Overview

I Artificial Intelligence

II Problem Solving

3. Solving Problems by Searching
4. Search in Compex Environments
5. Adversarial Search and Games
6. **Constraint Satisfaction Problems**

III Knowledge, Reasoning, Planning

IV Uncertain Knowledge and Reasoning

V Machine Learning

VI Communicating, Perceiving, and Acting

VII Conclusions

Constraint Satisfaction Problems (CSPs)

- **Problem:** Factored representation of each state (i.e. set of variables) with constraints among the variables
- **Solution:** Assignment of a value to each variable with constraints not violated

Survey:

- Defining CSPs
- Constraint Propagation: Inference in CSPs
- Backtracking Search for CSPs
- Local Search for CSPs
- The Structure of Problems

Example CSP: SUDOKU

Problem formulation:

- 81 variables with domain {1..9} for each
- Some variables are preset to a particular value (e.g. A3 = 3)
- 27 Alldiff-Constraints (A1 to A9) … (I1 to I9), (A1 to I1) … (A9 to I9), (A1 to C3) … (G7 to I9)
- From the Alldiff-Constraints, many binary constraints like A1 ≠ A2 can be inferred.

Problem

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Solution

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

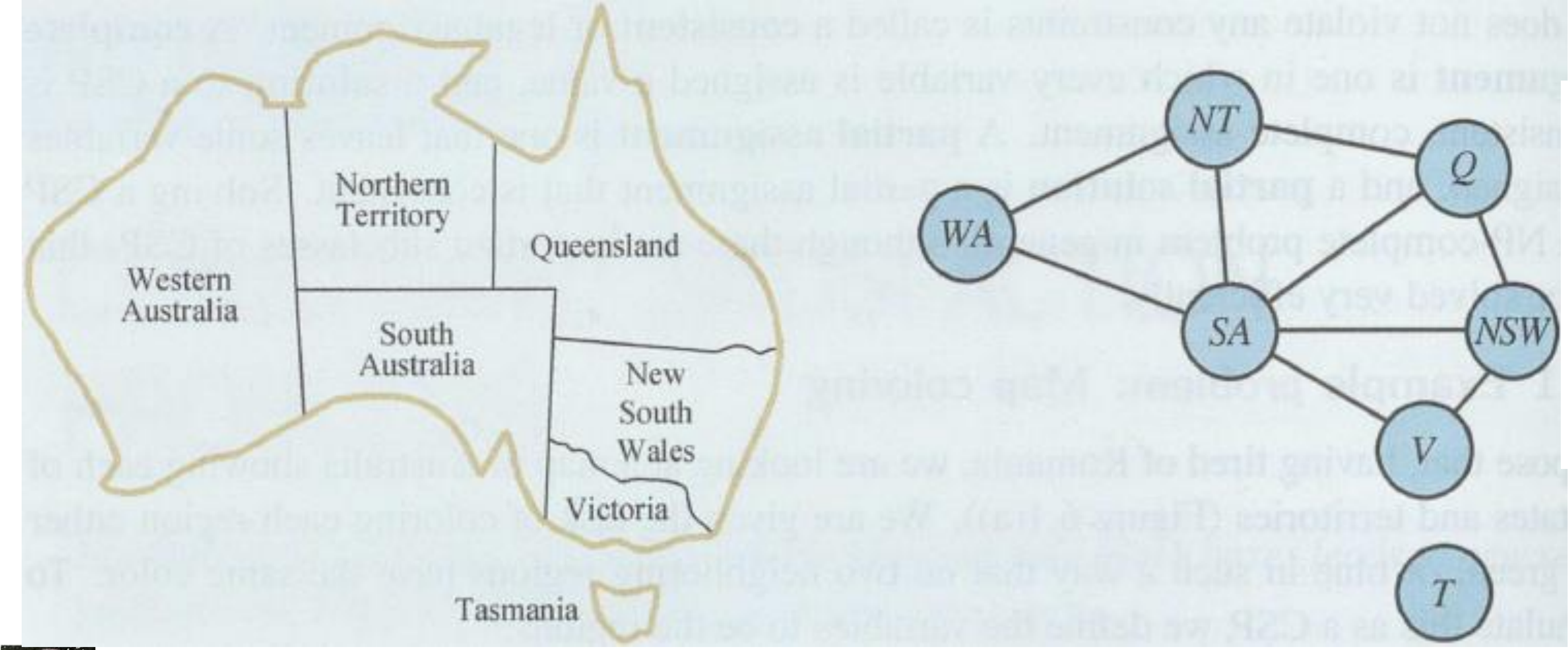
Defining CSPs

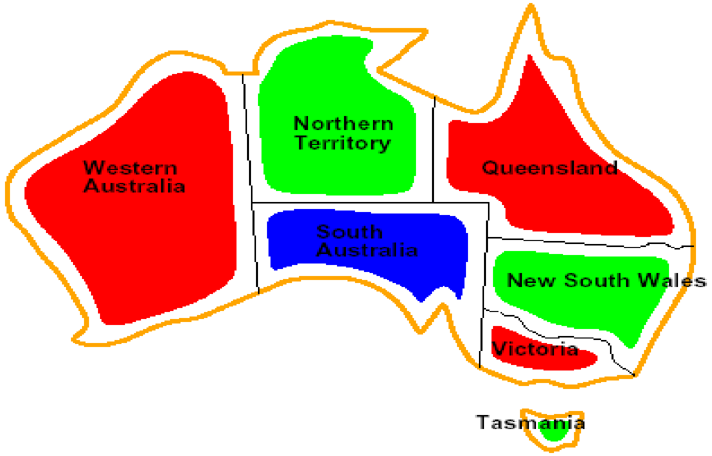
- CSP defined by variables with domains and constraints:
  - X is a set variables {X_1, … X_n)
  - D is a set of domains {D_1, … D_n}, one for each variable
  - C ist a set of constraints specifying allowable combinations of variables
    - e.g. (U= R*I) or (traffic_light_A ≠ traffic_light_B)
- State is defined by assignment of values to variables
  - Consistent assignment does not violate constraints
  - Solution is a consistent and complete assignment

Example Problem 1: Map Coloring

Color regions of Australia with three colors so that no two neighboring regions have same color.

