Constraint Programming

Justin Pearson

Uppsala University

1st July 2016

Special thanks to Pierre Flener, Joseph Scott and Jun He
Outline

1. Introduction to Constraint Programming (CP)
2. Introduction to MiniZinc
3. Constraint Programming with Strings
4. Some recent, and not so Recent Advances
5. Final Thoughts
Constraint Programming in a Nutshell

Slogan of CP

Constraint Program = Model [ + Search ]

CP provides:

- high level declarative modelling abstractions,
- a framework to separate search from from modelling.

Search proceeds by intelligent backtracking with intelligent inference called propagation that is guided by high-level modelling constructs called global constraints.
Outline

1. Introduction to Constraint Programming (CP)
   - Modelling
   - Propagation
   - Systematic Search
   - History of CP

2. Introduction to MiniZinc

3. Constraint Programming with Strings

4. Some recent, and not so Recent Advances

5. Final Thoughts
### Constraint-Based Modelling

**Example (Port tasting, PT)**

<table>
<thead>
<tr>
<th>Year</th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1994</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1977</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1970</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Constraints to be satisfied:**

- Equal jury size: Every wine is evaluated by 3 judges.
- Equal drinking load: Every judge evaluates 3 wines.
- Fairness: Every port pair has 1 judge in common.
Example (Port tasting, PT)

<table>
<thead>
<tr>
<th></th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1994</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1977</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1970</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Constraints to be **satisfied**:

- Equal jury size: Every wine is evaluated by 3 judges.
- Equal drinking load: Every judge evaluates 3 wines.
- Fairness: Every port pair has 1 judge in common.
### Example (Port tasting, PT)

<table>
<thead>
<tr>
<th></th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>2003</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>2000</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1994</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>1977</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>1970</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>1966</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
</tbody>
</table>

### Constraints to be satisfied:

- Equal jury size: Every port is evaluated by 3 judges.
- Equal drinking load: Every judge evaluates 3 wines.
- Fairness: Every port pair has 1 judge in common.
## Constraint-Based Modelling

### Example (Port tasting, PT)

<table>
<thead>
<tr>
<th>Year</th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2003</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1994</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1977</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1970</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1966</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Constraints to be **satisfied**:

- Equal jury size: Every port is evaluated by 3 judges.
- Equal drinking load: Every judge evaluates 3 wines.
- Fairness: Every port pair has 1 judge in common.
Example (PT integer model: \( \checkmark \sim \sim 1 \) and \( \sim \sim 0 \))

```plaintext
1  int: NbrVint; int: NbrJudges;
2  set of int: Wines = 1..NbrVint;
3  set of int: Judges = 1..NbrJudges;
4  int: JurySize; int: Capacity; int: Fairness;
5  array[Wines,Judges] of var 0..1: PT;
6  solve satisfy;
7  forall(t in Wines)
8     (JurySize = sum(j in Judges)(PT[t,j]));
9  forall(j in Judges)
10     (Capacity = sum(t in Wines)(PT[t,j]));
11  forall(t,t’ in Wines where t<t’)
12     (Fairness = sum(j in Judges)(PT[t,j]*PT[t’,j]));
```

Example (Instance data for the SMT PT instance)

```plaintext
1  NbrVint = 7; NbrJudges = 7;
2  JurySize = 3; Capacity = 3; Fairness = 1;
```
Example (Idea for another PT model)

<table>
<thead>
<tr>
<th>Year</th>
<th>Judges</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>{Alva, Dan, Eva}</td>
</tr>
<tr>
<td>2003</td>
<td>{Alva, Jim, Leo}</td>
</tr>
<tr>
<td>2000</td>
<td>{Alva, Mia, Ulla}</td>
</tr>
<tr>
<td>1994</td>
<td>{Dan, Jim, Mia}</td>
</tr>
<tr>
<td>1977</td>
<td>{Dan, Leo, Ulla}</td>
</tr>
<tr>
<td>1970</td>
<td>{Eva, Jim, Ulla}</td>
</tr>
<tr>
<td>1966</td>
<td>{Eva, Leo, Mia}</td>
</tr>
</tbody>
</table>

Constraints to be **satisfied**:
- Equal jury size: Every port is evaluated by 3 judges.
- Equal drinking load: Every judge evaluates 3 wines.
- Fairness: Every port pair has 1 judge in common.
Example (PT set model: each port has a judge set)

arr[Wines] of var set of Judges: PT;

forall(t in Wines)
  (JurySize = card(PT[t]));
forall(j in Judges)
  (Capacity = sum(t in Wines)(bool2int(j in PT[t])));
forall(t,t’ in Wines where t<t’)
  (Fairness = card(PT[t] inter PT[t’]));

