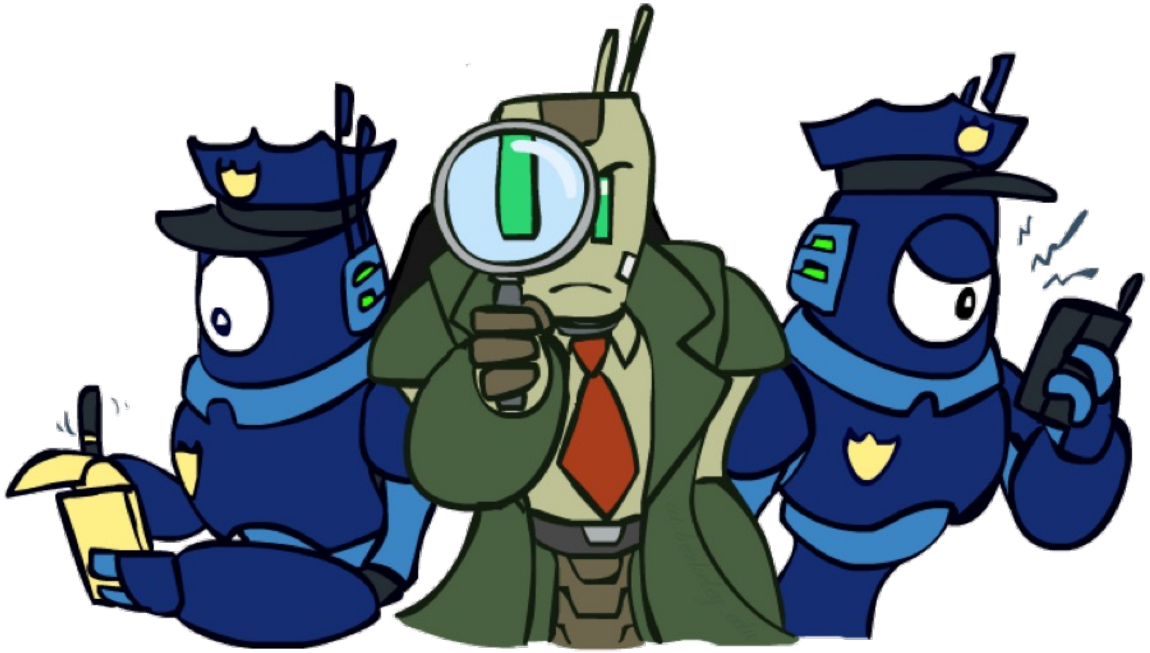
CMSC 471

Constraint Satisfaction Problems III



Instructor: KMA Solaiman

These slides were modified from Dan Klein and Pieter Abbeel at UC Berkeley [ai.berkeley.edu] and Frank Ferraro [ferraro@umbc.edu].

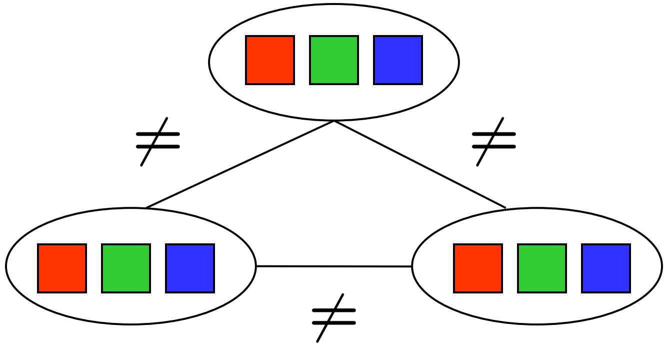
Today

- Efficient Solution of CSPs

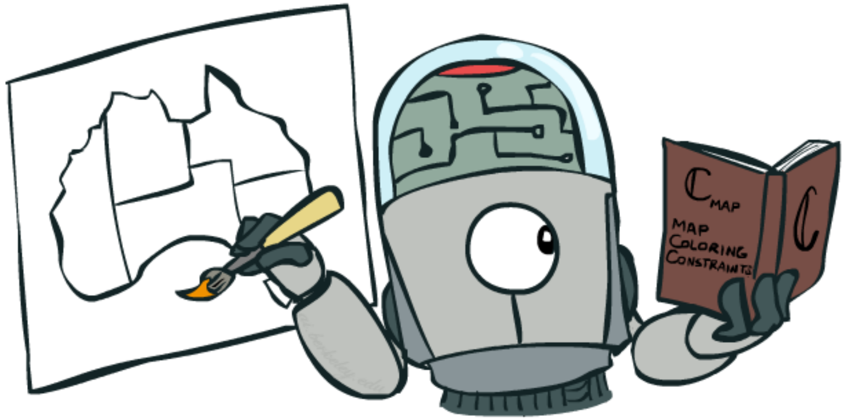
- Local Search

Reminder: CSPs

- CSPs:
  - Variables
  - Domains
  - Constraints
    - Implicit (provide code to compute)
    - Explicit (provide a list of the legal tuples)
    - Unary / Binary / N-ary

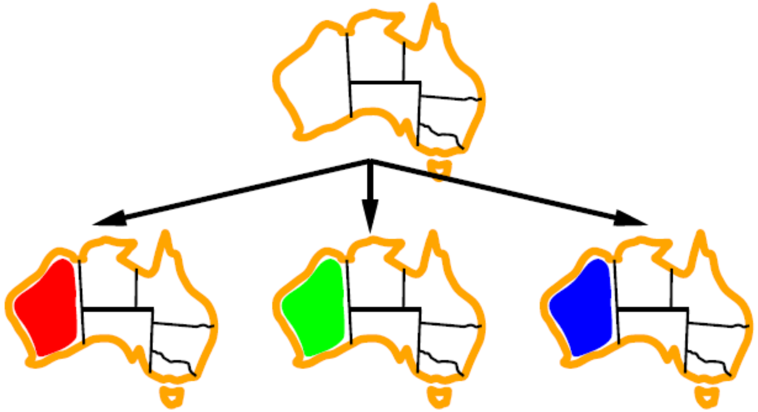


- Goals:
  - Here: find any solution
  - Also: find all, find best, etc.

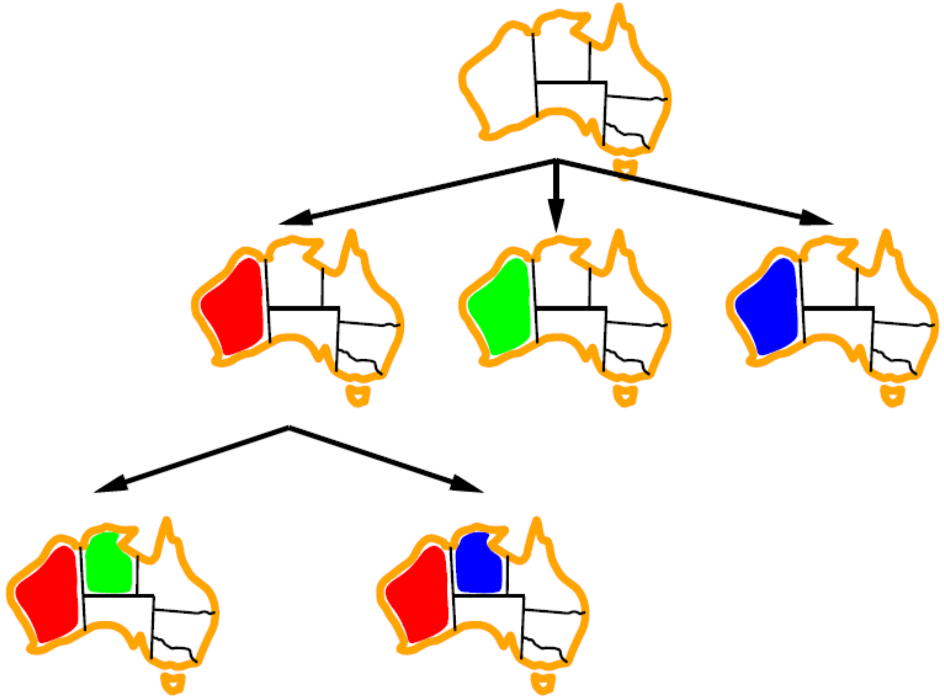


Backtracking Example

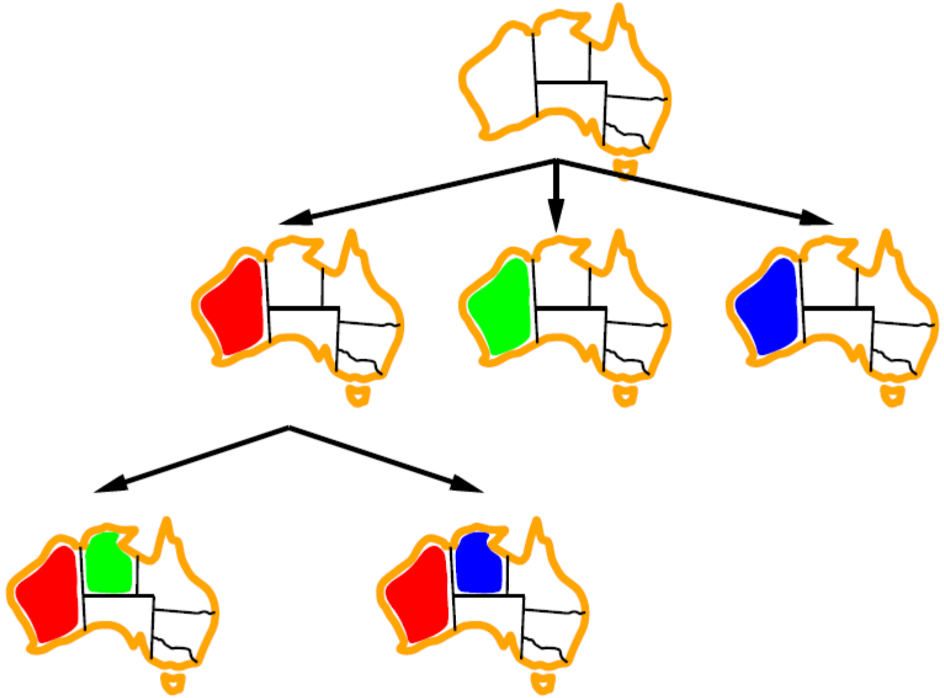
Backtracking Example

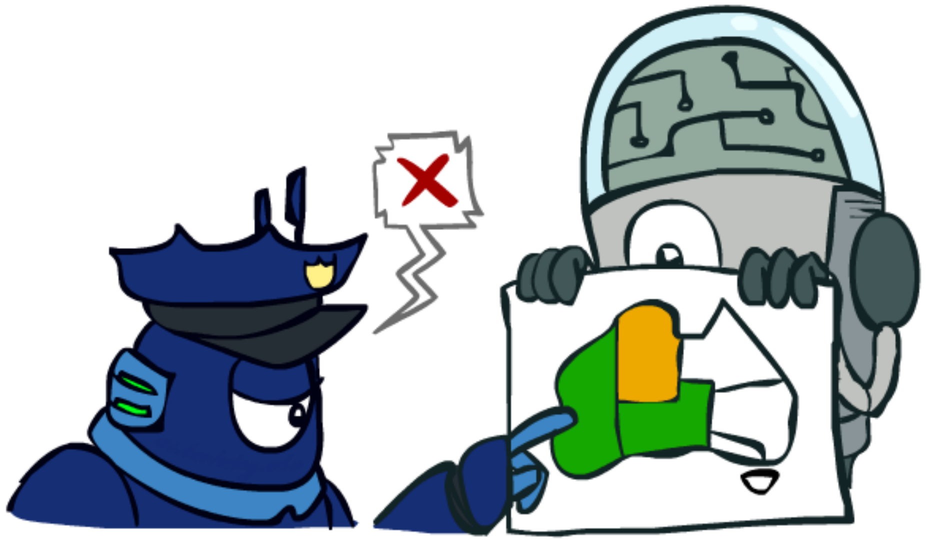


Backtracking Example

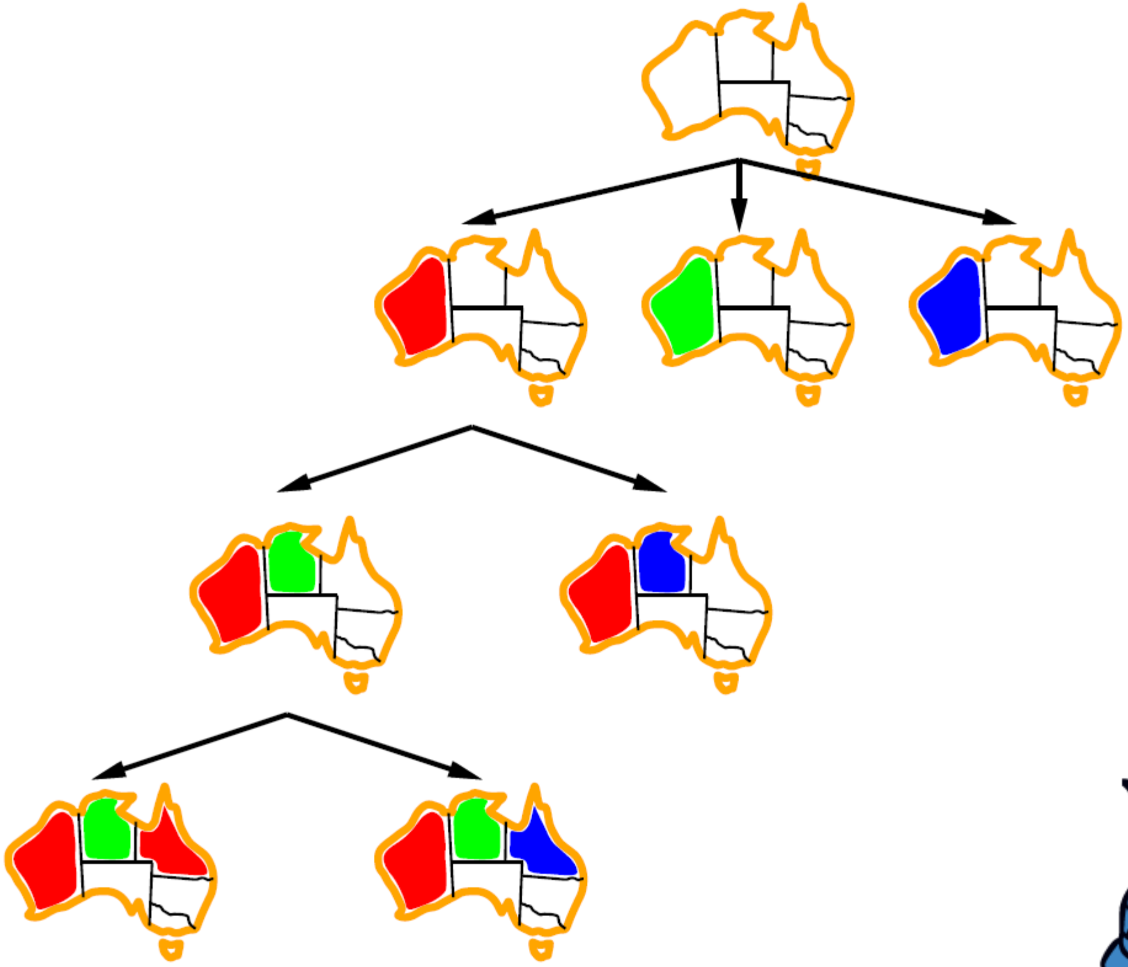


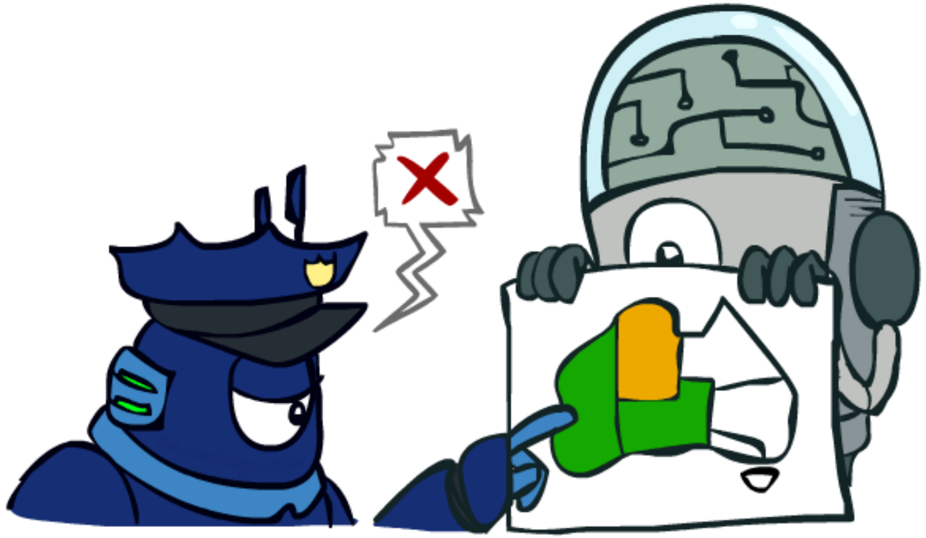
Backtracking Example





Backtracking Example





Improving Backtracking

- General-purpose ideas give huge gains in speed
  - … but it's all still NP-hard

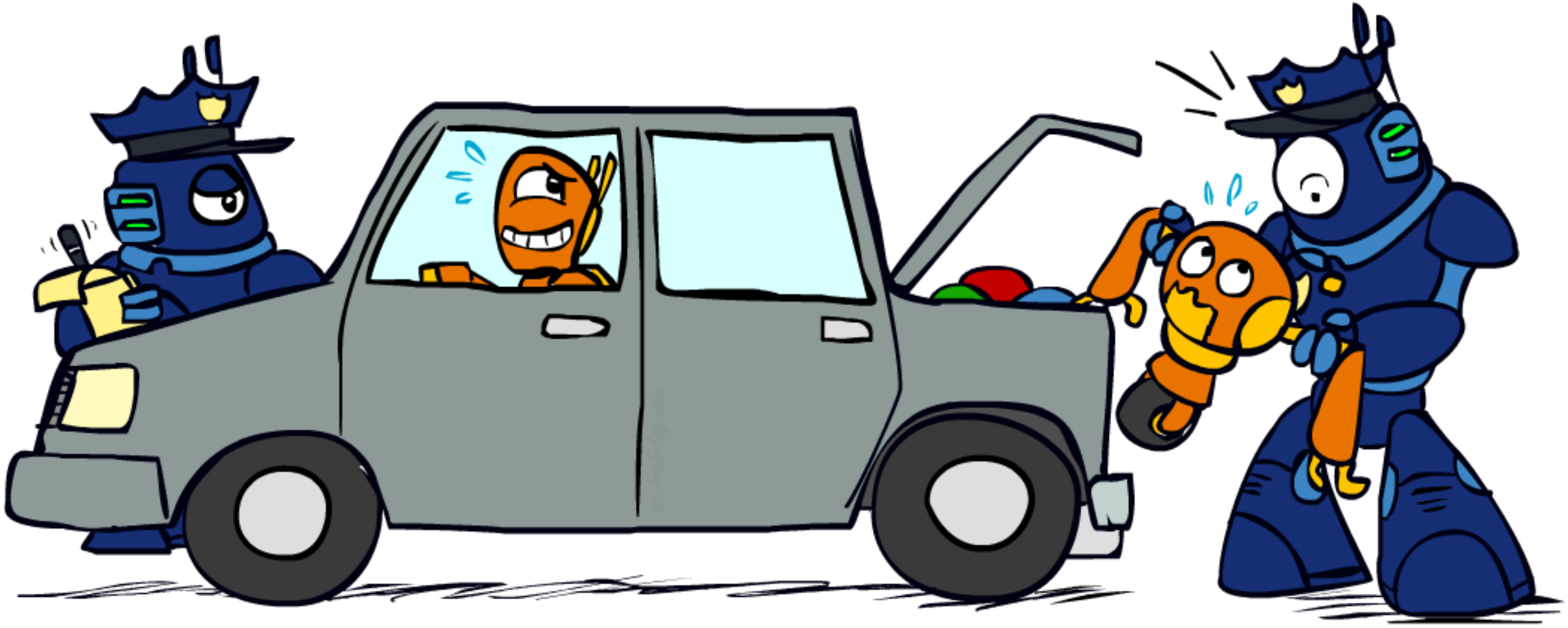
- Filtering: Can we detect inevitable failure early?

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

- Structure: Can we exploit the problem structure?

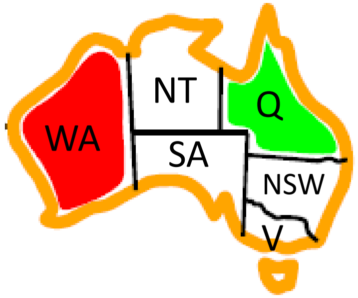
Arc Consistency and Beyond

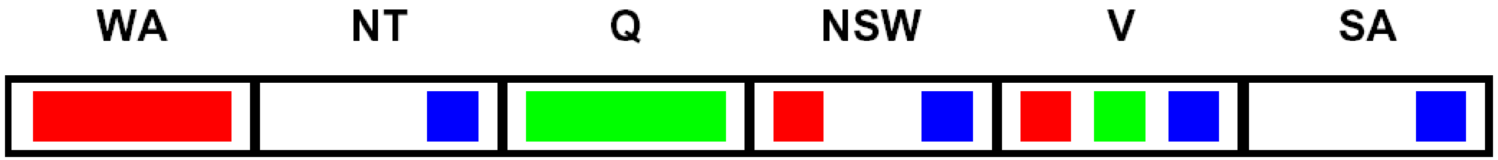
Arc Consistency and Beyond



Arc Consistency of an Entire CSP

- A simple form of propagation makes sure all arcs are consistent:

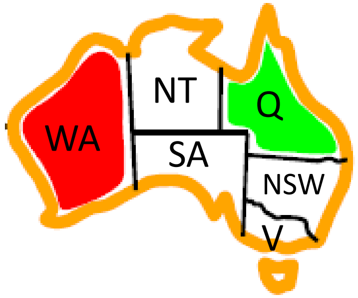


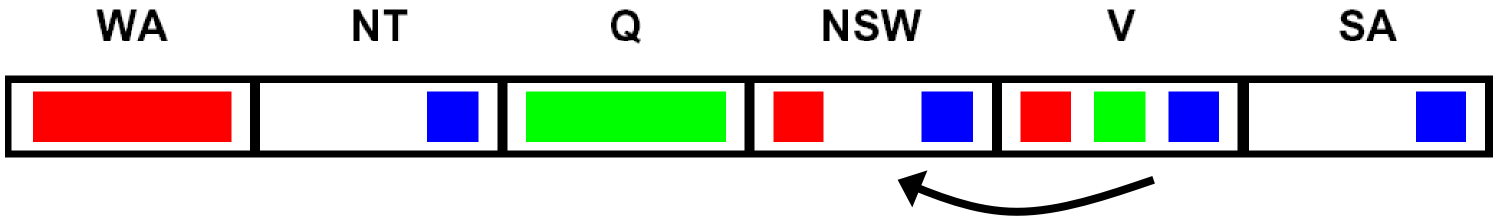


Remember: Delete from the tail!

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure all arcs are consistent:

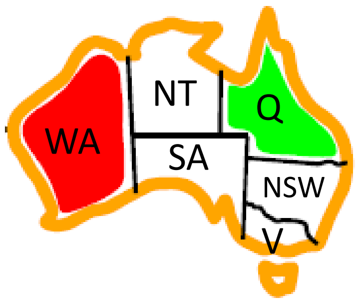


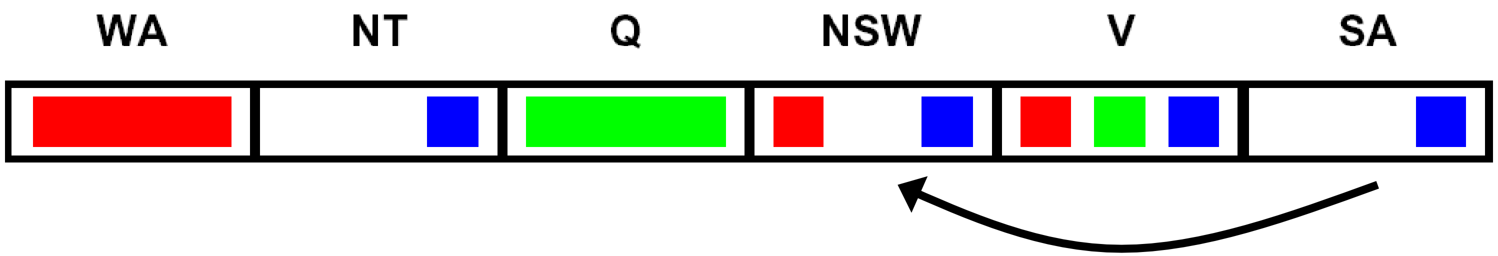


Remember: Delete from the tail!

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure all arcs are consistent:

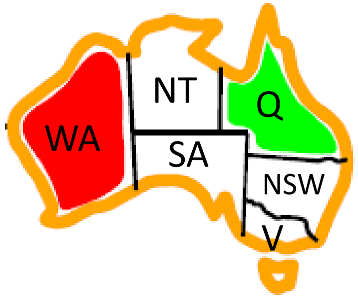


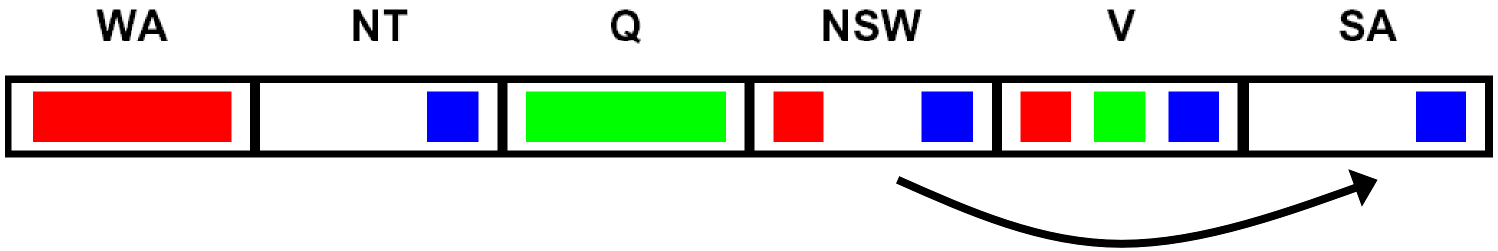


Remember: Delete from the tail!

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure all arcs are consistent:

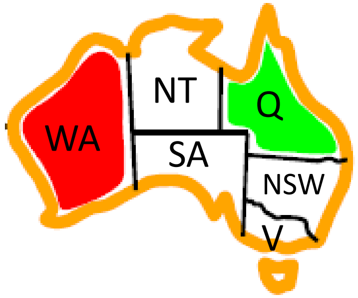


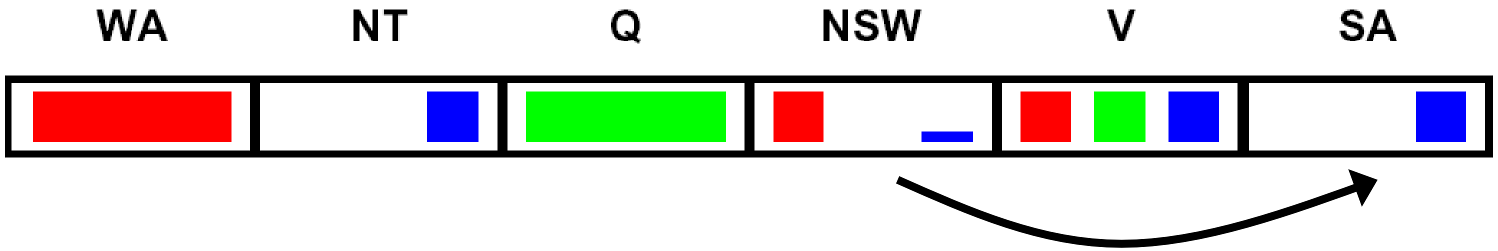


Remember: Delete from the tail!

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure all arcs are consistent:

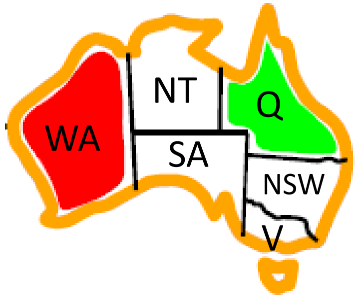


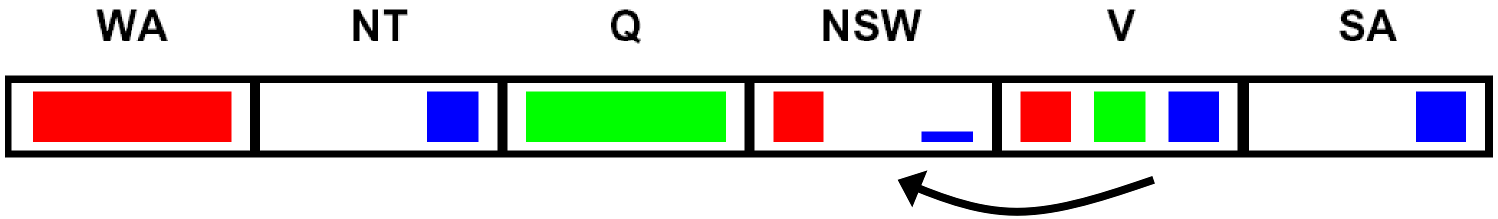


Remember: Delete from the tail!

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure all arcs are consistent:

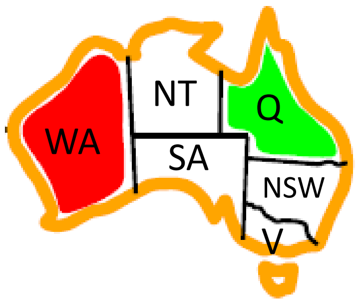


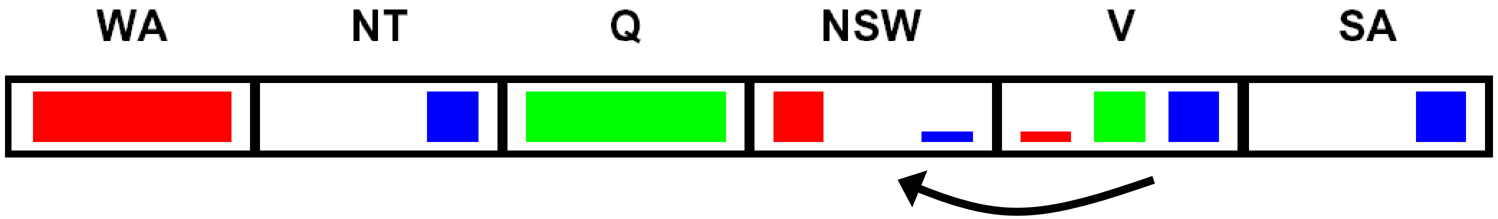


Remember: Delete from the tail!

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure all arcs are consistent:

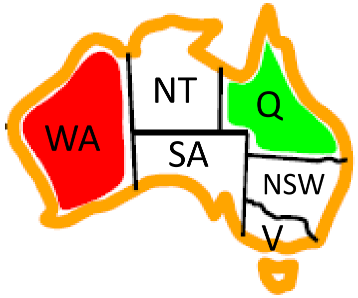


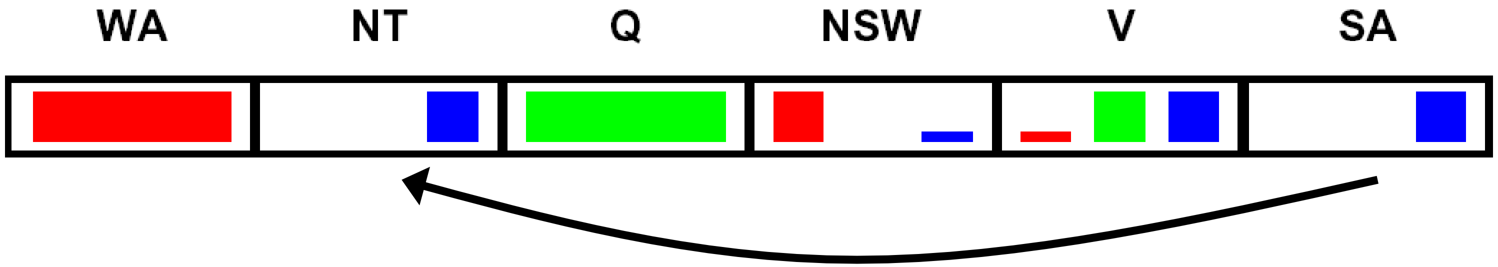


Remember: Delete from the tail!

