Constraint Satisfaction Problems

What is search for?

- Model of the world: single agents, deterministic actions, fully observed state, discrete state space

- Search problems:

  (a) Plan: Finding sequences of actions that leads us to a goal

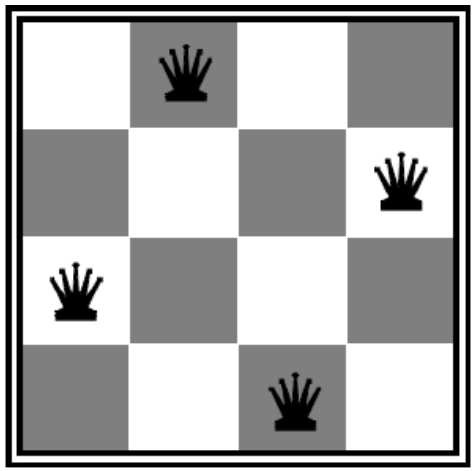
  – The path to the goal is the important thing
  – Paths have various costs, depths
  – Heuristics to guide, fringe to keep backups

  (b) Identification: finding proper assignments for 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: any function over states
  - Successor function can 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

- Allows useful general-purpose algorithms with more power than standard search algorithms

Example: Map-Coloring

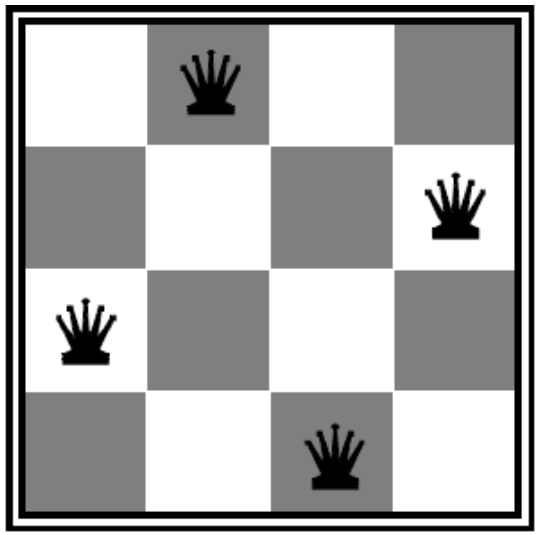
- Variables: $WA,\ NT,\ Q,\ NSW,\ V,\ SA,\ T$
- Domain: $D = \{red, green, blue\}$
- Constraints:
  adjacent regions must have
  different colors
  - Implicit $WA \neq NT$
  - Explicit $(WA, NT) \in \{(red, green), (red, blue), (green, red), \ldots\}$

- Solutions are assignments satisfying all constraints, e.g.:

$$\{WA = red, NT = green, Q = red,$$
$$NSW = green, V = red, SA = blue, T = green\}$$

Example: N-Queens

- Formulation 1:



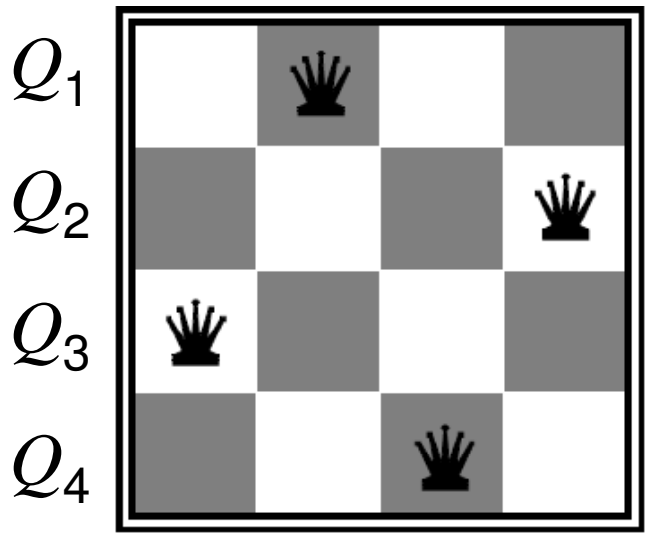
  - Variables: X_{ij}
  - Domains: $\{0,1\}$
  - Constraints

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

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

Example: N-Queens

- Formulation 2:
  - Variables: Q_k
  - Domains: $\{1, 2, 3, ..., N\}$
  - Constraints
    - Implicit $\forall i, j \;\; \text{non-threatening}(Q_i, Q_j)$
    - Explicit $(Q_1, Q_2) \in \{(1,3), (1,4), \ldots\}$

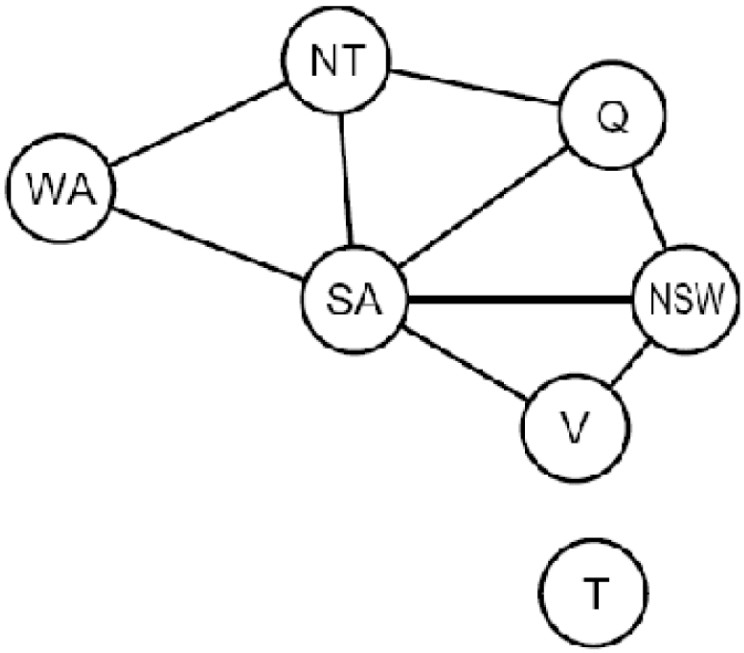


Constraint Graphs

- Binary CSP: each constraint relates (at most) two variables

- Binary constraint graph: nodes are variables, arcs show constraints

- Useful: General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!



Example: Cryptarithmetic

- Each letter represents a different digit
- Variables (circles):

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

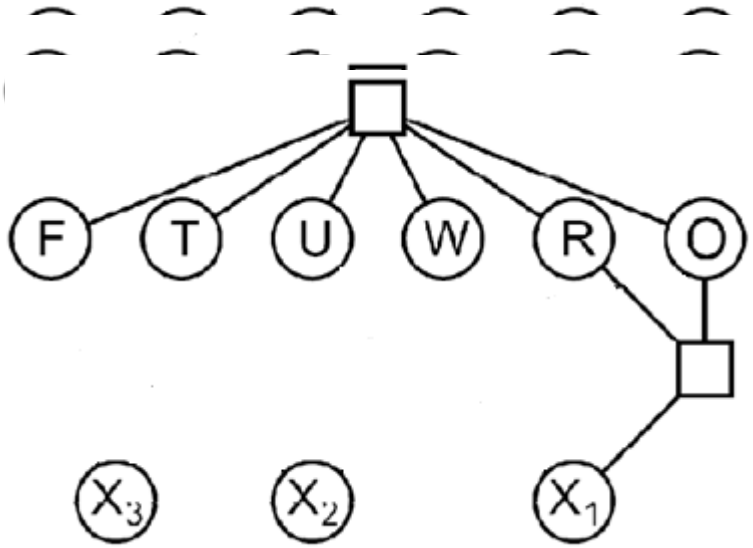
- Domains:

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

- Constraints (boxes):

$X_3\ \ X_2\ \ X_1$

$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$



Example: Cryptarithmetic

- Each letter represents a different digit
- Variables (circles):

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

- Domains:

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

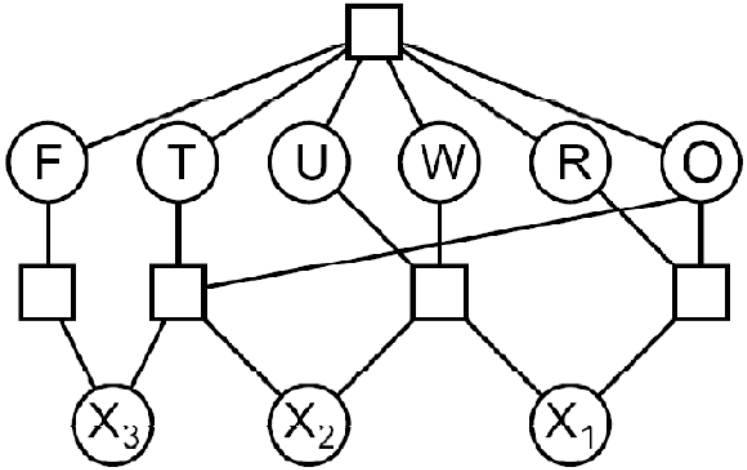
- Constraints (boxes):

