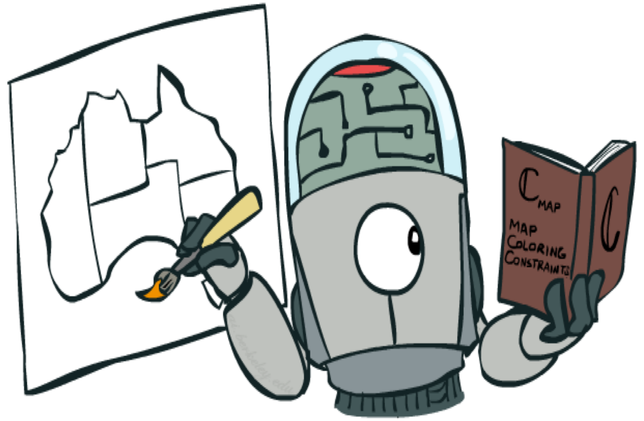
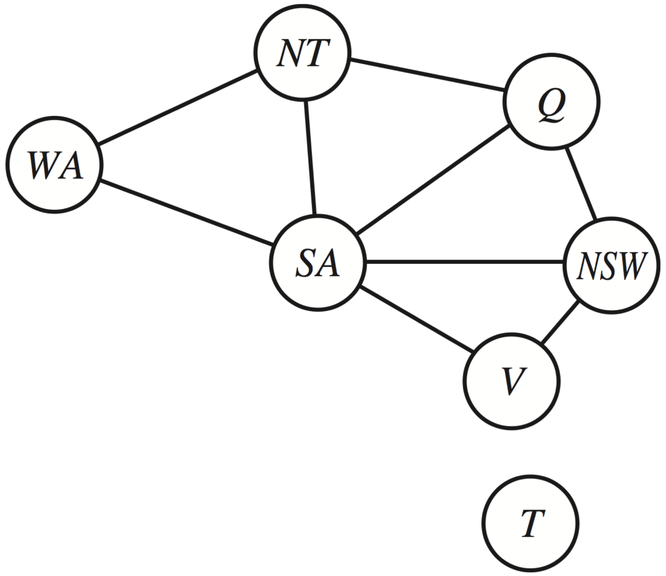
PURDUE CS47100

# INTRODUCTION TO AI

# RECAP: CSP

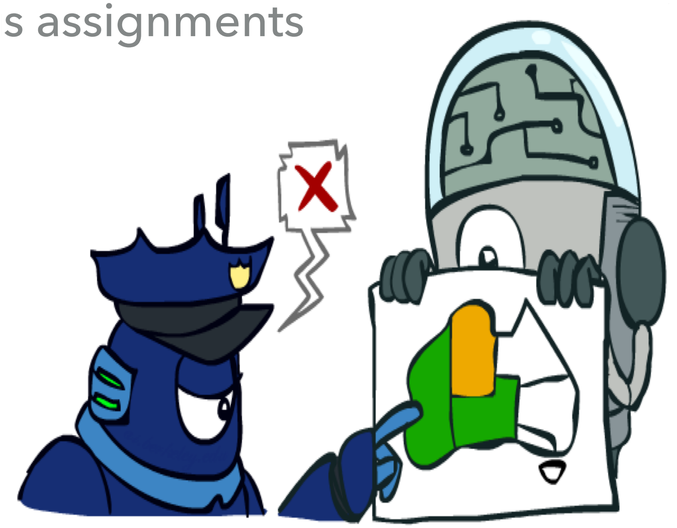
- Constraint satisfaction problems (CSPs) – a special subset of search problems
  - State is defined by **variables $X_i$** with values from a **domain D**
  - Goal test is a set of **constraints** specifying allowable combinations of values for subsets of variables
- Binary constraint graph: nodes are variables, arcs show constraints



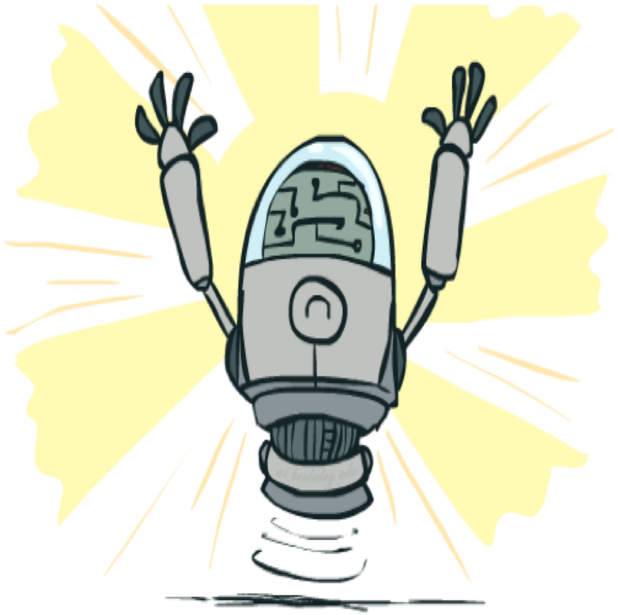
# RECAP: BACKTRACKING SEARCH

- Backtracking search is the basic uninformed algorithm for  solving CSPs
- Idea 1: One variable at a time
  - Variable assignments are **commutative**, i.e., [WA = red then NT = green] is the same as [NT = green then WA = red]
  - Fix ordering of variables and only consider assignments to a single variable at each step
- Idea 2: Check constraints as you go
  - "Incremental goal test" i.e. consider only values which do not conflict previous assignments
  - Might have to do some computation to check the constraints
- Depth-first search with these two improvements is called **backtracking search** (not the best name)
- Can solve n-queens for n ≈ 25



