FoPra: Implementation and Evaluation of Advanced Propagation Algorithms for Global Constraints

Patrick Pekczynski

Supervisor: Dipl.-Inf. Guido Tack

Programming Systems Lab
Department of Computer Science
Saarland University, Saarbrücken

17.02.2005

Constraint Programming in a nutshell

Constraint Programming (CP)

1. has emerged from artificial intelligence
2. basic idea: combine
    - existing search-methods (backtracking, branch-and-bound, dfs, ...)
    - constraint propagation techniques
3. CP is applied in widespread areas:
    - natural language processing
    - artificial intelligence
    - operations research
    - genome sequencing
    - combinatorial optimization
    - computer algebra
    - electrical engineering
    - ...

⇒ CP is a method for modeling and solving many types of problems

CP in a nutshell ctd.

Definition (Constraint Satisfaction Problem)

CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ with:

$\mathcal{X}$	$= \{x_1, \ldots, x_n\}$	set of variables
$\mathcal{D}$	$= \{D_1, \ldots, D_n\}$	respective domains of the variables
$\mathcal{C}$	$= \{c_1, \ldots, c_m\}$	constraints on subsequences of variables
	$c_i \subseteq D_1 \times \cdots \times D_k$	

Constraint Programming

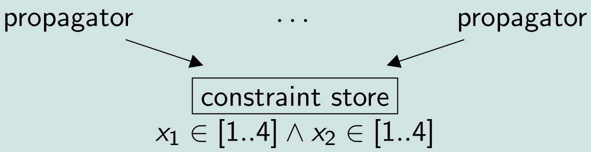
1. alternative approach to programming
2. programming = generation of requirements (**constraints**)
3. about formulating (**modeling**) and **solving** of CSPs
4. important aspect for solving of CSPs: **constraint propagation**

What is Constraint Propagation?

Constraint Propagation

- process of reducing a given CSP to an equivalent but simpler CSP
- reducing search space of CSP while maintaining equivalence
- pruning the domains of the variables in a given CSP
- propagators implement constraints on CSP
- essential component of a computation space

Computation Space



propagator . . . propagator

constraint store

$x_1 \in [1..4] \wedge x_2 \in [1..4]$

What is Constraint Propagation? - ctd.

Constraint Propagation

- inference component Prop
- implement the constraints $c_j \in \mathcal{C}$ of a given CSP
- narrow a variable domain $D_i \in \mathcal{D}$ until
  1. failure
     $Prop(D_i) = \bot$
     $\Rightarrow$ CSP inconsistent (no solution possible)
  2. entailment
     $\forall D_j : D_j \subseteq D_i : Prop(D_j) = D_j$
     $\Rightarrow$ constraint already fulfilled by variable domains
  3. success
     $Prop(D_i) = Prop(Prop(D_i))$
     $\Rightarrow$ CSP consistent (propagation reaches a fixpoint)
- variables x_i only common communication channel between propagators

FoPra task

Goals

1. implementation of advanced propagation algorithms for global constraints:
    - *Sortedness*
    - *PermSort*
    - *Global Cardinality*
2. use and evaluation of **staged propagation** as novel scheduling technique for propagators
3. comparsion to other implementations of the same constraints

FoPra task

Framework

- generic
- constraint
- development
- environment

[**Gecode**, a C++ library for constraint programming.]
More information soon available on: http://www.gecode.org

Developers

- **Dr. Christian Schulte** (KTH, Stockholm, Sweden)
- **Guido Tack** (PS Lab, Saabrücken, Germany)

Sortedness-constraint

Definition (*Sortedness*)

$Sortedness(x_1, \ldots, x_n ; y_1, \ldots, y_n)$

1. input: 2 sequences of n variables x_i and y_i
2. output: Is 2^{nd} sequence obtained by sorting 1^{st} in non-decreasing order?

Example (*Sortedness*)

Sortedness		
$Sortedness(1, 3, 1; 1, 1, 3)$	holds	✓
$Sortedness(5, 2, 3; 3, 2, 5)$	violated	↯

▸ details

PermSort-constraint

Definition (*PermSort*)

$PermSort(x_1, \ldots, x_n \,;\, y_1, \ldots, y_n \,;\, p_1, \ldots, p_n)$

1. input: 3 sequences of n variables x_i, y_i and p_i
2. output: Is 2^{nd} sequence obtained by sorting 1^{st} in non-decreasing order AND is 3^{rd} sequence a permutation of $(1, \ldots, n)$, s.t.:
   - $\forall i \in \{1, \ldots, n\} : x_i = y_{p_i}$
3. equals *Sortedness* with additional permutation variables

Example (*PermSort*)

PermSort		
$PermSort(1,3,1;1,1,3;1,3,2)$	holds	✓
$PermSort(5,2,3;2,3,5;1,2,3)$	violated	↯

▸ details

Global Cardinality -constraint

Definition (*Global Cardinality*)

$GCC(x_1, \ldots, x_n \,;\, l_1, \ldots, l_d \,;\, u_1, \ldots, u_d)$

1. input: a sequence of n variables x_i, defined on a set of values $D = \{v_1, \ldots, v_d\}$ and for each value v_i a pair $[l_i, u_i]$
2. output: Is it possible to narrow the domains of the variables x_i, s.t.: $\forall i \in \{1, \ldots, d = |D|\} \; \forall v_i \in D : l_i \leq \#v_i \leq u_i$?
3. generalization of the *Alldifferent* constraint:
    - $Alldifferent(x_1, \ldots, x_n) = GCC(x_1, \ldots, x_n, \,;\, l_1, \ldots, l_d \,;\, u_1, \ldots, u_d)$ where $\forall i \in \{1, \ldots, d\} : l_i = 0 \wedge u_i = 1$

Example (*Global Cardinality*)

