Fundamentals of Artificial Intelligence

Constraint Satisfaction Problems

Constraint Satisfaction Problems

- A **constraint satisfaction problem** consists of three components, X, D, and C:
  X is a set of variables, $\{X_1, \ldots, X_n\}$.
  D is a set of domains, $\{D_1, \ldots, D_n\}$, one for each variable.
  C is a set of constraints that specify allowable combinations of values.

- Each domain D_i consists of a set of allowable values, $\{v_1, \ldots, v_k\}$ for variable X_i.
- Each constraint C_i consists of a pair {scope, rel }, where *scope* is a tuple of variables that participate in the constraint and *rel* is a relation that defines the values that those variables can take on.
  - A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations.
  - If X_1 and X_2 both have the domain {A,B}, then the constraint saying the two variables must have different values can be written as $\langle (X_1,X_2), [(A,B), (B,A)] \rangle$ or as $\langle (X_1,X_2), X_1 \neq X_2 \rangle$

Constraint Satisfaction Problems

- In ***standard search problem***:
  - a state is a black box with no internal structure.
  - it that supports goal test, eval, successor.

- In **CSP**:
  - A **state** is defined by ***variables*** $\boldsymbol{X_i}$ ***with values from domain*** $\boldsymbol{D_i}$
  - A **goal test** is a ***set of constraints specifying allowable combinations of values for subsets of variables***

- CSP allows useful general-purpose algorithms with more power than standard search algorithms

Constraint Satisfaction Problems

- Each **state in a CSP** is defined by ***an assignment of values to some or all of the variables***, $\{X_i{=}v_i, X_j{=}\ v_j\ , \ldots\}$.
- An *assignment* that does not violate any constraints is called a **consistent (or legal) assignment**.
- A **complete assignment** is ***an assignment in which every variable is assigned***.
- A **solution to a CSP** is a **consistent, complete assignment**.
- A **partial assignment** is one that assigns values to only some of the variables.

- ***In order to solve a CSP, a consistent complete assignment must be found*** (be searched).

Example: Map-Coloring

- We are given the task of coloring each region of Australia either red, green, or blue in such a way that ***no neighboring regions have the same color***.

- To formulation as a CSP,

Variables: Each region is a variable: $X = \{WA, NT, Q, NSW, V, SA, T\}$.

Domains: The domain of each variable is the set $D_i = \{red, green, blue\}$.

Constraints require neighboring regions to have distinct colors.

- Since there are nine places where regions border, there are nine constraints:

$$C = \{SA \neq WA,\ SA \neq NT,\ SA \neq Q,\ SA \neq NSW,\ SA \neq V,$$
$$WA \neq NT,\ NT \neq Q, Q \neq NSW,\ NSW \neq V\}\ .$$

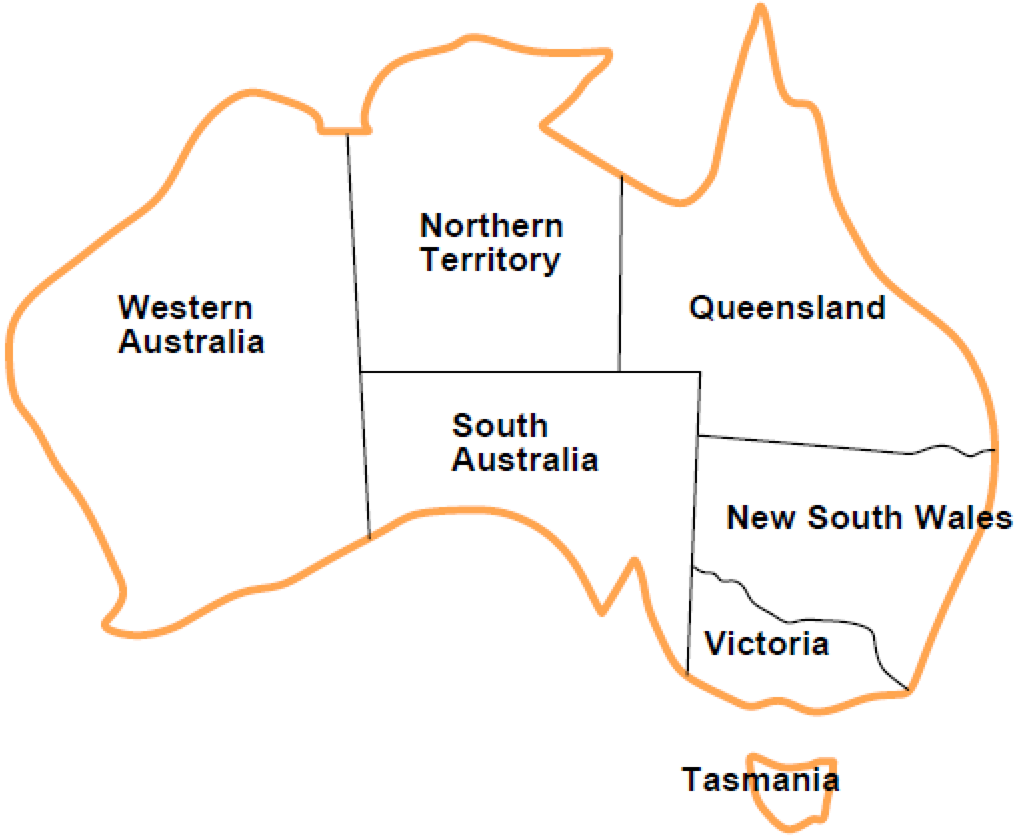
Example: Map-Coloring

Variables WA, NT, Q, NSW, V , SA, T

Domains D_i = {red, green, blue}

Constraints:

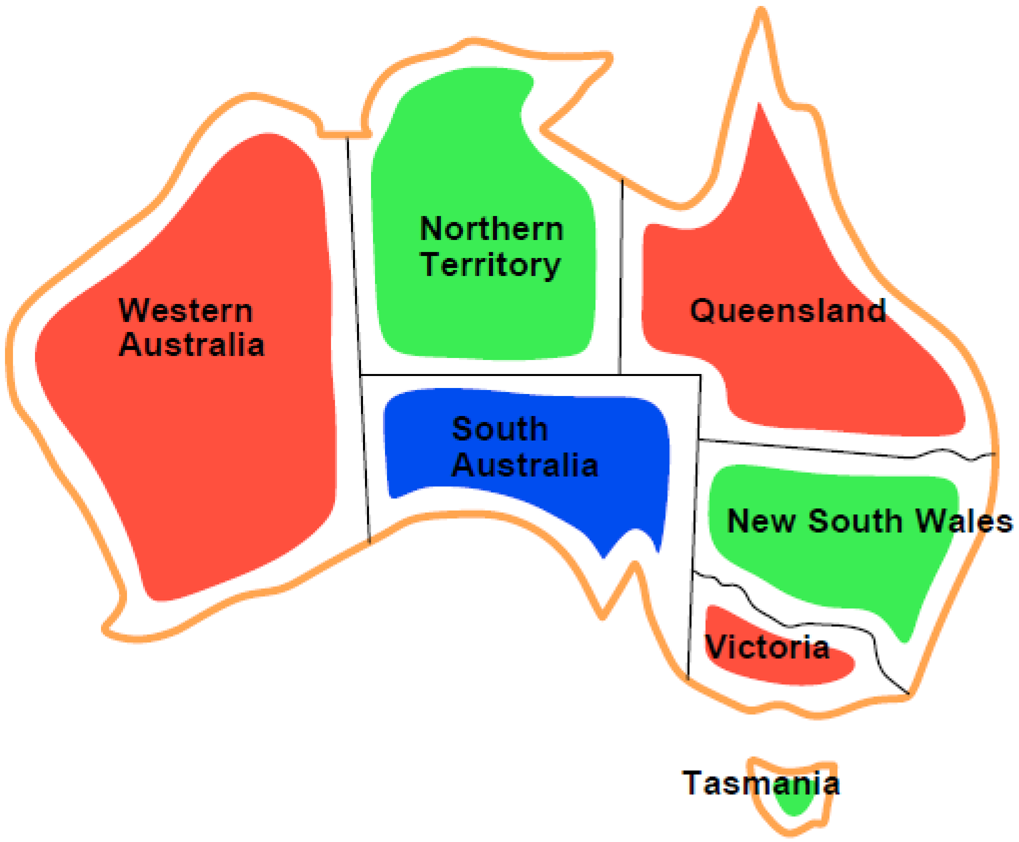
- adjacent regions must have different colors
- e.g., WA≠NT (if the language allows this), or (WA,NT) ∊ {(red,green), (red,blue),(green,red),…}



