, 2022

Solving problems by Search

Goal-based Agents

Atomic-Factored-Structured

Dr. Bilal Hoteit

Introduction to Artificial Intelligence

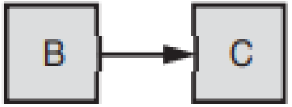
Table of contents

Table of contents

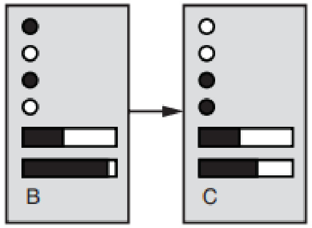
- **Requirements: Linear Programming**
- **Lecture 3 , p:67 - 73**
- **Src3 – production.py**

Difference between Representations

- Atomic representation means a state representing single variable.
- Black box, no internal structure.



- Factored representation means a State consists of a vector of attribute values.
- Each state has several variables and may be each variable can have a set of values.



Constraint Satisfaction problem

- So far, states are evaluated using either heuristics or goals.
- CSP = factored representation of the state.
- A state has several variables
- Each variable has a set of values
- CSP is finding a solution or no solution exists.
- CSP:
  a. X is a set of variables, {X1,...,Xn}.
  b. D is a set of domains, {D1,...,Dn}, one for each variable X.
     - Di = {v1,…vn} for Xi.
  c. C is a set of constraints specifying combinations of values?
     - Ci = {scope, rel}.

Constraint Satisfaction problem

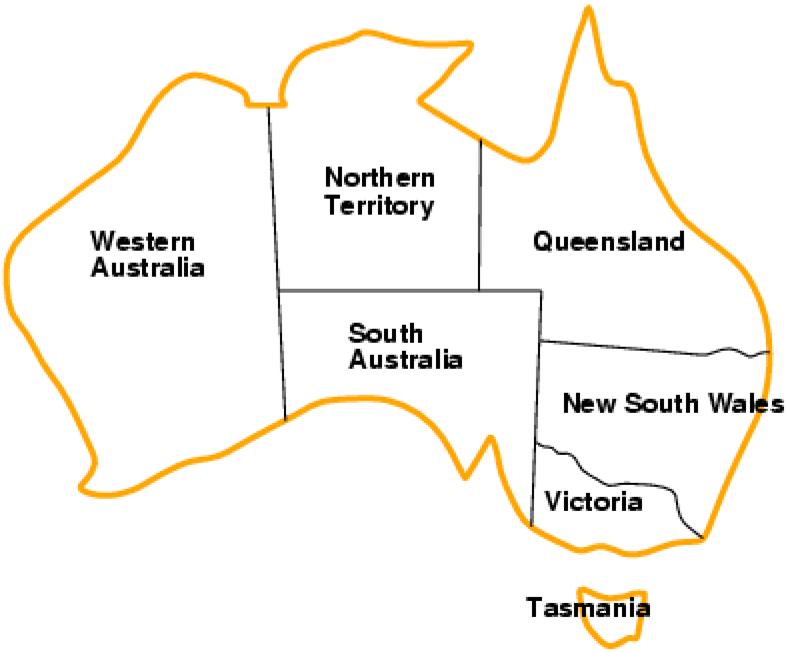
a. Simulation go to word file in chapter seven folder.

Simulation for back tracking algorithm:

Constraint Satisfaction: introduction - ...

