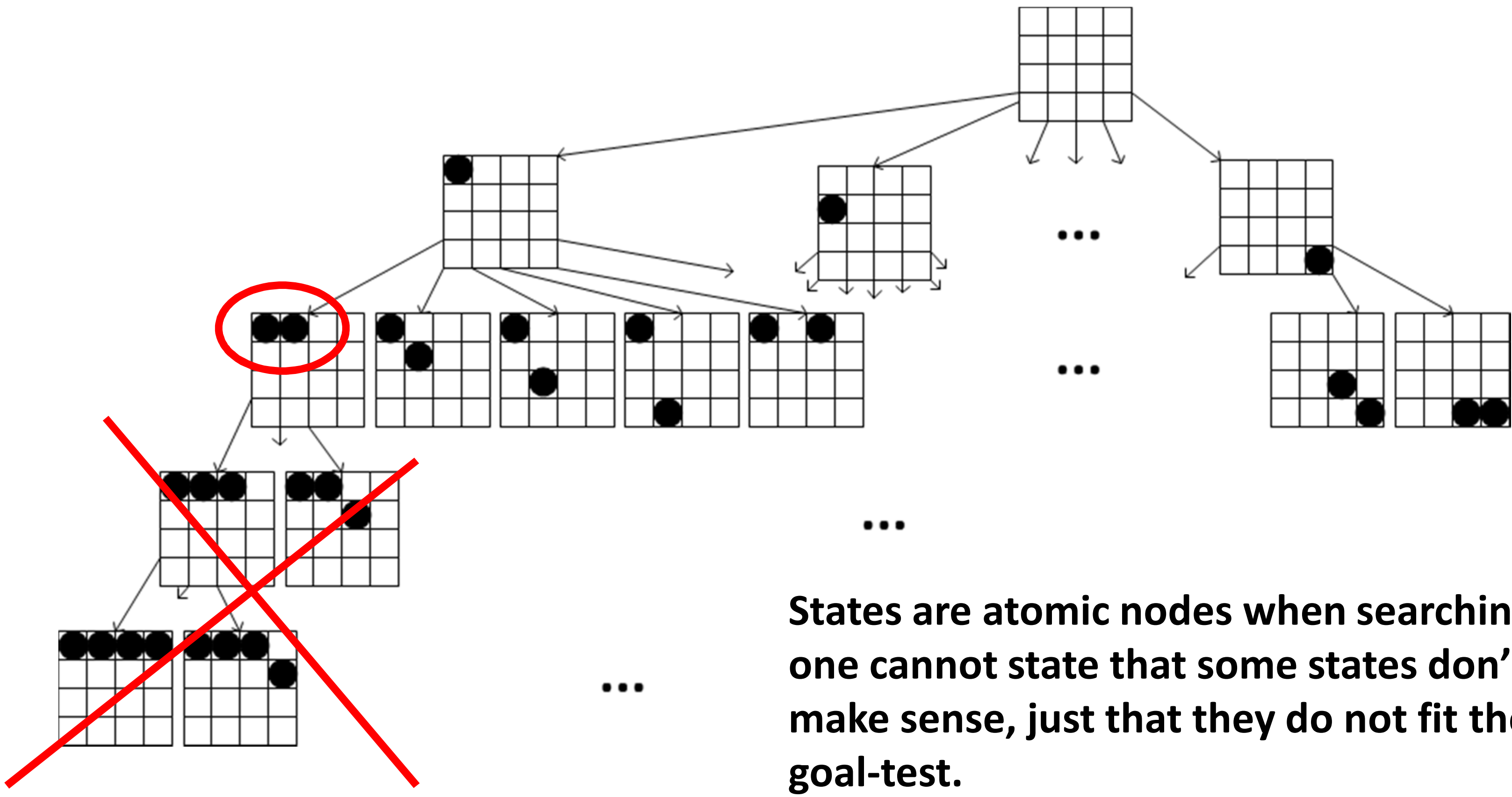
Constraint Satisfaction

- What are Constraint Satisfaction Problems (CSPs)? Examples
- Backtracking Search for CSPs
- Problem Structure and Problem Decomposition
- Local Search for CSPs

Literature:
Ghallab, Nau, Traverso, Sections 8.2 and 8.4
Russell & Norvig, Chapter 6;

Standard State-space Search



States are atomic nodes when searching, one cannot state that some states don't make sense, just that they do not fit the goal-test.

Constraint Satisfaction Problem

- Standard Search → State is a data structure that supports functions such as goal-test, eval, successor
- **But for a particular problem class, we know more!**

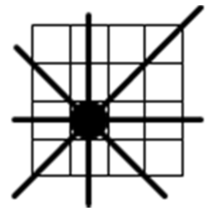
- A Constraint Satisfaction Problem has 3 elements: X, D, C:
  - State X is defined based on **variables** X_1, X_2, ... X_n
  - D is a set of **domains**, a domain D_i for each variable
  - C is a set fo **constraints that specify allowable combination of values** for subsets of variables; a Constraint is a pair <*scope,rel*> with *scope* a tuple of variables and *rel* a relation that defines values that the variables from scope can take.

Constraint Satisfaction Problem (CSP)

- Solution is a **consistent, complete assignment** of values to variables:
  - Consistent assignment in which all constraints are satisfied;
  - Complete assignment: all variables have an assigned value
- State is not a black box, but **factored structure**
  → Simple example of a formal representation language
- Allows useful general-purpose algorithms with more power than standard search algorithms (although they are based on search algorithms)

N-Queens as CSP

- Pre-assume that each queen occupies one column:
- **Variables**:
  $queen_1$ ... $queen_N$
- **Domains**:
  $queen_i \in \{row_1, \ldots row_N\}$
- **Constraints**
  no queen in the same row: $queen_1 \neq queen_2 \neq \ldots \neq queen_N$
  no queen on diagonals:
  - $\forall j$: $queen_i \neq (queen_{i+j}) + j$
  - $\forall j$: $queen_i \neq (queen_{i+j}) - j$
  - $\forall j$: $queen_i \neq (queen_{i-j}) - j$
  - $\forall j$: $queen_i \neq (queen_{i-j}) + j$

