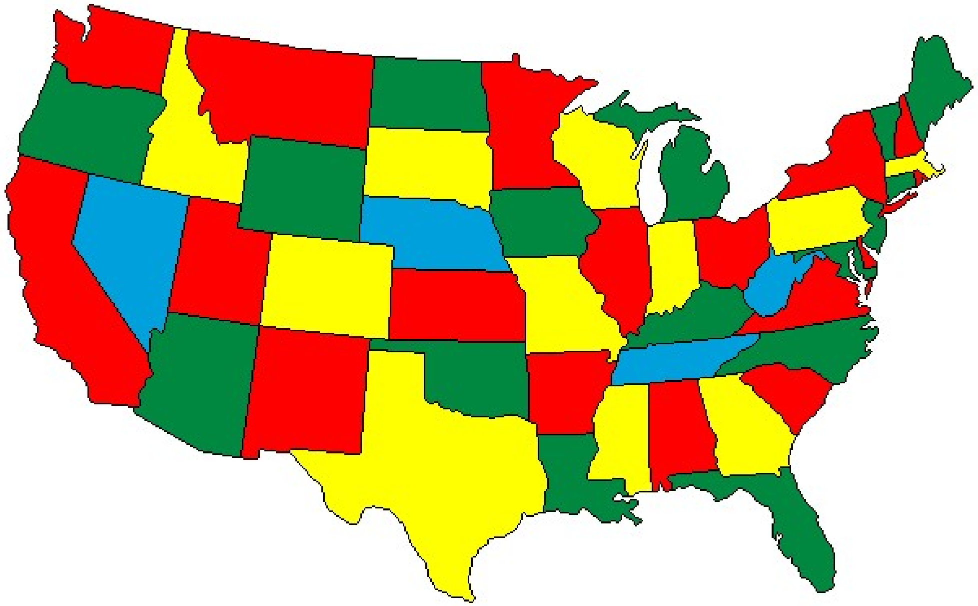
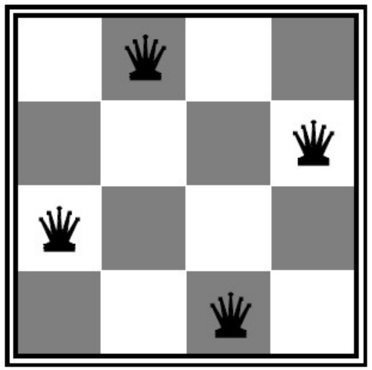
Constraint Satisfaction Problems





8			4		6			7
						4		
	1					6	5	
5		9		3		7	8	
				7				
	4	8		2		1		3
	5	2					9	
		1						
3			9		2			5

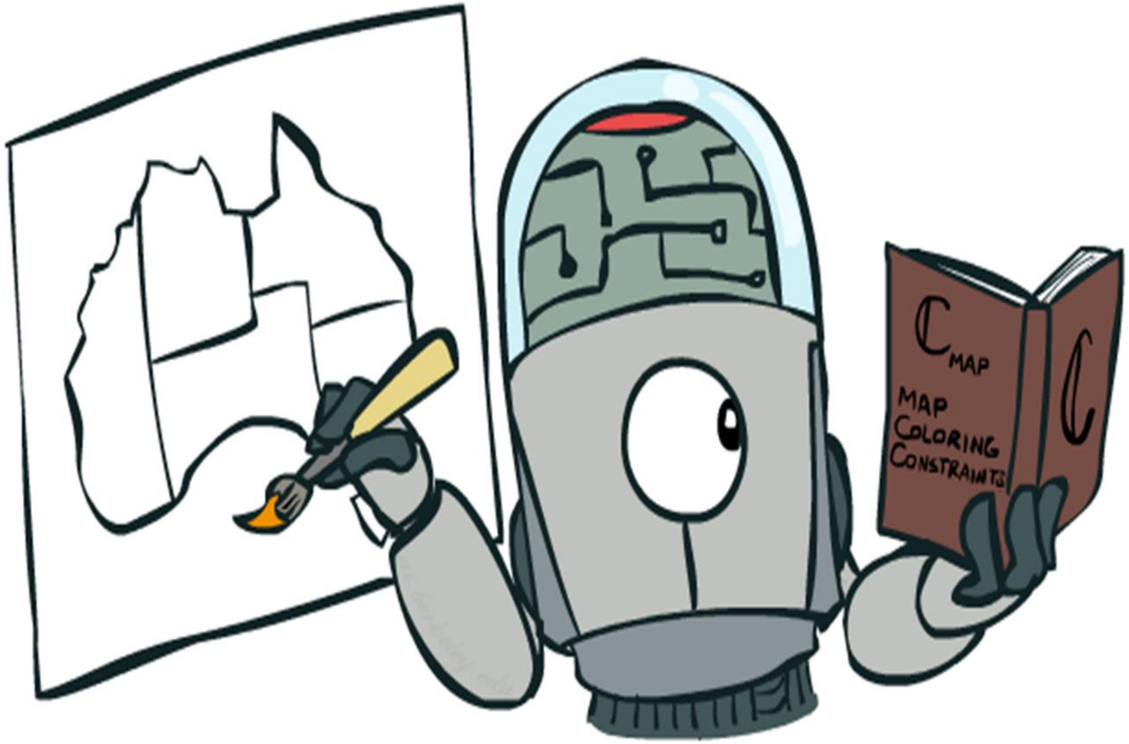
What is Search For?

- Assumptions about the world: 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 formulat
  - CSPs are specialized for identification problem

Constraint Satisfaction Problems



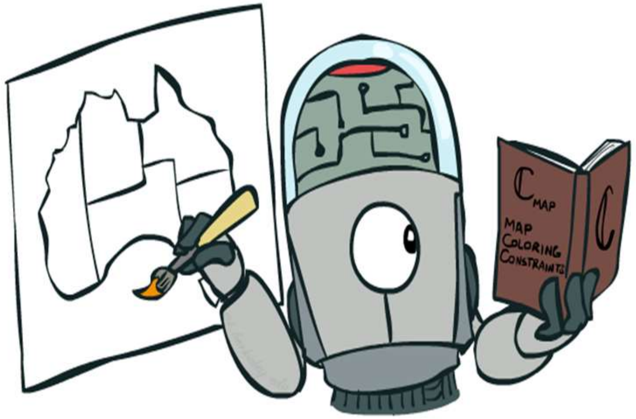
Constraint Satisfaction Problems

- 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

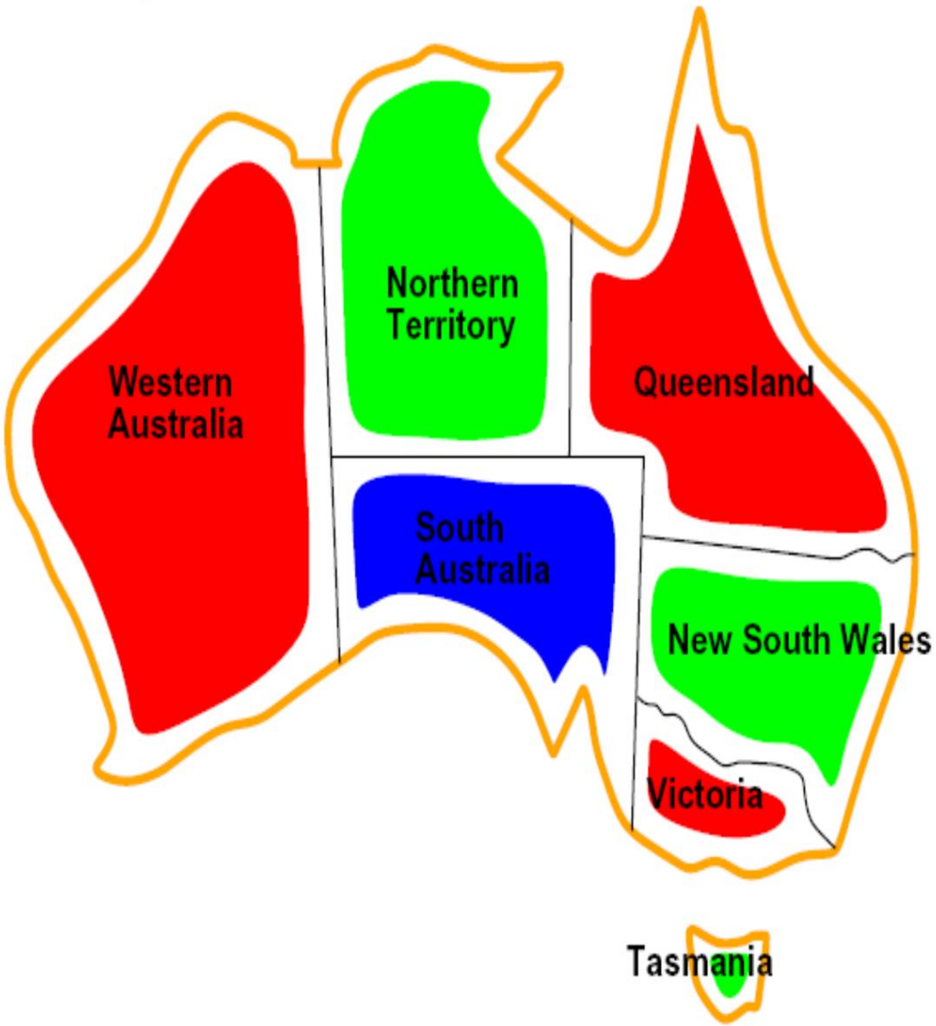
- Constraint satisfaction problems (CSPs):
  - A special subset of search problems
  - State is defined by variables X_i with values from a domain D (sometimes D depends on i)
  - Goal test is a set of constraints specifying allowable combinations of values for subsets of variables

- Simple example of a *formal representation language*

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



CSP Examples



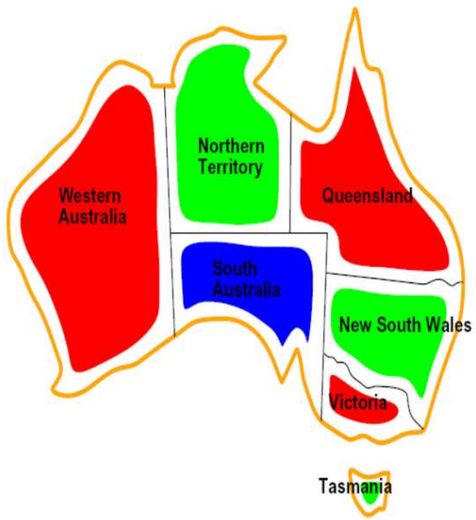
Example: Map Coloring

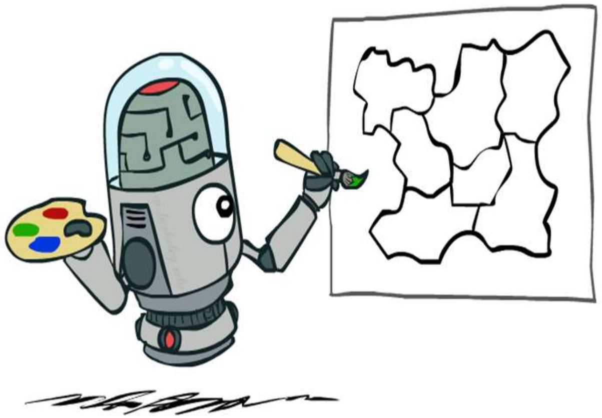
- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: $D = \{\text{red}, \text{green}, \text{blue}\}$
- Constraints: adjacent regions must have different colors

  Implicit: $WA \neq NT$

  Explicit: $(WA, NT) \in \{(\text{red}, \text{green}), (\text{red}, \text{blue}), \ldots\}$
- Solutions are assignments satisfying all constraints, e.g.:

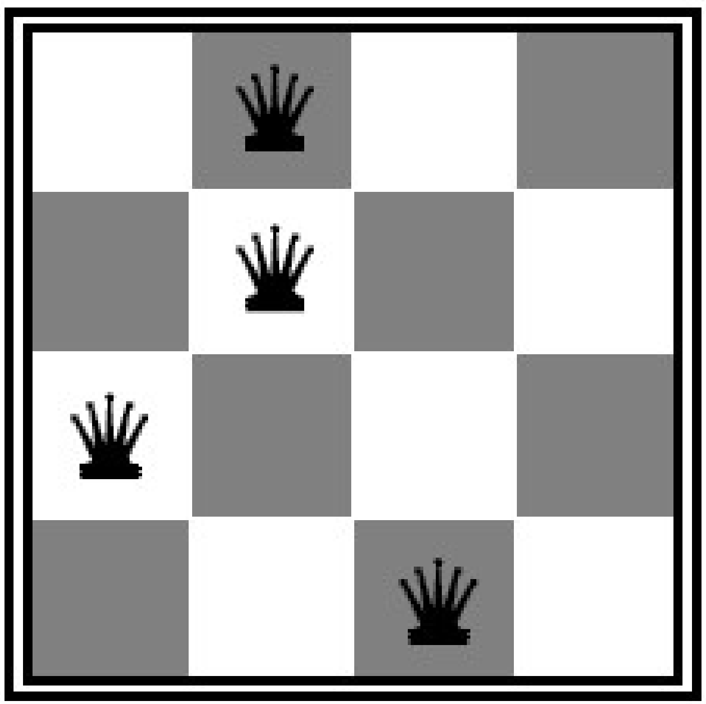
  {WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}

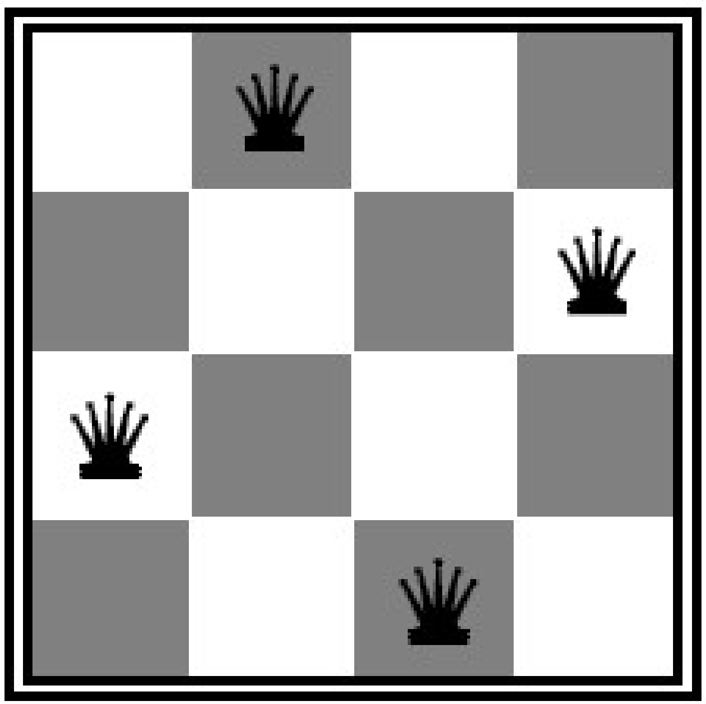




Example: *n*-queens problem

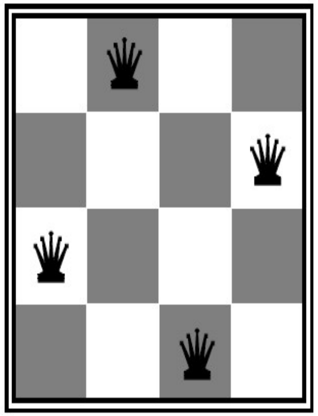
- Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

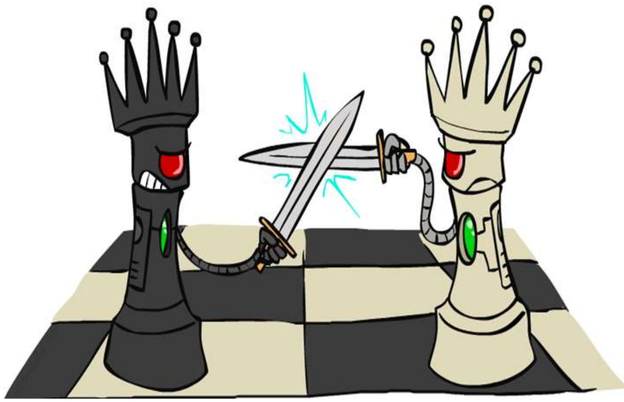




Example: N-Queens

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



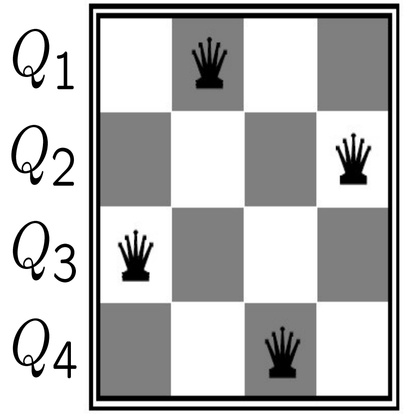


$$\forall i, j, k \ \ (X_{ij}, X_{ik}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \ \ (X_{ij}, X_{kj}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \ \ (X_{ij}, X_{i+k,j+k}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \ \ (X_{ij}, X_{i+k,j-k}) \in \{(0,0), (0,1), (1,0)\}$$

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

Example: N-Queens

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

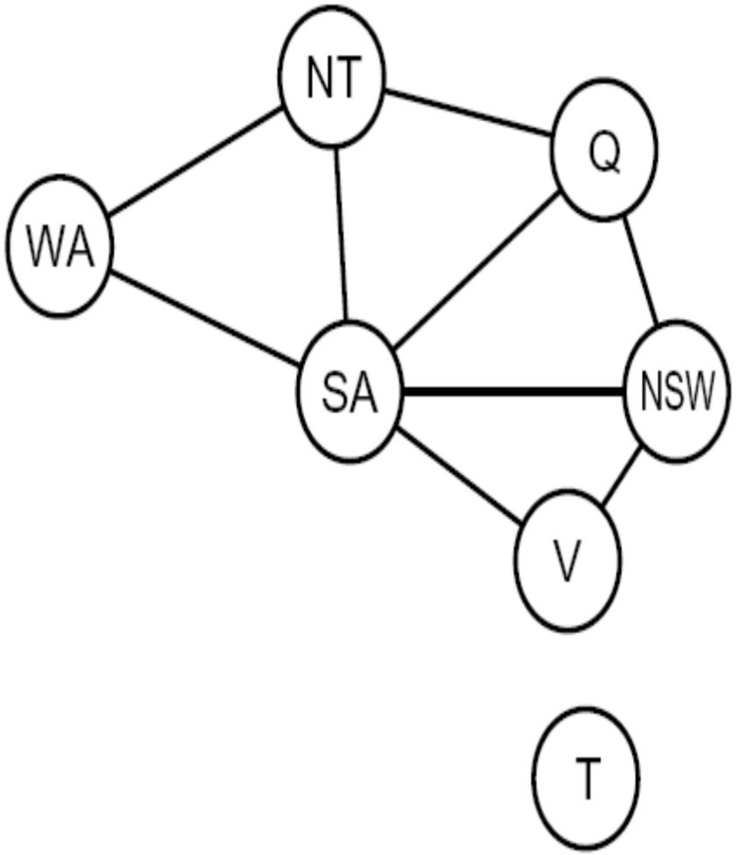


Implicit: $\forall i, j \;\; \text{non-threatening}(Q_i, Q_j)$

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

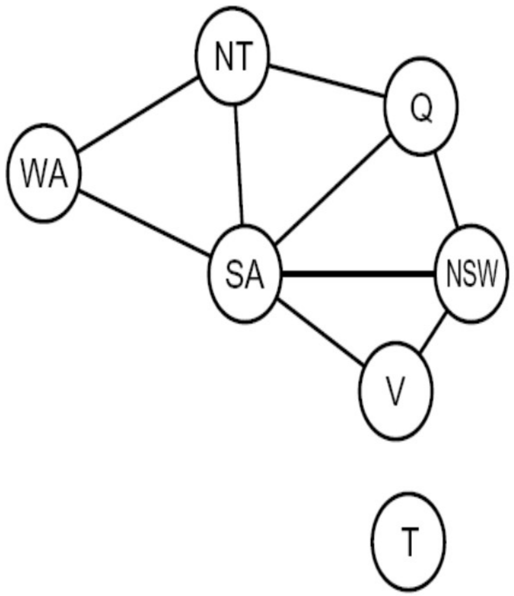
$\dots$

Constraint Graphs



Constraint Graphs

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

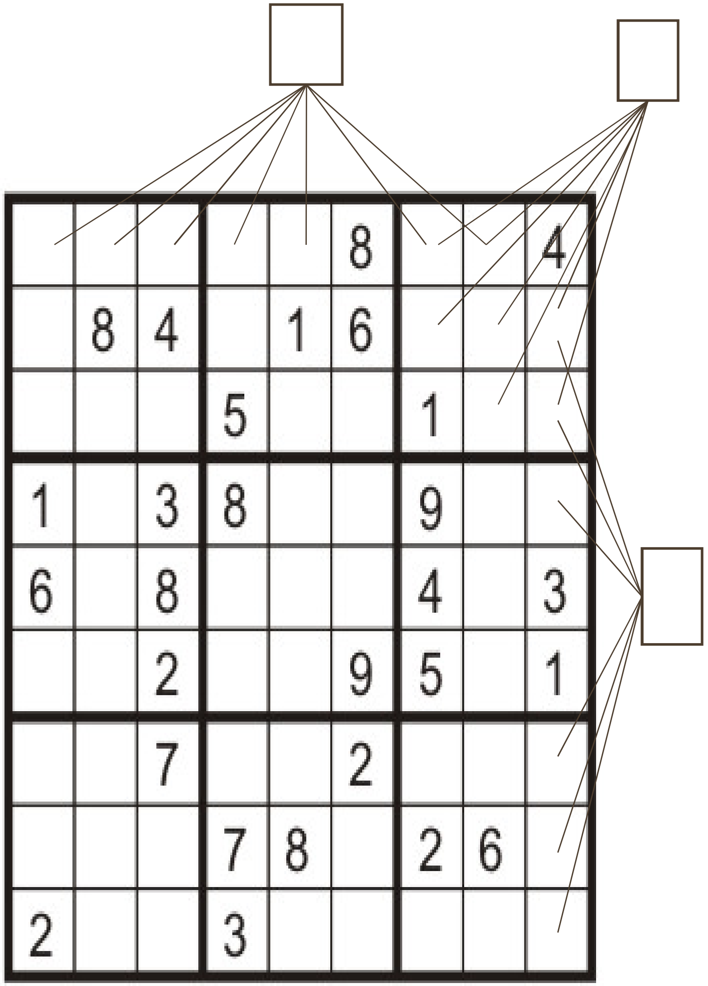


- Binary constraint graph: nodes are variables, arcs show constraints

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

[Demo: CSP applet (made available by

Example: Sudoku



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

9-way alldiff for each column

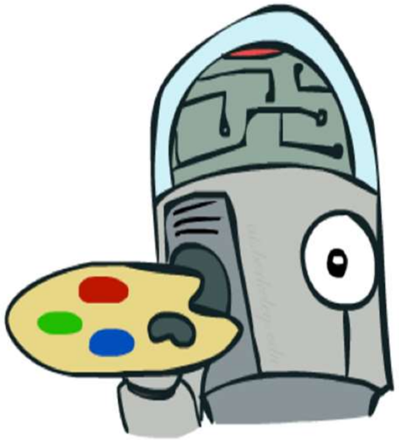
9-way alldiff for each row

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.:

$$SA \neq green$$

  - Binary constraints involve pairs of variables, e.g.:

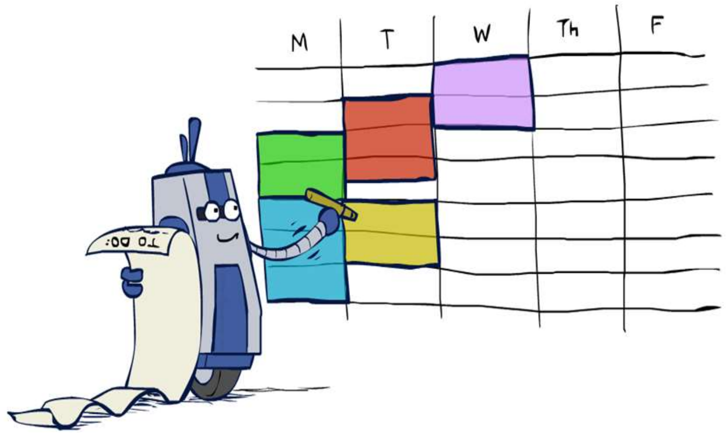
$$SA \neq WA$$

  - Higher-order constraints involve 3 or more variables:

- Preferences (soft constraints):
  - E.g., red is better than green
  - Often representable by a cost for each variable assignment
  - Gives constrained optimization problems
  - (We'll ignore these until we get to Bayes' nets)

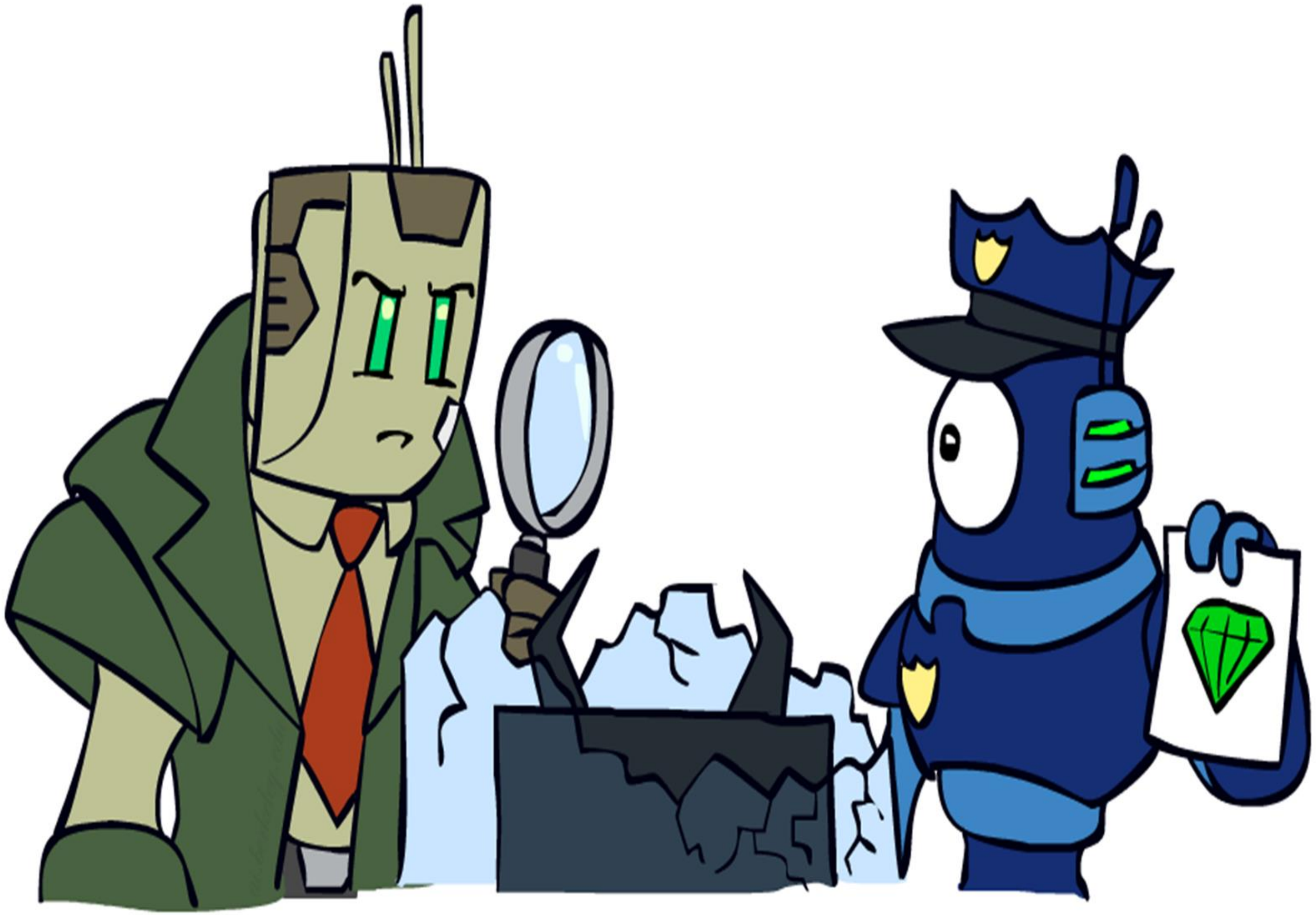
Real-World CSPs

- 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
- Circuit layout
- … lots more!



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

Solving CSPs

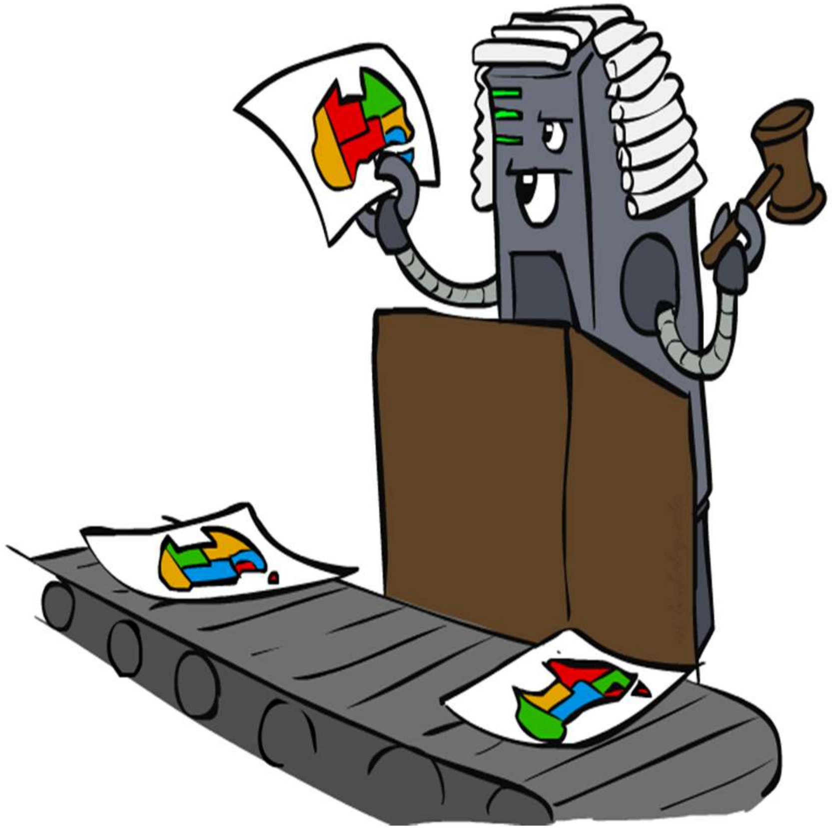


Standard Search Formulation

- Standard search formulation of CSPs

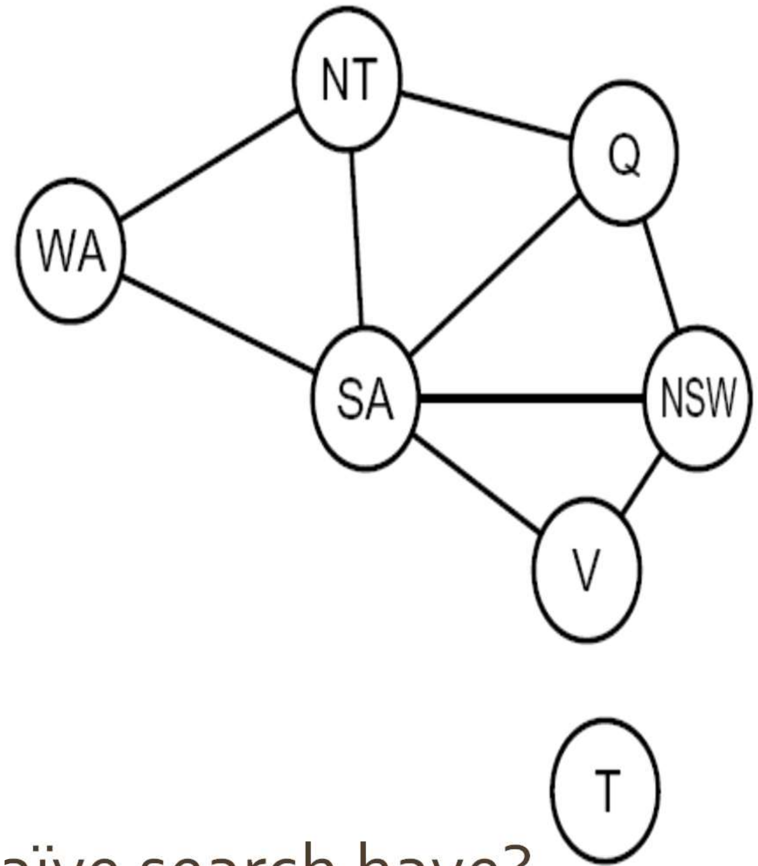
- States defined by the values assigned so far (partial assignments)
  - Initial state: the empty assignment, {}
  - Successor function: assign a value to an unassigned variable
  - Goal test: the current assignment is complete and satisfies all constraints

- We'll start with the straightforward, naïve approach, then improve it



Search Methods

- What would BFS do?



