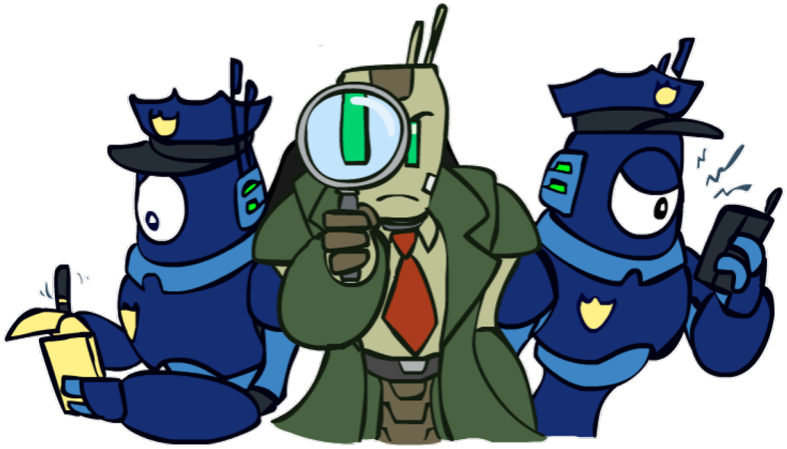
Announcements

- Project 1 and Homework 1 are out
  - A piece of advice: Start early!
- Project 1 is long
  - Real test of your programming skills and understanding of search
  - Make use of existing functions at Util.py and other files

CS 3568: Intelligent Systems

Constraint Satisfaction Problems (Part 1)



Instructor: Tara Salman

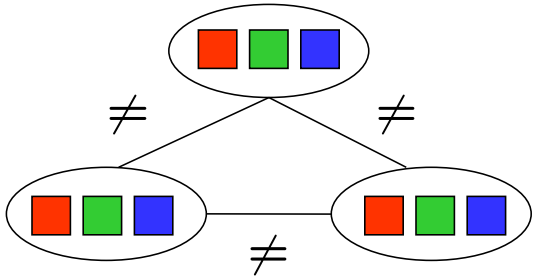
Texas Tech University

Computer Science Department

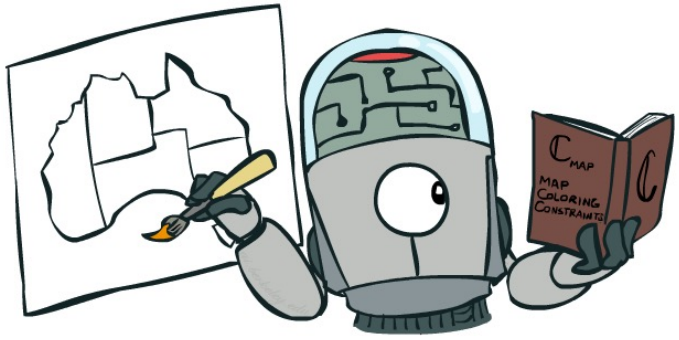
[These slides were created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley (ai.berkeley.edu).]

Reminder: CSPs

- CSPs:
  - Variables
  - Domains
  - Constraints
    - Implicit (provide code to compute)
    - Explicit (provide a list of the legal tuples)



- Goals:
  - This class: find any solution
  - Also: find all, find best, etc.



Example: Map Coloring

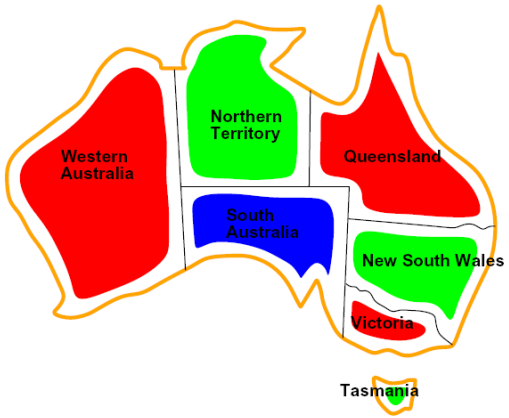
- Variables: $\mathrm{WA}, \mathrm{NT}, \mathrm{Q}, \mathrm{NSW}, \mathrm{V}, \mathrm{SA}, \mathrm{T}$
- Domains: $\mathrm{D} = \{\mathrm{red}, \mathrm{green}, \mathrm{blue}\}$
- Constraints: adjacent regions must have different colors

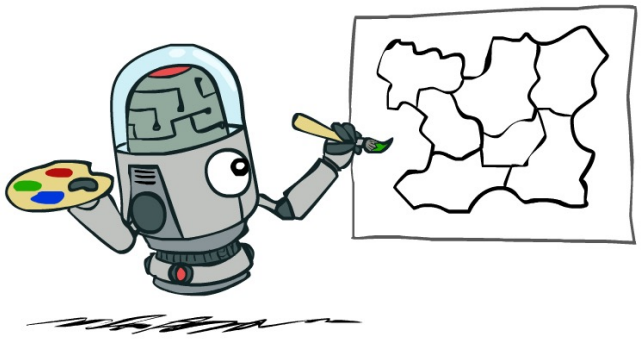
  Implicit: $\mathrm{WA} \neq \mathrm{NT}$

