DM841

Constraint Programming

Modeling Exercises

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

[*Based on slides by Christian Schulte, KTH Royal Institute of Technology*]

Outline

1. Magic Squares
2. Sudoku
3. Seat Planning
4. 8-Queens
5. Bin Packing
6. Summary

Outline

1. Magic Squares
2. Sudoku
3. Seat Planning
4. 8-Queens
5. Bin Packing
6. Summary

Magic Squares

2	9	4
7	5	3
6	1	8

Unique solution for n=3, upon the symmetry breaking of slide 99.

Magic Squares

- Find an $n \times n$ matrix such that
  - every field is integer between 1 and n^2
  - fields pairwise distinct
  - sums of rows, columns, two main diagonals are equal

- Very hard problem for large n
- Here: we just consider the case $n=3$

Model

- For each matrix field have variable x_{ij}
  - $x_{ij} \in \{1, .., 9\}$
- One additional variable s for sum
  - $s \in \{1, .., 9\times9\}$
- All fields pairwise distinct
  - distinct(x_{ij})
- For each row i have constraint
  - $x_{i0} + x_{i1} + x_{i2} = s$
  - columns and diagonals similar

Script

- Straightforward
- Branching strategy
  - first-fail
  - split again: arithmetic constraints
  - try to come up with something that is really good!
- Generalize it to arbitrary *n*

Symmetries

- Clearly, we can require for first row that first and last variable must be in order
- Also, for opposing corners
- In all (other combinations possible)
  - $x_{00} < x_{02}$
  - $x_{02} < x_{20}$
  - $x_{00} < x_{22}$

Important Observation

- We know the sum of all fields

  $1 + 2 + \ldots + 9 = 9(9+1)/2=45$
- We "know" the sum of one row

  s
- We know that we have three rows

  $3 \times s = 45$

Implied Constraints

- The constraint model already implies

  $3 \times s = 45$

  - implies solutions are the same
- However, adding a propagator for the constraint drastically improves propagation
- Often also: redundant or implied constraint

Effect

Simple model	92 nodes
Symmetry breaking	29 nodes
Implied constraint	6 nodes

Summary: Magic Squares

- Add implied constraints
  - are implied by model
  - increase constraint propagation
  - reduce search space
  - require problem understanding
- Also as usual
  - break symmetries
  - choose appropriate branching

Magic Squares: MiniZinc Model

```
include "alldifferent.mzn";

int: n = 4;
set of int: NUMBERS = 1..n^2;
set of int: ROW = 1..n;
set of int: COL = 1..n;

int:l = sum(NUMBERS) div n;

array[ROW,COL] of var NUMBERS: pos;

constraint alldifferent ([pos[i,j] | i in ROW, j in COL]);
constraint forall(i in ROW)(sum(j in COL)(pos[i,j]) = l);
constraint forall(j in COL)(sum(i in ROW)(pos[i,j]) = l);
constraint sum(i in 1..n)(pos[i,i])= l;
constraint sum(i in 1..n)(pos[i,n-i+1])=l;

% Symmetry breaking constraints
constraint pos[n,1] < pos[1,n];
constraint pos[1,1] < pos[1,n];
constraint pos[1,1] < pos[n,1];

solve satisfy;
output[if  j = 1 then "\n" else " " endif ++
    show(pos[i,j])| i in ROW,j in COL] ++ ["\n"];
```

Outline

1. Magic Squares
2. Sudoku
3. Seat Planning
4. 8-Queens
5. Bin Packing
6. Summary

Example: Sudoku

Model and solve the following Sudoku in MIP and CP

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

Sudoku: ILP model

Let y_{ijt} be equal to 1 if digit t appears in cell (i, j). Let N be the set $\{1, \ldots, 9\}$, and let J_{kl} be the set of cells (i, j) in the 3×3 square in position k, l.