Example: Map-Coloring

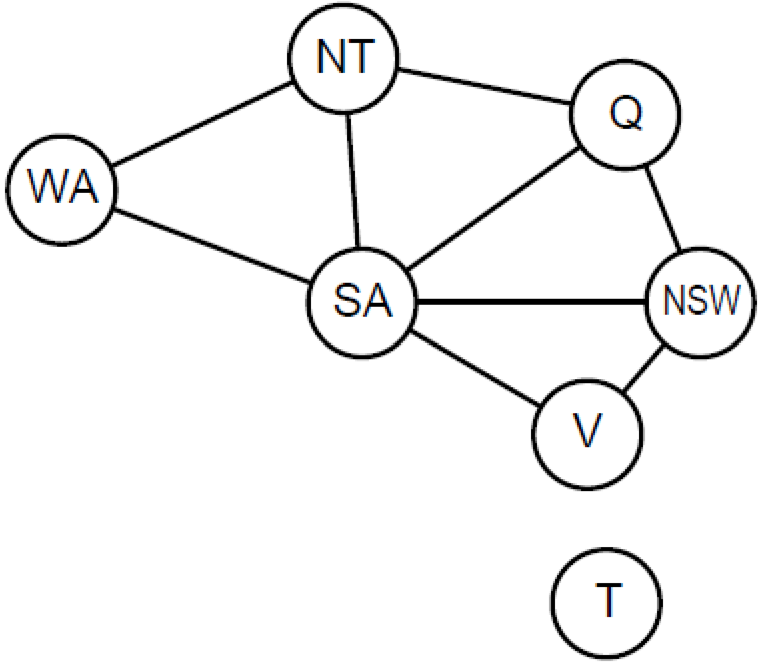
- **Solutions are assignments satisfying all constraints,**

  e.g., {WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}



Constraint Graph

- It can be helpful to visualize a CSP as a **constraint graph**.
- The **nodes of the constraint graph** correspond to ***variables of the problem***,
- A **link of the constraint graph** connects ***two variables in a constraint***. (**Binary CSP**)
- General-purpose CSP algorithms use the graph structure to speed up search.



Why formulate a problem as a CSP?

- CSPs yield a natural representation for a wide variety of problems.
- If we already have a CSP-solving system, it is often easier to solve a problem using it than to design a custom solution using another search technique.
- CSP solvers can be faster than state-space searchers because the CSP solver can quickly eliminate large portions of the search space.
    - For example, once we have chosen {SA=blue} in the Australia problem, we can conclude that none of the five neighboring variables can take on the value blue.
    - Without taking advantage of constraint propagation, a search procedure would have to consider 3^5 =243 assignments for the five neighboring variables;
    - With constraint propagation we never have to consider blue as a value, so we have only 2^5 =32 assignments to look at, a reduction of 87%.
- With CSPs, once we find out that a partial assignment is not a solution, we can immediately discard further refinements of the partial assignment.
- Many problems that are intractable for regular state-space search can be solved quickly when formulated as a CSP.

Variations on the CSP formalism
types of variables

- The simplest kind of CSP involves variables that have **discrete, finite domains**.
  - Map-coloring problems, scheduling with time limits and 8-queens problem are finite-domain CSP.
- A **discrete domain** can be **infinite**: the set of integers or the set of strings
- With **infinite domains**, *constraints cannot be enumerated by all allowed combinations of values*.
- With *infinite domains*, a **constraint language** must be used to describe constraints such as $T_1+d_1 \leq T_2$ directly, without enumerating the set of pairs of allowable values for (T_1, T_2).
- **Linear constraints** on integer variables are solvable.
- No algorithm exists for solving general **nonlinear constraints** on integer variables.
- **Continuous-domain CSPs with linear constraints** are *solvable in polynomial time* by **linear programming** methods.

Variations on the CSP formalism
types of constraints

- **Unary Constraint** restricts the value of a single variable.
  - Ex: <(SA),SA=green>
- **Binary Constraint** relates two variables.
  - For example, SA≠NSW is a binary constraint.
- A ***binary CSP*** can be represented as a *constraint graph*.
- **Higher-order constraints (global constraints)** involve 3 or more variables.
  - A CSP can be transformed into a CSP with only binary constraints
- Violation of **absolute constraints** rules out a potential solution.
- Many real-world CSPs include **preference constraints** indicating which solutions are preferred.
  - For example, in a university class-scheduling problem there are ***absolute constraints*** that no professor can teach two classes at the same time.
  - But we also may allow ***preference constraints***: Prof. X might prefer teaching in the morning, whereas Prof. Y prefers teaching in the afternoon.
  - ***CSPs with preferences*** can be solved with optimization search methods, and they are called as **constraint optimization problems.**

Example: Cryptarithmetic

- In a **Cryptarithmetic puzzle**, each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct.

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

- The **global constraint** *Alldiff* (F, T,U,W,R,O) represents distinction of each letter.
- The constraints on the four columns of the puzzle can be written as n-ary constraints.

$$\begin{aligned} &O + O = R + 10 \cdot C_{10} \\ &C_{10} + W + W = U + 10 \cdot C_{100} \\ &C_{100} + T + T = O + 10 \cdot C_{1000} \\ &C_{1000} = F\ , \end{aligned}$$

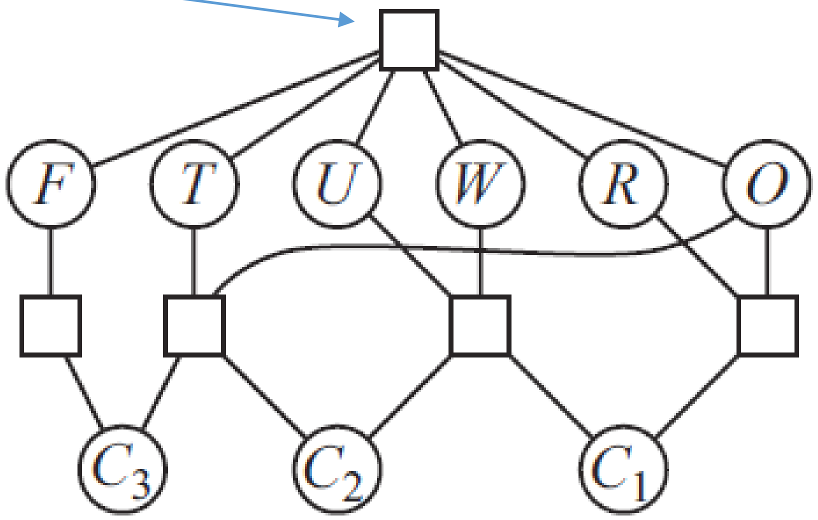
where C_{10}, C_{100}, and C_{1000} are auxiliary variables representing the digit carried over into the tens, hundreds, or thousands column.