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

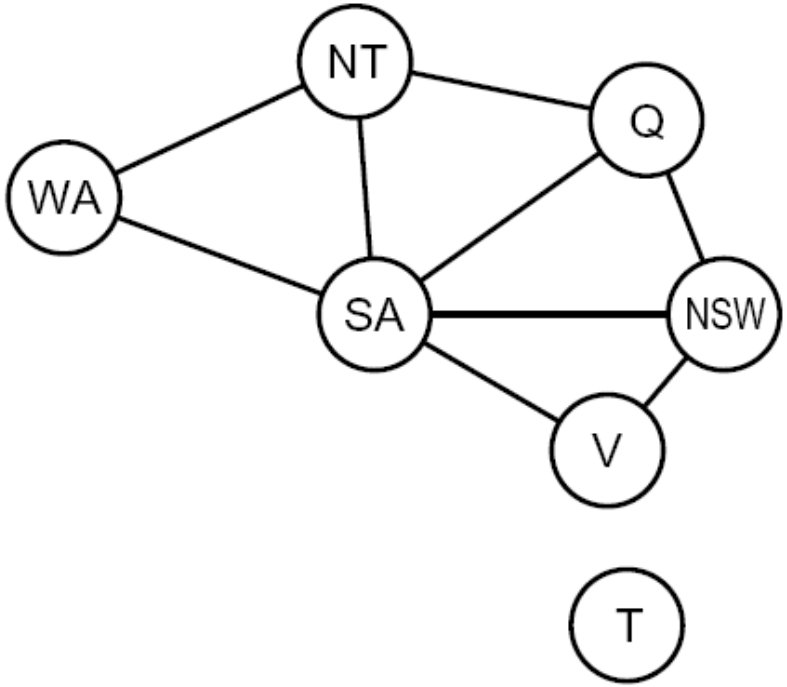
- Solutions are assignments satisfying all constraints, e.g.:

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



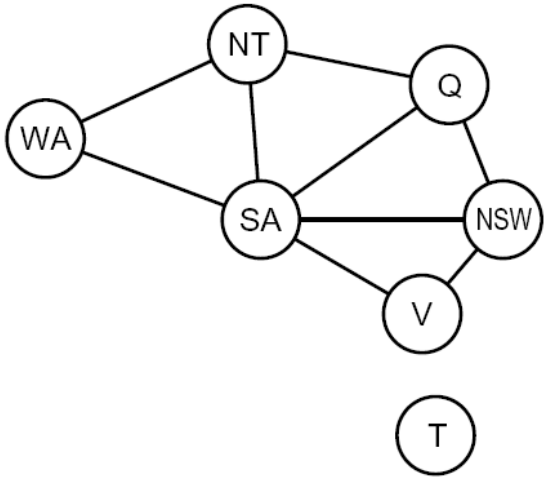


Constraint Graphs

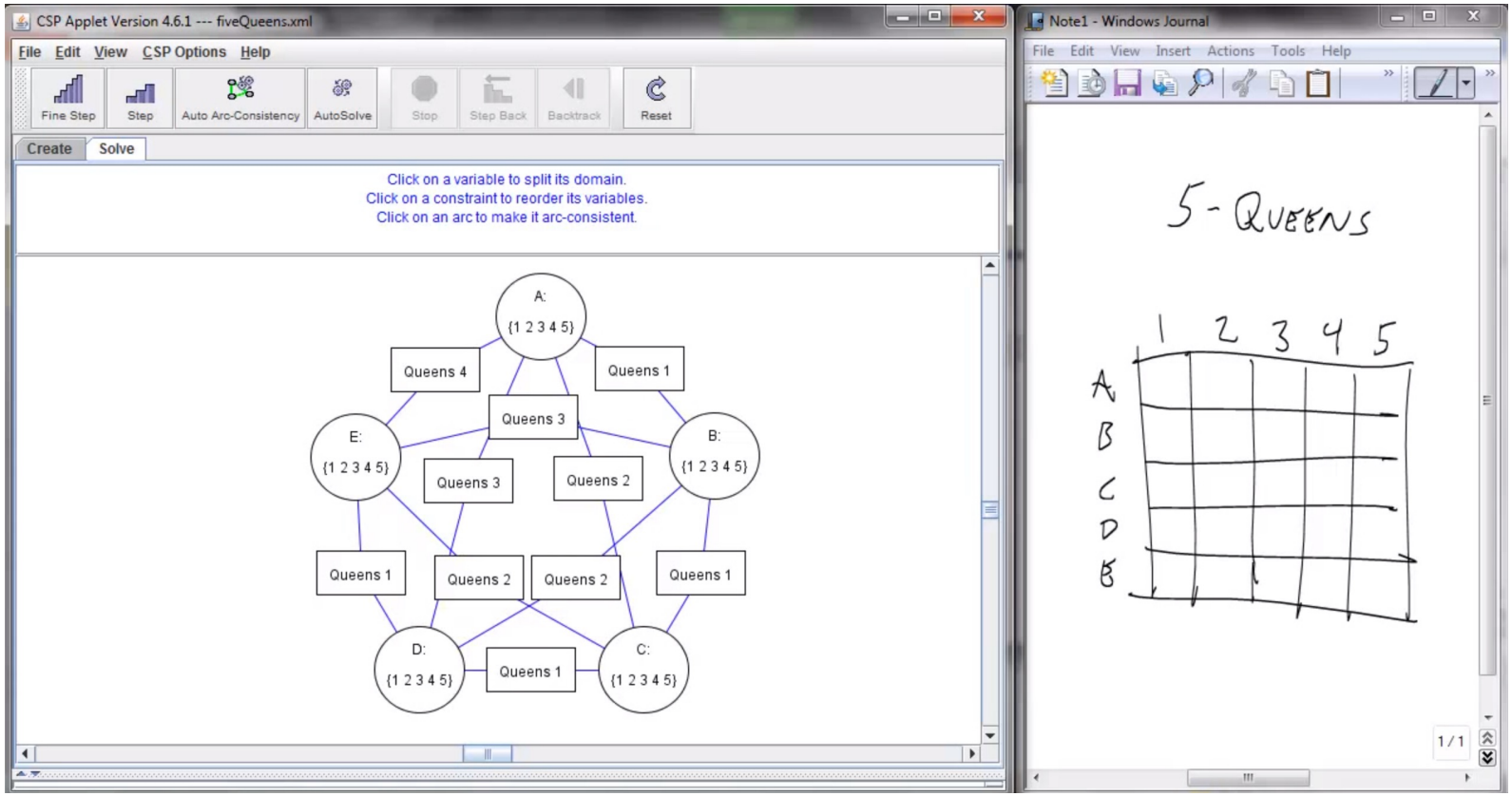


Constraint Graphs

- 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!



Screenshot of Demo N-Queens



Example: Cryptarithmetic

- Variables:

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

- Domains:

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

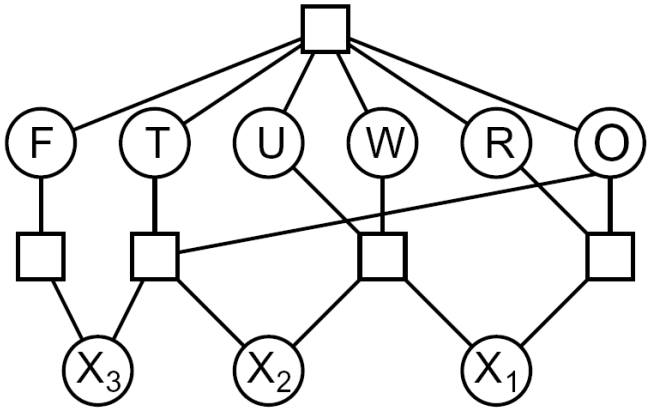
- Constraints:

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

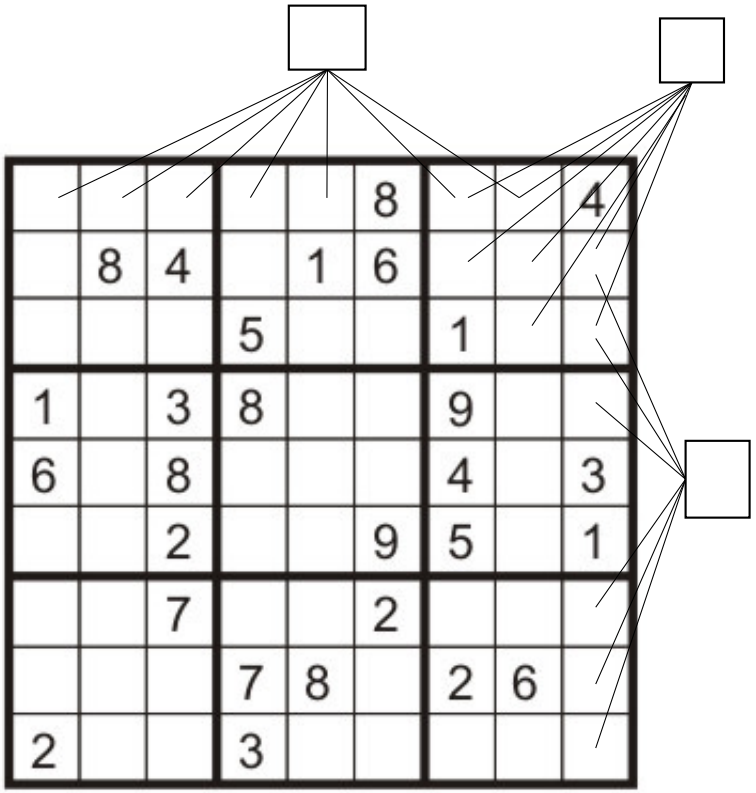
$$O + O = R + 10 \cdot X_1$$

$$\cdots$$

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



Example: Sudoku



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

  9-way alldiff for each column

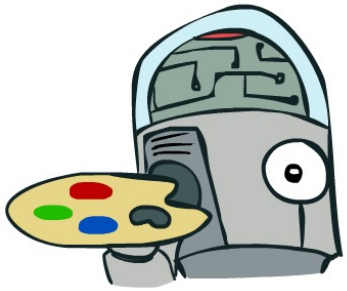
  9-way alldiff for each row

  9-way alldiff for each region

  (or can have a bunch of pairwise inequality constraints)

Varieties of CSPs and Constraints

Varieties of CSPs



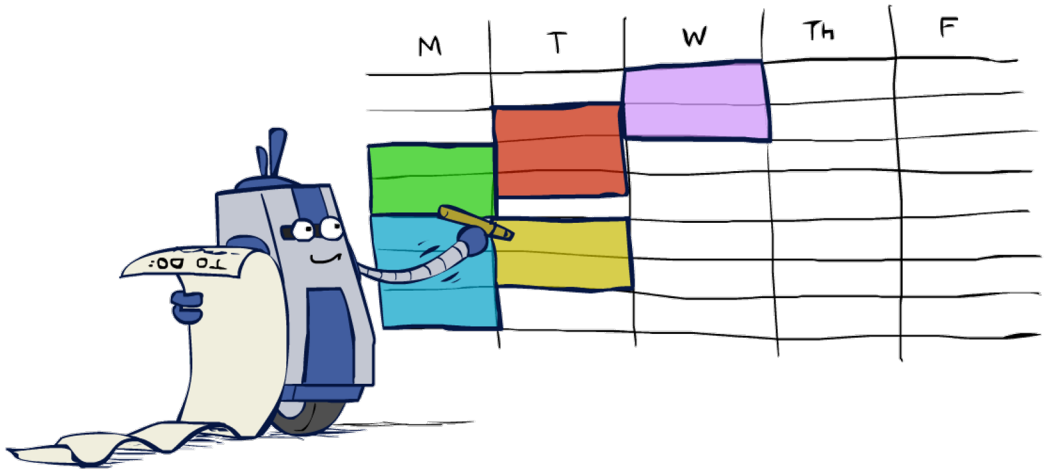
- Variables
  - Finite domains
  - Infinite domains (integers, strings, etc.)
  - Continuous variables

- 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!

