Foundations of constraint satisfaction

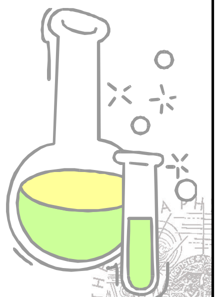
Roman Barták
Charles University in Prague

bartak@ktiml.mff.cuni.cz
http://ktiml.mff.cuni.cz/~bartak

What is the course about?

Constraint satisfaction problems
Algorithms for solving constraint satisfaction problems

- Local search
  - HC, MC, RW, Tabu Search
- Search algorithms
  - GT, BT, BJ, BM, DB, LDS
- Consistency techniques
  - NC, AC, DAC, PC, DPC, RPC, SC
- Search and constraint propagation
  - FC, PLA, LA
- Optimisation problems
  - B&B
- Over-constrained problems
  - PCSP, constraint hierarchies



What is a constraint?

Constraint is an arbitrary relation over the set of variables.
- every variable has a set of possible values - a domain
  - this course covers discrete finite domains only
- the constraint restricts the possible combinations of values

Some examples:
- the circle C is inside a square S
- the length of the word W is 10 characters
- X is less than Y
- a sum of angles in the triangle is 180°
- the temperature in the warehouse must be in the range 0-5°C
- John can attend the lecture on Wednesday after 14:00

Constraint can be described:
- intentionally (as a mathematical/logical formula)
- extensionally (as a table describing compatible tuples)

Constraint Satisfaction Problem

CSP (Constraint Satisfaction Problem) consists of:
- a finite set of variables
- domains - a finite set of values for each variable
- a finite set of constraints

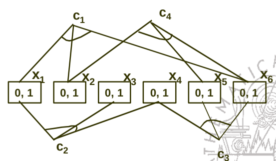
A solution to CSP is a complete assignment of variables satisfying all the constraints.

CSP is often represented as a (hyper)graph.

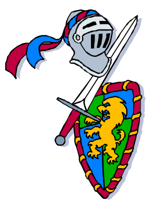
Example:

variables $x_1,\dots,x_6$
domain {0,1}

c_1: $x_1+x_2+x_6=1$
c_2: $x_1-x_3+x_4=1$
c_3: $x_4+x_5-x_6>0$
c_4: $x_2+x_5-x_6=0$



A bit of history



Artificial Intelligence
  Scene labelling (Waltz 1975)

Interactive graphics
  Sketchpad (Sutherland 1963)
  ThingLab (Borning 1981)

Logic programming
  unification ® constraint solving (Gallaire 1985, Jaffar, Lassez 1987)

Operations research and discrete mathematics
  NP-hard combinatorial problems

Some toy problems

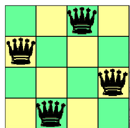
SEND + MORE = MONEY

assign different numerals to different letters
  S and M are not zero

A constraint model (with a carry bit):

```
E,N,D,O,R,Y in 0..9, S,M in 1..9, P1,P2,P3::0..1
all_different(S,E,N,D,M,O,R,Y)
     D+E = 10*P1+Y
  P1+N+R = 10*P2+E
  P2+E+O = 10*P3+N
  P3+S+M = 10*M +O
```

N-queens problem



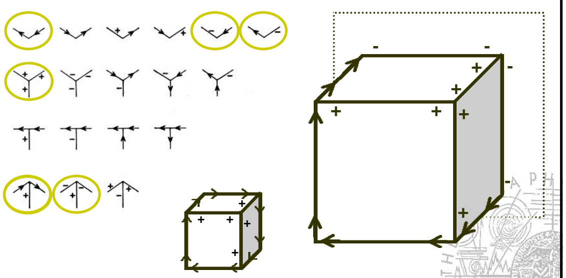
allocate N queens to the chessboard
  the queens do not attack each other

A constraint model:

queens in columns `"i r(i) in 1..N`
no conflict
  `"i¹j   r(i)¹r(j) & |i-j|¹|r(i)-r(j)|`

Constraints in scene labelling (Waltz 1975)

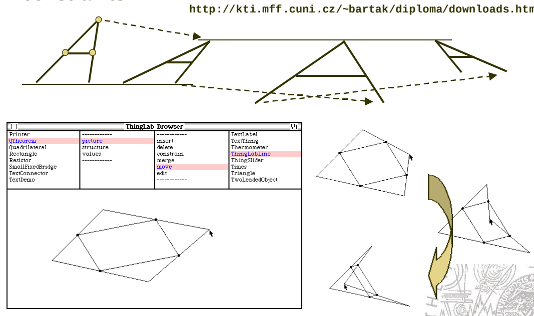
Looking for feasible interpretation of 3D lines in 2D drawing
First usage of constraint propagation techniques



Constraints in interactive graphics

How to manipulate a graphical object described by constraints?

http://kti.mff.cuni.cz/~bartak/diploma/downloads.html

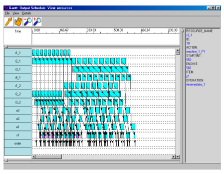


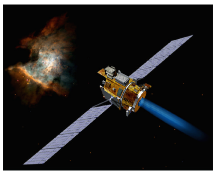
http://www.cs.washington.edu/research/constraints/

Constraints in A.I. planning and scheduling

Scheduling problem =
a set of activities has to be processed by a limited number of resources in a limited amount of time.

Combinatorial optimisation





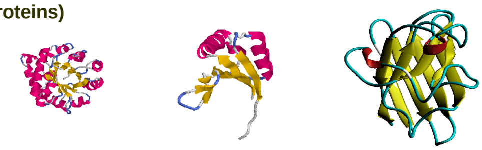
Planning problem =
find a set of activities to achieve a given goal

Deep Space One
- autonomous planning of spacecraft activities

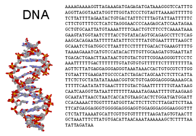
Constraints in bioinformatics

http://www.soi.city.ac.uk/~drg/bioinformatics/

Design of a 3D protein structure from the sequence of amino-acids (3D structure determines features of proteins)

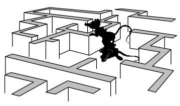


Analysing a sequence of DNA, estimating a distance between DNAs, comparing DNAs



Solving constraints by enumeration

Constraints are used only as a test
assign values to variables ...
... and see what happens



systematic search
explores the space of all assignments systematically
GT, BT, BJ, BM, DB, LDS

non-systematic search
some assignments may be skipped during search
Credit Search, Bounded Backtrack

local search
explore the search space by small steps
HC, MC, RW, Tabu, *GSAT, Genet, simulated annealing*

Systematic search

Explore systematically the space of all assignments
systematic = every valuation will be explored sometime

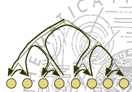
Features:
+ complete (if there is a solution, the method finds it)
- it could take a lot of time to find the solution

Basic classification:

Explore complete assignments

generate and test
such search space is used by local search (non-systematic)

Extending partial assignments
tree search



Generate and test (GT)

The most general problem solving method
1) generate a candidate for solution
2) test if the candidate is really a solution

How to apply GT to CSP?
1) assign values to all variables
2) test whether all the constraints are satisfied

GT explores complete but inconsistent assignments until a (complete) consistent assignment is found.

Procedure GT(X:variables, C:constraints)

```
V ¬ construct a first complete assignment of X
while V does not satisfy all the constraints C do
   V ¬ construct systematically a complete assignment next to V
end while
return V
```


Weaknesses and improvements of GT

The greatest weakness of GT is exploring too many "visibly" wrong assignments.

Example:
X,Y,Z::{1,2}   X = Y, X ¹ Z, Y > Z

X	1	1	1	1	2	2	2	2
Y	1	1	2	2	1	1	2	2
Z	1	2	1	2	1	2	1	2

How to improve generate and test?

smart generator

smart (perhaps non-systematic) generator that uses result of test
å local search techniques

earlier detection of clash

constraints are tested as soon as the involved variables are instantiated ® backtracking-based search

Local search

Generate and test explores complete but inconsistent assignments until a complete consistent assignment is found.

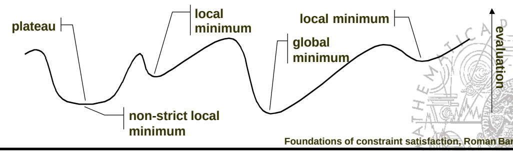
Weakness of GT - the generator does not use result of test

The next assignment can be constructed in such a way that constraint violation is smaller.
- only "small" changes of the assignment are allowed
- next assignment should be "better" than previous
  - better = more constraints are satisfied
- assignments are not necessarily generated systematically
  - we lost completeness but we (hopefully) get better efficiency

Local Search is a technique of searching solution by small changes (local steps) to the solution candidate.

