Constraint Satisfaction

Russell & Norvig Ch. 6.1-6.4

Informal Definition of CSP

- CSP = Constraint Satisfaction Problem
- Given

  (1) a finite set of variables

  (2) each with a domain of possible values (often finite)

  (3) a set of constraints that limit the values the variables can take on

- A **solution** is an assignment of a value to each variable such that the constraints are all satisfied.

- Tasks might be to decide if a solution exists, to find a solution, to find all solutions, or to find the "best solution" according to some metric (objective function).

Today's Class

- Constraint Processing / Constraint Satisfaction Problem (CSP) paradigm
- Algorithms for CSPs
  - Backtracking (systematic search)
  - Constraint propagation (k-consistency)
  - Variable and value ordering heuristics
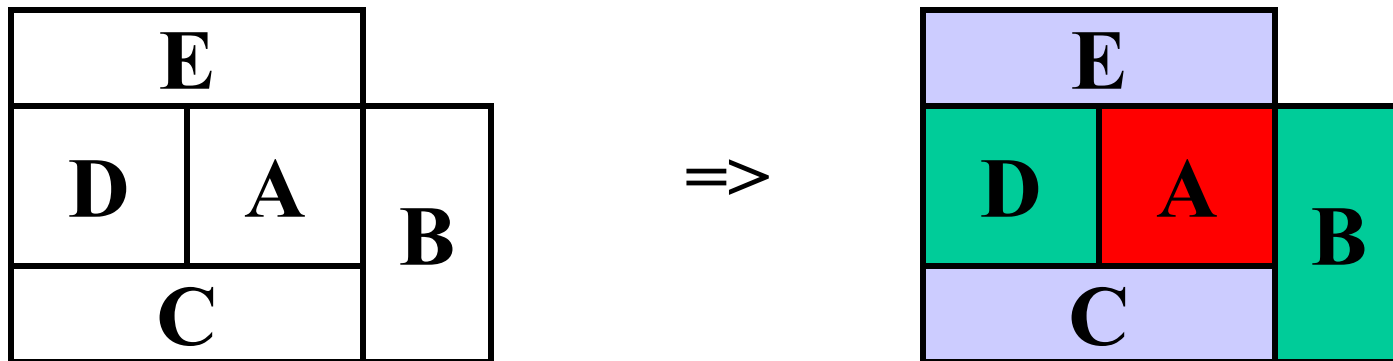 - Intelligent backtracking

Informal Example: Map Coloring

- Color the following map using three colors (red, green, blue) such that no two adjacent regions have the same color.
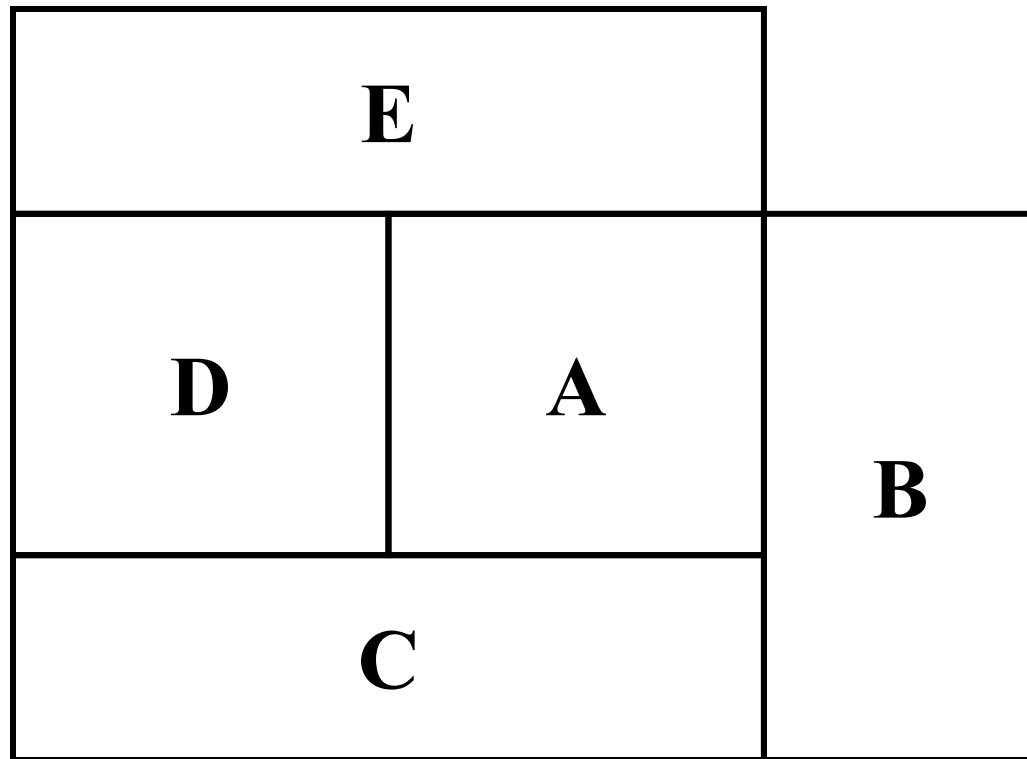
Map Coloring II

- Variables: A, B, C, D, E all of domain RGB
- Domains: RGB = {red, green, blue}
- Constraints: A≠B, A≠C,A ≠ E, A ≠ D, B ≠ C, C ≠ D, D ≠ E
- One solution: A=red, B=green, C=blue, D=green, E=blue

Formal Definition of a Constraint Network (CN)

A constraint network (CN) consists of

- a set of variables $X = \{x_1, x_2, \ldots x_n\}$
  - each with an associated domain of values $\{d_1, d_2, \ldots d_n\}$.
  - the domains are typically finite

- a set of constraints $\{c_1, c_2 \ldots c_m\}$ where
  - each constraint defines a predicate which is a relation over a particular subset of X.
  - e.g., C_i involves variables $\{X_{i1}, X_{i2}, \ldots X_{ik}\}$ and defines the relation $R_i \subseteq D_{i1} \times D_{i2} \times \ldots D_{ik}$

- **Unary** constraint: only involves one variable
- **Binary** constraint: only involves two variables

Example (Class Scheduling)

- Given a list of courses to be taught, classrooms available, time slots, and professors who can teach certain courses, can classes be scheduled?

- Variables: Courses offered $(C_1, \ldots, C_i)$, classrooms $(R_1, \ldots, R_j)$, time $(T_1, \ldots, T_k)$.

- Domains:
  - DC_i = {professors who can teach course i}
  - DR_j = {room numbers}
  - DT_k = {time slots}

- Constraints:
  - Maximum 1 class per room in each time slot.
  - A professor cannot teach 2 classes in the same time slot.
  - A professor cannot teach more than 2 classes.

