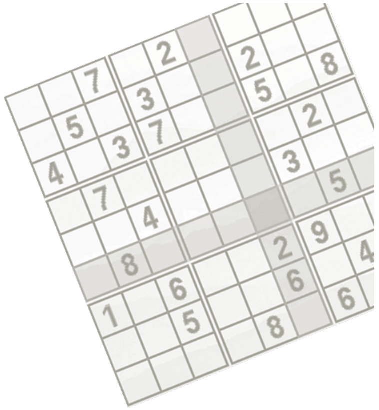


Constraint Programming

Roman Barták

Department of Theoretical Computer Science and Mathematical Logic

Consistency Techniques: Beyond AC and PC

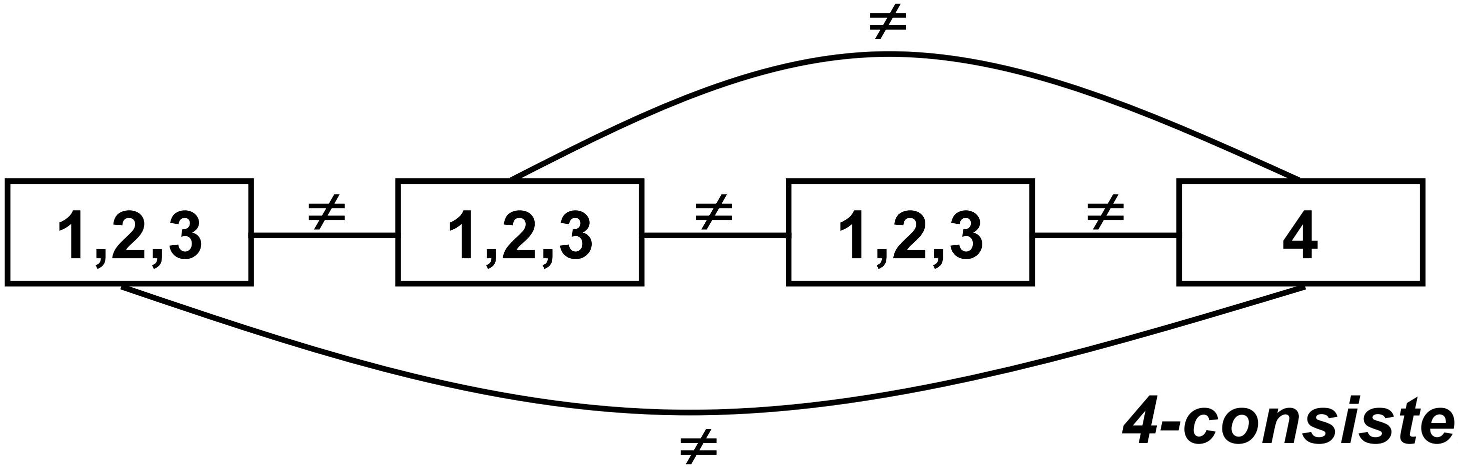
k-consistency

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

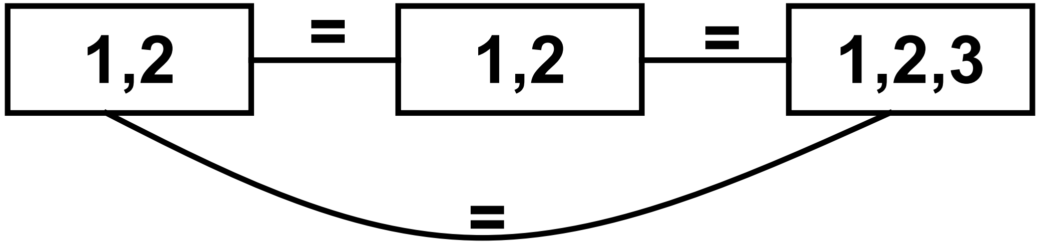
Definition:

CSP is k-consistent 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 ≤ k.

