Modeling and Solving Nontraditional Optimization Problems

Session 1b: Current Features

Robert Fourer

Industrial Engineering & Management Sciences
Northwestern University

AMPL Optimization LLC

4er@northwestern.edu — 4er@ampl.com

Chiang Mai University International Conference
Workshop
Chiang Mai, Thailand — 4-5 January 2011

Session 1b: Current Features

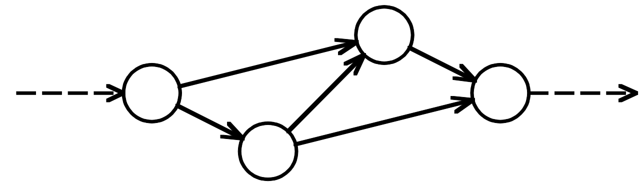
Focus

- ❖ Simple nontraditional features already implemented
- ❖ Incorporation in the AMPL language
- ❖ Handling by solvers

Topics

- ❖ Networks
- ❖ Separable piecewise-linear terms
- ❖ Disconnected variable domains
  - ∗ union of points
  - ∗ union of points & intervals
  - ∗ zero or interval
- ❖ Implications
  - ∗ indicator constraints
  - ∗ piecewise-nonlinear terms

Network Flows



Definition

- Minimize total cost of flows
- Subject to
  - ∗ Flow balance at nodes
  - ∗ Flow limits on arcs

Representations

- Algebraic variable-constraint
  - ∗ define arc variables
  - ∗ define node flow balances using variables
- Network node-arc
  - ∗ define network nodes
  - ∗ define arcs connecting the nodes

. . . arguably more natural

Generic Model

Variable-constraint formulation

```
set CITIES;
set LINKS within (CITIES cross CITIES);

param supply {CITIES} >= 0;   # amounts available at cities
param demand {CITIES} >= 0;   # amounts required at cities

   check: sum {i in CITIES} supply[i] = sum {j in CITIES} demand[j];

param cost {LINKS} >= 0;      # shipment costs/1000 packages
param capacity {LINKS} >= 0;  # max packages that can be shipped

var Ship {(i,j) in LINKS} >= 0, <= capacity[i,j];
                              # packages to be shipped

minimize Total_Cost:
   sum {(i,j) in LINKS} cost[i,j] * Ship[i,j];

subject to Balance {k in CITIES}:
   supply[k] + sum {(i,k) in LINKS} Ship[i,k]
      = demand[k] + sum {(k,j) in LINKS} Ship[k,j];

                              # supply plus total flow in equals
                              # demand plus total flow out
```

Generic Model

Node-arc formulation

```
set CITIES;
set LINKS within (CITIES cross CITIES);

param supply {CITIES} >= 0;   # amounts available at cities
param demand {CITIES} >= 0;   # amounts required at cities

   check: sum {i in CITIES} supply[i] = sum {j in CITIES} demand[j];

param cost {LINKS} >= 0;      # shipment costs/1000 packages
param capacity {LINKS} >= 0;  # max packages that can be shipped

minimize Total_Cost;

node Balance {k in CITIES}: net_in = demand[k] - supply[k];

arc Ship {(i,j) in LINKS} >= 0, <= capacity[i,j],
   from Balance[i], to Balance[j], obj Total_Cost cost[i,j];
```

AMPL Applications (1)

Product distribution (nodes)

```
minimize cost;

node RT:  rtmin <= net_out <= rtmax;

                        # Source of all regular-time crews

node OT:  otmin <= net_out <= otmax;

                        # Source of all overtime hours

node P_RT {fact};       # Sources of regular-time crews at factories
node P_OT {fact};       # Sources of overtime hours at factories

node M {prd,fact};      # Sources of manufacturing

node D {prd,dctr};      # Sources of distribution:

node W {p in prd, w in whse}:  net_in = dem[p,w];

                        # Locations of warehousing
```

AMPL Applications (1)