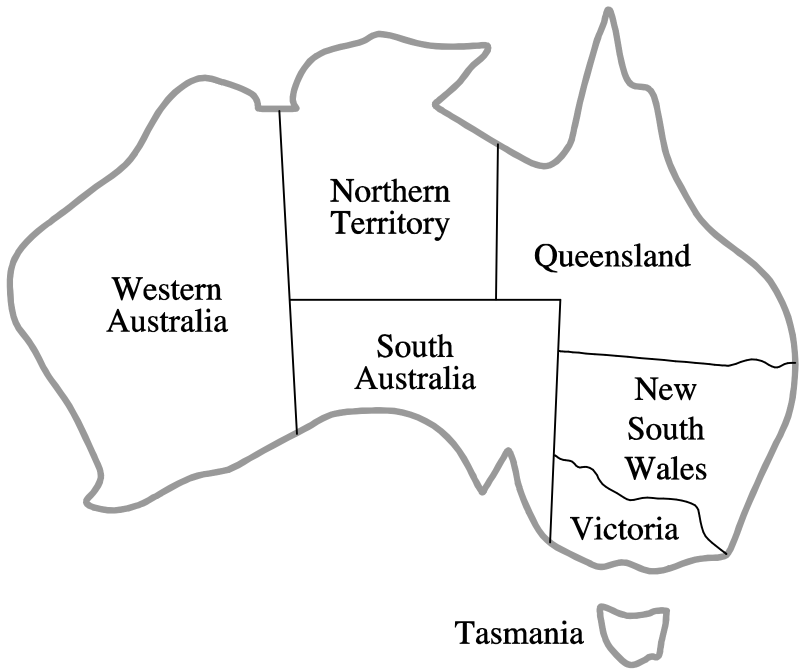


Constraint Satisfaction Problem

- Natural representation for wide variety of problems
- CSP solving can be faster than state-space search because CSP can quickly eliminate large area of the search space (e.g. If chosen {SA=*green*}, none of the five neighbouring variables can take *green* → 2^5 instead of 3^5 assignments to look at.
- In state space only test for goal possible – in CSP information why a state is not a goal → which variable violates a constraint → focus attention on relevant variables

→ Many problems which are untractable for state-space search can be solved by CSP

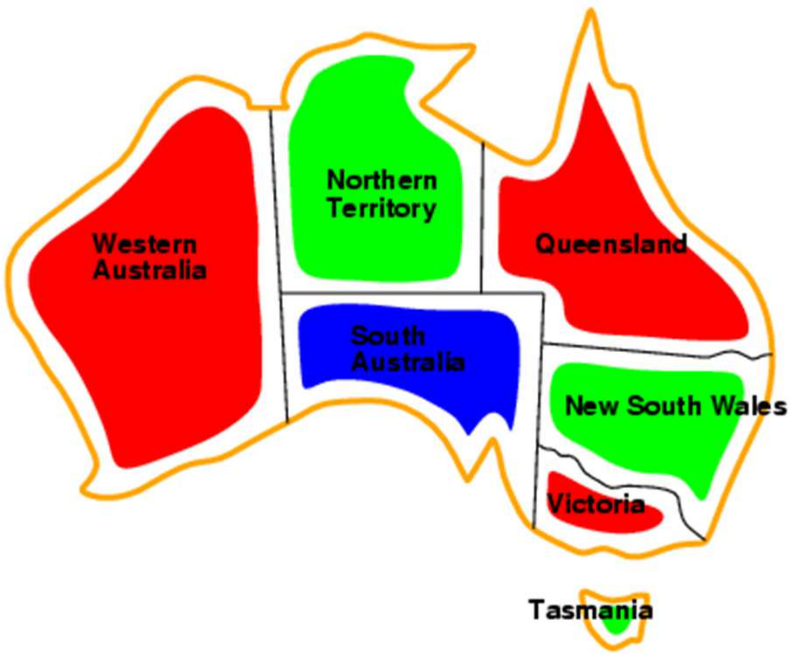
Example: Map Coloring



- Variables:
  WA, NT, SA, Q, NSW, V, T
- Domain: {red, blue, green}
- Adjacent regions must have different colors,
  e.g. <(WA,NT), WA≠NT>
  <(WA,NT),[(red, green), (red, blue), (green,red)..]>

Russell&Norvig, 2010, chapter 6

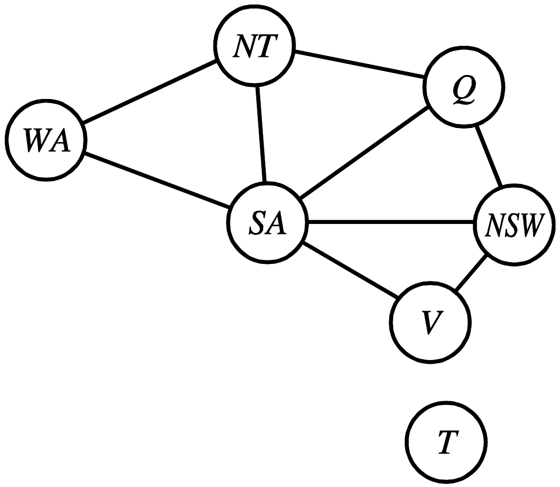
Example: Map-Coloring



- Solutions are complete and consistent assignments,
- Example solution:
  WA = red, NT = green,
  Q = red, NSW = green,
  V = red, SA = blue,
  T = green

Representation as Constraint Graph

- Constraint graph: nodes are variables, arcs show constraints/connect variables that participate in a constraint



- General-purpose CSP algorithms may use the graph structure to speed up search → identification of independent sub-problems e.g., Tasmania

Example: Cryptoarithmetic Puzzle

$$
\begin{array}{cccc}
 & T & W & O \\
+ & {}_{X_3}T & {}_{X_2}W & {}_{X_1}O \\
\hline
F & O & U & R
\end{array}
$$

- **Variables**: *F, T, U, W, R, O* and auxiliary variables X_1 $X_2 X_3$
- Domains: {*0,1,2,3,4,5,6,7,8,9*} and {*0,1*} for X_1 $X_2 X_3$
- Constraints: $F \neq T \neq U \neq W \neq R \neq O$ (all different)
- $O + O = R + 10 \cdot X_1$
- $X_1 + W + W = U + 10 \cdot X_2$
- $X_2 + T + T = O + 10 \cdot X_3$
- $X_3 = F$, $T \neq 0$, $F \neq 0$

Variations of CSPs

- **Discrete variables**
  - finite domains:
    - *n* variables, domain size *d* → in the order of d^n complete assignments
    - Combinatorial problems
  - infinite domains:
    - integers, strings, etc.
    - no enumeration of possible combinations feasible:
      need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
    - Linear constraints solvable, no general algorithm for non-linear constraints on integer variables
    - e.g., job scheduling, variables are start/end days for each job

- **Continuous variables → Operations Research**

Real-World CSPs

- Assignment problems e.g., who teaches which class, who sits in which office, who shares a ride with whom…
- Timetabling problems e.g., which class is offered when and where?
- All forms of scheduling (transportation, planning,...)
- Hardware configuration
- ...

