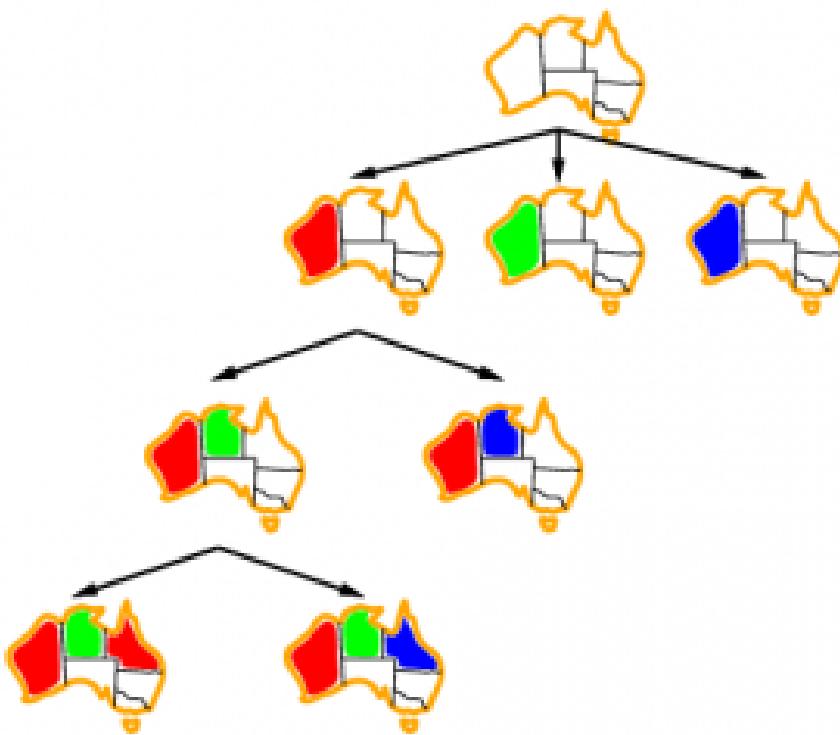


# CAB320 - Constraint Satisfaction



*Credits: most material borrowed from AIMA or Dan Klein and Pieter Abbeel*

# Reading for this week

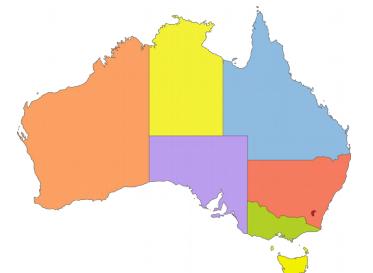
- Chapter 6, Sections 6.1 to 6.6 of AIMA
  - Russell and Norvig Textbook:  
*Artificial Intelligence, a Modern Approach*  
3rd edition

# Today's Menu

- Different types of search
- Examples
- Constraint graphs
- Search methods
- Exploiting problem structure

# Different types of search

- Assumptions about the world so far: a single agent, deterministic actions, fully observed state, discrete state space
- Planning: sequences of actions
  - The path to the goal is the important thing
  - Paths have various costs, depths
  - Heuristics give problem-specific guidance
- Identification: assignments to variables
  - The goal itself is important, not the path
  - All paths at the same depth (for some formulations)
  - CSPs are specialized for identification problems



# Constraint Satisfaction Problems

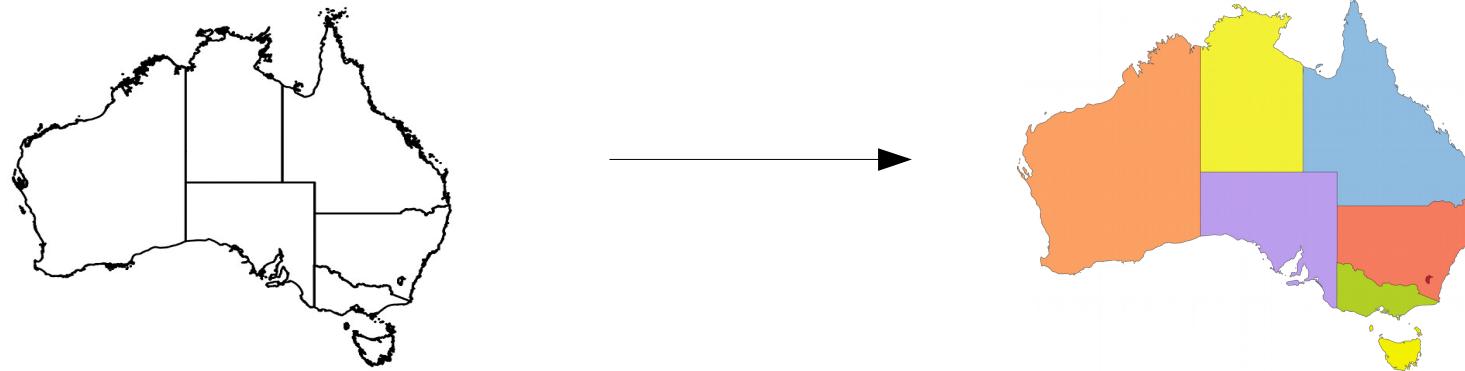
- Standard search problems:
  - State is a “black box”: arbitrary data structure
  - Goal test can be any function over states
  - Successor function can also be anything
- Constraint satisfaction problems (CSPs):
  - A special subset of search problems
  - State is defined by **variables  $X_i$**  with values from a **domain  $D$**  (sometimes  $D$  depends on  $i$ )
  - **Goal test** is a set of constraints specifying allowable combinations of values for subsets of variables



What about Sokoban?

# Example: Map Colouring

- **Variables:** WA, NT, Q, NSW, V, SA, T
- **Domains:**  $D = \{\text{red, green, blue}\}$
- **Constraints:** adjacent regions must have different colors
  - Implicit:  $\text{WA} \neq \text{NT}$
  - Explicit:  $(\text{WA}, \text{NT}) \in \{(\text{red, green}), (\text{red, blue}), \dots\}$
- **Solutions are assignments satisfying all constraints, e.g.:**
$$\{\text{WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}\}$$