Typical Tasks for CSP

- Solutions:
  - Does a solution exist?
  - Find one solution
  - Find all solutions
  - Given a partial instantiation, do any of the above
- Transform the CN into an equivalent CN that is easier to solve.

Solving Constraint Problems

- Systematic search
  - Generate and test
  - Backtracking
- Constraint propagation (consistency)
- Variable ordering heuristics
- Value ordering heuristics
- Backjumping and dependency-directed backtracking

Systematic Search: Backtracking
(a.k.a. depth-first search!)

- Consider the variables in some order

- Pick an unassigned variable and give it a provisional value such that it is consistent with all of the constraints

- If no such assignment can be made, we've reached a dead end and need to backtrack to the previous variable

- Continue this process until a solution is found or we backtrack to the initial variable and have exhausted all possible values

Problems with Backtracking

- Thrashing: keep repeating the same failed variable assignments
  - Consistency checking can help
  - Intelligent backtracking schemes can also help
- Inefficiency: can explore areas of the search space that aren't likely to succeed
  - Variable ordering can help

Consistency

- Node consistency
  - A node X is **node-consistent** if every value in the domain of X is consistent with X's unary constraints
  - A graph is node-consistent if all nodes are node-consistent
- Arc consistency
  - An arc (X, Y) is **arc-consistent** if, for every value x of X, there is a value y for Y that satisfies the constraint represented by the arc.
  - A graph is arc-consistent if all arcs are arc-consistent.
- To create arc consistency, we perform **constraint propagation**: that is, we repeatedly reduce the domain of each variable to be consistent with its arcs

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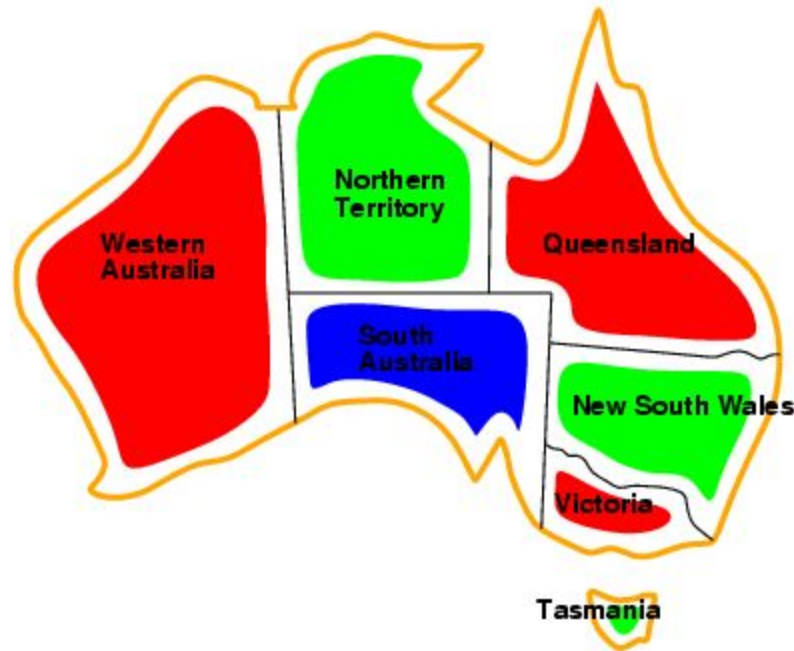
- Solutions are complete and consistent assignments
- e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

Constraint graph

- Binary CSP: each constraint relates two variables
- Constraint graph: nodes are variables, arcs are constraints
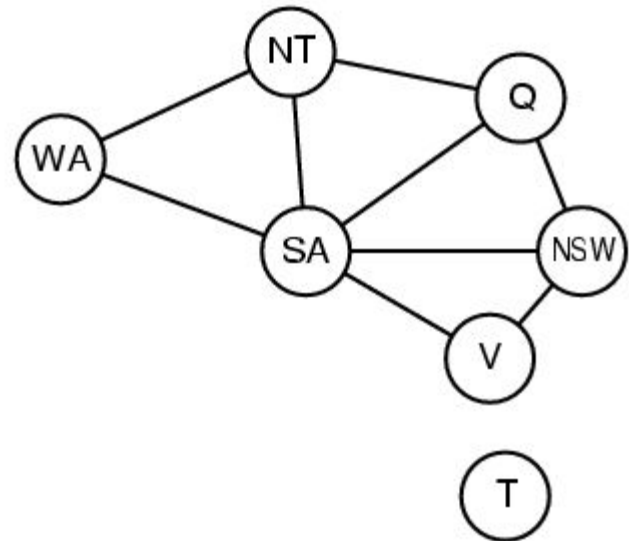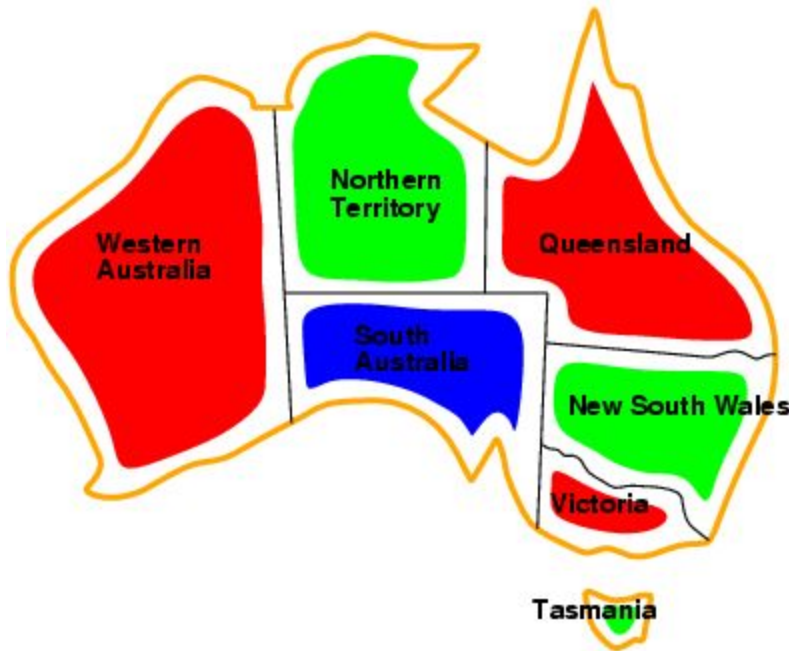
Varieties of constraints

