Recap: Tree Search

- Traverse all reachable nodes from the start until finding the goal

```
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
```

Recap: Graph Search

- Traverse all reachable nodes from the start until finding the goal
- Closed set: avoid expanding a state more than once

```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

Recap: A*

- A* expands the fringe node with lowest f value where
  - $f(n) = g(n) + h(n)$
  - $g(n)$ is the cost to reach n
  - $h(n)$ is an admissible estimate of the least cost from n to a goal node: $0 \leq h(n) \leq h^*(n)$
- A* tree search is optimal
- Its performance depends heavily on the heuristic h

Recap:“Good” Heuristics

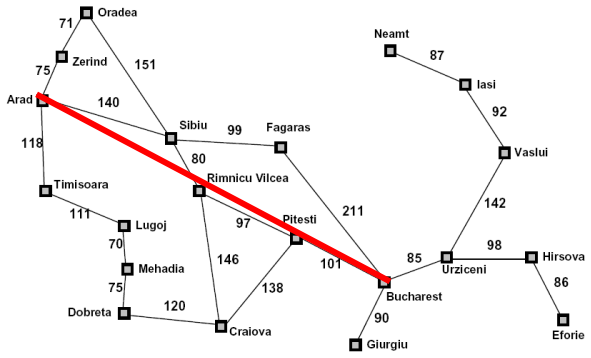
- Main idea: estimated heuristic costs ≤ actual costs
  - **Admissibility**: heuristic cost-to-goal ≤ actual cost-to-goal

    h(A) ≤ actual cost from A to G
  - **Consistency**: heuristic “arc” cost ≤ actual arc cost

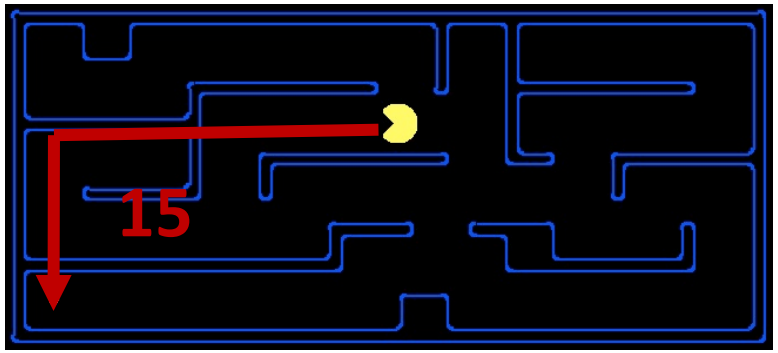
    h(A) – h(C) ≤ cost(A to C)

Recap: Creating Admissible Heuristics

- Often, admissible heuristics are solutions to ***relaxed problems***, where new actions are available



366



- Problem P_2 is a relaxed version of P_1 if $\mathcal{A}_2(s) \supseteq \mathcal{A}_1(s)$ for every s
- Theorem: $h_2^*(s) \leq h_1^*(s)$ for every s, so $h(s) = h_2^*(s)$ is admissible for P_1

Recap: Combining heuristics

- Dominance: $h_1 \geq h_2$ if

  $$\forall n \;\; h_1(n) \geq h_2(n)$$

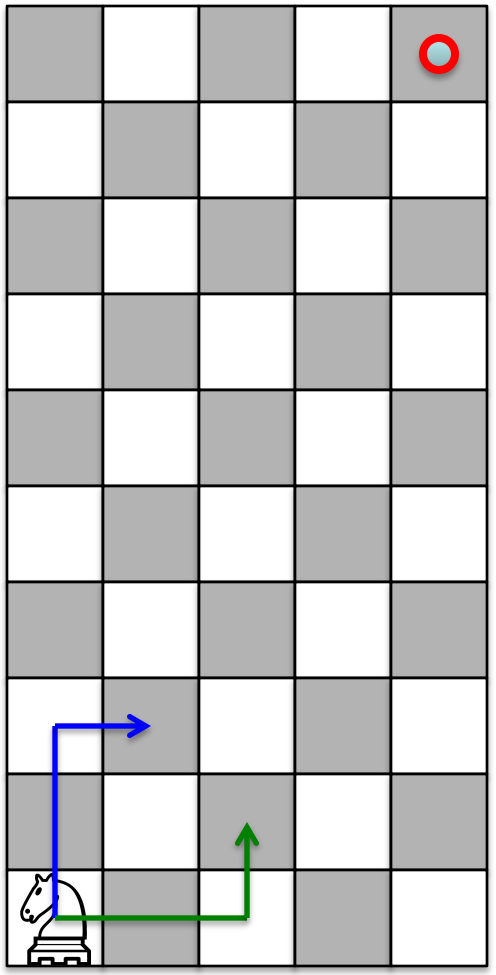
  - Roughly speaking, larger is better (tighter) as long as both are admissible
  - The zero heuristic is pretty bad (what does A* do with h=0?)
  - The exact heuristic is pretty good, but usually too expensive!
- What if we have two heuristics, neither dominates the other?
  - Form a new heuristic by taking the max of both:

    $$h(n) = \max(\, h_1(n), h_2(n))$$

  - Max of admissible heuristics is admissible and dominates both!

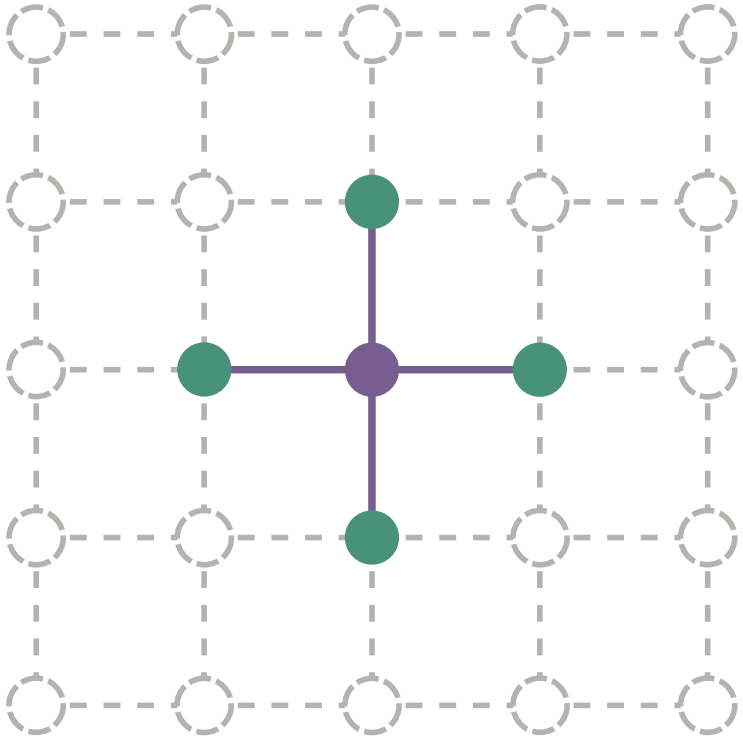
Example: Knight's moves

- Minimum number of knight's moves to get from A to B?
  - h_1 = (Manhattan distance)/3
    - h_1' = h_1 rounded up to correct parity (even if A, B same color, odd otherwise)
  - h_2 = (Euclidean distance)/√5 (rounded up to correct parity)
  - h_3 = (max x or y shift)/2 (rounded up to correct parity)
- $h(n) = \max(h_1'(n), h_2(n), h_3(n))$ is admissible!



Quiz: State Space Graphs vs. Search Trees

Consider a rectangular grid:



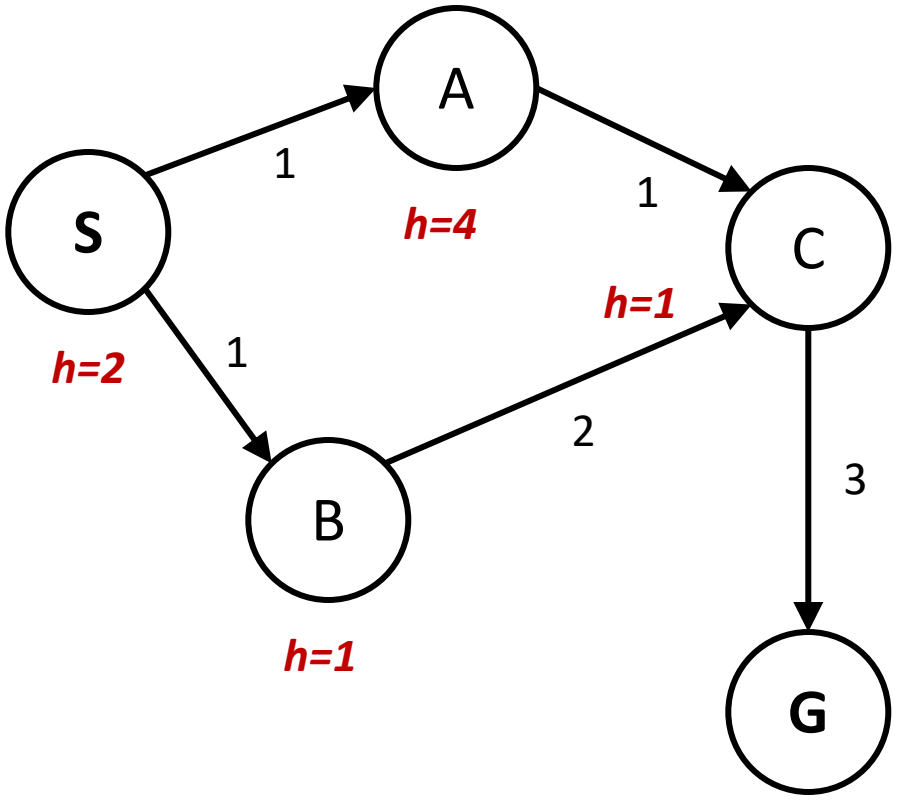
How many states within *d* steps of start?

How many states in search tree of depth *d*?

