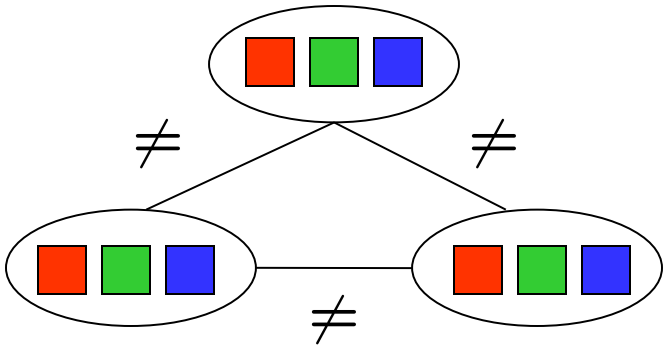
Announcements

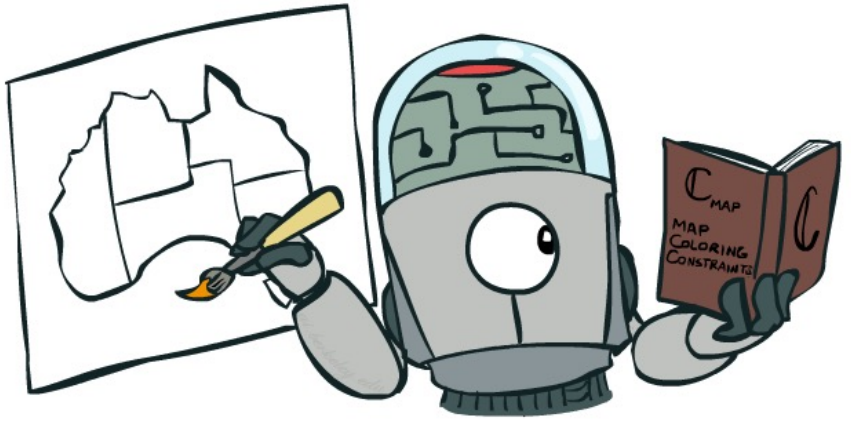
- Project is out, due **December 22**, 23:59
  - Please start early! Reserve sufficient time for iterating your solution.
- HW2 will be release by tomorrow

Recap: CSPs

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

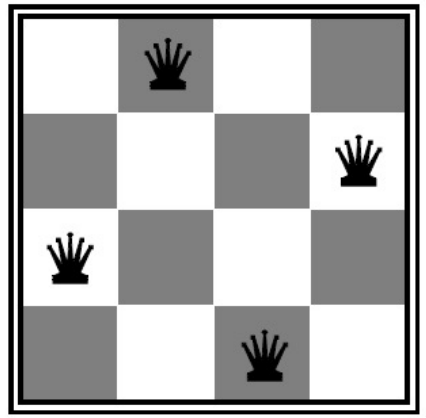


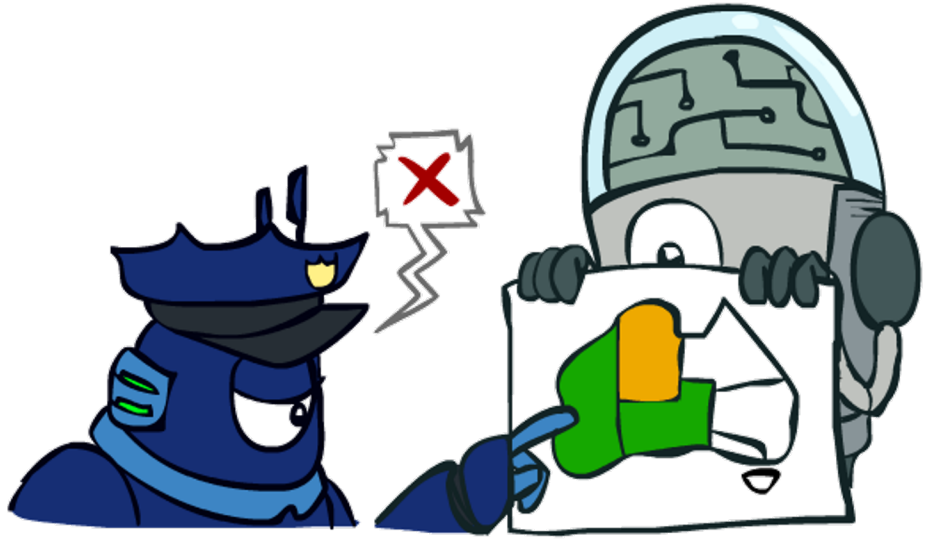
- Goals:
  - Here: identify any solution
  - Also: identify all, identify best, etc.



Recap: Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Depth-first search with two improvements:
  - Idea 1: Fix ordering & one variable at a time
  - Idea 2: Check constraints as you go
- Can solve n-queens for n ≈ 25

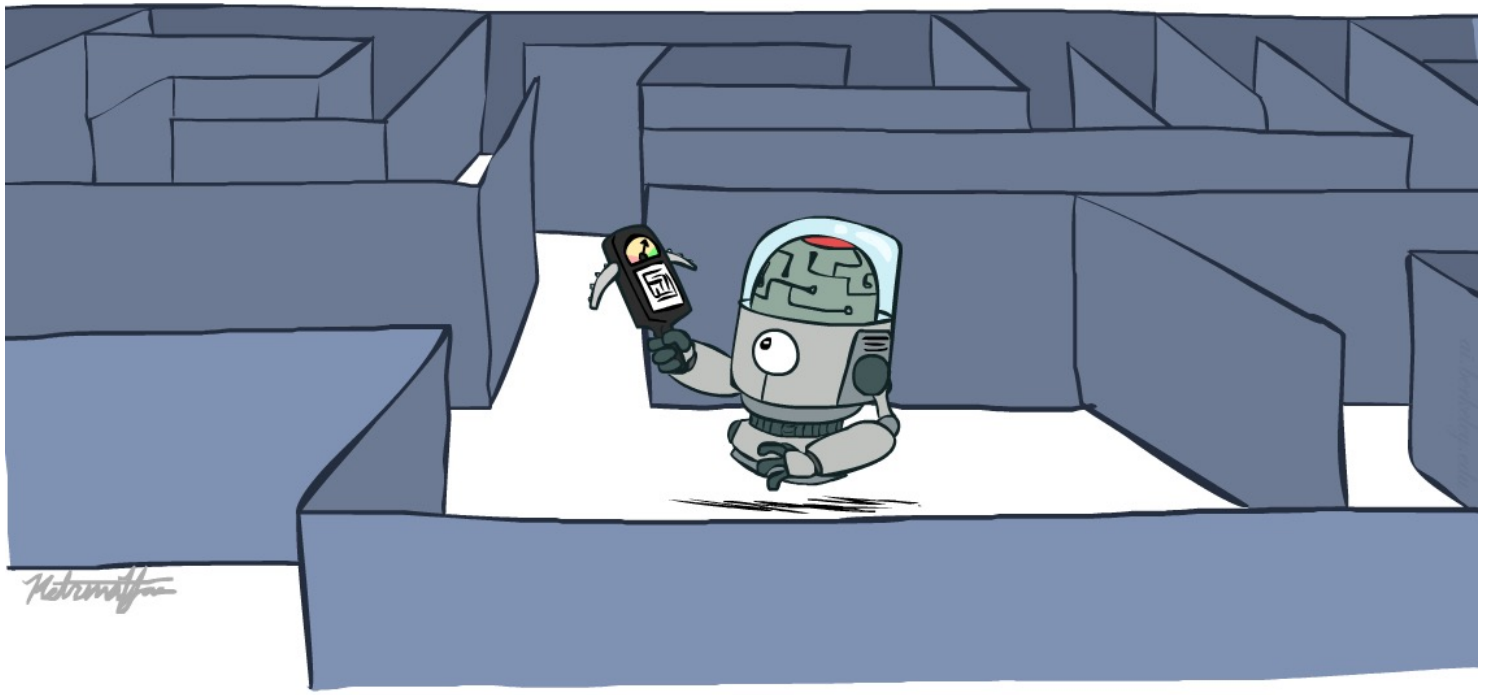




[Demo: coloring -- backtracking]

