Constraint Programming

Solving combinatorial puzzles when you are lazy

Håkan Kjellerstrand (hakank@gmail.com)

2023-05-19

http://hakank.org/cp_mensa_2023/

Overview

Overview

- Presentation of me
- A little on Combinatorial Puzzles, Constraint Programming (CP), and MiniZinc
- SEND+MORE=MONEY
- Sudoku
- More puzzles showing features of CP

About me

- Håkan Kjellerstrand (hakank@gmail.com)
  http://hakank.org/
  http://hakank.org/minizinc/
- GitHub: https://github.com/hakank/hakank
- Twitter: https://twitter.com/hakankj
- Facebook: https://www.facebook.com/hakankj
- StackOverflow:
  https://stackoverflow.com/users/195636/hakank

Background

- First: Tester, Technical Support, Technical Writer (1982-1994)
- Then: Software developer (1996-2019)
- 2008: Constraint Programming as a hobby
- Now: Independent Researcher / Consultant
  Constraint Programming, Logic Programming, etc.

What do I do with CP?

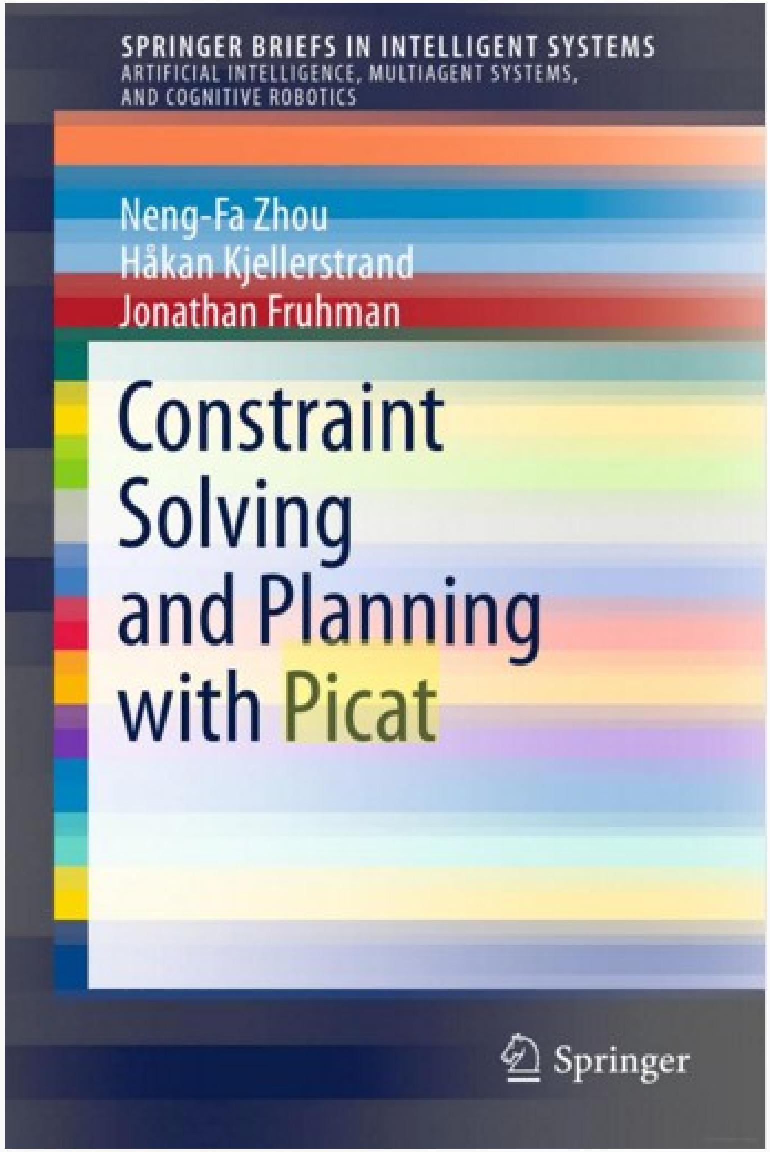
- Constraint models on puzzles, combinatorial problems, and some serious stuff: consulting, mostly scheduling problems
- Testing different CP systems (~30) and complains about bugs/missing features/etc
  http://www.hakank.org/common_cp_models/
- First CP dedicated blog (2009):
  My Constraint Programming Blog

  http://hakank.org/constraint_programming_blog/

Some of my puzzle models

A Digital Difficulty, A Round of Golf, ABC Endview, Age of three Children, All interval, Ambigous dates, Another kind of Magic Square, Archery Match, Archery puzzle, Arch Friends, Autoref, Balanced brackets, Bales of Hay, Bank card, Barrells puzzle, Binero/Binoxxo/Binary Sudoku, Birthday coins, Book buy, Bridge and Torch problem, Broken weights, Calculs d'Enfer, Chandelier balancing, Circling squares, Clock triplets, Coin problems (coin changes etc), Combination locks, Consecutive digits, Controversy about the weekday, Countdown, Crossfigure, Crosswords, Crypta, Crypto, Crystal maze, Curious set of integers, Curious numbers, de Bruijn sequences, Dice with a difference, Digits of the square, Dividing the spoils, Divisible by 9 through 1, Divisible by 1 to 9, Domino, Drive Ya Nuts, Bishop placement, Dudeney numbers, Einstein puzzle / Zebra puzzle, Some Enigma puzzles, Farmer and cow problem, Fill a pix, Five brigades, Five brigands, Five elements, Five statements, Five words that share no letters, Four islands, Funny dice, Futoshiki, Golomb ruler, Grocery puzzle, Hanging weights, Hitori, Gunport problem, Harry Potter Seven Potions, Hidato, Ice cream, Jive turkeys, Jobs puzzle, Just forgotten, Kakurasu, Kakuro, KenKen, Killer Sudoku, Knight tour, Kojun, Kyudoku, Labeled dice, Langford's number problem, Least difference, Letter square, Lights out, M12 puzzle, Magic sequences, Magic series, Magic square and cards, Magic squares, Magic Sudoku, Manasa and stones, Map coloring, Minesweeper, Mislabeled boxes, Missing digit, Monkey & Coconuts, Monks and doors, Monorail, Move one coins, Multi Sudoku, Music Men, N-queens, Non dominant queens, Nonograms, Nontransitive dice, N-puzzle, Number locks, Numberlink, Numbrix, One off digit problem, Ormat games, Pandigital numbers, Perfect square sequence, Photo problem, Pi Days Sudoku, Pool ball triangles, Prime multiplication, Pyramid of numbers, Rectangle placements, Rogo, Rookwise chain, Safe cracking, Samurai puzzle, Sangraal puzzle, Secret santa, Self referential quiz, Self referential sentence, Rotation puzzle, SEND+MORE=MONEY, SEND+MOST=MONEY, Seseman puzzle, Shikaku, Sicherman dice, Ski assignment, SET puzzle, Skyscraper, Smullyan's Knight and Knaves problem, Solitaire, Square root of Wonderful, Stamp licking, Strimko, Sudoku, Suguru, Sumaddle, Sumbrero, Survo puzzle, Takuzu, Ten statements, The Paris Marathon problem, The Vicar's age, Three jugs problem, Three in a row puzzle, Twelve statements, Twin letters, Two cube calendar, Uniform dice, Who killed Agatha, Wine cask puzzle, Word golf, Wijuko

The Picat book (2015)



Zhou, Kjellerstrand, Fruhman:
Constraint Solving and Planning with Picat
Springer (2015)

http://picat-lang.org/picatbook2015.html

(Free PDF available)

Especially the chapters on CP:
- 2. Basic Constraint Modeling
- 3. Advanced Constraint Modeling

My Picat page: http://hakank.org/picat/

Combinatorial puzzles

Combinatorial puzzles

- Not well defined
- Single person puzzles based on integers/finite domains (including booleans).
- Logicial puzzles, mathematical recreation problems, pen-and-paper/grid puzzles
- Sometimes with some initial hints
- Sometimes exactly one solution

Constraint Programming

What is CP used for?

- Scheduling, Resource allocation, Staff rostering
- Packing problems
- Vehicle / transport routing / TSP
- Constraint satisfaction problems (CSP)
- Combinatorial search and optimization
- Etc.
- And: Solving puzzles!

General concepts in CP

- **Decision variables** with **finite domains** (integers)
- **Constraints** relating these variables to each other
- Find a **solution** (or many/all solutions) satisfiing all the constraints and the domains of the variables. Or show than there is no solution.

MiniZinc

MiniZinc

- https://www.minizinc.org/
  https://github.com/MiniZinc
- MiniZinc Handbook:
  https://www.minizinc.org/doc-latest/index.html
- MiniZinc-Python
  https://minizinc-python.readthedocs.io/en/latest/
- MiniZinc Challenge:
  https://www.minizinc.org/challenge.html
- My MiniZinc Page
  http://hakank.org/minizinc/

MiniZinc

- High level constraint modeling language
- Many different constraint solvers
- Support for many global constraints
- Not a full fledged programming language.
  For more complex tasks a proper programming language might be needed, e.g. MiniZinc-Python

MiniZinc: parts of a model

- Include statement
- Parameters, fixed data (hints)
  Can be in a separate data file
- Decision variables with domains
- Constraints
- Solve statement
- Output section

SEND+MORE=MONEY
First puzzle

SEND+MORE=MONEY

- Assign a distinct digit (0..9) to each of the letters (S,E,N,D,M,O,R,Y) so this equation is satisfied:

  SEND+MORE=MONEY

  with S and M > 0

SEND+MORE=MONEY: Parameters, decision variable with domains

```
% Fixed parameter
int: N = 9;  % upper bound of the domain

% Decision variables with domains
var 0..N: s;    % 's' can be assigned to any values between 0..9
var 0..N: e;
var 0..N: n;
var 0..N: d;
var 0..N: m;
var 0..N: o;
var 0..N: r;
var 0..N: y;
```

SEND+MORE=MONEY: The constraints

```
% All values must be distinct
constraint all_different([s,e,n,d,m,o,r,y]);

% The equation SEND + MORE = MONEY
constraint
            1000*s + 100*e + 10*n + d  +
            1000*m + 100*o + 10*r + e  =
  10000*m + 1000*o + 100*n + 10*e + y

  /\ % leading digits cannot be 0
  s > 0 /\ m > 0;
```

SEND+MORE=MONEY: Solve statement

```
% We want all solutions

solve satisfy;
```

SEND+MORE=MONEY: The complete model

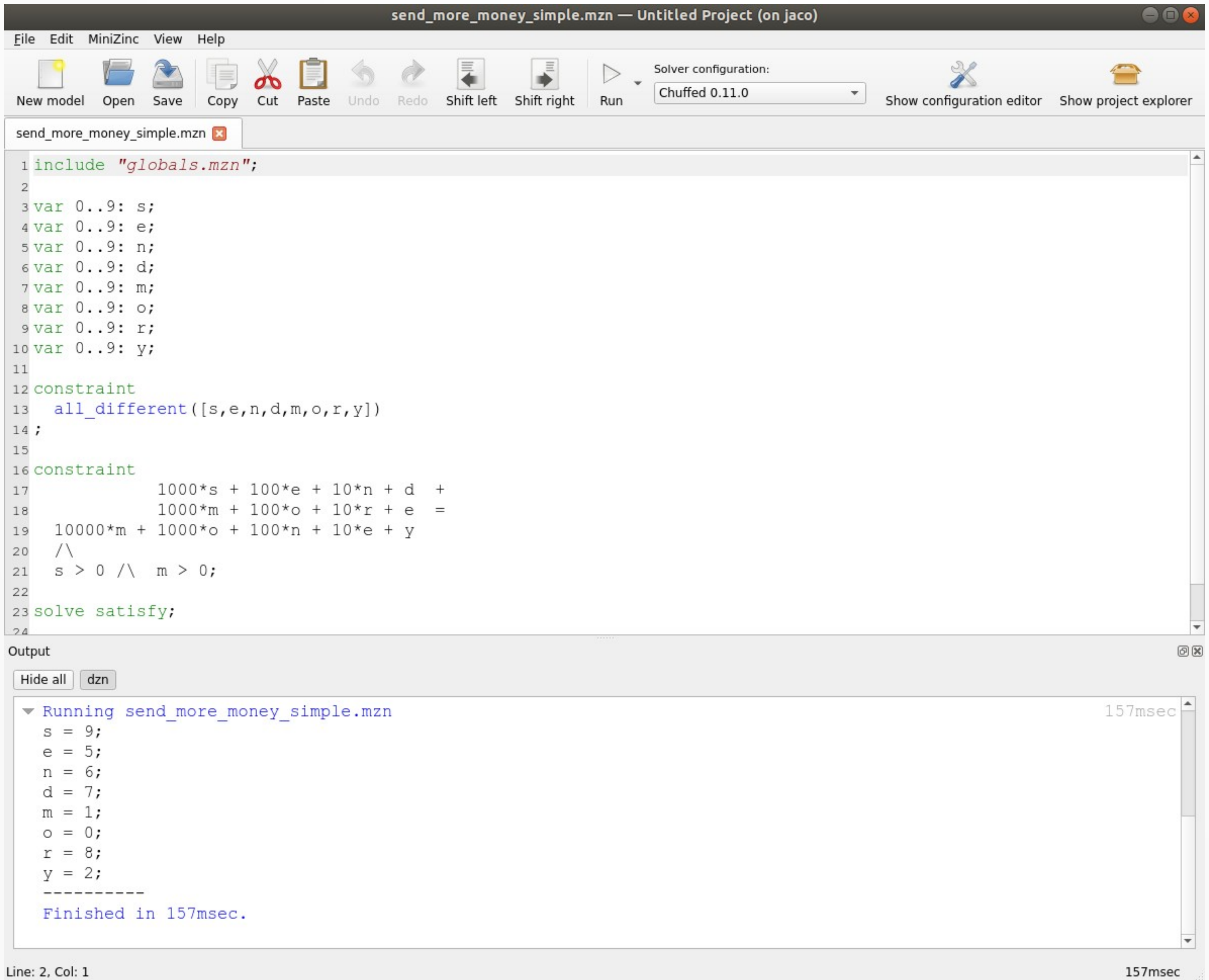
```
include("globals.mzn"); % for loading definition of all_different

int: N = 9;

var 0..N: s; var 0..N: e; var 0..N: n; var 0..N: d;
var 0..N: m; var 0..N: o; var 0..N: r; var 0..N: y;

constraint all_different([s,e,n,d,m,o,r,y]);
constraint
            1000*s + 100*e + 10*n + d  +
            1000*m + 100*o + 10*r + e  =
  10000*m + 1000*o + 100*n + 10*e + y
  /\
  s > 0 /\m > 0;

solve satisfy;
```

SEND+MORE=MONEY: Solution

```
$ minizinc send_more_money.mzn -a

s = 9;
e = 5;
n = 6;
d = 7;
m = 1;
o = 0;
r = 8;
y = 2;
----------
==========


SEND + MORE = MONEY
9567 + 1085 = 10652
```

Command line: Using -a (all solutions) to ensure a unique solution.
In MiniZincIDE there's an option to show all solutions.

MiniZincIDE



Though I tend to use Emacs

MiniZincIDE: model

```
include "globals.mzn";

var 0..9: s;
var 0..9: e;
var 0..9: n;
var 0..9: d;
var 0..9: m;
var 0..9: o;
var 0..9: r;
var 0..9: y;

constraint
  all_different([s,e,n,d,m,o,r,y])
;

constraint
              1000*s + 100*e + 10*n + d  +
              1000*m + 100*o + 10*r + e  =
  10000*m + 1000*o + 100*n + 10*e + y
  /\
  s > 0 /\  m > 0;

solve satisfy;
```

MiniZincIDE output

```
Running send_more_money_simple.mzn                                  127msec
s = 9;
e = 5;
n = 6;
d = 7;
m = 1;
o = 0;
r = 8;
y = 2;
----------
==========
Finished in 127msec.
```

Sudoku

Sudoku

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

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

Source: Wikipedia

Sudoku

- Given a N x N grid with values 1..N, together with some hints, ensure that:
- All values in each row are all different
- All values in each column are all different
- All values in each sub grid ($\sqrt{N}$ x $\sqrt{N}$) are all different

Sudoku

The rules of Sudoku = The constraints

Sudoku: The setup, include, parameters and decision variables

```
include "globals.mzn";

% parameters
int: n;                       % size of grid (n x n)
int: m = ceil(sqrt(n)); % size of sub regions

% decision variables
array[1..n, 1..n] of var 1..n: x;

solve satisfy;
```

Sudoku: Convert the rules to constraints

```
constraint

  forall(i in 1..n) (
    % All values in each row are all different
    all_different([x[i,j] | j in 1..n]) /\

    % All values in each column are all different
    all_different([x[j,i] | j in 1..n])
  )

  /\
  % All values in each sub grid (√N x √N) are all different
  forall(i in 0..m-1,j in 0..m-1) (
    all_different([x[r,c] | r in i*m+1..i*m+m, c in j*m+1..j*m+m])
  );
```

Sudoku: Simple problem instance (4x4)

```
n = 4;

%
% The integers are the given hints.
% '_' represents an unknown value.
%
x = array2d(1..n, 1..n, [
   4, _,  _, _,
   3, 1,  _, _,

   _, _,  4, 1,
   _, _,  _, 2,
]);
```

Sudoku: solution

```
4 2 1 3
3 1 2 4
2 3 4 1
1 4 3 2
----------
==========
```

Sudoku 4x4
Simple constraint propagation

Constraint Propagation

- A simplified example of how a CP solver solves a problem using Constraint Propagation
- Not all constraint solvers use this technique, but it can be instructive to see what is happening under the hood of a constraint solver.
- Some other solving techniques: SAT, Linear Programming, Integer programming, SMT, Lazy Clause Generation, Local Search.

