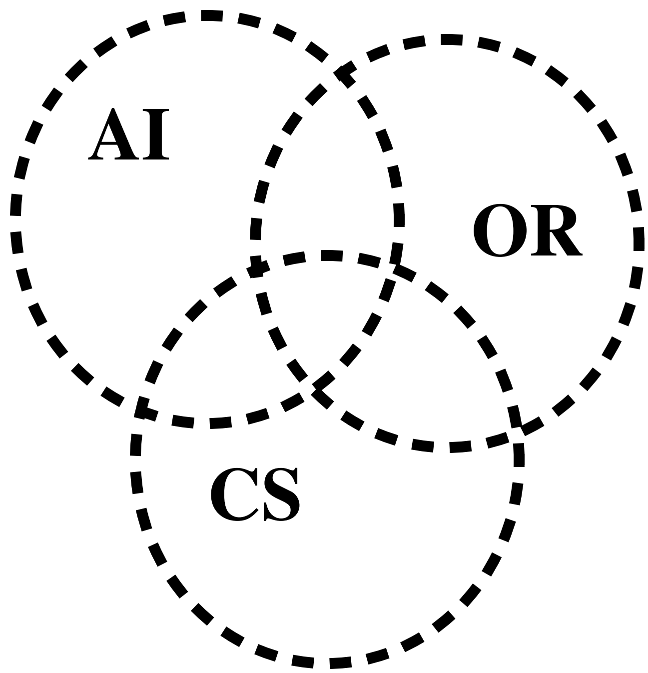
Constraint Programming and Operations Research

Ken McAloon
Carol Tretkoff

Logic Based Systems Lab
Brooklyn College and
CUNY Graduate Center


AI
OR
CS

1. **2LP and continuous variables**

   The feasible region

2. **Linear programs**

   The Fortune Teller (Boole's probabilistic logic)

3. **Discriminating Functions**

   Cell discrimination

4. **Integer programs**

   Call 911

5. **And/Or**

   Fast food

6. **Randomized search strategies**

   The set covering problem

7. **The Injury Method (Branch-n-Bound)**

Proving optimality

Dual thinking and the injury method,

Least discrepancy search

Shuffles

8. Disjunctive Linear Programming

Definitions

A blending problem

Discrete disjunctive programming

9. Cuts

The Pigeon Hole Principle

Capacitated Warehouse Location Problem

10. Libraries and Languages

Informal discussion

2LP

- Is a logic-based language with C-like Syntax
- Captures and extends the practice of linear and integer programming
- Adds a new type `continuous`
- Supports linear constraints on the type `continuous` by means of a linear programming engine
- Supports calls to external C functions and conversely
- Supports failure of constraints by interpreting the logical connectives as in logic programming
- Supports procedural programming for mixed discrete and continuous methods
- Supports new search strategies
- Supports disjunctive linear programming
- Expands to parallel and distributed computing

Creating the Feasible Region

```
2lp_main()
{
continuous X,Y,Z;

    X + Y + Z <= 1;

    X + Y == 1;

    X == Y;

    printf("The coordinates of the wp are\n");
    printf("( %.1f, %.1f, %.1f )\n",X,Y,Z);
}
```

- **Output**

```
The coordinates of the witness point are
( 0.5, 0.5, 0.0 )
```

The Witness Point

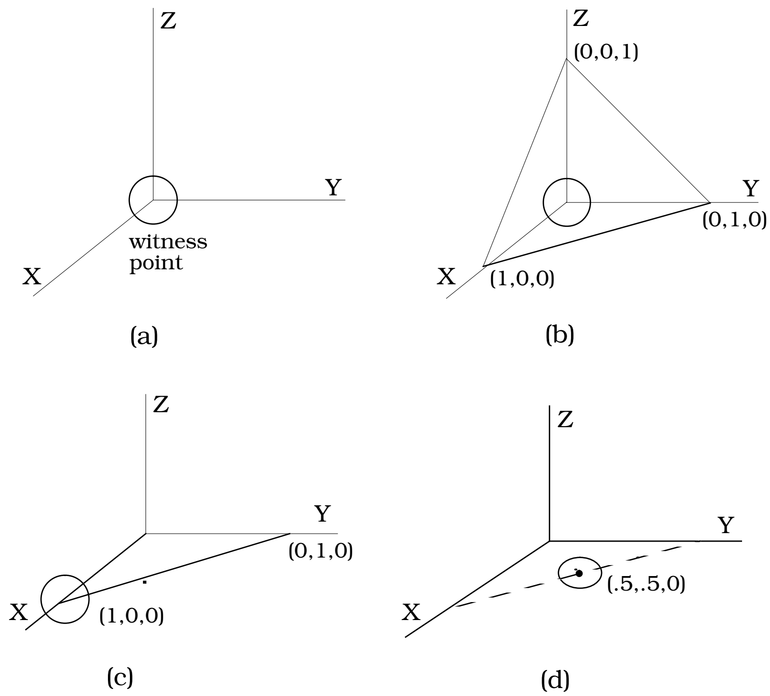


FIGURE 10.1 (a) The positive orthant, (b) the triangular solid defined by the constraints, (c) the line joining (1,0,0) and (0,1,0), and (d) the point (.5,.5,0)

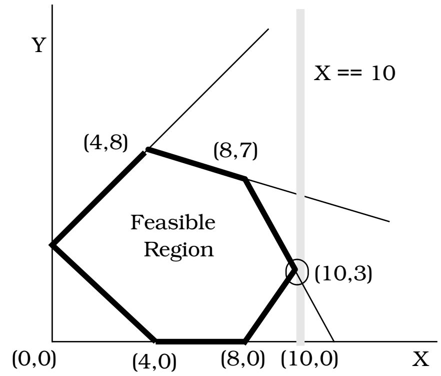


FIGURE 10.2 The vertex that maximizes X is shown as (10, 3).

Figure 10.2

- **The simplex algorithm is also able to find a vertex which optimizes a variable or a linear expression**

```
2lp_main()
{
continuous X,Y;

   -X + Y <= 4;
   X + 4*Y <= 36;
   2*X + Y <= 23;
   3*X - 2*Y <= 24;
   X + Y >= 4;
   max: X;
   max: X + Y;
}
```

The Fortune-teller

An Irish country lass enters the tent of the fortune-teller at the fair and asks about the future. She wants to know the answers to three questions: Will she be rich, will she happy, and will she be long-lived. Most especially, she wants to know if she will be happy and long-lived. Of course the fortune-teller doesn't come straight out with answers to these questions. Rather she skirts around them and points out likelihoods of various combinations. Thus she says that the likelihood of the lass' being poor and unhappy is no more than 5%, and so on.

Naturally the country lass has an excellent memory and when she leaves the fortune-teller's tent, she quickly writes down what she was told. This has been transcribed into the table below.

TABLE 10.1 What the Fortune-teller Said

Predictions	Probabilities
NotRich and NotHappy	At most .05
Rich and NotLongLived	.10
NotRich Happy and LongLived	.38
Rich and Happy	At least .33

The country lass still wants to have a clearer answer to the question whether she will be both happy and long-lived. So she journeys to the city of Cork and consults Professor George Boole there who has a method for extracting the range of probabilities for the question she is interested in.