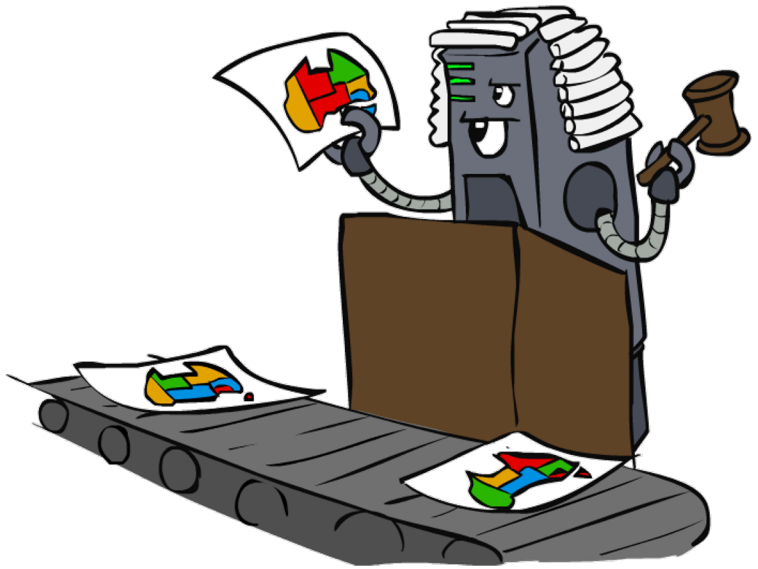


- Many real-world problems involve real-valued variables…

Solving CSPs

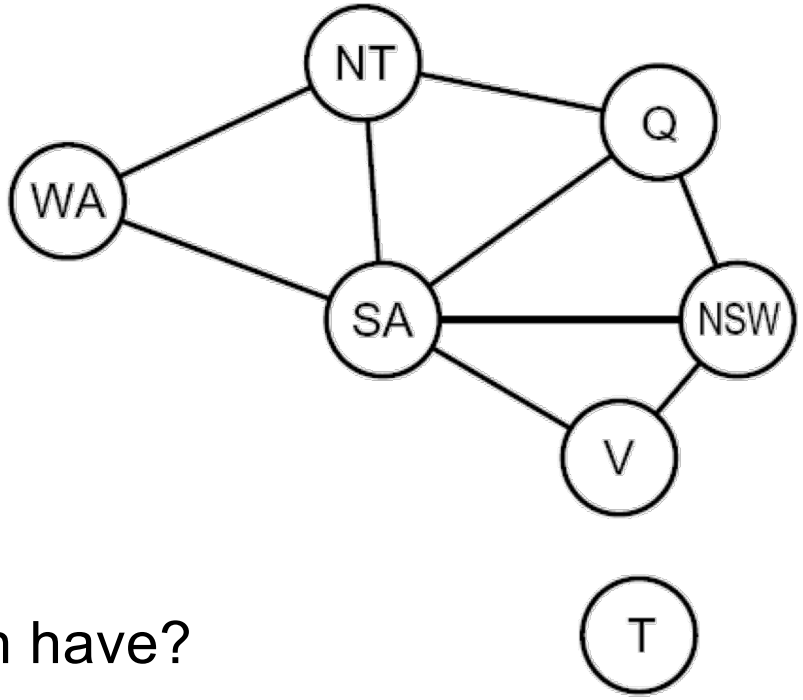
Standard Search Formulation

- 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



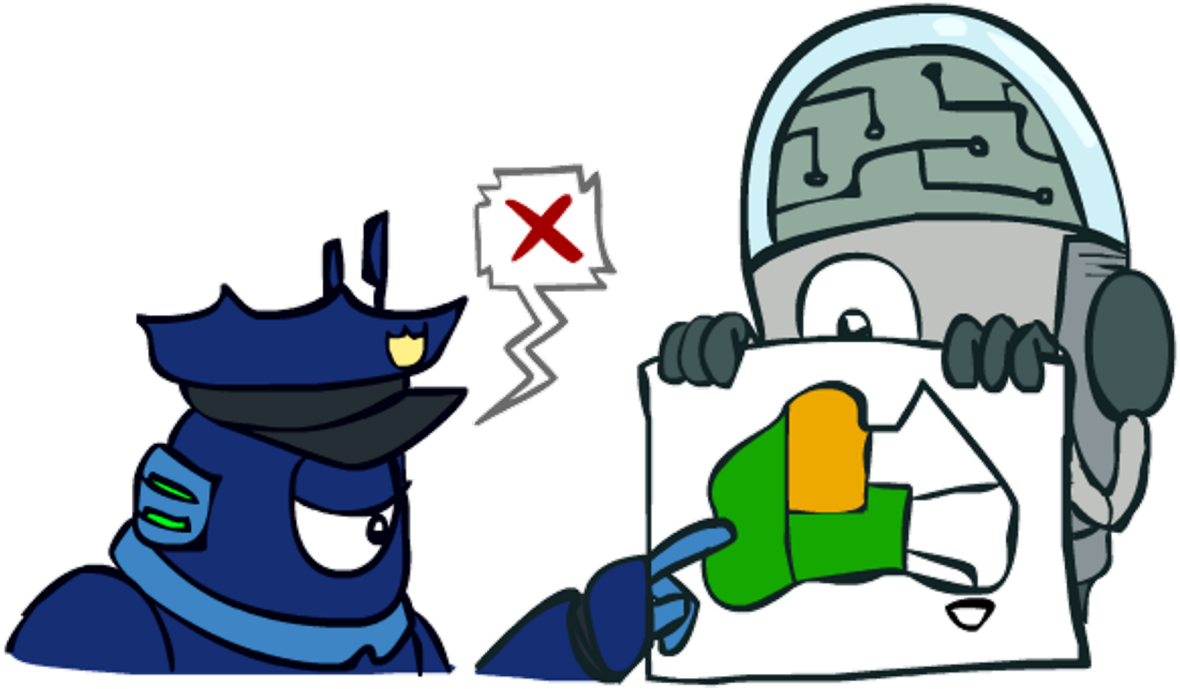
Search Methods

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



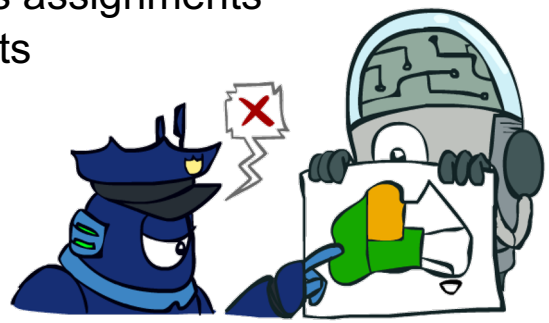
https://inst.eecs.berkeley.edu/~cs188/fa19/assets/demos/csp/csp_demos.html

