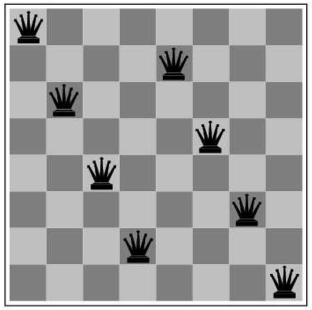
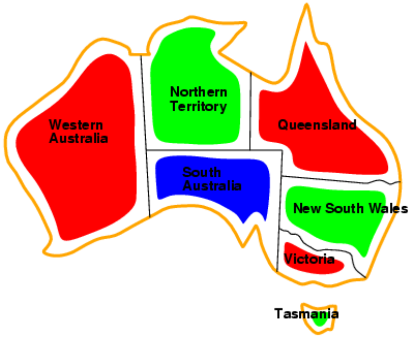
Constraint Satisfaction Problems

We've seen CSP before!

- Constraint satisfaction problem (CSP) is a special class of search problem





- Each problem has a set of variables (e.g. A,B,C,D,E)
- Each variable take a value from a domain (e.g. {T,F})
- Each problem has a set of constraints
- Objective: find a complete assignment of variables that satisfies all the constraints.
- What are v/v/d/c of 8-queen? Map coloring?

CSP definition

- CSP is a triplet $\{V, D, C\}$
- $V = \{V_1, V_2, \ldots, V_n\}$ a finite set of variables
- Each variable may be assigned a value from domain D_i
- Each member of C is a pair
  - First member: a subset of variables
  - Second member: a set of valid values
- Example:

$V = \{V_1, V_2, \ldots, V_7\}$

$D = \{\mathrm{R, G, B}\}$

$C = \{\ (V_1, V_2)$:{(R,G), (R,B), (G,B), (G,R), (B,G), (B,R)},

(V_1, V_3):{(R,G), (R,B), (G,B), (G,R), (B,G), (B,R)},

…

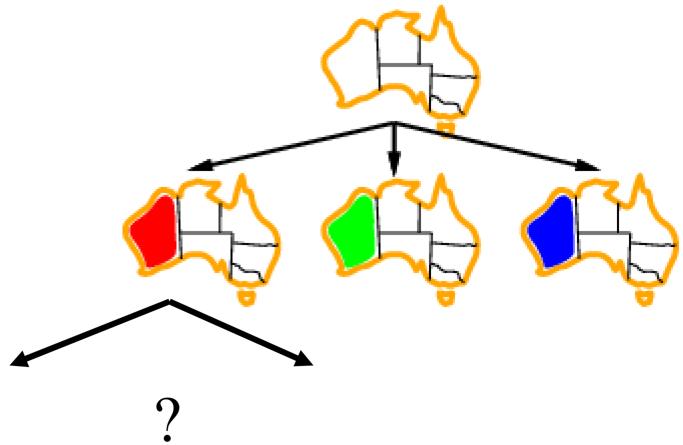
} (obvious point: C is often represented as a function)

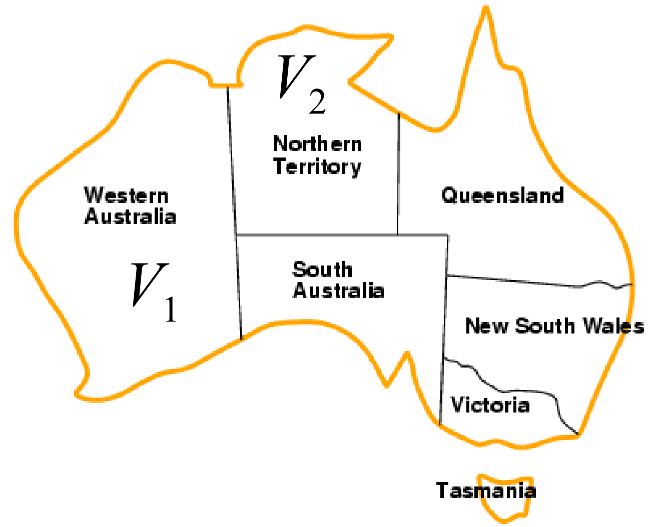
- How did we solve this?

Old solution #2: BFS, DFS, …

- State: partial assignment. ($V_1 \ldots V_{k-1}$ assigned, $V_k \ldots V_n$ not yet).
- Start state: all variables unassigned
- Goal state: all assigned, constraints satisfied
- Successor of ($V_1 \ldots V_{k-1}$ assigned, $V_k \ldots V_n$ not yet): assign V_k with a value from D_k
- Cost on transitions: 0 is fine. We don't care. We just want any solution.

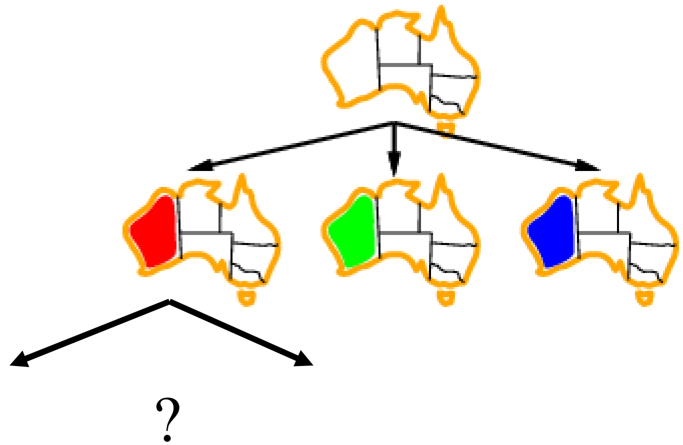
Map coloring example

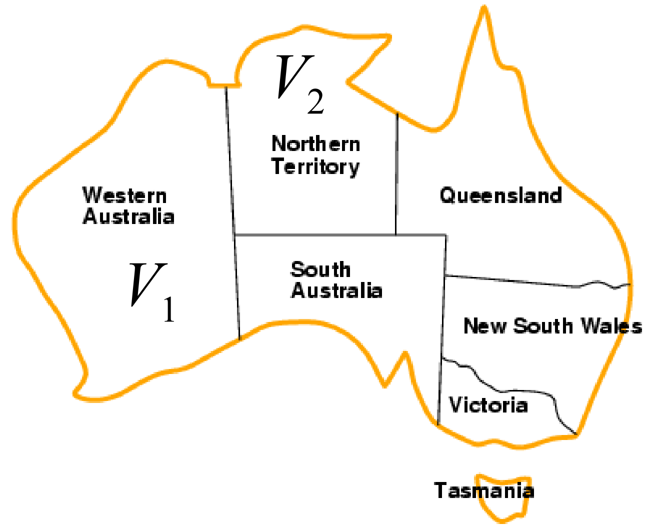




- It turns out BFS is bad. Why?

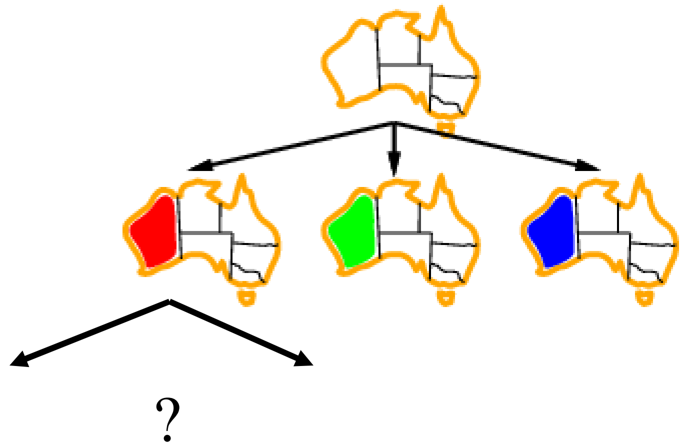
Map coloring example

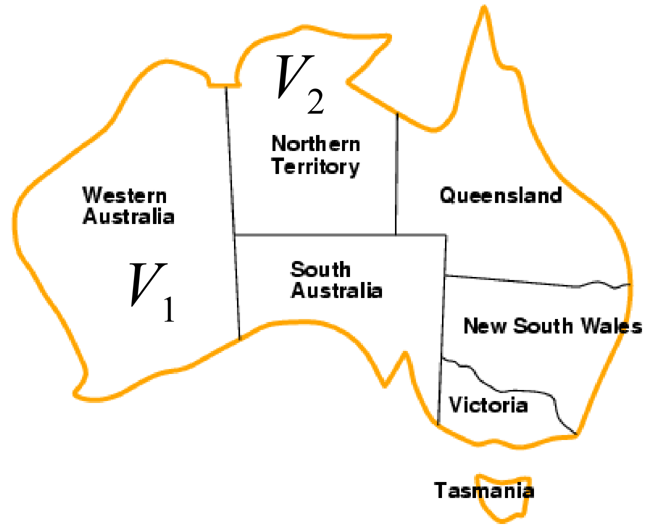




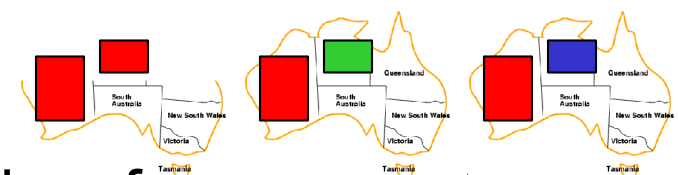
- It turns out BFS is bad. Why? Goal @ search tree leaf level.
- What are the successors above?

