Modeling in GNU MathProg language - a short introduction

Paweł Zieliński

Institute of Mathematics and Computer Science,
Wrocław University of Technology, Poland

Wrocław University of Technology

General informations

- Webpages:
  - My home page: http://www.im.pwr.wroc.pl/ ~pziel/
- The GNU Linear Programming Kit (GLPK): `glpsol` solver plus the GNU MathProg modeling language The software is free and can be downloaded:
  - GUSEK (GLPK Under Scite Extended Kit): The GLPK + IDE - Windows version: http://gusek.sourceforge.net/,
  - GLPK (GNU Linear Programming Kit) for Windows (without IDE): http://gnuwin32.sourceforge.net/packages/glpk.htm,
  - GLPK (GNU Linear Programming Kit) sources (without IDE): http://www.gnu.org/software/glpk/glpk.html

Linear programming problem

$$\sum_{j=1}^{n} c_j x_j \to \min(\max) \qquad \text{(a linear objective function)}$$

$$\sum_{j=1}^{n} a_{ij} x_j = (\leq, \geq) b_i, \quad i = 1, \ldots, m \text{ (linear constraints)}$$

$$x_j \geq 0, \qquad j = 1, \ldots, n \text{ (nonnegative variables)}$$

- Parameters (input data):
  - c_j, $j = 1, \ldots, n$, objective function coefficients,
  - b_i, $i = 1, \ldots, m$, the right hand side vector coefficients,
  - a_{ij}, $i = 1, \ldots, m$, $j = 1, \ldots, n$, matrix coefficients.
- x_j, $j = 1, \ldots, n$, decision variables (output).

Linear programming problem (LP)

Example 1: Solve the following linear programming problem:

$$
\begin{aligned}
& 4x_1 + 5x_2 \to \max \\
& x_1 + 2x_2 \leq 40 \\
& 4x_1 + 3x_2 \leq 120 \\
& x_1 \geq 0, x_2 \geq 0
\end{aligned}
$$

```
/* decision variables*/
var x1 >= 0;
var x2 >=0;
/* Objective function */
maximize  label : 4*x1 +5*x2;
/* Constraints */

subject to label1:     x1 + 2*x2 <= 40;
s.t.           label2: 4*x1 + 3*x2 <= 120;

end;
```

Solve the above model (`feasible.mod`) in `glpsol`.

```
glpsol --model feasible.mod
glpsol --model feasible.mod --output feasible.txt
```

Linear programming problem

Example 2: Solve the following linear programming problem:

$$
\begin{aligned}
& 4x_1 + 2x_2 \rightarrow \max \\
& x_1 + x_2 = 40 \\
& x_1 + x_2 \geq 120 \\
& x_1 \geq 0, x_2 \geq 0
\end{aligned}
$$

The set of feasible solutions is empty - there is no a feasible solution.

```
/* The declaration of decision variables x1, x2  */
var x1 >= 0;
var x2 >=0;

/* Objective function */
maximize  ObjectiveFunctionLabel : 4*x1 +2*x2;
/* Constraints */
s.t.        label1:     x1 + x2       = 2;
s.t.        label2:     x1 + x2       >= 4;
end;
```

Solve the above model (`infeasible.mod`) in `glpsol`.

```
glpsol --model infeasible.mod
```

Linear programming problem

Exercise: Implement in `GNU MathProg` and try to solve the following linear programming model (surprise!):

$$
\begin{array}{rcrcl}
4x_1 & + & 2x_2 & \rightarrow & \max \\
3x_1 & + & 6x_2 & \geq & 18 \\
x_1 & - & 2x_2 & \leq & 4 \\
x_1 \geq 0, & & x_2 \geq 0 & &
\end{array}
$$

LP: a production planning problem

Example 3: (S.P. Bradley, A.C. Hax, T.L. Magnanti, *Applied Mathematical Programming*, 1977)

The Candid Camera Company manufactures three lines of cameras: the Cub, the Quickiematic and the VIP, whose contributions are \$3, \$9, and \$25, respectively. The distribution center requires that at least 250 Cubs, 375 Quickiematics, and 150 VIPs be produced each week. Each camera requires a certain amount of time in order to: (1) manufacture the body parts; (2) assemble the parts (lenses are purchased from outside sources and can be ignored in the production scheduling decision); and (3) inspect, test, and package the final product. The Cub takes 0.1 hours to manufacture, 0.2 hours to assemble, and 0.1 hours to inspect, test, and package. The Quickiematic needs 0.2 hours to manufacture, 0.35 hours to assemble, and 0.2 hours for the final set of operations. The VIP requires 0.7, 0.1, and 0.3 hours, respectively. In addition, there are 250 hours per week of manufacturing time available, 350 hours of assembly, and 150 hours total to inspect, test, and package. Maximize contribution.