Example: Cryptarithmetic

- **Constraints** can be represented in a **constraint hypergraph**,
- A **hypergraph** consists of ordinary nodes and hypernodes which represent n-ary constraints.

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

Alldiff (F, T,U,W,R,O)



$$\begin{aligned} &O + O = R + 10 \cdot C_{10} \\ &C_{10} + W + W = U + 10 \cdot C_{100} \\ &C_{100} + T + T = O + 10 \cdot C_{1000} \\ &C_{1000} = F\ , \end{aligned}$$

Real-world CSPs

- Assignment problems
  - e.g., who teaches what class
- Timetabling problems
  - e.g., which class is offered when and where?
- Hardware configuration
- Transportation scheduling
- Factory scheduling

- Many real-world problems involve real-valued variables

Search Formulation for CSPs

States are defined by the values assigned so far (partial assignments).

Initial State: the empty assignment, {}

Successor Function: assign a value to an unassigned variable that does not conflict with current assignment.

- If there is no legal assignments, cause failure

Goal Test: the current assignment is complete and satisfies all constraints.

- ie. a goal state is a complete and consistent assignment

Naïve Formulation:

- This search formulation is the same for all CSPs!
- Every solution appears at depth n with n variables ➔ use ***depth-first search***
- There are ***d^n complete assignments*** for a CSP with n variables of domain size d,.
- **Branching factor:** ***nd*** at the first level and ***(n-d)h*** at depth h, hence ***$n!d^n$*** leaves!
    - It is not so practical.

Commutativity

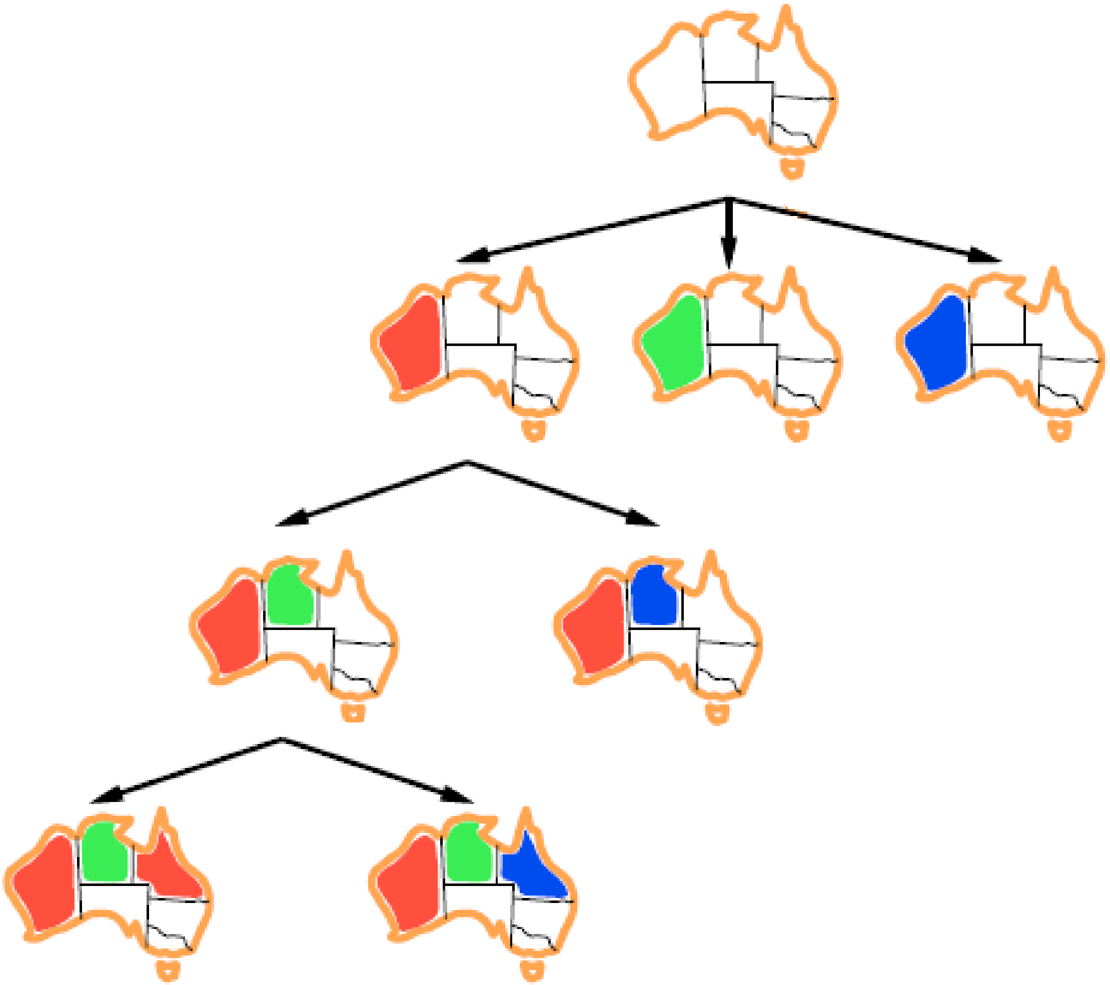
- A problem is **commutative** if the order of application of any given set of actions has no effect on the outcome.
- CSPs are **commutative** because when assigning values to variables, we reach the same partial assignment regardless of order.
- We need only consider a single variable at each node in the search tree.
  - For example, at the root node of a search tree for coloring the map of Australia, we might make a choice between SA=red, SA=green, and SA=blue, but we would never choose between SA=red and WA=blue.
- With this restriction (commutative property of CSPs), ***the number of leaves is d^n***.
- ***All CSP search algorithms consider a single variable assignment at a time***, thus there are d^n leaves.
- The **backtracking search** for CSPs is ***a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign***.
- ***Backtracking search*** is the basic uninformed algorithm for CSPs.

Backtracking Search

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/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 given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

Backtracking Search: Example



Improving Backtracking Efficiency

- General-purpose methods can give huge gains in speed:

    1. Which variable should be assigned next?

    2. In what order should its values be tried?

    3. Can we detect inevitable failure early?

    4. Can we take advantage of problem structure?

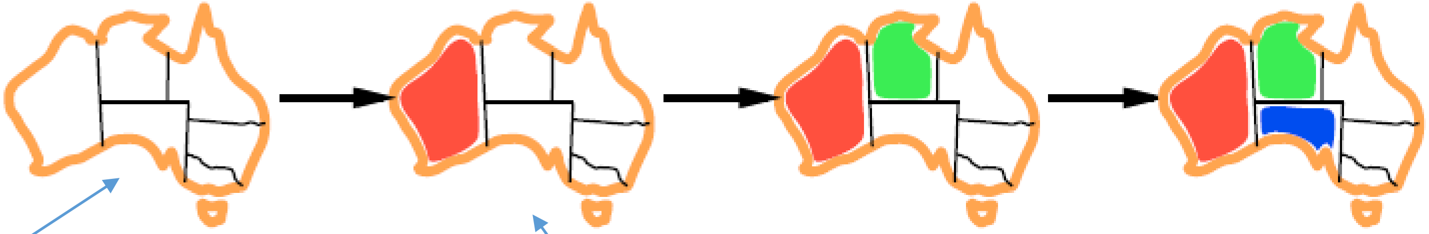
Minimum Remaining Values

a heuristic for variable selection

- The simplest strategy for ***variable selection*** is to choose the next unassigned variable in order, $\{X_1, X_2, \ldots\}$.
  - This static variable ordering may NOT produce an efficient search.