- **Variables** (Regions): {WA, NT, Q, NSW, V, SA, T}
- **Domains** (Colors): {red, green, blue} for all variables
- **Constraints**: {SA ≠ WA, SA ≠ NT, SA ≠ Q, SA ≠ NSW; SA ≠ V, WA ≠ NT, NT ≠ Q, Q ≠ NSW, NSW ≠ V}





Possible solution

Example Problem 2: Job Shop Scheduling

- Factories manufacture orders subject to various constraints
- An order consists of tasks with a duration and with sequence constraints
- Tasks are manufactured on a machine and may need ressources (temporarily or permanent)
  - Machines can do only one task at a time
- Solution is a plan with tasks assigned to machines with time intervals
- Constraints are mentioned above (task sequence and machine constraints, maybe also ressources constraints)

Example Problem 2: Crypt-Arithmetic Puzzle

Example:

```
   TWO
+  TWO
= FOUR
```

Constraints:

- Alldiff (F, T, U, W, R, O),
- Domains of variables: F, T, U, W, R, O in {0, 1, …, 9}
- no leading zeros: T, F ≠ 0
- O + O = R + 10*X1
- X1 + W + W = U + 10 * X2
- X2 + T + T = O + 10 * X3
- X3 = F
- Domains of variables for Carryover: X1, X2, X3: in {0, 1}

Constraint problems in schools

- **Assignment of students for elective courses with capacity constraints**
  - Input: Students with first, second and third course selection; courses with capacities
  - Output: Optimal assignment of students to courses
  - Constraints: No overbooked courses; students should get (first) selected course
- **Assignment of students to four sport courses in halfyears in Q11 and Q12**
  - Input: Students with four course preferences
  - Output: Distribution of courses to halfyears; assignment of students to courses
  - Constraints: Each student should have his/her four courses in different halfyears
- **Colloquium planning in Abitur examination**
  - Input: Students with two Colloquia, two examiners for each Colloquium
  - Output: Schedule for each Colloquium within two weeks
  - Constraints: Distance of both Colloquia for a student more than x days, compact schedule for examiners

Variations on CSP formalism

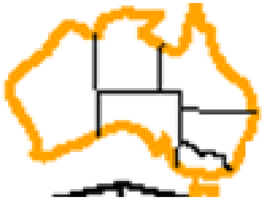
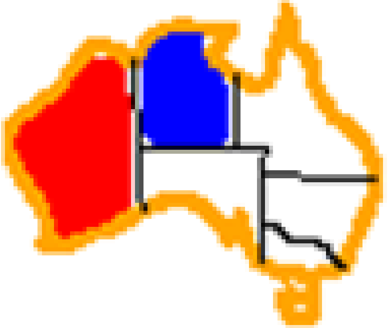
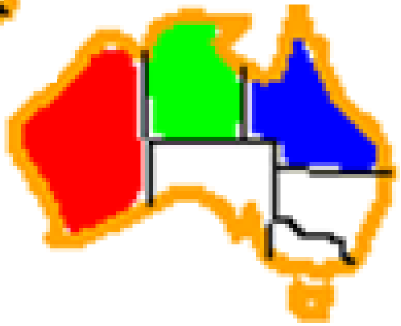
- **Types of variables:**
  - Discrete, finite domain, e.g traffic light {red, yellow, green}
  - Discrete, infinite domain, e.g. integers or strings
  - Continous domain, e.g. real number or times
    - Can be solved with linear programming
- **Types of constraints:**
  - Unary (concerning only one variable, e.g. domain restrictions)
  - Binary (concering two variable, e.g. A ≠ B)
  - Higher order (concerning more than two variables, e.g. U = R * I
  - Global (e.g. „alldiff“ constraints in SUDOKU: alldiff (A1, B1, C1, D1, E1, F1, G1, H1, I1)
- **Preference Constraints** (soft constraints)
  - Implies constrained optimization problem

Constraint Propagation: Inference in CSPs

- CSPs can be solved by search and/or inference (constraint propagation)

- Core idea: Local consistency, to reduce number of legal values for a variable
  - **Node consistency**: Unary constraints (1 variable)
  - **Arc consistency**: Binary constraints (2 variables)
  - **Path consistency**: Looks at 3 variables connected by constraints
  - **k-consistency**: Node, arc and path consistency are also calls {1, 2, 3}-consistency. Higher order consistencies are rarely used.

