CS 348
Intro to Artificial Intelligence

Day 6
Constraint Satisfaction

(slides based on Downy, Sood, Dan Klein, Pieter Abbeel)

- Class business
  - Lab 1 grades on canvas (60% of class received 100/100)
    - First resubmission due today 7pm
  - Lab 2 due Thursday
  - Lab 3 due 4/28 7pm (A*)

- Constraint satisfaction(6.1-6.6)
- Introduce lab 4 (Tic-Tac-Toe)
- Answer lab 2 questions

Homework submission

- 1 time no questions asked late submission, 143 hours extra time
  - Save for emergencies
- Please check your submitted code on NU servers
  - Simple errors need to be corrected with the first resubmission
- Feedback and testcases only provided after first resubmission deadline.
- Please don’t copy any of the main.py functions / data into submitted student code.
  - Can interfere with autograder.

What is Search For?

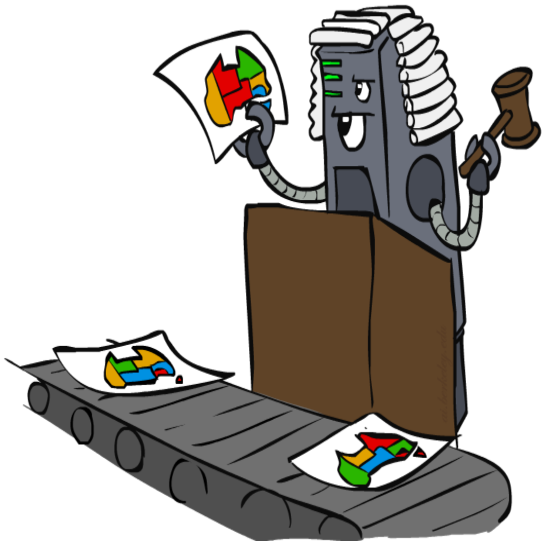
- Assumptions about the world: a single agent, deterministic actions, fully observed state, discrete state space

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

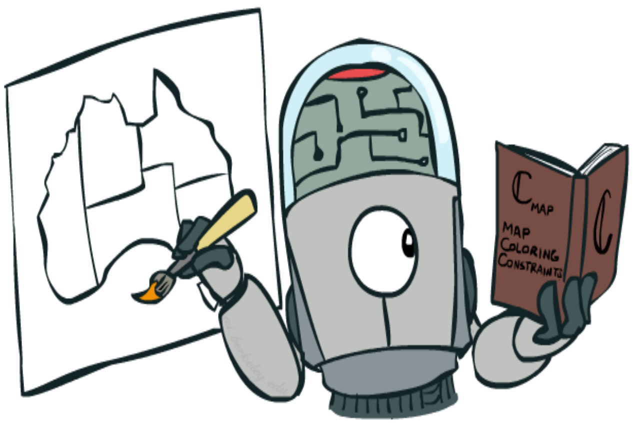
- Identification: assignments to variables
  - The goal itself is important, not the path
  - All paths at the same depth (for some formulations)
  - CSPs are a specialized form of identification problems

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

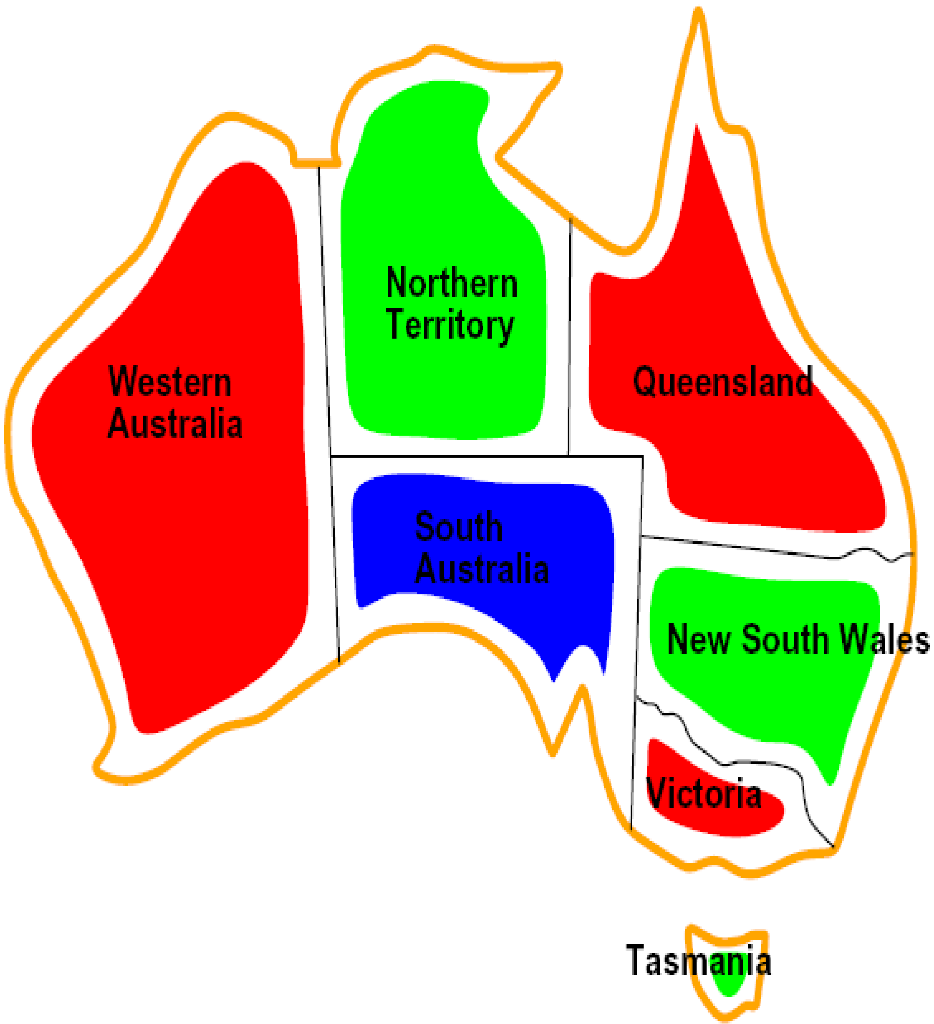


- Constraint satisfaction problems (CSPs):
  - A special subset of search problems
  - State is defined by variables X_i with values from a domain D (sometimes D depends on i)
  - Goal test is a set of constraints specifying allowable combinations of values for subsets of variables



- Allows useful general-purpose algorithms with more power than standard search algorithms

CSP Examples



Example: Map Coloring

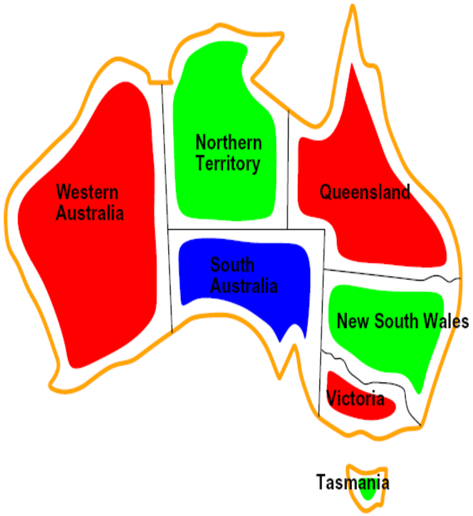
- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: $D = \{\text{red}, \text{green}, \text{blue}\}$
- Constraints: adjacent regions must have different colors

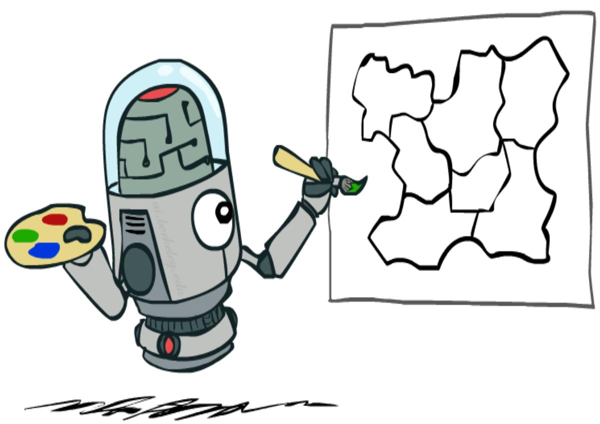
  Implicit: $WA \neq NT$

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

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

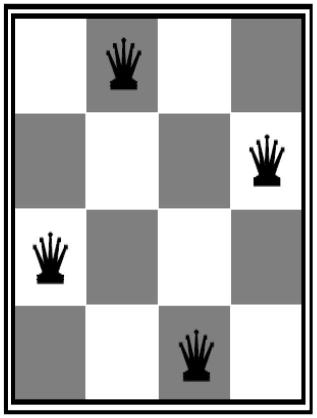
  $\{WA{=}\text{red},\ NT{=}\text{green},\ Q{=}\text{red},\ NSW{=}\text{green},\ V{=}\text{red},\ SA{=}\text{blue},\ T{=}\text{green}\}$

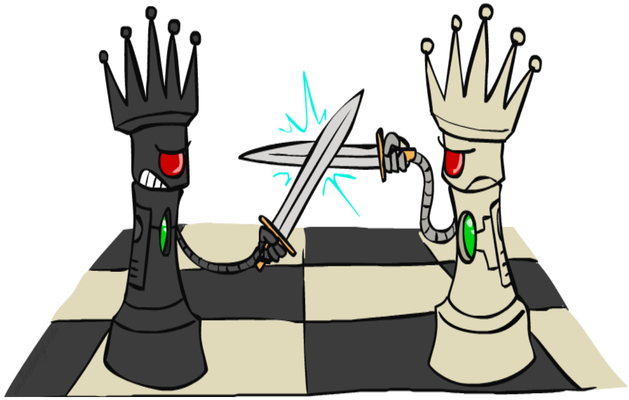




Example: N-Queens

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





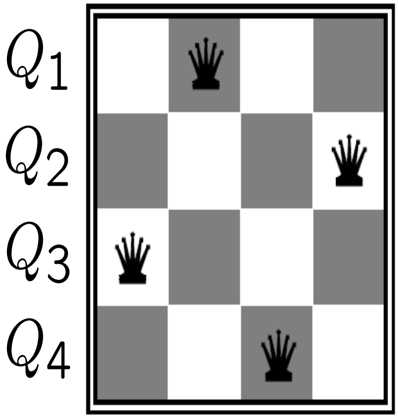
$$\forall i, j, k \;\; (X_{ij}, X_{ik}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \;\; (X_{ij}, X_{kj}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \;\; (X_{ij}, X_{i+k,j+k}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \;\; (X_{ij}, X_{i+k,j-k}) \in \{(0,0), (0,1), (1,0)\}$$