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure all arcs are consistent:

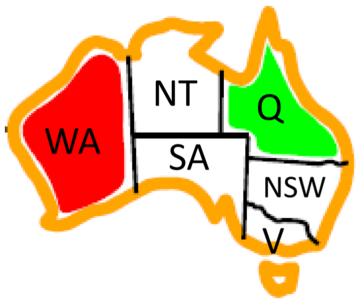


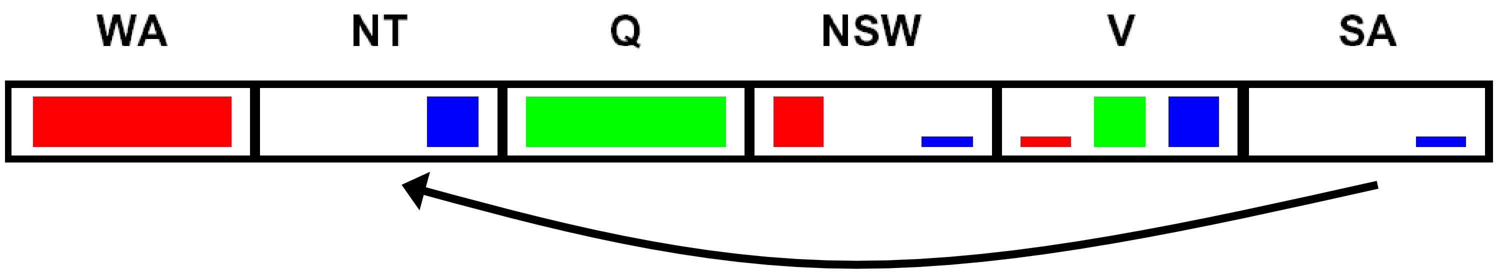


Remember: Delete from the tail!

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 {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

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

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

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

[Demo: CSP applet (made available by aispace.org) -- n-queens]

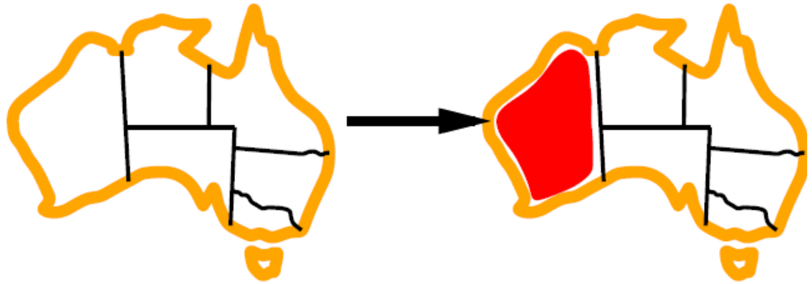
Ordering

Ordering: Minimum Remaining Values

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

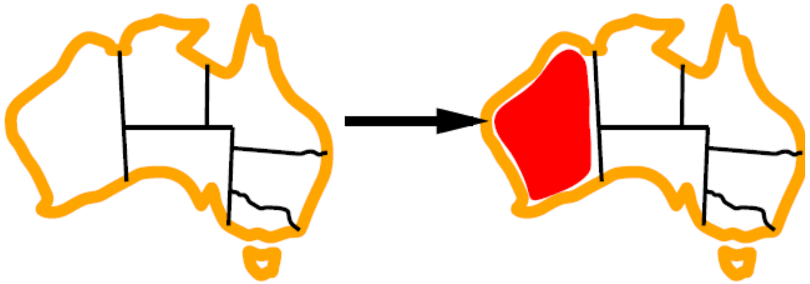
Ordering: Minimum Remaining Values

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



Ordering: Minimum Remaining Values

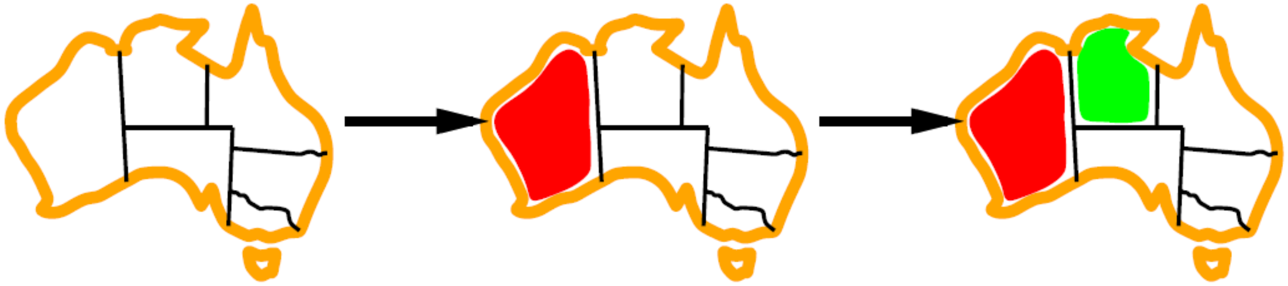
- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain
  - Aka most constrained variables



After assigning value to WA, both NT and SA have only two values in their domains
– choose one of them rather than Q, NSW, V or T

Ordering: Minimum Remaining Values

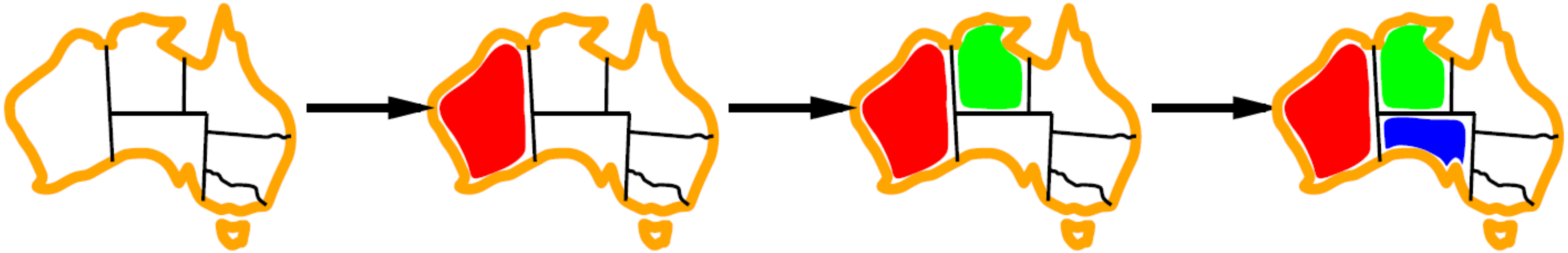
- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain
  - Aka most constrained variables



After assigning value to WA, both NT and SA have only two values in their domains
– choose one of them rather than Q, NSW, V or T

Ordering: Minimum Remaining Values

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

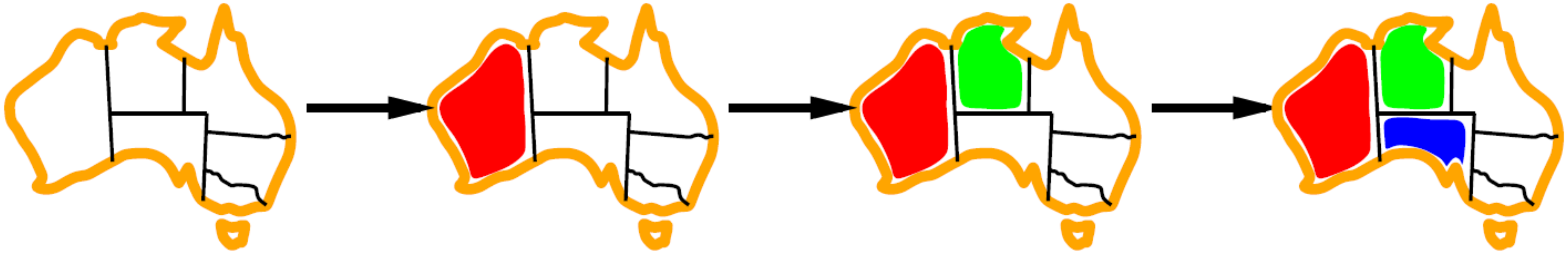


After assigning value to WA, both NT and SA have only two values in their domains

– choose one of them rather than Q, NSW, V or T

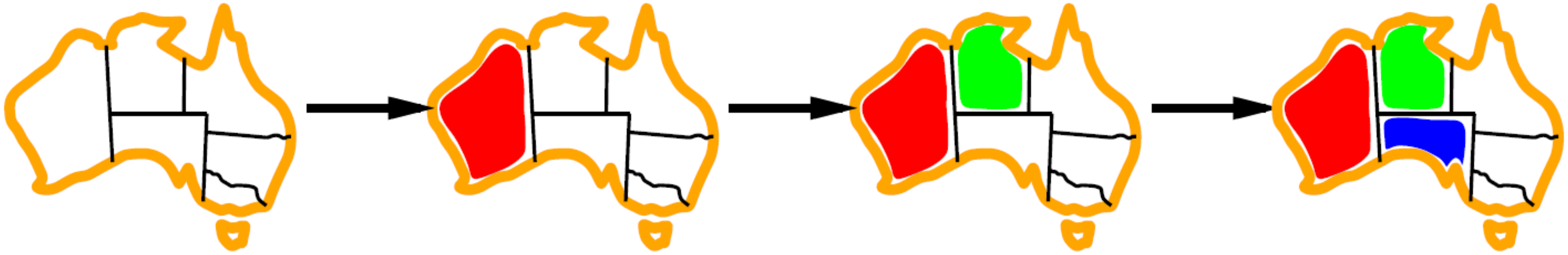
Ordering: Minimum Remaining Values

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



Ordering: Minimum Remaining Values

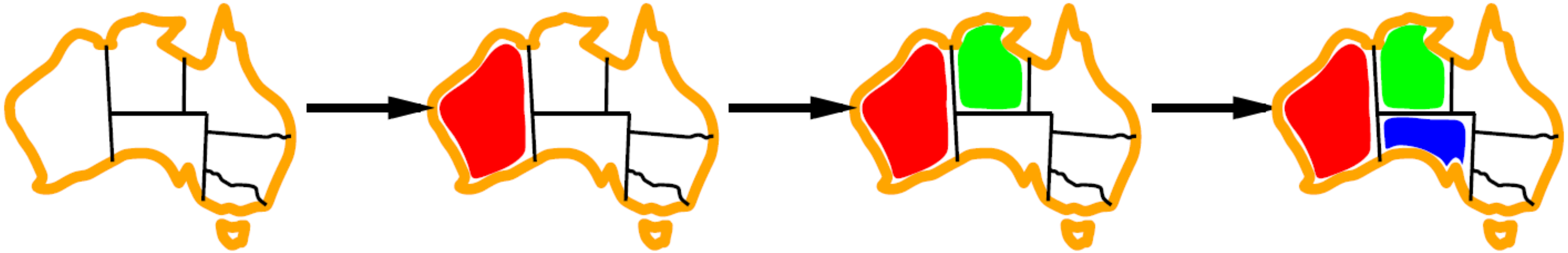
- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain
  - Aka most constrained variables



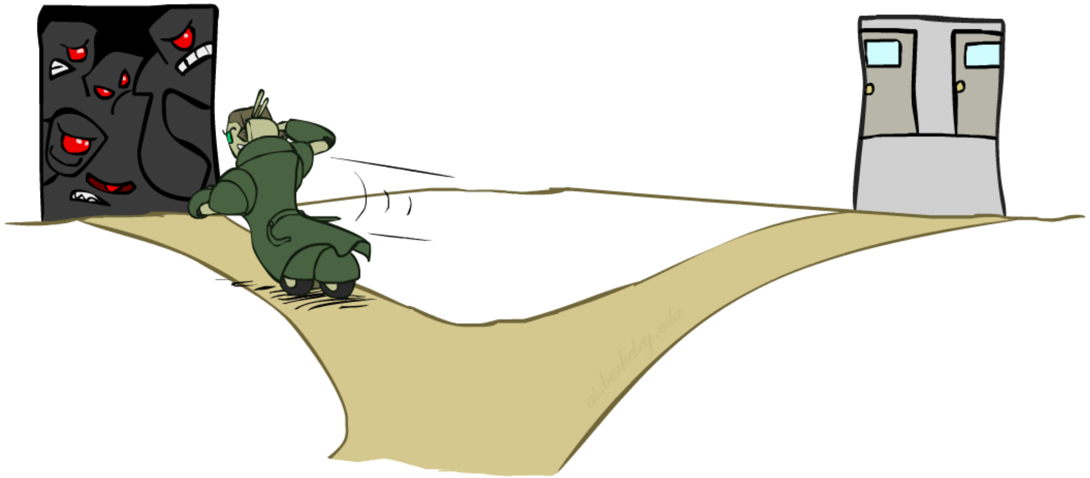
- Why min rather than max?

Ordering: Minimum Remaining Values

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

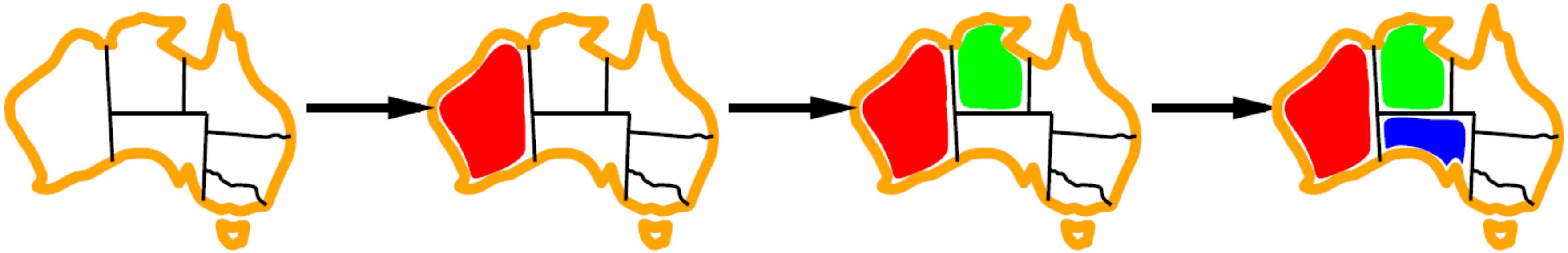


- Why min rather than max?

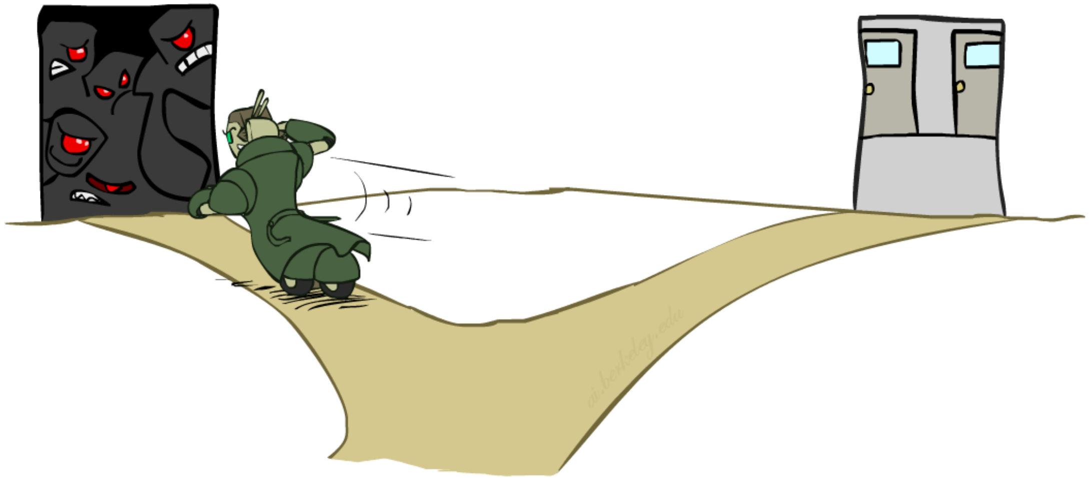


Ordering: Minimum Remaining Values

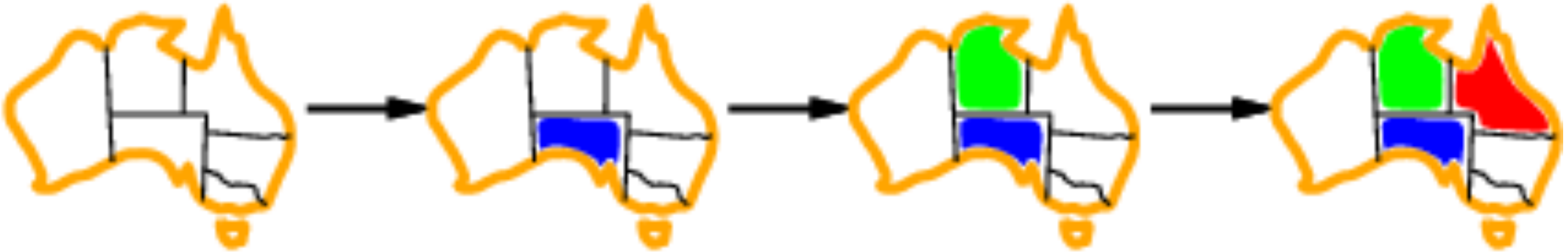
- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain
  - Aka most constrained variables



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



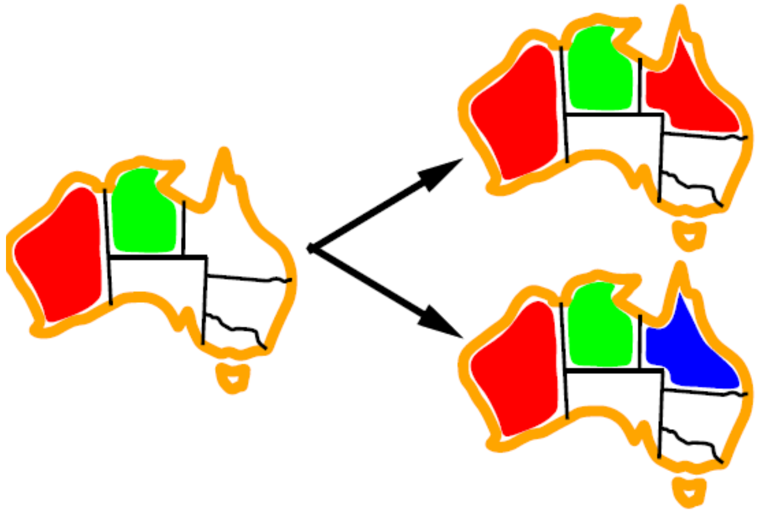
- Tie-breaker among Minimum remaining values
- Choose variable involved in largest # of constraints on remaining variables



- After assigning SA to be blue, WA, NT, Q, NSW and V all have just two values left.
- But WA and V have only one constraint (WA has constraint with NT, and V with NSW) on remaining variables and T none, so choose one of NT, Q & NSW (each of which has 2 cons. left)

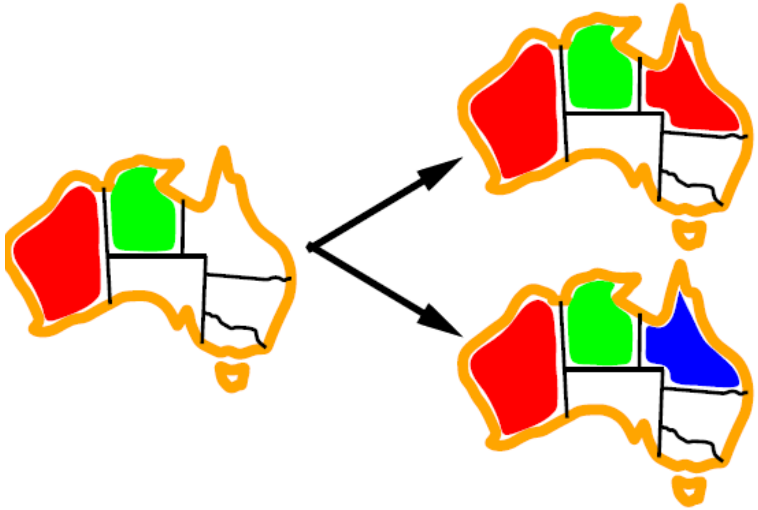
Ordering: Least Constraining Value

- 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)



Ordering: Least Constraining Value

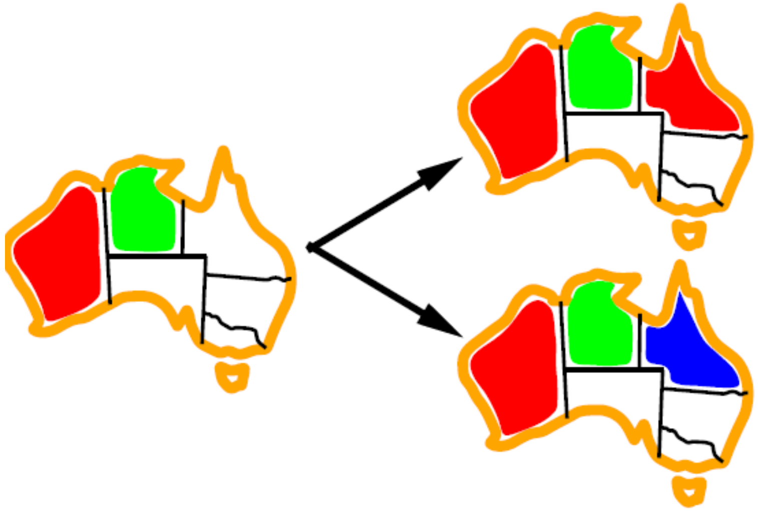
- Value Ordering: Least Constraining Value
  - Given a choice of variable, choose the *least constraining value*
  - I.e., the one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this! (E.g., rerunning filtering)



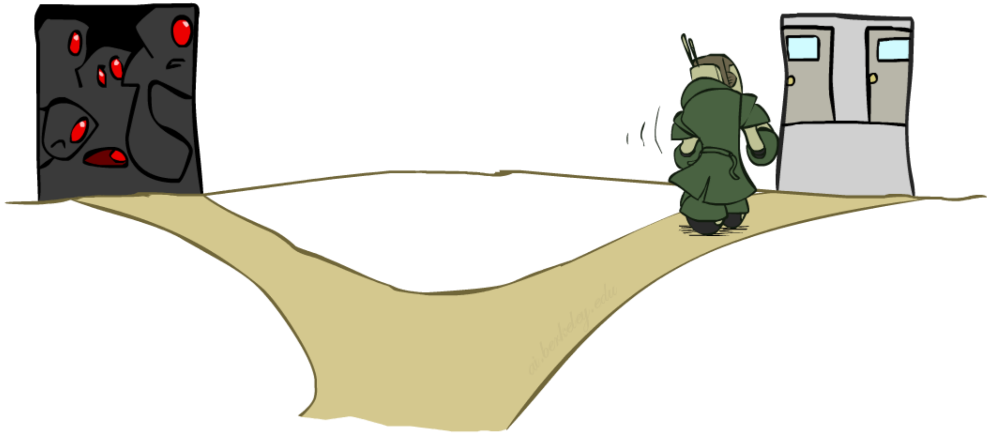
- Why least rather than most?

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)

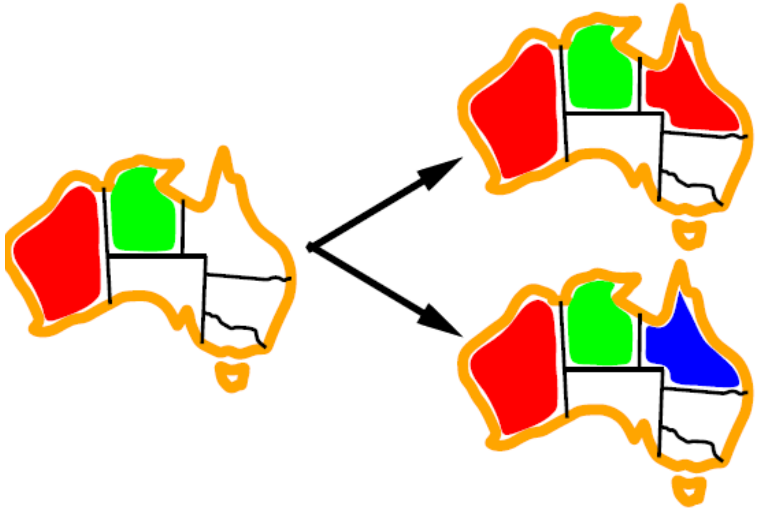


- Why least rather than most?



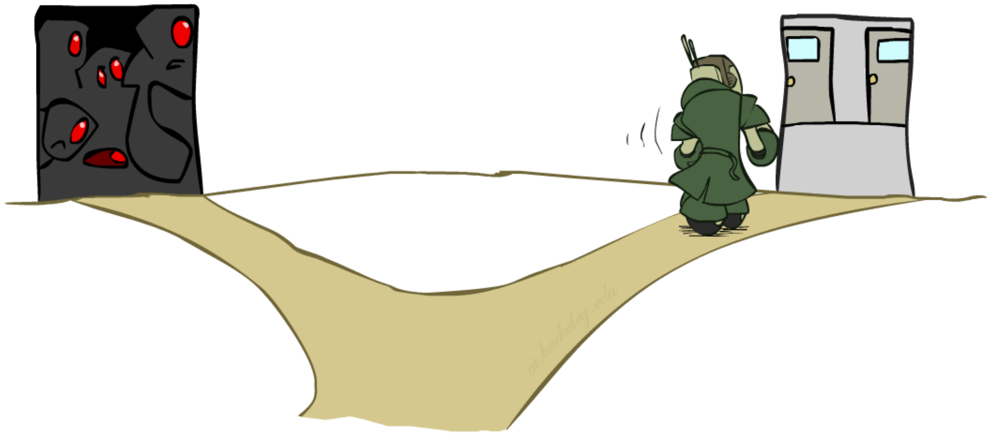
Ordering: Least Constraining Value

- 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)



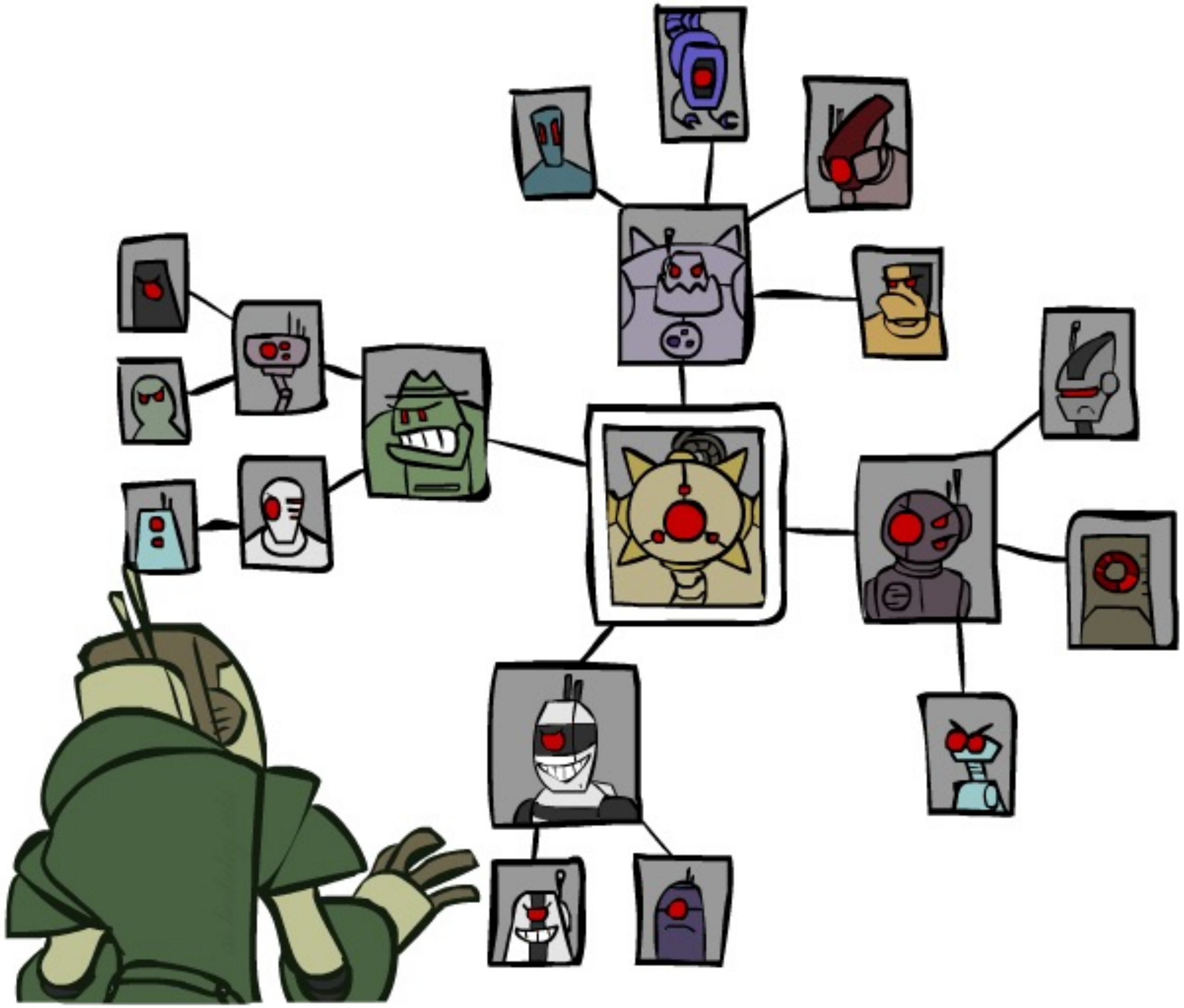
- Why least rather than most?