The professor rises to the occasion and analyzes the situation in terms of a sooth table:

TABLE 10.2 The Sooth Values

Outcome	Rich	Happy	LongLived
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

- **With each of these eight outcomes, the professor associates a continuous variable. This leads to the declaration**

```
continuous X[8];
```

- **Something must happen:**

```
X[0] + X[1] + X[2] + X[3]
+ X[4] + X[5] + X[6] + X[7] == 1.0;
```

- **The set of outcomes where the country lass will be neither rich nor happy has likelihood at most .05; these are outcomes 0 and 1:**

```
X[0] + X[1] <= .05;
```

Continuing this way, a constraint is written down for each prediction made by the fortune teller.

- **Then the task is to find the minimum and maximum likelihoods that she will be happy and long lived. This outcome is given by**

```
X[3] + X[7]
```

and so this expression must be minimized and maximized.

- **The code**

```
2lp_main()
{
continuous X[8];

   X[0] + X[1] + X[2] + X[3] + X[4]
   + X[5] + X[6] + X[7] == 1.0;
   X[0] + X[1] <= .05;
   X[4] + X[6] == .10;
   X[3] == .38;
   X[6] + X[7] >= .33;
   min: X[3] + X[7];
   printf("%.2f and ",X[3] + X[7]);
   max: X[3] + X[7];
   printf("%.2f",X[3] + X[7]);
}
```

- **The results of this consultation are that Happiness and Long Life lies between**

```
      0.61
```

- **and**

```
      0.90
```

- **In this model there were 3 propositional variables and 2^3 continuous variables. If the number of propositional variables is much greater, the resulting linear programs will be intractable.**

- **These issues are addressed in Jaumard, Hansen and Poggi de Aragaö (1991) where a technique known as *column generation* is used to deal with oversized linear programs.**

- **For the relationship of Boole's probabilistic logic with those of Nilsson, Dempster-Shafer, and others, see Andersen and Hooker (1996).**

- **For Boole's work, see Hailperin (North-Holland 1976).**

Conjunction

- **juxtaposition**

```
X[0] + X[1] + X[2] + X[3] + X[4]
+ X[5] + X[6] + X[7] == 1.0;
X[0] + X[1] <= .05;
X[4] + X[6] == .10;
X[3] == .38;
X[6] + X[7] >= .33;
```

- **and loop**

```
and(int i=0;i<8;i++) X[i]<= 1.0;
```

- **The language also supports a** `sigma` **loop:**

```
// Something must happen
sigma(int i=0;i<8;i++) X[i]== 1.0;
```

Integer Programming

- **Finding a solution to a set of linear constraints in which all the variables take integer values in called an integer programming problem (IP)**
- **Finding a solution to a set of linear constraints in which all the variables in some given set take integer values in called a mixed integer programming problem (MIP)**

P-Complete: Does a set of linear constraints have a rational solution ?

NP-Complete: Does a set of linear constraints have an integer solution ?

NP-Complete: Does a set of linear constraints have a solution which is integer in some given set of variables ?

Call 911

The service provided by the 911 system is a critically important part of the city's emergency response effort. The phones must be manned 24 hours a day, 7 days a week; the people answering the phones are known as Personal Communication Technicians or PCTs, a job title that describes the combination of skills required. After a careful analysis of the patterns of calls and demographic information for the coming year, analysts have drawn up a schedule of the number of operators who should be on hand at each of the 168 hours during the week in order to assure the necessary response rate.

Both in order to economize on labor costs and to avoid the problems of overstaffing, what is needed is a way to meet this demand with a minimal or near-minimal number of PCTs. Because there is a union structure, a collective contract is negotiated, and a single schedule for the entire work force can be used. On the other hand, the union has insisted that no one be asked to arrive at work between the hours of 1 AM and 4 AM. Otherwise, all workers arrive on the hour, spend 8 hours at their post except for a one hour break after 4 hours. Each PCT has a work week of 5 consecutive days and then has 2 days off.

TABLE

TABLE 10.1 Staffing Needs

	Number of Personal Communication Technicians Needed for Each of 168 Hours											
Mon	30	24	18	15	14	14	15	25	34	36	38	40
	41	43	46	57	57	59	61	59	55	50	45	38
Tues	32	25	20	17	15	13	17	25	32	35	38	40
	42	43	47	58	57	57	59	57	55	52	47	41
Wed	33	25	20	17	15	13	15	25	32	33	37	39
	42	43	47	57	56	57	57	56	53	50	47	41
Thurs	34	27	22	19	16	15	16	25	31	35	37	40
	44	45	48	57	57	56	58	56	53	53	46	41
Fri	34	28	23	19	16	15	17	25	33	37	39	42
	45	47	51	59	58	60	61	61	57	56	57	55
Sat	48	41	35	30	26	20	18	22	26	32	42	46
	49	53	54	56	56	56	59	59	57	57	56	56
Sun	52	46	41	34	29	23	18	19	25	31	36	41
	46	50	52	53	52	53	54	53	50	49	45	40

- **A continuous variable**

```
Pct[h]
```

will represent the number of PCTs who start their work week at hour `h`,

```
0 <= h < 168.
```

- **Declarations**

```
int wneeded[168];
continuous Pct[168];
continuous Z; //Objective function
```

The goal is to meet staffing needs while minimizing the total number of workers involved. The objective function is

```
Z == sigma(int h=0;h<168;h++) Pct[h];
```

- **The number of PCTs working at hour `h`: Certainly those who start the work week at `h` are there, as are those who started their work week an hour earlier and so on going back to hour `h-8`, with the exception of those who started at `h-4` and who are now on break:**

```
sigma(int j=0;j<8;j++) Pct[h-j]
      - Pct[h-4];
```

- **This also applies to the previous 4 days. When the smoke clears, for every hour of the week `h`, the following constraint expresses that a sufficient number of PCTs are on duty:**

```
wneeded[h] <=
   sigma(int k=0; k<5; k++) (
      sigma(int j=k*24; j<k*24+8; j++)
            Pct[ (h+168-j)%168 ]
      -
      Pct[ (h+168-(k*24+4))%168 ]
   )
```

- **With these constraints, one can minimize the objective function `z` to obtain a solution with fractional workers.**

The linear model is only a fuzzy approximation to what we want.

- **Roundup strategy: Take the linear solution and round up all the `wp(Pct[h])`.**

```
and(int h=0;h<168;h++)
   Pct[h] == ceil(Pct[h]);
```

This gives a solution.

- **Progressive Roundup strategy: Minimize `Z`, take the linear solution and round up `wp(Pct[i])`, fix `Pct[i]` at this value**

```
min: Z;

Pct[i] == ceil(Pct[i]) ;
```

- **Then re-minimize `Z`, and do the same thing to `Pct[i+1]`, etc**

```
min: Z;

Pct[1] == ceil(Pct[1]) ;
```

- **This too is a loop:**

```
and(int i=0;i<168;i++) {
    min: Z;
    Pct[i] == ceil(Pct[i]);
}
```

- **We encapsulate it in a procedure:**

```
progressive_roundoff()
{
   and(int i=0;i<168;i++) {
      min: Z;
      Pct[i] == ceil(Pct[i]);
   }

   printf("PR yields %.0f PCTs\n",Z);
}
```

- **This is a local search strategy.**

- **The main procedure is**

```
// 911 with progressive roundoff

int wneeded[168];
continuous Pct[168];
continuous Z; // The objective function

2lp_main()
{
        data();
        setup();
        naive_roundoff();
        progressive_roundoff();
}
```

- **The naive strategy yields a solution that requires 259 workers, the progressive strategy yields one with 209.**

- **Secret: If there are no lunch breaks, the progressive strategy finds the optimal solution - Bartholdi, Orlin, Ratliff (1980).**

Conditional Disjunction

In symbolic logic, disjunction is denoted by the binary connective $\vee$. **The notation** $p \vee q$ **expresses that at least one of** p **or** q **is true; the terms of a disjunction are called** *disjuncts*.

