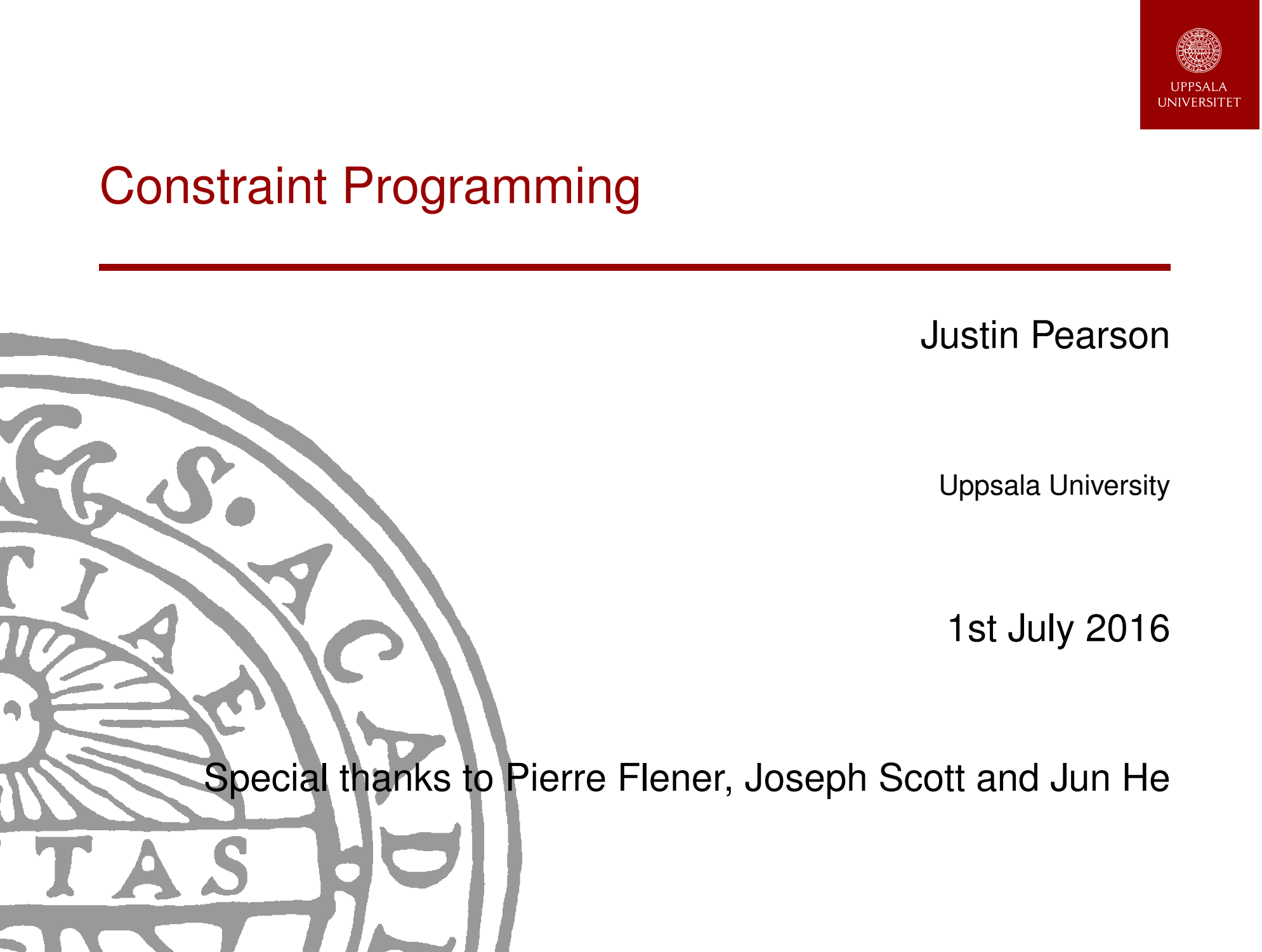


Constraint Programming

Justin Pearson

Uppsala University

1st July 2016

Special thanks to Pierre Flener, Joseph Scott and Jun He

Outline

UPPSALA UNIVERSITET

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.

UPPSALA UNIVERSITET

Outline

UPPSALA UNIVERSITET

Constraint-Based Modelling

Example (Port tasting, PT)

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011							
2003							
2000							
1994							
1977							
1970							
1966							

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.

UPPSALA UNIVERSITET

Constraint-Based Modelling

Example (Port tasting, PT)

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	✓	✓	✓				
2003	✓			✓	✓		
2000	✓					✓	✓
1994		✓		✓		✓	
1977		✓			✓		✓
1970			✓	✓			✓
1966			✓		✓	✓	

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.

UPPSALA UNIVERSITET

Constraint-Based Modelling

Example (Port tasting, PT)

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	✓	✓	✓	–	–	–	–
2003	✓	–	–	✓	✓	–	–
2000	✓	–	–	–	–	✓	✓
1994	–	✓	–	✓	–	✓	–
1977	–	✓	–	–	✓	–	✓
1970	–	–	✓	✓	–	–	✓
1966	–	–	✓	–	✓	✓	–

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.

UPPSALA UNIVERSITET

Constraint-Based Modelling

Example (Port tasting, PT)

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	1	1	1	0	0	0	0
2003	1	0	0	1	1	0	0
2000	1	0	0	0	0	1	1
1994	0	1	0	1	0	1	0
1977	0	1	0	0	1	0	1
1970	0	0	1	1	0	0	1
1966	0	0	1	0	1	1	0

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.