  $\text{alldiff}(F, T, U, W, R, O)$

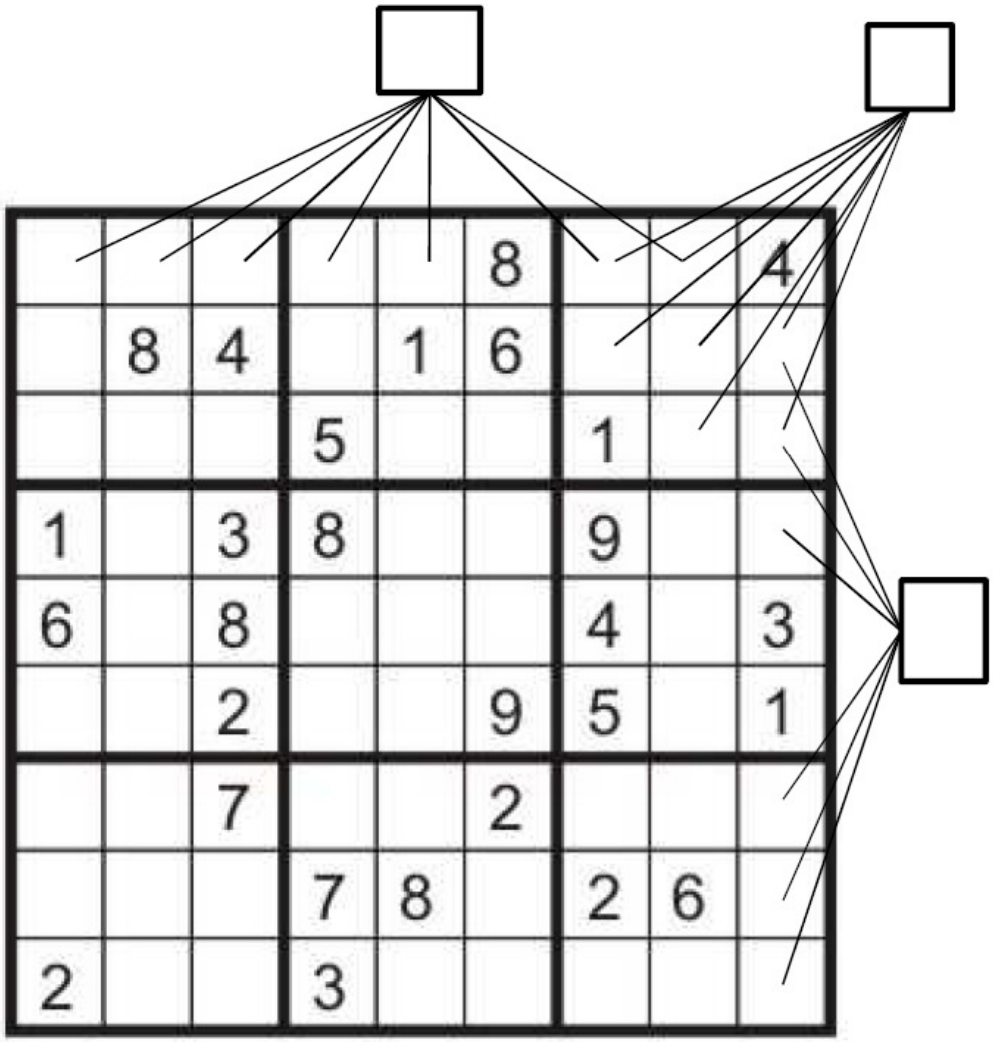
  $O + O = R + 10 \cdot X_1$

  $\cdots$

$$\begin{array}{cccc} X_3 & X_2 & X_1 & \\ & \mathsf{T} & \mathsf{W} & \mathsf{O} \\ + & \mathsf{T} & \mathsf{W} & \mathsf{O} \\ \hline \mathsf{F} & \mathsf{O} & \mathsf{U} & \mathsf{R} \end{array}$$



Example: Sudoku



- Variables:
  - Each (open) square
- Domains:
  - {1, 2, …, 9}
- Constraints:

  9-way alldiff for each row

  9-way alldiff for each column

  9-way alldiff for each region

  (or can have a bunch of pairwise inequality constraints)

Varieties of CSPs

- Discrete Variables
  - Finite domains
    - Size d domain 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 constraints undecidable

- Continuous variables
  - E.g., start/end times for Hubble Telescope observations
  - Linear constraints solvable in polynomial time by Linear Programming methods

Varieties of Constraints

- Varieties of Constraints
  - Unary constraints involve a single variable (equiv. to shrinking domains): : $SA \neq green$
  - Binary constraints involve pairs of variables: $SA \neq WA$
  - Higher-order constraints involve 3 or more variables:

    e.g., cryptarithmetic column constraints

- Preferences (soft constraints):
  - E.g., *red* is better than *green*
  - Often represented by a COST for each variable assignment
  - Constrained optimization problems solved by Optimization Search Methods
  - (We'll ignore these until we get to Bayes' nets)

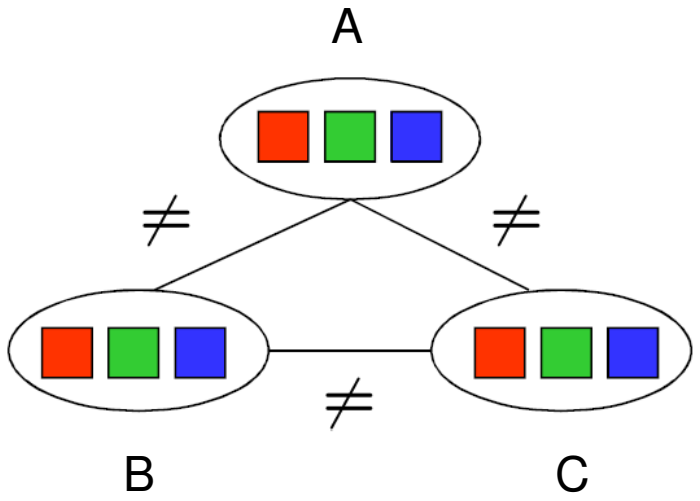
Real-World CSPs

- 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
- Fault diagnosis
- … lots more!

- Many real-world problems involve real-valued variables…

Reminder: CSPs

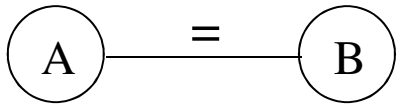
- CSPs:
  - Variables
  - Domains
  - Constraints
    - Implicit (provide code to compute)
    - Explicit (provide a subset of the possible tuples)
    - Unary / Binary / N-ary
- Goals:
  - Usually: find any solution
  - Find all, find best, etc



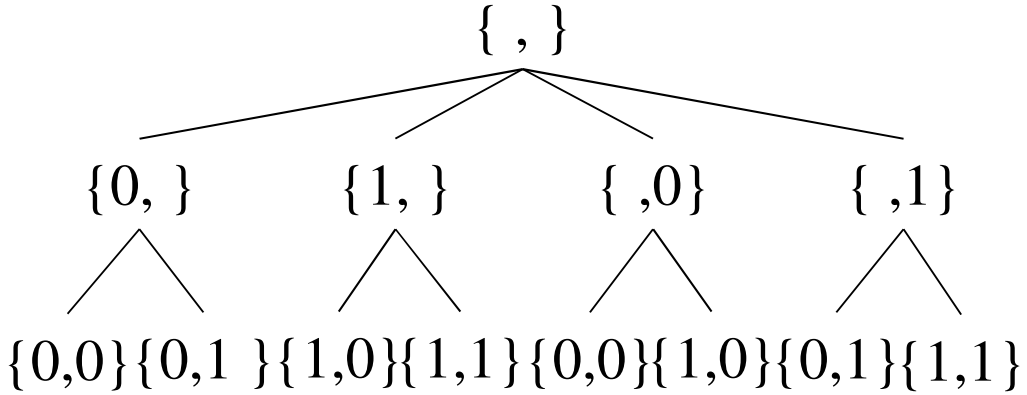
Standard Search Formulation

- Standard search formulation of CSPs (incremental)
- Let's start with the straightforward, dumb approach, then fix it
- 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

- Simplest CSP ever: two bits, constrained to be equal



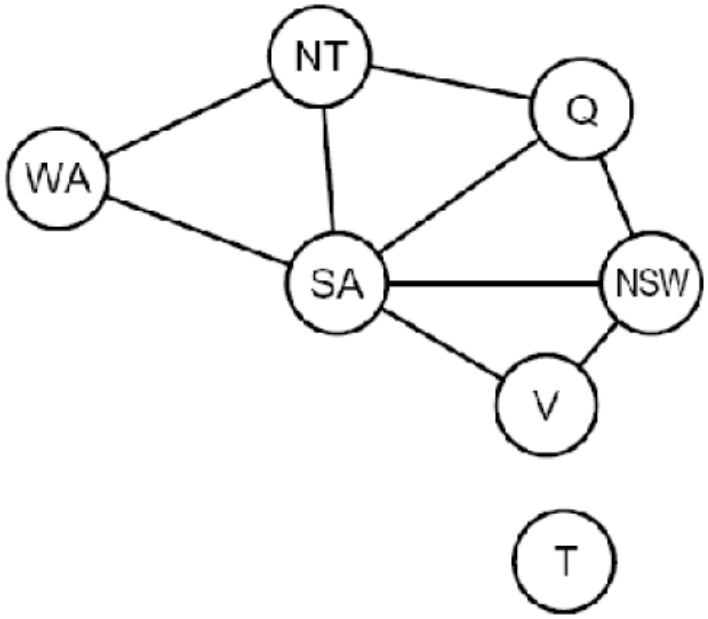
Problems:
1. High branching factor
   ($b = n.d$ at first level)
2. Duplications in the tree.