Local search - Terminology

state - a complete assignment of values to variables
evaluation - a value of the objective function (# violated constraints)
neighbourhood - a set of states locally different from the current state (the states differ from the current state in the value of one variable)
local optimum - a state that is not optimal and there is no state with better evaluation in its neighbourhood
strict local optimum - a state that is not optimal and there are only states with worse evaluation in its neighbourhood
non-strict local optimum - local optimum that is not strict
global optimum - the state with the best evaluation
plateau - a set of neighbouring states with the same evaluation



Hill Climbing

Hill climbing is perhaps the most known technique of local search.
- start at randomly generated state
- look for the best state in the neighbourhood of the current state
  - *neighbourhood* = differs in the value of any variable
  - neighbourhood size = $s_{i=1..n}(D_i-1)$ (= n*(d-1))
- "escape" from the local optimum via restart

Algorithm Hill Climbing

```
procedure hill-climbing(Max_Flips)
   restart: s ¬ random assignment of variables;
   for j:=1 to Max_Flips do          % restricted number of steps
      if eval(s)=0 then return s
      if s is a strict local minimum then
         go to restart
      else
         s ¬ neighbourhood with the smallest evaluation value
      end if
   end for
   go to restart
end hill-climbing
```


Min-Conflicts (Minton, Johnston, Laird 1997)

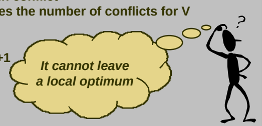
Observation:
- the hill climbing neighbourhood is pretty large (n*(d-1))
- only change of a conflicting variable may improve the valuation

Min-conflicts method

select randomly a varible in conflict and try to improve it
- *neighbourhood* = different values for the selected variable *i*
- neighbourhood size = (D_i-1) (= (d-1))

Algorithm Min-Conflicts

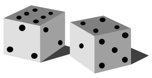
```
procedure MC(Max_Moves)
   s ¬ random assignment of variables
   nb_moves ¬ 0
   while eval(s)>0 & nb_moves<Max_Moves do
      choose randomly a variable V in conflict
      choose a value v' that minimises the number of conflicts for V
      if v' ¹ current value of V then
         assign v' to V
         nb_moves ¬ nb_moves+1
      end if
   end while
   return s
end MC
```



Random walk

How to leave the local optimum without a restart (i.e. via a local step)?

By adding some "noise" to the algorithm!



Random walk

- a state from the neighbourhood is selected randomly (e.g., the value is chosen randomly)
- such technique can hardly find a solution
- so it needs some guide

Random walk can be combined with the heuristic guiding the search via probability distribution:

- *p* - probability of using the random walk
- *(1-p)* - probability of using the heuristic guide

Min-Conflicts Random Walk

MC guides the search (i.e. satisfaction of all the constraints) and RW allows us to leave the local optima.

Algorithm Min-Conflicts-Random-Walk

```
procedure MCRW(Max_Moves,p)
    s ¬ random assignment of variables
    nb_moves ¬ 0
    while eval(s)>0 & nb_moves<Max_Moves do
        if probability p verified then
            choose randomly a variable V in conflict
            choose randomly a value v' for V
        else
            choose randomly a variable V in conflict
            choose a value v' that minimises the number of conflicts for V
        end if
        if v' ı current value of V then
            assign v' to V
            nb_moves ¬ nb_moves+1
        end if
    end while
    return s
end MCRW
```

0.02 £ p £ 0.1

Steepest Descent Random Walk

Random walk can be combined with the hill climbing heuristic too. Then, no restart is necessary.

Algorithm Steepest-Descent-Random-Walk

```
procedure SDRW(Max_Moves,p)
    s ¬ random assignment of variables
    nb_moves ¬ 0
    while eval(s)>0 & nb_moves<Max_Moves do
        if probability p verified then
            choose randomly a variable V in conflict
            choose randomly a value v' for V
        else
            choose a move <V,v'> with the best performance
        end if
        if v' ı current value of V then
            assign v' to V
            nb_moves ¬ nb_moves+1
        end if
    end while
    return s
end SDRW
```

Tabu list

Observation:

Being trapped in local optimum is a special case of cycling.

How to avoid cycles in general?

- Remember already visited states and do not visit them again.
  - memory consuming (too many states)
- It is possible to remember just few last states.
  - prevents „short" cycles

Tabu list = a list of forbidden states

- the state can be represented by a selected attribute
  - ıvariable, valueñ - describes the change of the state (a previous value)
- tabu list has a fix length *k* (tabu tenure)
  - „old" states are removed from the list when a new state is added
- state included in the tabu list is forbidden (it is tabu)

Aspiration criterion = enabling states that are tabu

- i.e., it is possible to visit the state even if the state is tabu
- *example*: the state is better than any state visited so far

Tabu search (Galinier, Hao 1997)

The tabu list prevents short cycles.

It allows only the moves out of the tabu list or the moves satisfying the aspiration criterion.

Algorithm Tabu Search

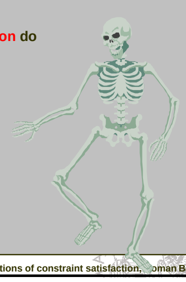
```
procedure tabu-search(Max_Iter)
    s ¬ random assignment of variables
    nb_iter ¬ 0
    initialise randomly the tabu list
    while eval(s)>0 & nb_iter<Max_Iter do
        choose a move <V,v'> with the best performance among the non-tabu
            moves and the moves satisfying the aspiration criteria
        introduce <V,v> in the tabu list, where v is the current value of V
        remove the oldest move from the tabu list
        assign v' to V
        nb_iter ¬ nb_iter+1
    end while
    return s
end tabu-search
```


Localizer (Michel, Van Hentenryck 1997)

The local search algorithms have a similar structure that can be encoded in the common skeleton. This skeleton is filled by procedures implementing a particular technique.

Local Search Skeleton

```
procedure local-search(Max_Tries,Max_Moves)
    s ¬ random assignment of variables
    for i:=1 to Max_Tries while Gcondition do
        for j:=1 to Max_Moves while Lcondition do
            if eval(s)=0 then
                return s
            end if
            select n in neighbourhood(s)
            if acceptable(n) then
                s ¬ n
            end if
        end for
        s ¬ restartState(s)
    end for
    return best s
end local-search
```



Foundations
of constraint satisfaction

2

Roman Barták
Charles University in Prague

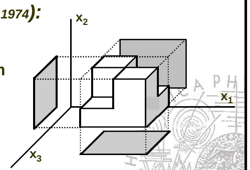
bartak@ktiml.mff.cuni.cz
http://ktiml.mff.cuni.cz/~bartak

Binary constraints

World is not binary …
but it could be transformed to a binary one!

Each CSP can be transformed to an equivalent binary CSP
– many CSP algorithms designed for binary problems
– still open efficiency issues

Projection technique (Montanary 1974):
- straightforward but
- does not give an equivalent problem
- bound consistency
  - better efficiency
  - weaker pruning



Dual encoding

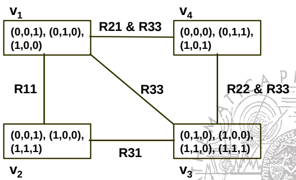
Swapping variables and constraints.

k- ary constraint c is converted to
a dual variable v_c with the domain consisting of compatible tuples

for each pair of constraints c a c' sharing some variables there is
a binary constraint between v_c a $v_{c'}$ restricting the dual variables
to tuples in which the original shared variables take the same value

Example:
variables $x_1,\dots,x_6$
with domain {0,1}

c_1: $x_1+x_2+x_6=1$
c_2: $x_1-x_3+x_4=1$
c_3: $x_4+x_5-x_6>0$
c_4: $x_2+x_5-x_6=0$



Hidden variable encoding

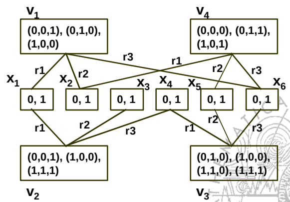
New dual variables for (non-binary) constraints.

k- ary constraint c is translated to
a dual variable v_c with the domain consisting of compatible tuples

for each variable x in the constraint c there is a constraint between
x a v_c restricting tuples of dual variable to be compatible with x

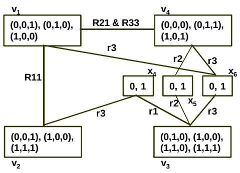
Example:
variables $x_1,\dots,x_6$
with domains {0,1}

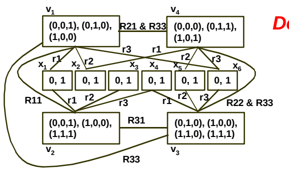
c_1: $x_1+x_2+x_6=1$
c_2: $x_1-x_3+x_4=1$
c_3: $x_4+x_5-x_6>0$
c_4: $x_2+x_5-x_6=0$



Other encodings

Hybrid encoding
transformation between dual and
hidden variable encoding
contains parts of both encodings





Double encoding
hidden and original variables
are included
constraints from both
encodings are used
improved propagation

Backtracking

Probably the most widely used systematic search algorithm
basically it is depth-first search

Using backtracking to solve CSP
1) assign values gradually to variables
2) after each assignment test the constraints over the assigned variables (and backtrack upon failure)

Extends a partial consistent assignment until a complete consistent assignment is found.

Open questions:
what is the order of variables?
- variables with a smaller domain first
- variables participating in more constraints first
- "key" variables first

what is the order of values?
- problem dependent

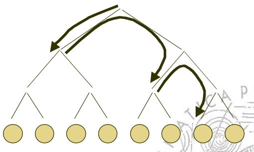
Algorithm chronological backtracking

A recursive definition

Algorithm BT(X:variables, V:assignment, C:constraints)

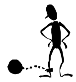
if X={} then return V
x ¬ select a not-yet assigned variable from X
for each value h from the domain of x do
  if constraints C are satisfied over V+x/h then
    R ¬ BT(X-x, V+x/h, C)
    if R¹fail then return R
end for
return fail

top call BT(X, {}, C)



Backtracking is always better than generate and test!

Weaknesses of backtracking



thrashing

throws away the reason of the conflict
Example: A,B,C,D,E:: 1..10, A>E
BT tries all the assignments for B,C,D before finding that A¹1
Solution: backjumping (jump to the source of the failure)

redundant work

unnecessary constraint checks are repeated
Example: A,B,C,D,E:: 1..10, B+8<D, C=5*E
when labelling C,E the values 1,..,9 are repeatedly checked for D
Solution: backmarking, backchecking (remember (no-)good assignments)

late detection of the conflict

constraint violation is discovered only when the values are known
Example: A,B,C,D,E::1..10, A=3*E
the fact that A>2 is discovered when labelling E
Solution: forward checking (forward check of constraints)

Backjumping (Gaschnig 1979)

Backjumping is used to remove thrashing.