- **Unary** constraints involve a single variable,
  - e.g., SA $\neq$ green

- **Binary** constraints involve pairs of variables,
  - e.g., SA $\neq$ WA

- **Higher-order** constraints involve 3 or more variables,
  - e.g., cryptarithmetic column constraints

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

- Depth-first search for CSPs with single-variable assignments is called backtracking search

- Can solve n-queens for $n \approx 25$

Backtracking search

```
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 example
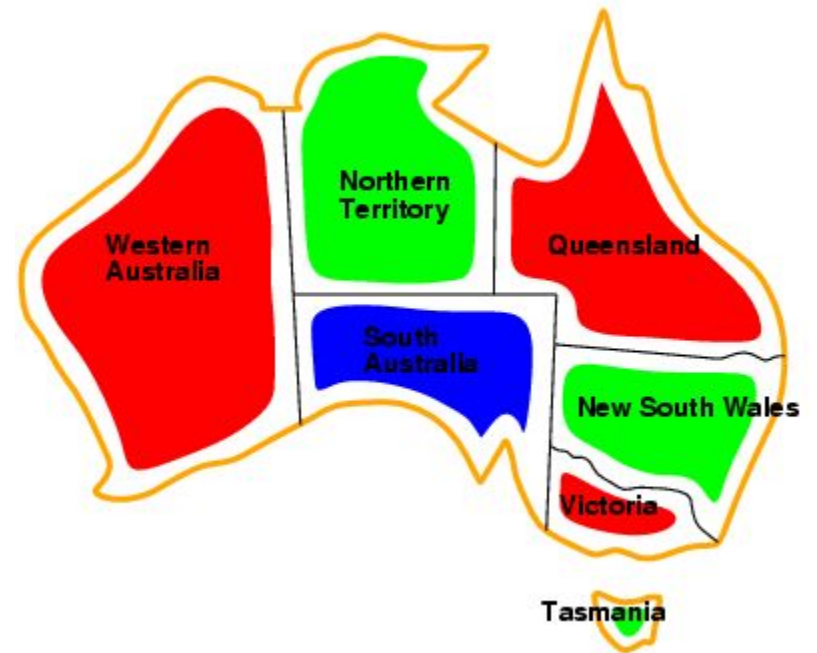
Backtracking example
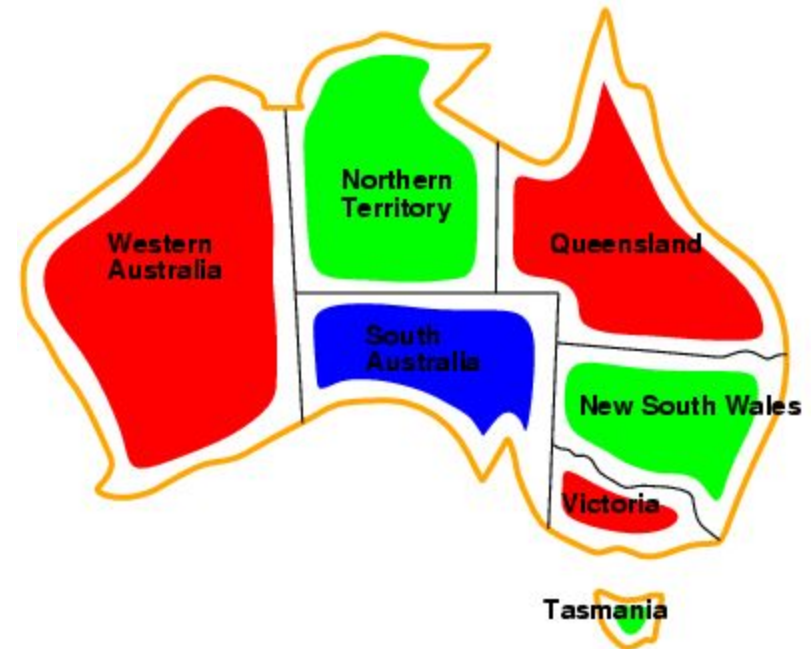
Backtracking example
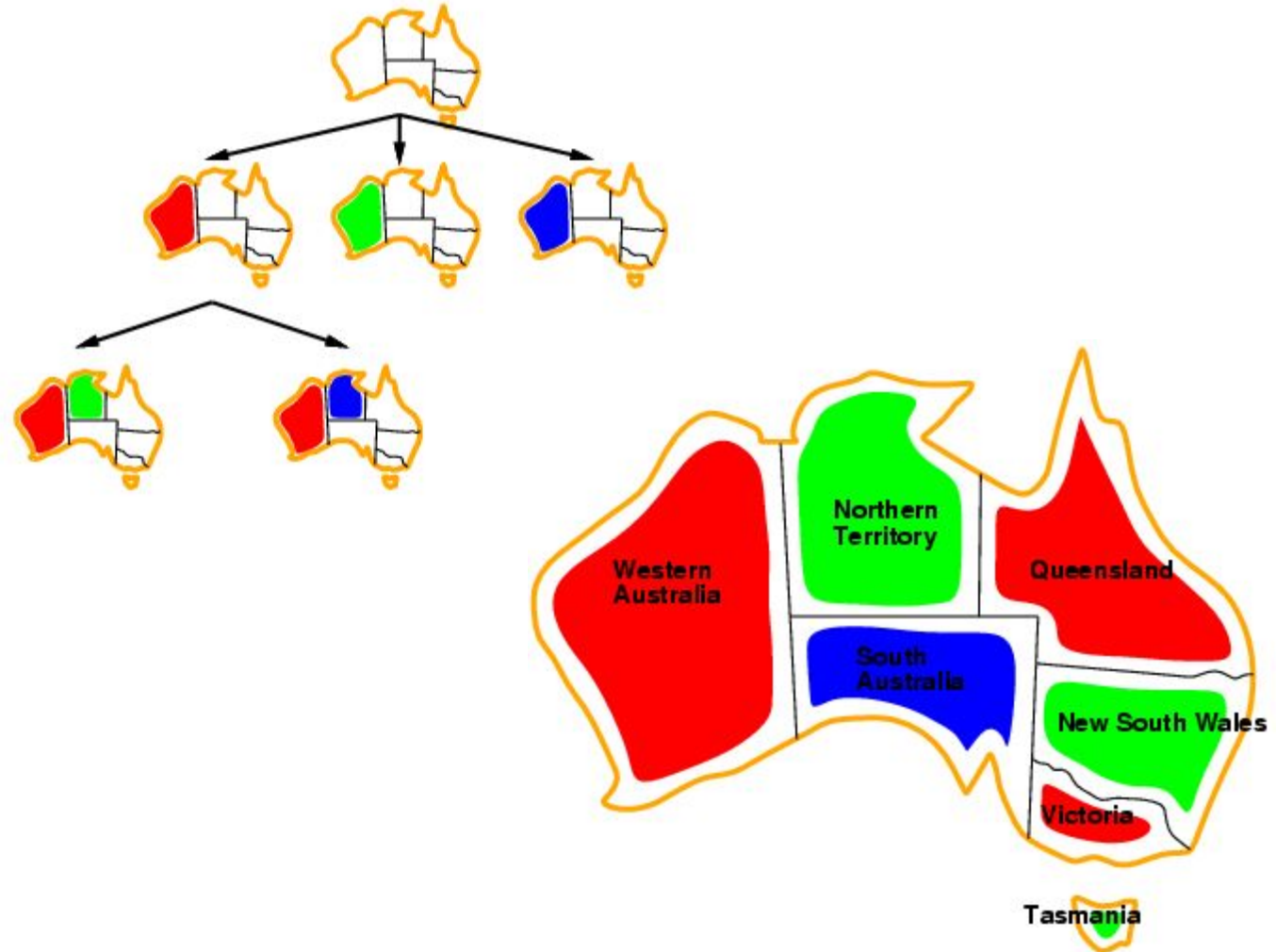
Backtracking example
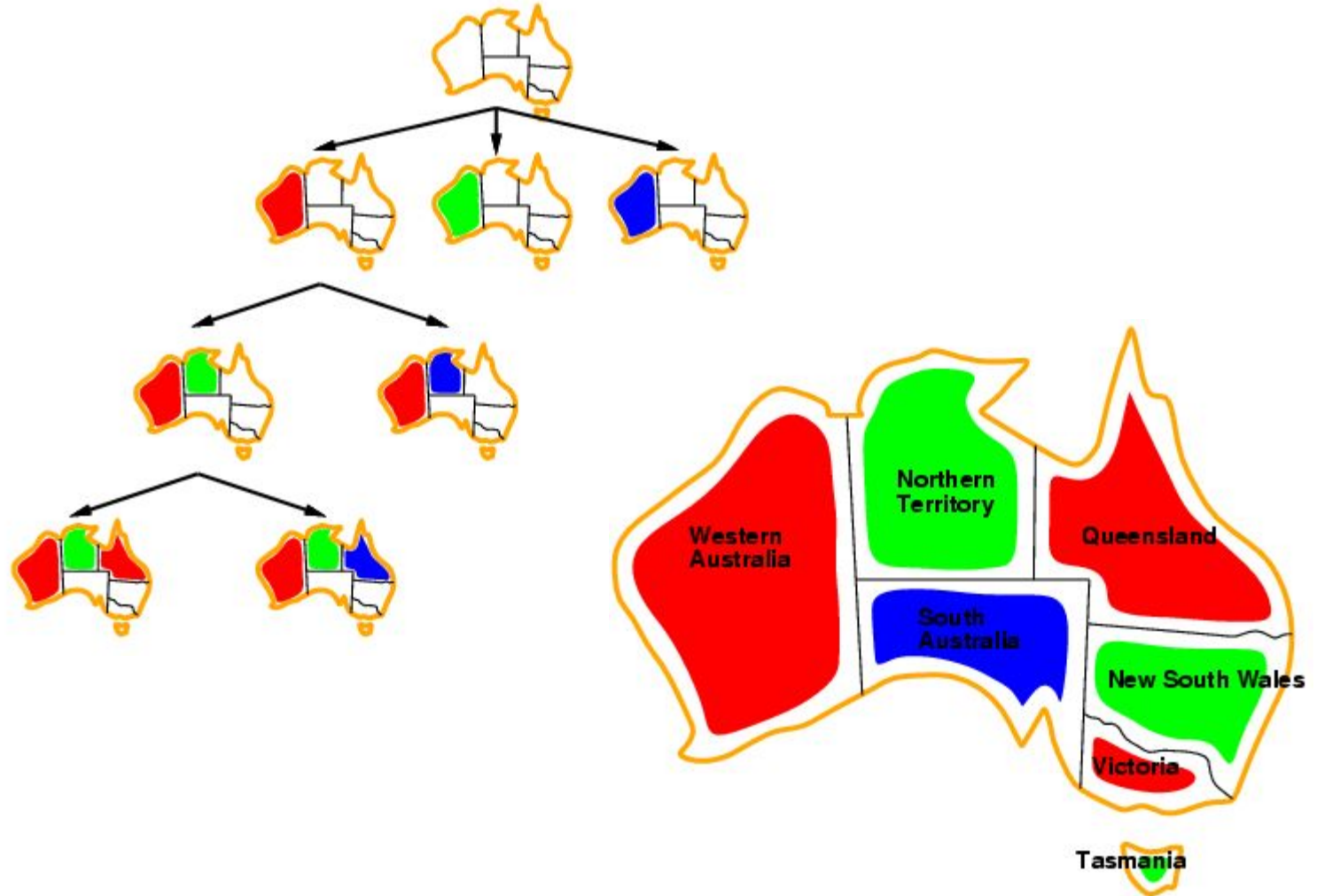
