Introduction to Artificial Intelligence

Chapter 2: Solving Problems by Searching (7)
Constraint Satisfaction Problems

Nguyễn Hải Minh, Ph.D
nhminh@fit.hcmus.edu.vn

Outline

1. Constraint Satisfaction Problems (CSP)
2. Constraint Satisfaction Problem as a Search
3. Constraint Propagation: Inference in CSPs

Constraint Satisfaction Problem

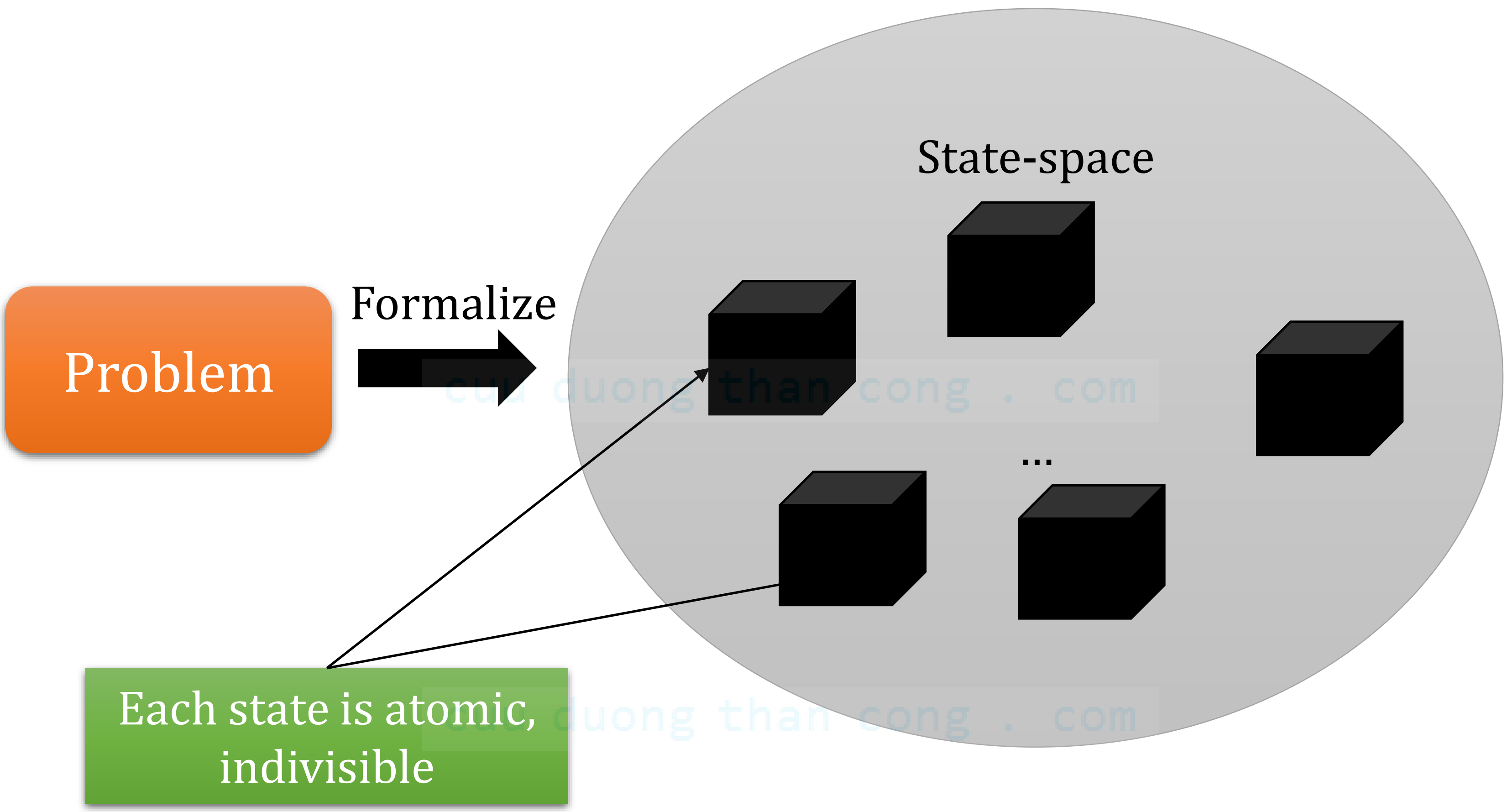
What is a Constraint Satisfaction Problem?

- A problem with ***constraints***!
  - Time constraints
  - Budget constraints
- Few constraints, handle them manually
- Many constraints, we need computers
- Here we will only touch on problems that have **finite** domain variables.
  - This means that the domains are a finite set of integers, as opposed to a real-valued domain that would include an infinite number of real-values between two bounds.

