Constraint Satisfaction Problems

T-622-ARTI

Artificial Intelligence, Spring 2023

Outline

- **Constraint Satisfaction Problems (CSPs):**
  - Postponing making difficult decisions until they become easy to make
- **Examples**
- **Backtracking search for CSPs**
- **Constraint propagation**
- **Arc consistency**
- **Exploiting the structure of CSPs**

What we will try to do …

- Search techniques make choices in an often arbitrary order. Often little information is available to make each of them.
- In many problems, the same states can be reached independent of the order in which choices are made (“commutative” actions).
- Can we solve such problems more efficiently by picking the order appropriately? Can we even avoid making any choice?

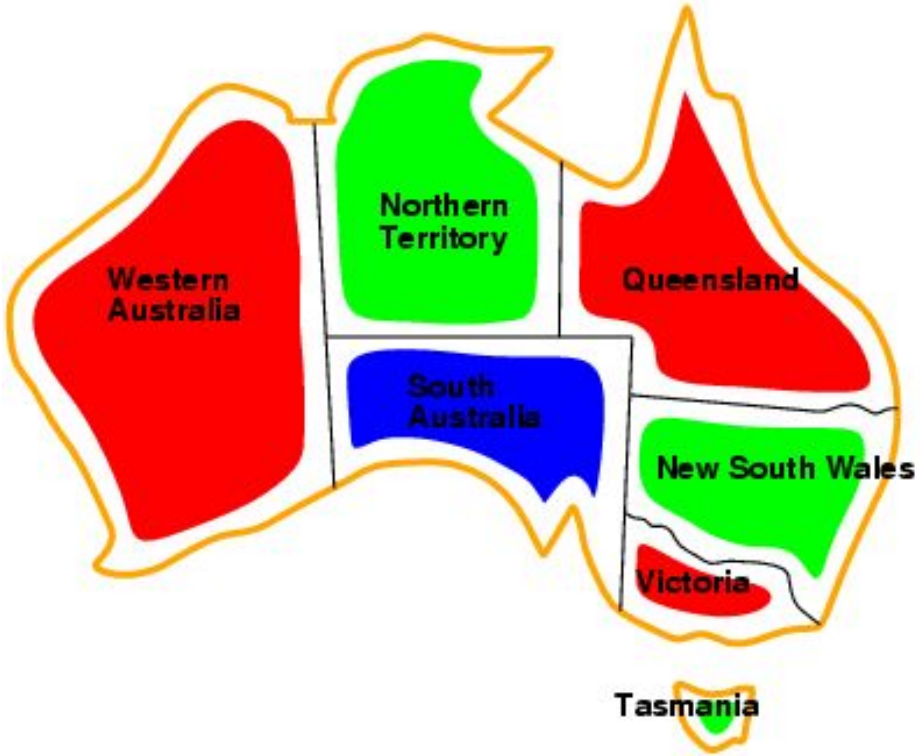
Constraint Satisfaction Problems (CSPs)

- **Standard search problem:**
  - State is a “black box” – any data structure that supports successor function, heuristic function, and goal test
- **CSP:**
  - State is defined by variables X_i with values from domain D_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

Example: Map coloring

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

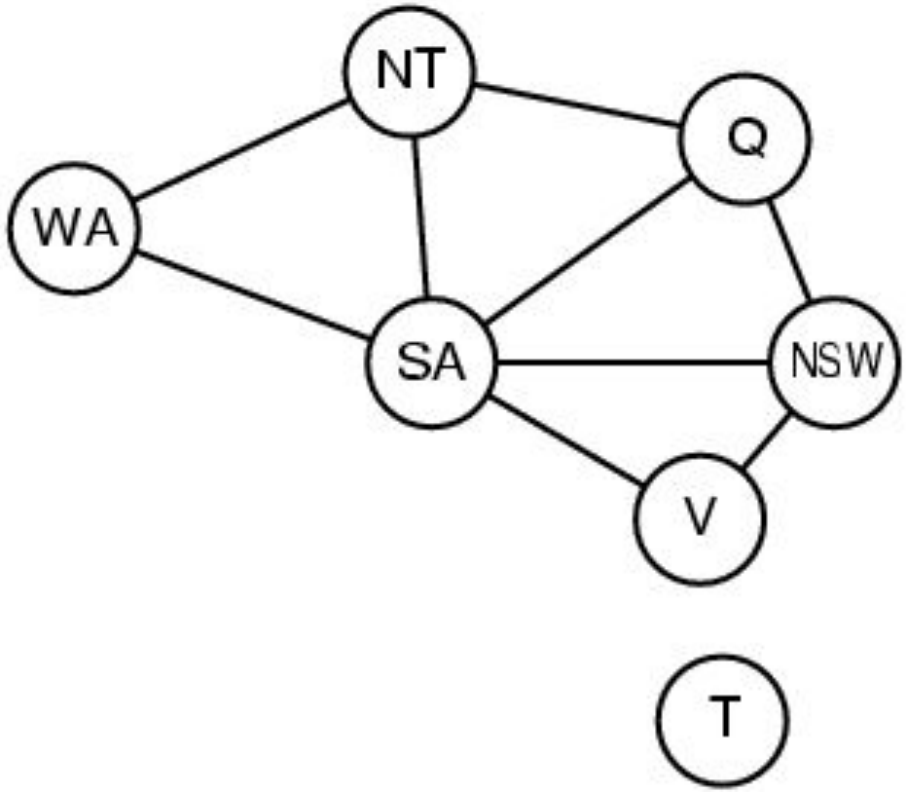
Example: Map coloring



- **Solutions are complete and consistent assignments:**
  - E.g. WA = red, NT = green,Q = red,NSW = green,V = red,SA = blue,T = green

Constraint Graph

- **Binary CSP:**
  - Each constraint relates two variables
- **Constraint graph:**
  - Nodes are variables, arcs are constraints



CSP - Definition

- Set of variables $\{X_1, X_2, \ldots, X_n\}$
- Each variable X_i has a domain D_i of possible values. Usually, D_i is finite
- Set of constraints $\{C_1, C_2, \ldots, C_p\}$
- Each constraint relates a subset of variables by specifying the valid combinations of their values
- **Goal:** Assign a value to every variable such that all constraints are satisfied
- A solution to a CSP is a complete assignment that satisfies all constraints

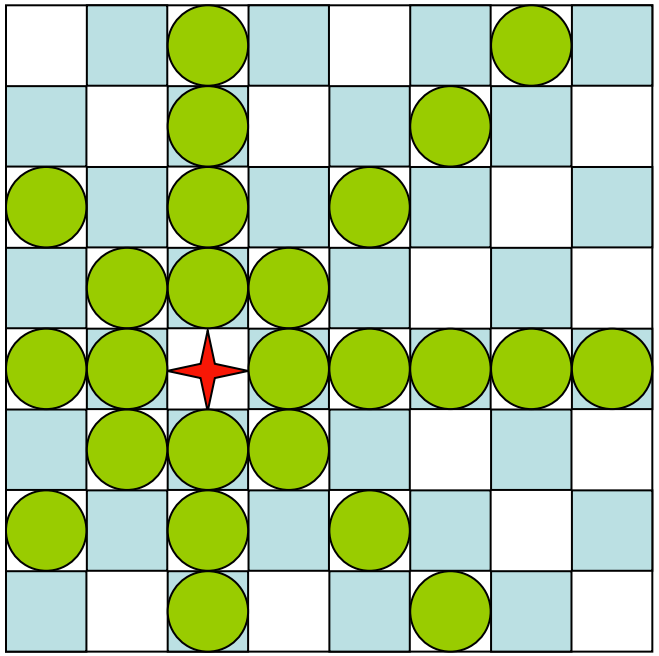
CSP Examples

- Map coloring / Image segmentation
- Crossword puzzle
- Sudoku
- Job scheduling
- Classroom assignment
- Formal verification of programs

8-Queen Problem

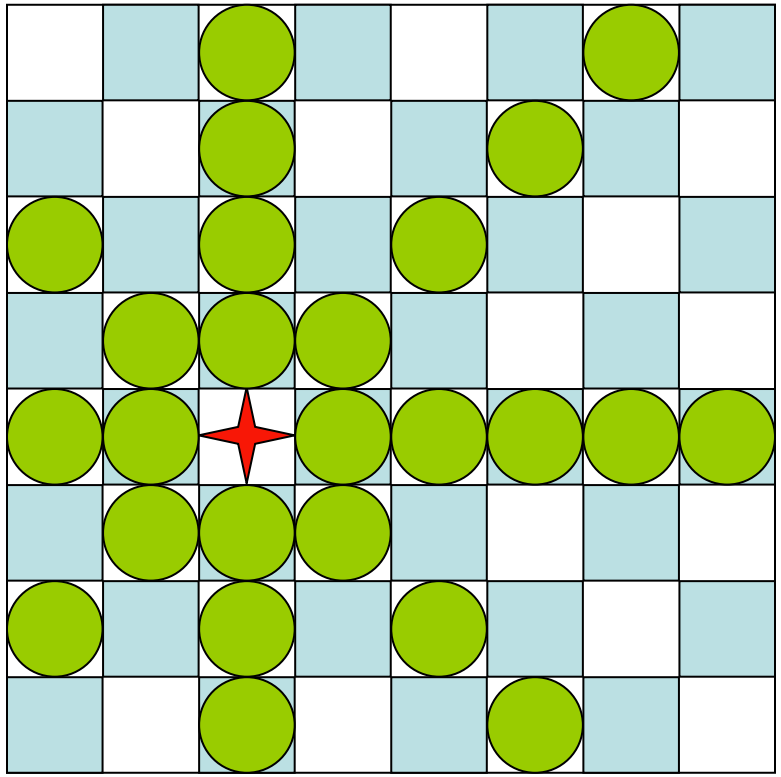
- Variables?
- Domains?
- Constraints?

Constraint Propagation



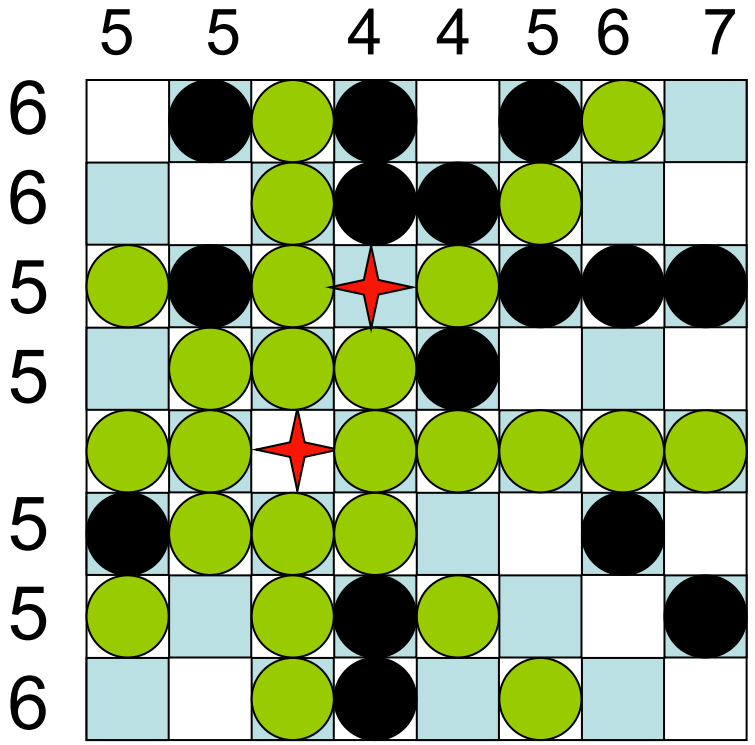
- Place a queen in a square
- Remove the attacked squares from future consideration

Heuristics

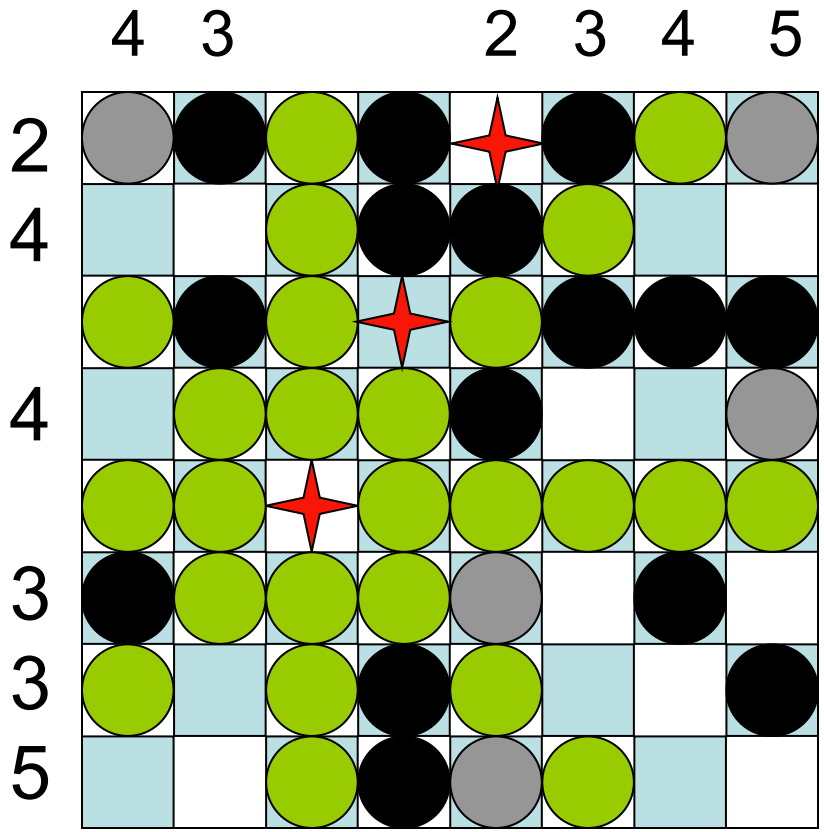


- Where to place the next queen?

