Constraint Satisfaction Problems

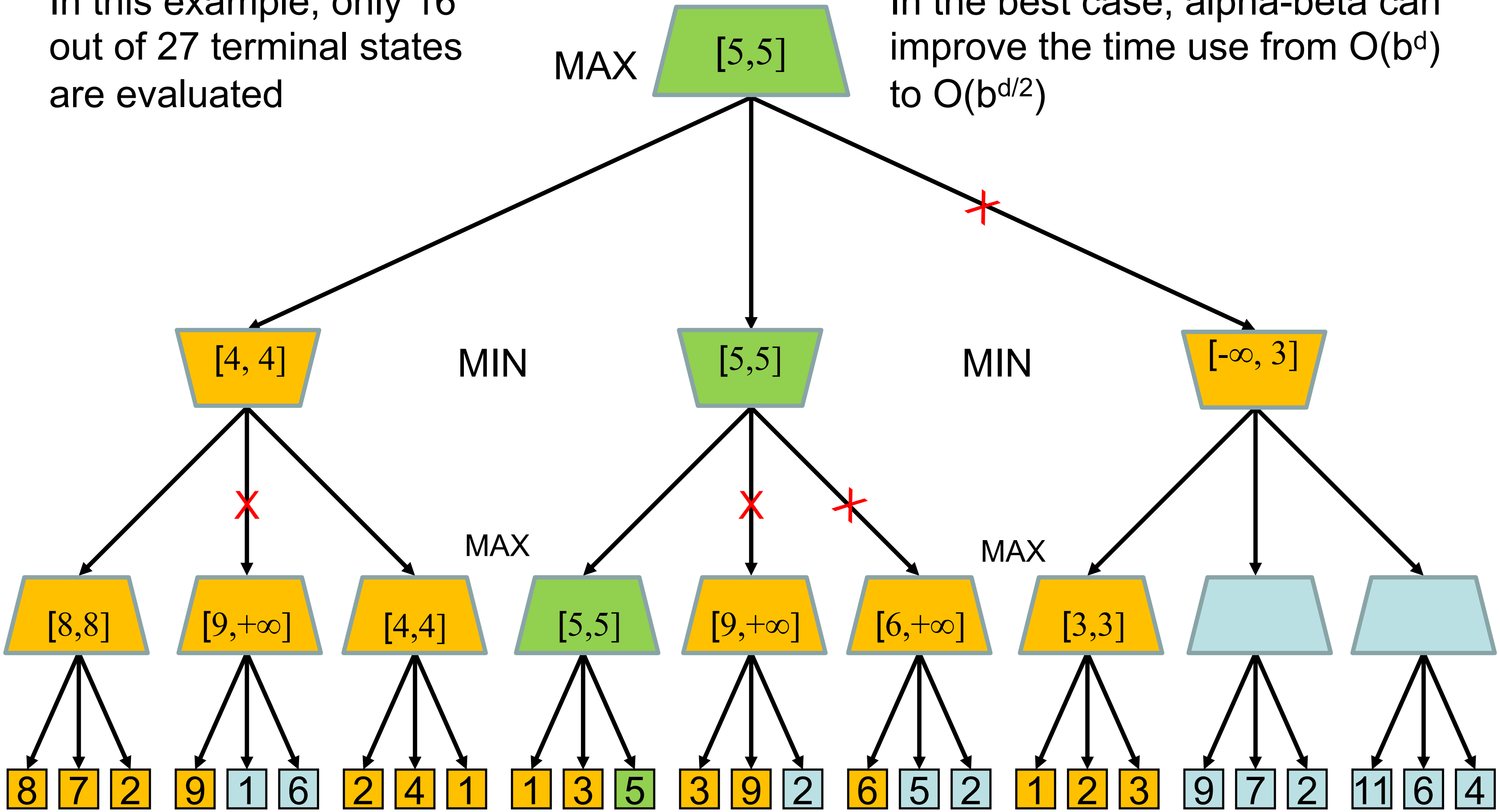
CPSC 470 – Artificial Intelligence

Brian Scassellati

Alpha-Beta Pruning Example

In this example, only 16 out of 27 terminal states are evaluated

In the best case, alpha-beta can improve the time use from $O(b^d)$ to $O(b^{d/2})$



Sudoku

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

Example puzzle with a unique solution

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

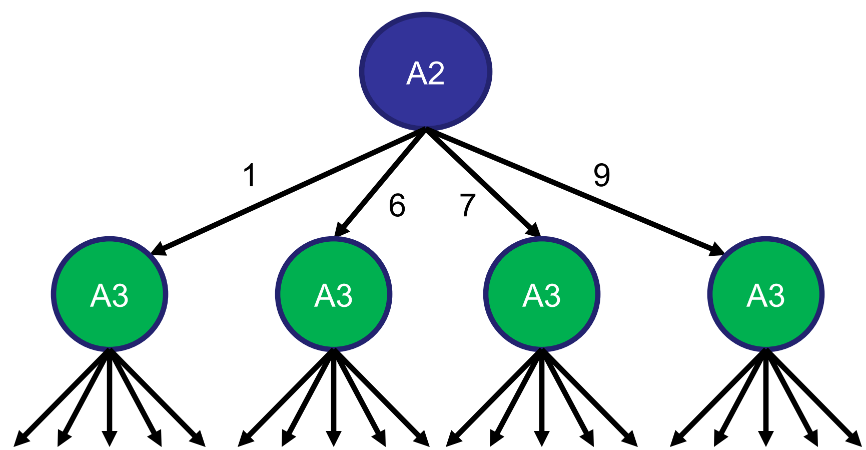
No duplicates in row, column, or 3x3 box

Solving Sudoku via Search

4	A2	A3				8		5
	3							
			7					
	2						6	
				8		4		
	4			1				
			6		3		7	
5		3	2		1			
1		4						

- 61 depth, max 8 branching factor
- 4.6 x 10^{38} possibilities
- Even on 1 million 10GHz, 1024 core machines, this is 1300 billion years!

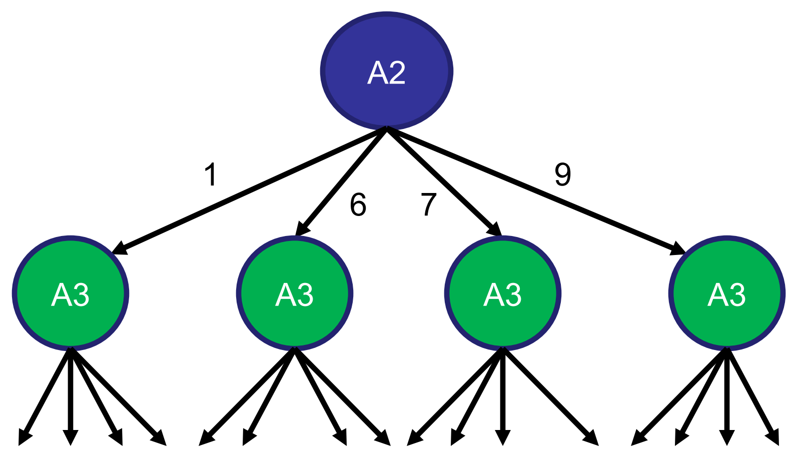
- 20 squares fixed and 61 need to be solved
- Find possible entries
  - A2: 1 ~~2~~ ~~3~~ ~~4~~ ~~5~~ 6 7 ~~8~~ 9
  - A3: 1 2 ~~3~~ ~~4~~ ~~5~~ 6 7 ~~8~~ 9
- Build a tree:



A Smarter Way

4	A2	A3				8		5
	3							
			7					
	2						6	
				8		4		
	4			1				
			6		3		7	
5		3	2		1			
1		4						

- Find possible entries
  - A2: 1 ~~2~~ ~~3~~ ~~4~~ ~~5~~ 6 7 ~~8~~ 9
  - A3: 1 2 ~~3~~ ~~4~~ ~~5~~ 6 7 ~~8~~ 9
- Once we choose A2, that further limits our choices



Constraint Satisfaction Problems

- In a typical search problem
  - state is a "black box" – any data structure that supports successor function, heuristic function, and goal test
- In a constraint satisfaction problem (CSP):
  - state is an assignment of values from a domain D_i to a set of variables X_i
  - goal test is a set of constraints specifying allowable combinations of values for subsets of variables
- A solution to a CSP is one that is complete (all variables are assigned) and consistent (no constraints are violated)
- Simple example of a formal representation language

Sudoku as a CSP

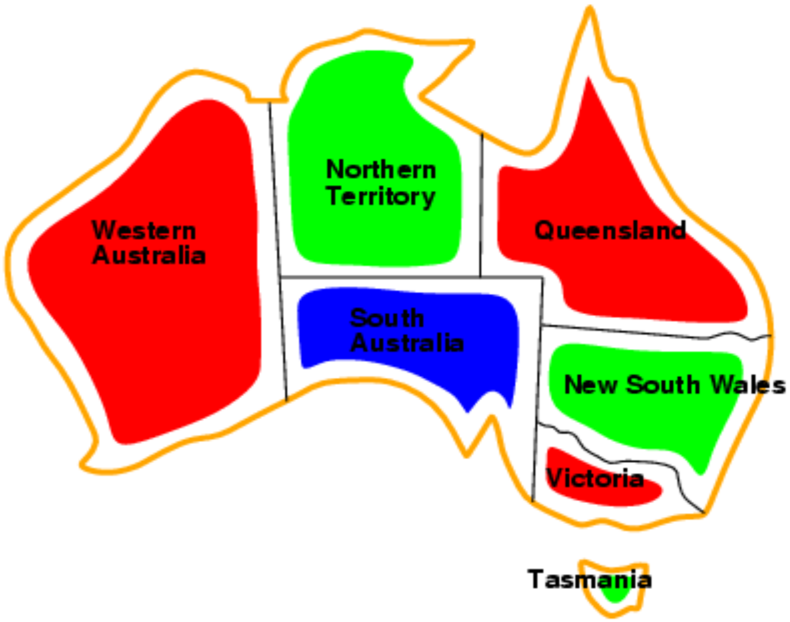
4						8		5
	3							
			7					
	2						6	
				8		4		
	4			1				
			6		3		7	
5		3	2		1			
1		4						

