Knowledge Representation and Reasoning Introduction to Constraint Programming with MiniZinc

Antoni Ligęza

ligeza@agh.edu.pl

AGH University of Science and Technology
Kraków, Poland

Knowledge Representation and Reasoning
AGH Kraków
2020

Outline

Presentation Outline

Introduction: An Example to Start

Abduction

- Abduction — principal way of problem solving — generation of hypotheses,
- Abduction — hypotheses generation performed with backtracking search,
- Abduction — produces numerous, admissible solutions

Abduction: Logical model

$$\frac{\alpha \Longrightarrow \beta, \beta}{\alpha}$$

$$HYP^{+} \cup HYP^{-} \cup KB \models OBS^{+} \cup OBS^{-}$$

$$HYP^{+} \cup HYP^{-} \cup KB \cup OBS^{+} \cup OBS^{-} \not\models \bot$$

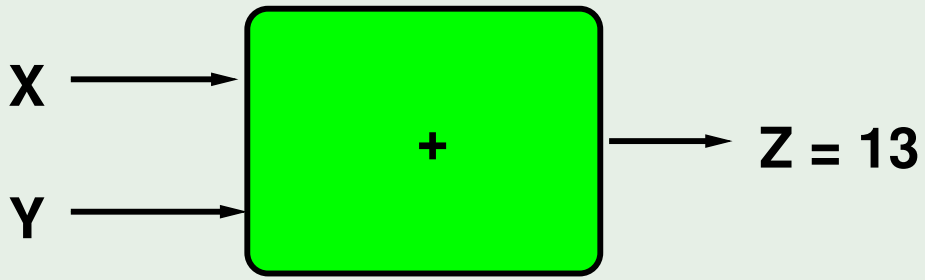
An intuitive example: find explanations for *wet_street*

- $rain \longrightarrow water$
- $sprinkler \longrightarrow water$
- $snow \wedge temperature \longrightarrow water$
- $water \longrightarrow wet\_street$,
- $cleaning \longrightarrow wet\_street$
- $oil \longrightarrow wet\_street$

The role of constraints in abduction

Abductive problem without constraints

- X, Y, Z - variables, $X, Y \in \{0, 1, 2, \ldots, 9\}$, $Z \in \{0, 1, 2, \ldots, 18\}$,
- system: $Z = X + Y$



- Observed: $Z = 13$
- Possible explanations:
  - $(X = 4$ and $Y = 9)$,
  - $(X = 5$ and $Y = 8)$,
  - ... ,
  - $(X = 9$ and $Y = 4)$.
- 6 admissible solutions.

The role of constraints in abduction

Abductive problem with constraints

- X, Y, Z - variables, $X, Y, Z \in \{0, 1, 2, \ldots, 9\}$,

$$Z = X + Y$$

- **Constraint:**

$$Y < X - 3$$

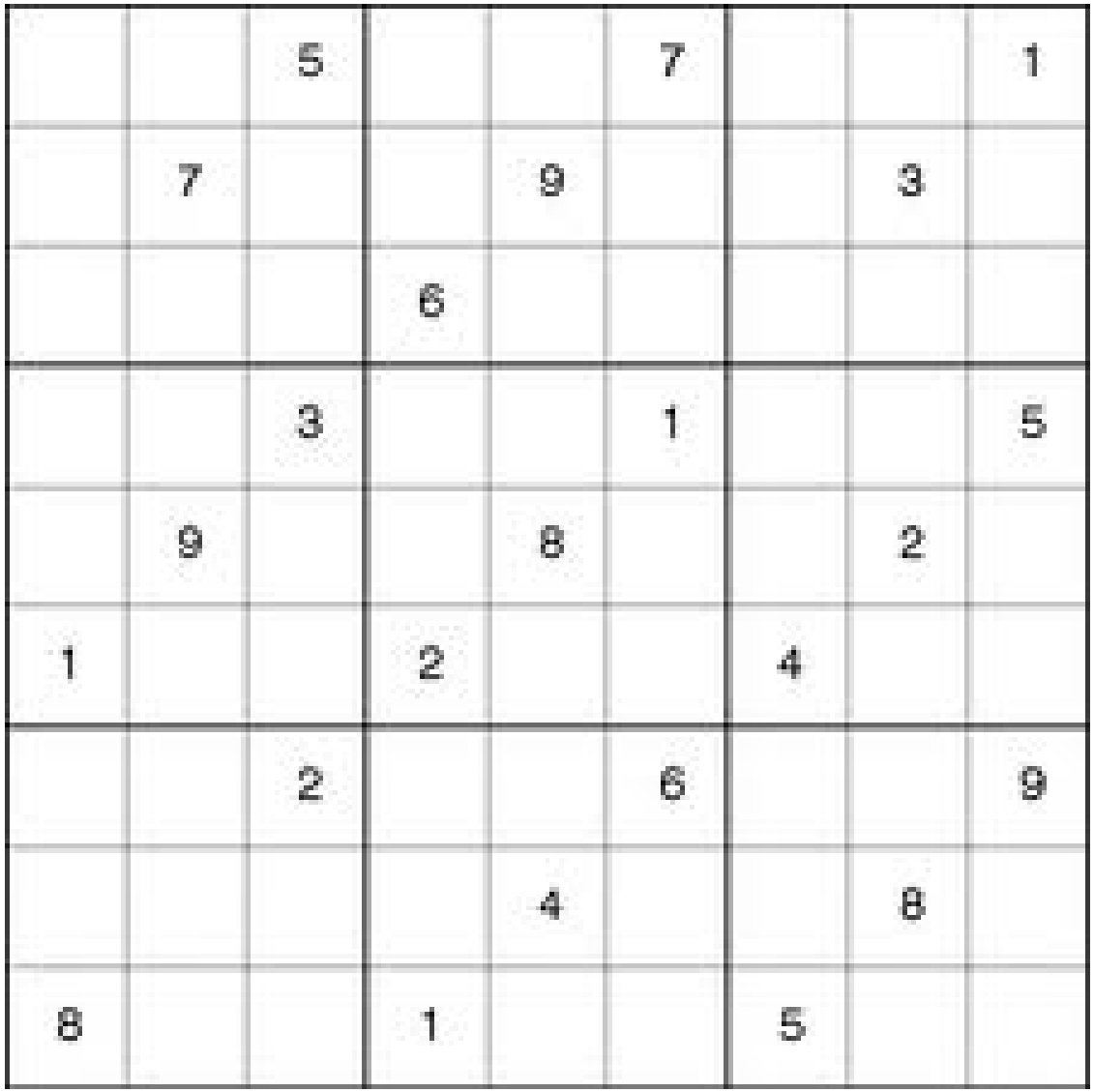
- Observed: $Z = 13$
- Possible explanations: $(X = 9$ and $Y = 4)$,
- 1 admissible solution.

