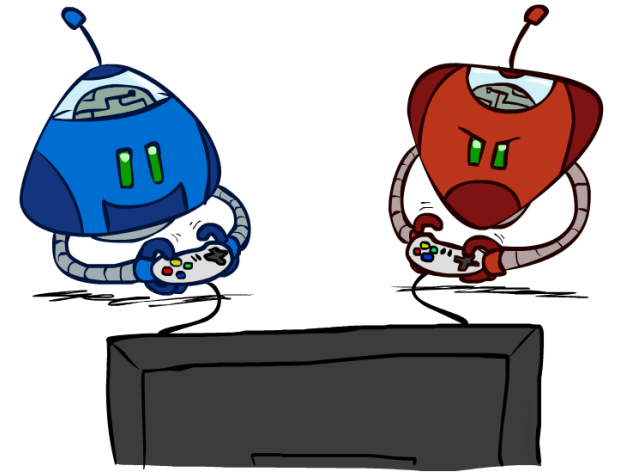


Constraint Satisfaction

CSE 473: Introduction to Artificial Intelligence

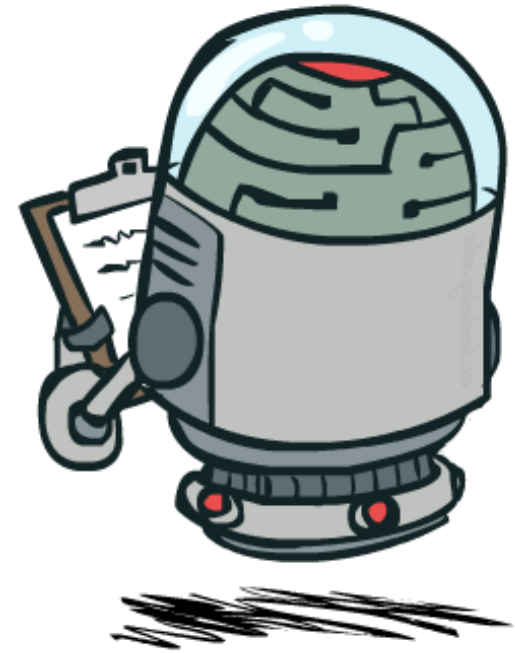


Administrivia

ANNOUNCEMENTS

ASSIGNMENTS

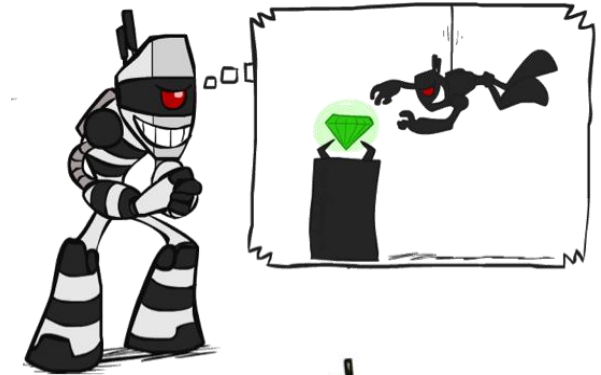
- **Homework 1** due Wednesday
- **Project 1** due next Monday (10.19)



What is Search For?

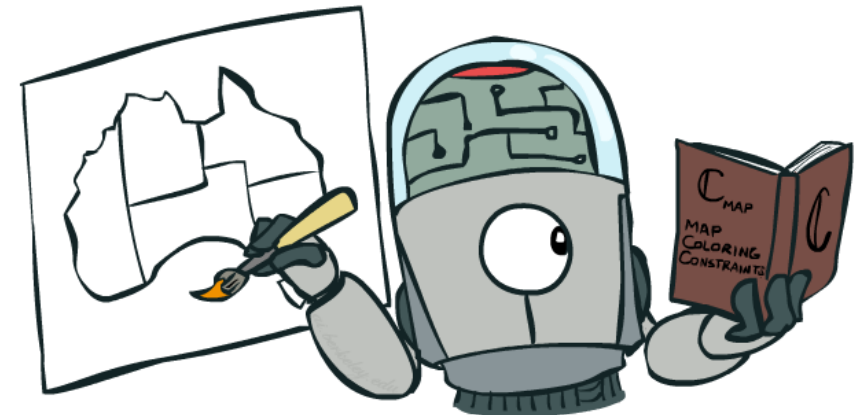
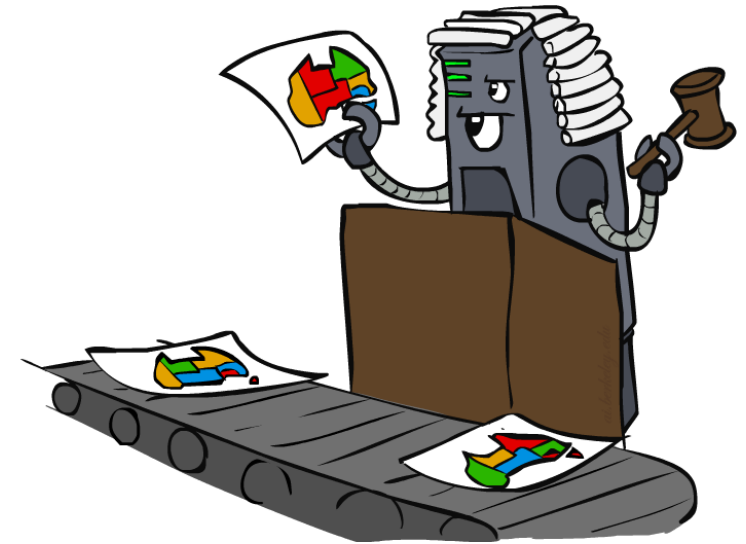
FOUNDATIONS

- Assumptions about the world: a single agent (?), deterministic actions (?), fully observed state, discrete state space
- Planning: sequences of actions
 - The path to the goal is the important thing
 - Paths have various costs, depths
 - Heuristics give problem-specific guidance
 - In games, a path may be a sequence of plays
- Identification: assignments to variables
 - The goal itself is important, not the path
 - All paths at the same depth (for some formulations)
 - CSPs are specialized for identification problems