Map coloring example





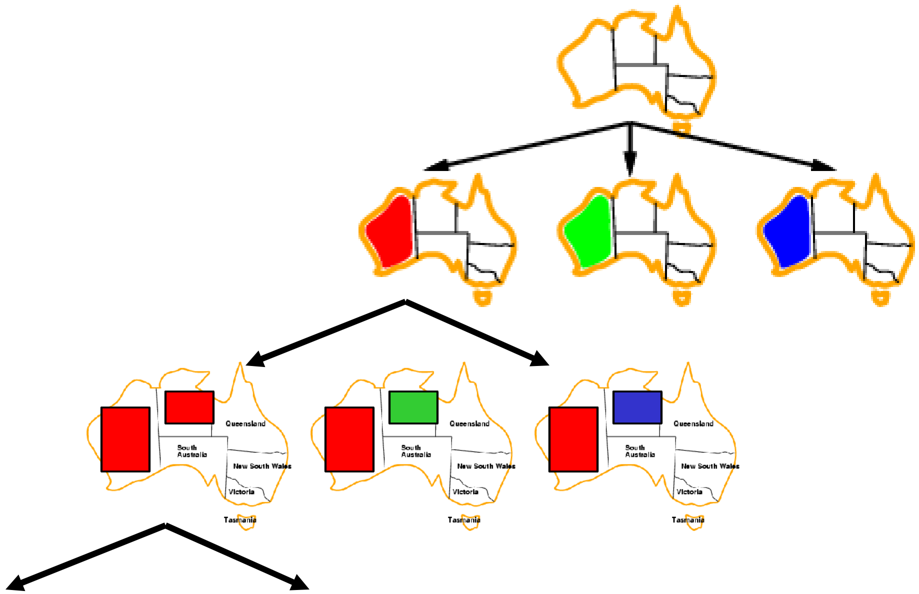
- It turns out BFS is bad. Why? Goal @ search tree leaf level.
- What are the successors above?



- Let's say for every variable the order of assignment is R, G, B. There's something wrong with DFS, can you see why?

Map coloring example

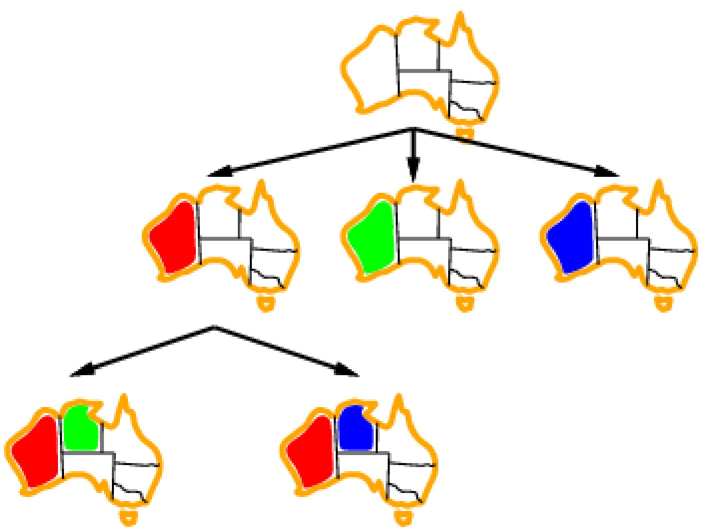
There's something wrong with DFS, can you see why?



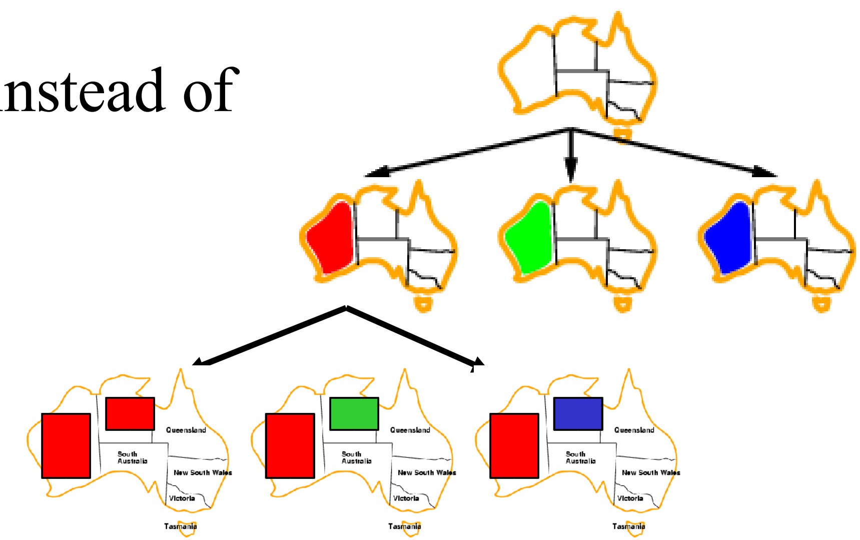
Shouldn't search
anything down here!

#1 Obvious improvement: backtracking search

- Succs() should check the constraints and not propose a successor assignment that conflicts with other already-assigned variables.
- ‘backtracking’ happens when no value is valid for that successor.



instead of

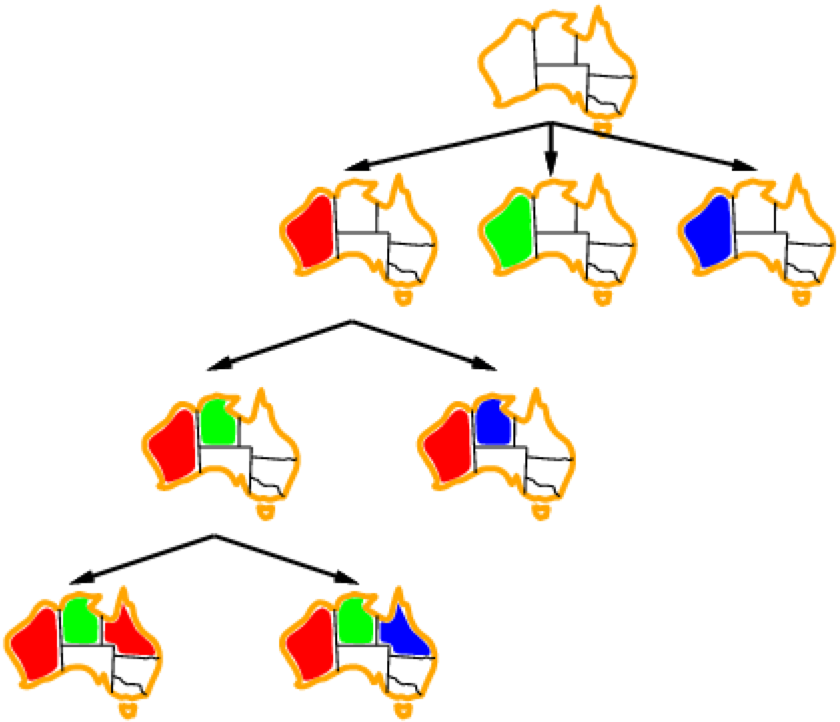


Backtracking search

function BACKTRACKING-SEARCH(csp) **returns** $solution/failure$
  return RECURSIVE-BACKTRACKING($[\,]$, csp)