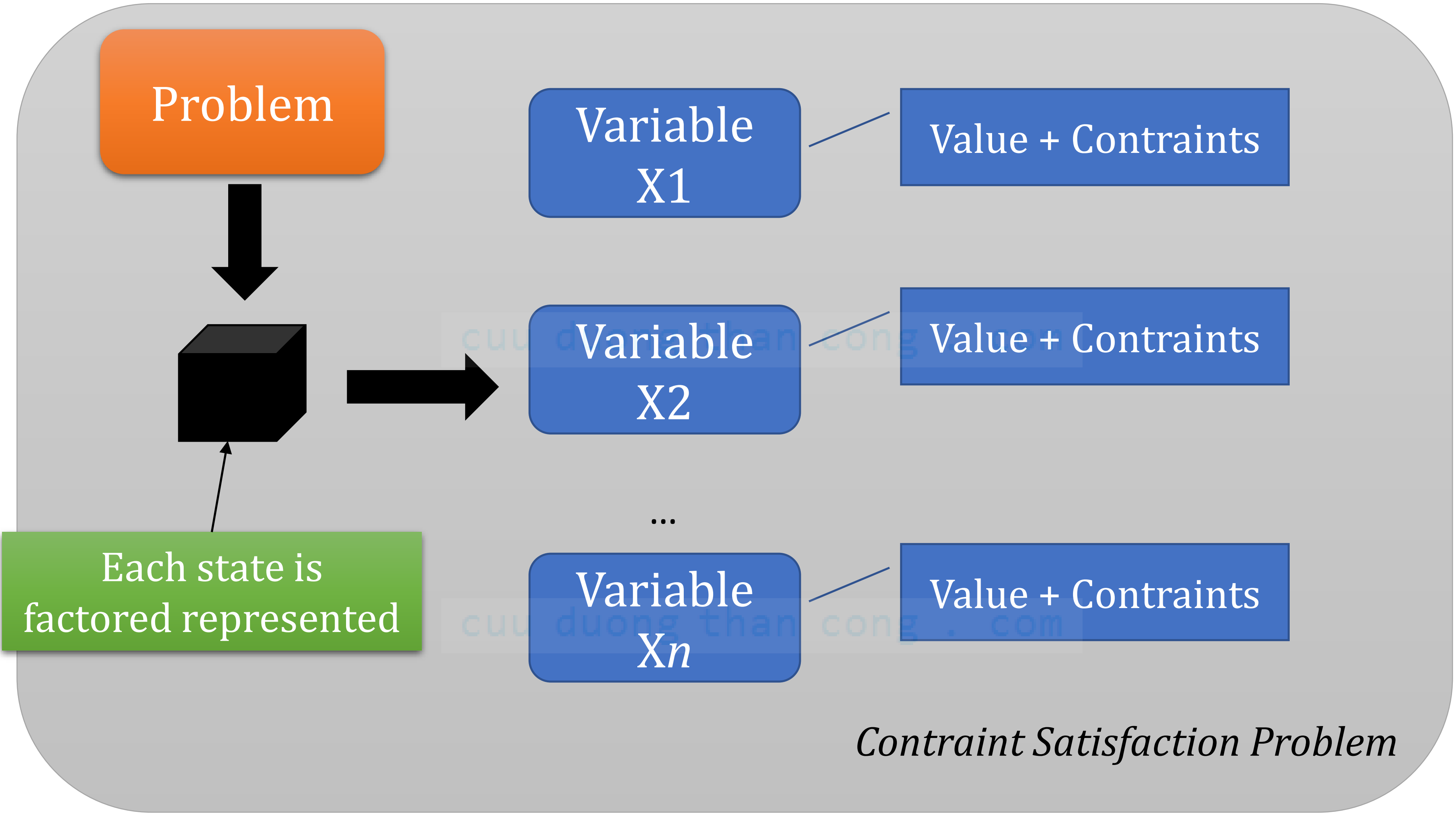
Solving Constrain Satisfaction Problem

- Like many other AI problems, CSP's are solved using ***search***
- Unlike other AI problems, CSP's exhibit a **standard structure**
- Knowledge about this structure (as heuristics) can be incorporated in the solution process

State-space search



Constraint Satisfaction Problem



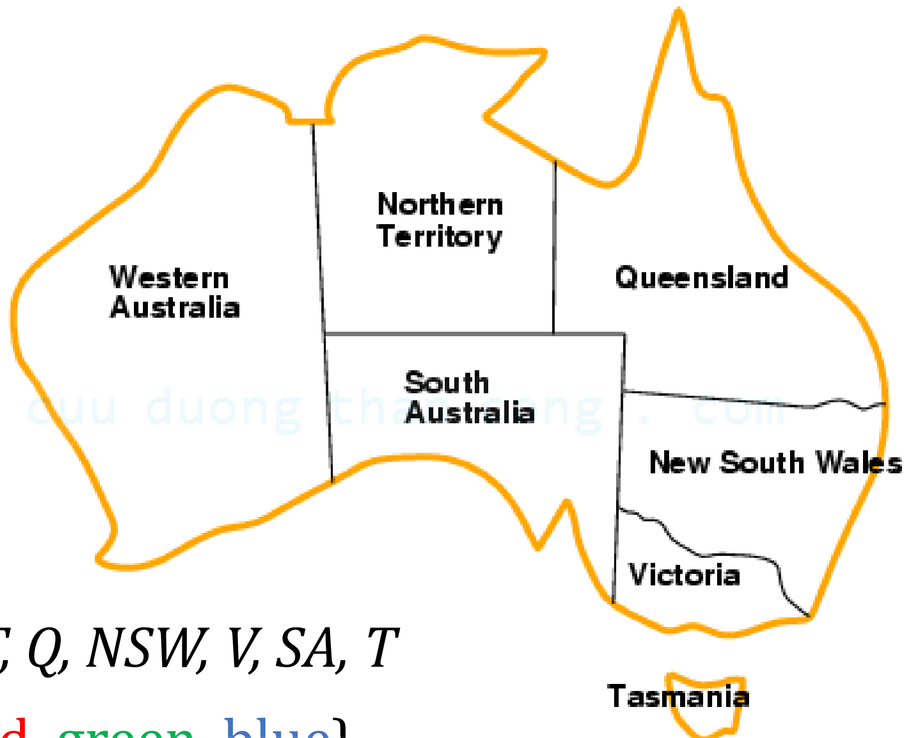
Constraint satisfaction problems (CSPs)

- Standard search problem:
  - state is a "black box" – any data structure that supports successor function, heuristic function, and goal test
- CSP:
  - state is defined by variables X_i with values from domain D_i
  - goal test is a set of constraints C specifying allowable combinations of values for subsets of variables

Constraint satisfaction problems (CSPs)

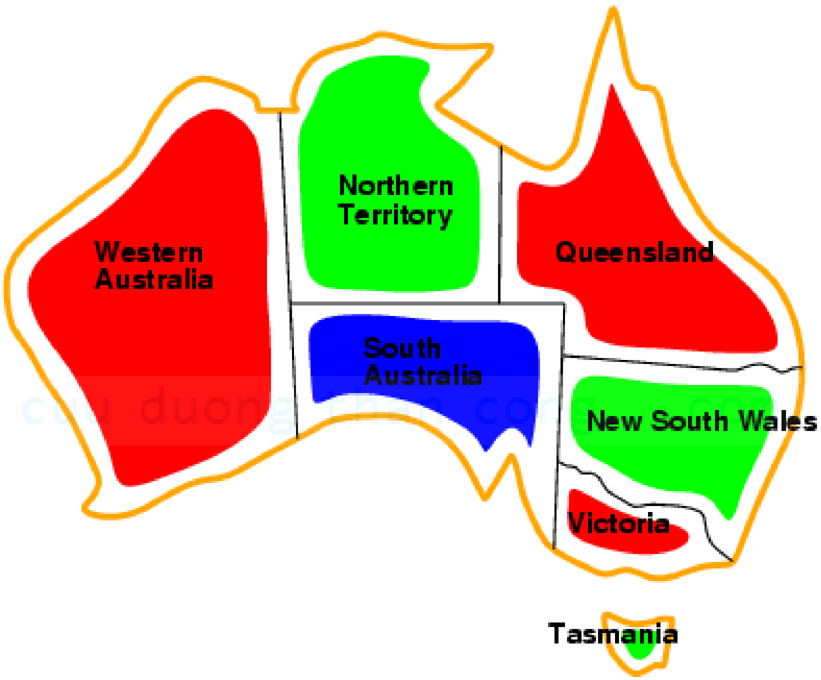
- An assignment is *complete* when every variable is mentioned.
- *Consistent assignment*
  - Each assignment (a state change or step in a search) of a value to a variable must be consistent: it must not violate any of the constraints.
- A *solution* to a CSP is a complete assignment that satisfies all constraints.
- Some CSPs require a solution that maximizes an *objective function*.
- CSP benefits
  - Standard representation pattern
  - Generic goal and successor functions
  - Generic heuristics (no domain specific expertise)

