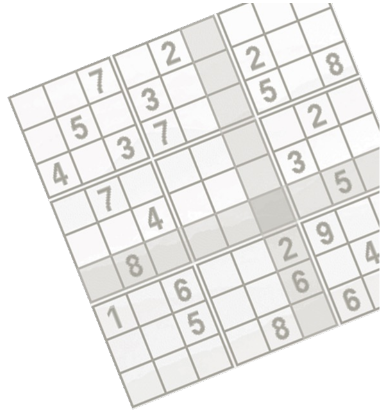


ESSAI & ACAI 2024
ATHENS - GREECE

Introduction to Constraint Satisfaction

Roman Barták

Charles University, Prague (CZ)

Higher-level consistency techniques

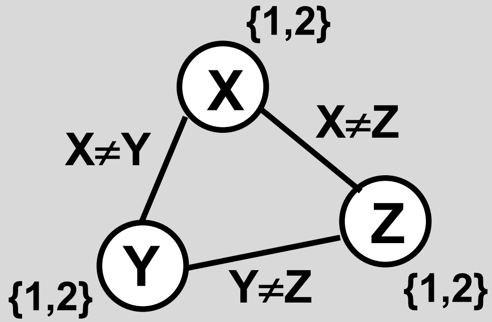
Arc consistency:

- **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:



CSP is arc consistent
but there is no solution!

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

How to strengthen the consistency level?

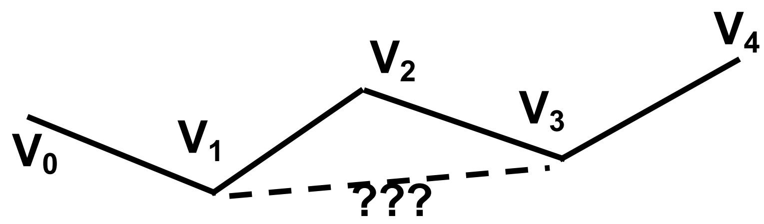
More constraints are assumed **together**!

Definition:

- **The path** (V_0,V_1,…, 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,…,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

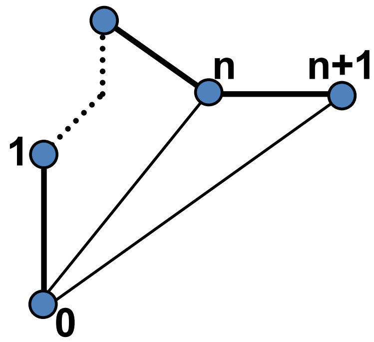
b) N+1 (assuming that the theorem holds for N)

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

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

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

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



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

Representation of constraints

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

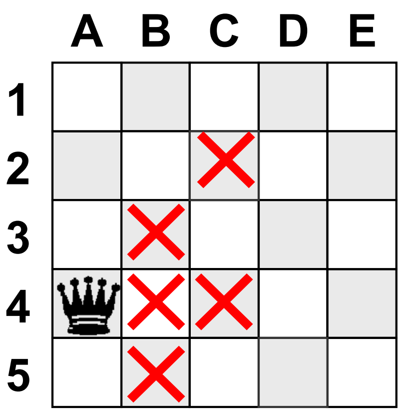
1 – pair of values is consistent

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

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

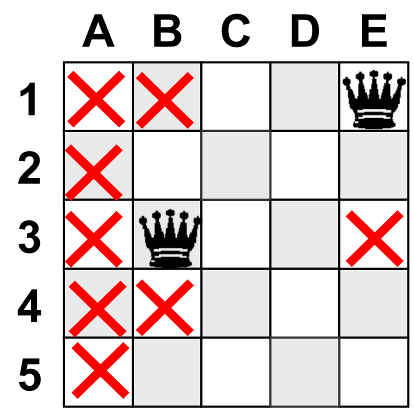
Binary matrix multiplication

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

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:

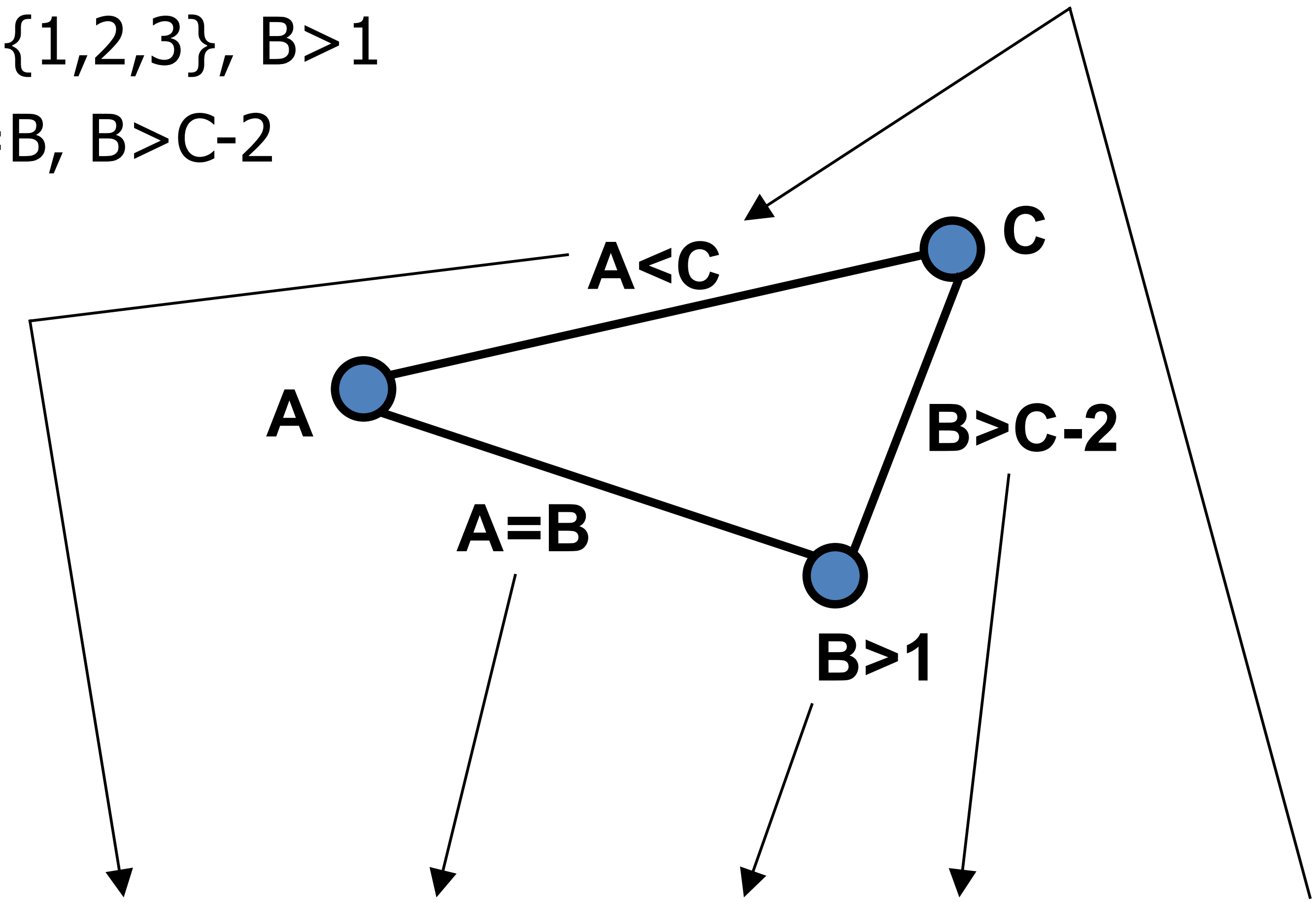
$R_{ij} = R^T_{ji}$, R_{ii} is a diagonal matrix representing the domain of variable

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

Composing constraints

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

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



$$\begin{bmatrix}011\\001\\000\end{bmatrix} \;\&\; \begin{bmatrix}100\\010\\001\end{bmatrix} * \begin{bmatrix}000\\010\\001\end{bmatrix} * \begin{bmatrix}110\\111\\111\end{bmatrix} = \begin{bmatrix}000\\001\\000\end{bmatrix}$$

Mackworth (1977)

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

How to improve PC-1?

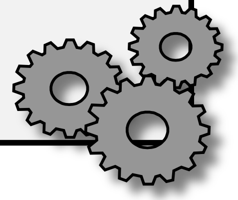
Is there any inefficiency in PC-1?

- 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
    but it is enough to revise just the influenced paths

Algorithm of path revision

procedure REVISE_PATH((i,k,j))
  $Z \leftarrow Y_{ij}$ & $(Y_{ik} * Y_{kk} * Y_{kj})$
  if $Z=Y_{ij}$ **then return** false
  $Y_{ij} \leftarrow Z$
  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):

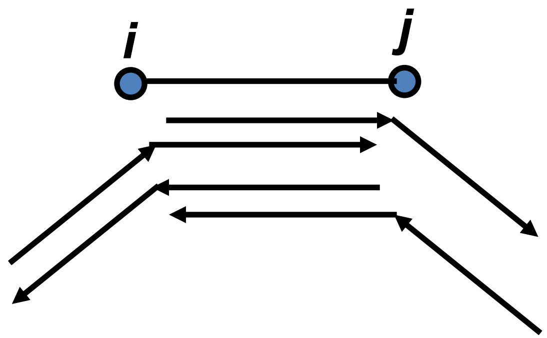
Situation a: i<j

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)

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

$| S_a | = 2n-2$



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

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

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

Mackworth (1977)