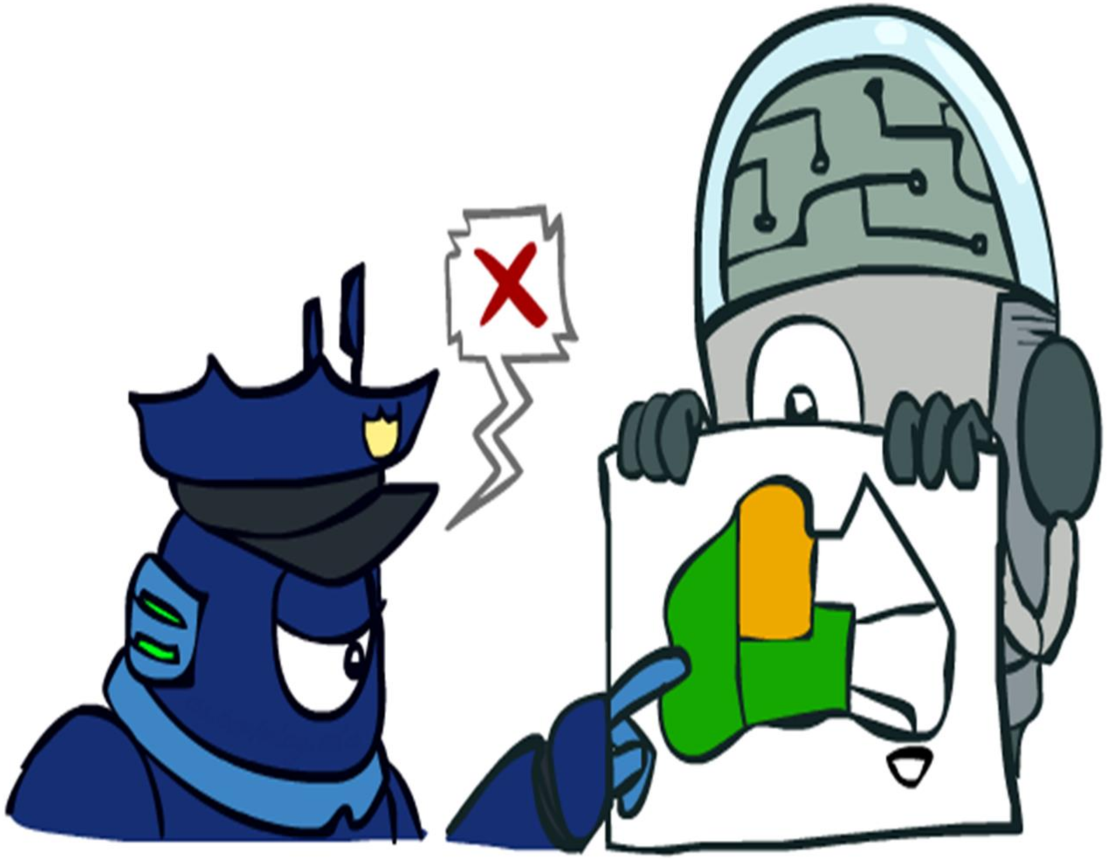
- What would DFS do?

- What problems does naïve search have?

[Demo: coloring -- ...]

Video of Demo Coloring -- DFS

Backtracking Search



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

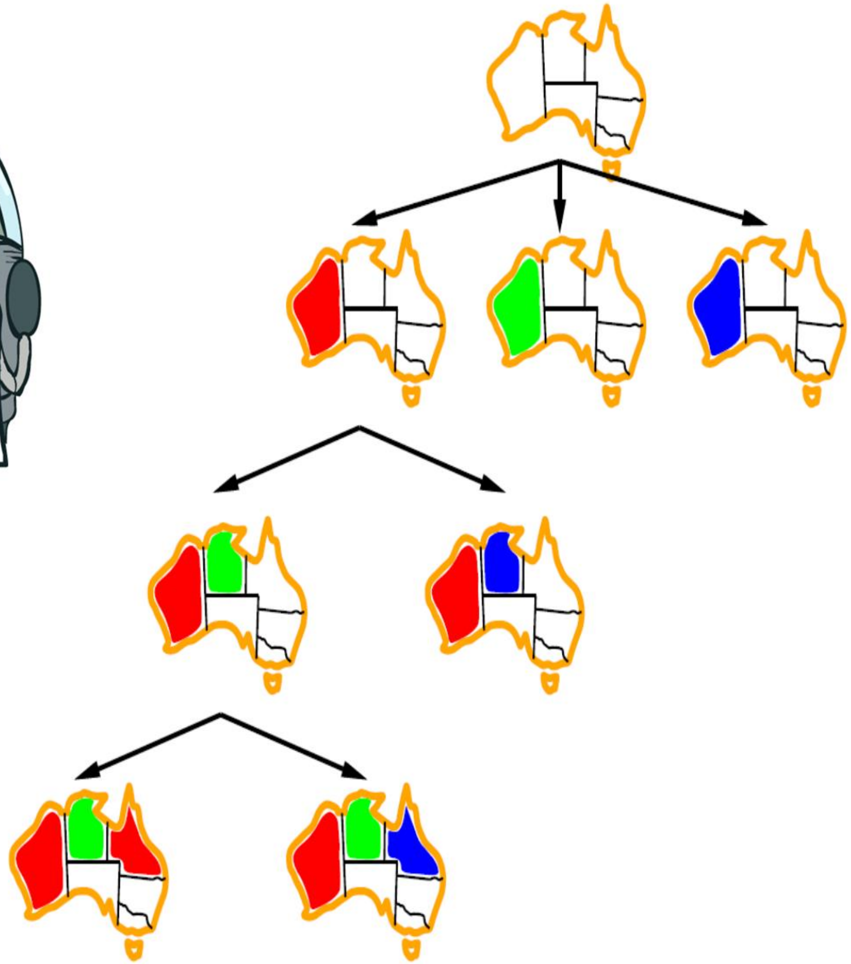
- Idea 2: Check constraints as you go
  - I.e. consider only values which do not conflict previous assignments
  - Might have to do some computation to check the constraints
  - “Incremental goal test”

- 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



[Demo: coloring – backtracking Simple]

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

[Demo: coloring – backtracking Complex]

Video of Demo Coloring – Backtracking

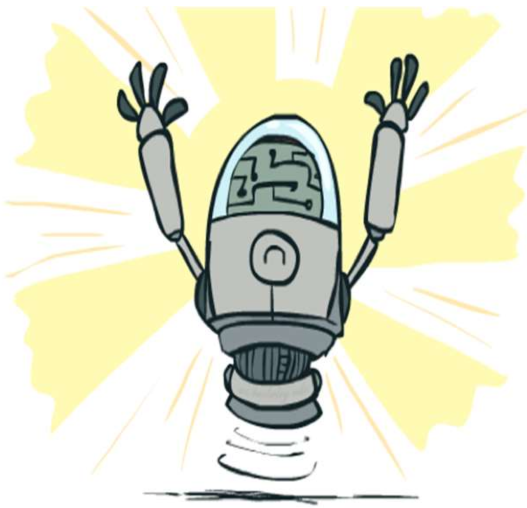
Improving Backtracking

- 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?

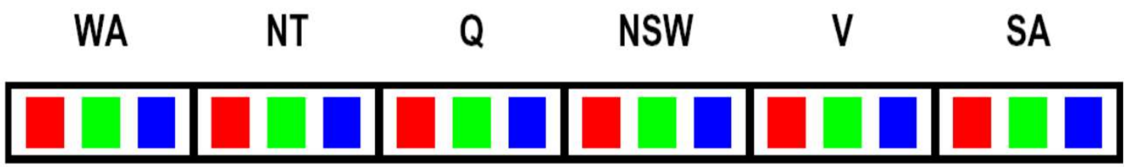
- Structure: Can we exploit the problem structure?



Filtering

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]

Which variable should be assigned next?

- **Most constrained variable:**
  - Choose the variable with the fewest legal values
  - A.k.a. **minimum remaining values** (MRV) heuristic

Which variable should be assigned next?

- **Most constrained variable:**
  - Choose the variable with the fewest legal values
  - A.k.a. **minimum remaining values** (MRV) heuristic

Which variable should be assigned next?

- **Most constraining variable:**
  - Choose the variable that imposes the most constraints on the remaining variables

Which variable should be assigned next?

- **Most constraining variable:**
  - Choose the variable that imposes the most constraints on the remaining variables

Given a variable, in which order should its values be tried?

- Choose the **least constraining value**:
  - The value that rules out the fewest values in the remaining variables

Given a variable, in which order should its values be tried?

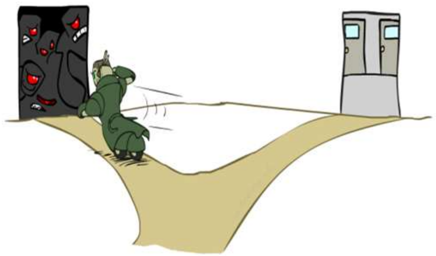
- Choose the **least constraining value**:
  - The value that rules out the fewest values in the remaining variables

Which assignment for Q should we choose?

Ordering: Minimum Remaining Values

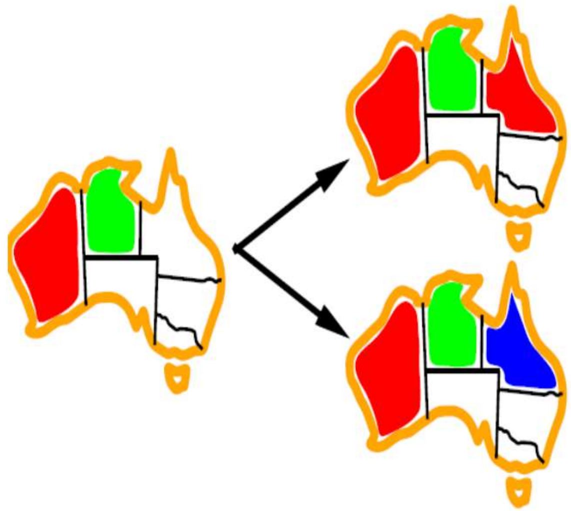
- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain

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



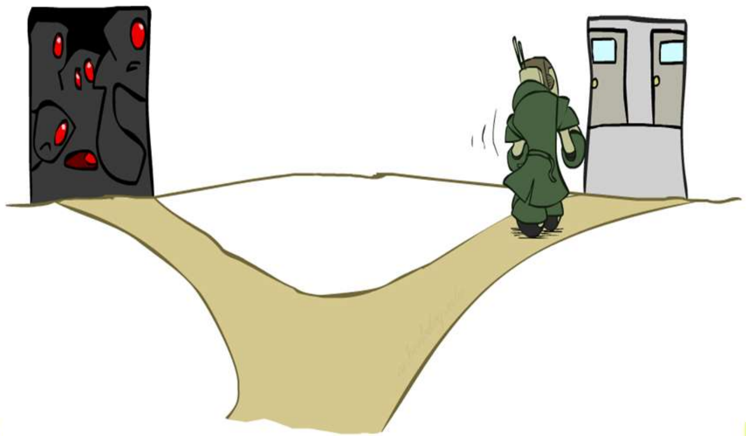
Ordering: Least Constraining Value

- 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)



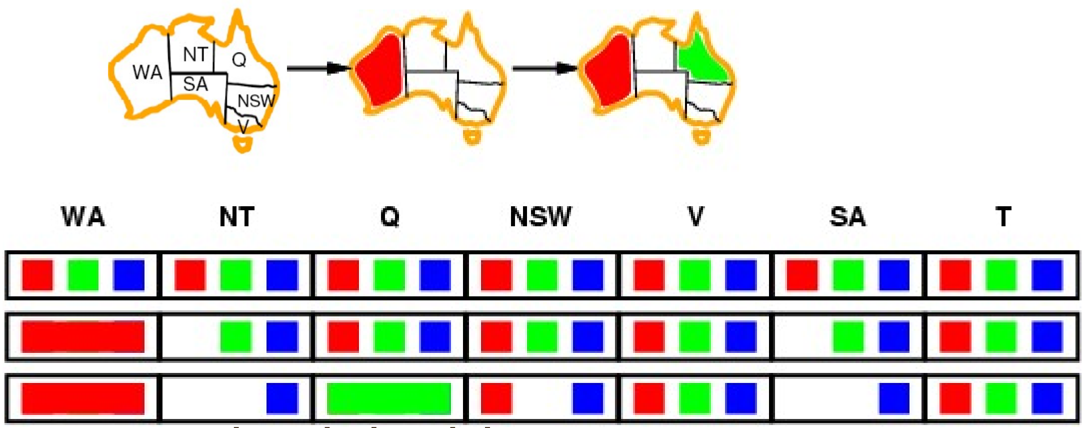
- Why least rather than most?