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, or (WA,NT) in {(red,green), (red,blue), (green,red), (green,blue), (blue,red), (blue,green)}

Example: Map-Coloring

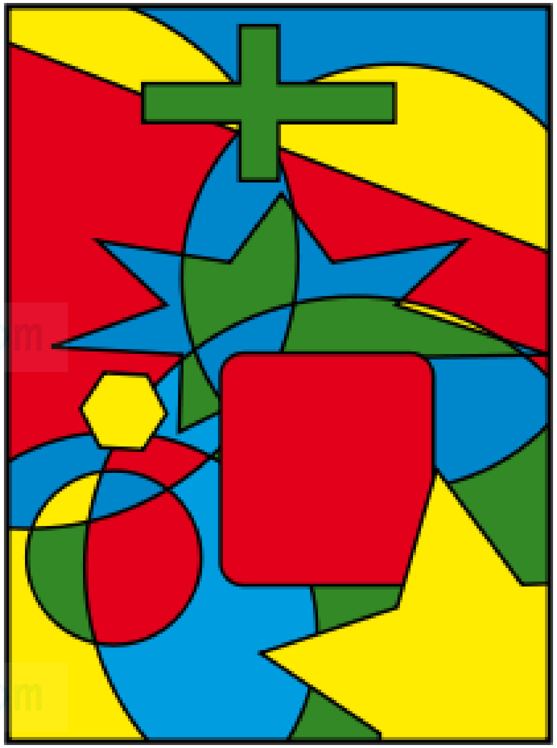


- Solutions are complete and consistent assignments
  - e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

Aside: Famous Graph Coloring Porblem

- ❑More general problem than map coloring
- ❑Planar graph = graph in the 2d plane with no edge crossings
- ❑Guthrie's conjecture (1852)
  - o *Every planar graph can be colored with 4 colors or less*

  → Proved (using a computer) in 1977 (Appel and Haken)



Real-world CSPs

- ❑Operations Research (scheduling, timetabling)
- ❑Bioinformatics (DNA sequencing)
- ❑Electrical engineering (circuit layout-ing)
- ❑Telecommunications
- ❑Scheduling the time of observations on the Hubble Space Telescope
- ❑Airline schedules
- ❑Cryptography
- ❑Computer vision -> image interpretation

→ *Notice that many real-world problems involve real-valued variables*

Varieties of CSPs

- **Discrete variables**
  - finite domains:
    - n variables, domain size d → $O(d^n)$ complete assignments
    - e.g., N-Queens
  - infinite domains:
    - integers, strings, etc.
    - e.g., job scheduling, variables are start/end days for each job
    - need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$

Varieties of CSPs

- Continuous variables
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in polynomial time by linear programming

Varieties of constraints

- Unary constraints involve a single variable,
  - e.g., SA ≠ green

- Binary constraints involve pairs of variables,
  - e.g., SA ≠ WA

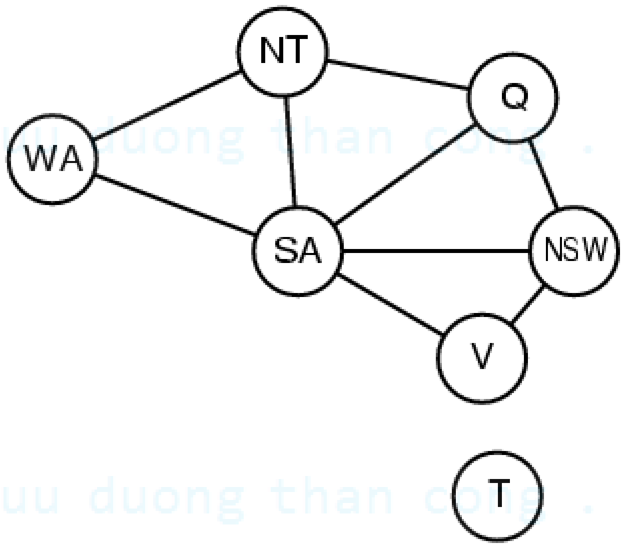
- Higher-order constraints involve 3 or more variables,
  - e.g., Professors A, B, and C cannot be on a committee together
  - Y = D + E or D + E - 10
  - Can always be represented by multiple binary constraints

Varieties of constraints

- Inequality constraints on continuous variables
  - $endjob_1 + 5 \leq startjob_3$
- Preference (soft constraints)
  - e.g. *red* is better than *green* often can be represented by a cost for each variable assignment
  - combination of optimization with CSPs
- the *Alldifferent* constraint forces all the variables it touches to have different values

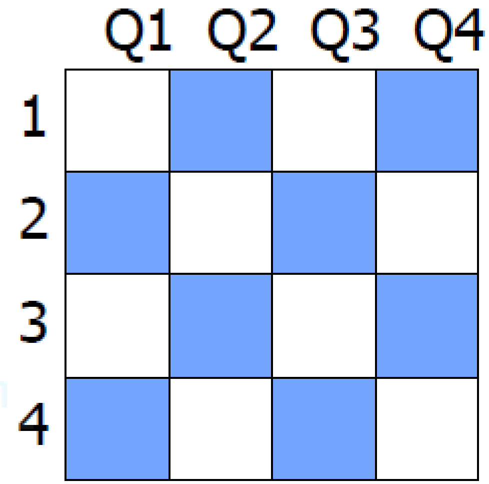
Constraint graph

- Constraint graph: nodes are variables, arcs are constraints



Example: 4-Queens Problem

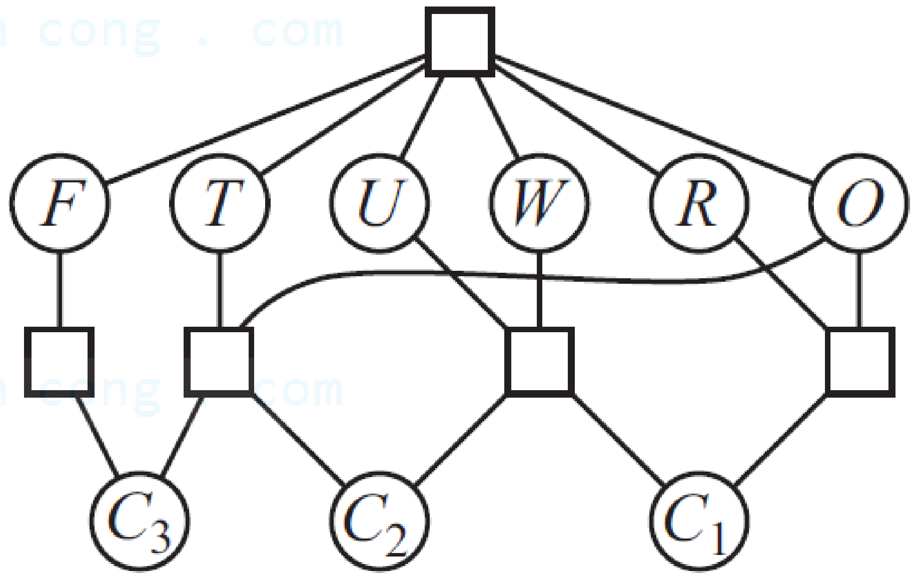
- **Variables**: $Q1, Q2, Q3, Q4$
- **Domains**: $\mathrm{D} = \{1,2,3,4\}$
- **Constraints:**
  - $Qi \neq Qj$ (cannot be in the same row)
  - $Qi - Qj \neq i - j$ (cannot be in the same diagonal)