UPPSALA UNIVERSITET

Example (PT integer model: ✓ $\rightsquigarrow$ 1 and − $\rightsquigarrow$ 0)

```
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)

```
1  NbrVint = 7; NbrJudges = 7;
2  JurySize = 3; Capacity = 3; Fairness = 1;
```

UPPSALA UNIVERSITET

Example (Idea for another PT model)

2011	{	*Alva*,		*Dan*,		*Eva*					}
2003	{	*Alva*,					*Jim*,		*Leo*		}
2000	{	*Alva*,								*Mia*,	*Ulla*}
1994	{			*Dan*,			*Jim*,		*Mia*		}
1977	{			*Dan*,				*Leo*,		*Ulla*}	
1970	{				*Eva*,		*Jim*,			*Ulla*}	
1966	{				*Eva*,			*Leo*,	*Mia*		}

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.

UPPSALA UNIVERSITET

Example (PT set model: each port has a judge *set*)

```
1  ...
2  ...
3  ...
4  ...
5  array[Wines] of var set of Judges: PT;
6  ...
7  forall(t in Wines)
8     (JurySize = card(PT[t]));
9  forall(j in Judges)
10    (Capacity = sum(t in Wines)(bool2int(j in PT[t])));
11 forall(t,t' in Wines where t<t')
12    (Fairness = card(PT[t] inter PT[t']));
```

Example (Instance data for the PT instance)