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

Solving CSPs

- Standard Search → all solutions at depth n (n variables) and there are many leafs.
- Local Heuristic Search
  - We need explicitly allow states with unsatified/violated constraint
  - Local operators for reassign variable values;
  - The n-queens example was actually a CSP
  - Applicable Algorithms
    - Hillclimbing: find a new ”better” next overall state – select randomly a variable, try to find an assignment to that variable that improves overall score
    - Simulated Annealing
    - Genetic Algorithm

→ Specific Search for Constraint Satisfaction Problems: Backtracking Search

Backtracking Search

- Basic Assumption:
  - Variable assignments are **commutative**
    → **Sequence of assigning is not important:** [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 and backtracking when a variable has no legal values to assign is called **backtracking search**

- Backtracking search is the basic uninformed algorithm for CSPs

Backtracking Search Pseudo Code

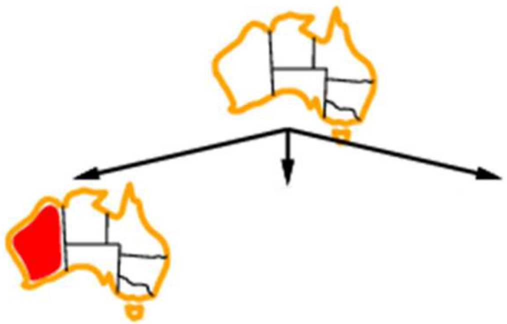
function BACKTRACKING-SEARCH(*csp*) **returns** a solution, or failure
  return RECURSIVE-BACKTRACKING({}, *csp*)

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*

From Russell&Norvig, 2010, p. 215

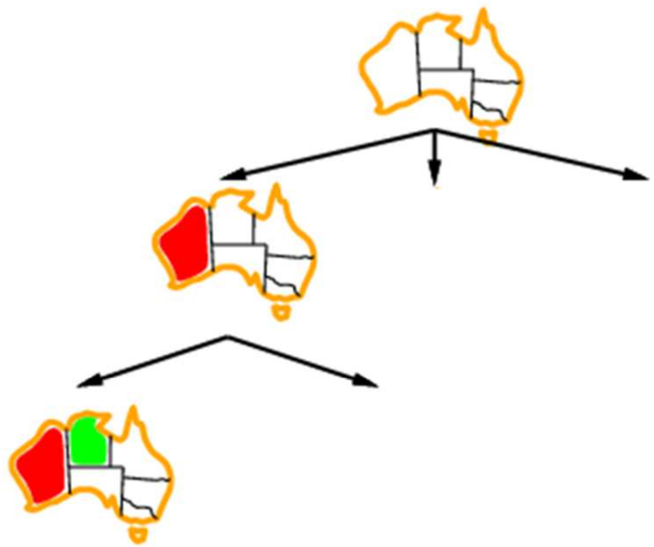
Backtracking Example

Backtracking Example



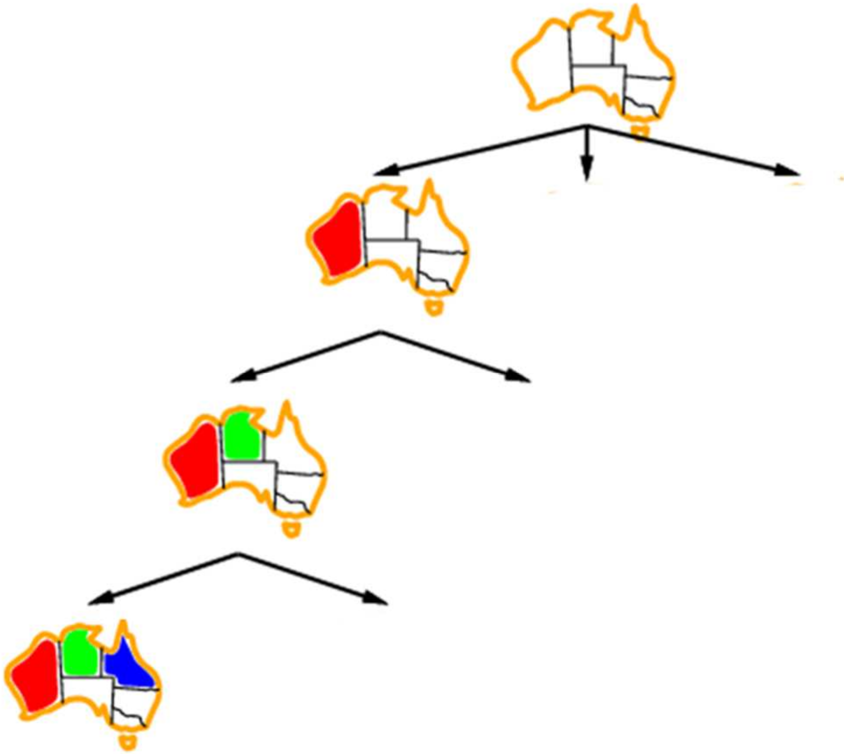
Add {WA=red}

Backtracking Example



Add {NT=green}

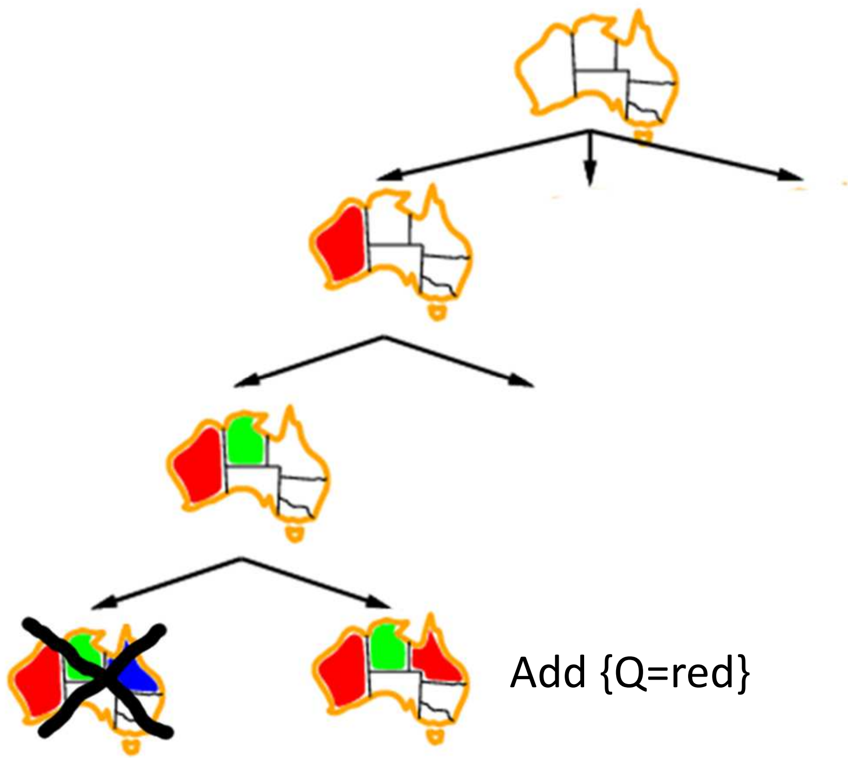
Backtracking Example



Add {Q=blue}

Next variable = SA, but
there is no assignment that
does not violate a constraint

Backtracking Example



Add {Q=red}

Remove {Q=blue}

Further steps:
add {SA=blue}
add {NSW = green}
add {V=red} ....