In classical logic, a disjunction can be known to be true without determining the truth or falsity of its disjuncts. Thus for example if x **and** y **vary over the nonnegative real numbers and we have shown that** xy >= 100,**the statement** (x >= 10) $\vee$ (y >= 10)**can be assumed to be true without first proving** x **to be** >= 10**or else first proving** y **to be** >= 10.

In traditional programming languages, the analog of classical disjunction is not available; in order for the code to proceed through a disjunction, at least one of the disjuncts must first be determined to hold, as in intuitionistic logic.

```
if ( p() || q() || r() || s() )   ...
```

This is called (Gries)

Conditional disjunction or c-or

- **In 2LP,**

```
c_either p();
or q();
or r();
```

```
c_or(int i=0;i<M;i++)
   p(i);
```

- **The Drop/Add heuristic**

```
drop_add()
{
   and(int i=0;i<168;i++) {
      min: Z;
      c_either Pct[i] == floor(Pct[i]);
      or Pct[i] == ceil(Pct[i]);
   }
   printf("Drop/Add yields %.0f\n",Z);
}
```

- **Also finds a very good solution.**

Pattern Recognition

- A pattern is a point in *n*-dimensional space.
- We will consider the case where the patterns are divided into two classes.
- A *discriminating function* is a map that is strictly positive on one class of patterns and strictly negative on the other.

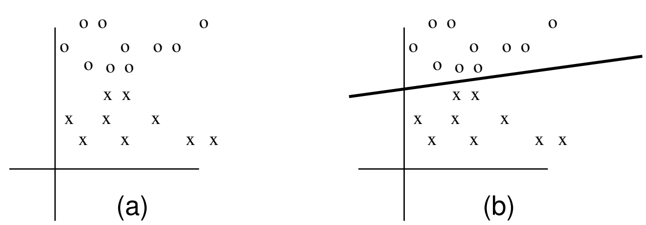


FIGURE 10.3 (a) Two classes of patterns; (b) the patterns separated by the level curve of a linear function.

- A linear discriminating function is an example of a *perceptron*, Rosenblatt (1962); see Minsky and Papert (1988).

- **The Exclusive-Or relation is the classic example of two classes of patterns that cannot be separated by a hyperplane.**
- **However, they clearly can be separated by a quadratic function.**

$$q(x,y) = ax^2 + bxy + cy^2 + dx + ey + f$$

such that

$$q(0,0) \geq \varepsilon,\ q(1,1) \geq \varepsilon \text{ and } q(0,1) \leq -\varepsilon,\ q(1,0) \leq -\varepsilon.$$

- **The values of `x`, `y` that interest us are given. What we have to determine are the coefficients `a, b, c, d, e, f`.**

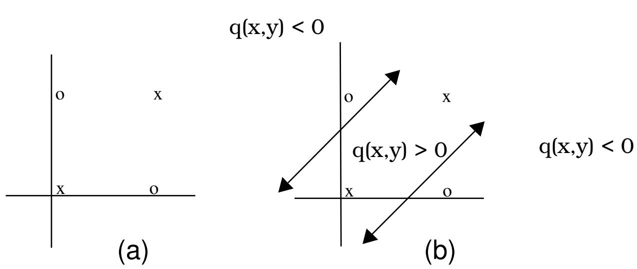


FIGURE 10.4 (a) The exclusive or relation; (b) the level curve of a discriminating function for the exclusive or.

- **Reversing roles and looking at the dual situation, we consider the function**

$$x^2 a + xy\, b + y^2 c + x\, d + y\, e + f\,,$$

making the terms a,b,c,d,e,f **the variables.**

- **Since there is no reason for these terms to be nonnegative, we need to simulate free continuous variables to determine them. For that we split a free variable into a pair of nonnegative continuous variables.**

```
V -> Vp - Vm
```

```
continuous
Ap,Am,Bp,Bm,Cp,Cm,Dp,Dm,Ep,Em,Fp,Fm;
#define EPSILON 1e-2

2lp_main()
{
   0*(Ap-Am)+0*(Bp-Bm)+0*(Cp-Cm)+0*(Dp-
Dm)+0*(Ep-Em)
      + Fp-Fm >= EPSILON;
   1*(Ap-Am)+1*(Bp-Bm)+1*(Cp-Cm)+1*(Dp-
Dm)+1*(Ep-Em)
      + Fp-Fm >= EPSILON;
   0*(Ap-Am)+0*(Bp-Bm)+1*(Cp-Cm)+0*(Dp-
Dm)+1*(Ep-Em)
      + Fp-Fm <= -EPSILON;
   1*(Ap-Am)+0*(Bp-Bm)+0*(Cp-Cm)+1*(Dp-
Dm)+0*(Ep-Em)
      + Fp-Fm <= -EPSILON;

   printf("a is %f\n",Ap-Am);
   printf("b is %f\n",Bp-Bm);
   printf("c is %f\n",Cp-Cm);
   printf("d is %f\n",Dp-Dm);
   printf("e is %f\n",Ep-Em);
   printf("f is %f\n",Fp-Fm);
}
```

- **The solution is**

$$a = -.02,\ b = .04,\ c = -.02,\ d = 0.0,\ e = 0.0,\ f = .01$$

$$q(x, y) = -.02\,x^2 + .04\,xy + -.02\,y^2 + .01$$

- **This function separates the two classes by means of a pair of straight lines**

- **Linear programming can be used to look for both linear and quadratic discriminators because polynomial functions are linear in their coefficients.**
- **This extends the Perceptron Learning Theorem of Rosenblatt (1962) which deals with the case of a linear discriminating function and which rediscovers Motzkin's Iterative Method.**

Cell Discrimination

The data in the handout labeled Cell Discrimination come from the University of Wisconsin Medical School. The question is whether there is a linear or quadratic discriminating function to separate the two classes?

```
#define N 9
#define SZ 50
#define EPSILON 1e-3
#define BENIGN 1
#define MALIGNANT -1

continuous Qp[N][N],Qm[N][N];
continuous Lp[N],Lm[N],Cp,Cm;

int be[SZ][N], ma[SZ][N];
```

- **The main procedure**

```
2lp_main()
{
   data();
   setup();
   c_either linear();
   or quadratic();
}
```

- **The setup is simple**

```
setup()
{// Zero out unneeded variables
   and(int i=0;i<N;i++)
      and(int j=0;j<i;j++) {
         Qp[i][j] == 0;
         Qm[i][j] == 0;
      }
}
```

- **The quadratic case**

```
quadratic()
{
   and(int n=0;n<SZ;n++) {
      approx(be[n],BENIGN);
      approx(ma[n],MALIGNANT);
   }
   printf("Quadratic exists:\n");
}
```

- **Generating the constraints**

```
approx(int c[],int sign)
{
   sign*( //+1 in benign case

   sigma(int i=0;i<N;i++)
      sigma(int j=i;j<N;j++)
         c[i]*c[j]*(Qp[i][j]-Qm[i][j])
   +
   sigma(int k=0;k<N;k++)
      c[k]*(Lp[k]-Lm[k])
   +
   (Cp-Cm) ) >=  EPSILON;
}
```

- **The linear case**

```
linear()
{
   and(inti=0;i<N;i++)
      and(int j=0;j<N;j++) {
         Qp[i][j] == 0.0;
         Qm[i][j] == 0.0;
      }
   and(int n=0;n<SZ;n++) {
      approx(be[n],BENIGN);
      approx(ma[n],MALIGNANT);
   }
   printf("Linear exists:\n");
}
```

- **With these data, there is no linear separator but there is a quadratic separator.**

- **Neural nets can be built by piecing together dicriminating functions of this kind**

Ferris and Mangasarian (1995)

Mangasarian (1995)

Roy and Mukhopadhyay (1991)

Ignizio and Cavalier (1994).

