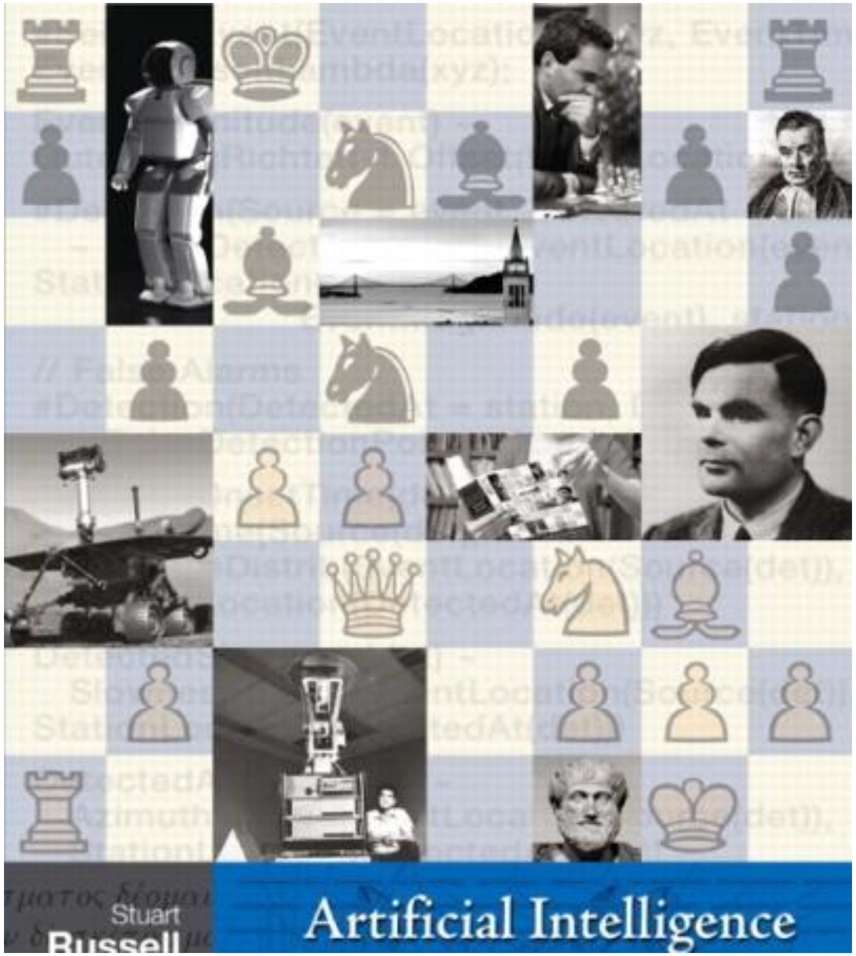
Lecture 5: Constraint Satisfaction Problems
Russell and Norvig Chapter 5



CS-4820/5820

Tu/Th 12:15 PM-1:30 PM

Room: Centennial Hall 106

Instructor: Adham Atyabi

Office: Engineering 243

Office Hours: Mon 9:00 AM-14:00 PM.

Email: aatyabi@uccs.edu

Teaching Assistant: Ali Al Shami (aalshami@uccs.edu)

Outline

- Constraint Satisfaction Problems (CSP)
- Backtracking search for CSPs
- Local search for CSPs

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

Constraint Satisfaction Problems

- What is a CSP?
  - Finite set of variables $X_1, X_2, \ldots, X_n$
  - Nonempty domain of possible values for each variable $D_1, D_2, \ldots, D_n$
  - Finite set of constraints $C_1, C_2, \ldots, C_m$
    - ***Each constraint C_i limits the values that variables can take,***
    - ***e.g., $X_1 \neq X_2$***
  - Each constraint C_i is a pair <scope, relation>
    - ***Scope =*** *Tuple of variables that participate in the constraint.*
    - ***Relation =*** *List of allowed combinations of variable values.*

      May be an explicit list of allowed combinations.

      May be an abstract relation allowing membership testing and listing.

CSPs --- what is a solution?

- A *state* is an *assignment* of values to some or all variables.
  – An assignment is *complete* when every variable has a value.
  – An assignment is *partial* when some variables have no values.

- ***Consistent assignment***
  – assignment does not violate the constraints

- A ***solution*** to a CSP is a complete and consistent assignment.

- Some CSPs require a solution that maximizes an *objective function*.

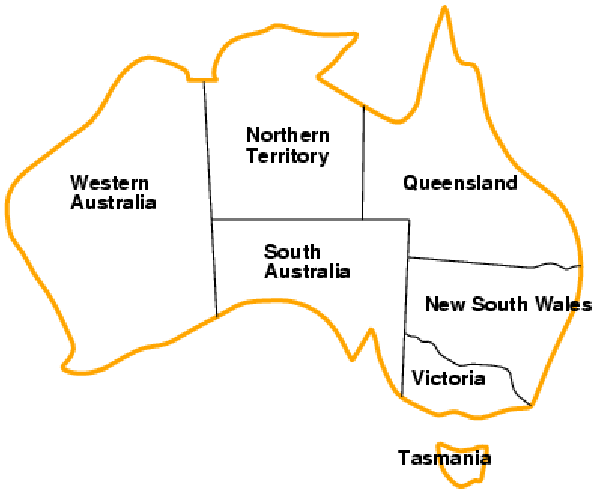
8-Queens

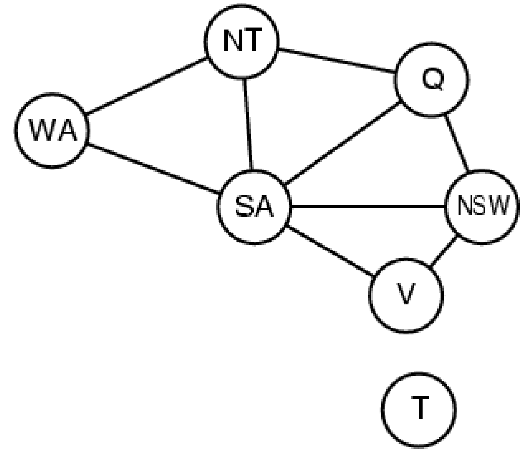
- Variables: Queens, one per column
  - Q1, Q2, …, Q8

- Domains: row placement, {1,2,…,8}

- Constraints:
  - $Q_i \mathrel{!}= Q_j$ $(j \mathrel{!}= i)$
  - $|Q_i - Q_j| \mathrel{!}= |i - j|$

Example: Map-Coloring





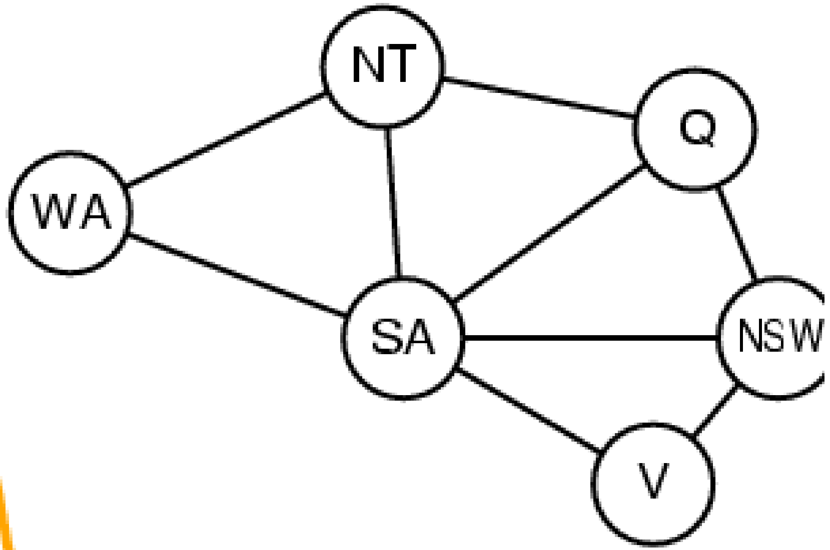
- Variables X={*WA, NT, Q, NSW, V, SA, T*}
- Domains D_i = {red,green,blue}
- Constraints: adjacent regions must have different colors

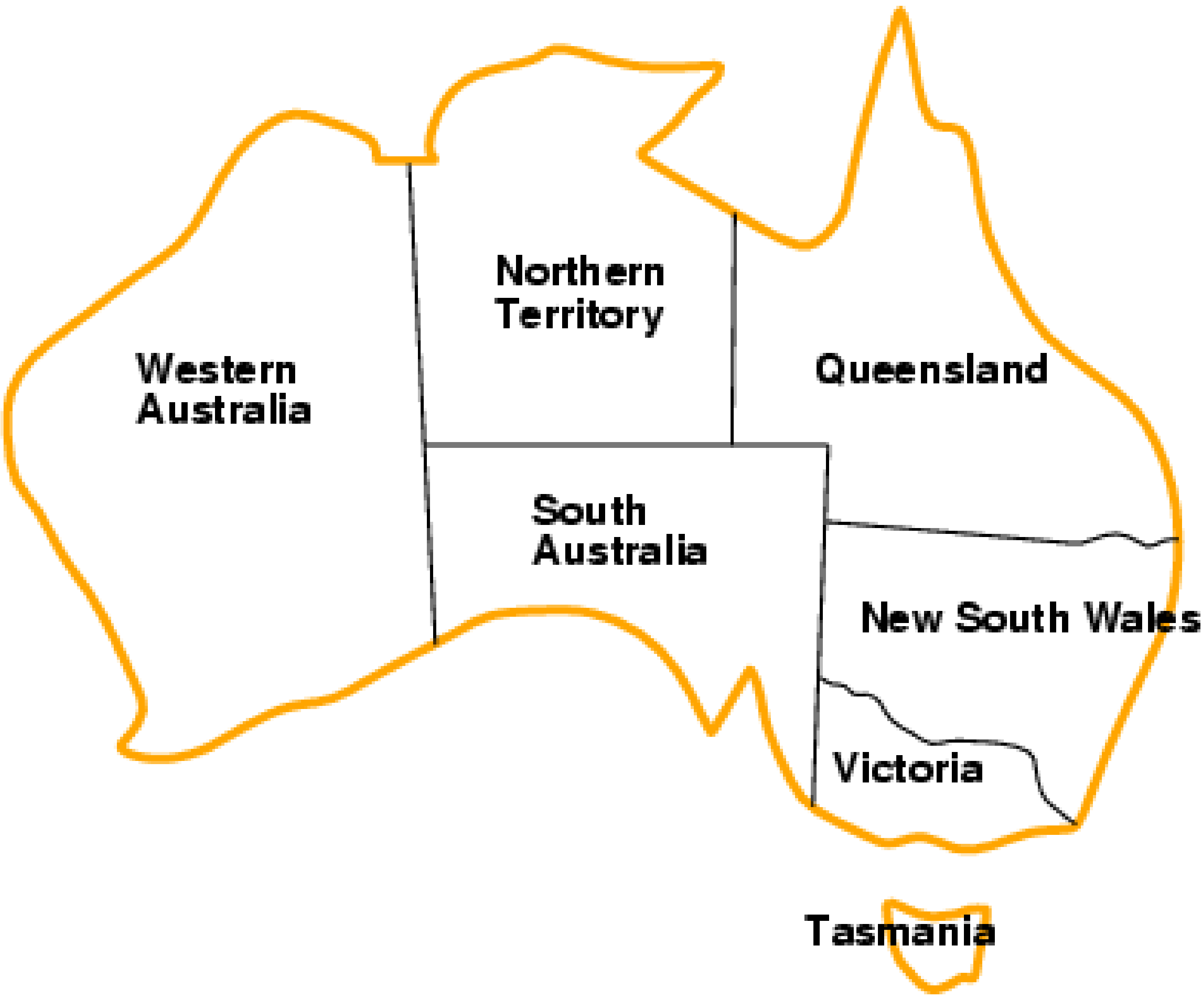
Or in a mathematical representation:

$$C=\{<(\forall x_j, x_i \text{ such that } x_i \text{ touches } x_j),(Color(x_i) \neq Color(x_j))>\}$$

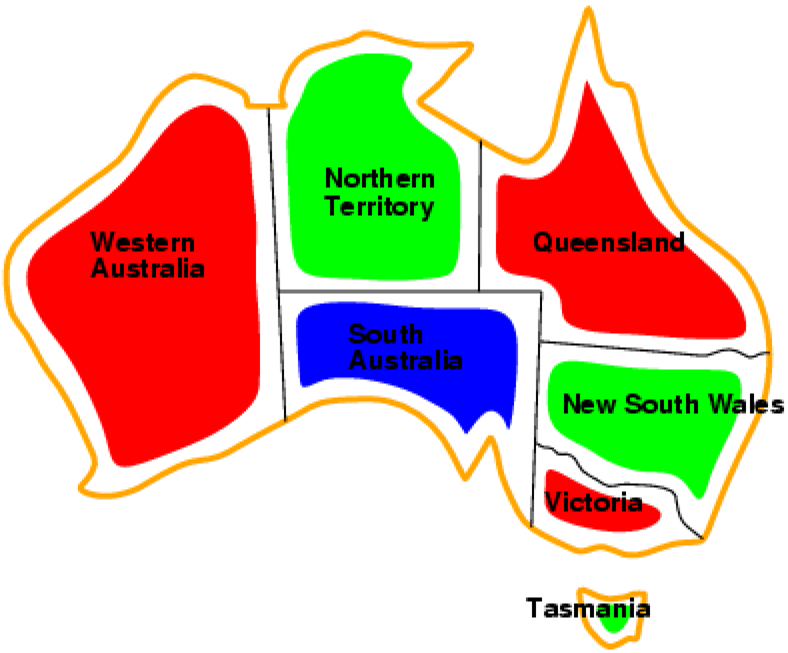
- e.g., WA ≠ NT, or (WA,NT) in {(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green)}

What is the Solution?