Arc Consistency

- Definition: for each variable V there exist at least one consistent value for all variables V‘ connected to V by a (binary) constraint
- Implementation: Check all arcs (binary constraints) and eliminate values from the domain of a variable inconsistent with arc-connected variables
  - Example1: $X \in \{0,1,2,3\}$, Y = Integer, Constraint: $Y = X^2$ → $Y \in \{0,1,4,9\}$,
  - Example2: Map-Coloring:
    - Situation 1:  No elimination possible
    - Situation 2 (WA=red, NT=blue):  SA ∈ {~~red~~, ~~blue~~, green); Q ∈ {red, ~~blue~~, green)
    - Situation 3 (WA=red, NT=green, Q=blue):  SA ∈ {~~red~~, ~~blue~~, ~~green~~);

Arc Consistency Algorithm AC-3

Idea:

- For each arc, check its variables for elimination (a binary constraint implies two arcs!)
- If successful, add all arcs of a domain-reduced variable to the arcs to be checked

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  queue ← a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← POP(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 ← 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 ← true
  return revised
```

Complexity of AC-3

- Each arc c must be checked
- Checking an arc with two variables, each of them having at most d values, requires at most d^2 comparisons
- In the worst case, for each arc, there will be only one value eliminated, i.e. the arc must be checked d times.
- Resulting complexity: $O(cd^2d) = O(cd^3)$
  - With n nodes, there are at most n^2 arcs c: $O(n^2d^3)$

Example SUDOKU

- 81 variables (of which 32 are set in the example problem)
- 27 all-diff constraints equivalent to 36*9 (rows) + 36*9 (columns) + 18*9 (squares) = 810 binary constraints
- Application of AC-3 algorithms solves the problem without search:
  - E6 = 4
  - I6 = 7
  - A6 =1
  - etc.

Problem

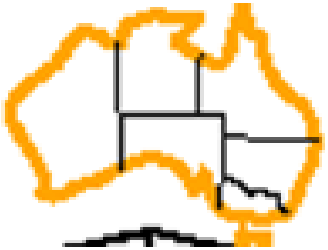
	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Solution

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

Path Consistency

- A two-variable set $\{X_i, X_j\}$ is path-consistent with respect to a third variable X_m, if for every assignment $\{X_i = a, X_j = b\}$ consistent with the constraint on $\{X_i, X_j\}$, there is an assignment X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_j, X_m\}$.
- Example: Map-Coloring with only two colors red and green
  - Arc-consistency would not detect the unsolvibility in the start situation
  - Path-consistency would detect it, because the pair (WA, NT) has the possible assignments {red, green} and {green, red}. They are both not consistent with a color for SA, yielding { }.



Global Constraints

- Valid for many variables, e.g.
  - **Alldiff:** all variables must have different values
  - **Atmost:** Resource-Constraints with upper bound for sum of variables (e.g. course-capacity)
- Often suited to **prove unsolvebility** of CSPs, e.g.:
  - Alldiff constraint with m variables and n possible values, where m > n
  - Atmost (10 P1, P2, P3, P4) with Pi $\in$ {3, 4, 5, 6}
- **Elimination of values** in the domain of variables, e.g.
  - Alldiff (A, B, C) and A, B, C $\in$ {red, blue, green); A = red $\rightarrow$ B, C $\in$ {blue, green);
  - Atmost (10 P_1, P_2, P_3, P_4) with $P_i \in \{2, 3, 4, 5, 6\} \rightarrow P_i \in \{2, 3, 4\}$
- Exploited by **special algorithms** or by **transformation in binary constraints**
- Useful for **bounds propagation**: e.g. two flights with capacity [0, 100] and 150 passengers for both flights results in new bounds [50, 100] for each flight.

CSP as search problem

- Not all constraint problems can be solved by inference, often search is necessary
- Naive solution: depth-limited search with all variables and domains
  - Action is assignment of a value to a variable
  - Branching factor with n variables and d values: n * d
    - Size of search tree: $n! * d^n$
    - Slightly better, if commutativity in CSPs is exploited: n^d (n variables with d values)

Resulting Backtracking Search Algorithm

- Backtracking search uses only one representation of a state, which is extended or - in case of failures – altered by **recursive depth-first search**
- Functions **Select-UnAssigned-Variables** and **Order-Domain-Values** implement general-purpose heuristics (see below)
- **Inference** can optionally implement **arc-, path-** or **k-consistency**

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

function BACKTRACK(csp, assignment) returns a solution or failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
   for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
      if value is consistent with assignment then
         add {var = value} to assignment
         inferences ← INFERENCE(csp, var, assignment)
         if inferences ≠ failure then
            add inferences to csp
            result ← BACKTRACK(csp, assignment)
            if result ≠ failure then return result
            remove inferences from csp
         remove {var = value} from assignment
   return failure
```

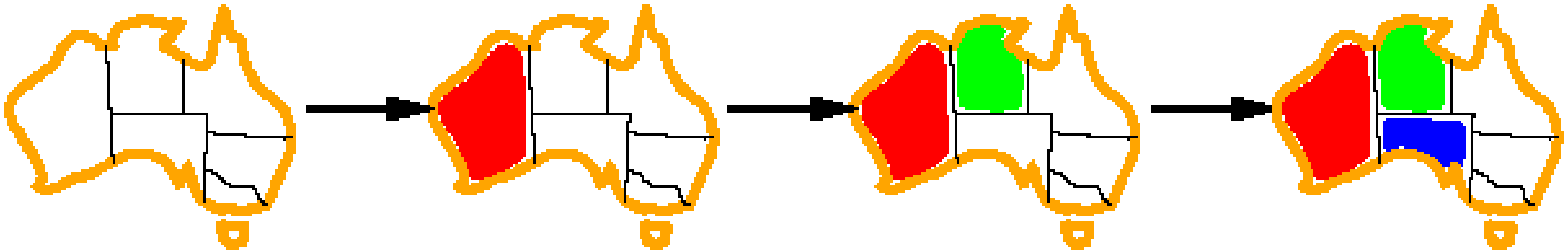
Heuristics and Inferences

- Which variable and which value of its domain should be selected first in search procedure?
  - Select-Unassigned-Variables (csp):
    - **Minimimum-Remaining-Value (MRV) heuristic**: Choose the variable having the least values
      - Reasong: To detect failure early.
    - **Degree-Heuristic**: Select variable being involed in the most constraints,
      - Reasong: To reduce the branching factor of future choices
  - **Order-Domain-Values**:
    - Least-Constraining-Value heuristic: Select value for a variable, that rules out the fewest choices for the neighboring variables
      - Reason: Increase chances for finding a legal assignment

- Question: Why is **variable selection** by MRV-heuristic **fail-first** and **value selection fail-last**?
  - Answer: Every variable has to be assigned eventually, but not every value. Choosing variables which are likely to fail first reduces amount of backtracking.

