Constraint Satisfaction Problems

Robert Platt
Northeastern University

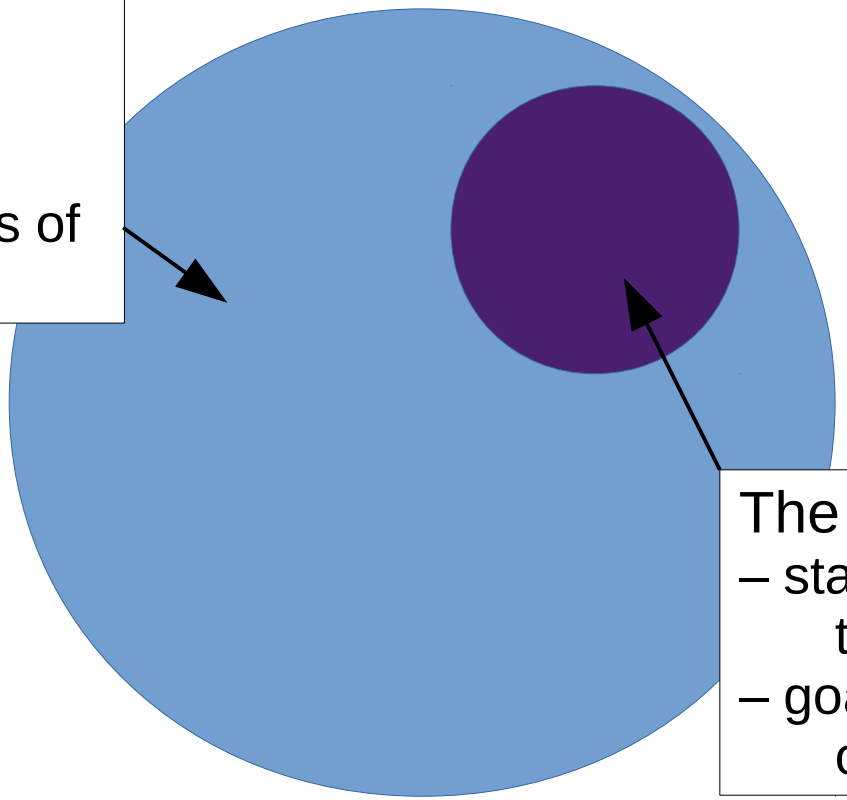
Some images and slides are used from:
1. AIMA

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

What is a CSP?

CSPs $\subset$ All search problems

The space of all search problems
– states and actions are atomic
– goals are arbitrary sets of states



The space of all CSPs
– states are defined in terms of variables
– goals are defined in terms of constraints

A CSP is defined by:
1. a set of variables and their associated domains
2. a set of constraints that must be satisfied.

CSP example: map coloring

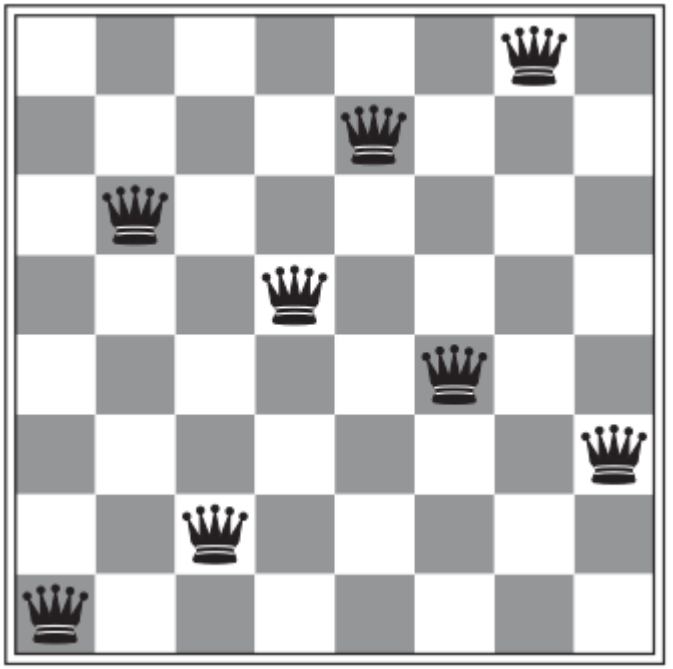
Problem: assign each territory a color such that no two adjacent territories have the same color

Variables: $X = \{WA, NT, Q, NSW, V, SA, T\}$

Domain of variables: $D = \{r, g, b\}$

Constraints: $C = \{SA \neq WA, SA \neq NT, SA \neq Q, \dots\}$

CSP example: n-queens



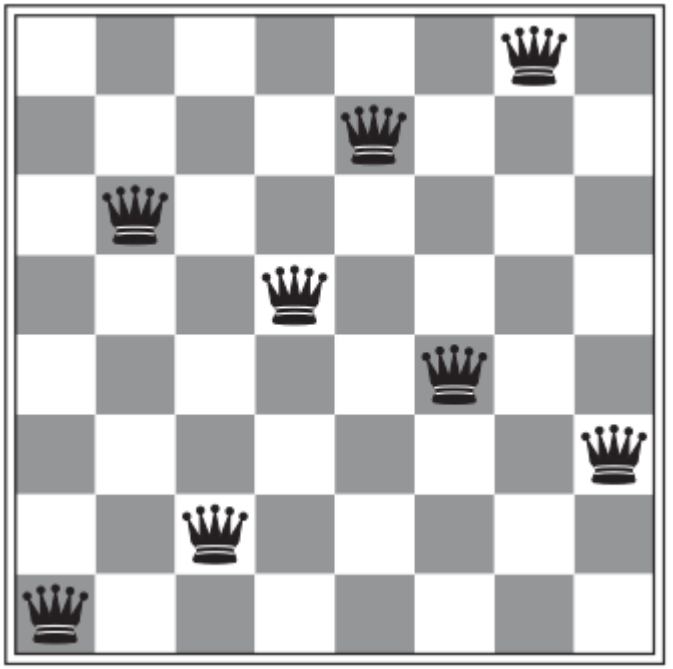
Problem: place n queens on an nxn chessboard such that no two queens threaten each other

Variables: $X = ?$

Domain of variables: $D = ?$

Constraints: $C = ?$

CSP example: n-queens



Problem: place n queens on an nxn chessboard such that no two queens threaten each other

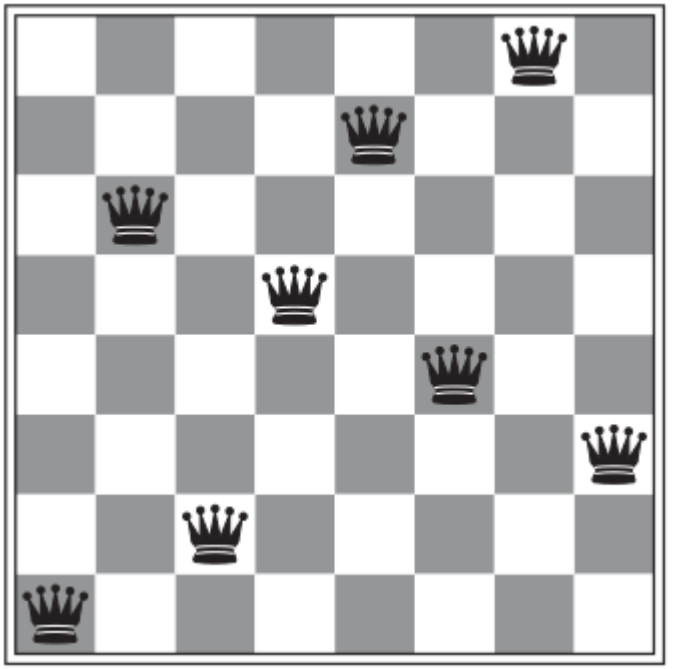
Variables: $X =$ One variable for every square

Domain of variables: $D =$ Binary

Constraints: $C =$ Enumeration of each possible disallowed configuration

– why is this a bad way to encode the problem?

CSP example: n-queens



Problem: place n queens on an nxn chessboard such that no two queens th

Variables

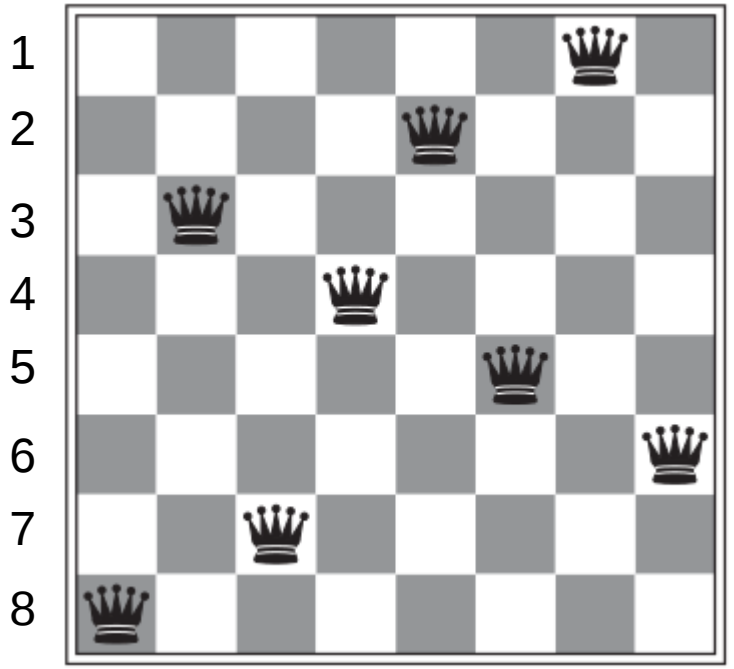
Domain d

Constrain ved configuration

Is there a better way?

– why is this a bad way to encode the problem?

CSP example: n-queens



Problem: place n queens on an nxn chessboard such that no two queens threaten each other

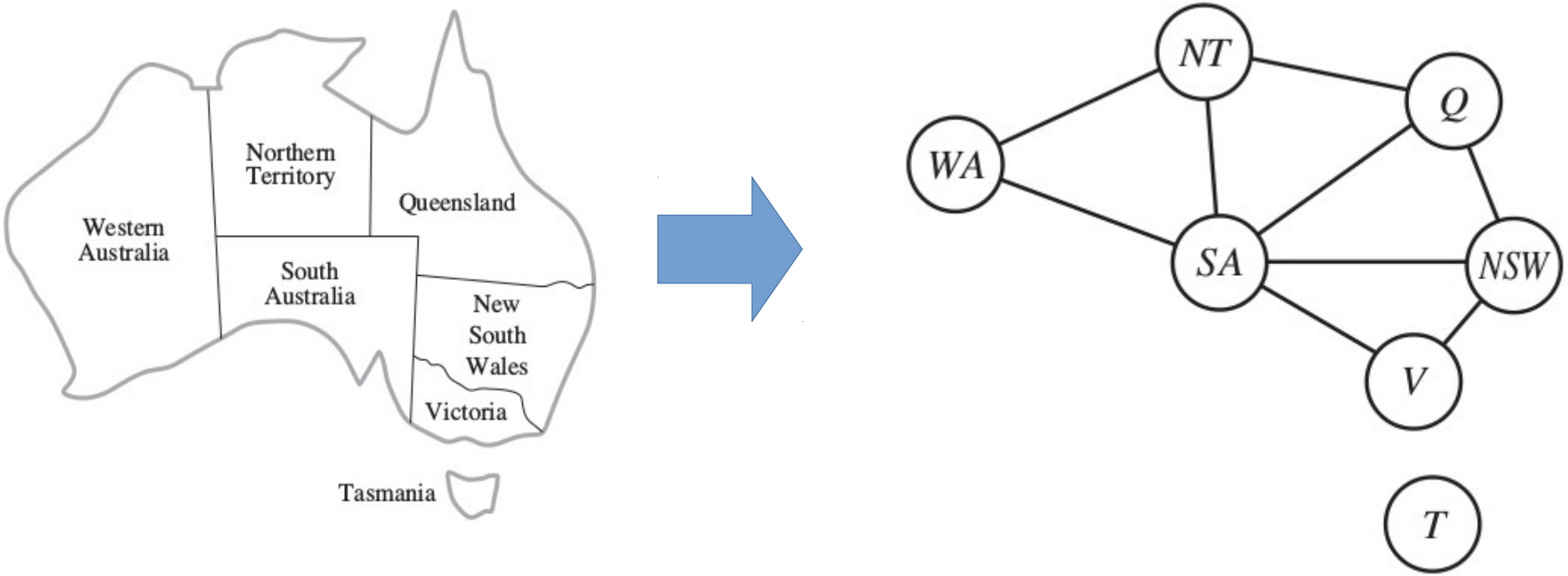
Variables: $X =$ One variable for each row

Domain of variables: $D =$ A number between 1 and 8

Constraints: $C =$ Enumeration of disallowed configurations

– why is this representation better?

The constraint graph



Variables represented as nodes (i.e. as circles)

Constraint relations represented as edges

– map coloring is a binary CSP, so it's easier to represent...