Constraint Satisfaction problem formulation

- **Example: Map Austria color**

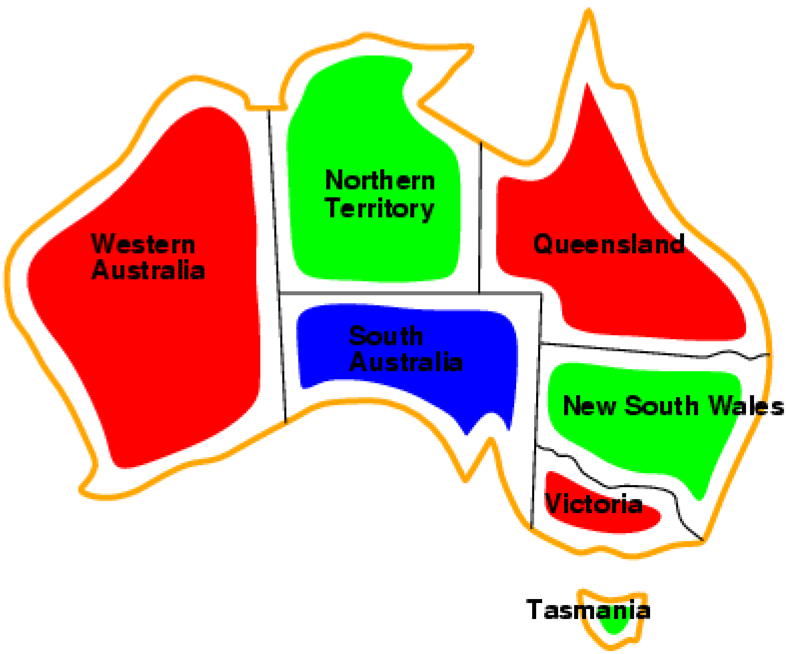


- Variables *WA, NT, Q, NSW, V, SA, T* .
- Domains D_i = {red,green,blue} .
- Constraints: adjacent regions must have different colors .

e.g., WA ≠ NT, or (WA,NT) in { (red,green), (red,blue), (green,red), (green,blue), (blue,red), (blue,green) }

Constraint Satisfaction problem formulation

- **Example: Map Austria color**

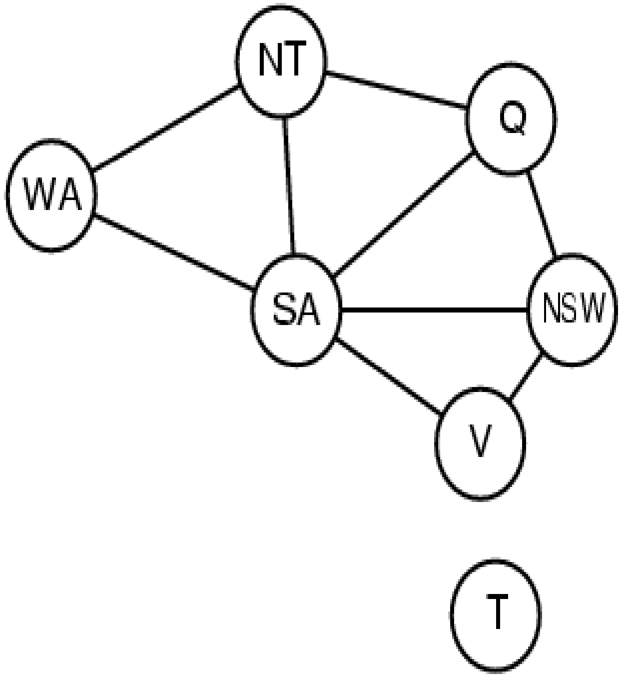


- Variables *WA, NT, Q, NSW, V, SA, T* .
- Domains D_i = {red,green,blue} .
- Constraints: adjacent regions must have different colors .

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

Constraint Satisfaction problem formulation

- **Example: Map Austria color**
  - Constraint graph: nodes are variables, arcs are constraints.



- Unary constraints involve a single variable, e.g., SA ≠ green
- Binary constraints involve pairs of variables, e.g., SA ≠ WA
- Higher-order constraints involve 3 or more variables.

Constraint Satisfaction problem formulation

- **Another Example.**

Scheduling example…

- **Lecture 3 , p:74 - 157**

Note: Node and ARC consistency

Note: Then back tracking algorithm

Code: back tracking algorithm demonstration

Src3 – scheduling0.py

python-constraint

Src3 – scheduling1.py

Constraint Satisfaction problem formulation

- **Inference (optimization)….**

Scheduling example…

- **Lecture 3 , p:158 - 184**

Note: Arc consistency

- **Which variable should be assigned next?**
- **In what order should its values be tried?**
- **Can we detect inevitable failure early?**

Then: Another problem simulation….

Constraint Satisfaction problem

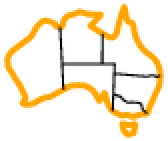
- Back track algorithm

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

function RECURSIVE-BACKTRACKING(*assignment*, *csp*) **returns** a solution, or failure
  if *assignment* is complete **then return** *assignment*
  var ← SELECT-UNASSIGNED-VARIABLE(*Variables[csp]*, *assignment*, *csp*)
  for each *value* **in** ORDER-DOMAIN-VALUES(*var*, *assignment*, *csp*) **do**
    if *value* is consistent with *assignment* according to Constraints[*csp*] **then**
      add { *var* = *value* } to *assignment*
      result ← RECURSIVE-BACKTRACKING(*assignment*, *csp*)
      if *result* ≠ *failue* **then return** *result*
      remove { *var* = *value* } from *assignment*
  return *failure*

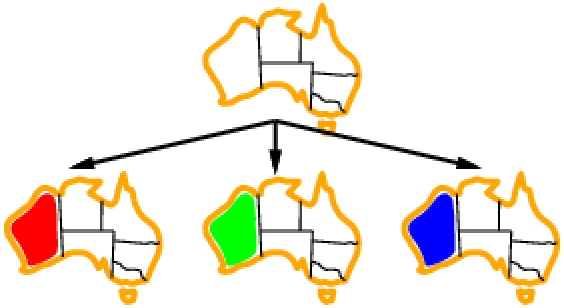
Constraint Satisfaction problem

- Back track algorithm simulation.