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_i depends on i)
 - *A partially solved problem*
 - A successor function is assigning a value to a variable
 - Goal test is a **set of constraints** specifying allowable combinations of values for subsets of variables
- Allows useful general-purpose algorithms with more power than standard search algorithms



Example: Map Coloring

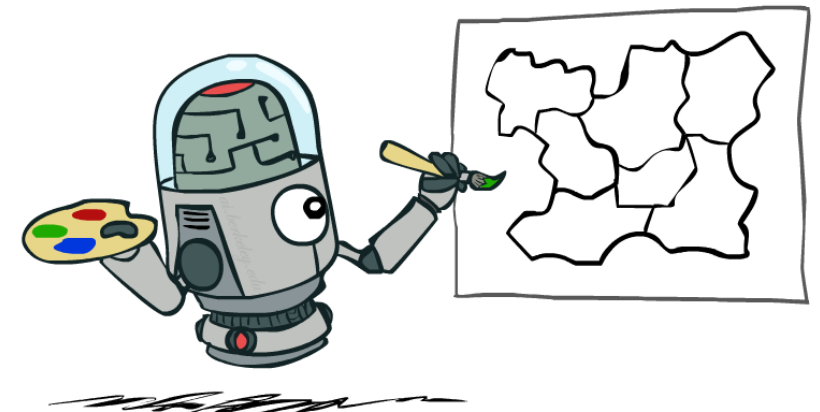
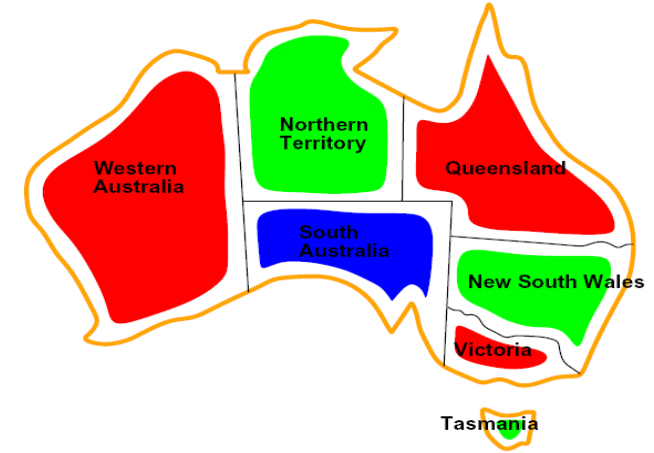
- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: $D = \{\text{red, green, blue}\}$
- Constraints: adjacent regions must have different colors

Implicit: $WA \neq NT$

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

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

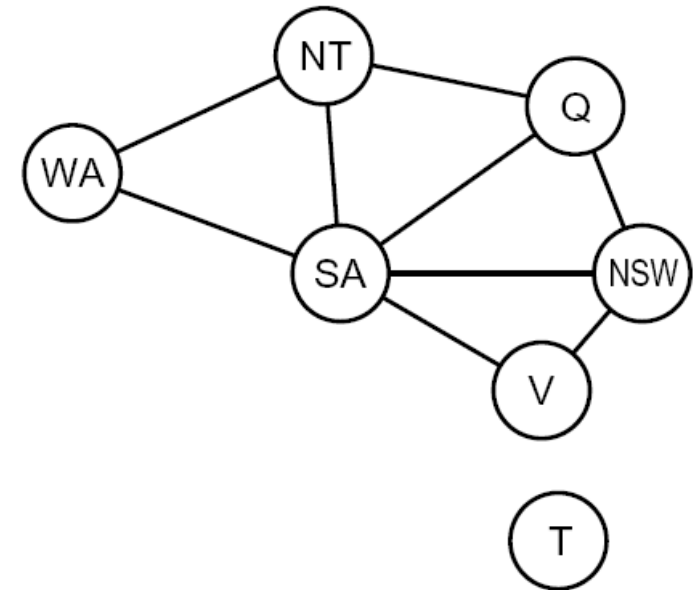
$\{WA=\text{red}, NT=\text{green}, Q=\text{red}, NSW=\text{green}, V=\text{red}, SA=\text{blue}, T=\text{green}\}$



Constraint Graphs

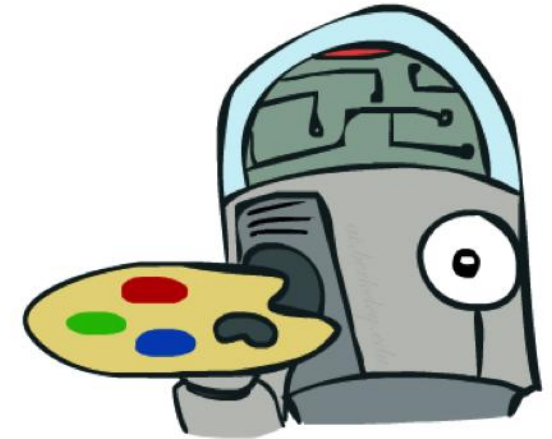


- Binary CSP: each constraint relates (at most) two variables
- Binary constraint graph: nodes are variables, arcs show constraints
- General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent sub-problem!



