A First Step into MiniZinc

Peter Stuckey

Overview

- A few simple example models in MiniZinc

MiniZinc

- MiniZinc is a modeling language being developed by NICTA with Univ of Melbourne and Monash University.
- Depending on the kind of model it can be solved with constraint programming or MIP or SAT or SMT techniques.
- It is a subset of the more powerful modeling language Zinc.

First Example: ToyProblem

- The problem:
- A toy manufacturer must determine how many bicycles, B, and tricycles, T, to make in a 40 hr week given that
  - the factory can produce 200 bicycles per hour or 140 tricycles
  - the profit for a bicycle is \$25 and for a tricycle it is \$30
  - no more than 6,000 bicycles and 4,000 tricycles can be sold in a week

$$\text{Maximise } 25B + 30T$$

Subject to

$$(1/200)B + (1/140)T \leq 40 \wedge$$

$$0 \leq B \leq 6000 \wedge 0 \leq T \leq 4000$$

A First MiniZinc Model

```
solve maximize 25*B + 30*T;

constraint 140*B+200*T <= 40*200*140;

var 0..6000: B;
var 0..4000: T;

output  ["B=\(B) T=\(T)\n"];
```

Maximise $25B + 30T$

Subject to

$(1/200)B + (1/140)T \leq 40 \wedge$

$0 \leq B \leq 6000 \wedge 0 \leq T \leq 4000$

A First MiniZinc Model

```
var 0..6000: B;
var 0..4000: T;

constraint 140*B+200*T <= 40*200*140;

solve maximize 25*B + 30*T;

output  ["B=\(B) T=\(T)\n"];
```

Maximise $25B + 30T$

Subject to

$(1/200)B + (1/140)T \leq 40 \wedge$

$0 \leq B \leq 6000 \wedge 0 \leq T \leq 4000$

A First MiniZinc Model

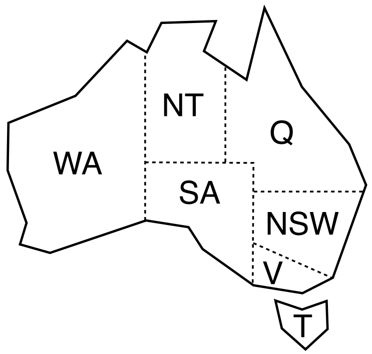
- We can run our MiniZinc model as follows
  `$ minizinc toyproblem.mzn`
- This results in

```
B=6000 T=1400
----------
==========
```

- The line ---------- indicates a solution
- The line ========== indicates no better solution (that this is the best solution)
- MiniZinc models must end in .mzn
- There is also an IDE for MiniZinc

Second Example: AustColor

- Given a map of Australian states and territories
- Color it in so no two adjacent regions are colored the same.



A Second MiniZinc Model

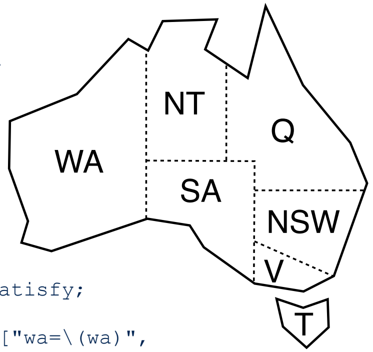
```
% Colouring Australia using 4 colors
int: nc = 4;

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;

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;

solve satisfy;

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



A Second MiniZinc Model

- We can run our MiniZinc model as follows

```
$ minizinc aust_color.mzn
```

- This results in

```
wa=1 nt=3 sa=2
q=1 nsw=3 v=1
t=1
----------
```

- We can change the model to use 2 colors by instead using the line

```
int: nc = 2;
```

- This results in

```
=====UNSATISFIABLE=====
```


Overview

- Two examples models
- Optimization
  - ToyProblem
- Satisfaction
  - AustColor

EOF

MiniZinc Basic Components

Peter Stuckey

Overview

- Basic modeling features in MiniZinc
  - Parameters
  - Decision Variables
  - Types
  - Arithmetic Expressions
  - (Arithmetic) Constraints
  - Structure of a model

Parameters

In MiniZinc there are two kinds of variables:

Parameters-These are like variables in a standard programming language. They must be assigned a value (but only one).

They are declared with a type (or a range/ set).

You can use par but this is optional

The following are logically equivalent