LP: a production planning problem

Definitions of decision variables: *cub* - total number of the Cub produced, *quick* - total number of the Quickiematic produced, *vip* - total number of the VIP produced.

The constraints:

$$\begin{aligned}
0.1\,cub + 0.2\,quick + 0.7\,vip &\leq 250,\\
0.2\,cub + 0.35\,quick + 0.1\,vip &\leq 350,\\
0.1\,cub + 0.2\,quick + 0.3\,vip &\leq 150,\\
cub &\geq 250,\\
quick &\geq 375,\\
vip &\geq 150.
\end{aligned}$$

The objective function: $3\,cub + 9\,quick + 25\,vip \rightarrow \max$.

The first approach (old one) - a mixture of data and model

```
/* decision variables */

var cub   >=0;       #  the number of the Cub produced
var quick >=0;       #  the number of the Quickiematic produced
var vip   >=0;       #  the number of the VIP produced

/* objective function represents profit */

maximize profit: 3*cub + 9*quick + 25*vip;

/* constraints determine composition manufacturing cameras */

s.t. time_manufacture: 0.1*cub + 0.2*quick  +  0.7*vip <= 250;
s.t. time_assemble:    0.2*cub + 0.35*quick +  0.1*vip <= 350;
s.t. time_inspect:     0.1*cub + 0.2*quick +   0.3*vip <= 150;
s.t. requirements_cub:     cub                         >= 250;
s.t. requirements_quick:             quick             >= 375;
s.t. requirements_vip:                           vip   >= 150;

end;
```

```
glpsol --model camera.mod --output camera.txt
```

Wrocław University of Technology

Towards isolating the data from the model - arrays and sets

```
/* Arrays and sets */
/* enumerated set Cameras represents the set of cameras manufactured by the company */

set Cameras;

/* array of three decision variables: production['cub'], production['quick']
   and production['vip'] */

var production{Cameras} >=0;

/* objective function represents profit */
maximize profit: 3*production['cub'] + 9*production['quick'] + 25*production['vip'];

/* constraints determine composition manufacturing cameras */

s.t. man: 0.1*production['cub'] + 0.2*production['quick']   +0.7*production['vip'] <= 250;
s.t. ass:   0.2*production['cub'] + 0.35*production['quick'] +0.1*production['vip'] <= 350;
s.t. insp:  0.1*production['cub'] +0.2*production['quick']    +0.3*production['vip'] <= 150;

s.t. requirements_cub:     production['cub']                          >= 250;
s.t. requirements_quick:             production['quick']              >= 375;
s.t. requirements_vip:                        production['vip']      >= 150;

data;
/* Definition of the set Cameras */
set Cameras:= 'cub' 'quick' 'vip';
end;
```

Model: `camera_arrays.mod`

Towards isolating the data from the model - arrays and sets

- The declaration of the set of cameras manufactured by the company:
  `set Cameras;`
  The initialization of the set `Cameras`
  `set Cameras:= 'cub' 'quick' 'vip';`
- The declaration of array of three nonnegative decision variables indexed by `Cameras` (`production['cub']`, `production['quick']` and `production['vip']`):
  `var productionCameras >=0`
- Other examples:
  ```
  set Range:= 1..n;
  set MyIntegerSet:= 4 8 9 10 ;
  var array{1..m};
  ```