Conclusion

- CONSTRAINTS can refine results of ABDUCTION; less models generated,
- propagation of CONSTRAINTS can reduce computational effort,
- ABDUCTION + CONSTRAINTS = CONSTRUCTIVE ABDUCTION

Presentation Outline

Constraint Satisfaction Problems: Examples



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

Figure : Sudoku: An example Constraint Satisfaction Problem

Constraint Satisfaction Problems: Examples

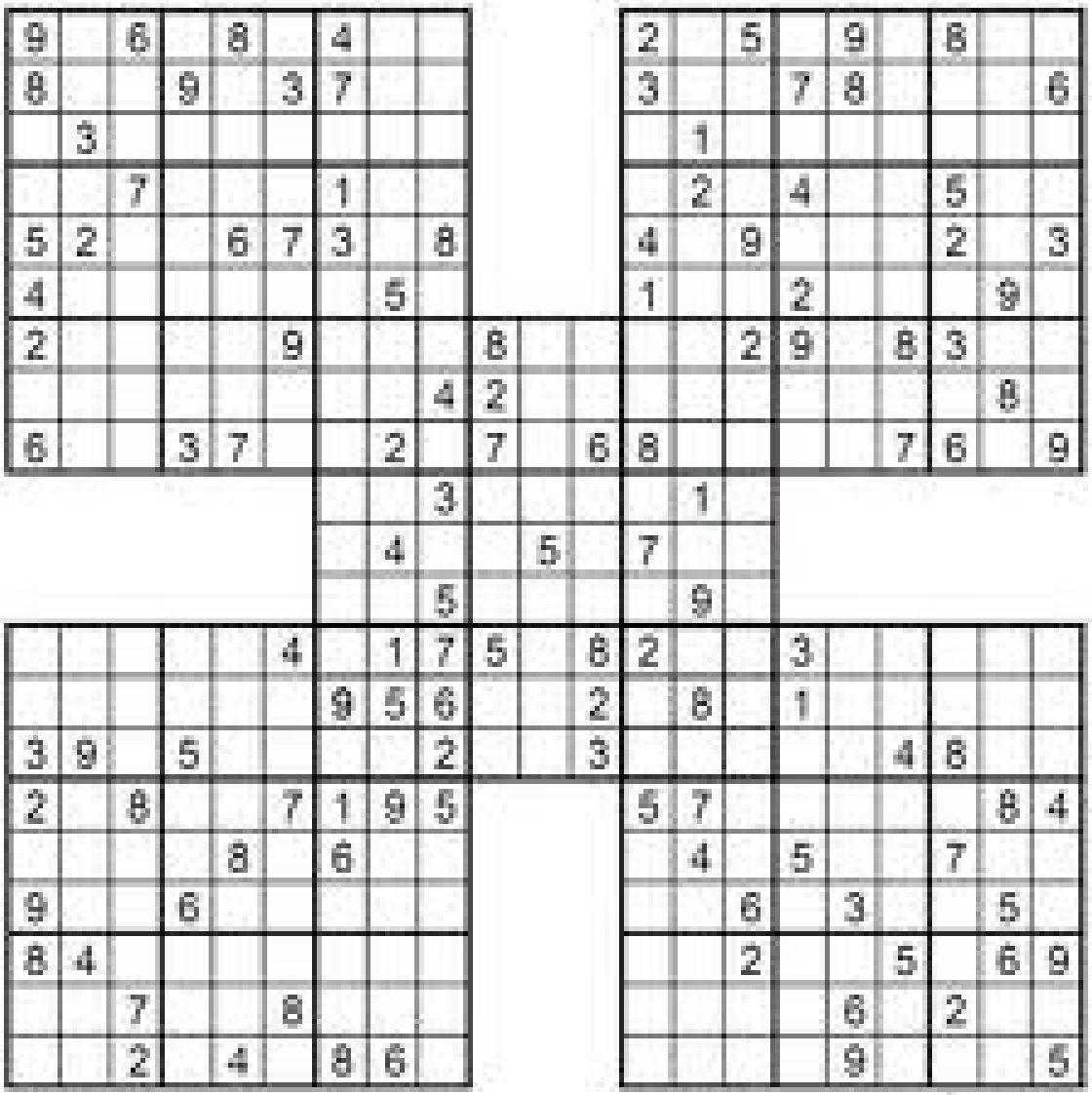


Figure : Sudoku: Yet another example Constraint Satisfaction Problem

Constraint Satisfaction Problems: Examples

Figure : An example Constraint Satisfaction Problem

Constraint Satisfaction Problems: Examples

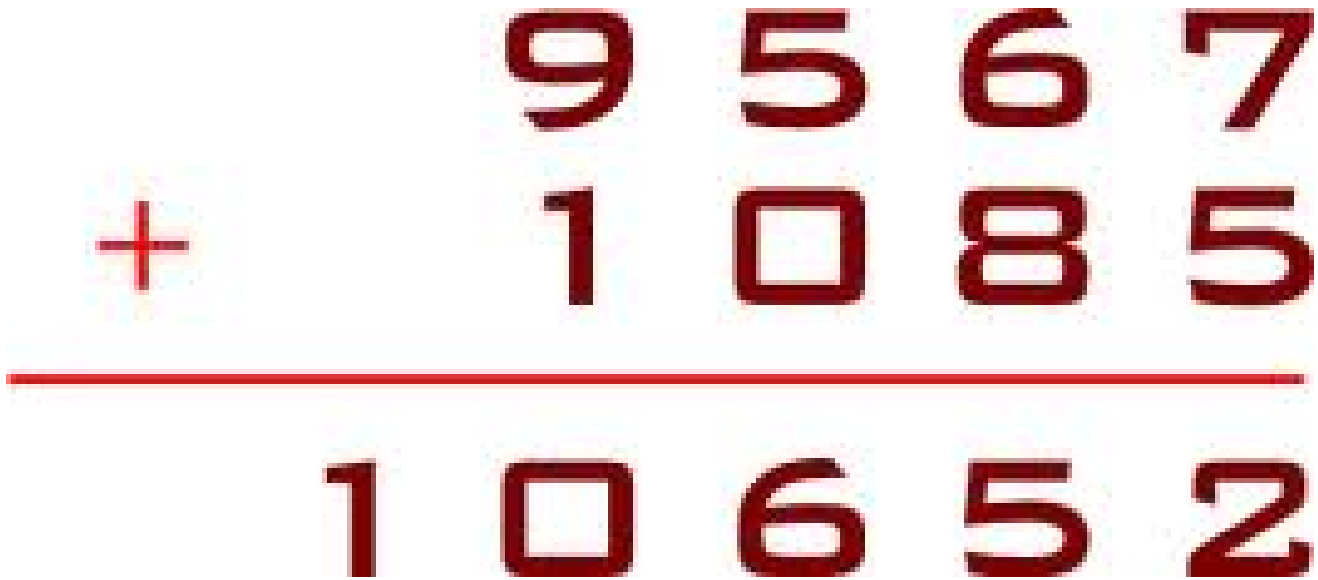


Figure : The unique solution of the example Constraint Satisfaction Problem

Constraint (Logic) Programming

- Declarative Programming + Multi-Purpose Models:
  - simple, transparent statement; practical problem,
  - zero, one, or many solutions,
  - problem: combinatorial explosion.
- Decision factors in CP/CLP:
  - variable — which variable too choose,
  - value — which value to choose,
  - propagation — how to propagate constraints,
  - heuristics — what heuristics can be used.
- CP/CLP basic solution paradigm:
  - select a variable,
  - select a value,
  - propagate constraints,
  - if conflict — backtrack; if unique — return solution; else — loop.

Constraint (Logic) Programming

- Declarative Programming + Multi-Purpose Models:
  - simple, transparent statement; practical problem,
  - zero, one, or many solutions,
  - problem: combinatorial explosion.
- Decision factors in CP/CLP:
  - variable — which variable too choose,
  - value — which value to choose,
  - propagation — how to propagate constraints,
  - heuristics — what heuristics can be used.
- CP/CLP basic solution paradigm:
  - select a variable,
  - select a value,
  - propagate constraints,
  - if conflict — backtrack; if unique — return solution; else — loop.

Constraint (Logic) Programming

- Declarative Programming + Multi-Purpose Models:
  - simple, transparent statement; practical problem,
  - zero, one, or many solutions,
  - problem: combinatorial explosion.
- Decision factors in CP/CLP:
  - variable — which variable too choose,
  - value — which value to choose,
  - propagation — how to propagate constraints,
  - heuristics — what heuristics can be used.
- CP/CLP basic solution paradigm:
  - select a variable,
  - select a value,
  - propagate constraints,
  - if conflict — backtrack; if unique — return solution; else — loop.

Constraint (Logic) Programming

- Declarative Programming + Multi-Purpose Models:
  - simple, transparent statement; practical problem,
  - zero, one, or many solutions,
  - problem: combinatorial explosion.
- Decision factors in CP/CLP:
  - variable — which variable too choose,
  - value — which value to choose,
  - propagation — how to propagate constraints,
  - heuristics — what heuristics can be used.
- CP/CLP basic solution paradigm:
  - select a variable,
  - select a value,
  - propagate constraints,
  - if conflict — backtrack; if unique — return solution; else — loop.