- Domain = {1, 2, 3, 4, 5, 6, 7, 8, 9}
- Variables = { A1, A2, ... A9,
  B1, B2, … B9,
  …
  I1, I2, … I9 }
- Constraints from row, column, and 3x3 cell restrictions
- Constraints =
  {A1≠A2, A1≠A3, A1≠A4, …
  A1≠B1, A1≠C1, A1≠D1, …
  A1≠B2, A1≠B3, A1≠C1, …}

Simpler Example: Map Coloring

- Variables $V_i = \{WA, NT, Q, NSW, V, SA, T\}$
- Domain D_i = {red, green, blue}
- Constraints: adjacent regions must have different colors
  - e.g., $WA \neq NT$, or (WA,NT) in {(red,green), (red,blue), (green,red), (green,blue), (blue,red), (blue,green)}

Simpler Example: Map Coloring



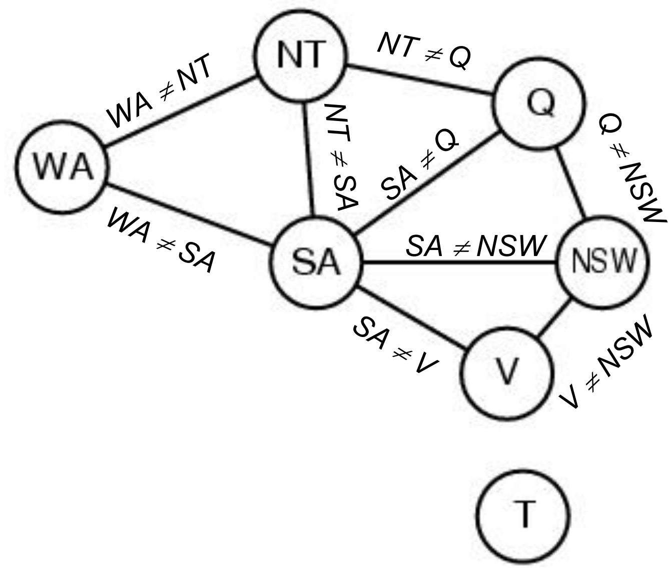
- Solutions are complete and consistent assignments
- One solution is shown above

  WA = red, NT = green, Q = red, NSW = green,

  V = red, SA = blue, T = green

Constraint Graph

- Constraint graph:
  - nodes are variables
  - arcs are constraints
- CSP benefits
  - Standard representation pattern: variables with values
  - Generic goal, successor functions
  - Generic heuristics (no domain specific expertise)
  - Graph can simplify search.
    - e.g. Tasmania is an independent subproblem.

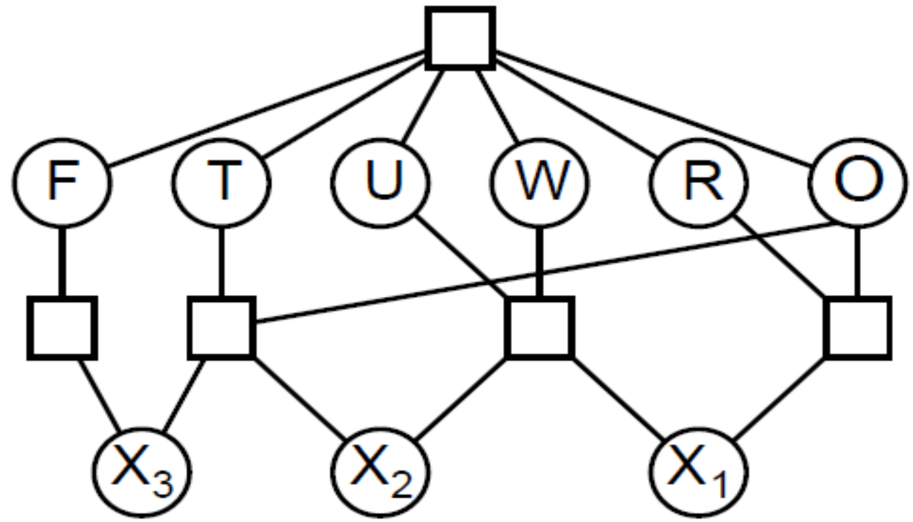


Another Example: Cryptarithmetic

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

Another Example: Cryptarithmetic

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



Variables: $F, O, U, R, T, W, X_1, X_2, X_3$

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

Constraints: $Alldiff\,(F, O, U, R, T, W)$

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

$$X_1 + W + W = U + 10 \cdot X_2$$

$$X_2 + T + T = O + 10 \cdot X_3$$

$$X_3 = F,\ T \neq 0,\ F \neq 0$$

Varieties of CSPs

- Discrete variables
  - finite domains:
    - *n* variables, domain size *d* → $O(d^n)$ complete assignments
    - e.g., Boolean CSPs, incl.~Boolean satisfiability (NP-complete)
  - infinite domains:
    - integers, strings, etc.
    - e.g., job scheduling, variables are start/end days for each job
    - need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
- Continuous variables
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in polynomial time by linear programming

Varieties of constraints

- Unary constraints involve a single variable,
  - e.g., SA ≠ green
- Binary constraints involve pairs of variables,
  - e.g., SA ≠ WA
- Higher-order constraints involve 3 or more variables,
  - e.g., cryptarithmetic column constraints

Real-world CSPs

- Assignment problems
  - e.g., who teaches what class
- Timetabling problems
  - e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling
- Circuit layout

- Notice that many real-world problems involve real-valued variables

Solving CSPs

- Let's start with a straightforward approach, then fix it.
- Just like we did with Sudoku, let's treat this as a search problem.
  - Initial state: the empty assignment { }
  - Successor function: assign a value to an unassigned variable that does not conflict with current assignment

    → fail if no legal assignments
  - Goal test: the current assignment is complete

Backtracking search

- Variable assignments are commutative, i.e.,

  [WA = red] followed by [NT = green] is the same as
  [NT = green] followed by [WA = red]
- Only need to consider assignments to a single variable at each depth of the tree
- Depth-first search for CSPs with single-variable assignments is called backtracking search
- Backtracking search is the basic uninformed algorithm for CSPs
- Can solve n-queens for $n \approx 25$

Backtracking search

function RECURSIVE-BACKTRACKING is preceded by:

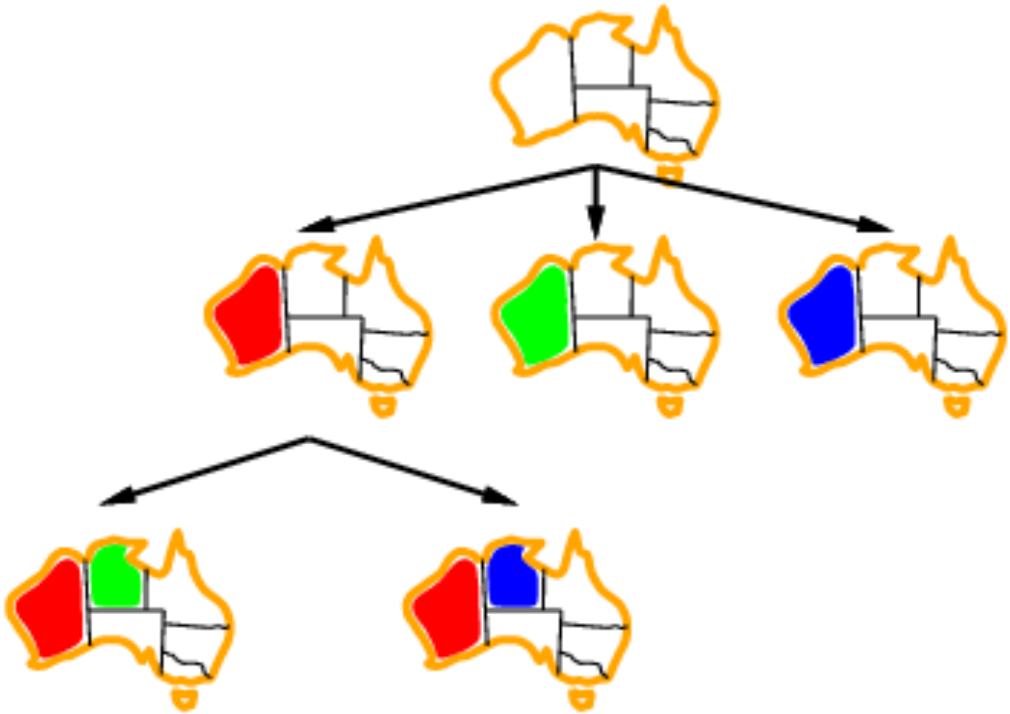
function BACKTRACKING-SEARCH(*csp*) **returns** a solution, or failure
  return RECURSIVE-BACKTRACKING({}, *csp*)

function RECURSIVE-BACKTRACKING(*assignment*, *csp*) **returns** a solution, or failure
  if *assignment* is complete **then return** *assignment*
  var ← SELECT-UNASSIGNED-VARIABLE(*Variables[csp]*, *assignment*, *csp*)
  for each *value* **in** ORDER-DOMAIN-VALUES(*var*, *assignment*, *csp*) **do**
    if *value* is consistent with *assignment* according to Constraints[*csp*] **then**
      add { *var* = *value* } to *assignment*
      result ← RECURSIVE-BACKTRACKING(*assignment*, *csp*)
      if *result* ≠ *failue* **then return** *result*
      remove { *var* = *value* } from *assignment*
  return *failure*