Improving backtracking efficiency

- 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?

Variable and Value Selection

- Selecting variables and assigning values using a static list is not always the most efficient approach.
    - Difficult to make the "right" choice for picking and setting the next variable.

HEURISTICS can help here, e.g.,

- "Minimum remaining values" heuristic – choose the variable with the smallest number of remaining values in its domain.
    - Also called "most constrained variable" heuristic

- "Degree heuristic" – choose the variable that is part of the most remaining unsatisfied constraints.
    - Useful to select first variable to assign.

- "Least-constraining-value" heuristic – once a variable is chosen, choose its value as the one that rules out the fewest choices for neighboring variables.
    - Keeps maximum flexibility for future variable assignments.

Most constrained variable

- Most constrained variable:

  choose the variable with the fewest legal values

- a.k.a. minimum remaining values (MRV) heuristic
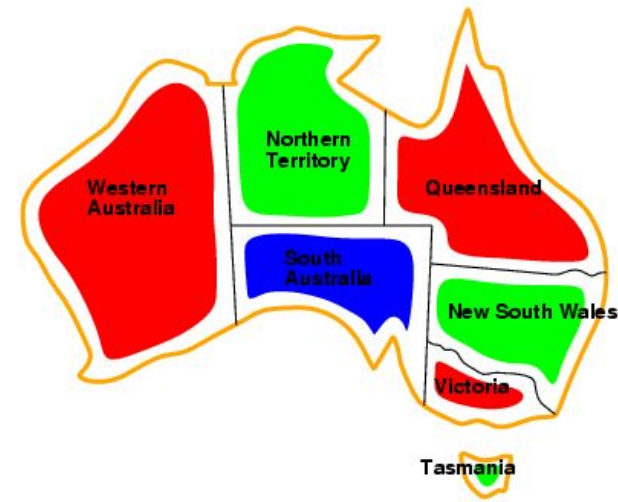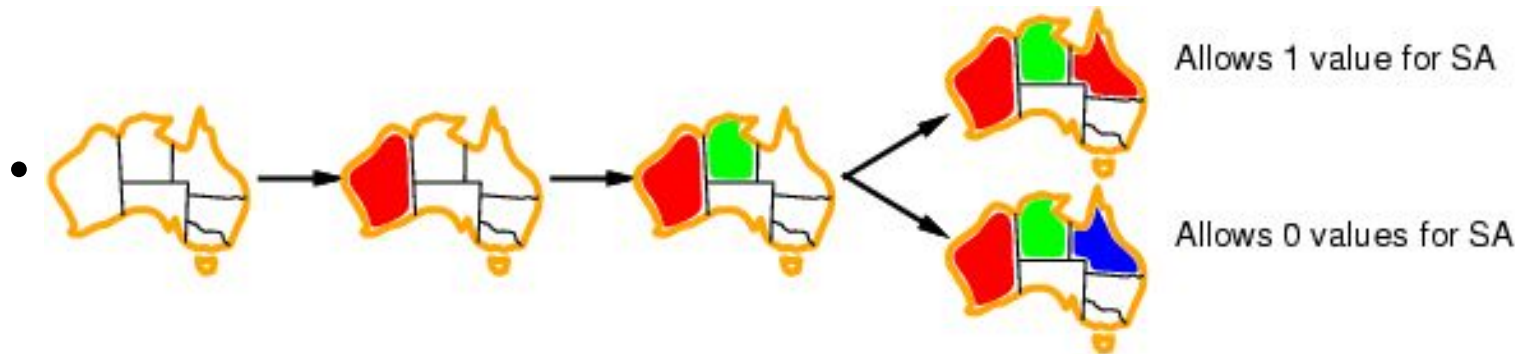
Degree Heuristic
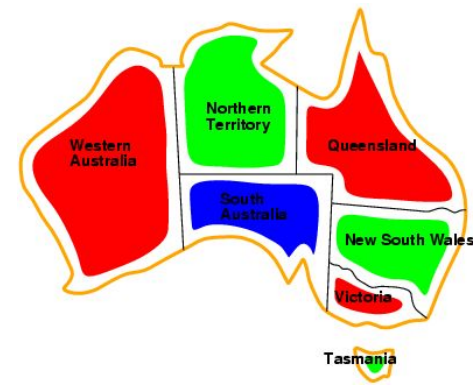
- A good idea is to use it as a tie-breaker among most constrained variables

- Most constraining variable:
  - choose the variable with the most constraints on remaining variables
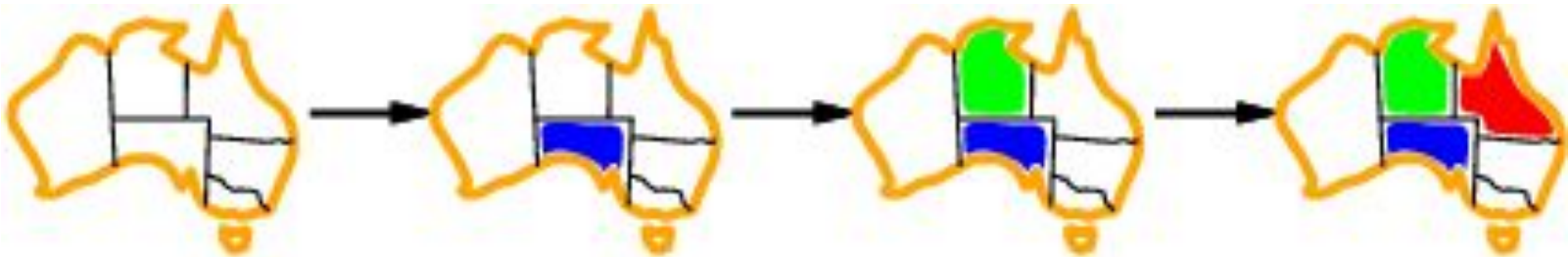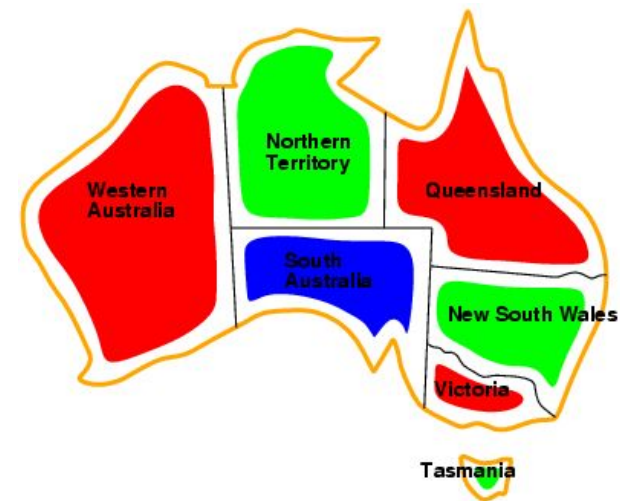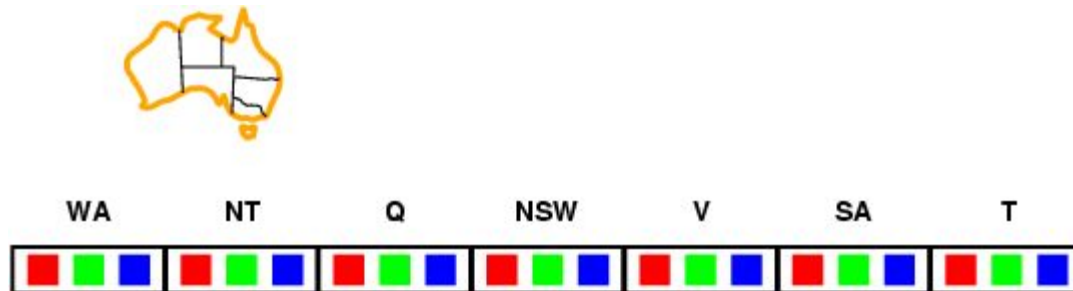
Least constraining value

