

Announcements

- HW1 is due Wednesday, February 5, 11:59 PM PT
- Project 1 is due Friday, February 7, 11:59 PM PT
- Sections start this week - go to any

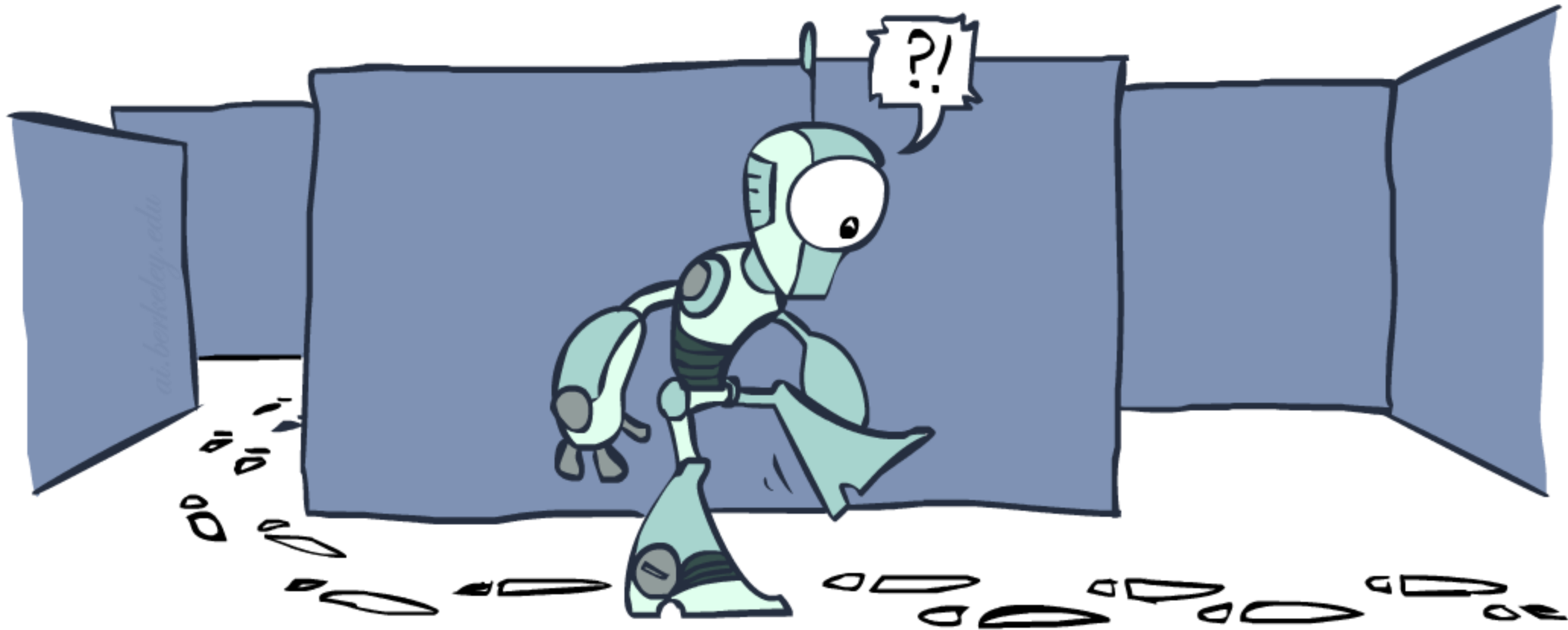
CS 188: Artificial Intelligence

Constraint Satisfaction Problems



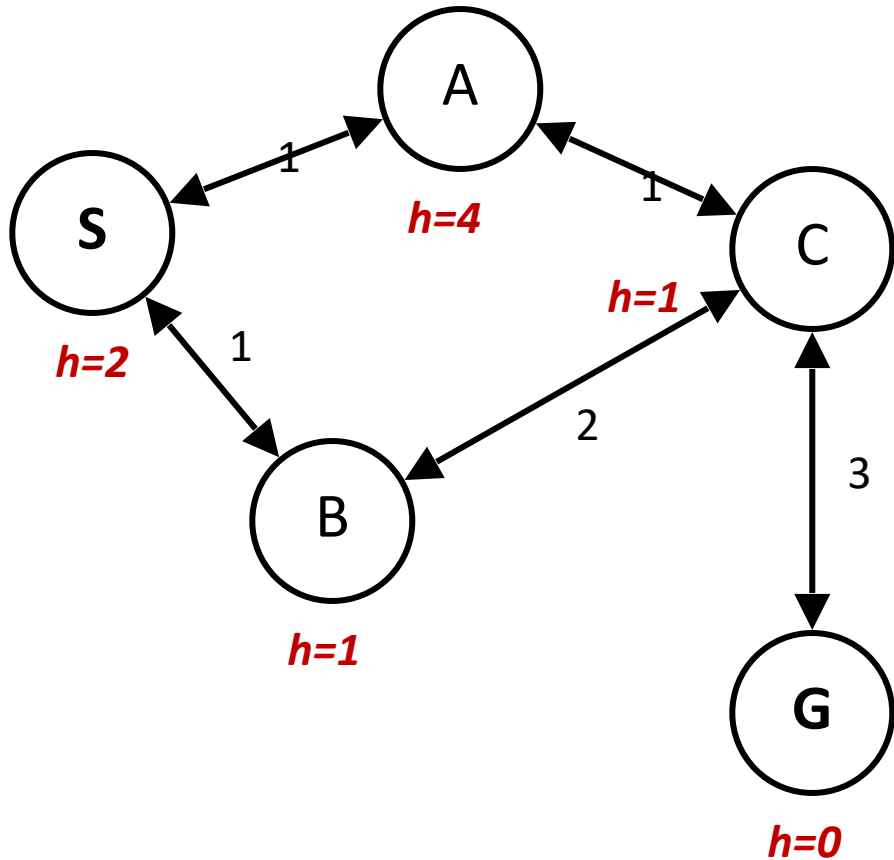
John Canny, Oliver Grillmeyer
University of California, Berkeley

Graph Search and Consistency

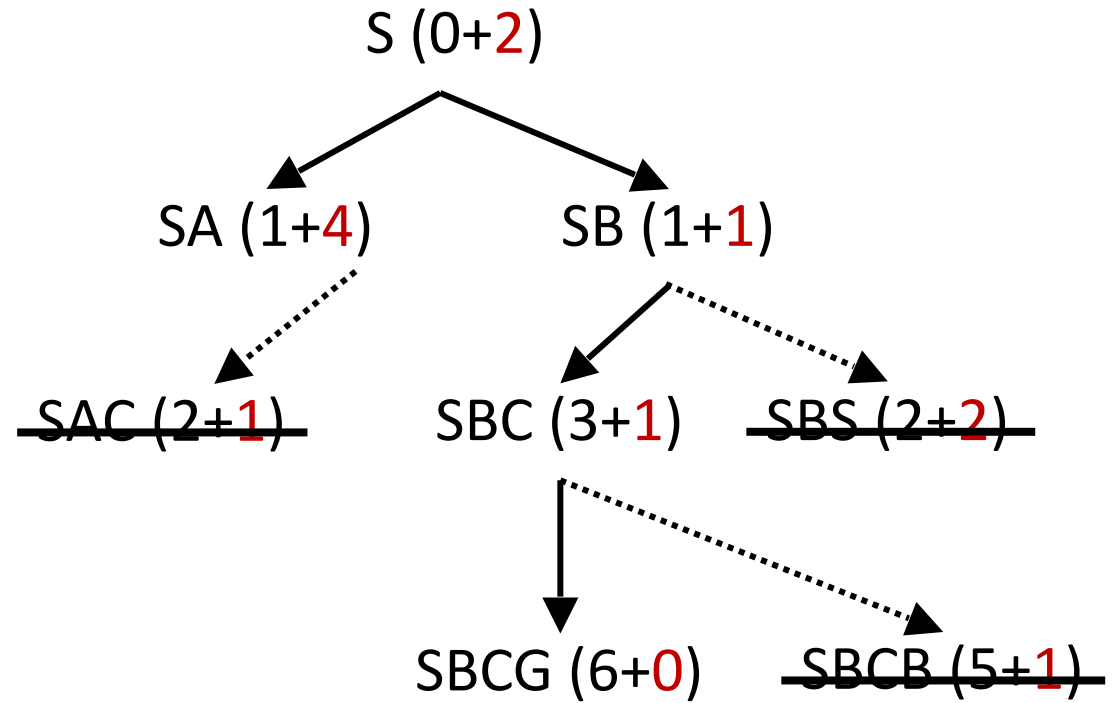


A* Graph Search Gone Wrong?

