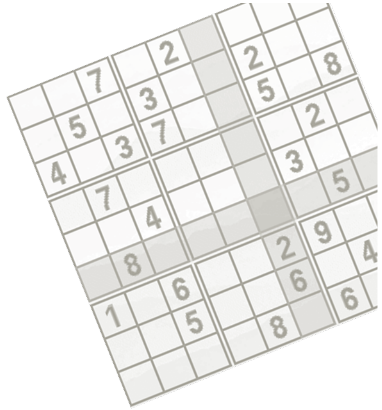


Constraint Programming

Roman Barták
Department of Theoretical Computer Science and Mathematical Logic

Consistency Techniques: Arc Consistency

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?

Unary constraints are converted into variables' domains.

Definition:

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

Algorithm NC

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

Arc consistency (AC)

Since now we will assume binary CSPs only

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

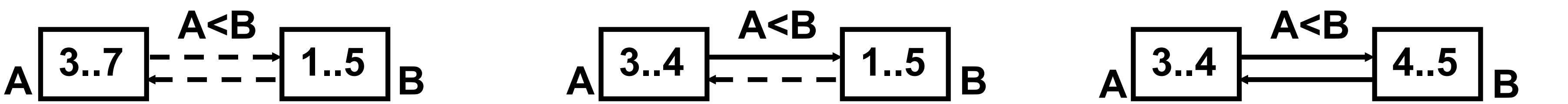
Definition:

- **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 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 valuation $V_i = x$, $V_j = y$ satisfies all the binary constrains on V_i a V_j).

Algorithm of arc revision

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

The procedure also reports the deletion of some value.

How to make a CSP arc consistent?

- Do revision of every arc.

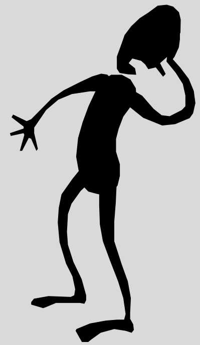
Beware, this is not enough! Pruning the domain may make some already revised arcs inconsistent again.

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
```



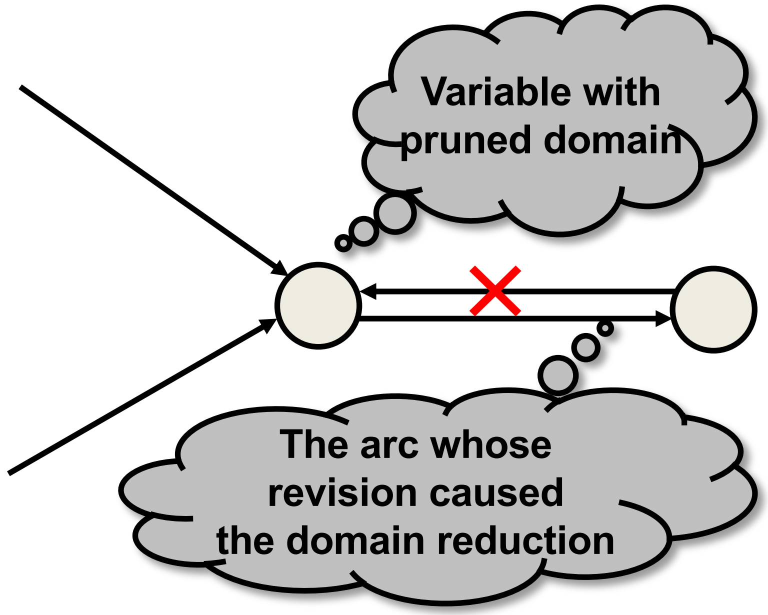
- If a single domain is pruned then revisions of all the arcs are repeated even if the pruned domain does not influence most of these arcs.

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).



A generalised version of the Waltz's labelling algorithm.

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

Algorithm AC-2

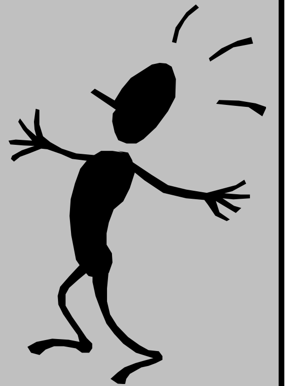
```
procedure AC-2(G)
    for i ← 1 to n do                              % n is a number of variables
        Q ← {(i,j) | (i,j)∈arcs(G), j<i}           % arcs for the base revision
        Q' ← {(j,i) | (i,j)∈arcs(G), j<i}          % arcs for re-revision
        while Q non empty do
            while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