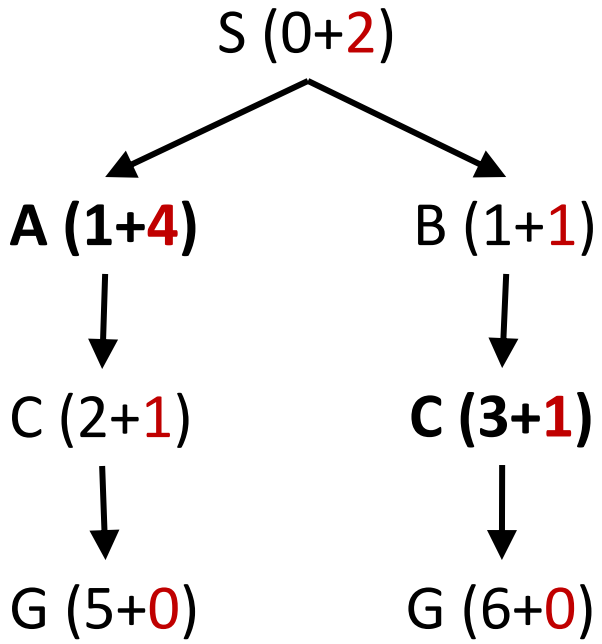
Basic idea of graph search: don't re-expand a state that has been expanded previously

Recap: Graph Search Gone Wrong

State space graph

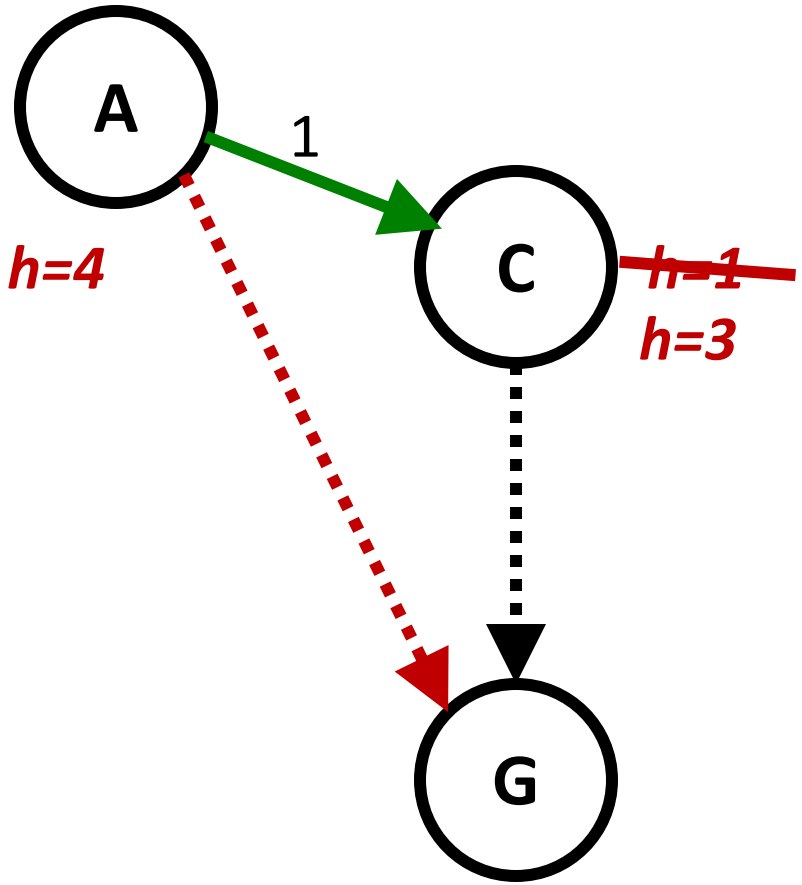


Search tree



With consistency: f(A) ≤ f(optimal C) ≤ f(suboptimal C)

Recap: Consistency of Heuristics



- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal

    $h(A) \leq h^*(A)$
  - **Consistency: heuristic "arc" cost ≤ actual arc cost**

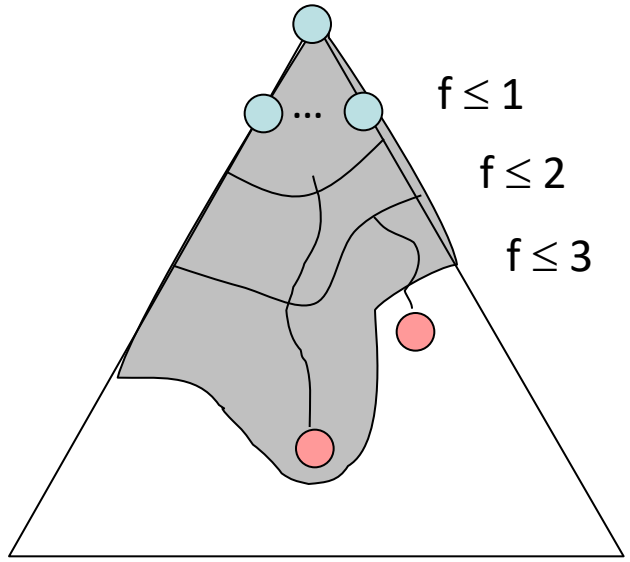
    $h(A) - h(C) \leq c(A,C)$

    or $h(A) \leq c(A,C) + h(C)$ (triangle inequality)
    - Note: h* *necessarily* satisfies triangle inequality
- Consequences of consistency:
  - **The *f* value along a path never decreases:**

    $h(A) \leq c(A,C) + h(C) \Rightarrow g(A) + h(A) \leq g(A) + c(A,C) + h(C)$
  - Nodes along the optimal path can never be **blocked** by the closed set.
  - A* graph search is optimal

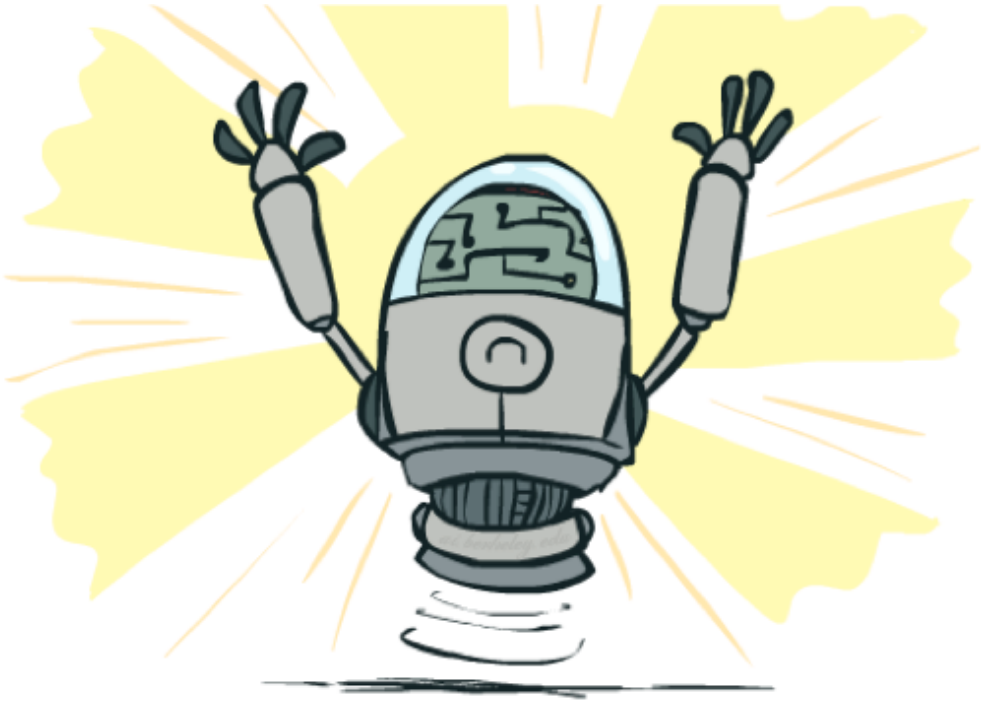
Optimality of A* Graph Search

- Sketch: consider what A* does with a consistent heuristic:
  - Fact 1: In tree search, A* expands nodes in increasing total f value (f-contours)
  - Fact 2: For **every state s** (not only G), nodes that reach s optimally are expanded before nodes that reach s suboptimally
  - Result: blocking will never happen, A* graph search is optimal



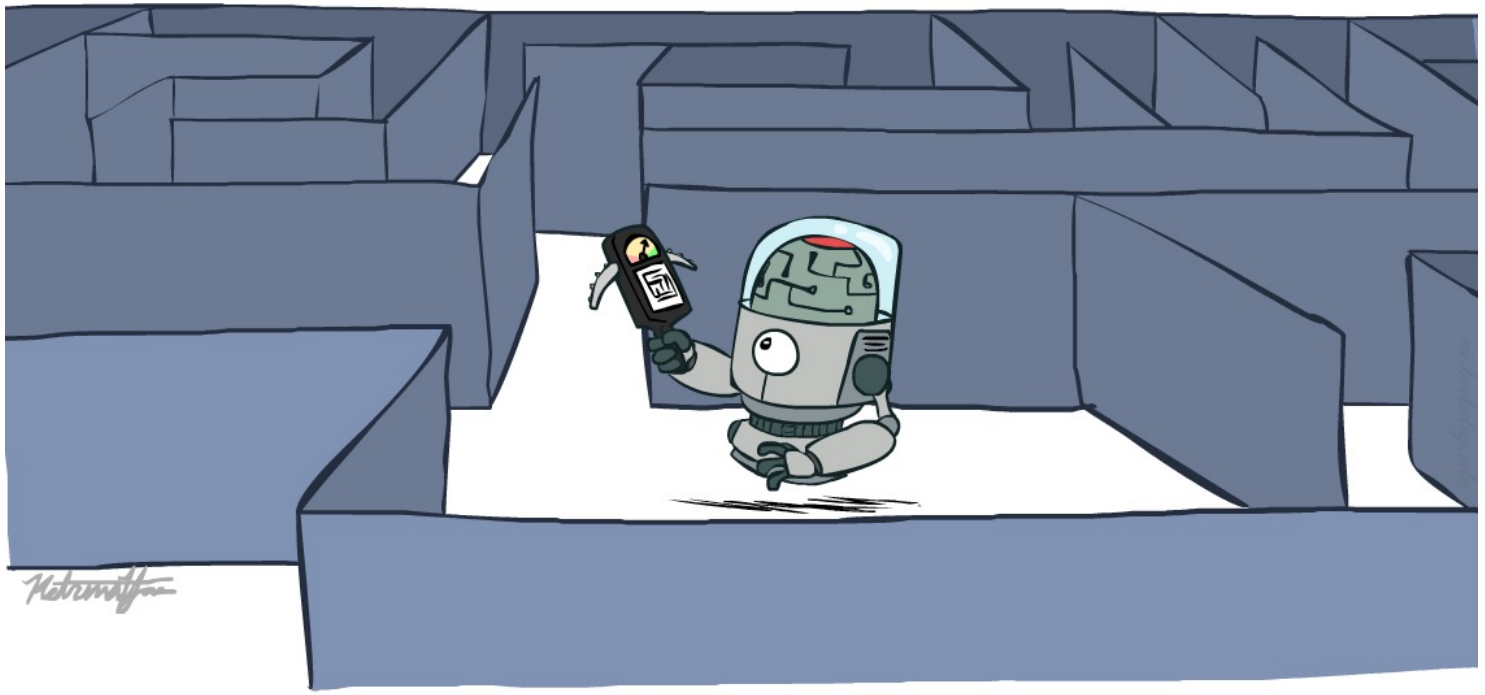
Recap: Summary of A*

- Tree search:
  - A* is optimal if heuristic is admissible
- Graph search:
  - A* optimal if heuristic is consistent
- Consistency implies admissibility
- Most natural admissible heuristics tend to be consistent, especially if from relaxed problems



CS 3317: Artificial Intelligence

Constraint Satisfaction Problems



Instructors: **Cai Panpan**

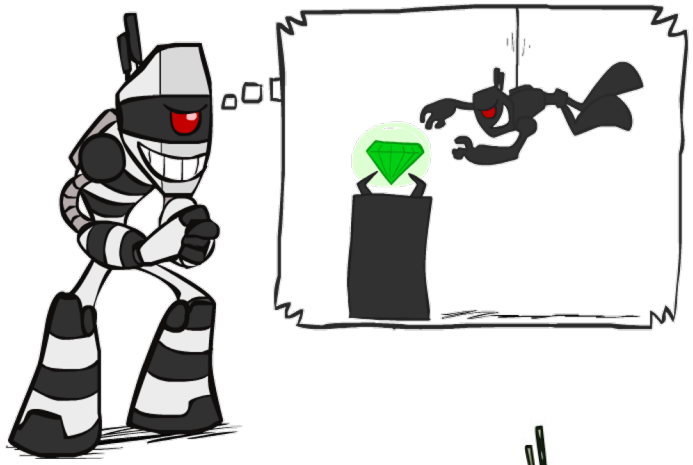
Shanghai Jiao Tong University

(slides adapted from UC Berkeley CS188)

What is Search For?