How?

1) identify the source of the conflict (impossible to assign a value)
2) jump to the past variable in conflict

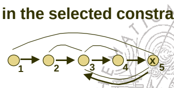
The same run like in backtracking, only the back-jump can be longer, i.e. irrelevant assignments are skipped!

How to find a jump position? What is the source of the conflict?

select the constraints containing just the currently assigned variable and the past variables

select the closest variable participating in the selected constraints

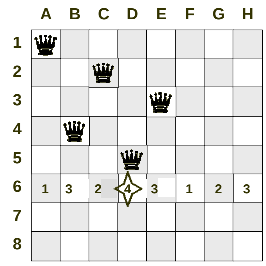
Graph-directed backjumping



Enhancement: use only the violated constraints

Conflict-directed backjumping in practice

N-queens problem



Queens in rows are allocated to columns.

6th queen cannot be allocated!

1. Write a number of conflicting queens to each position.

2. Select the farthest conflicting queen for each position.

3. Select the closest conflicting queen among positions.

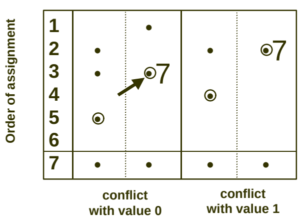
Note:
Graph-directed backjumping has no effect here (due to complete graph)!

Identification of the conflicting variable

How to find out the conflicting variable?

Situation:

assume that the variable no. 7 is being assigned (values are 0, 1)
the symbol · marks the variables participating the violated constraints (two constraints for each value)



Neither 0 nor 1 can be assigned to the seventh variable!

1. Find the closest variable in each violated constraint (o).

2. Select the farthest variable from the above chosen variables for each value (7).

3. Choose the closest variable from the conflicting variables selected for each value and jump to it.

Consistency check for backjumping

In addition to the test of satisfaction of the constraints, the closest conflicting level is computed

```
procedure consistent(Labelled, Constraints, Level)
  J ¬ Level          % the level to which we will jump
  NoConflict ¬ true  % remember if there is any conflict
  for each C in Constraints do
    if all variables from C are Labelled then
      if C is not satisfied by Labelled then
        NoConflict ¬ false
        J ¬ min {J, max{L | X in C & X/V/L in Labelled & L<Level}}
      end if
    end if
  end for
  if NoConflict then return true
               else return fail(J)
end consistent
```

Algorithm backjumping

```
procedure BJ(Unlabelled, Labelled, Constraints, PreviousLevel)
    if Unlabelled = {} then return Labelled
    pick first X from Unlabelled
    Level ¬ PreviousLevel+1
    Jump ¬ 0
    for each value V from D_X do
        C ¬ consistent({X/V/Level} È Labelled, Constraints, Level)
        if C = fail(J) then
            Jump ¬ max {Jump, J}
        else
            Jump ¬ PreviousLevel
            R ¬ BJ(Unlabelled-{X},{X/V/Level} È Labelled,Constraints, Level)
            if R ¹ fail(Level) then return R          % success or back-jump
        end if
    end for
    return fail(Jump)     % jump to the conflicting variable
end BJ
```

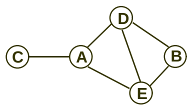
top call BJ(Variables,{},Constraints,0)

Weakness of backjumping

When jumping back the in-between assignment is lost!

Example:

colour the graph in such a way that the connected vertices have different colours



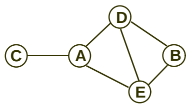
node	vertex	
A	1	1
B	2	1
C	1 2	1 2
D	1 2 3	1 2
E	1 2 3	1 2 3

(backjump: from E to B)

During the second attempt to label C superfluous work is done - it is enough to leave there the original value 2, the change of B does not influence C.

Dynamic backtracking - example

The same graph (A,B,C,D,E), the same colours (1,2,3) but a different approach.



Backjumping
+ remember the source of the conflict
+ carry the source of the conflict
+ change the order of variables

= DYNAMIC BACKTRACKING

node	1	2	3
A	·		
B		·	
C	A	·	
D	A	B	·
E	A	B	D

jump back
+ carry the conflict source

node	1	2	3
A	·		
B		·	
C	A	·	
D	A	B	AB
E	A	B	

jump back
+ carry the conflict source
+ change the order of B, C

node	1	2	3
A	·		
C	A	·	
B	·	A	
D	A	·	
E	A	B	·

· selected colour
AB a source of the conflict

The vertex C (and the possible sub-graph connected to C) is not re-coloured.

Algorithm dynamic backtracking (Ginsberg 1993)

```
procedure DB(Variables, Constraints)
    Labelled ¬ {};   Unlabelled ¬ Variables
    while Unlabelled ¹ {} do
        select X in Unlabelled
        Values_X ¬ D_X - {values inconsistent with Labelled using Constraints}
        if Values_X = {} then
            let E be an explanation of the conflict (set of conflicting variables)
            if E = {} then failure
            else
                let Y be the most recent variable in E
                unassign Y (from Labelled) with eliminating explanation E-{Y}
                remove all the explanations involving Y
            end if
        else
            select V in Values_X
            Unlabelled ¬ Unlabelled - {X}
            Labelled ¬ Labelled È {X/V}
        end if
    end while
    return Labelled
end DB
```

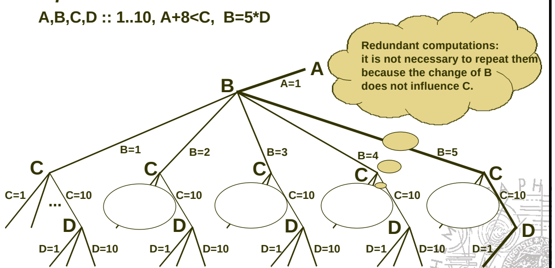

Redundant work in backtracking

What is redundant work?

repeated computation whose result has already been obtained

Example:

A,B,C,D :: 1..10, A+8<C, B=5*D

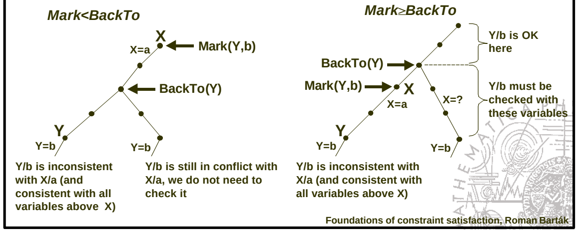


Backmarking (Haralick, Elliot 1980)

Removes redundant constraint checks by memorising negative and positive tests:

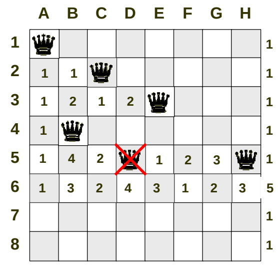
- *Mark(X,V)* is the farthest (instantiated) variable in conflict with the assignment X=V
- *BackTo(X)* is the farthest variable to which we backtracked since the last attempt to instantiate X

Now, some constraint checks can be omitted:



Backmarking in practice

N-queens problem



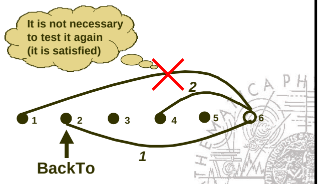
1. Queens in rows are allocated to columns.
2. Latest choice level is written next to chessboard (BackTo). At beginning 1s.
3. Farthest conflict queen at each position (MarkTo). At beginning 1s.
4. 6th queen cannot be allocated!
5. Backtrack to 5, change BackTo.
6. When allocating 6th queen, all the positions are still wrong (MarkTo<BackTo).

Note:
backmarking can be combined with backjumping (for free)

Consistency check for backmarking

Only the constraints where any value is changed are re-checked, and the farthest conflicting level is computed.

```
procedure consistent(X/V, Labelled, Constraints, Level)
    for each Y/VY/LY in Labelled such that LY³BackTo(X) do
        % only possible changed variables Y are explored
        % in the increasing order of LY (first the oldest one)
        if X/V is not compatible with Y/VY using Constraints then
            Mark(X,V) ¬ LY
            return fail
        end if
    end for
    Mark(X,V) ¬ Level-1
    return true
end consistent
```



Algorithm backmarking

```
procedure BM(Unlabelled, Labelled, Constraints, Level)
  if Unlabelled = {} then return Labelled
  pick first X from Unlabelled        % fix order of variables
  for each value V from D_X do
    if Mark(X,V) ³ BackTo(X) then % re-check the value
      if consistent(X/V, Labelled, Constraints, Level) then
        R ¬ BM(Unlabelled-{X}, Labelled È{X/V/Level}, Constraints, Level+1)
        if R ¹ fail then return R     % solution found
      end if
    end if
  end for
  BackTo(X) ¬ Level-1                 % jump will be to the previous variable
  for each Y in Unlabelled do         % tell everyone about the jump
    BackTo(Y) ¬ min {Level-1, BackTo(Y)}
  end for
  return fail     % return to the previous variable
end BM
```


Tree search and heuristics

Observation 1:
The search space for real-life problems is so huge that it cannot be fully explored.

Heuristics **- a guide of search**
- they recommend a value for assignment
- quite often leads to solution

What to do upon a failure of the heuristics?
BT cares about the end of search (a bottom part of the search tree)
- so it rather repairs later assignments than the earliest ones
- it assumes that the heuristic guides it well in the top part

Observation 2:
The heuristics are less reliable in the earlier parts of the search (as search proceeds, more information for better decision is available).