Sudoku 4x4 – simple propagation example

```
4 _   _ _
3 1   _ _

_ _   4 1
_ _   _ 2
```

```
The (unique) solution

4 2   1 3
3 1   2 4

2 3   4 1
1 4   3 2
```

How does a CP solver reach this solution?

Sudoku 4x4 – simple propagation example

4	1234	1234	1234
3	1	1234	1234
1234	1234	4	1
1234	1234	1234	2

Add DOMAINS (1..4) to all unknown variables.
Hints are FIXED already.

Now we will propagate the three alldifferent constraints:

- all_different(ROW)
- all_different(COLUMN)
- all_different(BLOCK)

This is a very simplified example.
Real CP systems use more intelligent propagation.

Sudoku 4x4 – simple propagation example

4	2	1234	1234
3	1	1234	1234
1234	1234	4	1
1234	1234	1234	2

Cell (1,1): Fixed value (4).

Cell (1,2): Reduce:
- remove 4 (row, block)
- remove 1 (column, block)
- remove 3 (block)

→ Single value: 2

Sudoku 4x4 – simple propagation example

4	2	1 3	1234
3	1	1234	1234
1234	1234	4	1
1234	1234	1234	2

Cell (1,3): Reduce:

- remove 4 (row, column)
- remove 2 (row)

→ {1 3}

Sudoku 4x4 – simple propagation example

4	2	1 3	3
3	1	1234	1234
1234	1234	4	1
1234	1234	1234	2

Cell (1,4): Reduce:
- remove 4 (row)
- remove 1 (column)
- remove 2 (column)

→ 3

Note: Here we don't go back to fix cell (1,3).

Cell (2,1): fixed (3)
Cell (2,2): fixed (1)

Sudoku 4x4 – simple propagation example

4	2	1 3	3
3	1	2	1234
1234	1234	4	1
1234	1234	1234	2

Cell (2,3): Reduce:

- remove 3 (row)
- remove 1 (row)
- remove 4 (column)

→ 2

Sudoku 4x4 – simple propagation example

4	2	1 3	3
3	1	2	4
1234	1234	4	1
1234	1234	1234	2

Cell (2,4): Reduce:

- remove 3 (row)
- remove 1 (row, column)
- remove 2 (row, column)

→ 4

Sudoku 4x4 – simple propagation example

4	2	1 3	3
3	1	2	4
2	1234	4	1
1234	1234	1234	2

Cell (3,1): Reduce:
- remove 4 (row, column)
- remove 1 (row)
- remove 3 (column)

→ 2

Sudoku 4x4 – simple propagation example

4	2	1 3	3
3	1	2	4
2	3	4	1
1234	1234	1234	2

Cell (3,2): Reduce:

- remove 1 (row, column, block)
- remove 2 (row)
- remove 4 (row)

$\rightarrow$ 3

Sudoku 4x4 – simple propagation example

4	2	1 3	3
3	1	2	4
2	3	4	1
1234	1234	1234	2

Cell (3,3): Fixed.
Cell (3,4): Fixed.

Sudoku 4x4 – simple propagation example

4	2	1 3	3
3	1	2	4
2	3	4	1
1	1234	1234	2

Cell (4,1): Reduce

- remove 2 (row)
- remove 4 (column)
- remove 3 (column)

→ 1

Sudoku 4x4 – simple propagation example

4	2	1 3	3
3	1	2	4
2	3	4	1
1	4	1234	2

Cell (4,2): Reduce

- remove 1 (row)
- remove 2 (row)
- remove 3 (column)

→ 4

Sudoku 4x4 – simple propagation example

4	2	1 3	3
3	1	2	4
2	3	4	1
1	4	3	2

Cell (4,3): Reduce

- remove 2 (row, block)
- remove 4 (column, block)
- – - remove 1 (block)

→ 3

Cell (4,4): Fixed 2

Are we finished? No!

There is still a variable/cell with no single assignment, i.e. Cell (1,3).

Sudoku 4x4 – simple propagation example

4	2	1	3
3	1	2	4
2	3	4	1
1	4	3	2

Cell (1,3): Reduce
- remove 3 (row, block)

→ 1

And now all variables has been assigned to a single value.

Sudoku 4x4 – simple propagation example

4	2	1	3
3	1	2	4
2	3	4	1
1	4	3	2

… and we got a solution!

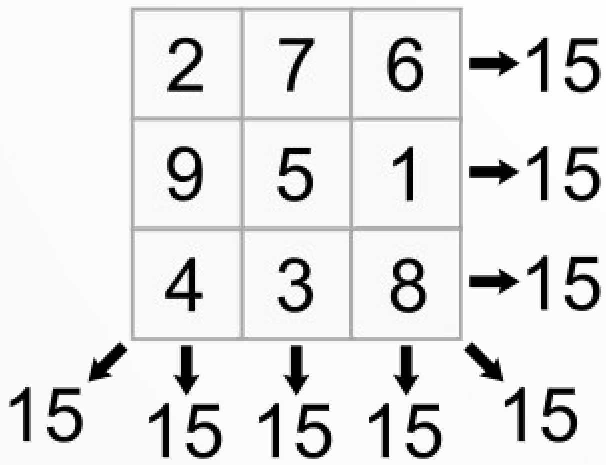
It is unique – as a Sudoku should be.

Magic squares

Magic squares

- Place all numbers 1..N*N in a NxN grid with a magic total (M) such that
- The sum of each row = M
- The sum of each column = M
- The sum of main diagonal = M
- The sum of opposite diagonal = M
- The magic total M = N*(N*N+1) // 2

Magic squares: 3x3



Source: https://en.wikipedia.org/wiki/Magic_square

Magic squares: Modeling

- What are the parameters?
- What are the decision variables and their domains?
  How to represent them?
- What are the constraints?
  How to model them?
- One, two, all solutions?

Magic squares: Parameters

```
include "globals.mzn";
int: n;
int: total = (n*(n*n + 1)) div 2;
```

Magic squares: Decision variables

```
array[1..n,1..n] of var 1..n*n: magic;
```

Magic squares: Constraints

```
constraint
  all_different(magic)

  /\ % rows
  forall(i in 1..n) (
    sum(j in 1..n) (magic[i,j]) = total
  )
  /\ % columns
  forall(j in 1..n) (
    sum(i in 1..n) (magic[i,j]) = total
  )
  /\ % main diagonal (/)
  sum(i in 1..n) (magic[i,i]) = total

  /\ % secondary diagonal (\)
  sum(i in 1..n) (magic[i,n-i+1]) = total
;
```

Magic squares: Complete model (slightly shorter)

```
include "globals.mzn";
int: n;
int: total = (n*(n*n + 1)) div 2;

array[1..n,1..n] of var 1..n*n: magic; % decision variables

solve satisfy;

constraint
  all_different(magic)
  /\
  forall(i in 1..n) (
    sum(j in 1..n) (magic[i,j]) = total /\ % rows
    sum(j in 1..n) (magic[j,i]) = total    % columns
  )
  /\ % main diagonal (/)
  sum(i in 1..n) (magic[i,i]) = total
  /\ % secondary diagonal (\)
  sum(i in 1..n) (magic[i,n-i+1]) = total
;
```

Magic squares: Solutions (n=3, all 8 solutions)

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

```
2   7   6
9   5   1
4   3   8
----------
4   3   8
9   5   1
2   7   6
----------
8   1   6
3   5   7
4   9   2
----------
6   1   8
7   5   3
2   9   4
----------
==========
```

Magic squares
Symmetry breaking
(Frenicle standard form)

Symmetry breaking

- For certain problems there are symmetries in the solutions.
- If we are not interested in all solutions, we can break symmetries by some ordering constraint. For example the increasing constraint.
- Can make the solving - sometimes considerably - faster

Magic Square: Frénicle form

Frénicle standard form (after Bernard Frénicle de Bessy):

- The element at position magic[1,1] is the smallest of the four corner elements
- The element at position magic[1,2] is smaller than the element in magic[2,1].
- This removes the 8 symmetries (rotations, flips, etc)

Magic squares: Symmetry breaking, Frénicle form

```
constraint
  magic[1,1] = min([magic[1,1], magic[1,n], magic[n,1], magic[n,n]])
  /\
  magic[1,2] < magic[2,1]
;
```

Magic squares: Number of solutions

N	W/o symmetry breaking	With Frénicle form	
1	1	-	
2	0	0	
3	8	1	(8/8)
4	7040	880	(7040/8)
5	2202441792	275305224	

Babysittning
Logic puzzle
Element constraint

Element constraint

- CP’s version of indexing an array/matrix
- In MiniZinc, this is stated as

  z = x[y]

- x: an array of integers or decision variables
- y: integer/enum or decision variable
- z: integer/enum or decision variable
- In other CP systems this is called element(y,x,z) etc

Babysittning puzzle (1/2)

(Dell Logic puzzle, 1998)

Each weekday, Bonnie takes care of five of the neighbors' children. The children's names are Keith, Libby, Margo, Nora, and Otto; last names are Fell, Gant, Hall, Ivey, and Jule. Each is a different number of years old, from two to six. Can you find each child's full name and age?

(Next: The hints)

Babysittning puzzle (2/2)

The hints:

1. One child is named Libby Jule.
2. Keith is one year older than the Ivey child, who is one year older than Nora.
3. The Fell child is three years older than Margo.
4. Otto is twice as many years old as the Hall child.

Determine: First name - Last name - Age

Babysitting: Parameters and decision variables

```
include "globals.mzn";

% Parameters
set of int: r = 1..5;
enum first_name = {Keith, Libby, Margo, Nora, Otto};

% Decision variables

array[r] of var 2..6: age;

var r: Fell;
var r: Gant;
var r: Hall;
var r: Ivey;
var r: Jule;
array[r] of var r: last_name = [Fell, Gant, Hall, Ivey, Jule];
% For the presentation
array[r] of string: last_name_s = ["Fell", "Gant", "Hall", "Ivey", "Jule"];
```

Babysitting: Constraints

```
constraint
  all_different(last_name) /\
  all_different(age)   /\

  %  1. One child is named Libby Jule.
  Jule = Libby /\

  %  2. Keith is one year older than the Ivey child, who is one
  %     year older than Nora.
  Keith != Ivey /\ Ivey != Nora /\
  age[Keith] = age[Ivey] + 1 /\ % element with decision variables
  age[Ivey] = age[Nora] + 1 /\

  %  3. The Fell child is three years older than Margo.
  Fell != Margo /\
  age[Fell] = age[Margo] + 3 /\

  %  4. Otto is twice as many years old as the Hall child.
  Otto != Hall /\
  age[Otto] = age[Hall] * 2;
```

Babysitting: Solution

```
first name: {Keith, Libby, Margo, Nora, Otto}
last_name : [1, 4, 3, 5, 2] % lookup
age       : [5, 6, 2, 3, 4]

Keith Fell (5 yo)
Libby Jule (6 yo)
Margo Hall (2 yo)
Nora Gant (3 yo)
Otto Ivey (4 yo)
----------
==========
```

```
last_name_s = ["Fell", "Gant", "Hall", "Ivey", "Jule"];
```

Babysitting: Output section

```
output
[
  "first name: \(first_name)\n",
  "last_name : \(last_name)\n" ++
  "age       : \(age)\n\n"
]
++
[
 "\(first_name[i]) " ++
 % Lookup of last name
 [last_name_s[j] | j in r where fix(last_name[j]) = i][1] ++ " " ++
 "(\(age[i]) yo)\n"
 | i in r
];
```

Global constraints

Special designed algorithm for common constraints.

- `all_different`: all values must be distinct
- `element`: decision variables as indices in an array (as z=x[y])
- `increasing`: ordered values, symmetry breaking
- `global_cardinality`: counting the occurrence of values
- `cumulative`: scheduling
- `regular`: automata / regular expression
- `table`: allow only certain combinations of decision variables

Divisible by 1 to 9 Predicates

Divisible by 1 to 9

- Find a 10 digit number that uses each of the digits 0 to 9 exactly once and where the number formed by the first n digits of the number is divisible by n.

  (Source: Classic, via MindYourDecisions)

Divisible by 1 to 9

- Let A, B, C, D, E, F, G, H, I, J be 10 different digits (with domains 0..9). Then

  A mod 1 = 0

  AB mod 2 = 0

  ...

  ABCDEFGHI mod 9 = 0

  ABCDEFGHIJ mod 10 = 0

Divisible by 1 to 9

- One approach would be the approach we used in SEND+MORE=MONEY:
- A mod 1 = 0 /\
  (A*10 + B) mod 2 = 0 /\
  ...
  (A*... + B* ... + C* ... + J) mod 10 = 0
- But that’s no fun. Let’s automate this using a predicate.

Divisible by 1 to 9: Predicate to_num

```
/*
  to_num(a, n, base)
  Ensure that the digits in array `a` corresponds to the number `n`,
  in base `base`.
  Both `a` and/or `n` can be decision variables.
  `base` is fixed

  Example: to_num([1,2,3], 123, 10).

*/
predicate to_num(array[int] of var int: a, var int: n, int: base) =
  let {
    int: len = length(a)
  } in
  n = sum(i in 1..len) ( base^(len-i) * a[i] )
;
```

Divisible by 1 to 9: Parameters, decision variables

```
int: base;
int: n = base;
int: m = ceil(pow(n,base))-1;    % 999999999 for base 10: 10^10-1

% Decision variables
array[1..n] of var 0..n-1: x;    % the digits: 0..9
array[1..n] of var 0..m: t;      % the numbers. t[n] contains the answer


base = 10;
```

Divisible by 1 to 9: Constraints

```
constraint
  all_different(x) /\

  %
  % ensure that x[1..1] is divisible by 1
  % ensure that x[1..2] is divisible by 2
  % ...
  % ensure that x[1..9] is divisible by 9
  % ensure that x[1..10] is divisible by 10
  %
  forall(i in 1..n) (
    % t[i] corresponds to the number for x[1..i]
    to_num(x[1..i], t[i], base) /\
    t[i] mod i = 0 % divisibility
  );
```

Divisible by 1 to 9: Solution

```
base: 10
x: [3, 8, 1, 6, 5, 4, 7, 2, 9, 0]
t: [3, 38, 381, 3816, 38165, 381654, 3816547, 38165472, 381654729, 3816547290]
----------
==========

% Another base:
base: 8
x: [5, 2, 3, 4, 7, 6, 1, 0] % ← base 8 digits
t: [5, 42, 339, 2716, 21735, 173886, 1391089, 11128712]  % ← base 10 numbers
----------
base: 8
x: [3, 2, 5, 4, 1, 6, 7, 0]
t: [3, 26, 213, 1708, 13665, 109326, 874615, 6996920]
----------
base: 8
x: [5, 6, 7, 4, 3, 2, 1, 0]
t: [5, 46, 375, 3004, 24035, 192282, 1538257, 12306056]
----------
==========
```

Furniture moving
The “serious example”
Scheduling

Furniture moving

- Requirements
  - Piano: 3 persons, 30 min
  - Chair: 1 person, 10 min
  - Bed: 3 persons, 15 min
  - Table: 2 persons, 15 min
- Precedence constraint: The bed must be moved before the piano
- (From Marriott & Stuckey: “Programming with constraints”, 1998)

Furniture moving: Variables and data

```
include "globals.mzn";

enum Furnitures = {Piano, Chair, Bed, Table};
int: n; % number of things
int: upper_limit;
array[1..n] of int: durations;
array[1..n] of int: resources;
array[1..n] of string: names;

% decision variables
array[1..n] of var 0..upper_limit: start_times;
array[1..n] of var 0..upper_limit*2: end_times;
var 0..100: num_persons;
var 0..100: end_time;

% data
n = 4;
upper_limit = 160;
durations = [30,10,15,15];
resources = [3,1,3,2];
names     = ["Piano","Chair","Bed","Table"];
```

Furniture moving: Constraints

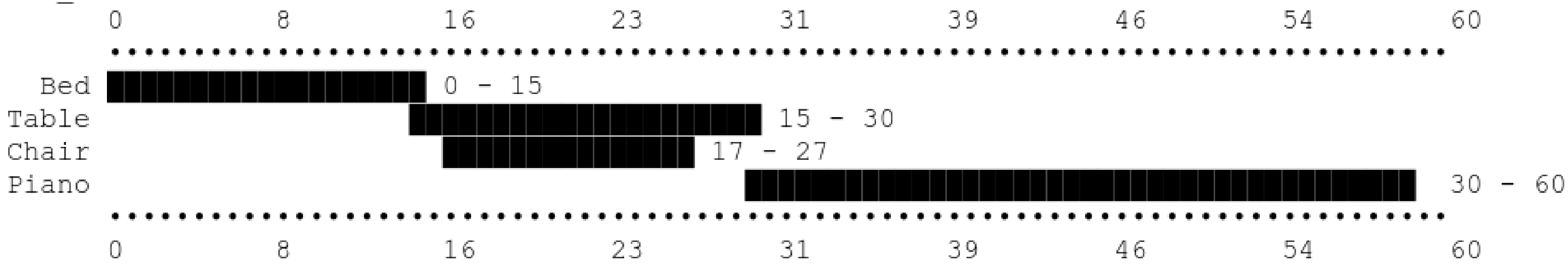
```
constraint

   % cumulative(start_times,durations,required_resources,max_resource)
   cumulative(start_times, durations, resources, num_persons)

   /\ % calculate end time for each task
   forall(i in 1..n) (end_times[i] = start_times[i] + durations[i])

   /\ % first job starts at time 0
   min(start_times) = 0

   /\
   end_time = max(end_times)

   /\ % move the bed before the piano
   end_times[Bed] < start_times[Piano]

   /\ % max number of people
   num_persons <= 4
;
```

Furniture moving: (multi) objective and output

```
% …

solve minimize num_persons * end_time; % multi-objective

output [
       "num_persons: \(num_persons)\n”,
       "resources  : \(resources)\n",
       "start_times: \(start_times)\n",
       "durations  : \(durations)\n",
       "end_times  : \(end_times)\n",
       "end_time   : \(end_time)\n",
       show_gantt(start_times,durations,names)
];
```