A harder CSP to represent: Cryptarithmetic

Variables: F, T, U, W, R, O, X_1, X_2, X_3

Domain of variables: integers between 0 and 9

Constraints:

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

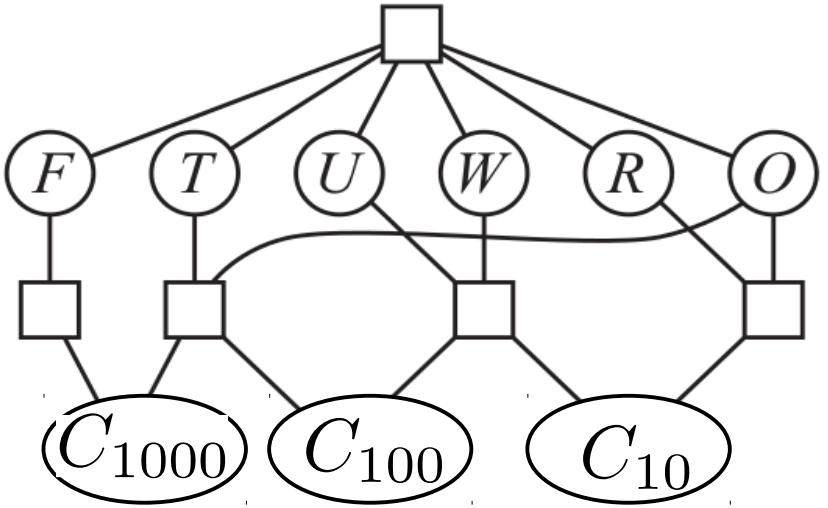
$$O + O = R + 10 \cdot C_{10}$$

$$C_{10} + W + W = U + 10 \cdot C_{100}$$

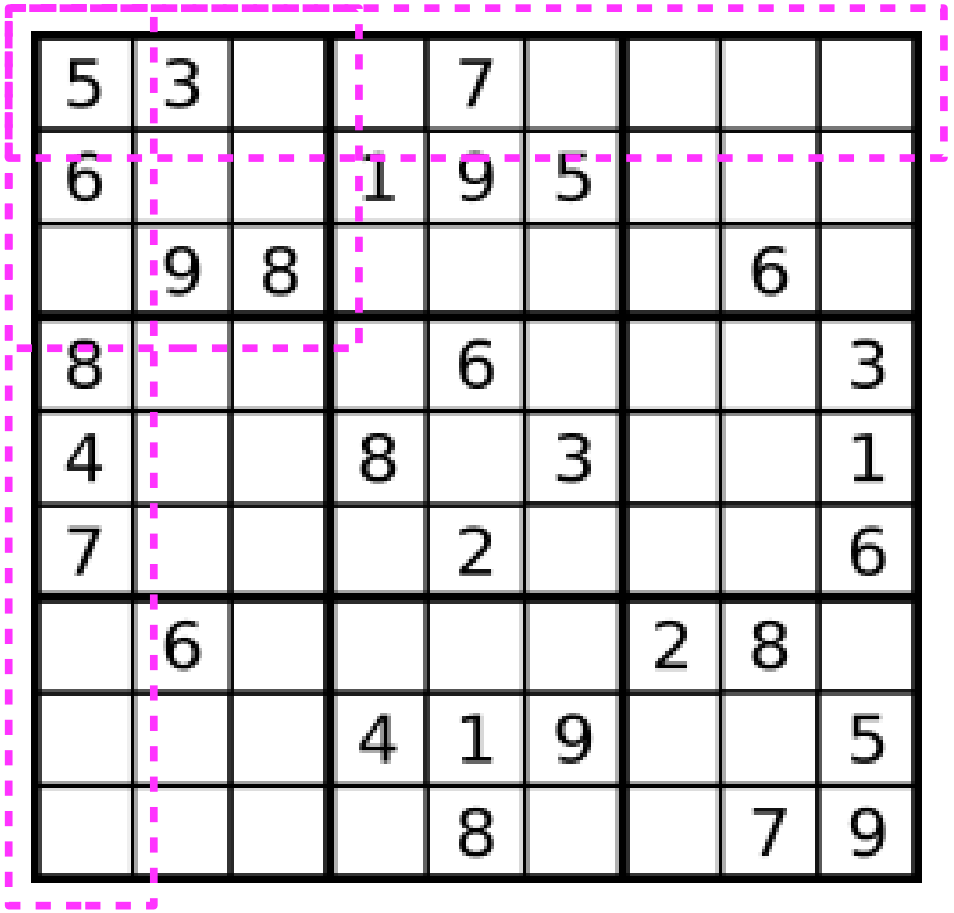
$$C_{100} + T + T = O + 10 \cdot C_{1000}$$

$$C_{1000} = F\ ,$$

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



Another example: sudoku



Variables: empty squares
Domains: 1-9
Constraints:
– alldiff for cols, rows, and 9-regions

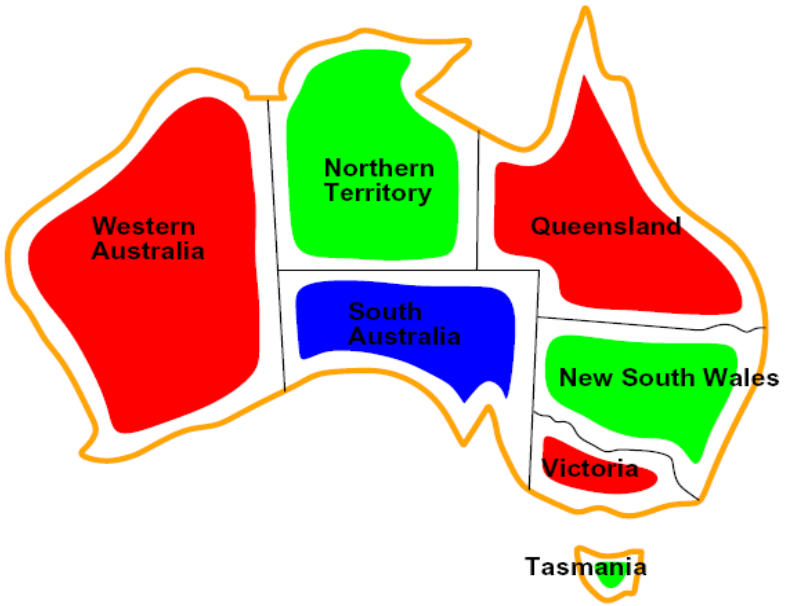
Types of Constraints

Some constraints are more complex than others:

Unary constraints: involve only one variable, e.g. WA=RED

Binary constraints: involve two variables, e.g. NT != Q

Higher-order constraints: ...



Types of CSPs

Finite domain:

- d^n complete assignments (where d = size of domain and n = num variables)
- NP-complete in the worst case, e.g. boolean SAT

Infinite domain:

- linear constrains → linear programming
- non-linear constraints → undecidable

Naive solution: apply BFS, DFS, A*, ...

Which would be better: BFS, DFS, A*?

– remember: it doesn't know if it reached
a goal until all variables are assigned ...

Naive solution: apply BFS, DFS, A*, ...

_ _ _ _ _ _ _

R _ _ _ _ _ _

R G _ _ _ _ _

R G R _ _ _ _

⋮

⋮

R G R R R R R

How many leaf nodes are expanded in the worst case?

Naive solution: apply BFS, DFS, A*, ...

_ _ _ _ _ _ _

R _ _ _ _ _ _

R G _ _ _ _ _

R G R _ _ _ _

⋮

⋮

R G R R R R R

How many leaf nodes are expanded in the worst case? $3^7 = 2187$

Naive solution: apply BFS, DFS, A*, ...

_ _ _ _ _ _ _

R _ _ _ _ _ _

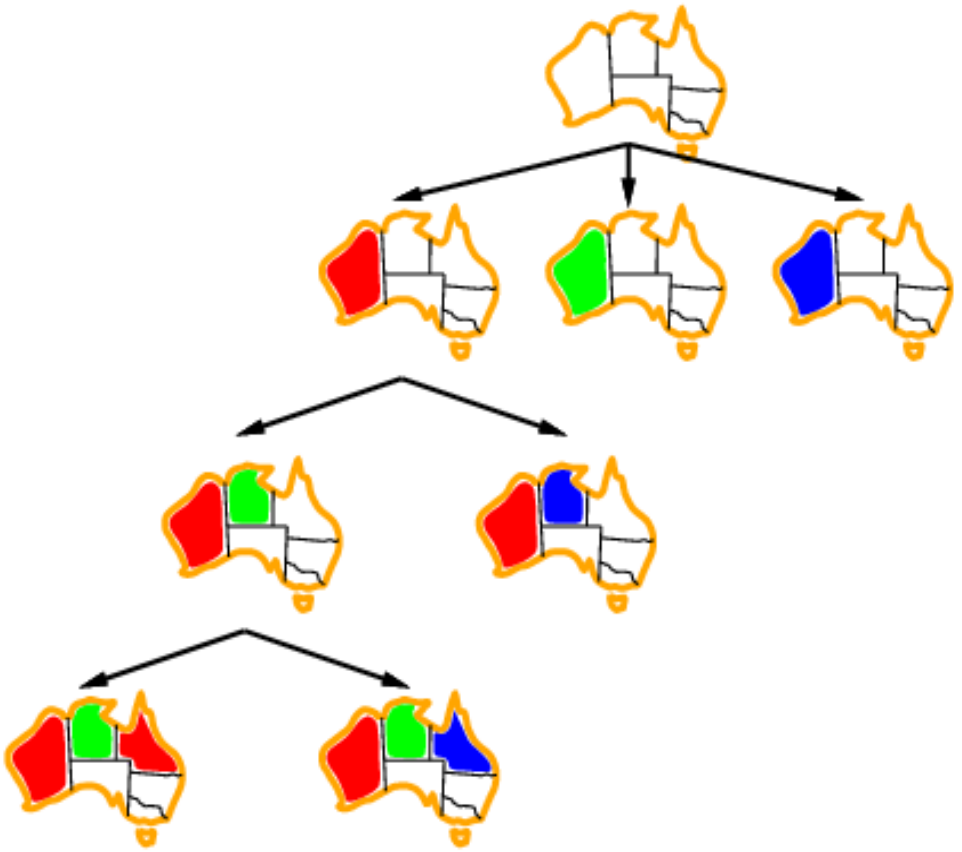
This will take a long time.
How can we speed it up?

R G R R R R R

How many leaf nodes are expanded in the worst case? $3^7 = 2187$

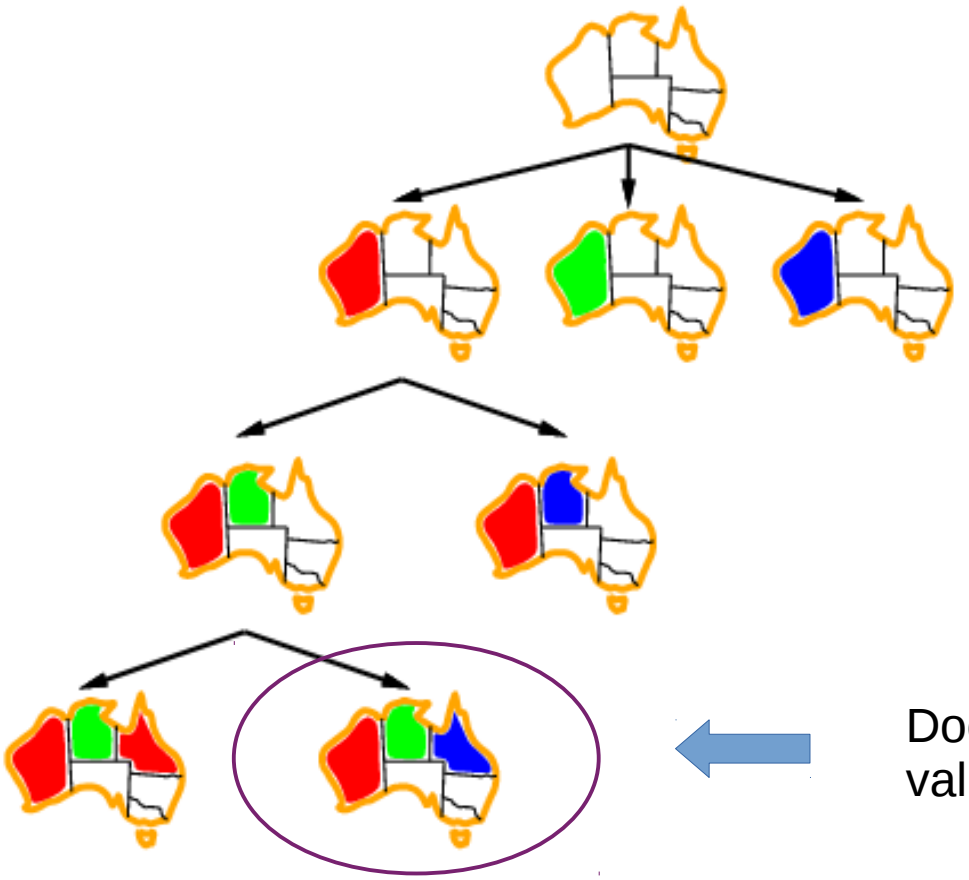
Backtracking search

When a node is expanded, check that each successor state is consistent before adding it to the queue.



Backtracking search

When a node is expanded, check that each successor state is consistent before adding it to the queue.



Does this state have any valid successors?