- Combining these ordering ideas makes
  1000 queens feasible



Constraint propagation

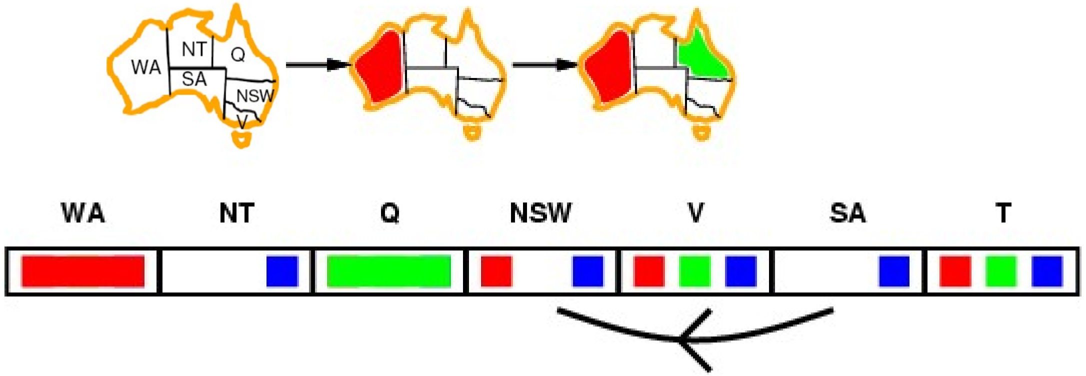
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures



- NT and SA cannot both be blue!
- **Constraint propagation** repeatedly enforces constraints *locally*

Arc consistency

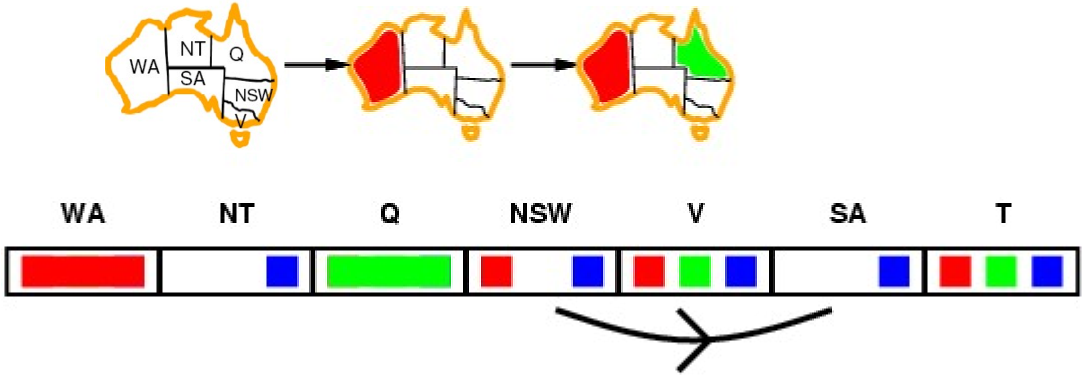
- Simplest form of propagation makes each pair of variables **consistent:**
  - $X \rightarrow Y$ is consistent iff for every value of X there is some allowed value of Y



Consistent!

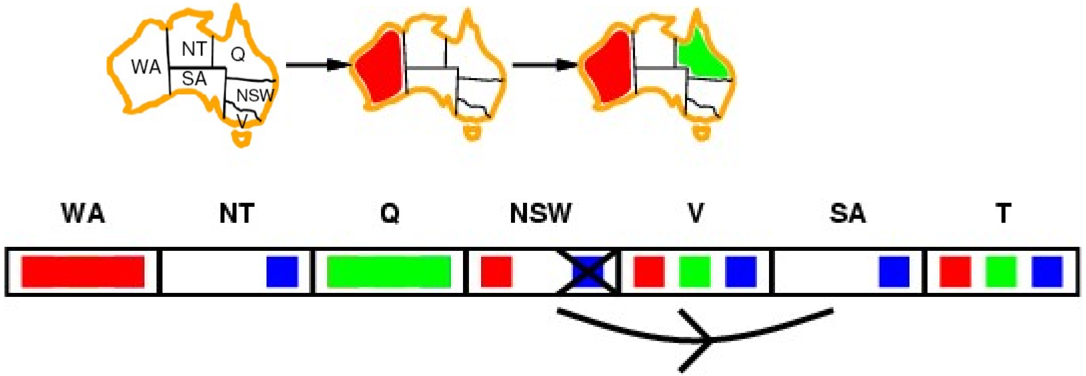
Arc consistency

- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*



Arc consistency

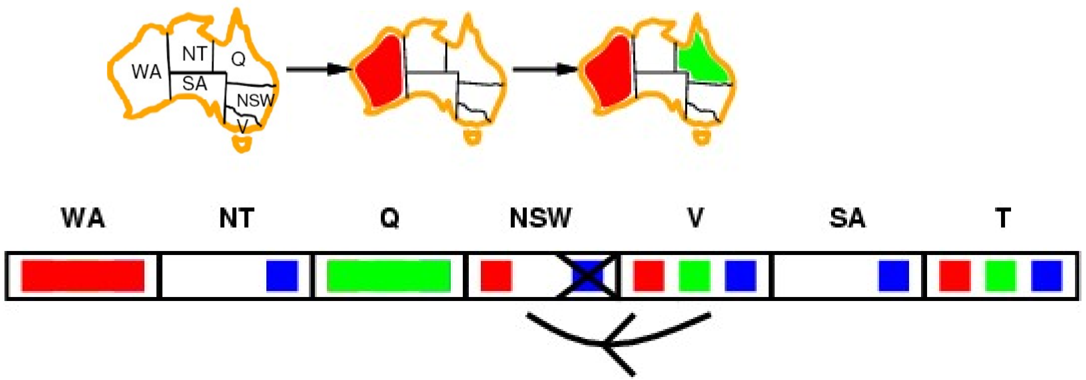
- Simplest form of propagation makes each pair of variables **consistent:**
  - $X \rightarrow Y$ is consistent iff for every value of X there is some allowed value of Y
  - When checking $X \rightarrow Y$, throw out any values of X for which there isn't an allowed value of Y



- If X loses a value, all pairs $Z \rightarrow X$ need to be rechecked

Arc consistency

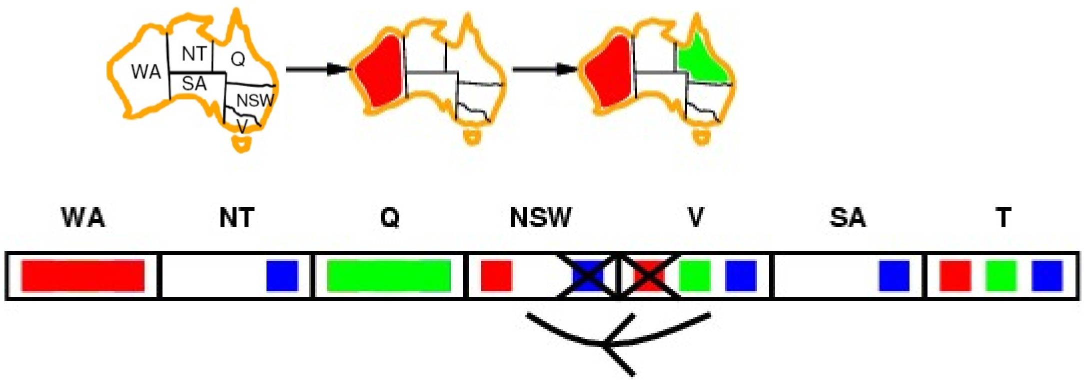
- Simplest form of propagation makes each pair of variables **consistent:**
  - $X \rightarrow Y$ is consistent iff for every value of X there is some allowed value of Y
  - When checking $X \rightarrow Y$, throw out any values of X for which there isn't an allowed value of Y



- If X loses a value, all pairs $Z \rightarrow X$ need to be rechecked

Arc consistency

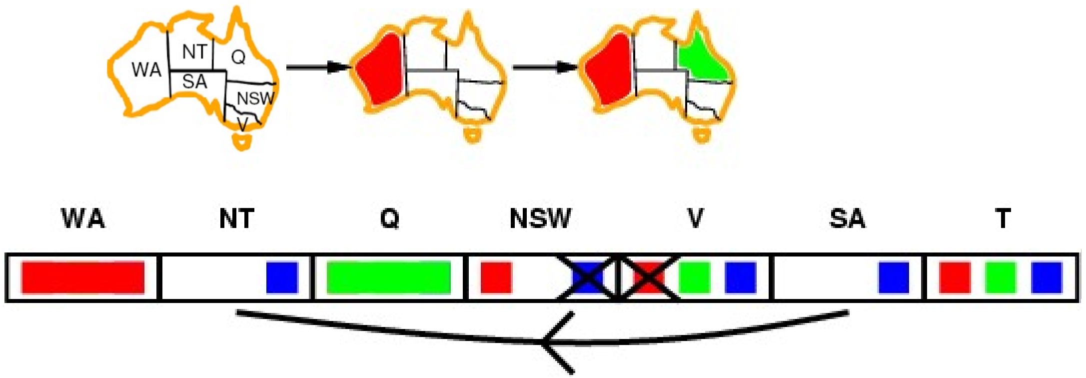
- Simplest form of propagation makes each pair of variables **consistent:**
  - $X \rightarrow Y$ is consistent iff for every value of X there is some allowed value of Y
  - When checking $X \rightarrow Y$, throw out any values of X for which there isn't an allowed value of Y



- If X loses a value, all pairs $Z \rightarrow X$ need to be rechecked

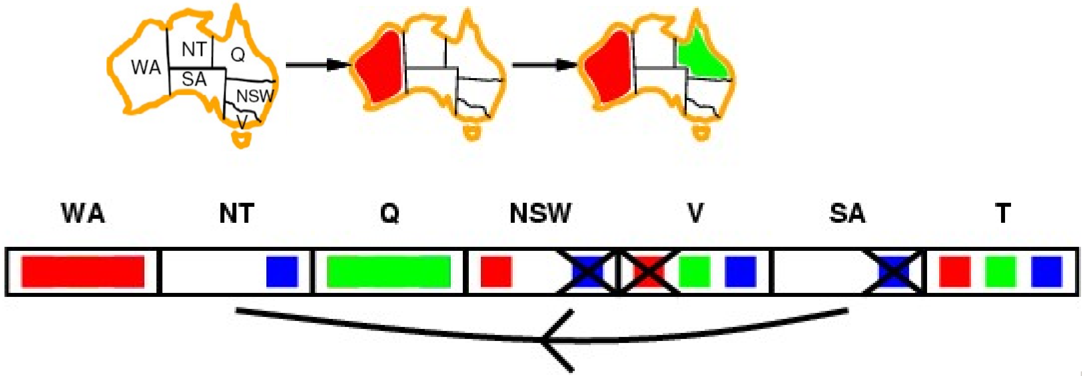
Arc consistency

- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y



Arc consistency

- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y

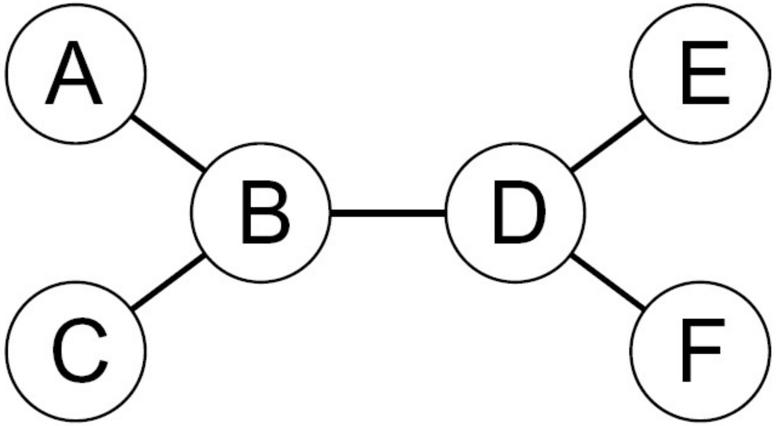


- Arc consistency detects failure earlier than forward checking
- Can be run before or after each assignment

Tree-structured CSPs

- Certain kinds of CSPs can be solved without resorting to backtracking search!