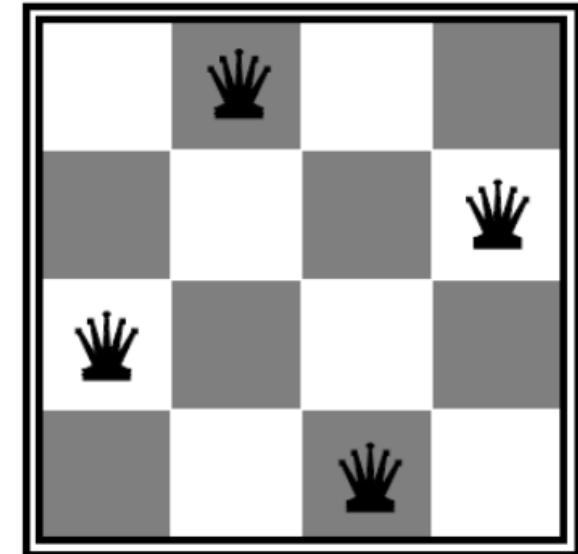


# Example: N-Queens



## ■ Formulation 1:

- Variables:  $X_{ij}$
- Domains:  $\{0, 1\}$
- Constraints



$$\forall i, j, k \quad (X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\}$$

$$\forall i, j, k \quad (X_{ij}, X_{kj}) \in \{(0, 0), (0, 1), (1, 0)\}$$

$$\forall i, j, k \quad (X_{ij}, X_{i+k, j+k}) \in \{(0, 0), (0, 1), (1, 0)\}$$

$$\forall i, j, k \quad (X_{ij}, X_{i+k, j-k}) \in \{(0, 0), (0, 1), (1, 0)\}$$

$$\sum_{i,j} X_{ij} = N$$

# Example: N-Queens

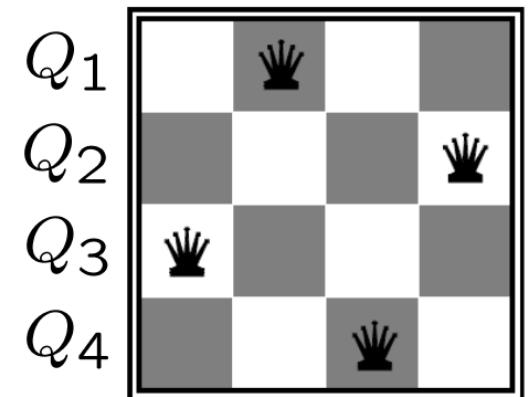
- Formulation 2:

- Variables:  $Q_k$
- Domains:  $\{1, 2, 3, \dots, N\}$
- Constraints:

Implicit:  $\forall i, j \text{ non-threatening}(Q_i, Q_j)$

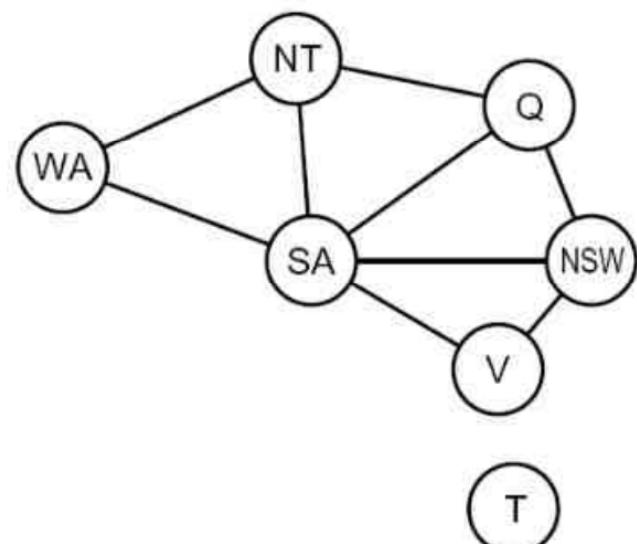
Explicit:  $(Q_1, Q_2) \in \{(1, 3), (1, 4), \dots\}$

• • •



# Constraint Graphs

- Binary CSP: each constraint relates (at most) two variables
- Binary constraint graph: nodes are variables, arcs show constraints
- General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!



# Example: Cryptarithmetic

- Variables:

$F \ T \ U \ W \ R \ O \ X_1 \ X_2 \ X_3$

- Domains:

$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

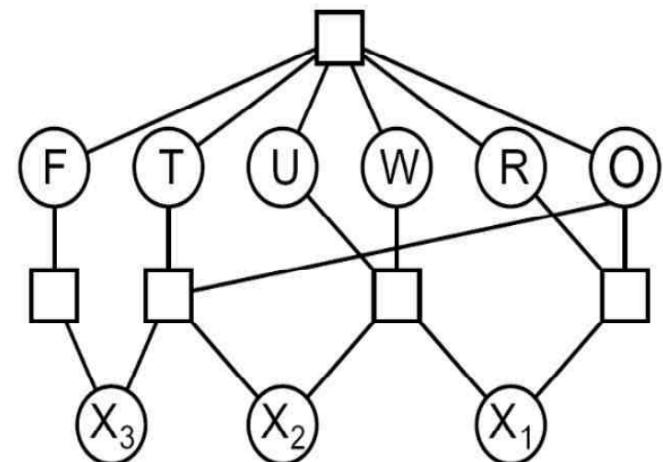
- Constraints:

`alldiff( $F, T, U, W, R, O$ )`

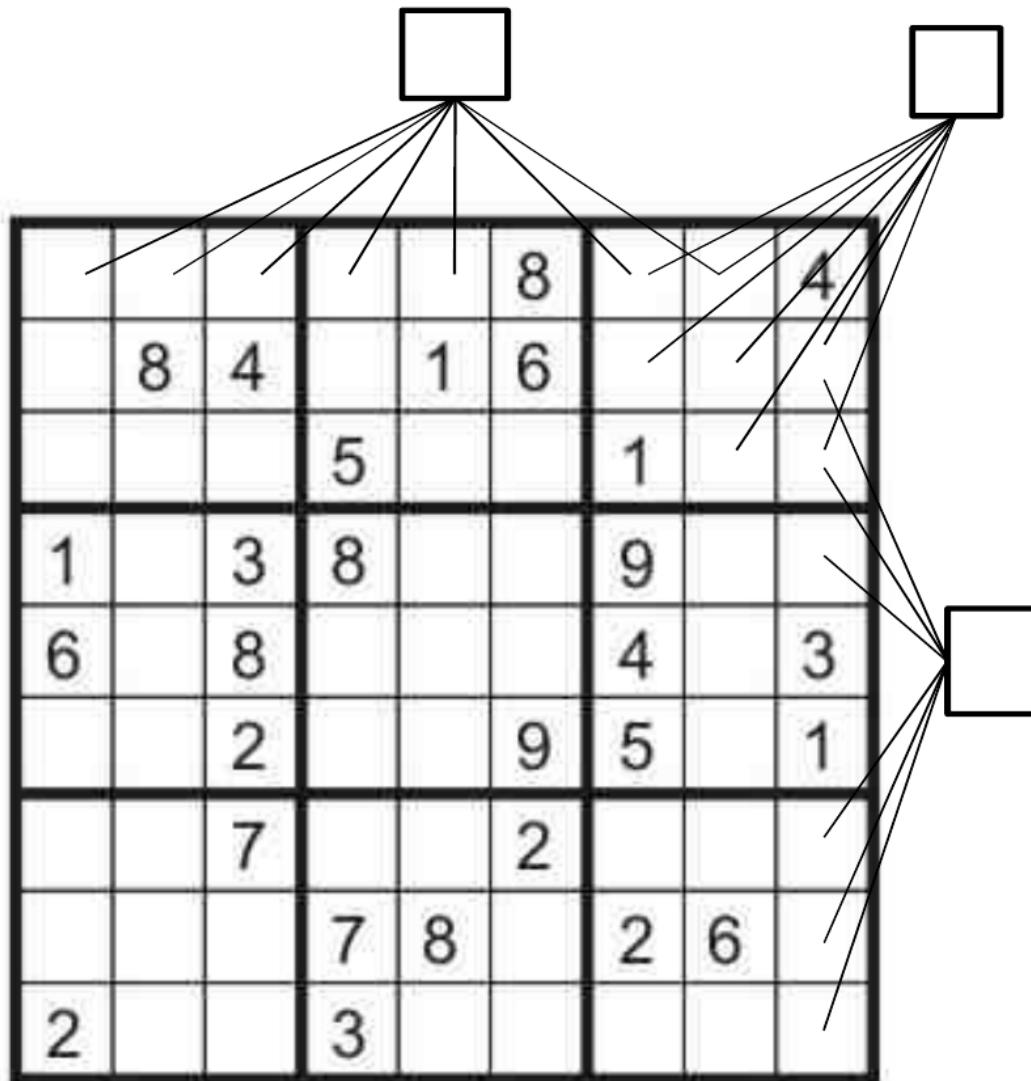
$$O + O = R + 10 \cdot X_1$$

• • •

$$\begin{array}{r} \text{T} \ \text{W} \ \text{O} \\ + \ \text{T} \ \text{W} \ \text{O} \\ \hline \text{F} \ \text{O} \ \text{U} \ \text{R} \end{array}$$



# Example: Sudoku



- Variables:
  - Each (open) square
- Domains:
  - $\{1, 2, \dots, 9\}$
- Constraints:
  - 9-way alldiff for each column
  - 9-way alldiff for each row
  - 9-way alldiff for each region
  - (or can have a bunch of pairwise inequality constraints)

# Varieties of CSPs

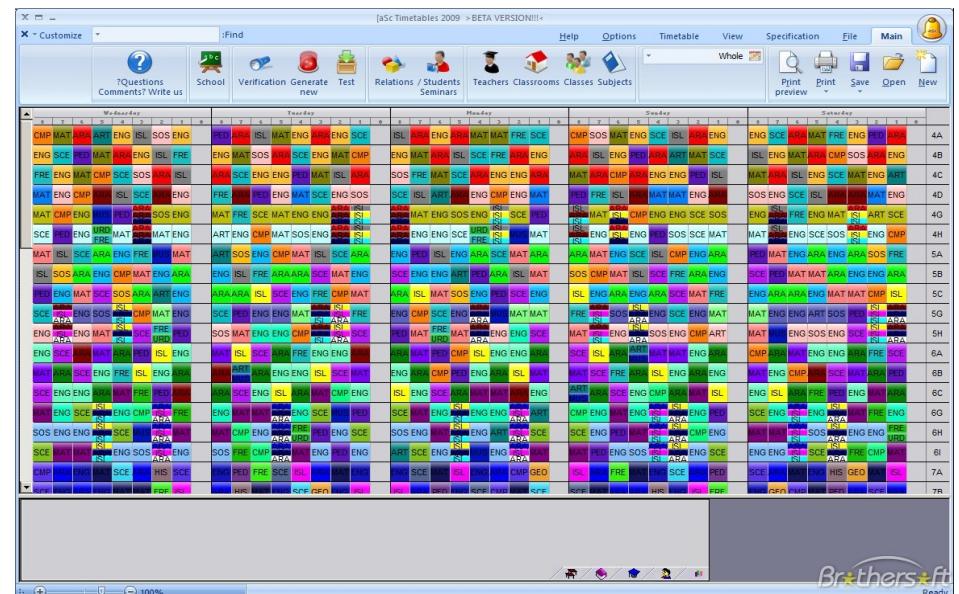
- Discrete Variables
  - Finite domains
    - Size  $d$  means  $O(d^n)$  complete assignments
    - E.g., Boolean CSPs, including Boolean satisfiability (NP-complete)
  - Infinite domains (integers, strings, etc.)
    - E.g., job scheduling, variables are start/end times for each job
    - Linear constraints solvable, nonlinear undecidable
- Continuous variables
  - E.g., start/end times for Hubble Telescope observations
  - Linear constraints solvable in polynomial time by LP methods

# Varieties of Constraints

- Varieties of Constraints
  - Unary constraints involve a single variable (equivalent to reducing domains), e.g.:  
 $SA \neq \text{green}$
  - Binary constraints involve pairs of variables, e.g.:  
 $SA \neq WA$
  - Higher-order constraints involve 3 or more variables:  
e.g., cryptarithmetic column constraints
- Preferences (soft constraints):
  - E.g., red is better than green
  - Often representable by a cost for each variable assignment
  - Gives constrained optimization problems

# Real-World Applications CSPs

- Assignment problems: e.g., who teaches what class
- Timetabling problems: e.g., which class is offered when and where?
- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Circuit layout
- Fault diagnosis
- ... lots more!



