

Constraint Programming

Roman Barták

Department of Theoretical Computer Science and Mathematical Logic

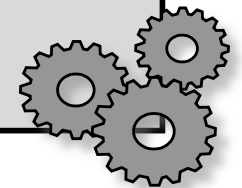
Look-back Search Algorithms

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

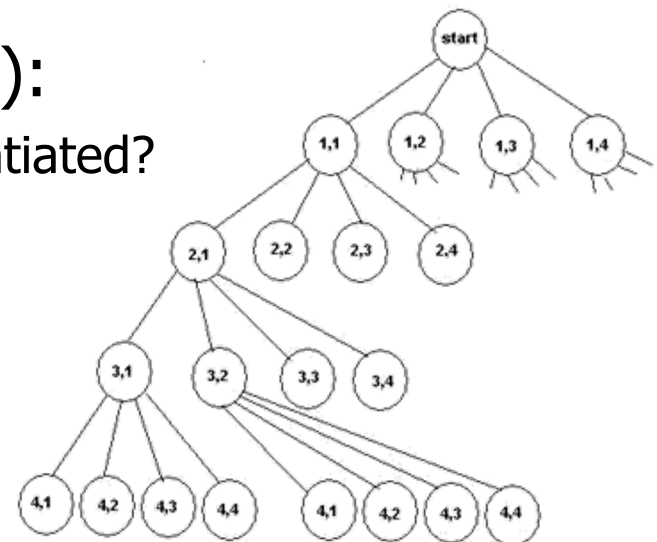
The core principle of applying backtracking to solve a CSP:

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):

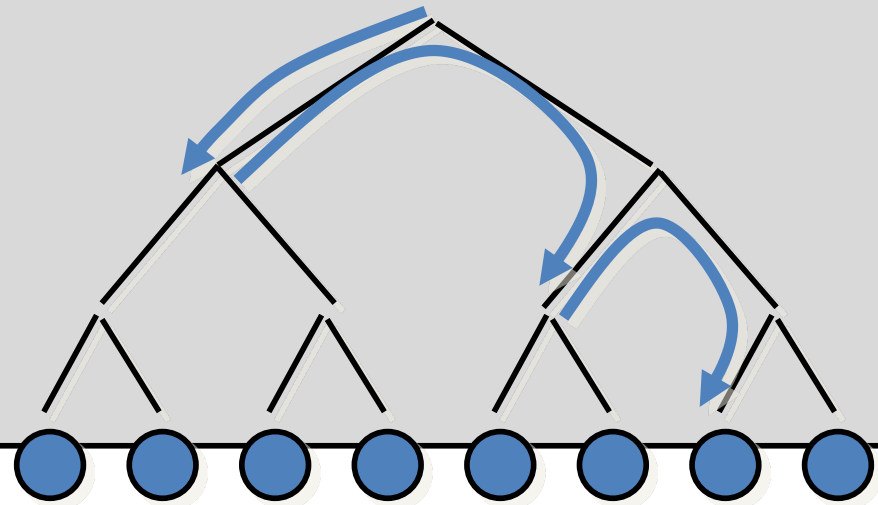
- What is the order of variables being instantiated?
- What is the order of values tried?

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  $V \cup \{x/h\}$  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)
```

```
   $i \leftarrow 1, D'_i \leftarrow D_i$ 
```

```
  while  $1 \leq i \leq n$  do
```

```
    instantiate_and_check(i, C)
```

```
    if  $x_i = \text{null}$  then  $i \leftarrow i - 1$  else  $i \leftarrow i + 1, D'_i \leftarrow D_i$ 
```

```
  end while
```

```
  if  $i = 0$  then return fail
```

```
  return  $\{x_1, \dots, x_n\}$ 
```

```
end BT
```

```
procedure instantiate_and_check(i, C:constraints)
```

```
  while  $D'_i$  is not empty do
```

```
    select and delete some element  $b$  from  $D'_i$ 
```

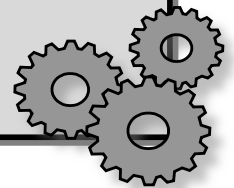
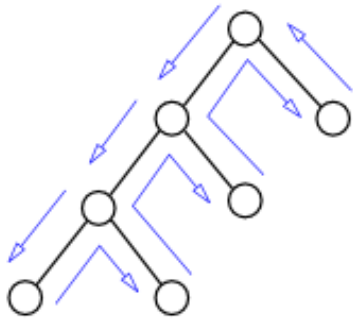
```
     $x_i \leftarrow b$ 
```