- Given a variable to assign, choose the least constraining value:
  - the one that rules out the fewest values in the remaining variables

Allows 1 value for SA

Allows 0 values for SA

Forward checking

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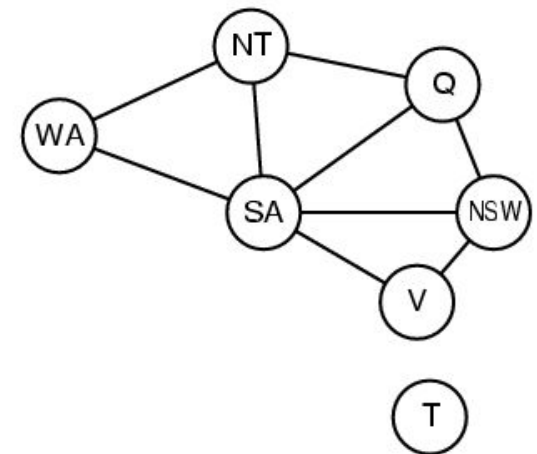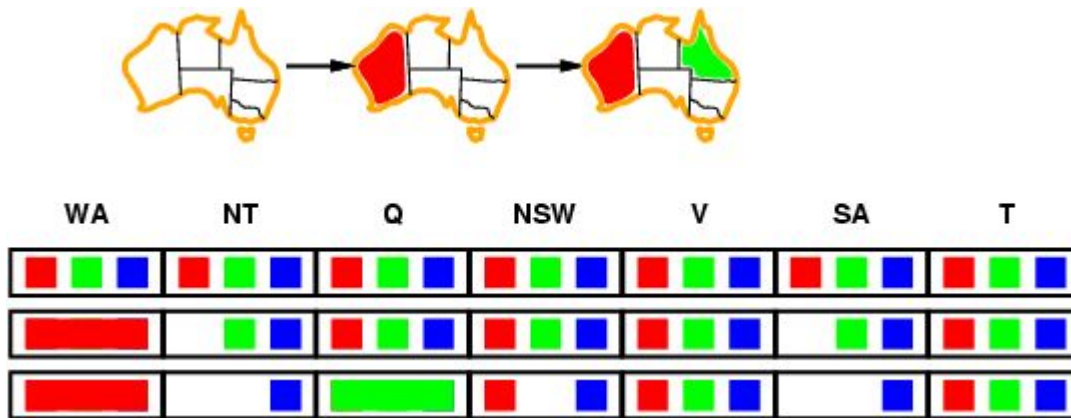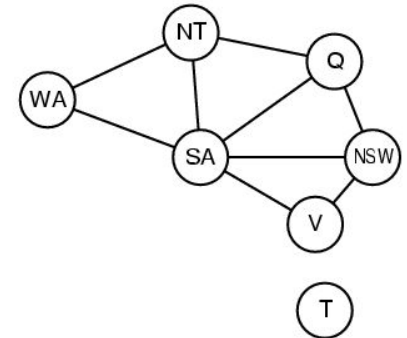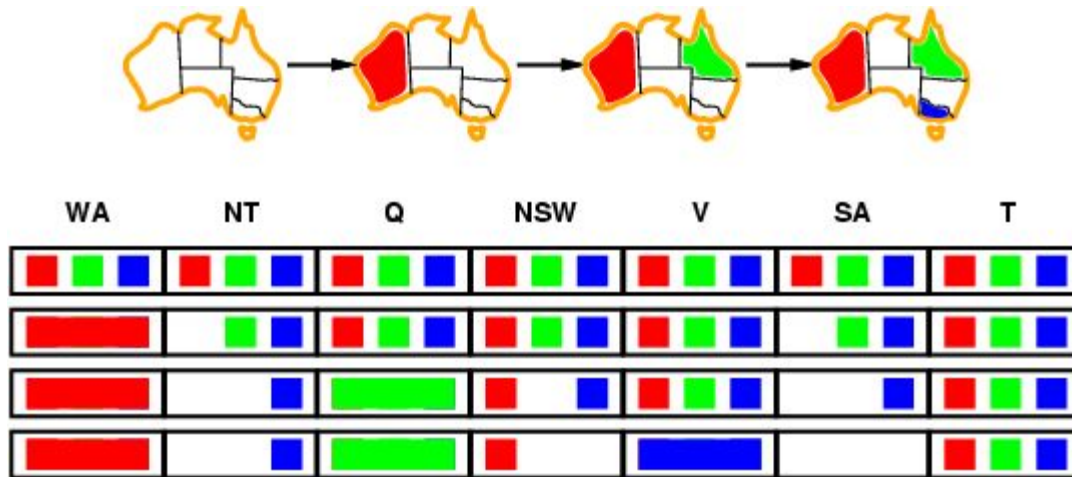
Forward checking

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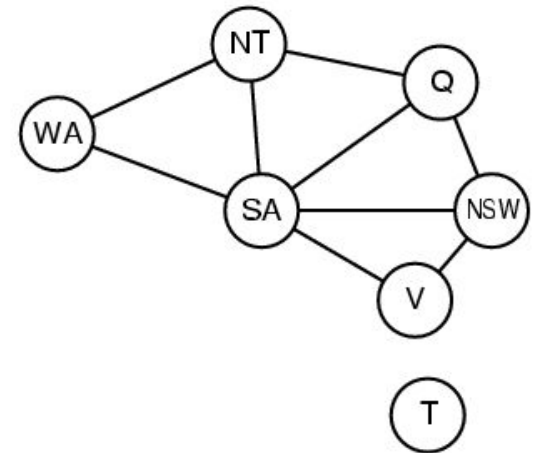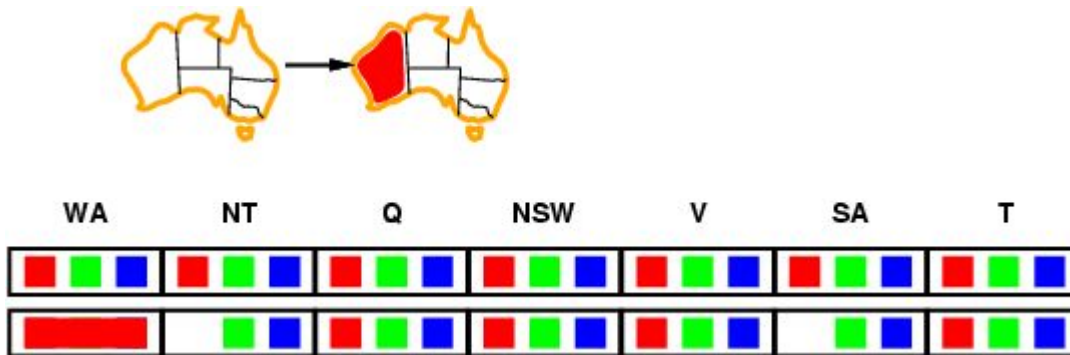
Forward checking

- 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

	WA	NT	Q	NSW	V	SA	T
Domains	RGB	RGB	RGB	RGB	RGB	RGB	RGB
After WA	R	GB	RGB	RGB	RGB	GB	RGB
After Q	R	B	G	RB	RGB	B	RGB