$$\sum_{j \in N} y_{ijt} = 1, \qquad \forall i, t \in N,$$

$$\sum_{j \in N} y_{jit} = 1, \qquad \forall i, t \in N,$$

$$\sum_{i,j \in J_{kl}} y_{ijt} = 1, \qquad \forall k, l = \{1, 2, 3\}, t \in N,$$

$$\sum_{t \in N} y_{ijt} = 1, \qquad \forall i, j \in N,$$

$$y_{i,j,a_{ij}} = 1, \qquad \forall i, j \in \text{ given instance.}$$

Sudoku: CP model

Model:

$$
\begin{aligned}
&X_{ij} \in N, && \forall i, j \in N, \\
&X_{ij} = a_{ij}, && \forall i, j \in \text{ given instance}, \\
&\text{alldifferent}([X_{1i}, \ldots, X_{9i}]), && \forall i \in N, \\
&\text{alldifferent}([X_{i1}, \ldots, X_{i9}]), && \forall i \in N, \\
&\text{alldifferent}(\{X_{ij} \mid ij \in J_{kl}\}), && \forall k, l \in \{1, 2, 3\}.
\end{aligned}
$$

Search: backtracking

Sudoku: **MiniZinc model**

```
include "alldifferent.mzn";

int: n = 9;
set of int: NUMS = 1..9;
set of int: SQUARES = 1..3;

array[NUMS, NUMS] of 0..n: sudoku;
array[NUMS, NUMS] of var NUMS: solution;

% Fill sudoku with initial board
constraint forall(i, j in NUMS)(
 if sudoku[i, j] != 0 then solution[i, j] = sudoku[i, j] else true endif
);

% Rows, columns, and squares must each contain numbers 1–9
constraint forall(n in NUMS)(alldifferent(row(solution, n)));
constraint forall(n in NUMS)(alldifferent(col(solution, n)));
constraint forall(r, c in SQUARES)(alldifferent(
          [solution[3*(r-1) + i, 3*(c-1) + j] | i in SQUARES, j in
             SQUARES]
          ));

solve satisfy;
```

```
output [
  show(solution[i, j]) ++
  if j = n then
    if i mod 3 = 0 /\ i != n
         then
      "\n-----------------"
    else
      ""
    endif ++ "\n"
  elseif j mod 3 = 0 then
    "|"
  else
    " "
  endif
  | i in NUMS, j in NUMS
];
```

```
sudoku = [|
  0, 4, 3, 0, 8, 0, 2, 5, 0 |
  6, 0, 0, 0, 0, 0, 0, 0, 0 |
  0, 0, 0, 0, 0, 1, 0, 9, 4 |
  9, 0, 0, 0, 0, 4, 0, 7, 0 |
  0, 0, 0, 6, 0, 8, 0, 0, 0 |
  0, 1, 0, 2, 0, 0, 0, 0, 3 |
  8, 2, 0, 5, 0, 0, 0, 0, 0 |
  0, 0, 0, 0, 0, 0, 0, 0, 5 |
  0, 3, 4, 0, 9, 0, 7, 1, 0
|];
```

Sudoku: CP model (revisited)

$$
\begin{aligned}
& X_{ij} \in N, && \forall i, j \in N, \\
& X_{ij} = a_t, && \forall i, j \in \text{ given instance}, \\
& \text{alldifferent}([X_{1i}, \ldots, X_{9i}]), && \forall i \in N, \\
& \text{alldifferent}([X_{i1}, \ldots, X_{i9}]), && \forall i \in N, \\
& \text{alldifferent}(\{X_{ij} \mid ij \in J_{kl}\}), && \forall k, l \in \{1, 2, 3\}.
\end{aligned}
$$

Redundant Constraint:

$$
\begin{aligned}
& \sum_{j \in N} X_{ij} = 45, && \forall i \in N, \\
& \sum_{j \in N} X_{ji} = 45, && \forall i \in N, \\
& \sum_{ij \in J_{kl}} X_{ij} = 45, && k, l \in \{1, 2, 3\}.
\end{aligned}
$$