Backtracking search

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

– backtracking enables us the ability to solve a problem as big as 25-queens

Backtracking search

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences ← INFERENCE(csp, var, value)
            if inferences ≠ failure then
                add inferences to assignment
                result ← BACKTRACK(assignment, csp)
                if result ≠ failure then
                    return result
        remove {var = value} and inferences from assignment
    return failure
```

We'll talk about the inference part in a moment...

– backtracking enables us the ability to solve a problem as big as 25-queens

Forward checking

Sometimes, failure is inevitable:

Can we detect this situation in advance?

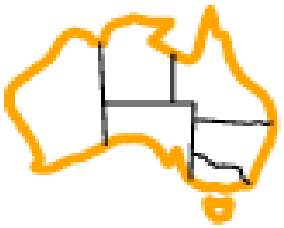
Forward checking

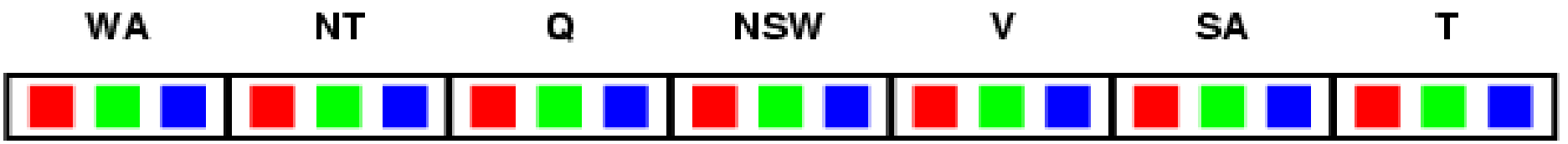
Sometimes, failure is inevitable:

Can we detect this situation in advance?

Yes: keep track of viable variable assignments as you go

Forward checking



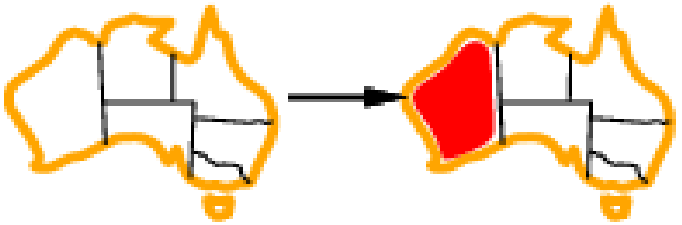


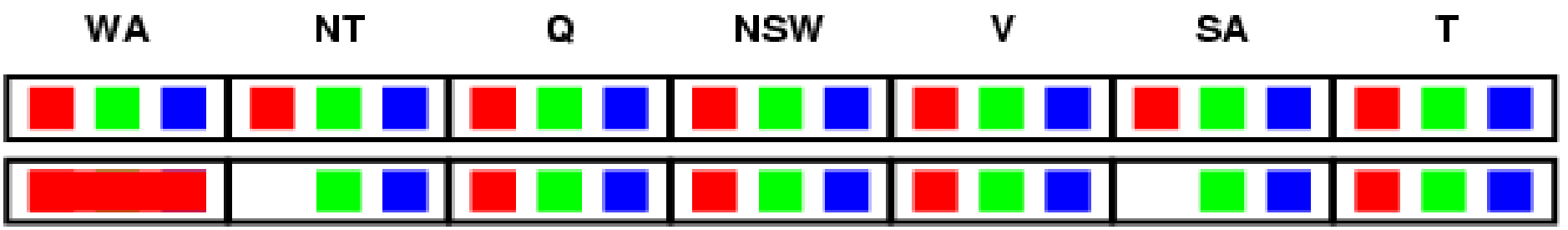
Track domain for each unassigned variable

– initialize w/ domains from problem statement

– each time you expand a node, update domains of all unassigned variables

Forward checking



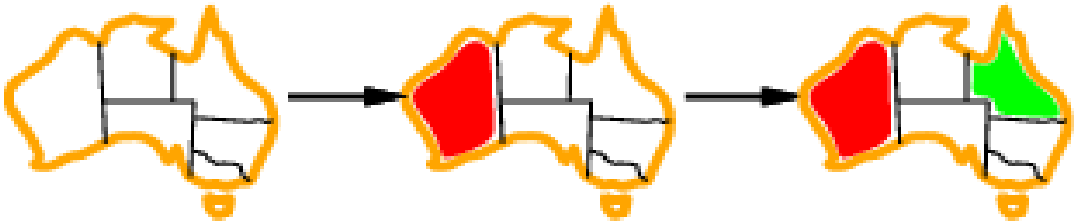


Track domain for each unassigned variable

– initialize w/ domains from problem statement

– each time you expand a node, update domains of all unassigned variables

Forward checking

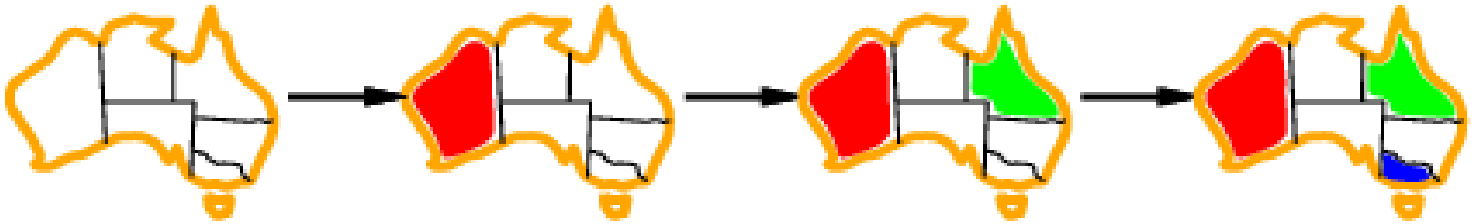


Track domain for each unassigned variable

– initialize w/ domains from problem statement

– each time you expand a node, update domains of all unassigned variables

Forward checking



Track domain for each unassigned variable

– initialize w/ domains from problem statement

– each time you expand a node, update domains of all unassigned variables

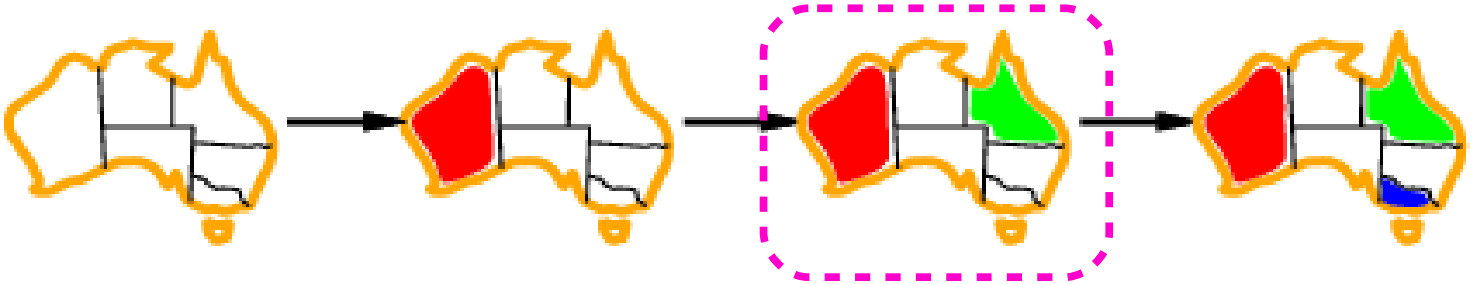