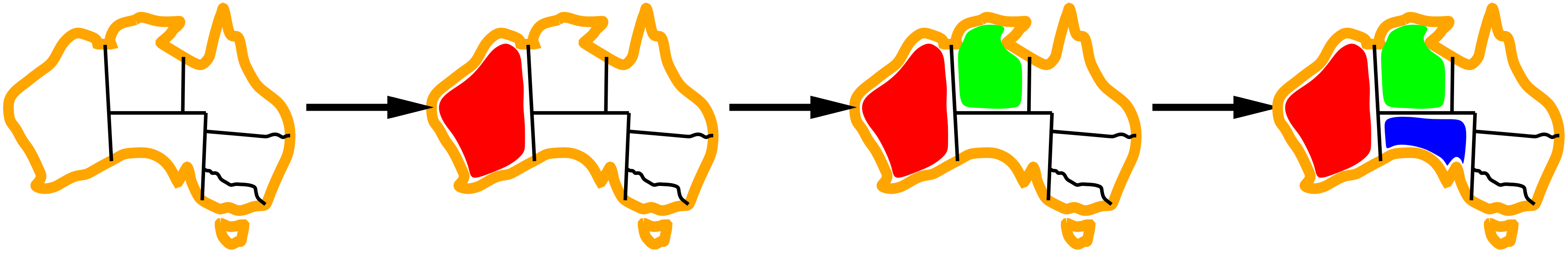
function RECURSIVE-BACKTRACKING($assigned$, csp) **returns** $solution/failure$
  if $assigned$ is complete **then return** $assigned$
  $var \leftarrow$ SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], $assigned$, csp)
  for each $value$ **in** ORDER-DOMAIN-VALUES(var, $assigned$, csp) **do**
    if $value$ is consistent with $assigned$ according to CONSTRAINTS[csp] **then**
      $result \leftarrow$ RECURSIVE-BACKTRACKING($[var = value | assigned]$, csp)
      if $result \neq failure$ **then return** $result$
  return $failure$

Backtracking search example



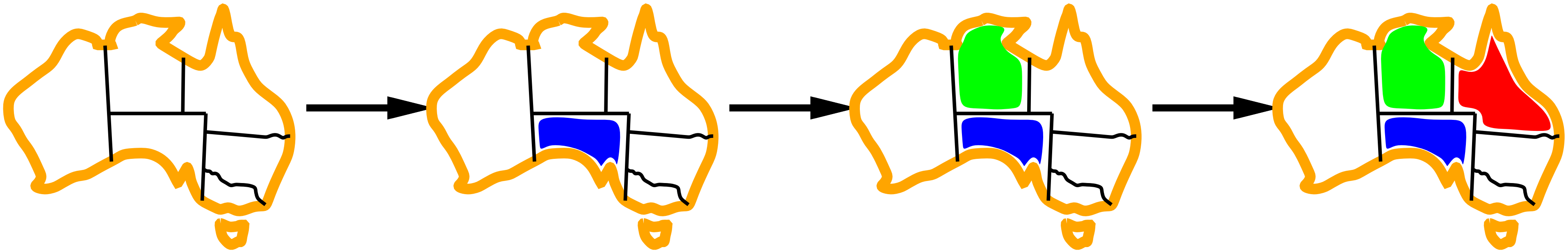
Minimum remaining values (MRV)

- aka most constrained variable
- choose the variable with the fewest legal values
  - most likely to cause early failure (**prune** the search tree)
  - e.g. variable with 0 values should cause immediate failure



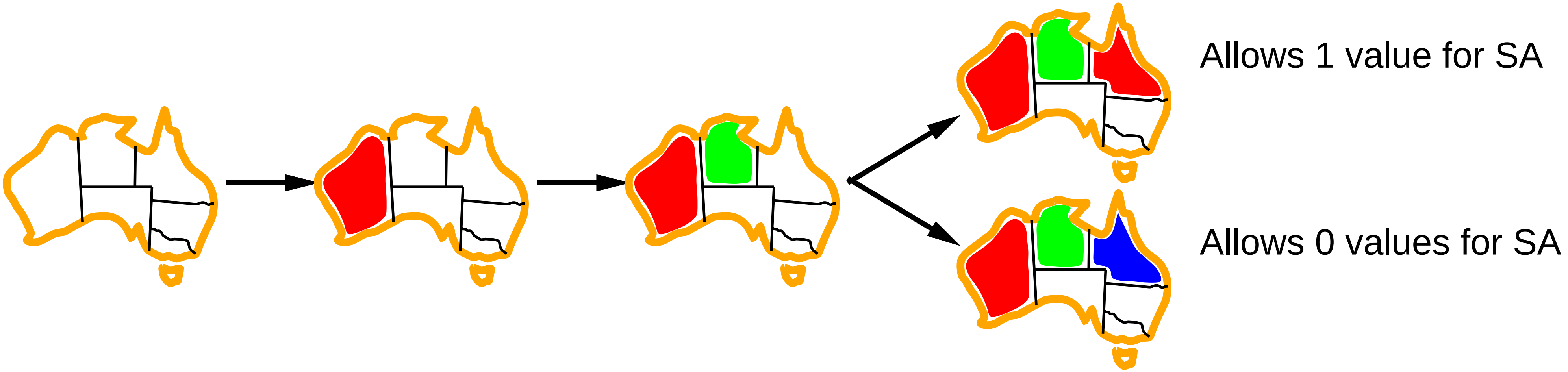
Degree heuristic

- there can be many variables with the same number of values
- choose variable with most constraints on remaining variables
  - reduces branching factor in future choices
  - used as tie-breaker among most constrained variables



Least constraining value

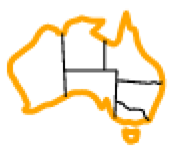
- given a variable, how to order the values to try
- choose the least constraining value
  - maximum flexibility for assignments on other vars

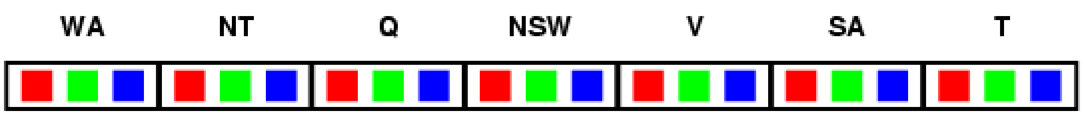


- doesn't matter if
  - we're looking for all the solutions, or
  - there's no solution

#2 Less obvious improvement: forward checking

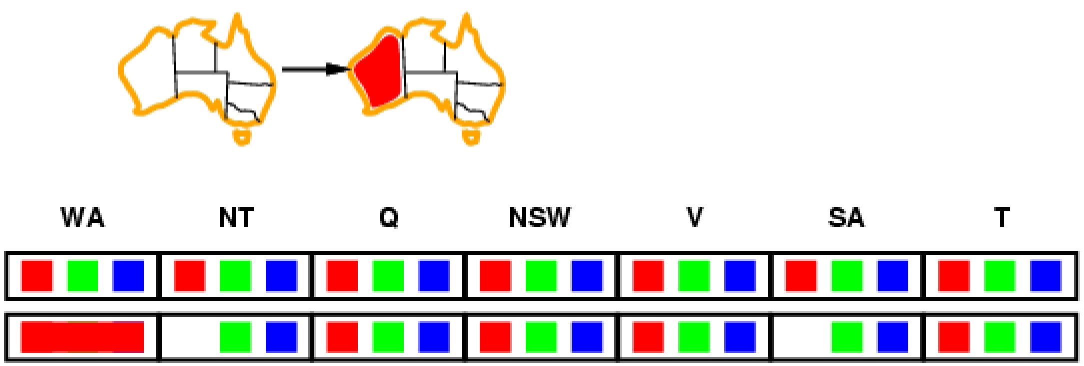
- Keep a list of candidate values for each unassigned variable.
- After assigning $V_i = v$, cross out conflicting candidates in other unassigned variables.
- If any unassigned variable's candidate list becomes empty, backtrack immediately.





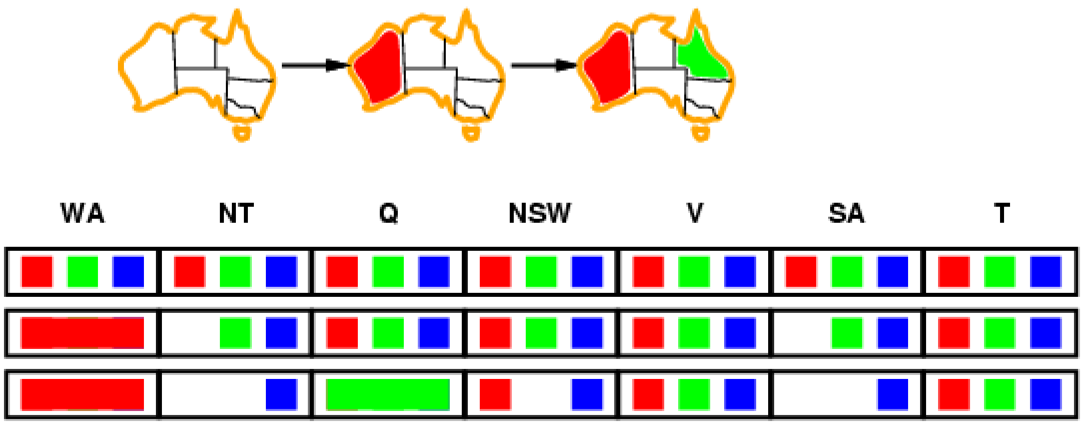
Less obvious: forward checking

- Keep a list of candidate values for each unassigned variable.
- After assigning $V_i = v$, cross out conflicting candidates in other unassigned variables.
- If any unassigned variable's candidate list becomes empty, backtrack immediately.



