Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.

Roman Barták
Charles University in Prague
bartak@ktiml.mff.cuni.cz
http://ktiml.mff.cuni.cz/~bartak

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°
- 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.

Example:
variables X₁…X₆
domain {0,1}
c₁: X₁+X₂+X₆=1
c₂: X₁-X₃+X₄=1
c₃: X₄+X₅-X₆>0
c₄: X₂+X₅-X₆=0

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:

A bit of history

Artificial Intelligence
Scene labelling (Waltz 1975)

Interactive graphics
Sketchpad (Sutherland 1963)

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

Operations research and discrete mathematics
NP-hard combinatorial problems
**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?*

**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**

*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*

**CP and others**

- various domains
- arbitrary constraints
- heterogeneous problems

**What is the course about?**

*Constraint satisfaction problems*

*Algorithms for solving constraint satisfaction problems*

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

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

Binary constraints

- Each CSP can be transformed to an equivalent binary CSP for systematic search

Projection technique (Montanary 1974):

- straightforward but bound consistency
- does not give an equivalent problem
- better efficiency
- weaker pruning

Hidden variable encoding

New dual variables for (non-binary) constraints.

- any constraint $c$ is translated to a dual variable $v_x$ with the domain consisting of compatible tuples

Example:

- variables $x_1, ..., x_6$ with domains {0,1}
- $c_1: x_1 \land x_2 \land x_3 = 1$
- $c_2: x_1 \land x_3 \land x_4 = 1$
- $c_3: x_1 \land x_5 = 0$
- $c_4: x_2 \land x_6 = 0$

Solving constraints by enumeration

Constraints are used only as a test assign values to variables ...

- 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.

Example:

<table>
<thead>
<tr>
<th>Variables</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1, x_2$</td>
<td>{0,1}</td>
</tr>
<tr>
<td>$y_1, y_2$</td>
<td>{0,1}</td>
</tr>
</tbody>
</table>

Procedure GT($X$:variables, $C$:constraints)

1. construct a first complete assignment of $X$
2. while $V$ does not satisfy all the constraints $C$ do
3. construct systematically a complete assignment next to $V$
4. if $V$ satisfies $C$ return $V$

Dual encoding

Swapping variables and constraints.

- each constraint $c$ is converted to a dual variable $v_x$ with the domain consisting of compatible tuples

Example:

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

<table>
<thead>
<tr>
<th>Variables</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1, x_2$</td>
<td>{0,1}</td>
</tr>
<tr>
<td>$y_1, y_2$</td>
<td>{0,1}</td>
</tr>
</tbody>
</table>
**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

- only “small” changes of the assignment are allowed
- next assignment should be “better” than previous
- assignments are not necessarily generated systematically

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

---

**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 variable in conflict and try to improve it
- neighbourhood is different values for the selected variable
- neighbourhood size = (D_i-1) = (d-1)

Procedure MC(Max_Moves)

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

**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:

```plaintext
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 minimizes the number of conflicts for V
    end if
    if v' ≠ current value of V then
      assign v' to V
      nb_moves ← nb_moves+1
    end if
  end while
  return s
```

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' ≠ current value of V then
      assign v' to V
      nb_moves ← nb_moves+1
    end if
  end while
  return s
```

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

```
procedure 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 consistent with V+{x/h} then
      R ← BT(X-x, V+{x/h}, C)
      if R ≠ fail then return R
    end for
  end if
  return fail
```

call BT(X, {}, C)

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

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 A≠1
- 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+C=D, C=5*E
- the fact that A≠2 is discovered 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)
**Conflict-directed backjumping in practice**

*N-queens problem*

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
```

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.

**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
    NoConflict ← true
    % indicator of a conflict
    % the level to which we will jump
    for each C in Constraints do
        if all variables from C are Labelled then
            NoConflict ← false
        end if
        J ← min(J, max(L | X in C & X/V/L in Labelled & L<Level))
    end for
    if NoConflict then return true
    end if
    R ← BJ(Unlabelled-{X},{X/V/Level}|
    if R
    % success or backjump
    end if
end consistent
```

**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)

```
1 2 3 4 5 6 7
```

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

**Algorithm backjumping**

```
procedure BJ(Unlabelled, Labelled, Constraints, PreviousLevel)
    if Unlabelled = {} then return Labelled
    call BJ(Variables,{},Constraints,0)
end BJ
```

**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 1 2 3
A  B  C
--+--+--
A +B C
```