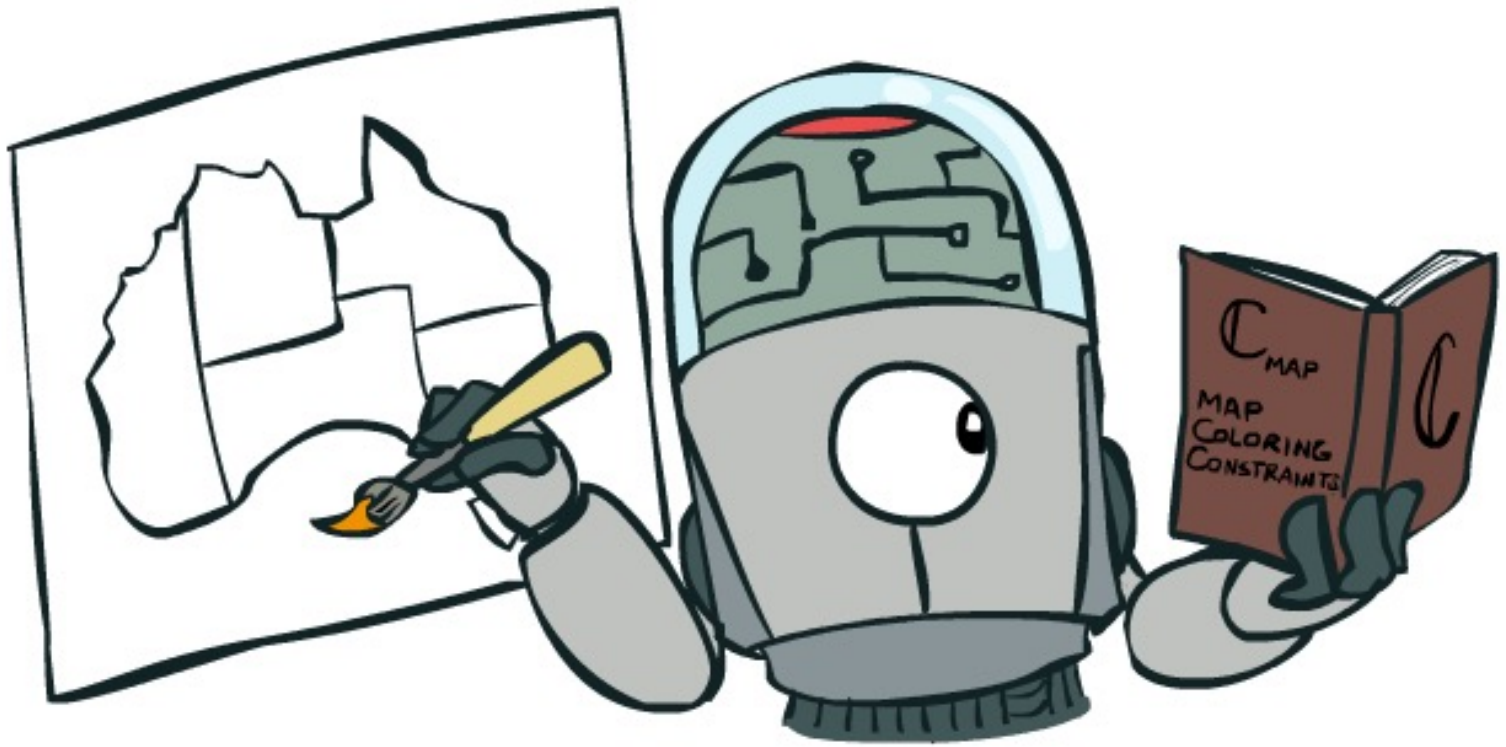
- Assumptions about the world: a single agent, deterministic actions, fully observed state, discrete state space

- Planning: sequences of actions
  - The path to the goal is the important thing
  - Paths have various costs, depths
  - Heuristics give problem-specific guidance



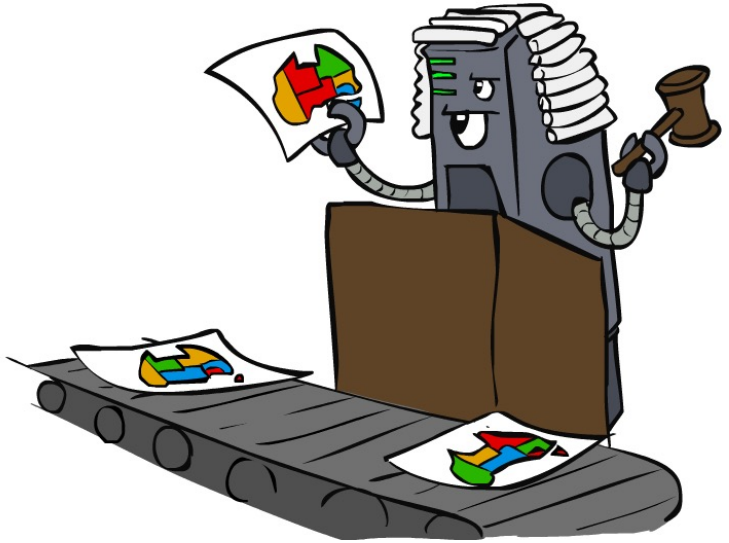
- Identification: assignments to variables
  - The goal itself is important, not the path
  - All paths at the same depth (for some formulations)
  - CSPs are a specialized class of identification problems

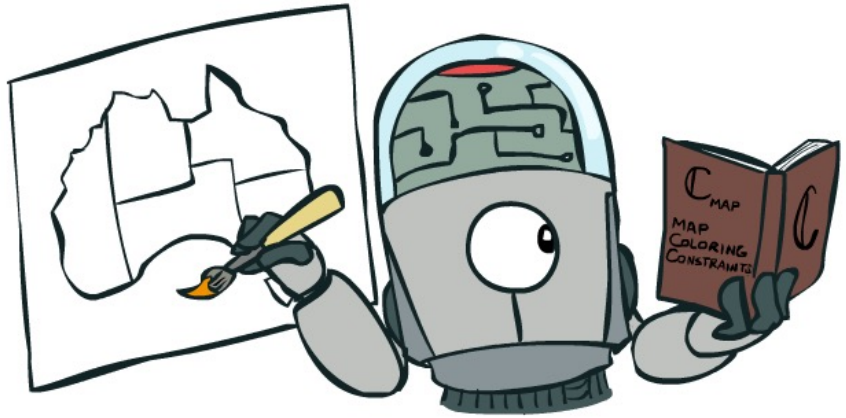
Constraint Satisfaction Problems



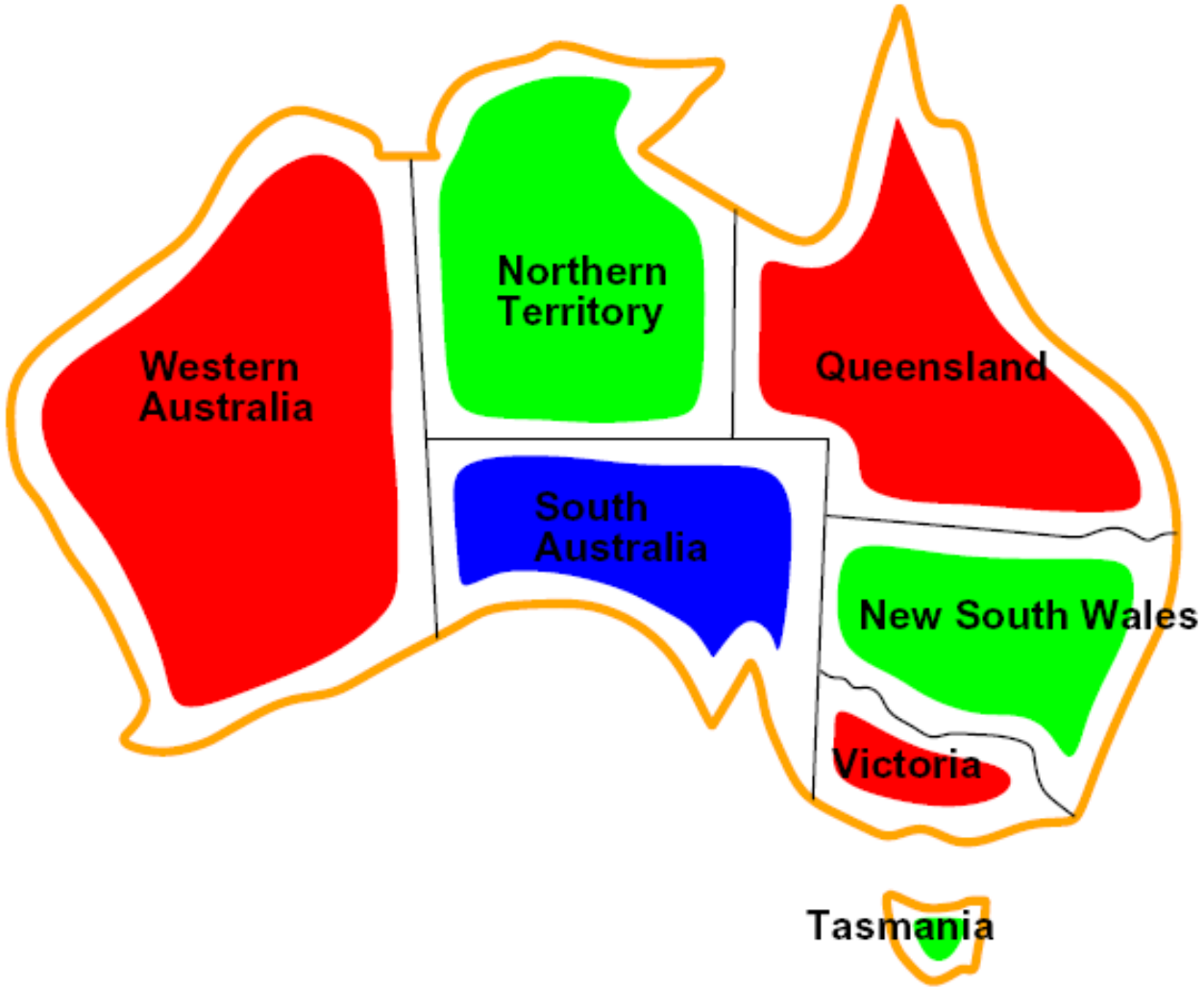
Constraint Satisfaction Problems

- 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 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



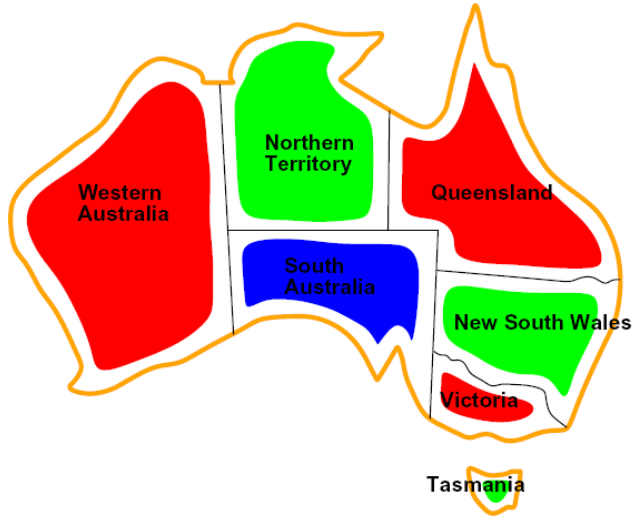
Example: Map Coloring

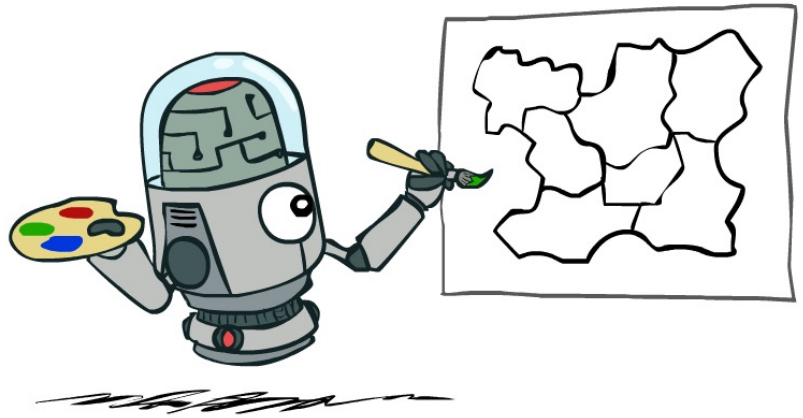
- Variables: $\mathrm{WA},\ \mathrm{NT},\ \mathrm{Q},\ \mathrm{NSW},\ \mathrm{V},\ \mathrm{SA},\ \mathrm{T}$
- Domains: $\mathrm{D} = \{\mathrm{red}, \mathrm{green}, \mathrm{blue}\}$
- Constraints: adjacent regions must have different colors

  Implicit: $\mathrm{WA} \neq \mathrm{NT}$