```
int: i=3;
par int: i=3;
int: i;  i=3;
```


Decision Variables

Decision variables-These are like variables in mathematics. They are declared with a type and the var keyword. Their value is computed by a solver so that they satisfy the model.

Typically they are declared using a range or a set rather than a type name

The following are logically equivalent

```
var int: i; constraint i >= 0; constraint i <= 4;
var 0..4: i;
var {0,1,2,3,4}: i;
```

Types

Allowed types for variables are

- Integer `int` or range `1..n` or set of integers
  - `l..u` is integers {l, l+1, l+2, .., u}
- Floating point number `float` or range `1.0 .. f` or set of floats
- Boolean `bool`
- Strings `string` (but these cannot be decision variables)
- Arrays
- Sets

Instantiations

Variables have an instantiation which specifies if they are parameters or decision variables.

The type + instantiation is called the type-inst.

MiniZinc errors are often couched in terms of mismatched type-insts…

Comments

- Comments in MiniZinc files are
  - anything in a line after a `%`
  - anything between `/*` and `*/`
- (Just like in programming) It is valuable to
  - have a header comment describing the model at the top of the file
  - describe each parameter
  - describe each decision variable
  - and describe each constraint

Strings

Strings are provided for output

- An output item has form

  `output` <list of strings>;
- String literals are like those in C:
  - enclosed in `" "`
- They cannot extend across more than one line
- Backslash for special characters \n \t etc
- Built in functions are
  - `show(v)`
  - `\(v)` show v inside a string literal
  - `"house"++"boat"` for string concatenation

Arithmetic Expressions

MiniZinc provides the standard arithmetic operations

- Floats: `*` `/` `+` `-`
- Integers: `*` `div` `mod` `+` `-`

Integer and float literals are like those in C

There is automatic coercion from integers to floats. The builtin `int2float`(intexp) can be used to explicitly coerce them

Builtin arithmetic functions:

`abs, sin, cos, atan, …`

Constraints

- Basic arithmetic constraints are built using the arithmetic relational operators are
  `==` `!=` `>` `<` `>=` `<=`
- Constraints in MiniZinc are written in the form
  `constraint` <constraint-expression>

Basic Structure of a Model

A MiniZinc model is a sequence of items

The order of items does not matter

The kinds of items are

- An inclusion item
  `include` <filename (which is a string literal)>;
- An output item
  `output` <list of string expressions>;
- A variable declaration
- A variable assignment
- A constraint
  `constraint` <Boolean expression>;

Basic Structure of a Model

The kinds of items (cont.)

- A solve item (a model must have exactly one of these)
  `solve satisfy;`
  `solve maximize` <arith. expression>;
  `solve minimize` <arith. expression>;
- Predicate, function and test items
- Annotation items

- Identifiers in MiniZinc start with a letter followed by other letters, underscores or digits
- In addition, the underscore `_' is the name for an anonymous decision variable

EOF

Models and Instances

Peter Stuckey

Models and Instances

- A model is a formal description of a class of (in our case) optimization problems
- An instance if one particular optimization problem
- To make a model into an instance
  - just add DATA
- In MiniZinc this is managed with
  - data files (.dzn)

Example: AustColorD Model

```
% Colouring Australia using nc colors
int: nc;

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;

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;
solve satisfy;
```

Adding Data

- Data specifying 4 colors in file nc4.dzn

```
nc = 4;
```

- Then running

```
$ minizinc aust_colord.mzn nc4.dzn
```

- Results in the same as before

- Alternatively

```
$ minizinc aust_colord.mzn -D”nc = 4;”
```

- has the same effect

Example: SimpleLoan

- Consider a simple loan made over a year
- We borrow an amount *P* the principal
- Every quarter we make a repayment *R*
- The quarterly interest rate is *I*
- At the end of the *i*-th quarter we owe a balance *Bi*

SimpleLoan Model

```
% variables
var float: R; % quarterly repayment
var float: P; % principal initially borrowed
var 0.0 .. 2.0: 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 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;

output …
```


First Instance

- I want to borrow $1000 at 4% repaying $260, what do I owe at the end
- Data file defining the instance
  - loan1.dzn

