example Suppose we have a set of 4 objects which have a weight w_i and a value v_i as indicated in the table. Assume the capacity of the knapsack is 16.

i	v_i	w_i
1	\$30	5
2	\$45	3
3	\$10	5
4	\$45	9

We will construct the state space where

- each node contains the total current value v in the knapsack,
- the total current weight w of the contents of the knapsack
- the maximum potential value that the knapsack can hold.

We will also keep a record of the maximum value of any node (partially or completely filled knapsack) found thus far.

When we perform the depth-first traversal of the state-space tree, we will determine that a node is "promising" if its maximum potential value is greater than this current best value.

Before we start, we sort the items (as in the Greedy approach) by the value of each item normalized by its weight. We renumber the items as follows:

i	v_i	w_i	$\frac{v_i}{w_i}$
1	\$45	3	\$15
2	\$30	5	\$6
3	\$45	9	\$5
4	\$10	5	\$2

We begin by starting with an empty knapsack so its current weight, W_{cur}, is zero and its current value $V_{\text{cur}} = 0$.

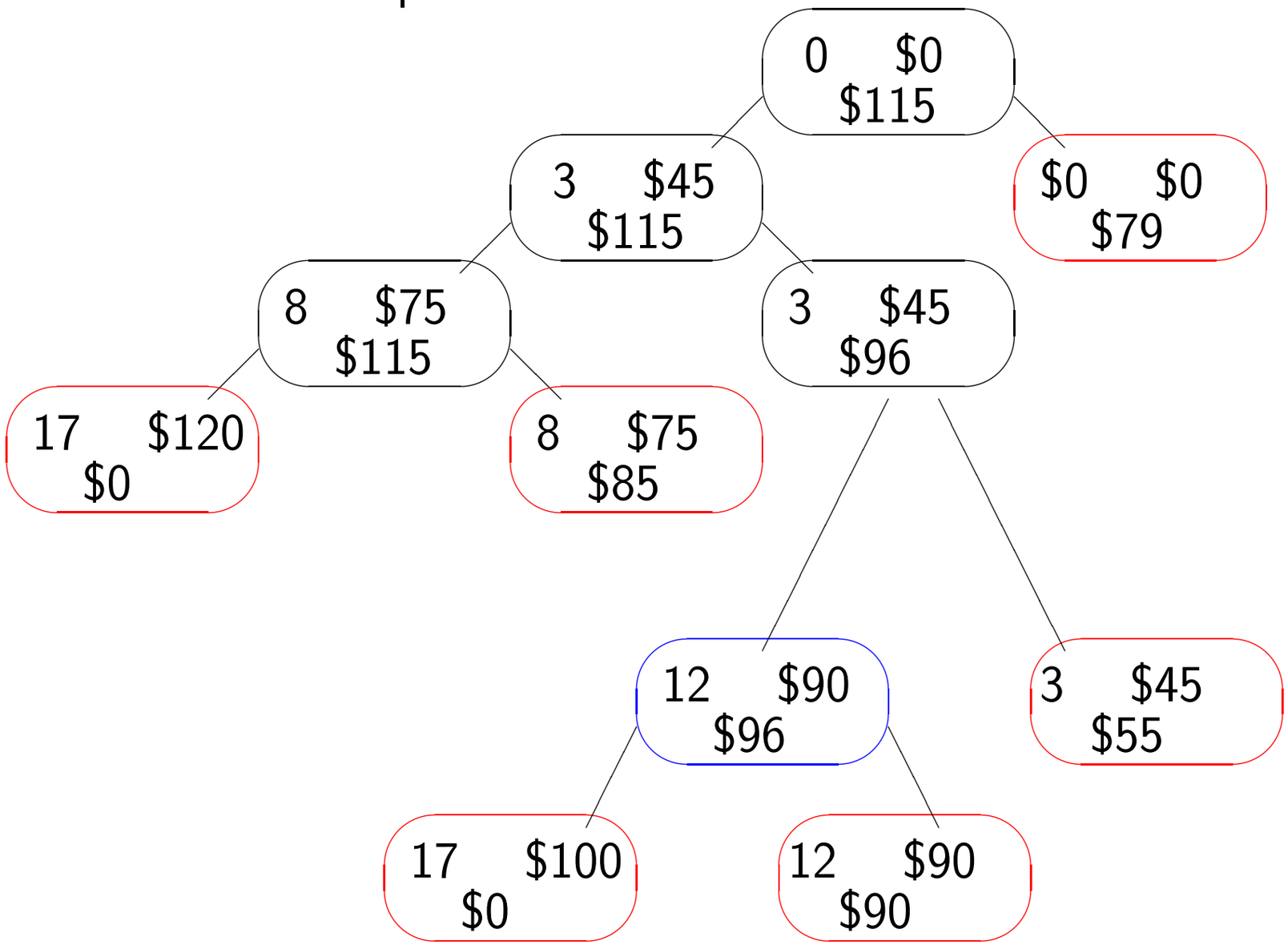
Step 1 Find the maximum potential value treating the problem as if it were the fractional knapsack problem and use the greedy algorithmic solution to that problem.