Varieties of CSPs

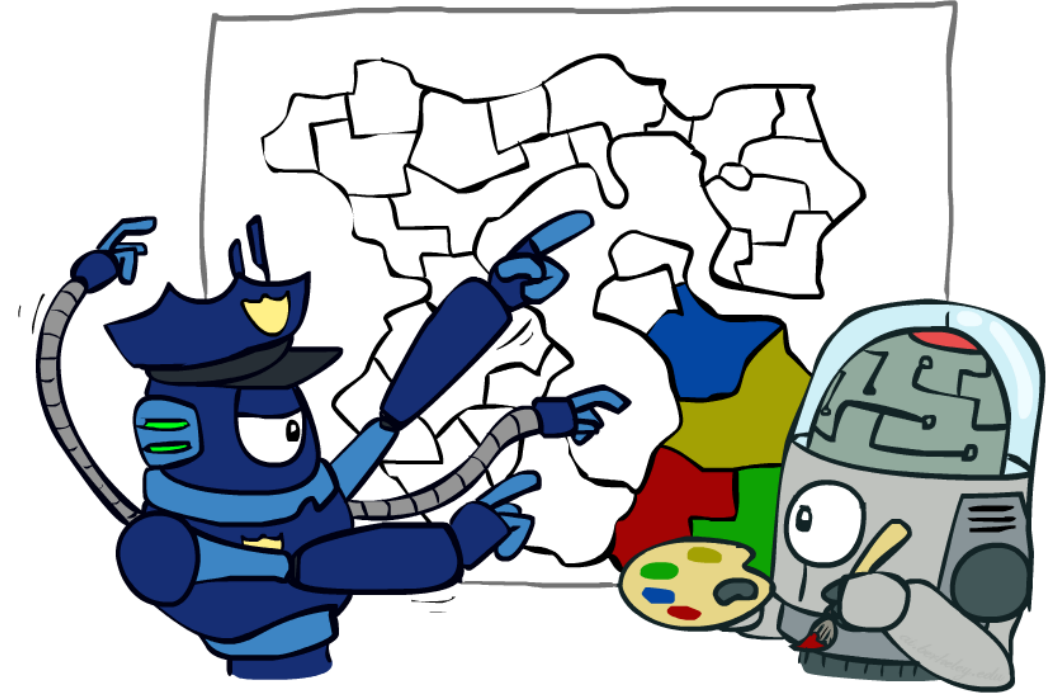
- Discrete Variables
 - Finite domains
 - Size d means $O(d^n)$ complete assignments
 - E.g., Boolean CSPs, including Boolean satisfiability (NP-complete)
 - Infinite domains (integers, strings, etc.)
 - E.g., job scheduling, variables are start/end times for each job
 - Linear constraints solvable, nonlinear undecidable
- Continuous variables
 - E.g., start/end times for Hubble Telescope observations
 - Linear constraints solvable in polynomial time by LP methods (see cs170 for a bit of this theory)



Varieties of Constraints



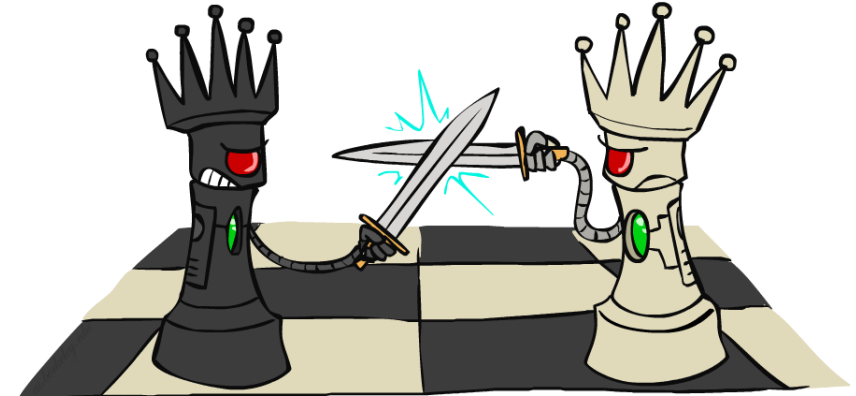
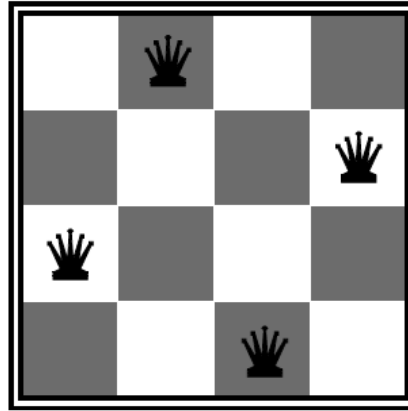
- Varieties of Constraints
 - Unary constraints involve a single variable (equivalent to reducing domains), e.g.:
 $SA \neq \text{green}$
 - Binary constraints involve pairs of variables, e.g.:
 $SA \neq WA$
 - Higher-order constraints involve 3 or more variables:
- Preferences (soft constraints):
 - E.g., red is better than green
 - Often represented by a cost for each variable assignment
 - Gives constrained optimization problems
 - (We'll ignore these until we get to Bayes' nets)



Example: N-Queens

- Formulation 1:

- Variables: X_{ij}
- Domains: $\{0, 1\}$
- Constraints



Same...

domain

row $\forall i, j, k \quad (X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\}$

column $\forall i, j, k \quad (X_{ij}, X_{kj}) \in \{(0, 0), (0, 1), (1, 0)\}$

diagonal $\forall i, j, k \quad (X_{ij}, X_{i+k, j+k}) \in \{(0, 0), (0, 1), (1, 0)\}$

$\forall i, j, k \quad (X_{ij}, X_{i+k, j-k}) \in \{(0, 0), (0, 1), (1, 0)\}$

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

Example: N-Queens (simpler?)

- Formulation 2:

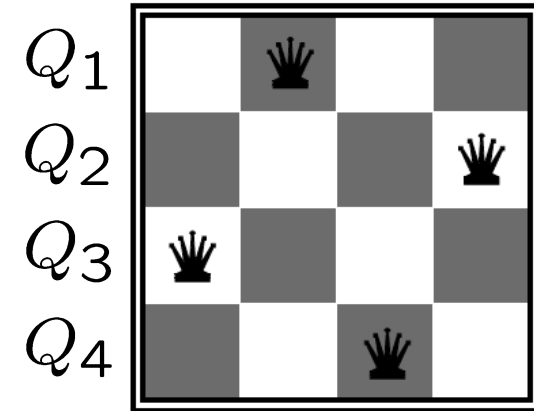
- Variables:

(rows) Q_k

- Domains:

(column in row) $\{1, 2, 3, \dots, N\}$

- Constraints:



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

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

...