Example for MRV Heuristic

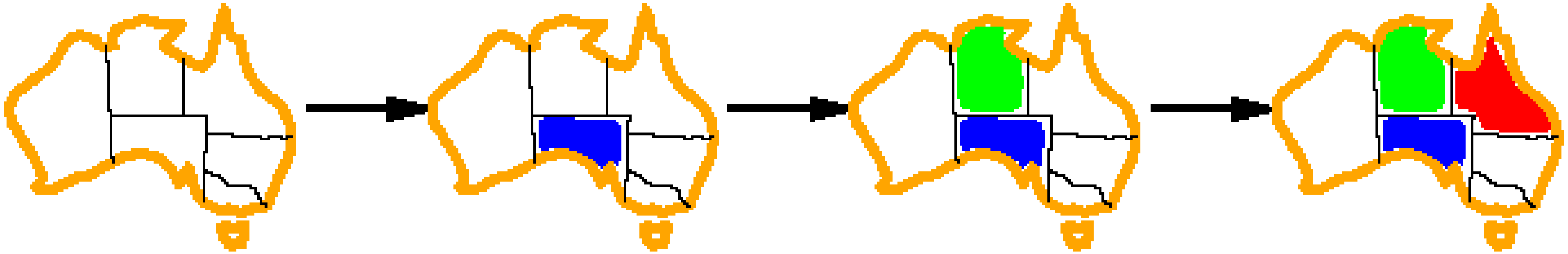
Choose always the variable with the least legal values in its domain.



- First step: no preference (all variables have three legal values), Let‘s choose WA = red
- Second step: NT and SA habe only two legal values (green and blue). Let‘s choose NT = green
- Third step: SA has only one legal value (blue) and is chosen.

Example for Degree Heuristic

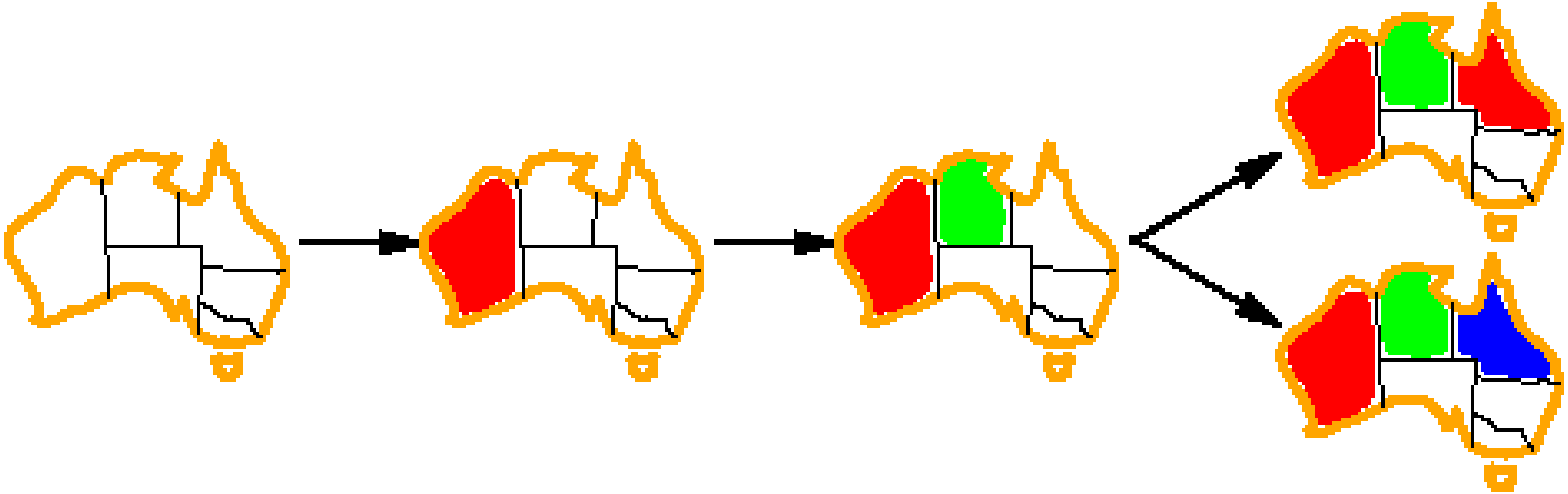
- If MRV has no preferences, the degree heuristic is applied
  - Choose the variable involved in the most constraints



- First step: Choose SA (involved in 5 constraints). Let‘s choose SA = blue
- Second step: NT, Q, NSW are involved in 2 constraints. Let‘s choose NT = green
- Third step: MRV-Heuristic would select WA and Q (both have one value); WA has one constraint, Q has zero constraings, Degree Heuristic would prefer Q. Choose Q = red

Example for Least Constraining Value Heuristic

- If a Variable is selected, the Least Constraining Value Heuristic is applied
  - choose for a variable a value, that avoid restricting other options as far as possible



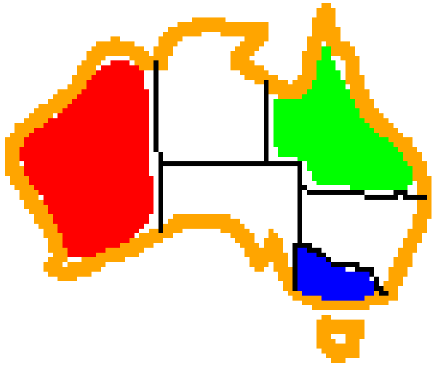
- For first and second step no preference
- Third step: Choose Q = red because
  - Q = red → SA has 1 possible value
  - Q = blue → SA has 0 possible value

Interleaving Search and Inference

1. **Forward Checking:** Eliminate illegal values from the domains of all variables based on set variables and their constraints immediately
2. **Maintaining Arc Consistency (MAC):** Check arc consistency with AC-3 for those arcs being connected with a variable, whose value has changed.

