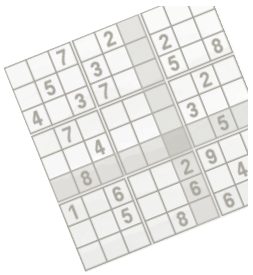


Constraints in Picat (and Prolog)

Practical Exercises

Unification?

Recall:

```
?-3=1+2.
no
?-X=1+2
X=1+2;
no
?-3=X+1
no
```

What is the problem?

Term has no meaning (even if it consists of numbers), it is just a syntactic structure!

We would like to have:

```
?-X=1+2.
X=3

?-3=X+1.
X=2

?-3=X+Y,Y=2.
X=1

?-3=X+Y,Y>=2,X>=1.
X=1
Y=2
```

- For each variable we define its **domain.**
  - we will be using discrete finite domains only
  - such domains can be mapped to integers
- We define **constraints/relations** between the variables.

```
[X,Y] :: 0..100, 3#=X+Y, Y#>=2, X#>=1.
```

- Recall a **constraint satisfaction problem.**
- We want the system to find the values for the variables in such a way that all the constraints are satisfied.

```
X=1, Y=2
```


How is constraint satisfaction realized?

- For each variable the system keeps its actual domain.
- When a constraint is added, the inconsistent values are removed from the domain.

Example:

	X	Y
	inf..sup	inf..sup
[X,Y] :: 0..100	0..100	0..100
3#=X+Y	0..3	0..3
Y#>=2	0..1	2..3
X#>=1	1	2

Picat is a programming language incorporating features from multiple programming paradigms.

The purpose is to bridge the gap between imperative and declarative languages.

```
qsort([]) = [].
qsort([H|T]) = L =>
  L = qsort([X : X in T, X =< H])
        ++ [H]
        ++ qsort([X : X in T, X > H]).
```

www.picat-lang.org

SEND+MORE=MONEY

Assign different digits to letters such that SEND+MORE=MONEY holds and S≠0 and M≠0.

Idea:

generate assignments with different digits and check the constraint


```
crypto_naive(Sol) =>
   Sol = [S,E,N,D,M,O,R,Y],
   Digits1_9 = 1..9,
   Digits0_9 = 0..9,
   member(S, Digits1_9),
   member(E, Digits0_9), E!=S,
   member(N, Digits0_9), N!=S, N!=E,
   member(D, Digits0_9), D!=S, D!=E, D!=N,
   member(M, Digits1_9), M!=S, M!=E, M!=N, M!=D,
   member(O, Digits0_9), O!=S, O!=E, O!=N, O!=D, O!=M,
   member(R, Digits0_9), R!=S, R!=E, R!=N, R!=D, R!=M, R!=O,
   member(Y, Digits0_9), Y!=S, Y!=E, Y!=N, Y!=D, Y!=M, Y!=O, Y!=R,
             1000*S + 100*E + 10*N + D +
             1000*M + 100*O + 10*R + E =
   10000*M + 1000*O + 100*N + 10*E + Y.
```

SEND+MORE=MONEY (better)

```
crypto_better(Sol) =>
   Sol = [S,E,N,D,M,O,R,Y],
   Digits1_9 = 1..9,
   Digits0_9 = 0..9,
   % D+E = 10*P1+Y
   member(D, Digits0_9),
   member(E, Digits0_9), E!=D,
   Y is (D+E) mod 10, Y!=D, Y!=E,
   P1 is (D+E) // 10, % carry bit

   % N+R+P1 = 10*P2+E
   member(N, Digits0_9), N!=D, N!=E, N!=Y,
   R is (10+E-N-P1) mod 10, R!=D, R!=E, R!=Y, R!=N,
   P2 is (N+R+P1) // 10,

   % E+O+P2 = 10*P3+N
   O is (10+N-E-P2) mod 10, O!=D, O!=E, O!=Y, O!=N, O!=R,
   P3 is (E+O+P2) // 10,

   % S+M+P3 = 10*M+O
   member(M, Digits1_9), M!=D, M!=E, M!=Y, M!=N, M!=R, M!=O,
   S is 9*M+O-P3,
   S>0,S<10, S!=D, S!=E, S!=Y, S!=N, S!=R, S!=O, S!=M.
```