- Combining these ordering ideas makes 1000 queens feasible



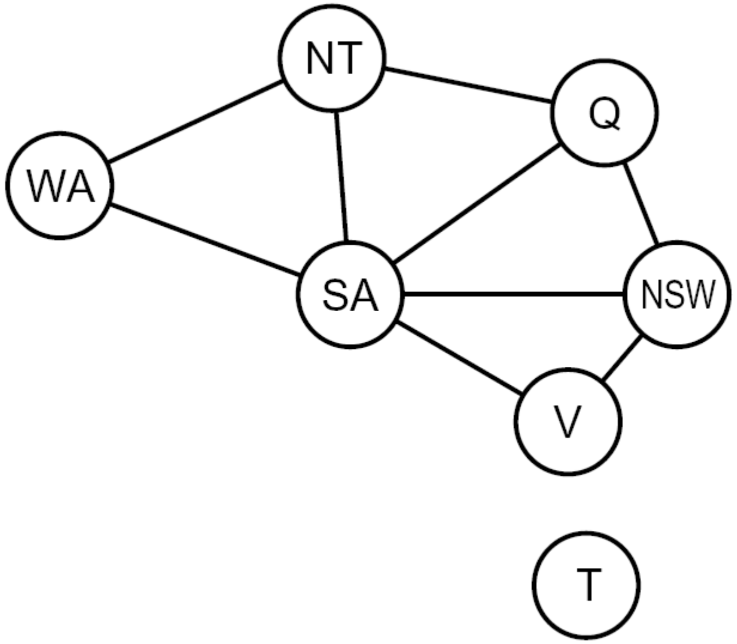
Demo: Coloring -- Backtracking + Forward Checking + Ordering

Structure

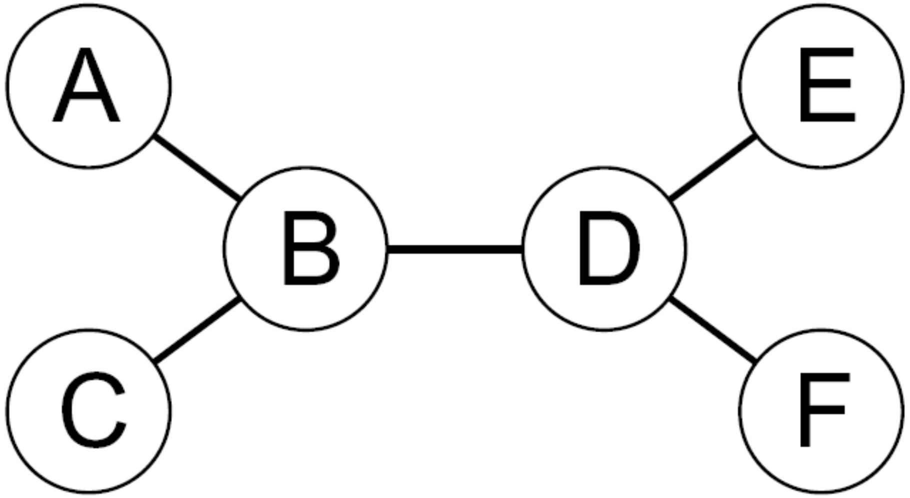


Problem Structure

- 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

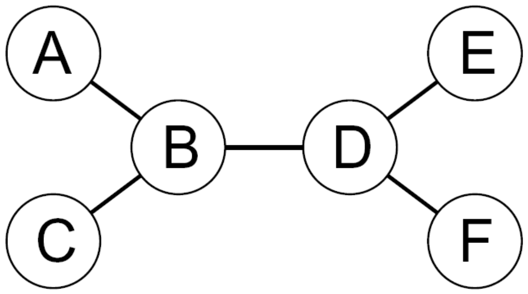


- Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\ d^2)$ time
  - Compare to general CSPs, where worst-case time is $O(d^n)$

- This property also applies to probabilistic reasoning (later): an example of the relation between syntactic restrictions and the complexity of reasoning

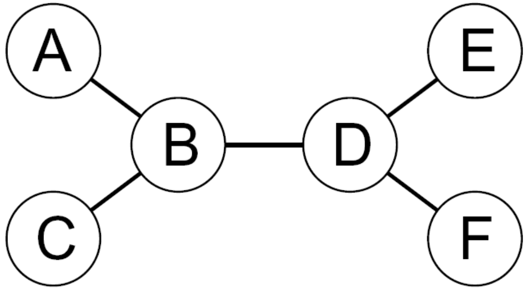
Tree-Structured CSPs

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



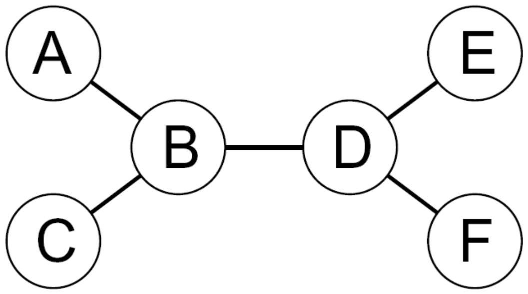
Tree-Structured CSPs

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



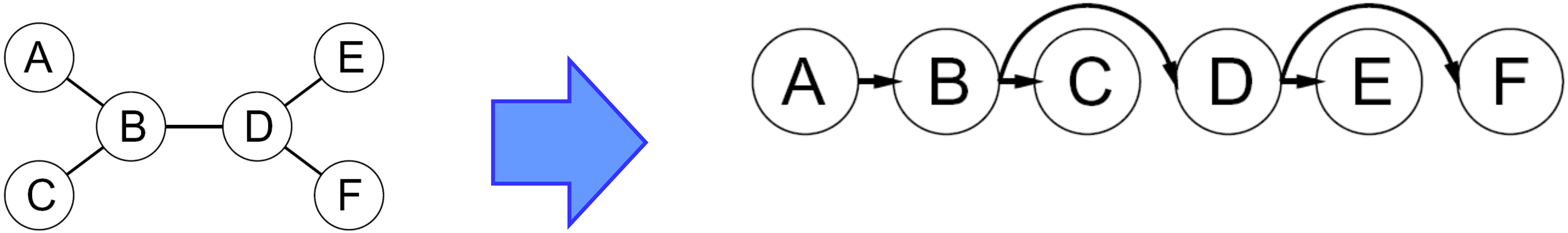
Tree-Structured CSPs

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



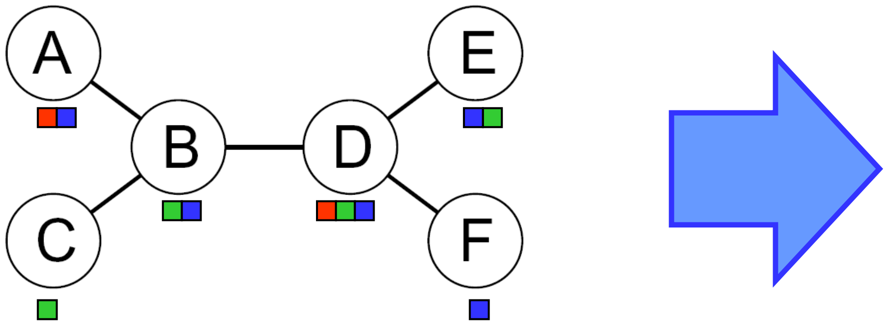
Tree-Structured CSPs

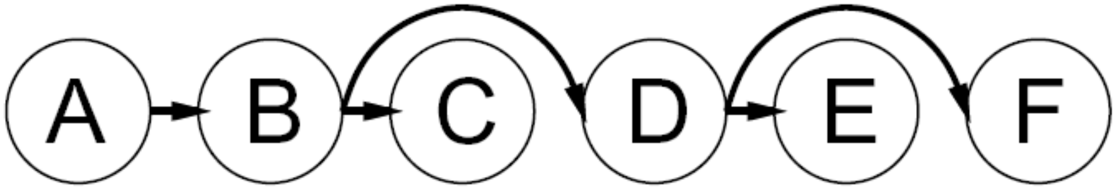
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



Tree-Structured CSPs

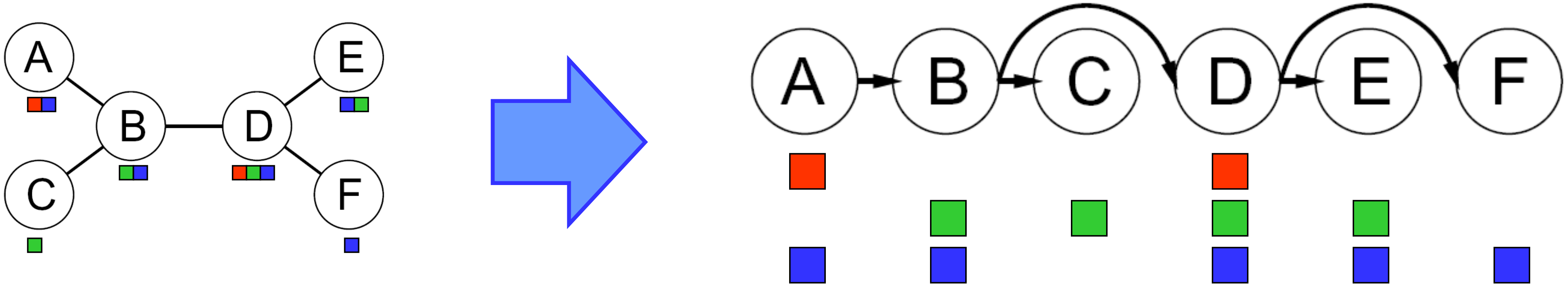
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children





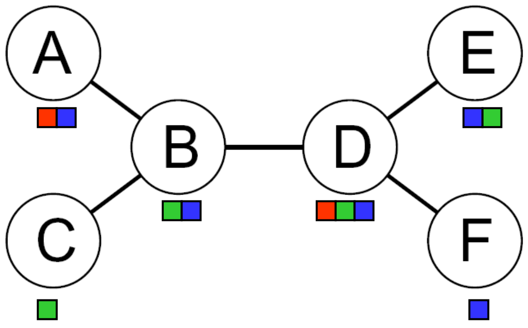
Tree-Structured CSPs

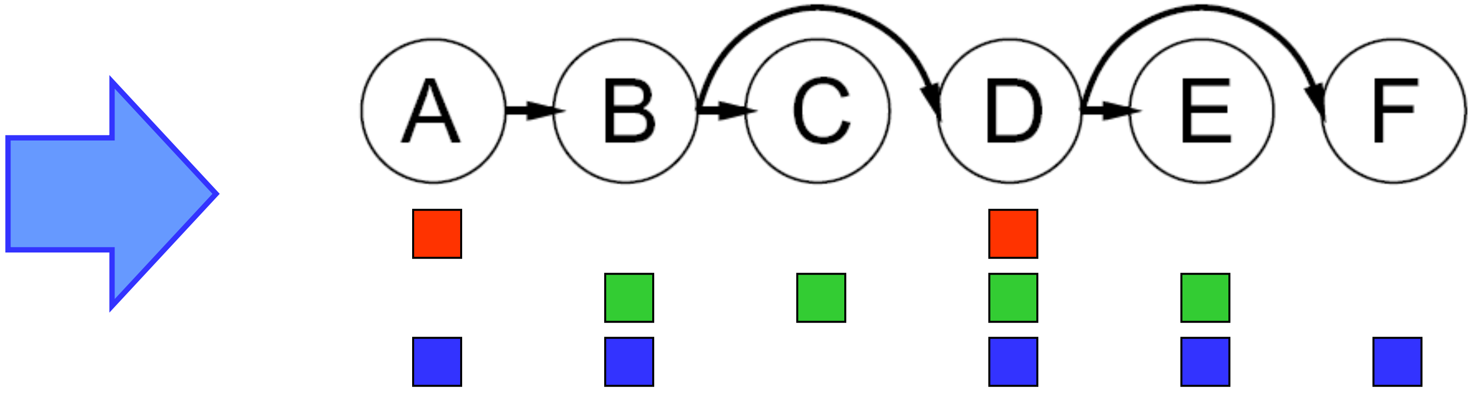
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



Tree-Structured CSPs

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

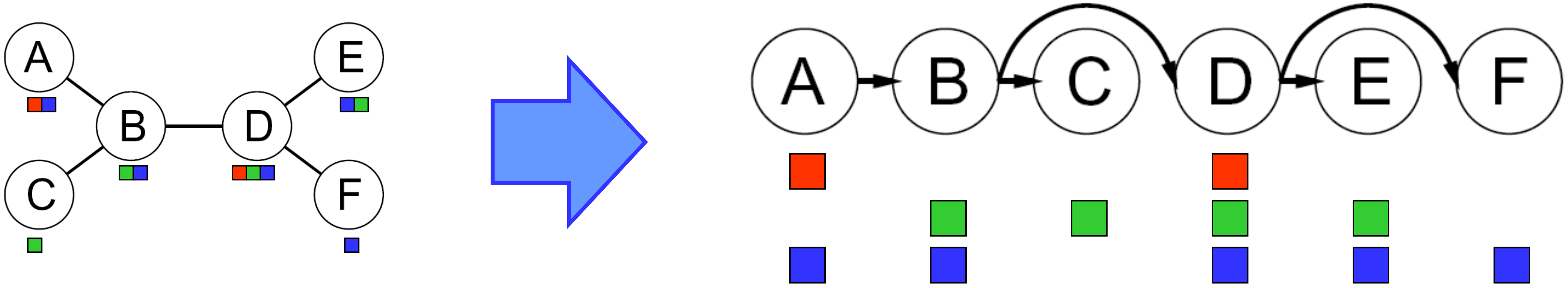




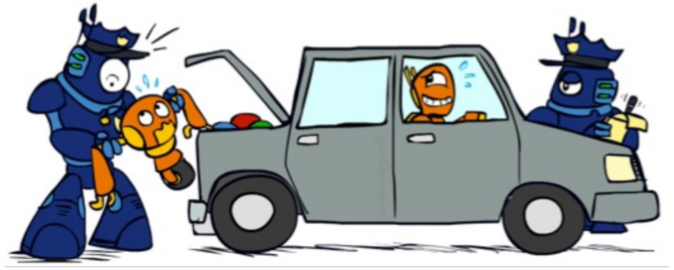
  - Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)

Tree-Structured CSPs

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

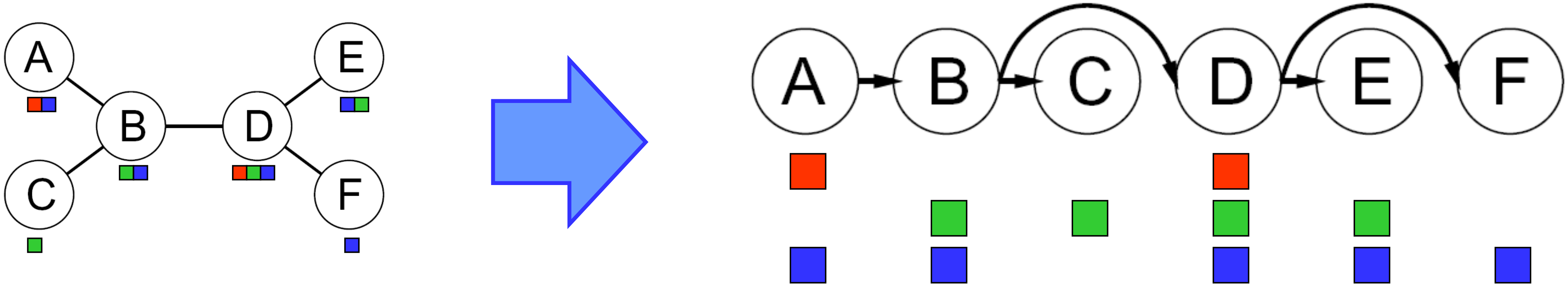


  - Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)

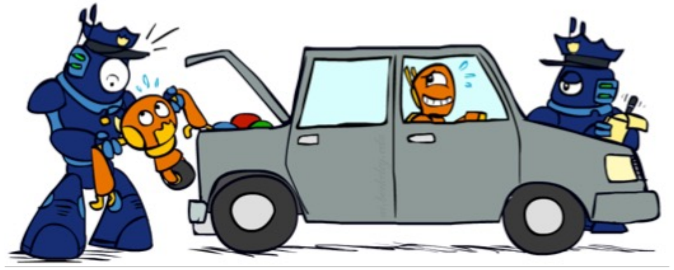


Tree-Structured CSPs

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

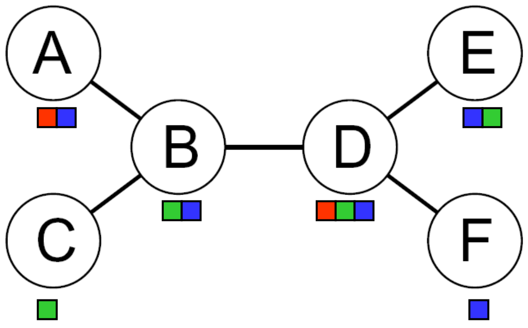


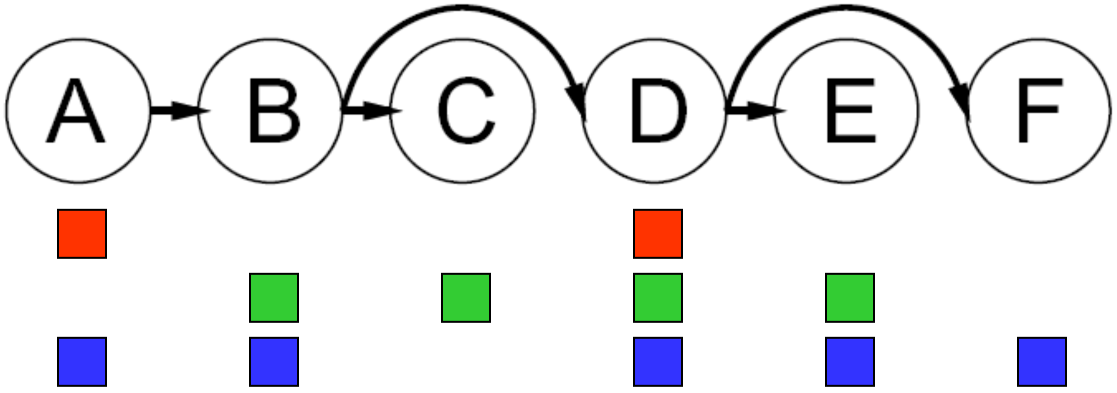
  - Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)
  - Assign forward: For i = 1 : n, assign X_i consistently with Parent(X_i)



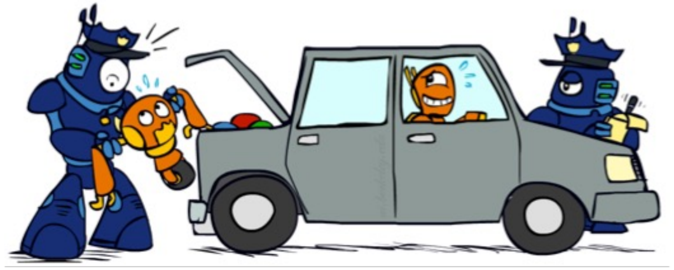
Tree-Structured CSPs

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

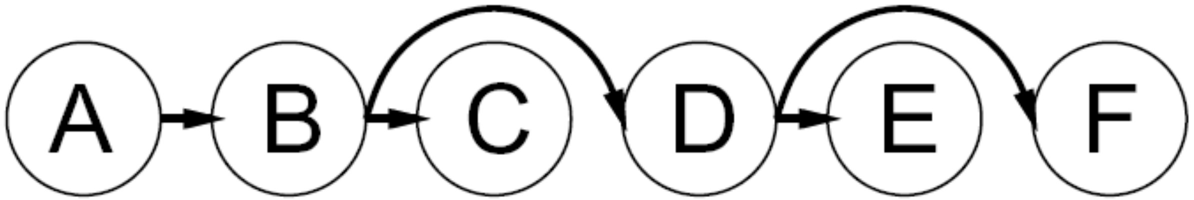




  - Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)
  - Assign forward: For i = 1 : n, assign X_i consistently with Parent(X_i)
- Runtime: $O(n\ d^2)$ (why?)

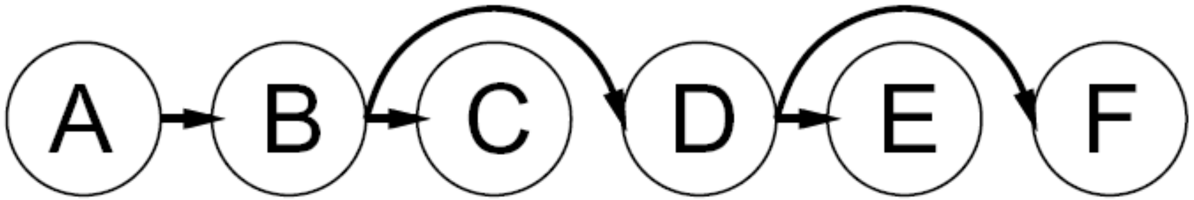


Tree-Structured CSPs



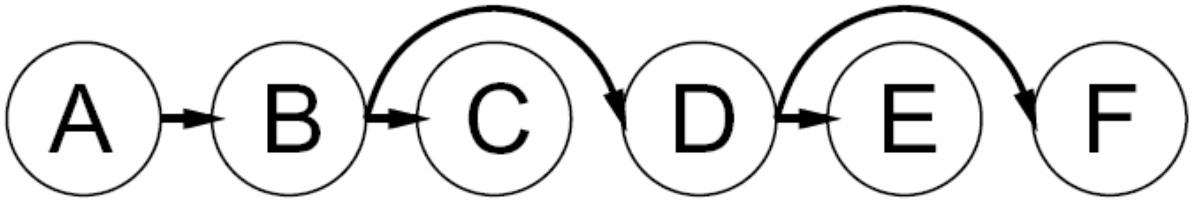
Tree-Structured CSPs

- Claim 1: After backward pass, all root-to-leaf arcs are consistent



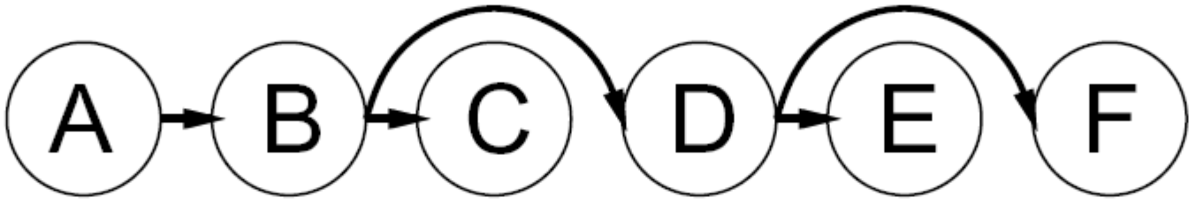
Tree-Structured CSPs

- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Proof: Each X→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)



Tree-Structured CSPs

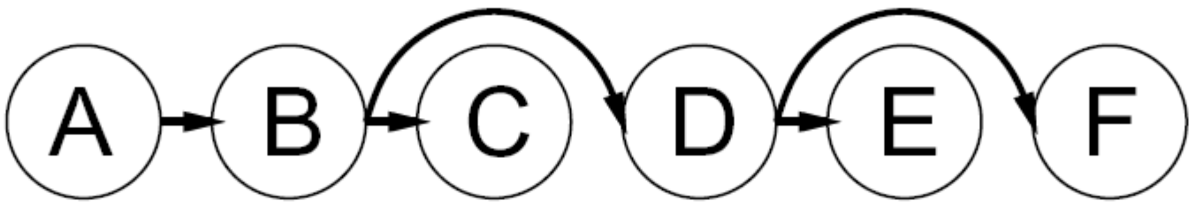
- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Proof: Each X→Y was made consistent at one point and Y's domain could not have been reduced thereafter (because Y's children were processed before Y)



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

Tree-Structured CSPs

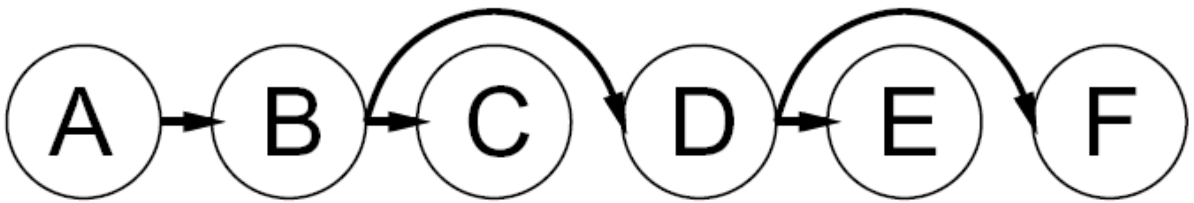
- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Proof: Each X→Y was made consistent at one point and Y's domain could not have been reduced thereafter (because Y's children were processed before Y)



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

Tree-Structured CSPs

- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Proof: Each X→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)

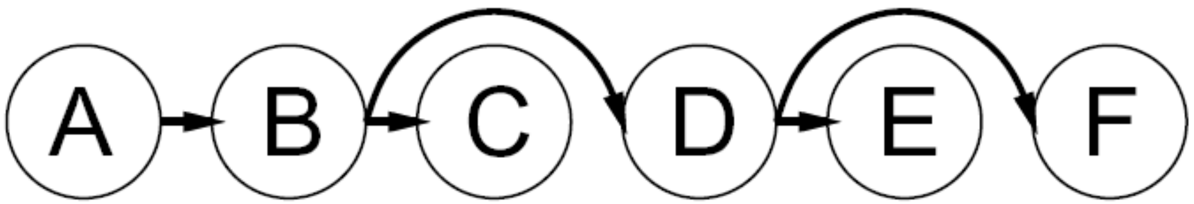


- Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
- Proof: Induction on position

- Why doesn't this algorithm work with cycles in the constraint graph?

Tree-Structured CSPs

- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Proof: Each X→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)

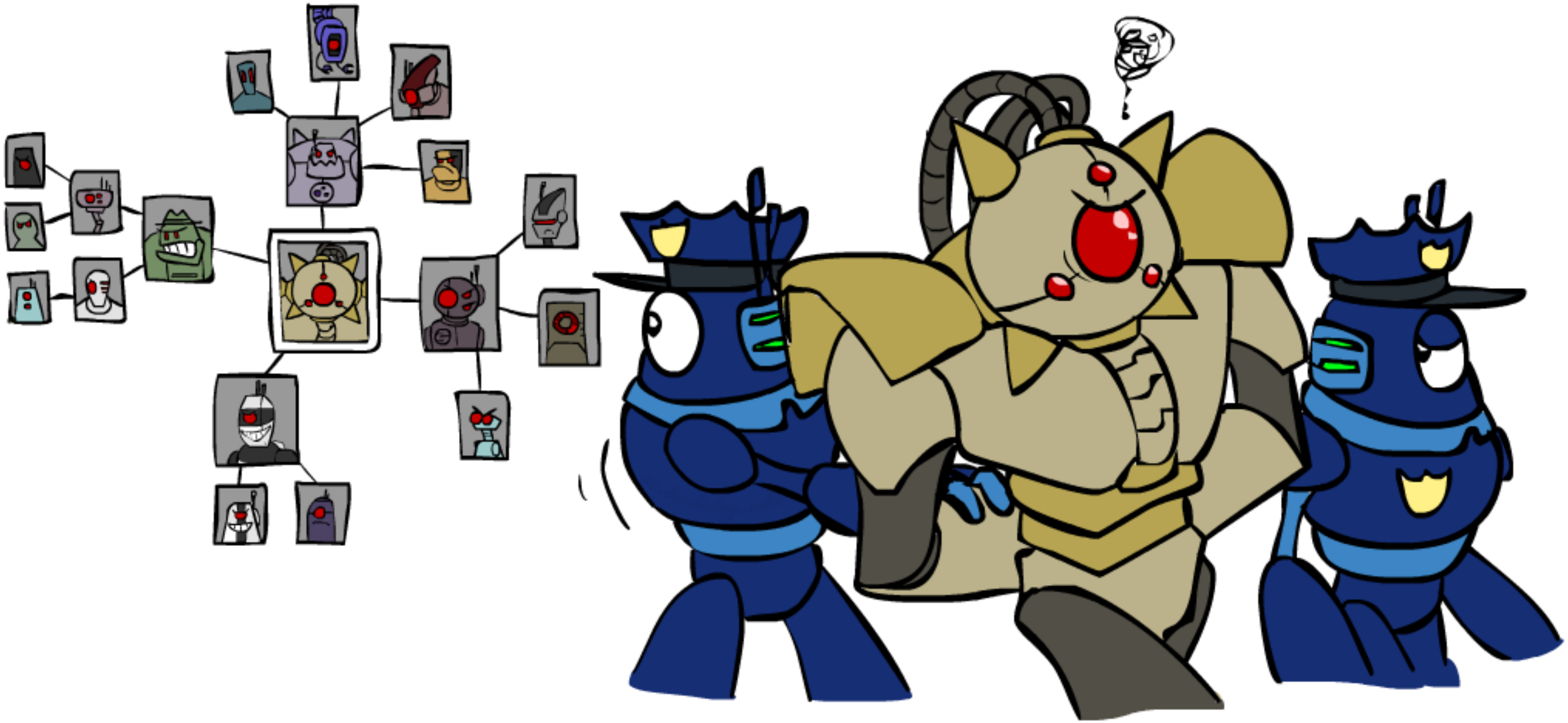


- Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
- Proof: Induction on position

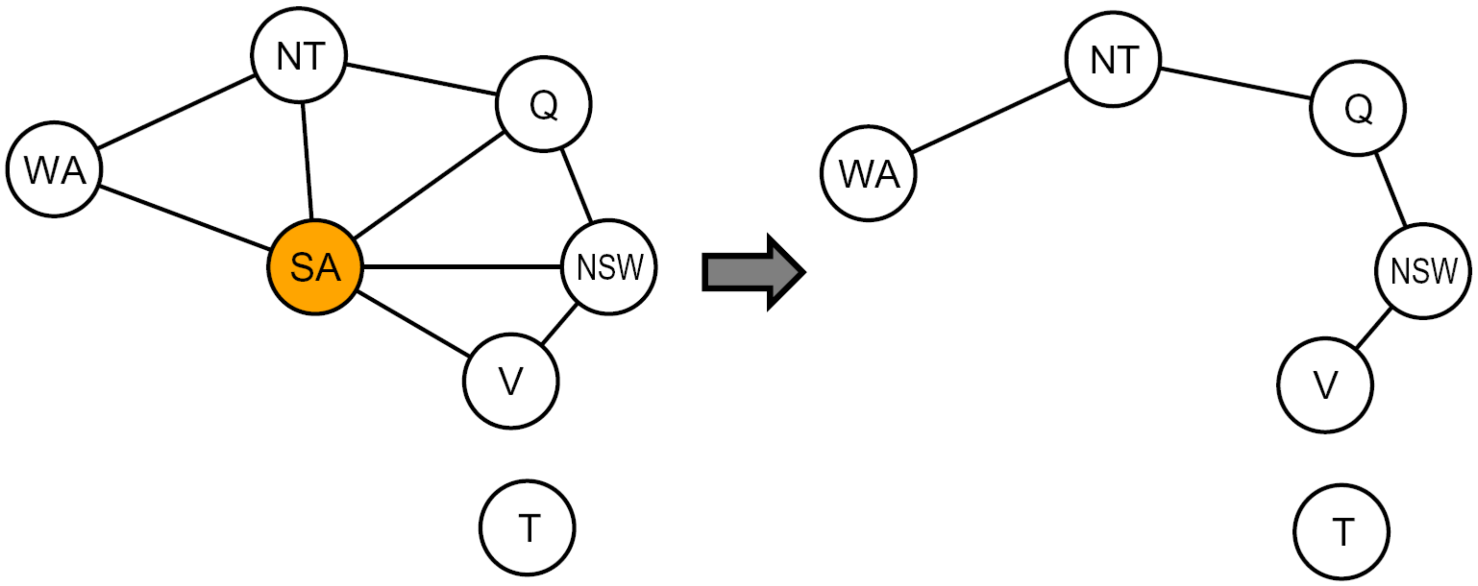
- Why doesn't this algorithm work with cycles in the constraint graph?