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

1. one queue of arcs for (re-)revisions is enough
2. only the arcs affected by domain reduction are added to the queue (like AC-2)

Algorithm AC-3

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

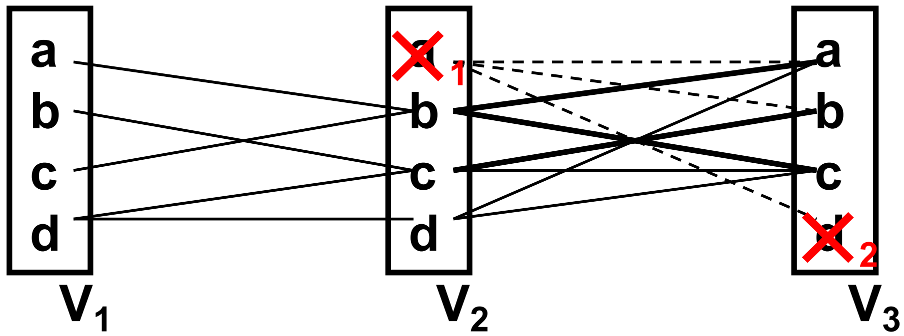


AC-3 is the most widely used consistency algorithm but it is still not optimal.

Observation (AC-3):

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

– These tests are repeated every time the arc is revised.



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

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

Observation:

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

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

Can we compute the support sets once and then use them during re-revisions?

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

Computing and counting supporters

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

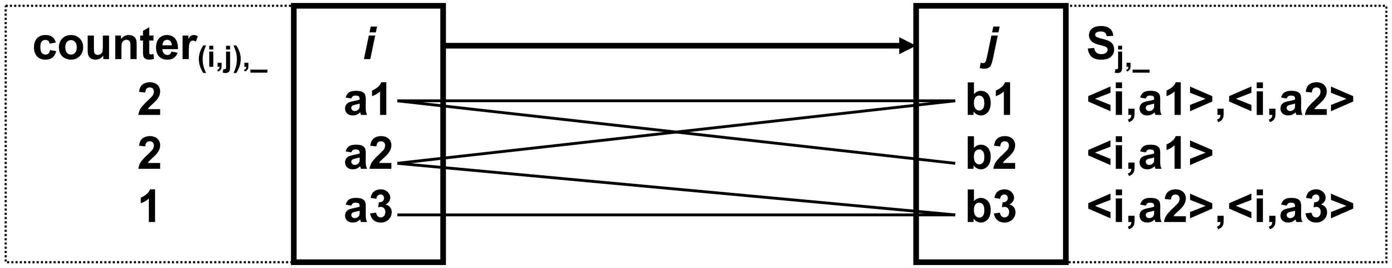
***$S_{j,b}$* - a set of pairs <i,a> such that <j,b> supports them**