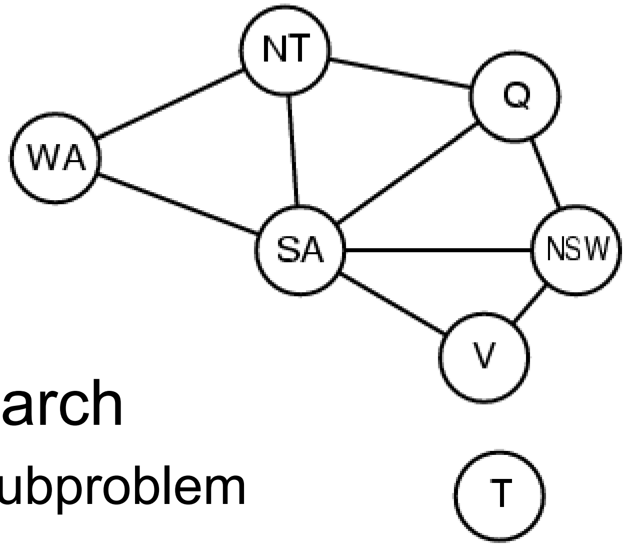
CSP Example: Map-Coloring



- Solutions are complete and assignments are satisfying consistent, 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 binary constraints



- Graph can be used to simplify search

    e.g. Tasmania is an independent subproblem

Sudoku as a Constraint Satisfaction Problem (CSP)

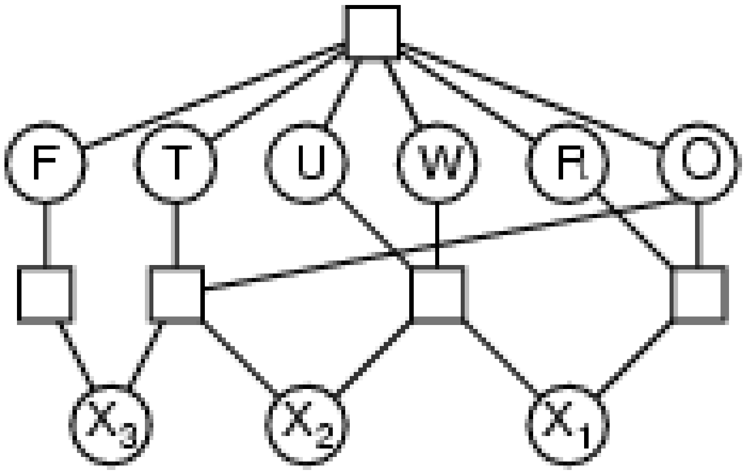
- Variables: 81 variables
  - A1, A2, A3, …, I7, I8, I9
  - Letters index rows, top to bottom
  - Digits index columns, left to right
- Domains: The nine positive digits
  - A1 ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9}
  - Etc.
- Constraints: ***27 Alldiff*** constraints
  - *Alldiff* (A1, A2, A3, A4, A5, A6, A7, A8, A9)
  - Alldiff(A1, B1, C1, D1, E1, F1, G1, H1, I1)
  - Etc.

Alldiff (All-Different) is a global constraint that enforces all variables in a given set to take distinct values.

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

Example: Cryptarithmetic puzzle

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



- 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: *Alldiff (F,T,U,W,R,O)*
  - $O + O = R + 10 \cdot X_1$
  - $X_1 + W + W = U + 10 \cdot X_2$
  - $X_2 + T + T = O + 10 \cdot X_3$
  - $X_3 = F,\ T \neq 0,\ F \neq 0$

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 are 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
- Preference (soft constraints)
  - e.g. red is better than green often can be represented by a cost for each variable assignment
  - combination of optimization with CSPs

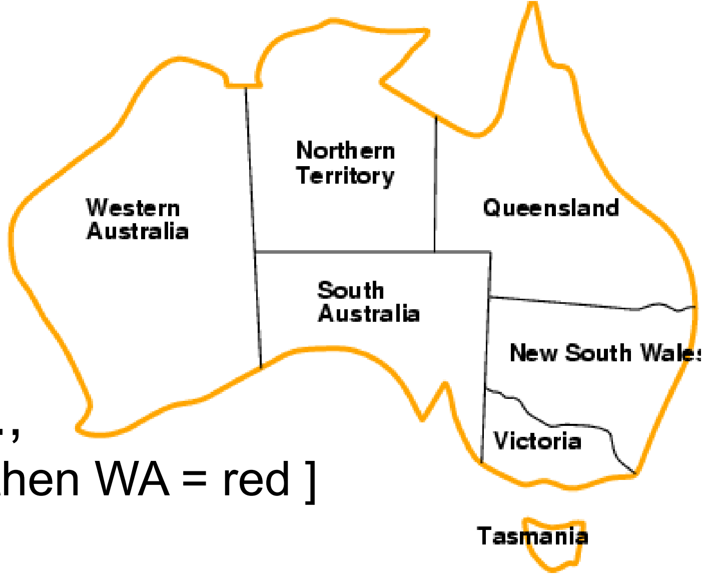
Real-world CSPs

- Assignment problems
  – e.g., who teaches what class
- Timetabling problems
  – e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling
- Notice that many real-world problems involve real-valued variables

Search Depth Limit while solving CSPs

- *CSP with n* variables with domain size *d*
- It will have a branching factor at top = nd
- At the next level: (n-1)d
- In the end, $n!d^n$ leaves. But only d^n possible complete assignments

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

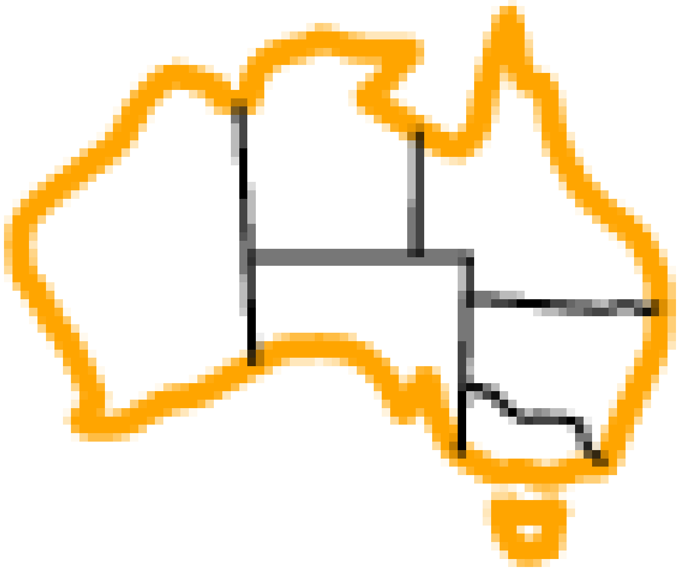
- Can solve *n*-queens for $n \approx 25$

Backtracking search

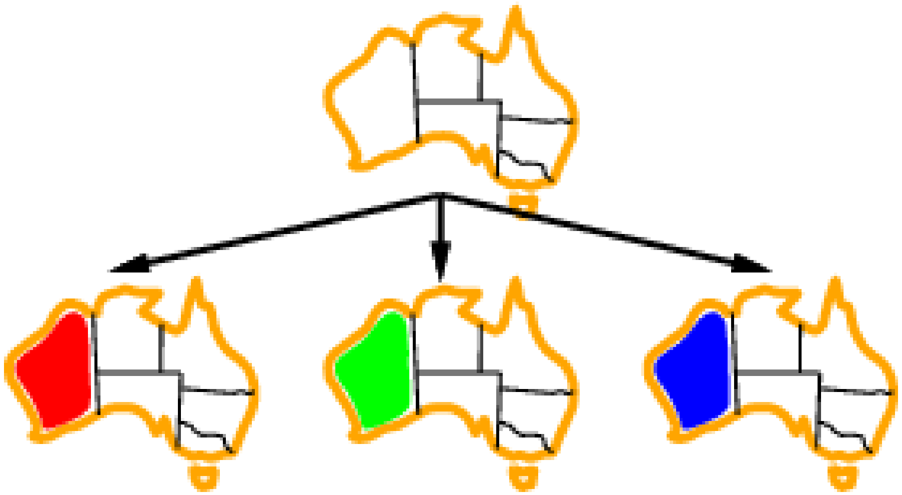
```
function BACKTRACKING-SEARCH( csp) returns a solution, or failure
    return RECURSIVE-BACKTRACKING({}, csp)

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
```

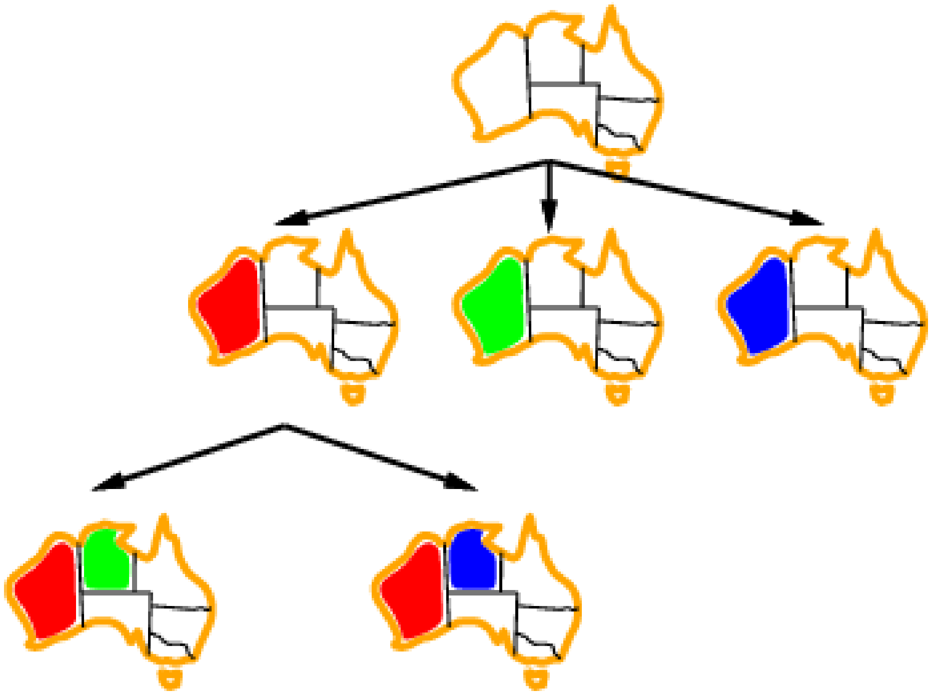
Backtracking example



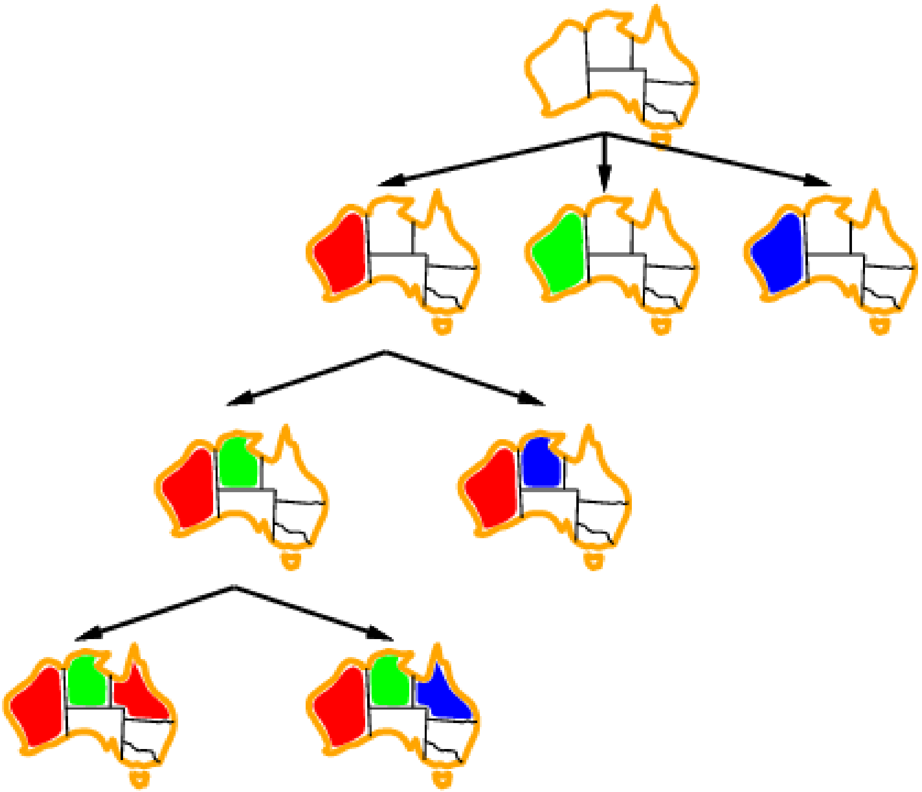
Backtracking example



Backtracking example



Backtracking example



Improving backtracking efficiency

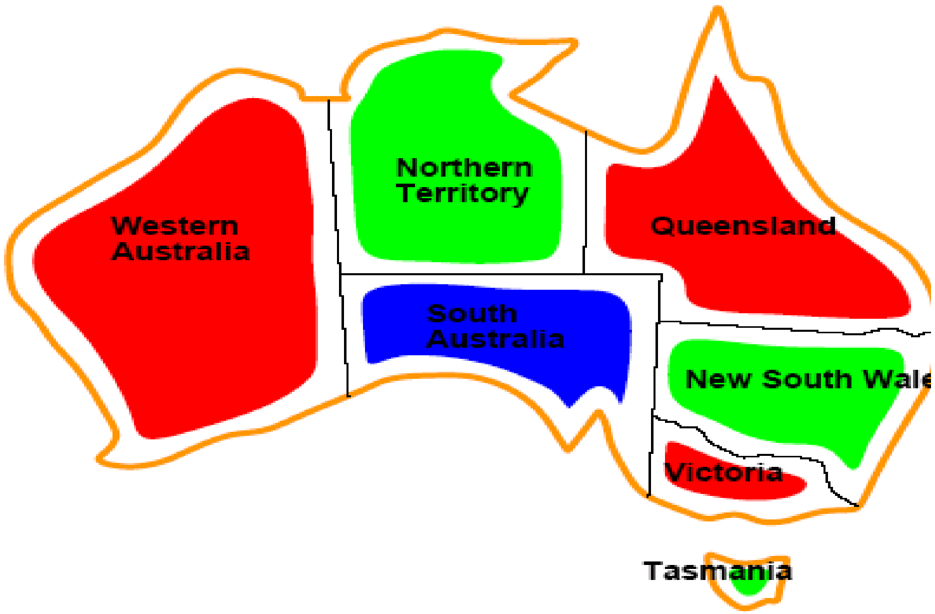
- In CSPs, General-purpose methods can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?