Constraint (Logic) Programming

- Declarative Programming + Multi-Purpose Models:
  - simple, transparent statement; practical problem,
  - zero, one, or many solutions,
  - problem: combinatorial explosion.
- Decision factors in CP/CLP:
  - variable — which variable too choose,
  - value — which value to choose,
  - propagation — how to propagate constraints,
  - heuristics — what heuristics can be used.
- CP/CLP basic solution paradigm:
  - select a variable,
  - select a value,
  - propagate constraints,
  - if conflict — backtrack; if unique — return solution; else — loop.

Constraint (Logic) Programming

- Declarative Programming + Multi-Purpose Models:
  - simple, transparent statement; practical problem,
  - zero, one, or many solutions,
  - problem: combinatorial explosion.
- Decision factors in CP/CLP:
  - variable — which variable too choose,
  - value — which value to choose,
  - propagation — how to propagate constraints,
  - heuristics — what heuristics can be used.
- CP/CLP basic solution paradigm:
  - select a variable,
  - select a value,
  - propagate constraints,
  - if conflict — backtrack; if unique — return solution; else — loop.

Constraint (Logic) Programming

- Declarative Programming + Multi-Purpose Models:
  - simple, transparent statement; practical problem,
  - zero, one, or many solutions,
  - problem: combinatorial explosion.
- Decision factors in CP/CLP:
  - variable — which variable too choose,
  - value — which value to choose,
  - propagation — how to propagate constraints,
  - heuristics — what heuristics can be used.
- CP/CLP basic solution paradigm:
  - select a variable,
  - select a value,
  - propagate constraints,
  - if conflict — backtrack; if unique — return solution; else — loop.

Constraint (Logic) Programming

- Declarative Programming + Multi-Purpose Models:
  - simple, transparent statement; practical problem,
  - zero, one, or many solutions,
  - problem: combinatorial explosion.
- Decision factors in CP/CLP:
  - variable — which variable too choose,
  - value — which value to choose,
  - propagation — how to propagate constraints,
  - heuristics — what heuristics can be used.
- CP/CLP basic solution paradigm:
  - select a variable,
  - select a value,
  - propagate constraints,
  - if conflict — backtrack; if unique — return solution; else — loop.

Constraint (Logic) Programming

- Declarative Programming + Multi-Purpose Models:
  - simple, transparent statement; practical problem,
  - zero, one, or many solutions,
  - problem: combinatorial explosion.
- Decision factors in CP/CLP:
  - variable — which variable too choose,
  - value — which value to choose,
  - propagation — how to propagate constraints,
  - heuristics — what heuristics can be used.
- CP/CLP basic solution paradigm:
  - select a variable,
  - select a value,
  - propagate constraints,
  - if conflict — backtrack; if unique — return solution; else — loop.

Constraint (Logic) Programming

- Declarative Programming + Multi-Purpose Models:
  - simple, transparent statement; practical problem,
  - zero, one, or many solutions,
  - problem: combinatorial explosion.
- Decision factors in CP/CLP:
  - variable — which variable too choose,
  - value — which value to choose,
  - propagation — how to propagate constraints,
  - heuristics — what heuristics can be used.
- CP/CLP basic solution paradigm:
  - select a variable,
  - select a value,
  - propagate constraints,
  - if conflict — backtrack; if unique — return solution; else — loop.

Constraint (Logic) Programming

- Declarative Programming + Multi-Purpose Models:
  - simple, transparent statement; practical problem,
  - zero, one, or many solutions,
  - problem: combinatorial explosion.
- Decision factors in CP/CLP:
  - variable — which variable too choose,
  - value — which value to choose,
  - propagation — how to propagate constraints,
  - heuristics — what heuristics can be used.
- CP/CLP basic solution paradigm:
  - select a variable,
  - select a value,
  - propagate constraints,
  - if conflict — backtrack; if unique — return solution; else — loop.

Constraint (Logic) Programming

- Declarative Programming + Multi-Purpose Models:
  - simple, transparent statement; practical problem,
  - zero, one, or many solutions,
  - problem: combinatorial explosion.
- Decision factors in CP/CLP:
  - variable — which variable too choose,
  - value — which value to choose,
  - propagation — how to propagate constraints,
  - heuristics — what heuristics can be used.
- CP/CLP basic solution paradigm:
  - select a variable,
  - select a value,
  - propagate constraints,
  - if conflict — backtrack; if unique — return solution; else — loop.

Constraint (Logic) Programming

- Declarative Programming + Multi-Purpose Models:
  - simple, transparent statement; practical problem,
  - zero, one, or many solutions,
  - problem: combinatorial explosion.
- Decision factors in CP/CLP:
  - variable — which variable too choose,
  - value — which value to choose,
  - propagation — how to propagate constraints,
  - heuristics — what heuristics can be used.
- CP/CLP basic solution paradigm:
  - select a variable,
  - select a value,
  - propagate constraints,
  - if conflict — backtrack; if unique — return solution; else — loop.

Constraint (Logic) Programming

- Declarative Programming + Multi-Purpose Models:
  - simple, transparent statement; practical problem,
  - zero, one, or many solutions,
  - problem: combinatorial explosion.
- Decision factors in CP/CLP:
  - variable — which variable too choose,
  - value — which value to choose,
  - propagation — how to propagate constraints,
  - heuristics — what heuristics can be used.
- CP/CLP basic solution paradigm:
  - select a variable,
  - select a value,
  - propagate constraints,
  - if conflict — backtrack; if unique — return solution; else — loop.

Presentation Outline

Constraint Satisfaction Problem

CSP statement

- $X = \{X_1, X_2, \ldots, X_k\}$ — a set of variables,
- $D = \{D_1, D_2, \ldots, D_k\}$ — their domains,
- $C = \{(S_i, R_i) : i = 1, 2, \ldots, n\}$ — constraints,
  - S_i — scope — a selection of variables,
  - R_i — relation defined over Cartesian Product of domains appropriate for the scope variables,

CSP solution

A solution to CSP given by (X, D, C) is any assignment of values to variables of X of the form

$$\{X_1 = d_1, X_2 = d_2, \ldots, X_k = d_k\},$$

such that $d_i \in D_i$, and for any constraint in $(S_i, R_i) \in C$, R_i is satisfied by the appropriate projection of the solution vector $(d_1, d_2, \ldots, d_k)$ over variables of S_i.

CP/CSP — Characteristics; CSP vs. OPT

CP vs. OPT

- CSP: first solution counts,
- OPT:best solution counts.

Binary vs. finite domains; SAT

- SAT: binary domains (0 or 1; true or false),
- CSP: finite discrete domains.

CSP: Problems

- large number of variables,
- large domains,
- numerous constraints,
- different types of constraints,
- unpredictable, irregular, hard to trace.