- Note: we'll see this basic idea again with Bayes' nets

Improving Structure



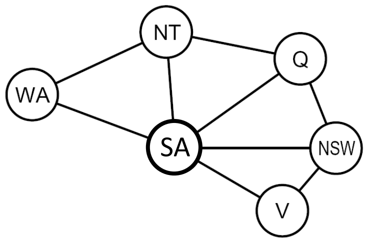
Nearly Tree-Structured CSPs



- 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

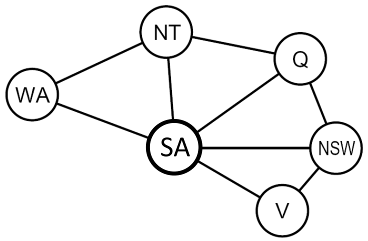
Cutset Conditioning

Cutset Conditioning



Cutset Conditioning

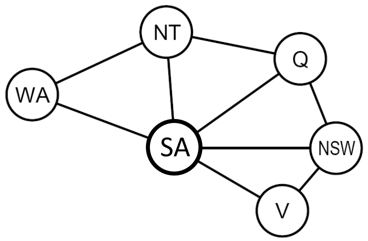
Choose a cutset



Cutset Conditioning

Choose a cutset

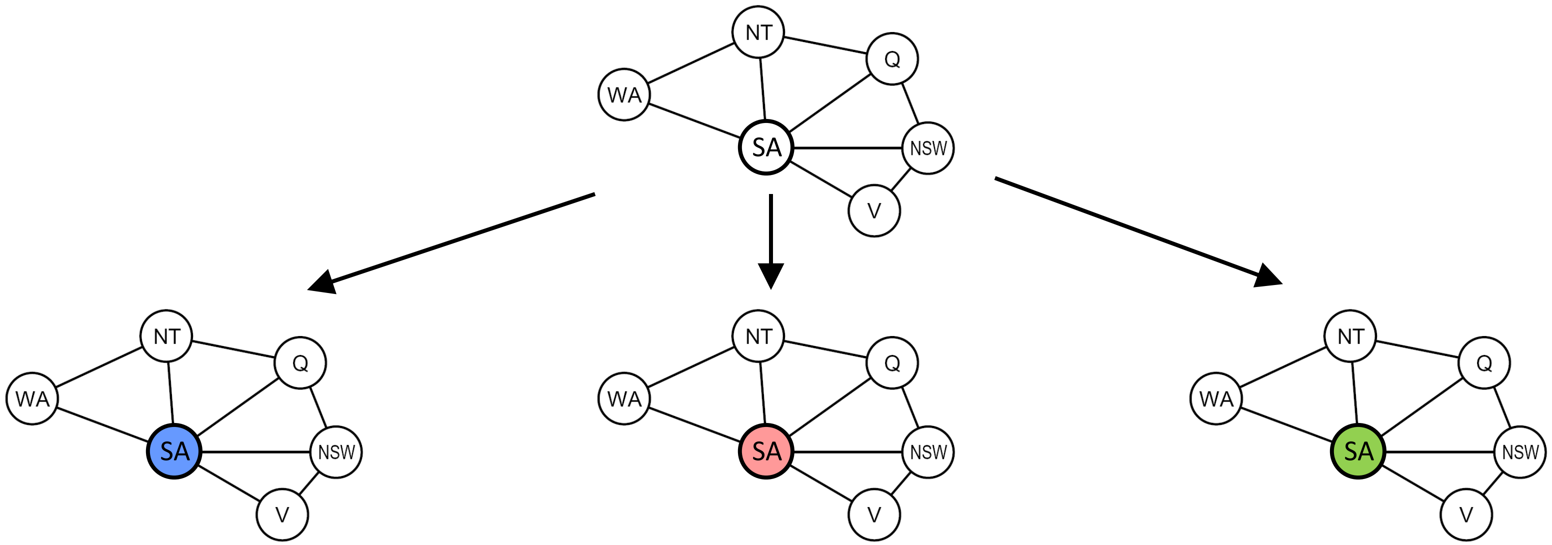
Instantiate the cutset
(all possible ways)



Cutset Conditioning

Choose a cutset

Instantiate the cutset
(all possible ways)

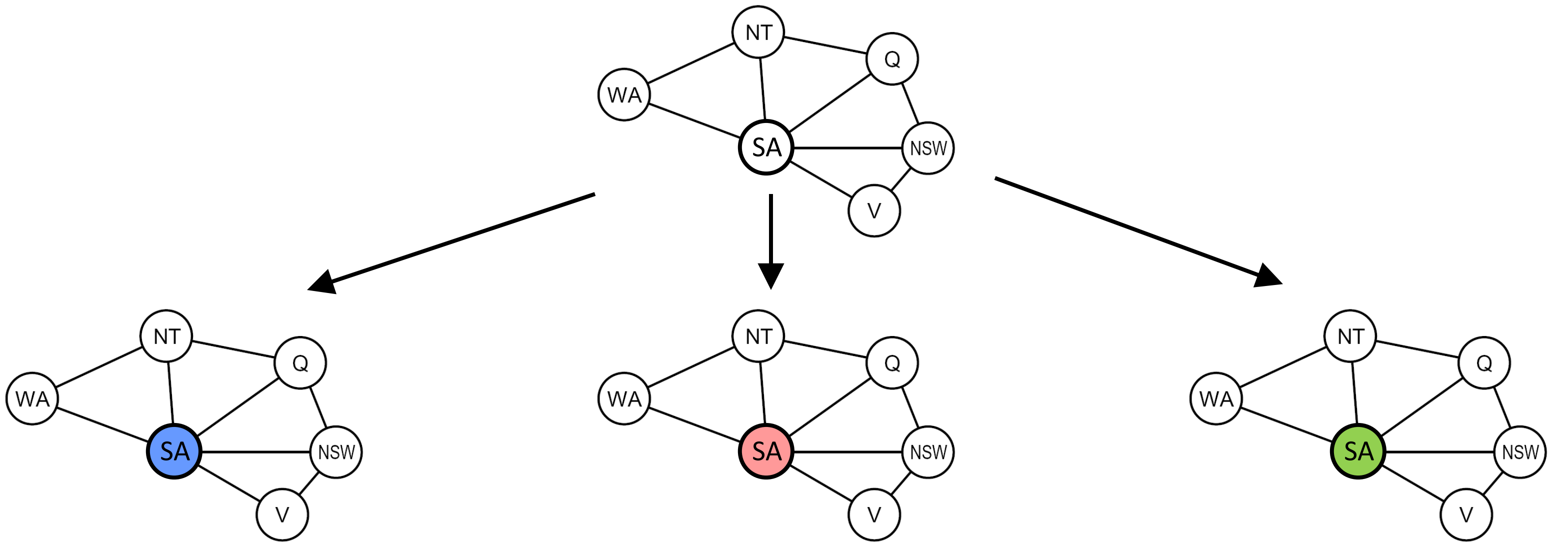


Cutset Conditioning

Choose a cutset

Instantiate the cutset (all possible ways)

Compute residual CSP for each assignment

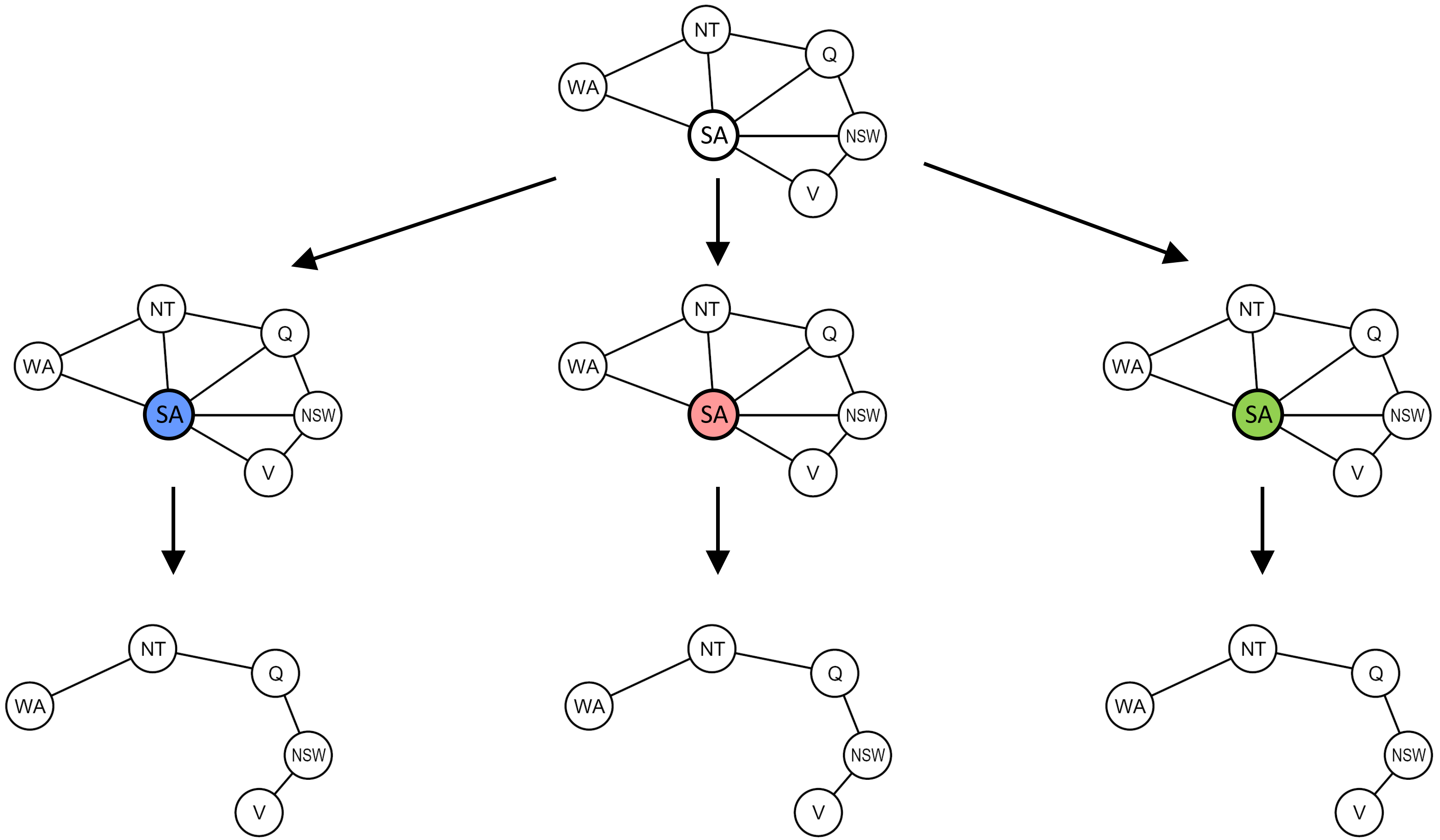


Cutset Conditioning

Choose a cutset

Instantiate the cutset (all possible ways)

Compute residual CSP for each assignment



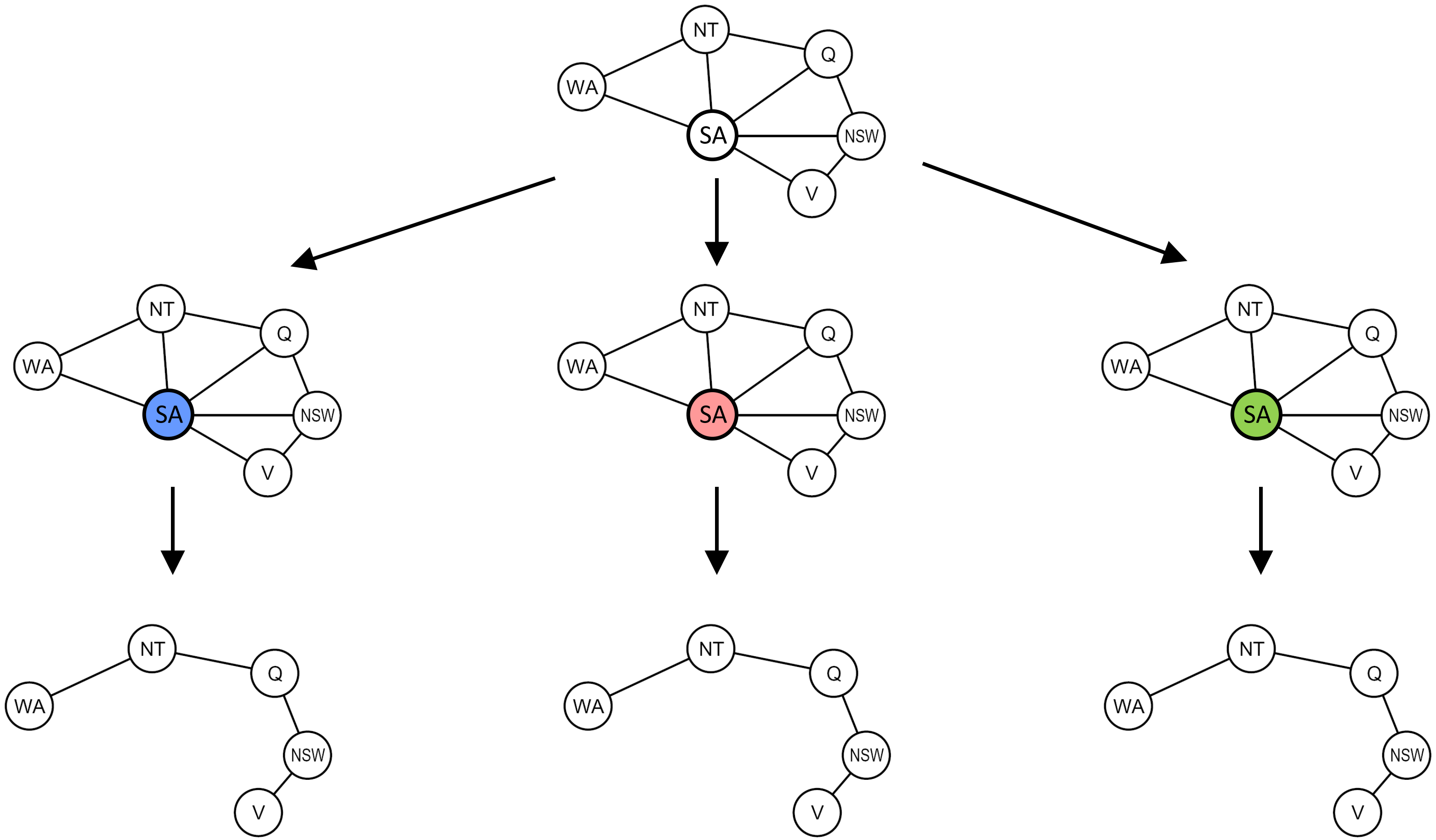
Cutset Conditioning

Choose a cutset

Instantiate the cutset (all possible ways)

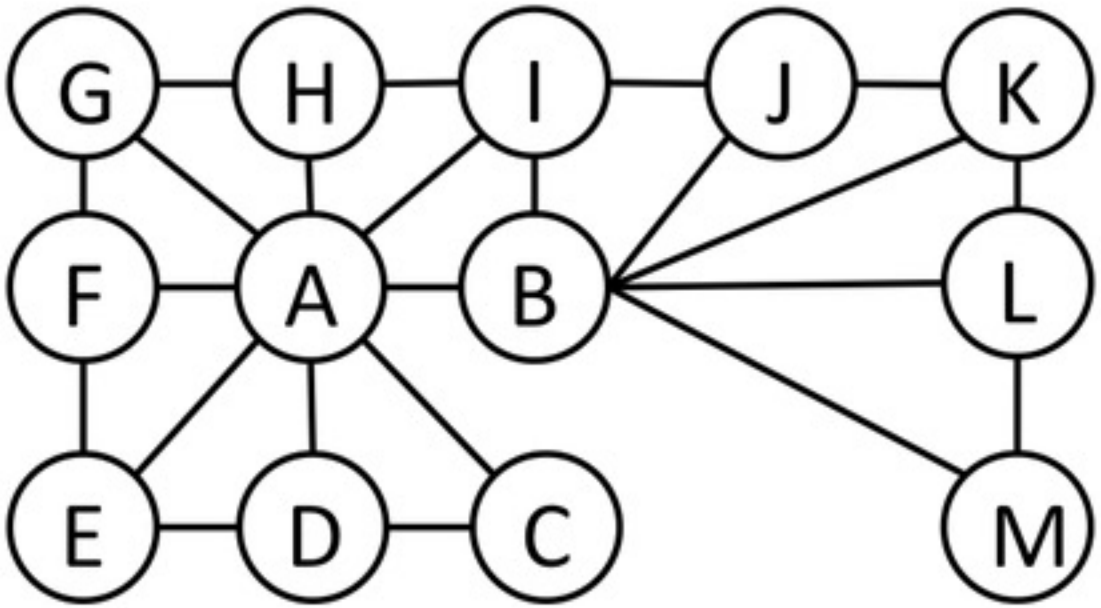
Compute residual CSP for each assignment

Solve the residual CSPs (tree structured)

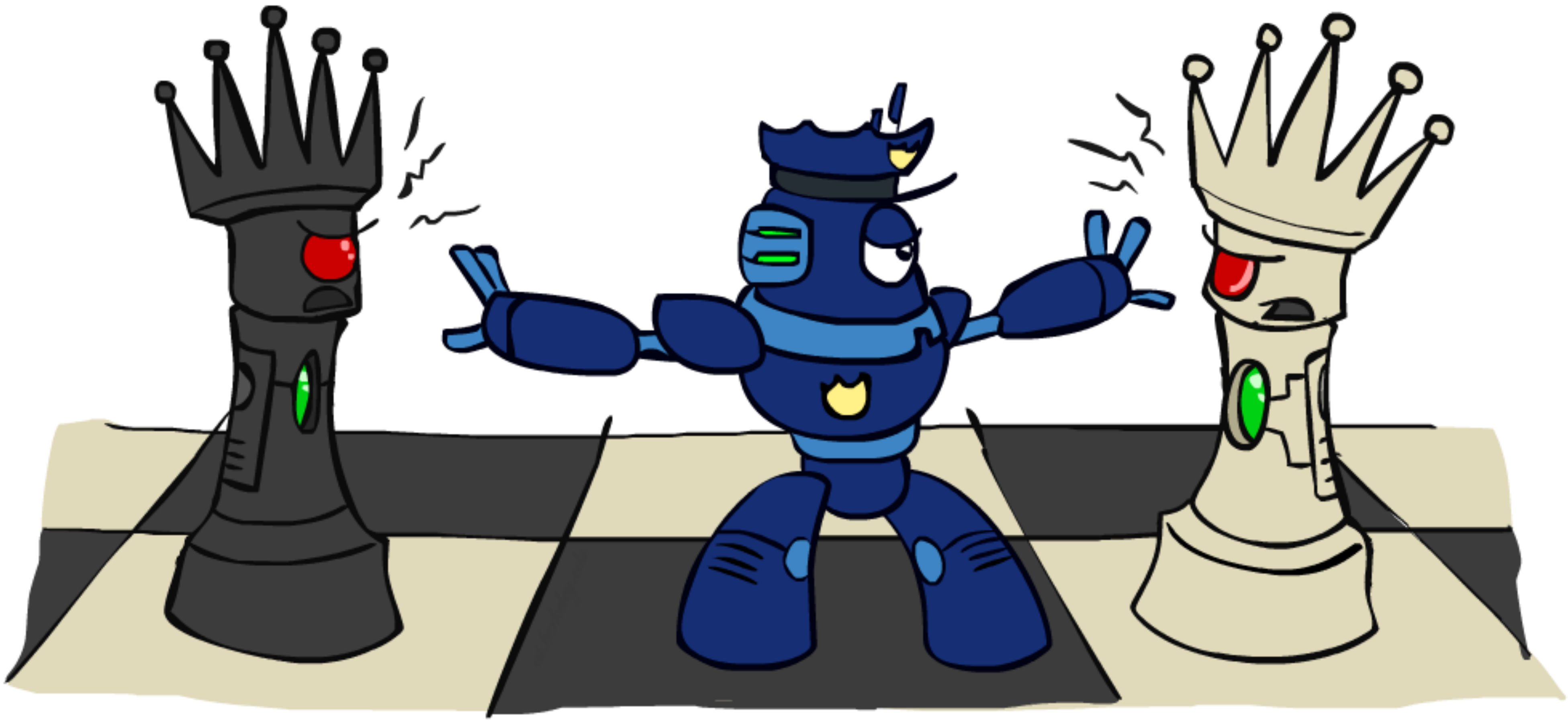


Cutset Quiz

- Find the smallest cutset for the graph below.



Iterative Improvement

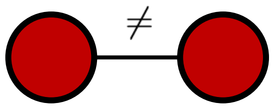


Iterative Algorithms for CSPs

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

Iterative Algorithms for CSPs

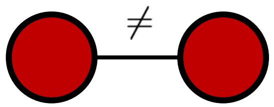
- Local search methods typically work with "complete" states, i.e., all variables assigned



Iterative Algorithms for CSPs

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

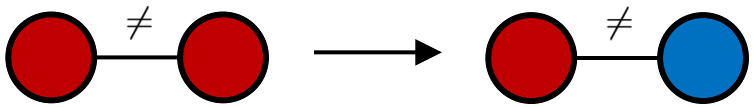
- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No fringe! Live on the edge.



Iterative Algorithms for CSPs

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

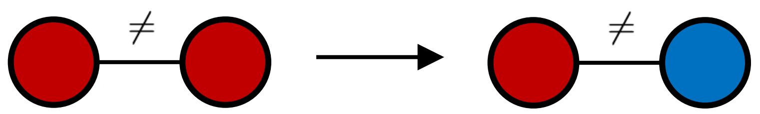
- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No fringe! Live on the edge.



Iterative Algorithms for CSPs

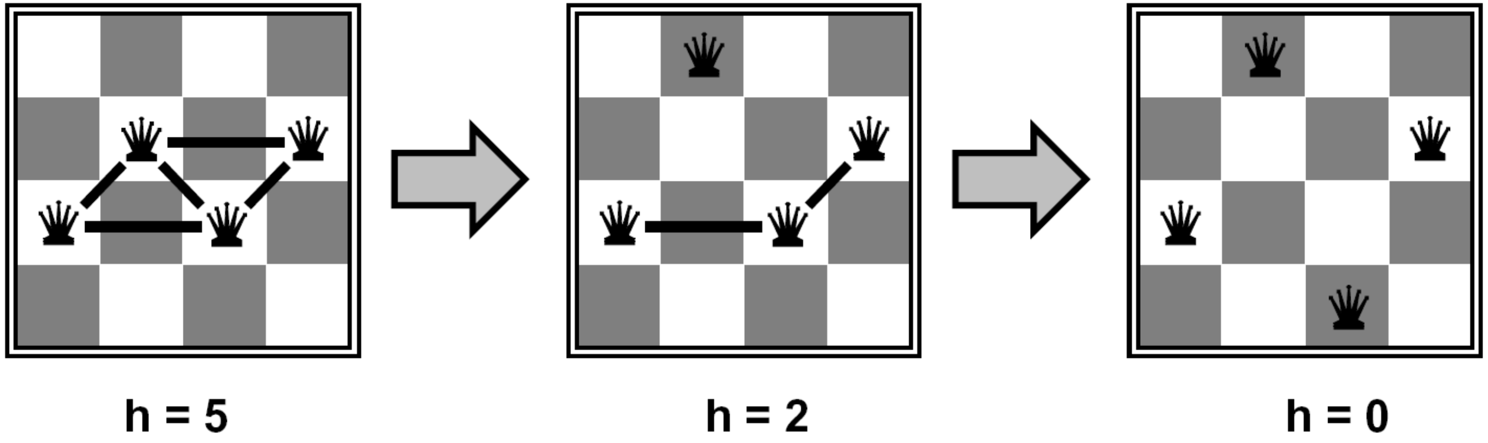
- Local search methods typically work with “complete” states, i.e., all variables assigned

- 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) = total number of violated constraints

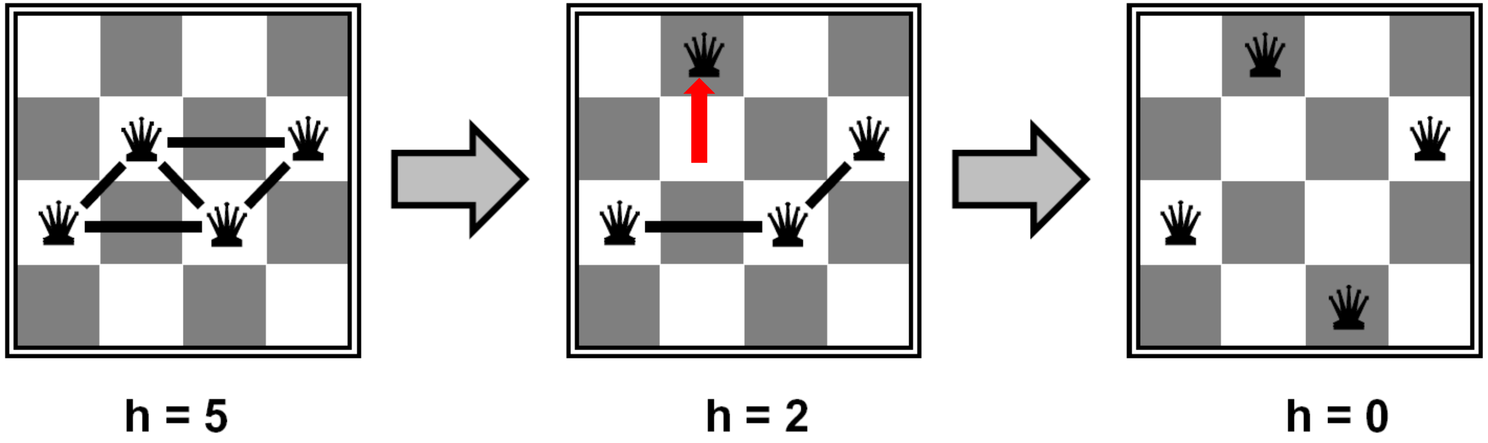
Example: 4-Queens



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

[Demo: n-queens – iterative improvement (L5D1)]
[Demo: coloring – iterative improvement]

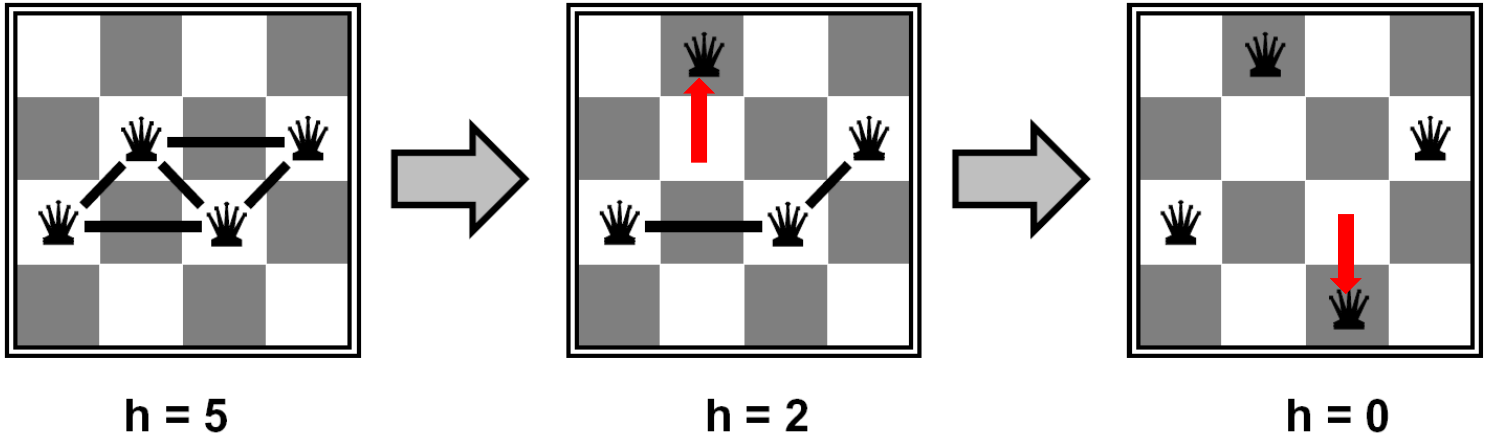
Example: 4-Queens



- 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

[Demo: n-queens – iterative improvement (L5D1)]
[Demo: coloring – iterative improvement]

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

[Demo: n-queens – iterative improvement (L5D1)]
[Demo: coloring – iterative improvement]

Basic Local Search Algorithm

Assign one domain value d_i to each variable v_i

while no solution & not stuck & not timed out:

  bestCost $\leftarrow \infty$;   bestList $\leftarrow$ [ ];

  for each variable v_i where Cost(Value(v_i)) > 0

    for each domain value d_i of v_i

      if Cost(d_i) < bestCost

        bestCost $\leftarrow$ Cost(d_i)

        bestList $\leftarrow$ [d_i]

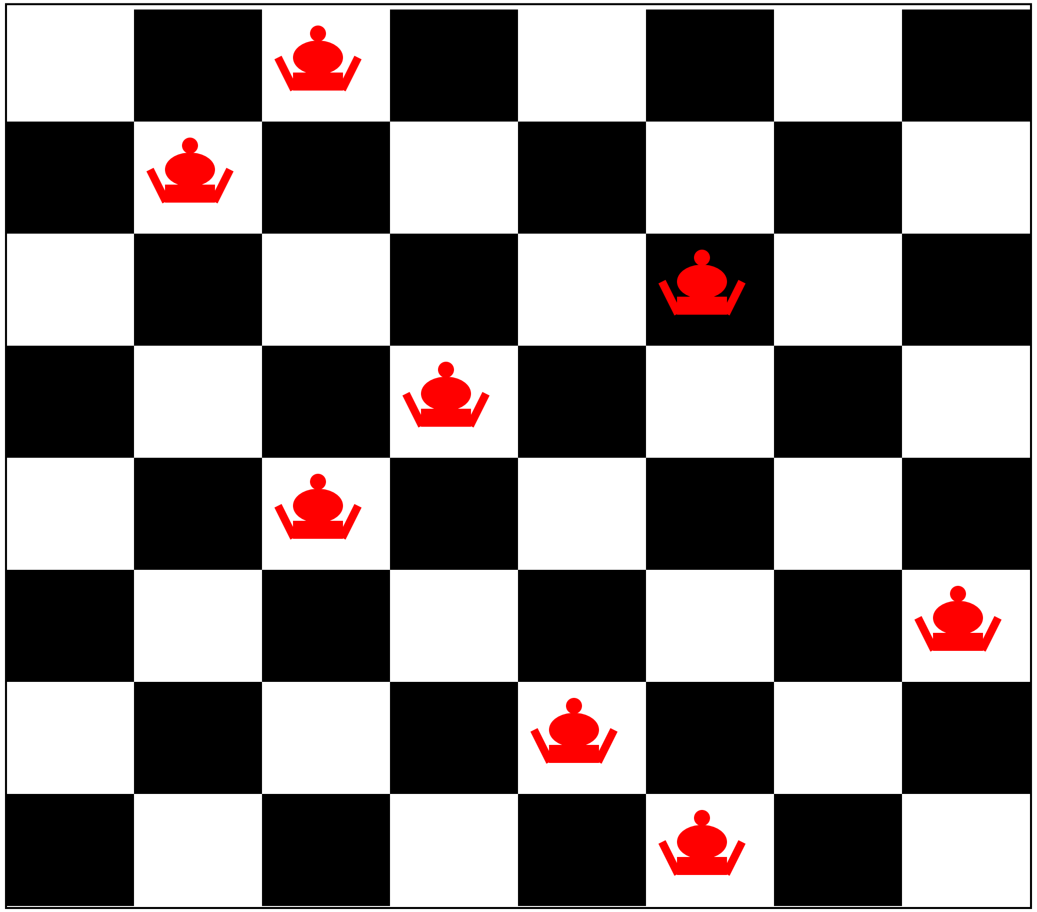
      else if Cost(d_i) = bestCost

        bestList $\leftarrow$ bestList $\cup$ d_i

  Take a randomly selected move from bestList

Eight Queens using Local Search

Place 8 Queens randomly on the board

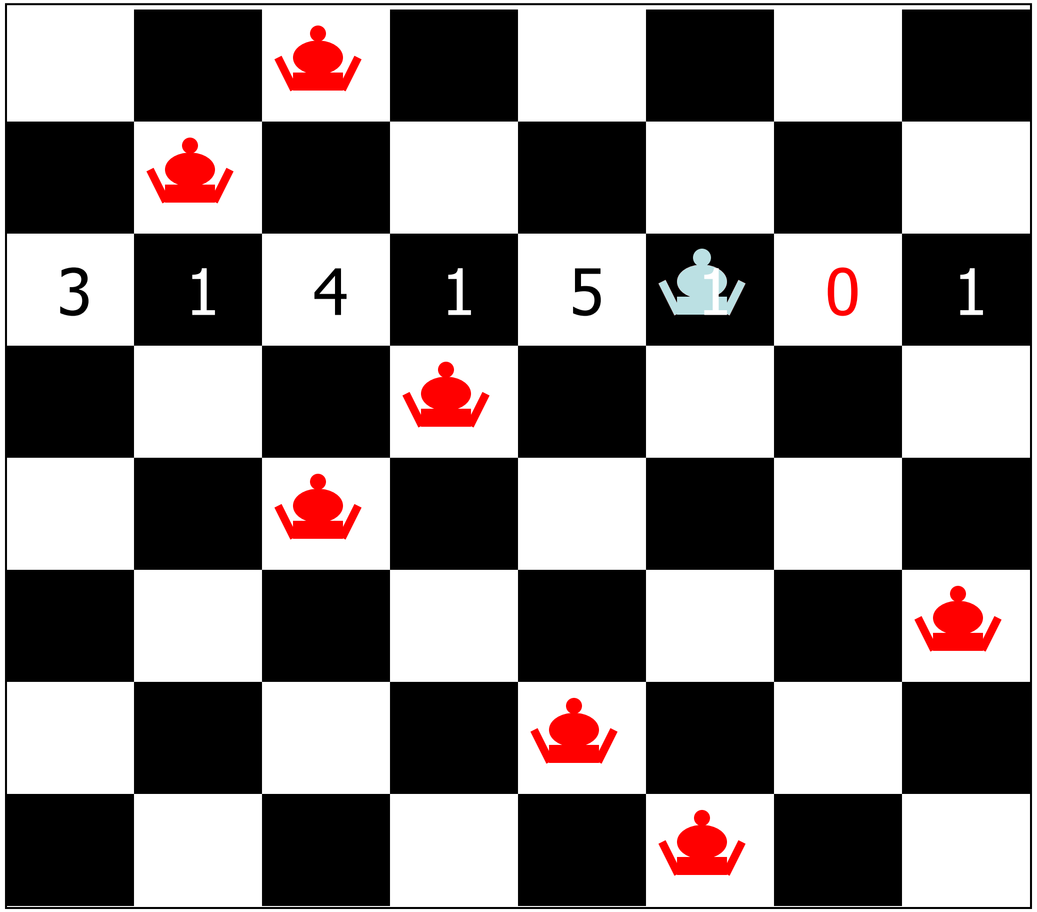


Eight Queens using Local Search

Pick a Queen:

Calculate cost

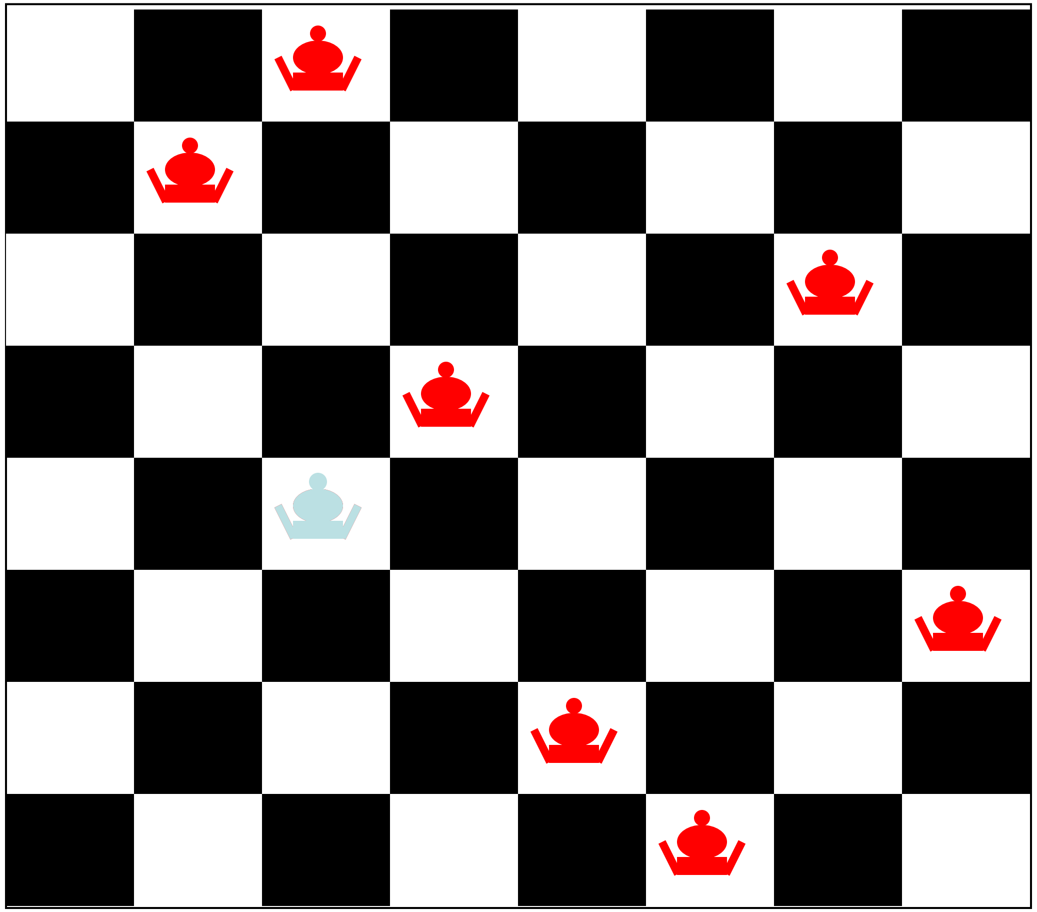
of each move



Slide

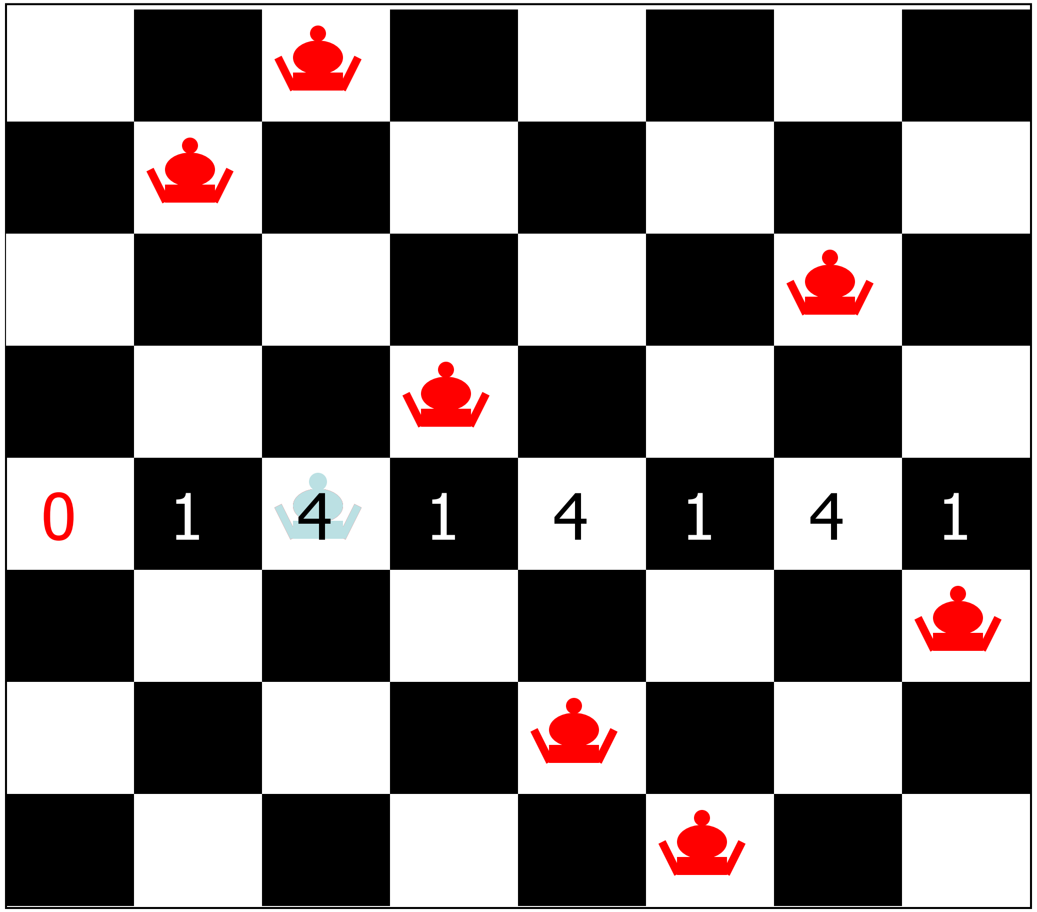
Eight Queens using Local Search

Take least cost move then try another Queen



Eight Queens using Local Search

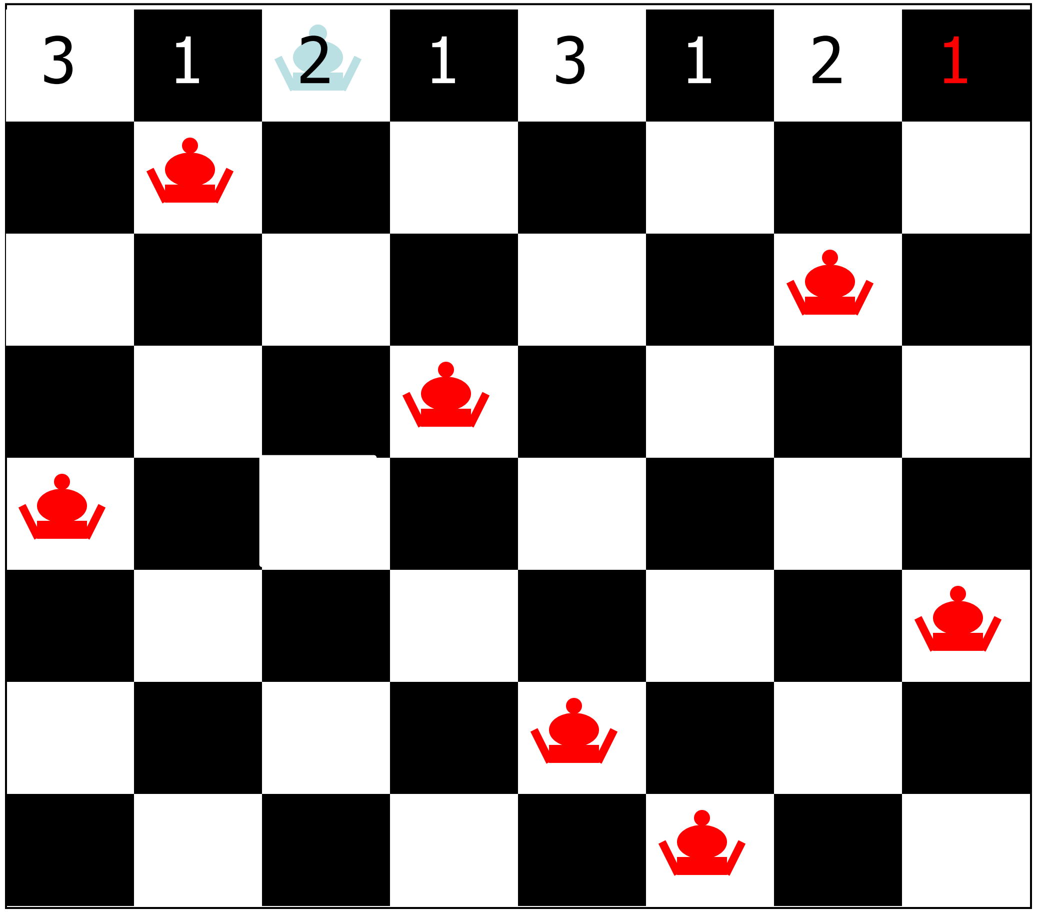
Take least cost move then try another Queen



Slide

Eight Queens using Local Search

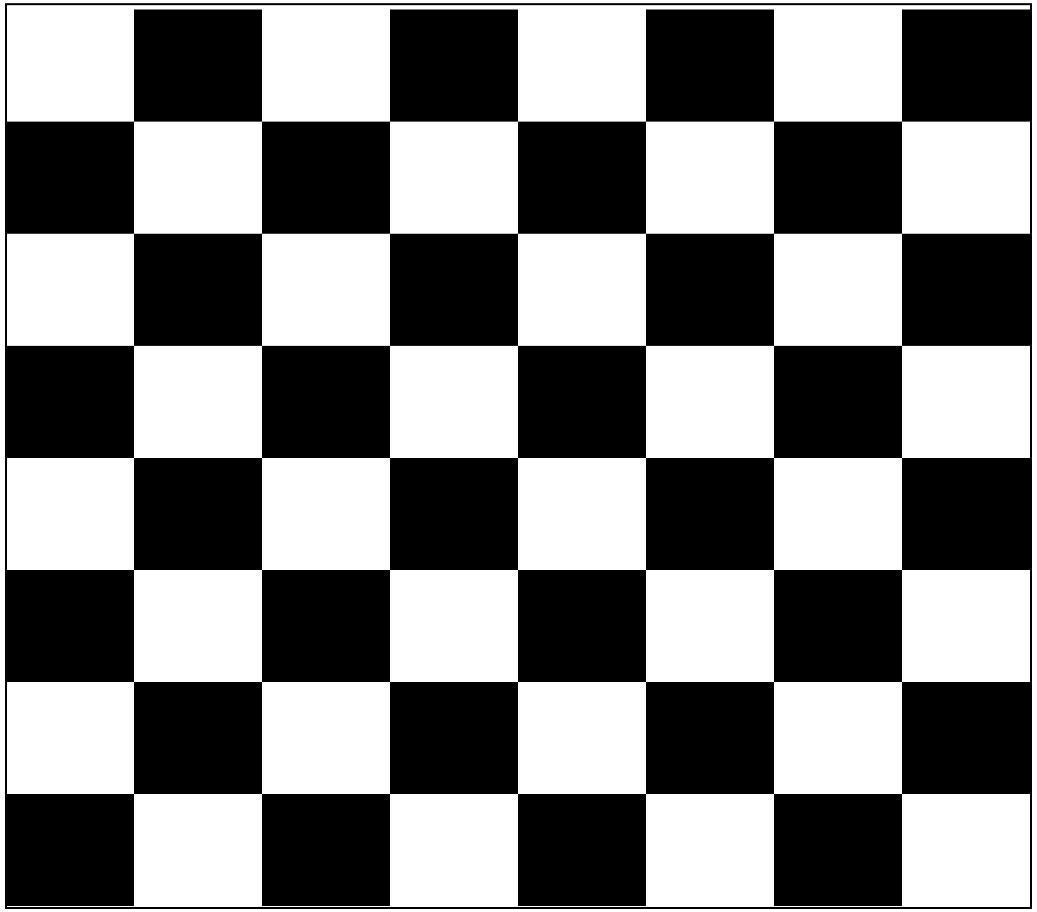
Take least cost move then try another Queen



…and so on, until….

Slide

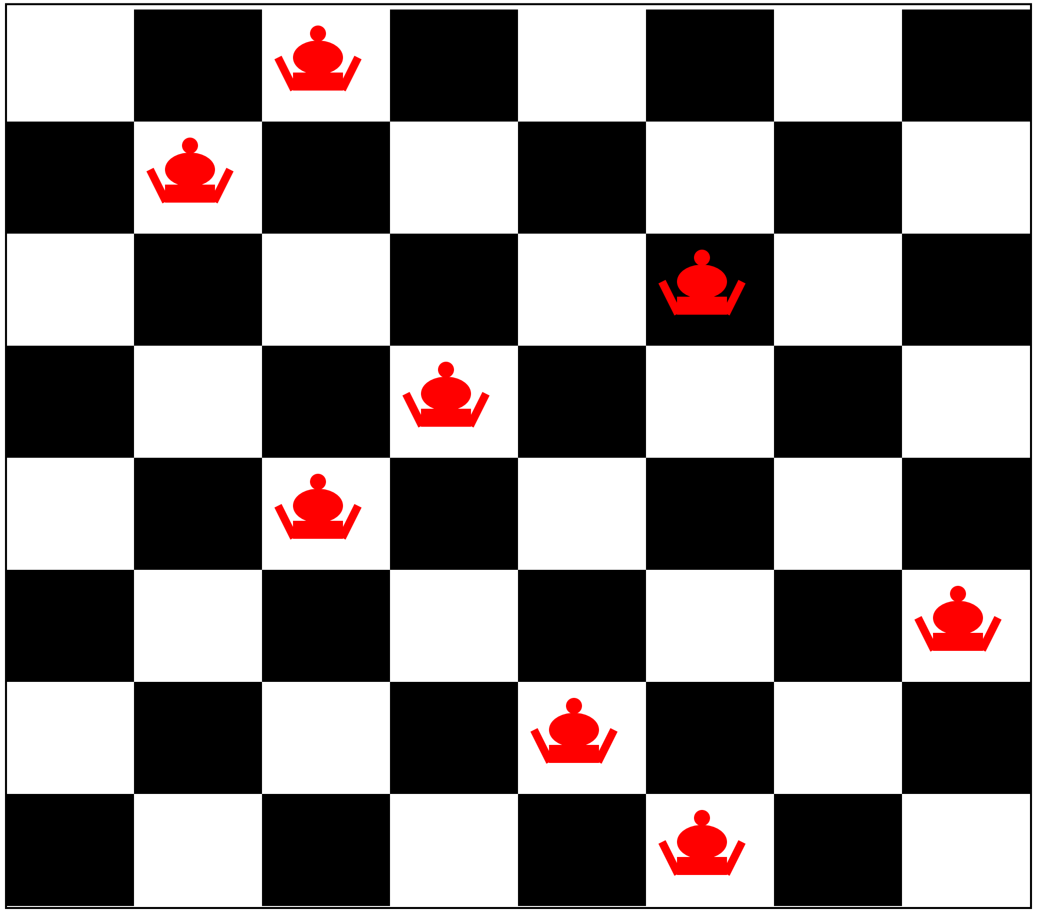
Eight Queens using Local Search



Slide

Eight Queens using Local Search

Place 8 Queens randomly on the board



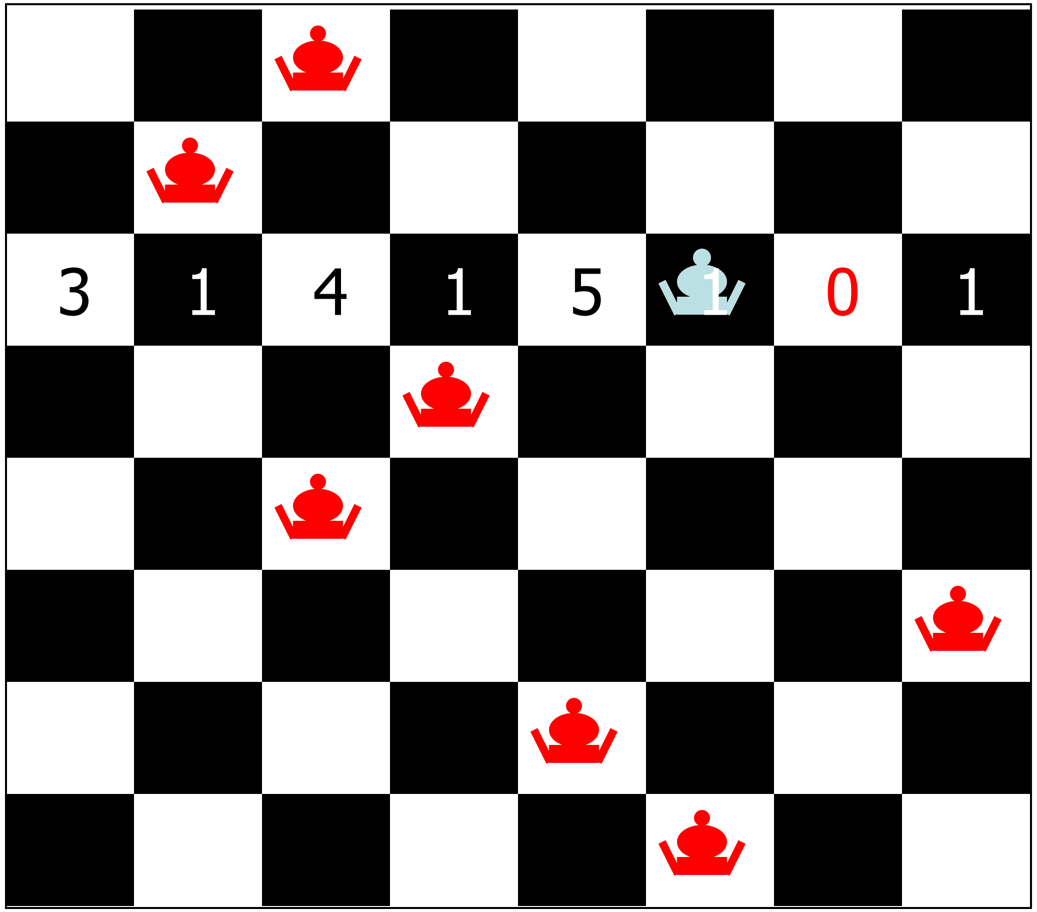
Slide

Eight Queens using Local Search

Pick a Queen:

Calculate cost

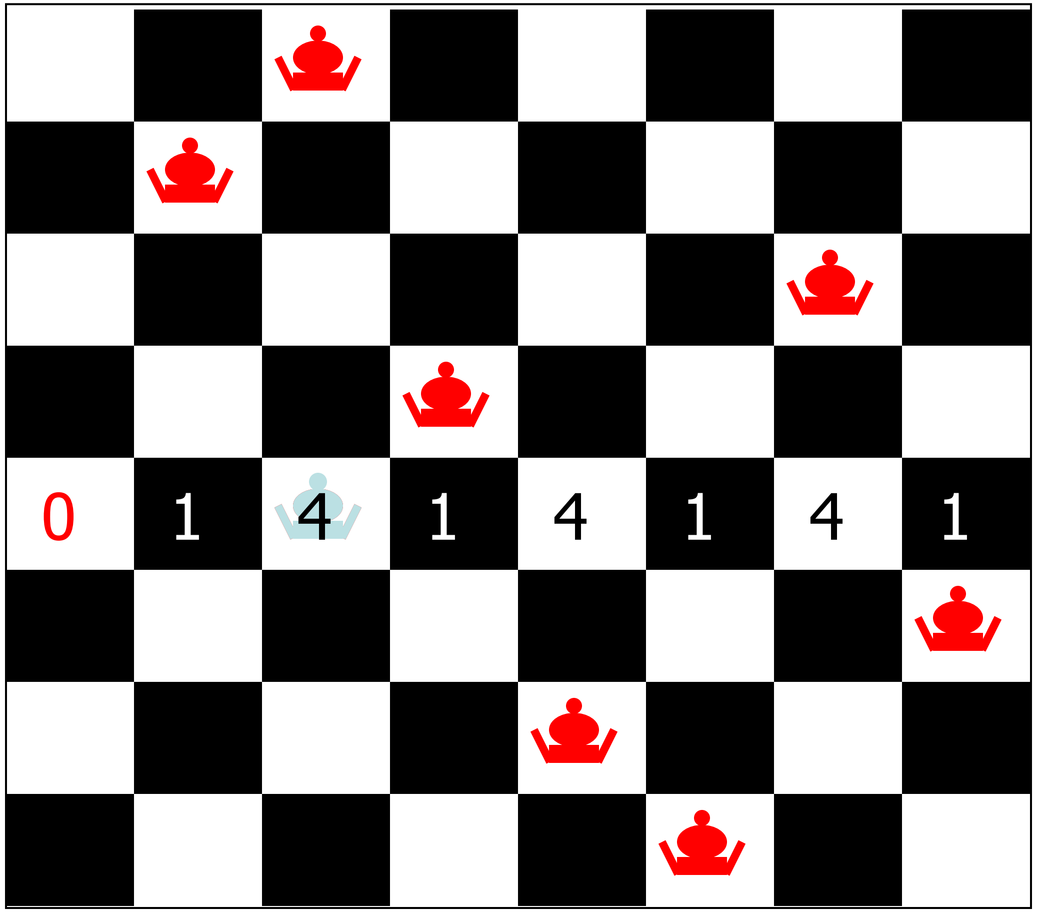
of each move



Slide

Eight Queens using Local Search

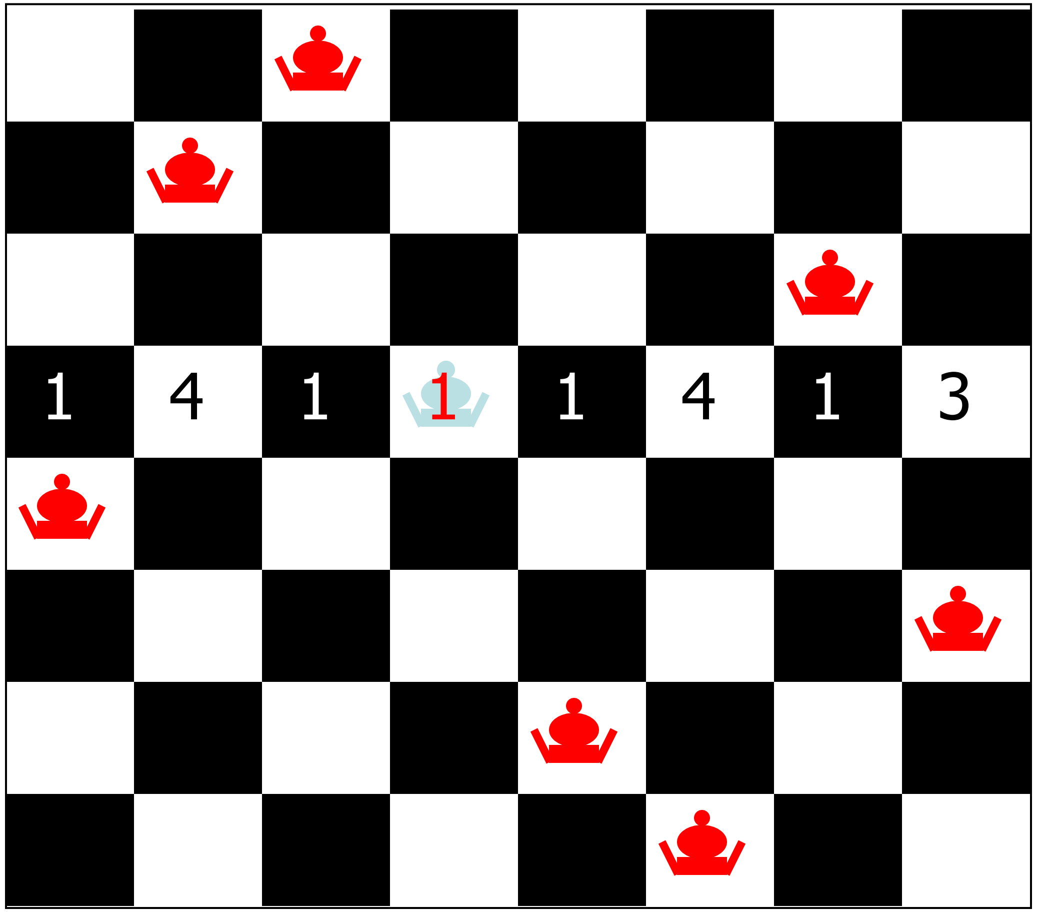
Take least cost move then try another Queen



Slide

Eight Queens using Local Search

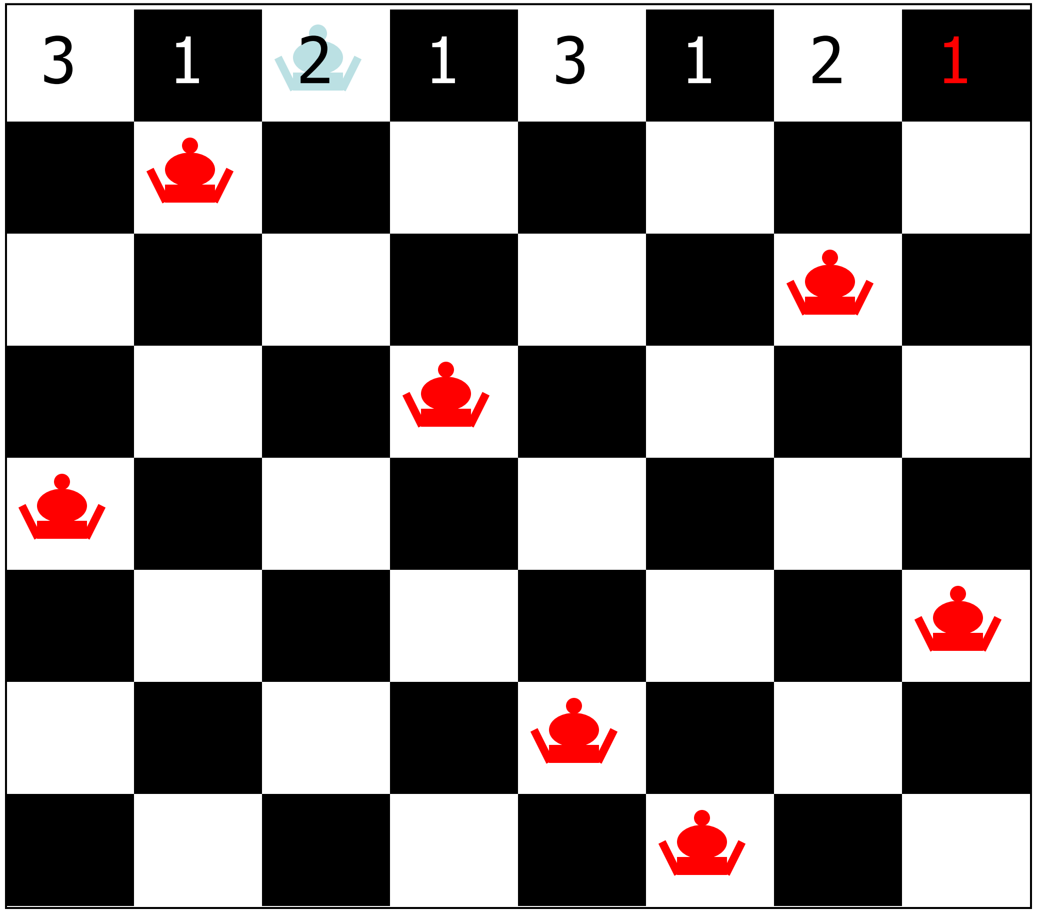
Take least cost move then try another Queen



Slide

Eight Queens using Local Search

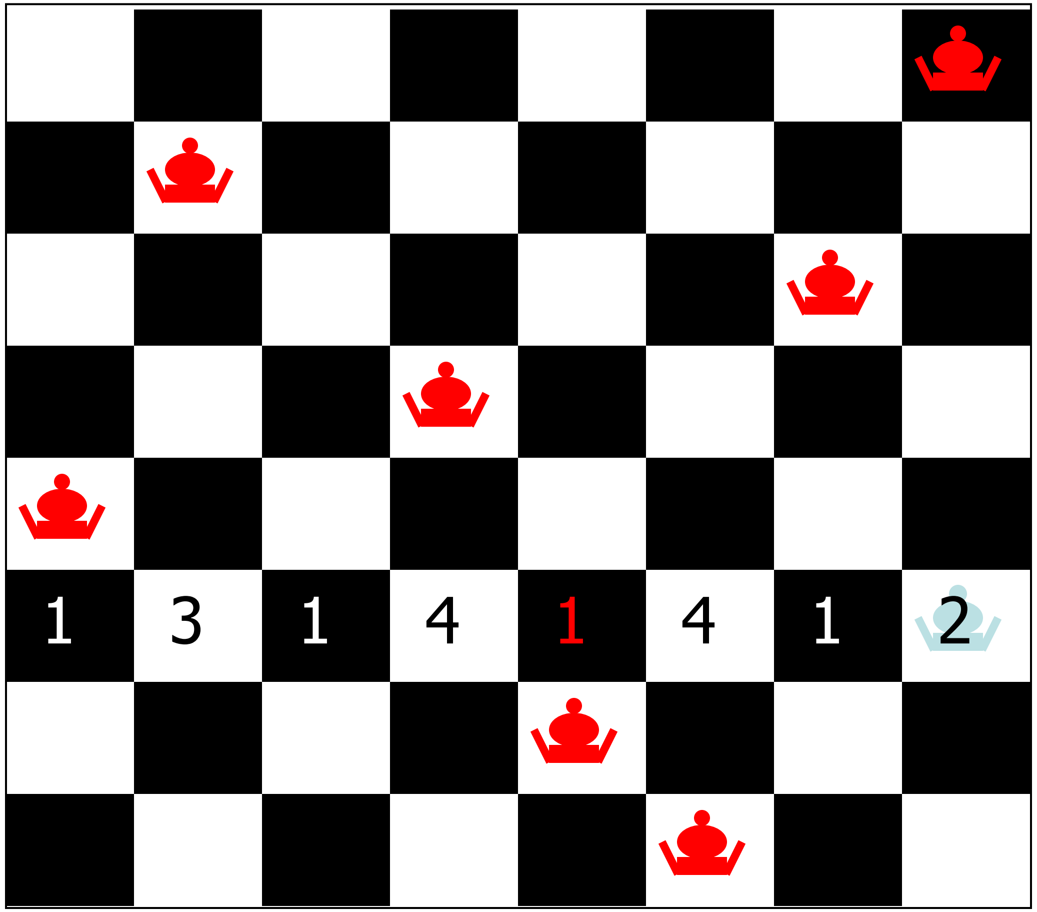
Take least cost move then try another Queen