State space graph



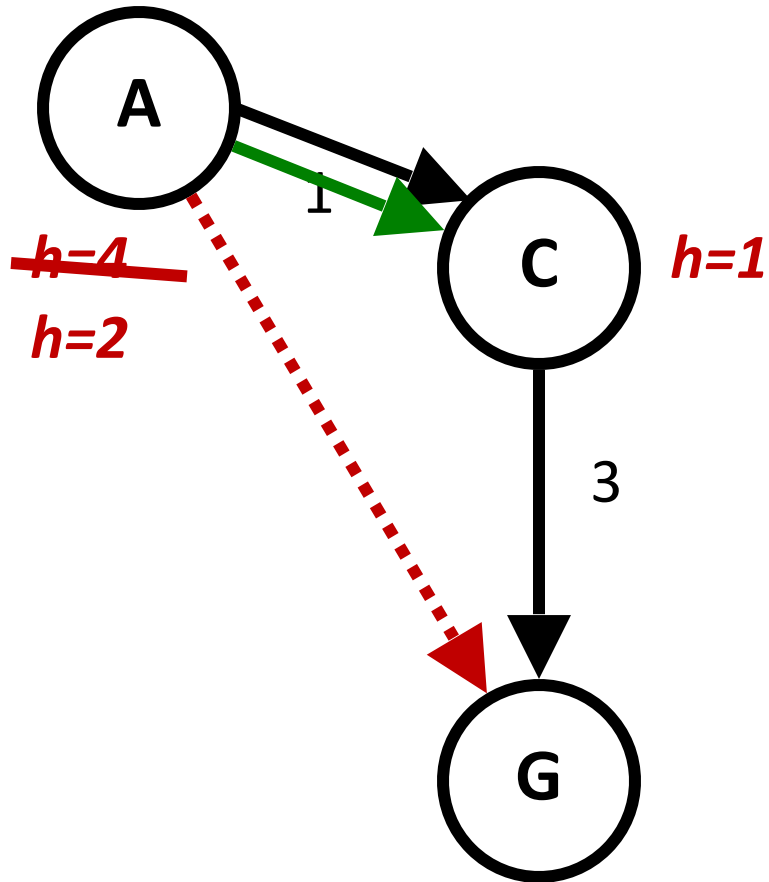
Search tree



Closed set

{ S B C G }

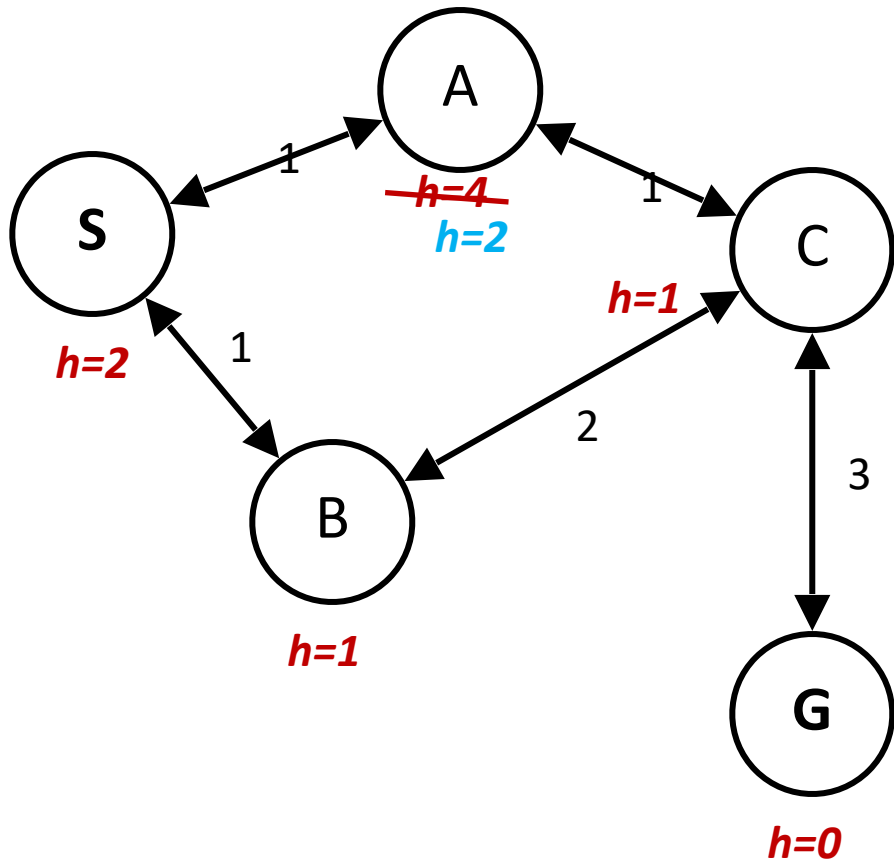
Consistency of Heuristics



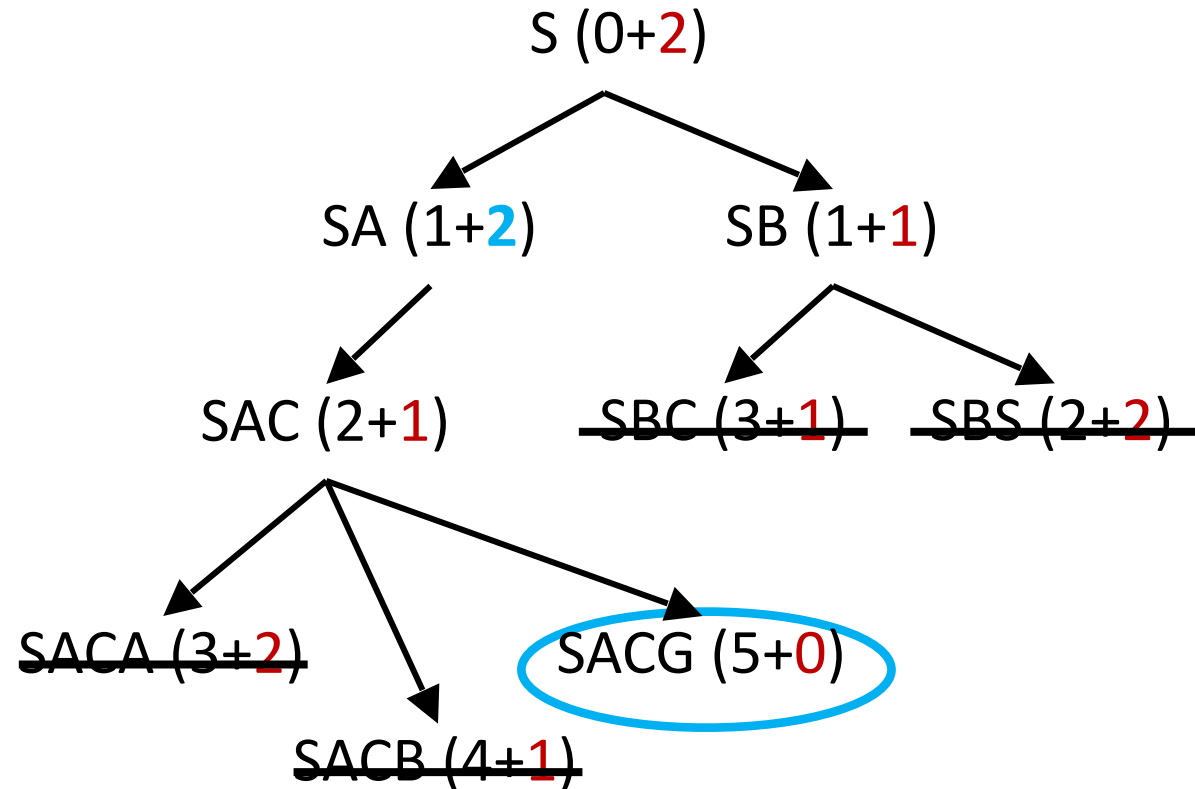
- Main idea: estimated heuristic costs \leq actual costs
 - Admissibility: heuristic cost \leq actual cost to goal
$$h(A) \leq \text{actual cost from A to G}$$
 - Consistency: heuristic "arc" cost \leq actual cost for each arc
$$h(A) - h(C) \leq \text{cost}(A \text{ to } C)$$
- Consequences of consistency:
 - The f value along a path never decreases
$$h(A) \leq \text{cost}(A \text{ to } C) + h(C)$$
 - A* graph search is optimal

A* Graph Search with Consistent Heuristic

State space graph



Search tree

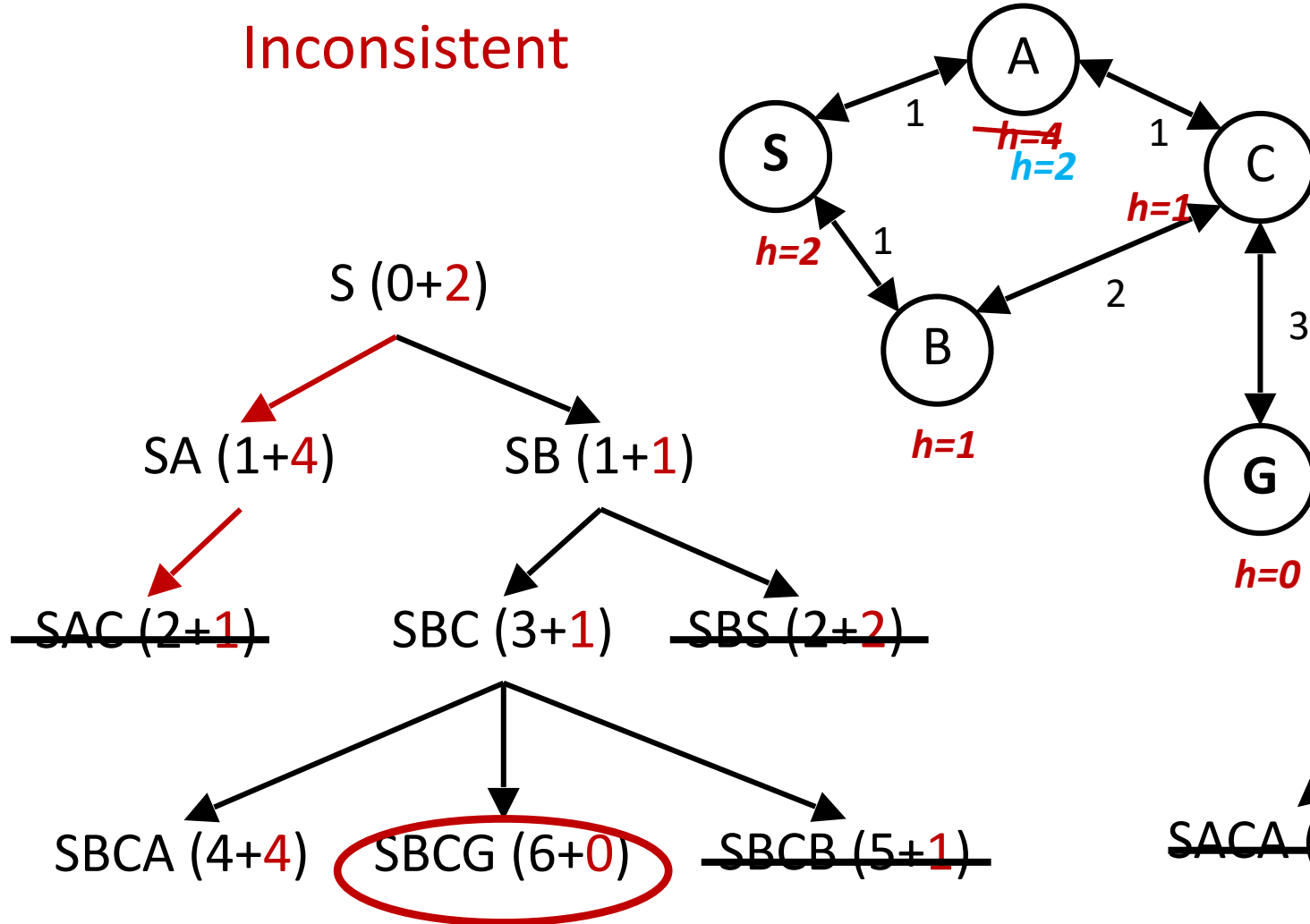


Closed set

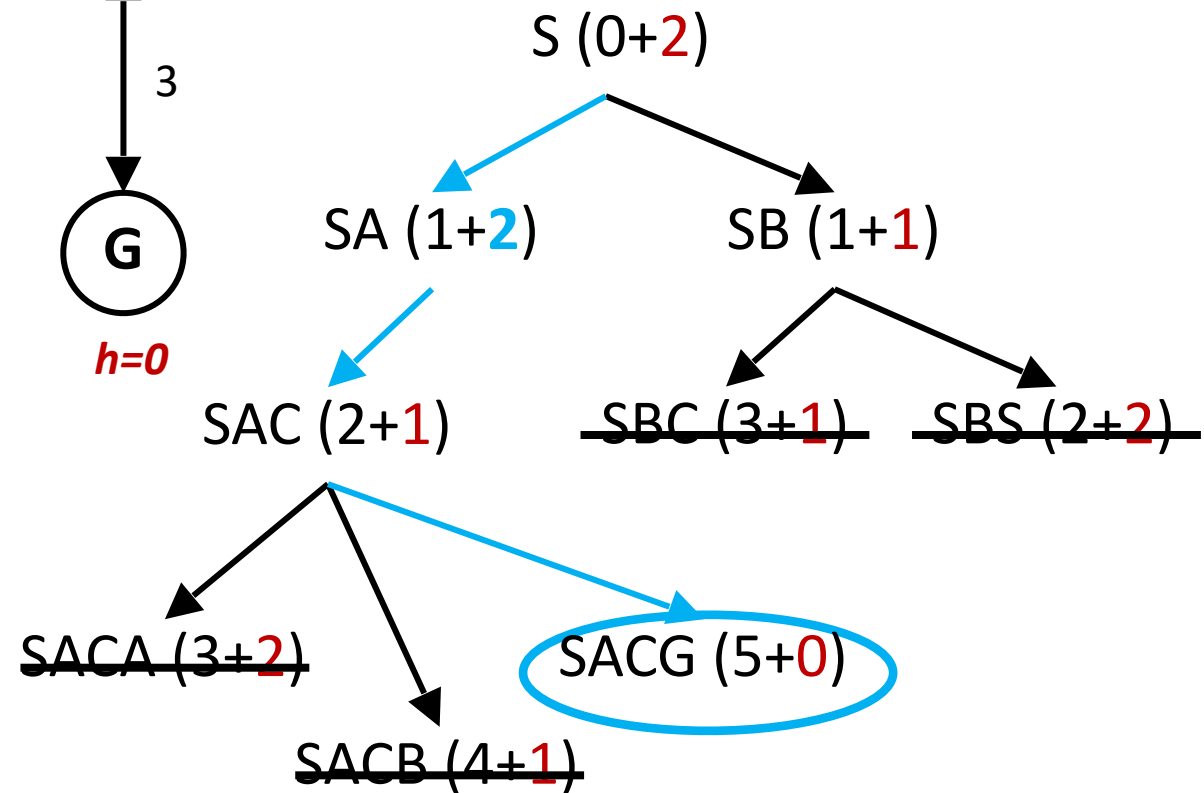
{ S B A C }

Consistency => non-decreasing f-score

Inconsistent

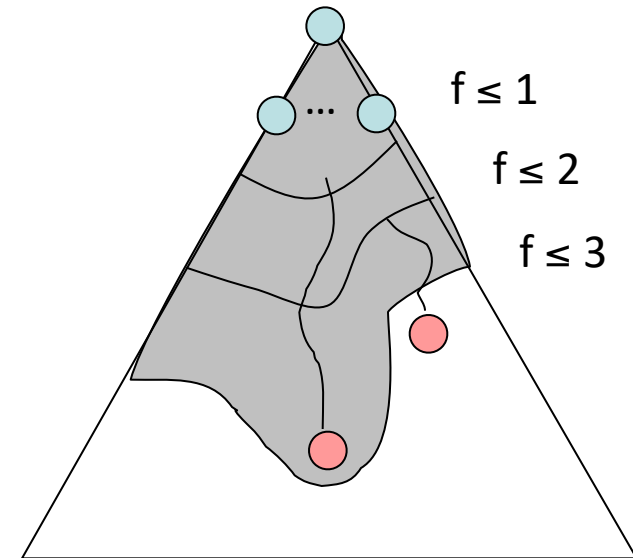


Consistent

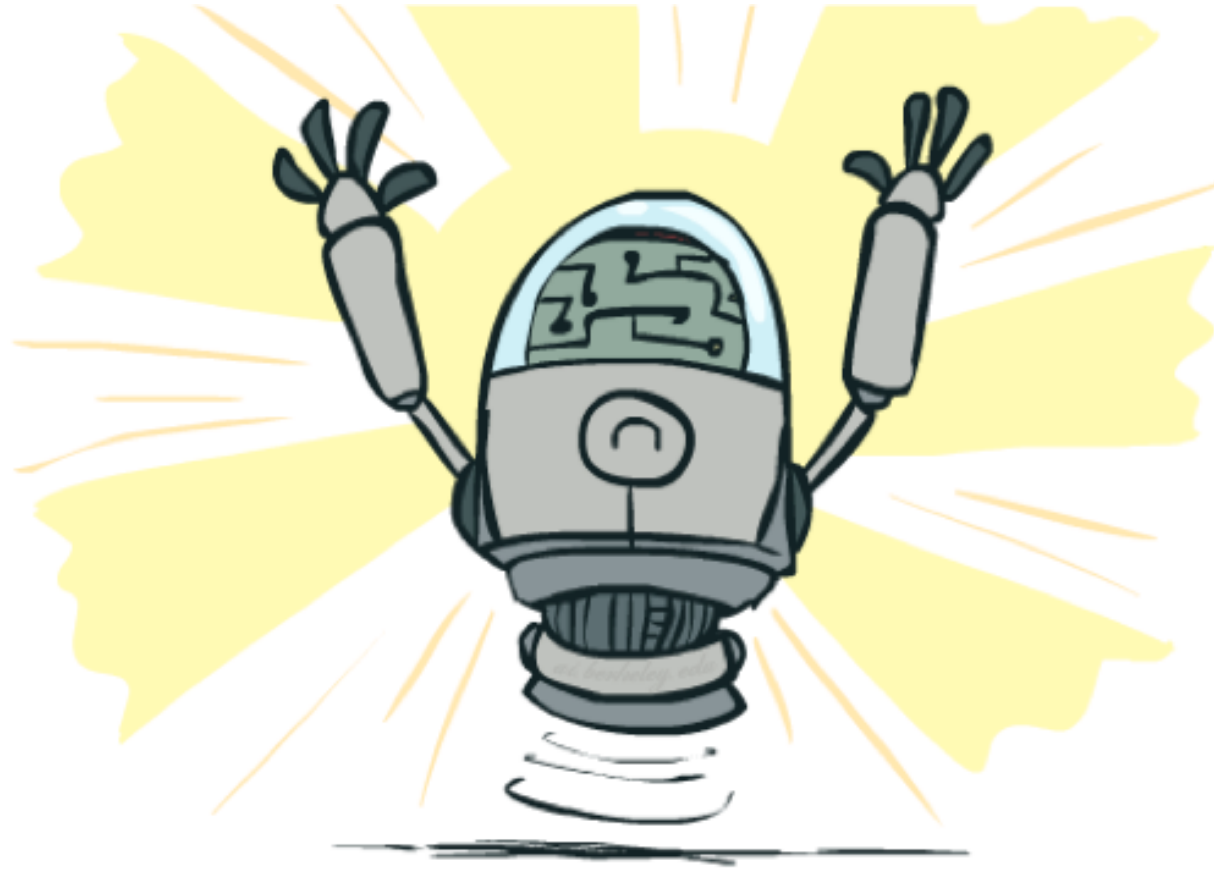


Optimality of A* Graph Search

- 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, nodes that reach s optimally are expanded before nodes that reach s suboptimally
 - Result: A* graph search is optimal