Presentation Outline

Introduction to MiniZinc

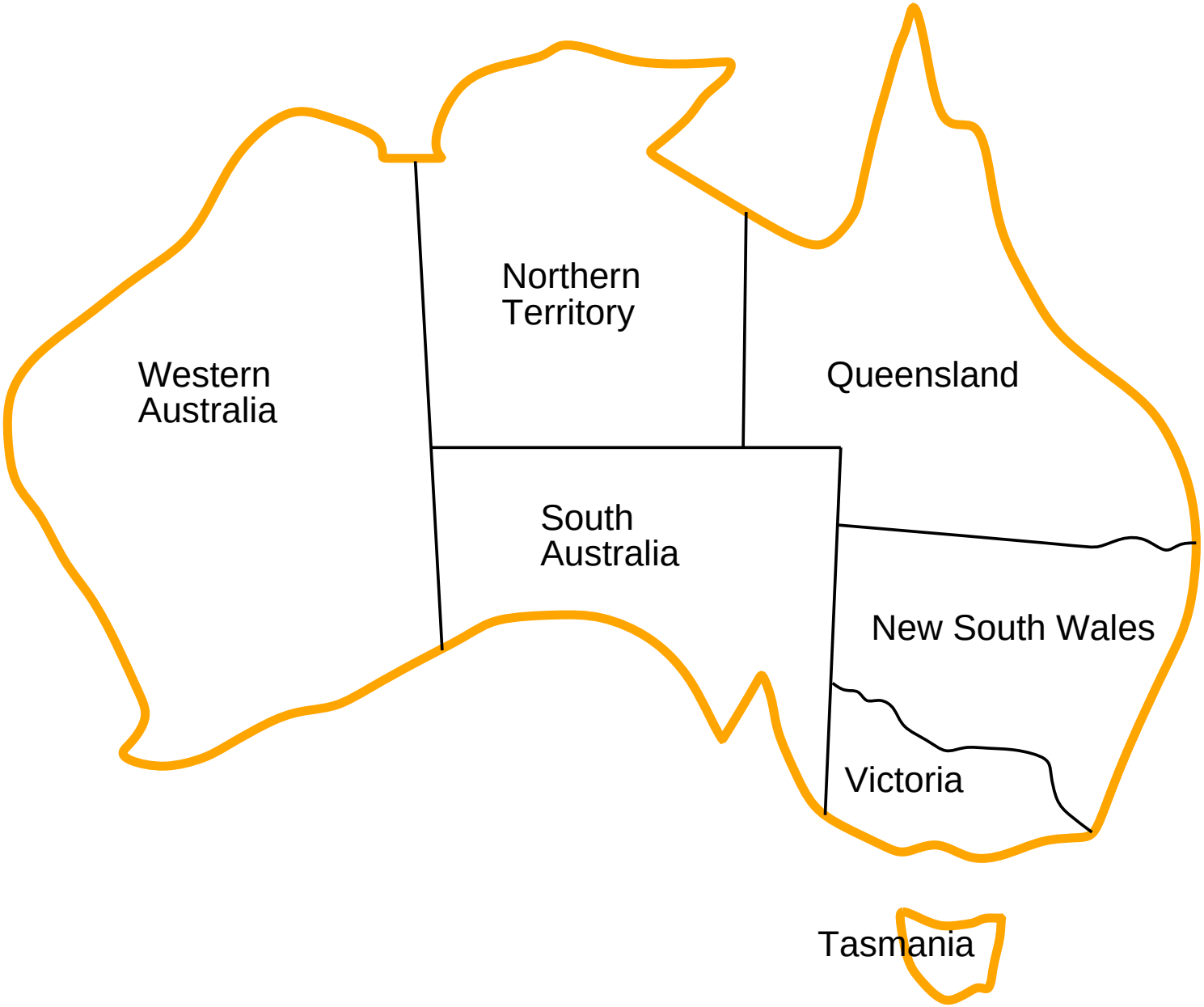
Some important ideas:

- MiniZinc belongs to Declarative Programming Paradigm,
- MiniZinc provides a high=level language for constraint specification,
- the constraints are translated into FlatZinc model,
- the constraints can be processed with several lower-level tools (*backend solvers*; model once, solve everywhere.
- the same MODEL can be processed with several data/goals (*backend solvers*; model once, solve what-you-need.

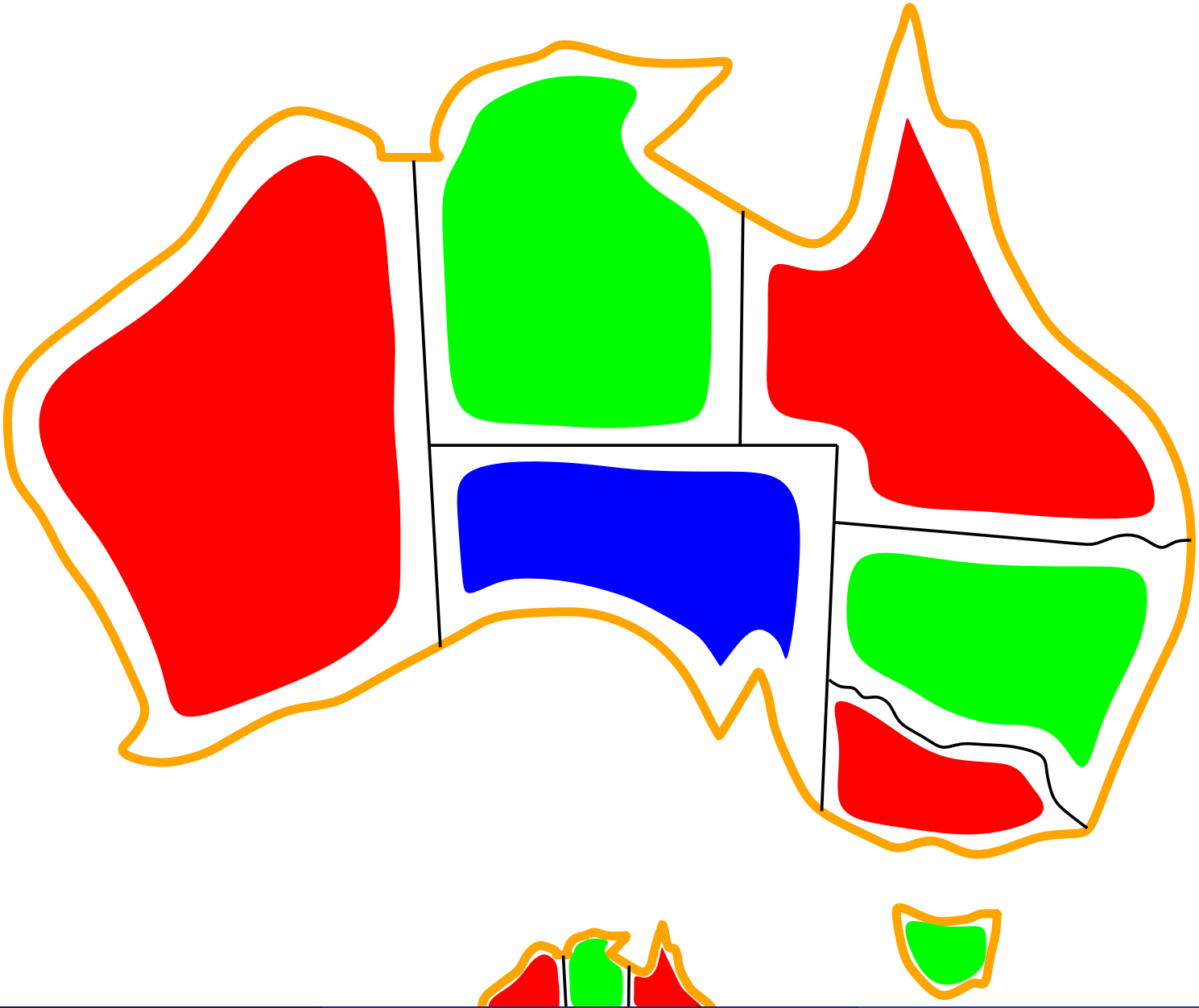
Composition of MiniZinc program:

- **parameters** definition — if any,
- **variables** definition,
- **constraints** definitions,
- **objective function** definition — if any (other than SAT),
- **solve** command and parameters,
- output specification.

