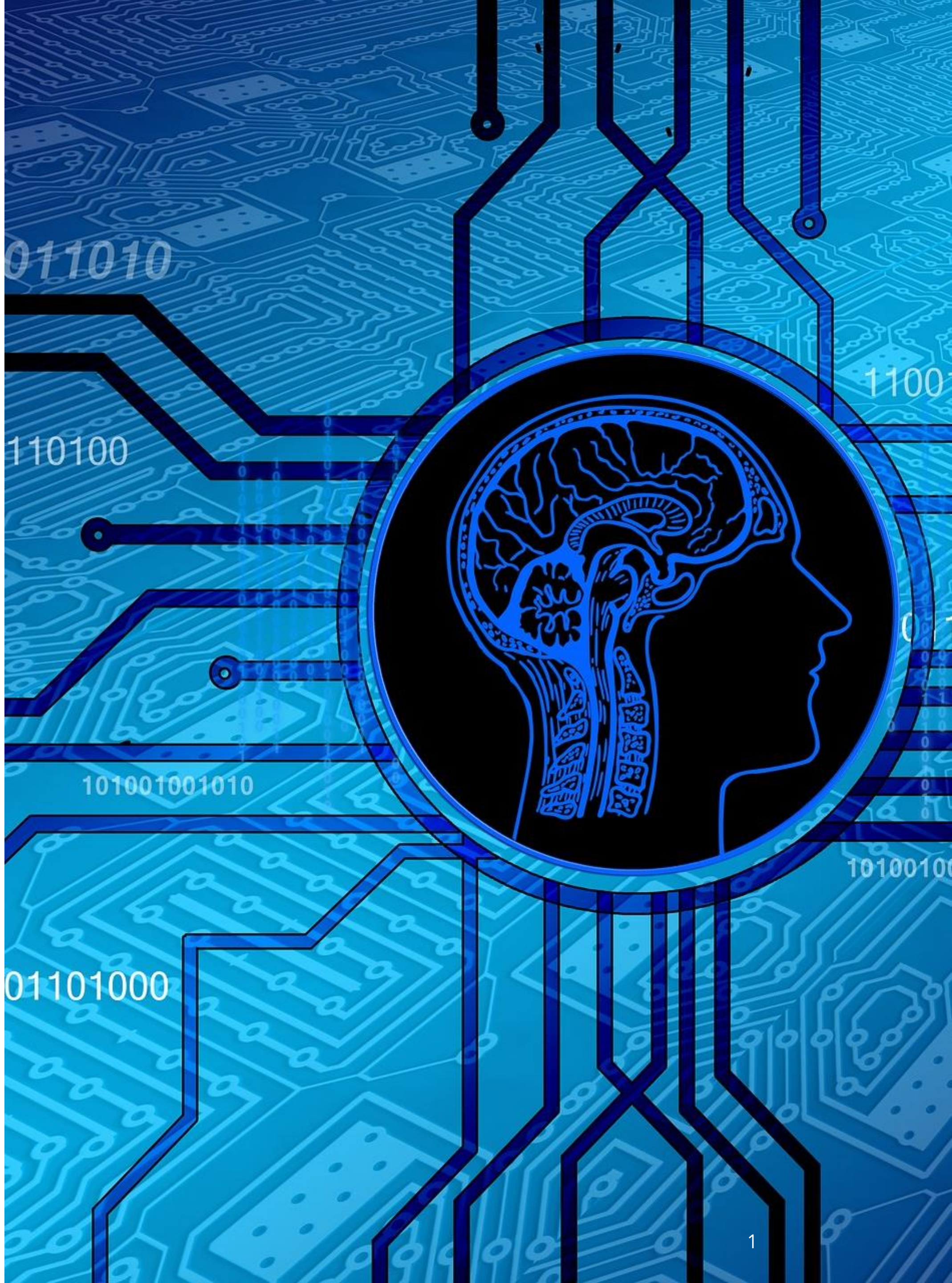
Smart Search using Constraints

Informatics 2D: Reasoning and Agents

Lecture 5

Adapted from slides provided by Dr Petros Papapanagiotou



Constraint satisfaction problems (CSPs)

State

- Set of *variables* X_i with *values* from *domain* D_i

Actions

- *Assign* a value to a variable

Goal test

- A set of *constraints* specifying allowable combinations of values for subsets of variables

Path cost

- None

Constraint satisfaction problems (CSPs)

State

- Set of *variables* X_i with *values* from *domain* D_i

Actions

- *Assign* a value to a variable

Goal test

- A set of *constraints* specifying allowable combinations of values for subsets of variables

Pa

Simple example of a *formal representation language*.

Allows useful <u>*general-purpose*</u> algorithms with more power than standard search algorithms.

Structure of a CSP

- A set of **variables**: $X=\{X_1,\dots X_n\}$
- A set of **domains**: $D=\{D_1,\dots D_n\}$
  - each domain D_i is a set of possible values for variable X_i
- A set of **constraints** C that specify acceptable combinations of values.
  - Each $c \in C$ consists of:
    - a **scope** – tuple of variables (neighbours) involved in the constraint
    - a **relation** that defines the values that the variables can take

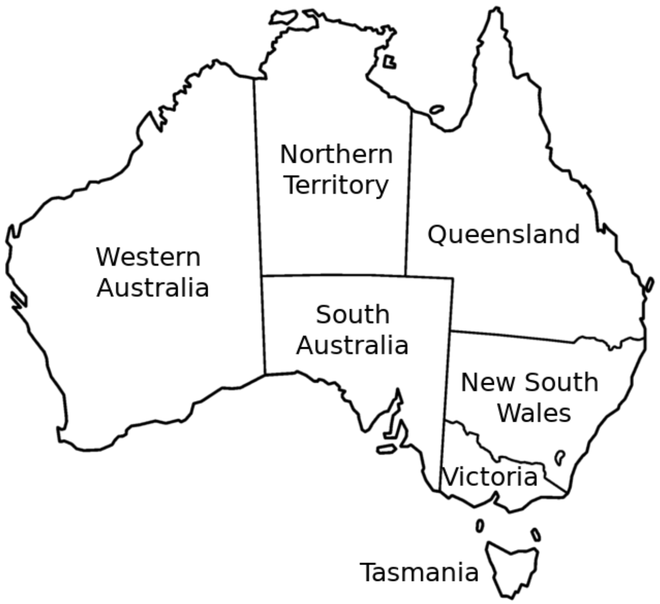
Example: Map-Colouring

Variables: {WA, NT, Q, NSW, V, SA, T}

Domains: D_i = {red, green, blue}

Constraints: adjacent regions must have different colours,

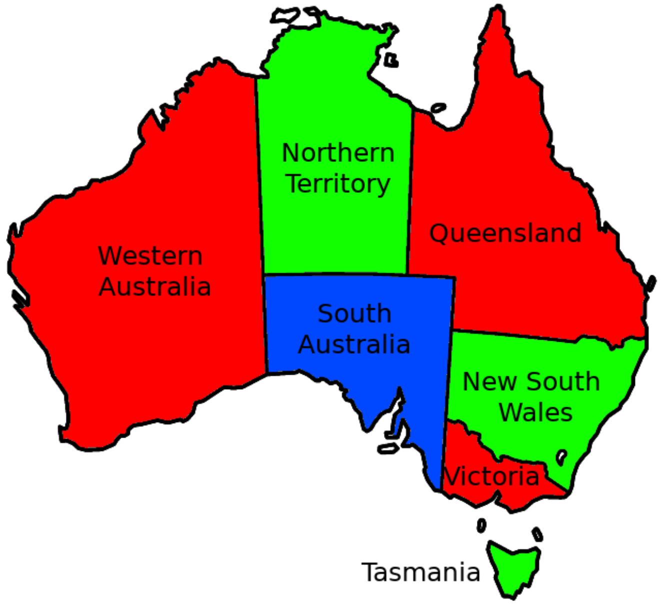
- e.g. WA ≠ NT,
- or (WA,NT) ∈ {(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)}.



Example: Map-Colouring

Solutions are *complete* and *consistent* assignments,

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



Constraint graph

Binary CSP:
each constraint relates two variables.

Constraint graph:
nodes are variables, arcs are constraints.

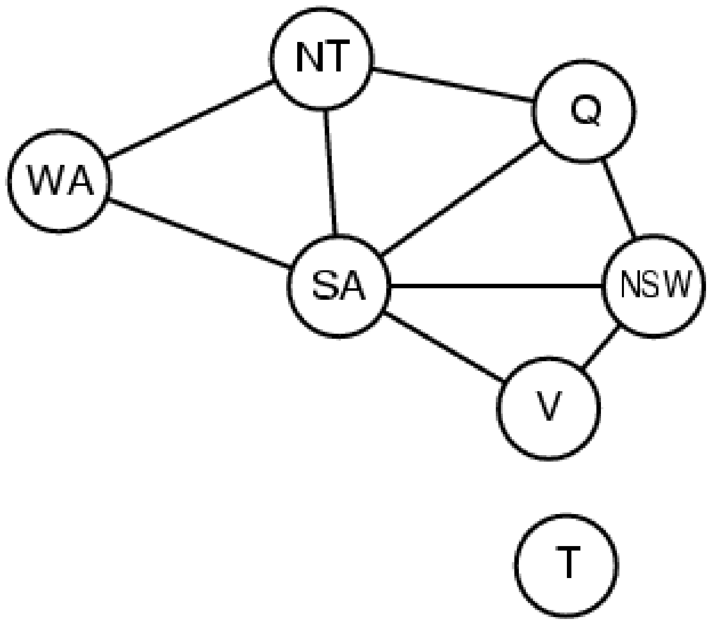
Constraint graph

Binary CSP:

each constraint relates two variables.