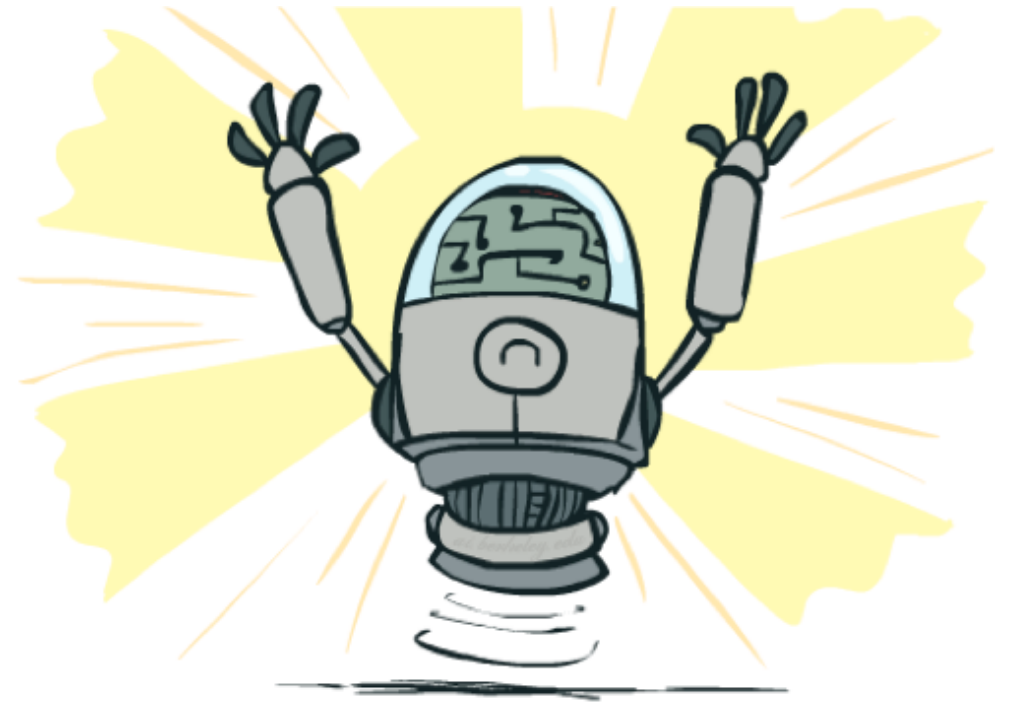


Optimality of A* Graph Search



Optimality

- Tree search:
 - A* is optimal if heuristic is admissible
 - UCS is a special case ($h = 0$)
- Graph search:
 - A* optimal if heuristic is consistent
 - UCS optimal ($h = 0$ is consistent)
- Consistency implies admissibility
- In general, most natural admissible heuristics tend to be consistent, especially if from relaxed problems

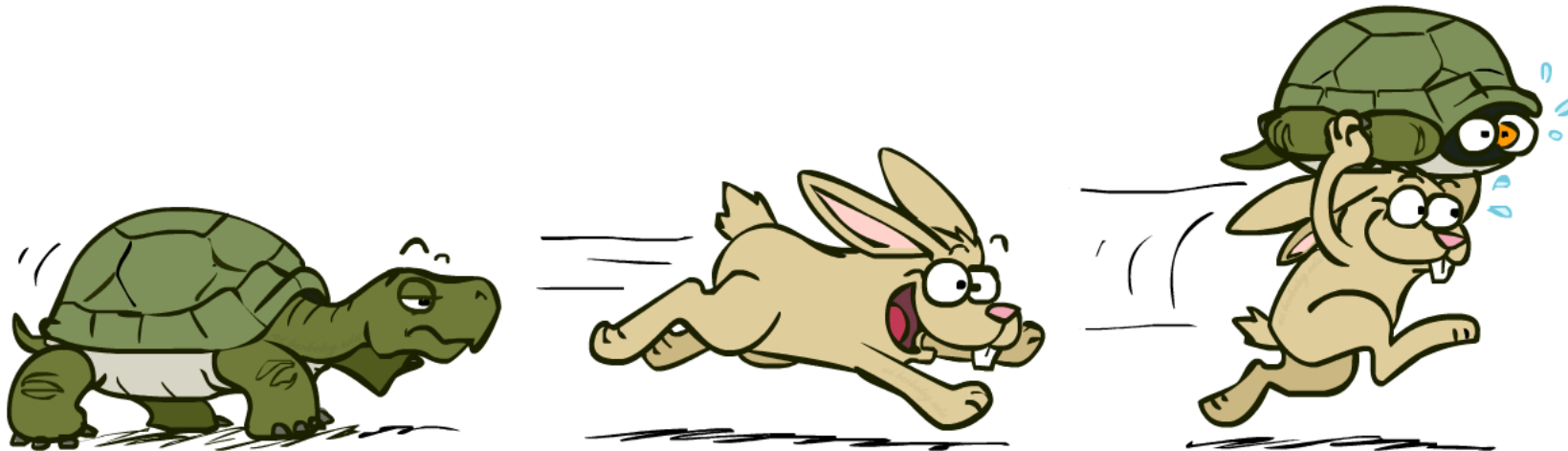


A*: Summary



A*: Summary

- A* uses both backward costs and (estimates of) forward costs
- A* is optimal with admissible / consistent heuristics
- Heuristic design is key: often use relaxed problems



Appendix: Search Pseudo-Code

Tree Search Pseudo-Code

```
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
```

Graph Search Pseudo-Code

```
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
  end
```

CS 188: Artificial Intelligence

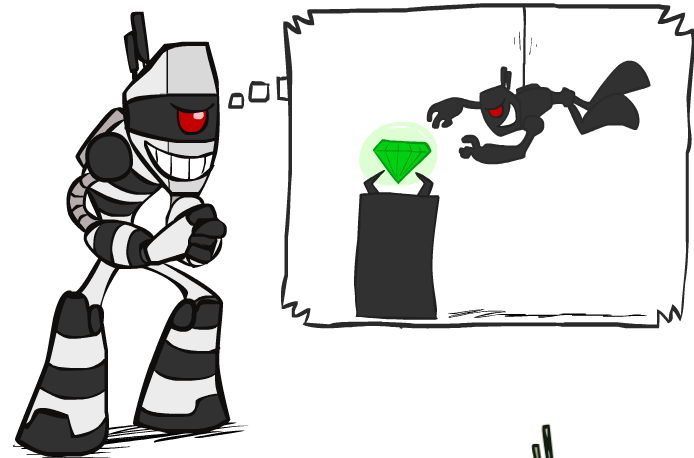
Constraint Satisfaction Problems



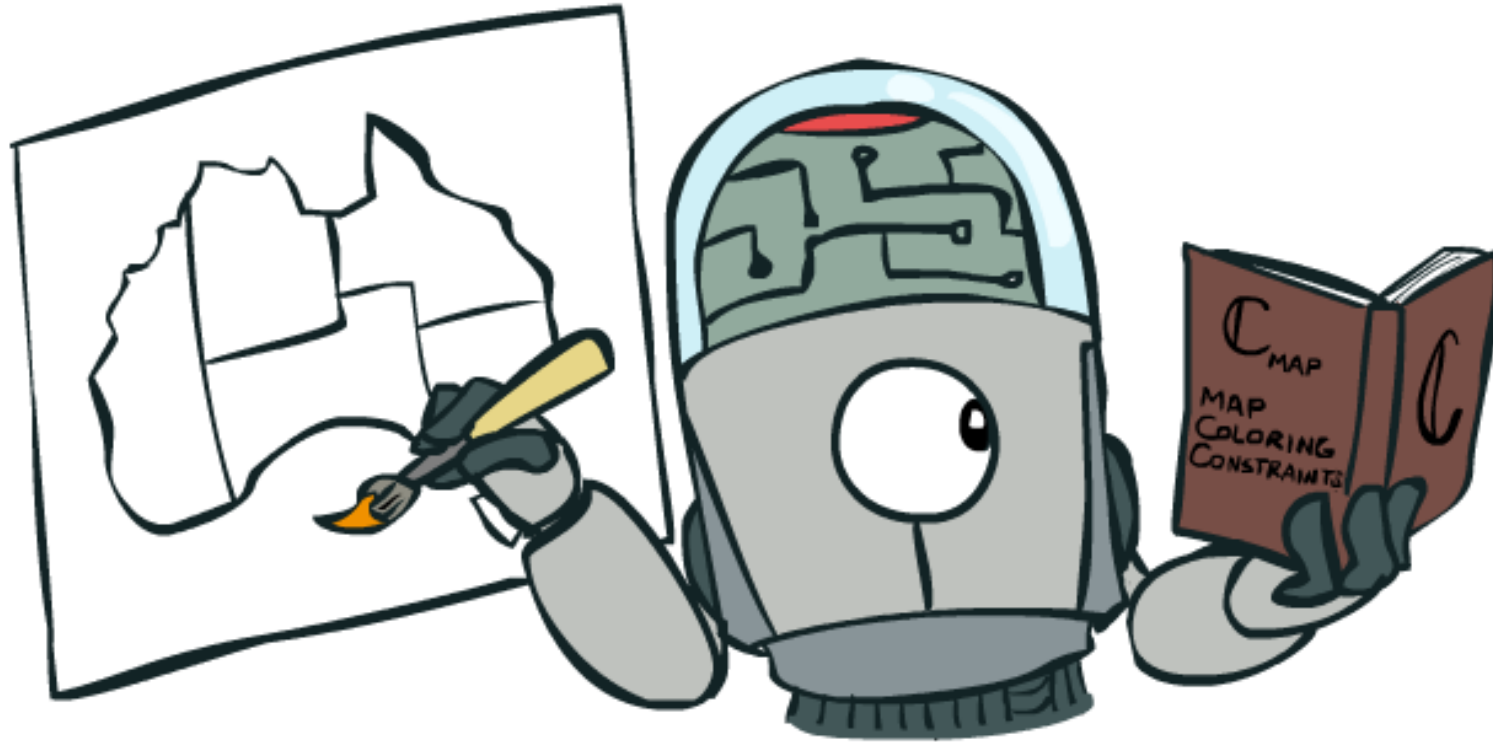
John Canny, Oliver Grillmeyer
University of California, Berkeley

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

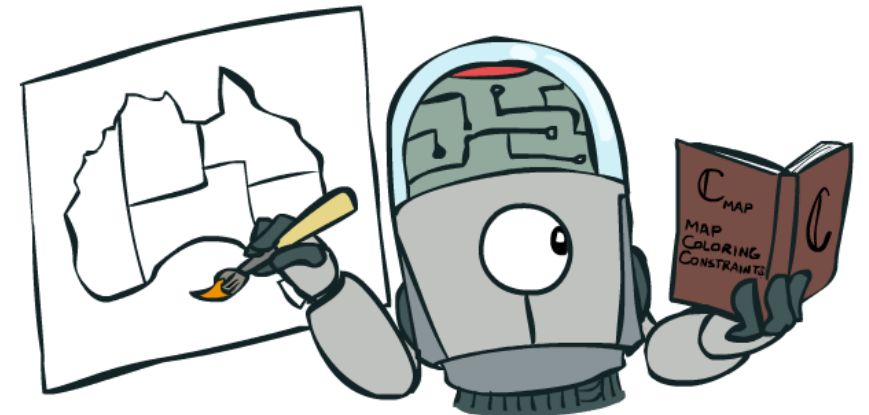
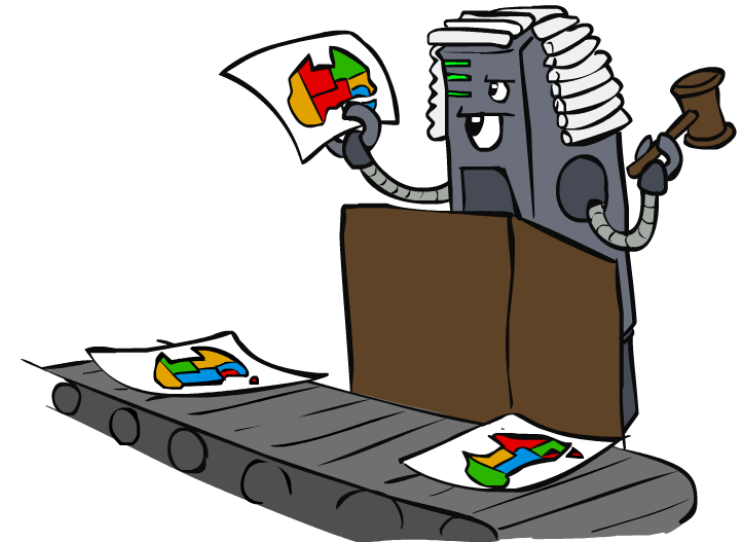