Global Cardinality	
$GCC(2, [1..2], [2..3], [2..3], [1..4], [3..4] \,;\, 1, 1, 1, 2 \,;\, 3, 3, 3, 3)$	holds [✓]
$GCC(2, [1..2], [2..3], [2..3], [1..4], [3..4] \,;\, 1, 1, 1, 1 \,;\, 1, 1, 1, 1)$	violated [↯]

Bounds Consistent Algorithm for *Global Cardinality*

Paper

An Efficient Bounds Consistency Algorithm for the *Global Cardinality* Constraint, van Beek et. al. [qui03]

Algorithm

3 Steps: $(n = |X|,\ t = \text{sorting time})$

1. sort X variables according to lower and upper bounds $[O(t)]$
2. UBC: value v_i occurs at most u_i times $[O(n \cdot log(n))]$
3. LBC: value v_i occurs at least l_i times $[O(n \cdot log(n))]$

$\Rightarrow$ complexity $O(t + n \cdot log(n))$

▸ consistency levels

The theory behind [Philip Hall ⋆1904 - †1982]

Theorem (Hall's Theorem on bipartite perfect matching)

For any node set S and its set of neighbors $N(S)$:

$$|S| \leq |N(S)|$$

$\Rightarrow$ *any set S has at least as many neighbors as elements*
(if not $\Rightarrow$ there is no perfect matching)

The theory behind [Philip Hall ⋆1904 - †1982]

Definition (Hall interval / Hall set)

- $S \subseteq \mathcal{D} : C(S) = \#$ variables $v_i : D_i \subseteq S$
- $S \subseteq \mathcal{D} : I(S) = \#$ variables $v_i : D_i \cap S \neq \emptyset$
- $\lceil S \rceil = \sum_{v \in S} u_v$, $\lfloor S \rfloor = \sum_{v \in S} l_v$
- **Hall interval** $= H \subseteq \mathcal{D} : |H| = C(H)$
- **Hall set** $= H \subseteq \mathcal{D} : \lceil H \rceil = C(H)$

Extension to GCC

- extend Hall interval to Hall sets
- Hall proved: *Alldifferent* satisfiable $\Leftrightarrow \forall S : C(S) \leq |S|$
- UBC($\{1, 2, 3, 4\}$,$(3, 3, 3, 3)$) $\Leftrightarrow$ AllDifferent($1a, 1b, 1c, 2a, 2b, 2c, 3a, 3b, 3c, 4a, 4b, 4c$)
- UBC satisfiable $\Leftrightarrow \forall S : C(S) \leq \lceil S \rceil$

The theory behind [Philip Hall $\star$1904 - †1982]

Definition (Hall interval / Hall set)

- $S \subseteq \mathcal{D} : C(S) = \#$ variables $v_i : D_i \subseteq S$
- $S \subseteq \mathcal{D} : I(S) = \#$ variables $v_i : D_i \cap S \neq \emptyset$
- $\lceil S \rceil = \sum_{v \in S} u_v$, $\lfloor S \rfloor = \sum_{v \in S} l_v$
- **Hall interval** $= H \subseteq \mathcal{D} : |H| = C(H)$
- **Hall set** $= H \subseteq \mathcal{D} : \lceil H \rceil = C(H)$

Extension to GCC

- **Failure set** $F \subseteq \mathcal{D} : I(F) < \lfloor F \rfloor$
- **Unstable set** $U \subseteq \mathcal{D} : I(U) = \lfloor U \rfloor$
- **Stable set** $S \subseteq \mathcal{D} : C(S) > \lfloor S \rfloor \wedge \forall U \, \forall F : \cap U = \emptyset \wedge S \cap F = \emptyset$
- LBC satisfiable $\Leftrightarrow \neg \exists S \in \mathcal{D}$: S failure set

The theory behind [Philip Hall $\star$1904 - †1982]

Definition (Hall interval / Hall set)

- $S \subseteq \mathcal{D} : C(S) = \#$ variables $v_i : D_i \subseteq S$
- $S \subseteq \mathcal{D} : I(S) = \#$ variables $v_i : D_i \cap S \neq \emptyset$
- $\lceil S \rceil = \sum_{v \in S} u_v$, $\lfloor S \rfloor = \sum_{v \in S} l_v$
- **Hall interval** $= H \subseteq \mathcal{D} : |H| = C(H)$
- **Hall set** $= H \subseteq \mathcal{D} : \lceil H \rceil = C(H)$

Extension to GCC

- **Failure set** determines whether an LBC is satisfiable
- **Unstable set** indicates where domains have to be pruned
- **Stable set** indicates which domains must not be pruned

Domain Consistent Algorithm for *Global Cardinality*

Paper

Improved Algorithms for the *Global Cardinality* Constraint,
van Beek et. al. [imp04]

Algorithm

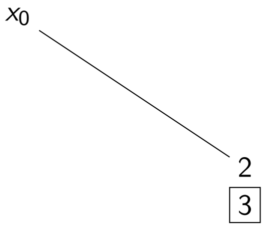
5 Steps: $(n = |X|,\ D = \bigcup_{i\in\{1,\dots,n\}} D_i,\ d = |D|,\ m = \#\ of\ edges)$;

- build a bipartite variable-value graph G=(<X,D>,E) $[O(n \cdot d)]$
- compute a matching in G $[O(log(n) + n \cdot log(n) + n \cdot d)]$
  - UBC: matching (cardinality $|M_u| = |X|$)
  - LBC: matching (cardinality $|M_l| = \sum l_i$)
- compute the SCCs in $[O(n + m)]$
- find even alternating paths starting from free nodes $[O(n + m)]$
- updating the domains according to remaining edges $[O(n \cdot d)]$

$\Rightarrow$ complexity $O(n \cdot log(n) + n \cdot d) < O(n^{\frac{3}{2}} \cdot d)$

Running example: UBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

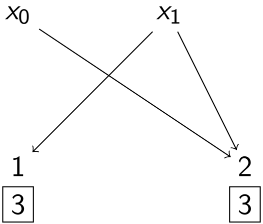


- build the variable-value graph
- each value node has a capacity denoting who often the value node can be matched