Example (Instance data for the PT instance)

...
Example (Sudoku model)

1. array[1..9,1..9] of var 1..9: Sudoku;
2. ...
3. solve satisfy;
4. forall(r in 1..9) (ALLDIFFERENT([Sudoku[r,c] | c in 1..9]));
5. forall(c in 1..9) (ALLDIFFERENT([Sudoku[r,c] | r in 1..9]));
6. forall(i,j in {1,4,7}) (ALLDIFFERENT([Sudoku[r,c] | r in i..i+2, c in j..j+2]));
Global Constraints

Global constraints such as `ALLDIFFERENT` and `SUM` enable the preservation of combinatorial sub-structures of a constraint problem, both while modelling it and while solving it. Dozens of $n$-ary constraints (Catalogue) have been identified and encapsulate complex propagation algorithms declaratively., including

- n-ary linear and non-linear arithmetic
- Rostering under balancing & coverage constraints
- Scheduling under resource & precedence constraints
- Geometrical constraints between points, segments, . . .

There are many more.
CP Solving = Search + Propagation

A CP solver conducts search interleaved with propagation:

- Familiar idea as in SAT solvers.
  - Propagate until fix point.
  - Make a choice.
  - Backtrack on failure.

- Because we have global constraints we can often do more propagation than unit-propagation of clauses at each step.
The **ALLDIFFERENT** constraint

Consider the $n$-ary constraint **ALLDIFFERENT**, with $n = 4$:

$$\text{ALLDIFFERENT}([a, b, c, d]) \quad (1)$$

Modelling: (1) is equivalent to $\frac{n(n-1)}{2}$ binary constraints:

$$a \neq b \land a \neq c \land a \neq d \land b \neq c \land b \neq d \land c \neq d \quad (2)$$

Inference: (1) propagates much better than (2). Example:

$$a \in \{4, 5\}, \ b \in \{4, 5\}, \ c \in \{3, 4\}, \ d \in \{1, 2, 3, 4, 5\}$$

No domain pruning by (2).
The *ALLDIFFERENT* constraint

Consider the $n$-ary constraint *ALLDIFFERENT*, with $n = 4$:

\[
\text{ALLDIFFERENT}([a, b, c, d])
\]  

(1)

Modelling: (1) is equivalent to $\frac{n(n-1)}{2}$ binary constraints:

\[
a \neq b \land a \neq c \land a \neq d \land b \neq c \land b \neq d \land c \neq d
\]  

(2)

Inference: (1) propagates much better than (2). Example:

\[
a \in \{4, 5\}, \quad b \in \{4, 5\}, \quad c \in \{3, 4\}, \quad d \in \{1, 2, 3, 4, 5\}
\]

No domain pruning by (2).
The **ALLDIFFERENT** constraint

Consider the \( n \)-ary constraint **ALLDIFFERENT**, with \( n = 4 \):

\[
\text{ALLDIFFERENT}([a, b, c, d])
\]  

(1)

Modelling: (1) is equivalent to \( \frac{n(n-1)}{2} \) binary constraints:

\[
a \neq b \land a \neq c \land a \neq d \land b \neq c \land b \neq d \land c \neq d
\]  

(2)

Inference: (1) propagates much better than (2). Example:

\[
a \in \{4, 5\}, \ b \in \{4, 5\}, \ c \in \{3, 4\}, \ d \in \{1, 2, 3, 4, 5\}
\]

No domain pruning by (2).
The **ALLDIFFERENT** constraint

Consider the $n$-ary constraint **ALLDIFFERENT**, with $n = 4$:

$$\text{ALLDIFFERENT}([a, b, c, d]) \quad (1)$$

Modelling: (1) is equivalent to $\frac{n(n-1)}{2}$ binary constraints:

$$a \neq b \land a \neq c \land a \neq d \land b \neq c \land b \neq d \land c \neq d \quad (2)$$

Inference: (1) propagates much better than (2). Example:

$$a \in \{4, 5\}, \ b \in \{4, 5\}, \ c \in \{3, 4\}, \ d \in \{1, 2, 3, 4, 5\}$$

No domain pruning by (2). But perfect propagation by (1)
Propagator for **ALLDIFFERENT**

Solutions to `ALLDIFFERENT([u, v, w, x, y, z])` correspond to maximum matchings in a bipartite graph for the domains:

- $u \in \{0, 1\}$
- $v \in \{1, 2\}$
- $w \in \{0, 2\}$
- $x \in \{1, 3\}$
- $y \in \{2, 3, 4, 5\}$
- $z \in \{5, 6\}$
Propagator for **ALLDIFFERENT**