After V	R	B	G	R	B	✗	RGB

Constraint propagation

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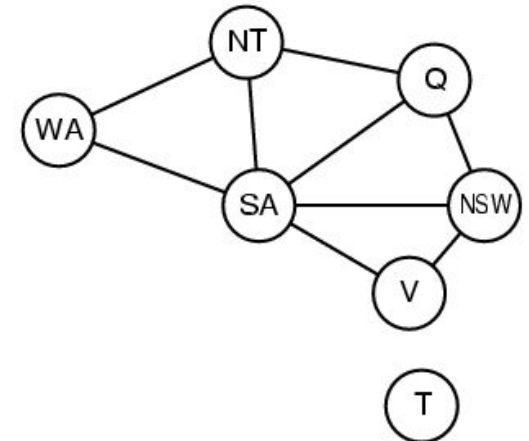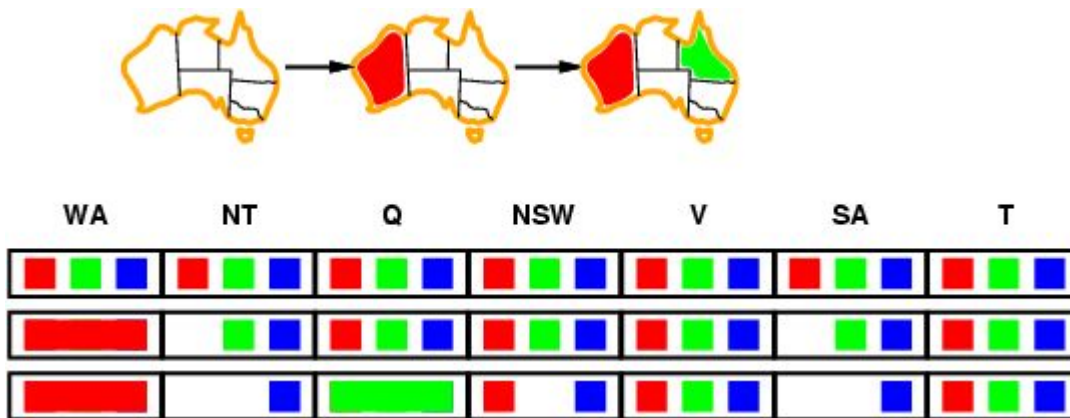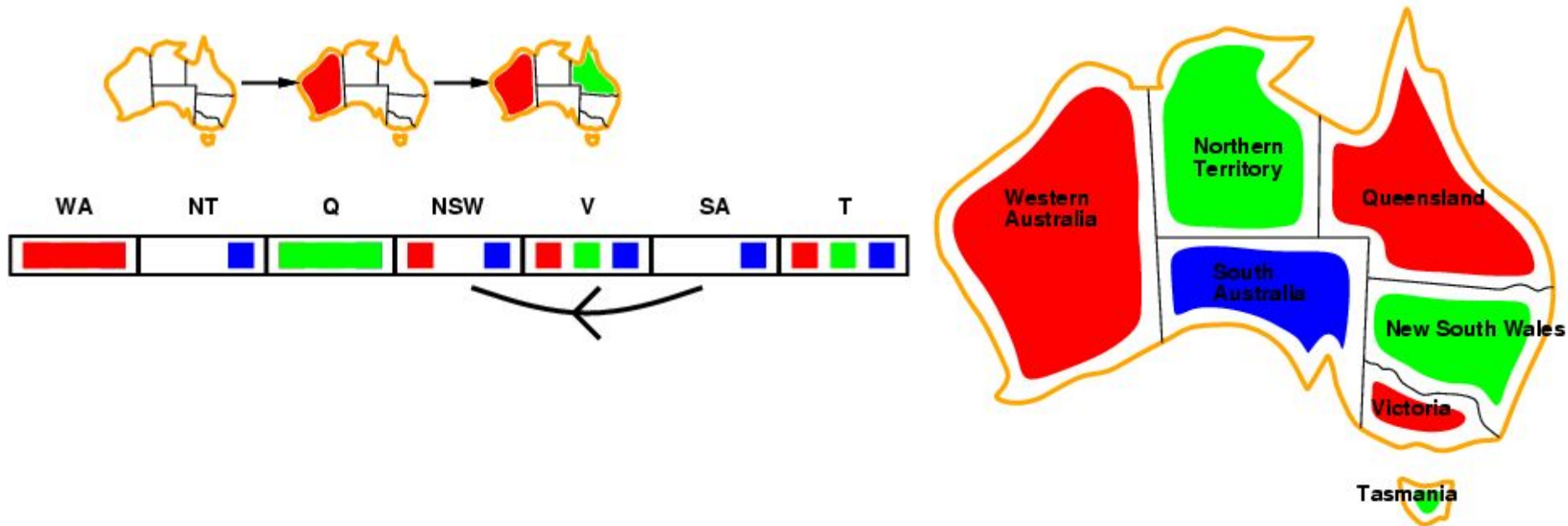

- NT and SA cannot both be blue!

- Constraint propagation algorithms repeatedly enforce constraints locally…

Arc consistency

- Simplest form of propagation makes each arc consistent
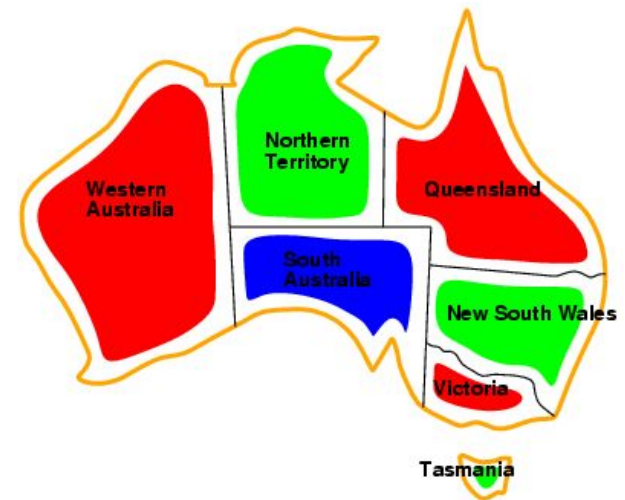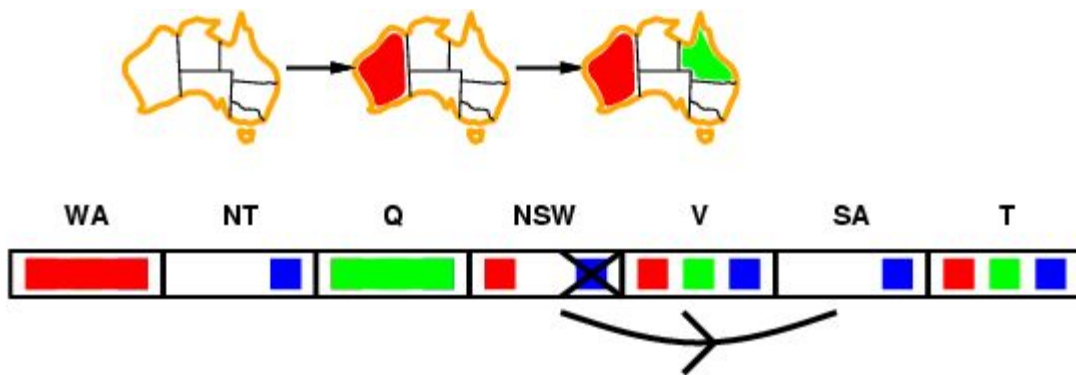- $X \rightarrow Y$ is consistent iff

  for every value x of X there is some allowed y

Arc consistency

- Simplest form of propagation makes each arc consistent
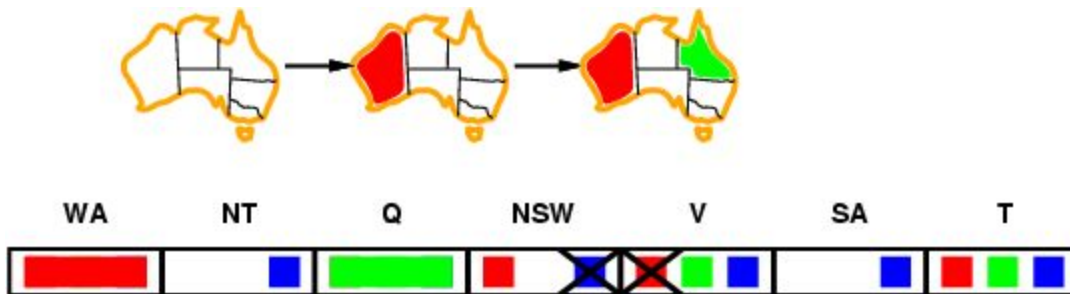- $X \square Y$ is consistent iff