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 node 1 2 3 node 1 2 3
A +  B +  C +  A +  
C +  D +  E +  C +  
B +  A +  B +  A +  
A +  B +  A +  A +  
E +  A +  B +  E +  
```

The vertex C (and the possible sub-graph connected to C) is not re-coloured.
Algorithm dynamic backtracking (Ginsberg 1990)

(procedure DB(Variables, Constraints))
Labelled ← ∅, Unlabelled ← Variables
while Unlabelled do
select X in Unlabelled
ValuesX ← ∅ - (values inconsistent with Labelled using Constraints)
if ValuesX = ∅ 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
end if
else
select V in ValuesX
Unlabelled ← Unlabelled - (X)
Labelled ← Labelled ∪ (X/V)
end if
end while
end DB

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:

Mark: BackTo
Yb is inconsistent with Xa (and consistent with all variables above X)

Mark: BackTo
Yb is consistent with Xa (and consistent with all variables above X)

Mark: BackTo
X is OK there

Mark: BackTo
Yb must be unchecked with these variables

Backmarking in practice

N-queens problem

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/V/Y/LY in Labelled such that LY: BackTo(X) do
if only possible changed variables Y are explored % in the increasing order of LY (first the oldest one)
if X/Y is not compatible with Y/V using Constraints then
Mark(X/V) ← LY
return fail
end if
end for
Mark(X/V) ← Level-1
end consistent

Algorithm backmarking

(procedure BM(Unlabelled, Labelled, Constraints, Level))
if Unlabelled ≠ ∅ then return Labelled
pick first X from Unlabelled % fix order of variables
% fix order of variables
for each value V from D do
if Mark(X/V) = BackTo(X) then % re-check the value
if consistent(X/V, Labelled, Constraints, Level) then
R ← BM(Unlabelled - (X), Labelled - (X/V), Constraints, Level+1)
if R = fail then return % solution found
end if
end if
end for
MarkTo<BackTo)
if E = {} then failure
else
let Y be the most recent variable in E
unassign Y (from Labelled) with eliminating explanation E-{Y}
end if
end for
select X in Unlabelled % in the increasing order of LY (first the oldest one)
% only possible changed variables Y are explored
end BM

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

Redundant computations:
- It is not necessary to repeat them because the change of B does not influence C.
- 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

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

Redundant computations:
- It is not necessary to repeat them because the change of B does not influence C.
- 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

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

Redundant computations:
- It is not necessary to repeat them because the change of B does not influence C.
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
if unary constraint on X is inconsistent with V then
   delete V from D
   return LDS-PROBE(Unlabelled-{X}, Labelled, Constraints, D)
else select HV in Values
   for each value V from Values
   do if D=0 then return LDS-PROBE(Unlabelled-{X}, Labelled, Constraints, D-1)
      fi
      return LDS-PROBE(Unlabelled-{X}, Labelled, Constraints, D)
   end for
   return fail
end for
end if
end for
return LDS-PROBE(Unlabelled-{X}, Labelled, Constraints, D)
end LDS-PROBE

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 represents 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
      if unary constraint on X is inconsistent with V then
         delete V from D
      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 \(V_i = x\) and \(V_j = y\) satisfies all the binary constraints on \(V_i, V_j\).
- CSP is arc consistent iff every arc \((V_i, V_j)\) is arc consistent (in both directions).

Example:
\[
\begin{align*}
&\text{A in 3..7, B in 1..5} & \text{the variables' domains} \\
&\text{A<B} & \text{the constraint} \\
&\text{many inconsistent values can be removed we get A in 3..4, B in 4..5} \\
&\text{Note: it does not mean that all the remaining combinations of the values are consistent (for example A=4, B=6 is not consistent)}
\end{align*}
\]

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

Arc consistency (AC) is an upper bound of node consistency.
Algorithm for arc revisions

How to make (V, V) arc consistent?
Delete all the values x from the domain D that are inconsistent with all the values in D (there is no value y in D such that the valuation V_i = x, V_j = y satisfies all the binary constraints on V_i, V_j).

Algorithm for arc revision

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

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-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: (3,7,1,5,1-5) (3,4,1,5,1-5) (3,4,4,1,5) (3,4,4,5) (3,4,5)
Thus the arc revisions will be repeated until any domain is changed.

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-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)

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.

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

Acadia University
Unfortunately the average efficiency is not so good
Is arc consistency enough?

Example:

\[
\begin{align*}
X &\rightarrow Y \\
Y &\rightarrow Z \\
X &\rightarrow Z
\end{align*}
\]

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 \(\Rightarrow\) no solution exists
- all the domains are singleton \(\Rightarrow\) we have a solution

In general, AC prunes the search space.
**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 if every path of length 2 is PC.

**Proof:**
1. PC → paths of length 2 are PC
2. (paths of length 2 are PC ⇒ ∀N paths of length N are PC) ⇒ PC

Induction using the path length
- N+1 visibly satisfied
- take arbitrary N+1 vertices $V_i, V_j, \ldots, V_N$

**Binary constraint = (0,1)-matrix**

Example:

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

Induction using the path length

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

**Algorithm PC-1 (Mackworth 1977)**

- How to make the path $(i,j,k)$ consistent?
  - $R_i \leftarrow R_i \land (R_j \land \neg R_k)$
- How to make a CSP consistent?
  - Repeated revisions of all paths (of length 2) while any domain changes.

**Composing the constraints on the path**

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

**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

**Operations over the constraints**

- Intersection $R_i \land R_j$
- Composition $R_i \cdot R_j \rightarrow R_k$

**Relation between PC and AC**

- Does PC subsume AC (i.e., if CSP is PC, is it AC as well)?
  - the arc $(i,j)$ is consistent (AC) if the path $(i,j)$ is consistent (PC)
  - thus PC implies AC

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

**Notes:**

- $R_i = R'_{ij}$, $R_i$ is a diagonal matrix representing the domain
- REVISE $(i,j)$ from AC is equivalent to $R_i \leftarrow R_i \land (R_j \land \neg R_k)$

**Composition $R_i \cdot R_j \rightarrow R_k$**

- $10101 \cdot 00011 = 00011$
- $11110 \cdot 10110 = 10110$
- $10110 \cdot 11001 = 11001$
- $01101 \cdot 11001 = 01101$
- $01101 \cdot 11011 = 01101$
- $01101 \cdot 11101 = 01101$

**Operations over the constraints**

- Intersection $R_i \land R_j$
- Composition $R_i \cdot R_j \rightarrow R_k$

**The induced constraint is joined with the original constraint**

- $R_i \land R_j \rightarrow R_k$
- $R_i \cdot R_j \rightarrow R_k$
- $R_i \leftarrow R_i \land R_j$
- $R_i \rightarrow R_i \land R_j$

**Notes:**

- $R_i = R'_{ij}$, $R_i$ is a diagonal matrix representing the domain
- REVISE $(i,j)$ from AC is equivalent to $R_i \leftarrow R_i \land (R_j \land \neg R_k)$

Foundations of Constraint Programming, Roman Barták
**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>B, A>C, A>D, B>C, B>D, C>D
- Is PC but has no 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 (Berlander 1993)**
- 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

---

**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 \(Y^i\)
  - One copy and a bit indicating the change is enough
  - Some operations produce no modification (\(Y^i_a = Y^i_{a+1}\))
  - Half of the operations can be removed (\(Y^i_a = Y^i_{a+1}\))

The grand problem
- After domain change all the paths are re-revised
  - It is enough to revise just the influenced paths
  - The paths \((i,j)\) where \(i<j\)

**Procedure REVISE PATH**

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