Search Methods

- What would BFS do?

  BFS will search
  every single assignment before
  possibly finding the goal
  in the last level



- What would DFS do?
  - What problems does this approach have?

  1) DFS does not detect the failures until the very end

  2) More subtle problem: even if the first level assignment is wrong, it tries all the permutation in the last levels before getting back to the first levels and fixing the wrong assignments there.

Backtracking Search

- Idea 1- Only consider a single variable at each point
  - 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
  - How many leaves are there?

Number of Leaves:
No ordering: $n!d^n$
Ordering: d^n

- Idea 2 - Constraint checking: Only allow legal assignments at each point
  - I.e. consider only values which do not conflict previous assignments
  - Might have to do some computation to figure out whether a value is ok
  - "Incremental goal test"
- Depth-first search for CSPs with these two improvements is called *backtracking search*
- Backtracking search is the basic uninformed algorithm for CSPs
- Can solve n-queens for $n \leq 25$

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 + var-ordering + fail-on-constraintViolation
- What are the choice points?

Improving Backtracking

- *General-purpose* ideas (heuristics) give huge gains in speed
  - ...but it is all still NP-hard

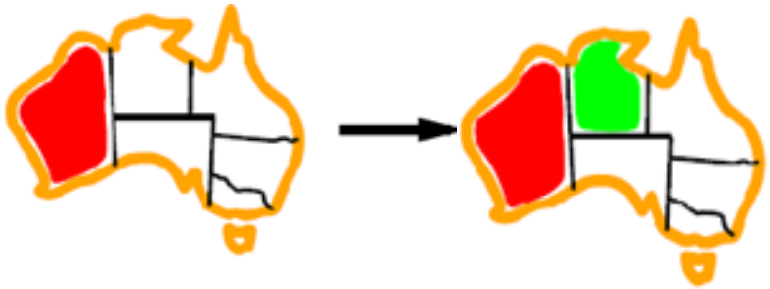
- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?

- Filtering: Can we detect inevitable failure early?

- Structure: Can we exploit the problem structure?

Ordering: Minimum Remaining Values

- Minimum remaining values (MRV):
  - Choose the variable with the fewest "legal" values
  - The variable that is most likely to cause a failure



  - Minimizes the number of nodes in the search tree by pruning larger parts of the tree earlier

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

Ordering: Degree Heuristic

- Degree heuristic:
  - Choose the variable involved in the largest number of constraints on other unassigned variables

  - Reduces branching factor on future choices

- *MRV* more powerful guide,

  but *degree* heuristic useful as a tie-breaker

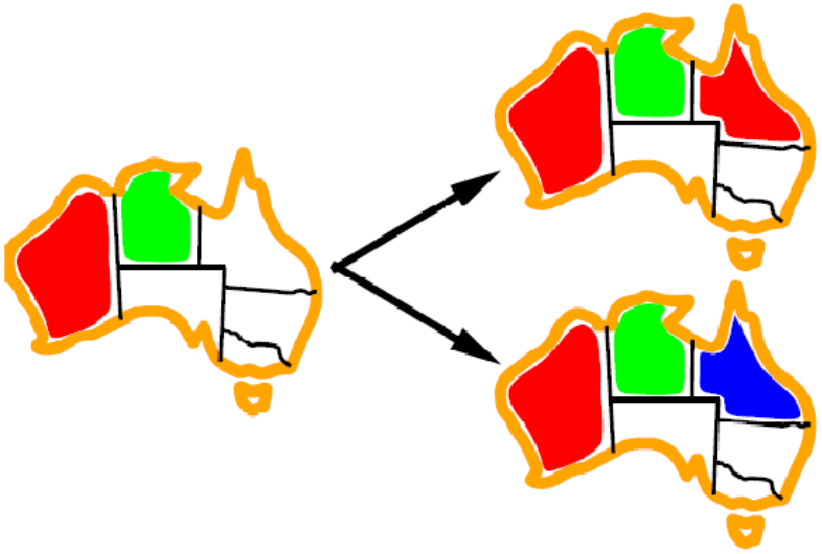
Ordering: Least Constraining Value

- Given a choice of variable:
  - Choose the least constraining value
  - The one that rules out the fewest values in the remaining variables
  - Leaves the max flexibility for subsequent variables
  - Note that it may take some computation to determine this!

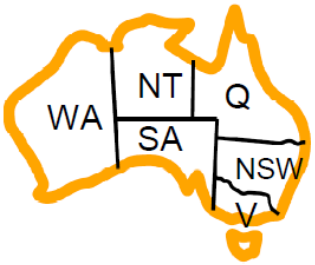
- Why least rather than most?

- "fail-last" ordering

- Combining these heuristics makes 1000 queens feasible



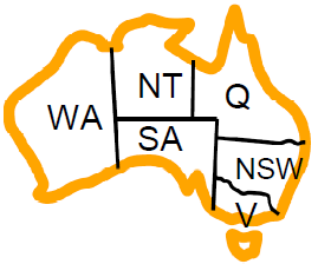
Filtering: Forward Checking



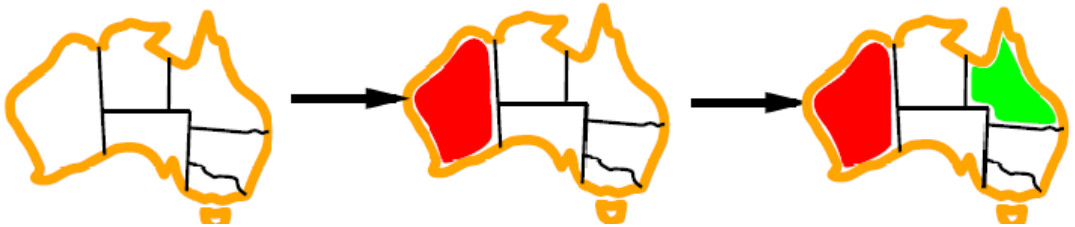
Can we detect inevitable failure early?

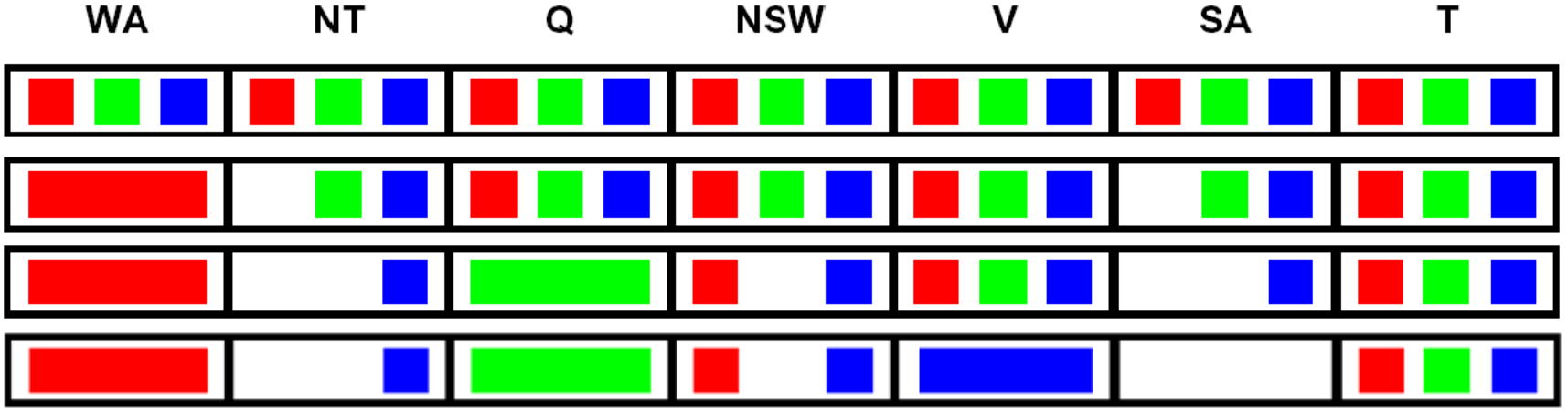
- Idea of Filtering: Keep track of remaining legal values for unassigned variables
- Forward checking: check only immediate constraints (that is, in map coloring update domains of only adjacent variables)
- Terminate when any variable has no legal values

Forward Checking: Local Consistency



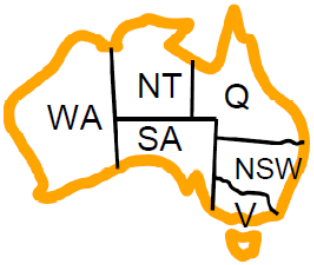
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures (or inconsistencies):



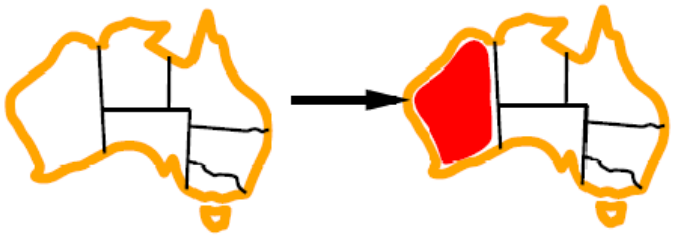


- NT and SA cannot both be blue!
- Why didn't we detect this early?