Mark all edges of *some* maximum matching; Hopcroft-Karp algorithm takes $O(m\sqrt{n})$ time for $n$ variables and $m$ values:

- $u \in \{0, 1\}$
- $v \in \{1, 2\}$
- $w \in \{0, 2\}$
- $x \in \{1, 3\}$
- $y \in \{2, 3, 4, 5\}$
- $z \in \{5, 6\}$
Propagator for **ALLDIFFERENT**

Mark all edges of *some* maximum matching; Hopcroft-Karp algorithm takes $O(m\sqrt{n})$ time for $n$ variables and $m$ values:

- $u \in \{0, 1\}$
- $v \in \{1, 2\}$
- $w \in \{0, 2\}$
- $x \in \{1, 3\}$
- $y \in \{2, 3, 4, 5\}$
- $z \in \{5, 6\}$
Propagator for **ALLDIFFERENT**

Mark all other edges in *all* other maximum matchings.

\[ u \in \{0, 1\} \]
\[ v \in \{1, 2\} \]
\[ w \in \{0, 2\} \]
\[ x \in \{1, 3\} \]
\[ y \in \{2, 3, 4, 5\} \]
\[ z \in \{5, 6\} \]
Propagator for **ALLDIFFERENT**

Mark all other edges in *all* other maximum matchings.

\[
\begin{align*}
  u & \in \{0, 1\} \\
  v & \in \{1, 2\} \\
  w & \in \{0, 2\} \\
  x & \in \{1, 3\} \\
  y & \in \{2, 3, 4, 5\} \\
  z & \in \{5, 6\}
\end{align*}
\]
Propagator for **ALLDIFFERENT**

Every still unmarked edge is in *no* maximum matching. Propagate accordingly within the current domains:

- $u \in \{0, 1\}$
- $v \in \{1, 2\}$
- $w \in \{0, 2\}$
- $x \in \{1, 3\}$
- $y \in \{2, 3, 4, 5\}$
- $z \in \{5, 6\}$
Propagator for **ALLDIFFERENT**

Every still unmarked edge is in *no* maximum matching. Propagate accordingly within the current domains:

- \( u \in \{0, 1\} \)
- \( v \in \{1, 2\} \)
- \( w \in \{0, 2\} \)
- \( x \in \{1, 3\} \)
- \( y \in \{2, 3, 4, 5\} \)
- \( z \in \{5, 6\} \)
Search + Propagation in the PT example

<table>
<thead>
<tr>
<th></th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>2003</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>2000</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1994</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>1977</td>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1970</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Alva cannot be a judge of 1977 as that would violate the second constraint (every judge evaluates 3 wines). Actually, Alva cannot be a judge of 1994, 1970, 1966 either, for the same reason, and this was already propagated when trying the search guess that Alva be a judge of 2000!
### Search + Propagation in the PT example

<table>
<thead>
<tr>
<th>Year</th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>2003</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>2000</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>1994</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>1977</td>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1970</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Alva cannot be a judge of 1977 as that would violate the second constraint (every judge evaluates 3 wines).
## Search + Propagation in the PT example

<table>
<thead>
<tr>
<th>Year</th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2003</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2000</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1994</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>1977</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>1970</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Alva cannot be a judge of 1977 as that would violate the second constraint (every judge evaluates 3 wines).
## Search + Propagation in the PT example

<table>
<thead>
<tr>
<th></th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>2003</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>2000</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1994</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1977</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1970</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Alva cannot be a judge of 1977 as that would violate the second constraint (every judge evaluates 3 wines). Actually, Alva cannot be a judge of 1994, 1970, 1966 either, for the same reason, and this was already propagated when trying the search guess that Alva be a judge of 2000!
### Search + Propagation in the PT example

<table>
<thead>
<tr>
<th></th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>2003</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>2000</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1994</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>1977</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1970</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Alva cannot be a judge of 1977 as that would violate the second constraint (every judge evaluates 3 wines). Actually, Alva cannot be a judge of 1994, 1970, 1966 either, for the same reason, and this was already propagated when trying the search guess that Alva be a judge of 2000!
PT: partial assignment

<table>
<thead>
<tr>
<th>Year</th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>2003</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>2000</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1994</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1977</td>
<td>✗</td>
<td>?</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1970</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### PT: partial assignment

<table>
<thead>
<tr>
<th></th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2003</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2000</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>1994</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1977</td>
<td>x</td>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1970</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Search guess: Dan is a judge of 1977. (✓ guesses first.)
PT: partial assignment

<table>
<thead>
<tr>
<th>Year</th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2003</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2000</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1994</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1977</td>
<td>x</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1970</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Search guess: Dan is a judge of 1977. (✓ guesses first.)
**PT: partial assignment**