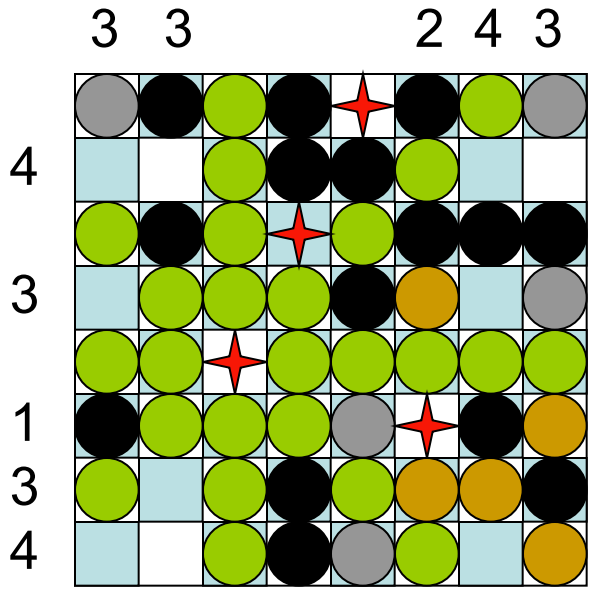
Heuristics

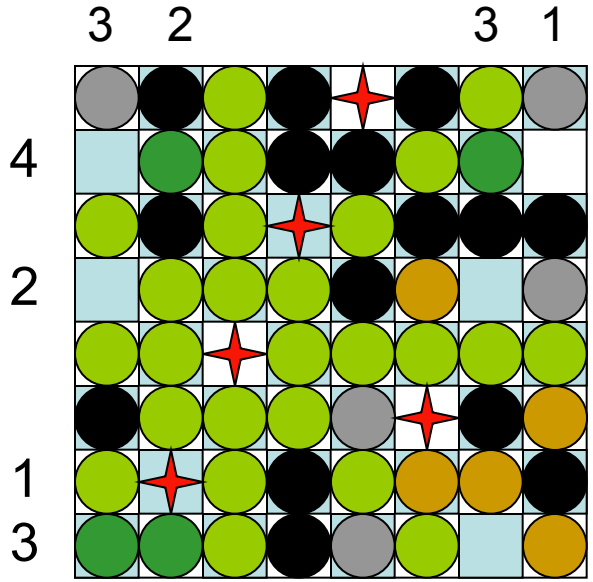


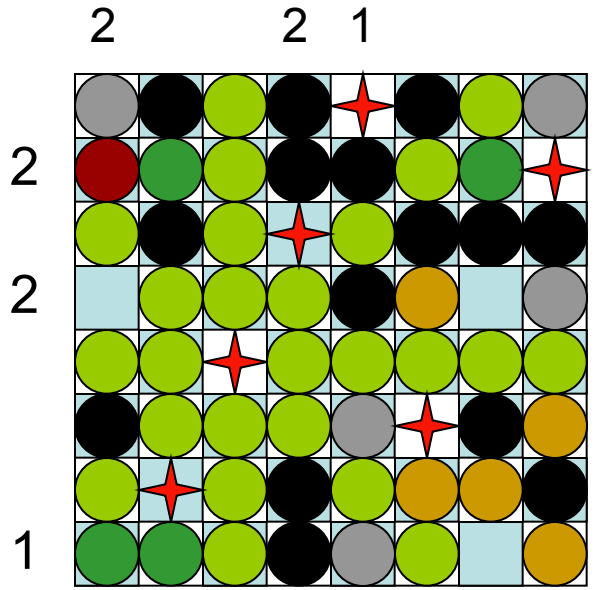
- Count the number of non-attacked squares in every row and column
- Place a queen in a row or column with minimum number
- Remove the attacked squares from future consideration

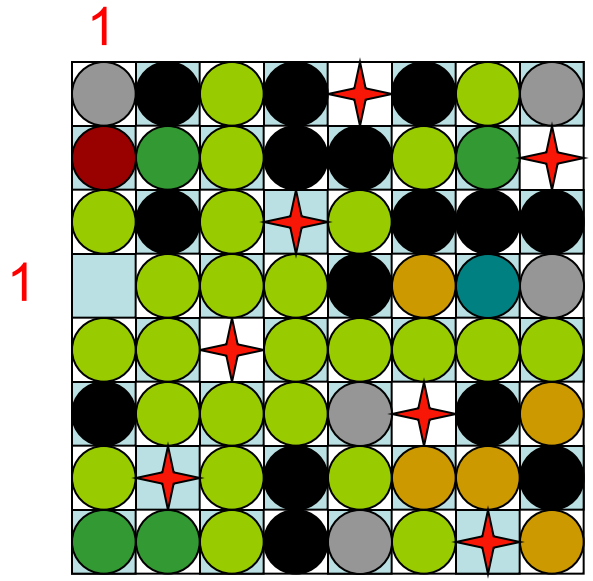


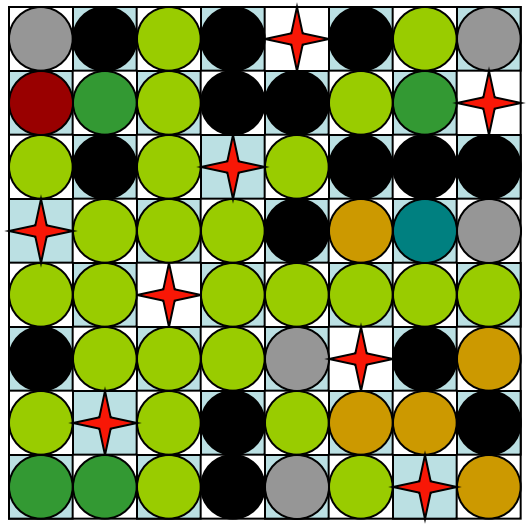
- Repeat


3 3 2 4 3
4
3
1
3
4

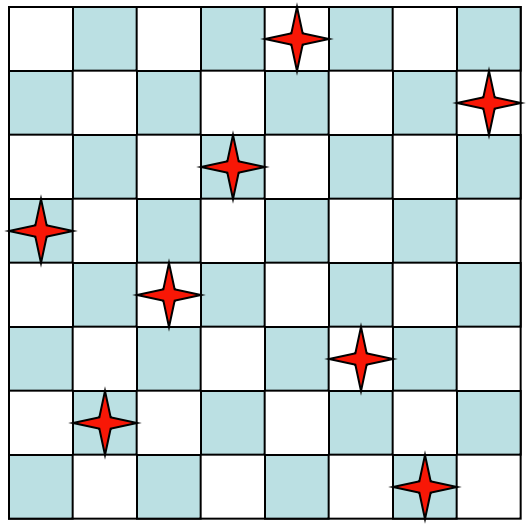

3 2 3 1
4
2
1
3


2
2
1
2
2
1


1
1

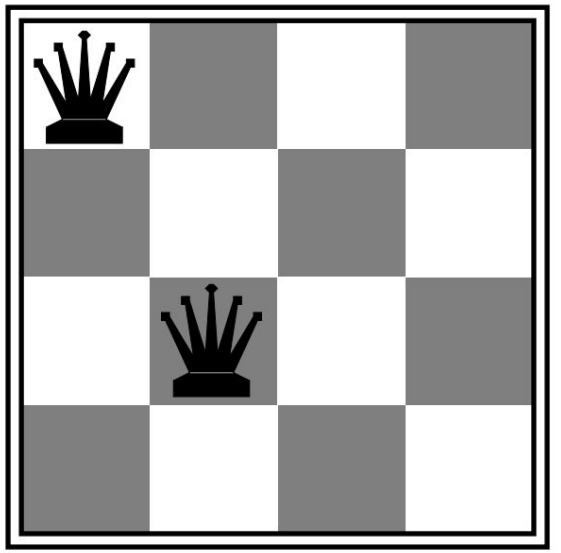


Constraint Propagation



8-Queen Problem

- 8 variables $Q_i, i = 1\ to\ 8$
- The domain of each variable
  - $\{1, 2, \ldots, 8\}$
- Constraints:
  - $Q_i \neq Q_j$ (cannot be in the same row)
  - $|Q_i - Q_j| \neq |i - j|$ (or same diagonal)



- All constraints are binary

Finite vs. Infinite CSP

- **Finite CSP:**
  - Each variable has a finite domain of values
- **Infinite CSP:**
  - Some or all variables have an infinite domain

- We will only consider finite CSP

Standard Search Formulation

- Start with the straightforward, dumb approach, then fix it
- States are defined by the values assigned so far
  - 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 (not fixable!)
  - Goal test: the current assignment is complete

1. This is the same for all CSPs
2. Every solution appears at depth n with n variables ⇒ use depth-first search
3. $b = (n - l)d$ at depth l, hence $n!\, d^n$ leaves

Backtracking Search

- Variable assignments are commutative, 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 node

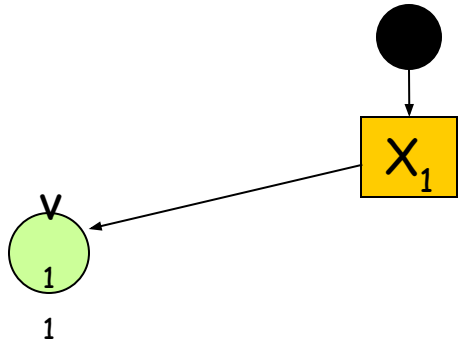
  ⇒ $b = d$ and there are d^n leaves

- 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 ≈ 25

Backtracking Search
(3 variables)

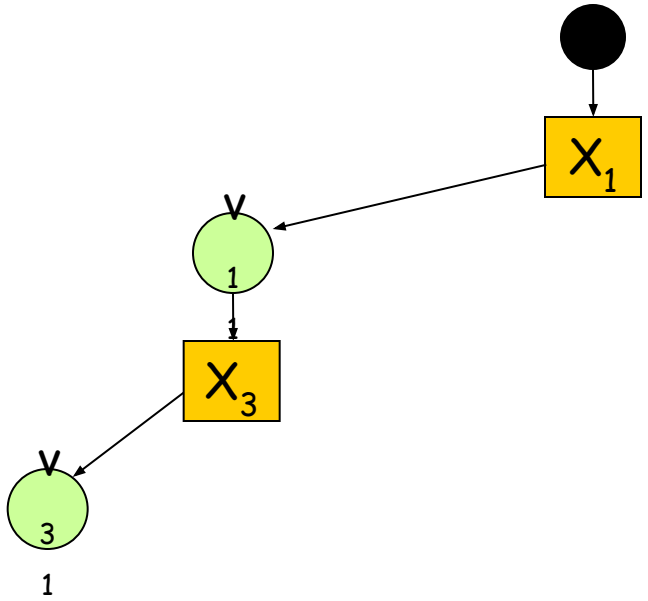
Assignment = {}

Backtracking Search
(3 variables)



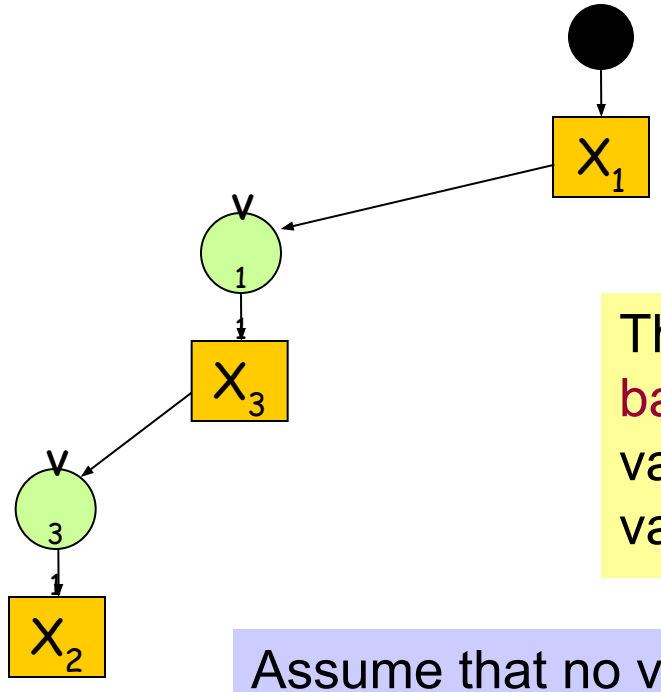
Assignment = $\{(X_1, v_{11})\}$

Backtracking Search
(3 variables)



Assignment = $\{(X_1,v_{11}), (X_3,v_{31})\}$

Backtracking Search
(3 variables)

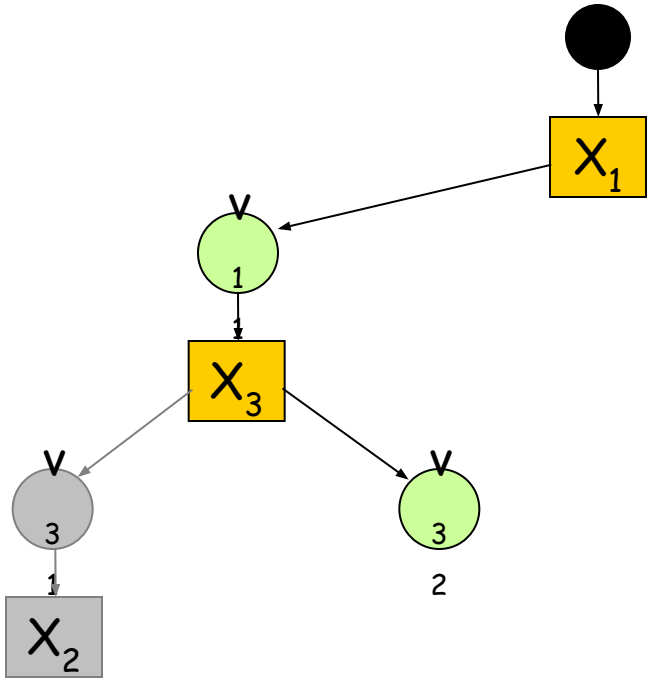


Then, the search algorithm backtracks to the previous variable (X_3) and tries another value

Assume that no value of X_2 leads to a valid assignment

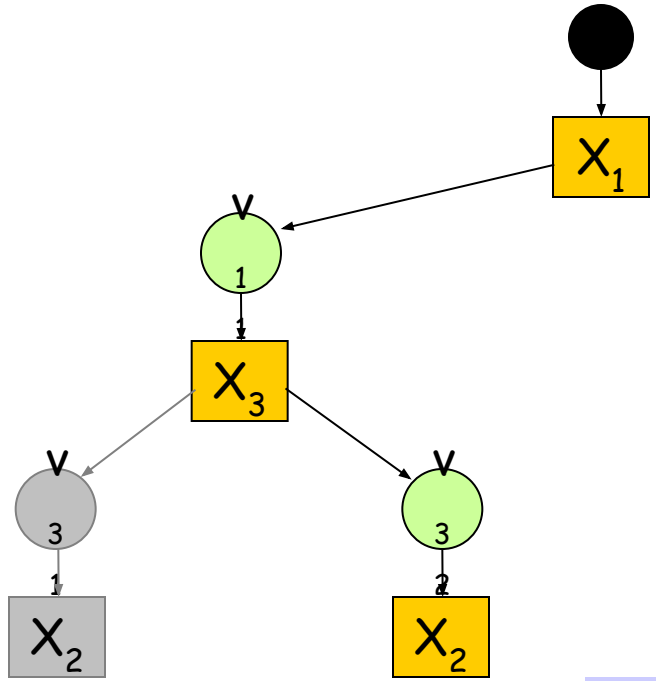
Assignment = $\{(X_1,v_{11}), (X_3,v_{31})\}$