Some letters can be computed from other letters and invalidity of the constraint can be checked before all letters are know

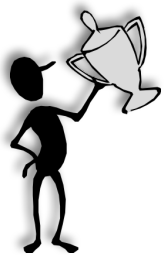
0.001 s

SEND+MORE=MONEY (CLP)

Domain filtering can take care about computing values for letters that depend on other letters.

```
import cp.
crypto(Sol) =>
   Sol=[S,E,N,D,M,O,R,Y],
   Sol :: 0..9,
   S #!= 0, M #!= 0,
              1000*S + 100*E + 10*N + D +
              1000*M + 100*O + 10*R + E #=
   10000*M + 1000*O + 100*N + 10*E + Y,
   all_different(Sol),
   solve(Sol).
```

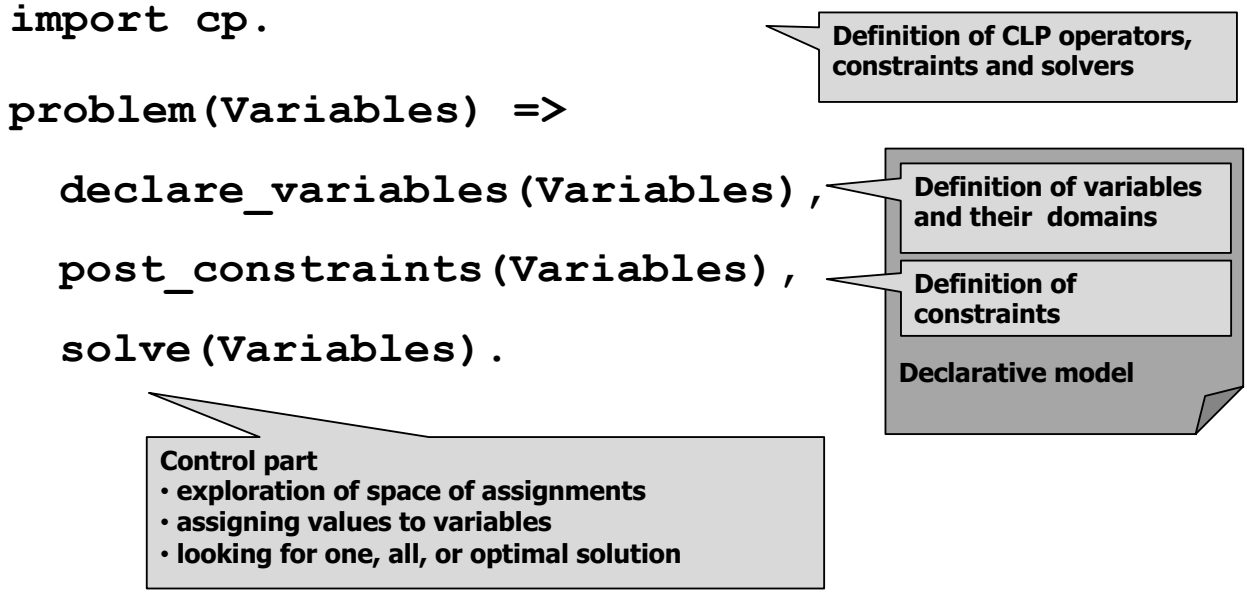
0.0 s



assign values (from domains) to variables – depth first search

Note: It is also possible to use a model with carry bits.

A typical structure of CLP programs in Picat:



Domain in Picat is a set of integers

- other values must be mapped to integers
- integers are naturally ordered

Frequently, domain is an interval

- `ListOfVariables :: MinVal..MaxVal`
- defines variables with the initial domain {MinVal,…,MaxVal}

For each variable we can define a separate domain (it is possible to use any expression providing a list of integers)

- `X :: Expr`
- `X :: [1,2,3,8,9,15]++[27,28]`

Classical arithmetic constraints with operations +,-, *, /, abs, min, max,... operations are built-in

It is possible to use comparison to define a constraint #=, #<, #>, #=<, #>=, #!=

```
Picat> A+B #=< C-2.
```

What if we define a constraint before defining the domains?