Product distribution (arcs)

```
arc Work_RT {f in fact}
   from RT  to P_RT[f]  >= rmin[f],  <= rmax[f];

                        # Regular-time crews allocated to each factory

arc Work_OT {f in fact}
   from OT  to P_OT[f]  >= omin[f],  <= omax[f];

                        # Overtime hours allocated to each factory

arc Manu_RT {p in prd, f in fact: rpc[p,f] <> 0} >= 0
   from P_RT[f]  to M[p,f] (dp[f] * hd[f] / pt[p,f])
   obj cost (rpc[p,f] * dp[f] * hd[f] / pt[p,f]);

                        # Regular-time crews allocated to
                        # manufacture of each product at each factory

arc Manu_OT {p in prd, f in fact: opc[p,f] <> 0} >= 0
   from P_OT[f]  to M[p,f] (1 / pt[p,f])  obj cost (opc[p,f] / pt[p,f]);

                         # Overtime hours allocated to
                         # manufacture of each product at each factory
```

AMPL Applications (1)

Product distribution (arcs)

```
arc Prod_L {p in prd, f in fact} >= 0
   from M[p,f]  to W[p,f];

                       # Manufacture of each product at each factory
                       # to satisfy local demand, in 1000s of units

arc Prod_D {p in prd, f in fact} >= 0
   from M[p,f]  to D[p,f]; \\[\Sa]

                       # Manufacture of each product at each factory,
                       # for distribution elsewhere, in 1000s of units

arc Ship {p in prd, (d,w) in rt} >= 0
   from D[p,d]  to W[p,w]  obj cost (sc[d,w] * wt[p]);

                       # Shipments of each product on each allowed route

arc Trans {p in prd, d in dctr} >= 0
   from W[p,d]  to D[p,d]  obj cost (tc[p]);

                        # Transshipments of each product at each
                        # distribution center
```

AMPL Applications (2)

Train car allocation (nodes)

```
minimize cars;           # Number of cars in the system:
                         # sum of unused cars and cars in trains during
                         # the last time interval of the day

minimize miles;          # Total car-miles run by
                         # all scheduled trains in a day

node N {cities,times};   # For every city and time:
                         # unused cars in present interval will equal
                         # unused cars in previous interval,
                         # plus cars just arriving in trains,
                         # minus cars just leaving in trains
```

AMPL Applications (2)

Train car allocation (arcs)

```
arc U {c in cities, t in times} >= 0
      from N[c,t]  to N[c,next(t)]
      obj {if t = last} cars 1;
                  # U[c,t] is the number of unused cars stored
                  # at city c in the interval beginning at time t

arc X {(c1,t1,c2,t2) in schedule}
      >= low[c1,t1,c2,t2]  <= high[c1,t1,c2,t2]
      from N[c1,t1]  to N[c2,t2]
      obj {if t2 < t1} cars 1
      obj miles distance[c1,c2];
                  # X[c1,t1,c2,t2] is the number of cars assigned
                  # to the scheduled train that leaves c1 at t1
                  # and arrives in c2 at t2
```

Conversion for Solver

Equivalent linear program

- Generate variables & constraints
- Mark as network
  - facilitate solution by specialied network simplex method

Extensions

- Multipliers
  - gains or losses
  - change of units
- Network embedded in larger model
  - side constraints
  - side variables

Conversion for Solver *(cont'd)*

Train car allocation (simplex solve)

```
ampl: model train2.mod;
ampl: data train2.dat;

ampl: option solver cplexamp;
ampl: solve;

Presolve eliminates 219 constraints and 1 variable.
Adjusted problem:
410 variables, all linear
192 constraints, all linear; 820 nonzeros
2 objectives, all linear; 235 nonzeros.

CPLEX 12.2.0.0: LP Presolve eliminated 0 rows and 50 columns.
Reduced LP has 85 rows, 253 columns, and 506 nonzeros.

optimal solution; objective 129
57 dual simplex iterations (0 in phase I)
```

