DM841

Discrete Optimization

Lecture 4

Introduction to MiniZinc

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

Resume

- Modelling in MILP and CP
  - First example: graph labelling with consecutive numbers
  - Second example: Cryptarithmetic (or verbal arithmetic or cryptarithm): Send + More = Money
- Overview on constraint programming:
  representation (modeling language) + reasoning (search + propagation)
  - search = backtracking + branching
  - propagate, filtering, pruning
  - level of consistency (arc/generalized + value/bound/domain)

Outline

MiniZinc

- language for specifying: constrained optimization and decision problems over integers and real numbers.
- existential and universal quantifiers, sums over index sets, or logical connectives like implications and if-then-else statements
- supports defining predicates and functions that let users structure their models
- models are parametric (problem vs instance)
- MiniZinc compiler transforms model file + data file into FlatZinc model via libraries and predicate definitions that target the specific solvers, such as Constraint Programming (CP), Mixed Integer Linear Programming (MIP) or Boolean Satisfiability (SAT) solvers.
  Eg: MiniZinc allows the specification of global constraints by decomposition.
- allows annotations of the model to let the user fine tune the behaviour of the solver, independent of the declarative meaning of the model.
- third-party solvers need an executable and a solver specific MiniZinc library. They must be specified in a MiniZinc configuration file.

A First Script

```
% Colouring Australia using nc colours
int: nc = 3;
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)\t nt=\(nt)\t sa=\(sa)\n",
"q=\(q)\t nsw=\(nsw)\t v=\(v)\n",
"t=", show(t), "\n"];
```

Parameters and Variables

- comments `%` or `/* */`
- basic built-in, scalar types are integers (`int`), floating point numbers (`float`), Booleans (`bool`) and strings (`string`). Further: enumerated types; two compound built-in types: sets and multi-dimensional arrays; and the user extensible annotation type `ann`.
- decision variables are variables in the sense of mathematical or logical variables not in the sense of programming language variables.
- the domain can be given as part of the variable declaration and the type is inferred from there
- decision variables can be Booleans, integers, floating point numbers, or sets. Moreover, arrays of decision variables
- identifiers are made of characters or _ and must start with a char.

Parameters:

```
int : <var-name>
<l>..<u> : <var-name>
```

Decision variables:

```
var int : <var-name>
var <l>..<u> : <var-name>
```

variables instantiation vs type

The instantiation of a variable or value indicates if it is fixed to a known value or not.

- Parameters are instantiated at instance-time.
- Decision variables are instantiated at tune-time.

type-inst if both present at same time.

Constraints

Relational operators defined for built-in scalar types:

equal	= or ==
not equal	!=
strictly less than	<
strictly greater than	>
less than or equal to	<=
greater than or equal to	>=

Output

- between quotes for strings and enclosed by a `show` call if MiniZinc elements (or interpolation `"\(e)"`).
- string concatenation via operator ++
- formatted output: `show_int(n,X)` and `show_float(n,d,X)`

Command Line Execution

```
$ minizinc --solver gecode aust.mzn
```

```
$ minizinc -h
```

```
$ minizinc -a --solver gecode aust.mzn
```

(find all soluti)ons

Another Example: Production Planning

```
% Baking cakes for the school party
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
solve maximize 400*b + 450*c;
output ["no. of banana cakes = \(b)\n",
  "no. of chocolate cakes = \(c)\n"];
```

Arithmetics:

Addition	+	integer division	div
subtraction	-	integer modulus	mod
multiplication	*	absolute value	abs
power function	pow	float division	/

Data Files

```
$ minizinc cakes2.mzn -D "flour=4000;banana=6;sugar=2000;butter=500;cocoa=500;"

$ minizinc cakes2.mzn pantry.dzn
```

- solution output is in the same format as the dzn
- check parameters `assert(B,S)`

```
constraint assert(flour >= 0,"Invalid datafile: " ++ "Amount of flour should be non-negative");
```

A Real Numbers Example

```
% 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;


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"
];
```

- if I borrow \$1000 at 4% and repay \$260 per quarter, how much do I end up owing?
- if I want to borrow \$1000 at 4% and owe nothing at the end, how much do I need to repay?
- if I can repay \$250 a quarter, how much can I borrow at 4% to end up owing nothing?