Constraint Satisfaction Problems

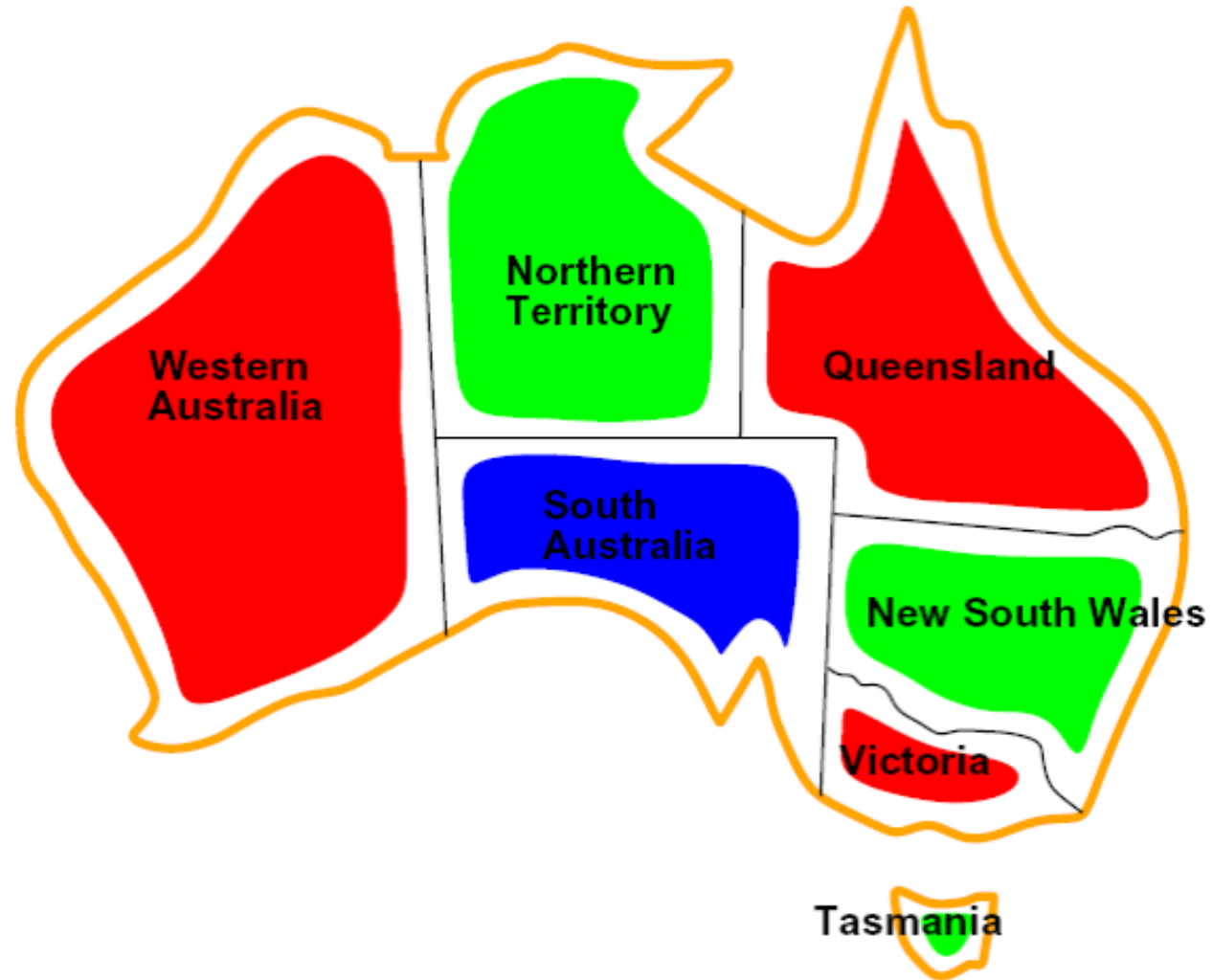


Constraint Satisfaction Problems

- 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

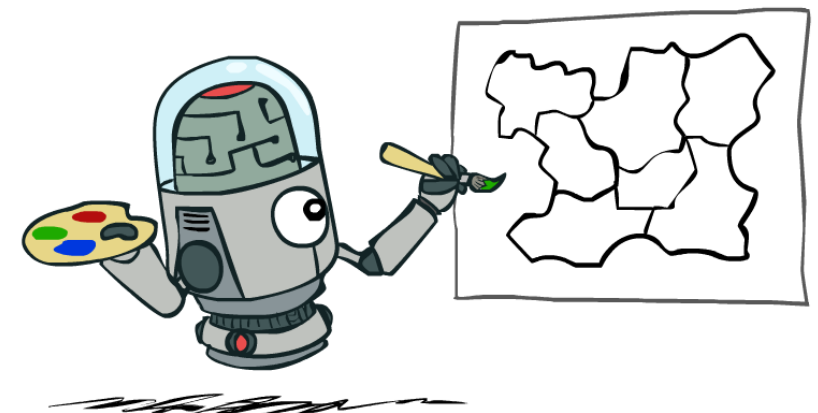
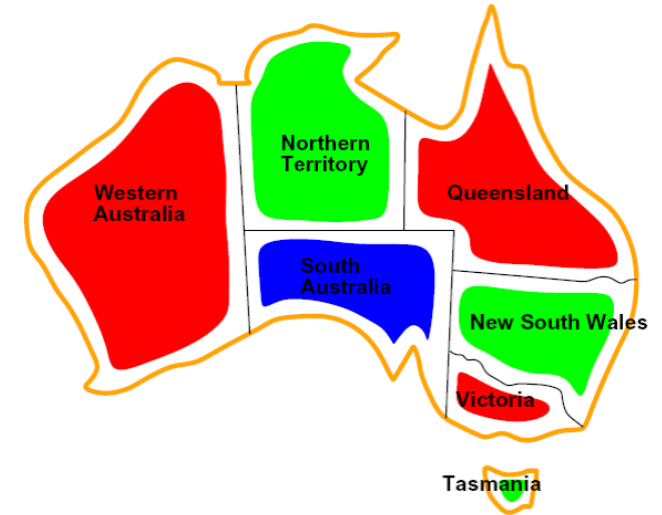
- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: $D = \{\text{red, green, blue}\}$
- Constraints: adjacent regions must have different colors

Implicit: $WA \neq NT$

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

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

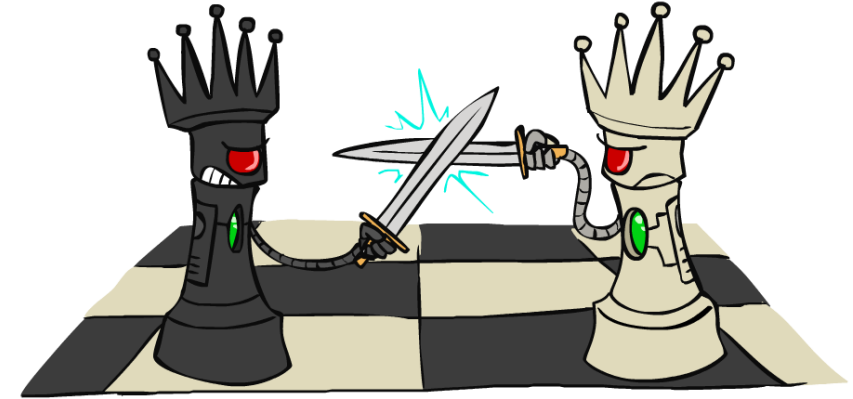
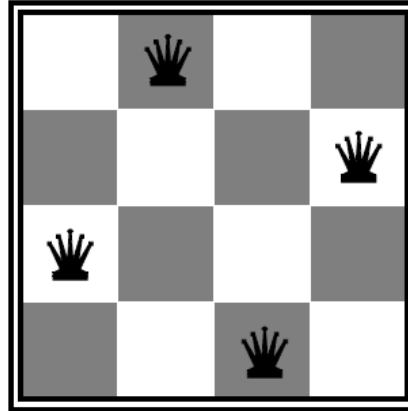
$\{WA=\text{red}, NT=\text{green}, Q=\text{red}, NSW=\text{green}, V=\text{red}, SA=\text{blue}, T=\text{green}\}$



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:

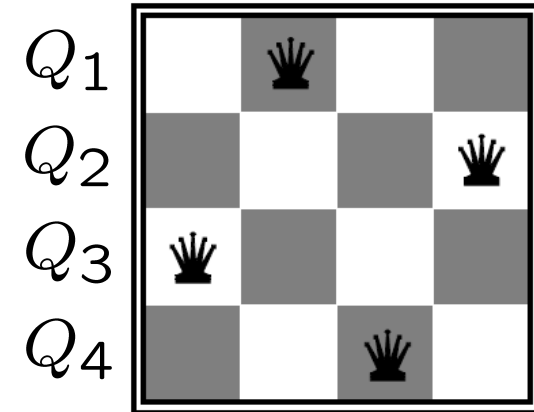
- Variables: Q_k
- Domains: $\{1, 2, 3, \dots, N\}$

■ Constraints:

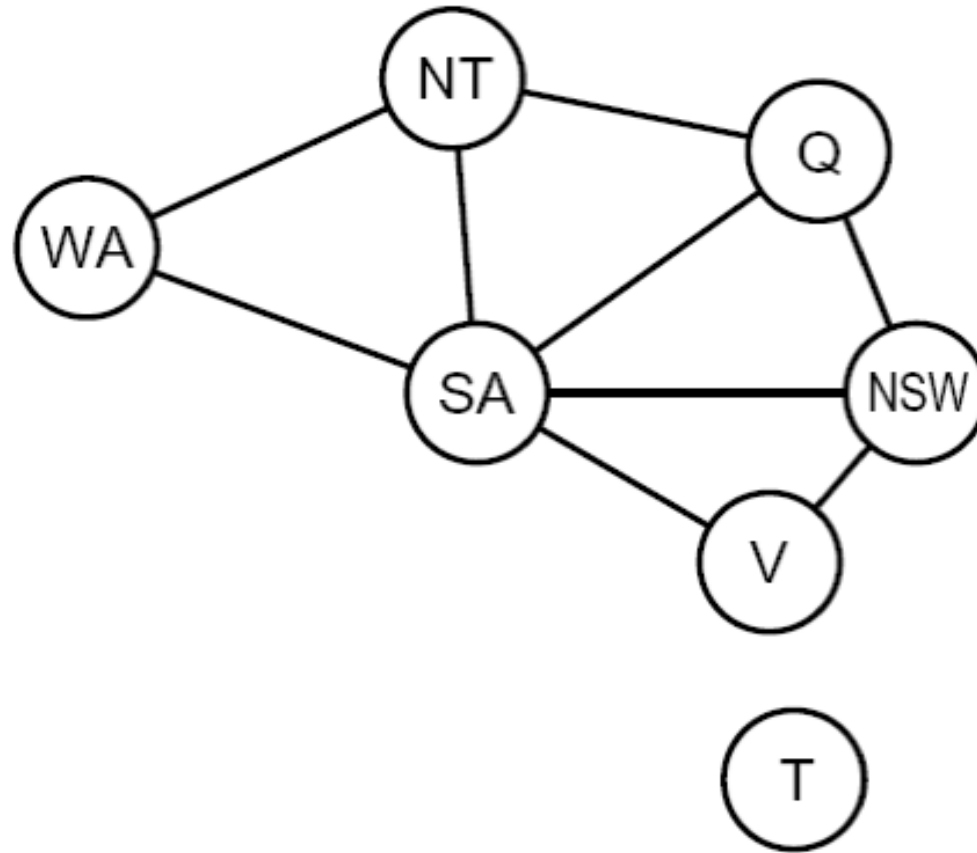
Implicit: $\forall i, j$ non-threatening(Q_i, Q_j)

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

...

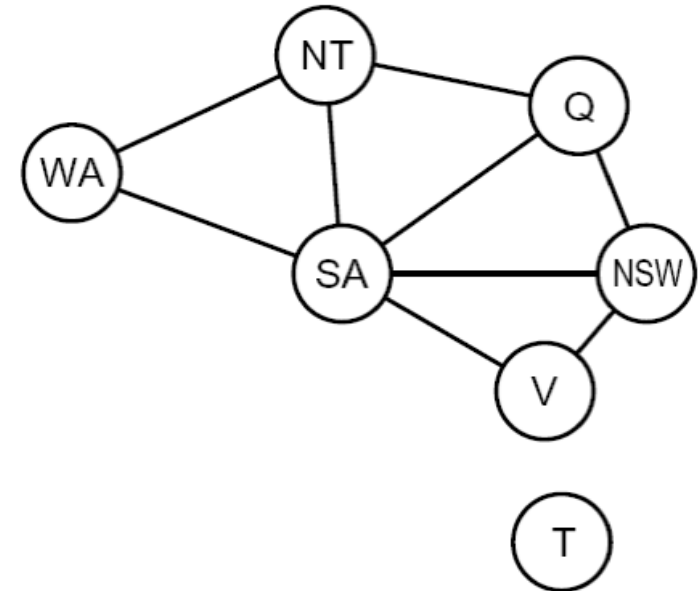


Constraint Graphs



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 speed up search. E.g., Tasmania is an independent subproblem!



