Lecture 6

Instructor: Amit Kumar Das

Senior Lecturer,

Department of Computer Science & Engineering,

East West University

Dhaka, Bangladesh.

Which variable should be assigned next?

- **Most constrained variable:**
  - Choose the variable with the fewest legal values
  - A.k.a. **minimum remaining values** (MRV) heuristic

Which variable should be assigned next?

- **Most constrained variable:**
  - Choose the variable with the fewest legal values
  - A.k.a. **minimum remaining values** (MRV) heuristic

Which variable should be assigned next?

- **Most constraining variable:**
  – Choose the variable that imposes the most constraints on the remaining variables
  – Tie-breaker among most constrained variables

Which variable should be assigned next?

- **Most constraining variable:**
  - Choose the variable that imposes the most constraints on the remaining variables
  - Tie-breaker among most constrained variables

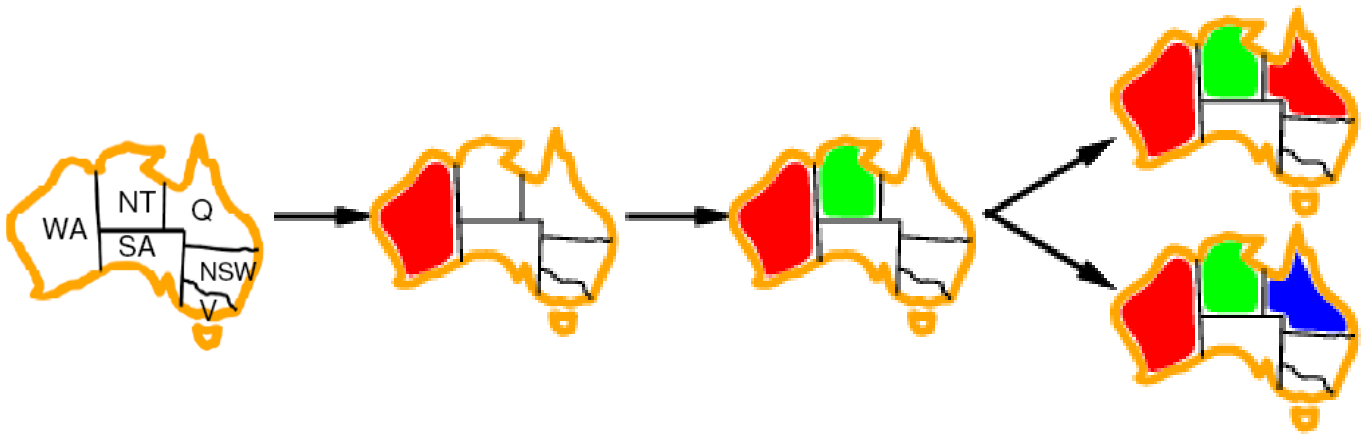
Given a variable, in which order should its values be tried?

- Choose the **least constraining value**:
  - The value that rules out the fewest values in the remaining variables

Given a variable, in which order should its values be tried?

- Choose the **least constraining value**:
  - The value that rules out the fewest values in the remaining variables

Which assignment for Q should we choose?

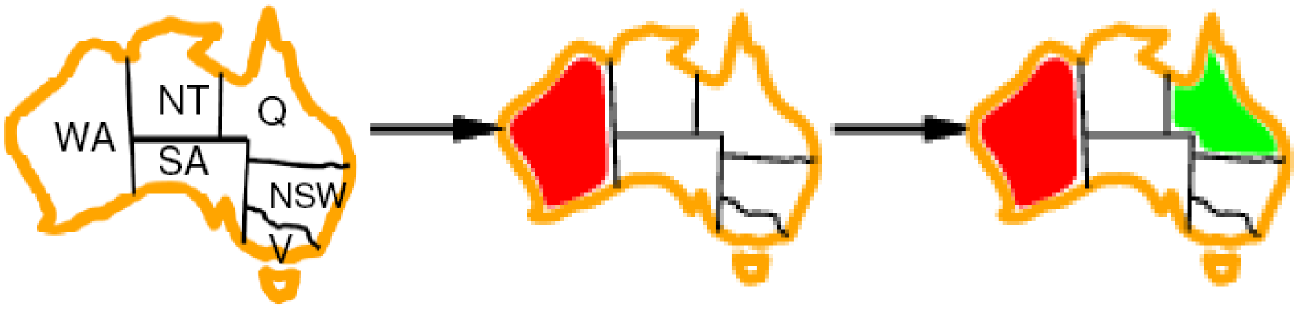


Early detection of failure

function RECURSIVE-BACKTRACKING(*assignment*, *csp*)
    if *assignment* is complete **then return** *assignment*
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*], *assignment*, *csp*)
    for each *value* **in** ORDER-DOMAIN-VALUES(*var*, *assignment*, *csp*)
        if *value* is consistent with *assignment* given CONSTRAINTS[*csp*]
            add $\{var = value\}$ to *assignment*
            result ← RECURSIVE-BACKTRACKING(*assignment*, *csp*)
            if $result \neq failure$ **then return** *result*
            remove $\{var = value\}$ from *assignment*
    return *failure*

Apply *inference* to reduce the space of possible assignments and detect failure early