```
    if constraints  $C$  are consistent with  $\{x_1, \dots, x_i\}$  then return
```

```
  end while
```

```
   $x_i \leftarrow \text{null}$ 
```

```
end instantiate_and_check
```



Look Back

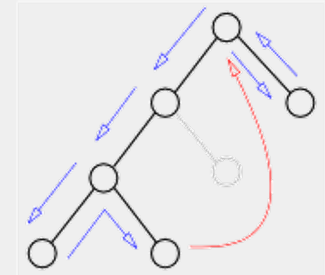
- **thrashing**

- throws away the reason of the conflict

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

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

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

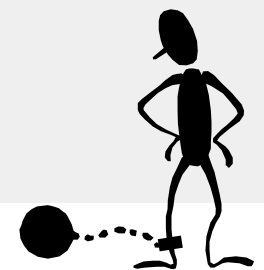
- **late detection of the conflict**

- 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?

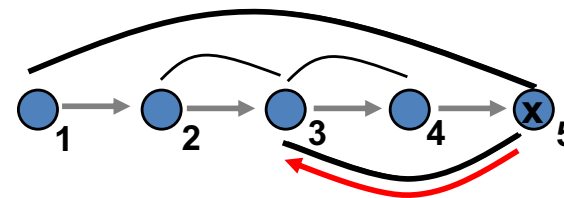
- 1) identify the source of the conflict (impossibility to assign a value)
- 2) jump to the past variable in conflict

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

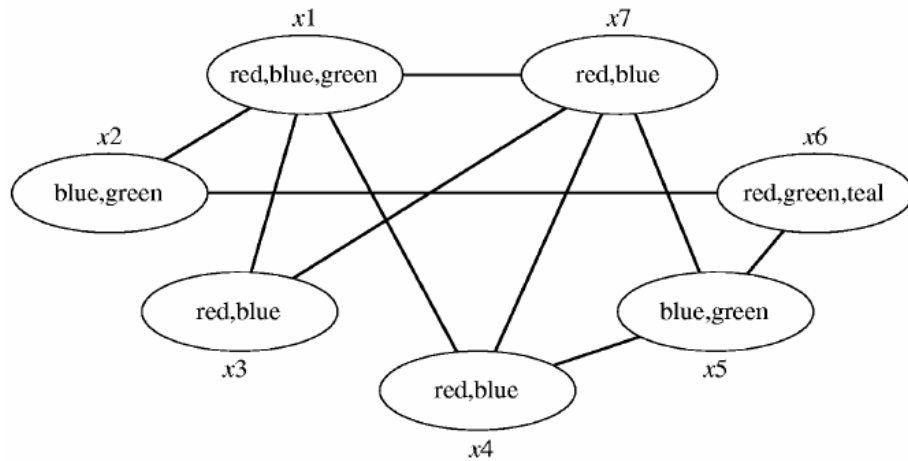
How to find a jump position? What is the source of the conflict?

- 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

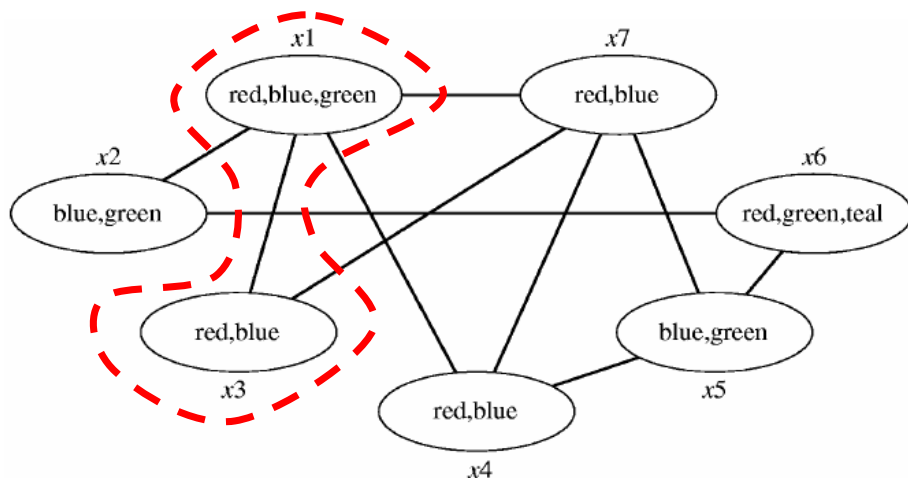


Assume the **graph colouring** problem, where the nodes are coloured in the order x_1, x_2, \dots, x_7 .



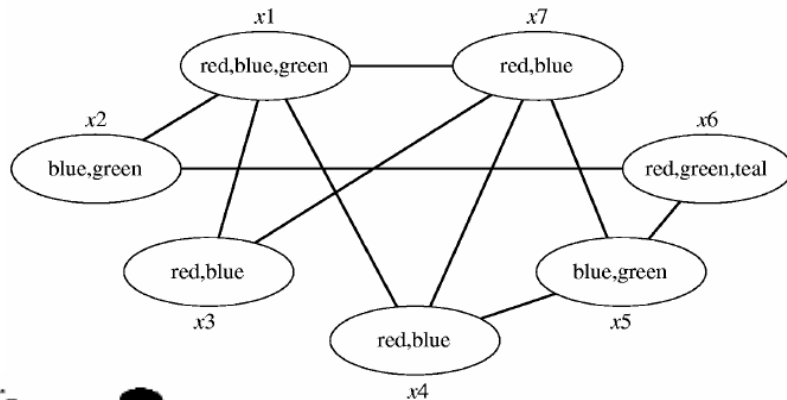
- Where to jump if colouring x_4 fails?
 - to **x_1**
- Where to jump if colouring x_5 fails?
 - to **x_4**
 - And what if x_4 cannot be coloured?
 - to **x_1**

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

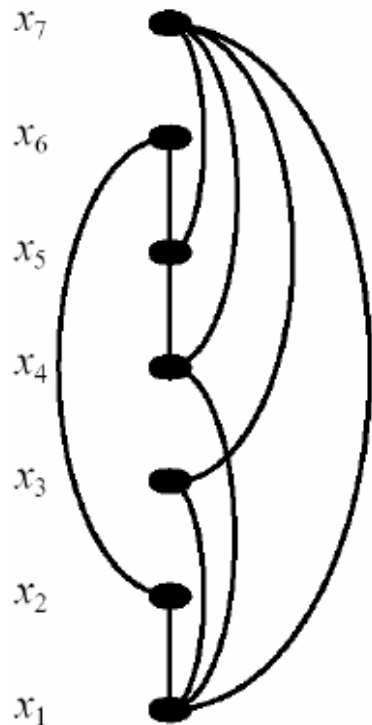


