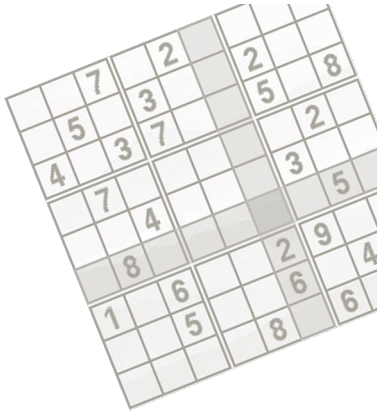


Introduction to Constraint Satisfaction

Roman Barták

Charles University, Prague (CZ)

On-line Materials

So far, we used constraints in a passive way (as a test).

In the best case we analysed the reason of the conflict.

Can 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?

We will assume binary CSPs only (recall binarization)

i.e. a constraint corresponds to an arc (edge) in the constraint network.

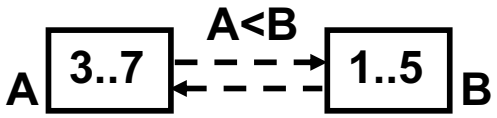
Definition:

- **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 assignment $V_i = x$ a $V_j = y$ satisfies all the binary constraints on V_i, V_j.

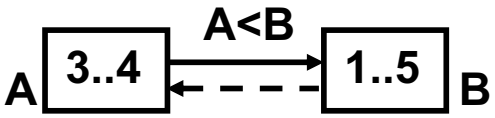
Note: The concept of arc consistency is directional, i.e., arc consistency of (V_i, V_j) does not guarantee consistency of (V_j, V_i).

- **CSP** is **arc consistent** iff every arc (V_i, V_j) is arc consistent (in both directions).

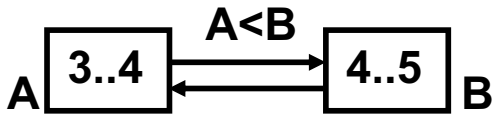
Example:



no arc is consistent



(A,B) is consistent



(A,B) and (B,A) are consistent

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 instantiation V_i = x, V_j = y satisfies all the binary constraints on V_i and V_j).

Algorithm of arc revision

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

The procedure also reports the deletion of some value.

How to make a CSP arc consistent?

- Do revision of every arc.

