# Techniques in Artificial Intelligence

## Constraint Satisfaction Problems (CSP)

# Constraint satisfaction problems (CSPs)

Standard search problem:
  state is a "black box"—any old data structure
    that supports goal test, eval, successor

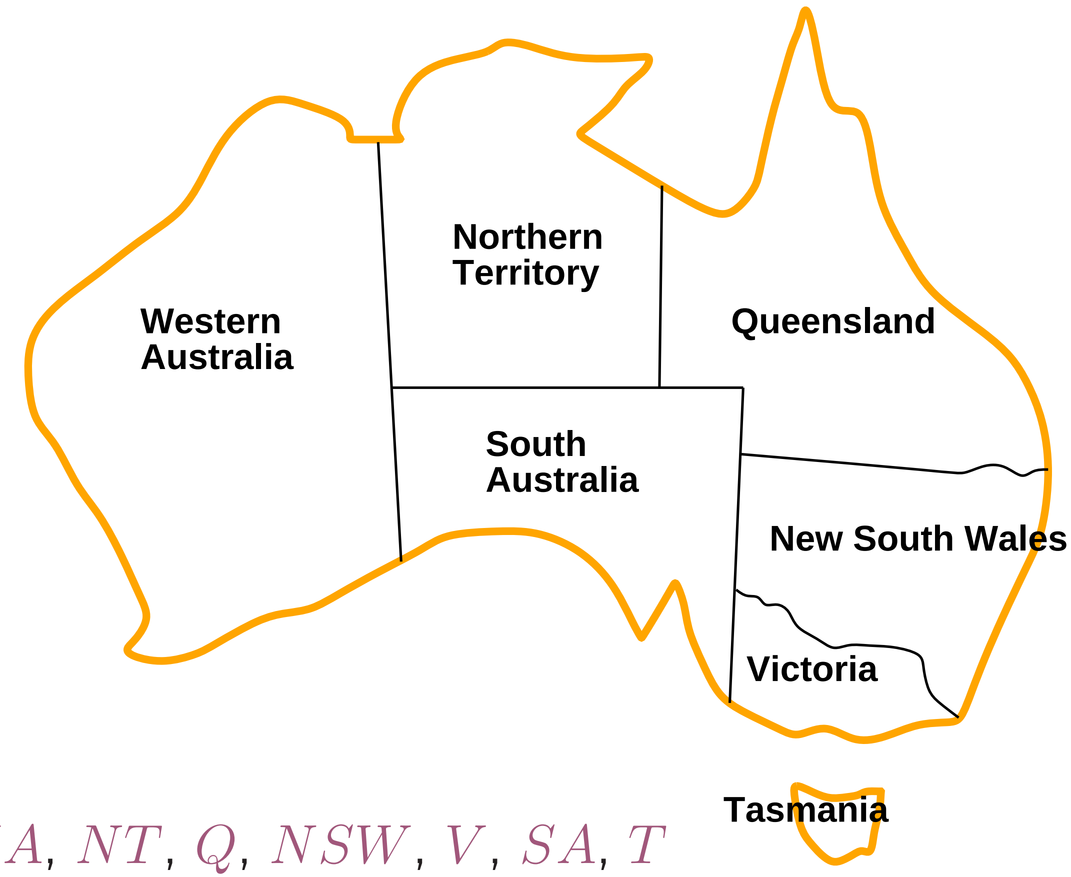
CSP:
  state is defined by variables $X_i$ with values from domain $D_i$

  goal test is a set of constraints specifying
    allowable combinations of values for subsets of variables

Simple example of a **formal representation language**

Allows useful **general-purpose** algorithms with more power than standard search algorithms

