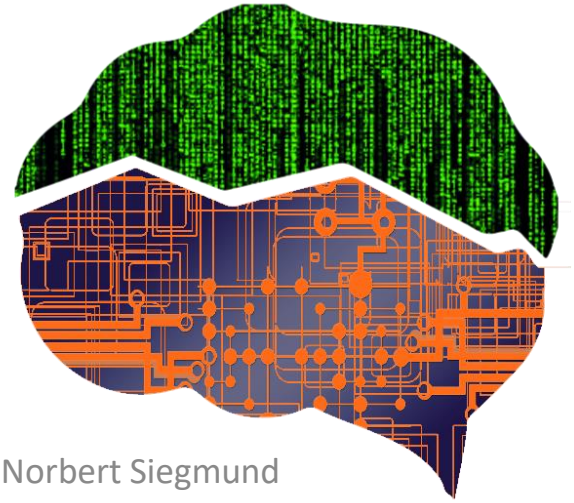
Machine Learning for Software Engineering

Constraints Satisfaction Problem



Prof. Dr.-Ing. Norbert Siegmund
Intelligent Software Systems

Bauhaus-Universität Weimar

Recap I

- Combinatorial optimization, what is special?
  - Fixed set of components with values/costs assigned
  - Optimization goal is to have an optimal combination under given constraints

- GRASP
  - Single-state optimization method
  - Construct a valid solution from components with a higher value / lower cost
  - Do hill climbing on them

Recap II

- Ant Colony Optimization (Ant System):
  - Maintain historical quality of the components (pheromones)
  - Construct new solutions based on pheromones and cost/value
  - Proportionate-pheromone selection
  - Evaporation by a fixed amount or learning rate

- Ant Colony System:
  - Elitism for pheromone updates (update pheromones of for those components that participate in the best solutions)
  - Considers linkage between components

- Guided Local Search:
  - Like Tabu Search, where components that appear often in the best individuals get penalized to do more exploration

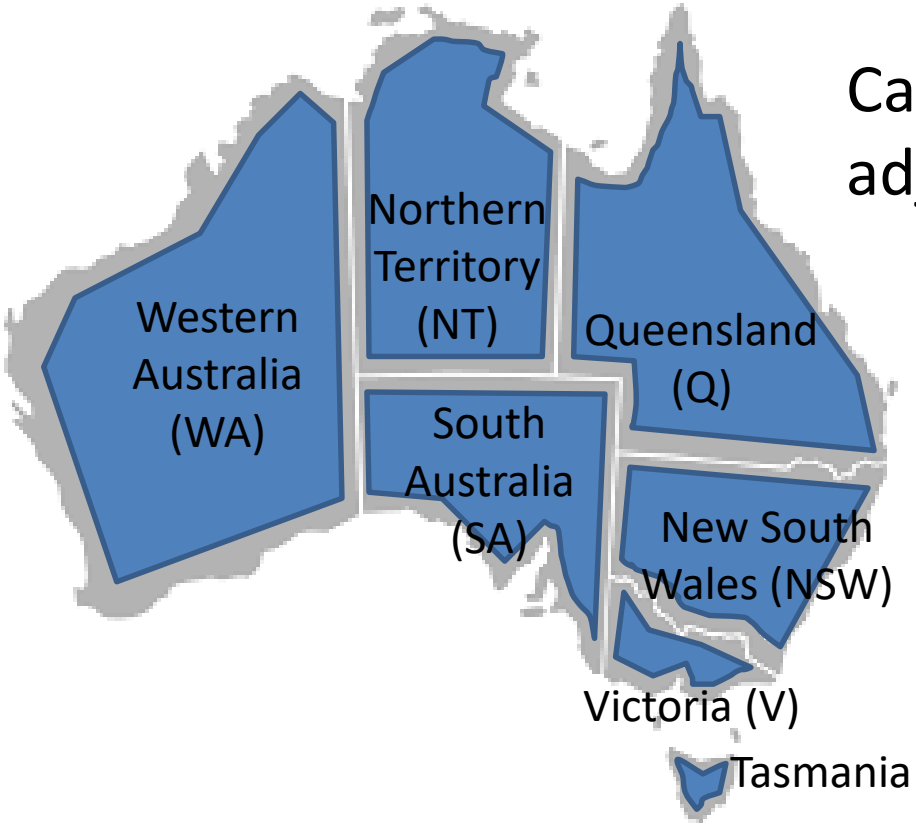
Constraint Satisfaction Problem (CSP)

Constraint Satisfaction Problems

- Search problem: Find a valid solution / model / state
  - State is black box (data structure is unknown or does not matter)
  - Need successor function: Build iteratively a solution until it is complete
- A solution is defined by a set of variables X with values from a domain D
- To determine whether a solution is valid, it makes a goal test
  - Verifies the current solution whether it satisfies a set of given constraints that specify allowable combinations of values for subsets of variables
  - Solution is a satisfying assignments of all variables

Example: Map Coloring

A solution={WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}



Can I color each state such that no adjacent state has the same color?

Snippet of code

Implicit constraints: $WA \neq NT$

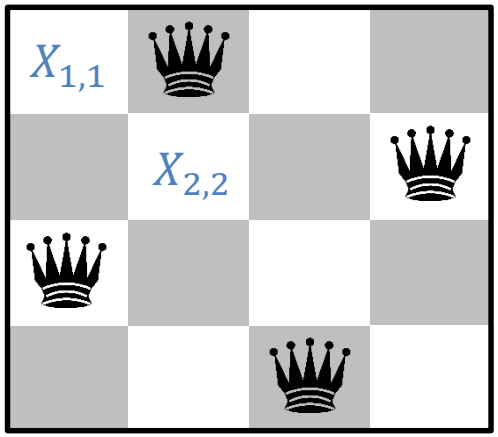
Explicit: $(WA, NT) \in \{(red, green), (red, blue), \dots\}$

Enumerates all valid combinations

- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: D = {red, green, blue}
- Constraints: adjacent regions must have different colors

Example: N-Queens

- $N * N$ chess board
- Goal: place n queens such that they don't fight
- Formulation 1:
  - Variables: $X_{i,j}$
  - Domains: $\{0,1\}$
  - Constraints:



$(X_{1,1}, X_{2,2}) \in \{(0,0), (1,0), (0,1)\}$

Explicit

$\forall i,j,k\ (X_{i,j}, X_{i,k}) \in \{(0,0), (0,1), (1,0)\}$ ← No two queens in a row

$\forall i,j,k\ (X_{i,j}, X_{k,j}) \in \{(0,0), (0,1), (1,0)\}$ ← No two queens in a column

$\forall i,j,k\ (X_{i,j}, X_{i+k,j+k}) \in \{(0,0), (0,1), (1,0)\}$ ← No two queens in a diagonal

$\forall i,j,k\ (X_{i,j}, X_{i+k,j-k}) \in \{(0,0), (0,1), (1,0)\}$ ← No two queens in a diagonal

Implicit

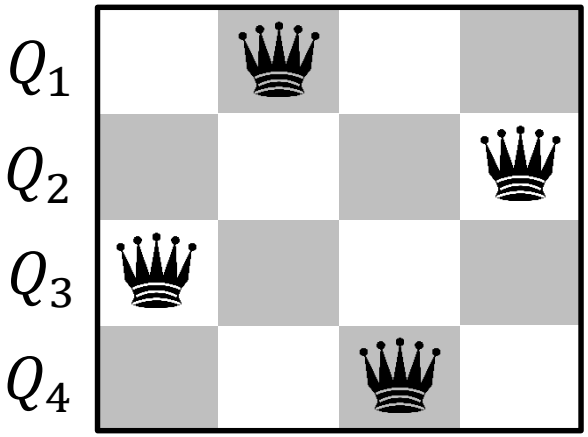
$$\sum_{i,j} X_{i,j} = N$$

Example: N-Queens

- Formulation 2:
  - Variables: Q_k
  - Domains: $\{1,2,3,\dots,N\}$
  - Constraints:

Implicit → $\forall i,j \quad non\text{-}threatening(Q_i, Q_j)$

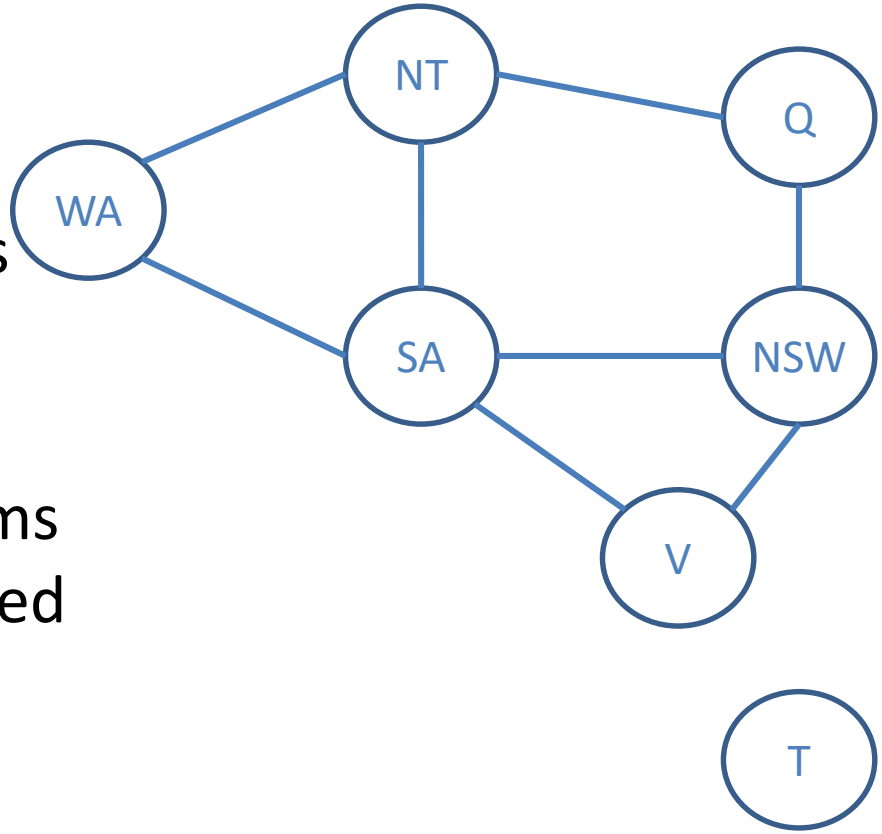
Explicit → $(Q_1, Q_2) \in \{(1,3),(1,4),\dots\}$



- Which formalization works better?
  - Probably 2, because more domain knowledge is encoded
  - Usually, not known beforehand

Visualizing Constraints: Constraint Graphs

- Binary CSP: each constraint relates (at most) two variables
- Binary constraint graph: nodes are variables, arcs show constraints
- General-purpose CSP algorithms use the graph structure to speed up search (e.g., Tasmania is an independent subproblem)



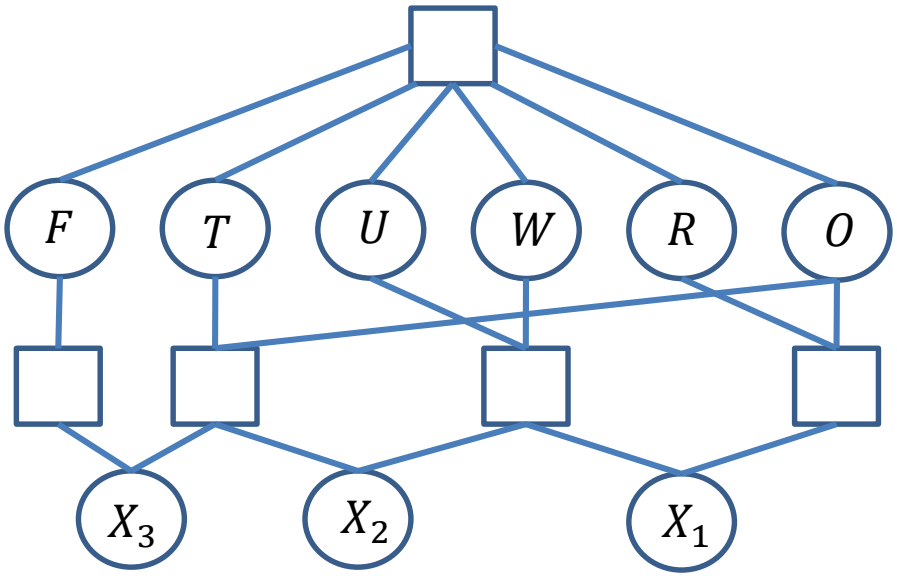
Constraints

- Hard constraints: limitations on the domain of variables
  - Unary constraint:
    - Restriction on a single variable
    - For example, x > 4 or SA != blue
  - Binary constraint:
    - Between two variables (e.g., SA != WA)
  - N-ary constraint:
    - Restrictions on the domain of n variables
    - Can always be transferred to binary constraints
- Soft constraints:
  - Red is better than blue
  - Cost of each variable assignment
  - Here is where it becomes an optimization problem!

Example: Cryptarithmetic

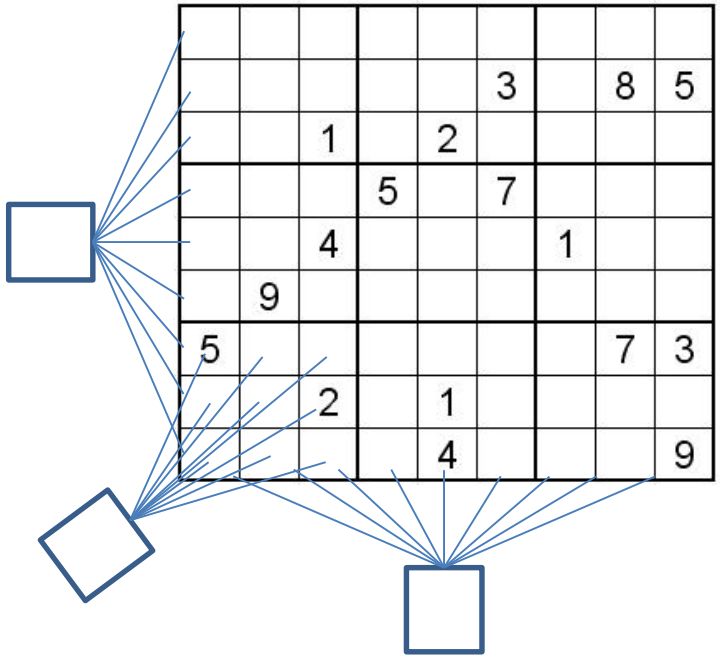
- Variables:
  $F\ T\ U\ W\ R\ O\ \mathrm{X}_1\ \mathrm{X}_2\ \mathrm{X}_3$
- Domains:
  $\{0,1,2,3,4,5,6,7,8,9\}$
- Constraints:
  $allDiff(F,T,U,W,R,O)$
  $O + O = R + 10 * X_1$
  …

```
  T W O
+ T W O
-------
F O U R
```



Example: Sudoku

- Variables:
  - Each (open) square
- Domains:
  - $\{1,2,3,4,5,6,7,8,9\}$
- Constraints:
  - 9-way allDiff for each column
  - 9-way allDiff for each row
  - 9-way allDiff for each region



Varieties of CSPs

- Discrete variables
  - Finite domains
    - Size d means $O(d^n)$ complete assignments
    - For example, Boolean CSPs (NP-complete)
  - Infinite domains (integers, strings, etc.)
    - For example, job scheduling in OS
    - Linear constraints are solvable, nonlinear are undecidable
- Continuous variables
  - For example, start/end times for Hubble Telescope observations
  - Linear constraints here mean LP methods (LP is subset of CSP)

Real-World CSPs

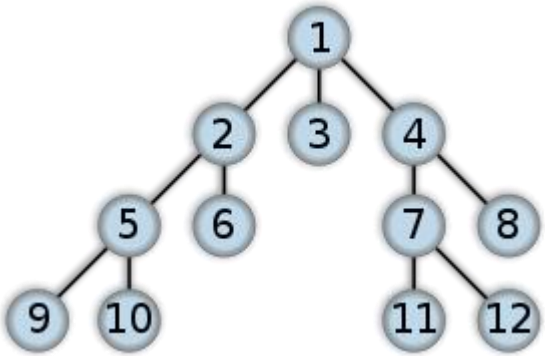
- Assignment problems: students to projects based on their preference and available places
- Timetable problems: Room planning, which courses to take
- Hardware and software configuration: Find a Linux kernel configuration for my hardware with certain options enabled
- Factory and production scheduling
- Circuit layout
- Fault diagnosis
- Economic optimization

Solving CSP Problems

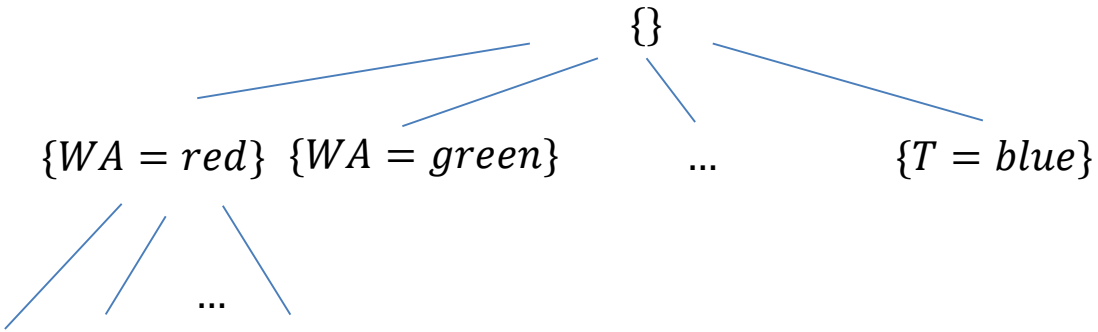
Formulating as Search Problem

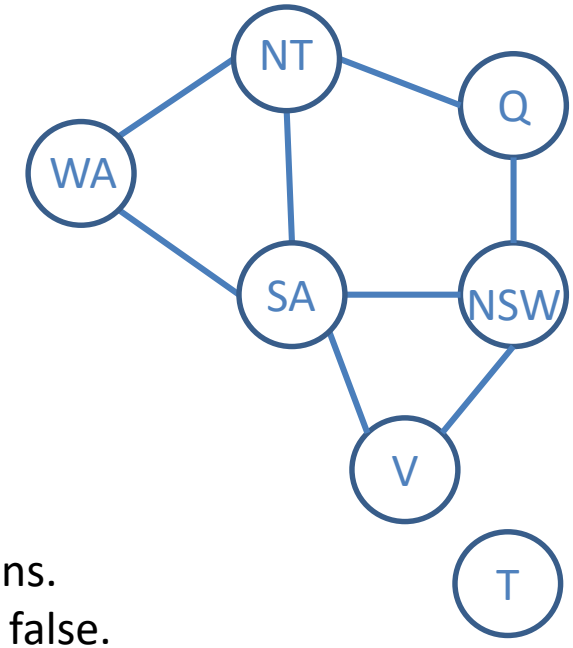
- Partial solutions (ie., partial assignments) are denoted as states
  - Initial state is the empty assignment: {}
  - Successor function: Assigns a value to an unassigned variable from its domain
  - Goal test: Tests whether the current assignment is complete and satisfies all contraints

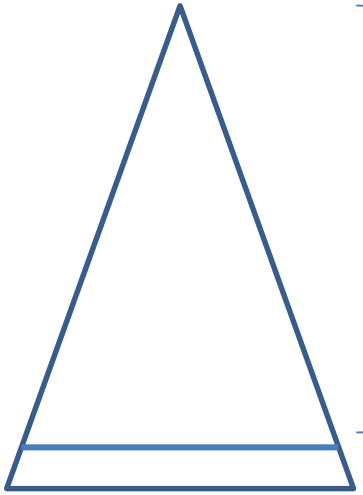
- Start with Breadth First Search
  - Traversing or search a tree or graph
  - Explores neighbors first before moving to the next level



BFS for Map Coloring





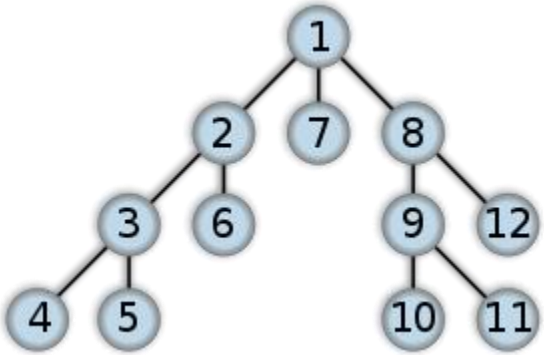


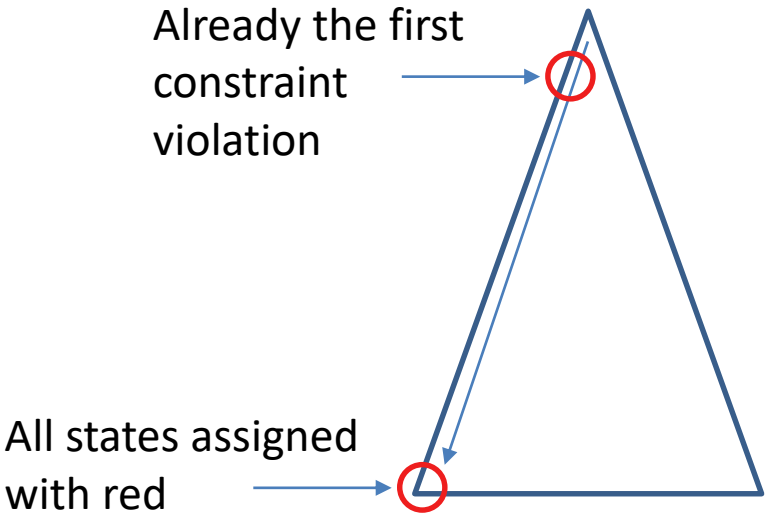
All partial solutions.
Goal test returns false.
Will be computed first.

Complete assignments.
Will be computed last.

DFS for Map Coloring

- Depth-First Search
  - Explores as far as possible one branch



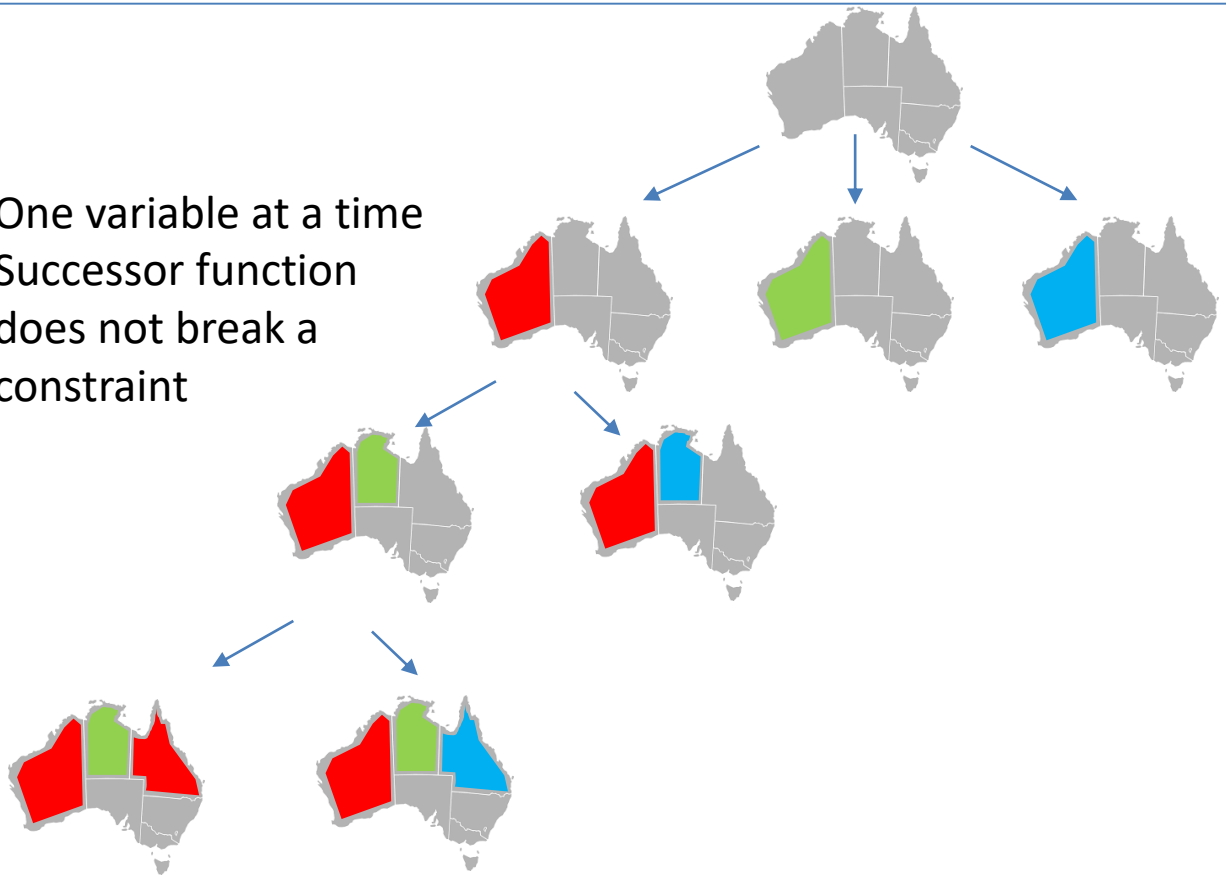


Backtracking

- Basic uninformed search algorithm for CSPs
- Idea: Consider only one variable at a time
  - Fix the ordering of assignments, since assignments are commutative {WA=red} then {NT=blue} == {NT=blue} then WA=red}
- Idea: Check constraints in each step
  - For assignments, we use only values that are not in conflict with already made assignments
  - Called incremental goal test
- DFS + these concepts = Backtracking search (solves 25-queens problem)

Going Through Backtracking

One variable at a time
Successor function does not break a constraint



Backtracking Algorithm

$S \leftarrow \{\quad\}$ initial solution with no assignments
$CSP \leftarrow CSP$ model, containing variables and constraints
return $backtrack(S, CSP)$

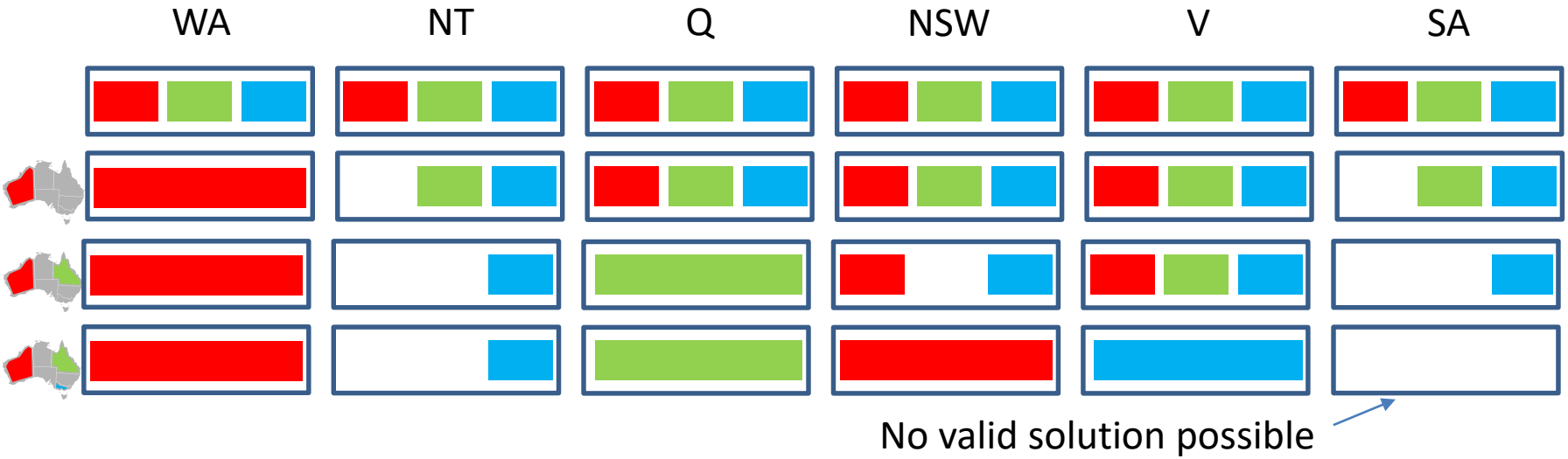
procedure $backtrack(S, CSP)$
  if S is complete **then**
    return S
  $var \leftarrow SelectUnassignedVariable(S, CSP)$
  for each $value \in OrderDomainValues(var, S, CSP)$ **do**
    if $isConsistentAssignment(value, S, CSP)$ **then**
      $S \leftarrow S \cup \{var \leftarrow value\}$
      $result \leftarrow backtrack(S, CSP)$
      if $result \neq failure$ **then**
        return $result$
      $S \leftarrow S/\{var \leftarrow value\}$
  return $failure$

Improving Backtracking

- These ideas yield a huge search-speed improvement and apply to a large class of problems
- **Ordering**:
  - Do we find a solution faster when we adjust the assignment order?
  - Should we also consider the ordering of the domain values?
- **Filtering**:
  - Can we detect states early on that will always lead to a failed solution later on?
- **Structure**:
  - Are there structural properties we might exploit for search?

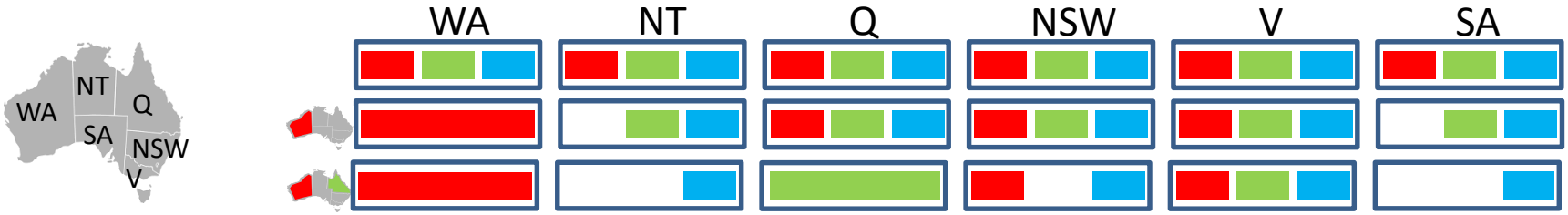
Filtering using Forward Checking

- Idea: Keep track of unassigned variables such that we know what values might be assigned in future to them and can remove bad options
- Forward Checking:
  - **Remove values** that would violate a constraint when applied to the current state



Filtering using Constraint Propagation

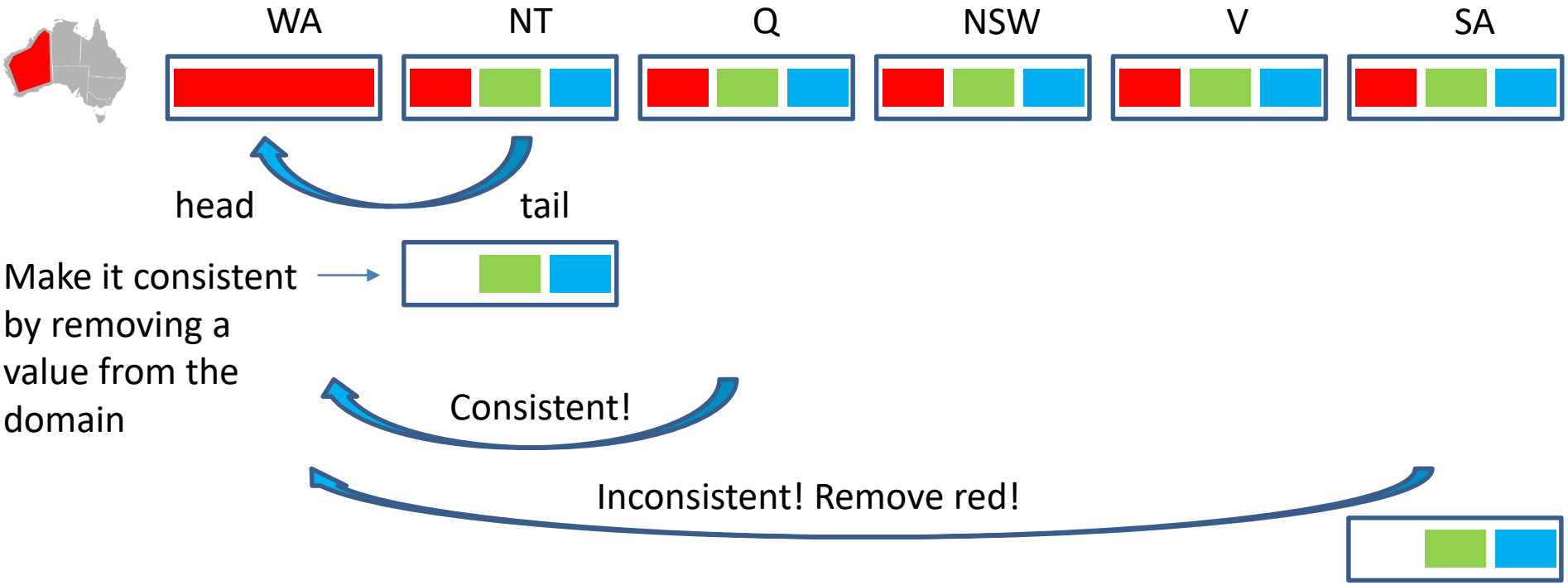
- Forward Checking: No early detection for failure states, because only forward checking between assigned to unassigned variables



- SA and NT will have a conflict in the future
- Idea: Detect this situation as early as possible
- Solution: Constraint propagation -> reason from constraint to constraint

Arc Consistency

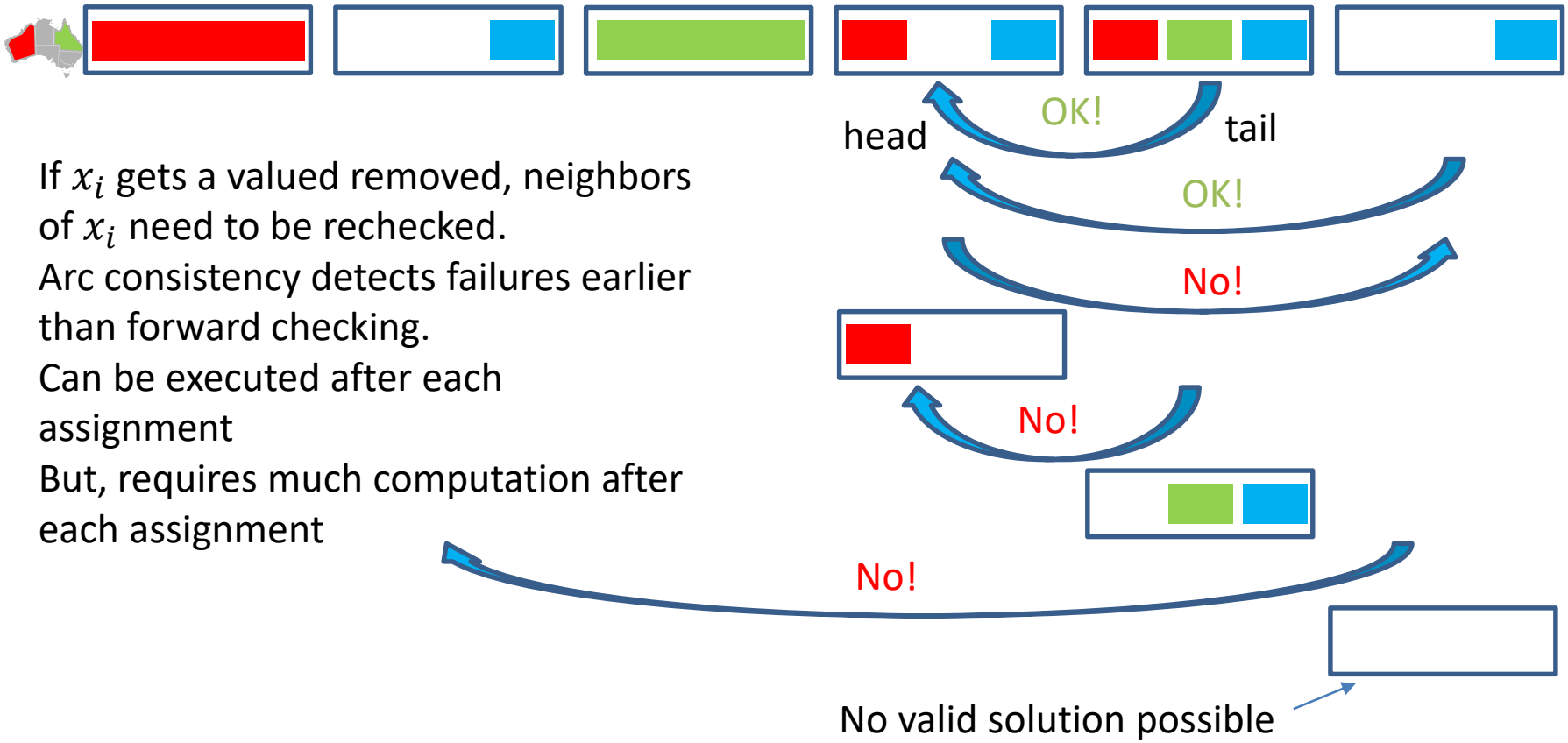
- An arc $x_i \rightarrow x_j$ is consistent iff for $\forall a \in domain(x_i)$ in the tail of the arc, there $\exists\ b \in domain(x_j)$ in the head such that (a, b) satisfies the binary constraint between x_i and x_j



Forward checking enforces consistency of arcs pointing to each new assignment

Ensuring Arc Consistency of the Entire CSP

- Idea: Propagate consistency checks through all assignments to make sure that **all** arcs are consistent



- If x_i gets a valued removed, neighbors of x_i need to be rechecked.
- Arc consistency detects failures earlier than forward checking.
- Can be executed after each assignment
- But, requires much computation after each assignment

Arc Consistency Algorithm: AC-3

$CSP \leftarrow CSP$ model, containing variables and constraints

$Queue \leftarrow \{\quad\}$
for $x_i \in getVariables(CSP)$ **do**
  for $x_j \in getVariables(CSP)$ **do**
    if $x_i \neq x_j$ **then**
      $Queue \leftarrow Queue \cup \{(x_i, x_j), (x_j, x_i)\}$

Add all arcs from the CSP to the $queue$

while $Queue$ is not empty **do**
  $(x_i, x_j) \leftarrow Pop(queue)$
  if $RemoveInconsistentValues(x_i, x_j)$ **then**
    for each $x_k \in getNeighbors(X_i)$ **do**
      $Queue \leftarrow Queue \cup \{(x_k, x_i)\}$

Go through each arc and enforce consistency by deleting values from the domain. If values have been removed, we have to recheck all dependent arcs.

Remove Inconsistent Values

$CSP \leftarrow CSP$ model, containing variables and constraints
$x_i \leftarrow$ tail of the arc
$x_j \leftarrow$ head of the arc

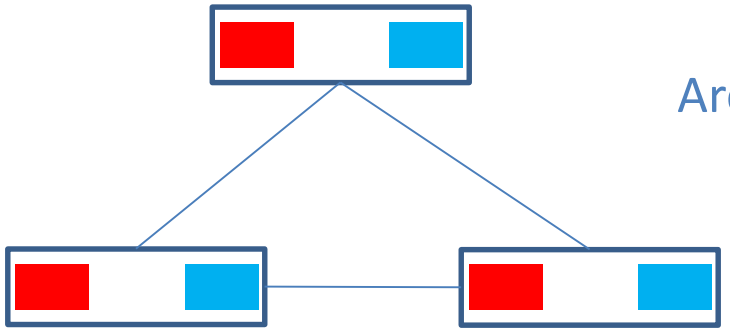
$removed \leftarrow \boldsymbol{false}$
for each $a \in getDomainValues(x_i)$ **do**
  if no value $b \in getDomainValues(x_j)\ satisfies\ (x_i, x_j)\ with\ (a, b)$ **then**
    $removeDomainValue(CSP, x_i, a)$
    $removed \leftarrow \boldsymbol{true}$
return $removed$

Has a quadratic complexity wrt. the number of values in a domain

The algorithm makes a CSP arc consistent!

Limitations of Arc Consistency

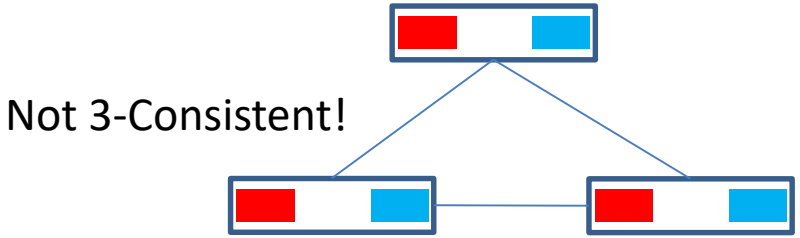
- After applying AC-3:
  - One solution left
  - Multiple solutions left
  - Can have no solutions left without knowing it!



Arc consistent with no valid solution!

Going Beyond Arcs: K-Consistency

- Consistency degrees:
  - 1-Consisntency (node consistency): Each single node's domain contains a value that meets that node's unary constraints
  - 2-Consistency (arc consistency): For each pair of nodes, any consistent assignment to one can be extended to the other
  - K-Consistency: For each k nodes, any consistent assignment to k-1 can be extended to the k^{th} node



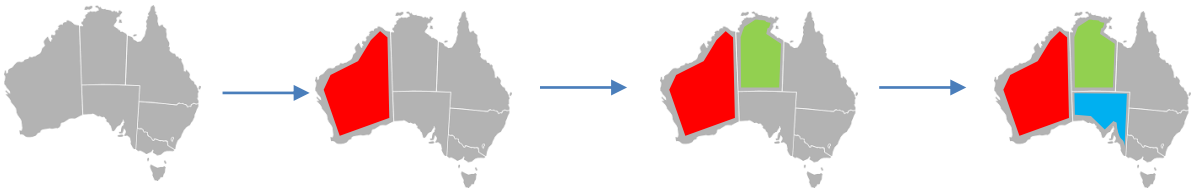
- Higher k require more computationally expensive checks

Strong K-Consistency

- Strong k-consistency: also k-1, k-2, …, 1 consistent
- Is strong n-consistent CSP solvable without backtracking?
  - Choose any assignment to any variable
  - Choose a new variable
  - By 2-consistency, there is a choice consistent with the first
  - Choose a new variable
  - By 3-consistency, there is a choice consistent with the first 2
  - …

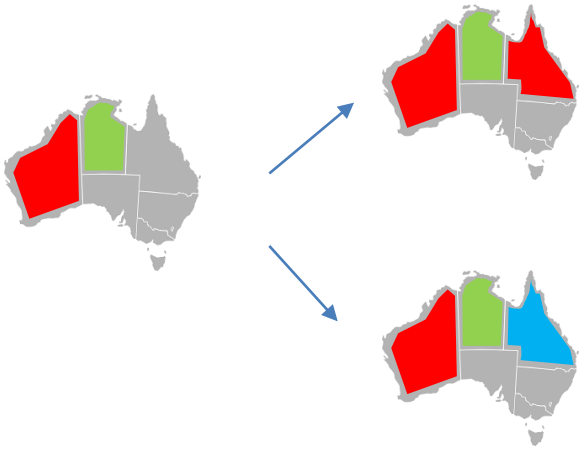
Ordering: MRV

- Idea: Assign values for variables that has the fewest number of values in the domain left
  - Minimum remaining values (MRV) algorithm
  - Why using minimum not maxim?
    - Most constrained variable
    - Fail-fast ordering



Ordering: LCV

- Idea: Select the value of a domain, choose the **least constraining value**
  - The value that rules out the fewest values in other variables
  - Might be computation intensive



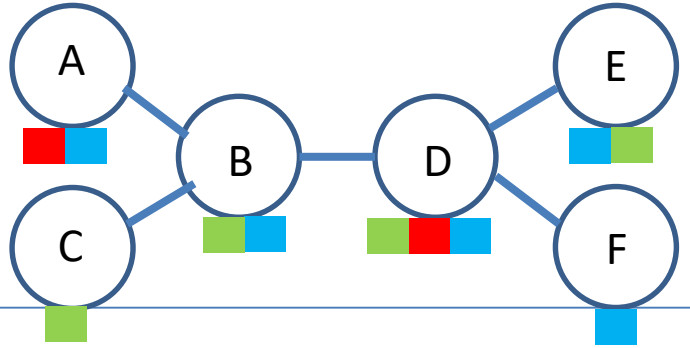
Use this option, as it does not remove blue from SA

- So, choose hardest variable, but easiest domain value
  - We touch every variable anyway, but we do not want to touch every value! (1000-queens with ordering possible!)

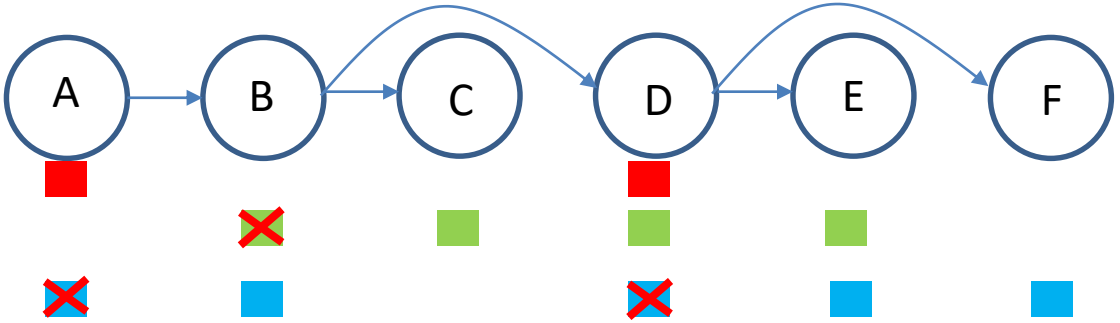
Improve with Problem Structure

- Extreme case: independent subproblems
  - For example, Tasmania and mainland have no constraints
- Independent subproblems are identifiable as connected components of constraint graph
- Example: n variables can be broken into subproblems of only c variables
  - Worst-case solution: $O(\left(\frac{n}{c}\right)(d^c))$, which is linear in n
  - $n = 80, d = 2, c = 20$: $2^{80} = 4\text{billion years at } 10\text{mio}\,\frac{\text{nodes}}{\text{s}}$
  - $(4)(2^{20}) = 0.4\text{ seconds at } 10\text{mio}\,\frac{nodes}{s}$

Tree-Structured CSPs

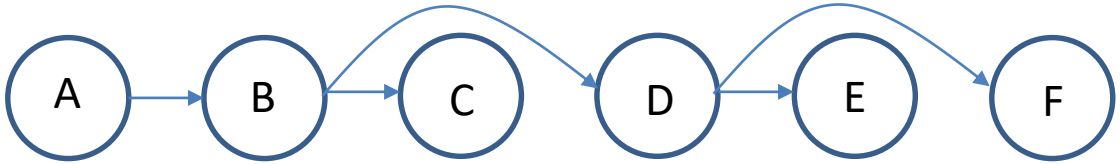


- Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\ d^2)$ time (vs. $O(d^n)$)
- Algorithm:
  - Order: Choose a root variable, order variables so that parents precede children
  - Remove backward: **For** $i = n: 2$, apply $RemoveInconsistent(Parent(\mathrm{x_i}), \mathrm{x_i})$
  - Assign forward: **For** $i = 1: n$, assign x_i consistently with $Parent(x_i)$



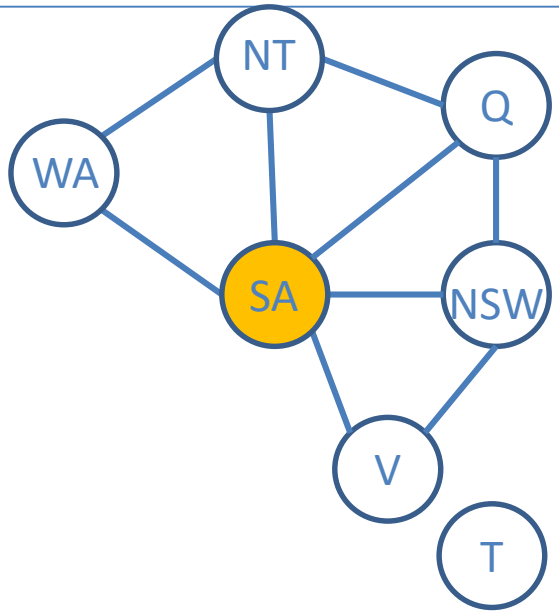
Properties of Tree-Structured CSPs

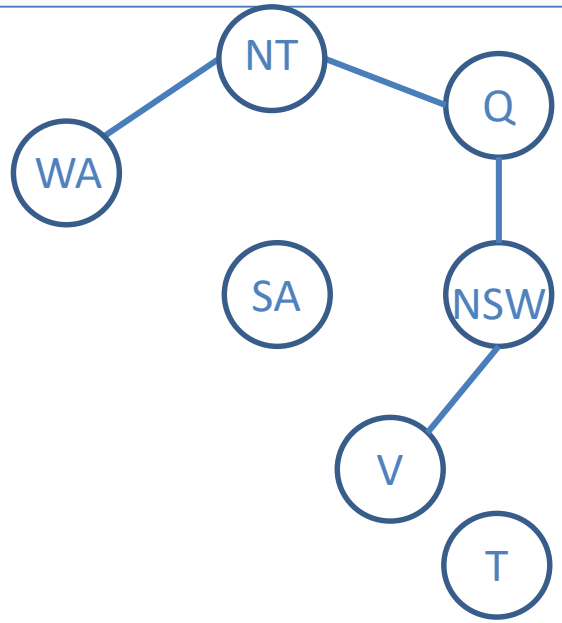
- Property 1: After backward pass, all root-to-leaf arcs are consistent
  - Proof: Each $x_i \rightarrow x_j$ was made consistent at one point and x_j's domain could not have been reduced thereafter, because x_j's children were processed before x_j and only these could affect the values in the domain



- Property 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
  - Proof: Induction on position

Improving Structure: Nearly Tree-Structured CSPs





- Conditioning: Instantiate a variable and prune its neighbor's domains
- Cutset conditioning: Instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

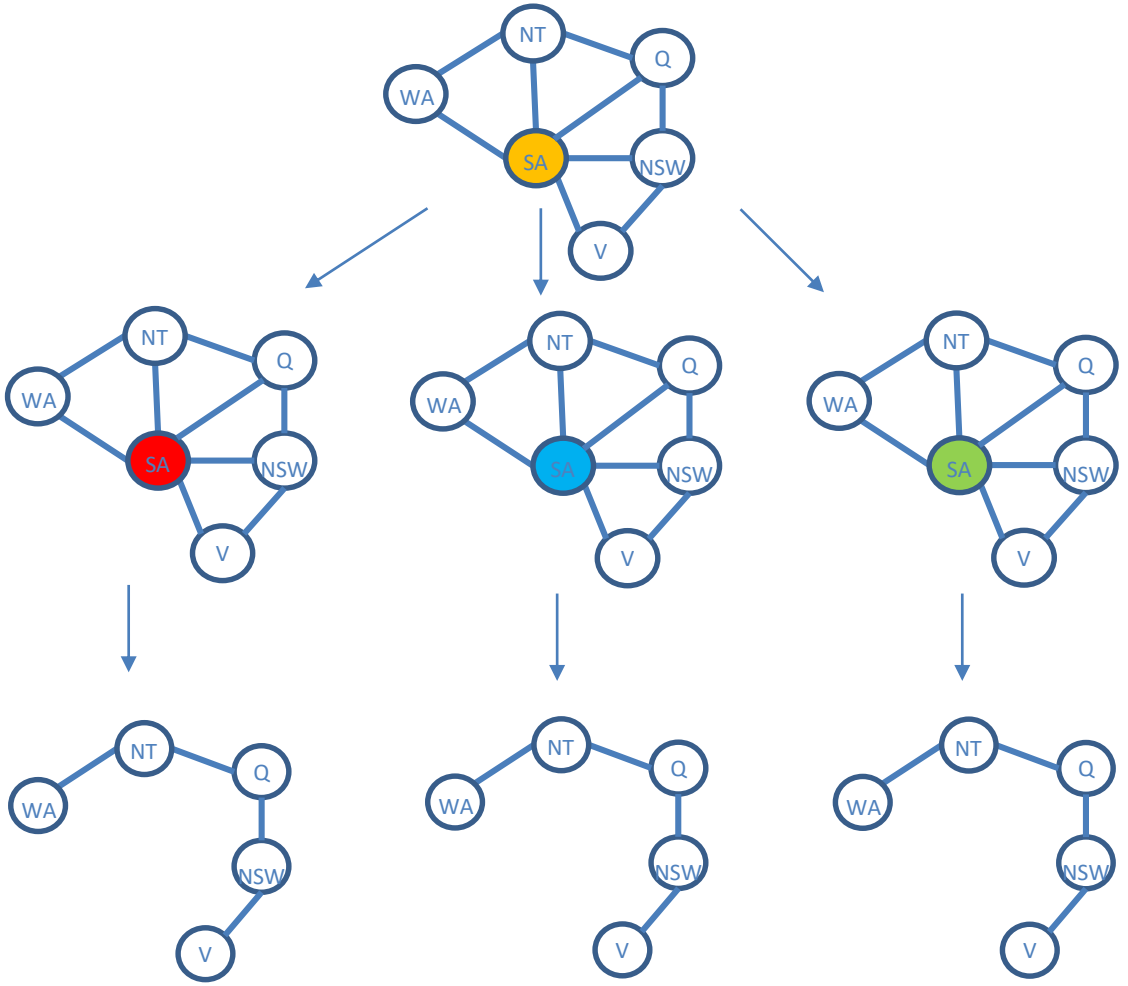
Cutset Condition

Choose a cutset

Instantiate the cutset (all possible ways)

Compute residual CSP for each assignment

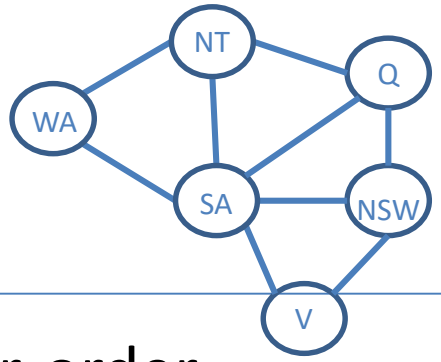
Solve the residual CSPs (tree structured)



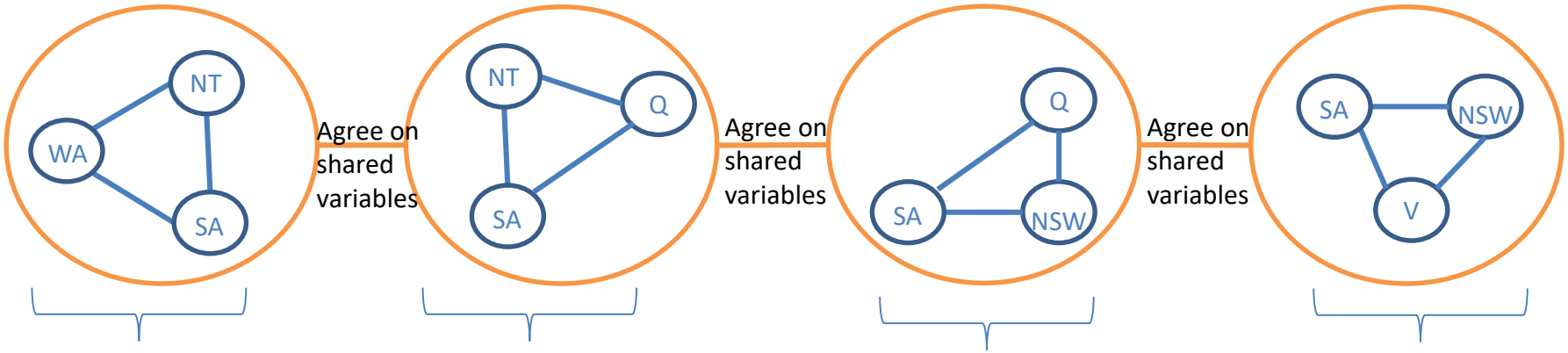
Exponential in size of the cutset

Linear in the size of what is left

Improving Structure: Tree Decomposition



- Idea: Create a tree-structured graph of higher-order variables
- Each higher-order variable encodes a subgraph
- Subproblems need to overlap to ensure valid solutions



$V_1 = \{(WA = r, SA = g, NT = b), \ldots\}$

$V_2 = \{(SA = g, NT = b, Q = r), \ldots\}$

$V_3 = \{\ldots\}$

$V_4 = \{\ldots\}$

Solving for higher-order variables can now be applied with tree-structured algorithm

Iterative Algorithms for CSPs

- Idea: Take any assignment, which even might break constraints and iteratively fix the constraints

- Algorithm:
  - Generate a random variable assignment for each variable
  - While not valid solution:
    - Variable selection: Randomly select any conflicted variable
    - Value selection: min-conflict heuristic:
      - Reassign that variable a value that violates the fewest constraints

- Solves 10,000,000-queens problem in constant time with high probability!

Take Home Message:

- Constraint satisfaction problem is a search technique to find a valid solution in a constraint solution space
  - A solution is a complete assignment of variables from values of their respective domains such that all constraints are satisfied
- Usually represented with graphs and binary constraints between variables
- General purpose optimization techniques vastly improve search time
  - Backtracking, filtering (consistency), ordering, structure

Next Lecture

- Dimensionality reduction
  - Principal component analysis
  - Feature Selection