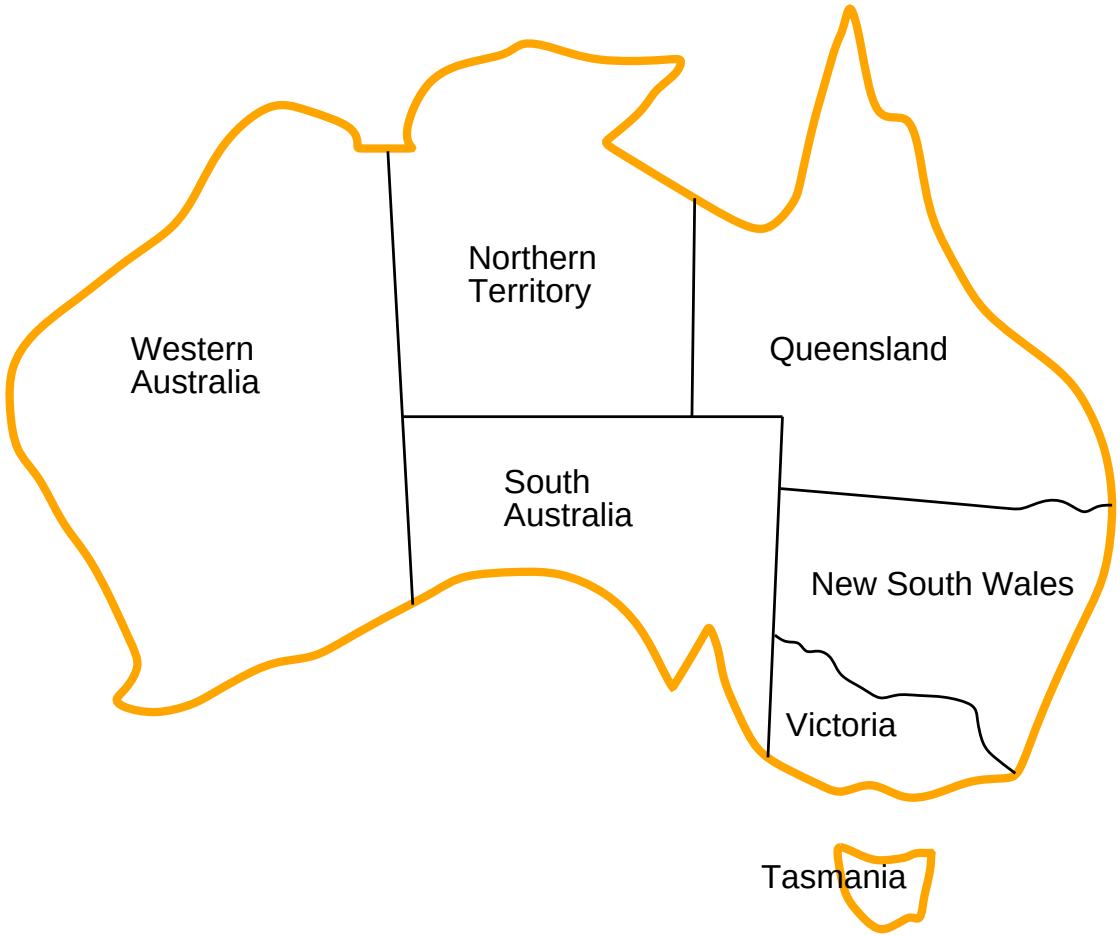
A Map Coloring Problem

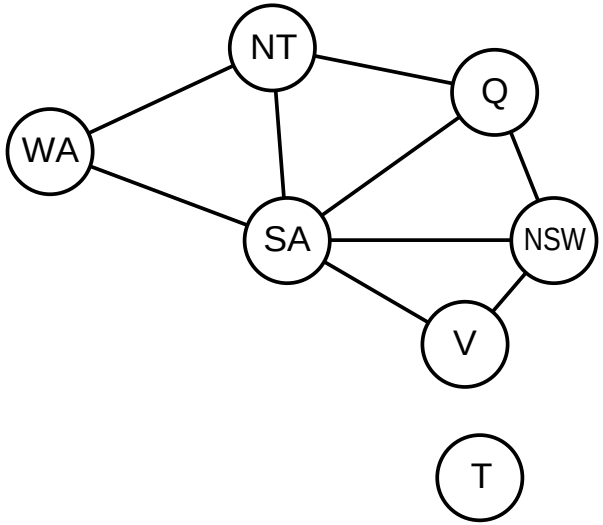


A Map Co-louring Problem Solved



What about Constraints?





A Map Coloring Example I

```
% Colouring Australia using nc colours
int: nc = 3; % a single parameter

% variables
var 1..nc: wa;   var 1..nc: nt;  var 1..nc: sa;   var 1..nc: q;
var 1..nc: nsw;  var 1..nc: v;   var 1..nc: t;

%constraints
constraint wa != nt;
constraint wa != sa;
constraint nt != sa;
constraint nt != q;
constraint sa != q;
constraint sa != nsw;
constraint sa != v;
constraint q != nsw;
constraint nsw != v;
```

A Map Coloring Example II

```
% solution type declaration
solve satisfy;

% output specification
output ["wa=", show(wa), "\t nt=", show(nt),
        "\t sa=", show(sa), "\n", "q=", show(q),
        "\t nsw=", show(nsw), "\t v=", show(v), "\n",
         "t=", show(t),  "\n"];
```

Code Specification: Some Basic Ideas

- a `parameter` — type and value,
- a parameter cannot be changed (but re-specified for a next run),
- parameters can be specified in a separate file, given by hand, or modified **before** compilation,
- supported types: `int`, `float`, `bool`, `string`; **also** `array`, **and** `set` ,
- a `variable` is assigned *domain* (or type),
- variables can be: `bool`, `int`, `float`, `set`,
- arrays of variables are accessible,
- a variable can be *instantiated* with a value of an appropriate type only!
- `constraints` (basic): $=$ ($==$), $>$, $<$, $<=$, $>=$,
- **constraints** are Boolean expressions – *what does this mean?*,
- `solve satisfy;` defines the goal,
- output specification (long strings can be split over lines with the ++ for concatenation).

Presentation Outline

Example: SEND+MORE=MONEY

Figure : An example Constraint Satisfaction Problem

- 8 variables: `S, E, N, D, M, O, R, Y`,
- 10-values in each domain,
- alldifferent(S, E, N, D, M, O, R, Y),
- basic search-space size: 10^8,
- reduced search-space size:
  $9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 = 362880$

A simple CLP code I

```
sendmoremoney(Vars) :-
    Vars = [S,E,N,D,M,O,R,Y],
    Vars ins 0..9,
    S #\= 0,
    M #\= 0,
    all_different(Vars),
                  1000*S + 100*E + 10*N + D
    +             1000*M + 100*O + 10*R + E
    #= 10000*M + 1000*O + 100*N + 10*E + Y.

solve(Vars):- Vars=[S,E,N,D,M,O,R,Y],
              sendmoremoney([S,E,N,D,M,O,R,Y]),
              label(Vars).
```

Pretty good results! I

```
?- time(sendmoremoney(V)).
% 6,758 inferences, 0.00 CPU in 0.00 seconds (0% CPU, Infinite Lips
V = [9, _G11470, _G11473, _G11476, 1, 0, _G11485, _G11488],
_G11470 in 4..7,
all_different([_G11470, _G11473, _G11476, _G11485, _G11488, 0, 1, 9
1000*9+91*_G11470+ -90*_G11473+_G11476+ -9000*1+ -900*0+10*_G11485+
_G11473 in 5..8,
_G11476 in 2..8,
_G11485 in 2..8,
_G11488 in 2..8.

?- time(solve(V)).
% 10,088 inferences, 0.01 CPU in 0.00 seconds (308% CPU, 1008800 Li
V = [9, 5, 6, 7, 1, 0, 8, 2].
```

MiniZinc Coding I

```
include "alldifferent.mzn";

var 1..9: S;
var 0..9: E;
var 0..9: N;
var 0..9: D;
var 1..9: M;
var 0..9: O;
var 0..9: R;
var 0..9: 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;

constraint alldifferent([S,E,N,D,M,O,R,Y]);
```

MiniZinc Coding II

```
solve satisfy;

output ["   ",show(S),show(E),show(N),show(D),"\n",
        "+  ",show(M),show(O),show(R),show(E),"\n",
        "= ",show(M),show(O),show(N),show(E),show(Y),"\n"];
```

Presentation Outline

A Constraint Optimization Problme: SEND+MOST=MONEY I

What about the following Constraint Optimization Problem:

```
  SEND
 +MOST
=======
 MONEY
```