Example: Cryptarithmetic

- **Variables**: $F\ T\ U\ W\ R\ O\ C_1\ C_2\ C_3$
- **Domains**: {0,1,2,3,4,5,6,7,8,9}
- **Constraints:**
  - $Alldiff(F, T, U, W, R, O)$
  - $C_3 = F, T \neq 0, F \neq 0$
  - …

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



CSP as a Search

Backtracking search

Forward checking

Standard search formulation (incremental)

- Let's start with the straightforward approach, then fix it
- States are defined by the values assigned so far
  - Initial state: the empty assignment { }
  - Successor function: assign a value to an unassigned variable that does not conflict with current assignment

    → *fail if no legal assignments*
  - Goal test: the current assignment is complete
- This is the same for all CSPs
  - Every solution appears at depth n with n variables

    → *use depth-first search*
  - If using BFS: b = (n - ℓ)d at depth ℓ, n! · d^n leaves even though there are only d^n complete assignments!

Backtracking search

- Variable assignments are commutative,
  - i.e., [WA = red then NT = green] same as [NT = green then WA = red]
- Only need to consider assignments to a single variable at each node

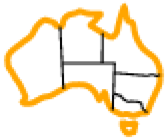
  → $b = d$ and there are d^n leaves
- Depth-first search for CSPs with single-variable assignments is called **backtracking** search
- Backtracking search is the basic uninformed algorithm for CSPs
  - Can solve n-queens for $n \approx 25$

Backtracking search

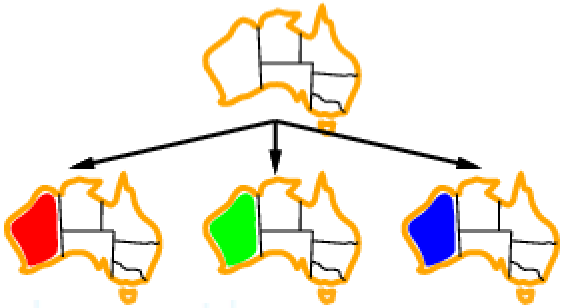
```
function BACKTRACKING-SEARCH(csp) return a solution or failure
    return RECURSIVE-BACKTRACKING({} , csp)

function RECURSIVE-BACKTRACKING(assignment, csp) return 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 ← RRECURSIVE-BACTRACKING(assignment, csp)
                    if result ≠ failure  then return result
                    remove {var=value} from assignment
    return failure
```

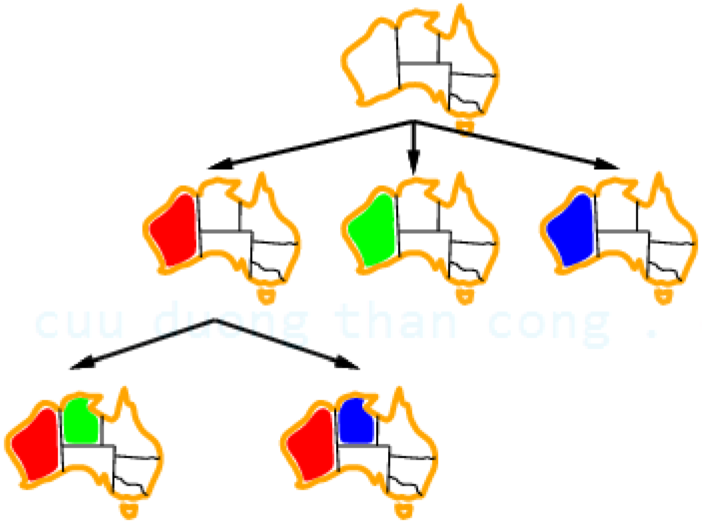
Backtracking example



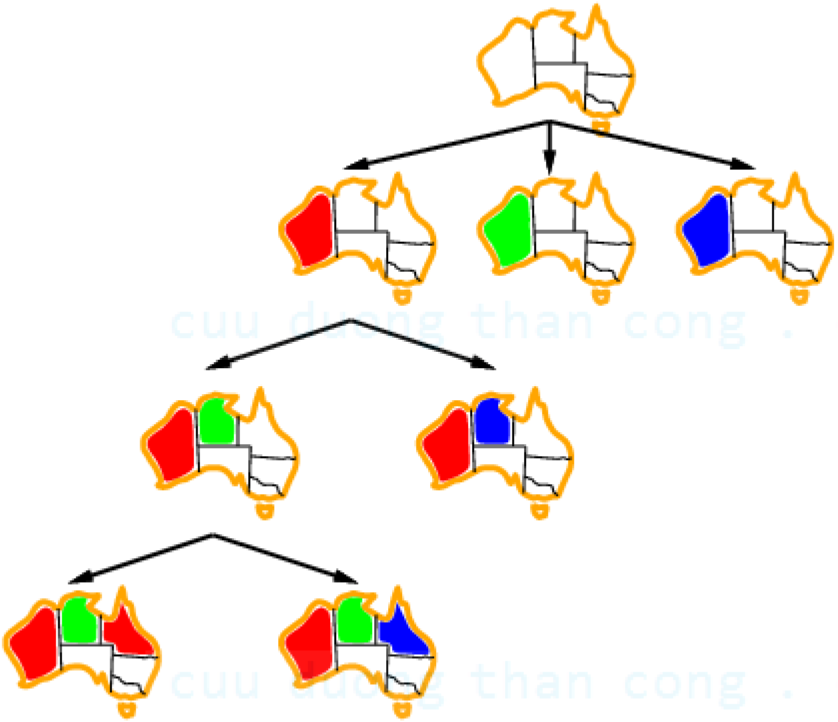
Backtracking example



Backtracking example



Backtracking example



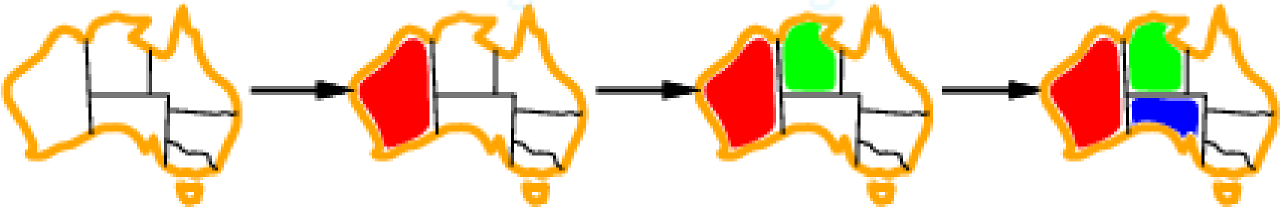
Improving backtracking efficiency

- 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?

