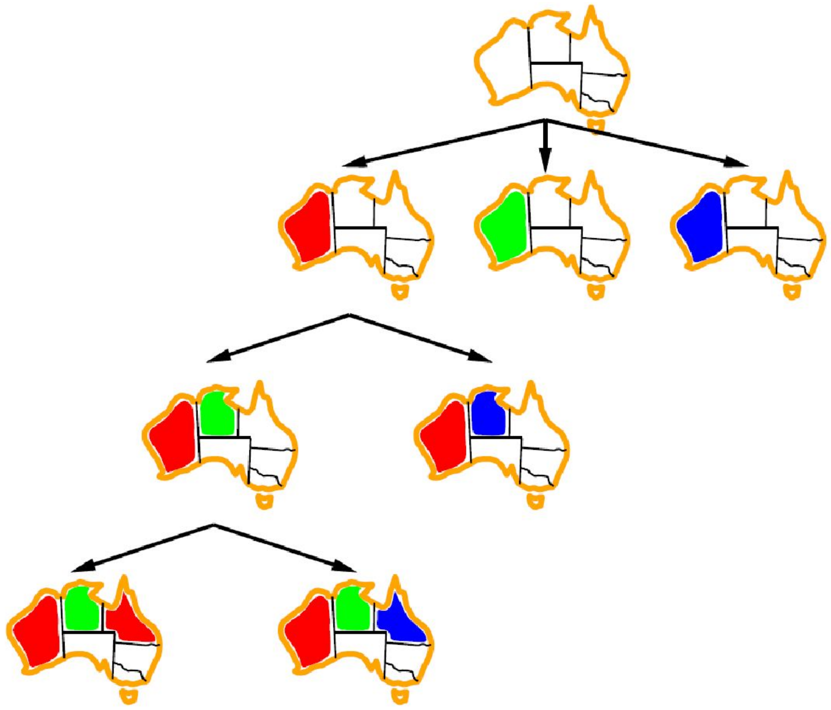
Constraint Satisfaction Problems



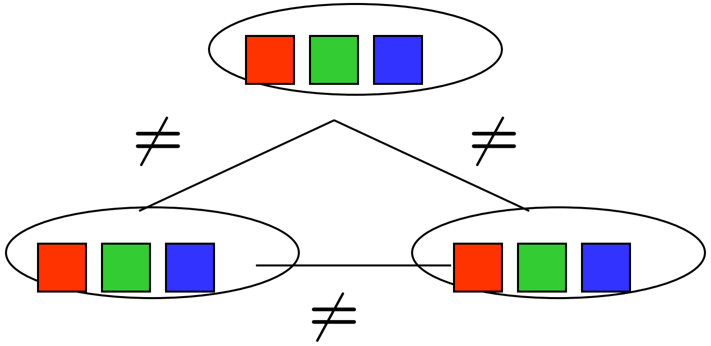
These slides are based on the slides created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley-http://ai.berkeley.edu.
The artwork is by Ketrina Yim.

Today

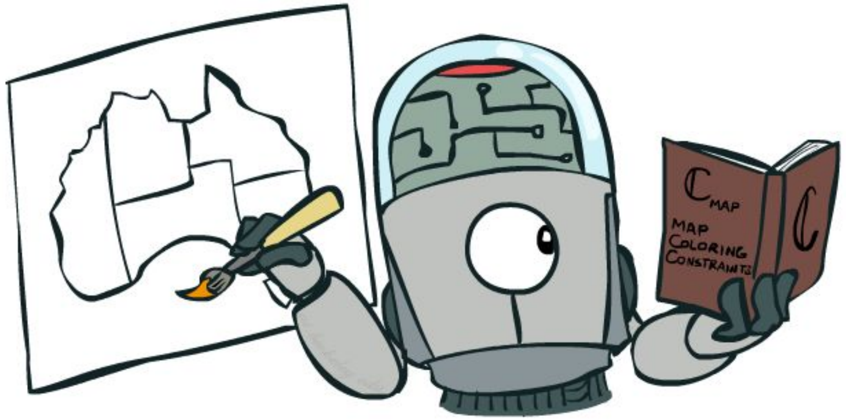
- Efficient Solution of CSPs

Reminder: CSPs

- CSPs:
  - Variables
  - Domains
  - Constraints
    - Implicit (provide code to compute)
    - Explicit (provide a list of the legal tuples)
    - Unary / Binary / N-ary



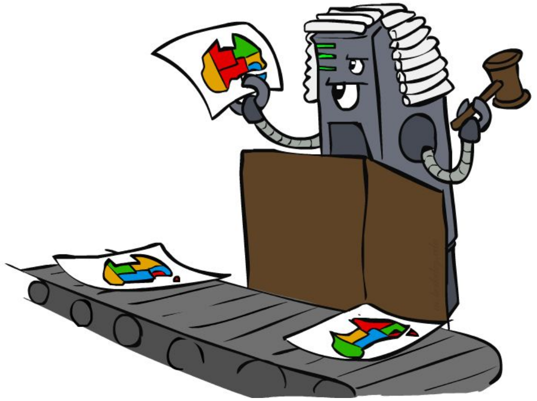
- Goals:
  - find some solution



Standard Search Formulation of CSPs

States are defined by the values assigned so far (partial assignments).

- Initial state: the empty assignment, {}
- Successor function: assign a value to an unassigned variable
- Goal test: the current assignment is complete and satisfies all constraints

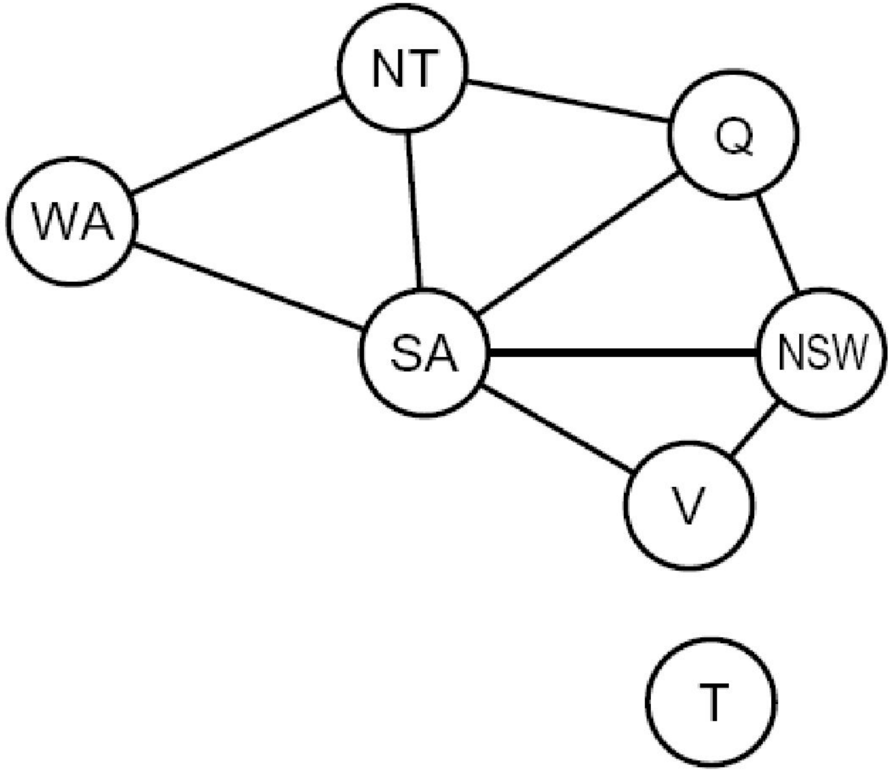


Search Methods

- BFS is a really bad choice
  root ({}) has n * d successors
  all solutions are at depth n