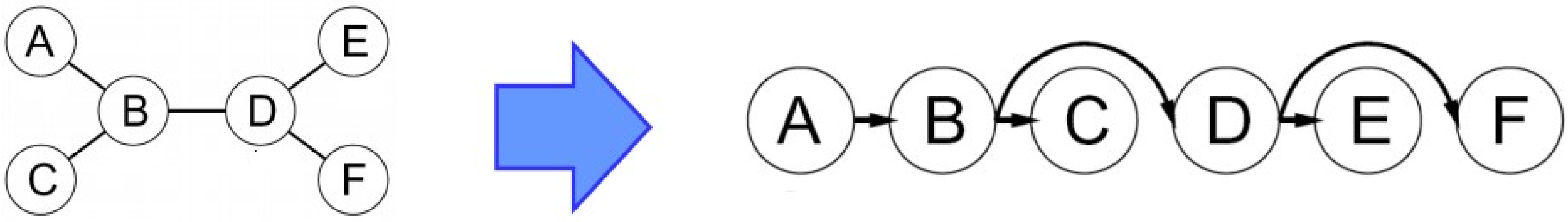
- *Tree-structured CSP*: constraint graph does not have any loops



Source: P. Abbeel, D. Klein, L. Zettlemoyer

Algorithm for tree-structured CSPs

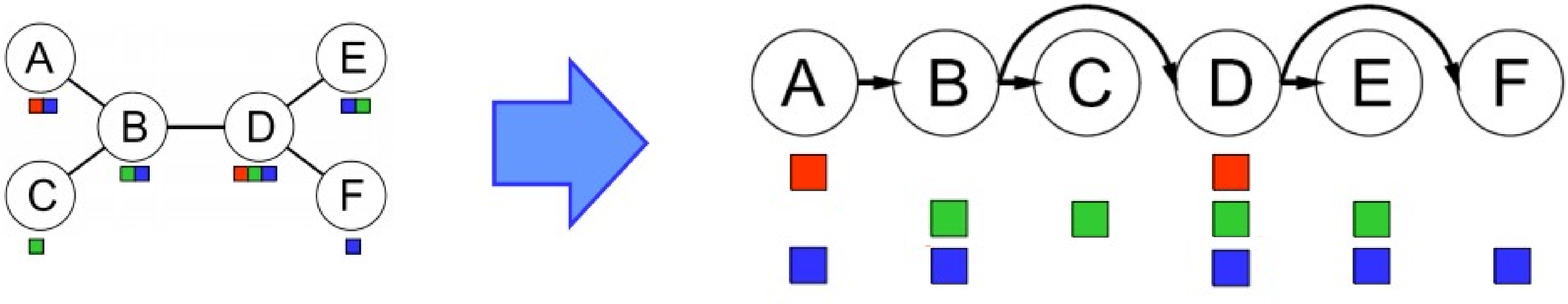
- Choose one variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering



http://cs188ai.wikia.com/wiki/Tree_Structure_CSPs

Algorithm for tree-structured CSPs

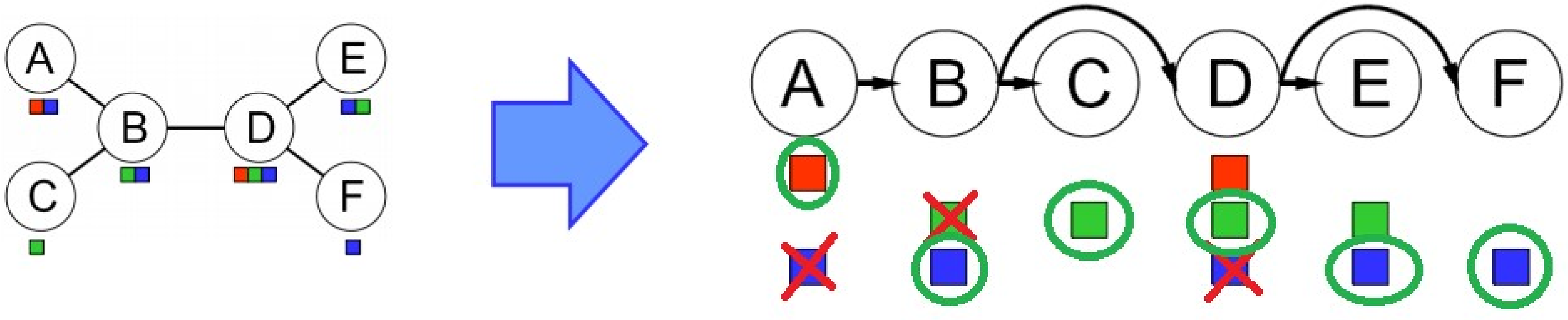
- Choose one variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering
- **Backward removal phase**: check arc consistency starting from the rightmost node and going backwards



http://cs188ai.wikia.com/wiki/Tree_Structure_CSPs

Algorithm for tree-structured CSPs

- Choose one variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering
- **Backward removal phase**: check arc consistency starting from the rightmost node and going backwards
- **Forward assignment phase**: select an element from the domain of each variable going left to right. We are guaranteed that there will be a valid assignment because each arc is consistent



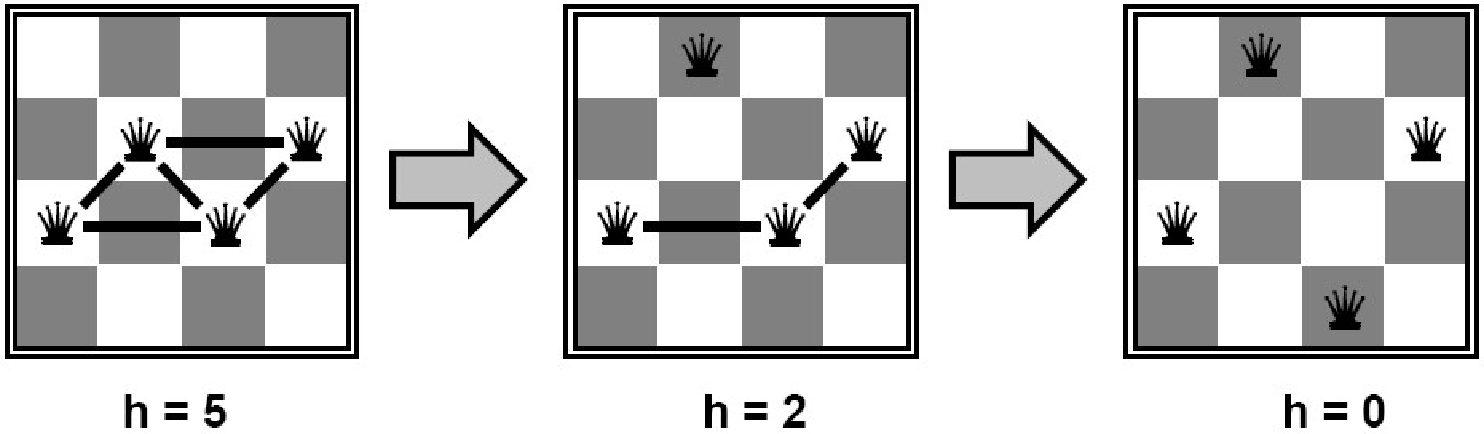
http://cs188ai.wikia.com/wiki/Tree_Structure_CSPs

Algorithm for tree-structured CSPs

- If *n* is the number of variables and *m* is the domain size, what is the running time of this algorithm?
  - $O(nm^2)$: we have to check arc consistency once for every node in the graph (every node has one parent), which involves looking at pairs of domain values
- What about backtracking search for general CSPs?
  - Worst case $O(m^n)$

Local search for CSPs

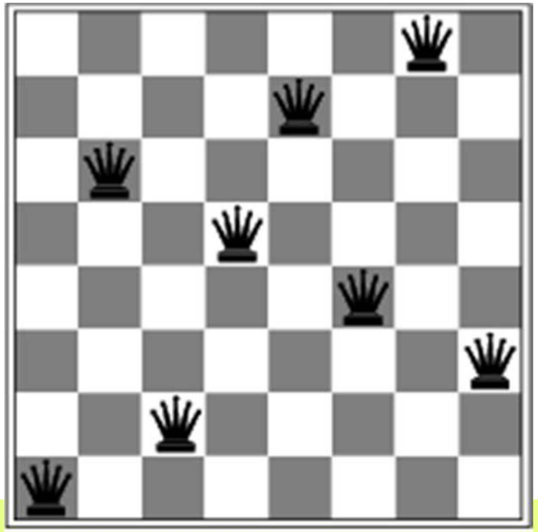
- Start with “complete” states, i.e., all variables assigned
- Allow states with unsatisfied constraints
- Attempt to improve states by reassigning variable values
- Hill-climbing search:
  - In each iteration, randomly select any conflicted variable and choose value that violates the fewest constraints
  - I.e., attempt to greedily minimize total number of violated constraints



h = number of conflicts

Local search for CSPs

- Start with "complete" states, i.e., all variables assigned
- Allow states with unsatisfied constraints
- Attempt to improve states by reassigning variable values
- Hill-climbing search:
  - In each iteration, randomly select any conflicted variable and choose value that violates the fewest constraints
  - I.e., attempt to greedily minimize total number of violated constraints
  - Problem: *local minima*