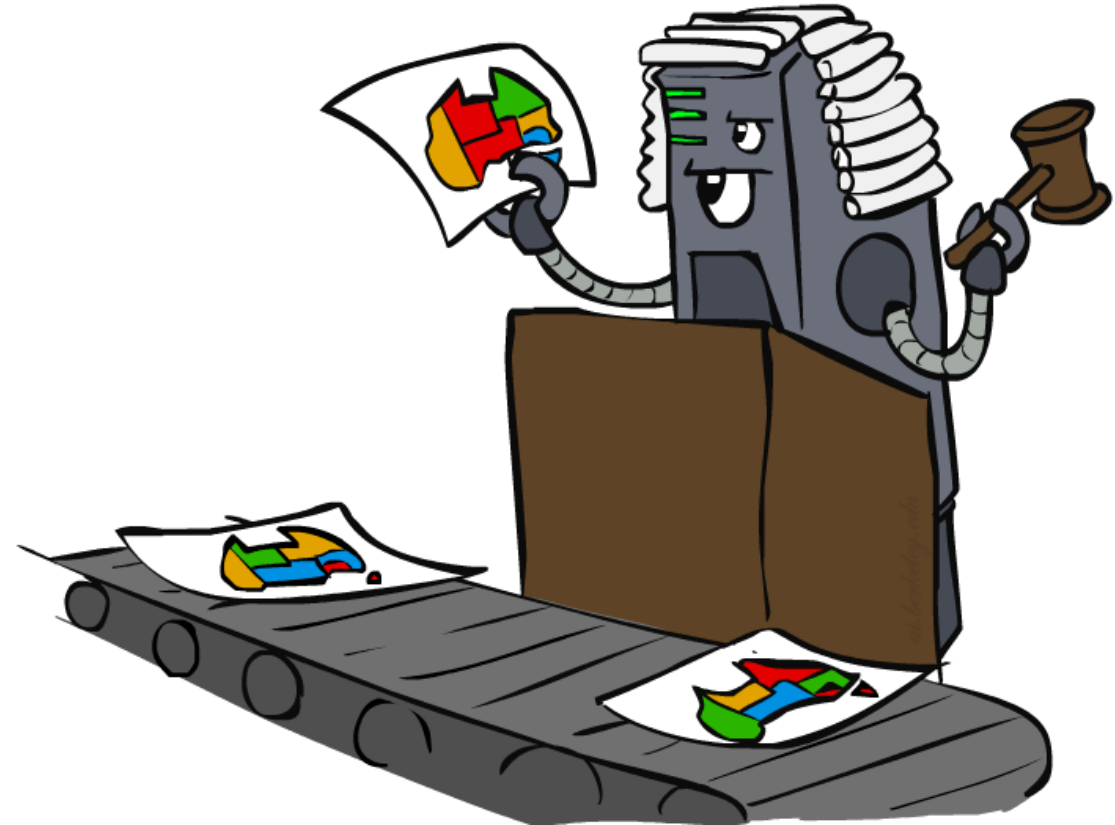
Solving CSPs



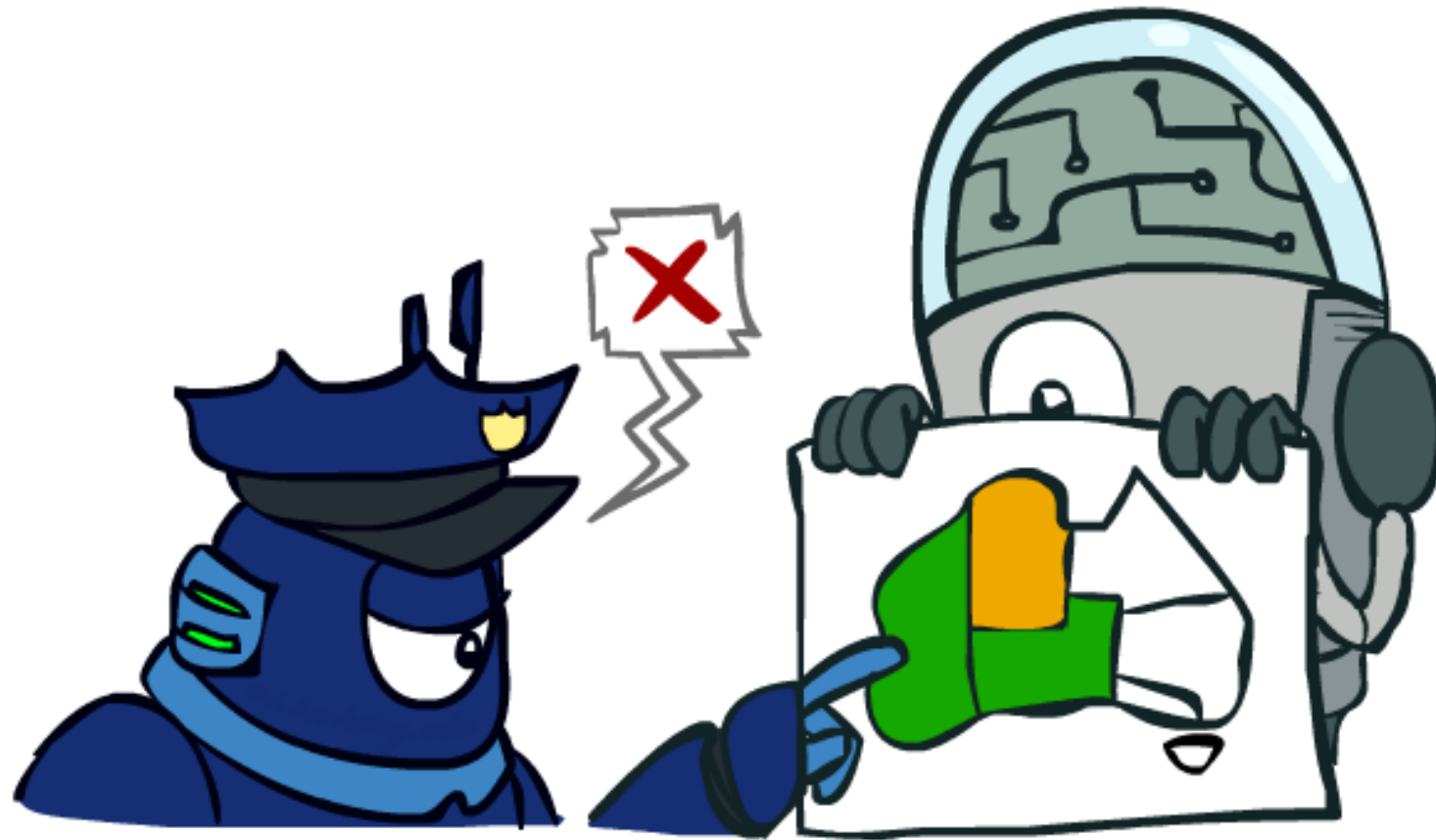
Constraint Satisfaction

Standard Search Formulation

- Standard search formulation of CSPs
- States defined by the values assigned so far (partial assignments)
 - Initial state: the empty assignment, $\{\}$
 - Successor function: assign a value to an unassigned variable
 - Goal test: the current assignment is complete and satisfies all constraints
- We'll start with the straightforward, naïve approach, then improve it