Isolating the data from the model - the data

```
/* Data section */
data;

/* Definitions of the sets */
set Cameras:= cub quick vip;
set Times:= manufacture assemble inspect;

/* The initialization of the parameters */

param profit:= cub    3 quick 9 vip      25;

param capacity:= manufacture 250 assemble      350 inspect           150;

param demand:=cub    250 quick 375 vip     150;

param consumption:   cub   quick vip:=
      manufacture 0.1 0.2    0.7
      assemble 0.2 0.35   0.1
      inspect 0.1 0.2    0.3;
end;
```

Model: `camera_isolation.mod`

Isolating the data from the model - the data

- Declaring and initializing one dimensional array of profit:

```
param profit{Cameras} >=0;
param profit:= cub 3 quick 9 vip 25;
```

- Declaring and initializing one dimensional array of amount of times:

```
param capacity{Times} >=0;
param capacity:= manufacture 250 assemble 350
inspect 150;
```

- Similarly declaring and initializing one dimensional array of the distribution center requirements.
- Declaring and initializing two dimensional array:

```
param consumption{Times,Cameras} >=0;
param  consumption:  cub  quick  vip:=
       manufacture   0.1  0.2    0.7
       assemble      0.2  0.35   0.1
       inspect       0.1  0.2    0.3;
```

Isolating the data from the model - the model

```
set Cameras;
set Times;

/* Parameters */

param profit{Cameras} >=0;        # one dimensional array of profit
param consumption{Times,Cameras} >=0;  # two dimensional array
param capacity{Times} >=0;       # one dimensional array of amount of times
param demand{Cameras} >=0;  # one dimensional array of the distribution center requirements

/* Variables */

var production{j in Cameras} >=demand[j]; # decision variables plus trivial bounds

/* objective function represents profit */

maximize Profit: sum{j in Cameras} profit[j]*production[j];

/* constraints determine composition manufacturing cameras */

s.t. time{i in Times}: sum{j in Cameras} consumption[i,j]*production[j] <=capacity[i];
```

Model: `camera_isolation.mod`

Aggregate operators and quantifiers

- The aggregate operator `sum` in the objective function:
  `sum{j in Cameras} profit[j]*production[j];`
  Other aggregate operators: `prod`, `max`, `min`.
- The universal quantifier: `time{i in Times}` - closely related constraints
  `s.t. time{i in Times}: sum{j in Cameras} ...;`
- The trivial bounds:
  `var production{j in Cameras} >=demand[j];`
- One may add the bounds to the constraints using universal quantifier:
  `requirements{j in Cameras}`
  `s.t. requirements{j in Cameras}:`
  `production[j]>=demand[j];`

Solving and checking the model

- Solving the model:

```
glpsol --model camera_isolation.mod
```

- Solving the model and writing results to the file:

```
glpsol --model camera_isolation.mod --output camera_isolation.txt
```

- Checking the model without solving it:

```
glpsol --check --model camera_isolation.mod
```

- Checking the model without solving it and writing the generated model to the file:

```
glpsol --check --model camera_isolation.mod --wcpxlp camera_isolation.lp
```

```
\* Problem: camera_isolation *\
Maximize
 Profit: + 3 production(cub) + 9 production(quick) + 25 production(vip)

Subject To
 time(manufacture): + 0.1 production(cub)
 + 0.2 production(quick) + 0.7 production(vip) <= 250
 time(assemble): + 0.2 production(cub)
 + 0.35 production(quick) + 0.1 production(vip) <= 350
 time(inspect): + 0.1 production(cub)
 + 0.2 production(quick) + 0.3 production(vip) <= 150

Bounds
 production(cub) >= 250
 production(quick) >= 375
 production(vip) >= 150
End
```

Displaying results, the data section in a separated file

- Displaying results:

```
maximize Profit: sum{j in Cameras} profit[j]*production[j];
s.t. time{i in Times}: sum{j in Cameras} consumption[i,j]*production[j] <=capacity[i];

solve; /* solve command is needed !!!*/

display production;

display '-----------more elegant way -------------';
display 'profit =', sum{j in Cameras} profit[j]*production[j];
display{j in Cameras}  production[j];
```

- the data section in a separated file `camera_isolation1.dat`

```
data;
set Times:= manufacture assemble inspect;
param: Cameras: profit demand := cub    3 250
                                 quick  9 375
                                 vip    25 150;
param capacity:= manufacture 250
                 assemble    350
                 inspect     150;
param consumption:  cub  quick vip:=
manufacture 0.1     0.2   0.7
assemble 0.2     0.35  0.1
inspect     0.1     0.2   0.3;
end;
```

