Artificial Intelligence
(IT3160E)

Than Quang Khoat

khoattq@soict.hust.edu.vn

School of Information and Communication Technology
Hanoi University of Science and Technology

2022

Content:

- Introduction of Artificial Intelligence
- Intelligent agent
- **Problem solving:** Search, **Constraint satisfaction**
- Logic and reasoning
- Knowledge representation
- Machine learning

Constraint

- A **constraint** is a relation on a set of variables
  - A variable has a set of possibly assigned values – domain
  - In this course, we consider only **finite** domains of discrete values
- A constraint can be represented by
  - A (math/logic) expression
  - A table that lists the appropriate value assignments for variables
  - …
- Examples of constraint
  - The sum of the angles in a triangle is 180^o
  - The length of word W is 10 characters
  - $X < Y$
  - Tuan can attend the seminar on Wednesday after 14:00
  - etc.

Constraint satisfaction problem

- A constraint satisfaction problem (CSP) consists of:
  - A finite set of variables *X*
  - A domain (i.e., a finite set of values) for each variable *D*
  - A finite set of constraints *C*
- A **solution** for a constraint satisfaction problem is a **full assignment** of the values of the variables so that all the constraints are satisfied
- A constraint satisfying problem can be represented by a graph

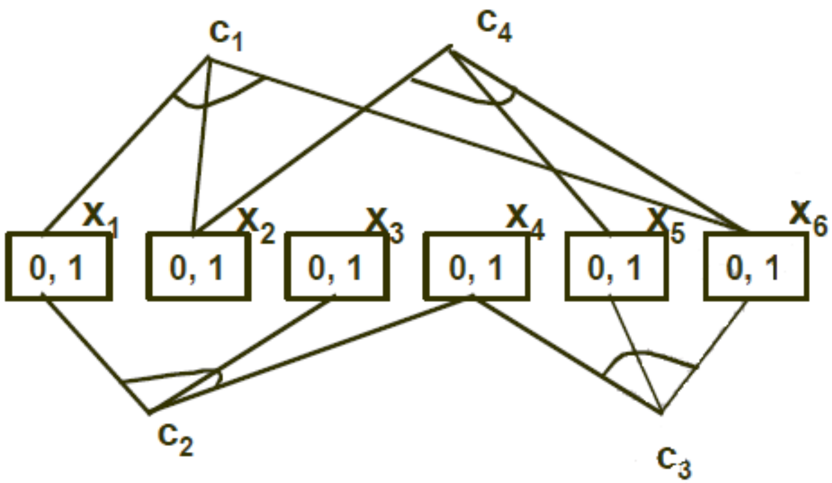
Example:

Variables: $x_1, \ldots, x_6$.

Value domain: {0,1}.

Constraints:

- $x_1 + x_2 + x_6 = 1$
- $x_1 - x_3 + x_4 = 1$
- $x_4 + x_5 - x_6 > 0$
- $x_2 + x_5 - x_6 = 0$

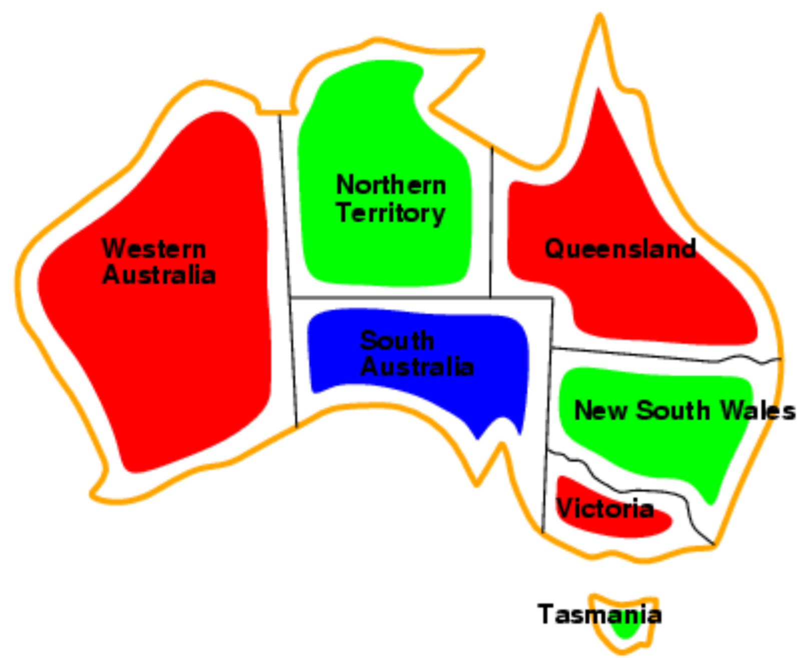


Example: Map-coloring problem (1)

- Variables: *WA, NT, Q, NSW, V, SA, T*
- Domain: D_i = {red, green, blue}
- Constraints: Adjacent regions must have different colors
  - WA ≠ NT
  - (WA,NT) = {(red,green), (red,blue), (green,red), (green,blue), (blue,red), (blue,green)}

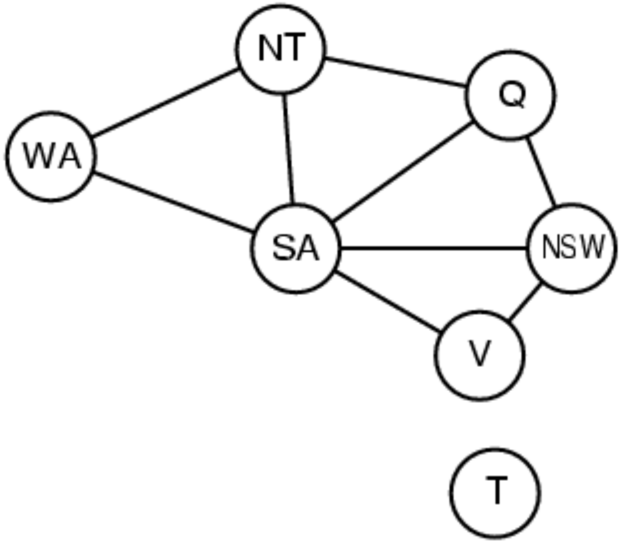
Example: Map-coloring problem (2)