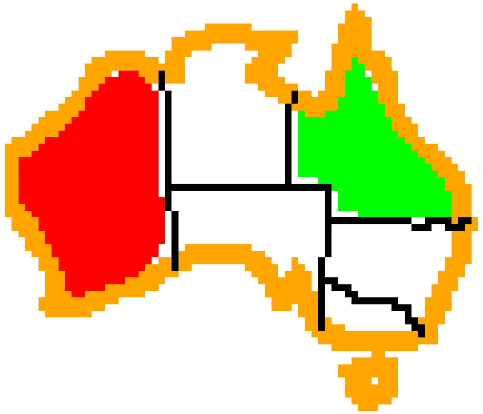
Forward Checking

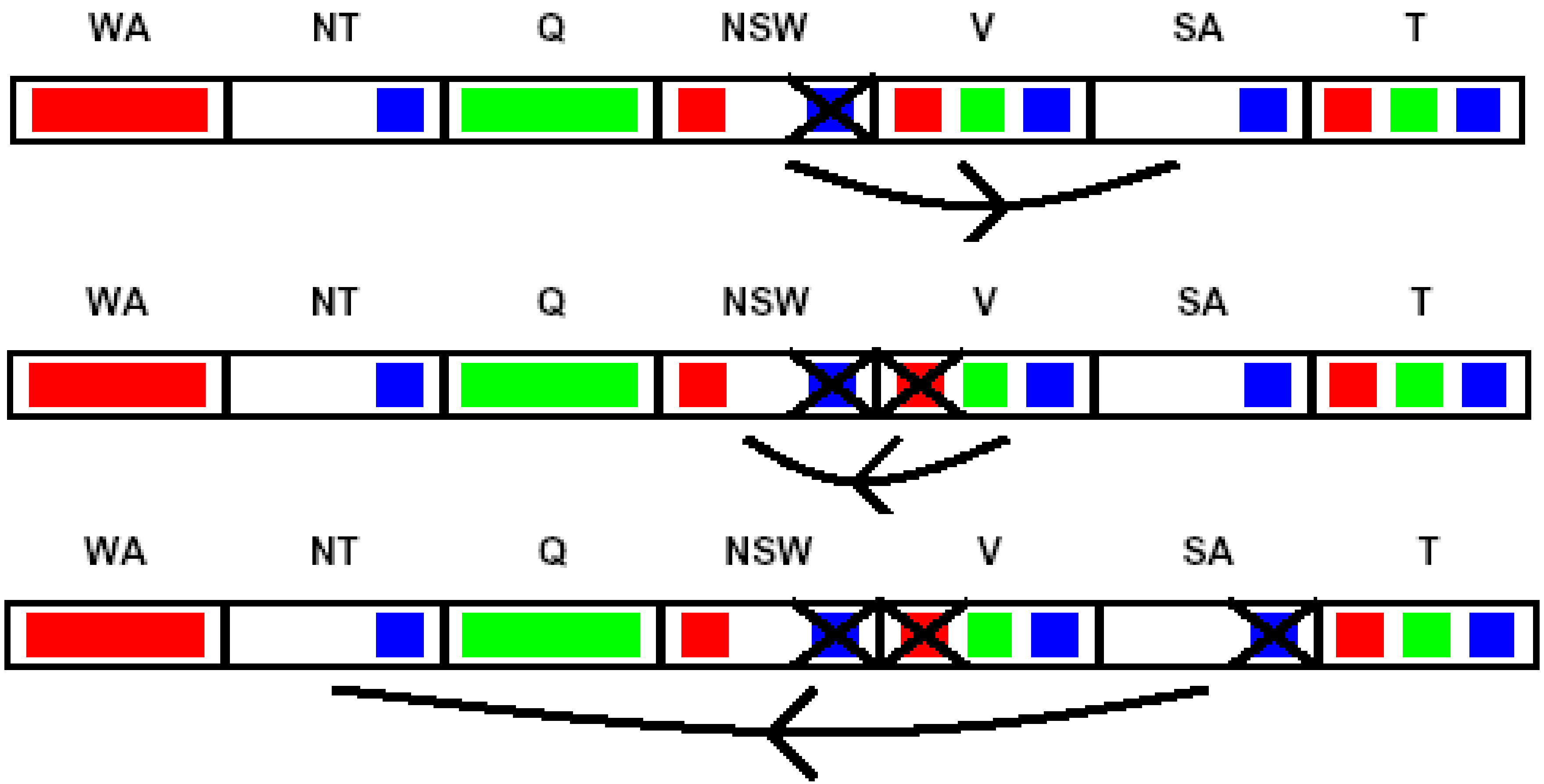
- Idea: Maintain for each not assigned variable the currently legal values
  - If a variable has only one value left, choose that variable next
  - If a variable has no values left, stop search and backtrack
- Example:
  - 1. line: Start state
  - 2. line: After assignment of WA = red
  - 3. line: After assignment Q = green
  - 4. line: After assignment V = blue → SA = Ø (STOP)