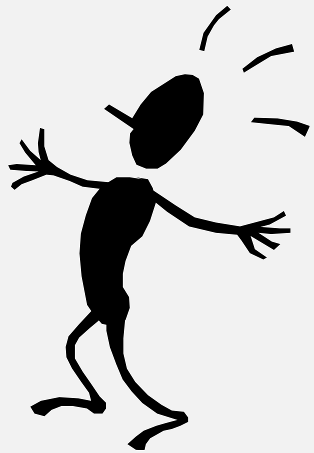
Algorithm PC-2

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

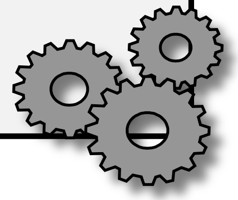
After every revision, the **affected paths are re-revised**

Algorithm PC-2

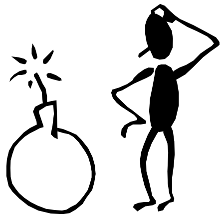
```
procedure PC-2(G)
    n ← |nodes(G)|
    Q ← {(i,k,j) | 1 ≤ i ≤ j ≤ n & i≠k & j≠k}
    while Q non empty do
        select and delete (i,k,j) from Q
        if REVISE_PATH((i,k,j)) then
            Q ← Q ∪ RELATED_PATHS((i,k,j))
    end while
end PC-2
```



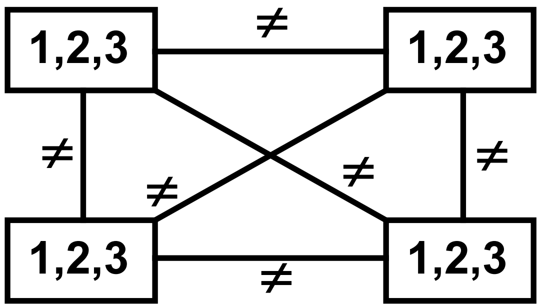
```
procedure RELATED_PATHS((i,k,j))
    if i<j then return S_a else return S_b
end RELATED_PATHS
```



- **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



- **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 density, 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 no solution

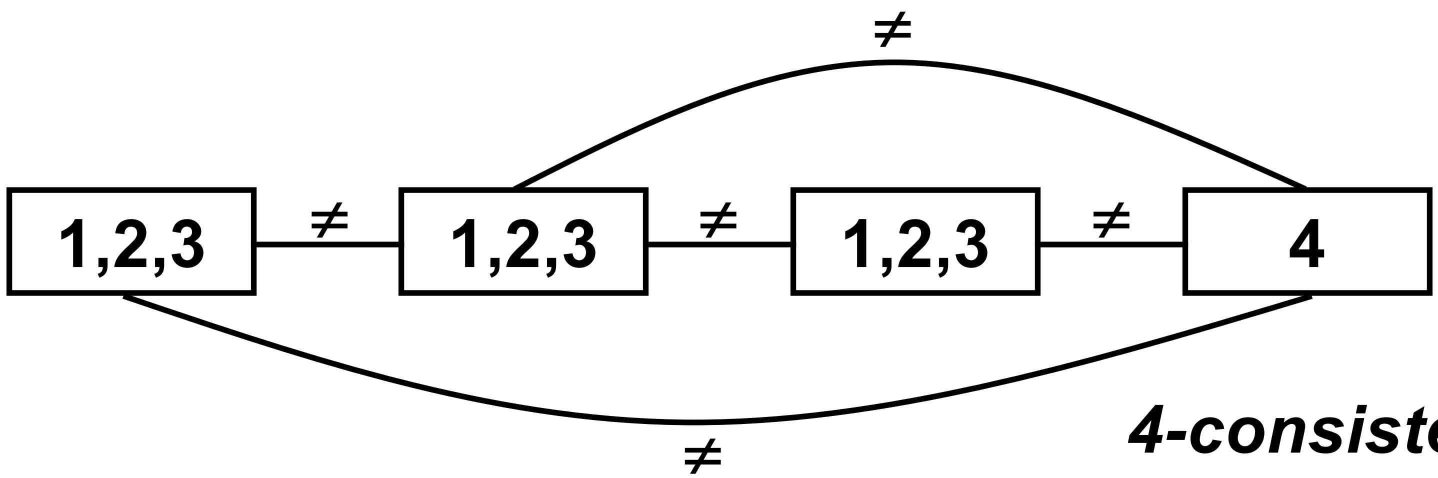


Is there a common formalism for AC and PC?

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

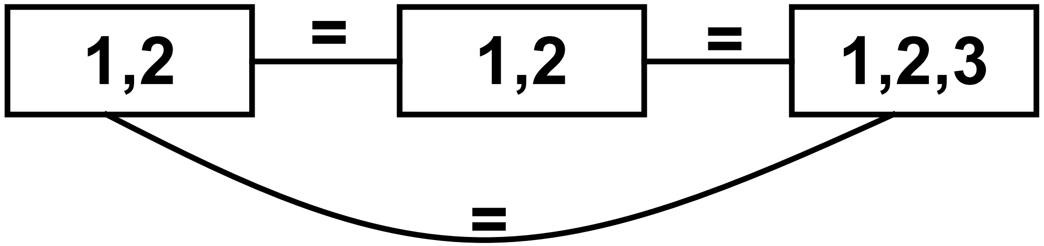
Definition:

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

Strong k-consistency



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:

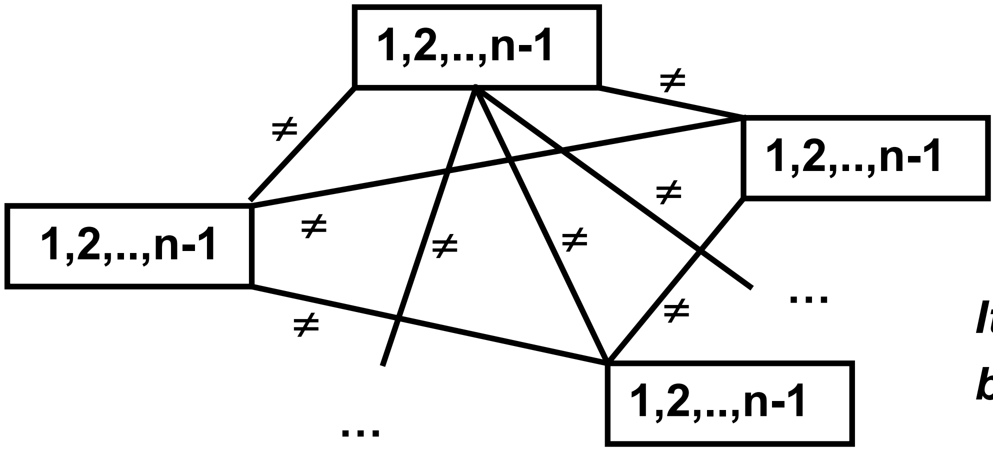
- **strong k-consistency ⇒ k-consistency**
- **strong k-consistency ⇒ j-consistency ∀j≤k**
- **k-consistency ⇒ strong k-consistency** *does not hold in general*

Naming scheme

- 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?

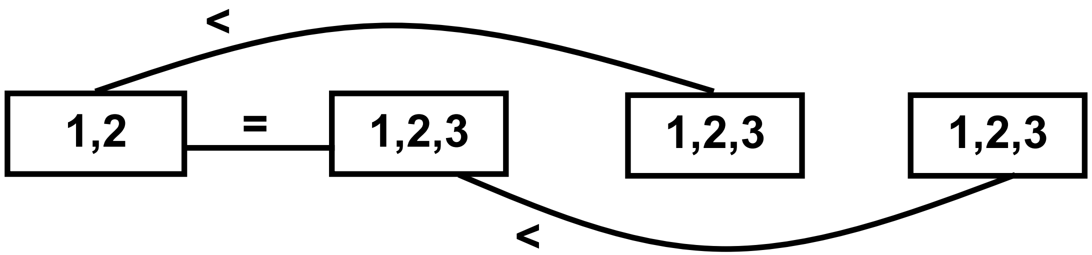
- Assume that the number of vertices is *n*. What level of consistency do we need to find out the solution?
- **Strong *n*-consistency for graphs with *n* vertices!**
  - n-consistency is not enough - see the previous example
  - strong k-consistency where k<n is not enough as well



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

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

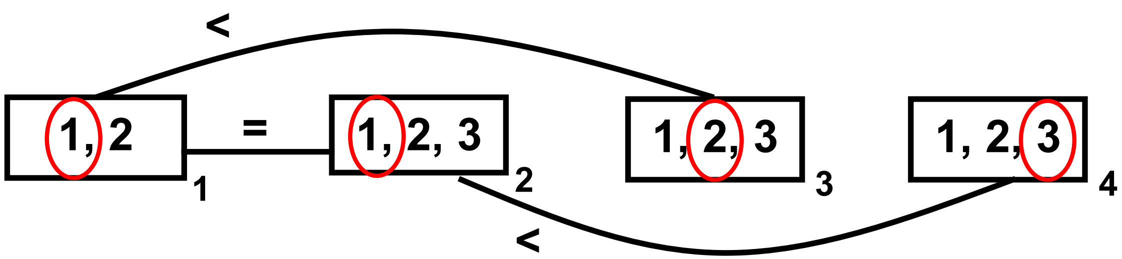
And what about this graph?



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.