Conversion for Solver *(cont'd)*

Train car allocation (network simplex solve)

```
ampl: model train2.mod;
ampl: data train2.dat;

ampl: option solver cplexamp:
ampl: option cplex_options 'netopt 2';

ampl: solve;

Presolve eliminates 219 constraints and 1 variable.
Adjusted problem:
410 variables, all linear
192 constraints, all linear; 820 nonzeros
2 objectives, all linear; 235 nonzeros.

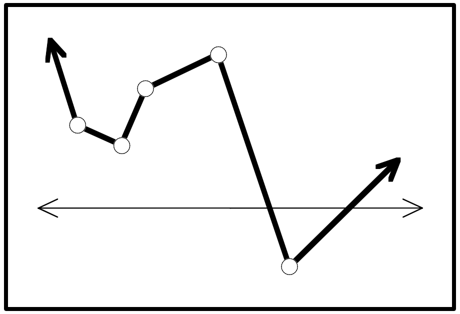
CPLEX 12.2.0.0: netopt 2
CPLEX 12.2.0.0: optimal solution; objective 129

Network extractor found 192 nodes and 410 arcs.
333 network simplex iterations.
```

Piecewise-Linear

Definition

- Function of one variable
- Linear on intervals
- Continuous



Issues

- Describing the function
  - choice of specification
  - syntax in the modeling language
- Communicating the function to a solver
  - direction description
  - transformation to linear or linear-integer

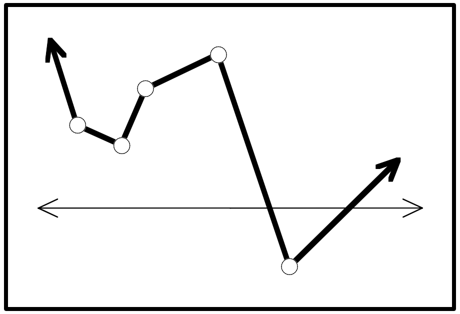
Specification

Possibilities

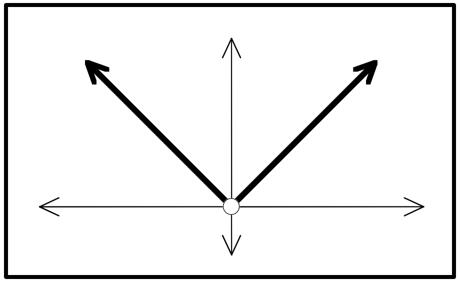
- List of breakpoints and either:
  - change in slope at each breakpoint
  - value of the function at each breakpoint
- List of slopes and either:
  - distance between breakpoints bounding each slope
  - value of intercept associated with each slope
- **Lists of breakpoints and slopes**

Also needed in some cases

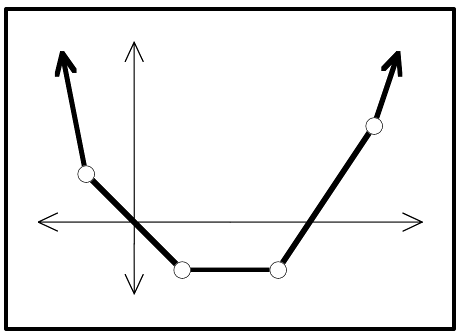
- One particular breakpoint
- One particular slope
- **Value at one particular point**



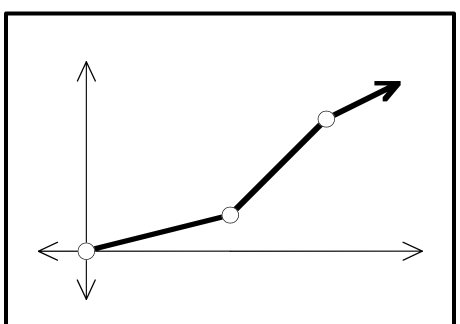
AMPL Specification: Examples



```
<<0; -1,1>> x[j]
```



```
<<-1,1,3,5; -5,-1,0,1.5,3>> x[j]
```



```
<<3,5; 0.25,1.00,0.50>> x[j]
```

AMPL Specification: Syntax

General forms

- <breakpoint-list; slope-list> variable
  - Zero at zero
  - Bounds on variable specified independently
- <breakpoint-list; slope-list> (variable, zero-point)
  - Zero at *zero-point*
- <breakpoint-list; slope-list> variable + constant
  - Has value *constant* at zero

Breakpoint & slope list forms

- Simple list
  - `<<lim1[i,j],lim2[i,j]; r1[i,j],r2[i,j],r3[i,j]>>`
- Indexed list
  - ```
    << {k in 1..nlim[i,j]} lim[i,j,k];
       {k in 1..nlim[i,j]+1} r[i,j,k]>>
    ```
    ```

AMPL Applications (1)

Design of a planar structure