- Solutions are **complete** and **consistent** assignments (i.e., satisfying all constraints)
- Example of a solution: *WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green*



Constraint graph

- Binary CSP: Each constraint relates two variables
- Constraint graph:
  - Nodes are variables,
  - Arcs are constraints



Types of CSP

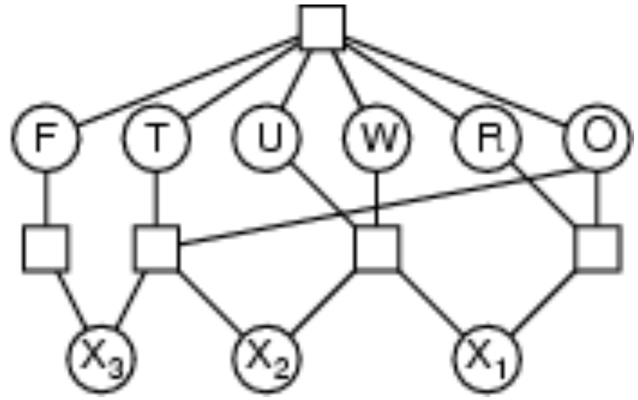
- Discrete variables
  - Finite domains:
    - For *n* variables and domain size *d*, the number of complete assignments is $O(d^n)$
    - Example: Boolean CSPs
  - Infinite domains:
    - Domains of integers, strings, etc.
    - Example: In the job scheduling problem, the variables are the start and end dates for each job
    - Need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
- Continuous variables

Types of constraint

- **Unary** constraints involve a single variable
  - Example: SA ≠ green
- **Binary** constraints involve pairs of variables
  - Example: SA ≠ WA
- **Higher-order** constraints involve 3 or more variables
  - Example: Constraints in the cryptarithmetic problem (presented in the next slide)

Example: Cryptarithmetic problem

```
   T W O
 + T W O
 -------
 F O U R
```

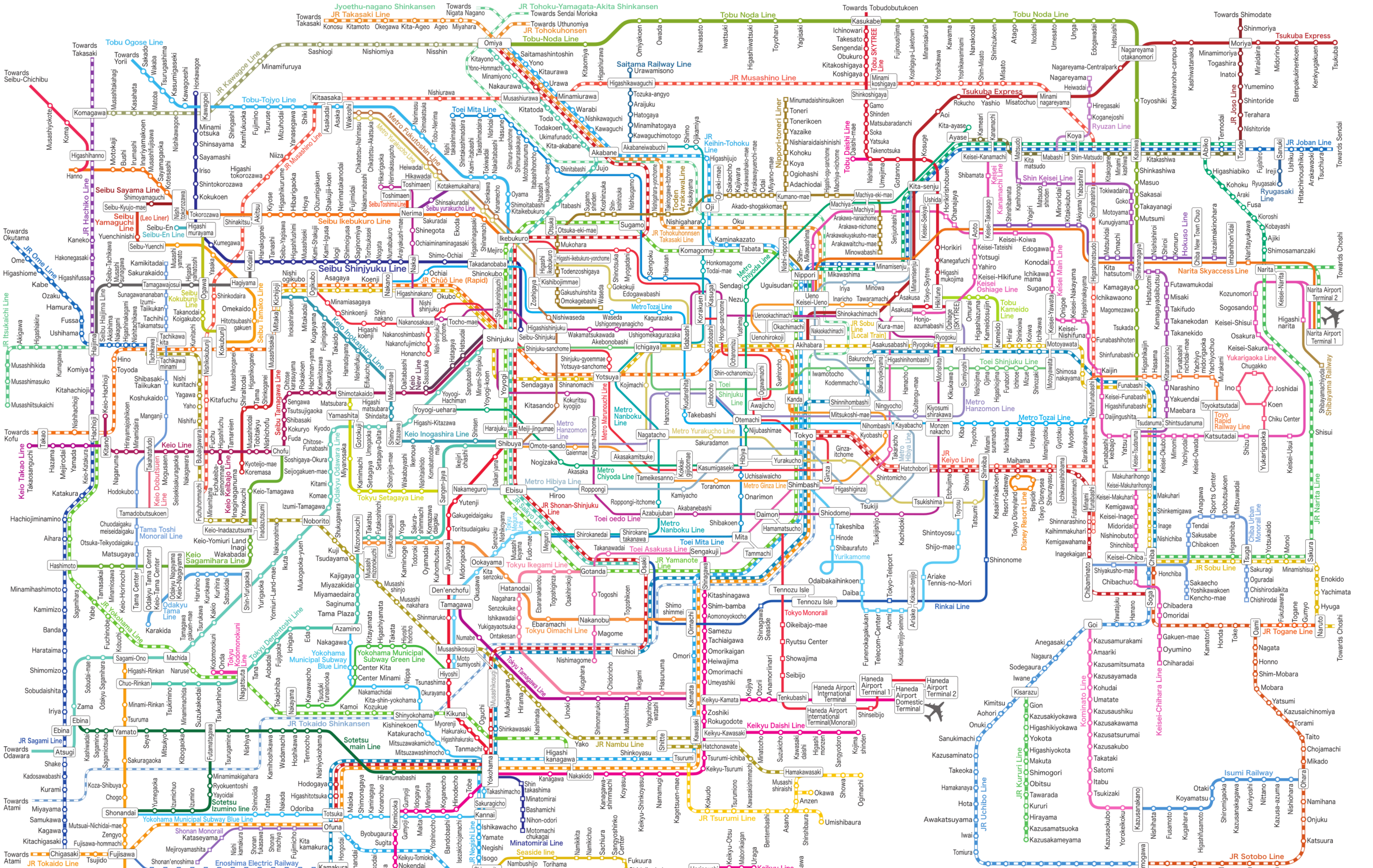


- Variables: *F T U W R O* $X_1\ X_2\ X_3$ (the memories of the operator "+")
- Value domain: {*0,1,2,3,4,5,6,7,8,9*} for the 6 variables of F, T, U, W, R, O, and {0, 1} for the 3 variables of X_1, X_2, X_3
- Constraints: The values of the variables F, T, U, W, R, O are different
  - $O + O = R + 10 * X_1$
  - $X_1 + W + W = U + 10 * X_2$
  - $X_2 + T + T = O + 10 * X_3$
  - $X_3 = F$
  - $T \neq 0$
  - $F \neq 0$

Real-world CSPs

- Assignment problems
  - Example: Who teaches what class?
- Timetabling (i.e., scheduling) problems
  - Example: Which class is offered when and where?
- Transportation scheduling
- Production scheduling in factories
- Note: Many real-world problems involve real (i.e., continuous) -valued variables

Scheduling for Tokyo railway?