- For such variables, the system assumes initially the infinite domain -MinInt..+MaxInt

Arithmetic (reified) constraints can be connected using logical operations:

- `#~ :Q` negation
- `:P #/\ :Q` conjunction
- `:P #\/ :Q` disjunction
- `:P #=> :Q` implication
- `:P #<=> :Q` equivalence

P and Q could be Boolean variables (constants) or arithmetic, domain or Boolean constraints

Constraints alone frequently do not set the values to variables. We need to instantiate the variables via search.

- **`indomain(X)`**
  - assign a value to variable X (values are tried in the increasing order upon backtracking)
- **`solve(Vars)`**
  - instantiate variables in the list Vars
  - algorithm MAC – maintaining arc consistency during backtracking

`solve(:Options, +Variables)`

- variable ordering
  - **`forward, backward, degree, constr, min, max, min, ff, ffc, ffd,`** …
- value ordering
  - **`split, reverse_split`**
  - **`down, rand`**
- optimization
  - **`$min(X), $max(X)`**

Which **decision variables** are needed?

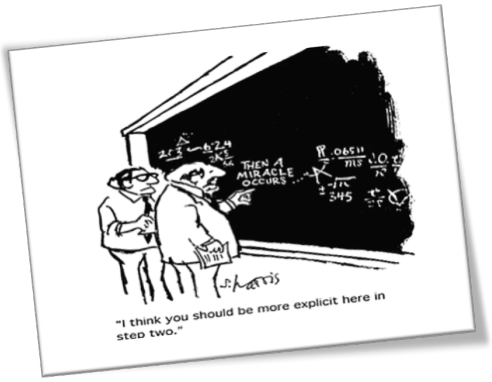
- variables denoting the problem solution
- they also define the search space

Which **values** can be assigned to variables?

- the definition of domains influences the constraints used

How to formalise **constraints**?

- available constraints
- auxiliary variables may be necessary

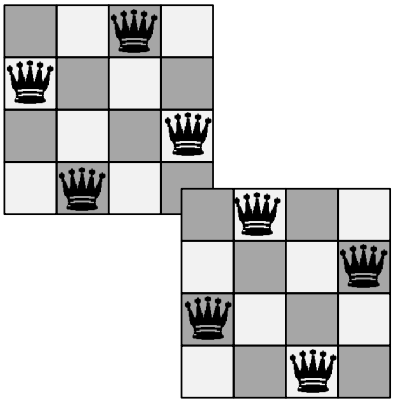


Propose a constraint model for solving the N-queens problem (place four queens to a chessboard of size N ✖ N such that there is no conflict).

```
import cp.

queens(N,Queens) =>
   QR = new_list(N), QR :: 1..N,      % position in rows
   QC = new_list(N), QC :: 1..N,      % position in columns
   Queens = zip(QR,QC),               % coordinates of queens
   foreach(I in 1..N, J in (I+1)..N)
       QR[I] #!= QR[J],               % different rows
       QC[I] #!= QC[J],               % different columns
       QC[I]-QR[I] #!= QC[J]-QR[J],   % different diagonals
       QC[I]+QR[I] #!= QC[J]+QR[J]
   end,
   solve(QR++QC).
```

4-queens: analysis

```
Picat> queens(4,Q).
Q = [{1,2},{2,4},{3,1},{4,3}] ? ;
Q = [{1,3},{2,1},{3,4},{4,2}] ? ;
Q = [{1,2},{2,4},{4,3},{3,1}] ? ;
Q = [{1,3},{2,1},{4,2},{3,4}] ? ;
Q = [{1,2},{3,1},{2,4},{4,3}] ? ;
Q = [{1,3},{3,4},{2,1},{4,2}] ? ;
Q = [{1,2},{3,1},{4,3},{2,4}] ? ;
Q = [{1,3},{3,4},{4,2},{2,1}] ? ;
...
```



Where is the problem?

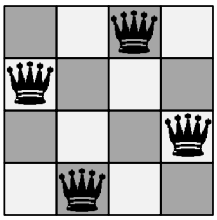
- Different assignments describe the same solution!
- There are only two different solutions (very „similar" solutions).
- The search space is non-necessarily large.