```
if REVISE_PATH((i,k,j)) then
  select and delete (i,k,j) from Q
  related_paths((i,k,j))
  end while
```

End REVISE PATH

**Algorithm of path revision**

- the grand problem
  - just a few "bits"
  - It is enough to revise just the influenced paths after domain change all the paths are re-revised
  - Half of the operations can be removed (\(Y^i_a = Y^i_{a+1}\))

**Is there any inefficiency in PC-1?**

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 \((i,j,m)\), \((j,i,m)\) are not revised again (no change)
- \(S_k = \{(j,i,m) | 1 \leq m \leq n \& (m,i)\} \bigcup \{(i,j,m) | 1 \leq m \leq n \& (m,j)\} \bigcup \{(m,i,j) | 1 \leq m < i\})\)
- \(|S_k| = 2m^2\)

**Situation b: \(i=j\)**
- All the paths containing / in the middle of the path are re-revised
- The paths \((i,j)\) and \((k,i,k)\) are not revised again
- \(S_k = \{(i,j,m) | 1 \leq m \leq n \& 1 \leq p \leq m \& (i,j,p)\} \bigcup \{(i,j),(k,i)\}\)
- \(|S_k| = n^2(n-1)/2 - 2\)

**Which paths are influenced by the revision?**

Because \(Y^i = Y^i\) it is enough to revise only the paths \((i,k,j)\) where \(i<j\).

**Other path consistency algorithms**

**PC-3 (Mohr, Henderson 1995)**
- 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
**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.

![4-consistent graph](image)

**Strong k-consistency**

\[
\begin{array}{ccc}
1,2 & = & 1,2 \\
1,2 & = & 1,2,3 \\
\end{array}
\]

3-consistent graph

but not 2-consistent graph!

**Definition:** CSP is strongly k-consistent iff it is j-consistent for every j ≤ k.

Visibly: strong k-consistency ⇒ k-consistency
Moreover: strong k-consistency ⇒ j-consistency ∀j ≤ k
In general: ¬k-consistency ⇒ 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

**Which 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

\[
\begin{array}{ccc}
1,2,...,n-1 & = & 1,2,...,n-1 \\
1,2,...,n-1 & = & 1,2,...,n-1 \\
\end{array}
\]

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

And what about this graph?

\[
\begin{array}{ccc}
1,2 & = & 1,2,3 \\
1,2 & = & 1,2,3 \\
\end{array}
\]

(D)AC is enough!

Because tree is tree.

**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 ≥ A-consistency
2) A-consistency ≥ B-consistency ⇒ singleton A-consistency ≥ singleton B-consistency
3) singleton (i,j)-consistency = (i,j)-consistency (SAC,PIC)
4) strong (i+1,j)-consistency = singleton (i,j)-consistency (PC,PIC)
Consistency techniques at glance

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

a stronger technique
incomparable techniques

SRPC
SAC
PIC
RPC
AC

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
    do
      if consistent(G,cv) then
        R ¬ LBL(G,cv+1)
        if R ≠ fail then return R
      end if
    end for
  return fail
end LBL

A “hook” for consistency

Backward consistency checks

procedure AC-BT(G,cv)
  Q ¬ {(V_i,V_cv) in arcs(G),i<cv}
  consistent ¬ true
  while consistent & Q non empty 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

Backjumping & comp. uses information about the 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_j) in arcs(G),i>cv}
  % arcs to future variables
  consistent ¬ true
  while consistent & Q non empty do
    select and delete any arc (V_i,V_j) from Q
    if REVISE(V_i,V_j) then
      consistent ¬ not empty D_i
    end if
  end while
  return consistent
end AC-FC

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 & j≥cv do
      if REVISE(V_i,V_j) then
        if empty D_i then return fail
      end if
    end for
    return true
  end for
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)
% start with arcs going to cv
consistent ← true
while consistent & Q non empty do
  select and delete any arc (V_iV_m) from Q
  if REVISE(V_iV_m) then
    Q ← Q \ {V_iV_m}
    if REVISE(V_iV_m) then
      Q ← Q \ {V_iV_m}
      consistent ← not empty Q
  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)**

Backtracking is not very good
19 attempts

And the winner is **Look Ahead**
2 attempts

**Constraint propagation at glance**

Past (already labelled) variables ⇒ CV ⇒ Future (free) variables

- Propagating through more constraints remove more inconsistencies
- Forward Checking does not 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)

**FIRST-SUCCEED principle**

- “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 backtrace is necessary).

What value ordering 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

**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 $h(x) \leq f(x)$ (in case of maximisation $f(x) \leq h(x)$)
- 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

$\text{bound} \leq h(x)$, where $\text{bound}$ 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 ($=\text{Bound}$)
- repeat until no more feasible solution exist

Algorithm Branch & Bound