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

$\forall i$ For all instances of i

Example: N-Queens

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

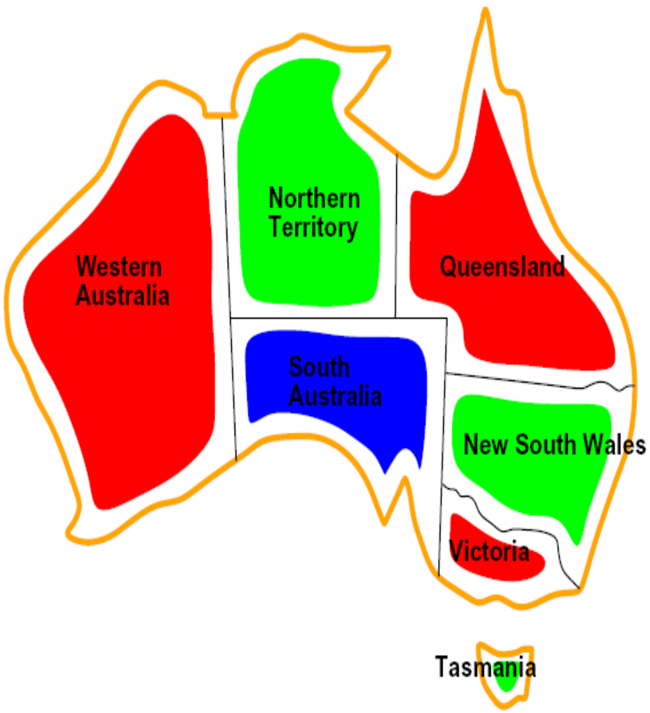


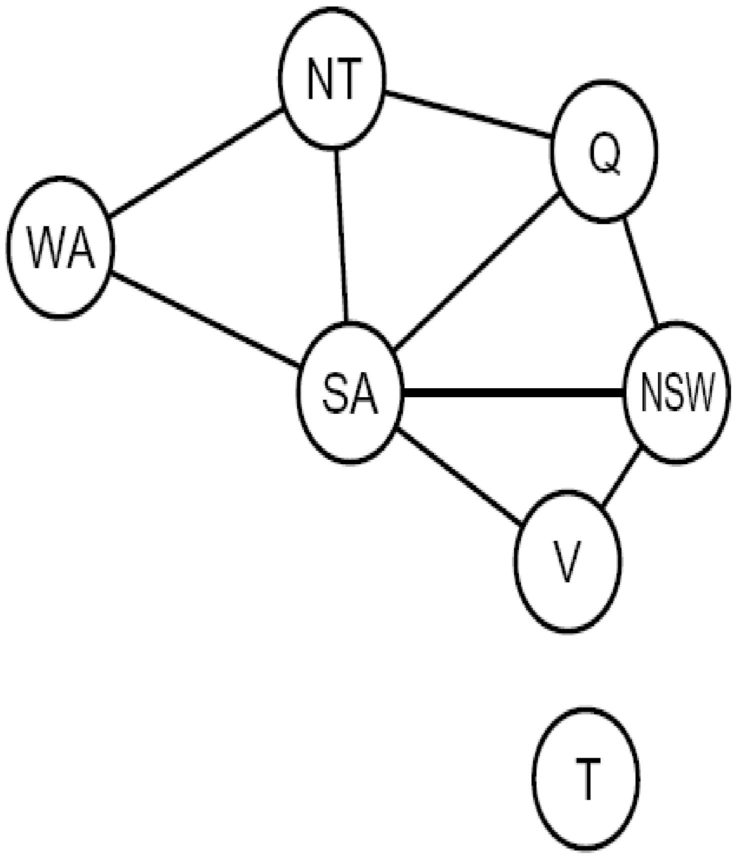
Implicit: $\forall i, j$ non-threatening(Q_i, Q_j)

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

$\cdots$

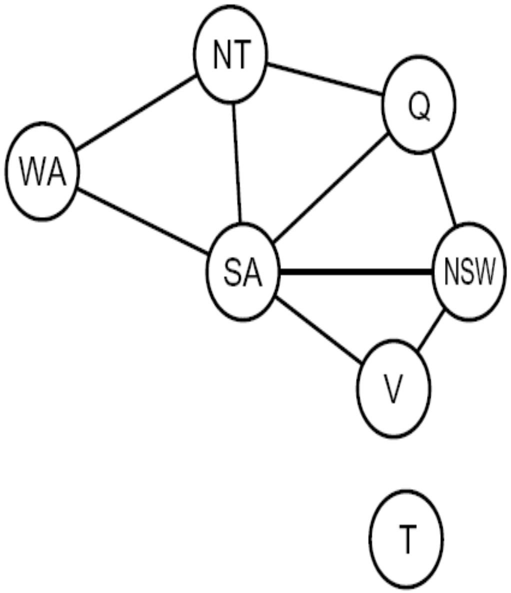
Constraint Graphs

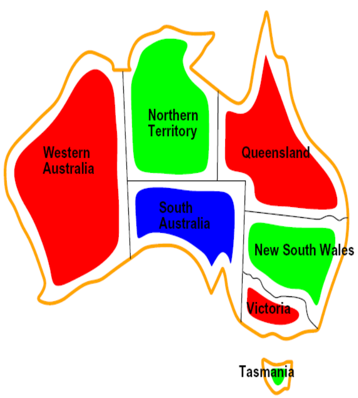




Constraint Graphs

- 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: Cryptarithmetic

- Variables:

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

- Domains:

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

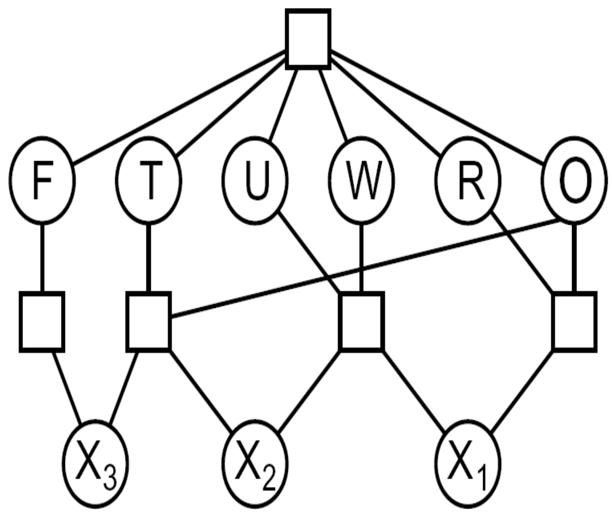
- Constraints:

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

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

  $\ldots$

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

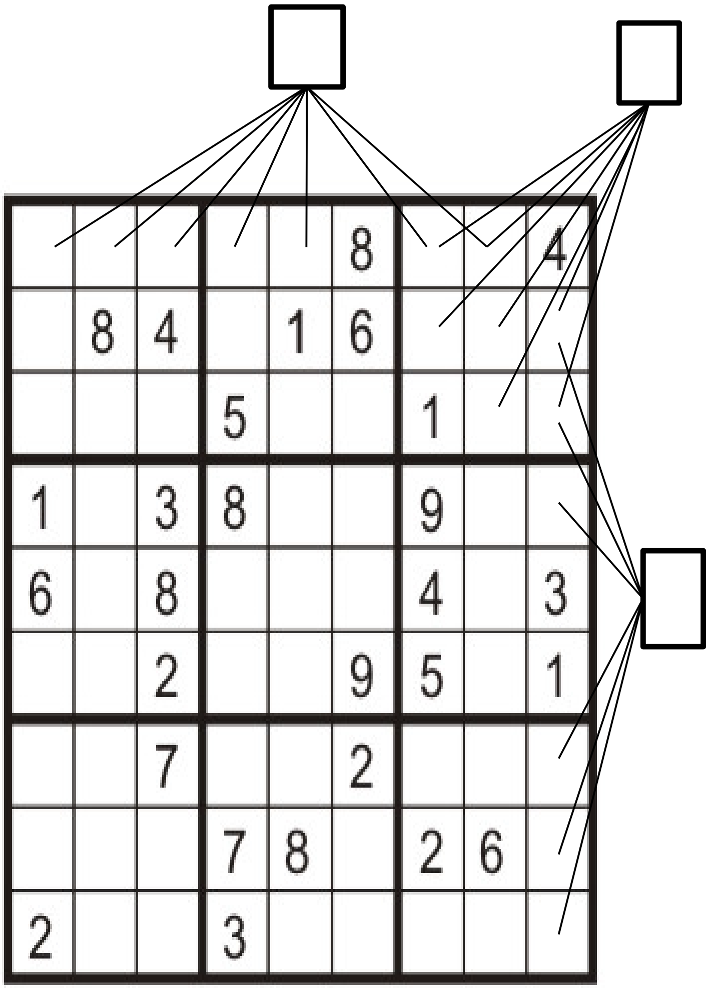


Example: Sudoku

					8			4
	8	4		1	6			
			5			1		
1		3	8			9		
6		8				4		3
		2			9	5		1
		7			2			
			7	8		2	6	
2			3					

- Variables:
- Domains:
- Constraints:

Example: Sudoku



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

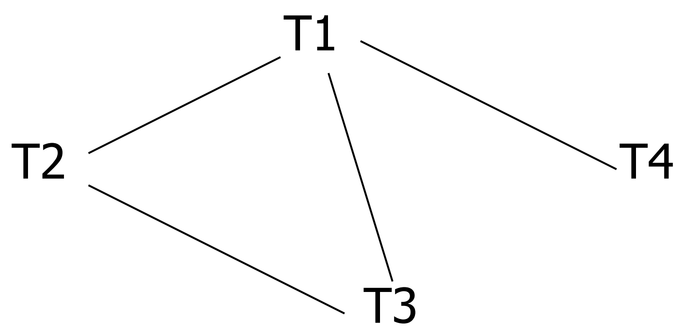
9-way alldiff for each column

9-way alldiff for each row

9-way alldiff for each region

(or can have a
bunch of pairwise
inequality
constraints)

Example: Task Scheduling



T1 must be done during T3
T2 must be achieved before T1 starts
T2 must overlap with T3
T4 must start after T1 is complete