```
I = 0.04;
P = 1000.0;
R = 260.0;
```

- Running the model with the data file

```
$ minizinc loan.mzn loan1.dzn
```

- Result

```
Borrowing 1000.00 at 4.0% interest, and repaying 260.00
per quarter for 1 year leaves 65.78 owing
```

Second Instance

- I want to borrow $1000 at 4% and owe nothing at the end, how much do I repay?
- Data file defining the instance
  - loan2.dzn

```
I = 0.04;
P = 1000.0;
B4 = 0.0;
```

- Running the model with the data file

```
$ minizinc loan.mzn loan2.dzn
```

- Result

```
Borrowing 1000.00 at 4.0% interest, and repaying 275.69
per quarter for 1 year leaves 0.00 owing
```


Data files

- MiniZinc data files must end in .dzn
- Data files only contain assignment items
  - usually only for parameters
- Any parameters not assigned in the model must be assigned in the data file
- You can add multiple data files assigning different parameters/variables, e.g.

```
$ minizinc model.mzn d1.dzn d2.dzn
```


Overview

- Models and instances are not the same
- Model + Data = Instance

EOF

Modeling Objects

Peter Stuckey

Modeling Objects

- Often combinatorial problems involve a set of objects which we need to make decisions about
- How can we efficiently represent these objects and their different characteristics?
- Arrays (indexed by object IDs)

Smuggler's Knapsack

A smuggler with a knapsack with capacity 18, needs to choose items to smuggle to maximize profit

Object	Profit	Size
Whiskey	29	8
Perfume	19	5
Cigarettes	8	3

$$\begin{array}{ll} \text{maximize} & 29W + 19P + 8C \\ \text{subject to} & 8W + 5P + 3C \leq 18 \end{array}$$

Smuggler's Knapsack

- But what if the data is different:
  - Capacity 200

Object	Profit	Size
Gold	1300	90
Silver	1000	72
Copper	520	43
Bronze	480	40
Tin	325	33

- We want a model to be reused with different sized data!

Knapsack Model

```
int: n; % number of objects
set of int: OBJ = 1..n;
int: capacity;
array[OBJ] of int: profit;
array[OBJ] of int: size;

array[OBJ] of var int: x; % how

constraint forall(i in OBJ)(x[i] >= 0);
constraint sum(i in OBJ)(size[i] * x[i]) <= capacity;
solve maximize sum(i in OBJ)(profit[i] * x[i]);

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

set declarations

array declarations

array lookups

forall expressions

sum expressions

New MiniZinc Features

- Range:
  - l..u is integers {l, l+1, l+2, .., u}
  - Can also be a float range e.g. 1.5 .. 2.745
- Sets
  - `set of` *type*
- Arrays of parameters and variables
  - `array`[*range*] `of` *variable declaration*
- Array lookup
  - *array-name*[*index-exp*]
- Generator expressions
  - `forall`(*i* `in` *range*)(*bool-expression*)
  - `sum`(*i* `in` *range*)(*expression*)

Data Files

```
n = 3;
capacity = 18;
profit = [29,19,8];
size = [8,5,3];
```

knapsack1.dzn

- `$ minizinc knapsack.mzn knapsack1.dzn`

`x = [1, 2, 0]`	solution
`----------`	solution found
`==========`	optimal proved

```
n = 5;
capacity = 200;
profit = [1300,1000,520,480,325];
size = [90,72,43,40,33];
```

knapsack2.dzn

- `$ minizinc knapsack.mzn knapsack2.dzn`

```
x = [1, 1, 0, 0, 1]
```


Modeling Objects

- Create a set naming the objects: `OBJ`
- Create a parameter array for each attribute of the object: `size`, `profit`
- Create a variable array for each decision of the object: `x`
- Build constraints over the set using comprehensions
- Note a model may have many sets of objects

Overview

- (Fixed) Sets to represent sets of objects
- Arrays over the object set to represent
  - object attributes
  - decisions about objects
- Generator expressions
  - To construct expressions over multiple objects

EOF

Arrays, Sets, Comprehensions

Peter Stuckey

Production Planning Example

A problem with the ToyProblem model is that the production rules and the available resources are hard wired into the model.