Constraint graph:

nodes are variables, arcs are constraints.



Varieties of CSPs

Discrete variables:

- finite domains:
  - *n* variables, domain size *d* → $O(d^n)$, 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. $StartJob_1 + 5 \leq StartJob_3$.

Continuous variables:

- e.g. start/end times for Hubble Space Telescope observations.
- linear constraints solvable in polynomial time by linear programming.

Real-world CSPs

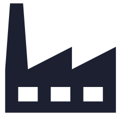
Assignment problems

e.g., who teaches what class.

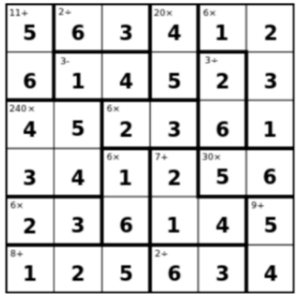
Timetabling problems

e.g., which class is offered when and where.

Transportation scheduling



Factory scheduling



Games

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

Varieties of constraints

Unary constraints involve a single variable,

- e.g., SA ≠ green.

Binary constraints involve pairs of variables,

- e.g., SA ≠ WA.

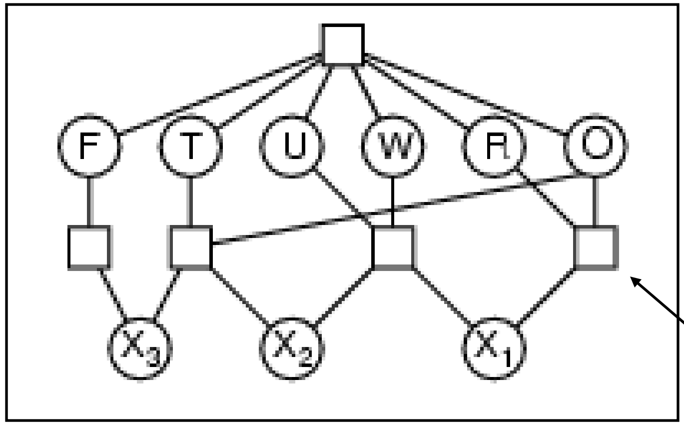
Higher-order constraints involve 3 or more variables,

- e.g., crypt-arithmetic column constraints.

Global constraints involve an arbitrary number of variables

Example: Crypt-arithmetic

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



constraint hypergraph

Hypernode
(n-ary constraint)

Variables: $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3.$

Domains: $\{0,1,2,3,4,5,6,7,8,9\}$.

Constraints:

- *Alldiff (F,T,U,W,R,O)* ← Global constraint
- $O + O = R + 10 \cdot X_1$
- $X_1 + W + W = U + 10 \cdot X_2$
- $X_2 + T + T = O + 10 \cdot X_3$
- $X_3 = F, T \neq 0, F \neq 0$

Search in CSPs

Standard search formulation (incremental)

- 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
  → fail if no legal assignments.

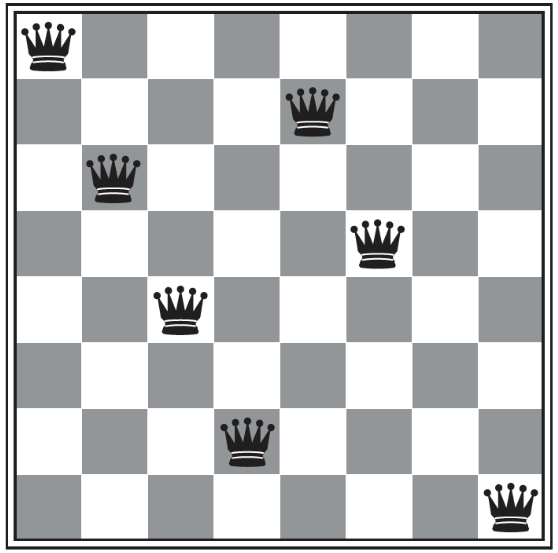
  Goal test: the current assignment is complete.

- For a CSP with *n variables,* every solution appears at depth *n*
  → use depth-first search!

Backtracking search

- Variable assignments are *commutative*,
  - e.g., [WA = red then NT = green] same as [NT = green then WA = red].
- Only need to consider assignments to a single variable at each node
- Depth-first search for CSPs with single-variable assignments is called *backtracking* search.
- Backtracking search is the basic uninformed algorithm for CSPs.