CS 3317: Artificial Intelligence

Constraint Satisfaction Problems II



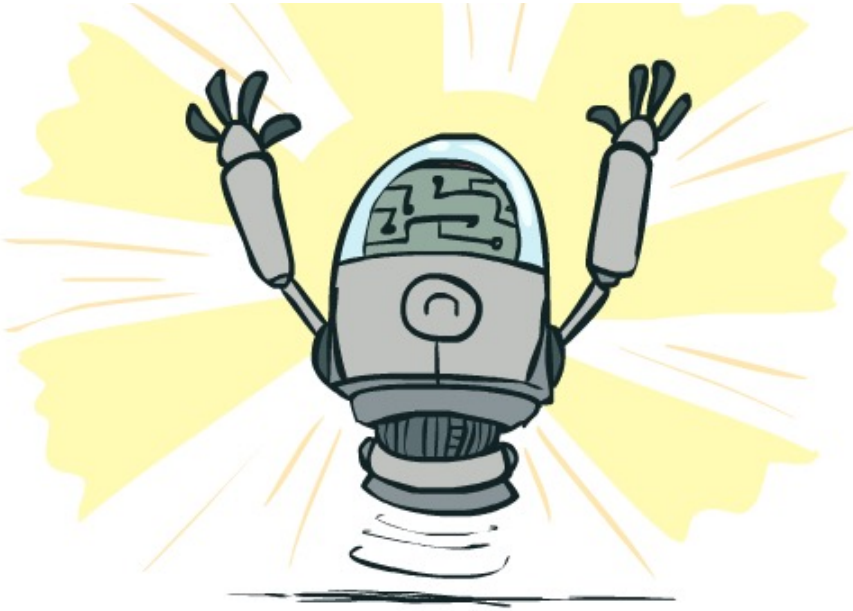
Instructors: **Cai Panpan**

Shanghai Jiao Tong University

(slides adapted from UC Berkeley CS188)

Improving Backtracking

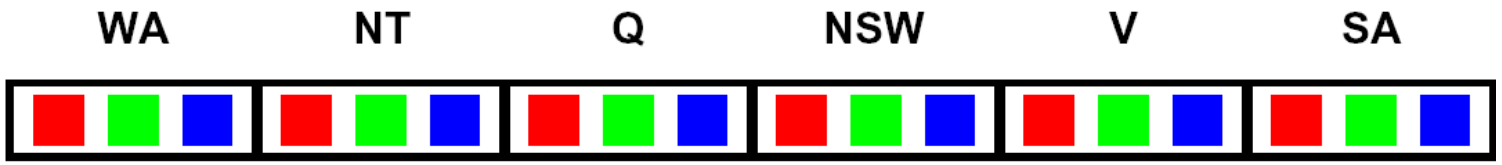
- General-purpose ideas give huge gains in speed
- Ordering:
  - Which variable should be assigned next?
  - In what order should I try the values of variables?
- Filtering: Can we detect inevitable failure early?
- Structure: Can we exploit the problem structure?



Filtering

Filtering: Forward Checking

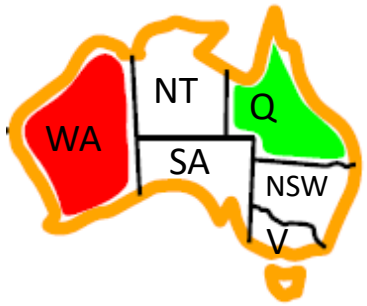
- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment

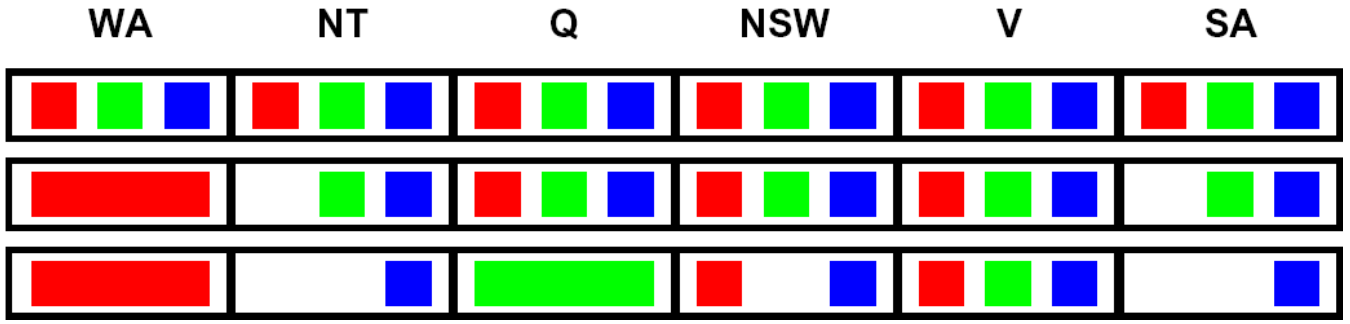


[Demo: coloring -- forward checking]

Filtering: Constraint Propagation

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

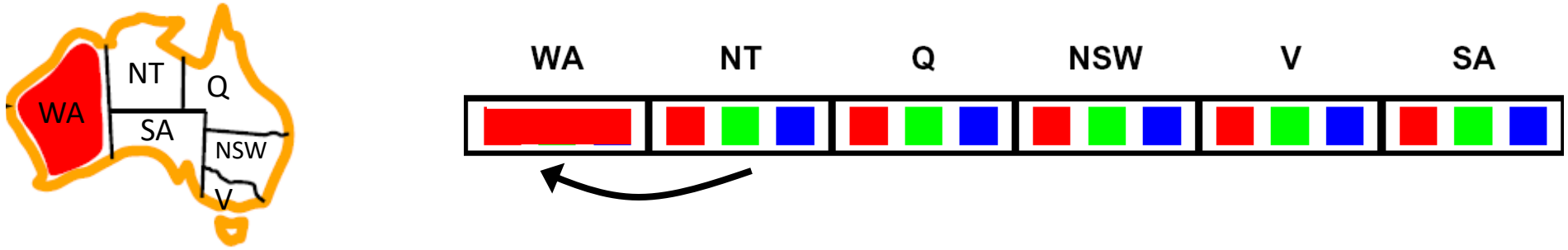




- NT and SA cannot both be blue!
- Why didn’t we detect this yet?
- *Constraint propagation:* reason from constraint to constraint

