05

Chapiter

Constraint Satisfaction Problems

2I1AE1: Artificial Intelligence

Régis Clouard, ENSICAEN - GREYC

"Saying Deep Blue doesn't really think about chess
is like saying an airplane doesn't fly
because it doesn't flap its wings."
Drew McDermot

In this chapter

- **Constraint Satisfaction Search**
  - In which we see how treating states as more than just little black boxes.

1) Constraint Satisfaction Problems
2) Solving CSP with Search Algorithms
3) Solving CSP with specific Algorithms
   1) Forward Checking
   2) Arc Consistency
4) Heuristics for CSP

Configuration Search Problem

- Case of search problems of type "configuration finding"
  - **State space search problems**
    - Each state is a **black box** with no discernible internal structure.
      - States are handled by **problem-specific routines**:
        - `get-successors()` and `is-goal()` functions.
    - Search algorithms can use **problem-specific heuristics** to speed-up the search
  - **Constraint satisfaction problems (CSP)**
    - States conform to a **standard and very simple representation (white box)**.
    - `get-successors()` and `is-goal()` are **general functions**.
    - Search algorithms use **general-purpose heuristics**.

Constraint Satisfaction Problem

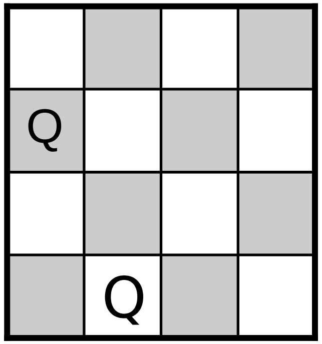
- Constraint Satisfaction Problem is a subpart of configuration search problem where:
  - A **state** can be defined by a **set of variables**: X_1, X_2,.., X_n.
    - Each variable X_i has a nonempty **domain** D_i of valid values.
    - A set of **constraints** exists on possible variable values: C_1, C_2,.., C_m.
  - A complete **assignment** is one in which every variable is mentioned, and a **solution** to a CSP is a complete assignment that satisfies all the constraints.
- Special properties of the CSP lead to special solving algorithms.

Variety of Constraints

- Unary constraints involve a single variable.
  - e.g., A ≠ green, A > 3
- Binary constraints involve pairs of variables.
  - e.g., A ≠ C, A > C
- Higher-order constraints involve 3 or more variables.
  - e.g., function allDiff()

Example 1. N-Queens

- **Goal**
  - *N* queens placed in non-attacking positions on the board.
- **Variables**:
  - Represent queens row, one for each column:
    - Q_1, Q_2, Q_3, Q_4
- **Domain**:
  - Row placement of each queen on the board:
    - $Q_i \in \{1, 2, 3, 4\}$
- **Constraints**:
  - $Q_i \neq Q_j$: two queens not in the same row.
  - $| Q_i - Q_j | \neq | i - j |$: two queens not on the same diagonal.



$Q_1 = 2, Q_2 = 4$

Example 2. Cryptarithmetic Puzzles

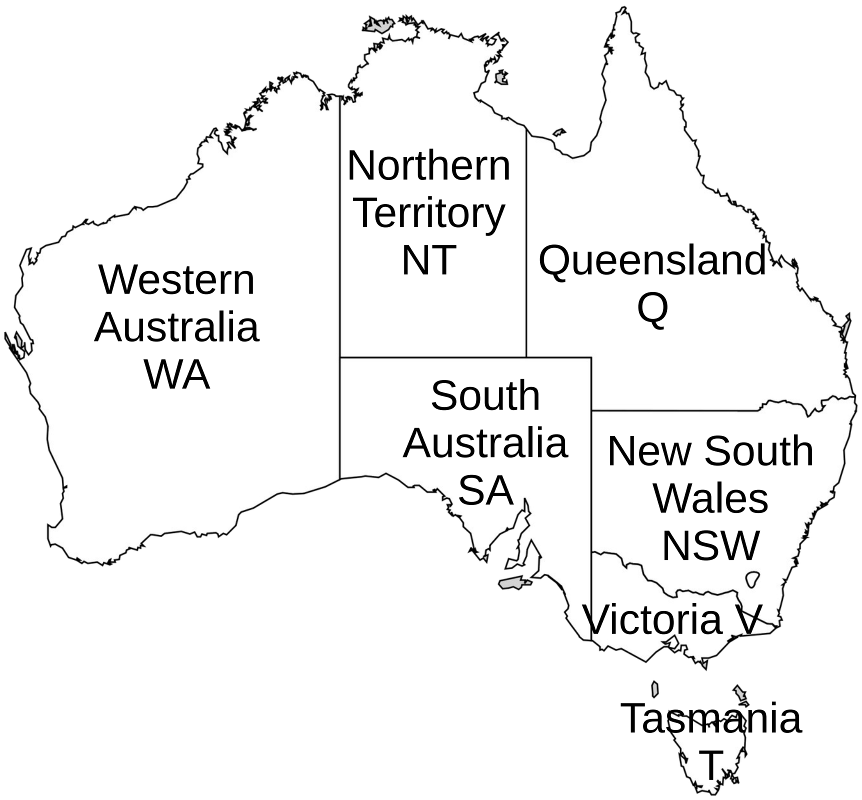
- Decipher the letters using the constraints that no two letters can have the same numerical value and the letters conform to the operation:

$$
\begin{array}{cccccc}
 & X_4 & X_3 & X_2 & X_1 & \\
 & & S & E & N & D \\
 & + & M & O & R & E \\
\hline
= & M & O & N & E & Y
\end{array}
$$

- **Variables?**
  - S, E, N, D, M, O, R, Y
  - X_1, X_2, X_3, X_4
- **Domain?**
  - S, E, N, D, M, O, R, Y $\in [0; 9]$; $X_i \in \{0,1\}$
- **Constraints?**
  - allDiff(S, E, N, D, M, O, R, Y)
  - $D + E = Y + 10. X_1$
  - $X_1 + N + R = E + 10. X_2$
  - etc

Example 3. Map Coloring

- Color the Australian map using 3 different colors such that no adjacent countries have the same color.
- **Variables?**
  - Represent countries.
    - WA, NT, SA, Q, NSW, V, T
- **Domain?**
  - Country ϵ {Red, Blue, Green}
- **Constraints?**
  - WA ≠ NT,
  - WA ≠ SA
  - NT ≠ SA, …



Solving CSP with State Space Search Algorithms

- Formulation of a CSP as a Search Problem:
  - **States**
    - Domain D_i of each variables X_i.
    - Assignment of variables X_i with values from the domain D_i.
  - **Initial state**
    - Domains: all values for all variables.
    - No variable is assigned a value: {}.
  - **Actions**
    - Assign a value to one of the unassigned variables such that it doesn't violate the constraints.
    - Remove assigned variable from domains
  - **Goal test**
    - All variables are assigned and no constraints are violated.
  - **Path cost**
    - A constant cost for every step (e.g., 1).
  - **Type**
    - Configuration finding

What search algorithm to use?

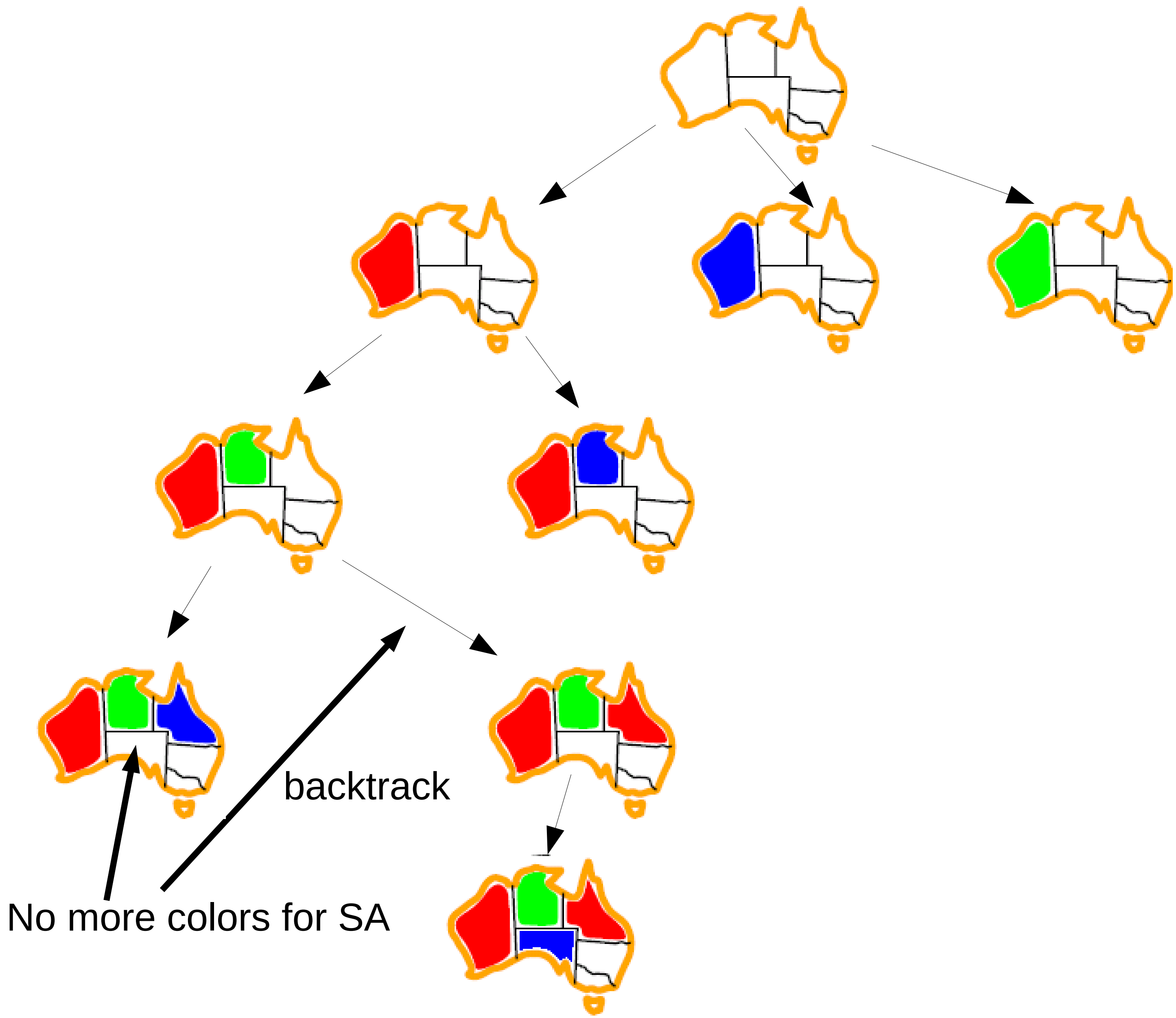
- **Informed search**
  - Problem: no available heuristics.
- **Breadth-first search algorithm**
  - Problem: time and space complexity: $O(b^d)$.
  - where $b = \max|D_i|$: maximum number of values for the variables
    $d = |X|$: number of variables in the CSP.
- **Depth-first search algorithm**
  - Since we know the depth d of the tree ($m=d$), it is complete.
  - Since we search for a satisfying configuration not an optimal path, depth-first is relevant.
  - Time complexity: $O(b^d)$ but space complexity: $O(b.d)$.
  - So, we can use DFS and no need to add a closed-list (space complexity is still $O(b.d)$).