▸ LBC ▸ Skip

Running example: UBC

$$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$$

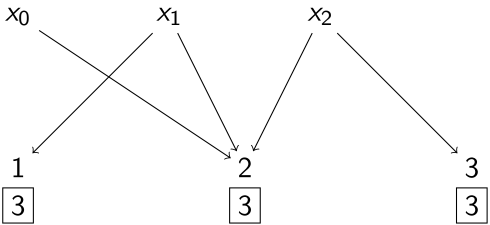


- build the variable-value graph
- each value node has a capacity denoting who often the value node can be matched

▸ LBC ▸ Skip

Running example: UBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

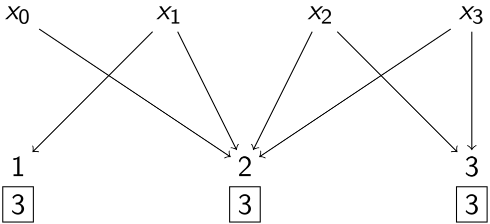


- build the variable-value graph
- each value node has a capacity denoting who often the value node can be matched

▸ LBC ▸ Skip

Running example: UBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

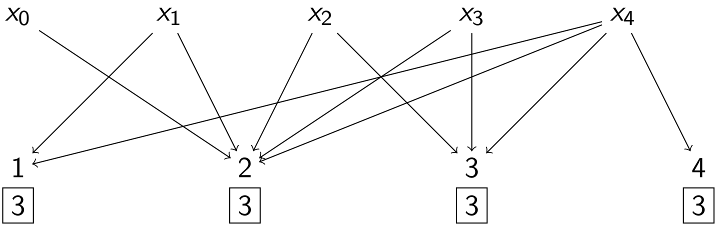


- build the variable-value graph
- each value node has a capacity denoting who often the value node can be matched

▸ LBC ▸ Skip

Running example: UBC

$$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$$

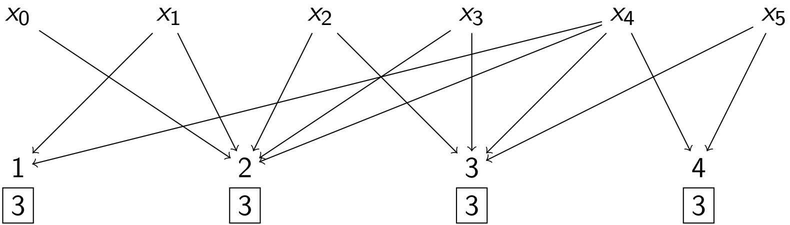


- build the variable-value graph
- each value node has a capacity denoting who often the value node can be matched

▸ LBC ▸ Skip

Running example: UBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

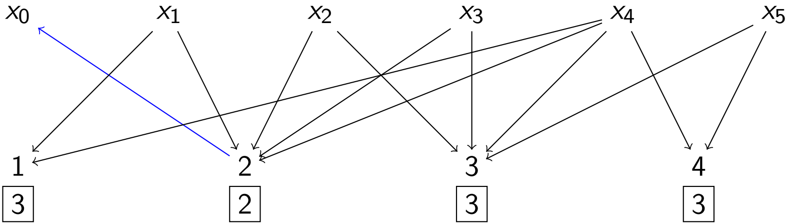


- build the variable-value graph
- each value node has a capacity denoting who often the value node can be matched

▸ LBC ▸ Skip

Running example: UBC

$x_0 = 2 \quad x_1 = [1, 2] \quad x_2 = [2, 3] \quad x_3 = [2, 3] \quad x_4 = [1, 4] \quad x_5 = [3, 4]$