Backtracking Search

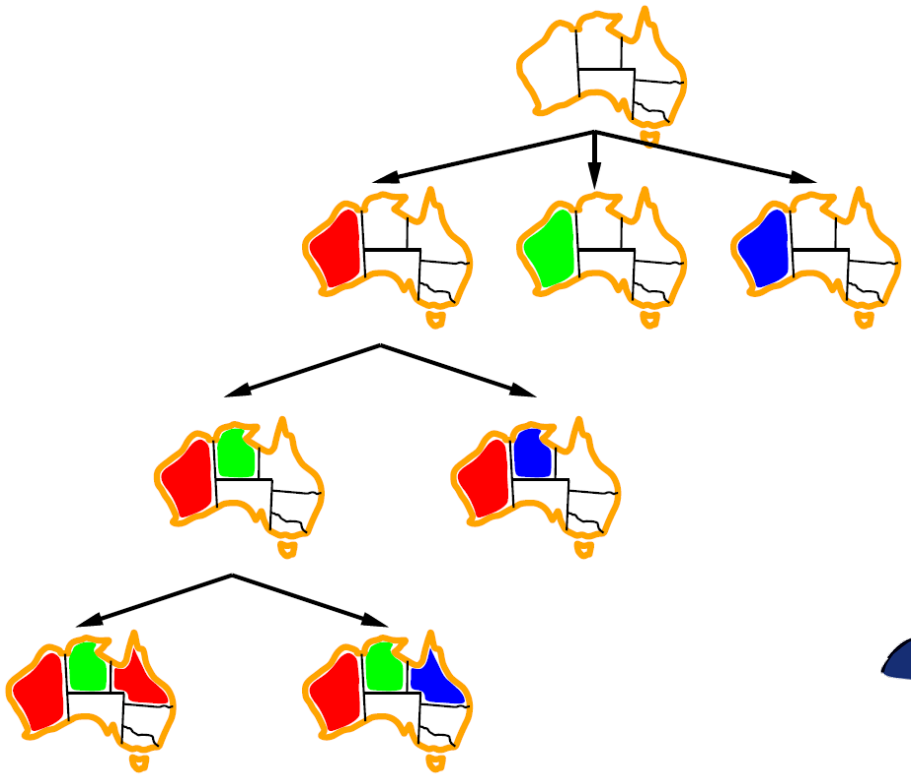


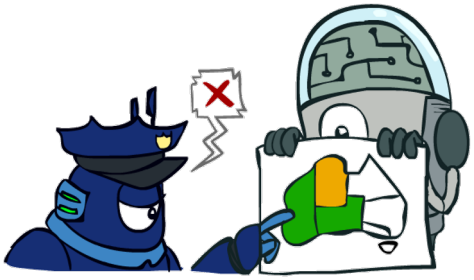
Backtracking Search

- 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

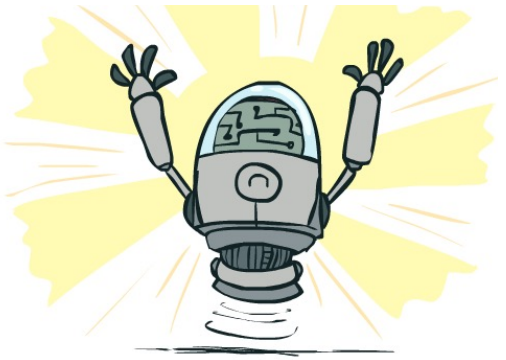
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

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

https://inst.eecs.berkeley.edu/~cs188/fa19/assets/demos/csp/csp_demos.html

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?

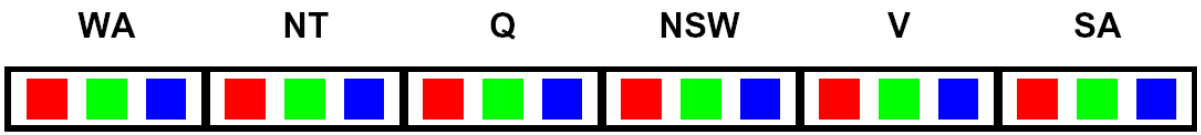


- Filtering: Can we detect inevitable failure early?
- Structure: Can we exploit the problem structure?