```
var Force {bars};    # Forces on bars:
                     # positive in tension, negative in compression

minimize TotalWeight:  (density / yield_stress) *

   sum {(i,j) in bars} length[i,j] * <<0; -1,+1>> Force[i,j];

                     # Weight is proportional to length
                     # times absolute value of force

subject to Xbal {k in joints: k <> fixed}:

     sum {(i,k) in bars} xcos[i,k] * Force[i,k]
   - sum {(k,j) in bars} xcos[k,j] * Force[k,j] = xload[k];

subject to Ybal {k in joints: k <> fixed and k <> rolling}:

     sum {(i,k) in bars} ycos[i,k] * Force[i,k]
   - sum {(k,j) in bars} ycos[k,j] * Force[k,j] = yload[k];

                     # Forces balance in
                     # horizontal and vertical directions
```

Piecewise-Linear

AMPL Applications (2)

Data fitting for credit scoring

```
var Wt_const;                # Constant term in computing all scores

var Wt {j in factors} >= if wttyp[j] = 'pos' then 0 else -Infinity
                      <= if wttyp[j] = 'neg' then 0 else +Infinity;

                             # Weights on the factors

var Sc {i in people};        # Scores for the individuals

minimize Penalty:            # Sum of penalties for all individuals

   Gratio * sum {i in Good} << {k in 1..Gpce-1} if Gbktyp[k] = 'A'

                                   then Gbkfac[k]*app_amt

                                   else Gbkfac[k]*bal_amt[i];

                               {k in 1..Gpce} Gslope[k] >> Sc[i] +

   Bratio * sum {i in Bad}  << {k in 1..Bpce-1} if Bbktyp[k] = 'A'

                                   then Bbkfac[k]*app_amt

                                   else Bbkfac[k]*bal_amt[i];

                               {k in 1..Bpce} Bslope[k] >> Sc[i];
```

Conversion for Solver: Example

Transportation costs

```
param rate1 {i in ORIG, j in DEST} >= 0;
param rate2 {i in ORIG, j in DEST} >= rate1[i,j];
param rate3 {i in ORIG, j in DEST} >= rate2[i,j];

param limit1 {i in ORIG, j in DEST} >= 0;
param limit2 {i in ORIG, j in DEST} >= limit1[i,j];

var Trans {ORIG,DEST} >= 0;

minimize Total_Cost:

   sum {i in ORIG, j in DEST}
      <<limit1[i,j], limit2[i,j];
        rate1[i,j], rate2[i,j], rate3[i,j]>> Trans[i,j];
```

Minimizing Convex Costs

Equivalent linear program

```
ampl: model trpl2.mod; data trpl.dat; solve;

Substitution eliminates 15 variables.
21 piecewise-linear terms replaced by 35 variables and 15 constraints.

Adjusted problem:
41 variables, all linear
10 constraints, all linear; 82 nonzeros
1 linear objective; 41 nonzeros.

CPLEX 10.1.0: optimal solution; objective 199100
12 dual simplex iterations (0 in phase I)

ampl: display Trans;

:       DET   FRA   FRE   LAF   LAN   STL   WIN   :=

CLEV    500     0   200   500   500    500   400
GARY      0     0   900   300     0    200     0
PITT    700   900     0   200   100   1000     0 ;
```

Minimizing Non-Convex Costs

Equivalent mixed-integer program

```
model trpl3.mod; data trpl.dat; solve;

Substitution eliminates 18 variables.
21 piecewise-linear terms replaced by 87 variables and 87 constraints.

Adjusted problem:
90 variables:
        41 binary variables
        49 linear variables
79 constraints, all linear; 251 nonzeros
1 linear objective; 49 nonzeros.

CPLEX 10.1.0: optimal integer solution; objective 256100
189 MIP simplex iterations
144 branch-and-bound nodes

ampl: display Trans;

:       DET    FRA   FRE    LAF    LAN   STL    WIN   :=

CLEV    1200     0      0   1000     0      0   400
GARY       0     0   1100      0   300      0     0
PITT       0   900      0      0   300   1700     0
```

Minimizing Non-Convex Costs *(cont'd)*

. . . with SOS type 2 markers in output file

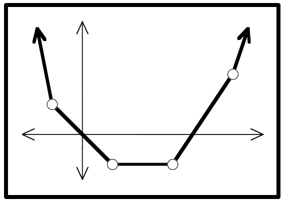
```
S0 87 sos
 3 16
49 18
 4 16
50 18  ...

