Intelligent Systems Programming

Lecture 8: **Constraint Programming**

IT University of Copenhagen

Today's Program

- [10:00-10:50]
  - Constraint Satisfaction Problems (CSPs)
  - Constraint Propagation
  - Backtracking
- [11:00-12:00]
  - Variable and value selection
  - Forward Checking
  - Maintaining arc consistency (MAC) algorithm
  - Global constraints
  - Modern constraint propagation systems

Constraint Satisfaction Problems (CSPs)

	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		

Container Vessel Slot Planning

Definition of CSPs

A CSP is a triple $<X,D,C>$, where:

$X = \{X_1,\ldots,X_n\}$ is a finite set of variables.

$D = \{D_1,\ldots,D_n\}$ is a set of domains of possible values for each variable, where $D_i = \{v_1,\ldots,v_{ki}\}$

$C = \{C_1,\ldots,C_m\}$ is a set of constraints, where $C_i = <\text{scope, relation}>$

e.g. $X_1 \in \{A,B\}$, $X_2 \in \{A,B\}$

Implicit constraint representation: $<(X_1,X_2), X_1 \neq X_2>$

Explicit constraint representation: $<(X_1,X_2), [(A,B), (B,A)]>$

Typical short notation: $X_1 \neq X_2$

CSP Solutions

- Partial Assignment : Values assigned to only some of the variables.
- Complete Assignment : Each variable has a value assigned.
- Consistent Assignment : Each constraint, where all variables in its scope are assigned, is satisfied.
- Solution : Complete consistent assignment.

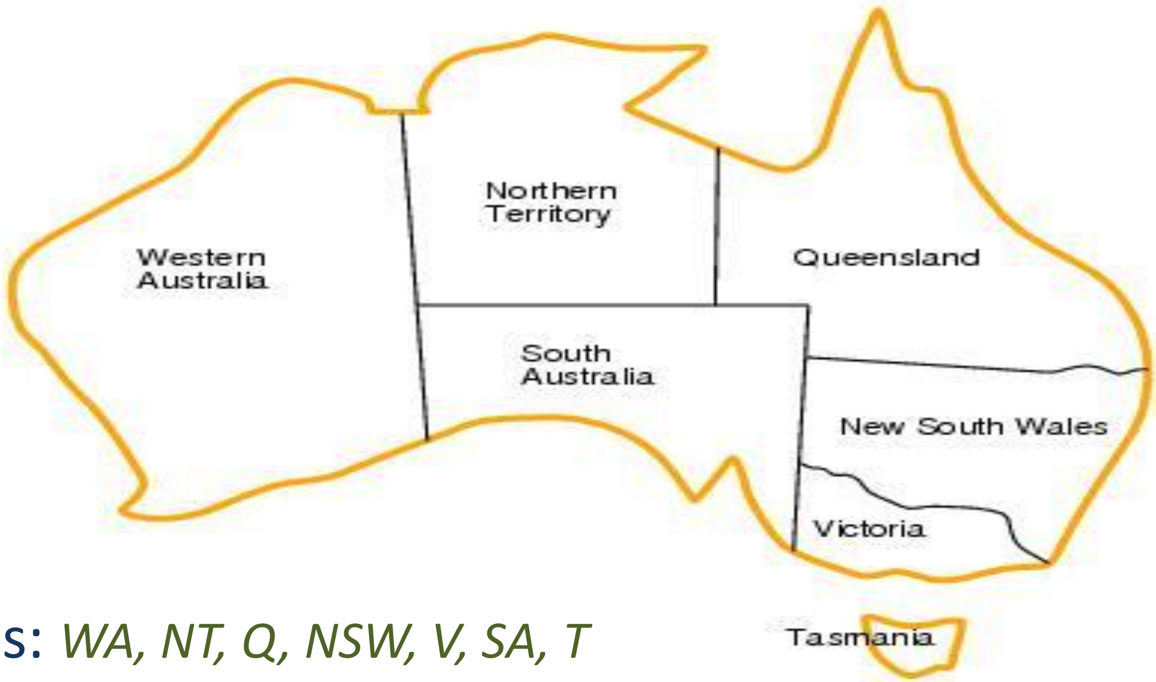
Types of Constraints

Different Arity

- Unary constraints involve a single variable.
  - e.g. $X \neq 12$
- Binary constraints involve pairs of variables.
  - e.g. $X > Y$, $P = MIB \wedge C = Black$
- Global constraints involve arbitrary number of variables.
  - e.g. *AllDifferent*

This lecture: all* algorithms assume binary constraints

CSP Example: Map Coloring

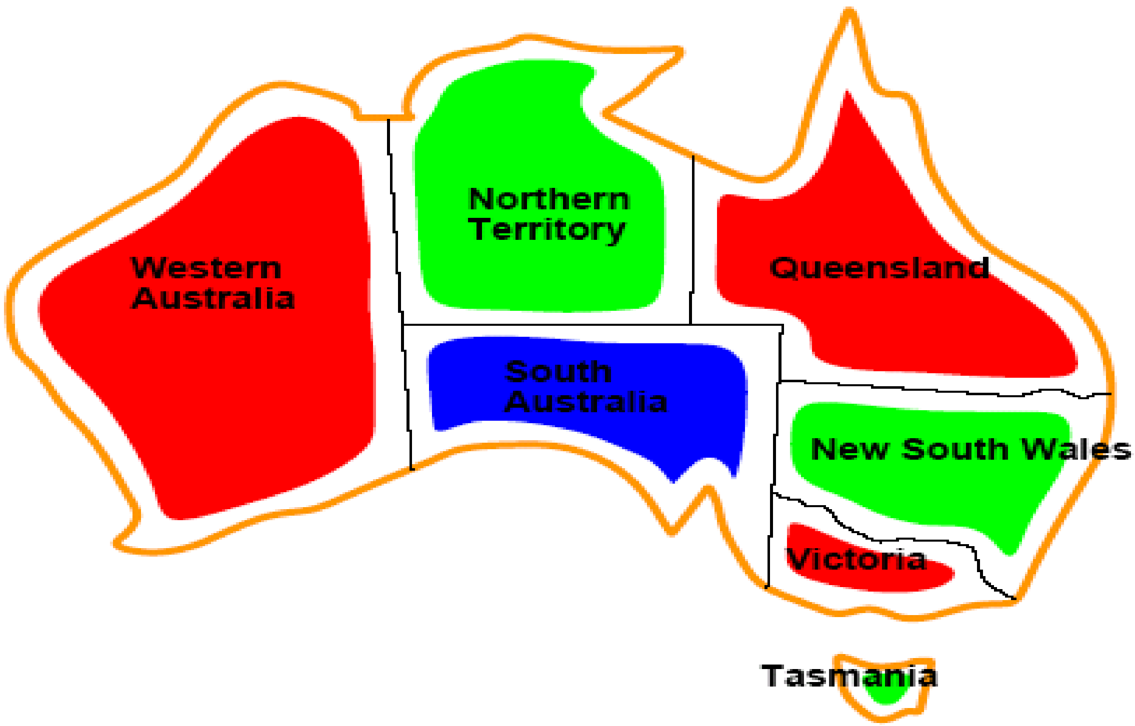


- Variables: *WA, NT, Q, NSW, V, SA, T*
- Domains: $D_i = \{red, green, blue\}$
- Constraints: adjacent regions must have different colors.