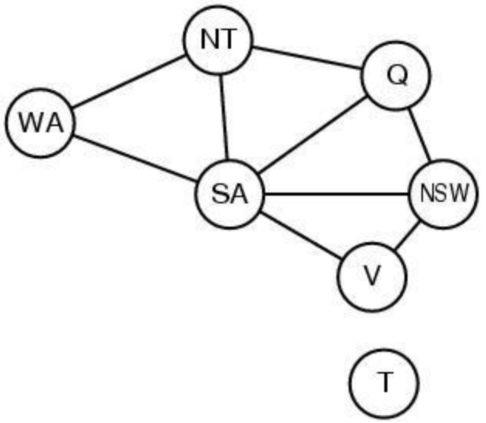
Backtracking search

SELECT-UNASSIGNED-VARIABLE

- Minimum Remaining Values (MRV)
  - Most constrained variable
  - Most likely to fail soon (so prunes pointless searches)
- If a tie (such as choosing the start state), choose the variable involved in the most constraints (degree heuristic) E.g., in the map example, SA adjacent to the most states.
  - Reduces branching factor, since fewer legal successors of that node
- Least constraining value (prefers flexibility for the future)

CSP example: map coloring

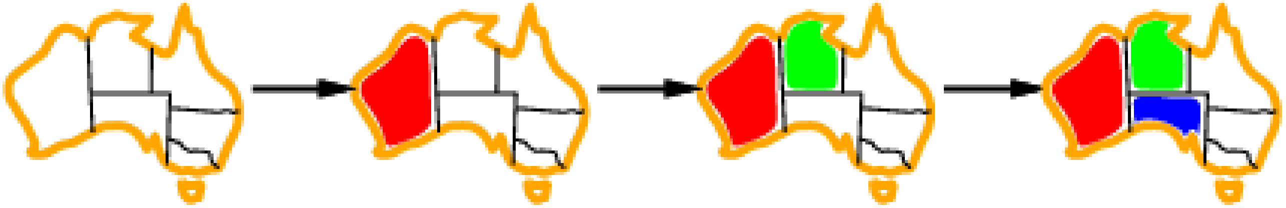




- Solutions are assignments satisfying all constraints, e.g.
  {WA=red,NT=green,Q=red,NSW=green,V=red,SA=blue,T=green}

Most constrained variable

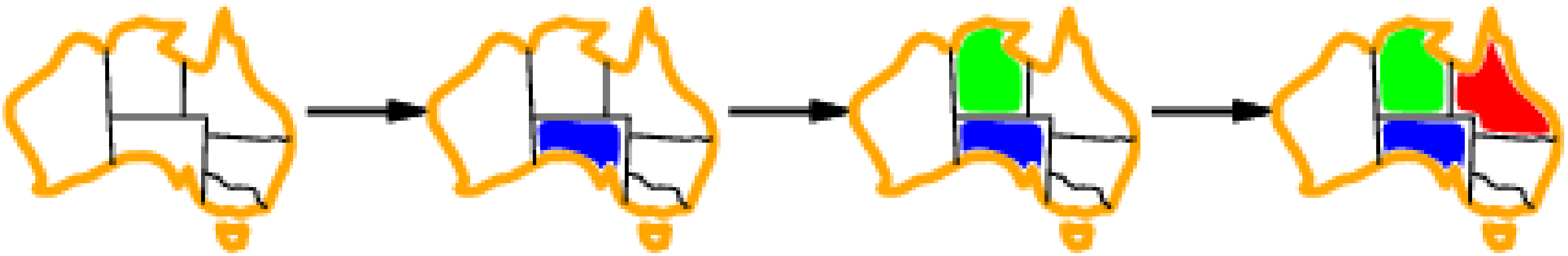
- Most constrained variable:
  choose the variable with the fewest legal values



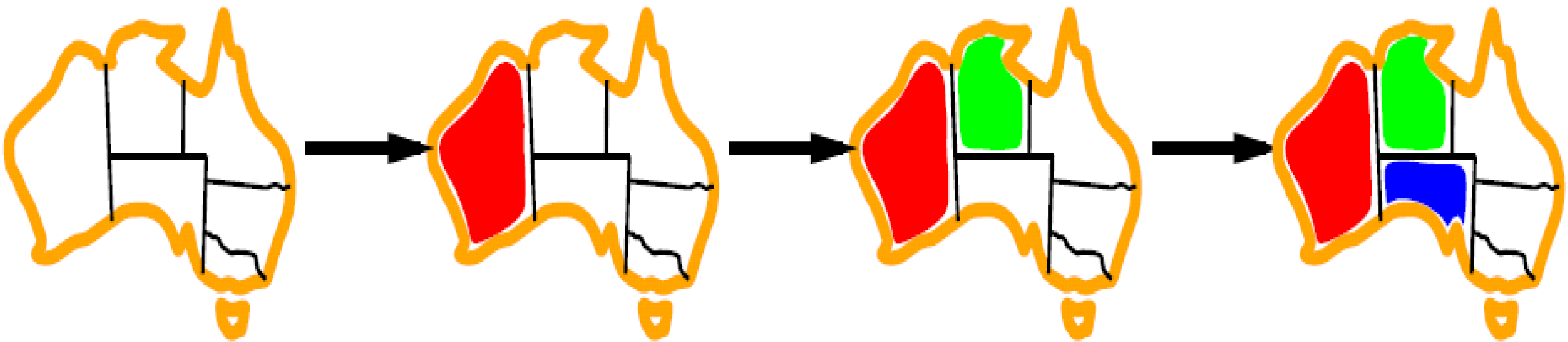
- a.k.a. minimum remaining values (MRV) heuristic

Most constraining variable

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



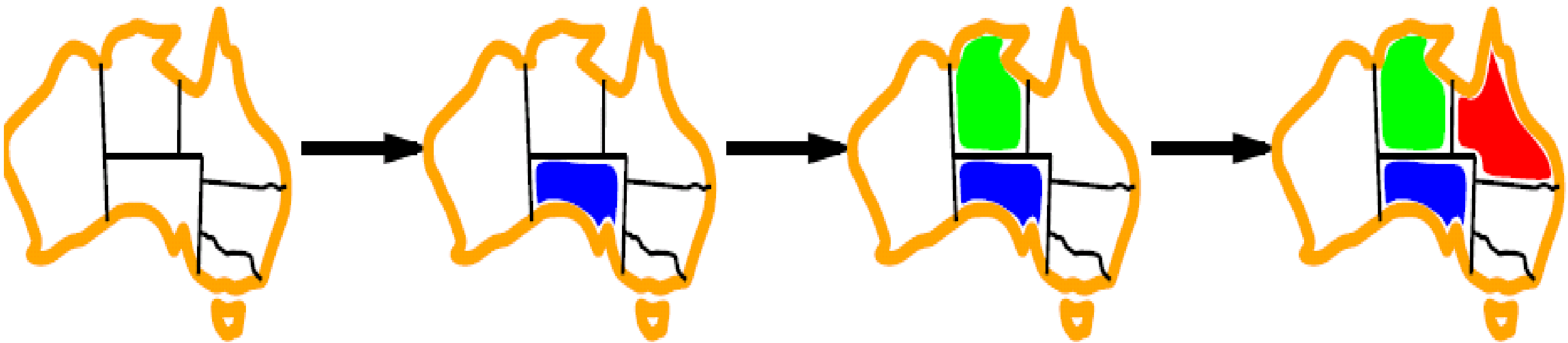
Minimum remaining values (MRV)



var ← SELECT-UNASSIGNED-VARIABLE(*assignment*,*csp*)

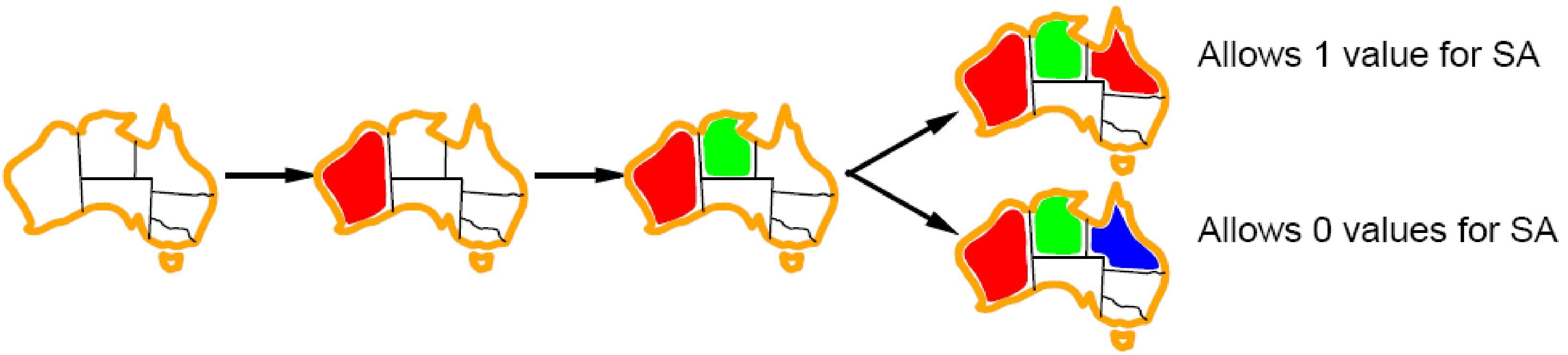
- Before the assignment to the rightmost state: one region has one remaining; one region has two; three regions have three. Choose the region with only one remaining

Degree heuristic for resolving ties among variables



- Degree heuristic can be useful as a tie breaker.
- Before the assignment to the rightmost state, WA and Q have the same number of remaining values ({R}). So, choose the one adjacent to the most states. This will cut down on the number of legal successor states to it.

Least constraining value for value-ordering

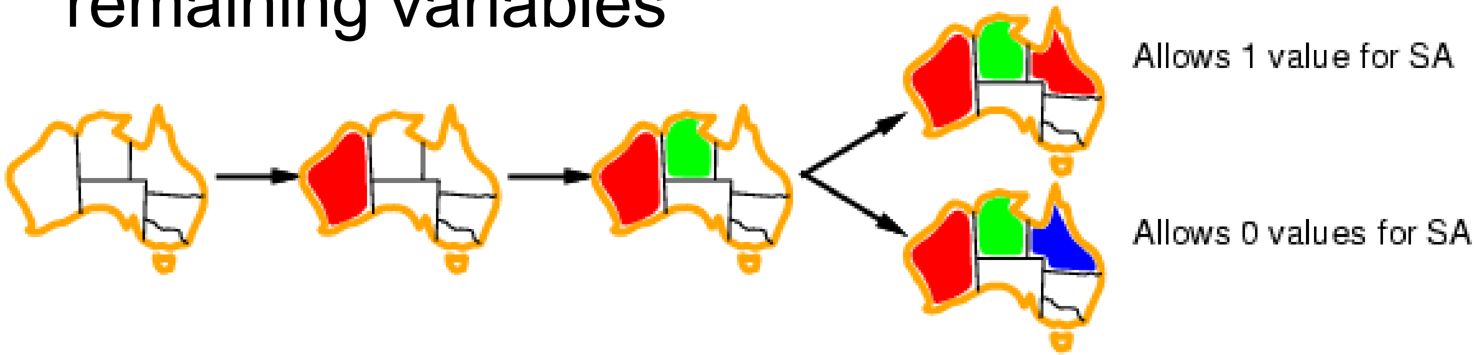


- Least constraining value heuristic

- Heuristic Rule: given a variable choose the least constraining value
  - leaves the maximum flexibility for subsequent variable assignments

Least constraining value

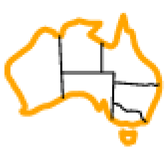
- Given a variable, choose the least constraining value:
  - the one that rules out the fewest values in the remaining variables

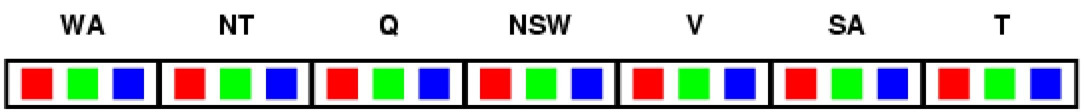


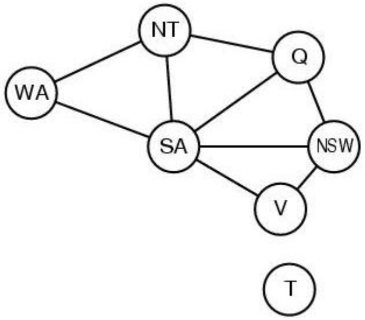
- Combining these heuristics makes 1000 queens feasible

Forward checking

- Can we detect inevitable failure early?
  - And avoid it later?
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values