Filtering

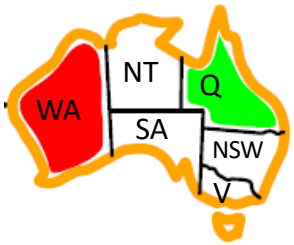
Filtering: Forward Checking

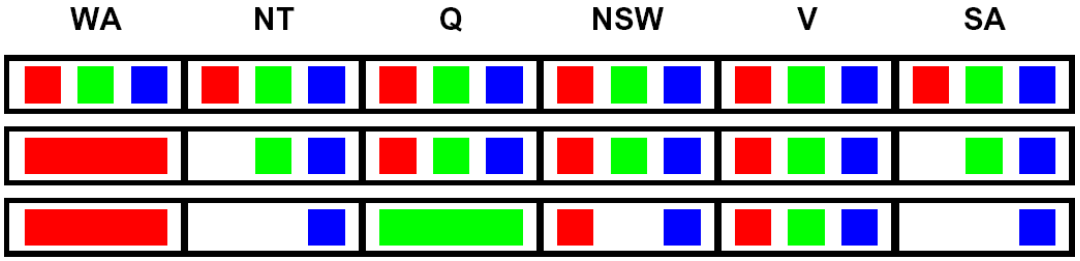
- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



Filtering: Constraint Propagation

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



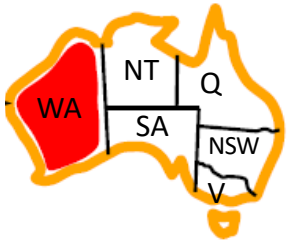


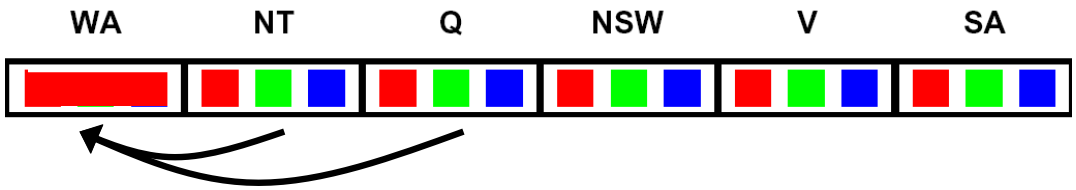
- NT and SA cannot both be blue!
- Why didn't we detect this yet?
- *Constraint propagation:* reason from constraint to constraint

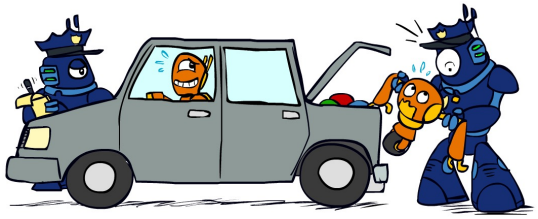
https://inst.eecs.berkeley.edu/~cs188/fa19/assets/demos/csp/csp_demos.html

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





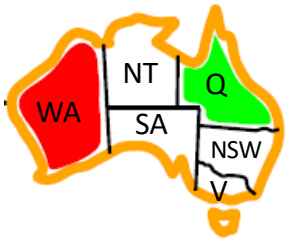


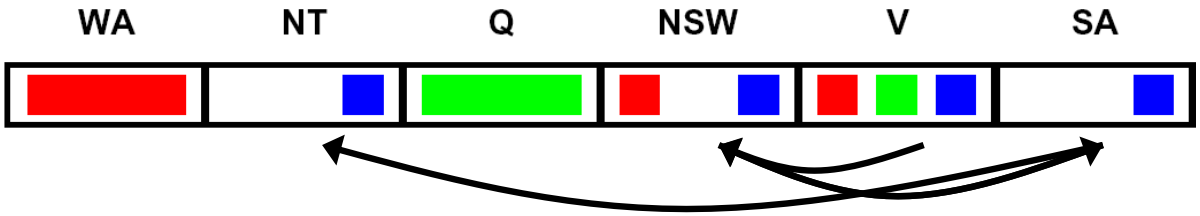
Delete from the tail!

- 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!