Consistency of A Single Arc

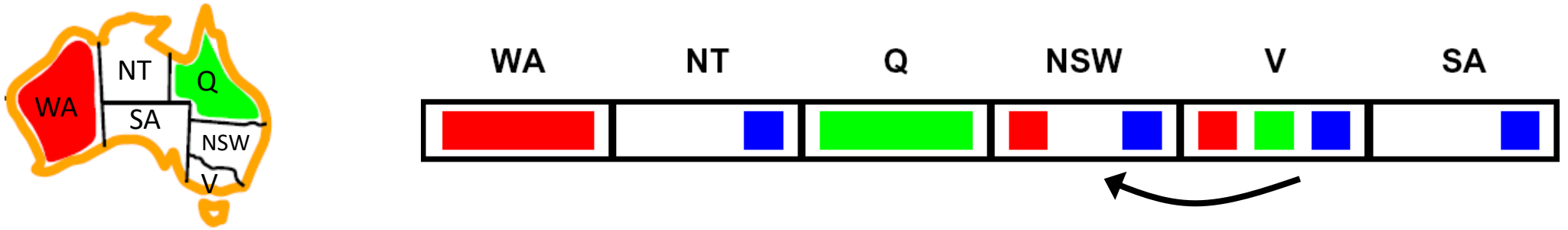
- An arc X → Y is consistent iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint



- Tail = NT, head = WA
  - If NT = blue: we could assign WA = red
  - If NT = green: we could assign WA = red
  - If NT = red: there is no remaining assignment to WA that we can use
  - Deleting NT = red from the tail makes this arc consistent
- Forward checking: Enforcing consistency of arcs pointing to each new assignment

Arc Consistency of an Entire CSP (1/6)

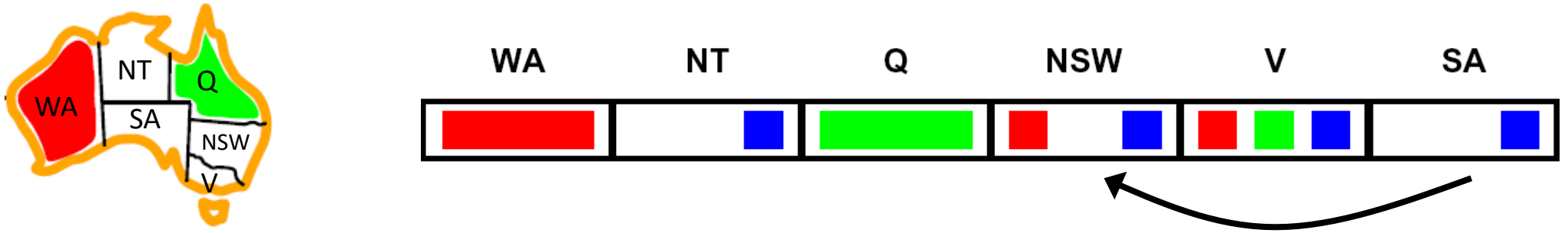
- A simple form of propagation makes sure all arcs are consistent:



- Arc V to NSW is consistent: for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint

Arc Consistency of an Entire CSP (2/6)

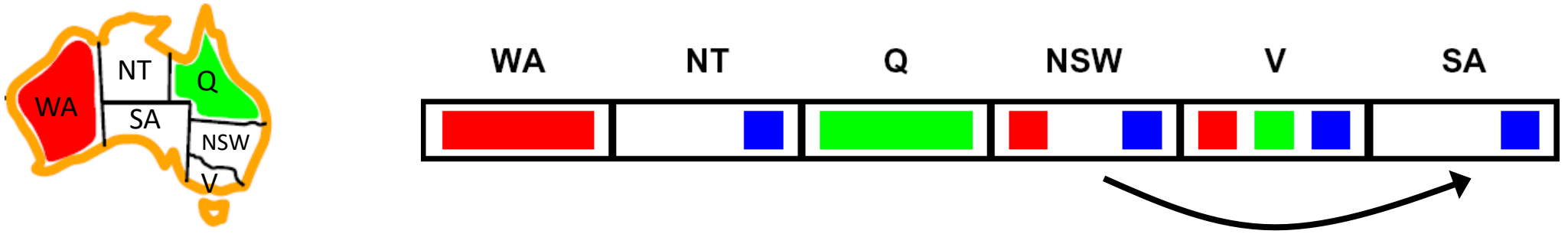
- A simple form of propagation makes sure all arcs are consistent:



- Arc SA to NSW is consistent: for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint

Arc Consistency of an Entire CSP (3/6)

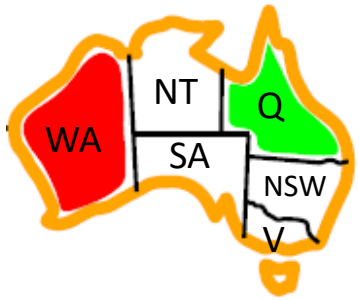
- A simple form of propagation makes sure all arcs are consistent:

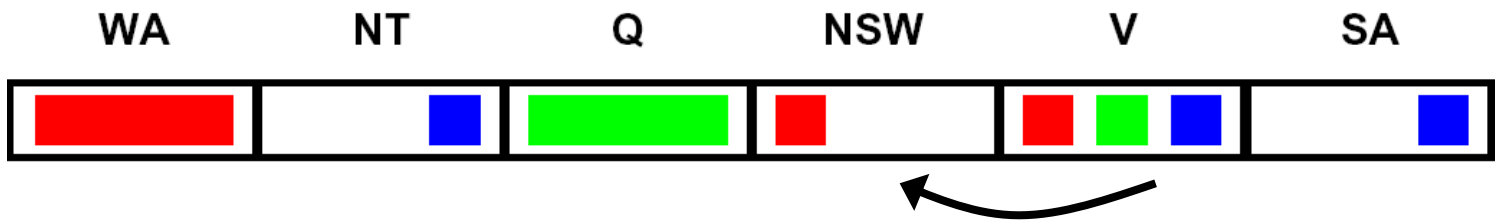


- Arc NSW to SA is not consistent: if we assign NSW = blue, there is no valid assignment left for SA
- To make this arc consistent, we delete NSW = blue (deleting from the tail)

Arc Consistency of an Entire CSP (4/6)