```
1  ...
2  ...
```

UPPSALA UNIVERSITET

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

Example (Sudoku model)

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

UPPSALA UNIVERSITET

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.

UPPSALA UNIVERSITET

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.

UPPSALA UNIVERSITET

The AllDifferent constraint

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

$$\text{AllDifferent}([a, b, c, d]) \tag{1}$$

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

$$a \neq b \wedge a \neq c \wedge a \neq d \wedge b \neq c \wedge b \neq d \wedge c \neq d \tag{2}$$

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

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

No domain pruning by (2).

UPPSALA UNIVERSITET

The AllDifferent constraint

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

$$\text{AllDifferent}([a, b, c, d]) \tag{1}$$

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

$$a \neq b \wedge a \neq c \wedge a \neq d \wedge b \neq c \wedge b \neq d \wedge c \neq d \tag{2}$$

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

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

No domain pruning by (2).

UPPSALA UNIVERSITET

The AllDifferent constraint

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

$$\text{AllDifferent}([a, b, c, d]) \tag{1}$$

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

$$a \neq b \wedge a \neq c \wedge a \neq d \wedge b \neq c \wedge b \neq d \wedge c \neq d \tag{2}$$

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).

UPPSALA UNIVERSITET

The AllDifferent constraint

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

$$\text{AllDifferent}([a, b, c, d]) \tag{1}$$

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

$$a \neq b \wedge a \neq c \wedge a \neq d \wedge b \neq c \wedge b \neq d \wedge c \neq d \tag{2}$$

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

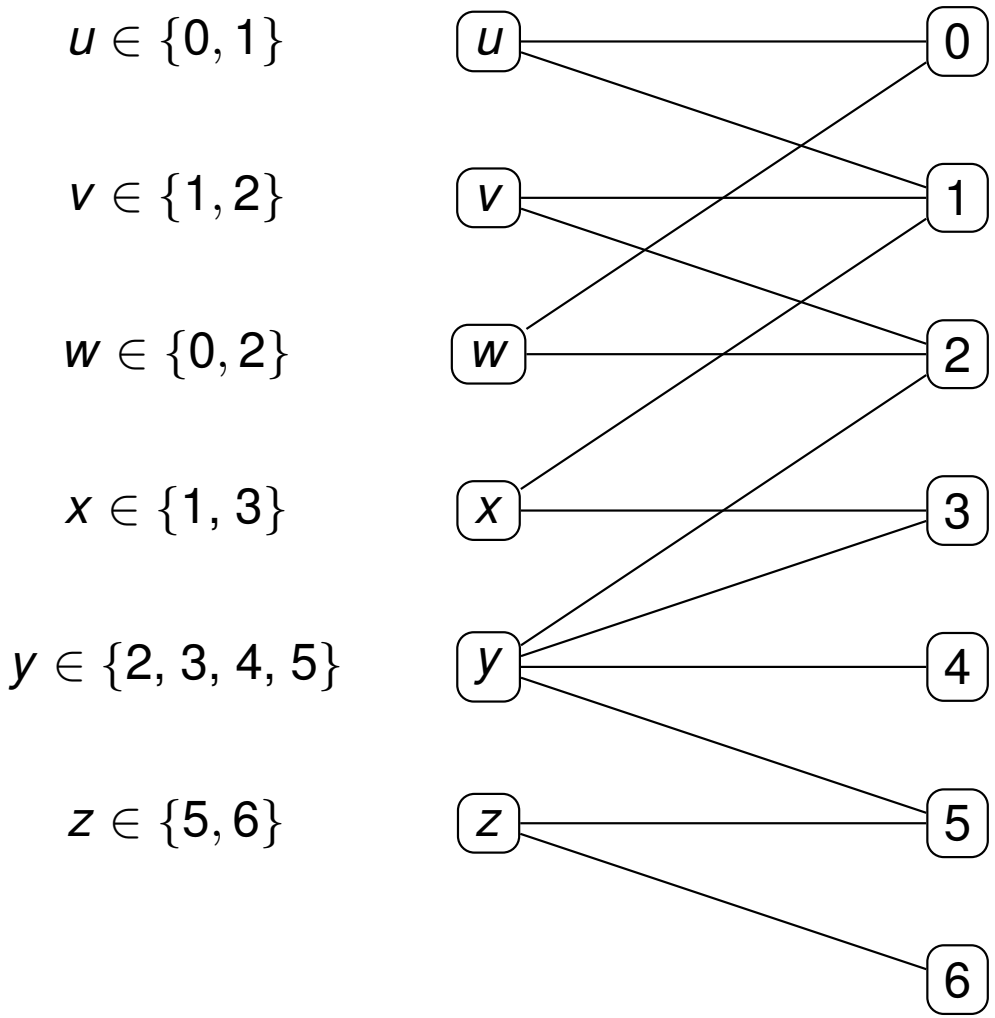
$$a \in \{4, 5\},\ b \in \{4, 5\},\ c \in \{3, \cancel{4}\},\ d \in \{1, 2, \cancel{3, 4, 5}\}$$