Depth-First Search Algorithm

- No need to use a closed-list just use a subtle generation of successors → **backtracking search:**
  - Choose values for one variable at a time that keeps the solution consistency.
  - Backtrack when a variable has no legal value left to assign.

```
function BACKTRACKING-SEARCH(problem) returns a solution, or failure
  return BACKTRACK({ }, problem) # set initial assignment
function BACKTRACK(assignment, problem) returns a solution, or failure
  IF assignment is complete THEN return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(problem)
  FOREACH value in ORDER-DOMAIN-VALUES(var, assignment, problem) DO
    add {var = value} to assignment
    IF value is consistent with assignment THEN
      result ← BACKTRACK(assignment, problem)
      IF result ≠ failure THEN
        return assignment
    remove {var = value} and inferences from assignment
  return failure
```

Backtracking Example: Map Coloring



[sudoku.sh/takuzu.sh]

Backtracking Search

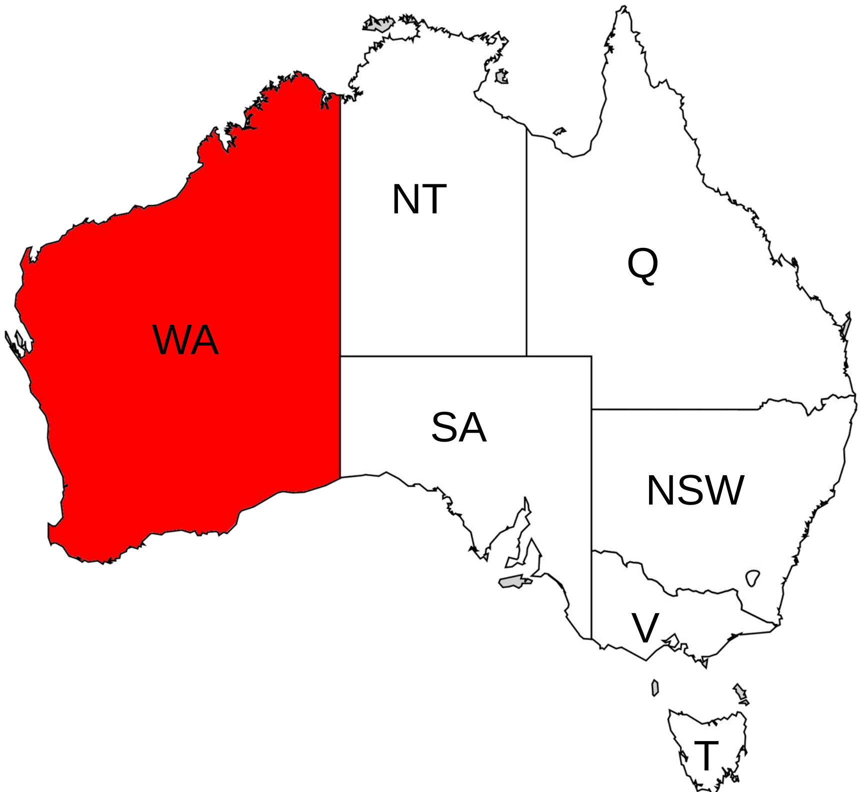
- Not efficient
  - Based on combination of all possible values for each variable.
  - Time complexity $O(b^d)$
- Can we do better?
  - Yes. Take benefit from constraint propagation.
    - The choice of one value for one variable reduces the domain for other variables.

Constraint Propagation

- A state is defined by
  - A set of **variables**.
  - The possible **values** of each variables: domains.
  - List of **legal and illegal assignments** for unassigned variables.
- Legal and illegal assignments are represented via:
  - equations
  - inequations
  - disequations
- Example: map coloring
  - Equation: WA = Red
  - Disequation: NT ≠ Red

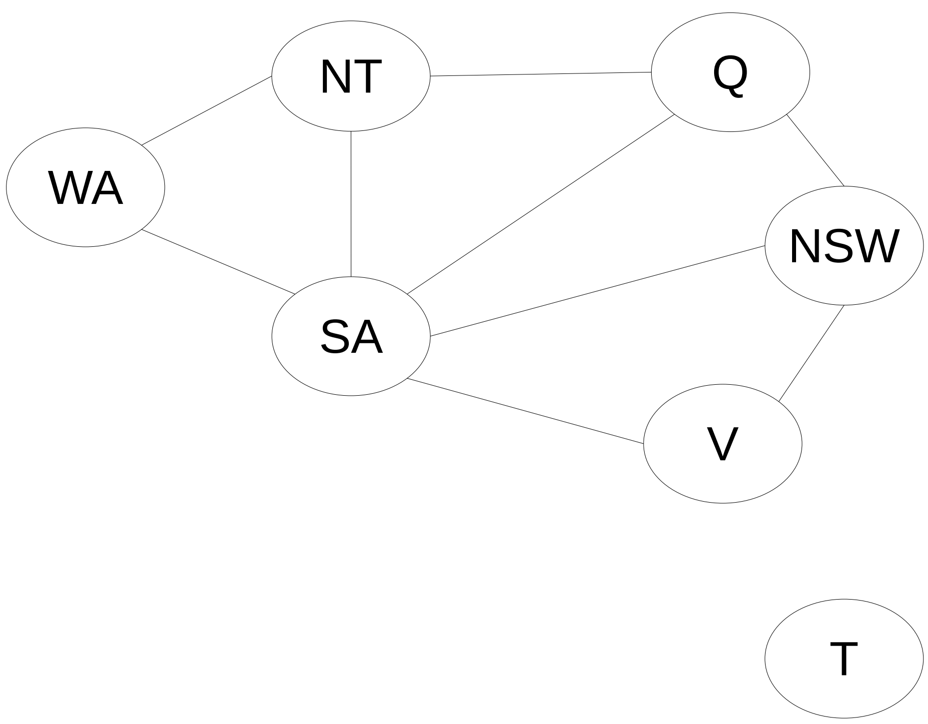
Constraint Propagation

- Constraints + assignments can entail new equations and disequations.
  - e.g. WA = Red → NT ≠ Red, SA ≠ Red
- **Constraint propagation**
  - The process of inferring new equations and disequations from existing equations, disequations and inequations.
  - This can drastically reduce the search space (branching factor).



Constraint Graph

- Constraint graph is an efficient representation of constraints → identify coupling between variables
  - **nodes** are variables.
  - **arcs** are constraints.
- Example: Map coloring of Australia territories with three colors Red, Green, Blue.



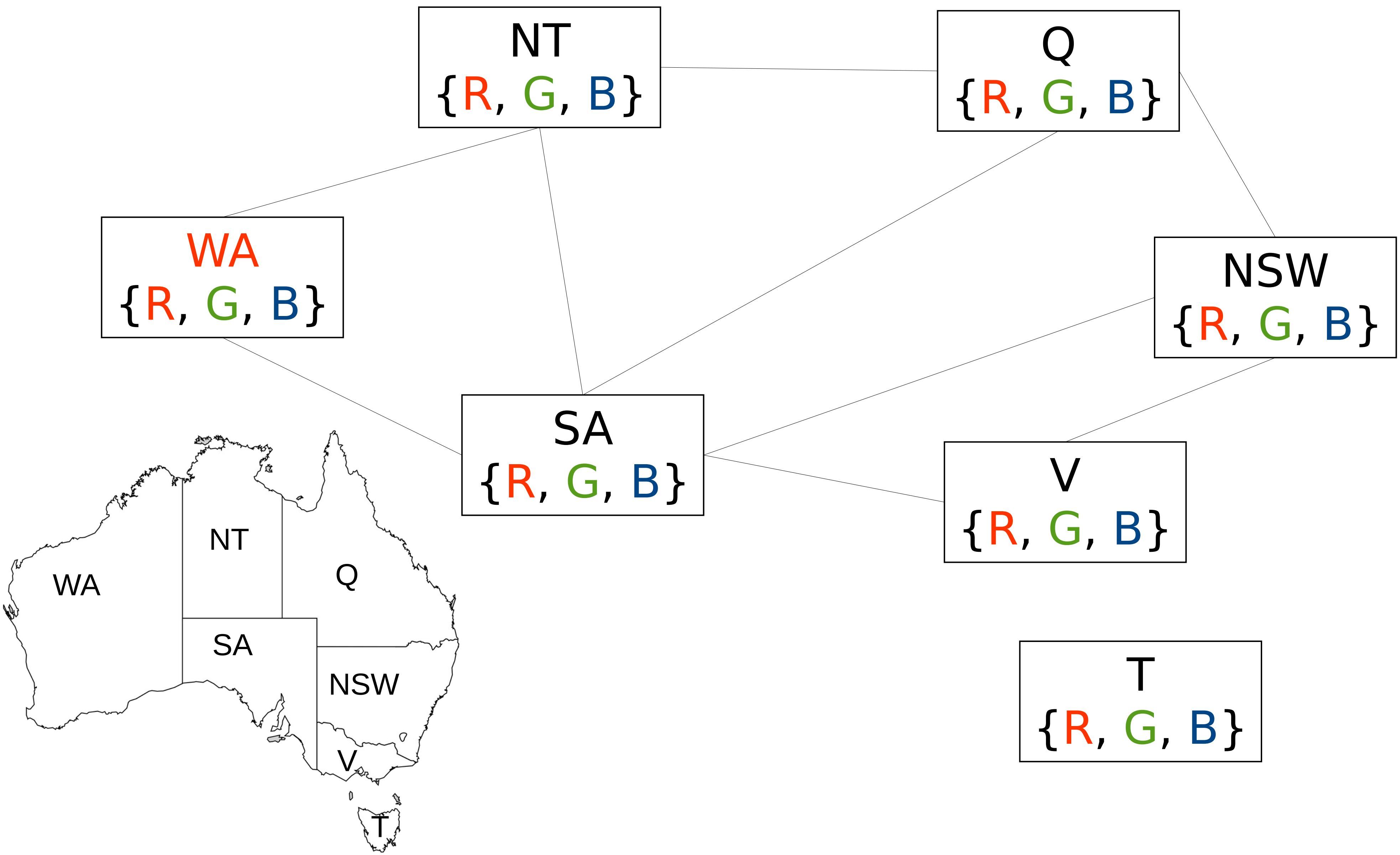
Constraint Propagation Techniques

- Backtracking Search is a **look-back algorithm**.
  - Check consistency only for complete assignments : too late.
- Constraint propagation leads to **look-ahead algorithm**.
  - Check consistency after each assignment through constraint propagation and prune the search tree.
  - Two look-ahead algorithms:
    - **Forward checking**
    - **Arc consistency**