- Using Heuristic Rules:
  - Minimum Remaining Value (MRV)
  - Degree Heuristic (DH)
  - Least Constraining Value (LCV)
  - ...

Most constrained variable

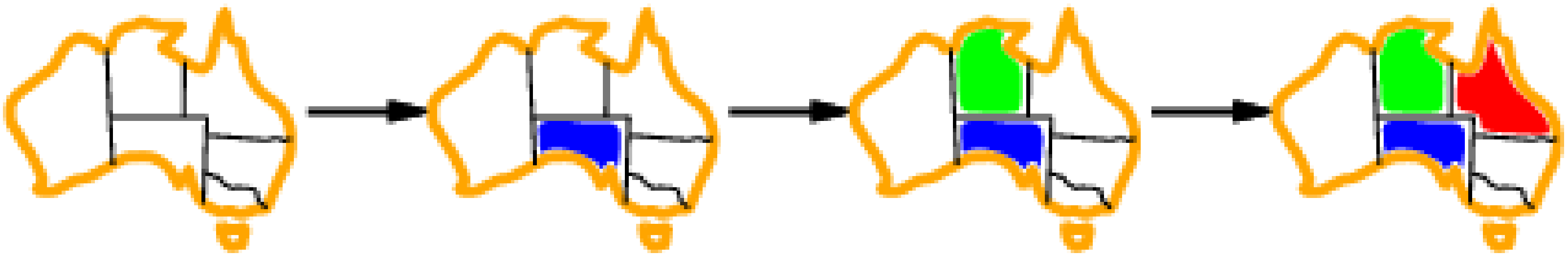
- Heuristic Rule:
  - choose the variable with the fewest legal values
    - E.g., will immediately detect failure if X has no legal values



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

Most constraining variable

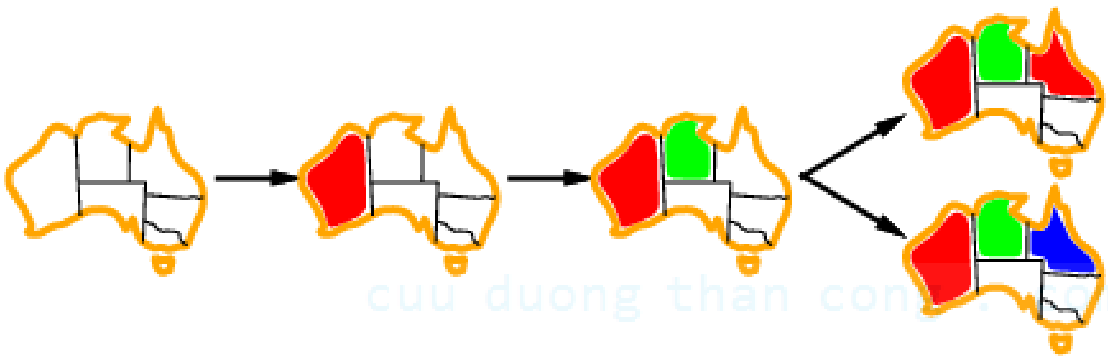
- ❑Tie-breaker among most constrained variables
- ❑Most constraining variable:
  - o choose the variable with the most constraints on remaining variables (most edges in graph)



- ❑a.k.a. Degree Heuristic (DH)

Least constraining value

- ❑Least constraining value heuristic:
  - o Given a variable, choose the least constraining value:
    - the one that rules out the fewest values in the remaining variables
    - leaves the maximum flexibility for subsequent variable assignments

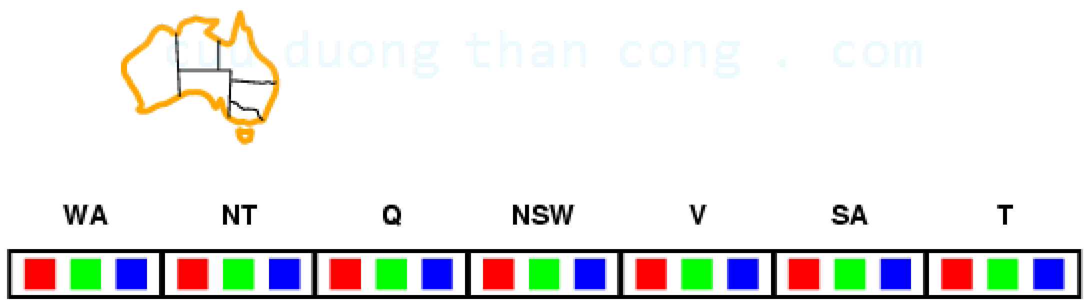


- ❑Combining these heuristics makes 1000 queens feasible

Forward checking

❑Idea:

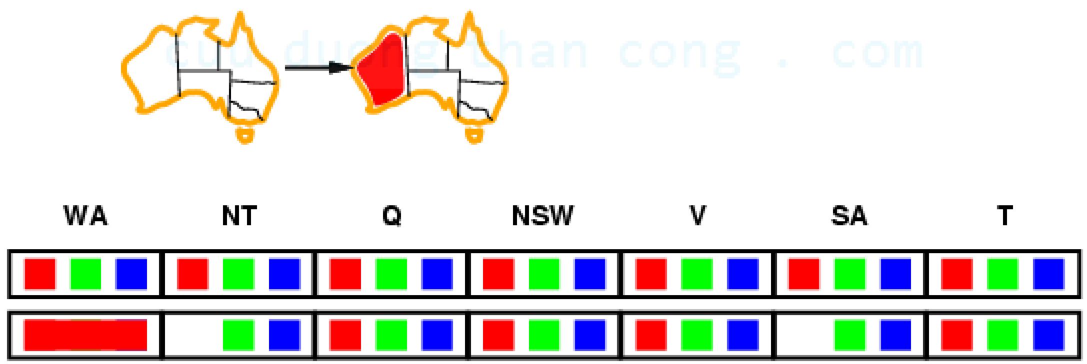
- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



Forward checking

❑Idea:

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

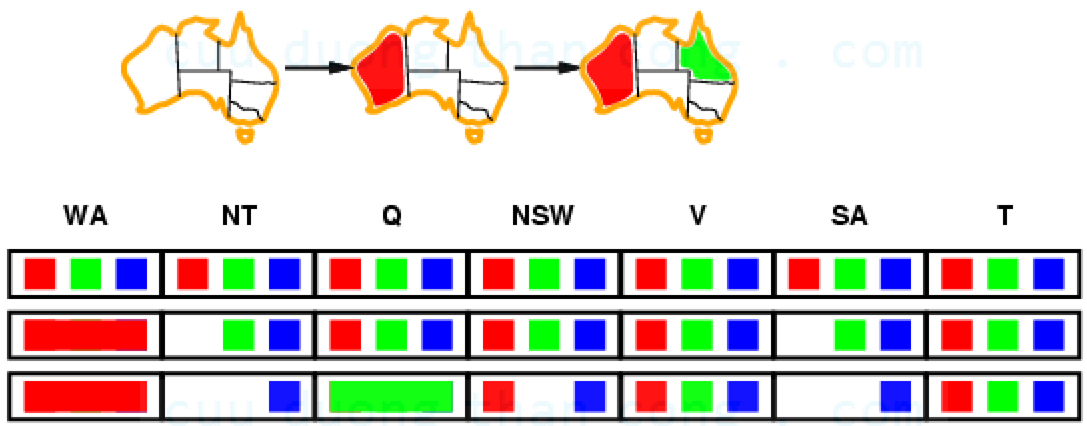


- ✓ Assign *{WA=red}*
- ✓ Effects on other variables connected by constraints to WA
  - – *NT can no longer be red*
  - – *SA can no longer be red*

Forward checking

❑Idea:

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



✓ Assign *{Q=green}*

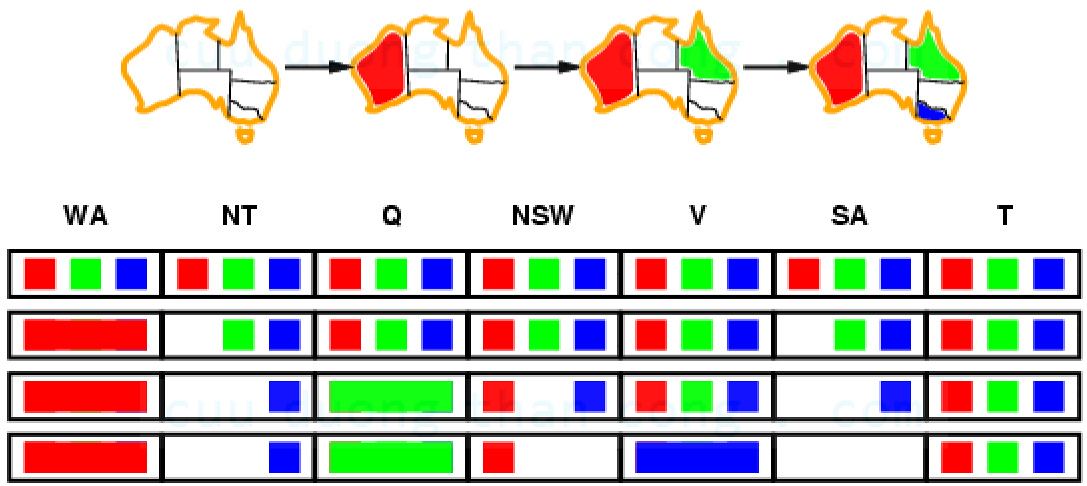
✓ Effects on other variables connected by constraints to Q

- *NT can no longer be green*
- *SA can no longer be green*
- *NSW can no longer be green*

→ *MRV heuristic* would automatically select NT or SA next

Forward checking

FC has detected that partial assignment is *inconsistent* with the constraints and backtracking can occur.

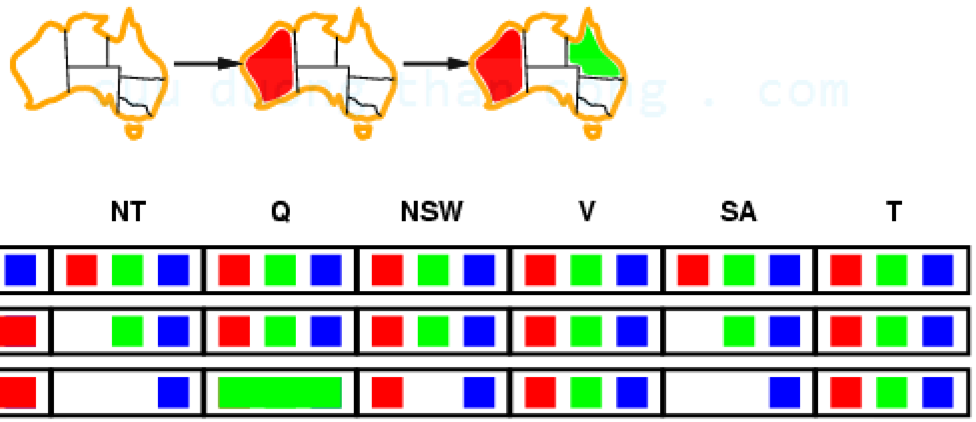


- ✓ Assign *{V=blue}*
- ✓ Effects on other variables connected by constraints to *V*
  - *NSW can no longer be blue*
  - *SA is empty*

Constraint Propagation: Inference in CSPs

Constraint propagation

- Forward checking propagates information from assigned to unassigned variables. How about constraints between two unassigned states?



  - NT and SA cannot both be blue!
  - Constraint propagation repeatedly enforces constraints locally