# Example: Map-Coloring

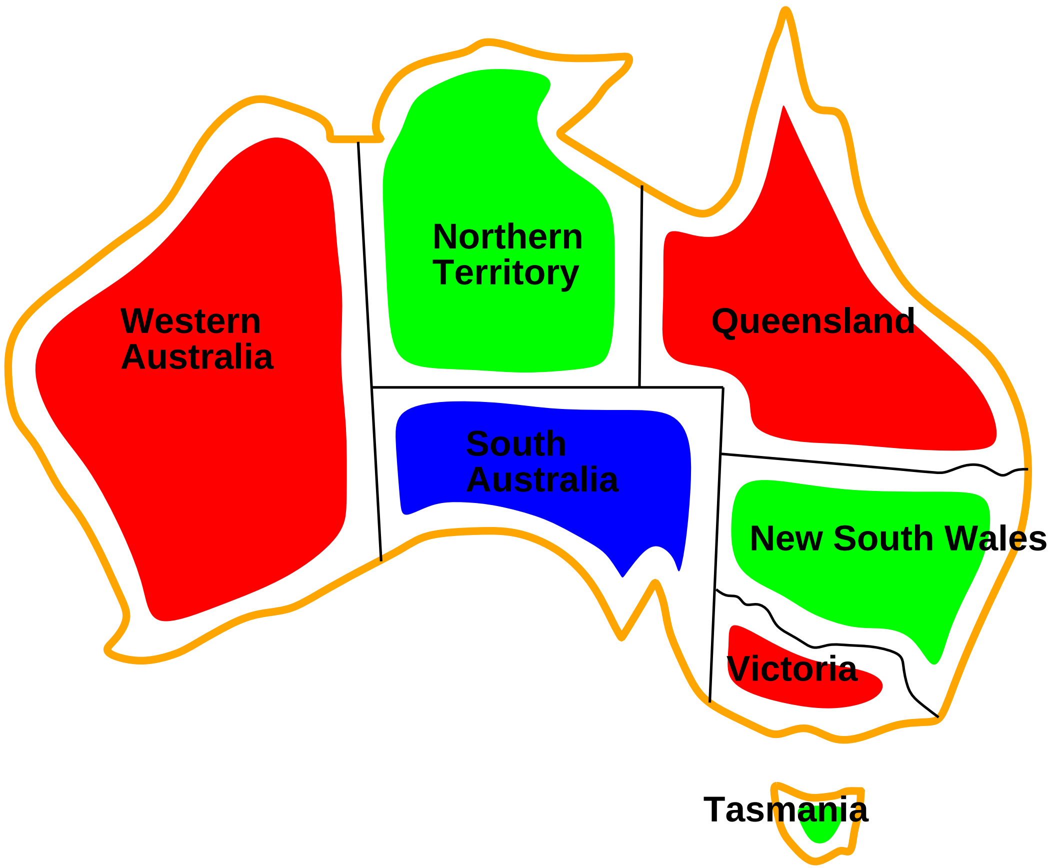



Variables $WA$, $NT$, $Q$, $NSW$, $V$, $SA$, $T$
Domains $D_i = \{red, green, blue\}$
Constraints: adjacent regions must have different colors

e.g., $WA \neq NT$ (if the language allows this), or

$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

# Example: Map-Coloring contd.

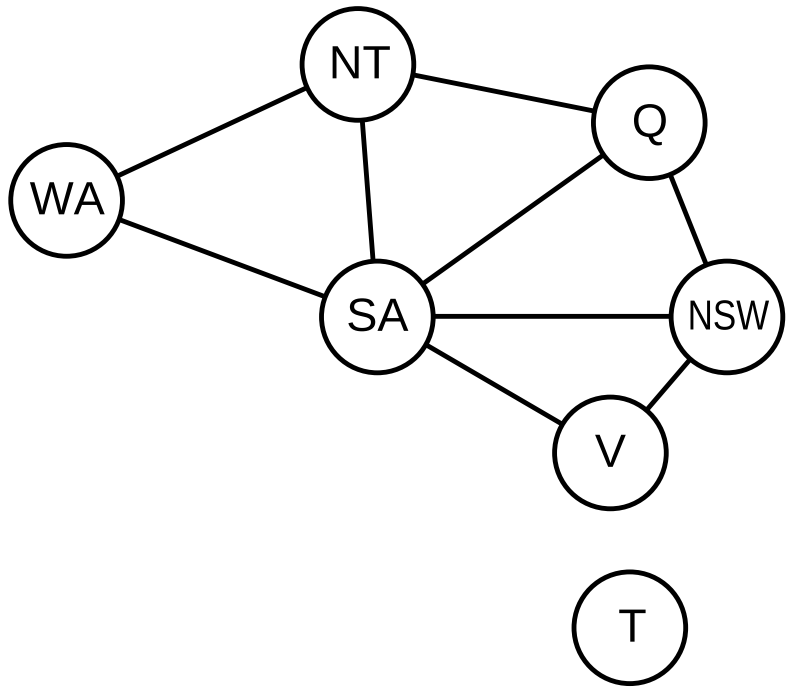



Solutions are assignments satisfying all constraints, e.g.,
$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$

# Constraint graph

Binary CSP: each constraint relates at most two variables

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!

# Varieties of CSPs

Discrete variables

finite domains; size $d \Rightarrow O(d^n)$ complete assignments

- e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)

infinite domains (integers, strings, etc.)

- e.g., job scheduling, variables are start/end days for each job
- need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
- linear constraints (i.e. variables appear in linear form) are solvable
- nonlinear constraints are undecidable

Continuous variables

- e.g., start/end times for Hubble Telescope observations
- linear constraints solvable in polynomial time by LP methods

# Varieties of constraints

Unary constraints involve a single variable,
e.g., $SA \neq green$

Binary constraints involve pairs of variables,
e.g., $SA \neq WA$

Higher-order constraints involve 3 or more variables,

Preferences (soft constraints), e.g., $red$ is better than $green$
often representable by a cost for each variable assignment
$\rightarrow$ constrained optimization problems

# Real-world CSPs

Assignment problems
e.g., who teaches what class

Timetabling problems
e.g., which class is offered when and where?

Hardware configuration

Spreadsheets

Transportation scheduling

Factory scheduling

Floorplanning

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

# Standard search formulation (incremental)

Let's start with the straightforward, dumb approach, then fix it

States are defined by the values assigned so far

◊ Initial state: the empty assignment, $\{\ \}$

◊ Successor function: assign a value to an unassigned variable
that does not conflict with current assignment.
$\Rightarrow$ fail if no legal assignments (not fixable!)

◊ Goal test: the current assignment is complete

◊ Path cost: constant cost (e.g. 1) for every step

This is the same for all CSPs!

If we have $n$ variables, every solution appears at depth $n$
$\Rightarrow$ depth-first search algorithms are popular for CSP

# Problem!

How bad can things get?

Say we have $n$ variables each taking a maximum of $d$ possible values

Suppose we apply breadth-first search.

Branching factor at top level is $n \times d$

Branching factor at next level is $(n-1) \times d$

Branching factor at depth $l$ is $(n-l) \times d$

And so on for $n$ levels

Hence, we generate a tree with $n!d^n$ leaves!!!! ☹

# Backtracking search

However, there is good news: Path is irrelevant!

Variable assignments are commutative, i.e.,

$[WA = red$ then $NT = green]$ same as $[NT = green$ then $WA = red]$

Why not order our variables, and assign values to them in that order?