e.g. <(*WA*,*NT*), $WA \neq NT$>

<(*WA*,*NT*) , [(*red, green*), (*red, blue*), (*green, red*), ...]>

CSP Example: Map Coloring

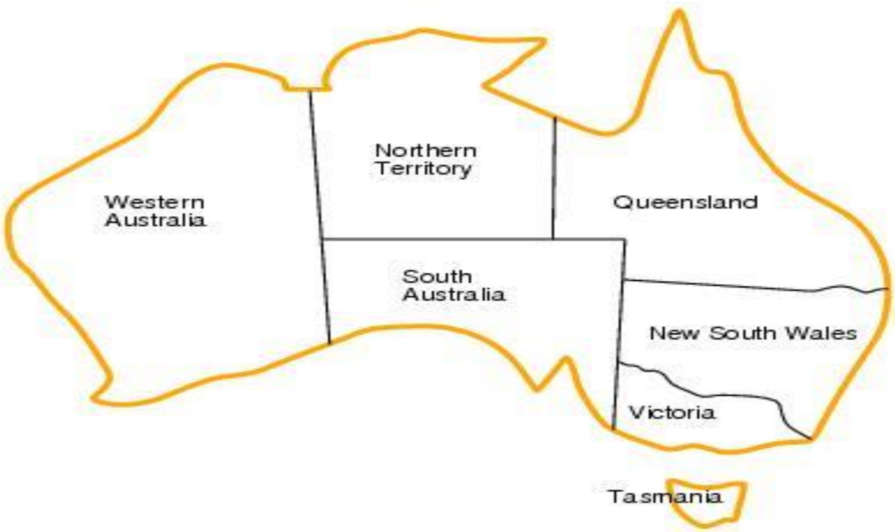


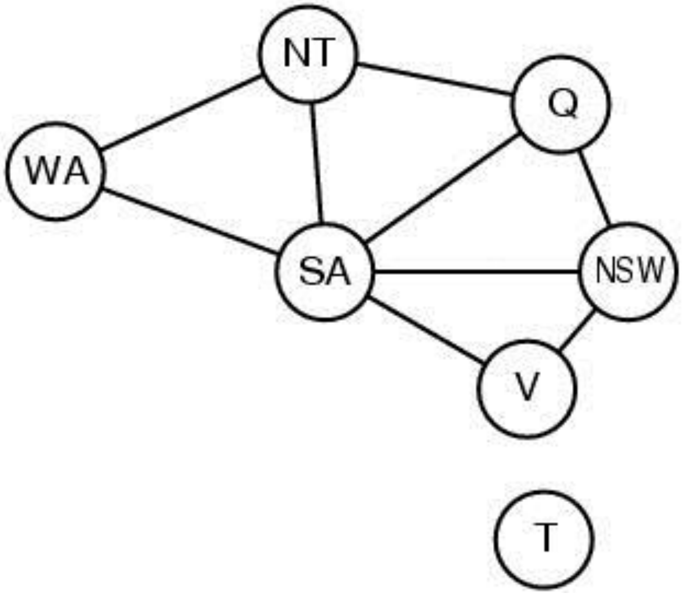
- Solutions are assignments satisfying all constraints, e.g.

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

Constraint Graph Representation of CSP

- Nodes are variables
- Edges are binary constraints





Constraint Propagation (Rule Inference)

CSPs are solved combining search and constraint propagation

Constraint Propagation

- Node consistency : for every value v_i of variable X, all unary constraints of X are satisfied.

Example

$SA \in \{red, green, blue\}, SA \neq \{green\}$

$SA \in \{red, blue\}$

- Arc consistency: for every $X \rightarrow Y$ arc, every value v_i in X has a support value u_j in Y.

Arc Consistency Examples

1) $SA \in \{red, green, blue\}$, $NT \in \{blue\}$, $SA \neq NT$

$SA \rightarrow NT$ arc-consistent: $SA \in \{red, green\}$

2) $X,Y \in \{0,1,\ldots,9\}$, $Y=X^2$

$X \rightarrow Y$ arc-consistent : $X \in \{0,1,2,3\}$, $Y \in \{0,1,\ldots,9\}$

$Y \rightarrow X$ arc-consistent: $X \in \{0,1,\ldots,9\}$, $Y \in \{0,1,4,9\}$

3) $SA \in \{red, green, blue\}$, $WA \in \{red, green, blue\}$, $SA \neq WA$

$SA \rightarrow WA$ / $WA \rightarrow SA$ arc-consistent: can we prune any values?

Arc Consistency Algorithm AC-3

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

Obs: two arcs for each binary constraint!

  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 $revised$

Complexity of AC-3

Assume

- n variables,
- at most d values in domains
- c binary constraints

Observations

- An arc can at most be added to Q d times
- An arc can be revised in d^2
- Thus, worst case runtime is $O(cd^3)$

CSP Solving

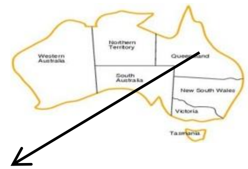
Search in CSP

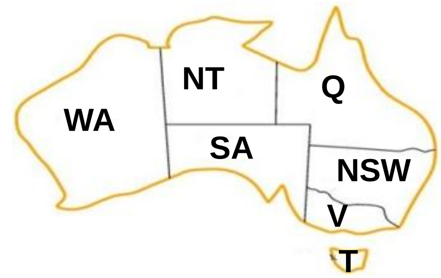
- Inference is not enough
- Apply depth-first search:
  - State: Partial assignment
  - Action: *var* = *value*
- **Complexity**
  - Branching factor b at the top level is $\boldsymbol{nd}$
  - $b = (n-l)d$ at depth l, hence $n!d^n$ leaves
  - But only d^n complete assignments?!

Backtracking Search

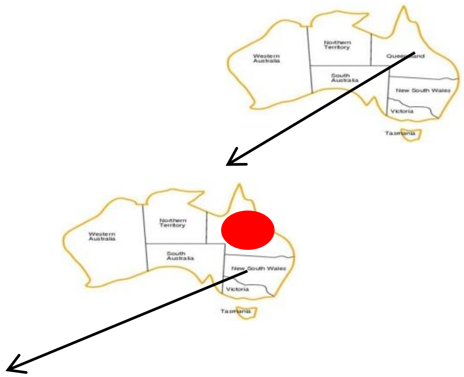
- **Insight**:
  - If we assign the first *k* variables to values, it does not matter in what order we did it in.
  - Thus, after choosing which variable to assign in a node, do not change it.
- **Backtracking Algorithm:**
  - Choose values for one variable at the time.
  - Backtrack when a variable has no legal values left.

