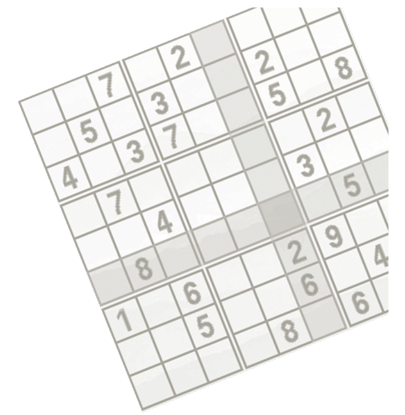


# Constraint Programming

**Roman Barták**
Department of Theoretical Computer Science and Mathematical Logic

**Combining Search and Inference**

So far we have two methods to solve CSPs:

- **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!

**A core constraint satisfaction method:**

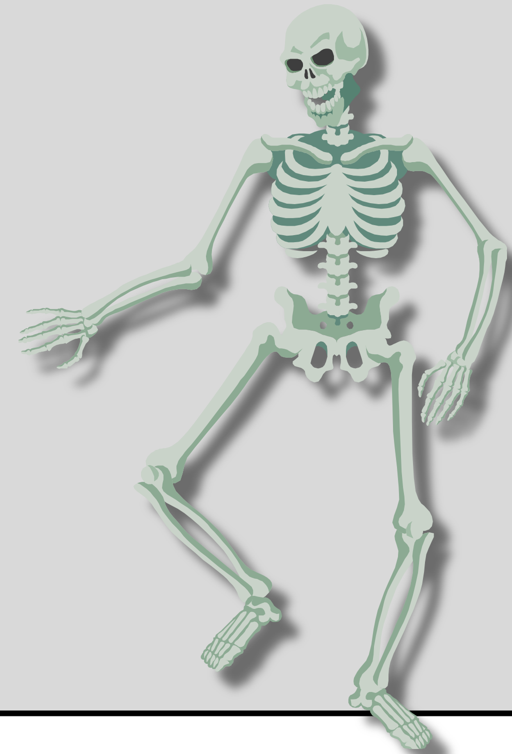
- **label (instantiate) the variables** one by one
  - the variables are ordered and instantiated in that order
- **verify (maintain) consistency** after each assignment

**Skeleton of the search algorithm**

```
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_cv 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 procedure*



"Maintain" **consistency among the already instantiated variables**.

- „look back" = look to already labelled variables

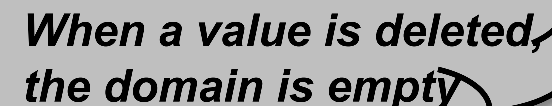
What's result of consistency maintenance among labelled variables?

- a **conflict** (and/or its source - a violated constraint)

Backtracking is a basic look-back method.

**Backward consistency checks**

```
procedure AC-BT(G,cv)
    Q ← {(V_i,V_cv) in arcs(G), i<cv}      % arcs to labelled variables.
    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.

**It is better to prevent failures than to detect them only!**

Consistency techniques can remove incompatible values for future (=not yet instantiated) variables.

Forward checking ensures consistency between the currently instantiated variable and the variables connected to it via constraints.

**Forward consistency checks**

```
procedure AC-FC(G,cv)
    Q ← {(V_i,V_cv) in arcs(G), i>cv}     % arcs to future variables
    consistent ← true
    while consistent & Q non empty do
        select and delete any arc (V_k,V_m) from Q
        if REVISE(V_k,V_m) then
            consistent ← not empty D_k
        end if
    end while
    return consistent
end AC-FC
```

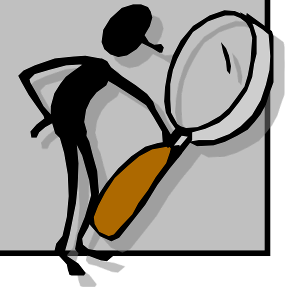
# Partial look-ahead

**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 look-ahead 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 for
    end for
    return true
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)!