Slide

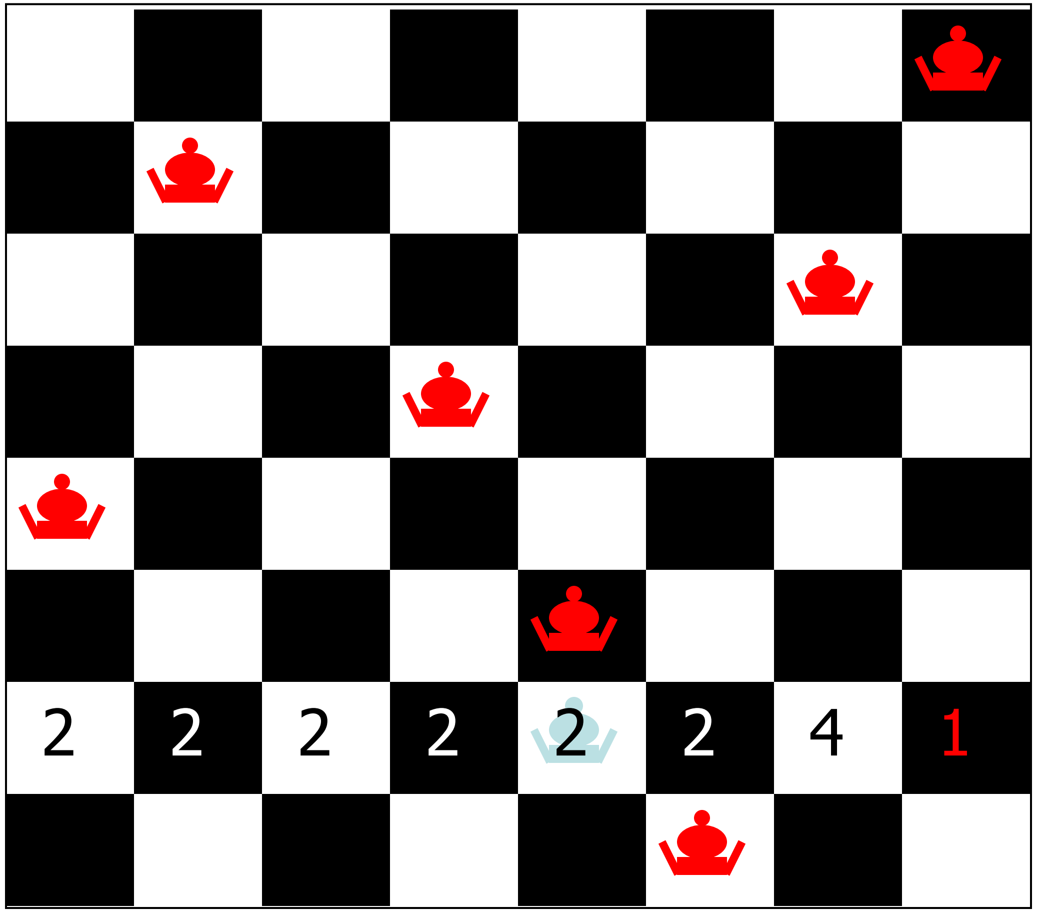
Eight Queens using Local Search

Take least cost move then try another Queen



Eight Queens using Local Search

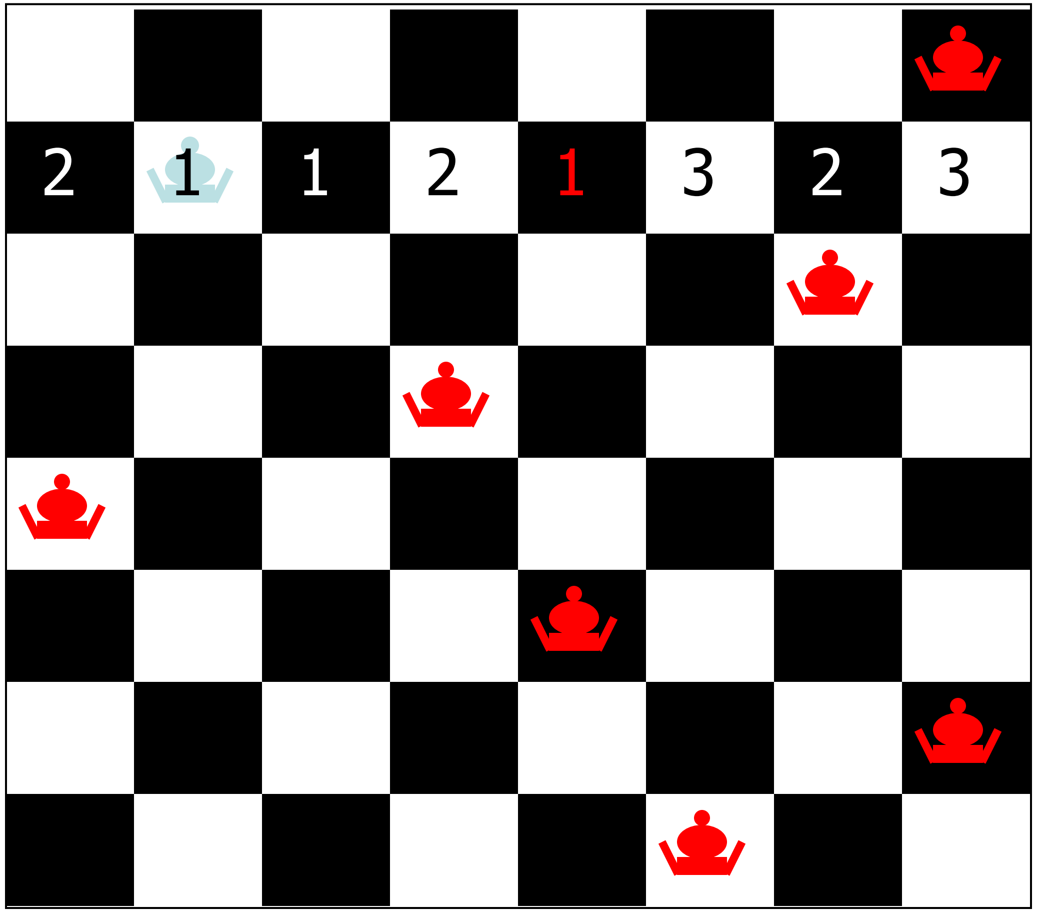
Take least cost move then try another Queen



Slide

Eight Queens using Local Search

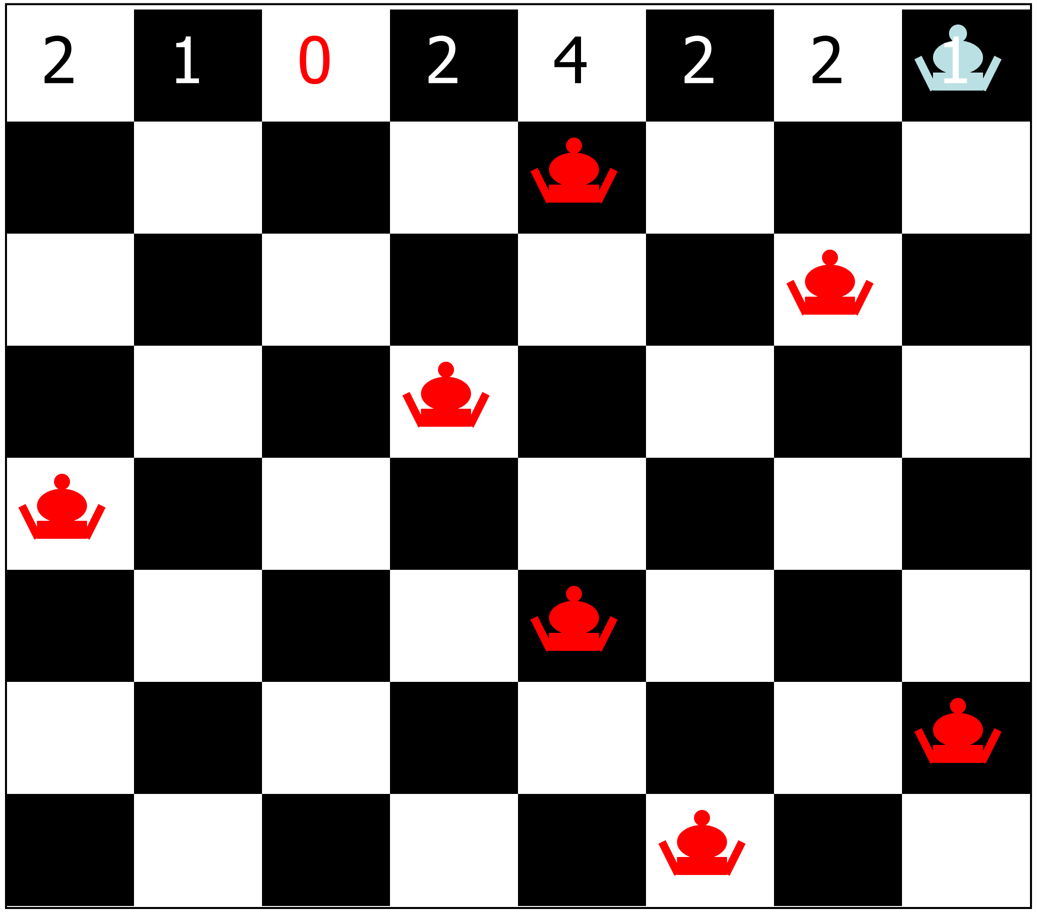
Take least cost move then try another Queen



Slide

Eight Queens using Local Search

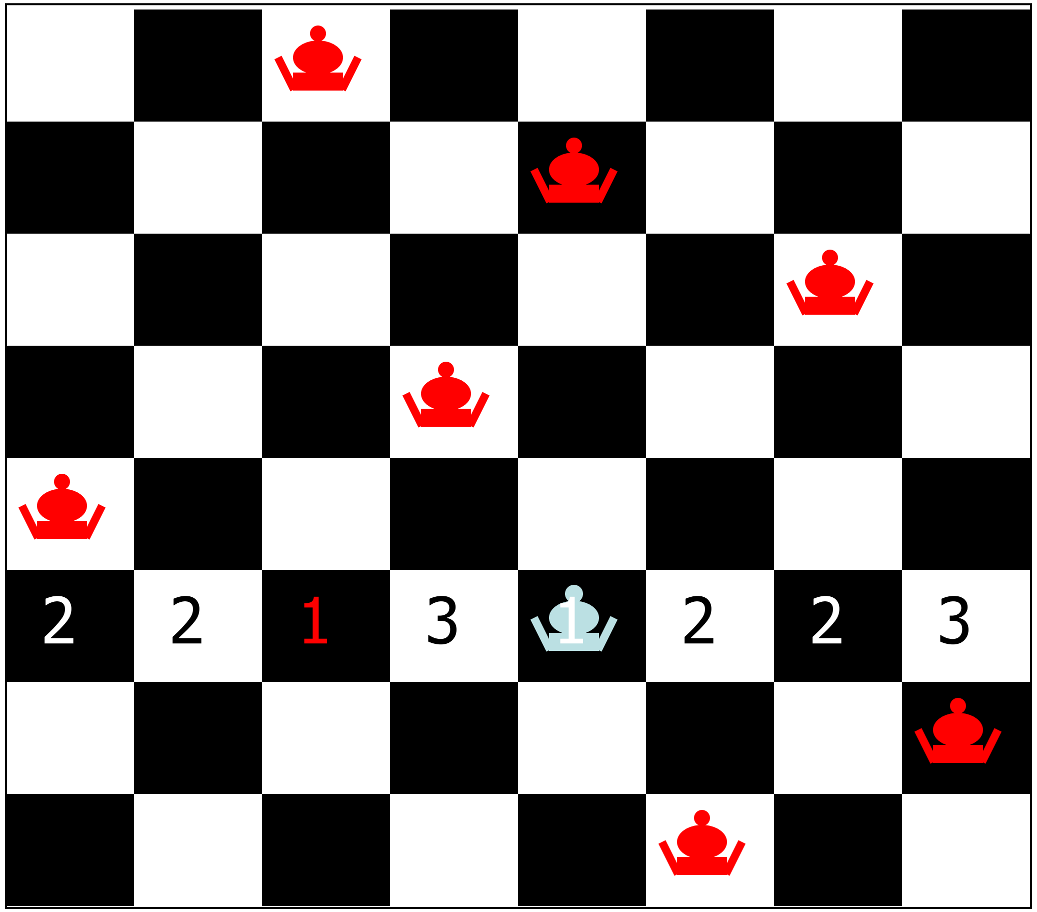
Take least cost move then try another Queen



Slide

Eight Queens using Local Search

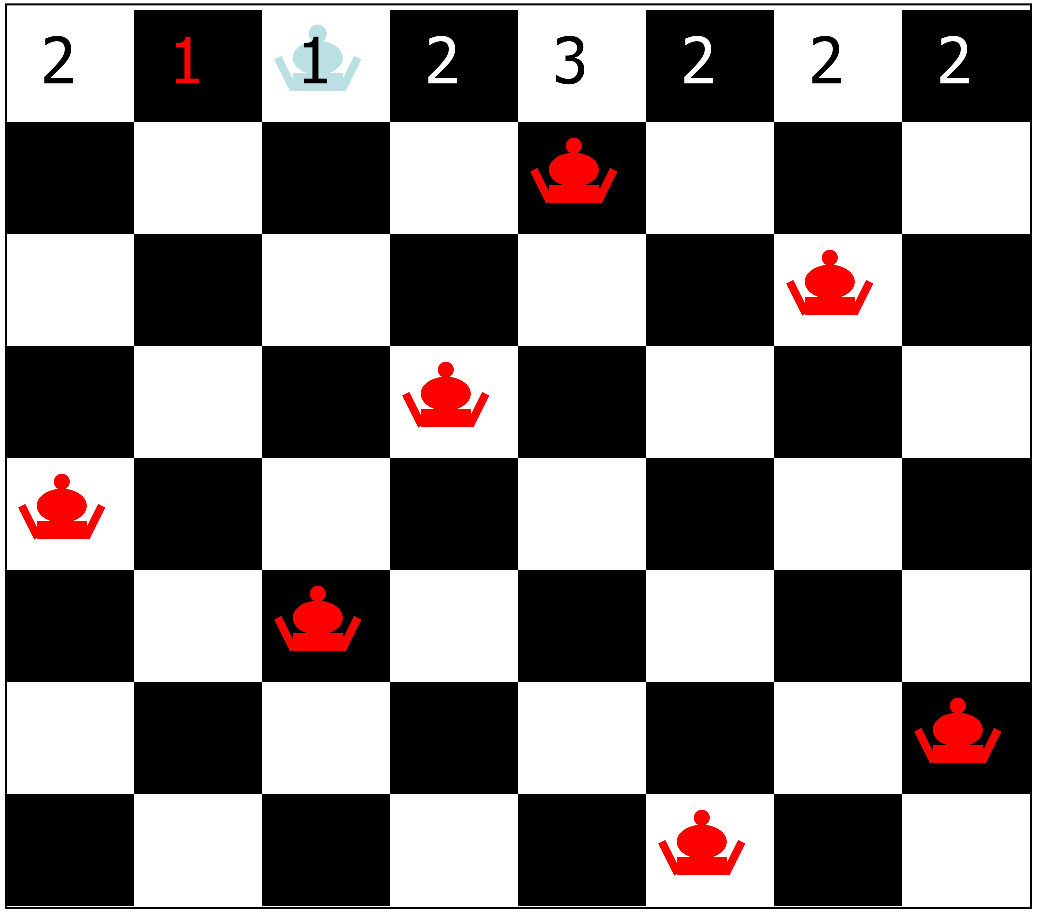
Take least cost move then try another Queen



Slide

Eight Queens using Local Search

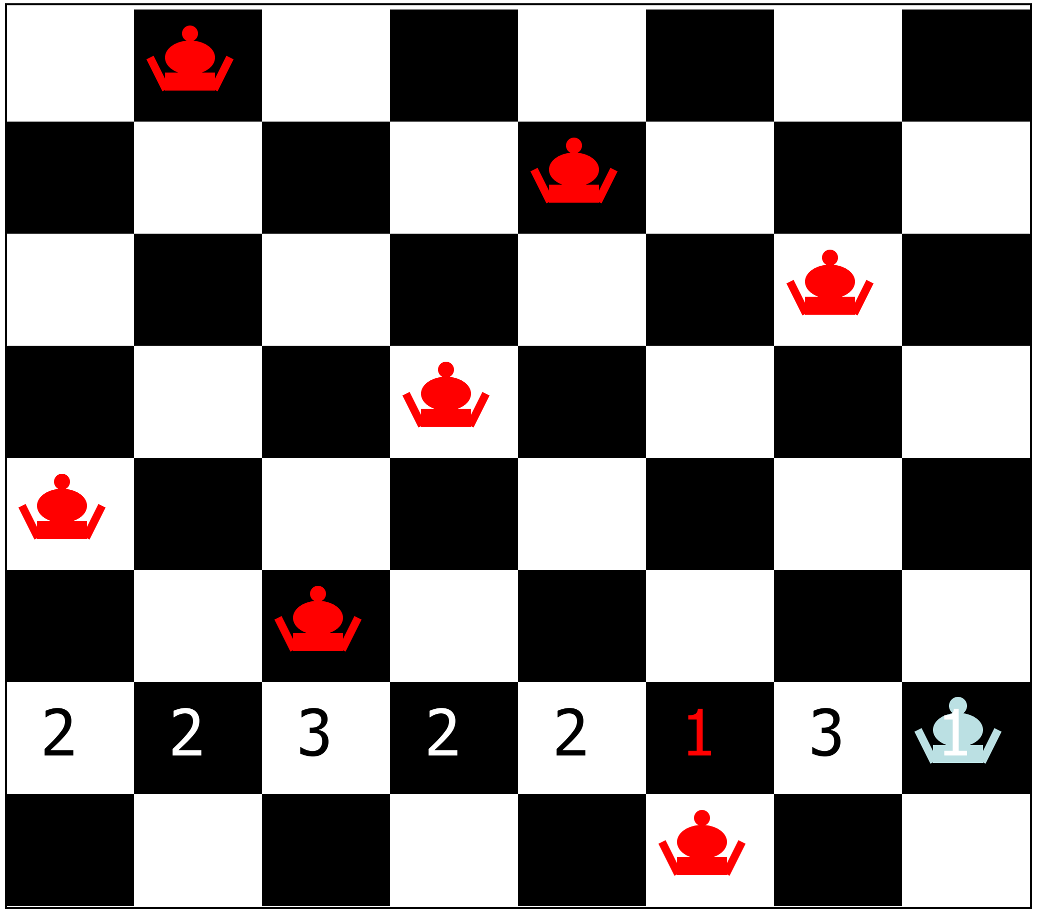
Take least cost move then try another Queen



Slide

Eight Queens using Local Search

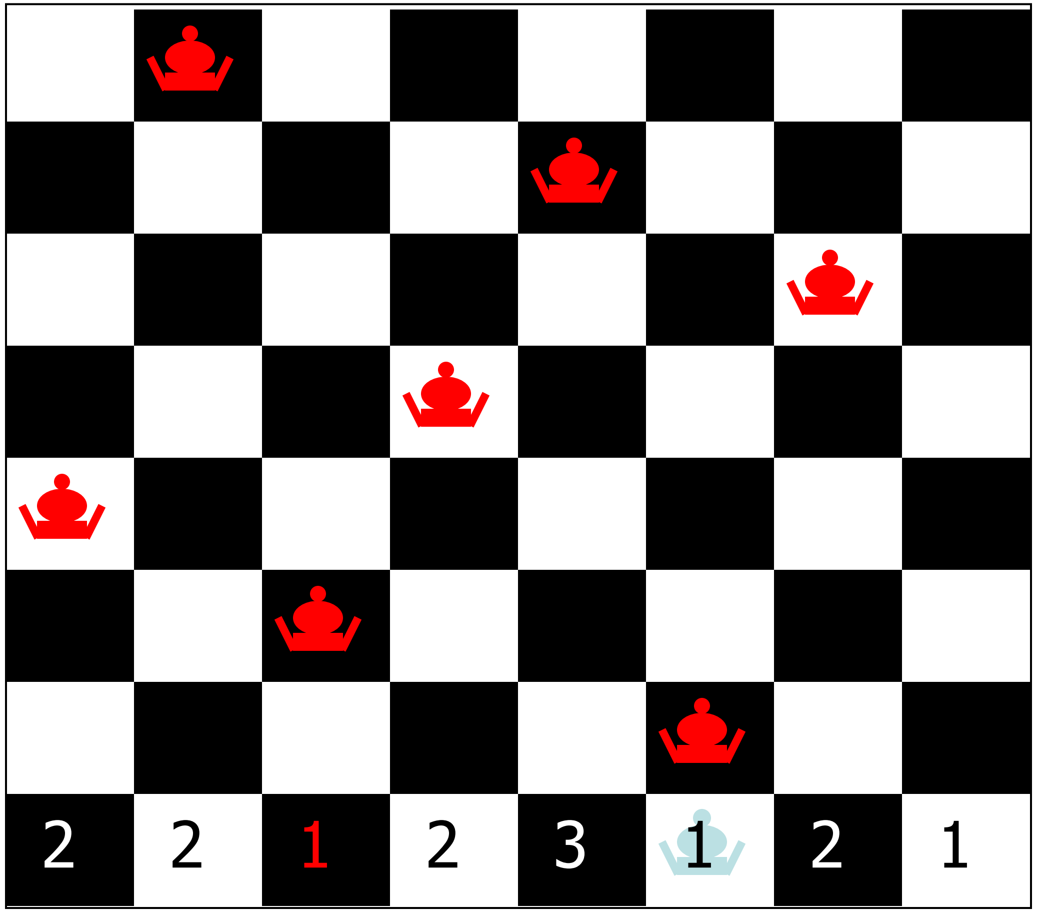
Take least cost move then try another Queen



Slide

Eight Queens using Local Search

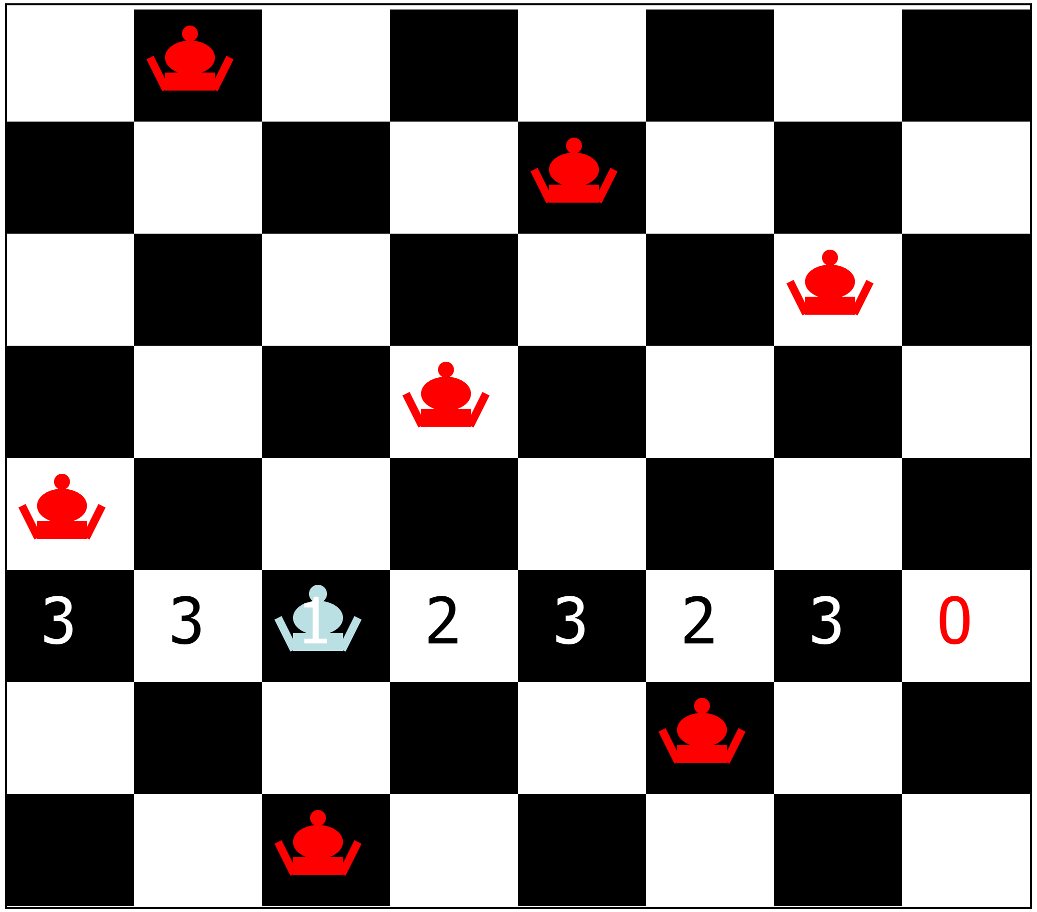
Take least cost move then try another Queen



Slide

Eight Queens using Local Search

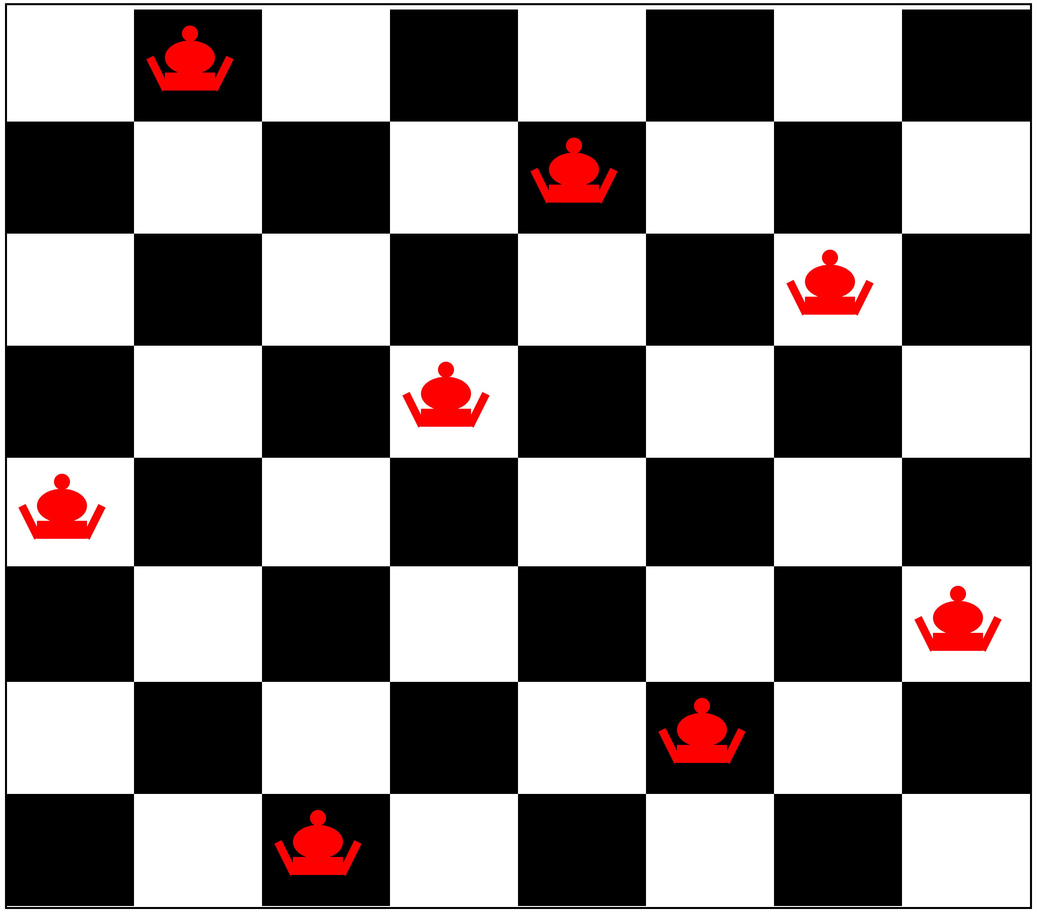
Take least cost move then try another Queen