<table>
<thead>
<tr>
<th>Year</th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>2003</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>2000</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1994</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>1977</td>
<td>X</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1970</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Propagation: Dan cannot be a judge of 1970 and 1966 as otherwise the second constraint (every judge evaluates 3 wines) would be violated for Dan.
### PT: partial assignment

<table>
<thead>
<tr>
<th>Year</th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>2003</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>2000</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1994</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1977</td>
<td>✗</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1970</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Propagation: Dan cannot be a judge of 1970 and 1966 as otherwise the second constraint (every judge evaluates 3 wines) would be violated for Dan.
**PT: partial assignment**

<table>
<thead>
<tr>
<th>Year</th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>2003</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>2000</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>1994</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1977</td>
<td>✗</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1970</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Propagation: Eva, Jim, and Mia cannot be judges of 1977 as otherwise the third constraint (every port pair has 1 judge in common) would be violated.
PT: partial assignment

<table>
<thead>
<tr>
<th>Year</th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>2003</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>2000</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1994</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>1977</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>1970</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Propagation: Eva, Jim, and Mia cannot be judges of 1977 as otherwise the third constraint (every port pair has 1 judge in common) would be violated.
PT: partial assignment

<table>
<thead>
<tr>
<th>Year</th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>2003</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>2000</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1994</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1977</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>1970</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Propagation: Leo and Ulla must be judges of 1977 as otherwise the first constraint (every port is evaluated by 3 judges) would be violated for 1977.
### PT: partial assignment

<table>
<thead>
<tr>
<th></th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>2003</td>
<td>✓</td>
<td></td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>2000</td>
<td>✓</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>1994</td>
<td></td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>1977</td>
<td></td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>1970</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Propagation: Leo and Ulla must be judges of 1977 as otherwise the first constraint (every port is evaluated by 3 judges) would be violated for 1977.
**PT: partial assignment**

<table>
<thead>
<tr>
<th>Year</th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2003</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2000</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1994</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>1987</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>1970</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Propagation: Eva must be a judge of 1970 and 1966 as otherwise the second constraint (every judge evaluates 3 wines) would be violated for Eva.
PT: partial assignment

<table>
<thead>
<tr>
<th>Year</th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>2003</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>2000</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1994</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>1977</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>1970</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>1966</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Propagation: Eva must be a judge of 1970 and 1966 as otherwise the second constraint (every judge evaluates 3 wines) would be violated for Eva.
PT: partial assignment

<table>
<thead>
<tr>
<th>Year</th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>2003</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>2000</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1994</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>1977</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1970</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Common fixpoint reached: No more propagation possible.
## PT: partial assignment

<table>
<thead>
<tr>
<th>Year</th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>2003</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>2000</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>1994</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>1977</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>1970</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Search guess: Jim is a judge of 1970. (✔ guesses first.)

Propagation: etc.
Systematic Search Algorithm, with Propagation

1: **post** all given constraints, and propagate
2: **while** there is at least one suspended constraint **do**
3: **pick** a variable \( x \) with \( |\text{domain}(x)| \geq 2 \)
4: **pick** some value \( d \in \text{domain}(x) \)
5: **try** one of mutually exclusive guesses (constraints), say \( x = d \) & \( x \neq d \), or \( x > d \) & \( x \leq d \), and propagate

Heuristics

- Line 3: variable ordering heuristic: smallest domain . . .
- Lines 4 & 5: value & guess ordering heuristic: max, . . .
- Tree exploration: depth-first, . . ., with backtracking

Example (PT search)

```plaintext
1 solve :: int_search(PT,input_order,indomain_max,complete)
2 satisfy;
```
### History of CP

**Stand-alone languages:**

- **ALICE** by Jean-Louis Laurièrè, France, 1976
- **CHIP** at ECRC, Germany, 1987 – 1990, then marketed by Cosytec, France
- **OPL**, by P. Van Hentenryck, USA, and ILOG, France: front-end to both ILOG CP Optimizer and ILOG CPLEX
- **Comet**, by P. Van Hentenryck and L. Michel, USA
- **MiniZinc**, by the ORG group at CSIRO/NICTA, Australia

**Libraries (the ones listed before “;” are open-source):**

- **Prolog**: ECLiPSe, . . . ; SICStus Prolog, . . .
- **C++**: Gecode, Google or-tools; IBM CP Optimizer, CHIP
- **Java**: Choco, Google or-tools, JaCoP; . . .
- **Scala**: OscaR; . . .
Outline

1. Introduction to Constraint Programming (CP)

2. Introduction to MiniZinc
   - Features
   - Backends
   - Internals

3. Constraint Programming with Strings

4. Some recent, and not so Recent Advances

5. Final Thoughts
Overview

MiniZinc is a declarative language to model combinatorial problems using constraints.