Only need to consider assignments to a single variable at each node

$\Rightarrow$ branching factor $= d$ and there are $d^n$ leaves

Depth-first search for CSPs with single-variable assignments
is called backtracking search

Backtracking search is the basic uninformed algorithm for CSPs

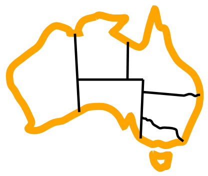
Can solve $n$-queens for $n \approx 25$

# Backtracking search

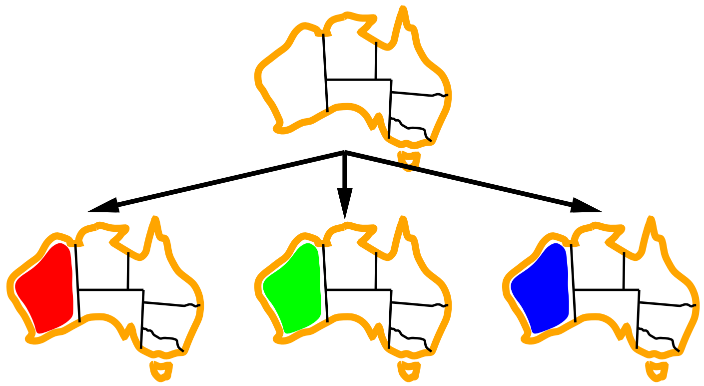
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)