Beware, 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..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-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
```

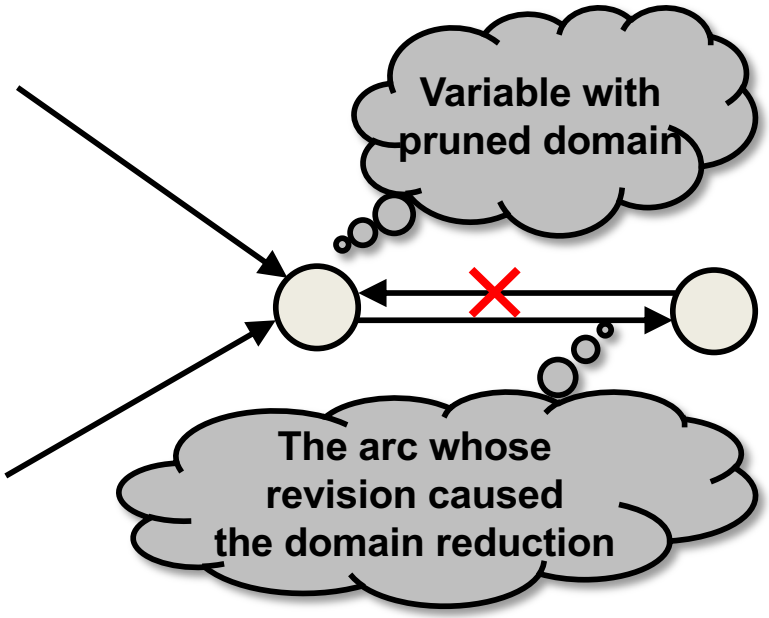
- If a single domain is pruned, then revisions of all the arcs are repeated even if the pruned domain does not influence most of these arcs.

Which 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).

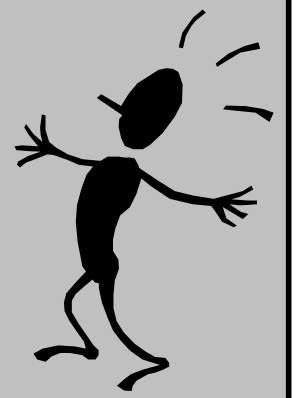


Principles of AC-3:

1. using a queue of arcs for (re-)revisions
2. only the arcs affected by domain reduction are added to the queue

Algorithm AC-3

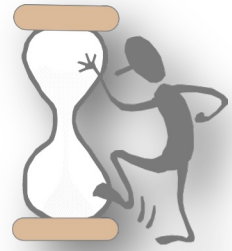
```
procedure AC-3(G)
    Q ← {(i,j) | (i,j)∈arcs(G), i≠j}        % queue of arcs for revision
    while Q nonempty do
        select and delete (k,m) from Q
        if REVISE((k,m)) then
            Q ← Q ∪ {(i,k) | (i,k)∈arcs(G), i≠k, i≠m}
        end if
    end while
end AC-3
```



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

- **AC-4 (Mohr, Henderson 1986)**
  - find supports for each value of every variable
  - removes values that have no support (lost support)
  - optimal worst-case time complexity
- **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-3.1: optimal AC-3



Some observations:

- AC-3 is not (theoretically) optimal
- AC-4 is (theoretically) optimal but (practically) slow
- AC-6 is (practically) faster than AC-4, but quite complicated

What is inefficient in AC-3?

- Looking for supports in REVISE starts from scratch!

if „there is no such Y in D_j such that (X,Y) is consistent“ **then**

AC-3.1

- same run as AC-3
- but for each value, it remembers the last support in the constraint and the next time, it starts looking for a support at this value

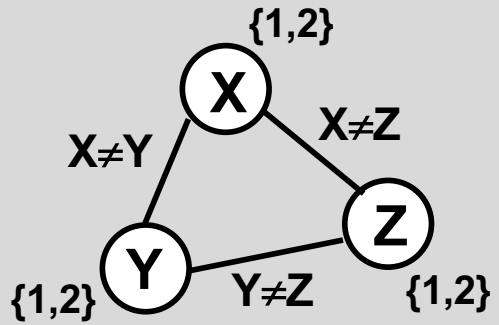
```
procedure EXIST((i,x),j)
    y ← last((i,x),j)
    if y∈Dj then return true
    while y←next(y,Dj) & y≠nil do
        if (x,y)∈C(i,j) then
            last((i,x),j) ← y
            return true
    end while
    return false
end EXIST
```

By applying AC we remove many inconsistent values.

- **Did we solve the problem?**
- **Do we know that a solution exists?**

NO and NO!

Example:



CSP is arc consistent
but there is no solution!

What is advantage of using AC?

- Sometimes **AC directly provides a solution**.
  - any domain is empty → no solution exists
  - all domains are singleton → this is a solution
- In general, AC **reduces the search space.**

How to strengthen the consistency level?

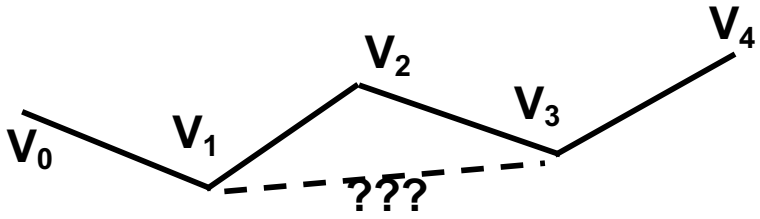
More constraints are assumed **together**!

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.

Beware:

- only the **constraints between the neighboring variables** must be satisfied



PC and paths of length two

It is not very practical to make all paths consistent.

Fortunately, it is enough to make path of length 2 consistent!

Theorem: CSP is PC if and only if all paths of length 2 are PC.

Proof:

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

2) All 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 trivially true

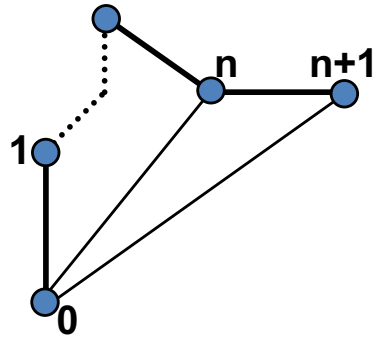
b) N+1 (assuming that the theorem holds for N)

i) take any N+2 nodes $V_0, V_1, \ldots, V_{n+1}$

ii) take any two consistent values $x_0 \in D_0$ and $x_{n+1} \in D_{n+1}$

iii) using a) find the value $x_n \in D_n$ s.t. $P_{0,n}$ and $P_{n,n+1}$ holds

iv) using induction find the other values $V_0, V_1, \ldots, V_n$



Does PC cover AC (if CSP PC, then is it also AC)?

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

Is PC stronger than AC (is there any CSP whish 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 is not consistent over X,Y,Z)

AC removes inconsistent values from the domains.

What is done by PC algorithms?

- **PC removes pairs of inconsistent values**
- PC makes all relations explicit ($A<B, B<C \Rightarrow A+1<C$)
- unary constraint = domain of the variable

PC algorithms will remove pairs of values

↳ we need to represent the constraints explicitly

Binary constraints = {0,1}-matrix

0 – pair of values is inconsistent

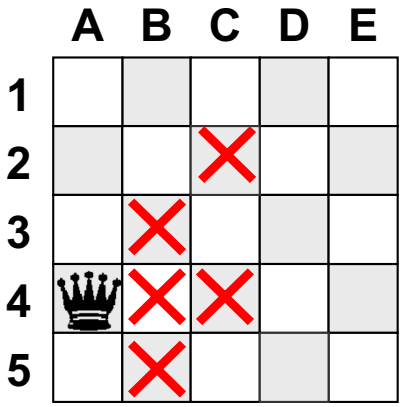
1 – pair of values is consistent

Example (5-queens problem)

constraint between queens **i** and **j**: r(i) ≠ r(j) & |i-j| ≠ |r(i)-r(j)|

Matrix representation for constraint A(1) - B(2)

0	0	1	1	1
0	0	0	1	1
1	0	0	0	1
1	1	0	0	0
1	1	1	0	0



Matrix representation for constraint A(1) - C(3)

0	1	0	1	1
1	0	1	0	1
0	1	0	1	0
1	0	1	0	1
1	1	0	1	0

Operations over constraints

Constraint 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

Constraint join $R_{ik} * R_{kj} \rightarrow R_{ij}$

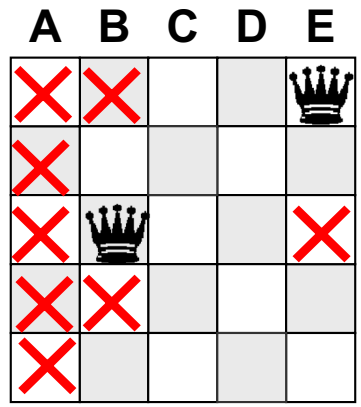
Binary matrix multiplication

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

Induced constraint is intersected with the original constraint

R_{ij} & (R_{ik} * R_{kj}) → 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



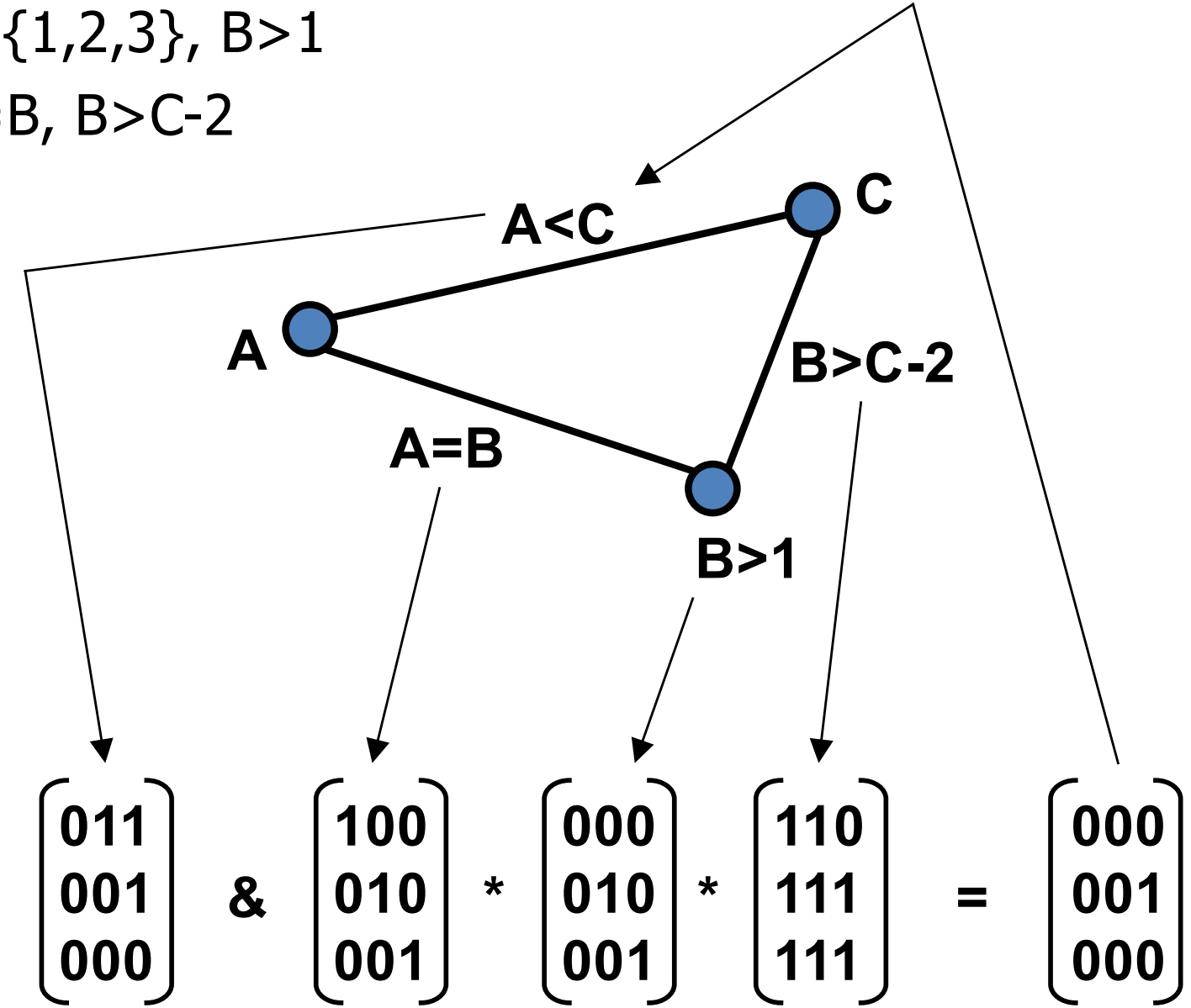
Notes:

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

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

A,B,C in {1,2,3}, B>1

A<C, A=B, B>C-2



Algorithm PC-1

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

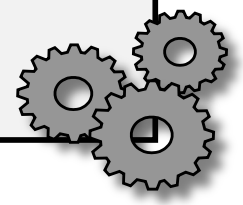
$R_{ij} \leftarrow R_{ij}$ & $(R_{ik} * R_{kk} * R_{kj})$

How to make a CSP path consistent?

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