Furniture moving: Solution for minimize num_persons*end_time

```
num_persons: 3
resources   : [3, 1, 3, 2]
start_times: [30, 17, 0, 15]
durations   : [30, 10, 15, 15]
end_times   : [60, 27, 15, 30]
end_time    : 60
```



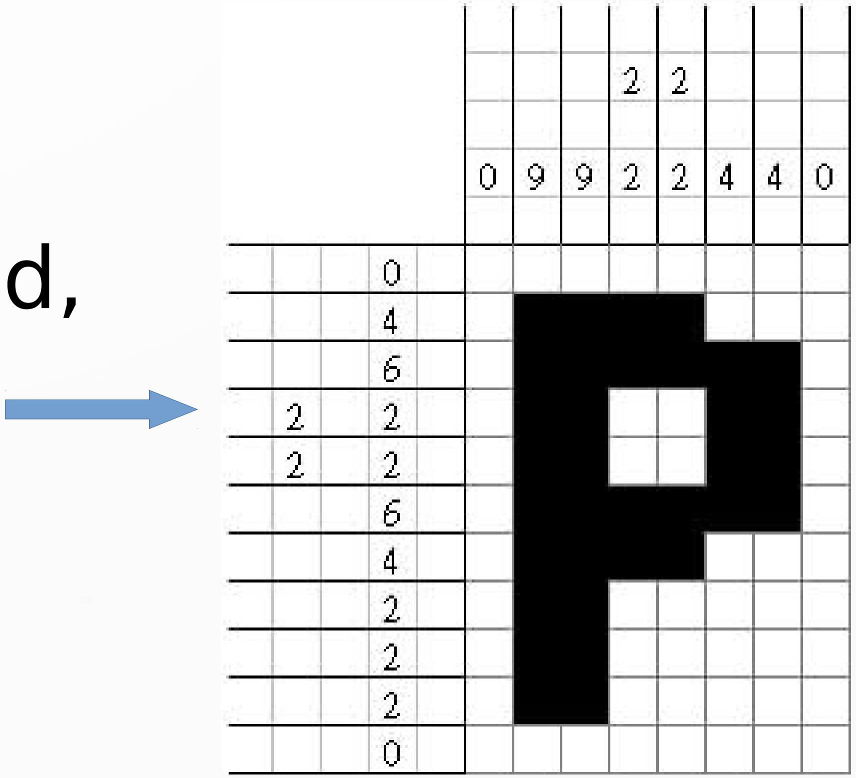

```
----------
----------
```

Nonogram
Regular constraint

Nonogram

https://www.csplib.org/Problems/prob012/

- A grid puzzle where the hints are the number of chunks of filled cells
- The hint “2 2” means that there must be two cells filled, followed by at least one empty cell, followed by two filled cells.



Source: Wikipedia

Nonogram - regexes

- Regular expressions to the rescue!
- We model this as a grid of 0s (empty cells) and 1s (filled cells).
- The hint “2 2” can then be translated to the regex
  ”0*110+110*”
  or
  ”0*1{2}0+1{2}0*”
- The global constraint `regular` supports this.

Nonogram: Setup and output

```
include "globals.mzn";

% Parameters
int: r; % number of rows
int: c; % number of columns
array[1..r] of string: rows; % row hints
array[1..c] of string: cols; % column hints

% Decision variables
array[1..r,1..c] of var 0..1: x; % 1: filled, 0: not filled

solve satisfy;

output [
  if j = 1 then "\n" else "" endif ++
     if fix(x[i,j]) = 0 then " " else "#" endif
  | i in 1..r, j in 1..c
] ++ [ "\n" ];
```

Nonogram: Problem instance (the “P” instance)

```
r = 11; c = 8;
rows = ["0+",                             %   0
        "0* 1{4} 0*",                     %   4
        "0* 1{6} 0*",                     %   6
        "0* 1{2} 0+ 1{2} 0*",             % 2 2
        "0* 1{2} 0+ 1{2} 0*",             % 2 2
        "0* 1{6} 0*",                     %   6
        "0* 1{4} 0*",                     %   4
        "0* 1{2} 0*",                     %   2
        "0* 1{2} 0*",                     %   2
        "0* 1{2} 0*",                     %   2
        "0+"];                            %   0
cols = [
        "0+",                             %   0
        "0* 1{9} 0*",                     %   9
        "0* 1{9} 0*",                     %   9
        "0* 1{2} 0+ 1{2} 0*",             % 2 2
        "0* 1{2} 0+ 1{2} 0*",             % 2 2
        "0* 1{4} 0*",                     %   4
        "0* 1{4} 0*",                     %   4
        "0+"];                            %   0
```

Nonogram: Constraints

```
constraint
  % Row hints
  forall(i in 1..r) (
    regular([x[i,j] | j in 1..c], rows[i])
  )
  /\
  % Column hints
  forall(j in 1..c) (
    regular([x[i,j] | i in 1..r], cols[j])
  )
;
```

Nonogram: Solution

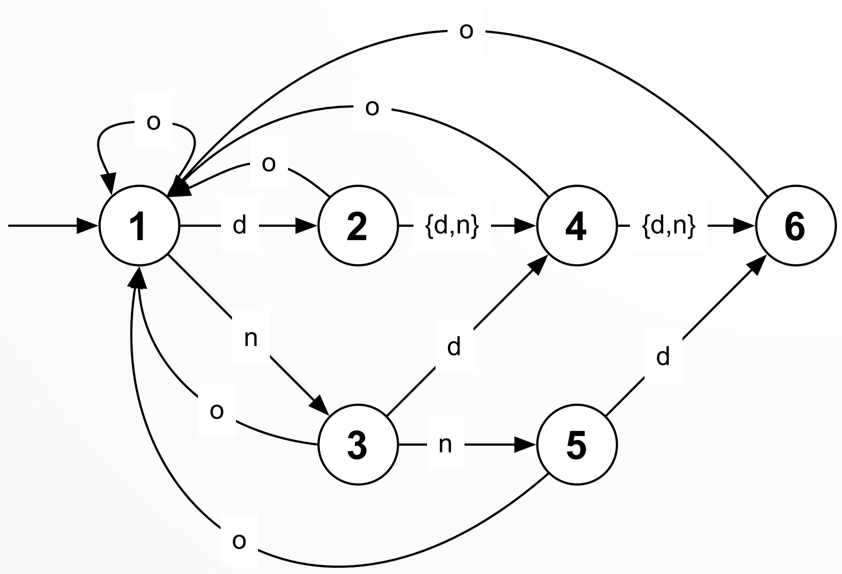
```
 ####
 ######
 ##  ##
 ##  ##
 ######
 ####
 ##
 ##
 ##

----------
==========
```

Regular constraint

- regular/2 is a wrapper for the general regular/6 constraint: a constraint for an automaton
  `regular(automaton, n_states, input_max, transition,initial_state,accepting_states)`
- Rostering, scheduling, sequencing, etc.
  And Puzzles: pentonomies, the 3 jugs problem, etc
- My original Nonogram solver used this version of regular: quite hairy MiniZinc code to convert hints to automaton

Regular constraint

Simple automaton for nurse rostering shifts:



Shifts d:day, n:night, o:off (From the MiniZinc Tutorial)

Crossword
Table constraint

Crossword

Problem instance from Bratko "Prolog Programming for AI", 4th ed (2011, p 27)

L1	L2	L3	L4	L5	
L6		L7		L8	
L9	L10	L11	L12	L13	L14
L15				L16	

```
% Words that may be used in the solution

word( d,o,g).          word( r,u,n).          word( t,o,p).          word( f,i,v,e).
word( f,o,u,r).        word( l,o,s,t).        word( m,e,s,s).        word( u,n,i,t).
word( b,a,k,e,r).      word( f,o,r,u,m).      word( g,r,e,e,n).      word( s,u,p,e,r).
word( p,r,o,l,o,g).    word( v,a,n,i,s,h).    word( w,o,n,d,e,r).    word( y,e,l,l,o,w).
```

The table constraint

- Restricts the allowed assignments for a collection of decision variables.
- The constraint

```
table([A,B,C,D], array2d(1..3,1..4,
                        [1,2,3,4,
                         2,3,4,1,
                         3,4,1,2]));
```

restricts the variables A, B, C, and D to be either
A=1, B=2, C=3, D=4 or
A=2, B=3, C=4, D=1 or
A=3, B=4, C=1, D=2

Crossword: Data, the words (skipping some declarations)

```
enum alpha = {a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z};
words3 = array2d(1..num_words3, 1..3,
     [d,o,g,
      r,u,n,
      t,o,p]);
words4 = array2d(1..num_words4, 1..4,
     [f,i,v,e,
      f,o,u,r,
      l,o,s,t,
      m,e,s,s,
      u,n,i,t]);
words5 = array2d(1..num_words5, 1..5,
     [b,a,k,e,r,
      f,o,r,u,m,
      g,r,e,e,n,
      s,u,p,e,r]);
words6 = array2d(1..num_words6, 1..6,
     [p,r,o,l,o,g,
      v,a,n,i,s,h,
      w,o,n,d,e,r,
      y,e,l,l,o,w]);
```

Crossword: Data, the problem instance

```
%
%  L1   L2    L3   L4    L5   XXX
%  L6   XXX   L7   XXX   L8   XXX
%  L9   L10   L11  L12   L13  L14
%  L15  XXX   XXX  XXX   L16  XXX
%

problem = array2d(1..rows, 1..cols,
   [ 1,    2,    3,    4,    5,    0,
     6,    0,    7,    0,    8,    0,
     9,   10,   11,   12,   13,   14,
    15,    0,    0,    0,   16,    0]);
```

Crossword: Constraints

```
%
%  L1   L2   L3   L4   L5   XXX
%  L6   XXX  L7   XXX  L8   XXX
%  L9   L10  L11  L12  L13  L14
%  L15  XXX  XXX  XXX  L16  XXX
%

%
% Find the words
%
constraint
   % rows
   table([L[1],L[2],L[3],L[4],L[5]], words5)             /\
   table([L[9],L[10],L[11],L[12],L[13],L[14]], words6)   /\

   % columns
   table([L[1],L[6],L[9],L[15]], words4)                 /\
   table([L[3],L[7],L[11]], words3)                      /\
   table([L[5],L[8],L[13],L[16]], words4)
;
```

Crossword: Solution (unique)

```
%
%   L1   L2   L3   L4   L5   XXX
%   L6   XXX  L7   XXX  L8   XXX
%   L9   L10  L11  L12  L13  L14
%   L15  XXX  XXX  XXX  L16  XXX
%
```

```
f o r u m _
i _ u _ e _
v a n i s h
e _ _ _ s _

----------
==========
```

L1	L2	L3	L4	L5	
L6		L7		L8	
L9	L10	L11	L12	L13	L14
L15				L16	

```
% Words that may be used in the solution

word( d,o,g).        word( r,u,n).         word( t,o,p).          word( f,i,v,e).
word( f,o,u,r).      word( l,o,s,t).       word( m,e,s,s).        word( u,n,i,t).
word( b,a,k,e,r).    word( f,o,r,u,m).     word( g,r,e,e,n).      word( s,u,p,e,r).
word( p,r,o,l,o,g).  word( v,a,n,i,s,h).   word( w,o,n,d,e,r).    word( y,e,l,l,o,w).
```

Crossword: Larger instances

- In 2011, I did some experiments with crossword grids of different sizes (5x5..23x23) and a much larger word list
- MiniZinc: http://www.hakank.org/minizinc/crossword3/
- In Picat: http://hakank.org/picat/crossword3/

Crossword: Problem #39 21x21 chars (* is a blank)

_ _ _ _ _ _ _ * _ _ _ _ _ * _ _ _ _ _ _ _
_ _ _ _ _ _ _ * _ _ _ _ _ * _ _ _ _ _ _ _
_ _ _ _ _ _ _ * _ _ _ _ _ * _ _ _ _ _ _ _
_ _ _ * _ _ _ _ _ _ _ _ _ _ _ _ _ * _ _ _
_ _ _ _ * _ _ _ _ _ _ _ * _ _ _ * _ _ _ _
_ _ _ _ _ * _ _ _ _ _ * _ _ _ * _ _ _ _ _
_ _ _ _ _ _ * _ _ _ * _ _ _ * _ _ _ _ _ _
* * * _ _ _ _ _ _ * _ _ _ _ _ _ _ _ * * *
_ _ _ _ _ _ _ _ * _ _ _ _ _ _ _ * _ _ _ _
_ _ _ _ _ _ _ * _ _ _ _ _ _ _ * _ _ _ _ _
_ _ _ _ _ _ * _ _ _ _ _ _ _ * _ _ _ _ _ _
_ _ _ _ _ * _ _ _ _ _ _ _ * _ _ _ _ _ _ _
_ _ _ _ * _ _ _ _ _ _ _ * _ _ _ _ _ _ _ _
* * * _ _ _ _ _ _ _ _ * _ _ _ _ _ _ * * *
_ _ _ _ _ _ * _ _ _ * _ _ _ * _ _ _ _ _ _
_ _ _ _ _ * _ _ _ * _ _ _ _ _ * _ _ _ _ _
_ _ _ _ * _ _ _ * _ _ _ _ _ _ _ * _ _ _ _
_ _ _ * _ _ _ _ _ _ _ _ _ _ _ _ _ * _ _ _
_ _ _ _ _ _ _ * _ _ _ _ _ * _ _ _ _ _ _ _
_ _ _ _ _ _ _ * _ _ _ _ _ * _ _ _ _ _ _ _
_ _ _ _ _ _ _ * _ _ _ _ _ * _ _ _ _ _ _ _

Crossword: Problem #39 Solution (English words)

```
salmons*imams*corrupt
amoebic*marco*oceania
leonine*pucks*stapler
ask*tenderhearted*erg
bloc*sterile*oat*care
lauri*spicy*cur*corot
entomb*ale*tot*mounts
***spears*fruition***
prosiest*diurnal*tags
limbers*palsied*merle
averts*arieses*garden
taney*smarter*cartons
else*junkies*sauterne
***diapered*berlin***
amping*sis*cat*snooze
gains*fit*barth*array
agog*bra*forbear*sane
inn*electioneered*nil
needles*altar*perigee
steuben*beige*evilest
tornado*steed*repasts
```

Crossword: Problem #39 (Swedish words)

```
absiden*kosta*avkylas
mikaela*älvor*variant
tvingad*raabe*bringor
mal*skjortlinning*fri
axla*earlens*eda*gödd
nerts*skara*ada*forne
stalin*ass*dra*daddan
***asiens*barnbeck***
skostans*datafel*ädel
mambons*tidebön*anala
albins*bevarar*monsun
klene*synodal*nyrakad
sard*bestred*multnats
***nånstans*bostad***
skvimp*ena*dat*anette
tvina*ord*durka*snara
rigg*ren*pianino*agar
yls*betacellulosa*gul
kleresi*allen*ragtime
erlades*slang*aningar
rasmark*herse*knotans
```

Magic Sequence
Redundant constraints
Reversibility

Redundant constraints

- Sometimes it is possible to add extra – **redundant** – constraints which sometimes can speed things up
- They do not remove any solutions from the “base model”
- Contrast with **symmetry breaking** constraints which also often speed things up, but they remove solutions

Magic sequence

https://www.csplib.org/Problems/prob019/

- A **magic sequence** of length N is a sequence of integers x[0] . . x[N-1] between 0 and N-1, such that for all i in 0 to N-1, the number i occurs exactly x[i] times in the sequence.
- For n= 10

  `6,2,1,0,0,0,1,0,0,0`

  is a magic sequence since ‘0’ occurs 6 times, ‘1’ occurs twice, ‘6’ occurs 1 time (and the rest 0 times)
- This is a self referential sequence

Magic sequence: First model (“direct” encoding)

```
int: n
array[0..n-1] of var 0..n-1: s;

solve satisfy;

constraint
  forall(i in 0..n-1) (
    s[i] = sum([s[j] = i | j in 0..n-1])
  )
;
```

Quite straightforward: the value of s[i] is the number of occurrences in s which contains the value i.

Magic sequence: Second model, add redundant constraints

```
int: n
array[0..n-1] of var 0..n-1: s;

solve satisfy;

constraint
  forall(i in 0..n-1) (
    s[i] = sum([s[j] = i | j in 0..n-1])
  )
  /\
  sum(s) = n
  /\
  sum(i in 0..n-1) (s[i]*i) = n
;
```

Adding some ‘redundant’ constraints to speed up the search:
- the sum of s is n
- the sum of s[i]*i is also n

Magic sequence: Third model, using global_cardinality

```
int: n
array[0..n-1] of var 0..n-1: s;

solve satisfy;

constraint

  global_cardinality(s,array1d(0..n-1, index_set(s)), s)
  /\
  sum(s) = n
  /\
  sum(i in 0..n-1) (s[i]*i) =n
;
```

Replace the first sum with the global constraint global_cardinality
(a.k.a. global_cardinality_count):

```
    global_cardinality(a,cover,counts)
```

where counts[i] is the number of occurrences of cover[i] (here 0..n-1) in array a

Magic sequence: Comparing models (with Gecode solver)

N	Model	Time (s)
10	model1	0.15s
10	model2	**0.08s**
10	model3	0.20s
100	model1	0.41s
100	model2	0.40s
100	model3	**0.13s**
500	model1	17.49s
500	model2	10.45s
500	model3	**0.15s**
1000	model2	43.05s
1000	model3	**0.37s**
10000	model3	**31.78s**

(Removed models which timed out: > 60s)

Reversibility

- A.k.a. bidirectionality, multidirectionality (cf Prolog)
- A decision variable can be input and/or output
- Given the decision variables A, B, and C and constraint A + B = C
  * Known A and B → C
  * Known B and C → A
  * Known A and C → B
  * Known A → Domain reduction in B and C (perhaps)

Conclusions/Summary

Conclusions/Summary

Constraint Programming / Modeling

- Powerful
- Is fun
- Can be used to explore combinatorial problems
- A special mindset is required
- Though it’s not a silver bullet. Sometimes special algorithms might be faster or better suited.