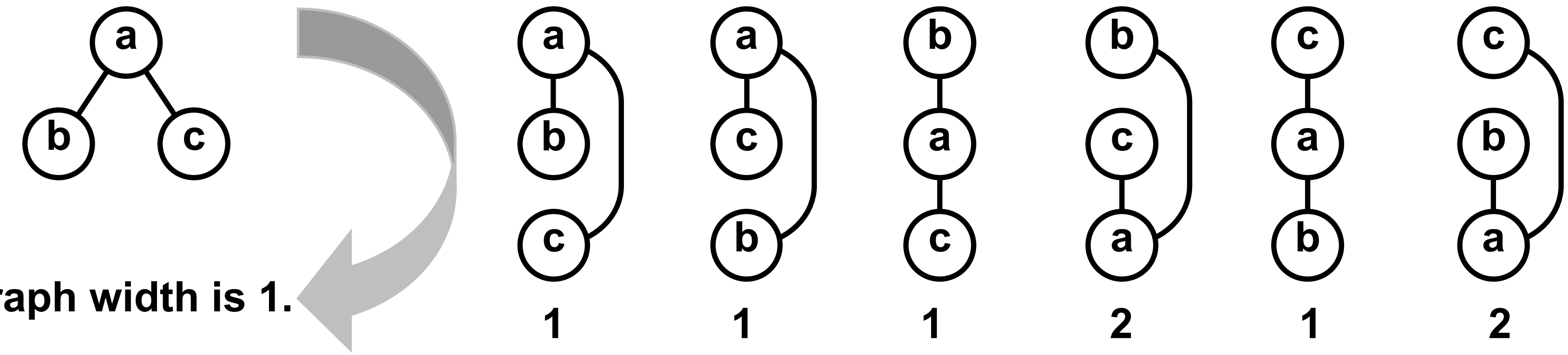


How to find out a sufficient consistency level for a given graph?

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.



Graph width is 1.

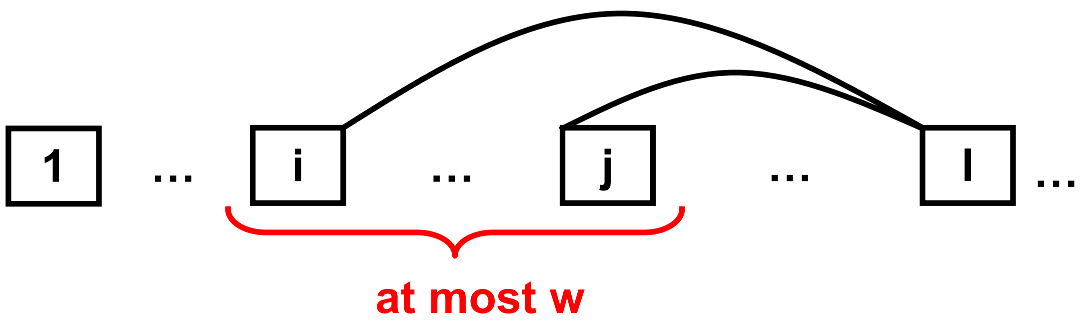
```
procedure MinWidthOrdering((V,E))
    Q ← {}
    while V not empty do
        N ← select and delete node with the smallest #edges from (V,E)
        enqueue N to Q
    return Q
end MinWidthOrdering
```

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 ≤ w
  - the graph is (m+1)-consistent, so the value must exist



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.

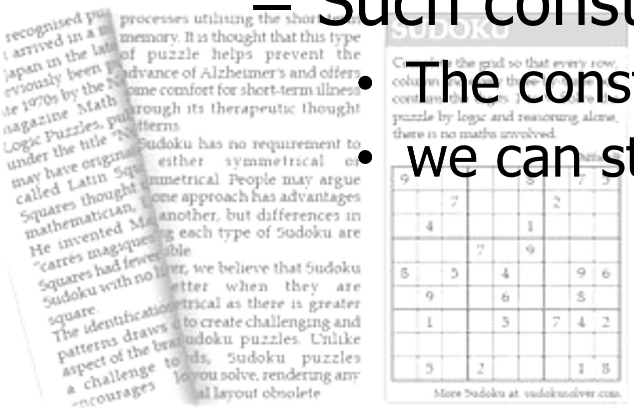
☞ **global constraints** ☜

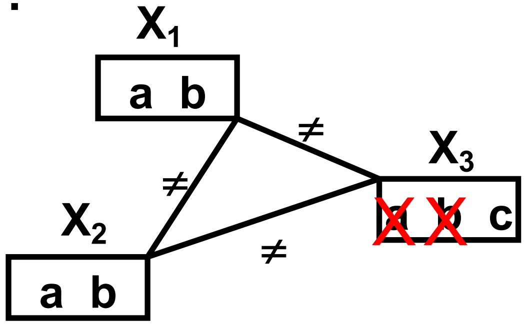
Logic-based puzzle, whose goal is to enter digits 1-9 in cells of 9×9 table in such a way, that no digit appears twice or more in every row, column, and 3×3 sub-grid.

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

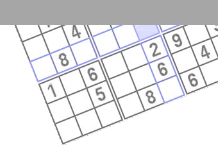
How to model such a problem?

- 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.





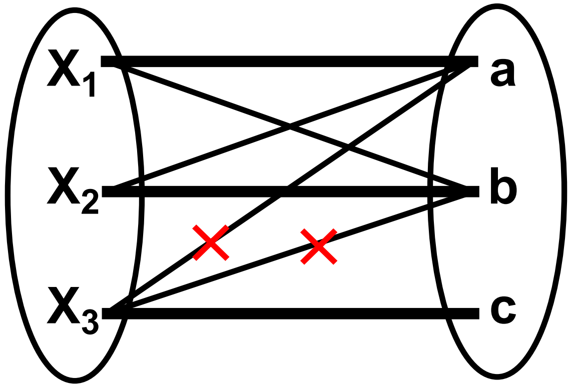
all-different



This constraint models a complete set of binary inequalities.