1. Forward Checking Algorithm (Haralick 1980)

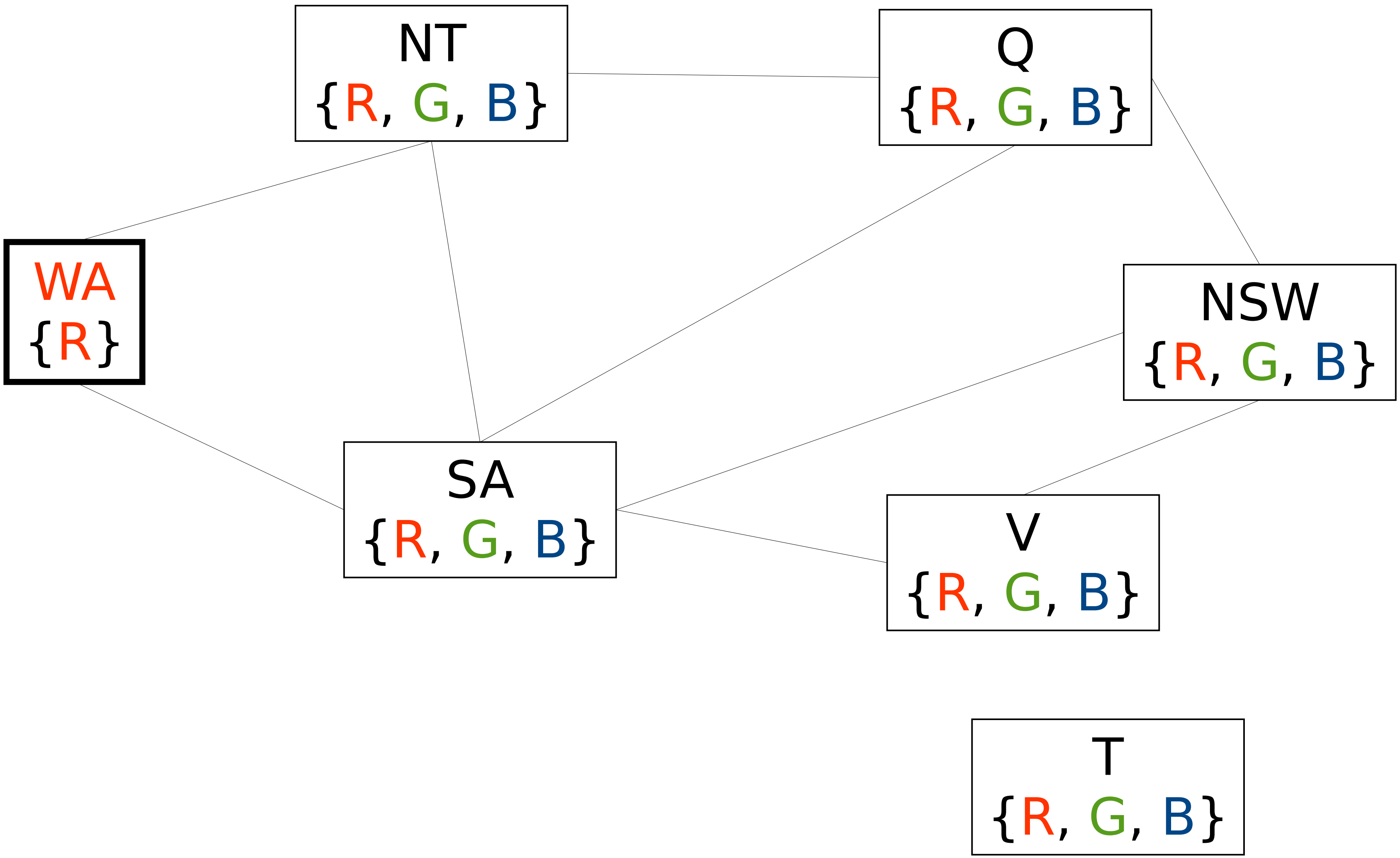
- **Idea**
  - After each assignment prune the domains of variables connected in the constraint graph.
- Algorithm
  - At the start, record the set of all legal values.
  - If you assign a variable, remove values that are now not legal anymore from the connected variables.
  - If a node's set of legal values becomes empty, then backtrack immediately.
- Time complexity $O(b^d)$ but often better.