- Solving the model: `glpsol --model camera_isolation1.mod --data camera_isolation1.dat`

Integer programming problem (IP)

$$\sum_{j=1}^{n} c_j x_j \to \min(\max) \qquad \text{(a linear objective function)}$$

$$\sum_{j=1}^{n} a_{ij} = (\leq, \geq) b_i, \qquad i = 1, \ldots, m \text{ (linear constraints)}$$

$$x_j \geq 0, \qquad j = 1, \ldots, n \text{ (nonnegative variables)}$$

$$x_j \text{ integer, (binary)} \qquad j = 1, \ldots, n$$

The integrality constraints on variables make the general integer programming problem NP-hard and thus very hard from computational point of view.
If there exist real nonnegative variables and integer variables in a model, then we call the problem the Mixed Integer programming Problem (MIP)

MIP=LP+IP

Mixed Integer programming Problem (MIP)

Example 4: Solve the following mixed integer programming problem:

$$
\begin{aligned}
&-3x_1 - 2x_2 + 10 \rightarrow \max \\
&x_1 - 2x_2 + x_3 = 2.5; \\
&2x_1 + x_2 + x_4 \geq 1.5 \\
&x_1, x_2, x_3, x_4 \geq 0 \\
&x_2, x_3 \text{ integer}
\end{aligned}
$$

```
var x1 >= 0;
var x4 >=0;
/* The declaration of nonnegative integer decision variables*/
var x2 integer >= 0;
var x3 integer >=0;
/* Objective function */
maximize  ObjectiveFunctionLabel : -3*x1 -2*x2+10;

/* Constraints */
s.t.       label1:     x1 - 2*x2 + x3      = 2.5;
s.t.       label2:   2*x1 + x2          +x4 >= 1.5;
end;
```

Solve the above model (`mip.mod`) in `glpsol`.

```
glpsol --model mip.mod
glpsol --model mip.mod --output mip.txt
```

Integer programming Problem (IP)

Example 5: Let $E = \{1, 2, \ldots, n\}$ be a given set of items. A nonnegative real cost c_i is associated with every item $i \in E$ and we wish to choose a subset $X \subseteq E$ that contains exactly p items, whose total cost $\sum_{i \in X} c_i$ is minimal.
The model has the following form:

$$\sum_{i=1}^{n} c_i x_i \to \min$$

$$\sum_{i=1}^{n} x_i = p$$

$$x_i \in \{0, 1\}, i = 1, \ldots, n$$

x_i is a binary decision variable that takes value 1 if and only if i-th item belongs to X.

Integer programming Problem (IP)

```
/* input data */
param n, integer, >= 1;      # the number of items
param p, integer, >= 1, <n; # the number of items for selecting
set E:={1..n};               # the set of items
param c{E} >=0;              # the costs of items
/* The variables */
var x{E} binary;

/* The objective function */
minimize TotalCost: sum{i in E} c[i]*x[i];
/* The constraint */
s.t. exact_p: sum{i in E} x[i] = p;
solve;

/* Displaying  results */
display 'solution X';
display{i in E: x[i]=1 }: x[i];
display 'total costs=',sum{i in E} c[i]*x[i];

/* Data section */
data;
param n:=10;
param p:=6;
param c:=[1] 3 [2] 2 [3] 6 [4] 3 [5] 9 [6] 5 [7] 8 [8]  1 [9] 2 [10] 6;

end;
```

Solve the above model (`selecting.mod`) in `glpsol`.

```
glpsol --model selecting.mod
```

Integer programming Problem (IP)

Example 6: The multidimensional zero-one knapsack problem can be described as follows: given two sets of n items and m knapsack constraints (or resources), for each item j a profit p_j is assigned and for each constraint i a consumption value r_{ij} is designated. The goal is to determine a set of items that maximizes the total profit, not exceeding the given constraint capacities c_i. The problem is a well-known NP-Hard combinatorial optimization problem. The multidimensional zero-one knapsack problem can modeled:

$$\sum_{j=1}^{n} p_j x_i \rightarrow \max$$

$$\sum_{j=1}^{n} r_{ij} x_j \leq c_i, \qquad i = 1, \ldots, m$$

$$x_j \in \{0, 1\}, \qquad j = 1, \ldots, n$$

$x_j = 1$ if and only if the j-th item is chosen.