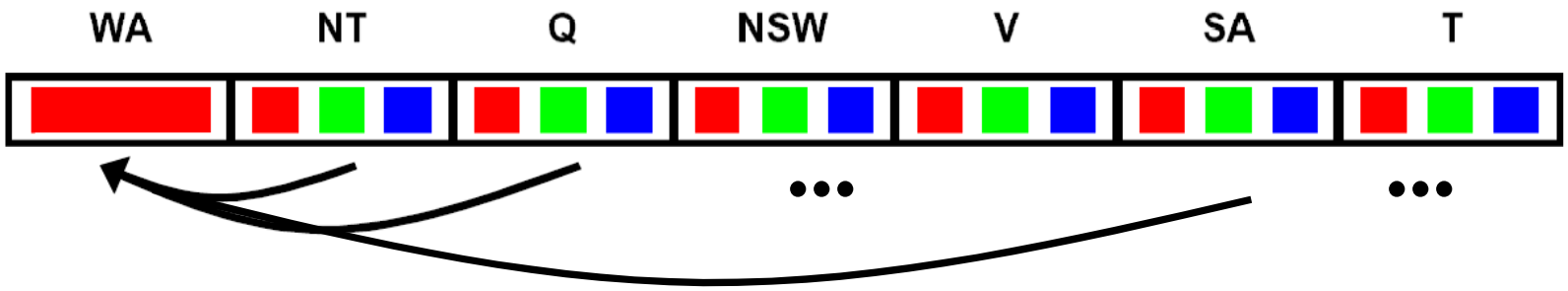
Consistency of an Arc



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

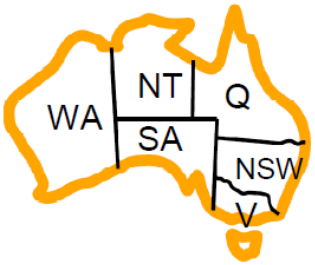


Delete from tail!

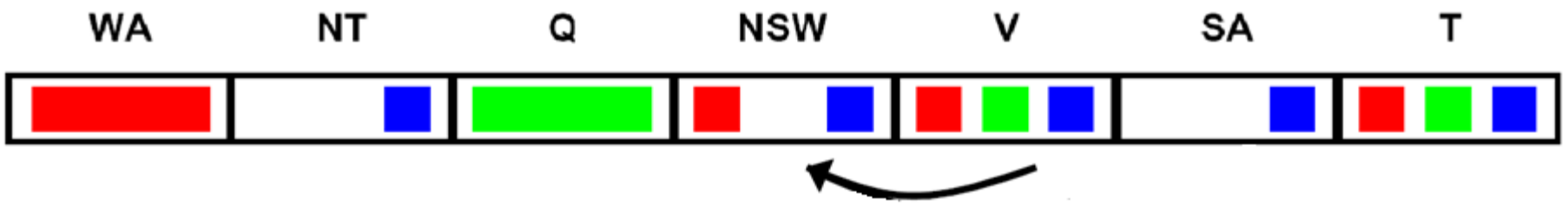


- Forward checking = Enforcing consistency of each arc pointing to the new assignment

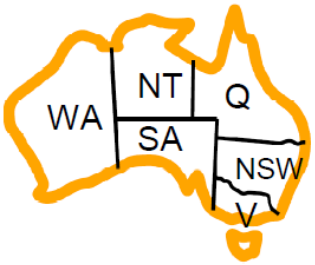
Consistency of an Arc



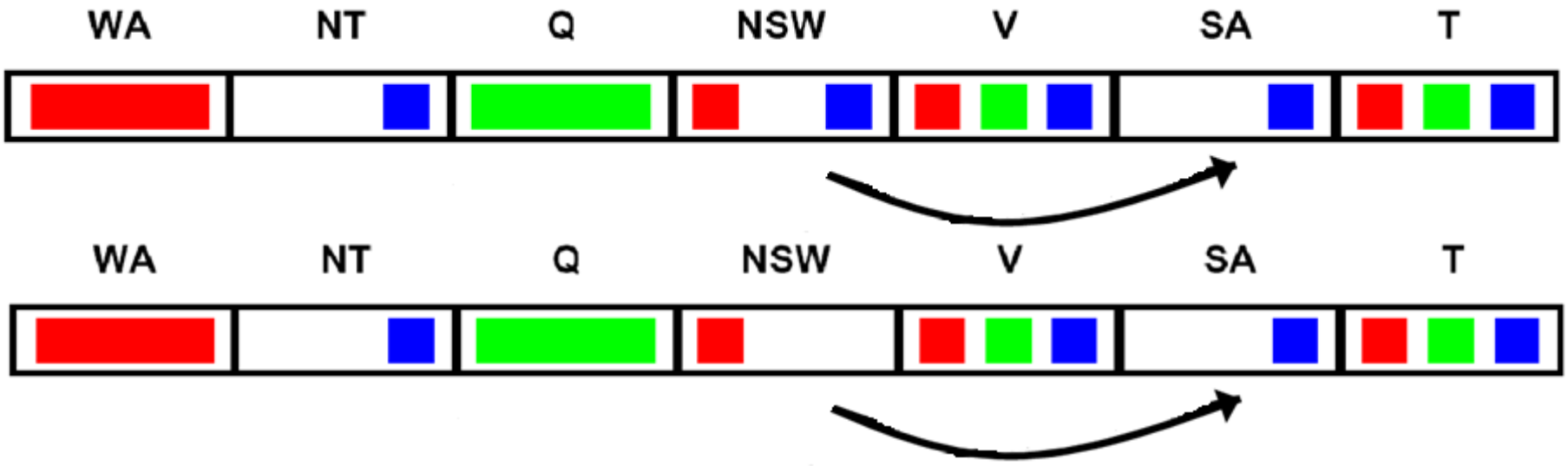
Delete from tail!



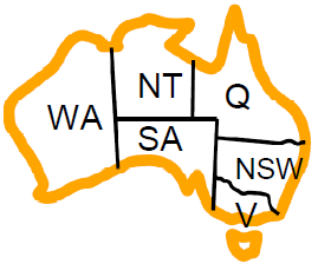
Consistency of an Arc



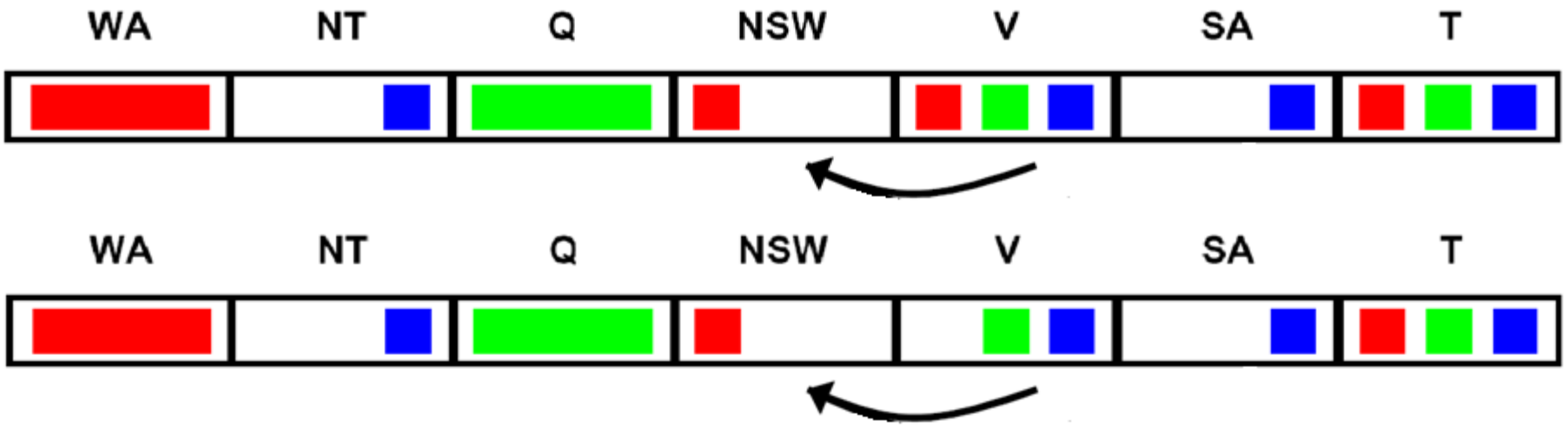
Delete from tail!