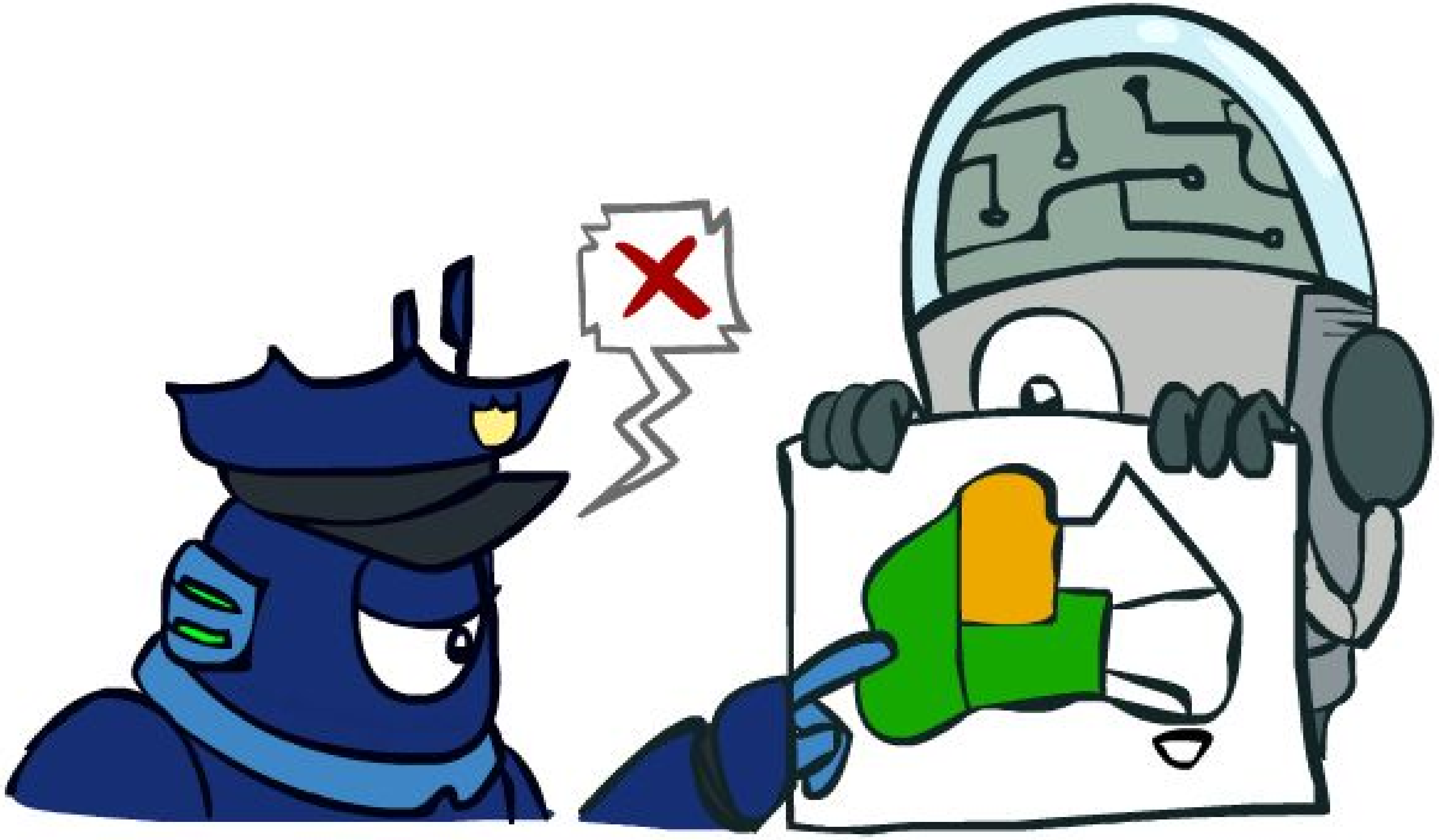
- DFS has issues

  go down to depth n even though failure
  happens much earlier



Backtracking Search

Backtracking search is the basic uninformed algorithm for solving CSPs



Backtracking Search

Idea 1: One variable at a time

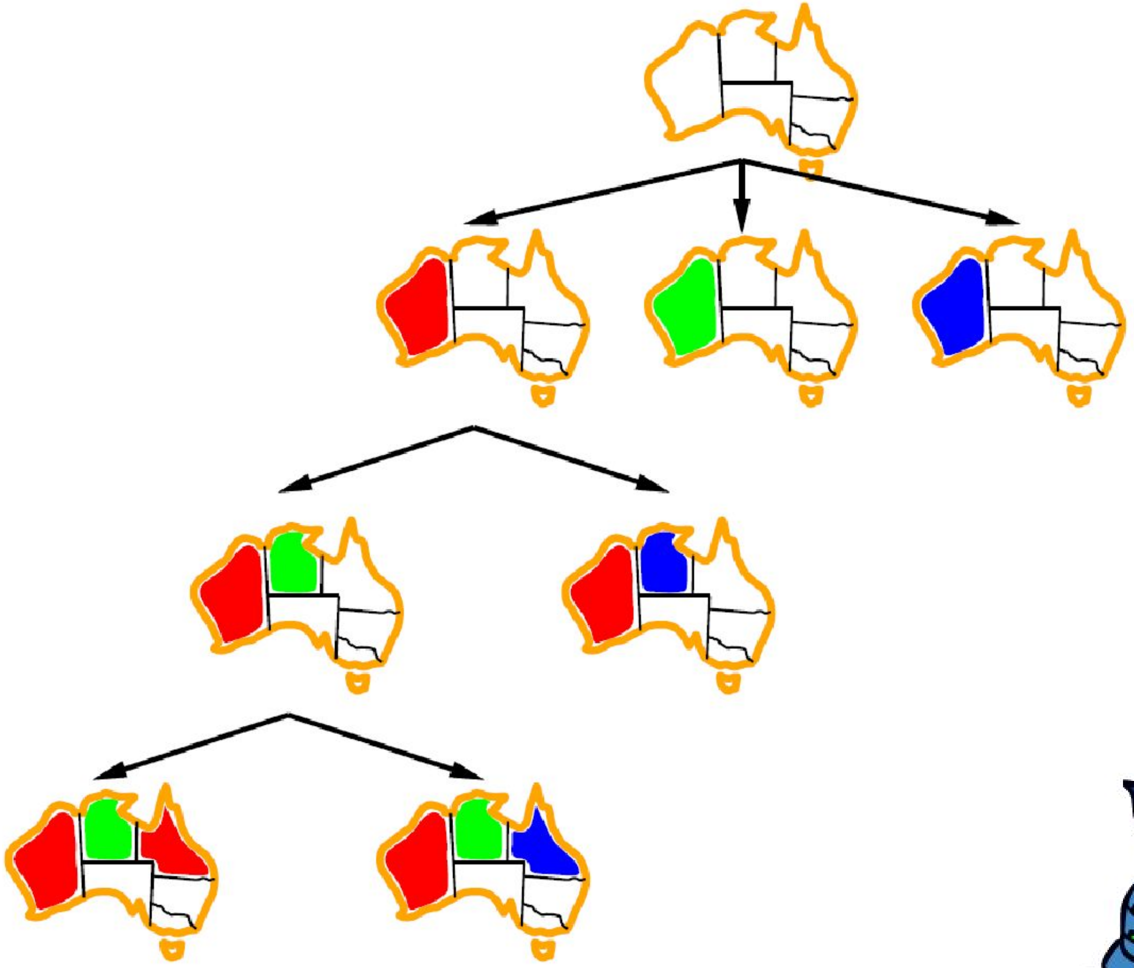
- Variable assignments are commutative, so fix ordering
- I.e., [WA = red then NT = green] same as [NT = green then WA = red]
- Only need to consider assignments to a single variable at each step

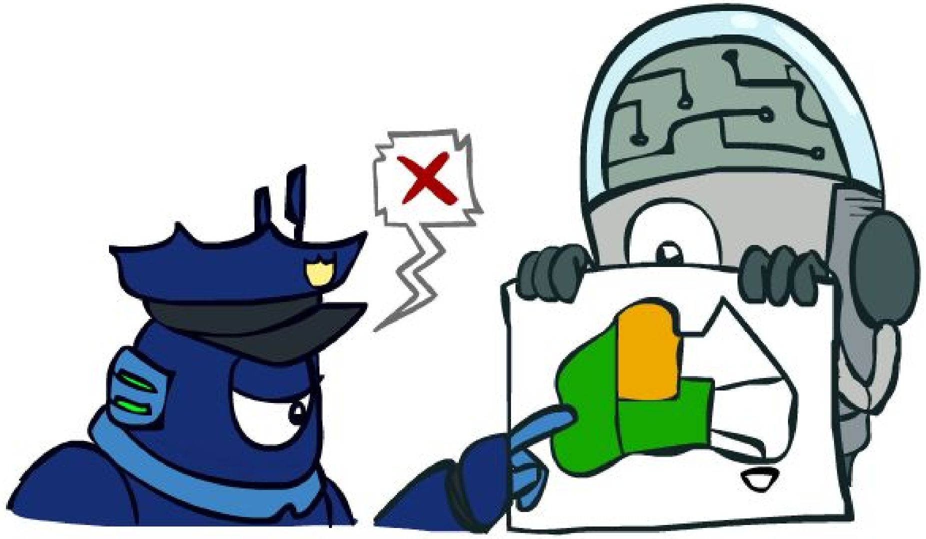
Idea 2: Check constraints as we go

- Consider only values which do not conflict previous assignments
- Might have to do some computation to check the constraints
- "Incremental goal test" (failure test)