Early detection of failure



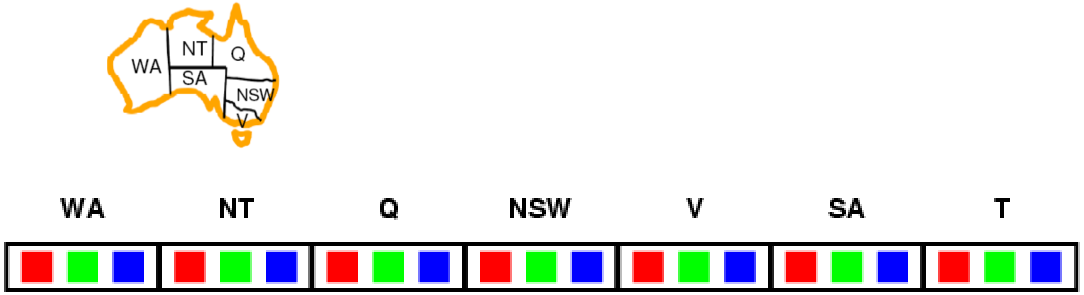
Apply *inference* to reduce the space of possible assignments and detect failure early

Early detection of failure: Forward checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values

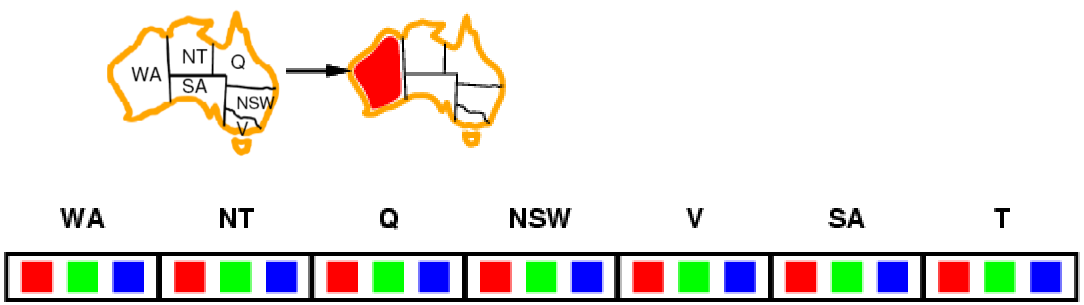
Early detection of failure: Forward checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



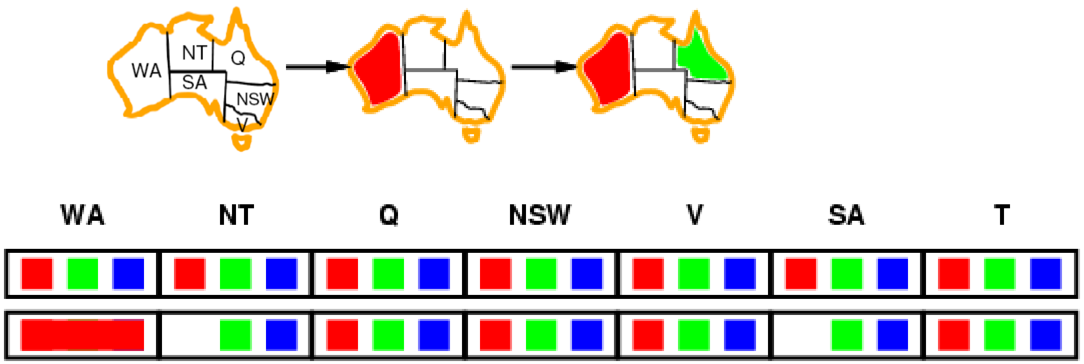
Early detection of failure: Forward checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



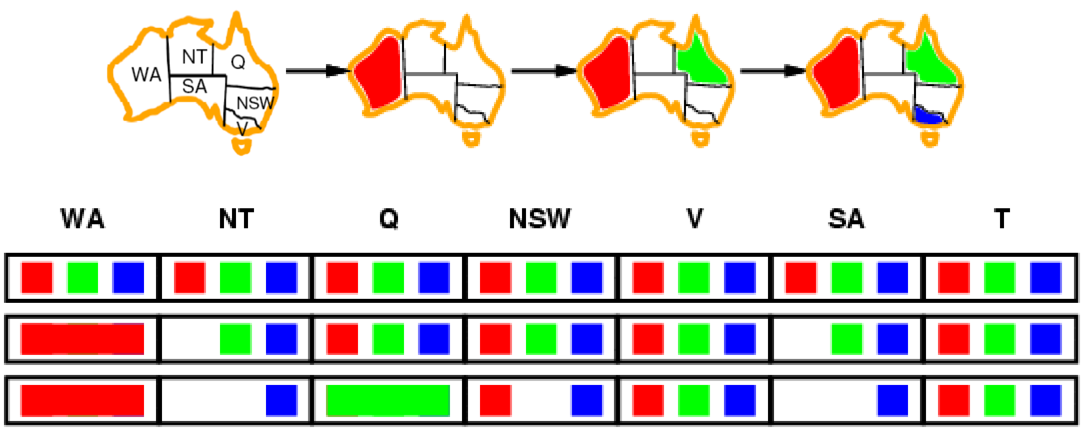
Early detection of failure: Forward checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