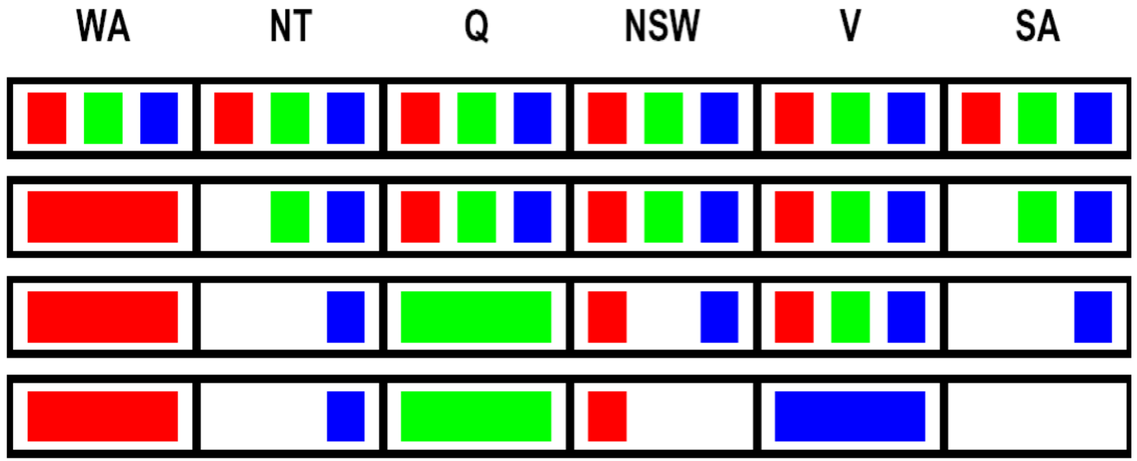
# IMPROVING BACKTRACKING

- General-purpose ideas can result in huge gains in speed
- 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?

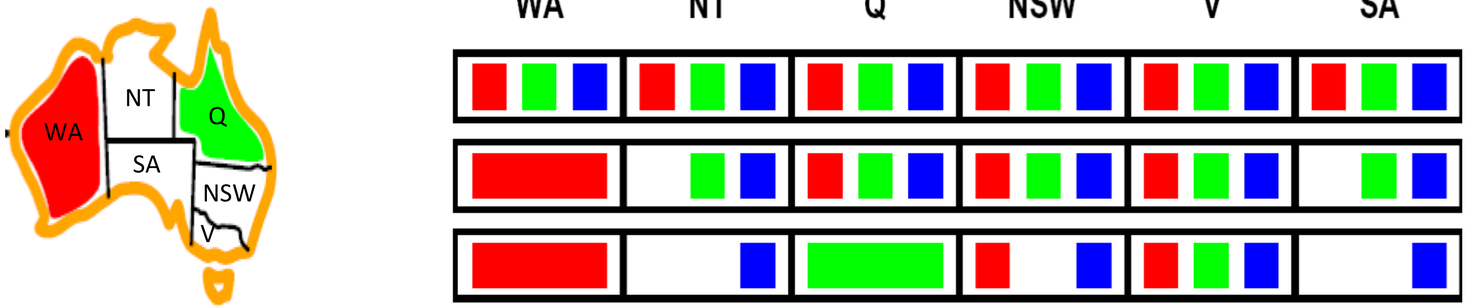


# FILTERING: FORWARD CHECKING

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

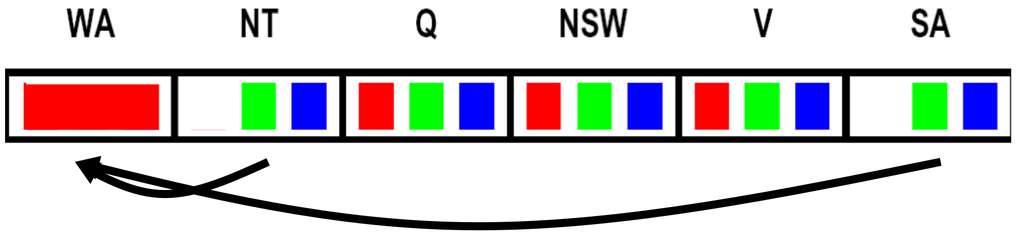
# IS FORWARD CHECKING ENOUGH?

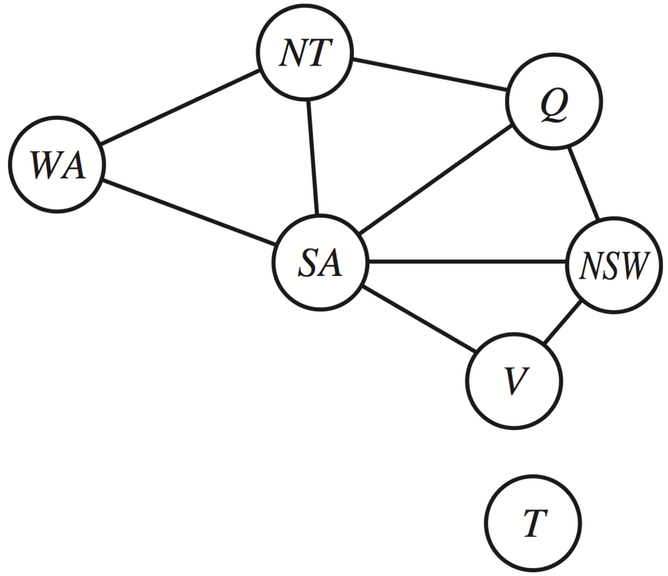



- NT and SA cannot both be blue! Why didn't we detect this yet?
- Forward checking propagates information from assigned variable to unassigned variables, but doesn't provide early detection for all failures…
  - Because delete a value in an unassigned variable may cause some constraints impossible to be satisfied!
- **Constraint propagation**: reason from constraint to constraint