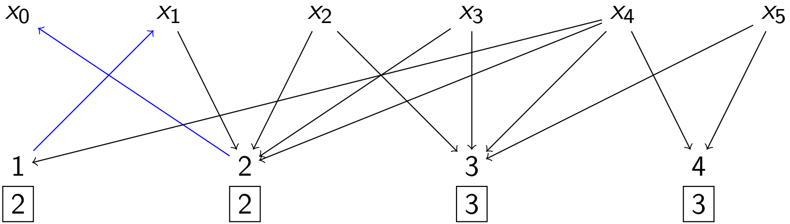


- compute a matching M_u for X on G

▸ LBC ▸ Skip

Running example: UBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

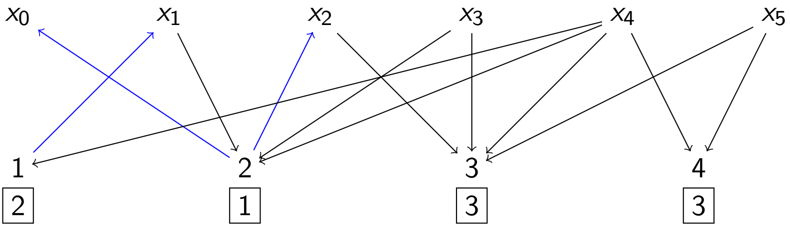


- compute a matching M_u for X on G

▸ LBC ▸ Skip

Running example: UBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

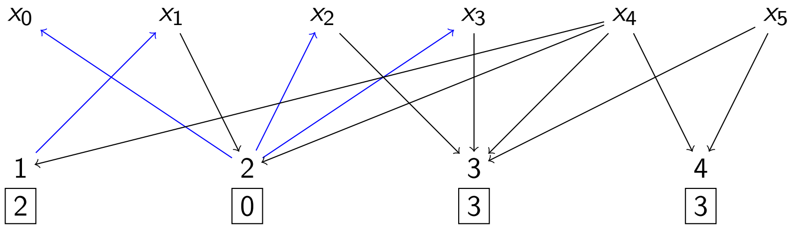


- compute a matching M_u for X on G

▸ LBC ▸ Skip

Running example: UBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

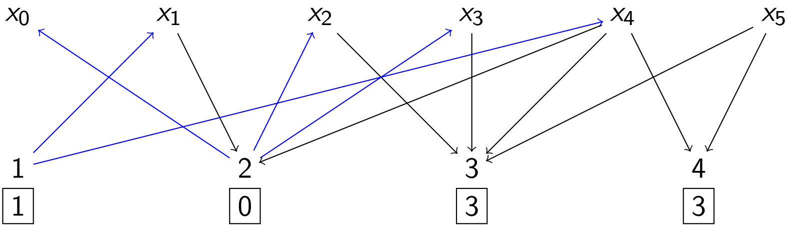


- compute a matching M_u for X on G

▸ LBC ▸ Skip

Running example: UBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

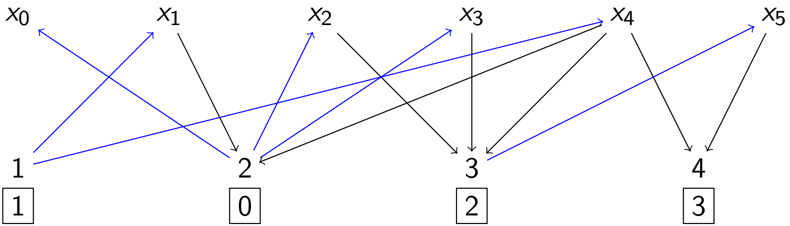


- compute a matching M_u for X on G

▸ LBC ▸ Skip

Running example: UBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

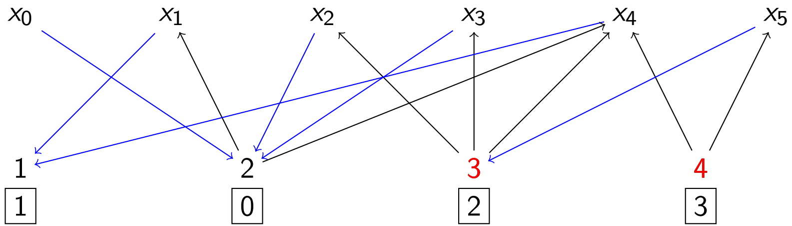


- $|M_u| = |X| \Rightarrow$ satisfies UBC ✓
- no strongly connected components

▸ LBC ▸ Skip

Running example: UBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

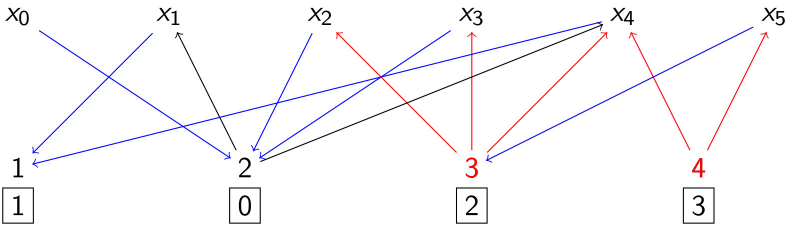


- compute free alternating paths starting from 3 and 4

▸ LBC ▸ Skip

Running example: UBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

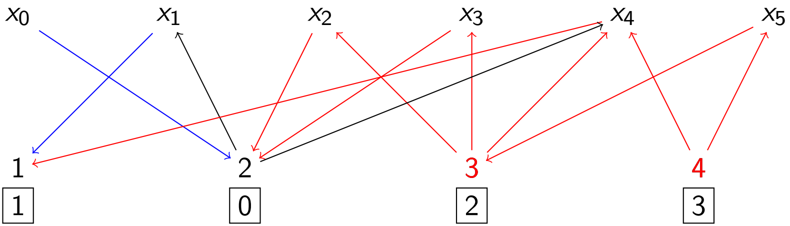


- compute free alternating paths starting from 3 and 4

▸ LBC ▸ Skip

Running example: UBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

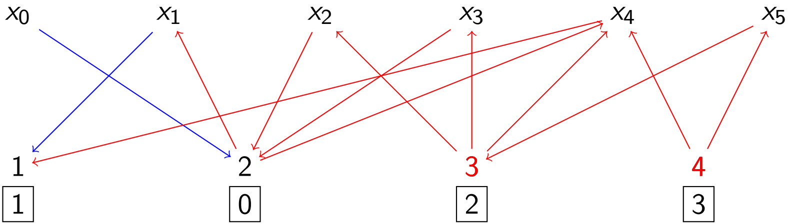


- compute free alternating paths starting from 3 and 4

▸ LBC ▸ Skip

Running example: UBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

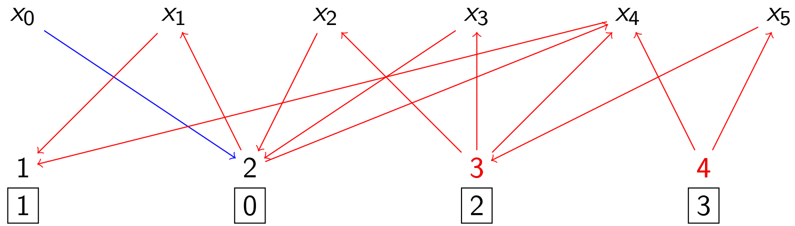


- compute free alternating paths starting from 3 and 4

▸ LBC ▸ Skip

Running example: UBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

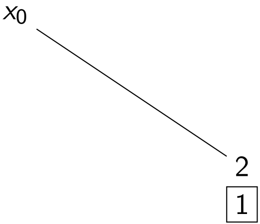


- compute free alternating paths starting from 3 and 4

▸ LBC ▸ Skip

Running example: LBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

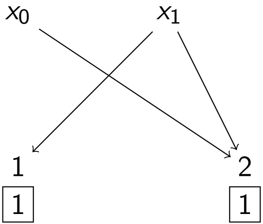


- build the variable-value graph
- each value node has a capacity denoting who often the value node can be matched