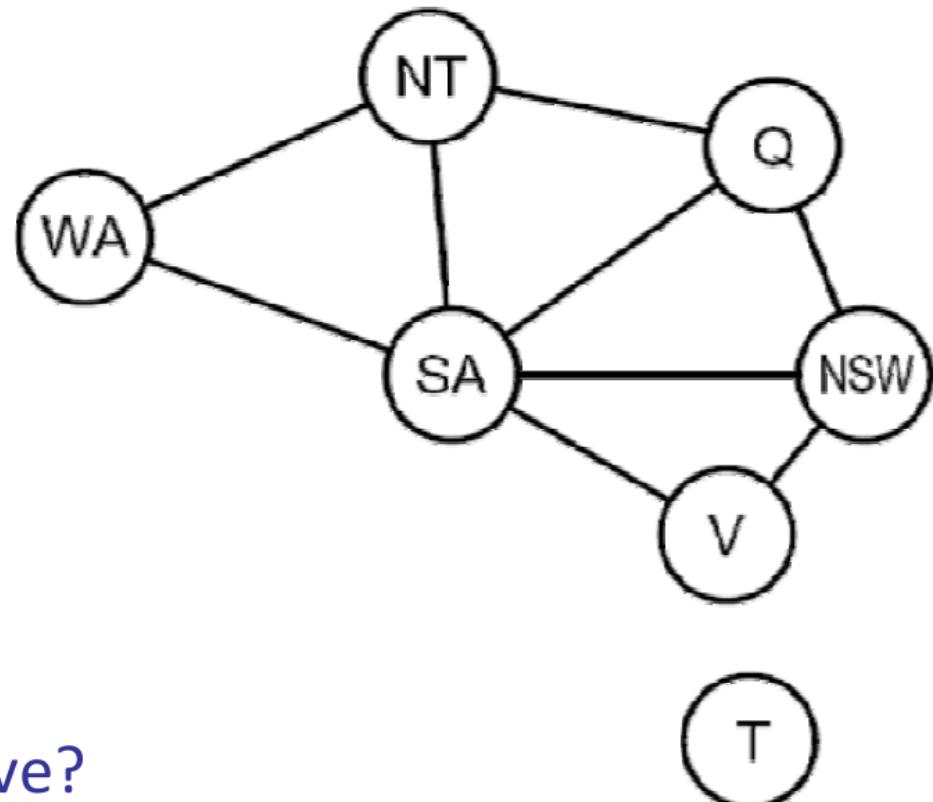
# Standard Search Formulation

- Standard search formulation of CSPs
- States 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
- We'll start with the straightforward, naïve approach, then improve it

# Search Methods



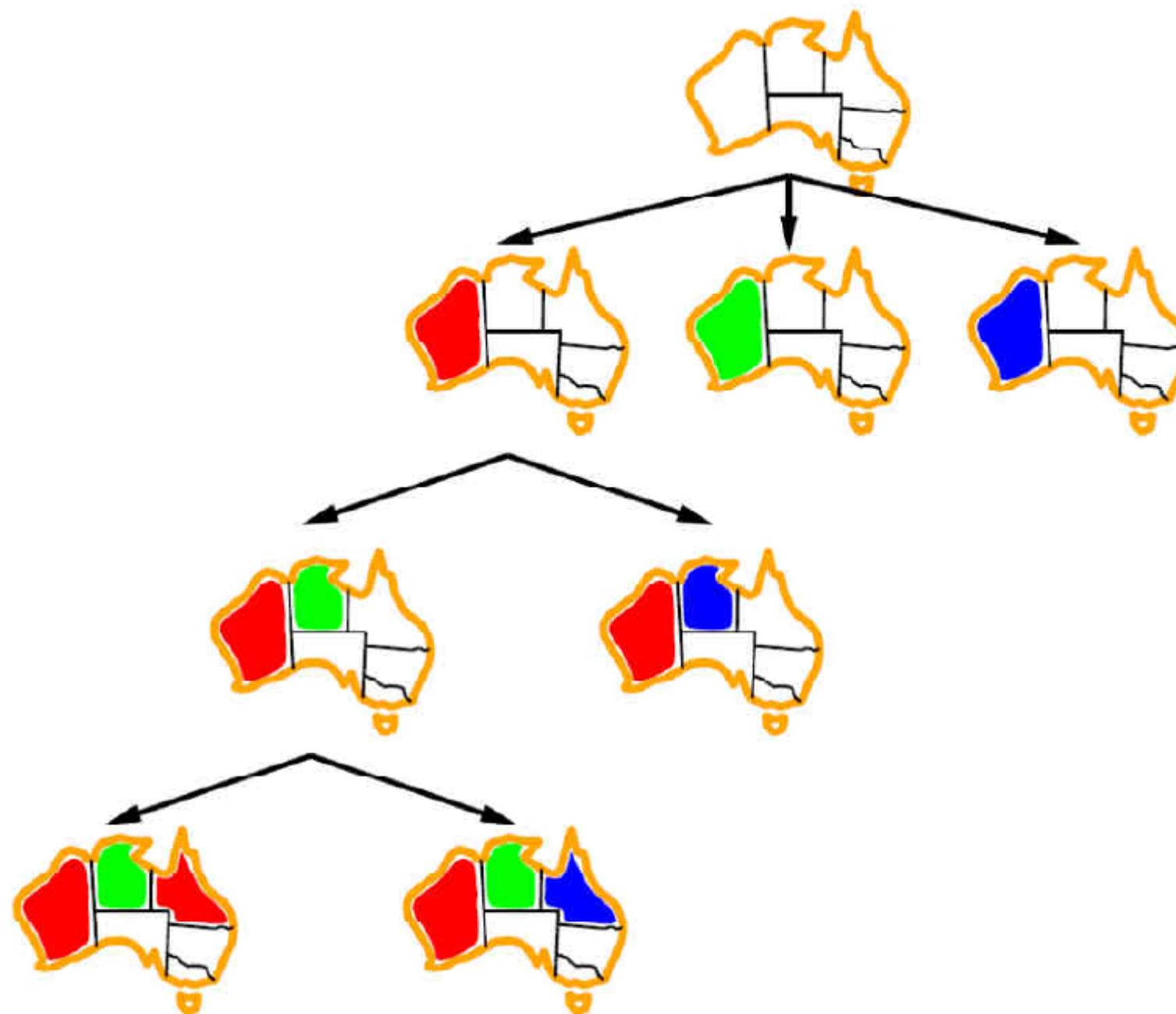
- What would BFS do?
- What would DFS do?
- What problems does naïve search have?



# Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- 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 you go
  - I.e. consider only values which do not conflict previous assignments
  - Might have to do some computation to check the constraints
  - “Incremental goal test”
- Depth-first search with these two improvements is called *backtracking search* (not the best name)
- Can solve n-queens for  $n \approx 25$