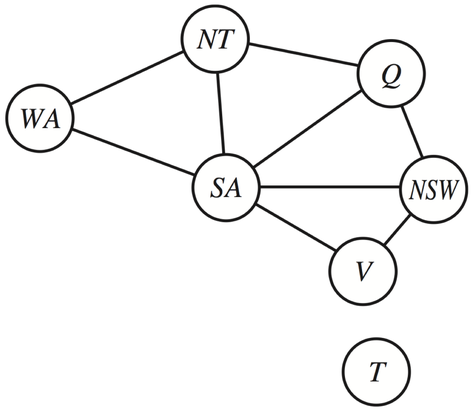
# CONSISTENCY OF A SINGLE ARC

- 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
- Forward checking: Enforcing consistency of arcs from unassigned variables (i.e., X) pointing to the variable with a new assignment (i.e., Y)

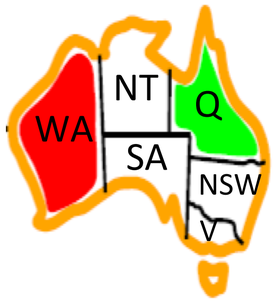



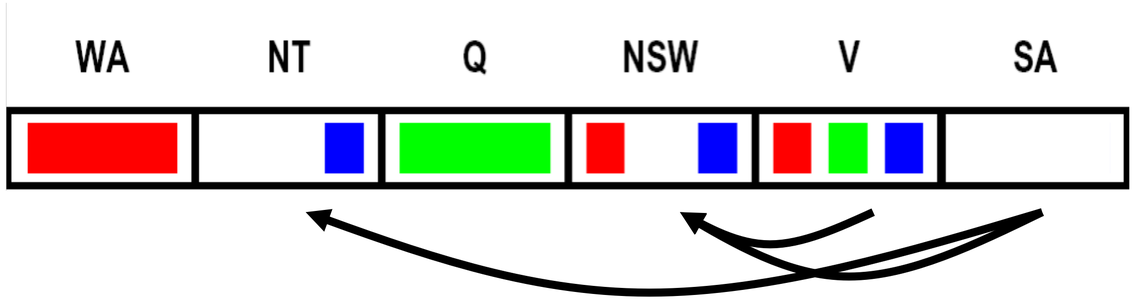



# ARC CONSISTENCY OF AN ENTIRE CSP




- Important: If X loses a value, neighbors of X need to be rechecked!
  - X is now the “Y”, while neighbors of X are the new “X”s!
- A simple form of propagation makes sure **all** arcs are consistent:
  - After Q=green, NSW, NT, SA loses green in its domain…




Failure!

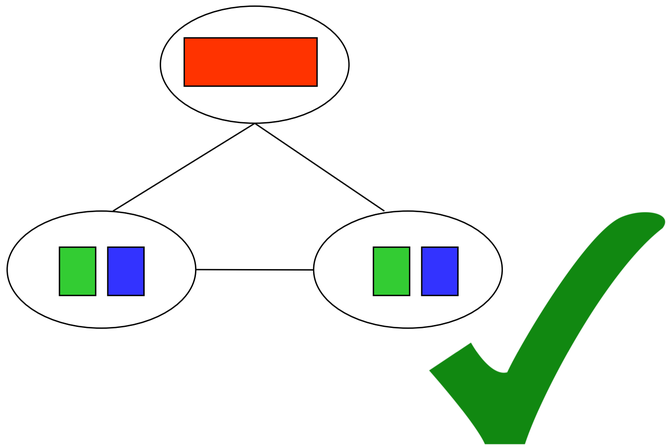
# ENFORCING ARC CONSISTENCY IN A CSP

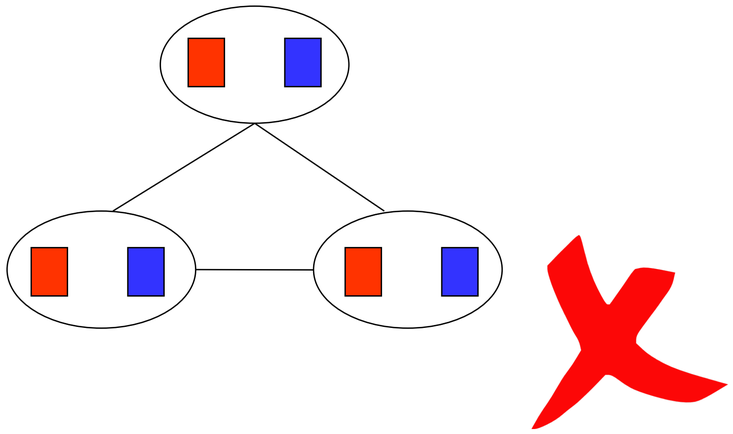
```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (Xi, Xj) ← REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
      for each Xk in NEIGHBORS[Xi] do
        add (Xk, Xi) to queue

---

function REMOVE-INCONSISTENT-VALUES( Xi, Xj) returns true iff succeeds
  removed ← false
  for each x in DOMAIN[Xi] do
    if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
      then delete x from DOMAIN[Xi]; removed ← true
  return removed
```

- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment

# 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)

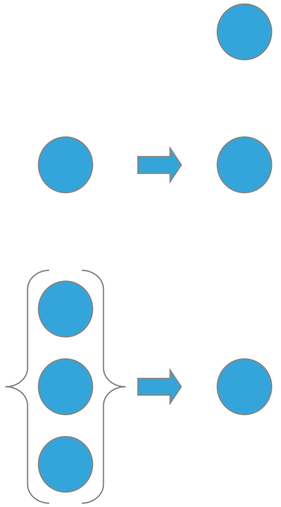


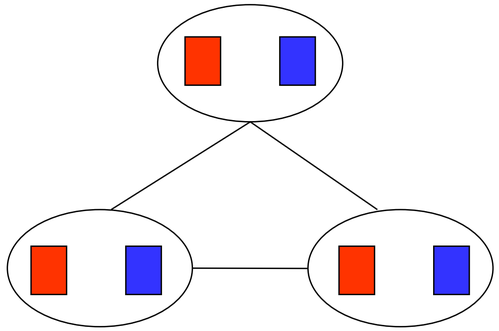


What went wrong here?

