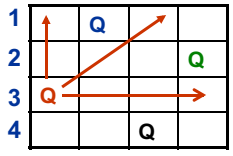
Solving Constraint Programs using Conflicts and Backjumping

Slides draw upon material from:
Prof. Patrick Prosser, Glasow University

Brian C. Williams
16.410-13
September 29th, 2010

Search Performance on N Queens



- **Standard Search**
- **Backtracking**
- **BT with Forward Checking**
- **Dynamic Variable Ordering**
- **Iterative Repair**
- **Conflict-directed Back Jumping**

- A handful of queens
- About 15 queens
- About 30 queens
- About 1,000 queens
- About 10,000,000 queens (except truly hard problems)

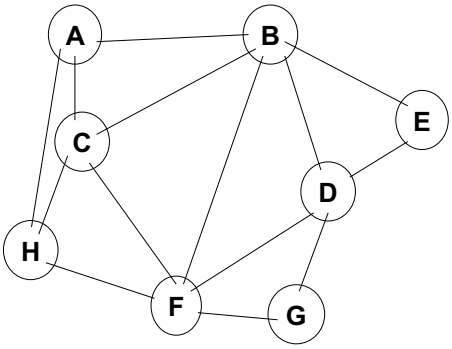
Back Jumping

Backtracking At dead end, backup to the most recent variable.

Backjumping At dead end, backup to the most recent variable that eliminated some value in the domain of the dead end variable.

3

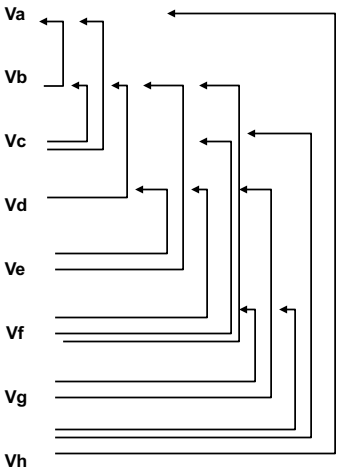
Example of a CSP



1 = red
2 = blue
3 = green

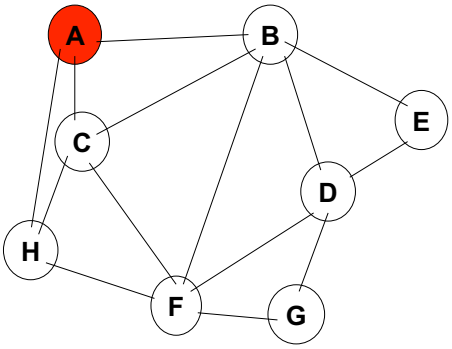
Variables and Instantiation Order

Checking back



Slide progression due to Prosser [4C presentation, 2003]