Backtracking search

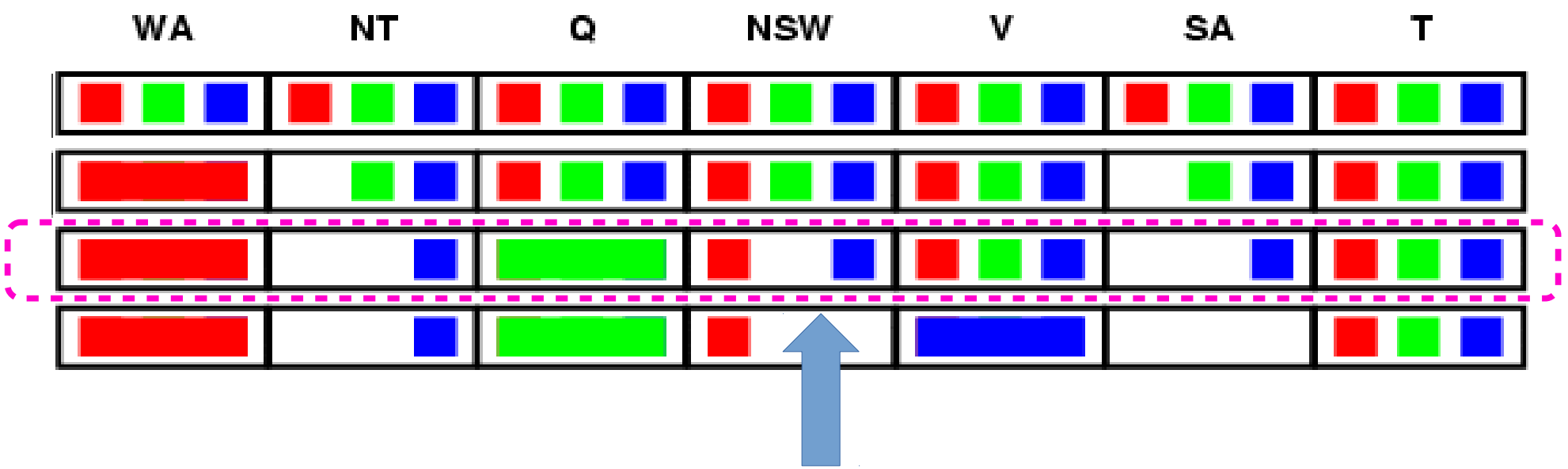
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

Could infer a partial assignment using forward checking...

– backtracking enables us the ability to solve a problem as big as 25-queens

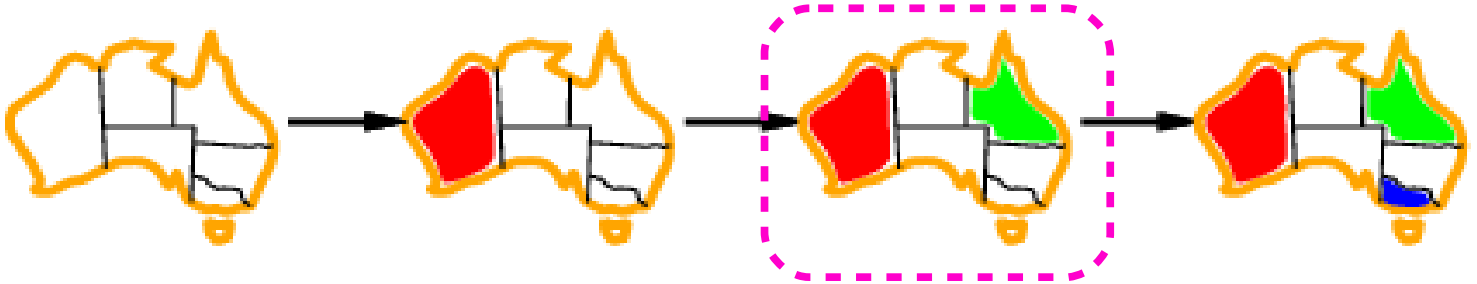
Forward checking

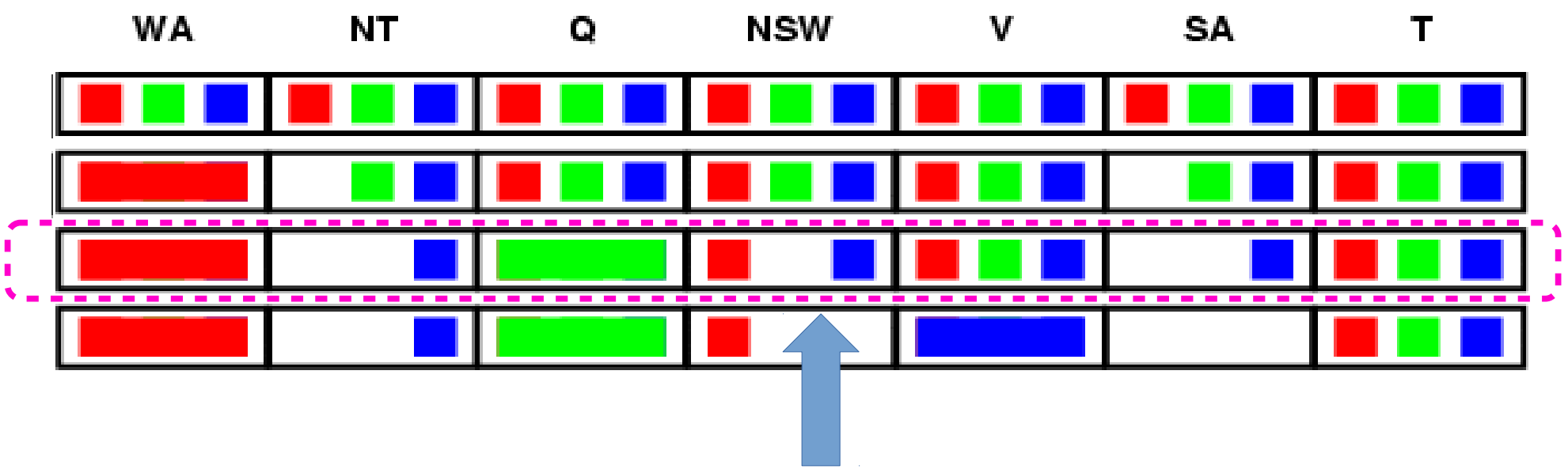




But, failure was inevitable here!
– what did we miss?

Forward checking





This is failure is implied, not explicit.
– need to do inference to detect it...

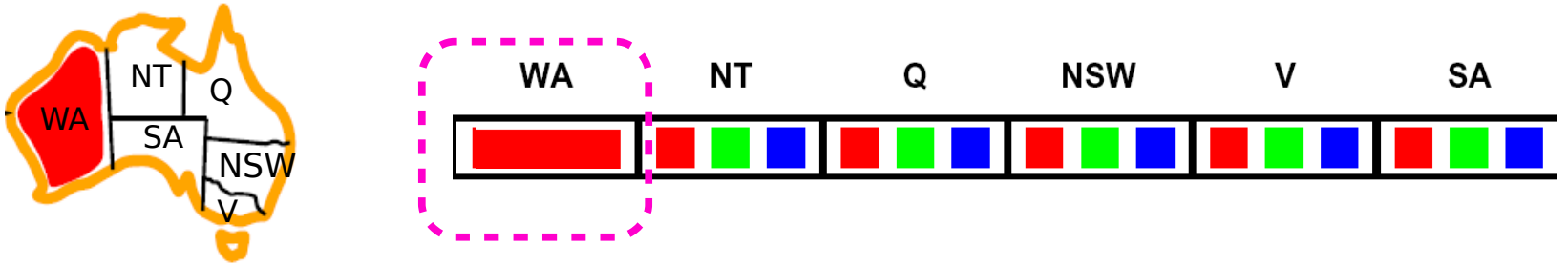
Arc consistency

Want to detect implied constraint violations:
– iterate a rule called arc consistency
– subsumes forward checking

Arc Consistency:

Arc consistency

Want to detect <u>implied</u> constraint violations:
– iterate a rule called arc consistency
– subsumes forward checking

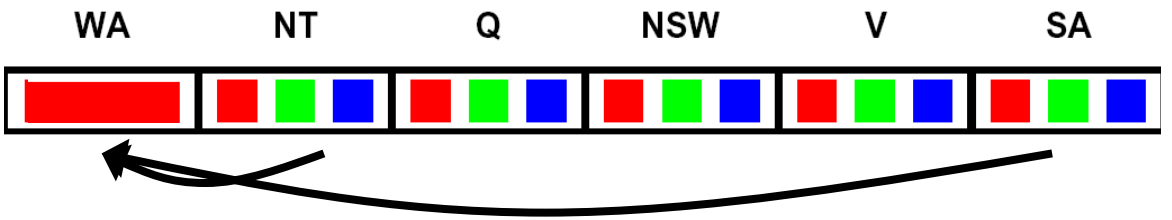


<u>Arc Consistency:</u>

1. given a changed variable:

Arc consistency

Want to detect implied constraint violations:
– iterate a rule called arc consistency
– subsumes forward checking

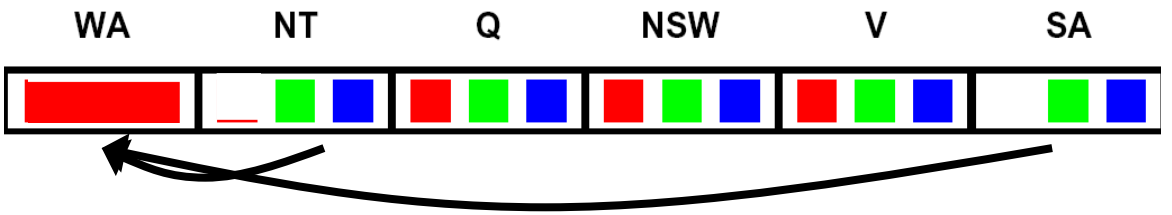


Arc Consistency:

1. given a changed variable:
2. draw arcs pointing from all adjacent variables to changed variable

Arc consistency

Want to detect implied constraint violations:
– iterate a rule called arc consistency
– subsumes forward checking

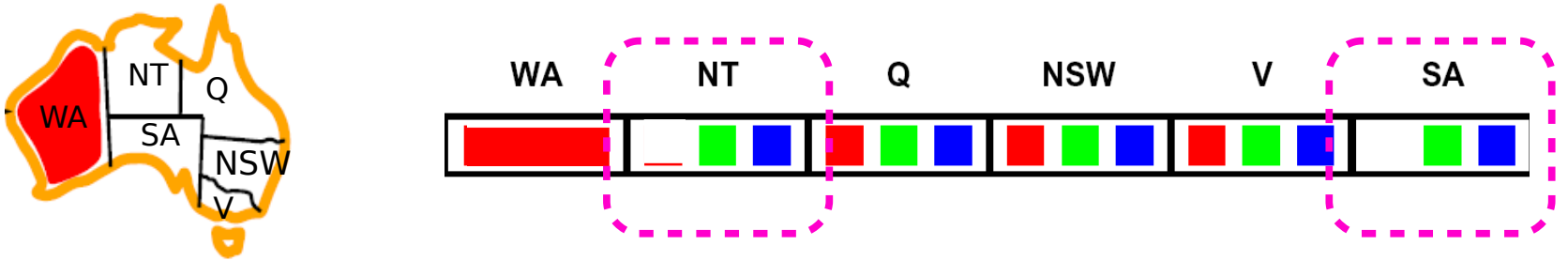


Arc Consistency:

1. given a changed variable:
2. draw arcs pointing from all adjacent variables to changed variable
3. delete conflicting values from domains *at the tail*
   – *for every value in tail, there must be some value in head that is consistent.*

Arc consistency

Want to detect implied constraint violations:
– iterate a rule called arc consistency
– subsumes forward checking

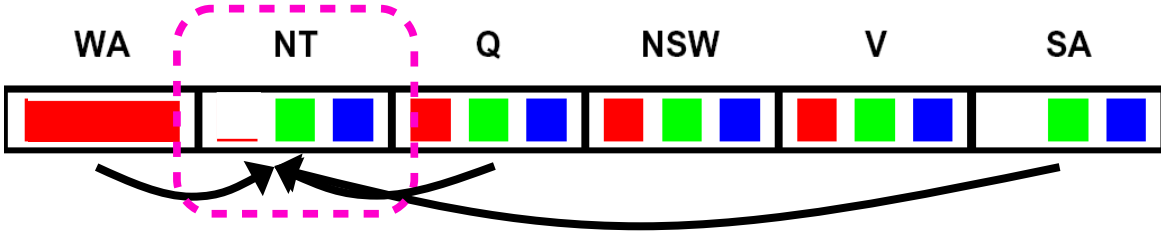


Arc Consistency:

1. given a changed variable:
2. draw arcs pointing from all adjacent variables to changed variable
