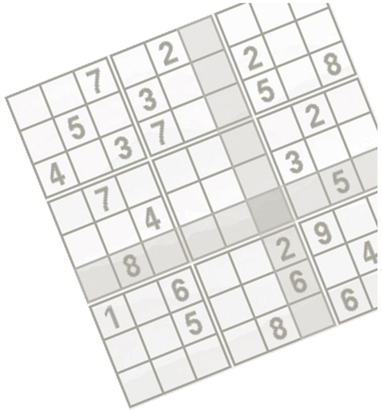


ESSAI & ACAI 2024
ATHENS - GREECE

Introduction to Constraint Satisfaction

Roman Barták
Charles University, Prague (CZ)

Search techniques to solve CSPs

Constraint Satisfaction Problem (CSP) consists of:

- a finite set of **variables**
- **domains** – finite sets of possible values for variables
- a finite set of **constraints**
  - constraint **arity** = the number of constrained variables

A feasible solution of a constraint satisfaction problem is a complete consistent assignment of values to variables.

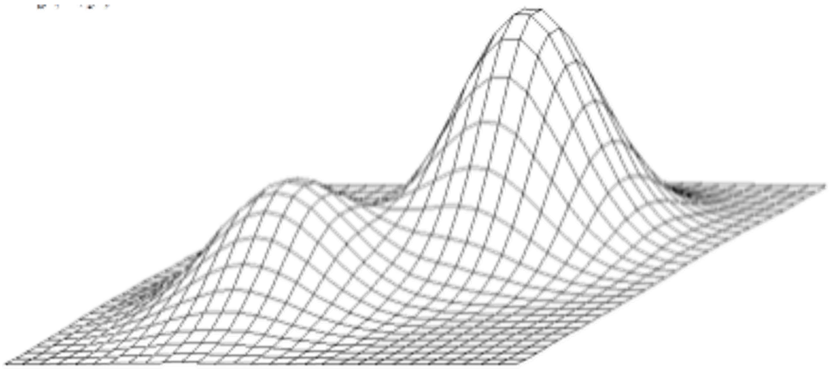
- **complete** = each variable has assigned a value
- **consistent** = all constraints are satisfied

Generate and test explores complete but inconsistent assignments until a complete consistent assignment is found.

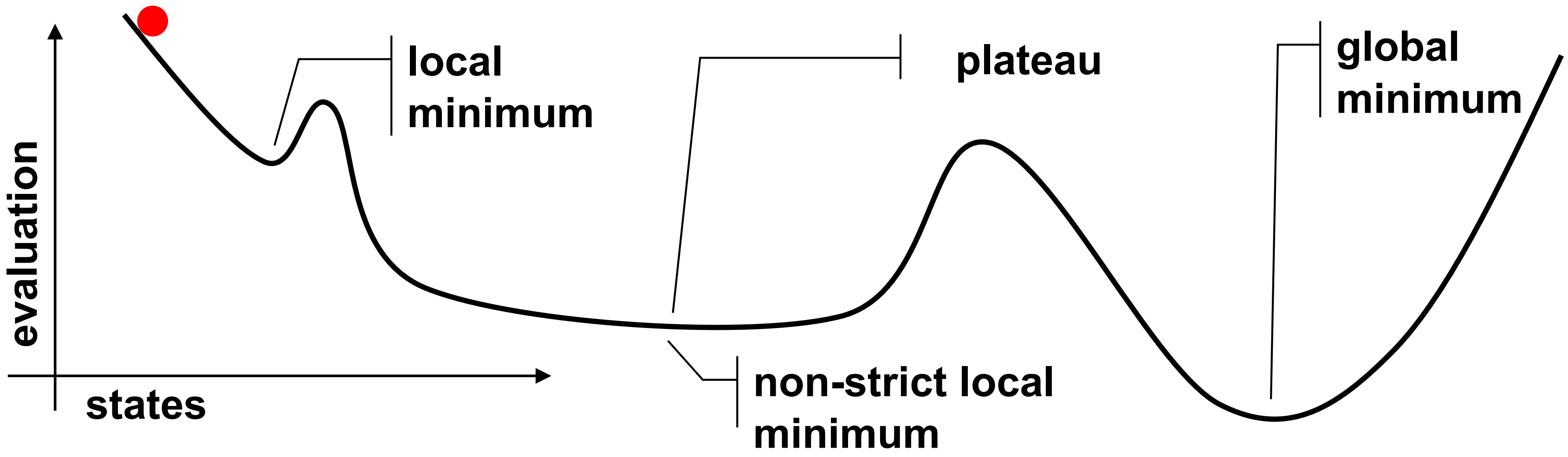
Weakness of GT – the generator does not exploit fully the result of testing

The next assignment can be constructed in such a way that constraint violation is smaller.

- only “small” (**local**) **changes** of the assignment are allowed
- the next assignment should be “better” than the current one
  - better = more constraints are satisfied
- assignments are not necessarily generated systematically
  - we lost completeness, but
  - we (hopefully) get better efficiency



- **state** - a complete assignment of values to variables
- **evaluation** - a value of the objective function (# violated constraints)
- **neighbourhood** - a set of states locally different from the current state (the states differ from the current state in the value of one variable)
- **local optimum** - a state that is not optimal and there is no state with better evaluation in its neighbourhood
- **strict local optimum** - a state that is not optimal and there are only states with worse evaluation in its neighbourhood
- **non-strict local optimum** - local optimum that is not strict
- **plateau** - a set of neighbouring states with the same evaluation
- **global optimum** - the state with the best evaluation



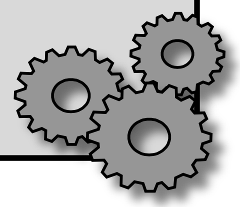
Hill Climbing

Hill climbing is perhaps the most known technique of local search.