```
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
$Y^k_{ii} \leftarrow Y^{k-1}_{ii}$ & $(Y^{k-1}_{ik} * Y^{k-1}_{kk} * Y^{k-1}_{ki})$
then we get AC-1



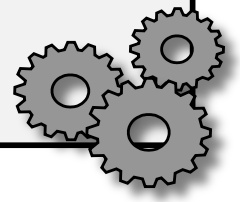
Is there any inefficiency in PC-1?

- 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
    but it is enough to revise just the influenced paths

Algorithm of path revision

```
procedure REVISE_PATH((i,k,j))
    Z ← Yij & (Yik * Ykk * Ykj)
    if Z=Yij then return false
    Yij ← Z
    return true
end REVISE_PATH
```

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



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

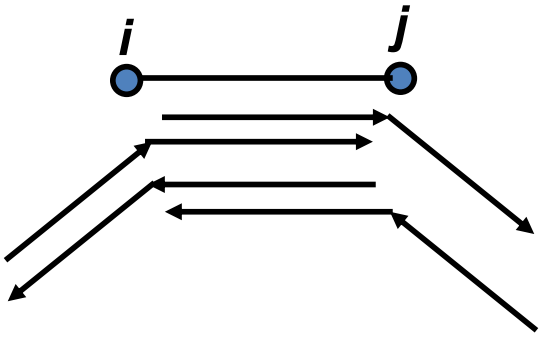
Situation a: i<j

all the paths containing (i,j) or (j,i) must be re-revised

but the paths (i,j,j), (i,i,j) are not revised again (no change)

$S_a = \{(i,j,m) \mid i \leq m \leq n \;\&\; m \neq j\}$
$\cup \{(m,i,j) \mid 1 \leq m \leq j \;\&\; m \neq i\}$
$\cup \{(j,i,m) \mid j < m \leq n\}$
$\cup \{(m,j,i) \mid 1 \leq m < i\}$

$|S_a| = 2n-2$



Situation b: i=j

all the paths containing i in the middle of the path are re-revised

but the paths (i,i,i) and (k,i,k) are not revised again

$S_b = \{(p,i,m) \mid 1 \leq m \leq n \;\&\; 1 \leq p \leq m\} - \{(i,i,i),(k,i,k)\}$

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

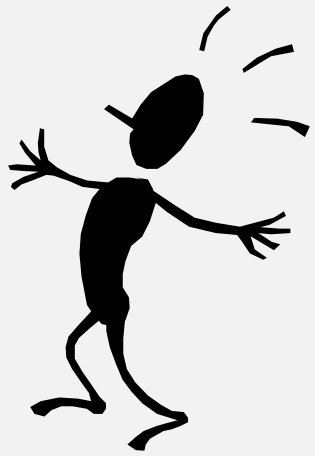
Algorithm PC-2

Paths in one direction only (attention, this is not DPC!)

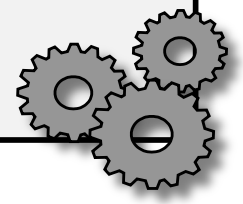
After every revision, the **affected paths are re-revised**

Algorithm PC-2

```
procedure PC-2(G)
    n ← |nodes(G)|
    Q ← {(i,k,j) | 1 ≤ i ≤ j ≤ n & i≠k & j≠k}
    while Q non empty do
        select and delete (i,k,j) from Q
        if REVISE_PATH((i,k,j)) then
            Q ← Q ∪ 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
```