Integer programming Problem (IP)

```
/* Parameters */
param n>0 integer; /* the number of items */
param m>0 integer; /* the number of resources */

/* Sets */
set Items:=1..n;
set Resources:=1..m;

/* parametry */

param capacity{Resources}>=0;  /* array  represents the capacity of the resources*/
param consumption{Resources,Items}>=0; /* the consumption of resource  by item */
param profit{Items}>=0;  /* array  the value of each item */

/* Decision variables */

/* variable */
var choose{Items} >=0 binary;

/* Objective function */

maximize Value: sum{j in Items} profit[j]*choose[j];

/* Constraints */
s.t. ResourceConstraints{i in Resources}: sum{j in Items} consumption[i,j]*choose[j] <= capac
solve;

display{j in Items: choose[j]=1} choose[j];
```

Solve the above model (`knapsack.mod`) in `glpsol`.

```
glpsol --model knapsack.mod
```

Dynamic Lot Sizing with Backorders (DLS)

Example 7: We are given T periods. For period t, $t = 1, \ldots, T$ let d_t be the demand in period t, $d_t \geq 0$. We wish to meet prescribed demand d_t for each of T periods $t = 1, \ldots, T$ by either producing an amount x_t up to u_t (the production capacity limit on x_t) in period t and/or by drawing upon the inventory I_{t-1} carried from the previous period. Furthermore, we might not fully satisfy the demand of any period from the production in that period or from current inventory, but could fulfill the demand from production in future periods - we permit backordering. The costs of carrying one unit of inventory from period t to period $t+1$ is given by $c_t^I \geq 0$ and the costs of backordering one unit from period $t+1$ to period t is given by $c_t^B \geq 0$. The unit production cost in period t is c_t. We assume that the total production capacity is at least as large as the total demand. So, we wish to find a production plan x_t, $t = 1, \ldots, T$, that minimizes the total cost of production, storage and backordering subject to the conditions of satisfying each demand.

DLS - a model

The Dynamic lot sizing with backorders can be formulated as follows:

$$\sum_{t=1}^{T}(c_t x_t + c_t^I I_t + c_t^B B_t) \quad \rightarrow \min$$

$$\begin{aligned} B_t - I_t &= \sum_{j=1}^{t}(d_j - x_j), && t = 1, \ldots, T, \\ x_t &\leq u_t, && t = 1, \ldots, T, \\ x_t, B_t, I_t &\geq 0, && t = 1, \ldots, T. \end{aligned}$$

Decision variables:

- x_t - production amount in period t,
- I_t - inventory amount carried from period t to period $t+1$,
- B_t - backordering amount carried from period $t+1$ to period t.

DLS - implementation (`lotsizing.mod`)

```
/* input data */
param T, integer,>=1;     # number of periods
set Periods:={1..T};         # set of Periods
param cI{Periods} >=0;  #  costs of carrying one unit of inventory
param cB{Periods} >=0; # costs of backordering one unit
param c{Periods} >=0;   # unit production costs
param u{Periods}>=0;   #  the production capacity limits
param d{Periods}>=0;   # demands

/* Checking  the total production capacity is at least
   as large as the total demand*/
check sum{t in Periods} d[t]<= sum{t in Periods} u[t];

var x{t in Periods}>=0,<=u[t]; # production plan
var I{Periods}>=0;                  #inventory amount
var B{Periods}>=0;                 # backordering amount

minimize TotalCost: sum{t in Periods} (c[t]*x[t]+cI[t]*I[t]+cB[t]*B[t]);

s.t. balance{t in Periods}: B[t]-I[t]=sum{j in Periods : j<=t}(d[j]-x[j]);
solve;

/* Displaying results */
display 'production plan';
display {t in Periods}: x[t];
display 'total cost=',  sum{t in Periods} (c[t]*x[t]+cI[t]*I[t]+cB[t]*B[t]);
display {t in Periods}: I[t];
display {t in Periods}: B[t];
```

Exercise: Provide a separated data file and solve the problem.