Backtracking Search
(3 variables)



Assignment = $\{(X_1,v_{11}), (X_3,v_{32})\}$

Backtracking Search
(3 variables)

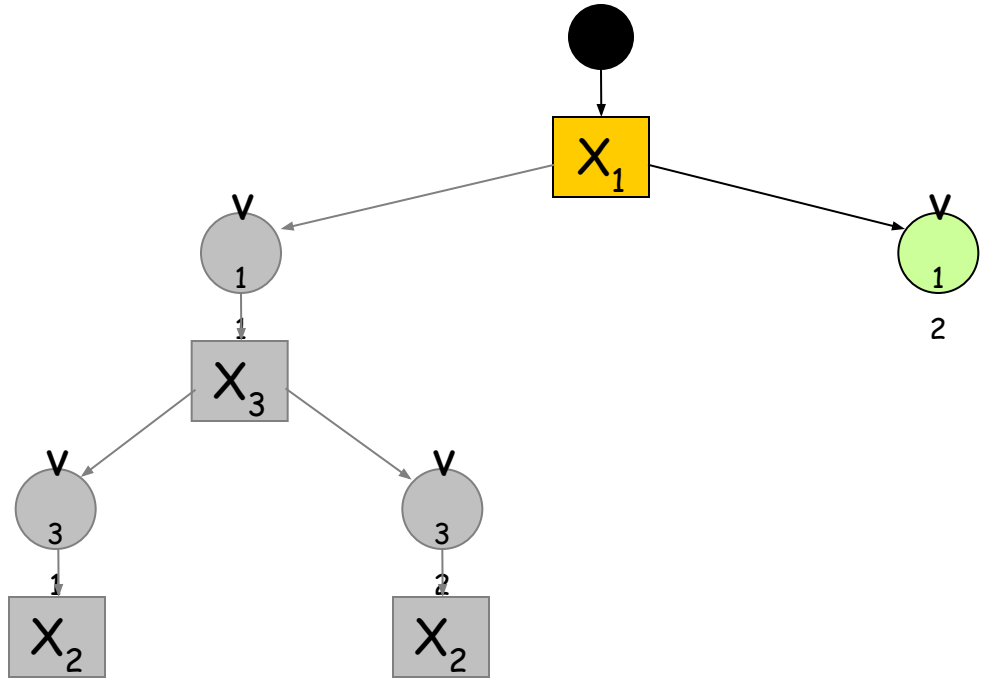


The search algorithm backtracks to the previous variable (X_3) and tries another value. But assume that X_3 has only two possible values. The algorithm backtracks to X_1

Assume again that no value of X_2 leads to a valid assignment

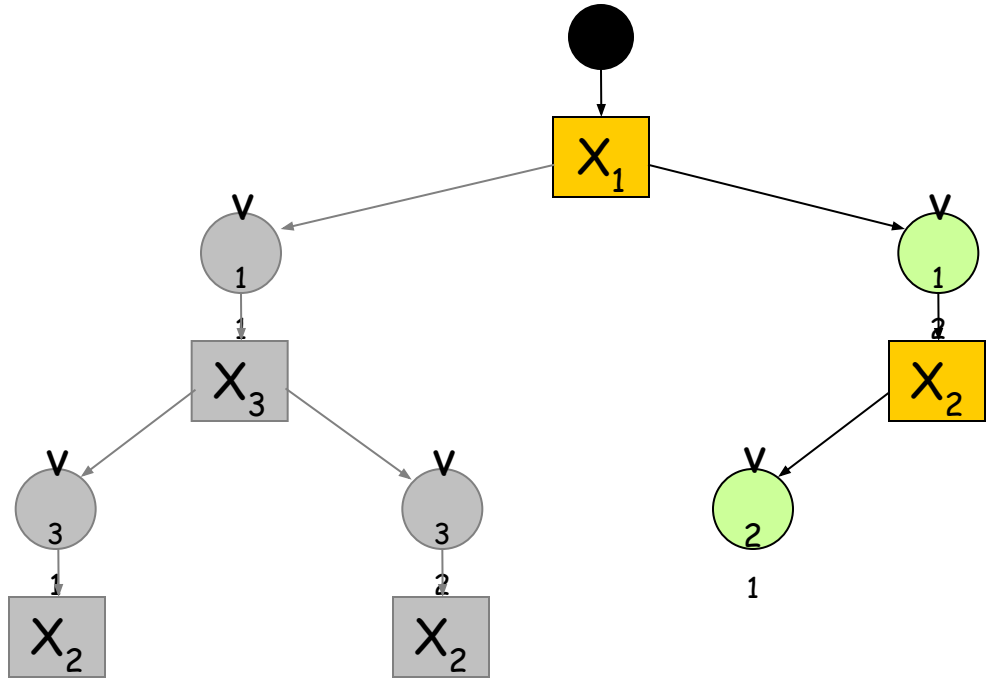
Assignment = $\{(X_1,v_{11}), (X_3,v_{32})\}$

Backtracking Search
(3 variables)



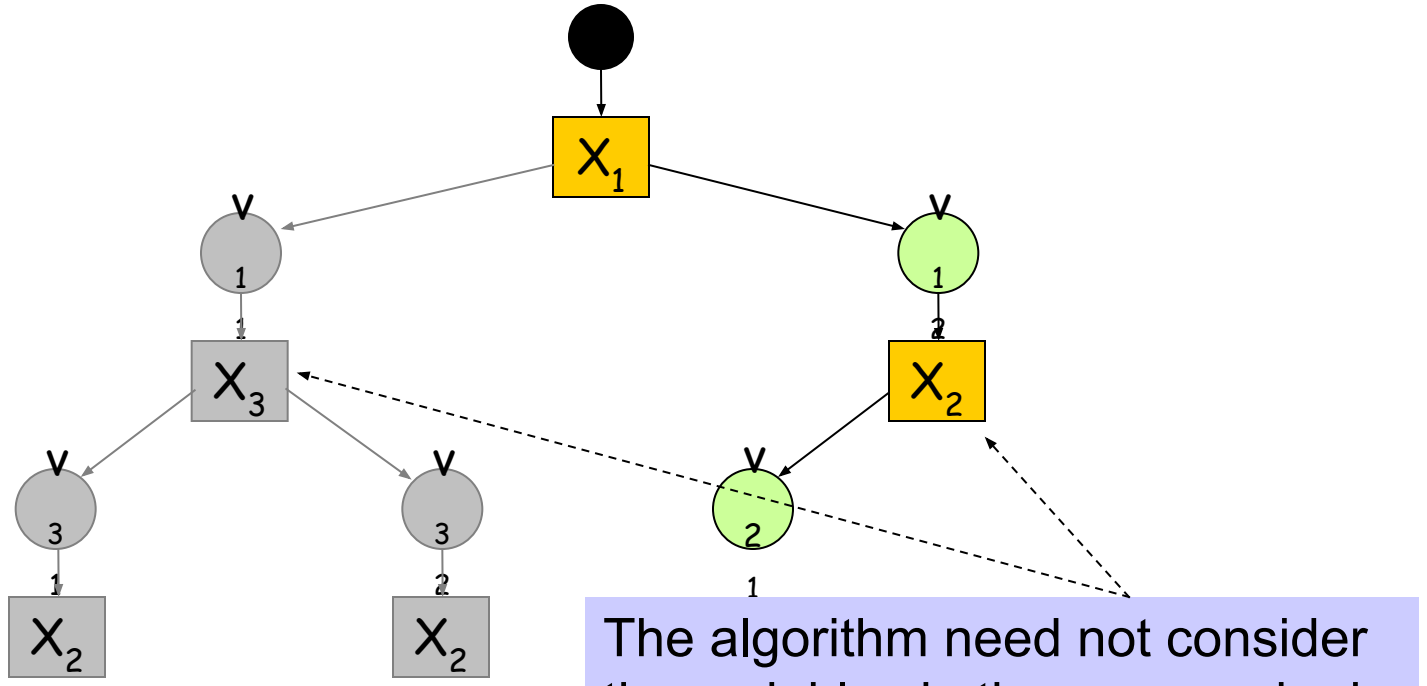
Assignment = $\{(X_1, v_{12})\}$

Backtracking Search
(3 variables)



Assignment = $\{(X_1,v_{12}), (X_2,v_{21})\}$

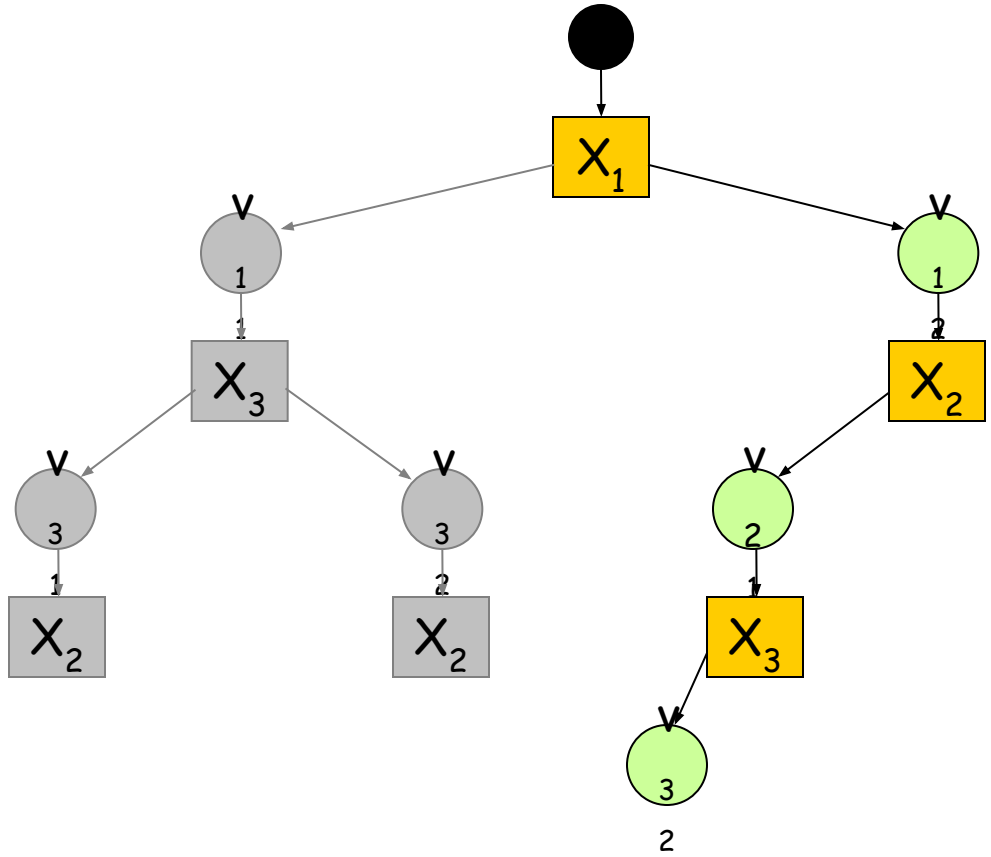
Backtracking Search
(3 variables)



The algorithm need not consider the variables in the same order in this sub-tree as in the other

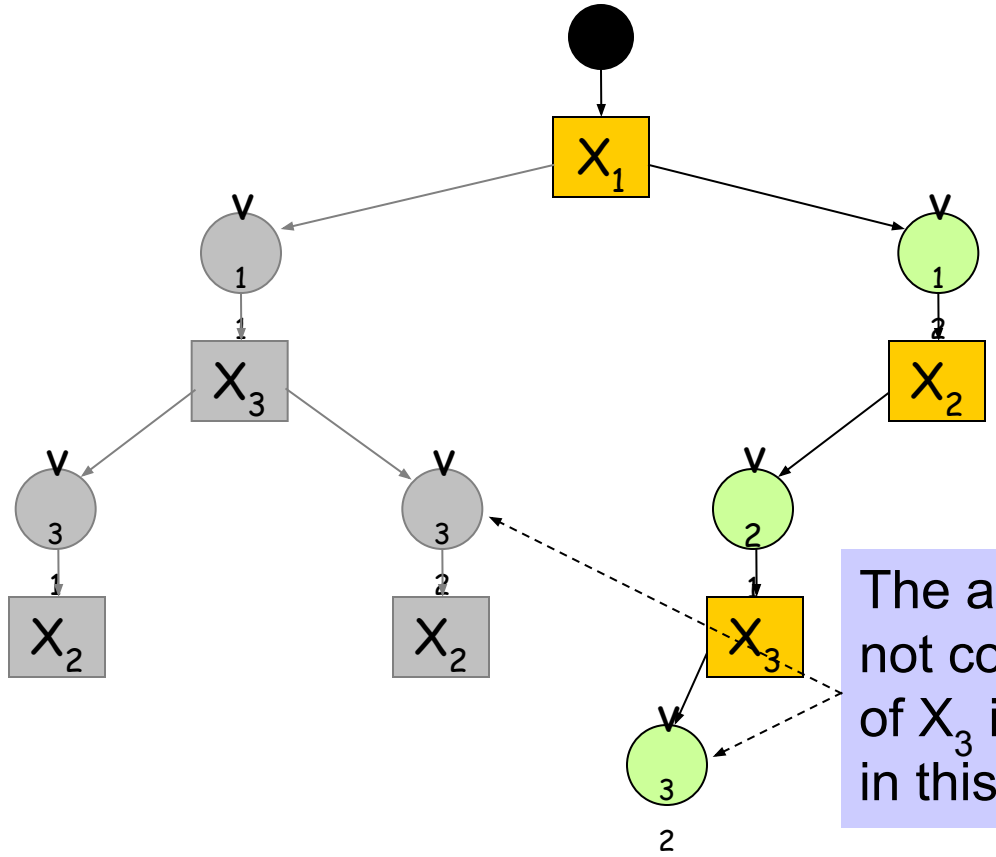
Assignment = $\{(X_1,v_{12}), (X_2,v_{21})\}$

Backtracking Search
(3 variables)