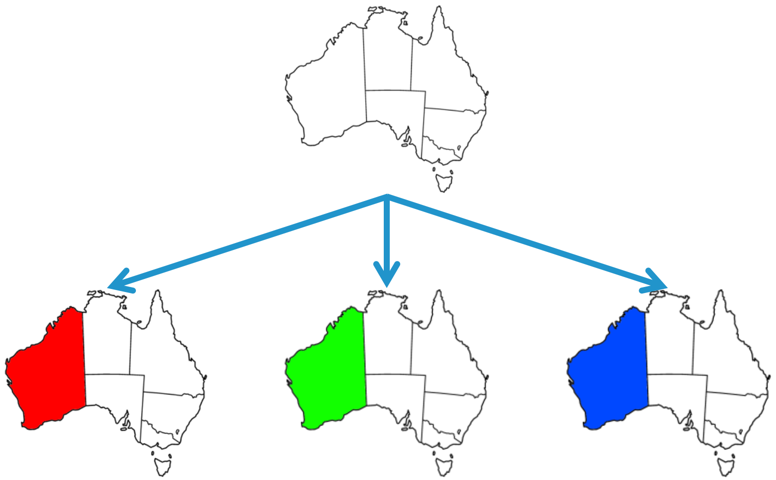
8-queens problem



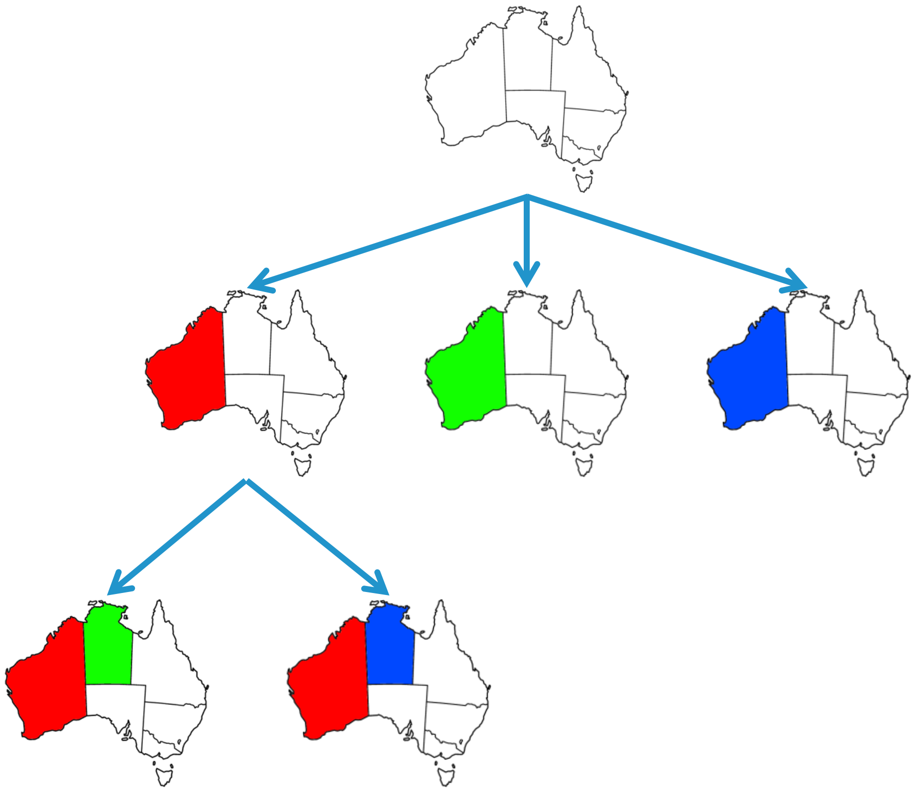
Backtracking search

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

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

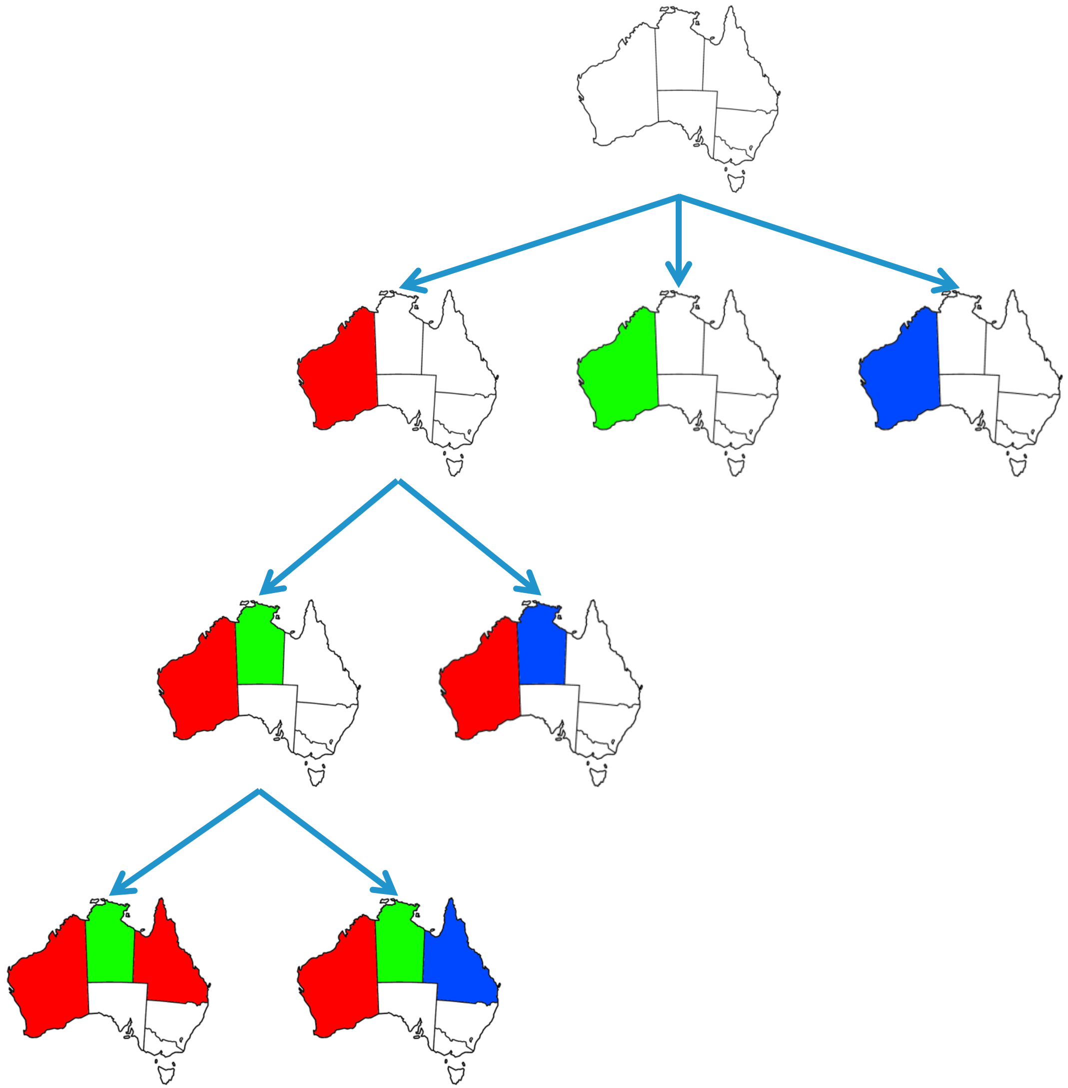
Backtracking example



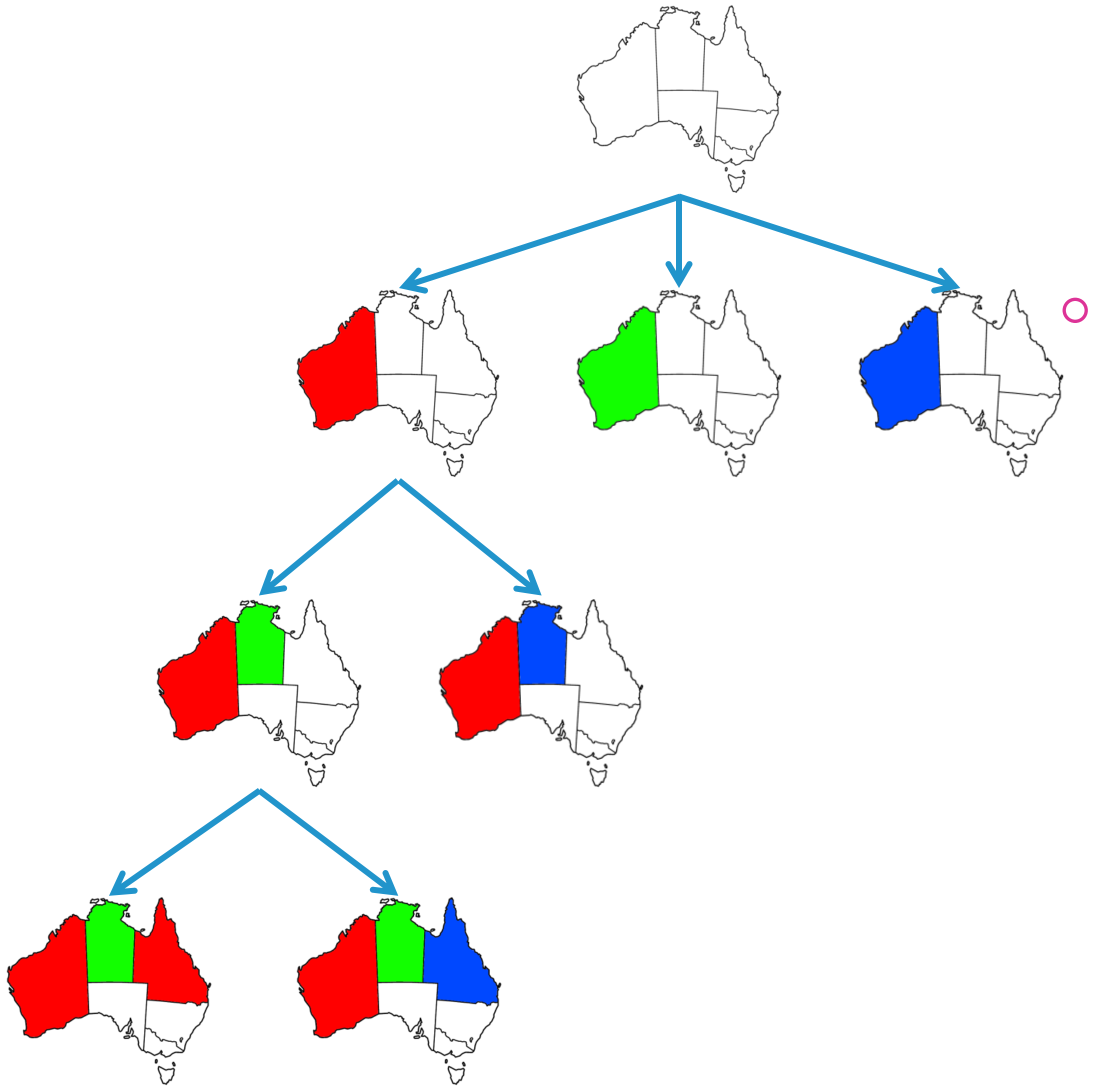
Backtracking example

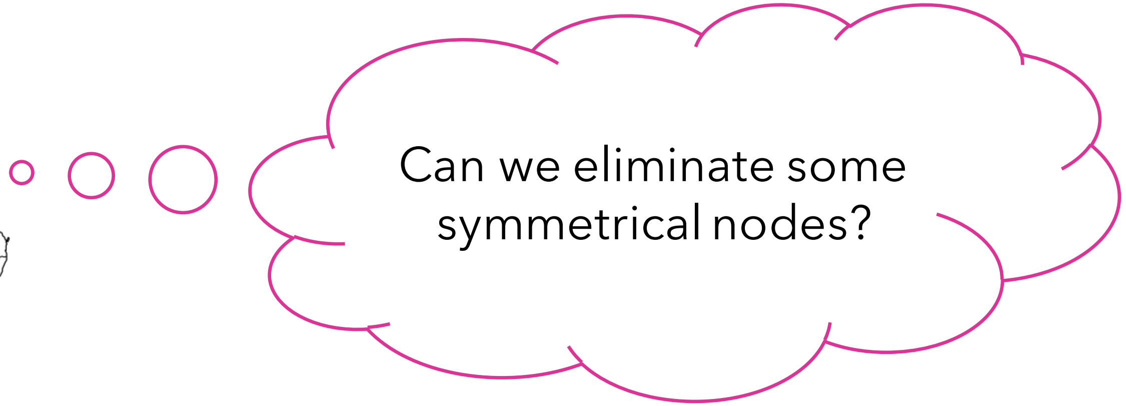


Backtracking example

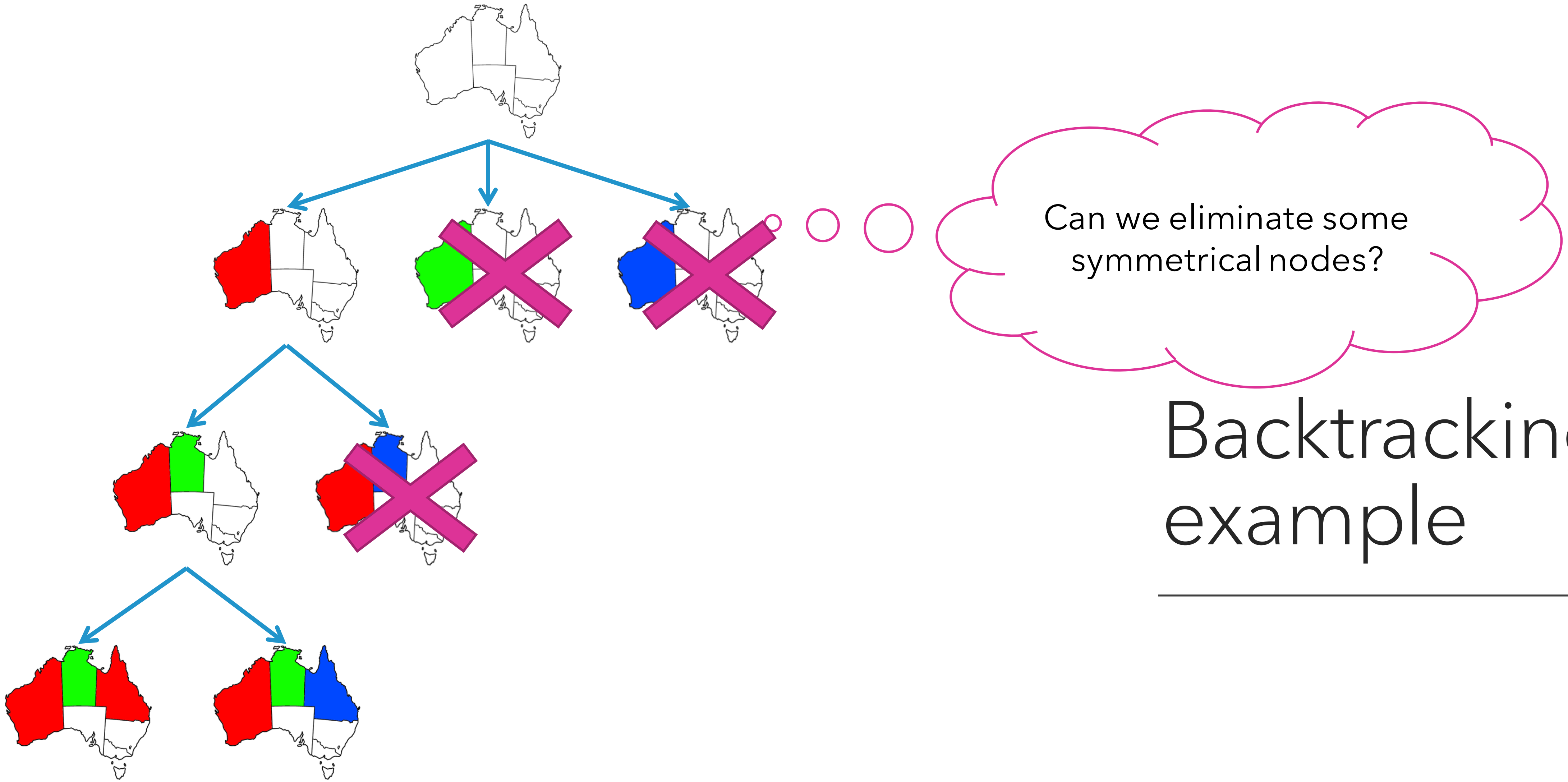


Backtracking example





Backtracking example



Backtracking example

Smart Search in CSPs

… or how to improve from backtracking

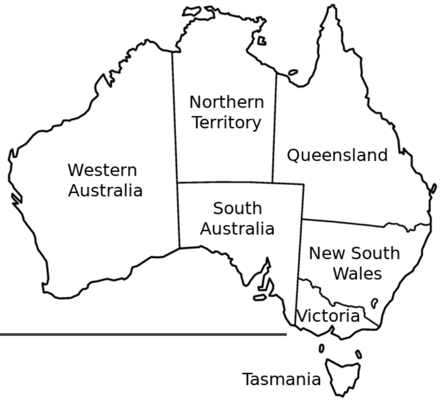
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

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

Improving backtracking efficiency

General-purpose methods can give huge gains in speed:

- Which variable should be assigned next?
  - **SELECT-UNASSIGNED-VARIABLE**
- In what order should its values be tried?
  - **ORDER-DOMAIN-VALUES**
- What inferences should be performed at each step of the search?