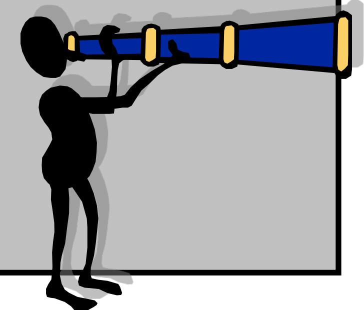
# Full look-ahead

**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**

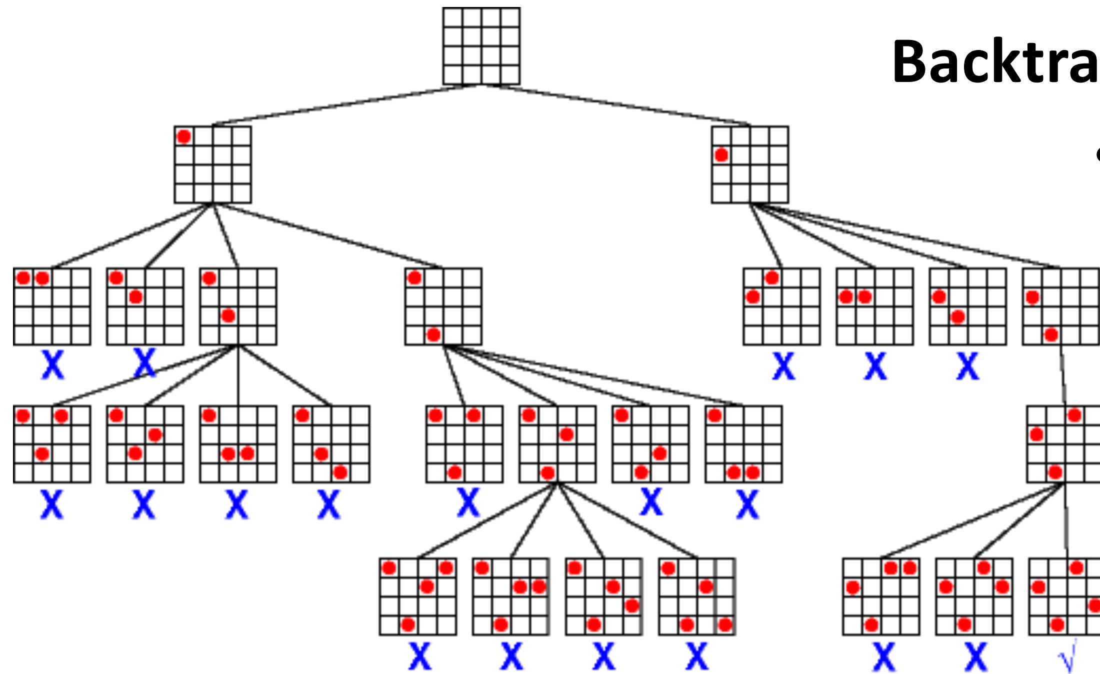
**procedure** AC3-LA(G,cv)
  $Q \leftarrow \{(V_i,V_{cv})$ in arcs(G),i>cv$\}$    % start with arcs going to cv
  consistent $\leftarrow$ true
  **while** consistent & Q non empty **do**
    select and delete any arc $(V_k,V_m)$ from Q
    **if** REVISE$(V_k,V_m)$ **then**
      $Q \leftarrow Q \cup \{(V_i,V_k) \mid (V_i,V_k)$ in arcs(G),$i \neq k, i \neq m$, i>cv$\}$
      consistent $\leftarrow$ not empty $D_k$
    **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)*