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

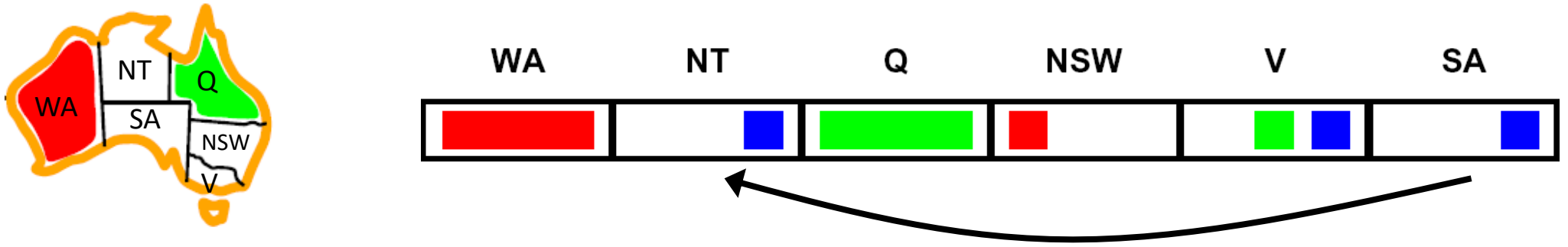




- Remember that arc V to NSW was consistent, when NSW had red and blue in its domain
- After removing blue from NSW, this arc might not be consistent anymore! We need to recheck this arc.
- Important: *If X loses a value, neighbors of X need to be rechecked!*

Arc Consistency of an Entire CSP (5/6)

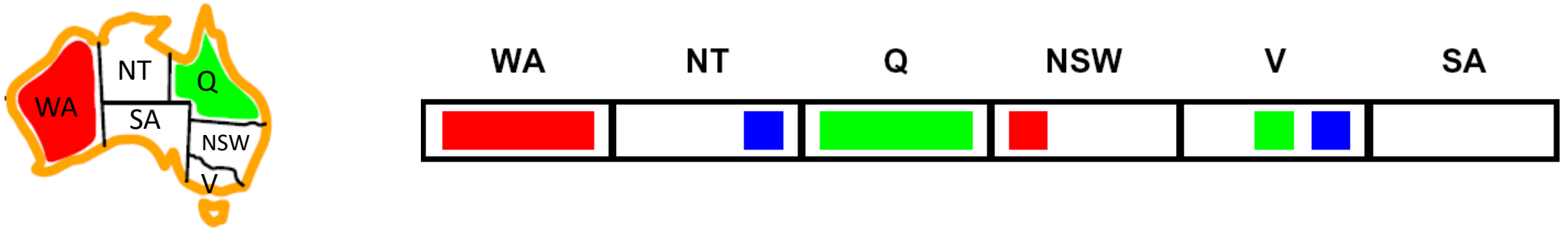
- A simple form of propagation makes sure all arcs are consistent:



- Arc SA to NT is inconsistent. We make it consistent by deleting from the tail (SA = blue).

Arc Consistency of an Entire CSP (6/6)

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



- SA has an empty domain, so we detect failure. There is no way to solve this CSP with WA = red and Q = green, so we backtrack.
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment

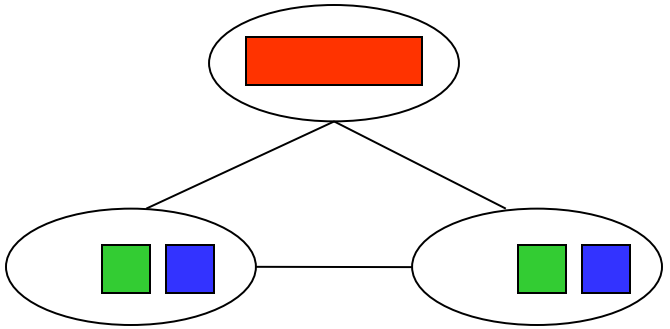
Enforcing Arc Consistency in a CSP

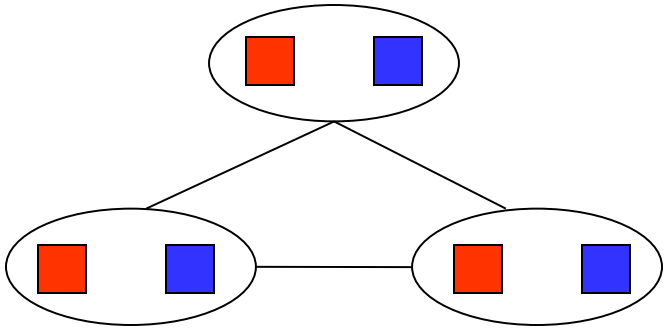
```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X_1, X_2, ..., X_n}
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← 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 ← 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 ↔ X_j
      then delete x from DOMAIN[X_i]; removed ← true
  return removed
```

- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- … but detecting all possible future problems is NP-hard – why?

Arc Consistency Limitations

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



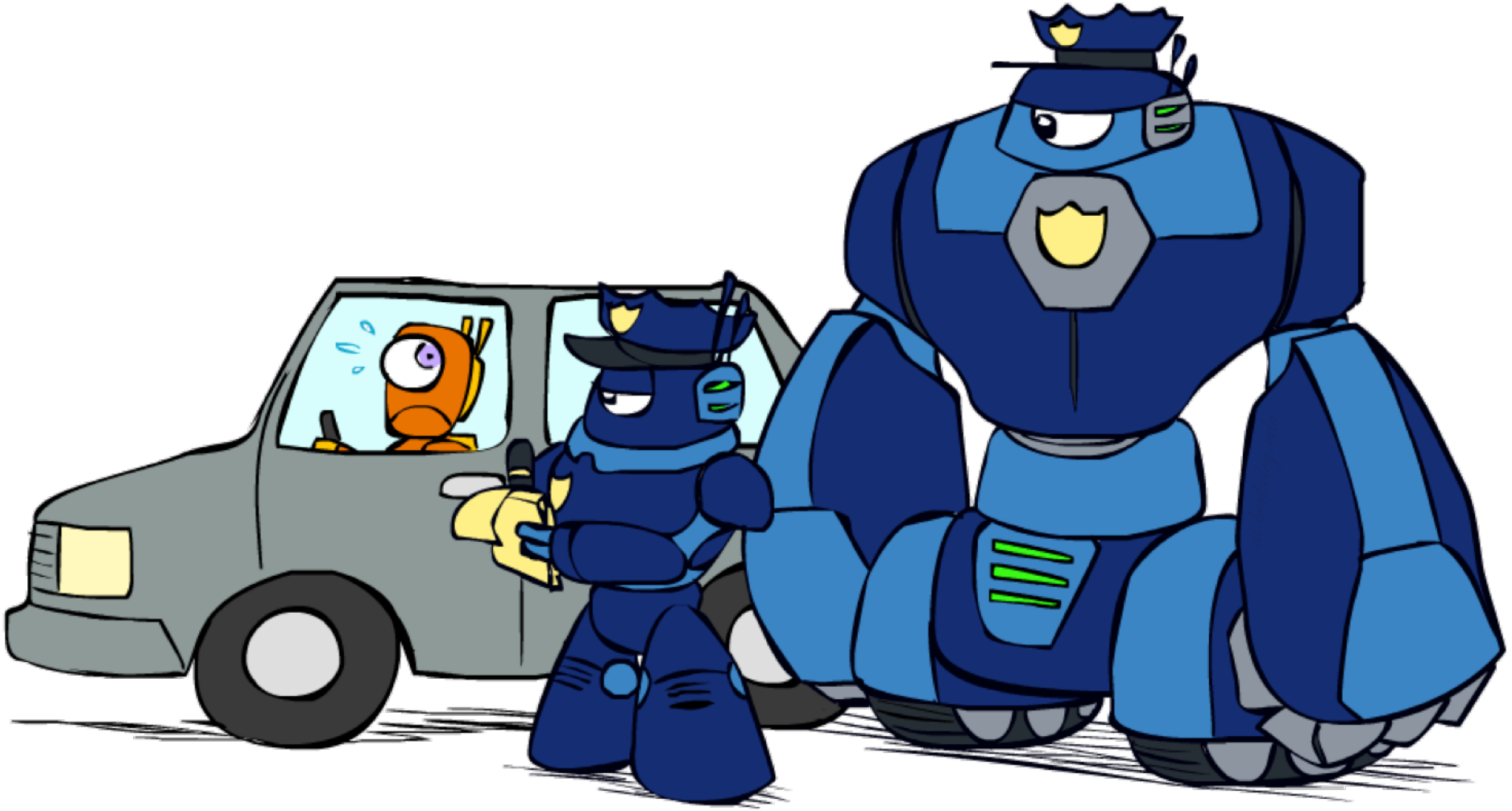


What went wrong here?