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

Using support sets

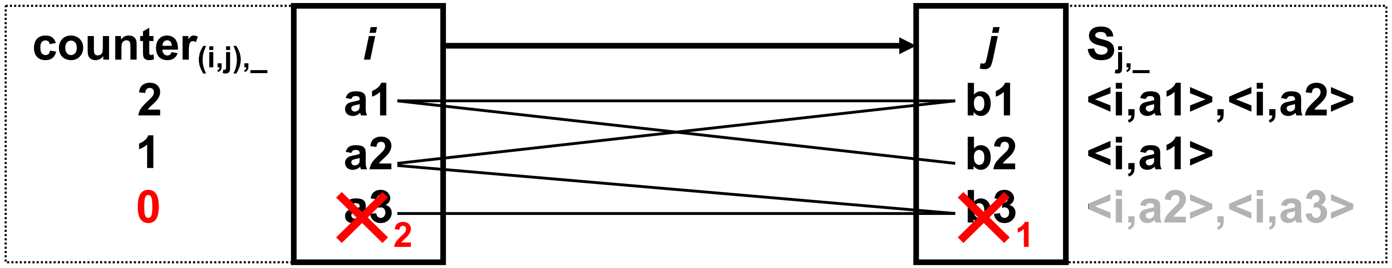
Situation:

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



Using the support sets:

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



Algorithm AC-4

The algorithm AC-4 has the optimal worst-case time complexity!

Algorithm AC-4

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

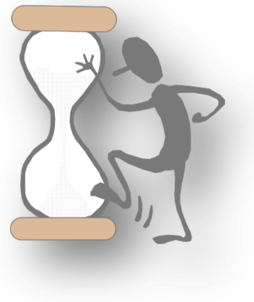
Unfortunately the average efficiency is not so good

… plus there is a big memory consumption!

- **AC-5 (Hentenryck, Deville, Teng 1992)**
  - a generic arc-consistency algorithm
  - can be reduced both to AC-3 and AC-4
  - exploits semantic of the constraint
  - functional, anti-functional, and monotonic constraints
- **AC-6 (Bessiere 1994)**
  - improves memory complexity and average time complexity of AC-4
  - keeps one support only, the next support is looked for when the current support is lost
- **AC-7 (Bessiere, Freuder, Regin 1999)**
  - based on computing supports (like AC-4 and AC-6)
  - exploits symmetry of the constraint

Zhang, Yap (2001)

AC-3.1: optimal AC-3



Some observations:

- AC-3 is not (theoretically) optimal
- AC-4 is (theoretically) optimal but (practically) slow
- AC-6/7 are (practically) faster than AC-4, but quite complicated

What is inefficient in AC-3?

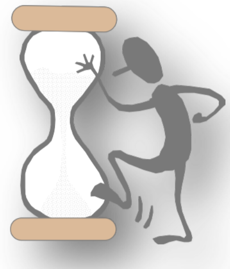
- Looking for supports in REVISE starts from scratch!

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



Version of AC-3 with the queue of variables (AC-8)

```
procedure AC-2001(G)
   Q ← {i | i∈nodes(G)}  % a queue of nodes for revision
   while Q non empty do
       select and delete j from Q
       for each i∈nodes(G) such that (i,j)∈arcs(G) do
           if REVISE2001(i,j) then
               if D_i=∅ then return fail
               Q ← Q ∪ {i}
       end for
   end while
   return true
end AC-2001
```

```
procedure REVISE2001(i,j)
   DELETED ← false
   for each x in D_i do
       if last((i,x),j)∉D_j then
           if ∃y∈D_j y>last((i,x),j)
           & (x,y)∈C(i,j) then
               last((i,x),j) ← y
           else
               delete x from D_i
               DELETED ← true
           end if
   end for
   return DELETED
end REVISE2001
```

Note:

- The algorithm works with difference sets – for each variable we know a set of values deleted since the last revision.

Observation 1:

AC has a directional character but a CSP is not directional.

Observation 2:

AC has to repeat arc revisions and the number of revisions depends on the number of arcs and on the domain size (the while loop).

Can we weaken AC somehow so each arc is revised exactly once?