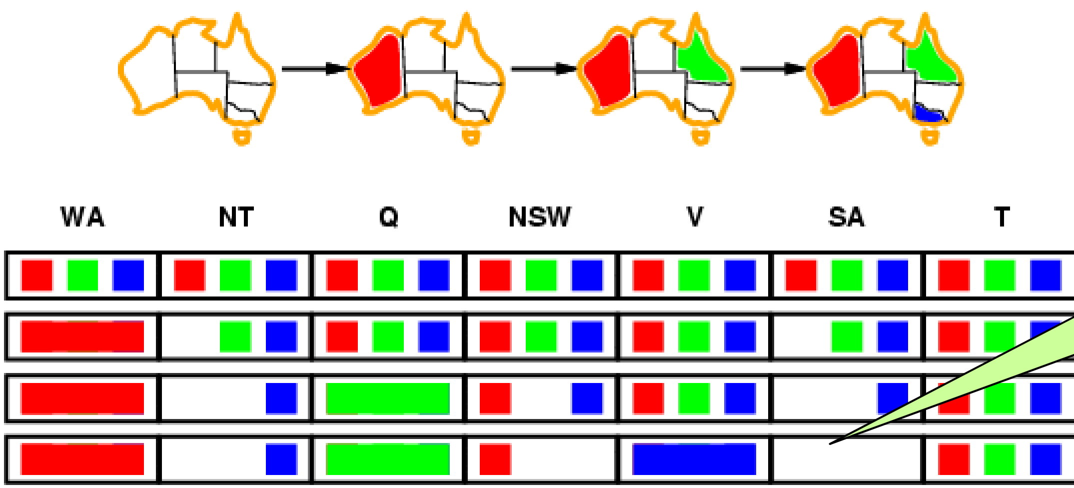
Less obvious: forward checking

- Keep a list of candidate values for each unassigned variable.
- After assigning $V_i=v$, cross out conflicting candidates in other unassigned variables.
- If any unassigned variable's candidate list becomes empty, backtrack immediately.

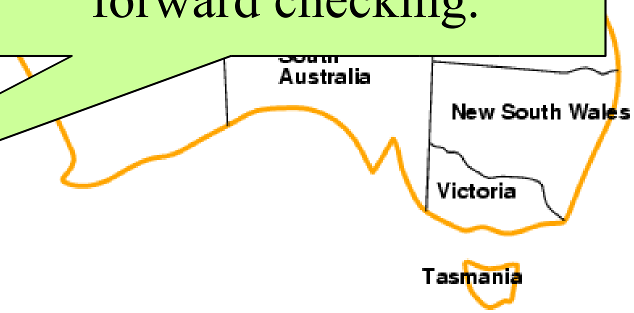


Less obvious: forward checking

- Keep a list of candidate values for each unassigned variable.
- After assigning V_i=*v*, cross out conflicting candidates in other unassigned variables.
- If any unassigned variable's candidate list becomes empty, backtrack immediately.

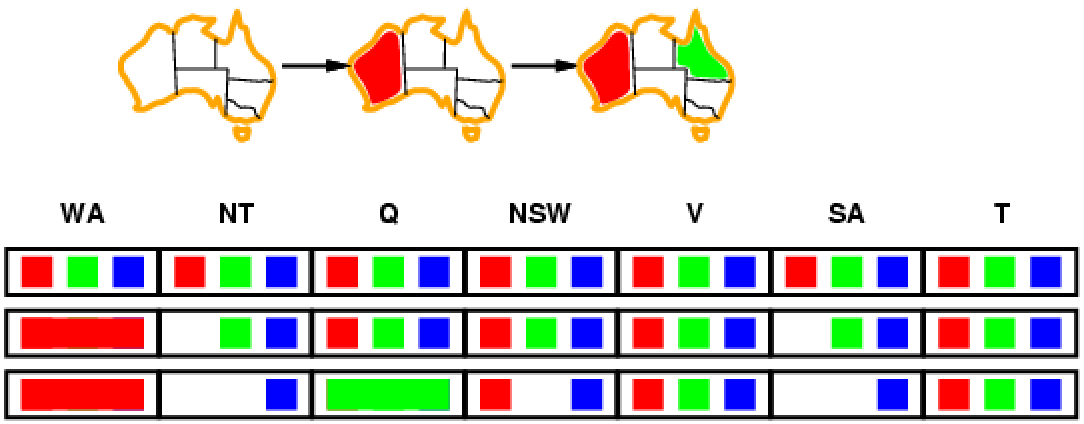


SA may not be the next variable we assign. Thus backtracking search has slower response than forward checking.



#3 Not obvious: constraint propagation

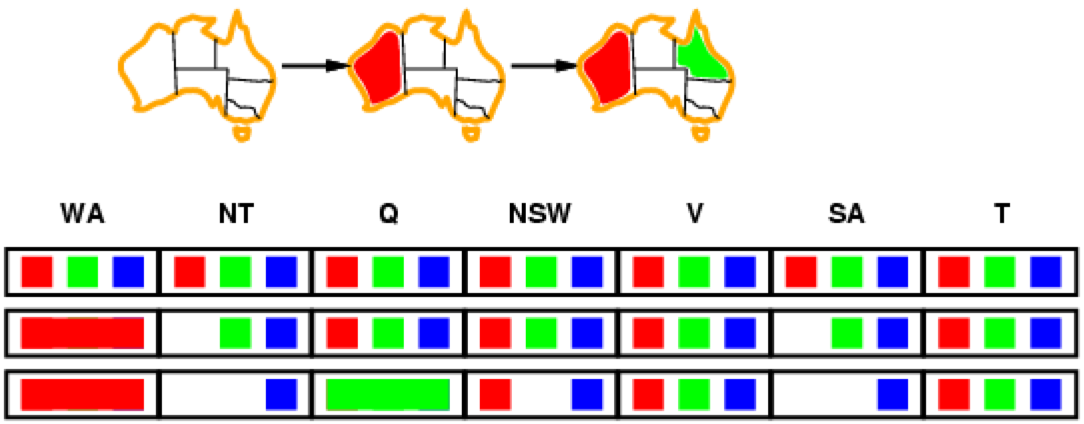
- Can NSW have the candidate value 'B'?

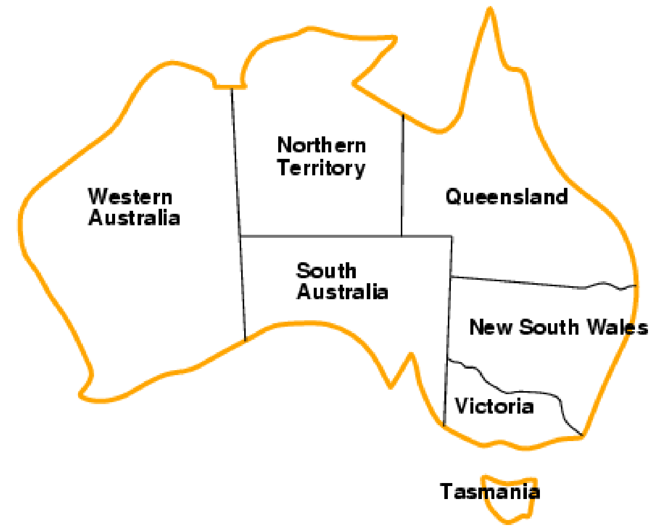


- Suppose NSW=B, would this cause problem for another unassigned variable?

#3 Not obvious: constraint propagation

- Can NSW have the candidate value ‘B’?