Example: Cryptarithmic

- Variables:

$F T U W R O X_1 X_2 X_3$

- Domains:

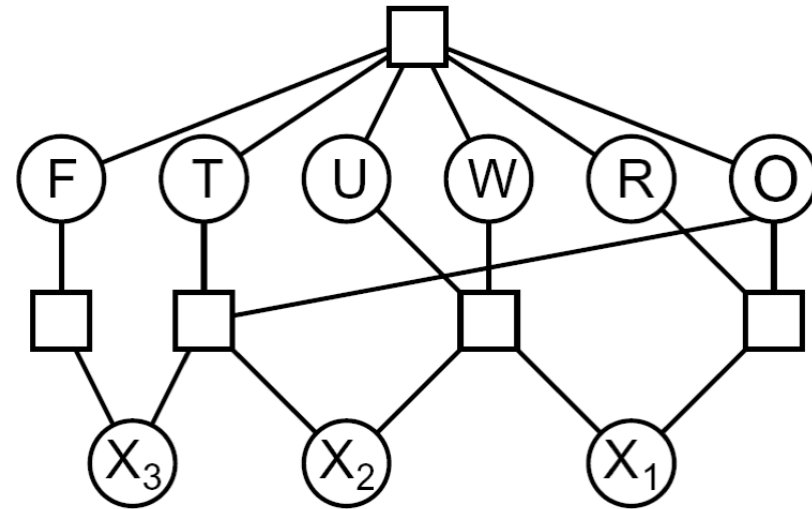
$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

- Constraints:

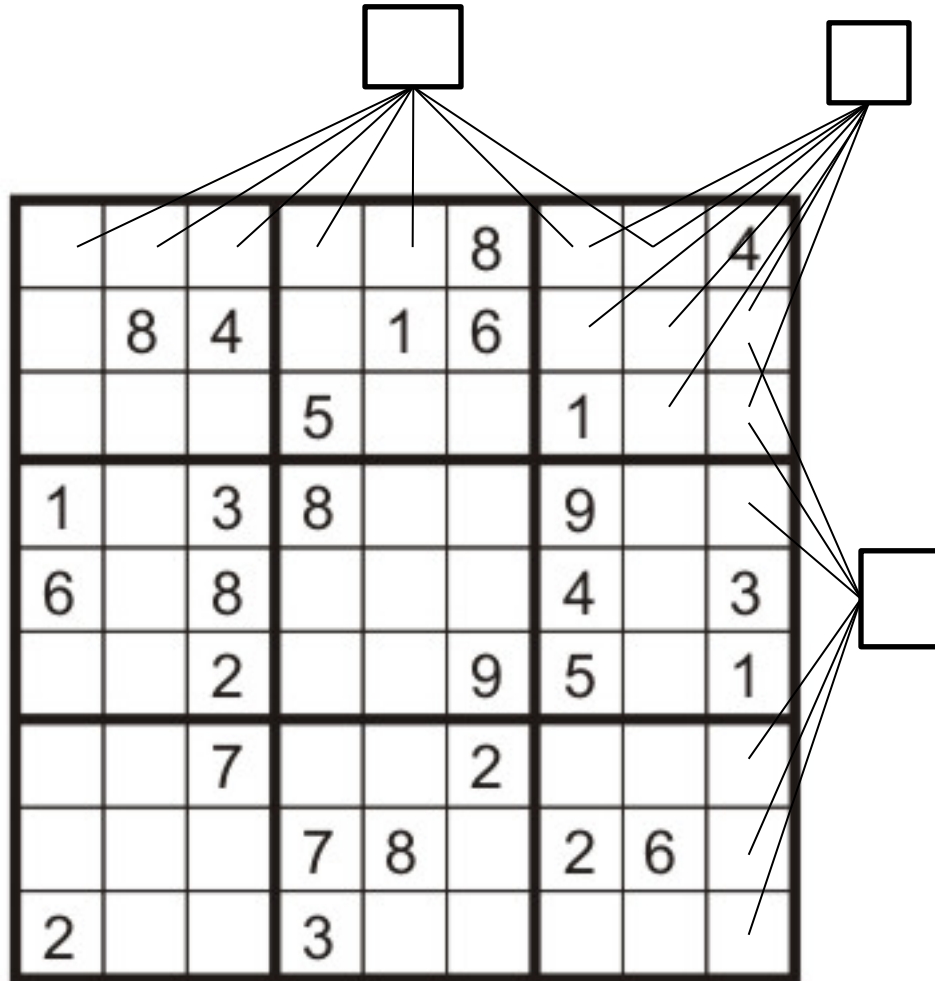
$\text{alldiff}(F, T, U, W, R, O)$

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

...

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


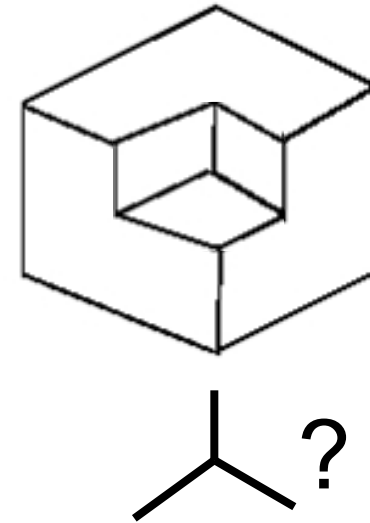
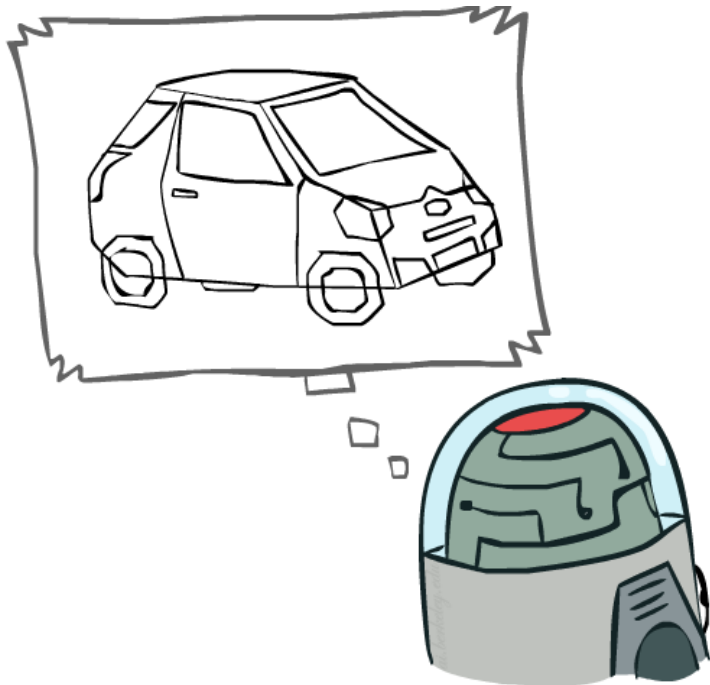
Example: Sudoku



- 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)

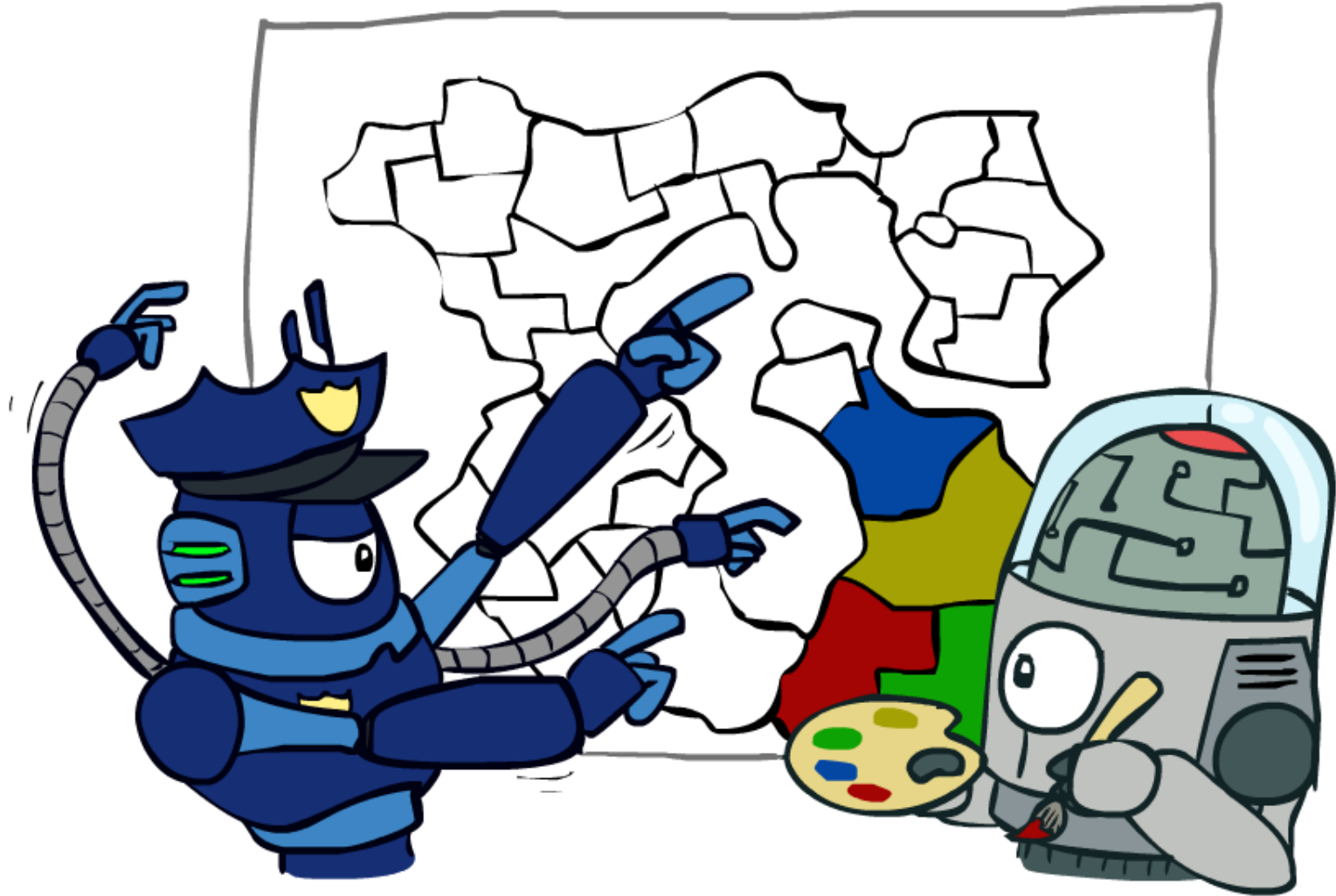
Example: The Waltz Algorithm

- The Waltz algorithm is for interpreting line drawings of solid polyhedra as 3D objects
- An early example of an AI computation posed as a CSP



- Approach:
 - Each intersection is a variable
 - Adjacent intersections impose constraints on each other
 - Solutions are physically realizable 3D interpretations

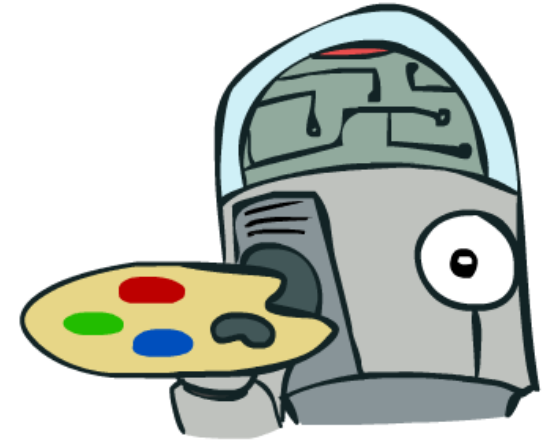
Varieties of CSPs and Constraints



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.:

$$SA \neq \text{green}$$

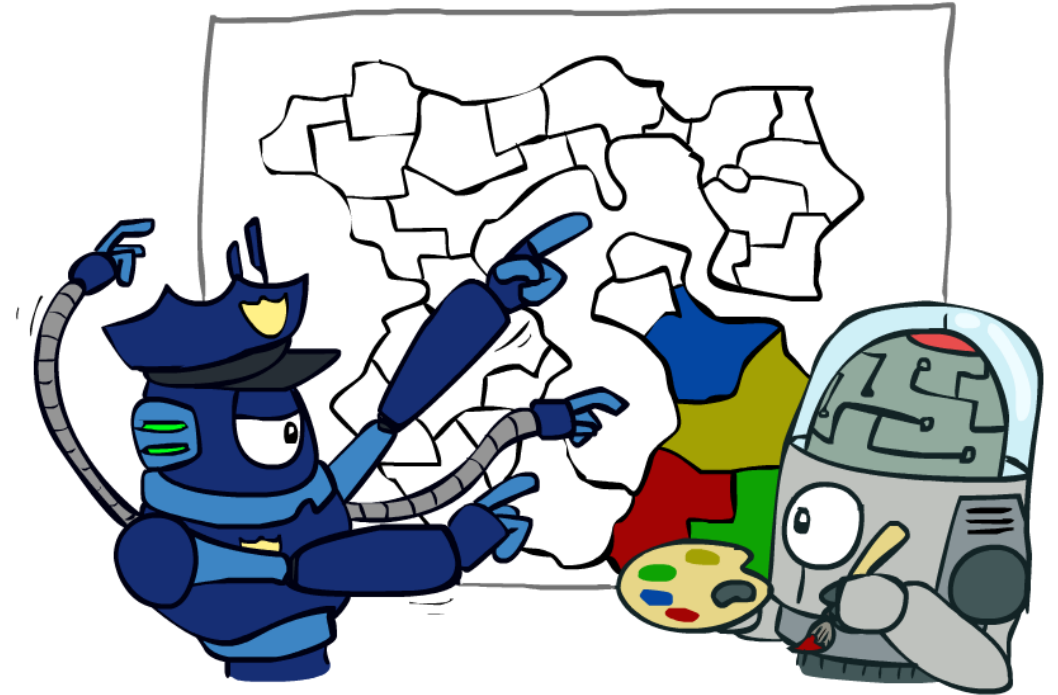
- Binary constraints involve pairs of variables, e.g.:

$$SA \neq WA$$

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

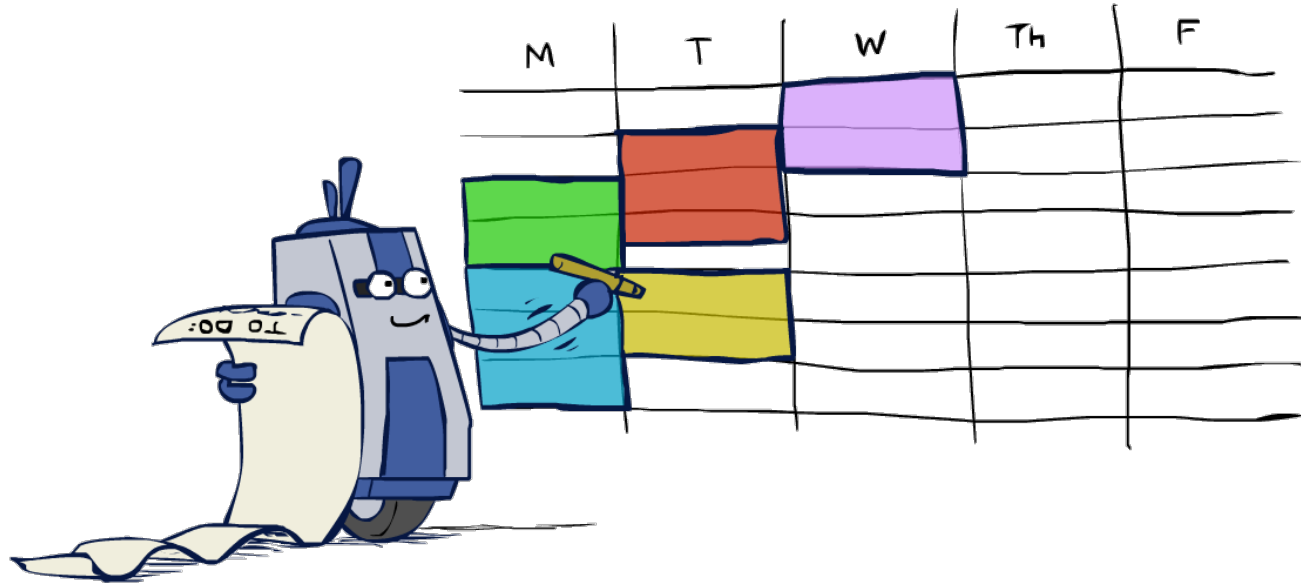
- Preferences (soft constraints):

- E.g., red is better than green
- Often representable by a cost for each variable assignment
- Gives 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!



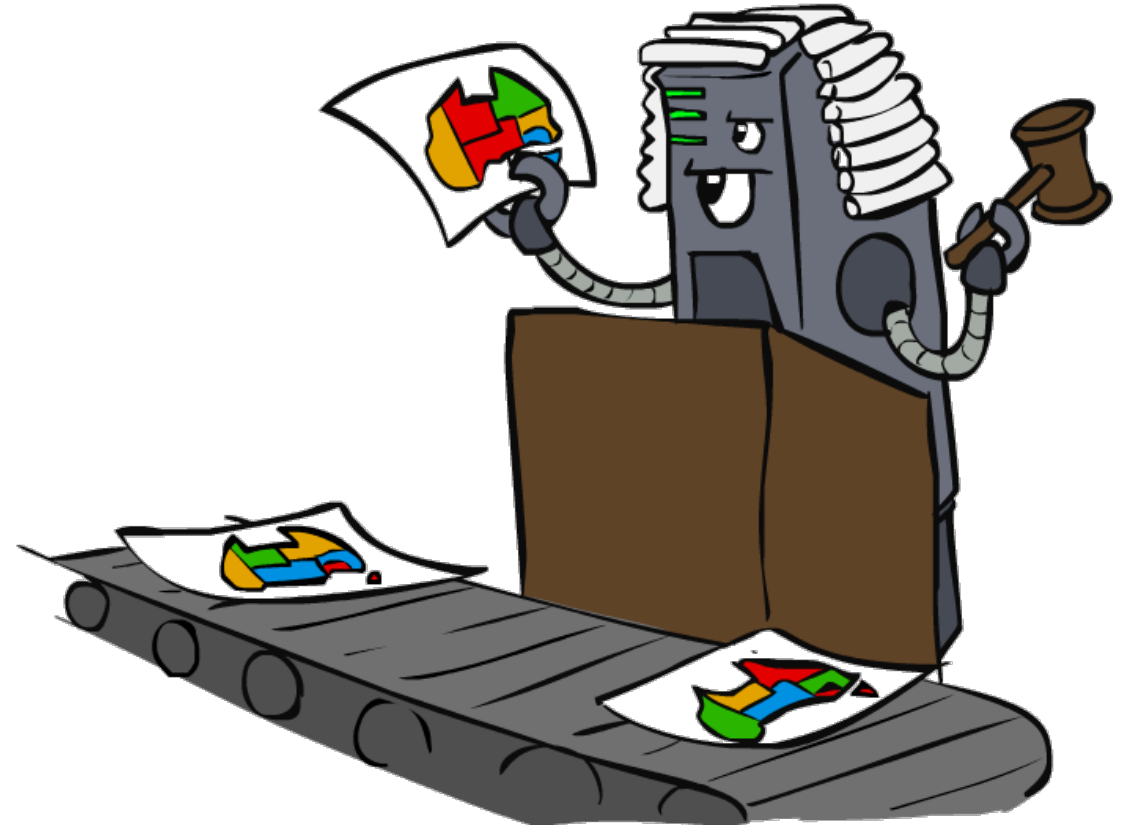
- Many real-world problems involve real-valued variables...

Solving CSPs



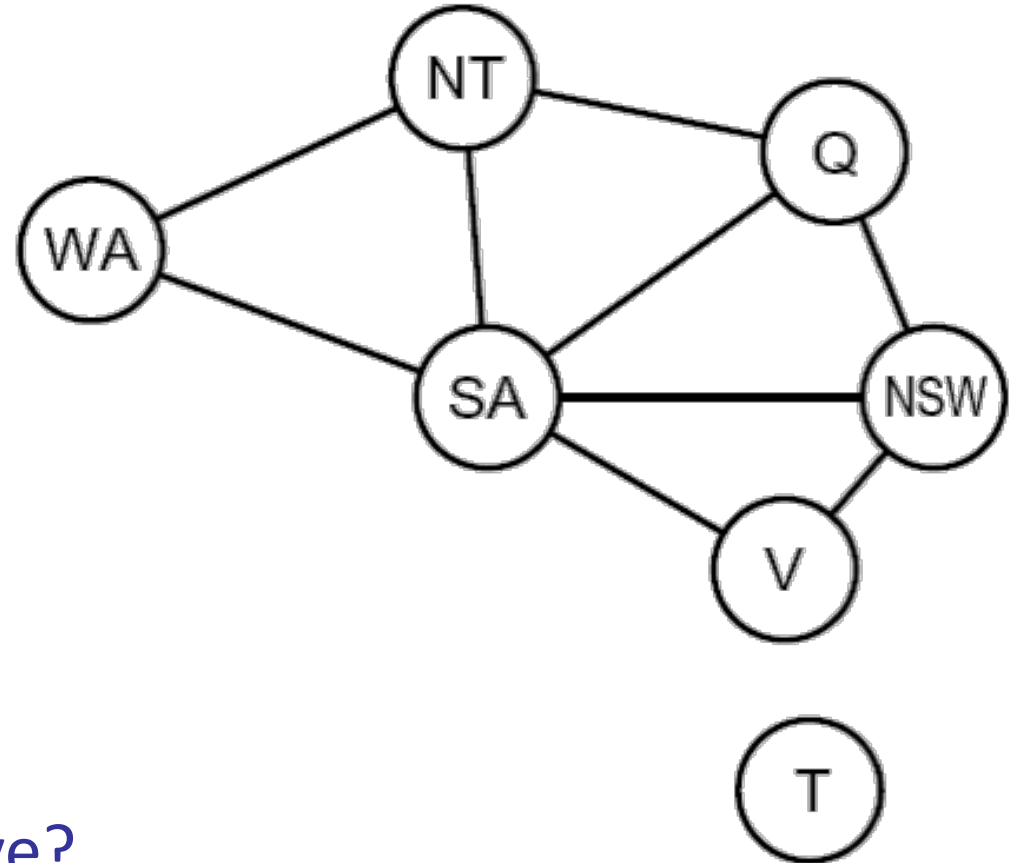
Standard Search Formulation

- 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

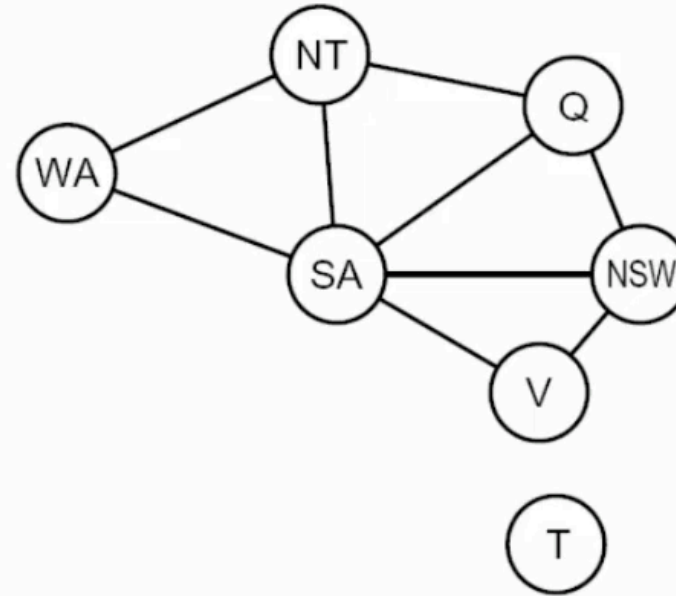
- What would BFS do?
- What would DFS do?
- What problems does naïve search have?



Video of Demo Coloring -- DFS

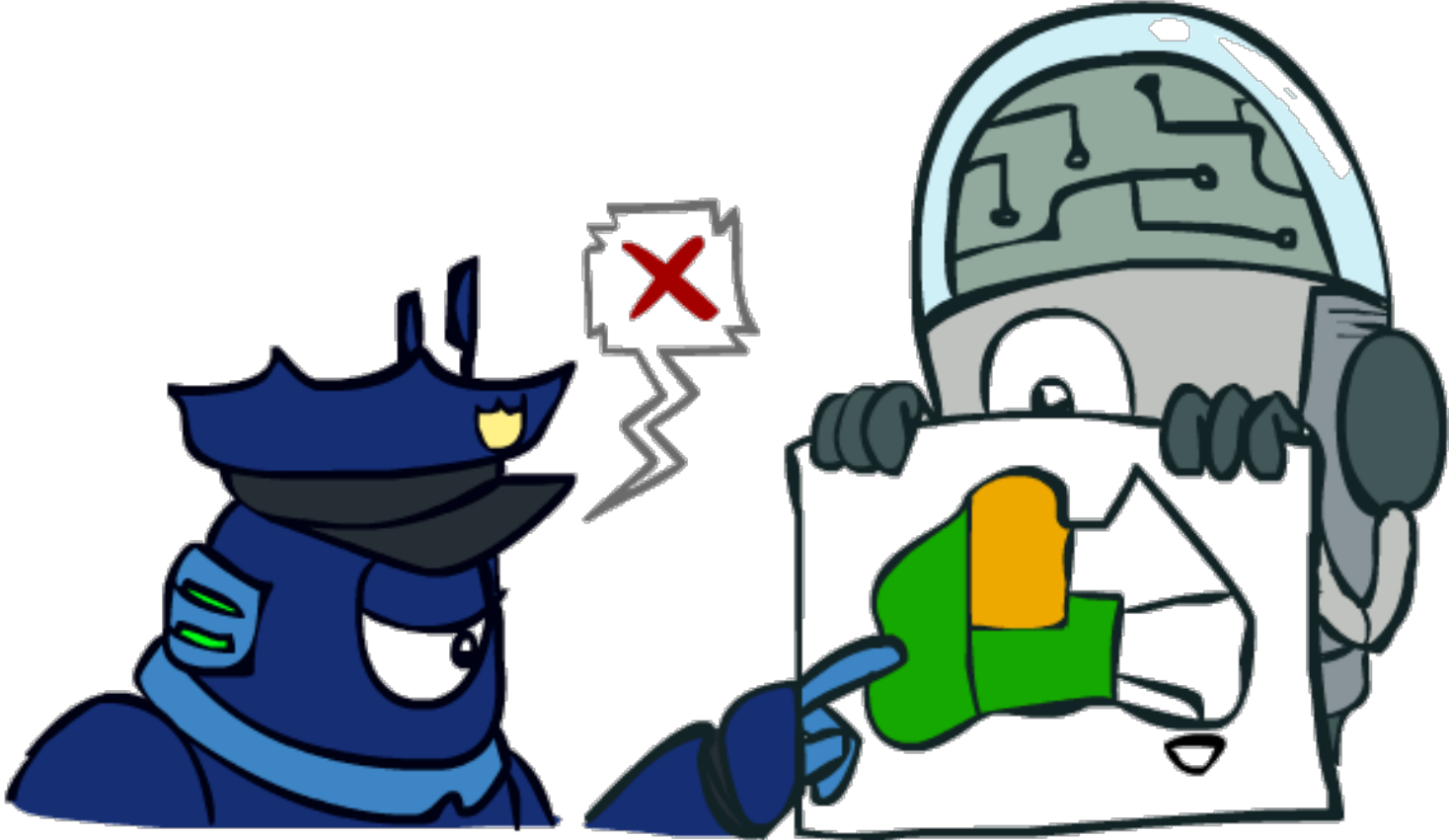
Search Methods

- What would BFS do?
- What would DFS do?