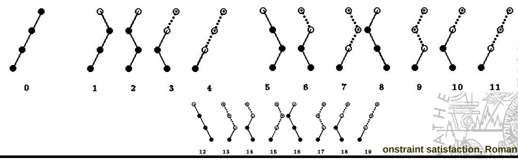
Observation 3:
The number of heuristic violations is usually small.

Limited Discrepancy Search

Discrepancy **= heuristic is not followed**
(a value different from the heuristic is chosen)

Idea of *Limited Discrepancy Search* (LDS):
- first, follow the heuristic
- when a failure occurs then explore the paths when the heuristic is not followed maximally once (start with earlier violations)
- after next failure occurs then explore the paths when the heuristic is not followed maximally twice...

Example:
the heuristic proposes to use the left branches



Algorithm LDS (Harvey, Ginsberg 1995)

```
procedure LDS-PROBE(Unlabelled,Labelled,Constraints,D)
    if Unlabelled = {} then return Labelled
    select X in Unlabelled
    Values_X ¬ D_X - {values inconsistent with Labelled using Constraints}
    if Values_X = {} then return fail
    else select HV in Values_X using heuristic
        if D=0 then return LDS-PROBE(Unlabelled-{X}, LabelledÈ{X/HV}, Constraints, 0)
        for each value V from Values_X -{HV} do
            R ¬ LDS-PROBE(Unlabelled-{X}, LabelledÈ{X/V}, Constraints, D-1)
            if R¹ fail then return R
        end for
        return LDS-PROBE(Unlabelled-{X}, LabelledÈ{X/HV}, Constraints, D)
    end if
end LDS-PROBE

procedure LDS(Variables,Constraints)
    for D=0 to |Variables| do             % D is a number of allowed discrepancies
        R ¬ LDS-PROBE(Variables,{},Constraints,D)
        if R¹ fail then return R
    end for
    return fail
end LDS
```

Foundations of constraint satisfaction

3

Roman Barták
Charles University in Prague

bartak@ktiml.mff.cuni.cz
http://ktiml.mff.cuni.cz/~bartak

Introduction to consistency techniques

So far we used constraints in a passive way (as a test) …
…in the best case we analysed the reason of the conflict.

Cannot we use the constraints in a more active way?

Example:

A in 3..7, B in 1..5	the variables' domains
A<B	the constraint

many inconsistent values can be removed

we get A in 3..4, B in 4..5

Note: it does not mean that all the remaining combinations of the values are consistent (for example A=4, B=4 is not consistent)

How to remove the inconsistent values from the variables' domains in the constraint network?

Node consistency (NC)

Unary constraints are converted into variables' domains.

Definition:

- *The vertex* representing the variable X is *node consistent* iff every value in the variable's domain D_x satisfies all the unary constraints imposed on the variable X.
- *CSP* is *node consistent* iff all the vertices are node consistent.

Algorithm NC

```
procedure NC(G)
    for each variable X in nodes(G)
        for each value V in the domain D_X
            if unary constraint on X is inconsistent with V then
                        delete V from D_X
        end for
    end for
end NC
```


Arc consistency (AC)

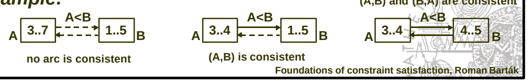
Since now we will assume binary CSP only
i.e. a constraint corresponds to an arc (edge) in the constraint network.

Definition:

- *The arc* (V_i,V_j) is *arc consistent* iff for each value *x* from the domain D_i there exists a value *y* in the domain D_j such that the valuation $V_i = x$ a $V_j = y$ satisfies all the binary constraints on V_i, V_j.

  Note: The concept of arc consistency is directional, i.e., arc consistency of (V_i,V_j) does not guarantee consistency of (V_j,V_i).
- *CSP* is *arc consistent* iff every arc (V_i,V_j) is arc consistent (in both directions).

Example:



Algorithm for arc revisions

How to make (V_i,V_j) arc consistent?

Delete all the values *x* from the domain D_i that are inconsistent with all the values in D_j (there is no value *y* in D_j such that the valuation $V_i = x$, $V_j = y$ satisfies all the binary constrains on V_i a V_j).

Algorithm of arc revision

```
procedure REVISE((i,j))
    DELETED ¬ false
    for each X in D_i do
        if there is no such Y in D_j such that (X,Y) is consistent, i.e.,
                (X,Y) satisfies all the constraints on V_i, V_j then
            delete X from D_i
            DELETED ¬ true
        end if
    end for
    return DELETED
end REVISE
```

The procedure also reports the deletion of some value.

Algorithm AC-1 (Mackworth 1977)

How to make CSP arc consistent?

Do revision of every arc.

But this is not enough! Pruning the domain may make some already revised arcs inconsistent again.

A<B, B<C: (<u>3..7</u>,*1..5*,1..5) (*3..4*,<u>1..5</u>,1..5) (3..4,<u>4..5</u>,*1..5*) (3..4,*4*,<u>1..5</u>) (<u>3..4</u>,*4*,5) (3,4,5)

Thus the arc revisions will be repeated until any domain is changed.

Algorithm AC-1

```
procedure AC-1(G)
    repeat
        CHANGED ¬ false
        for each arc (i,j) in G do
            CHANGED ¬ REVISE((i,j)) or CHANGED
        end for
    until not(CHANGED)
end AC-1
```

What is wrong with AC-1?

If a single domain is pruned then revisions of all the arcs are repeated even if the pruned domain does not influence most of these arcs.

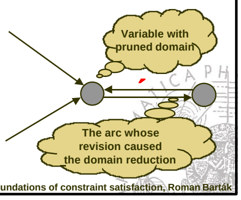
What arcs should be reconsidered for revisions?

The arcs whose consistency is affected by the domain pruning

i.e., the arcs pointing to the changed variable.

We can omit one more arc!

Omit the arc running out of the variable whose domain has been changed (this arc is not affected by the domain change).



Algorithm AC-2 (Mackworth 1977)

A generalised version of the Waltz's labelling algorithm.

In every step, the arcs going back from a given vertex are processed (i.e. a sub-graph of visited nodes is AC)

Algorithm AC-2

```
procedure AC-2(G)
   for i ← 1 to n do                          % n is a number of variables
      Q ← {(i,j) | (i,j)∈arcs(G), j<i}       % arcs for the base revision
      Q' ← {(j,i) | (i,j)∈arcs(G), j<i}      % arcs for re-revision
      while Q non empty do
         while Q non empty do
            select and delete (k,m) from Q
            if REVISE((k,m)) then
               Q' ← Q' ∪ {(p,k) | (p,k)∈arcs(G), p≤i, p≠m }
         end while
         Q ← Q'
         Q' ← empty
      end while
   end for
end AC-2
```


Algorithm AC-3 (Mackworth 1977)

Re-revisions can be done more elegant than in AC-2.

1) one queue of arcs for (re-)revisions is enough

2) only the arcs affected by domain reduction are added to the queue (like AC-2)

Algorithm AC-3

```
procedure AC-3(G)
   Q ← {(i,j) | (i,j)∈arcs(G), i≠j}     % queue of arcs for revision
   while Q non empty do
      select and delete (k,m) from Q
      if REVISE((k,m)) then
         Q ← Q ∪ {(i,k) | (i,k)∈arcs(G), i≠k, i≠m}
      end if
   end while
end AC-3
```

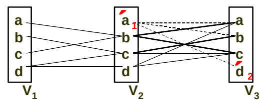
AC-3 is the most widely used consistency algorithm but it is still not optimal.

Looking for (and remembering of) the support

Observation (AC-3):

Many pairs of values are tested for consistency in every arc revision.

These tests are repeated every time the arc is revised.



1. When the arc V_2,V_1 is revised, the value *a* is removed from domain of V_2.

2. Now the domain of V_3, should be explored to find out if any value *a,b,c,d* loses the support in V_2.

Observation:

The values *a,b,c* need not be checked again because they still have a support in V_2 different from *a*.

The support set for $a \in D_i$ is the set $\{<j,b> \mid b \in D_j, (a,b) \in C_{i,j}\}$

Cannot we compute the support sets once and then use them during re-revisions?

Computing support sets

A set of values supported by a given value (if the value disappears then these values lost one support), and a number of own supporters are kept.

Computing and counting supporters

```
procedure INITIALIZE(G)
   Q ← {} , S ← {}                              % emptying the data structures
   for each arc (Vi,Vj) in arcs(G) do
      for each a in Di do
         total ← 0
         for each b in Dj do
            if (a,b) is consistent according to the constraint Ci,j then
               total ← total + 1
               Sj,b ← Sj,b ∪ {<i,a>}
            end if
         end for
         counter[(i,j),a] ← total
         if counter[(i,j),a] = 0 then
            delete a from Di
            Q ← Q ∪ {<i,a>}
         end if
      end for
   end for
   return Q
end INITIALIZE
```

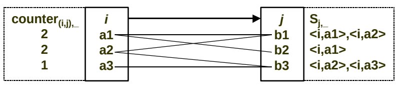
Sj,b - a set of pairs <i,a> such that <j,b> supports them

counter[(i,j),a] - number of supports for the value *a* from Di in the variable Vj

Computing supports and how to use them

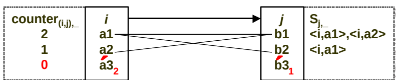
Situation:

we have just processed the arc (i,j) in INITIALIAZE



Using the support sets:

1. Let b3 is deleted from the domain of j (for some reason).
2. Look at $S_{j,b3}$ to find out the values that were supported by b3 (i.e. <i,a2>,<i,a3>).
3. Decrease the counter for these values (i.e. tell them that they lost one support).
4. If any counter is zero (a3) then delete the value and repeat the procedure with the respective value (i.e., go to 1).