- Mainly developed at Nicta (Now centre 61), Australia
- Introduced in 2007
- Version 2.0 in 2014, with ongoing development.
- Annual solver competition since 2008

http://www.minizinc.org
MiniZinc Features

- Declarative language
- High-level syntax
- Solving technology independent
- Solver independent
- Separation of *model* and *data*
- Open-source toolchain
- Library of many predefined constraint predicates
- Support for user-defined predicates and functions
- Support for annotations
MiniZinc Backends

- Constraint programming – CP: Gecode, SICStus Prolog, Choco, Google or-Tools, JaCoP, Mistral, ...
MiniZinc Backends

- Constraint programming – CP: Gecode, SICStus Prolog, Choco, Google or-Tools, JaCoP, Mistral, ...
- Lazy-clause generation – LCG: Opturion CPX, Chuffed, ...
MiniZinc Backends

- Constraint programming – CP: Gecode, SICStus Prolog, Choco, Google or-Tools, JaCoP, Mistral, ...
- Lazy-clause generation – LCG: Opturion CPX, Chuffed, ...
- Mixed Integer programming – MIP: G12/MIP, CPLEX, Gurobi, ...
MiniZinc Backends

- Constraint programming – CP: Gecode, SICStus Prolog, Choco, Google or-Tools, JaCoP, Mistral, . . .
- Lazy-clause generation – LCG: Opturion CPX, Chuffed, . . .
MiniZinc Backends

- Constraint programming – CP: Gecode, SICStus Prolog, Choco, Google or-Tools, JaCoP, Mistral, . . .
- Lazy-clause generation – LCG: Opturion CPX, Chuffed, . . .
- SAT: G12/SAT, . . .
MiniZinc Backends

- Constraint programming – CP: Gecode, SICStus Prolog, Choco, Google or-Tools, JaCoP, Mistral, ...
- Lazy-clause generation – LCG: Opturion CPX, Chuffed, ...
- Mixed Integer programming – MIP: G12/MIP, CPLEX, Gurobi, ...
- Local search and metaheuristics – LS: OscaR/CBLS, EasyLocal++, YACS, ...
- SAT: G12/SAT, ...
- SAT modulo theories – SMT: fzn2smt with Yices 2, ...
MiniZinc Backends

- Constraint programming – CP: Gecode, SICStus Prolog, Choco, Google or-Tools, JaCoP, Mistral, . . .
- Lazy-clause generation – LCG: Opturion CPX, Chuffed, . . .
- SAT: G12/SAT, . . .
- SAT modulo theories – SMT: fzn2smt with Yices 2, . . .
- Hybrids: iZplus, Minisat(ID), . . .
MiniZinc Backends

- Constraint programming – CP: Gecode, SICStus Prolog, Choco, Google or-Tools, JaCoP, Mistral, …
- Lazy-clause generation – LCG: Opturion CPX, Chuffed, …
- Mixed Integer programming – MIP: G12/MIP, CPLEX, Gurobi, …
- Local search and metaheuristics – LS: OscaR/CBLS, EasyLocal++, YACS, …
- SAT: G12/SAT, …
- SAT modulo theories – SMT: fzn2smt with Yices 2, …
- Hybrids: iZplus, Minisat(ID), …
- Portfolios of solvers: sunny-cp, …
# MiniZinc Challenge 2015: Some Results

Winners per model (no portfolio or parallel categories):

<table>
<thead>
<tr>
<th>Model</th>
<th>Winner</th>
<th>Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>costas array</td>
<td>Mistral</td>
<td>CP</td>
</tr>
<tr>
<td>capacitated VRP</td>
<td>iZplus</td>
<td>hybrid</td>
</tr>
<tr>
<td>gfd schedule</td>
<td>Chuffed</td>
<td>LCG</td>
</tr>
<tr>
<td>grid colouring</td>
<td>MinisatID</td>
<td>hybrid</td>
</tr>
<tr>
<td>instruction scheduling</td>
<td>Chuffed</td>
<td>LCG</td>
</tr>
<tr>
<td>large scheduling</td>
<td>OR-Tools</td>
<td>CP</td>
</tr>
<tr>
<td>application mapping</td>
<td>JaCoP</td>
<td>CP</td>
</tr>
<tr>
<td>multi-knapsack</td>
<td>CPLEX</td>
<td>MIP</td>
</tr>
<tr>
<td>portfolio design</td>
<td>OscaR/CBLS</td>
<td>LS</td>
</tr>
<tr>
<td>open stacks</td>
<td>Chuffed</td>
<td>LCG</td>
</tr>
<tr>
<td>project planning</td>
<td>Chuffed</td>
<td>LCG</td>
</tr>
<tr>
<td>radiation</td>
<td>Gurobi</td>
<td>MIP</td>
</tr>
<tr>
<td>satellite management</td>
<td>Gurobi</td>
<td>MIP</td>
</tr>
<tr>
<td>time dependent TSP</td>
<td>G12/FD</td>
<td>CP</td>
</tr>
<tr>
<td>zephyrus configuration</td>
<td>CPLEX</td>
<td>MIP</td>
</tr>
</tbody>
</table>
Solving a MiniZinc Model