3. delete conflicting values from domains *at the tail*
   – *for every value in tail, there must be some value in head that is consistent.*
4. rinse and repeat

Arc consistency

Want to detect implied constraint violations:
– iterate a rule called arc consistency
– subsumes forward checking



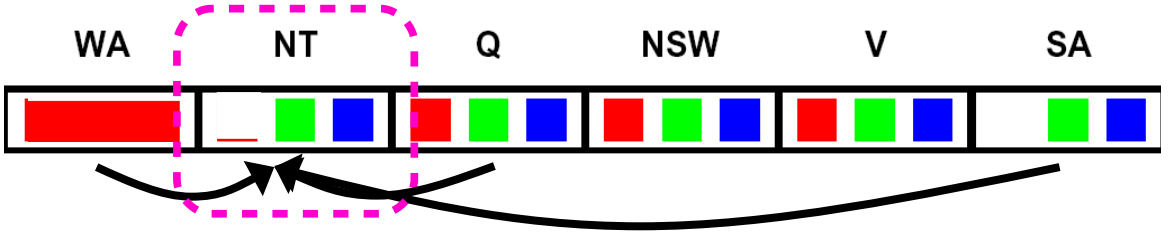
Arc Consistency:

1. given a changed variable:
2. draw arcs pointing from all adjacent variables to changed variable
3. delete conflicting values from domains *at the tail*
   – *for every value in tail, there must be some value in head that is consistent.*
4. rinse and repeat

Arc consistency

Want to detect implied constraint viola
– iterate a rule called arc consistency
– subsumes forward checking

Okay?

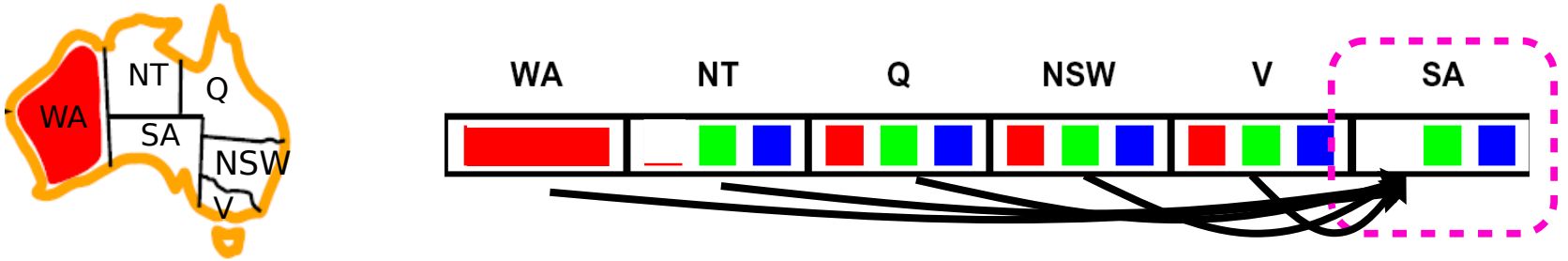


Arc Consistency:

1. given a changed variable:
2. draw arcs pointing from all adjacent variables to changed variable
3. delete conflicting values from domains *at the tail*
   – *for every value in tail, there must be some value in head that is consistent.*
4. rinse and repeat

Arc consistency

Want to detect implied constraint violations:
– iterate a rule called arc consistency
– subsumes forward checking



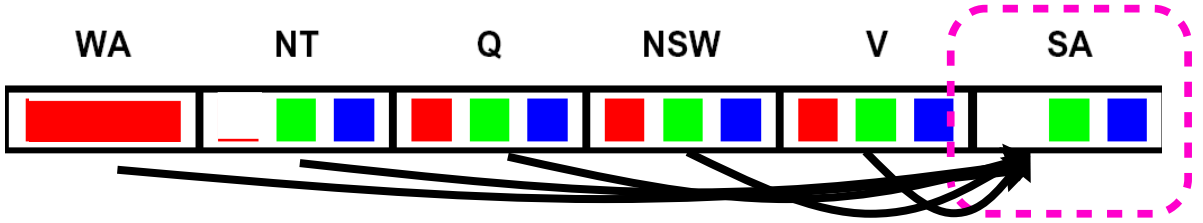
Arc Consistency:

1. given a changed variable:
2. draw arcs pointing from all adjacent variables to changed variable
3. delete conflicting values from domains *at the tail*
   – *for every value in tail, there must be some value in head that is consistent.*
4. rinse and repeat

Arc consistency

Want to detect implied constraint viola
– iterate a rule called arc consistency
– subsumes forward checking

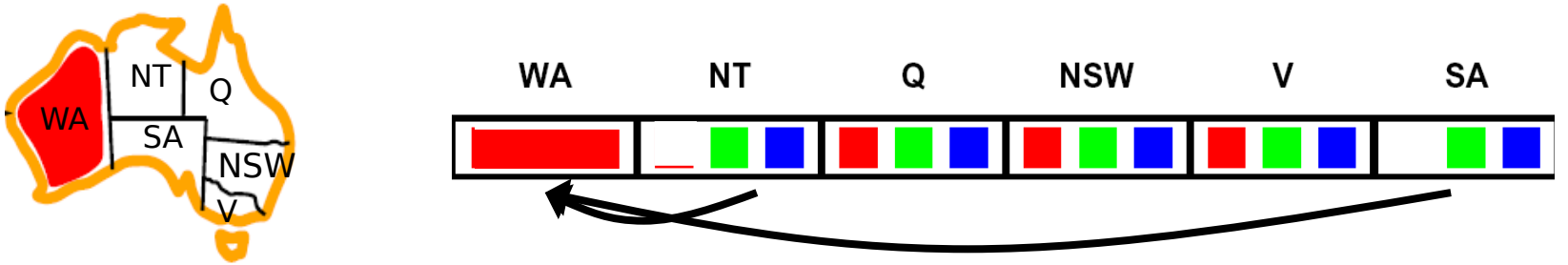
Okay?



Arc Consistency:

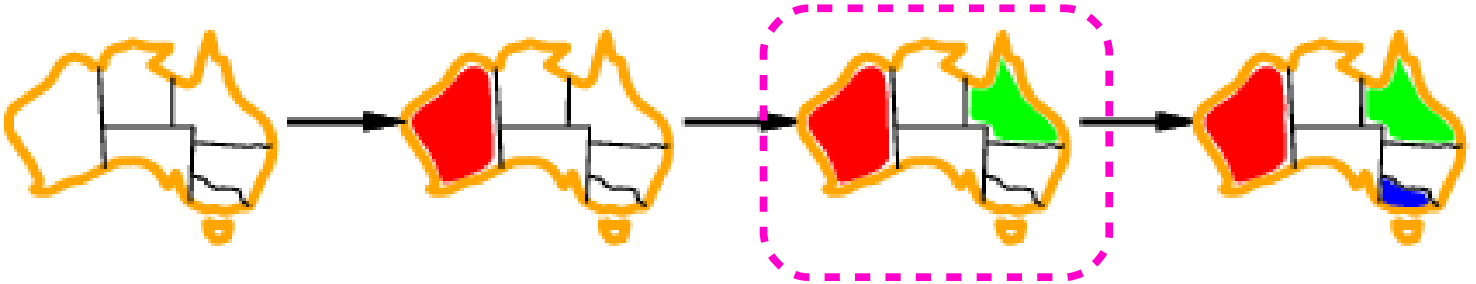
1. given a changed variable:
2. draw arcs pointing from all adjacent variables to changed variable
3. delete conflicting values from domains *at the tail*
   – *for every value in tail, there must be some value in head that is consistent.*