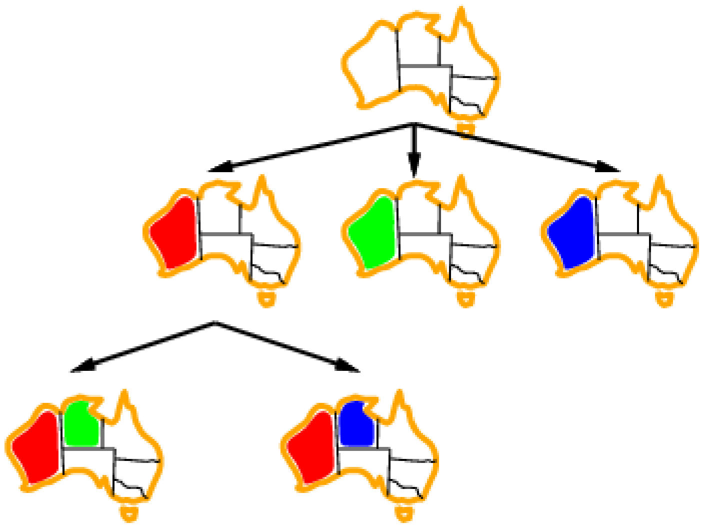
Constraint Satisfaction problem

- Back track algorithm simulation.



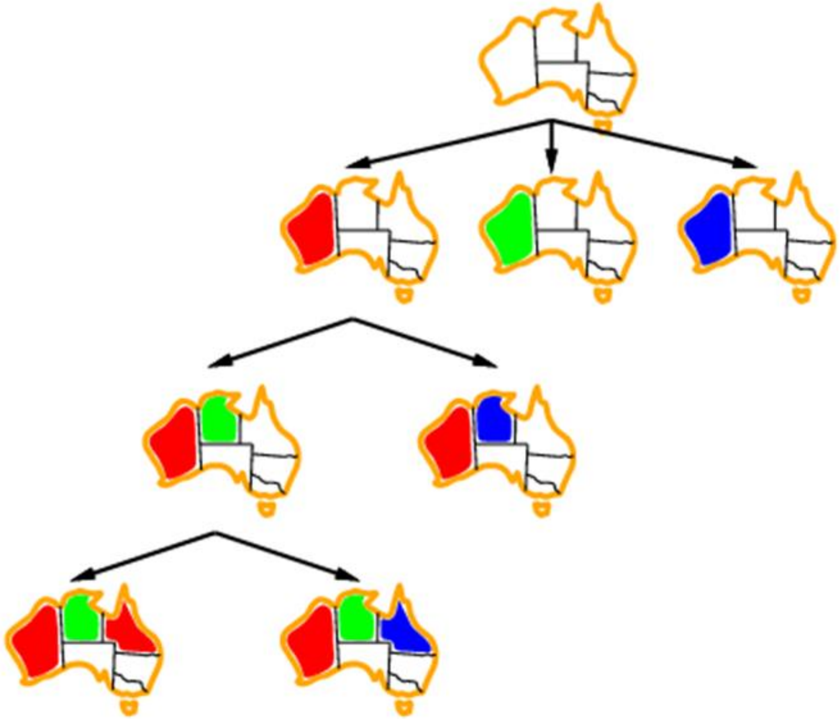
Constraint Satisfaction problem

- Back track algorithm simulation.



Constraint Satisfaction problem

- Back track algorithm simulation.