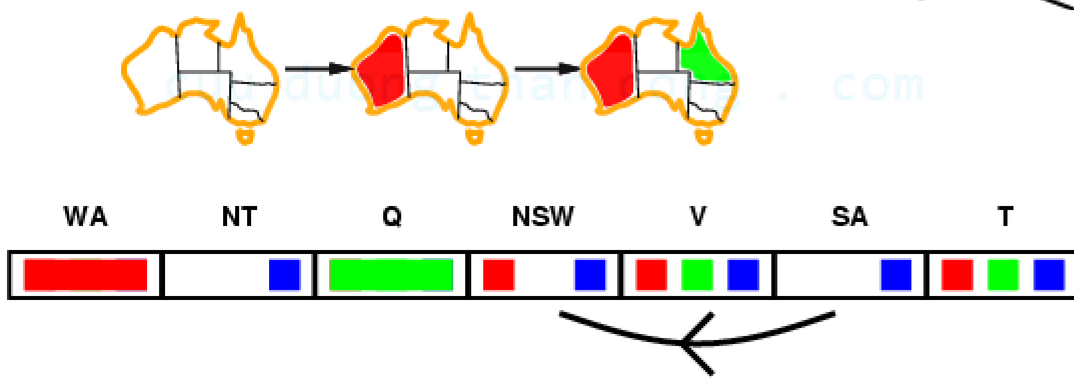
Inference in CSPs

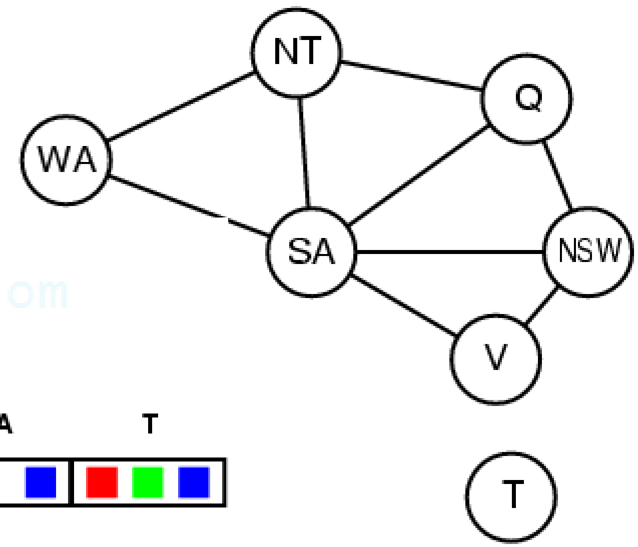
- In regular state-space search: Search only
- In CSPs: Search or Inference
- **Constraint propagation**:
  - Using constraints to *reduce number of legal values* for a variable, which in turn can reduce the legal values for another variable, and so on.
  - Can be intertwined with search, or a preprocessing step
  - Sometimes this preprocessing can solve the whole problem
- Idea: **local consistency**
  - Node: variable
  - Arc: binary constraint

  → *Enforcing local consistency will eliminate inconsistent values*

Arc consistency

- An arc $X \rightarrow Y$ is consistent iff for every value x of X there is some allowed y



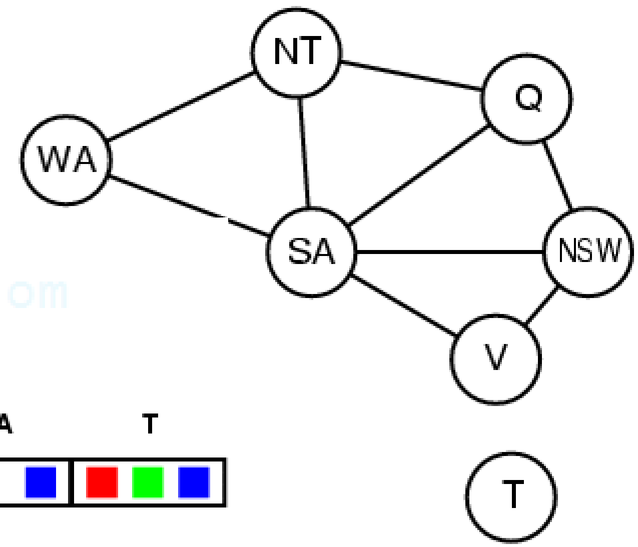


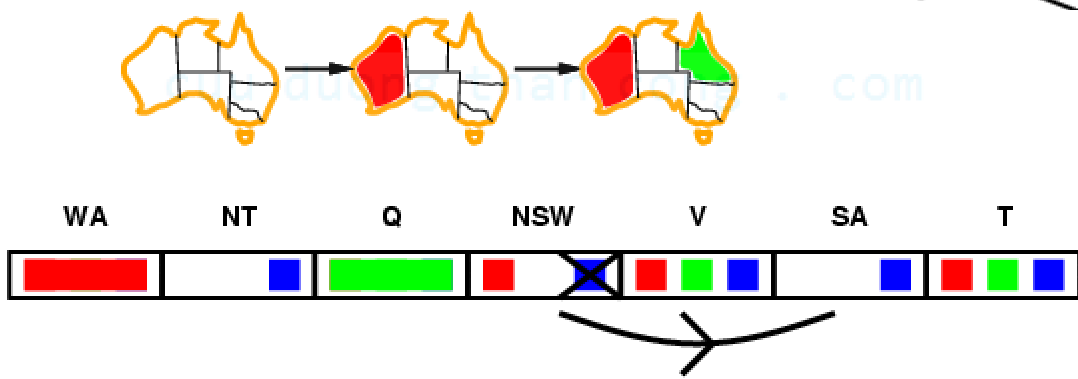
Consider state of search after *WA* and *Q* are assigned:

- $SA \rightarrow NSW$ is consistent if SA=*blue* and NSW=*red*

Arc consistency

- An arc $X \rightarrow Y$ is consistent iff for every value x of X there is some allowed y





$NSW \rightarrow SA$ is consistent if

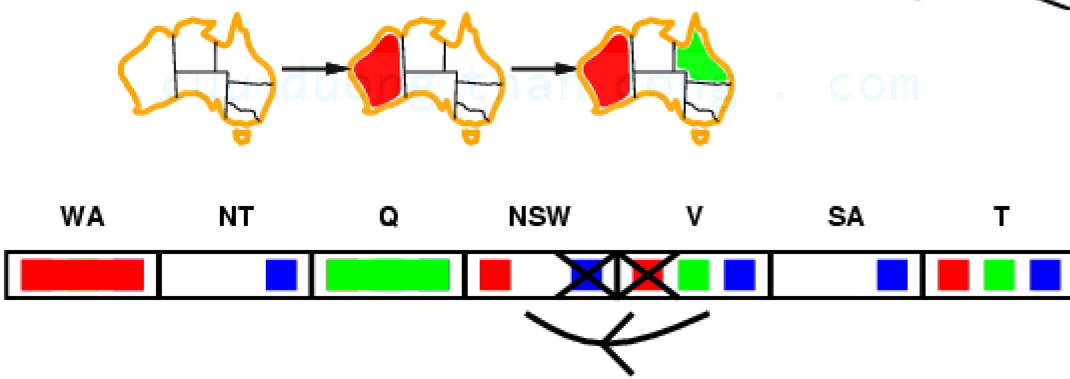
NSW=red and *SA=blue*

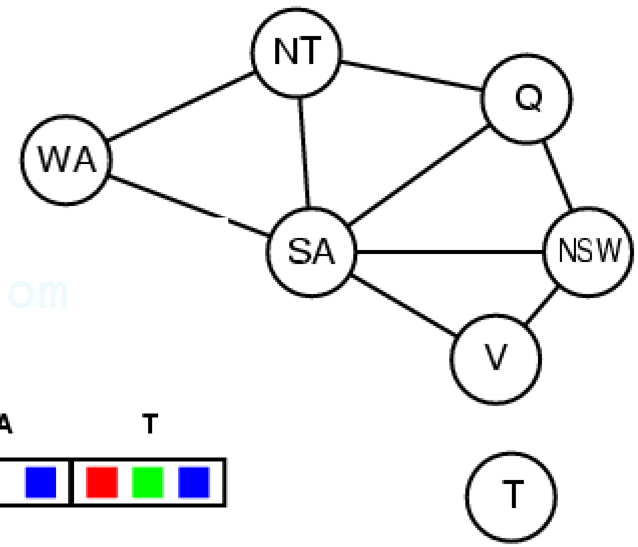
NSW=blue and SA=???

