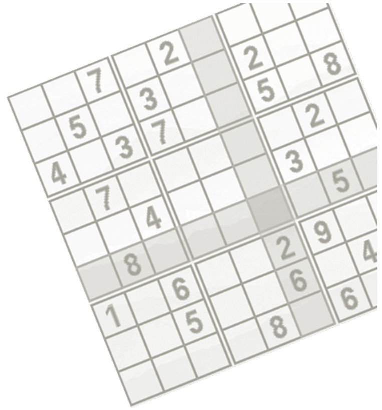


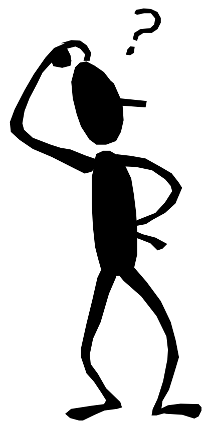
Constraint Programming

Roman Barták
Department of Theoretical Computer Science and Mathematical Logic

Over-constrained problems

Motivation

Dress a robot using minimal wardrobe and fashion rules.

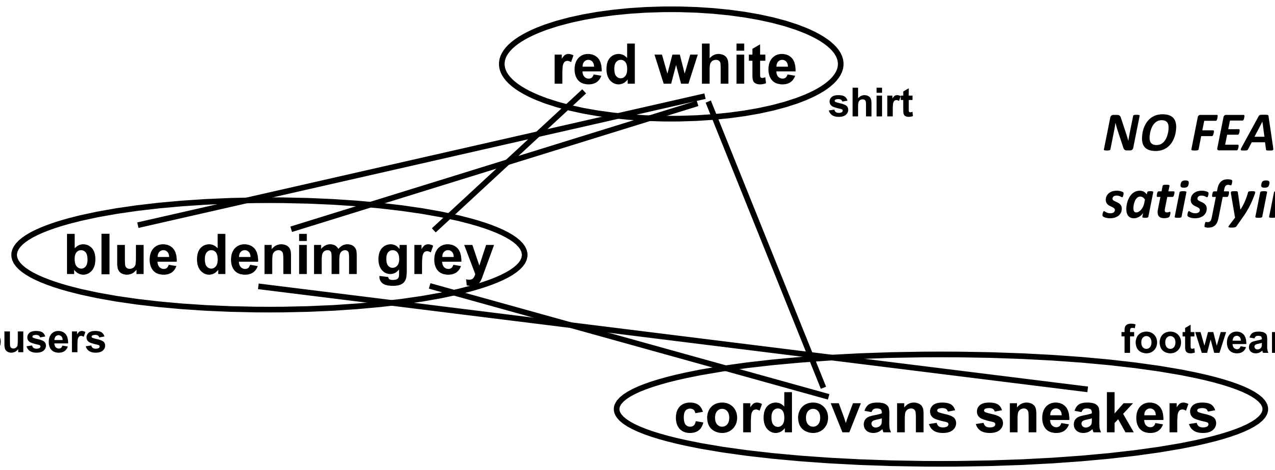


Variables:

- shirt: {red, white}
- footwear: {cordovans, sneakers}
- trousers: {blue, denim, grey}

Constraints:

- shirt x trousers: {red-grey, white-blue, white-denim}
- footwear x trousers: {sneakers-denim, cordovans-grey}
- shirt x footwear: {white-cordovans}



NO FEASIBLE SOLUTION
satisfying all the constraints

We call the problems where no feasible solution exists **over-constrained problems.**