Definition:

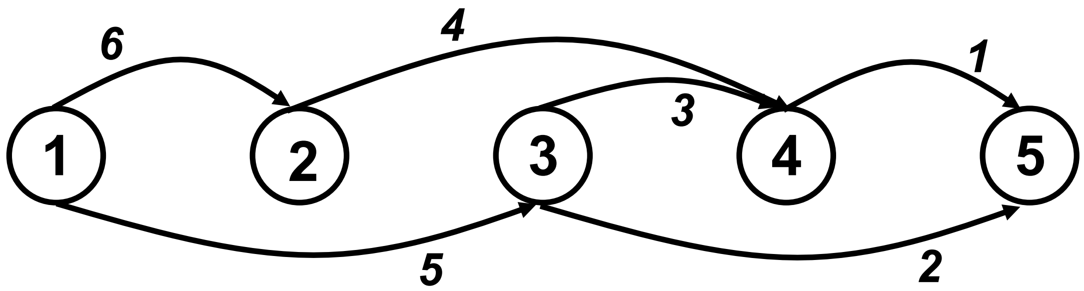
CSP is **directional arc consistent** for a given order of variables if and only if each arc (i,j), such that i<j, is consistent.

Again, each arc is checked once, but only in a one direction.

Algorithm DAC-1

1. Arc consistency is required in one direction only
2. Variables are ordered

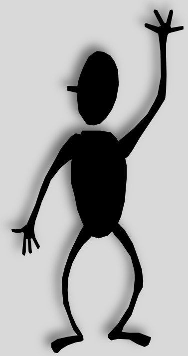
    ↳ no directed cycle!



If arcs are revised in the right order then no revision needs to be repeated!

Algorithm DAC-1

```
procedure DAC-1(G)
    for j = |nodes(G)| to 1 by -1 do
        for each arc (i,j) in G such that i<j do
            REVISE((i,j))
            if Di=∅ then stop with fail
        end for
    end for
end DAC-1
```



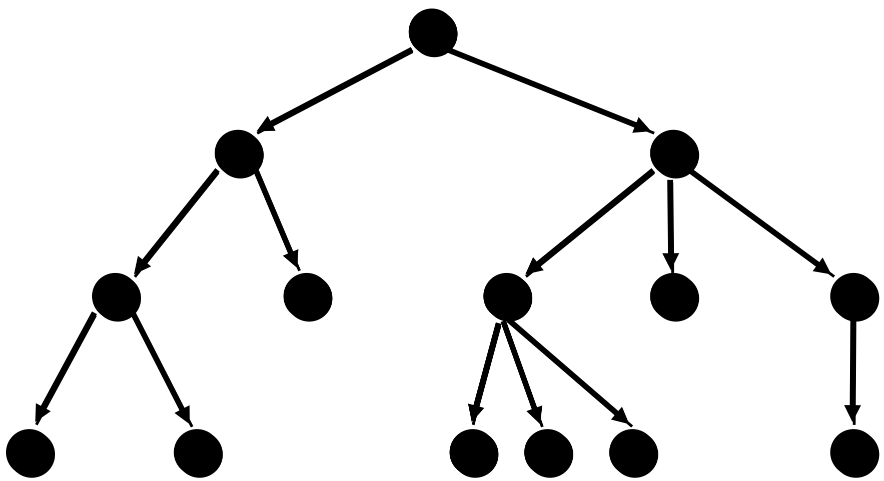
Obviously AC covers DAC (if CSP is AC, then it is DAC).

Is DAC anyhow useful?

- DAC-1 is clearly more efficient than AC-x
- Moreover, there are problems where DAC is enough.

Example: If the constraint network is a tree then we can use DAC to solve the problem in a backtrack-free way.

- **How to order the nodes for DAC?**
- **How to order the nodes for labelling (search)?**



1. Apply DAC in the order of nodes from the root.

2. Label (assign) the nodes starting at root.

DAC ensures that for each child node there is a value compatible with the parent node.

Observation:

A CSP is arc consistent if for some ordering of variables the problem is directional arc consistent in both directions (according to that ordering).