Depth-first search with these two improvements is called *backtracking search*

Backtracking Example





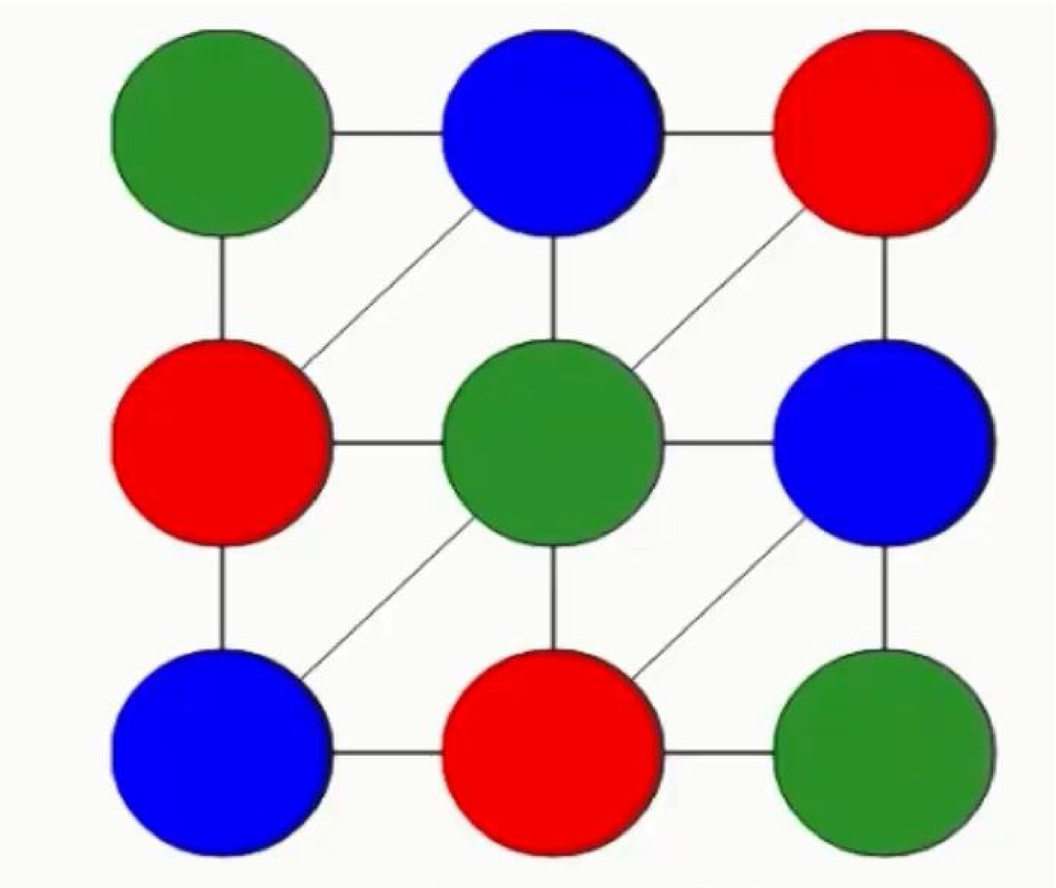
Backtracking Search

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
   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) do
       if value is consistent with assignment given CONSTRAINTS[csp] then
           add {var = value} to assignment
           result ← RECURSIVE-BACKTRACKING(assignment, csp)
           if result ≠ failure then return result
           remove {var = value} from assignment
   return failure
```

- Backtracking = DFS + single variable assignments + fail-on-violation
- Choice points? order of variable assignments, values

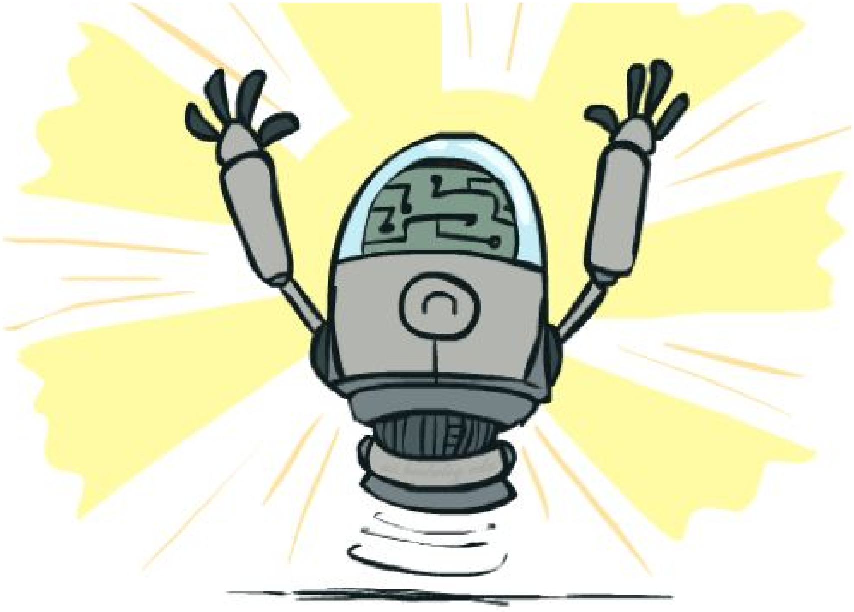
Demo Coloring – Backtracking



Improving Backtracking

General-purpose ideas give huge gains in speed

… but it's all still NP-hard



- Filtering: Can we detect inevitable failure early?
- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?
- Structure: Can we exploit the problem structure?

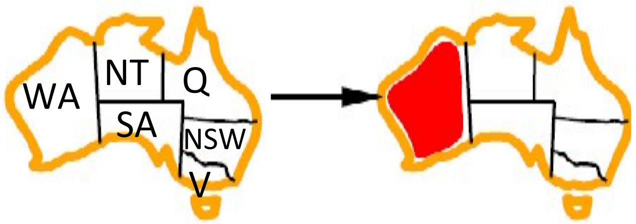
Filtering

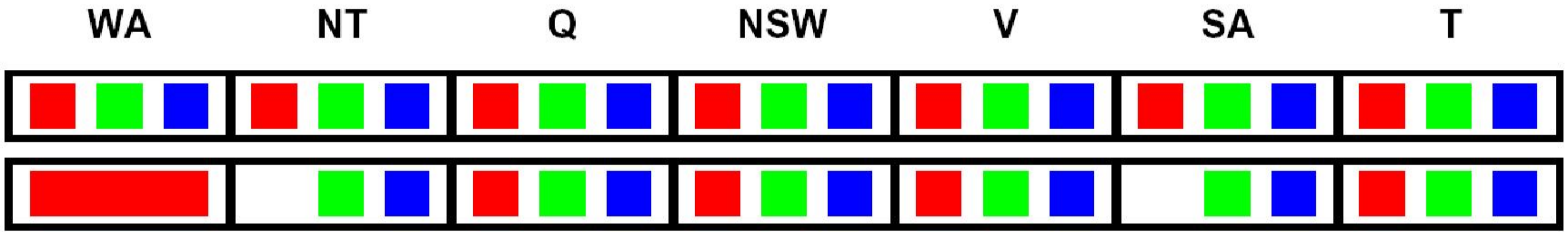
Filtering: Forward Checking

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

Filtering: Forward Checking

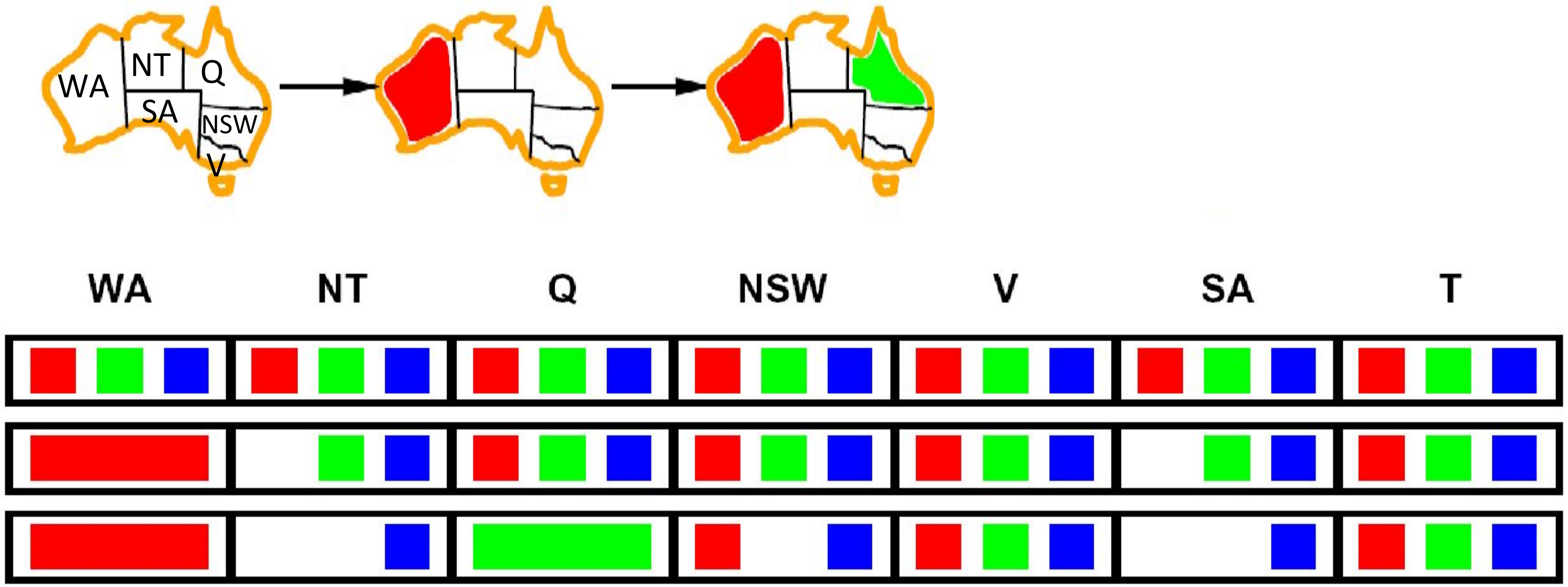
- Idea: Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values