Step 2 We place each of the remaining objects, in turn, into the knapsack until the next selected object is too big to fit into the knapsack.

Step 3 We then use the fractional amount of that object that could be placed

in the knapsack to determine the maximum potential value.

Here we give $W_{\text{cur}}, V_{\text{cur}}$ and indicate the bound found by using our greedy algorithm for the continuous problem.



Initially we set $W_{\text{cur}} = V_{\text{cur}} = 0$ and we compute the lower bound by considering the fractional knapsack problem. Because our capacity is 16 lbs we can put in all of items 1 and 2 but only 8/9 of item 3. Thus our initial bound is

$$\$45 + \$30 + \frac{8}{9}\Big(\$45\Big) = \$75 + \$40 = \$115$$

Next we either add item # 1 or not. When we add it, we have increased the current weight to $W_{\text{cur}} = 3$pounds and the current value to $V_{\text{cur}} = \$45$. The bound is found by adding all of item # 2 and 8/9 of item # 3 so it is still \$115. If we elect not to add item # 1 then $W_{\text{cur}} = V_{\text{cur}} = 0$ and we compute the lower bound by using the second and third items (total of 14 pounds) and 2/5 of item # 4. Thus the lower bound is

$$\$30 + \$45 + \frac{2}{5}\Big(\$10\Big) = \$75 + \$4 = \$79$$

Now this is lower than our previous value of \$115 so we can't find a larger value in this portion of the tree so we prune it and make this a terminal leaf node.

At the next level we either add item # 2 or not. If we add it, then our current weight becomes 8 pounds and the current value \$75. Again the lower bound is

\$115. If we don't add it then the current weight and value remain the same as the parent node. The lower bound is computed by using items # 1 and 3 (12 pounds) and 4/5 of item # 4 to get

$$\$45 + \$45 + \frac{4}{5}\Big(\$10\Big) = \$90 + \$8 = \$98$$

We continue in this manner and see that the largest value of V_{cur} occurs when we use items 1 and 3 with a total weight of 12 pounds and a value of \$90.

Solving the Knapsack Problem Using Dynamic Programming

We can use Dynamic Programming to solve the Knapsack Problem if the solution to the problem can be obtained recursively in terms of subproblems; i.e., the problem exhibits optimal substructure. So the first thing we must do is define our subproblems such that we can describe our final solution in terms of the solutions to the subproblems as we did in the Coin Changing Problem. In addition, we must have a recursive formula for the subproblems.

As before assume we have n items and a backpack with maximum capacity of W. For concreteness, let's consider the case of 5 items with the following weights and values and a capacity of 20.

item #	weight	value
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

An obvious choice is to define subproblems with weight $1, 2, \cdots, W$. Does this work?

As another approach, let's try a subproblem where we would find an optimal solution for

$$S_k = \{\text{items labeled } 1, 2, \cdots, k\}$$

Does this work? Let's look at our example to see why this doesn't work either. For example, for S_4 we are using items 1, 2, 3, and 4. This four items weigh 14 and using all four is the optimal solution with a value of 20. Now when we proceed to S_5, which is the desired problem, we see that the solution to S_4 is not part of the optimal solution. Because the optimal solution for S_5 uses items

#1, 3, 4 and 5 for a weight of 20 and a value of 26.

So we have to be a bit more clever in defining our subproblems in order to use Dynamic Programming.

We define a function $V(k, \omega)$ where $k = 0, 1, 2, \ldots, n$ (the items) and $\omega = 0, 1, 2, \ldots, W$ (the range of weights) which will give the maximum value for weight ω using items $0, 1, 2, \ldots, k$. If we compute all entries of this array then we will have our result.

We need a recursive relationship between the subproblems. We set

$$V(k, \omega) = \begin{cases} V(k-1, \omega) & \text{if } w_k > \omega \\ \max\left[V(k-1, \omega), V(k-1, \omega - w_k) + v_k\right] & \text{else} \end{cases}$$

If $w_k > \omega$ then this says the item $\#k$ has a weight greater than ω so it can't be added to the knapsack and we use the value of the previous problem $V(k-1, \omega)$.

Otherwise, the item $\#k$ can be added to the knapsack and we have to update the value $V(k, \omega)$. To do this we compute the value when we add item #k so this is the value v_k plus the maximum value using $k-1$ items for a weight of $\omega - w_k$. Then we must make sure that this value is greater than the value when we didn't use item k, i.e., $V(k-1, \omega)$.