Improving Backtracking Efficiency

- Randomly selecting a variable and try the values one after the after is quite naïve → improvements possible just based on the general structure of a CSP, already without further problem-specific information

- Strategies can give huge gains in speed:
  - Which variable should be assigned next?
    → SELECT-UNASSIGNED-VARIABLE
  - In what order should its values be tried?
    → ORDER-DOMAIN-VALUES

- Can we detect inevitable failure early?

Selecting next variable for assignment: ”Most Constrained Variable Heuristic”

- Most **constrained** variable:

  choose the variable with the **fewest legal remaining values** in its domain

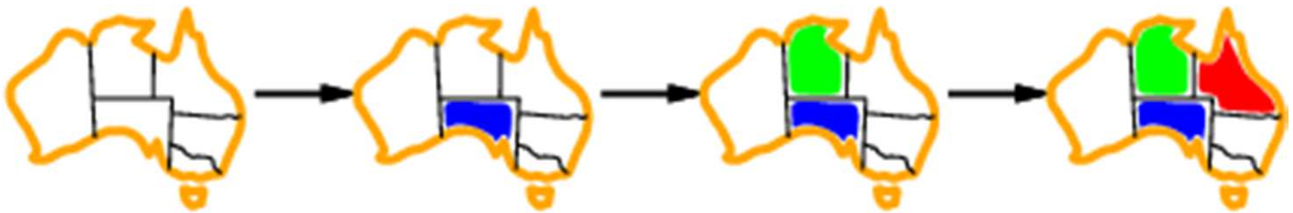
SA has left {blue}
Q has left {blue, red}
All others: {greeen, blue, red}

NT has left {green, blue}
SA has left {green, blue}
All others have left: {greeen, blue, red}

Simple Implementation: For each variable determine which values would work and count the number of those values. Select the variable with the lowest number for assignment.

Tie Break?

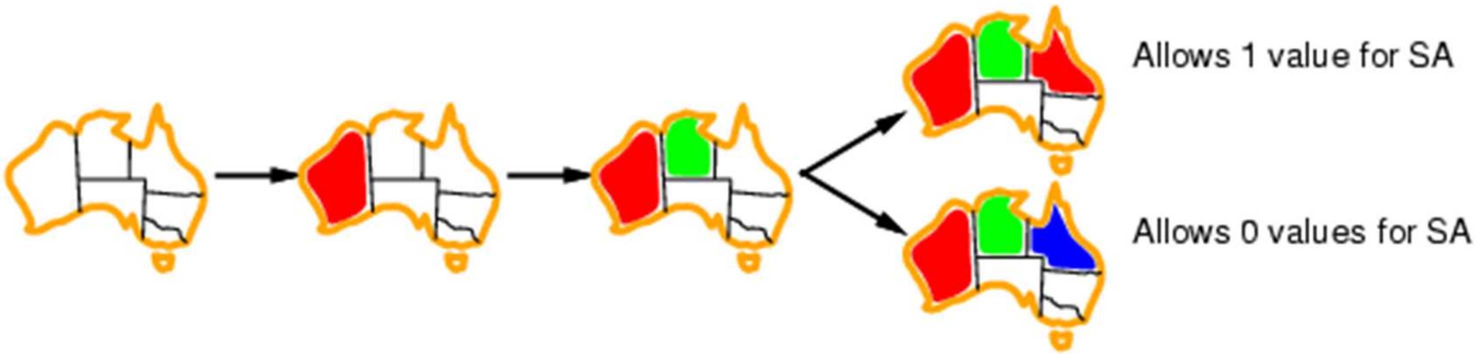
- What if there are 2 or more variables with the name number?
(e.g. In the beginning all variables have same number of values)
- Most constraining variable / Degree heuristic as **tie breaker:**
choose the variable which is involved in the most constraints on other variables



WA has 2 neighbours
NT has 3 neighbours
SA has 5 neighbours ← **start with SA**
Q: 3 neighbours
NSW: 3 neighbours
V: 2 neighbours

Select values:
Least Constraining Value Heuristic

- Choose the value that rules out the fewest values in the remaining variables → the least constraining value



- How shall we determine that value?
  For each value for the currently selected variable do a testwise assignment and check how many values would be left for all then un-assigned variables. The sum of the number of values tells how many options would be left → we select the value for real assignment that has the maximum values left.

Detect inevitable failure early?

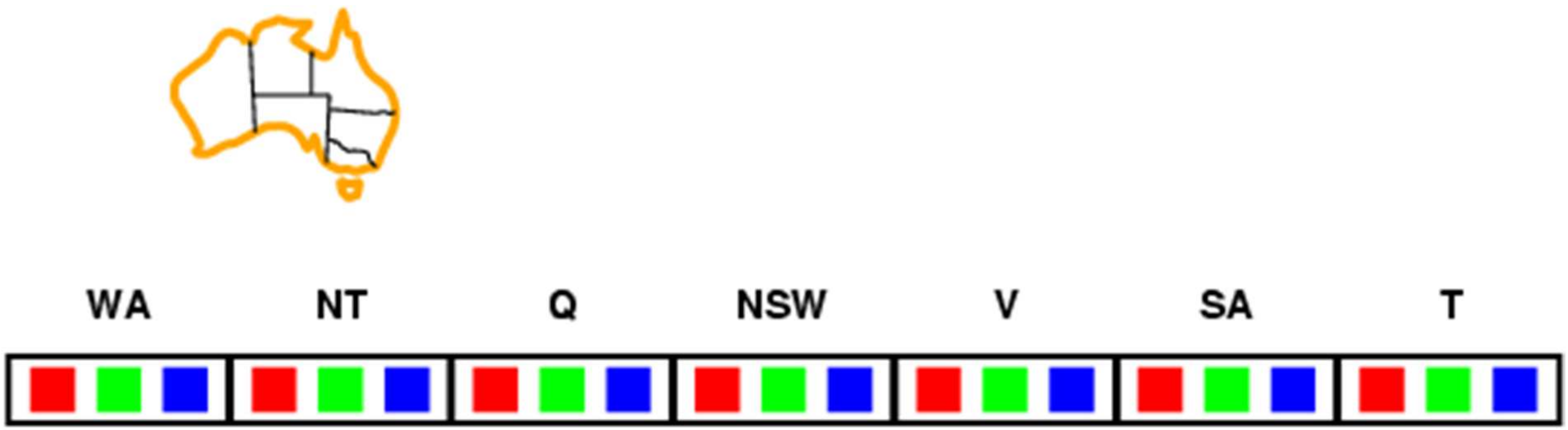
- Prune paths during search that are bound for failure
- Idea: **Whenever there is a value for a variable assigned (or something else changes), check the values of the neighbours for consistency and delete values from the domains which are not consistent any more.**