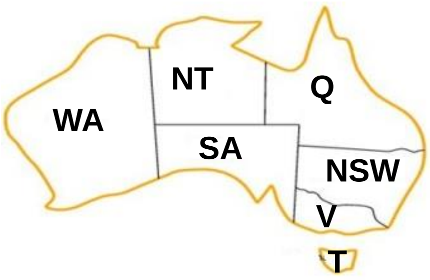
Backtracking Example



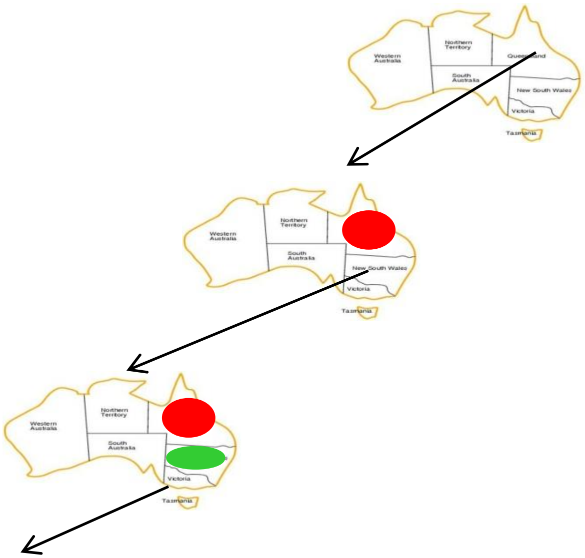


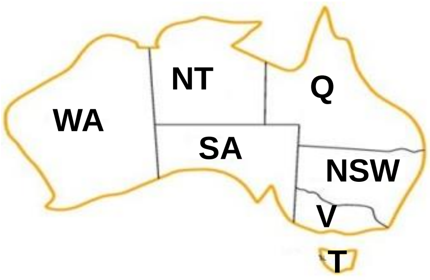
Backtracking Example



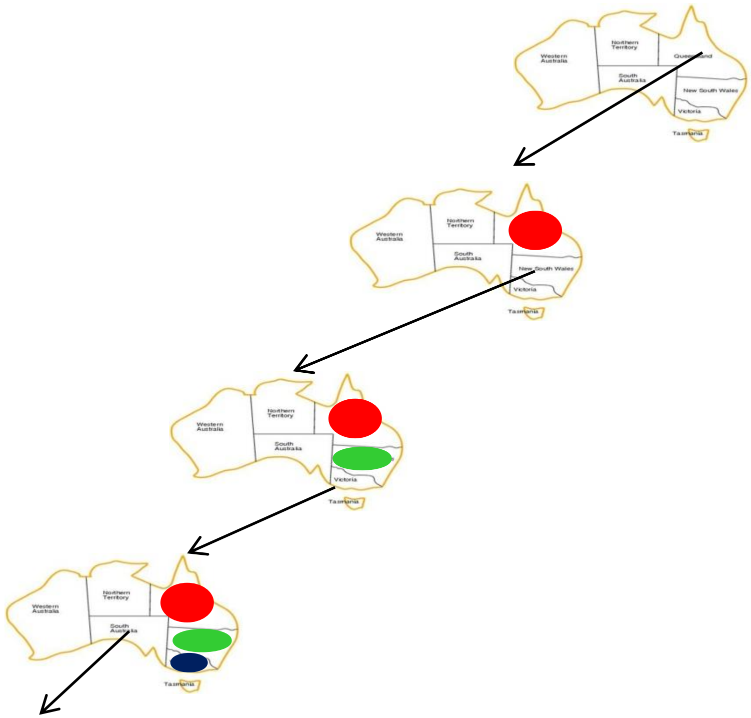


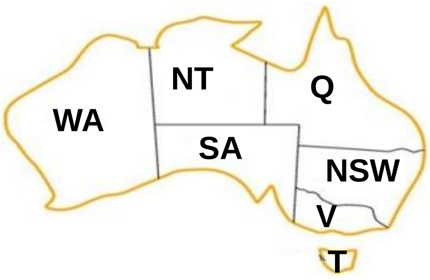
Backtracking Example



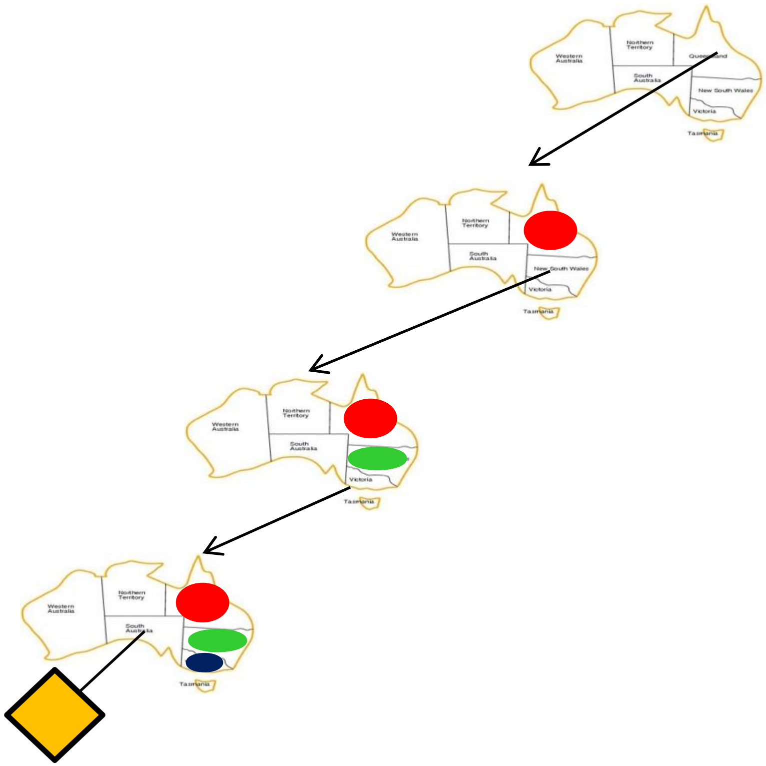


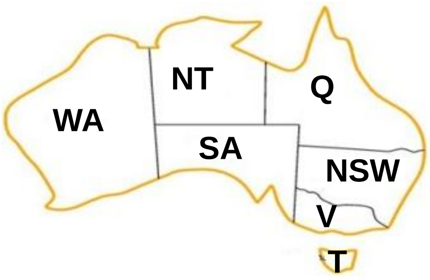
Backtracking Example



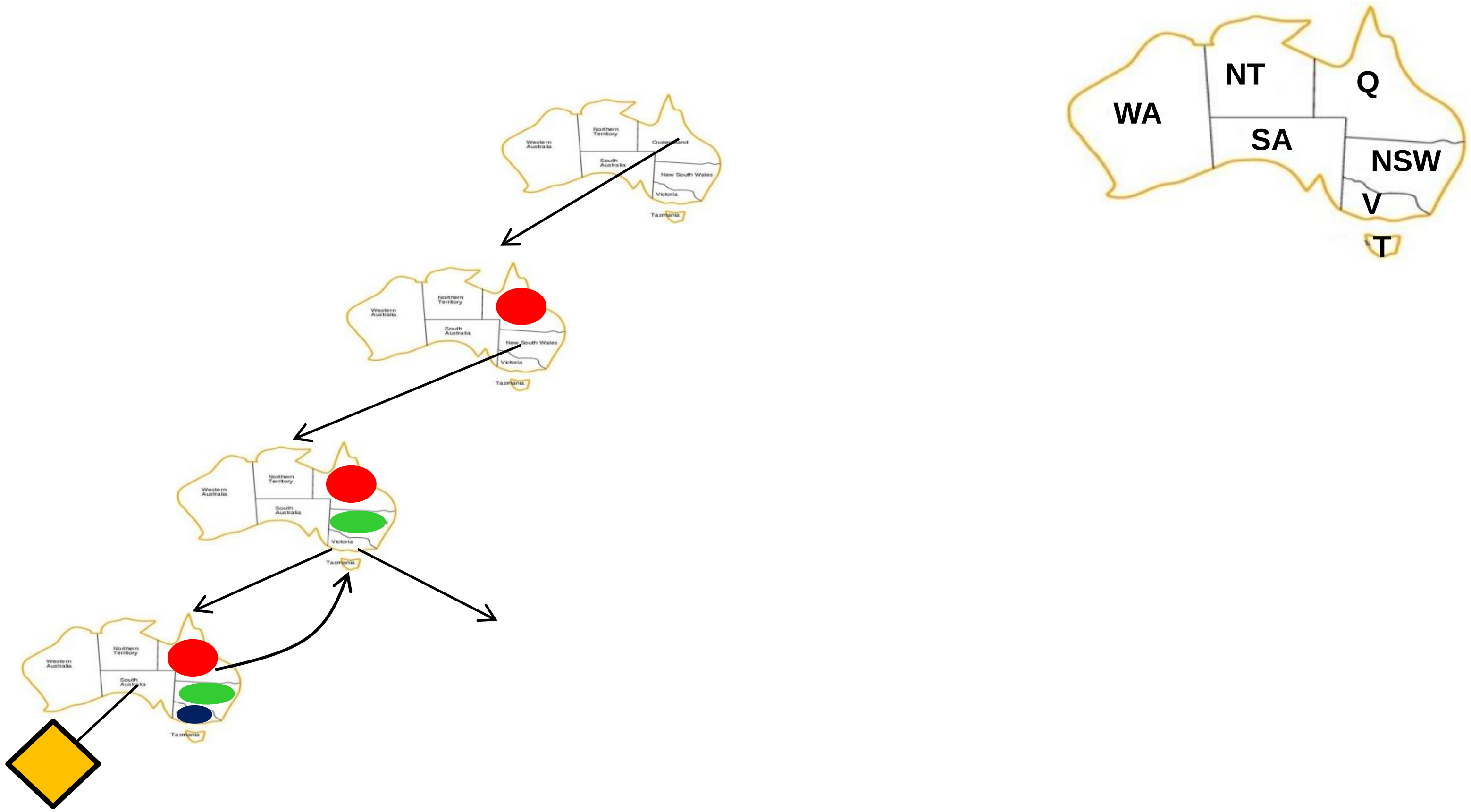