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
```

The line: *var* ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*], *assignment*, *csp*) selects the next unassigned variable (in the order given by the list VARIABLES[*csp*].

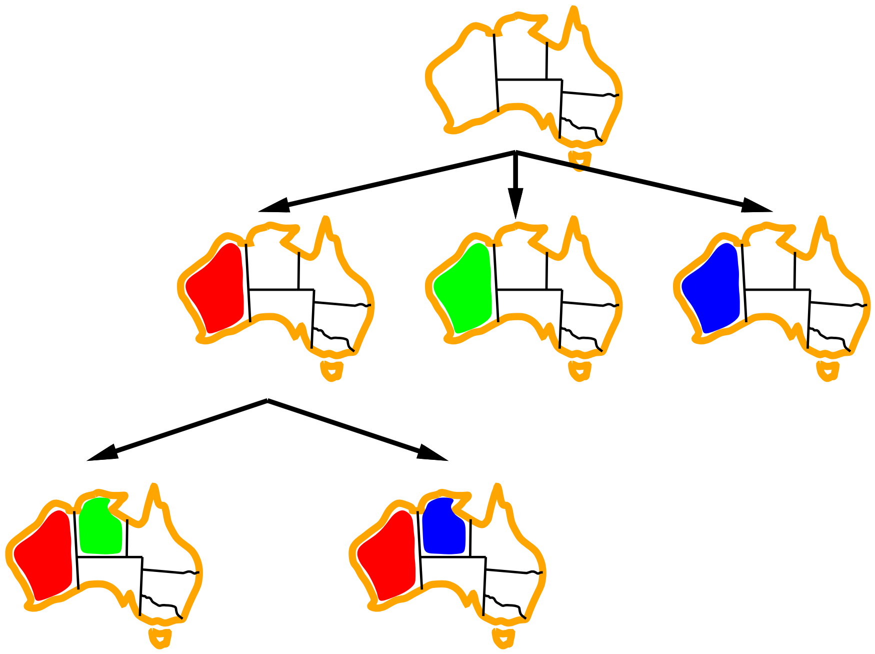
# Backtracking example



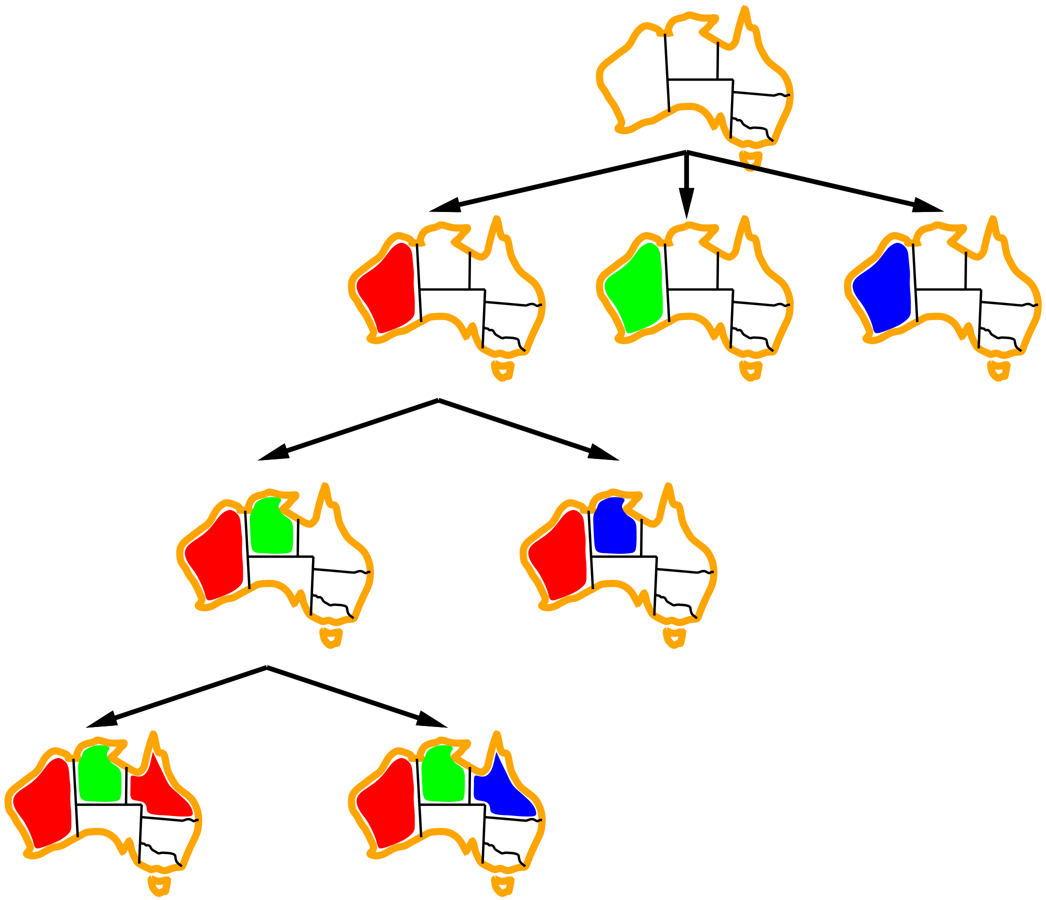
# Backtracking example



# Backtracking example



# Backtracking example



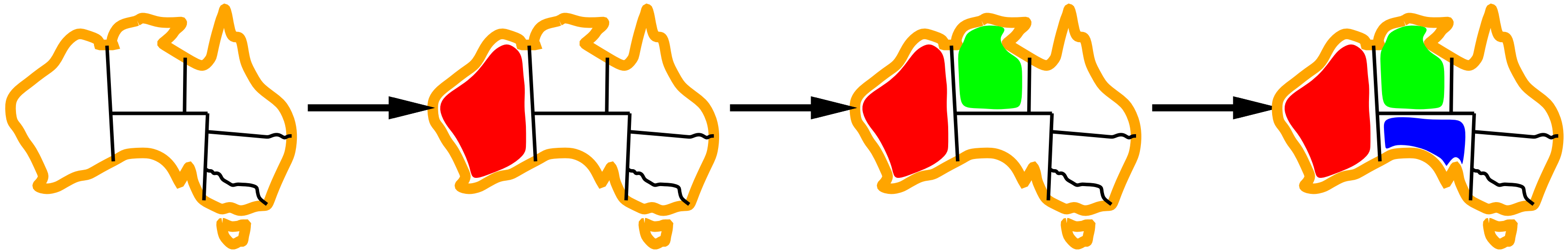
# Improving backtracking efficiency

**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

Minimum remaining values (MRV):
choose the variable with the fewest legal values



Idea: Variables with less remaining values are more likely to cause failure soon in the current branch.

If there is a variable $X$ with zero legal values remaining, the MRV heuristic will select $X$ and failure is detected immediately (avoiding pointless search through other variables which always will fail when $X$ is finally selected).

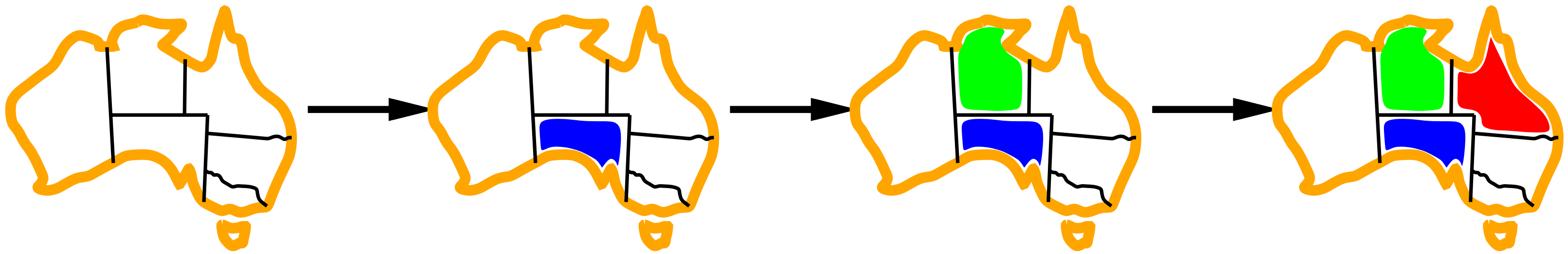
# Degree heuristic

When choosing first Australian state, cannot use MRV!

Degree of a variable: the number of constraints on the variable

Degree heuristic:
    choose the variable involved in most constraints on remaining variables

$SA$ has degree 5, other have degree 2 or 3, except $T$ which has 0