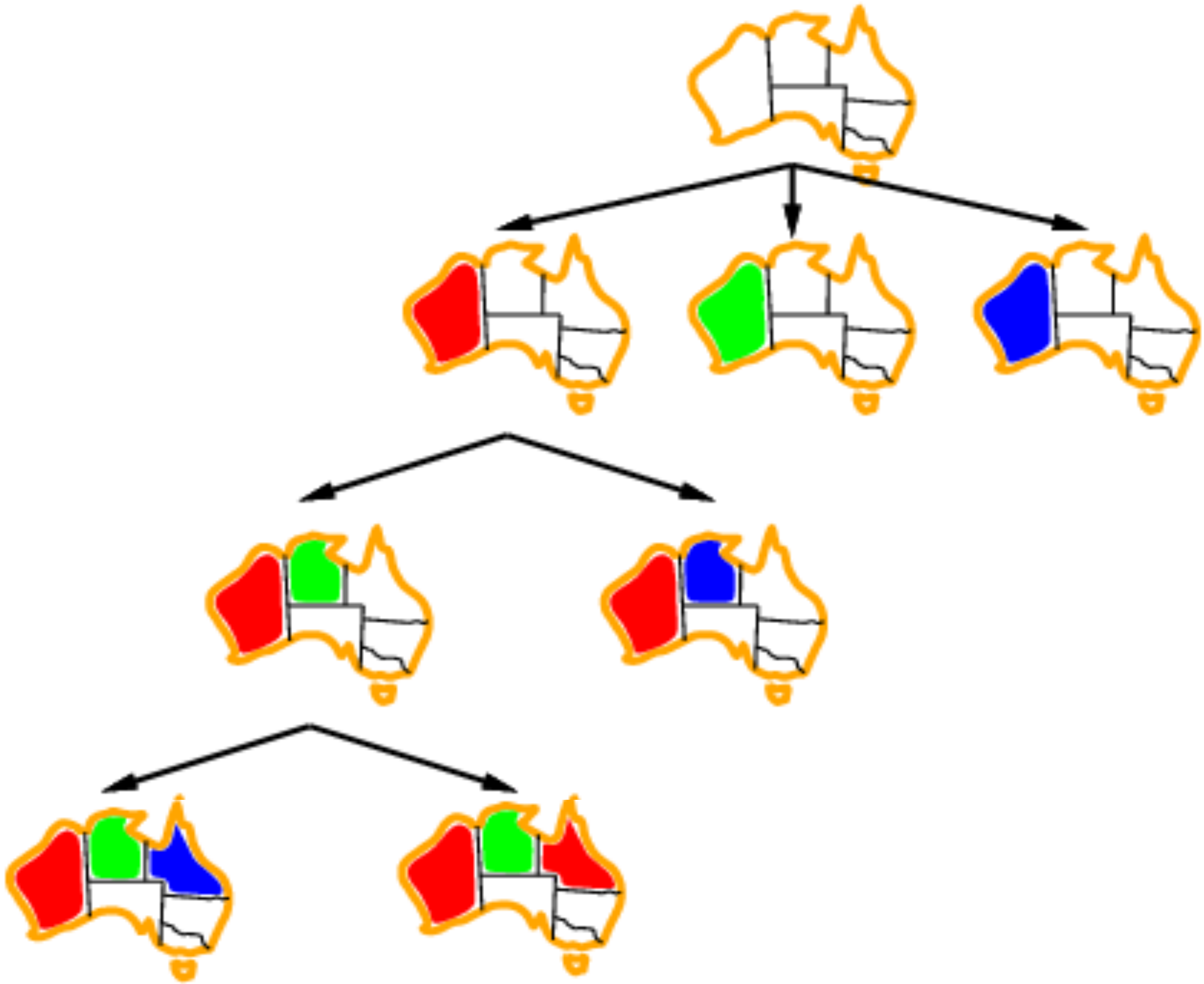
Backtracking example

Backtracking example

Backtracking example



Backtracking example



Improving backtracking efficiency

- General-purpose methods can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?
- Heuristics:
  1. Most constrained variable
  2. Most constraining variable
  3. Least constraining value
  4. Forward checking

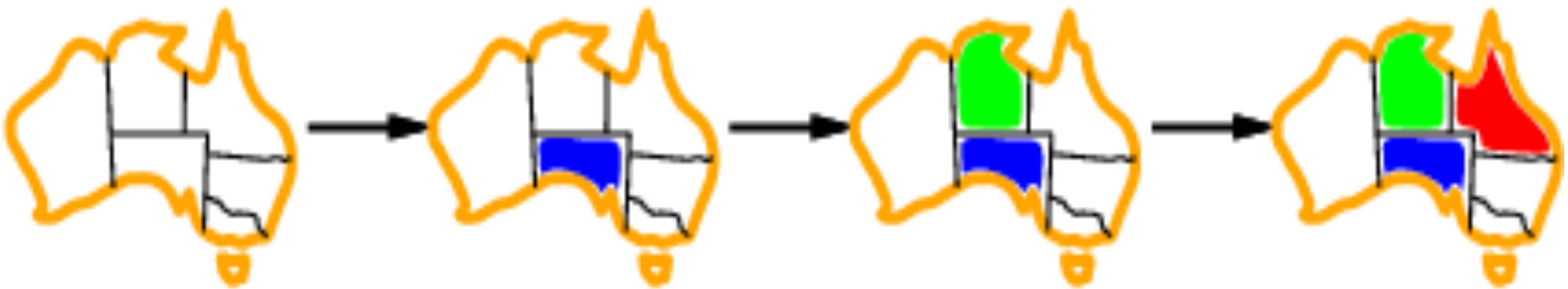
H1: Most constrained variable

- Most constrained variable:

  choose the variable with the fewest legal values

- a.k.a. minimum remaining values (MRV) heuristic

H2: Most constraining variable



- Tie-breaker among most constrained variables
- Most constraining variable:
  - choose the variable with the most constraints on remaining variables

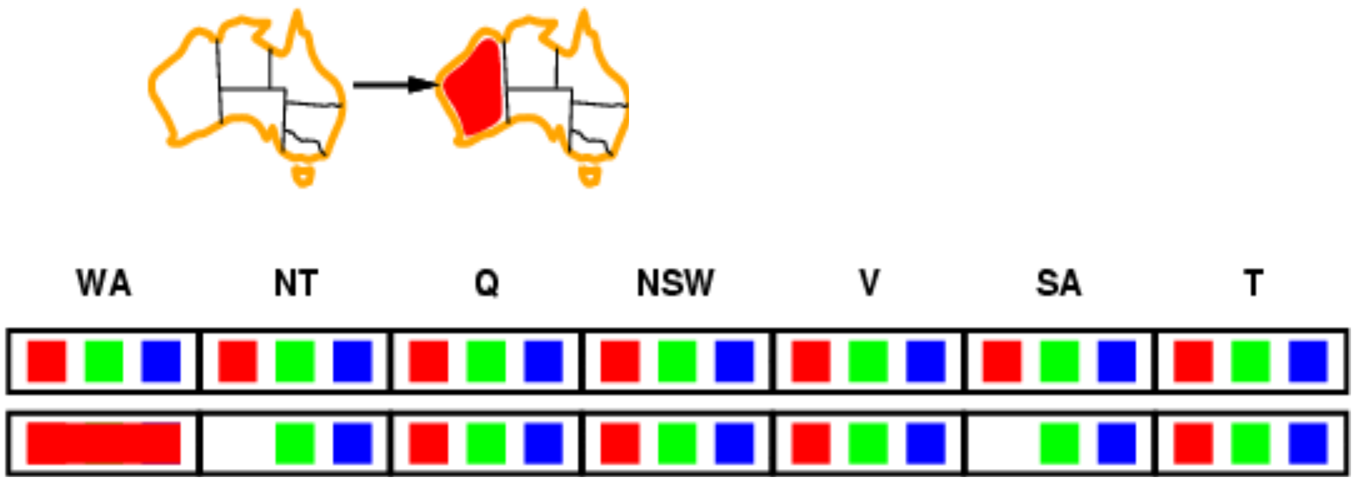
H3: Least constraining value

- Given a variable, choose the least constraining value:
  - the one that rules out the fewest values in the remaining variables
- Combining these heuristics makes 1000 queens feasible