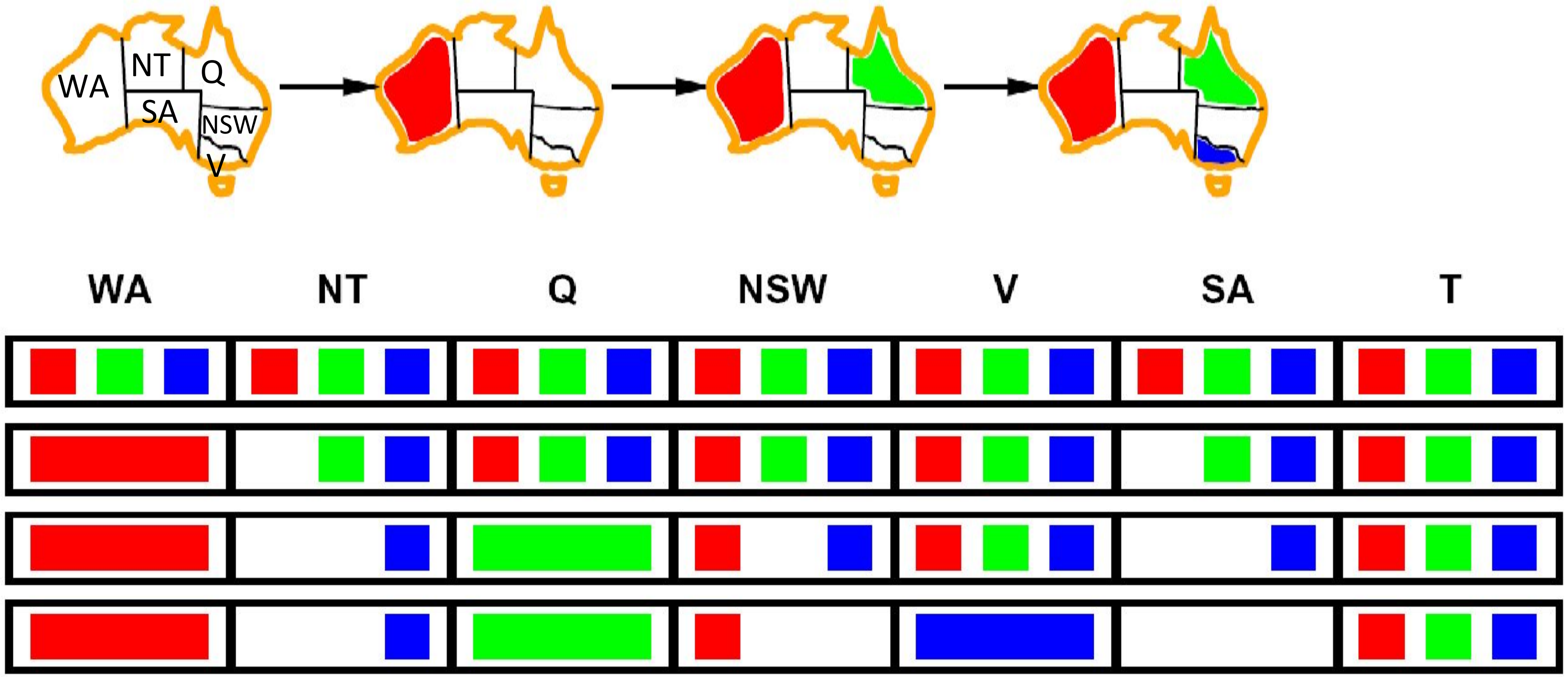
Filtering: Forward Checking

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



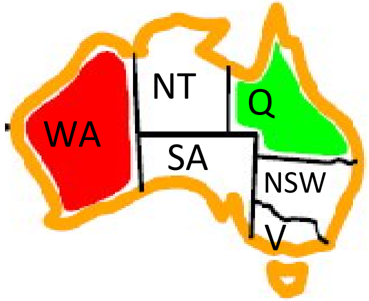
Filtering: Forward Checking

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



Limitation: Forward Checking

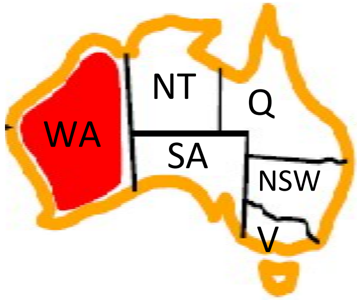
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

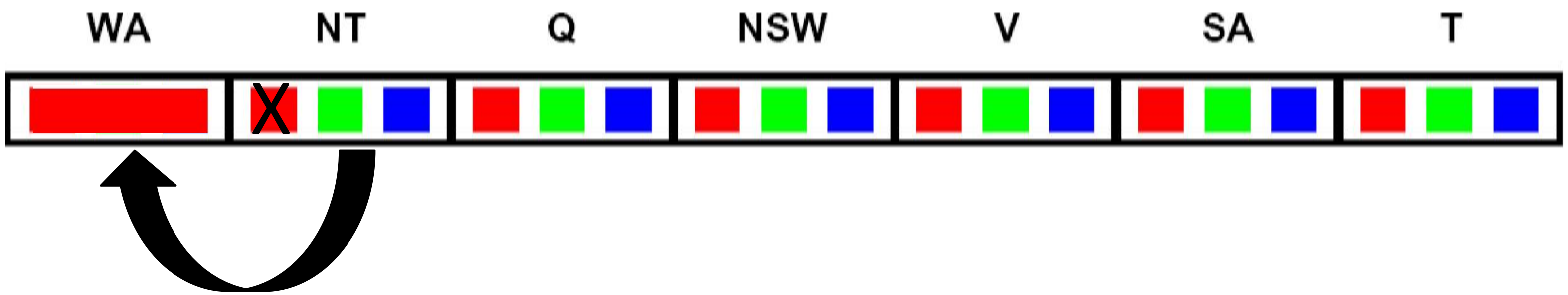


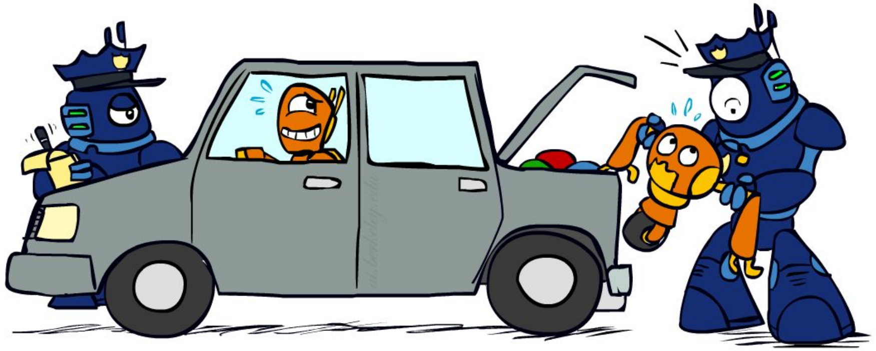
- NT and SA cannot both be blue!
- Why didn’t we detect this yet?
- *Constraint propagation:* repeatedly enforces constraints locally

Consistency of A Single Arc

- An arc X → Y is consistent iff for *every* x in the tail there is *some allowed* y in the head which could be assigned without violating a constraint



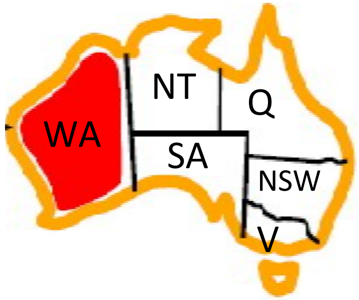


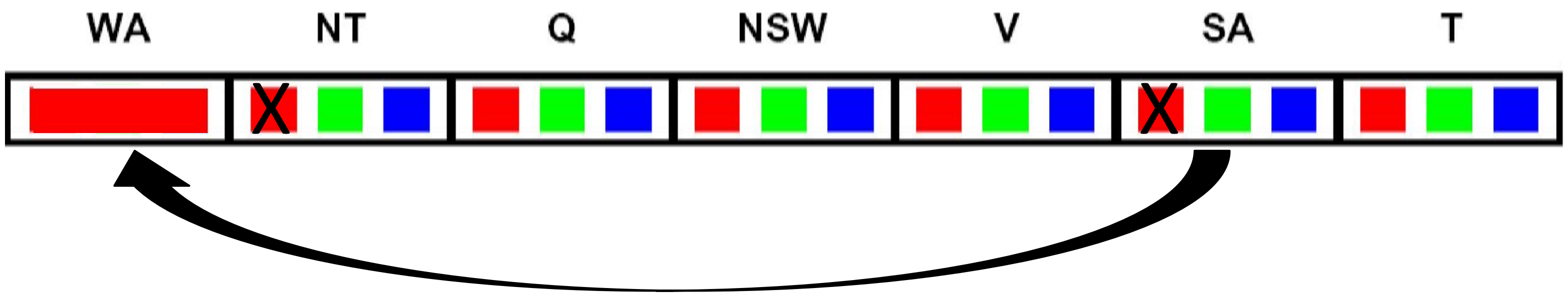


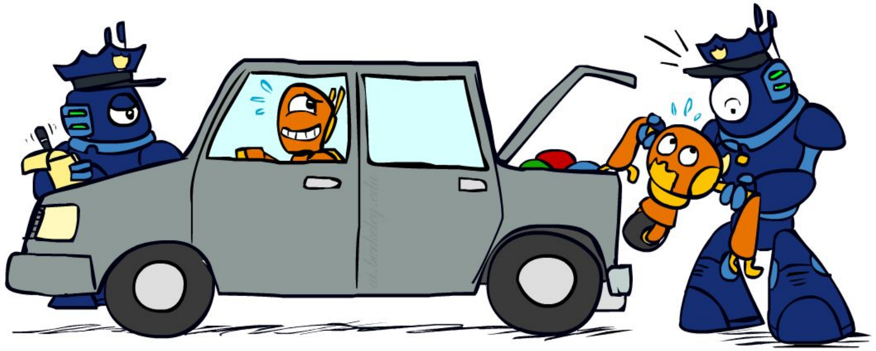
Delete from the tail!

Consistency of A Single Arc

- An arc X $\longrightarrow$ Y is consistent iff for *every* x in the tail there is *some allowed* y in the head which could be assigned without violating a constraint





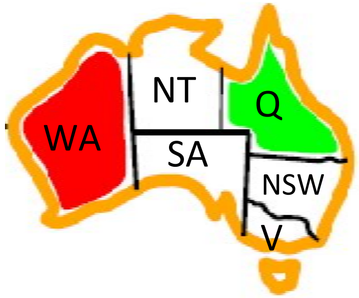


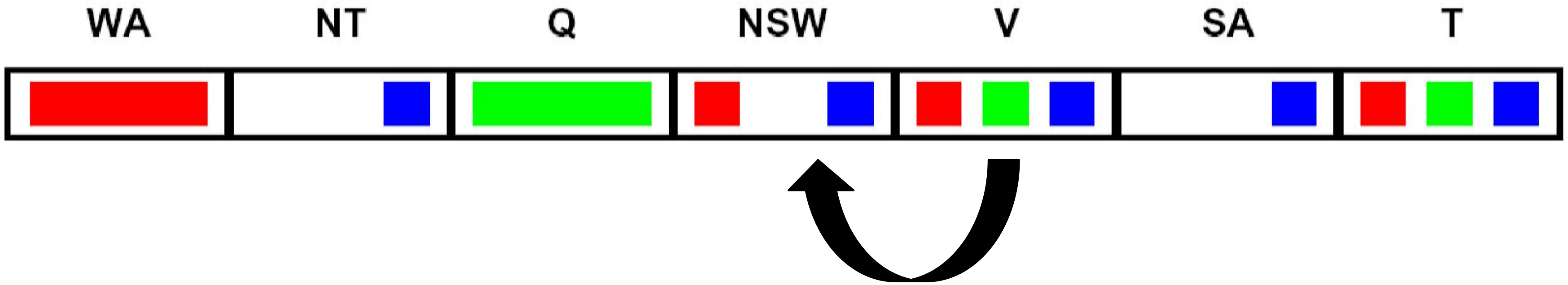
Delete from the tail!

- Forward checking: Enforcing consistency of arcs pointing to each new assignment