# 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 kth node.



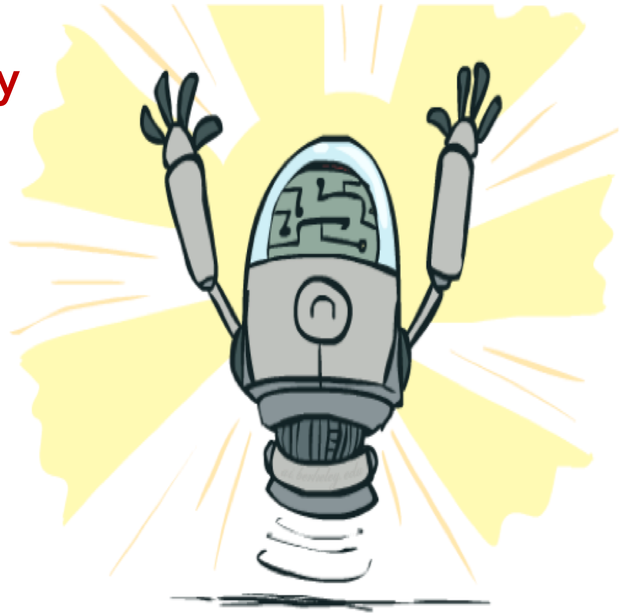


Does not satisfy 3-consistency!

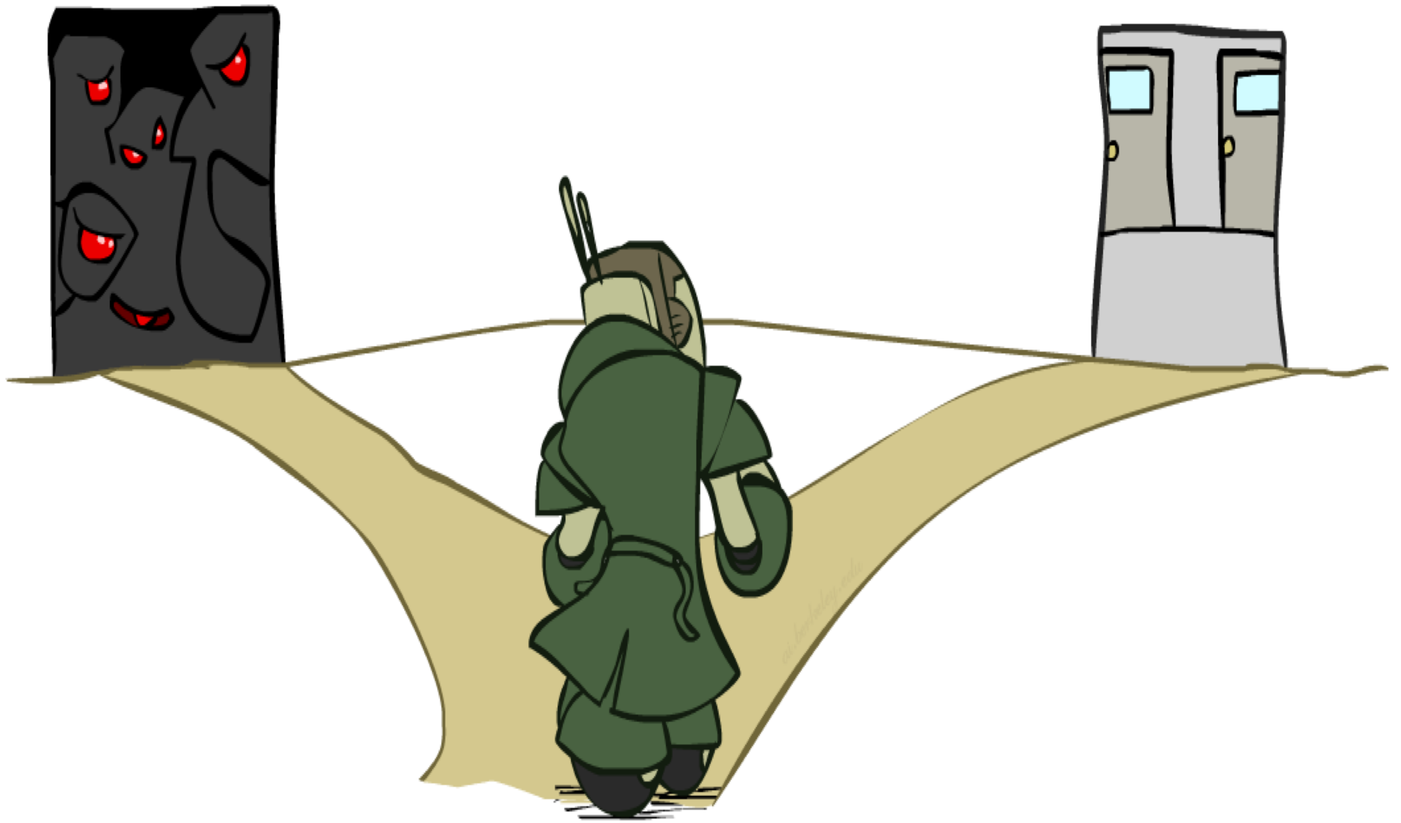
Higher k is more expensive to compute!

# IMPROVING BACKTRACKING

- General-purpose ideas can result in huge gains in speed
- Filtering: **Forward-checking, constraint propagation, k-consistency**
  - 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?