Algorithm AC-4 (Mohr, Henderson 1986)

The algorithm AC-4 has the optimal worst case!

Algorithm AC-4

```
procedure AC-4(G)
    Q ← INITIALIZE(G)
    while Q non empty do
        select and delete any pair <j,b> from Q
        for each <i,a> from S_j,b do
            counter[(i,j),a] ← counter[(i,j),a] - 1
            if counter[(i,j),a] = 0 & "a" is still in D_i then
                    delete "a" from D_i
                    Q ← Q ∪ {<i,a>}
            end if
        end for
    end while
end AC-4
```

Unfortunately the average efficiency is not so good

… plus there is a big memory consumption!

Other arc consistency algorithms

AC-5 (Hentenryck, Deville, Teng 1992)
- a generic arc-consistency algorithm
- can be reduced both to AC-3 and AC-4
- exploits semantic of the constraint
  functional, anti-functional, and monotonic constraints

AC-6 (Bessiere 1994)
- improves memory complexity and average time complexity of AC-4
- keeps one support only, the next support is looked for when the current support is lost

AC-7 (Bessiere, Freuder, Regin 1999)
- based on computing supports (like AC-4 and AC-6)
- exploits symmetry of the constraint

Directional arc consistency (DAC)

Observation 1: AC has a directional character but CSP is not directional.

Observation 2: AC has to repeat arc revisions; the total number of revisions depends on the number of arcs but also on the size of domains (while cycle).

Is it possible to weaken AC in such a way that every arc is revised just once?

Definition: CSP is *directional arc consistent* using a given order of variables iff every arc (i,j) such that i<j is arc consistent.

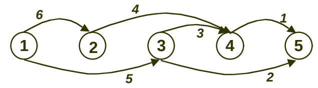
Again, every arc has to be revised, but revision in one direction is enough now.

Algorithm DAC-1

1) Consistency of the arc is required just in one direction.

2) Variables are ordered

⇒ there is no directed cycle in the graph!



If the arc are explored in a „good" order, no revision has to be repeated!

Algorithm DAC-1

```
procedure DAC-1(G)
    for j = |nodes(G)| to 1 by -1 do
        for each arc (i,j) in G such that i<j do
            REVISE((i,j))
        end for
    end for
end DAC-1
```


How to use DAC

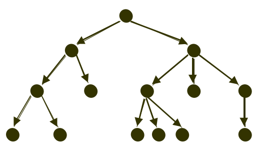
AC visibly covers DAC (if CSP is AC then it is DAC as well)

So, is DAC useful?
- DAC-1 is surely much faster than any AC-x
- there exist problems where DAC is enough

Example: If the constraint graph forms a tree then DAC is enough to solve the problem without backtracks.

How to order the vertices for DAC?

How to order the vertices for search?



1. Apply DAC in the order from the root to the leaf nodes.

2. Label vertices starting from the root.

DAC guarantees that there is a value for the child node compatible with all the parents.

Relation between DAC and AC

Observation: CSP is arc consistent iff for some order of the variables, the problem is directional arc consistent in both directions.

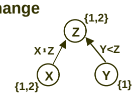
Is it possible to achieve AC by applying DAC in both primal and reverse direction?

In general NO, but …

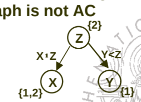
Example:

X in {1,2}, Y in {1}, Z in {1,2}, X≠Z, Y<Z

using the order X,Y,Z there is no domain change



using the order Z,Y,X, the domain of Z is changed but the graph is not AC



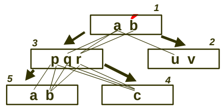
However if the order Z,Y,X is used then we get AC!

From DAC to AC for tree-structured CSP

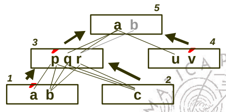
If we apply DAC to tree-structured CSP first using the order from the root to the leaf nodes and second in the reverse direction then we get (full) arc consistency.

Proof:

the first run of DAC ensures that any value in the parent node has a support (a compatible value) in all the child nodes



if any value is deleted during *the second run of DAC* (in the reverse direction) then this value does not support any value in the parent node (the values in the parent node does not lose any support)



together: every value has some support in the child nodes (the first run) as well as in the parent node (the second run), i.e., we have AC

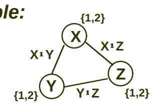
Is arc consistency enough?

By using AC we can remove many incompatible values

– Do we get a solution?
– Do we know that there exists a solution?

Unfortunately, the answer to both above questions is NO!

Example:



CSP is arc consistent but there is no solution

So what is the benefit of AC?

Sometimes we have a solution after AC

- any domain is empty ⇒ no solution exists
- all the domains are singleton ⇒ we have a solution

In general, AC prunes the search space.

Consistency techniques in practice

N-ary constraints are processed directly!

The **constraint C_Y is arc consistent** iff for every variable *i* constrained by C_Y and for every value $v \in D_i$ there is an assignment of the remaining variables in C_Y such that the constraint is satisfied.

Example: A+B=C, A in 1..3, B in 2..4, C in 3..7 is AC

Constraint semantics is used!

Interval consistency

working with intervals rather than with individual values

interval arithmetic

Example: after change of A we compute A+B ⇒ C, C-A ⇒ B

bounded consistency

only lower and upper bound of the domain are propagated

Such techniques do not provide full arc consistency!

It is possible to use different levels of consistency for different constraints!

Base propagation algorithm

Based on generalisation of AC-3.

Repeat constraint revisions until any domain is changed.

```
procedure AC-3(C)
    Q ← C                % a list of constraints for revision
    while Q non empty do
        select and delete c from Q
        REVISE(c,Q)
    end while
end AC-3
```

The REVISE procedure is customised for each constraint.

we get algorithms with various consistency levels

Constraint planning

How to choose the order of constraints for revisions (a queue Q)?

Event driven programming

event = domain change

REVISE generates new events that evoke further filtering

Design of consistency algorithms

The user can often define the code of REVISE procedure.

How to do it?

1) Decide about the event to evoke the filtering

when the domain of involved variable is changed

- whenever the domain changes
- when minimum/maximum bound is changed
- when the variable becomes singleton

different suspensions for different variables

Example: A<B filtering evoked after change of min(A) or max(B)

- directional consistency

2) Design the filtering algorithm for the constraint

the result of filtering is the change of domains

more filtering procedures for a single constraint are allowed

Example: A<B

min(A): B in min(A)+1..sup,   max(B): A in inf..max(B)-1

Definition of a constraint (SICStus Prolog)

How to describe propagation through A<B?

bound consistency is enough for full consistency!

```
less_then(A,B):-
  fd_global(a2b(A,B),no_state,[min(A)]),
  fd_global(b2a(A,B),no_state,[max(B)]).

dispatch_global(a2b(A,B),S,S,Actions):-
  fd_min(A,MinA), fd_max(A,MaxA), fd_min(B,MinB),
  (MaxA<MinB ->
     Actions = [exit]
  ;  LowerBoundB is MinA+1,
     Actions = [B in LowerBoundB..sup]).

dispatch_global(b2a(A,B),S,S,Actions):-
  fd_max(A,MaxA), fd_min(B,MinB), fd_max(B,MaxB),
   (MaxA<MinB ->
     Actions = [exit]
  ;  UpperBoundA is MinB-1,
     Actions = [A in inf..UpperBoundA]).
```

Foundations of constraint satisfaction 4

Roman Barták
Charles University in Prague

bartak@ktiml.mff.cuni.cz
http://ktiml.mff.cuni.cz/~bartak

Path consistency (PC)

How to strengthen the consistency level?

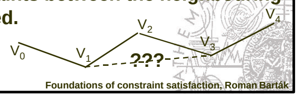
More constraints are assumed together!

Definition:

- *The path* ($V_0, V_1, \ldots, V_m$) is *path consistent* iff for every pair of values $x \in D_0$ a $y \in D_m$ satisfying all the binary constraints on V_0, V_m there exists an assignment of variables $V_1, \ldots, V_{m-1}$ such that all the binary constraints between the neighbouring variables V_i, V_{i+1} are satisfied.
- CSP is *path consistent* iff every path is consistent.

Attention!

Path consistency does not guarantee that all the constraints among the variables on the path are satisfied; only the constraints between the neighbouring variables must be satisfied.



PC and paths of length 2 (Montanari)

It is not very practical to ensure consistency of all paths

fortunately, only the paths of length 2 can be explored!

Theorem: CSP is PC iff every path of length 2 is PC.

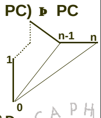
Proof:

1) PC $\Rightarrow$ paths of length 2 are PC

2) (paths of length 2 are PC $\Rightarrow$ $\forall$N paths of length N are PC) $\Rightarrow$ PC

induction using the path length

- a) N=2 visibly satisfied
- b) N+1 (proposition already holds for N)
  - i) take arbitrary N+1 vertices $V_0, V_1, \ldots, V_n$
  - ii) take arbitrary pair of compatible values $x_0 \in D_0$ a $x_n \in D_n$
  - iii) from a) we can find $x_{n-1} \in D_{n-1}$ s.t. constraints $C_{0,n-1}$ a $C_{n-1,n}$ hold
  - iv) from the induction we can find the values for $V_0, V_1, \ldots, V_{n-1}$



Foundations of constraint satisfaction, Roman Barták

Relation between PC and AC

Does PC subsumes AC (i.e. if CSP is PC, is it AC as well)?

- the arc (i, j) is consistent (AC) if the path (i,j,i) is consistent (PC)
- thus PC implies AC

Is PC stronger than AC (is there any CSP that is AC but not PC)?