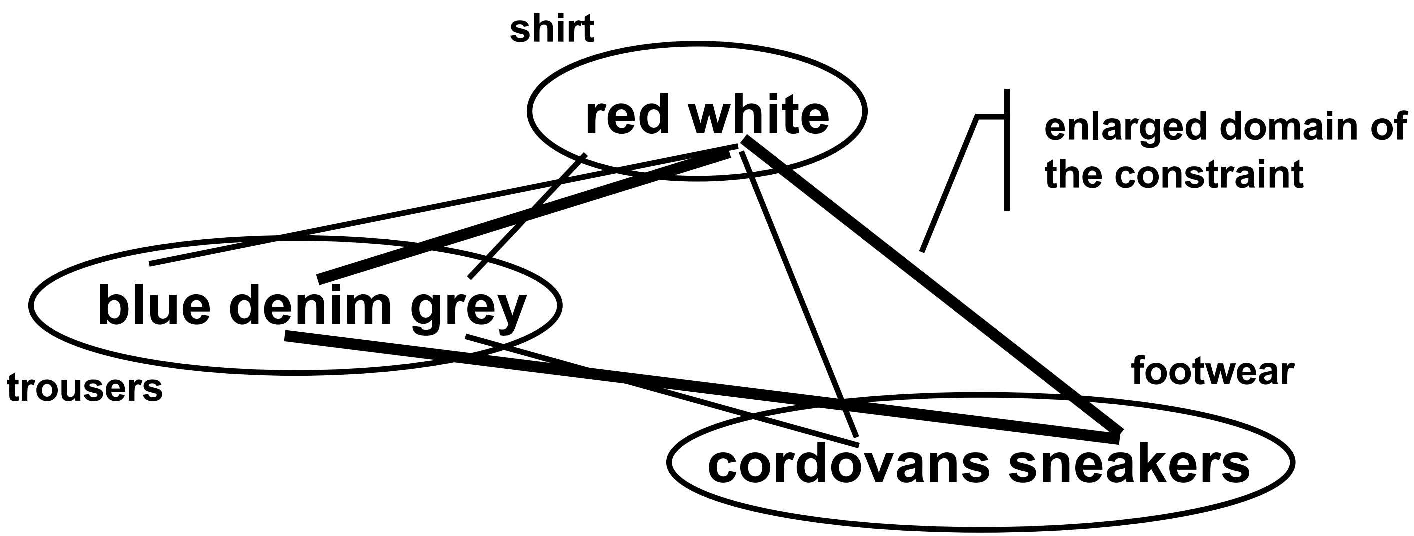
There is no feasible solution, but the robot cannot stay naked!

1) buy new wardrobe
   enlarge domain of some variable
2) use less elegant wardrobe
   enlarge domain of some constraint
3) no matching shoes and shirt
   remove some constraint
4) do not wear shoes
   remove some variable

domain is defined by a unary constraint

all tuples satisfy the constraint

delete all constraints with the variable



First, let us define a **problem space** as a partially ordered set of CSPs $(PS,\leq)$, where $P_1 \leq P_2$ iff the solution set of P_2 is a subset of the solution set of P_1.

The problem space can be obtained by weakening the original problem.

Partial Constraint Satisfaction Problem (PCSP) is a quadruple $\langle P,(PS,\leq),M,(N,S)\rangle$

- P is the original problem
- $(PS,\leq)$ is a problem space containing P
- M is a metric on the problem space defining the problem distance
  - M(P,P‘) could be the number of different solutions of P a P‘
  - or the number of different tuples in the constraint domains
- N is a maximal allowed distance of the problems
- S is a sufficient distance of the problems (S<N)

Solution to a PCSP is a problem P‘ and its solution such that $P' \in PS$ and $M(P,P') < N$. **A sufficient solution** is a solution s.t. $M(P,P') \leq S$.
An optimal solution is a solution with the minimal distance to P.

Partial constraint satisfaction in practice

When solving a PCSP we do not explicitly generate the new problems

- an evaluation function *g* is used instead; it assigns a numeric value to each (even partial) valuation
- the goal is to find assignments minimising/maximising g

PCSP is a generalisation of CSOP:

- $g(x) = f(x)$, if the valuation x is a solution to CSP
- $g(x) = \infty$, otherwise

PCSP is used to solve:

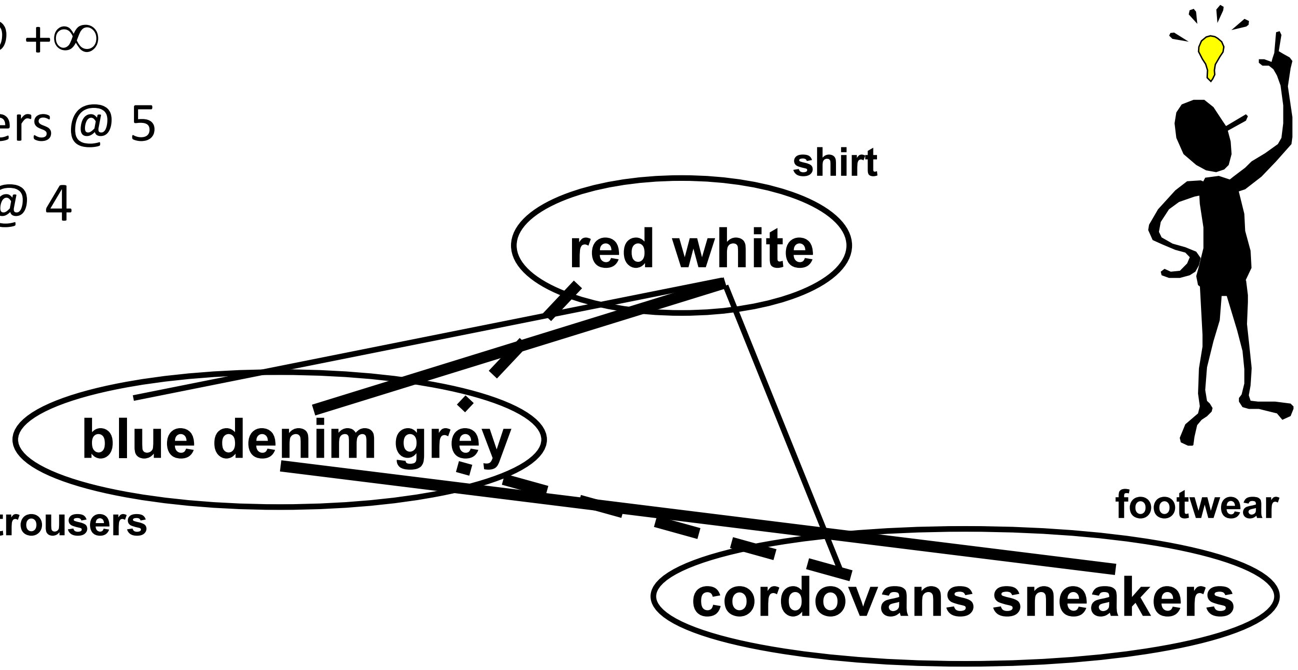
- over-constrained problems
- too complicated problems
- problems using given resources (e.g. time)
- problems in real time (anytime algorithms)

PSCP can be solved using local search, branch and bound, or special propagation algorithms.

Each constraint can be annotated by a **weight** describing importance of satisfying the constraint.

The task is to **minimize the sum of weights** of violated constraints.

- shirt x trousers @ +∞
- footwear x trousers @ 5
- shirt x footwear @ 4



The above problem is called a **Weighted CSP**.
The idea can be further generalised to a **Valued CSP**.

The core idea:

- each constraint is annotated by a certain **valuation**
- then we **aggregate** valuations of violated constraints
- the instantiation with the **least aggregated valuation** is the **solution**

Valuation structure is $(E,\otimes,>,\perp,T)$, where:

- E is a set of valuations that is linearly ordered using >
  with the minimal valuation element $\perp$ and maximal valuation element T
- $\otimes$ is a commutative and associative binary operation on E
  with a unary element $\perp$ ($\perp\otimes a=a$) and absorbing element T ($T\otimes a=T$), that preserves monotony ($a \geq b \Rightarrow a\otimes c \geq b\otimes c$)

Constraints C are mapped to valuations in E using $\varphi: C\rightarrow E$.

A solution is an instantiation A of variables that minimizes the aggregated valuation v(A) given by:

$$v(A) = \bigotimes_{\substack{c\in C \\ \text{A violates c}}} \varphi(c)$$

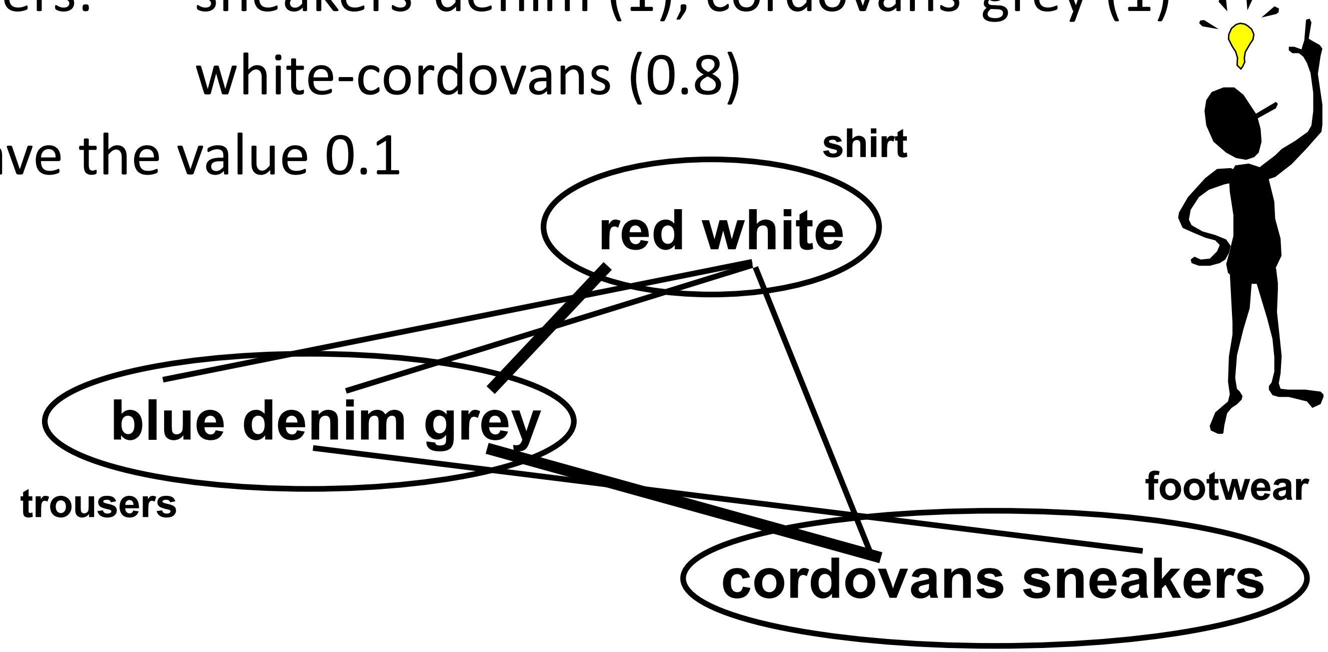
Framework	**E**	$\otimes$	**>**	$\perp$	**T**
Classical CSP	{true,false}	$\wedge$	>	true	false
Weighted CSP	$\mathrm{N} \cup \{+\infty\}$	+	>	0	$+\infty$
Probabilistic CSP	$\langle 0,1 \rangle$	$\times$	<	1	0
Possibilistic CSP	$\langle 0,1 \rangle$	max	>	0	1
Lexicographic CSP	$\mathrm{N}^{\langle 0,1)} \cup \{T\}$	$\cup$	$>_{lex}$	$\varnothing$	T

Each value tuple is annotated by a **preference** of its satisfaction describing how well the tuple satisfies the constraints.

The task is to maximize the product of preferences over all the constraints.

shirt x trousers:	red-grey (1), white-blue (1), white-denim (0.9)
footwear x trousers:	sneakers-denim (1), cordovans-grey (1)
shirt x footwear:	white-cordovans (0.8)

all other pairs have the value 0.1



The above problem is called a **Probabilistic CSP.**

The idea can be generalized to a so called **Semiring-based CSP**.

The core idea:

- each value tuple is annotated by a **preference** describing how well the tuple satisfies the constraint
- a given instantiation of variables is **projected** to each constraint and the obtained preferences are **aggregated**
- the instantiation with the **largest aggregated preference** is the **solution**

C-semi-ring is (A,+,$\times$,0,1), where

- A is a set of preferences,
- + is a commutative, associative, and idempotent (a+a=a) binary operation over A with a unit element 0 (0+a=a) and absorbing element 1 (1+a=1)
  this operation is used to define ordering $a \leq b \Leftrightarrow a+b=b$.
- $\times$ is a commutative and associative binary operation over A with a unit element 1 (1$\times$a=a) and absorbing element 0 (0$\times$a=0), that is distributive over +.

A solution is an instantiation V of variables giving the largest aggregated preference p(V) given by:

$$p(V) = \mathop{\times}_{c \in C} \delta_c\left(V \downarrow vars(c)\right)$$

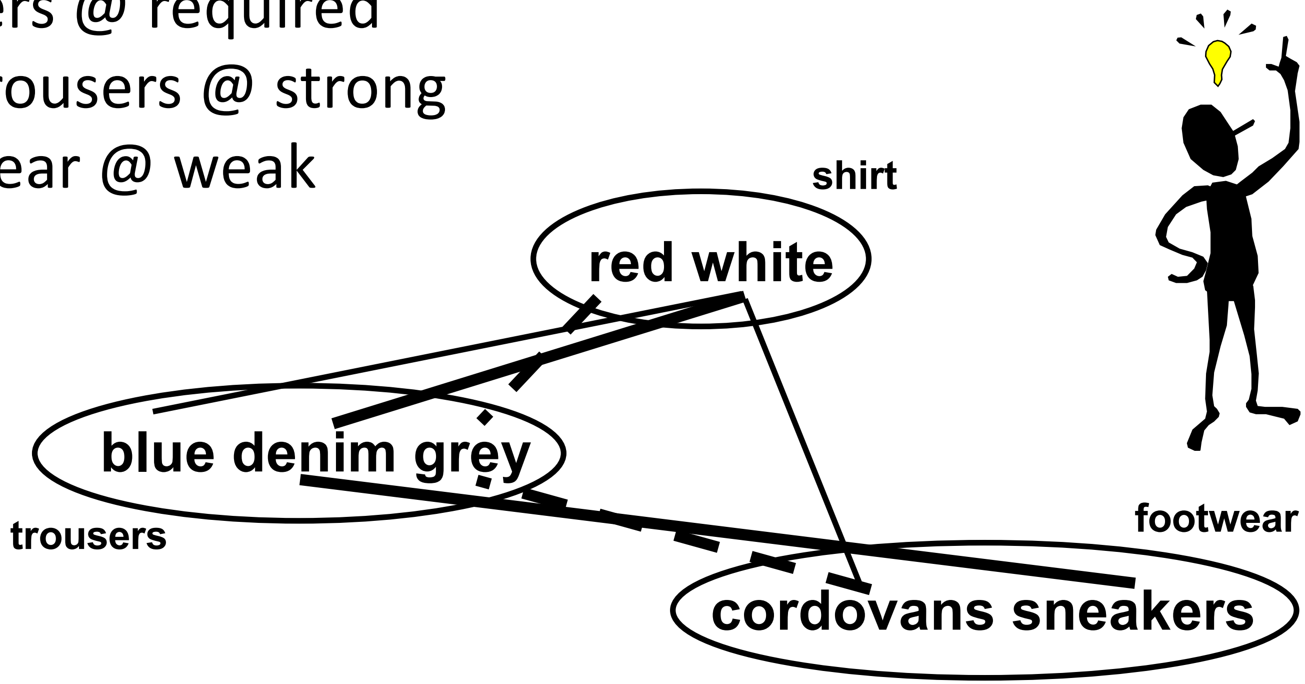
Semiring-based CSP: frameworks

Framework	***A***	**+**	**$\times$**	**1**	**0**
Classical CSP	{false,true}	$\vee$	$\wedge$	true	false
Weighted CSP	$\mathrm{N} \cup \{+\infty\}$	min	+	0	$+\infty$
Probabilistic CSP	$\langle 0,1 \rangle$	max	$\times$	1	0
Possibilistic CSP	$\langle 0,1 \rangle$	min	max	0	1
Fuzzy CSP	$\langle 0,1 \rangle$	max	min	1	0
Lexicographic CSP	$\mathrm{N}^{\langle 0,1)} \cup \{\mathrm{T}\}$	$\max_{lex}$	$\cup$	$\varnothing$	T

Constraints can be annotated by **preferences** describing which constraints are preferred to be satisfied.

Now, the preferences are **strict**! Satisfaction of a stronger constraint is preferred to satisfaction of any weaker constraint.

- shirt x trousers @ required
- footwear x trousers @ strong
- shirt x footwear @ weak



The above model is called a **constraint hierarchy** – constraints with the same preference form a layer in this hierarchy.

Each constraint is annotated by a **symbolic preference** (preferences are linearly ordered).

- there is a specific preference *required* to denote constraints that must be satisfied – **hard constraints**
- other constraints may be violated – **soft constraints**

Constraint hierarchy H is a finite (multi)set of constraints.

- H_0 is a set of required constraints (without the preference)
- H_1 is a set of the most preferred constraints
- …

A solution is an instantiation of variables satisfying all hard constraints and satisfying soft constraints as well as possible.

- $S_{H,0} = \{\sigma \mid \forall c \in H_0,\ c\sigma \text{ holds}\}$
- $S_H = \{\sigma \mid \sigma \in S_{H,0} \wedge \forall \omega \in S_{H,0}\ \neg\, \text{better}(\omega,\sigma,H)\ \}$

Constraint hierarchies: comparators

How to compare instantiations with respect to a hierarchy?