- Arc consistency still runs inside a backtracking search!

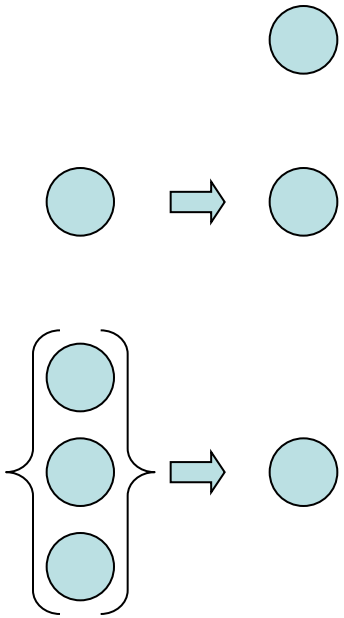
[Demo: coloring -- forward checking]
[Demo: coloring -- arc consistency]

K-Consistency

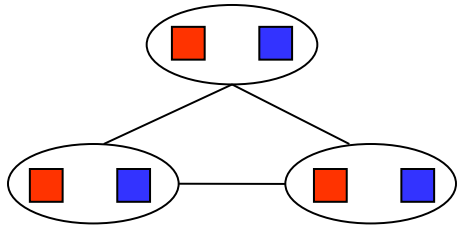


K-Consistency

- 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 of them can be extended to the k^{th} node.



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



Strong K-Consistency

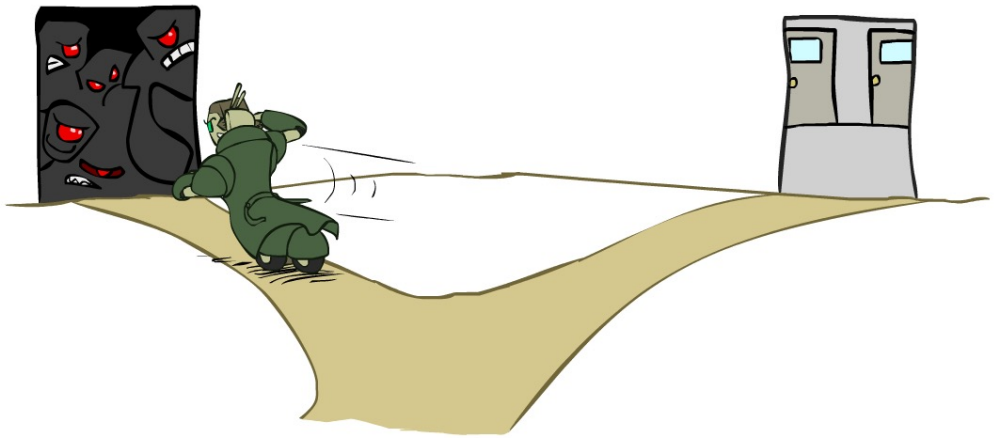
- Strong k-consistency: also k-1, k-2, … 1 consistent
- Claim: strong n-consistency means we can solve without backtracking!
- Why?
  - Choose any assignment to any variable
  - Choose a new variable
  - By 2-consistency, there is a choice consistent with the first
  - Choose a new variable
  - By 3-consistency, there is a choice consistent with the first 2
  - …
- Lots of middle ground between arc consistency and n-consistency! (e.g. k=3, called path consistency)

Ordering

Ordering: Minimum Remaining Values

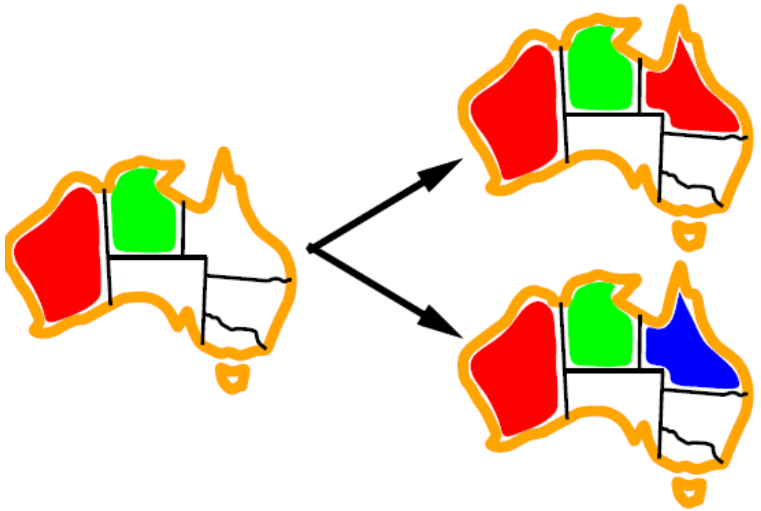
- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal values remained in its domain

- Why min rather than max?
- "Fail-fast" ordering
- Also called "most constrained variable"



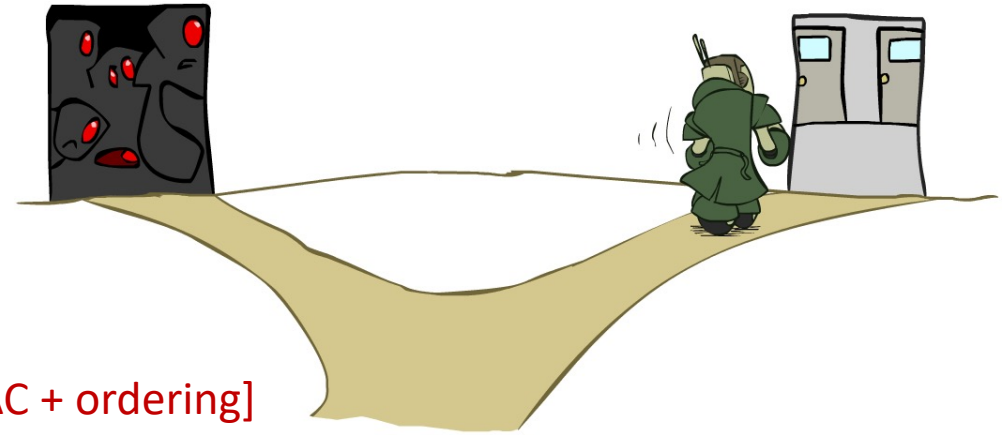
Ordering: Least Constraining Value

- 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! (E.g., rerunning filtering)