Solution

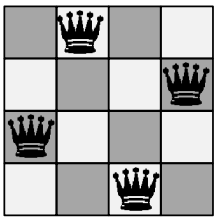
- pre-assign queens to rows (or to columns)

4-queens: a better model

```
import cp.

queens2(N,Queens) =>
   QR = 1..N,
   QC = new_list(N), QC :: 1..N,
   Queens = zip(QR,QC),
   all_different(QC),
   all_different([$QC[I]-I : I in 1..N]),
   all_different([$QC[I]+I : I in 1..N]),
   solve(QC).
```



```
Picat> queens2(4,Q).
Q = [{1,2},{2,4},{3,1},{4,3}] ?;
Q = [{1,3},{2,1},{3,4},{4,2}] ?;
no
```



Model properties:

- less variables (= smaller state space)
- less constraints (= faster propagation)

Homework:

- think about further improvements (symmetry breaking)

4-queens: a dual model

A dual model swaps the roles of values and variables.

Instead of looking for positions of queens we will be deciding whether or not a given cell contains a queen.

```
import cp.

queens_dual(N,Board) =>
   Board = new_array(N,N),
   Board :: 0..1,
   foreach(R in 1..N)    % at most one queen per row
        sum([Board[R,C] : C in 1..N]) #=< 1
   end,
   foreach(C in 1..N)    % at most one queen per column
        sum([Board[R,C] : R in 1..N]) #=< 1
   end,
   foreach(D in 0..(N-1)) % at most one queen per diagonal
        sum([Board[I,I+D] : I in 1..(N-D)]) #=< 1,
        sum([Board[I+D,I] : I in 1..(N-D)]) #=< 1,
        sum([Board[N-I+1,I+D] : I in 1..(N-D)]) #=< 1,
        sum([Board[N-I+1-D,I] : I in 1..(N-D)]) #=< 1
   end,
   sum([Board[R,C] : R in 1..N, C in 1..N]) #= N,
   solve(Board).
```

```
Picat> queens2(4,B).
B = {{0,0,1,0},{1,0,0,0},{0,0,0,1},{0,1,0,0}} ?;
B = {{0,1,0,0},{0,0,0,1},{1,0,0,0},{0,0,1,0}} ?;
no
```

model	#backtracks (8 queens)
naive	24
classical	24
dual	8540

Comments:

- The above model is less appropriate for CP due to Boolean domains and weak constraints. Better suited for SAT.

Back to Sudoku

```
import cp.

sudoku(Board) =>
   N = Board.length,
   N1 = ceiling(sqrt(N)),
   Board :: 1..N,
   foreach(R in 1..N)
       all_different([Board[R,C] :
                      C in 1..N])
   end,
   foreach(C in 1
       all_differ
   end,
   foreach(R in 1
       all_differ

   end,
   solve(Board).
```

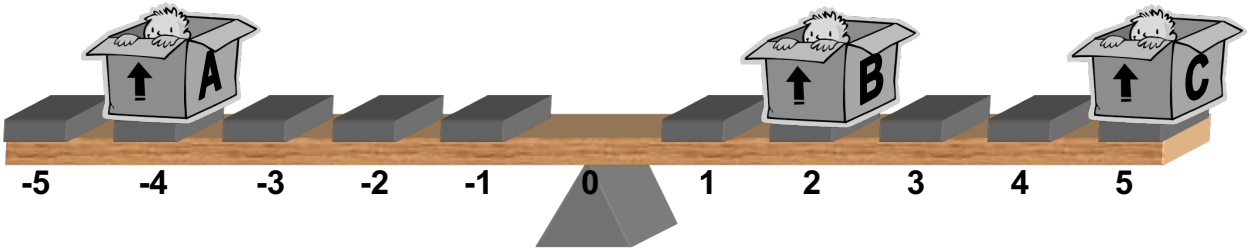
9	6	3	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	8	1	7	3

```
board(Board) =>
  Board = {{_, 6, _, 1, _, 4, _, 5, _},
           {_, _, 8, 3, _, 5, 6, _, _},
           {2, _, _, _, _, _, _, _, 1},
           {8, _, _, 4, _, 7, _, _, 6},
           {_, _, 6, _, _, _, 3, _, _},
           {7, _, _, 9, _, 1, _, _, 4},
           {5, _, _, _, _, _, _, _, 2},
           {_, _, 7, 2, _, 6, 9, _, _},
           {_, 4, _, 5, _, 8, _, 7, _}}.
```

Seesaw problem

The problem:

Adam (36 kg), Boris (32 kg) and Cecil (16 kg) want to sit on a seesaw with the length 10 foots such that the minimal distances between them are more than 2 foots and the seesaw is balanced.