Example: Map Coloring Problem



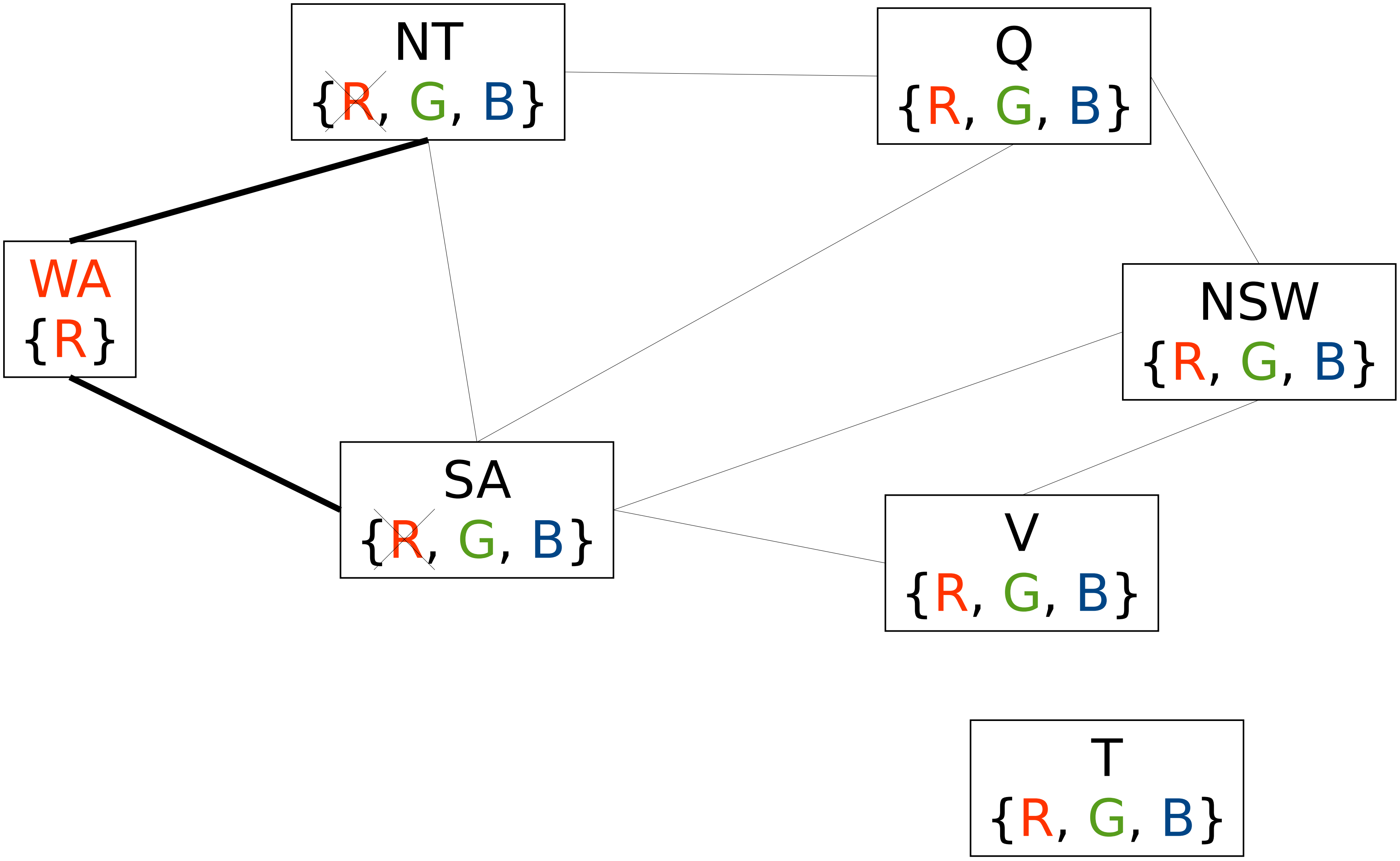
Example: Map Coloring Problem

- Assignment: WA = {R}



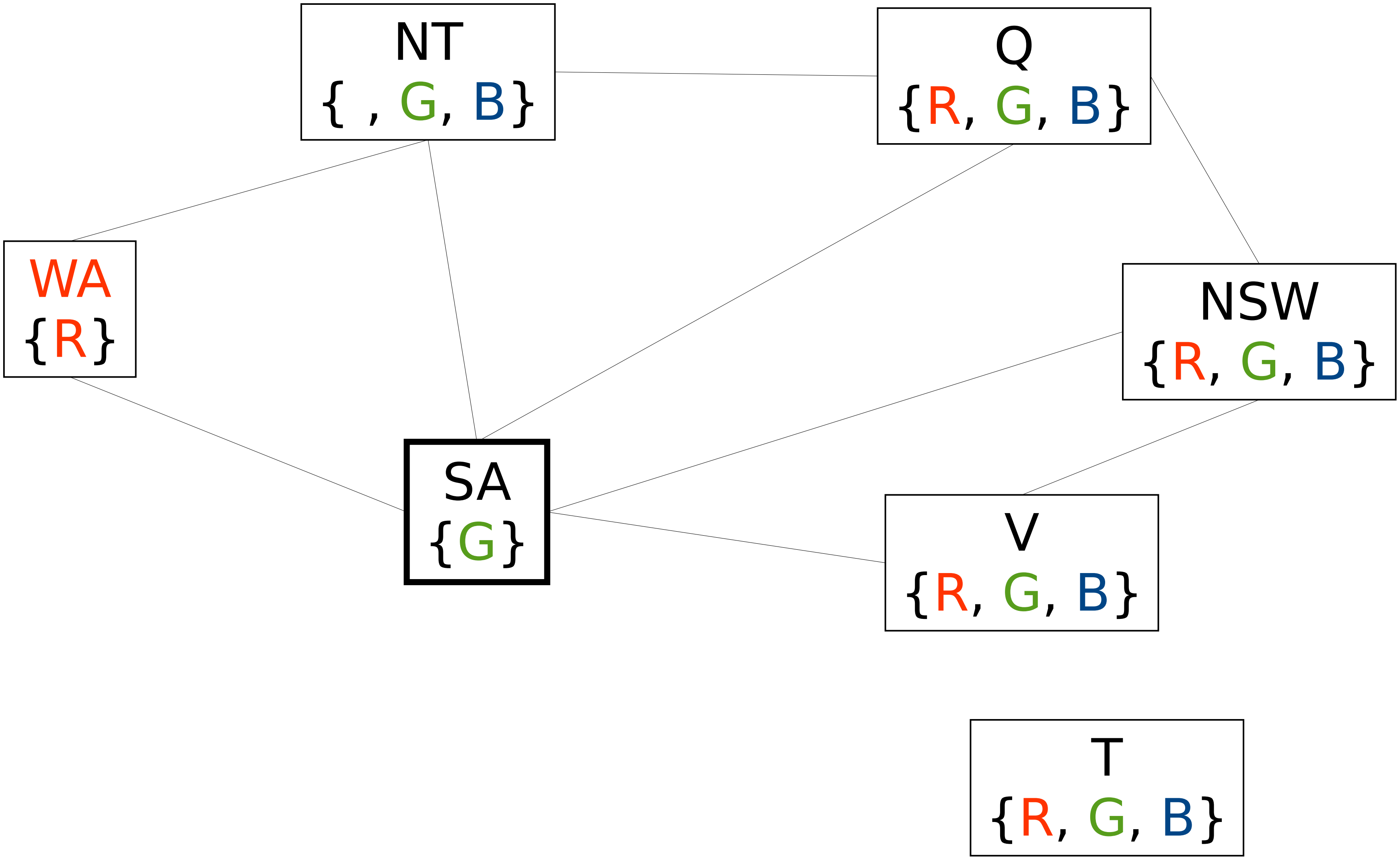
Example: Map Coloring Problem

- Constraint propagation



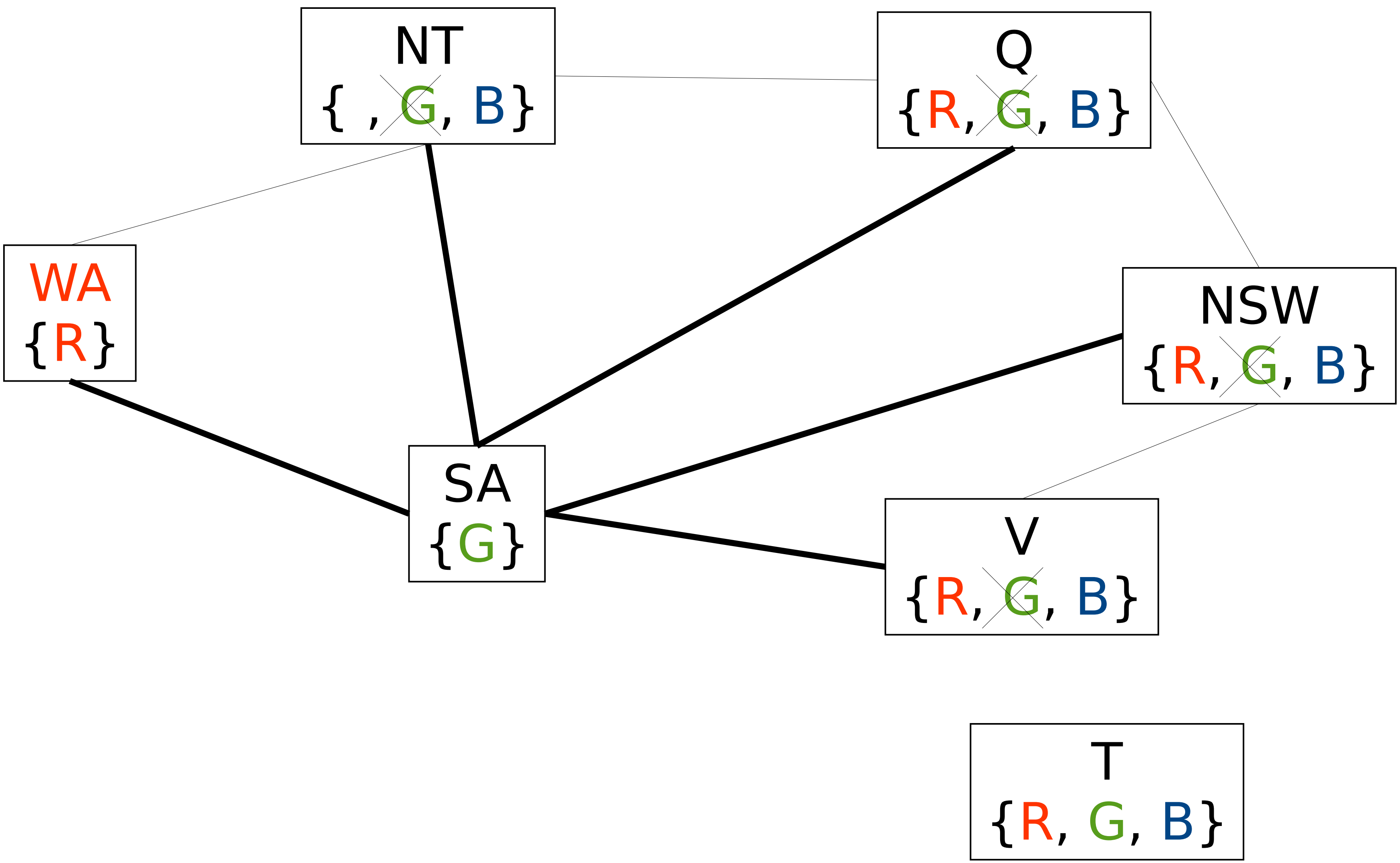
Example: Map Coloring Problem

- Assignment: SA = {G}



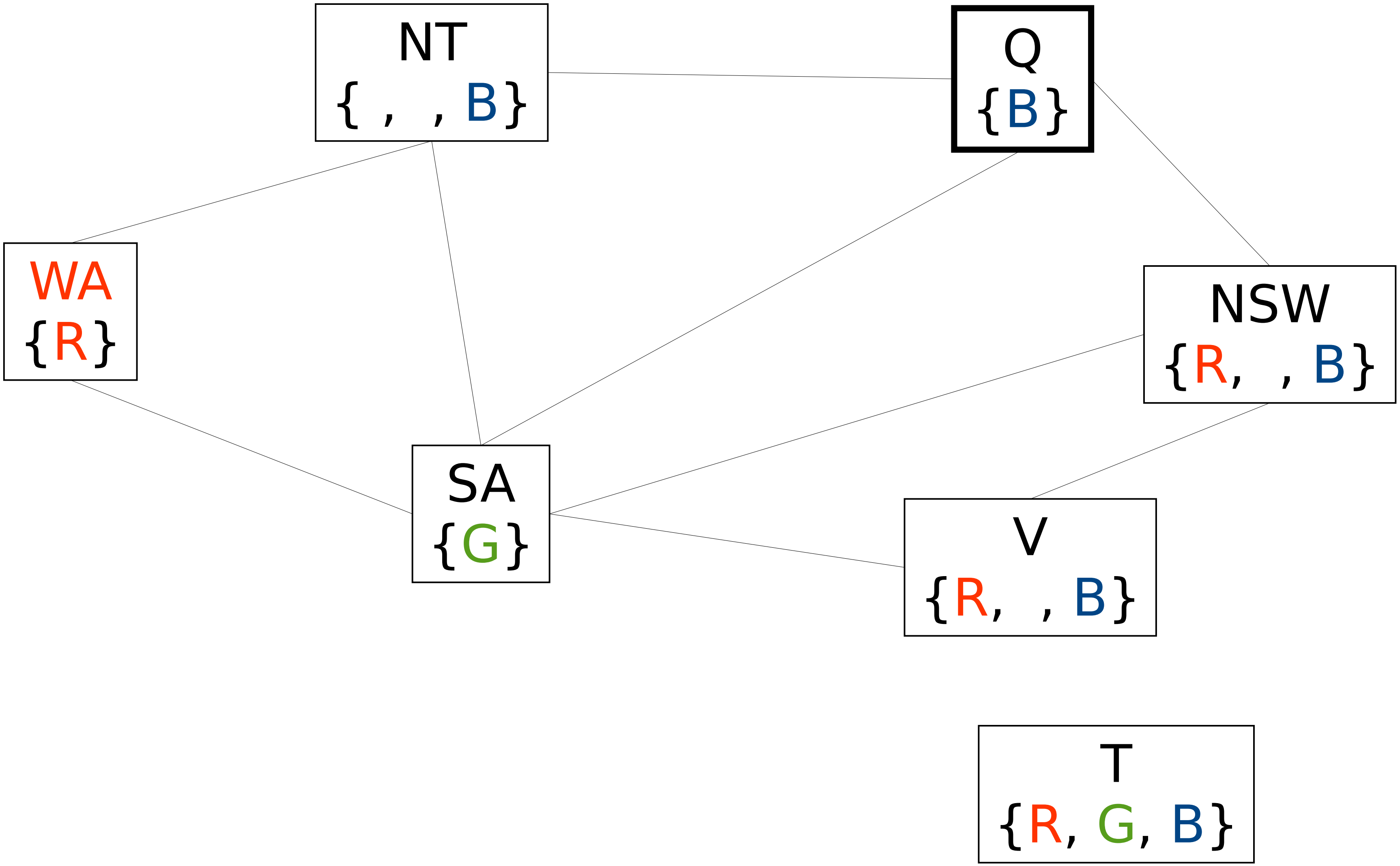
Example: Map Coloring Problem

- Constraint propagation



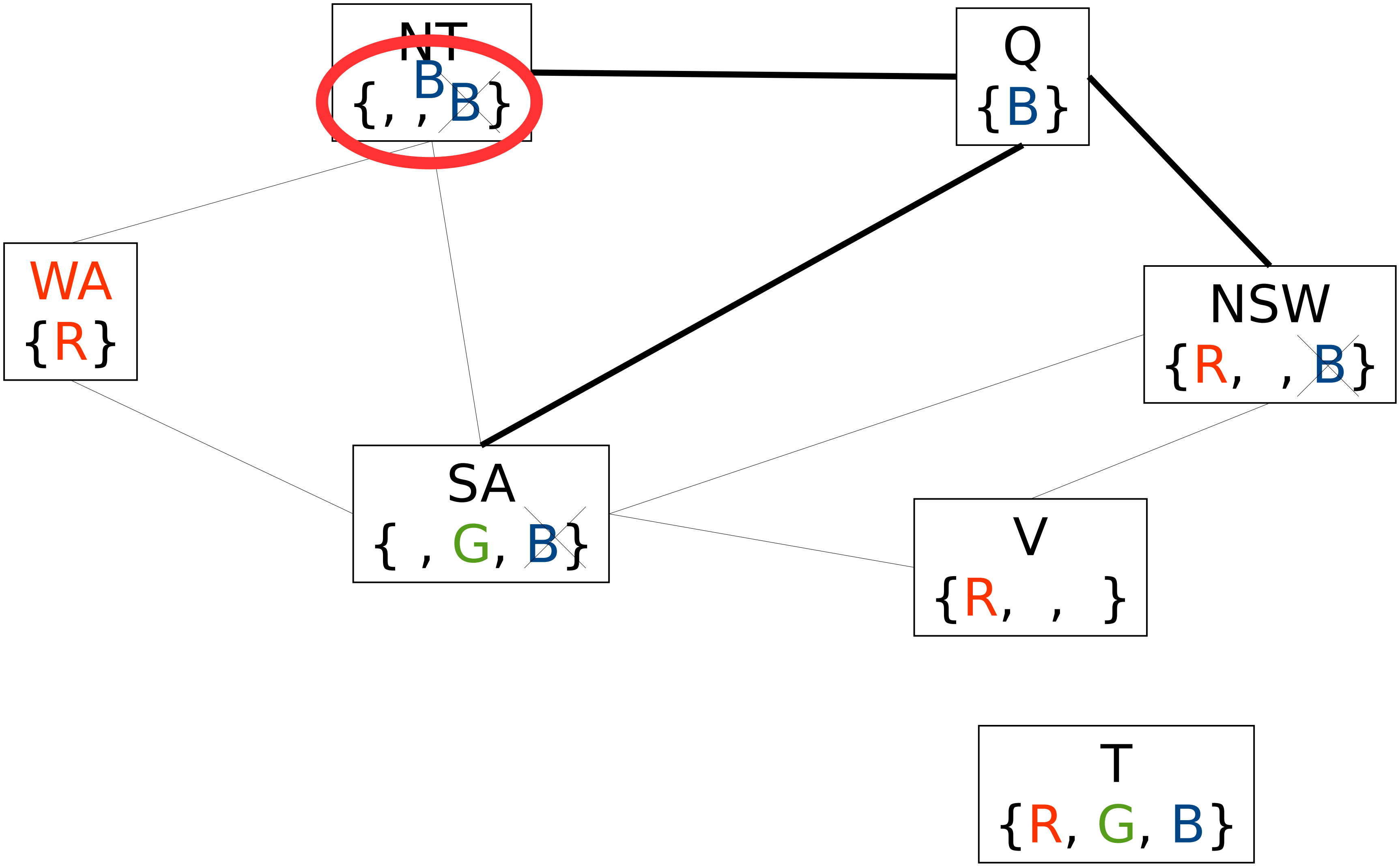
Example: Map Coloring Problem

- Assignment: Q = {B}



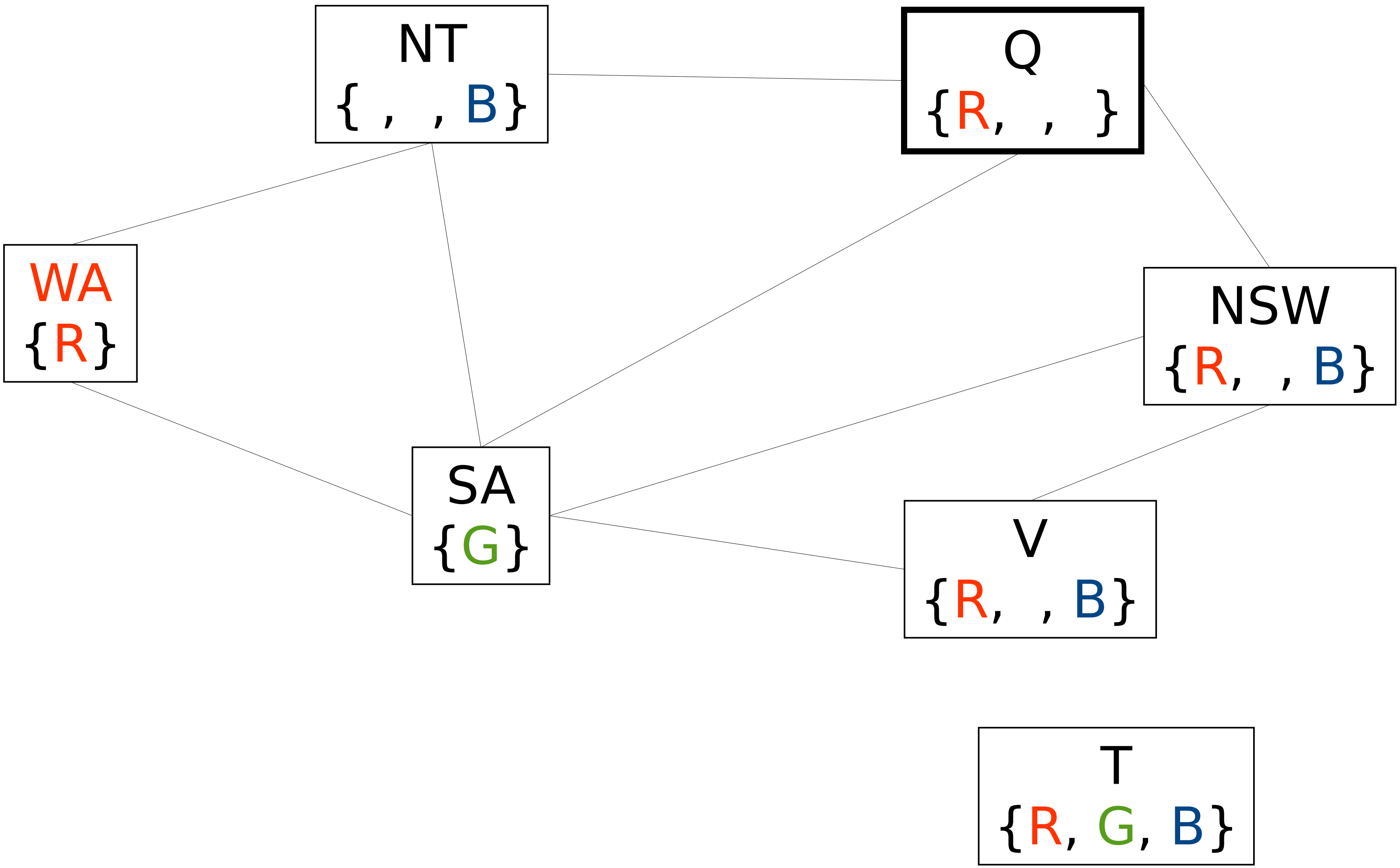
Example: Map Coloring Problem

- Constraint propagation



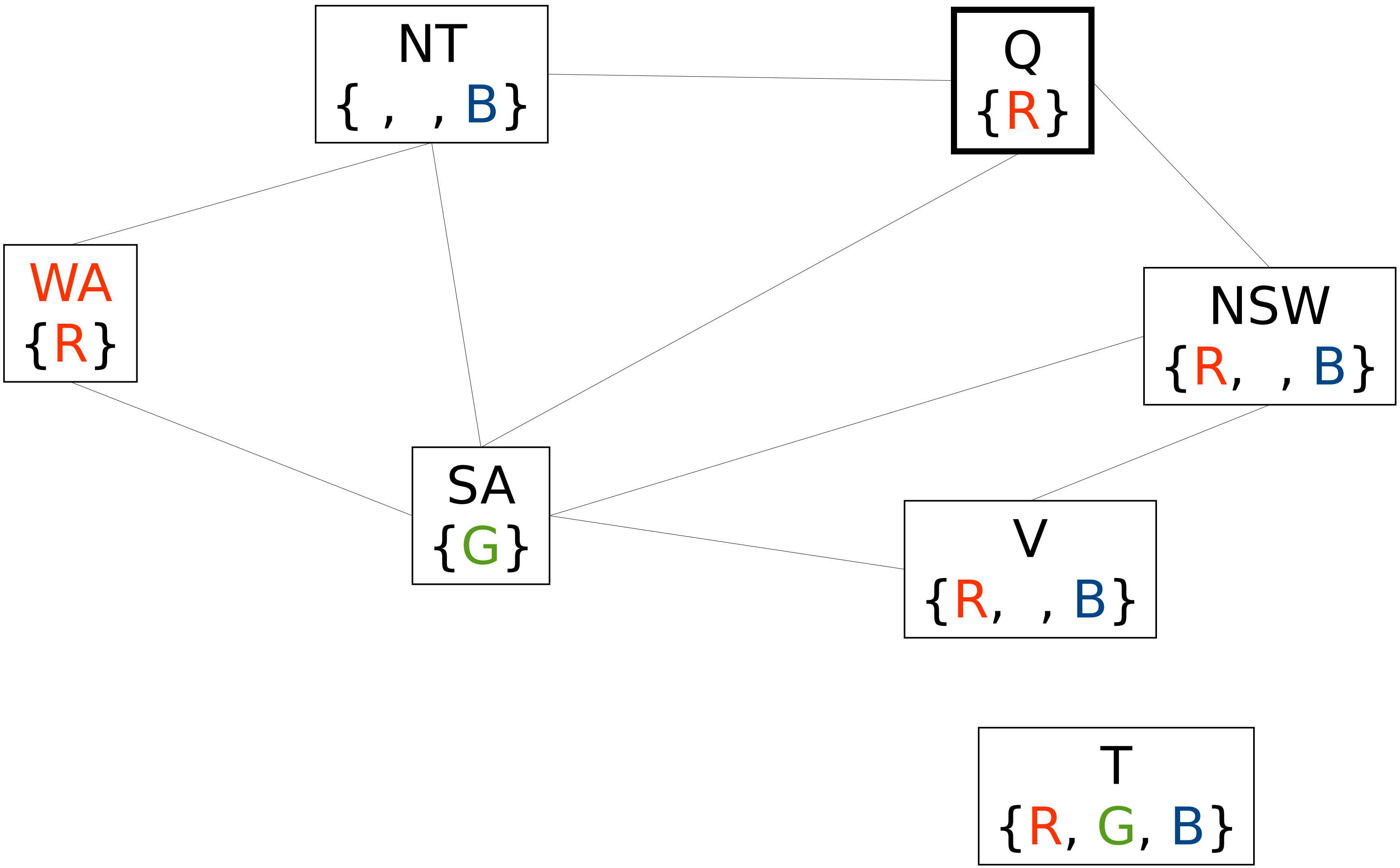
Example: Map Coloring Problem

- Backtrack to last assignment Q= {B}



Example: Map Coloring Problem

- Assignment: Q = {R}, etc



Forward Checking Algorithm

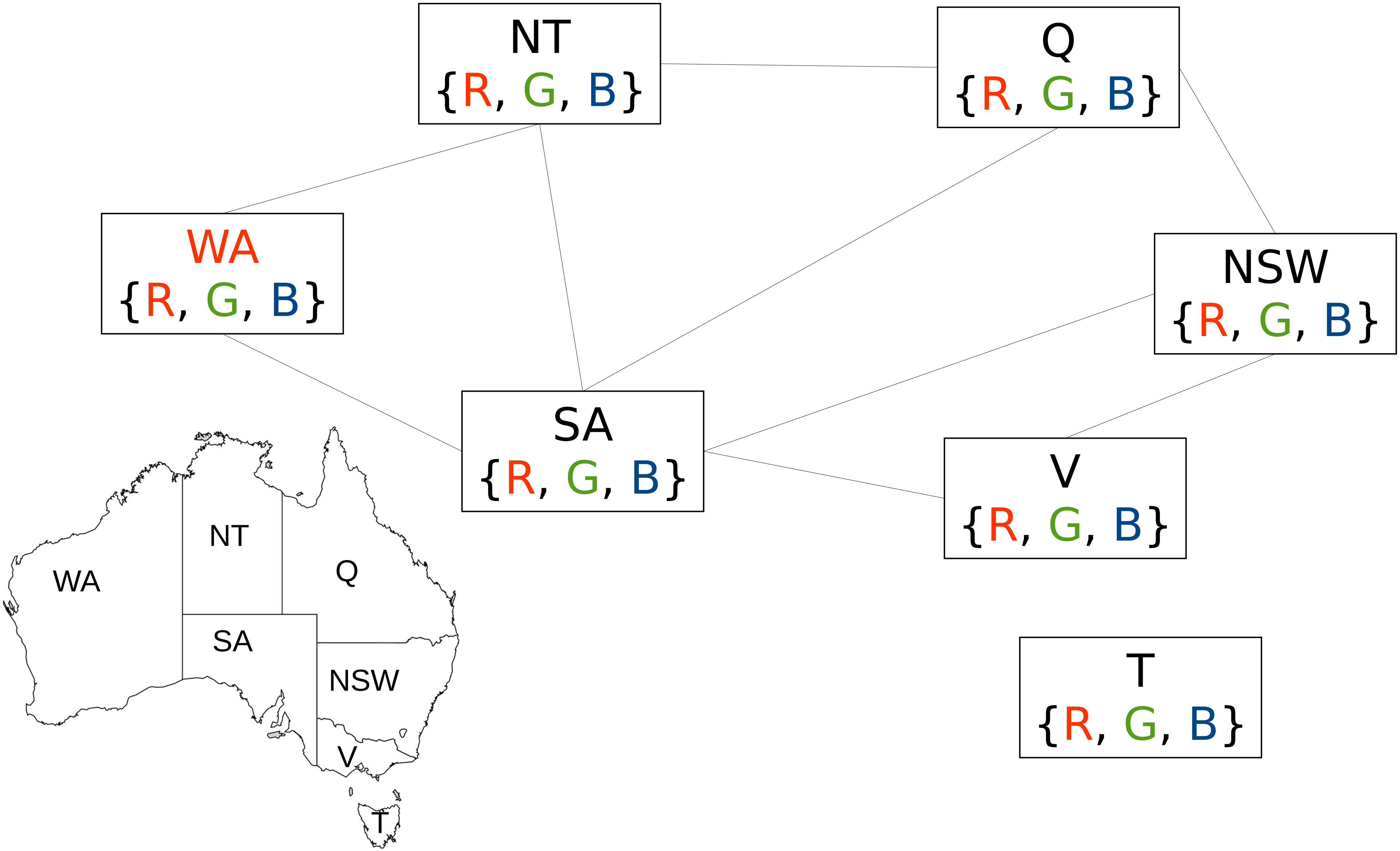

```
function FC-SEARCH(domains) returns solution/failure