Example Let's fill in the array for V for a small knapsack problem with capacity of 5 and four items

item #	weight	value
1	2	3
2	3	4
3	4	5
4	5	6

$\omega \rightarrow$ $i \downarrow$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

pseudo code

Given n items with weight w_i and value v_i and a total capacity of W

Initialize

for $\omega = 0, 1, \ldots, W$ set $V(0, \omega) = 0$

for $i = 1, \ldots, n$ set $V(i, 0) = 0$

for $i = 1, \ldots, n$

for $\omega = 1, \ldots, W$

if $w_i \leq \omega$ (item can be added)

if $v_i + V(i-1, \omega - w_i) > V(i-1, \omega)$

$$V(i, \omega) = v_i + V(i-1, \omega - w_i)$$

else

$$V(i, \omega) = V(i-1, \omega)$$

end if

else

$V(i, \omega = V(i-1, \omega)$ (item can't be added)

end if

end for

end for

How does this compare with the Brute Force approach which is basically $\mathcal{O}(2^n)$?

Our outside loop is from 1 to n and the inside loop is from $\omega = 1$ to W so basically we have $\mathcal{O}(nW)$.

For example, if $W = 20$ and $n = 10$, $2^{10} = 1024$ and (10*20)=200.

Another way to look at this is with the Brute Force approach if it takes K units of time to determine the solution with $n = 20$ then for $n = 30$ it takes $2^{30} = 2^{10}2^{20}$, i.e., $2^{10} \approx 1000$ times longer. For the same weight Dynamic Programming takes $30W$ versus $20W$ which is 1.5 times longer.

Solving the Knapsack Problem using Simulated Annealing

In order to use Simulated Annealing (basically the Metropolis Algorithm) we need an energy to minimize and a way to change from one state to another.

For the Knapsack Problem a solution consists of all zeros and ones so we could start with a vector of length n (for n items) and randomly set each entry to 0 or 1. To alter the state we could just randomly toggle one or more of the entries from 0 to 1 or 1 to 0.

We want to minimize an energy so if we have a new state, say `xnew`, then we first calculate the total weight corresponding to this choice of which items to include. If the weight is feasible, i.e., less than or equal to the capacity of the knapsack, then we just take the energy to be the negative of the corresponding values for these items; i.e., minus the dot product of `xnew` and a vector of the item values. However, if the weight of these items is greater than the knapsack's capacity we want to penalize this choice and set a large positive value for the energy.

Once we have the new state `xnew` and its corresponding energy `Enew` we need to decide whether to accept this state or not. If the energy corresponding to `xnew` is less than the previous energy, `Eold` then we accept it. However, if it is not, this is where the Monte Carlo aspect of the algorithm comes in to play. We generate a random number, say σ, between 0 and 1. If

$$\sigma < e^{(\texttt{Enew}-\texttt{Eold})/T} \quad \Longrightarrow \quad \text{accept } \texttt{xnew}\text{; otherwise reject}$$

where T is the temperature. After the cycle is complete, we increase T by some amount.

pseudo code

Input: n, the number of items, a vector w of weights for each item and a vector v of values of each item; W the capacity of knapsack

Output: `xnew` final solution and corresponding value `-Enew`

Initialize

randomly set `xold` to be a vector of length n with entries 0 or 1

compute energy `Eold` corresponding to `xold`

for $i = 1$, number of cycles

create `xnew` by changing at least one entry of `xold`

calculate `Enew`

if `Enew` $<$`Eold`

accept step so `xold = xnew; Eold = Enew`

else

$\sigma =$ `rand(1,1)`

if ($\sigma < e^{\texttt{Enew}-\texttt{Eold})/T}$)

accept step so `xold = xnew; Eold = Enew`

end if

    end if

    increase temperature, check for convergence

end for

Then we would need a function to get a new state `xnew` from `xold` and another function to calculate the energy given a state.

Calculation of new state:

Input: `xold` and n

Output: new state `xnew`

`xnew = xold`

while `xnew` $\neq$ `xold`

flip_probability $= 1/n$

bits_to_flip $=$ (rand(1,n) $<$ flip_probability)

`xnew = xold`

`xnew`(bits_to_flip) $=$ xor (xnew(bits_to_flip), 1)

end while

This uses a new Matlab command `xor` which is a logical EXCLUSIVE OR.

XOR(S,T) is the logical symmetric difference of elements S and T. The result is logical 1 (TRUE) where either S or T, but not both, is nonzero. The result is logical 0 (FALSE) where S and T are both zero or nonzero. S and T must have the same dimensions (or one can be a scalar).