- Where to jump if colouring x_7 fails?
 - to **x_5**
 - What if colouring x_5 fails now?
 - to **x_4**
 - And what if x_4 cannot be coloured?
 - to **x_3**

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 ($x7 \rightarrow x5, x4, x3, x1$)
- from the inner node x jump to the closest predecessor of all dead-end nodes visited during the jumps ($x7 \rightarrow x5, x4, x3, x1 \rightarrow x4, x3, x1 \rightarrow x3, x1$)

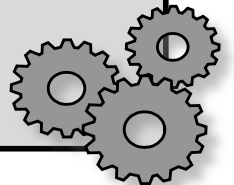


- let **anc(x)** be the **ancestors of x** in the constraint network ordered using the labelling order (can be decided based on the network structure)
 - $\text{anc}(x7) = \{x5, x4, x3, x1\}$
- let us backjump to node x from the nodes $y1, \dots, yk$ and let there be no more value for variable x
- then **jump to the closest variable from the set $\text{anc}(x) \cup \text{anc}(y1) \cup \dots \cup \text{anc}(yk) - \{x, y1, \dots, yk\}$**

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)
```

```
   $i \leftarrow 1$ ,  $D'_i \leftarrow D_i$ ,  $l_i \leftarrow \text{anc}(x_i)$ 
```

```
  while  $1 \leq i \leq n$  do
```

```
    instantiate_and_check(i, C)
```

```
    if  $x_i = \text{null}$  then
```

```
       $i_{\text{prev}} \leftarrow i$ ,  $i \leftarrow \text{latest index in } l_i$ ,  $l_i \leftarrow l_i \cup l_{i_{\text{prev}}} - \{x_i\}$ 
```

```
    else
```

```
       $i \leftarrow i + 1$ ,  $D'_i \leftarrow D_i$ ,  $l_i \leftarrow \text{anc}(x_i)$ 
```