- **If no strict discriminator can be found, an approximate discriminator can be built by introducing continuous variables to measure the error and then minimizing the L_∞ (Chebychev approximation) or L_1 norm of the error term.**

Persistent Disjunction

- **Law of the Excluded Middle**

p or ~ p

Prove that there exist two irrational numbers *a*,*b* **(not necessarily distinct) such that** a^b **is rational**.

Either $\sqrt{2}^{\sqrt{2}}$ is rational or $\sqrt{2}^{\sqrt{2}}$ is not rational. We now show that either branch of the disjunction leads to a solution of the problem. Indeed, first suppose that $\sqrt{2}^{\sqrt{2}}$ is rational; since $\sqrt{2}$ is irrational, we can set $a = b = \sqrt{2}$ to solve the problem. Backtracking to the other alternative, we suppose that $\sqrt{2}^{\sqrt{2}}$ is irrational; but now we can take $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$. So we have proved the existence of a remarkable pair of irrational numbers *a*,*b* without having to determine the true values of *a* and *b*!

Similarly: Prove that either e*pi **or** e+pi **is transcendental**.

- **Proof by contradiction**

~~p -> p

Programming Approximations

- setjmp/longjmp (C,C++)
- cc-call (Scheme)
- catch-and-throw (LISP)

- persistent disjunction in 2LP

```
either p();
or q();


or(int i=0;i<N;i++)
   p(i);
```

- Cf. work of Griffin, Constable, Underwood, Murthy, Krivine

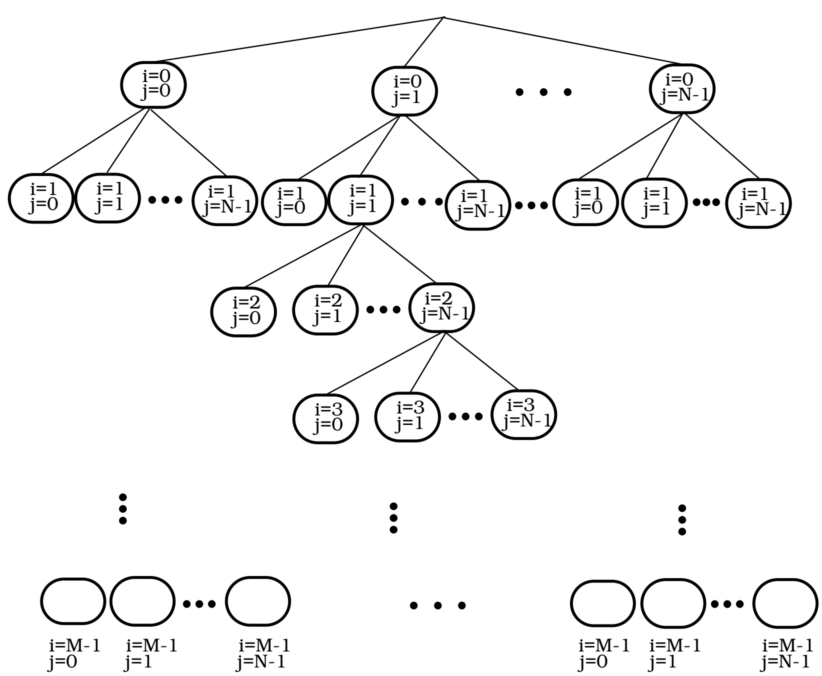
The Canonical NP-Complete Problem

- **CNF Structure**

```
and(int i=0;i<M;i++)
   or(int j=0;j<N;j++)
      p(i,j);
```

- **This code defines a tree with branching factor N and depth M.**

. .




```
and(int i=0;i<M;i++)
   or(int j=0;j<N;j++) {    }
```

FIGURE 10.5 The search tree for nested and/or loops.

Fast Food

A college student would like to know if he can eat all his meals at the local fast food outlet for no more than \$10 a day and still meet the requirements of a balanced diet. In the student newspaper it is reported that a balanced day's food should have at least 100% of the US RDA of vitamins A, C, B_1, B_2, niacin, calcium, and iron. It should also contain at least 55 grams of protein and at most 3000 milligrams of sodium. Furthermore 9 calories come from each gram of fat, and as is well known, at most 30% of one's calories should come from fat. For the student to keep up with his studies and other activities, the diet should provide for at least 2000 calories a day. The student has obtained nutritional and pricing information from the fast food outlet; this information for the student's favorites is given in the table below. The student figures that he can eat at most three servings of any one of these foods in a given day; his other stipulation is that he'll only have milk with cereal and not as a stand-alone drink. Can this be done?

TABLE 10.1 Prices and Nutrition Facts

Item	Price	Calories	Protein	Fat	Sodium	Vit A	Vit C	Vit B_1	Vit B_2	Niacin	Calcium	Iron
Burger	0.59	255	12	9	490	4	4	20	10	20	10	15
Lean B	1.79	320	22	10	670	10	10	25	20	35	15	20
Big B	1.65	500	25	26	890	6	2	30	25	35	25	20
Fries	0.68	220	3	12	110	*	15	10	*	10	*	2
Nuggets	1.56	270	20	15	580	*	*	8	8	40	*	6
Honey	0.00	45	0	0	0	*	*	*	*	*	*	*
Chef Sal	2.69	170	17	9	400	100	35	20	15	20	15	8
Gar Sal	1.96	50	4	2	70	90	35	6	6	2	4	8
EggSand	1.36	280	18	11	710	10	*	30	20	20	25	15
Cereal	1.09	90	2	1	220	20	20	20	20	20	2	20
Yogurt	.63	105	4	1	80	2	*	2	10	2	10	*
Milk	.56	110	9	2	130	10	4	8	30	*	30	*
Orange J	.88	80	1	0	0	*	120	10	*	*	*	*
Gfruit J	.68	80	1	0	0	*	100	4	2	2	*	*
Apple J	.68	90	0	0	5	*	2	2	*	*	*	4

```
#define ITEMS 15
double prices[ITEMS];

#define CALOR 0
#define PROT 1
      ...
#define IRON 10

int food[ITEMS][IRON+1];
```

The unknowns of the problem are the number of servings of each menu item to order during the day:

```
continuous Serv[ITEMS];
```

For the objective function and for formulating the constraint on calories from fat, we declare the following continuous variables:

```
continuous Cost;
continuous Fat, Calories;
```

Finally, we have symbolic constants for some of the parameters of the application:

```
#define CEREAL 9
#define MILK 11
#define LIMIT 3
```

```
2lp_main()
{
        data();
        setup();
        find_meal();
        output();
}


setup()
{
   ...

   // At least 100% of these nutrients
   and(int i=VITA;i<=IRON;i++)
      sigma(int j=0;j<ITEMS;j++)
         food[j][i]*Serv[j] >= 100;

   // At most 3000 mg of sodium
   sigma(int j=0;j<ITEMS;j++)
      food[j][SODIUM]*Serv[j] <= 3000;

   ...
}
```

```
find_meal()
{
   Cost <= 10.00;  // Under 10 dollars

   and(int i=0;i<ITEMS;i++)
      or(int k=0;k<=LIMIT;k++)
         Serv[i] == k;
}
```

P-Complete: Does a set of linear constraints have a rational solution ?

