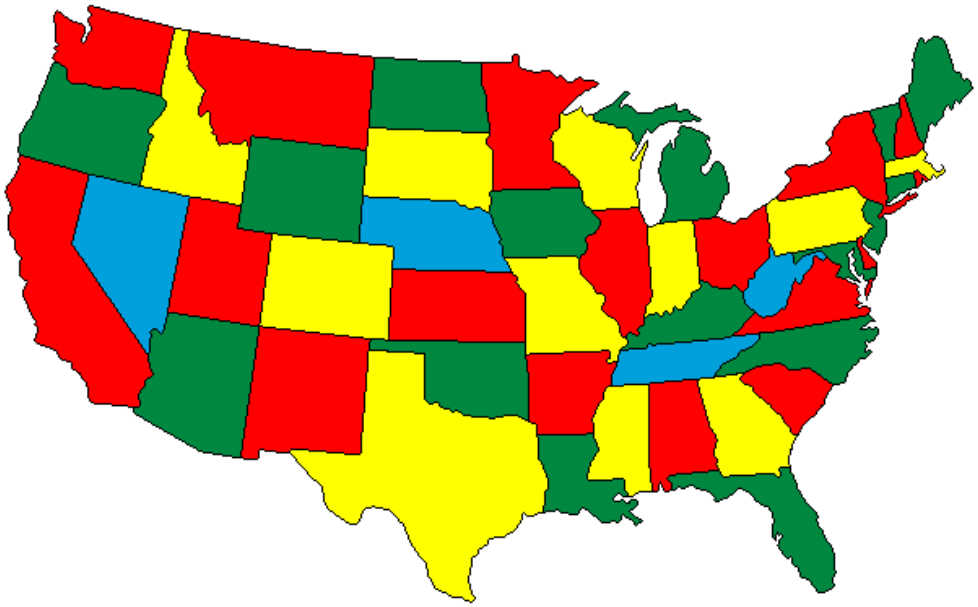
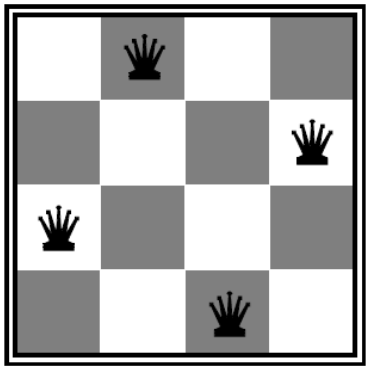
Lecture 5

Instructor: Amit Kumar Das

Senior Lecturer,
Department of Computer Science & Engineering,
East West University
Dhaka, Bangladesh.

Constraint Satisfaction Problems (Chapter 6)





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: single agent, deterministic, fully observable, discrete environment
- **Search for *planning***
  - The path to the goal is the important thing
  - Paths have various costs, depths
- **Search for *assignment***
  - Assign values to variables while respecting certain constraints
  - The goal (complete, consistent assignment) is the important thing

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

Constraint satisfaction problems (CSPs)

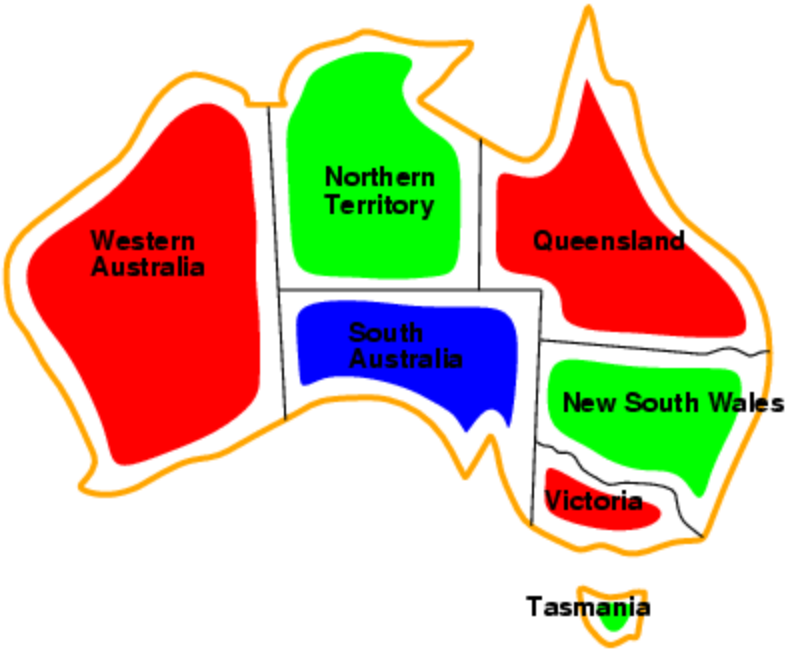
- Definition:
  - **State** is defined by variables X_i with values from domain D_i
  - **Goal test** is a set of constraints specifying allowable combinations of values for subsets of variables
  - **Solution** is a complete, consistent assignment

- How does this compare to the “generic” tree search formulation?
  - A more structured representation for states, expressed in a formal representation language
  - Allows useful general-purpose algorithms with more power than standard search algorithms

Example: Map Coloring

- **Variables:** WA, NT, Q, NSW, V, SA, T
- **Domains:** {red, green, blue}
- **Constraints:** adjacent regions must have different colors
  e.g., WA ≠ NT, or (WA, NT) in {(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)}