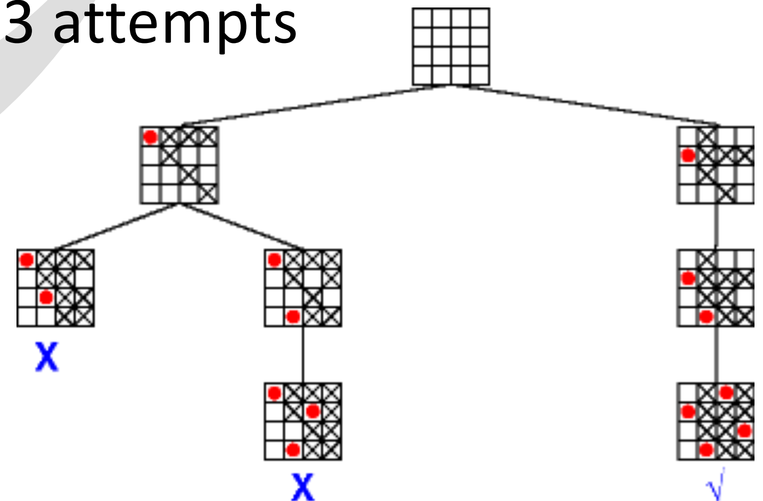



**Backtracking** is not very good

- 19 attempts

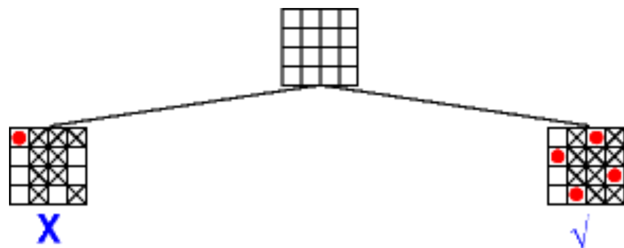
**Forward checking** is better

3 attempts

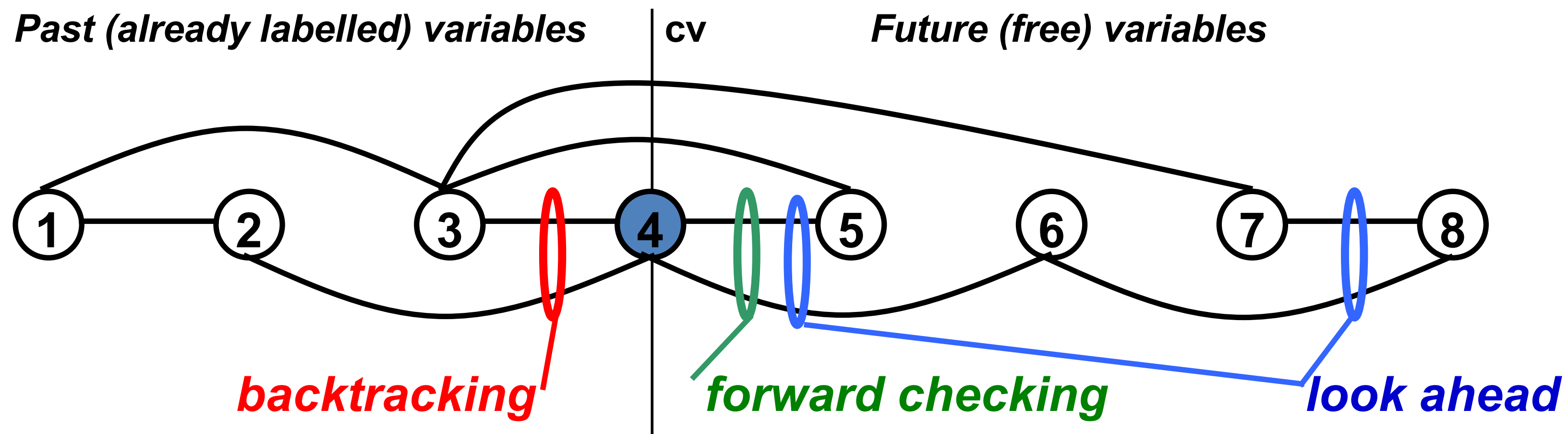



And the winner is **Look Ahead**

2 attempts



# Constraint propagation at glance




- Propagating through more constraints removes more inconsistencies (BT < FC < PLA < LA), of course it increases complexity of the labelling step.
- **Forward Checking does no 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 in labelling influences significantly efficiency of constraint solvers (e.g. in a tree-structured CSP).