Minimum Remaining Values (MRV) Heuristic:

Choose the variable with the fewest legal values.



all of them selectable here

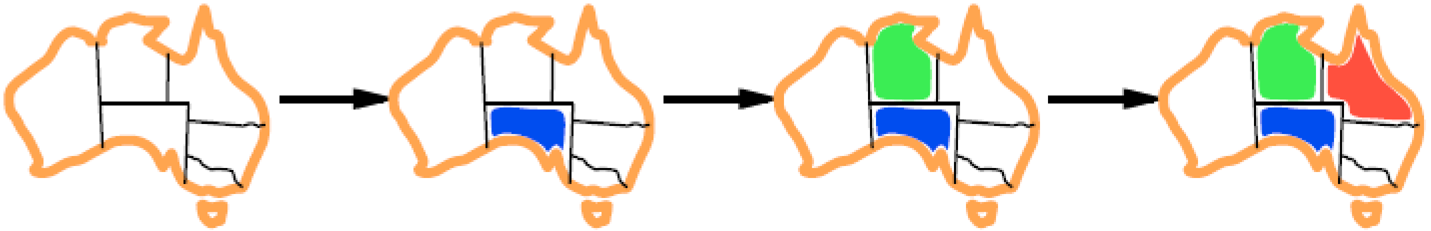
Select NT or SA here

Degree Heuristic
a heuristic for variable selection

- A tie-breaker among MRV variables

Degree Heuristic:

- choose the variable with the most constraints on remaining variables



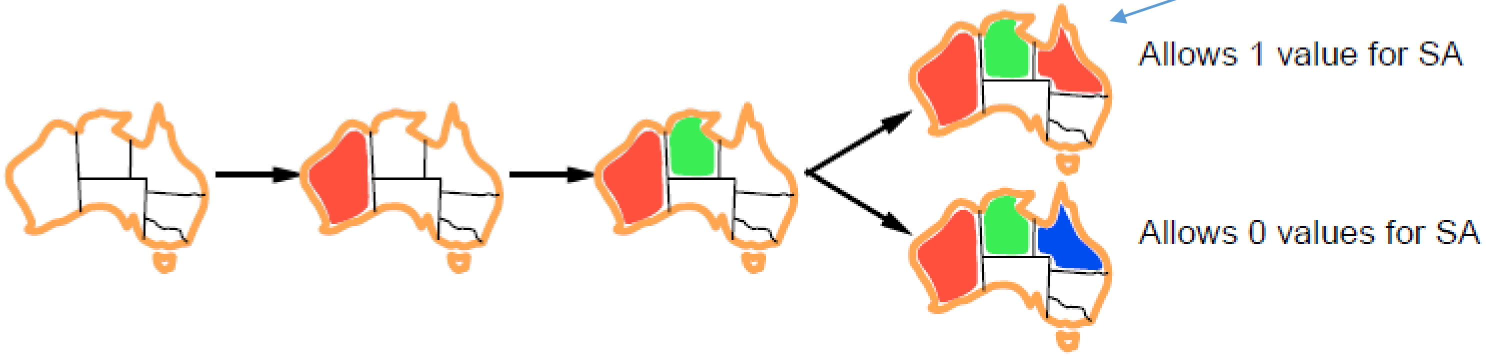
- The minimum-remaining values heuristic is usually a more powerful guide, but the degree heuristic can be useful as a tie-breaker.

Least-Constraining-Value
a heuristic for value selection

- Once a variable has been selected, the algorithm must decide on the order in which to examine its values.

- Given a variable, **least constraining value** heuristic prefers the *value that rules out the fewest choices for the neighboring variables in the constraint graph*.

LCV heuristic prefers this one



Variable and Value Ordering

- ***Minimum Remaining Values (MRV) heuristic picks a variable that is most likely to cause a failure soon, thereby pruning the search tree.***
  - MRV also has been called the **most constrained variable** or **fail-first** heuristic.
  - If some variable X has no legal values left, the MRV heuristic will select X and failure will be detected immediately—avoiding pointless searches through other variables.

- ***Least-Constraining-Value heuristic picks a value that is most likely to cause a failure last.***
  - We only need one solution; therefore it makes sense to look for the most likely values first.
  - If we wanted to enumerate all solutions rather than just find one, then value ordering would be irrelevant.

Interleaving Search and Inference

- Some **inference algorithms** can ***infer reductions in the domain of variables*** before we begin the search

- **Inference algorithms** can infer new domain reductions on the neighboring variables every time we make a choice of a value for a variable.

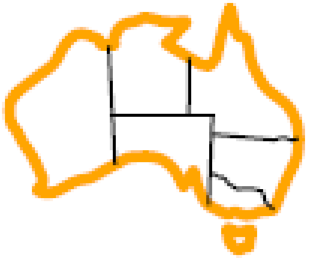
Inference: Forward Checking

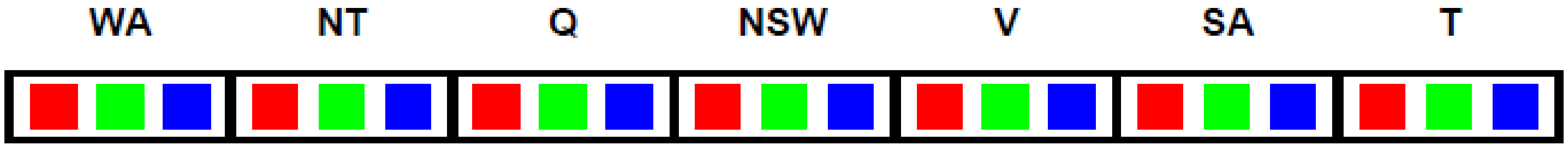
- Whenever a variable X is assigned, the **forward-checking process** establishes **arc consistency** for it:
  - For each unassigned variable Y that is connected to X by a constraint, delete from Y ’s domain any value that is inconsistent with the value chosen for X.
  - Because forward checking only does arc consistency inferences, there is no reason to do forward checking if we have already done arc consistency as a preprocessing step.

- For many problems the search will be more effective if we combine the MRV heuristic with forward checking.

- ***Forward Checking Idea:*** Keep track of remaining legal values for unassigned variables and terminate search when any variable has no legal values.