Backtracking Example

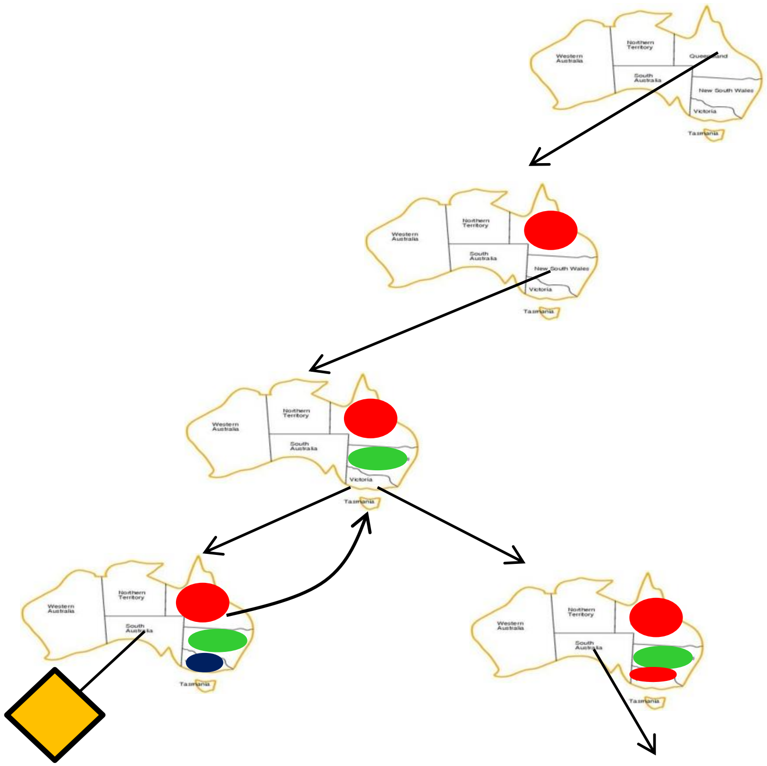


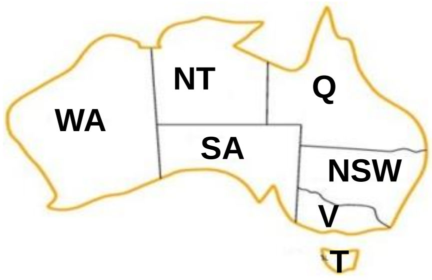


Backtracking Example

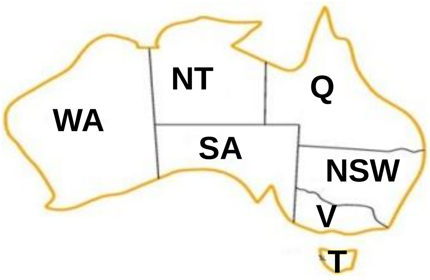


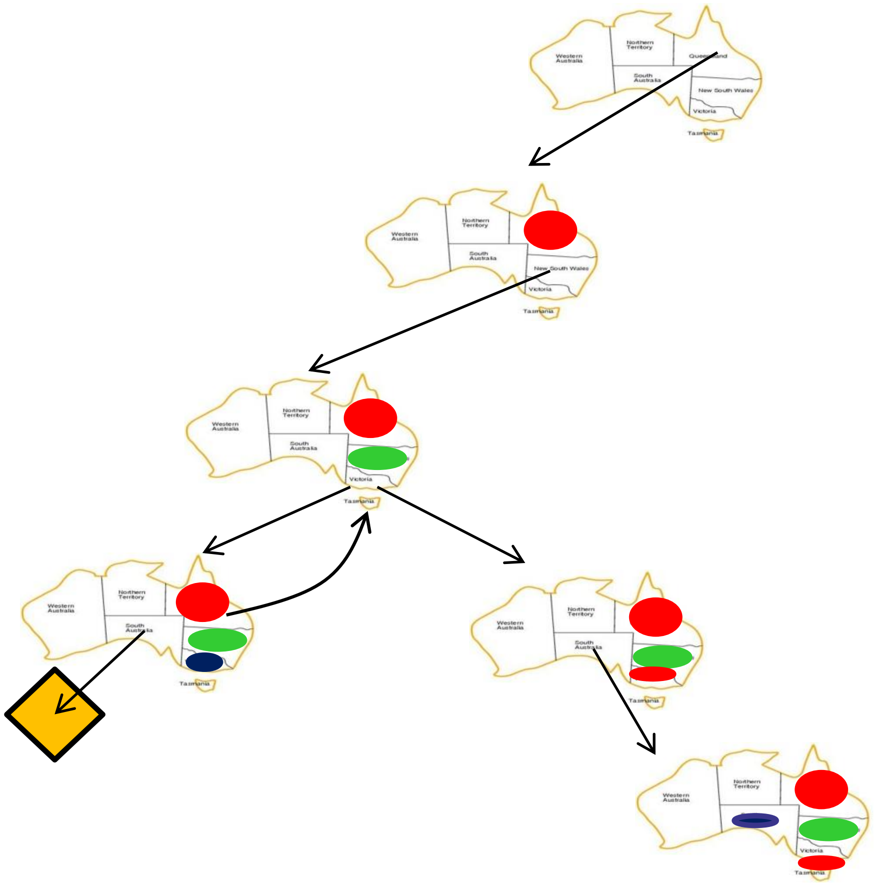
Backtracking Example





Backtracking Example





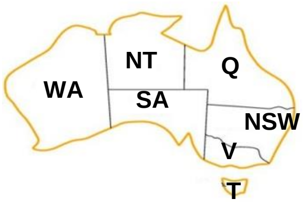
. . .

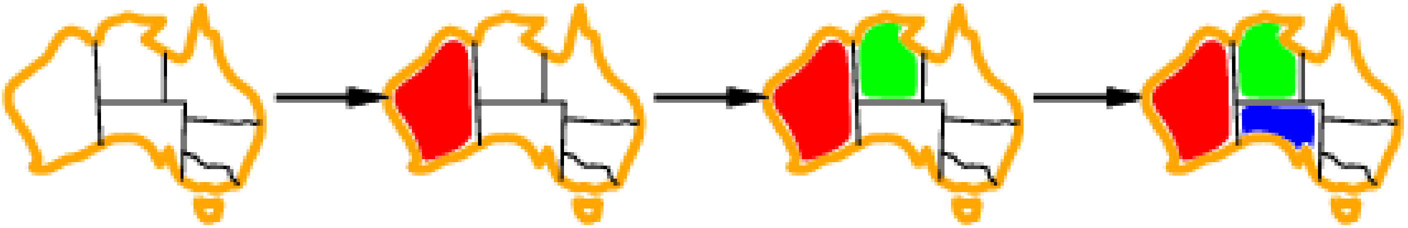
Backtracking Algorithm

```
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
```