Arc Consistency of an Entire CSP

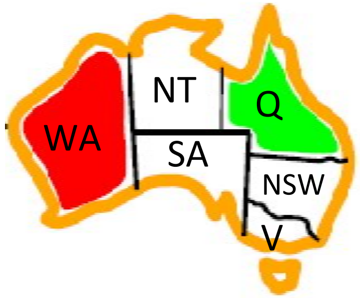
- A simple form of propagation makes sure all arcs are consistent:

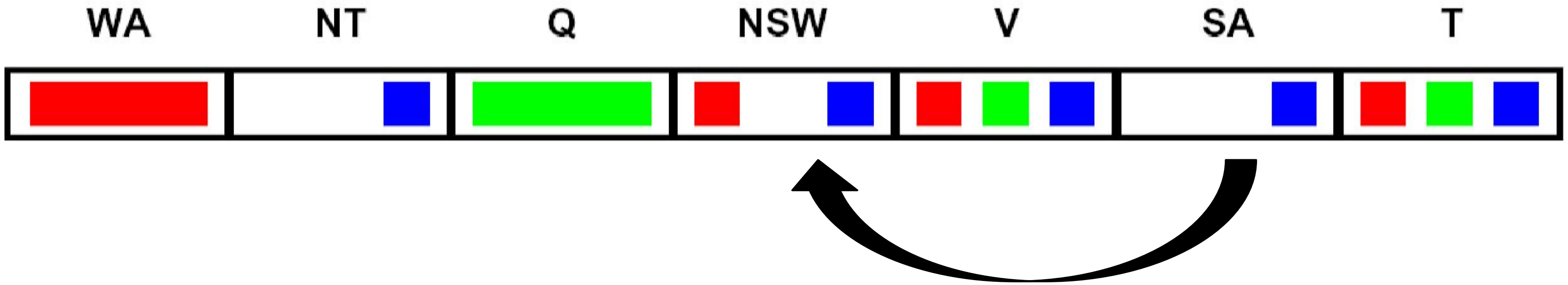




Arc Consistency of an Entire CSP

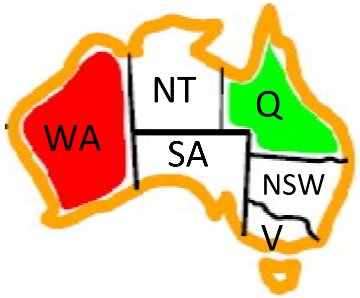
- A simple form of propagation makes sure all arcs are consistent:

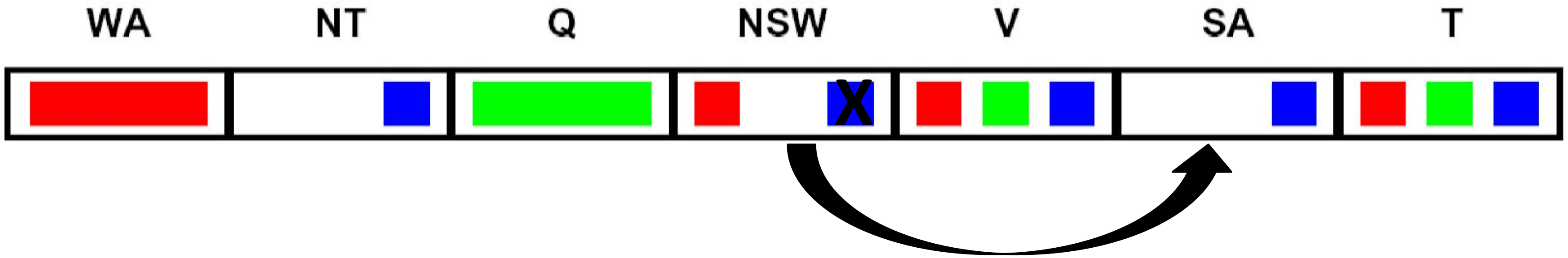




Arc Consistency of an Entire CSP

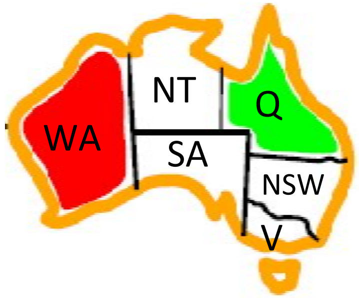
- A simple form of propagation makes sure all arcs are consistent:

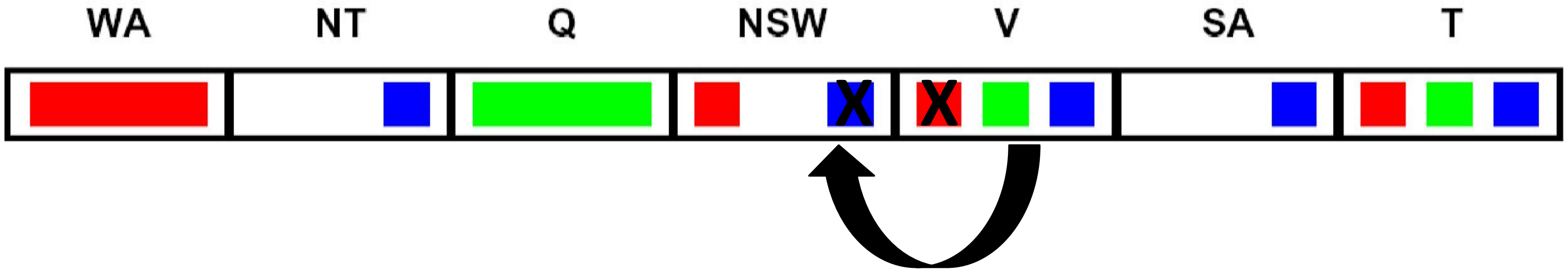




Arc Consistency of an Entire CSP

- A simple form of propagation makes sure all arcs are consistent:

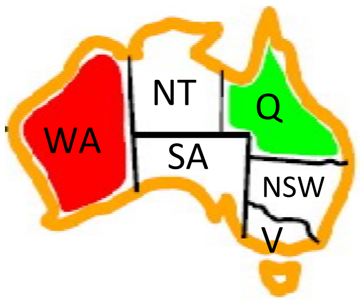


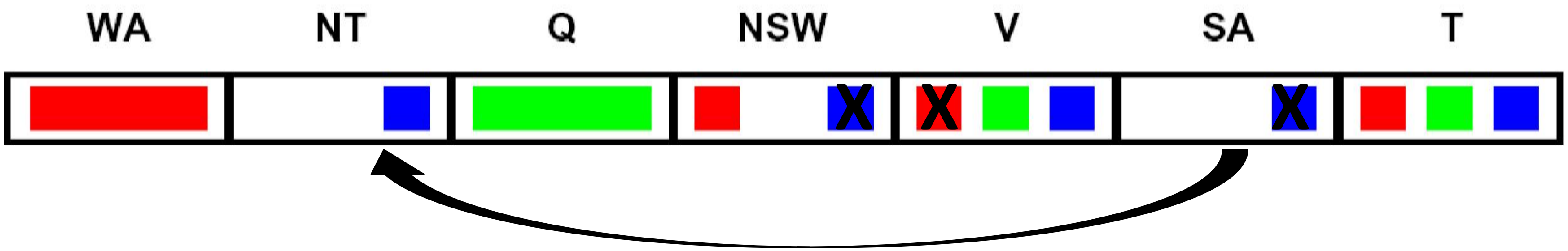


- Important: If X loses a value, neighbors of X need to be rechecked!

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure all arcs are consistent:





- Important: If X loses a value, neighbors of X need to be rechecked!
- Arc consistency detects failure earlier than forward checking
- What's the downside of enforcing arc consistency?

Enforcing Arc Consistency in a CSP

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

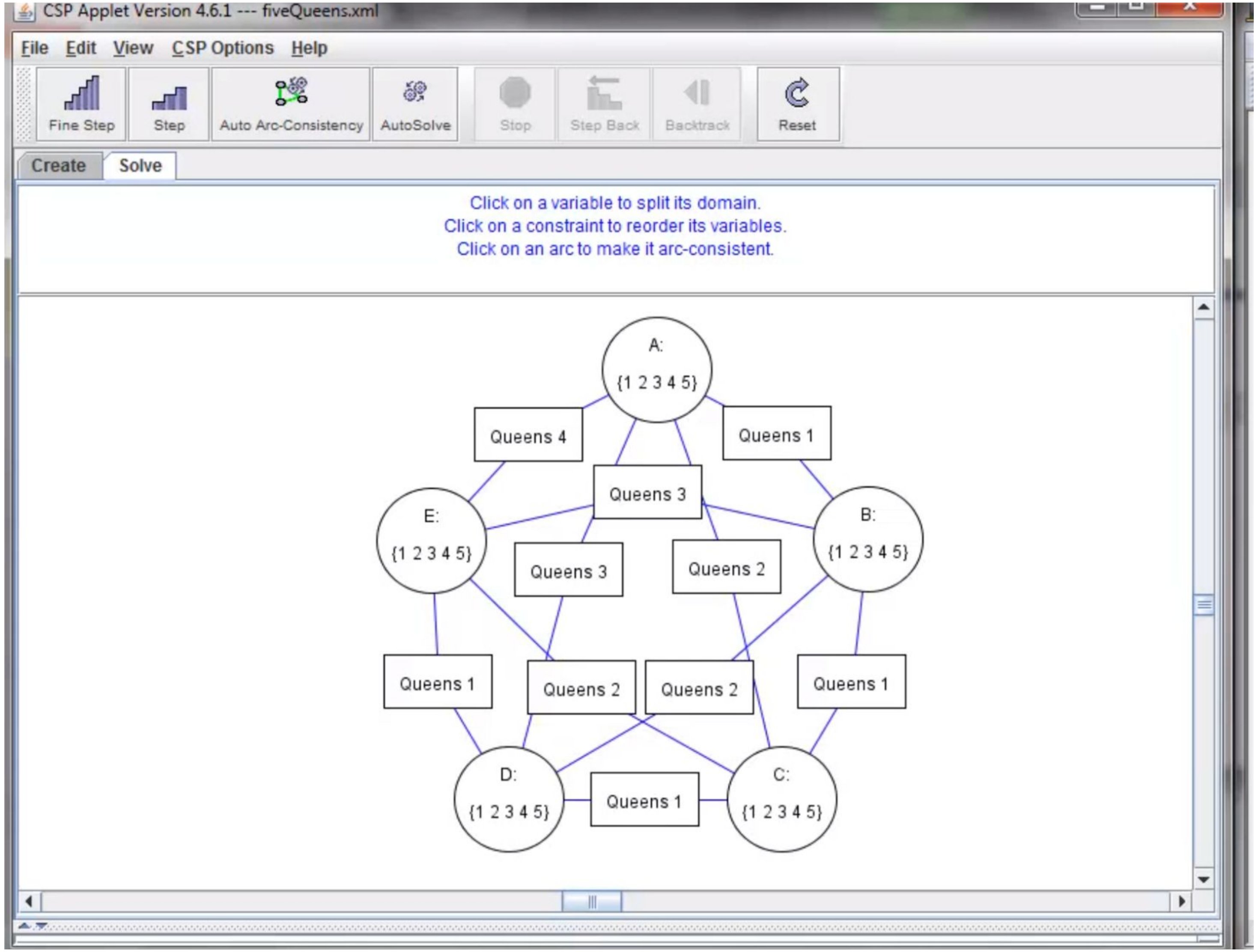
  while $queue$ is not empty **do**
    $(X_i,\ X_j) \leftarrow$ REMOVE-FIRST($queue$)