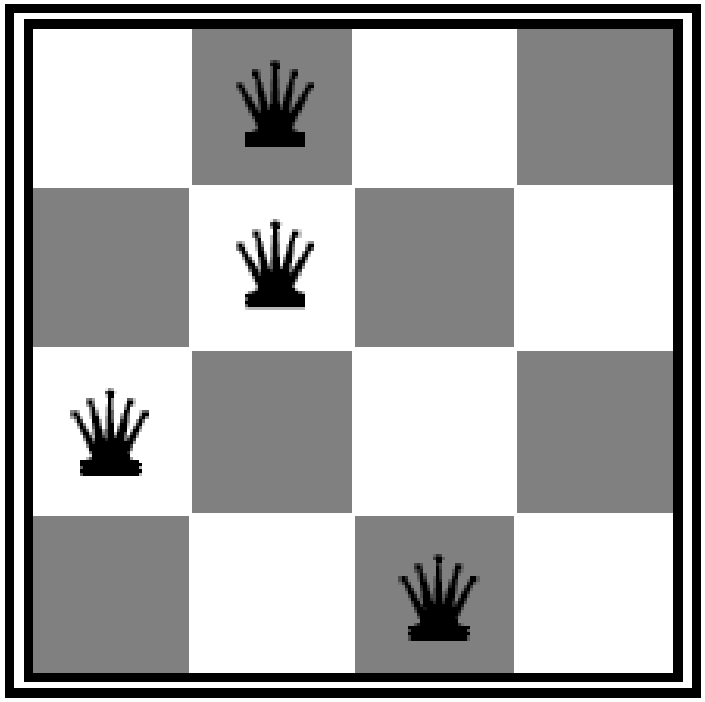
Example: Map Coloring

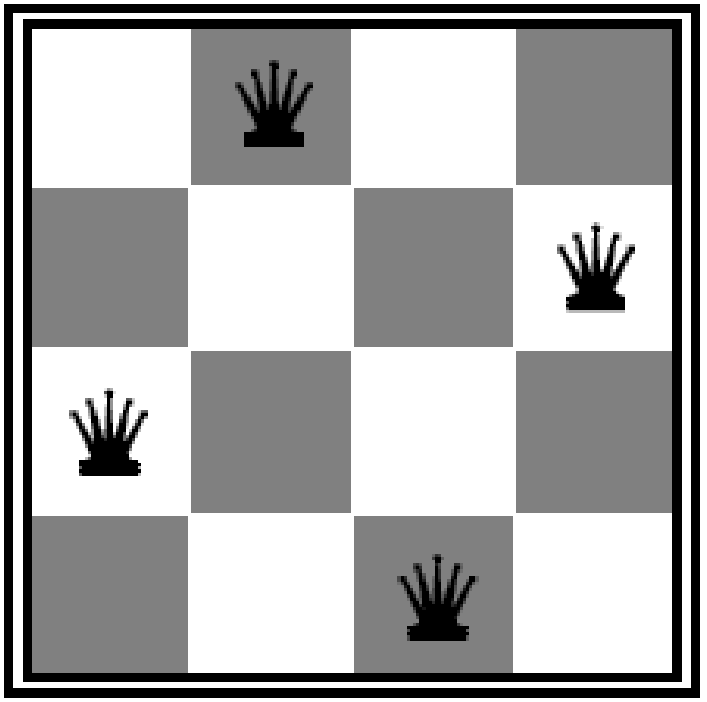


- **Solutions** are *complete* and *consistent* assignments, e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