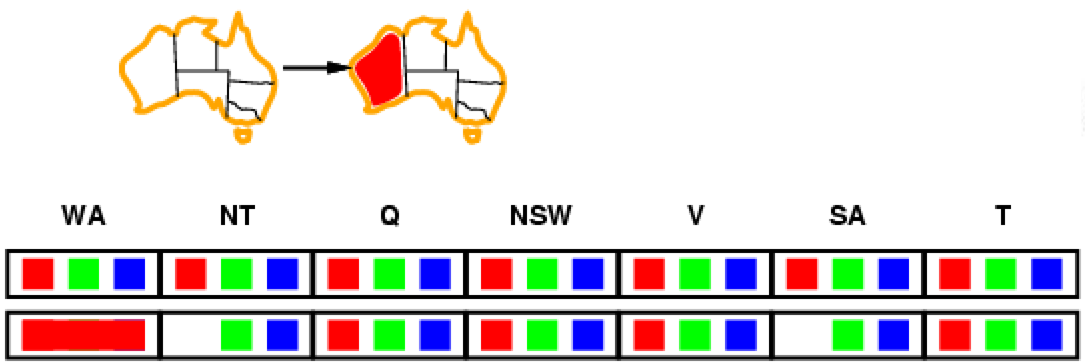


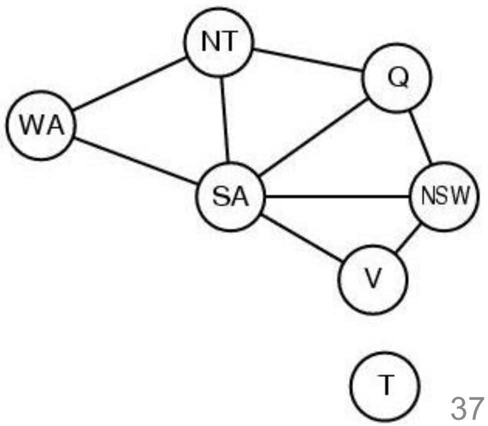
Forward checking

```
function ForwardChecking(csp) //returns a new domain for
    each var
    for each variable X in csp do
        for each unassigned variable Y connected to X do
            for each value d in Domain(Y)
                if d is inconsistent with Value(X)
                    Domain(Y)={Domain(Y)-d}
    return csp //whith modified domains
```

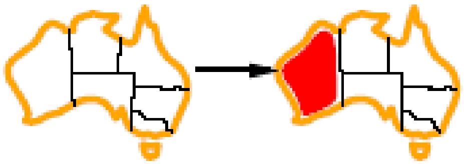
Forward checking

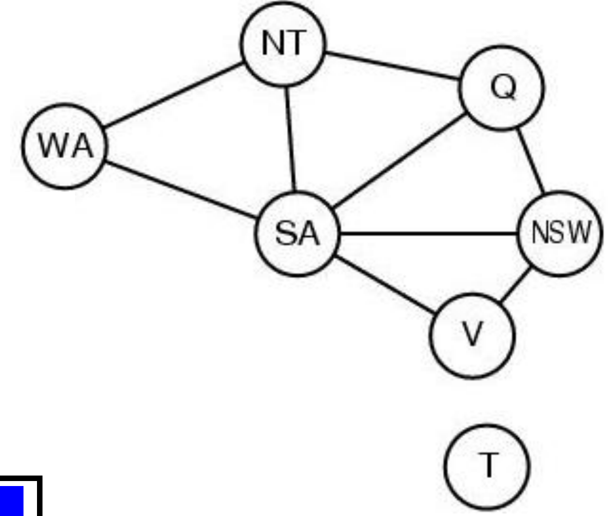
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values

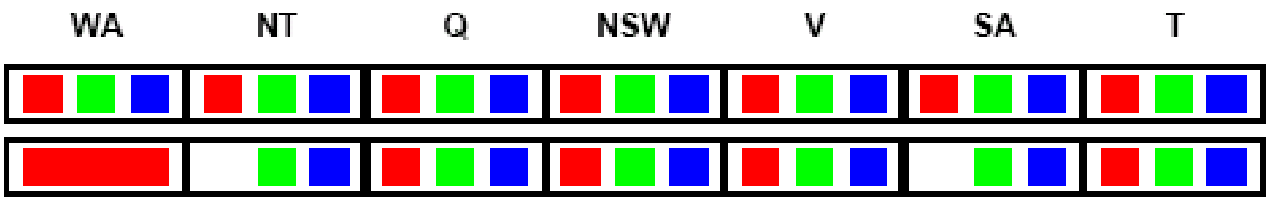




Forward checking







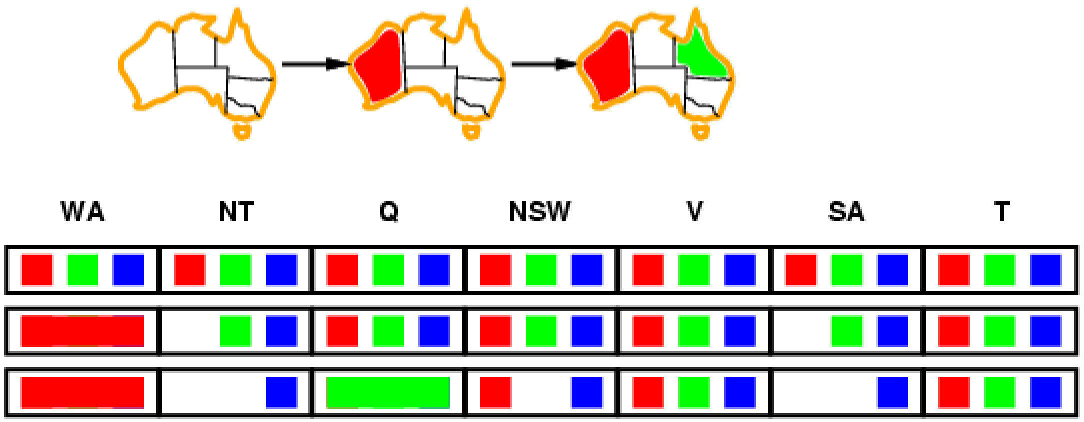
- **Assign *{WA=red}***

- **Effects on other variables connected by constraints to WA**
  - *NT can no longer be red*
  - *SA can no longer be red*

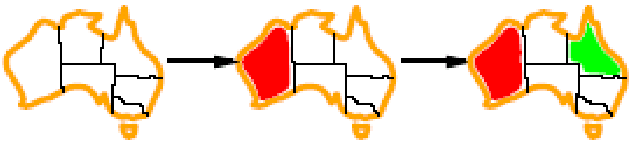
- ***Note: this example is not using MRV; if it were, we would choose NT or SA next. But we will choose Q next. This example is from the text. It shows the example here, then talks through what would happen if we had used MRV.***

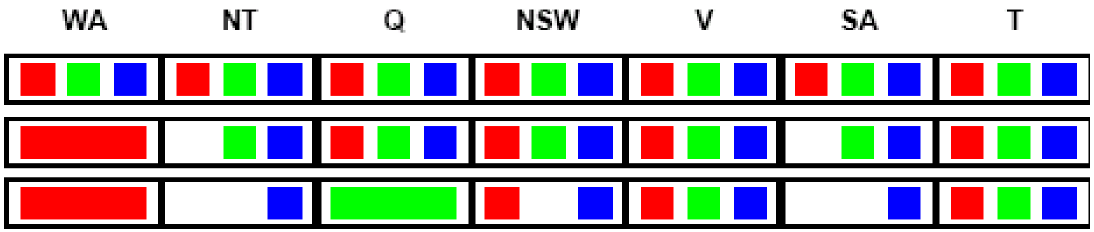
Forward checking

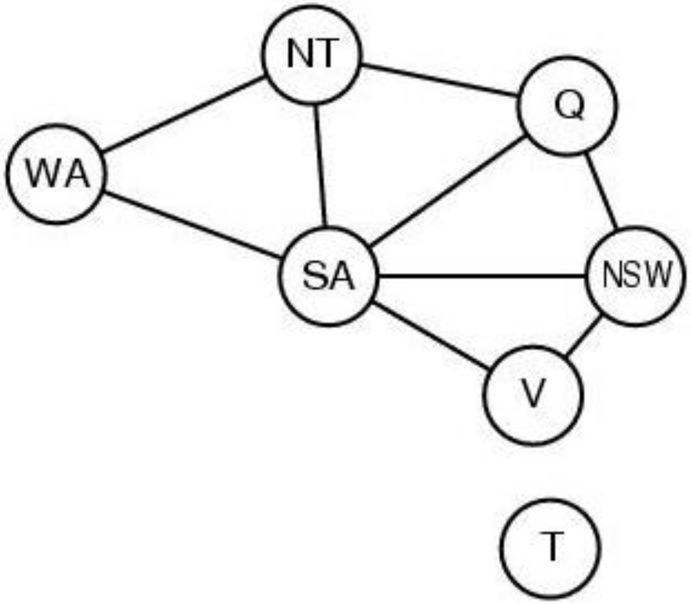
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



Forward checking





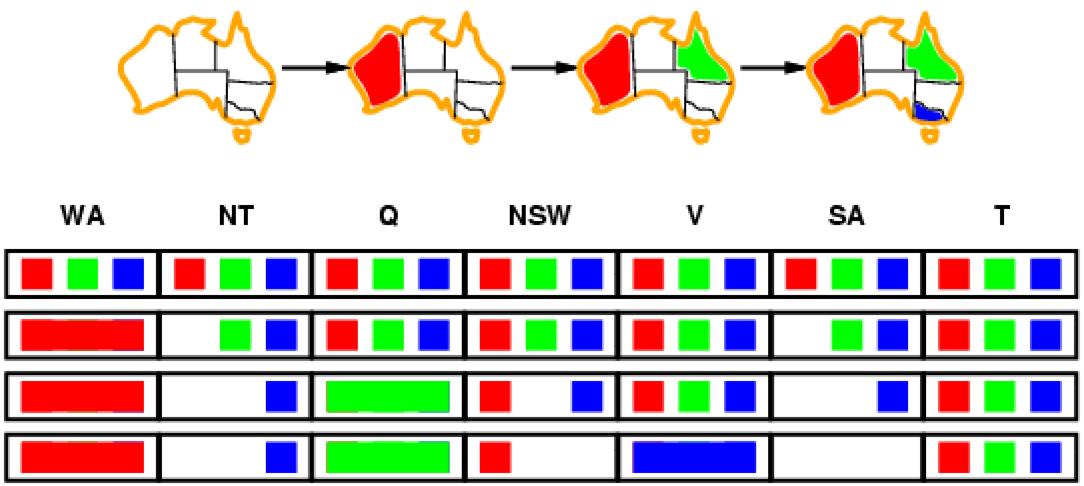


- Assign *{Q=green}*

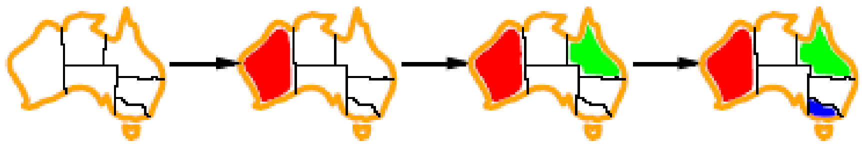
- Effects on other variables connected by constraints with WA
  - *NT can no longer be green*
  - *NSW can no longer be green*
  - *SA can no longer be green*

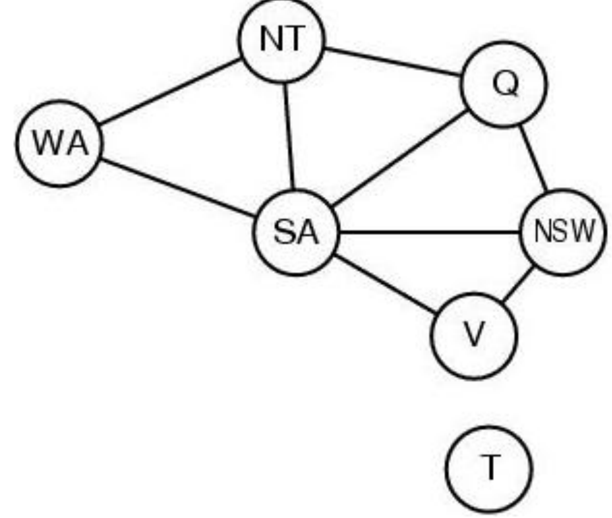
Forward checking

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



Forward checking





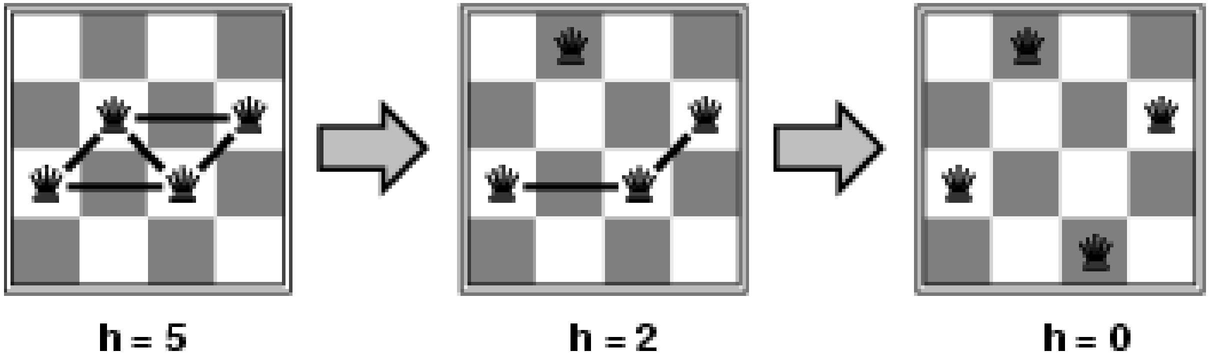
- Assign *{V=blue}*

- Effects on other variables connected by constraints with WA
  - *NSW can no longer be blue*
  - *SA is empty*

- Forward Checking has detected that partial assignment is *inconsistent* with the constraints and backtracking can occur.