It is an example of simple kind of production planning problem in which we wish to

- determine how much of each kind of product to make to maximize the profit where
- manufacturing a product consumes varying amounts of some fixed resources.

- We can use a generic MiniZinc model to handle this kind of problem.

Production Planning Data

```
% Number of different products
int: nproducts;
set of int: PRODUCT = 1..nproducts;

% Profit per unit for each product
array[PRODUCT] of float: profit;

% Number of resources
int: nresources;
set of int: RESOURCE= 1..nresources;

% Amount of each resource available
array[RESOURCE] of float: capacity;

% Units of each resource required to produce
%       1 unit of product
array[PRODUCT,RESOURCE] of float: consumption;
```


Production Planning Constraints

```
% Variables: how much should we make of each product
array[PRODUCT] of var float: produce;

% Must produce a non-negative amount
constraint forall(p in PRODUCT)
                 (produce[p] >= 0.0);

% Production cannot only use the available resources:
constraint forall (r in RESOURCE)(
 sum (p in PRODUCT)(consumption[p, r] * produce[p])
  <= capacity[r]
);

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

output [ show(produce) ];
```


Production Planning Examples

- ToyProblem

```
nproducts = 2;
profit = [25.0,30.0];
nresources = 1; % hours
capacity = [40.0];
consumption = [| 1.0/200.0 | 1.0/140.0 |];
```

- CakeBaking

```
nproducts = 2;  % banana and chocolate cakes
profit = [4.0, 4.5];
nresources = 5; % flour, banana, sugar, butter, cocoa
capacity = [4000.0, 6.0, 2000.0, 500.0, 500.0];

consumption= [| 250.0, 2.0,  75.0, 100.0,   0.0
              | 200.0, 0.0, 150.0, 150.0, 75.0 |];
```


Sets

Sets are declared by

`set of` *type*

They may be sets of integers, floats or Booleans.

Set expressions:

- Set literals are of form `{e1,…,en}`
- Integer or float ranges are also sets
- Standard set operators are provided: `in, union, intersect, subset, superset, diff, symdiff`
- The size of the set is given by `card`

Some examples:

```
set of int: PRODUCT= 1..nproducts;
{1,2} union {3,4}
```

Sets can be used as types.

Arrays

An array can be multi-dimensional. It is declared by

`array`[*index_set1*,*index_set 2*, …,] `of` *type*

The index set of an array needs to be

- an integer range or
- a fixed set expression whose value is an integer range.

The elements in an array can be anything except another array, e.g.

```
array[PRODUCT, RESOURCE] of int: consume;
array[PRODUCTS] of var 0..mproducts: produce;
```

The built-in function `length` returns the number of elements in a 1-D array

Arrays (Cont.)

1-D arrays are initialized using a list

```
profit = [400, 450];
capacity = [4000, 6, 2000, 500, 500];
```

2-D array initialization uses a list with ``|" separating rows

```
consumption= [| 250, 2, 75,  100,  0
              | 200, 0, 150, 150, 75 |];
```

Arrays of any dimension (well ≤ 3) can be initialized from a list using the `array`*n*`d` family of functions:

```
consumption= array2d(1..2,1..5,
  [250,2,75,100,0,200,0,150,150,75]);
```

The concatenation operator ++ can be used with 1-D arrays: `profit = [400]++[450];`

Array & Set Comprehensions

MiniZinc provides comprehensions (like ML)

A set comprehension has form

- `{` *expr* | *generator1*, *generator2*, … `}`
- `{` *expr* | *generator1*, *generator2*, … `where` *bool-expr* `}`

An array comprehension is similar

- `[` *expr* | *generator1*, *generator2*, … `]`
- `[` *expr* | *generator1*, *generator2*, … `where` *bool-expr* `]`

E.g. `{i + j | i, j in 1..4 where i < j}`

```
= {1 + 2, 1 + 3, 1 + 4, 2 + 3, 2 + 4, 3 + 4}
= {3, 4, 5, 6, 7}
```