S1 64 sos
10 19
11 18
12 18
14 35  ...

S4 46 sosref
3 -501
4  751
5 -501
6  500  ...
```

Conversion for Solver: Principles

Equivalent linear program if . . .



- Objective
  - minimizes convex (increasing slopes) *or*
  - maximizes concave (decreasing slopes)
- Constraints expressions
  - convex and on the left-hand side of a ≤ constraint
  - convex and on the right-hand side of a ≥ constraint
  - concave and on the left-hand side of a ≥ constraint
  - concave and on the right-hand side of a ≤ constraint

Equivalent mixed-integer program otherwise

- At least one binary variable per piece
- Enhanced branching in solver
  - "special ordered sets of type 2"

Discrete Variable Domains

Continuous domain

```
var Buy {j in FOOD} >= 0;
```

Semi-continuous domain

```
var Buy {j in FOOD} in {0} union interval[30,40];
```

Discrete domain

```
var Buy {j in FOOD} in {1,2,5,10,20,50};
```

. . . many generalizations possible

Semi-Continuous Domain

Continuous

```
CPLEX 10.1.0: optimal solution; objective 88.2
1 dual simplex iterations (0 in phase I)

ampl: display Buy;

BEEF  0        FISH  0         MCH 46.6667    SPG  0
 CHK  0         HAM  0         MTL  0         TUR  0
```

Semi-Continuous

```
CPLEX 10.1.0: optimal integer solution; objective 116.4

65 MIP simplex iterations
27 branch-and-bound nodes

ampl: display Buy;

BEEF  0        FISH  0         MCH 30         SPG  0
 CHK  0         HAM  0         MTL 30         TUR  0
```

Semi-Continuous Domain *(cont'd)*

Converted to MIP with extra variables . . .

```
minimize Total_Cost:
95.7*(Buy[BEEF]+lambdaL) + 127.6*(Buy[BEEF]+lambdaU) +
77.7*(Buy[CHK]+lambdaL) + 103.6*(Buy[CHK]+lambdaU) +
68.7*(Buy[FISH]+lambdaL) + 91.6*(Buy[FISH]+lambdaU) +
86.7*(Buy[HAM]+lambdaL) + 115.6*(Buy[HAM]+lambdaU) +
56.7*(Buy[MCH]+lambdaL) + 75.6*(Buy[MCH]+lambdaU) +
59.7*(Buy[MTL]+lambdaL) + 79.6*(Buy[MTL]+lambdaU) +
59.7*(Buy[SPG]+lambdaL) + 79.6*(Buy[SPG]+lambdaU) +
74.7*(Buy[TUR]+lambdaL) + 99.6*(Buy[TUR]+lambdaU);

subject to Diet['A']:
700 <= 1800*(Buy[BEEF]+lambdaL) + 2400*(Buy[BEEF]+lambdaU) +
240*(Buy[CHK]+lambdaL) + 320*(Buy[CHK]+lambdaU) +
240*(Buy[FISH]+lambdaL) + 320*(Buy[FISH]+lambdaU) +
1200*(Buy[HAM]+lambdaL) + 1600*(Buy[HAM]+lambdaU) +
450*(Buy[MCH]+lambdaL) + 600*(Buy[MCH]+lambdaU) +
2100*(Buy[MTL]+lambdaL) + 2800*(Buy[MTL]+lambdaU) +
750*(Buy[SPG]+lambdaL) + 1000*(Buy[SPG]+lambdaU) +
1800*(Buy[TUR]+lambdaL) + 2400*(Buy[TUR]+lambdaU) <= 10000;  ...
```

Semi-Continuous Domain *(cont'd)*

and extra constraints