Example Set $n = 5$ and `xold`=(1, 0, 0, 0, 1)

flip_probability $= 0.2$

rand(1,5) $=$ (.0975, .2785, .5469, .9575, .9649)

bits_to_flip $=$ (1,0,0,0,0)

xnew $=$ xold $=$ (1, 0, 0, 0, 1)

xnew(bits_to_flip)$=$ xor (xnew(bits_to_flip), 1) $=$ (0, 0, 0, 0, 1)

Calculation of energy of state

Input: state x, vector v of values of all items, vector w of weights of all items, W the total capacity of backpack

Output: energy E of state

items_weight $=$ dot (x,w)

if (items_weight $< W$) then

$E =$ -dot(x, v)

else

penalty $=$ items_weight - W

$E=$ 100 * penalty

end if

The penalty coefficient of 100 is random here; we just need a "large penalty".

Example Let's return to our small knapsack problem with capacity of 5 pounds and four items and look at some results generated by our algorithm. We start with an initial guess of $x = (1, 0, 0, 0)$. Remember that the optimal solution is using items 1 and 2.

item #	**weight**	**value**
1	2	3
2	3	4
3	4	5
4	5	6

cycle	xold	Eold	xnew	Enew	accept
1	(0,0,0,0)	0	(1,0,0,0)	-3	yes
2	(1,0,0,0)	-3	(1,0,1,0)	92	no
3	(1,0,0,0)	-3	(0,0,0,1)	-6	yes
4	(0,0,0,1)	-6	(0,1,0,1)	290	no
5	(0,0,0,1)	-6	(0,0,0,0)	0	no
⋮	⋮	⋮	⋮		
	(1,1,0,0)	-7			

The Traveling Salesman Problem

Origin The traveling salesman problem (TSP) was studied in the 18th century by a mathematician from Ireland named Sir William Rowam Hamilton and by a British mathematician named Thomas Penyngton Kirkman.

Problem Definition: Given a set of n cities and the cost of travel or the distance between each possible pair, the TSP is to find the best possible way of visiting all the cities and returning to the starting point that minimize the travel cost or travel distance.

If n is the number of cities to be visited, the total number of possible routes covering all cities can be given as a set of feasible solutions of the TSP is

$$\frac{(n-1)!}{2}$$

To demonstrate how fast the factorial grows, in the table below we give some values for the complexity as a function of n.

n	4	5	6	7	8	9	10
$\frac{(n-1)!}{2}$	3	12	60	360	2520	20,160	181,440

Clearly the Brute Force approach is not feasible unless n is quite small.

For the Traveling Salesman Problem , no serial algorithm exists that runs in time polynomial in n, only in time exponential in n, and it is widely believed that no polynomial time algorithm exists. Oftentime one has to settle for computing an approximate solution, i.e. a single tour whose length is as short as possible, in a given amount of time. For this reason, many of the approaches for solving the Traveling Salesman Problem are modified to take advantage of parallel architectures; we will not consider those here.

Most applications of the TSP problem are in routing and scheduling. We list a few real world applications here. These are taken from *The Traveling Salesman Problem* by Rajesh Matai, Surya Prakash Singh and Murari Lal Mittal (available

free as a pdf download).

1. Vehicle routing. Suppose that in a city n mail boxes have to be emptied every day within a certain period of time. The problem is to find the minimum number of trucks to do this and the shortest time to do the collections using this number of trucks. As another example, suppose that n customers require certain amounts of some commodities and a supplier has to satisfy all demands with a fleet of trucks. The problem is to find an assignment of customers to the trucks and a delivery schedule for each truck so that the capacity of each truck is not exceeded and the total travel distance is minimized. Several variations of these two problems, where time and capacity constraints are combined, are common in many realworld applications. This problem is solvable as a TSP if there are no time and capacity constraints and if the number of trucks is fixed. In this case we obtain an multi-salesman problem. Nevertheless, one may apply methods for the TSP to find good feasible solutions.