Features:

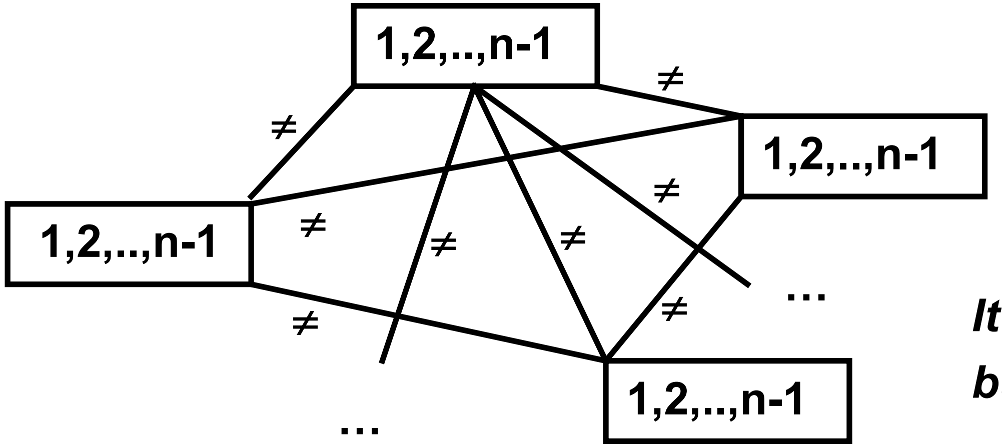
- **strong k-consistency ⇒ k-consistency**
- **strong k-consistency ⇒ j-consistency ∀j≤k**
- **k-consistency ⇒ strong k-consistency** *does not hold in general*

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?

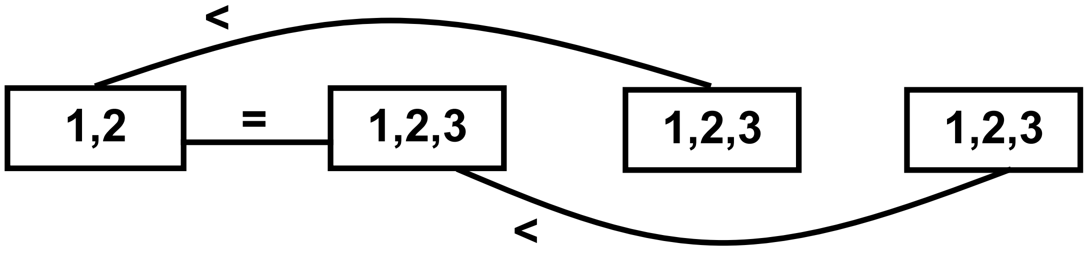
- Assume that the number of vertices is *n*. What level of consistency do we need to find out the solution?
- **Strong *n*-consistency for graphs with *n* vertices!**
  - n-consistency is not enough - see the previous example
  - strong k-consistency where k<n is not enough as well



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?

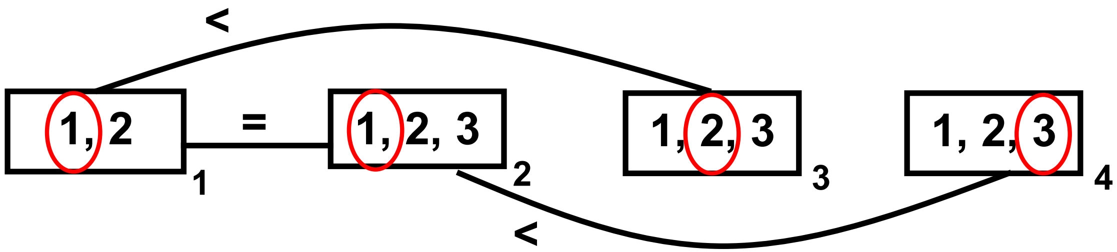


AC is enough!
Because this a tree..

Backtrack-free search

Definition:

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



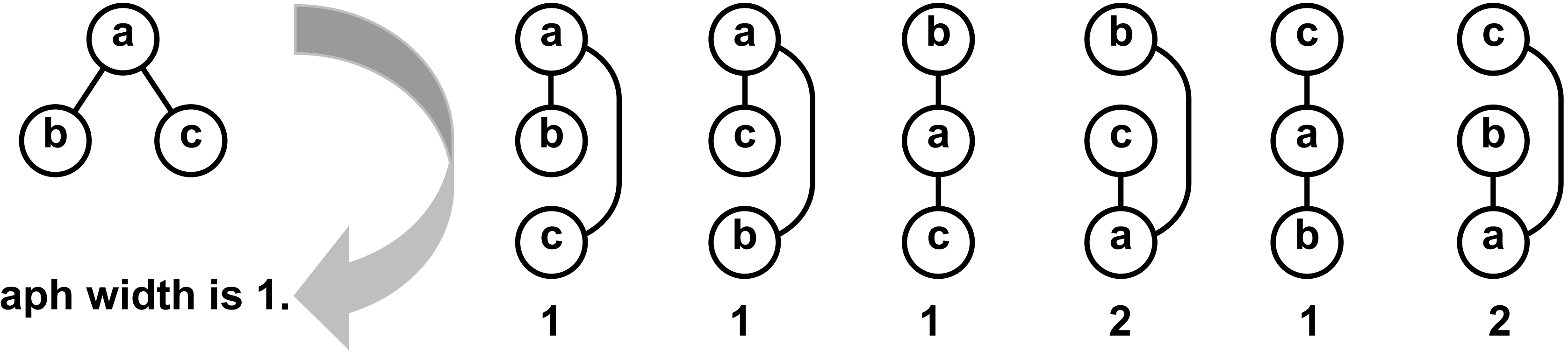
How to find out a sufficient consistency level for a given graph?

Some observations:

- variable must be compatible with all the “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