  return RECURSIVE-FC-SEARCH({ }, domains)
function RECURSIVE-FC-SEARCH(assignment, domains) returns solution
  IF assignment is complete THEN return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(assignment, domains)
  FOREACH value in ORDER-DOMAIN-VALUES(var, assignment, domains) DO
      add {var = value} to assignment
      domains1 ← FORWARD-CHECKING(var, value, copy(domains))
      IF domains1 != failure THEN
         result ← RECURSIVE-FC-SEARCH(assignment, domains1)
         IF result != failure THEN return assignment
      remove {var = value} from assignment
  return failure
function FORWARD-CHECKING(var,value,domains) returns domains/failure
  FOREACH xi in domains whose values are constrained by var
    IF ∃ v in domains st xi=v () is inconsistent with var=value THEN
       remove v from the domain of xi in domains
       IF the domain of xi is empty THEN return failure
  return domains
```

[sudoku.sh]

2. Arc Consistency Algorithm

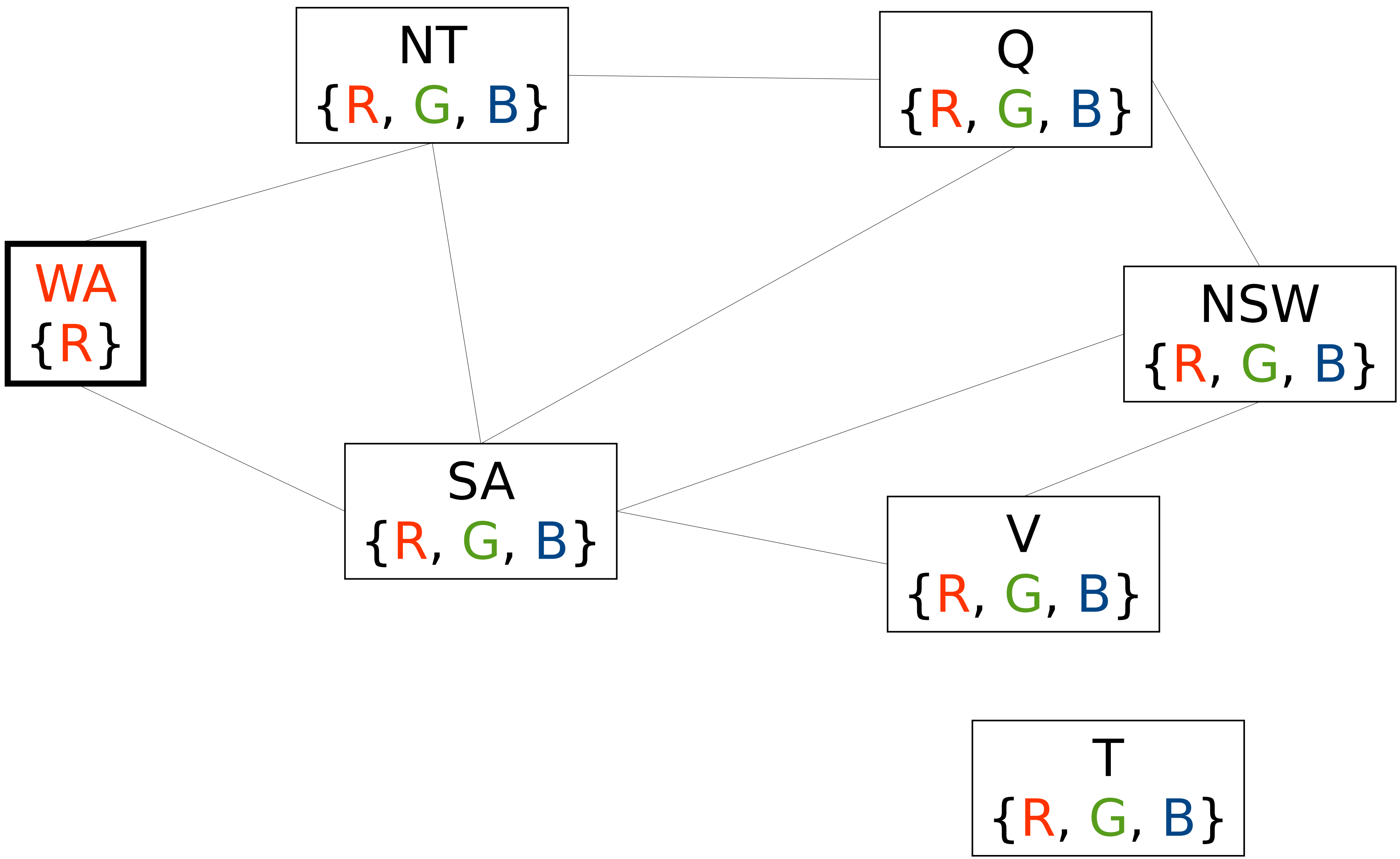
- Constant
  - Forward checking does not not look far enough forward After each variable assignment X_i, forward checking is limited to a propagation of constraints to the variables X_j directly connected to X in the constraint graph.
- Idea
  - After each assignment, prune the domains of all the connected variables and all variables connected to variable whose domain has been modified.
- Definition
  - 'Arc' refers to direct arc in the constraint graph.
  - $X \rightarrow Y$ is consistent iff for every value of X, there is some value of Y after applying all the constraints.
- Algorithm
  - Consider all arcs $X \rightarrow Y$.
  - Remove all values from X that makes the $X \rightarrow Y$ inconsistent.
  - If X looses a value, neighbors of X need to be rechecked.
  - If X is empty, then backtrack.
- Time complexity: $O(b^2d^3)$