**Which variable ordering should be chosen in general?**

**FAIL FIRST principle**

**„select the variable whose instantiation will lead to a 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)

**„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

Order of values in labelling influences significantly efficiency (if we choose the right value each time, no backtrack 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

**Note:** SUCCEED FIRST does not go against FAIL FIRST !

- **prefer values with more supports**
  - this information can be found in AC-4
- **prefer a value leading to less domain reduction**
  - this information can be computed using singleton consistency
- **prefer a value simplifying the problem**
  - solve approximation of the problem (e.g. a tree)

**Generic heuristics are usually too complex** for computation.

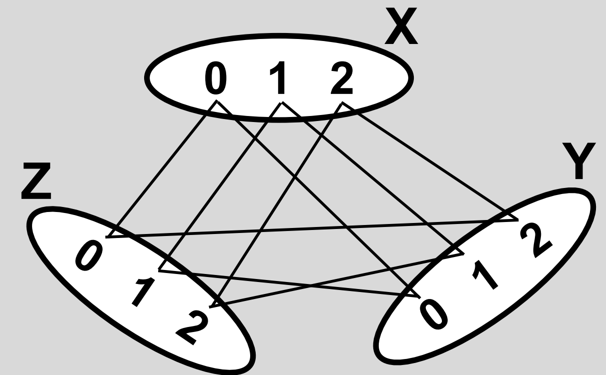
**It is better to use problem-driven heuristics that propose the value!**

So far we assumed search by labelling, i.e. assignment of values to variables.

- assign a value, propagate and backtrack in case of failure (try other value)
  - this is called **enumeration**
- propagation is used only after instantiating a variable

**Example:**

- X,Y,Z in 0,…,N-1 (N is constant)
- X=Y, X=Z, Z=(Y+1) mod N
  - problem is AC, but has no solution
  - enumeration will try all the values
  - for n=$10^7$ runtime 45 s. (at 1.7 GHz P4)




**Can we use faster labelling?**

## Other branching strategies

**Enumeration** resolves disjunctions in the form $\mathbf{X=0 \vee X=1 \ldots X=N-1}$

- if there is no correct value, the algorithm tries all the values

**We can use propagation when we find some value to be wrong!**

- that value is deleted from the domain which starts propagation that filters out other values
- we solve disjunctions in the form $\mathbf{X=H \vee X\neq H}$
- this is called **step labelling** (usually a default strategy)
- the previous example solved in 22 s. by trying and refuting value 0 for X