Back track algorithm

- Improving backtracking efficiency

- General-purpose methods can give huge gains in speed:

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

Back track algorithm

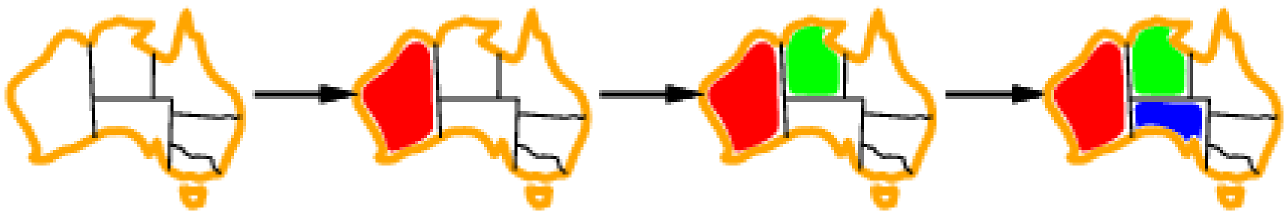
- **Improving backtracking efficiency**
- **General-purpose methods can give huge gains in speed:**
  - **Which variable should be assigned next?**
  - **In what order should its values be tried?**
  - **Can we detect inevitable failure early?**

Back track algorithm

- **Improving backtracking efficiency**

Most constrained variable:

choose the variable with the fewest legal values



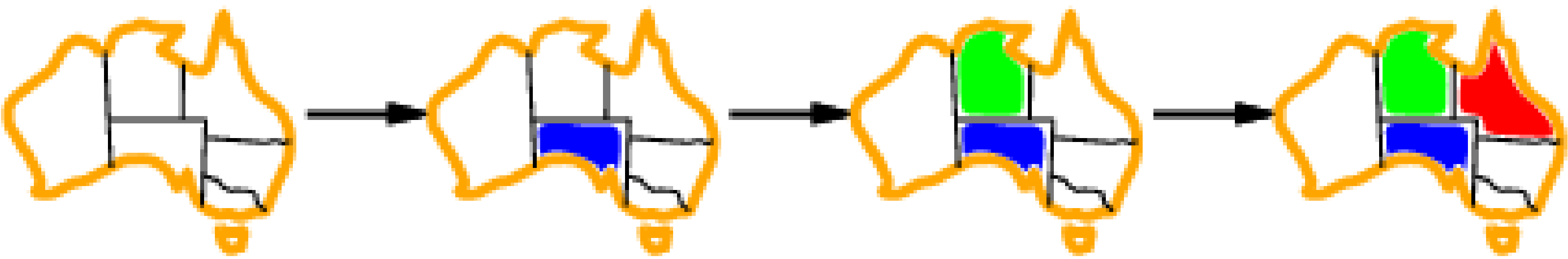
a.k.a. minimum remaining values (MRV) heuristic

Back track algorithm

- **Improving backtracking efficiency**

Most constrained variable:

choose the variable with the fewest legal values



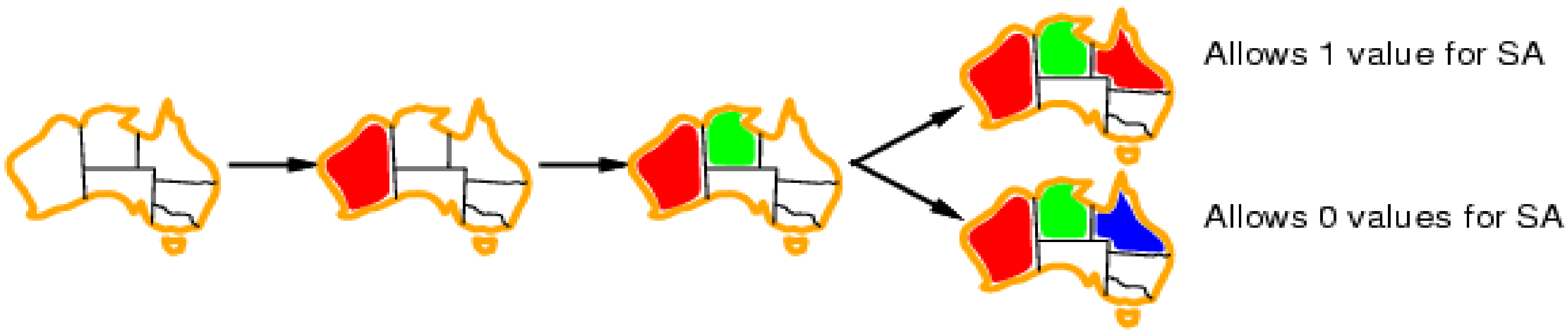
choose the variable with the most constraints on remaining variables

Back track algorithm

- **Improving backtracking efficiency**

Least constraining value:

The one that rules out the fewest values in the remaining values



Combining these heuristics makes 1000 queens feasible.