Example of a CSP

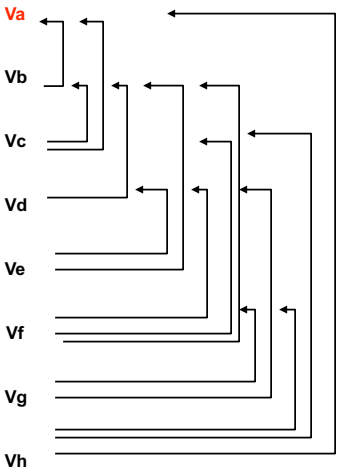


1 = red
2 = blue
3 = green

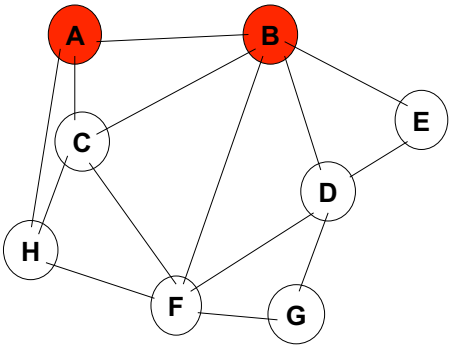
Find solution using Backtracking

Variables and Instantiation Order

Checking back



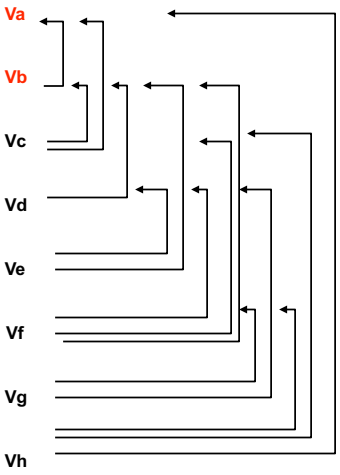
Example of a CSP



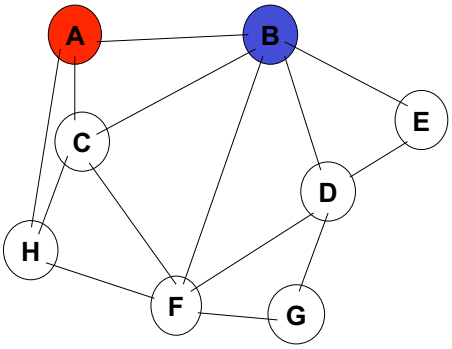
1 = red
2 = blue
3 = green

Variables and Instantiation Order

Checking back



Example of a CSP



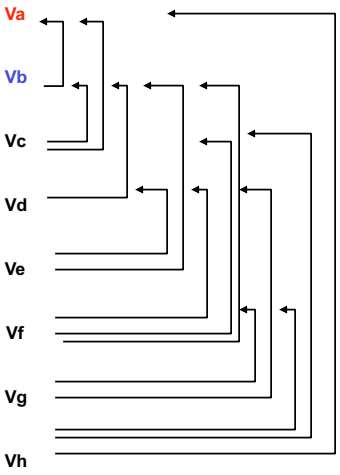
1 = red

2 = blue

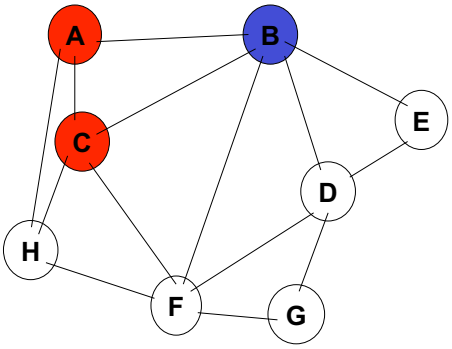
3 = green

Variables and Instantiation Order

Checking back



Example of a CSP



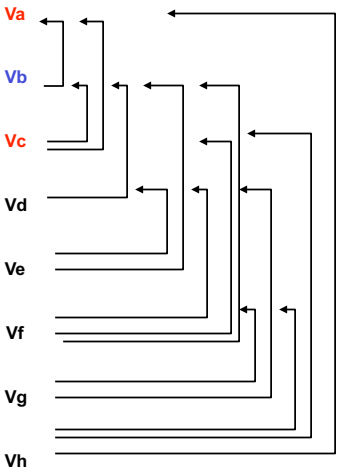
1 = red

2 = blue

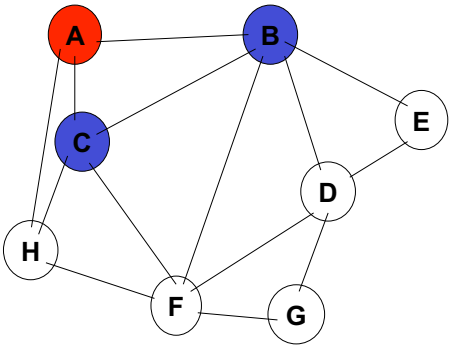
3 = green

Variables and Instantiation Order

Checking back



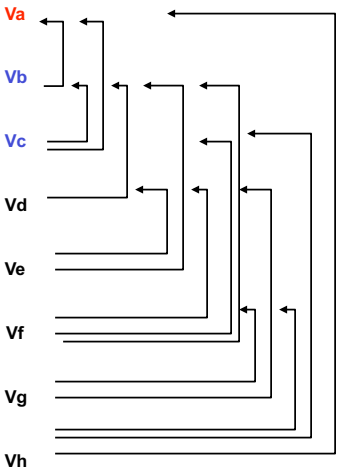
Example of a CSP



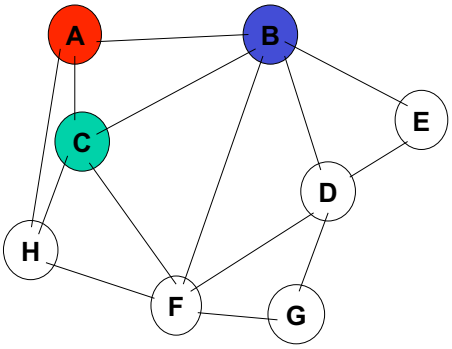
1 = red
2 = blue
3 = green

Variables and Instantiation Order

Checking back



Example of a CSP

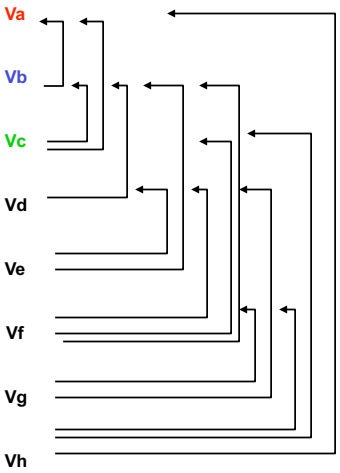


1 = red
2 = blue
3 = green

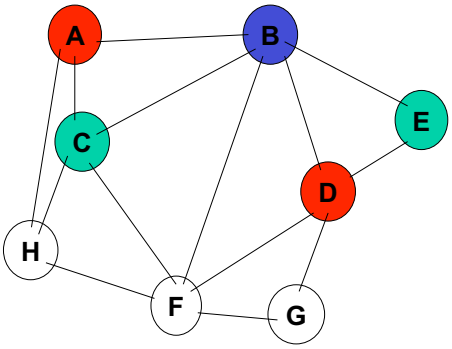
Fast forward .....