Assignment = $\{(X_1,v_{12}), (X_2,v_{21}), (X_3,v_{32})\}$

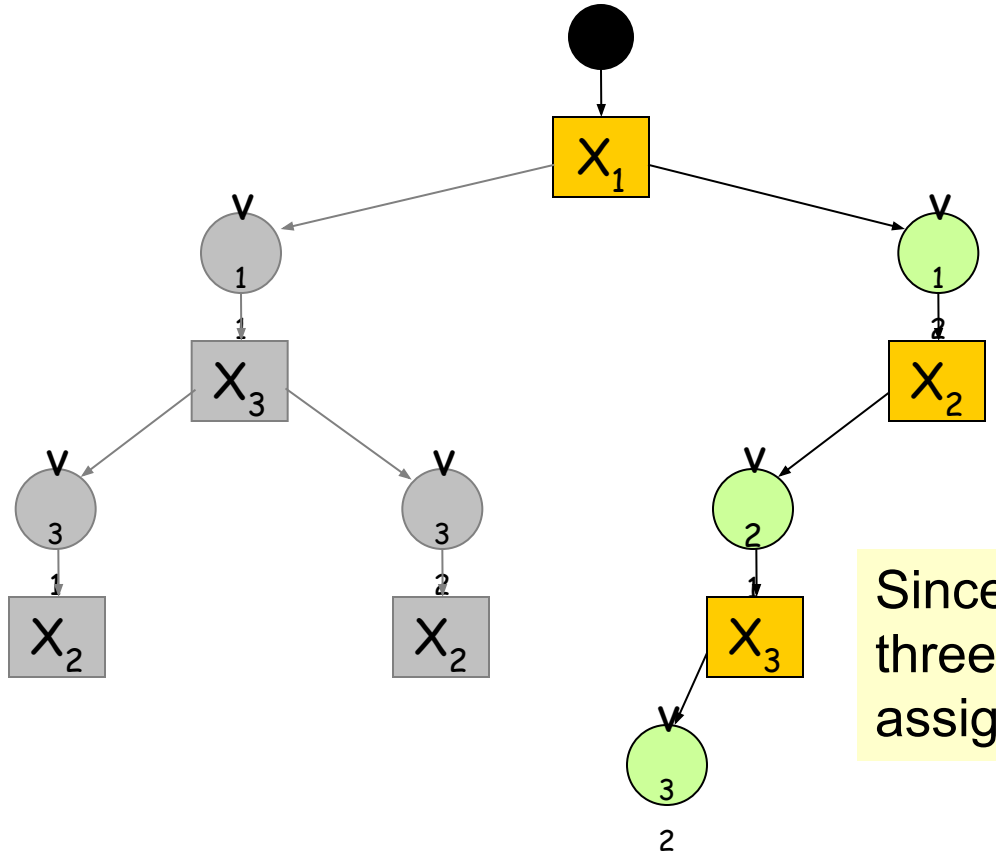
Backtracking Search
(3 variables)



The algorithm need not consider the values of X_3 in the same order in this sub-tree

Assignment = $\{(X_1,v_{12}), (X_2,v_{21}), (X_3,v_{32})\}$

Backtracking Search
(3 variables)



Since there are only three variables, the assignment is complete

Assignment = $\{(X_1,v_{12}), (X_2,v_{21}), (X_3,v_{32})\}$

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
```

CRF Demo

$A = \{1, 2, 3\}$

$B = \{1, 2, 3\}$

$C = \{1, 2, 3\}$

$A > B$

$B \neq C$

$A \neq C$

Improving Backtracking Efficiency

General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next?
2. In what order should its values be tried?
3. Can we detect inevitable failure early?
4. Can we take advantage of problem structure?

Critical Questions for the Efficiency of CSP-Backtracking

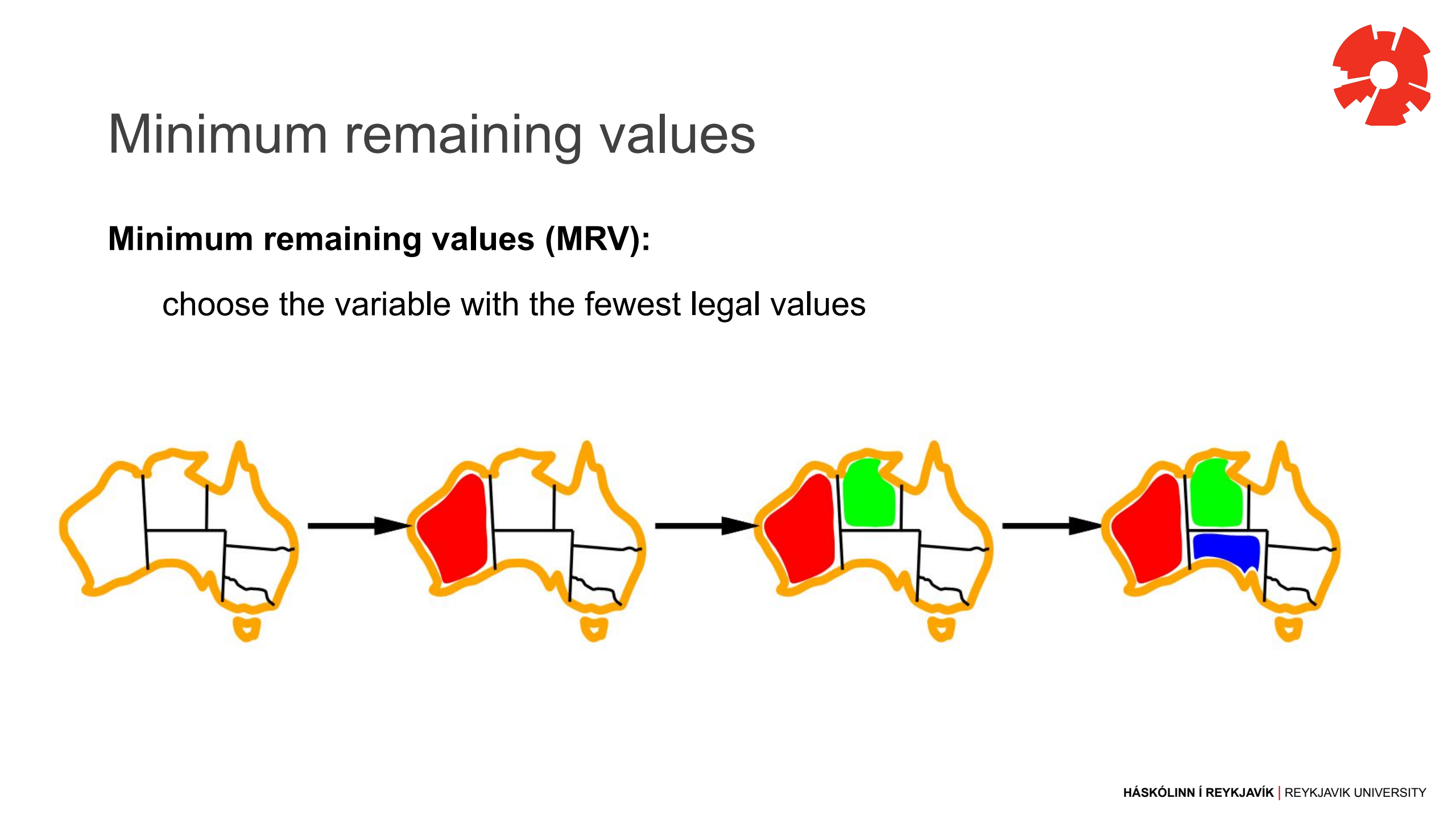
1. Which variable X should be assigned a value next?

The current assignment may not lead to any solution, but the algorithm does not know it yet. Selecting the right variable X may help discover the contradiction more quickly

2. In which order should X's values be assigned?

The current assignment may be part of a solution. Selecting the right value to assign to X may help discover this solution more quickly

More on these questions very soon ...



Minimum remaining values

Minimum remaining values (MRV):

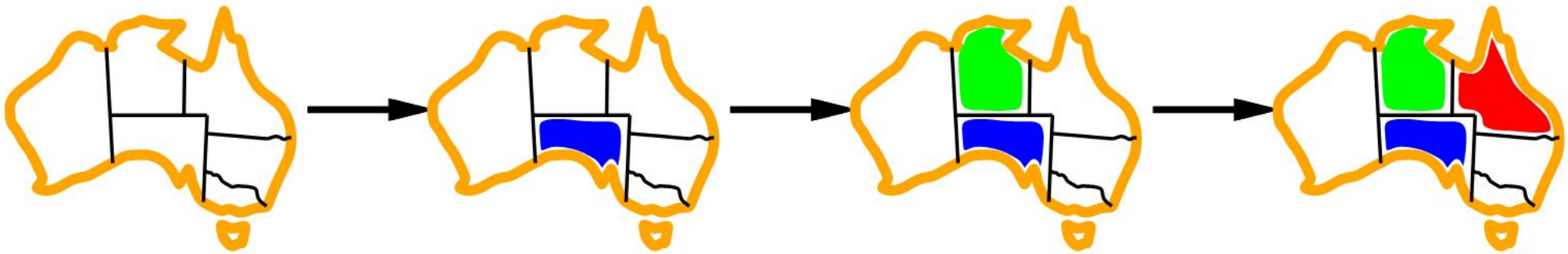
choose the variable with the fewest legal values

Degree heuristic

Tie-breaker among MRV variables

Degree heuristic:

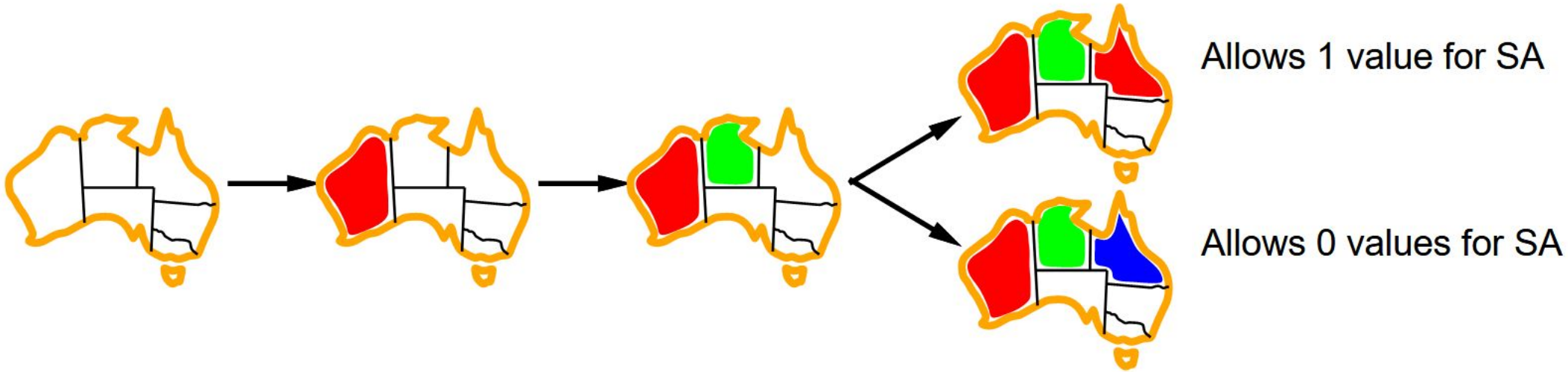
choose the variable with the most constraints on remaining variables



Least constraining value

Given a variable, choose the least constraining value:

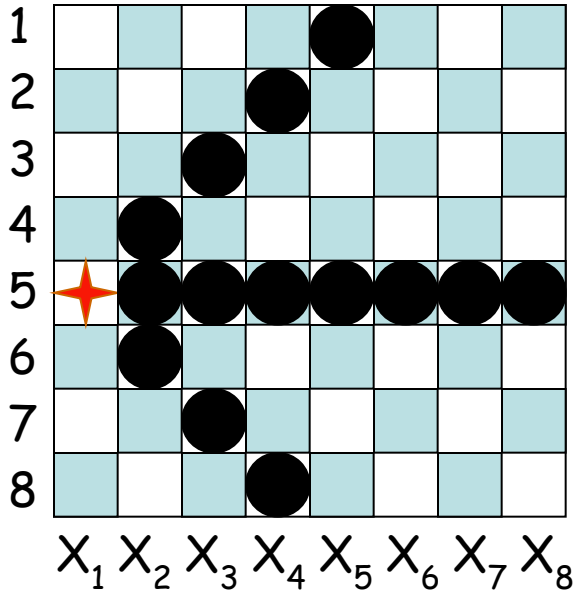
the one that rules out the fewest values in the remaining variables



Combining these heuristics makes 1000 queens feasible

Forward Checking

A simple constraint-propagation technique:



Assigning the value 5 to X_1 leads to removing values from the domains of X_2, X_3, ..., X_8

Forward checking

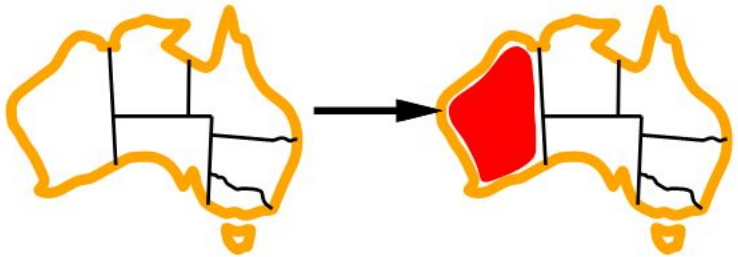
Idea: **Keep track of remaining legal values for unassigned variables**

Terminate search when any variable has no legal values

Forward checking

Idea: **Keep track of remaining legal values for unassigned variables**

Terminate search when any variable has no legal values



Forward checking

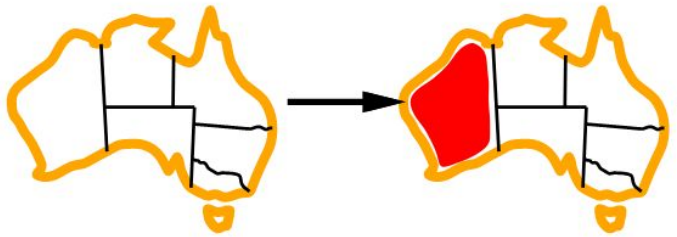
Idea: **Keep track of remaining legal values for unassigned variables**

Terminate search when any variable has no legal values

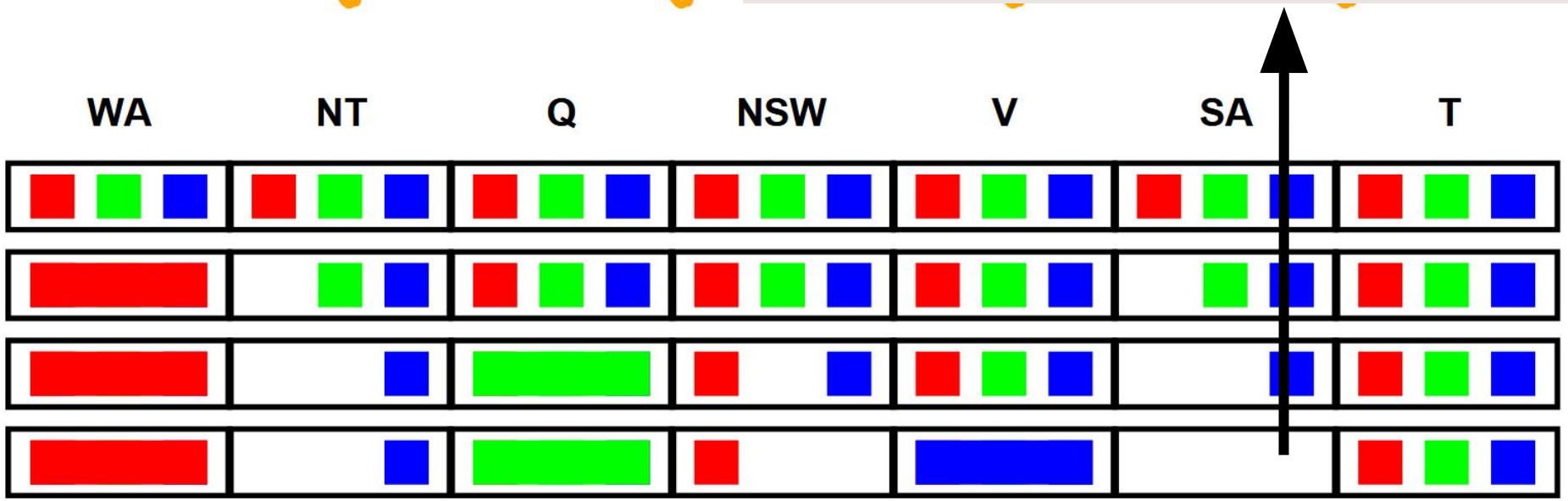
Forward checking

Idea: Keep track of remaining legal values for unassigned variables

Terminate search when any variable has no legal values



Empty set: the current assignment {(WA □ R), (Q □ G), (V □ B)} does not lead to a solution



Backtracking search with forward checking