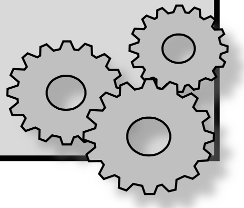
Graph width

- **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
```



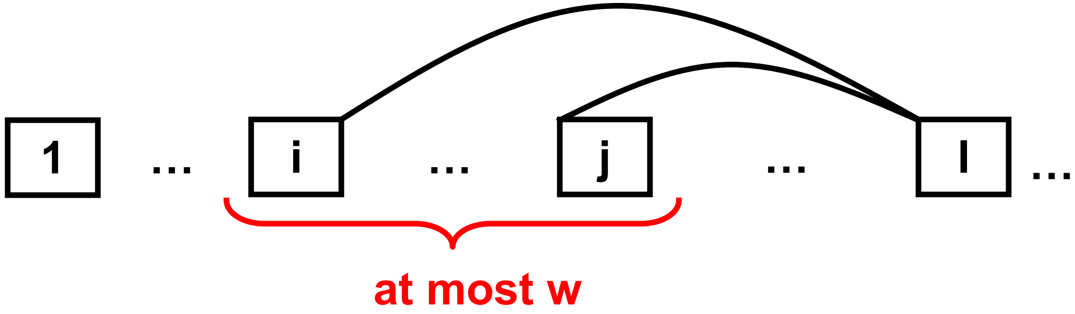
Consistency level based on graph width

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

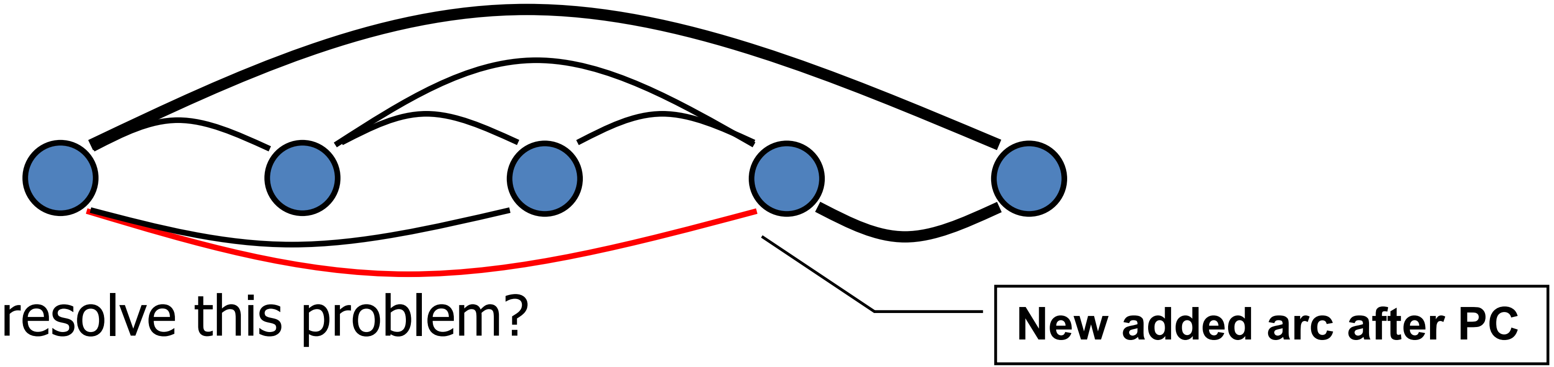


Directional consistency

AC (strong 2-consistency) is enough for trees (the width equals 1).

What about PC and stronger consistencies?

- PC modifies the graph structure – it adds edges!
- So, if we start with a graph of width 2 and make it PC then we may increase graph width!



How to resolve this problem?

Observation 1:

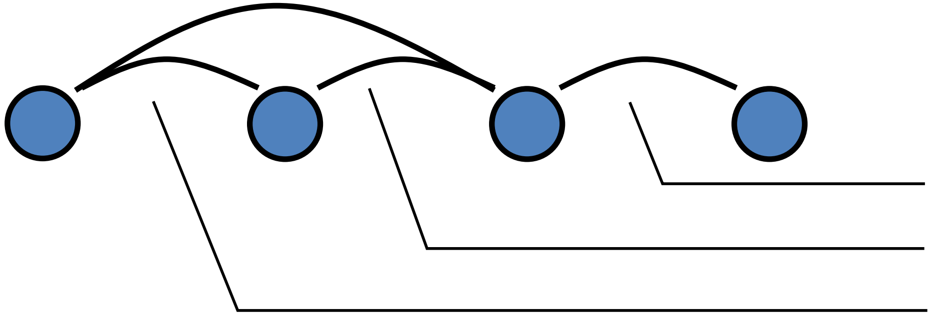
- DAC is enough for trees (we do not need full AC)

Definition:

CSP is directional k-consistent for some order of variables, if any consistent (k-1) tuple of values can be consistently extended to any variable k, that is positioned after all the variables in the tuple.

Observation2:

- we do not need the same consistency level in the whole graph



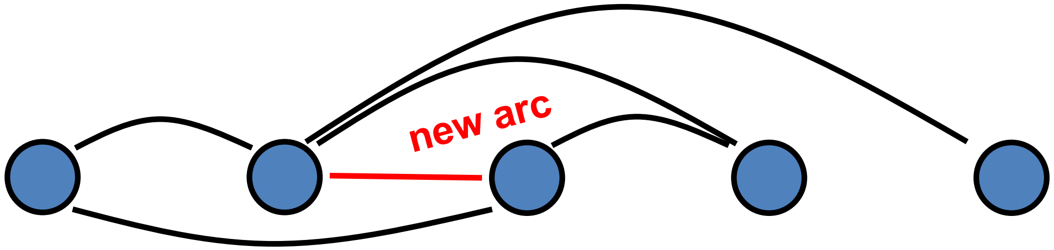
AC is enough

PC is enough

AC is enough

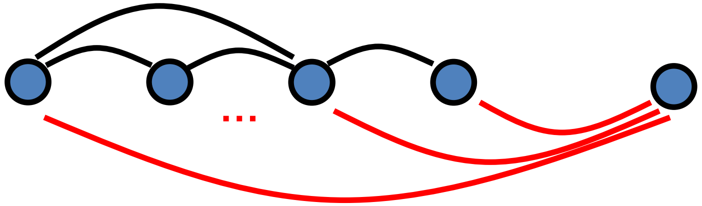
Adaptive consistency

- we can ensure directional i-consistency where i depends on the node width
- nodes are processed upstream the order of nodes in the graph
- new arcs can appear only in the not-yet processed part of the graph
- the final width of the graph can be estimated before running the algorithm