    if REMOVE-INCONSISTENT-VALUES($X_i,\ X_j$) **then**
      for each X_k **in** NEIGHBORS[X_i] **do**
        add $(X_k,\ X_i)$ to $queue$

---

function REMOVE-INCONSISTENT-VALUES($X_i,\ X_j$) **returns** true iff succeeds
  $removed \leftarrow false$
  for each x **in** DOMAIN[X_i] **do**
    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 $removed$

- Runtime: $O(n^2d^3)$

Demo Arc Consistency – CSP Applet – aispace.org

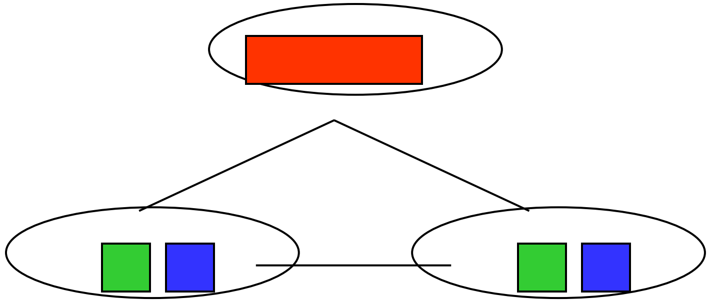


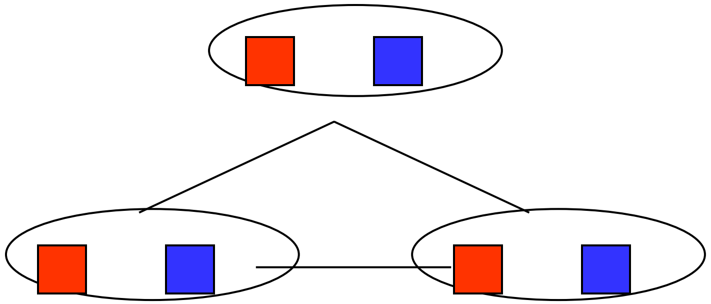
	`1	2	3	4	5
A					
B					
C					
D					
E					

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)

- Arc consistency still runs inside a backtracking search!



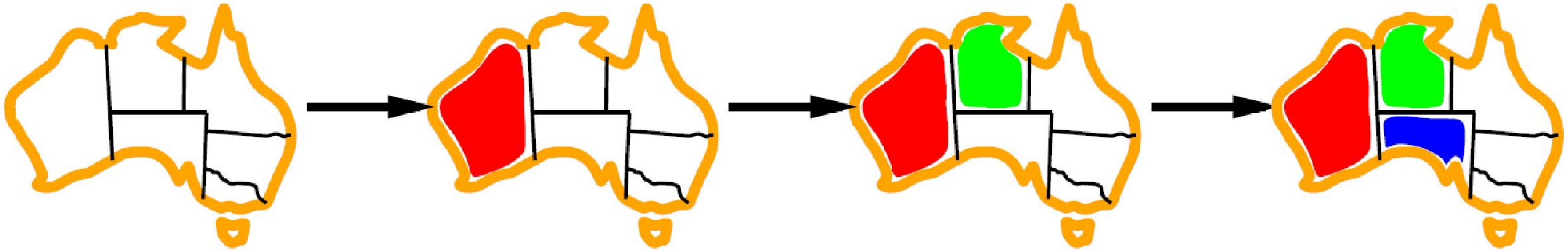


What went wrong here?

Ordering

Ordering: Minimum Remaining Values

- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain

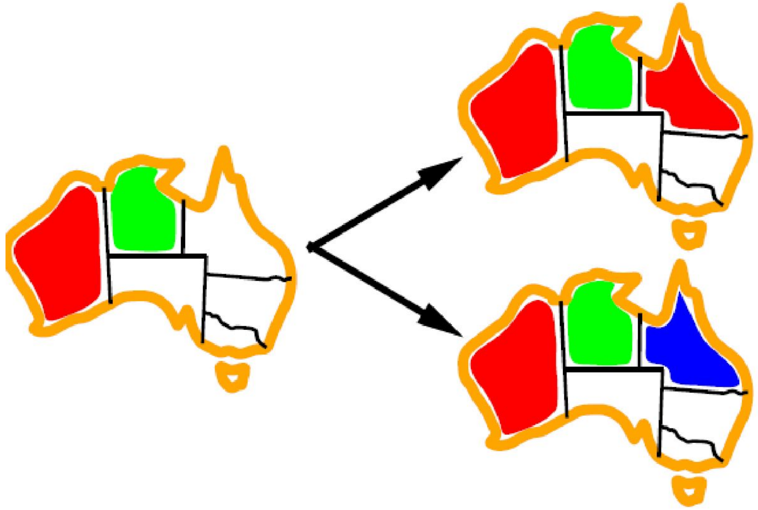


- Also called “most constrained variable”
- “Fail-fast” ordering

Ordering: Least Constraining Value

Value Ordering: Least Constraining Value

- Given a choice of variable, choose the *least constraining value, i*.e., the one that rules out the fewest values in the remaining variables
- Note that it may take some computation to determine this!

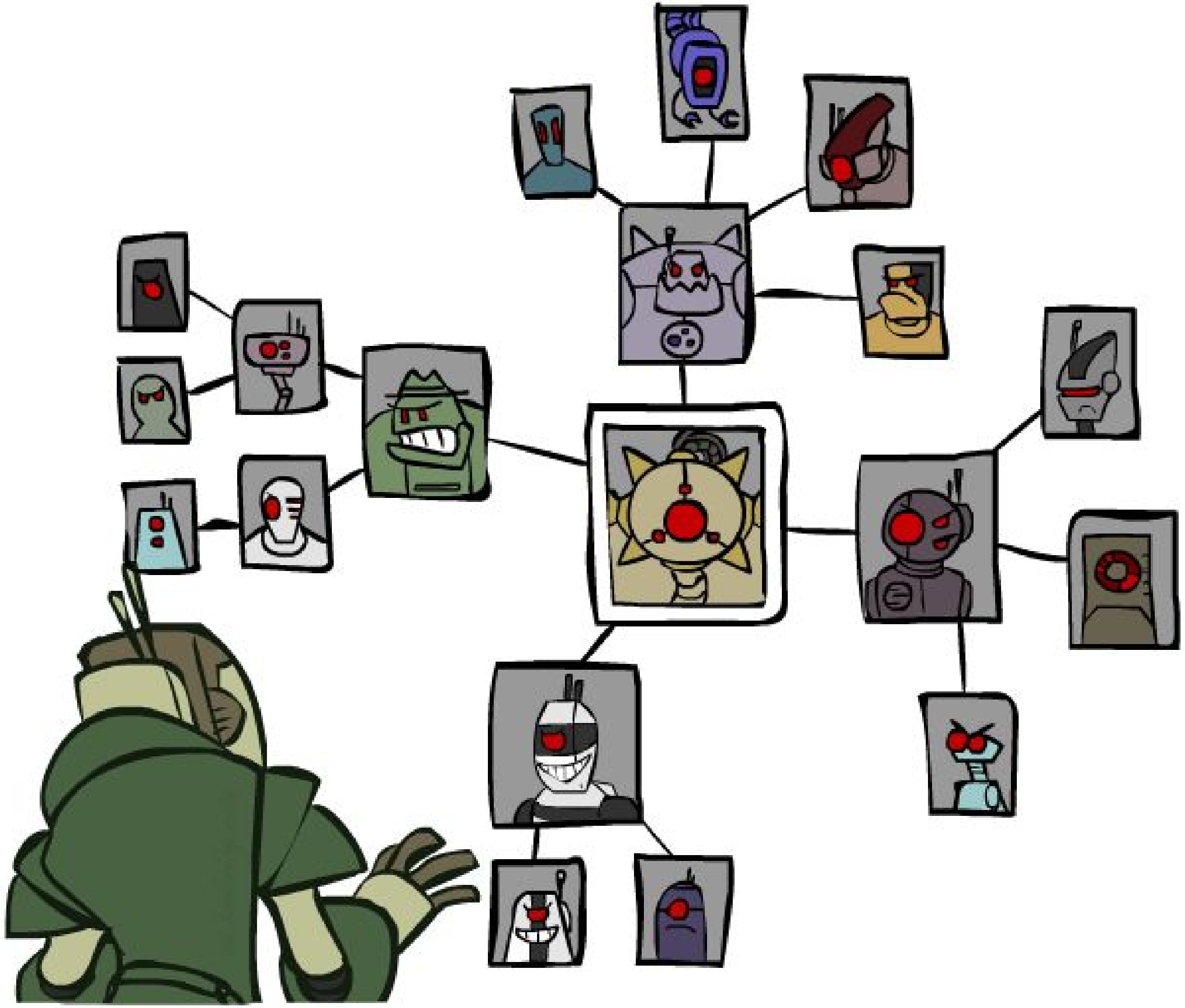


allows 1 value (blue) for SA