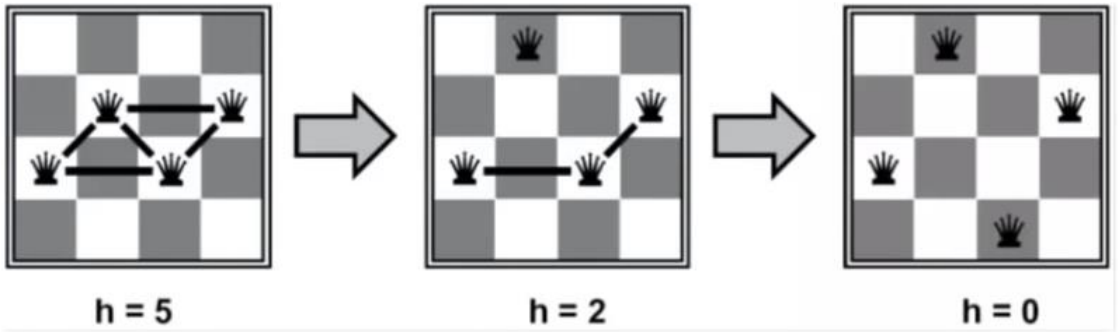
Example: *n*-queens problem

- Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal



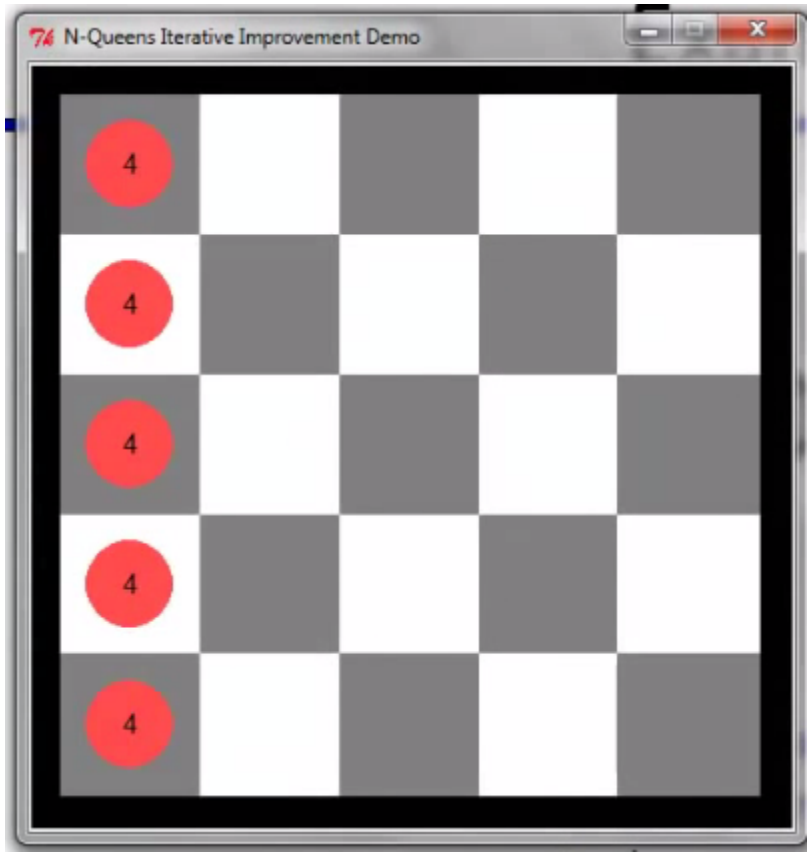


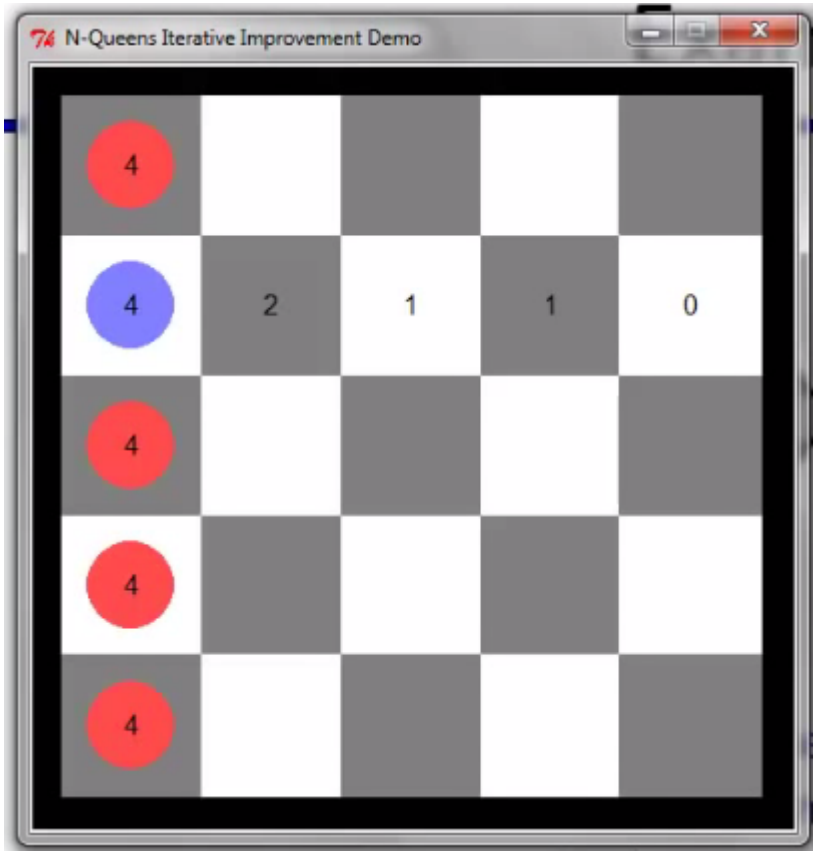
Example: 4-Queens

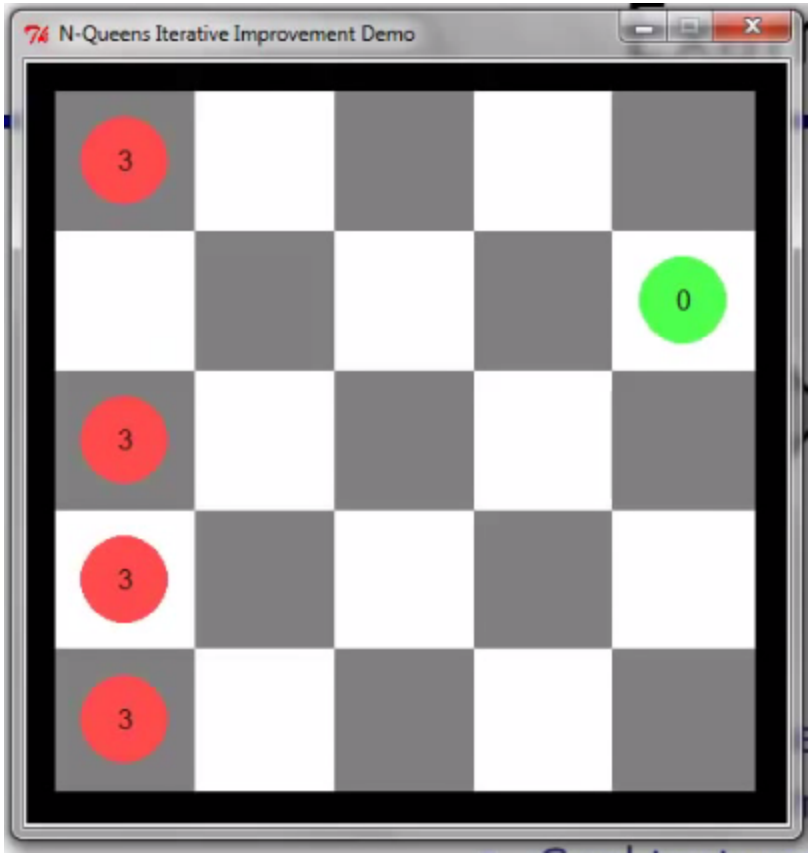


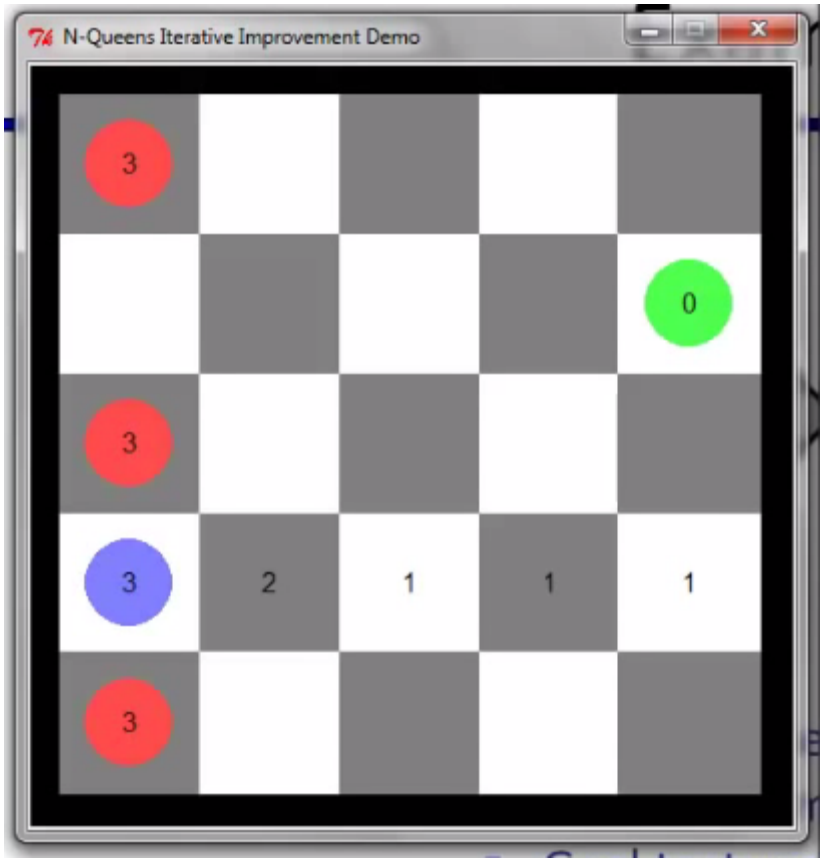
- States: 4 queens in 4 columns ($4^4 = 256$ states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: c(n) = number of attacks

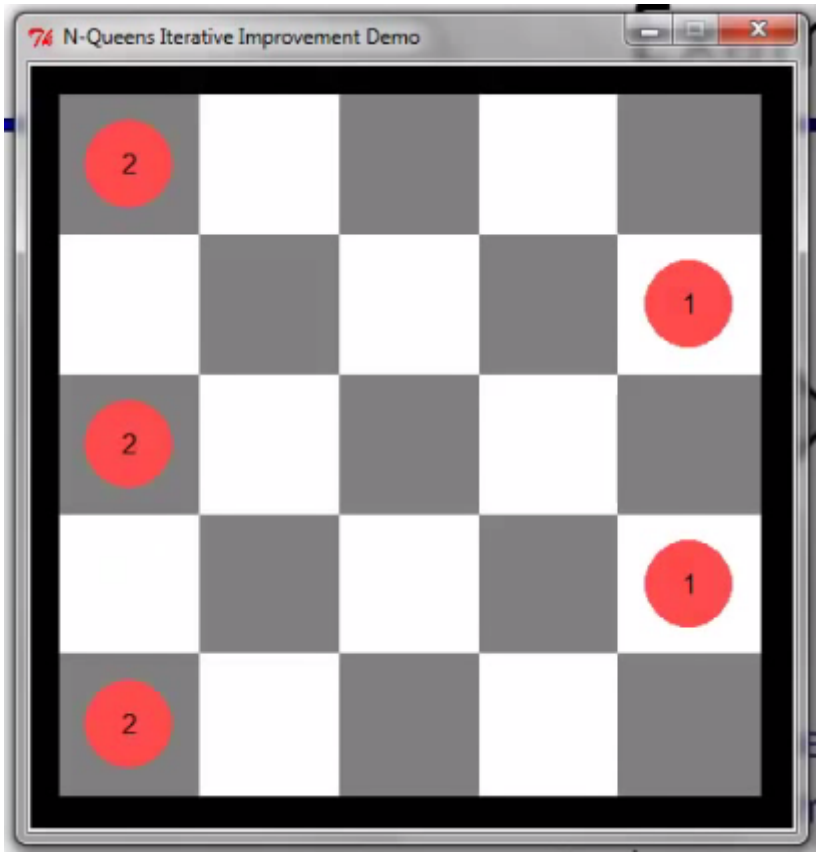
[demos: iterative n-queens, map coloring]