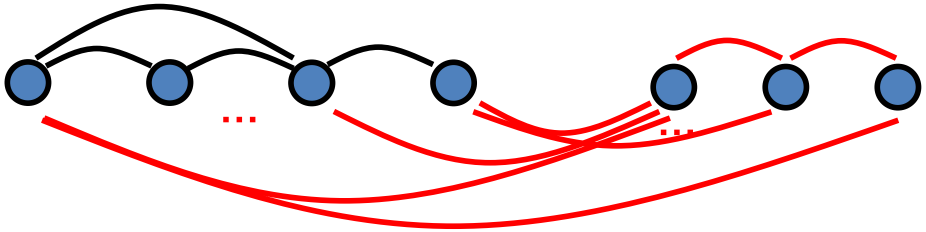


(i,j)-consistency

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



We can do even more!



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

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

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

Inverse consistency

Let i >1 in (i,j)-consistency, then we need to **work with i-tuples** which require a lot of memory (see PC).

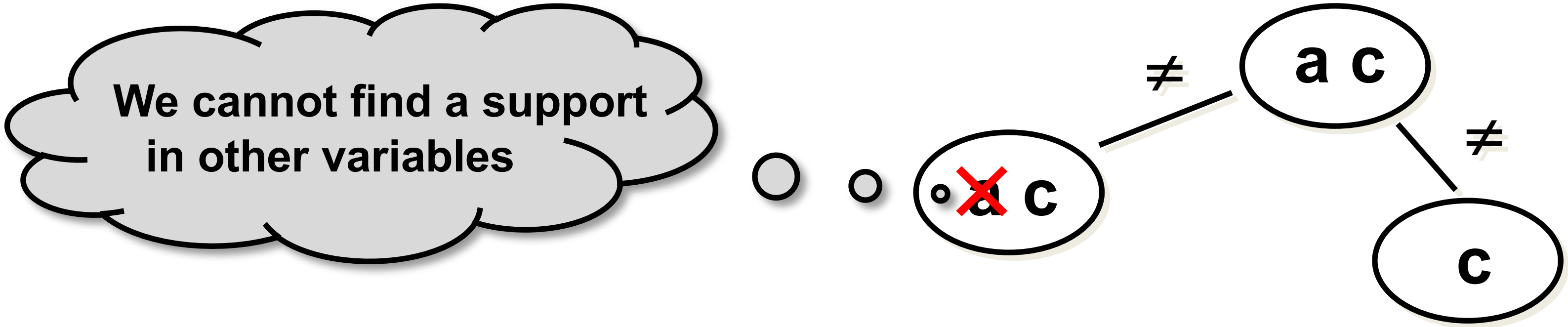
What about **keeping i=1 and increasing j**??

We already did something similar in RPC:
RPC is (1,1)-consistency and sometimes (1,2)-consistency

Definition:

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

  For a given value we are looking for support in other variables.
  If there is no support, we can filter the value out of the domain.
- **arc inverse consistency** = arc consistency
- **path inverse consistency (PIC)** = (1,2)-consistency



Neighbourhood inverse consistency

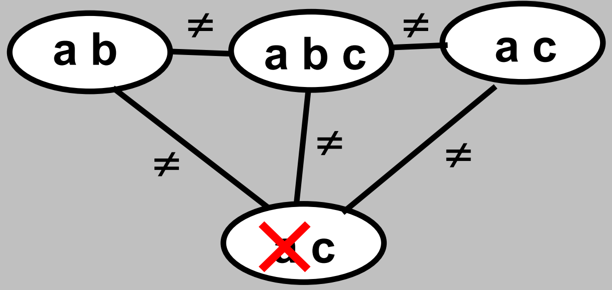
Observation:

Ensuring inverse consistency is useful when at least one of the variables is connected to the core variable.

We can make the neighbourhood of variable consistent.

Definition:

CSP is neighbourhood inverse consistent (NIC) if and only if for each value h of any variable X there exists an assignment of variables in the neighbourhood of X satisfying all the constraints.