- start at randomly generated state
- look for the best state in the neighbourhood of the current state
  - **neighbourhood** = differs in the value of any variable
  - neighbourhood size = $\Sigma_{i=1..n}(D_i-1)$ (= $n*(d-1)$)
- "escape" from the local optimum via restart

Algorithm Hill Climbing

```
procedure hill-climbing(Max_Steps)
    restart: s ← random assignment of variables;
    for j:=1 to Max_Steps do                % restricted number of steps
        if eval(s)=0 then return s
        if s is a strict local minimum then
            go to restart
        else
            s ← neighbourhood with the smallest evaluation value
        end if
    end for
    go to restart
end hill-climbing
```



Observation:

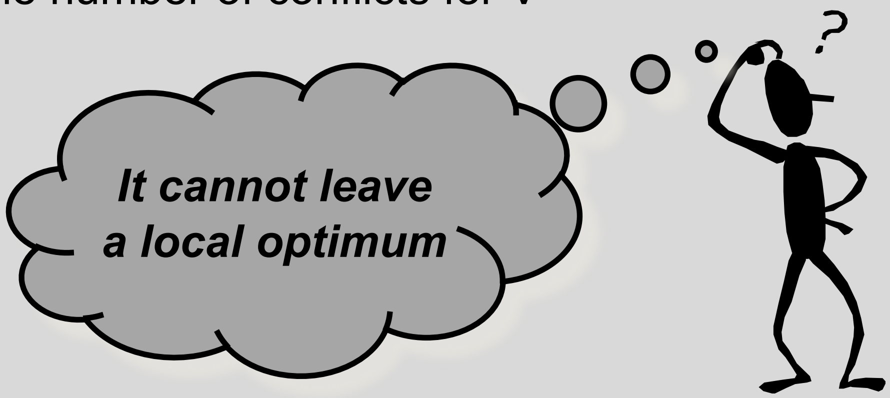
- the hill climbing neighbourhood is pretty large (n*(d-1))
- only change of a conflicting variable may improve the evaluation

Min-conflicts method

- select randomly a variable in conflict and try to improve it
  - **neighbourhood** = different values for the selected variable *i*
  - neighbourhood size = (D_i-1) (= (d-1))

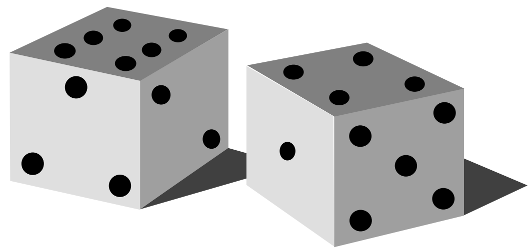
Algorithm Min-Conflicts

```
procedure MC(Max_Moves)
    s ← random assignment of variables
    nb_moves ← 0
    while eval(s)>0 and nb_moves<Max_Moves do
        choose randomly a variable V in conflict
        choose a value v' that minimises the number of conflicts for V
        if v' ≠ current value of V then
            assign v' to V
            nb_moves ← nb_moves+1
        end if
    end while
    return s
end MC
```



How to leave a local optimum without restarting (i.e. via a local step)?

– By adding some "noise" to the algorithm!



Random walk

– **a state from the neighbourhood is selected randomly** (e.g., the value is chosen randomly)
– such technique can hardly find a solution
– so it needs some guide
  - Random walk can be combined with the heuristic guiding the search process via probability distribution:
    – p - probability of using a random step
    – (1-p) - probability of using the heuristic guide

Min-Conflicts Random Walk

MC guides the search (i.e. satisfaction of all the constraints) and RW allows us to leave the local optima.

Algorithm Min-Conflicts-Random-Walk

```
procedure MCRW(Max_Moves,p)
    s ← random assignment of variables
    nb_moves ← 0
    while eval(s)>0 and nb_moves<Max_Moves do
        if probability p verified then
            choose randomly a variable V in conflict
            choose randomly a value v' for V
        else
            choose randomly a variable V in conflict
            choose a value v' that minimises the number of conflicts for V
        end if
        if v' ≠ current value of V then
            assign v' to V
            nb_moves ← nb_moves+1
        end if
    end while
    return s
end MCRW
```

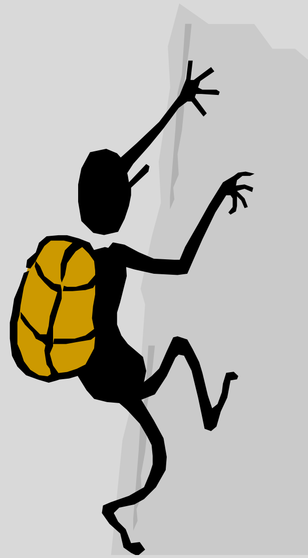
$0.02 \leq p \leq 0.1$

Random walk can be combined with the hill climbing heuristic too.

Then, no restart is necessary.

Algorithm Steepest-Descent-Random-Walk

```
procedure SDRW(Max_Moves,p)
    s ← random assignment of variables
    nb_moves ← 0
    while eval(s)>0 and nb_moves<Max_Moves do
        if probability p verified then
            choose randomly a variable V in conflict
            choose randomly a value v' for V
        else
            choose a move <V,v'> with the best performance
        end if
        if v' ≠  current value of V then
            assign v' to V
            nb_moves ← nb_moves+1
        end if
    end while
    return s
end SDRW
```

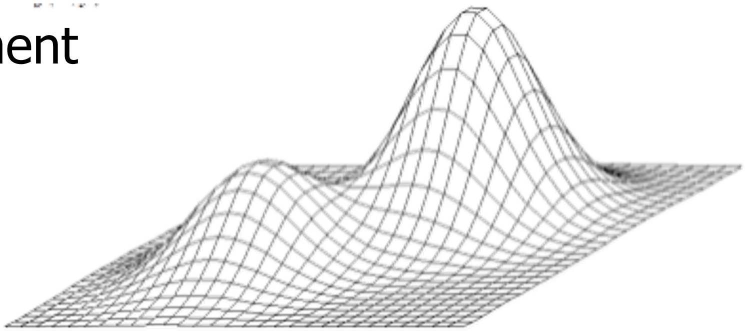


LS methods explore complete but possible inconsistent assignments until a consistent assigned is found

- opposite to GT, they generate a new assignment based on the current assignment with the goal to increase the number of satisfied constraints

Local search algorithm is defined by:

- **neighbourhood** of the current assignment (state) and a method to **select the next assignment** from the neighbourhood (**intensification**)
  - HC heuristic – select the best assignment different at one variable from the current assignment
    - sometimes, the first better assignment from the neighbourhood is taken
  - MC heuristic – select the best assignment different at one selected conflict variable from the current assignment
- a method for **escaping from a local optimum** (**diversification**)
  - restart – start in a completely new assignment
  - RW – select the next assignment randomly
  - Tabu – forbid some assignments



Back to GT

Back to **generate-and-test**:

- **generates** a solution candidate (a complete assignment) and **tests** all the constraints together at the end
- solution candidates are generated systematically, for example:

```
procedure generate_first(Variables)
        for each V in Variables do
                label V by the first value in D_V
        end for
end generate_first
procedure generate_next(Assignment)
        find first X in Assignment such that all following variables are labelled by the last value
        from their respective domains (name the set of these variables Vs)
        if X is labelled by the last value then return fail
        label X by next value in D_X
        for each Y in Vs do
                assign first value in D_Y to Y
        end for
end generate_next
```

We can verify satisfaction of a constraint as soon as we know the values of all constrained variables!