Slide

Eight Queens using Local Search

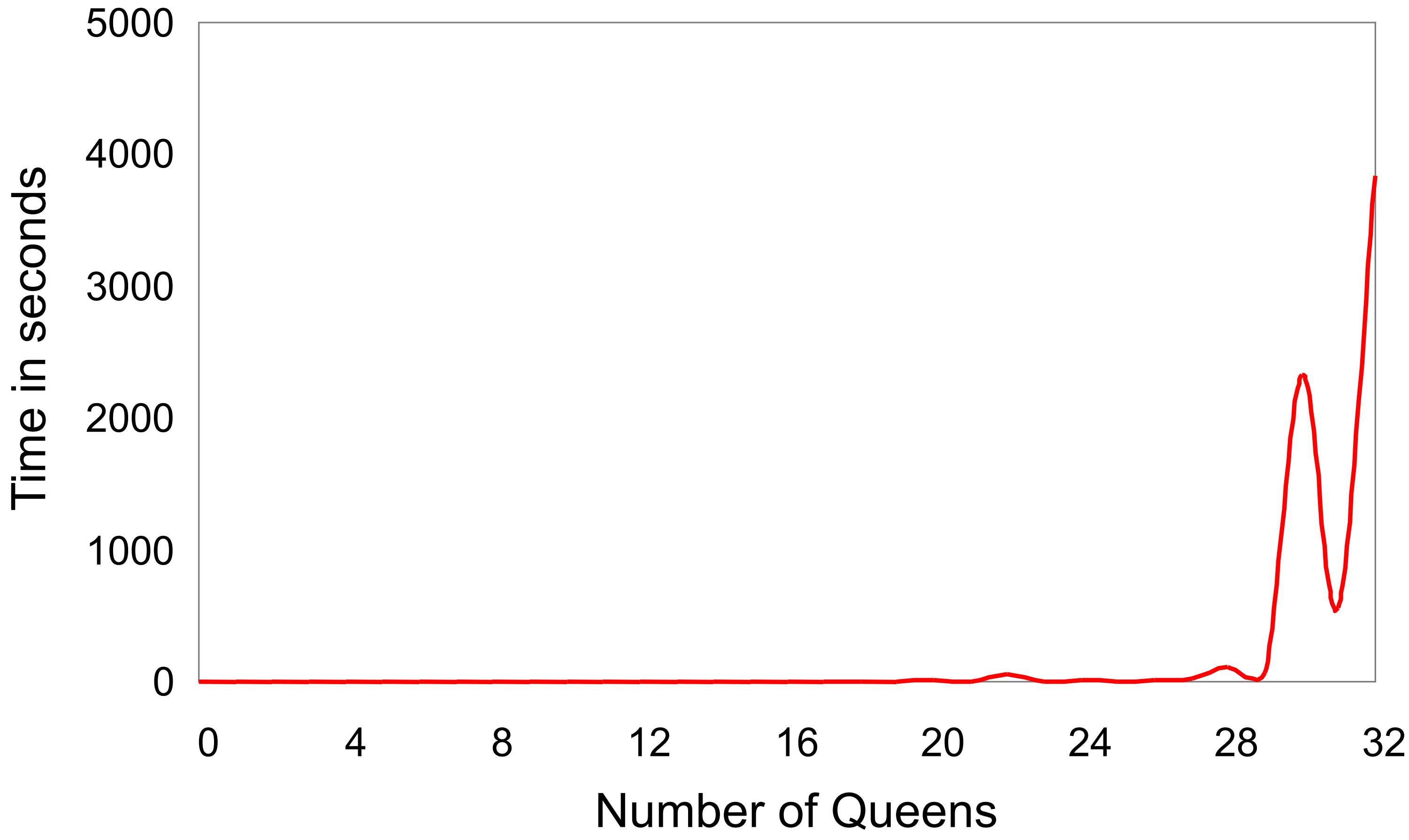
Answer Found



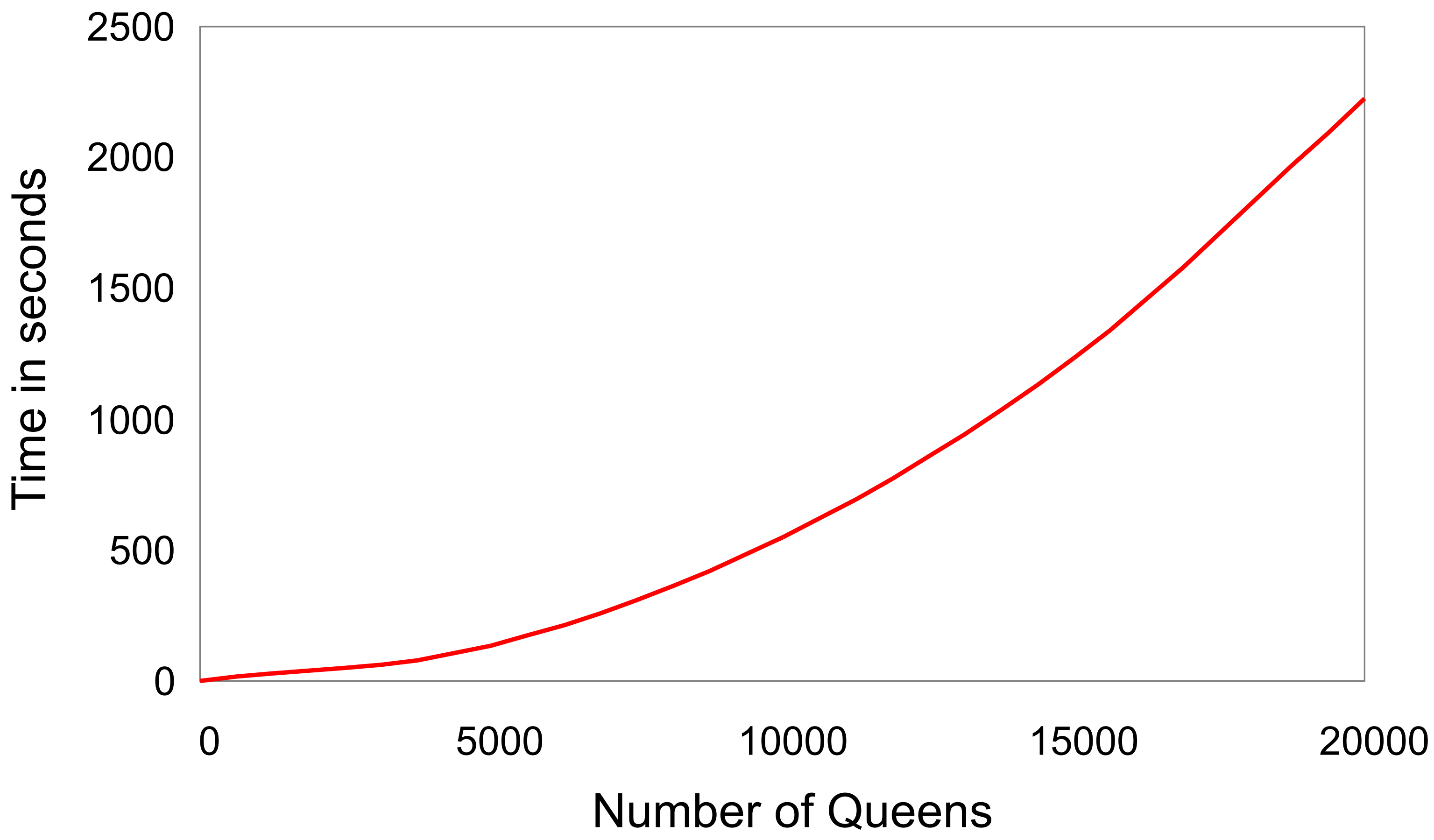
Slide

Video of Demo Iterative Improvement – Coloring

Backtracking Performance



Local Search Performance



Performance of Min-Conflicts

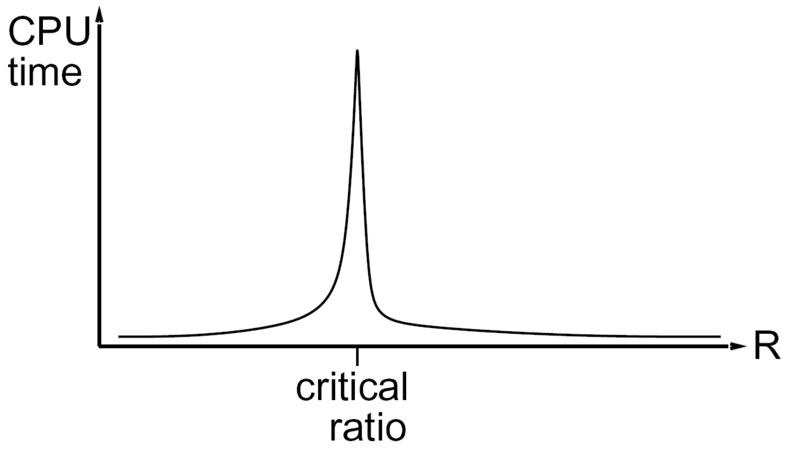
- Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., n = 10,000,000)

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

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

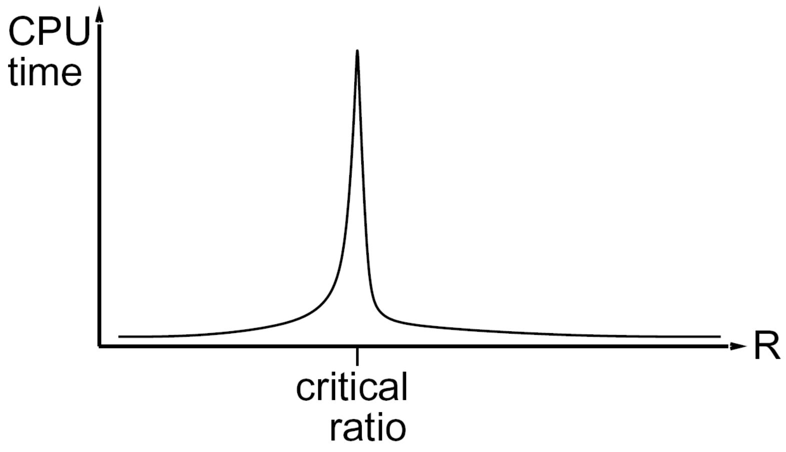


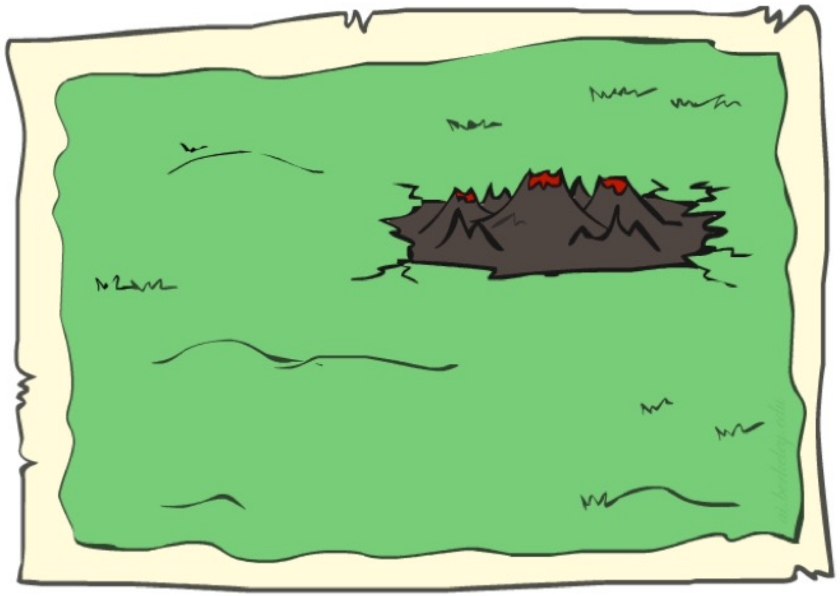
Performance of Min-Conflicts

- 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

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



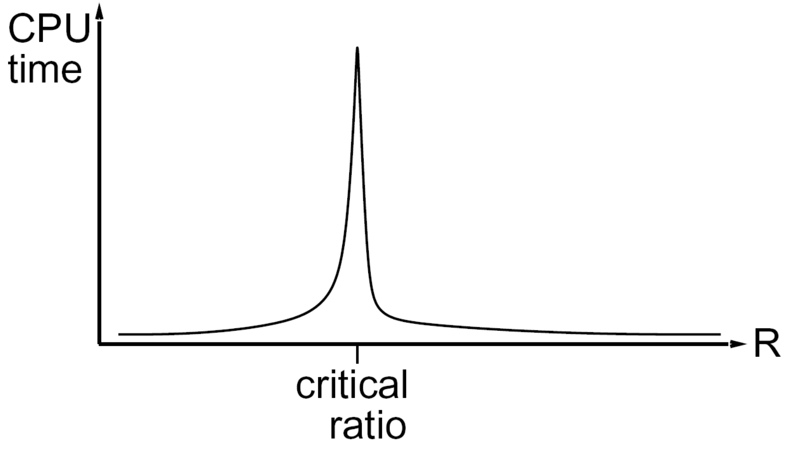


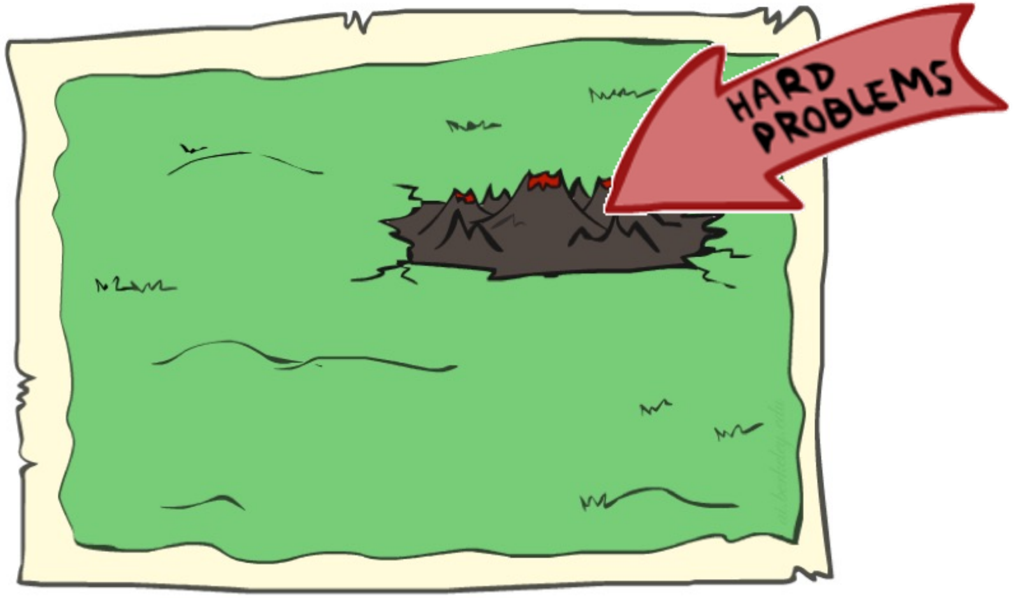
Performance of Min-Conflicts

- 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

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



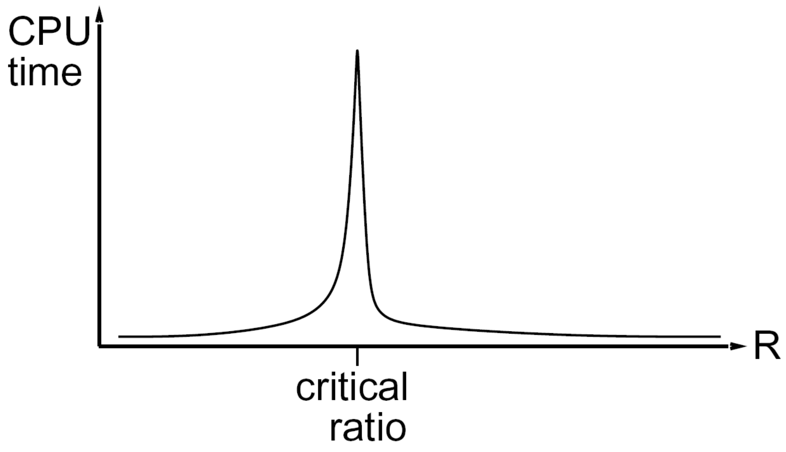


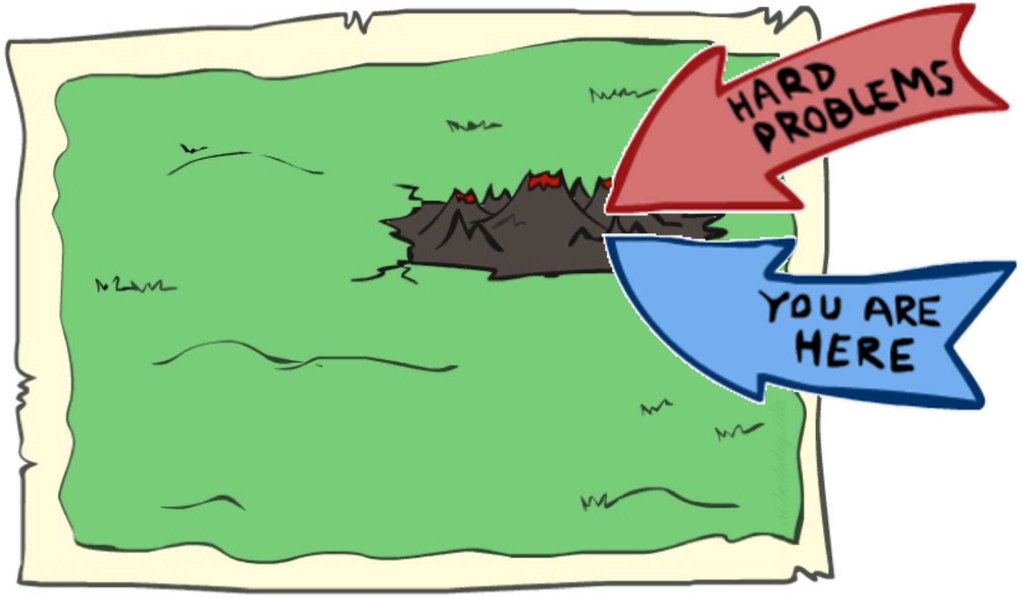
Performance of Min-Conflicts

- 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

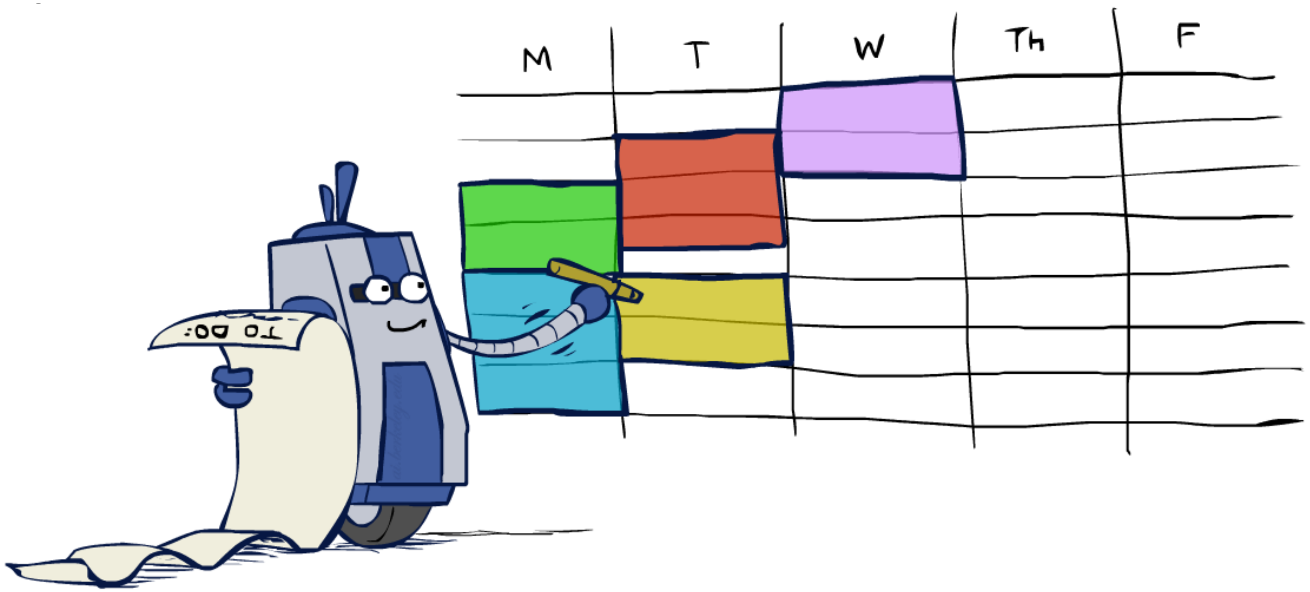
$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$





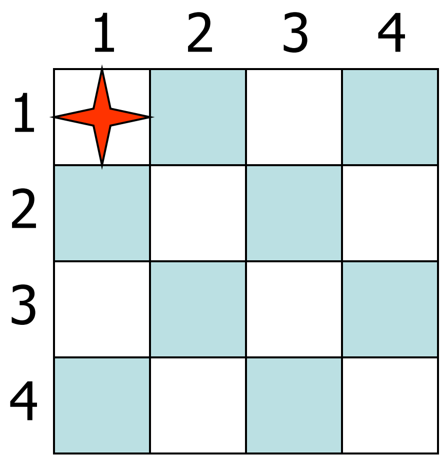
Summary: CSPs

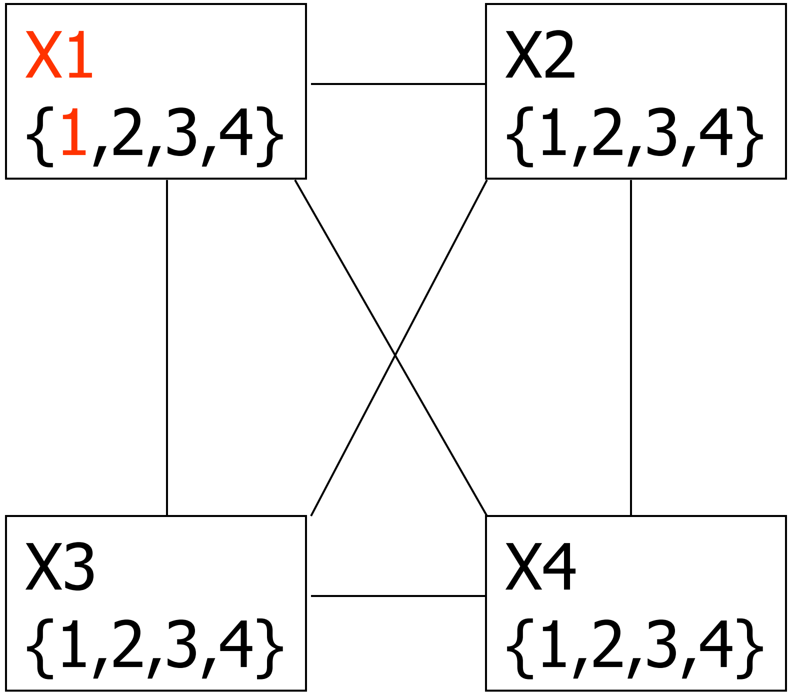
- CSPs are a special kind of search problem:
  - States are partial assignments
  - Goal test defined by constrai
- Basic solution: backtracking sea
- Speed-ups:
  - Ordering
  - Filtering
  - Structure
- Iterative min-conflicts is often effective in practice



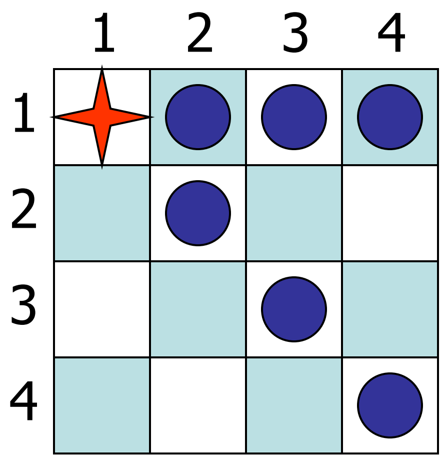
More Examples

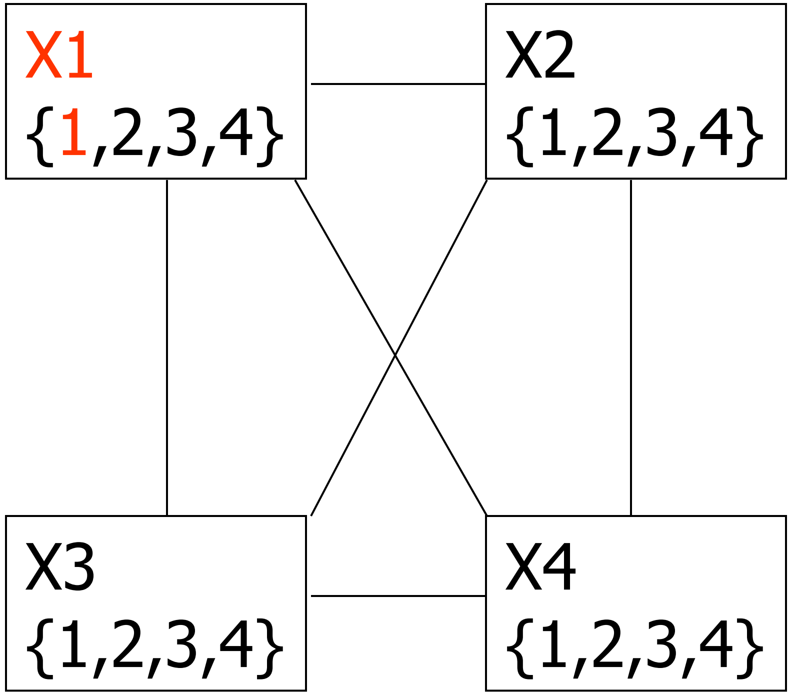
4-Queens Problem



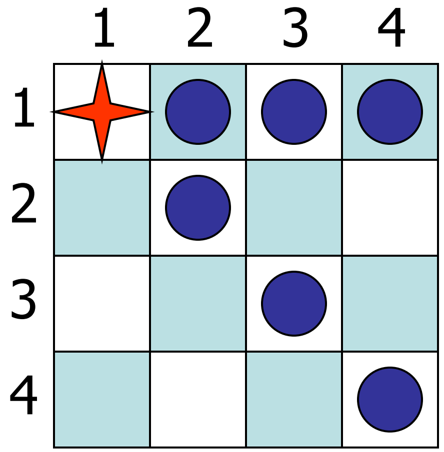


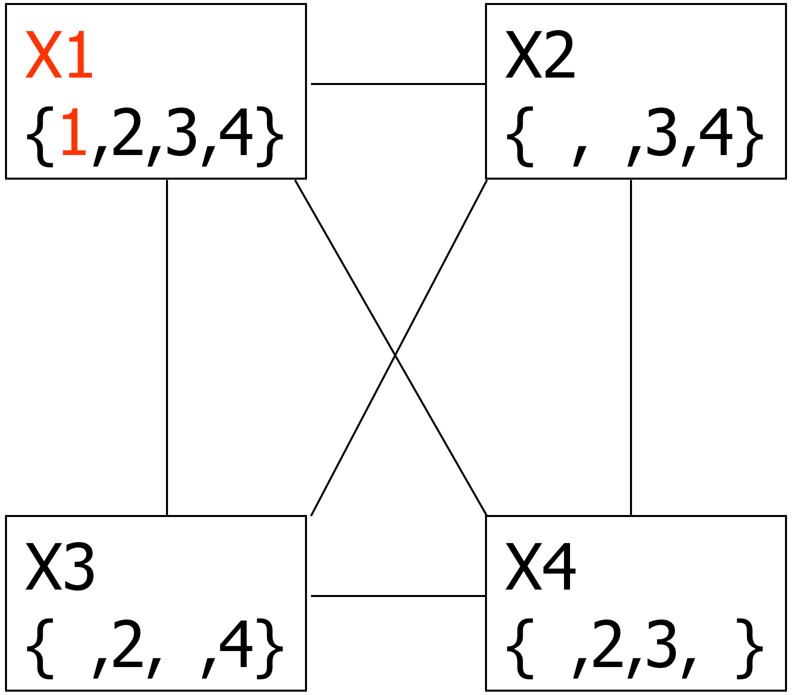
4-Queens Problem



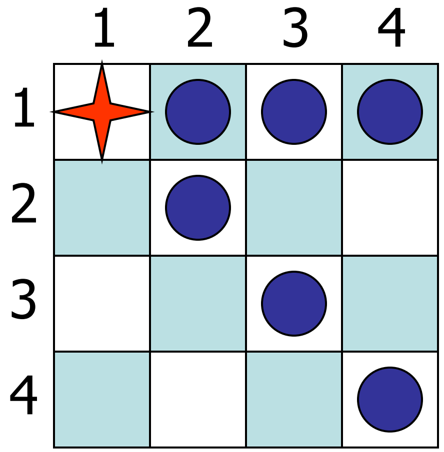


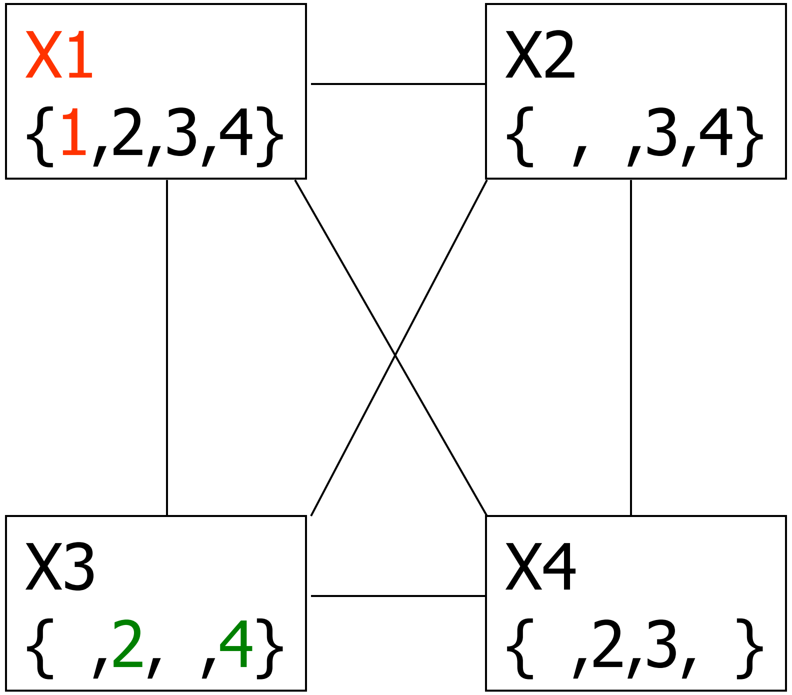
4-Queens Problem



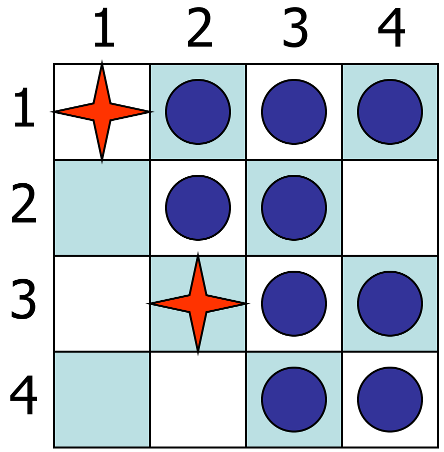


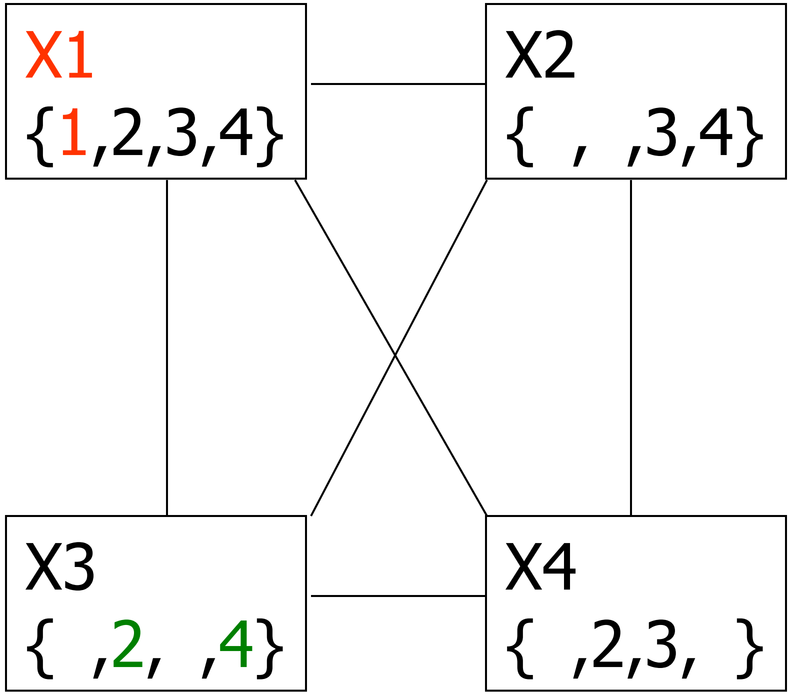
4-Queens Problem



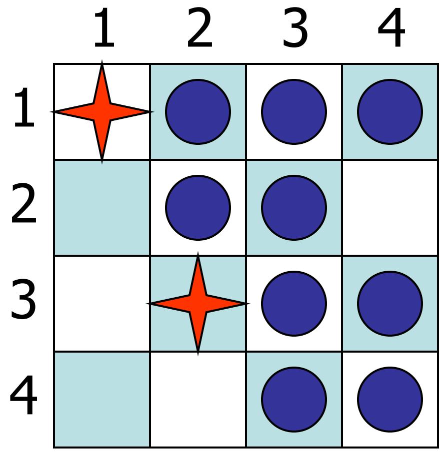


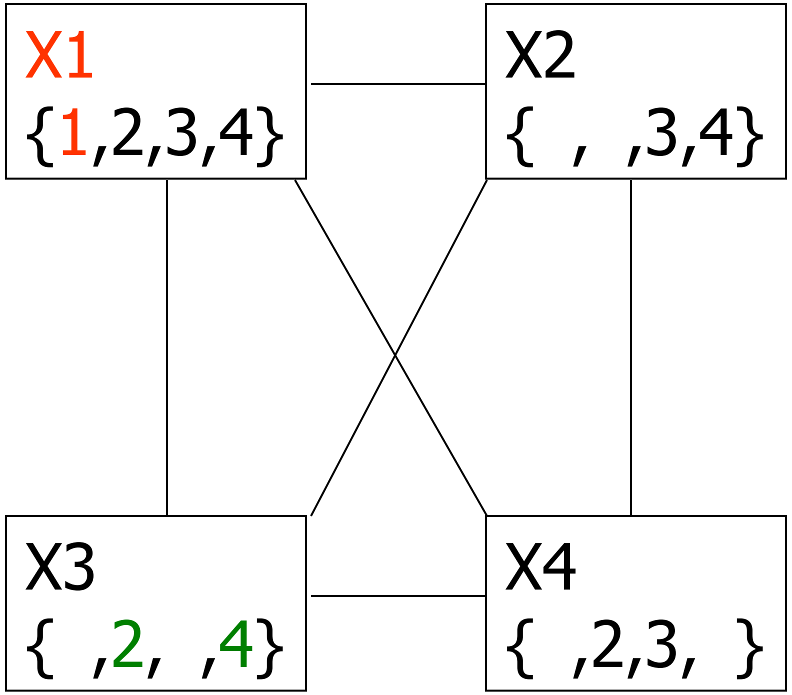
4-Queens Problem





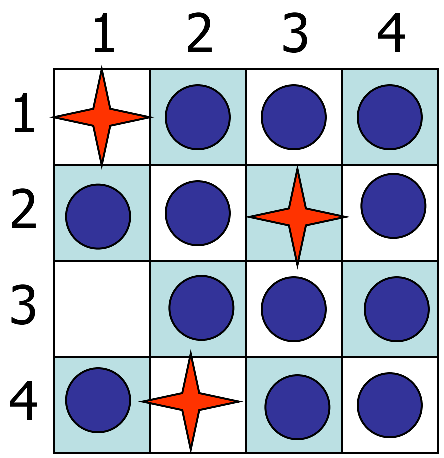
4-Queens Problem

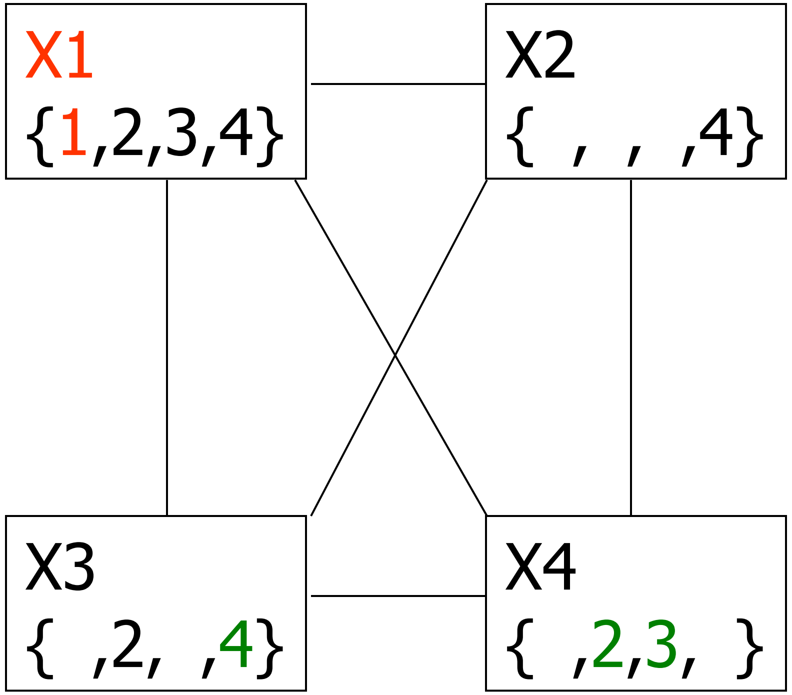




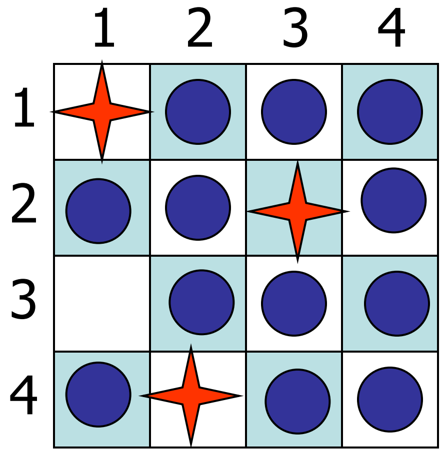
X2=3 eliminates { X3=2, X3=3, X3=4 }
⇒ inconsistent!