# 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  $\leftarrow$  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  $\{ \text{var} = \text{value} \}$  to assignment
      result  $\leftarrow$  RECURSIVE-BACKTRACKING(assignment, csp)
      if result  $\neq$  failure then return result
      remove  $\{ \text{var} = \text{value} \}$  from assignment
  return failure
```



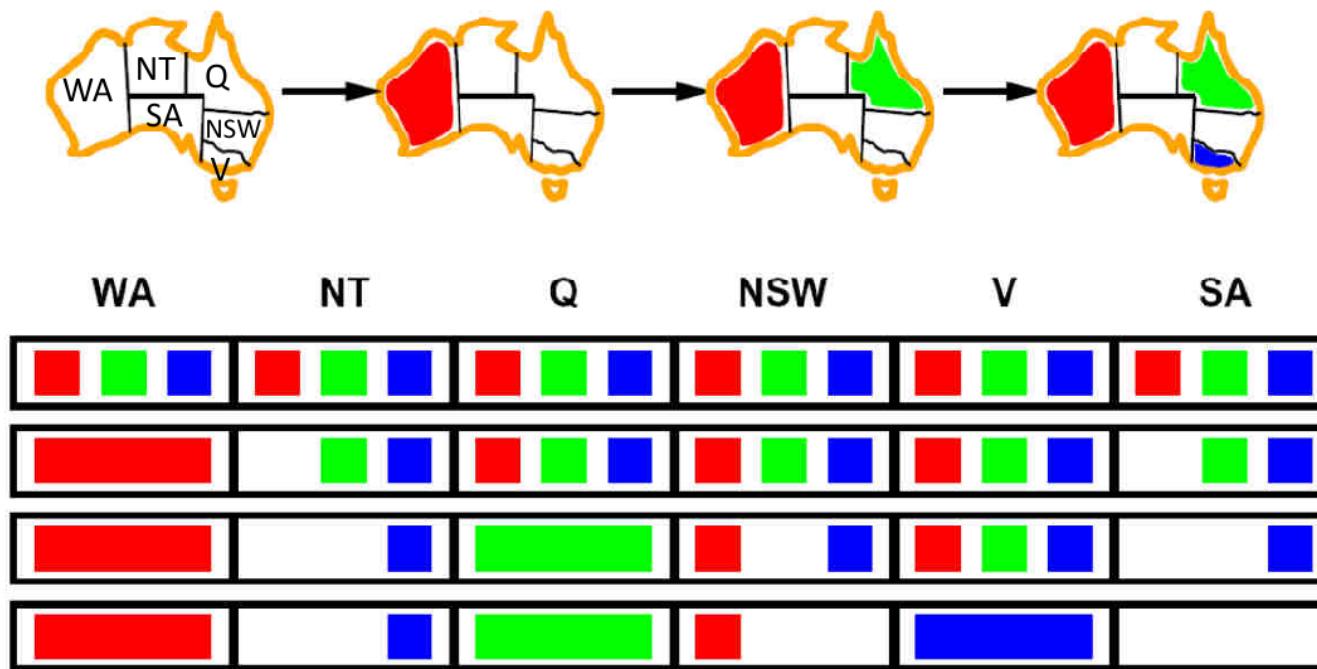
- Backtracking = DFS + variable-ordering + fail-on-violation
- What are the choice points?

# Improving Backtracking

- General-purpose ideas give huge gains in speed
- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?
- Filtering: Can we detect inevitable failure early?
- 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 added to the existing assignment



# Filtering: Constraint Propagation

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

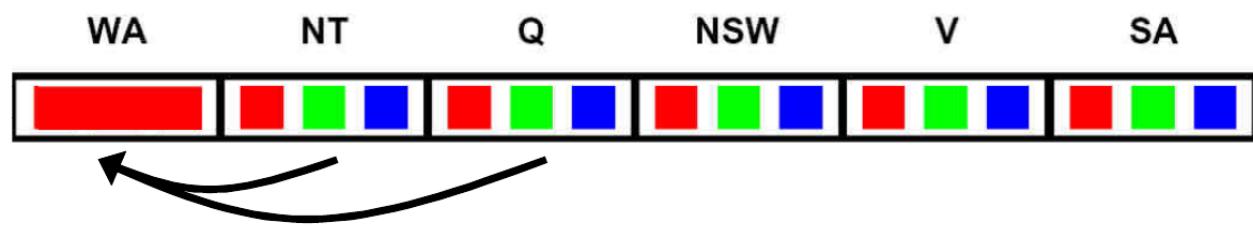


WA	NT	Q	NSW	V	SA
Red	Green	Blue	Red	Green	Blue
Red	Green	Blue	Red	Green	Blue
Red		Blue	Red	Blue	Blue

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

# Consistency of A Single Arc

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

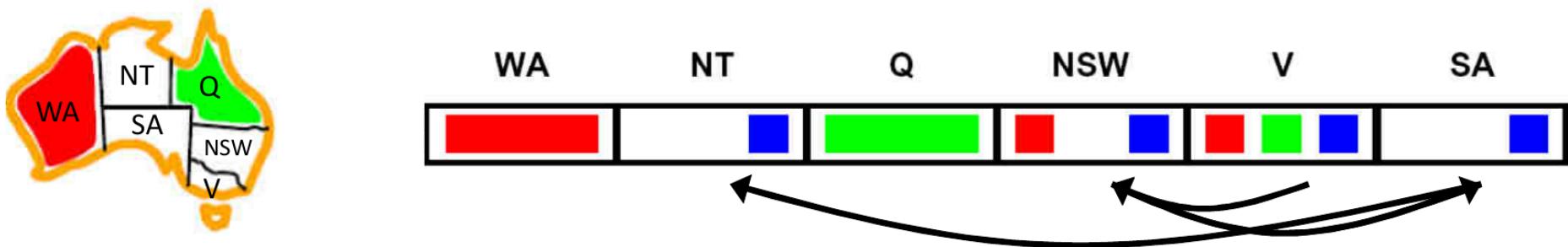


*Delete from the tail!*

- 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:



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

*Remember:  
Delete from  
the tail!*

# 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
       $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\textit{queue})$ 
      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
```