Example: X in {1,2}, Y in {1,2}, Z in {1,2}, $X \neq Z$, $X \neq Y$, $Y \neq Z$

it is AC, but not PC (X=1, Z=2 cannot be extended to X,Y,Z)

AC removes incompatible values from the domains, what will be done in PC?

- PC removes pairs of values
- PC makes constraints explicit (A<B,B<C $\Rightarrow$ A+1<C)
- a unary constraint = a variable's domain

Foundations of constraint satisfaction, Roman Barták

A matrix representation of constraints

In PC we need to exclude the pairs of values

↳ the constraints must be represented in explicit form

Binary constraint = {0,1}-matrix

0 - the values are incompatible

1 - the values are compatible

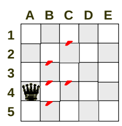
Example:

5-queens problem

the constraint between queens *i*, *j*: $r(i) \neq r(j)$ & $|i-j| \neq |r(i)-r(j)|$

a matrix for queens A(1), B(2)

```
00111
00011
10001
11000
11100
```



a matrix for queens A(1), C(3)

```
01011
10101
01010
10101
11010
```

Foundations of constraint satisfaction, Roman Barták

Operations over the constraints

Intersection R_{ij} & R'_{ij}

bitwise AND

A<B	&	*A≥B-1*	→	*B-1≤A<B*
011		110		010
001	&	111	=	001
000		111		000

Composition $R_{ik} * R_{kj} \rightarrow R_{ik}$

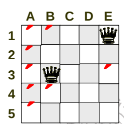
binary matrix multiplication

A<B	*	*B<C*	→	*A<C-1*
011		011		001
001	*	001	=	000
000		000		000

The induced constraint is joined with the original constraint

R_{ij} & $(R_{ik} * R_{kj}) \rightarrow R_{ij}$

R_{25}	&	(R_{21}	*	R_{15})	→	R_{25}
01101		00111		01110		01101
10110		00011		10111		10110
11011	&	10001	*	11011	=	01010
01101		11000		11101		01101
10110		11100		01110		10110



Notes:

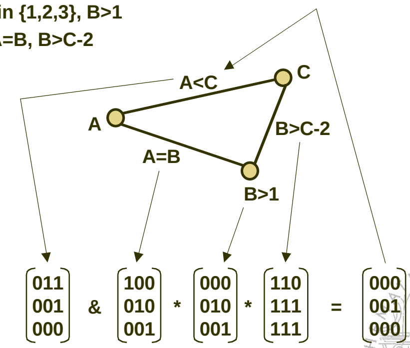
$R_{ij} = R^T_{ji}$, R_{ii} is a diagonal matrix representing the domain

REVISE((i,j)) from AC is equivalent to $R_{ii} \leftarrow R_{ii}$ & $(R_{ij} * R_{jj} * R_{ji})$

Foundations of constraint satisfaction, Roman Barták

Composing the constraints on the path

A,B,C in {1,2,3}, B>1
A<C, A=B, B>C-2



Algorithm PC-1 (Mackworth 1977)

How to make the path (i,k,j) consistent?

R_{ij} ¬ R_{ij} & (R_{ik} * R_{kk} * R_{kj})

How to make a CSP consistent?

Repeated revisions of all paths (of length 2) while any domain changes.

Algorithm PC-1

procedure PC-1(Vars,Constraints)
 n ¬ |Vars|, Y^n ¬ Constraints
 repeat
  Y^0 ¬ Y^n
  for k = 1 to n do
   for i = 1 to n do
    for j = 1 to n do
     Y^k_{ij} ¬ Y^{k-1}_{ij} & (Y^{k-1}_{ik} * Y^{k-1}_{kk} * Y^{k-1}_{kj})
 until Y^n=Y^0
 Constraints ¬ Y^0
end PC-1

If we use
Y^k_{ii} ¬ Y^{k-1}_{ii} & (Y^{k-1}_{ik} * Y^{k-1}_{kk} * Y^{k-1}_{ki})
then we get AC-1

How to improve PC-1?

Is there any inefficiency in PC-1?

just a few „bits“
- it is not necessary to keep all copies of Y^k
  one copy and a bit indicating the change is enough
- some operations produce no modification (Y^k_{kk} = Y^{k-1}_{kk})
- half of the operations can be removed (Y_{ji} = Y^T_{ij})

the grand problem
- after domain change all the paths are re-revised
  it is enough to revise just the influenced paths

Algorithm of path revision

procedure REVISE_PATH((i,k,j))
 Z ¬ Y_{ij} & (Y_{ik} * Y_{kk} * Y_{kj})
 if Z=Y_{ij} then return false
 Y_{ij} ¬ Z
 return true
end REVISE_PATH

If the domain is pruned
then the influenced
paths will be revised.

Which paths are influenced by the revision?

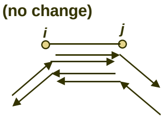
Because Y_{ji} = Y^t_{ij} it is enough to revise only the paths (i,k,j) where i£j.
Let the domain of the constraint (i,j) is changed when revising (i,k,j):

Situation a: i<j

all the paths containing (i,j) or (j,i) must be re-revised
the paths (i,j,j), (i,i,j) are not revised again (no change)

S_a = {(i,j,m) | i £ m £ n & m¹j}
 È {(m,i,j) | 1 £ m £ j & m¹i}
 È {(j,i,m) | j < m £ n}
 È {(m,j,i) | 1 £ m < i}

| S_a | = 2n-2



Situation b: i=j

all the paths containing *i* in the middle of the path are re-revised
the paths (i,i,i) and (k,i,k) are not revised again

S_b = {(p,i,m) | 1 £ m £ n & 1 £ p £ m} - {(i,i,i),(k,i,k)}

| S_b | = n*(n-1)/2 - 2

Algorithm PC-2 (Mackworth 1977)

Paths in one direction only (attention, this is not DPC!)
After every revision, the affected paths are re-revised

Algorithm PC-2

procedure PC-2(G)
 n ¬ |nodes(G)|
 Q ¬ {(i,k,j) | 1 £ i £ j £ n & i¹k & j¹k}
 while Q non empty do
  select and delete (i,k,j) from Q
  if REVISE_PATH((i,k,j)) then
   Q ¬ Q È RELATED_PATHS((i,k,j))
 end while
end PC-2

procedure RELATED_PATHS((i,k,j))
 if i<j then return S_a else return S_b
end RELATED_PATHS

Other path consistency algorithms

PC-3 (Mohr, Henderson 1986)
- based on computing supports for a value (like AC-4)
- this algorithm is not sound!
  If the pair (*a*,*b*) at the arc (i,j) is not supported by another variable, then *a* is removed from D_i and *b* is removed from D_j.

PC-4 (Han, Lee 1988)
- correction of the PC-3 algorithm
- based on computing supports of pairs (b,c) at arc (i,j)

PC-5 (Singh 1995)
- uses the ideas behind AC-6
- only one support is kept and a new support is looked for when the current support is lost

Drawbacks of path consistency

Memory consumption

- because PC eliminates pairs of values, we need to keep all the compatible pairs extensionally, e.g. using {0,1}-matrix

Bad ratio strength/efficiency

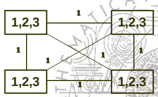
- PC removes more (or same) inconsistencies than AC, but the strength/efficiency ratio is much worse than for AC

Modifies the constraint network

- PC adds redundant arcs (constraints) and thus it changes connectivity of the constraint network
- this complicates using heuristics derived from the structure of the constraint network (like tightness, graph width etc.)

PC is still not a complete technique

- A,B,C,D in {1,2,3}
  $A \neq B$, $A \neq C$, $A \neq D$, $B \neq C$, $B \neq D$, $C \neq D$
  is PC but has not solution



Half way between AC and PC

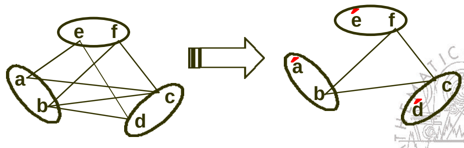
Can we make an algorithm:

- stronger than AC,
- without drawbacks of PC (memory consumption, changing the constraint network)?

Restricted path consistency (Berlandier 1995)

based on AC-4 (uses the support sets)

as soon as a value has only one support in another variable, PC is evoked for this pair of values



k-consistency

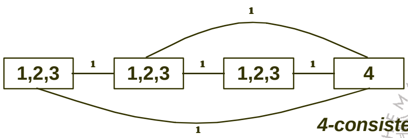
Is there a common formalism for AC and PC?

AC: a value is extended to another variable

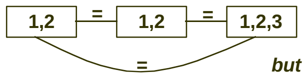
PC: a pair of values is extended to another variable

… we can continue

Definition: CSP is k-consistent iff any consistent valuation of (k-1) different variables can be extended to a consistent valuation of one additional variable.



Strong k-consistency



3-consistent graph

but not 2-consistent graph!

Definition: CSP is strongly k-consistent iff it is j-consistent for every $j \leq k$.

Visibly: strong k-consistency $\Rightarrow$ k-consistency

Moreover: strong k-consistency $\Rightarrow$ j-consistency $\forall j \leq k$

In general: $\neg$ k-consistency $\Rightarrow$ strong k-consistency

NC = strong 1-consistency = 1-consistency

AC = (strong) 2-consistency

PC = (strong) 3-consistency

sometimes we call NC+AC+PC together *strong path consistency*

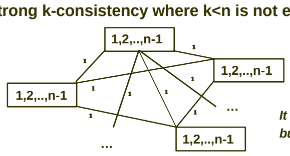
What *k-consistency* is enough?

Assume that the number of vertices is *n*. What level of consistency do we need to find out the solution?