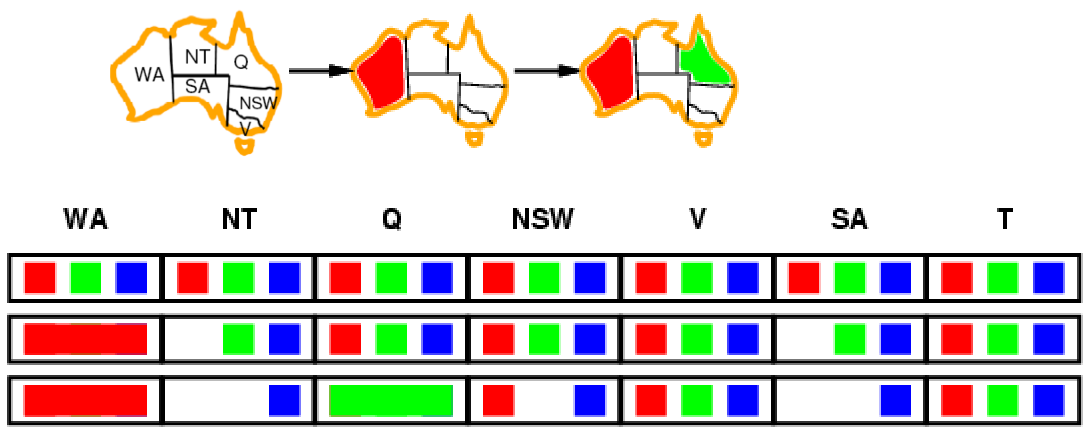
Early detection of failure: Forward checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



Constraint propagation

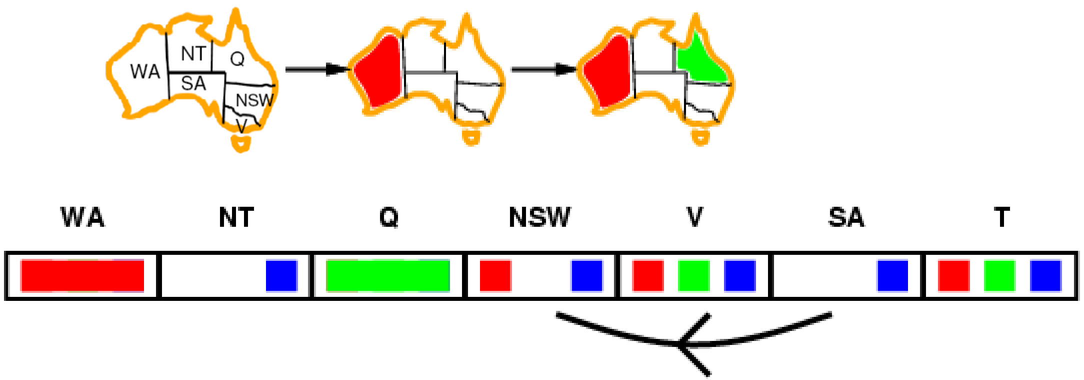
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures



- NT and SA cannot both be blue!
- **Constraint propagation** repeatedly enforces constraints *locally*

Arc consistency

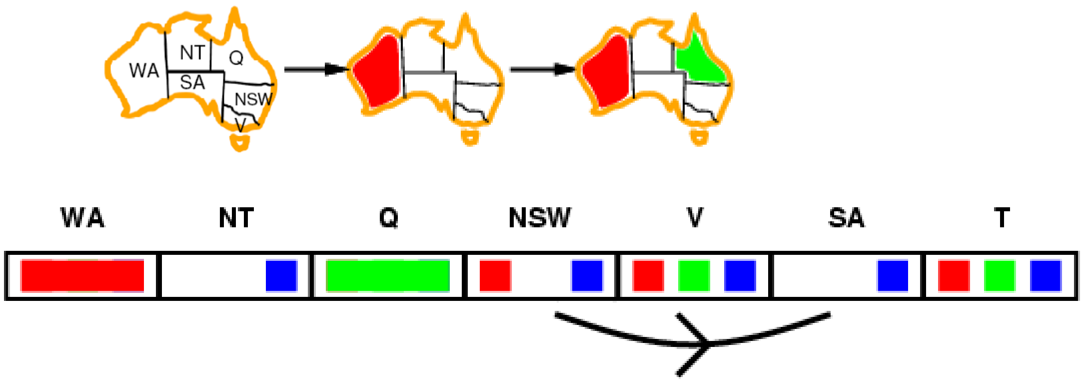
- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*



Consistent!