NP-Complete: Does a set of linear constraints have an integer solution ?

NP-Complete: Does a set of linear constraints have a solution which is integer in some given set of variables ?

```
Total cost is 9.62
Total fat is 65.00
Total calories are 2020.00

Buy 1 of item burger
Buy 3 of item fries
Buy 3 of item honey
Buy 1 of item egg sandwich
Buy 3 of item cereal
Buy 3 of item milk
Buy 1 of item apple juice
```

co-NP-Complete: Is this one an optimal solution ?

Randomized Algorithms

- Something you can not live without

  Try it on N-Queens

- Classic: Lin and Lin-Kernighan on TSP
- Genetic Algorithms
- Simulated Annealing
- We will discuss some simple examples in order to show how to work with continuous models

Set Covering Problem

INSTANCE: Collection C of subsets of a finite set S, positive integer $K \leq |C|$.

QUESTION: Does C contain a cover for S of size K or less, i.e., a subset C' of C with $|C'| \leq K$ such that every element of S belongs to at least one member of C'.

Setting up the Code

- Unknown: Whether or not an individual subset is to be in the cover.
- Introduce a continuous variable `X[j]` for each subset

```
continuous X[N],Z;
```

- Represent membership in the cover by fixing this variable at `1`, and at `0` otherwise.
- `Z` represents the objective function.

```
Z == sigma(int j=0;j<N;j++) X[j];
```

- For every element `i`,

$$\sum_{\text{j such that element i is in subset j}} X[j] \geq 1$$

```
sigma(int k=b;k<e;k++) X[s[k]]>= 1;
```

Top of the Model

```
#define NUM ...   // Num of subsets
#define ELMS ..   // Num of elements
#define TOT ..    // Num data entries
int indices[TOT],sz[ELMS+1];
continuous Z,X[NUM];

2lp_main()
{
   data();
   setup();

   randomized_prog_roundoff();
// randomized_drop_add();
// vanilla_branch_and_bound();
// dual_branching();
// least_discrepancy_search();
}
```

DATA

```
data()
{
   indices = {
      2,4,7,18,22,
      8,0,6,19,32,
      7,3,6,13,25,
      4,5,14,29,
      ...
   };

   sz = { 5,5,5,4,3,...,4 };
}
```

- **Code to deal with sparse data**

```
setup()
{
int begin,end;

   and(int j=0;i<N;j++)
      X[j] <= 1;

   end = 0;
   and(int i=0;i<ELMS;i++) {
      begin = end;
      end = end + sz[i];
      // Constraints
      sigma(int j=begin;j<end;j++)
         X[indices[j]] >= 1;
   }

   Z == sigma(k=0;k<N;k++) X[k];
}
```

Negation-as-failure

- **Syntax**

```
not <statement>
```

- **Semantics**

When the code

```
not <statement>
```

is reached,

```
<statement>
```

is executed and its truth value is reversed; *the feasible region is restored to its previous state*.

- **This is analogous to negation in logic programming for the continuous variables**
- **All `double`s and `int`s keep the values given to them in `<statement>`.**

Randomizing

```
#define ITERATIONS 10

randomized_prog_roundoff()
{
double best;

   best = 1e5;
   and(int k=0;k<ITERATIONS;k++)
      if not random_run(k,best);
      then printf(“Iter failed\n”);

   printf("Best was %f\n",best);
}
```

A Random Run

```
random_run(int k, double & best)
{
int q[N];

   printf("Entering %d\n",k);

   find_next_random_permu(q);
   Z <= best-1;
   and(int k=0;k<N;k++) {
      min: Z;
      X[q[k]] == ceil(X[q[k]]);
   }
   printf("Best is now %f\n",Z);
   best = wp(Z);
}
```

- **This is progressive roundoff where the variables have been randomly permuted.**

Branch-and-Bound Search

- **For the Fast Food model the search code was**

```
and(int i=0;i<ITEMS;i++)
   or(int k=0;k<=LIMIT;k++)
      Serv[i] == k;
```

- **An integer variable** `cnt` **can count the number of nodes visited by this code:**

```
naive_search()
{
  and(int i=0;i<ITEMS;i++)
    branch_on(i,cnt);
}

branch_on(int i,int & cnt)
{
  or(int k=0;k<=LIMIT;k++) {
    cnt = cnt+1;
    Serv[i] == k;
  }
}
```

- **To carry out a branch-and-bound search for the optimal solution:**

```
find_min: Z;
subject_to
   naive_search();
```

- **Printing the value of `cnt` at the end of the program, we find that 2832 nodes have been generated. (By the way $8.71 is optimal.)**

- **In the best of all possible worlds, the optimal solution to the linear relaxation would provide a solution in which all the values** `wp(Serv[i])` **were integral.**
- **If this happens at the outset of the program, so much the better. If not, we can work to try to have it take effect locally at a node in the search.**
- **A variable** `Serv[i]` **for which** `wp(Serv[i])` **is not integral is said to be *injured*.**
- **One way to formulate our search task is as a campaign to reduce the number of injured variables until it is zero.**

- **The following routine will hunt for injured variables. It will fail if there are none. If there are injured variables, the index `i` of the first injured variable will be returned in the storeback parameter `bv`:**

```
#define fail 0==1
can_find(int & bv,continuous V[],int n)
{
   and(int i=0;i<n;i++)
      if not integral(V[i]);
      then {
         bv = i;
         return;
      }
   fail;
}
#undef fail
```

- **In the Fast Food example, we will check for injured variables after calling** `min:Cost` **to ensure that the witness point is optimal for the linear relaxation at this node in the search tree.**

```
smart_search()
{
int bv;  // Storeback variable

   and(;;) {
      min: Cost;
      if can_find(bv,Serv,ITEMS);
      then branch_on(bv,cnt);
      else break; // Eureka effect
   }
}
```

- **With this new code,** `smart_search` **replaces** `naive_search` **and the optimal solution is found in 644 nodes, a significant improvement. Further progress can be made by working harder.**

- **Use convexity in branching**

```
land_and_doig(continuous X, int lo,hi)
{
int start;

   start = floor(X);
   either {
      or(int k=start;k>=lo;k--){
         cnt = cnt+1;
         c_either X == k;
         or break;
      }
   }
   or {
      or(int k=start+1;k<=hi;k++){
         cnt = cnt+1;
         c_either X == k;
         or break;
      }
   }
}
```