Outline

1. Magic Squares
2. Sudoku
3. Seat Planning
4. 8-Queens
5. Bin Packing
6. Summary

Version 1: from [SMT]

```
enum Guests = { bride, groom, bestman, bridesmaid, bob, carol, ted, alice, ron, rona, ed, clara };
set of int: Seats = 1..12;
set of int: Hatreds = 1..5;
array[Hatreds] of Guests: h1 = [groom, carol, ed, bride, ted];
array[Hatreds] of Guests: h2 = [clara, bestman, ted, alice, ron];
set of Guests: Males = { groom, bestman, bob, ted, ron, ed };
set of Guests: Females = { bride, bridesmaid, carol, alice, rona, clara };

array[Guests] of var Seats: pos; % seat of guest
array[Hatreds] of var Seats: p1; % seat of guest 1 in hatred
array[Hatreds] of var Seats: p2; % seat of guest 2 in hatred
array[Hatreds] of var bool: sameside; % seats of hatred on same side
array[Hatreds] of var Seats: cost; % penalty of hatred

constraint alldifferent(pos);

% Males and females in odd and even positions
constraint forall(m in Males)( pos[m] mod 2 == 1 );
constraint forall(w in Females)( pos[w] mod 2 == 0 );

% Ed not on corners
constraint not (pos[ed] in {1, 6, 7, 12});

% Bride and groom next to each other
constraint abs(pos[bride] - pos[groom]) <= 1 /\ (pos[bride] <= 6 <-> pos[groom] <= 6);
```

```
% Cost of positioning based on hatreds (use auxillary arrays to find cost)
constraint forall(h in Hatreds)(
  p1[h] = pos[h1[h]] /\
  p2[h] = pos[h2[h]] /\
  sameside[h] = p1[h] <= 6 <-> p2[h] <= 6 /\
  cost[h] = sameside[h] * abs(p1[h] - p2[h]) + (1 - sameside[h]) * (abs(13 - p1[h] - p2[h]) + 1)
);

solve minimize sum(h in Hatreds)(cost[h]);

output [ "\(g) " | s in Seats, g in Guests where fix(pos[g]) == s];
```

Version 2: Different Tables — Set Variables

```
include "all_disjoint.mzn";

int: n;
set of int: PERSON = 1..n;
int: T; % number of tables
set of int: TABLE = 1..T;
int: S; % table size
array[int, 1..2] of PERSON: couples;
array[int, 1..3] of PERSON: hatreds;

% Result is the sets of people on each table (unknown seats)
array[TABLE] of var set of PERSON: table;

predicate same_table(PERSON: p1, PERSON: p2) = exists(t in TABLE)({p1, p2} subset table[t]);

% Tables seat at most S people each, and each person has one seat
constraint forall(t in TABLE)(card(table[t]) <= S);
constraint forall(p in PERSON)(exists(t in TABLE)(p in table[t]));
% exists is logical disjunction hence a person can still be in more than
% one table:
constraint all_disjoint(table);

% Ensure couples sit together
constraint forall(p in index_set_1of2(couples))(same_table(couples[p, 1], couples[p, 2]));
```

```
% Objective function — cost of seating, based on hatreds
% Unhappiness of a table is just the maximum unhappiness within that table
var int: obj = sum(t in TABLE)(
  max(c in index_set_1of2(hatreds))(
    hatreds[c, 3] * same_table(hatreds[c, 1], hatreds[c, 2])
  )
);

solve minimize obj;

output ["\(table) = \(obj)"];
```

```
n = 10;
T = 3;
S = 4;
couples = [| 1, 2
           | 4, 7
           | 8, 9 |];
```

```
hatreds = [| 1, 3, 2
           | 1, 6, 8
           | 1, 9, 3
           | 2, 5, 4
           | 2, 6, 9
           | 2, 10, 4
           | 3, 6, 1
           | 3, 8, 2
           | 4, 5, 2
           | 4, 9, 5
           | 5, 10, 3
           | 7, 8, 6
           | 8, 10, 2
           | 9, 10, 4 |];
```