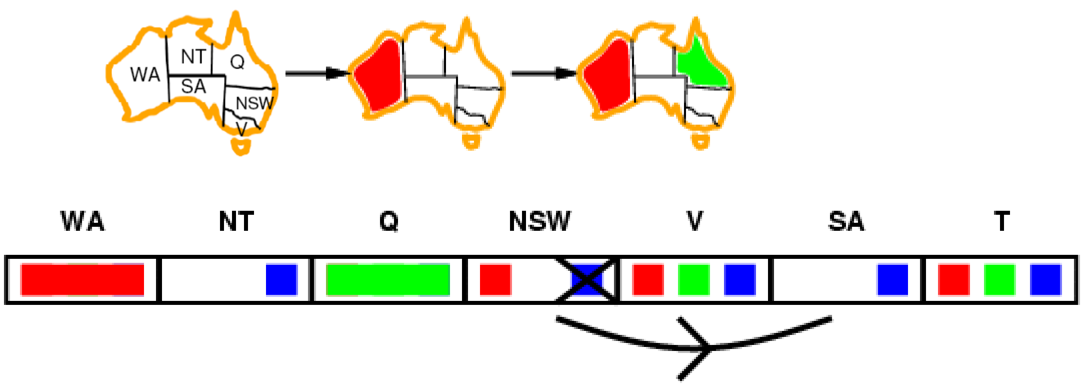
Arc consistency

- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*



Arc consistency

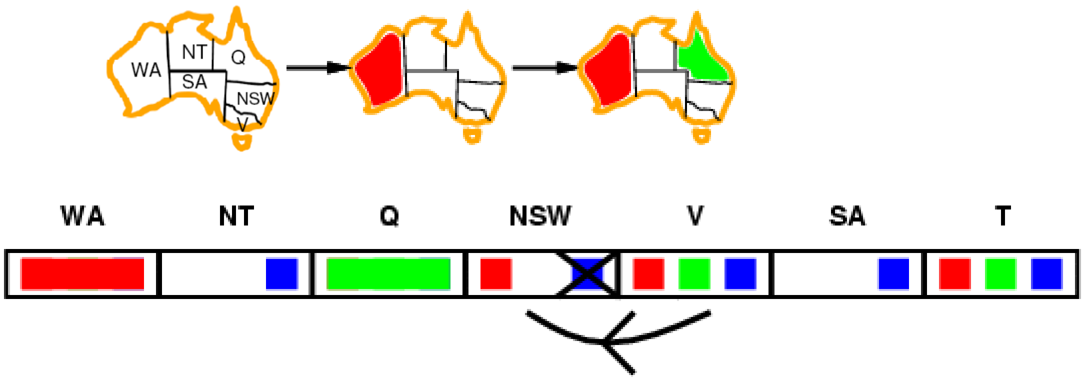
- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y



- If *X* loses a value, all pairs *Z* → *X* need to be rechecked

Arc consistency

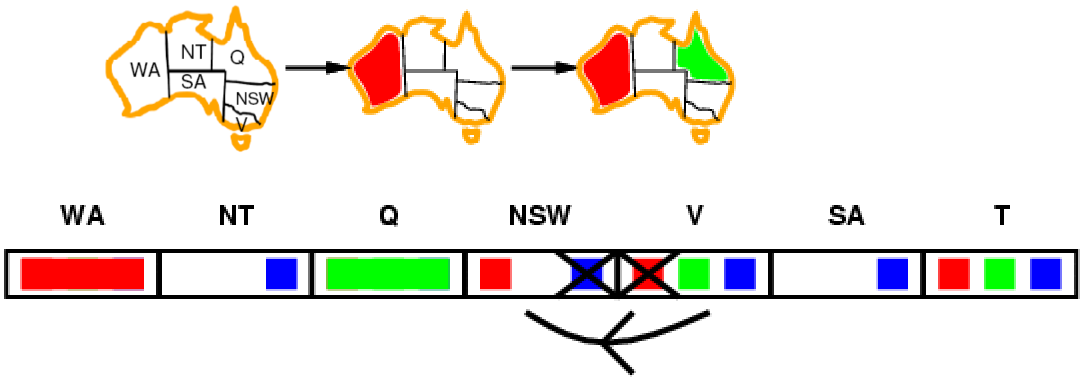
- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y



- If *X* loses a value, all pairs *Z* → *X* need to be rechecked

Arc consistency

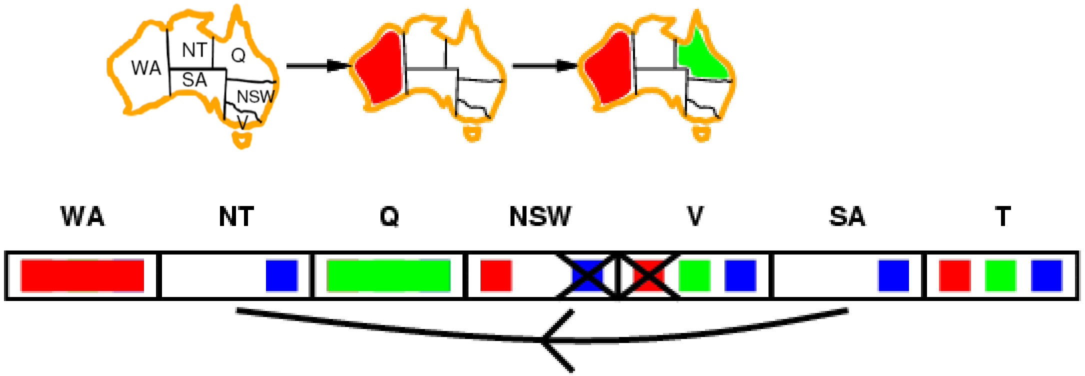
- Simplest form of propagation makes each pair of variables **consistent:**
  - $X \rightarrow Y$ is consistent iff for every value of X there is some allowed value of Y
  - When checking $X \rightarrow Y$, throw out any values of X for which there isn't an allowed value of Y



- If X loses a value, all pairs $Z \rightarrow X$ need to be rechecked

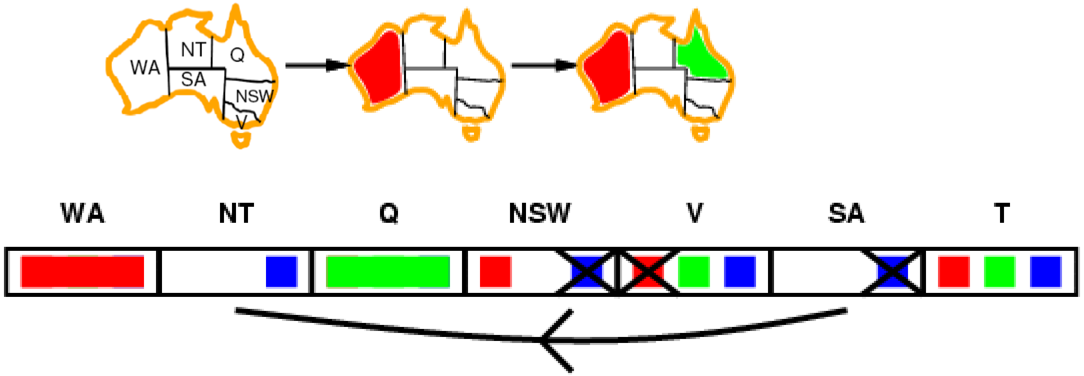
Arc consistency

- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y



Arc consistency

- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y



- Arc consistency detects failure earlier than forward checking
- Can be run before or after each assignment