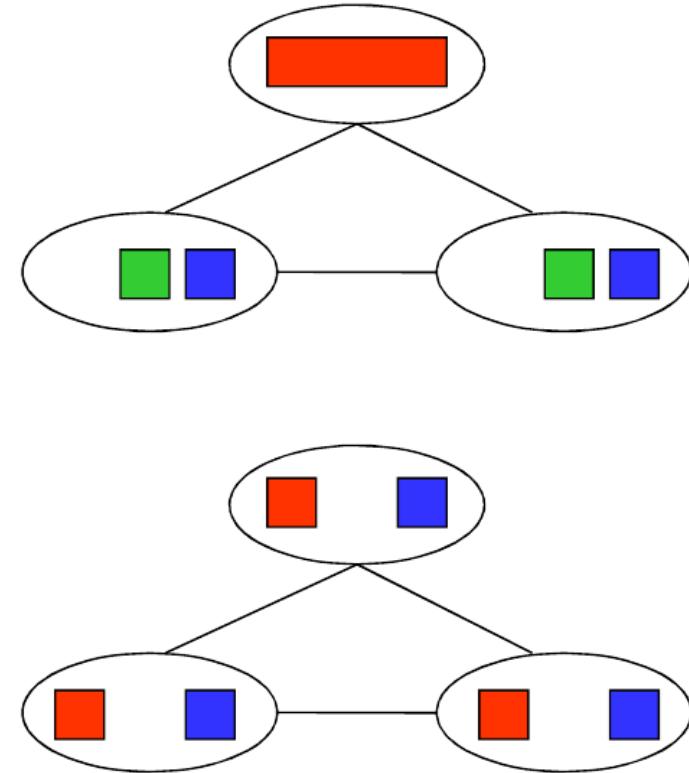
---

```
function REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) returns true iff succeeds
  removed  $\leftarrow \text{false}$ 
  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  $\leftarrow \text{true}$ 
  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?

# 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: Minimum Remaining Values

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

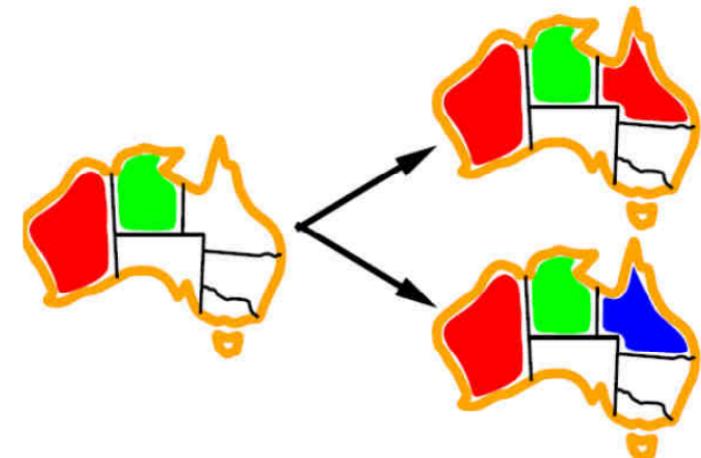


- Why min rather than max?
- Also called “most constrained variable”
- “Fail-fast” ordering

# Ordering: Least Constraining Value

- **Value Ordering: Least Constraining Value**

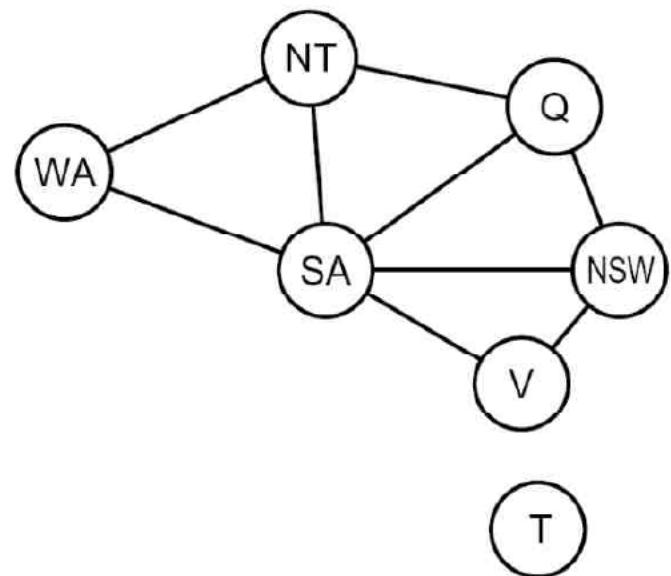
- Given a choice of variable, choose the *least constraining value*
- I.e., the one that rules out the fewest values in the remaining variables
- Note that it may take some computation to determine this! (E.g., rerunning filtering)



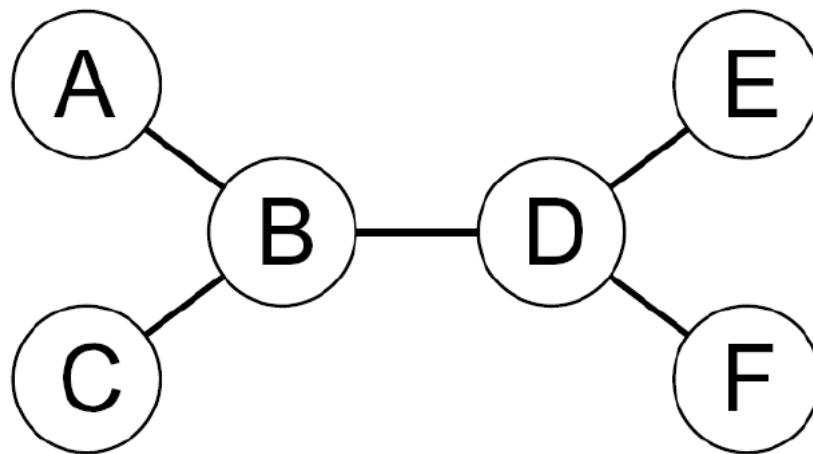
- Why least rather than most?
- Combining these ordering ideas makes 1000 queens feasible