DLS, a positive initial inventory (`lotsizing1.mod`)

If initial inventory is positive I_0, then one can append period 0 and assign $x_0 = I_0$ and $d_0 = 0$ with zero inventory cost.

```
param InitInvent>=0, default 0; # initial inventory

param T, integer,>=1; # number of periods
/* Adding period 0*/
set Periods:=if InitInvent =0 then {1..T} else {0} union {1..T};

param cI{t in Periods}>=0; #  costs of carrying one unit of inventory
param cB{Periods}>=0; # costs of backordering one unit
param c{Periods}>=0; # unit production costs
param u{Periods}>=0; #  the production capacity limits
param d{Periods}>=0; # demands
/* input data with period 0 */
param c0I{t in Periods}:=if t=0 then 0 else cI[t];
param c0B{t in Periods}:=if t=0 then 0 else cB[t];
param c0{t in Periods}:=if t=0 then 0 else c[t];
param u0{t in Periods}:=if t=0 then InitInvent else u[t];
param d0{t in Periods}:=if t=0 then 0 else d[t];

/* Assigning x_0 = I_0 */
var x{t in Periods} >=(if t=0 then InitInvent else 0), <=u0[t]; # production plan
var I{Periods}>=0; #inventory amount
var B{Periods}>=0; # backordering amount

minimize TotalCost: sum{t in Periods} (c0[t]*x[t]+c0I[t]*I[t]+c0B[t]*B[t]);
s.t. balance{t in Periods}: B[t]-I[t]=sum{j in Periods : j<=t}(d0[j]-x[j]);
```

The minimum cost flow problem

Example 8: Consider the problem of shipment of a commodity through a network in order to satisfy demands at certain nodes V_3 from available supplies at other nodes V_1. For each $i \in V_1$ supply a_i is given, for each $i \in V_3$ demand b_i is given. Each arc (i, j) has an associated cost c_{ij} that denotes the cost per unit flow on that arc. A capacity u_{ij} is also associated with each arc (i, j) that denotes the maximum amount that can flow on the arc. The problem consists in finding a least cost flow.
Given a network $G = (V, A)$, $V = \{1, \ldots, n\}$. The set of nodes V is partitioned into V_1 (sources - supply nodes), V_2 (transshipment nodes), V_3 (sinks - demand nodes). For every $i \in V$ the following sets are defined

$$S(i) = \{j \mid (i, j) \in A\} \text{ and } P(i) = \{j \mid (j, i) \in A\}$$

$$\sum_{(i,j)\in A} c_{ij} x_{ij} \to \min$$

$$\sum_{j\in S(i)} x_{ij} - \sum_{j\in P(i)} x_{ji} = \begin{cases} a_i & i \in V_1, \\ 0 & i \in V_2, \\ -b_i & i \in V_3, \end{cases}$$

$$0 \le x_{ij} \le u_{ij}, \quad (i, j) \in A.$$

The minimum cost flow problem - an implementation

```
param n, integer, >= 2; #the number of nodes

set V:={1..n}; # the set of nodes
set V1 within V; # sources - supply nodes
set V3 within V; # sinks - demand nodes
set V2:=V diff V1 diff V3; # transshipment nodes
check: (V1 inter V3) within {}; # check if V1 and V3 are disjoint sets
set A  within V cross V; # the set of arcs

set S{i in V}:={j in V: (i,j) in A}; # the set of direct successors of i
set P{i in V}:={j in V: (j,i) in A}; # the set of direct predecessors of i

param a{V1}>=0; # the supplies
param b{V3}>=0; # the demands
check sum{i in V1} a[i] = sum{i in V3} b[i]; # check if the problem is balanced

param c{A}>= 0; # the arc costs
param u{A}>= 0; # the capacities of arcs

var x{(i,j) in A}>= 0, <= u[i,j]; # the flow on arc (i,j)

minimize Cost: sum{(i,j) in A} c[i,j]*x[i,j];

s.t. supplies{i in V1}:sum{j in S[i]} x[i,j]-sum{j in P[i]}x[j,i]=a[i];
s.t. trans{i in V2}:   sum{j in S[i]} x[i,j]-sum{j in P[i]}x[j,i]=0;
s.t. demands{i in V3}: sum{j in S[i]} x[i,j]-sum{j in P[i]}x[j,i]=-b[i];
solve;
```

```
glpsol --model mincostflow.mod
```