- **Forward Checking**: keep track of remaining legal values for unassigned variables; Terminate search/backtrack when any variable has no legal values (good to combine with the minimum remaining values heuristic)

- **Arc Consistency:**
A variable X_i is **arc-consistent** with respect to variable X_j, if for every value in the current domain D_i, there is some value in the domain D_j that satisfies the (binary) constraint on the arc (X_i,X_j).
A network is arc-consistent if every variable is arc-consistent with every other variable

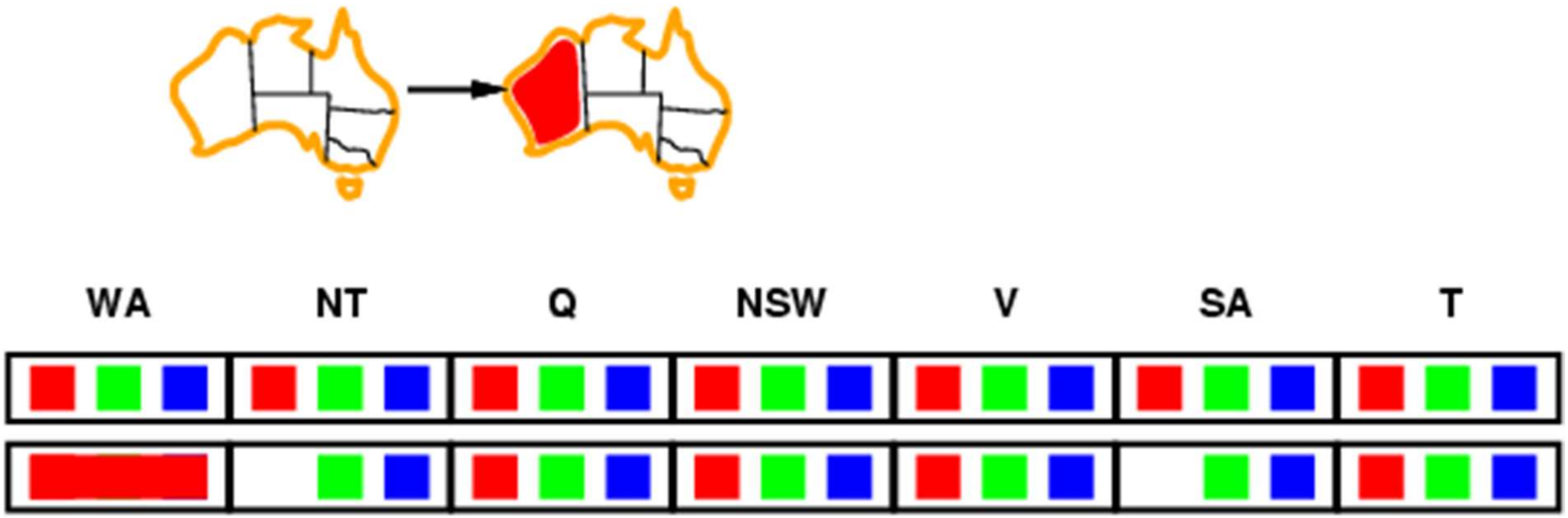
Forward Checking

- keep track of remaining legal values for unassigned variables;
  erminate search in this branch when any variable has no legal values



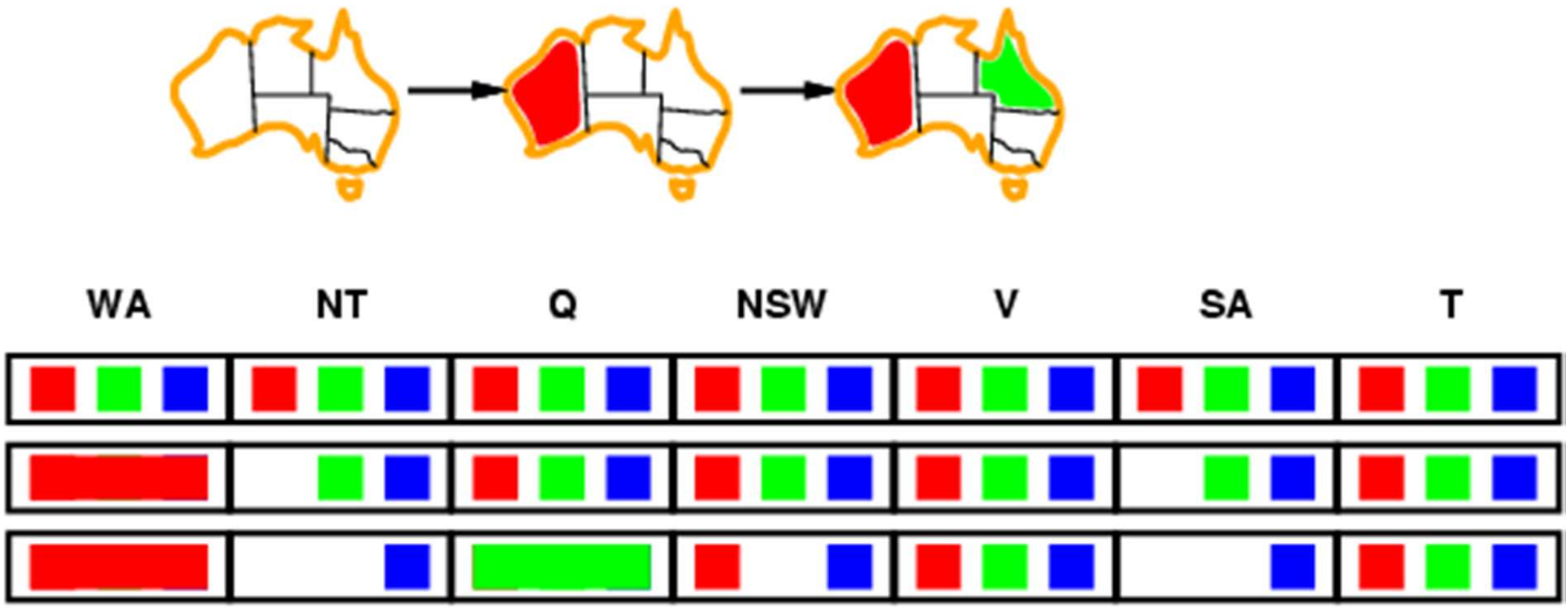
Constraint Propagation: Forward Checking

- Idea: keep track of remaining legal values for unassigned variables; Terminate search when any variable has no legal values