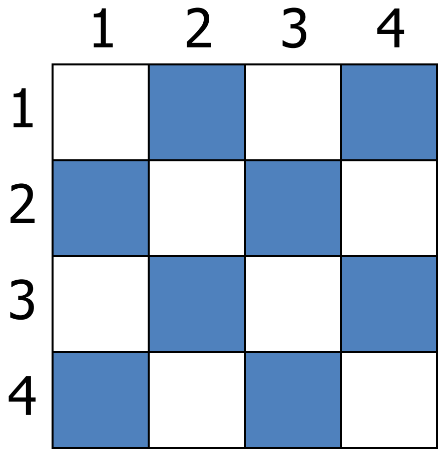
Example: 4-Queens

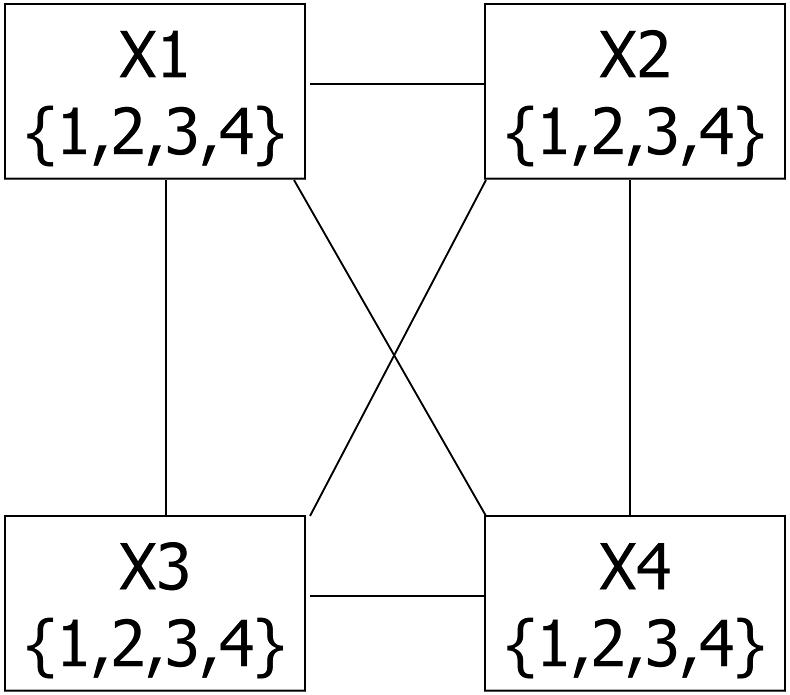
- States: 4 queens in 4 columns (4^4 = 256 states)
- Actions: move queen in column
- Goal test: no attacks
- Evaluation: *h(n)* = number of attacks



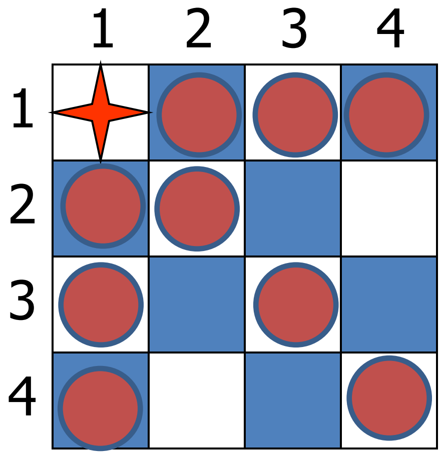
- Given random initial state, can solve *n*-queens in almost constant time for arbitrary *n* with high probability (e.g., *n* = 10,000,000)