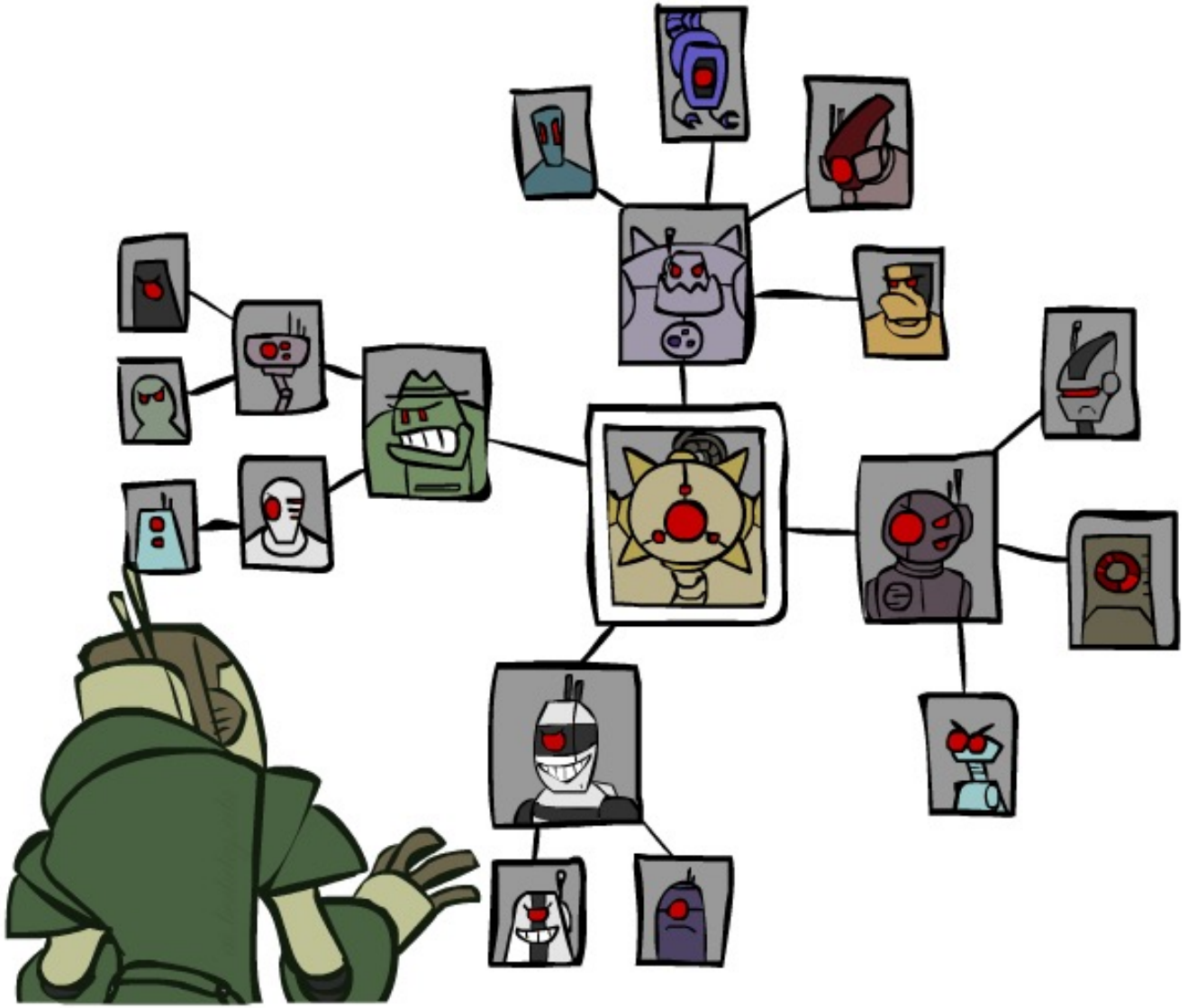
- Why least rather than most?

- Combining these ordering ideas makes 1000 queens feasible



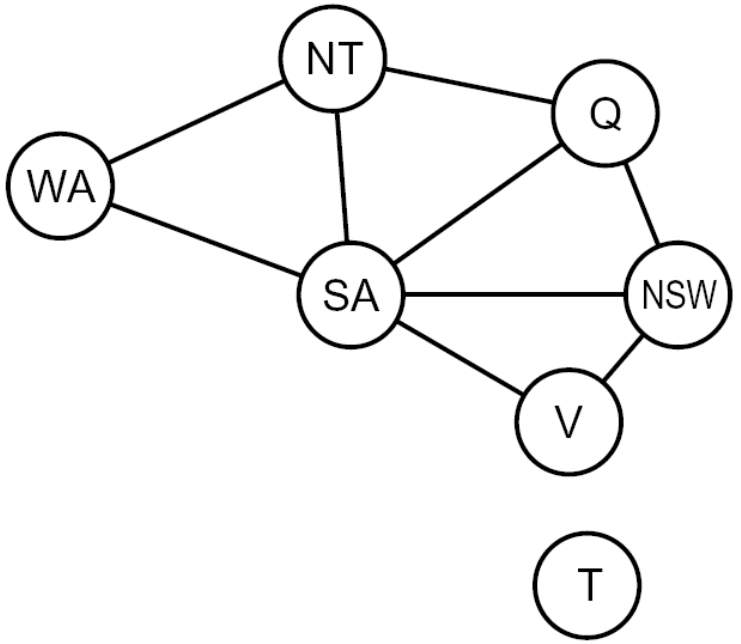
[Demo: coloring – backtracking + AC + ordering]