- the test stage is done during the generation stage

Probably the most widely used systematic search algorithm that **verifies the constraints as soon as possible**.

- upon failure (any constraint is violated) the algorithm goes back to the last instantiated variable and tries a different value for it
- depth-first search

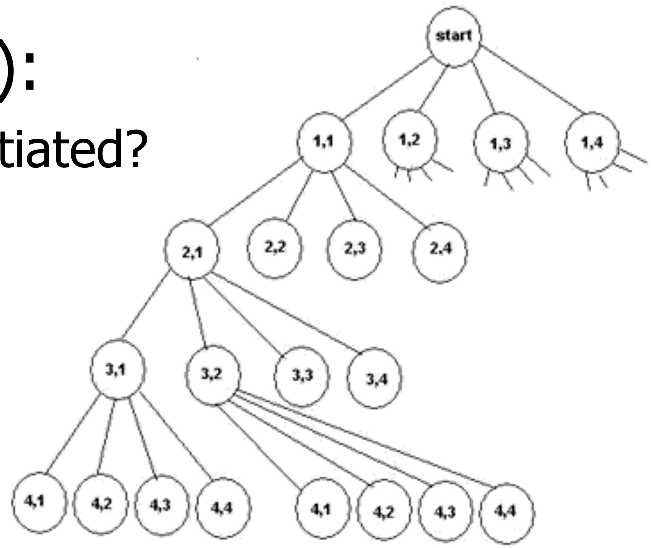
The core principle of applying backtracking to solve a CSP:

1. assign values to variables one by one
2. after each assignment verify satisfaction of constraints with known values of all constrained variables

Open questions (to be answered later):

- What is the order of variables being instantiated?
- What is the order of values tried?

Backtracking explores partial consistent assignments until it finds a complete (consistent) assignment.



Chronological Backtracking (a recursive version)

procedure BT(X:variables, V:assignment, C:constraints)

   if X={} **then** return V

   x ← select a not-yet assigned variable from X

   for each value h from the domain of x **do**

      if constraints C are consistent with V ∪ {x/h} **then**

         R ← BT(X – {x}, V ∪ {x/h}, C)

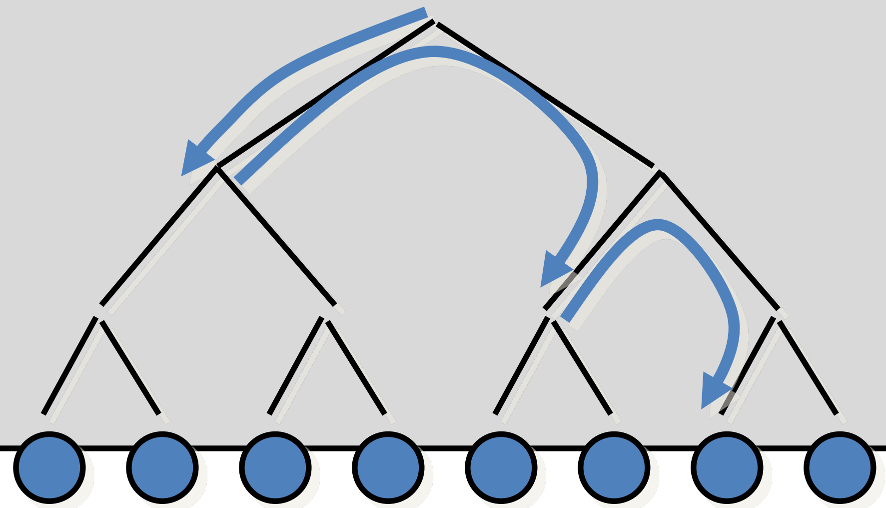
         if R ≠ fail **then** return R

   end for

   return fail

end BT

Call as BT(X, {}, C)



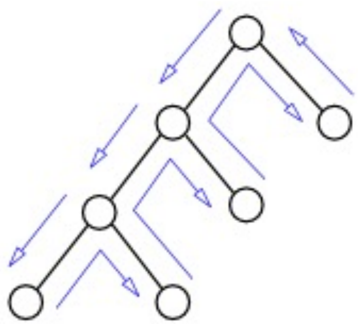
Note:

If it is possible to perform the test stage for a partially generated solution candidate then BT is always better than GT, as BT does not explore all complete solution candidates.

Chronological Backtracking (an iterative version)

```
procedure BT(X:variables, C:constraints)
    i ← 1, D‘i ← Di
    while 1 ≤ i ≤ n do
        instantiate_and_check(i, C)
        if xi = null then i ← i – 1 else i ← i + 1, D‘i ← Di
    end while
    if i = 0 then return fail
    return {x1,…, xn}
end BT
```

```
procedure instantiate_and_check(i, C:constraints)
    while D‘i is not empty do
        select and delete some element b from D‘i
        xi ← b
        if constraints C are consistent with {x1,…, xi} then return
    end while
    xi ← null
end instantiate_and_check
```



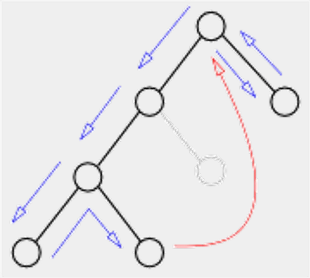
Look Back

- **thrashing**
  - throws away the reason of the conflict

Example: A,B,C,D,E :: 1..10, A>E

- BT tries all the assignments for B,C,D before finding that A≠1

Solution: **backjumping** (jump to the source of the failure)



- **redundant work**
  - unnecessary constraint checks are repeated

Example: A,B,C,D,E :: 1..10, B+8<D, C=5*E

- when labelling C,E the values 1,..,9 are repeatedly checked for D

Solution: **backmarking**, **backchecking** (remember (no-)good assignments)

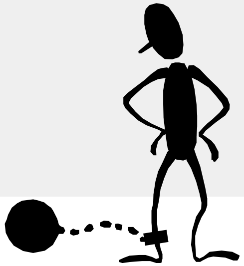
Look Ahead

- **late detection of the conflict**
  - constraint violation is discovered only when the values are known

Example: A,B,C,D,E :: 1..10, A=3*E

- the fact that A>2 is discovered when labelling E

Solution: **forward checking** (forward check of constraints)



Backjumping is a method to **remove thrashing.**

How to do it?