▸ UBC

Running example: LBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

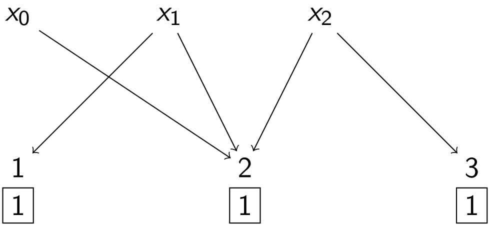


- build the variable-value graph
- each value node has a capacity denoting who often the value node can be matched

▸ UBC

Running example: LBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

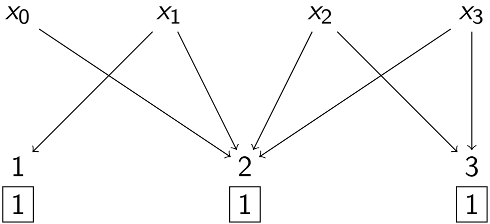


- build the variable-value graph
- each value node has a capacity denoting who often the value node can be matched

▸ UBC

Running example: LBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

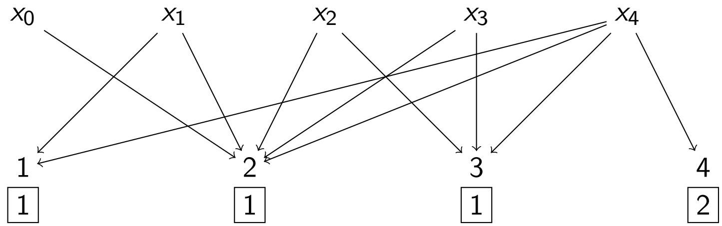


- build the variable-value graph
- each value node has a capacity denoting who often the value node can be matched

▸ UBC

Running example: LBC

$$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$$

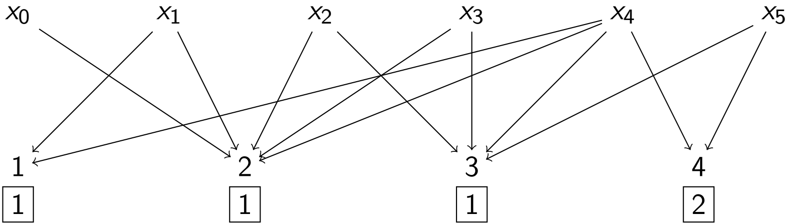


- build the variable-value graph
- each value node has a capacity denoting who often the value node can be matched

▸ UBC

Running example: LBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

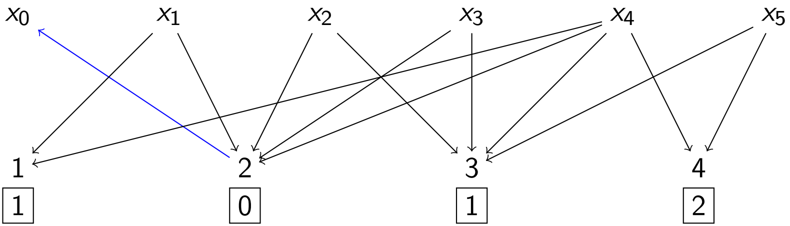


- build the variable-value graph
- each value node has a capacity denoting who often the value node can be matched

▸ UBC

Running example: LBC

$$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$$

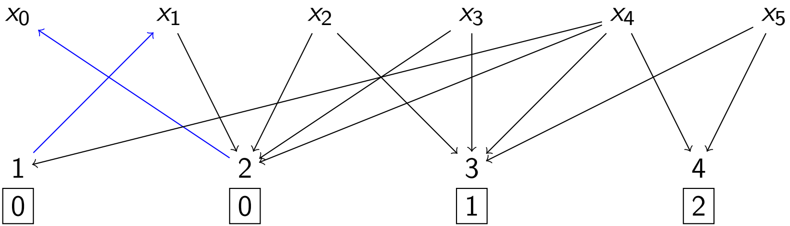


- compute a matching M_l for D on G

▸ UBC

Running example: LBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

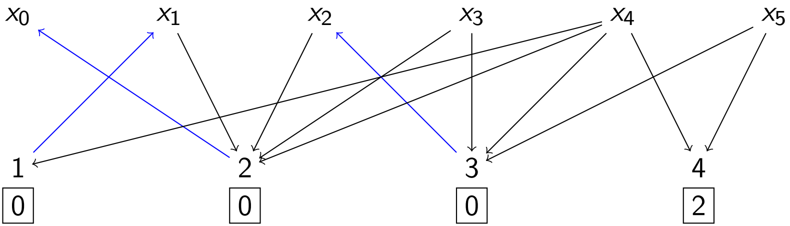


- compute a matching M_l for D on G

UBC

Running example: LBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

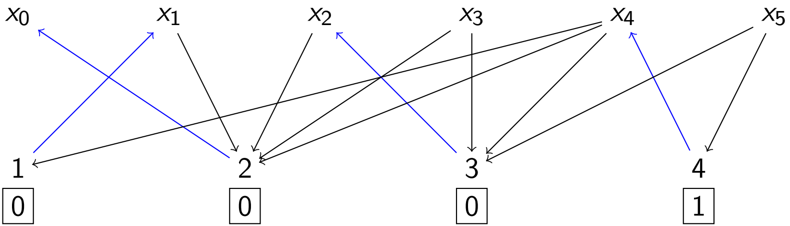


- compute a matching M_l for D on G

▸ UBC

Running example: LBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

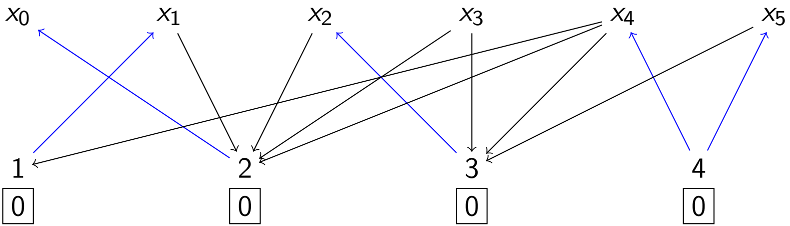


- compute a matching M_l for D on G

▸ UBC

Running example: LBC

$x_0 = 2 \quad x_1 = [1, 2] \quad x_2 = [2, 3] \quad x_3 = [2, 3] \quad x_4 = [1, 4] \quad x_5 = [3, 4]$