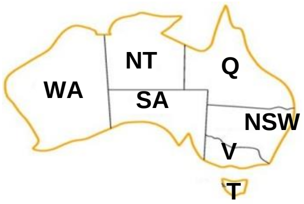
Select-Unassigned-Variable

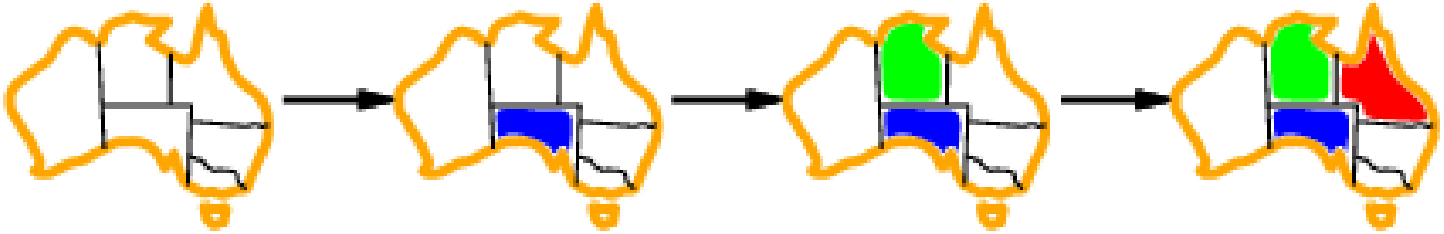




- Minimum remaining values (MRV)
- Rule: choose variable with the **fewest legal values**

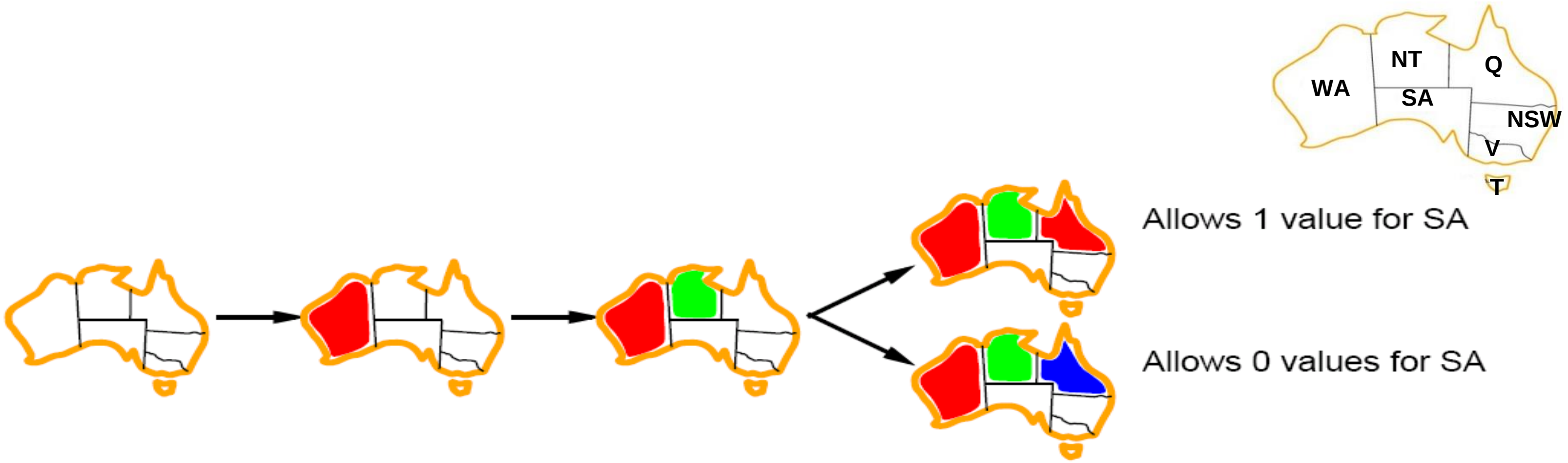
Select-Unassigned-Variable





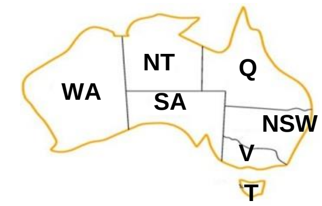
- Degree heuristic
- Rule: select variable that is involved in the largest number of constraints on other unassigned variables.
- Degree heuristic is very useful as a tie breaker

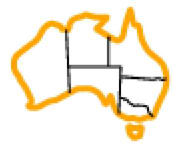
Order-Domain-Values

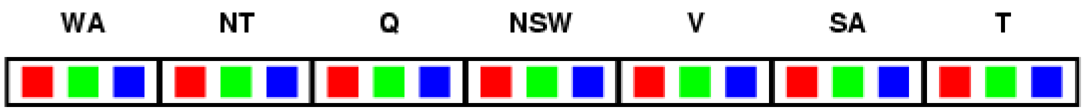


- Least constraining value heuristic
- Rule: given a variable choose the least constraining value i.e., the one that leaves the maximum flexibility for subsequent variable assignments.

Forward Checking

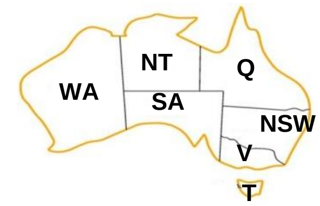


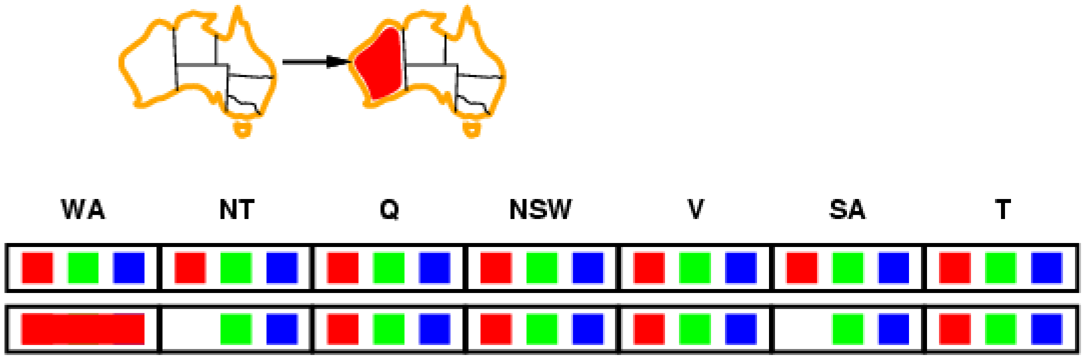




- Forward checking: Whenever a value v is assigned to a variable X_i, make all variables **consistent** with **this** assignment.
- Terminates search when any variable has no legal values.

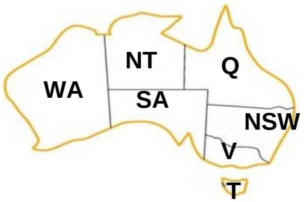
Forward Checking

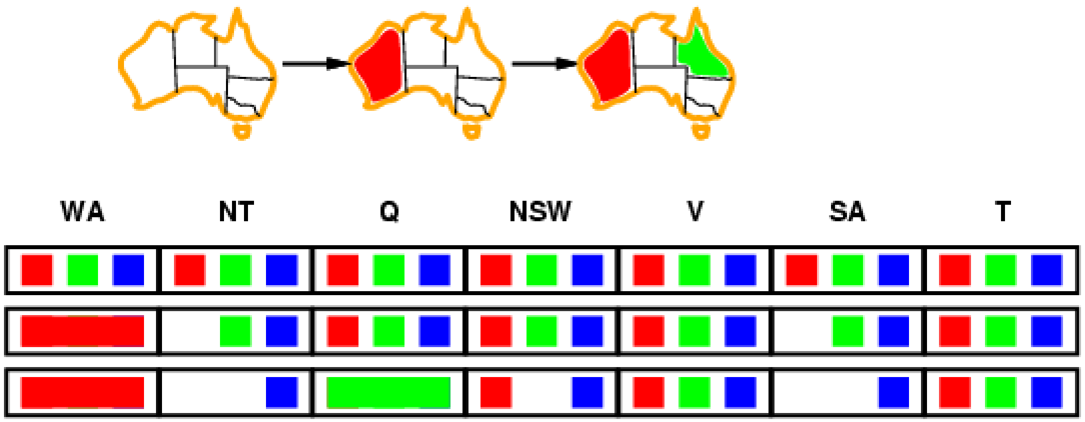




- Forward checking: Whenever a value *v* is assigned to a variable X_i, make all variables **consistent** with **this** assignment.
- Terminates search when any variable has no legal values.