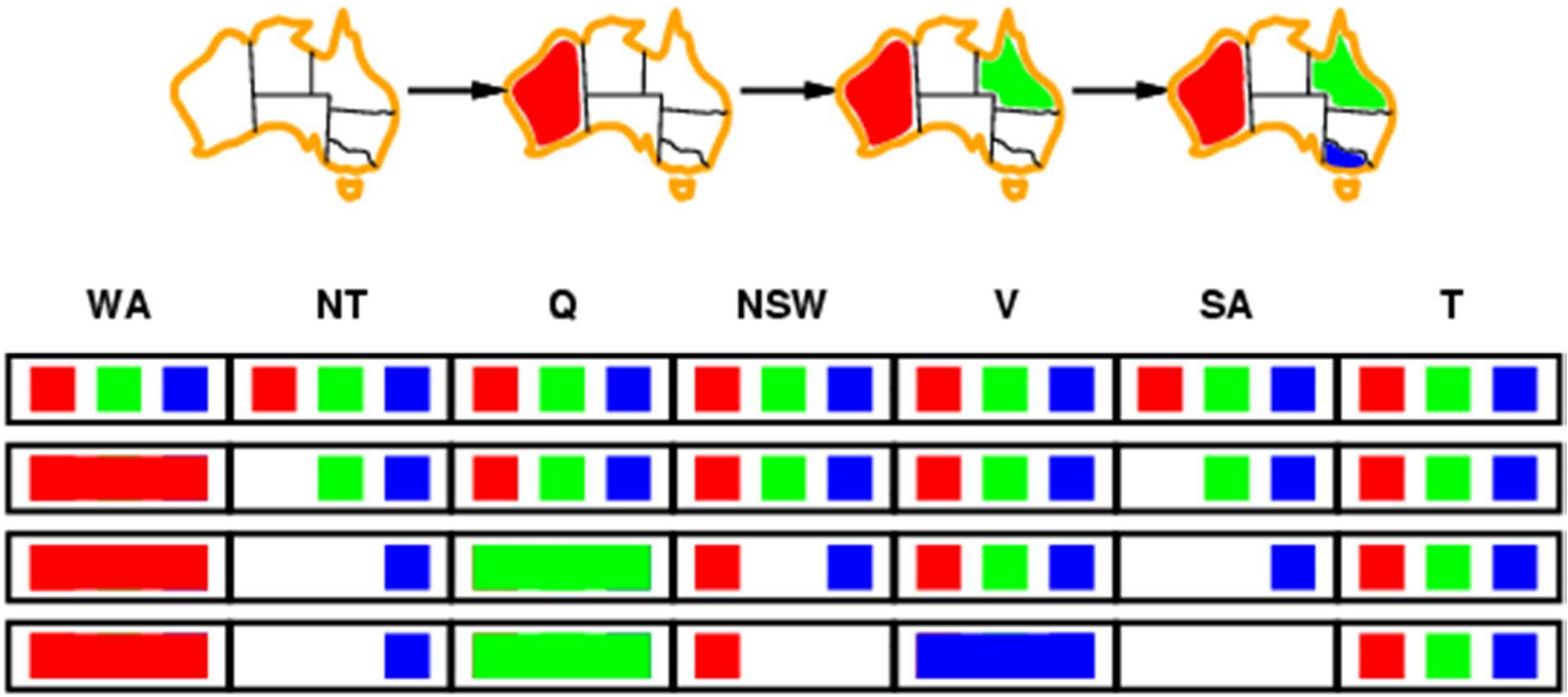
Constraint Propagation: Forward Checking

- Idea: keep track of remaining legal values for unassigned variables; Terminate search when any variable has no legal values



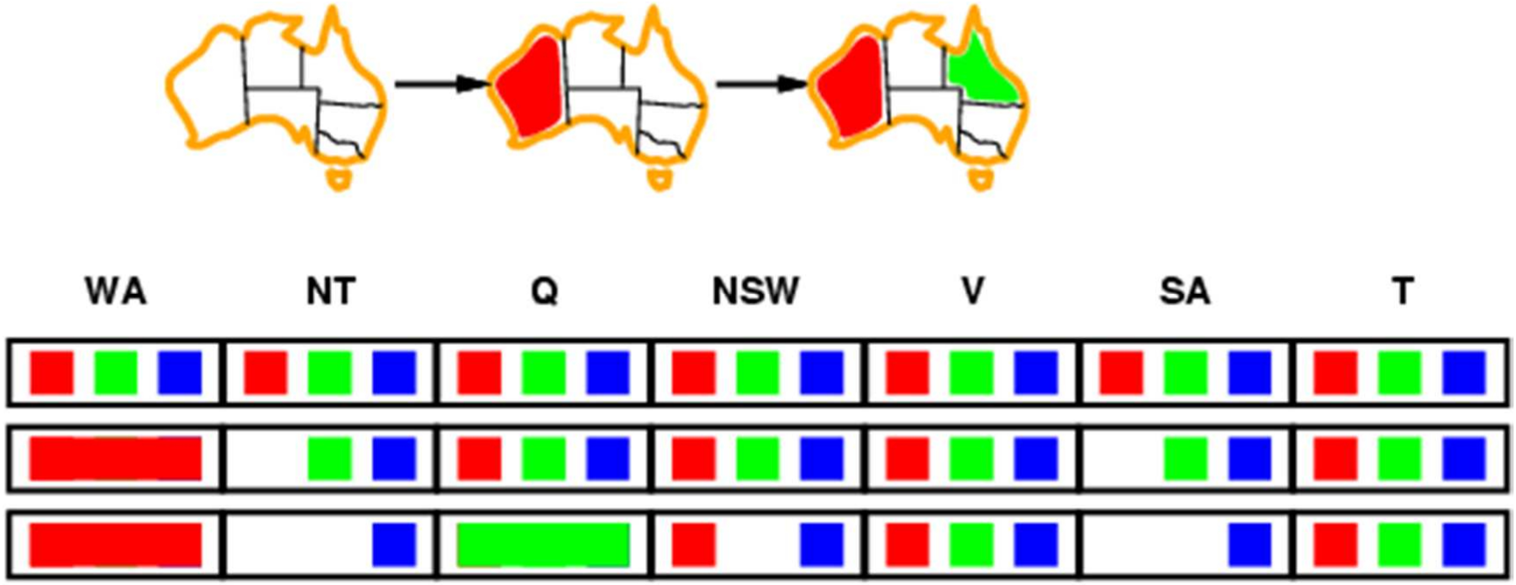
Constraint Propagation: Forward Checking

- Idea: keep track of remaining legal values for unassigned variables; Terminate search when any variable has no legal values



Can we do more?