4. rinse and repeat

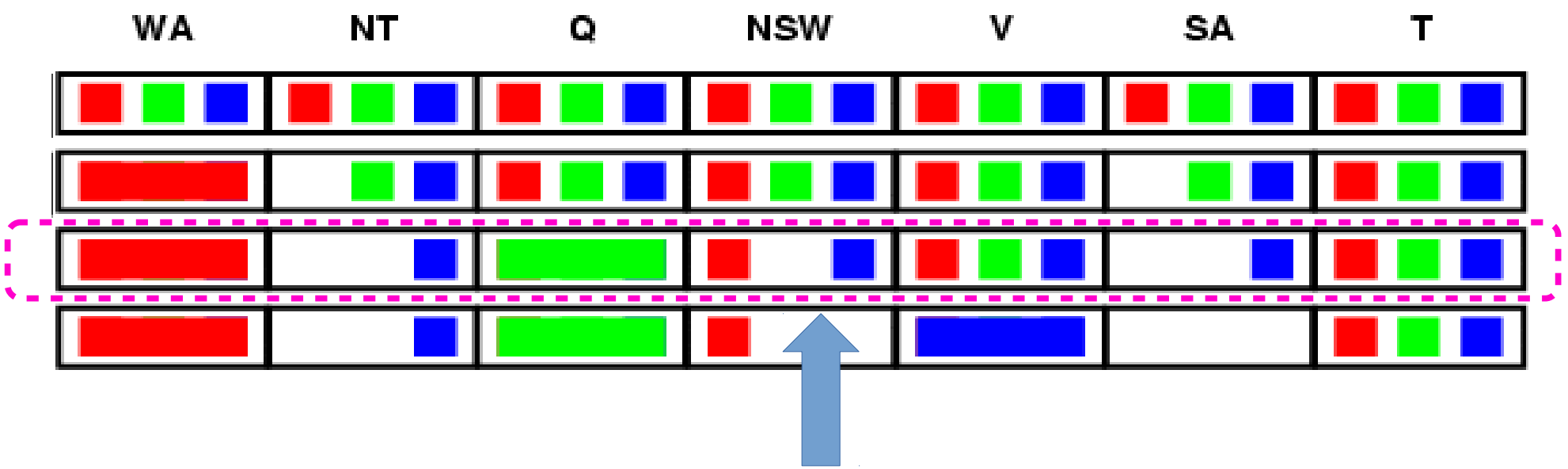
Forward Checking & Arc consistency



Forward checking is arc consistency applied once to each new variable assignment.

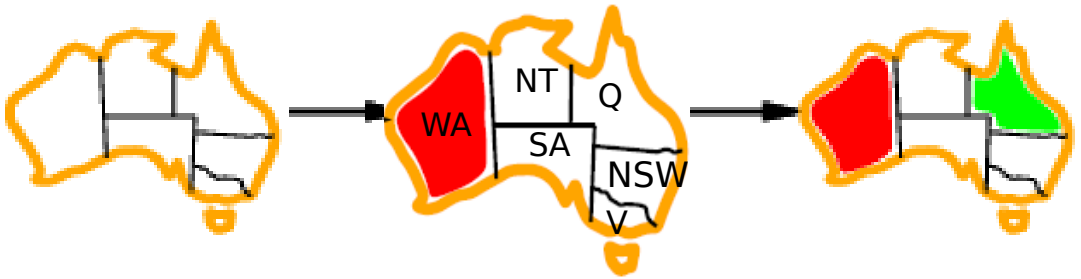
Forward checking

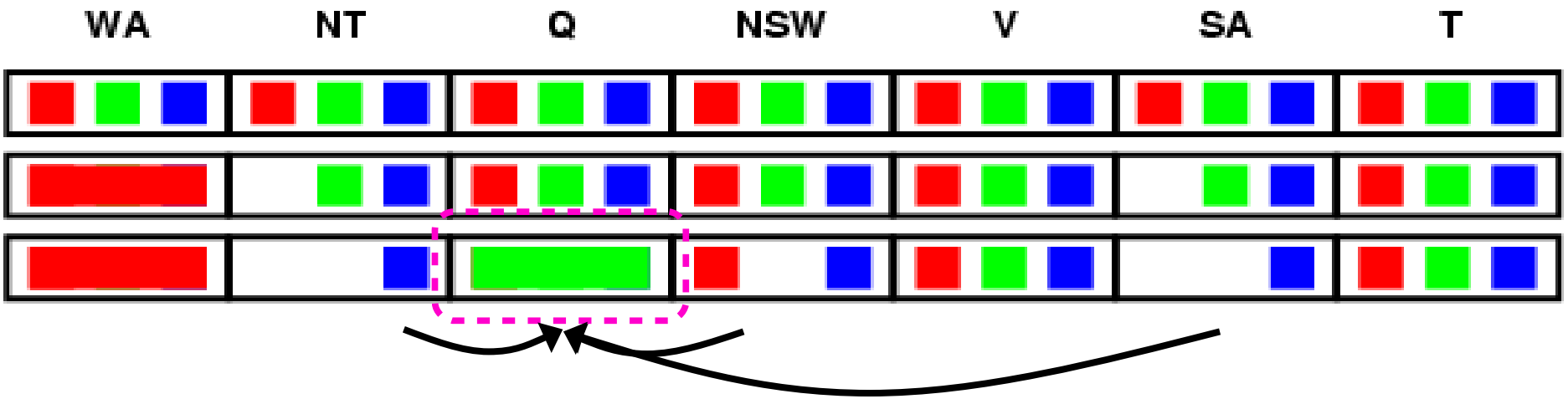




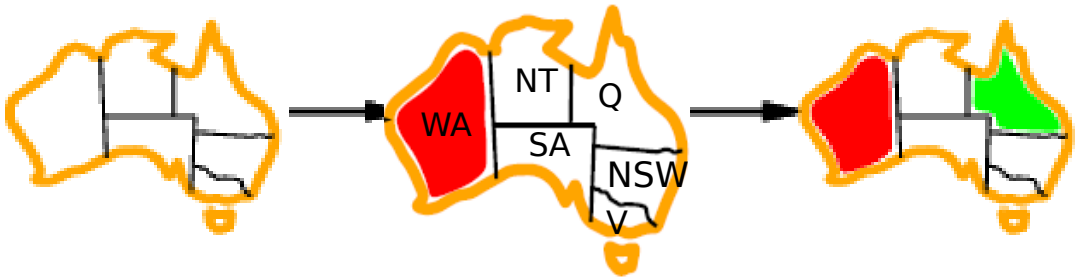
But, failure was inevitable here!
– what did we miss?

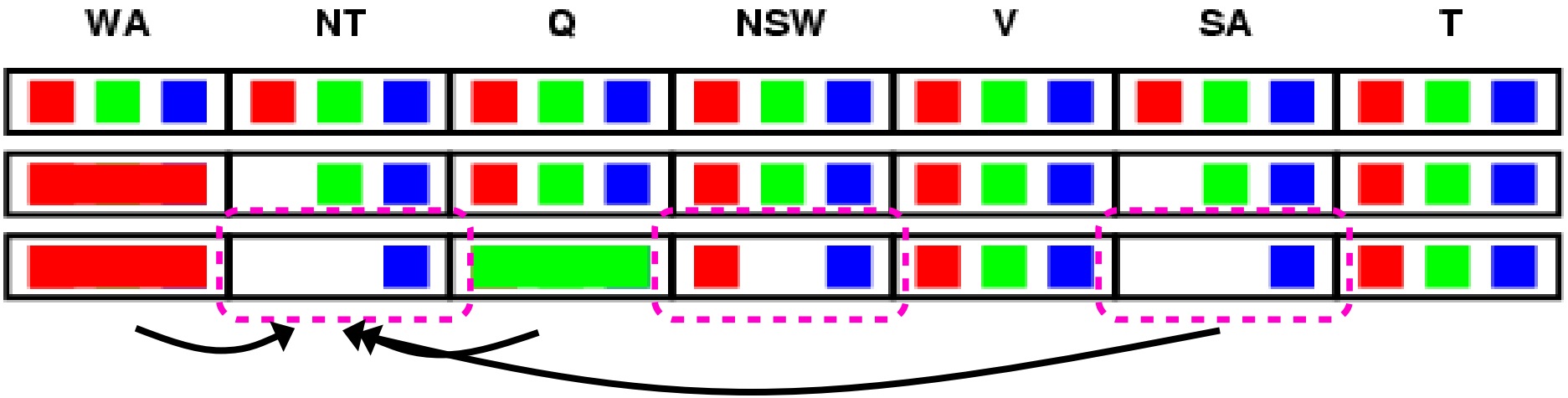
Arc consistency



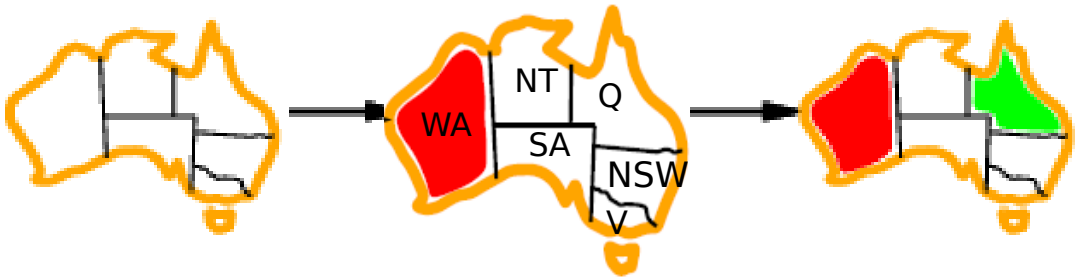


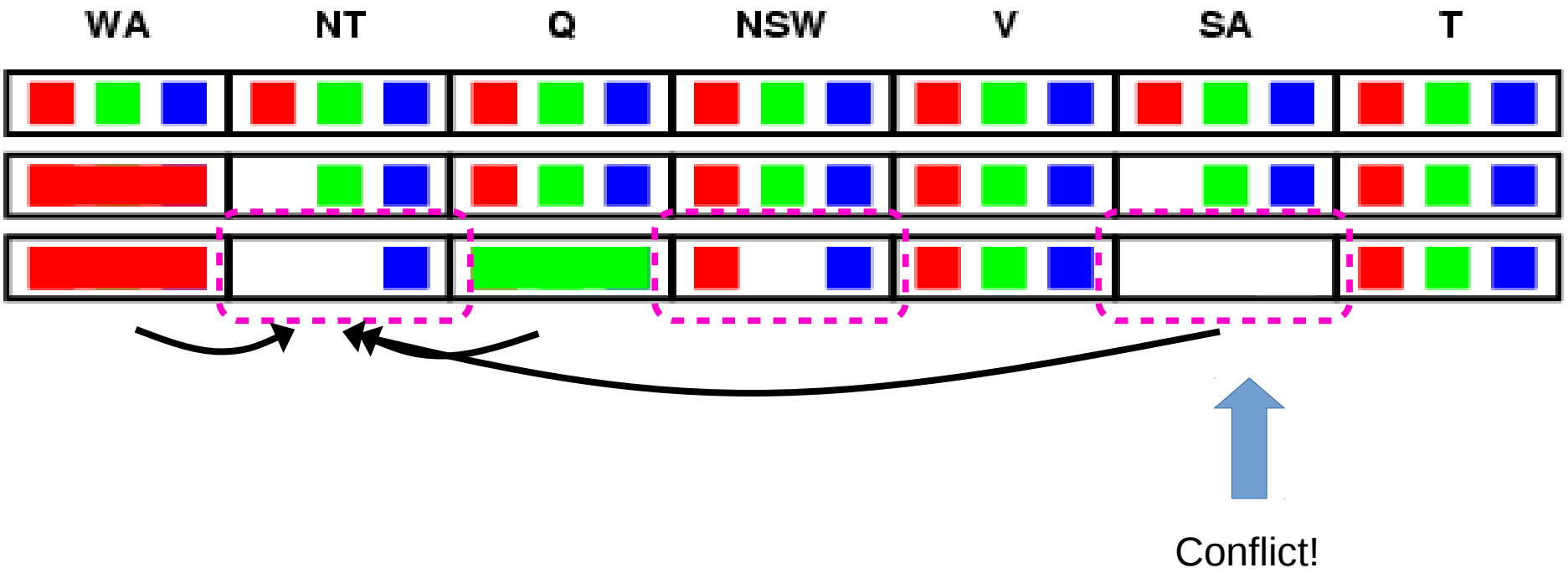
Arc consistency





Arc consistency





Conflict!

Arc consistency

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

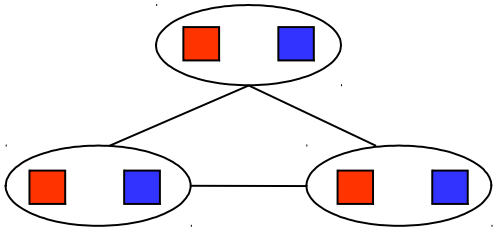
  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)
    if REVISE(*csp*, X_i, X_j) **then**
      if size of $D_i = 0$ **then return** *false*
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add (X_k, X_i) to *queue*
  return *true*

---

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

Why does this algorithm converge?

Arc consistency?



Is this arrangement arc-consistent?

Does a feasible solution exist?

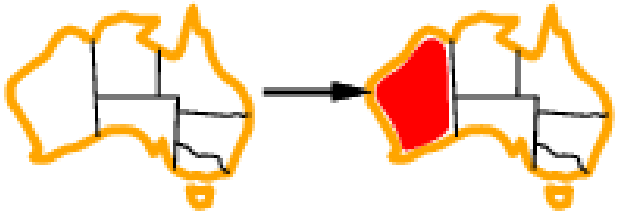
What happened?

Image: Berkeley CS188 course notes (downloaded Summer 2015)

Heuristics for improving CSP performance

Minimum remaining values (MRV) heuristic:

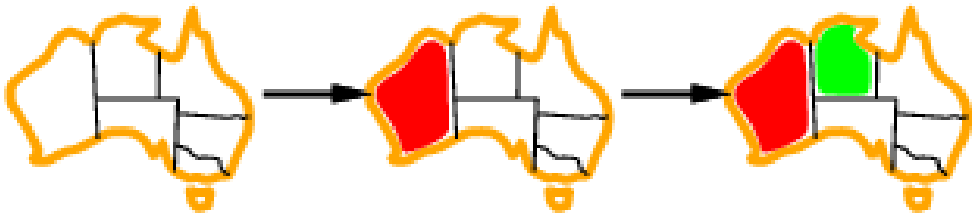
– expand variables w/ minimum size domain first



Heuristics for improving CSP performance

Minimum remaining values (MRV) heuristic:

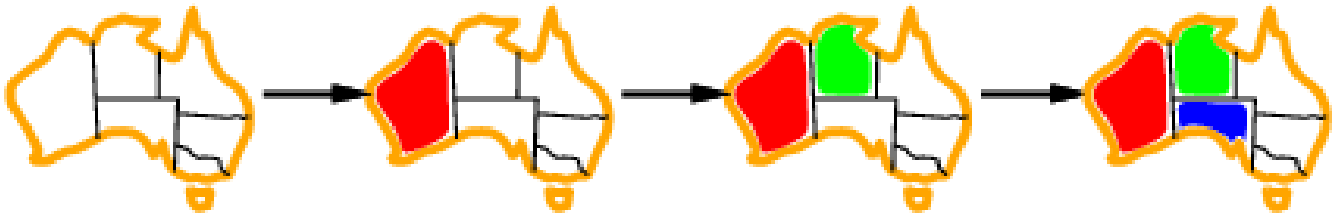
– expand variables w/ minimum size domain first



Heuristics for improving CSP performance

Minimum remaining values (MRV) heuristic:

– expand variables w/ minimum size domain first

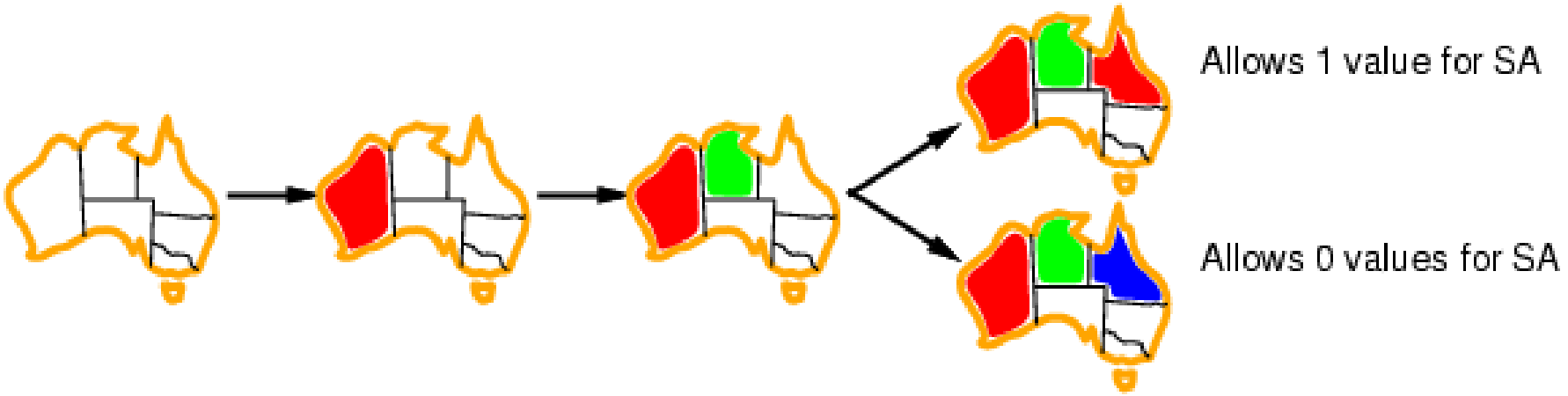


Heuristics for improving CSP performance

Least constraining value (LCV) heuristic:

– consider how domains of neighbors would change under A.C.