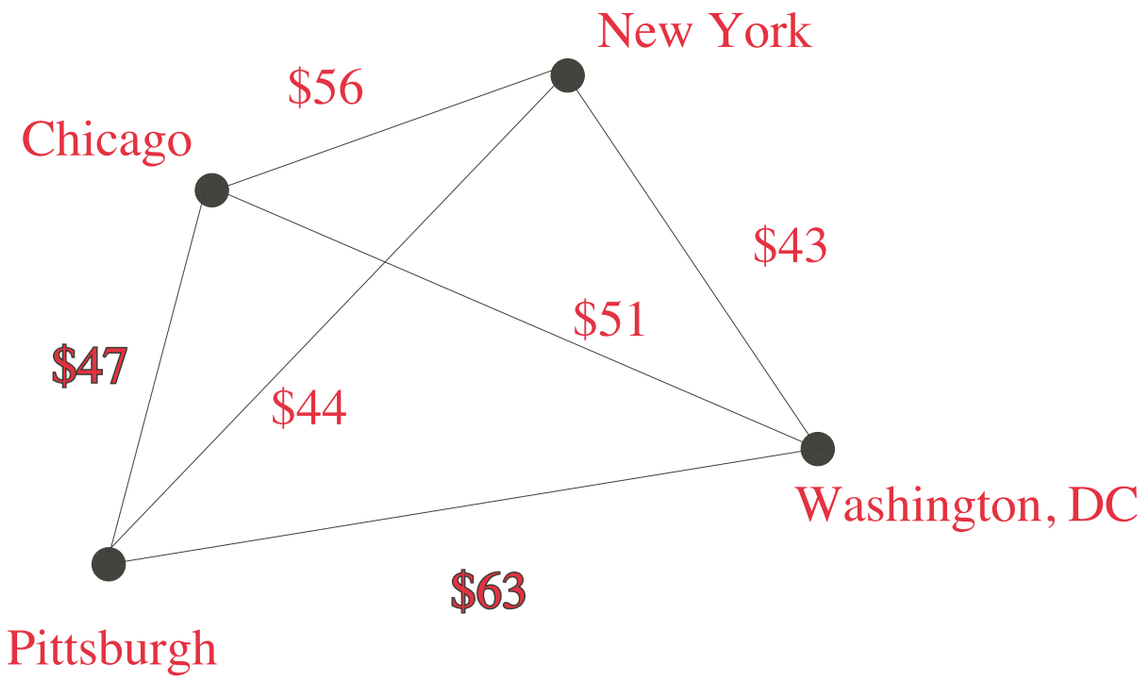
2. Drilling problem of printed circuit boards. To connect a conductor on one layer with a conductor on another layer, or to position the pins of integrated circuits, holes have to be drilled through the board. The holes may be of different

sizes. To drill two holes of different diameters consecutively, the head of the machine has to move to a tool box and change the drilling equipment. This is quite time consuming. Thus it is clear that one has to choose some diameter, drill all holes of the same diameter, change the drill, drill the holes of the next diameter, etc. Thus, this drilling problem can be viewed as a series of TSPs, one for each hole diameter, where the "cities" are the initial position and the set of all holes that can be drilled with one and the same drill. The "distance" between two cities is given by the time it takes to move the drilling head from one position to the other. The aim is to minimize the travel time for the machine head.

3. Overhauling gas turbine engines. This application occurs when gas turbine engines of aircraft have to be overhauled. To guarantee a uniform gas flow through the turbines there are nozzle-guide vane assemblies located at each turbine stage. Such an assembly basically consists of a number of nozzle guide vanes affixed about its circumference. All these vanes have individual characteristics and the correct placement of the vanes can result in substantial benefits (reducing vibration, increasing uniformity of flow, reducing fuel consumption). The problem of placing the vanes in the best possible way can be modeled as a TSP with a special objective function.

4. The order-picking problem in warehouses. This problem is associated with material handling in a warehouse. Assume that at a warehouse an order arrives for a certain subset of the items stored in the warehouse. Some vehicle has to collect all items of this order to ship them to the customer. The relation to the TSP is obvious. The storage locations of the items correspond to the nodes of the graph. The distance between two nodes is given by the time needed to move the vehicle from one location to the other. The problem of finding a shortest route for the vehicle with minimum pickup time can now be solved as a TSP.

Example As a small example of a TSP problem which we can solve by the Brute Force approach of complete enumeration consider the following problem where a salesperson wants to start and end at Chicago and visit New York, Pittsburgh and Washington, D.C. The costs for a train ticket between each pair of cities is given on the graph.



In graph theory terms, we are looking for a circuit that visits each vertex once which is called a Hamilton circuit (as opposed to an Euler circuit which tranverses each edge once).

Because the number of cities is small we can enumerate all possible choices.