Structure

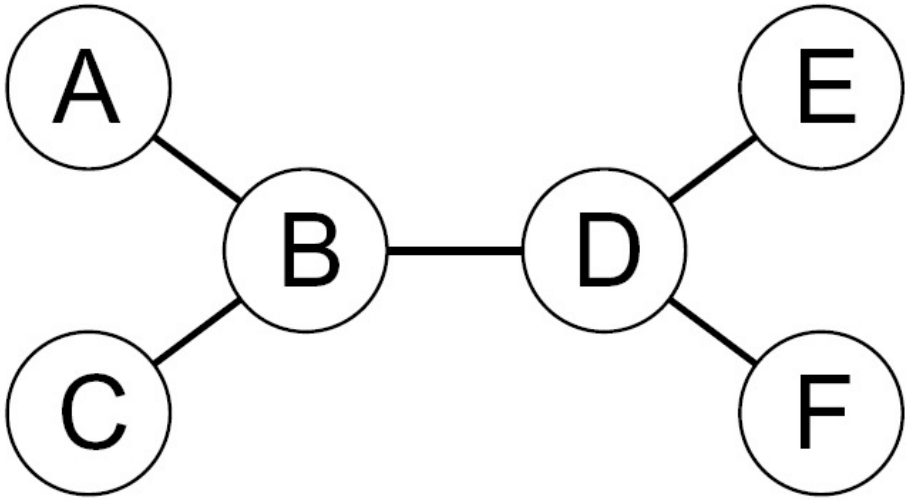


Problem Structure

- 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

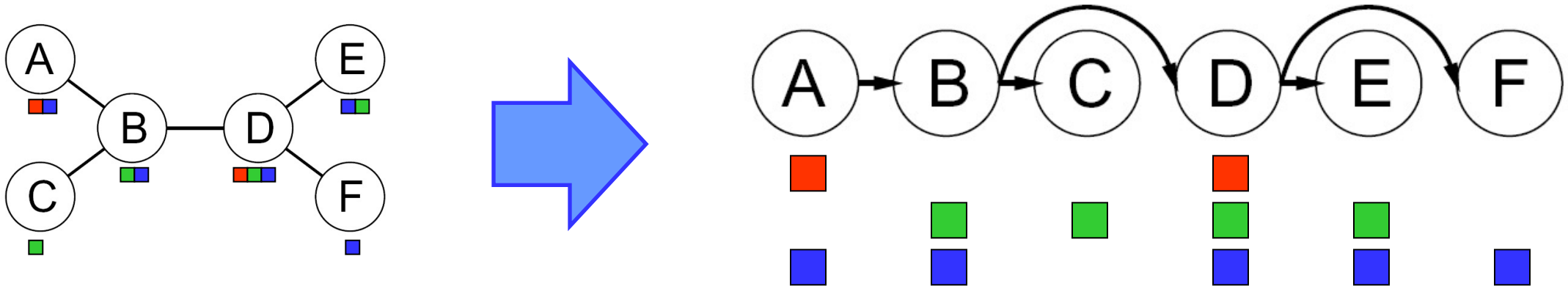


- Theorem: if the constraint graph has no loops (tree), the CSP can be solved in $O(n\ d^2)$ time
  - Compare to general CSPs, where worst-case time is $O(d^n)$

- This property also applies to probabilistic reasoning (later): an example of the relation between syntactic restrictions and the complexity of reasoning

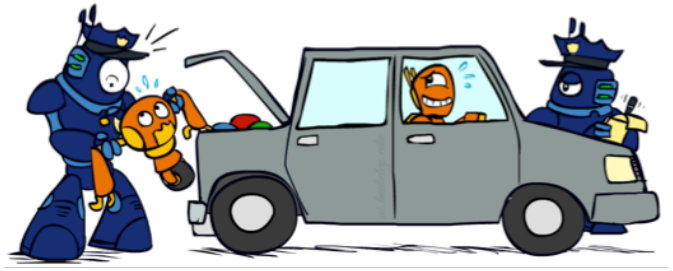
Tree-Structured CSPs

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

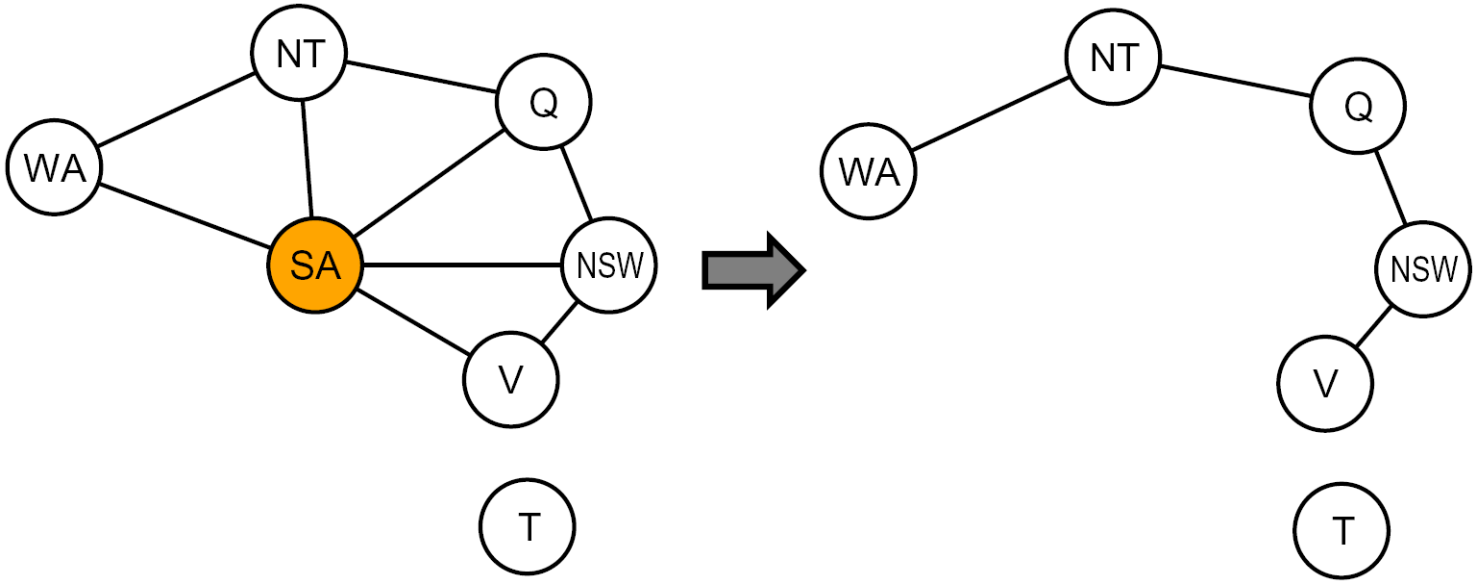


  - Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)
  - Assign forward: For i = 1 : n, assign X_i consistently with Parent(X_i)

- Runtime: $O(n\ d^2)$ (why?)