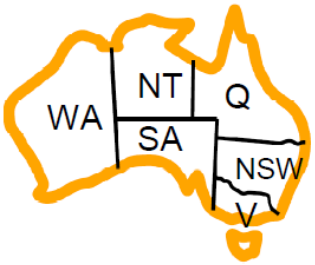
Consistency of an Arc



Delete from tail!

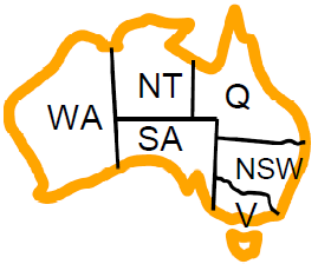


Consistency of an Arc



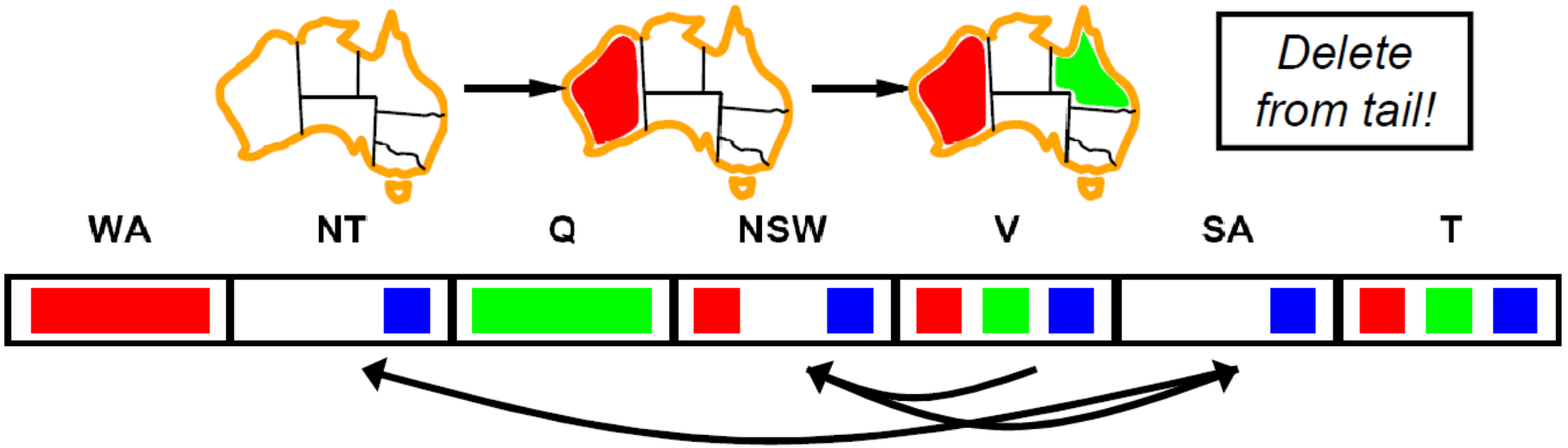
Delete from tail!

MAC Algorithm



- Maintaining Arc Consistency (MAC):

  Uses ***constraint propagation***: make sure all arcs are consistent:



- If X loses a value, neighbors of X need to be rechecked!
- MAC detects failure earlier than forward checking
- Arc consistency can be run as a preprocessor or after each assignment inside a search algorithm

Arc Consistency as Preprocessor

function AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise
  inputs: *csp*, a binary CSP with components (X, D, C)
  local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

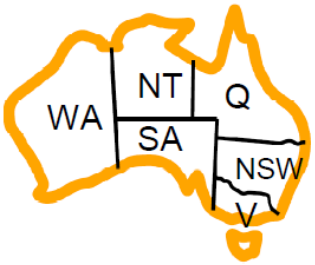
  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)
    if REVISE(*csp*, X_i, X_j) **then**
      if size of $D_i = 0$ **then return** *false*
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add (X_k, X_i) to *queue*
  return *true*

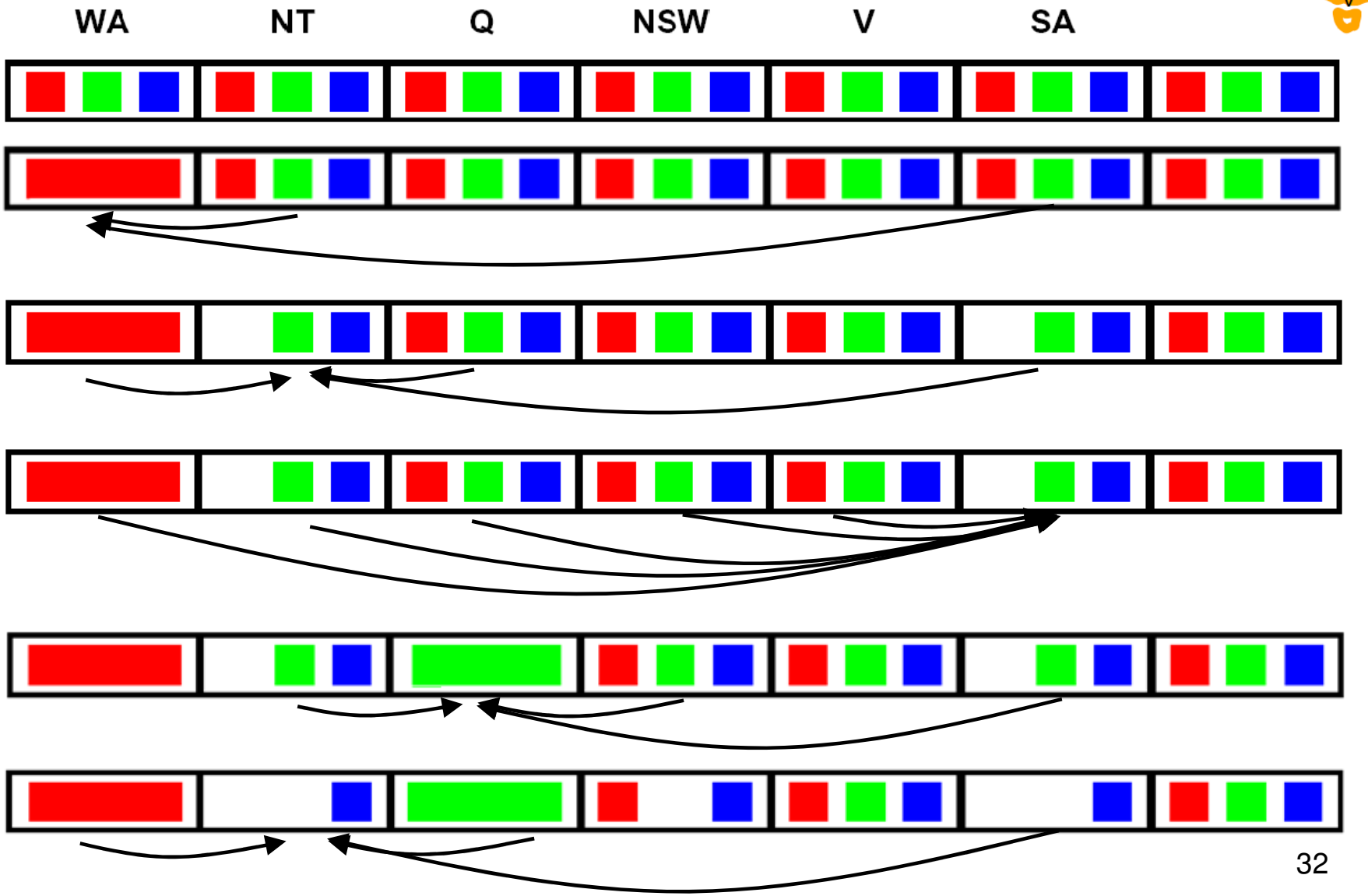
---

function REVISE(*csp*, X_i, X_j) **returns** true iff we revise the domain of X_i
  revised $\leftarrow$ *false*
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      revised $\leftarrow$ *true*
  return *revised*

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

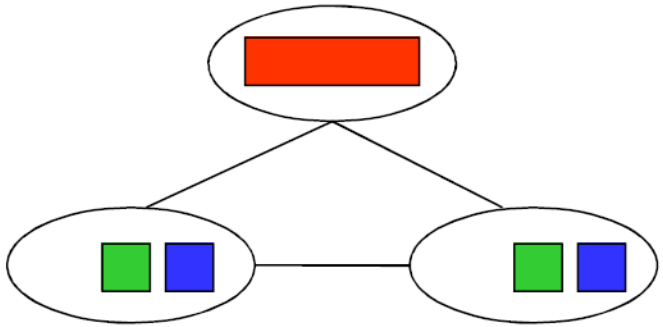
AC as Filtering in Backtracking Search (Example)