  for every value x of X there is some allowed y

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

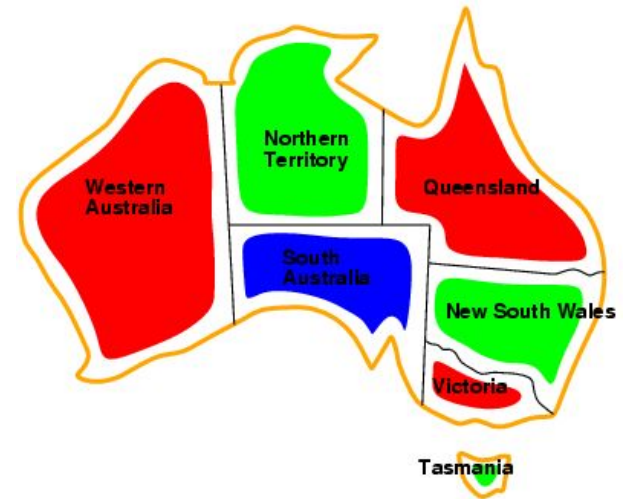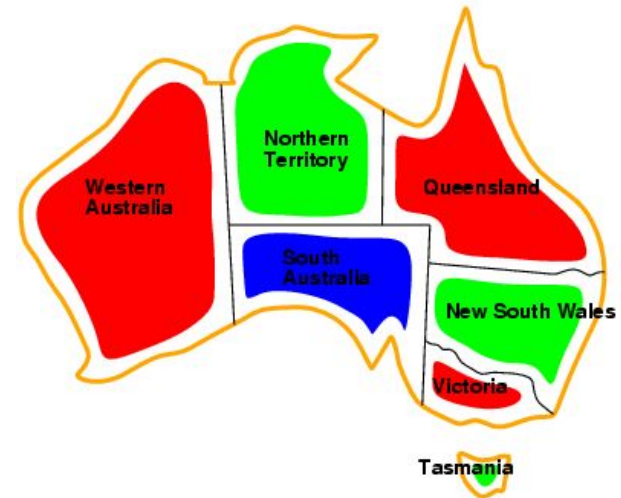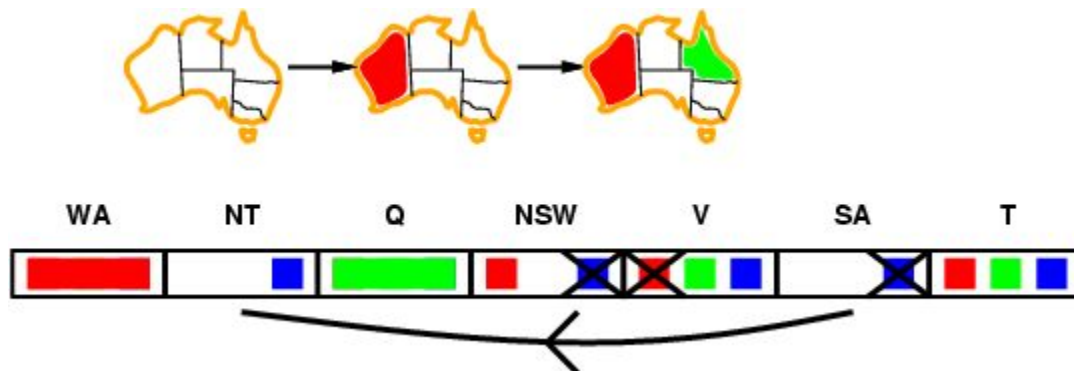If X loses a value, neighbors of X need to be rechecked

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

- 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 $\{X_1, X_2, \ldots, X_n\}$
    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 RM-INCONSISTENT-VALUES(X_i, X_j) then
            for each X_k in NEIGHBORS[X_i] do
                add (X_k, X_i) to $queue$

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

- Time complexity: $O(\#\text{constraints} \, |\text{domain}|^3)$

Checking consistency of an arc is $O(|\text{domain}|^2)$