- **138 nodes**

Branch-and-Bound for Set Problem Cover

```
vanilla_branch_and_bound()
{
int bv;
int cnt;

   cnt=0;
   find_min: Z;
   subject_to
      and(;;) {
         absgap(1.0);
         min: Z;
         if can_find(bv,X,NUM);
         then bin_branch(bv,cnt);
         else break;
      }
   printf("Optimal value %f found
            in %d nodes\n",Z,cnt);
}
```

```
bin_branch(int bv,int & cnt)
{
   either {
      cnt = cnt+1;
      X[bv]==1;
   }
   or {
      cnt = cnt + 1;
      X[bv] == 0;
   }
}
```

Dualized Branching

There is duality between points and sets

- **the** `X[j]` **represent subsets**
- **the linear forms**

```
sigma(int k=begin;k<end;k++)
          X[indices[k]]
```

represent elements

- **If we give this model to a MIP solver with the stipulation that the** `X[j]` **are 0-1 variables, then the solver has no choice but to branch on sets.**
- **Bisschop and Fourer:**

As long as modeling languages (or other optimization problem formats) tend to force the modeler to work in terms of integer variables, however, there is limited motivation to generalize branch and bound solvers to handle combinatorial constraints.

- **Some extra machinery**

```
#define N 10// >= log(NUM) base 2
int cnt;    // To count nodes
int depth[ELMS];  // Trail stacks
int iter[ELMS][N];
```

- **Make the interval** `[sz[e],sz[e+1]-1]` **the location of the indices of the sets containing** `e`

```
set_machinery()
{
   and(int i=1;i<ELMS;i++)
      sz[i] = sz[i] + sz[i-1];
   and(int i=ELMS;i>0;i--)
      sz[i] = sz[i-1];
   sz[0] = 0;
}
```

- **branch on elements not sets**

```
dual_branching()
{

   set_machinery();

   find_min: Z;
   subject_to
      search_on_elements();

   printf("Found %f in %d \n",Z,cnt);
}

search_on_elements()
{
int bv;

      and(;;){
         min: Z;
         absgap(1.0);
         if determine_element(bv);
         then
         tend_to_element(bv,X);
         else break;
      }
}
```

Selecting Injured Element

- **Select injured element occurring in fewest subsets**

```
determine_element(int &bv)
{
int gauge;

   gauge =NUM+1;
   bv = ELMS;
   and(int e=0;e<ELMS;e++)
      if {
         injured(e);
         sz[e+1]-sz[e] < gauge;
      }
      then {
         bv = e;
         gauge = sz[e+1]-sz[e];
      }
   bv < ELMS;
}
```

- **Element `e` is injured if no set containing it has witness point value `1.0`**

```
injured(int e)
{
   and(int i=sz[e];i<sz[e+1];i++)
         wp(X[indices[i]])< 1.0;
}
```

- **INVARIANT: With each element is associated a collection of sets that contain it, some one of which must appear in the cover. The set is given by a subinterval** `[m,n]` **of** `[0,sz[e+1]-sz[e]-1]`**. This is assured by the constraint**

```
sigma(int k=m;k<=n;k++)
   X[indices[sz[e]+k]] >= 1.0;
```

- **When an element is tended-to this interval is split into 2 (disjoint) halves. Two branches are created, one for each of these subintervals.**
- **An element can be re-injured even after it has been tended to. Machinery is needed to get us back to the interval** `[m,n]` **currently in force in order to refine it further if possible.**
- **For that we use a trail stack and a marking technique**

- **Marking and trailing**

```
tend_to_element(int r,continuous X[])
{
int m,mid,n;
   m = 0;
   n = sz[r+1] - sz[r] - 1;
   and(int i=0;i<depth[r];i++)
      if iter[r][i] == LEFT;
      then n = m+(n - m)/2;
      else m = m + (n - m)/2+1;

   mid = m+(n-m)/2;
   depth[r] = depth[r]+1;      // Trail
   if
      sigma(int i=sz[r]+m;
               i<=sz[r]+mid ;i++)
            wp(X[indices[i]])
        >
      sigma(int i=mid+1;i<=n;i++)
         wp(X[indices[sz[r] + i]]);
   then
   left_then_right(r,m,mid,n,X) ;
   else
   right_then_left(r,m,mid,n,X) ;
}
```

```
left_then_right(int r,m,mid,n,
                        continuous X[])
{
   either left(r,m,mid,n,X);
   or right(r,m,mid,n,X);
   or untrail(r);
}
```

- **Mark left or right**

```
left(int r,m,mid,n,continuous X[])
{
   cnt = cnt+1;
   iter[r][depth[r]-1] = LEFT;// Mark
   shrink_sets(r,m,mid);
}
```

- **Pop the trail stack**

```
untrail(int r)
{
   depth[r] = depth[r] - 1;
   fail;
}
```

Iterative Deepening

- In AI work, a great deal of attention has been paid to iterative deepening strategies.
- The idea is to capture the advantages of both breath-first and depth-first search
- The trick is to fix a “limit” for a depth first search, perform dfs to this depth; if necessary increase the limit and try again
- The way “limit” is computed depends on the semantics of the application

IDA* - Least Discrepancy Search

- **Use injuries to compute "depth"**

```
least_discrepancy_search()
{
int f[NUM];
int bv,num;
int cnt;
   cnt = 0;
   c_or(int k=0;k<NUM;k++) {
      initialize(f,k);
      find_min: Z;
      subject_to {
         absgap(1.0);
         and(int i=0;i<NUM;i++) {
            min: Z;
            comp_injury(bv,num);
            num <= NUM - i - f[i];
            if bv < NUM;
            then branch(bv,cnt);
            else break;
         }
      }
      printf("Depth was %d\n",k);
   }
   printf("LDS found %f in %d\n",Z,cnt);
}
```

OR Library Example

On Problem E.1 from the Imperial College Library:

50 elements, 500 sets

- **Naive roundoff: 50**
- **Progressive roundoff : 6**
- **Drop/Add : 16**
- **Randomized progessive roundoff : 5**
- **Randomized drop/add : 6**
- **Vanilla branch and bound : 88 nodes**
- **Dualized branching : 78 nodes**
- **Least Discrepancy Search : 6 at depth 6**

J. E. Beasley, Algorithms for the Set Covering Problem, European Journal of Operational Research, 1987

Shuffling

Suppose that we have a MIP program which requires that `X[i]` take an integer value for `i < M`. Let us consider the situation where we can solve this problem for `K << M` variables to optimality but not for `M`. Let us also suppose that through a quick-and-dirty routine we know that there is a solution, given by the `M` vector $e_0,...,e_{M-1}$. Then randomly select a subset of `[0,...,M−1]` of length `M−K`; using negation-as-failure, fix `X[i]` to e_i for `i` in this subset and solve for the optimal solution in the remaining `K` variables. Record this new solution in $e_0,...,e_{M-1}$ and repeat the process.