More on CP

References (mine)

- Homepage: http://hakank.org/
- My MiniZinc page: http://hakank.org/minizinc/
- My Picat page: http://hakank.org/picat/
- Common CP models: http://hakank.org/common_cp_models/
- The Picat book http://picat-lang.org/picatbook2015.html (PDF available for free)

References

- CP/MiniZinc-courses (Coursera):
  - Basic Modeling for Discrete Optimization
  - Solving Algorithm for Discrete Optimization
  - Advanced Modeling for Discrete Optimization
- The NordConsNet site
  http://www.it.uu.se/research/NordConsNet
  has a lot of information and references on CP and constraint modeling

Some Constraint systems

Some great Constraint systems/solvers (not necessary CP)

- **MiniZinc**: The system used in this talk
- **Google OR-tools** (Python, C#, C++): Often very fast (CP-SAT)
- **Chuffed** (in MiniZinc)
- Gecode (C++)
- Choco, JaCoP (Java)
- **CPMPy** (Python): high level wrapper around MiniZinc, OR-tools, PySAT and Z3
- Prolog (CLP): SICStus Prolog, ECLiPSe CLP, SWI-Prolog, etc
- Microsoft's Z3 theorem prover: Many nice features
- **Picat** - my "Thinking language"
  ("Prolog" + constraints + functions and imperative constructs. CP/SAT/SMT/MIP constraint solvers)

Some CP related conferences
CP 2023
NordConsNet 2023

Conferences

- The 29th International Conference on Principles and Practice of Constraint Programming (“CP 2023”)
  August 27 - 31, 2023, Toronto, Canada
  https://cp2023.a4cp.org/index.html
- The Nordic Network for researchers and practitioners of Constraint programming (NordConsNet)
  June 8 – 9, 2023, Odense, Denmark
  https://event.sdu.dk/nordconsnet2023/

  (See http://www.it.uu.se/research/NordConsNet)

Thank you!
Questions?

http://hakank.org/cp_mensa_2023/

Post talk slides
(including quite a few other models)

all_different_except_0
Reification

Reification

“Reasoning” about constraints/boolean variables

- Implication: constraint1 → constraint2
- Equivalence: constraint1 ↔ constraint2
- not
- /\: and
- \/: or
- false: 0, true: 1

all_different_except_0

```
%
% all_different_except_0(x)
% Ensures that all values that are != 0 are distinct.
%
% Note: This constraint has another definition in the MiniZinc
%       distribution.
%
predicate all_different_except_0(array[int] of var int: x) =
  foreach(i, j in index_set(x) where i < j) (
    (x[i] != 0 /\ x[j] != 0) -> x[i] != x[j]
  );
```

Selected publications

- Soto, Kjellerstrand, et.al: Cell formation in group technology using constraint programming and Boolean satisfiability (2012)
- Kjellerstrand: Picat: A logic-based multi-paradigm language (2014)
- Zhou, Kjellerstrand: Solving several planning problems with Picat (2014)
- Zhou, Kjellerstrand: The Picat-SAT compiler (2016)
- Rohner, Kjellerstrand: Using logic programming for theory representation and scientific inference (2021)
- And ...

Constraint Modeling

This talk focuses on the modeling part and should really be called

”**Constraint Modeling** -
Solving combinatorial puzzles when you are lazy”

Important features of CP

- Propagation
- Global constraints
- Reification
- Reversibility
- Symmetry breaking
- Redundant constraints

Debugging in CP

- Test early and often
  While learning CP: test after adding each constraint
- Check the domains
- First test a small instance for which you know the answer
- If the model does not work:
  - remove one constraint after another and test again
  - check the domains again

Conclusions/Summary

Compared to imperative programming languages:

- There are no (re)assignments: if the model tries to assign a decision variable with two different values then it is a failure → backtracks to another possible solution.
- Forall loops are not like imperative for loops; they are only used to create constraints (in arrays)
- There are no while loops
- Debugging might be harder in CP than in imperative programming languages.

MiniZinc syntax

Syntax: Parameters/data

- Parameters, fixed data (hints)

```
int: n=4;
array[1..n] of int: a =[1,2,3,4]; % default 1-based

% When using a specific datafile (.dzn)

% This is in the model (.mzn) file
int: m;
array[1..m] of int: y;

% In the data .dzn file
m = 5;
y = [2,3,4,5,6];
```

Syntax:Variables / Domains

- **Decision variables** with an apropriate finite domain (integers, enums).
- The **unknowns** that we want to find out the values for.
- Beware: Not like variables in Python, Java, C++, etc.

```
var bool: b;
array[1..n] of var 1..n: a;
array[1..n, 1..n] of var 1..n: x; % 2d array
var int: z = sum(x);

enum vals = {A,B,C,D};
array[1..3] of var vals: y;
```

Syntax: Constraints

- **Constraints**: Connecting decision variables

```
c = a + b                   % arithmetic constraint

% Global constraints
all_different(x)
increasing(x)               % symmetry breaking
z = x[y]                    % element constraint

% Reification
x[1] > 10 -> x[2] < 2 % implication
a != 1 <-> b = 1            % equivalence
```

- `include "globals.mzn" % definitions for constraints`

Syntax:Solve

- Solving / optimization

```
% any solution, 1, 2, … all solutions
solve satisfy;

% optimization
solve minimize z; % or solve maximize y;

% search heuristics / labeling
solve::int_search(x,first_fail,indomain_min)
satisfy;
```

Syntax: Output section

- Output section

```
output [ show(x) ];

output [ "\(x)\n" ];

output [
  "\(i): \(x[i])\n"
  | i in 1..n
]
++
["\(z)\n"];
```

Domains

- Domains

```
var 0..9: a;
array[1..n] of var 0..9: x;
```

- Restricts the possible values of the decision variable, here the integers 0..9.
- Used in the solving phase where the current domain is **propagated** to the solver and can be **reduced** by activating the constraints. We see an example on this soon.
- Try to get the domains as small as possible (but not smaller)

Global constraints

- Special crafted (efficient) algorithms for common types of constraints, common structures
- Kind of “Patterns” / “Tool of Thought” when modeling
- `all_different(x)`
  Ensure that all values in the array x are distinct
- We will see more global variables in this talk
- Global Constraint Catalog (almost 300 different global constraints)
  https://sofdem.github.io/gccat/gccat/titlepage.html

CP: Overview

- Searches through the complete search space with intelligence: constraint propagation, domain reduction, and search heuristics (pruning the search space)
- Most CP solvers use some smart techique for searching and pruning the search tree.

CP: The declarative ideal

"Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it."

[E. Freuder, "In Pursuit of the Holy Grail", 1997]

Sudoku: 25x25 problem instance

11	23	13	10	19	16	6	2	24	7	5	9	1	20	17	15	8	18	25	3	4	12	21	22	14
15	16	_	22	_	11	8	_	_	_	25	_	14	_	_	_	12	19	_	_	17	_	_	_	_
_	_	_	_	_	_	_	_	_	_	_	16	_	4	_	17	_	13	_	24	_	23	19	10	2
_	_	_	_	_	19	_	14	23	4	_	21	6	22	10	_	11	_	2	_	_	_	_	_	_
17	14	_	_	2	_	_	13	12	_	_	_	_	_	15	4	20	22	10	_	11	_	9	24	8
22	_	_	_	_	6	2	_	_	_	4	7	12	1	9	_	_	_	_	_	_	14	5	_	_
_	18	2	_	8	22	_	19	16	21	_	_	_	10	13	23	_	_	20	_	_	3	_	15	7
_	_	17	3	_	5	_	_	8	9	_	_	_	_	18	_	19	_	_	_	_	_	23	21	_
1	11	_	_	9	_	15	10	25	_	6	_	23	_	_	_	_	5	3	7	_	17	_	_	24
_	_	_	_	_	_	1	_	_	23	_	_	_	24	_	_	_	21	12	_	6	8	_	25	16
20	24	10	_	15	23	11	17	_	_	_	_	_	7	_	12	_	_	_	_	_	22	_	_	6
4	5	_	14	12	25	_	18	_	_	23	_	15	_	19	1	_	_	_	22	20	_	7	9	_
18	_	21	_	_	8	_	24	_	_	9	_	25	_	_	_	10	_	_	_	2	_	1	19	_
_	_	6	2	1	_	13	_	22	_	_	_	_	_	11	8	21	16	_	_	25	_	_	12	17
_	17	25	_	23	7	14	_	21	1	_	_	_	_	3	_	_	11	_	_	24	_	16	4	5
_	_	_	_	11	18	24	_	_	_	_	5	_	12	_	25	_	_	_	15	23	4	8	14	_
_	_	_	15	21	_	_	_	_	_	2	_	13	17	_	_	1	7	_	_	5	9	24	_	_
_	_	18	_	22	15	_	_	2	16	_	23	_	_	_	10	6	24	_	17	12	_	25	11	_
7	2	_	1	_	_	21	_	_	_	18	22	_	9	6	14	_	4	5	16	_	_	_	_	_
_	_	9	_	_	_	7	22	_	_	10	_	24	_	_	_	18	_	_	_	21	_	_	_	_
_	12	_	19	10	_	_	_	_	_	_	_	_	_	1	_	_	_	_	_	14	_	4	8	_
24	_	11	18	_	_	_	_	_	_	_	25	17	21	_	6	_	_	1	_	_	_	_	5	12
16	6	22	_	_	_	23	4	15	18	8	_	_	_	20	_	_	17	_	14	_	_	_	_	_
_	21	_	_	4	_	9	1	7	_	_	_	_	11	14	_	16	8	15	_	22	_	18	_	_
8	15	_	_	_	_	_	_	5	_	24	3	_	_	4	_	_	_	9	_	_	_	_	_	20

Sudoku: 25x25 solution (PicatSAT: 0.2s)

11	23	13	10	19	16	6	2	24	7	5	9	1	20	17	15	8	18	25	3	4	12	21	22	14
15	16	4	22	18	11	8	21	20	10	25	2	14	13	24	7	12	19	23	9	17	5	6	1	3
21	1	5	20	25	3	18	15	9	22	11	16	8	4	12	17	14	13	6	24	7	23	19	10	2
3	8	12	9	24	19	17	14	23	4	7	21	6	22	10	16	11	1	2	5	15	18	20	13	25
17	14	7	6	2	1	5	13	12	25	3	18	19	23	15	4	20	22	10	21	11	16	9	24	8
22	19	23	21	13	6	2	3	17	24	4	7	12	1	9	11	15	25	16	8	18	14	5	20	10
25	18	2	24	8	22	4	19	16	21	14	11	5	10	13	23	17	6	20	1	9	3	12	15	7
6	10	17	3	16	5	12	7	8	9	15	20	2	25	18	22	19	14	24	13	1	11	23	21	4
1	11	14	12	9	20	15	10	25	13	6	8	23	16	21	18	4	5	3	7	19	17	22	2	24
5	20	15	4	7	14	1	11	18	23	17	19	3	24	22	9	2	21	12	10	6	8	13	25	16
20	24	10	13	15	23	11	17	19	3	21	1	16	7	2	12	5	9	4	25	8	22	14	18	6
4	5	16	14	12	25	10	18	6	2	23	13	15	8	19	1	24	3	17	22	20	21	7	9	11
18	22	21	11	3	8	16	24	4	12	9	17	25	14	5	20	10	15	7	6	2	13	1	19	23
19	7	6	2	1	9	13	5	22	15	20	24	4	18	11	8	21	16	14	23	25	10	3	12	17
9	17	25	8	23	7	14	20	21	1	12	10	22	6	3	2	13	11	19	18	24	15	16	4	5
10	13	19	16	11	18	24	6	3	17	1	5	20	12	7	25	9	2	21	15	23	4	8	14	22
12	25	8	15	21	10	19	23	14	11	2	4	13	17	16	3	1	7	22	20	5	9	24	6	18
14	4	18	5	22	15	20	9	2	16	19	23	21	3	8	10	6	24	13	17	12	7	25	11	1
7	2	24	1	20	12	21	25	13	8	18	22	11	9	6	14	23	4	5	16	10	19	17	3	15
23	3	9	17	6	4	7	22	1	5	10	14	24	15	25	19	18	12	8	11	21	20	2	16	13
13	12	20	19	10	17	3	16	11	6	22	15	7	5	1	21	25	23	18	2	14	24	4	8	9
24	9	11	18	14	13	22	8	10	19	16	25	17	21	23	6	7	20	1	4	3	2	15	5	12
16	6	22	25	5	2	23	4	15	18	8	12	9	19	20	24	3	17	11	14	13	1	10	7	21
2	21	3	23	4	24	9	1	7	20	13	6	10	11	14	5	16	8	15	12	22	25	18	17	19
8	15	1	7	17	21	25	12	5	14	24	3	18	2	4	13	22	10	9	19	16	6	11	23	20

Minesweeper Reversibility

Minesweeper

Minesweeper – in this version – is a simple grid problem:

```
..2.3.
2.....
..24.3
1.34..
.....3
.3.3..
```

Each number represents how many bombs there are in the 8 nearby cells.

The “.” (dot) represents an unknown cell: either a bomb or not bomb.

Where are the bombs?

Minesweeper

```
..2.3.
2.....
..24.3
1.34..
.....3
.3.3..
```

For the green cell, ensure that there are exactly 4 bombs among the 8 (vertical, horizontal, diagonal) neighbours.

A cell with a hint can not be a bomb.

Minesweeper: The setup, parameters and decision variables

```
% >= 0 for number of mines in the Moore neighbourhood
% (vertical, horizontal, and diagonal neighbours)
array[1..r, 1..c] of -1..8: game;  % the hints

% decision variables: 0/1 for no bomb/bomb
array[1..r, 1..c] of var 0..1: mines;

% the hints
int: X = -1; % representing the unknowns in the hints
int: r = 6; % rows
int: c = 6; % column
game = array2d(1..r, 1..c, [
   X,X,2,X,3,X,
   2,X,X,X,X,X,
   X,X,2,4,X,3,
   1,X,3,4,X,X,
   X,X,X,X,X,3,
   X,3,X,3,X,X,
]);
```

Minesweeper: Constraints

```
% game[1..n, 1..n]: the given hints
% mines[1..n, 1..n]: 0/1 where 1 represent a bomb
% X: -1 represents the unknown
constraint
  forall(i in 1..r, j in 1..c) (
    % If the cell contains a hint
    if game[i,j] > X then
      % the number in the hint is the number
      % of all the surrounded bombs
      game[i,j] = sum(a,b in {-1,0,1} where
                                    i+a in 1..r /\
                                    j+b in 1..c /\
                                    (a != 0 \/ b != 0)
                  ) (mines[i+a,j+b])

      /\ % if a hint, then it can't be a bomb
      mines[i,j] = 0

    endif

  );
```

Minesweeper: Solution

```
..2.3.
2.....
..24.3
1.34..
.....3
.3.3..

100001   % 1: Bomb, 0: no bomb
010110
000010
000010
011100
100011
```

Magic squares: solution for 15x15 (0.7s with Gecode)

107	55	213	186	21	140	171	147	114	204	80	49	81	30	97
57	73	44	126	88	154	12	28	35	225	104	200	185	166	198
144	224	90	141	219	153	212	170	217	14	17	3	11	46	34
207	60	158	211	134	45	129	161	61	65	184	102	95	19	64
31	210	117	190	111	131	75	105	4	223	127	115	146	86	24
193	23	139	125	197	196	50	29	222	62	32	214	179	8	26
172	167	175	40	59	176	128	9	165	188	178	37	77	122	2
206	74	13	84	174	116	162	6	203	71	132	83	218	110	43
112	91	48	87	163	157	56	143	180	47	138	195	135	67	76
7	216	53	189	89	191	106	183	78	68	164	79	22	41	209
70	208	98	93	96	20	16	169	5	159	42	155	182	181	201
103	94	160	168	149	99	123	151	100	15	66	25	85	221	136
119	38	187	1	69	51	192	101	121	142	124	173	33	194	150
58	152	52	18	54	39	145	156	108	120	177	63	133	205	215
109	10	148	36	72	27	118	137	82	92	130	202	113	199	220

Furniture moving: Solution for minimize num_persons

```
% One optimal solution of many
num_persons: 3
resources  : [3, 1, 3, 2]
start_times: [70, 23, 55, 40]
durations  : [30, 10, 15, 15]
end_times  : [100, 33, 70, 55]
end_time   :  100
----------
==========


At least 3 people are needed.
- first start time: 23 !
- end_time: 100 !

Can we do better?
a) First start time = 0
b) Better end_time?
```