Example: Map Coloring Problem



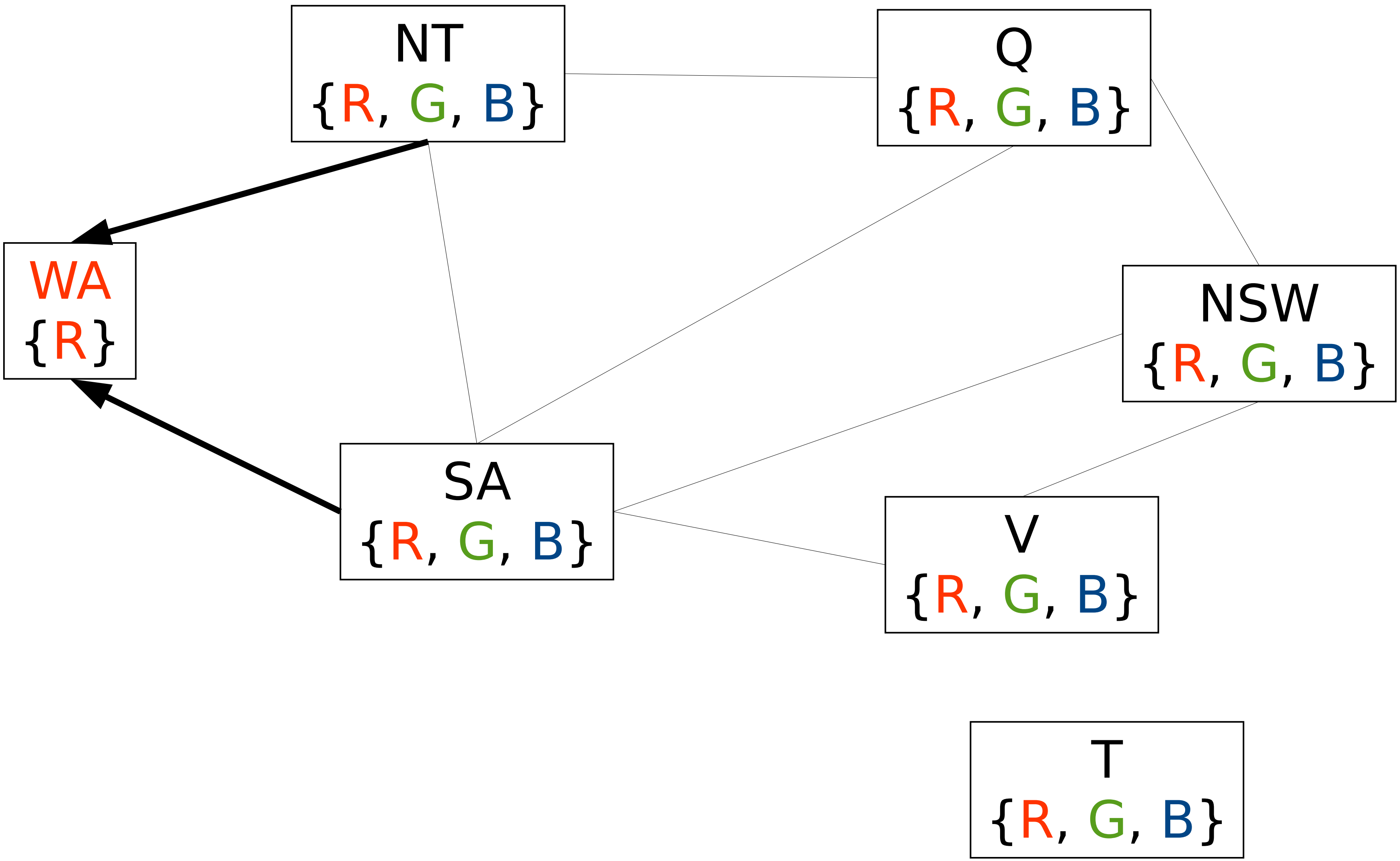
Example: Map Coloring Problem

- Assignment: WA = {R}



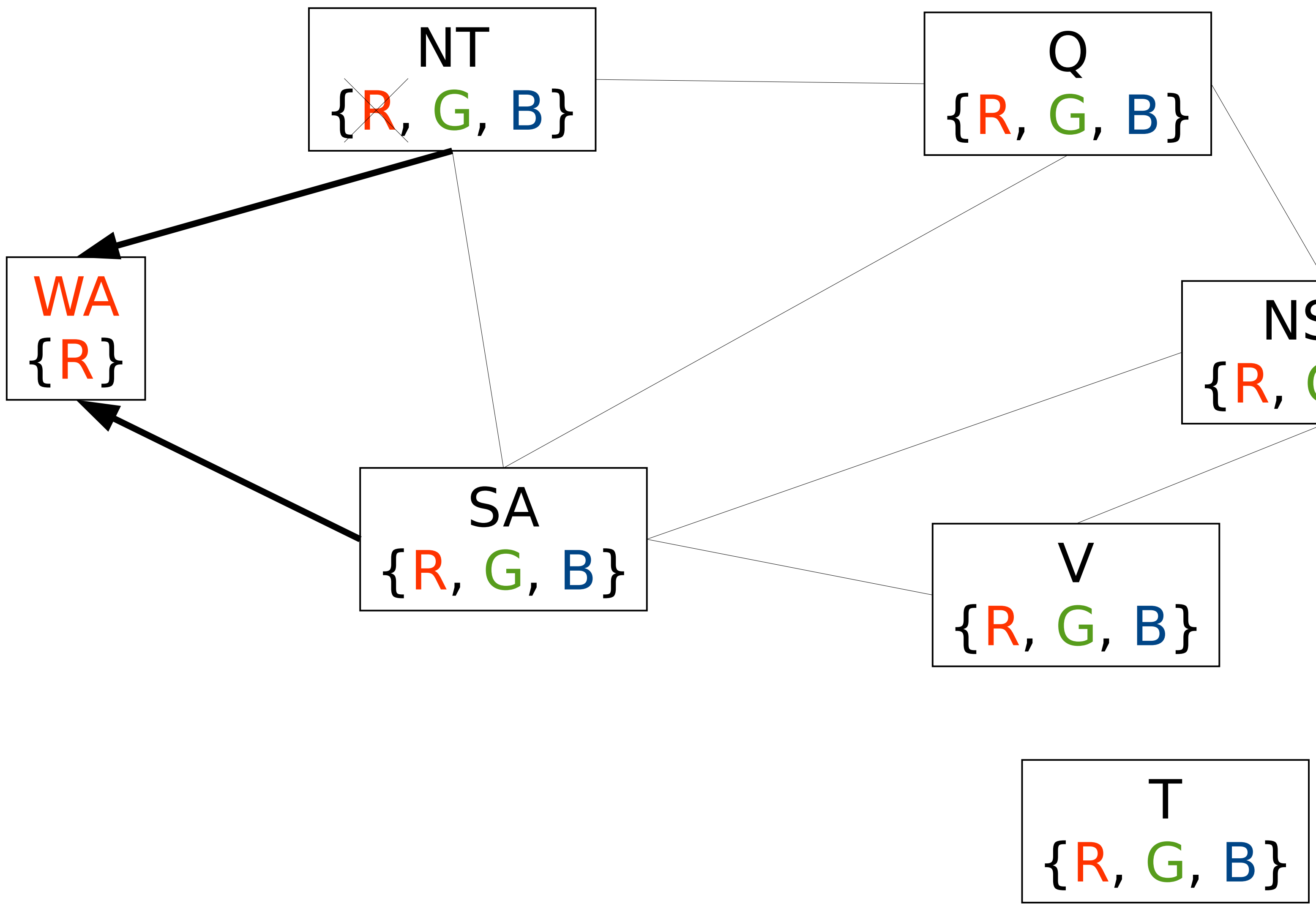
Example: Map Coloring Problem

- Arc consistency: check arcs {NT → WA, SA → WA}



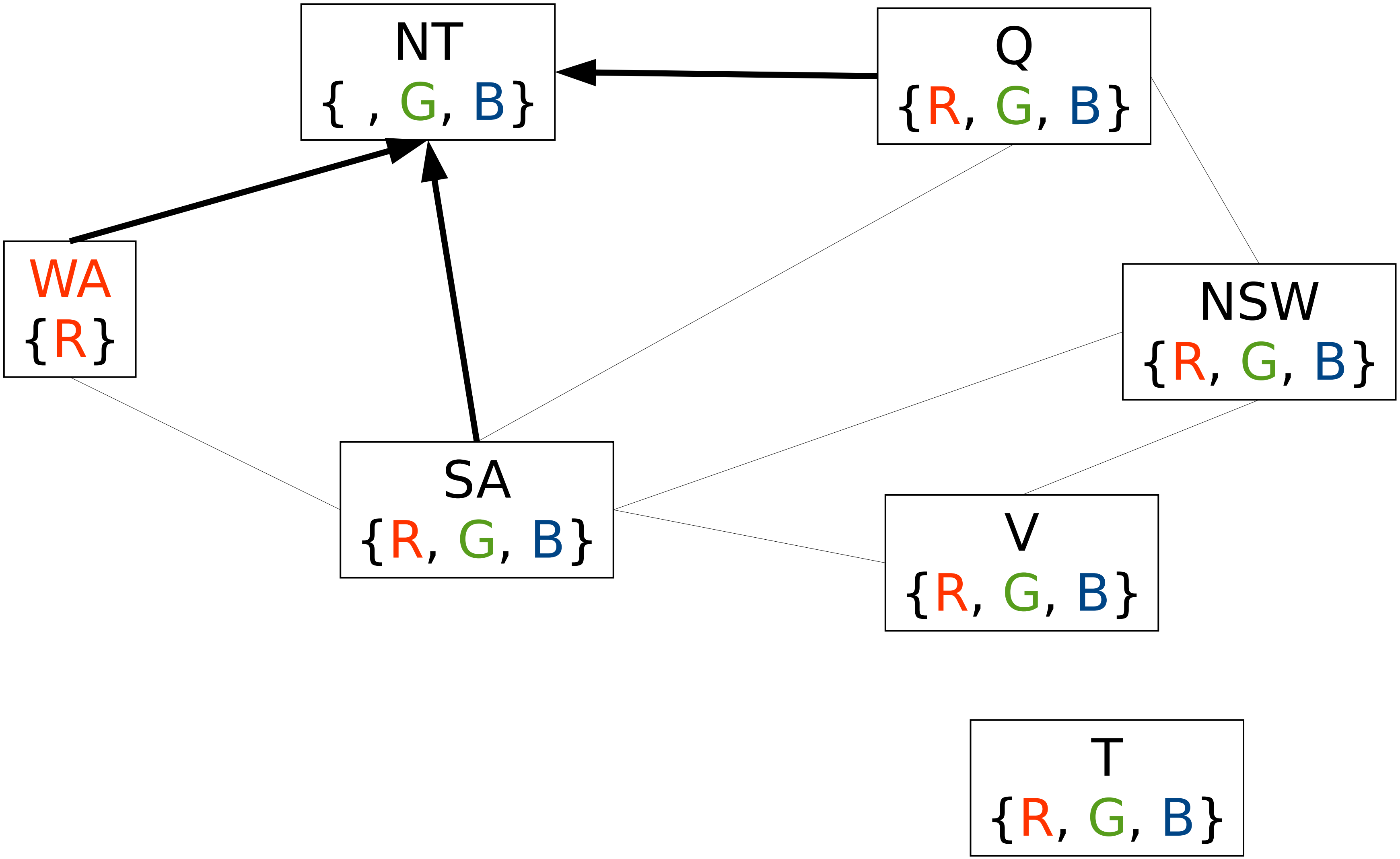
Example: Map Coloring Problem

- NT → WA: R makes the node inconsistent, so remove R from NT



Example: Map Coloring Problem

- Since NT domain was modified, add arcs WA→NT, SA→ NT, Q→NT
- In the list {SA → WA, WA→NT, SA→ NT, Q→NT}



Arc Consistency Algorithm: AC-3


```
function AC3(domains) returns domains
  queue ← all the arcs in domains
  WHILE queue is not empty DO
    (Xi, Xj) ← REMOVE-FIRST(queue)
    IF RM-INCONSISTENT-VALUES(Xi, Xj, domains) THEN
      IF domains[Xi] is empty THEN return failure
      FOREACH Xk in NEIGHBOURS[Xi] DO
        add (Xk, Xi) to queue
  return domains

function RM-INCONSISTENT-VALUES(Xi, Xj, domains) returns boolean
  removed ← false
  FOREACH x in domains[Xi] DO
   IF no value y in domains[Xj] allows (x,y) to satisfy constraint (Xi, Xj)
    THEN delete x from domains[Xi]
         removed ← true
  return removed
```

Using Arc Consistency

- Pros and cons
  - The number of backtracking is reduced with arc consistency (less steps).
  - The computation time is increased at each step with arc consistency.
- Arc consistency (eg, AC-3) can be used in two ways:
  - **Preprocessing**
    - Pruning the domain of variables before the beginning of the search process.
    - Then use forward checking for example.
  - **Maintaining Arc Consistency**
    - Propagation step after every assignment during search (like forward checking).
    - Add in the queue only the arc to the assigned variable.

[sudoku.sh]

Heuristics for CSPs

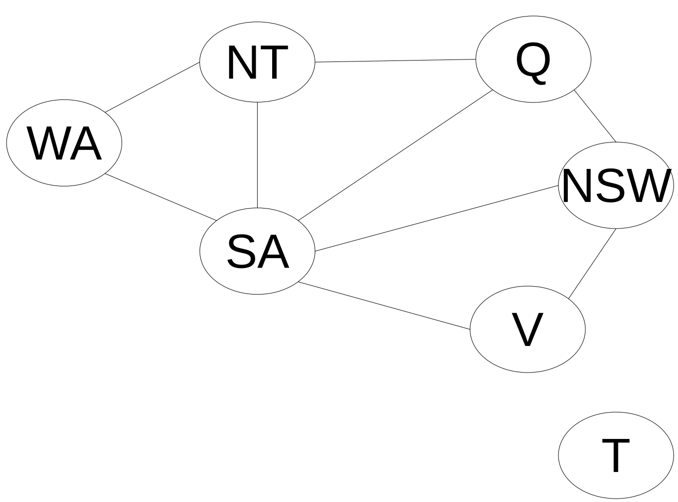
- Previous CSP algorithms leave two things unspecified:
  - Which variable to assign next? (ie, content of method `SELECT-UNASSIGNED-VARIABLE()` of the general algorithm)
  - Which value to choose first? (ie, content of method `ORDER-DOMAIN-VALUES()` of the general algorithm)
- Heuristics
  - Variable ordering
    - **Most constrained variable**
      - Choose the variable with the minimum remaining values.
    - **Degree heuristic**