A CSP model:

- A,B,C in -5..5 — position
- 36*A+32*B+16*C = 0 — equilibrium state
- |A-B|>2, |A-C|>2, |B-C|>2 — minimal distances

Seesaw problem - implementation

```
import cp.

seesaw(Sol) =>
    Sol = [A,B,C],
    Sol :: -5..5,

    36*A+32*B+16*C #= 0,
    abs(A-B)#>2, abs(A-C)#>2, abs(B-C)#>2,

    solve(Sol).
```

```
Picat> seesaw(X).

X = [-4,2,5] ? ;
X = [-4,4,1] ? ;
X = [-4,5,-1] ? ;
X = [4,-5,1] ? ;
X = [4,-4,-1] ? ;
X = [4,-2,-5] ? ;

no
```

Symmetry breaking

– important to reduce search space

```
import cp.

seesaw(Sol) =>
    Sol = [A,B,C],
    Sol :: -5..5,

    A #=< 0,
    36*A+32*B+16*C #= 0,
    abs(A-B)#>2, abs(A-C)#>2, abs(B-C)#>2,

    solve(Sol).
```

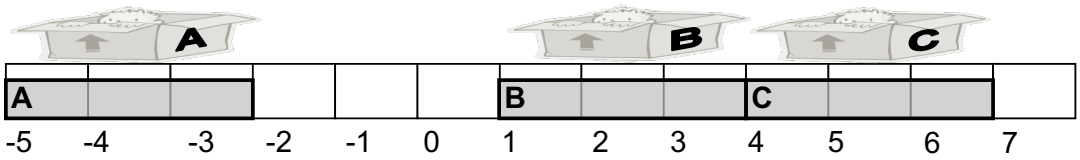
```
Picat> seesaw(X).

X = [-4,2,5] ? ;
X = [-4,4,1] ? ;
X = [-4,5,-1] ? ;

no
```

Seesaw problem - a different perspective

```
[A,B,C] :: -5..5,
A #=< 0,
36*A+32*B+16*C #= 0,
abs(A-B)#>2,
abs(A-C)#>2,
abs(B-C)#>2
```

```
A in -5..0
B in -2..5
C in -5..5
```

A set of similar constraints typically indicates a structured sub-problem that can be represented using a **global constraint.**



We can use a global constraint describing **allocation of activities to exclusive resource**.

```
[A,B,C] :: -5..5,
A #=< 0,
36*A+32*B+16*C #= 0,
cumulative([A,B,C],[3,3,3],[1,1,1],1)
```

```
A in -5..0
B in -2..5
C in -5..5
```

cumulative(starts,durations,resources,limit)

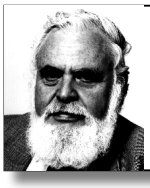
Golomb ruler

A **ruler with M marks** such that **distances** between any two marks are **different**.

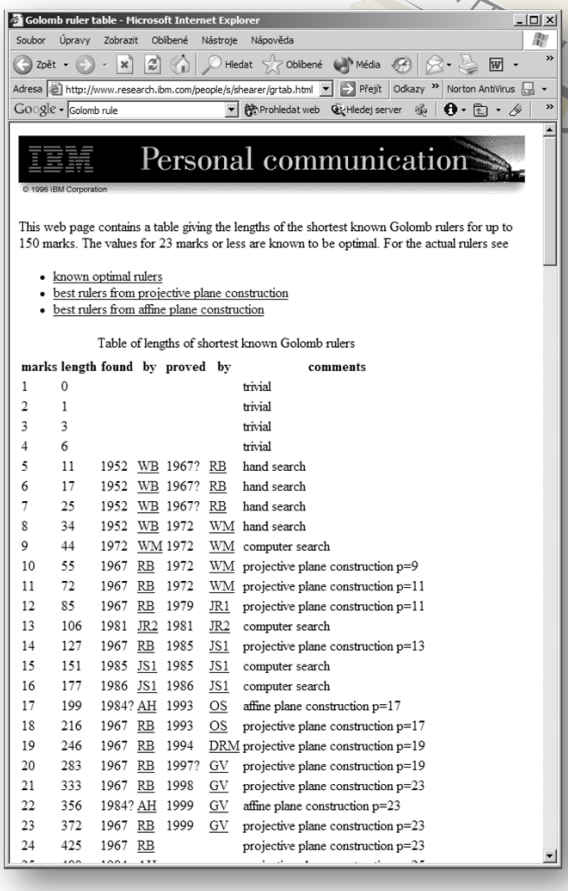
The **shortest ruler** is the optimal ruler.