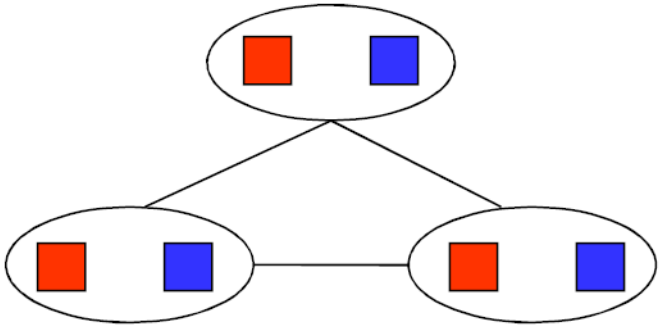




Limitations of Arc Consistency

- After running arc consistency :
  - Can have one solution left (no need for search)
  - Can have multiple solutions left (Search required)
  - Can have no solutions left (and not know it)

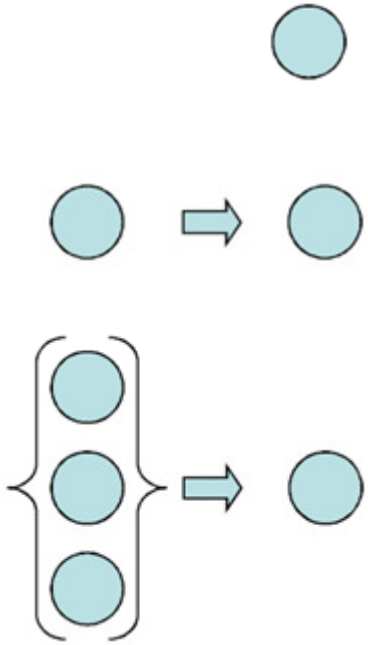




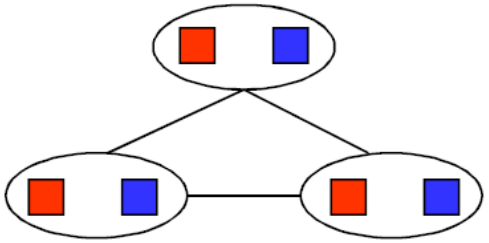
What went wrong here?

K-Consistency

- Increasing degrees of consistency
  - 1-Consistency (Node Consistency): Each single variable's domain has only values which meet that variable's unary constraints
  - 2-Consistency (Arc Consistency): For each pair of variables, any consistent assignment to one can be extended to the other
  - K-Consistency: For each k variables, any consistent assignment to k–1 variable can be extended to the k^{th} node.



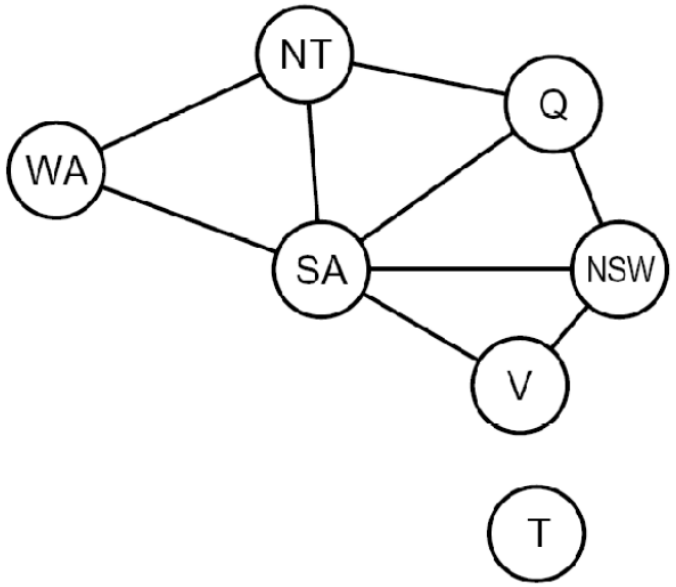
- Higher k more expensive to establish consistencies



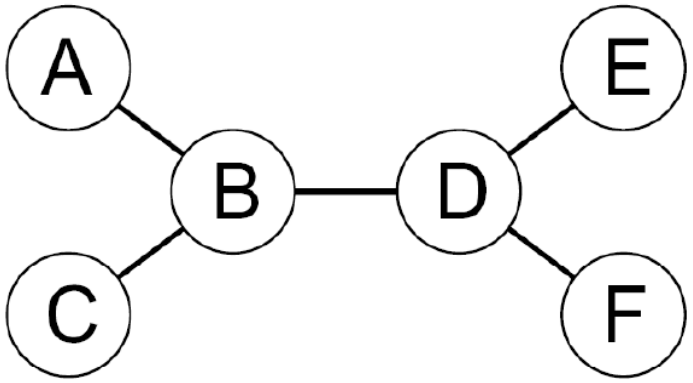
- In this course need to know the k=2 algorithm (AC-3)

Problem Structure

- Coloring Tasmania and mainland are independent subproblems
- Identifiable as connected components of constraint graph
- Suppose each subproblem has c variables out of n total variables
  - Worst-case solution cost
    with decomposition $O((n/c)(d^c))$, linear in n
    without decomposition $O(d^n)$, exponential in n
  - E.g., a boolean CSP ($d = 2$) with $n = 80$, $c = 20$
    $(4)(2^{20}) = 0.4$ seconds at 10 million nodes/sec
    $2^{80} = 4$ billion years at 10 million nodes/sec



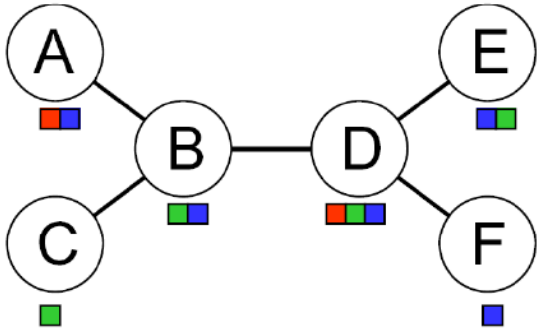
Tree-Structured CSPs

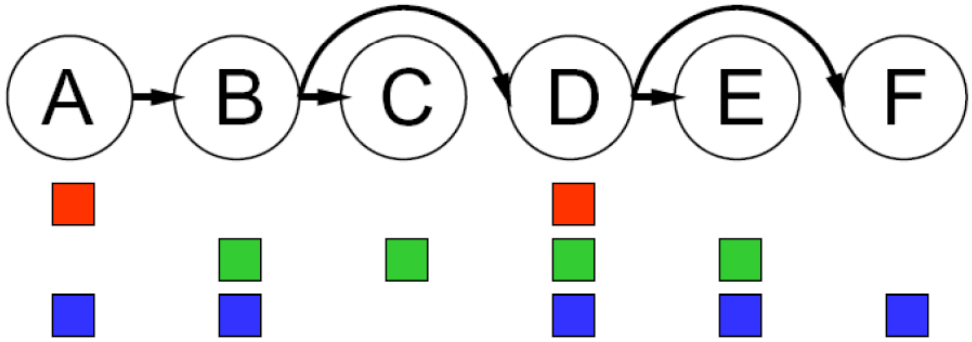


- **Theorem:** if the constraint graph has no loops (tree), the CSP can be solved in $O(nd^2)$ time
  - Compare to general CSPs, where worst-case time is $O(d^n)$

Solving Tree-Structured CSPs

- Topological Sort:
  - Choose a variable as root,
  - order variables (in a linear form) from root to leaves such that each variable appears after its parent in the tree

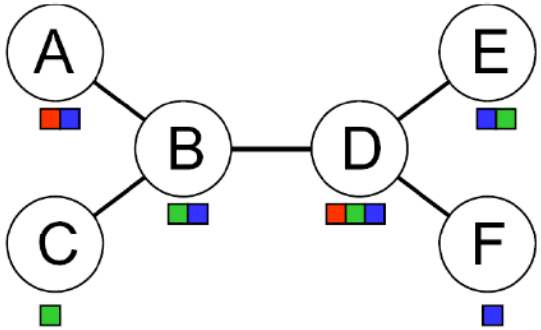


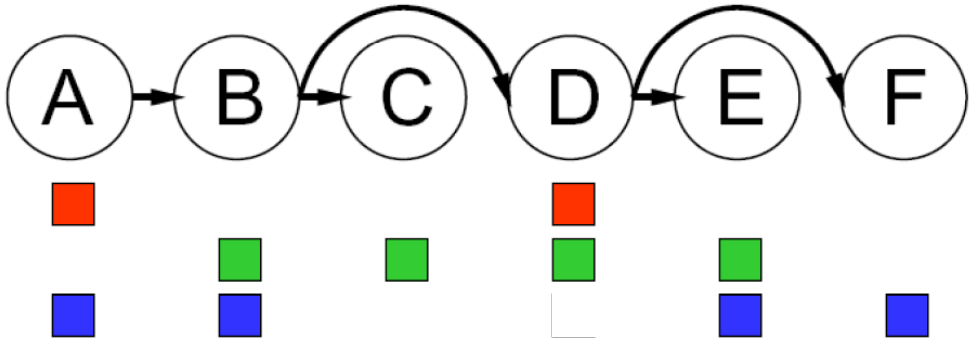


- Consistency establishment:
  - From F to B (in order) Apply RemoveInconsistent(*Parent*(X_i), X_i)