No domain pruning by (2). But perfect propagation by (1)

UPPSALA UNIVERSITET

Propagator for AllDifferent

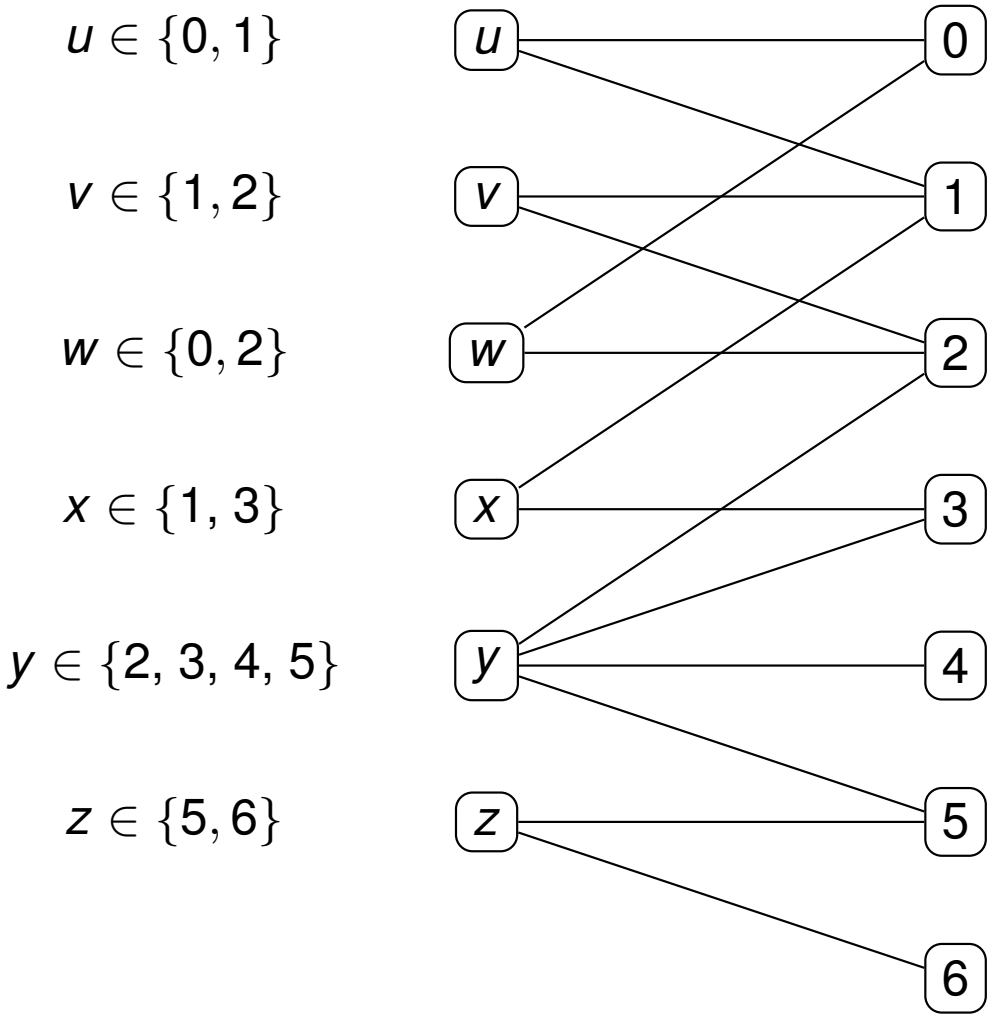
Solutions to AllDifferent($[u, v, w, x, y, z]$) correspond to maximum matchings in a bipartite graph for the domains:



UPPSALA UNIVERSITET

Propagator for AllDifferent

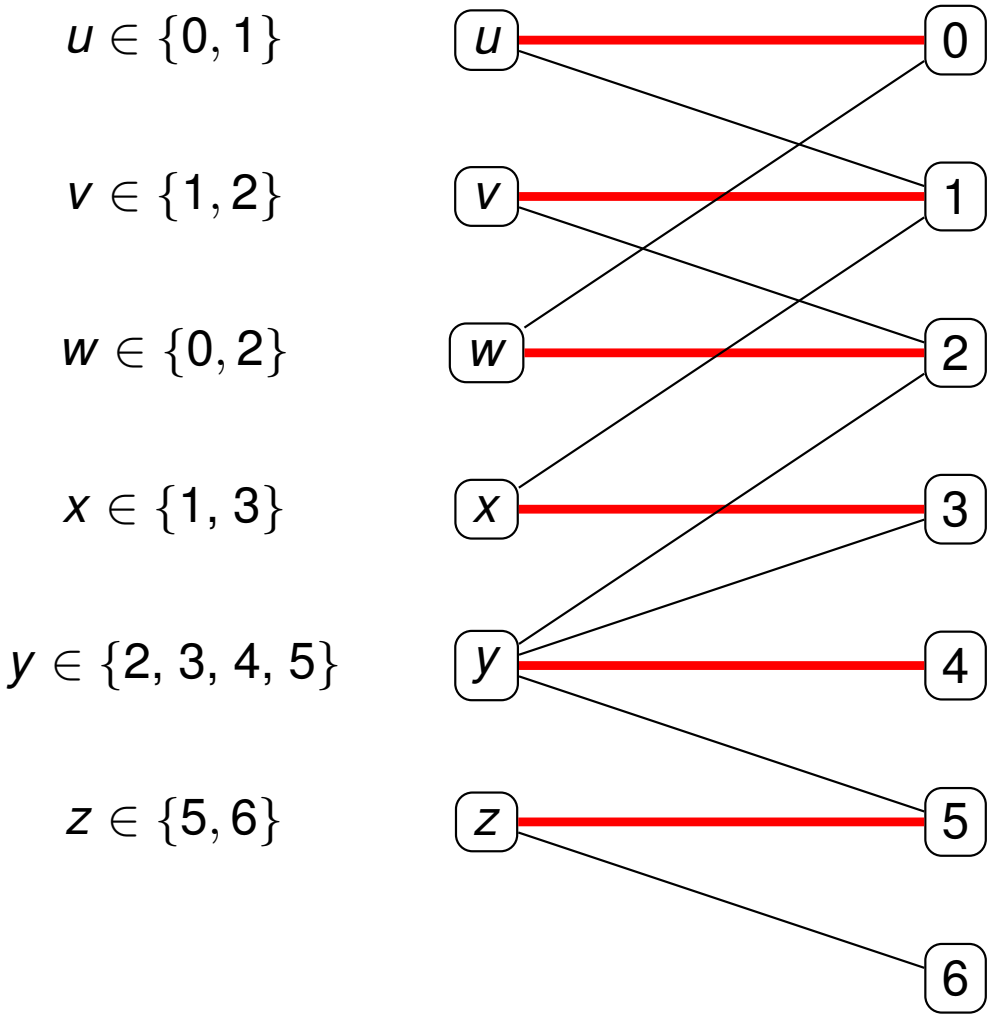
Mark all edges of *some* maximum matching; Hopcroft-Karp algorithm takes $O(m\sqrt{n})$ time for n variables and m values:



UPPSALA UNIVERSITET

Propagator for AllDifferent

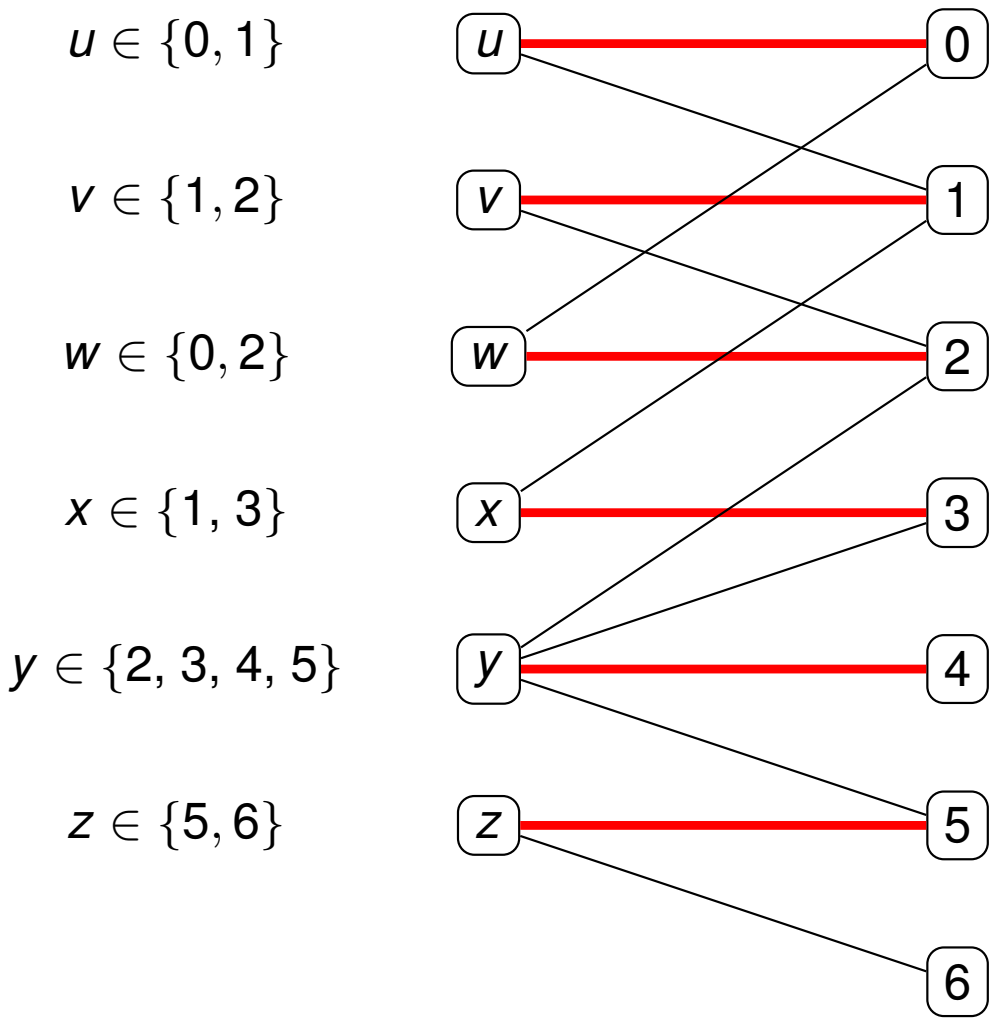
Mark all edges of *some* maximum matching; Hopcroft-Karp algorithm takes $O(m\sqrt{n})$ time for n variables and m values:



UPPSALA UNIVERSITET

Propagator for AllDifferent

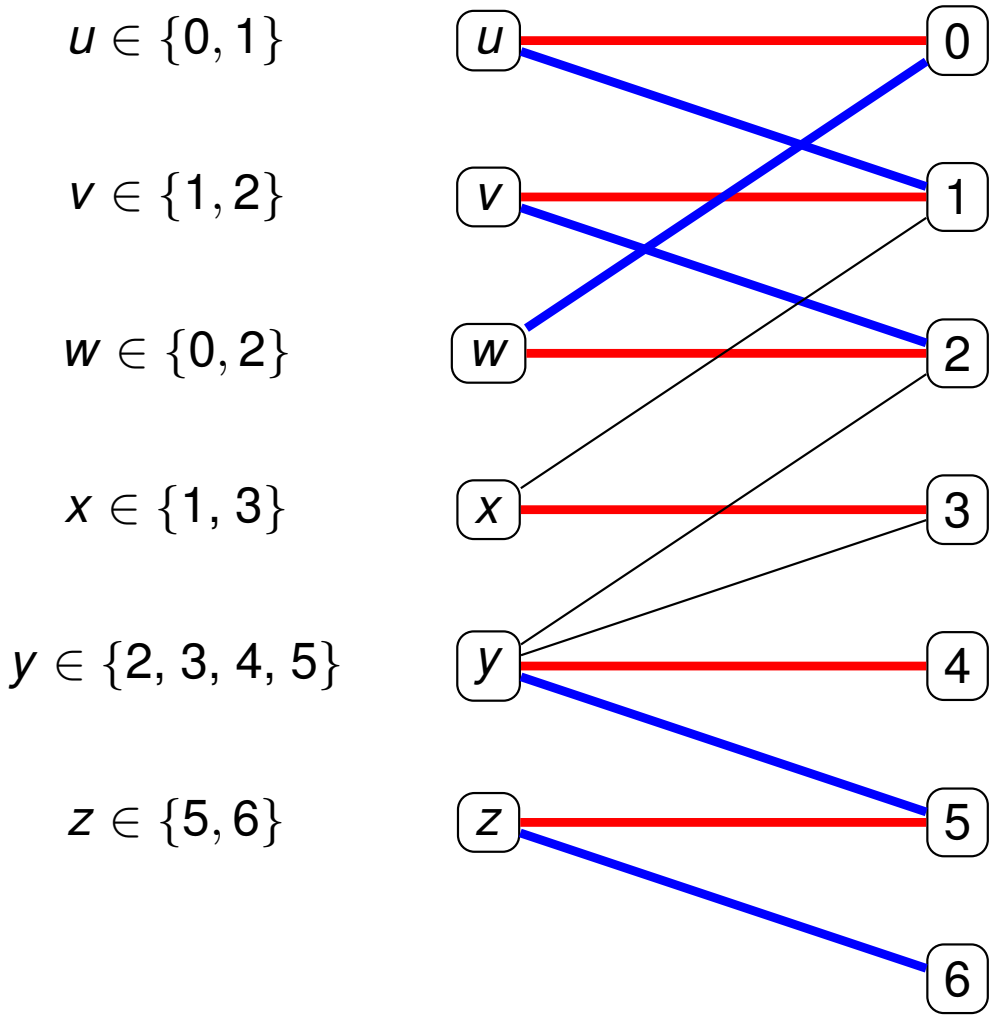
Mark all other edges in *all* other maximum matchings.



UPPSALA UNIVERSITET

Propagator for AllDifferent

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

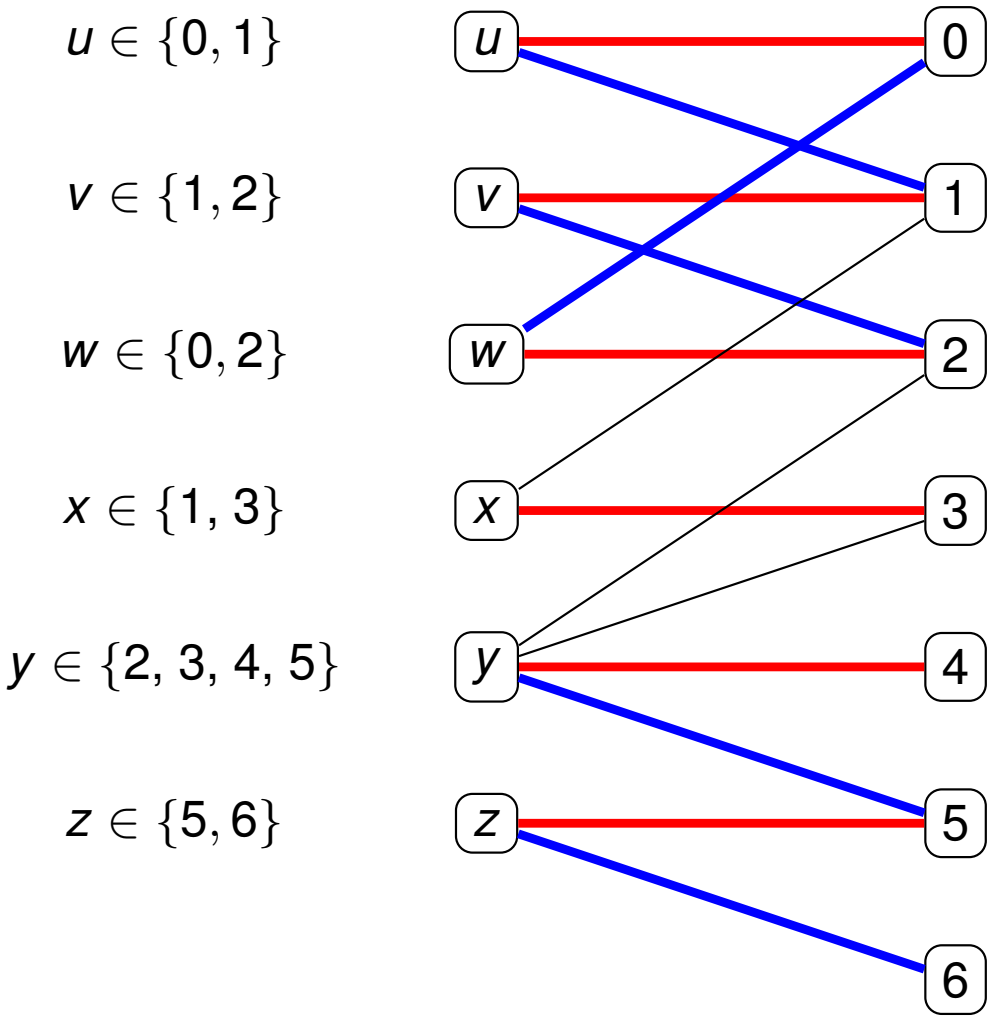


UPPSALA UNIVERSITET

Propagator for AllDifferent

Every still unmarked edge is in *no* maximum matching.

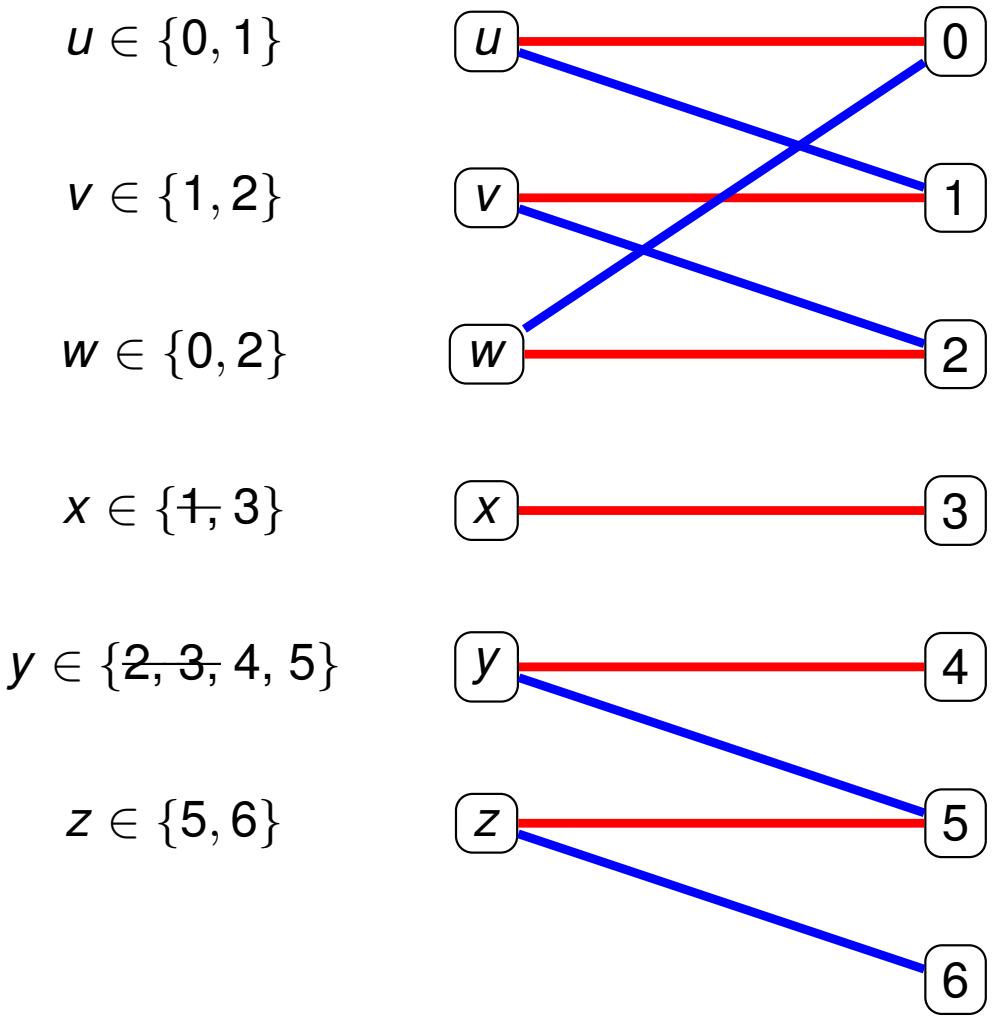
Propagate accordingly within the current domains:



Propagator for AllDifferent

Every still unmarked edge is in *no* maximum matching.

Propagate accordingly within the current domains:



UPPSALA UNIVERSITET

Search + Propagation in the PT example

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	✓	✓	✓	✗	✗	✗	✗
2003	✓	✗	✗	✓	✓	✗	✗
2000	✓	✗	✗	✗	✗	✓	✓
1994	✗	✓	✗	✓	✗	✓	✗
1977	?						
1970							
1966							

UPPSALA UNIVERSITET

Search + Propagation in the PT example

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	✓	✓	✓	✗	✗	✗	✗
2003	✓	✗	✗	✓	✓	✗	✗
2000	✓	✗	✗	✗	✗	✓	✓
1994	✗	✓	✗	✓	✗	✓	✗
1977	?						
1970							
1966							

Alva cannot be a judge of 1977 as that would violate the second constraint (every judge evaluates 3 wines).

UPPSALA UNIVERSITET

Search + Propagation in the PT example

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	✓	✓	✓	✗	✗	✗	✗
2003	✓	✗	✗	✓	✓	✗	✗
2000	✓	✗	✗	✗	✗	✓	✓