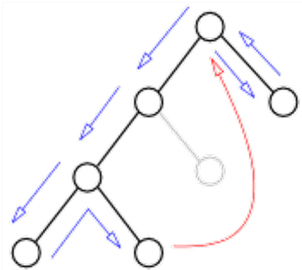
```
    end if
```

```
  end while
```

```
  if  $i = 0$  then return fail
```

```
  return  $\{x_1, \dots, x_n\}$ 
```

```
end GraphBJ
```



```
procedure instantiate_and_check(i, C:constraints)
```

```
  while  $D'_i$  is not empty do
```

```
    select and delete some element  $b$  from  $D'_i$ 
```

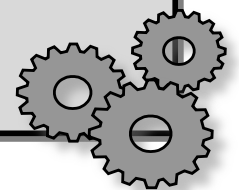
```
     $x_i \leftarrow b$ 
```

```
    if constraints  $C$  consistent with  $\{x_1, \dots, x_i\}$  then return
```







```
  end while
```

```
   $x_i \leftarrow \text{null}$ 
```

```
end instantiate_and_check
```



N-queens problem

	A	B	C	D	E	F	G	H
1								
2								
3								
4								
5								
6	1	3,4	2,5		3,5	1	2	3
7								
8								

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:

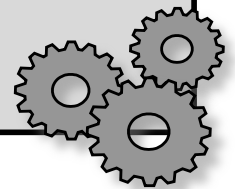
Graph-directed backjumping has no effect here (due to a complete graph)!

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
```

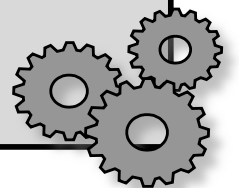
- V is a value of X
- L is the depth level of X during labelling



Gaschnig Backjumping (a recursive version)

```
procedure GBJ(Unlabelled, Labelled, Constraints, PreviousLevel)
  if Unlabelled = {} then return Labelled
  pick first X from Unlabelled
  Level  $\leftarrow$  PreviousLevel+1
  Jump  $\leftarrow$  0
  for each value V from  $D_x$  do
    C  $\leftarrow$  consistent({X/V/Level}  $\cup$  Labelled, Constraints, Level)
    if C = fail(J) then
      Jump  $\leftarrow$  max {Jump, J}
    else
      Jump  $\leftarrow$  PreviousLevel
      R  $\leftarrow$  GBJ(Unlabelled- $\{X\}$ , {X/V/Level}  $\cup$  Labelled, Constraints, Level)
      if R  $\neq$  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)
```

```
   $i \leftarrow 1$ ,  $D'_i \leftarrow D_i$ ,  $\text{jump}_i \leftarrow 0$ 
```

```
  while  $1 \leq i \leq n$  do
```

```
     $x_i \leftarrow \text{select\_value}(i, C)$ 
```

```
    if  $x_i = \text{null}$  then
```

```
       $i \leftarrow \text{jump}_i$ 
```

```
    else
```

```
       $i \leftarrow i + 1$ 
```

```
       $D'_i \leftarrow D_i$ 
```

```
       $\text{jump}_i \leftarrow 0$ 
```

```
    end if
```

```
  end while
```

```
  if  $i = 0$  then return fail
```

```
  return  $\{x_1, \dots, x_n\}$ 
```

```
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  $k > \text{jump}_i$  then  $\text{jump}_i \leftarrow k$ 
```