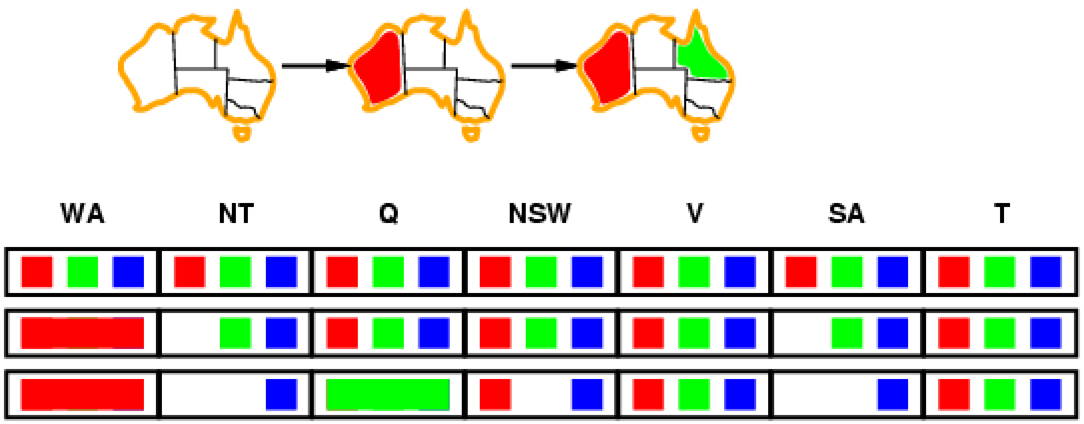


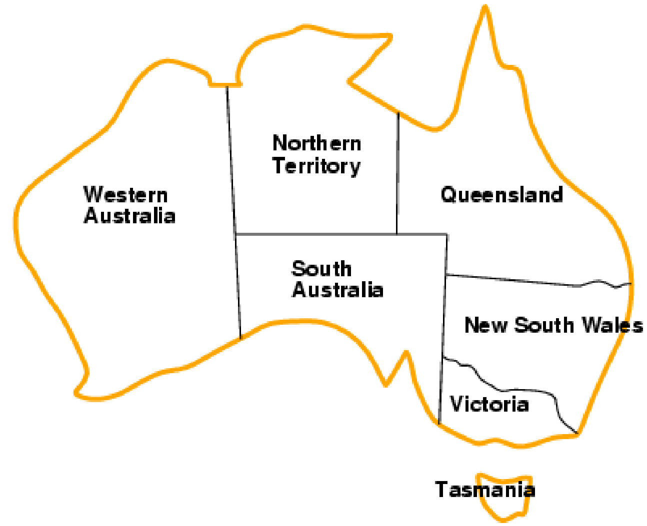


- Suppose NSW=B, would this cause problem for another unassigned variable? Yes! SA has no value to avoid a conflict!
- Because SA is not accommodating, we have to remove B from NSW’s candidates.

#3 Not obvious: constraint propagation

- Can NSW have the candidate value ‘B’?

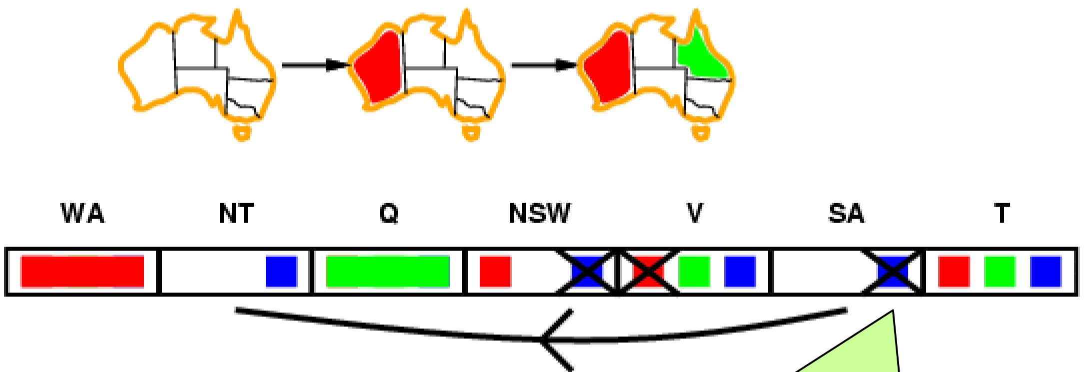




- Suppose NSW=B, would this cause problem for another unassigned variable? Yes! SA has no value to avoid a conflict!
- Because SA is not accommodating, we have to remove B from NSW’s candidates.
- But this makes NSW less accommodating. Another variable might lose a candidate value because of NSW now.
- That variable becomes less accommodating. And so on…

Constraint propagation

- After the dust settles, the candidate lists should be smaller (or at worst the same)
- If a variable loses all its candidates during this process, the current (partial) assignment is invalid, and we backtrack.



Constraint propagation detects failure one expansion earlier than forward checking, in this example.

Arc consistency algorithm

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 REMOVE-INCONSISTENT-VALUES(X_i, X_j) **then**
      for all X_k **in** NEIGHBORS[X_i] **do**
        add (X_k, X_i) to $queue$

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **returns** true/false
  $removed \leftarrow false$
  for all x **in** DOMAIN[X_i] **do**
    if ($\neg \exists y \in$ DOMAIN[X_j] s.t. $(x, y) \in$ constraint (X_i, X_j)) **then**
      delete x from DOMAIN[X_i]; $removed \leftarrow true$
  return $removed$

AC-3 called as preprocessing or after each assignment

Constraint propagation

- This is called arc consistency
- This is also known as 2-consistency. More generally *k*-consistency requires that

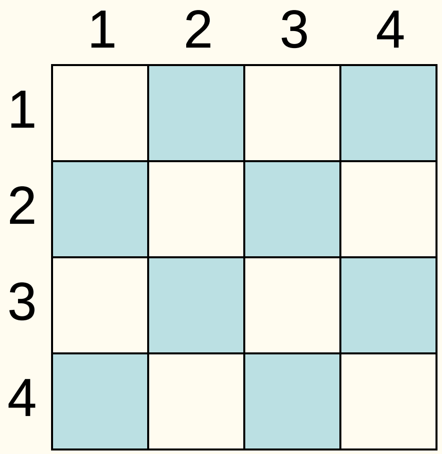
> For all groups of *k* variables, for all consistent combination of candidate values of the first *k*-1 variables, we can find a consistent candidate value for the k^{th} variable.

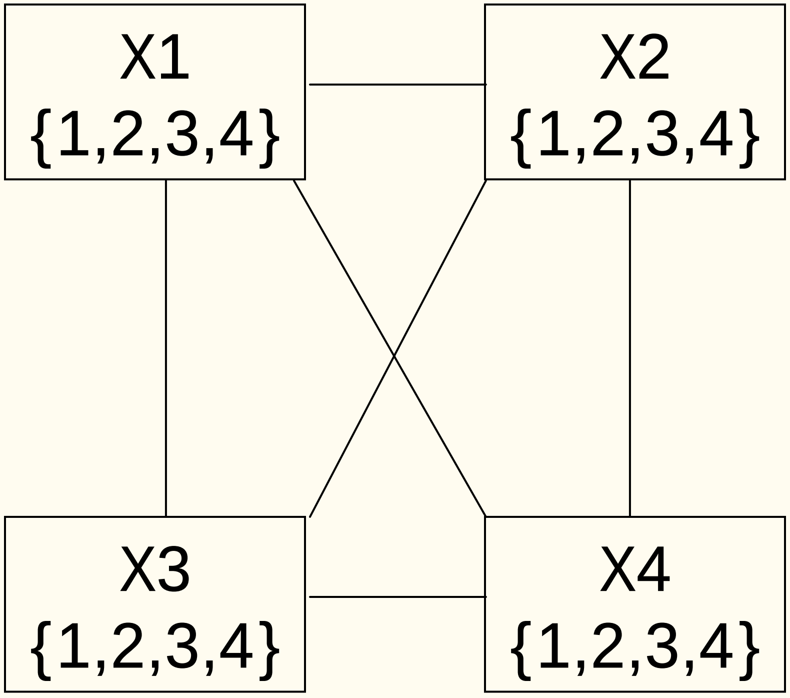
- More powerful, but exponentially more expensive to check. When *k*=*n* by definition it gives us the CSP solution!