# ORDERING



# 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
```

Should we always follow a fixed order?

# ORDERING OF VARIABLES: MINIMUM REMAINING VALUES

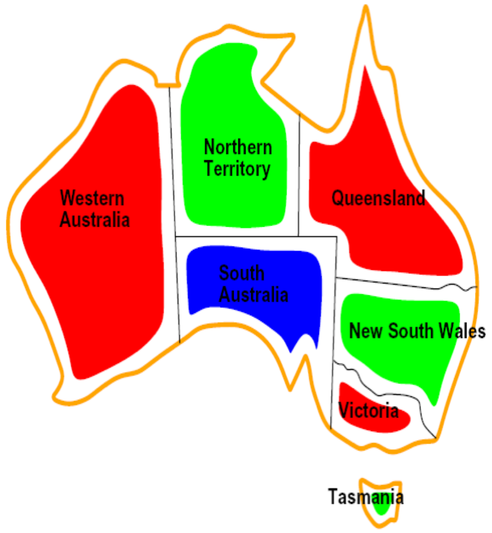
$var \leftarrow \text{SELECT-UNASSIGNED-VARIABLE}(\text{VARIABLES}[csp], assignment, csp)$

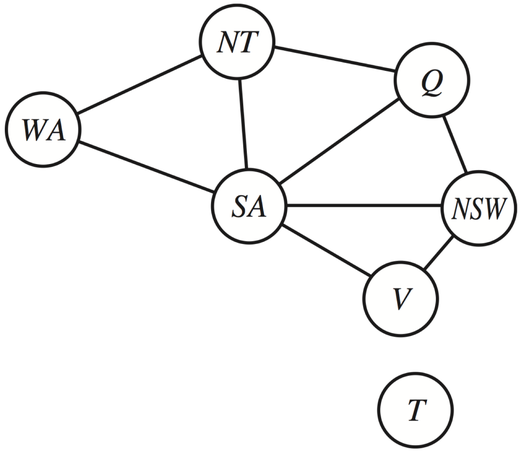
- Variable Ordering: the **Minimum Remaining Values (MRV)** heuristic
  - Choose the variable with the fewest number of remaining values in its domain
  - Also called “most constrained variable”

- Why min rather than max?
- “Fail-fast” ordering: If a partial assignment will fail, let it fail as early as possible!

# ORDERING OF VARIABLES: DEGREE HEURISTIC

- What about the first variable to choose?
- **Degree heuristic**: select the variable that is involved in the largest number of constraints
  - Makes the largest number of other variables more “constrained” and enable early failures indirectly!

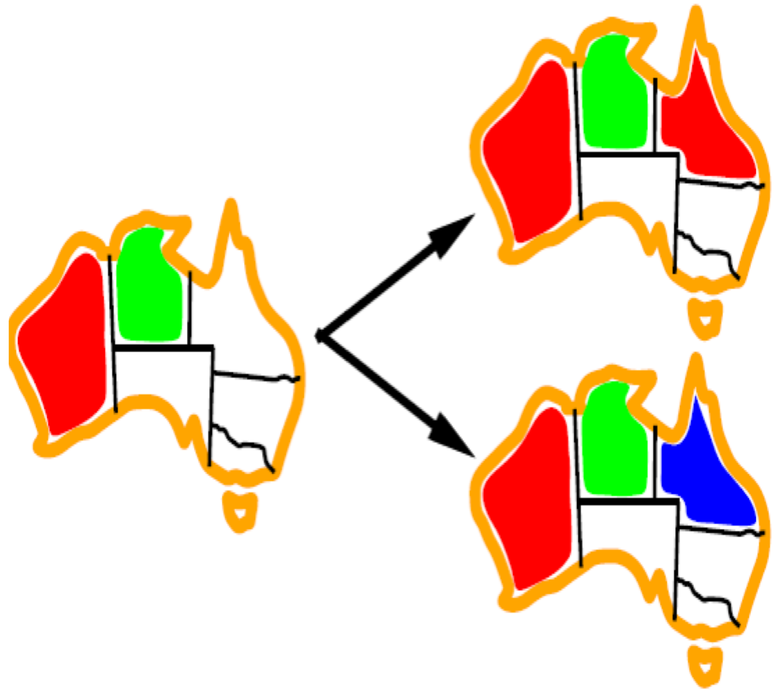


# ORDERING: LEAST CONSTRAINING VALUE

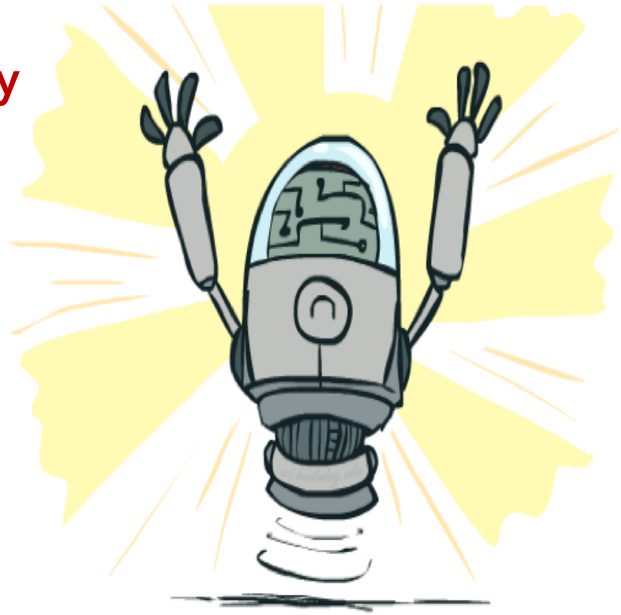
for each *value* in ORDER-DOMAIN-VALUES(*var*, *assignment*, *csp*) do