```
subject to (Buy[BEEF]+ldef):
-(Buy[BEEF]+b) + (Buy[BEEF]+lambdaL) + (Buy[BEEF]+lambdaU) = 0;

subject to (Buy[CHK]+ldef):
-(Buy[CHK]+b) + (Buy[CHK]+lambdaL) + (Buy[CHK]+lambdaU) = 0;

subject to (Buy[FISH]+ldef):
-(Buy[FISH]+b) + (Buy[FISH]+lambdaL) + (Buy[FISH]+lambdaU) = 0;  ...
```

. . . *with extra binary variables*

Discrete Domain

Continuous

```
CPLEX 10.1.0: optimal solution; objective 88.2
1 dual simplex iterations (0 in phase I)

ampl: display Buy;

BEEF  0        FISH  0         MCH 46.6667    SPG  0
 CHK  0         HAM  0         MTL   0        TUR  0
```

Discrete

```
CPLEX 10.1.0: optimal integer solution; objective 95.49

47 MIP simplex iterations
8 branch-and-bound nodes

ampl: display Buy;

BEEF  1   FISH  1    MCH 10    SPG  5
 CHK 20    HAM  1    MTL  2    TUR  1
```

Discrete Domain *(cont'd)*

Converted to MIP with extra binary variables . . .

```
minimize Total_Cost:
3.19*(Buy[BEEF]+b)[0] + 6.38*(Buy[BEEF]+b)[1] +
15.95*(Buy[BEEF]+b)[2] + 31.9*(Buy[BEEF]+b)[3] +
63.8*(Buy[BEEF]+b)[4] + 159.5*(Buy[BEEF]+b)[5] +
2.59*(Buy[CHK]+b)[0] + 5.18*(Buy[CHK]+b)[1] +
12.95*(Buy[CHK]+b)[2] + 25.9*(Buy[CHK]+b)[3] +
51.8*(Buy[CHK]+b)[4] + 129.5*(Buy[CHK]+b)[5] + ...

subject to Diet['A']:
700 <= 60*(Buy[BEEF]+b)[0] + 120*(Buy[BEEF]+b)[1] +
300*(Buy[BEEF]+b)[2] + 600*(Buy[BEEF]+b)[3] +
1200*(Buy[BEEF]+b)[4] + 3000*(Buy[BEEF]+b)[5] +
8*(Buy[CHK]+b)[0] + 16*(Buy[CHK]+b)[1] + 40*(Buy[CHK]+b)[2] +
80*(Buy[CHK]+b)[3] + 160*(Buy[CHK]+b)[4] + 400*(Buy[CHK]+b)[5] + ...
```

Discrete Domain *(cont'd)*

and SOS type 1 constraints . . .

```
subject to (Buy[BEEF]+sos1):
(Buy[BEEF]+b)[0] + (Buy[BEEF]+b)[1] + (Buy[BEEF]+b)[2] +
(Buy[BEEF]+b)[3] + (Buy[BEEF]+b)[4] + (Buy[BEEF]+b)[5] = 1;

subject to (Buy[CHK]+sos1):
(Buy[CHK]+b)[0] + (Buy[CHK]+b)[1] + (Buy[CHK]+b)[2] +
(Buy[CHK]+b)[3] + (Buy[CHK]+b)[4] + (Buy[CHK]+b)[5] = 1;  ...
```

Discrete Domain *(cont'd)*

with SOS type 1 markers in output file

```
S0 48 sos
0 20
1 20
2 20
3 20
4 20
5 20
6 36
7 36  ...

S4 48 sosref
0 1
1 2
2 5
3 10
4 20
5 50
6 1
7 2  ...
```

Conversion for Solver: Principles

General case

- Arbitrary union of points and intervals
- Auxiliary binary variable for each point or interval
- 3 auxiliary constraints for each variable

Union of points

- Auxiliary binary variable for each point
- Auxiliary constraint for each variable
- Enhanced branching in solver
  - “special ordered sets of type 1”

Zero union interval (semi-continuous)

- Auxiliary binary variable for each variable
- 2 auxiliary constraints for each variable
- Enhanced branching in solver

Implications

General possibilities

- Conditional expression
- Conditional constraint
- Conditional command

AMPL syntax choices

- `if` *condition* `then` *expr1* `else` *expr2*
- *condition* ==> *constraint1* `else` *constraint2*
  - also <== and <==>
- `if` *condition* `then` {*commands*} `else` {*commands*}

Currently supported forms

- Nonlinear if-then-else
- CPLEX indicator constraints

Nonlinear *if-then-else*

More stable expression near zero