Variables and Instantiation Order

Checking back



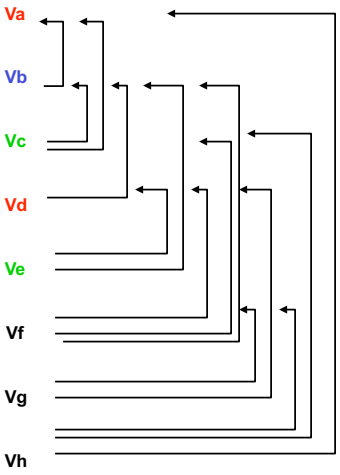
Example of a CSP



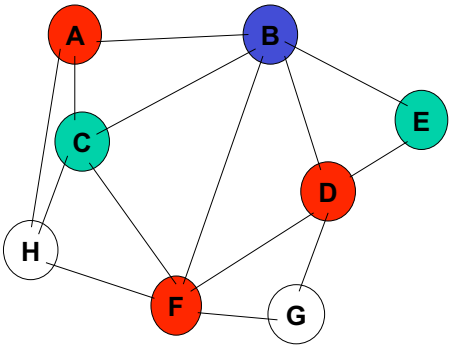
1 = red
2 = blue
3 = green

Variables and Instantiation Order

Checking back



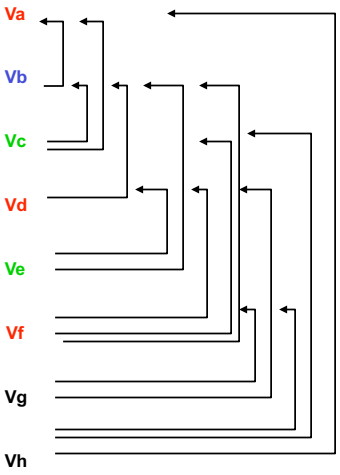
Example of a CSP



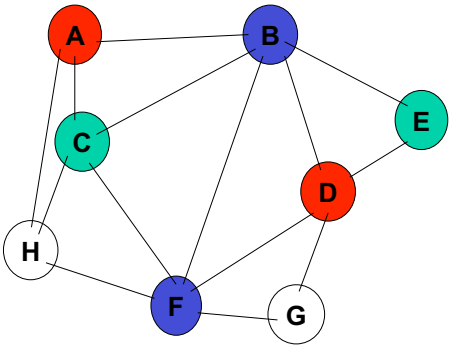
1 = red
2 = blue
3 = green

Variables and Instantiation Order

Checking back



Example of a CSP



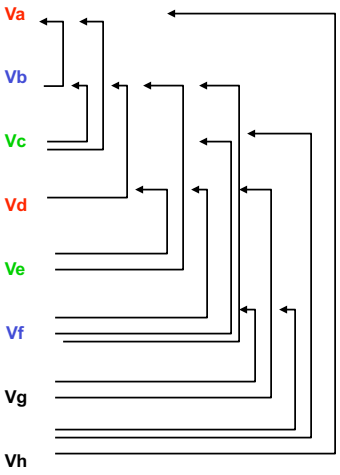
1 = red

2 = blue

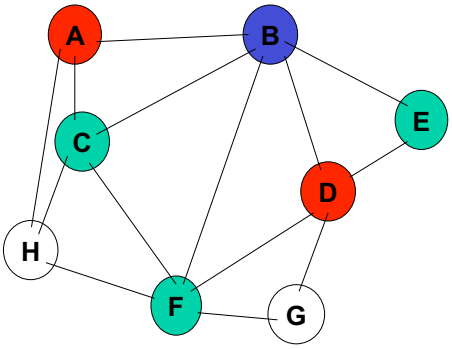
3 = green

Variables and Instantiation Order

Checking back



Example of a CSP



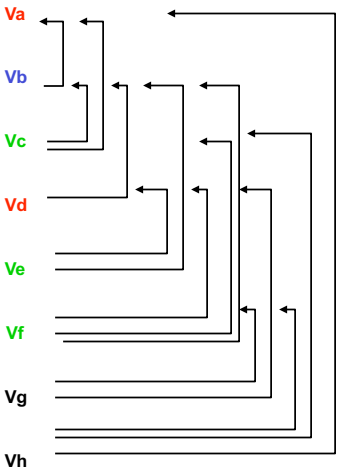
1 = red

2 = blue

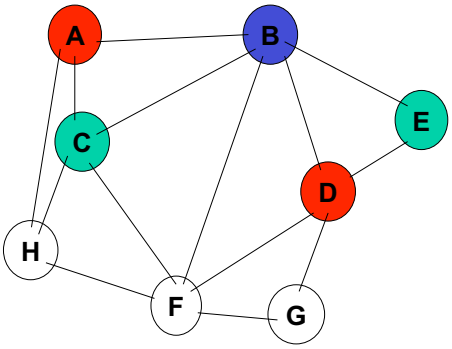
3 = green

Variables and Instantiation Order