1994	✗	✓	✗	✓	✗	✓	✗
1977	✗						
1970							
1966							

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

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	✓	✓	✓	✗	✗	✗	✗
2003	✓	✗	✗	✓	✓	✗	✗
2000	✓	✗	✗	✗	✗	✓	✓
1994	✗	✓	✗	✓	✗	✓	✗
1977	✗						
1970							
1966							

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!

UPPSALA UNIVERSITET

Search + Propagation in the PT example

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	✓	✓	✓	✗	✗	✗	✗
2003	✓	✗	✗	✓	✓	✗	✗
2000	✓	✗	✗	✗	✗	✓	✓
1994	✗	✓	✗	✓	✗	✓	✗
1977	✗						
1970	✗						
1966	✗						

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!

UPPSALA UNIVERSITET

PT: partial assignment

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	✓	✓	✓	✗	✗	✗	✗
2003	✓	✗	✗	✓	✓	✗	✗
2000	✓	✗	✗	✗	✗	✓	✓
1994	✗	✓	✗	✓	✗	✓	✗
1977	✗	?					
1970	✗						
1966	✗						

UPPSALA UNIVERSITET

PT: partial assignment

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	✓	✓	✓	✗	✗	✗	✗
2003	✓	✗	✗	✓	✓	✗	✗
2000	✓	✗	✗	✗	✗	✓	✓
1994	✗	✓	✗	✓	✗	✓	✗
1977	✗	?					
1970	✗						
1966	✗						

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

UPPSALA UNIVERSITET

PT: partial assignment

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	✓	✓	✓	✗	✗	✗	✗
2003	✓	✗	✗	✓	✓	✗	✗
2000	✓	✗	✗	✗	✗	✓	✓
1994	✗	✓	✗	✓	✗	✓	✗
1977	✗	✓					
1970	✗						
1966	✗						

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

UPPSALA UNIVERSITET

PT: partial assignment

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	✓	✓	✓	✗	✗	✗	✗
2003	✓	✗	✗	✓	✓	✗	✗
2000	✓	✗	✗	✗	✗	✓	✓
1994	✗	✓	✗	✓	✗	✓	✗
1977	✗	✓					
1970	✗						
1966	✗						

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.

UPPSALA UNIVERSITET

PT: partial assignment

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	✓	✓	✓	✗	✗	✗	✗
2003	✓	✗	✗	✓	✓	✗	✗
2000	✓	✗	✗	✗	✗	✓	✓
1994	✗	✓	✗	✓	✗	✓	✗
1977	✗	✓					
1970	✗	✗					
1966	✗	✗					

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.

UPPSALA UNIVERSITET

PT: partial assignment

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	✓	✓	✓	✗	✗	✗	✗
2003	✓	✗	✗	✓	✓	✗	✗
2000	✓	✗	✗	✗	✗	✓	✓
1994	✗	✓	✗	✓	✗	✓	✗
1977	✗	✓					