Backtracking Search



-

Backtracking Search Algorithm

```
function BACKTRACKING-SEARCH(csp) returns solution or failure
    return RECURSIVE-BACKTRACKING({}, csp)
function RECURSIVE-BACKTRACKING(assignment, csp) returns solution or 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
            var ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

Improving Backtracking Search

```

function BACKTRACKING-SEARCH(csp) returns solution or failure
    return RECURSIVE-BACKTRACKING({}, csp)
function RECURSIVE-BACKTRACKING(assignment, csp) returns solution or 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
            var ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment

    return failure
  
```

*Which order
should we assign
values to
variables in?*

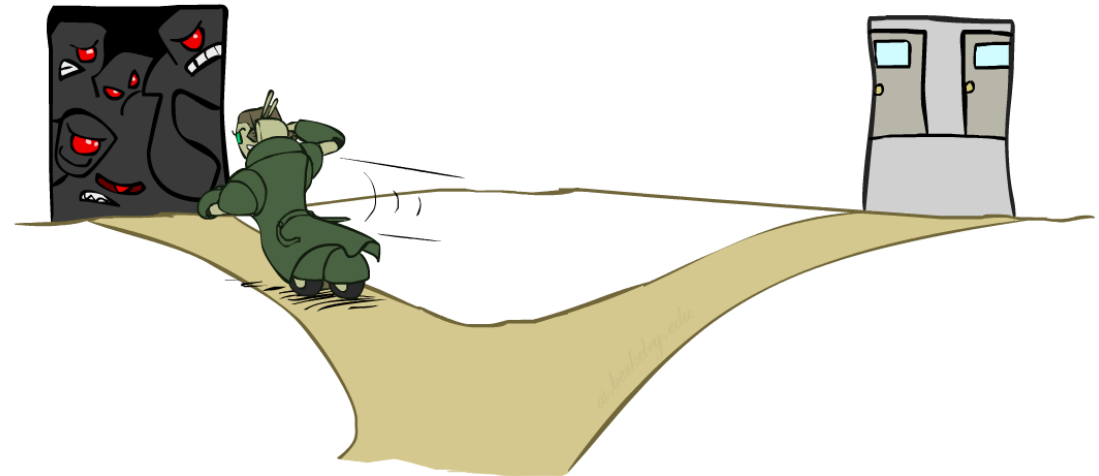
*Can we detect
failure early?*