# 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



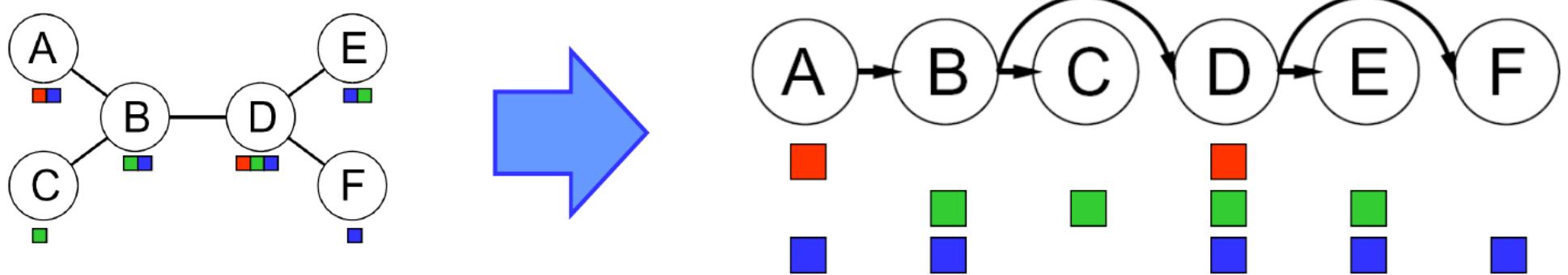
# Tree-Structured CSPs



- **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

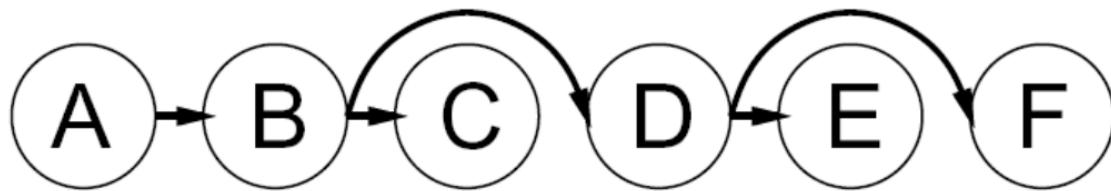
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



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

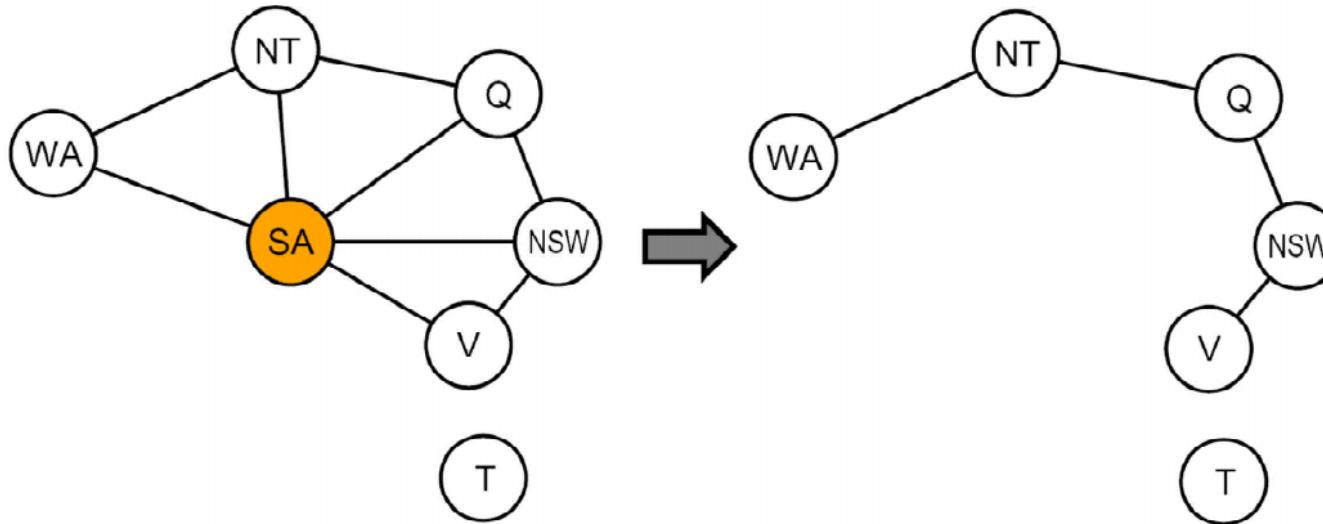
# Tree-Structured CSPs

- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Proof: Each  $X \rightarrow Y$  was made consistent at one point and  $Y$ 's domain could not have been reduced thereafter (because  $Y$ 's children were processed before  $Y$ )



- Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
- Proof: Induction on position

# 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  $c$

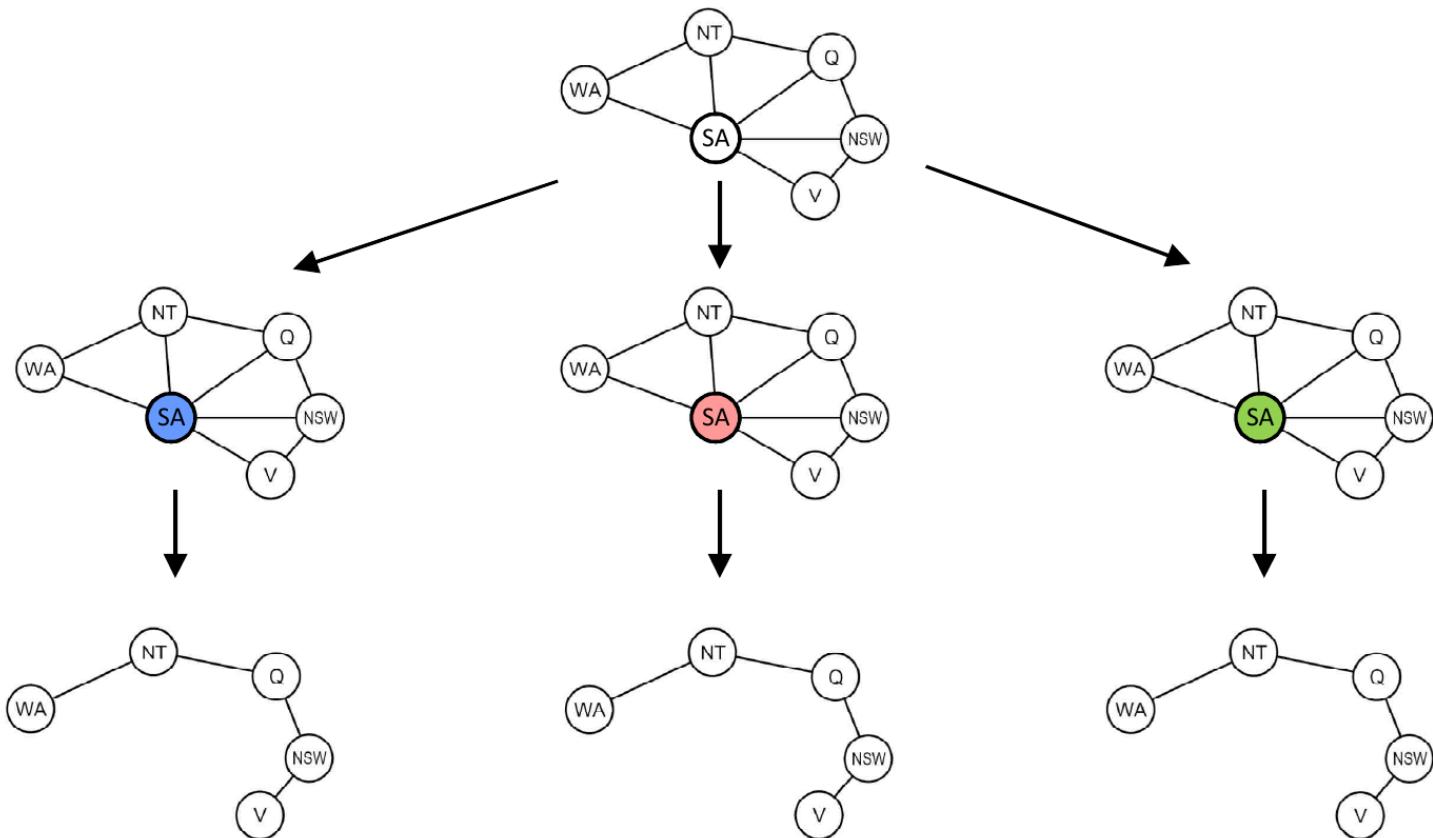
# Cutset Conditioning

Choose a cutset

Instantiate the cutset  
(all possible ways)

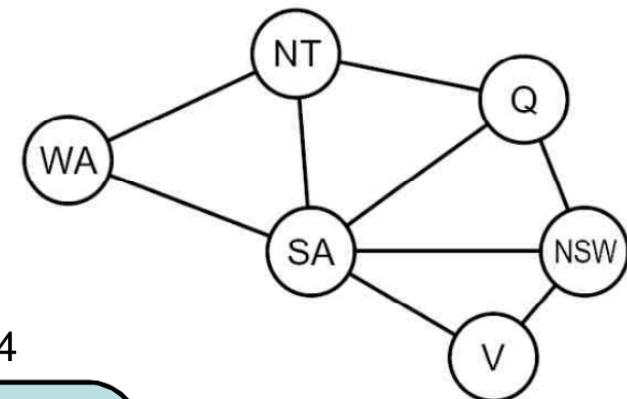
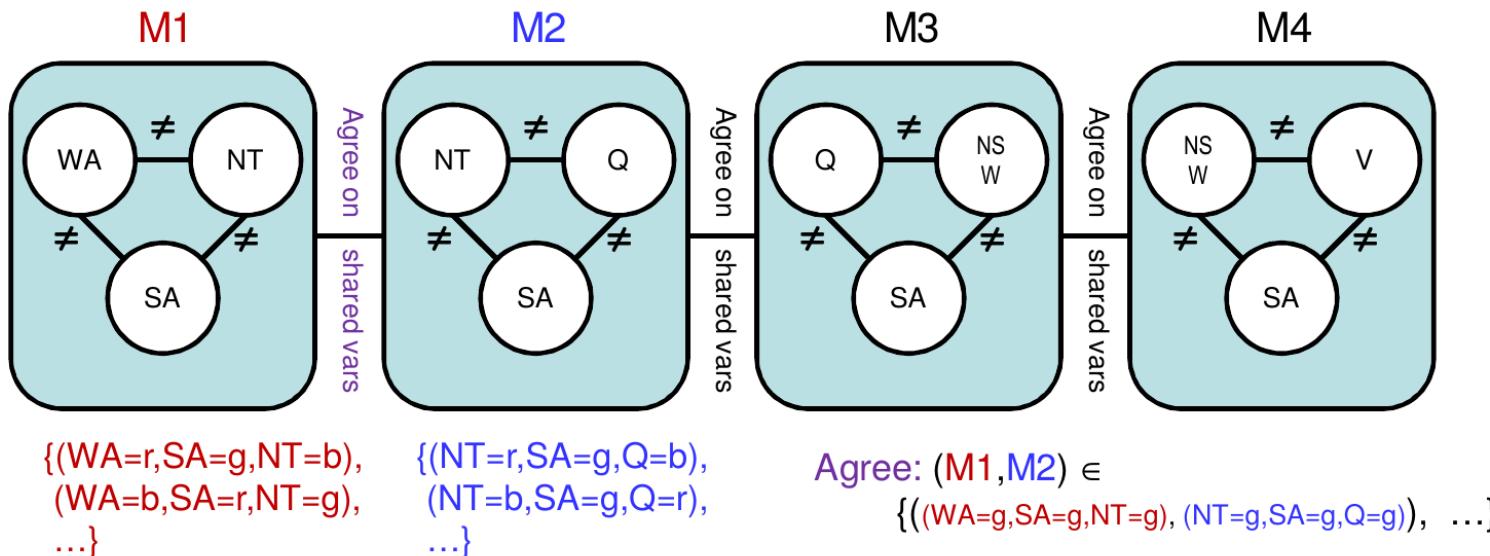
Compute residual CSP  
for each assignment

Solve the residual CSPs  