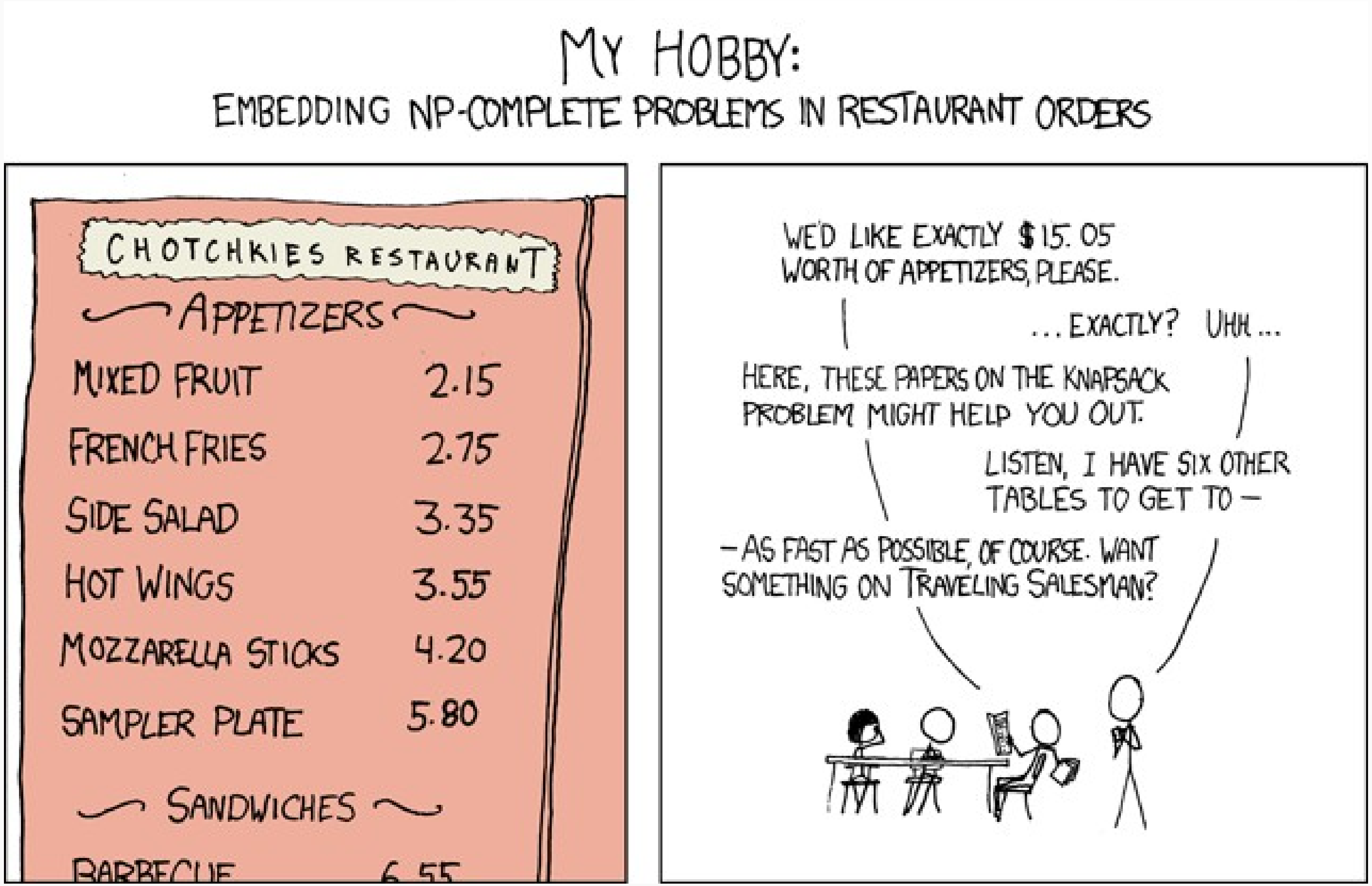
Multi-objective

- In many applications there can be more than one objective, such as
  - minimize the resources AND
  - minimize the end time
  This is called **multi-objective**.
- Alas, MiniZinc does not supports this directly
- One approach is to combine different objectives

XKCD problem #287
subset sum

XKCD #287

From http://xkcd.com/287



XKCD #287: Problem

P: **We'd like exactly $15.05 worth of appetizers,** please.

Waiter: ... exactly? Ummm..

P: Here'm these papers on the Knapsack problem might help you out

Waiter: Listen, I have six other tables to get to -

P: ... as fast as possible, of course. Want something on Traveling Salesman?

XKCD #287

Appetizers	
Mixed Fruit	2.15
French Fries	2.75
Side Salad	3.35
Hot Wings	3.55
Mozarella Sticks	4.20
Sampler Plate	5.80

Since we are using finite domain, we multiply all values with 100: 215, 275, 335, 355, 420, 580.

And the total 15.05: 1505

Subset sum

- This is actually a subset sum problem (not Traveling Salesperson Problem, TSP)
- Given a list of values and a target, find all the values that sums to target.
- Subset sum **is** NP complete, i.e. there’s no general algorithm that can solve arbitrary problems in polynomial time.
  Which does not mean that it’s impossible to solve some of these problems, even large problems.

XKCD #287: Model

```
% parameters
int: num_appetizers;
array[1..num_appetizers] of int: price;
int: total;

% decision variables
array[1..num_appetizers] of var 0..100000: x; % items of each dish

constraint total = sum(i in 1..num_appetizers) (x[i]*price[i]);

solve satisfy;

% data
num_appetizers = 6;
% Multiply by 100 → integers
price = [215, 275, 335, 355, 420, 580];
total = 1505;
```

XKCD #287: Output

```
x = [7, 0, 0, 0, 0, 0];
----------
x = [1, 0, 0, 2, 0, 1];
----------
==========
```

XKCD problem #287
subset sum + optimization

Minimize number of dishes

- Here is a variant of the original problem
- Minimize the number of dishes

XKCD #287: Model minimizing the number of dishes

```
int: num_appetizers;
array[1..num_appetizers] of int: price;
int: total;

array[1..num_appetizers] of var 0..100000: x; % items of each dish
var int: z = sum(x); % sum of the number of dishes

solve minimize z;

constraint total = sum(i in 1..num_prices) (x[i]*price[i]);


num_appetizers = 6;
price = [215, 275, 335, 355, 420, 580]; % Multiply by 100 → integers
total = 1505;

output ["z: \(z)\nx: \(x)\n"];
```

XKCD #287: Model minimizing the number of dishes, output

```
x: [1, 0, 0, 2, 0, 1]
z: 4
----------
==========
```

```
% With the 'fancy' output
z: 4
Mixed Fruit    : 1 ($2.15)
Hot Wings      : 2 ($7.10)
Sampler Plate  : 1 ($5.80)
----------
==========
```

XKCD #287: Fancy output

```
% …
array[1..num_appetizers] of string: name;
% …

name = ["Mixed Fruit","French Fries","Side Salad",
        "Hot Wings","Mozarella Sticks","Sampler Plate"];
output[
  if fix(x[i]) > 0 then
    name[i] ++ "\t: \(x[i]) ($" ++ show_float(3,2,x[i]*price[i]/100) ++ ")\n"
  endif
  | i in 1..num_appetizers
];
```

XKCD #287: Fancy output

```
Mixed Fruit    : 7 ($15.05)
----------
Mixed Fruit    : 1 ($2.15)
Hot Wings      : 2 ($7.10)
Sampler Plate  : 1 ($5.80)
----------
==========
```

Monks and doors
Reification

Reification

“Reasoning” about constraints/boolean variables

- Implication: constraint1 → constraint2
- Equivalence: constraint1 ↔ constraint2
- not
- /\: and
- \/: or
- false: 0, true: 1

Monks and doors

There is a room with four doors and eight monks. One of the doors is an exit. Each monk is either telling a lie or the truth. The monks make the following statements:

Monk 1: Door A is the exit.

Monk 2: At least one of the doors B and C is the exit.

Monk 3: Monk 1 and Monk 2 are telling the truth.

Monk 4: Doors A and B are both exits.

Monk 5: Doors A and B are both exits.

Monk 6: Either Monk 4 or Monk 5 is telling the truth.

Monk 7: If Monk 3 is telling the truth, so is Monk 6.

Monk 8: If Monk 7 and Monk 8 are telling the truth, so is Monk 1.

Which door is an exit and what monk(s) are telling the truth?

Monks and doors: Parameters and decision variables

```
enum doors = {A,B,C,D};
int: num_monks = 8;
% Decision variables
array[doors] of var bool: Door;
array[1..num_monks] of var bool: M;

solve satisfy;
```

Monks and doors: Constraints (1/2)

```
Constraint

  % Monk 1: Door A is the exit.
  (M[1] <-> Door[A]) /\

  % Monk 2: At least one of the doors B and C is the exit.
  (M[2] <-> (Door[B] \/ Door[C]) /\

  % Monk 3: Monk 1 and Monk 2 are telling the truth.
  (M[3] <-> (M[1] /\ M[2])) /\

  % Monk 4: Doors A and B are both exits.
  (M[4] <-> (Door[A] /\ Door[B])) /\

  % Monk 5: Doors A and C are both exits.
  (M[5] <-> (Door[A] /\ Door[C])).
```

Monks and doors: Constraints (2/2)

```
constraint

  % Monk 6: Either Monk 4 or Monk 5 is telling the truth.
  (M[6] <-> (M[4] \/ M[5])) /\

  % Monk 7: If Monk 3 is telling the truth, so is Monk 6.
  (M[7] <-> (M[3] -> M[6])) /\

  % Monk 8: If Monk 7 and Monk 8 are telling the truth, so is Monk 1.
  (M[8] <-> ((M[7] /\ M[8]) -> M[1])) /\

  % Exactly one door is an exit.
  sum(Door) = 1;
```

Monks and doors: Solution

```
door:  A      B      C      D
      [true, false, false, false]
monk: 1      2      3      4      5      6      7     8
      [true, false, false, false, false, false, true, true]
----------
==========


Door A the exist door.
Monks 1, 7, and 8 are telling the truth.
```

Broken weights
Bachet’s weighing problem

Broken weights

- A merchant had a forty pound measuring weight that broke into four pieces as the result of a fall. When the pieces were subsequently weighed, it was found that the weight of each piece was a whole number of pounds and that the four pieces could be used to weigh every integral weight between 1 and 40 pounds. What were the weights of the pieces?

  (Bachet, 1612)

- Assume a balance scale with two pans.

Source: Wikipedia

Broken weights

- In short: Using 4 weights that sum to 40, how can we measure each value 1..40 using a balance scale?

- What are the parameters?
- What are the decision variables and domains?
- How to represent the balance scale?
- What are the constraints?

Broken weights: Parameters, decision variables

```
int: n = 4;                            % the number of different weights
int: m = 40;                           % original/total weight

array[1..n] of var 1..m: weights;  % the weights
% The combinations:
% -1: left side, 1: right side, 0: not used
array[1..m, 1..n] of var -1..1: x;

solve satisfy;
```

Broken weights: Constraints

```
constraint
  sum(weights) = m

  /\ % Ensure that all weights from 1 to 40 (m) can be made.
  forall(w in 1..m) (
    sum([x[w,i]*weights[i] | i in 1..n]) = w
  )

  % symmetry breaking
  /\ increasing(weights);
```

Broken weights: Solution

```
W:     1    3    9   27    % The weights
 1:    1    0    0    0
 2:   -1    1    0    0    % 1 pound in left, 3 pound in right: 3 - 1 = 2
 3:    0    1    0    0
 4:    1    1    0    0
 5:   -1   -1    1    0
 6:    0   -1    1    0
 7:    1   -1    1    0
 8:   -1    0    1    0
 9:    0    0    1    0
...
32:   -1   -1    1    1
33:    0   -1    1    1    % 3 in left, 9 and 27 in right: 27+9-3=33
34:    1   -1    1    1
35:   -1    0    1    1
36:    0    0    1    1
37:    1    0    1    1
38:   -1    1    1    1
39:    0    1    1    1
40:    1    1    1    1
----------
==========
```

Zebra puzzle
“Einstein puzzle type”
Predicates

Zebra puzzle

- 1. There are five houses, each of a different color and inhabited by men of different nationalities, with different pets, drinks, and cigarettes.
- 2. The Englishman lives in the red house.
- 3. The Spaniard owns the dog.
- 4. Coffee is drunk in the green house.
- ...
- 15. The Norwegian lives next to the blue house.
- Who drinks water? And who owns the zebra?

Zebra puzzle: Full problem statement

1. There are five houses, each of a different color and inhabited by men of different nationalities, with different pets, drinks, and cigarettes.
2. The Englishman lives in the red house.
3. The Spaniard owns the dog.
4. Coffee is drunk in the green house.
5. The Ukrainian drinks tea.
6. The green house is immediately to the right of the ivory house.
7. The Old Gold smoker owns snails.
8. Kools are smoked in the yellow house.
9. Milk is drunk in the middle house.
10. The Norwegian lives in the first house on the left.
11. The man who smokes Chesterfields lives in the house next to the man with the fox.
12. Kools are smoked in the house next to the house where the horse is kept.
13. The Lucky Strike smoker drinks orange juice.
14. The Japanese smoke Parliaments.
15. The Norwegian lives next to the blue house.

NOW, who drinks water? And who owns the zebra?

Zebra puzzle: The setup, including helper predicates

```
enum Nationalities= {English,Spanish,Ukrainian,Norwegian,Japanese};
enum Colours       = {Red,Green,Ivory,Yellow,Blue};
enum Animals       = {Dog,Fox,Horse,Zebra,Snails};
enum Drinks        = {Coffee,Tea,Milk,OrangeJuice,Water};
enum Cigarettes    = {OldGold,Kools,Chesterfields,LuckyStrike,Parliaments};
set of int: Houses= 1..5;

array[Nationalities] of var Houses: nation;
array[Colours] of var Houses: colour;
array[Animals] of var Houses: animal;
array[Drinks] of var Houses: drink;
array[Cigarettes] of var Houses: smoke;

% Helper predicates
predicate nextto(var Houses:h1, var Houses:h2) =
       h1 == h2 + 1 \/ h2 == h1 + 1; % or abs(h1-h2) = 1
predicate rightof(var Houses:h1, var Houses:h2) = h1 == h2 + 1;
predicate middle(var Houses:h) = h == 3;
predicate left(var Houses:h) = h = 1;
```

Zebra puzzle: Constraints (full)

```
constraint
    all_different(nation) /\ all_different(colour) /\
    all_different(animal) /\ all_different(drink) /\
    all_different(smoke) /\
    nation[English] = colour[Red] /\ % 2
    nation[Spanish] = animal[Dog] /\ % 3
    drink[Coffee] = colour[Green] /\ % 4
    nation[Ukrainian] = drink[Tea] /\ % 5
    rightof(colour[Green], colour[Ivory]) /\ % 6
    smoke[OldGold] = animal[Snails] /\ % 7
    smoke[Kools] = colour[Yellow] /\ % 8
    middle(drink[Milk]) /\ % 9
    left(nation[Norwegian]) /\ % 10
    nextto(smoke[Chesterfields], animal[Fox]) /\ % 11
    nextto(smoke[Kools], animal[Horse]) /\ % 12
    smoke[LuckyStrike] = drink[OrangeJuice] /\ % 13
    nation[Japanese] = smoke[Parliaments] /\ % 14
    nextto(nation[Norwegian], colour[Blue]); % 15

solve satisfy;
```

Zebra puzzle: Constraints (selected)

```
constraint
    % ...
    % 2. The Englishman lives in the red house.
    nation[English] = colour[Red] /\
    % ...

    % 6. The green house is immediately to the right
    % of the ivory house.
    rightof(colour[Green], colour[Ivory]) /\
    % ...

    % 9. Milk is drunk in the middle house.
    middle(drink[Milk]) /\
    % ...

    % 10. The Norwegian lives in the first house on the left.
    left(nation[Norwegian]) /\

    % ...
```

Zebra: solution

```
nation=[English:3, Spanish:4, Ukrainian:2, Norwegian:1, Japanese:5];
colour=[Red:3, Green:5, Ivory:4, Yellow:1, Blue:2];
animal=[Dog:4, Fox:1, Horse:2, Zebra:5, Snails:3];
Drink =[Coffee:5, Tea:2, Milk: 3, OrangeJuice: 4, Water:1];
Smoke =[OldGold:3, Kools:1, Chesterfields:2, LuckyStrike:4, Parliaments:5];
----------
==========


% The Norwegian drinks water: Drink Water = 1 → Nation 1 = Norwegian
% The Japanese owns the Zebra: Animal Zebra = 5 → Nation 5 = Japanese
```

Langford's number problem
Element, Symmetry breaking

Langford’s number problem

Langford's number problem (CSP lib problem 24)

http://www.csplib.org/prob/prob024/

http://www.dialectrix.com/langford.html

- Arrange 2 sets of positive integers 1..k to a sequence, such that, following the first occurence of an integer i, each subsequent occurrence of i, appears i+1 indices later than the last.
  For example, for k=4, a solution would be 41312432
- K=12: 1,9,1,8,3,12,10,11,3,4,5,9,8,7,4,6,5,10,12,11,2,7,6,2
- Only for k mod 4 == 0 or k mod 4 == 3

Langford’s number problem

Two decision variables:

- Positions: for each index in 1..k: each subsequent occurrence of i, appears i+1 indices later than the last
- Solution: Place the (two) i’s in the assigned positions

Langford’s problem: The model

```
int: k = 4;
set of int: pos_domain = 1..2*k;                 % domain of the positions
array[pos_domain] of var pos_domain: pos; % the positions
array[pos_domain] of var 1..k: sol;          % the solution

constraint
  forall(i in 1..k) (
    % positions:
    % “each subsequent occurrence of i, appears i+1 indices
    % later than the last”
    pos[i+k] = pos[i] + i+1 /\
    all_different(pos)      /\

    % solution: the values in pos[i] and pos[k+i] should both
    % have the value i
    sol[pos[i]]   = i       /\ % element
    sol[pos[k+i]] = i          % element
  )
  % symmetry breaking
  /\ sol[1] < sol[2*k]
;
```

Langford’s problem: Element

```
% ...
constraint
  forall(i in 1..k) (
    pos[i+k] = pos[i] + i+1 /\
    sol[pos[i]]   = i       /\ % element
    sol[pos[k+i]] = i          % element
  )
  % ...
;
```

Ensure that for the two positions pos[i] and pos[k+i] (with k indices apart), the solution (sol) in these positions should both have the value of i.