  Explicit: $(\mathrm{WA}, \mathrm{NT}) \in \{(\mathrm{red}, \mathrm{green}), (\mathrm{red}, \mathrm{blue}), \ldots\}$
- Solutions are assignments satisfying all constraints, e.g.:

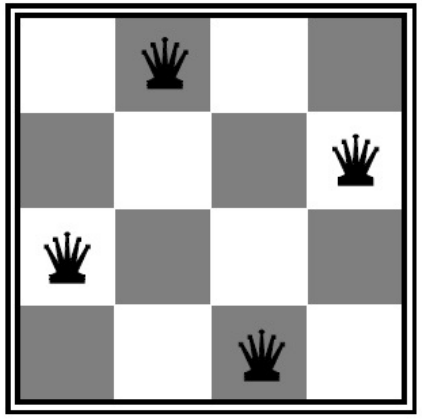
  $\{\mathrm{WA}{=}\mathrm{red},\ \mathrm{NT}{=}\mathrm{green},\ \mathrm{Q}{=}\mathrm{red},\ \mathrm{NSW}{=}\mathrm{green},$
  $\mathrm{V}{=}\mathrm{red},\ \mathrm{SA}{=}\mathrm{blue},\ \mathrm{T}{=}\mathrm{green}\}$

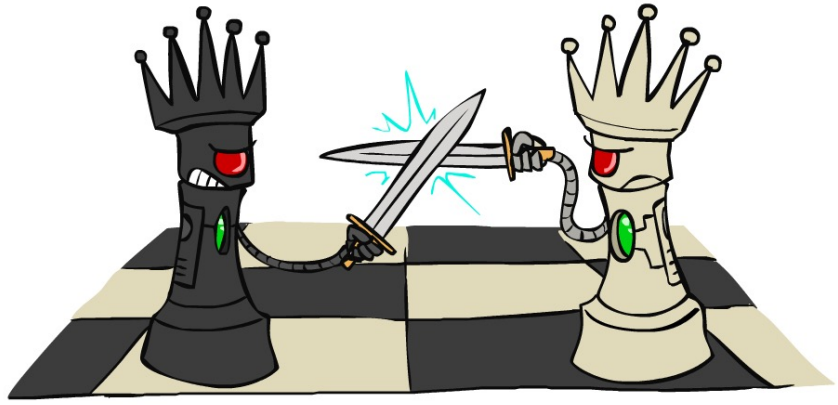




Example: N-Queens

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



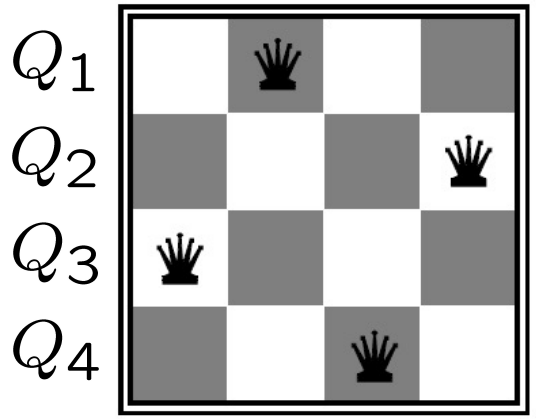


$$\forall i, j, k \quad (X_{ij}, X_{ik}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{kj}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{i+k,j+k}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{i+k,j-k}) \in \{(0,0), (0,1), (1,0)\}$$

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

Example: N-Queens

- Formulation 2:
  - Variables: Q_k
  - Domains: $\{1, 2, 3, \ldots N\}$
  - Constraints:

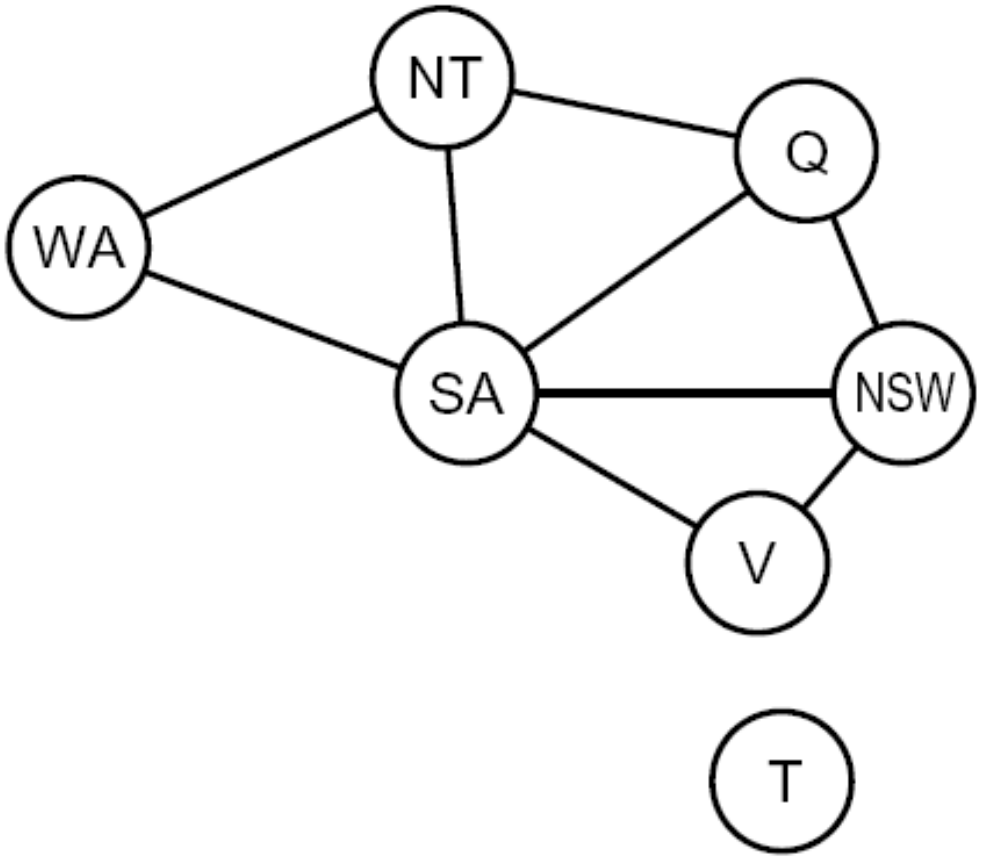


Implicit: $\forall i, j$ non-threatening(Q_i, Q_j)

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

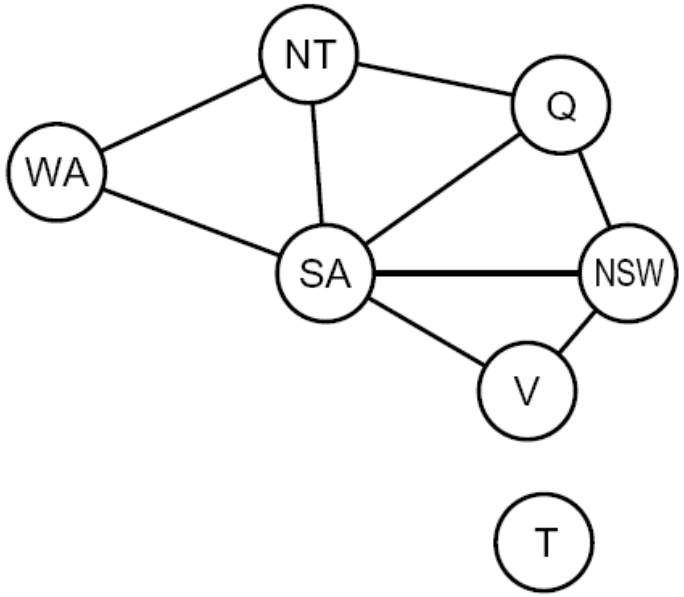
$\ldots$

Constraint Graphs



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 analyze the problem and speed up search.