Maintaining Arc Consistency (MAC)

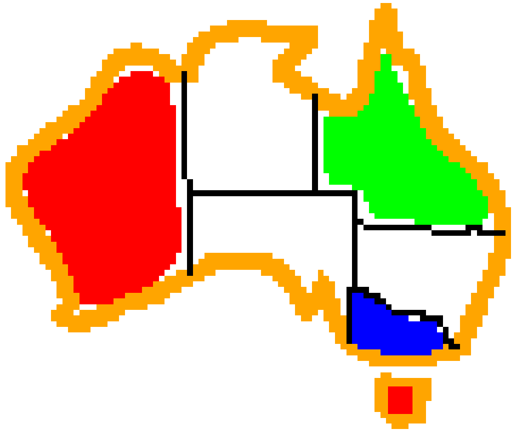
- Forward checking detects many inconsistencies, but MAC even more
- MAC: If a variable has changed, apply AC-3 for that (small) subset of constraints, which connect the changed variable to unassigned variables
- Example:





Intelligent Backtracking: Looking Backward

- Idea:
  - If search fails, chronical backtracking would try different value for most recent decision.
  - A better approach would be to backtrack to a variable, that participated in the fail
    - Computation of conflict set for fail, i.e. variable with empty domain
    - Backjump to a variable in conflict set and change its value



- Example:
  - Assume: {Q = red, NSW = green, V = blue, T = red}
  - If we try next to assign a value to SA, the assignment fails
  - Backtracing to T (Tasmania) and trying a new color doesn‘t help
  - Instead, the conflict set for SA is computed: {Q = red, NSW = green, V = blue}
  - Backjumping choose to go back to the most recent assignment of the conflict set, i.e. V

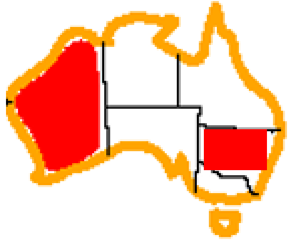
Discussion of Intelligent Backtracking

- Forward Checking (and MAC) detect empty domains of variables and induces an immediate backtrack
  - ➢ When Forward Checking (and MAC) is used, intelligent backtracking is redundant!

- Refinement necessary: Conflict-directed backjumping

Conflict-directed Backjumping

- Idea:
  - Extend the notion of a conflict set from the variables immediately involved to the variables indirectly involved
    - Immediately involved variables: those variables V‘ causing an empty domain for a variable V (as before)
    - Indirectly involved variable: If a variable V‘ has been assigned a value because of constraints from other variables V‘‘, these variables V‘‘ should belong to the conflict set of V too (replacing V‘)
- Example
  - {WA = red, NSW = red}
  - Next assignments: T, NT, Q, V, SA (SA empty)
  - Conflict set of SA: e.g. {WA, NT, Q}; Backtrack to most recent assignments. i.e.
  - Backtrack to Q, which has conflict set {NT, NSW} → new conflict set: {WA, NT, NSW}
  - Backtrack to NT, which has conflict set {WA} → new conflict set: {WA, NSW}
  - Backtrack to NSW, which solves the problem

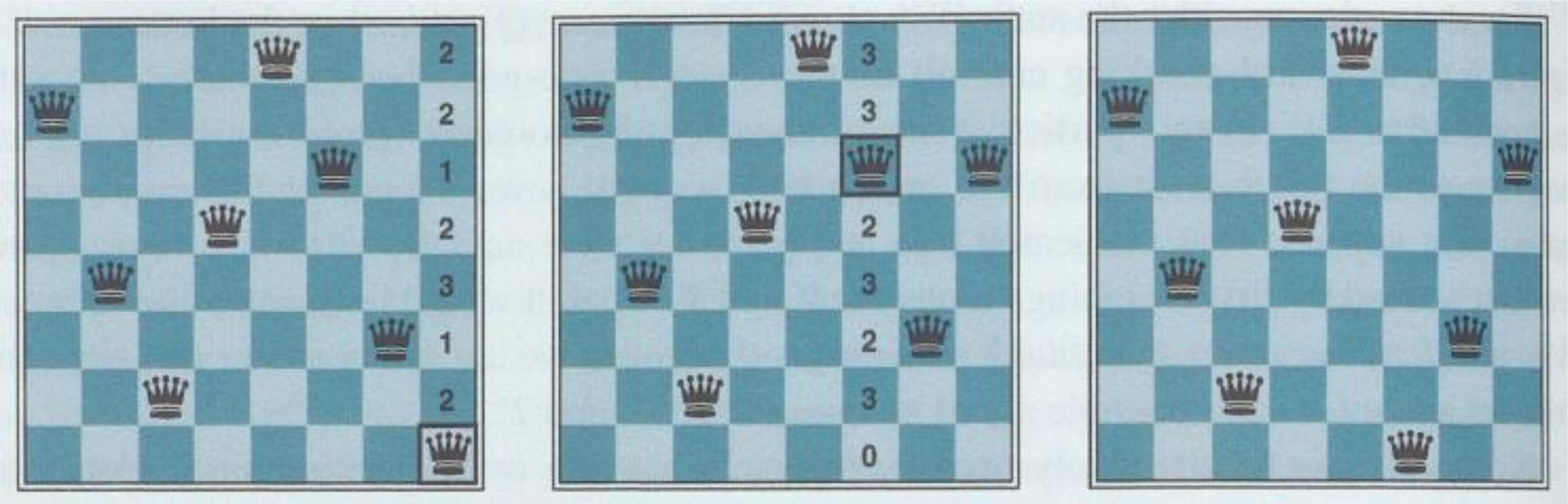


Constraint Learning

- When reaching a contradiction (empty set), it might be useful to store the set of variables causing the contradiction, in order to avoid in future search this constellation.
  - **No-Good**: Set of variables with values known to be unsolvable
- No-Goods can be effectively used by forward checking of by backjumping
- Important technique to increase efficiency in complex constraint problems