  - **INFERENCE**
- Can we detect inevitable failure early?

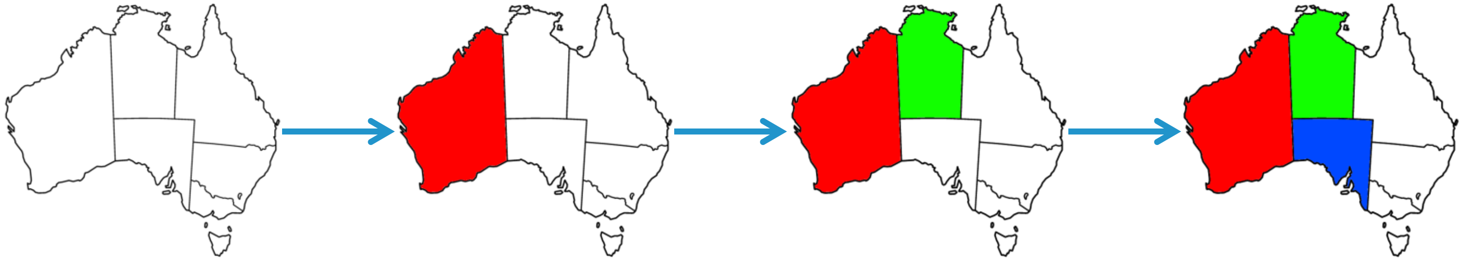
Most constrained variable



var <- SELECT-UNASSIGNED-VARIABLE(*csp*)

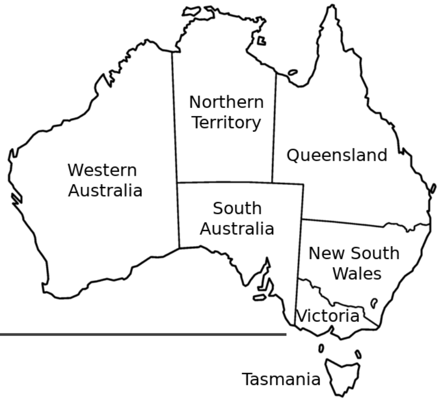
- Most constrained variable:
  - *choose the variable with the fewest legal values.*

a.k.a. *minimum-remaining-values* (MRV) heuristic.



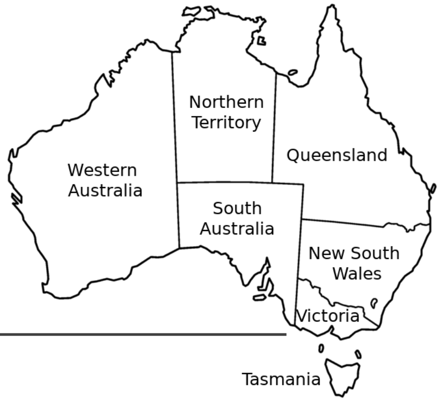
Variable Ordering

The Degree Heuristic



- Good to identify an initial state
- Tie-breaker among most constrained variables.
- Most constraining variable:
  - *choose the variable with the most constraints on remaining variables thus reducing branching.*

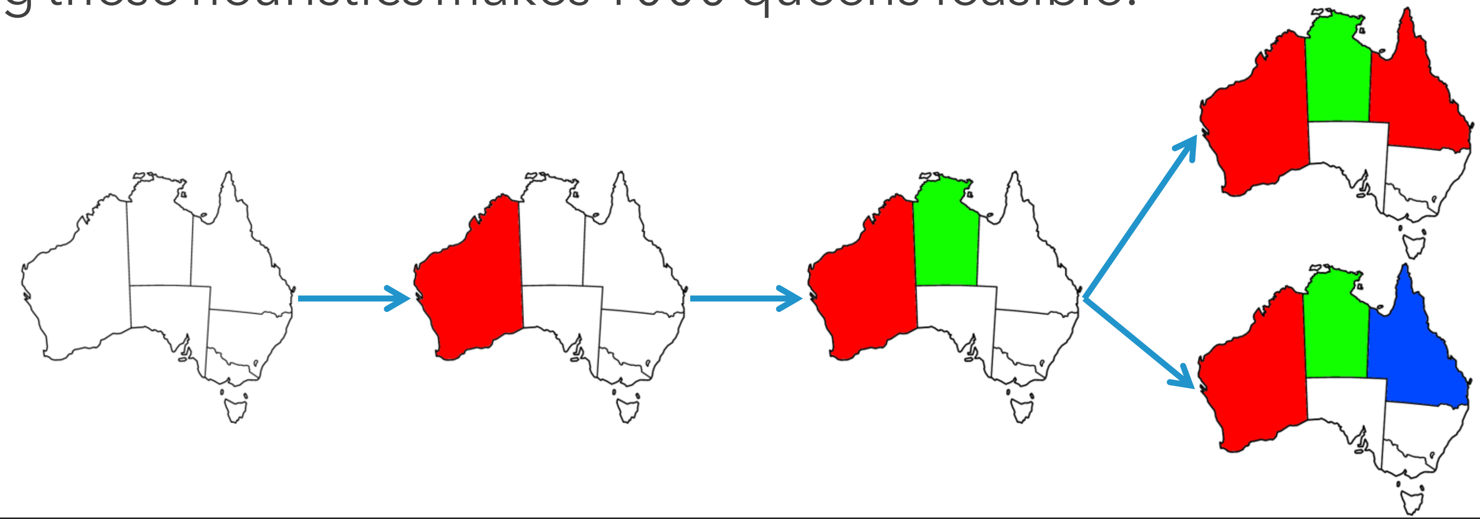
Variable Ordering → Value Ordering



Least constraining value

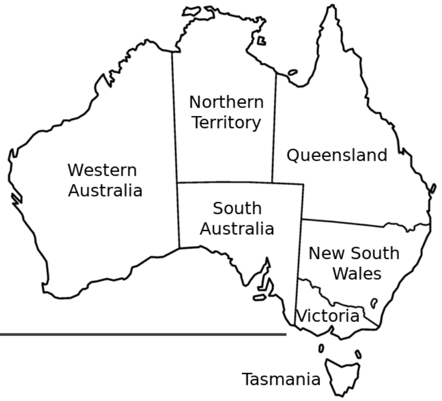
```
for value in ORDER-DOMAIN-VALUES(var, assignment, csp)
```

- Least constraining value:
  - *given a variable, choose the value that rules out the fewest values in the remaining variables.*
  - Combining these heuristics makes 1000 queens feasible!



