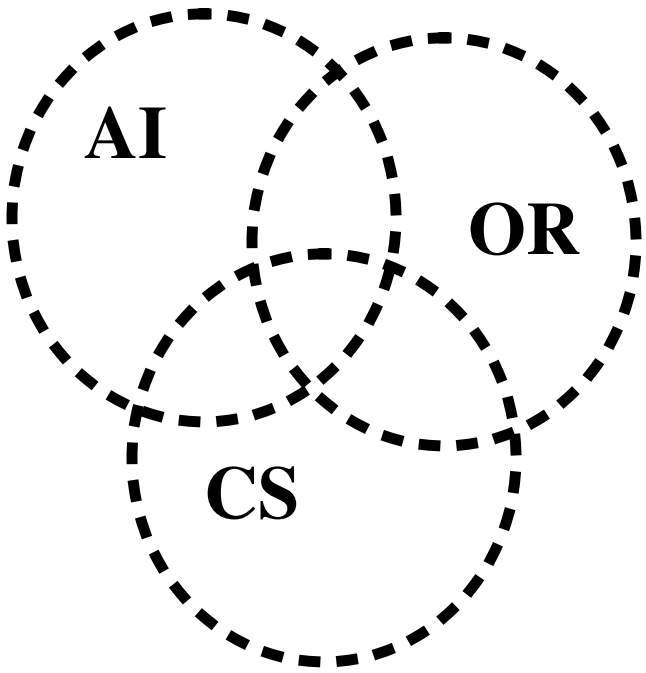
2LP: Linear Programming and Logic Programming

Ken McAloon
Carol Tretkoff

Logic Based Systems Lab
Brooklyn College and
CUNY Graduate Center


AI
OR
CS

Progress in OR

Interior Point Methods

Better and Better Simplex Code

Cut Technology

Local Search

Better Modeling Languages

...

Progress in AI

Neural Nets

Randomized Algorithms

Local Search

Declarative vs Procedural

Constraint Logic Programming

...

Progress in CS

Abstract Data Types

Object Orientation

Logic Programming Compilation (WAM)

Parallel and Distributed Computing

Faster Machines

...

Constraint Logic Programming

- **Prolog III (Colmerauer)**

- **CLP Scheme (Jaffar-Lassez)**

CLP(R)

- **CHIP (ECRC, Cosytec)**

Embeds a discrete CSP solver in a programming language

- **Charme**

Recent ORSA Journal Paper

- **Ilog Solver**

C++ Library

- **2LP**

2LP

- Captures and extends the practice of linear and integer programming in a small programming language
- Has C-like Syntax
- 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
- 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;

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

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

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

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

The Witness Point

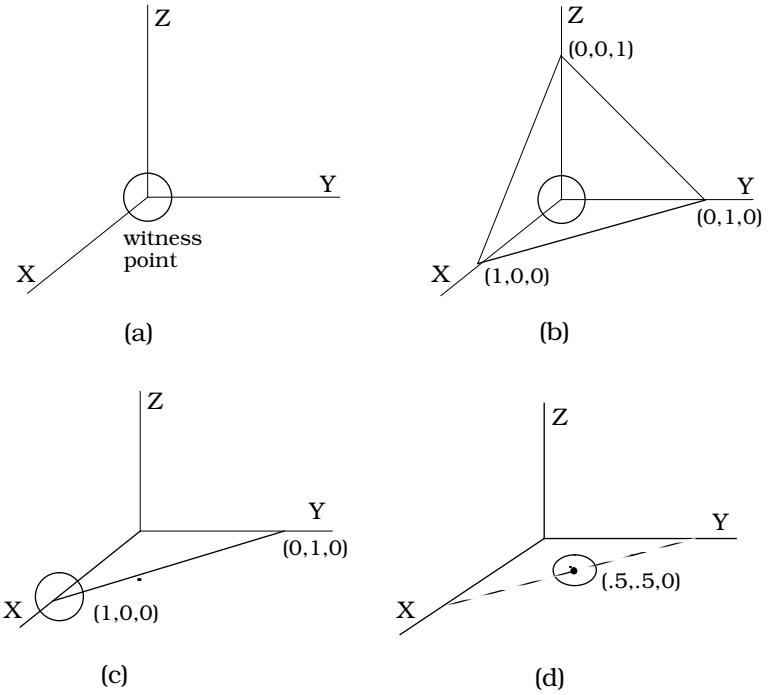


FIGURE 0.1 (a) The positive orthant, (b) the triangular solid defined by the constraints, (c) the line joining (1,0,0) and (0,1,0), and (d) the point (.5,.5,0)

When are LP and MIP modeling not enough ?