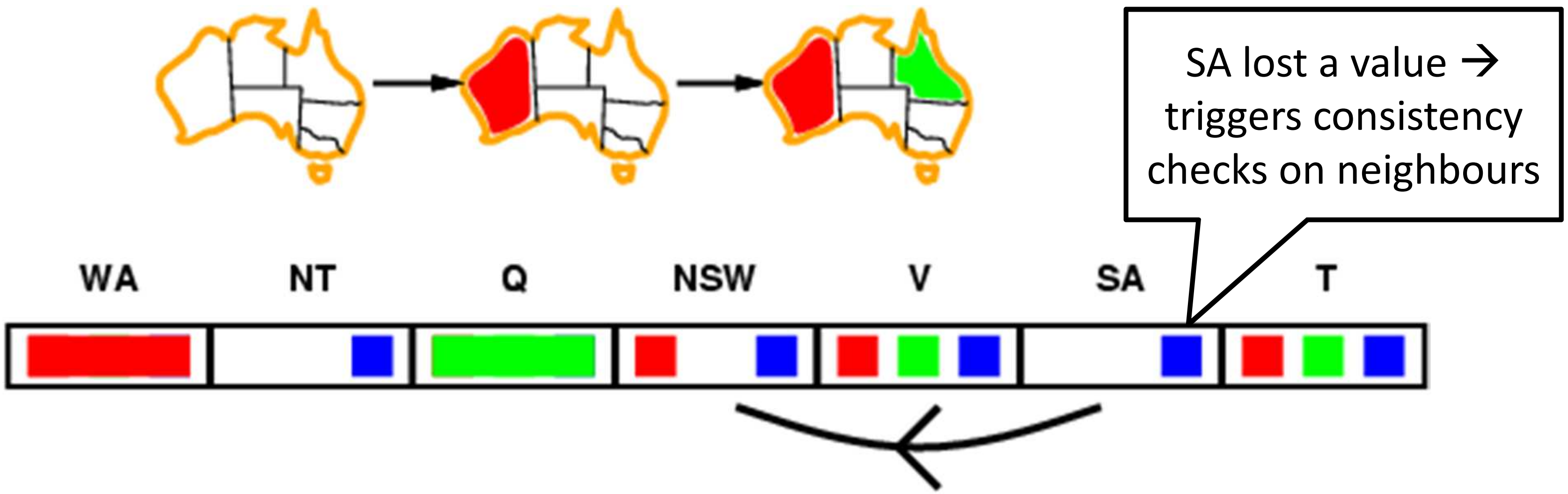
- Forward checking propagates information from assigned to unassigned variables, but does not provide early detection for all failures



- NT nd SA cannot **BOTH** be blue → {Q=green} should be backtracked
  Can we detect that earlier than with forward checking?

Constraint Propagation

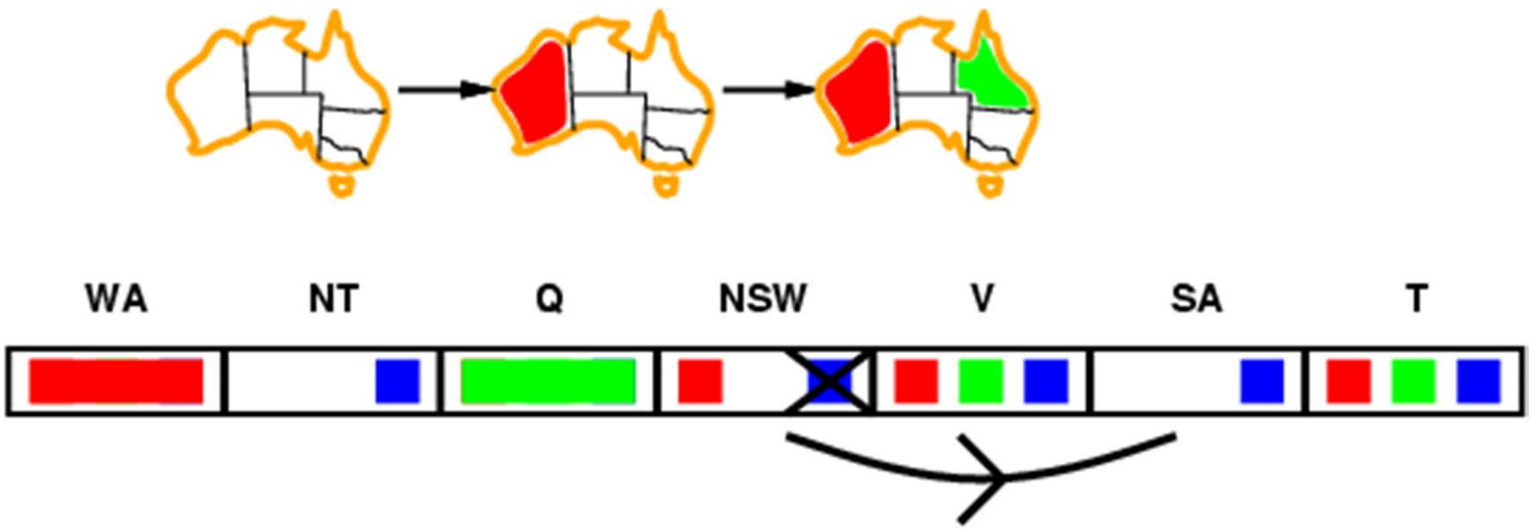
- Propagation of consistency by further checking consistency along all connections in the constraint graph



Constraint Propagation

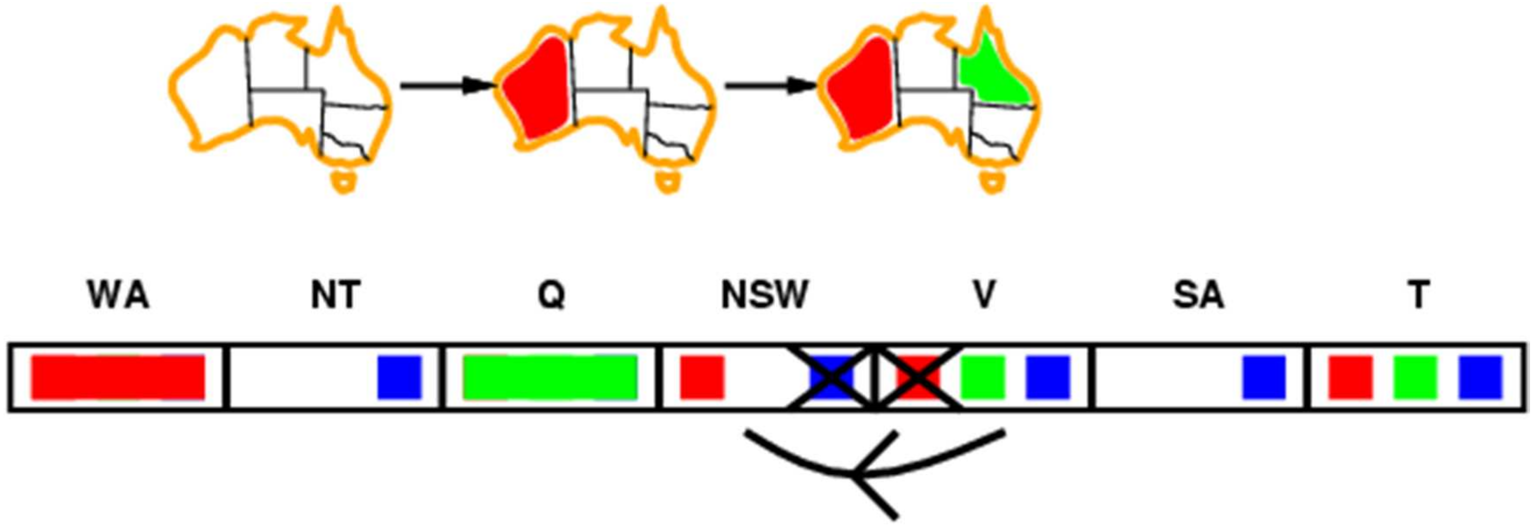
Propagation consistency trough the network.