Back track algorithm

❑ Improving backtracking efficiency

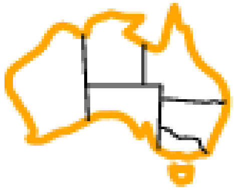
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

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
```

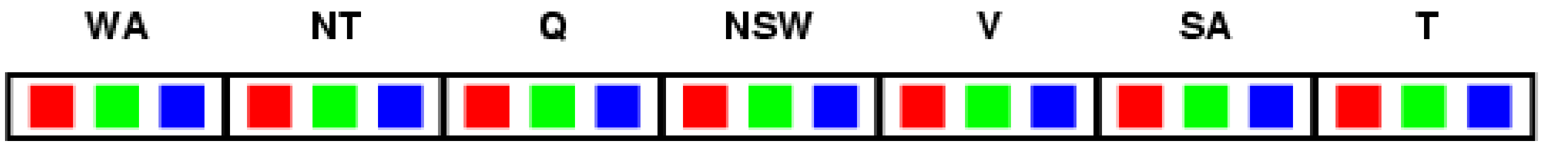
Back track algorithm

❑ Improving backtracking efficiency

Idea: Inference mechanisms (propagation using forward checking, or CS3)

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



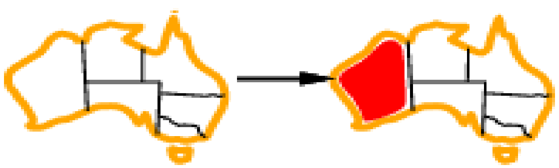


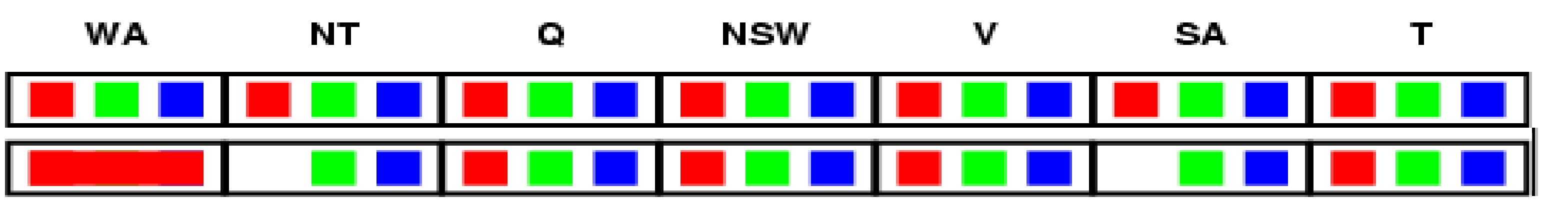
Back track algorithm

❑ Improving backtracking efficiency

Idea: Inference mechanisms (propagation using forward checking, or CS3)

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





Back track algorithm

- **Improving backtracking efficiency**

Idea: Inference mechanisms (propagation using forward checking, or CS3)