1. Chicago $\longrightarrow$ Pittsburgh $\longrightarrow$ DC $\longrightarrow$New York $\longrightarrow$ Chicago $\Longrightarrow$ cost = \$209
2. Chicago $\longrightarrow$ DC$\longrightarrow$ Pittsburgh $\longrightarrow$New York $\longrightarrow$ Chicago $\Longrightarrow$ cost = \$214
3. Chicago $\longrightarrow$ New York$\longrightarrow$ DC $\longrightarrow$ Pittsburgh $\longrightarrow$ Chicago $\Longrightarrow$ cost = \$209
4. Chicago $\longrightarrow$ Pittsburgh $\longrightarrow$ New York $\longrightarrow$DC$\longrightarrow$ Chicago $\Longrightarrow$ cost = \$185

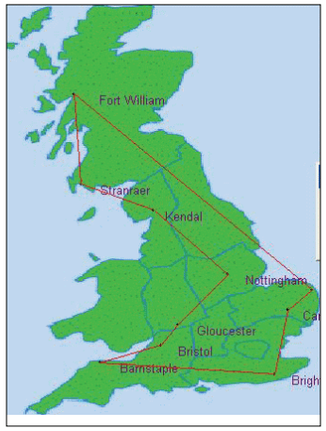
5. Chicago $\rightarrow$ DC$\rightarrow$ New York $\rightarrow$ Pittsburgh $\rightarrow$ Chicago $\implies$ cost = \$185
6. Chicago $\rightarrow$ New York$\rightarrow$ Pittsburgh $\rightarrow$ DC $\rightarrow$ Chicago $\implies$ cost = \$214

Our goal now is to try to find a better way to solve the TSP using some of the techniques we have learned.

Traveling salesman

What is the shortest path to loop through N cities?

[http://www.informatik.uni-leipzig.de/~meiler]

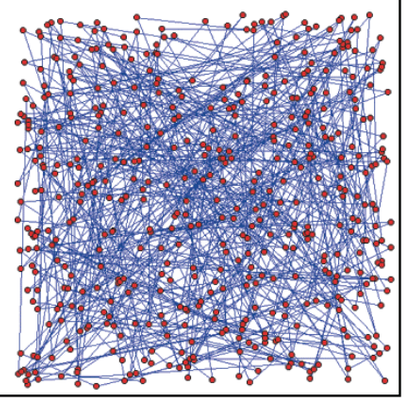


[http://www.superbasescientific.com/]

Traveling salesman: engineering applications

What is the shortest path to loop through N cities?

500 cities, random solution!

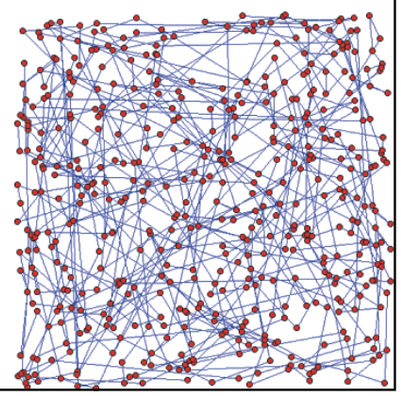


[http://www.logicalgenetics.com/]

Traveling salesman: engineering applications

What is the shortest path to loop through N cities?

500 cities, a better solution!

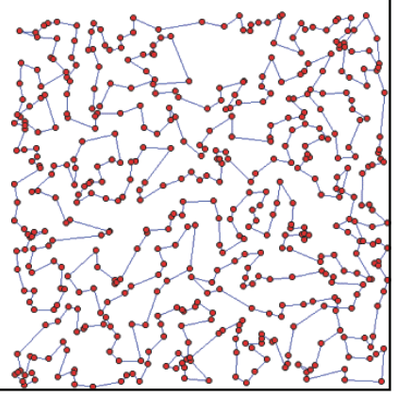


[http://www.logicalgenetics.com/]

Traveling salesman: engineering applications

What is the shortest path to loop through N cities?

500 cities, a much better solution!



[http://www.logicalgenetics.com/]

Solving the Traveling Salesman Problem Using Branch and Bound

We can apply the Branch and Bound approach to solve our problem. Remember that we need to have bounds in order to decide whether to prune a section of the solution set.

An obvious approach is to initially calculate the cost of the path

$$1 \rightarrow 2 \rightarrow 3 \rightarrow \cdots \rightarrow n-1 \rightarrow n \rightarrow 1$$

and this will be our current best solution, i.e., a bound on our result. When we calculate a path that is less than this, we update our value. We will prune the tree when we are calculating a partial path and its cost is already greater than our current lower bound; i.e., there is no need to continue searching that path.

There are other ways to employ Branch and Bound techniques to solve the Traveling Salesman Problem which are especially amenable to parallel computing.

Solving the Traveling Salesman Problem using Dynamic Programming

Remember that to use Dynamic Programming we have to define subproblems S_k which have an optimality property so that to find the solution to S_n we can use the solutions to S_k for $k < n$; in addition we have to have a recursive formula. Recall that we break our problem into stages and then the stages into states which allow us to get from one stage to the next.

The obvious choice is to let stage k represent visiting k cities (including the starting city) and let the decision be where to go next. If the city we are in is the state then this is not enough information to decide where to go to next. Instead the state has to include information about all the cities we have visited plus the city we have ended in. So we could think of a state as a pair (S, k) where S is the set of cities already visited and k is the last city visited.