Version 2: Integer Variables + Set Variables

```
include ‘‘globals.mzn’’;

int: n;
set of int: PERSON = 1..n;
int: T; % number of tables
set of int: TABLE = 1..T;
int: S; % tables size
array[int,1..2] of PERSON: couples;

array[PERSON] of var TABLE: seat;
array[TABLE] of var set of PERSON: table;

predicate not_same_table(PERSON:p1, PERSON: p2) =
          seat[p1] != seat[p2];

constraint global_cardinality_low_up(seat, [t|t in TABLE],
                          [0|t in TABLE], [S|t in TABLE]);

constraint forall(c in index_set_1of2(couples))
          (not_same_table(couples[c,1],couples[c,2]));

var int: obj = sum(c in index_set_1of2(couples))
          (seat[couples[c,1]] + seat[couples[c,2]]);
```

```
constraint forall(t in TABLE, p in PERSON)
          (p in table[t] <-> seat[p] = t);

solve minimize obj;

output [show(table), " = ", show(obj)];
```

```
n = 20;
T = 5;
S = 5;
couples = [| 1, 2 | 4, 5 | 6, 7 | 8, 10
           | 11, 12 | 13, 14 | 17, 18 |];
```

Solution:

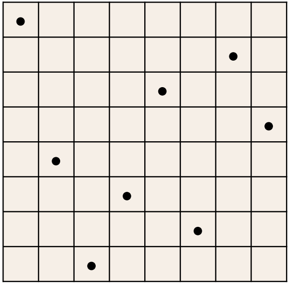
```
[{1,4,6,8,11}, {2,5,7,13,17},
     {3,10,12,14,18}, {9,15,16,19,20}, {}]
     = 27
```

But it took long. Symmetry breaking?

Outline

1. Magic Squares
2. Sudoku
3. Seat Planning
4. 8-Queens
5. Bin Packing
6. Summary

Problem Statement



- Place 8 queens on a chess board such that the queens do not attack each other
- Straightforward generalizations
  - place an arbitrary number: *n* Queens
  - place as closely together as possible

What Are the Variables?

- Representation of position on board
- First idea: two variables per queen
  - one for row
  - one for column
  - $2 \cdot n$ variables
- Insight: on each column there will be a queen!

Fewer Variables…

- Have a variable for each column
  - value describes row for queen
  - *n* variables

- Variables: $x_0, \ldots, x_7$

  where $x_i \in \{0, \ldots, 7\}$

Other Possibilities

- For each field: number of queen
  - which queen is not interesting, so…
  - n^2 variables
- For each field on board: is there a queen on the field?
  - 8×8 variables
  - variable has value 0: no queen
  - variable has value 1: queen
  - n^2 variables

Constraints: No Attack

- not in same column
  - by choice of variables
- not in same row
  - $x_i \neq x_j$ for $i \neq j$
- not in same diagonal
  - $x_i - i \neq x_j - j$ for $i \neq j$
  - $x_i - j \neq x_j - i$ for $i \neq j$
- $3 \cdot n \cdot (n - 1)$ constraints

Fewer Constraints…

- Sufficient by symmetry

  $i < j$ instead of $i \neq j$

- Constraints
  - $x_i \neq x_j$ for $i < j$
  - $x_i - i \neq x_j - j$ for $i < j$
  - $x_i - j \neq x_j - i$ for $i < j$

- $3/2 \cdot n \cdot (n - 1)$ constraints

Even Fewer Constraints

- Not same row constraint

  $x_i \neq x_j$ for $i < j$

  means: values for variables pairwise distinct
- Constraints
  - distinct(x_0, …, x_7)
  - $x_i - i \neq x_j - j$ for $i < j$
  - $x_i - j \neq x_j - i$ for $i < j$

Pushing it Further…

- Yes, also diagonal constraints can be captured by distinct constraints
  - ~~see assignment~~

```
distinct(x0,     x1,    ...,  x7)
distinct(x0-0,  x1-1, ..., x7-7)
distinct(x0+0, x1+1, ..., x7+7)
```

Good Branching?