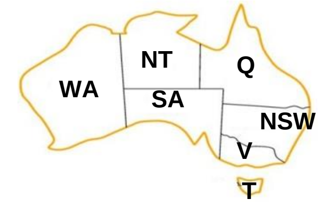
Forward Checking

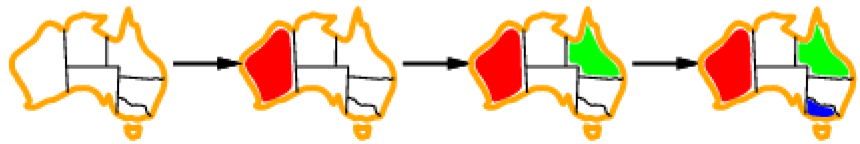


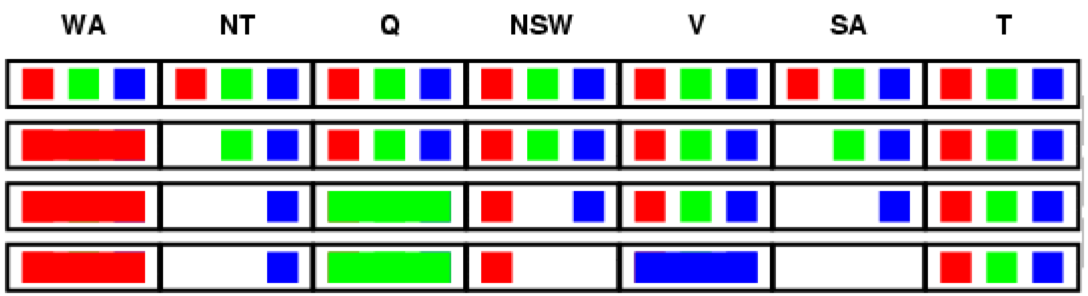


- Forward checking: Whenever a value *v* is assigned to a variable X_i, make all variables **consistent** with **this** assignment.
- Terminates search when any variable has no legal values.

Forward Checking







- Forward checking: Whenever a value v is assigned to a variable X_i, make all variables **consistent** with **this** assignment.
- Terminates search when any variable has no legal values.

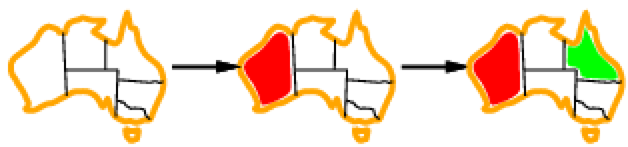
Forward Checking Algorithm

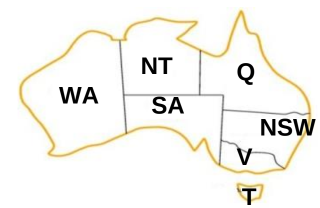
```
function  FORWARD-CHECKING-SEARCH(csp) returns a solution or failure
  return RECURSIVE-FORWARD-CHECKING({},csp)

function RECURSIVE-FORWARD-CHECKING(assignment,csp) returns a solution or
      failure
  if assigment is complete then return assigment
    var ←SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp],assigment,csp)
    for each value in ORDER-DOMAIN-VALUES(var, assigment, csp) do
      if value is consistent with assigment do
        add{var=value} to assigment
        inferences ← remove all domain values of remaining variables
                                        inconsistent with {var = value}
        if inferences≠failure then
          add inferences to assigment and update csp
          result←RECURSIVE-FORWARD-CHECKING(assigment, csp)
          if result≠failure then
            result result
      remove {var=value} and inferences from assigment and csp
  return failure
```

Forward Checking

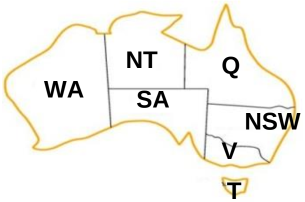
- **Forward checking** propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

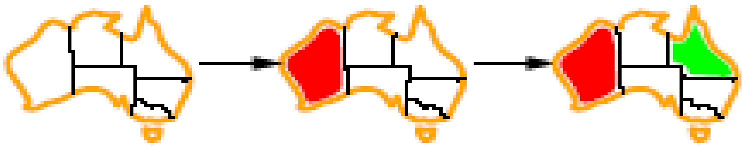


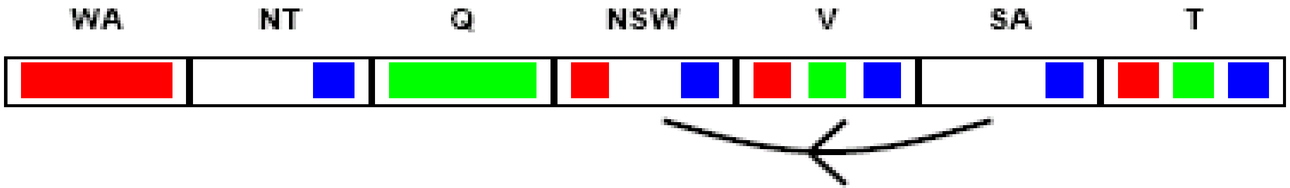


NT and SA cannot both be blue!

Arc Consistency

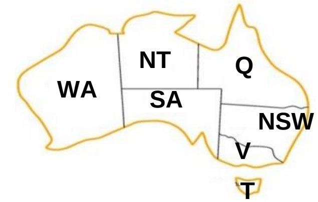


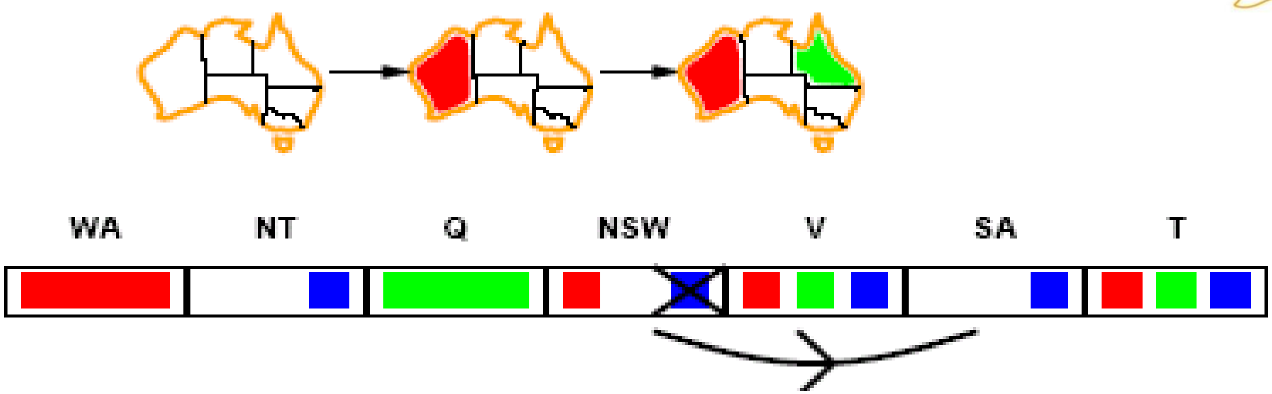




- $X \rightarrow Y$ is consistent iff for every value u_i of X there is some allowed v_j value in Y