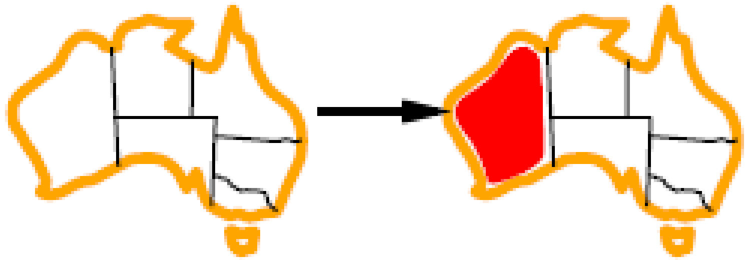
Forward Checking

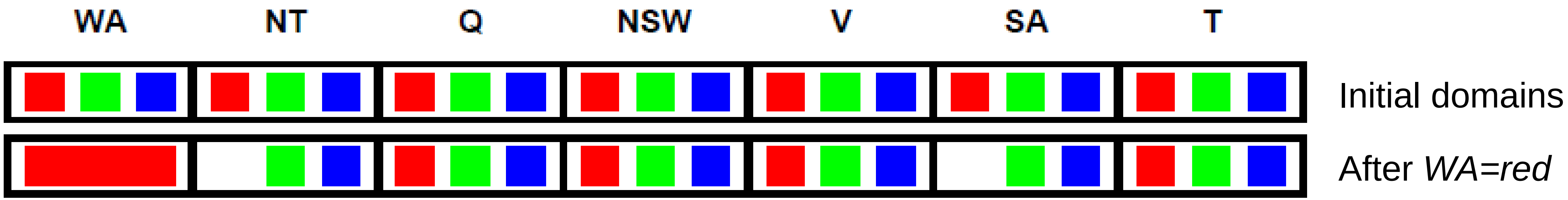




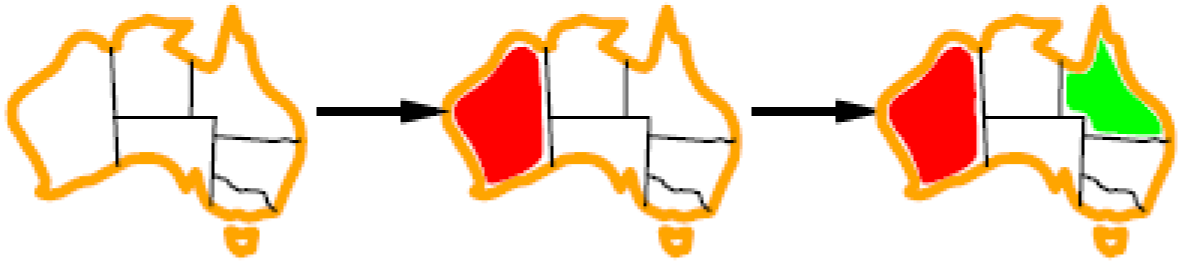
Initial domains

Forward Checking

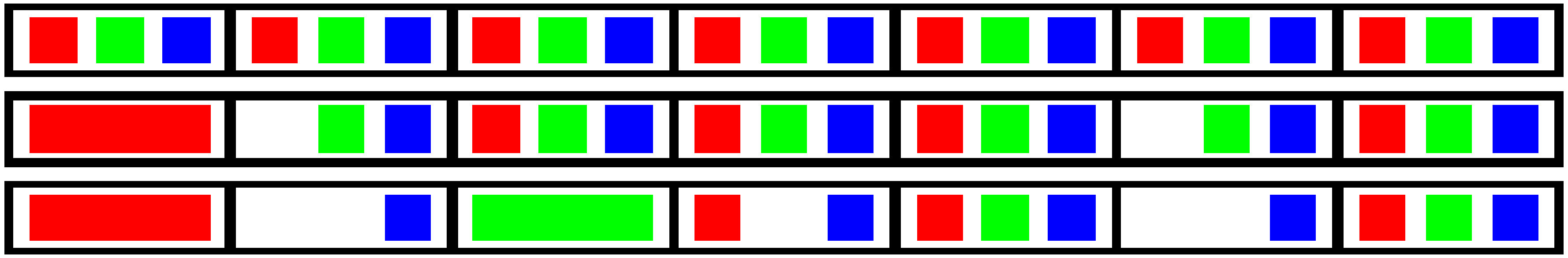




Forward Checking



WA NT Q NSW V SA T

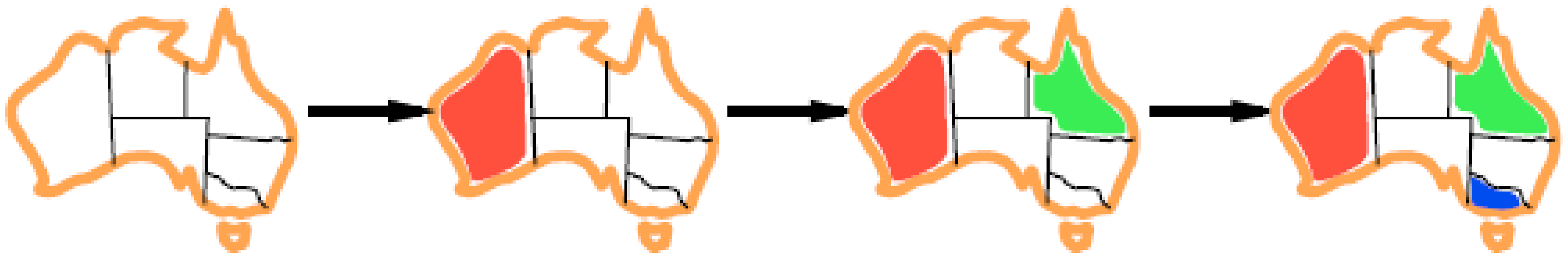


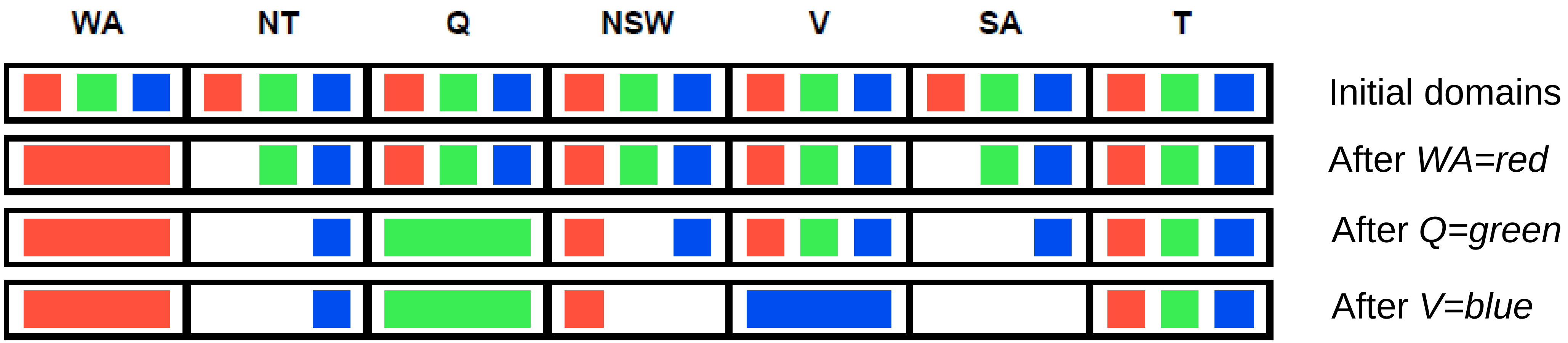
Initial domains

After *WA=red*

After *Q=green*

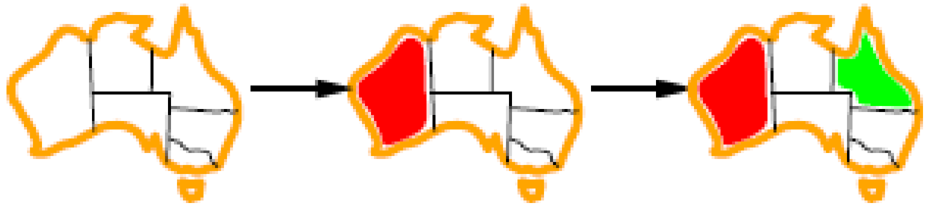
Forward Checking





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** repeatedly enforces constraints locally

Arc Consistency

- Simplest form of **constraint propagation** makes each arc **consistent.**
- A variable in a CSP is **arc-consistent** if every value in its domain satisfies the variable's binary constraints.
- X_i is **arc-consistent** with respect to another 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).