Solving Tree-Structured CSPs

- Topological Sort:
  - Choose a variable as root,
  - order variables (in a linear form) from root to leaves such that each variable appears after its parent in the tree

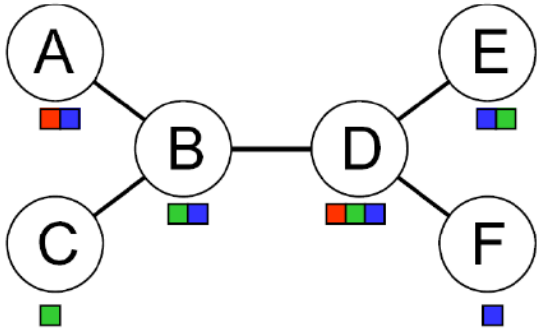


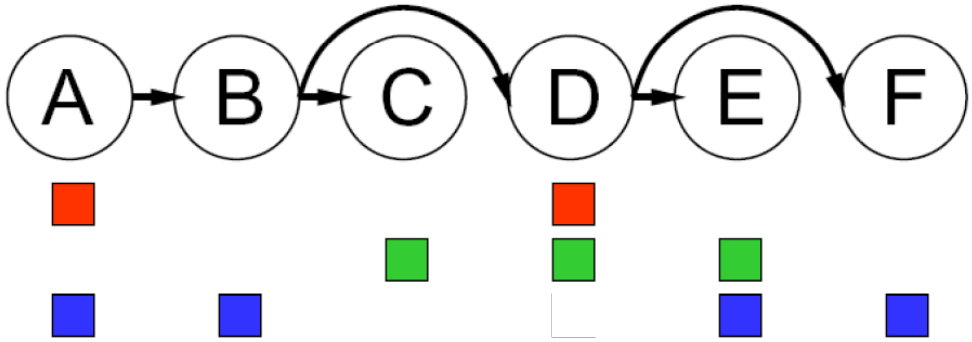


- Consistency establishment:
  - From F to B (in order) Apply RemoveInconsistent(*Parent*(X_i), X_i)

Solving Tree-Structured CSPs

- Topological Sort:
  - Choose a variable as root,
  - order variables (in a linear form) from root to leaves such that each variable appears after its parent in the tree

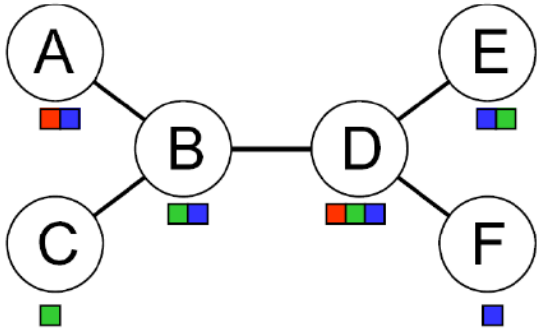


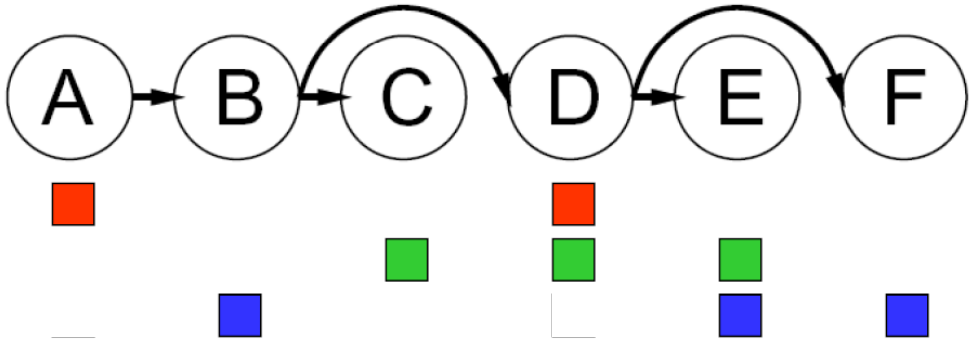


- Consistency establishment:
  - From F to B (in order) Apply RemoveInconsistent(*Parent*(X_i), X_i)

Solving Tree-Structured CSPs

- Topological Sort:
  - Choose a variable as root,
  - order variables (in a linear form) from root to leaves such that each variable appears after its parent in the tree

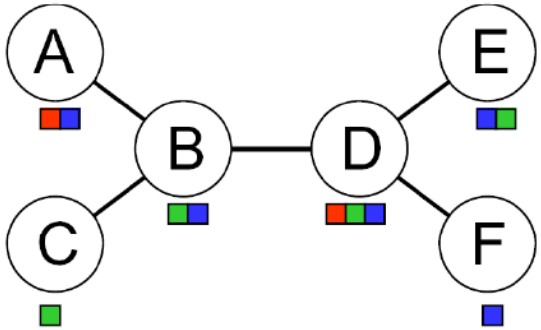


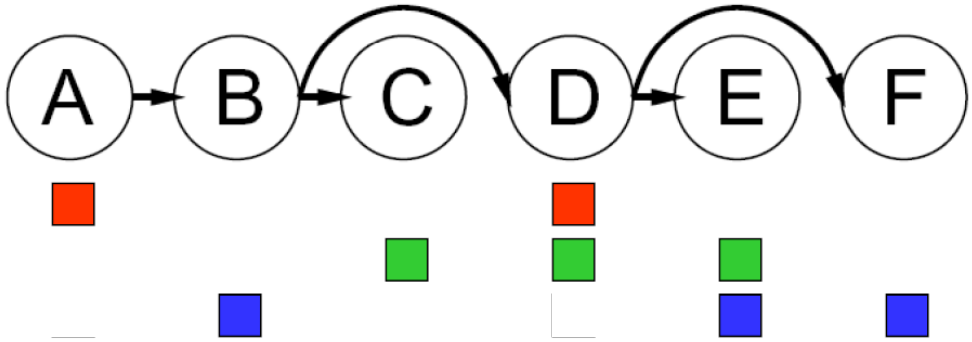


- Consistency establishment:
  - From F to B (in order) Apply RemoveInconsistent(*Parent*(X_i), X_i)

Solving Tree-Structured CSPs

- Topological Sort:
  - Choose a variable as root,
  - order variables (in a linear form) from root to leaves such that each variable appears after its parent in the tree



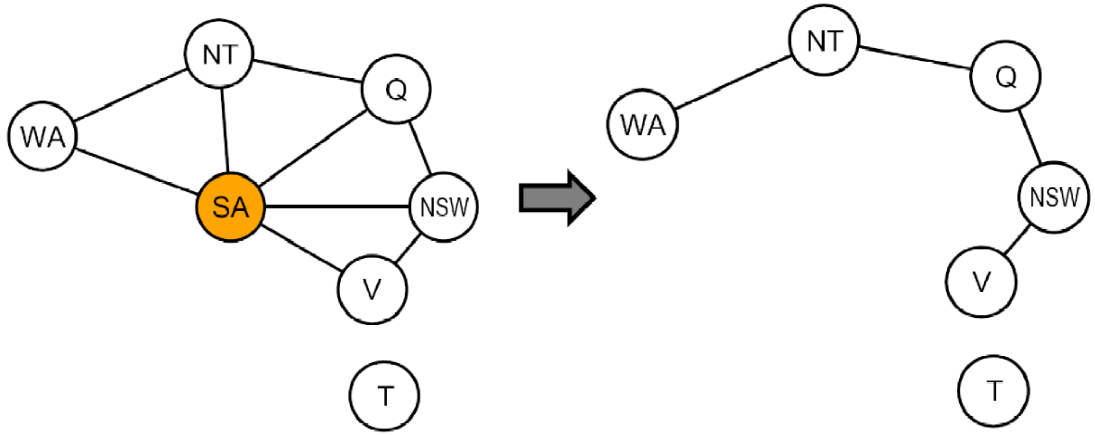


Runtime: Linear
$O(n\ d^2)$
(Why?)