Arc can be made consistent by removing *blue* from *NSW*

Arc consistency

- ❑An arc $X \rightarrow Y$ 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

Continue to propagate constraints….

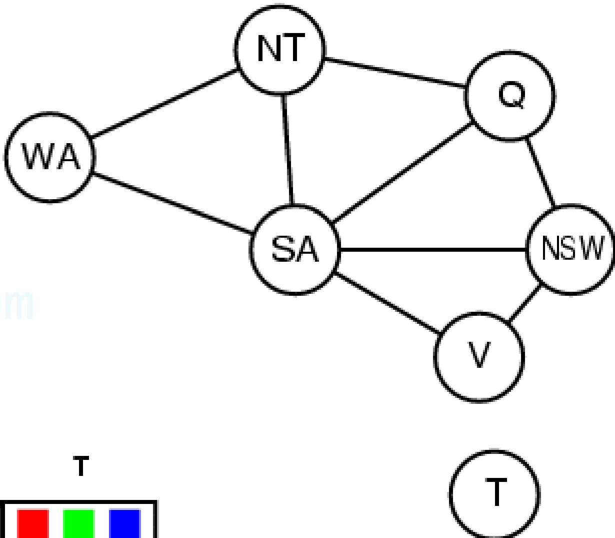
– Check $V \rightarrow NSW$

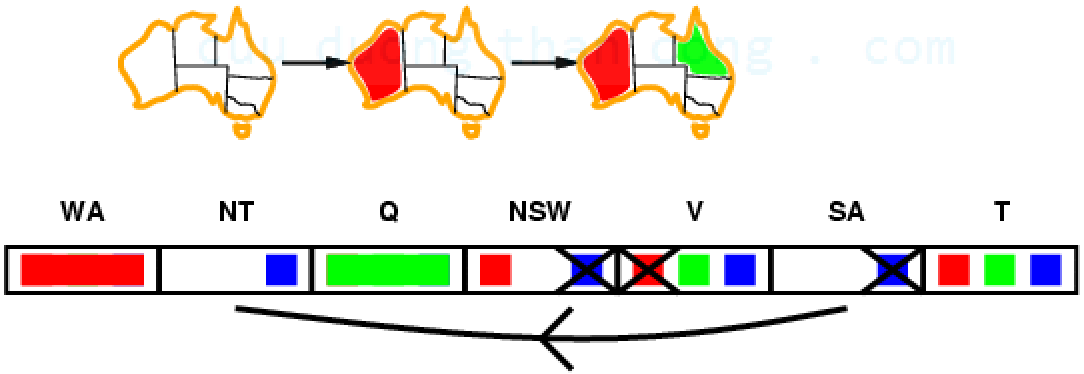
– Not consistent for V = red

– Remove red from V

Arc consistency

- An arc $X \rightarrow Y$ 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

Arc consistency vs Forward checking

- Given a constraint C_{XY} between two variables X and Y, for any value of X, there is a consistent value that can be chosen for Y such that C_{XY} is satisfied, and **visa versa**.
- Thus, unlike forward checking, arc consistency is directed and is checked in both directions for two connected variables.

→ *This makes it stronger than forward checking.*

Arc consistency

- ❑Can be run as a preprocessor or after each assignment
  - Or as preprocessing before search starts
- ❑AC must be run repeatedly until no inconsistency remains
- ❑Trade-off
  - Requires some overhead to do, but generally more effective than direct search
  - It can eliminate large (inconsistent) parts of the state space more effectively than search can
- ❑Need a systematic method for arc-checking
  - If X loses a value, neighbors of X need to be rechecked:
  - i.e. incoming arcs can become inconsistent again (outgoing arcs will stay consistent).

Arc consistency algorithm AC-3

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 $revised$

Time complexity: O(cd^3)

Local search for CSPs

- Hill-climbing, simulated annealing typically work with "complete" states, i.e., all variables assigned
- To apply to CSPs:
  - allow states with unsatisfied constraints
  - operators reassign variable values
- Variable selection: randomly select any conflicted variable
- Value selection by min-conflicts heuristic:
  - choose value that violates the fewest constraints
  - i.e., hill-climb with *h(n)* = total number of violated constraints

MIN-CONFLICT Algorithm

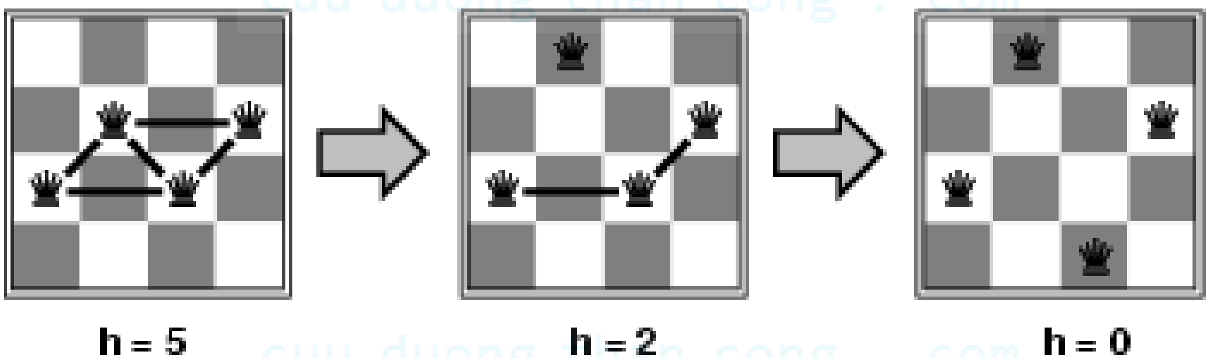
function MIN-CONFLICTS(csp, max_steps) **returns** a solution or failure
  inputs: csp, a constraint satisfaction problem
    max_steps, the number of steps allowed before giving up

  $current \leftarrow$ an initial complete assignment for csp
  for i = 1 to max_steps **do**
    if $current$ is a solution for csp **then return** $current$
    $var \leftarrow$ a randomly chosen conflicted variable from csp.VARIABLES
    $value \leftarrow$ the value v for var that minimizes CONFLICTS(var, v, $current$, csp)
    set $var = value$ in $current$
  return $failure$

Count the number of constraints violated by a particular value, given the rest of the current assignment

Example: 4-Queens

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



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

Summary

- CSPs are a special kind of problem:
  - states defined by values of a fixed set of variables
  - goal test 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 guarantee later failure
- Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies
- Iterative min-conflicts is usually effective in practice

Next week

- Individual Assignment 3 (I3)
- Chapter 3: Knowledge Representation and Reasoning
  - Propositional Logic
  - First Order Logic