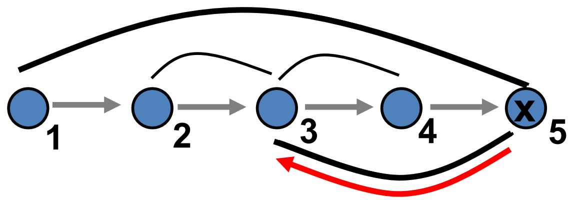
1) identify the source of the conflict (impossibility to assign a value)
2) jump to the past variable in conflict

The same forward run as in backtracking, only the back-jump can be longer to skip irrelevant assignments!

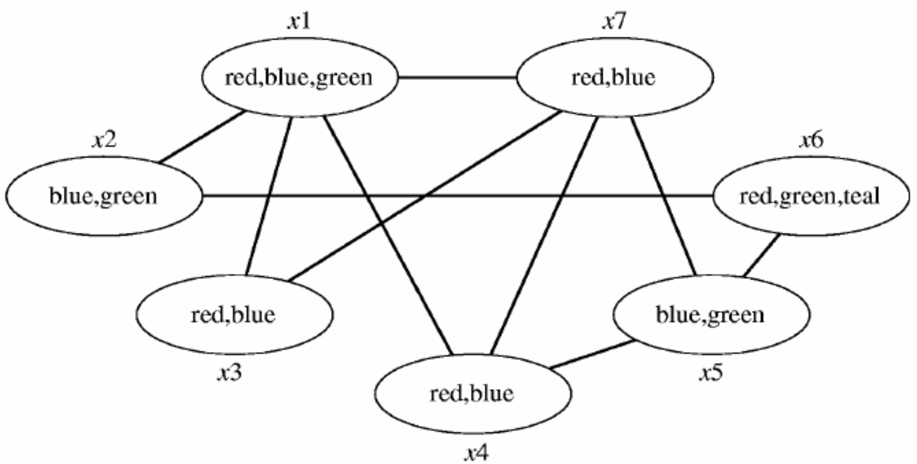
How to find a jump position? What is the source of the conflict?

- select the constraints containing just the currently instantiated variable and the past variables
- select the closest variable participating in the selected constraints

Graph-directed backjumping

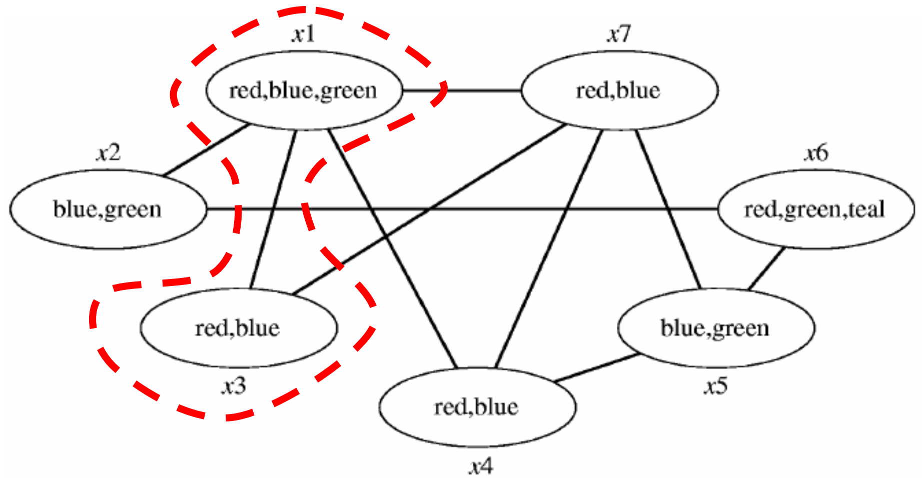


Assume the **graph colouring** problem, where the nodes are coloured in the order x1, x2, …, x7.



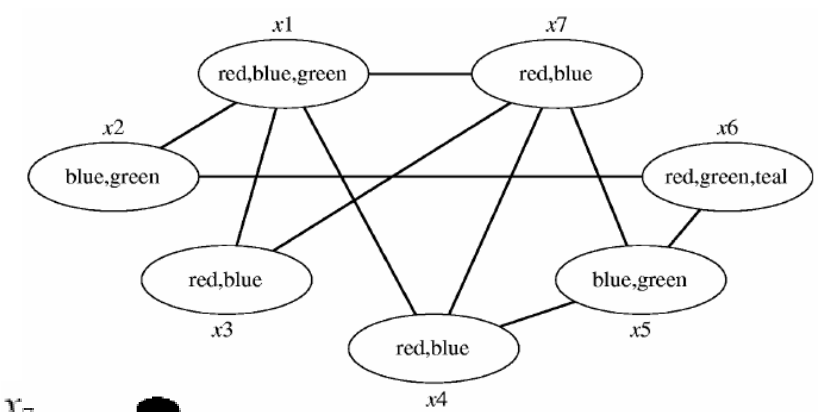
- Where to jump if colouring x4 fails?
  - ➢ to **x1**
- Where to jump if colouring x5 fails?
  - ➢ to **x4**
  - And what if x4 cannot be coloured?
  - ➢ to **x1**

It looks like after failure with some node, we can **jump to its closest predecessor** (in the colouring order), but …

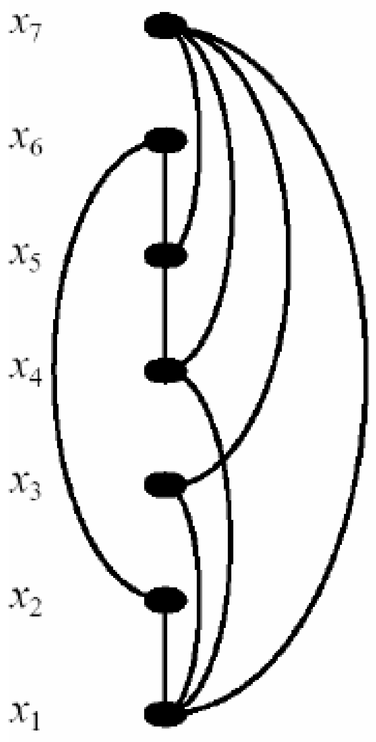


- Where to jump if colouring x7 fails?
- ➢ to **x5**
  - What if colouring x5 fails now?
  - ➢ to **x4**
  - And what if x4 cannot be coloured?
  - ➢ to **x3**

When jumping back, it is enough to free some **dead-end** (dead-end = a node/variable, that cannot be instantiated).