- Value Ordering: the **Least Constraining Value** heuristic
  - Given a choice of variable, choose the *least constraining value,* i.e., the one that rules out the fewest values in the other unassigned variables
  - It may take some computation to determine this (e.g., rerun filtering)
- Why least rather than most?
- Once the variable to be assigned is determined, we only need one single consistent solution!
- Combining these ordering ideas makes n-queens feasible for n=1000



# IMPROVING BACKTRACKING

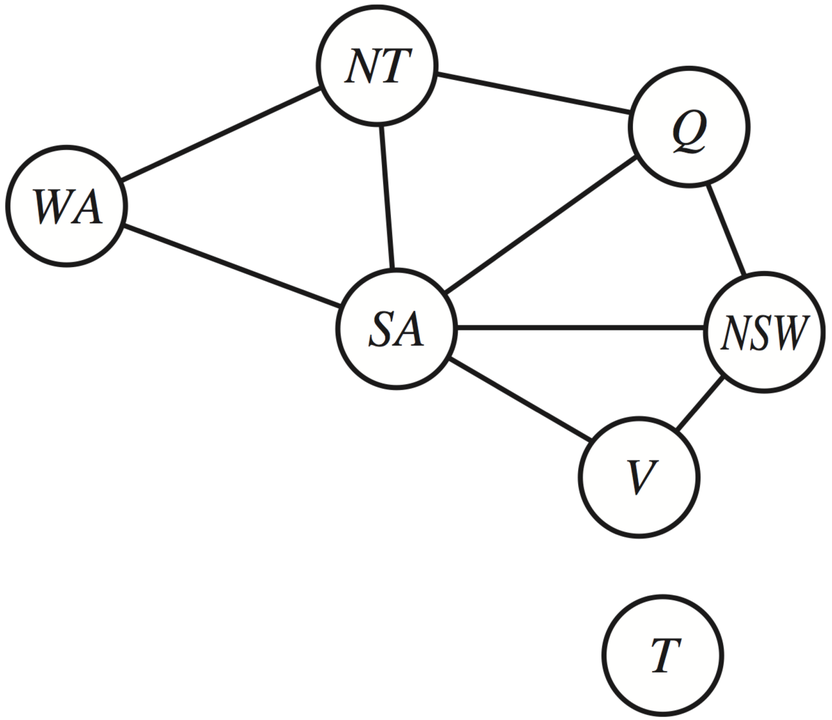
- General-purpose ideas can result in huge gains in speed
- Filtering: **Forward-checking, constraint propagation, k-consistency**
  - Can we detect inevitable failure early?
- Ordering:
  - Which variable should be assigned next? **MRV, degree**
  - In what order should its values be tried? **LCV**
- Structure:
  - Can we exploit the problem structure?



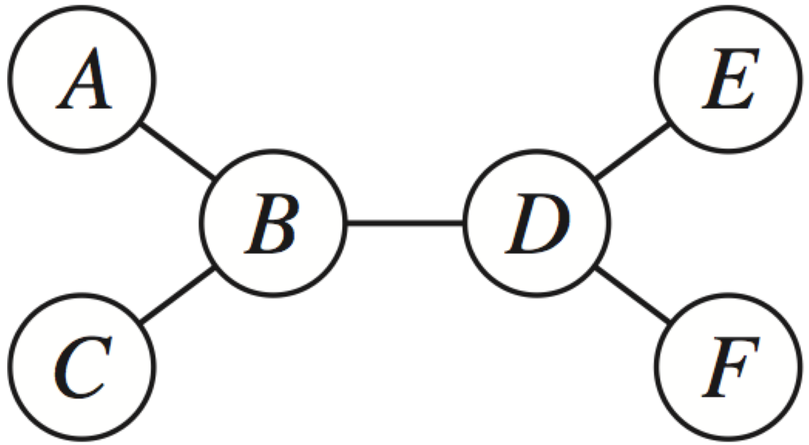
# PROBLEM STRUCTURE

# PROBLEM STRUCTURE

- Extreme case: independent subproblems
  - Example: Tasmania and mainland do not interact