What you should know

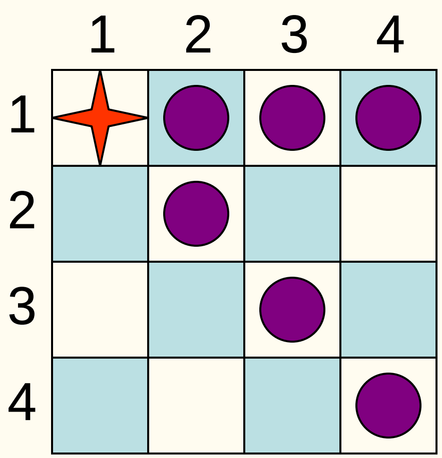
- How to formalize problems as CSP
- Backtracking search, forward checking, constraint propagation
- Variable ordering and value ordering

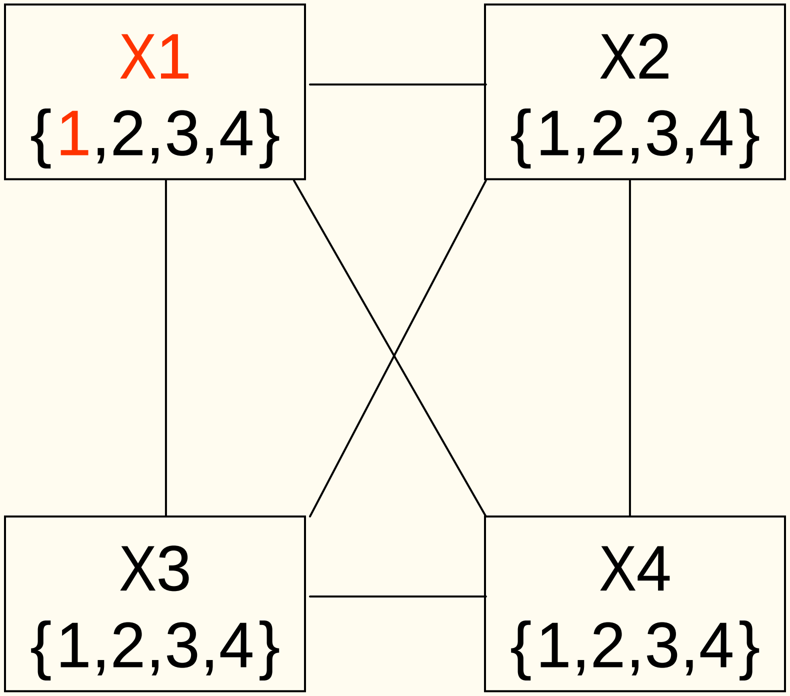
Example: 4-Queens Problem



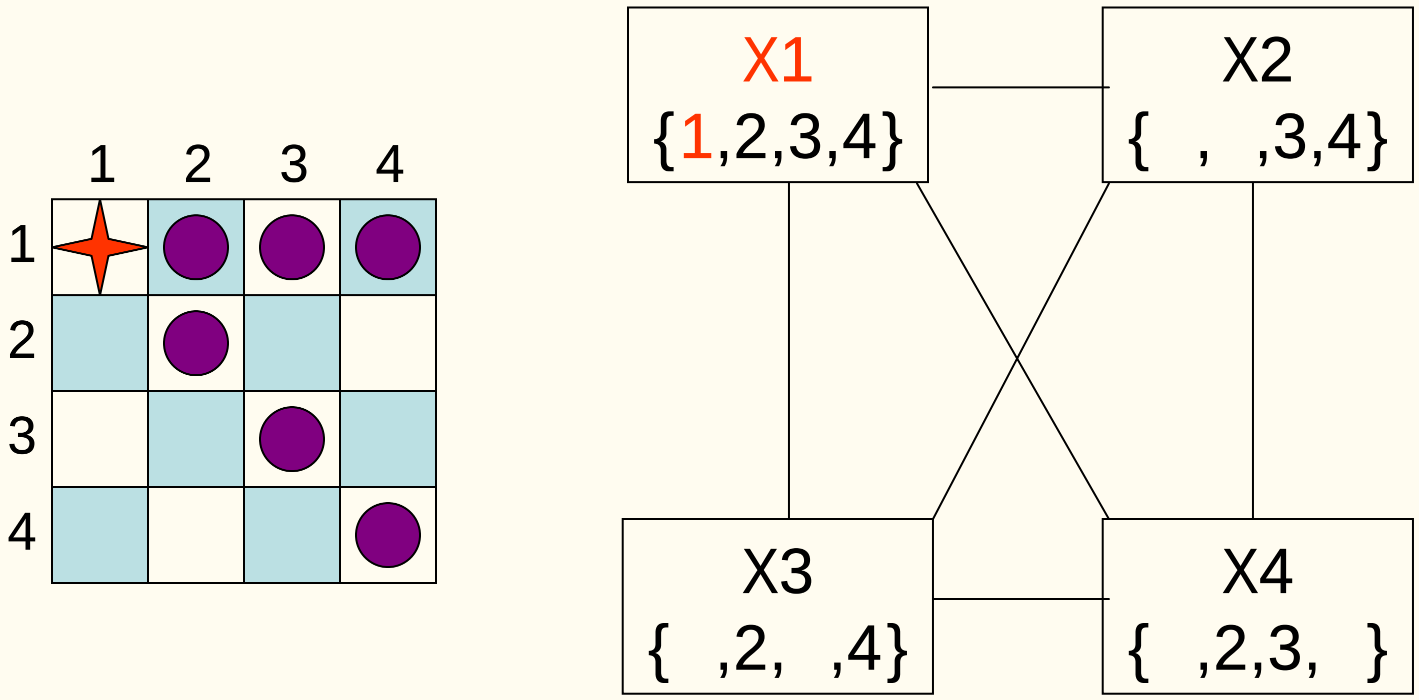


Example: 4-Queens Problem

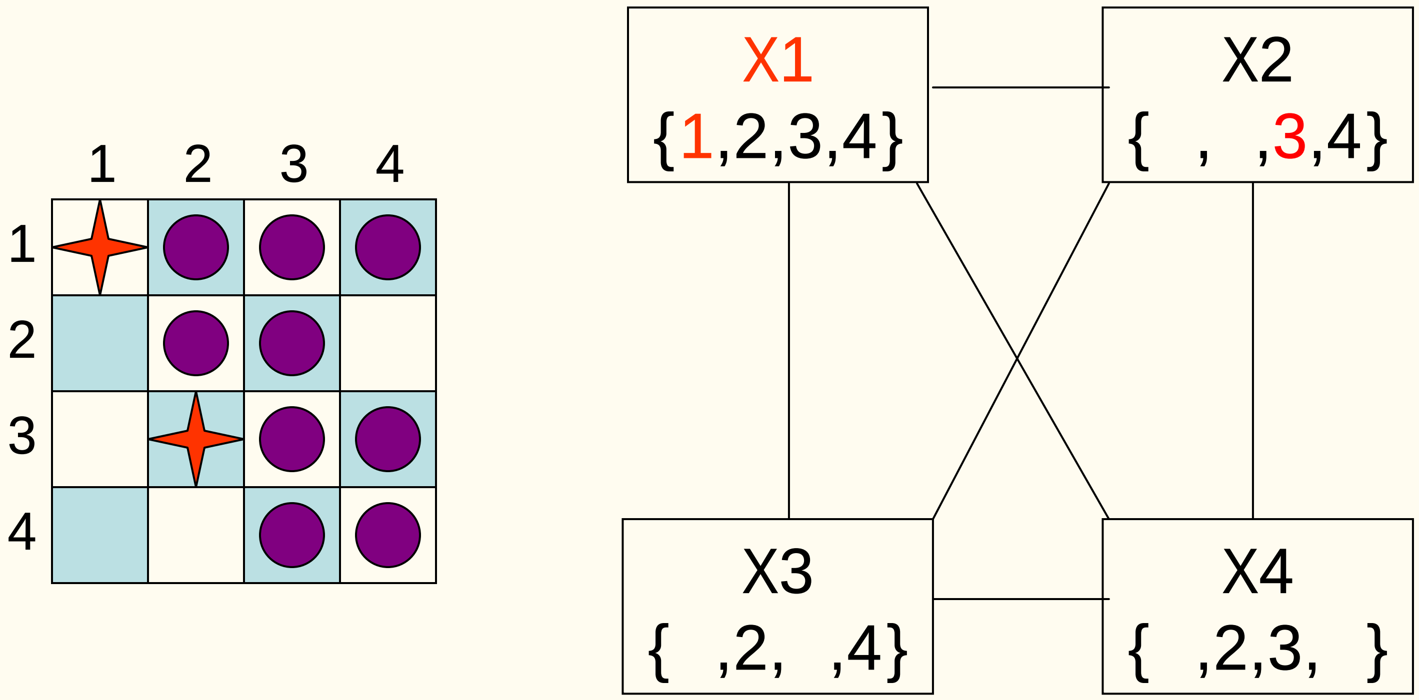




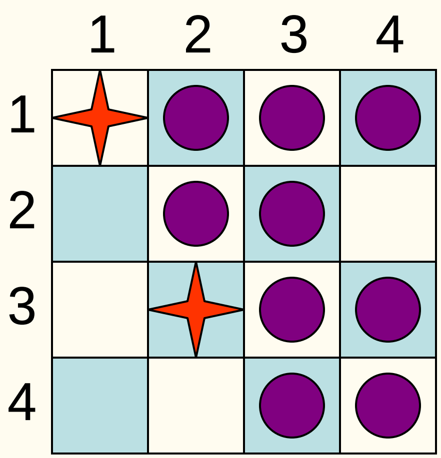