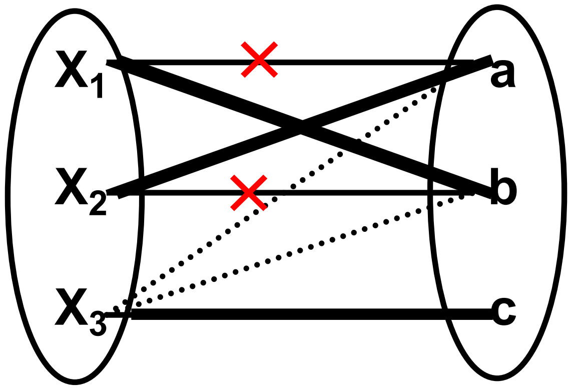
`all_different`$(\{X_1,\ldots, X_k\}) = \{(d_1,\ldots, d_k) \mid \forall i\ \ d_i \in D_i\ \ \&\ \forall i \neq j\ \ d_i \neq d_j\}$

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



Initialization:

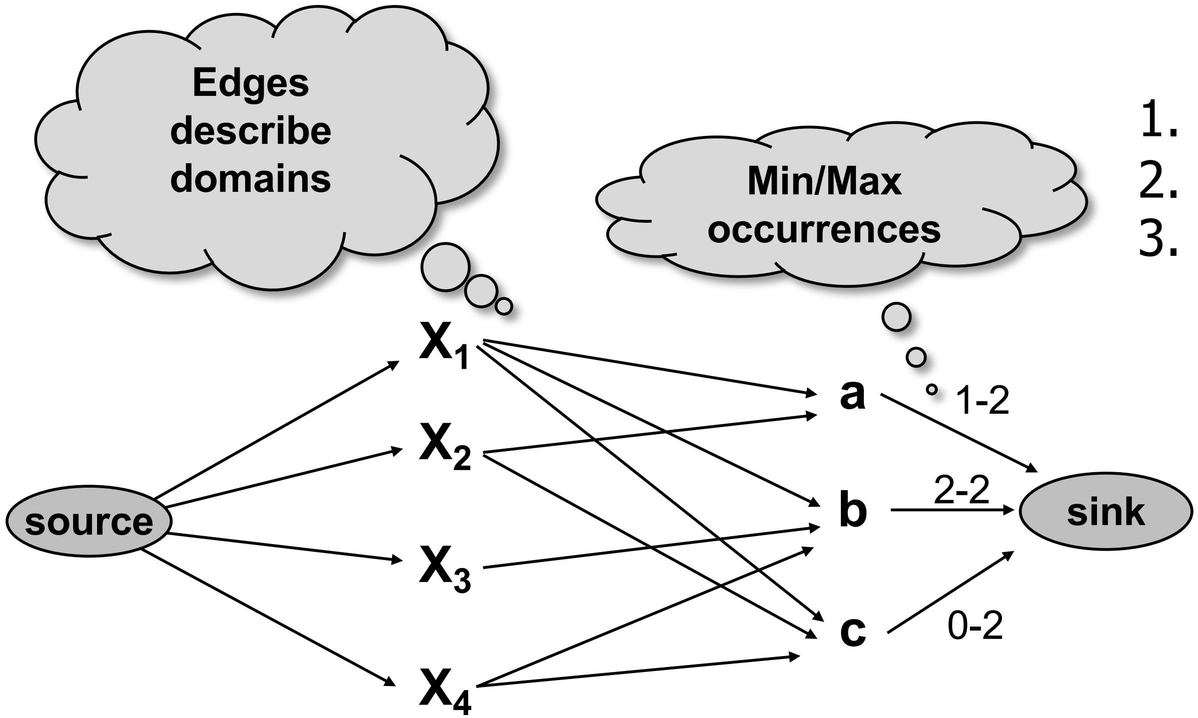
1) find a maximum matching
2) remove all edges that do not belong to any maximum matching



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

- A generalization of all-different
  - the number of occurrences of a value in a set of variables is restricted by minimal and maximal numbers of occurrences
- Efficient filtering is based on **network flows.**



1. make a value graph
2. add sink and source
3. set upper and lower bounds and edge capacities (0-1 and value occurrences)

A maximal flow corresponds to a feasible assignment of variables! We will find edges with zero flow in each maximal flow and then we will remove the corresponding edges.

Symmetry breaking

Existence of **symmetrical solutions** decreases efficiency of constraint satisfaction (symmetrical search spaces are explored).

A classical example with many symmetries – **sports tournament scheduling.**

- there are n teams
- each team plays will all other teams, i.e., (n-1) rounds
- each team plays as a home team or a guest team

How to model such a problem?