```
      if  $x_i = b$  consistent with  $\{x_1, \dots, x_k\}$  in  $C$  then
```

```
         $k \leftarrow k + 1$ 
```

```
      else consistent  $\leftarrow$  false
```

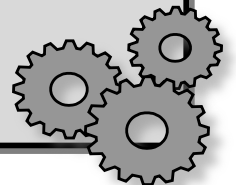
```
    end while
```

```
    if consistent then return  $b$ 
```

```
  end while
```

```
  return null
```

```
end select_value
```

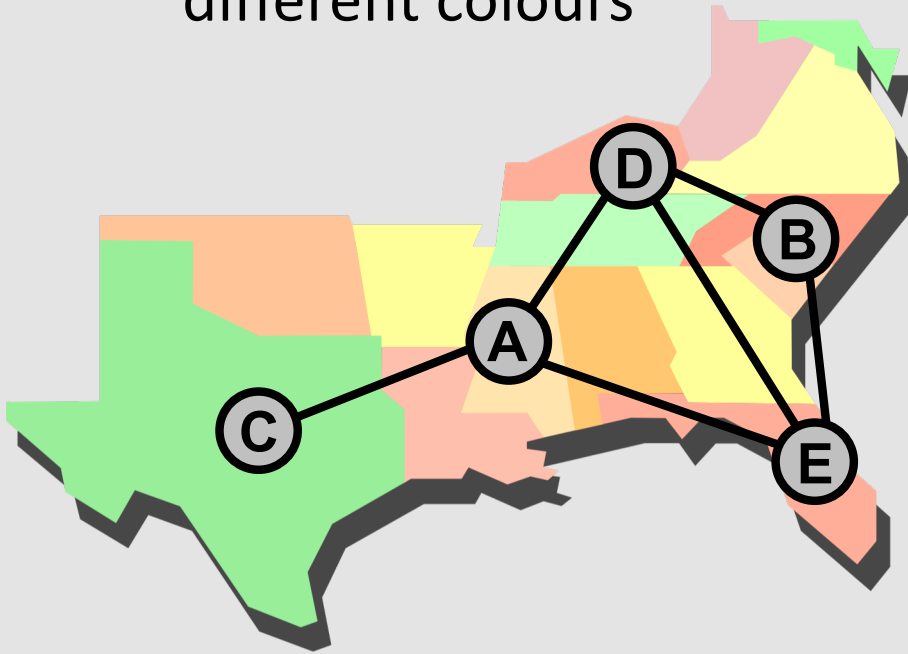


- **Graph-directed Backjumping**
 - driven by the structure of constraint network only (does not assume (dis)satisfaction of constraints)
 - can do several jumps in a sequence
- **Gaschnig Backjumping**
 - assumes which constraints are violated
 - just one back-jump (if the assignment fails again, only 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

When jumping back the in-between assignment is lost!

Example:

- colour the graph in such a way that the connected vertices have different colours



node	vertex
A	1
B	2
C	1 2
D	1 2 3
E	1 2 3

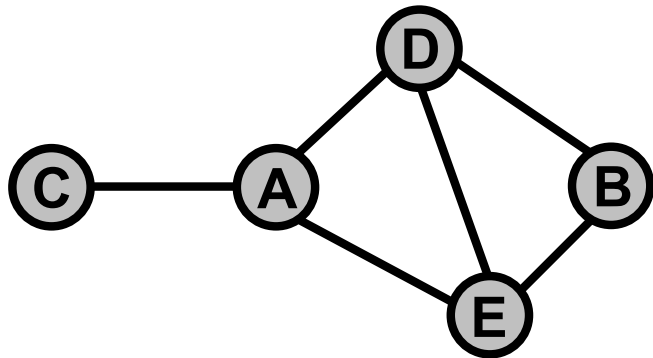
A	1
B	2
C	1 2
D	1 2
E	1 2 3

backjump

During the second attempt to label C superfluous work is done - it is enough to leave there the original value 2, the change of B does not influence C.

Dynamic Backtracking in an example

The same graph (A,B,C,D,E), the same colours (1,2,3) but a different approach.



Backjumping

- + remember the source of the conflict
- + carry the source of the conflict
- + change the order of variables

= DYNAMIC BACKTRACKING

node	1	2	3
A	•		
B		•	
C	A	•	
D	A	B	•
E	A	B	D

jump back
+ carry the conflict source

node	1	2	3
A	•		
B		•	
C	A	•	
D	A	B	AB
E	A	B	

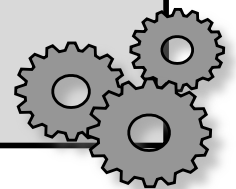
jump back
+ carry the conflict source
+ change the order of B, C

node	1	2	3
A	•		
C	A	•	
B	•	A	
D	A	•	
E	A	D	•

• selected colour
AB a source of the conflict

The vertex C (and the possible sub-graph connected to C) is not re-coloured.