- $X \rightarrow Y$ is consistent iff

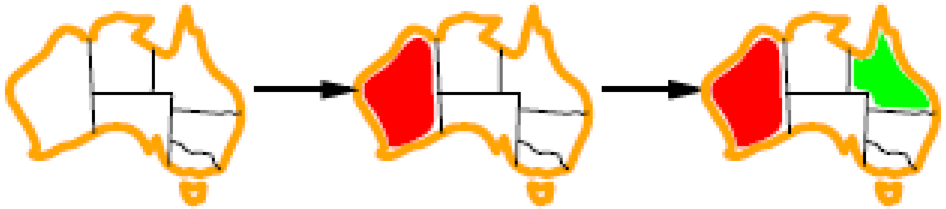
  for every value x of X there is some allowed y for Y

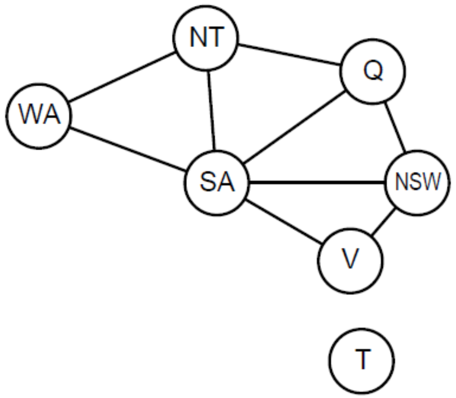
- ***A network is arc-consistent*** if every variable is arc consistent with every other variable.

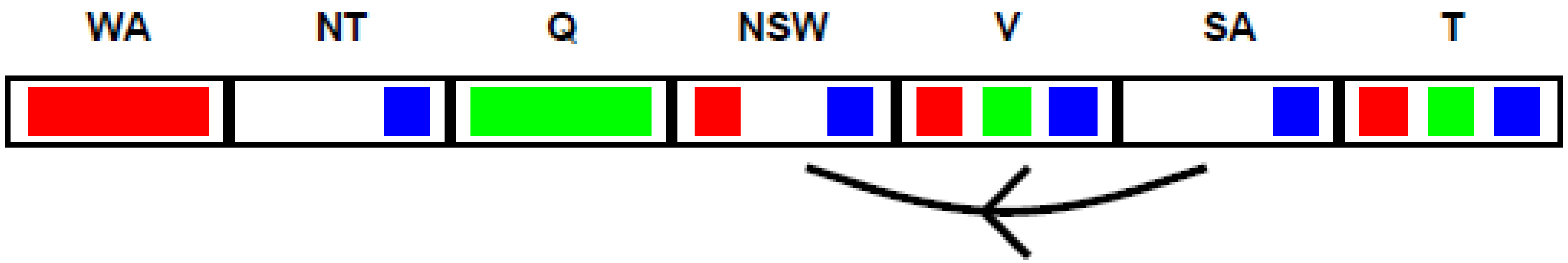
Arc Consistency

$X \rightarrow Y$ is consistent iff

for every value x of X there is some allowed y for Y



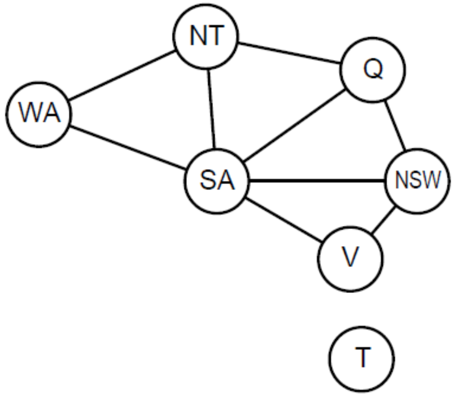


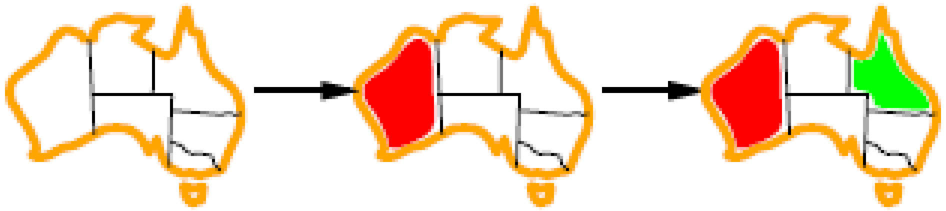


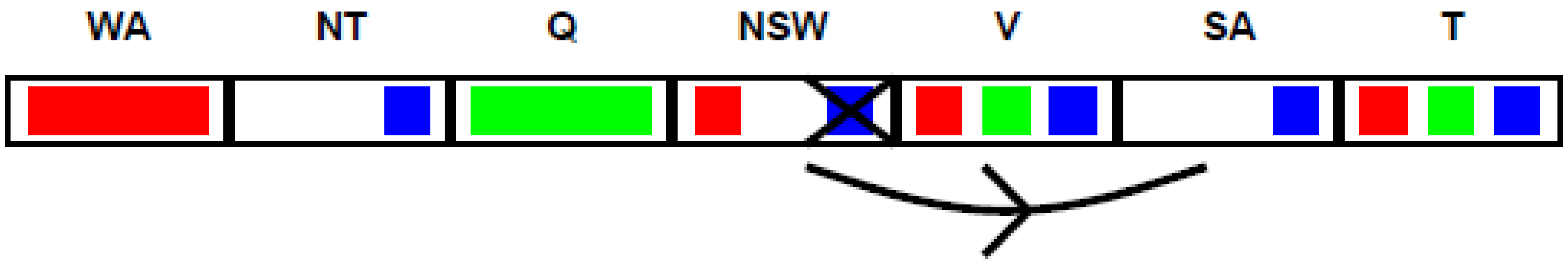
Arc Consistency

X → Y is consistent iff

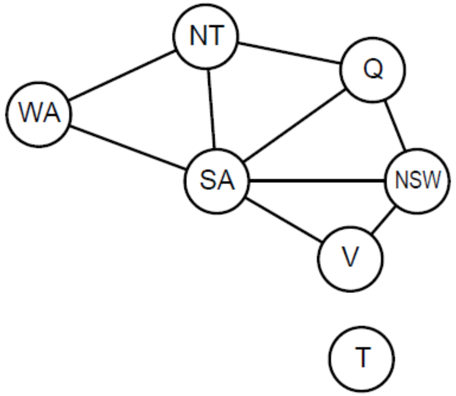
for every value x of X there is some allowed y for Y