- from the leaf x jump to the closest predecessor of x in the constraint network (x7 → x5, x4, x3, x1)
- from the inner node x jump to the closest predecessor of all dead-end nodes visited during the jumps (x7 → x5, x4, x3, x1 → x4, x3, x1 → x3, x1)



– let **anc(x)** be the **ancestors of x** in the constraint network ordered using the labelling order (can be decided based on the network structure)
  - anc(x7) = {x5, x4, x3, x1}

– let us backjump to node x from the nodes y1,…,yk and let there be no more value for variable x

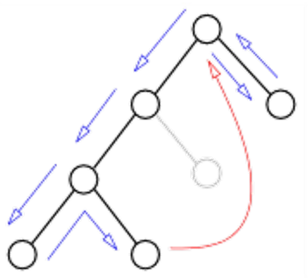
– then **jump to the closest variable from the set anc(x) ∪ anc(y1) ∪ … ∪ anc(yk) – {x,y1,…,yk}**

Graph-Directed Backjumping (a recursive version)

```
procedure GraphBJ(X:variables, V:assignment, C:constraints)
    if X = {} then return V
    x ← select a not-yet assigned variable from X
    conflict ← anc(x)
    for each value h from the domain of x do
        if constraints C are consistent with V ∪ {x/h} then
            R ← GraphBJ(X – {x}, V ∪ {x/h}, C)
            if R = fail(JumpSet) then                % backjump
                if x∉JumpSet then return R           % to a variable before x
                conflict ← conflict ∪ JumpSet – {x}  % to x
            else return R                            % solution found
    end for
    return fail(conflict)
end GraphBJ

Call as GraphBJ(X, {}, C)
```

Graph-Directed Backjumping (an iterative version)

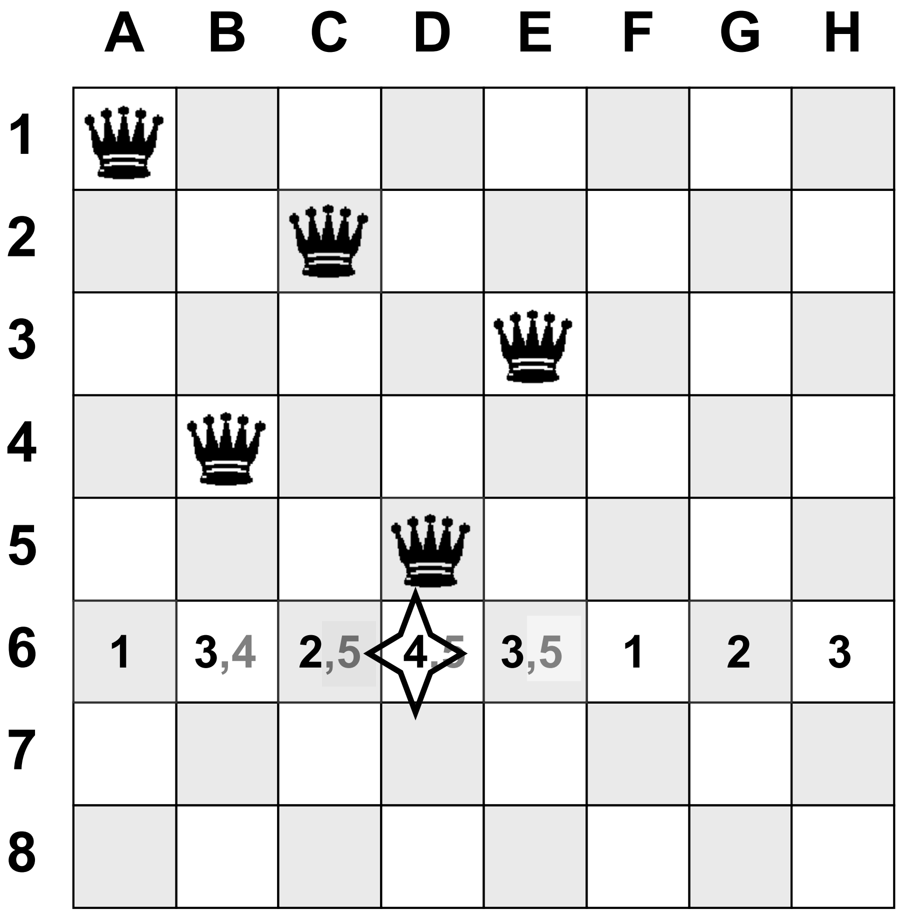
procedure GraphBJ(X:variables, C:constraints)
  $i \leftarrow 1$, $D'_i \leftarrow D_i$, $I_i \leftarrow anc(x_i)$
  while $1 \leq i \leq n$ **do**
    instantiate_and_check(i, C)
    if x_i = null **then**
      iprev $\leftarrow$ i, i $\leftarrow$ latest index in I_i, $I_i \leftarrow I_i \cup I_{iprev} - \{x_i\}$
    else
      $i \leftarrow i + 1$, $D'_i \leftarrow D_i$, $I_i \leftarrow anc(x_i)$
    end if
  end while
  if $i = 0$ **then** return fail
  return $\{x_1,\ldots, x_n\}$
end GraphBJ



procedure instantiate_and_check(i, C:constraints)
  while D'_i is not empty **do**
    select and delete some element b from D'_i
    $x_i \leftarrow b$
    if constraints C consistent with $\{x_1,\ldots, x_i\}$ **then** return
  end while
  $x_i \leftarrow$ null
end instantiate_and_check

Another view of Backjumping

N-queens problem



Queens in rows are allocated to columns.

6th queen cannot be allocated!