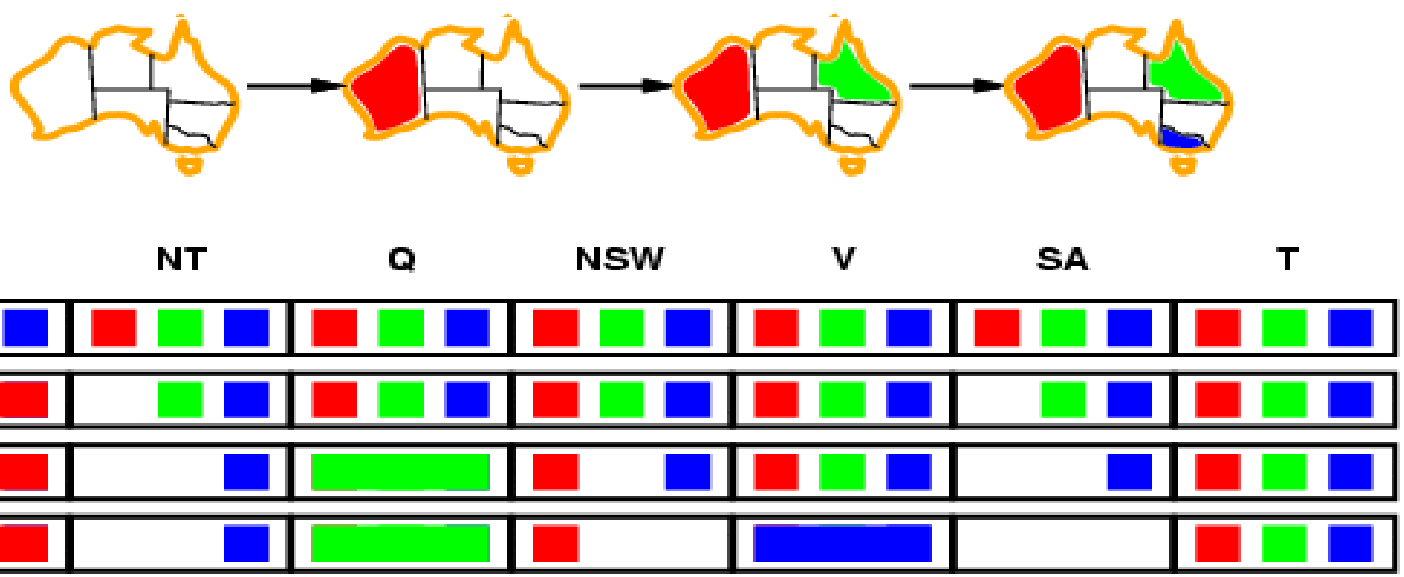
- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values

Back track algorithm

❑ Improving backtracking efficiency

Idea: Inference mechanisms (propagation using forward checking, or CS3)

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



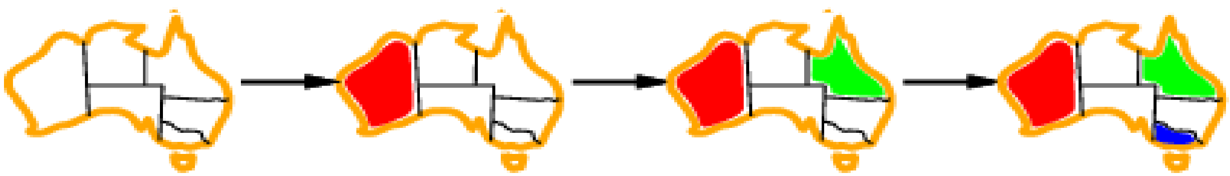
Back track algorithm

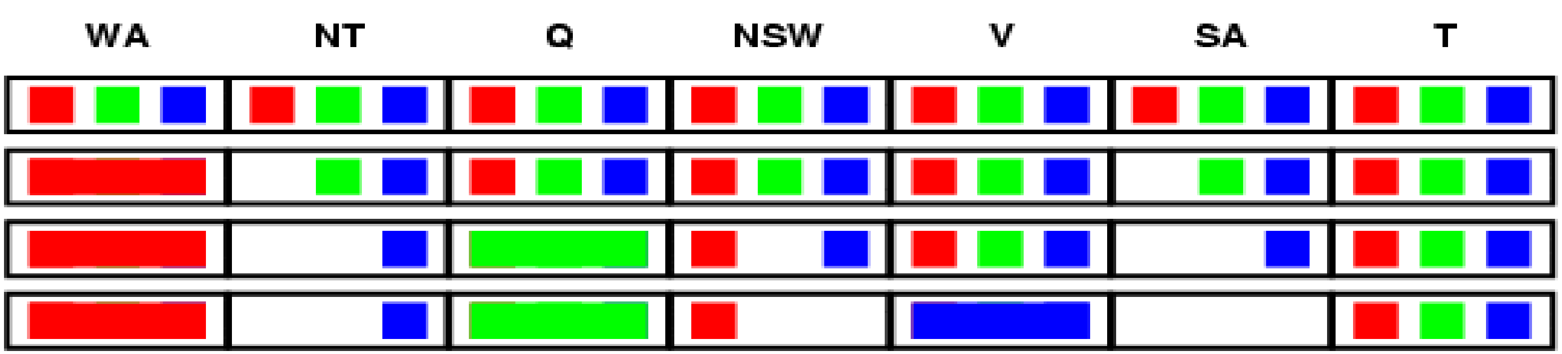
- **Improving backtracking efficiency**

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

NT and SA cannot both be blue!

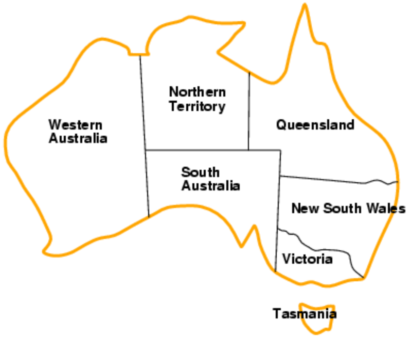
Constraint propagation repeatedly enforces constraints locally.





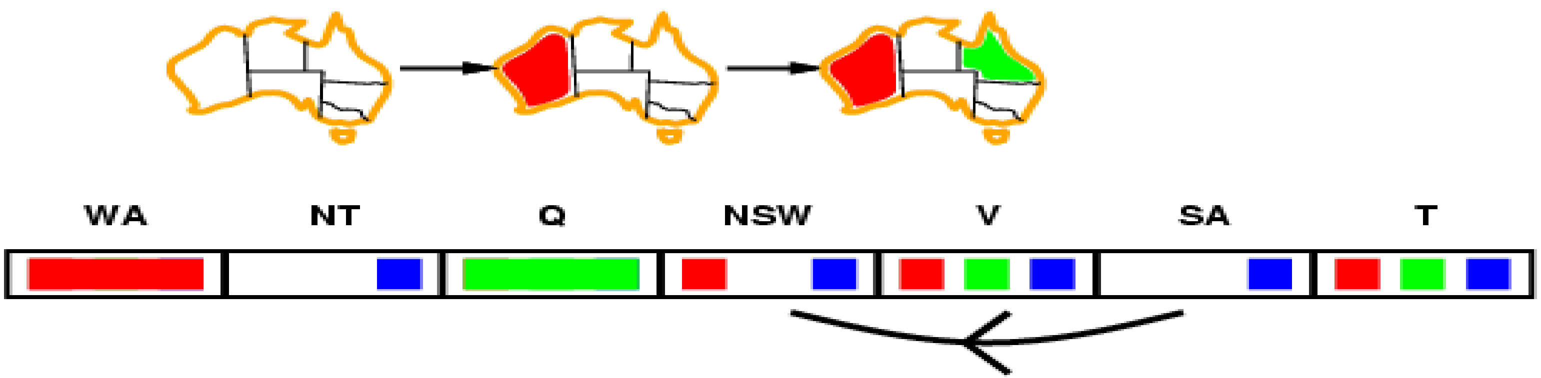
Back track algorithm

❑ **Improving backtracking efficiency**



Arc consistency:
Simplest form of propagation makes each arc consistent
$X \rightarrow Y$ is consistent iff:

for every value *x* of *X* there is some allowed *y*

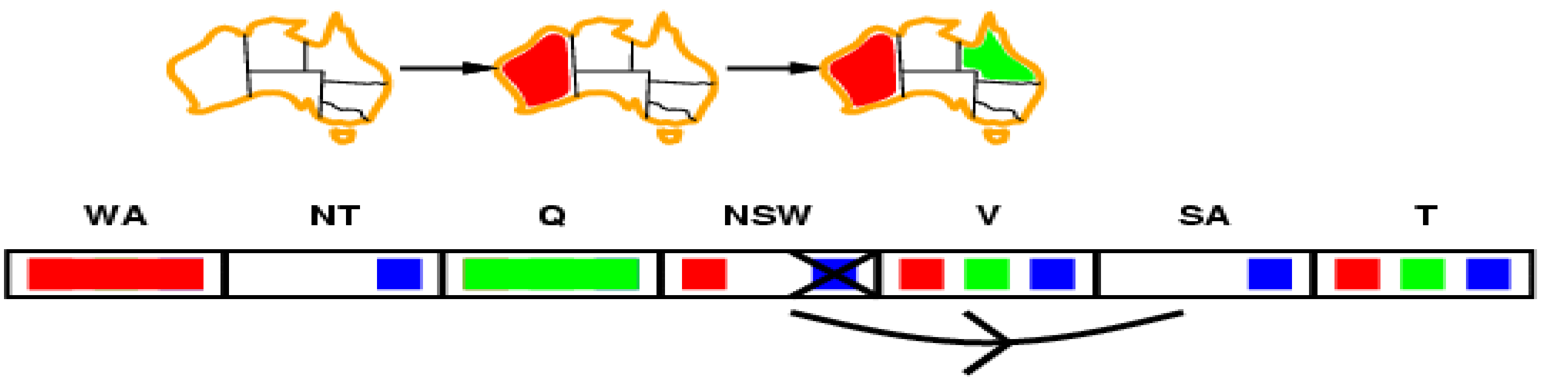


Back track algorithm

- **Improving backtracking efficiency**

Arc consistency:
Simplest form of propagation makes each arc consistent
$X \rightarrow Y$ is consistent iff:

for every value x of X there is some allowed y

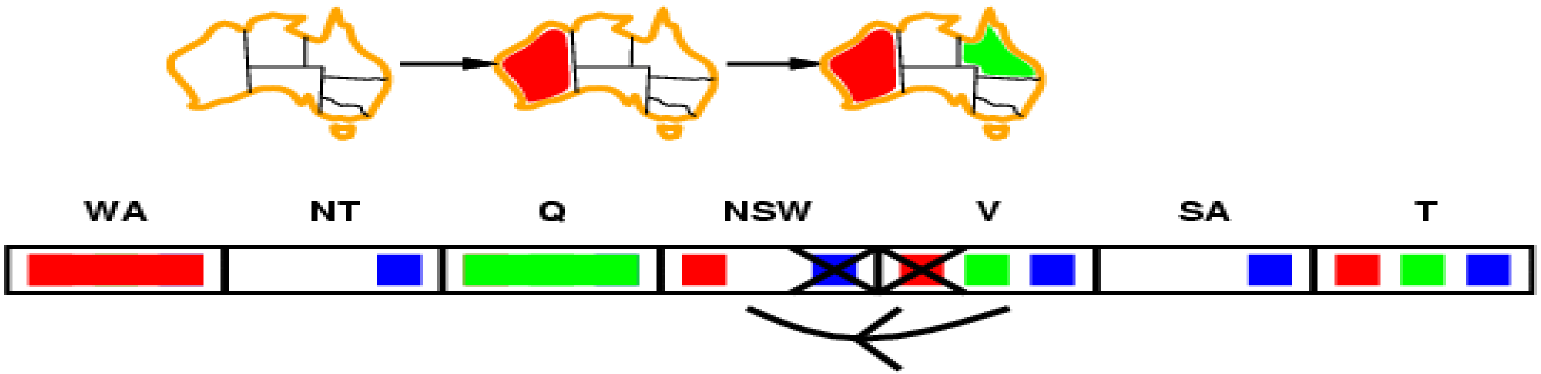


Back track algorithm

❑ Improving backtracking efficiency

Arc consistency:
Simplest form of propagation makes each arc consistent
X → *Y* is consistent iff:

for every value *x* of *X* there is some allowed *y*



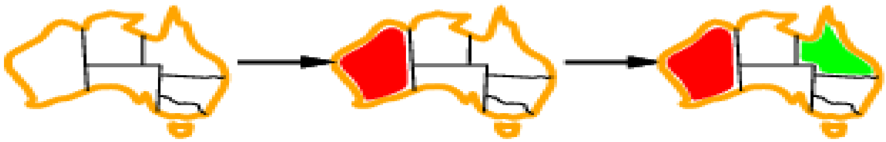
If *X* loses a value, neighbors of *X* need to be rechecked

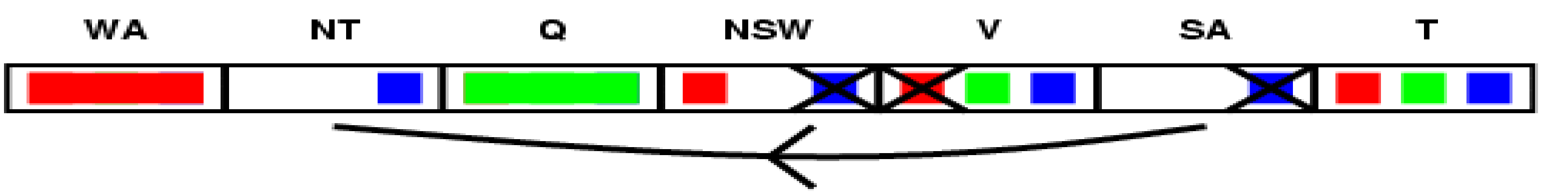
Back track algorithm

- **Improving backtracking efficiency**

Arc consistency:
Simplest form of propagation makes each arc consistent
X → *Y* is consistent iff:

for every value *x* of *X* there is some allowed *y*





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.

Arc consistency algorithm AC-3

```
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 RM-INCONSISTENT-VALUES(Xi, Xj) then
            for each Xk in NEIGHBORS[Xi] do
                add (Xk, Xi) to queue
```

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

Local search for CSPs

- Hill-climbing, simulated annealing typically work with "complete" states, i.e., all variables assigned
- To apply to CSPs:
    a. allow states with unsatisfied constraints.
    b. operators reassign variable values.
- Variable selection: randomly select any conflicted variable
- Value selection by min-conflicts heuristic:
    a. Choose value that violates the fewest constraints

Thank you for your attention!

Questions?