Arc consistency algorithm AC-3

function AC-3(csp) **returns** the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$
  local variables: $queue$, a queue of arcs, initially all the arcs in csp

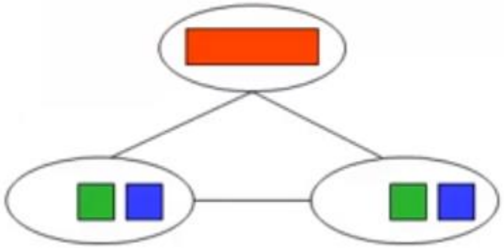
  while $queue$ is not empty
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST($queue$)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) **then**
      for each X_k **in** NEIGHBORS[X_i] **do**
        add (X_k, X_i) to $queue$

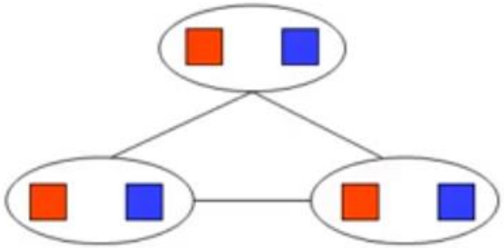
---

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **returns** true iff succeeds
  $removed \leftarrow false$
  for each x **in** DOMAIN[X_i]
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint $X_i \leftrightarrow X_j$
      then delete x from DOMAIN[X_i]; $removed \leftarrow true$
  return $removed$

Limitations of Arc Consistency

- After enforcing arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)

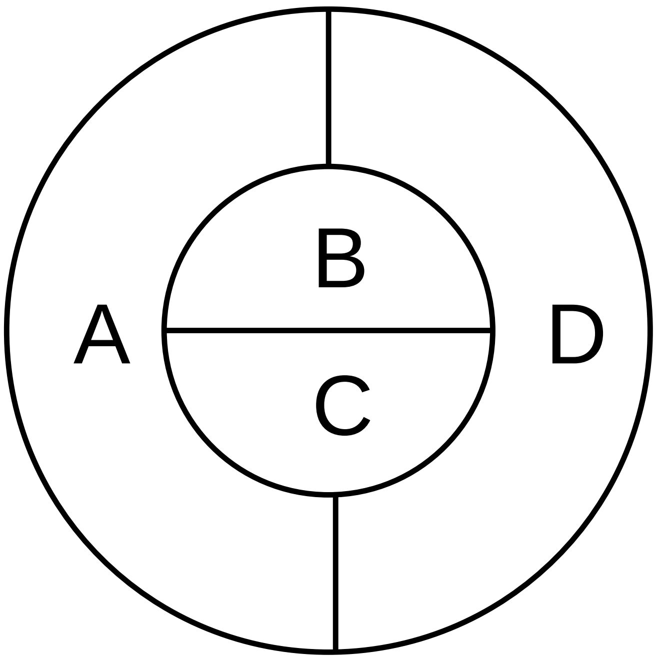


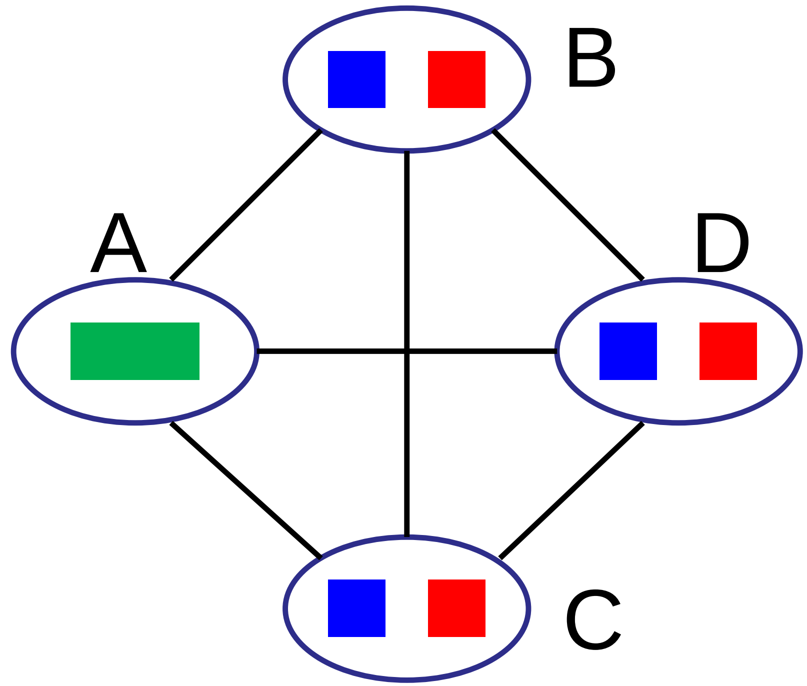


What went wrong here?

[demo: arc consistency]

Does arc consistency always detect the lack of a solution?