Checking back



Example of a CSP



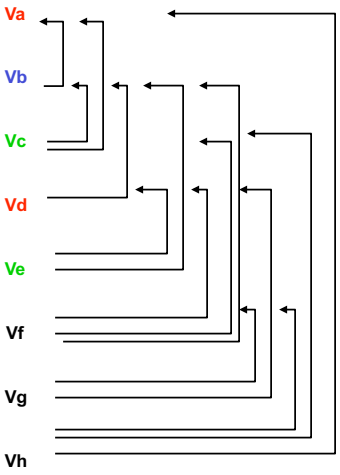
1 = red

2 = blue

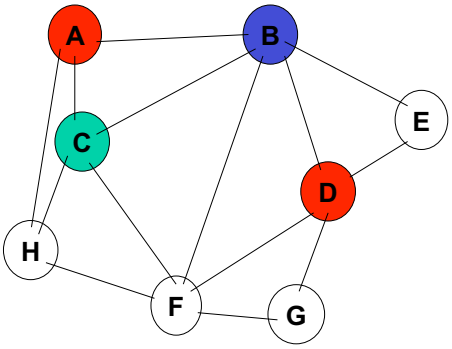
3 = green

Variables and Instantiation Order

Checking back



Example of a CSP



Why did it backtrack to E?

That was dumb!

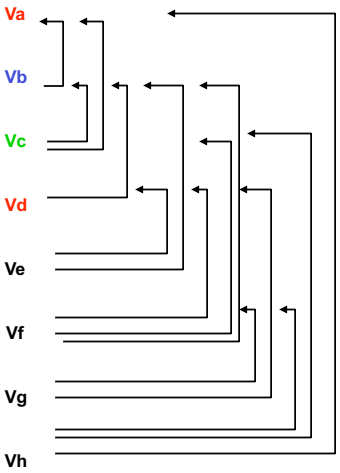
1 = red

2 = blue

3 = green

Variables and Instantiation Order

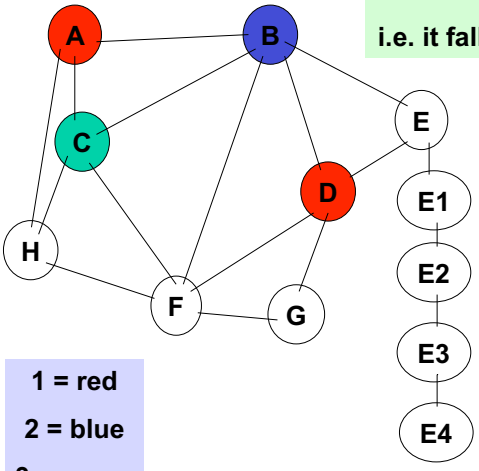
Checking back



Example of a CSP

What would have happened if we had the E* intermediate variables?

i.e. it falls back on E4, then E3, towards E?



1 = red

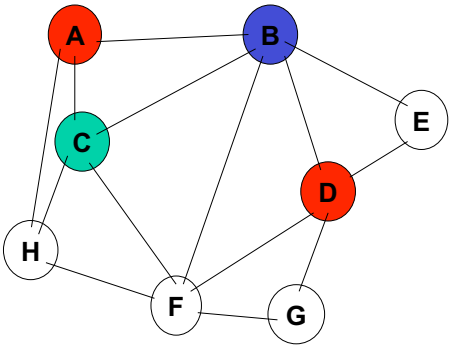
2 = blue

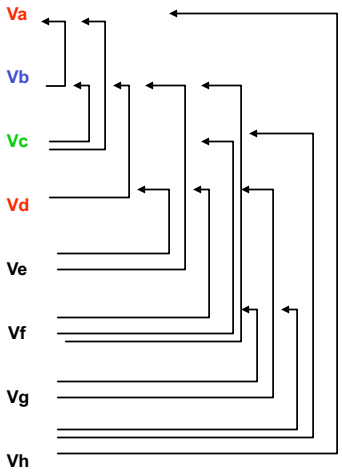
3 = green

Example of a CSP

Variables and Instantiation Order

Checking back





Why did it backtrack to E?