Langford's problem: Solution (k=4)

```
position: [5, 1, 2, 3, 7, 4, 6, 8]
solution: [2, 3, 4, 2, 1, 3, 1, 4]
----------
==========
```

Langford’s problem: Solution (k=4)

```
%            1  2  3  4  1  2  3  4
position: [5, 1, 2, 3, 7, 4, 6, 8]
solution: [2, 3, 4, 2, 1, 3, 1, 4]
----------
==========



pos[1] = 5 → sol[5] = 1
pos[2] = 1 → sol[1] = 2
pos[3] = 2 → sol[2] = 3
pos[4] = 3 → sol[3] = 4

% pos[i+k] = pos[i] + i+1
pos[1+4=5] = 7 (5+1+1) → sol[7] = 1
pos[2+4=6] = 4 (1+2+1) → sol[4] = 2
pos[3+4=7] = 6 (2+3+1) → sol[6] = 3
pos[4+4=8] = 8 (3+4+1) → sol[8] = 4

% sol[pos[i]]   = i
% sol[pos[i+k]] = i
```

Element constraint

- One of the most common and powerful constraint
- Z = X[Y]
  where X is an array of decision variables , Y and Z are decision variables.
- Given X and Z → Y (reversibility)
- Given pairs of Zs and Ys → X
- 2D arrays: V = X[Y,Z]
- In other CP systems: element(Y,X,Z)

Symmetry breaking

- Pruning symmetric solutions can speed up the solve time.
- For n=4, there are two symmetric solutions and we remove one of them

```
sol[1] < sol[2*k]
```

- solution: [**2**, 3, 4, 2, 1, 3, 1, **4**]

  solution: [**4**, 1, 3, 1, 2, 4, 3, **2**] This is removed
- Global constraints for symmetry breaking: increasing, decreasing, lex_lt, lex2, all_different_except_0, value_precede_chain

Langford’s problem generalized

Langford: generalized

Langford's number problem (CSP lib problem 24)

http://www.csplib.org/prob/prob024/

http://www.dialectrix.com/langford.html

Generalized version:

The problem generalizes to the L(k,n) problem, which is to arrange k sets of numbers 1 to n, so that each appearance of the number m is m numbers on from the last.

For example, the L(3,9) problem is to arrange 3 sets of the numbers 1 to 9 so that the first two 1's and the second two 1's appear one number apart, the first two 2's and the second two 2's appear two numbers apart, etc.

For L(3,n) there is only a solution if n mod 9 = (0,1,8)

Example: L(3,9):

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

Langford problem - generalized: Model

```
int: n; % 1..n: the numbers to place
int: k; % number of occurrences of each number
array[1..k*n] of var 1..n: sol;   % solution
array[1..k*n] of var 1..k*n: pos; % positions

solve satisfy;

constraint
  all_different(pos) /\
  forall(i in 1..n) (
    let {
      % temporary decision variable: the possible index
      var 1..k*n - ((k-1)*i): j;
    } in
    forall(c in 0..k-1) (
      sol[j+(i*c)+c] = i /\
      pos[(i-1)*k+c+1] = j+(i*c)+c
    )
  )
  /\ global_cardinality(sol, [i | i in 1..n], [k  | i in 1..n])
  /\ sol[1] < sol[k*n];
```

Hidato grid puzzle
Temporary decision variables

Hidato grid puzzle

- http://www.hidato.com/
- Given a grid of Rows x Cols with some pre-filled numbers, including 1 and Rows*Cols (first and last).
- Place all numbers 1..Rows*Cols such that adjacent numbers touch each other horizontally, vertically, or diagonally.

Hidato: Problem instance (0s are the unknowns)

```
% http://www.hidato.com/ Problem 188 (Genius)
r = 12;
puzzle = array2d(1..r, 1..c,
 [
     0,   0,134,   2,   4,   0,   0,   0,   0,   0,   0,   0,
   136,   0,   0,   1,   0,   5,   6,  10,115,106,   0,   0,
   139,   0,   0,124,   0,122,117,   0,   0,107,   0,   0,
     0,131,126,   0,123,   0,   0,  12,   0,   0,   0,103,
     0,   0,144,   0,   0,   0,   0,   0,  14,   0,  99,101,
     0,   0,129,   0,  23,  21,   0,  16,  65,  97,  96,   0,
    30,  29,  25,   0,   0,  19,   0,   0,   0,  66,  94,   0,
    32,   0,   0,  27,  57,  59,  60,   0,   0,   0,   0,  92,
     0,  40,  42,   0,  56,  58,   0,   0,  72,   0,   0,   0,
     0,  39,   0,   0,   0,   0,  78,  73,  71,  85,  69,   0,
    35,   0,   0,  46,  53,   0,   0,   0,  80,  84,   0,   0,
    36,   0,  45,   0,   0,  52,  51,   0,   0,   0,   0,  88,
 ]);
```

Hidato: Model

```
% ...
constraint
  % all distinct integers from 1..r*c
  all_different(x) /\

  % place the hints
  forall(i in 1..r, j in 1..c) (
      if puzzle[i,j] > 0 then x[i,j] = puzzle[i,j] endif
  ) /\

  % identify all k's (1..r*c)
  forall(k in 1..r*c-1) (
      let {
          % temporary decision variables
          var 1..r: i, var 1..c: j, var {-1,0,1}: a, var {-1,0,1}: b
      } in
      k = x[i, j] /\ % fix this k
      i+a >= 1 /\ j+b >=  1 /\ i+a <= r /\ j+b <= c % inside the grid
      /\ not(a = 0 /\ b = 0) /\
      k + 1 = x[i+a, j+b] % find the next k
  )
```

Hidato: Solution

137	135	134	**2**	**4**	**7**	**8**	9	114	113	112	111
136	138	133	**1**	**3**	**5**	**6**	10	115	106	105	110
139	132	125	124	121	122	117	116	11	107	109	104
140	131	126	127	123	120	118	12	13	108	102	103
141	130	**144**	128	22	119	17	15	14	98	99	101
142	**143**	129	24	23	21	18	16	65	97	96	100
30	29	25	26	20	19	61	62	64	66	94	95
32	31	28	27	57	59	60	75	63	67	93	92
33	40	42	55	56	58	76	74	72	70	68	91
34	39	41	43	54	77	78	73	71	85	69	90
35	38	44	46	53	49	50	79	80	84	86	89
36	37	45	47	48	52	51	81	82	83	87	88

Traveling Salesperson Problem (TSP)
Circuit constraint

TSP

- Basic problem description (from Wikipedia):
  """
  Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?
  """
- There are many variants on this problem, but let's keep it simple.

Global constraint circuit

- Given a list of integers (representing the cities), the circuit constraint shows what city (node) should be visited next.
- For 4 cities the circuit

  [2,4,1,3]

  means
  City 1 → City 2
  City 2 → City 4
  City 3 → City 1
  City 4 → City 3
- The constraint assumes that we start at city 1
- The path is thus 1 → 2 → 4 → 3 → 1

  Note: The circuit constraint does not show the path directly.

TSP: Data (distance between the cities)

```
n = 7;
distances = array2d(1..n, 1..n,
[
   0, 4, 8,10, 7,14,15,
   4, 0, 7, 7,10,12, 5,
   8, 7, 0, 4, 6, 8,10,
  10, 7, 4, 0, 2, 5, 8,
   7,10, 6, 2, 0, 6, 7,
  14,12, 8, 5, 6, 0, 5,
  15, 5,10, 8, 7, 5, 0,
]);


% From Ulf Nilsson
% "Transparencies for the course TDDD08 Logic Programming"
```

TSP: The setup

```
int: n; % number of cities

array[1..n, 1..n] of int: distances;       % distance matrix
% domains for d, the distances of the travelled path
int: min_val = min([distances[i,j] | i,j in 1..n where distances[i,j] > 0]);
int: max_val = max([distances[i,j] | i,j in 1..n]);

% decision variabls
array[1..n] of var 1..n: x;                  % the circuit
array[1..n] of var 1..n: p;                  % the path
array[1..n] of var min_val..max_val: d;      % the distances for the path
var int: distance = sum(d);                  % total distance (to be minimized)


solve minimize distance;
```

circuit_path constraint

- Since the circuit constraint does not show the path, let's write a decomposition for converting a circuit to a path.
- circuit_path(circuit, path)
  Converts the information in circuit into a path.
- The path [2,4,3,1] represents the path
  1 → 2 → 4 → 3 → 1
- We always assume that city 1 is visted first (and last).

TSP: The circuit_path(circuit,path) decomposition

```
%
% circuit_path(x,p)
% Ensures that x is a circuit and that p is a path for that circuit
%
predicate circuit_path(array[int] of var int: x,
                       array[int] of var int: p) =
  let {
    int: len = length(x)
  } in
  circuit(x) /\
  all_different(p) /\

  % always starts the path at city 1
  p[1] = x[1] /\ % start at city 1
  p[len] = 1  /\ % back to city 1
  forall(i in 2..len) (
    p[i] = x[p[i-1]]  % connection between city i and the next city
  )
;
```

TSP: Constraints

```
constraint
     circuit_path(x,p)
     /\
     % d[i] is the distance for the ith visited city:
     % the distance between the city i and the next city x[i]
     % (again, the element constraint is used)
     forall(i in 1..n) (
       distances[i,x[i]] = d[i]
     )
;
```

TSP: Solution

```
%    1   2   3   4   5   6   7
x: [2, 7, 1, 3, 4, 5, 6]   % The circuit
p: [2, 7, 6, 5, 4, 3, 1]   % The path
dist: 34


The path is thus:
1 → 2
2 → 7
7 → 6
6 → 5
5 → 4
4 → 3
3 → 1    (back to city 1)
```

Code golfing

Code golfing

From http://codegolf.stackexchange.com/questions/8429/can-you-golf-golf/

You are required to generate a random 18-hole golf course.

Example output:

[3 4 3 5 5 4 4 4 5 3 3 4 4 3 4 5 5 4]

Rules:

- Your program must output a list of hole lengths for exactly 18 holes
- Each hole must have a length of 3, 4 or 5
- The hole lengths must add up to 72 for the entire course
- Your program must be able to produce every possible hole configuration with some non-zero-probability (the probabilities of each configuration need not be equal, but feel free to claim extra kudos if this is the case)

Code golfing

```
% The complete model:
array[1..18]of var 3..5:x;constraint sum(x)=72


Run with
$ minizinc 18_hole_golf.mzn -a -s
x = [3, 3, 3, 3, 3, 3, 3, 3, 3, 5, 5, 5, 5, 5, 5, 5, 5, 5];
----------
x = [3, 3, 4, 3, 3, 3, 3, 4, 3, 3, 5, 5, 5, 5, 5, 5, 5, 5];
----------
x = [3, 3, 4, 3, 4, 3, 3, 3, 3, 3, 5, 5, 5, 5, 5, 5, 5, 5];
----------
x = [3, 3, 4, 3, 3, 3, 3, 3, 3, 4, 5, 5, 5, 5, 5, 5, 5, 5];
----------
x = [3, 3, 5, 3, 3, 3, 3, 3, 3, 3, 5, 5, 5, 5, 5, 5, 5, 5];
----------
...

% Number of solutions: 44152809
% Time                : 23min01.36s
```

Smullyan’s Knights and Knaves reification

Knights and Knaves

- From Raymond Smullyan’s excellent “What is the name of this book?”
- A knight always tells the truth
- A knave always lies
- “Liar paradox”:
  - A knave cannot say “I’m lying” (‘cause it’s true)
  - A knight cannot say “I’m lying” (‘cause it’s false)

Knights and Knaves: #26

- Problem #26:
  B says: A says he is a knave
  C says: B is a knave
  What are B and C?

Knights and Knaves: Problem #26 - model

```
% a knight alway tells the truth
% a knave always lies
enum P = {knight,knave};
var P: A; var P: B; var P: C;

% says(kind of person, what the person say: a boolean)
predicate says(var P: kind, var bool: says) =
   (kind = knight <-> says = true )
   /\
   (kind = knave  <-> says = false )
;

solve satisfy;
constraint
    % B: A says he is a knave
    says(B, says(A, A = knave))
    /\
    % C: B is a knave
    says(C, B = knave)
;
```

Knights and Knaves: Problem #26 - solution

Problem #26:
B: A says he is a knave
C: B is a knave

There are two solutions:
p: [knave, knave, knight]
p: [knight, knave, knight]

Which means that
A is unknown (either a knave or knight)
B is a knave (lying)
C is a knight (telling the truth)

Manual reasoning:
* B is lying since it's impossible that A says he's a knave
  → B is a knave
* And since B is lying (is a knave)
  then C is telling the truth
  → C is a knight.

Knights and Knaves: Alternative definition using *V and Λ*

```
Instead of <-> (and /\) we can use /\ (and \/).

predicate says(var P: kind, var bool: says) =
    (kind = knight /\ says = true )
    \/
    (kind = knave  /\ says = false )
;
```

N-queens problem
different encodings

N-queens problem

- Place N queens on a NxN chess board such that no queens attack each other.
- Here we see some different encodings:
  - simple version
  - using all_different
  - using a 0/1 grid
- For the first two, an 1d array is used representing the N rows.

N-queens problem (n=8)

```
6 4 7 1 3 5 2 8

. . . . . Q . .      row 1, col 6
. . . Q . . . .      row 2  col 4
. . . . . . Q .      row 3  col 7
Q . . . . . . .      row 4  col 1
. . Q . . . . .      row 5  col 3
. . . . Q . . .      row 6  col 5
. Q . . . . . .      row 7  col 2
. . . . . . . Q      row 8  col 8
```

Different encodings

- A problem can often be modeled in different ways using different views of representations, etc.
- The best/good model might depend on the strengths of the used solver.
- For SAT/MIP solvers a model using 0/1 (boolean) variables can be quite fast, but not always

N-queens: Simple model

```
int: n;

array [1..n] of var 1..n: q;
constraint
  forall (i in 1..n, j in i+1..n) (
    q[i]     != q[j]     /\ % different rows
    q[i] + i != q[j] + j /\ % different / diagonals
    q[i] - i != q[j] - j    % different \ diagonals
  );

solve satisfy;
```

N-queens: Using all_different

```
int: n;

array [1..n] of var 1..n: q;
constraint
  % Rows are different
  all_different(q) /\

  % "/" diagonals are different
  all_different([q[i]+i | i in 1..n]) /\

  % "\" diagonals are different
  all_different([q[i]-i | i in 1..n])
;

solve satisfy;
```

N-queens: 0/1 variables on a NxN grid

```
int: n;
array[1..n,1..n] of var 0..1: q;
var int: obj = sum(i,j in 1..n) (q[i,j]);
constraint
   % one queen per row
   forall(i in 1..n) ( sum(j in 1..n) (x[i,j]) = 1) /\

   % one queen per column
   forall(j in 1..n) ( sum(i in 1..n) (x[i,j]) = 1) /\

   % at most one queen can be placed in each "/"-diagonal
   forall(k in 2-n..n-2) (
       sum(i,j in 1..n where i-j == k) (x[i,j]) <= 1
   ) /\
   % at most one queen can be placed in each "\"-diagonal
   forall(k in 3..n+n-1) (
       sum(i,j in 1..n where i+j == k) (x[i,j]) <= 1
   )
  /\ obj = n;
```

N-queens: Number of solutions

N	Number of solutions
0	1
1	1
2	0
3	0
4	2
5	10
6	4
7	40
8	**92**
9	352
10	724
11	2680
12	14200
13	73712
14	365596
15	2279184

[1,1,0,0,2,10,4,40,92,352,724,2680,14200,73712,365596,2279184]

Number of solutions: OEIS

- Online Encyclopedia of Integer Sequences: https://oeis.org/

- https://oeis.org/A000170
  """
  A000170 Number of ways of placing n nonattacking queens on an n X n board.
  1, 1, 0, 0, 2, 10, 4, 40, 92, 352, 724, 2680, 14200, 73712, 365596, 2279184, 14772512, 95815104, 666090624, 4968057848, 39029188884, 314666222712, 2691008701644, 24233937684440, 227514171973736, 2207893435808352, 22317699616364044, 234907967154122528
  """

Magic sequence: Is there a pattern in the solutions?

```
  N  Solution
 -------------------------------------------
  4: [ 1,2,1,0]
  5: [ 2,1,2,0,0]
  6: no solution
  7: [ 3,2,1,1,0,0,0]
  8: [ 4,2,1,0,1,0,0,0]
  9: [ 5,2,1,0,0,1,0,0,0]
 10: [ 6,2,1,0,0,0,1,0,0,0]
 11: [ 7,2,1,0,0,0,0,1,0,0,0]
 12: [ 8,2,1,0,0,0,0,0,1,0,0,0]
 13: [ 9,2,1,0,0,0,0,0,0,1,0,0,0]
 14: [10,2,1,0,0,0,0,0,0,0,1,0,0,0]
 15: [11,2,1,0,0,0,0,0,0,0,0,1,0,0,0]
 16: [12,2,1,0,0,0,0,0,0,0,0,0,1,0,0,0]
 17: [13,2,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0]
 18: [14,2,1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0]
 19: [15,2,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0]
```

Magic sequence: Is there a pattern? (Yes, at least for N>=7)

```
 N  Solution
 -------------------------------------------
 4: [ 1,2,1,0]
 5: [ 2,1,2,0,0]
 6: no solution
 7: [ 3,2,1,1,0,0,0]
 8: [ 4,2,1,0,1,0,0,0]
 9: [ 5,2,1,0,0,1,0,0,0]
10: [ 6,2,1,0,0,0,1,0,0,0]
11: [ 7,2,1,0,0,0,0,1,0,0,0]
12: [ 8,2,1,0,0,0,0,0,1,0,0,0]
13: [ 9,2,1,0,0,0,0,0,0,1,0,0,0]
14: [10,2,1,0,0,0,0,0,0,0,1,0,0,0]
15: [11,2,1,0,0,0,0,0,0,0,0,1,0,0,0]
16: [12,2,1,0,0,0,0,0,0,0,0,0,1,0,0,0]
17: [13,2,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0]
18: [14,2,1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0]
19: [15,2,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0]
```

Magic sequence: Non CP algorithm in Python3, for n >= 7

```
def magic_sequence(n):
    """
    This works for n>=7.
    """
    if n < 7:
        return []
    else:
        s = [0]*n
        s[0] = n-4
        s[1] = 2
        s[2] = 1
        s[n-4] = 1
        return s

print(magic_sequence(10))   # [6, 2, 1, 0, 0, 0, 1, 0, 0, 0]
```

For n=10 000 this program takes 0.04s. (Gecode takes 31.78s.)