1. Write a number of conflicting queens to each position.

2. Select the farthest conflicting queen for each position.

3. Select the closest conflicting queen among positions.

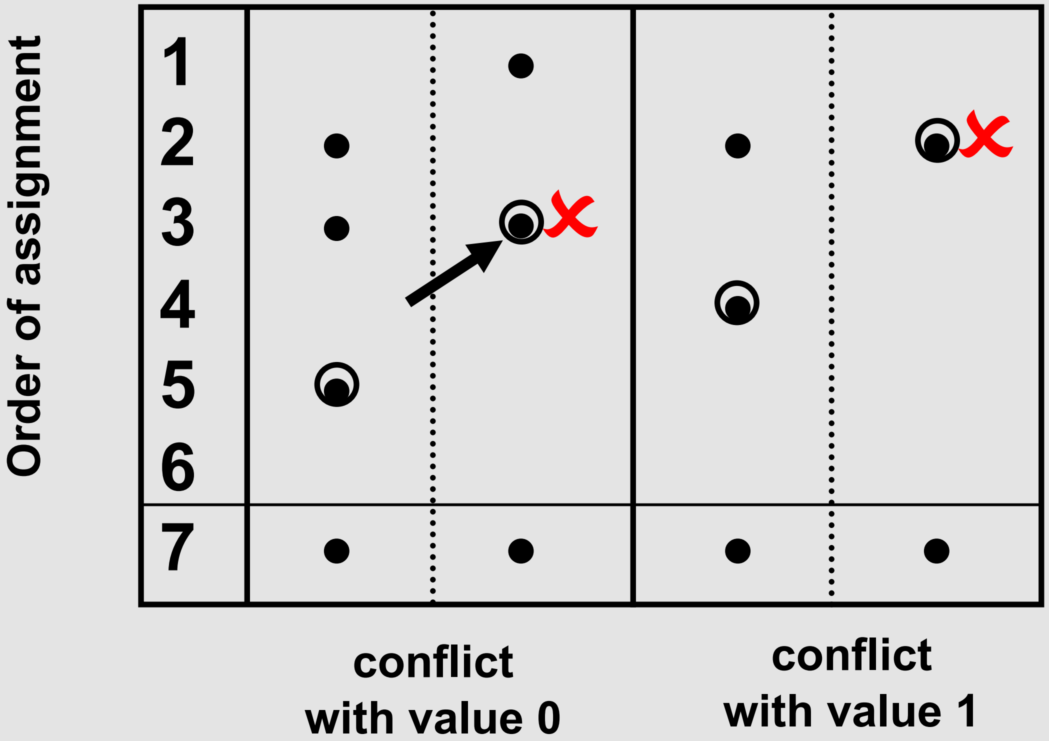
Note:

Graph-directed backjumping has no effect here (due to a complete graph)!

How to find out the conflicting variable?

Situation:

- assume that the variable no. 7 is being assigned (values are 0, 1)
- the symbol • marks the variables participating in the violated constraints (two constraints for each value)



Neither 0 nor 1 can be assigned to the seventh variable!

1. Find the closest variable in each violated constraint (o).

2. Select the farthest variable among the above chosen variables for each value (×).

3. Choose the closest variable among the conflicting variables selected for each value and jump to it.

Gaschnig Backjumping – consistency check

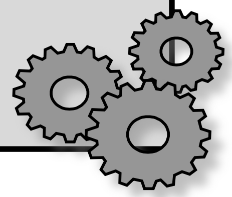
In addition to consistency check we can also find out the conflicting level!

```
procedure consistent(Labelled, Constraints, Level)
    J ← Level                          % the level to jump to
    NoConflict ← true                  % is there any conflict?
    for each C in Constraints do
      if all variables from C are Labelled then
          if C is not satisfied by Labelled then
               NoConflict ← false
               J ← min {J, max{L | X ∈ vars(C) & X/V/L in Labelled & L<Level}}
          end if
      end if
    end for
    if NoConflict then return true
                 else return fail(J)
end consistent
```

- **V is a value of X**
- **L is the depth level of X during labelling**

Gaschnig Backjumping (a recursive version)

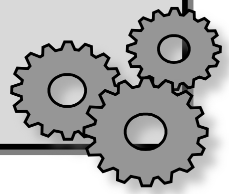
```
procedure GBJ(Unlabelled, Labelled, Constraints, PreviousLevel)
    if Unlabelled = {} then return Labelled
    pick first X from Unlabelled
    Level ← PreviousLevel+1
    Jump ← 0
    for each value V from D_X do
      C ← consistent({X/V/Level} ∪ Labelled, Constraints, Level)
      if C = fail(J) then
          Jump ← max {Jump, J}
      else
          Jump ← PreviousLevel
          R ← GBJ(Unlabelled-{X},{X/V/Level} ∪ Labelled,Constraints, Level)
          if R ≠ fail(Level) then return R          % success or jump further
      end if
    end for
    return fail(Jump)                               % jump to the conflicting variable
end GBJ

Call as GBJ(Variables,{},Constraints,0)
```



Gaschnig Backjumping (an iterative version)

procedure GBJ(X:variables, C:constraints)
    $i \leftarrow 1$, $D'_i \leftarrow D_i$, $jump_i \leftarrow 0$
    while $1 \leq i \leq n$ **do**
        $x_i \leftarrow$ select_value(i, C)
        if x_i = null **then**
            $i \leftarrow jump_i$
        else
            $i \leftarrow i + 1$
            $D'_i \leftarrow D_i$
            $jump_i \leftarrow 0$
        end if
    end while
    if $i = 0$ **then** return fail
    return $\{x_1,\ldots, x_n\}$
end GBJ

procedure select_value(i, C:constraints)
    while D'_i is not empty **do**
        select and delete some element b from D'_i
        consistent $\leftarrow$ true
        $k \leftarrow 1$
        while $k<i$ and consistent **do**
            if $k>jump_i$ **then** $jump_i \leftarrow k$
            if x_i=b consistent with $\{x_1,\ldots, x_k\}$ in C **then**
                    $k \leftarrow k + 1$
            else   consistent $\leftarrow$ false
        end while
        if consistent **then** return b
    end while
    return null
end select_value