- Independent subproblems are identifiable as **connected components** of constraint graph
  - Each subproblem can be solved independently!
- Suppose a graph of *n* variables can be broken into subproblems of only *c* variables:
  - Without decomposing: Worst-case solution cost $O(d^n)$
  - With decomposing: $O((n/c)(d^c))$, linear in n
  - E.g., n = 80, d = 2, c =20
  - $2^{80}$ = 4 billion years at 10 million combinations/sec
  - $(4)(2^{20})$ = 0.4 seconds at 10 million combinations/sec



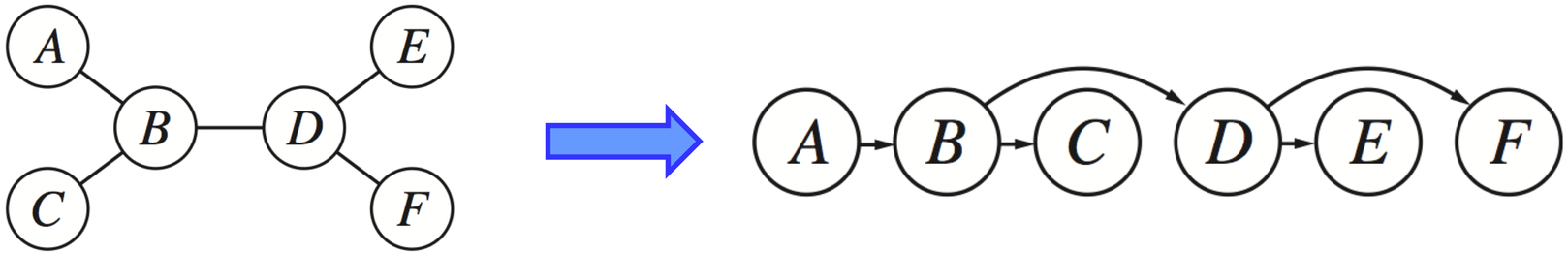
# TREE-STRUCTURED CSPS




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

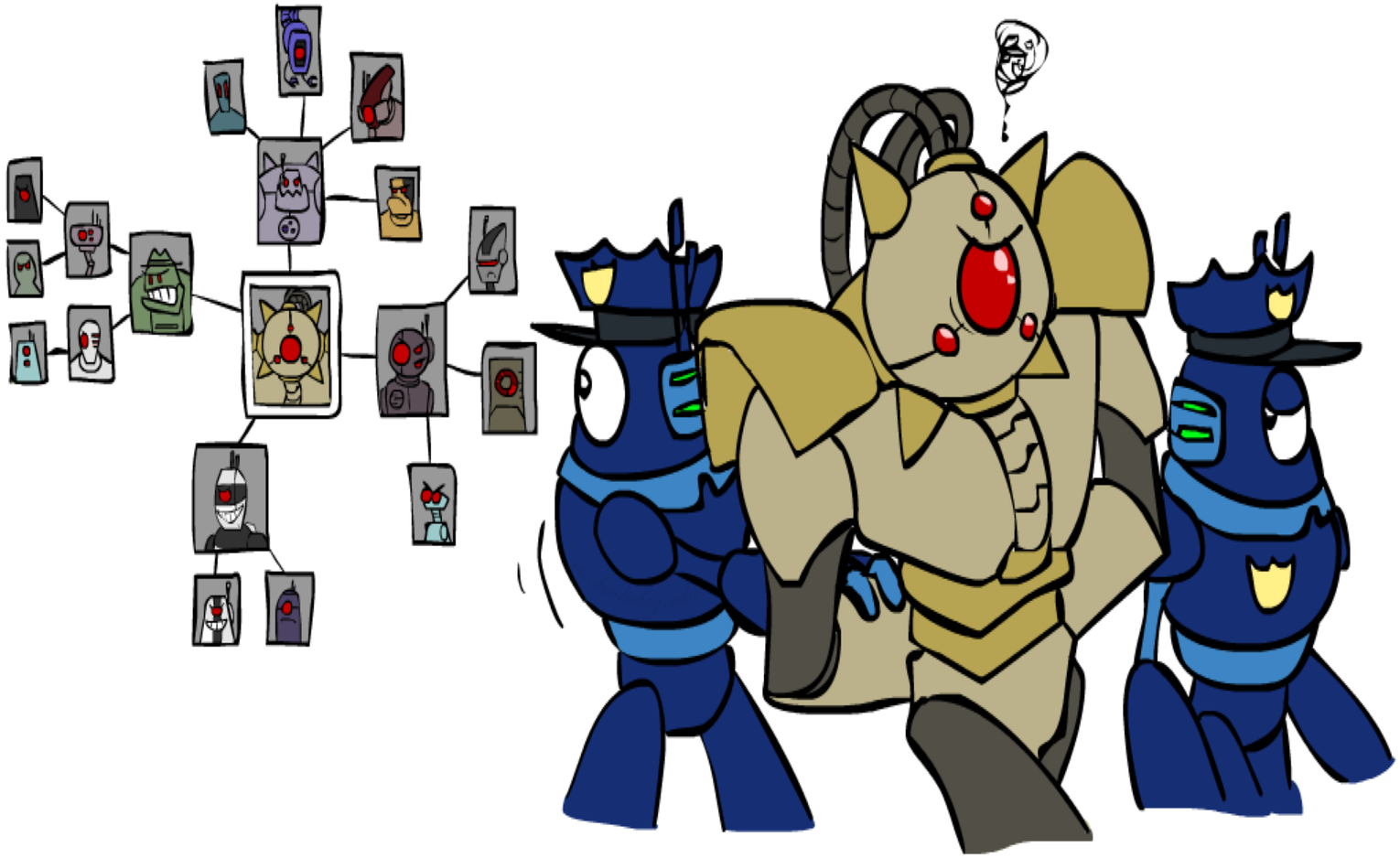
# TREE-STRUCTURED CSPS

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

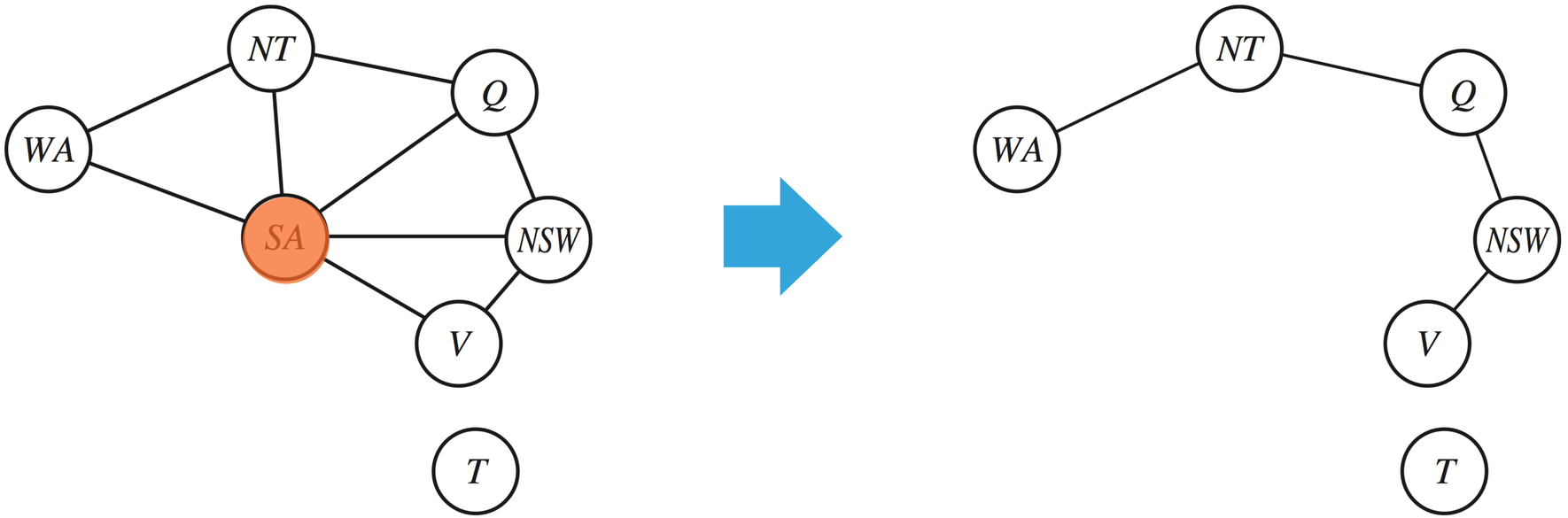



- Enforce directed arc consistency from the leaf to root
  - Runtime: $O(nd^2)$
- Assign values from root to leaf: directed arc consistency ensures that a consistent solution exists!

# IMPROVING STRUCTURE



# NEARLY TREE-STRUCTURED CSPS




- **Cutset**: a set of variables such that after removing variables in this set and the constraints associated with variables in this set, the constraint graph becomes a tree