- needs appropriate solvers
- addition (+), subtraction (-), multiplication (*) and floating point division (/).
- `int2float` (implicitly used when `a / b` and `a` and `b` are integers)
- arithmetic functions:

absolute value (abs),	sine (sin),	hyperbolic sine (sinh),
square root (sqrt) and power (pow)	cosine (cos),	hyperbolic cosine (cosh),
natural logarithm (ln),	tangent (tan),	hyperbolic tangent (tanh),
logarithm base 2 (log2),	arcsine (asin),	hyperbolic arcsine (asinh),
logarithm base 10 (log10),	arc-cosine (acos),	hyperbolic arccosine (acosh),
exponentiation of e (exp),	arctangent (atan),	hyperbolic arctangent (atanh)

- . sign for decimal
- Solve using LP ut only if linear constraints

Basic Structure

```
include <filename>;

<type inst expr>: <variable> [ = ] <expression>;

<variable> = <expression>;

constraint <Boolean expression>;

solve satisfy;
solve maximize <arithmetic expression>;
solve minimize <arithmetic expression>;

output [ <string expression>, ..., <string expression> ];
```

Arrays

one- and multi-dimensional arrays which are declared using the type:

```
array [ <index-set-1>, ..., <index-set-n> ] of <type-inst>
```

They can contain: integers, enums, Booleans, floats or strings both as parameters and as variables (except for string)

Index sets are:

- an integer range,
- a set variable initialised to an integer range
- an enumeration type.

```
[ <expr-1>, ..., <expr-n> ] % 1d
[| <expr-1-1>, ..., <expr-1-n> |
             ...               |
   <expr-m-1>, ..., <expr-m-n> |]
```

```
array2d(1..3, 1..2, [1, 2, 3, 4, 5, 6])
% is equivalent to
[|1, 2 |3, 4 |5, 6|]
```

- concatenation operator ++, eg:
  `[4000, 6] ++ [2000, 500, 500]` evaluates to `[4000, 6, 2000, 500, 500]`
- a list is a one-dimensional array
- the built-in function `length` returns the length of a one-dimensional array.

Array

A new example:

- modelling temperatures on a rectangular sheet of metal.
- approximate the temperatures across the sheet by breaking the sheet into a finite number of elements in a two-dimensional matrix.

```
set of int: HEIGHT = 0..h;
set of int: CHEIGHT = 1..h-1;
set of int: WIDTH = 0..w;
set of int: CWIDTH = 1..w-1;
array[HEIGHT,WIDTH] of var float: t; % temperature at point (i,j)

% access as t[i,j]
```

- there is no explicit list type, one-dimensional arrays with an index set $1..n$ behave like lists, we can use them as lists.

Laplace's equation states that when the plate reaches a steady state the temperature at each internal point is the average of its orthogonal neighbours.

```
% Laplace equation: each internal temp. is average of its neighbours
constraint forall(i in CHEIGHT, j in CWIDTH)(
                 4.0*t[i,j] = t[i-1,j] + t[i,j-1] + t[i+1,j] + t[i,j+1])
```

On the edges the temperature is equal

```
constraint forall(i in CHEIGHT)(t[i,0] = left);
constraint forall(i in CHEIGHT)(t[i,w] = right);
constraint forall(j in CWIDTH)(t[0,j] = top);
constraint forall(j in CWIDTH)(t[h,j] = bottom);
```

```
% corner constraints
constraint t[0,0]=0.0;
constraint t[0,w]=0.0;
constraint t[h,0]=0.0;
constraint t[h,w]=0.0;
```