Strong n-consistency for graphs with *n* vertices!

n-consistency is not enough - see the previous example

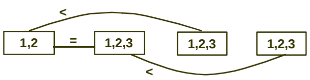
strong k-consistency where k<n is not enough as well



graph with n vertices
domains 1..(n-1)

It is strongly k-consistent for k<n but it has no solution

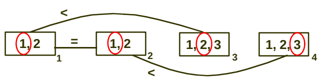
And what about this graph?



(D)AC is enough! Because this a tree..

Backtrack-free search

Definition: CSP is solved using backtrack-free search if for some order of variables we can find a value for each variable compatible with the values of already assigned variables.



How to find out a sufficient consistency level for a given graph?

Some observations:

- variable must be compatible with all the "former" variables
  i.e., across the „backward" edges
- for *k* „backward" edges we need (*k*+1)-consistency
- let *m* be the maximum of backward edges for all the vertices, then strong (*m*+1)-consistency is enough
- the number of backward edges is different for different variable order
- of course, the order minimising *m* is looked for

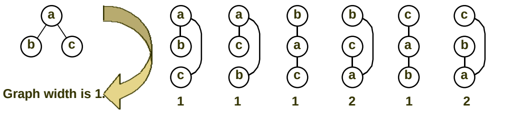
Graph width

Ordered graph is a graph with a given total order of vertices.

Vertex width in the ordered graph is the number of edges going back from this vertex.

Width of the ordered graph is maximum among the width of vertices.

Graph width is the maximum among the widths of its ordered graphs.




```
procedure MinWidthOrdering((V,E))
    Q ¬ {}
    while V not empty do
        N ¬ select and delete node with the smallest #edges from (V,E)
        enqueue N to Q
    return Q
end MinWidthOrdering
```

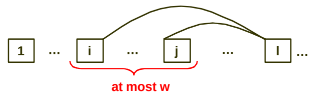

Graph width and consistency level

Theorem: Let w be the width of the constraint graph. If the constraint graph is strongly k-consistent for any k>w then there exists an order of variables giving backtrack-free solution.

Proof:

w is a graph width, i.e., there is some ordered graph of this width
thus the max. number of backward edges for each vertex is w
let us assign the variables in the order given by this ordered graph
now, if the variable is being labelled:

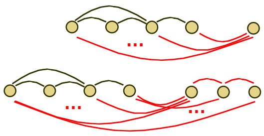
- we must find a value compatible with the labelled variables connected with the current variable
- let there is m such variables, then m £ w
- the graph is (m+1)-consistent, thus a compatible value must exist



(i,j)-consistency

k-consistency extends instantiation of (k-1) variables to a new variable, we remove (k-1)- tuples that cannot be extended to another variable.

We can do even more!



Definition: **CSP is (i,j)-consistent** iff every consistent instantiation of *i* variables can be extended to a consistent instantiation of any *j* additional variables.

CSP is strongly (i,j)-consistent, iff it is (k,j)-consistent for every k£i.

k-consistency	= (k-1,1) consistency
AC	= (1,1) consistency
PC	= (2,1) consistency

Inverse consistencies

Worst case time and space complexity of (i,j)-consistency is **exponential in *i***, moreover we need to **record forbidden *i*-tuples** extensionally (see PC).

What about keeping *i*=1 and increasing *j*?

We already have such an example:
RPC is (1,1)-consistency and sometimes (1,2)-consistency

Definition: (1,k-1)-consistency is called **k-inverse consistency**.

We remove values from the domain that cannot be consistently extended to additional (k-1) variables.

Inverse path consistency (PIC) = (1,2)-consistency

Neighbourhood inverse consistency (NIC) (Freuder , Elfe 1996)

We remove values of *v* that cannot be consistently extended to the set of variables directly linked to *v*.

Singleton consistencies

Can we strengthen any consistency technique?
YES! Let's assign a value and make the rest of the problem consistent.

Definition: **CSP P is singleton A-consistent** for some notion of A-consistency iff for every value *h* of any variable *X* the problem $P_{|X=h|}$ is A-consistent.

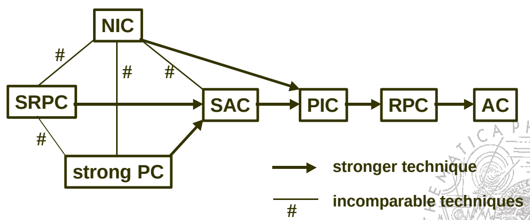
Features:

+ we remove only values from variable's domain - like NIC and RPC
+ easy implementation (meta-programming)
- not so good time complexity (be careful when using SC)

1) singleton A-consistency ³ A-consistency
2) A-consistency ³ B-consistency Þ
   singleton A-consistency ³ singleton B-consistency
3) singleton (i,j)-consistency > (i,j+1)-consistency (SAC>PIC)
4) strong (i+1,j)-consistency > singleton (i,j)-consistency (PC>SAC)

Consistency techniques at glance

NC = 1- consistency
AC = 2- consistency = (1,1)- consistency
PC = 3- consistency = (2,1)- consistency
PIC = (1,2)- consistency



Foundations of constraint satisfaction 5

Roman Barták
Charles University in Prague

bartak@ktiml.mff.cuni.cz
http://ktiml.mff.cuni.cz/~bartak

How to solve the constraint problems?

So far we have two methods:

- search
  - complete (finds a solution or proves its non-existence)
  - too slow (exponential)
    - explores "visibly" wrong valuations
- consistency techniques
  - usually incomplete (inconsistent values stay in domains)
  - pretty fast (polynomial)

Share advantages of both approaches - combine them!

– label the variables step by step (backtracking)
– maintain consistency after assigning a value

Do not forget about traditional solving techniques!

Linear equality solvers, simplex …
such techniques can be integrated to global constraints!

Core search procedure - depth-first search

The basic constraint satisfaction technology:

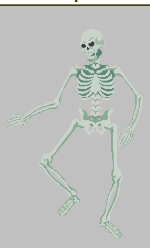
– label the variables step by step
  the variables are marked by numbers and labelled in a given order
– ensure consistency after variable assignment

A skeleton of search procedure

```
procedure Labelling(G)
    return LBL(G,1)
end Labelling

procedure LBL(G,cv)
    if cv>|nodes(G)| then return nodes(G)
    for each value V from D_cv do
        if consistent(G,cv) then
            R ¬ LBL(G,cv+1)
            if R ¹ fail then return R
        end if
    end for
    return fail
end LBL
```

A „hook" for consistency procedure



Look back techniques

"Maintain" consistency among the already labelled variables.

„look back" = look to already labelled variables

What's result of consistency maintenance among labelled variables?

a conflict (and/or its source - a violated constraint)

Backtracking is the basic method of look back.

Backward consistency checks

```
procedure AC-BT(G,cv)
    Q ¬ {(V_i,V_cv) in arcs(G),i<cv}      % arcs to labelled variables.
    consistent ¬ true
    while not Q empty & consistent do
        select and delete any arc (V_k,V_m) from Q
        consistent ¬ not REVISE(V_k,V_m)
    end while
    return consistent
end AC-BT
```

When a value is deleted, the domain is empty

Backjumping & comp. uses information about violated constraints.

Forward checking

It is better to prevent failures than to detect them only!

Consistency techniques can remove incompatible values for future (=not yet labelled) variables.

Forward checking ensures consistency between the currently labelled variables and the variables connected to it via constraints.

Forward consistency checks

```
procedure AC-FC(G,cv)
    Q ¬ {(V_i,V_cv) in arcs(G),i>cv}      % arcs to future variables
    consistent ¬ true
    while not Q empty & consistent do
        select and delete any arc (V_k,V_m) from Q
        if REVISE(V_k,V_m) then
            consistent ¬ not empty D_k
        end if
    end while
    return consistent
end AC-FC
```

Empty domain implies inconsistency

Partial look ahead

We can extend the consistency checks to more future variables!

The value assigned to the current variable can be propagated to all future variables.

Partial lookahead consistency checks

```
procedure DAC-LA(G,cv)
    for i=cv+1 to n do
        for each arc (V_i,V_j) in arcs(G) such that i>j & j³cv do
            if REVISE(V_i,V_j) then
                if empty D_i then return fail
        end for
    end for
    return true
end DAC-LA
```

Notes:

In fact DAC is maintained (in the order reverse to the labelling order).

Partial Look Ahead or DAC - Look Ahead

It is not necessary to check consistency of arcs between the future variables and the past variables (different from the current variable)!

Full look ahead

Knowing more about far future is an advantage!

Instead of DAC we can use a full AC (e.g. AC-3).