      - Choose the variable involved in the largest number of constraints.

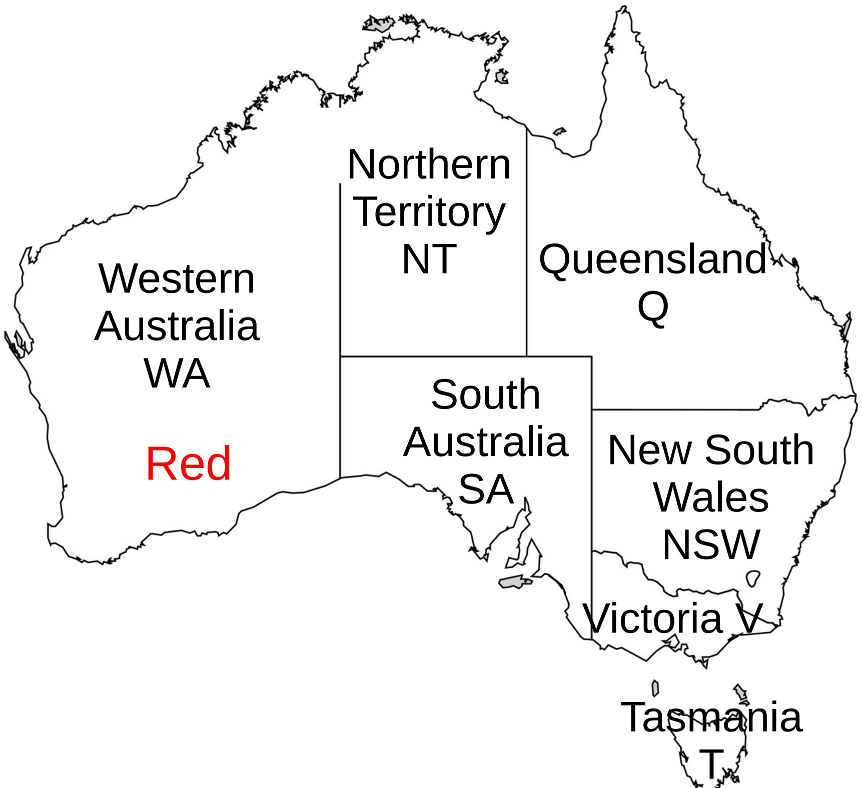
    → Constraint propagation will reduce the branching factor of search tree.
  - Value ordering
    - **Least constraining value**
      - Choose the value removing the least values from the domain of the neighbor variables

    → Try to leave the maximum flexibility for subsequent variable assignments.

Example. Map Coloring

- Heuristics
  - **Most constrained variable**
    - Countries **NT** and **SA** are the most constrained one (cannot use Red).
  - **Degree heuristic**
    - Country **SA** is the variable involved in the largest number of constraints.
  - **Least constraining value**
    - Red is the least constraining valid color for Q.





Conclusion

- Constraint networks consist of variables associated with finite domains and constraints.
  - A partial assignment maps some variables to values, a total assignment does so for all variables.
  - A partial assignment is consistent if it complies with all constraints.
  - A consistent total assignment is a solution.
- The constraint satisfaction problem (CSP) consists in finding a solution for a constraint network.
- In practice
  - Experimental results have shown that in most cases a good constraint propagation algorithm (like Forward Checking), preceded by Arc Consistency Checking with a good set of heuristics (like Minimum Remaining Values or Least Constraint Value) can go a long way in solving difficult CSP problems.

Demos

- Backtracking
  - Sudoku1-1.sh on grid 0 (explored states : **27**)
  - On other grids, no solution in reasonable time
- Forward Checking
  - Sudoku2-1.sh sur la grille 0 (time: 0s, explored states: **18**)
  - Sudoku2-2.sh sur la grille 1 (1s, **5942**)
- AC3 et MAC
  - Sudoku3-1.sh AC3 on grid 1 (0s, **81**)
  - Sudoku3-2.sh MAC on grid 1 (0.1s, **82**)
  - Sudoku3-3.sh AC3 on grid 5 (0.6s, **5343**)
  - Sudoku3-4.sh MAC on grid 5 : (0.6s, **460**)
- Heuristic
  - Sudoku4-1.sh FC without heuristic on grid 5 (1.8s, **10,074**)
  - Sudoku4-2.sh FC with heuristic on grid 5 (0s, **232**)
  - Sudoku4-3.sh AC3 with heuristic on grid 5 (0s, **232**)
  - Sudoku4-4.sh MAC with heuristic on grid 5 (longer: 0.1s, **96**)

https://fr.puzzle-binairo.com/ [sudoku.sh]