That was dumb!

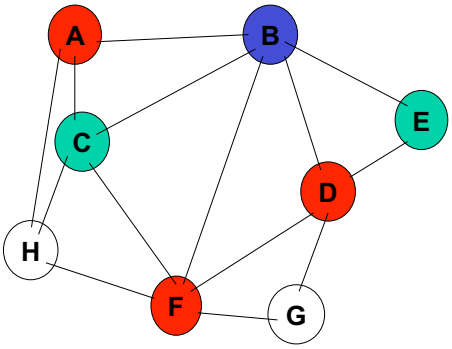
Whats better and why?

1 = red

2 = blue

3 = green

Example of a CSP

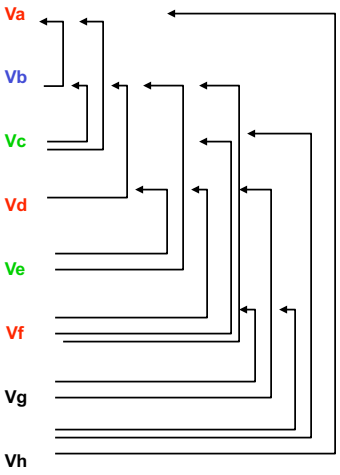


Why backtrack to D?

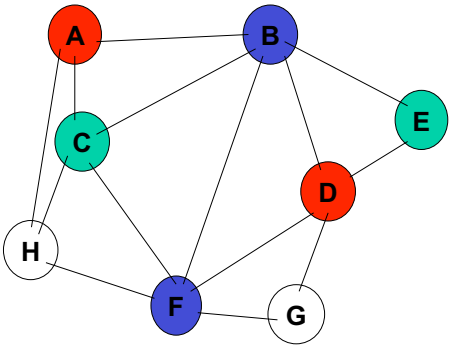
D ➔ not F

Variables and Instantiation Order

Checking back



Example of a CSP



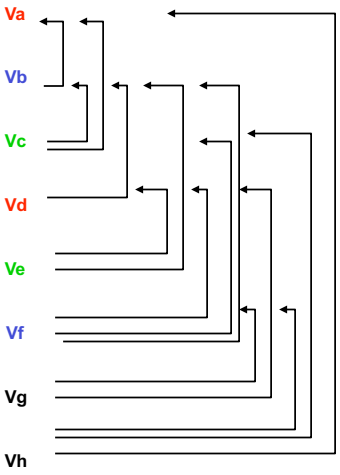
Why backtrack to D?

D ➔ not F

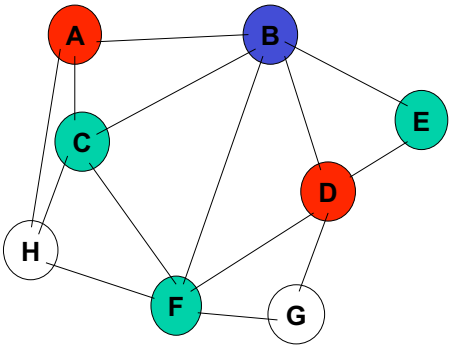
B ➔ not F

Variables and Instantiation Order

Checking back



Example of a CSP



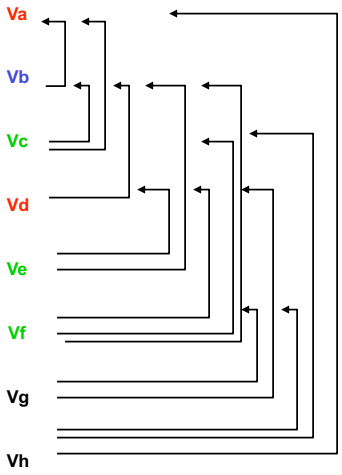
Why backtrack to D?