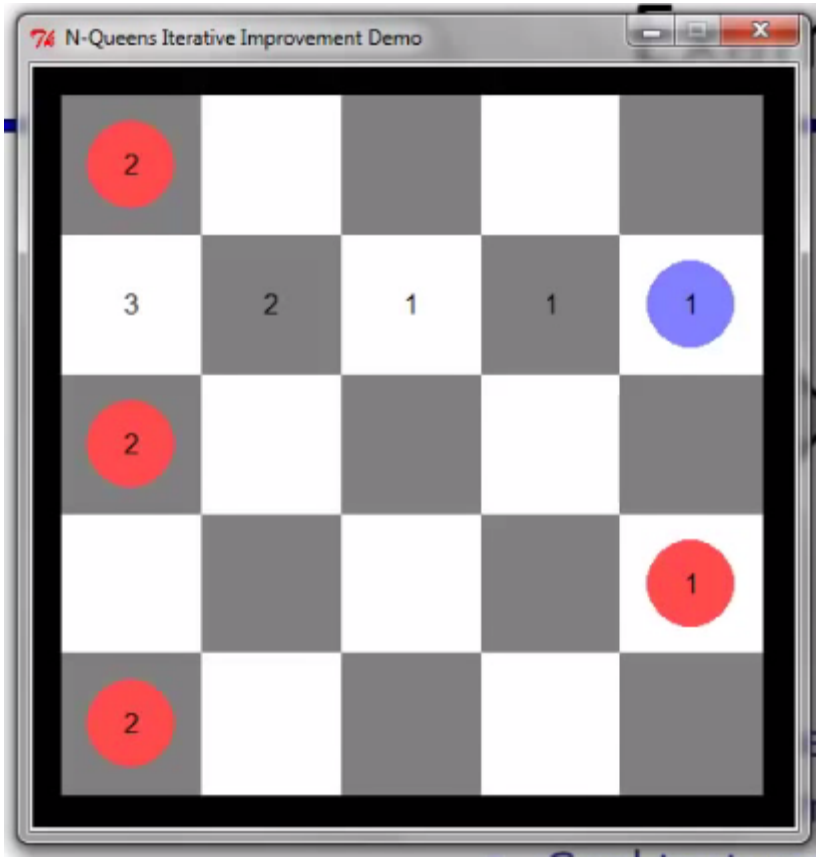

N-Queens Iterative Improvement Demo
4
4
4
4
4

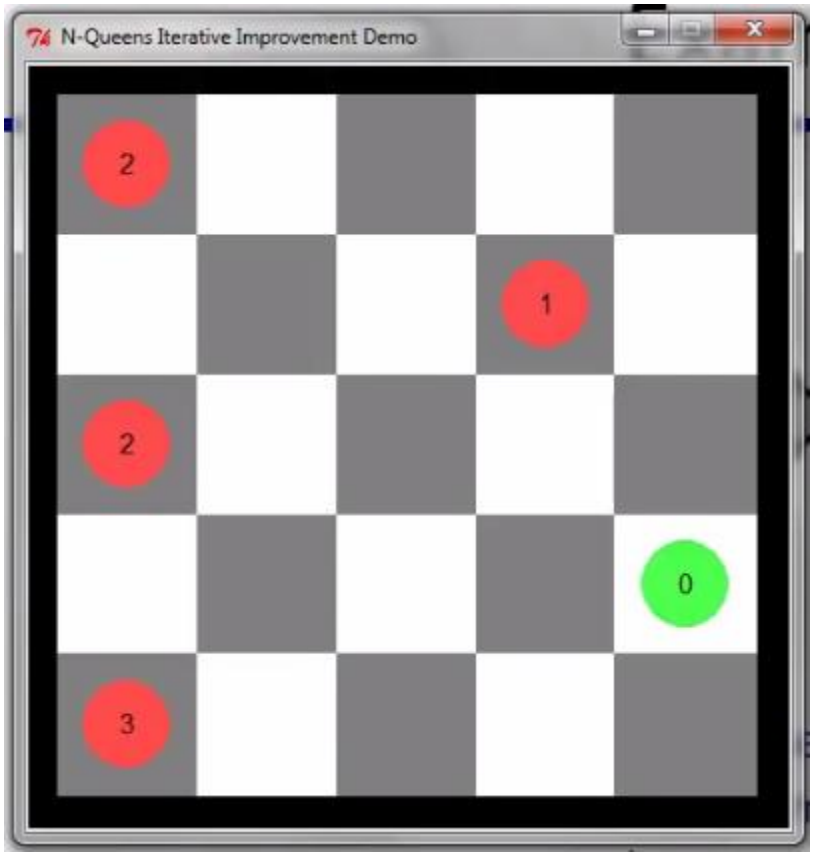

N-Queens Iterative Improvement Demo
4
4
2
1
1
0
4
4
4

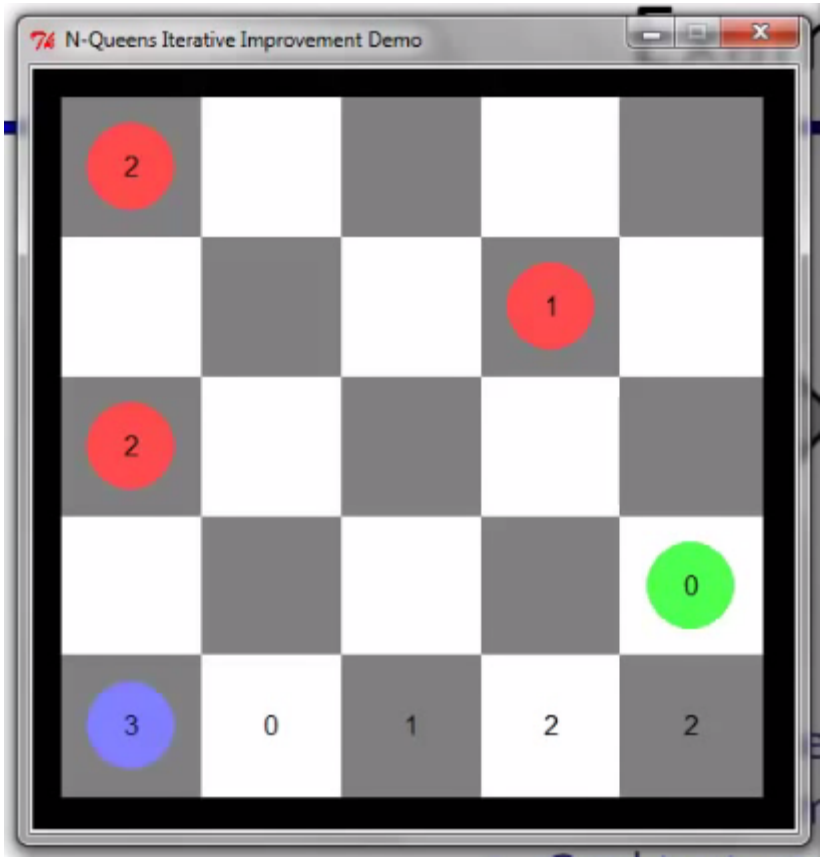

N-Queens Iterative Improvement Demo
3
0
3
3
3

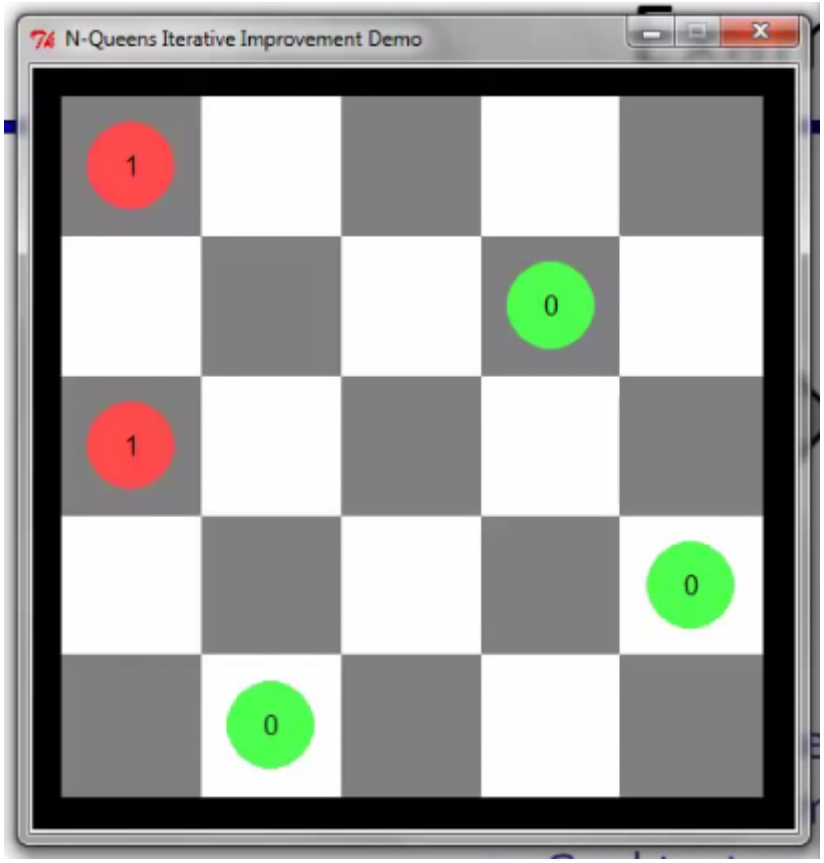

N-Queens Iterative Improvement Demo
3
0
3
3
2
1
1
1
3

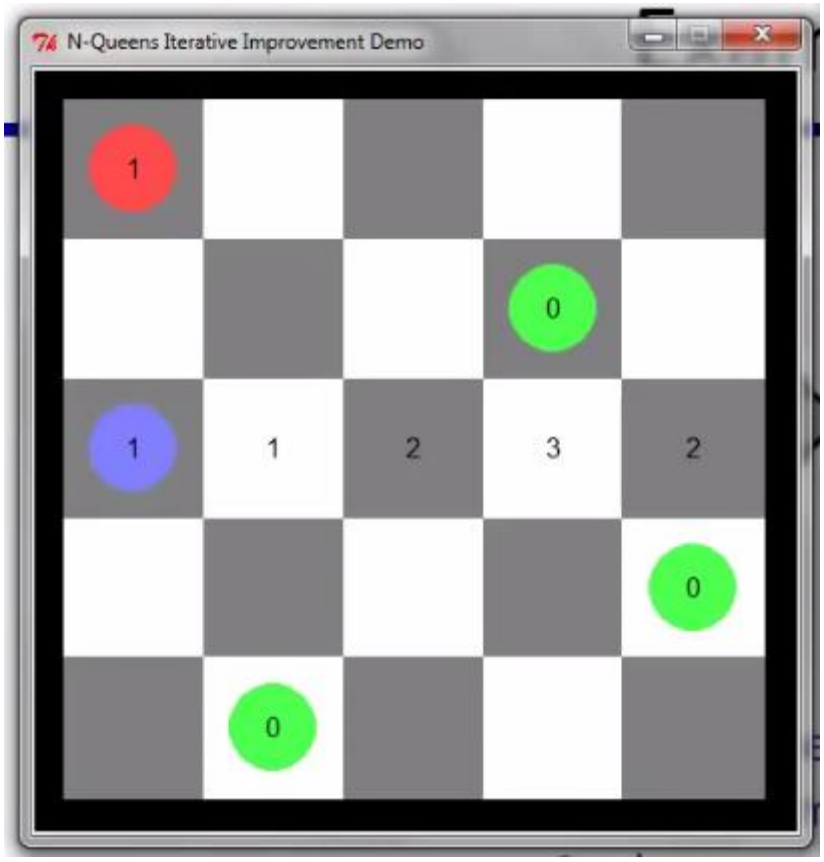

N-Queens Iterative Improvement Demo
2
1
2
1
2

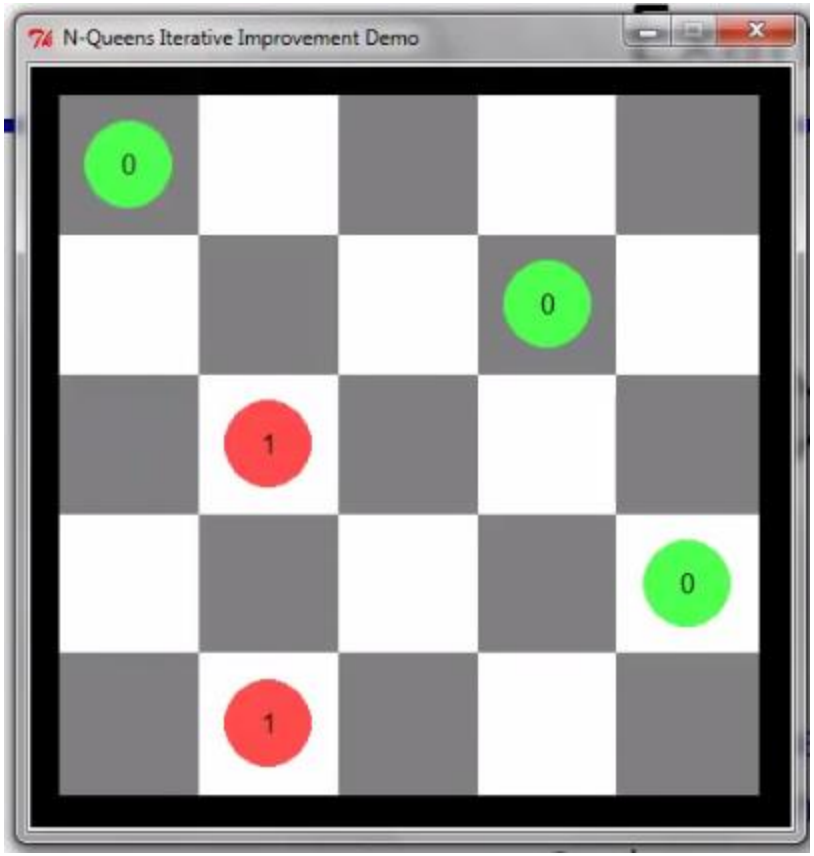

N-Queens Iterative Improvement Demo
2
3
2
1
1
1
2
1
2

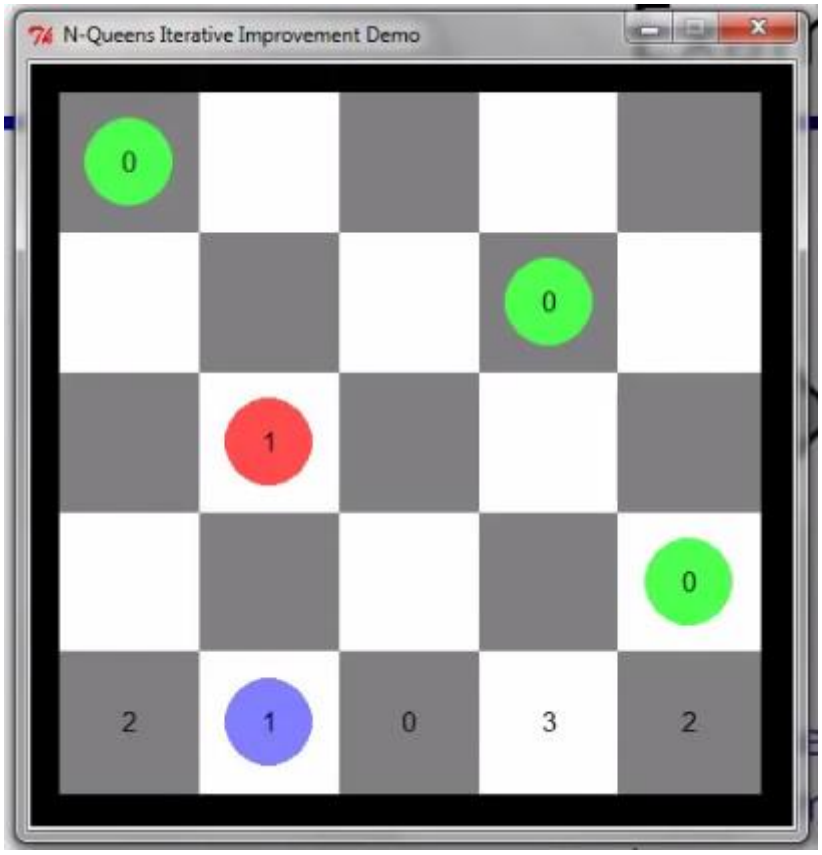

N-Queens Iterative Improvement Demo
2
1
2
0
3

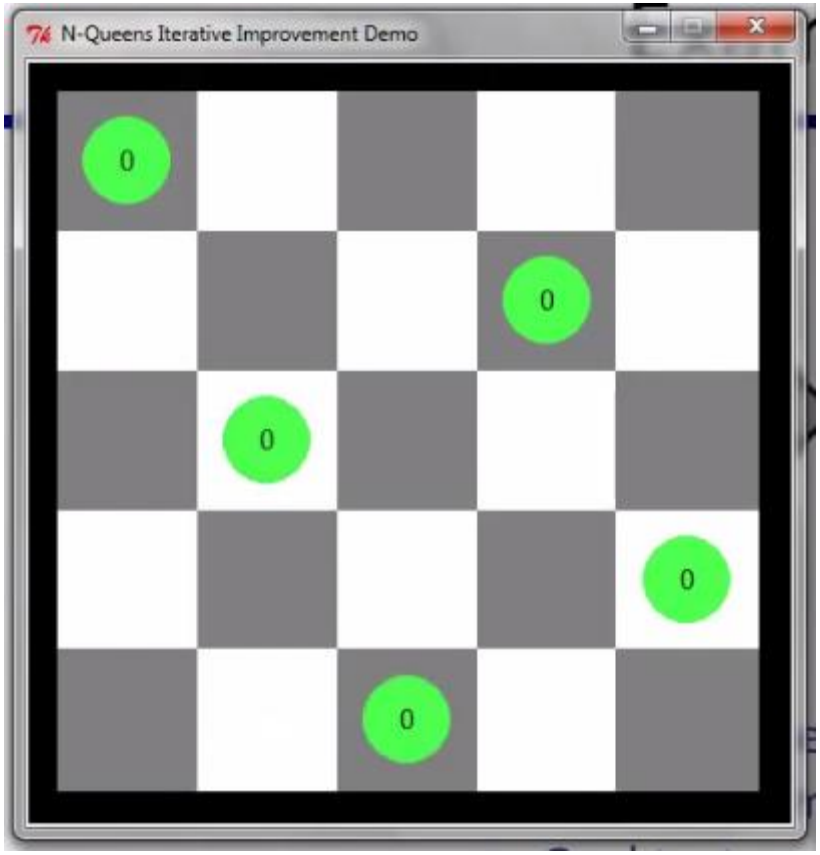

N-Queens Iterative Improvement Demo
2
1
2
0
3
0
1
2
2


N-Queens Iterative Improvement Demo
1
0
1
0
0


N-Queens Iterative Improvement Demo
1
0
1
1
2
3
2
0
0


N-Queens Iterative Improvement Demo
0
0
1
0
1


N-Queens Iterative Improvement Demo
0
0
1
0
2
1
0
3
2


N-Queens Iterative Improvement Demo