We need a recursive formula. Let

$C(S, k)$ denote the shortest path from city 1 to city k that visits all cities in S

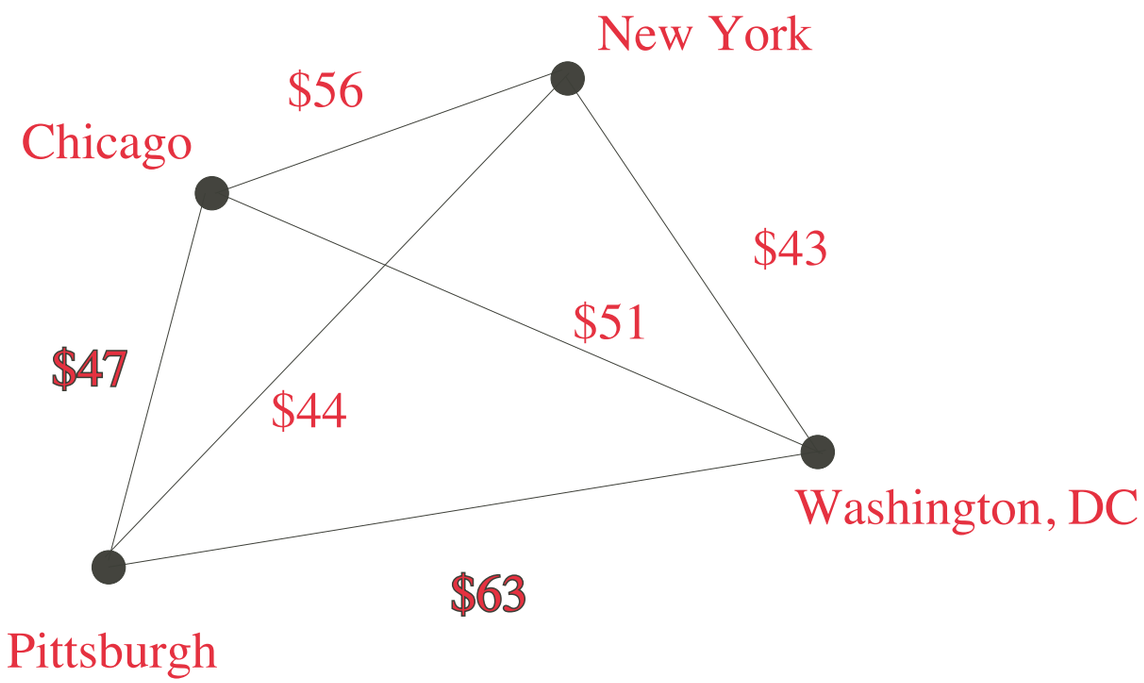
Then our recursive formula is

$$C(S,k) = \min_{m \in S-\{k\}} \Big[c_{mk} + C(S-\{k\}, m) \Big]$$

where c_{ij} represents the cost in going from city i to city j. The first term in this expression gives the shortest way to go to city m without going through city k.

Let's see if this would work for our small Traveling Salesman Problem where we want to start and stop in Chicago. Number the cities in a clockwise manner, i.e.,

Chicago $\longrightarrow$ 1; New York $\longrightarrow$ 2; Washington DC $\longrightarrow$ 3; Pittsburgh $\longrightarrow$ 4



- For $S = \{1\}$ we have $C(1, 1) = 0$
- For S to contain two cities (including # 1) we have to compute:
  - for $S = \{1, 2\}$

$$C(S, 2) = \Big[C(\{1\}, 1) + c_{12}\Big] = 0 + 56 = 56$$

  - for $S = \{1, 3\}$

$$C(S, 3) = \Big[C(\{1\}, 1) + c_{13}\Big] = 0 + 51 = 51$$

$$C(S, 1) = \Big[C(\{1\}, 1) + c_{13}\Big] =$$

  - for $S = \{1, 4\}$

$$C(S, 4) = \min\Big[C(\{1\}, 1) + c_{14}\Big] = 0 + 47 = 47$$

- For S to contain three cities we have to compute:
  - for $S = \{1, 2, 3\}$

$$C(S, 2) = \min_{m=3}\Big[C(\{1, 3\}, 3) + c_{32}\Big] = 51 + 43 = 94$$

$$C(S,3) = \min_{m=2} \Big[C(\{1,2\},2) + c_{23} \Big] = 56 + 43 = 99$$

– for $S = \{1,2,4\}$

$$C(S,2) = \min_{m=4} \Big[C(\{1,4\},4) + c_{42} \Big] = 47 + 44 = 91$$

$$C(S,4) = \min_{m=2} \Big[C(\{1,2\},2) + c_{24} \Big] = 56 + 44 = 100$$

– for $S = \{1,3,4\}$

$$C(S,3) = \min_{m=4} \Big[C(\{1,4\},4) + c_{43} \Big] = 47 + 63 = 110$$

$$C(S,4) = \min_{m=3} \Big[C(\{1,3\},3) + c_{34} \Big] = 51 + 63 = 114$$

- For S to contain four cities we have to compute:
  – for $S = \{1,2,3,4\}$

$$C(S,2) = \min_{m=3,4} \Big[C(\{1,3,4\},m) + c_{m2} \Big] = \min \Big[(110+43), (114+44) \Big] = 153$$