1970	✗	✗					
1966	✗	✗					

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.

UPPSALA UNIVERSITET

PT: partial assignment

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	✓	✓	✓	✗	✗	✗	✗
2003	✓	✗	✗	✓	✓	✗	✗
2000	✓	✗	✗	✗	✗	✓	✓
1994	✗	✓	✗	✓	✗	✓	✗
1977	✗	✓	✗	✗		✗	
1970	✗	✗					
1966	✗	✗					

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.

UPPSALA UNIVERSITET

PT: partial assignment

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	✓	✓	✓	✗	✗	✗	✗
2003	✓	✗	✗	✓	✓	✗	✗
2000	✓	✗	✗	✗	✗	✓	✓
1994	✗	✓	✗	✓	✗	✓	✗
1977	✗	✓	✗	✗		✗	
1970	✗	✗					
1966	✗	✗					

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.

UPPSALA UNIVERSITET

PT: partial assignment

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	✓	✓	✓	✗	✗	✗	✗
2003	✓	✗	✗	✓	✓	✗	✗
2000	✓	✗	✗	✗	✗	✓	✓
1994	✗	✓	✗	✓	✗	✓	✗
1977	✗	✓	✗	✗	✓	✗	✓
1970	✗	✗					
1966	✗	✗					

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.

UPPSALA UNIVERSITET

PT: partial assignment

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	✓	✓	✓	✗	✗	✗	✗
2003	✓	✗	✗	✓	✓	✗	✗
2000	✓	✗	✗	✗	✗	✓	✓
1994	✗	✓	✗	✓	✗	✓	✗
1977	✗	✓	✗	✗	✓	✗	✓
1970	✗	✗					
1966	✗	✗					

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.

UPPSALA UNIVERSITET

PT: partial assignment

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	✓	✓	✓	✗	✗	✗	✗
2003	✓	✗	✗	✓	✓	✗	✗
2000	✓	✗	✗	✗	✗	✓	✓
1994	✗	✓	✗	✓	✗	✓	✗
1977	✗	✓	✗	✗	✓	✗	✓
1970	✗	✗	✓				
1966	✗	✗	✓				

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.

UPPSALA UNIVERSITET

PT: partial assignment

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	✓	✓	✓	✗	✗	✗	✗
2003	✓	✗	✗	✓	✓	✗	✗
2000	✓	✗	✗	✗	✗	✓	✓
1994	✗	✓	✗	✓	✗	✓	✗
1977	✗	✓	✗	✗	✓	✗	✓
1970	✗	✗	✓				
1966	✗	✗	✓				

Common fixpoint reached: No more propagation possible.

UPPSALA UNIVERSITET

PT: partial assignment

	Alva	Dan	Eva	Jim	Leo	Mia	Ulla
2011	✓	✓	✓	✗	✗	✗	✗
2003	✓	✗	✗	✓	✓	✗	✗
2000	✓	✗	✗	✗	✗	✓	✓
1994	✗	✓	✗	✓	✗	✓	✗
1977	✗	✓	✗	✗	✓	✗	✓
1970	✗	✗	✓	✓			
1966	✗	✗	✓				

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

Propagation: etc.

UPPSALA UNIVERSITET

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)