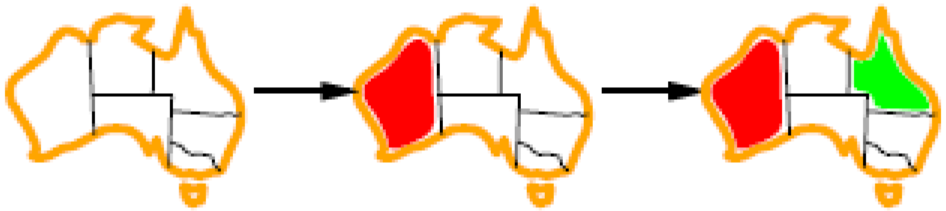


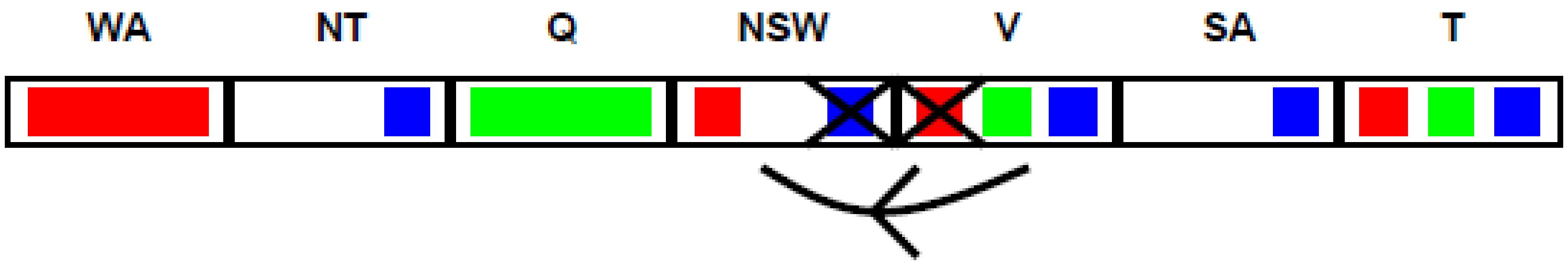
Arc Consistency



X → Y is consistent iff

for every value x of X there is some allowed y for Y



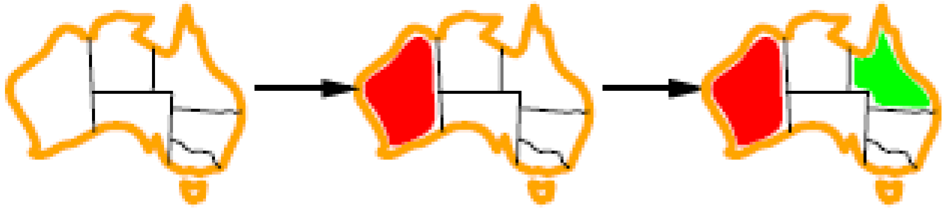


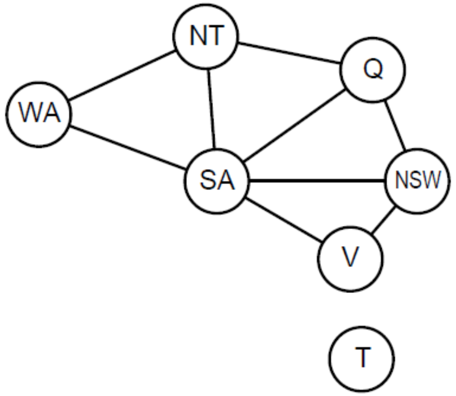
If X loses a value, neighbors of X need to be rechecked

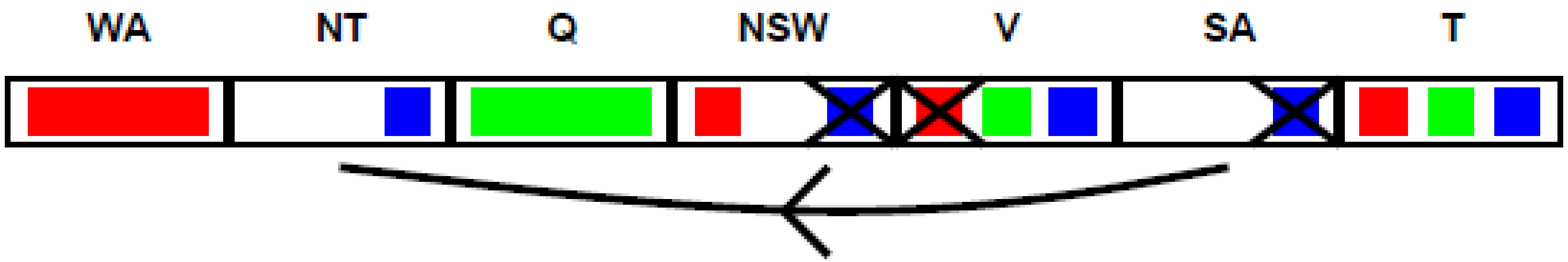
Arc Consistency

X → Y is consistent iff

for every value x of X there is some allowed y for 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

```
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 REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
            for each Xk in NEIGHBORS[Xi] do
                add (Xk, Xi) to queue
```

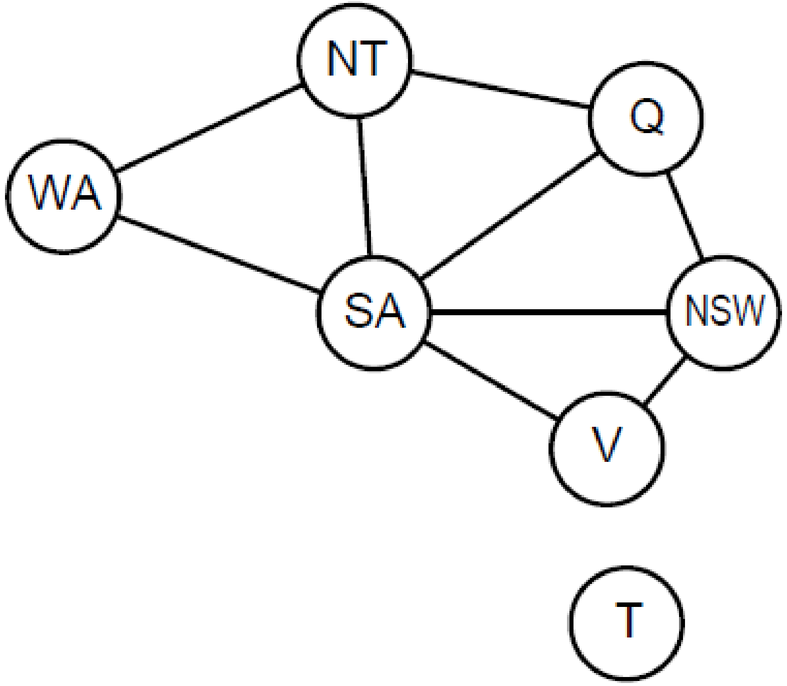
```
function REMOVE-INCONSISTENT-VALUES( Xi, Xj) returns true iff succeeds
    removed ← false
    for each x in DOMAIN[Xi] do
        if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
            then delete x from DOMAIN[Xi];  removed ← true
    return removed
```

Problem Structure

- The **structure of the problem** can be used to find solutions quickly.

- Coloring Tasmania and coloring the mainland are **independent subproblems**

- ***Independence*** can be ascertained simply by finding **connected components** of the constraint graph.
- Each component corresponds to a subproblem CSP_i.
- If assignment S_i is a solution of CSP_i, then $\bigcup_i S_i$ is a solution of $\bigcup_i CSP_i$.



Problem Structure

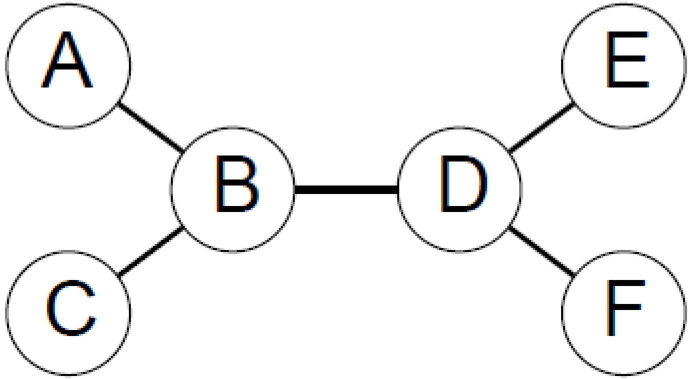
- ***Why independent subproblems are important?***

- Suppose each CSP_i has c variables from the total of n variables, where c is a constant.
- Then there are n/c subproblems, each of which takes at most d^c work to solve, where d is the size of the domain.
- Hence, the total work is $O(d^c\, n/c)$, which is linear in n.
- Without the decomposition, the total work is $O(d^n)$, which is exponential in n.