- There exist stronger notions of consistency (path consistency, k-consistency), but we won't worry about them

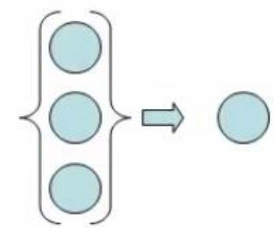
K-Consistency

K-Consistency

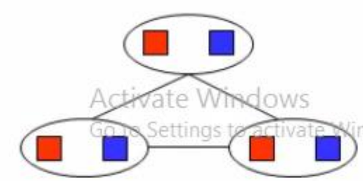
- Increasing degrees of consistency
  - 1-Consistency (Node Consistency): Each single node's domain has a value which meets that node's unary constraints

  - 2-Consistency (Arc Consistency): For each pair of nodes, any consistent assignment to one can be extended to the other

  - K-Consistency: For each k nodes, any consistent assignment to k-1 can be extended to the k^{th} node.



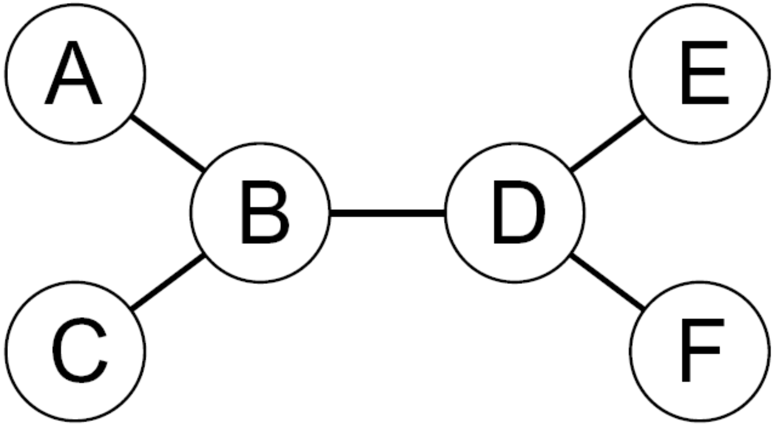
- Higher k more expensive to compute
- (You need to know the k=2 case: arc consistency)



Structure

Tree-structured CSPs

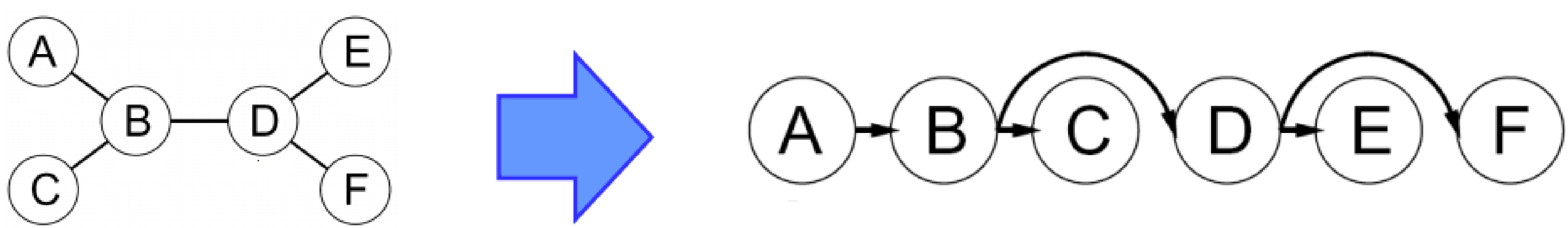
- Certain kinds of CSPs can be solved without resorting to backtracking search!
- *Tree-structured CSP*: constraint graph does not have any loops



Source: P. Abbeel, D. Klein, L. Zettlemoyer

Algorithm for tree-structured CSPs

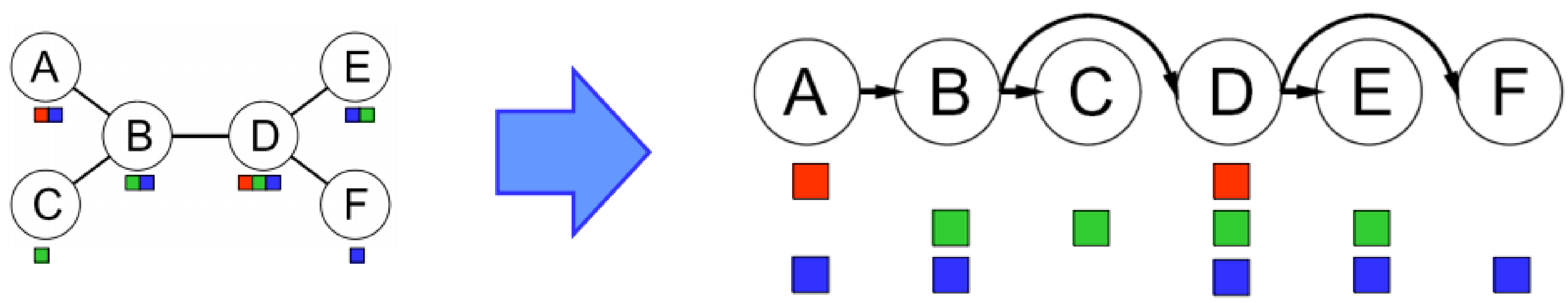
- Choose one variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering



http://cs188ai.wikia.com/wiki/Tree_Structure_CSPs

Algorithm for tree-structured CSPs

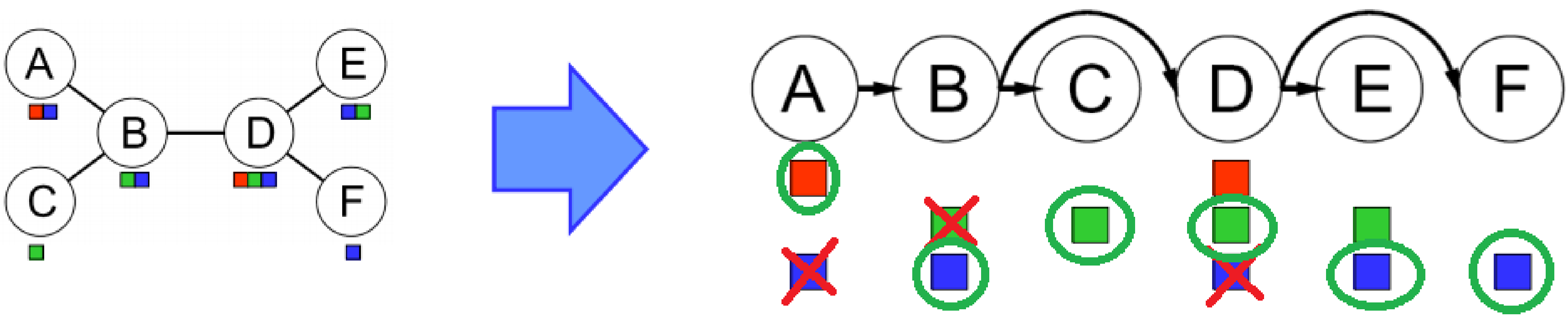
- Choose one variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering
- Backward removal phase: check arc consistency starting from the rightmost node and going backwards



http://cs188ai.wikia.com/wiki/Tree_Structure_CSPs

Algorithm for tree-structured CSPs

- Choose one variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering
- Backward removal phase: check arc consistency starting from the rightmost node and going backwards
- Forward assignment phase: select an element from the domain of each variable going left to right. We are guaranteed that there will be a valid assignment because each arc is consistent

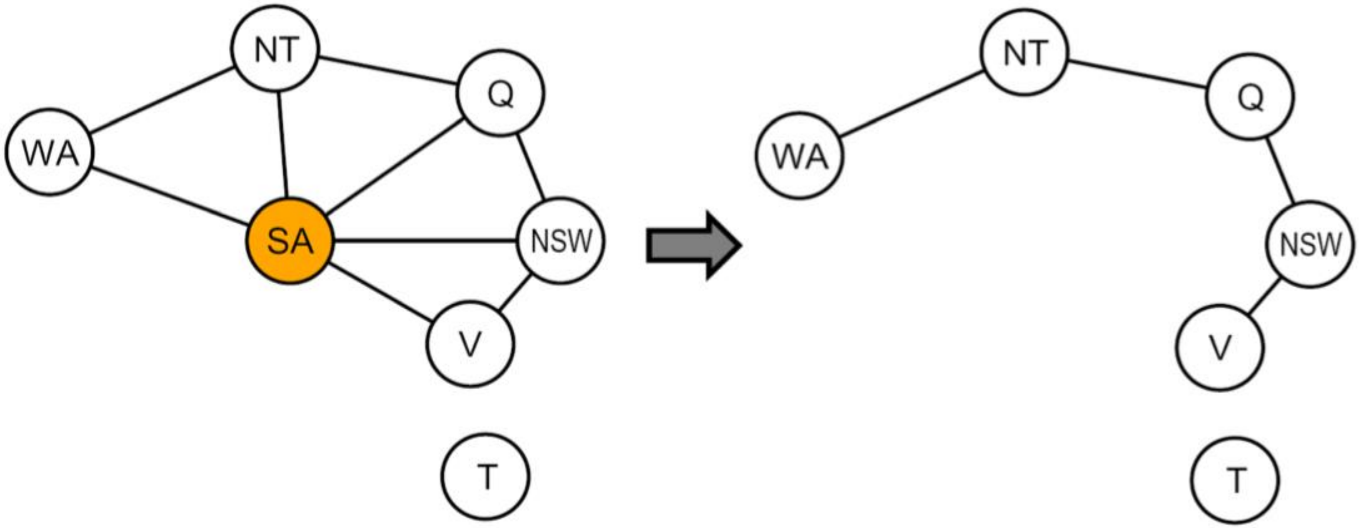


http://cs188ai.wikia.com/wiki/Tree_Structure_CSPs

Algorithm for tree-structured CSPs

- If n is the numebr of variables and m is the domain size, what is the running time of this algorithm?
  - O(nm^2): we have to check arc consistency once for every node in the graph (every node has one parent), which involves looking at pairs of domain values

Nearly tree-structured CSPs



- **Cutset conditioning:** find a subset of variables whose removal makes the graph a tree, instantiate that set in all possible ways, prune the domains of the remaining variables and try to solve the resulting tree-structured CSP
- Cutset size c gives runtime $O(m^c (n - c)m^2)$