allows 0 value for SA

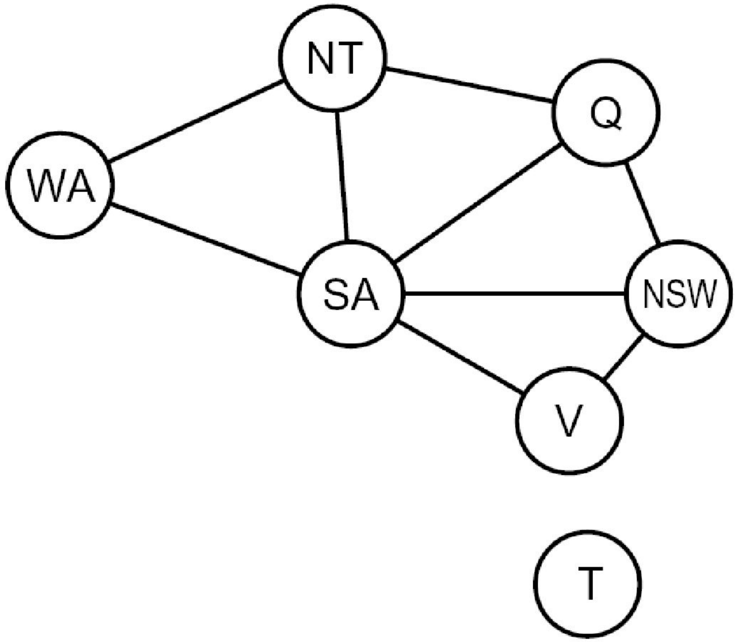
Combining these ordering ideas makes 1000 queens feasible

Structure

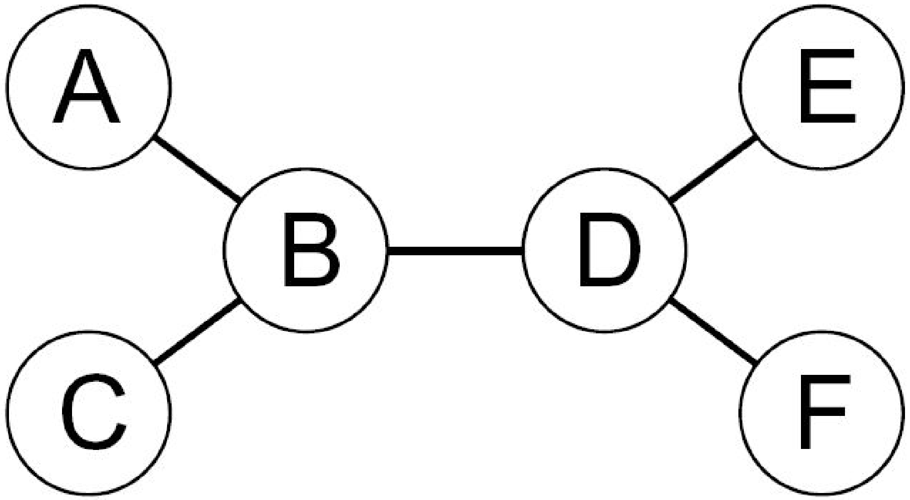


Problem Structure

- Extreme case: independent subproblems
  - Example: Tasmania and mainland do not interact
- Independent subproblems are identifiable as connected components of constraint graph
- Suppose a graph of n variables can be broken into subproblems of only c variables:
  - Worst-case solution cost is $O((n/c)(d^c))$, linear in n
  - E.g., n = 80, d = 2, c =20
  - 2^{80} = 4 billion years at 10 million nodes/sec
  - $(4)(2^{20})$ = 0.4 seconds at 10 million nodes/sec



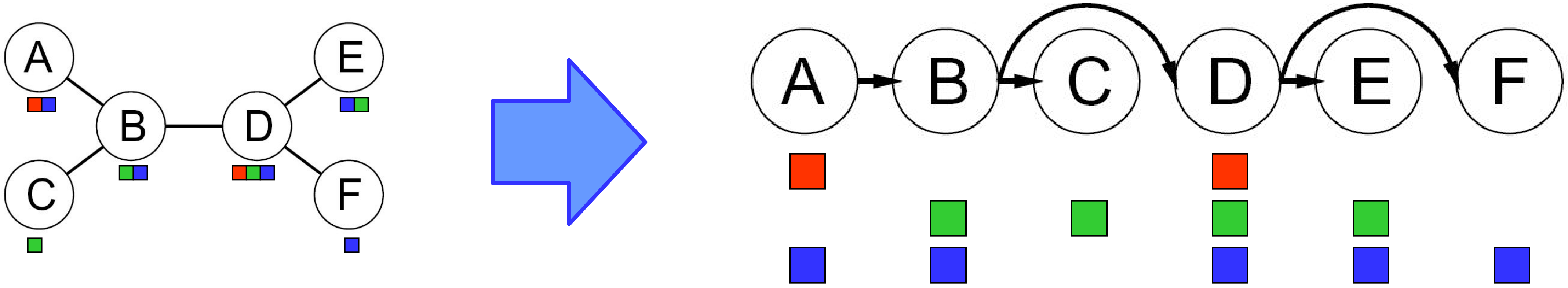
Tree-Structured CSPs



- Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\ d^2)$ time
  - Compare to general CSPs, where worst-case time is $O(d^n)$

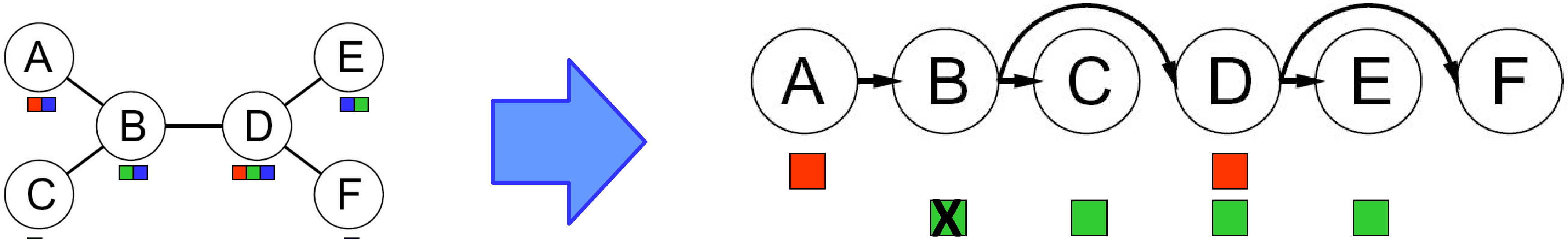
Tree-Structured CSPs

- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



Tree-Structured CSPs

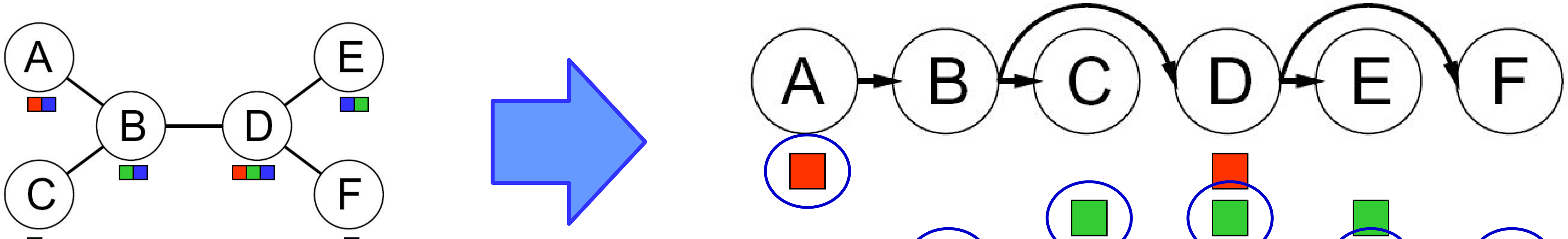
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



  - Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)

Tree-Structured CSPs

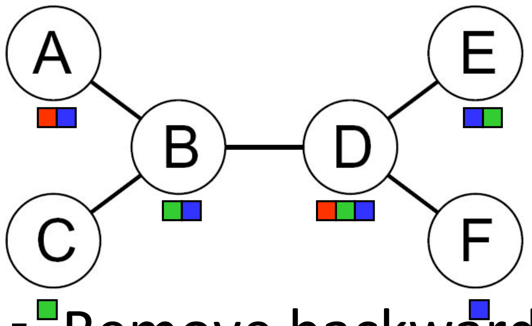
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children

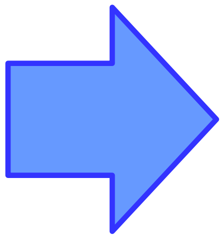