```
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
            csp.domains ← forwardChecking(csp.domains, X, v, A)
            if no empty domain {
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result }
            remove {var = value} from assignment
    return failure
```

1. Which variable X_i should be assigned a value next?
Most-constrained-variable heuristic
Most-constraining-variable heuristic

2. In which order should its values be assigned?
Least-constraining-value heuristic

These heuristics can be quite confusing

Keep in mind that **all** variables must eventually get a value, while only **one** value from a domain must be assigned to each variable

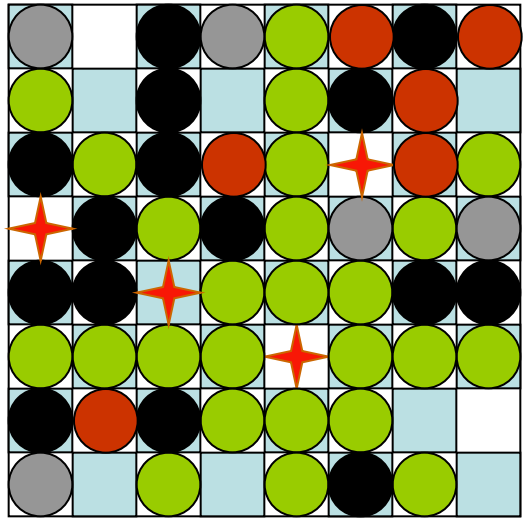
Most-Constrained-Variable Heuristic

1. Which variable X_i should be assigned a value next?

    Select the variable with the smallest remaining domain

    Rationale: Minimize the branching factor

8-Queens

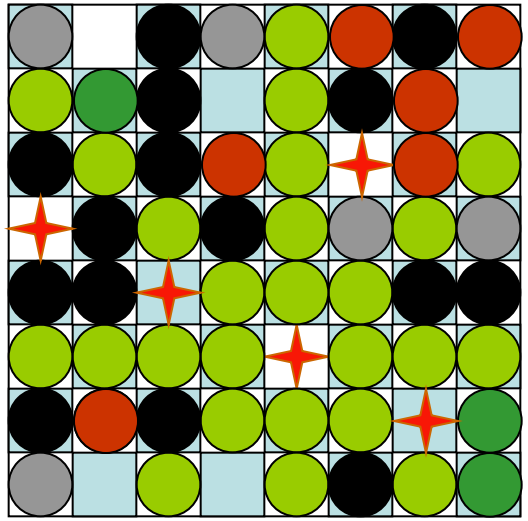


Forward checking

New assignment

Numbers
of values for
each un-assigned
variable

8-Queens



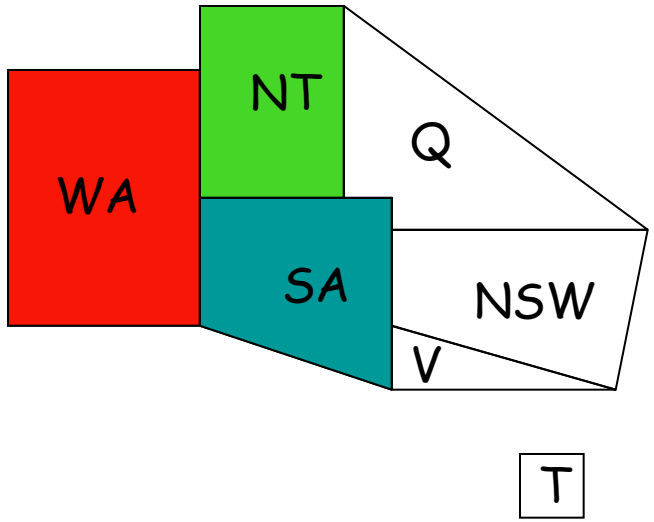
3 2 1 3

Forward checking

New assignment

New numbers of values for each un-assigned variable

Map Coloring



- SA's remaining domain has size 1 (value Blue remaining)
- Q's remaining domain has size 2
- NSW's, V's, and T's remaining domains have size 3

 Select SA

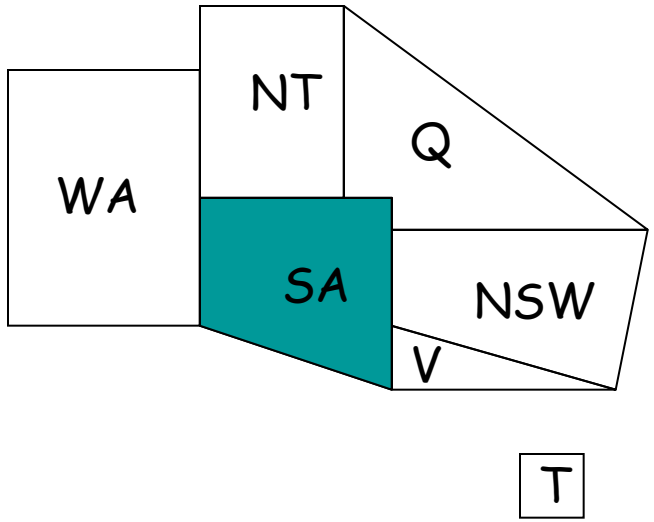
Most-Constraining-Variable Heuristic

1. Which variable X_i should be assigned a value next?

   Among the variables with the smallest remaining domains (ties with respect to the most-constrained-variable heuristic), select the one that appears in the largest number of constraints on variables not in the current assignment

   Rationale: Increase future elimination of values, to reduce future branching factors

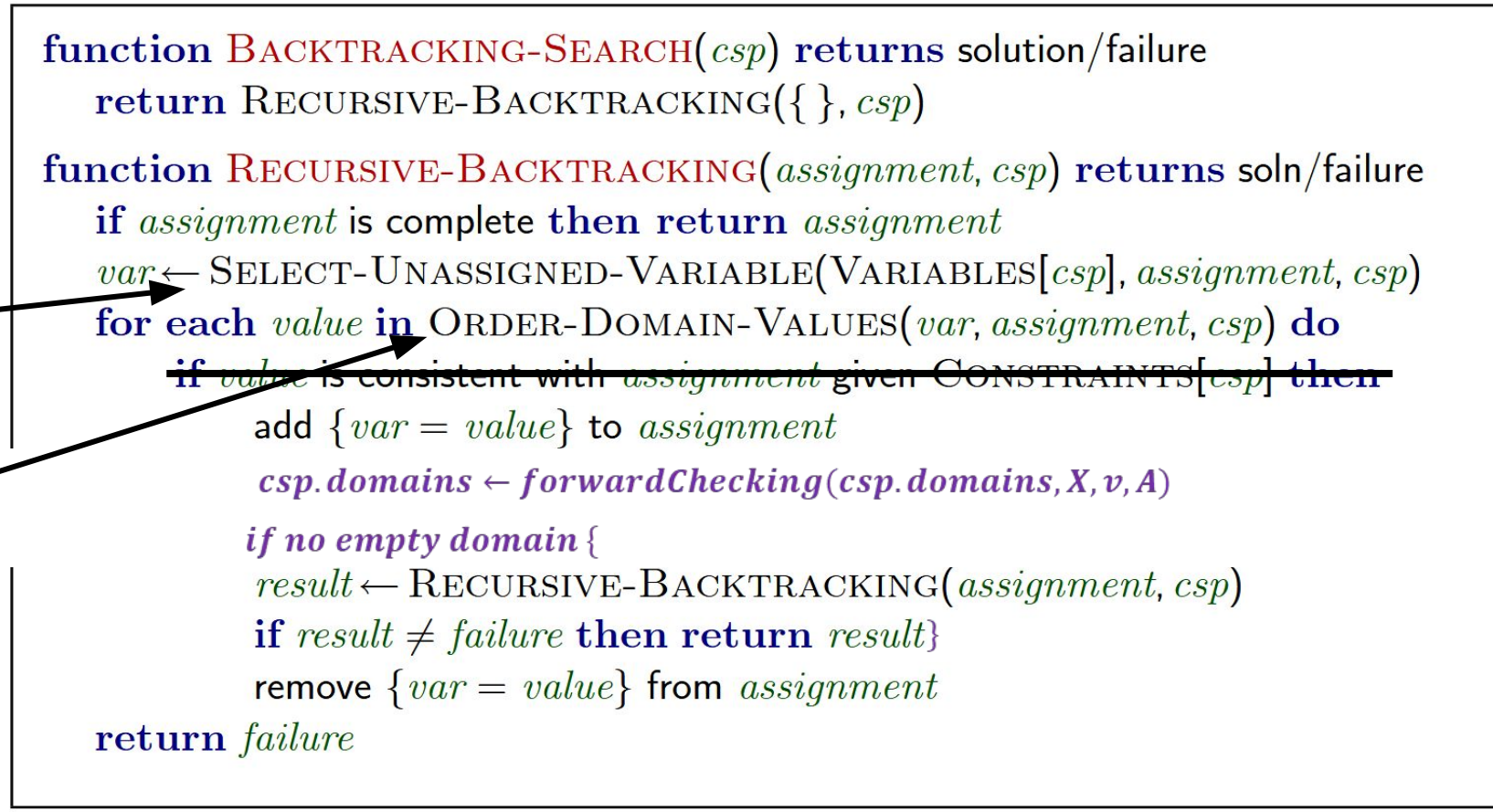
Map Coloring



- Before any value has been assigned, all variables have a domain of size 3, but SA is involved in more constraints (5) than any other variable

 Select SA and assign a value to it (e.g., Blue)

Backtracking search with forward checking and heuristics



1) Most-constrained-variable heuristic
2) Most-constraining-variable heuristic

3) Least-constraining-value heuristic

1) Select the variable with the smallest remaining domain
2) Select the variable that appears in the largest number of constraints on variables not in the current assignment

Least-Constraining-Value Heuristic

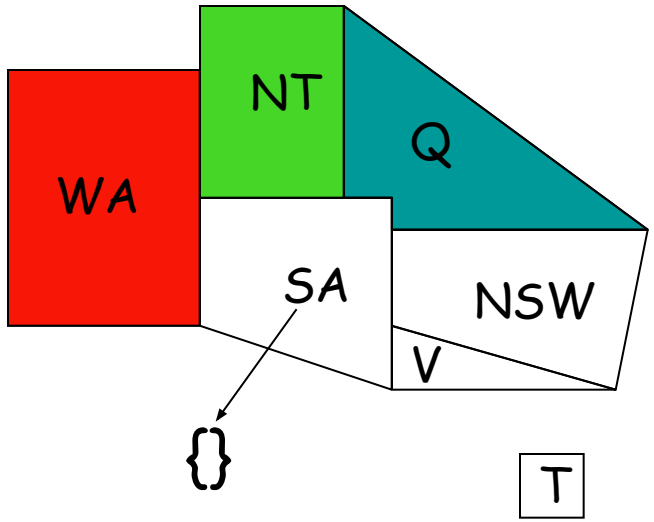
2. In which order should X's values be assigned?

   Select the value of X that removes the smallest number of values from the domains of those variables which are not in the current assignment

   Rationale: Since only one value will eventually be assigned to X, pick the least-constraining value first, since it is the most likely not to lead to an invalid assignment

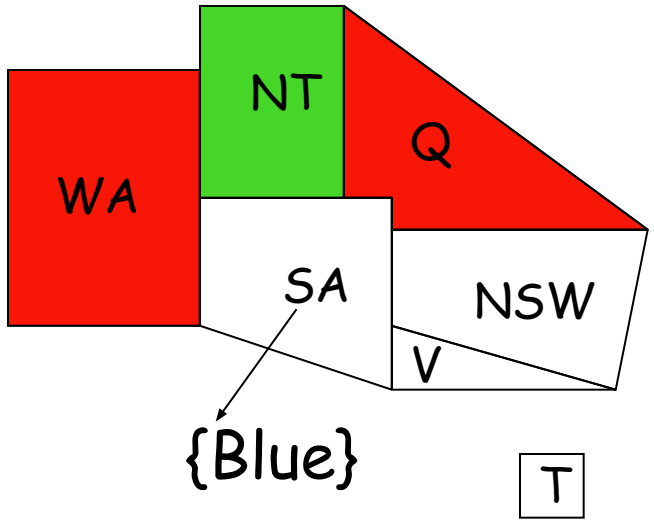
   Note: Using this heuristic requires performing a forward-checking step for every value, not just for the selected value

Map Coloring



- Q's domain has two remaining values: Blue and Red
- Assigning Blue to Q would leave 0 value for SA, while assigning Red would leave 1 value

Map Coloring



- Q's domain has two remaining values: Blue and Red
- Assigning Blue to Q would leave 0 value for SA, while assigning Red would leave 1 value

 So, assign Red to Q

Backtracking search with forward checking and heuristics

1) Most-constrained-variable heuristic
2) Most-constraining-variable heuristic

3) Least-constraining-value heuristic