- Round I is modelled by a sequence of **match codes** K_i.
  - $K_{i,j}$ is a code of j-th match at at round i
- We can swap matches at each round – **match symmetry**.
  - match symmetry is removed by constraint $K_{i,j} < K_{i,j+1}$
- We can swap complete rounds – **round symmetry**.
  - round symmetry is removed by constraint $K_i <_{lex} K_{i+1}$.

this constraint models **lexicographic ordering of two vectors**

lex$(\{X_1,\ldots, X_n\}, \{Y_1,\ldots, Y_n\}) \equiv (X_1 \leq Y_1) \wedge (X_1 = Y_1 \Rightarrow X_2 \leq Y_2) \wedge \ldots$
$\ldots \wedge (X_1 = Y_1 \wedge \ldots \wedge X_{n-1} = Y_{n-1} \Rightarrow X_n < Y_n)$

Global filtering procedure uses two pointers:

- α: the variables before α are all instantiated and pairwise equal
- β: vectors starting at β are lexicographically ordered but "oppositely"
  floor$(\{X_\beta,\ldots, X_n\}) >_{lex}$ ceiling$(\{Y_\beta,\ldots, Y_n\})$

$X = \langle \{2\}, \{1,3,4\},\{1,2,3,4,5\},\{1,2\},\{3,4,5\} \rangle$ first set the pointers
$Y = \langle \{0,1,2\}, \{1\}, \{0,1,2,3,4\},\{0,1\},\{0,1,2\} \rangle$
$\alpha\uparrow$ $\uparrow\beta$

$X = \langle \{2\}, \{1,3,4\},\{1,2,3,4,5\},\{1,2\},\{3,4,5\} \rangle$ change Y_1, so at least $X_1 = Y_1$
$Y = \langle \{2\}, \{1\}, \{0,1,2,3,4\},\{0,1\},\{0,1,2\} \rangle$ and shift pointer α
$\alpha\uparrow$ $\uparrow\beta$

$X = \langle \{2\}, \{1\}, \{1,2,3,4,5\},\{1,2\},\{3,4,5\} \rangle$ change X_2 so at least $X_2 = Y_2$
$Y = \langle \{2\}, \{1\}, \{0,1,2,3,4\},\{0,1\},\{0,1,2\} \rangle$ and again shift pointer α
$\alpha\uparrow$ $\uparrow\beta$

$X = \langle \{2\}, \{1\}, \{1,2,3\}, \{1,2\},\{3,4,5\} \rangle$ because $\alpha = \beta -1$
$Y = \langle \{2\}, \{1\}, \{2,3,4\}, \{0,1\},\{0,1,2\} \rangle$ force constraint $X_\alpha < Y_\alpha$
$\alpha\uparrow$ $\uparrow\beta$

Rostering

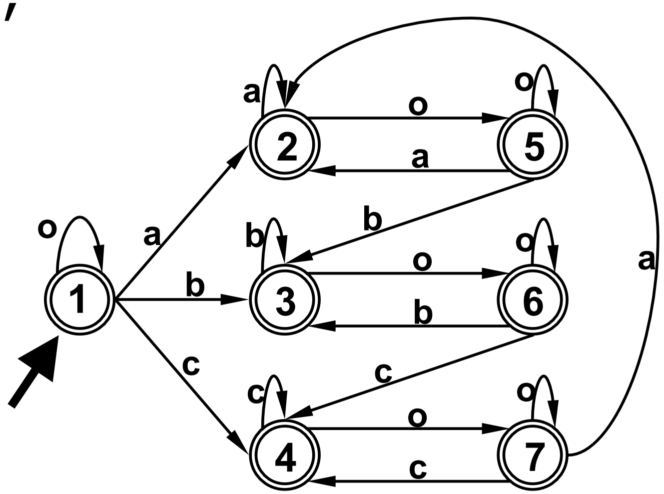
- **scheduling of shifts**, for example in hospitals
- There are typically specific shift sequencing constraints (given by trade unions, law etc.)

Example:

- shifts: a, b, c, o (o means a free shift)
- constraints:
  - the same shift can repeat each day
  - at least one o shift is between a, b, between b, c, and between c, a
  - a-o*-c, b-o*-a, c-o*-b are not allowed (o* is a sequence of o shifts)
- Any shift can be used the first day, only shifts b, o can be used the second day, shifts a, c, o for the third day, shifts a, b, o for the forth day, and shift a the fifth day.

How to model such a problem?