```
procedure NIC((V,E))
  Q ← V
  while Q not empty do
    V ← select and delete a variable from Q
    deleted ← false
    for each H in $D_V$ do
      if no solution for Neighbourhood(X) compatible with H then
        remove H from $D_V$
        deleted ← true
        if $D_V$ empty then return fail
    if deleted then Q ← Q ∪ Neighbourhood(X)
  return true
end NIC
```

Singleton consistencies

Can we strengthen any consistency technique?

YES! Let us 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 $\geq$ A-consistency

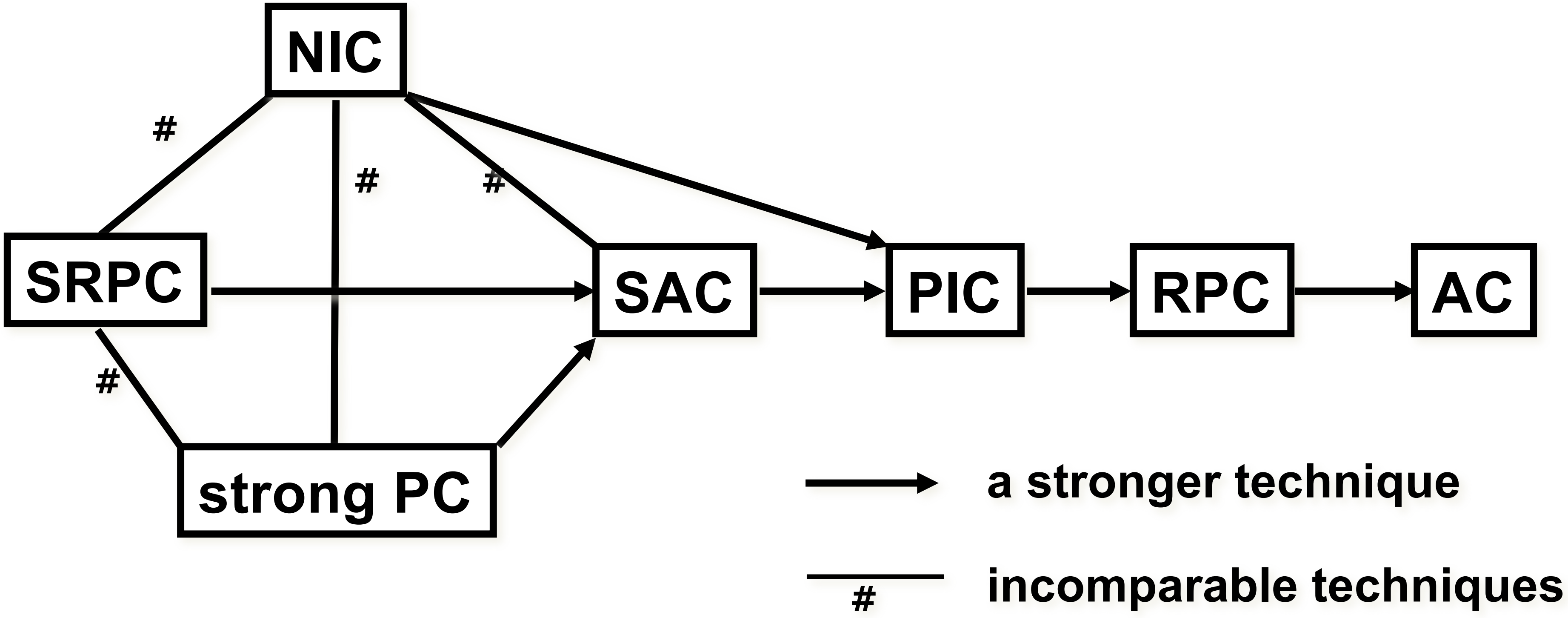
2) A-consistency $\geq$ B-consistency $\Rightarrow$
singleton A-consistency $\geq$ singleton B-consistency

3) singleton (i,j)-consistency > (i,j+1)-consistency (SAC>PIC)

4) strong (i+1,j)-consistency > singleton (i,j)-consistency (PC>SAC)

Consistency techniques at glance

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



Non-binary constraints

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**

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.

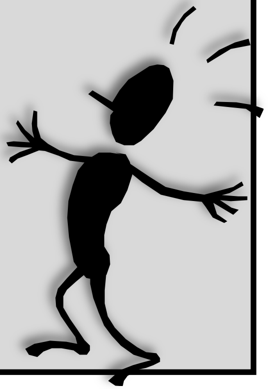
How to make a CSP GAC?

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
```



Bounds consistency

GAC can be computationally expensive, for example for large domains.

Note:

Directional GAC is of no help there, if applying a single method is expensive.

In such cases we can use a **weaker version of GAC**: the GAC property is required only for the boundary values of the domains.

Definition:

The variable V is **bounds consistent** with respect to constraint P, if and only if the bounds of the domain of V are GAC with respect to P.

Notes:

- we assume the values to be ordered
- each variable domain can then be represented as an interval, that is, using two bounds
- this is a frequently used technique in practice (ILOG Solver)

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