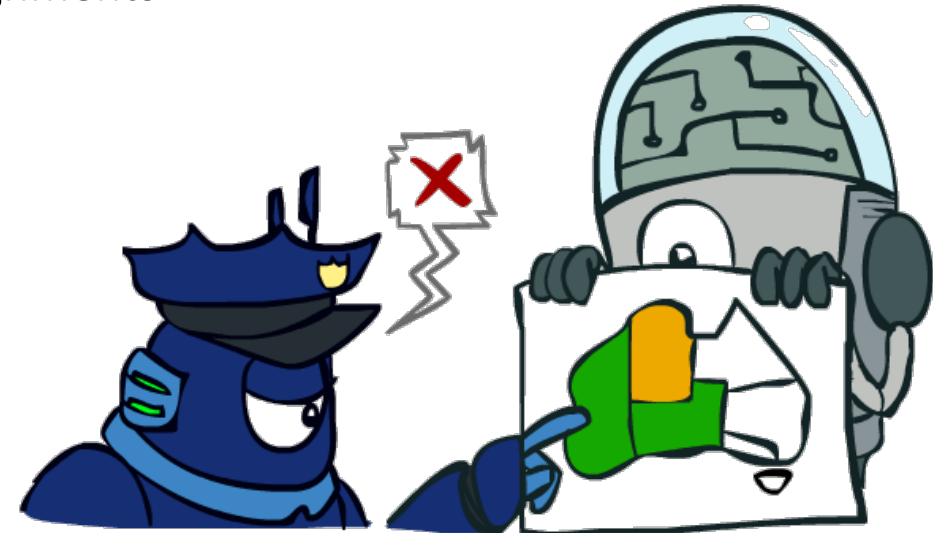
[demo: dfs]

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 $n \approx 25$



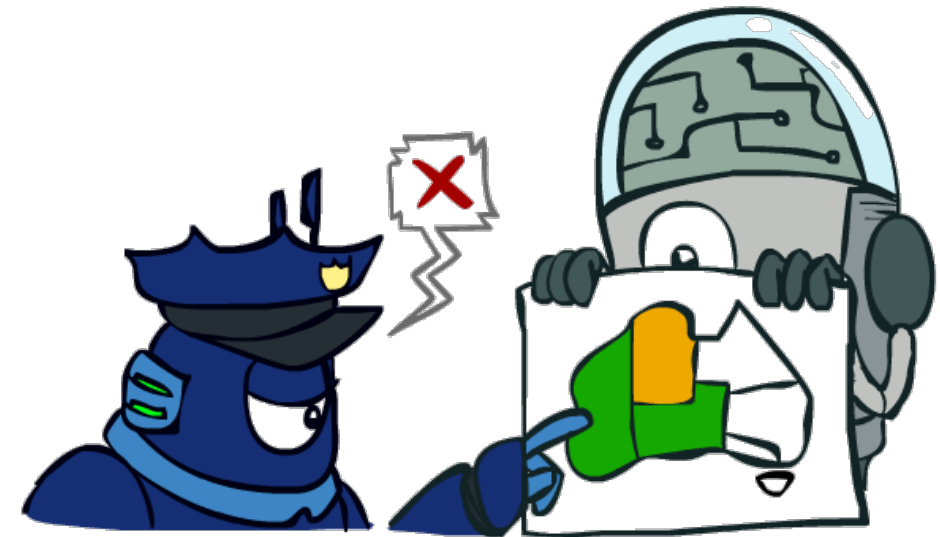
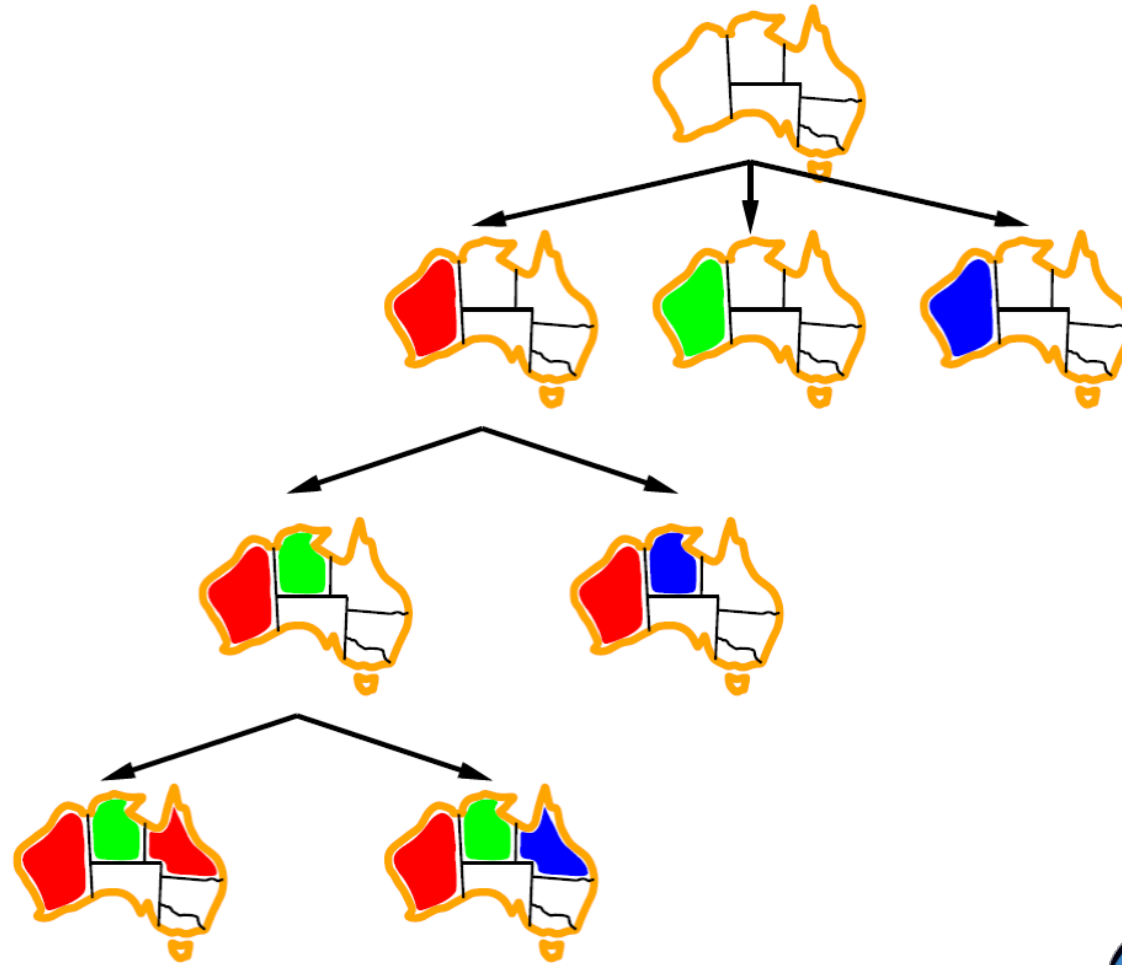
Video of Demo Coloring – Backtracking

The screenshot displays a web browser window with the address bar showing `beta.cs188.org/exercises/csps/forward_checking/forward_checking.html`. The main content area features a 3x3 grid of circular nodes connected by lines, representing a graph. Below the grid are control buttons: `Reset`, `Prev`, `Pause`, `Next`, `Play`, and `Faster`. To the right of the grid is a settings panel with the following sections:

- Graph**: A dropdown menu set to `Simple`.
- Algorithm**: A dropdown menu set to `Backtracking`.
- Ordering**: Radio buttons for `None` (selected), `MRV`, and `MRV with LCV`.
- Filtering**: Radio buttons for `None` (selected), `Forward Checking`, and `Arc Consistency`.
- Speed**: Two input fields: `Speedup` (set to `1` with a multiplier `x`) and `Frame Delay` (set to `700`).

The Windows taskbar at the bottom shows various application icons and system tray information, including the time `11:46 AM` and date `9/4/2012`.

Backtracking Example



Backtracking Search

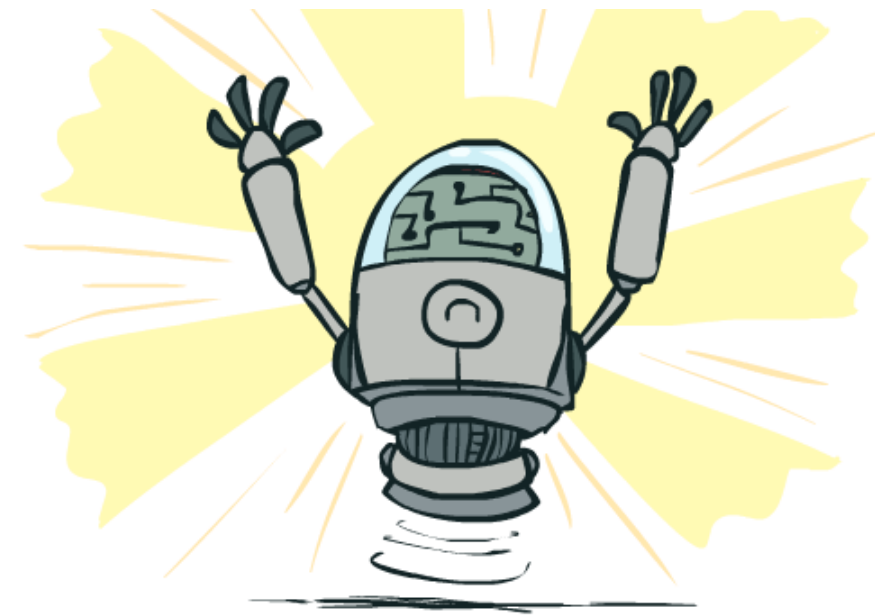
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
  return RECURSIVE-BACKTRACKING({ }, csp)

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?

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

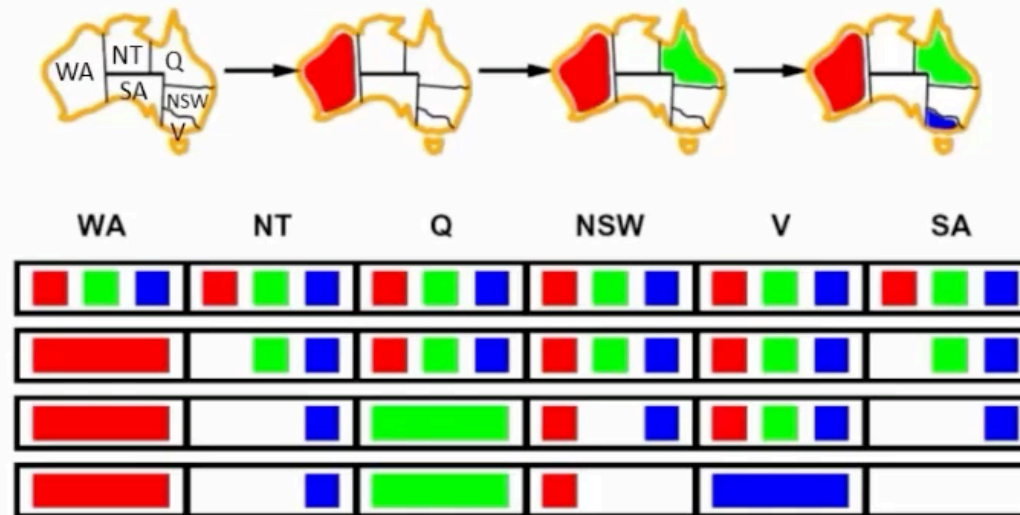
- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