- anti-reflexive, transitive comparison relation **respecting the hierarchy**
- if some instantiation satisfies all constraints up to level k, then any better instantiation has the same property

Error function $e(c,\sigma)$ – describes how well the constraint is satisfied

- predicate error function (satisfied/violated)
- metric error function – measures a distance from solution, $e(X\geq 5,\{X/3\}) = 2$

Local comparators

- compare errors of individual constraints

$$\text{locally\_better}(\omega,\sigma,H) \equiv \exists k>0$$
$$\forall i<k\ \forall c\in H_i\ e(c,\omega)=e(c,\sigma) \wedge \forall c\in H_k\ e(c,\omega) \leq e(c,\sigma) \wedge \exists c\in H_k\ e(c,\omega)<e(c,\sigma)$$

Global comparators

- aggregate all errors for constraints in the level using the function

$$\text{globally\_better}(\omega,\sigma,H) \equiv \exists k>0\ \forall i<k\ \ g(H_i,\omega)=g(H_i,\sigma) \wedge g(H_k,\omega)<g(H_k,\sigma)$$

  - we can use weighted sum, sum of squares, worst case, etc.

DeltaStar uses a method of **refining the set of candidate instantiations**.

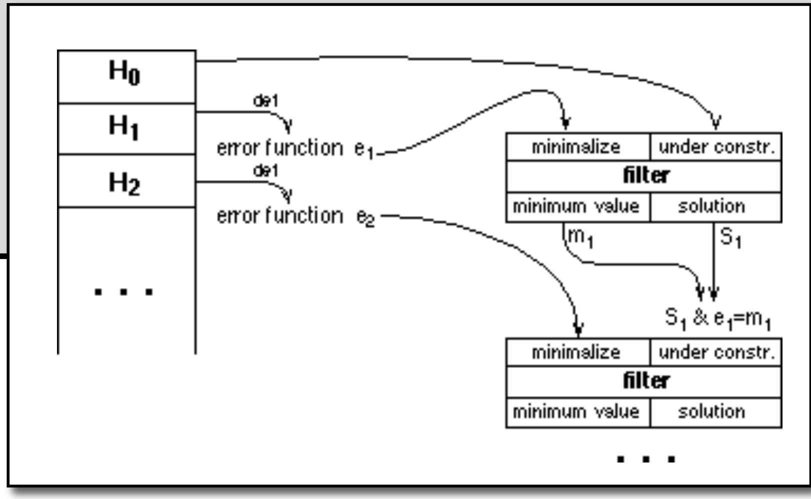
We need a "flat" constraint solver with function filter:

instantiations $\times$ *constraints* $\rightarrow$ *instantiations*

- from a set of instantiations select those instantiations that best satisfy the constraints (this implements the comparator)
- instantiations can be represented in an implicit form

Algorithm DeltaStar

procedure DeltaStar(H: constraint hierarchy)
  $i \leftarrow 1$
  Solution $\leftarrow$ all solutions of required constraints from H
  while not unique Solution and i<number of levels **do**
    Solution $\leftarrow$ filter(Solution,H_i)
    i++
  endwhile
  return Solution
end DeltaStar



- filter can be implemented using simplex
- constraints from the next level are part of the error function

We can incrementally satisfy the constraints.

- each constraint is described by a set of methods for its satisfaction by propagating values between variables

  A+B = C $A \leftarrow C\text{-}B$, $B \leftarrow C\text{-}A$, $C \leftarrow A\text{+}B$,

- for each variable we need one method computing its value
- then the computed value is used as input to other methods

advantages:

- changed values can be propagated through the network
- we can compile methods

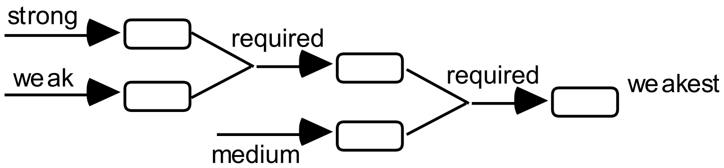
drawbacks:

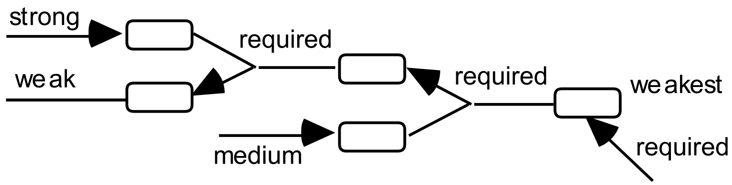
- works only for functional constraints (such as equalities)
- no cycle of methods in the network
- finds a single solution
- works with locally predicate comparators only

The algorithm works with single-output methods.