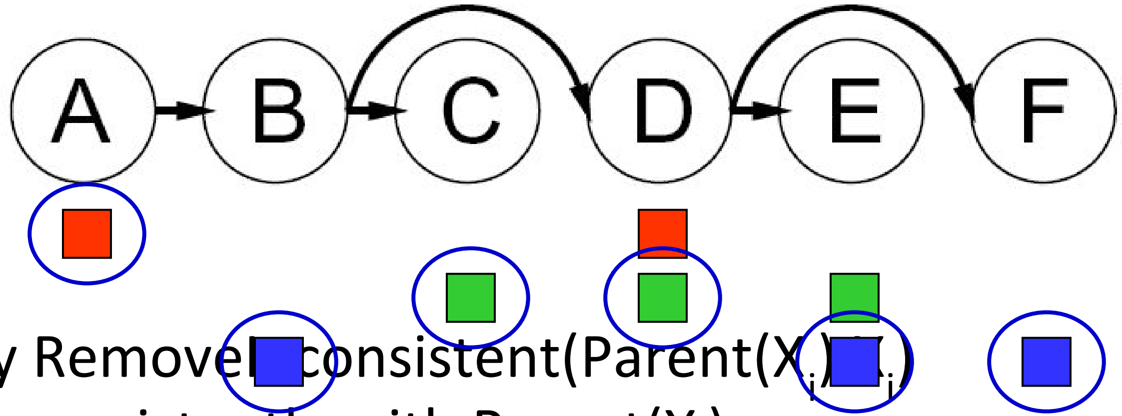
  - Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)
  - Assign forward: For i = 1 : n, assign X_i consistently with Parent(X_i)

Tree-Structured CSPs

- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



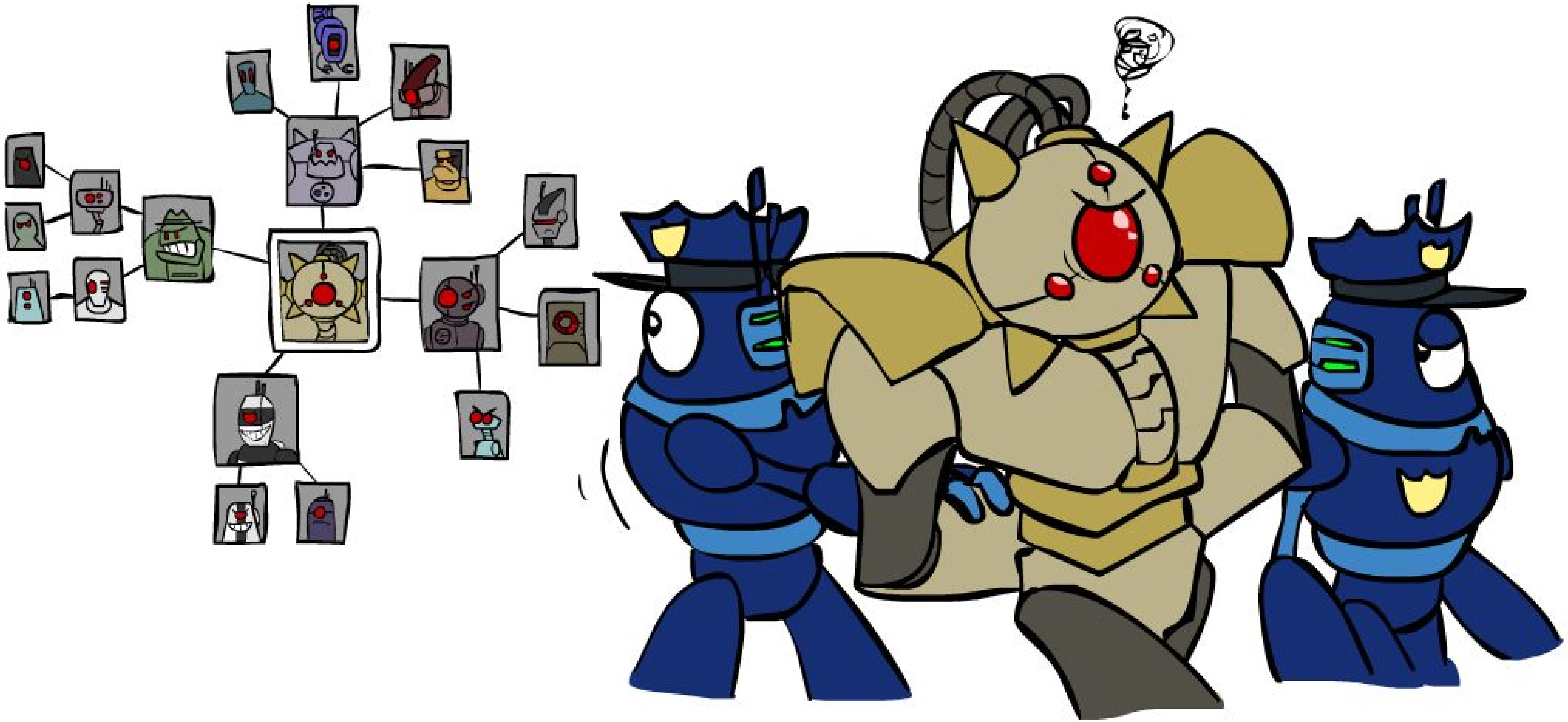




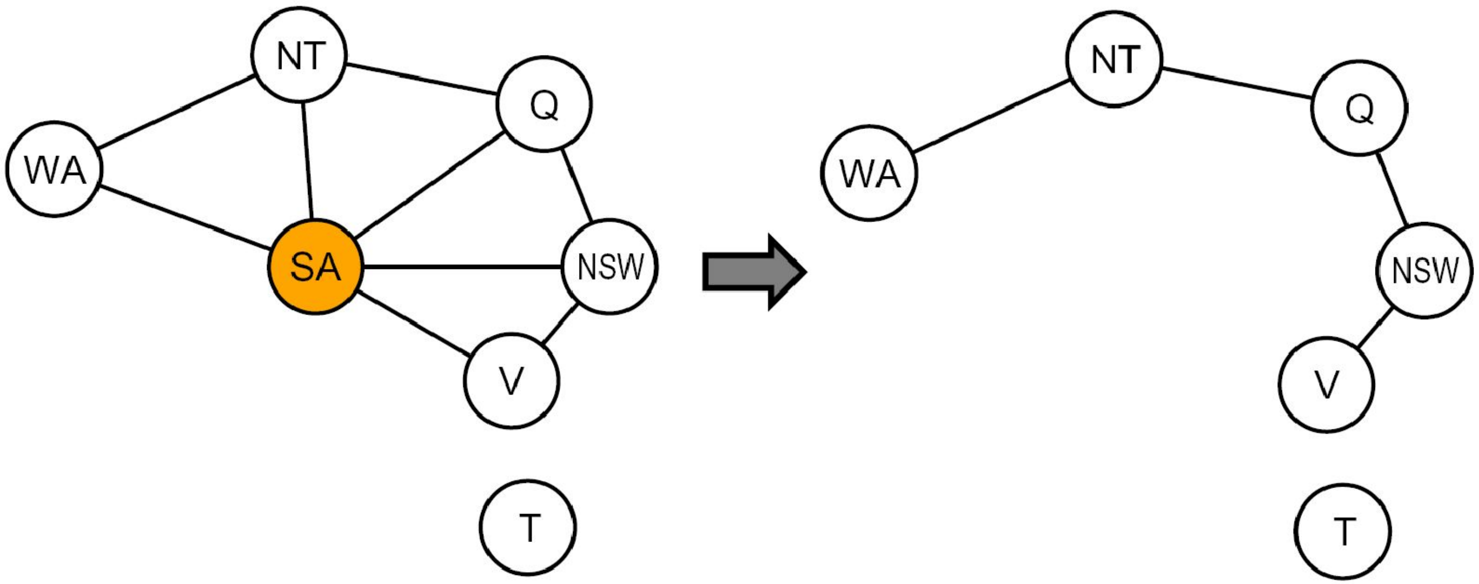
  - Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)
  - Assign forward: For i = 1 : n, assign X_i consistently with Parent(X_i)

- Runtime: O(n d^2)

Improving Structure



Nearly Tree-Structured CSPs



- Conditioning: instantiate a variable, prune its neighbors' domains
- Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree
- Cutset size c gives runtime $O((d^c) (n-c) d^2)$, very fast for small c

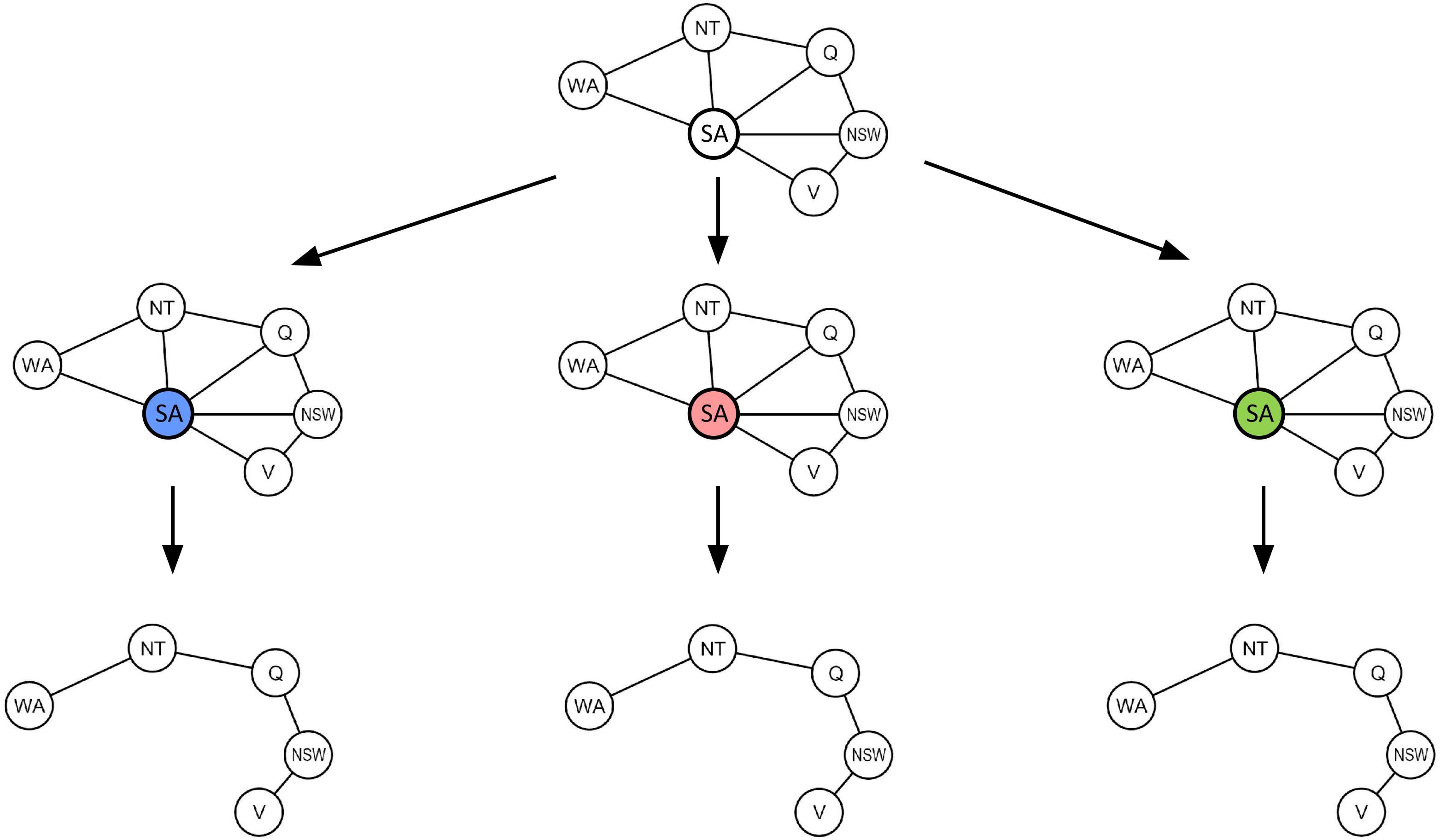
Cutset Conditioning

Choose a cutset

Instantiate the cutset (all possible ways)

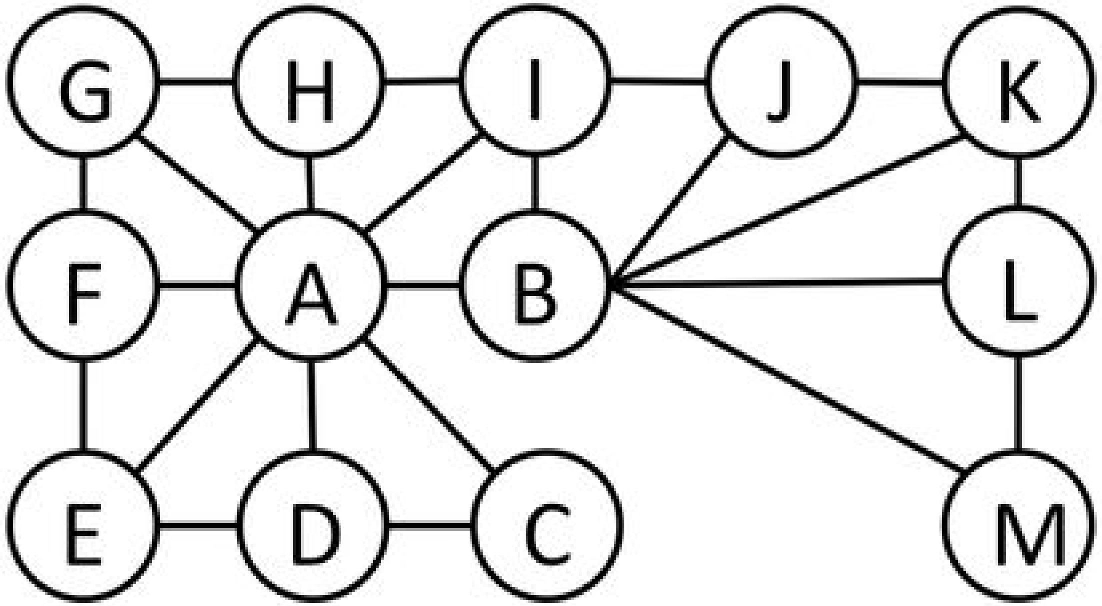
Compute residual CSP for each assignment

Solve the residual CSPs (tree structured)



Cutset

- Find the smallest cutset for the graph below.



A. A
B. B
C. A, B
D. A, B, J
E. I

Summary: CSPs

- CSPs are a special kind of search problem:
  - States are partial assignments
  - Goal test defined by constraints
- Basic solution: backtracking search
- Speed-ups:
  - Filtering
  - Ordering
  - Structure