The minimum cost flow problem - an implementation

- Sets:
  `set V1 within V;` the declaration of set V_1 such that $V_1 \subseteq V$
  `set V2:=V diff V1 diff V3;` the declaration of set V_2 of the form $V \setminus V_1 \setminus V_3$
  `set A within V cross V;` the declaration of set A such that $A \subseteq V \times V$ (the subset of the Cartesian product)
  `set S{i in V}:={j in V: (i,j) in A};`
  $S(i) = \{j \in V \mid (i,j) \in A\}$
  `set P{i in V}:={j in V: (j,i) in A};`
  $P(i) = \{j \in V \mid (j,i) \in A\}$
- Checking a value of logical expression:
  `check: (V1 inter V3) within {};` test: $V_1 \cap V_3 = \emptyset$; if test fail the model translator reports error
  `check sum{i in V1} a[i] = sum{i in V3} b[i];` test: $\sum_{i \in V_1} a_i = \sum_{i \in V_3} b_i$

The shortest path problem - a model

Example 9: We are given $G = (V, A)$ with distinguished nodes s and t, $s, t \in V$. A nonnegative cost c_{ij} is given for each arc $(i, j) \in A$. We wish to find a path from s to t whose total cost is minimal.
It is sufficient to set $V_1 = \{s\}$, $V_3 = \{t\}$, $a_1 = 1$, $b_n = 1$, $u_{ij} = 1$ for $(i, j) \in A$ in the model of the minimum cost flow problem (see Example 6) and so:

$$\sum_{(i,j)\in A} c_{ij} x_{ij} \to \min$$

$$\sum_{j\in S(i)} x_{ij} - \sum_{j\in P(i)} x_{ji} = \begin{cases} 1 & i = s, \\ 0 & i \neq s, i \neq t, \\ -1 & i = t, \end{cases}$$

$$0 \leq x_{ij} \leq 1, \quad (i, j) \in A.$$

The shortest path problem - an implementation

```
param n, integer, >= 2; # the number of nodes
set V:={1..n};                  # the set of nodes
set A  within V cross V;        # the set of arcs

param c{(i,j) in A} >= 0;      # cij  the cost of arc (i,j)
param s in V, default 1;       # source s
param t in V, != s, default n; # sink t

var x{(i,j) in A}, >= 0, <= 1;
/* x[i,j] =1 if  arc belongs to the shortest path, 0 otherwise*/

minimize Cost: sum{(i,j) in A} c[i,j]*x[i,j];
s.t. node{i in V}:
   sum{(j,i) in A} x[j,i] + (if i = s then 1)= sum{(i,j) in A} x[i,j] + (if i = t then 1);
```

```
glpsol --model path.mod
```

The shortest path problem - randomly generated costs

Example 10: We construct an acyclic and complete graph $G = (V, A)$, with arc costs c_{ij}, $(i, j) \in A$, randomly generated from interval $[a, b]$.

```
param n, integer, >= 2; # the number of nodes
set V :={1..n};                 # the set of nodes

set A:={i in V, j in V:i<j};
/* the set of arcs in the complete acyclic graph*/

param a >=0;
param b, >a;
/* the interval of costs */

param c{(i,j) in A}, >= 0 :=Uniform(a,b);  # cij  the cost of arc (i,j)
/* the costs are randomly generated according to uniform distribution */

/* The rest is the same as in Example 7 */
```

```
glpsol --model path1.mod --data path1.dat
```

The flow shop problem

Example 11: Given m different items that are to be routed through n machines. Each item must be processed first on machine 1, then on machine 2, and finally on machine n. The sequence of items may differ for each machine. Assume that the times p_{ij} required to perform the work on item i by machine j are known. Our objective is to minimize the total time necessary to process all the items called makespan.

The flow shop problem...