Sometimes CP is not the fastest approach; but it's often great for exploring problems.
The Python program was written after I played with the CP model.

Thirty bottles

Thirty bottles

- From Alcuins, via Paul Vaderlind "Klassisk Nöjesmatematik" (“Classical recreational mathematics”), 2003, page 38.
- A man died and left 30 bottles to his 3 sons. 10 bottles was filled with oil, 10 was half full with oil, and 10 was empty. The wish of the man was that all the sons should get the **same amount of bottles and the same amount of oil**. How to distribute bottles and oil in a fair way if it's not allowed to pour oil from one bottle to another.
- How many solutions are there?

30 bottles: Parameters and decision variables

```
% parameters
int: n = 3;                                  % number of bottle types

% how filled are the bottle types (the ratio)
%     [filled, half filled, empty] = [1,1/2,0]
array[1..n] of int: t = [2,1,0];             % converted to integers

int: b = [10,10,10];                         % number of bottles of each type
int: num_sons = 3;                           % number of sons

% derived parameters
int: tot_oil = sum([t[i]*b[i] | i in 1..n]); % total amount of oil
int: tot_bottles = sum(b);                   % total number of bottles

% decision variables
% How many bottles of each type should be distributed to each son
array[1..num_sons,1..n] of var 0..tot_oil: x;
```

30 bottles: Model

```
constraint
  forall(s in 1..num_sons) (
    % total number of bottles per son (row)
    % (convert to multiplication)
    num_sons*sum(x[s,..] ) = tot_bottles /\

    % total amount of oil per son
    num_sons*sum([x[s,j]*t[j] | j in 1..n]) = tot_oil

    /\ % symmetry breaking (lexicographic order of rows)
    if s < num_sons then
      lex_lesseq(x[s,..],x[s+1,..])
    endif
  )
  /\
  % check the the number of bottles of each type
  % i.e. the columns in the matrix.
  forall(j in 1..n) (
    sum(x[..,j]) = b[j]
  );
```

30 bottles: First solution

```
[3, 4, 3]  = 3+4+3 = 10 bottles     First son
[3, 4, 3]  = 3+3+3 = 10 bottles     Second son
[4, 2, 4]  = 4+2+4 = 10 bottles     Third son
---------
10 10 10  sums of columns (=number of bottles of each type)


How many liter oil per son?

We must use the original ratios [1,1/2,0],
not those in the model ([2,1,0]).

Son 1: [3, 4, 3]
3*1 + 4/2 + 3*0 = 3 + 2 + 0 = 5 liter oil

Son 2: [3, 4, 3]
3*1 + 4/2 + 3*0 = 3 + 2 + 0 = 5 liter oil

Son 3: [4, 2, 4]
4*1 + 2/2 + 4*0 = 4 + 1 + 0 = 5 liter oil
```

30 bottles: All solutions (i.e. 5 solutions)

```
[3, 4, 3]
[3, 4, 3]
[4, 2, 4]
----------
[2, 6, 2]
[4, 2, 4]
[4, 2, 4]
----------
[1, 8, 1]
[4, 2, 4]
[5, 0, 5]
----------
[0, 10, 0]
[5, 0, 5]
[5, 0, 5]
----------
[2, 6, 2]
[3, 4, 3]
[5, 0, 5]
----------
==========
```

Without symmetry breaking there are 21 solutions.

Thirty bottles, variant

- Paul Vaderlind "Klassisk Nöjesmatematik", 2003, page 40 (Problem 15)
- How to distribute 5 full, 8 half-full, and 11 empty bottles of wine between three persons if each person get the same number of bottles and the same amout of wine. Find all solutions.

30 bottles: Problem 15, parameters

```
% parameters
int: n = 3;                                    % number of bottle types
% how filled are the bottle types
array[1..n] of int: t = [2,1,0];
int: b = [5,8,11];                           % number of bottles of each type
int: num_sons = 3;                             % number of sons

% … as before
```

30 bottles: Problem 15, solutions

```
[1, 4, 3]
[2, 2, 4]
[2, 2, 4]
----------
[1, 4, 3]
[1, 4, 3]
[3, 0, 5]
----------
[0, 6, 2]
[2, 2, 4]
[3, 0, 5]
----------
==========
```

The Paris Marathon puzzle
A logic puzzle

The Paris Marathon puzzle

Dominique, Ignace, Naren, Olivier, Philippe, and Pascal have arrived as the first six at the Paris marathon. Reconstruct their arrival order from the following information:

a) Olivier has not arrived last

b) Dominique, Pascal and Ignace have arrived before Naren and Olivier

c) Dominique who was third last year has improved this year.

d) Philippe is among the first four.

e) Ignace has arrived neither in second nor third position.

f) Pascal has beaten Naren by three positions.

g) Neither Ignace nor Dominique are on the fourth position.

(From Guéret & Sevaux: “Programmation linéaire”, 2000)

The Paris Marathon: Parameters and decision variables

```
include "globals.mzn";
% Parameters
int: n = 6;
array[1..n] of string: runners_s =
      ["Dominique", "Ignace", "Naren", "Olivier", "Philippe", "Pascal"];


% Decision variables
var 1..n: Dominique;
var 1..n: Ignace;
var 1..n: Naren;
var 1..n: Olivier;
var 1..n: Philippe;
var 1..n: Pascal;
array[1..n] of var 1..n: runners =
               [Dominique, Ignace, Naren, Olivier, Philippe, Pascal];

solve satisfy;
```

The Paris Marathon: Constraints 1/2

```
constraint
  all_different(runners) /\

  % a: Olivier not last
  Olivier    != n /\

  % b: Dominique, Pascal and Ignace before Naren and Olivier
  Dominique  < Naren /\
  Dominique  < Olivier /\
  Pascal     < Naren /\
  Pascal     < Olivier /\
  Ignace     < Naren /\
  Ignace     < Olivier /\

  % c: Dominique better than third
  Dominique  < 3 /\

  % d: Philippe is among the first four
  Philippe   <= 4  /\

  % cont...
```

The Paris Marathon: Constraints 2/2

```
% (cont)

  % e: Ignace neither second nor third
  Ignace     != 2 /\
  Ignace     != 3 /\

  % f: Pascal three places earlier than Naren
  Pascal + 3 = Naren /\

  % g: Neither Ignace nor Dominique on fourth position
  Ignace     != 4 /\
  Dominique  != 4
;
```

The Paris Marathon: Output section

```
output
[
  "Runners: \(runners)\n"
]
++
[
 if fix(runners[j]) = i then "Place \(i): \(runners_s[j])\n" endif
  | i in 1..n, j in 1..n
];
```

The Paris Marathon: Solution

```
[2, 1, 6, 5, 4, 3]
Place 1: "Ignace"
Place 2: "Dominique"
Place 3: "Pascal"
Place 4: "Philippe"
Place 5: "Olivier"
Place 6: "Naren"

%          Dominique Ignace Naren Olivier Philippe Pascal
Runners: [2,        1,     6,    5,      4,       3]

The clues again:
a) Olivier has not arrived last
b) Dominique, Pascal and Ignace have arrived before Naren
   and Olivier
c) Dominique who was third last year has improved this year.
d) Philippe is among the first four.
e) Ignace has arrived neither in second nor third position.
f) Pascal has beaten Naren by three positions.
g) Neither Ignace nor Dominique are on the fourth position.
```

Labeled dice
Global cardinality count

Labeled dice

(From Humphrey Dudley via Jim Orlin)

- My daughter Jenn bough a puzzle book, and showed me a cute puzzle. There are 13 words as follows: BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, SWAG, VISA, WISH.
- There are 24 different letters that appear in the 13 words. The question is: can one assign the 24 letters to 4 different cubes so that the four letters of each word appears on different cubes. (There is one letter from each word on each cube.)

Labeled dice: The setup

```
int: n = 4;
int: num_words = 13;

enum letters = {A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,Y};
array[1..num_words, 1..n] of int: words = array2d(1..num_words, 1..n,
  [
   B,U,O,Y,   C,A,V,E,    C,E,L,T,   F,L,U,B,    F,O,R,K,
   H,E,M,P,   J,U,D,Y,    J,U,N,K,   L,I,M,N,    Q,U,I,P,
   S,W,A,G,   V,I,S,A,    W,I,S,H
  ]);

% Decision variables: At which die should a letter be placed?
array[1..24] of var 1..n: dice;

solve satisfy;
```

Labeled dice: Constraints and symmetry breaking

```
constraint
  % the letters in a word must be on a different die
  forall(i in 1..num_words) (
    alldifferent([dice[words[i,j]] | j in 1..n])
  )
  /\
  % there must be exactly 6 letters of each die
  global_cardinality(dice, [i | i in 1..n], [6 | i in 1..n]);

% There are 24 different solutions.
% This symmetry breaking yields just 1 solution.
constraint
  dice[ 1] < dice[ 7] /\ % first letter of die 1 vs die 2
  dice[ 7] < dice[13] /\ % die 2 vs die 3
  dice[13] < dice[19]    % die 3 vs die 4
;
```

Labeled dice: Output

```
{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, Y}
dice:
[1, 2, 4, 2, 2, 4, 2, 1, 2, 1, 2, 1, 3, 4, 1, 4, 1, 3, 4, 3, 3, 3, 3, 4]

die: 1: A H J L O Q
die: 2: B D E G I K
die: 3: M R T U V W
die: 4: C F N P S Y

BUOY
CAVE
CELT
FLUB
FORK
HEMP
JUDY
JUNK
LIMN
QUIP
SWAG
VISA
WISH
```

Five 5-letter words that share no common letter

Five words share no letters

- Find five five-letter words that has no letter in common. Get all possible solutions.
- From Matt Parker (Stand-Up Maths)
- https://www.youtube.com/watch?v=_-AfhLQfb6w
- Preprocessing:
  - sort words (10175 from a word list) and collect anagrams
  - convert words to list of integers
  - write as a MiniZinc datafile (.dzn)

  (→ 5977 anagrams)

Five letter words: Data file (converted from a word list)

```
num_words=5977;
% The anagrams
words = array2d(1..num_words,1..5,[
    1,2,3,5,8,  % abceh: 'bache' and 'beach'
    1,2,3,5,9,  % abcei: 'ceibal'
    1,2,3,5,12, % abcel: 'cable' and 'caleb'
    1,2,3,5,16, % abcep: 'becap'
    1,2,3,5,18, % abcer: 'acerb','brace','caber',and 'cabre'
     …
]);

% Words covered by an anagram
words_s = [
"[bache,beach]", % Words for the first anagram
"[ceiba]",
"[cable,caleb]",
"[becap]",
"[acerb,brace,caber,cabre]",
…
]);
```

Five letter words: The model

```
% ...
array[1..num_words,1..n] of int: words; % anagrams as integer arrays
array[1..num_words] of string: words_s; % covered words as strings

array[1..n] of var 1..num_words: x; % The words (index)
array[1..n,1..n] of var 1..26: y; % The individual characters

constraint
  % The words (anagrams) are distinct and ordered
  all_different(x) /\
  increasing(x) /\  % symmetry breaking

  % The letters are distinct
  all_different(y) /\

  % Connect the selected word and the characters
  forall(i,j in 1..n) (
    y[i,j] = words[x[i],j]
  )
;
output [ "\([words_s[fix(x[i])] | i in 1..n])\n"];
```

Five letter words: Solution

```
["[japyx]", "[bortz]", "[chivw]", "[dunks]", "[flegm]"]
-----------
["[knyaz]", "[bumps]", "[chivw]", "[fldxt]", "[jorge]"]
-----------
["[japyx]", "[bilks]", "[fconv]", "[zhmud]", "[grewt]"]
-----------
["[ampyx]", "[bortz]", "[chivw]", "[fjeld]", "[gunks]"]
-----------
["[japyx]", "[bongs]", "[chivw]", "[fremd]", "[klutz]"]
-----------
["[swack,wacks]", "[vibex]", "[fjord]", "[glyph]", "[muntz]"]
-----------
["[gravy]", "[bumph]", "[jocks]", "[fldxt]", "[winze,wizen]"]
-----------
...
["[whank]", "[gumby]", "[crips,crisp,scrip]", "[fldxt]", "[vejoz]"]
-----------
...
```

Five letter words: labeling

- In earlier CP talks, I talked quite much on search strategies (a.k.a. labeling):
  first_fail, most_constrained, indomain_split, etc.
- Nowadays, it’s easier to use “solve satisfy” and just testing different solvers, e.g.
  - OR-tools CP-SAT (with/without -f + -p <n_threads>)
  - Chuffed (with/without -f + -p)
  - PicatSAT
  - Gecode (with/without -f + -p)
  - HiGHs, Geas, etc

Five letter words: labeling

But.

- For this problem, the fastest configuration I’ve found is Gecode using
  - first_fail, indomain_reverse_split
  - p 22 (number of threads)
- Time to show all solutions: 16.8s (dedicated algos can be quite faster, <1s)

Just forgotten

Just forgotten

- Joe was furious when he forgot one of his bank account numbers.

  He remembered that it had all the digits 0 to 9 in some order, so he tried the following four sets without success:

  9 4 6 2 1 5 7 8 3 0

  8 6 0 4 3 9 1 2 5 7

  1 6 4 0 2 9 7 8 5 3

  6 8 2 4 3 1 9 0 7 5

- When Joe finally remembered his account number, he realised that **in each set just four of the digits were in their correct position** and that, if one knew that, it was possible to work out his account number. What was it? (Enigma puzzle #1517)

Just forgotten: The model

```
int: rows = 4;
int: cols = 10;
array[1..rows, 1..cols] of 0..9: a;
array[1..cols] of var 0..9: x;
solve satisfy;

constraint all_different(x);

% In each set exactly 4 digits are in the correct position
constraint
   forall(r in 1..rows) (
     sum([x[c] = a[r,c] | c in 1..cols]) = 4
   )
;

a = array2d(1..rows, 1..cols,
            [9,4,6,2,1,5,7,8,3,0,
             8,6,0,4,3,9,1,2,5,7,
             1,6,4,0,2,9,7,8,5,3,
             6,8,2,4,3,1,9,0,7,5]);
```

Just forgotten: Solution

```
x: [9, 6, 2, 4, 3, 1, 7, 8, 5, 0]
----------
==========
```

“Just four of the digits were in their correct position.”

9 4 6 2 1 5 **7** **8** 3 **0**
8 **6** 0 **4** **3** 9 **1** 2 **5** 7
1 **6** 4 0 2 9 7 **8** **5** 3
6 8 **2** **4** **3** 1 9 0 7 5

Just forgotten
Generating instances

Generating instances

- Use CP to generate instances.
- Add extra constraints to ensure all requirements
- To guarantee a **unique solution** of the instance, we have to check the number of solutions. This is not supported in MiniZinc.
- Here's an approach using MiniZinc-Python https://minizinc-python.readthedocs.io/en/latest/

Generating instances

Two steps:

- 10 Generate a candidate matrix A
- 20 If more than one solution (X) -> goto 10
- 30 Print A and X

Generating instances: The model

```
int: rows = 4;
int: cols = 10;
array[1..rows, 1..cols] of var 0..9: a;
array[1..cols] of var 0..9: x;

solve :: int_search(array1d(a) ++ x,first_fail,indomain_random)
      :: restart_linear(1000) % faster
      satisfy;

constraint
   all_different(x) /\
   forall(r in 1..rows) (
     all_different(a[r,..]) /\
     sum([x[c] = a[r,c] | c in 1..cols]) = 4
   )
   /\ % Each element in x[c] must have some match in a[..,c]
   forall(c in 1..cols) (
      sum([x[c] = a[r,c] | r in 1..rows]) >= 1
   );
```

Generating instances: MiniZinc-Python program

```
from minizinc import Instance, Model, Solver
import random

def gen(a=None):
    just_forgotten = Model("./just_forgotten_generate.mzn")
    sol = Solver.lookup("gecode")
    instance = Instance(sol, just_forgotten)
    # Step 1: Generate a candidate matrix a
    if a == None:
        result = instance.solve(random_seed=random.randint(0,1000000))
        return(result["a"])
    # Check the number of solutions
    instance["a"] = a
    result = instance.solve(nr_solutions=2) # we want only one solution
    num_sols = len(result)
    if num_sols == 1:
        return True, result[0,"x"]
    else:
        return False, ""
```

Generating instances: MiniZinc-Python

```
g = 0
while True:
    g += 1
    print("\ngeneration:",g)
    a = gen()
    ret,x = gen(a)
    if ret == True:
        % Output in .dzn format
        print("a = array2d(1..rows, 1..cols,[")
        for i in range(4):
            for j in range(10):
                print(a[i][j],end=", ")
            print()
        print("]);")
        print("% x:",x)
        break

print("generations:", g)
```

Generating instances: Output (2 different runs)

```
a = array2d(1..rows, 1..cols,[
9, 1, 6, 7, 8, 0, 3, 5, 2, 4,
1, 6, 0, 9, 3, 7, 2, 5, 8, 4,
7, 9, 2, 3, 8, 0, 6, 4, 1, 5,
9, 6, 1, 4, 3, 8, 0, 5, 2, 7,
]);
% x: [9, 6, 2, 3, 8, 7, 0, 5, 1, 4]
Generations: 1


###
a = array2d(1..rows, 1..cols,[
5, 2, 1, 4, 9, 6, 7, 3, 8, 0,
3, 0, 8, 1, 6, 4, 7, 5, 2, 9,
2, 8, 3, 0, 9, 1, 4, 5, 7, 6,
1, 6, 9, 4, 5, 2, 7, 3, 8, 0,
]);
% x: [2, 6, 3, 1, 9, 4, 7, 5, 8, 0]
generations: 1
```