  - E.g., Tasmania is an independent subproblem!



Example: Cryptarithmetic

- Variables:

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

- Domains:

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

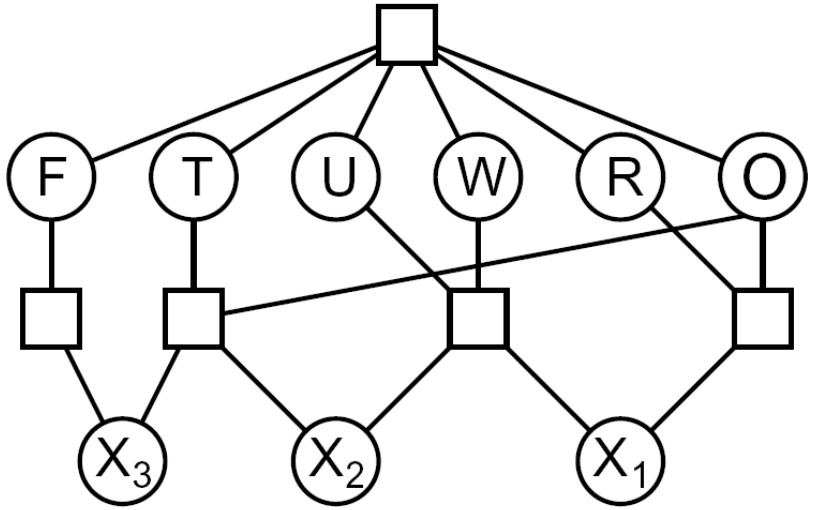
- Constraints:

  alldiff(F, T, U, W, R, O)

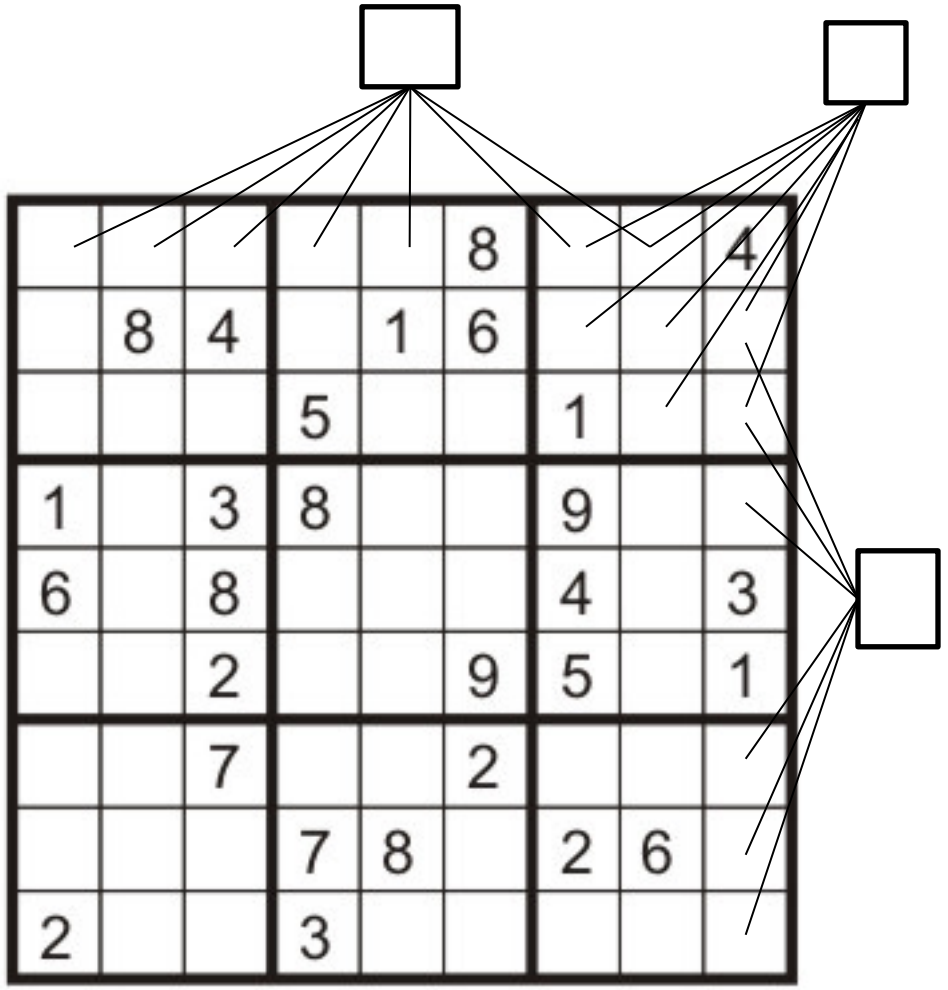
  $O + O = R + 10 \cdot X_1$

  $\ldots$

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



Example: Sudoku



- Variables:
  - Each (empty) cell
- Domains:
  - {1,2,…,9}
- Constraints:

  9-way alldiff for each column

  9-way alldiff for each row

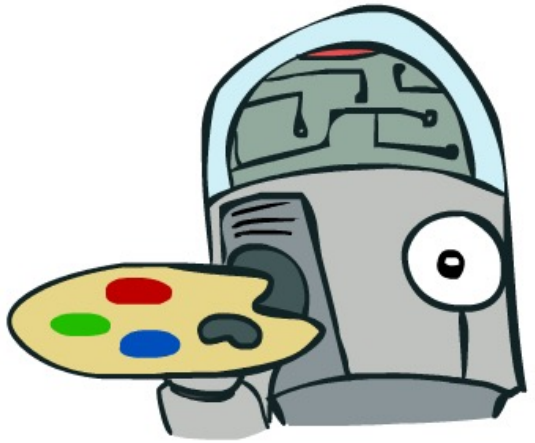
  9-way alldiff for each region

  (or can have a bunch of pairwise inequality constraints)

Varieties of CSPs and Constraints

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.:

    $$SA \neq green$$

  - Binary constraints involve pairs of variables, e.g.:

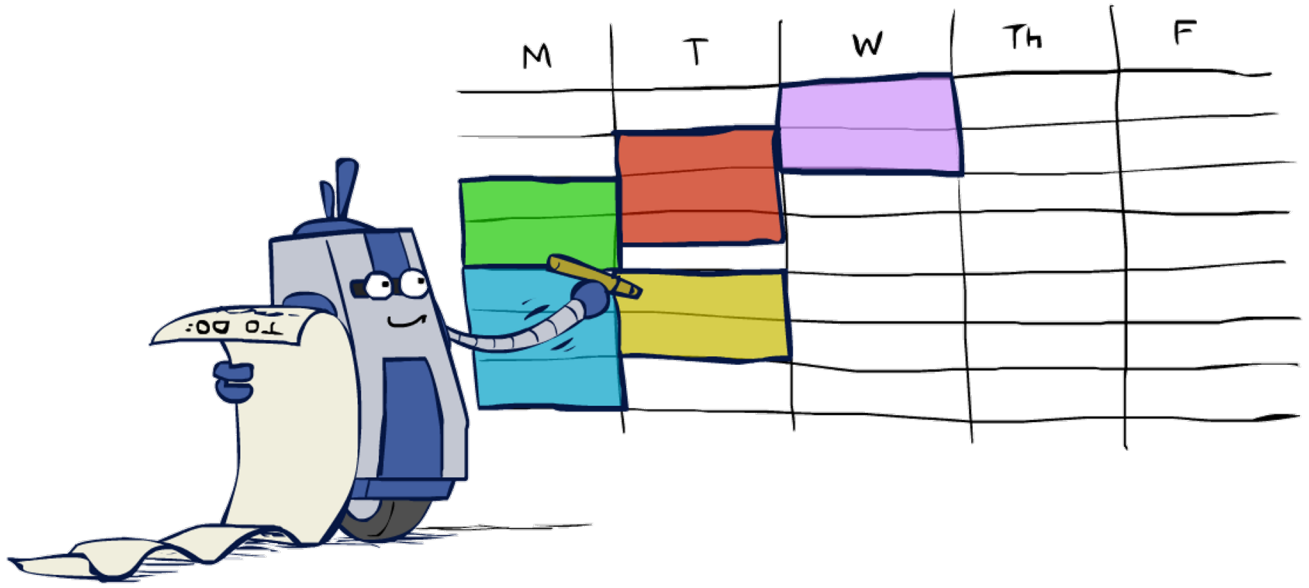
    $$SA \neq WA$$

  - Higher-order constraints involve 3 or more variables:
    e.g., cryptarithmetic column constraints

- Preferences (soft constraints):
  - E.g., red is better than green
  - Often representable by a cost for each variable assignment
  - Gives constrained optimization problems
  - (We'll ignore these until we get to Bayes' nets)

Real-World CSPs

- Scheduling problems: e.g., when can we all meet?
- Timetabling problems: e.g., which class is offered when and where?
- Assignment problems: e.g., who teaches what class
- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Circuit layout
- Fault diagnosis
- … lots more!

