# CMSC 471

Constraint Satisfaction Problems

Instructor: KMA Solaiman

These slides were created by Dan Klein and Pieter Abbeel at UC Berkeley. [ai.berkeley.edu]

Constraint Satisfaction Problems

- Standard search problems:
  - State is a "black box": arbitrary data structure
  - Goal test can be any function over states
  - Successor function can also be anything

- Constraint satisfaction problems (CSPs):
  - A special subset of search problems
  - State is defined by variables X_i with values from a domain D (sometimes D depends on 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
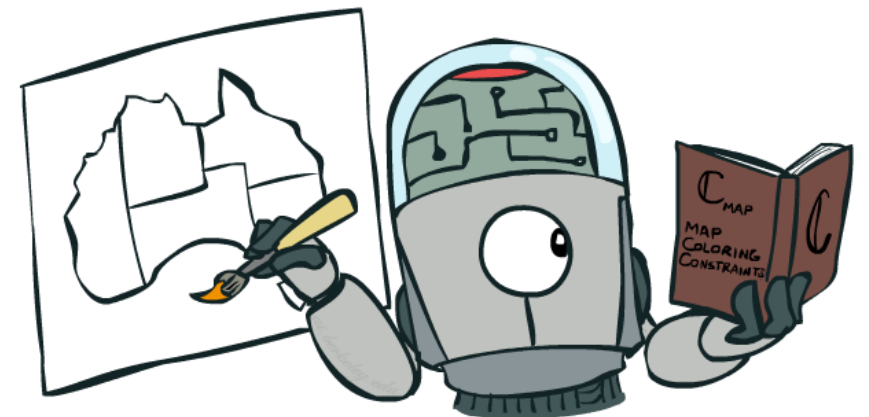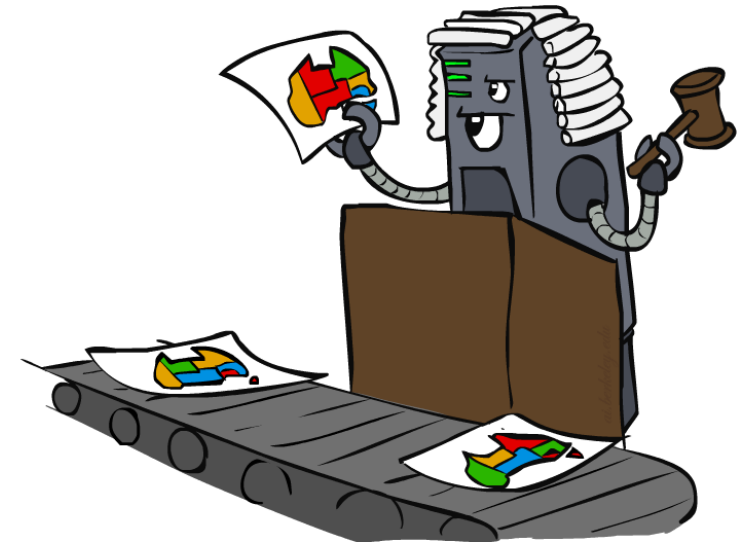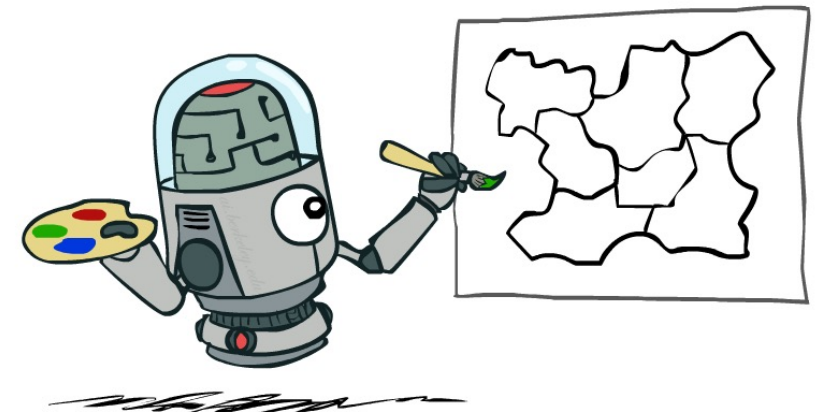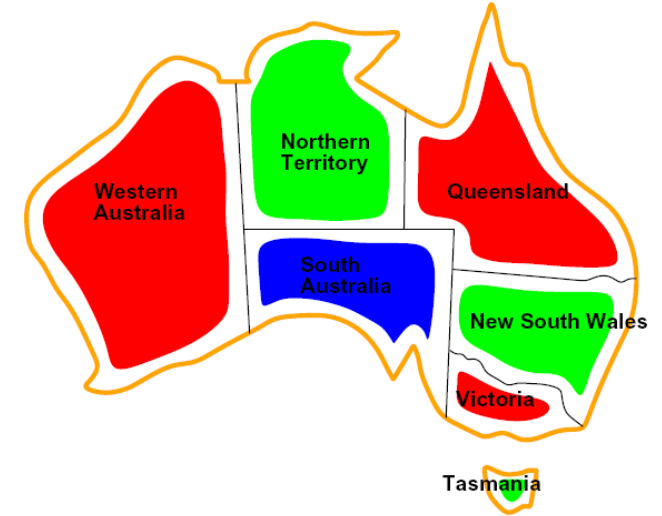
CSP Examples
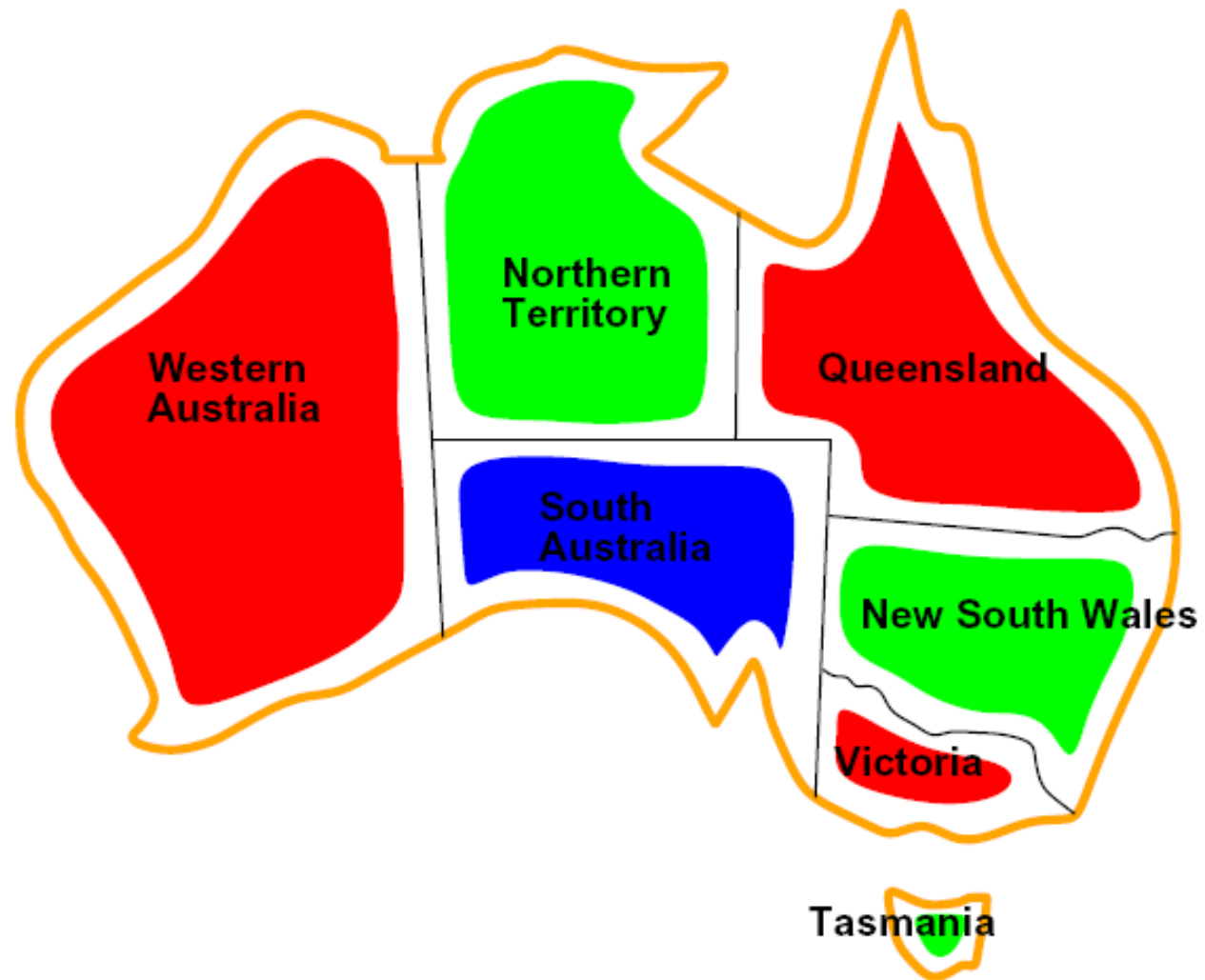
Example: Map Coloring

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

- Domains: $D = \{red, green, blue\}$

- Constraints: adjacent regions must have different colors

  Implicit: $WA \neq NT$

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

- Solutions are assignments satisfying all constraints, e.g.:

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

Solving CSPs

Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs

- Idea 1: One variable at a time
  - Variable assignments are commutative, so fix ordering
  - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
  - Only need to consider assignments to a single variable at each step

- Idea 2: Check constraints as you go
  - I.e. consider only values which do not conflict previous assignments
  - Might have to do some computation to check the constraint
  - "Incremental goal test"

- Depth-first search with these two improvements is called *backtracking search* (not the best name)
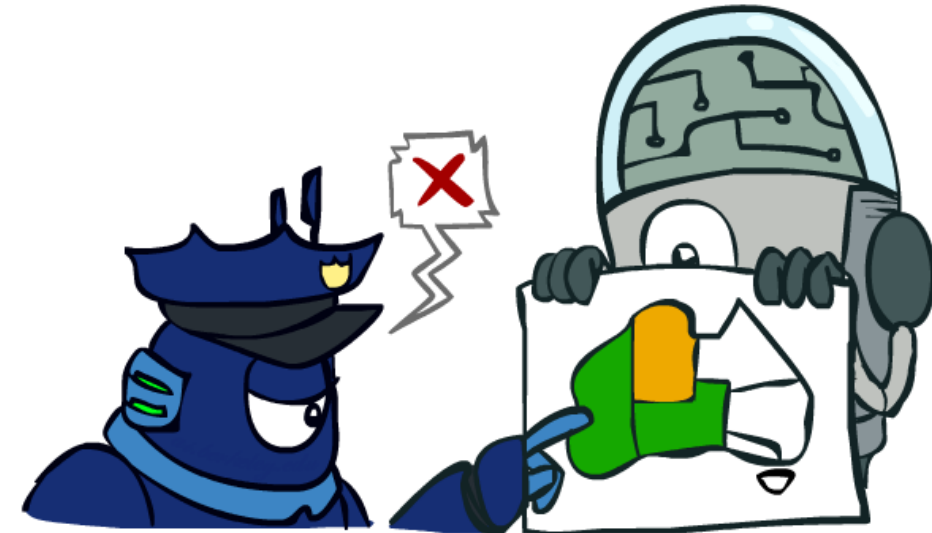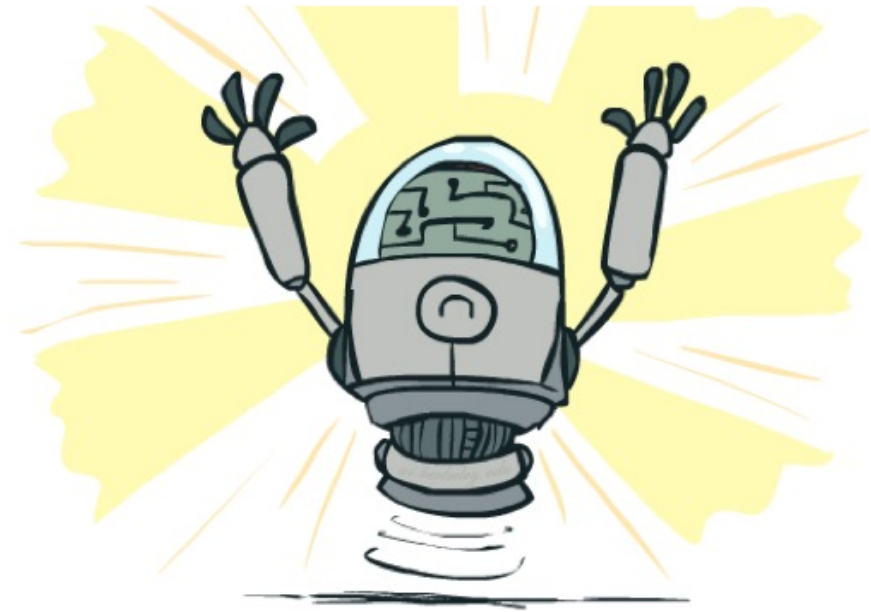
Backtracking Example
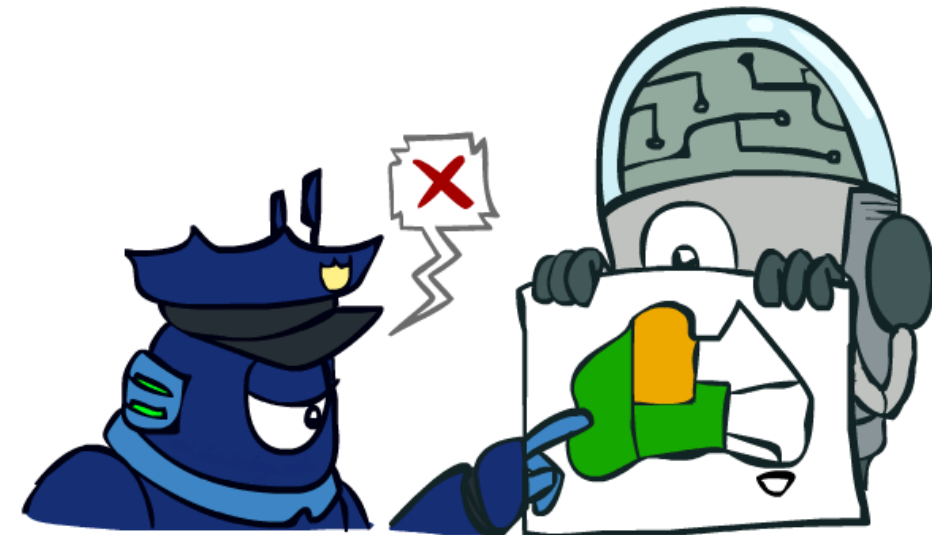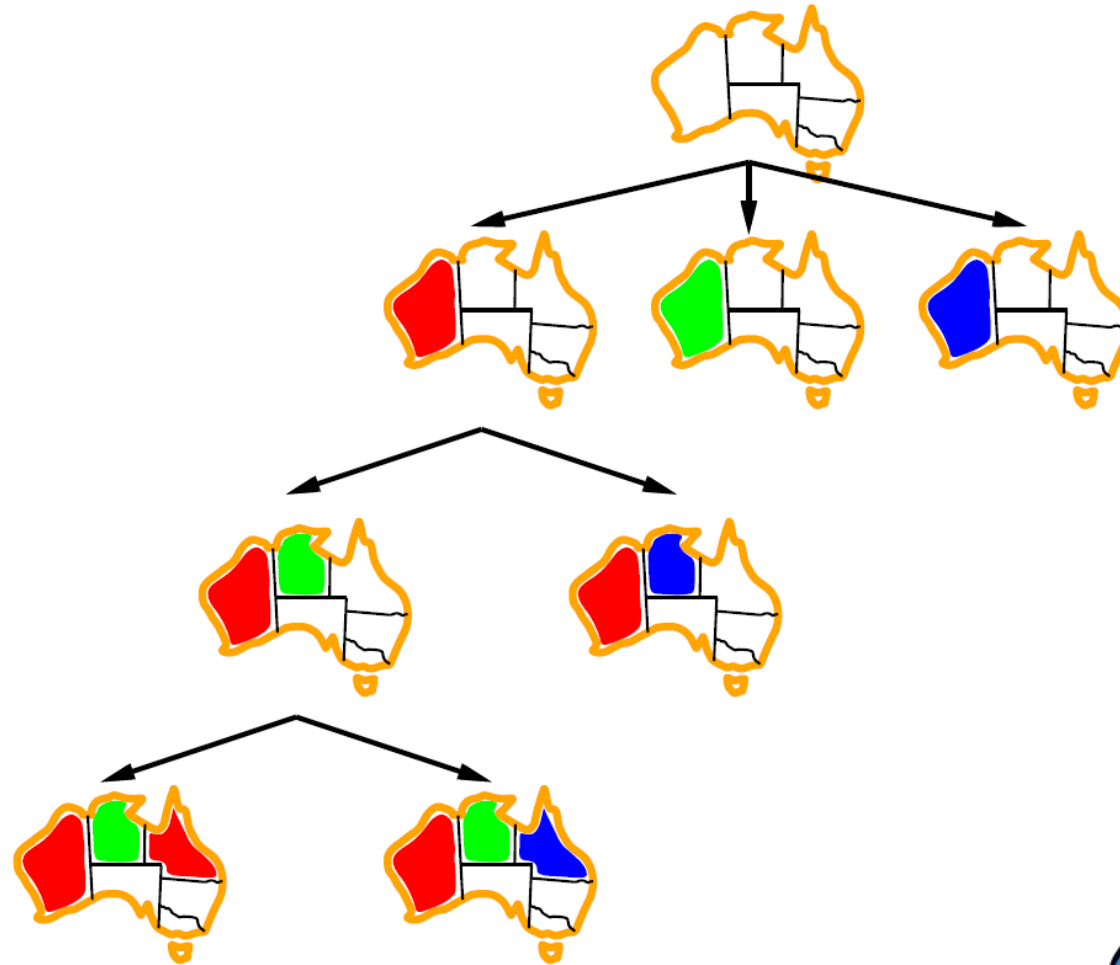
Improving Backtracking

- General-purpose ideas give huge gains in speed

- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?

- Filtering: Can we detect inevitable failure early?

- Structure: Can we exploit the problem structure?
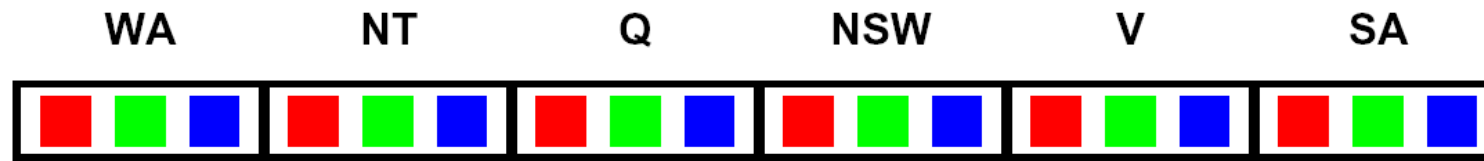
Filtering

Filtering

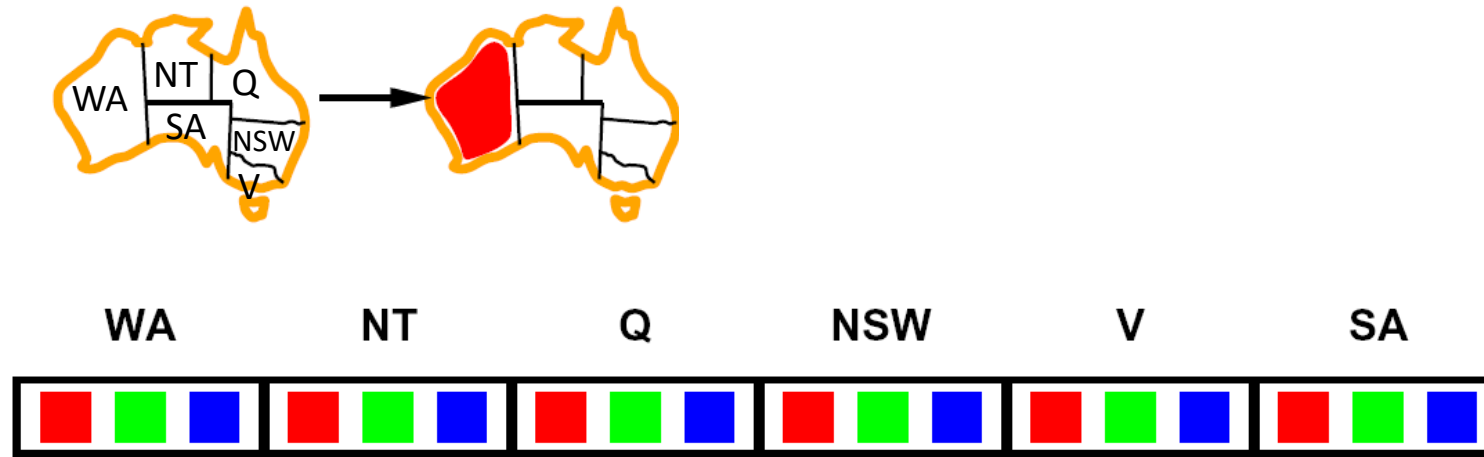Keeping track of domains for unassigned variables and cross off bad options
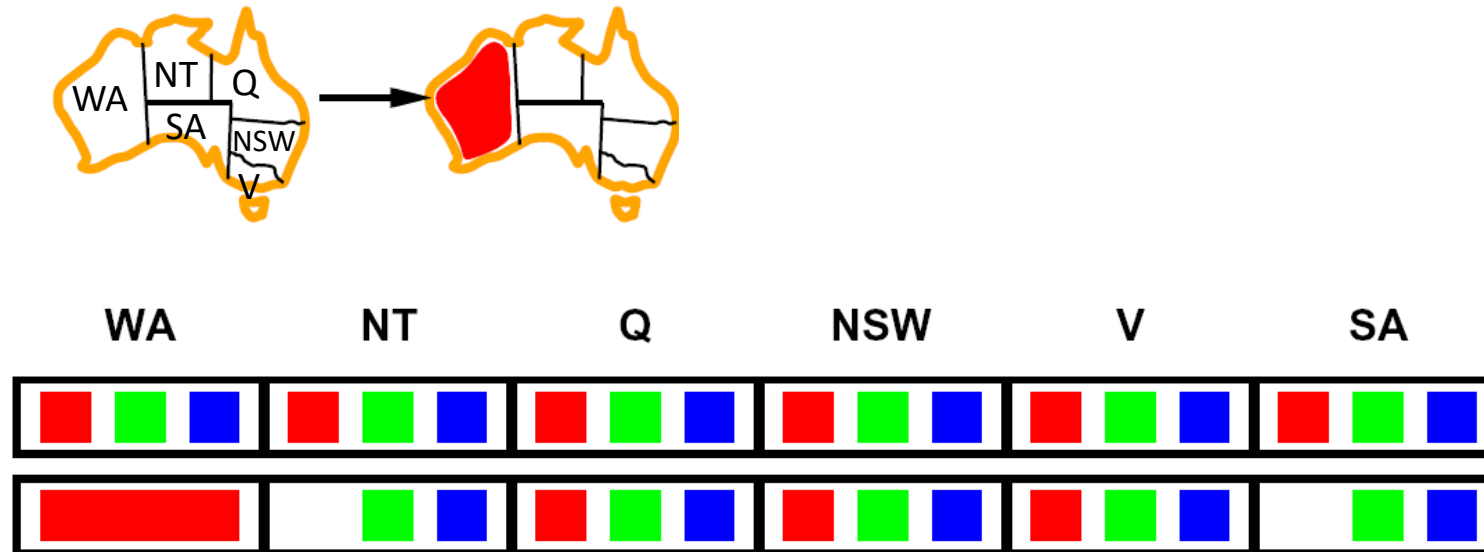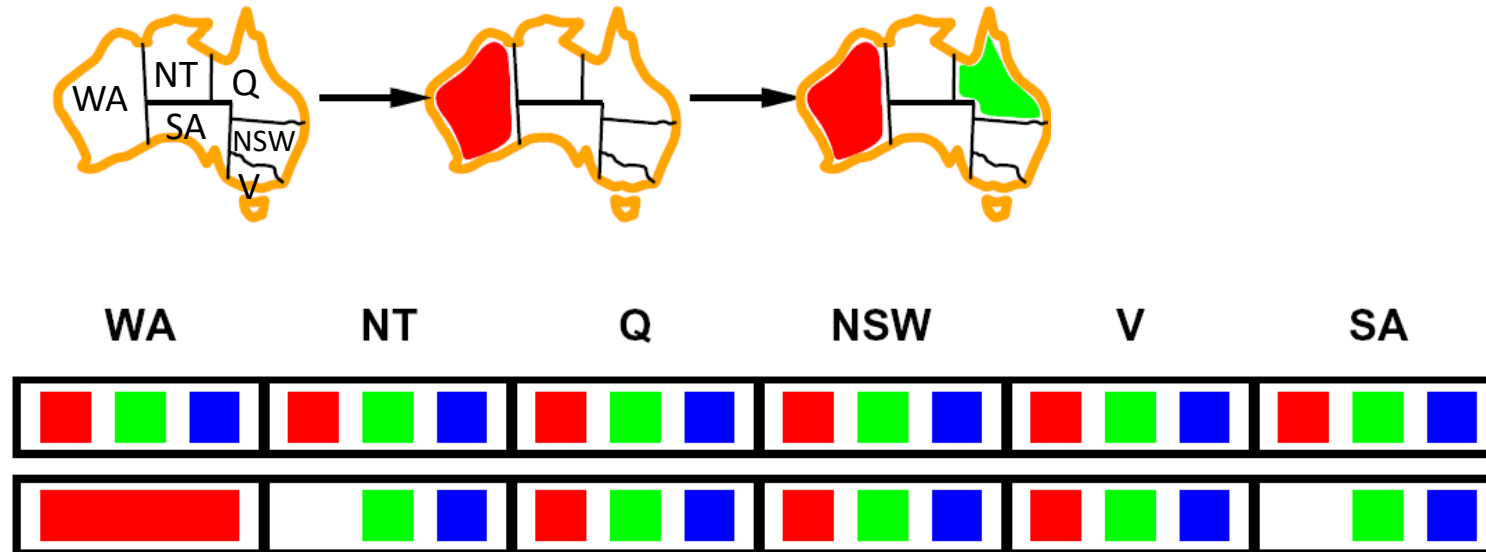
Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment
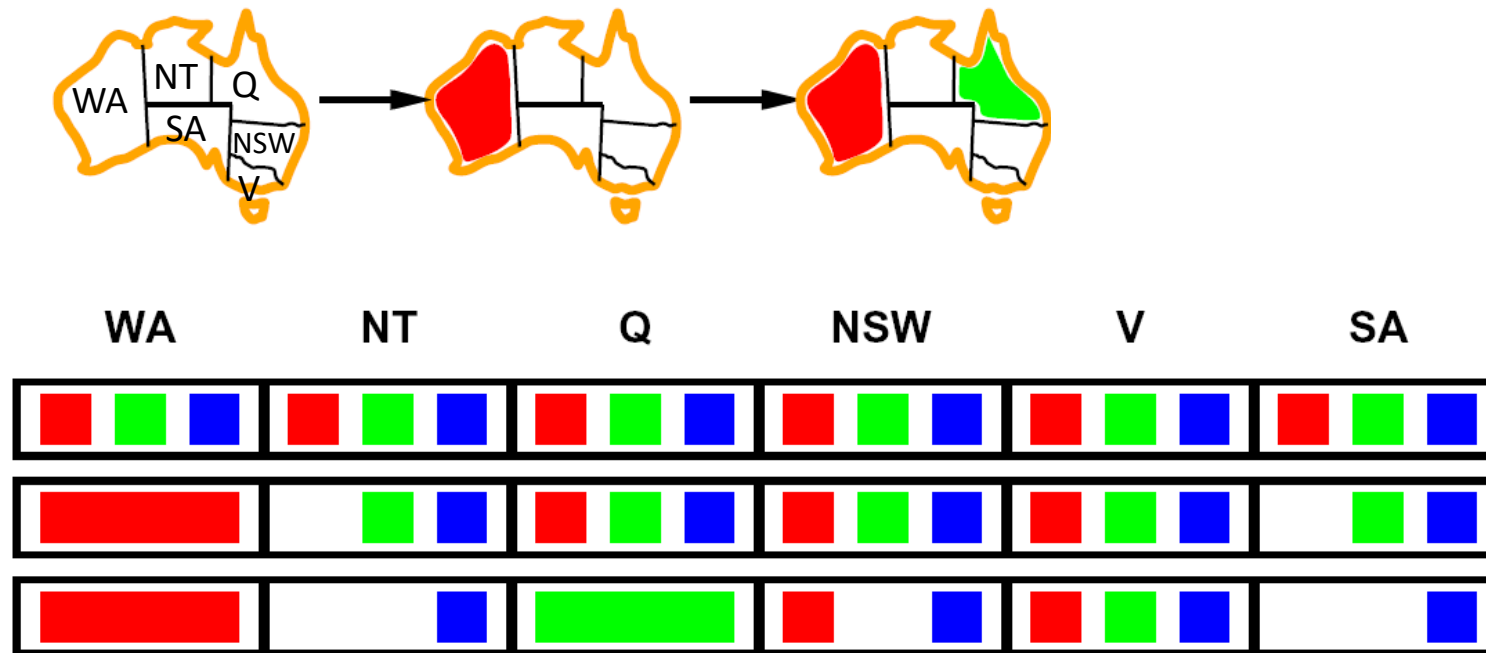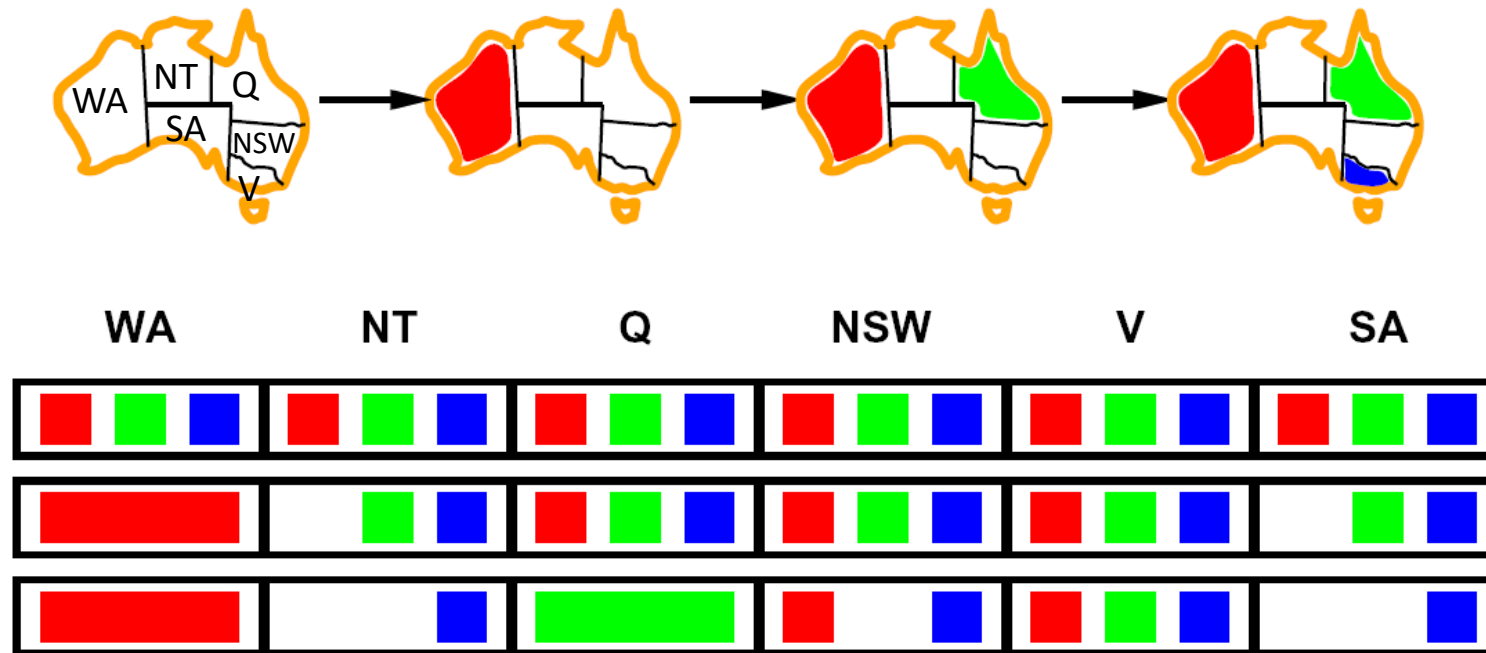
Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment

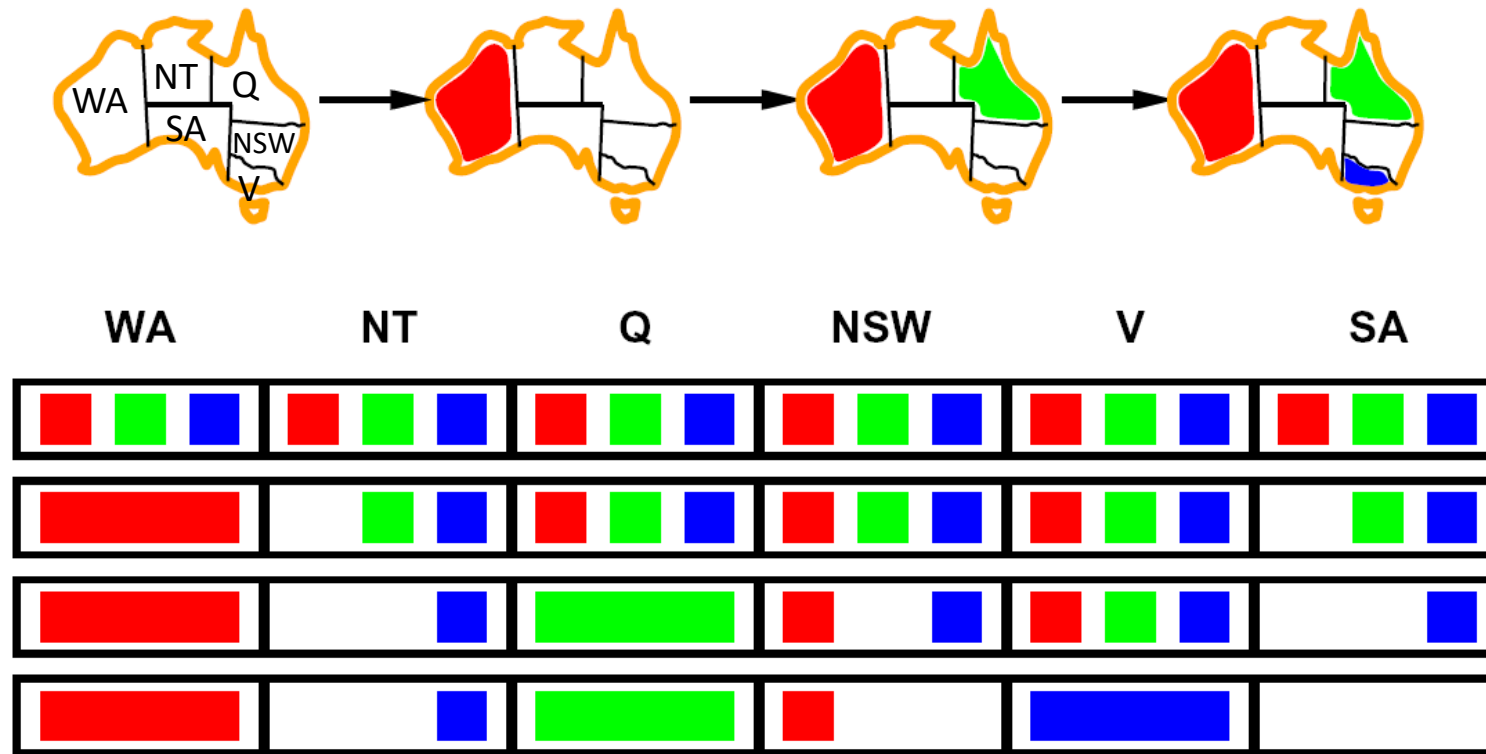| WA	NT	Q	NSW	V	SA
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦

Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options

- Forward checking: Cross off values that violate a constraint when added to the existing assignment

	WA	NT	Q	NSW	V	SA

Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment
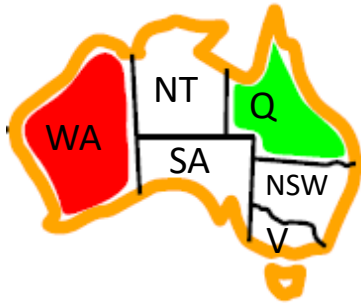
Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment

Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment

Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment

WA	NT	Q	NSW	V	SA

Video of Demo Coloring – Backtracking with Forward Checking

Filtering: Constraint Propagation

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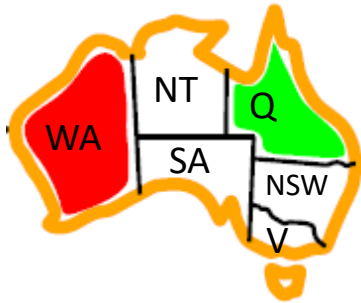
Filtering: Constraint Propagation

- 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!

Filtering: Constraint Propagation

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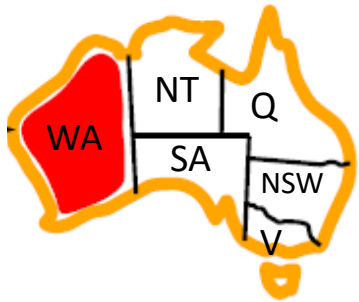

- NT and SA cannot both be blue!
- Why didn't we detect this yet?

Filtering: 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!
- Why didn't we detect this yet?
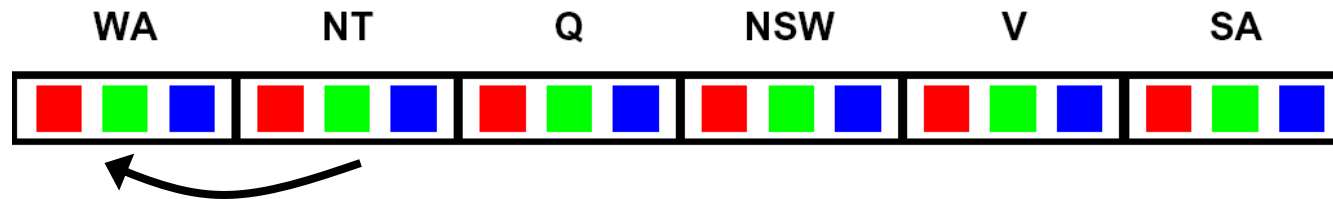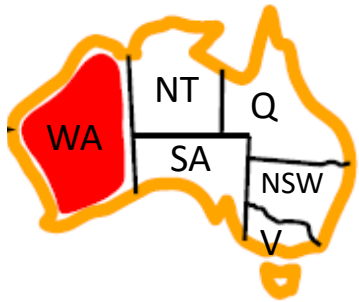- *Constraint propagation:* reason from constraint to constraint

Consistency of A Single Arc

- An arc X → Y is consistent iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint

Consistency of A Single Arc

- An arc X → Y is consistent iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint
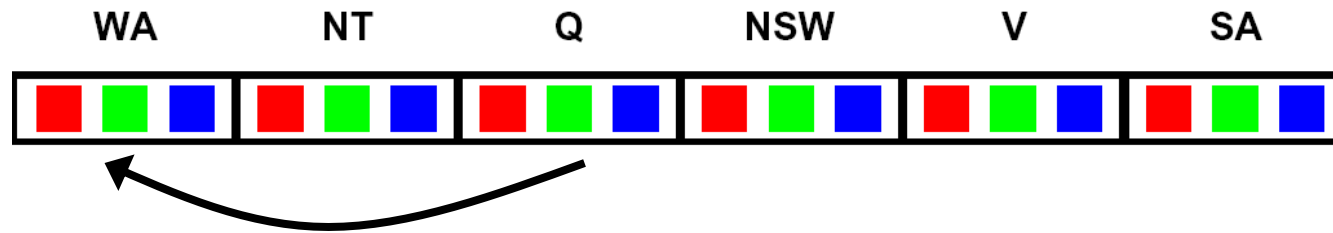
Consistency of A Single Arc

- An arc X → Y is consistent iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint
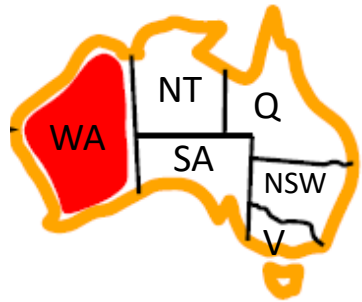
Consistency of A Single Arc

- An arc X → Y is consistent iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint
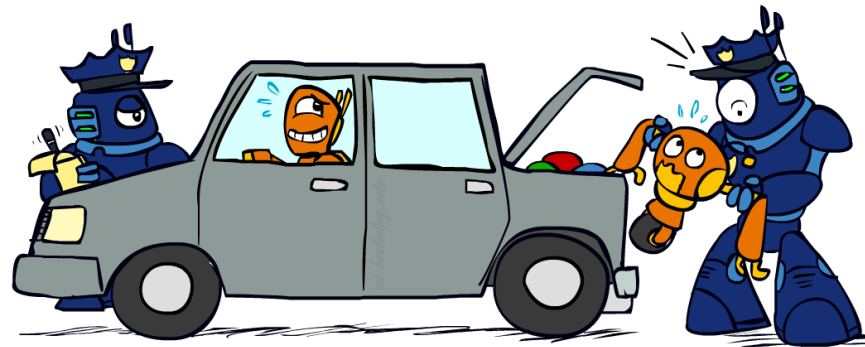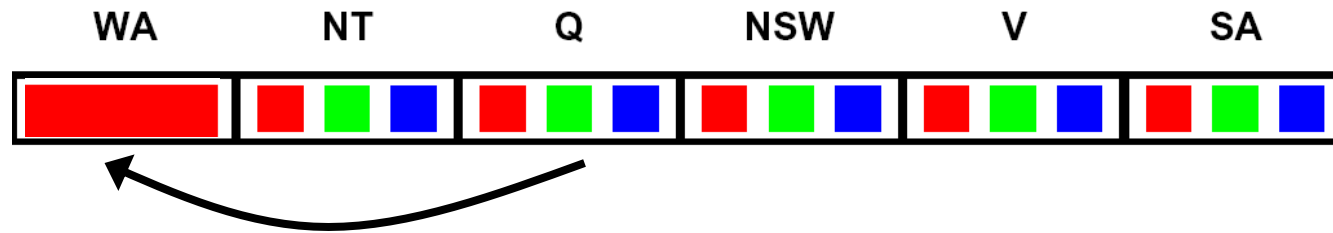
Consistency of A Single Arc

- An arc X → Y is consistent iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint

Delete from the tail!