Hard for M≥16, no exact algorithm for M ≥ 24!

Applied in **radioastronomy**.



Solomon W. Golomb
Professor
University of Southern California
http://csi.usc.edu/faculty/golomb.html



Golomb ruler table - Microsoft Internet Explorer

Personal communication

© 1996 IBM Corporation

This web page contains a table giving the lengths of the shortest known Golomb rulers for up to 150 marks. The values for 23 marks or less are known to be optimal. For the actual rulers see

- known optimal rulers
- best rulers from projective plane construction
- best rulers from affine plane construction

Table of lengths of shortest known Golomb rulers

marks	length	found	by	proved	by	comments
1	0					trivial
2	1					trivial
3	3					trivial
4	6					trivial
5	11	1952	WB	1967?	RB	hand search
6	17	1952	WB	1967?	RB	hand search
7	25	1952	WB	1967?	RB	hand search
8	34	1952	WB	1972	WM	hand search
9	44	1972	WM	1972	WM	computer search
10	55	1967	RB	1972	WM	projective plane construction p=9
11	72	1967	RB	1972	WM	projective plane construction p=11
12	85	1967	RB	1979	JR1	projective plane construction p=11
13	106	1981	JR2	1981	JR2	computer search
14	127	1967	RB	1985	JS1	projective plane construction p=13
15	151	1985	JS1	1985	JS1	computer search
16	177	1986	JS1	1986	JS1	computer search
17	199	1984?	AH	1993	OS	affine plane construction p=17
18	216	1967	RB	1993	OS	projective plane construction p=17
19	246	1967	RB	1994	DRM	projective plane construction p=19
20	283	1967	RB	1997?	GV	projective plane construction p=19
21	333	1967	RB	1998	GV	projective plane construction p=23
22	356	1984?	AH	1999	GV	affine plane construction p=23
23	372	1967	RB	1999	GV	projective plane construction p=23
24	425	1967	RB			projective plane construction p=23

Golomb ruler – a model

A base model:

Variables X_1, …, X_M with the domain 0..M*M

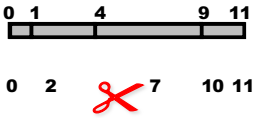
$X_1 = 0$ ***ruler start***

$X_1 < X_2 < \ldots < X_M$ ***no permutations of variables***

$\forall i<j\ D_{i,j} = X_j - X_i$ ***difference variables***

all_different({$D_{1,2}$, $D_{1,3}$, … $D_{1,M}$, $D_{2,3}$, … $D_{M-1,M}$})

Model extensions:



$D_{1,2} < D_{M-1,M}$ ***symmetry breaking***

better bounds (**implied constraints**) for $D_{i,j}$

$D_{i,j} = D_{i,i+1} + D_{i+1,i+2} + \ldots + D_{j-1,j}$

so **$D_{i,j} \geq \Sigma_{j-i} = (j-i)*(j-i+1)/2$** ***lower bound***

$X_M = X_M - X_1 = D_{1,M} = D_{1,2} + D_{2,3} + \ldots D_{i-1,i} + D_{i,j} + D_{j,j+1} + \ldots + D_{M-1,M}$