Two phases:

1. Compile (or flatten) the model into a FlatZinc model.
2. Interpret the FlatZinc model with a solver.

FlatZinc is a low-level subset of MiniZinc:

- No quantifiers or array comprehensions
- Only built-in constraint predicates of the targeted solver
- Only variables of types supported by the targeted solver
- Constraints only on (arrays of) variables and parameters
Flattening

- Unroll all quantifiers and array comprehensions
- Inline all function and predicate calls
- Evaluate all expressions not depending on the value of a variable
- Replace unsupported variable types
- Decompose complex expressions, introducing new variables

Flattening produces solver-specific models:

- Not all solvers have the same set of built-in constraint predicates
- Different solvers may use different function and constraint predicate definitions, also called decompositions
Example: **ALLDIFFERENT**

We have all ready seen that of a CP solver it is better to write:
Instead of writing:

```plaintext
constraint alldifferent(x);
```

instead of:

```plaintext
constraint forall(i in 1..n, j in i+1..n)
  (x[i] != x[j]);
```

Even if your (constraint)-solver does not support **ALLDIFFERENT**
There are strong advantages to using the former:

- More concise and clear model
- Better solving, **even for non-CP solvers**
Decompositions of \textsc{AllDifferent}

Decompositions can be provided in the model or each backend can provide its own decompositions.

**Default Decomposition**

\begin{verbatim}
predicate alldifferent(array[int] of var int: x) =
  forall(i,j in index_set(x) where i < j)
  ( x[i] != x[j] );
\end{verbatim}

Used, e.g., by the SMT backend \textit{fzn2smt}

**Built-in Predicate**

\begin{verbatim}
predicate alldifferent(array[int] of var int: x);
\end{verbatim}

Most CP solvers use a dedicated propagator.
Decompositions

Linear Decomposition for MIP

predicate alldifferent(array[int] of var int: x) =
    let {
        array[int,int] of var 0..1: y = eq_encode(x)
    } in forall(j in index_set_2of2(y))
        ( sum(i in index_set(x))(y[i,j]) <= 1 );

[...]

predicate equality_encoding(var int: xi, array[int] of var 0..1: yi) =
    sum(j in index_set(yi))(yi[j]) = 1
    /
    sum(j in index_set(yi))(j * yi[j]) = xi;
Decompositions