```
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
            csp.domains ← forwardChecking(csp.domains, X, v, A)
            if no empty domain {
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result}
            remove {var = value} from assignment
    return failure
```

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!

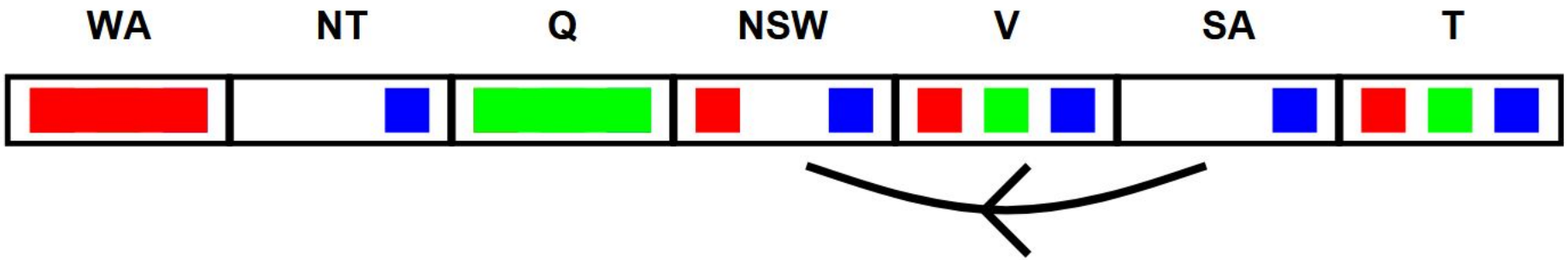
Constraint propagation repeatedly enforces constraints locally

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

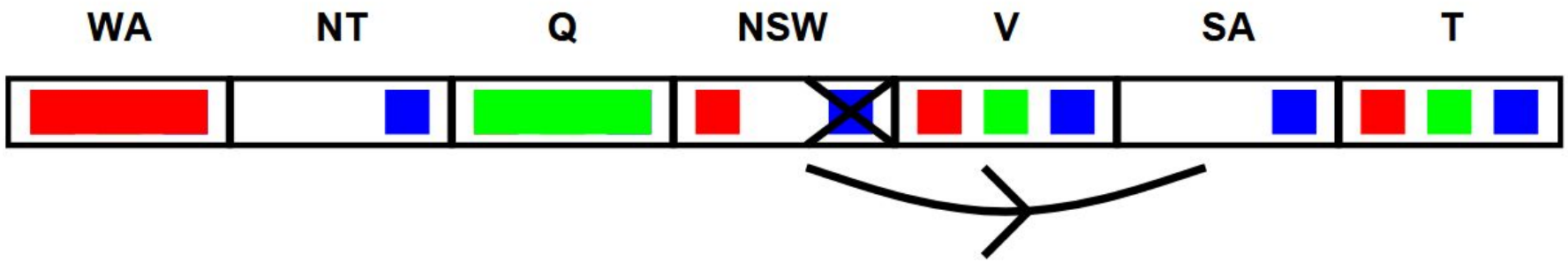


Arc consistency

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

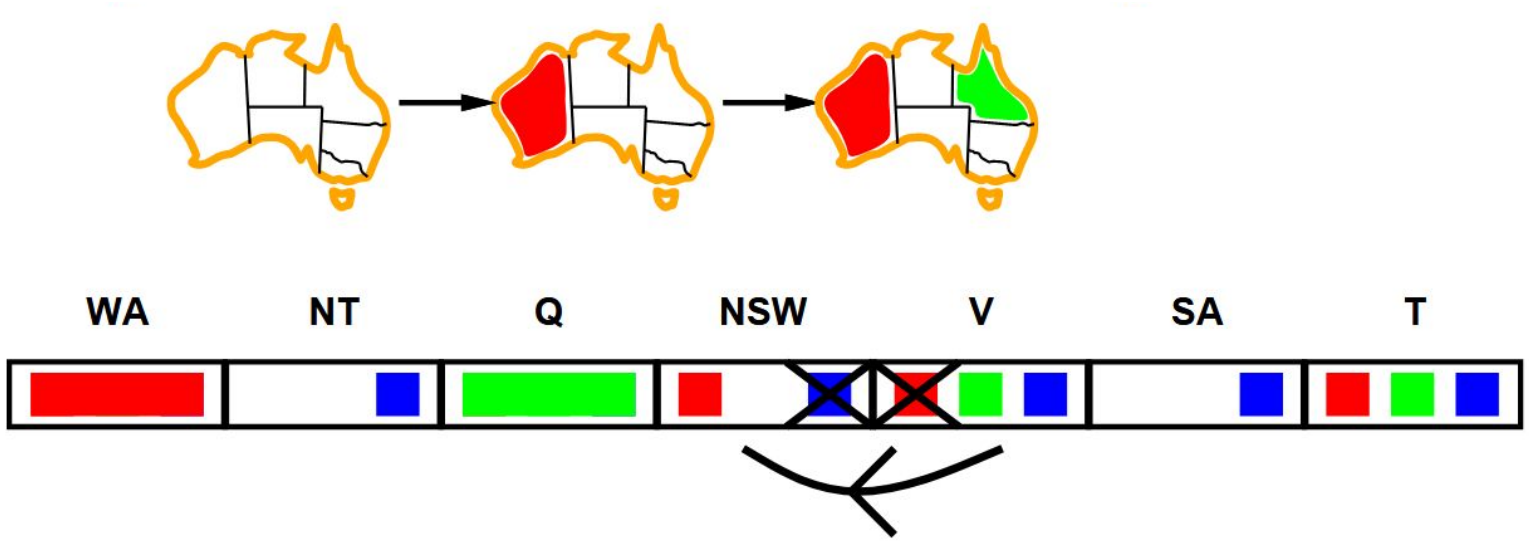


Arc consistency

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



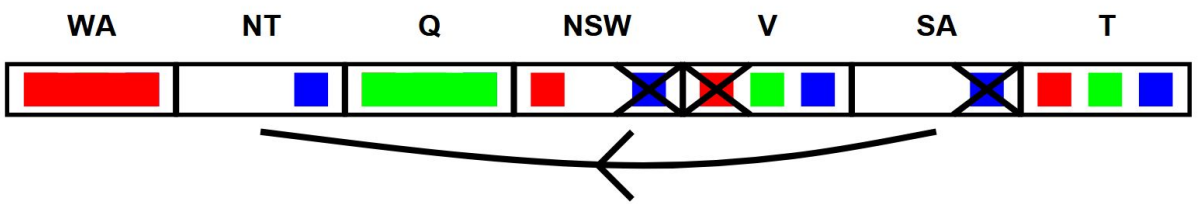
If X loses a value, neighbors of X need to be rechecked

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

$A = \{1, 2, 3\}$

$B = \{1, 2, 3\}$

$C = \{1, 2, 3\}$

$A > B$

$B = C$

Arc consistency algorithm (AC3)

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 each X_k **in** NEIGHBORS$[X_i]$ **do**
        add (X_k, X_i) to *queue*

---

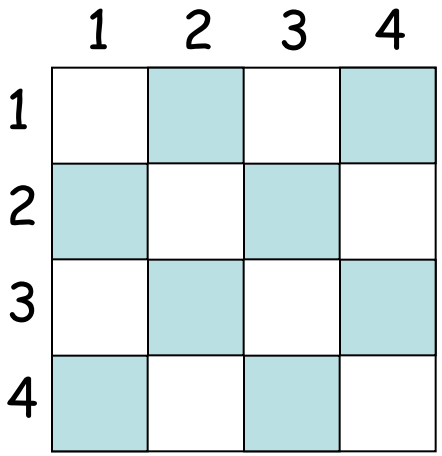
function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **returns** true iff succeeds
  removed $\leftarrow$ *false*
  for each x **in** DOMAIN$[X_i]$ **do**
    if no value y in DOMAIN$[X_j]$ allows (x,y) to satisfy the constraint $X_i \leftrightarrow X_j$
      then delete x from DOMAIN$[X_i]$; *removed* $\leftarrow$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

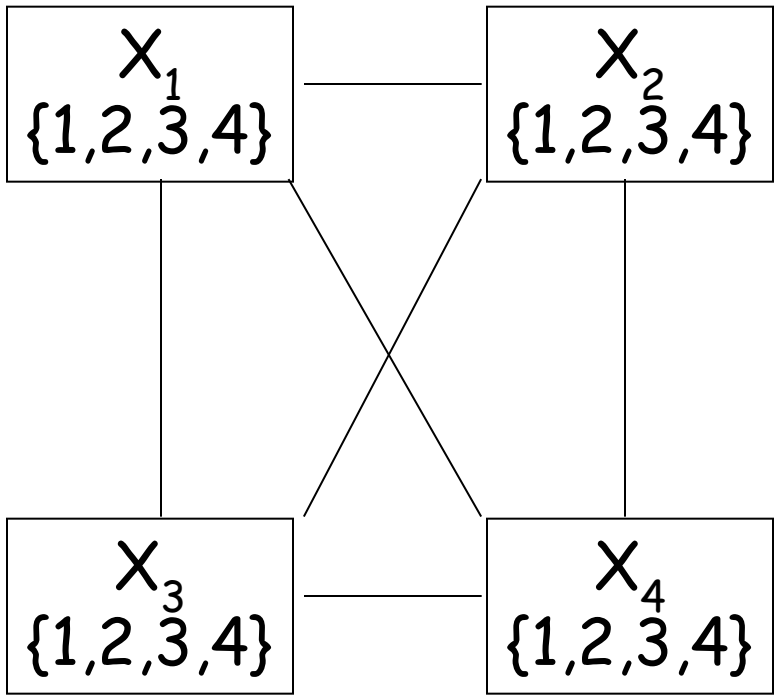
Modified Backtracking with AC3