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.:

$$SA \neq WA$$

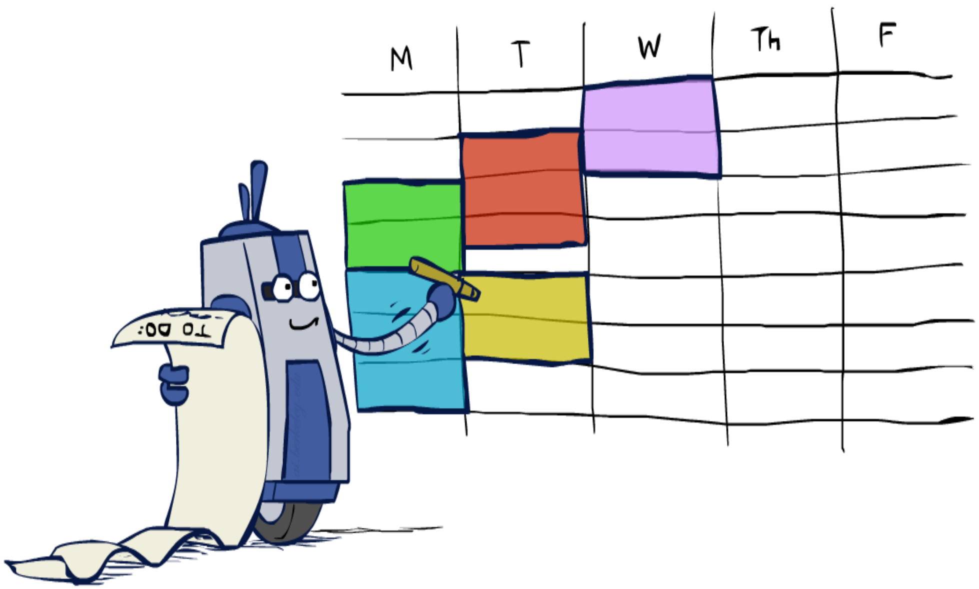
  - Higher-order constraints involve 3 or more variables:

    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

Real-World CSPs

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

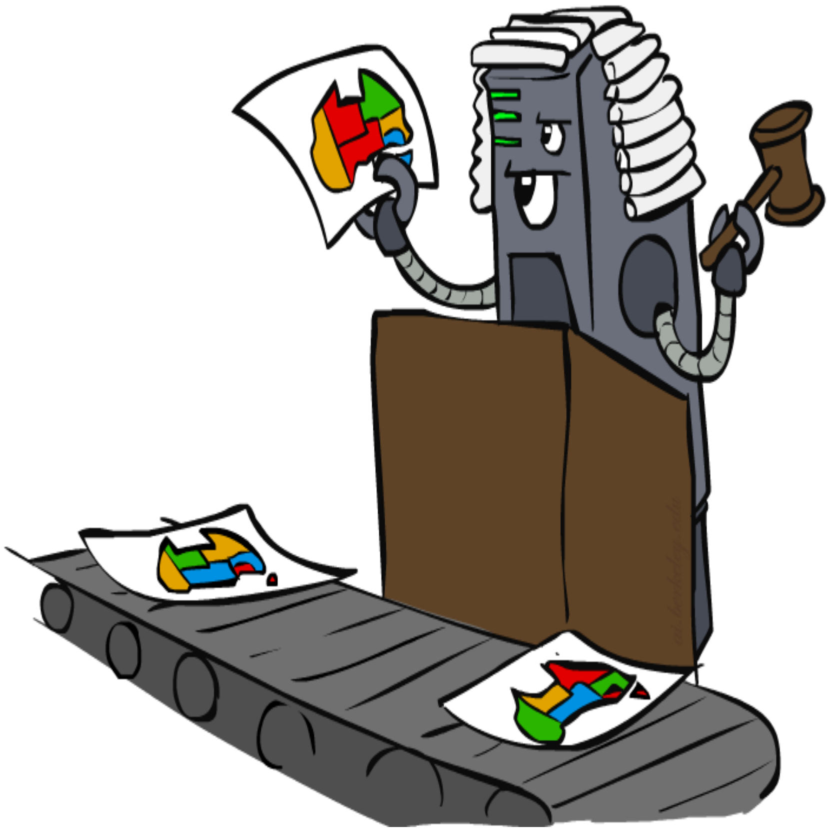


Standard Search Formulation

- Standard search formulation of CSPs

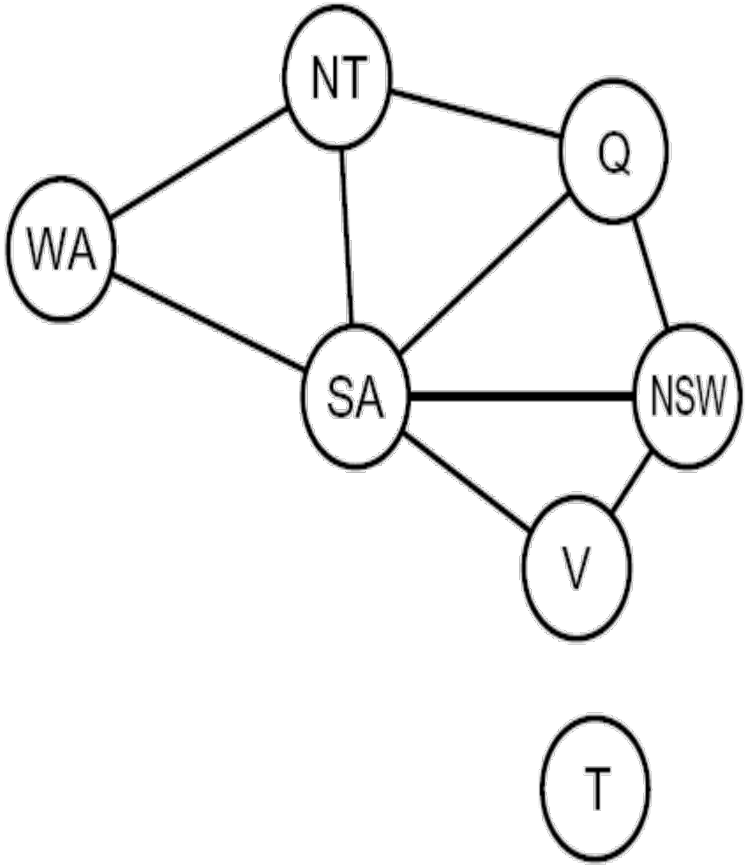
- States defined by the values assigned so far (partial assignments)
  - Initial state: the empty assignment, {}
  - Successor function: assign a value to an unassigned variable
  - Goal test: the current assignment is complete and satisfies all constraints

- We'll start with the straightforward, naïve approach, then improve it



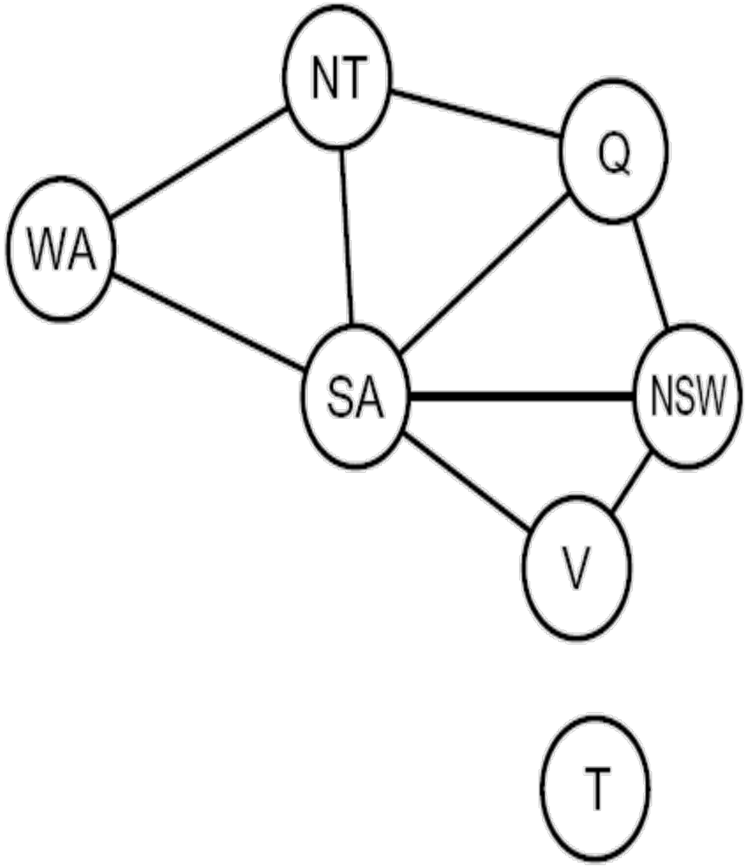
Search Methods

- What would BFS do?



Search Methods

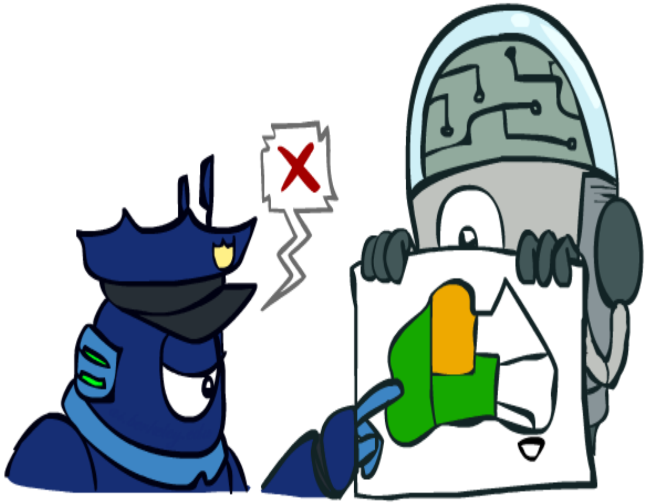
- What would DFS do?