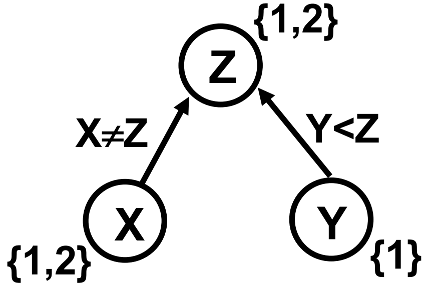
Can we make the problem AC by applying DAC in both directions?

In general NO, but…

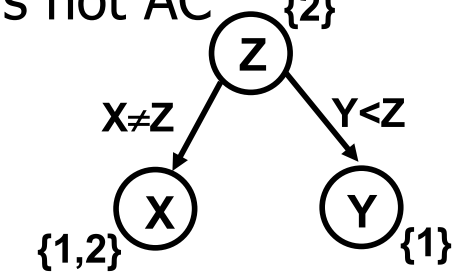
Example:

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

for ordering X,Y,Z there is no change in domains



for ordering Z,Y,X the domain of Z is pruned, but the problem is not AC

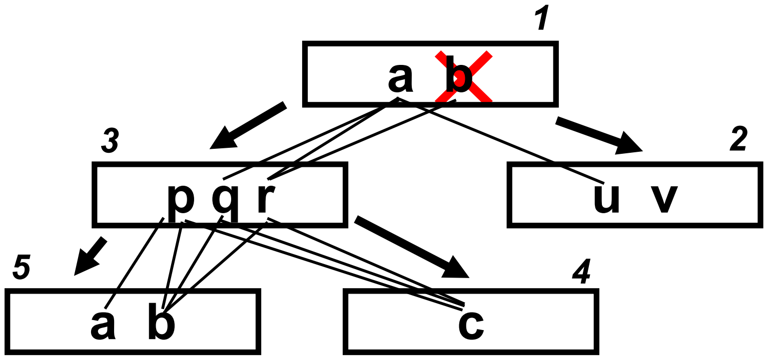


If we first try the ordering Z,Y,X, then we get AC!

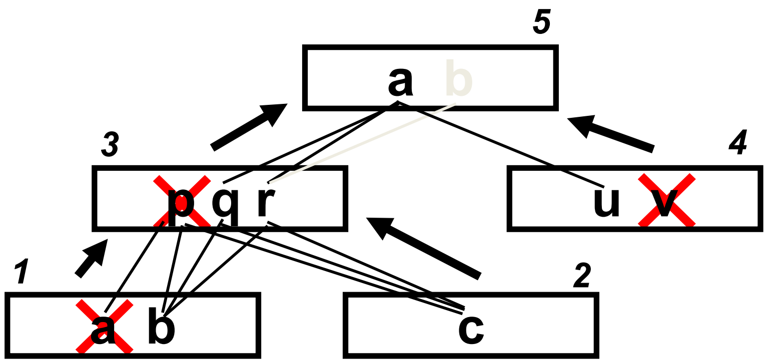
If we apply DAC to a tree-structured CSP first for the ordering from the root and then in the reverse direction from leafs then we obtain AC.

Proof:

after the first run of DAC we ensure that each value in a parent node has a support (a compatible value) in all child nodes



if some value is deleted during the **second run of DAC** (in the reverse order) then this value is not a support for any value in the parent node (so the values in the parent do not lose supports)



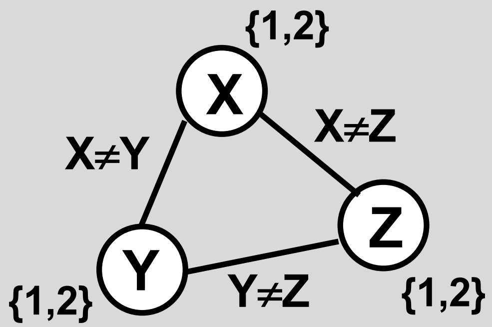
together: each value has a support in all child nodes (the first DAC run) and in the parent node too (the second DAC run) so the value is AC

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 **decreases the search space.**

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