```
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
    Run AC3 and update csp.domains accordingly
    If a variable has an empty domain, then return failure
    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
            csp.domains ← forwardChecking(csp.domains, X, v, A)
            if no empty domain {
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result }
            remove {var = value} from assignment
    return failure
```

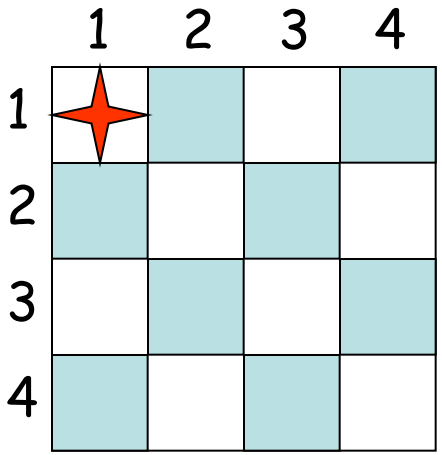
A Complete Example: 4-Queens Problem

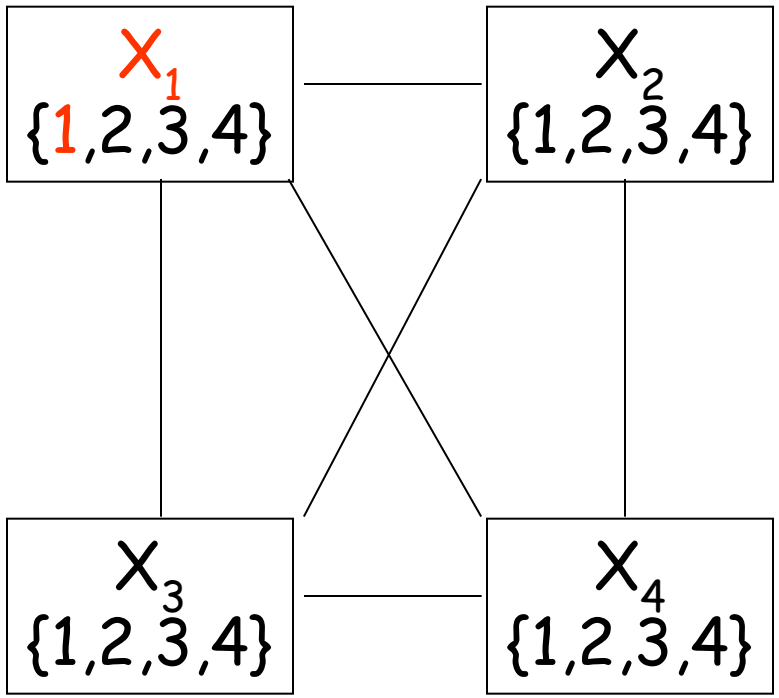




1) The modified backtracking algorithm starts by calling AC3, which removes no value

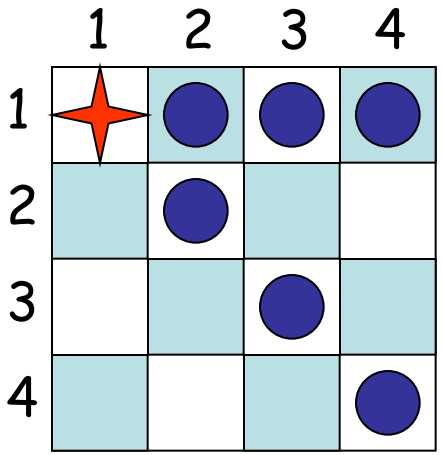
4-Queens Problem

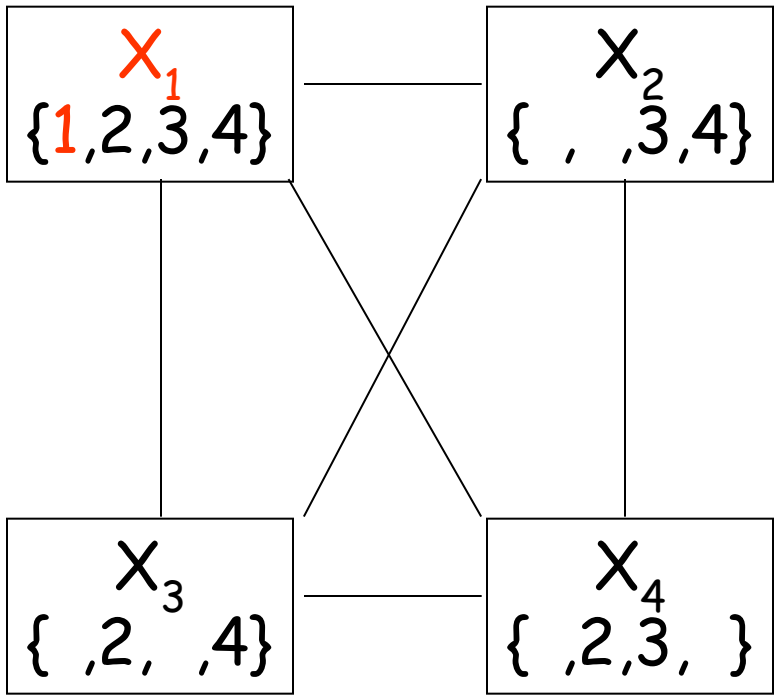




2) The backtracking algorithm then selects a variable and a value for this variable. No heuristic helps in this selection. X_1 and the value 1 are arbitrarily selected

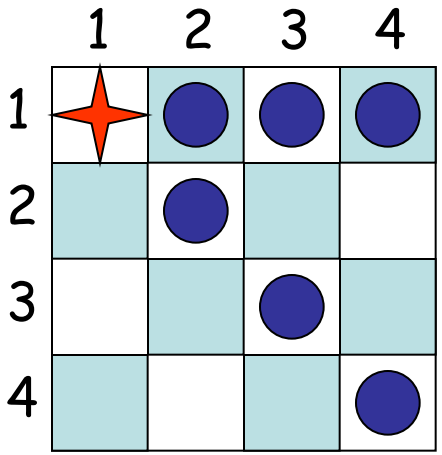
4-Queens Problem

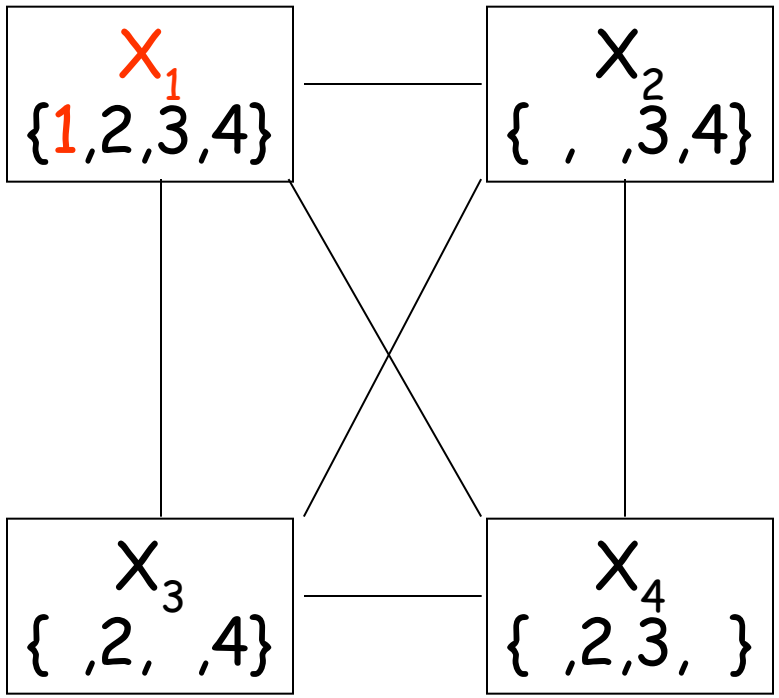




3) The algorithm performs forward checking, which eliminates 2 values in each other variable's domain

4-Queens Problem



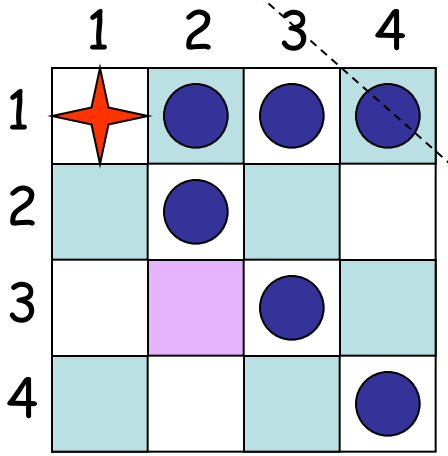


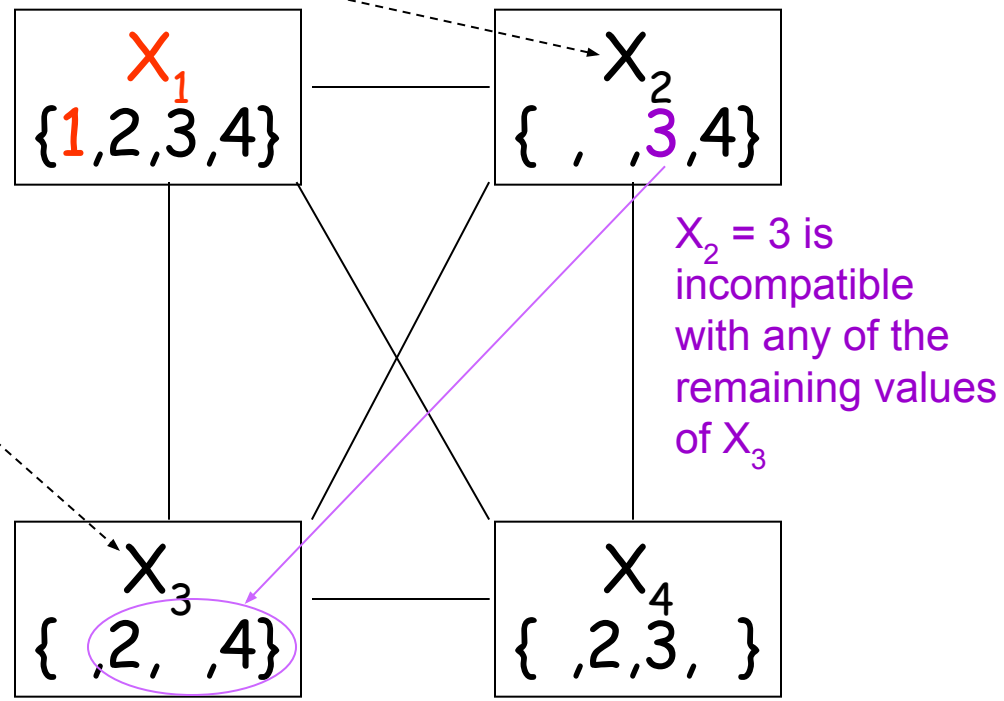
4) The algorithm calls AC3

4-Queens Problem

REMOVE-VALUES(X,Y)

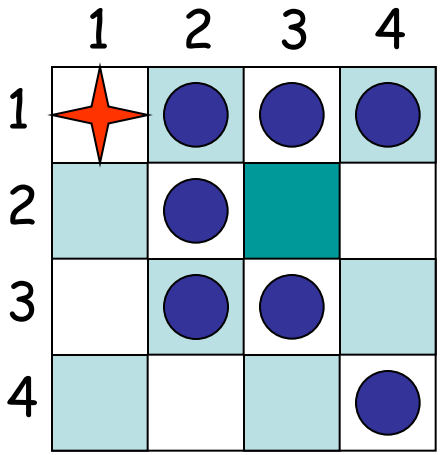
1. removed ← false
2. For every value v in the domain of Y do
   - If there is no value u in the domain of X such that the constraint on (x,y) is satisfied then
     a. Remove v from Y's domain
     b. removed ← true
3. Return removed

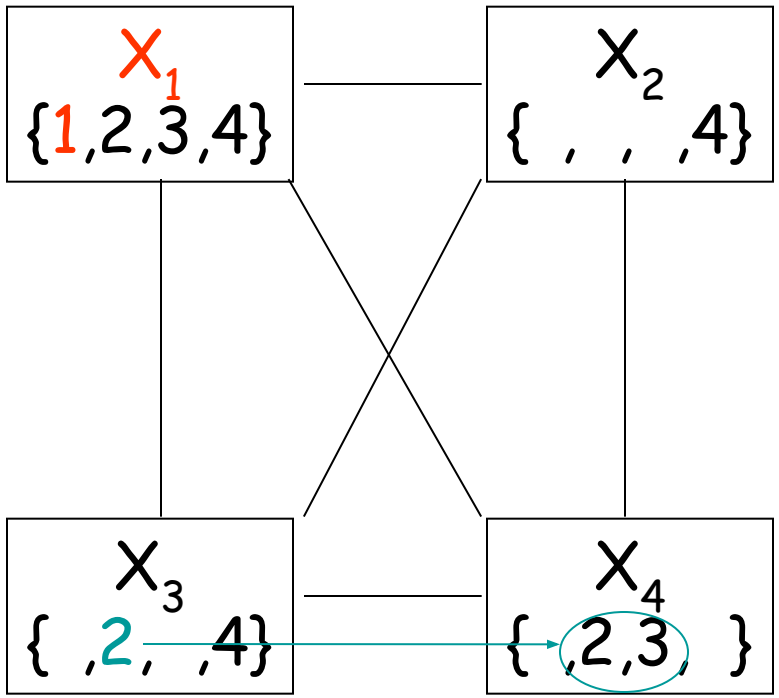




4) The algorithm calls AC3, which eliminates 3 from the domain of X_2

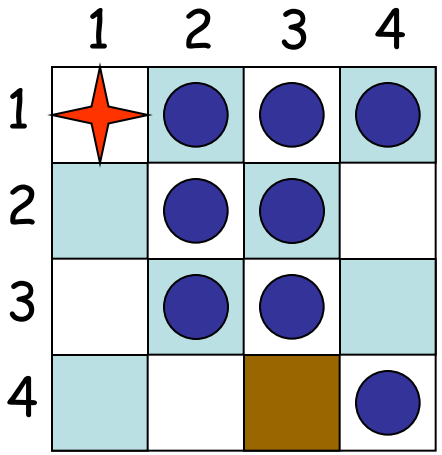
4-Queens Problem

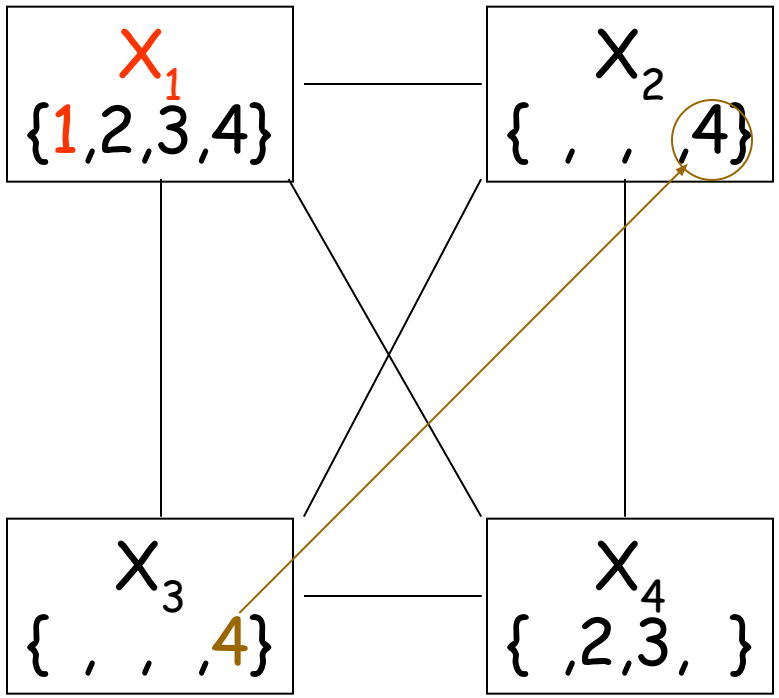




4) The algorithm calls AC3, which eliminates 3 from the domain of X_2, and 2 from the domain of X_3

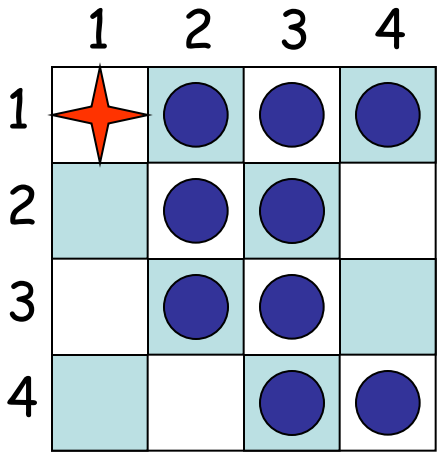
4-Queens Problem

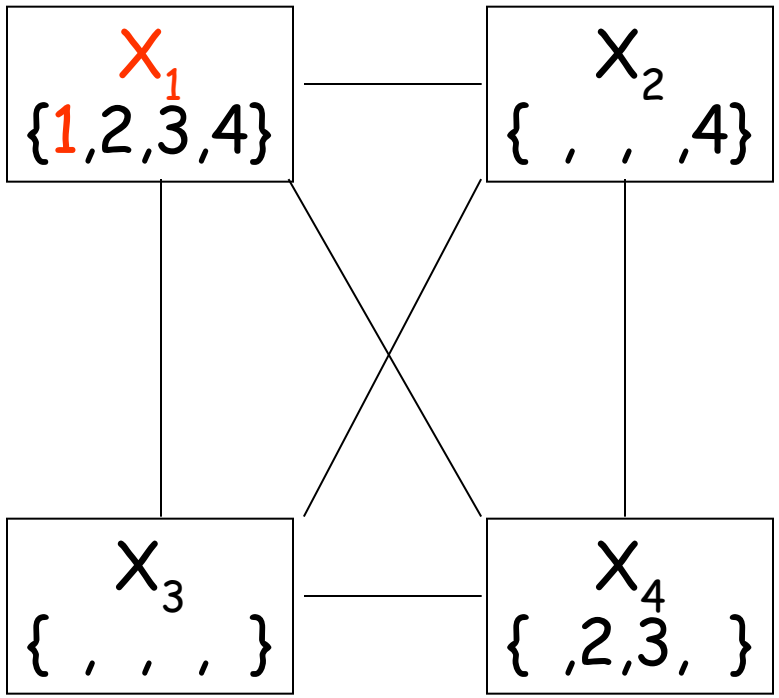




4) The algorithm calls AC3, which eliminates 3 from the domain of X_2, and 2 from the domain of X_3, and 4 from the domain of X_3

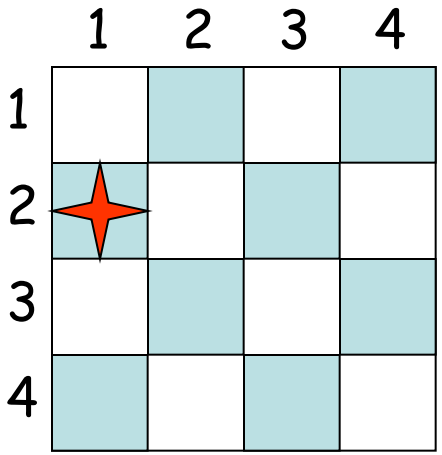
4-Queens Problem

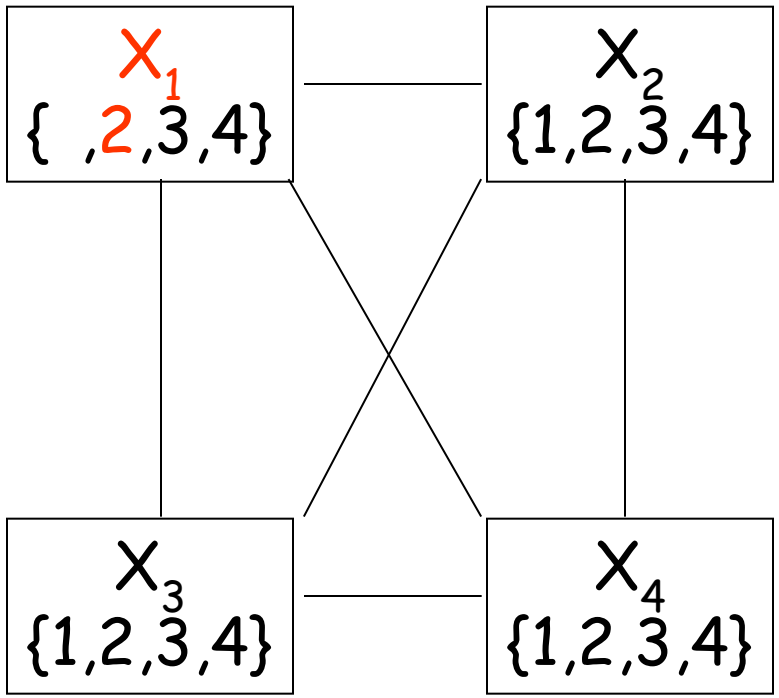




5) The domain of X_3 is **empty** → backtracking

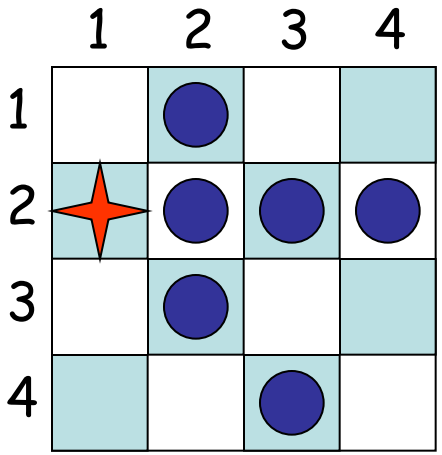
4-Queens Problem

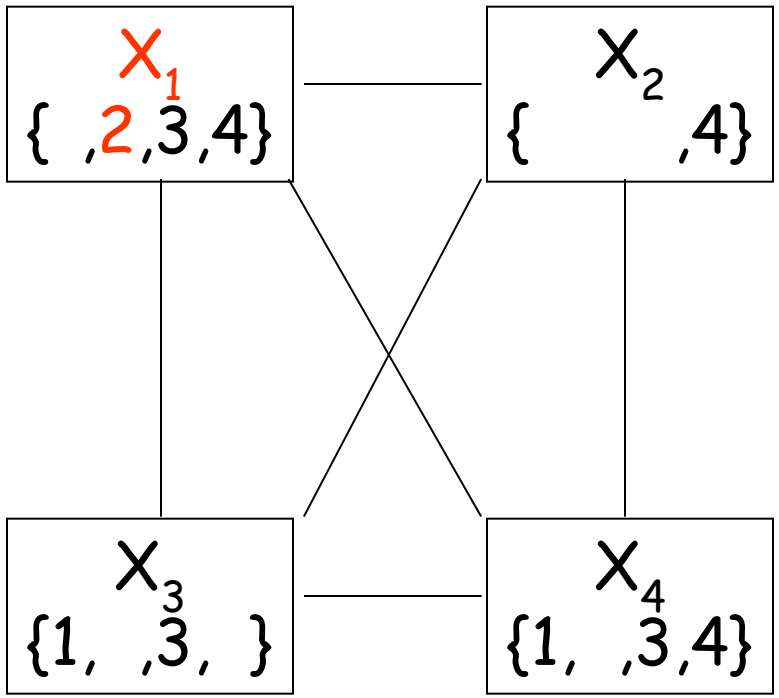




6) The algorithm removes 1 from X_1's domain and assign 2 to X_1

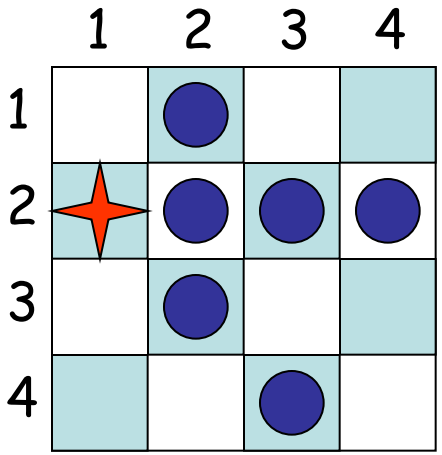
4-Queens Problem