Generating instances

- Some extra constraints are required to make the problem instance harder/easier, neater etc.
- Here’s one generated instance with three 7s in a column

5	2	1	4	**9**	6	**7**	3	**8**	**0**
3	0	8	**1**	6	**4**	**7**	**5**	2	9
2	8	**3**	0	**9**	1	4	**5**	7	6
1	**6**	9	4	5	2	**7**	3	**8**	**0**
						^			
						|			

- We want to ensure that there are at most 2 duplicate values, i.e. **at least 3 distinct values**
- Use a global constraint to count the distinct values: `nvalue(array)`

Generating instances: The MiniZinc model, adding nvalue/1

```
constraint
    % ...
    /\
    forall(c in 1..cols) (
        sum([x[c] = a[r,c] | r in 1..rows]) >= 1
        /\
        % at least 3 different values
        nvalue(a[..,c]) >= 3
    );

%%% Example output
% {0,5,6,8,3,4,9,2,7,1}
% {5,0,3,2,8,6,9,7,4,1}
% {1,2,7,9,3,4,5,8,0,6}
% {7,0,1,2,5,9,4,8,6,3}
% x = [5,0,1,2,3,4,9,8,7,6]
```

Generating instances: Picat

- Picat is a multi-paradigm programming language http://picat-lang.org/
- Logic programming: a large subset of Prolog (unification, non-determinism, etc)
- Constraints: CP, SAT, MIP, SMT
- Imperative: for-loop, while loop, reassignments, list/array comprehensions
- Functions
- Tabling (memoization)

Generating instances: Picat (the model)

```
just_forgotten(A,Xs) =>
   N = 10, M = 4,
   A = new_array(M,N), A :: 0..9, % decision variables
   Xs = new_list(10), Xs :: 0..9,

   foreach(I in 1..M)
     all_different(A[I])
   end,
   all_different(Xs),
   foreach(I in 1..M)
     sum([Xs[J] #= A[I,J] : J in 1..N]) #= 4
   end,
   foreach(J in 1..N)
     sum([Xs[J] #= A[I,J] : I in 1..M]) #>= 1,
     nvalue(C,[A[I,J] : I in 1..M]), C #>= 3
   end,

   Vars = Xs ++ A.vars,
   solve($[ff,split,limit(2)],Vars). % generate at most 2 solutions
```

Generating instances: Picat (caller program)

```
import cp. % or sat, mip, smt.
main =>
    _ = random2(),
    % Get a candidate for the A rows
    just_forgotten(A,_),

    % Check if unique solution
    All = find_all(Xs,just_forgotten(A,Xs)),
    if All.len == 1 then
       % Print the solution
       foreach(Row in A)
         println(Row)
       end,
       printf("% %w\n",All[1]),
    else
        % if not a unique solution: backtrack
        fail
    end,
    nl.
```

Generating instances: Picat output

{1,0,8,**2**,6,**4**,**5**,7,**9**,3}
{1,2,**7**,3,9,**4**,**5**,**0**,8,6}
{3,6,0,**2**,**8**,5,7,4,**9**,**1**}
{**6**,**3**,8,7,9,2,4,**0**,5,**1**}
X = [6,3,7,2,8,4,5,0,9,1]

%%%%%%
{**6**,5,**7**,8,**9**,1,0,2,4,**3**}
{7,3,8,2,**9**,6,**1**,**4**,**5**,0}
{**6**,9,8,7,4,2,**1**,0,**5**,**3**}
{9,**2**,1,**0**,6,**8**,5,**4**,3,7}
x = [6,2,7,0,9,8,1,4,5,3]

Generating instances: Specific solution

```
just_forgotten(A,Xs) =>
  % ...
  % We want this as a solution
  Xs = [5,0,1,2,3,4,9,8,7,6],
  % ...
```

```
%%%%%%% Solution
{0,5,6,8,3,4,9,2,7,1}
{5,0,3,2,8,6,9,7,4,1}
{1,2,7,9,3,4,5,8,0,6}
{7,0,1,2,5,9,4,8,6,3}
x = [5,0,1,2,3,4,9,8,7,6]
```

Sicherman Dice

Sicherman Dice

http://en.wikipedia.org/wiki/Sicherman_dice

"""

Sicherman dice are the only pair of 6-sided dice which are not normal dice, bear only positive integers, and have the same probability distribution for the sum as normal dice.

“””

Sicherman Dice: Model

```
include "globals.mzn";
int: n = 6;
int: m = 10; % max value

% standard distribution
array[2..12] of int: standard_dist = array1d(2..12, [1,2,3,4,5,6,5,4,3,2,1]);

% the two dice
array[1..n] of var 1..m: d1;
array[1..n] of var 1..m: d2;

constraint
   forall(k in 2..12) (
     standard_dist[k] = sum(i,j in 1..n) ( d1[i]+d2[j] == k))
   )
   % symmetry breaking
   /\ increasing(d1)
   /\ increasing(d2)
   /\ lex_lesseq(x1, x2)
;
```

Sicherman Dice: Solution

```
% The Sicherman Dice
x1: [1, 2, 2, 3, 3, 4]
x2: [1, 3, 4, 5, 6, 8]

% Plain dice
x1: [1, 2, 3, 4, 5, 6]
x2: [1, 2, 3, 4, 5, 6]
```

Sicherman Dice: Allowing 0 as a value

```
% ...
array[1..n] of var 0..m: d1; % instead of 1..m
array[1..n] of var 0..m: d2;

% …
```

Sicherman Dice: Allowing 0 as a value

```
x1: [0, 1, 1, 2, 2, 3]
x2: [2, 4, 5, 6, 7, 9]
----------
x1: [0, 1, 2, 3, 4, 5]
x2: [2, 3, 4, 5, 6, 7]
----------
x1: [0, 2, 3, 4, 5, 7]
x2: [2, 3, 3, 4, 4, 5]
----------
x1: [1, 2, 2, 3, 3, 4]
x2: [1, 3, 4, 5, 6, 8]
----------
x1: [1, 2, 3, 4, 5, 6]
x2: [1, 2, 3, 4, 5, 6]
----------
==========
```

Move one coin

Move one coin

- From this configuration of coins

  o oo ooo oooo

  move one coin to get the coins in the reverse order, i.e. the number of collected coins are 4, 3, 2, and 1.

  (Scam Nation video, Aug 19, 2021)

Move one coin: Model

```
int: n = 13;
% 0 represents an empty position
array[1..n] of int: goal = [1,0,1,1,0,1,1,1,0,1,1,1,1]; % initial pos
array[1..n] of int: init = [1,1,1,1,0,1,1,1,0,1,1,0,1]; % goal pos

% decision variables
var 1..n: from;
var 1..n: to;

solve satisfy;
constraint
  init[from]= 1 /\ init[to] = 0 /\
  forall(k in 1..n) (
    if k != from /\ k != to then
      goal[k] = init[k]
    endif
  );

output [
        "Move the coin in position \(from) to empty position \(to)\n",
];
```

Move one coin: Solution

```
Move the coin in position 12 to empty position 2
----------
==========



1234567890123    positions
-------------
o oo ooo oooo    init
           |
           |     position 12
  _________v
 |
 v               position 2
oooo ooo oo o    goal
```

A Round of Golf
Logic puzzle
Element constraint

Element constraint

- CP's version of indexing an array/matrix
- In MiniZinc, this is stated as

  z = x[y]

- x: an array of integers or decision variables
- y: integer/enum or decision variable
- z: integer/enum or decision variable
- In other CP systems this is called element(y,x,z) etc

A Round of Golf (I)

(Dell Favorite Logic Problems, Summer 2000)

Jack and three other golf club workers got together on their day off to play a round of eighteen holes of golf.

Afterward, all four, including Mr. Green, went to the clubhouse to total their scorecards. Each man works at a different job (one is a short-order cook), and each shot a different score in the game. No one scored below 70 or above 85 strokes.
(cont)

A Round of Golf (II)

From the clues below, can you discover each man's full name, job and golf score?

1. Bill, who is not the maintenance man, plays golf often and had the lowest score of the foursome.
2. Mr. Clubb, who isn't Paul, hit several balls into the woods and scored ten strokes more than the pro-shop clerk.
3. In some order, Frank and the caddy scored four and seven more strokes than Mr. Sands.
4. Mr. Carter thought his score of 78 was one of his better games, even though Frank's score was lower.
5. None of the four scored exactly 81 strokes.

A Round of Golf: Parameters and decision variables

```
include "globals.mzn";
set of int: d = 1..4;
enum first_name = {Jack, Bill, Paul, Frank}; % Fixed values

% decision variables
% Which first name (1..4) is a last name related to?
var d: Green;
var d: Clubb;
var d: Sands;
var d: Carter;
array[d] of var d: last_name = [Green, Clubb, Sands, Carter];

var d: cook;
var d: maintenance_man;
var d: clerk;
var d: caddy;
array[d] of var d: job = [cook, maintenance_man, clerk, caddy];

array[d] of var 70..85: score;
```

A Round of Golf: Constraints (1)

```
Constraint
  % implicit constraints
  all_different(last_name) /\
  all_different(job)       /\
  all_different(score)     /\ % This is stated explicit

  % 1. Bill, who is not the maintenance man, plays golf often and had
  % the lowest score of the foursome.
  Bill != maintenance_man   /\
  score[Bill] < score[Jack] /\  % Bill is a constant
  score[Bill] < score[Paul] /\
  score[Bill] < score[Frank]/\

  % 2. Mr. Clubb, who isn't Paul, hit several balls into the woods and
  %    scored ten strokes more than the pro-shop clerk.
  Clubb != Paul /\
  % Clubb is a decision variable
  score[Clubb] = score[clerk] + 10
;
```

A Round of Golf: Constraints (2)

```
constraint
  % 3. In some order, Frank and the caddy scored four and seven more
  %    strokes than Mr. Sands.
  Frank != caddy /\
  Frank != Sands /\
  caddy != Sands /\
  (
    (score[Frank] = score[Sands] + 4 /\
     score[caddy] = score[Sands] + 7 )
     \/
    (score[Frank] = score[Sands] + 7 /\
     score[caddy] = score[Sands] + 4 )
  )
  /\
  % 4. Mr. Carter thought his score of 78 was one of his better
  % games, even though Frank's score was lower.
  Frank != Carter /\
  score[Carter] = 78 /\
  score[Frank] < score[Carter]
;
```

A Round of Golf: Constraints (3)

```
constraint
  % 5. None of the four scored exactly 81 strokes.
  forall(i in d) (
    score[i] != 81
  )
;
```

A Round of Golf: Solution

```
first_name: {Jack, Bill, Paul, Frank}
last_name : [4, 1, 2, 3]
Job       : [2, 1, 4, 3]
score     : [85, 71, 78, 75]

Jack Clubb maintenance man 85
Bill Sands cook 71
Paul Carter caddy 78
Frank Green clerk 75
----------
==========

For Bill (id 2) we look up the value of 2 in last_name and job.

The lookup string arrays for last_name and job:

last_name_s = ["Green", "Clubb", "Sands", "Carter"];
job_s = ["cook", "maintenance man", "clerk", "caddy"];
```

A Round of Golf: Output section

```
array[d] of string: job_s = ["cook", "maintenance man", "clerk", "caddy"];
array[d] of string: last_name_s = ["Green", "Clubb", "Sands", "Carter"];

output [
  "first_name: \(first_name)\n",
  "last_name : \(last_name)\n",
  "job       : \(job)\n",
  "score     : \(score)\n\n",
]
++
[
 "\(first_name[i]) " ++

 % looking up which last_name[j] has the value i
 [last_name_s[j] | j in r where fix(last_name[j]) = i][1] ++ " " ++

 [job_s[j] | j in r where fix(job[j]) = i][1] ++ " " ++
 "\(score[i])\n"
 | i in r
];
```

Nontransitive dice

Nontransitive dice

http://en.wikipedia.org/wiki/Nontransitive_dice

“””

A set of dice is intransitive (or nontransitive) if it contains three dice, A, B, and C, with the property that A rolls higher than B more than half the time, and B rolls higher than C more than half the time, but it is not true that A rolls higher than C more than half the time.

“””

In short

  A |> B, B |> C, C |> A

where ‘|>’ means ‘rolls higher more than half the time’.

I.e. the relation is not transitive.

Nontransitive dice

- Simple example: Three d4 dice
  A: 1 2 4 5
  B: 1 3 4 4
  C: 3 3 3 4
- 1 2 4 5 : A win 0 + 1 + 2 + 4 = 7 (A > B)
  1 3 4 4 : B win 0 + 2 + 2 + 2 = 6
- 1 3 4 4 : B win 0 + 0 + 3 +3 = 6 (B > C)
  3 3 3 4 : C win 1 + 1 + 1 + 2 = 5
- 3 3 3 4 : C win 2 + 2 + 2 + 2 = 8 (C > A)
  1 2 4 5 : A win 0 + 0 + 3 + 4 = 7

Nontransitive dice: The setup

```
include "globals.mzn";
int: m = 3;         % number of dice
int: n = 4;         % number of sides of each die

int: max_val = 6; % max value of each die

% Decision variables: The dice
array[1..m, 1..n] of var 1..max_val: dice;

%
% The competitions:
%   die 1 vs die 2, die 2 vs die 1
%   die 2 vs die 3, die 3 vs die 2
%   ...
%   die m vs die 1, die 1 vs die m
%
array[0..m-1, 1..2] of var 0..n*n: comp;
```

Nontransitive dice: Constraints

```
constraint
   % Number of wins for [d1 vs d2, d2 vs d1]
   forall(d in 0..m-1) (
      let {
         int: d1 = 1+(d mod m);       % "This" die
         int: d2 = 1+((d + 1) mod m); % "Next" die
      } in
      comp[d,1] = sum(r1, r2 in 1..n) (dice[d1, r1] > dice[d2, r2]) /\
      comp[d,2] = sum(r1, r2 in 1..n) (dice[d2, r1] > dice[d1, r2])
   )
   /\
   % Nontransitivity
   % All dice 1..m-1 must beat the follower, and die m must beat die 1
   forall(d in 0..m-1) (
     comp[d,1] > comp[d,2]
   )
   /\ % Symmetry breaking: order the number of each die
   forall(d in 1..m) (
     increasing([dice[d,i] | i in 1..n])
   )
   /\ lex2(dice) % lexicographic order of the dice
;
```

Nontransitive dice: One solution for three d4

```
dice:
 1   2   4   5     % A
 1   3   4   4     % B
 3   3   3   4     % C
comp:
   7    6          % A > B
   6    5          % B > C
   8    7          % C > A
```

Nontransitive dice: Two solutions for four d6 (m=4, n=6)

```
dice:
 1   2   5   5   5   6
 1   4   4   4   6   6
 2   2   3   5   6   6
 2   2   5   5   5   6
comp:
 17  16
 17  15
 14  13
 13  11
----------
dice:
 1   2   2   6   6   6
 1   5   5   5   5   6
 2   4   4   4   6   6
 3   3   3   5   6   6
comp:
 17  15
 20  14
 17  15
 18  12
```

Nontransitive dice: Six d6, all_different(dice)

```
m=6;
n=6;
max_val=m*n;

constraint all_different(array1d(dice));
% and the same constraints as original model

% One solution of many
dice:
 1 10 11 14 34 35
 2  9 13 16 32 33
 3  5  7 29 31 36
 4 25 26 27 28 30
 6 17 18 19 22 23
 8 12 15 20 21 24
comp:
 19  17
 19  17
 19  17
 30   6
 19  17
 20  16
```

Huey, Dewey, and Louie Reification

Huey, Dewey, and Louie

Huey, Dewey and Louie are being questioned by their uncle. These are the statements they make:

- Huey: Dewey and Louie has equal share in it; if one is quitly, so is the other.
- Dewey: If Huey is guilty, then so am I.
- Louie: Dewey and I are not both quilty.
- Their uncle, knowing that they are cub scouts, realises that they cannot tell a lie. Has he got sufficient information to decide who (if any) are quilty?

(Marriott & Stuckey: “Programming with Constraints”, 1998, page 42)

Huey, Dewey, and Louie: Model

```
% decision variables
% true: is guilty  false: is not guilty
var bool: huey;
var bool: dewey;
var bool: louie;

solve satisfy;

constraint
   % Huey: Dewey and Louie has equal share in it;
   %        if one is quitly, so is the other.
   (dewey <-> louie)

   % Dewey: If Huey is guilty, then so am I.
   /\ (huey -> dewey)

   % Louie: Dewey and I are not both quilty.
   /\ (not (dewey /\ louie));
```

Huey, Dewey, and Louie: Solution

```
[false, false, false]
----------
==========
```

I.e. all three are innocent.

Short history of CP

- 60s-70s: using constraint satisfaction techniques, especially for graphical systems
- 80s: integrated with logic programming (Prolog) to create Constraint Logic Programming (CLP). Much theoretical work on the underlying principles as well as global constraints.
- 90s and onward: CP integrated in other systems (C++, Java, Python, etc)
- 2010s: Integration of CP with SAT and other techniques: Lazy Clause Generation, Hybrid CP-SAT systems.
  MiniZinc is recognized as a de facto standard for comparing constraint solvers. MiniZinc Challenge since 2008.
- 2020s: Still much theoretical work on principles and adding global constraints.

MiniZinc solving steps

Solving a MiniZinc problem is done in two steps:

- 1) First the model (.mzn) + data (.dzn) is converted to a FlatZinc file (.fzn) for the specific solver. This is a flattened version of the model.
- 2) Then the selected FlatZinc solver is called which then solves the problem