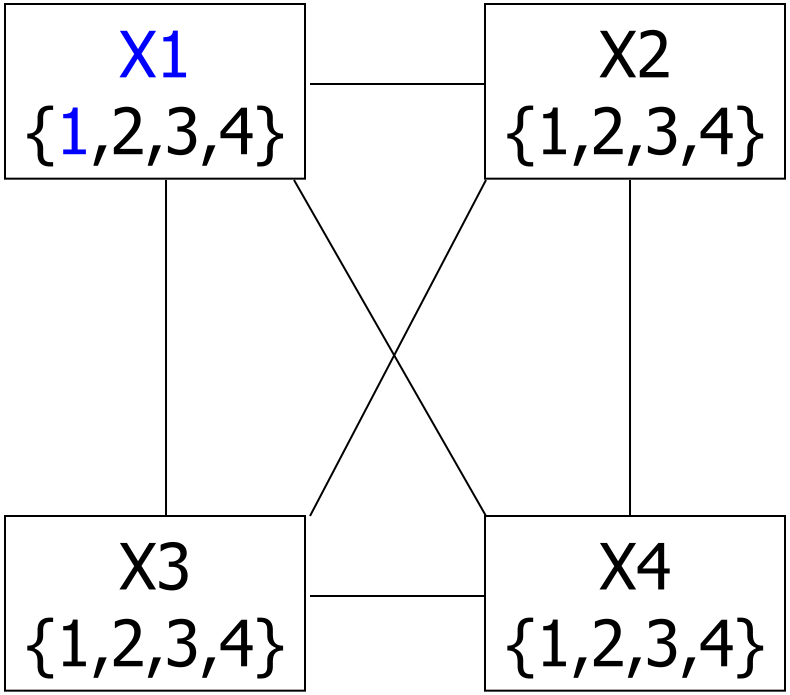
Example: 4-Queens Problem



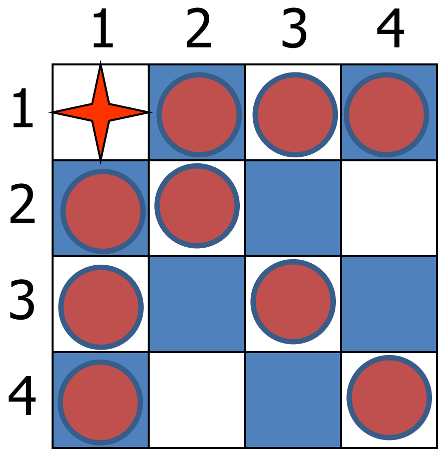


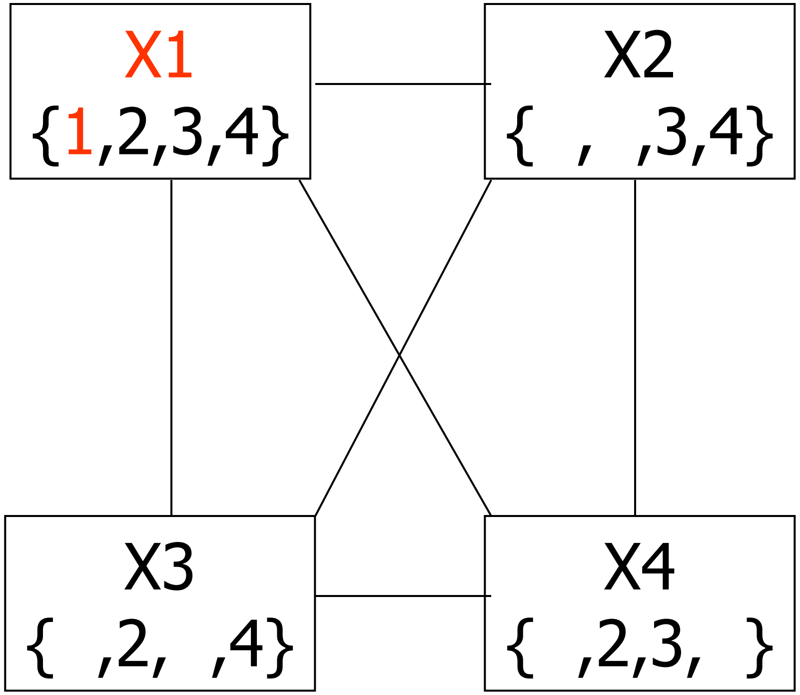
Example: 4-Queens Problem



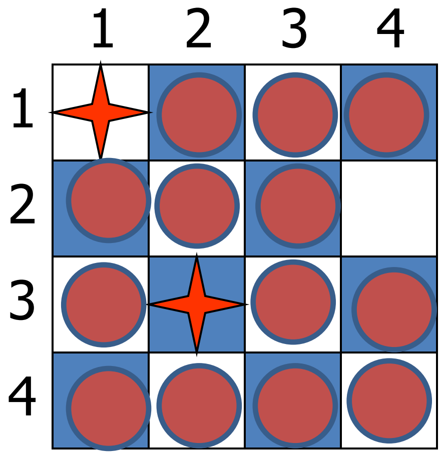


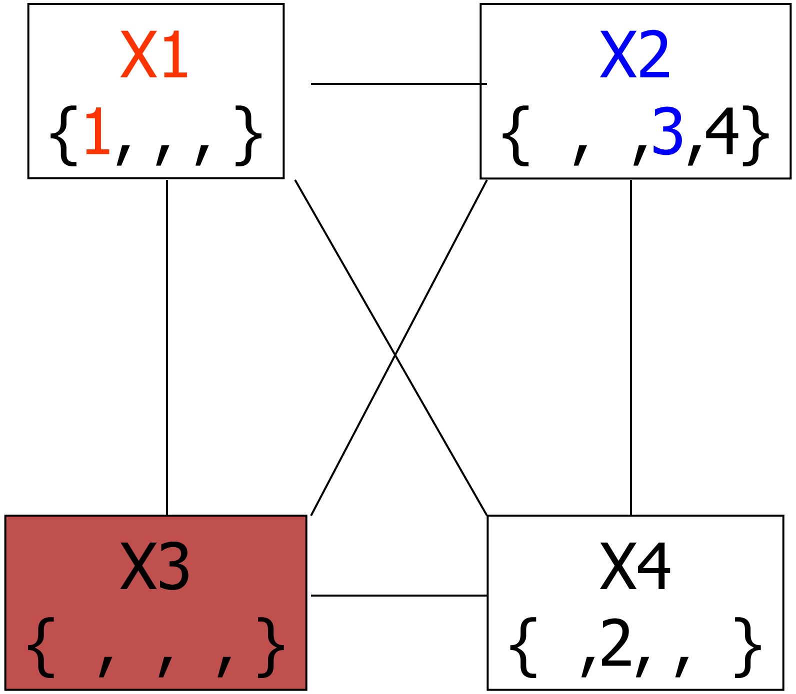
Example: 4-Queens Problem



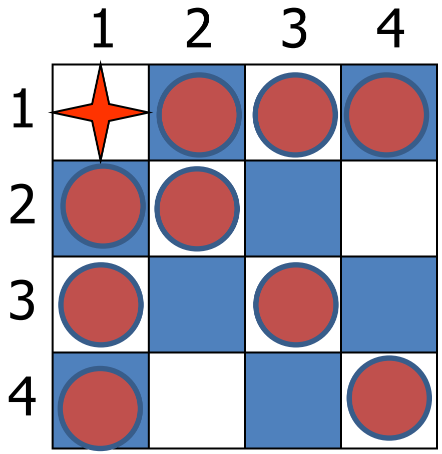


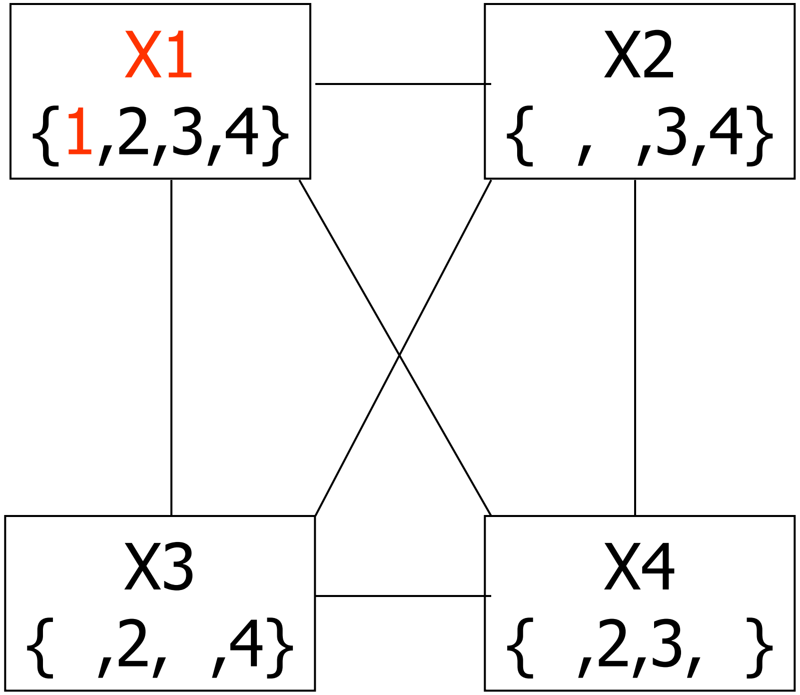
Example: 4-Queens Problem



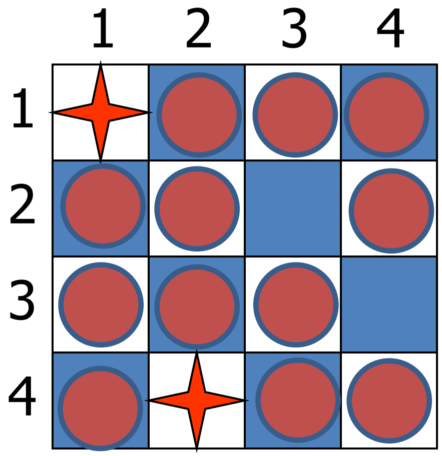


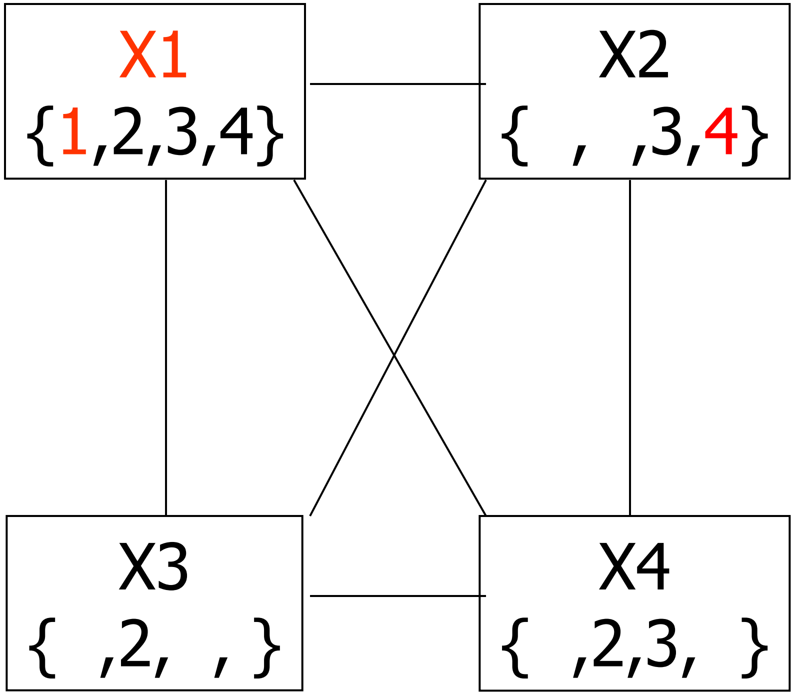
Example: 4-Queens Problem



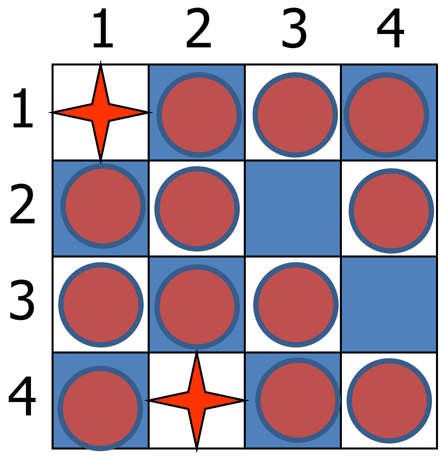


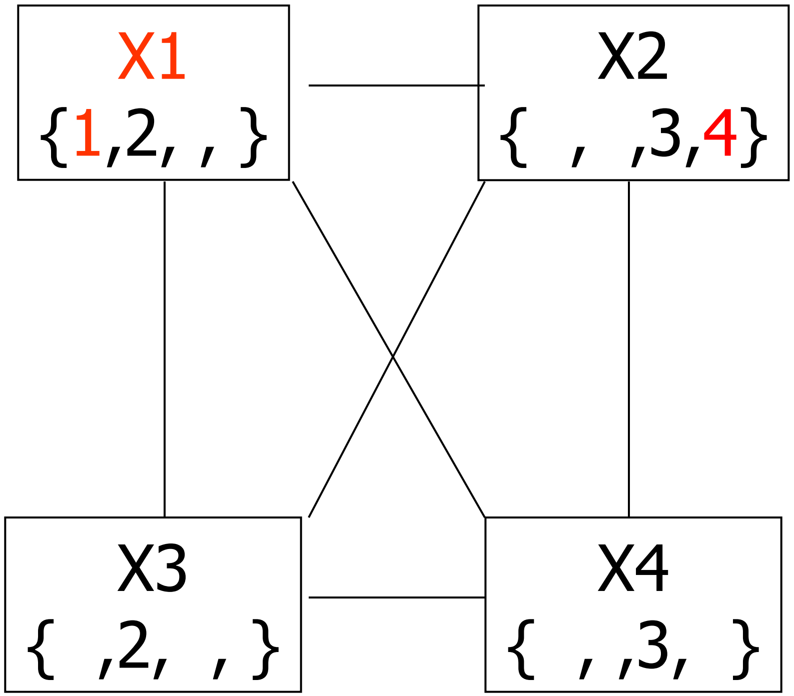
Example: 4-Queens Problem



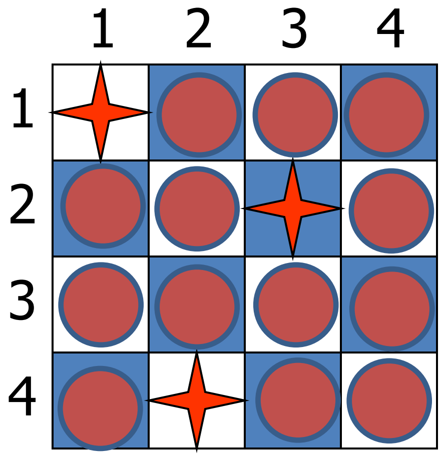


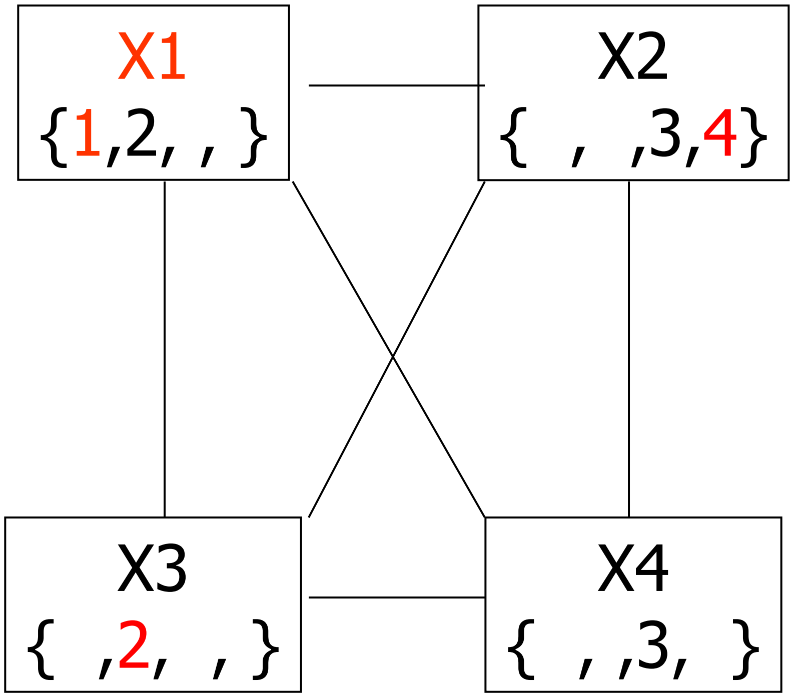
Example: 4-Queens Problem



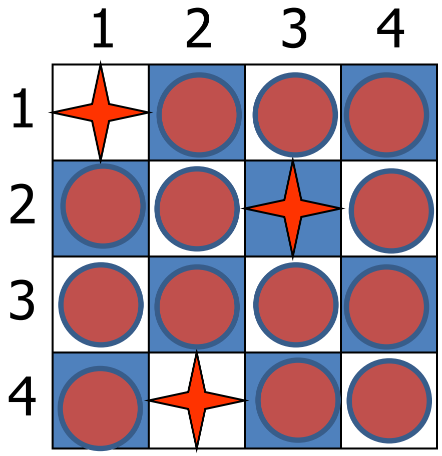


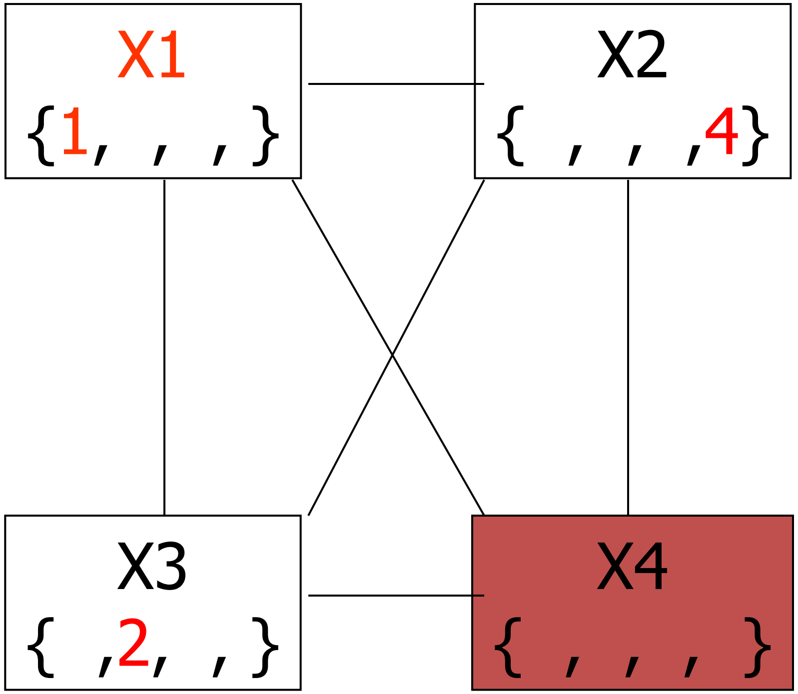
Example: 4-Queens Problem



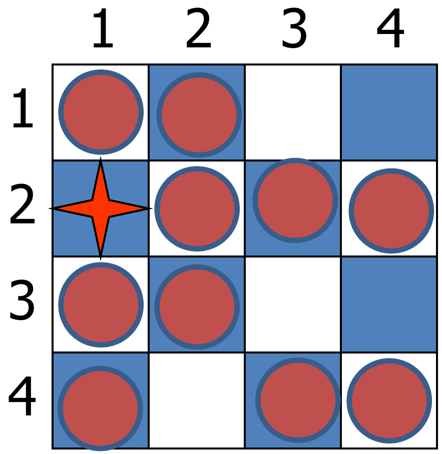


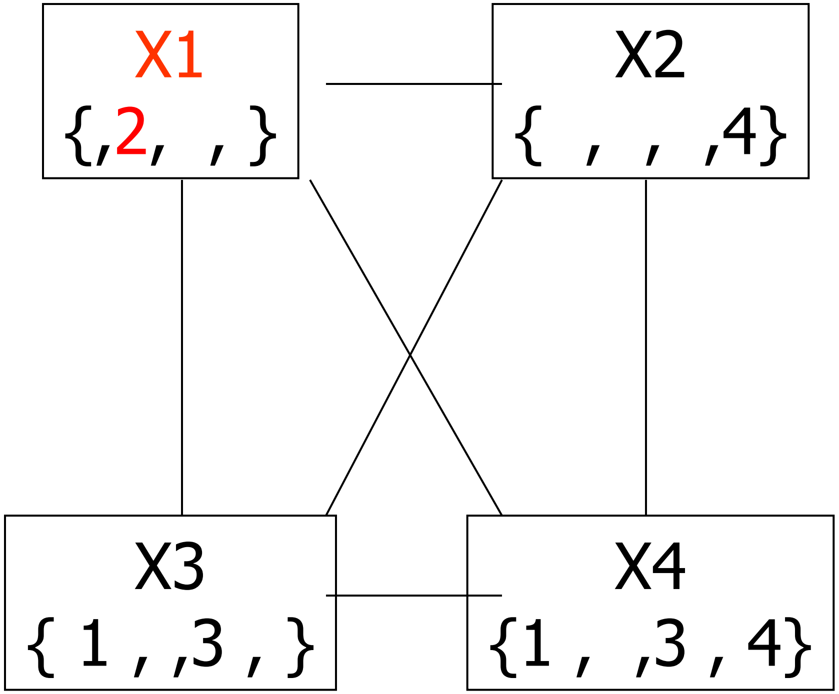
Example: 4-Queens Problem