Generate and test (1)

- Is the most general problem-solving method
- Generate and Test approach:
  - Generate a candidate for the solution
  - Check if this candidate is really a solution
- Apply the Generate and Test approach to CSP:
  - Step 1. Assign values to all variables
  - Step 2. Check if all the constraints are satisfied
  - Repeat these 2 steps until a satisfactory assignment has been found

Generate and test (2)

- A serious weakness: need to consider *too many assignment candidates that (obviously) do not satisfy the constraints*

- Example
  - Variables X,Y,Z, each takes values in {1, 2}
  - Constraints: $X=Y$, $X \neq Z$, $Y>Z$
  - Assignment candidates: (1,1,1); (1,1,2); (1,2,1); (1,2,2); (2,1,1); (2,1,2); (2,2,1)

Generate and test (3)

- How to improve the Generate and Test approach?
  - Better (i.e., smarter) generate assignment candidates
    - Not in sequential order
    - Use the results (information) obtained from the test step (Step 2)
  - Early detect contradictions
    - Constraints are checked immediately after each variable is assigned a value (i.e., not waiting until all the variables are assigned values)

Backtracking search

- **Backtracking** is the most commonly used search algorithm in CSP
  - Based on depth-first search
  - For each assignment, only assign value to one variable
    - (Generate and Test: For each assignment, assign values to all variables)
- Backtracking search for a CSP
  - Assign values to variables in turn – Assignment of one variable is only made after the assignment of another variable has been completed
  - After a variable's value assignment, checks if the constraints are satisfied by all variables that have been assigned up to the moment – Backtrack if a contradiction occurs (i.e., a constraint is not satisfied)

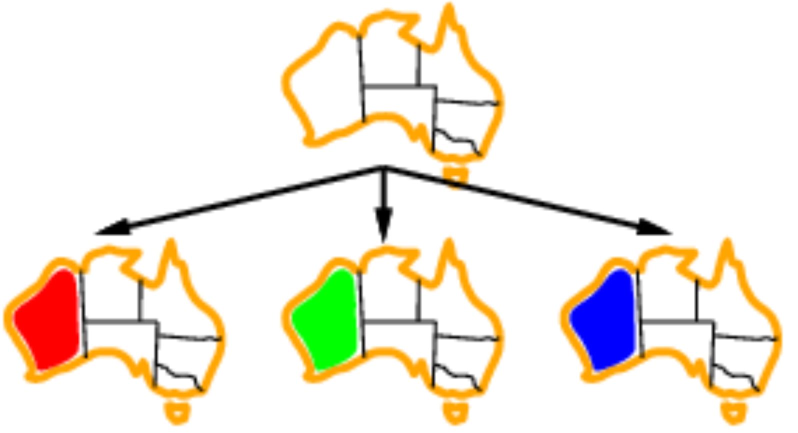
Backtracking search algorithm

function Backtracking-Search(*csp*) **returns** a solution, or failure
  return Recursive-Backtracking({}, *csp*)

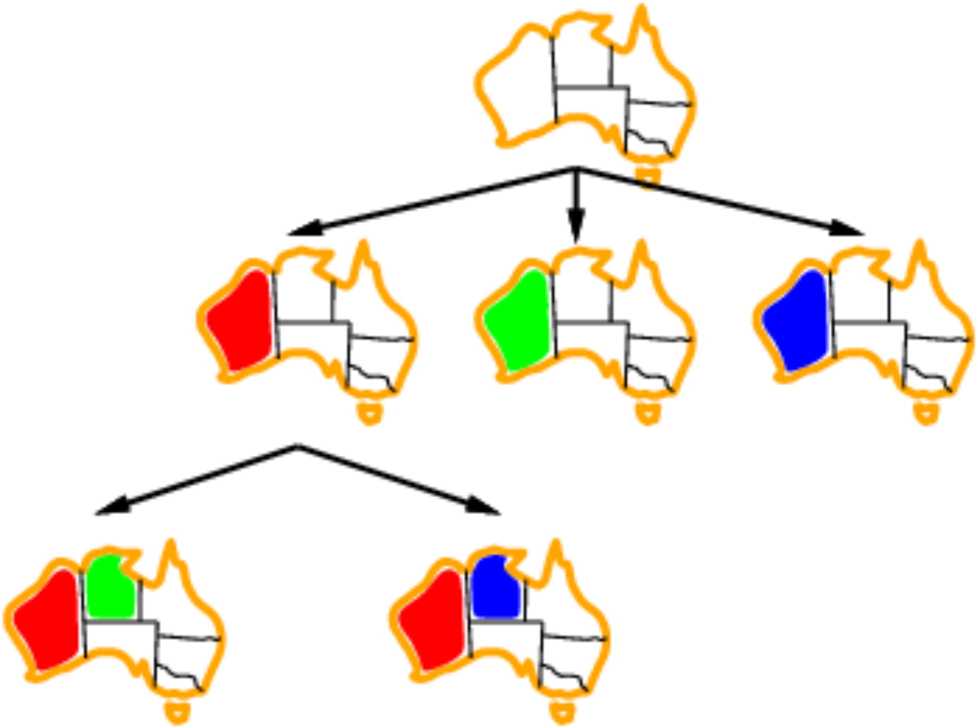
function Recursive-Backtracking(*assignment*, *csp*) **returns** a solution, or failure
  if *assignment* is complete **then return** *assignment*
  var ← Select-Unassigned-Variable(*Variables[csp]*, *assignment*, *csp*)
  for each *value* **in** Order-Domain-Values(*var*, *assignment*, *csp*) **do**
    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*

Backtracking search: Example (1)