H4: Forward checking

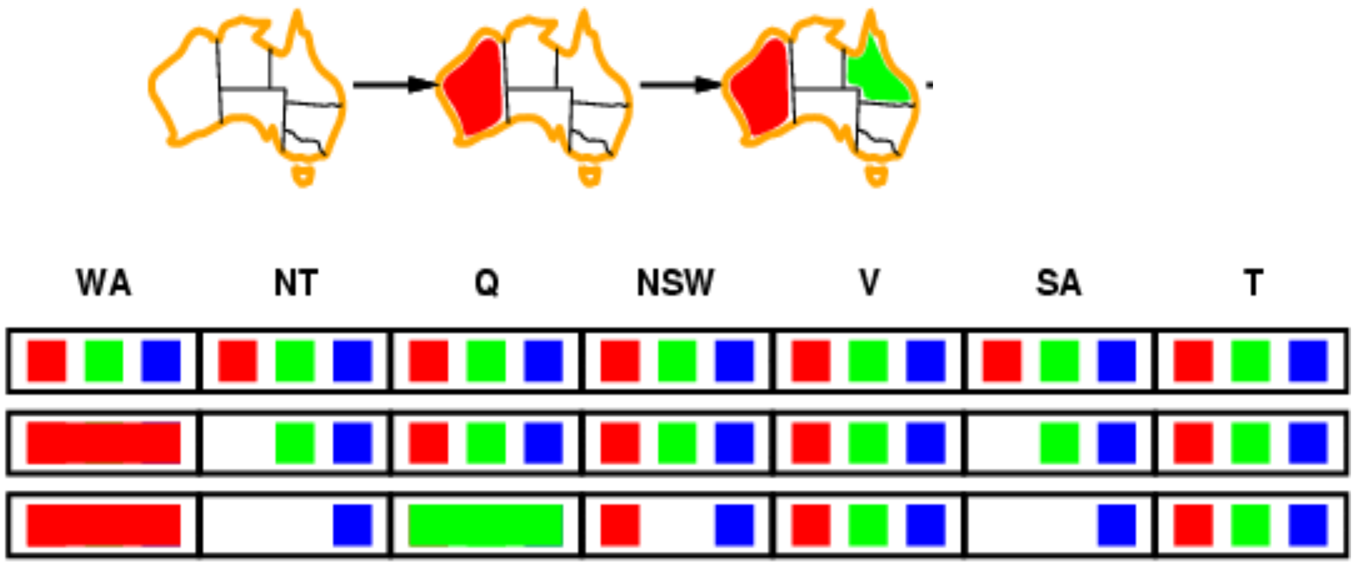
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values

H4: Forward checking



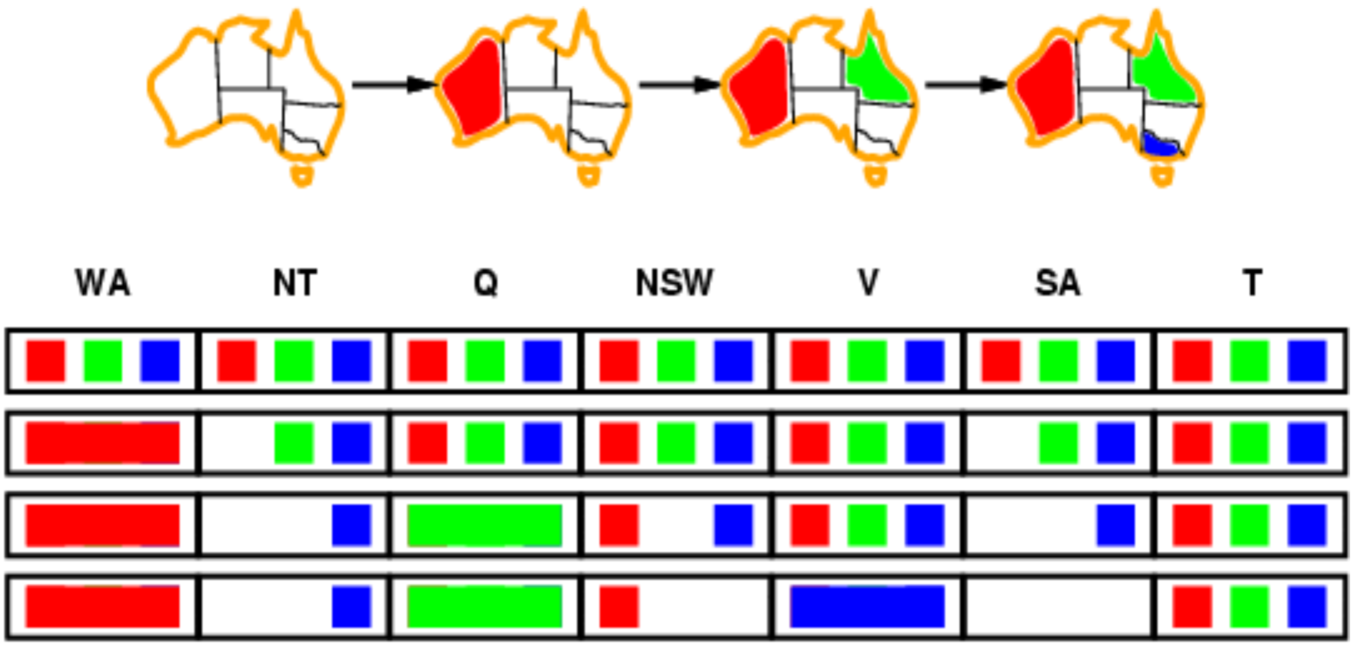
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values

H4: Forward checking



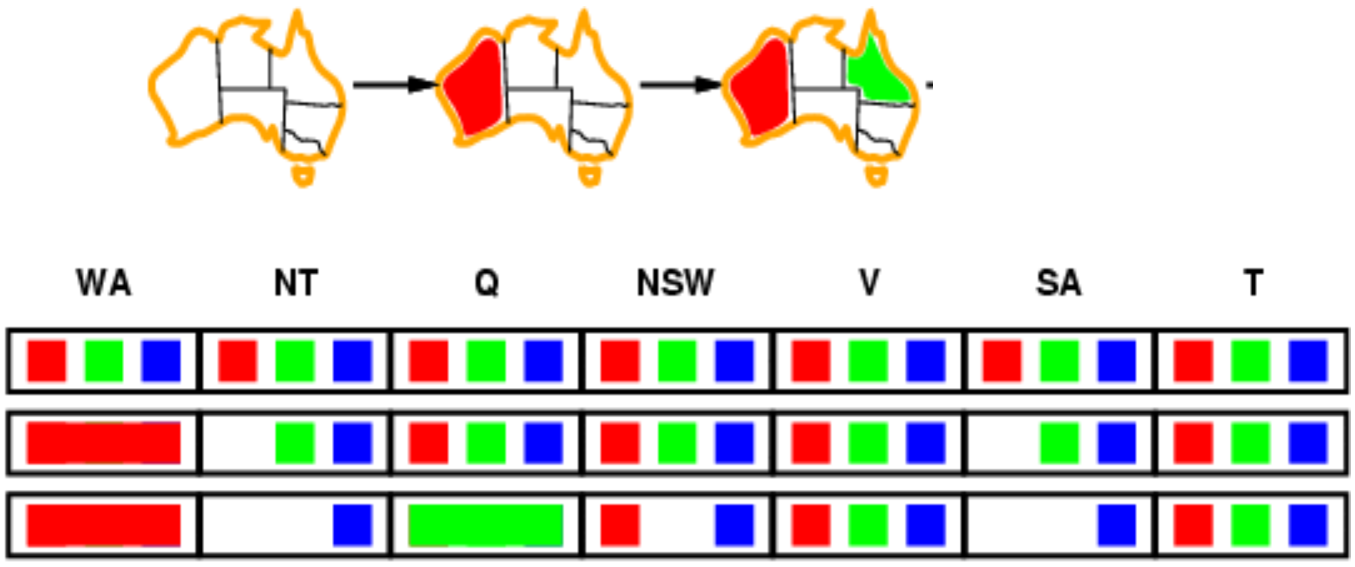
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values

H4: Forward checking



- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values

Constraint propagation



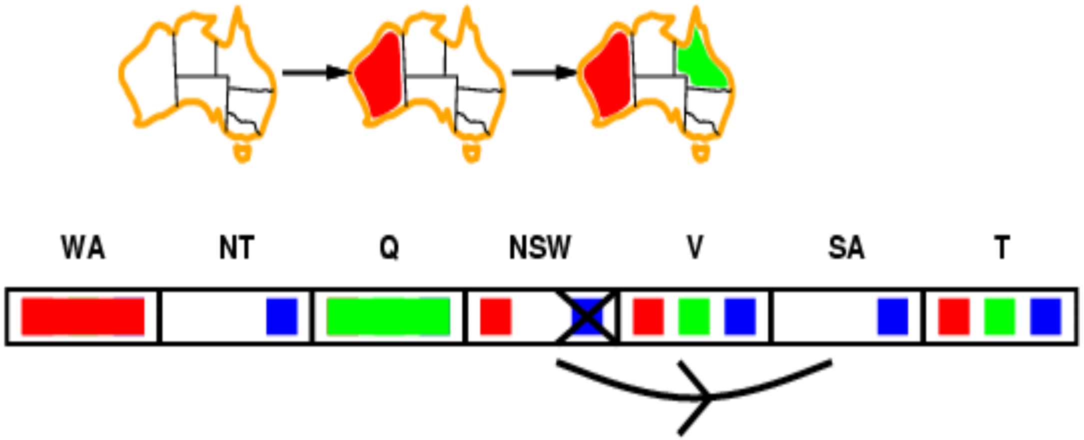
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:
  - NT and SA cannot both be blue!
- Constraint propagation repeatedly enforces constraints locally

Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value x of X there is some allowed y

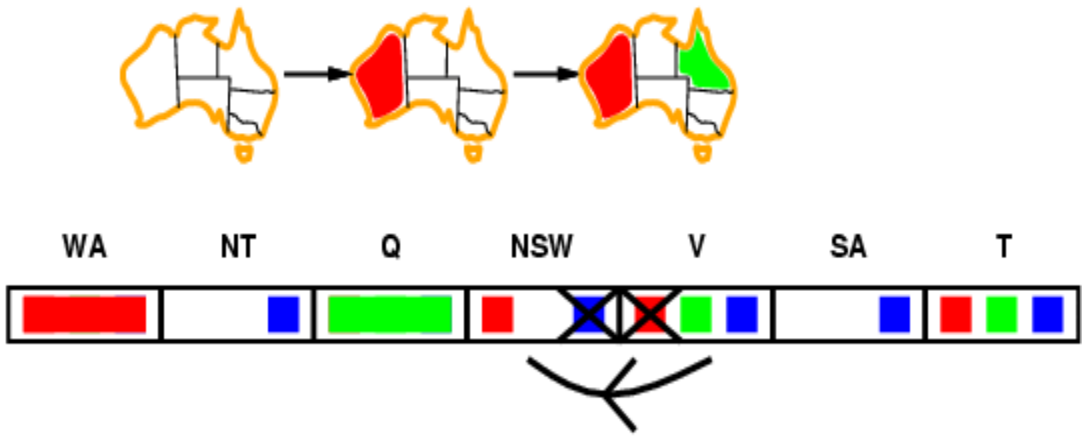
Arc consistency



- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value x of X there is some allowed y

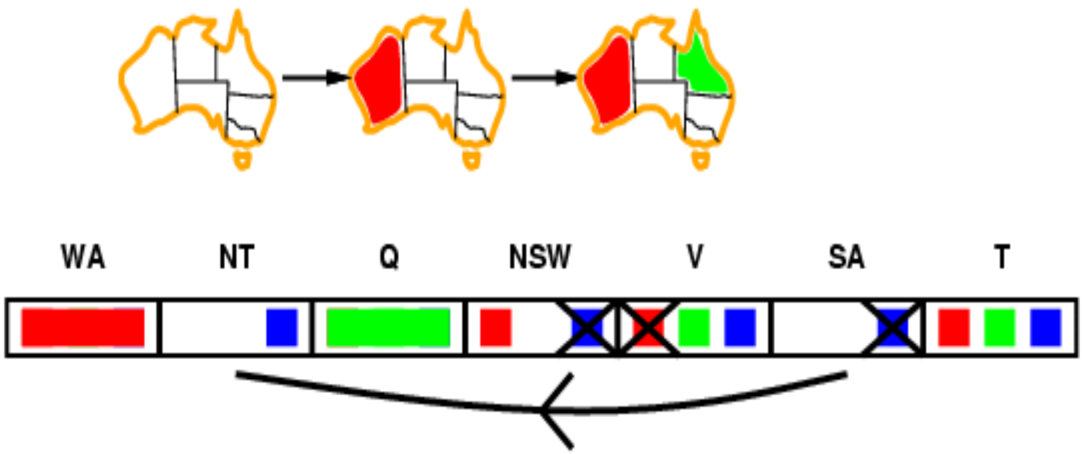
Arc consistency



- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value x of X there is some allowed y
- If X loses a value, neighbors of X need to be rechecked

Arc consistency



- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value x of X there is some allowed y
- 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

Arc consistency algorithm AC-3

function AC-3(*csp*) **returns** the CSP, possibly with reduced domains
  inputs: *csp*, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$
  local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

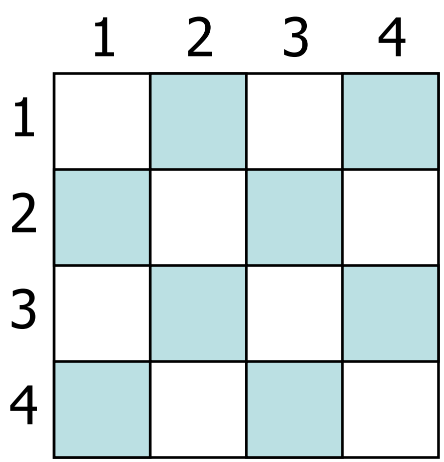
  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)
    if RM-INCONSISTENT-VALUES(X_i, X_j) **then**
      for each X_k **in** NEIGHBORS[X_i] **do**
        add (X_k, X_i) to *queue*

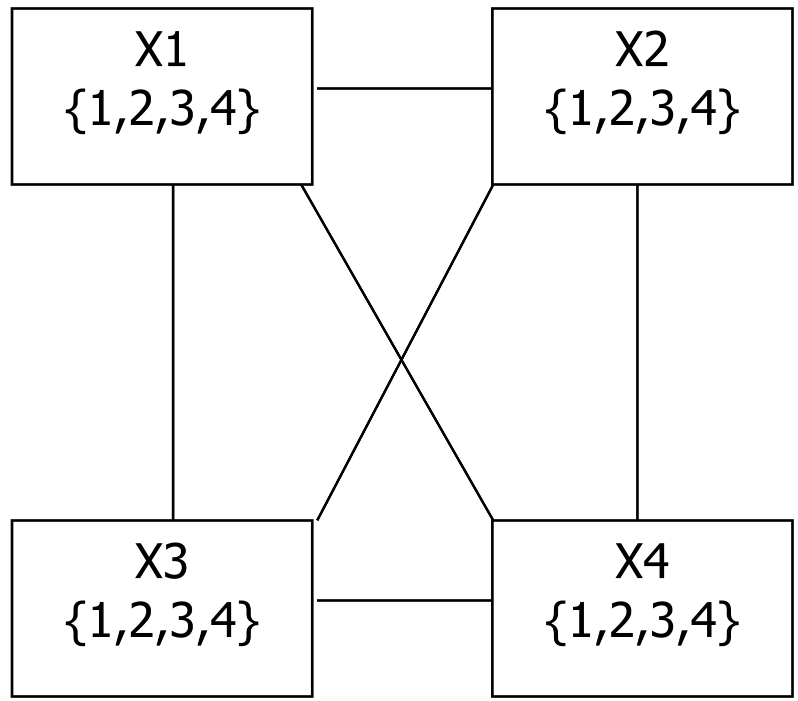
---

function RM-INCONSISTENT-VALUES(X_i, X_j) **returns** true iff remove a value
  removed $\leftarrow$ *false*
  for each x **in** DOMAIN[X_i] **do**
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy constraint(X_i, X_j)
      then delete x from DOMAIN[X_i]; *removed* $\leftarrow$ *true*
  return *removed*