Disjunctive Linear Programming

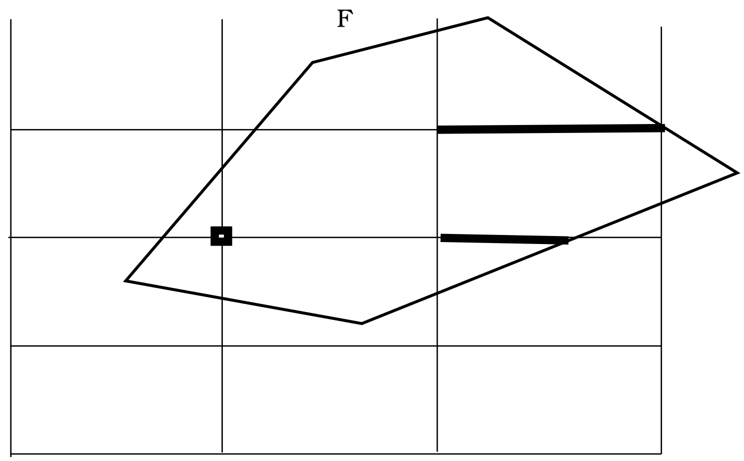
- **An extension of Mixed Integer Programming**

A union of polyhedral sets is called a disjunctive set. The problem of determining whether the intersection of a family of disjunctive sets is non-empty is called the disjunctive linear programming problem or simply disjunctive programming problem.

In mathematical notation the problem is to determine if a set of the following form is non-empty:

$$\bigcap_{i < M} \bigcup_{j < N_j} F_{ij}$$

This set is called the solution set of the disjunctive programming problem.



$$F \cap \bigcap_{i < M} \bigcup_{j < N} F_{ij}$$

- **Example in handout: A multiperiod planning problem where the logic is best handled directly and not reduced to integrality conditions**

- **Quintessential example is job shop - success story for discrete disjunctive programming:**

Temporal constraints stating that tasks within a job must follow an order

Disjunctive requirements: on each machine, either job i precedes job j or job j precedes job i

- **Back and forth between discrete and linear disjunctive programming is a current field of research (e.g. Session at INFORMS in Atlanta)**

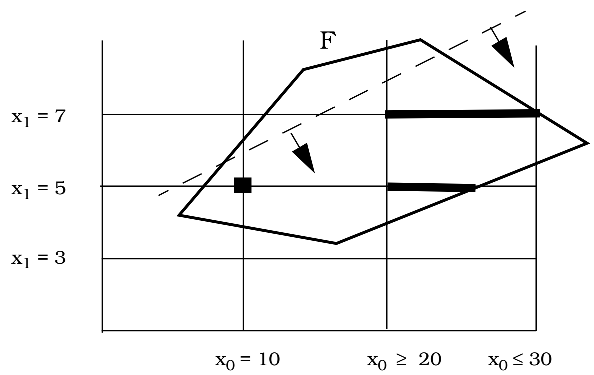


FIGURE 10.1 A disjunctive program with initial feasible region. The dashed line represents a cut.

Cuts

A valid cut is a constraint that is satisfied by the witness point whenever a solution to the constraints and logical requirements of the disjunctive program is found.

When a valid cut is given by an inequality constraint, the equality form of the constraint defines a hyperplane that cuts the feasible region of the linear relaxation into two parts, one of which can be ignored. For this reason, cuts are historically called cutting planes.

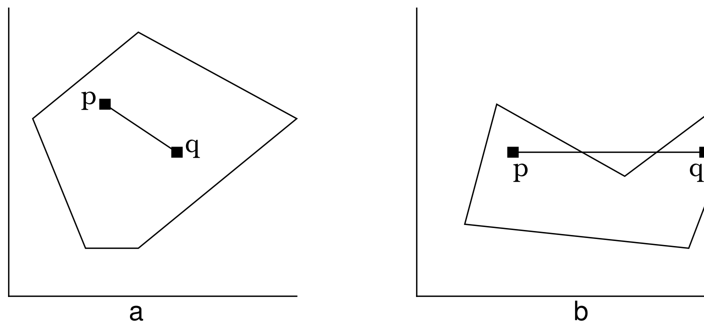


FIGURE 10.2 (a) A convex region; (b) a non convex region.

A set E in n-dimensional space is convex if for any pair of points p, q in E, the straight line segment from p to q is included in E .

The solution set of a disjunctive program is itself a disjunctive set and so has the form $\cup F_i$, where the F_i are feasible regions. In general, disjunctive sets are not convex. The mathematical result that underwrites the search for valid cuts is that the convex closure of a disjunctive set is itself a polyhedral set and that the vertices of this polyhedral set are among the vertices of the components F_i of the disjunctive set.

The Pigeon Hole Principle

```
2lp_main()
{
continuous P[N+1][N];

   and(int i=0;i<N+1;i++)
      and(int j=0;j<N;j++)
         P[i][j] <= 1;

   and(int i=0;i<N+1;i++)
   sigma(int j=0;j<N;j++) P[i][j] >= 1;

   and(int j=0;j<N;j++)
      and(int i=0;i<N+1;i++)
         and(int k=0;k<i;k++)
            (1-P[i][j]) + (1-P[k][j])
                        >= 1;

   <show no integer solution exists>
}
```

- **Treating this as an integer program will give you headaches**

- **A cut is needed: For N holes at most N pigeons**

```
sigma(int j=0;j<N;j++)
   sigma(int i=0;i<N+1;i++) P[i][j] /
<= N;
```

- **Adding this makes the linear constraints infeasible**

Capacitated Warehouse Location Problem

Thirty supermarkets that are part of a chain of stores in New York City have monthly demands for products to be supplied from the warehouses operated by the chain. Since the merchandise required by the stores will be very much the same, the total monthly demand of each store can be expressed as a single number.

To serve these supermarkets, 6 new warehouse sites have been prospected. Data has been gathered on the capacity of each warehouse and the cost of supplying the different supermarkets from each of these sites.

There are two types of cost. With each warehouse site is associated a fixed monthly cost. With each warehouse and with each supermarket is associated the cost of shipping a unit of supply; the total cost generated this way is then proportional to the amount shipped.

The objective is to minimize monthly cost while meeting the needs of the supermarkets. In other words, the problem is to determine which warehouses to use so as to minimize cost while meeting demand and to provide a schedule of how much of each supermarket's demand should be supplied from each warehouse.

```
#define M 6          // Warehouses
#define N 30         // Supermarkets

double c[M][N];      // Proportional cost
double fc[M];        // Fixed cost
double kap[M];       // Capacity
double d[N];         // Demand

continuous Y[M];
continuous X[M][N]
```

- `Y[i]` **is a fuzzy warehouse - it will be** `1` **if the warehouse is built,** `0` **otherwise**
- `X[i][j]` **is the fraction of supermarket** `j`**'s demand that is supplied by warehouse** `i`

- **Each supermarket must be supplied**

```
sigma(int i=0;i<M;i++) X[i][j] == 1.0;
```

- **Warehouse constraint - capacity cannot be exceeded**

```
sigma(int j=0;j<N;j++) d[j]*X[i][j]
<= kap[i];
```

- **Tighter warehouse constraint**

```
sigma(int j=0;j<N;j++) d[j]*X[i][j]
<= kap[i]*Y[i];
```

- **Objective function - fixed costs + proportional**

```
Cost ==
sigma(int i=0;i<M;i++) fc[i]*Y[i]
    +
sigma(int i=0;i<M;i++)
    sigma(int j=0;j<N;j++)
        c[i][j]*X[i][j];
```

- **The model**

```
2lp_main()
{
   data();
   setup();
   quick_and_dirty();

   find_min: Cost;
   subject_to
      search();

   output();
}
```

- **The cuts**

```
X[i][j] <= Y[i];
```

- **Only add if violated at this node**

```
add_cuts()
{
int flag;

   flag = 1;
   and(;flag != 0;){
      flag = 0;
      and(int i=0;i<M;i++)
         and(int j=0;j<N;j++)
            if wp(X[i][j]) > wp(Y[i]);
            then {
               X[i][j] <= Y[i];
               flag = 1;
               min: Cost;
            }
   }
}
```

- **Branch-and-Cut**

```
search()
{
int bv;

   and(;;) {
      min: Cost;
      add_cuts();
      if find_branching_variable(bv);
      then fix(Y[bv],X[bv]);
      else break;
   }
}
```

This code finds the optimal solution with no branching at all.