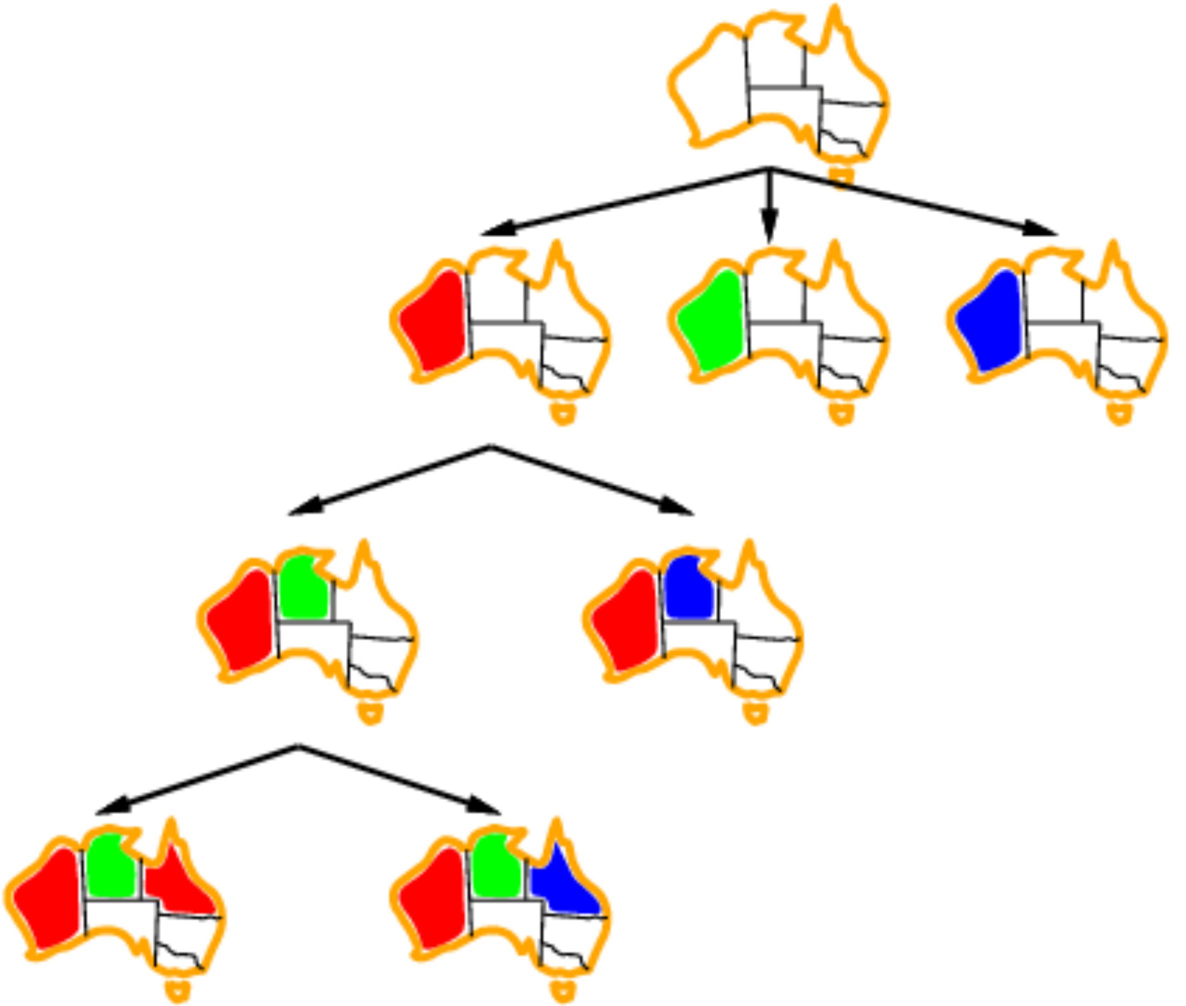
Backtracking search: Example (2)



Backtracking search: Example (3)



Backtracking search: Example (4)



Backtracking search: Properties (1)

- Factors that influence the backtracking method:
  - The order of consideration of the variables?
    - Prioritize the more important variables (defined by a specific problem)
    - Prioritize variables with fewer values (smaller value domain)
    - Prioritize variables involved in many constraints
  - For a variable, the order of consideration of the values?
    - Prioritize the more important values (defined by a specific problem)

Backtracking search: Properties (2)

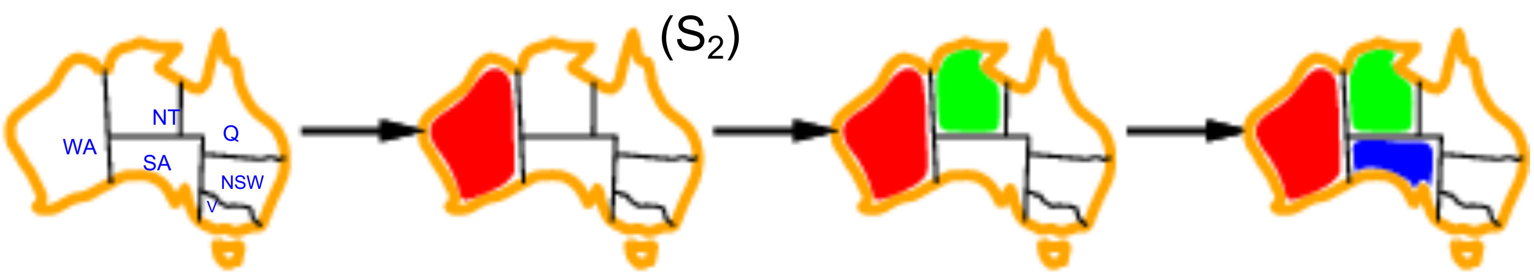
- Late detection of contradictions (i.e., constraints violation)
  - Limitation: Constraint violation is detected <u>only after</u> a value is assigned (for a variable)
  - Example:
    - The variables A,B,C,D,E take integer values in [1..10]
    - Constraint: A=3*E
    - Only after assigning value for variable E it discovers that A>2
  - Solution: Forward Checking (i.e., check the constraints in advance)

Backtracking search: Improvement

- The efficiency of the backtracking search method in CSP can be improved by:
  - Order of consideration of variables (for assigning values)
  - Order of consideration of values (for each variable)
  - Early detection of contradictions (i.e., constraints violation) that will occur