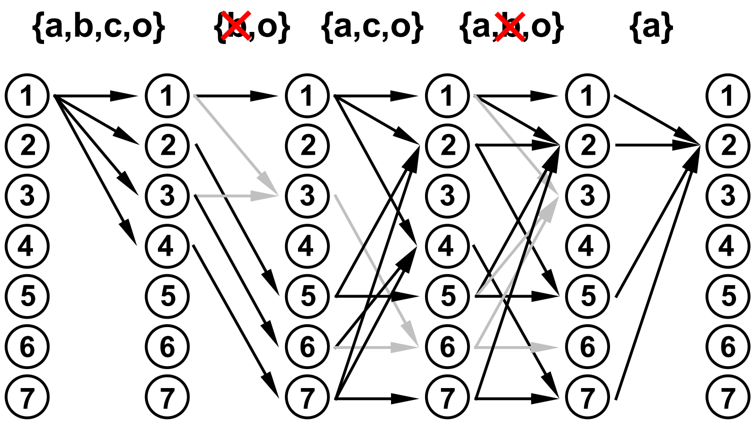
- variables describe shifts in days
- And what about constraints?
  - using a finite state automaton (FSA)



models a sequence of symbols **accepted by a FSA**

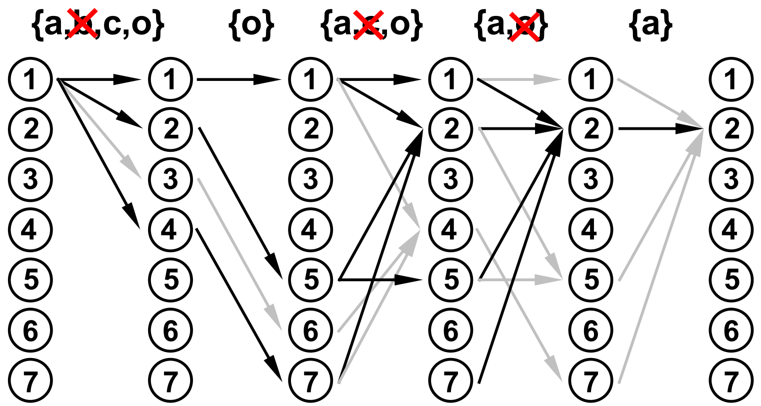
$\texttt{regular}(A, \{X_1,\ldots, X_k\}) = \{(d_1,\ldots, d_k) \mid \forall i \;\; d_i \in D_i \;\; \wedge \; d_1 \ldots d_k \in L(A)\}$

filtering is based on representing all possible computations of a FSA using a **layered directed graph** (layer=states, arc=transitions)



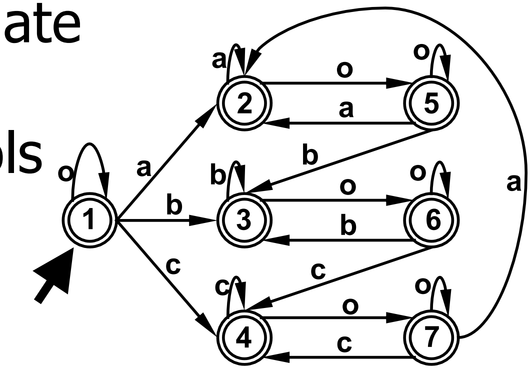
Initialisation

1. add arcs going from the initial state based on the symbols in the variables' domains
2. during the backward run, remove the arcs that are not on paths to the final states
3. remove the symbols without any arc



Incremental filtering ($X_4 \neq$ o):

1. remove arcs for the deleted symbol
2. propagate the update in both directions
3. remove the symbols without any arc



Let us go back to the **regular** constraint, which behaves like **sliding a special transition constraint over a sequence of variables**.
Such a principle can be generalized!

$\mathbf{slide_j}(C, \{X_1,\ldots, X_n\}) \equiv \forall i\ C(X_{ij+1},\ldots, X_{ij+k})$

- C is a k-ary constraint
- constant j determines the slide length

Some examples:

- $\mathbf{regular}(A, \{X_1,\ldots, X_n\}) \equiv \mathbf{slide_2}(C, \{Q_0,X_1,Q_1, \ldots, X_n, Q_n\})$
  $C(P,X,Q)$ represents a transition $\delta(P,X) = Q$, $Q_0 = \{q_0\}$, $Q_n = F$
- $\mathbf{lex}(\{X_1,\ldots, X_n\}, \{Y_1,\ldots, Y_n\}) \equiv \mathbf{slide_3}(C, \{B_0,X_1,Y_1,B_1, \ldots, X_n, Y_n, B_n\})$
  $C(B,X,Y,C) \equiv B=C=1$ or ($B=C=0$ and $X=Y$) or ($B=0$, $C=1$ and $X<Y$)
  $B_0 = 0$, $B_n = 1$ (strict lex), B_n in $\{0,1\}$ (non lex)
- $\mathbf{stretch}(\{X_1,\ldots, X_n\}, s, l , t) \equiv \mathbf{slide_2}(C, \{X_1,S_1, \ldots, X_n, S_n\})$
  $C(X_i, S_i, X_{i+1}, S_{i+1}) \equiv X_i = X_{i+1}, S_{i+1} = 1+S_i, S_{i+1} \leq l(X_i)$,
  or $X_i \neq X_{i+1}, S_i \geq s(X_i), S_{i+1} = 1, (X_i, X_{i+1}) \in t$
  $S_1 = 1$

…

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