```
1 solve :: int_search(PT,input_order,indomain_max,complete)
2         satisfy;
```

UPPSALA UNIVERSITET

History of CP

Stand-alone languages:

- ALICE by Jean-Louis Laurière, 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; . . .

UPPSALA UNIVERSITET

Outline

UPPSALA UNIVERSITET

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`

UPPSALA UNIVERSITET

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

UPPSALA UNIVERSITET

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, . . .

UPPSALA UNIVERSITET

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, . . .

UPPSALA UNIVERSITET

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, ...

UPPSALA UNIVERSITET

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, ...

UPPSALA UNIVERSITET

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,. . .

UPPSALA UNIVERSITET

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), . . .

UPPSALA UNIVERSITET

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, . . .

UPPSALA UNIVERSITET

MiniZinc Challenge 2015: Some Results

Winners per model (no portfolio or parallel categories):

Model	Winner	Technology
costas array	Mistral	CP
capacited VRP	iZplus	hybrid
gfd schedule	Chuffed	LCG
grid colouring	MinisatID	hybrid
instruction scheduling	Chuffed	LCG
large scheduling	OR-Tools	CP
application mapping	JaCoP	CP
multi-knapsack	CPLEX	MIP
portfolio design	OscaR/CBLS	LS
open stacks	Chuffed	LCG
project planning	Chuffed	LCG
radiation	Gurobi	MIP
satellite management	Gurobi	MIP
time dependent TSP	G12/FD	CP
zephyrus configuration	CPLEX	MIP

UPPSALA UNIVERSITET

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

UPPSALA UNIVERSITET

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

UPPSALA UNIVERSITET

Example: AllDifferent

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

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

instead of:

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

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

- More concise and clear model
- Better solving, even for non-CP solvers

Decompositions of AllDifferent

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

Default Decomposition