– choose value that contrains neighboring domains the **least**



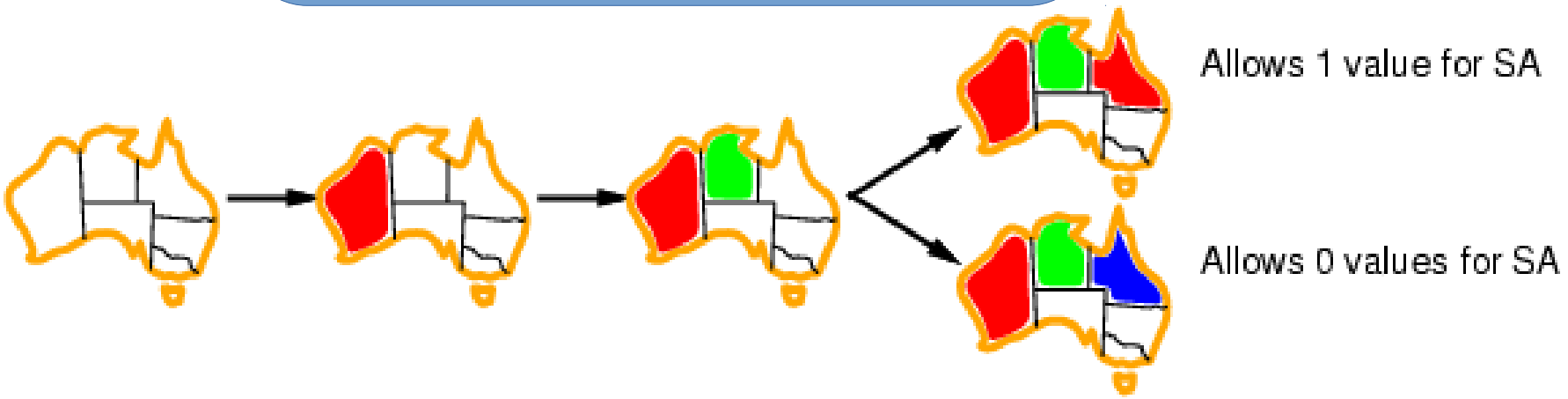
Heuristics for improving CSP performance

Least constraining value (LCV) heuristic:

– consid… would change
unde…

– choos… ring domains
the **le**…

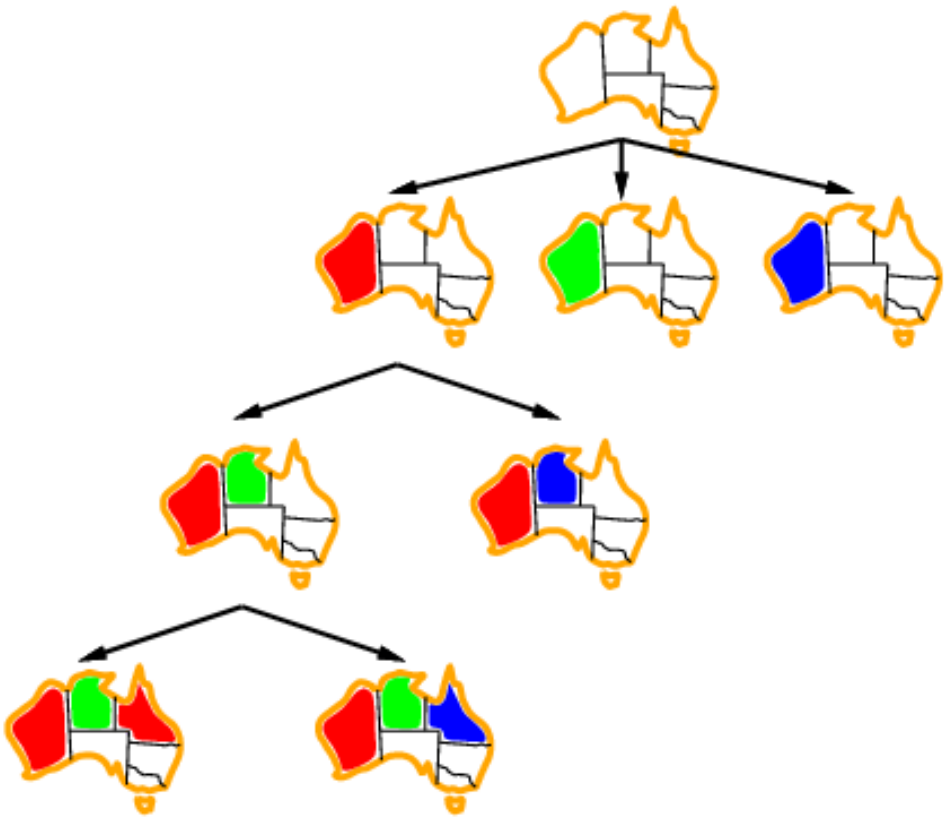
The combination of MRV and LCV w/ backtracking can solve the 1000-queens problem



Using structure to reduce problem complexity

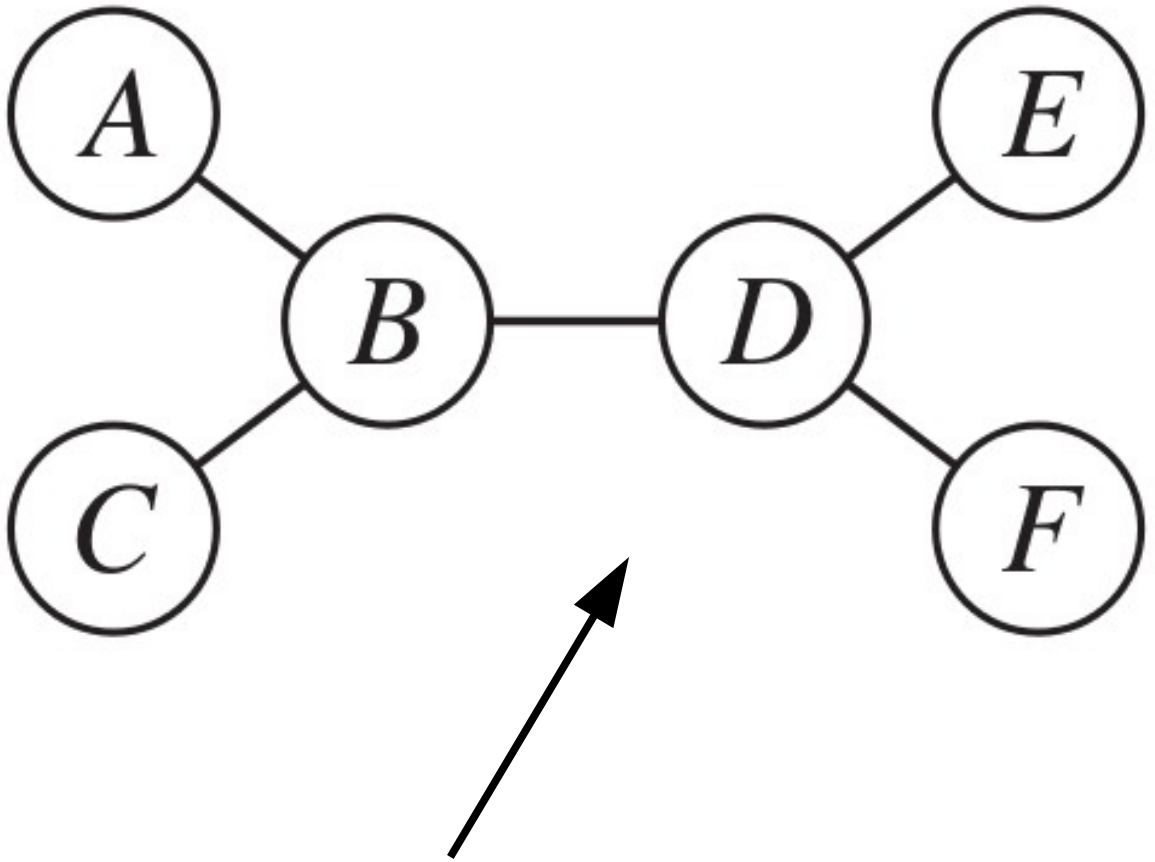
In general, what is the complexity of solving a CSP using backtracking?

(in terms of # variables, n, and max domain size, d)



But, sometimes CSPs have special structure that makes them simpler!

When the constraint graph is a tree



This CSP is easier to solve than the general case...

When the constraint graph is a tree

1. Do a *topological sort*
   – a partial ordering over variables

   i. choose any node as the root
   ii. list children after their parents

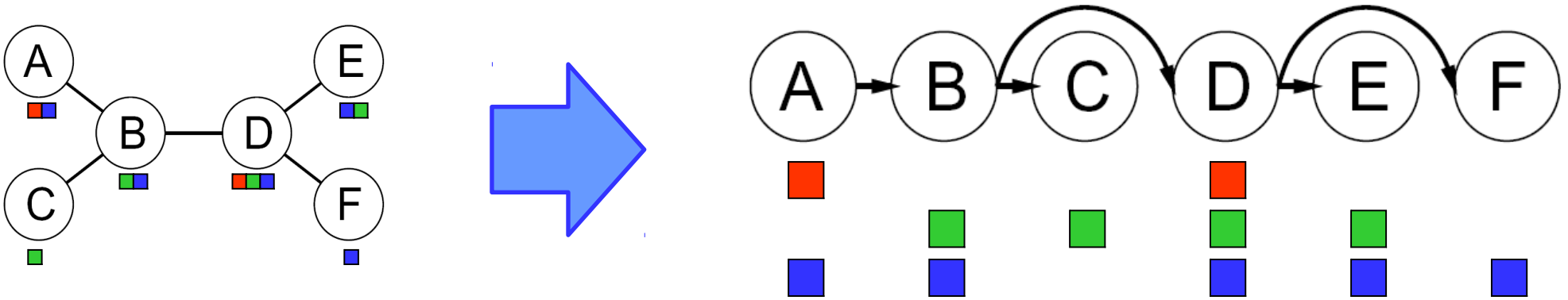


Image: Berkeley CS188 course notes (downloaded Summer 2015)

When the constraint graph is a tree

2. make the graph *directed arc consistent*
   – start w/ the tail and make each variable arc consistent wrt its parents

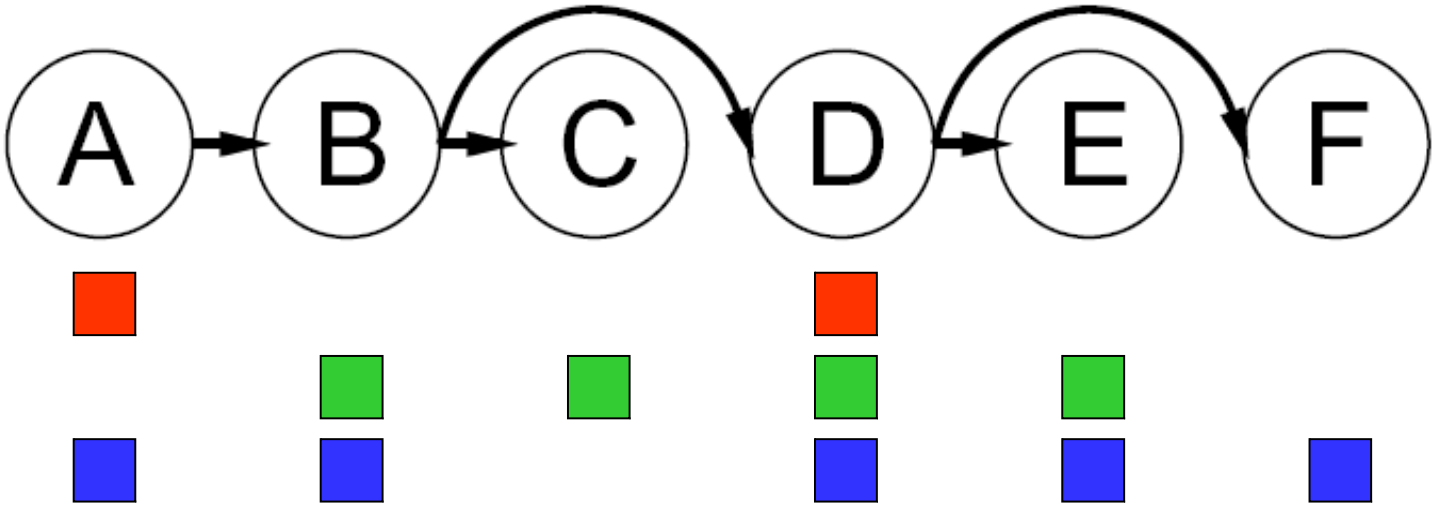
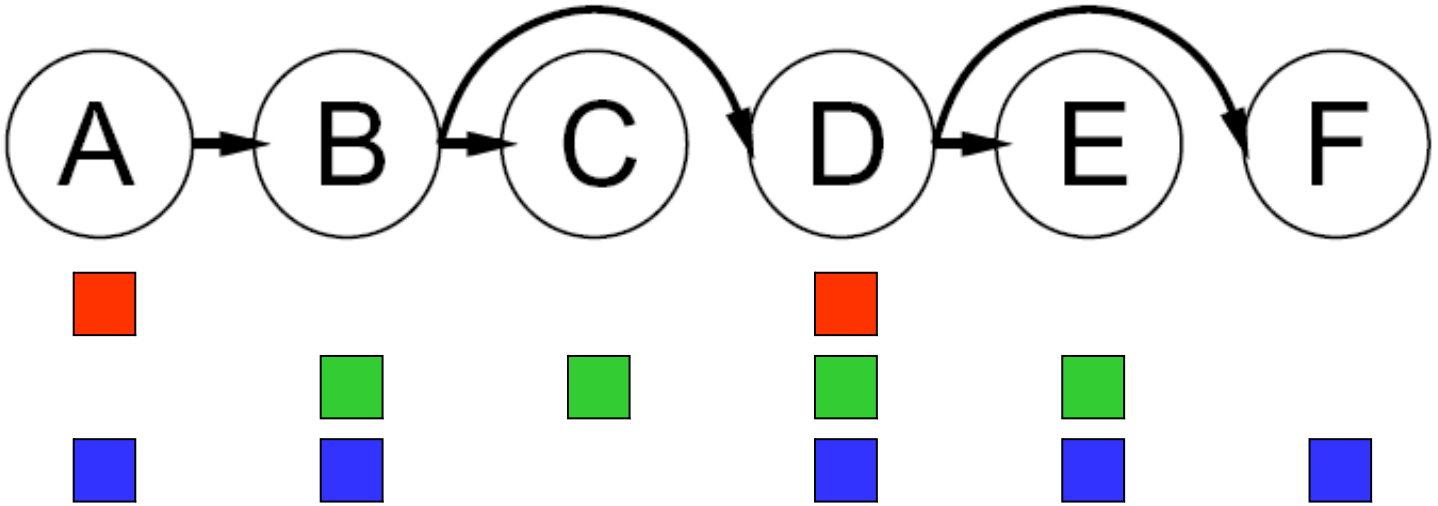


Image: Berkeley CS188 course notes (downloaded Summer 2015)

When the constraint graph is a tree

2. make the graph *directed arc consistent*
   – start w/ the tail and make each variable arc consistent wrt its parents



ok

Image: Berkeley CS188 course notes (downloaded Summer 2015)

When the constraint graph is a tree

2. make the graph *directed arc consistent*
   – start w/ the tail and make each variable arc consistent wrt its parents

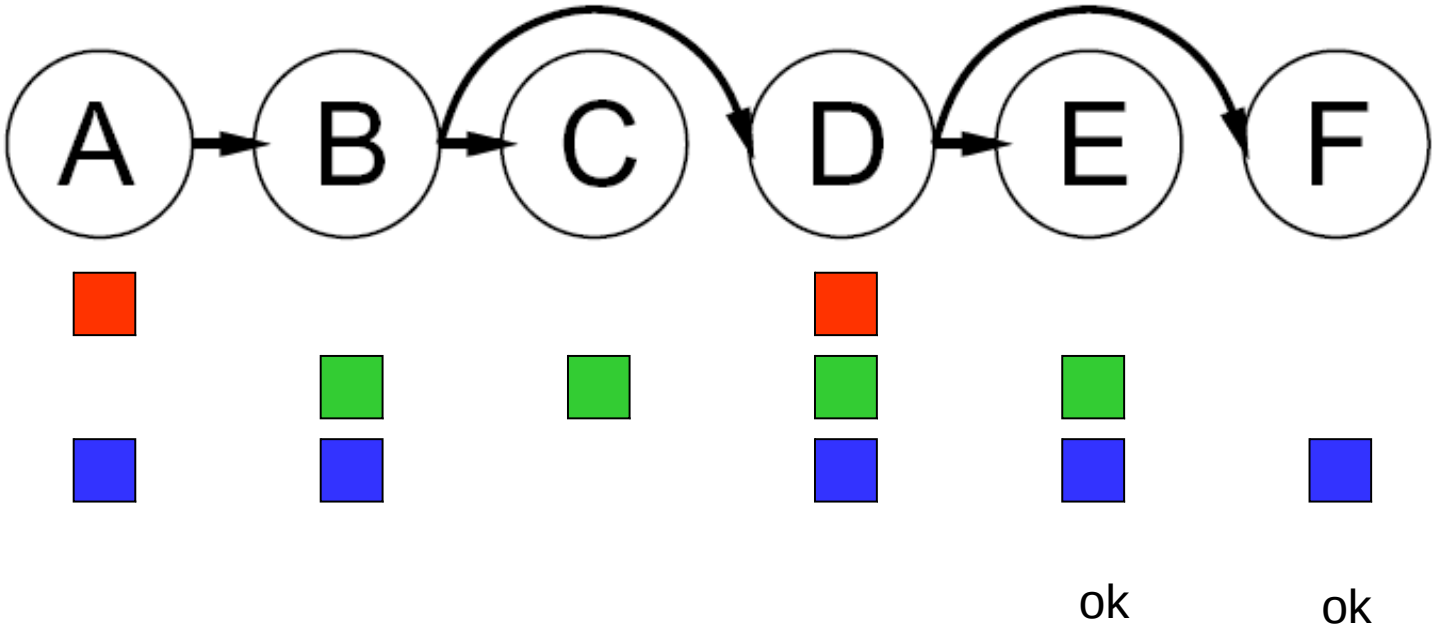


Image: Berkeley CS188 course notes (downloaded Summer 2015)

When the constraint graph is a tree

2. make the graph *directed arc consistent*
   – start w/ the tail and make each variable arc consistent wrt its parents