E.g., n=80, d=2, c=20

- 2^{80} = 4 billion years at 10 million nodes/sec
- $4 * 2^{20}$ = 0.4 seconds at 10 million nodes/sec

Tree-structured CSPs

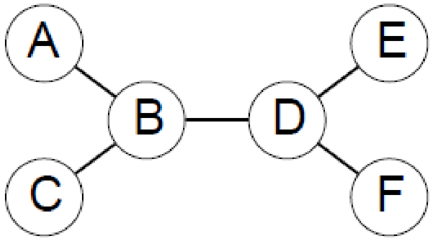


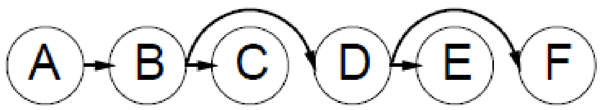
- A ***constraint graph is a tree*** when any two variables are connected by only one path (no loops).
- **Theorem**: If the constraint graph has no loops, the CSP can be solved in $O(n\ d^2)$ time (linear in n)
- Compare to general CSPs, where worst-case time is $O(d^n)$

Algorithm for tree-structured CSPs

function TREE-CSP-SOLVER(csp) **returns** a solution, or failure
  inputs: csp, a CSP with components X, D, C

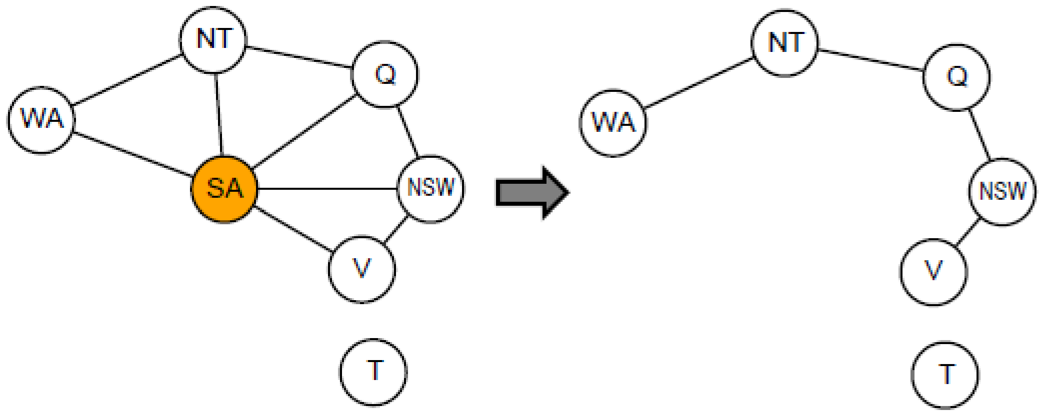
  $n \leftarrow$ number of variables in X
  $assignment \leftarrow$ an empty assignment
  $root \leftarrow$ any variable in X
  $X \leftarrow$ TOPOLOGICALSORT(X, $root$)
  for j = n **down to** 2 **do**
    MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
    if it cannot be made consistent **then return** $failure$
  for i = 1 **to** n **do**
    $assignment[X_i] \leftarrow$ any consistent value from D_i
    if there is no consistent value **then return** $failure$
  return $assignment$





Nearly tree-structured CSPs

- We have an efficient algorithm for trees, we can consider whether more general constraint graphs can be reduced to trees somehow.



- Delete South Australia, the graph would become a tree.
  - Instantiate SA to a value, and
  - From domains of other variables, delete any values that are inconsistent with the value chosen for SA.
  - Any solution for CSP after SA and its constraints are removed will be consistent with value chosen for SA.
  - The value chosen for SA could be the wrong one, so we would need to try each possible value.

Nearly tree-structured CSPs

General Algorithm:

1. Choose a subset S of the CSP's variables such that the constraint graph becomes a tree after removal of S.   S is called a **cycle cutset**.
2. For each possible assignment to the variables in S that satisfies all constraints on S,

   (a) remove from the domains of the remaining variables any values that are inconsistent with the assignment for S, and

   (b) If the remaining CSP has a solution, return it together with the assignment for S.

Cutset size c ➔ runtime $O(d^c\ (.n - c)\ d^2)$, very fast for small c

Local Search For CSPs

- Local search algorithms can be effective in solving many CSPs.
- **complete-state formulation** is used: the initial state assigns a value to every variable, and the search changes the value of one variable at a time.
- In choosing a new value for a variable, the most obvious heuristic is to select the valuethat results in the **minimum number of conflicts** with other variables—the **min-conflicts heuristic**.

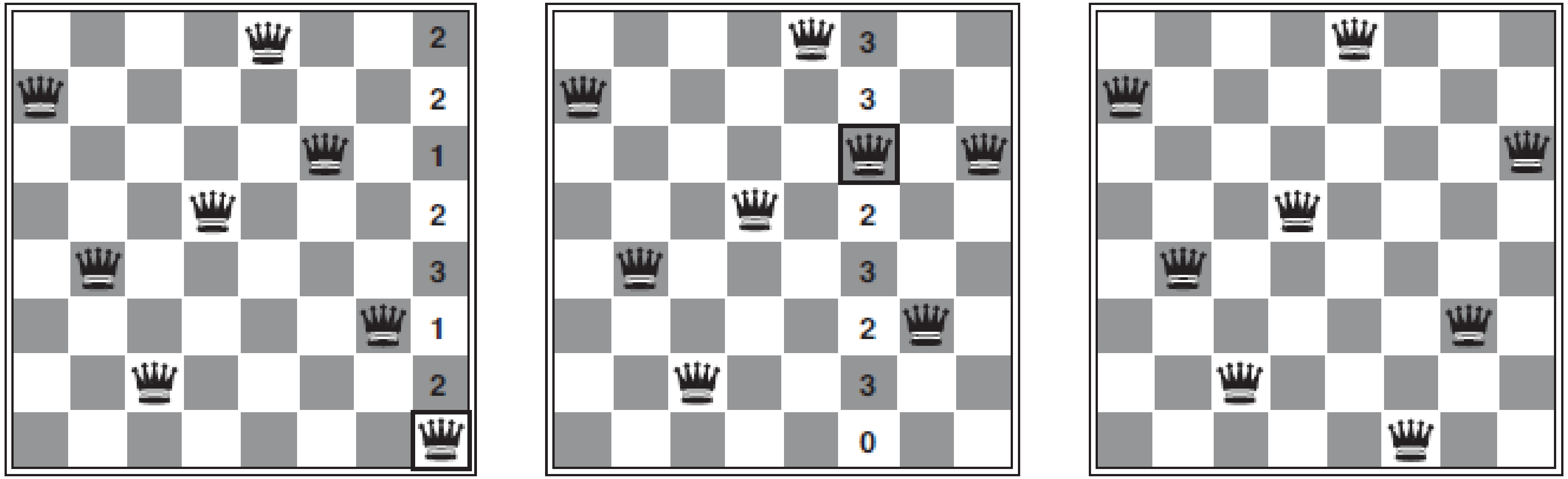
Min-conflicts

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

- Min-conflicts is surprisingly effective for many CSPs.
- On the n-queens problem, the run time of min-conflicts is roughly *independent of problem size*.
- It solves even the *million*-queens problem in an average of 50 steps

Min-conflicts on 8-queens problem



- A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column.
- The number of conflicts (***number of attacking queens***) is shown in each square.
- The algorithm moves the queen to the min-conflicts square, breaking ties randomly.

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
- The CSP representation allows analysis of problem structure
- Tree-structured CSPs can be solved in linear time
- Iterative min-conflicts is usually effective in practice