Example: 4-Queens Problem

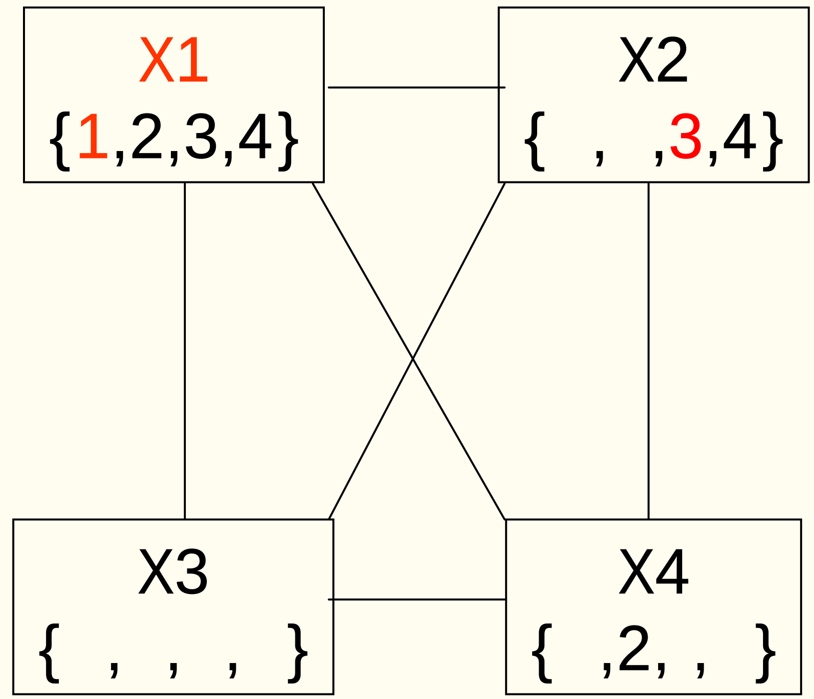


Example: 4-Queens Problem

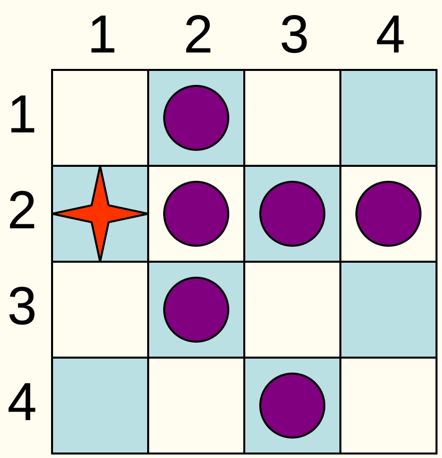


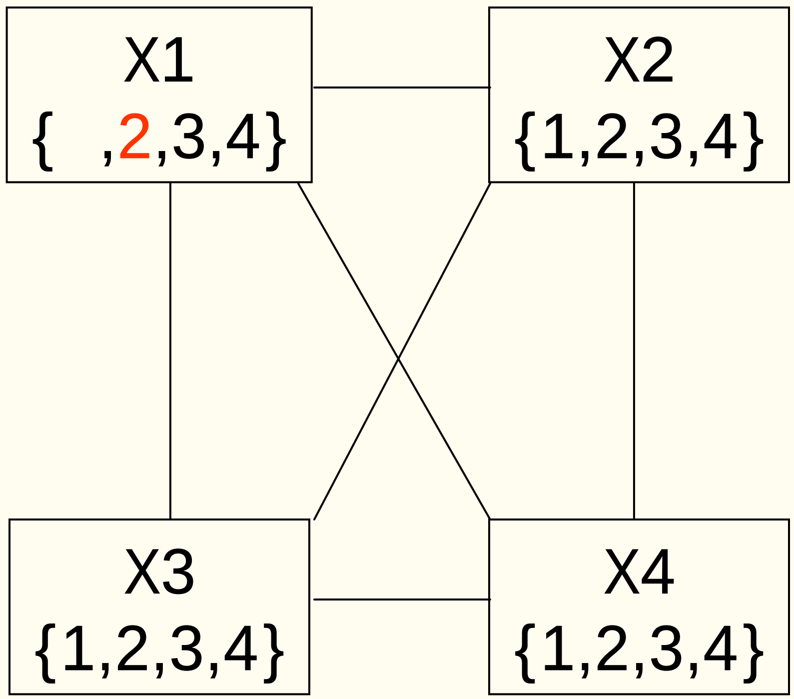
Example: 4-Queens Problem



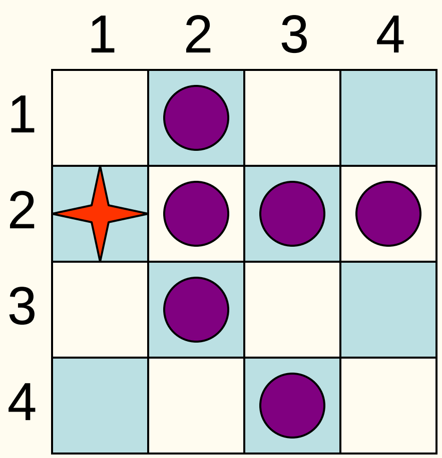


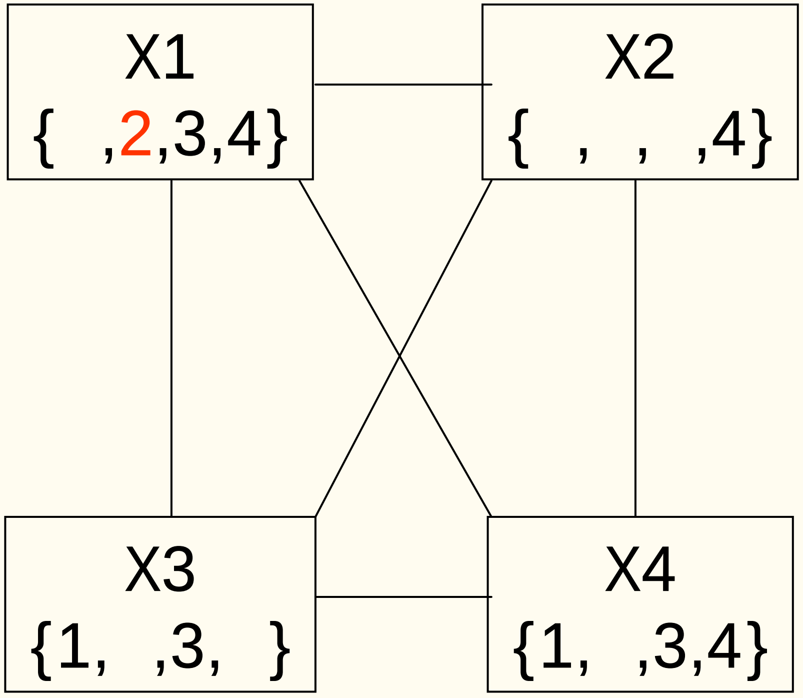
Example: 4-Queens Problem



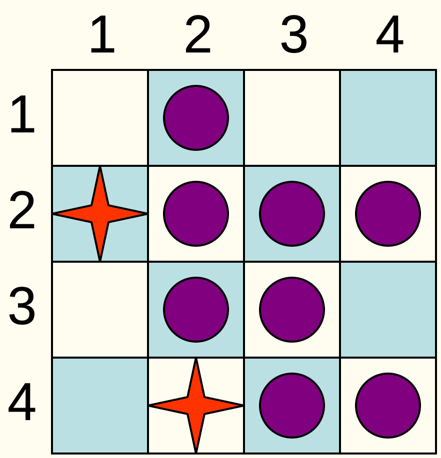


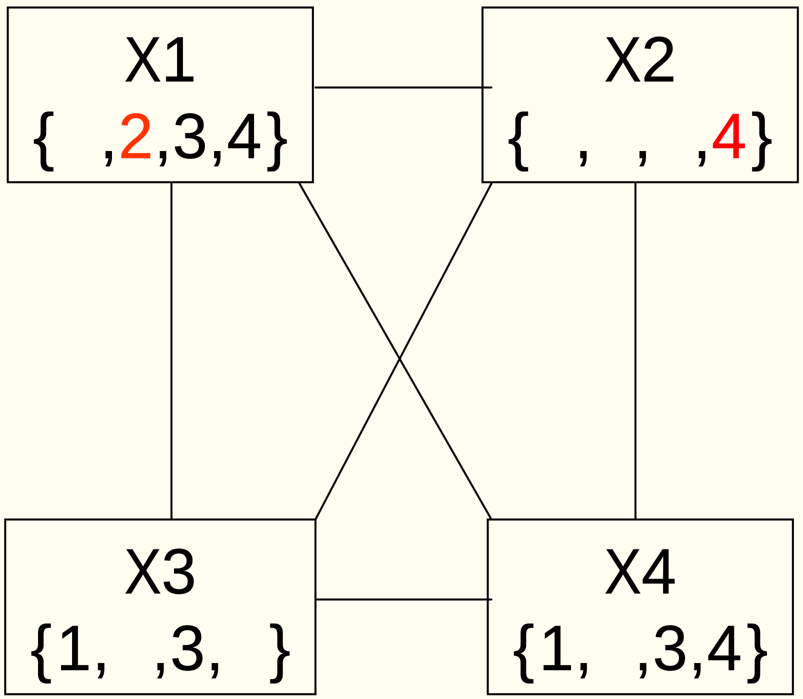