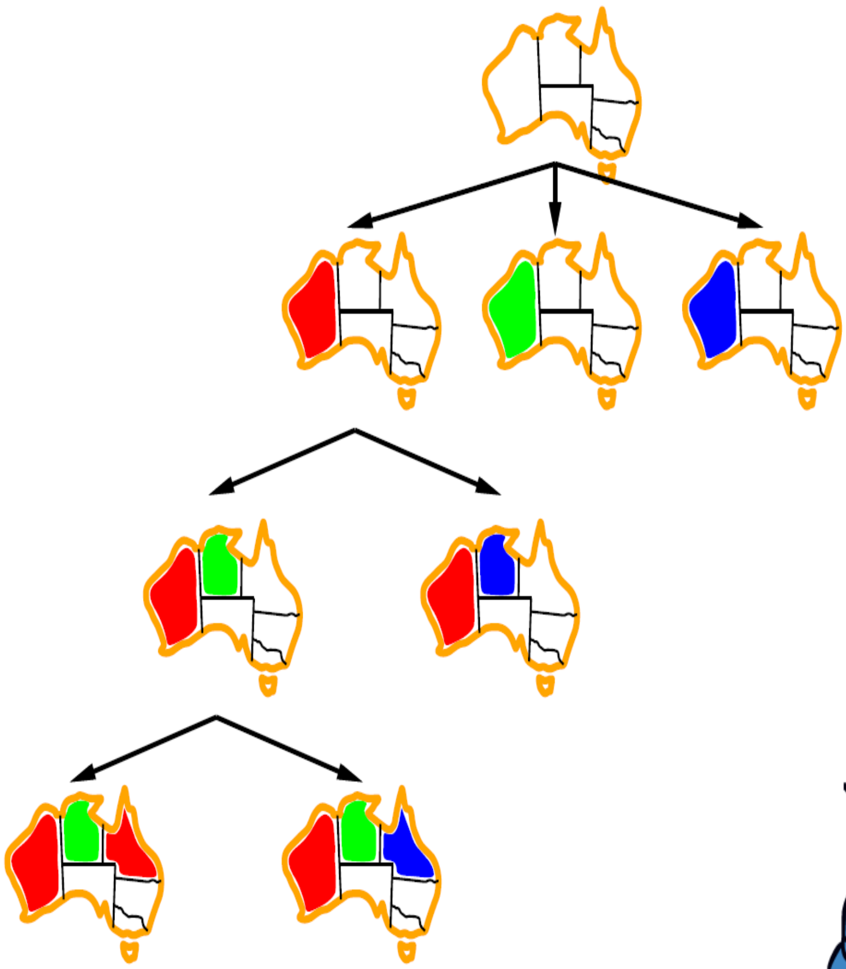
- What problems does naïve search have?

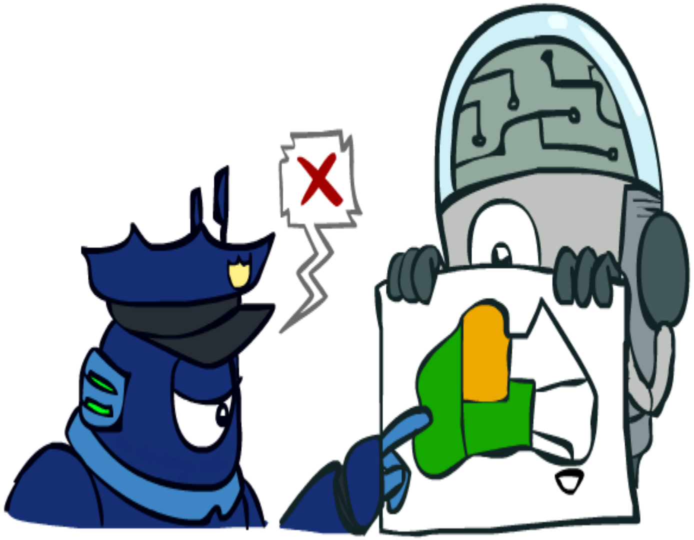
Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
  - Variable assignments are commutative, so fix ordering
  - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
  - Only need to consider assignments to a single variable at each step
- Idea 2: Check constraints as you go
  - I.e. consider only values which do not conflict previous assignments
  - Might have to do some computation to check the constraints
  - "Incremental goal test"
- Depth-first search with these two improvements is called *backtracking search*
- Can solve n-queens for n $\approx$ 25