Ordering: Minimum Remaining Values

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

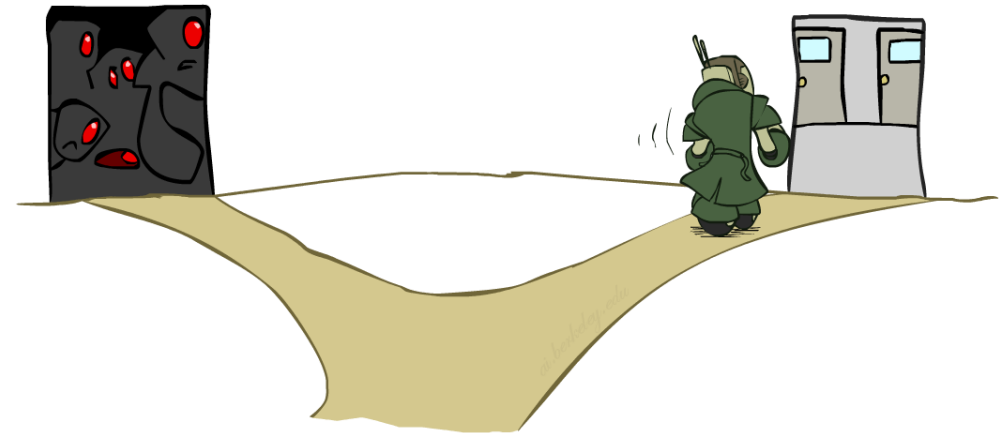
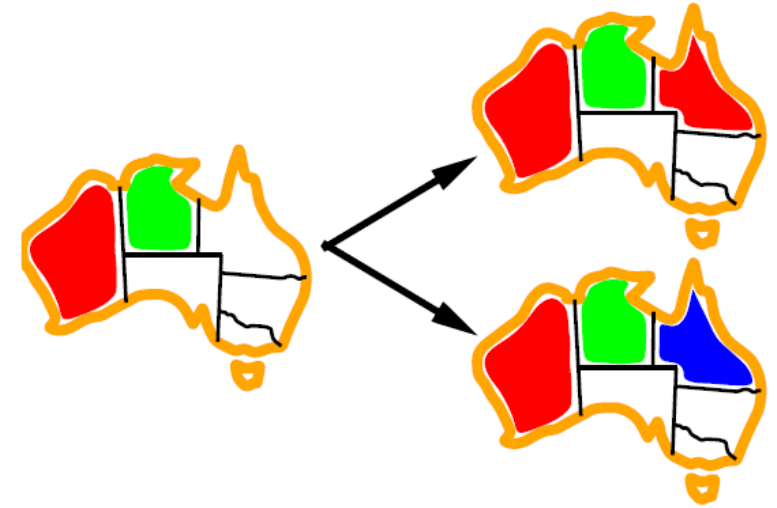


- Why min rather than max?
- Also called “most constrained variable”
- “Fail-fast” ordering



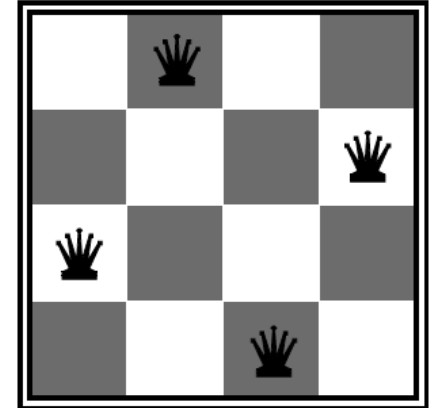
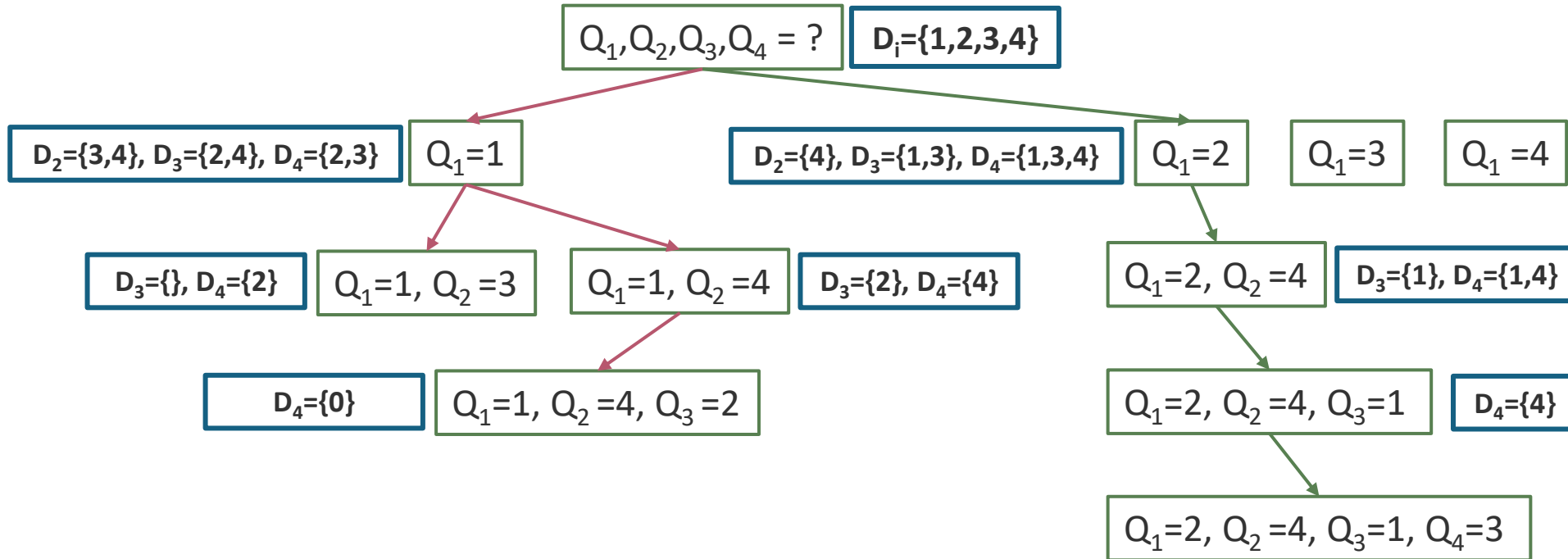
Ordering: Least Constraining Value

- 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)
- Why least rather than most?



Back-tracking tree soln for 4 queens.

EXAMPLE



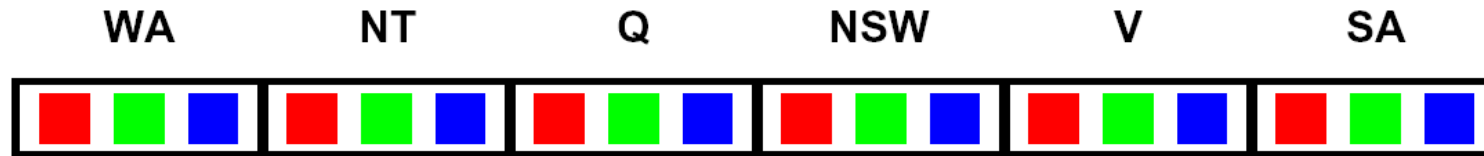
Filtering



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

Filtering: Forward Checking

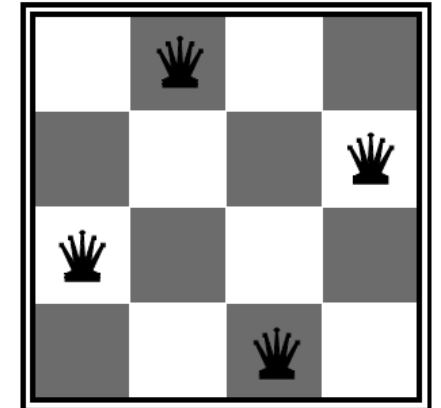
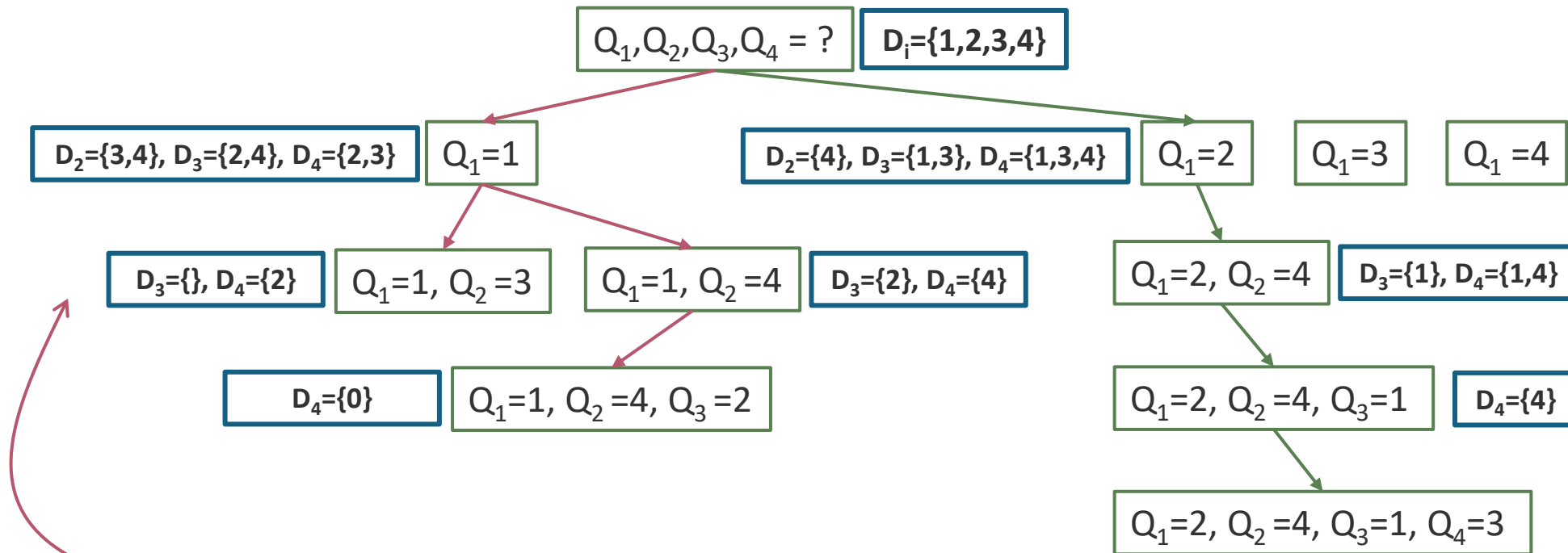
- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



Back-tracking soln using forward checking.