```
procedure DB(Variables, Constraints)
  Labelled  $\leftarrow$  {}; Unlabelled  $\leftarrow$  Variables
  while Unlabelled  $\neq$  {} do
    select X in Unlabelled
    ValuesX  $\leftarrow$  DX - {values inconsistent with Labelled using Constraints}
    if ValuesX = {} then
      let E be an explanation of the conflict (set of conflicting variables)
      if E = {} then failure
      else
        let Y be the most recent variable in E
        unassign Y (from Labelled) with eliminating explanation E- $\{Y\}$ 
        remove all the explanations involving Y
      end if
    else
      select V in ValuesX
      Unlabelled  $\leftarrow$  Unlabelled - {X}
      Labelled  $\leftarrow$  Labelled  $\cup$  {X/V}
    end if
  end while
  return Labelled
end DB
```

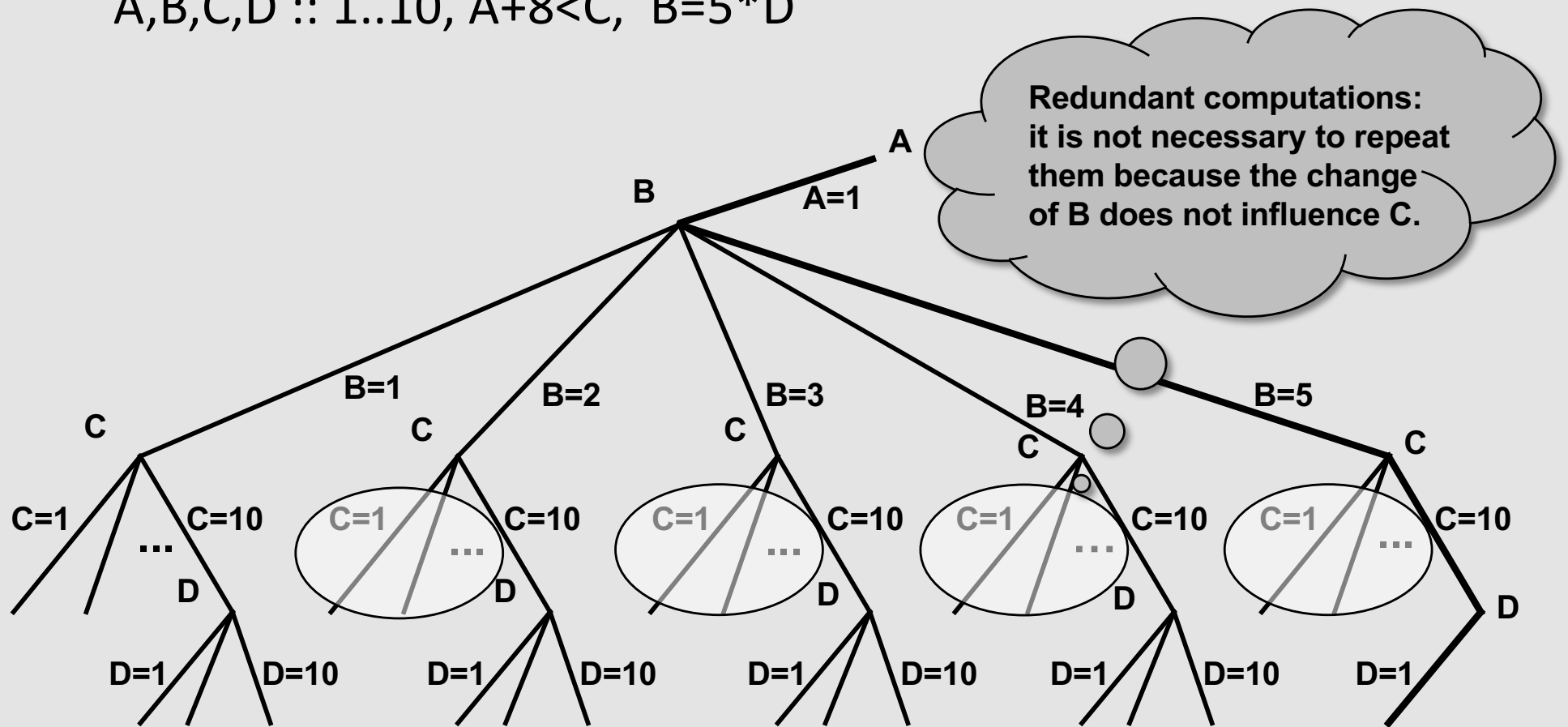


What is a redundant work?

- repeated computation whose result has already been obtained

Example:

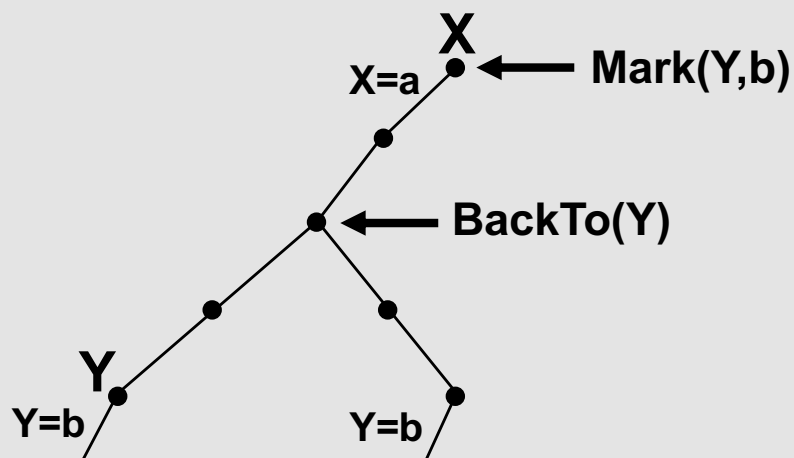