Example: 4-Queens Problem



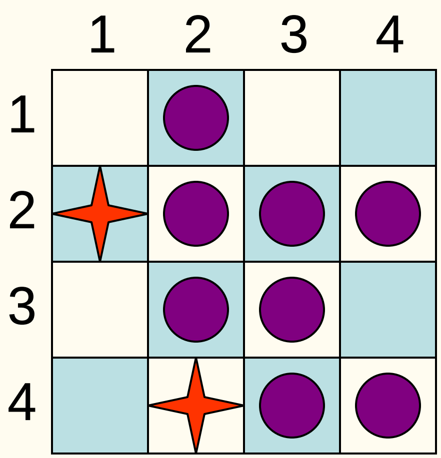


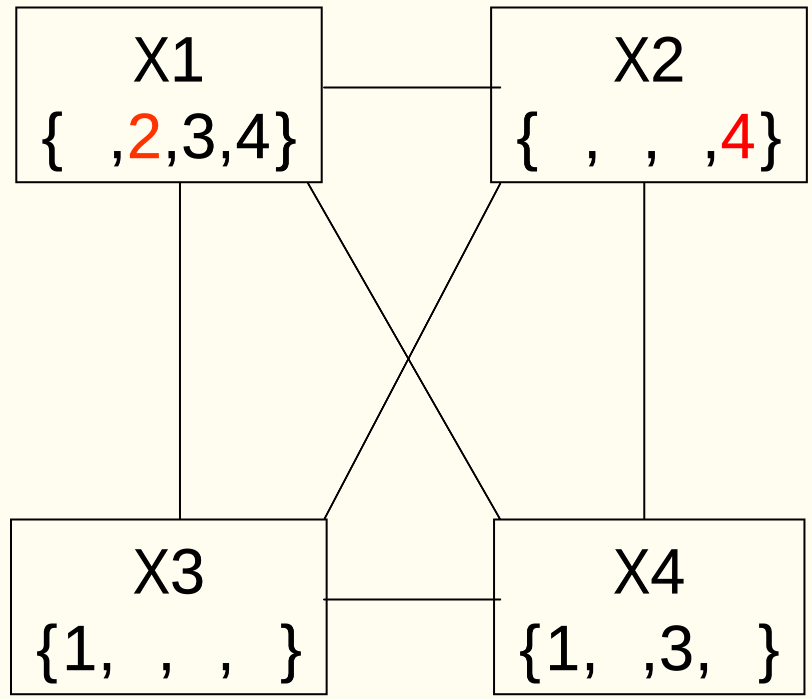
Example: 4-Queens Problem



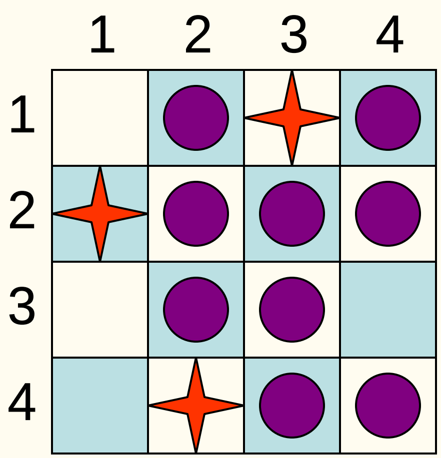


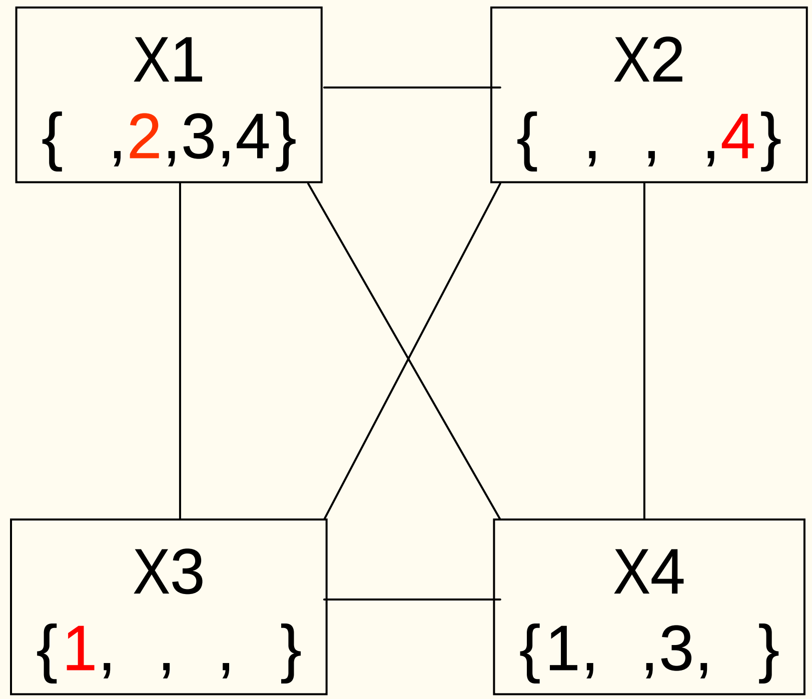
Example: 4-Queens Problem



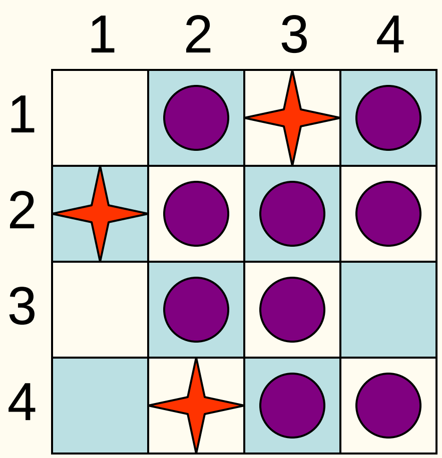


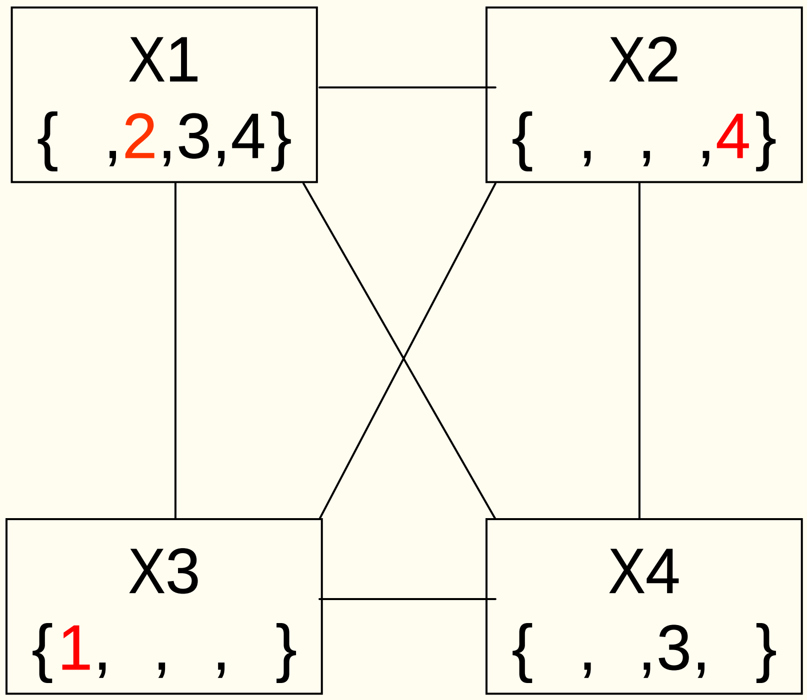
Example: 4-Queens Problem





Example: 4-Queens Problem





Example: 4-Queens Problem