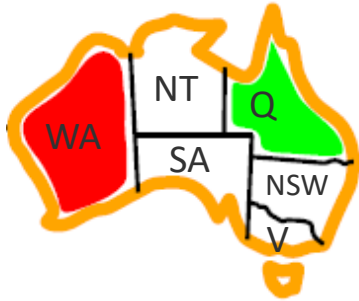
EXAMPLE



Without forward checking would also try $Q_2 = 1, Q_2 = 2$ etc.

Filtering: Constraint Propagation

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



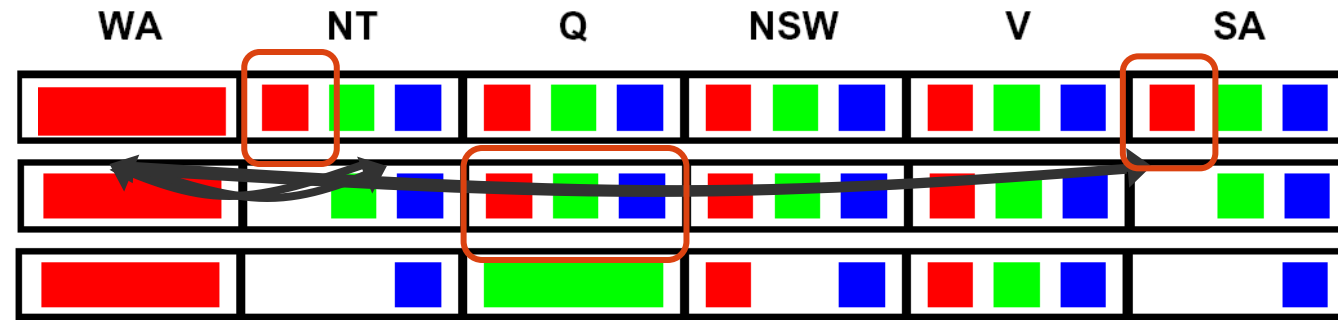
WA	NT	Q	NSW	V	SA

- 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

DEFINITION

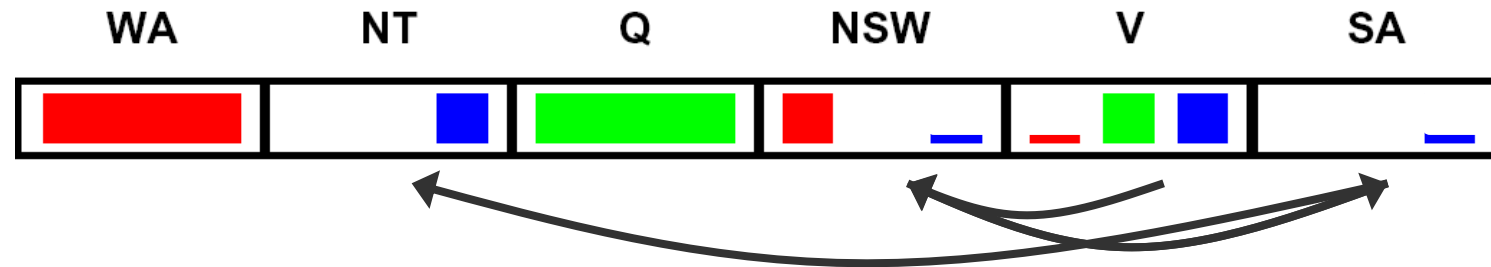
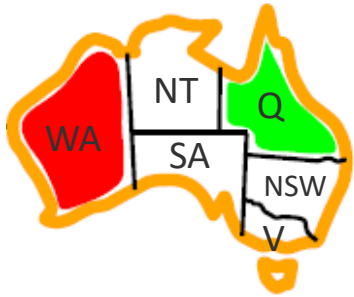
- An arc $X \rightarrow Y$ is **consistent** if and only if for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint



Assign a value to Queensland and run forward checking

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure **all** arcs are consistent
- Let's run arc consistency on this CSP:



- 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?
 - *It's expensive!*

*Remember: Delete
from the tail!*

Enforcing Arc Consistency in a CSP