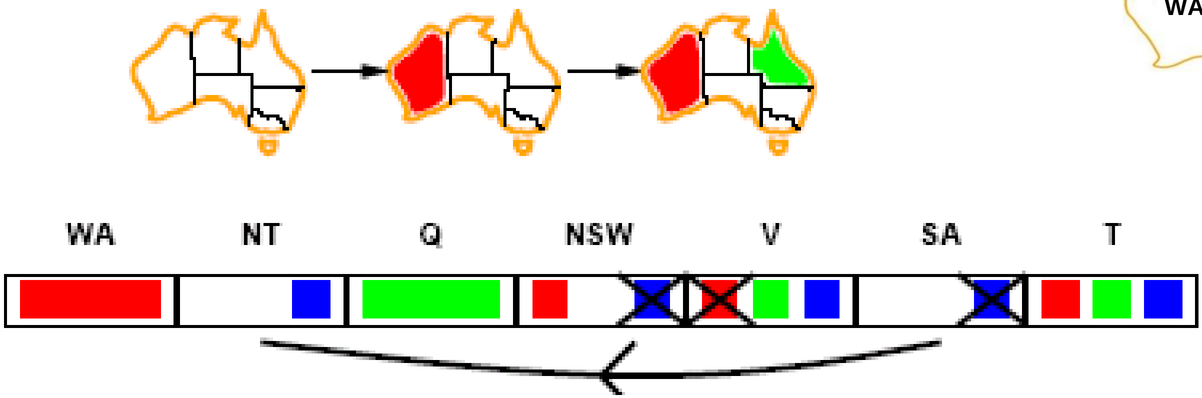
Arc Consistency

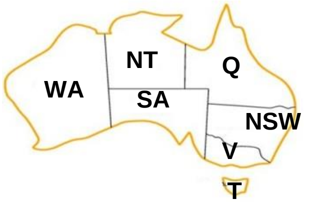




- $X \rightarrow Y$ is consistent iff for every value u_i of X there is some allowed v_j value in Y

Arc Consistency





- $X \rightarrow Y$ is consistent iff for every value u_i of X there is some allowed v_j value in Y.
- Arc consistency detects failure earlier than FC

MAC

```
function  MAC-SEARCH(csp) returns a solution or failure
  run AC-3(csp)
  return RECURSIVE-MAC({},csp)

function RECURSIVE-MAC(assignment,csp) returns a solution or failure
  if assigment is complete then return assigment
    var ←SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp],assigment,csp)
    for each value in ORDER-DOMAIN-VALUES(var, assigment, csp) do
      if value is consisten with assigment do
        add{var=value} to assigment
        inferences ←AC-3(csp)
        if inferences≠failure then
          add inferences to assigment and update csp
          result←RECURSIVE-MAC(assigment, csp)
          if result≠failure then
            result result
          remove {var=value} and inferences from assigment and csp
  return failure
```

Global Constraints

Definition

- Depends on arbitrary number of variables

Generalized Arc Consistency (GAC)

- For each value in domain of X_i, there exists a valid assignment for all the remaining variables in the constraint
- $x,y,z \in \{1,2,3\}$, $[x > y \wedge y = z]$: $\{(3,2,2),(3,1,1),(2,1,1)\}$
  GAC : $x \in \{2,3\}$, $y \in \{1,2\}$, $z \in \{1,2\}$

Alldifferent

- Alldifferent: all $X_i \in$ scope of constraint are assigned to different values.

  e.g. $X_1, \ldots, X_4 \in \{0, 1, 2, 3, 4\}$ ***Alldiff***(X_1, X_2, X_3, X_4)

  $X_1=0, X_2=1, X_3=2, X_4=3$ ✓

  $X_1=0, X_2=1, X_3=1, X_4=2$ ✗

- How to represent Alldifferent with binary ≠ constraints?

Simple approximation to AllDiff GAC

1. If #values < #vars then return failure

   $X_1=0, X_2\in\{1, 2\}, X_3\in\{1, 2\}, X_4=3$ ✓

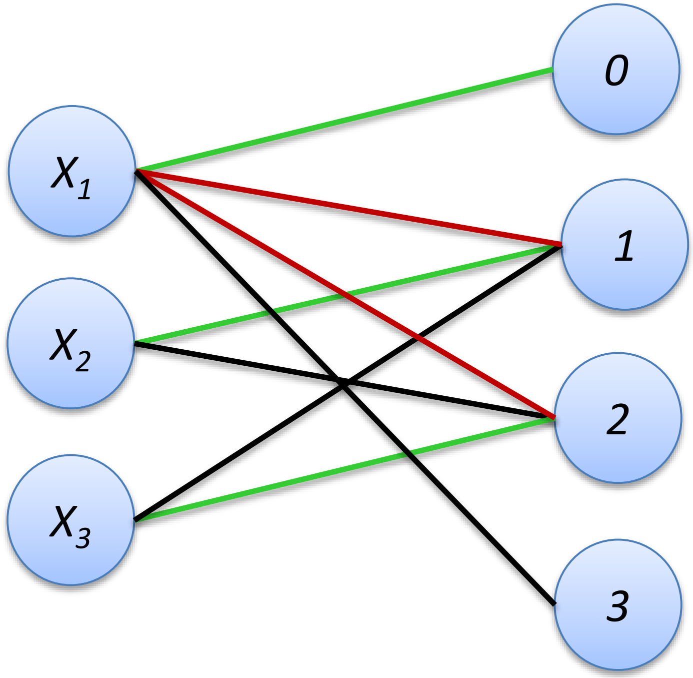
   $X_1\in\{1, 2\}, X_2\in\{1, 2\}, X_3\in\{1, 2\}, X_4=3$ ✗

2. Remove all vars with singleton domains
3. Remove singleton values from remaining domains
4. Goto 1

Fast Computation of AllDiff GAC

– GAC = Find all max matchings in a bipartite graph

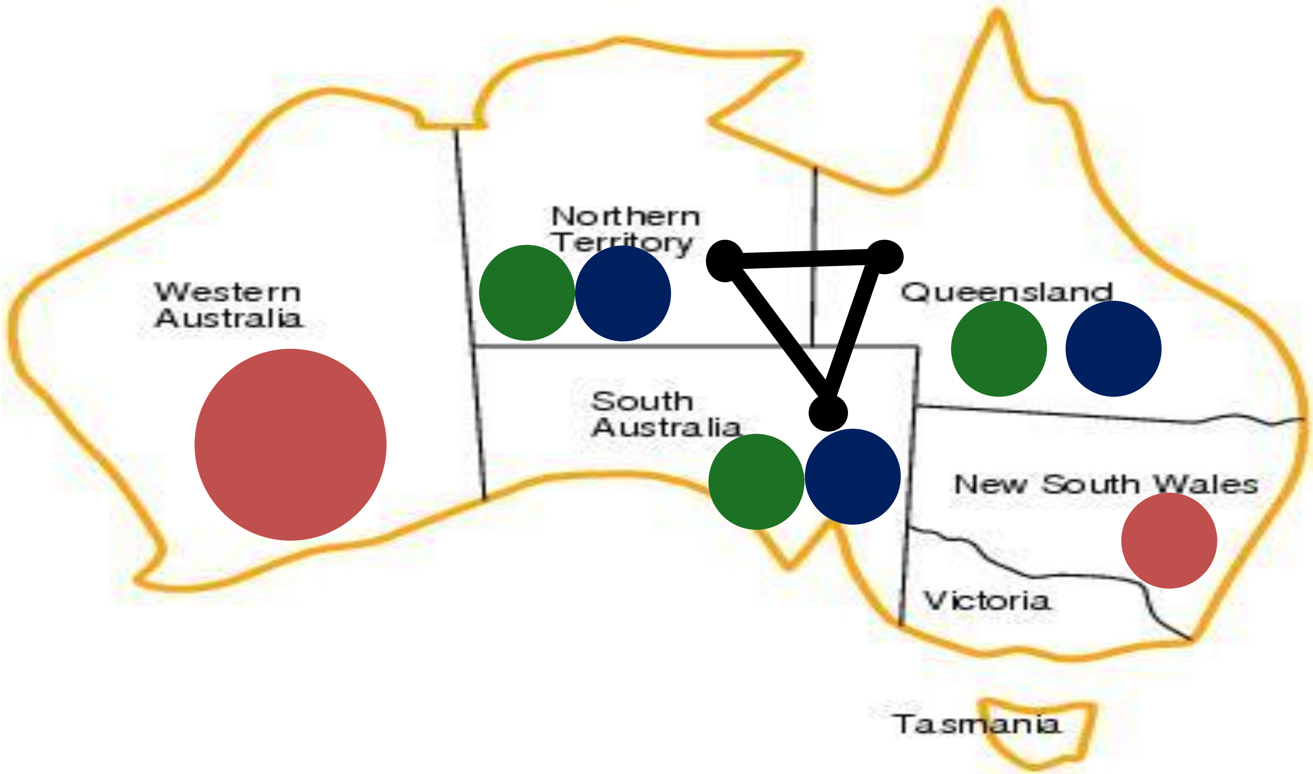
e.g. $X_1 \in \{0, 1, 2, 3\}$, $X_2 \in \{1, 2\}$, $X_3 \in \{1, 2\}$



Can be shown to be possible in $O(d\sqrt{n})$

Where AC-3 on $O(d^2)$ not equal constraints takes $O(d^5)$

AllDiff Stronger than binary constraints



Inconsistency detected by AllDiff GAC, but not by AC-3!