MiniZinc Coding: SEND+MOST=MONEY I

What about the following Constraint Optimization Problem:

```
include "alldifferent.mzn";
var 1..9: S;
var 0..9: E;
var 0..9: N;
var 0..9: D;
var 1..9: M;
var 0..9: O;
var 0..9: T;
var 0..9: Y;

constraint            1000 * S + 100 * E + 10 * N + D
                    + 1000 * M + 100 * O + 10 * S + T
        = 10000 * M + 1000 * O + 100 * N + 10 * E + Y;
constraint alldifferent([S,E,N,D,M,O,T,Y]);

solve maximize 10000 * M + 1000 * O + 100 * N + 10 * E + Y;
```

MiniZinc Coding: SEND+MOST=MONEY I

```
include "alldifferent.mzn";
var 1..9: S;
var 0..9: E;
var 0..9: N;
var 0..9: D;
var 1..9: M;
var 0..9: O;
var 0..9: T;
var 0..9: Y;
var int: sum;
constraint              1000 * S + 100 * E + 10 * N + D
                      + 1000 * M + 100 * O + 10 * S + T
       = 10000 * M + 1000 * O + 100 * N + 10 * E + Y;
constraint alldifferent([S,E,N,D,M,O,T,Y]);
constraint sum = 10000 * M + 1000 * O + 100 * N + 10 * E + Y;
solve maximize sum;
output["The sum of MONEY = ", show(sum)];
```

Integer Programming

Two types of cakes

- two types of cakes: b = banana, c = chocolate; b,c - output variables,
- each of them uses specific amount of limited resources,
- each of them provides some profit,
- the goal is to maximize the profit.

MiniZinc Coding: Banana & Chocolate Cakes I

```
% Baking cakes for the school fete

var 0..100: b; % no. of banana cakes
var 0..100: c; % no. of chocolate cakes

% flour
constraint 250*b + 200*c <= 4000;
% bananas
constraint 2*b  <= 6;
% sugar
constraint 75*b + 150*c <= 2000;
% butter
constraint 100*b + 150*c <= 500;
% cocoa
constraint 75*c <= 500;

% maximize our profit
```

MiniZinc Coding: Banana & Chocolate Cakes II

```
solve maximize 400*b + 450*c;

output ["no. of banana cakes = ", show(b), "\n",
        "no. of chocolate cakes = ", show(c), "\n"];
```

Presentation Outline

Using Data files

In order to change the parameters of the model it is convenient to specify them in a data-file (.dzn).

- a data-file contains a set of pre-declared parameters,
- there can be several files with different data,
- hence, the same model can be re-used in an easy way,
- it is reasonable to check the imported parameters,
- a check is done with the `assert(<condition>,<output>)` predicate (this is called *Defensive Programming*),
- `assert` acts as `Boolean` expression.

Example MiniZinc Code with Data-file I

```
% Baking cakes for the school fete (with data file)

int: flour;  %no. grams of flour available
int: banana; %no. of bananas available
int: sugar;  %no. grams of sugar available
int: butter; %no. grams of butter available
int: cocoa;  %no. grams of cocoa available

constraint assert(flour >= 0,"Invalid datafile: " ++
                  "Amount of flour is non-negative");
constraint assert(banana >= 0,"Invalid datafile: " ++
                  "Amount of banana is non-negative");
constraint assert(sugar >= 0,"Invalid datafile: " ++
                  "Amount of sugar is non-negative");
constraint assert(butter >= 0,"Invalid datafile: " ++
                  "Amount of butter is non-negative");
constraint assert(cocoa >= 0,"Invalid datafile: " ++
```

Example MiniZinc Code with Data-file II

```
                    "Amount of cocoa is non-negative");

var 0..100: b; % no. of banana cakes
var 0..100: c; % no. of chocolate cakes

% flour
constraint 250*b + 200*c <= flour;
% bananas
constraint 2*b  <= banana;
% sugar
constraint 75*b + 150*c <= sugar;
% butter
constraint 100*b + 150*c <= butter;
% cocoa
constraint 75*c <= cocoa;

% maximize our profit
solve maximize 400*b + 450*c;
```

Example MiniZinc Code with Data-file III

```
output ["no. of banana cakes = ", show(b), "\n",
        "no. of chocolate cakes = ", show(c), "\n"];
```

Presentation Outline

Model with Real Numbers

Some ideas concerning Constraint Programming and Optimization with real numbers (floats).

- the properties of the model change drastically,
- it may be necessary to use different solver! (in our case G12 MIP)
- if analytic model is accessible - try it!
- Linear Programming and Simplex may be a solution,
- Mixed Integer-Linear Programming models are hard,
- the same model can be used for answering different questions:

A Loan

P - amount borrowed, I – interest rate, R – rate (4 rates), B – balance

- given I, P, and R (rate), how much is the final balance?
- given I, P, and ensuring B4=0 (0 balance), what should be the rates?
- given I, R, and ensuring B4=0 (0 balance), how much can I borrow (P)?

- the model does not change; only the input parameters.

MiniZinc Coding – a Loan Example I

```
% variables
var float: R;          % quarterly repayment
var float: P;          % principal initially borrowed
var 0.0 .. 10.0: I;    % interest rate

% intermediate variables
var float: B1; % balance after one quarter
var float: B2; % balance after two quarters
var float: B3; % balance after three quarters
var float: B4; % balance owing at end

constraint B1 = P * (1.0 + I) - R;
constraint B2 = B1 * (1.0 + I) - R;
constraint B3 = B2 * (1.0 + I) - R;
constraint B4 = B3 * (1.0 + I) - R;

solve satisfy;
```

MiniZinc Coding – a Loan Example II

```
output [
 "Borrowing ", show_float(0, 2, P),
 " at ", show(I*100.0),
 "% interest, and repaying ", show_float(0, 2, R),
  "\nper quarter for 1 year leaves ",
   show_float(0, 2, B4), " owing\n"
];
```

Presentation Outline

Why Arrays and Sets?

- the number of variables can change with the size of the problem:
  - e.g. number of products (array: quantity of product),
  - e.g. number of rates to be paid (array: amount of rate),
  - e.g. number of components/blocks (array: component value),
- **array** can be of one, two, and many dimensions,
- notation `temp[3,7]`; a variable associated with the position `(3,7)` on a grid,
- one can define a single constraint over an array – all the variables!
- `array[1..5] = [1,3,5,7,11]` – one-dimensional array,
- `array[1..3,1..4] = [|1,2,3|,|4,5,6|,|7,8,9|,|10,11,12|]` – two-dimensional 4×3 array,
- `constraint forall(i in 1..w-1)(temp[i,h] = right);` – –example constraint over a border,
- list comprehension: special form of list specification, e.g. `forall( [a[i] != a[j] | i,j in 1..3 where i < j])`