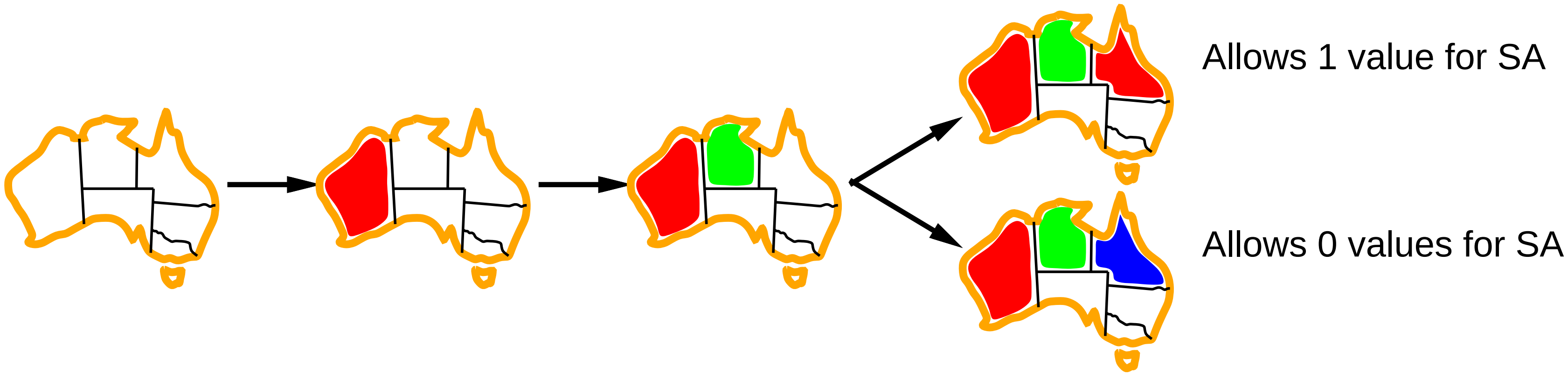


Degree heuristic can be used as a tie-breaker among MRV variables

Using degree heuristic, this colouring problem can be solved without any failed steps (i.e. without any backtracking). Check it yourself at home!

# Least constraining value

Given a variable, choose the least constraining value:
the one that rules out the fewest values in the remaining variables



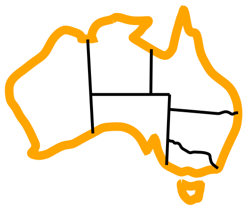

Idea: If we're trying to find one solution, this may help find it faster

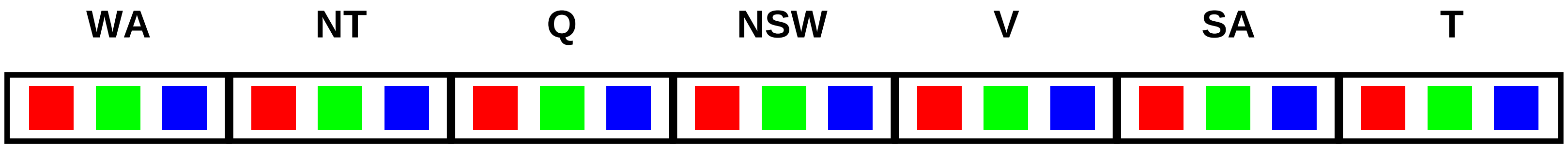
If we're trying to find all solutions, it doesn't make any difference

Combining all above heuristics allows solving massive problems (e.g. makes 1000 queens feasible)

# Forward checking

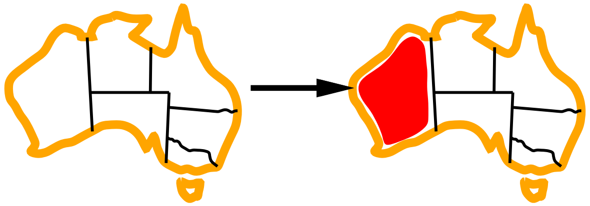
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

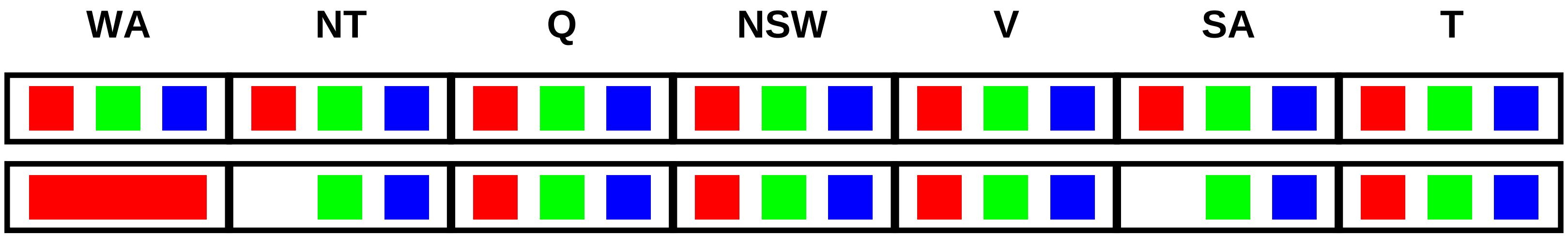




# Forward checking

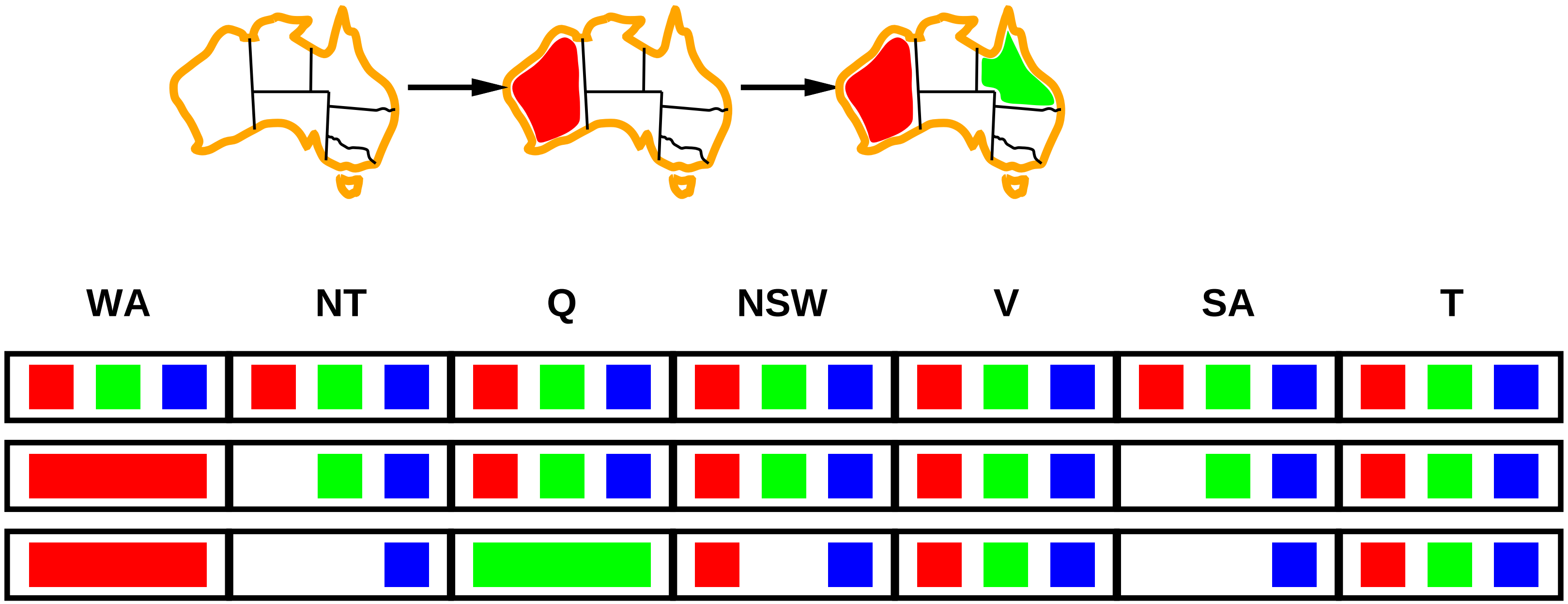
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values





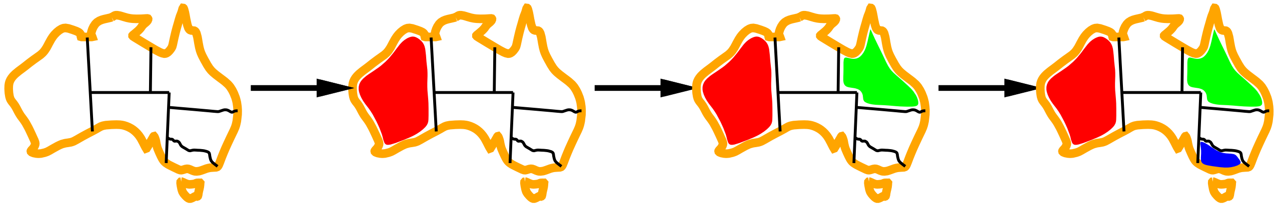
# Forward checking

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



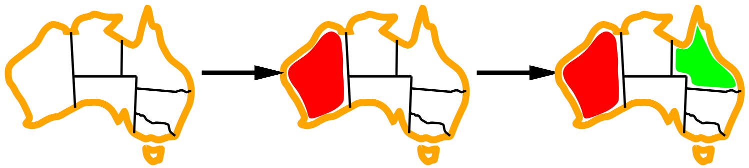
# Forward checking

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



# Constraint propagation

Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:




$NT$ and $SA$ cannot both be blue!