Most constrained variable

- Rule for choosing the order of variables: Prioritize the most constrained variable
  - Choose the variable with the fewest legal values
  - Example: At step S_2, variable NT is chosen because it has the least number of legal values (2)

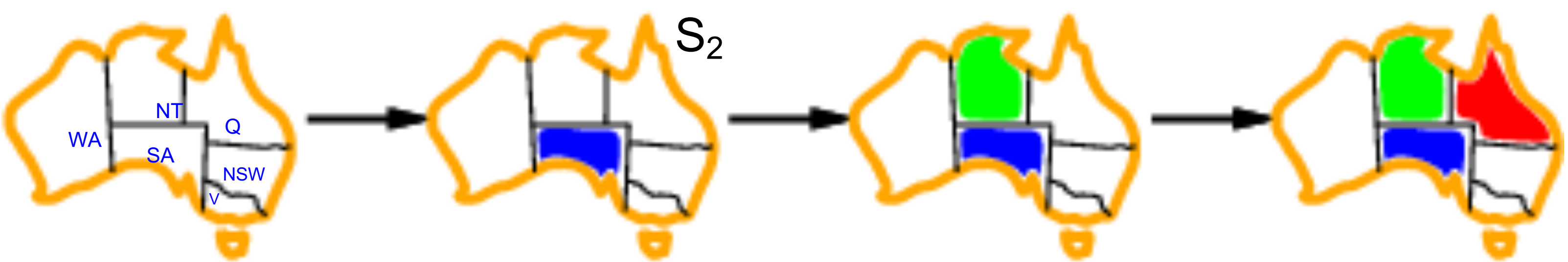


- As known as the MRV (Minimum Remaining Values) heuristic

Most constraining variable

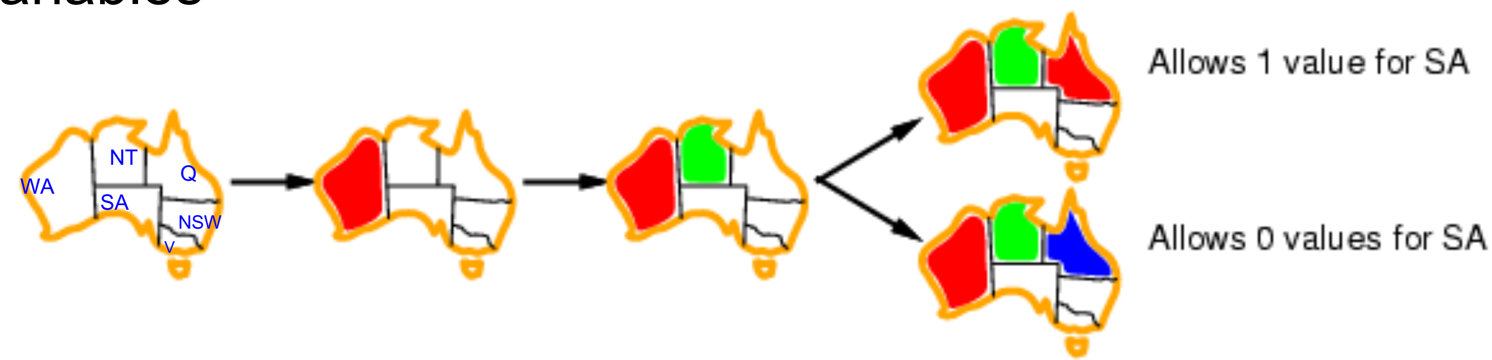
- When there are 2 or more variables with the same minimum number of legal values, select which one?
  - Example: In the previous slide, NT and SA have the same minimum number (=2) of legal values

- Choose the variable that constraints the other variables (<u>not yet assigned value</u>) at most
  - Example: At step S_2, variable SA should be considered before variable NT – because SA constraints 5 variables while NT constraints just 3 variables



Least constraining value

- For a variable, its values are considered (for assignment) in what order?
- Choose the value that constraints other variables (<u>not yet assigned value</u>) at least
  - The value that rules out the fewest values in the remaining variables



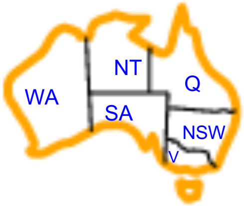
Q=Red reduces the value domain of SA to 1 value

Q=Blue reduces the value domain of SA to 0 values

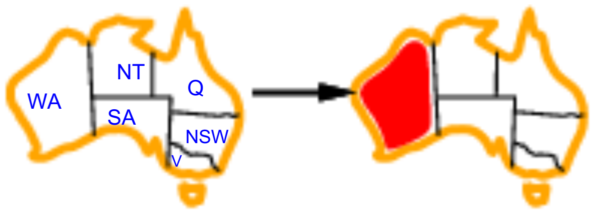
Forward checking

- Goal: Avoid failures by pre-checking constraints
- Forward checking ensures consistency between the variable being assigned the value and the other variables that are **directly related (constrained)** with it
- Idea:
  - When assigning value for a variable, keep track of legal values for unassigned variables
  - Stop the current search direction/path when there is an unassigned variable that has no legal values