Laplace: a Visualization

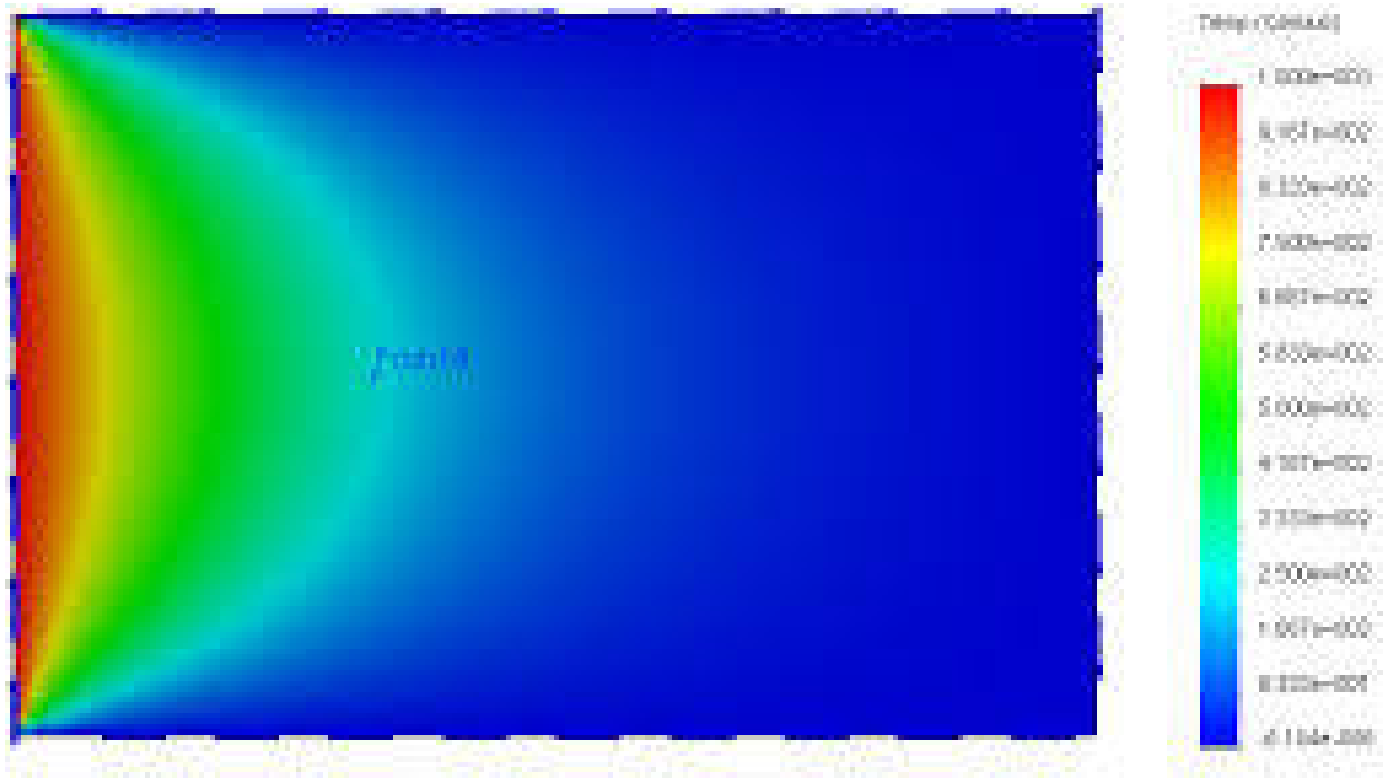


Figure : A Visualization of 2-D Temperature Distribution

MiniZinc Coding: Laplace I

```
int: w = 4;
int: h = 4;

% arraydec
array[0..w,0..h] of var float: t; % temperature at point (i,j)
var float: left;   % left   edge temperature
var float: right;  % right  edge temperature
var float: top;    % top    edge temperature
var float: bottom; % bottom edge temperature

% equation
% Laplace equation: each internal temp.
% is average of its neighbours
constraint forall(i in 1..w-1, j in 1..h-1)(
   4.0*t[i,j] = t[i-1,j] + t[i,j-1] + t[i+1,j] + t[i,j+1]);
% sides
% edge constraints
```

MiniZinc Coding: Laplace II

```
constraint forall(i in 1..w-1)(t[i,0] = left);
constraint forall(i in 1..w-1)(t[i,h] = right);
constraint forall(j in 1..h-1)(t[0,j] = top);
constraint forall(j in 1..h-1)(t[w,j] = bottom);
% corners
% corner constraints
constraint t[0,0]=0.0;
constraint t[0,h]=0.0;
constraint t[w,0]=0.0;
constraint t[w,h]=0.0;
left = 0.0;
right = 0.0;
top = 100.0;
bottom = 0.0;

solve satisfy;

output [ show_float(6, 2, t[i,j]) ++
```

MiniZinc Coding: Laplace III

```
        if j == h then "\n" else " " endif |
        i in 0..w, j in 0..h
];
```

Aggregation Functions

Arithmetic aggregation

- `sum()` – summation of elements on a list,
- `product()` – multiplication of elements on a list,
- `min()` – minimal element from a list,
- `max()` – maximal element on a list.

Logical aggregation (on arrays of Boolean expressions)

- `forall` – logical conjunction of expressions of an array,
- `exists` – logical disjunction of expressions of an array.

Using Sets

Sets are (unordered!) collections of items. Some features and use:

- sets can be of the following types: integers, floats, and Booleans,
- sets can be (and typically are (ordered!) *range expressions* of the form: `FIRST..LAST`,
- `set of int: Products = 1..nproducts` – declaration of a set of int,
- sets of literals are allowed $\{e_1, \ldots, e_k\}$,
- standard operations: `in, subset, superset, union, inter, diff, symdiff; card`.

MiniZinc Coding: Cakes Revisited I

```
% Number of different products
int: nproducts;
set of int: Products = 1..nproducts;
% profit per unit for each product
array[Products] of int: profit;
array[Products] of string: pname;
% Number of resources
int: nresources;
set of int: Resources = 1..nresources;
% amount of each resource available
array[Resources] of int: capacity;
array[Resources] of string: rname;

% units of each resource required to produce 1 unit of product
array[Products, Resources] of int: consumption;
constraint assert(forall (r in Resources, p in Products)
      (consumption[p,r] >= 0), "Error: negative consumption");
```

MiniZinc Coding: Cakes Revisited II

```
% bound on number of Products
int: mproducts = max (p in Products)
                (min (r in Resources where consumption[p,r] > 0)
                     (capacity[r] div consumption[p,r]));

% Variables: how much should we make of each product
array[Products] of var 0..mproducts: produce;
array[Resources] of var 0..max(capacity): used;

% Production cannot use more than the available Resources:
constraint forall (r in Resources) (
 used[r] = sum (p in Products)(consumption[p, r] * produce[p])
      /\ used[r] <= capacity[r]
);

% Maximize profit
solve maximize sum (p in Products) (profit[p]*produce[p]);
```

MiniZinc Coding: Cakes Revisited III

```
output [ show(pname[p]) ++ " = " ++ show(produce[p]) ++ ";\n" |
         p in Products ] ++
       [ show(rname[r]) ++ " = " ++ show(used[r]) ++ ";\n" |
         r in Resources ];
```

Global Constraints I

```
include "alldifferent.mzn";

var 1..9: S;
var 0..9: E;
var 0..9: N;
var 0..9: D;
var 1..9: M;
var 0..9: O;
var 0..9: R;
var 0..9: 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;

constraint alldifferent([S,E,N,D,M,O,R,Y]);
```

Global Constraints II

```
solve satisfy;

output ["   ",show(S),show(E),show(N),show(D),"\n",
        "+  ",show(M),show(O),show(R),show(E),"\n",
        "= ",show(M),show(O),show(N),show(E),show(Y),"\n"];
```

Conditional Expressions

Conditional expressions: *if COND then EXPR1 else EXPR2 endif;*
Example:

```
int:  r = if y != 0 then x div y else 0 endif;
```

Example – Sudoku initialization:

```
constraint forall(i,j in PuzzleRange)
(if start[i,j] > 0 then puzzle[i,j] = start[i,j] else true endif
);
```

MiniZinc Coding: Sudoku I

```
include "alldifferent.mzn";

int: S;
int: N = S * S;
int: digs = ceil(log(10.0,int2float(N))); % digits for output

set of int: PuzzleRange = 1..N;
set of int: SubSquareRange = 1..S;

array[1..N,1..N] of 0..N: start; %% initial board 0 = empty
array[1..N,1..N] of var PuzzleRange: puzzle;

% fill initial board
constraint forall(i,j in PuzzleRange)(
   if start[i,j] > 0 then puzzle[i,j] = start[i,j]
   else true endif );
```

MiniZinc Coding: Sudoku II

```
% All different in rows
constraint forall (i in PuzzleRange) (
   alldifferent( [ puzzle[i,j] | j in PuzzleRange ]) );

% All different in columns.
constraint forall (j in PuzzleRange) (
   alldifferent( [ puzzle[i,j] | i in PuzzleRange ]) );

% All different in sub-squares:
constraint
   forall (a, o in SubSquareRange)(
      alldifferent( [ puzzle[(a-1) *S + a1, (o-1)*S + o1] |
                      a1, o1 in SubSquareRange ] ) );

solve satisfy;

output  [ show_int(digs,puzzle[i,j]) ++ " " ++
          if j mod S == 0 then " " else "" endif ++
```

MiniZinc Coding: Sudoku III

```
if j == N /\ i != N then
    if i mod S == 0 then "\n\n" else "\n" endif
else "" endif
| i,j in PuzzleRange ] ++ ["\n"];
```

Complex Constraints I

- `/\` – concjunction,
- `\/` – disjunction,
- `->` – implication,
- `<-` – only-if (implication to the left),
- `<->` – if-and-only-if (equivalence),
- `not` – negation.