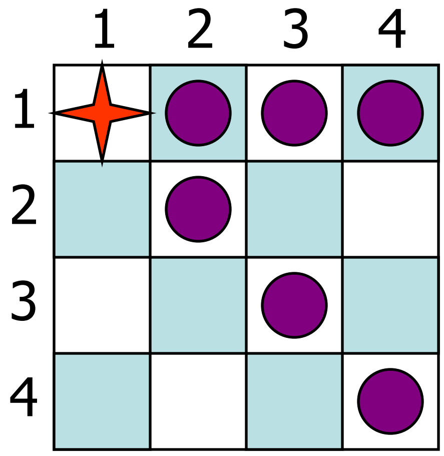
- Time complexity: $O(n^2d^3)$

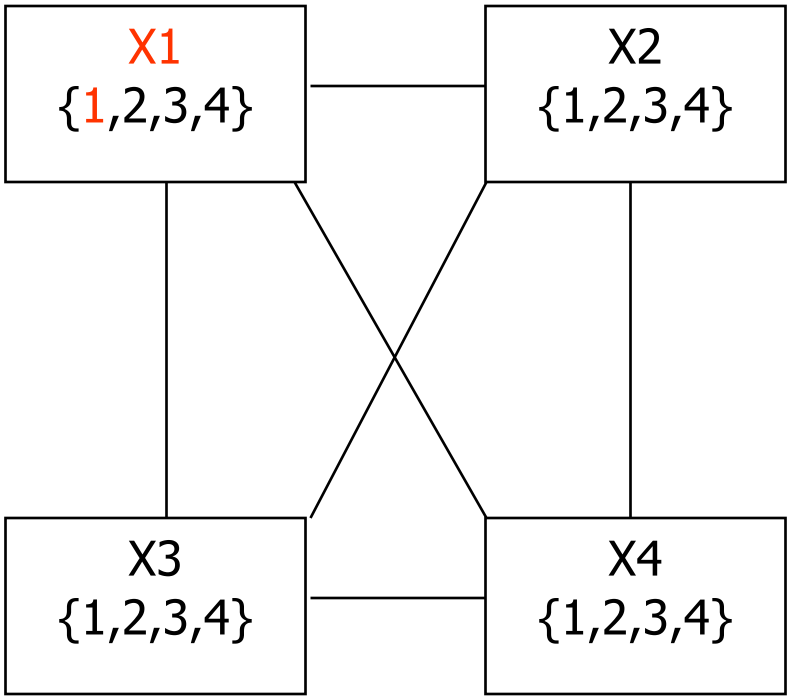
Example: 4-Queens Problem



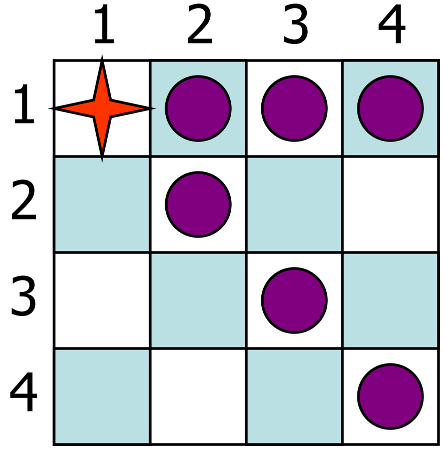


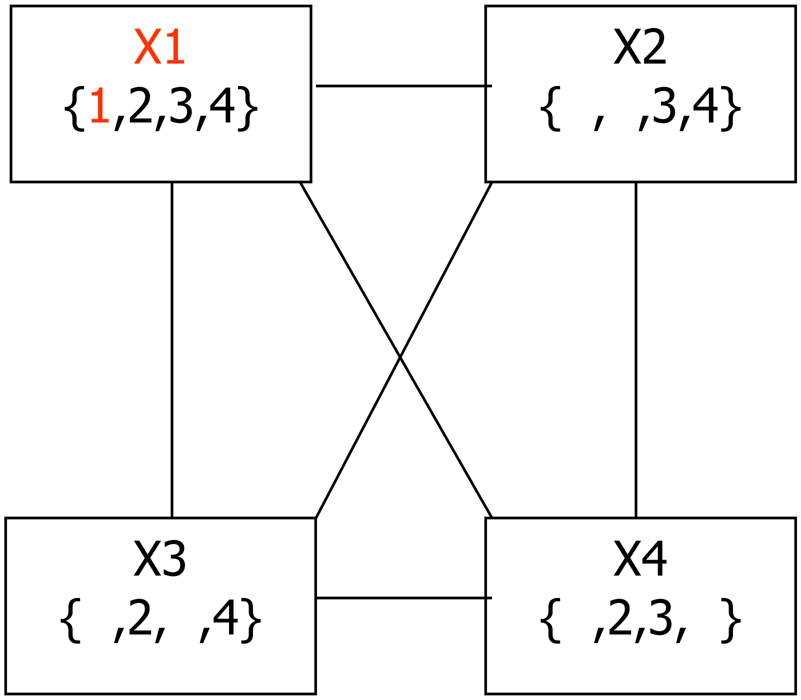
Example: 4-Queens Problem



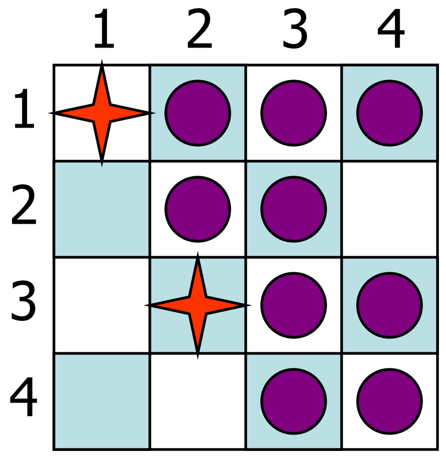


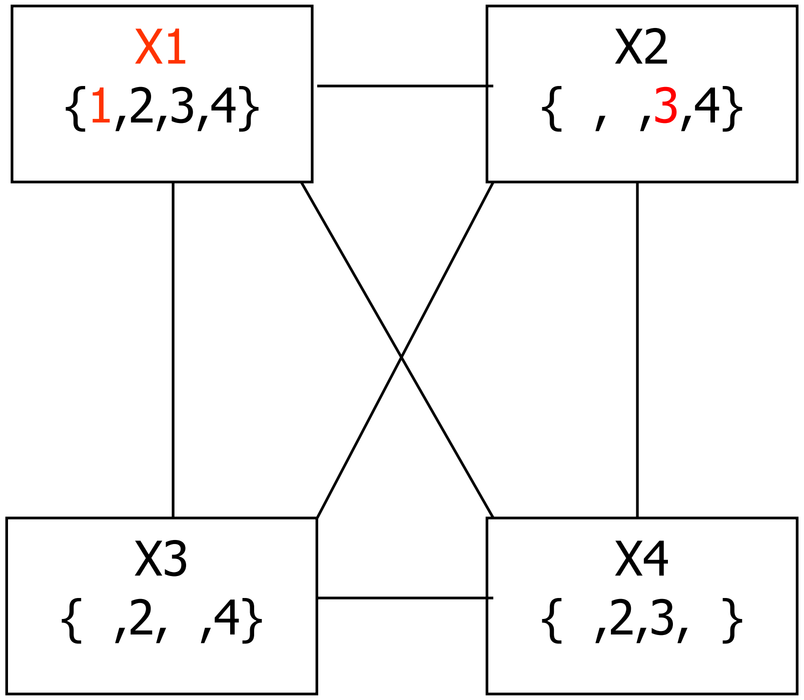
Example: 4-Queens Problem