Determine the temperatures in a plate broken into 5×5 elements with left, right and bottom temperature 0 and top temperature 100

Sets

For parameters: sets of integers, enums, floats or Booleans.
For variables: sets of integers or enums.

```
set of <type-inst> : <var-name> ;
{ <expr-1>, ..., <expr-n> } % set
 <expr-1> .. <expr-2> % range
```

Set operations

element membership	`in`
(non-strict) subset relationship	`subset`
(non-strict) superset relationship	`superset`
union	`union`
inter-section	`intersect`
set difference	`diff`
symmetric set difference	`symdiff`
number of elements in the set	`card`

Enumerations

Enumerated types `enums`

```
enum <var-name> ; % declaration as an unknown set of elements
enum <var-name> = { <var-name-1>, ..., <var-name-n> } ; % definition via assignment
```

```
enum Color;
Color = { red, yellow, blue };
```

Elements of an enumerated type of n elements internally are represented as integers $1 \ldots n$:

- they can be compared
- they are ordered, by the order they appear in the enumerated type definition,
- they can be iterated over,
- they can appear as indices of arrays,

They can be used for indexing

```
array[Products] of int: profit;
```

List Comprehensions

Array and list comprehensions:

```
  [i+j | i,j in 1..3 where j<i] % [2+1,3+1,3+2] = [3, 4, 5].
  {i + j | i, j in 1..3 where j < i}  %  {3, 4, 5}.
```

```
[ <expr> | <generator-exp> ]
<generator>
<generator> where <bool-exp>  % filtering
<identifier>, ..., <identifier> in <array-exp> % generator
```

Array comprehension can generate also variables while set comprehensions not.

Built-in functions

`forall` takes a one-dimensional array and aggregates the elements.
The array is made of Boolean expressions (that is, constraints) and **`forall`** returns a single Boolean expression which is the logical conjunction

```
array of 1..3: a;
forall( [a[i] != a[j] | i,j in 1..3 where i < j]);
forall(i,j in 1..3 where i < j]) ([a[i] != a[j]]); % equivalent to the above one
```

The expressions mean:

```
a[1] != a[2] /\ a[1] != a[3] /\ a[2] != a[3]
```

Aggregation Functions

Aggregation functions:

- for arithmetic arrays are: **sum**, **product**, **min**, **max**
  When applied to an empty array, **sum** returns 0, **product** returns 1, **min** and **max** give a run-time error.

- for arrays containing Boolean expressions are: **forall** (logical conjunction, all hold), **exists** (logical disjunction, at least one holds), **xorall** (ensures that an odd number hold), **iffall** (an even number of holds).

Generator Call Expression:

```
<agg-func> ( [ <exp> | <generator-exp> ] )
<agg-func> ( <generator-exp> ) ( <exp> )
```

(same syntax as **forall**)

Production Planning Revisited

```
enum Products; % Products to be produced
array[Products] of int: profit; % profit per unit for each product
enum Resources; % Resources to be used
array[Resources] of int: capacity; % amount of each resource available
array[Products, Resources] of int: consumption; % to produce 1 unit of product

constraint assert(forall (r in Resources, p in Products)
           (consumption[p,r] >= 0), "Error: negative consumption");

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

array[Products] of var 0..mproducts: produce; % Variables: how much should we make of each product
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])
);
constraint forall (r in Resources) (
      used[r] <= capacity[r]
);
% Maximize profit
solve maximize sum (p in Products) (profit[p]*produce[p]);
output [ "\(p) = \(produce[p]);\n" | p in Products ] ++
       [ "\(r) = \(used[r]);\n" | r in Resources ];
```

Data file for simple production planning model

```
Products = { BananaCake, ChocolateCake };
profit = [400, 450]; % in cents
Resources = { Flour, Banana, Sugar, Butter, Cocoa };
capacity = [4000, 6, 2000, 500, 500];
consumption= [| 250, 2, 75, 100, 0,
              | 200, 0, 150, 150, 75 |];
```