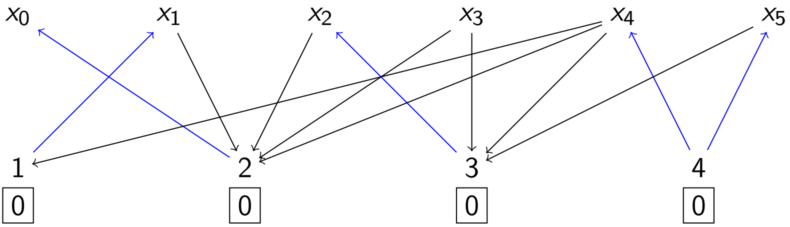


- compute a matching M_l for D on G

▸ UBC

Running example: LBC

$$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$$

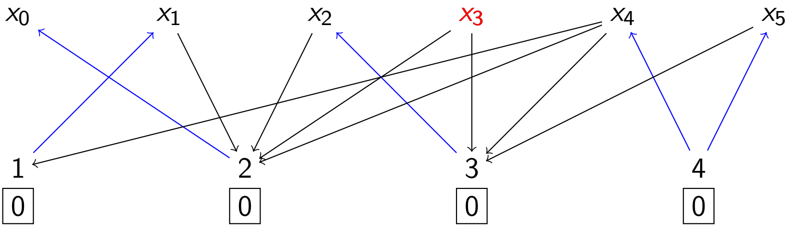


- $|M_l| = \sum l_i \Rightarrow$ satisfies LBC ✓
- no strongly connected components

▸ UBC

Running example: LBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

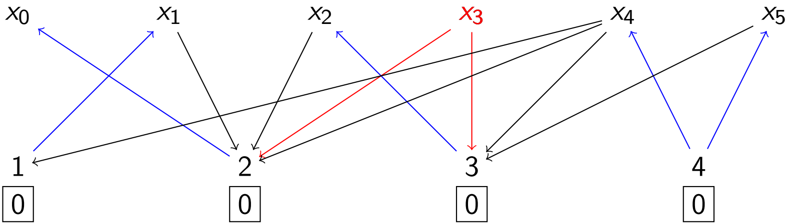


- compute free alternating paths starting from x_3

UBC

Running example: LBC

$$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$$

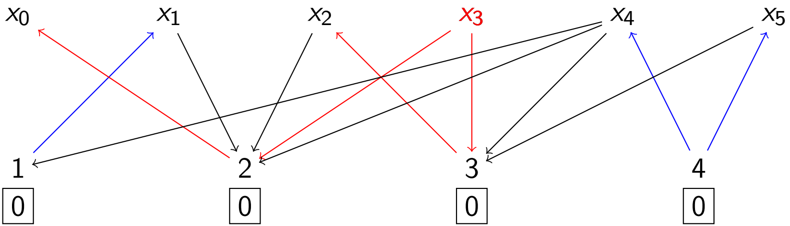


- compute free alternating paths starting from x_3

▸ UBC

Running example: LBC

$$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$$

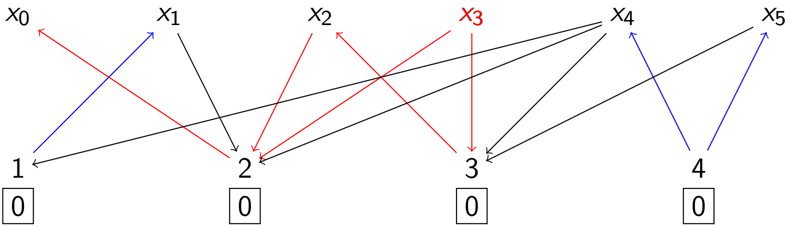


- compute free alternating paths starting from x_3

▸ UBC

Running example: LBC

$x_0 = 2 \quad x_1 = [1,2] \quad x_2 = [2,3] \quad x_3 = [2,3] \quad x_4 = [1,4] \quad x_5 = [3,4]$

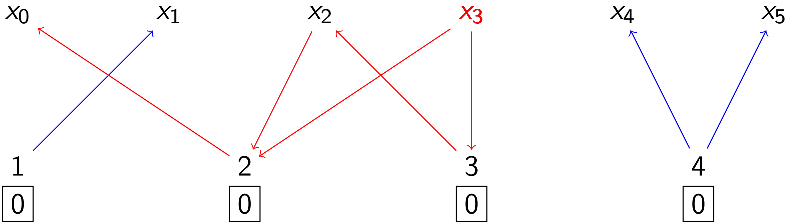


- omit all edges not in M_l, $M_l + p_{aug}$ or in SCCs

▸ UBC

Running example: LBC

$x_0 = 2$ $x_1 = 1$ $x_2 = [2, 3]$ $x_3 = [2, 3]$ $x_4 = 4$ $x_5 = 4$



- omit all edges not in M_l, $M_l + p_{aug}$ or in SCCs

UBC

Staged Propagation

Paper

Speeding Up Constraint Propagation,
Christian Schulte and Peter J. Stuckey, CP 2004 [SS04b]

Gecode

- assigns priorities to propagators, e.g. their respective running times (linear $<$ quadratic $<$ cubic $< \dots$)
- schedule propagators in priority queue
- **event** $=$ domain D_i of x_i is changed to D_i'

fix(x_i)	x_i is fixed	$\lvert D_i' \rvert = 1$
lbc(x_i)	lower domain bound is changed	$inf_{D_i'} > inf_{D_i}$
ubc(x_i)	upper domain bound is changed	$sup_{D_i'} < sup_{D_i}$
dmc(x_i)	domain of x_i is narrowed	$D_i' \subset D_i$