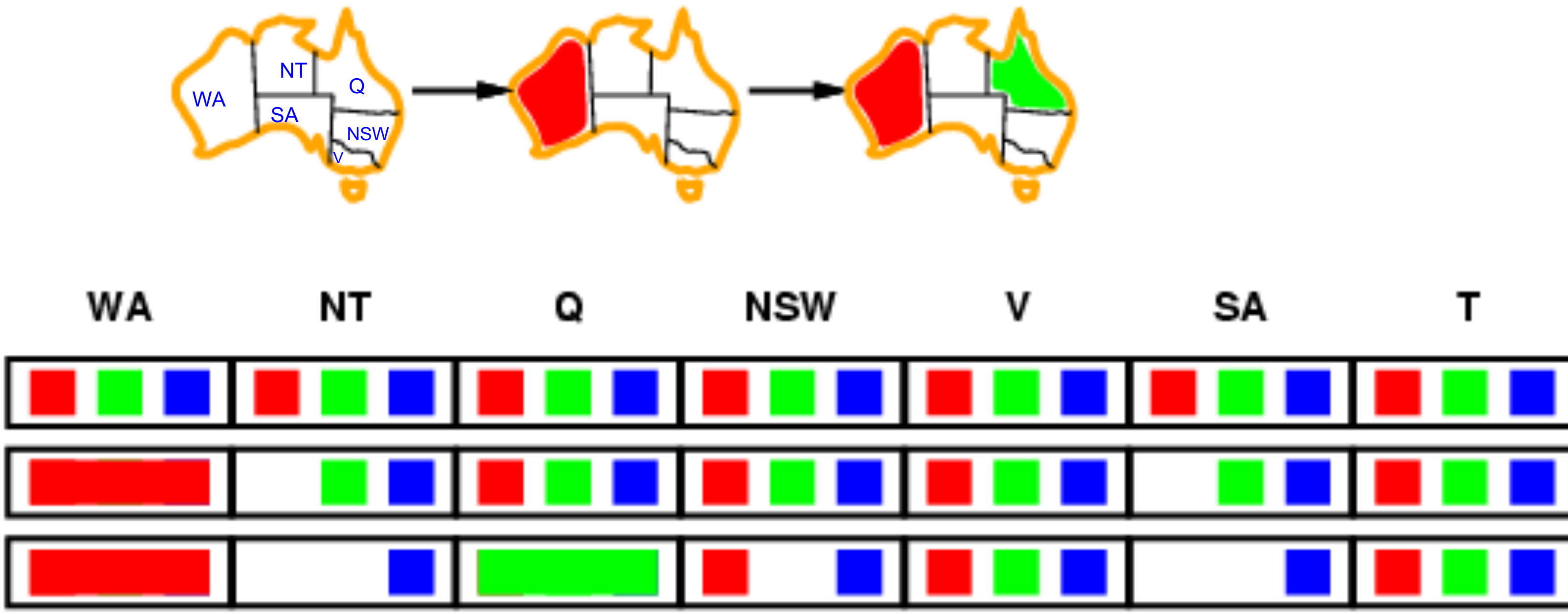
Forward checking: Example (1)



Forward checking: Example (2)

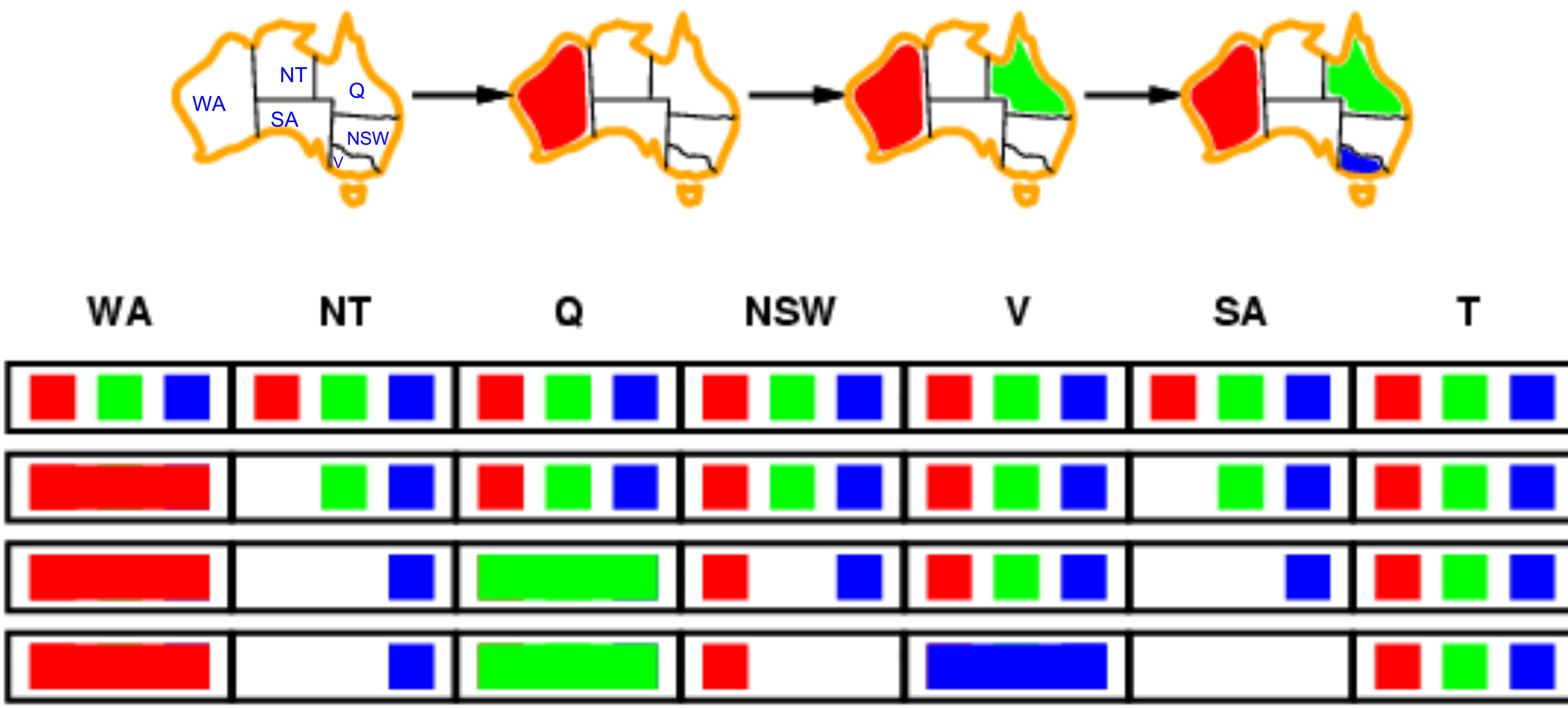


Forward checking: Example (3)



Artificial intelligence

Forward checking: Example (4)