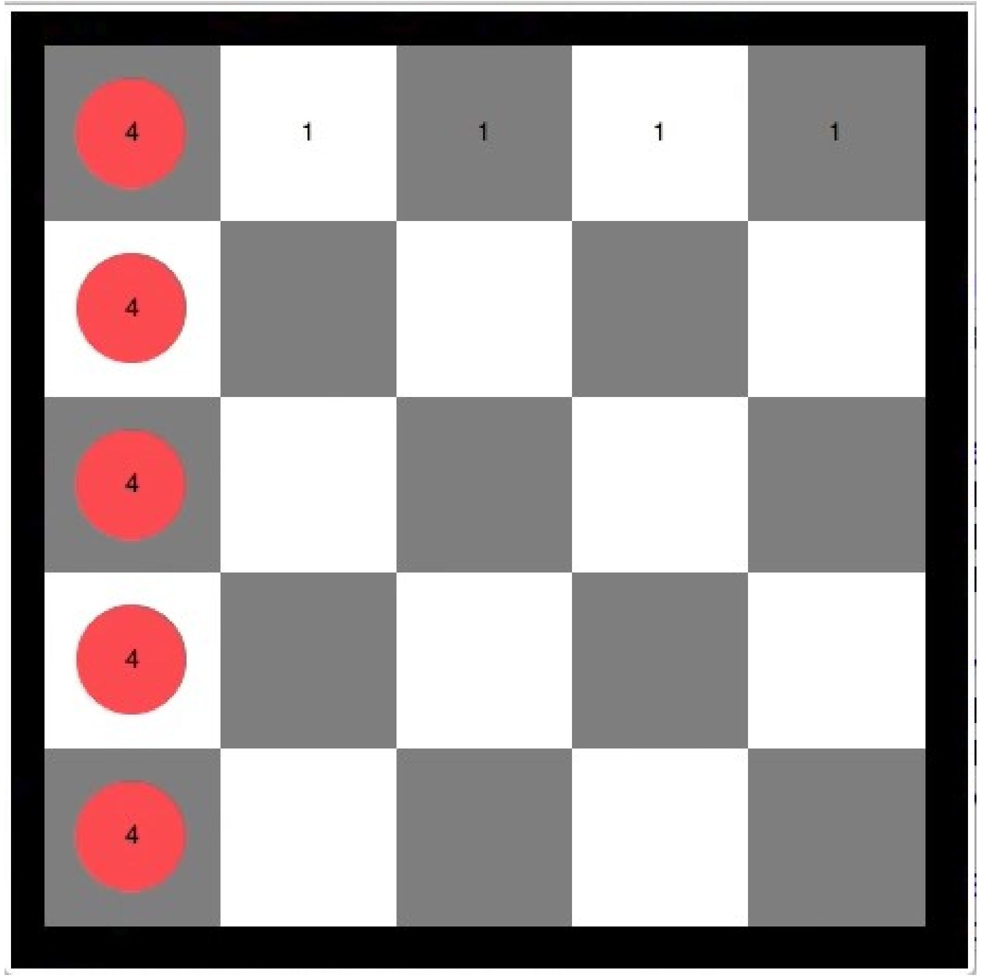


$h = 1$

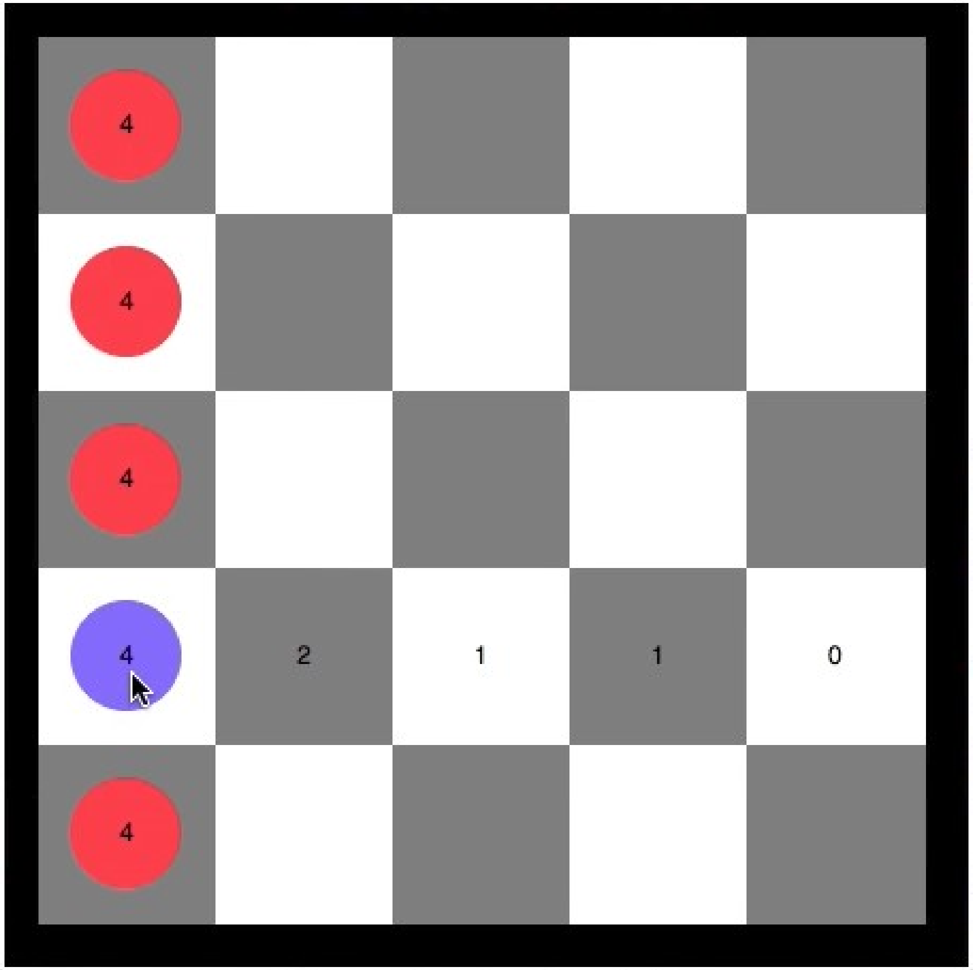
Local search for CSPs

- Start with "complete" states, i.e., all variables assigned
- Allow states with unsatisfied constraints
- Attempt to improve states by reassigning variable values
- Hill-climbing search:
  - In each iteration, randomly select any conflicted variable and choose value that violates the fewest constraints
  - I.e., attempt to greedily minimize total number of violated constraints
  - Problem: *local minima*
- For more on local search, see ch. 4

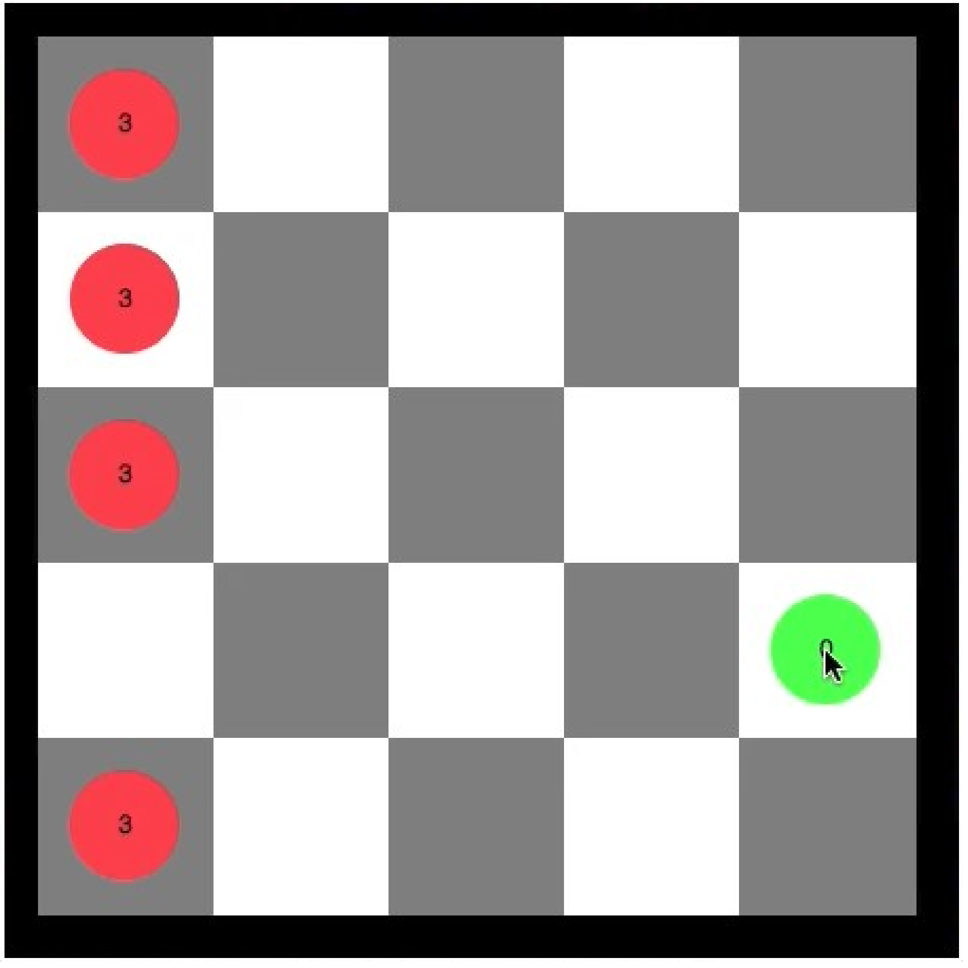
N-Queen Iterative Improvement



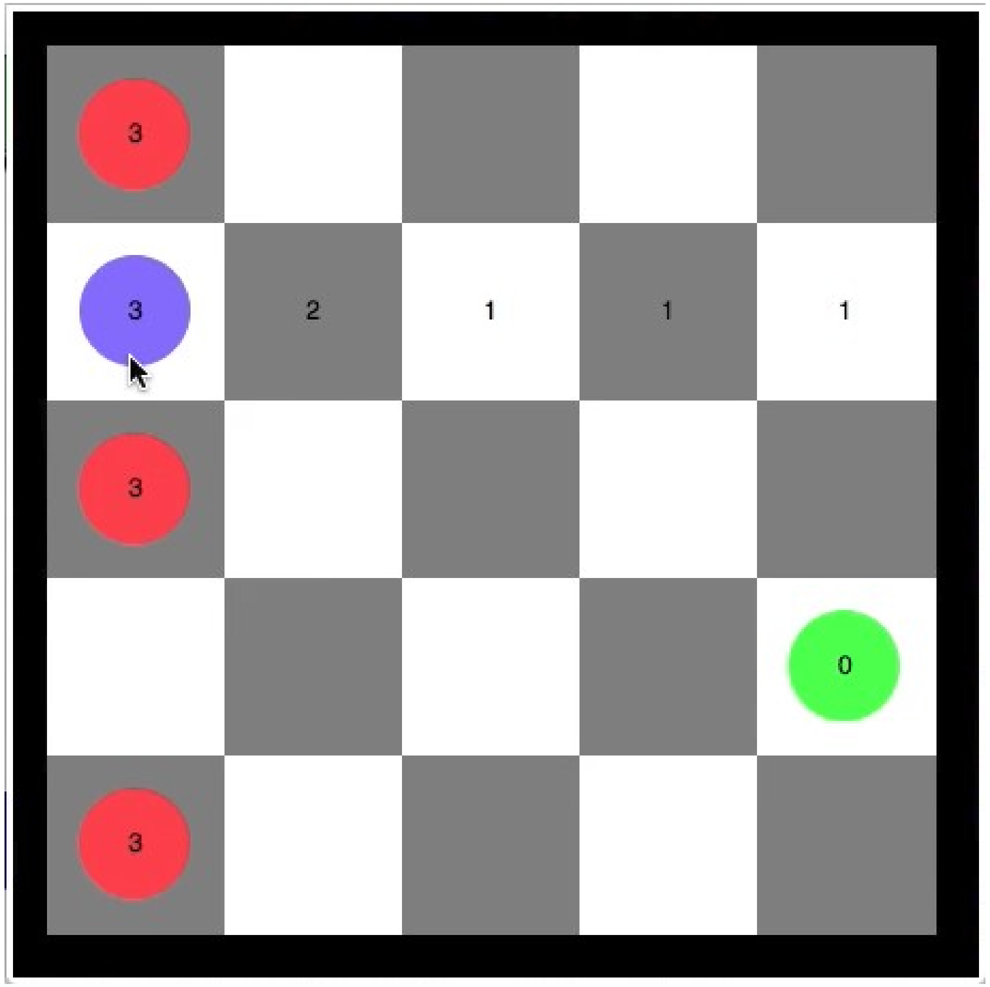
N-Queen Iterative Improvement



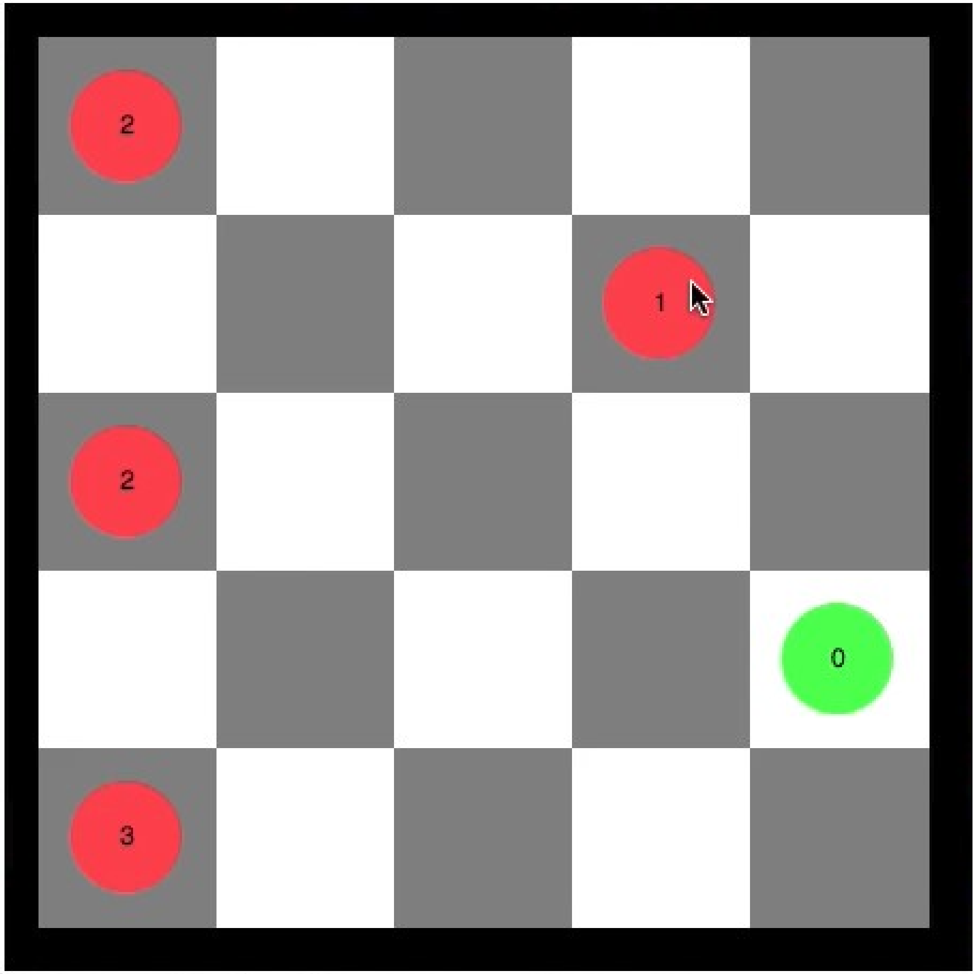
N-Queen Iterative Improvement



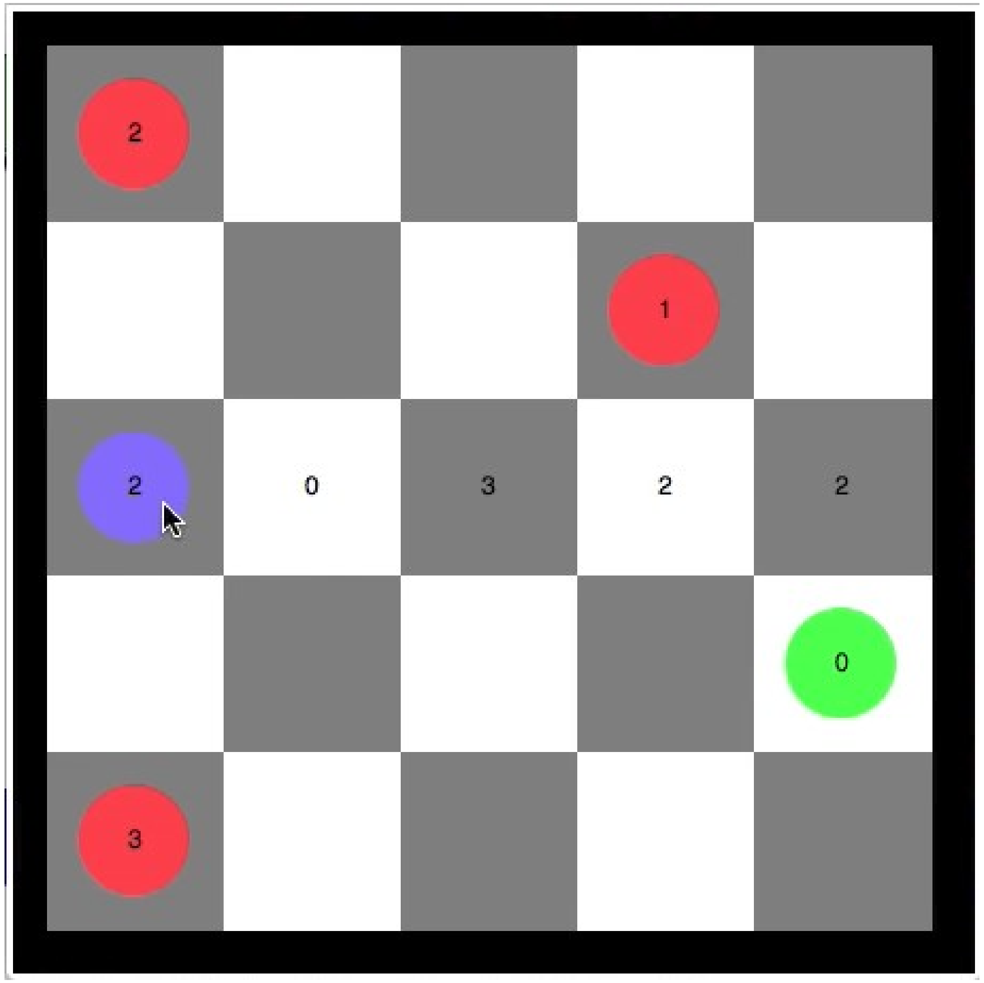
N-Queen Iterative Improvement



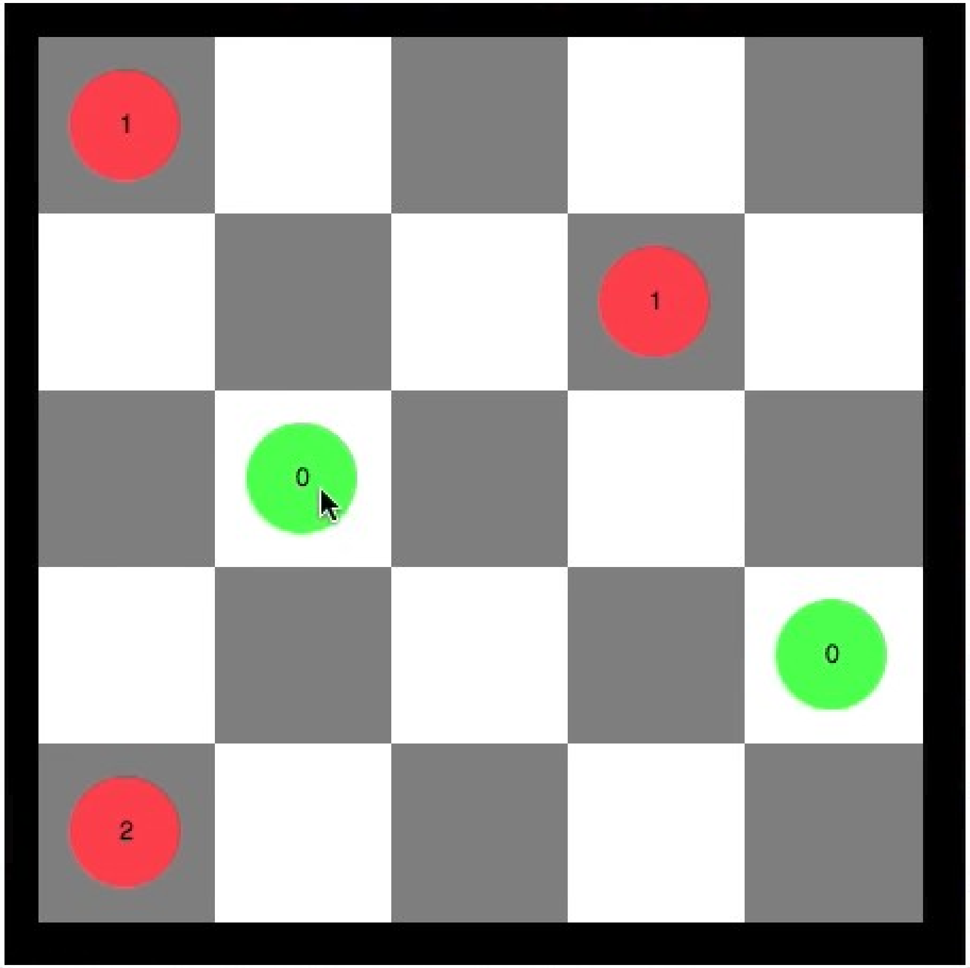
N-Queen Iterative Improvement



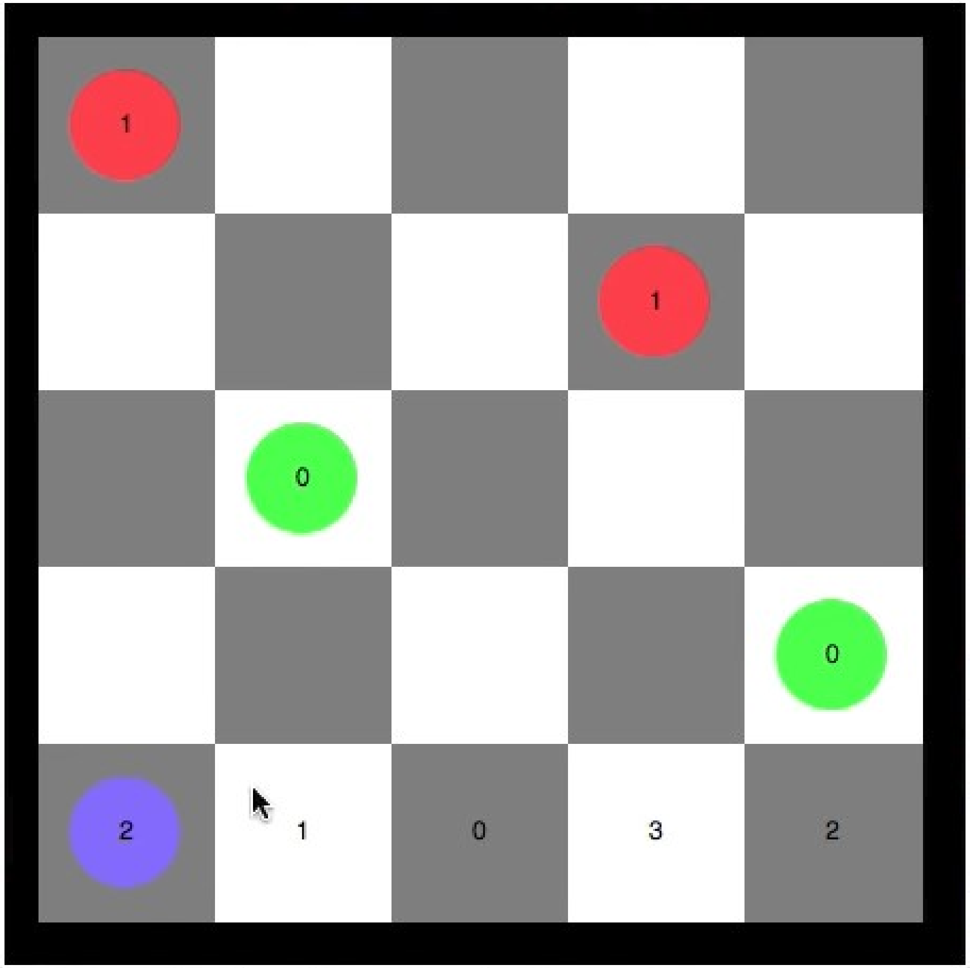
N-Queen Iterative Improvement



N-Queen Iterative Improvement



N-Queen Iterative Improvement



N-Queen Iterative Improvement