- After assigning {Q=green}, all neighbours of Q need to be updated → SA, NT and NSW loose the green value from their domains
- For (SA, NSW) the blue value at NSW is not consistent with the only remaining value of SA → delete blue from the domain of NSW



Constraint Propagation

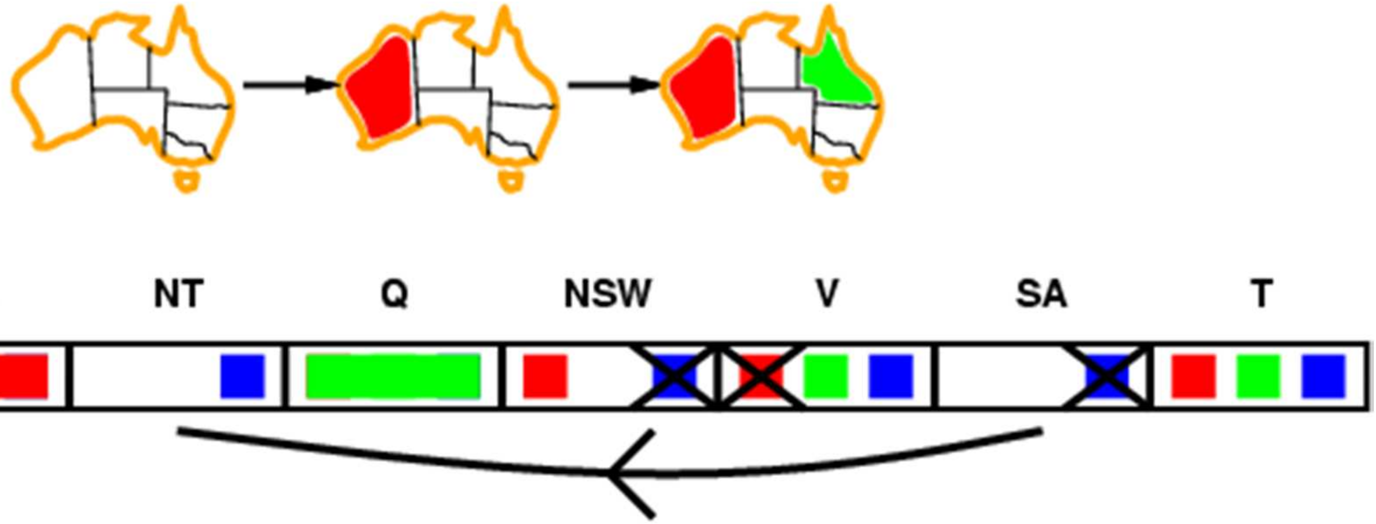
- After checking the arc consistency of variable SA and NSW, the domain of NSW lost the blue value. So its neighbours have to be checked for consistency with the remaining values of NSW with other domains



NSW has only red left → V looses the red value as they are neighbours

Constraint Propagation

- Each variable needs to be made arc consistent with each other for making the full network arc consistent.
- Checking the domains of SA against the domains of NT: both cannot be blue at the same time → if NT is set to blue, SA looses blue → current path does not lead to solution!



- **Full constraint propagation detects failures earlier than simple forward checking**
- Can be run as a preprocessor or after each assignment

Pseudo Code

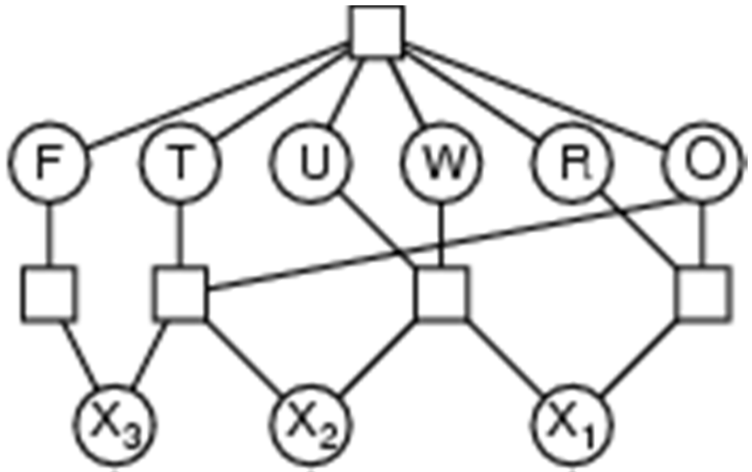
```
Function arc-consistency (csp): returns a csp
inputs: csp with binary constraints and variable X1,X2,...Xn
local: queue of arcs
while queue is not empty do:
    (Xi,Xj) ← REMOVE-FIRST(queue)
    if remove-inconsistent-values (Xi, Xj) then
        for each Xk in neighbours(Xi) do
            add (Xk, Xi) to queue

Function remove-inconsistent-values (Xi, Xj) returns true if
sucessful
removed ← false
for each x in Domain[Xi] do:
    if no value y in Domain[Xi] allows (x,y) for (Xj↔Xi) then
        delete x from Domain[Xi]; removed ← true
return removed;
```


Peter Norvig's AI Lecture based on the Russell&Norvig book

Example: Cryptoarithmetic Puzzle

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



- **Variables**: *F, T, U, W, R, O* and auxiliary variables $X_1\ X_2\ X_3$
- Domains: {*0,1,2,3,4,5,6,7,8,9*}
- Constraints: *Alldiff (F,T,U,W,R,O)*
- $O + O = R + 10 \cdot X_1$
- $X_1 + W + W = U + 10 \cdot X_2$
- $X_2 + T + T = O + 10 \cdot X_3$
- $X_3 = F$, $T \neq 0$, $F \neq 0$