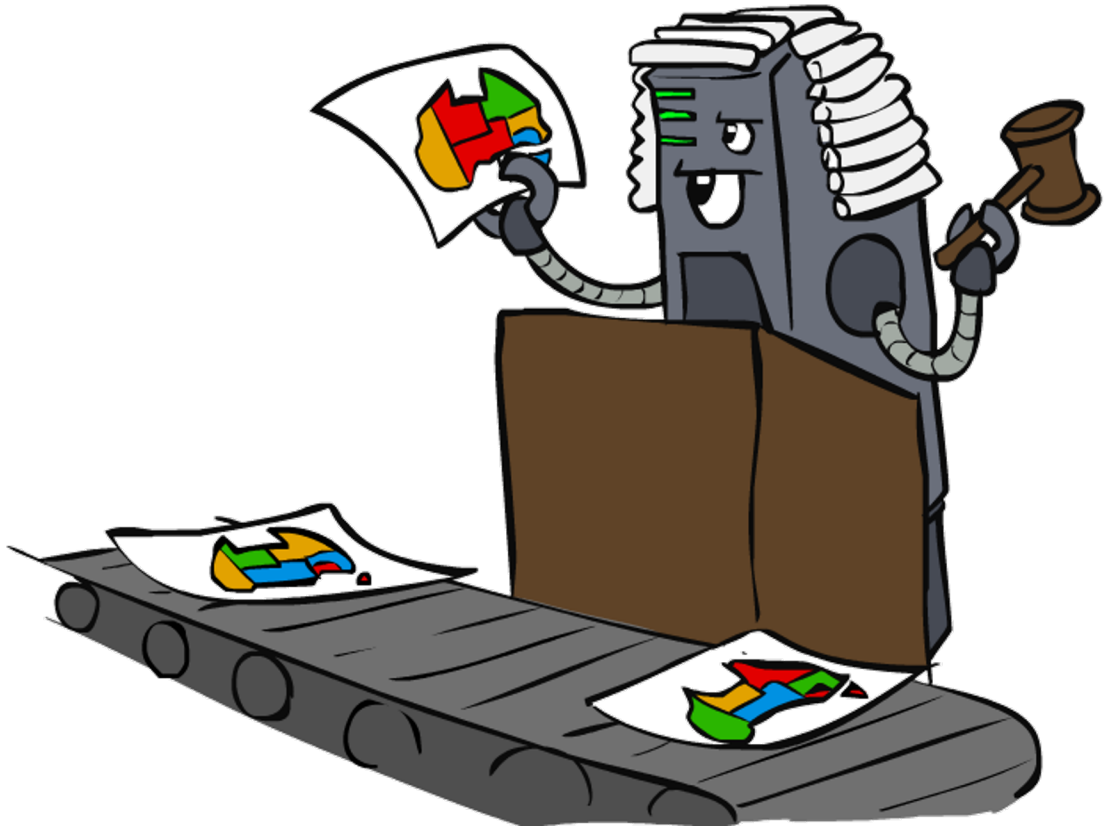


- Many real-world problems involve continuous variables…

Solving CSPs

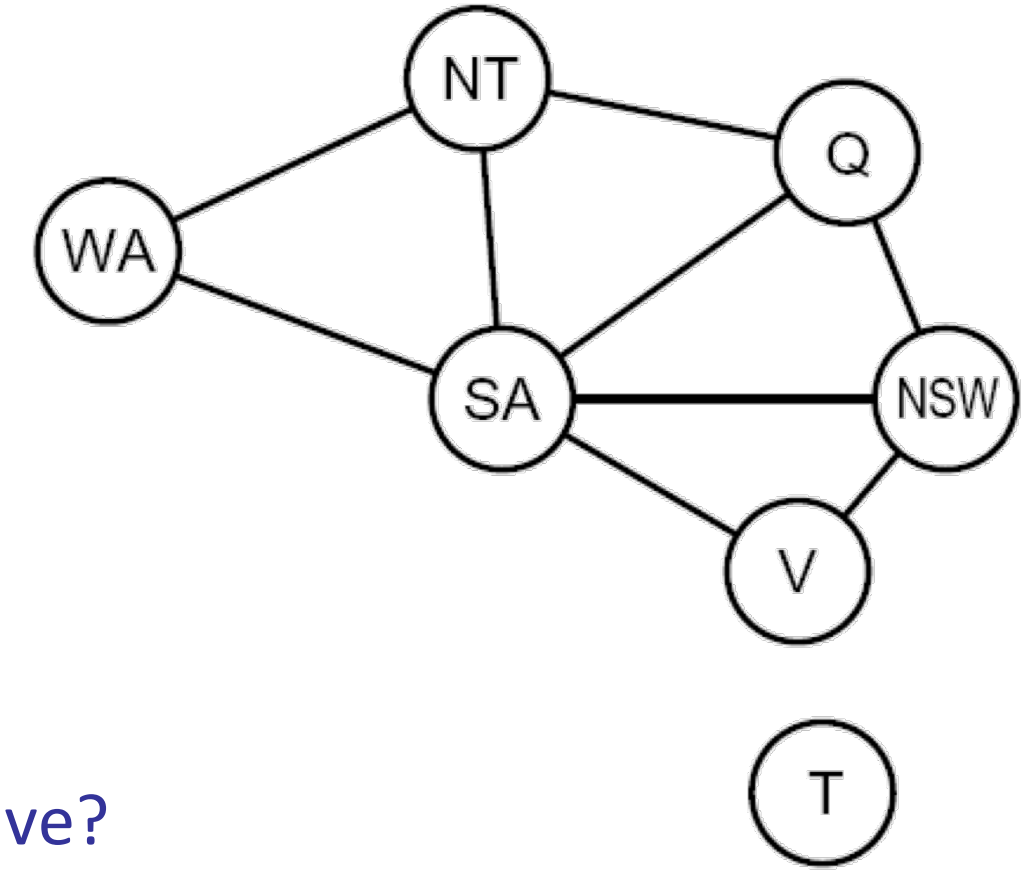
Standard Search Formulation

- 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



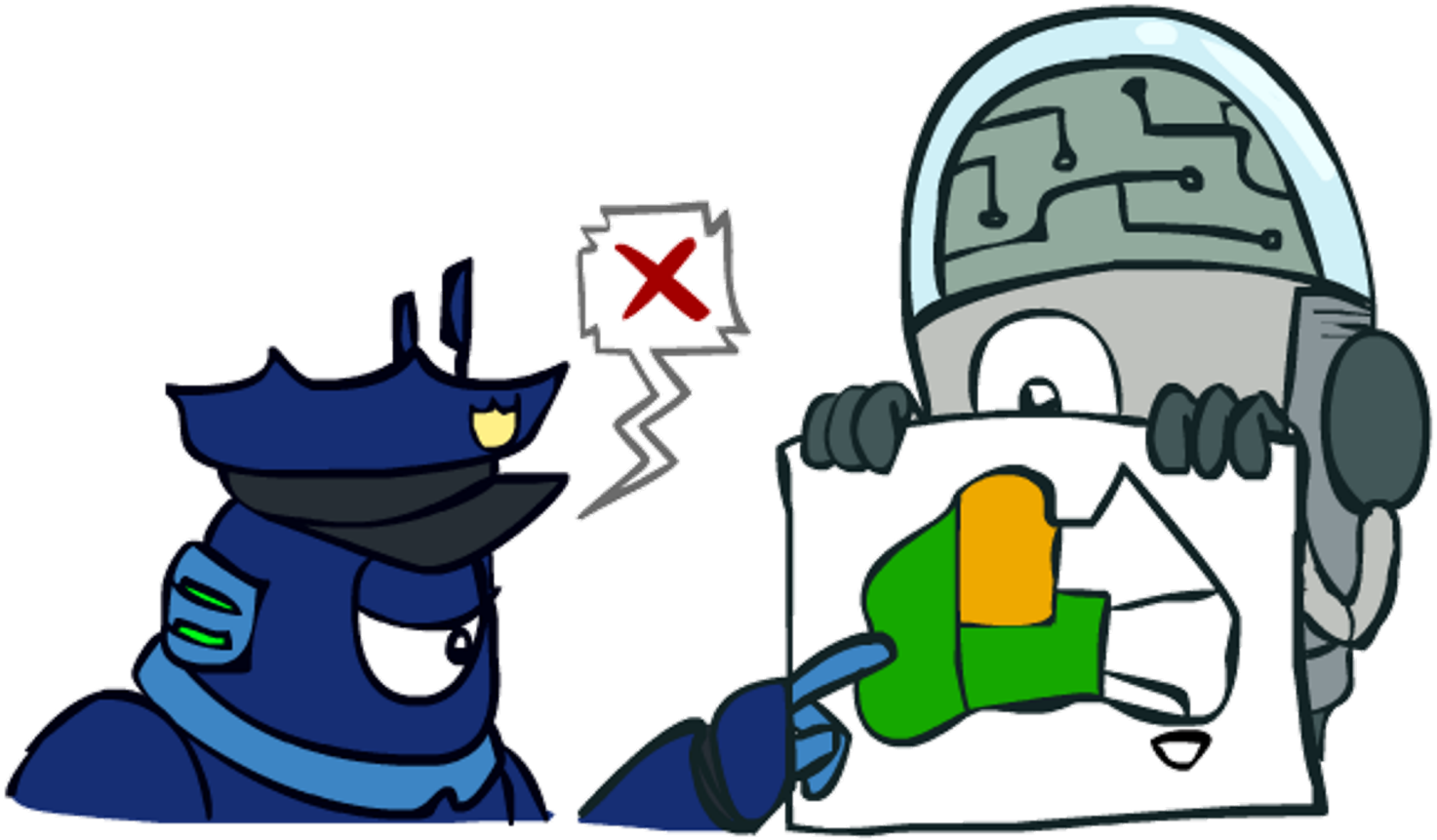
Search Methods

- What would BFS do?
- What would DFS do?
- What problems does naïve search have?