Source: P. Abbeel, D. Klein, L. Zettlemoyer

Algorithm for tree-structured CSPs

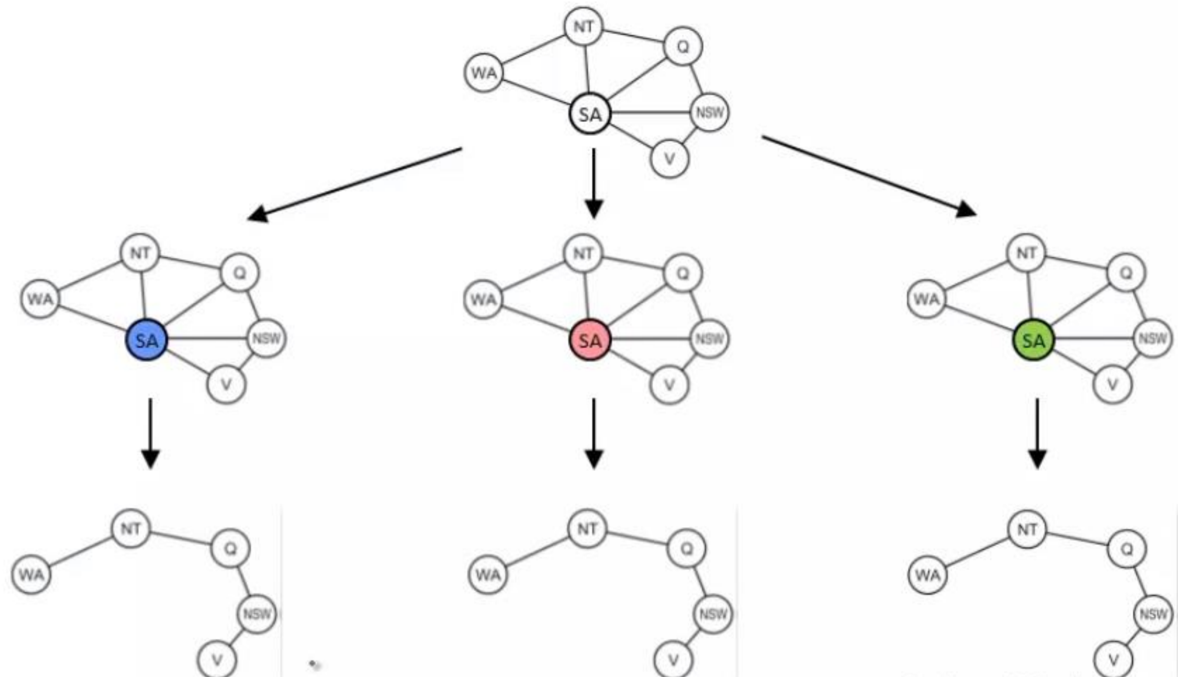
- Running time is $O(nm^2)$
  (n is the number of variables, m is the domain size)
  - We have to check arc consistency once for every node in the graph (every node has one parent), which involves looking at pairs of domain values
- What about backtracking search for general CSPs?
  - Worst case $O(m^n)$
- Can we do better?

Cutset Conditioning

Choose a cutset

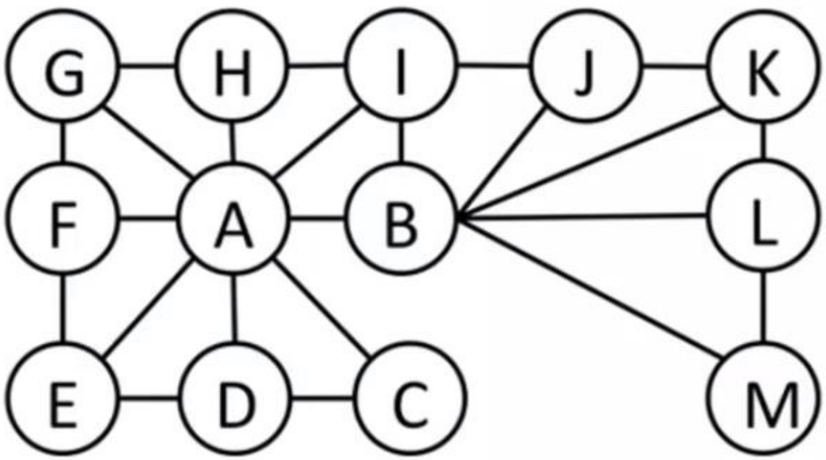
Instantiate the cutset (all possible ways)

Compute residual CSP for each assignment



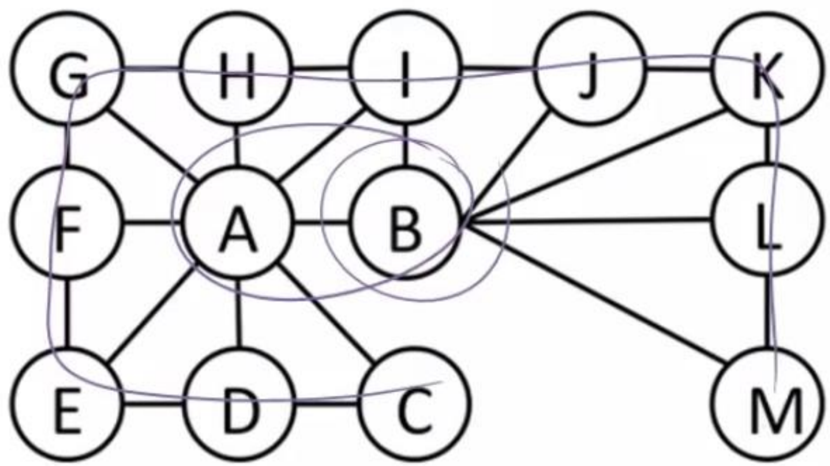
Cutset Quiz

- Find the smallest cutset for the graph below.



Cutset Quiz

- Find the smallest cutset for the graph below.



Thank You