```plaintext
procedure BB-Min(Variables, V, Constraints)
    Bound ← sup
    NewSolution ← fail
    repeat
        Solution ← NewSolution
        NewSolution ← Solve(Variables, Constraints ∪ {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

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 ∪ {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

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
2) less elegant wardrobe
3) no matching of shoes and shirt
4) do not wear shoes

Domain is defined by a unary constraint

All combinations are assumed feasible

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

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.
Third solution of the robot dressing problem

It is possible to assign a preference to each pair (tuple) in the constraint.
The task is to maximise the product of preferences for the assignment projections into all constraints.

<table>
<thead>
<tr>
<th>Pair</th>
<th>Preference</th>
</tr>
</thead>
<tbody>
<tr>
<td>shirt x trousers</td>
<td>red-grey (1), white-blue (1), white-denim (0.9)</td>
</tr>
<tr>
<td>footwear x trousers</td>
<td>sneakers-denim (1), cordovans-grey (1)</td>
</tr>
<tr>
<td>shirt x footwear</td>
<td>white-cordovans (0.8)</td>
</tr>
</tbody>
</table>

All other pairs have the value 0.1.

This Probabilistic CSP can be generalised into a Semiring-based CSP.

Why should we use CP?

Close to real-life (combinatorial) problems
- Everyone uses constraints to specify problem properties
- Real-life restrictions 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 Holy Grail of programming!