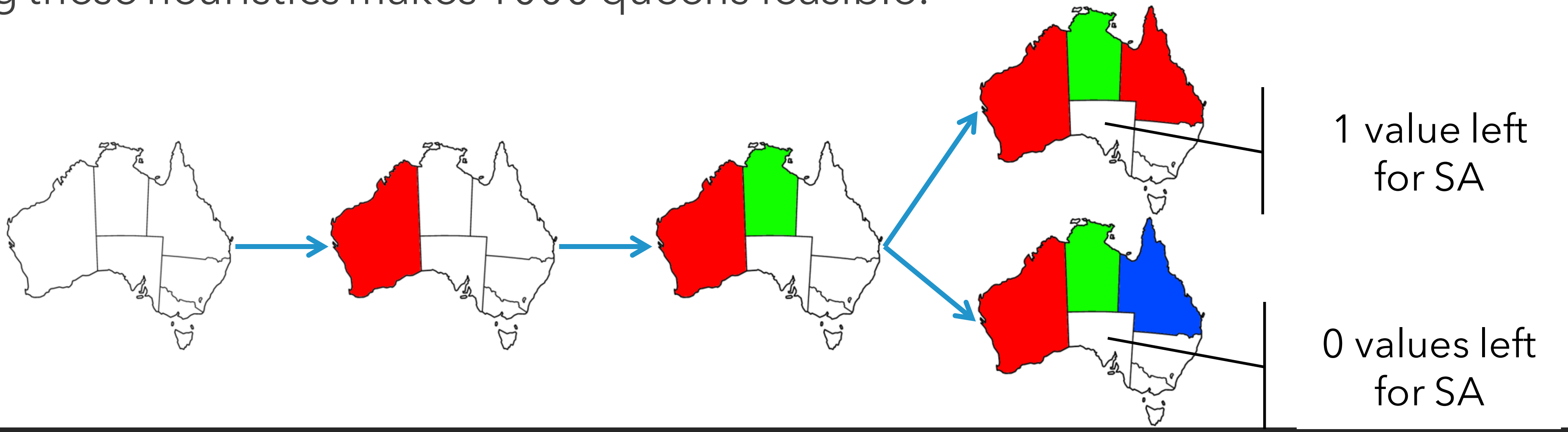
Variable Ordering → Value Ordering

Least constraining value

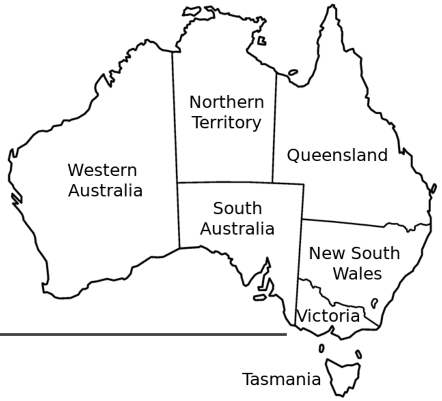



```
for value in ORDER-DOMAIN-VALUES(var, assignment, csp)
```

- Least constraining value:
  - *given a variable, choose the value that rules out the fewest values in the remaining variables.*
  - Combining these heuristics makes 1000 queens feasible!



Variable Ordering → Value Ordering → Inference



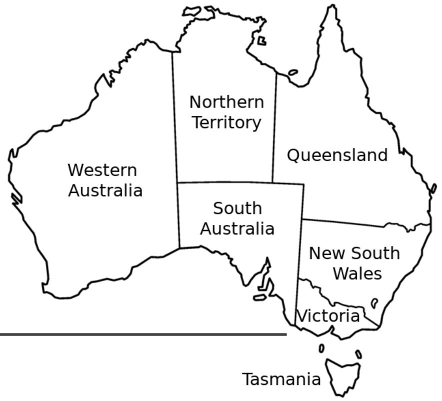
Forward checking

Idea:

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

WA	NT	Q	NSW	V	SA	T

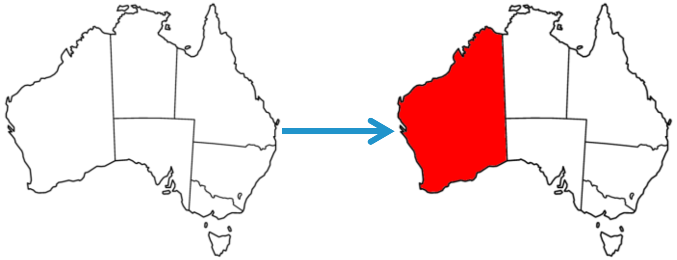
Variable Ordering → Value Ordering → Inference

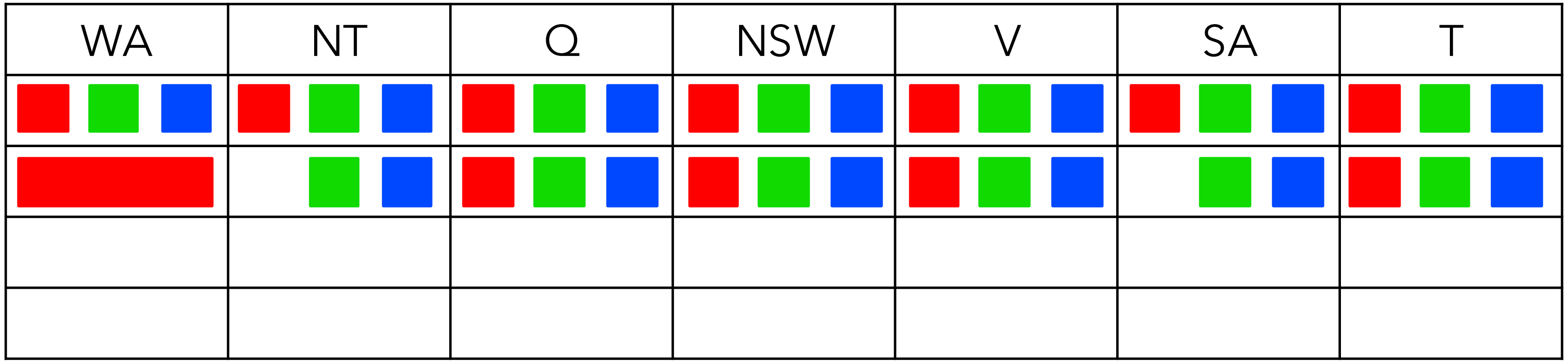


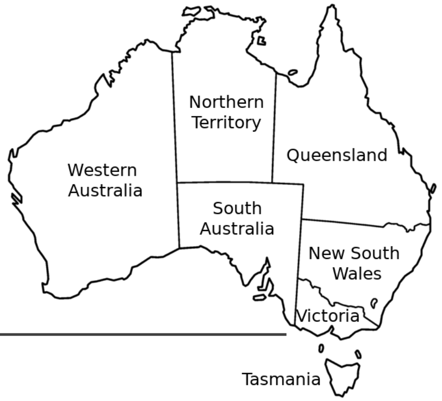
Forward checking

Idea:

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



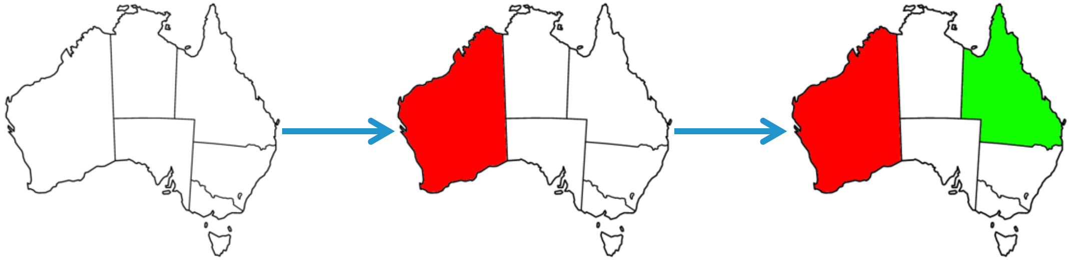


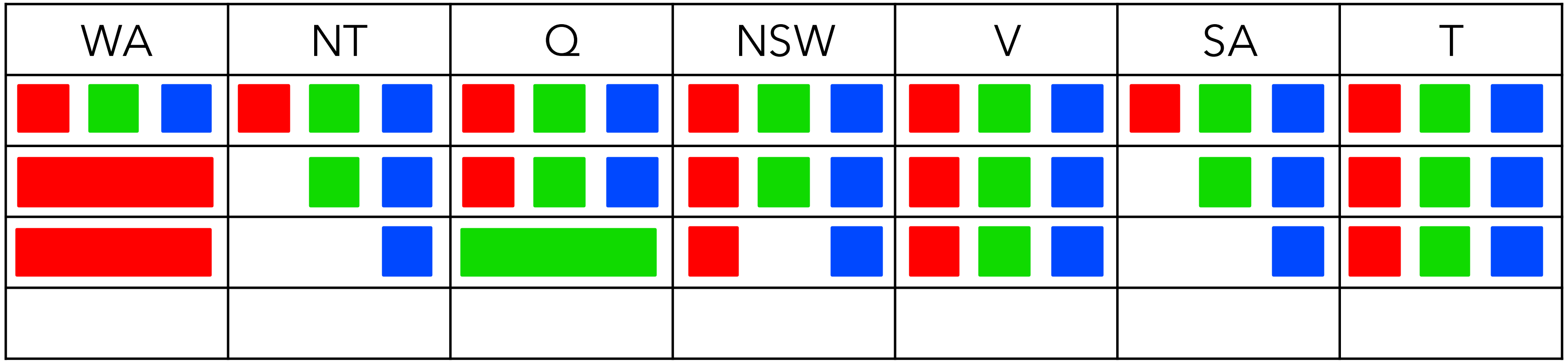


Forward checking

Idea:

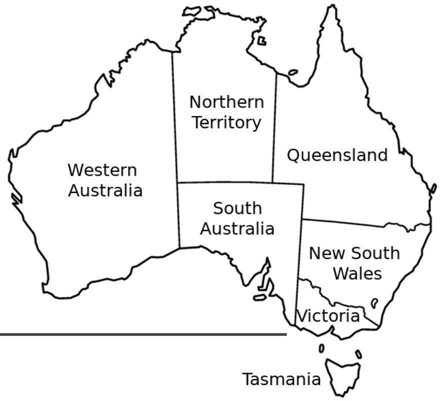
- Keep track of remaining legal values for unassigned variables.
- Terminate search when any variable has no legal values.





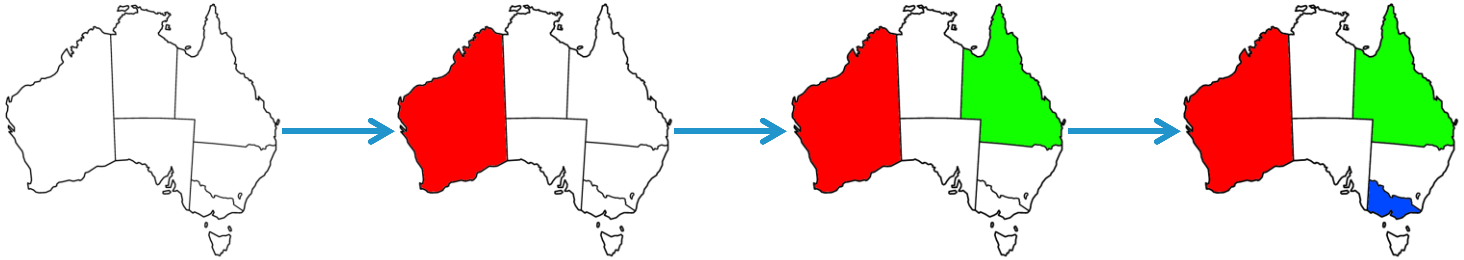
Variable Ordering → Value Ordering → Inference

Forward checking



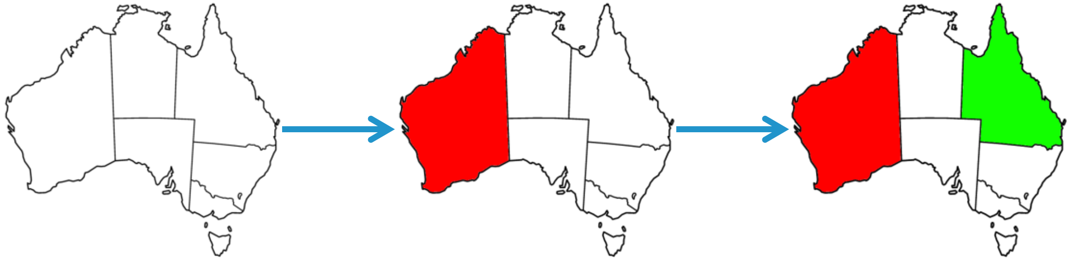
Idea:

- 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.

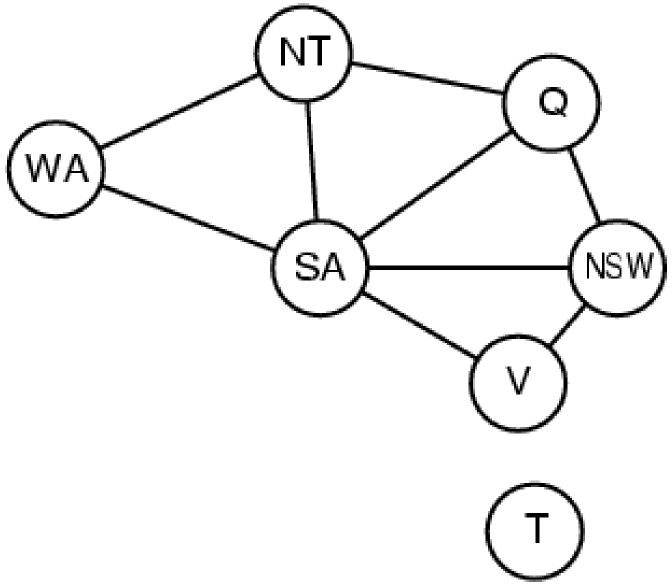
X →*Y* is consistent iff for every value *x* of in the domain of *X* there is some allowed *y in the domain of Y*.

Is there a value for X that makes the domain of Y empty?

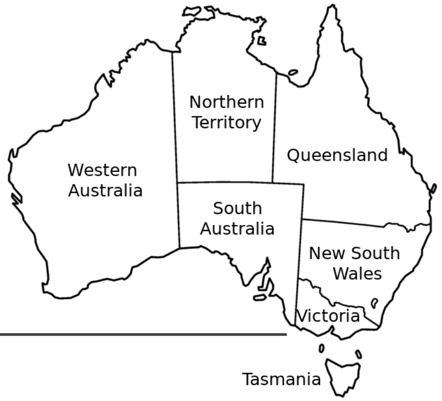
Can be run as a preprocessor or after each assignment.

Start with all directed arcs from the graph (18 here):

WA→NT, WA→SA, NT→WA, NT→SA, NT→Q, Q→NT, Q→SA, Q→NSW, SA→WA, SA→NT, SA→Q, SA→NSW, SA→V, NSW→Q, NSW→SA, NSW→V, V→SA, V→NSW



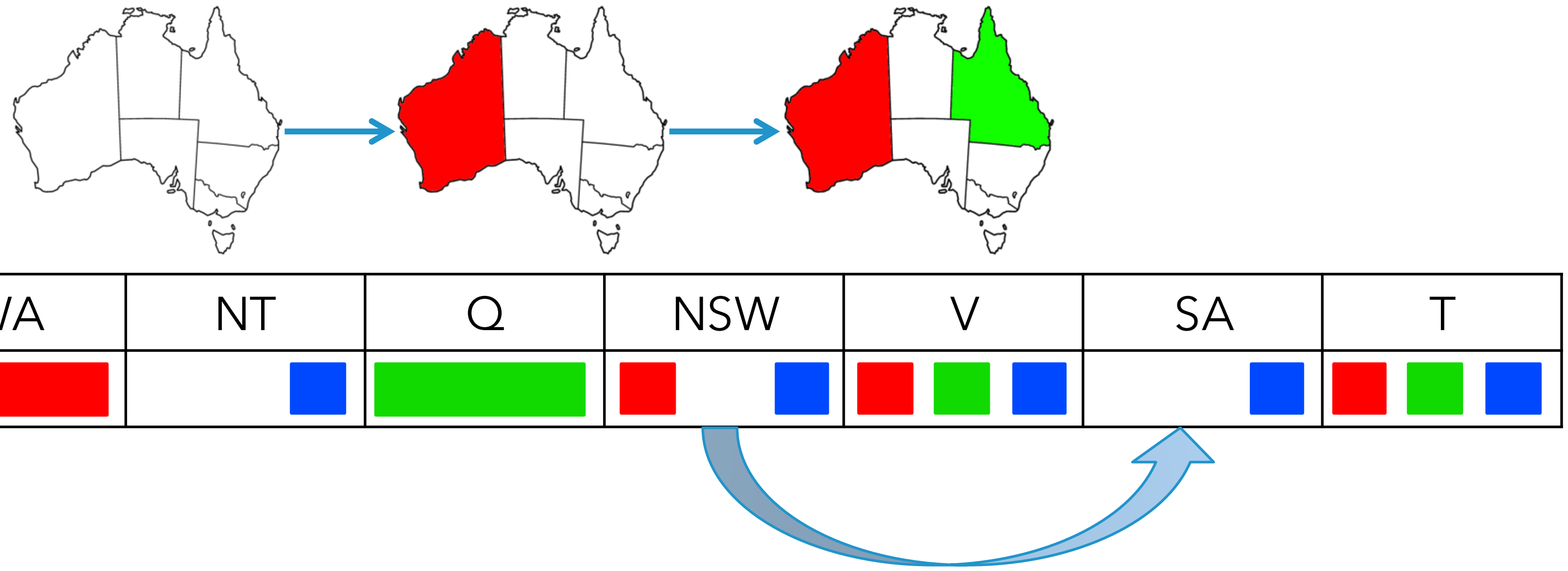
Variable Ordering → Value Ordering → Inference