Constraint propagation

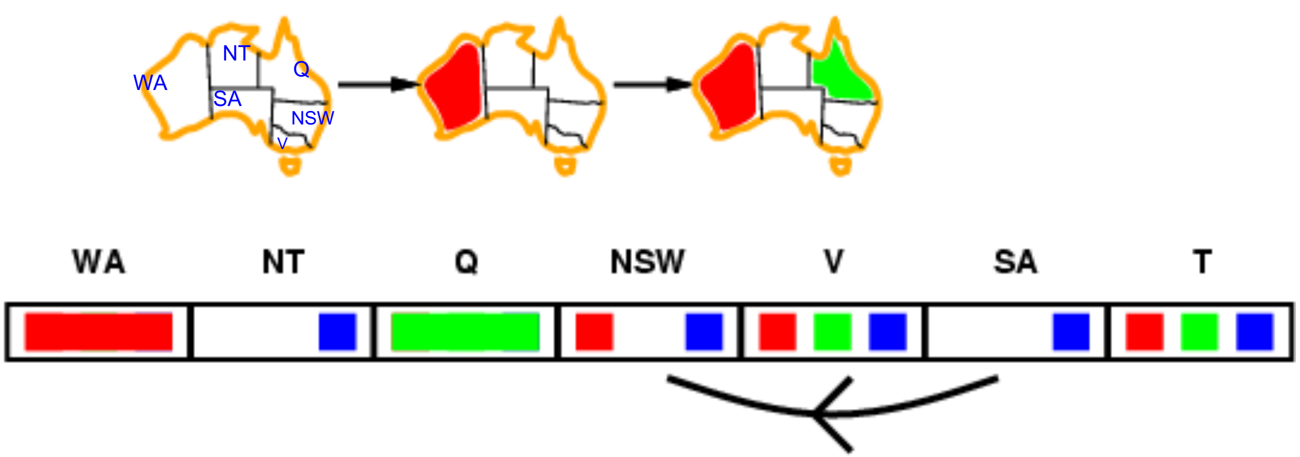
- Forward checking can propagate information (constraints) from assigned variables to the ones, which are available for assignment
- But: this method cannot prevent all future failures
  - E.g.: NT & SA cannot have the same color!

- Constraint propagation can only ensure **local consistency** of the constraints

Arc consistency in constraint graph (1)

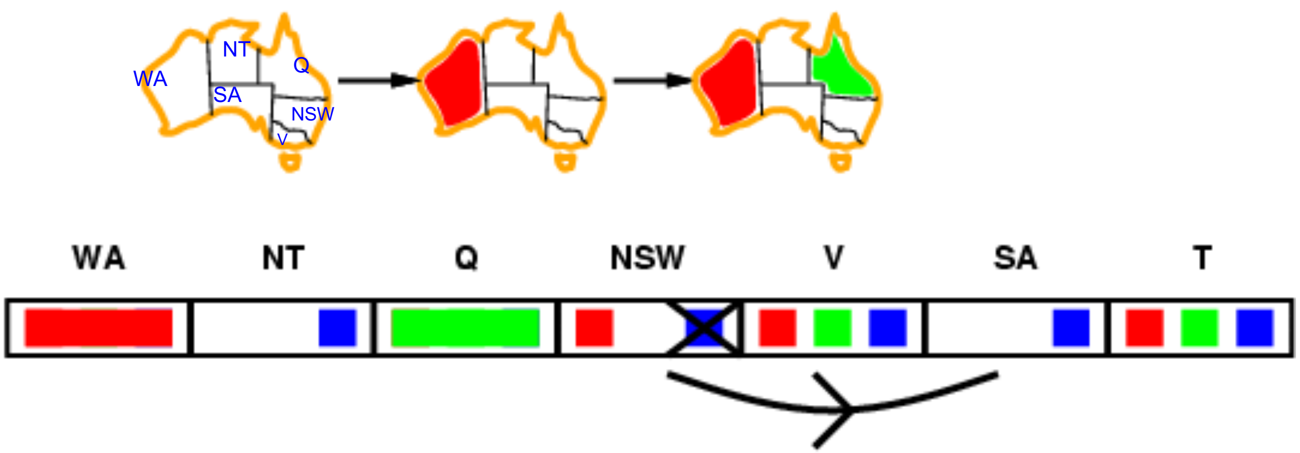
- In a constraint graph, an arc ($X \rightarrow Y$) is said to be *consistent (phù hợp, thống nhất)* if and only if, for each value *x* of variable *X*, there exists a value y of variable *Y* such that every constraint relating *X* and *Y* is satisfied.

- Note
  - Consistency of ($X \rightarrow Y$) <u>does not mean</u> consistency of ($Y \rightarrow X$)
  - Ex.: (*SA* → *NSW*) is consistent, but (*NSW* → *SA*) is inconsistent



Arc consistency in constraint graph (2)

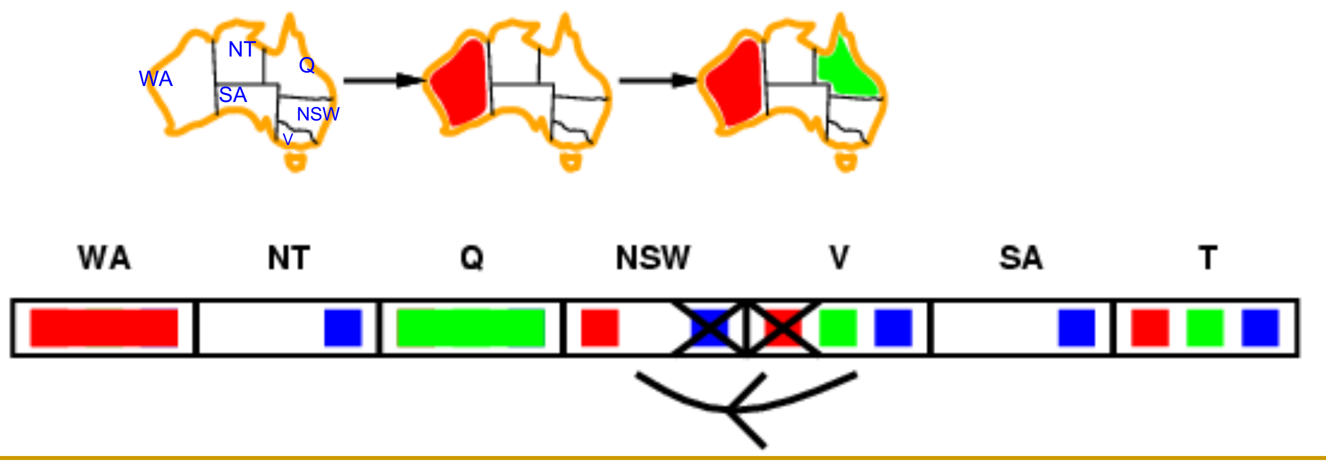
- To ensure consistency of arc (*X* → *Y*), we need to remove any value ***x* of variable *X*** which violates any constraint containing *Y*
  - Ex.: consistency of (*NSW* → *SA*) requires removal of "blue" from the list of admissible values for variable *NSW*