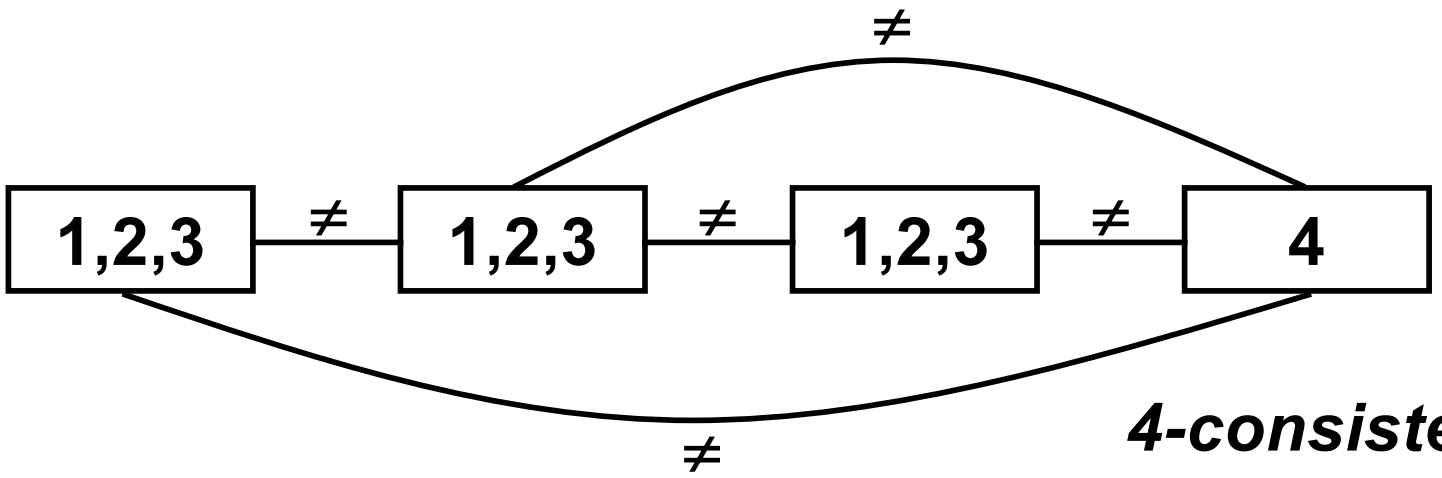
- **PC-3 (Mohr, Henderson 1986)**
  - based on computing supports for a value (like AC-4)
    - If pair (*a*,*b*) at arc (*i*,*j*) is not supported by another variable, then *a* is removed from D_i and *b* is removed from D_j.
  - **this algorithm is not sound!**
- **PC-4 (Han, Lee 1988)**
  - correction 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

Is there a common formalism for AC and PC?

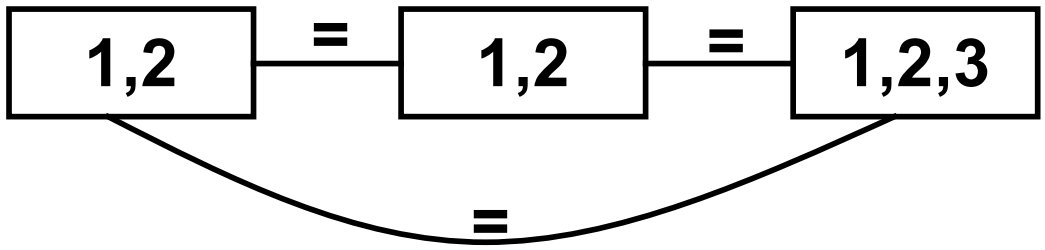
- AC: a value is extended to another variable
- PC: a pair of values is extended to another variable
- … we can continue

Definition:

CSP is k-consistent if and only if any consistent assignment of (k-1) different variables can be extended to a consistent assignment of one additional variable.



4-consistent graph



3-consistent graph

but not 2-consistent graph!

Definition:

A **CSP is strongly k-consistent** iff it is j-consistent for every $j \leq k$.

Features:

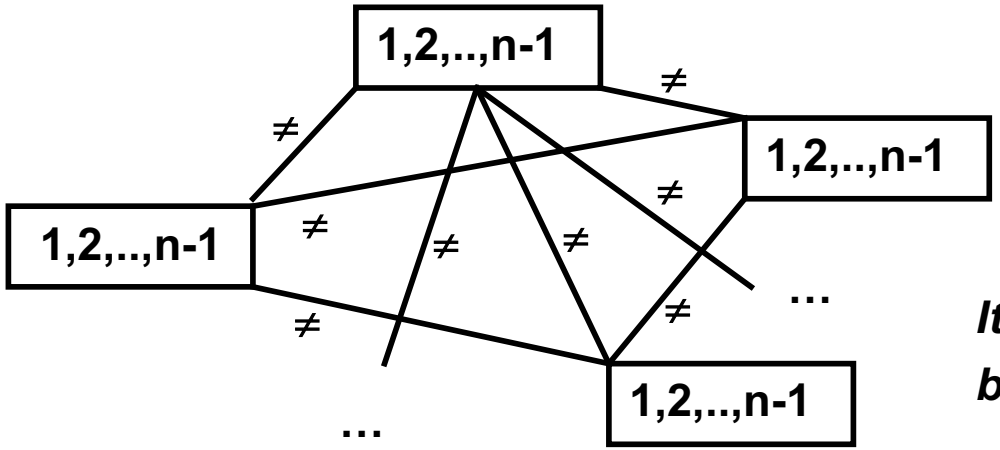
- **strong k-consistency $\Rightarrow$ k-consistency**
- **strong k-consistency $\Rightarrow$ j-consistency $\forall j \leq k$**
- **k-consistency $\Rightarrow$ strong k-consistency** *does not hold in general*

Naming scheme

- NC (node consistency) = 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?