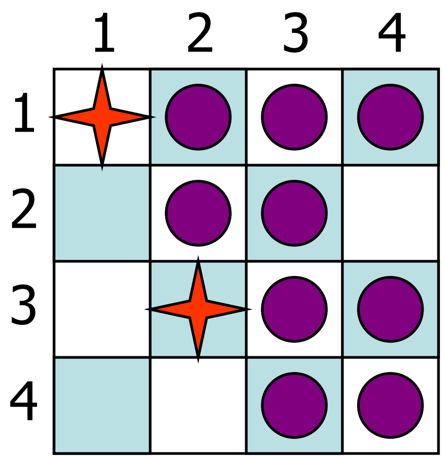


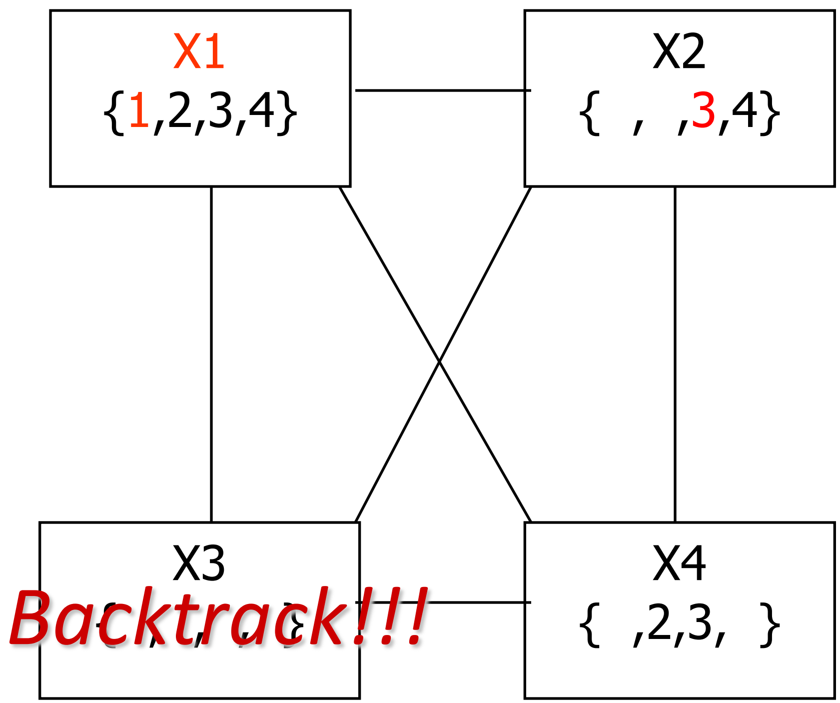
Example: 4-Queens Problem





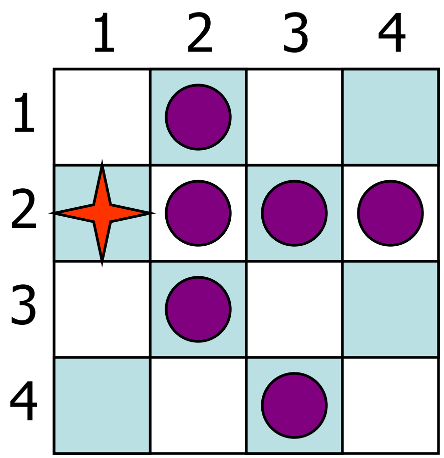
Example: 4-Queens Problem

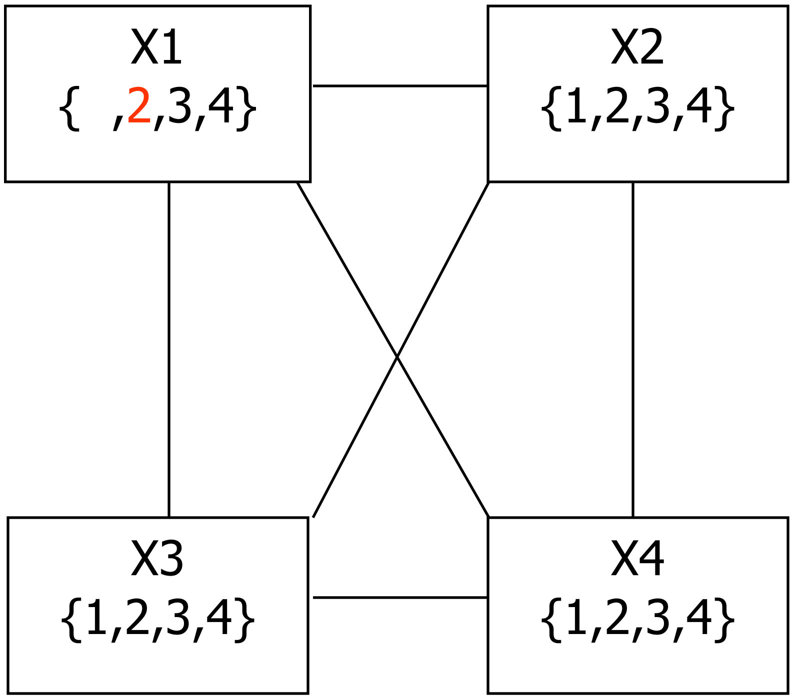




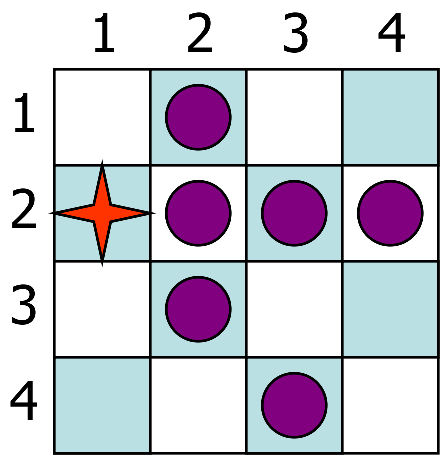
Example: 4-Queens Problem

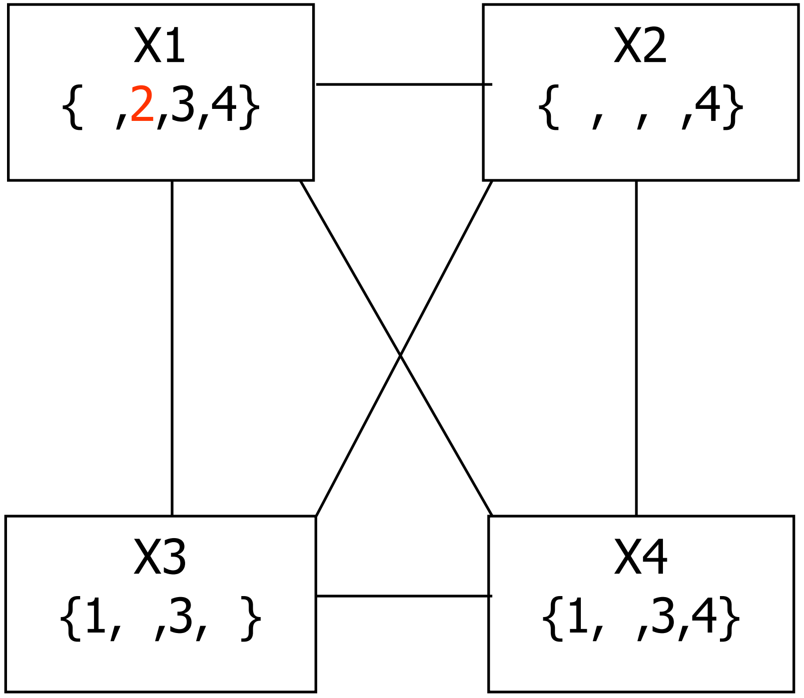
Picking up a little later after two steps of backtracking….