$D_{i,j} = X_M - (D_{1,2} + \ldots D_{i-1,i} + D_{j,j+1} + \ldots + D_{M-1,M})$

so **$D_{i,j} \leq X_M - (M-1-j+i)*(M-j+i)/2$** ***upper bound***

Golomb ruler in Picat

```
import cp.
golomb(M,X) =>
    X = new_list(M),
    X :: 0..(M*M),                          % domains for marks
    X[1] = 0,

    foreach(I in 1..(M-1))
        X[I] #< X[I+1]                      % no permutaions
    end,

    D = new_array(M,M),                     % distances
    foreach(I in 1..(M-1),J in (I+1)..M)
        D[I,J] #= X[J] - X[I],
        D[I,J] #>= (J-I)*(J-I+1)/2,         % bounds
        D[I,J] #=< X[M] - (M-1-J+I)*(M-J+I)/2
    end,

    D[1,2] #< D[M-1,M],                     % symmetry breaking
    all_different([$D[I,J] : I in 1..(M-1),
                             J in (I+1)..M]),
    solve($[min(X[M])],X).
```

Golomb ruler - some results

What is the effect of different constraint models?

size	base model	base model + symmetry	base model + symmetry + implied constraints
7	12	7	4
8	94	44	21
9	860	353	143
10	7 494	3 212	1 091
11	147 748	57 573	23 851

time in milliseconds on 1,7 GHz Intel Core i7, Picat 1.9#6

What is the effect of different search strategies?

size	fail first		leftmost first	
	enum	*split*	*enum*	*split*
7	9	9	5	4
8	67	68	23	21
9	537	537	170	143
10	4 834	4 721	1 217	1 091
11	134 071	132 046	26 981	23 851

time in milliseconds on 1,7 GHz Intel Core i7, Picat 1.9#6

Sky Observatory

- Assume a sky observatory with four **telescopes**:
  - Newton, Kepler, Dobson, Monar
- Each day, each telescope is used by one of the following **observers**:
  - scientists (3), students (2), visitors (1), nobody (0)
- Each day, we know the expected **weather**
  - ideal (0), worse (1), no-observations (2)
- and **phases of the moon**
  - 0 (new moon), ..., 4 (full moon), 5, 6.
- The **problem input** is defined by two lists (of equal length) of weather and moon conditions:
  - [1,1,0,0,1,2,1,0],
  - [1,1,2,2,3,3,4,4]

Sky Observatory - restrictions

- Newton and Kepler cannot be used together.
- Newton cannot be used by visitors.
- Scientists are never using Monar.
- Dobson cannot be used around full moon (3-5).
- Scientists (students) use at most one telescope each day.
- Students must use at least one telescope during the whole scheduling period.
- When the weather is ideal either students or scientists must use some telescope.

Sky Observatory - objectives

- Using each telescope costs some money (**expenses**), and visitors pay some money (**income**) for using the telescope according to the following table:

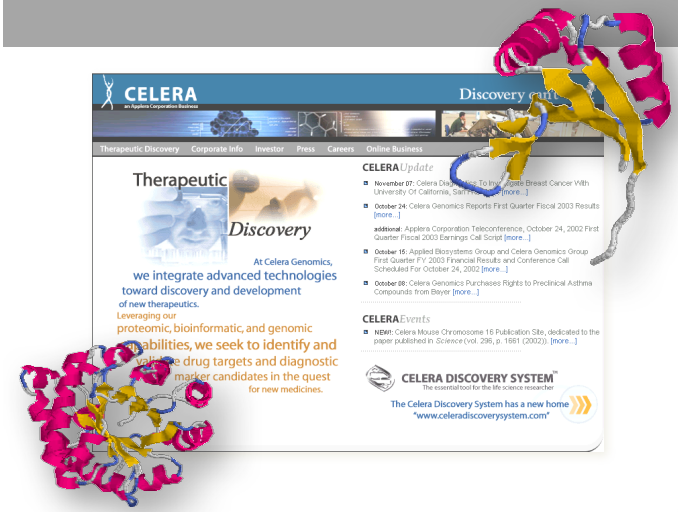
	Monar	Dobson	Kepler	Newton
expenses	10	50	60	70
income	20	60	100	100

- In case of bad weather or bad moon conditions (3-5) there is 50% **discount** for visitors when using Monar or Dobson.
- There is some **initial budget** given and the final **balance cannot be negative.**
- **Maximize scientific output** of observations (scientists are preferred over students that are preferred over the visitors).