Note that there are two paths to start at Chicago and end at New York; $153 is the minimum cost of the two paths. (If we go to Pittsburgh

first then we have 47+63+43= 153 and if we go to DC first we have 51 +63+44 $>$ 153.)

$$C(S,3) = \min_{m=2,4}\Big[C(\{1,2,4\},m)+c_{m3}\Big] = \min\Big[(91+43),(100+63)\Big] = 134$$

$$C(S,4) = \min_{m=2,3}\Big[C(\{1,2,3\},m)+c_{m4}\Big] = \min\Big[(94+44),(99+63)\Big] = 138$$

Now to finish we need to take $S = \{1,2,3,4\}$ and for each option $C(S,k)$ we need to add in c_{k1}; that is, the distance back to Chicago. We get

$$C(S,2)+56 = 153+56 = 209; \quad C(S,3) = 134+51 = 185; \quad C(S,4) = 138+47 = 185$$

So we find that we can go either of two routes (just as we discovered before)

Chicago $\longrightarrow$ Pittsburgh $\longrightarrow$ New York $\longrightarrow$DC$\longrightarrow$ Chicago

or

Chicago $\longrightarrow$ DC$\longrightarrow$ New York $\longrightarrow$ Pittsburgh $\longrightarrow$ Chicago

Solving the Traveling Salesman Problem Using Simulated Annealing

Lastly, we want to apply Simulated Annealing to the Traveling Salesman Problem

Remember that in Simulated Annealing we basically needed to determine two things. First, if we are at state `xold` then we need to be able to generate a new state `xnew`. There are different ways to do this. Secondly, we need to calculate the energy corresponding to a given state. Once we do this, then the algorithm is analogous to what we have done before.

In this exposition we assume that there are n cities and each is given a set of coordinates (x_i, y_i) so that we want to minimize the total path distance; of course we could just as easily have assigned a weight to each city.

To initialize, we set an initial value for our state `xold`. We can either take a random permutation of the vector $(1, 2, 3, \ldots, n)$ or just take the vector itself.

If we keep the ordering we set

$$\texttt{xold} = (1, 2, 3, \ldots, n, 1)$$

Now to obtain a new state `xnew` from `xold` two choices that are common in the literature are (remember that other choices exist)

- a random section of the path is removed and then replaced with the same cities running in the opposite order
- a random section of path is removed and then replaced in between two cities on another, randomly chosen, part of the path.

Now we need to decide how to calculate `Enew` corresponding to a new state `xnew`. Remember that when $\texttt{Enew} < \texttt{Eold}$ we always accept the new state. However, if this inequality does not hold then sometimes we still accept it, i.e., we "go up hill". Whether to "go up hill" or not is typically determined by generating a random number and comparing it with the equivalent of the Boltzmann energy probability.

For the Traveling Salesman Problem the new energy `Enew` is just the length of the path of the new state `xnew`. So if $j = 1, \ldots, n$ represents the indices of the new state with coordinates (x_j, y_j) then its energy is just

$$\sum_{j=1}^{n} \left[(x_{j+1} - x_j)^2 + (y_{j+1} - y_j)^2 \right]^{1/2}$$

where the node $n + 1$ is always associated with city # 1.

One can add "constraints" to the problem by penalizing the energy. For example, suppose that the salesman has an irrational fear of flying over the Mississippi River. In that case, we could assign each city a parameter μ_i which is 1 if the city is east of the Mississippi, and is -1 if it is west. Then we modify our energy by adding the term

$$\alpha|\mu_{i+1} - \mu_i| \quad \text{or} \quad \alpha\big(\mu_{i+1} - \mu_i\big)^2$$

so that if the cities are on the same side of the Mississippi, then this term is zero; however, if the cities are on opposite sides of the river then we penalize the energy by an amount 2α or in the case of the second term 4α The algorithm now attempts to find the shortest path which minimizes the number of crossings

of the Mississippi River.

There are many applets on the web that demonstrate the solution to this problem; one such is located at

`www.staff.science.uu.nl/ beuke106/anneal/anneal.html`