0
0
0
0
0

Example: N-Queens

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

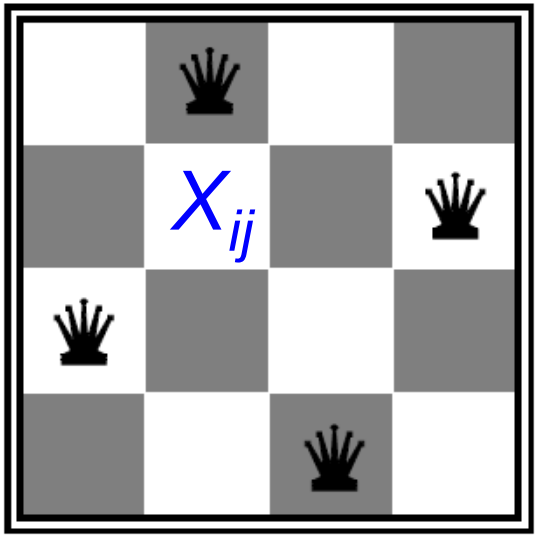
$\Sigma_{i,j}\, X_{ij} = N$

$(X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\}$

$(X_{ij}, X_{kj}) \in \{(0, 0), (0, 1), (1, 0)\}$

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

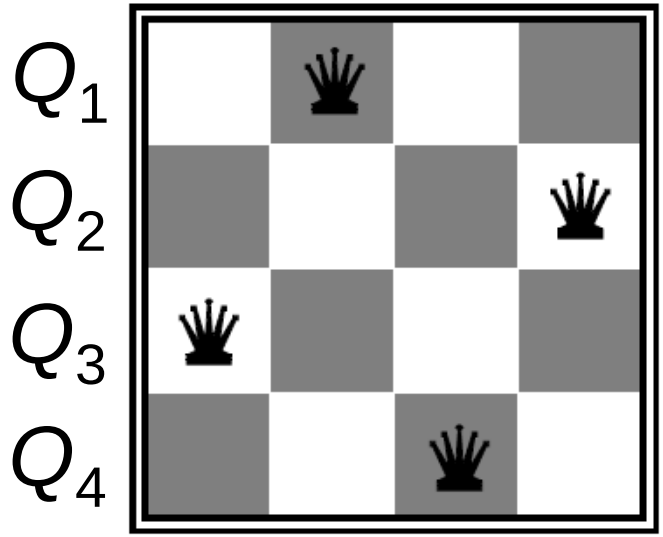
$(X_{ij}, X_{i+k,\, j-k}) \in \{(0, 0), (0, 1), (1, 0)\}$



N-Queens: Alternative formulation

- **Variables:** Q_i
- **Domains:** {1, … , N}
- **Constraints:**

  $\forall$ i, j non-threatening (Q_i , Q_j)



Example: Cryptarithmetic

- **Variables:** T, W, O, F, U, R

  X_1, X_2
- **Domains**: {0, 1, 2, …, 9}
- **Constraints:**

  $O + O = R + 10 * X_1$

  $W + W + X_1 = U + 10 * X_2$

  $T + T + X_2 = O + 10 * F$

  Alldiff(T, W, O, F, U, R)

  $T \neq 0, F \neq 0$

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

Example: Sudoku

- **Variables:** X_{ij}
- **Domains:** {1, 2, …, 9}
- **Constraints:**

  Alldiff(X_{ij} in the same *unit*)

					8			4
	8	4		1	6			
			5			1		
1		3	8			9		
6		8		X_{ij}		4		3
		2			9	5		1
		7			2			
			7	8		2	6	
2			3					

Real-world CSPs

- Assignment problems
  - e.g., who teaches what class
- Timetable problems
  - e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling

- More examples of CSPs: http://www.csplib.org/

Standard search formulation (incremental)

- **States:**
  - Variables and values assigned so far
- **Initial state:**
  - The empty assignment
- **Action:**
  - Choose any unassigned variable and assign to it a value that does not violate any constraints
    - Fail if no legal assignments
- **Goal test:**
  - The current assignment is complete and satisfies all constraints

Standard search formulation (incremental)

- What is the depth of any solution (assuming ***n*** variables)?

  n (this is good)

- Given that there are ***m*** possible values for any variable, how many paths are there in the search tree?