Sky Observatory - constraint model

```
sky(Moon,Weather,Budget, Schedule,Money) =>
     N = length(Moon),                    % number of days
     Schedule = [[_,_,_,_] : _ in 1..N],
     Money = new_list(N),

     foreach({M,W,B,S} in zip(Moon,Weather,Money,Schedule))
       S = [Newton, Kepler, Dobson, Monar],
       if W = 2 then S :: 0..0            % bad weather -> non observations
                else S :: 0..3            % possible users of telescopes
       end,
       Newton#=0 #\/ Kepler#=0,           % Newton and Kepler cannot be used together
       Newton #!= 1,                      % Newton cannot be used by visitors
       Monar #!= 3,                       % scientists are never using Monar
       if 3=<M, M=<5 then Dobson#=0 end, % Dobson cannot be used around full moon (3-5)
       [Nobody,Visitors,Students,Scientists] :: 0..4,
       global_cardinality(S, $[0-Nobody,1-Visitors,2-Students,3-Scientists]),
       Scientists #=< 1, Students #=< 1,
              % scientists (students) use at most one telescope each day
       if W=0 then Scientists+Students #> 0 end,
          % when the weather is ideal either students or scientists must use some telescope
       table_in({Monar,ME,MI}, [{0,0,0},{1,10,20},{2,10,0},{3,10,0}]),
       table_in({Dobson,DE,DI}, [{0,0,0},{1,50,60},{2,50,0},{3,50,0}]),
       table_in({Kepler,KE,KI}, [{0,0,0},{1,60,100},{2,60,0},{3,60,0}]),
       table_in({Newton,NE,NI}, [{0,0,0},{1,70,100},{2,70,0},{3,70,0}]),
       if ((3=<M, M=<5) ; W=1) then
              % bad weather or bad moon conditions -> 50% discount for Monar or Dobson
          B #= (ME+DE+KE+NE-MI/2-DI/2-KI-NI)
       else
          B #= (ME+DE+KE+NE-MI-DI-KI-NI)
       end
     end,

     Budget #>= sum(Money),
     Vars = flatten(Schedule),
     count(2,Vars,#>,0),                  % students must use at least one telescope
     Obj #= sum(Vars),                    % scientists first, then students, then visitors
     solve([max,$max(Obj)],Vars).
```

Some Real Applications



Bioinformatics

- DNA sequencing (Celera Genomics)
- deciding the 3D structure of proteins from the sequence of amino acids

Planning and Scheduling

- automated planning of spacecraft activities (Deep Space 1)
- manufacturing scheduling

Books

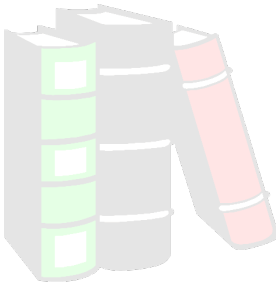
- P. Van Hentenryck: **Constraint Satisfaction in Logic Programming**, MIT Press, 1989
- E. Tsang: **Foundations of Constraint Satisfaction**, Academic Press, 1993
- K. Marriott, P.J. Stuckey: **Programming with Constraints: An Introduction**, MIT Press, 1998
- R. Dechter: **Constraint Processing**, Morgan Kaufmann, 2003
- **Handbook of Constraint Programming**, Elsevier, 2006
- N-F. Zhou, H. Kjellerstrand, J. Fruhman: **Constraint Solving and Planning in Pica**t, Springer 2015

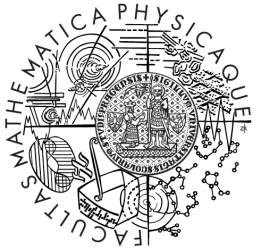
Journals

- **Constraints**, An International Journal. Springer Verlag
- **Constraint Programming Letters**, free electronic journal

On-line resources

- **Course Web** (transparencies)
  http://ktiml.mff.cuni.cz/~bartak/podminky/
- **On-line Guide to Constraint Programming** (tutorial)
  http://ktiml.mff.cuni.cz/~bartak/constraints/
- **Constraint Programming online** (community web)
  http://www.cp-online.org/





Roman Barták
Charles University, Faculty of Mathematics and Physics
bartak@ktiml.mff.cuni.cz