Array & Set Comprehensions Question

Exercise: What does b =?

```
set of int: COL = 1..5;
set of int: ROW = 1..2;
array[ROW,COL] of int: c =
    [| 250, 2,  75, 100,  0
     | 200, 0, 150, 150, 75 |];
b = array2d(COL, ROW,
   [c[j, i] | i in COL, j in ROW]);
```

Array & Set Comprehension Answer

- b is the transpose of c

```
[c[j, i] | i in COL, j in ROW ] =
[ 250, 200, 2, 0, 75,
  150, 100, 150, 0, 75]

b = [| 250,200
     |   2,  0
     |  75,150
     | 100,150
     |   0, 75
     |];
```


Iteration

MiniZinc provides a variety of built-in functions for operating over a list or set:

- Lists of numbers: `sum, product, min, max`
- Lists of constraints: `forall, exists`

MiniZinc provides a special syntax for calls to these (and other generator functions)

For example,

```
forall (i, j in 1..10 where i < j)
       (a[i] != a[j]);
```

is equivalent to

```
forall ([ a[i] != a[j]
        | i, j in 1..10 where i < j]);
```


Overview

- Real models apply to different sized data
- MiniZinc uses
  - sets to name objects
  - arrays to capture information about objects
  - comprehensions to build
    - constraints, and
    - expressions

    about different sized data

EOF

Linear Models

Peter Stuckey

Overview

- Many models involves
  - resources and limits
  - choices in production/transport
  - costs
- Constraints of this nature are often expressed as
  - linear constraints
- Solving technology for linear models is highly effective

Linear Constraints

- A linear expression is of the form
  - $\sum_{i=1..n} a_i x_i$
  - where a_i are constants and x_i are variables
- A linear inequality has the form
  - $\sum_{i=1..n} a_i x_i \leq a_0$
  - where a_i are constants and x_i are variables
- A linear equation has the form
  - $\sum_{i=1..n} a_i x_i = a_0$
  - where a_i are constants and x_i are variables
- Linear constraints are either
  - linear inequalities, or linear equations

Linear Models

- A linear model consists of
  - linear constraints, and
  - a linear objective
    - minimize <linear expression>, or
    - maximize <linear expression>
- Linear models are solvable using
  - linear programming (reals), and
  - (mixed) integer programming (integers)
- These solver technologies scale to
  - 100000 variables
  - 100000 constraints
  - and sometimes more

Shipping Problem

- A shipping company has to transport bags of cement to W warehouses from F factories daily. Each warehouse has a daily demand, and each factory a daily output. The cost of shipping one bag is given by a table, e.g.

cost	w1	w2	w3	w4
f1	6	5	7	9
f2	3	2	4	1
f3	7	3	9	5

- Find the minimal shipping costs

Shipping Problem: Data and Decisions

- Data

```
int: W; % number of Warehouses
set of int: WARE = 1..W;
int: F; % number of Factories
set of int: FACT = 1..F;
array[WARE] of int: demand;
array[FACT] of int: production;
array[FACT,WARE] of int: cost;
```

- Decisions

```
array[FACT,WARE] of var int: ship;
```


Shipping Problem: Constraints

- Only ship positive amounts

```
forall(f in FACT, w in WARE)
       (ship[f,w] >= 0);
```

- Ship to each warehouse its demand

```
forall(w in WARE)
       (sum(f in FACT)(ship[f,w])
        >= demand[w]);
```

- Dont ship more from each factory than it produces

```
forall(f in FACT)
       (sum(w in WARE)(ship[f,w])
        <= production[f]);
```


Shipping Problem: Objective

- Minimize total shipping costs

```
solve minimize
      sum(f in FACT, w in WARE)
         (cost[f,w]*ship[f,w]);
```

- ...

A Linear Model

- Each constraint is a linear constraint
- The objective is a linear term

- Solving with default solver
  - many solutions found, 43.246s

- Solving with MIP solver
  - one optimal solution found, 0.061s

Improving the Model

- Decisions

```
array[FACT,WARE] of var int: ship;
```

- Unbounded integers are bad for many solvers
  - can even make the problem intractable

- Limit the size of the variable!

```
int: m = max(production);
array[FACT,WARE] of var 0..m: ship;
```

- Remove the first set of constraints!
- But in this case makes no difference!