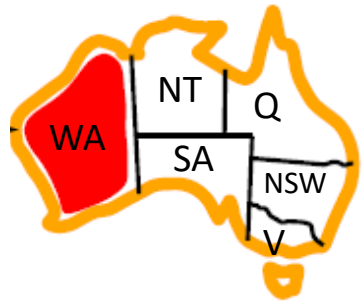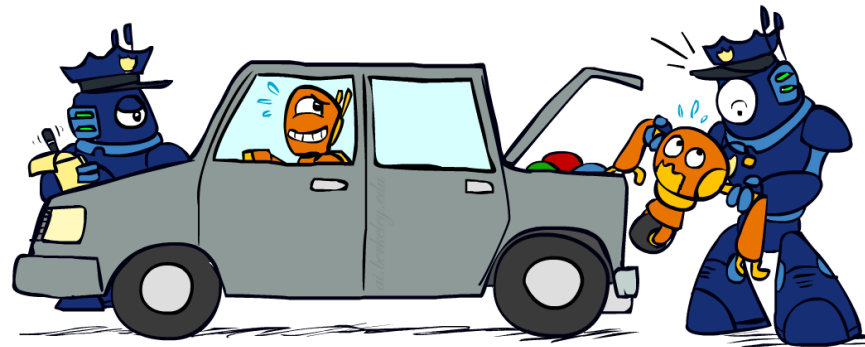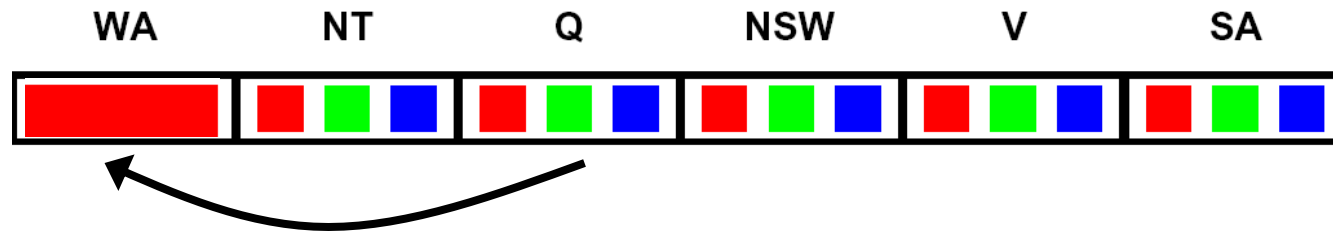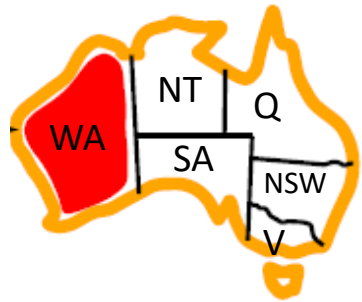# Consistency of A Single Arc
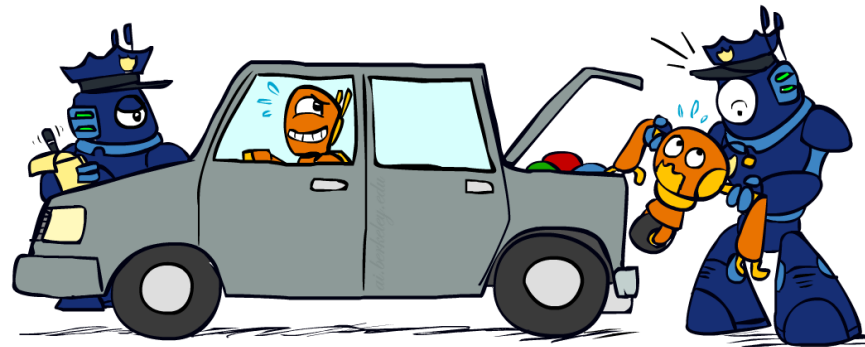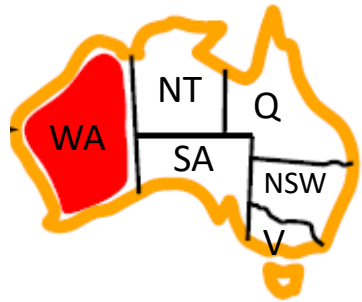
- An arc X → Y is consistent iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint

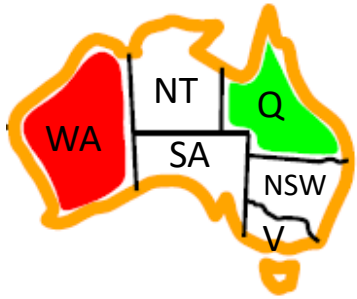
Delete from the tail!

- Forward checking: Enforcing consistency of arcs pointing to each new assignment

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure **all** arcs are consistent:

Remember: Delete from the tail!

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure **all** arcs are consistent:

Remember: Delete from the tail!

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure **all** arcs are consistent:

Remember: Delete from the tail!

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure **all** arcs are consistent:

Remember: Delete from the tail!
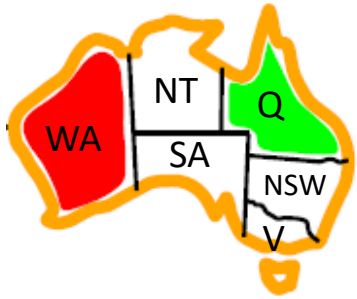
Arc Consistency of an Entire CSP

- A simple form of propagation makes sure **all** arcs are consistent:

Remember: Delete from the tail!
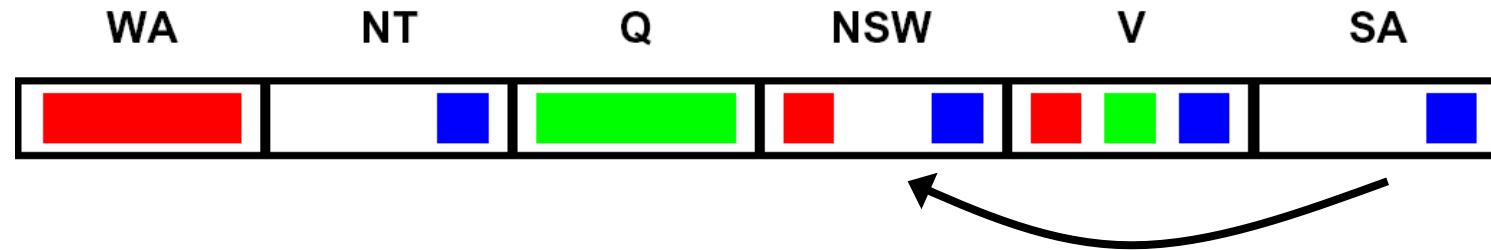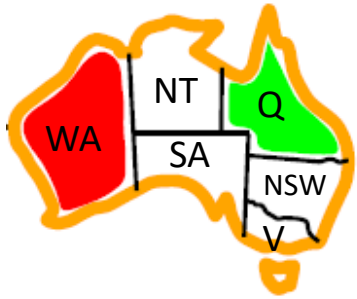
Arc Consistency of an Entire CSP

- A simple form of propagation makes sure **all** arcs are consistent:

Remember: Delete from the tail!

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure **all** arcs are consistent:

| WA | NT | Q | NSW | V | SA |

Remember: Delete from the tail!

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure **all** arcs are consistent:

Remember: Delete from the tail!
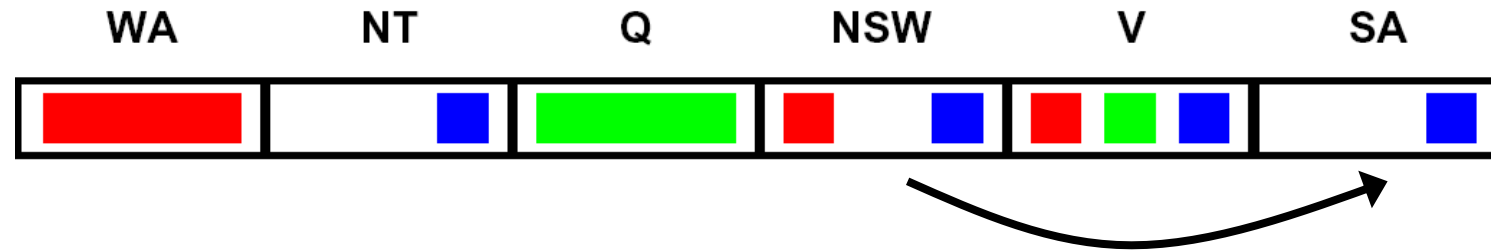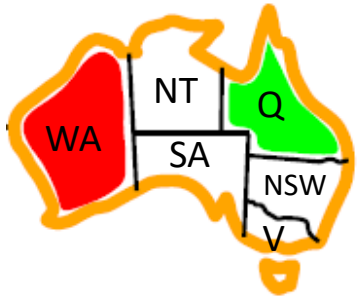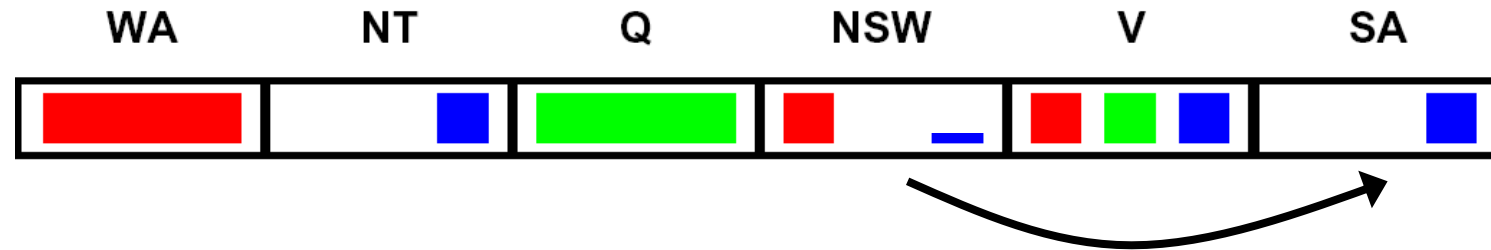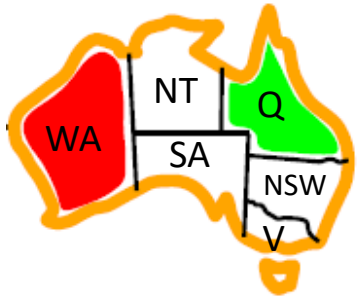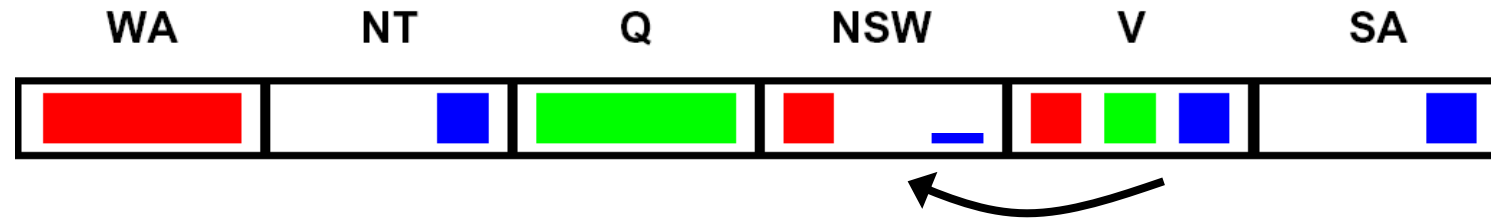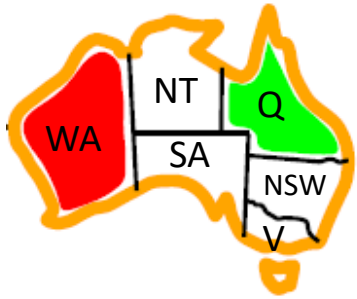
Arc Consistency of an Entire CSP

- A simple form of propagation makes sure **all** arcs are consistent:

- Important: 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
- What's the downside of enforcing arc consistency?

Remember: Delete from the tail!

Enforcing Arc Consistency in a CSP

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*

- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- … but detecting all possible future problems is NP-hard – why?

# Video of Demo Coloring – Backtracking with Forward Checking – Complex Graph

Video of Demo Coloring – Backtracking with Arc Consistency – Complex Graph

Limitations of Arc Consistency

■ After enforcing arc consistency:

- Can have one solution left
- Can have multiple solutions left
- Can have no solutions left (and not know it)

What went wrong here?

Limitations of Arc Consistency

- After enforcing arc consistency:
    - Can have one solution left
    - Can have multiple solutions left
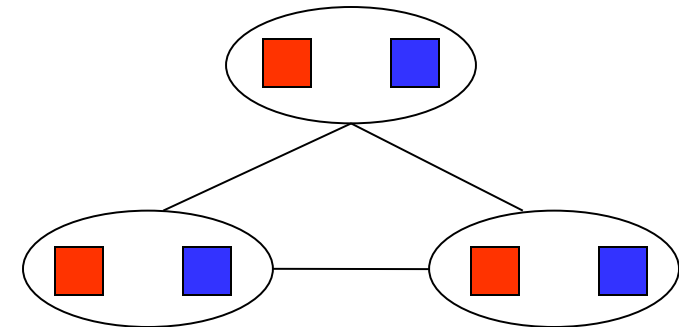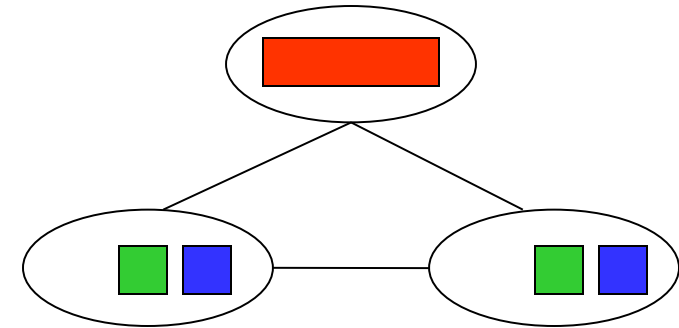   - Can have no solutions left (and not know it)

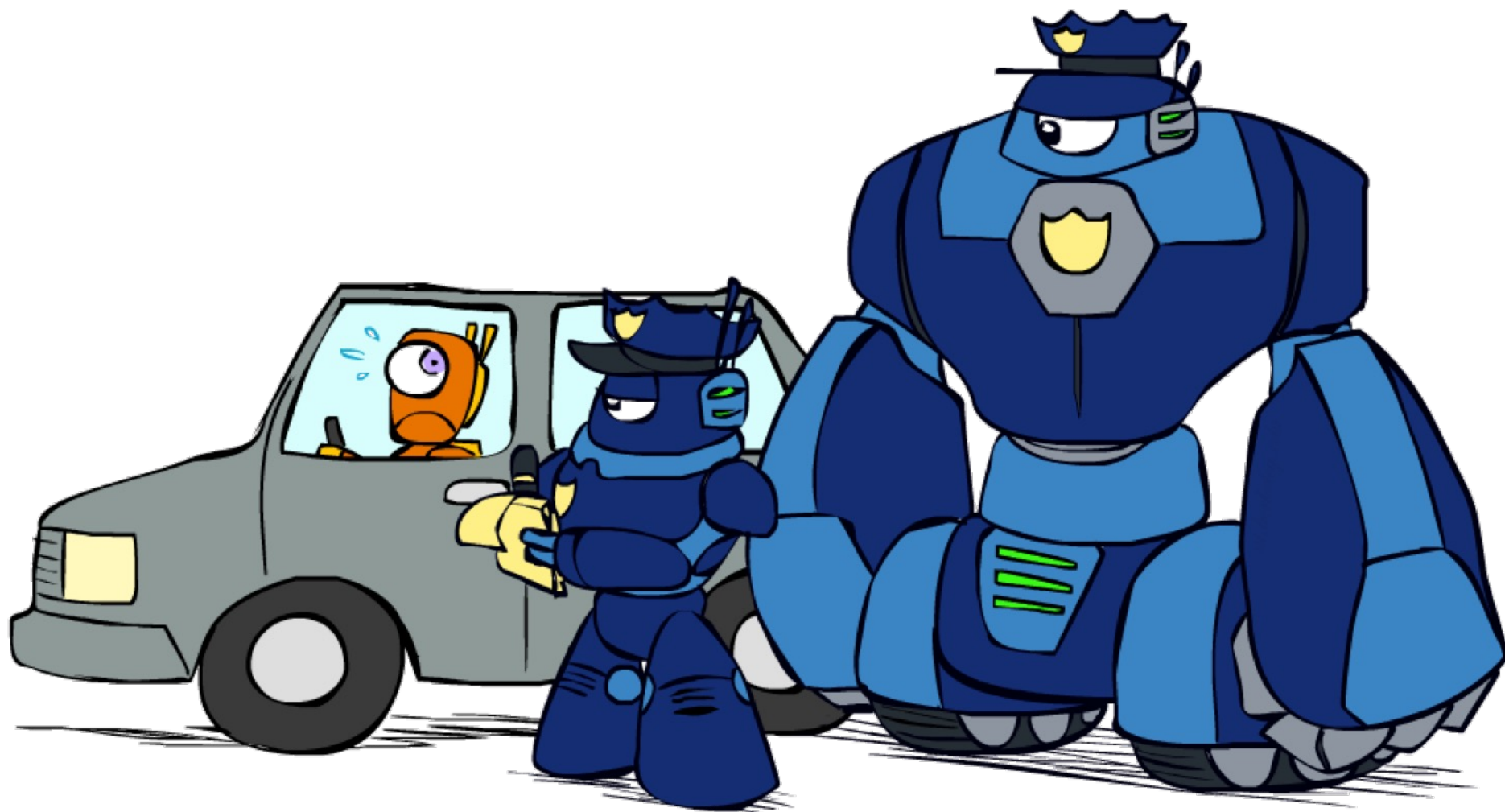- Arc consistency still runs inside a backtracking search!
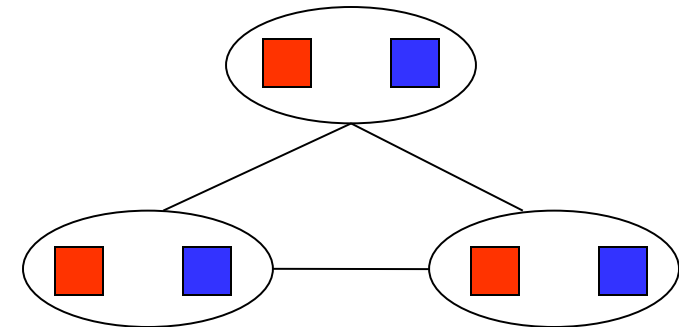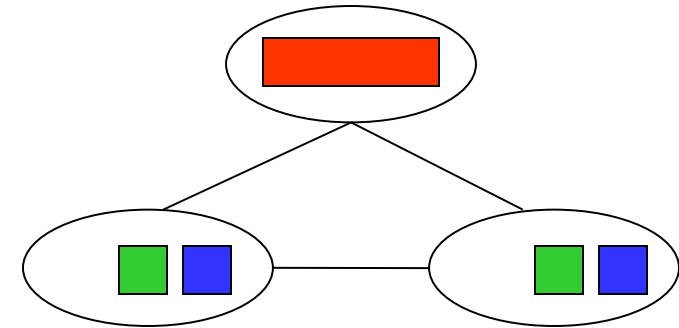
What went wrong here?

[Demo: coloring -- forward checking]

[Demo: coloring -- arc consistency]

K-Consistency

K-Consistency

- **Increasing degrees of consistency**

  - 1-Consistency (Node Consistency): Each single node's domain has a value which 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 kth node.

- **Higher k more expensive to compute**

- **(You need to know the k=2 case: arc consistency)**
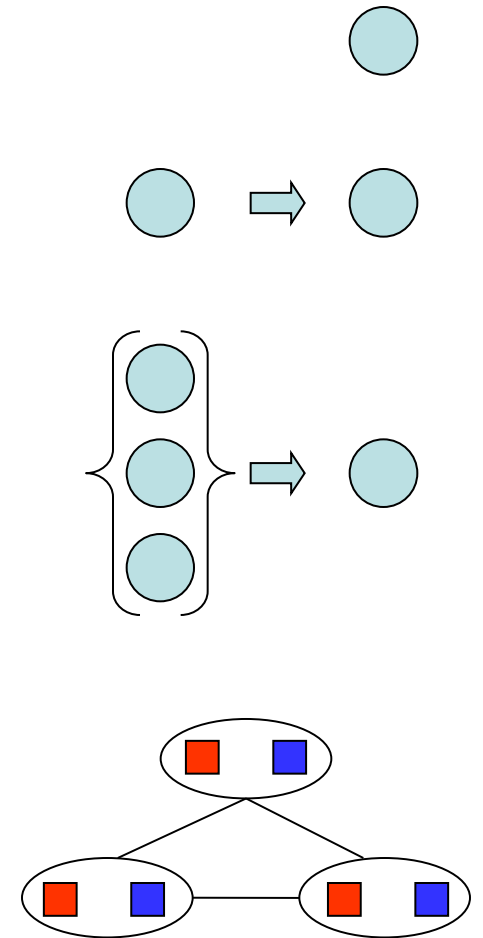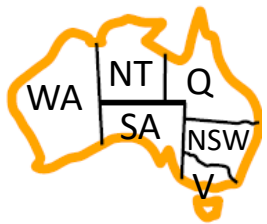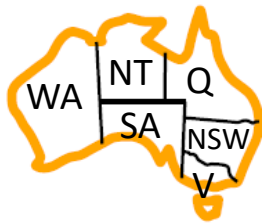
Ordering

Ordering: Minimum Remaining Values

- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain

Ordering: Minimum Remaining Values

- Variable Ordering: Minimum remaining values (MRV):
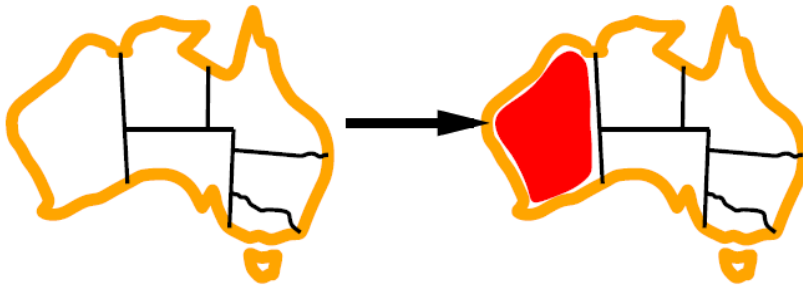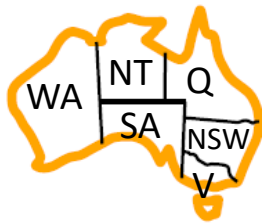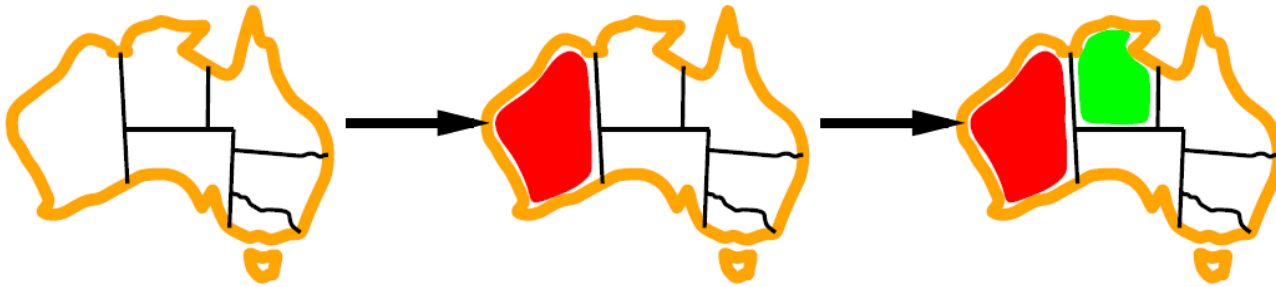 - Choose the variable with the fewest legal left values in its domain

Ordering: Minimum Remaining Values

- Variable Ordering: Minimum remaining values (MRV):
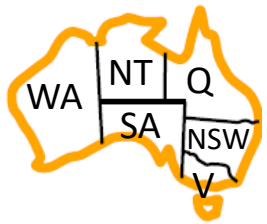 - Choose the variable with the fewest legal left values in its domain

Ordering: Minimum Remaining Values

- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain

Ordering: Minimum Remaining Values

- Variable Ordering: Minimum remaining values (MRV):
    - Choose the variable with the fewest legal left values in its domain
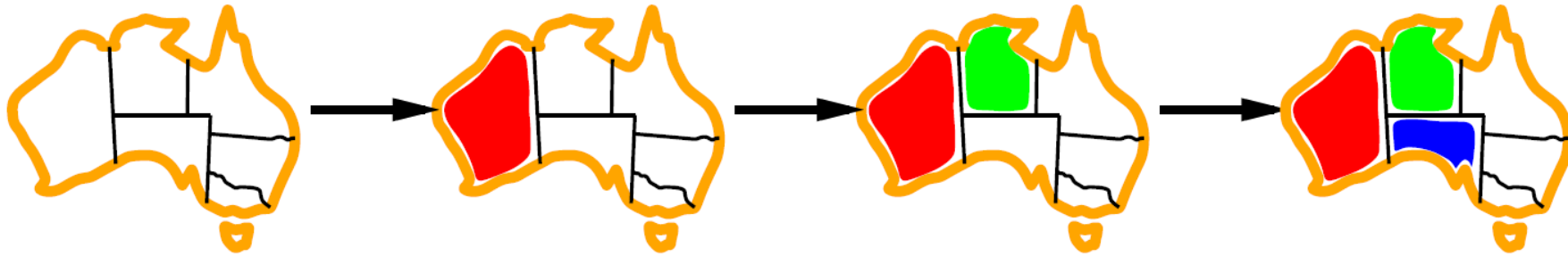

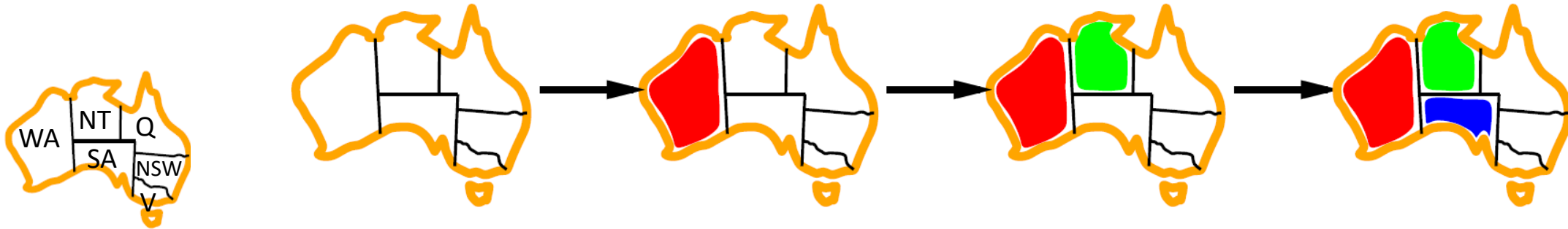- Why min rather than max?

Ordering: Minimum Remaining Values
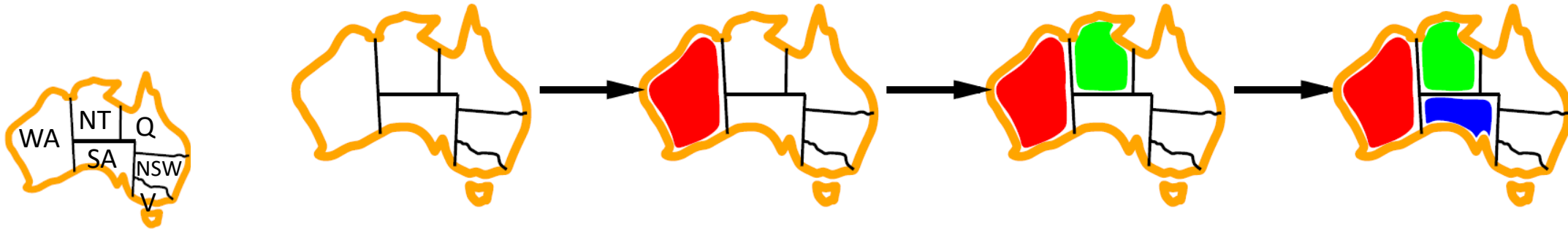
- Variable Ordering: Minimum remaining values (MRV):
    - Choose the variable with the fewest legal left values in its domain

- Why min rather than max?

Ordering: Minimum Remaining Values

- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain
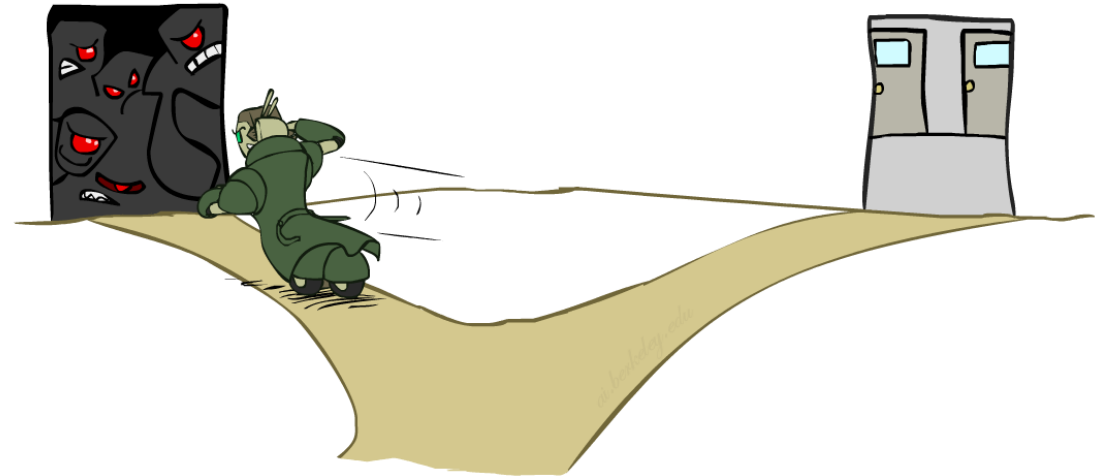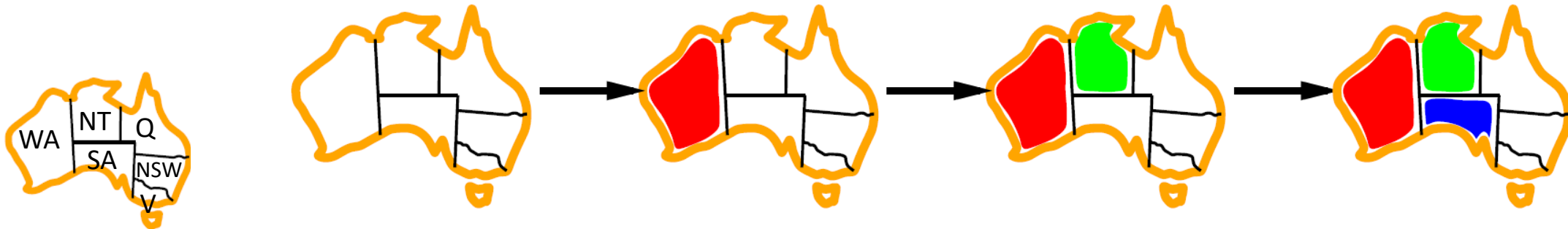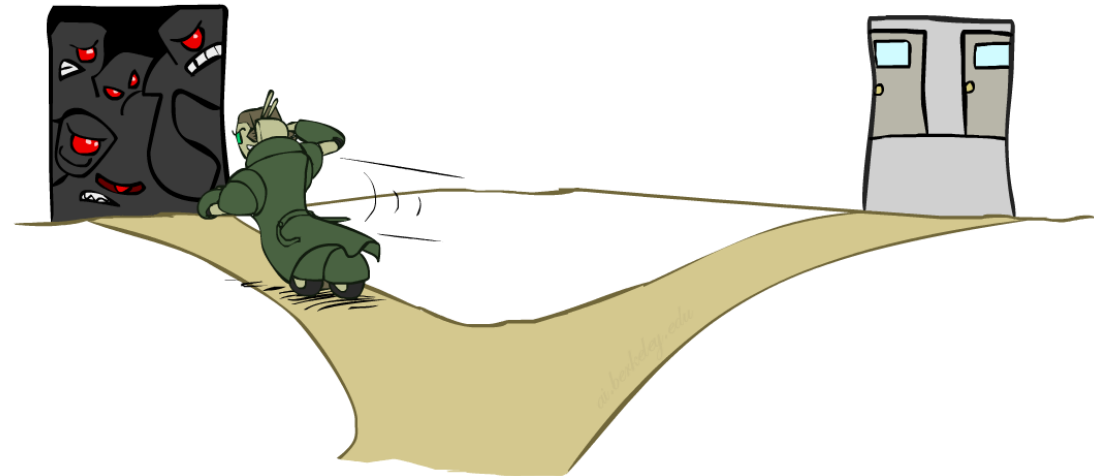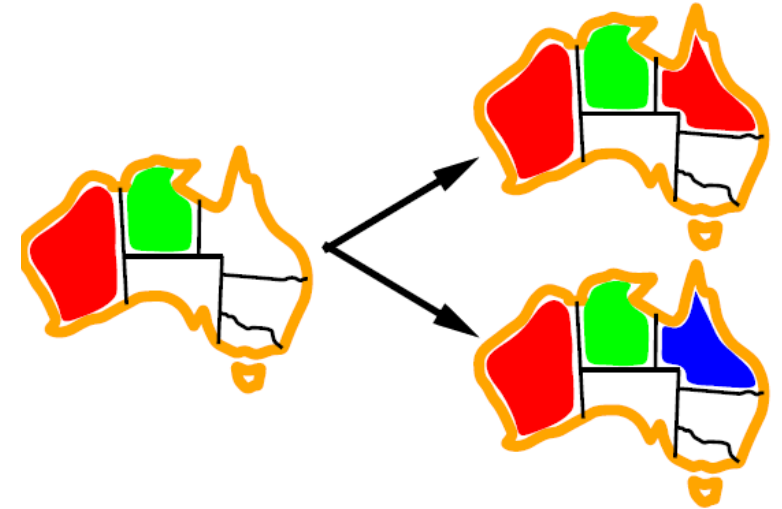

- Why min rather than max?
- Also called "most constrained variable"
- "Fail-fast" ordering

Ordering: Least Constraining Value

- **Value Ordering: Least Constraining Value**
  - Given a choice of variable, choose the *least constraining value*
  - I.e., the one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this! (E.g., rerunning filtering)
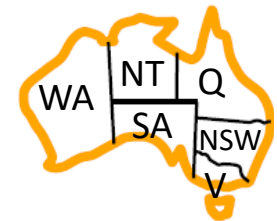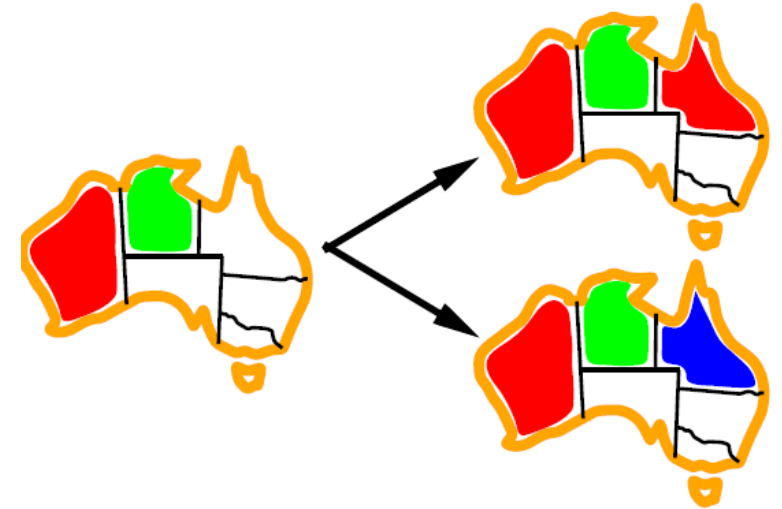
Ordering: Least Constraining Value

- Value Ordering: Least Constraining Value
  - Given a choice of variable, choose the *least constraining value*
  - I.e., the one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this! (E.g., rerunning filtering)

- Why least rather than most?

Ordering: Least Constraining Value

- **Value Ordering: Least Constraining Value**
  - Given a choice of variable, choose the *least constraining value*
  - I.e., the one that rules out the fewest values in the remaining variables
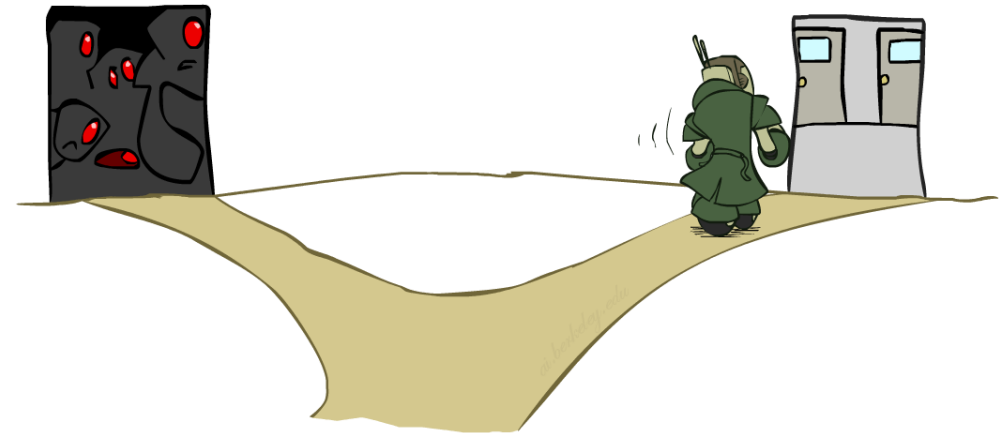 - Note that it may take some computation to determine this! (E.g., rerunning filtering)
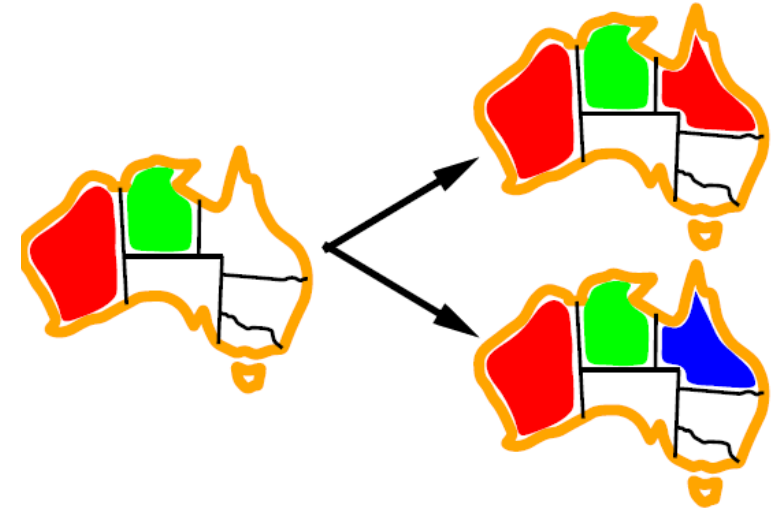
- **Why least rather than most?**

Ordering: Least Constraining Value

- **Value Ordering: Least Constraining Value**
  - Given a choice of variable, choose the *least constraining value*
  - I.e., the one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this! (E.g., rerunning filtering)

- **Why least rather than most?**

- **Combining these ordering ideas makes 1000 queens feasible**
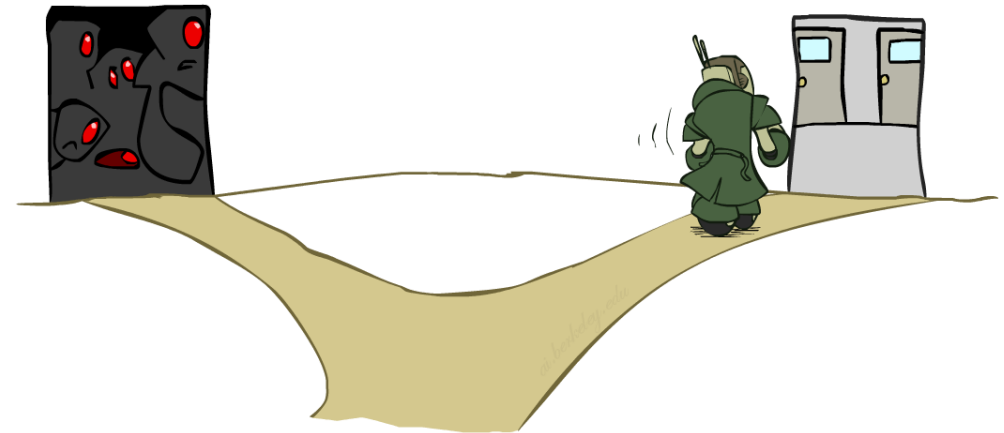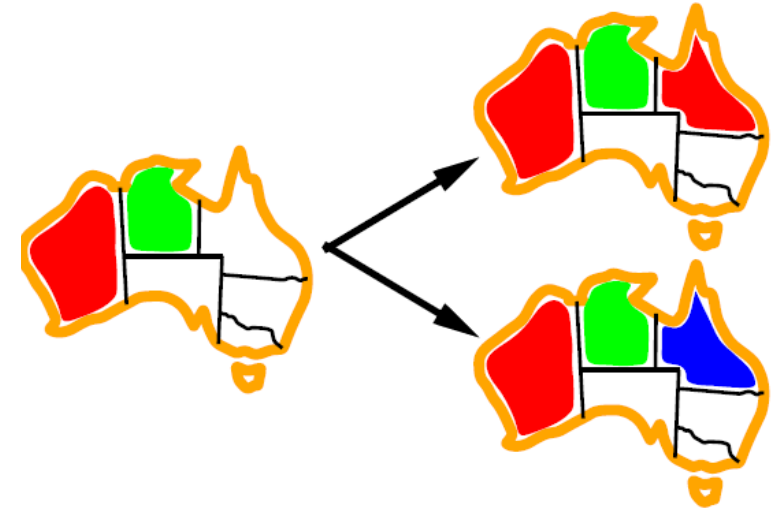
Demo: Coloring -- Backtracking + Forward Checking + Ordering