Backjumping – a short summary

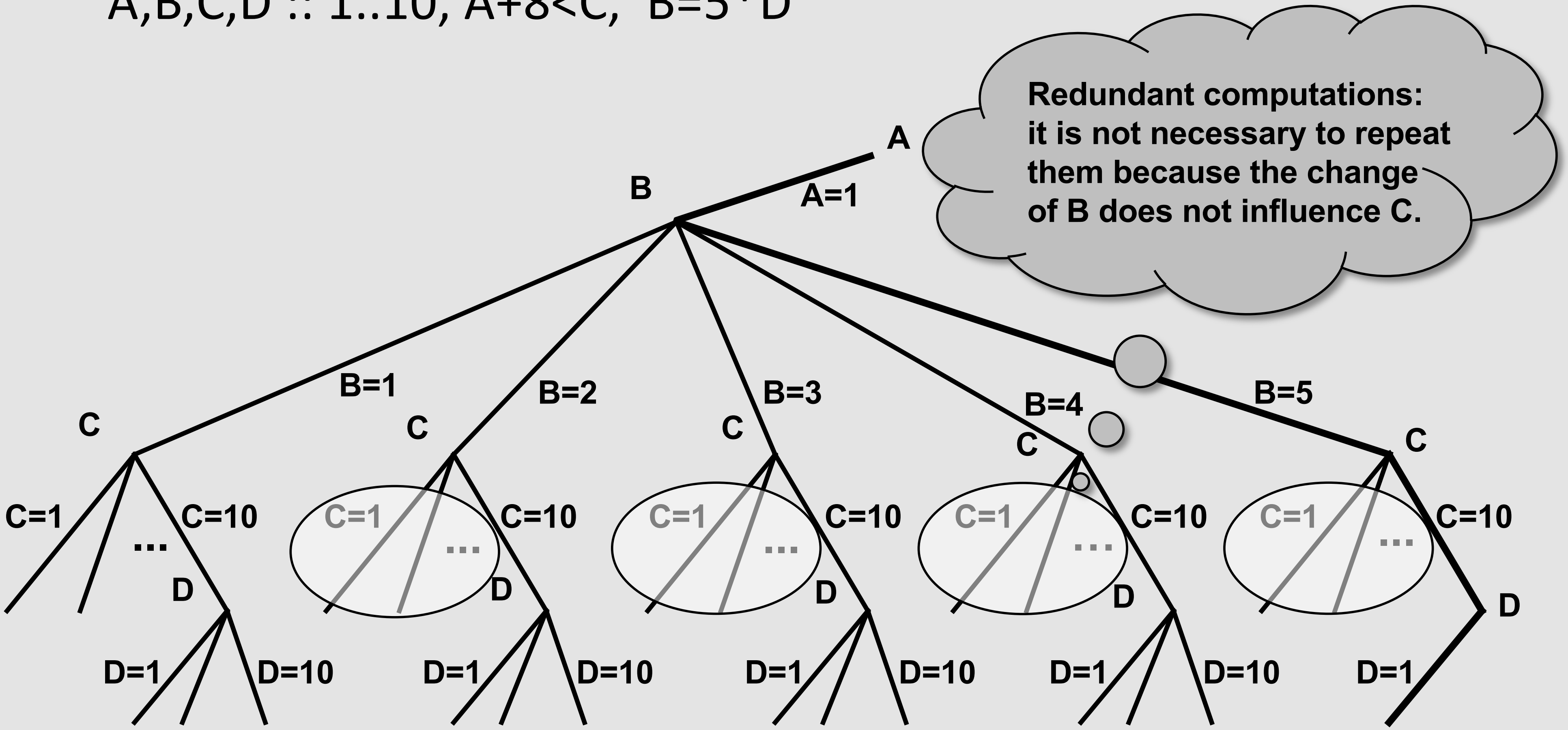
- **Graph-directed Backjumping**
  - driven by the structure of constraint network only (does not assume (dis)satisfaction of constraints)
  - can do several jumps in a sequence
- **Gaschnig Backjumping**
  - assumes which constraints are violated
  - just one back-jump (if the assignment fails again, only one level up is backtracked like in chronological backtracking)
- **Conflict-driven Backjumping (CBJ)**
  - we can join advantages of both methods (better target to jump to and more back-jumps in a sequence)
  - when jumping back we need to keep a **conflict set** of variables that is used for the next jump if no value is found for the current variable
    - we carry the source of conflict when backjumping

What is a redundant work?

– repeated computation whose result has already been obtained

Example:

A,B,C,D :: 1..10, A+8<C, B=5*D



Haralick, Elliot (1980)

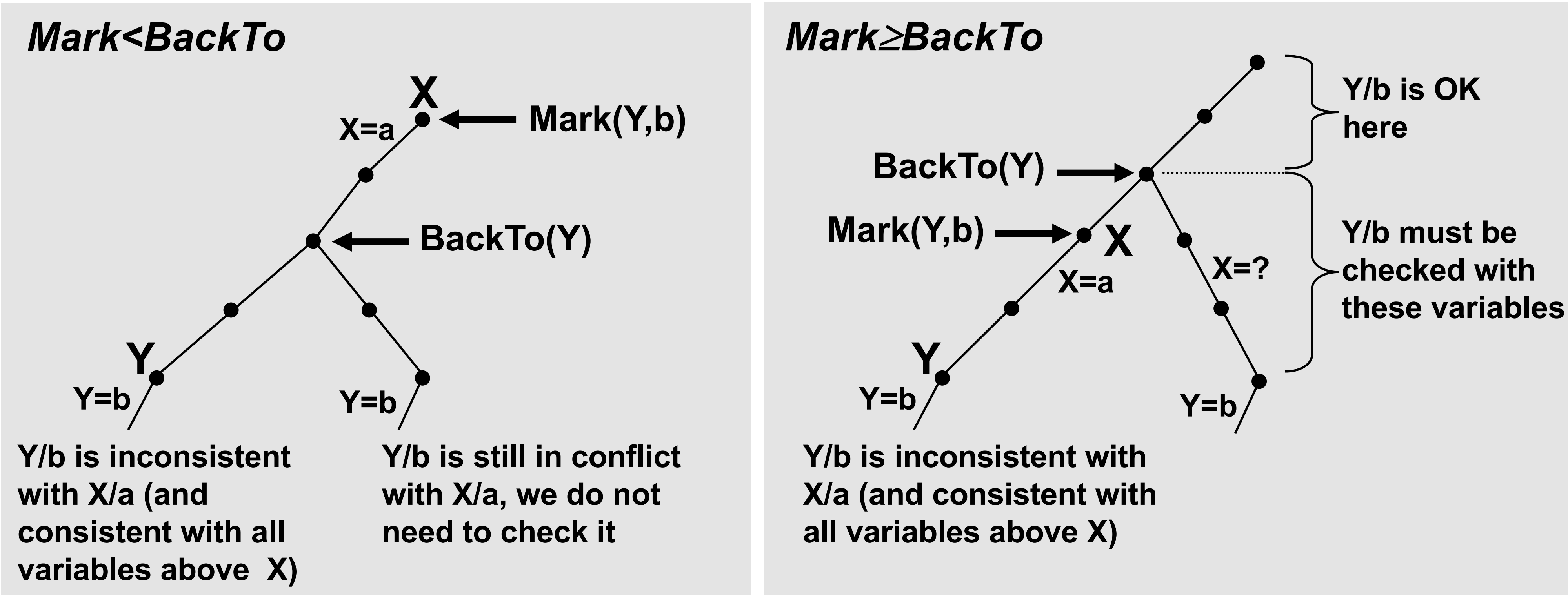
The principles of Backmarking

Remove redundant constraint checks by **memorising negative and positive results of tests**:

– **Mark(X,V)** is the farthest (instantiated) variable in conflict with the assignment X=V