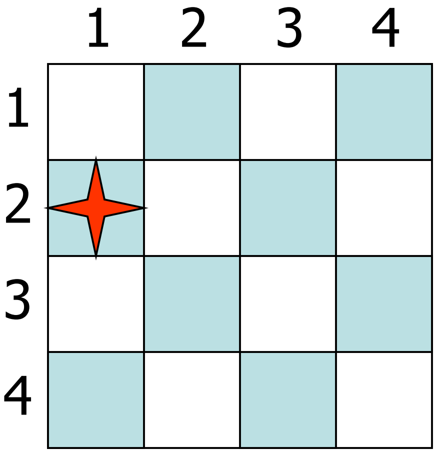


Backtracking Example





Backtracking



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
```

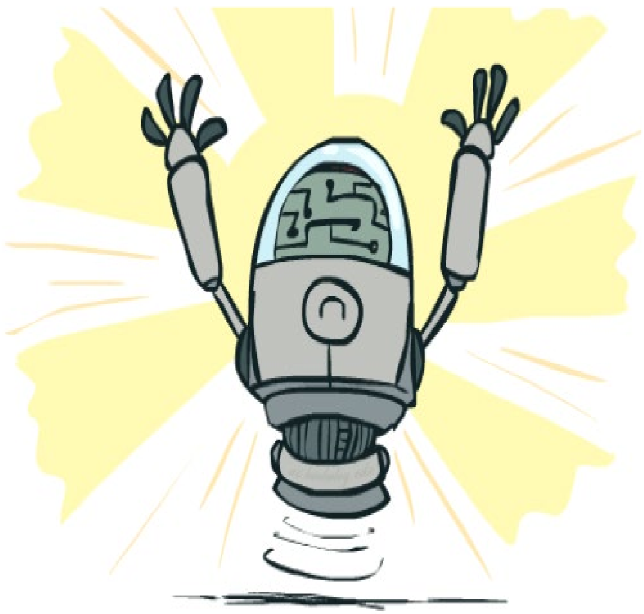
- Backtracking = DFS + variable-ordering + fail-on-violation
- What are the choice points?

Improving Backtracking

- General-purpose ideas give huge gains in speed

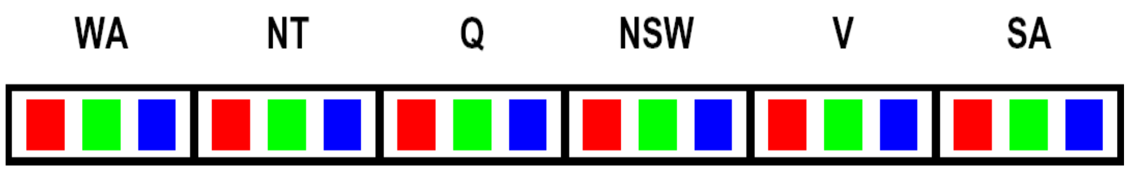
- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?

- Filtering: Can we detect inevitable failure early?



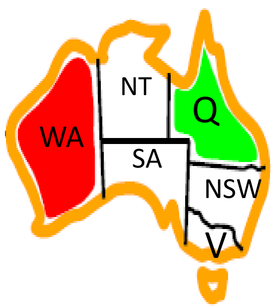
Filtering: Forward Checking

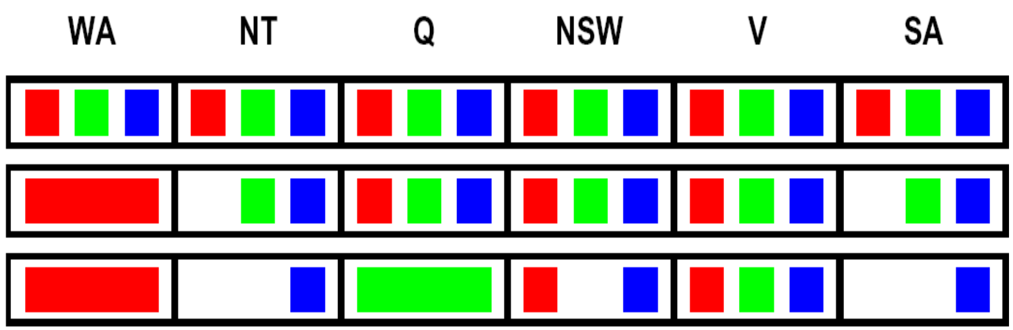
- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



Filtering: Constraint Propagation

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

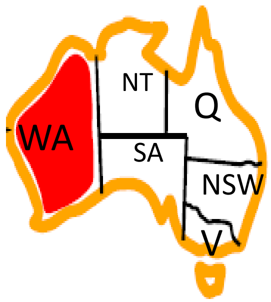


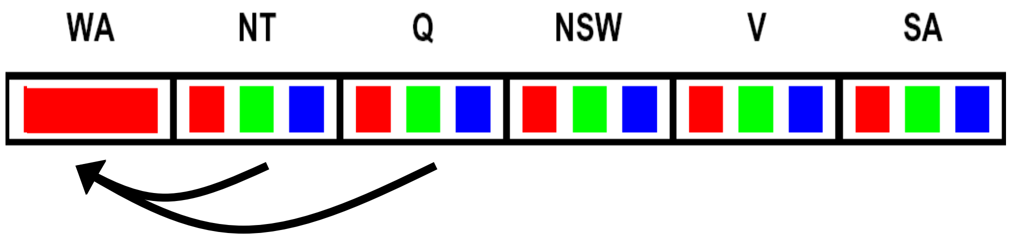


- NT and SA cannot both be blue!
- Why didn't we detect this yet?
- *Constraint propagation:* reason from constraint to constraint

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