Boolean Decomposition for SAT: Ladder Encoding

```plaintext
predicate alldifferent(array[int] of var int: x) =
  let {
    array[int,int] of var bool: y = int2bools(x);
    array[...,...] of var bool: a;
  } in forall(i in ..., j in ...)
    ((a[i-1,j] -> a[i,j])
     \n     (y[i,j] <-> (not a[i-1,j] \ a[i,j])));

function array[int,int] of var bool: int2bools
  (array[int] of var int: x) =
  [...]```
Decompositions

- `ALLDIFFERENT` is merely one example of how a (technology-specific) modelling idiom can be encapsulated by a constraint predicate.
Decompositions

- **ALLDIFFERENT** is merely one example of how a (technology-specific) modelling idiom can be encapsulated by a constraint predicate.
- Solver-specific decompositions exist for many other predefined constraint predicates.
Decompositions

- **ALLDIFFERENT** is merely one example of how a (technology-specific) modelling idiom can be encapsulated by a constraint predicate.

- Solver-specific decompositions exist for many other predefined constraint predicates.

- Solver developers and researchers can add their own decompositions for existing and new constraint predicates.
**Decompositions**

- **ALLDIFFERENT** is merely one example of how a (technology-specific) modelling idiom can be encapsulated by a constraint predicate.

- Solver-specific decompositions exist for many other predefined constraint predicates.

- Solver developers and researchers can add their own decompositions for existing and new constraint predicates.

- Decompositions are transparent to the modeller.
Recent and Current Research

MiniZinc and its toolchain are continuously extended, both by the core team at Monash University and the University of Melbourne, and by other researchers around the world.

One can divide into three categories:

- Tool support
- Modelling extensions
- Solving extensions
Tool Support

- Interface from general-purpose programming languages (coming soon)
- Integrated development environment (since 2014)
- Visualisation: CP search tree visualisation (CP 2015)
- Globaliser: suggests global constraints to replace parts of a given model (CP 2013)
Modelling Extensions

- User-defined functions (CP 2013)
- MiningZinc: extension targeted at itemset mining (CP 2013)
- Modelling with option types (CPAIOR 2014)
- Nested constraint programs (CP 2014)
- Stochastic programming support (CP 2014)
- CLPZinc: Prolog-based extension (PPDP 2015)
- Auto pre-solving of predicates.
- Ongoing work, strings.
Solving Extensions

- OscaR/CBLS, EasyLocal++, YACS: local search backends expand the solving capabilities (CPAIOR 2015, MIC 2015)

- sunny-cp: MiniZinc makes it easier to design powerful portfolio solvers (ICLP 2014, IJCAI 2015)

- New decompositions: linearisation, Boolean encoding, set variable elimination (CP 2015 and current research)

- Search specification and scripting languages: search combinators (CP 2011), MiniSearch (CP 2015), annotations for local search (current research)

- Ongoing, native string type of Gecode.
Outline

1. Introduction to Constraint Programming (CP)

2. Introduction to MiniZinc

3. Constraint Programming with Strings
   - Motivation
   - Decision Variables
   - Constraints
   - Bounded-length Strings in CP
   - Native
   - Experimental Evaluation

4. Some recent, and not so Recent Advances

5. Final Thoughts
String Constraint Problems

Constraints on strings occur in a wide variety of real-world application areas, such as:

- Web security: SQL code injection & XSS vulnerabilities
- Program testing: program test-case generation
- Program analysis
- Model checking
- Database testing
- Interactive configuration: of bikes or radar systems, say
- Biology: stem loops & pseudo-knots in bio-sequences
- Image processing: scene analysis
- Natural language processing: morphological analysis

Most of the work in the work is taken from [Joesph Scott’s Ph.D. Thesis].
Decision Variables

There are essentially three types of strings that are considered in CP and in verification:

- Fixed length strings,
- Unknown but bounded length string,
- Unbounded but finite length.

In the first two cases the domain of a string variable is a finite set of strings. The last case requires decision procedures. Even when the sets are finite we have to make approximations.
Constraints

Some of the constraints that we have considered, $s_j$ are string variables, $c_j$ are character variables and $i_j$ are integer variables.

- $\text{EQUAL}(s_1, s_2)$ if $s_1$ and $s_2$ are equal, that is $s_1 = s_2$
- $\text{REVERSE}(s_1, s_2)$ if $s_1 = c_1 c_2 \cdots c_n$ and $s_2 = c_n \cdots c_2 c_1$
- $\text{CONCAT}(s_1, s_2, s)$ if $s_1 \oplus s_2 = s$, with concatenation $\oplus$
- $\text{SUBSTRING}(s_1, i_1, i_2, s)$ if $s_1[i_1 : i_2] = s$
- $\text{CHARACTERAT}(s, i, c)$ if $\text{SUBSTRING}(s, i, i, \text{“c”})$
- $\text{LENGTH}(s, i)$ if $s$ has $i$ characters, that is $|s| = i$
- $\text{REGULAR}(s, \mathcal{R})$ if $s$ is a word of a regular language $\mathcal{R}$, given by a regular expression or a finite automaton
- $\text{CONTEXTFREE}(s, \mathcal{F})$ if $s$ is a word of a context-free language $\mathcal{F}$, given by a context-free grammar
- $\text{COUNT}(s, [c_1, \ldots, c_n], [i_1, \ldots, i_n])$ if in $s$ all $c_j$ occur $i_j$ times
Note that the decision variables and the strings live in a constraint solver not as a specialised solver but as first class citizens. Therefore you get all the reasoning on integers for free.
Bounded-length Strings

A bounded length is a string of a (possibly)-unknown that is bounded from above by some implementation specific constant. Possible implementations

- Decompose are arrays of characters with a length variable and a padding character (padded).
- Implement special propagators to work with the padding approach approach (aggregate)
- Implement a bespoke variable type inside a constraint solver (native).