```
subject to logRel {j in 1..N}:
   (if X[j] < -delta || X[j] > delta
      then log(1+X[j]) / X[j] else 1 - X[j] / 2) <= logLim;
```

CPLEX Indicator Constraints

Indicator constraints

- (*binary variable* = 0) implies *constraint*
- (*binary variable* = 1) implies *constraint*

. . . ***handled directly by solver***

AMPL "implies" operator

- Use `==>` for "implies"
- Also recognize an `else` clause
- Similarly define `<==` and `<==>`
  - `if-then-else` expressions & statements as before

Example 1

Multicommodity flow with fixed costs

```
set ORIG;   # origins
set DEST;   # destinations
set PROD;   # products

param supply {ORIG,PROD} >= 0;  # amounts available at origins
param demand {DEST,PROD} >= 0;  # amounts required at destinations
param limit {ORIG,DEST} >= 0;

param vcost {ORIG,DEST,PROD} >= 0; # variable shipment cost on routes
param fcost {ORIG,DEST} > 0;       # fixed cost on routes

var Trans {ORIG,DEST,PROD} >= 0;   # actual units to be shipped
var Use {ORIG, DEST} binary;       # = 1 iff link is used

minimize total_cost:
   sum {i in ORIG, j in DEST, p in PROD} vcost[i,j,p] * Trans[i,j,p]
 + sum {i in ORIG, j in DEST} fcost[i,j] * Use[i,j];
```

Example 1 *(cont'd)*

Conventional constraints

```
subject to Supply {i in ORIG, p in PROD}:
   sum {j in DEST} Trans[i,j,p] = supply[i,p];

subject to Demand {j in DEST, p in PROD}:
   sum {i in ORIG} Trans[i,j,p] = demand[j,p];

subject to Multi {i in ORIG, j in DEST}:
   sum {p in PROD} Trans[i,j,p] <= limit[i,j] * Use[i,j];
```

```
subject to Supply {i in ORIG, p in PROD}:
   sum {j in DEST} Trans[i,j,p] = supply[i,p];

subject to Demand {j in DEST, p in PROD}:
   sum {i in ORIG} Trans[i,j,p] = demand[j,p];

subject to UseDefinition {i in ORIG, j in DEST, p in PROD}:
   Trans[i,j,p] <= min(supply[i,p], demand[j,p]) * Use[i,j];
```

Example 1 *(cont'd)*

User cuts

```
subject to Multi {i in ORIG, j in DEST}:
   sum {p in PROD} Trans[i,j,p] <= limit[i,j] * Use[i,j];

subject to UseDefinition {i in ORIG, j in DEST, p in PROD}:
   Trans[i,j,p] <= min(supply[i,p], demand[j,p]) * Use[i,j];
```

Example 1 *(cont'd)*

Indicator constraint formulations

```
subject to DefineUsedA {i in ORIG, j in DEST}:
   Use[i,j] = 0 ==> sum {p in PROD} Trans[i,j,p] = 0;
```

```
subject to DefineUsedB {i in ORIG, j in DEST, p in PROD}:
   Use[i,j] = 0 ==> Trans[i,j,p] = 0;
```

```
subject to DefineUsedC {i in ORIG, j in DEST}:
   Use[i,j] = 0 ==> sum {p in PROD} Trans[i,j,p] = 0
               else sum {p in PROD} Trans[i,j,p] <= limit[i,j];
```

Example 1 *(cont'd)*

Results for 3 origins, 7 destinations, 3 products

	iters	nodes	cuts used
no cuts	374	79	
all cuts	317	39	
user cuts	295	42	18
indic A	355	77	
indic B	406	56	
indic C	277	57	

Example 2

Assignment to groups with "no one isolated"