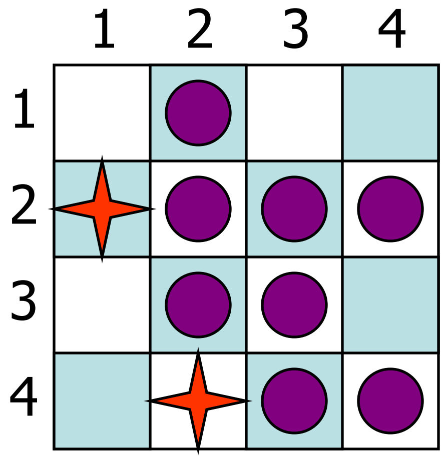


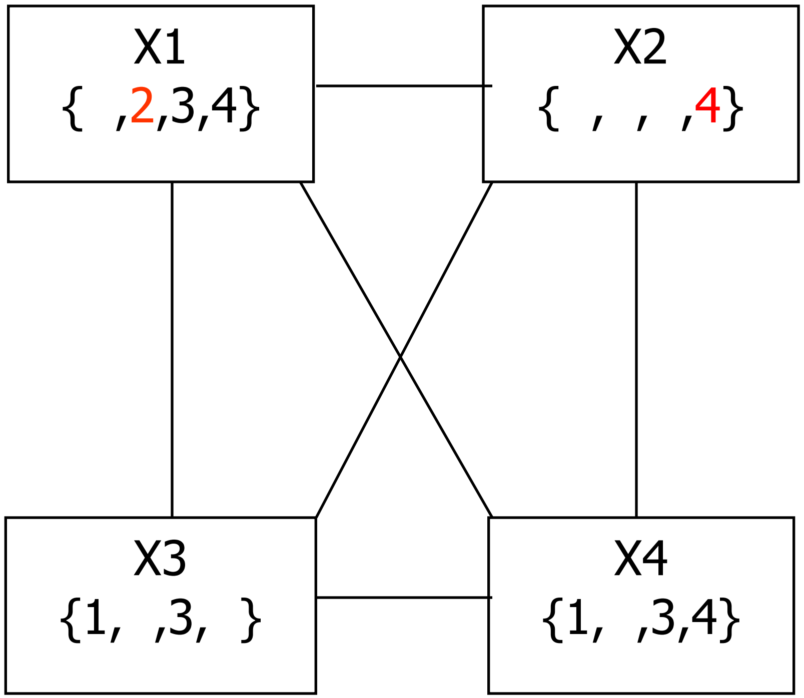
Example: 4-Queens Problem



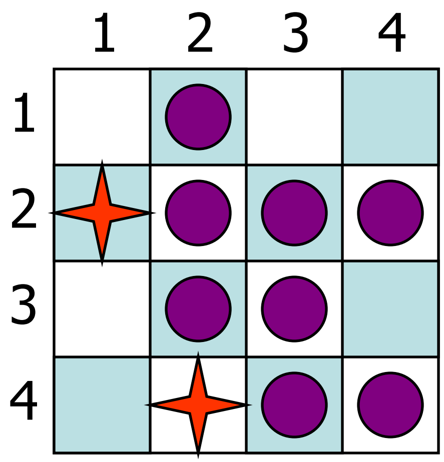


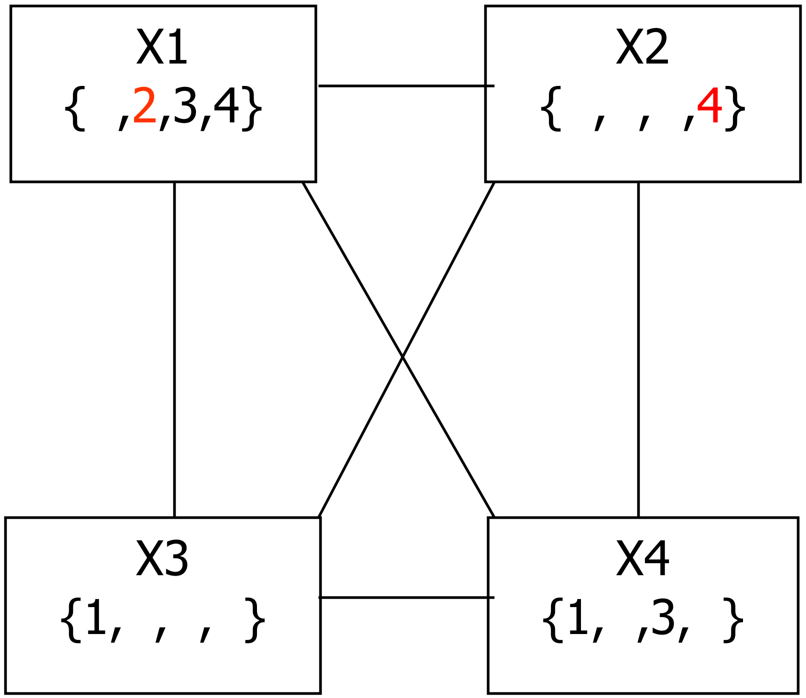
Example: 4-Queens Problem



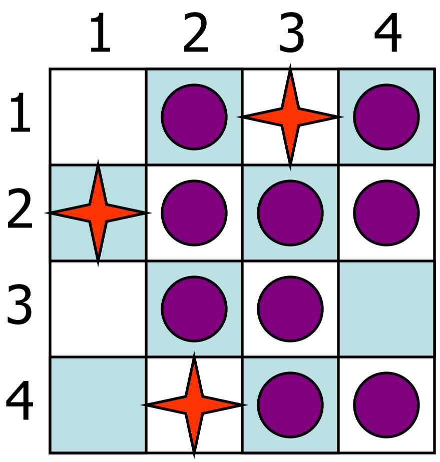


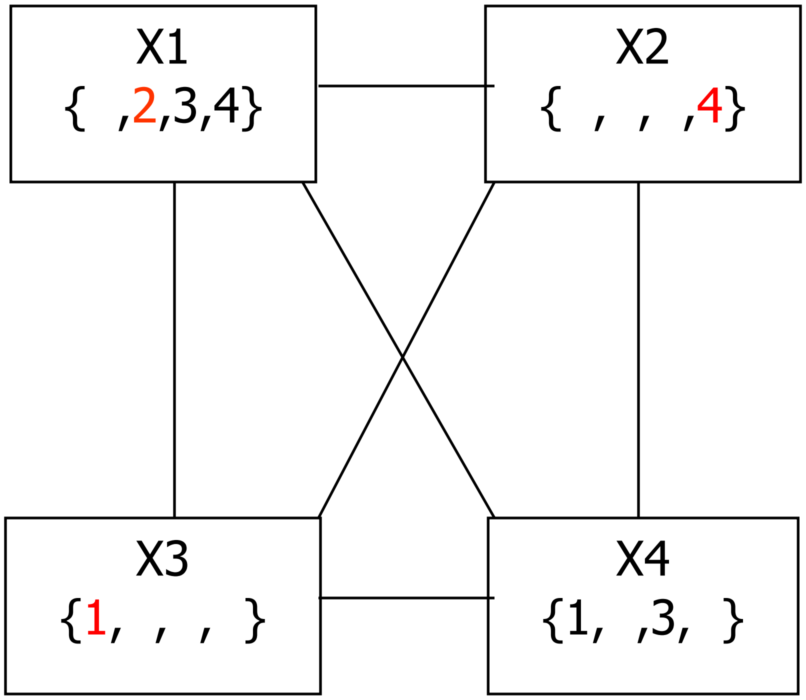
Example: 4-Queens Problem



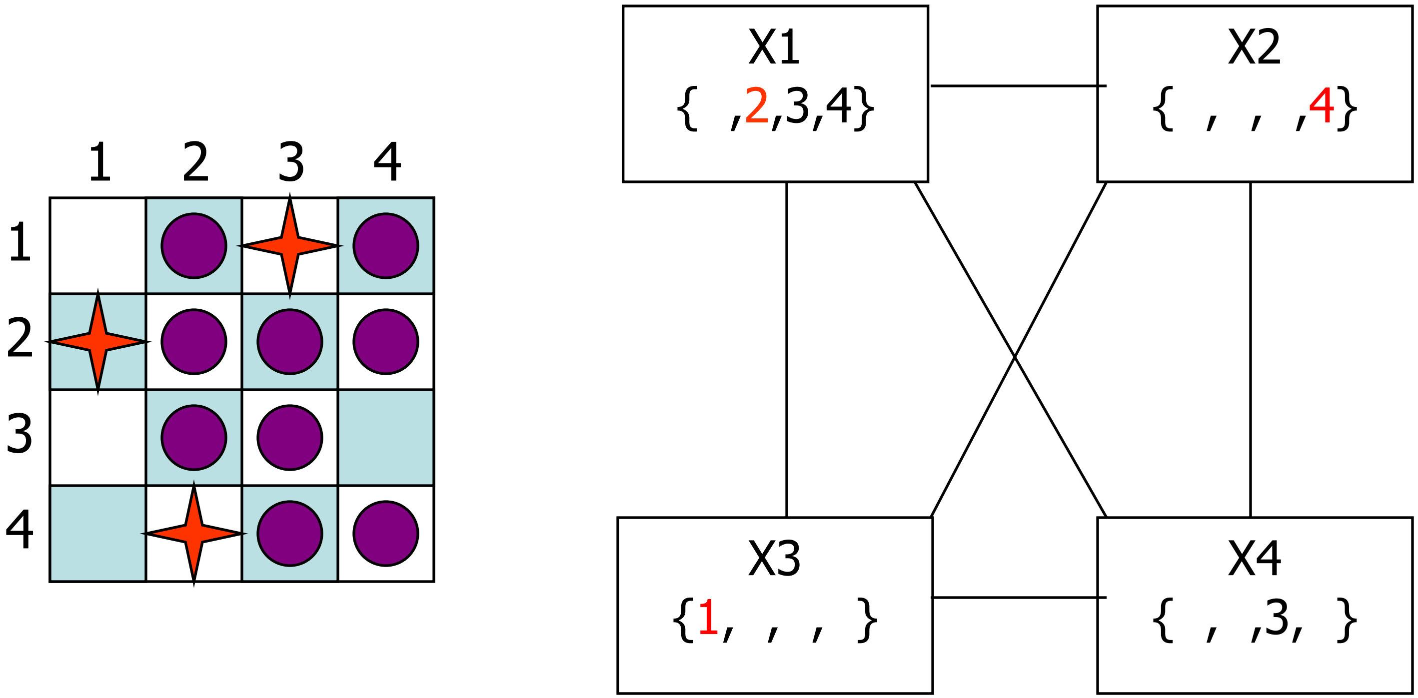


Example: 4-Queens Problem

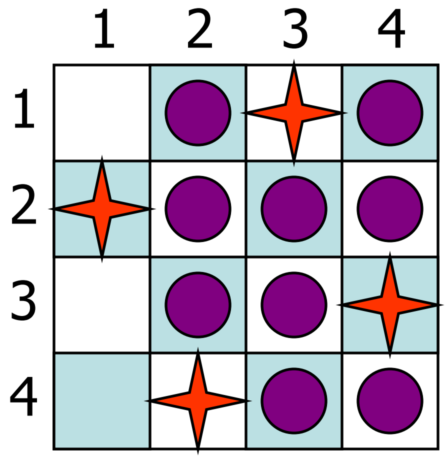


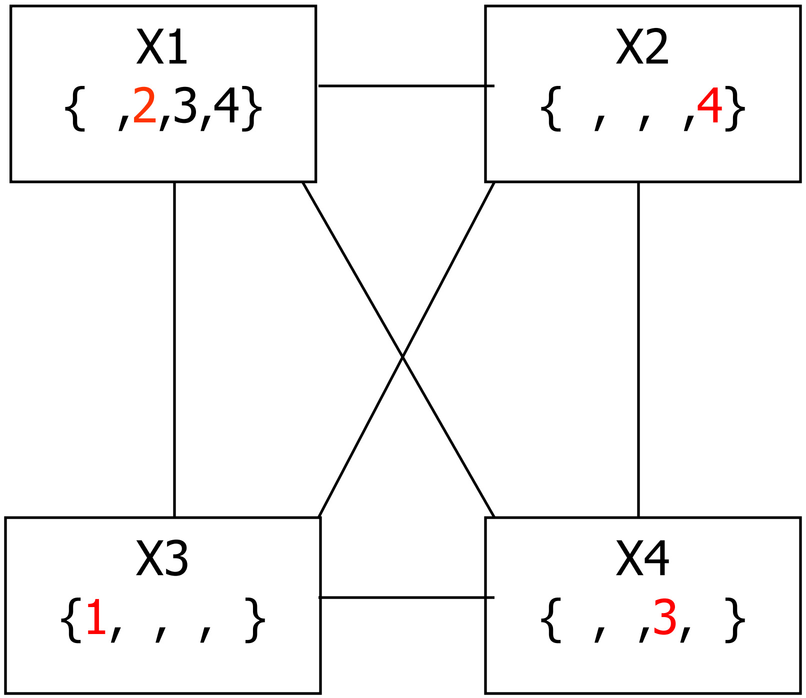


Example: 4-Queens Problem



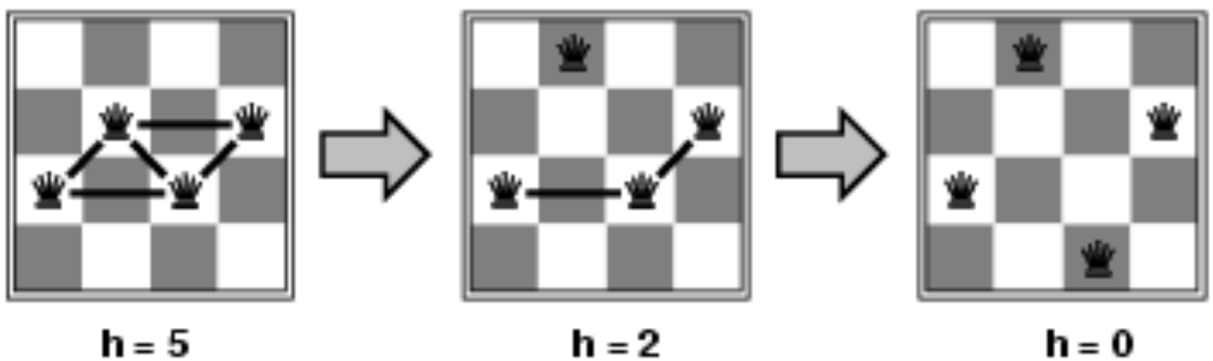
Example: 4-Queens Problem





Example: n-queens

- States: *n* queens in *n* columns (n^n states)
- Actions: move queen in column
- Goal test: no attacks
- Evaluation: $h(n)$ = number of attacks



- Given random initial state, AC-3 can solve *n*-queens in almost constant time for arbitrary *n* with high probability (e.g., *n* = 10,000,000)

Summary

- CSPs are a special kind of problem:
  - states defined by values of a fixed set of variables
  - goal test defined by constraints on variable values
- Backtracking = depth-first search with one variable assigned per node
- Heuristics help significantly
- Forward checking prevents assignments that guarantee later failure
- Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies