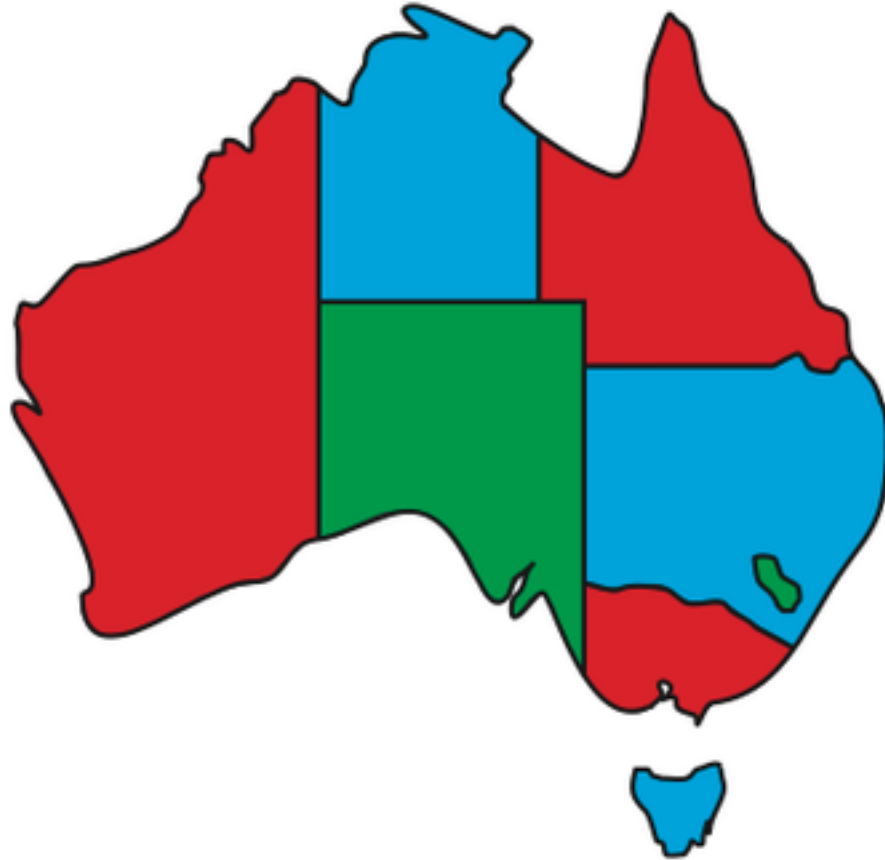
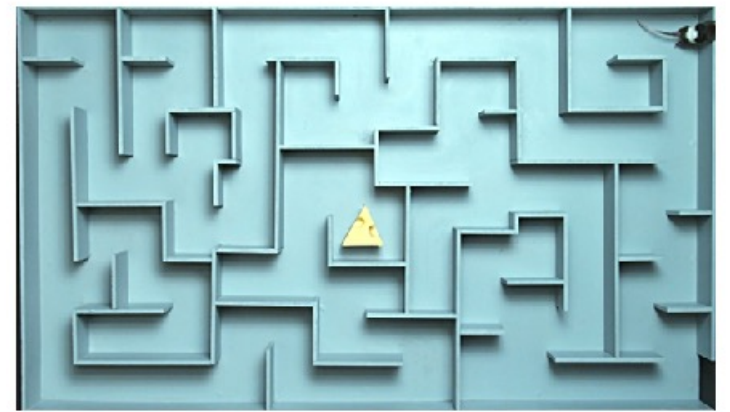
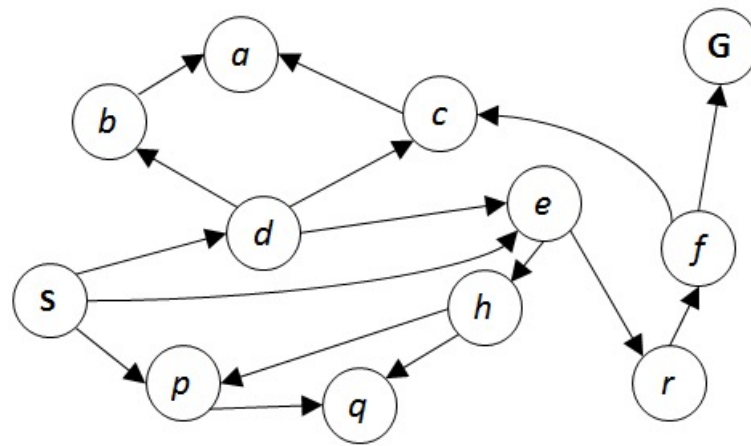


# COMP 341 Intro to AI

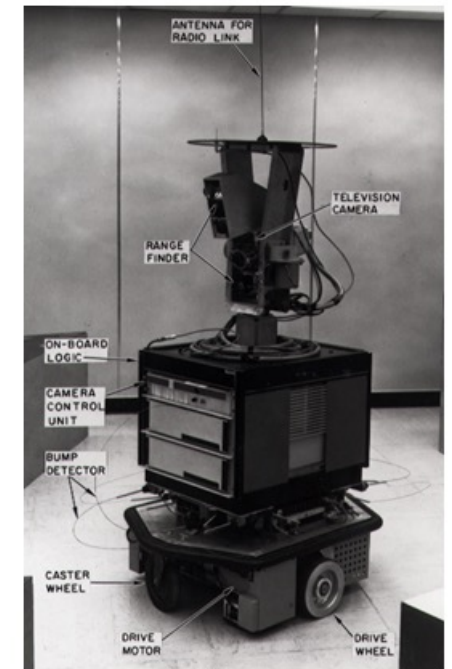
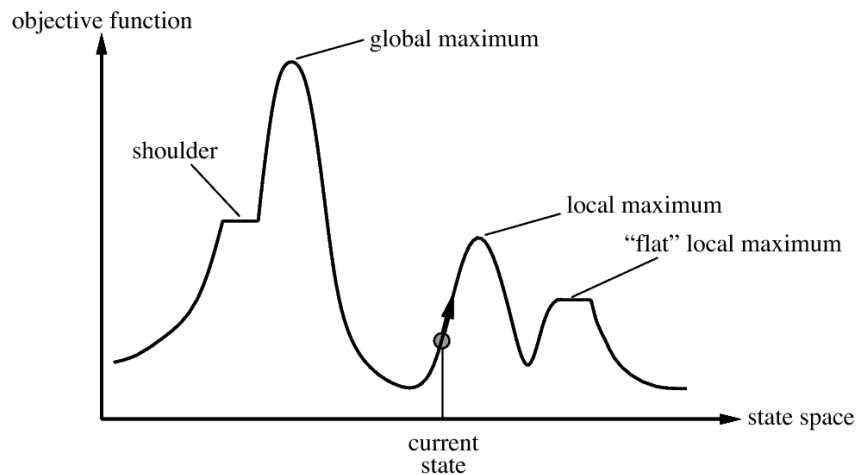
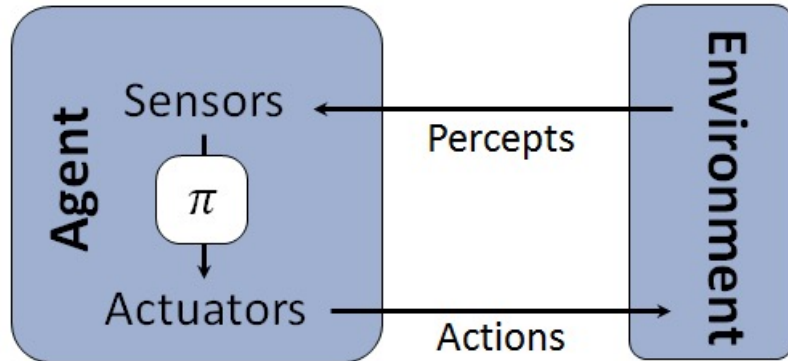
## Constraint Satisfaction Problems



Asst. Prof. Barış Akgün  
Koç University



# Previously on Intro to AI

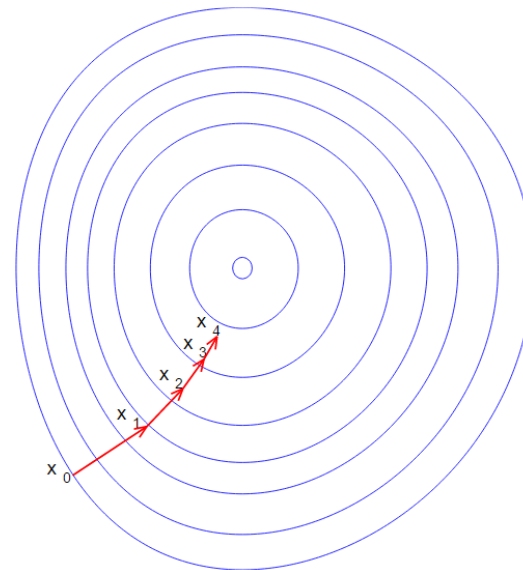
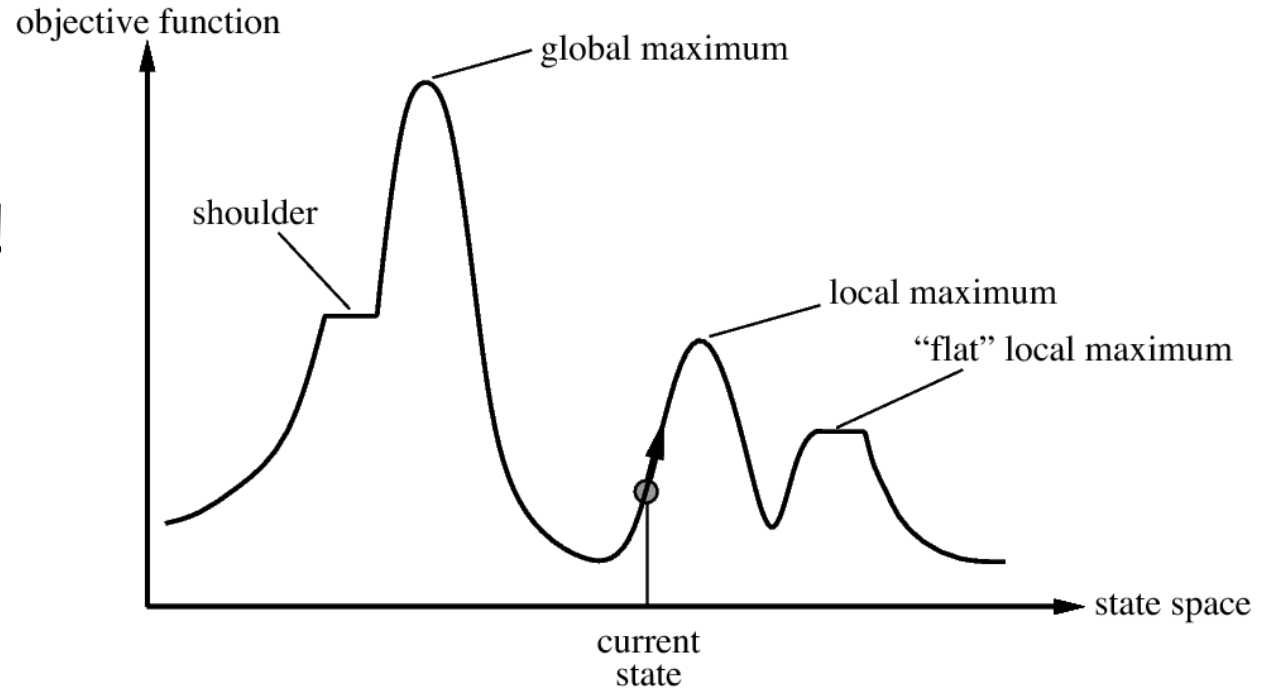


# Search

- Uninformed
- Informed
- Solution is a path to goal

# Local Search

- Solution is important, not the path!
- Hill Climbing
- Simulated Annealing
- Local Beam Search
- Genetic Algorithms
- Gradient Descent

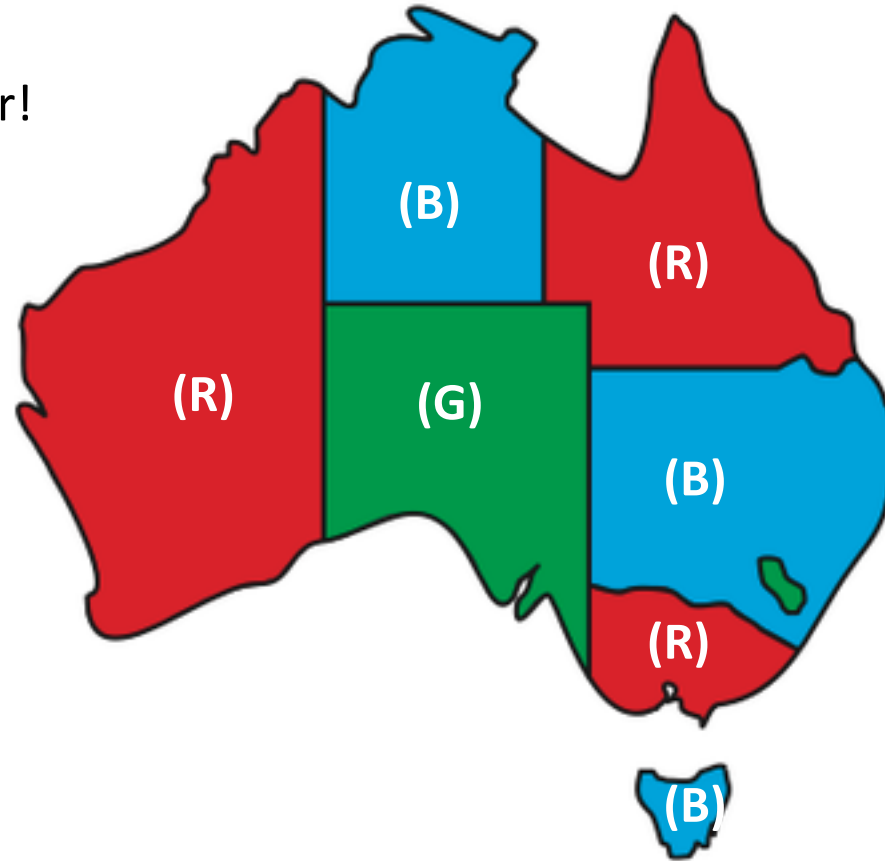


# Local Search

- Formulation:
  - Current State
  - Transition/Successor Function
  - Evaluation Function and State Space “Landscape”
- Algorithms: Move towards Better States (Where the “Local” comes from)
  - Complete: Find a solution if one exists
  - Optimal: Find the best state
- Usually easy to code!

# Constraint Satisfaction Problems

No neighbors with the same color!



# Search So Far

- Classical Search:
  - Solution is path to a goal state
- Local Search:
  - Solution is the goal state itself
- CSPs?
  - Goal matters
  - States and goal test have specific structure!
  - Allows for general heuristics

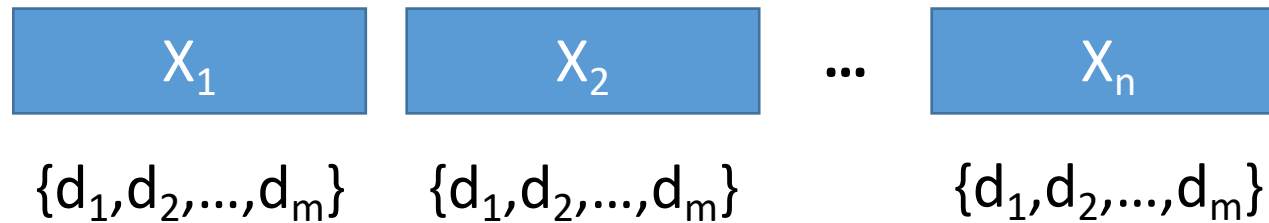
# Constraint Satisfaction Problems

- Standard Search
  - State is a **black box** data structure
  - Goal test: Can be **any Boolean function** of states
  - Successor Function: Can be **anything** that returns valid states
  - Heuristic function: Can be **anything** that maps states to a non-negative scalar
- CSPs
  - State is defined by **variables**  $X_i$  with **values** from **domain**  $D_i$ .
    - Map Coloring Example: Variables are the color of each Australian state and the domain is the set of allowable colors
  - Goal test is **a set of constraints** specifying **allowable combinations of values** for subsets of variables
    - Map Coloring Example: All states are colored and neighboring states do not have the same color
- This structure allows useful **general-purpose** algorithms with more power than standard search algorithms



# CSPs

- State is defined by **variables**  $X_i$  with **values** from **domain**  $D_i$



Domains of variables can be different!

- Goal test is **a set of constraints** specifying **allowable combinations of values** for subsets of variables. E.g.

$$\sum_{i \in A} X_i == k \qquad X_i \neq X_j \text{ for } i \neq j, i \in A, j \in A$$

# Real Life Example (!) - Carpool

- Ahmet, Elif, Mehmet, Zeynep want to carpool to Bolu
  - Variables are A,E,M,Z
- There are only 2 cars
  - Domains =  $\{C_1, C_2\}$
- The cars belong to Ahmet and Zeynep
  - Constraints:  $A = C_1$  and  $Z = C_2$
- Ahmet and Elif do not like each other
  - Constraint:  $A \neq E$
- Mehmet has a crush on Zeynep
  - Constraint:  $M = Z$
- A solution
  - $A=C_1$  ,  $E=C_2$  ,  $M= C_2$  ,  $Z=C_2$

Side Note: They should just sit Elif in the front and Ahmet in the back and take 1 car

But the problem does not model that!

# Solving CSPs

- Each state of the problem is a possible assignment to some or all the variables
- **Legal Assignment:** no violations
- **Complete Assignment:** every variable assigned

# Map Coloring

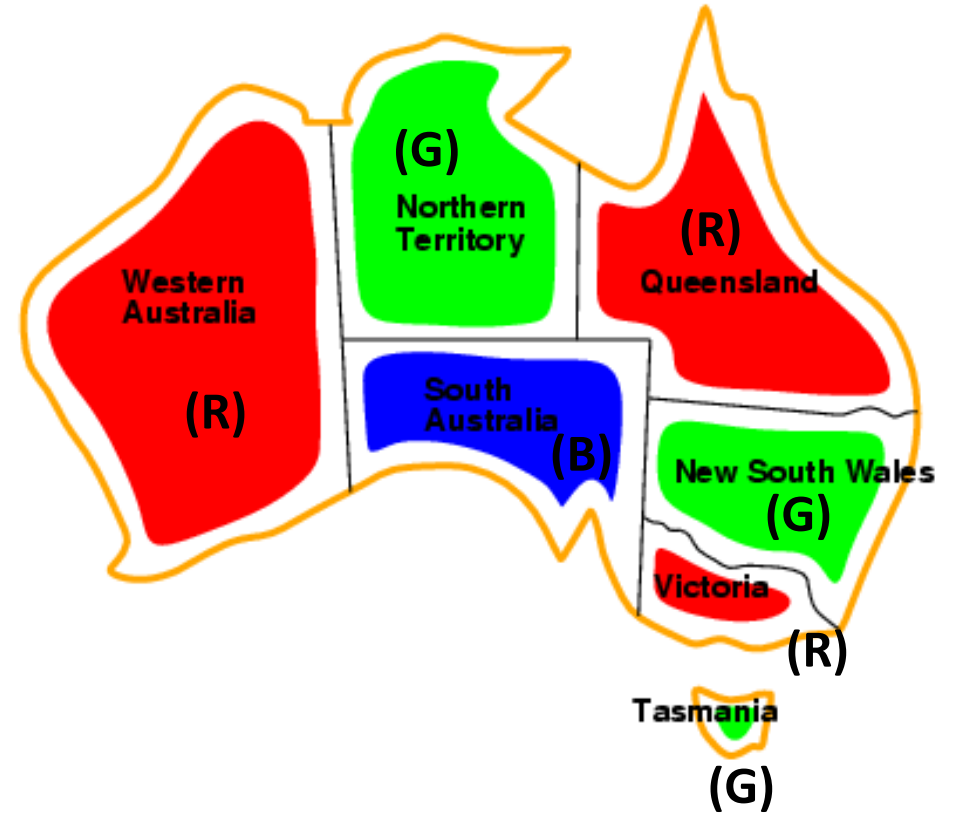
- Color the map such that no two neighbors have the same color
- Variables:
  - $WA, NT, Q, NSW, V, SA, T$
- Domains:
  - $D_i = \{\text{red, green, blue}\}$
- Constraints: adjacent regions must have different colors
  - Implicit:  $WA \neq NT$
  - Explicit:  $(WA, NT) \in \{(\text{red, green}), (\text{red, blue}), (\text{green, red}), (\text{green, blue}), \dots\}$



# Map Coloring

- Color the map such that no two neighbors have the same color
- Solutions are assignments satisfying all constraints, e.g.,

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

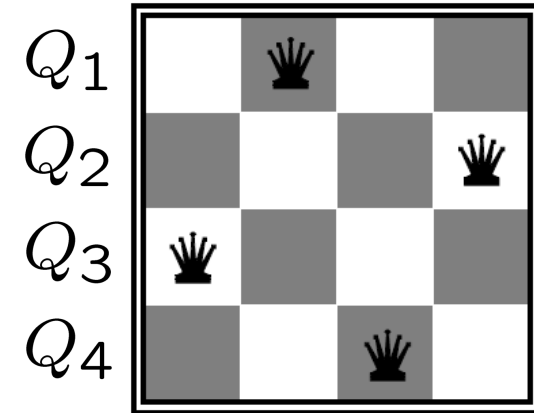


# Example: N-Queens

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

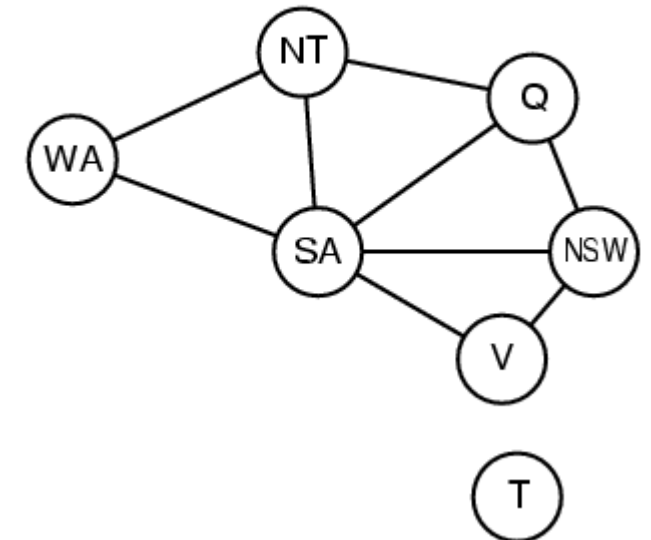
Implicit:  $\forall i, j \text{ non-threatening}(Q_i, Q_j)$

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



# Constraint Graph

- Binary CSP: each constraint relates (at most) two variables
- **Binary Constraint Graph** is a data structure we use to represent the problem
  - Nodes are variables
  - Arcs show which variables are constrained



# Example: Cryptarithmic

- Variables:

$F \ T \ U \ W \ R \ O \ X_1 \ X_2 \ X_3$

- Domains:

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

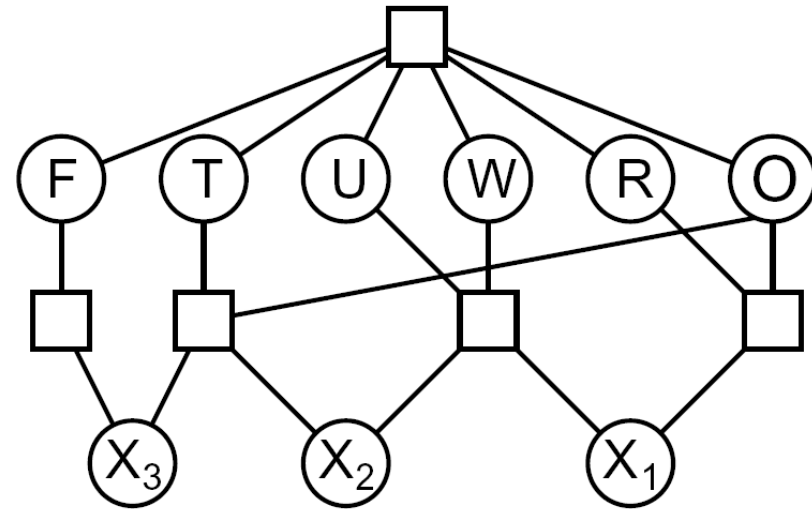
- Constraints:

$\text{alldiff}(F, T, U, W, R, O)$

$O + O = R + 10 \cdot X_1$

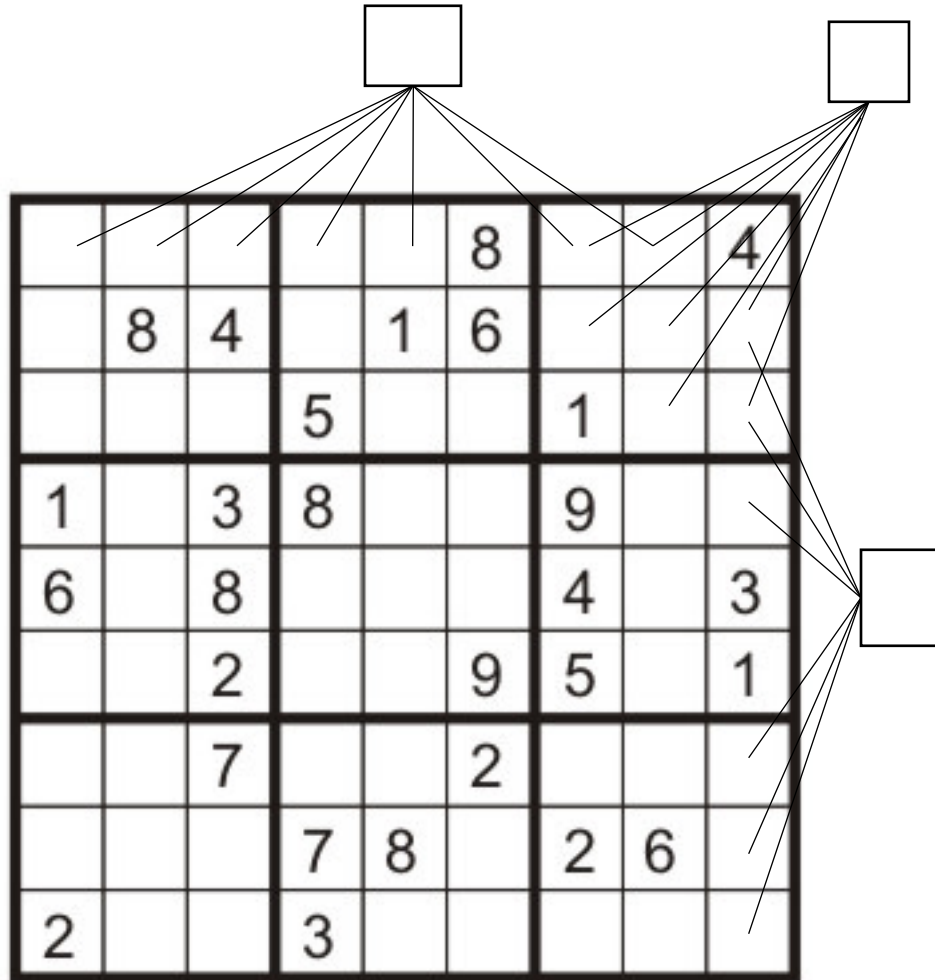
$\dots$

$$\begin{array}{r} T \ W \ O \\ + \ T \ W \ O \\ \hline F \ O \ U \ R \end{array}$$





# Most Famous CSP - Sudoku



- Variables:
  - Each (open) square
- Domains:
  - $\{1,2,\dots,9\}$
- Constraints:

9-way alldiff for each column

9-way alldiff for each row

9-way alldiff for each region

(or can have a bunch of pairwise inequality constraints)

# Other Real-World Examples

- Assignment problems: e.g., who teaches what class
  - Timetabling problems: e.g., which class is offered when and where?
  - Hardware configuration
  - Transportation scheduling
  - Factory scheduling
  - Floor Planning
  - ...
- 
- Many real-world problems involve real-valued variables...

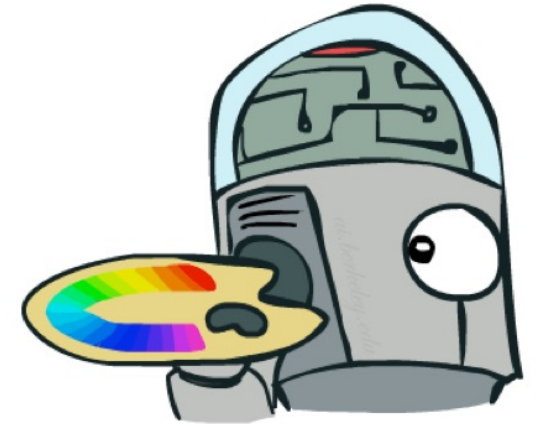
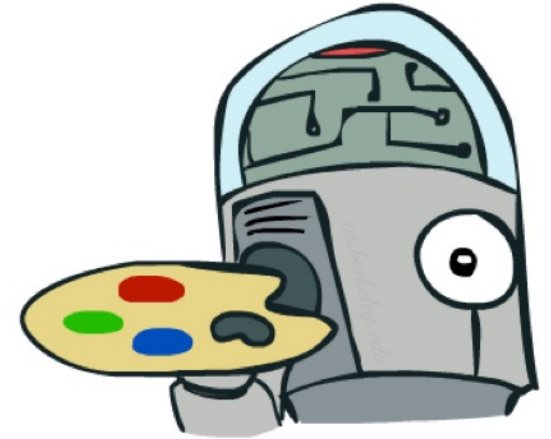
# Varieties of CSPs

- Discrete Variables

- Finite domains
  - Size  $d$  means  $O(d^n)$  complete assignments
  - E.g., Boolean CSPs, including Boolean satisfiability (NP-complete)
- Infinite domains (integers, strings, etc.)
  - E.g., job scheduling, variables are start/end times for each job
  - Need constraint language:  $Job1 + 5 < Job2$
  - Linear constraints solvable, nonlinear undecidable

- Continuous variables

- E.g., start/end times for Hubble Telescope observations
- Linear constraints solvable in polynomial time by Linear Programming methods (ever heard of the Simplex Method?)

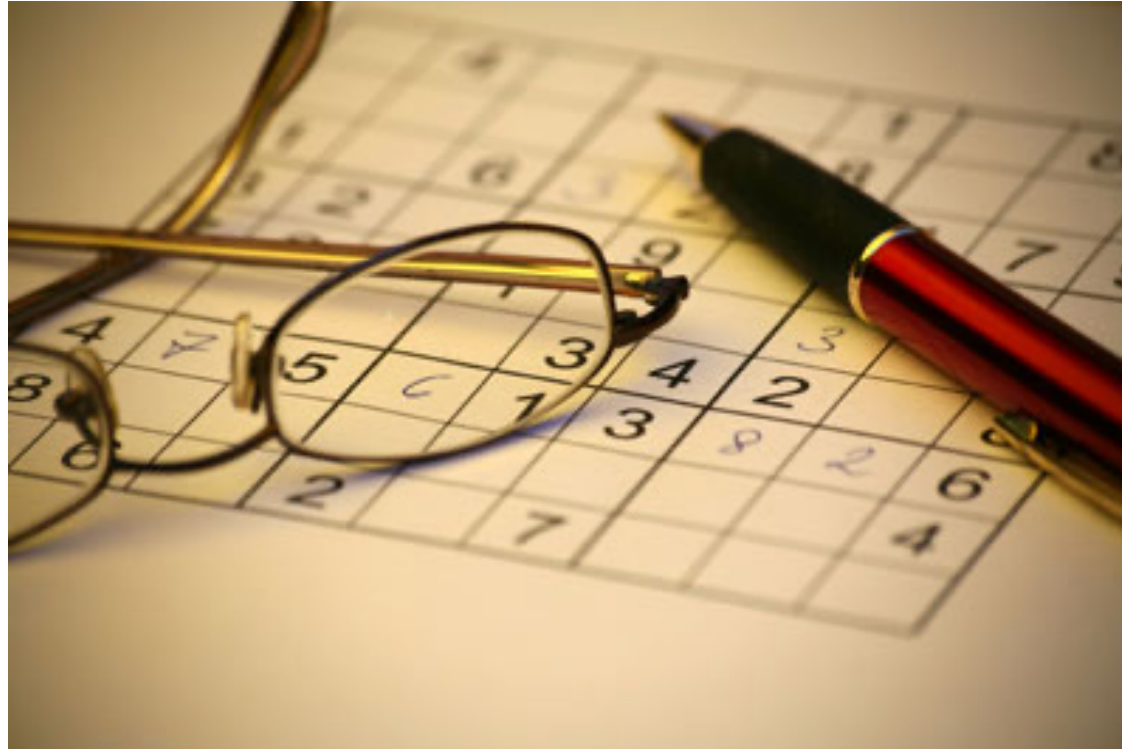


# Varieties of Constraints

- Varieties of Constraints
  - Unary constraints involve a single variable (equivalent to reducing domains)  
e.g.:  $SA \neq green$
  - Binary constraints involve pairs of variables,  
e.g.:  $SA \neq WA$
  - Higher-order constraints involve 3 or more variables:  
e.g.: cryptarithmic column constraints
- Preferences (soft constraints):
  - E.g., red is better than green
  - Often representable by a cost for each variable assignment
  - Gives rise to constrained optimization problems
  - More involved methods out of our scope but local search can be used (how?)



# Solving CSPs



Search Formulation

Local Search Formulation

# Search Formulation for CSPs

- Initial State: {}
- Successor(): assign a value (consistent with constraints) to an unassigned variable
- Goal Test(): All variables are assigned and all constraints are satisfied
- Failure: No legal assignment to do
- This is the **same** for all CSPs!
- Path is irrelevant
- Every solution appears at depth  $n$  with  $n$  variables
  - DFS anyone
- Complexity ( $n$  vars,  $d$  values)
  - Branch factor:  $(n-l)d$  at depth  $l$
  - $n!d^n$  leaves!

# 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 constraints
  - “Incremental goal test”
- Depth-first search with these two improvements is called *backtracking search* (not the best name)
- Can solve n-queens for  $n \approx 25$

# Detour: Recursive DFS

**function** RECURSIVE-DFS(*problem*) **returns** a solution, or failure  
    **return** RECURSIVE-DFS\_(MAKE-NODE(*problem*.INITIAL-STATE), *problem*)

**function** RECURSIVE-DFS\_(*node*,*problem*) **returns** a solution, or failure  
    **if** *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)  
    **for each** *action* in *problem*.ACTIONS(*node*.STATE) **do**  
        *child* ← CHILD-NODE(*problem*, *node*, *action*)  
        *result* ← RECURSIVE-DFS\_(*child*, *problem*)  
        **if** *result* != failure **then return** *result*  
    **return** failure



# 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

select a  
variable

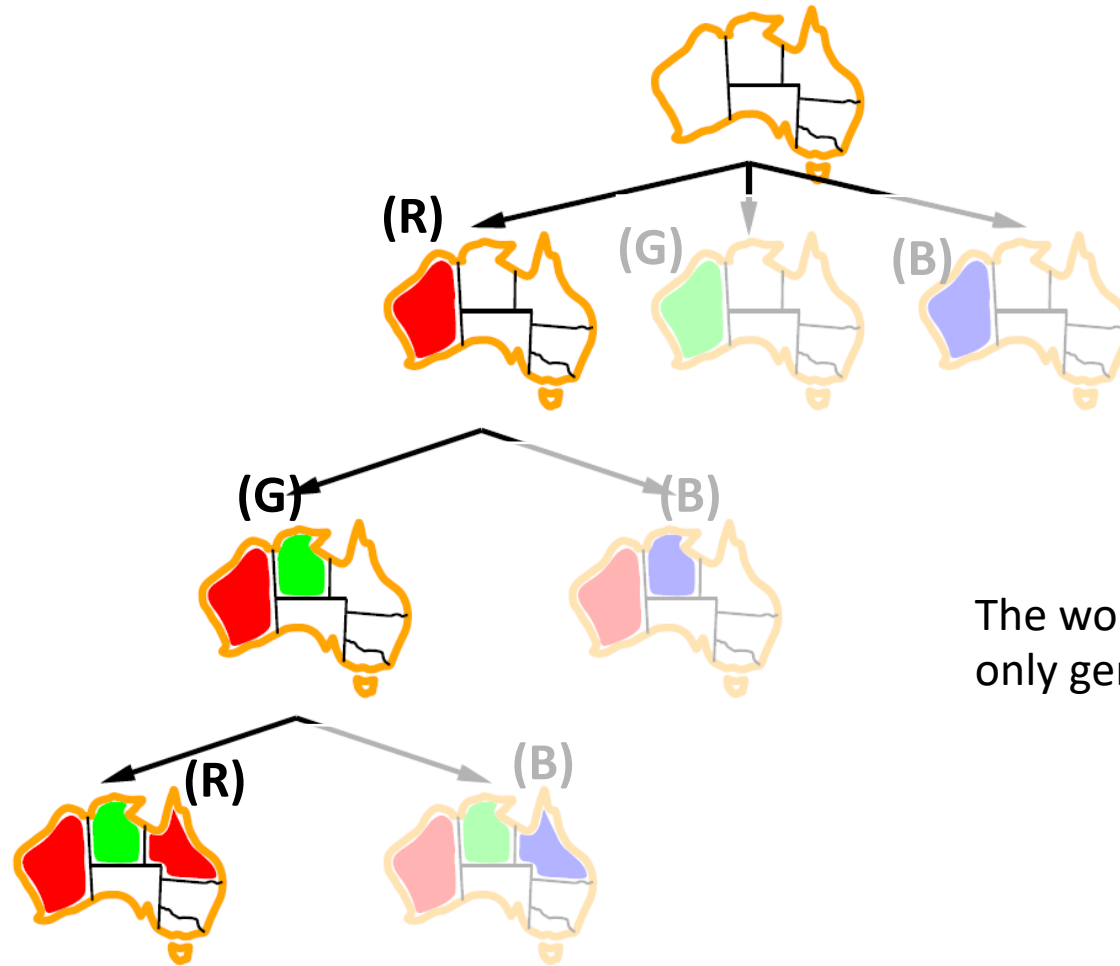
Find a value  
consistent with  
constraints

Recurse to  
assign another

only keeps a single  
representation of  
the assigned state!

If no consistent assignment exists, return  
failure, which causes another value to be  
tried

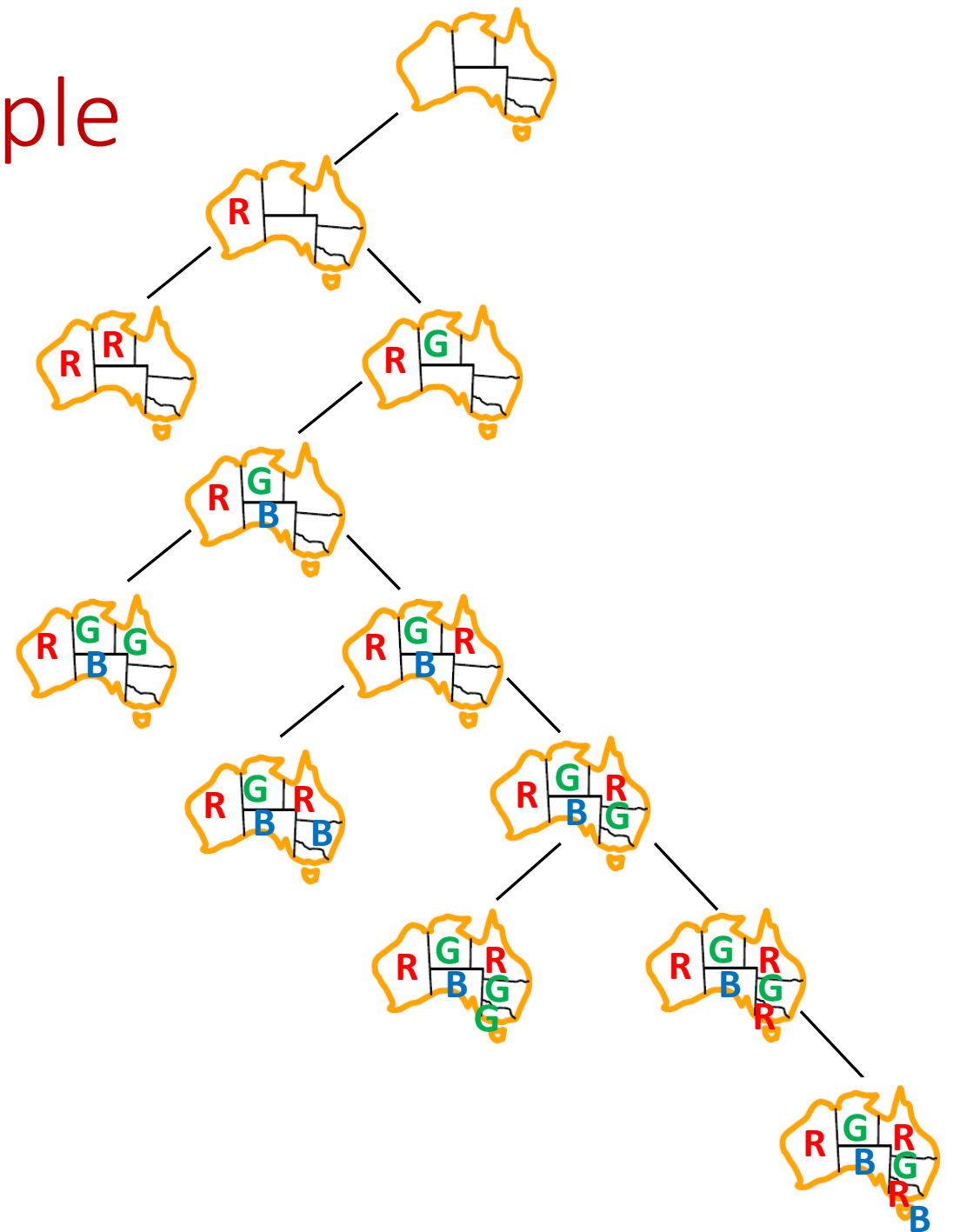
# Backtracking Example



The worn-out states are  
only generated if needed

# Another Backtracking Example

- Variable Order: Left to right, top to bottom
- Value Order: Random (assume we have the following)
  - Red
  - Red
  - Green
  - Blue
  - Green
  - Red
  - Blue
  - Green
  - Green
  - Red
  - Blue



# Improvements

- Backtracking: DFS + variable ordering + constraint checking
- Uninformed: Add heuristics to improve
- **General Purpose Heuristics** thanks to the structure of CSPs
- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?
- Filtering: What inference can be made to detect failures early ?
- Structure: Can we exploit the problem structure?

# Emphasis Slide

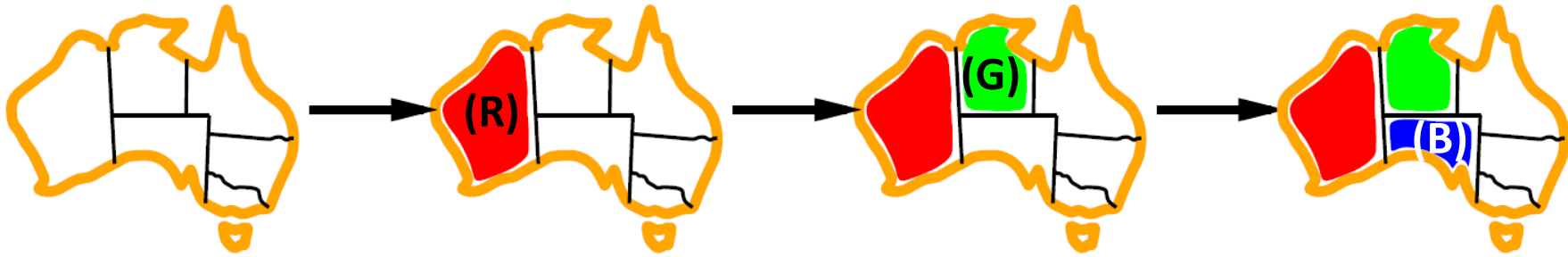
- Variables (e.g. map location) can take values (e.g. specific color) from their domain (e.g. set of colors)
- Ordering
  - Picking the **variable** to assign next
  - Picking the **value** to assign to the chose variable
- Filtering: Filter the domains
  - Removing the infeasible values from the domains(e.g. remove colors)

# Variable Ordering: What variable to assign next?

- Fixed order
- Random
- Other ideas?
  - Let's look at the number of constraints per variable!

# Minimum Remaining Values (MRV)

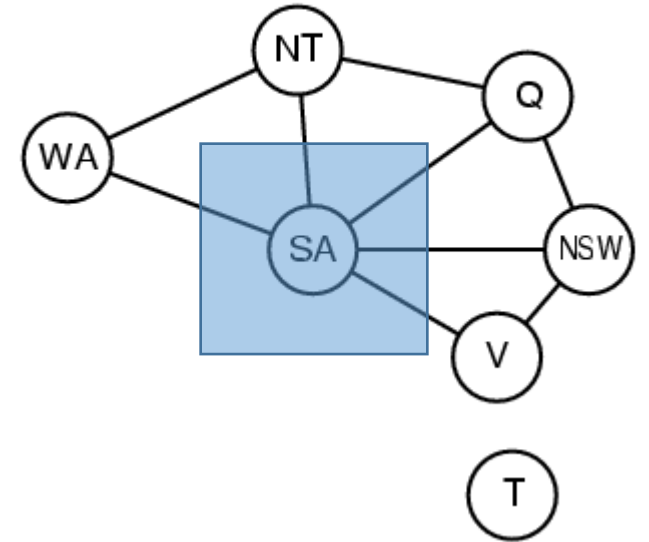
- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain



- Why min rather than max?
- Also called “most constrained variable”
- “Fail-fast” ordering + not running out of options

# Degree Heuristic

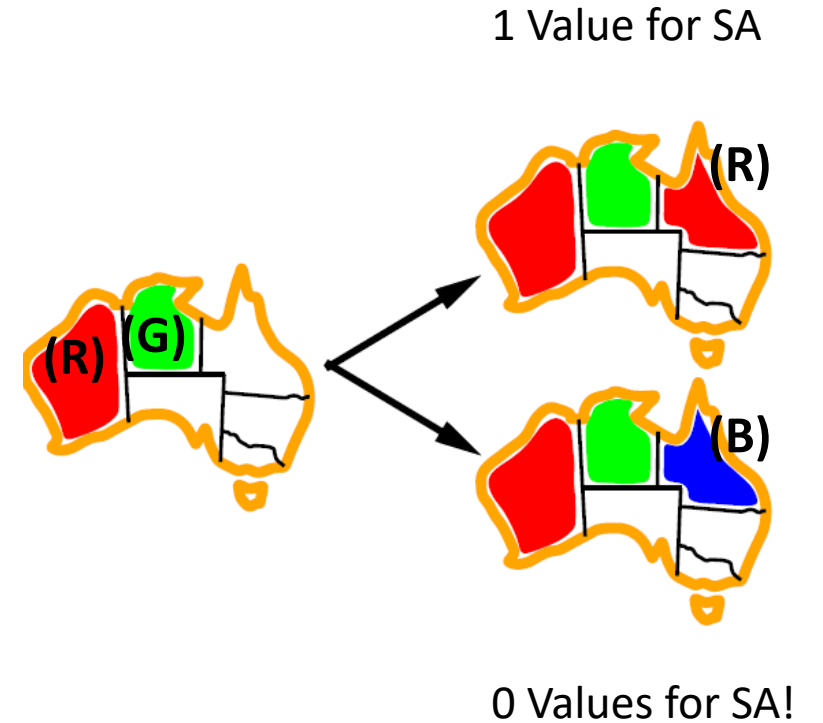
- Tie breaker among MRV variables
- Choose the variable with most constraints on remaining variables





# Least Constraining Value

- Which value to assign next?
- 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
- Why least rather than most?
- Combining these ordering ideas makes 1000 queens feasible



# Summary of Ordering

- Which variable to select next and which value to assign to it?
- Detect failures early (MRV + DH) – Picking variables
- Enter the most promising branch (LCV) – Picking values
- Note that the heuristics do not change the theoretical bounds!