```
predicate alldifferent(array[int] of var int: x) =
    forall(i,j in index_set(x) where i < j)
        ( x[i] != x[j] );
```

Used, e.g., by the SMT backend `fzn2smt`

Built-in Predicate

```
predicate alldifferent(array[int] of var int: x);
```

Most CP solvers use a dedicated propagator.

UPPSALA UNIVERSITET

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;
```

UPPSALA UNIVERSITET

Decompositions

Boolean Decomposition for SAT: Ladder Encoding

```
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])
     /\
     (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) =
    [...]
```

UPPSALA UNIVERSITET

Decompositions

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

UPPSALA UNIVERSITET

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.

UPPSALA UNIVERSITET

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.

UPPSALA UNIVERSITET

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.

UPPSALA UNIVERSITET

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

UPPSALA UNIVERSITET

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)

UPPSALA UNIVERSITET

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.

UPPSALA UNIVERSITET

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.

UPPSALA UNIVERSITET

Outline

UPPSALA UNIVERSITET

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].

UPPSALA UNIVERSITET

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.

UPPSALA UNIVERSITET

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.

- EQUAL(s_1, s_2) if s_1 and s_2 are equal, that is $s_1 = s_2$
- REVERSE(s_1, s_2) if $s_1 = c_1 c_2 \cdots c_n$ and $s_2 = c_n \cdots c_2 c_1$
- CONCAT(s_1, s_2, s) if $s_1 \oplus s_2 = s$, with concatenation $\oplus$
- SUBSTRING(s_1, i_1, i_2, s) if $s_1[i_1 : i_2] = s$
- CHARACTERAT(s, i, c) if SUBSTRING$(s, i, i, \text{“}c\text{”})$
- LENGTH(s, i) if s has i characters, that is $|s| = i$
- REGULAR$(s, \mathcal{R})$ if s is a word of a regular language $\mathcal{R}$, given by a regular expression or a finite automaton
- CONTEXTFREE$(s, \mathcal{F})$ if s is a word of a context-free language $\mathcal{F}$, given by a context-free grammar
- COUNT$(s, [c_1, ..., c_n], [i_1, ..., i_n])$ if in s all c_j occur i_j times

UPPSALA UNIVERSITET

Constraints and Decision Variables

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.

UPPSALA UNIVERSITET

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.

UPPSALA UNIVERSITET

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.

UPPSALA UNIVERSITET

Bounded-length Strings in CP

bounded-length sequence representation

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

$$\langle\langle \mathcal{A}[1], \ldots, \mathcal{A}[b]\rangle, \mathcal{N}\rangle$$

Each $\mathcal{A}[i]$ is a set of characters. $\mathcal{N}$ 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.

UPPSALA UNIVERSITET

Implementation 1: Aggregate Strings

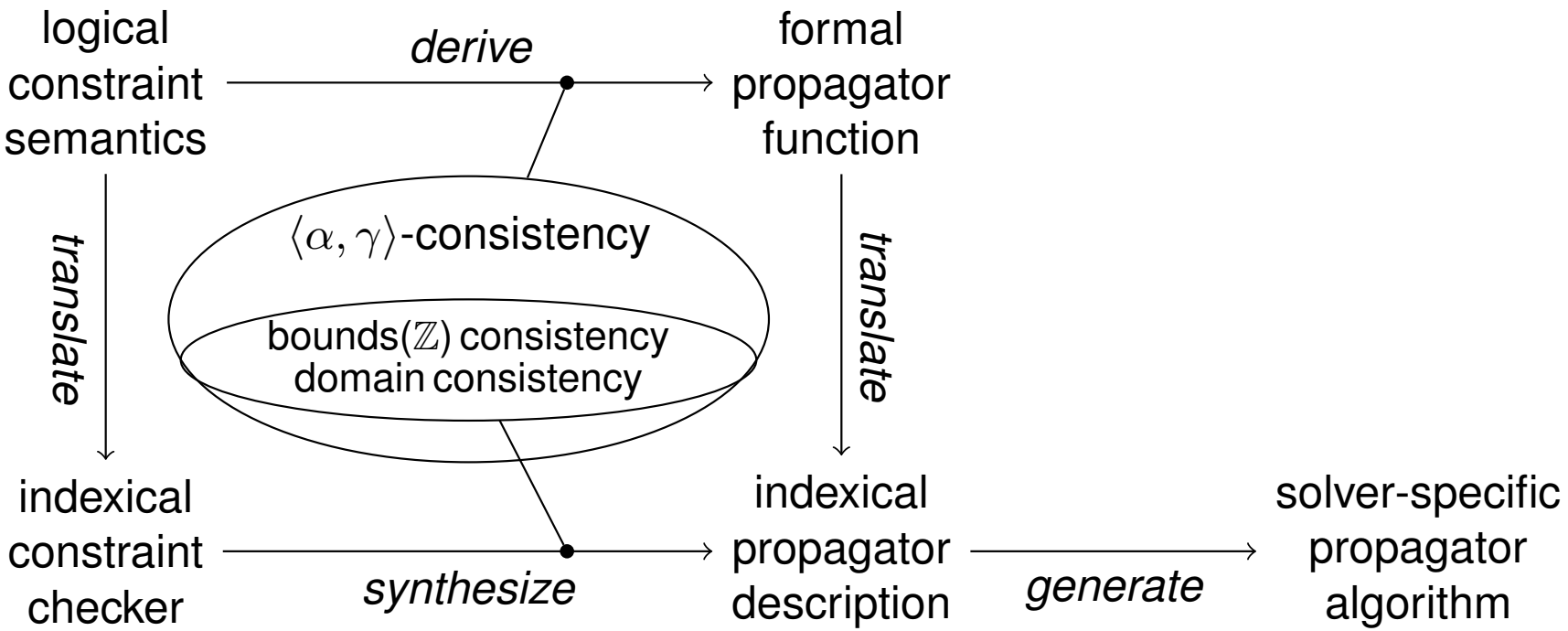
- Idea: represent the components as an aggregation of pre-existing variable types (e.g., integers, sets, Booleans, etc)
- For example, a b-length sequence can be represented by:

$$\langle\langle \mathcal{A}[1], \ldots, \mathcal{A}[b]\rangle, \mathcal{N}\rangle \simeq \langle\langle D(X_1), \ldots, D(X_b)\rangle, D(N)\rangle$$

- Complications
  - undefinedness: empty variable domains $\Leftrightarrow$ failure!
  - representation invariant: implemented as an additional constraint

Implementation 1: Aggregate Strings

Aggregate representation easy to implement in existing CP solvers



Code generation and propagator synthesis:

[Monette et al @ CP'12] and [Monette et al @ CP'15]

UPPSALA UNIVERSITET

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

UPPSALA UNIVERSITET

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

UPPSALA UNIVERSITET

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

UPPSALA UNIVERSITET

Outline

UPPSALA UNIVERSITET

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 Melborne]

UPPSALA UNIVERSITET

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.

UPPSALA UNIVERSITET

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?

UPPSALA UNIVERSITET

References

To read about Strings and Galois Connections see:

J. Scott. Other Things Besides Number: Abstraction, Constraint Propagation, and String Variable Types. PhD thesis, Department of Information Technology, Uppsala University, Sweden, March 2016.
`http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-273311`

The CFG constraint:

J. He. Constraints for Membership in Formal Languages under Systematic Search and Stochastic Local Search. PhD thesis, Department of Information Technology, Uppsala University, Sweden, 2013
`http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-196347`

UPPSALA UNIVERSITET

References

- 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`