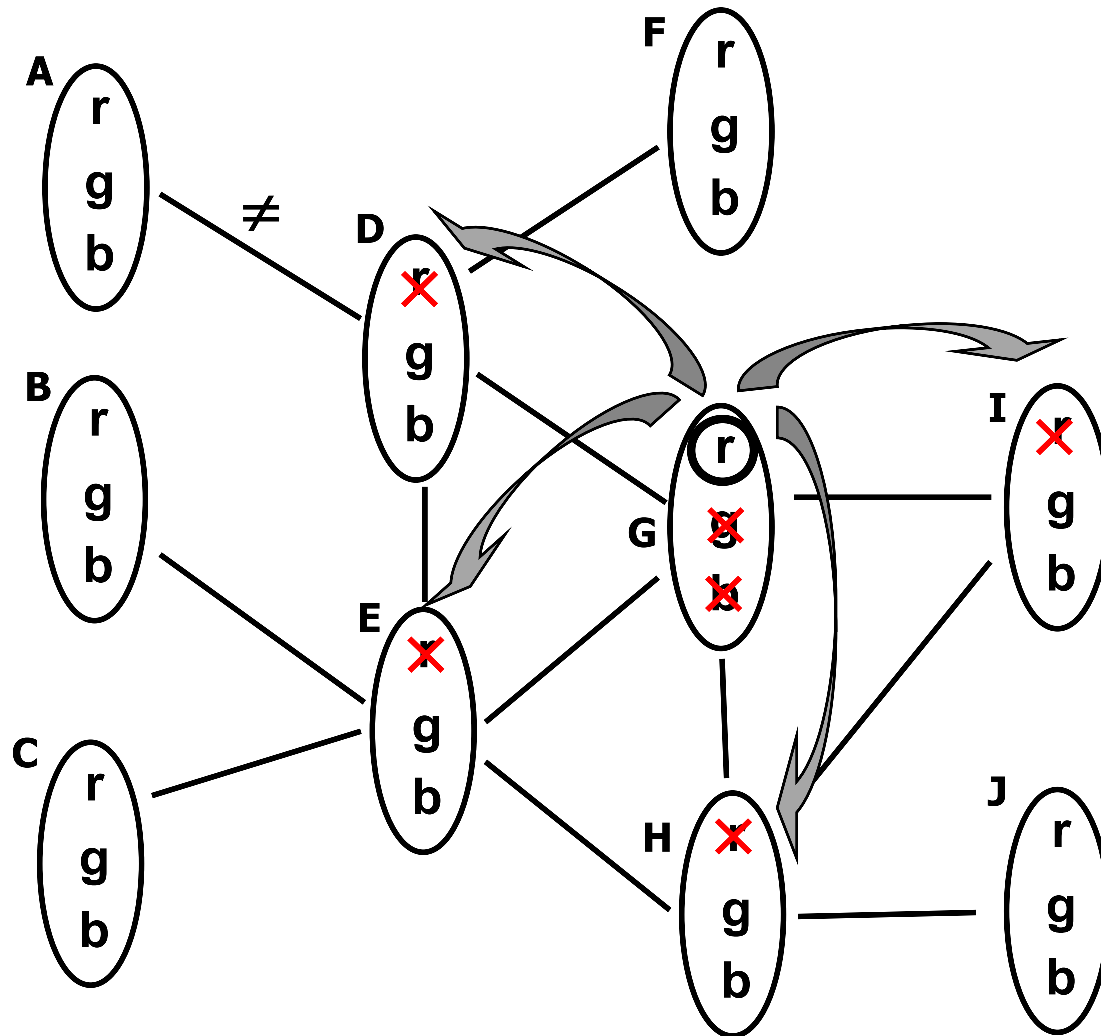
  Why so long?

  - In each AC cycle we remove just one value.

Another typical branching is **bisection/domain splitting**

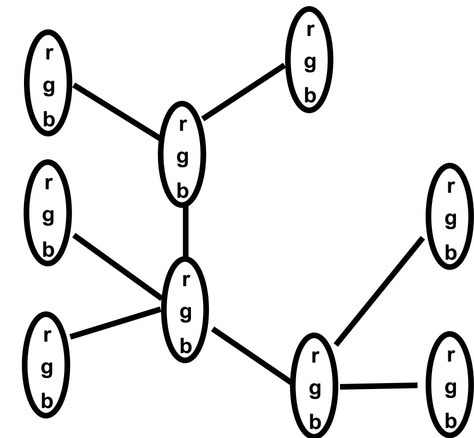
- we solve disjunctions in the form $\mathbf{X\leq H \vee X>H}$, where H is a value in the middle of the domain

# Motivation example




1) instantiate variable
2) make the problem AC
3) solve the rest without backtracks

Why?




- If the constraint network is acyclic then we can find a solution using backtrack-free search (AC is enough)!
- However, constraint networks are not usually acyclic!

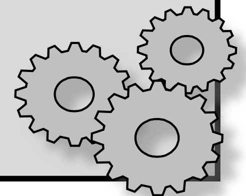
If we can make a constraint network acyclic, we can solve the problem with backtrack-free search.