  $n! \cdot m^n$ (this is bad)

- How can we reduce the branching factor?

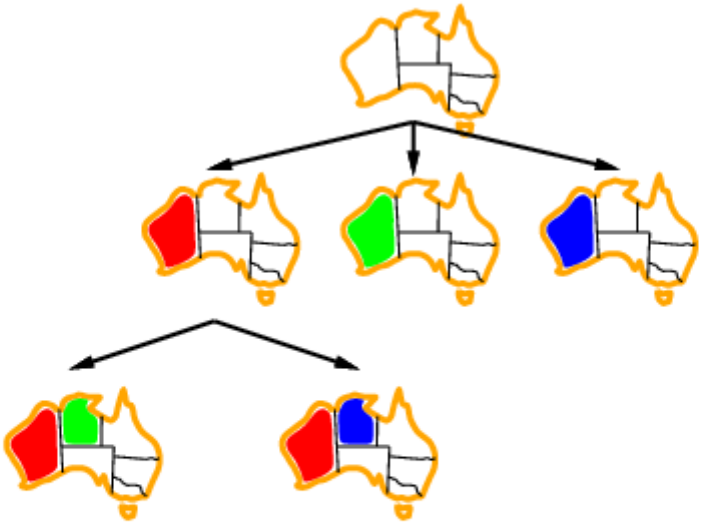
Backtracking search

- In CSP's, variable assignments are **commutative**
  - For example, *[WA = red then NT = green]* is the same as *[NT = green then WA = red]*
- We only need to consider assignments to a single variable at each level (i.e., we fix the order of assignments)
  - Then there are only $\boldsymbol{m^n}$ leaves
- Depth-first search for CSPs with single-variable assignments is called **backtracking search**

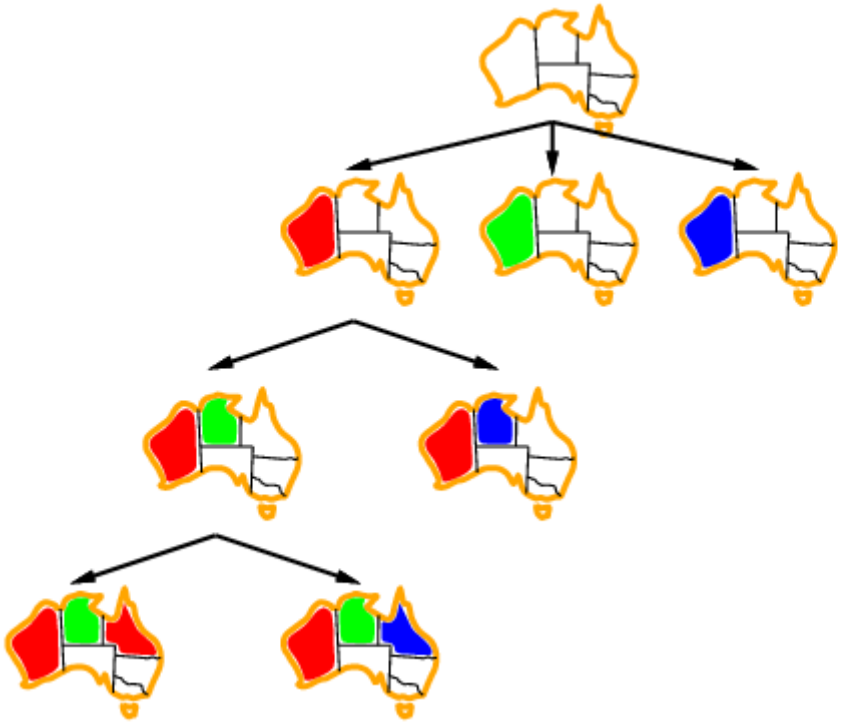
Example

Example

Example



Example



Backtracking search algorithm

```
function RECURSIVE-BACKTRACKING(assignment, csp)
  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)
    if value is consistent with assignment given CONSTRAINTS[csp]
      add {var = value} to assignment
      result ← RECURSIVE-BACKTRACKING(assignment, csp)
      if result ≠ failure then return result
      remove {var = value} from assignment
  return failure
```

- Making backtracking search efficient:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?

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
  - Tie-breaker among most constrained variables

Which variable should be assigned next?

- **Most constraining variable:**
  - Choose the variable that imposes the most constraints on the remaining variables
  - Tie-breaker among most constrained variables

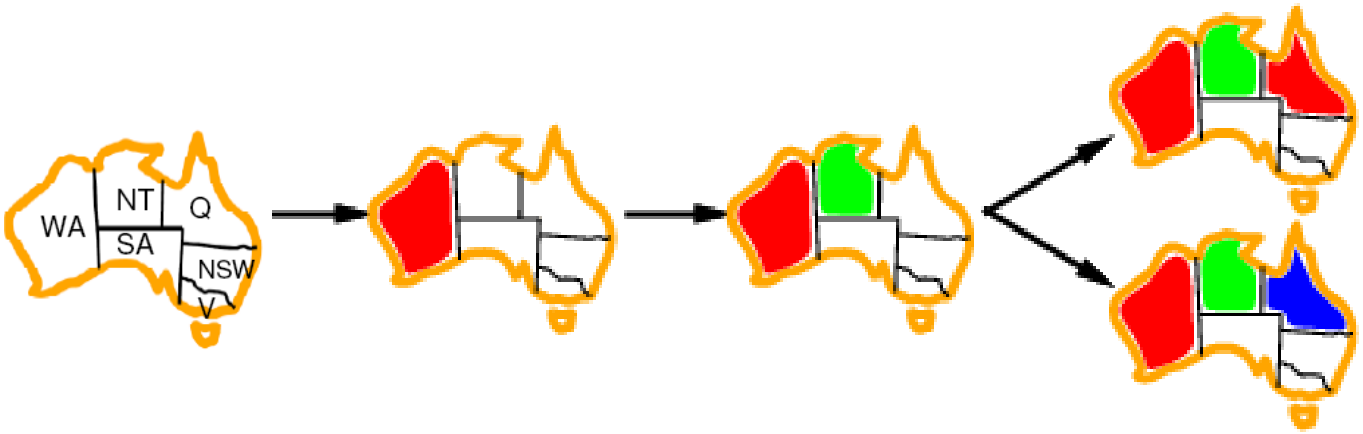
Given a variable, in which order should its values be tried?

- 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?