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

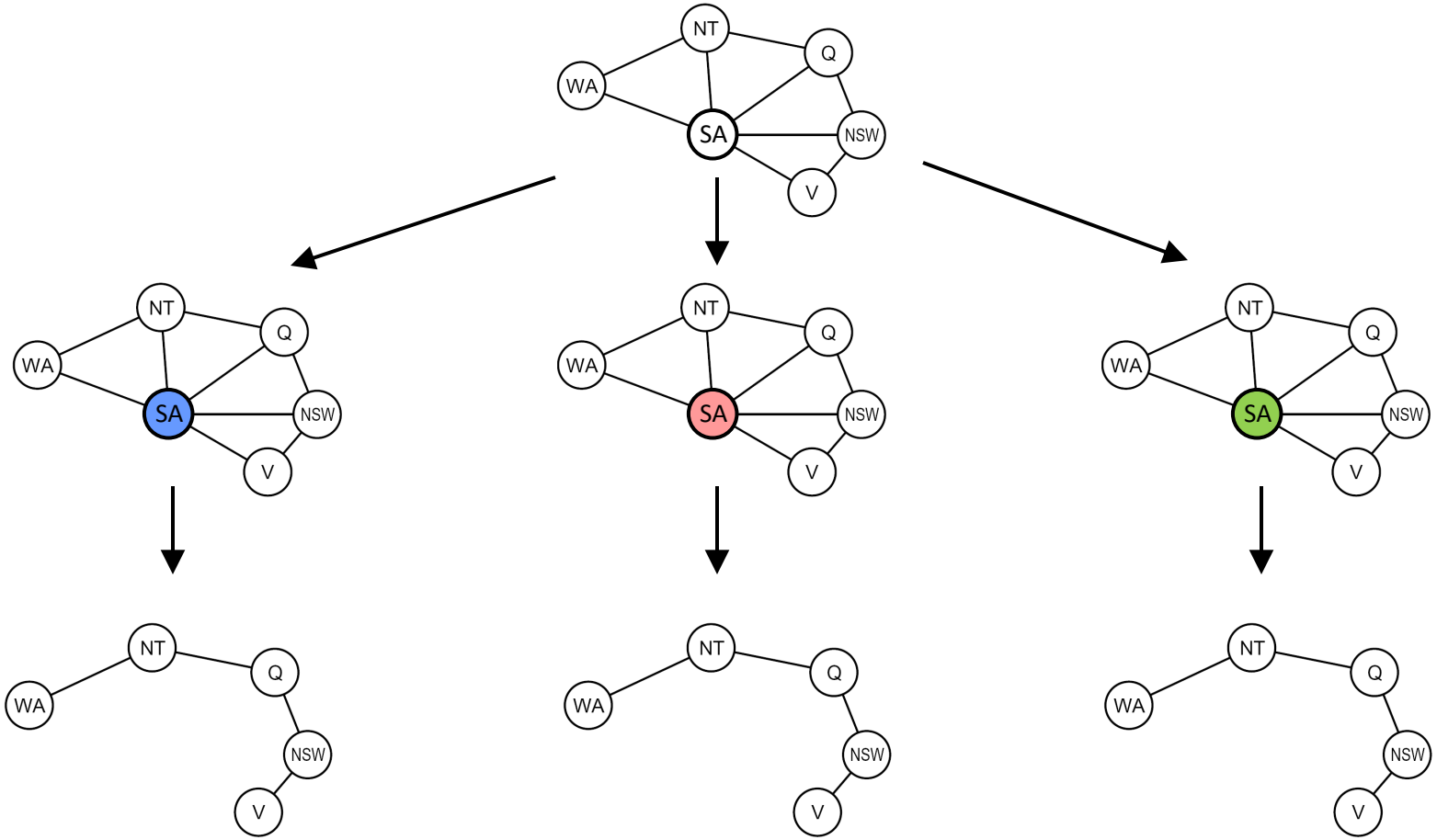
Cutset Conditioning

Choose a cutset

Instantiate the cutset (all possible ways)

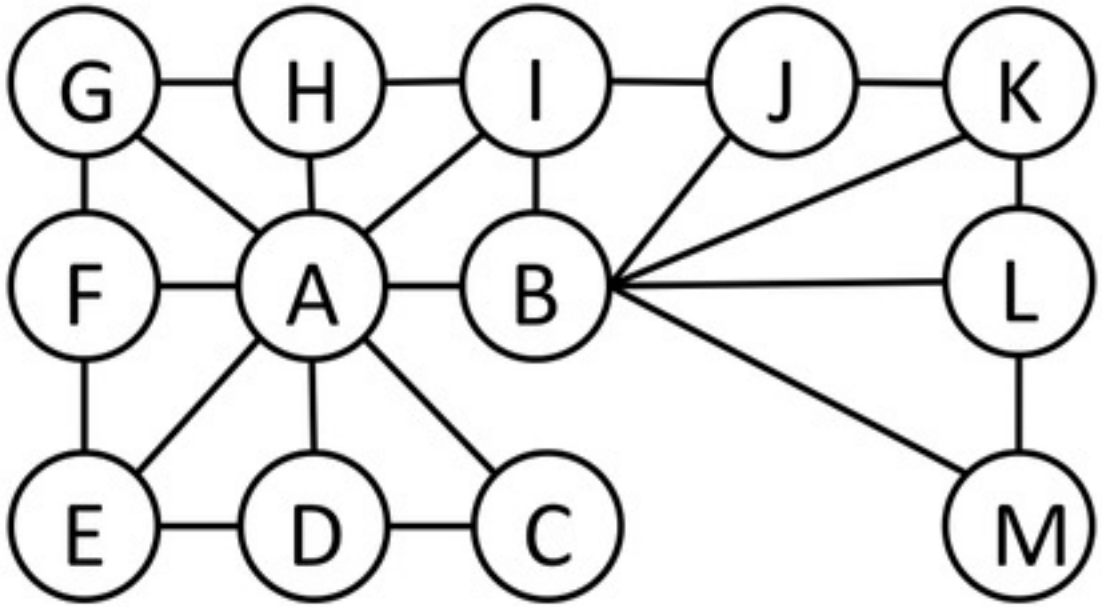
Compute residual CSP for each assignment

Solve the residual CSPs (tree structured)



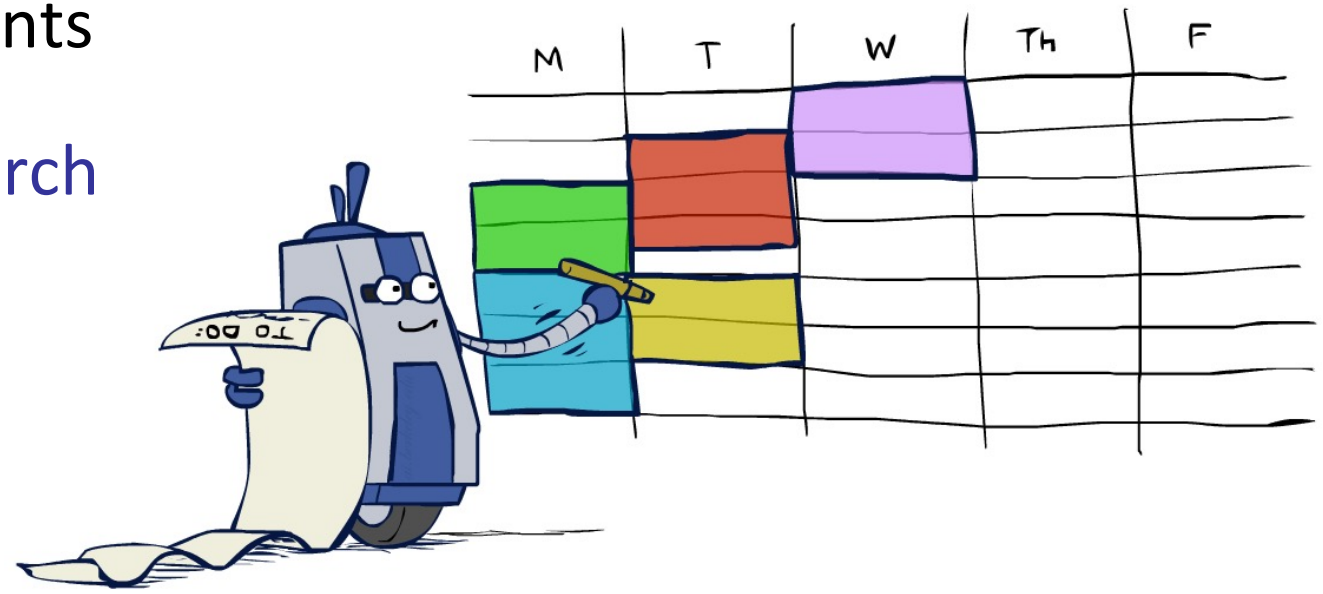
Cutset Quiz

- Find the smallest cutset for the graph below.



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:
  - Ordering
  - Filtering
  - Structure



Next Time: Adversarial Search!