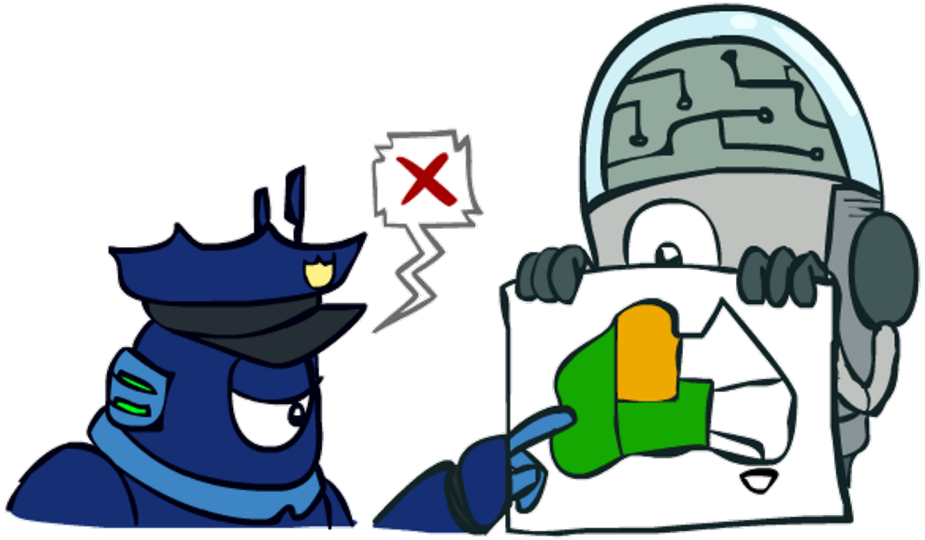
[Demo: coloring -- dfs]

Backtracking Search

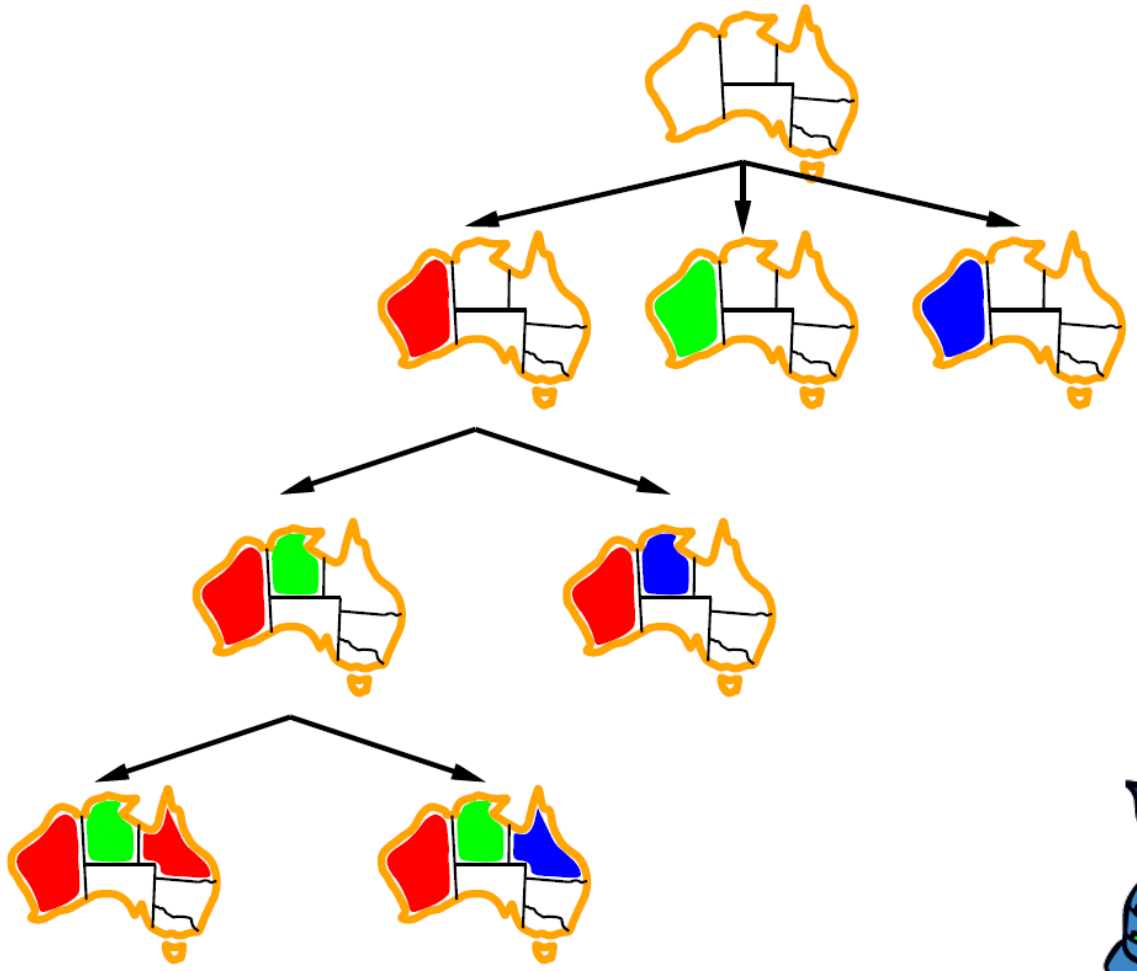


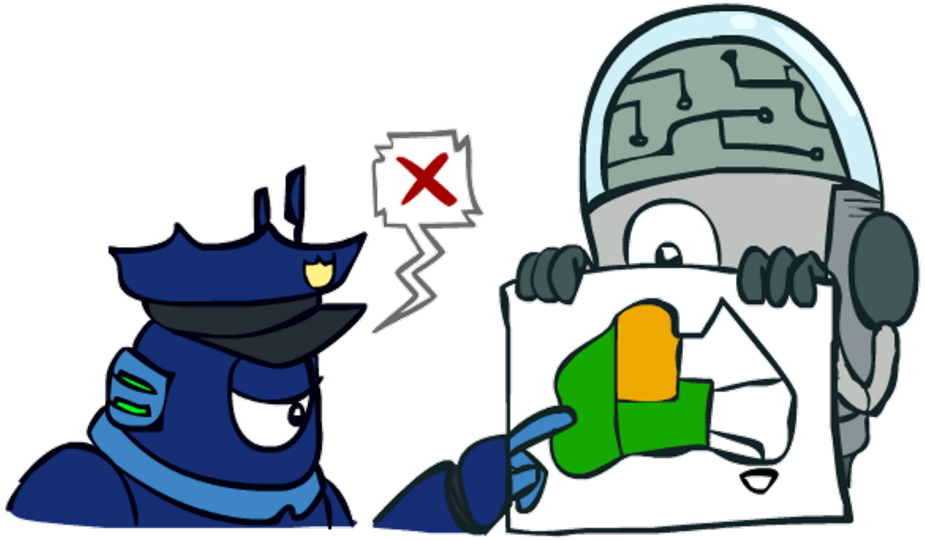
Backtracking Search

- 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 with 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 ≈ 25



Backtracking Example





Backtracking Search

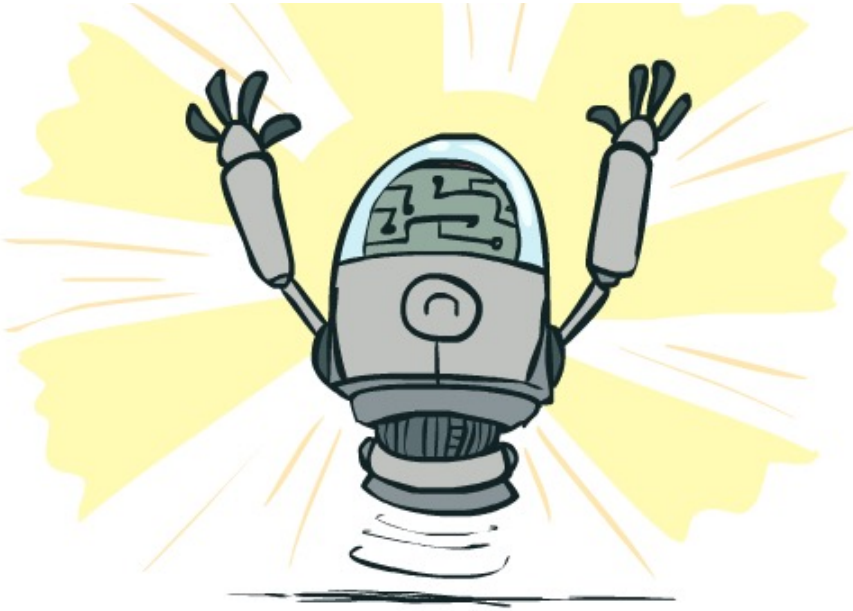
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

- Backtracking = DFS + variable-ordering + fail-on-violation
- What are the choice points?

[Demo: coloring -- backtracking]

Improving Backtracking

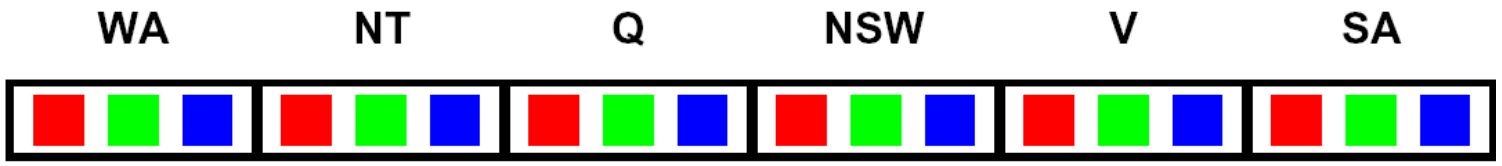
- General-purpose ideas give huge gains in speed
- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?
- Filtering: Can we detect inevitable failure early?
- Structure: Can we exploit the problem structure?



Filtering

Filtering: Forward Checking

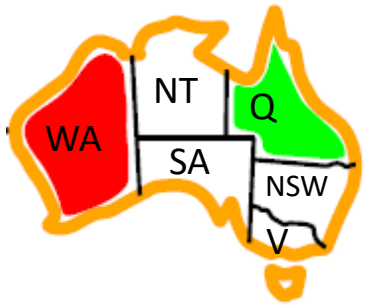
- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment

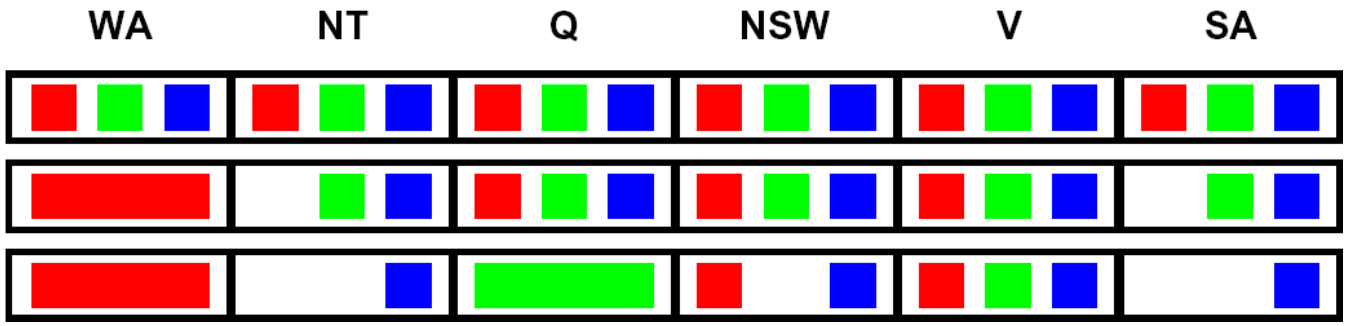


[Demo: coloring -- forward checking]

Filtering: Constraint Propagation

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

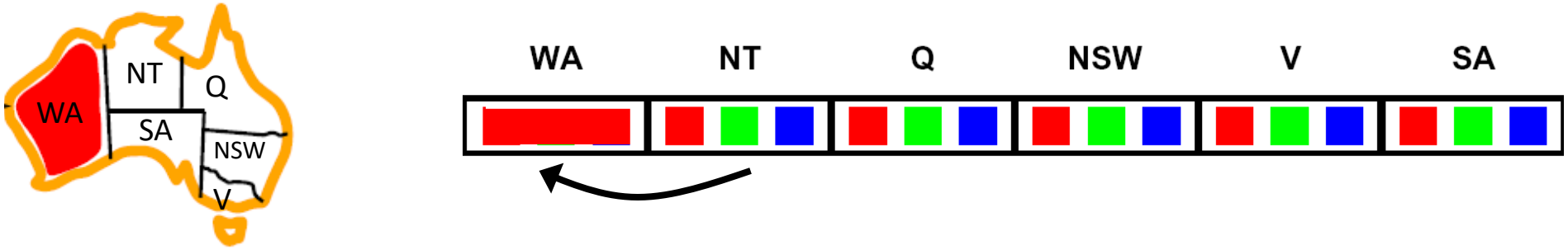




- NT and SA cannot both be blue!
- Why didn’t we detect this yet?