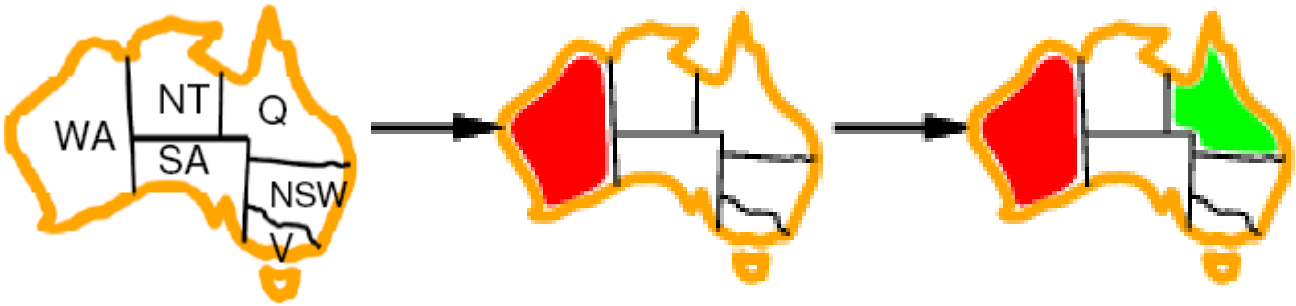


Early detection of failure

```
function RECURSIVE-BACKTRACKING(assignment, csp)
    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)
        if value is consistent with assignment given CONSTRAINTS[csp]
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

Apply *inference* to reduce the space of possible assignments and detect failure early

Early detection of failure



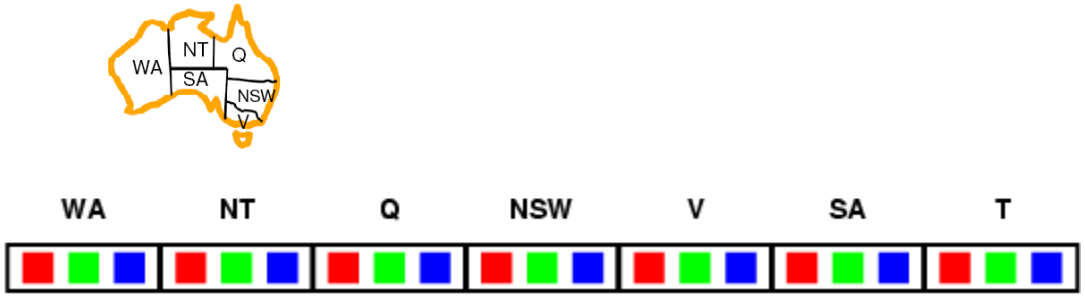
Apply *inference* to reduce the space of possible assignments and detect failure early

Early detection of failure: Forward checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values

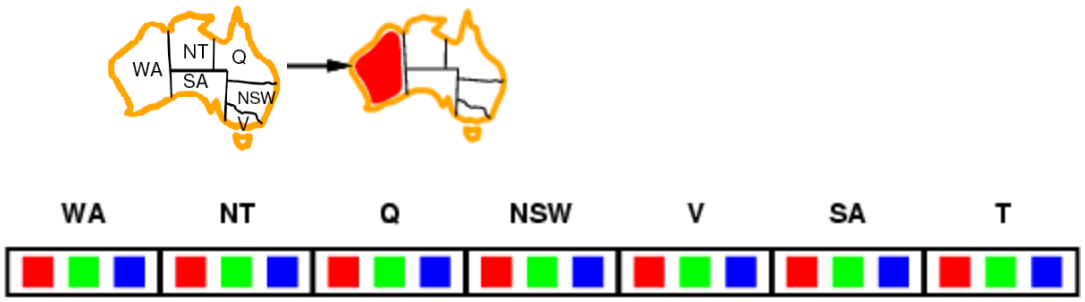
Early detection of failure: Forward checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



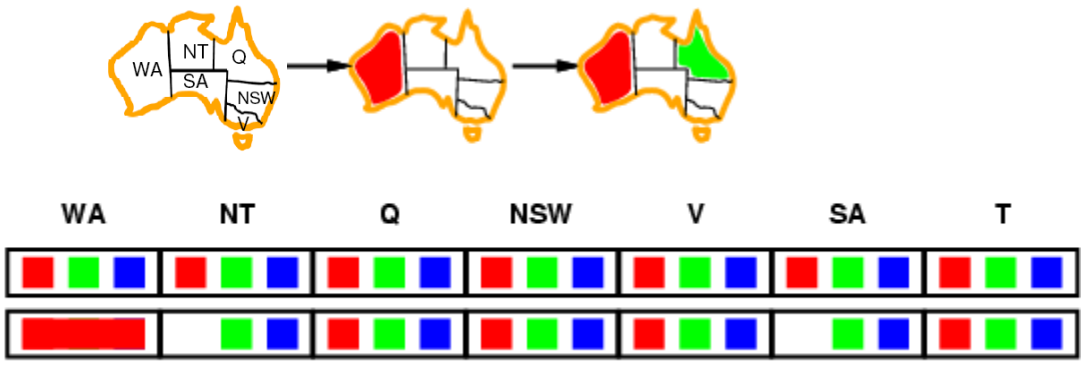
Early detection of failure: Forward checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



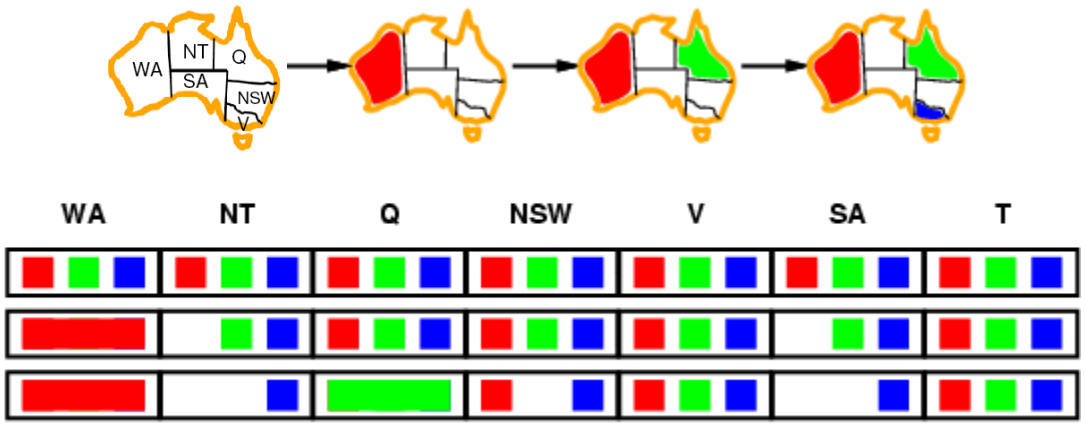
Early detection of failure: Forward checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



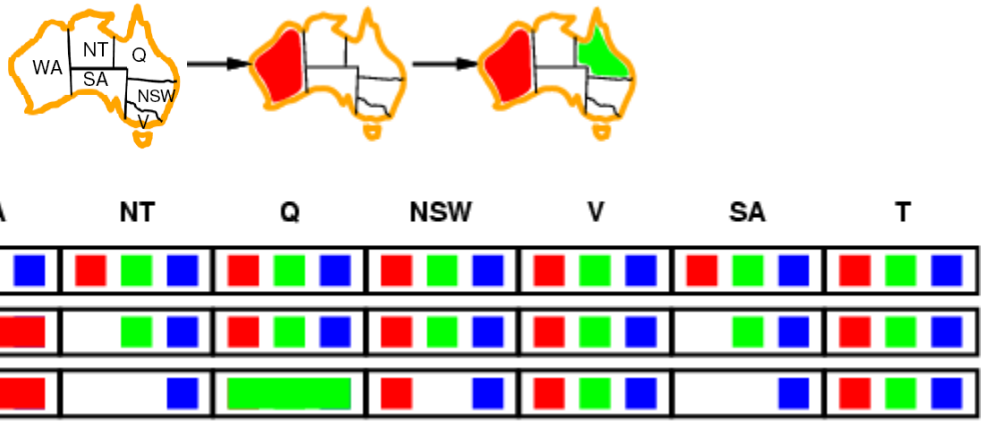
Early detection of failure: Forward checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



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*

Thank You