Local Search for CSPs

- Local Search with complete-state formulation (i.e. every variable has an value, but there may be constraint violations) is be very efficient for solving many CSPs
  - Hill-Climbing with plateau search and Min-Conflicts as Heuristic
    - Choose in each step a variable and change its value to minimize the total number of conflicts (i.e. constraint violations)
    - Initialization randomly or with a greedy strategy
- Example:



Min-Conflicts Algorithm

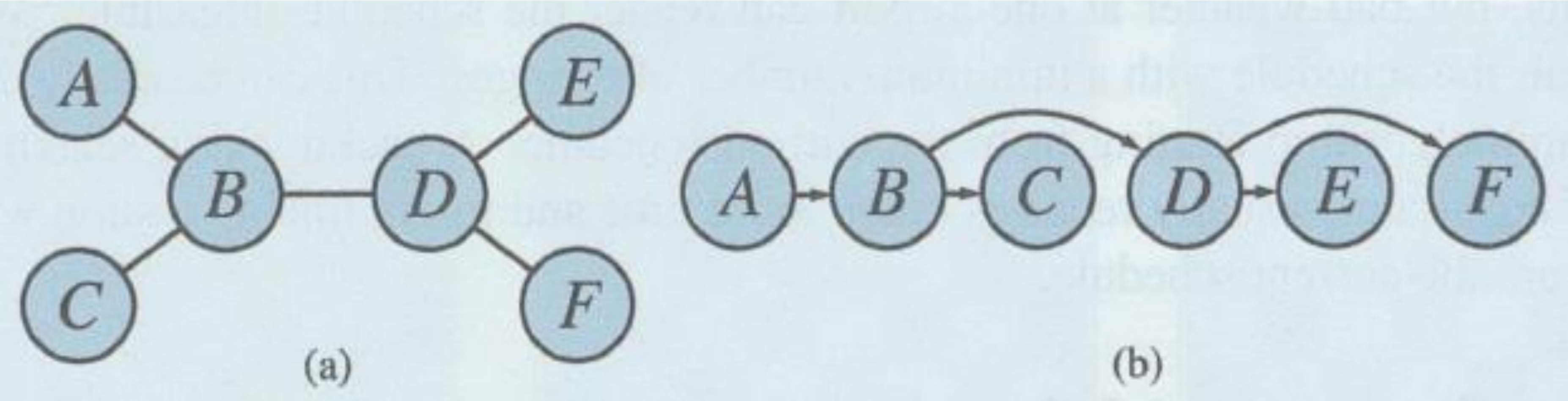

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up

  current ← 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(csp, var, v, current)
     set var = value in current
  return failure
```

- Improvements:
  - **Plateau search** with **tabu search**: Store recently visited states to avoid returning to them
  - **Constraint weighting**: In each step, increment the weight of violated constraints to focus search on the difficult constraints

Structure of Problems

- Standard technique for reducing complexity of problems: **Decomposition in subproblems**
  - Ideal but rare: **Decomposition in independant subproblems**
    - Example in Map Coloring problem: Tasmania and Rest of Australia
    - Detection of independance: find connected components
    - Advantage in complexity: Cuts down the exponent of exhaustive search (d^n)
- Also easy to solve: Tree CSPs
  - Def.: Constraint graph is a tree, when any two variables are connected by only one path
  - Can be solved in linear time in the number of variables
  - Solution idea: Topological sort, so that the parents are alway before the children
    - Example:



Tree-CSP-Solver Algorithm

1. Sort variables topologically
2. Apply in reverse order arc-consistency algorithms between a node and its parent
3. Choose an Assignment for every node in normal order compatible to its parent node

4. Complexity: $O(nd^2)$
   - n = number of variables
   - d = number of values

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

   n ← number of variables in X
   assignment ← an empty assignment
   root ← any variable in X
   X ← 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] ← any consistent value from D_i
      if there is no consistent value then return failure
   return assignment
```

Transformation of a CSP in a Tree CSP

- If a CSP is not a tree, it can be transformed in a tree!
- Two approaches:
  - Removing nodes: **Cutset Conditioning**
  - Uniting nodes: **Tree Decomposition**
- Efficient, if not too many nodes must be removed or united, because for the removed or united nodes, all value combinations must be tested
- Total effort = Effort for restructering + effort for solving the restructured problem

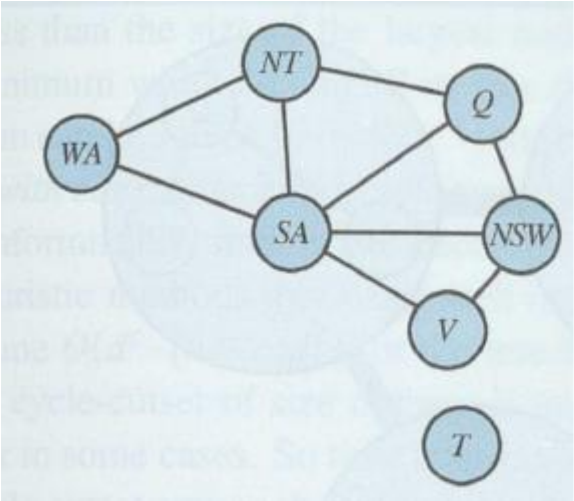
Cutset Conditioning

1. Choose a subset S of the variables of the CSP („**cycle cutset**“) such that the constraint graph becomes a tree after removal of S
2. For each possible assignment of the variables in S (fulfilling their constraints):
   a. Remove from the domains of all remaining variables all values inconsistent with the assignment
   b. Solve the problem with the Tree-CSP-Solver and combine the solution (if existing) with assignment for S