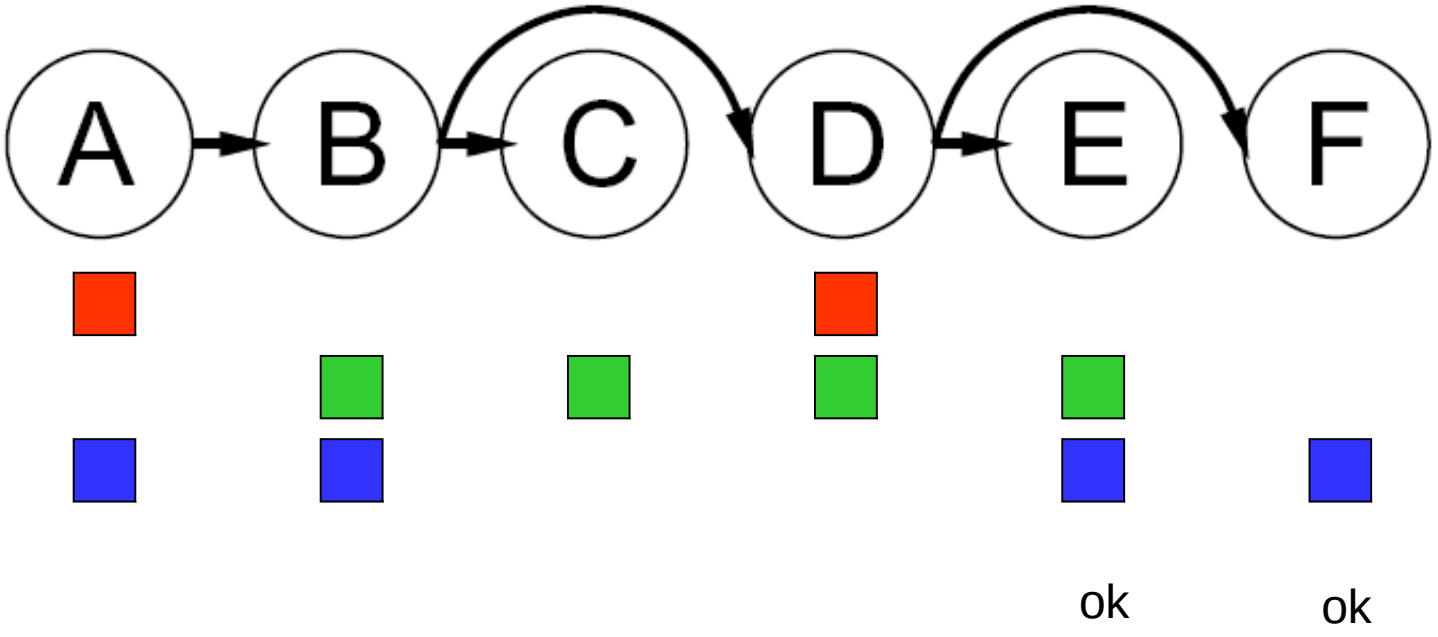


Image: Berkeley CS188 course notes (downloaded Summer 2015)

When the constraint graph is a tree

2. make the graph *directed arc consistent*
   – start w/ the tail and make each variable arc consistent wrt its parents

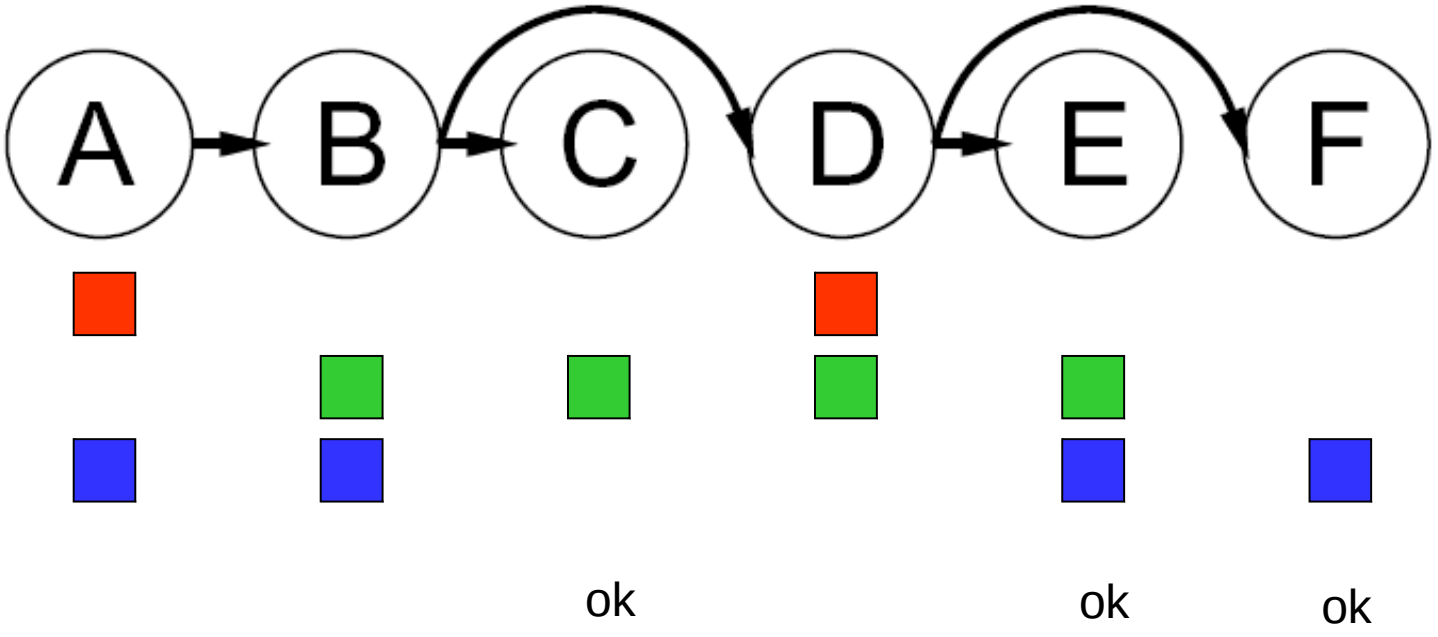


Image: Berkeley CS188 course notes (downloaded Summer 2015)

When the constraint graph is a tree

2. make the graph *directed arc consistent*
   – start w/ the tail and make each variable arc consistent wrt its parents

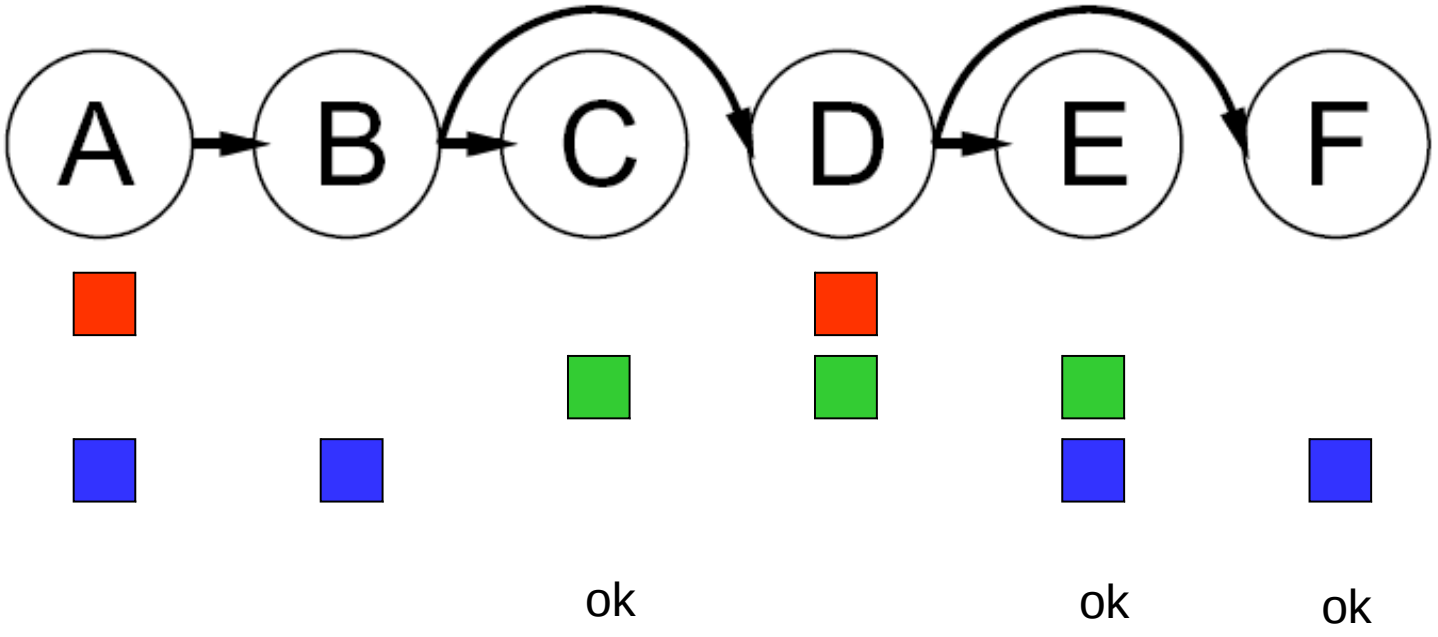
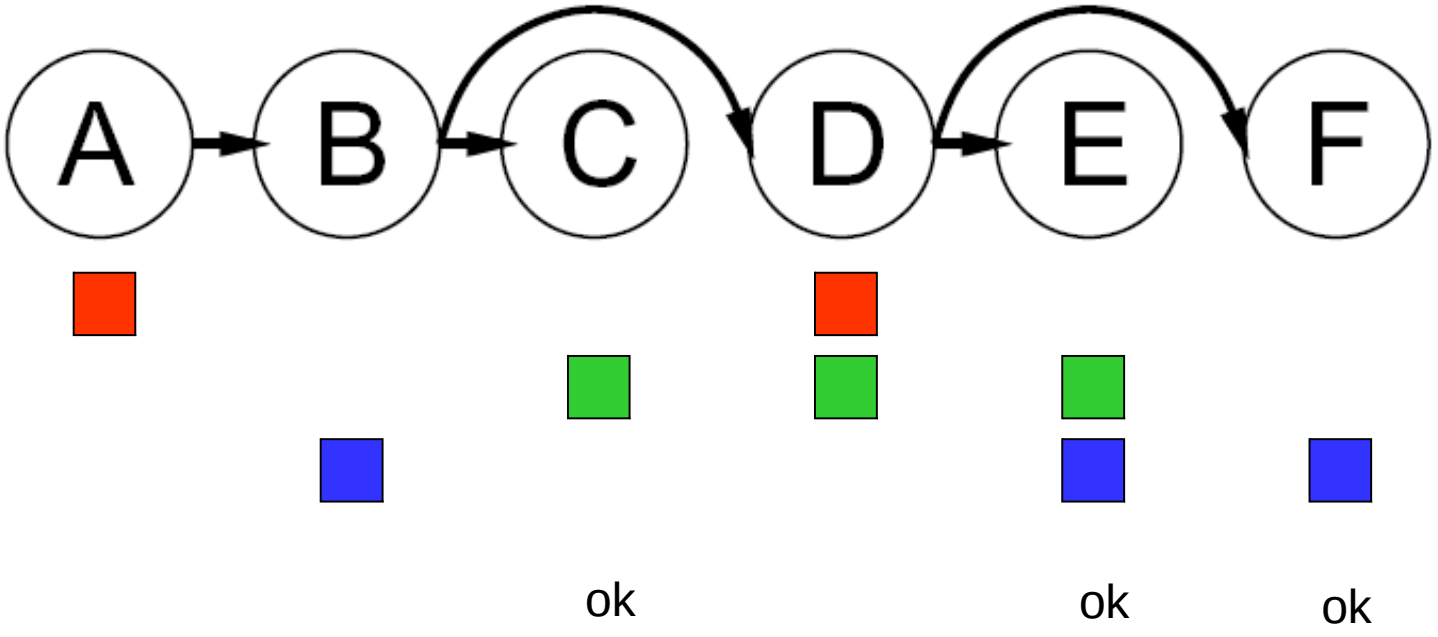


Image: Berkeley CS188 course notes (downloaded Summer 2015)

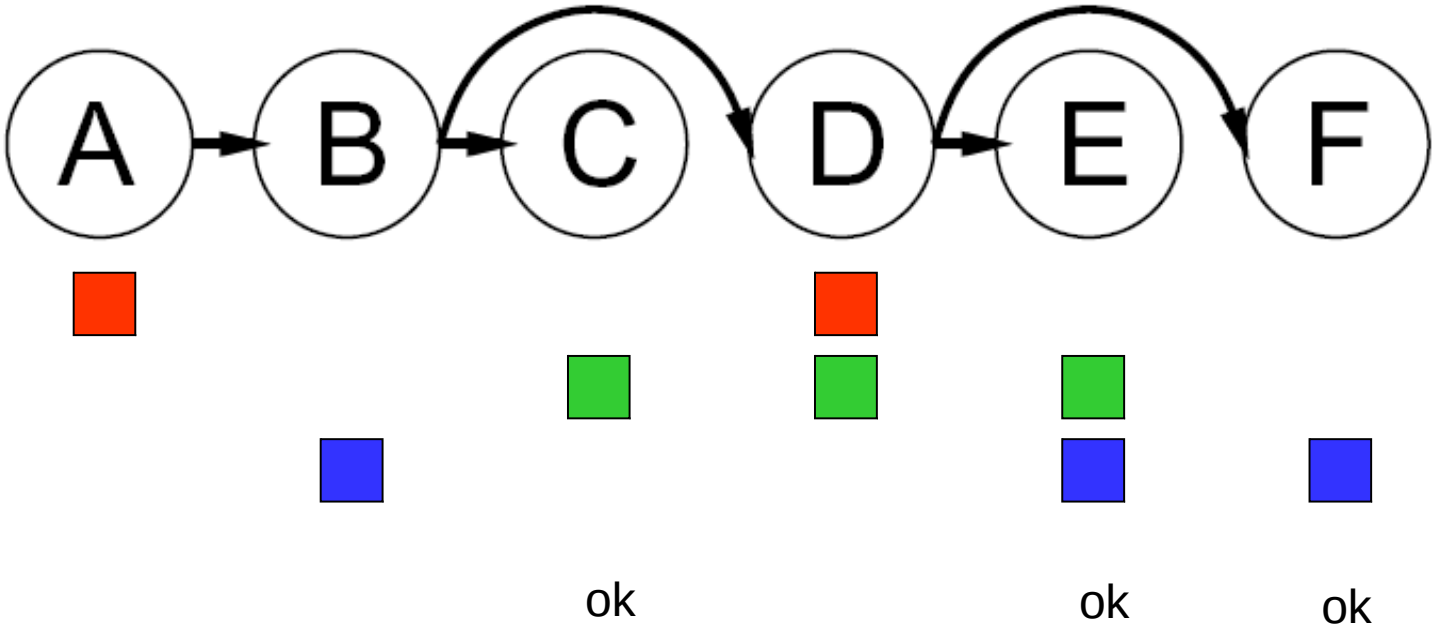
When the constraint graph is a tree

2. make the graph *directed arc consistent*
   – start w/ the tail and make each variable arc consistent wrt its parents



When the constraint graph is a tree

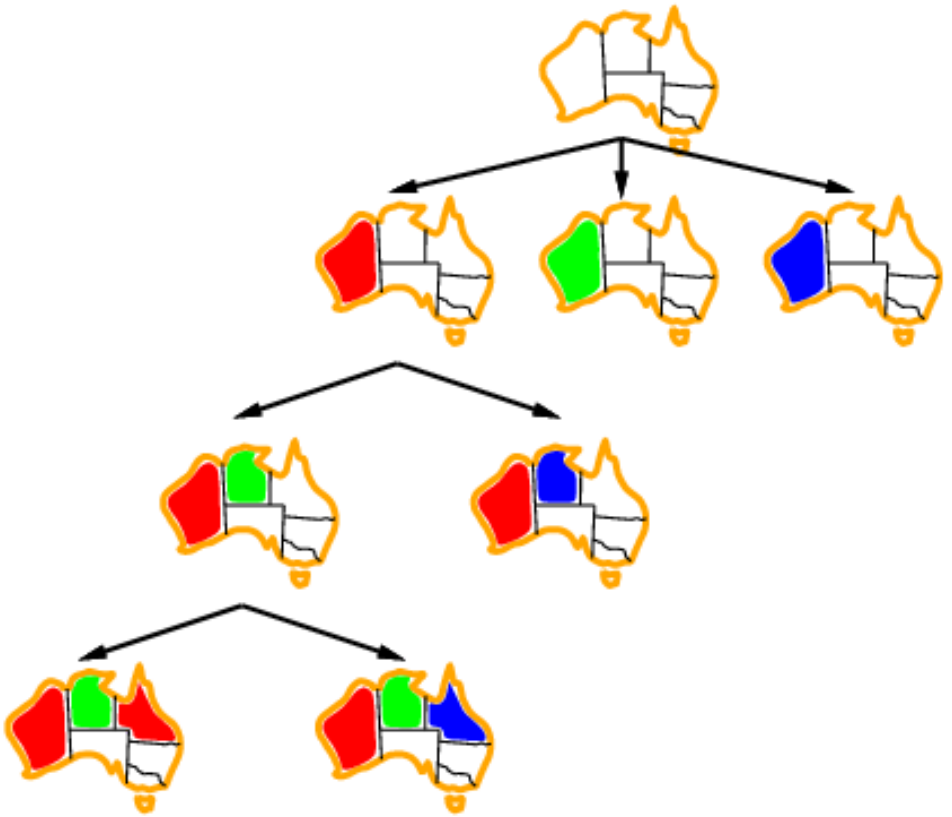
3. Now, start at the root and do backtracking
   – will backtracking ever actually backtrack?



So, what's the time complexity of this algorithm?

Using structure to reduce problem complexity

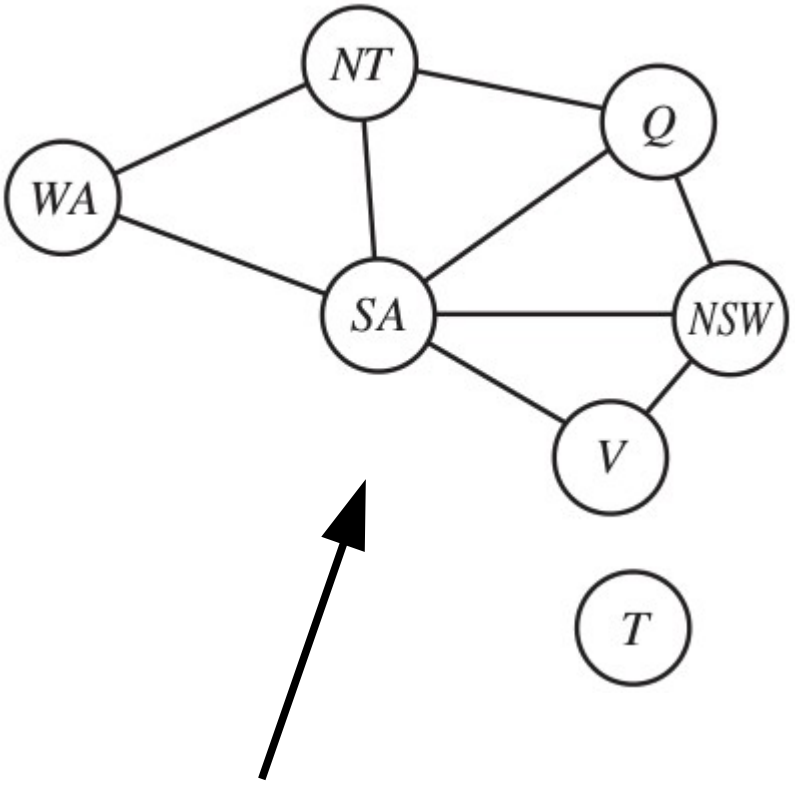
But, what if the constraint graph is not a tree?
– is there anything we can do?



But, sometimes CSPs have special structure that makes them simpler!

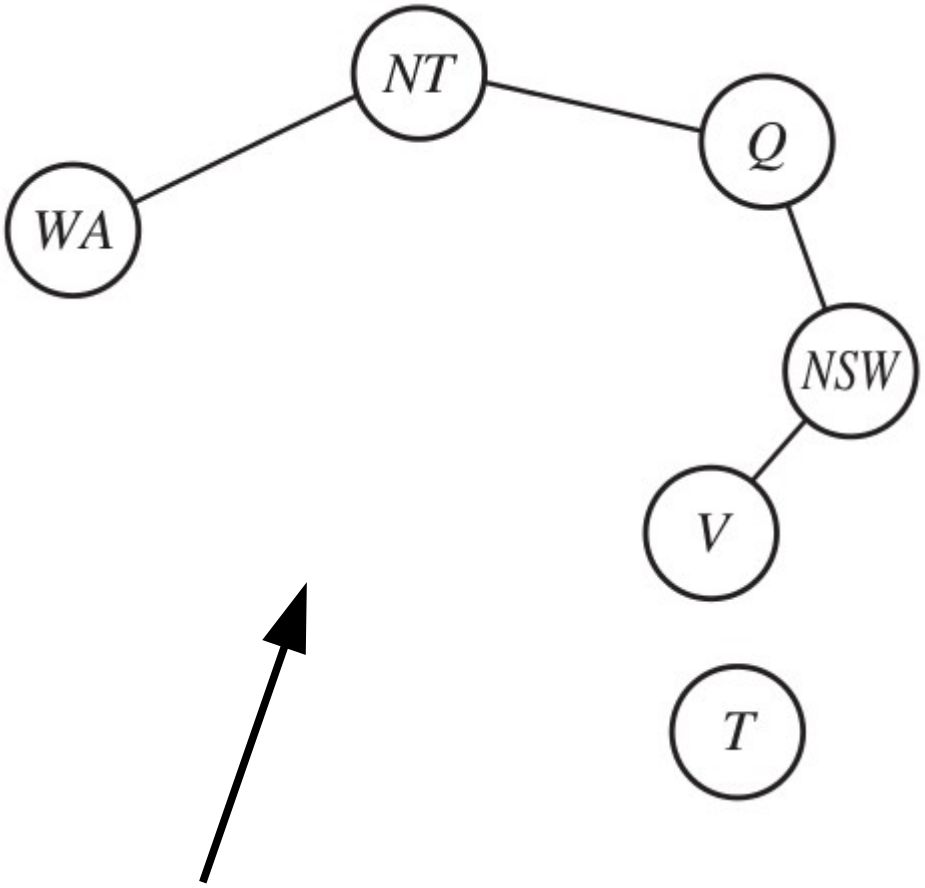
Using structure to reduce problem complexity

But, what if the constraint graph is not a tree?
– is there anything we can do?



This is not a tree...

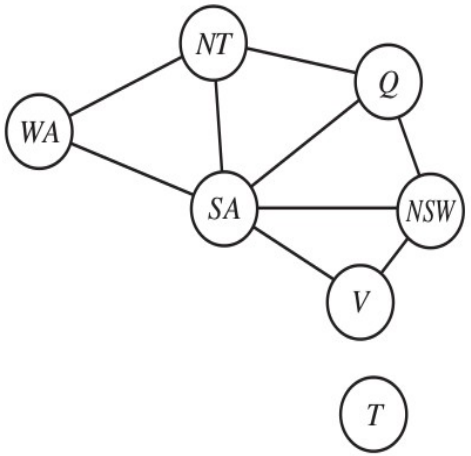
Cutset conditioning



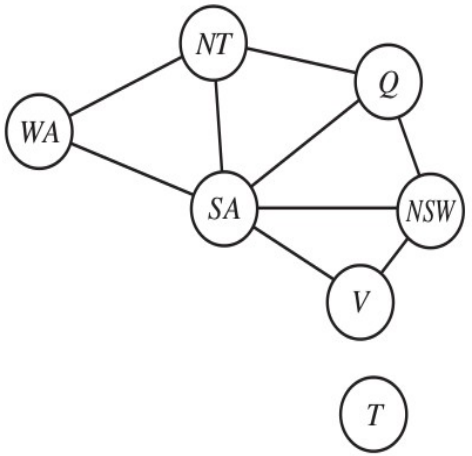
This is a tree...

Cutset conditioning

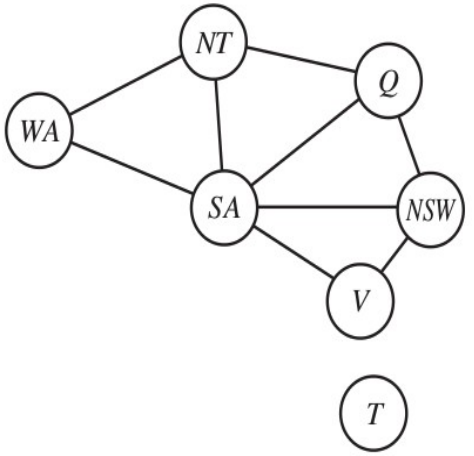
Solve three versions of the “tree-version” of the problem
– one for SA=RED, SA=BLUE, SA=GREEN



SA=RED



SA=BLUE



SA=GREEN

If a solution exists, then you'll find it this way!
– less complex than general version of problem