- Consistency establishment:
  - From F to B (in order) Apply RemoveInconsistent($Parent(X_i)$, X_i)
- Solution:
  - From A to F (in order) select a value from current domain of each variable

Nearly Tree-Structured CSPs

Idea: assign values to some variables so that the remaining variables form a tree



- Conditioning:

  Fix a value for a variable, prune domains of its neighbors, solve the tree

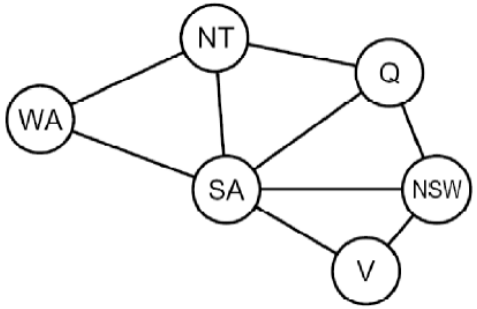
- Cutset conditioning:
  - Choose a subset S of variables such that constraint graph becomes a tree (cycle cutset)
  - For each consistent assignment to variables in S:
    - Remove from domain of tree nodes any value inconsistent with the assignment for S
    - If the tree CSP has a solution, return it with the assignment for S.

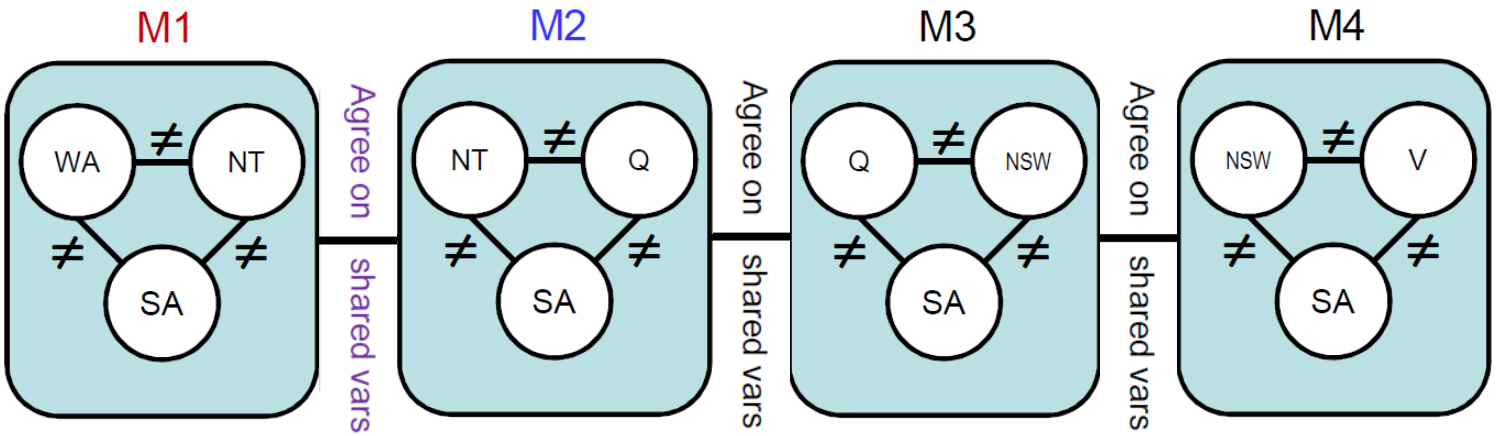
Cutset size c gives runtime $O(d^c (n-c) d^2)$ very fast for small c

Tree Decompositions*

- Create a tree-structured graph of overlapping subproblems (*mega-variable*)
- Find the solutions for each subproblem

  The set of solutions is the domain of a mega-variable
- Solve the CSP over subproblems using our efficient tree-structured CSP algorithm. (subproblems must agree on shared variables)





{(WA=r,SA=g,NT=b),
(WA=b,SA=r,NT=g),
…}

{(NT=r,SA=g,Q=b),
(NT=b,SA=g,Q=r),
…}

Agree: (M1,M2) ∈
{((WA=g,SA=g,NT=g), (NT=g,SA=g,Q=g)), …}

Requirements of Tree Decomposition

- A tree decomposition must satisfy the following three requirements:
  - Every variable in the original problem
    appears in at least one of the subproblems.
  - If two variables are connected by a constraint
    in the original problem, they must appear together
    (along with the constraint) in at least one of the subproblems.
  - If a variable appears in two subproblems in the tree,
    it must appear in every subproblem
    along the path connecting those subproblems

Iterative Algorithms for CSPs

- Local search methods typically work with "complete" states, i.e., all variables assigned

- To apply to CSPs:
  - Start with some assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No fringe! Live on the edge.

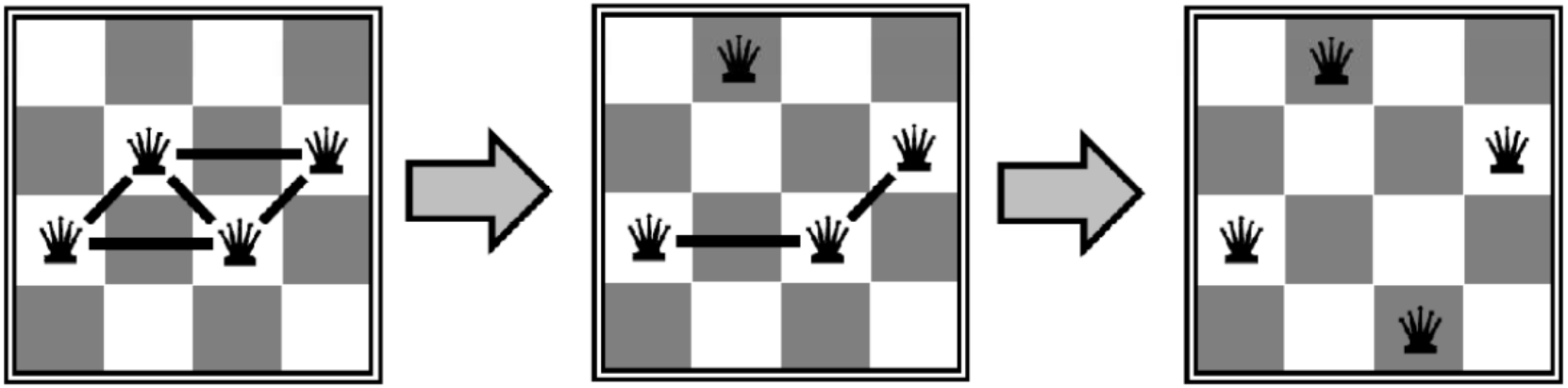
- Variable selection:

  One possible strategy: randomly select any conflicting variable

- Value selection:

  One specific strategy: use min-conflicts heuristic:
  - Choose a value that violates the fewest constraints

Example: 4-Queens

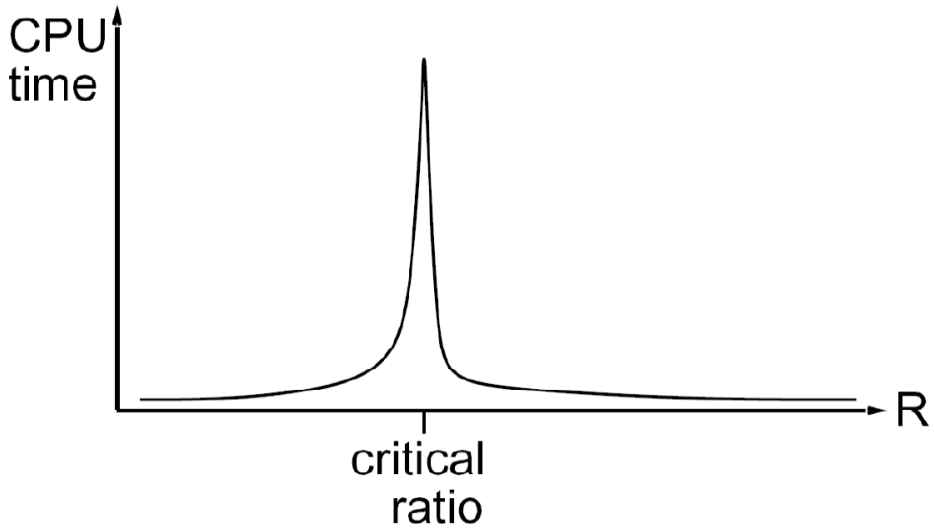


- Variables: Q_1, Q_2, Q_3, Q_4
- States: 4 queens in 4 columns ($4^4 = 256$ states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: $h(n)$ = number of attacks

Performance of Min-Conflicts

- Given random initial state, can solve *n*-queens in almost constant time for arbitrary *n* with high probability (e.g., n=10,000,000)
- The same appears to be true for any randomly-generated CSP *except* in a narrow range of the ratio

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$



CSP Summary

- CSPs are a special kind of search problems:
  - States defined by values of a fixed set of variables
  - Goal test defined by constraints on variable values
- Backtracking = depth-first search with one legal variable assigned per node
- Variable ordering and value selection heuristics help significantly
- Forward checking prevents assignments that guarantee later failure
- Constraint propagation (e.g., enforcing arc consistency) does additional work to constrain values and detect inconsistencies
- Constraint graphs allow for analysis of problem structure
- Tree-structured CSPs can be solved in linear time
- Iterative min-conflicts is usually effective in practice