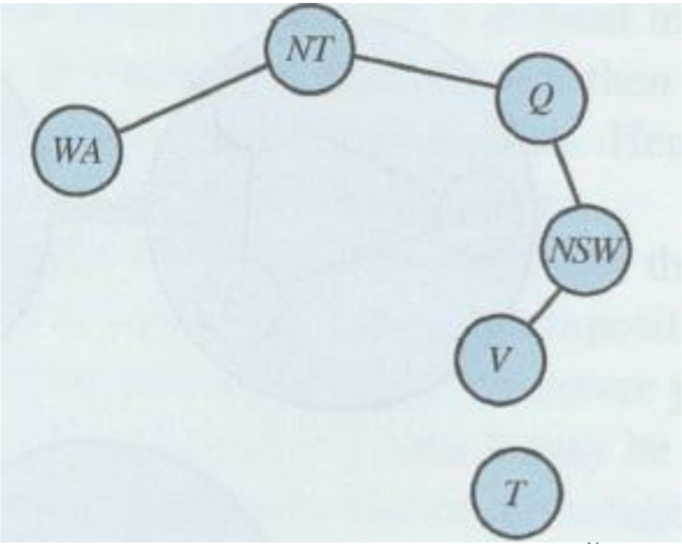
➢ Complexity: $O(d^c * (n-c) * d^2)$ mit c = |cycle cutset|

- Example:

Original problem:



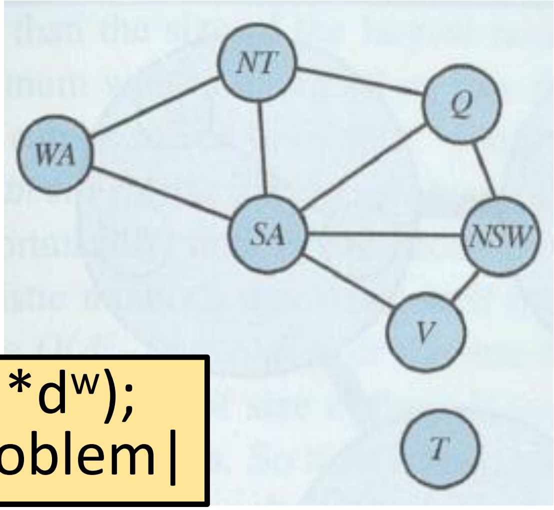
Tree CSP problem after removal of SA:



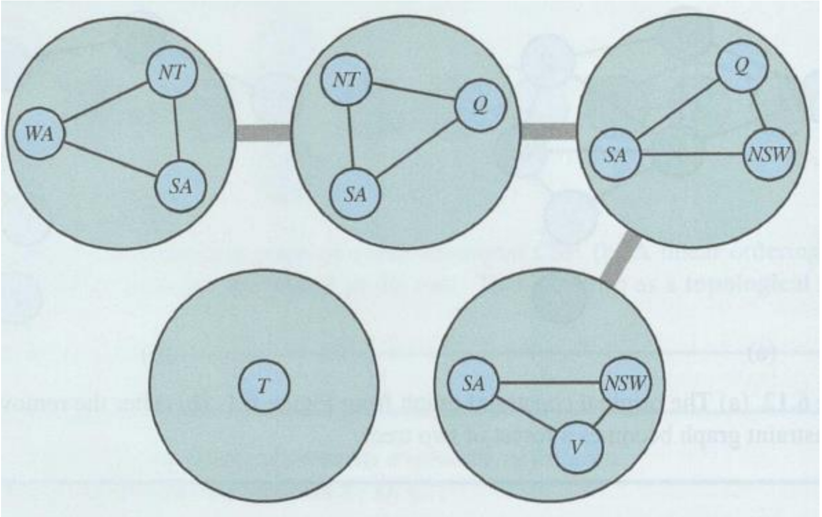
Tree Decomposition

- Transform the original graph in a tree, where each node of the tree consists of a set of variables with three requirements
  - Every variable of the original problem must appear in at least one tree node
  - If two variables are connected by a constraint in the original problem, they must appear (with the constraint) in at least one tree node
  - If a variable appears in two nodes in the tree, it must appear in every node along the path connecting those nodes
- Solve the first subproblem locally, e.g. WA-NT-SA = {red, green, blue}, then move to the next subproblem NT-Q-SA, where SA=blue and NT=red are taken over from first node, so that a solution is {green, red, blue}, until all mega-nodes of the tree have values
- Example:

  Original problem:



Tree decompensation satisfying above requirements:



Complexity: $O(n*d^w)$;
w = max |subproblem|

Value Symmetry

- Observation: The map coloring problem has many symmetric solutions, because it doesn‘t matter, what color e.g. the first node has
- Goal: Avoid such **value symmetries** (with 3 colors 3! = 6 value symmetries) thus reducing the search space
- Solution: Introduction of a **symmetry-breaking constraint**
  - In the example, NT, SA and WA must have different colors.
  - Therefore, we impose an arbitrary ordering constraint NT < SA < WA requiring the three values in alphabetica order (blue > green > red)
  - Prevents examination of color permutations (only solution: NT=blue, SA=green, WA=red)
  - In general, difficult, but rewarding task