# Emphasis Slide

- Variables (e.g. map location) can take values (e.g. specific color) from their domain (e.g. set of colors)
- Ordering
  - Picking the **variable** to assign next – MRV (+ DH as tie breaker)
  - Picking the **value** to assign to the chose variable - LCV

# Backtracking Search + Heuristics



- Variable Domains
  - WA: {**R**, **G**, **B**}
  - NT: {**R**, **G**, **B**}
  - SA: {**R**, **G**, **B**}
  - Q: {**R**, **G**, **B**}
  - NSW: {**R**, **G**, **B**}
  - V: {**R**, **G**, **B**}
  - T: {**R**, **G**, **B**}
- Tie Breaking Variable Order up-down, left-right from above
- Tie Breaking Value Order: **R** – **G** – **B**

Step	MRV	DH	LCV	Assignment
1				
2				
3				
4				
5				
6				
7				



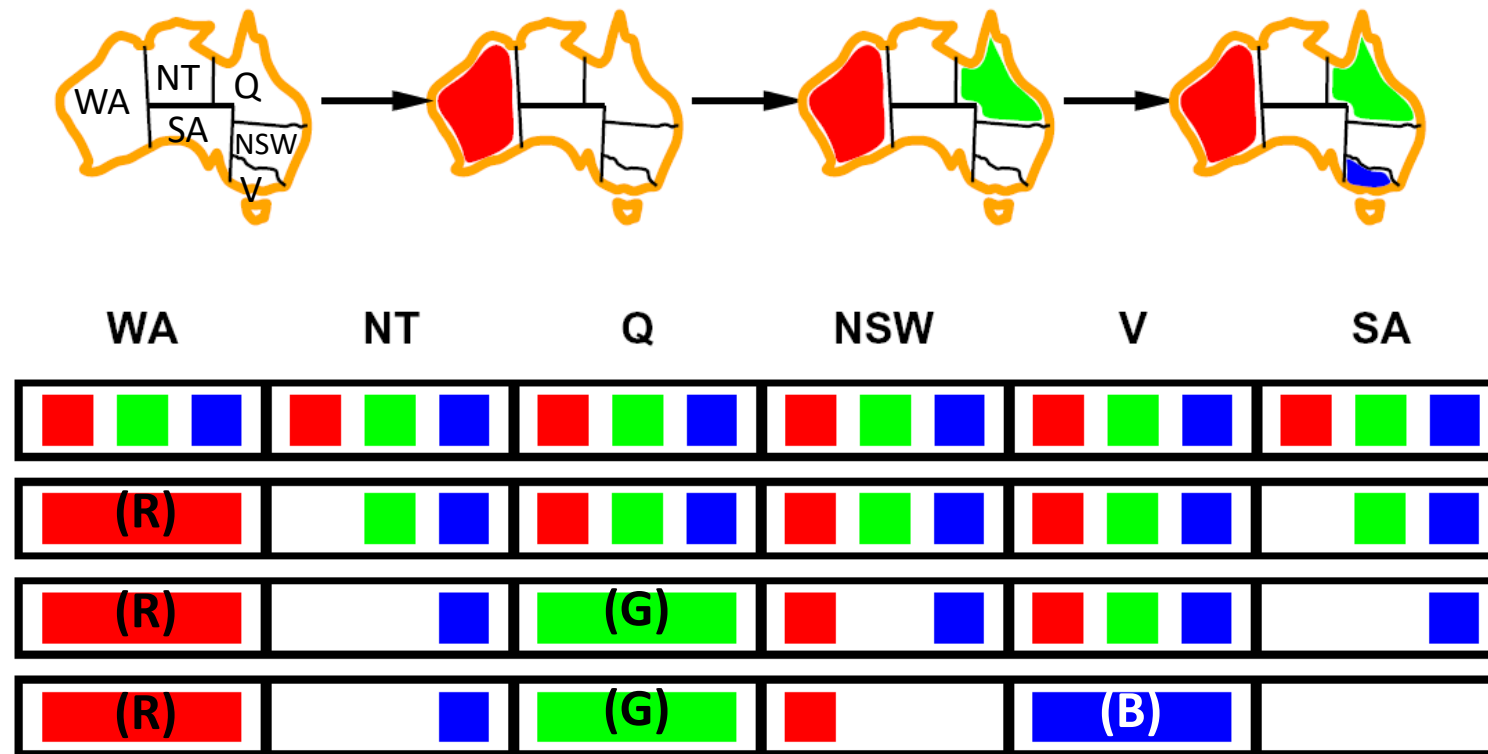
We didn't need to backtrack for this example, but this is not the norm

# Filtering

- How to detect failures early?
- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward Checking
- Constraint Propagation - Arc consistency

# Forward Checking

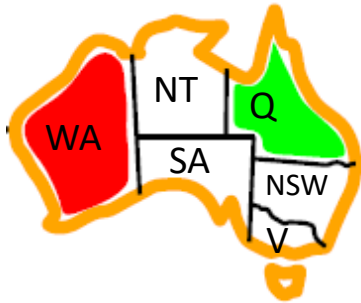
- Forward checking: Cross off values that violate a constraint when added to the existing assignment
- Backtrack when no assignments left



We are already doing FC for MRV!

# Constraint Propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures
- After deleting neighbors, check constraints for all other variables



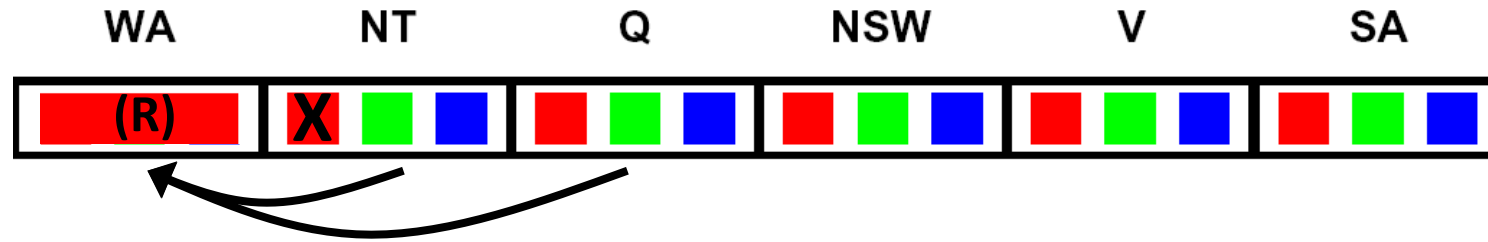
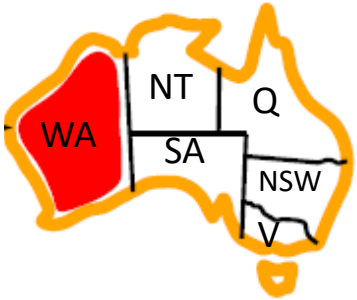
WA	NT	Q	NSW	V	SA
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div>(R)</div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div>(R)</div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

- NT and SA cannot both be blue!
- Why didn't we detect this yet?
- *Constraint propagation*: reason from constraint to constraint

Legend:  
Left: Red  
Middle: Green  
Right: Blue

# Consistency of A Single Arc

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



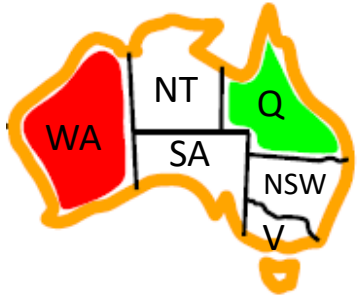
Legend:  
Left: Red  
Middle: Green  
Right: Blue

- Delete from 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:

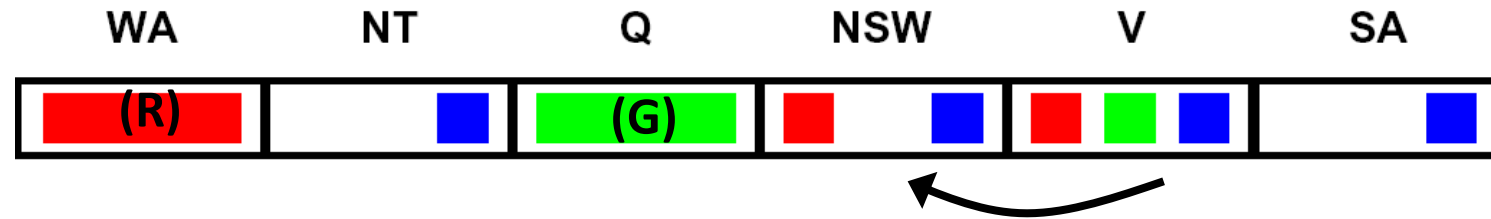
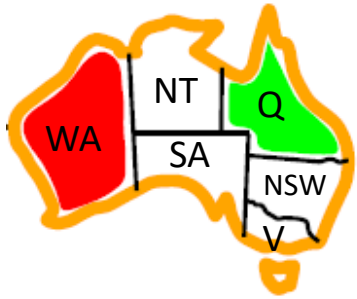


Legend:  
Left: Red  
Middle: Green  
Right: Blue

*Remember: Delete  
from the tail!*

# Arc Consistency of an Entire CSP

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

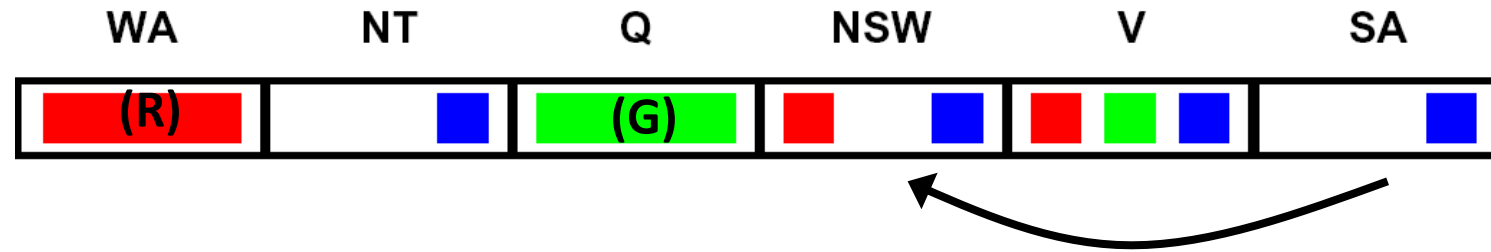
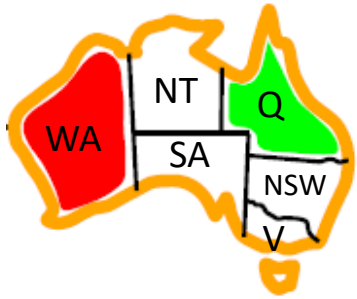