- first, select a method for each constraint (planning)
- then propagate values through the methods (execution)

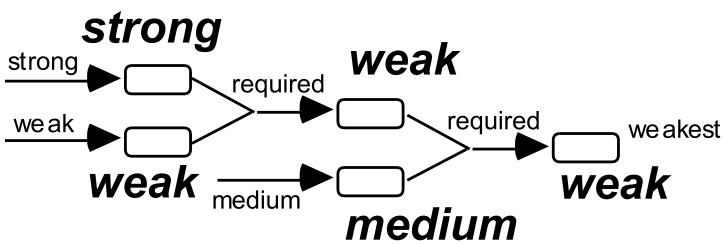
Incremental planning – modify the network after adding a constraint





The algorithm uses **walkabout strengths** to guide the network planner.

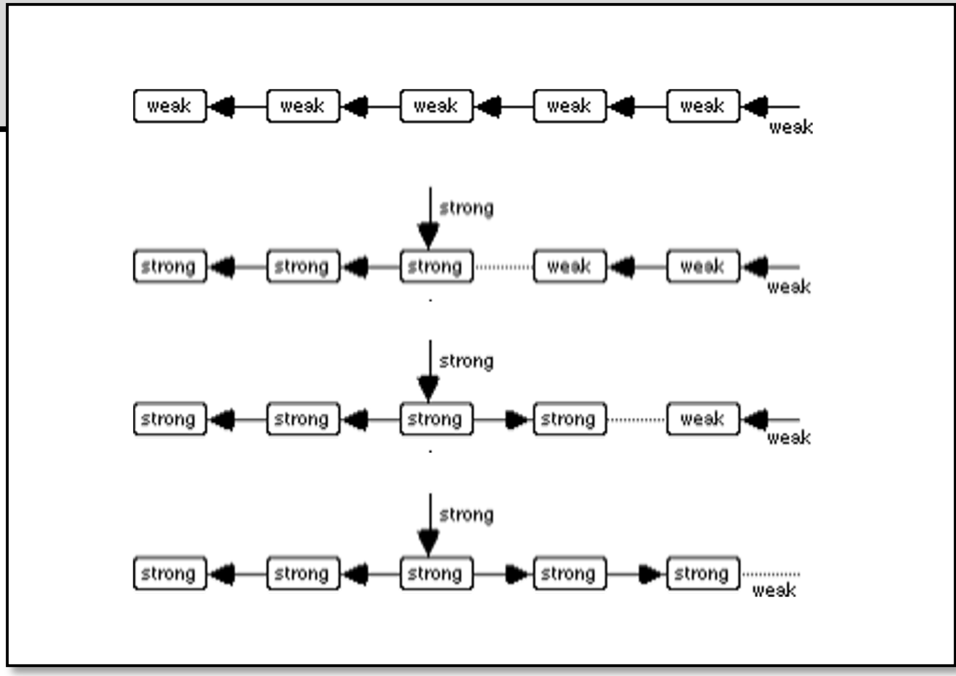
Walkabout strength of the variable is the weakest preference among the preference of the method outputting the variable and the walkabout strengths of variables that are outputs of other (not selected) methods of the same constraint.



Algorithm DeltaBlue

procedure AddConstraint(c: constraint)
select the potential output variable V of c with the weakest walkabout strength
if walkabout strength of V is weaker than strength of c **then**
c‘← the constraint currently determining the variable V
make c' unsatisfied
select the method determining V in c
re-compute walkabout strengths of downstream variables
AddConstraint(c')
endif
end AddConstraint

- after adding a constraint, the network is locally modified
- re-compute the walkabout preferences
- try to add the removed constraints back



DeltaBlue works with functional constraints modelled using single-output methods only.

SkyBlue generalises DeltaBlue to support multi-output methods.

Both algorithms construct an acyclic network of methods (if possible) and do not support non-functional constraints such as A<B.

Algorithm **Indigo** was designed for acyclic networks with non-functional constraints and locally metric comparator.