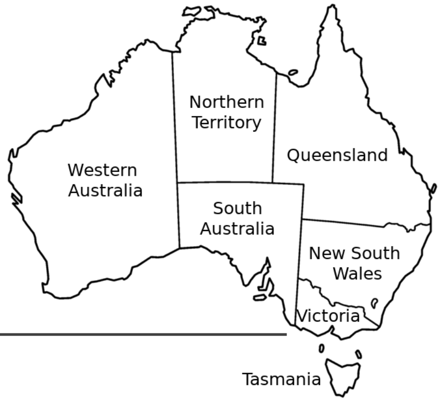
Arc consistency

X→Y : *Is there a value for X that makes the domain of Y empty?*

e.g. NSW → SA

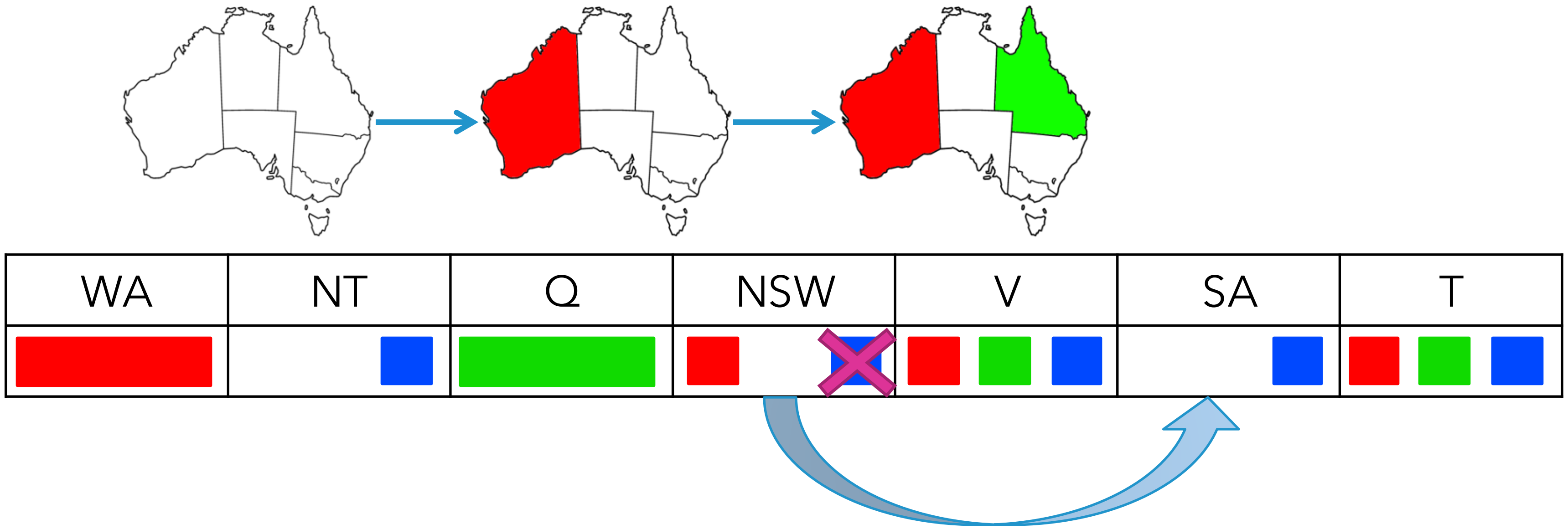


Arc consistency

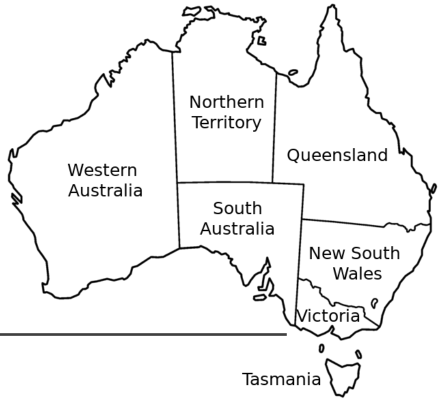


X→Y : *Is there a value for X that makes the domain of Y empty?*

e.g. NSW → SA



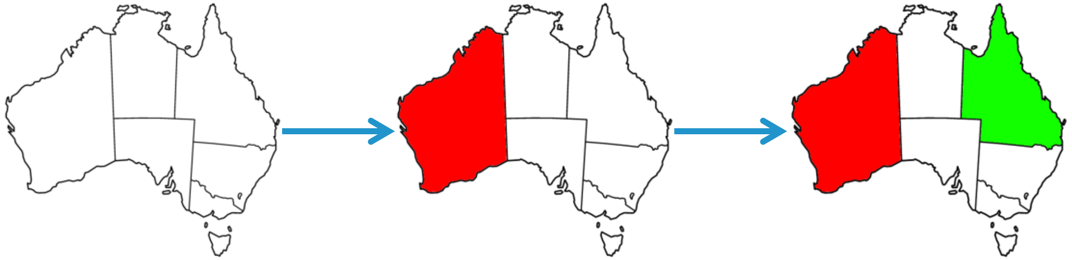
Arc consistency

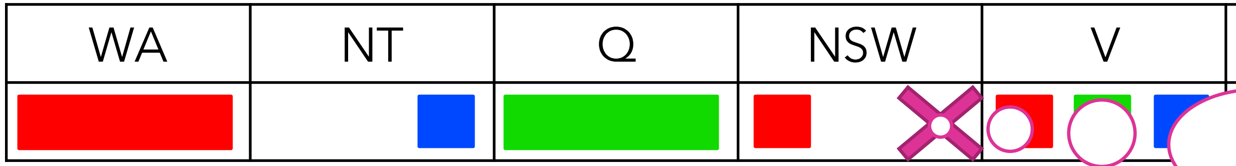


X→Y : *Is there a value for X that makes the domain of Y empty?*

e.g. NSW → SA

Once a value is removed, add all arcs pointing to X back in the queue!





Domain of NSW became smaller, so some arcs may have become *inconsistent!*

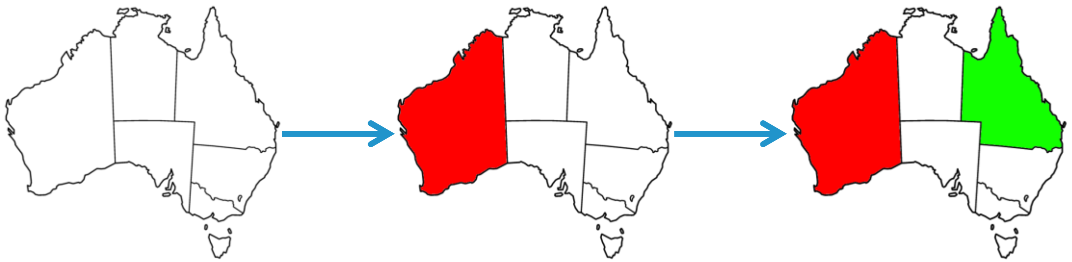
Arc consistency

Add:
V→NSW
SA→NSW
Q→NSW

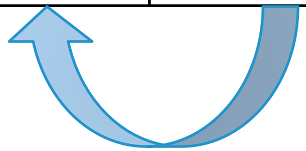
X→Y : *Is there a value for X that makes the domain*

e.g. NSW → SA

Once a value is removed, add all arcs pointing to X back in the queue!