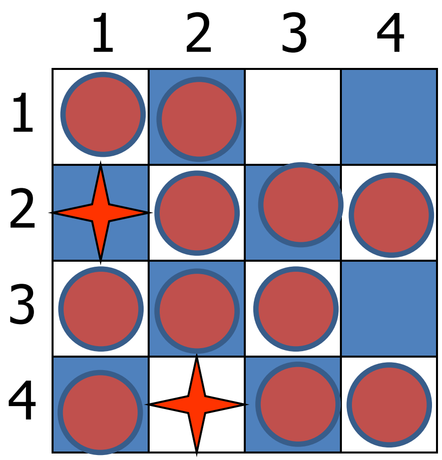


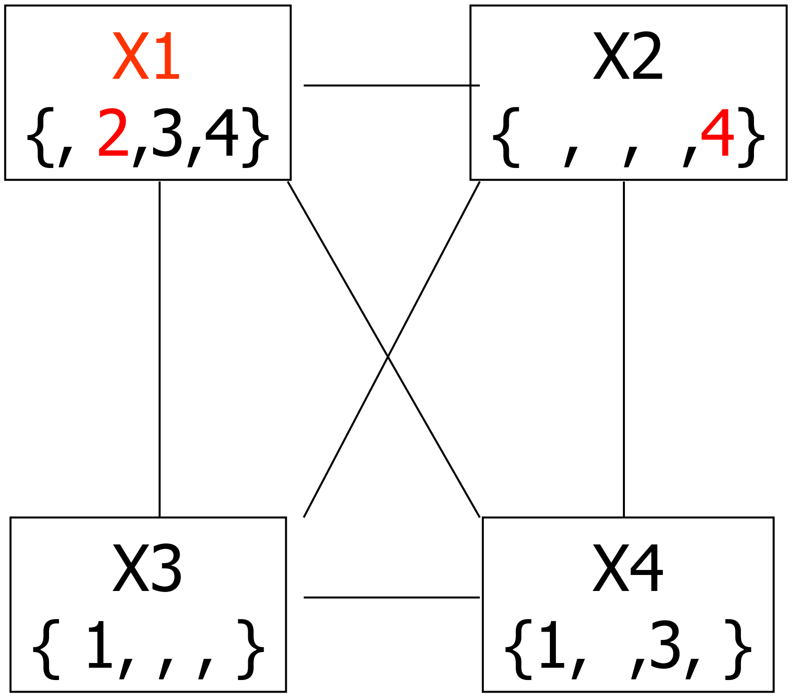
Example: 4-Queens Problem



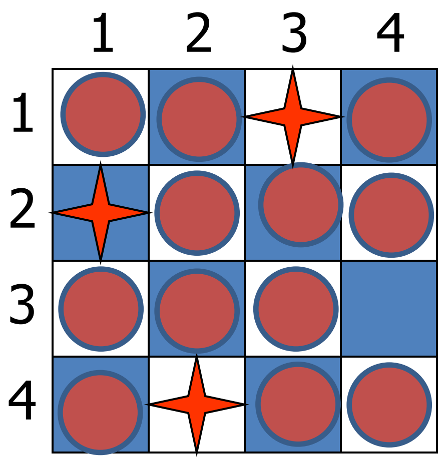


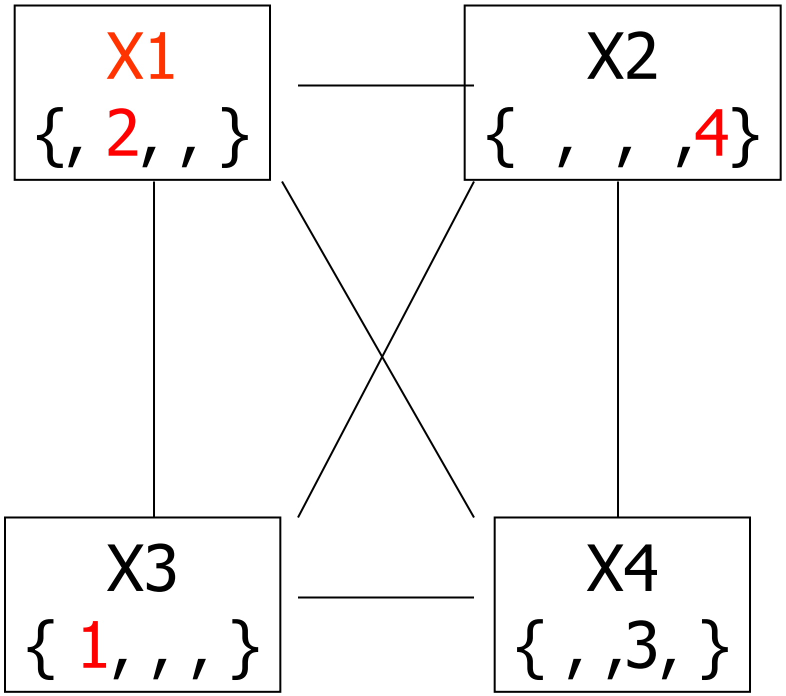
Example: 4-Queens Problem



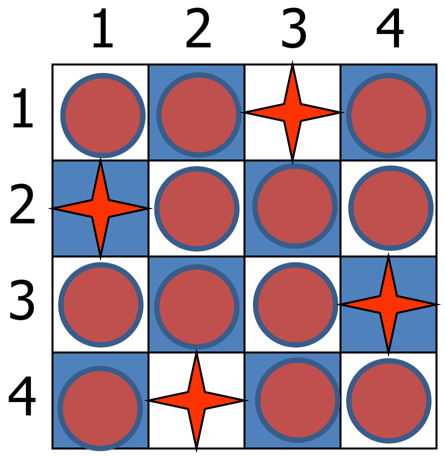


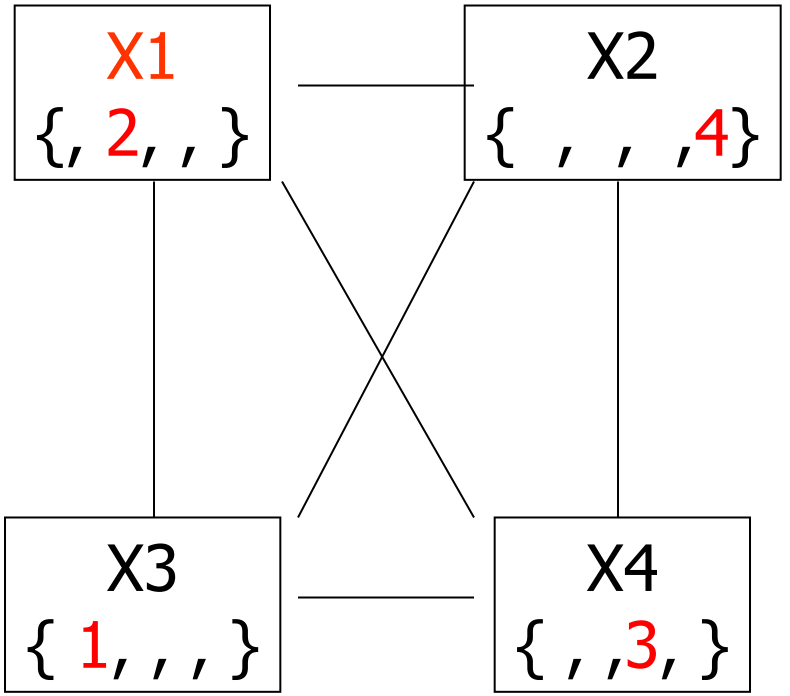
Example: 4-Queens Problem



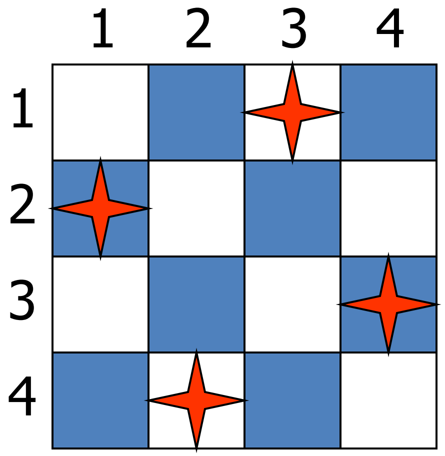


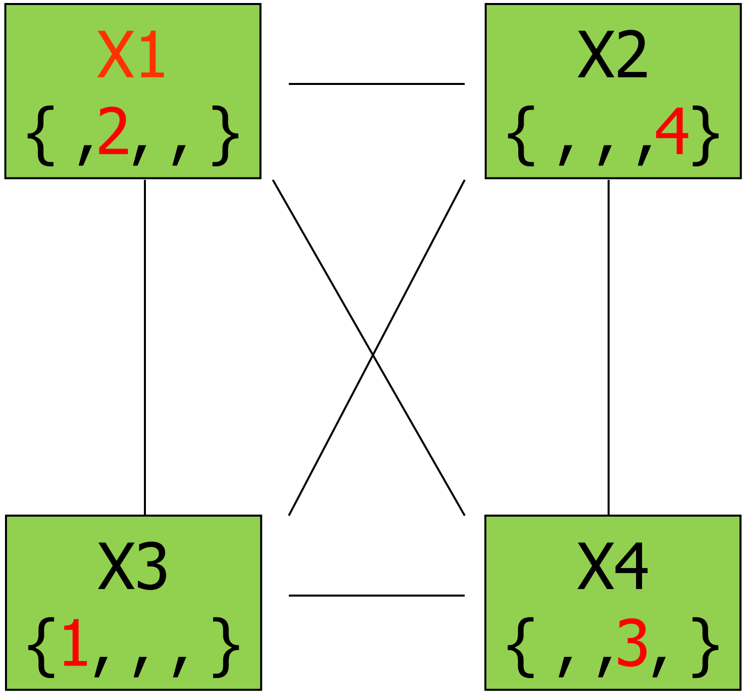
Example: 4-Queens Problem





Example: 4-Queens Problem

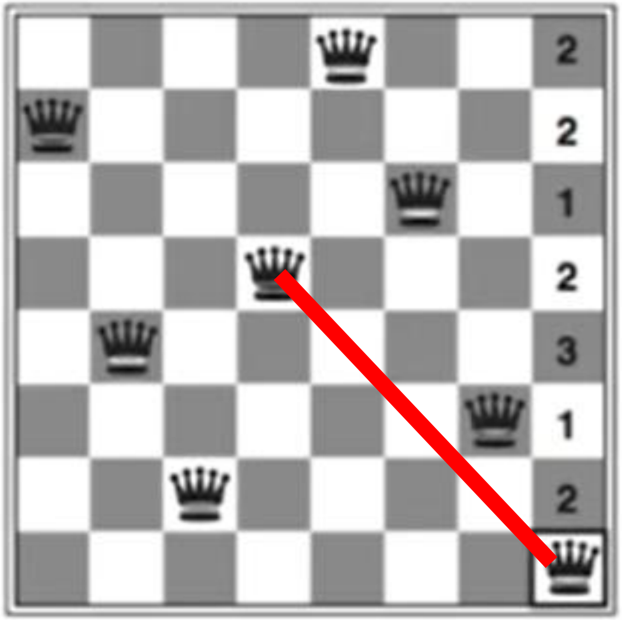




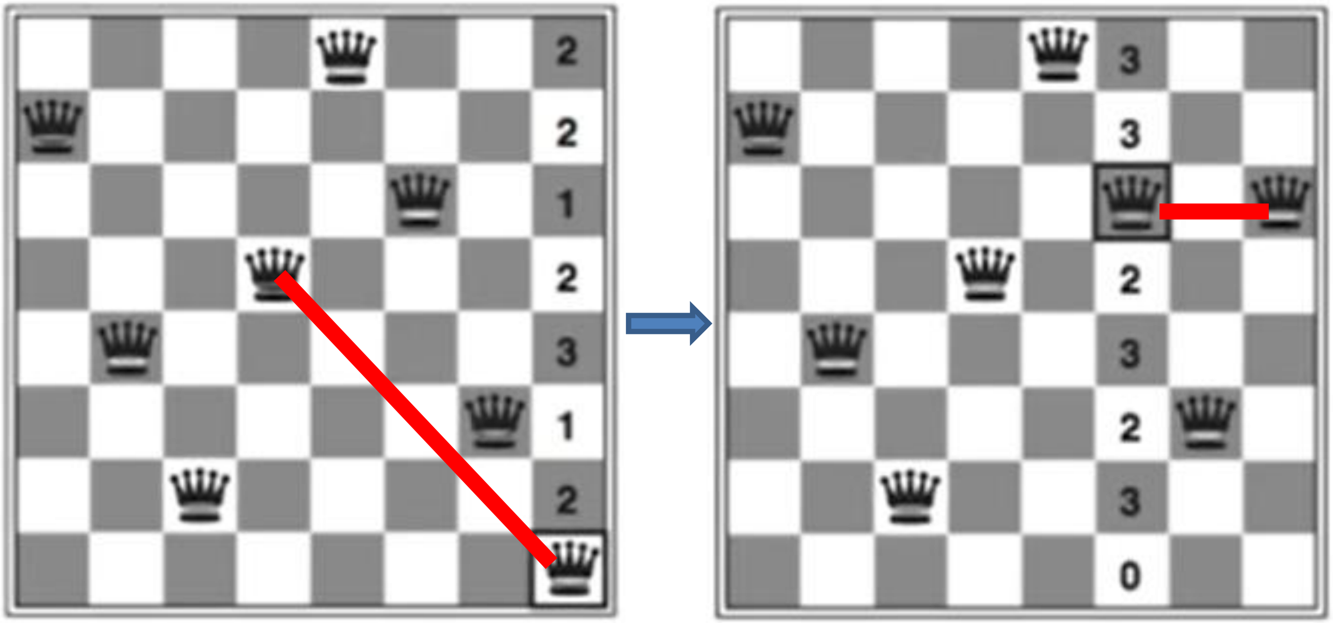
Minimum Conflicts heuristics

```
function MinConflicts(csp,max_steps)
// csp, max_steps is num of steps before giving up
    current = an initial assignment for csp
    for i=1 to max_steps do
        if current is a solution for csp
            return current
        var = a randomly chosen conflicted variable in
   csp
        value = the value v for var that minimizes
   Conflicts
        set var = value in current
    return failure
```