```

function AC-3(csp) returns false if an inconsistency is found and true otherwise
    queue  $\leftarrow$  a queue of arcs, initially all the arcs in csp
    while queue is not empty do
        (Xi, Xj)  $\leftarrow$  POP(queue)
        if REVISE(csp, Xi, Xj) then
            if sizeof Di = 0 then return false
            for each Xk in Xi.NEIGHBORS – {Xj} do
                queue. PUSH(Xk, Xi)
    return true

function REVISE(assignment, csp) returns true if domain of Xi is revised
    revised  $\leftarrow$  false
    for each x in Di do
        if no y in Dj allows (x, y) to satisfy constraint between (Xi, Xj) then
            delete x from Di
            revised  $\leftarrow$  true
    return revised
  
```

Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$ (outer loop might add an arc back to the queue at most d times if poorly ordered)
 ... but detecting all possible future problems is NP-hard – why?

Arc consistency for 4 queens.

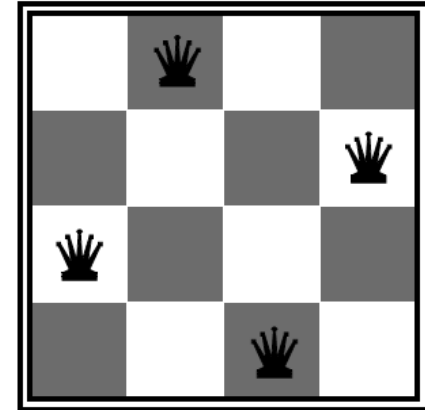
EXAMPLE

$Q_1, Q_2, Q_3, Q_4 = ?$

$D_i = \{1, 2, 3, 4\}$

$D_2 = \{3, 4\}, D_3 = \{2, 4\}, D_4 = \{2, 3\}$

$Q_1 = 1$



All the arcs

$Q_1 \rightarrow Q_2$	$Q_1 \rightarrow Q_3$	$Q_1 \rightarrow Q_4$
$Q_2 \rightarrow Q_1$	$Q_2 \rightarrow Q_3$	$Q_2 \rightarrow Q_4$
$Q_3 \rightarrow Q_1$	$Q_3 \rightarrow Q_2$	$Q_3 \rightarrow Q_4$
$Q_4 \rightarrow Q_2$	$Q_4 \rightarrow Q_3$	$Q_4 \rightarrow Q_1$

Starting Domains

$D_1 = \{1, 2, 3, 4\}$
 $D_2 = \{1, 2, 3, 4\}$
 $D_3 = \{1, 2, 3, 4\}$
 $D_4 = \{1, 2, 3, 4\}$

Filtered Domains

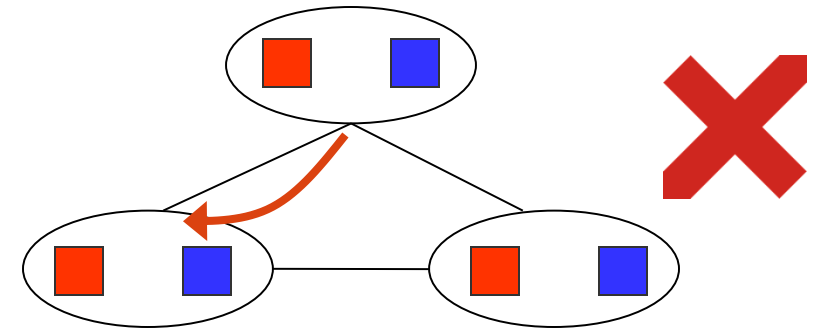
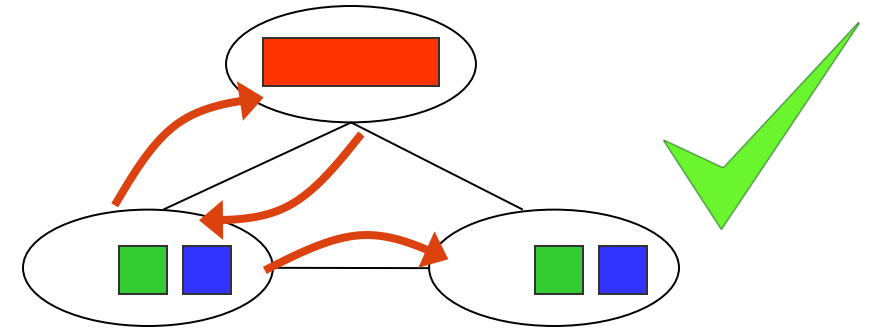
$D_1 = \{1\}$
 $D_2 = \{3, 4\}$
 $D_3 = \{2, 4\}$
 $D_4 = \{2, 3\}$

AC-3 Domains

$D_1 = \{1\}$
 $D_2 = \{3, 4\}$
 $D_3 = \{2\}$
 $D_4 = \{\}$

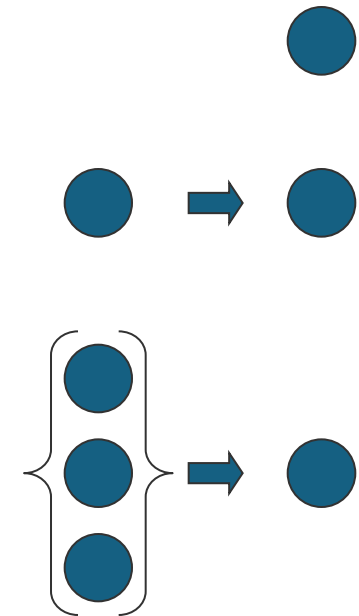
Limitations of Arc Consistency

- 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 may still run inside a backtracking search!



Solution: 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
 - 3-Consistency (Path Consistency iff binary): For every set of three nodes, any consistent assignment to two can be extended to the third
 - 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)



Strong K-Consistency

- Strong k-consistency: also k-1, k-2, ... 1 consistent
- Claim: strong n-consistency means we can solve without backtracking!
 - (where n is the size of entire CSP)
- Why?
 - Choose any assignment to any variable
 - Choose a new variable
 - By 2-consistency, there is a choice consistent with the first
 - Choose a new variable
 - By 3-consistency, there is a choice consistent with the first 2
 - ...
- Lots of middle ground for algorithms between arc consistency and n-consistency!
 - (e.g. k=3, called path consistency)

Runtime?

- After enforcing consistency must enumerate values
... $O(n^2 d)$
- Enforcing strong-k consistency is NP-complete
... $O(n^k d^k)$

Iterative Algorithms for CSPs

- Local search methods typically work with “complete” states, i.e., all variables assigned
- To apply to CSPs:
 - Take an assignment with unsatisfied constraints
 - Operators *reassign* variable values
 - No fringe! Live on the edge.
- Algorithm: While not solved,
 - Variable selection: randomly select any conflicted variable
 - Value selection: min-conflicts heuristic:
 - Choose a value that violates the fewest constraints
 - I.e., hill climb with $h(x)$ = total number of violated constraints



that's it for today!

SUMMARY

- Constraint Satisfaction Problems
- Solve via DFS with improvements
- But other solutions, such as global search algorithms, can be applied

UPCOMING

- Modeling Stochasticity
- Reinforcement Learning

REMINDERS

- Do **Project 1**