Legend:  
Left: Red  
Middle: Green  
Right: Blue

*Remember: Delete  
from the tail!*

# Arc Consistency of an Entire CSP

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

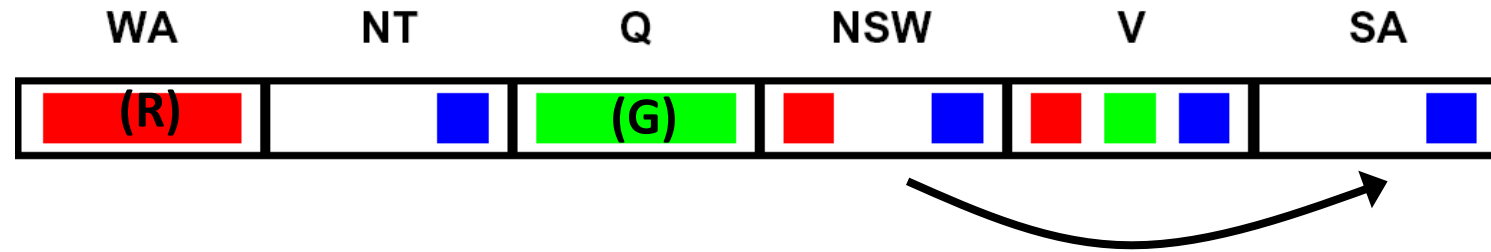
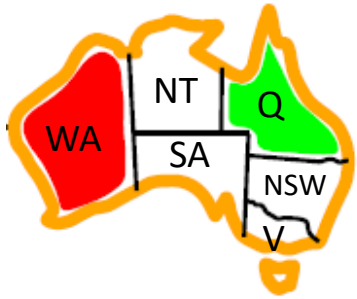


Legend:  
Left: Red  
Middle: Green  
Right: Blue

*Remember: Delete  
from the tail!*

# Arc Consistency of an Entire CSP

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

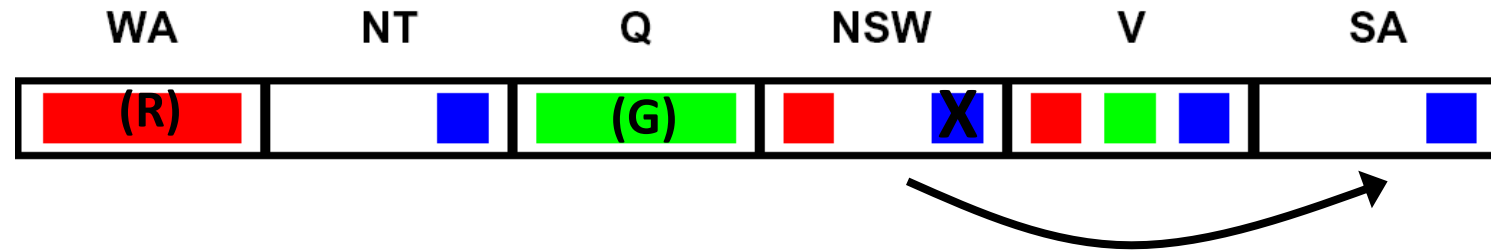
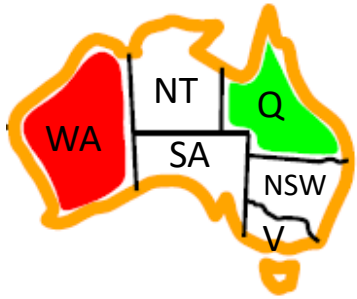


Legend:  
Left: Red  
Middle: Green  
Right: Blue

*Remember: Delete  
from the tail!*

# Arc Consistency of an Entire CSP

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

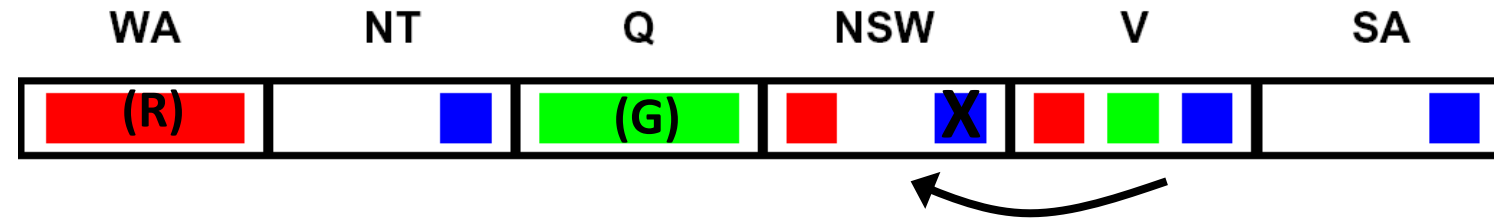
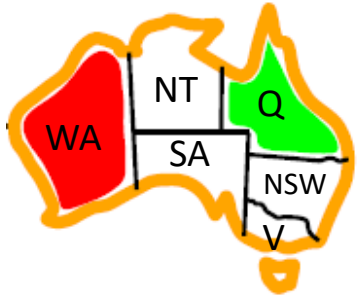


Legend:  
Left: Red  
Middle: Green  
Right: Blue

*Remember: Delete  
from the tail!*

# Arc Consistency of an Entire CSP

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

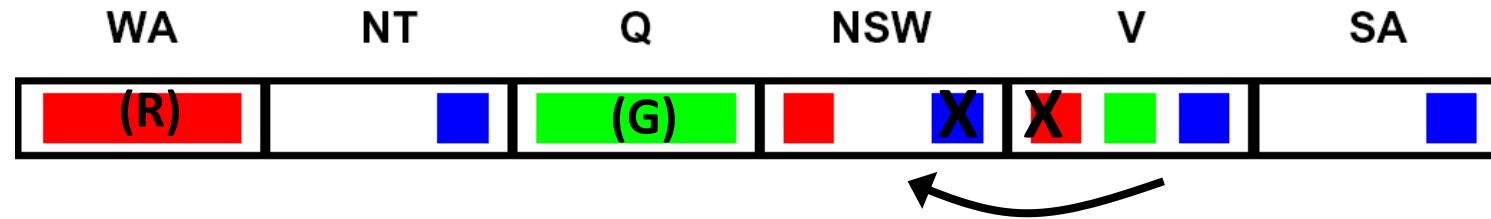
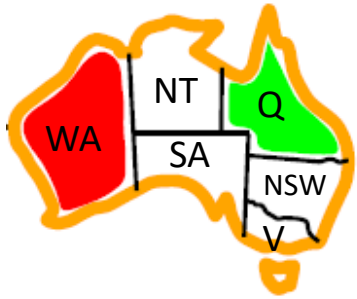


Legend:  
Left: Red  
Middle: Green  
Right: Blue

*Remember: Delete  
from the tail!*

# Arc Consistency of an Entire CSP

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

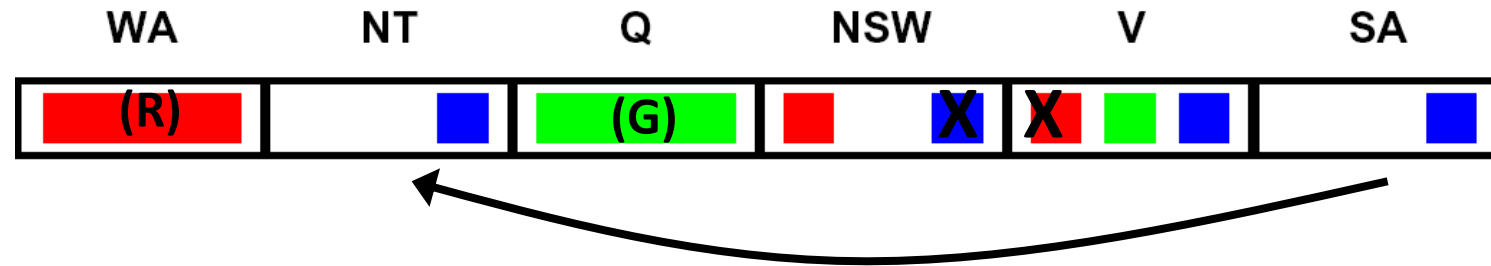
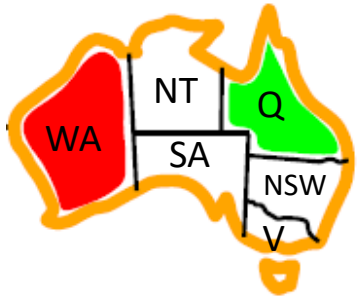


Legend:  
Left: Red  
Middle: Green  
Right: Blue

*Remember: Delete  
from the tail!*

# Arc Consistency of an Entire CSP

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



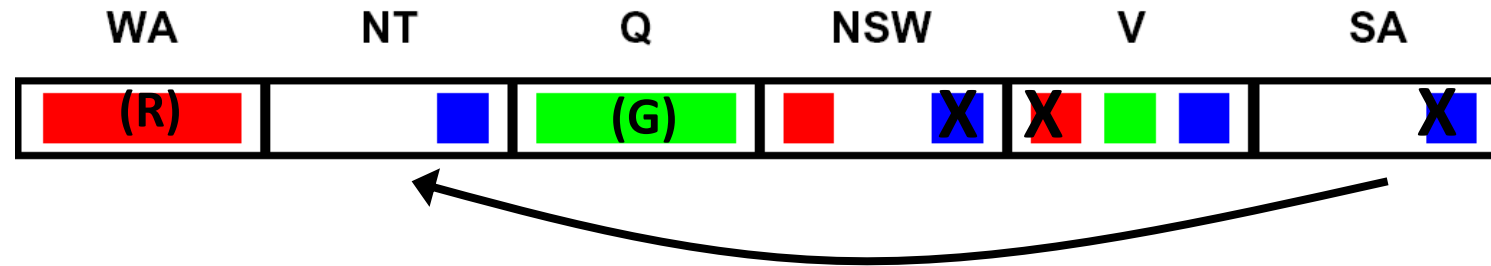
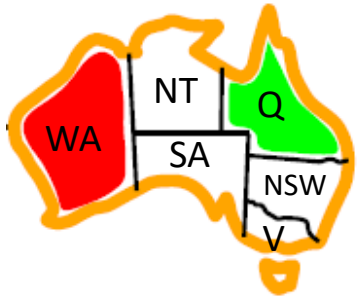
Legend:  
Left: Red  
Middle: Green  
Right: Blue

*Remember: Delete  
from the tail!*



# Arc Consistency of an Entire CSP

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

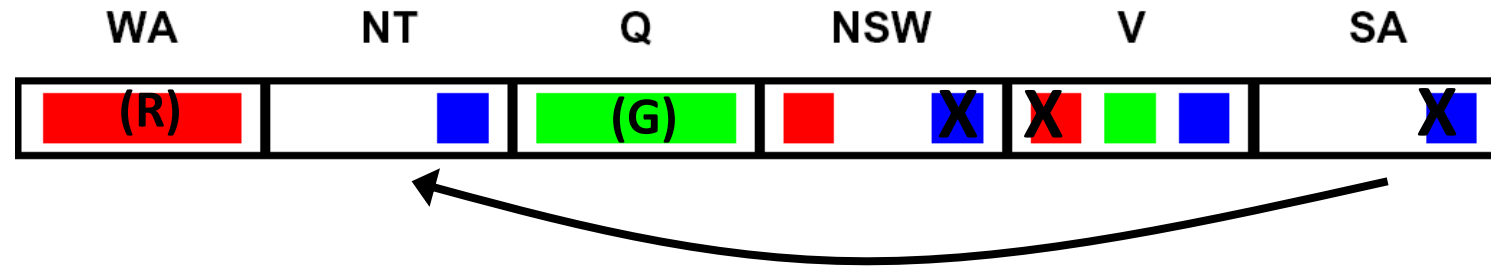
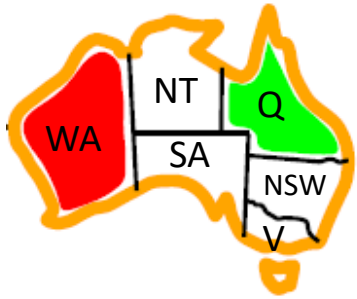


Legend:  
Left: Red  
Middle: Green  
Right: Blue

*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 before or after each assignment
- What's the downside of enforcing arc consistency?