- Disjunctive Programming
- Goal Programming
- Integer Goal Programming
- Building Neural Nets
- Customizing Branching Strategies
- Customizing Cut Generation
- Randomized Algorithms
- Local Search
- Etc, Etc

Alternate Title

Eight searches in search of a programming language

Used with permission of L. Pirandello

An Example

We will illustrate some of these points via the Minimum Cover Problem

- **Progressive roundoff**
- **Drop/Add**
- **Randomized progressive roundoff**
- **1-ply lookahead**
- **Vanilla Branch and Bound**
- **The Injury Method**
- **Dualized branching**
- **Least discrepancy search**

Minimum Cover

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 elements of S belongs to at least one member of C'.

Setting up the Code

The unknown here is whether or not an individual subset is to be in the cover. Let us introduce a continuous variable **`X[i]`** for each subset and represent membership in the cover by fixing this variable at 1, and at 0 otherwise. **`Z`** represents the objective function.

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

To express that at least one set in the cover contains element j, we can require that the sum of the **`X[i]`**, where **`i`** is a subset containing this element, must be greater than or equal to 1.

For every element j,

$$\sum_{\text{j in subset i}} X[i] \geq 1$$

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

2lp_main()

```
#define NUM 33  // Num of subsets
#define ELMS 42 // Num of elements
#define TOT 174 // Num data entries
int indices[TOT],sz[ELMS+1];
continuous Z,X[NUM];

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

   progressive_roundoff();
// drop_add();
// randomized_prog_roundoff();
// randomized_drop_add();
// one_ply_lookahead();
// 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;

   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;
   }
   // A set can only be in
   // the cover at most once

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

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

We have bounded the variables **X[i]** by 1 to insure that a set can only appear once in the cover!

Conjunction

- juxtaposition
- and loop

```
progressive_roundoff()
{
   and(int k=0;k<NUM;k++) {
      min:Z;
      X[k] == ceil(X[k]);
   }

   printf("The progressive roundoff
result is %f\n",Z);
}
```

Conditional Disjunction

- The usual disjunction of programming languages
- C and Lisp,
- intuitionnistic disjunction - Brouwer

```
drop_add()
{
   and(int i=0;i<NUM;i++){
      c_either X[i] == 0;
      or X[i] == 1;
   }
   printf("DA yields %f\n",Z);
}
```

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.

Randomizing

```
#define ITERATIONS 10
int rand;
extern void random(int &, int);


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[NUM];

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

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

Using rannyu

```
find_next_random_sequence(int q[])
{
int temp;

   and(int i=0;i<NUM;i++)
      q[i] = i;
   and(int i=0;i<NUM;i++) {
      random(rand,NUM-i);
   // random between 0 and NUM-i-1
      temp = q[i+rand];
      q[i+rand] = q[i];
      q[i] = temp;
   }
}
```

```
randomized_drop_add()
{
double best;

   best = 1e5;
   and(int k=0;k<ITERATIONS;k++)
      if
      not random_drop_add(k,best);
      then
      printf("Iter %d failed\n",k);

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

random_drop_add(int k, double & best)
{
int q[NUM];
   find_next_random_sequence(q);
   Z <= best-1;
   and(int k=0;k<NUM;k++) {
      c_either X[q[k]] == 0;
      or X[q[k]] == 1;
   }
   printf("The best is now %f\n",Z);
   best = wp(Z);
}
```

One Ply Lookahead

```
one_ply_lookahead()
{
double in,out;

   and(int i=0;i<NUM;i++) {
      min: Z;
      if integral(X[i]);
      then X[i] == wp(X[i]);
      else {
         eval(in,out,i);
         if in < out;
         then X[i] == 1;
         else X[i] == 0;
      }
   }

   printf("One-ply lookahead
                     found %f\n",Z);
}
```

The Evaluation Function

```
eval(double & in, double & out,
                                    int i)
{
   out = NUM+1;

   if not {
      X[i] == 1;
      min: Z;
      in = wp(Z);
   } then ;

   if not {
      X[i] == 0;
      min: Z;
      out = wp(Z);
   } then ;
}
```

Persistent Disjunction

- **mathematical disjunction**

Example: Law of the Excluded Middle

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!