D → not F
B → not F
C → not F
F or F or F

not D ← F
not B ← F
not C ← F
not D or not B or not C

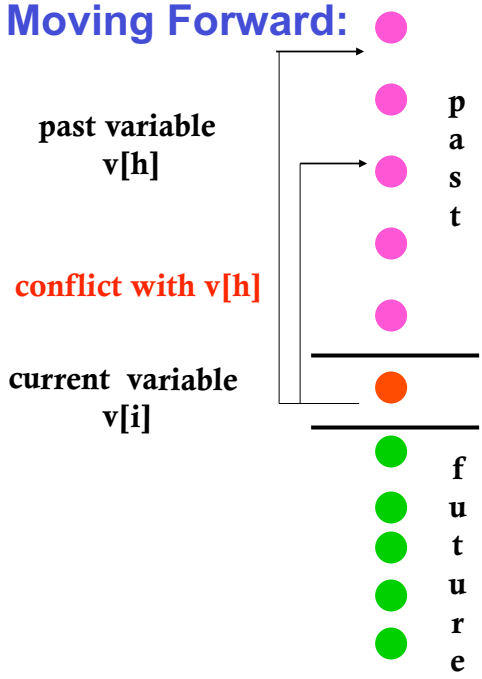
Variables and Instantiation Order

Checking back



Its safe to remove the deepest assignment.

Moving Forward:



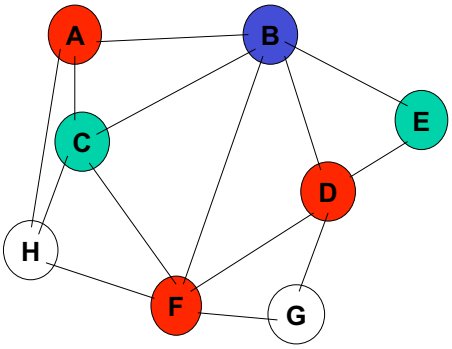
confSet[i] denotes past variables that conflict with values in the domain of v[i]

{v[1] … }

Move down like Backtrack Search:

- Instantiate v[i] := x, for next x in D[i]
- Check constraint (v[i],v[h]), if fails
  - say "v[i] is in conflict with v[h]"
  - add h to the set confSet[i]