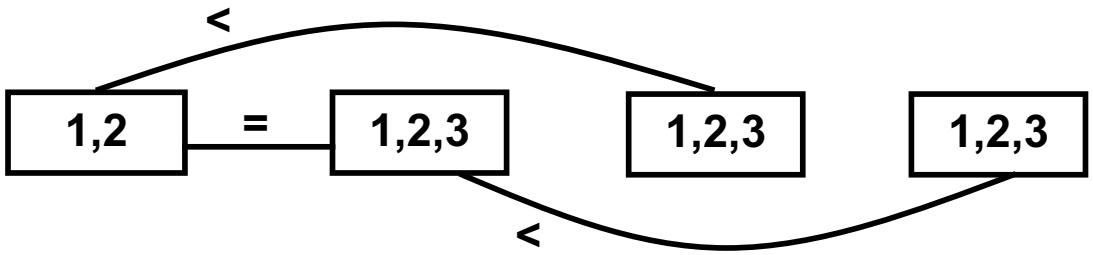
- Assume that the number of vertices is *n*. What level of consistency do we need to find out the solution?
- **Strong *n*-consistency for graphs with *n* vertices!**
  - n-consistency is not enough - see the previous example
  - strong k-consistency where k<n is not enough as well



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

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

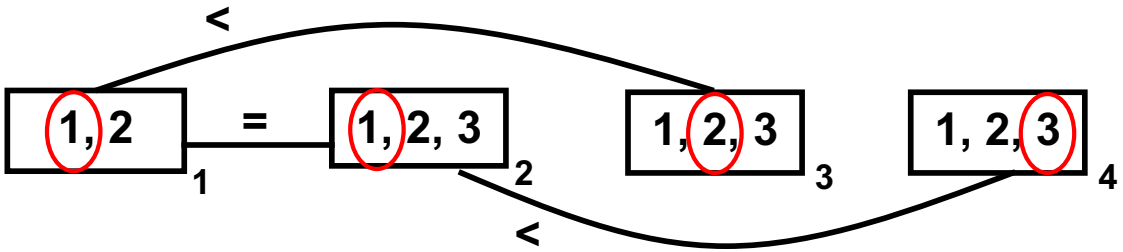
And what about this graph?



AC is enough!
Because this a tree..

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.

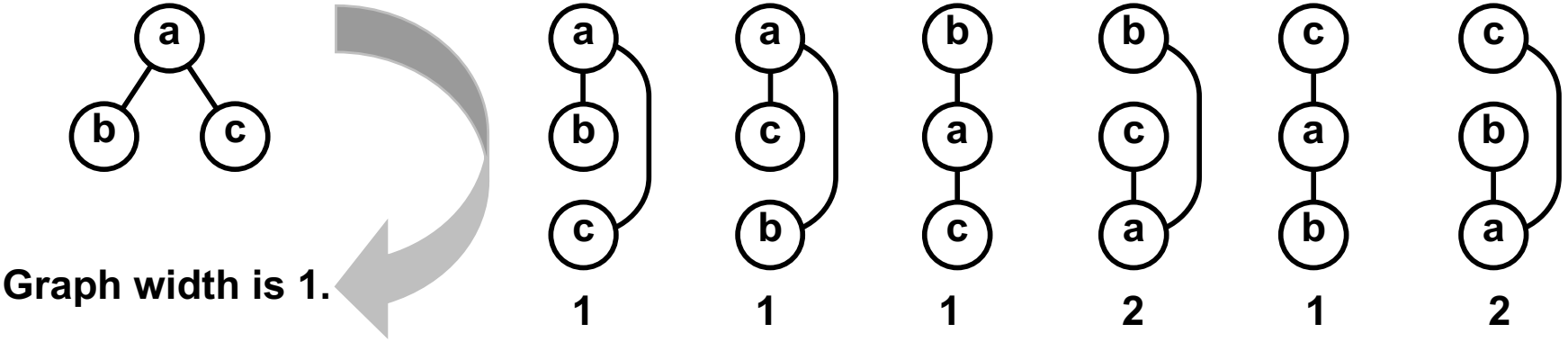


How to find out a sufficient consistency level for a given graph?