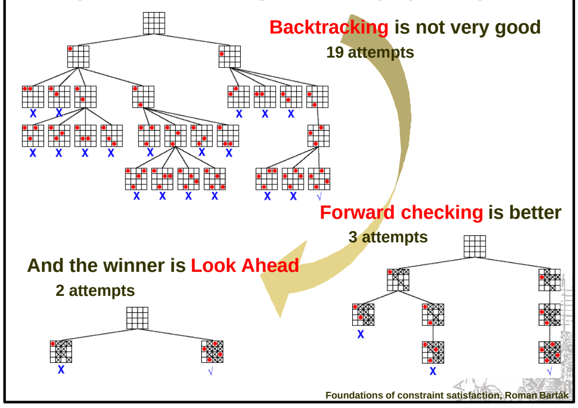
Full look ahead consistency checks

```
procedure AC3-LA(G,cv)
    Q ¬ {(Vi,Vcv) in arcs(G),i>cv}          % start with arcs going to cv
    consistent ¬ true
    while not Q empty & consistent do
        select and delete any arc (Vk,Vm) from Q
        if REVISE(Vk,Vm) then
            Q ¬ Q È {(Vi,Vk) | (Vi,Vk) in arcs(G),i¹k,i¹m,i>cv}
            consistent ¬ not empty Dk
        end if
    end while
    return consistent
end AC3-LA
```

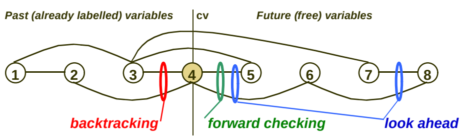
Notes:

- The arcs going to the current variable are checked exactly once.
- The arcs to past variables are not checked at all.
- It is possible to use other than AC-3 algorithms (e.g. AC-4)

Comparison of solving methods (4 queens)



Constraint propagation at glance



- Propagating through more constraints remove more inconsistencies (BT < FC < PLA < LA), of course it increases complexity of the step.
- Forward Checking does no increase complexity of backtracking, the constraint is just checked earlier in FC (BT tests it later).
- When using AC-4 in LA, the initialisation is done just once.
- Consistency can be ensured before starting search
  Algorithm MAC (Maintaining Arc Consistency)
  AC is checked before search and after each assignment
- It is possible to use stronger consistency techniques (e.g. use them once before starting search).

Variable ordering

Variable ordering in labelling influence significantly efficiency of solvers (e.g. in tree-structured CSP).

What variable ordering should be chosen in general?

FIRST-FAIL principle

„select the variable whose instantiation will lead to failure"

it is better to tackle failures earlier, they can be become even harder

- prefer the variables with smaller domain (dynamic order)
  - a smaller number of choices ~ lower probability of success
  - the dynamic order is appropriate only when new information appears during solving (e.g., in look ahead algorithms)

„solve the hard cases first, they may become even harder later"

- prefer the most constrained variables
  - it is more complicated to label such variables (it is possible to assume complexity of satisfaction of the constraints)
  - this heuristic is used when there is an equal size of the domains
- prefer the variables with more constraints to past variables
  - a static heuristic that is useful for look-back techniques

Value ordering

Order of values in labelling influence significantly efficiency (if we choose the right value each time, no backtrack is necessary).

What value order for the variable should be chosen in general?

SUCCEED FIRST principle

„prefer the values belonging to the solution"

if no value is part of the solution then we have to check all values

if there is a value from the solution then it is better to find it soon

SUCCEED FIRST does not go against FIRST-FAIL !

- prefer the values with more supporters
  - this information can be found in AC-4
- prefer the value leading to less domain reduction
  - this information can be computed using singleton consistency
- prefer the value simplifying the problem
  - solve approximation of the problem (e.g. a tree)

Generic heuristics are usually too complex for computation.

It is better to use problem-driven heuristics that propose the value!

Constraint optimisation

So far we have looked for feasible assignments only.

In many cases the users require optimal assignments where optimality is defined by an objective function.

Definition: Constraint Satisfaction Optimisation Problem (CSOP) consists of the standard CSP P and an objective function *f* mapping feasible solutions of P to numbers.

Solution to CSOP is a solution of P minimising / maximising the value of the objective function *f*.

To find a solution of CSOP we need in general to explore all the feasible valuations. Thus, the techniques capable to provide all the solutions of CSP are used.

Branch and bound

Branch and bound is perhaps the most widely used optimisation technique based on cutting sub-trees where there is no optimal (better) solution.

It is based on the **heuristic function** *h* that approximates the objective function.

- a sound heuristic for minimisation satisfies h(x)£f(x) [in case of maximisation f(x)£h(x)]
- a function closer to the objective function is better

During search, the sub-tree is cut if

- there is no feasible solution in the sub-tree
- there is no optimal solution in the sub-tree
  - *bound* £ h(x), where *bound* is max. value of feasible solution

How to get the bound?

It could be an objective value of the best solution so far.

BB and constraint satisfaction

Objective function can be modelled as a constraint

looking for the "optimal value" of *v*, s.t. v = f(x)

- first solution is found without any bound on *v*
- next solutions must be better then so far best (*v*<Bound)
- repeat until no more feasible solution exist

Algorithm Branch & Bound

```
procedure BB-Min(Variables, V, Constraints)
    Bound ¬ sup
    NewSolution ¬ fail
    repeat
        Solution ¬ NewSolution
        NewSolution ¬ Solve(Variables,Constraints È {V<Bound})
        Bound ¬ value of V in NewSolution (if any)
    until NewSolution = fail
    return Solution
end BB-Min
```


Some notes on branch and bound

Heuristic *h* is hidden in **propagation through the constraint** v = f(x).

Efficiency is dependent on:

- **a good heuristic** (good propagation of the objective function)
- **a good first feasible solution** (a good bound)
  - the initial bound can be given by the user to filter bad valuations

The optimal solution can be found fast

- **proof of optimality can be long** (exploring of the rest part of tree)

The optimality is often not required, **a good enough solution is OK.**

- BB can stop when reach a given limit of the objective function

Speed-up of BB: **both lower and upper bounds are used**

```
repeat
    TempBound ¬ (UBound+LBound) / 2
    NewSolution ¬ Solve(Variables,Constraints È {V£TempBound})
    if NewSolution=fail then
        LBound ¬ TempBound+1
    else
        UBound ¬ TempBound
until LBound = UBound
```


A motivation - robot dressing problem

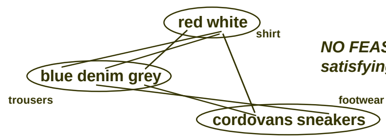
Dress a robot using minimal wardrobe and fashion rules.

Variables and domains:

- 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**.

First solution to the robot dressing problem

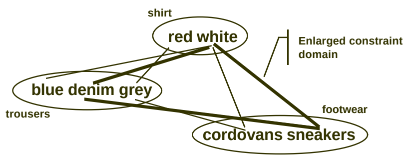
There is no feasible valuation but we need to dress robot!

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

Domain is defined by a unary constraint

All combinations are assumed feasible

Delete the constraint bounding the variable



Partial constraint satisfaction

First let us define a **problem space** as a partially ordered set of CSPs (PS,£), where P_1£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 ¡P,(PS,£),M,(N,S)ñ

- P is the original problem
- (PS,£) is a problem space containing P
- M is a metric on the problem space defining the problem distance
  - M(P,P') could be a 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 PCSP is a problem P' and its solution such that P'Î PS and M(P,P')<N. **A sufficient solution** is a solution s.t. M(P,P') £ S. **The optimal solution** is a solution with the minimal distance to P.

Partial constraint satisfaction in practice

When solving 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.

Second solution of the robot dressing problem

It is possible to assign a preference to each constraint to describe priorities of satisfaction of the constraints.

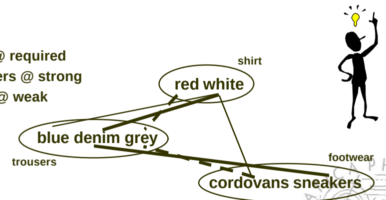
The preference describes a strict priority.

a stronger constraint is preferred to arbitrary number of weaker constraints

shirt x trousers @ required

footwear x trousers @ strong

shirt x footwear @ weak



Constraints marked by a preference make a hierarchy, thus we are speaking about constraint hierarchies.

Constraint hierarchies

Every constraint is labelled by a preference (the set of preferences is totally ordered)

- there is a special preference *required*, marking constraints that must be satisfied (hard constraints)
- the other constraints are preferential, their satisfaction is not required (soft constraints)

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

H_0 is a set of the required constraints (the label is removed)

H_1 is a set of the most preferred soft constraints

…

A solution to the hierarchy is an assignment satisfying all the required constraints and satisfying best the preferential constraints.

$S_{H,0} = \{\sigma \mid \forall c \in H_0,\ c\sigma \text{ holds}\}$

$S_H = \{\sigma \mid \sigma \in S_{H,0}\ \&\ \forall \omega \in S_{H,0}\ \neg\, better(\omega,\sigma,H)\}$

Comparators

Comparing the assignments according to a given hierarchy.

- anti-reflexive, transitive relation that *respects the hierarchy*
- if any assignment satisfies all the constraints till the level k, then every better assignment must satisfy these constraints as well

Error function $e(c,\sigma)$ - how good the constraint is satisfied

predicate error function (satisfied/violated)

metric error function - distance from solution, $e(X>5,\{X/3\}) = 2$

Local comparators

compare the assignments using the constraint individually

$locally\_better(\omega,\sigma,H) \equiv \exists k>0$

$\forall i<k\ \forall c \in H_i\ e(c,\omega)=e(c,\sigma)\ \&\ \forall c \in H_k\ e(c,\omega) \leq e(c,\sigma)\ \&\ \exists c \in H_k\ e(c,\omega)<e(c,\sigma)$

Global comparators

aggregate the individual errors at the level via the function g

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

weighted-sum, least-squares, and worst-case methods …

Why should we use CP?

Close to real-life (combinatorial) problems

- everyone uses constraints to specify problem properties
- real-life restriction can be naturally described using constraints

A declarative character

- concentrate on problem description rather than on solving

Co-operative problem solving

- unified framework for integration of various solving techniques
- simple (search) and sophisticated (propagation) techniques

Semantically pure

- clean and elegant programming languages
- roots in logic programming

Applications

- CP is not another academic framework, it is already used in many applications

Final notes

Constraints

- arbitrary relations over the problem variables
- express partial local information in a declarative way

Solution technology

- search combined with constraint propagation
- local search

It is easy to state combinatorial problems in terms of CSP

… but it is more complicated to design solvable models.

We still did not reach the Holy Grail of computer programming: *the user states the problem, the computer solves it.*

Constraint Programming is one of the closest approaches to the Holly Grail of programming!