Staged Propagation

Staged Propagation

- novel propagator scheduling technique
- multiple propagators for single constraint, e.g.
    1. G_b :=bounds consistent GCC: $O(n \cdot log(n))$
    2. G_d :=domain consistent GCC: $O(n \cdot log(n) + n \cdot d)$
- combine them into single propagator with internal state (**stage**)
- stage determines propagation:
    1. smallest running time first
    2. event-dependent propagation
        - $\overbrace{lbc \vee ubc \vee fix}^{B} \Rightarrow G_b$
        - $dmc \wedge \neg(B) \Rightarrow G_d$

Sports League Scheduling

Feasible Schedule for N = 8

	Col_1		Col_2		Col_3		Col_4		Col_5		Col_6		Col_7	
$Period_1$	0	1	0	2	1	2	5	7	3	6	3	7	4	5
$Period_2$	4	6	1	5	0	3	0	4	1	7	2	5	2	6
$Period_3$	2	7	3	4	4	7	1	6	0	5	0	6	1	3
$Period_4$	3	5	6	7	5	6	2	3	2	4	1	4	0	7

- search space $= O((\frac{N}{2} \cdot (N-1))!)$ for (i,j) with $0 < i < j < N$
- $\forall i \in \{1, \ldots, N-1\} : Alldifferent\ (Col_i)$
- $\forall i \in \{1, \ldots, \frac{N}{2}\} : GCC(Period_i; 0 \ldots 0; 2 \ldots 2)$
- $\forall (i,j) \in Schedule :$
  $Alldifferent\ (i,j) \Leftrightarrow N \cdot i + j = Unique\ matchup\ number$

Evaluation and running times

$N = 8$ (running times in ms)						
SOL=1	SP	VAL	BND	DOM	VB	ILOG
	+	290			90^*	no SP
	-	40	60	110	100	90

Table: Sports League Scheduling

Branching: smallest minimum(var) smallest value(val)

- SP = staged propagation
- VB = van Beek's BND-GCC implementation in ILOG
- ILOG = ILOG Solver 5.0 IlcCard-constraint
- VB^* = VB uses "value removal" ≈ SP

Evaluation and running times

$N = 8$ (running times in ms)						
SOL=100	SP	VAL	BND	DOM	VB	ILOG
	+	5280			2730*	no SP
	-	1000	1210	12780	3230	2610

Table: Sports League Scheduling

Branching: smallest minimum(var) smallest value(val)

- SP = staged propagation
- VB = van Beek's BND-GCC implementation in ILOG
- ILOG = ILOG Solver 5.0 IlcCard-constraint
- VB* = VB uses "value removal" ≈ SP

Evaluation and running times

$N = 10$ (running times in ms)						
SOL=1	SP	VAL	BND	DOM	VB	ILOG
	+	2620			25130*	no SP
	-	9290	11060	47280	31860	63640

Table: Sports League Scheduling

Branching: smallest minimum(var) smallest value(val)

- SP = staged propagation
- VB = van Beek's BND-GCC implementation in ILOG
- ILOG = ILOG Solver 5.0 IlcCard-constraint
- VB* = VB uses "value removal" ≈ SP

Results

1. Efficient implementation of global propagation algorithms for:
   - *Sortedness* $\Rightarrow BND[O(t+n)]$
   - no guaranteed bounds consistency for *Sortedness*-implementation with permutation variables. $\Rightarrow$ implementation of *PermSort*.
   - *PermSort* $\Rightarrow BND[O(n^2)]$
   - *Global Cardinality*
     - $\Rightarrow BND[O(t + n \cdot log(n))]$
     - $\Rightarrow DOM[O(n \cdot log(n) + n \cdot d)]$
2. evaluation of *Global Cardinality* against ILOG
   - use and evaluation of novel propagator scheduling techniques $\Rightarrow$ **Staged Propagation**

Results

1. Efficient implementation of global propagation algorithms for:
   - *Sortedness* $\Rightarrow BND[O(t+n)]$
   - no guaranteed bounds consistency for *Sortedness*-implementation with permutation variables. ⇒ implementation of *PermSort*.
   - *PermSort* $\Rightarrow BND[O(n^2)]$
   - *Global Cardinality*
     - $\Rightarrow BND[O(t+n \cdot log(n))]$
     - $\Rightarrow DOM[O(n \cdot log(n) + n \cdot d)]$
2. evaluation of *Global Cardinality* against ILOG
   - use and evaluation of novel propagator scheduling techniques ⇒ **Staged Propagation**
   - current benchmarks: **Gecode factor 2 faster than ILOG**

Hot Spots

Time spent on:

1. getting in touch with the Gecode-system
    - variables
    - events
    - propagators
2. reading the references
3. understand and adapt algorithms to the Gecode-system
4. implement datastructures fitting for the algorithms
5. getting in touch with the ILOG-system
6. CSP-models for evaluation against ILOG

Outlook

What can be done?

- add permutation variables with guaranteed bounds consistency to the implementation of the *Sortedness* constraint
- use cardinality variables instead of fixed integers for the upper and lower bounds in the *Global Cardinality* constraint

References I

[apt98] *The Essence of Constraint Propagation*, volume cs.AI/9811024, 1998.

[Apt99] *The Rough Guide to Constraint Propagation*. Springer-Verlag, 1999.

[Apt03] K. Apt.
Principles of Constraint Programming.
2003.

[BGC00] Noëlle Bleuzen-Guernalec and Alain Colmerauer.
Optimal Narrowing of a Block of Sortings in Optimal Time.
Constraints: An International Journal, 5(1/2):85–118m, January 2000.