Legend:  
Left: Red  
Middle: Green  
Right: Blue

*Remember: Delete  
from the tail!*

# Enforcing Arc Consistency in a CSP

Check consistency and  
remove value if necessary

Add all the neighbors to  
the queue if something is  
removed

Delete from tail to  
enforce consistency

**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**

$(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\text{queue})$

**if** REVISE(*csp*,  $X_i, X_j$ ) **then**

**if** size of  $D_i = 0$  **then return** false

**for each**  $X_k$  in  $X_i.\text{NEIGHBORS} - \{X_j\}$  **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  $\leftarrow$  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

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

# Arc Consistency and Forward Checking

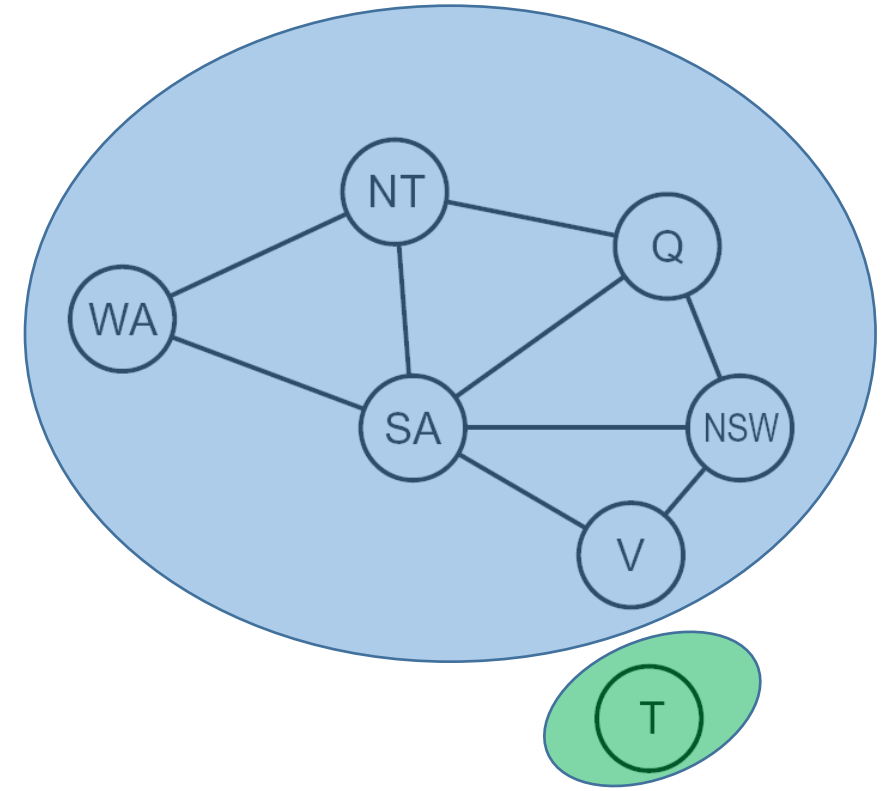
- FC is essentially an arc consistency check for a single node!
  - Head is the assigned node, tails are its neighbors
- If you ran AC, no need to run FC
- FC is faster per value-assignment
- AC can catch failures earlier

# Summary of Filtering

- FC: Remove values from the domains neighboring nodes
  - Fast to compute
  - Plays well with MRV
  - Does not catch some failures early
- Arc Consistency: Make all arcs consistent after an assignment
  - Keep checking arcs until there is no change
  - Entails FC
  - Earlier failure detection
  - Costly to Compute (especially if there are a lot of constraints)

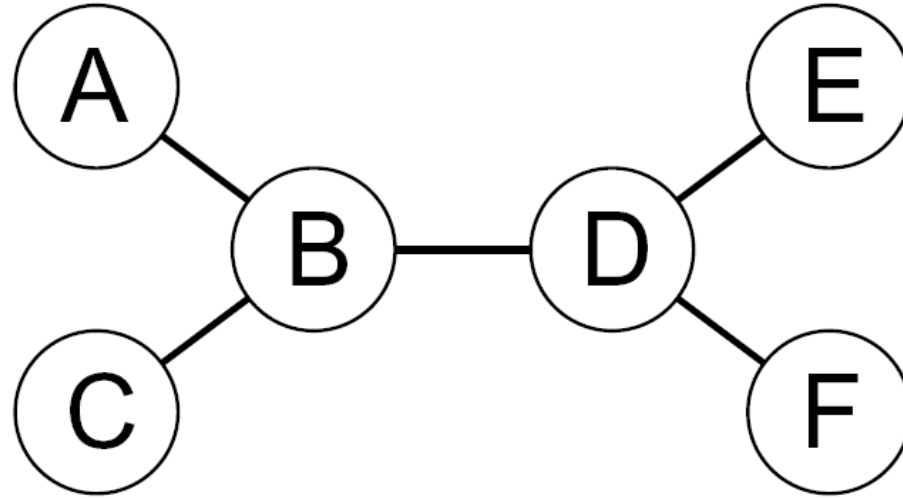
# Problem Structure

- Extreme case: independent subproblems
  - Example: Tasmania and mainland do not interact
- Independent subproblems are identifiable as connected components of constraint graph
- Suppose a graph of  $n$  variables can be broken into subproblems of only  $c$  variables:
  - Worst-case solution cost is  $O((n/c)(d^c))$
  - E.g.,  $n = 80$ ,  $d = 2$ ,  $c = 20$
  - $2^{80} = 4$  billion years at 10 million nodes/sec
  - $(4)(2^{20}) = 0.4$  seconds at 10 million nodes/sec



It's rare to find unconnected parts of the graph

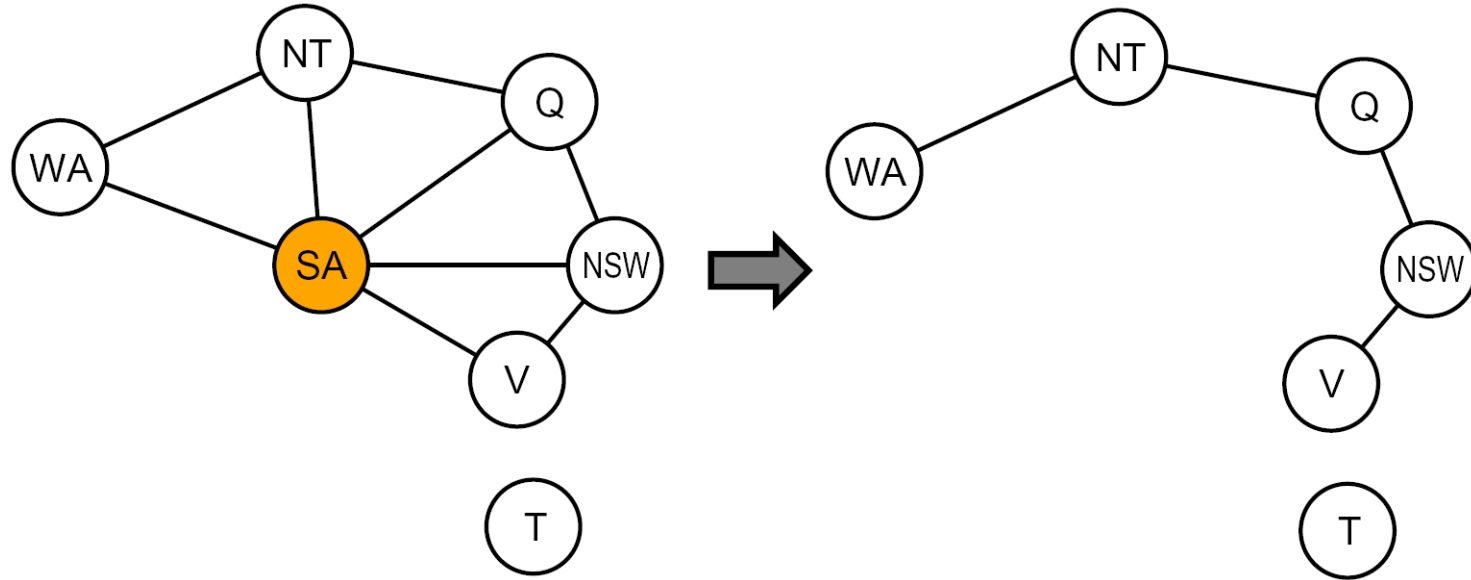
# Tree-Structured CSPs



- Theorem: if the constraint graph has no loops, the CSP can be solved in  $O(nd^2)$  time
- Compare to general CSPs, where worst-case time is  $O(d^n)$

(Skipping the algorithm this semester)

# 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  $c$  gives runtime  $O( (d^c) (n-c) d^2 )$ , very fast for small  $c$



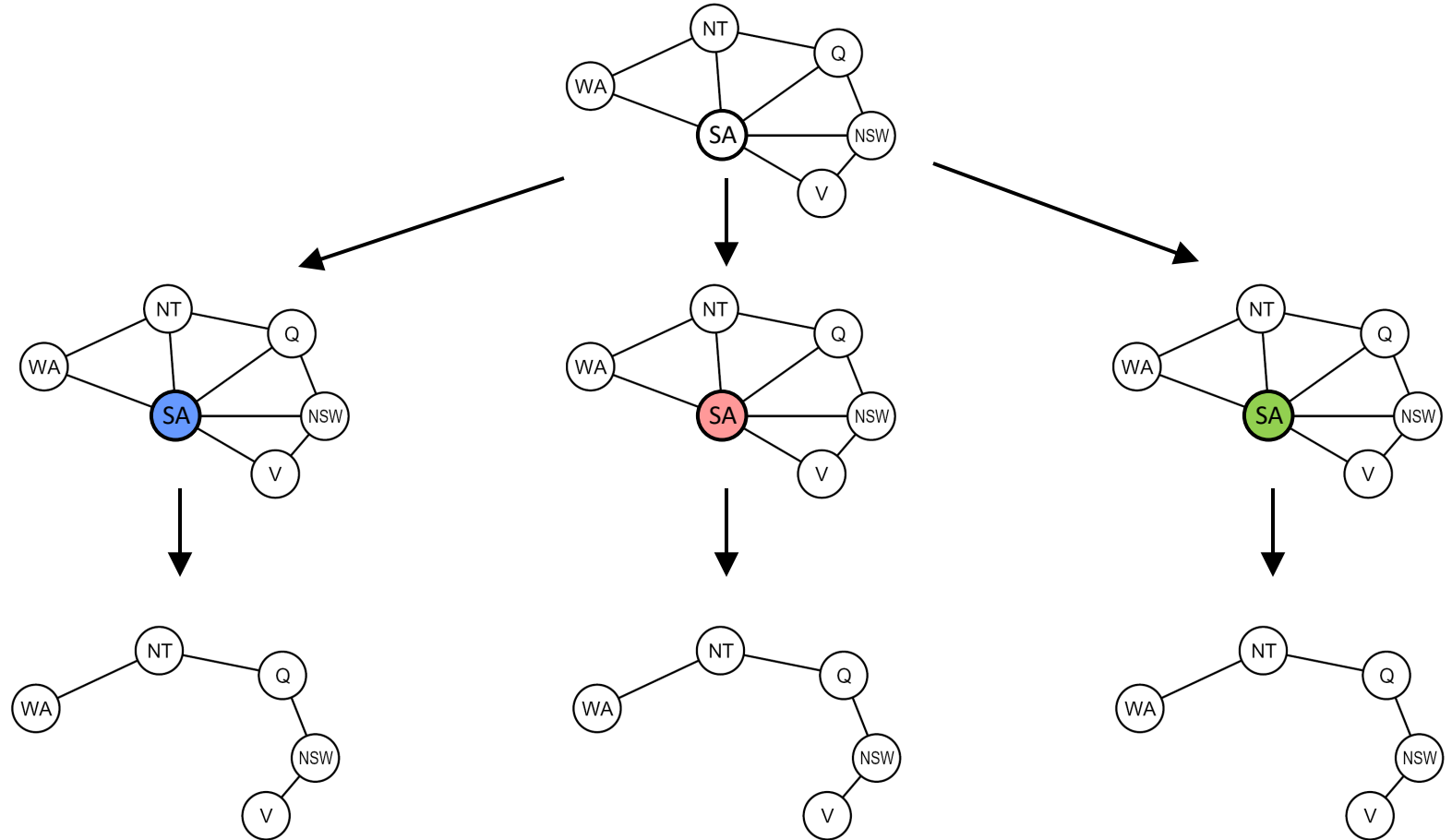
# Cutset Conditioning

Choose a cutset

Instantiate the cutset  
(all possible ways)