Moving Forward

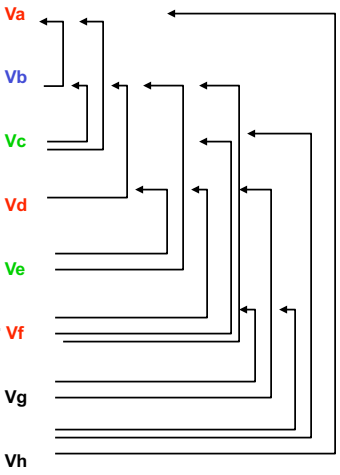


D ➔ not F

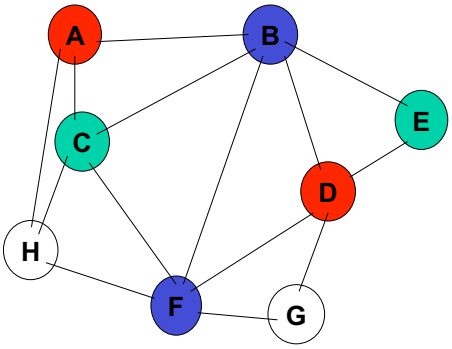
ConfSet[F] = {D,

Variables and Instantiation Order

Checking back



Moving Forward



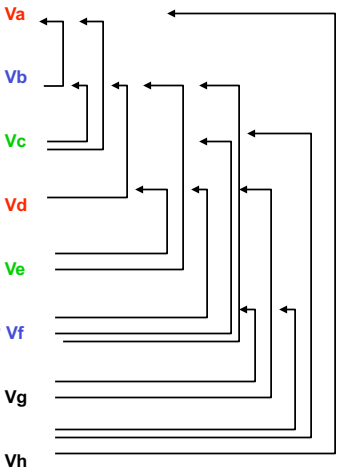
D ➔ not F

B ➔ not F

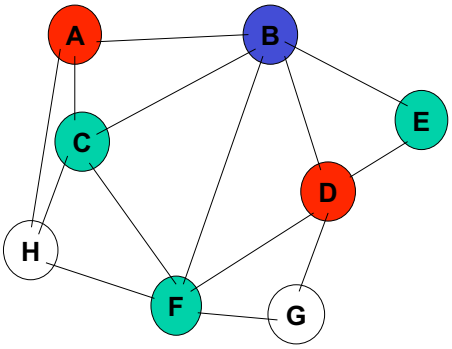
ConfSet[F] = {D, B,

Variables and Instantiation Order

Checking back

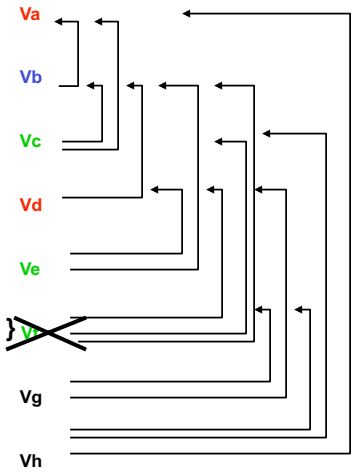


Moving Forward



Variables and Instantiation Order

Checking back



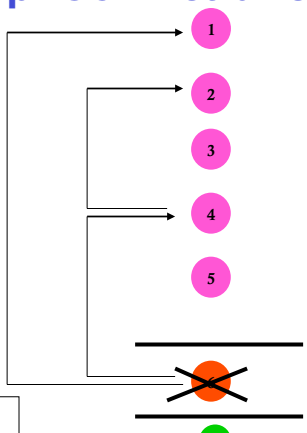
D ➔ not F

B ➔ not F

C ➔ not F

ConfSet[F] = {D, B, C }

Backing Up: Conflict-directed Back Jumping



{0}

conflict sets

{2,0}

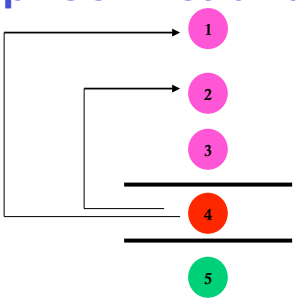
{4,1,0}

Image by MIT OpenCourseWare.

Backtrack when v[i] domain exhausted:

• Jump to deepest var h in ConfSet[i].

Backing Up: Conflict-directed Back Jumping



{0}

{2,1, ?}

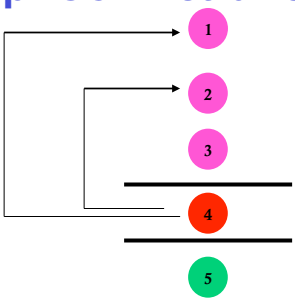
{4,1,0}

Image by MIT OpenCourseWare.

Backtrack when v[i] domain exhausted:

- **Jump to deepest var h in ConfSet[i].**
- **Update ConfSet[h].**

Backing Up: Conflict-directed Back Jumping



{0}

{2,1, ?}

Add {1, 0} to confset[4]

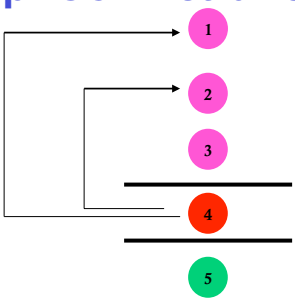
{4,1,0}

Image by MIT OpenCourseWare.

Backtrack when v[i] domain exhausted:

- **Jump to deepest var h in ConfSet[i].**
- **Update ConfSet[h] with ConfSet[i] / h.**

Backing Up: Conflict-directed Back Jumping



{0}

{2,1, 0}

{4,1,0}

Image by MIT OpenCourseWare.

Backtrack when v[i] domain exhausted:

- **Jump to deepest var h in ConfSet[i].**
- **Update ConfSet[h] with ConfSet[i] / h.**
- **Reset Domains, ConfSets below h.**
- **Move forward: try next h assignments.**

Backing Up: Conflict-directed Back Jumping

When jumping back from v[i] to v[h],

1. **Update conflict sets:**

   confSet[h] := confSet[h] ∪ confSet[i] \ {h}

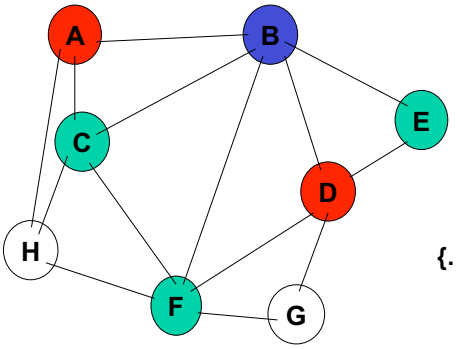
   confSet[i] := {0}

   - **This means:**
     if we later jump back from v[h],
     jump back to a variable that is
     in conflict with v[h] or with v[i].