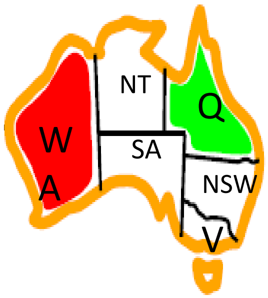


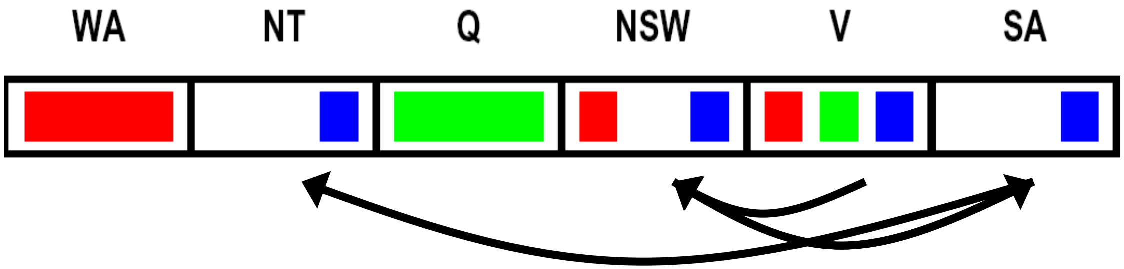
Delete from the tail!

- 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

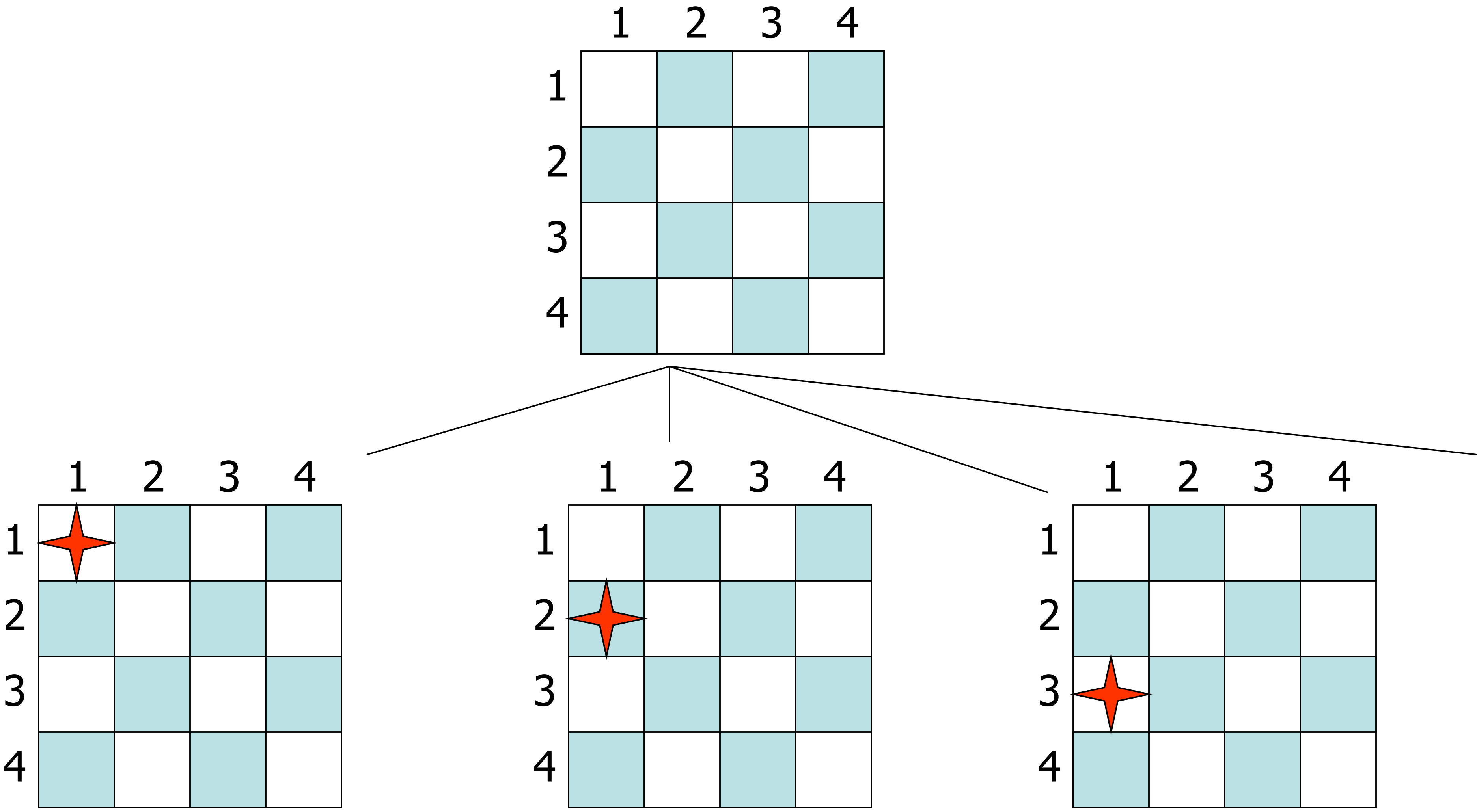




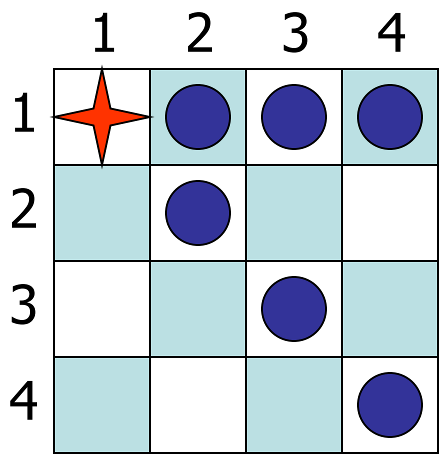
- Important: If X loses a value, neighbors of X need to be rechecked!
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment

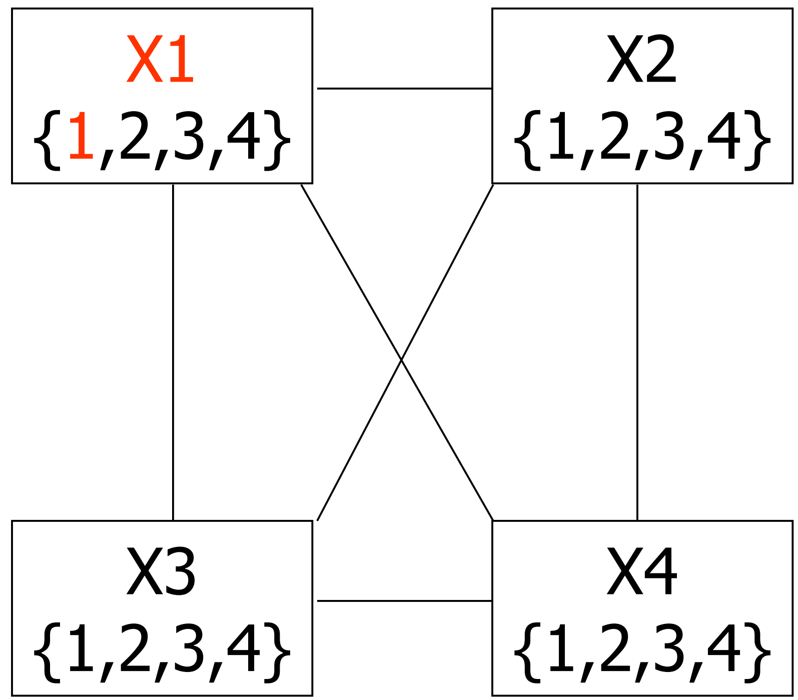
Remember: Delete from the tail!

4-Queens Problem

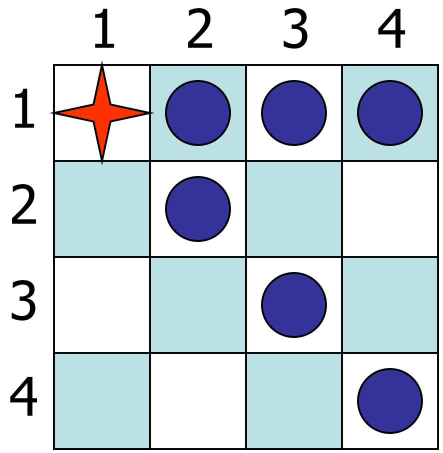


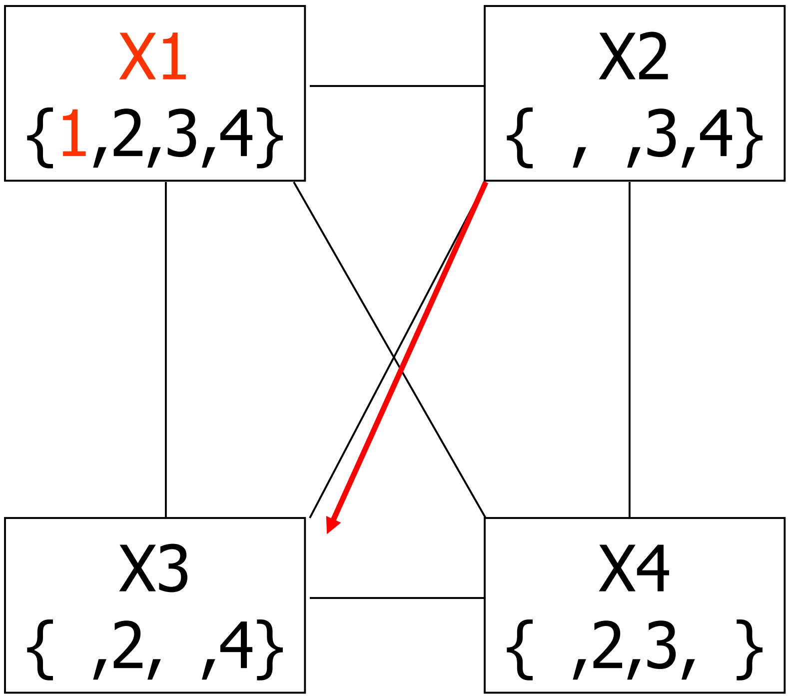
4-Queens Problem



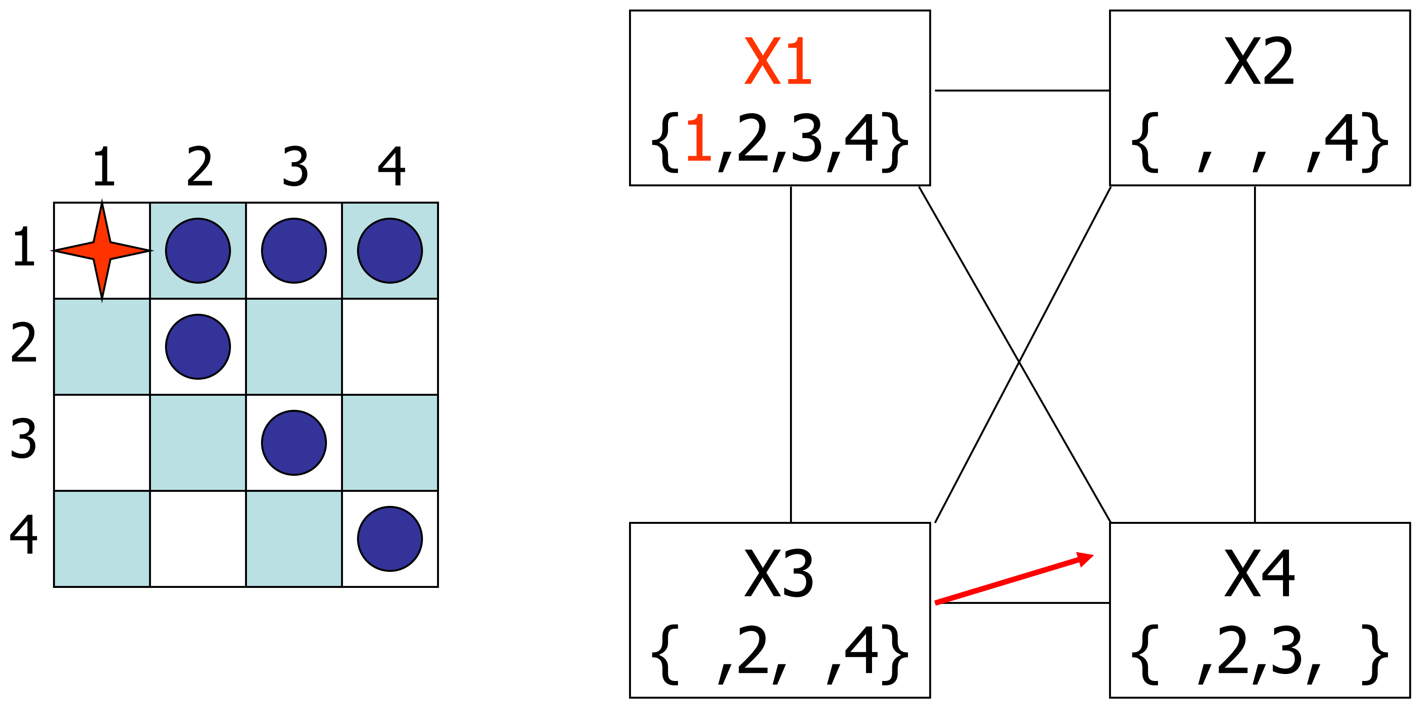


4-Queens Problem

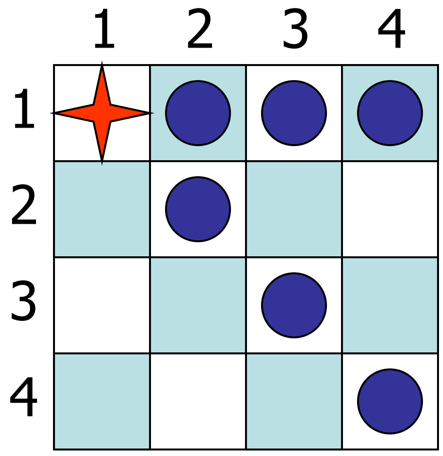


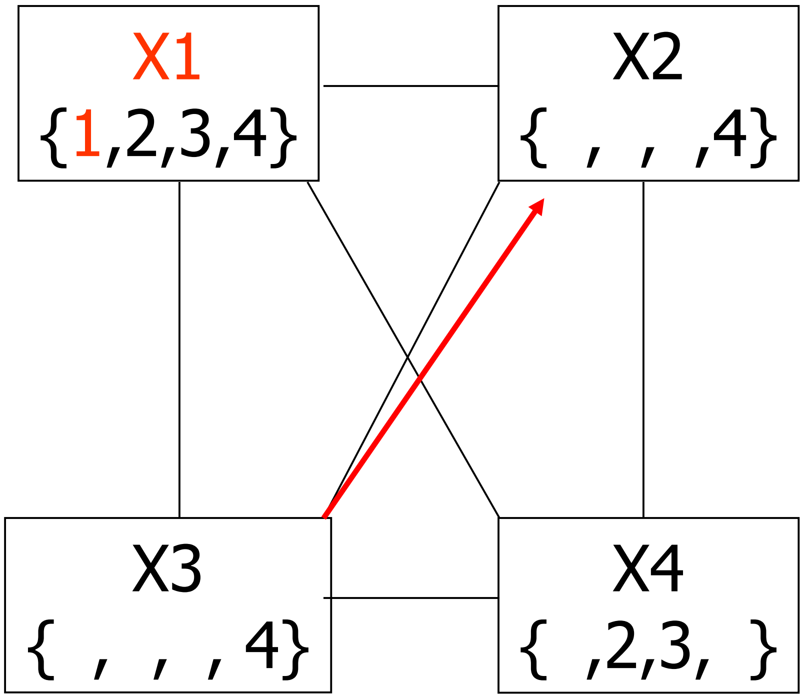


4-Queens Problem

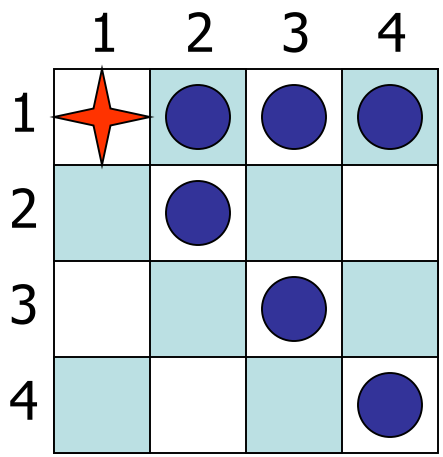


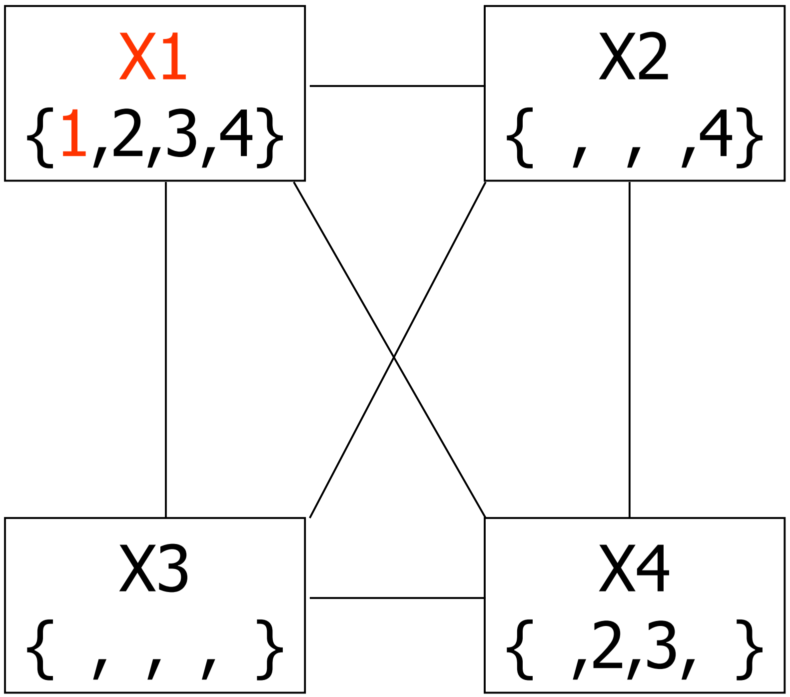
4-Queens Problem



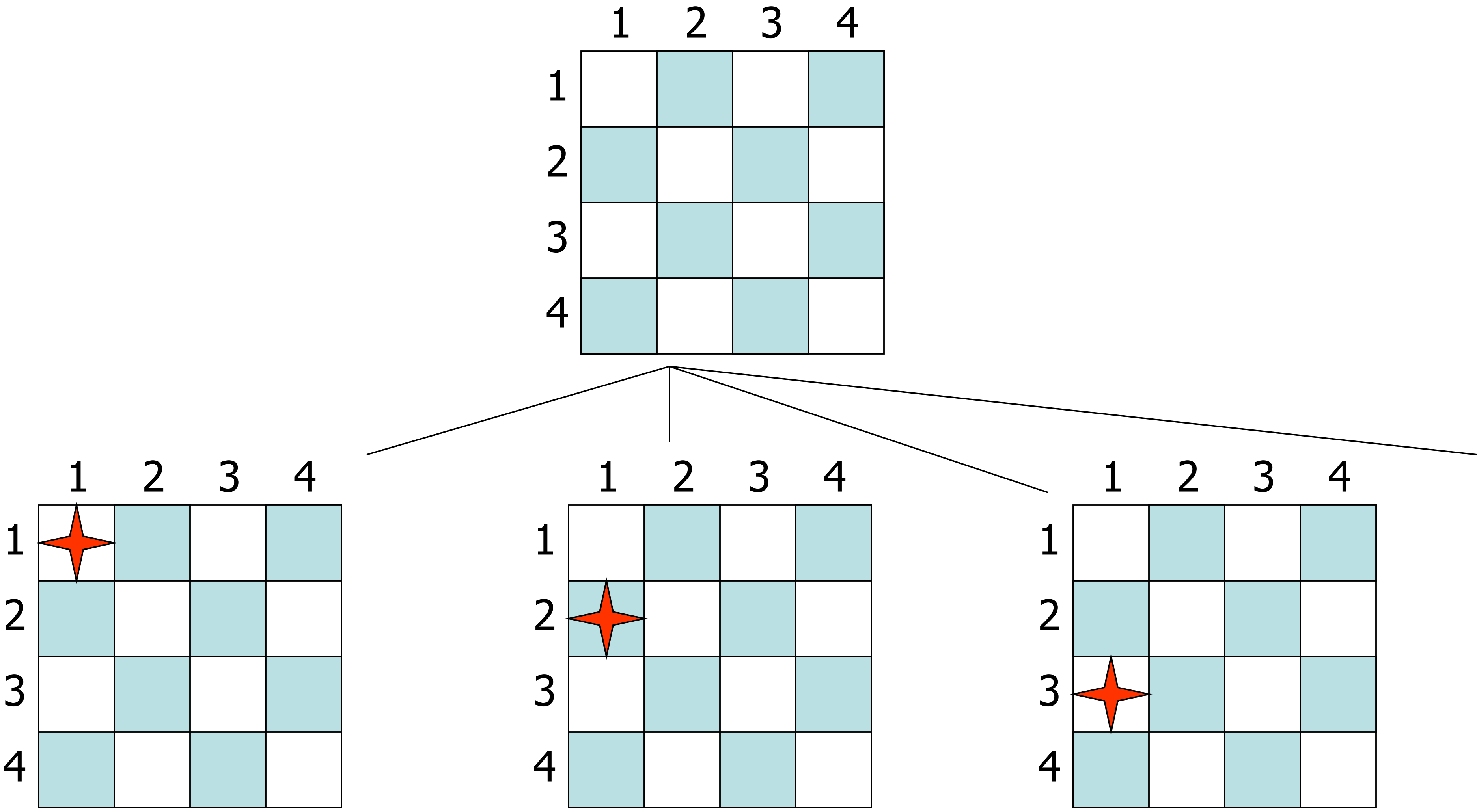


4-Queens Problem

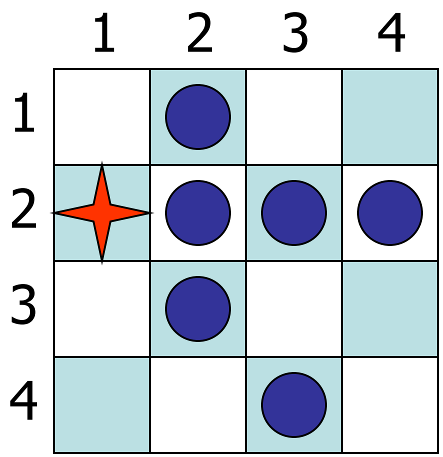


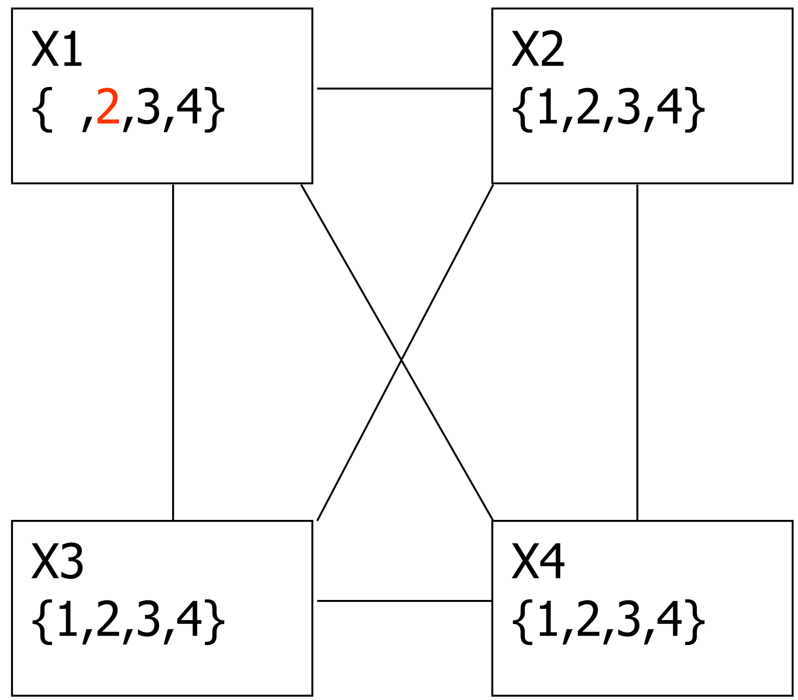


4-Queens Problem

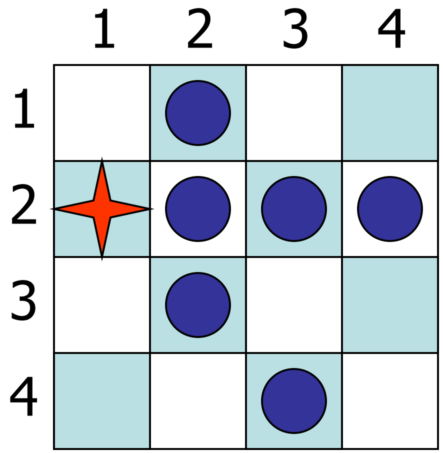