**How to remove cycles?**

- instantiate **variables on cycles** (instantiation = removal from a graph)
- We can instantiate just variables in a **cycle-cutset**!
- *cycle cutset* = a set of vertices such that their removal splits all cycles

**Algorithm Cycle Cutset**

```
procedure cycle-cutset(G)
    C ← find-cycle-cutset(G)
    while ex. assignment of variables in C satisfying all constraints do
        LC ← a(nother) assignment of variables in C satisfying all constraints
        enforce DAC from C to the remaining variables
        if all domains are non-empty then
            LR ← assignment of remaining variables (out of C) using backtrack-free
            search
            return LC+LR
        end if
    end while
    return fail
end cycle-cutset
```

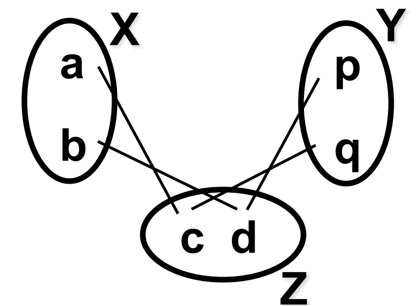


During labelling of the cycle-cutset there is thrashing.

**Example 1:**

- X,Y,Z – order of variables
- assume that there is no compatible value in Z for X=a
- hence, each time we use X=a, we get a failure

**We can make the problem arc consistent before search!**




**Example 2:**

- X,Y,Z – order of variables
- (a,c), (b,d) $\in C_{XZ}$, (a,d) $\notin C_{XZ}$, (p,c) $\notin C_{YZ}$, (p,d), (q,c) $\in C_{YZ}$
- each time we use X=a, Y=p together, we get a failure
- making the problem AC does not discover this problem

**we need maintaining arc consistency during search**

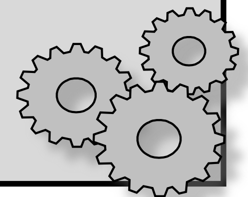
**What about combining MAC with the ideas of CC?**

# MAC Extended

- **less instantiations**
  - we instantiate (during search) only a small subset of variables
- **less filtering**
  - we maintain only partial arc consistency (that can be extended to complete arc consistency)

**Algorithm MACE**

1) make the problem arc consistent (**if** inconsistent **then** failure)
2) split variables to two sets:
   a cycle-cutset C
   a set U of variables outside any cycle
   (note that U is not necessarily a complement to C)
3) remove U from the network (U will not participate in AC)
4) **while** $C \neq 0$ **do**
   4a) select a value for some variable in C
   4b) make the problem arc consistent
   **if** inconsistent **then** go to 4a (or to the previously instantiated variable)
   4c) remove variables with singleton domains (put them to U)
   4d) remove variables outside any cycle (put them to U)
5) re-connect variables from U to the network
   and make the problem arc consistent
6) find instantiation of other variables using backtrack-free search



# MACE example

1. Make problem AC
2. Find C and U
3. Disconnect U
4. Assign value to C
5. Make problem AC
6. Disconnect variables outside cycles
7. Reconnect U
8. Backtrack-free search

CC and MACE need a cycle-cutset (smaller CC is better)

No polynomial time algorithm for finding minimal CC is known.

**Some heuristics:**

- put to CC variables according to their degree (higher degree first)
- "  + include only variables that are in some cycle
- put to CC variables according to number of cycles they participate in

**Cycle-cutset algorithm**

**procedure** find-cycle-cutset(G)
    (V,E) = G
    Q $\leftarrow$ order elements in V by descending order of their degrees in G
    CC $\leftarrow$ {}
    **while** the graph G is cyclic **do**
        V $\leftarrow$ first element in Q
        CC $\leftarrow$ CC $\cup$ {V}
        Q $\leftarrow$ Q - {V}
        remove V and edges involving it from the constraint graph G
    **end while**
    return CC
**end** find-cycle-cutset

**© 2020 Roman Barták**

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