- Naïve is not a good strategy for branching

- Try the following (see assignment)
  - first fail
  - place queen as much in the middle of a row
  - place queen in knight move fashion

Summary 8 Queens

- Variables
  - model should require few variables
  - good: already impose constraints
- Constraints
  - do not post same constraint twice
  - try to find “big” constraints subsuming many small constraints
    - more efficient
    - often, more propagation (to be discussed)

Outline

1. Magic Squares
2. Sudoku
3. Seat Planning
4. 8-Queens
5. Bin Packing
6. Summary

Bin-packing

Objects of different height must be packed into a finite number of bins or containers each of height H in a way that minimizes the number of bins used.
Model the problem and solve the following specific instance:

```
num_objs = 6;
objs = [360, 850, 630, 70, 700, 210]; % heights of objects
bin_capacity = 1440; % height of bins
```

Let m be the number of bins and n the number of items

Variables:
binary variables to represent for each bin whether the object is packed or not
$x_{ij} \in \mathbb{B}^{m \times n}$ for $i \in [1..m]$ and $j \in [1..n]$
Auxiliary variables to represent the load of a bin.

Bin-packing

```
% binary variables
array[1..num_bins, 1..num_stuff] of var 0..1: bins;
% calculate how many things a bin takes
array[1..num_bins] of var 0..bin_capacity: bin_loads;


% number of loaded bins (which we will minimize)
var 0..num_bins: num_loaded_bins;

% minimize the number of loaded bins
% solve minimize num_loaded_bins;

% alternative solve statement
solve :: int_search(
         [bins[i,j] | i in 1..num_bins, j in 1..num_stuff], % ++ bin_loads
         input_order, % first_fail,
         indomain_max,
         complete)
     minimize num_loaded_bins;
```

```
constraint
   % sanity clause: No thing can be larger than capacity.
   % forall(s in 1..num_stuff) (
   %    stuff[s] <= bin_capacity
   % )
   % /\ % the total load in the bin cannot exceed bin_capacity
   forall(b in 1..num_bins) (
      bin_loads[b] = sum(s in 1..num_stuff) (size[s]*bins[b,s])
   )
   /\ % calculate the total load for a bin
   sum(s in 1..num_stuff) (size[s]) = sum(b in 1..num_bins) (bin_loads[b])
   /\ % a thing is packed just once
   forall(s in 1..num_stuff) (
      sum(b in 1..num_bins) (bins[b,s])  = 1
   )
   % /\ % symmetry breaking:
   %    % if bin_loads[i+1] is > 0 then bin_loads[i] must be > 0
   %  forall(b in 1..num_bins-1) (
   %    (bin_loads[b+1] > 0 -> bin_loads[b] > 0)
   %    % /\ % and should be filled in order of weight
   %    % bin_loads[b] >= bin_loads[b+1]
   %  )
   /\
   decreasing(bin_loads) :: domain
   /\ % another symmetry breaking: first bin must be loaded
   bin_loads[1] > 0
   /\ % calculate num_loaded_bins
   num_loaded_bins = sum(b in 1..num_bins) (bool2int(bin_loads[b] > 0))
```

Outline

1. Magic Squares
2. Sudoku
3. Seat Planning
4. 8-Queens
5. Bin Packing
6. Summary

Common modeling principles

- what are the variables
- finding the constraints
- finding the propagators
- implied (redundant) constraints
- finding the branching
- symmetry breaking

Modeling Strategy

- Understand problem
  - identify variables
  - identify constraints
  - identify optimality criterion
- Attempt initial model $\rightsquigarrow$ simple?
  try on examples to assess correctness
- Improve model $\rightsquigarrow$ much harder!
  scale up to real problem size

Viewpoints

Viewpoint (definition of variables and domain extension $(\mathcal{X}, \mathcal{D})$):

- same solutions
- can be combined
- rule of thumb in choosing a viewpoint:
  it should allow the constraints to be easily and concisely expressed;
  the problem to be described using as few constraints as possible, as long as those constraints have efficient, low-complexity propagation algorithms

Releated concept: auxiliary variables and linking or channelling

Modeling Constraints

Better understood if:

- aware of the range of constraints supported by the constraint solver and the level of consistency enforced on each
- have some idea of the complexity of the corresponding propagation algorithms.
- combine them
- use global constraints
- extensional constraints
- implied constraints