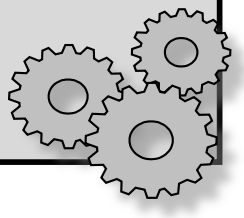
Some observations:

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

- **Ordered graph** is a graph with some total ordering of nodes.
- **Node width** in the ordered graph is the number of backward edges from this node.
- **Width of the ordered graph** is the maximal width of its nodes.
- **Graph width** is the minimal width among all possible node orders.




```
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
```

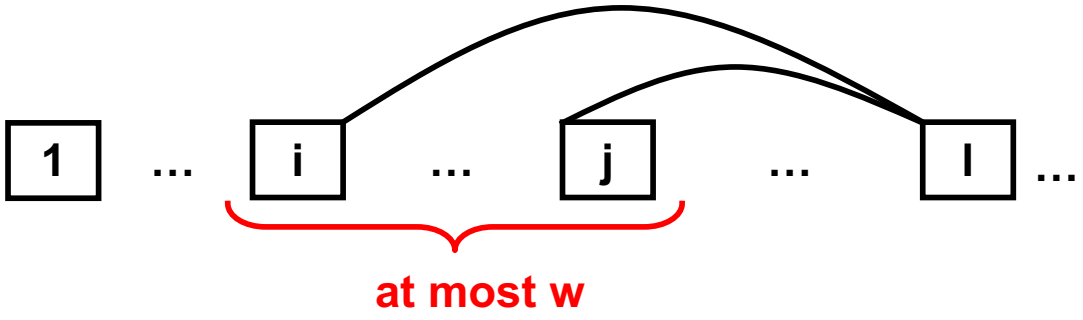


Theorem:

If the constraint graph is strongly k-consistent for some k>w, where w is the graph width, then there exists an order of variables giving a backtrack-free search solution.

Proof:

- there exists an ordering of nodes with the graph width w,
- in particular, the number of backward edges for each node is at most w,
- we will assign the variables in the order given by the above ordered graph
- now, when assigning a value to the variable:
  - we need to find a value consistent with the existing assignment, i.e., consistent with previous variables connected via arcs with the variable,
  - let m by the number of such variables, then $m \leq w$
  - the graph is (m+1)-consistent, so the value must exist



So far, we assumed mainly **binary constraints**.

We can use binary constraints, because **every CSP can be converted to a binary CSP**!

Is this really done in practice?

- in many applications, non-binary constraints are naturally used, for example, $a+b+c \leq 5$
- for such constraints we can do some local inference / propagation

  for example, if we know that $a,b \geq 2$, we can deduce that $c \leq 1$
- within a single constraint, we can restrict the domains of variables to the values satisfying the constraint

  ↳ **generalized arc consistency**

- **The value** x of variable V is **generalized arc consistent** with respect to constraint P if and only if there exist values for the other variables in P such that together with x they satisfy the constraint P

  Example: A+B≤C, A in {1,2,3}, B in {1,2,3}, C in {1,2,3}
  Value 1 for C is not GAC (it has no support), 2 and 3 are GAC.

- **The variable** V is **generalized arc consistent** with respect to constraint P, if and only if all values from the current domain of V are GAC with respect to P.

  Example: A+B≤C, A in {1,2,3}, B in {1,2,3}, C in {2,3}
  C is GAC, A and B are not GAC

- **The constraint** C is **generalized arc consistent**, if and only if all variables in C are GAC.

  Example: for A in {1,2}, B in {1,2}, C in {2,3} A+B≤C is GAC

- **The constraint satisfaction problem** P is **generalized arc consistent**, if and only if all the constraints in P are GAC.

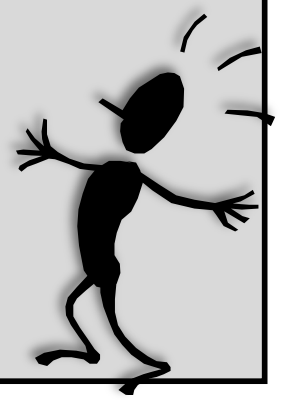
We will modify AC-3 for non-binary constraints.

- We can see a constraint as a set of propagation methods – each method makes one variable GAC:

  A + B = C: A + B $\rightarrow$ C, C – A $\rightarrow$ B, C – B $\rightarrow$ A

- By executing all the methods, we make the constraint GAC.
- We repeat revisions until any domain changes.

```
procedure GAC-3(G)
    Q ← {Xs →Y | Xs →Y is a method for some constraint in G}
    while Q non empty do
        select and delete (As→B) from Q
        if REVISE(As→B) then
            if D_B=∅ then stop with fail
            Q ← Q ∪ {Xs →Y | Xs →Y is a method s.t. B∈Xs}
        end if
    end while
end GAC-3
```



Can we achieve GAC **faster than a general GAC algorithm**?

- for example, revision of A < B can be done much faster via bounds consistency.

Can we write a **filtering algorithm for a constraint** whose **arity varies**?