References II

[HK73] John E. Hopcroft and Richard M. Karp.
An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs.
SIAM: Journal of Computing, 2(4):225–231, December 1973.

[HMPR99] Pascal Van Hentenryck, Laurent Michel, Laurent Perron, and Jean-Charles Régin.
Constraint programming in opl.
In Gopalan Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702 of *Lecture Notes in Computer Science*, pages 98–116, September 29 - October 1 1999.

References III

[imp04] *Improved Algorithms for the Global Cardinality Constraint*, volume 3528, Toronto, Canada, September 2004.

[I.S04] I.S.Laboratory.
SICStus Prolog user's manual, 3.11.1 Technical Report.
Swedish Institute of Computer Science, 2004.
Download PDF-File.

[lop03] *A Fast and Simple Algorithm for Bounds Consistency of the Alldifferent Constraint*, Acapulco, Mexico, August 2003.

References IV

[LOQTvB03] Alejandro López-Ortiz, Claude-Guy Quimper, John Tromp, and Peter van Beek.
A Fast and Simple Algorithm for Bounds Consistency of the Alldifferent Constraint, Technical Report.
Acapulco, Mexico, 2003.
Download PS-File.

[Meh84] Kurt Mehlhorn.
Data Structures and Algorithms, volume 2 Graph Algorithms and NP-Completeness of *EATCS Monographs*.
Springer Verlag, 1984.

[MTW97] K. McAloon, C. Tretkoff, and G. Wetzel.
Sport league scheduling, 1997.

References V

[OSvE95] W. J. Older, G. M. Swinkels, and M. H. van Emden. Getting to the real problem: Experience with bnr prolog in or. In *Proc.of the Third International Conference on the Practical Application of Prolog*, pages 465–478, Paris, 1995.

[qui03] *An Efficient Bounds Consistency Algorithm for the Global Cardinality Constraint*, volume 2833, Kinsale, Ireland, September 2003.

[QvBLO$^+$03] Claude-Guy Quimper, Peter van Beek, Alejandro López-Ortiz, Alexander Golynski, and Sayyed Bashir Sadjad. *An Efficient Bounds Consistency Algorithm for the Global Cardinality Constraint, Technical Report*. 2003. Download PS-File.

References VI

[Ŕ] Jean-Charles Régin.
Minimization of the number of breaks in sports scheduling problems using constraint programming.

[Ŕ96] J-C. Régin.
Generalized arc consistency for global cardinality constraint.
In *Proceedings of the 13th National Conference on AI (AAAI/IAAI'96)*, volume 1, pages 209–215, Portland, August 1996.

[Reg94] *A filtering algorithm for constraints of difference in CSPs*, volume 1, Seattle, July 31 - August 4 1994.

[S.A00] ILOG S.A.
ILOG Solver 5.0:Reference Manual.
2000.

References VII

[SS04a] Christian Schulte and Gert Smolka.
Finite Domain Constraint Programming in Oz. A Tutorial, 1.3.0 edition.
2004.
Download PDF-File.

[SS04b] Christian Schulte and Peter J. Stuckey.
Speeding up constraint propagation.
In Mark Wallace, editor, *Tenth International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, pages 619–633, Toronto, Canada, September 2004. Springer-Verlag.

References VIII

[Thi04] Sven Thiel.
Efficient Algorithms for Constraint Propagation and for Processing Tree Descriptions, PhD Thesis.
Universität des Saarlandes, Saarbrücken, Germany, 2004.
Download: PDF-File.

[WNS97] Mark Wallace, Stefano Novello, and Joachim Schimpf.
Eclipse: A platform for constraint logic programming. Technical report.
IC-Parc, Imperial College, London, UK, 1997.
Online-Version.

References IX

[Zho73] Jianyang Zhou.
A permutation-based-approach for solving the job-shop problem.
Constraints: An International Journal, 2(2):185–213, October 1973.

Bounds Consistent Algorithm for *PermSort*

Paper

A Permutation-Based Approach for Solving the Job-Shop Problem, Jianyang Zhou, Constraints an International Journal 1997 [Zho73]

Algorithm

4 steps: $(n = |X| = |Y| = |P|,\ t = \text{sorting time})$

- Check whether the Y variables are sorted $[O(n)]$
- The permutation variables are distinct and range from 1 to n $[O(n + O(distinct(P)))]$
- Guarantee that $\forall i \in \{1, \dots, n\} \exists p_i \in P : y_i = x_{p_i}$ $[O(n^2)]$
- Metaconstraint stating, that y_i ranks i-th in the ascending sorting of x_i $[O(t + n^2) = O(n^2)]$

$\Rightarrow$ complexity $O(n^2)$

▸ PermSort

Bounds Consistent Algorithm for *Sortedness*

Paper

Efficient Algorithms for Constraint Propagation and for Processing Tree Descriptions, PhD Sven Thiel, 2004 [Thi04]

Algorithm

5 steps: ($n = |X| = |Y|$, $t =$sorting time)

- Sort the domains of the X variables according to lower and upper interval endpoints $[O(t)]$
- Normalize the domains of the Y variables $[O(n)]$
- Compute matchings ϕ, ϕ' in the bipartite convex intersection graph with partitions X and Y $[O(n)]$
- Compute the SCC's in the oriented intersection graph $[O(n)]$
- Narrow the domains of the variables $[O(n)]$

$\Rightarrow$ complexity $O(n + t)$

▸ *Sortedness*

Domain vs. Bounds Consistency

Consistency Levels

Consider propagation for:

$2 \cdot x = y$	
$x \in [1 \dots 10]$	$y \in [1 \dots 7]$

1. **Domain consistency**:

   domain propagation narrows the domains as much as possible:

$x \in [1 \dots 3]$	$y \in \{2, 4, 6\}$

2. **Bounds consistency**:

   interval propagation only narrows the bounds (min,max)

   ⇒ *faster pruning*

$x \in [1 \dots 3]$	$y \in [2 \dots 6]$

▸ running example