```
var Lone {(i1,i2) in ISO, j in REST} binary;

param give {ISO} default 2;
param giveTitle {TITLE} default 2;
param giveLoc {LOC} default 2;

param upperbnd {(i1,i2) in ISO, j in REST} :=
   min (ceil((number2[i1,i2]/card {PEOPLE}) * hiDine[j]) + give[i1,i2],
        hiTargetTitle[i1,j] + giveTitle[i1],
        hiTargetLoc[i2,j] + giveLoc[i2], number2[i1,i2]);

subj to Isolation1 {(i1,i2) in ISO, j in REST}:
   Assign2[i1,i2,j] <= upperbnd[i1,i2,j] * Lone[i1,i2,j];

subj to Isolation2a {(i1,i2) in ISO, j in REST}:
   Assign2[i1,i2,j] >= Lone[i1,i2,j];

subj to Isolation2b {(i1,i2) in ISO, j in REST}:
   Assign2[i1,i2,j] +
      sum {ii1 in ADJACENT[i1]: (ii1,i2) in TYPE2} Assign2[ii1,i2,j]
         >= 2 * Lone[i1,i2,j];
```

Example 2

Same using indicator constraints

```
var Lone {(i1,i2) in ISO, j in REST} binary;

subj to Isolation1 {(i1,i2) in ISO, j in REST}:
   Lone[i1,i2,j] = 0 ==> Assign2[i1,i2,j] = 0;

subj to Isolation2b {(i1,i2) in ISO, j in REST}:
   Lone[i1,i2,j] = 1 ==> Assign2[i1,i2,j] +
     sum {ii1 in ADJACENT[i1]: (ii1,i2) in TYPE2} Assign2[ii1,i2,j] >= 2;
```

Example 3

Workforce planning

```
var LayoffCost {m in MONTHS} >=0;

subj to LayoffCostDefn1 {m in MONTHS}:
   LayoffCost[m]
      <= snrLayOffWages * 31 * maxNbrSnrEmpl * (1 - NoShut[m]);

subj to LayoffCostDefn2a {m in MONTHS}:
   LayoffCost[m] - snrLayOffWages * ShutdownDays[m] * maxNbrSnrEmpl
      <= maxNbrSnrEmpl * 2 * dayAvail[m] * snrLayOffWages * NoShut[m];

subj to LayoffCostDefn2b {m in MONTHS}:
   LayoffCost[m] - snrLayOffWages * ShutdownDays[m] * maxNbrSnrEmpl
      >= -maxNbrSnrEmpl * 2 * dayAvail[m] * snrLayOffWages * NoShut[m];
```

Example 3

Same using indicator constraints

```
var LayoffCost {m in MONTHS} >=0;

subj to LayoffCostDefn1 {m in MONTHS}:
   NoShut[m] = 1 ==> LayoffCost[m] = 0;

subj to LayoffCostDefn2 {m in MONTHS}:
   NoShut[m] = 0 ==> LayoffCost[m] =
      snrLayoffWages * ShutdownDays[m] * maxNumberSnrEmpl;
```

Example 4

Standard mixed-integer formulation

```
param least_assign >= 0;

var Work {SCHEDS} >= 0 integer;
var Use  {SCHEDS} >= 0 binary;

subject to Least_Use1 {j in SCHEDS}:
   Work[j] >= least_assign * Use[j];

subject to Least_Use2 {j in SCHEDS}:
   Work[j] <= (max {i in SHIFT_LIST[j]} required[i]) * Use[j];
```

Example 4 *(cont'd)*

Formulation using variable-domain specification

```
param least_assign >= 0;

var Work {j in SCHEDS} integer, in {0} union
   interval [least_assign, (max {i in SHIFT_LIST[j]} required[i])];
```

Example 4 *(cont'd)*

Formulation using "implies" operator

```
param least_assign >= 0;

var Work {SCHEDS} >= 0 integer;
var Use  {SCHEDS} >= 0 binary;

subject to Least_Use1_logical {j in SCHEDS}:
   Use[j] = 1 ==> Work[j] >= least_assign;

subject to Least_Use2_logical {j in SCHEDS}:
   Use[j] = 0 ==> Work[j] = 0;
```

```
param least_assign >= 0;

var Work {SCHEDS} >= 0 integer;
var Use  {SCHEDS} >= 0 binary;

subject to Least_Use_logical {j in SCHEDS}:
   Use[j] = 1 ==> least_assign <= Work[j] else Work[j] = 0;
```