Constraint propagation systems

World's best 08-12 **Gecode** (link on ISP learnit page)

- Constraint Store
- Propagators
- Propagator Loop
- Search

Constraint store

$x \in \{3,4,5\}$ $y \in \{3,4,5\}$

- Maps variables to possible values

Constraint store

finite domains

$x \in \{3,4,5\}$ $y \in \{3,4,5\}$

- Maps variables to possible values
- Others: finite sets, continuous domains, trees, ...

Constraints and Propagators

- Extensive use of global constraints
- Propagators **implement** constraints!
  - prune values in conflict with constraint
- Constraint propagation drives propagators for several constraints

Propagators

$x \geq y$ $y > 3$

$x \in \{3,4,5\}$ $y \in \{3,4,5\}$

- Amplify store by constraint propagation

Propagators

$x \geq y$　　　　$y > 3$

$x \in \{3,4,5\}$　$y \in \{3,4,5\}$

- Amplify store by constraint propagation

Propagators

$x \geq y$ $\quad$ $y > 3$

$x \in \{3,4,5\}$ $\quad$ $\boldsymbol{y \in \{4,5\}}$

- Amplify store by constraint propagation

Propagators

$x \geq y$ $\quad\quad$ $y > 3$

$x \in \{3,4,5\}$ $\;\; \mathbf{y \in \{4,5\}}$

- Amplify store by constraint propagation

Propagators

$x \geq y$ $\qquad$ $y > 3$

$x \in \{4,5\}$ $\quad$ $y \in \{4,5\}$

- Amplify store by constraint propagation
- Disappear when done (subsumed, entailed)
    - no more propagation possible

Propagators

$x \geq y$

$x \in \{4,5\}$ $y \in \{4,5\}$

- Amplify store by constraint propagation
- Disappear when done (subsumed, entailed)
  - no more propagation possible

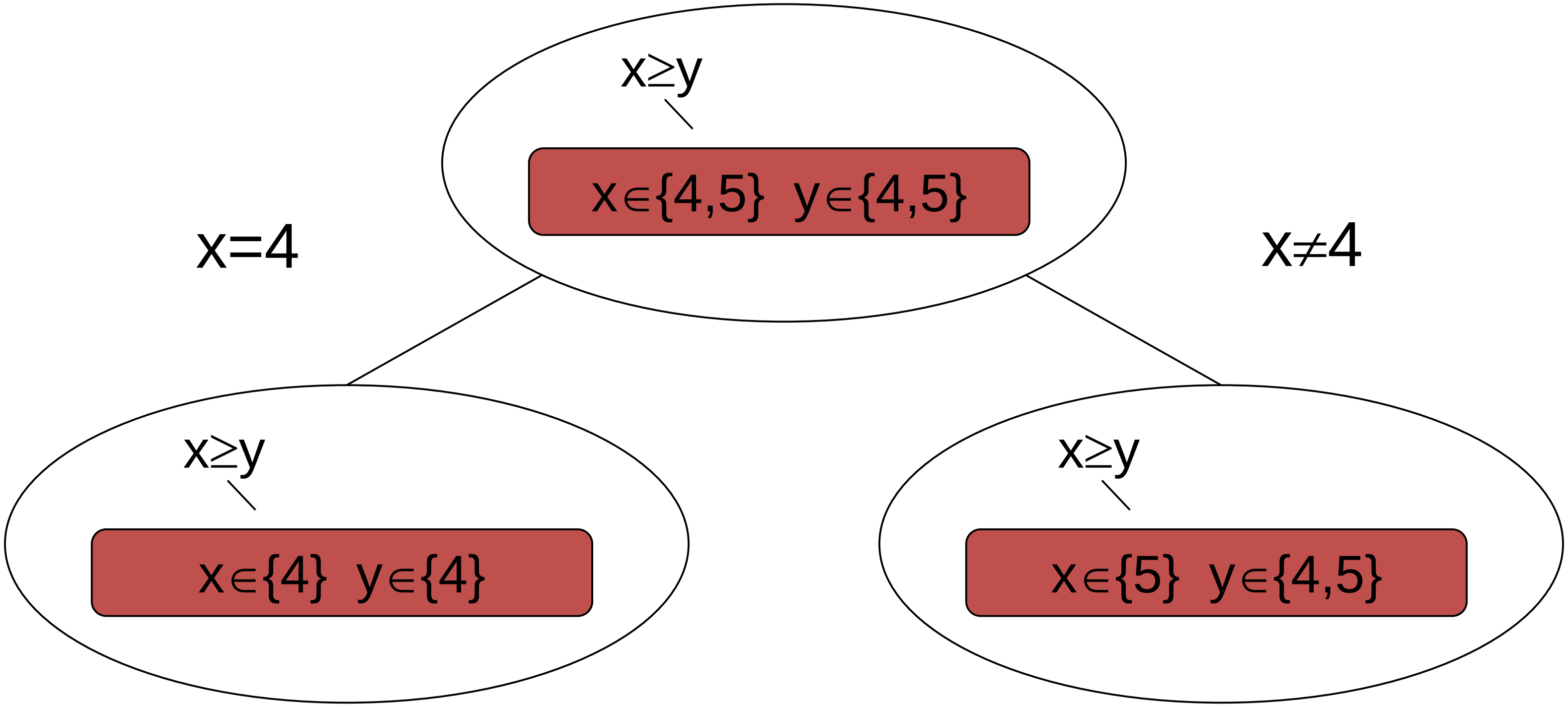
Propagation loop

```
function PROPAGATE(csp)
    Q←{c1, c2, …, cn}                              // Queue of propagators from csp
    while Q≠{} do
       c ←REMOVE-FIRST(Q)
       EXECUTE(c)                    // run propagator,  prune domains of variables
        if any domain in csp is empty then
             return false
       for each Xi ∊ SCOPE(c) s.t. Di was narrowed do
          ADD(GET-PROPS(Xi) - c) to Q               //Add new propagators
       end for
    end while                                          // Fixed point reached!!
  return true
```

Search

- Propagation **alone** is not enough!
- Search: Explore search tree for solutions (Backtracking)
- Branching: Select variable and value (define search tree)
  - Minimum remaining values
  - Degree
  - Least constraining value

Search: Branching



- Create subproblems with additional information
- Enable further constraint propagation

Constraint propagation systems

- Each node in the search tree has different information.
- Constraint store and propagators are encapsulated in a **Space**.
- Manipulate spaces: Copy , add new constraints, ask for solutions, etc.

Search algorithm

function SEARCH(*csp*) **returns** solution or failure
  csp ← PROPAGATE(*csp*)
  if *csp* is *failure* **then return** *failure*
  if *csp* is solved **then return** SOL(*csp*)
  csp' ← *csp*
  ADD(*csp'*, BRANCH(*csp'*,*1*))
  csp'' ← *csp*
  ADD(*csp''*, BRANCH(*csp''*,*2*))
  solution ← SEARCH(*csp'*)
  if *solution* is valid **then return** *solution*
  else return SEARCH(*csp''*)

function BRANCH(*csp*, *nbranch*)
  return the constraint representing the *nbranch* according to variable
    and value heuristic selection.