Video of Demo Coloring – Backtracking with Forward Checking

Filtering: Forward Checking

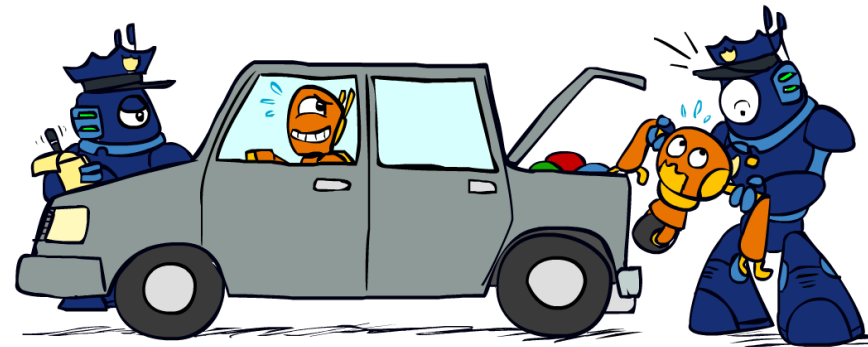
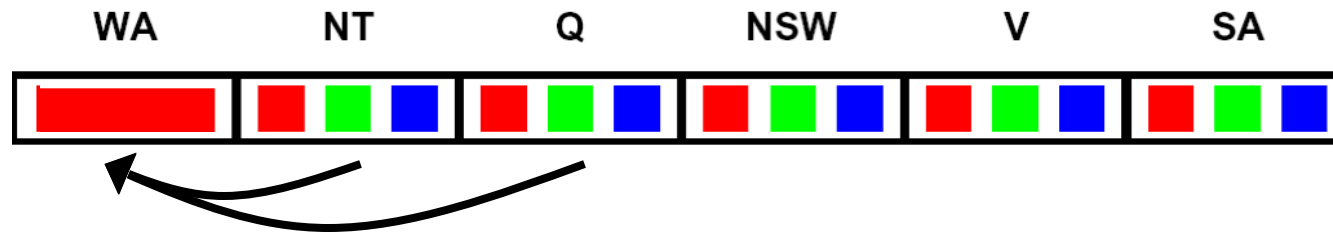
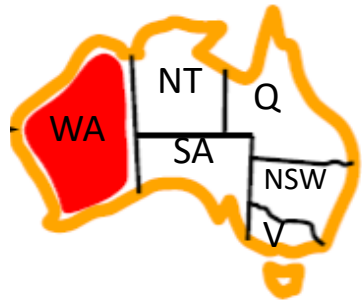
- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



[demo: forward checking]

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

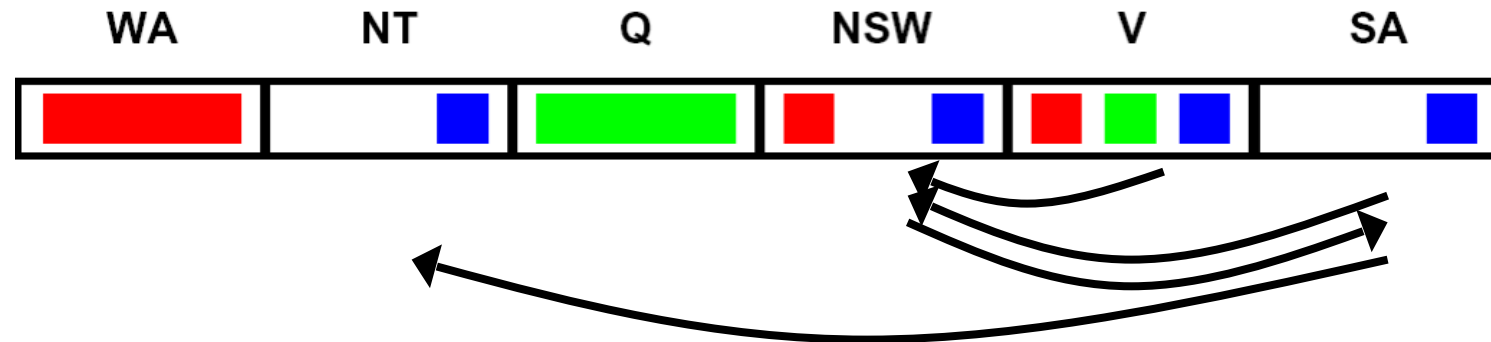
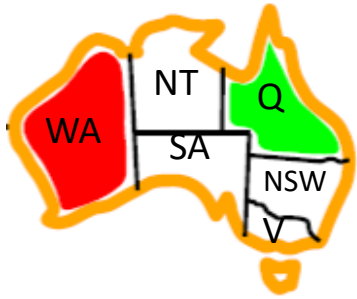


Delete from the tail!

- 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:



- 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  $\{X_1, X_2, \dots, X_n\}$ 
local variables: queue, a queue of arcs, initially all the arcs in csp

while queue is not empty do
   $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\textit{queue})$ 
  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

```

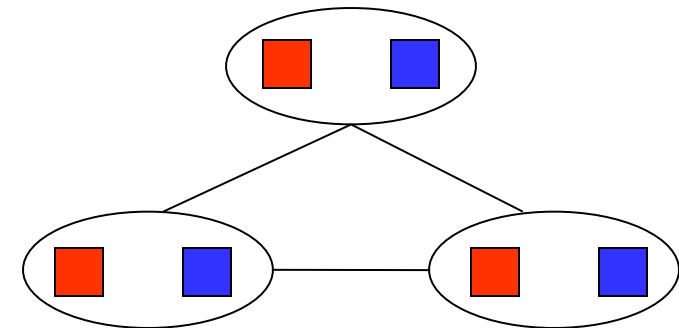
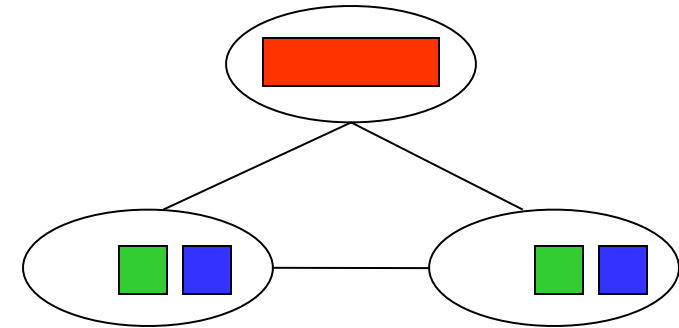
- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- ... but detecting all possible future problems is NP-hard – why?

Video of Demo Arc Consistency – CSP Applet – n Queens

The image shows a screenshot of a computer desktop with two windows open. The left window is titled "CSP Applet Version 4.6.1 --- fiveQueens.xml". It features a menu bar with "File", "Edit", "View", "CSP Options", and "Help". Below the menu is a toolbar with icons for "Fine Step", "Step", "Auto Arc-Consistency", "AutoSolve", "Stop", "Step Back", "Backtrack", and "Reset". The main area contains instructions: "Click on a variable to split its domain.", "Click on a constraint to reorder its variables.", and "Click on an arc to make it arc-consistent." Below the instructions is a constraint network diagram for the 5-Queens problem. The variables are represented by circles labeled A, B, C, D, and E, each with the domain (1 2 3 4 5). The constraints are represented by rectangles labeled "Queens 1" through "Queens 4", connecting the variables. The right window is titled "Note1 - Windows Journal" and contains a handwritten "5-QUEENS" problem. Below the title is a 5x5 grid with columns labeled 1 through 5 and rows labeled A through E. The grid is empty, with arrows pointing to the right in the bottom row. The Windows taskbar is visible at the bottom, showing the Start button, several application icons, and the system tray with a battery indicator at 100%, network and volume icons, and the date/time "12:09 PM 9/4/2012".

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]

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

The screenshot displays a web browser window with the URL `beta.cs188.org/exercises/csp/forward_checking/forward_checking.html`. The main content is a graph coloring interface. On the left, a complex graph is shown with 24 nodes arranged in a grid-like structure. Each node contains four colored dots: red, blue, green, and yellow. The graph is connected by edges, forming a complex network. Below the graph are control buttons: `Reset`, `Prev`, `Pause`, `Next` (with a mouse cursor over it), `Play`, and `Faster`.

On the right side, there are several configuration sections:

- Graph**: A dropdown menu set to `Complex`.
- Algorithm**: A dropdown menu set to `Backtracking`.
- Ordering**: Three radio buttons: `None` (selected), `MRV`, and `MRV with LCV`.
- Filtering**: Three radio buttons: `None`, `Forward Checking` (selected), and `Arc Consistency`.
- Speed**: Two input fields: `Speedup` (set to `1`) and `Frame Delay` (set to `700`).

The bottom of the image shows a Windows taskbar with various application icons and a system tray on the right displaying `100%` battery, network status, and the date/time `12:14 PM 9/4/2012`.

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

The screenshot displays a web browser window with the URL `beta.cs188.org/exercises/csps/forward_checking/forward_checking.html`. The main content is a graph coloring interface. On the left, a complex graph is shown with nodes represented as circles containing three colored dots (red, blue, green). The graph consists of a top row of six nodes, a middle section of nodes connected in a grid-like pattern, and a bottom row of six nodes. On the right, there are control panels:

- Graph:** A dropdown menu set to "Complex".
- Algorithm:** A dropdown menu set to "Backtracking".
- Ordering:** Radio buttons for "None", "MRV", and "MRV with LCV".
- Filtering:** Radio buttons for "None", "Forward Checking", and "Arc Consistency".
- Speed:** "Speedup" set to "1 x" and "Frame Delay" set to "700".

Below the graph, there are buttons for "Reset", "Prev", "Pause", "Next", "Play", and "Faster". The Windows taskbar at the bottom shows the system tray with a 100% battery indicator, network and volume icons, and the date/time "12:15 PM 9/4/2012".

Ordering

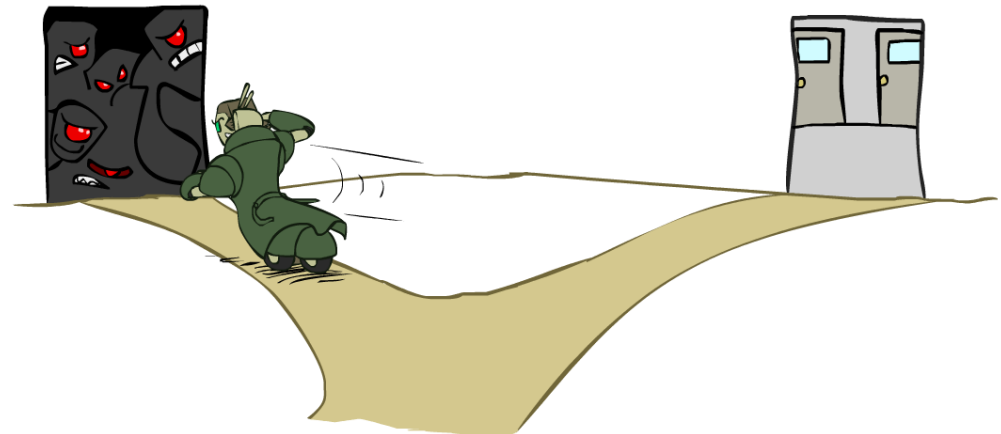


Ordering: Minimum Remaining Values

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

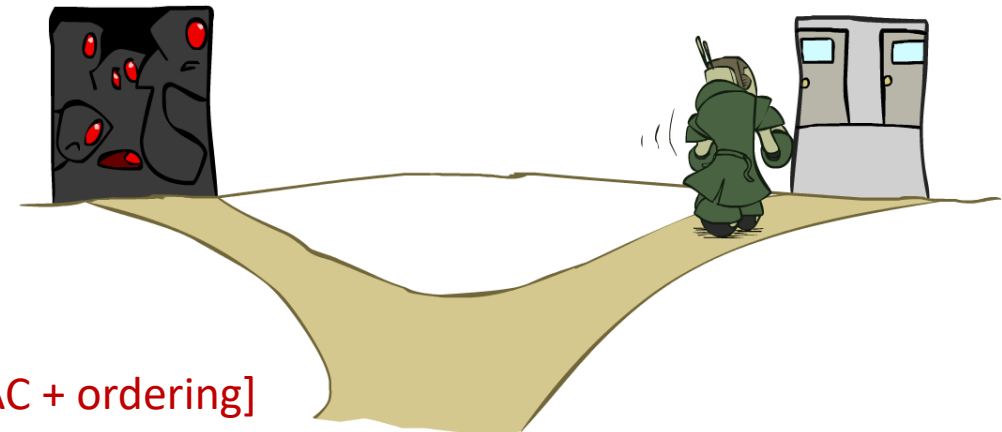
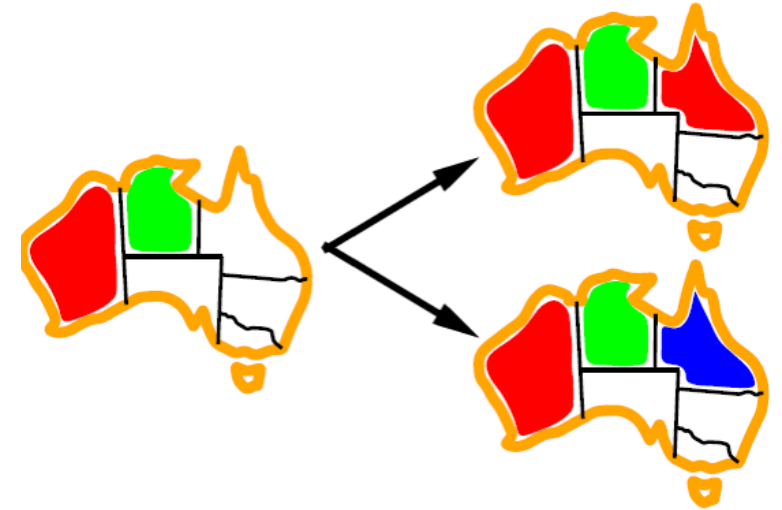


- Why min rather than max?
- Also called “most constrained variable”
- “Fail-fast” ordering



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?
- Combining these ordering ideas makes 1000 queens feasible



[Demo: coloring – backtracking + AC + ordering]