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);
```

The Canonical NP-Complete Problem

- **CNF Structure**

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

Vanilla Branch and Bound

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

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

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


#define fail 0==1
determine(int & bv)
{
   and(int n=0;n<NUM;n++)
      if not integral(X[n]);
      then {
         bv = n;
         return;
      }
   fail;
}
#undef fail
```

Dualized Branching

There is duality here

- **the X[i] represent subsets**
- **the expressions**

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

represent elements

- **If we give this model to a MIP solver with the stipulation that the X[i] 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.

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

dual_branching()
{
int bv;

   printf("Branch on elements\n");
   set_machinery();

   find_min: Z;
   subject_to
      and(;;){
         min: Z;
         absgap(1.0);
         if determine_element(bv);
         then tend_to_element(bv,X);
         else break;
      }
   printf("Dualized branching found
           %f in %d nodes\n",Z,cnt);
}
```

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 {
      and(int i=sz[e];i<sz[e+1];i++)
         wp(X[indices[i]])< 1.0;
      sz[e+1]-sz[e] < gauge;
      }
      then {
         bv = e;
         gauge = sz[e+1]-sz[e];
      }
   bv < ELMS;
}
```

```
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;
   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) ;
}
```

IDA* - Least Discrepancy Search

```
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;
            compute_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\n",Z);
   printf("node count is %d\n",cnt);
}
```

The Penalty Function

```
initialize(int f[],int k)
{
   and(int i=0;i<=k;i++)
      f[i] = 0;
   and(int i=k+1;i<NUM;i++)
      f[i] = NUM - k/2;
}
```

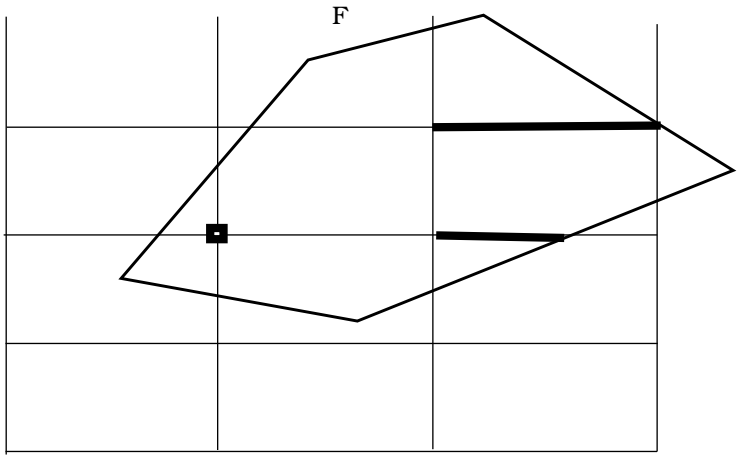
Disjunctive Programming

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;* context permitting, we simply say 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. In many situations, all the N_j will be equal, and so the problem can be denoted

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



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

Job Shop

N machines, N jobs each composed of N tasks. Each of the tasks for a job must go on a certain machine and takes a certain amount of time. Each task in a jobmust run on a different machine, and the tasks must be performed in order for each job. The goal is to find a schedule that minimizes finishing time.

- **Notoriously difficult**

- **A natural disjunctive program**

Constraint Propagation

GERALD + DONALD = ROBERT

Drives MIP solvers crazy
Needs more constraint propagation

Sources

Optimization and Computational Logic

Wiley, Discrete Mathematics and Optimization Series, to appear

2LP: Logic Programming and Linear Programming

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

For 2LP for Sparc, for Windows, for LINUX, for RS600

For .ps files of papers and book chapters

ftp as anonymous to acc5.its.brooklyn.cuny.edu

http://acc6.its.brooklyn.cuny.edu:5000/~lbslab