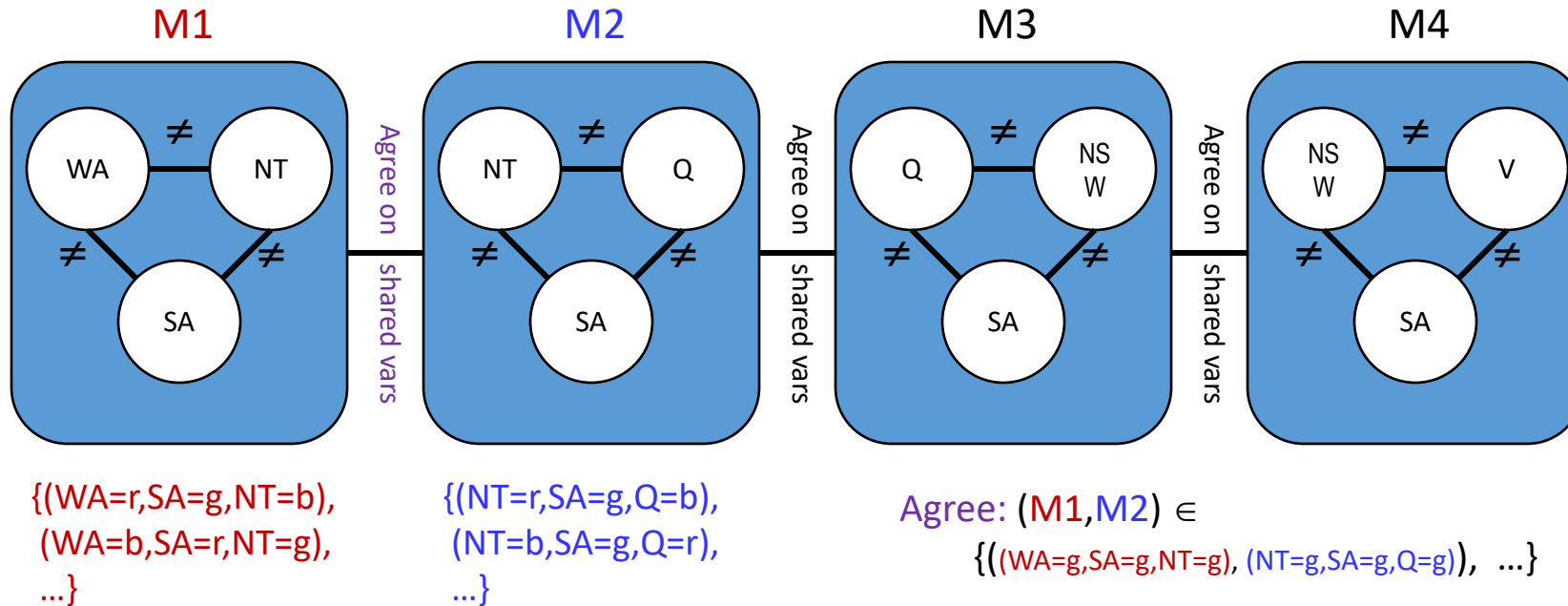
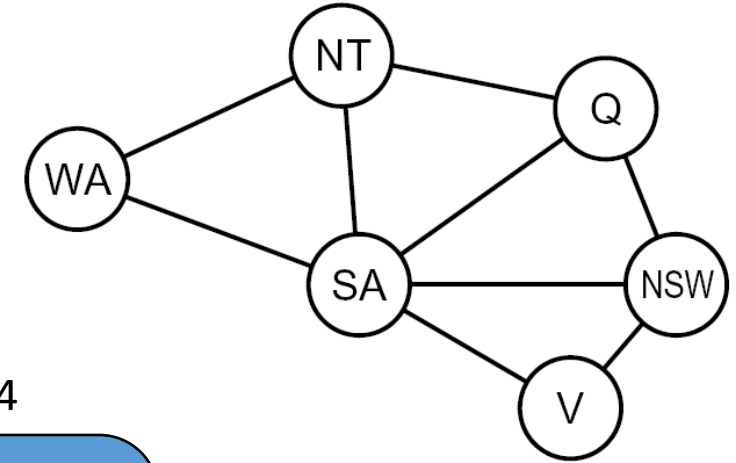
Compute residual CSP  
for each assignment

Solve the residual CSPs  
(tree structured)



# Tree Decomposition\*

- Idea: create a tree-structured graph of mega-variables
- Each mega-variable encodes part of the original CSP
- Subproblems overlap to ensure consistent solutions



# Local Search For CSPs – MIN-CONFLICTS

**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*  $\leftarrow$  an **initial complete assignment** for *csp*

**for**  $i = 1$  to *max\_steps* **do**

**if** *current* is a solution for *csp* **then return** *current*

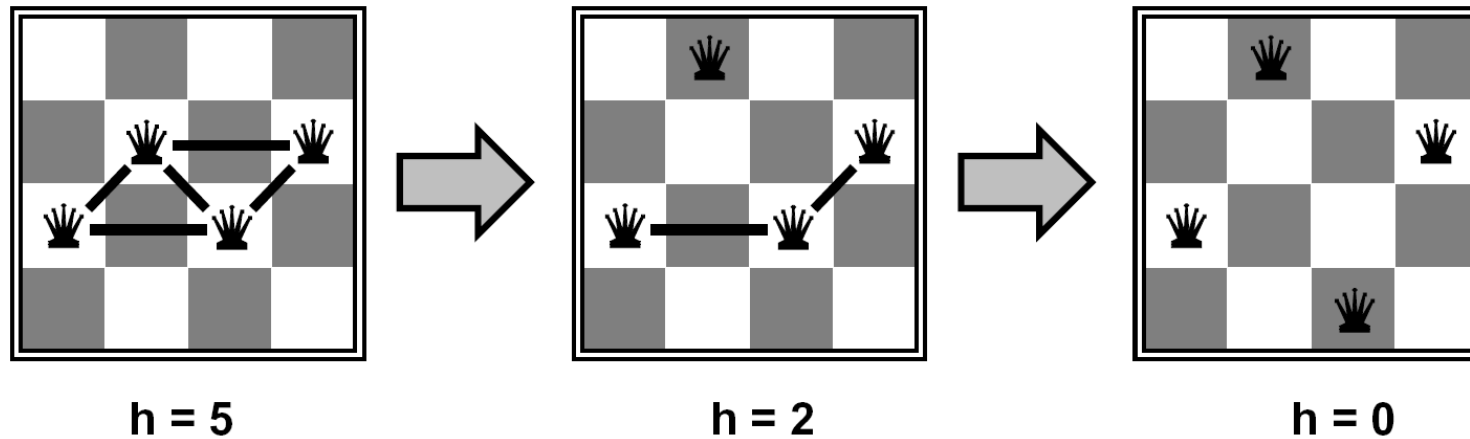
$var \leftarrow$  a **randomly chosen conflicted variable** from *csp*.VARIABLES

$value \leftarrow$  the value  $v$  for *var* that **minimizes CONFLICTS**(*var*,  $v$ , *current*, *csp*)

    set *var* = *value* in *current*

**return** failure

# Example: 4-Queens

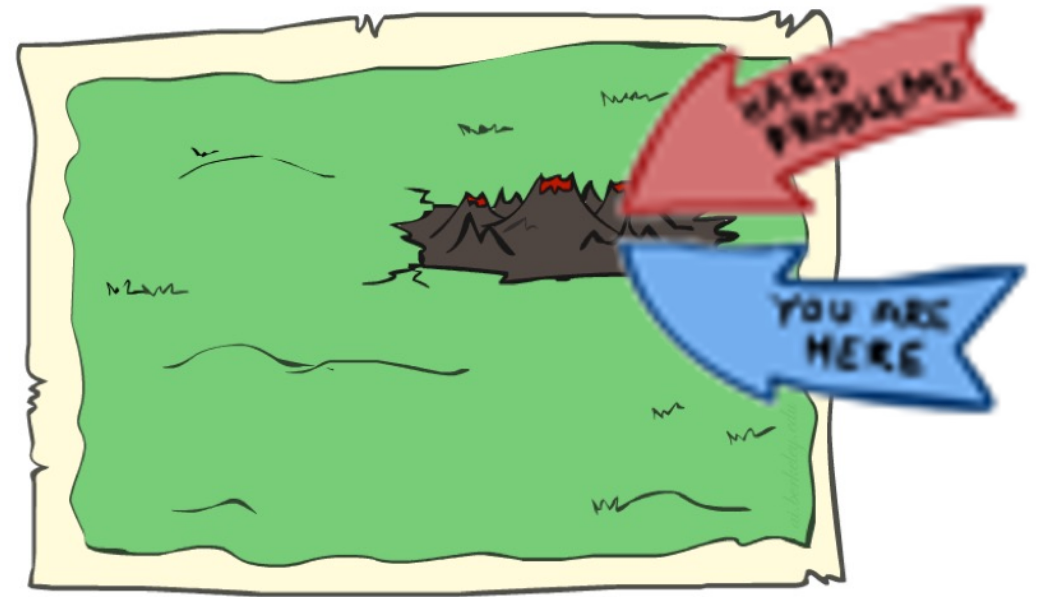
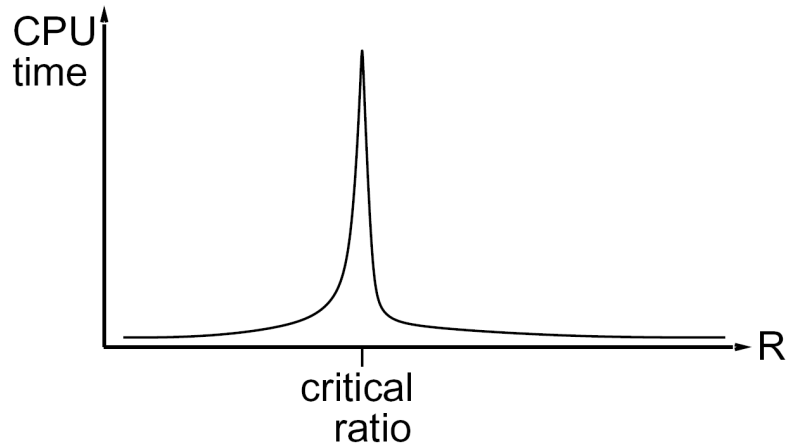


- States: 4 queens in 4 columns ( $4^4 = 256$  states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation:  $c(n)$  = number of attacks

# Performance of Min-Conflicts

- Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., n = 10,000,000)!
- The same appears to be true for any randomly-generated CSP *except* in a narrow range of the ratio

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$



Related to the “Phase Transition” phenomenon

# Summary of CSPs

- CSPs are a special kind of search problem:
  - States are partial assignments
  - Goal test defined by constraints
- Basic solution: backtracking search
- Speed-ups:
  - Ordering
  - Filtering
  - Structure
- Iterative min-conflicts is often effective in practice