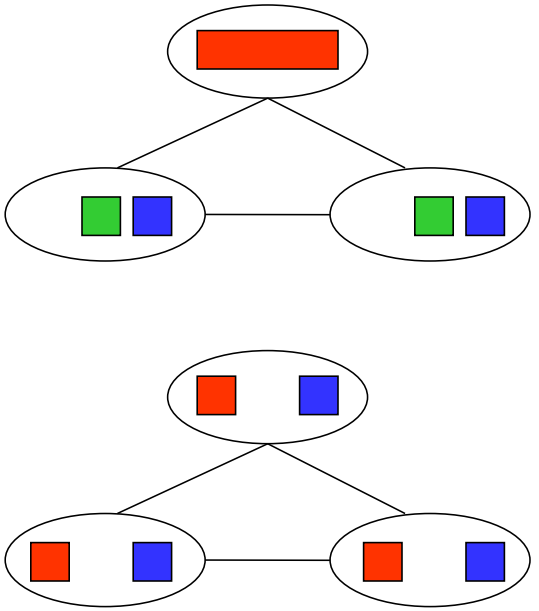
Enforcing Arc Consistency in a CSP

```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X_1, X_2, ..., 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!

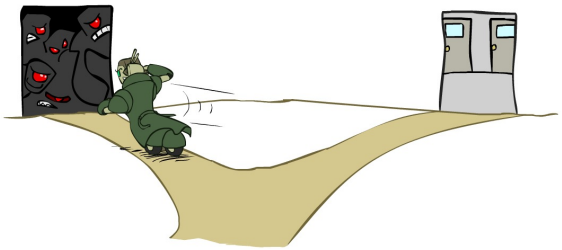
https://inst.eecs.berkeley.edu/~cs188/fa19/assets/demos/csp/csp_demos.html

Ordering

Ordering: Minimum Remaining Values

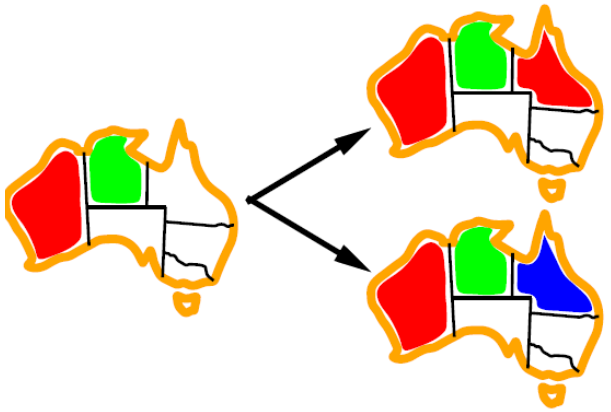
- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain

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

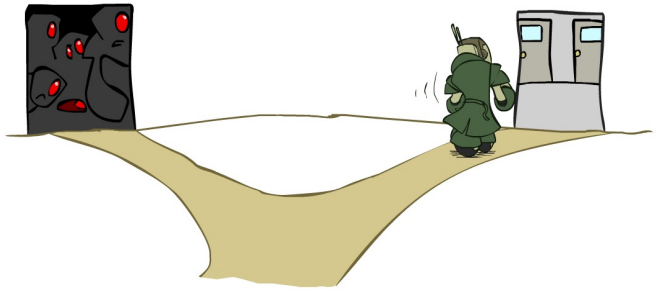


Ordering: Least Constraining Value

- 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?

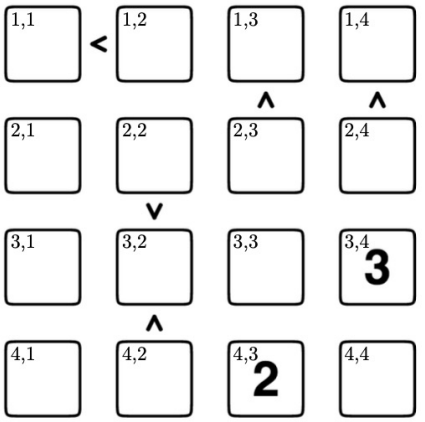


- Combining these ordering ideas makes 1000 queens feasible

Review Questions

Review Problem 1

- Futoshiki is a Japanese logic puzzle that is very simple, but can be quite challenging. You are given an n x n grid, and must place the numbers 1, . . . n in the grid such that every row and column has exactly one of each. Additionally, the assignment must satisfy the inequalities placed between some adjacent squares.
- To the right is an instance of this problem, for size n = 4. Some of the squares have known values, such that the puzzle has a unique solution. (The letters mean nothing to the puzzle, and will be used only as labels with which to refer to certain squares). Note also that in- equalities apply only to the two adjacent squares, and do not directly constrain other squares in the row or column
- Let's formulate this puzzle as a CSP. We will use 42 variables, one for each cell, with Xij as the variable for the cell in the ith row and jth column (each cell contains its i,j label in the top left corner). The only unary constraints will be those assigning the known initial values to their respective squares (e.g. X34 = 3).



Review Problem 1 Question

1. Complete the formulation of the CSP using only binary constraints (in addition to the unary constraints. In particular, describe the domains of the variables, and all binary constraints you think are necessary. You do not need to enumerate them all, just describe them using concise mathematical notation. You are not permitted to use n-ary constraints where $n \geq 3$. PS. You are required to write down variables, domains, and constraints.
2. After enforcing unary constraints, consider the binary constraints involving X_{14} and X_{24}. Enforce arc consistency on just these constraints and state the resulting domains for the two variables
3. Suppose we enforced unary constraints and ran arc consistency on the initial CSP in the figure above. What is the maximum possible domain size for a variable adjacent to an inequality?
4. By inspection of column 2, we find it is necessary that $X_{32} = 1$, despite not having found an assignment to any of the other cells in that column. Would running arc consistency find this requirement? Explain why or why not.