- *Constraint propagation:* reason from constraint to constraint

Consistency of A Single Arc

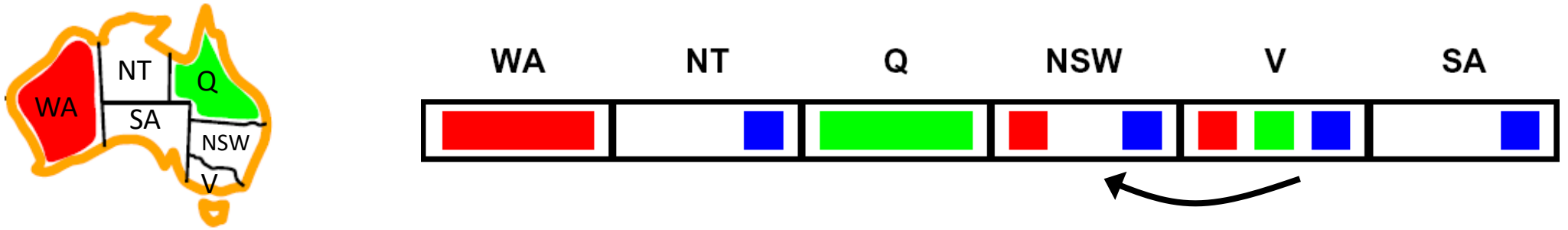
- An arc X → Y is consistent iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint



- Tail = NT, head = WA
  - If NT = blue: we could assign WA = red
  - If NT = green: we could assign WA = red
  - If NT = red: there is no remaining assignment to WA that we can use
  - Deleting NT = red from the tail makes this arc consistent
- Forward checking: Enforcing consistency of arcs pointing to each new assignment

Arc Consistency of an Entire CSP (1/6)

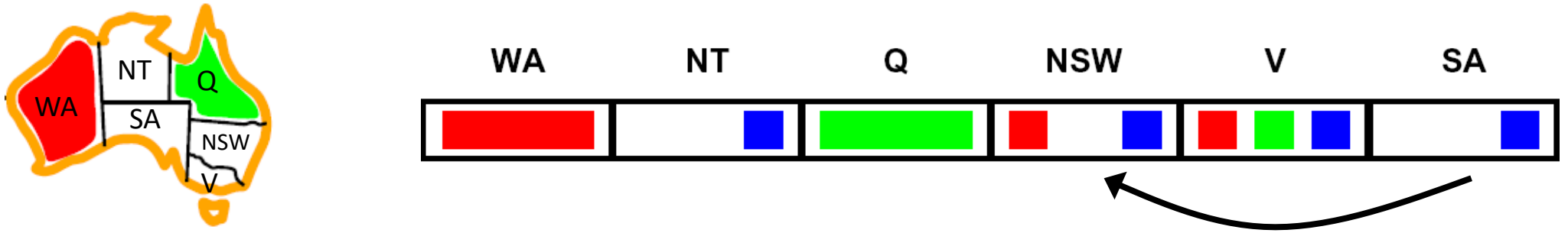
- A simple form of propagation makes sure all arcs are consistent:



- Arc V to NSW is consistent: for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint

Arc Consistency of an Entire CSP (2/6)

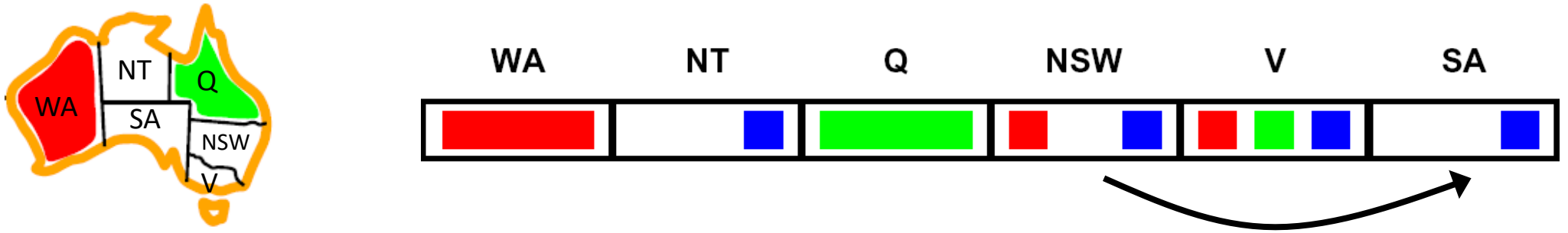
- A simple form of propagation makes sure all arcs are consistent:



- Arc SA to NSW is consistent: for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint

Arc Consistency of an Entire CSP (3/6)

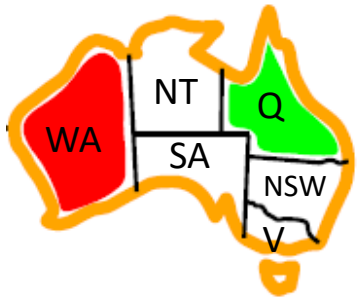
- A simple form of propagation makes sure all arcs are consistent:

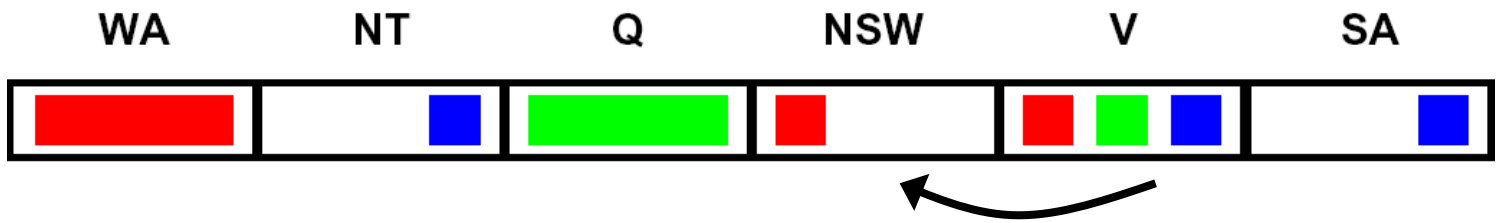


- Arc NSW to SA is not consistent: if we assign NSW = blue, there is no valid assignment left for SA
- To make this arc consistent, we delete NSW = blue from the **tail**

Arc Consistency of an Entire CSP (4/6)

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

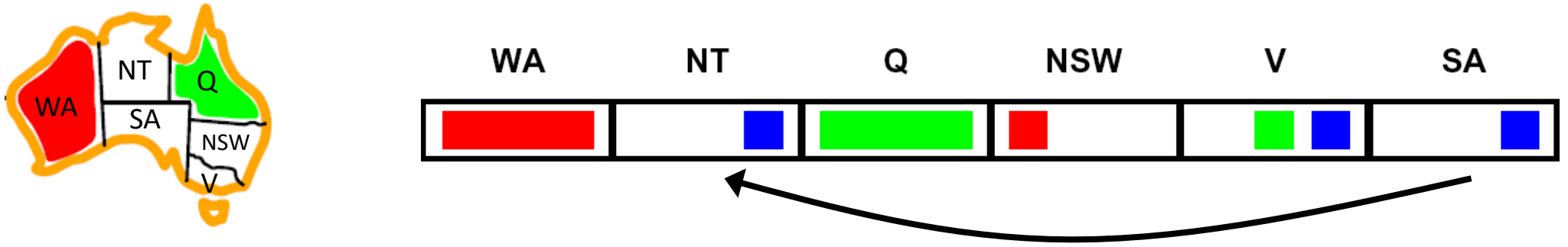




- Remember that arc V to NSW was consistent, when NSW had red and blue in its domain
- After removing blue from NSW, this arc might not be consistent anymore! We need to recheck this arc.
- Important: *If X loses a value, neighbors of X need to be rechecked!*

Arc Consistency of an Entire CSP (5/6)

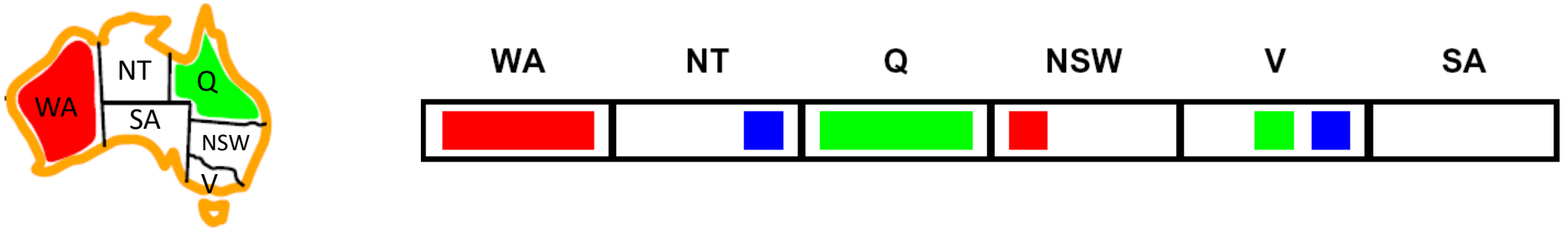
- A simple form of propagation makes sure all arcs are consistent:



- Arc SA to NT is inconsistent. We make it consistent by deleting from the tail (SA = blue).

Arc Consistency of an Entire CSP (6/6)

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



- SA has an empty domain, so we detect failure. There is no way to solve this CSP with WA = red and Q = green, so we backtrack.
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment

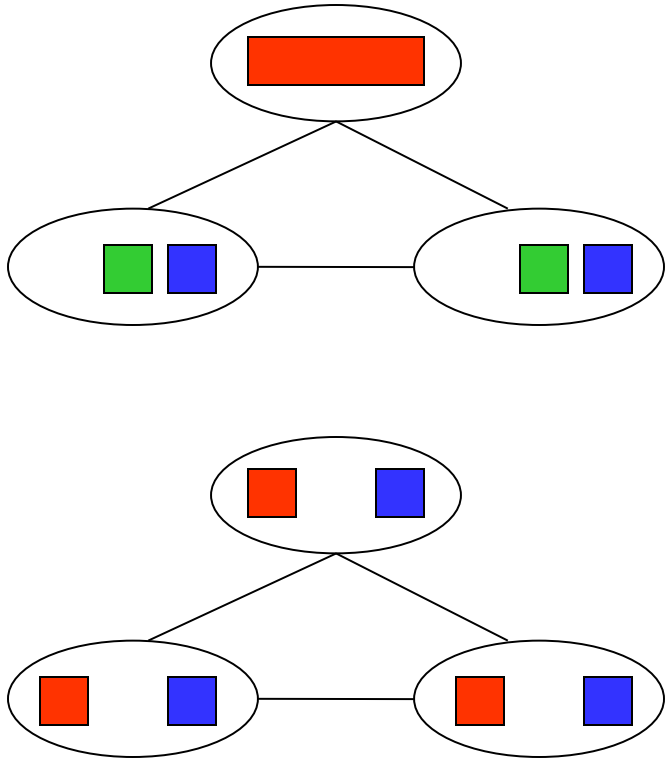
Enforcing Arc Consistency in a CSP

```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X_1, X_2, ..., 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) ← 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 ← 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 ↔ X_j
      then delete x from DOMAIN[X_i]; removed ← 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?

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?

- Arc consistency still runs inside a backtracking search!

[Demo: coloring -- forward checking]
[Demo: coloring -- arc consistency]