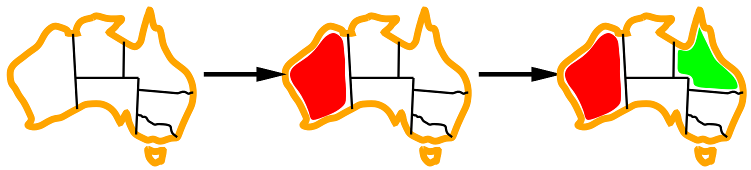
Constraint propagation repeatedly enforces constraints locally

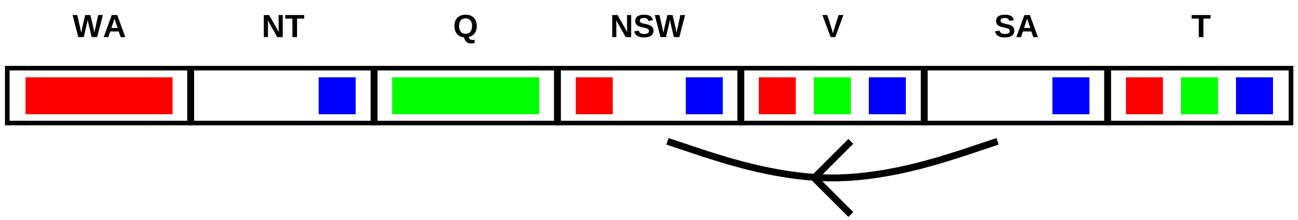
# Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff

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



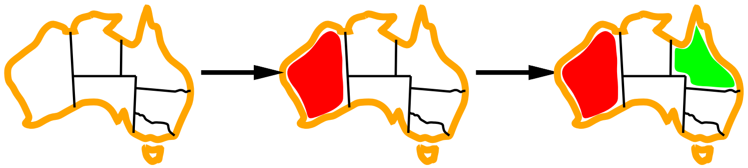


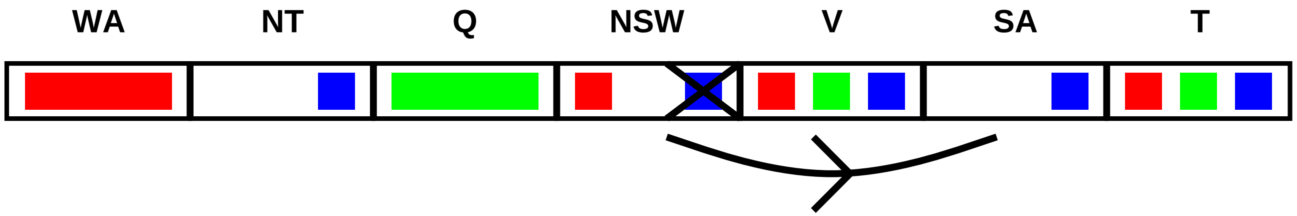
# Arc consistency

Simplest form of propagation makes each arc consistent

$X \to Y$ is consistent iff

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



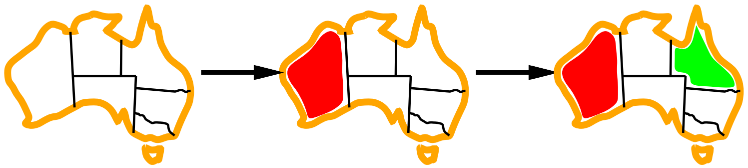


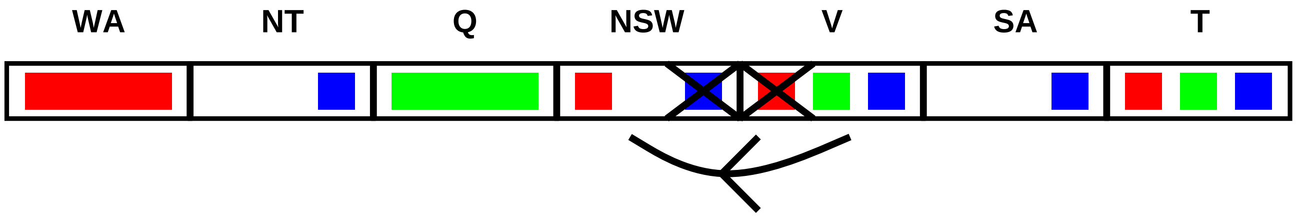
# Arc consistency

Simplest form of propagation makes each arc consistent

$X \to Y$ is consistent iff

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





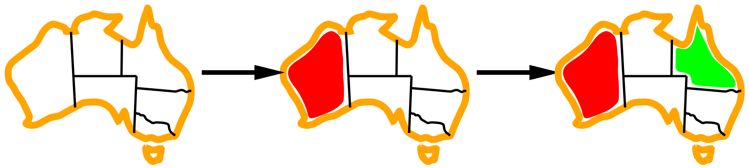

If $X$ loses a value, neighbors of $X$ need to be rechecked

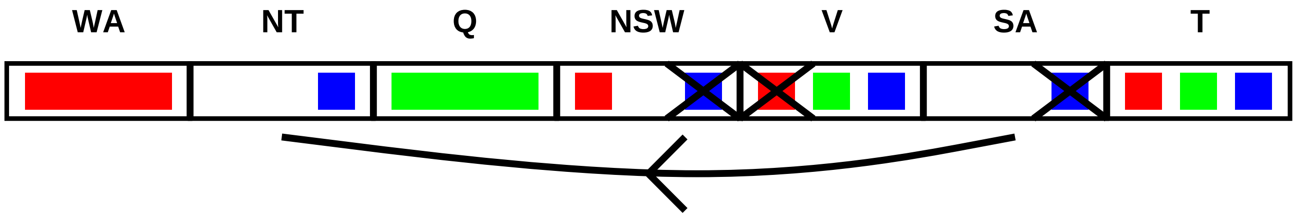
# Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff

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






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 $\{X_1, X_2, \ldots, X_n\}$
  **local variables**: *queue*, a queue of arcs, initially all the arcs in *csp*

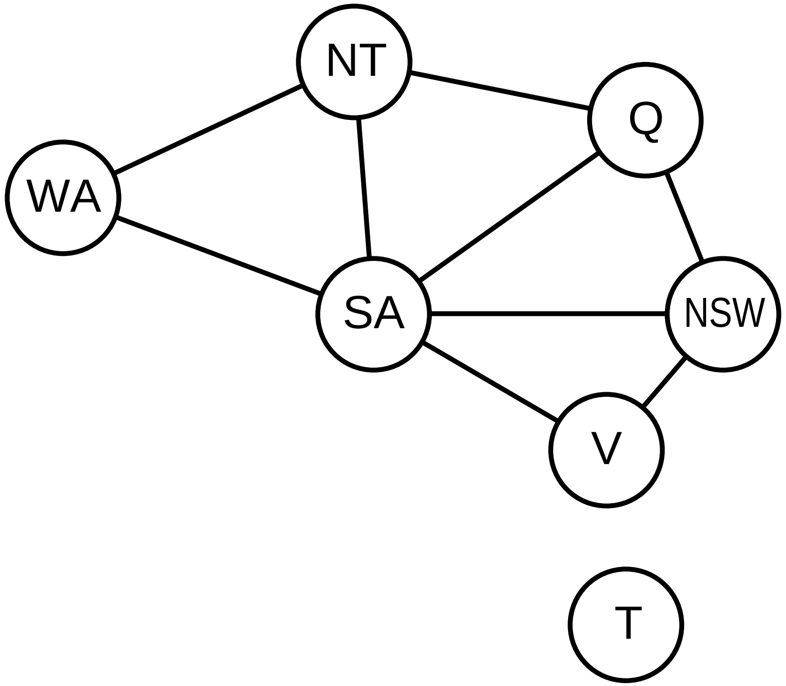
  **while** *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ Remove-First(*queue*)
    **if** Remove-Inconsistent-Values($X_i, X_j$) **then**