(tree structured)



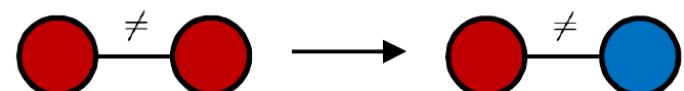
# Tree Decomposition

- Idea: create a tree-structured graph of mega-variables
- Each mega-variable encodes part of the original CSP
- Subproblems overlap to ensure consistent solutions

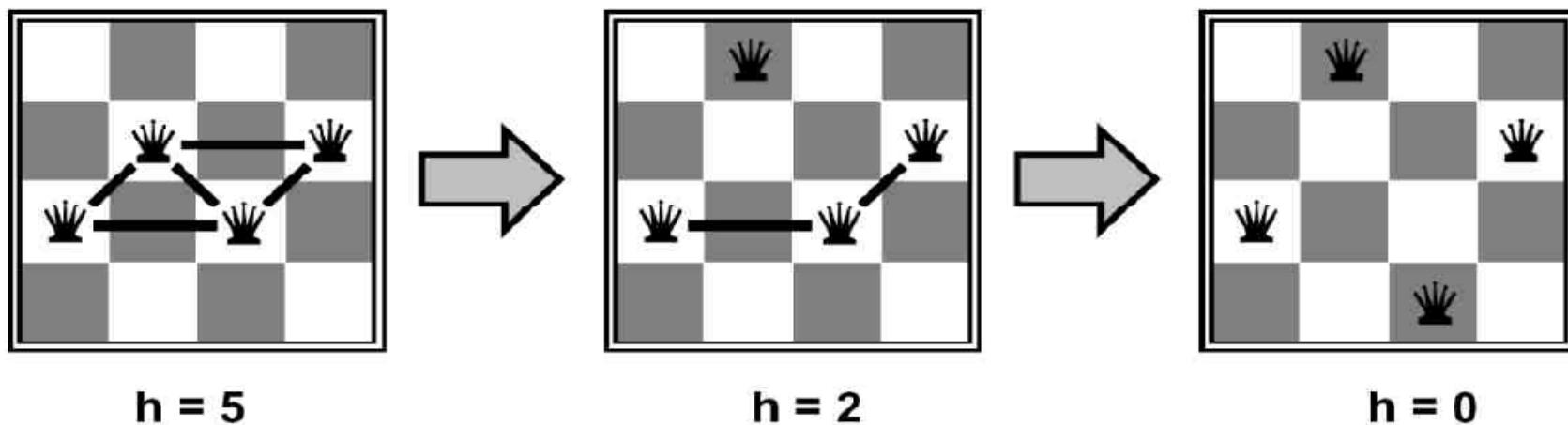


# Iterative Algorithms for CSPs

- Local search methods typically work with “complete” states, i.e., all variables assigned
- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No fringe! Live on the edge.
- Algorithm: While not solved,
  - Variable selection: randomly select any conflicted variable
  - Value selection: min-conflicts heuristic:
    - Choose a value that violates the fewest constraints
    - I.e., hill climb with  $h(n) = \text{total number of violated constraints}$



# Example: 4-Queens



- States: 4 queens in 4 columns ( $4^4 = 256$  states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation:  $c(n) = \text{number of attacks}$

# 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
- Iterative min-conflicts is often effective in practice