2. **Throw away everything CBJ knows about v[i].**

3. **Reset all variables from v[h+1] to v[i] (i.e. domain and confSet)**

Backing Up



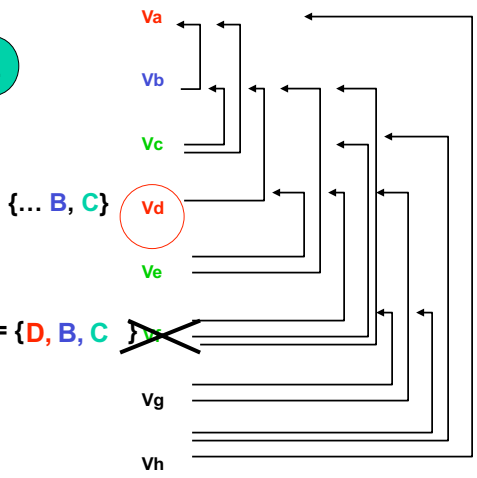
D ➔ not F

B ➔ not F

C ➔ not F

Variables and Instantiation Order

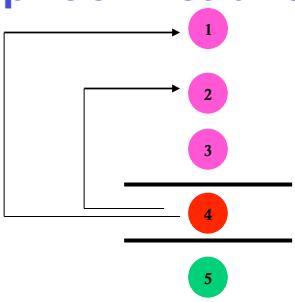
Checking back



{… B, C}

ConfSet[F] = {D, B, C }

Backing Up: Conflict-directed Back Jumping



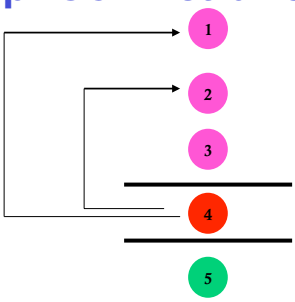
{0}

{2,1, 0}

What do we do if variable 4's domain is exhausted as well?

Image by MIT OpenCourseWare.

Backing Up: Conflict-directed Back Jumping



{0}

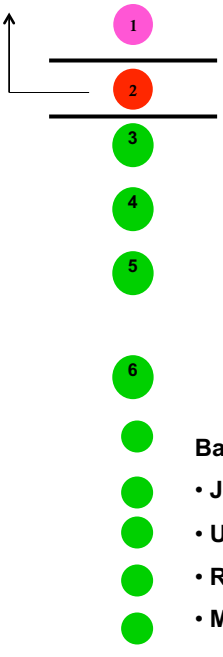
{2,1, 0}

Image by MIT OpenCourseWare.

Backtrack when v[h] domain exhausted:

- **Jump to deepest var g in ConfSet[h].**
- **Update ConfSet[g] with ConfSet[h] / g.**
- **Reset Domains, ConfSets below g.**
- **Move forward: try next g assignments.**

Backing Up: Conflict-directed Back Jumping



{1, 0}

Image by MIT OpenCourseWare.

Backtrack when v[h] domain exhausted:

- **Jump to deepest var g in ConfSet[h].**
- **Update ConfSet[g] with ConfSet[h] / g.**
- **Reset Domains, ConfSets below g.**
- **Move forward: try next g assignments.**

CBJ Supports Successive Jumps

If there are no values remaining for v[i]
Jump back to v[h], the deepest variable in conflict with v[i].
The hope: re-instantiating v[h] will allow us to find a good value for v[i]

If there are no values remaining for v[h]
Jump back to v[g], the deepest variable in conflict with v[i] or v[h].
The hope: re-instantiating v[g] will allow us to find a good value for v[i] or a good value for v[h] that will be good for v[i]

If there are no values remaining for v[g]
Jump back to v[f], the deepest variable in conflict with v[i] or v[h] or v[g]
The hope: re-instantiating v[f] will allow us to find a good value for v[i] or a good value for v[h] that will be good for v[i] or a good value for v[g] that will be good for v[h] and v[i]

CBJ: Prosser's Original Formulation

```
consistent := false
confSet[i] := {0}
for x in domain[i] while not(consistent)   // find a consistent value
begin
  consistent := true
  v[i] := x
  for h in (1 .. i-1) while consistent       // check backwards
  begin
    consistent := (check(v[i],v[h])
    if not(consistent) then confSet[i] := confSet[i] ∪ {h}
  end
  if not(consistent)
  then delete(x,domain[i])
end
```

CBJ: Dechter Formulation

Pseudocode for conflict-directed backjumping removed due to copyright restrictions.

Conflict-directed Back Jumping: Supporting definitions

Supporting definitions (earlier constraint, earlier minimal conflict set, jumpback set) removed due to copyright restrictions.

To Solve CSP <X,D,C> We Combine:

1. Reasoning - Arc consistency via constraint propagation
   - Eliminates values that are shown locally to not be a part of any solution.
2. Search
   - Explores consequences of committing to particular assignments.

Methods That Incorporate Search:

- Standard Search
- Back Track Search (BT)
- BT with Forward Checking (FC)
- Dynamic Variable Ordering (DV)
- Iterative Repair (IR)
- Conflict-directed Back Jumping (CBJ)

39

MIT OpenCourseWare
http://ocw.mit.edu

16.410 / 16.413 Principles of Autonomy and Decision Making

Fall 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.