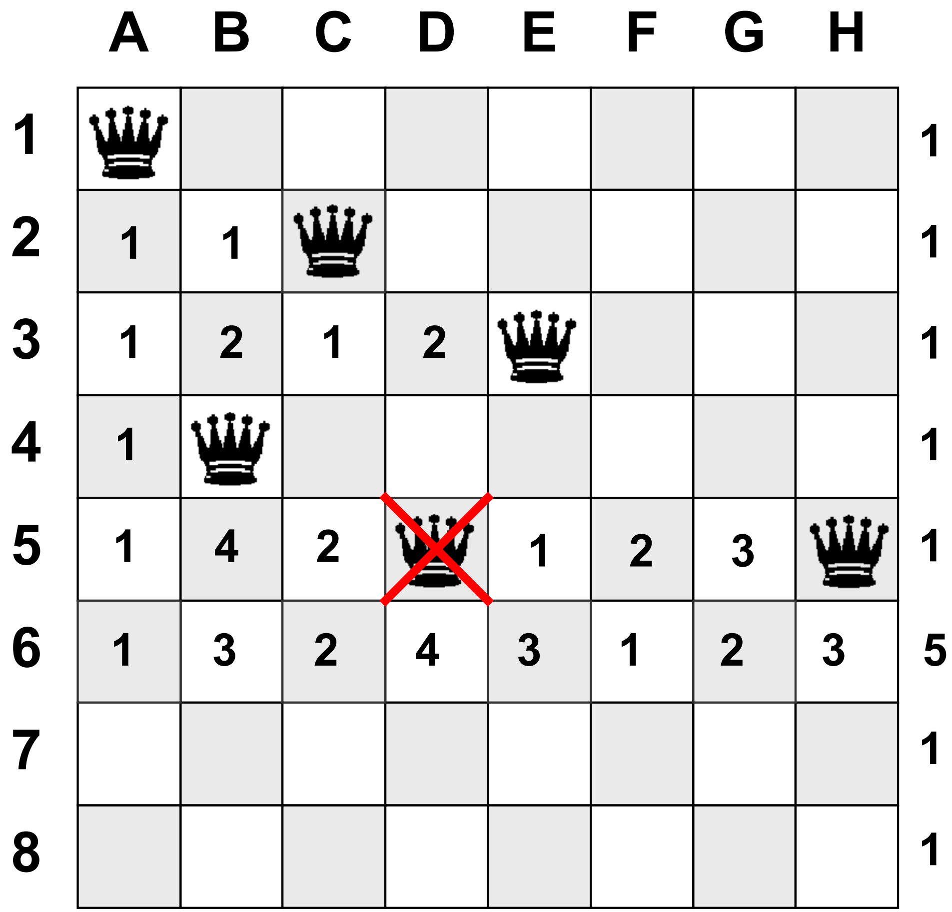
– **BackTo(X)** is the farthest variable to which we backtracked since the last attempt to instantiate X

- Now, some constraint checks can be omitted:



Backmarking in an example

N-queens problem



1. Queens in rows are allocated to columns.

2. Latest choice level is written next to chessboard (BackTo). At beginning 1s.

3. Farthest conflict queen at each position (Mark). At beginning 1s.

4. 6th queen cannot be allocated!

5. Backtrack to 5, change BackTo.

6. When allocating 6th queen, all the positions are still wrong (MarkTo<BackTo).

Note:

backmarking can be combined with backjumping (for free)

Only the constraints where any value is changed are re-checked,
and the farthest conflicting level is computed.

procedure consistent(X/V, Labelled, Constraints, Level)

  for each Y/VY/LY in Labelled such that LY≥BackTo(X) **do**

   % only possible changed variables Y are explored

   % in the increasing order of LY (first the oldest one)

   if X/V is not compatible with Y/VY using Constraints **then**

     Mark(X,V) ← LY

     return fail

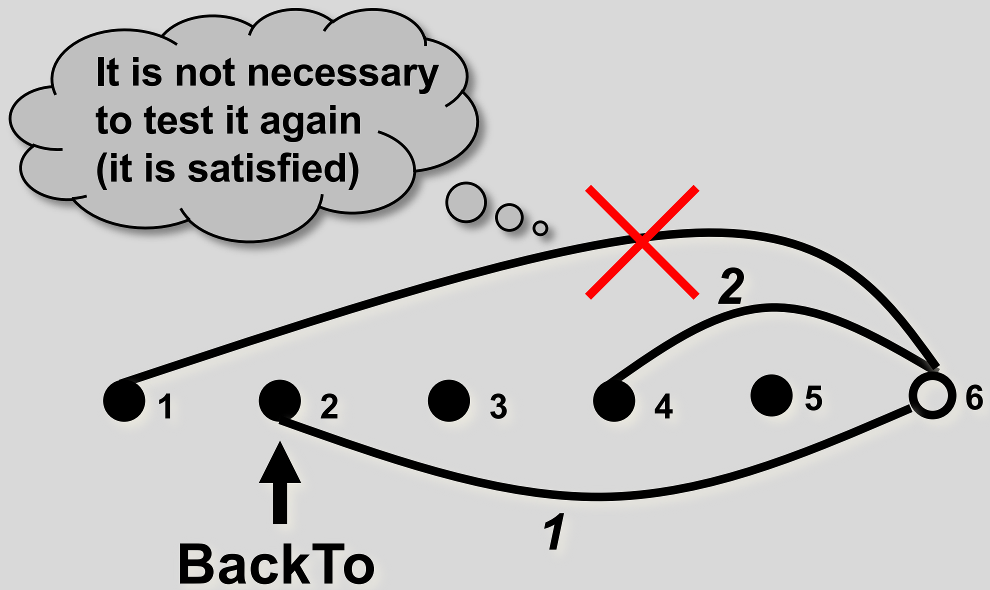
   end if

  end for

  Mark(X,V) ← Level-1

  return true

end consistent



Algorithm Backmarking

```
procedure BM(Unlabelled, Labelled, Constraints, Level)
  if Unlabelled = {} then return Labelled
  pick first X from Unlabelled          % fix order of variables
  for each value V from D_X do
     if Mark(X,V) ≥ BackTo(X) then       % re-check the value
        if consistent(X/V, Labelled, Constraints, Level) then
           R ← BM(Unlabelled-{X}, Labelled ∪ {X/V/Level}, Constraints, Level+1)
           if R ≠ fail then return R       % solution found
        end if
     end if
  end for
  BackTo(X) ←  Level-1                    % jump will be to the previous variable
  for each Y in Unlabelled do             % tell everyone about the jump
     BackTo(Y) ←  min {Level-1, BackTo(Y)}
  end for
  return fail                             % backtrack to the previous variable
end BM
```

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