We need padding characters because when a domain becomes empty a CP solver will fail at that node and backtrack.
New data-types in CP

Implementing a new data-type can be quite difficult.

- Choice of representation.
- How to interact with the propagation loop
  - Changes in domains are signalled by events that form a lattice.
  - A propagator subscribes to events to control how much information and how often the propagator is woken up.
- What exactly should we propagate?
  - A representation is an approximation of the mathematical reality.
  - We have a galois-based framework for specifying propagators and deriving what propagation should and can be done in different representations.
Bounded-length Strings in CP
bounded-length sequence representation

A \( b \)-length sequence over-approximates a set of strings of length \( \leq b \).

\[ \langle \langle A[1], \ldots, A[b] \rangle, N \rangle \]

Each \( A[i] \) is a set of characters. \( N \) in interval giving the lower and upper bound of the string length. Note that this is already an over approximation of a set of strings.
Implementation 1: Aggregate Strings

- Idea: represent the components as an aggregation of pre-existing variable types (e.g., integers, sets, Booleans, etc)
- For example, a $b$-length sequence can be represented by:
  \[
  \langle \langle A[1], \ldots, A[b] \rangle, N \rangle \simeq \langle \langle D(X_1), \ldots, D(X_b) \rangle, D(N) \rangle
  \]

- Complications
  - undefinedness: empty variable domains $\iff$ failure!
  - representation invariant: implemented as an additional constraint
Implementation 1: Aggregate Strings

Aggregate representation easy to implement in existing CP solvers

logical constraint semantics → derive → formal propagator function

indexical constraint checker → synthesize → indexical propagator description

⟨α, γ⟩-consistency → bounds(ℤ) consistency → domain consistency

Code generation and propagator synthesis:

[Monette et al @ CP’12] and [Monette et al @ CP’15]
Implementation 2: Native Strings

Three tightly related choices:

- **data structure**
  - candidate lengths $\subset \mathbb{N}$: range sequence, bitset, interval
  - candidate characters $\subset \mathbb{N}$: range sequence, bitset, interval
  - sequence: array, list, list of arrays, etc

- **restriction operations must consider undefinedness**
  - work by removing values from **components**
  - result is to remove strings from the **domain**

- **propagation events**
  - representation invariant: many promising looking event systems are not monotonic
Implementation 2: Native Strings

Three tightly related choices:

- **data structure**
  - candidate lengths $\subset \mathbb{N}$: range sequence, bitset, interval
  - candidate characters $\subset \mathbb{N}$: range sequence, bitset, interval
  - sequence: array, list, list of arrays, etc

- **restriction operations** must consider undefinedness
  - work by removing values from **components**
  - result is to remove strings from the **domain**

- **propagation events**
  - representation invariant: many promising looking event systems are not monotonic
Results

- Three CP-based methods investigated
  - padded
    - fixed-length, pessimistically large array of integer variables
    - multiple occurrences of a padding character allowed at end
  - aggregate
  - native

- 2 benchmark sets from software verification:
  - Kaluza
  - Norn

- even the padded method beats Kaluza [He et al @ CP '13]

- Results
  - Aggregate implementation: 3 to 4 times speed-up over padded
  - Native implementation: roughly 10 times speed-up over padded
Outline

1. Introduction to Constraint Programming (CP)
2. Introduction to MiniZinc
3. Constraint Programming with Strings
4. Some recent, and not so Recent Advances
   - Lazy Clause Generation
   - Hybrid’s with MIP
5. Final Thoughts
Lazy Clause Advantages

CP solvers are good at propagation, SAT solvers are good at clause learning. How can we combine them?

- Not such a good idea: Compile global constraints into SAT clauses. Resulting SAT problems are often too big.
- Better idea: Generate clauses on the fly during search.
- Even better idea: Have global constraints that also add clauses during search that encode propagation.
- State of the Art: Also have the global constraint give higher-level reasons for failure, referred to as explanations in the literature.

See the work (Chuffed) of [Peter Stuckey et. al. in Melbourne]
Hybrid’s with MIP

- Split a problem into sub-problems and solve some parts with CP and some parts with Mixed integer programming (MIP). With some care very good results can be obtained. [Logic based Benders Decomposition. John Hooker]

- Sometimes CP is not so good at propagating information from cost functions. Use ideas similar to Lagrangian Relaxation in CP and move problem constraints into the cost function.
Final Thoughts

Please don’t take this slide so seriously.

- In CP we are less interested in decision procedures and more interested in the actual solutions.
- We are much better at finding solutions than proving unsatisfiability.
- We almost always consider finite domains this makes integration with SMT much harder.
- Because we consider only finite domains, decidability is not a problem. We are not afraid of CFG intersection.
- Sometimes in SMT equality propagation really pays off. We have no real way of doing equality propagation.
Thank you

Questions?
To read about Strings and Galois Connections see:


http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-273311

The CFG constraint:


http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-196347
The work on propagator synthesis by Monette et.al. among other things can be found at http://www.it.uu.se/research/group/astra/publications

To learn about Lazy Clause Generation look at the papers of Peter Stuckey et.al. at http://people.eng.unimelb.edu.au/pstuckey/papers.html