- **uses bounds consistency**
  - always runs all the methods for a constraint
- **incrementally adds constraints** from strongest to weakest
  - after adding a constraint, ensures bounds consistency
  - by propagating the bounds to other variables

Constraint hierarchies: Indigo run

c1: required a>=10
c2: required b>=20
c3: required a+b=c

c4: required c+25=d
c5: strong d<=100
c6: medium a=50

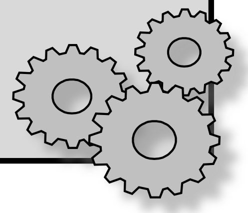
c7: weak a=5
c8: weak b=5
c9: weak c=100
c10: weak d=200

action	a	b	c	d	note
	(-inf,inf)	(-inf,inf)	(-inf,inf)	(-inf,inf)	*initial bounds*
add c1	[10,inf)	(-inf,inf)	(-inf,inf)	(-inf,inf)	
add c2	[10,inf)	[20,inf)	(-inf,inf)	(-inf,inf)	
add c3	[10,inf)	[20,inf)	[30,inf)	(-inf,inf)	
add c4	[10,inf)	[20,inf)	[30,inf)	[55,inf)	
add c5	[10,inf)	[20,inf)	[30,inf)	[55,100]	
	[10,inf)	[20,inf)	[30,75]	[55,100]	*propagate bounds using c4*
	[10,55]	[20,65]	[30,75]	[55,100]	*propagate bounds using c3*
add c6	[50,50]	[20,65]	[30,75]	[55,100]	
	[50,50]	[20,25]	[70,75]	[55,100]	*propagate bounds using c3*
	[50,50]	[20,25]	[70,75]	[95,100]	*propagate bounds using c4*
add c7	[50,50]	[20,25]	[70,75]	[95,100]	*c7 is unsatisfied*
add c8	[50,50]	[20,20]	[70,75]	[95,100]	*c8 is unsatisfied but its error is minimized*
	[50,50]	[20,20]	[70,70]	[95,100]	*propagate bounds using c3*
	[50,50]	[20,20]	[70,70]	[95,95]	*propagate bounds using c4*
add c9	[50,50]	[20,20]	[70,70]	[95,95]	*c9 is unsatisfied*
add c10	[50,50]	[20,20]	[70,70]	[95,95]	*c10 is unsatisfied*

Solving linear equalities and inequalities using Gauss and Fourier elimination.

- C(0,x) = constraints that do not contain x
- C(=,x) = equalities containing x
- C(+,x) = inequalities containing x, s.t. the constraint has a form $x \leq e$
- C(-,x) = inequalities containing x, s.t. the constraint has a form $e \leq x$

procedure project(C: set of constraints, x: variable)
    if $\exists c \in C(=,x)$ where c is x=e **then**
        $D \leftarrow C - \{c\}$ with every occurrence of x replaced by e
    else
        $D \leftarrow C(0,x)$
        for each c in C(+,x) where c is $x \leq e^{+}$ **do**
            for each c in C(-,x) where c is $e^{-} \leq x$ **do**
                $D \leftarrow D \cup \{e^{-} \leq e^{+}\}$
            endfor
        endfor
    endif
    return D
end project



Projection algorithm in practice

We first eliminate all variables by projection
and then via a backward run we calculate the values for variables.

$$\begin{aligned} 2x_m &= x_l + x_r \\ x_l + 10 &\le x_r \\ 0 &\le x_l, x_m, x_r \\ x_l, x_m, x_r &\le 100 \end{aligned}$$

project to x_l

$$\begin{aligned} x_m + 5 &\le x_r \\ 2x_m - 100 &\le x_r \\ x_r &\le 2x_m \\ x_m, x_r &\le 100 \end{aligned}$$

project to x_r

$$\begin{aligned} 5 &\le x_m \\ x_m &\le 95 \end{aligned}$$

project to x_m

$$5 \le 95$$

x_m in 5..95
e.g. x_m =50

x_r in 55..100
e.g. x_r =70

x_l=2x_m-x_r
i.e. x_l = 30

And what about supporting constraint hierarchies?

- for a locally metric comparator
  - constraints "e?b @ pref" are transformed to "e?v_e @ required", "v_e=b @ pref" (v_e is a new variable, ? is any relation =, ≤, ≥)
  - variables from the required constraints are eliminated from strongest to weakest
  - a value closest to b from the strongest constraint is used

Department of Theoretical Computer Science and Mathematical Logic
bartak@ktiml.mff.cuni.cz