A program of automating the process is undertaken in the MINTO system of Nemhauser, Savelsbergh, and Sigismondi (1994).

Integer Goal Programming

- **Linear goal programming**

Prioritized objectives

min: X,   min: Y,   min: Z, min: W

min: 1e6 X + 1e4 Y + 1e2 Z + W

- **Integer programming**

For backtracking in the branch-and-bound search, you (almost always) need to enforce a gap (say of 1) to avoid endless equivalent solutions

- **Integer goal programming**

min: X; store value in x, discard solution, X == x;

min: Y; store value in y, discard solution, Y == y;

min Z; store value in z, discard solution, Z == z;

min: W

Nederlandse Spoorwegen

```
continuous F[ARCS];// Type III
continuous G[ARCS];// Type IV
continuous Z;

2lp_main()
{
double cache[3];

   data();

   constraints();
   goal1(cache);
   goal2(cache);
   goal3(cache);

   printf("Node cnt is %d\n", cnt);
}
```

```
goal1(double cache[])
{
   if not {
      Z ==
         sigma(int i=0;i<4;i++)(
         3*F[overnight[i]] +
         4*G[overnight[i]]; )

      find_min: Z;  subject_to
         search(1.0);

         cache[0] = wp(Z);
         cache[1] =
      wp(sigma(int i=0;i<4;i++)
         ( F[overnight[i]] +
            G[overnight[i]]; )
   }
   then {
      printf("No solution\n");
      exit();
   }
   else {
      printf("First goal\n");
      output();
   }
}
```

One Search Does All

```
search(double gap)
{
int bv;

   absgap(gap);
   min: Z;
   and(;;) {
      if bi_injured(bv);
      then bi_branch(bv);
      else break;
   }
}
```

Customize the Branching Choice

```
bi_injured(int &bv)
{
int traffic;

   traffic = 0;
   bv = ARCS;
   and(int i=198;i<ARCS;i++)
      if not {
         integral(F[i]);
         integral(G[i]);
      }
      then {
         if
         traffic < demand_first[i]
             + demand_second[i];
         then {
         traffic = demand_first[i]
               + demand_second[i];
         bv=i;
      }
   }
   bv < ARCS;;
}
```

Customize the Branching Strategy

```
bi_branch(int n)
{
   if !integral(F[n]);
   then {
      //dakin_branch(F[n]);
      branch_on(F[n]);  // L&D like
      min: Z;
   }
   if !integral(G[n]);
   then {
      //dakin_branch(G[n]);
      branch_on(G[n]);// L&D like
      min: Z;
   }
}
```

- **This much is easy - there are also additional disjunctive requirements on coupling the cars that are difficult to translate into integrality requirements without increasing the size of the model beyond tractability**
- **A case for hybrid methods ??**

Constraint Propagation

GERALD + DONALD = ROBERT

Drives MIP solvers crazy
Needs more constraint propagation

Bibliography

Ait-Kaci, H. (1991) *Warren's Abstract Machine*: *A Tutorial Reconstruction*, MIT Press.

Andersen, K. A., and Hooker, J. N. (1996) A linear programming framework for logics of uncertainty, *Decision Support Systems* **16**, pp. 39-53.

Applegate, D., and Cook, W. (1991) A computational study of the job-shop scheduling problem, *ORSA Journal on Computing* **3**, pp. 149-156.

Anderson, S., and Patny, S. (1994) Keeping the Big Apple fiscally fit, *OR/MS Today*, August 1994, pp. 46-49.

Barth, P. (1996) *Logic Based 0-1 Constraint Programming*, Kluwer.

Bartholdi, J. J., Orlin, J. B., and Ratliff, H. D. (1980) Cycle scheduling via integer programs with circular ones, *Operations Research* **28** (5), Sept.-Oct., pp. 1074-1085.

Carlier, J., and Pinson, E. (1989) An algorithm for solving the job shop problem, *Management Science* **35**, pp. 164-176.

Carlier, J., and Pinson, E. (1990) A practical use of Jackson's preemptive schedule for solving the job-shop problem, *Annals of Operations Research* **26**, pp. 269-287.

Cox, J., McAloon, K., and Tretkoff, C. (1992) Computational complexity and constraint logic programming, *Annals of Mathematics and Artificial Intelligence* **5**, pp. 163-190.

Dincbas, M., van Hentenryck, P., Simonis, H., and Aggoun, A. (1988) The constraint logic programming language CHIP, in *Proceedings of the 2nd International Conference on Fifth Generation Computer Systems*, pp. 249-264.

Ferris, M., and Mangasarian, O. (1995) Breast cancer diagnosis via linear programming, *Computational Science and Engineering* **2**, pp. 70-71.

Hailperin, T. (1976) *Boole's Logic and Probability*, Studies in Logic and the Foundations of Mathematics **85**, North-Holland.

Jaumard, B., Hansen, P., and Poggi de Aragaö, M. (1991) Column generation methods for probabilistic logic, *ORSA Journal on Computing* **3**, pp. 135-148.

Ignizio, J., and Cavalier, T. (1994) *Linear Programming*, Prentice-Hall.

Krivine, J.-L. (1994) Classical logic, storage operators and second-order lambda-calculus, *Annals of Pure and Applied Logic* **68**, pp. 53-78.

Lin, S., and Kernighan, B. W. (1973) An effective heuristic algorithm for the traveling salesman problem, *Operations Research* **21**, pp. 498-516.

Mangasarian, O. L. (1993) Mathematical programming in neural networks, *ORSA Journal on Computing* **5**, pp. 349-380.

Nemhauser, G. L., and Wolsey, L. A. (1988) *Integer and Combinatorial Optimization*, Wiley.

Nemhauser, G., Savelsbergh, M., and Sigismondi G. (1994) MINTO, A Mixed INTeger Optimizer, *Operations Research Letters* **15**, pp. 47-58.

Rosenblatt F. (1962) *Principles of Neurodynamic*s, Spartan Books, New York.

Roy, A., and Mukhopadhyay, S. (1991) Pattern classification using linear programming, *ORSA Journal on Computing* **3**, pp. 66-80.

van Hentenryck, P. (1989) *Constraint Satisfaction in Logic Programming*, MIT Press.

Sources

Optimization and Computational Logic

Wiley, Discrete Mathematics and Optimization Series, August 1996

2LP: Logic Programming and Linear Programming

Principles and Practice of Constraint Programming, edited by Saraswat and VanHentenryck, MIT Press

Parallel Integer Goal Programming

(with David Arnow) Proceedings of the 1995 ACM Computer Science Conference pp 42-47

http://www.brooklyn.cuny.edu/lbslab

- **Windows code**
- **Sparc**
- **RS600**
- **Linux**
- **Papers**