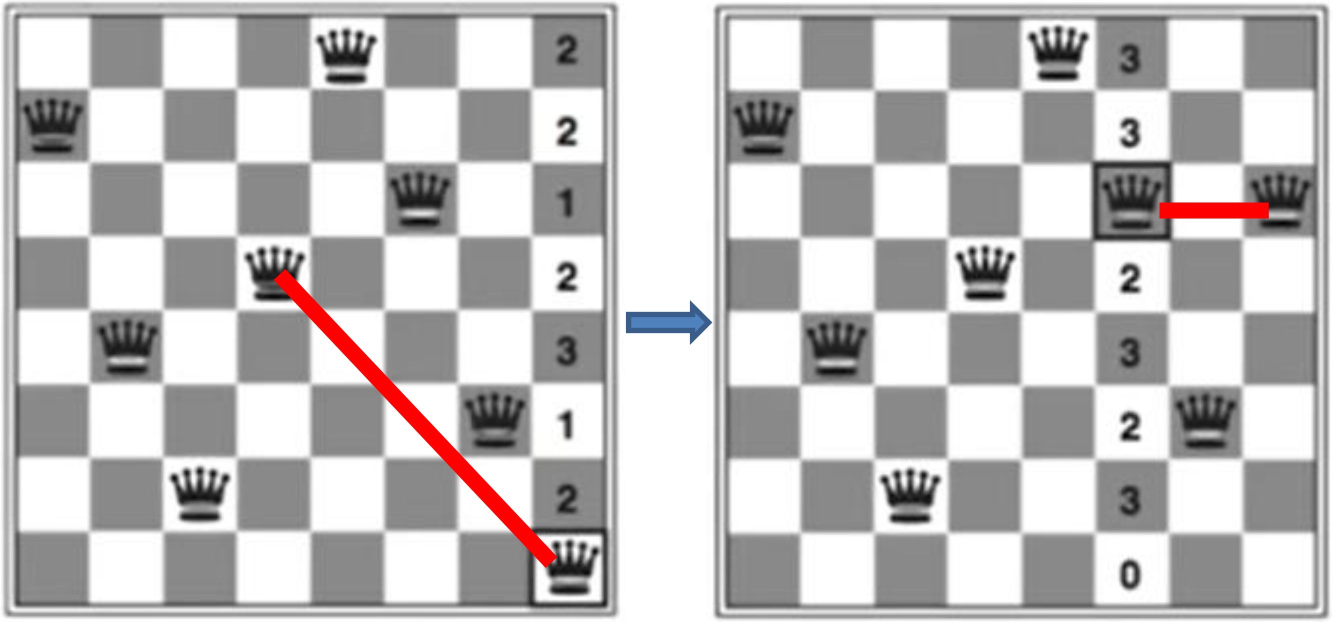
Minimum Conflicts heuristics



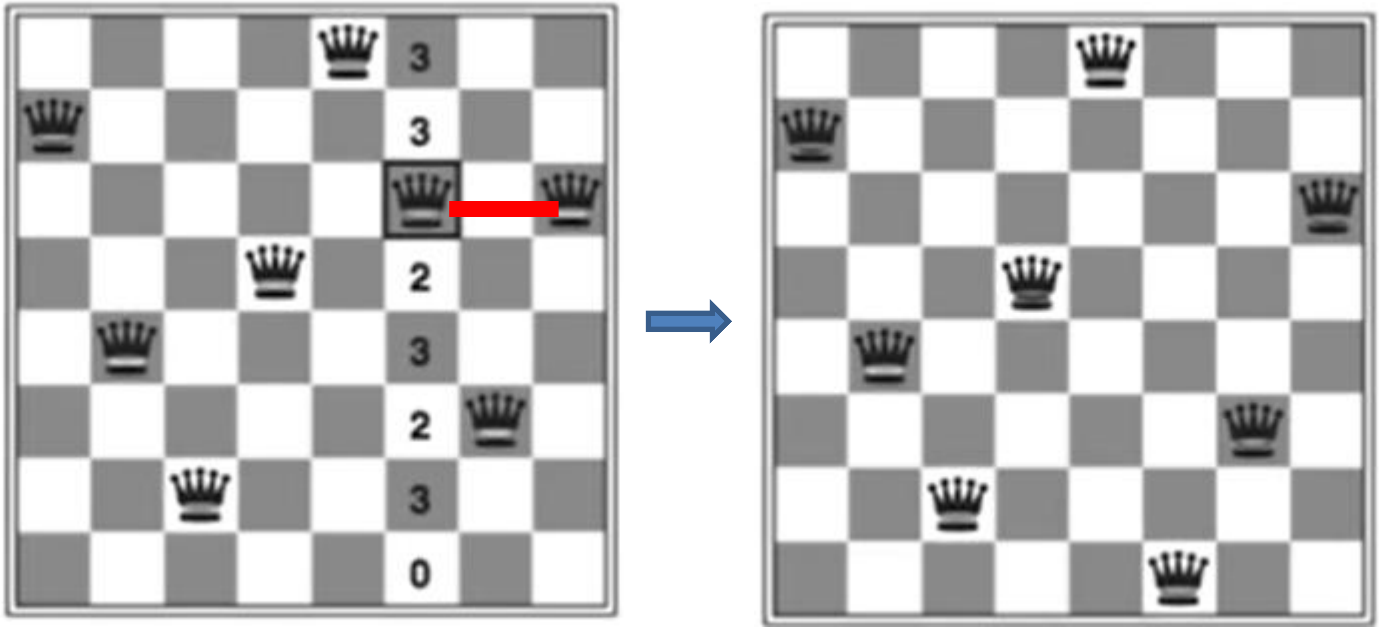
Minimum Conflicts heuristics



Minimum Conflicts heuristics



Minimum Conflicts heuristics



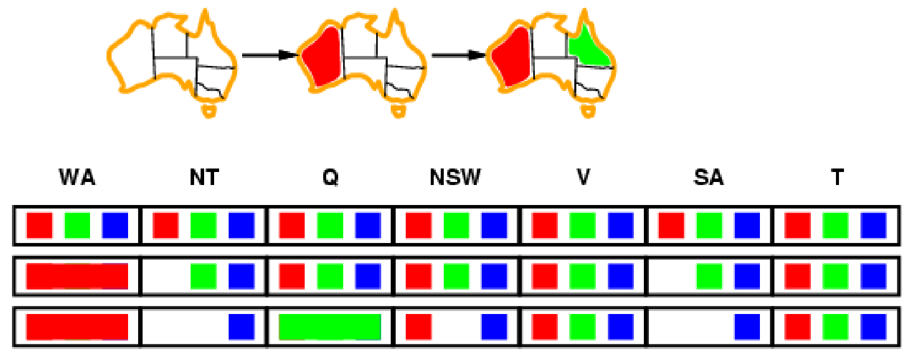
Minimum Conflicts heuristics

Problem	Backtracking	Forward Checking	Min Conflicts
n-queens	> 40,000K	> 40,000K	4K
USA states	> 1,000K	2K	64
Zebra	3,859K	35K	2K

From Russel and Norvig

Constraint propagation

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



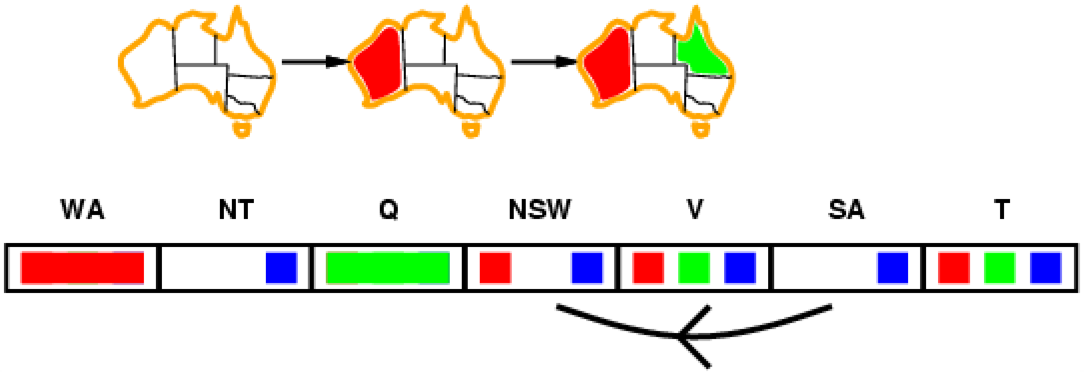
- WA=Red; NT=blue; Q=green; SA=blue
- Forward checking gives us the third row

- At this point, we can see that this is inconsistent, since NT and SA are forced to be blue, yet they are adjacent. Forward checking doesn't see this, and proceeds onward in the search from this state (as we saw earlier)
- NT and SA cannot both be blue!
- Constraint propagation repeatedly enforces constraints locally

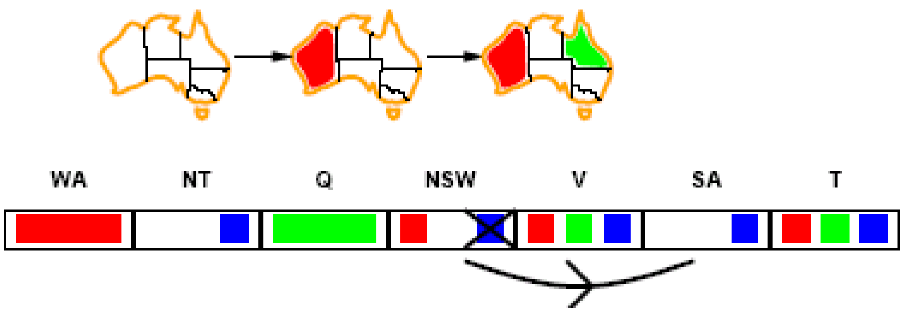
Arc consistency

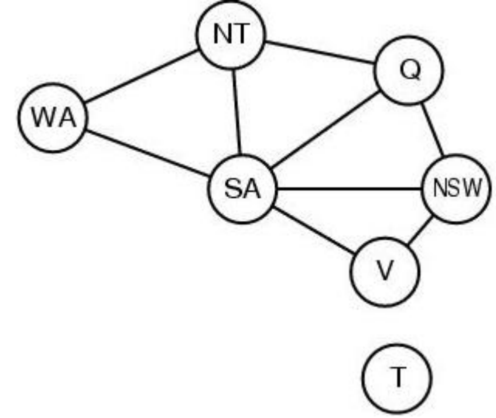
- Simplest form of propagation makes each arc consistent
- *An Arc X → Y* is consistent iff

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



Arc consistency





- *X* → *Y* is consistent if for *every* value *x* of *X* there is some value *y* consistent with *x*
- We will try to make the arc consistent by deleting x's for which there is no y (and then check to see if anything else has been affected – algorithm is in a few slides)

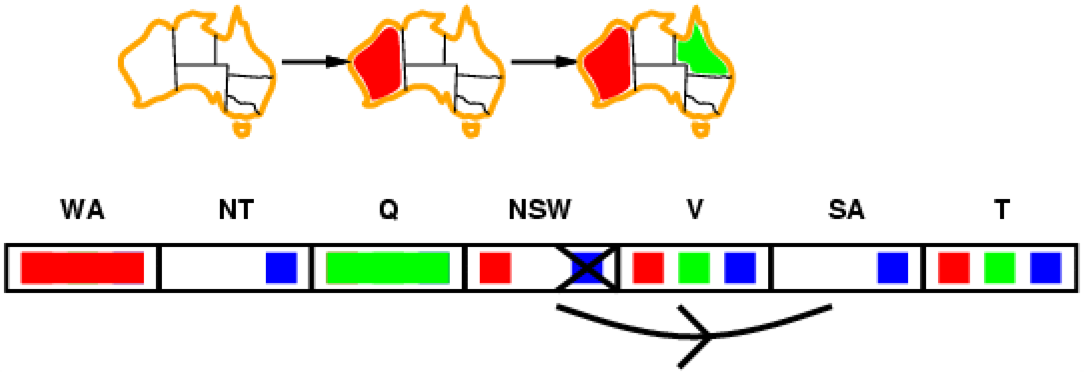
- *NSW → SA: if NSW=red SA could be =blue*
  But, if NSW=blue, there is no color for SA.
  So, remove blue from the domain of NSW
  Propagate the constraint: need to check Q → NSW SA → NSW V → NSW
  If we remove values from any of Q, SA, or V's domains, we will need to check THEIR neighbors

[continue process on next slide and board]

Arc consistency

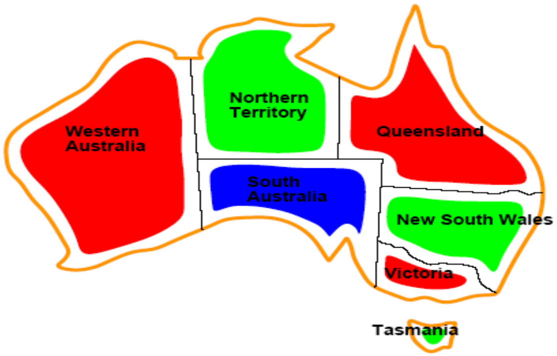
- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value x of X there is some allowed y

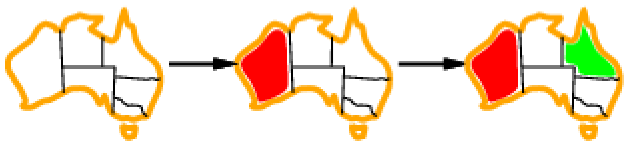


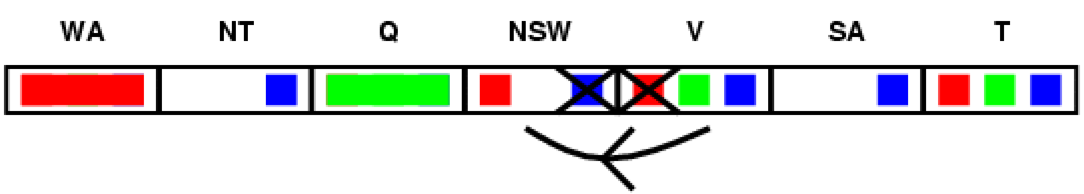
Arc consistency

- 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



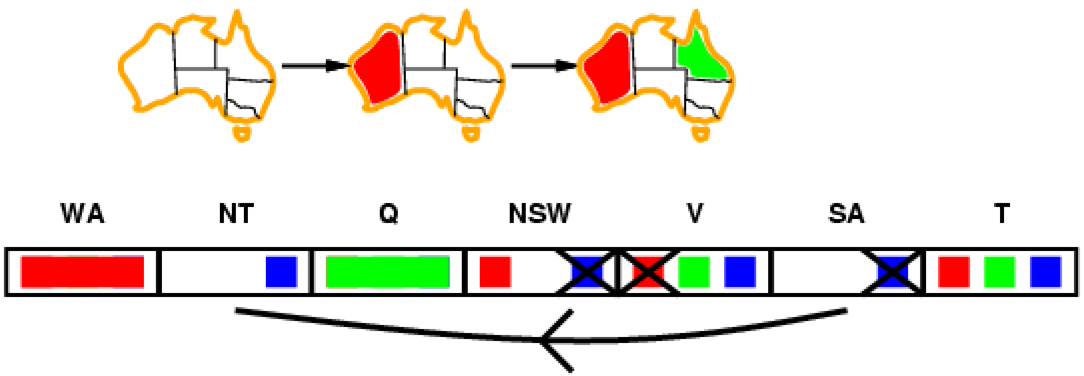


- If X loses a value, neighbors of X need to be rechecked

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: $O(n^2d^3)$

Local search for CSPs

- Hill-climbing and simulated annealing typically work with "complete" states, i.e., all variables assigned

- To apply to CSPs:
  - allow states with unsatisfied constraints
  - operators reassign variable values

- Variable selection: randomly select any conflicted variable

- Value selection by min-conflicts heuristic:
  - choose value that violates the fewest constraints
  - i.e., hill-climb with *h(n)* = total number of violated constraints

Solving CSPs (backtrack) with combination of heuristics plus forward checking is more efficient than either approach alone

But Forward Checking does not see all inconsistencies

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
- Variable ordering and value selection 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
- Iterative min-conflicts is usually effective in practice

University of Colorado
Colorado Springs

University of Colorado

Boulder | Colorado Springs | Denver | Anschutz Medical Campus