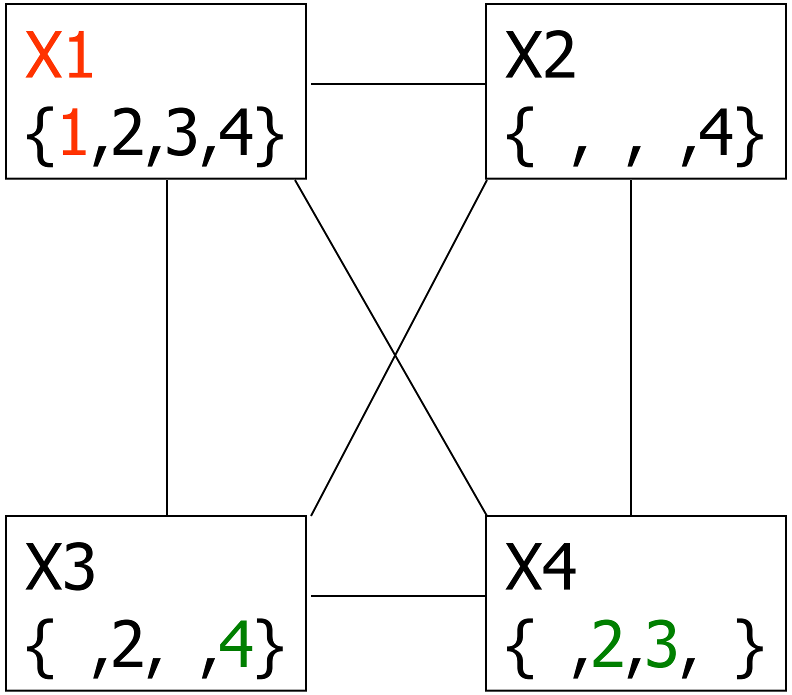
4-Queens Problem





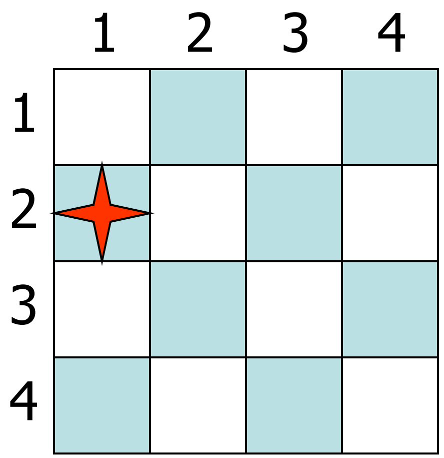
4-Queens Problem

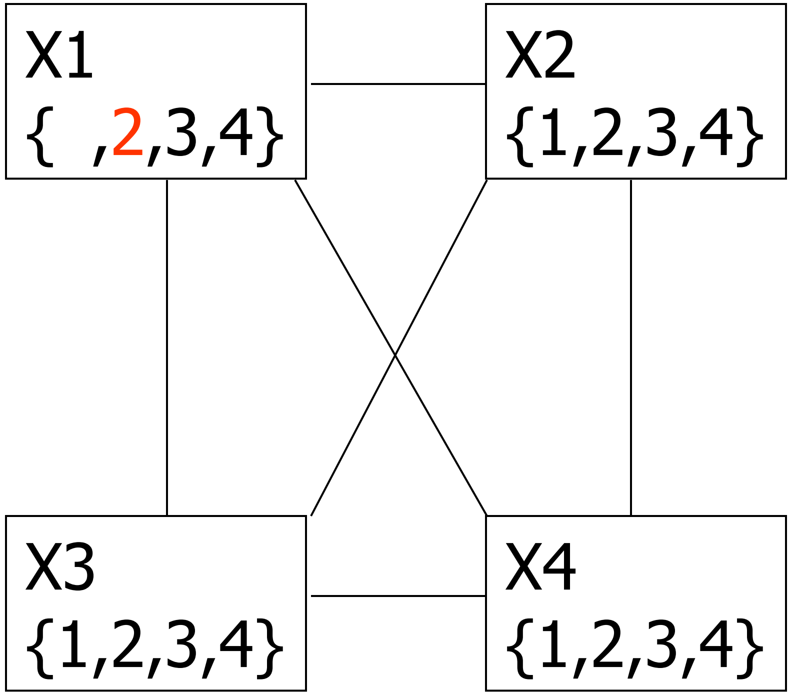




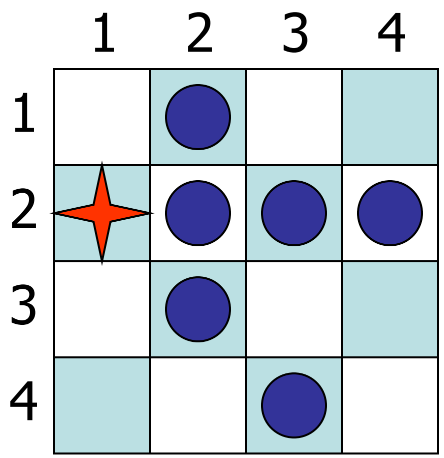
X2=4 ⇒ X3=2, which eliminates { X4=2, X4=3}
⇒ inconsistent!

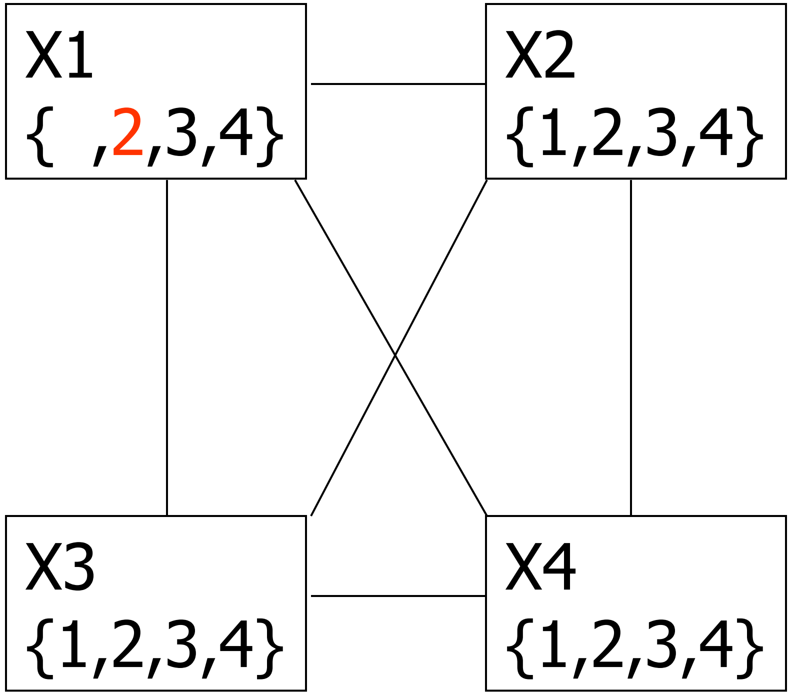
4-Queens Problem



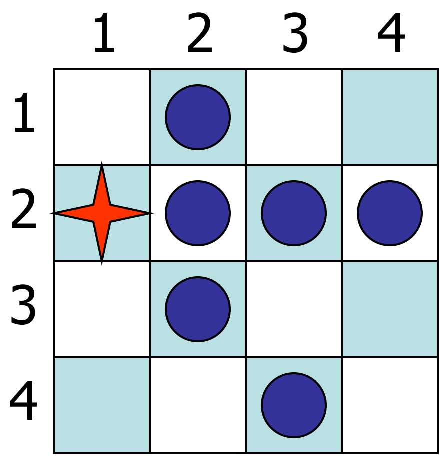


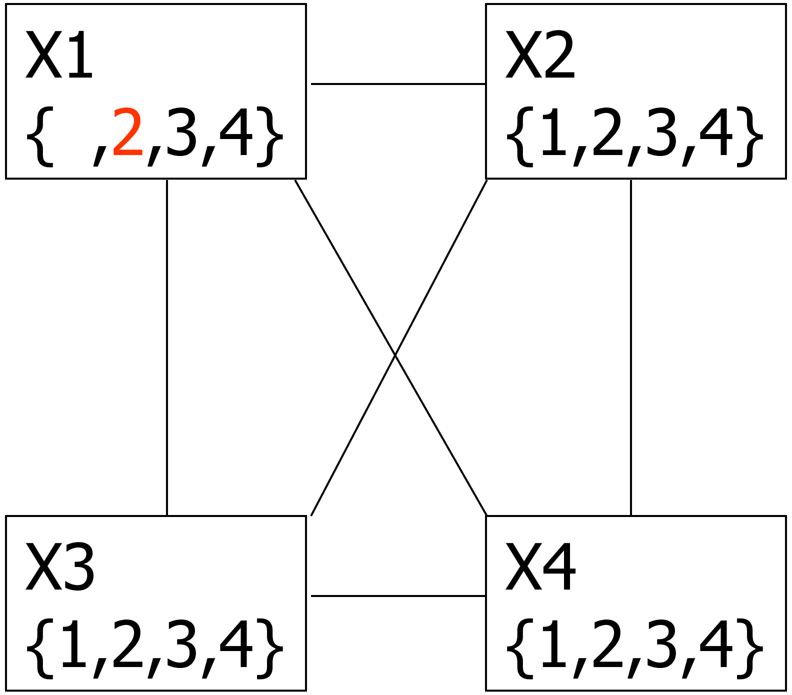
4-Queens Problem





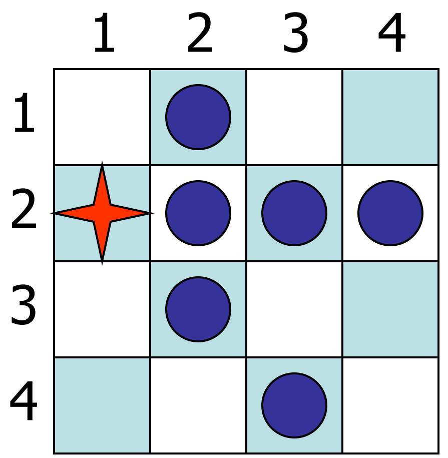
4-Queens Problem

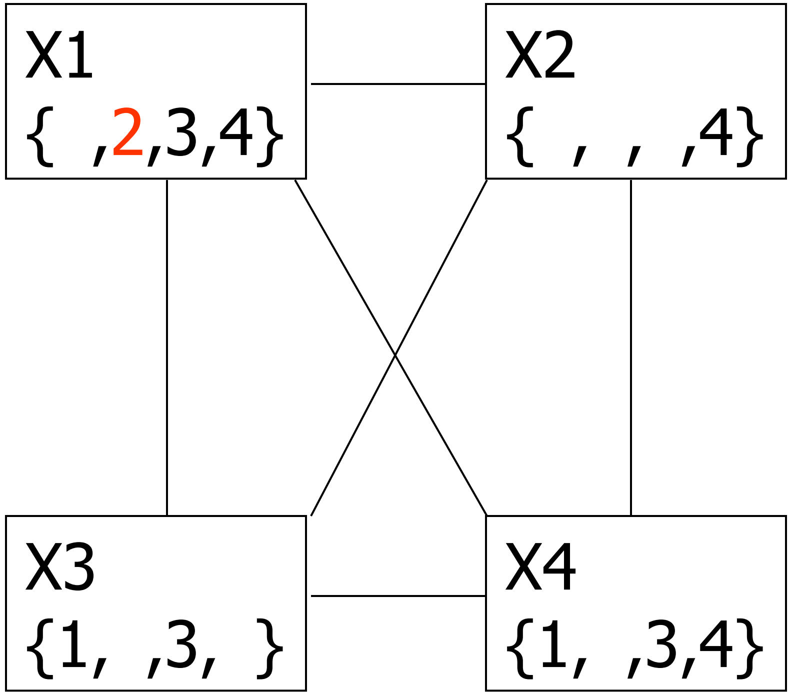




X1 can't be 1, let's try 2

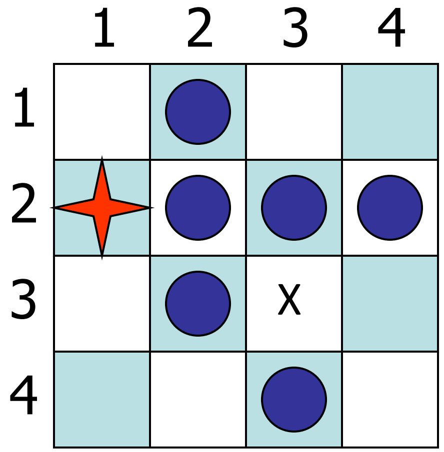
4-Queens Problem

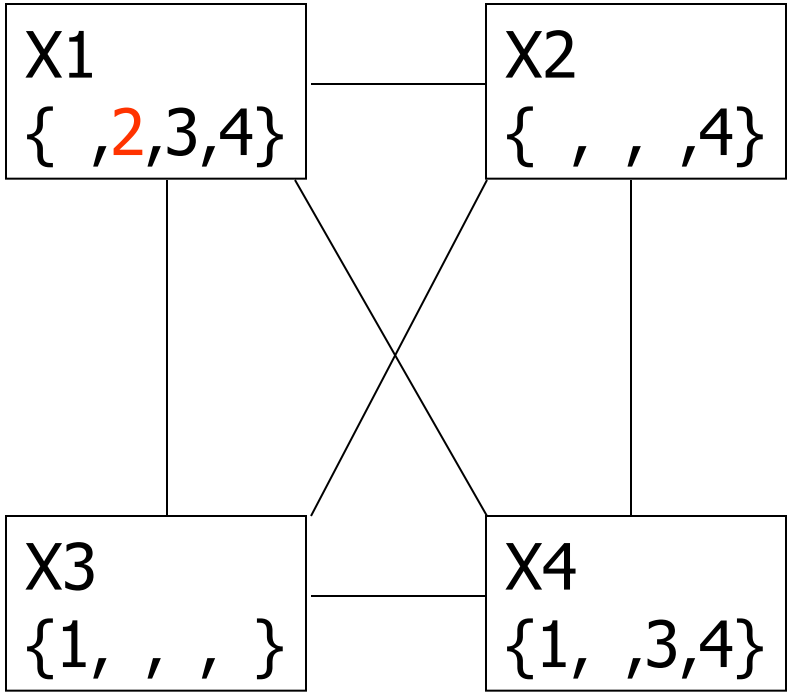




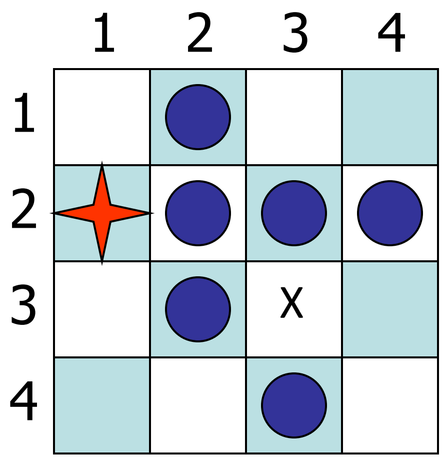
Can we eliminate any other values?

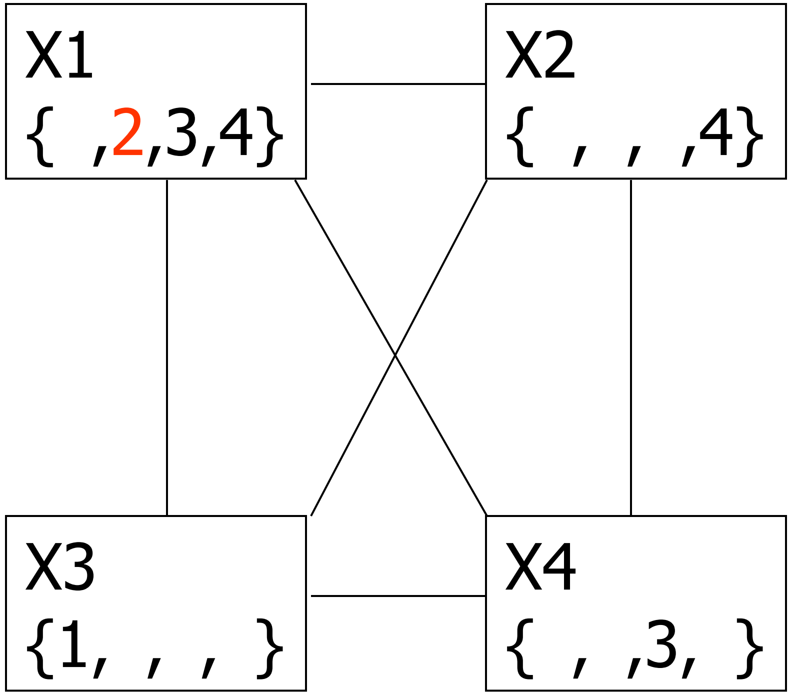
4-Queens Problem



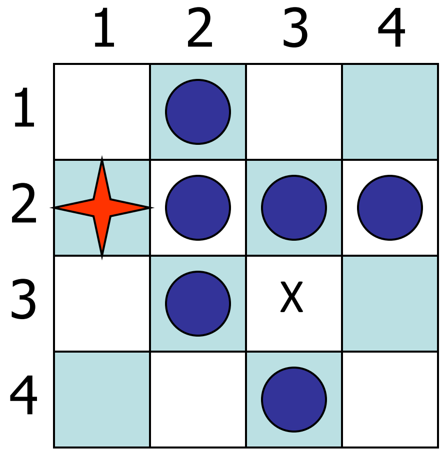


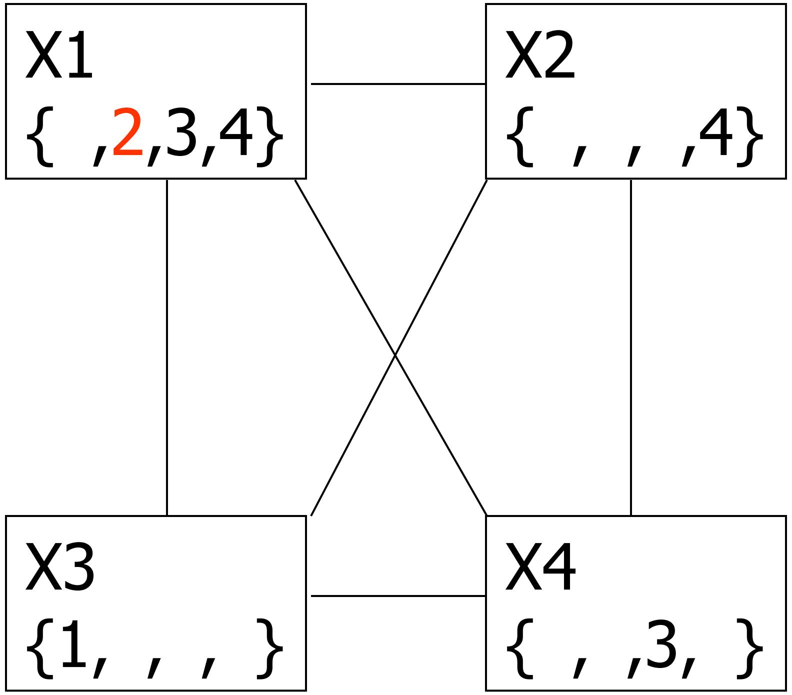
4-Queens Problem





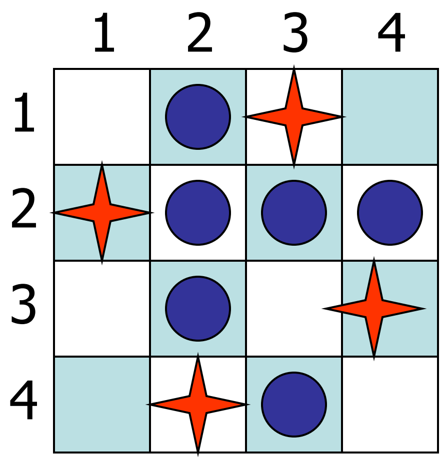
4-Queens Problem

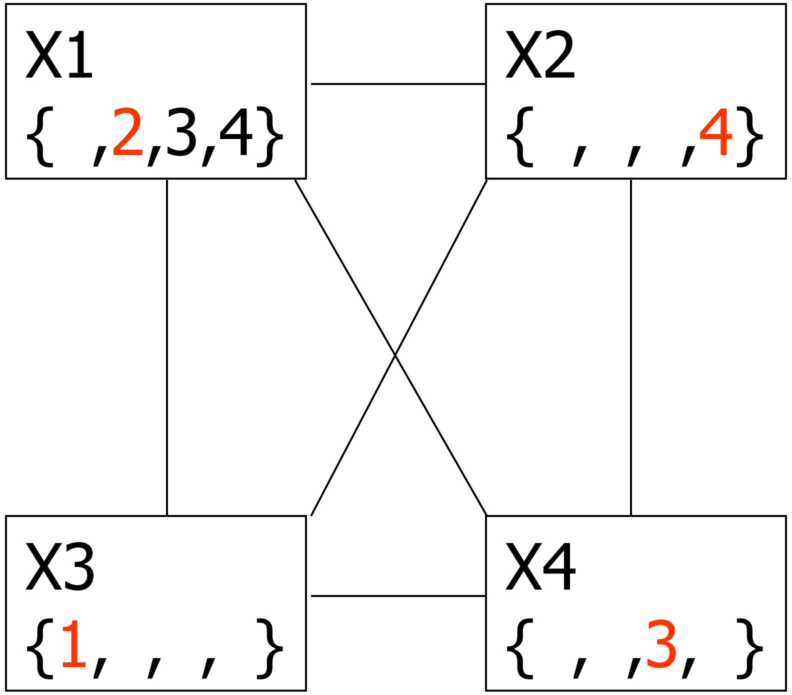




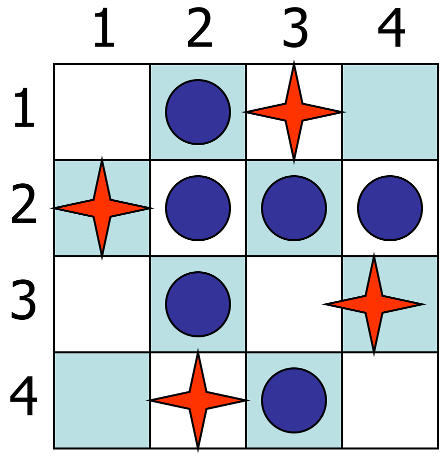
Arc constancy eliminates x3=3 because it's not consistent with X2's remaining values

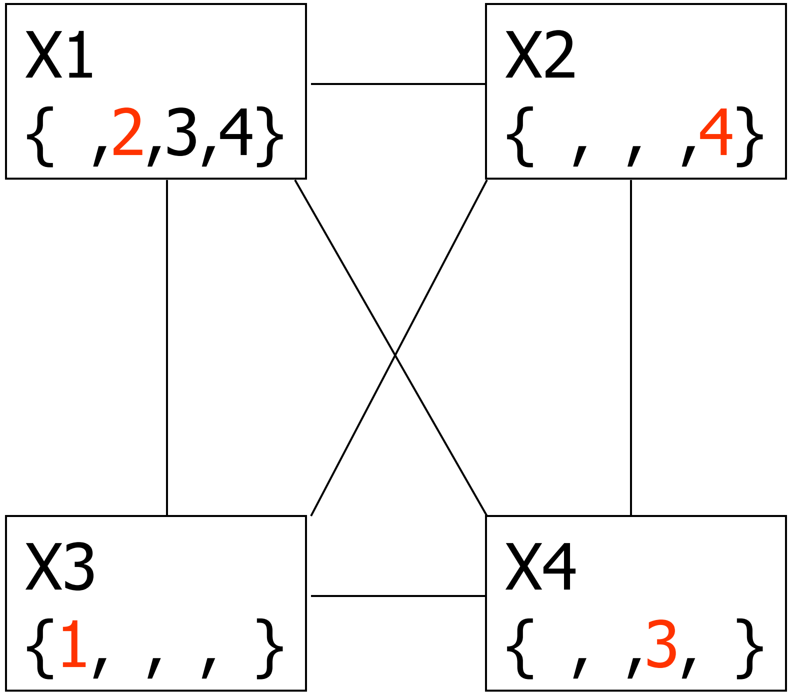
4-Queens Problem





4-Queens Problem





There is only one solution with X1=2

Sudoku

- Digit placement puzzle on 9x9 grid with unique answer
- Given an initial partially filled grid, fill remaining squares with a digit between 1 and 9
- Each column, row, and nine 3 × 3 sub-grids must contain all nine digits

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

- Some initial configurations are easy to solve and others very difficult

Sudoku Example

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

initial problem

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

a solution

How can we set this up as a CSP?

```
def sudoku(initValue):
  p = Problem()
  # Define a variable for each cell: 11,12,13...21,22,23...98,99
  for i in range(1, 10) :
    p.addVariables(range(i*10+1, i*10+10), range(1, 10))
  # Each row has different values
  for i in range(1, 10) :
    p.addConstraint(AllDifferentConstraint(), range(i*10+1, i*10+10))
  # Each column has different values
  for i in range(1, 10) :
    p.addConstraint(AllDifferentConstraint(), range(10+i, 100+i, 10))
  # Each 3x3 box has different values
  p.addConstraint(AllDifferentConstraint(), [11,12,13,21,22,23,31,32,33])
  p.addConstraint(AllDifferentConstraint(), [41,42,43,51,52,53,61,62,63])
  p.addConstraint(AllDifferentConstraint(), [71,72,73,81,82,83,91,92,93])

  p.addConstraint(AllDifferentConstraint(), [14,15,16,24,25,26,34,35,36])
  p.addConstraint(AllDifferentConstraint(), [44,45,46,54,55,56,64,65,66])
  p.addConstraint(AllDifferentConstraint(), [74,75,76,84,85,86,94,95,96])

  p.addConstraint(AllDifferentConstraint(), [17,18,19,27,28,29,37,38,39])
  p.addConstraint(AllDifferentConstraint(), [47,48,49,57,58,59,67,68,69])
  p.addConstraint(AllDifferentConstraint(), [77,78,79,87,88,89,97,98,99])

  # add unary constraints for cells with initial non-zero values
  for i in range(1, 10) :
    for j in range(1, 10):
      value = initValue[i-1][j-1]
      if value:
        p.addConstraint(lambda var, val=value: var == val, (i*10+j,))
  return p.getSolution()
```

```
# Sample problems
easy = [
 [0,9,0,7,0,0,8,6,0],
 [0,3,1,0,0,5,0,2,0],
 [8,0,6,0,0,0,0,0,0],
 [0,0,7,0,5,0,0,0,6],
 [0,0,0,3,0,7,0,0,0],
 [5,0,0,0,1,0,7,0,0],
 [0,0,0,0,0,0,1,0,9],
 [0,2,0,6,0,0,0,5,0],
 [0,5,4,0,0,8,0,7,0]]

hard = [
 [0,0,3,0,0,0,4,0,0],
 [0,0,0,0,7,0,0,0,0],
 [5,0,0,4,0,6,0,0,2],
 [0,0,4,0,0,0,8,0,0],
 [0,9,0,0,3,0,0,2,0],
 [0,0,7,0,0,0,5,0,0],
 [6,0,0,5,0,2,0,0,1],
 [0,0,0,0,9,0,0,0,0],
 [0,0,9,0,0,0,3,0,0]]

very_hard = [
 [0,0,0,0,0,0,0,0,0],
 [0,0,9,0,6,0,3,0,0],
 [0,7,0,3,0,4,0,9,0],
 [0,0,7,2,0,8,6,0,0],
 [0,4,0,0,0,0,0,7,0],
 [0,0,2,1,0,6,5,0,0],
 [0,1,0,9,0,5,0,4,0],
 [0,0,8,0,2,0,7,0,0],
 [0,0,0,0,0,0,0,0,0]]
```

Slide