- for example, all_different constraint

We can exploit **semantics of the constraint** for efficient filtering algorithms that can work with any number of variables.

☞ global constraints ☜

Recall Sudoku

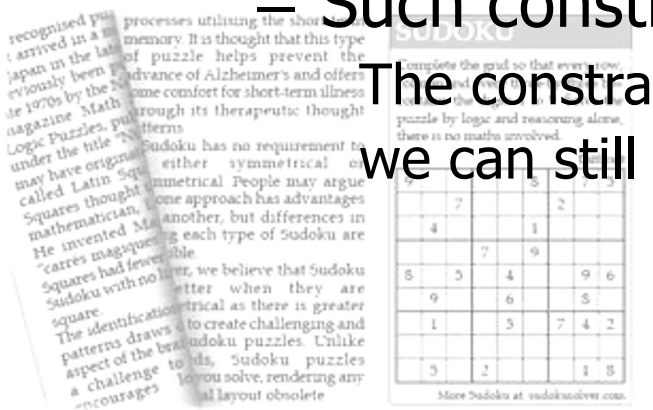
9	6	3	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	8	1	7	3

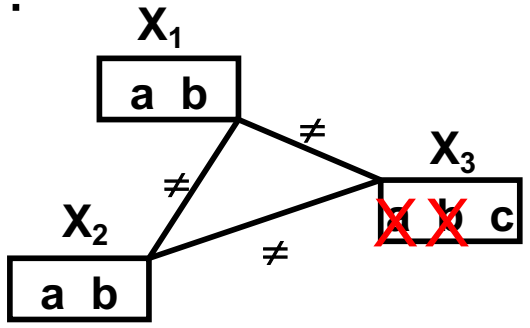
Logic-based puzzle, whose goal is to enter digits 1-9 in cells of 9×9 table in such a way, that no digit appears twice or more in every row, column, and 3×3 sub-grid.

How to model such a problem?

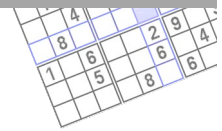
- variables **describe the cells**
- **inequality constraint** connect each pair of variables in each row, column, and sub-grid
- Such constraints do not propagate well!

The constraint network is AC, but we can still remove some values.





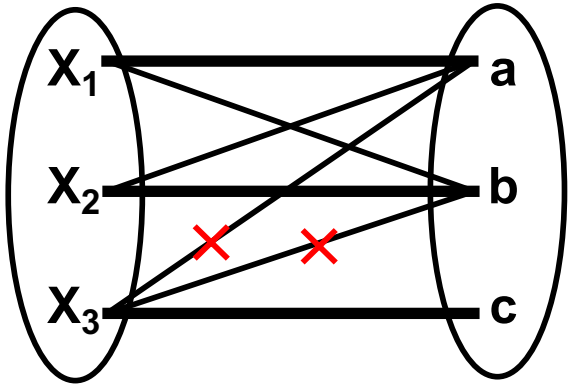
all-different



This constraint models a complete set of binary inequalities.

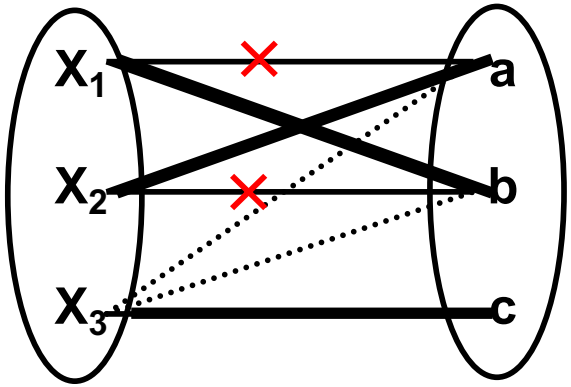
all_different($\{X_1,\ldots, X_k\}$) = $\{(d_1,\ldots, d_k) \mid \forall i\ d_i \in D_i\ \&\ \forall i \neq j\ d_i \neq d_j\}$

Domain filtering is based on **matching in bipartite graphs**
(nodes = variables+values, edges = description of domains)



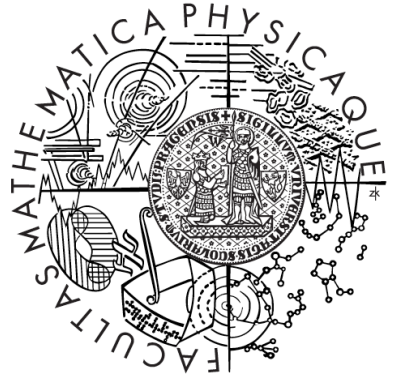
Initialization:

1) find a maximum matching
2) remove all edges that do not belong to any maximum matching



Incremental propagation ($X_1 \neq a$):

1) remove "deleted" edges
2) find a new maximum matching
3) remove all edges that do not belong to any maximum matching



Charles University, Faculty of Mathematics and Physics
bartak@ktiml.mff.cuni.cz