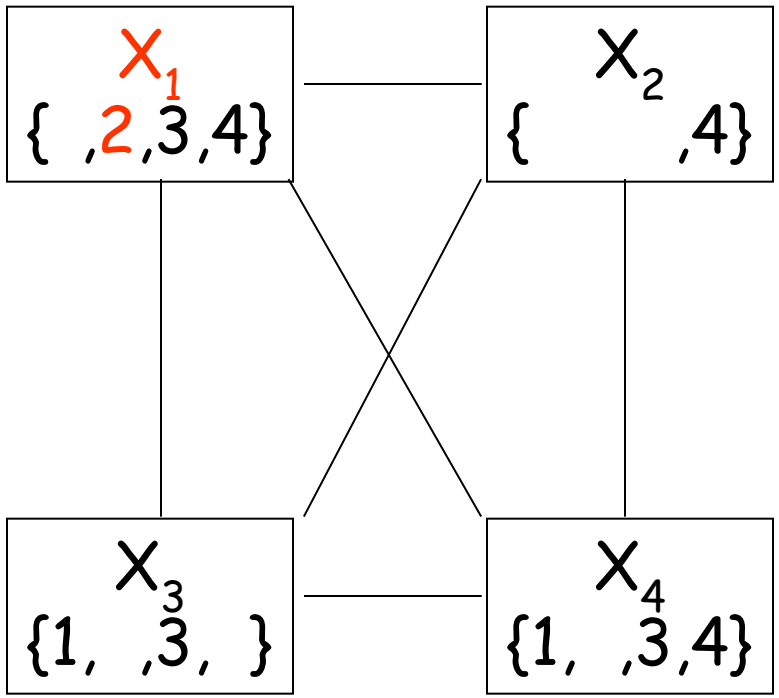




7) The algorithm performs forward checking

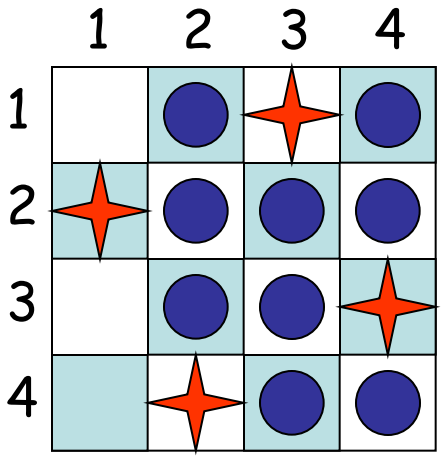
4-Queens Problem

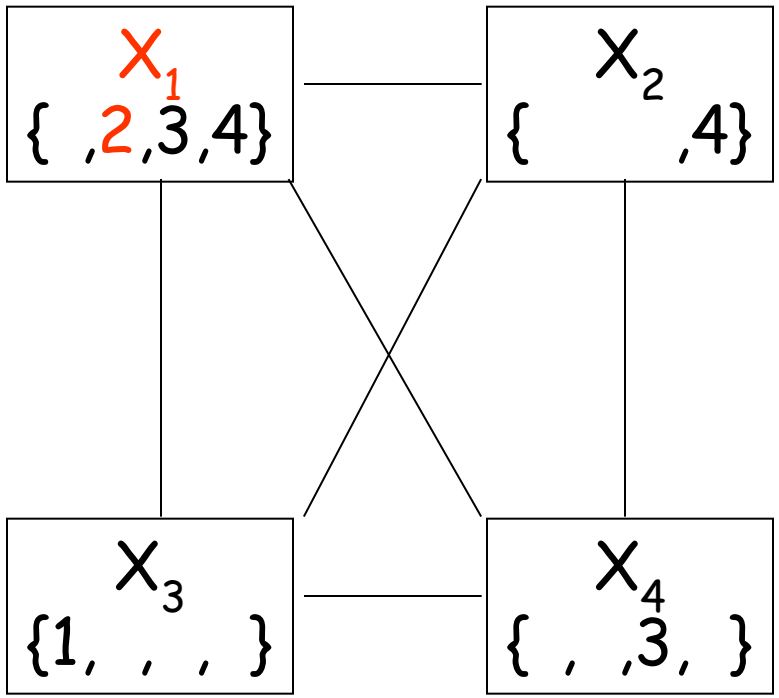




8) The algorithm calls AC3

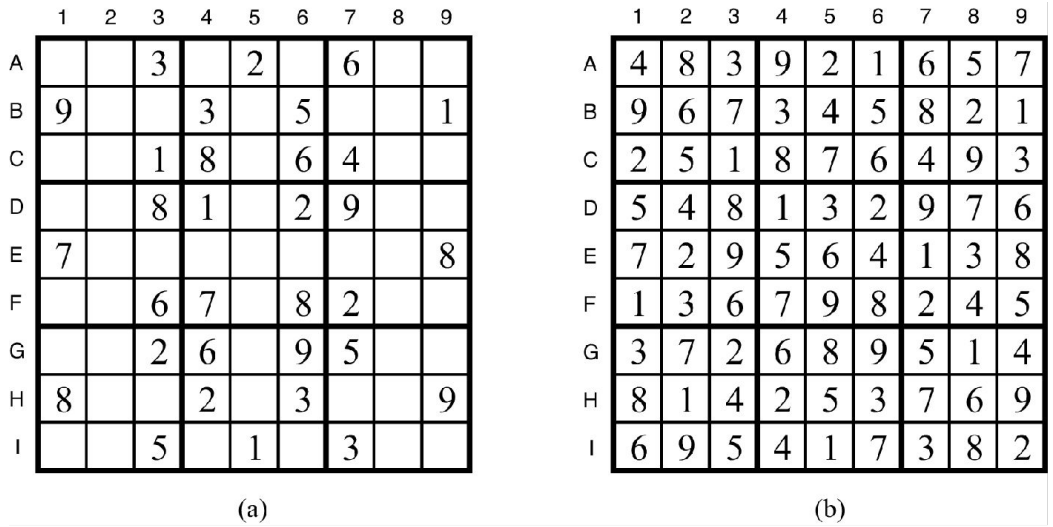
4-Queens Problem





8) The algorithm calls AC3, which reduces the domains of X_3 and X_4 to a single value

Example: Sudoku puzzle



	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Variables? $X_{A1}, X_{A2}, \ldots, X_{I9}$

Domains? {1,2,3,...,9} for all variables

Constraints? $X_{A1} != X_{A2}, \ldots \rightarrow$ 972 binary constraints

Global Constraints

- Involve arbitrary numbers of variablesI
- Usually come with specialized efficient propagation algorithm
- Example:

  Alldiff(X_1, …, X_n) meaning X_1, …, X_n must have all different values
- In Sudoku:

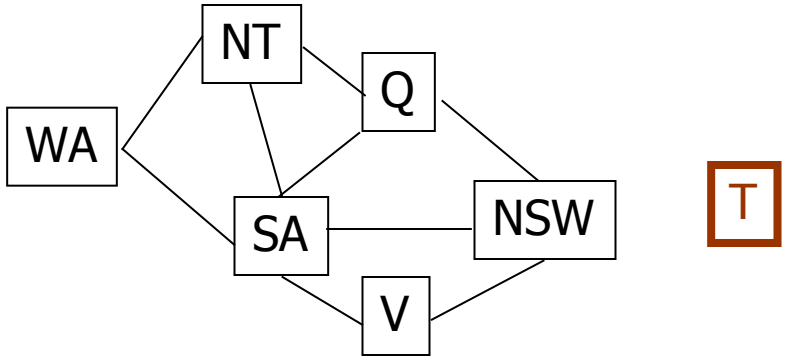
  Alldiff(X_{A1}, …, X_{A9}), Alldiff(X_{B1}, …, X_{B9}), …

  → only 27 constraints

  → better propagation

Exploiting the Structure of CSP

If the constraint graph contains several components, then solve one independent CSP per component



Size of the state space?

$d^{n1}+d^{n2}$ instead of $d^{(n1+n2)}$

($3+3^6 = 732$ instead of $3^7 = 2187$)

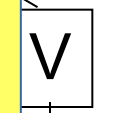
Exploiting the Structure of CSP

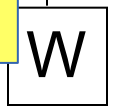
If the constraint graph is a tree, then:

1. Order the variables from the root to the leaves
   ⇒ $(X_1, X_2, \ldots, X_n)$
2. For j = n, n-1, …, 2 call REMOVE-VALUES(X_j, X_i) where X_i is the parent of X_j
3. Assign any valid value to X_1
4. For j = 2, …, n do
   Assign any value to X_j consistent with the value assigned to its parent X_i

Algorithm runs in time linear in the number of variables.

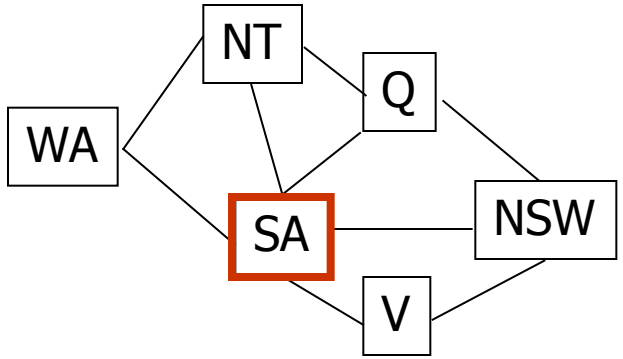
No backtracking required!





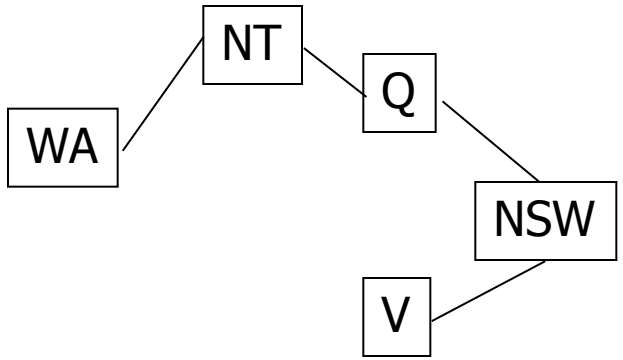
Exploiting the Structure of CSP

Whenever a variable is assigned a value by the backtracking algorithm, propagate this value and remove the variable from the constraint graph



Exploiting the Structure of CSP

Whenever a variable is assigned a value by the backtracking algorithm, propagate this value and remove the variable from the constraint graph



If the graph becomes a tree, then proceed as shown in previous slide

Summary

- CSPs are a special kind of search problems
- Backtracking search
- Constraint propagation (forward checking, AC3, consistency)
- Heuristics for variable selection and value ordering
- Exploiting the structure of CSPs can reduce complexity