Example:

```
constraint s1 + d1 <= s2 \/ s2 + d2 <= s1;
```

s – start of a task,
d – duration of a task
(execution of tasks cannot overlap on a single machine)
The Job-Shop example:

- a set of job must be completed,
- each job consists of sequential tasks,
- the tasks must be executed in order,
- and on separate machines.

`bool2int` – a function convert Boolean to integer (true $= 1$, false $= 0$).

MiniZinc Coding: A Job-Shop Example I

```
int: jobs;                                          % no of jobs
int: tasks;                                         % no of tasks per job
array [1..jobs,1..tasks] of int: d;                 % task durations
int: total = sum(i in 1..jobs, j in 1..tasks)
             (d[i,j]);                              % total duration
int: digs = ceil(log(10.0,int2float(total)));       % digits for output
array [1..jobs,1..tasks] of var 0..total: s;        % start times
var 0..total: end;                                  % total end time


constraint %% ensure the tasks occur in sequence
    forall(i in 1..jobs) (
        forall(j in 1..tasks-1)
            (s[i,j] + d[i,j] <= s[i,j+1]) /\
        s[i,tasks] + d[i,tasks] <= end
    );
```

MiniZinc Coding: A Job-Shop Example II

```
constraint %% ensure no overlap of tasks
    forall(j in 1..tasks) (
        forall(i,k in 1..jobs where i < k) (
            s[i,j] + d[i,j] <= s[k,j] \/
            s[k,j] + d[k,j] <= s[i,j]
        )
    );

solve minimize end;

output ["end = ", show(end), "\n"] ++
       [ show_int(digs,s[i,j]) ++ " " ++
         if j == tasks then "\n" else "" endif |
         i in 1..jobs, j in 1..tasks ];
```

Example: Stable Marriage

In MiniZinc decision variables can be used for array access.

Stable Marriage

Consider the (old-fashioned) stable marriage problem. We have n (straight) women and n (straight) men. Each man has a ranked list of women and vice versa. We want to find a husband/wife for each women/man so that all marriages are stable in the sense that:

- whenever m prefers another women o to his wife w, o prefers her husband to m, and
- whenever w prefers another man o to her husband m, o prefers his wife to w.

MiniZinc Coding: Stable Marriage I

```
int: n;

set of int: Men = 1..n;
set of int: Women = 1..n;

array[Women, Men] of int: rankWomen;
array[Men, Women] of int: rankMen;

array[Men] of var Women: wife;
array[Women] of var Men: husband;

% assignment
constraint forall (m in Men) (husband[wife[m]]=m);
constraint forall (w in Women) (wife[husband[w]]=w);
% ranking
constraint forall (m in Men, o in Women) (
     rankMen[m,o] < rankMen[m,wife[m]] ->
```

MiniZinc Coding: Stable Marriage II

```
        rankWomen[o,husband[o]] < rankWomen[o,m] );

constraint forall (w in Women, o in Men) (
     rankWomen[w,o] < rankWomen[w,husband[w]] ->
         rankMen[o,wife[o]] < rankMen[o,w] );
solve satisfy;

output ["wives= ", show(wife),"\n", "husbands= ", show(husband), "\
```

MiniZinc Coding I

Presentation Outline

MiniZinc Coding: Symmetry Braking I

```
int: n = 5; % number of variables in the sequence
int: range = 10; % range of variables
int: sum = 20;  % the required sum
array[1..n] of var 1..range: s;

constraint sum (i in 1..n)(s[i]) = sum;

% constraint forall (i in 1..n-1) (s[i] < s[i+1]);

solve satisfy;

output [ "s = ", show(s), ";\n" ] ;
```

Symmetry Braking and Higher-Order Constraints

Magic Series Problem

Given n, find a sequence (a list) of numbers

$$s = [s_0, s_1, \ldots, s_{n-1}]$$

such that:

$$s_i = Number\_of\_occurrences(i)$$

An example is s = [1, 2, 1, 0].

Higher-order constraint: the function `bool2int` takes as its argument an arbitrary boolean expression; the expression is evaluated to true/false and the results is converted to integer.

MiniZinc Coding: Magic Series Example I

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

constraint forall(i in 0..n-1) (
   s[i] = (sum(j in 0..n-1)(bool2int(s[j]=i))));

solve satisfy;

output [ "s = ", show(s), ";\n" ] ;
```

Presentation Outline

A Simple Knapsack Problem I

Set Valued Variables – a decision variable can take set as its value.
Example declaration:

```
var set of Items: knapsack;
```

Note: the var keyword comes before the set declaration indicating that the set itself is the decision variable. This contrasts with an array in which the var keyword qualifies the elements in the array rather than the array itself since the basic structure of the array is fixed, i.e. its index set.

Simple Knapsack Problem

Given a set of items, find optimal packing of a knapsack, so that:

- weight constraint: the total weight of the selected items is still under the knapsack capability,
- max of value: the total value of the selected elements is maximal.

Note: We do not know the final number of selected elements.
Note: No direct iteration over sets is admissible! The code below will result with an error.

A Simple Knapsack Problem II

```
constraint sum (i in knapsack) (weights[i]) <= capacity;

solve maximize sum (i in knapsack) (profits[i]) ;
```

MiniZinc Coding: Knapsack Problem I

```
int: n;
set of int: Items = 1..n;
int: capacity;

array[Items] of int: profits;
array[Items] of int: weights;

% setvar
var set of Items: knapsack;
% capacity
constraint sum (i in Items)
               (bool2int(i in knapsack)*weights[i]) <= capacity;

solve maximize sum (i in Items) (bool2int(i in knapsack)*profits[i]

output [show(knapsack),"\n"];
```

Presentation Outline

Cumulative Use of Resources I

The `cumulative` constraint is used for describing cumulative resource usage. Declaration:

```
cumulative(array[int] of var int: s, array[int] of var int: d,
array[int] of var int: r, var int: b)
```

Requires that a set of tasks given by start times s, durations d, and resource requirements r, never require more than a global resource bound b at any one time.

MiniZinc Coding: The Moving Example I

```
include "cumulative.mzn";

int: n; % number of objects;
set of int: OBJECTS = 1..n;
array[OBJECTS] of int: duration; % duration to move
array[OBJECTS] of int: handlers; % number of handlers required
array[OBJECTS] of int: trolleys; % number of trolleys required

int: available_handlers;
int: available_trolleys;
int: available_time;

array[OBJECTS] of var 0..available_time: start;
var 0..available_time: end;

constraint cumulative(start, duration, handlers, available_handlers
constraint cumulative(start, duration, trolleys, available_trolleys
```

MiniZinc Coding: The Moving Example II

```
constraint forall(o in OBJECTS)(start[o] +duration[o] <= end);

solve minimize end;

output [ "start = ", show(start), "\nend = ", show(end), "\n"];
```

MiniZinc Coding: The Moving Data I

```
n = 8;
% piano, fridge, double bed, single bed, wardrobe, chair, chair, ta

duration = [60, 45, 30, 30, 20, 15, 15, 15];
handlers = [3, 2, 2, 1, 2, 1, 1, 2];
trolleys = [2, 1, 2, 2, 2, 0, 0, 1];

available_time = 180;
available_handlers = 4;
available_trolleys = 3;
```