Overview

- Linear constraints are a major component of many models

- If we can build a linear model
  - or almost linear model
- Then we can solve it very efficiently

- Get used to modeling with linear constraints

EOF

Global Constraints

Peter Stuckey

Global Constraints

- Technically any constraint which can take an unbounded number of variables as input
  - so linear constraints are "global"
- Global constraints are
  - constraints that arise in many problems
- Global constraints make
  - models smaller
  - solving easier (since solvers can use the information of the structure)

alldifferent

- The alldifferent constraint
  - `alldifferent([x1,x2, …, xn])`
  - enforces that xi ≠ xj, for each i ≠ j
- Probably the most common global constraint
- alldifferent([7,3,2,5,1,6]) holds
- alldifferent([5,3,2,7,4,3]) does not hold

lex_less

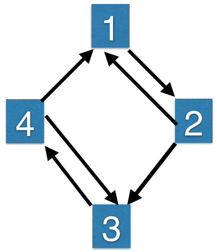
- The lexicographic less than constraint
  - `lex_less([x1, x2, …, xn], [y1, y2, … yn])`
  - requires that the [x1, x2, …, xn] is lexicographically smaller than [y1, y2, …, yn]
  - that is
    - x1 < y1 or (
    - x1 = y1 and (x2 < y2 or
    - x2 = y2 and (x3 < y3 or
    - ….
    - xn < yn) …)))
- Useful for symmetry breaking
- lex_less([7,3,5,4,2], [7,3,5,7,2]) holds

table

- The table constraint encodes arbitrary relations
  - `table([x1, x2, …, xn], T)`
  - requires that [x1,..,xn] take value from one row in the 2d array T
- table([x1,x2,x3], [| 3, 4, 5 | 5, 12, 13 | 6, 8, 10 |])
  - holds when [x1,x2,x3] = [5,12,13]
  - doesnt hold when x1 = 4

circuit

- The circuit constraint encodes Hamiltonian circuits, a single loop that visits each node in a graph exactly once
  - `circuit([x1, …, xn])`
  - xi = j means visit node j after node i
- For example
  - circuit([2,3,4,1]) holds
  - circuit([2,1,4,3]) doesn't hold
  - circuit([2,3,4,3]) doesn't hold

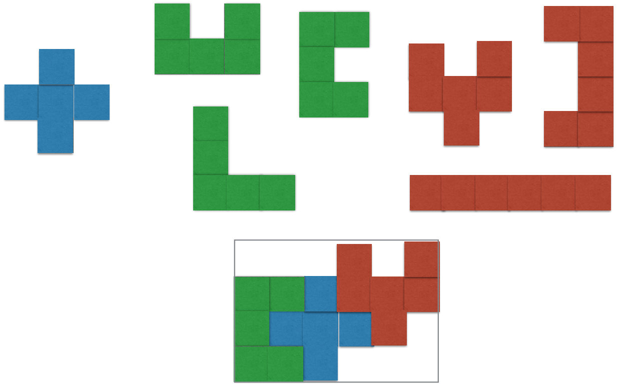


regular

- The regular constraint encodes that a sequence of values is part of a regular language
  - `regular([x1,…,xn], Q,S,d,q0,F)`
  - the sequence x1 x2 … xn is a member of the regular language defined by DFA (Q,S,d,q0,F)
- Useful for encoding complex state transitions, e.g. DFA for 1*((01)+1)*
  - regular([1,0,1,1,0,1,0,1,1], …) holds
  - regular([1,1,1,1,0,1,1], …) holds
  - regular([1,1,1,0,1,1,1], …) doesn't hold

geost

- Pack k dimensional objects with possibly different configurations so they dont overlap



Global Constraint Library

- MiniZinc includes a library of global constraints
  - Alldifferent and related constraints
  - Lexicographic constraints
  - Sorting constraints
  - Channeling constraints
  - Counting constraints
  - Scheduling constraints
  - Packing constraints
  - Extensional constraints (table, regular etc.)

Overview

- Global constraints are
  - important for making concise efficient models
- We will introduce more global constraints as their need arrives
- There are many global constraints
  - 100+ in MiniZinc
  - 300+ in the Global Constraint Catalog

EOF