- **Cutset conditioning**: instantiate (in all ways) the variables in the cutset, update the domains for the remaining variables, and solve the CSP subproblem represented by the remaining constraint tree
- When the number of variables in the cutset size is *c*, the worst case runtime is $O((d^c)\,(n-c)d^2)$, very fast for small *c*

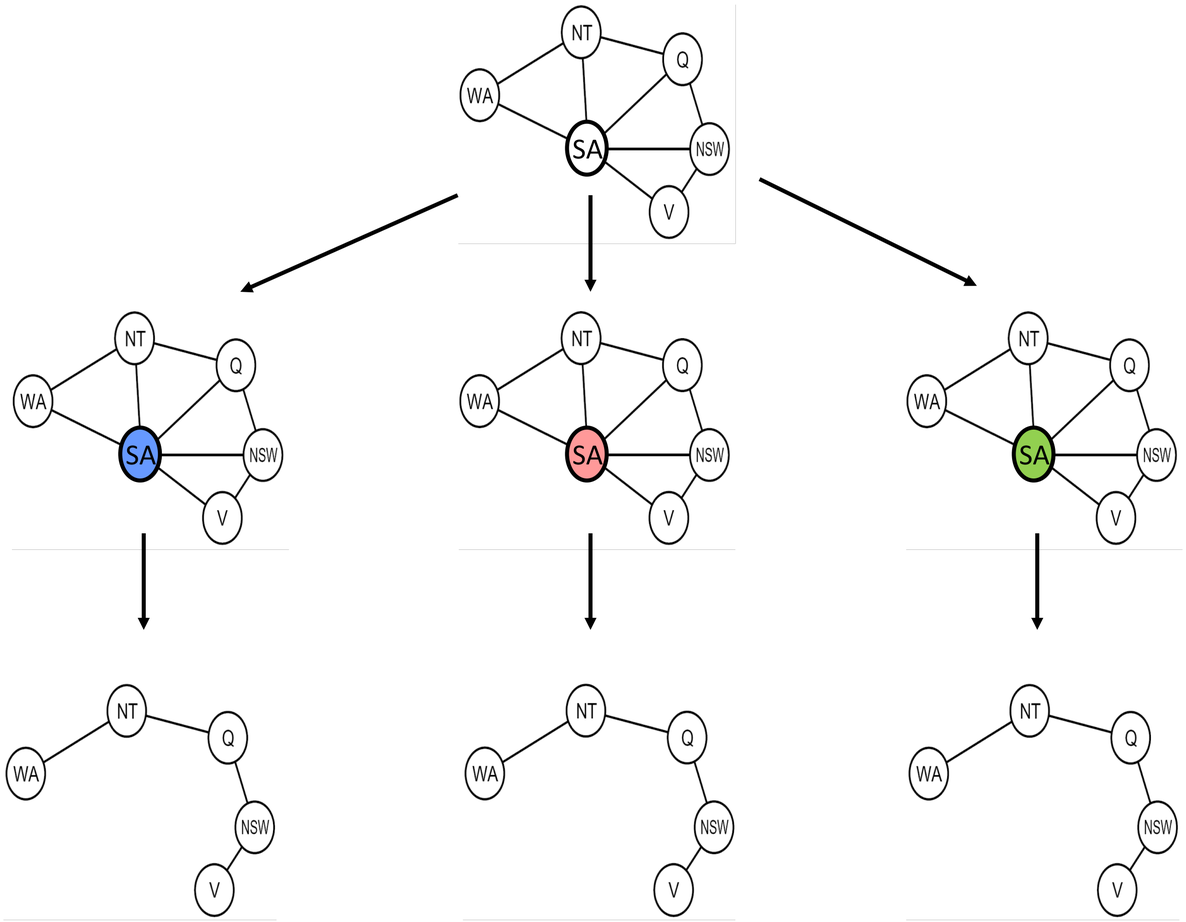
# CUTSET CONDITIONING

Choose a cutset

Instantiate the cutset (all possible ways)

Compute residual CSP for each assignment

Solve the residual CSPs (tree structured)



# IMPROVING BACKTRACKING

- General-purpose ideas can result in huge gains in speed
- Filtering: **Forward-checking, constraint propagation, k-consistency**
  - Can we detect inevitable failure early?
- Ordering:
  - Which variable should be assigned next? **MRV, degree**
  - In what order should its values be tried? **LCV**
- Structure:
  - Can we exploit the problem structure?

**Identify independent subproblems, leverage tree structures, and cutset conditioning**