$A, B, C, D :: 1..10, A+8 < C, B=5 * D$



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:

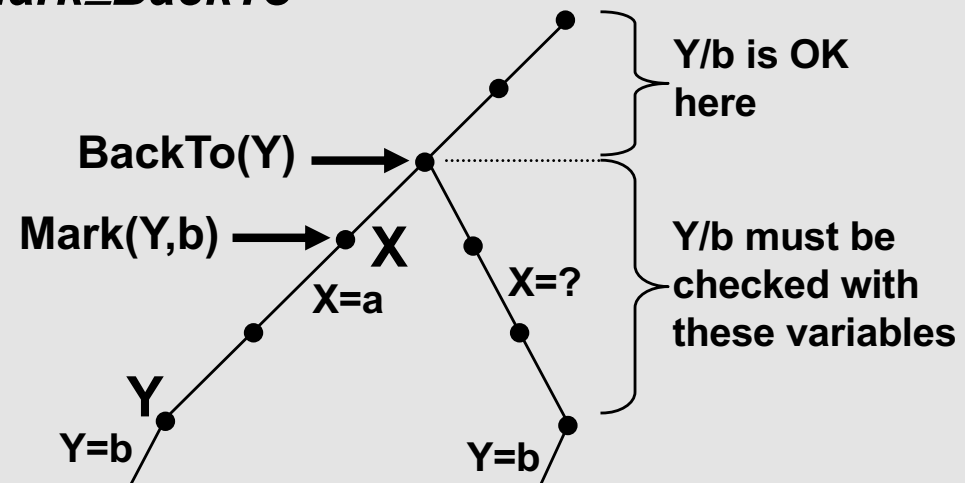
Mark < BackTo



Y/b is inconsistent with X/a (and consistent with all variables above X)

Y/b is still in conflict with X/a , we do not need to check it

Mark ≥ BackTo



Y/b is inconsistent with X/a (and consistent with all variables above X)

N-queens problem

	A	B	C	D	E	F	G	H	
1	♙								1
2	1	1	♙						1
3	1	2	1	2	♙				1
4	1	♙							1
5	1	4	2	♙	1	2	3	♙	1
6	1	3	2	4	3	1	2	3	5
7									1
8									1

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 (MarkTo). 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).

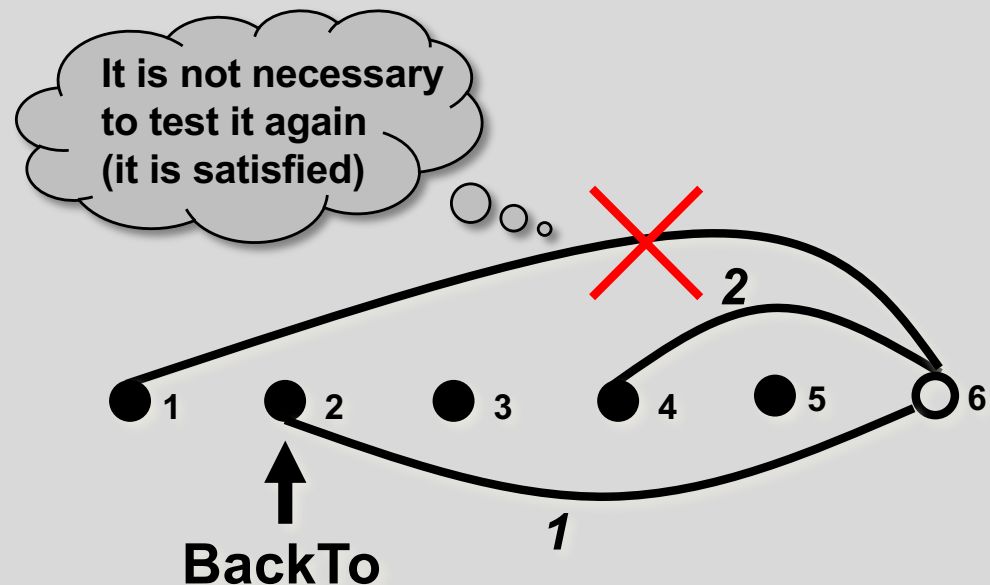
Note:

backmarking can be combined with backjumping (for free)

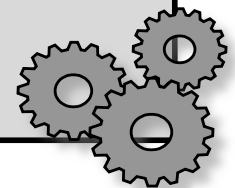
Consistency check for Backmarking

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  $LY \geq \text{BackTo}(X)$  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)  $\leftarrow$  LY
      return fail
    end if
  end for
  Mark(X,V)  $\leftarrow$  Level-1
  return true
end consistent
```




```
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)  $\geq$  BackTo(X) then       % re-check the value
      if consistent(X/V, Labelled, Constraints, Level) then
        R  $\leftarrow$  BM(Unlabelled- $\{X\}$ , Labelled  $\cup$  {X/V/Level}, Constraints, Level+1)
        if R  $\neq$  fail then return R       % solution found
      end if
    end if
  end for
  BackTo(X)  $\leftarrow$  Level-1                % jump will be to the previous variable
  for each Y in Unlabelled do           % tell everyone about the jump
    BackTo(Y)  $\leftarrow$  min {Level-1, BackTo(Y)}
  end for
  return fail                             % backtrack to the previous variable
end BM
```





© 2013 Roman Barták

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