WA	NT	Q	NSW	V	SA	T



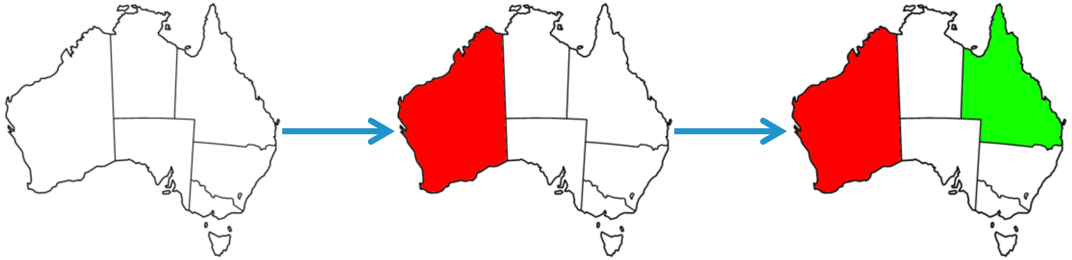
Arc consistency

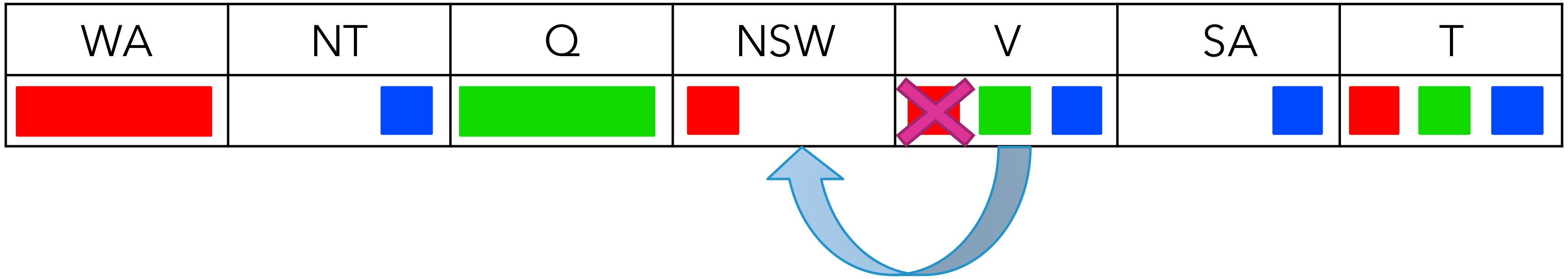
X→Y : *Is there a value for X that makes the domain*

Add:
V→NSW
SA→NSW
Q→NSW

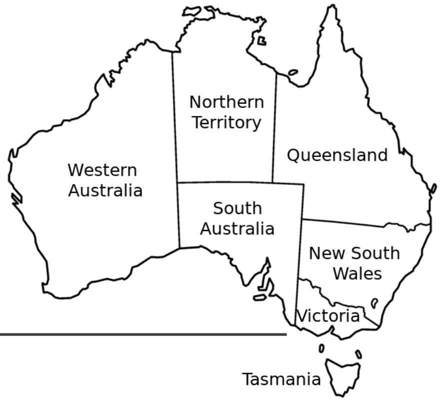
e.g. NSW → SA

Once a value is removed, add all arcs pointing to X back in the queue!





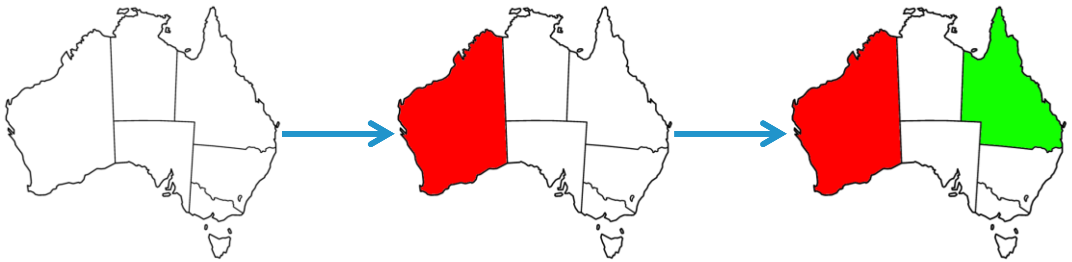
Variable Ordering → Value Ordering → Inference



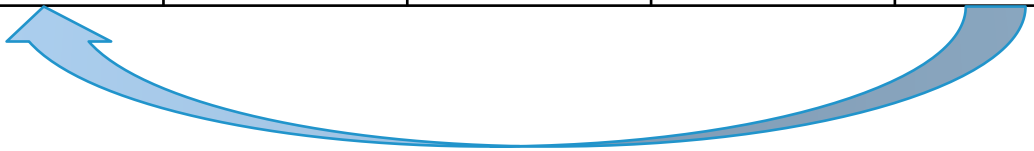
Arc consistency

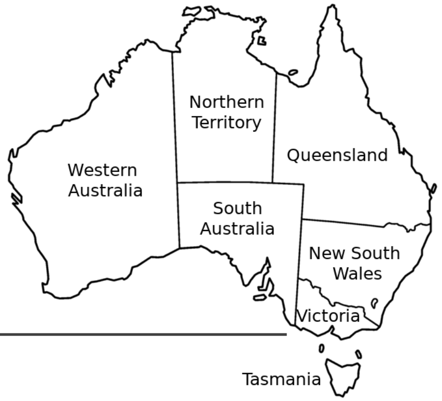
X→Y : *Is there a value for X that makes the domain of Y empty?*

Eventually check SA→NT



WA	NT	Q	NSW	V	SA	T

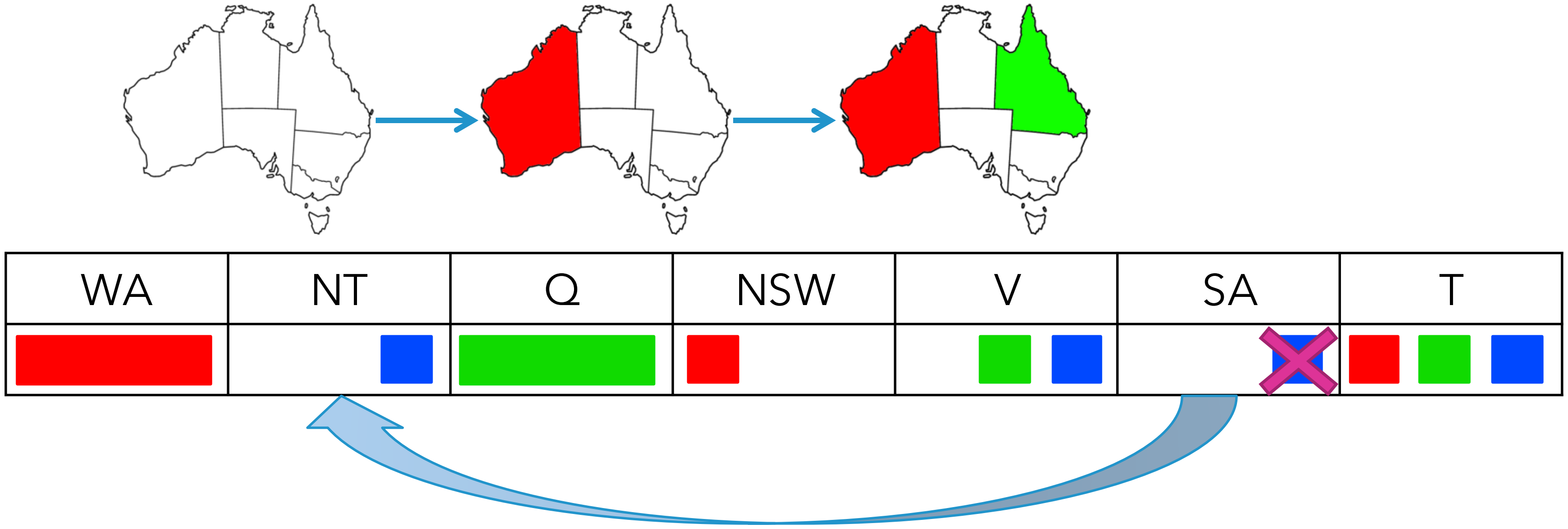




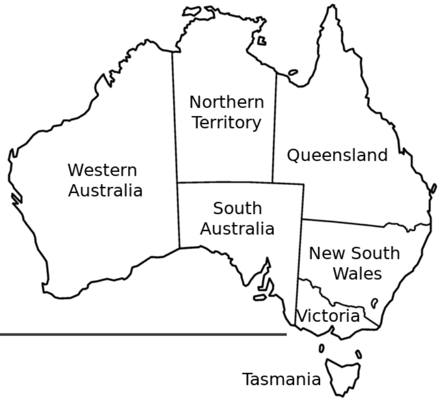
Arc consistency

X→Y : *Is there a value for X that makes the domain of Y empty?*

Eventually check SA→NT



Arc consistency



X→Y : Is there a value for X that makes the domain of Y empty?

Eventually check SA→NT

WA	NT	Q	NSW	V	SA	T
					Fail!	

Arc consistency detects failure earlier than forward checking.

Variable Ordering → Value Ordering → Inference

Arc consistency algorithm AC-3

function AC-3(csp) **returns** false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components $(X,\ D,\ C)$
  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 REVISE(csp, X_i, X_j) **then** ← ***Make Xi arc-consistent with respect to Xj***
      if size of $D_i = 0$ **then return** $false$ ← ***No consistent value left for Xi so fail***
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add $(X_k,\ X_i)$ to $queue$ ← ***Since revision occurred, add all neighbours of Xi for consideration (or reconsideration)***
  return $true$

---

function REVISE(csp, X_i, X_j) **returns** true iff we revise the domain of X_i
  $revised \leftarrow false$
  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
      $revised \leftarrow true$
  return $revised$

Time complexity: $O(cd^3)$, where:

- d is maximum size of each domain,
- c is the number of binary constraints (arcs).

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

Why?

- CSPs are prevalent in modern computation.
- Examples include: resource allocation, planning & scheduling, automated configuration, puzzles/games.
- More complex problem formulations exist: e.g., Distributed Constraint Optimisation Problems (DCOPs).
- Other solutions exist too: e.g., genetic algorithms, optimization