The flow shop problem can modeled as follows:

$ms \rightarrow \min$

precedence constraints:

$$t_{i,j+1} \geq t_{ij} + p_{ij} \qquad i = 1, \ldots, m, j = 1, \ldots, n-1$$

resource constraints:

$$t_{ij} + By_{jik} \geq t_{jk} + p_{kj} \qquad j = 1, \ldots, n, i = 1, \ldots, m-1, k = i+1, \ldots, m$$

$$t_{kj} + B(1 - y_{jik}) \geq t_{ij} + p_{ij} \qquad j = 1, \ldots, n, i = 1, \ldots, m-1, k = i+1, \ldots, m$$

$$t_{in} + p_{in} \leq ms \qquad i = 1, \ldots, m$$

$$t_{ij} \geq 0 \qquad i = 1, \ldots, m, j = 1, \ldots, n$$

$$y_{jik} \in \{0, 1\} \qquad j = 1, \ldots, n, i = 1, \ldots, m-1, k = i+1, \ldots, m$$

Decision variables: t_{ij} is the earliest starting time of processing item i on machine j, $y_{jik} = 1$ if and only if on machine j item i is processed before item k, ms is the makespan.

B is a big number, for instance: $B = 1 + \sum_i^m \sum_{j=1}^n p_{ij}$.

The flow shop problem...

Exercise: Implement the flow shop problem in `GNU MathProg` and solve it by `glpk` for the following data:

```
data;

param n:=3; # the number of machine

param m:=7; # the number of items

/* the times p_ij required to perform the work on item i by machine j */
param p:     1       2       3:=
       1     3       3       2
       2     9       3       8
       3     9       8       5
       4     4       8       4
       5     6      10      3
       6     6       3       1
       7    7       10       3;
end;
```

Exercises

Exercise 1: Generalize the model presented in Example 1 to m constraints and n variables. Isolate the model from data. Input data: n, m, $A \in \mathbb{R}^{m \times n}$ (the constraint matrix), $b \in \mathbb{R}^m$ (the right hand side vector), $c \in \mathbb{R}^n$ (the vector of objective function coefficients) Solve the model with the data provided in Example 1.

Exercise 2:(C.H. Papadimitriou, K. Steiglitz, 1998). Consider the problem faced by a homemaker when buying food to meet certain nutritional requirements. He has a set of different kinds of foods F (for example $F = \{\text{potato}, \text{carrot}, \text{bread}, \text{butter}, \text{apple}\}$) and a set of nutrients N (for example $N = \{\text{vitamin A}, \text{vitamin B}, \text{vitamin C}\}$. Each food from F has some of each nutrient from N. Namely, he has some information about amount a_{ij} of i-th nutrient, $i \in N$, in a unit of the j-th food, $j \in F$. The requirements of i-th nutrient are given by r_i, $i \in N$. The cost per unit of the j-th food is c_j, $j \in F$. The homemaker has to decide how many units of each food to buy in order to satisfy the nutritional requirements.

Formulate a linear programming model for finding the least expensive diet (which foods to buy) and implement in `GNU MathProg` and solve it for a sample data by `glpk`. The model must isolated from data. *Hint :* See Example 3.

Exercises

Exercise 3: (The constrained shortest path) Given a directed graph $G = (V, A)$ with distinguished nodes s and t, $s, t \in V$ and cost c_{ij}, length l_{ij} for each arc $(i, j) \in A$ and length L. We wish to find a path from s to t whose total cost is minimal and total length is at most L.
Formulate an integer programming model for finding the problem and implement in `GNU MathProg` and solve it for a sample data by `glpk`. The model must isolated from data.
Hint : Modify Example 9.
Exercise 4: Scheduling a set of jobs on identical parallel machines to minimize the makespan is one of the basic and most extensively studied scheduling problems . We are given a set of jobs $J = \{1, \ldots, n\}$ which must be processed on m identical machines $M_1, \ldots, M_m$. Each machine can process at most one job at a time. Each job $i \in J$ has a processing time p_i. We wish to assign each job to exactly one machine so that the maximum job completion time of the resulting schedule, called a *makespan*, is minimal.
Formulate an integer programming model for finding the problem and implement in `GNU MathProg` and solve it for a sample data by `glpk`. The model must isolated from data.