Arc consistency in constraint graph (3)

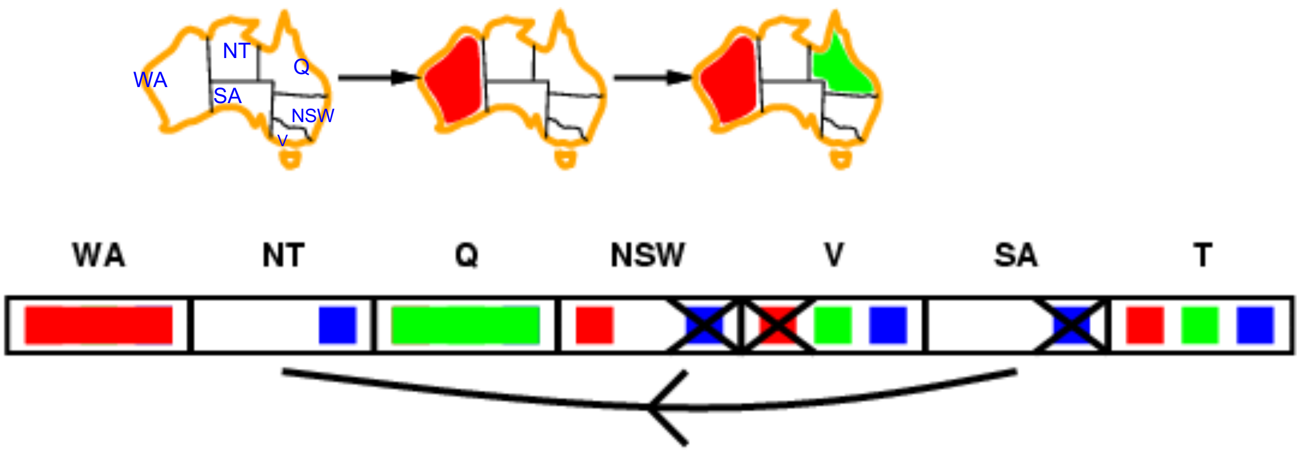
- After removal of value *x*, we need to take care of all constraints containing variable *X*: consider any arc (… → *X*)
  - Ex.: After removal of "blue" in variable *NSW*, we reconsider arcs (*V* → *NSW*), (*SA* → *NSW*), (*Q* → *NSW*)

  … To ensure consistency of (*V* → *NSW*), we need to remove "red" of variable *V*



Arc consistency in constraint graph (4)

- Detecting arc consistency can help discover failures more efficiently than Forward checking
- Consistency detection can be used before or after each assignment of a value to a variable



AC-3 algorithm

```
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 CSP (1)

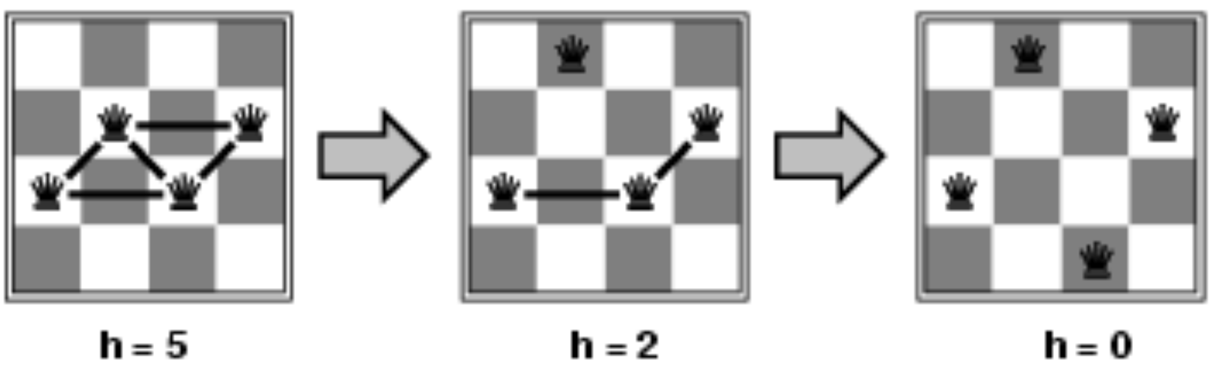
- Goal: To use local search methods (e.g., hill-climbing, simulated annealing) for constraint satisfaction problem
- Each state (of the search space) corresponds to a *complete* assignment of values to *all* variables
  - The search space includes also the states in which the constraints are violated
  - State transition = Assign new values to variables
- Goal state = The state in which all constraints are satisfied

Local search for CSP (2)

- Search process
  - Select a variable to assign a new value? $\rightarrow$ Randomly select a variable whose value violates the constraints
  - For a variable, select the new value? $\rightarrow$ Based on the **min-conflicts** heuristic: Choose the value that violates the constraints at least
- Example: Applying Hill-climbing, with the heuristic function *h(n)* = Number of violated constraints
  - Next neighbor state is one that has a better value of *h(n)* (less number of violated constraints)

Example: 4-queens problem

- States: Positions of 4 queens in 4 columns
  - Only one queen in each columns
  - The state space consists of 256 (=4x4x4x4) states
- Actions: Move queen in a column
- Goal state: No attacks
- Evaluation: *h(n)* = Number of attacks



CSP: Summary

- In a constraint satisfaction problem (CSP) :
  - States are defined by values of a fixed set of variables
  - Goal test is defined by constraints on variable values
- Backtracking = Depth-first search with one variable assigned per node
- Variable ordering and value selection heuristics help significantly
- Forward checking prevents assignments that lead to later failure
- A local search method using the min-conflicts heuristic is often effective in many real-world problems