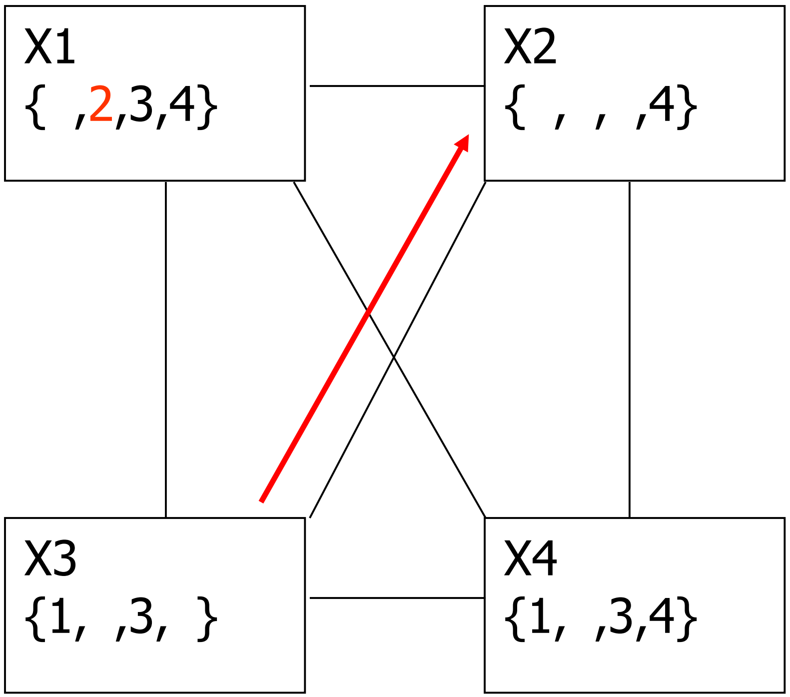
4-Queens Problem



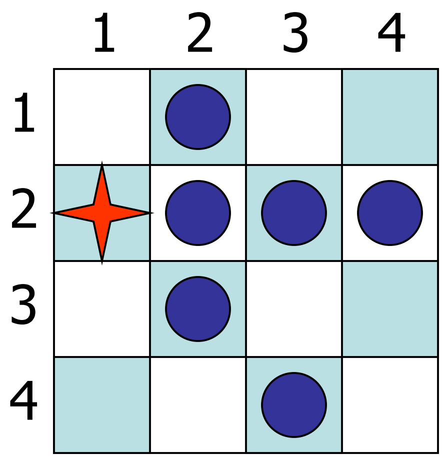


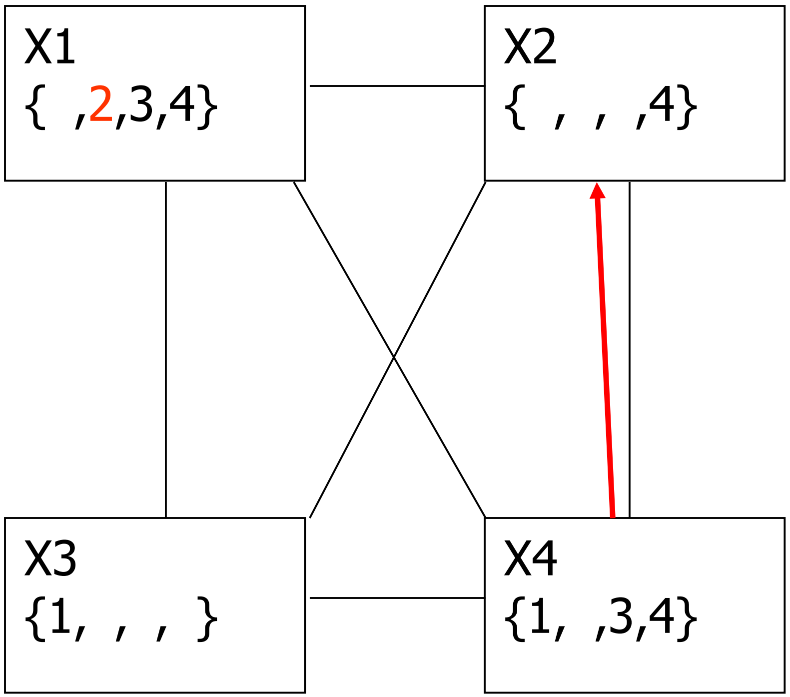
4-Queens Problem



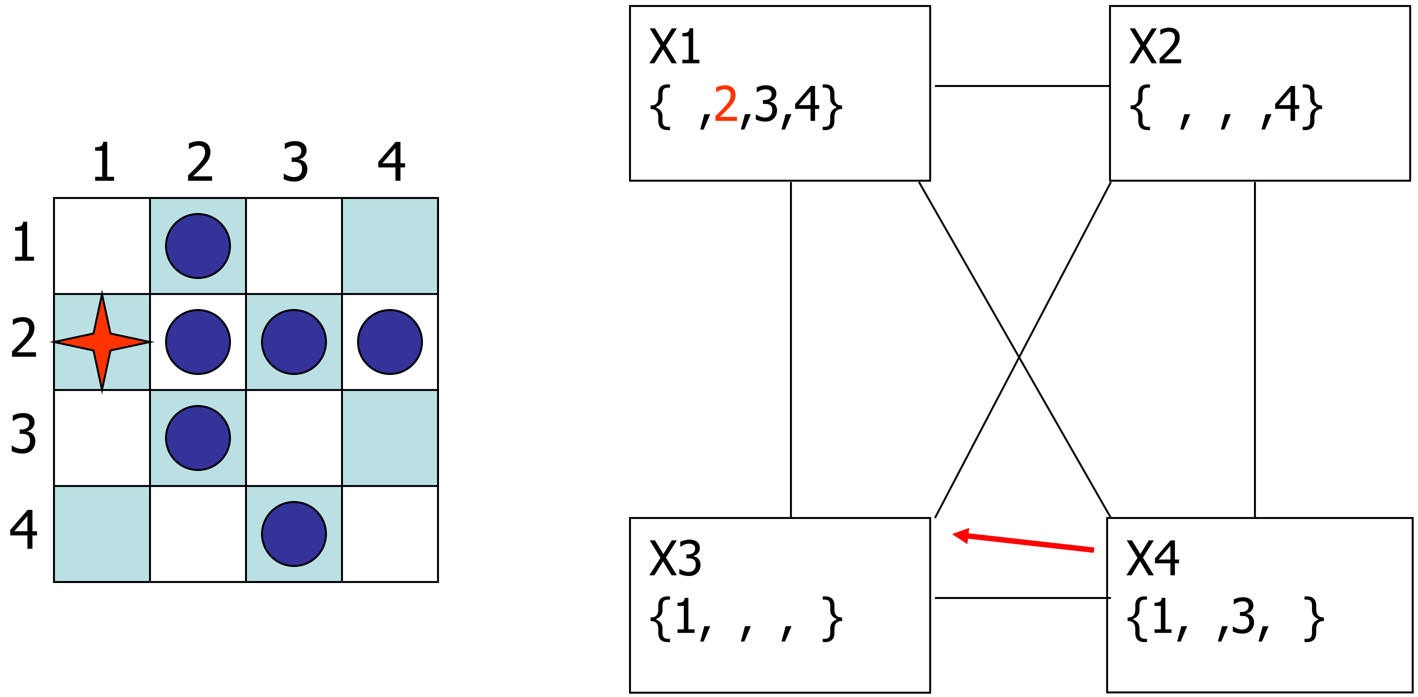


4-Queens Problem

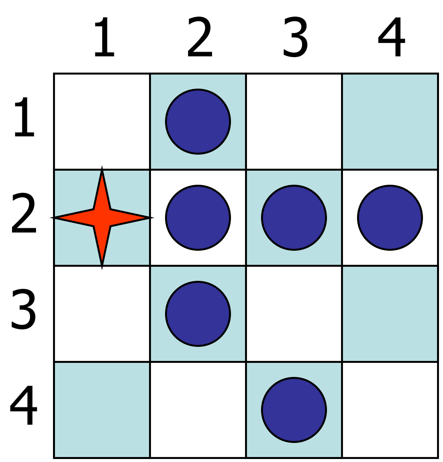


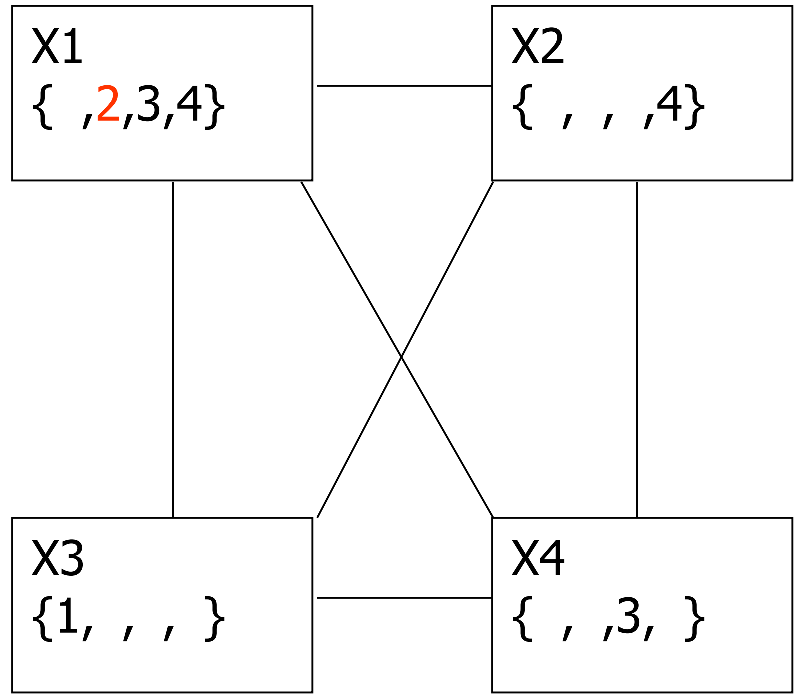


4-Queens Problem

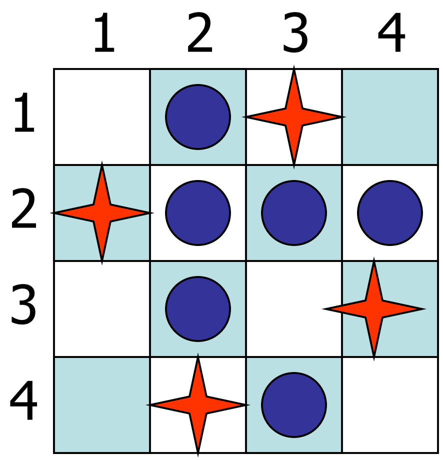


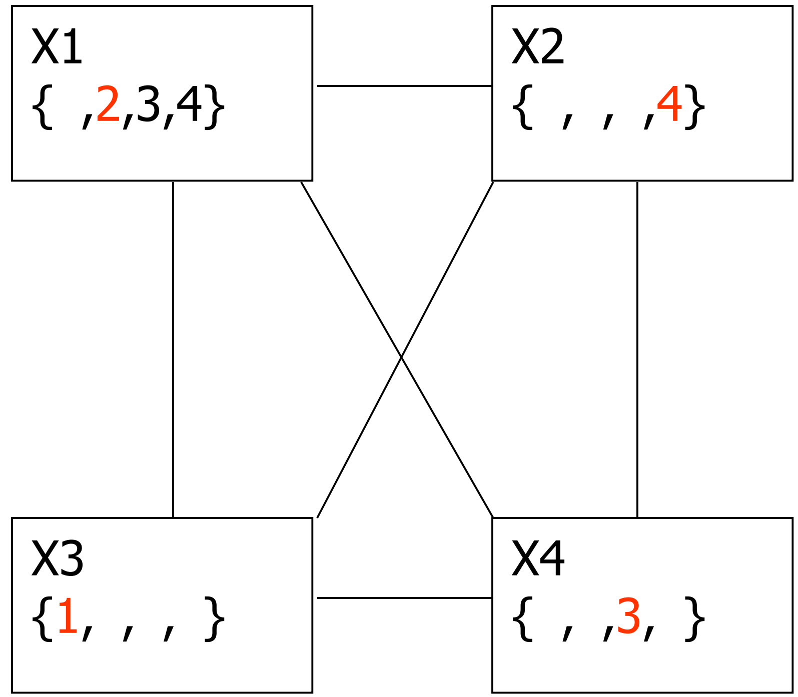
4-Queens Problem





4-Queens Problem

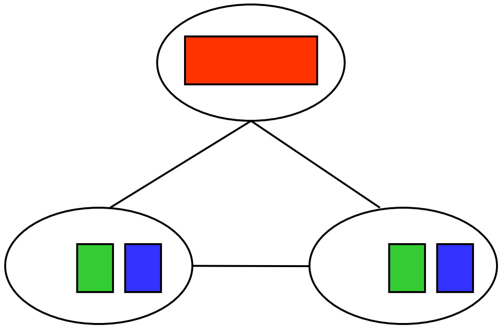


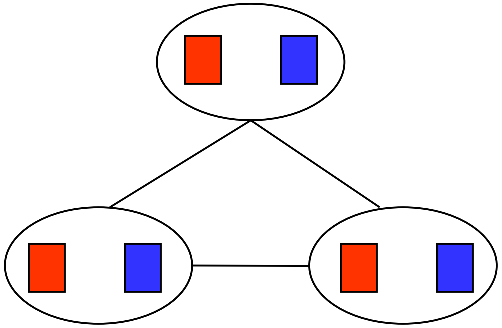


Limitations of Arc Consistency

- 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?

Backtracking Search

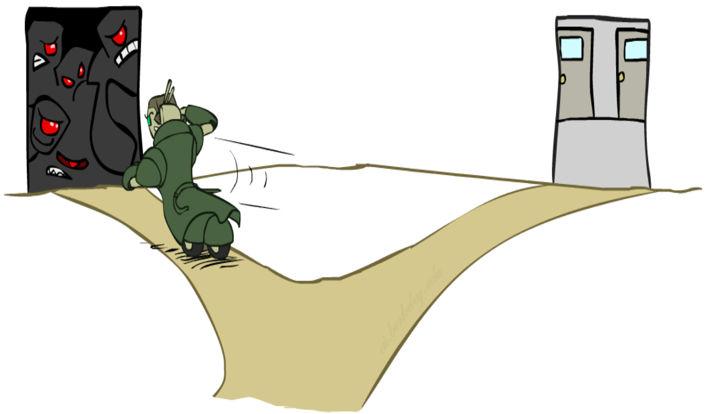
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)

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?

Ordering: Minimum Remaining Values

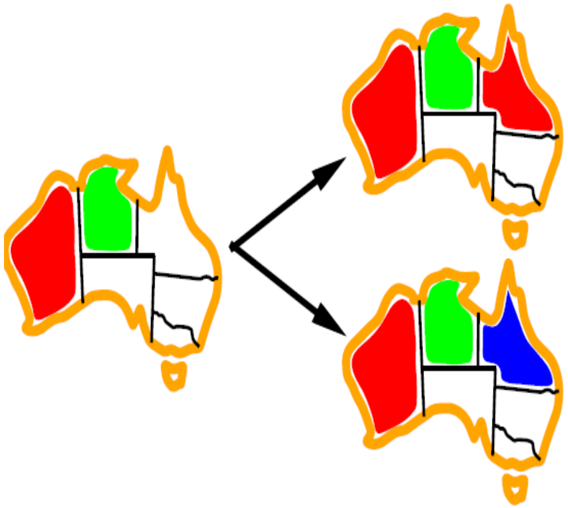
- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain

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

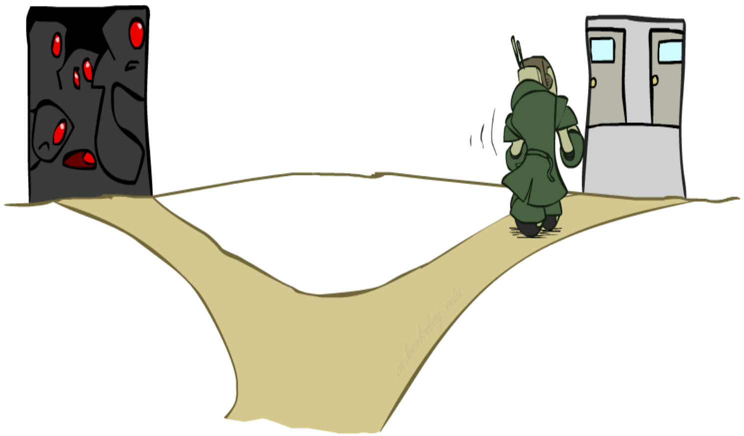


Ordering: Least Constraining Value

- 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)



- Combining these ordering ideas makes
  1000 queens feasible

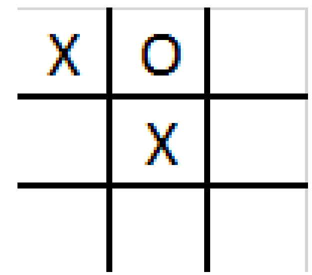


Summary

- CSP's
  - Uninformed search
  - Backtracking
  - Forward checking
  - Heuristic for expansion order
    - MRV
    - LCV

Lab 4 Due May 6, 7pm

- Create a TIC-TAC-TOE solver capable of predicting the result of a specific game when a board is provided.
- Complete 2 Functions
  - minmax_tictactoe(board, turn)
  - abprun_tictactoe(board, turn)



board=[1,2,0,0,1,0,0,0,0]

- Must use game_status(board) to check board
  - Use once per node in search tree

Lab4

- AB pruning must use exact algorithm from day 4 slides
  - Don’t use any additional information about game to improve it
- Helpful hint:
  - AB pruning code can be turned into mini-max with a very simple modification.
  - Must work for any board state, even those not reachable in standard play
- Program in python3
- Don’t use any additional modules other than the included common

Lab 2 questions?