      **for each** $X_k$ **in** Neighbors[$X_i$] **do**
        add $(X_k, X_i)$ to *queue*

---

**function** Remove-Inconsistent-Values( $X_i, X_j$) **returns** true iff succeeds
  *removed* $\leftarrow$ *false*
  **for each** $x$ **in** Domain[$X_i$] **do**
    **if** no value $y$ in Domain[$X_j$] allows $(x,y)$ to satisfy the constraint $X_i \leftrightarrow X_j$
      **then** delete $x$ from Domain[$X_i$]; *removed* $\leftarrow$ *true*
  **return** *removed*

$O(n^2d^3)$, can be reduced to $O(n^2d^2)$ (but detecting **all** is NP-hard)

# Problem structure




Tasmania and mainland are independent subproblems

Identifiable as connected components of constraint graph

Each connected component is a subproblem

# Problem structure contd.

Why does it matter?

Suppose each subproblem has $c$ variables out of $n$ total

Then, there are $n/c$ subproblems

Each subproblem takes at most $d^c$ to solve (where $d$ is the maximum number of values per variable)

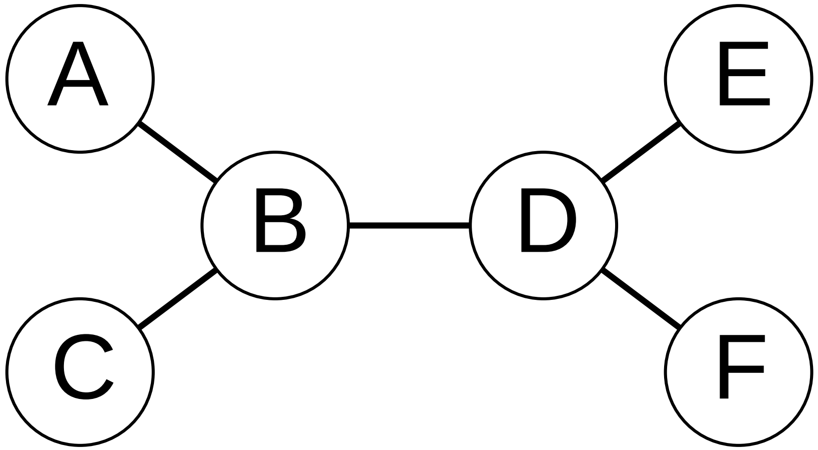
Therefore, worst-case solution cost is $n/c \cdot d^c$, **linear** in $n$

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

$2^{80}$ = 4 billion years at 10 million nodes/sec

$4 \cdot 2^{20}$ = 0.4 seconds at 10 million nodes/sec

# Tree-structured CSPs



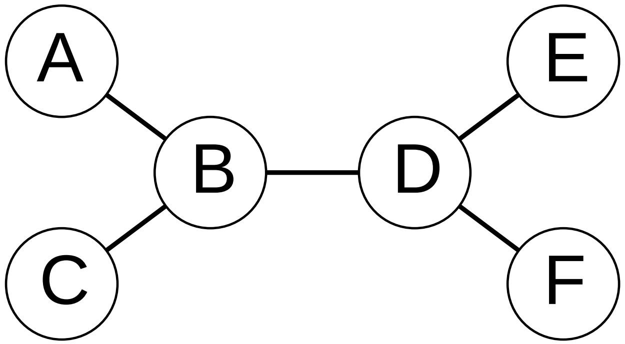

Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\,d^2)$ time

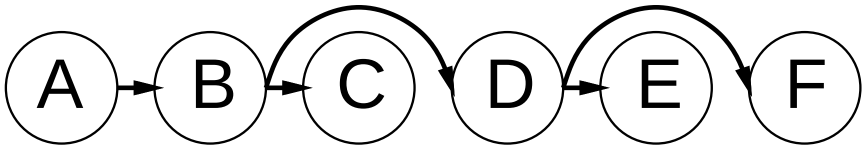
Compare to general CSPs, where worst-case time is $O(d^n)$

This property also applies to logical and probabilistic reasoning:
an important example of the relation between syntactic restrictions
and the complexity of reasoning.

# Algorithm for tree-structured CSPs

1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering.
Label the variables $X_1$ to $X_n$ in order.



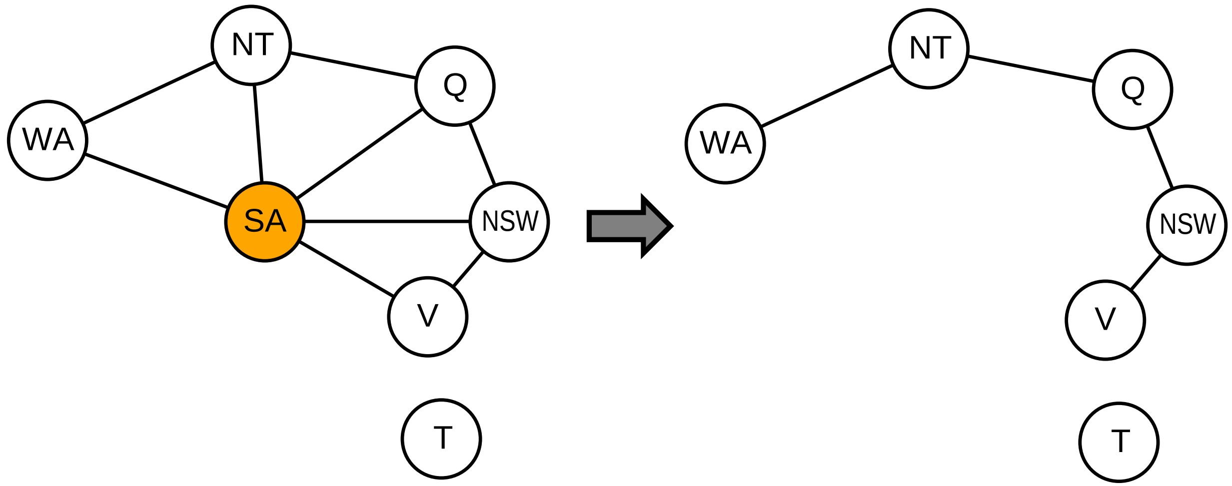

2. For $j$ from $n$ down to $2$, apply arc consistency to the arc $(X_i, X_j)$ for each $X_i \in Parents(X_j)$, removing values from $\textsc{Domain}[X_i]$ as necessary
I.e. apply $\textsc{RemoveInconsistent}(Parent(X_j), X_j)$

3. For $j$ from $1$ to $n$, assign $X_j$ consistently with $Parent(X_j)$

Why is this good? Because after applying step 2, the assignment of values in step 3 requires no backtracking.

# Nearly tree-structured CSPs

Conditioning: instantiate a variable, prune its neighbors' domains




Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

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

"complete" states, i.e., all variables assigned

# 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

# Outline

Required reading:
Russell & Norvig, AI: A Modern Approach, 3rd Edition, Chapter 6