Constraint Satisfaction problems

Dr.Sonali Patil

Outline

- CSP?
- Backtracking for CSP
- Local search for CSPs
- Problem structure and decomposition

Objectives

- Introduction to class of problems called constraint satisfaction problems
- Expressing different constrains formally
- Model CSP problems as search problems and use of DFS with backtracking to solve these problems
- Casting different CSP problems as search problems

Introduction

- Many problems in AI are CSP
  - Satisfiability
  - Scheduling
  - Timetabling
  - Graph Colouring
  - Puzzles
  - etc

Constraint satisfaction problems

- A CSP consists of
  - **Finite set of variables $V_1, V_2, ..., V_n$**
  - **Finite set of constrains $C_1, C_2, ..., C_n$**
  - **Nonemtpy domain of possible values for each variable $D_{V1}, D_{V2}, ... D_{Vn}$**
  - **Each constraint C_i limits the values that variables can take, e.g., $V_1 \neq V_2$ (**

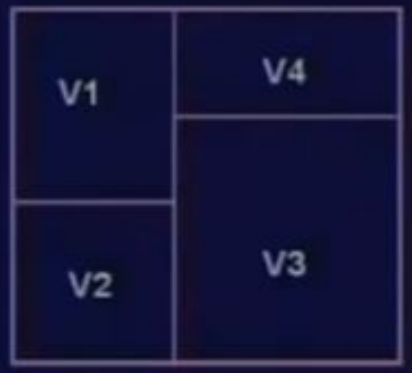
    (Vi > Vj Binary constraint) (Vi is set of even integers Unary Constraint) (Also multiple variable constraint)
- A *state* is defined as an *assignment* of values to some or all variables.
- *A Solution* is an assignment of a value in D_{vi} to each variable Vi such that every constraint is satisfied

Constraint satisfaction problems

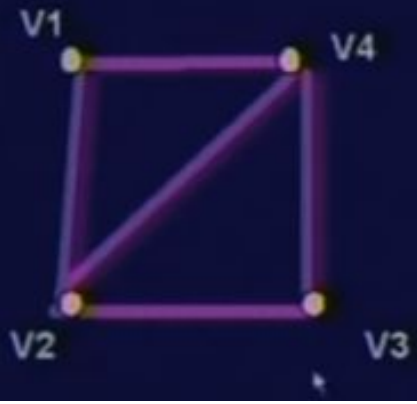
- An assignment is *complete* when every value is mentioned.
- A *solution* to a CSP is a complete assignment that satisfies all constraints.
- Some CSPs require a solution that maximizes an *objective function*.
- Applications: Scheduling the time of observations on the Hubble Space Telescope, Floor planning, Map coloring, Cryptography

Colouring as CSP

- Can we colour all 4 regions with 3 colours so that no two adjacent regions are the same colour?



- Variable for each node
  - All D_i = { red, green, blue}
- Constraint for each edge
  - all constraints of the form
    - $x_i \neq x_j$
- Solution gives a colouring
- It's a binary CSP



SAT (satisfiability problems in propositional logic) As CSP

- Variable in CSP for each variable/letter in SAT
- Each domain D_i = {true, false}
- Constraint corresponds to each clause
  - disallows unique tuple which falsifies clause
  - e.g. (not A) or (B) or (not C)
    - $\Rightarrow$ not < A = true, B = false, C = true >
- Not binary CSP unless all clauses 2-clauses

N-Queens Problem as CSP

- Chessboard puzzle
  - place 8 queens on a 8x8 chessboard so that no two attack each other
- Variable x_i for each row i of the board
- Domain = {1, 2, 3 … , n} for position in row

- Variable x_i for each row i of the board
- Domain = {1, 2, 3 … , n}
- Constraints are:
  - $x_i \neq x_j$ queens not in same column
  - $x_i - x_j \neq i-j$ queens not in same SE diagonal
  - $x_j - x_i \neq i - j$ queens not in SW diagonal

Formal Definition of Constraints

A constraint $C_{ijk...}$ involving variables $x_i, x_j, x_k ...$
- is any subset of combinations of values from $D_i, D_j, D_k ...$
- I.e. $C_{ijk...} \subseteq D_i \times D_j \times D_k ...$
- indicating the *allowed* set of values

- A number of ways to write constraints:
  - e.g. if $D_1 = D_2 = \{1,2,3\}$ ...
    - { (1,2), (1,3), (2,1), (2,3), (3,1), (3,2) }
    - $x_1 \neq x_2$

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. cryptharithmetic column constraints.**
- Preference (soft constraints) e.g. *red* is better than *green*
  - often representable by a cost for each variable assignment
  - → constrained optimization problems.

More complex constraints

- Cryptarithmetic: all letters different and sum correct
- Variables are D, E, M, N, O, R, S, Y
- Domains:
  - {0,1,2,3, … , 9} for D, E, N, O, R, Y
  - {1,2,3, … , 9} for S, M

```
  SEND
+ MORE
------
 MONEY
```

More complex constraints

Send + More = Money

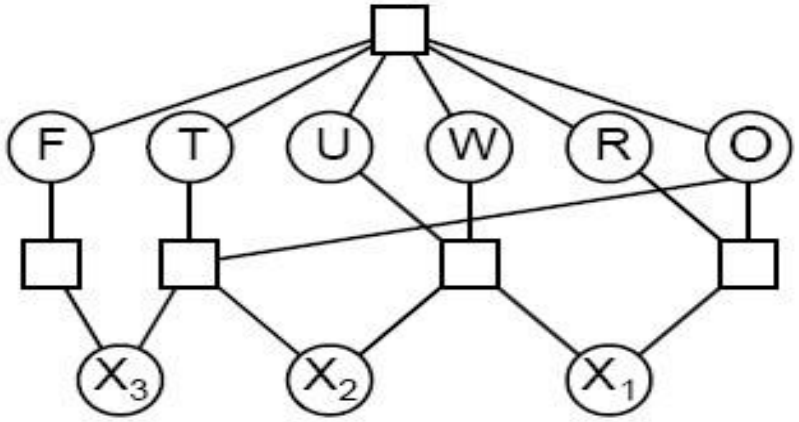
- We can write one long constraint for the sum:
  - 1000*S + 100*E + 10*N + D
  + 1000*M + 100*O + 10*R + E
  = 10000*M + 1000*O + 100*N + 10*E + Y

More complex constraints

- But what about the difference between variables?
  - Could write S =/= E, M=/=O … R =/= N
  - Or express it as a single constraint on all variables
  - *AllDifferent* (D, E, M, N, O, R, S, Y)
- These two constraints
  - express the problem precisely

Example: cryptharithmetic

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



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

$alldiff(F, T, U, W, R, O)$

$O + O = R + 10 \cdot X_1$, etc.

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

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

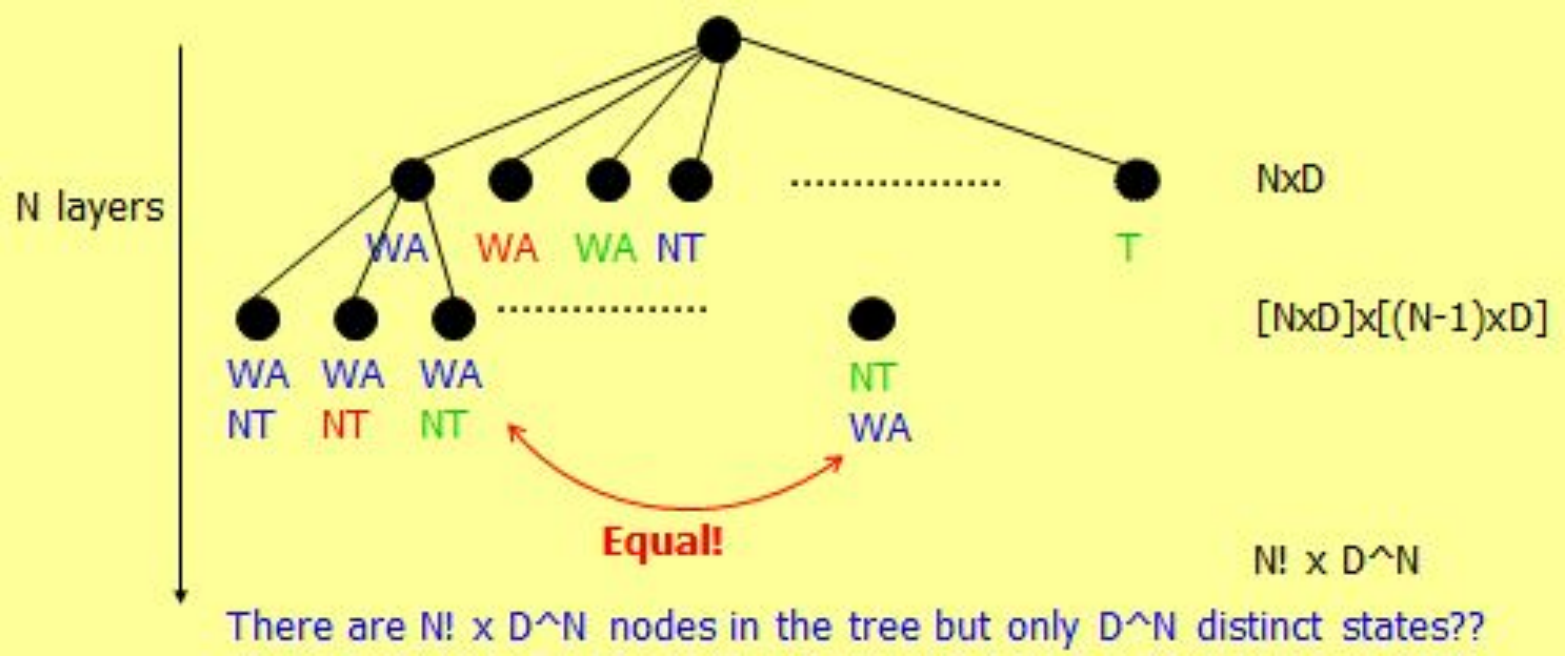
$X_3 = F$, $T \neq 0$, $F \neq 0$

Constrain Satisfaction

- Kind of search in which
  - States are factored into sets of variables
  - Search = assigning values to these variables
  - Structure of space is encoded with constraints
- Backtracking-style algorithms work e.g. DFS

  But other techniques add speed
  - Propagation
  - Variable ordering
  - Preprocessing

CSP as a standard search problem



Commutativity

- CSPs are commutative.
  - **The order of any given set of actions has no effect on the outcome.**
  - **Example: choose colors for Australian territories one at a time**
    - [WA=red then NT=green] same as [NT=green then WA=red]
    - All CSP search algorithms consider a single variable assignment at a time $\Rightarrow$ there are d^n leaves.

Backtracking search

- Depth-first search
- Chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.
- Uninformed algorithm
  - **No good general performance**

Backtracking search

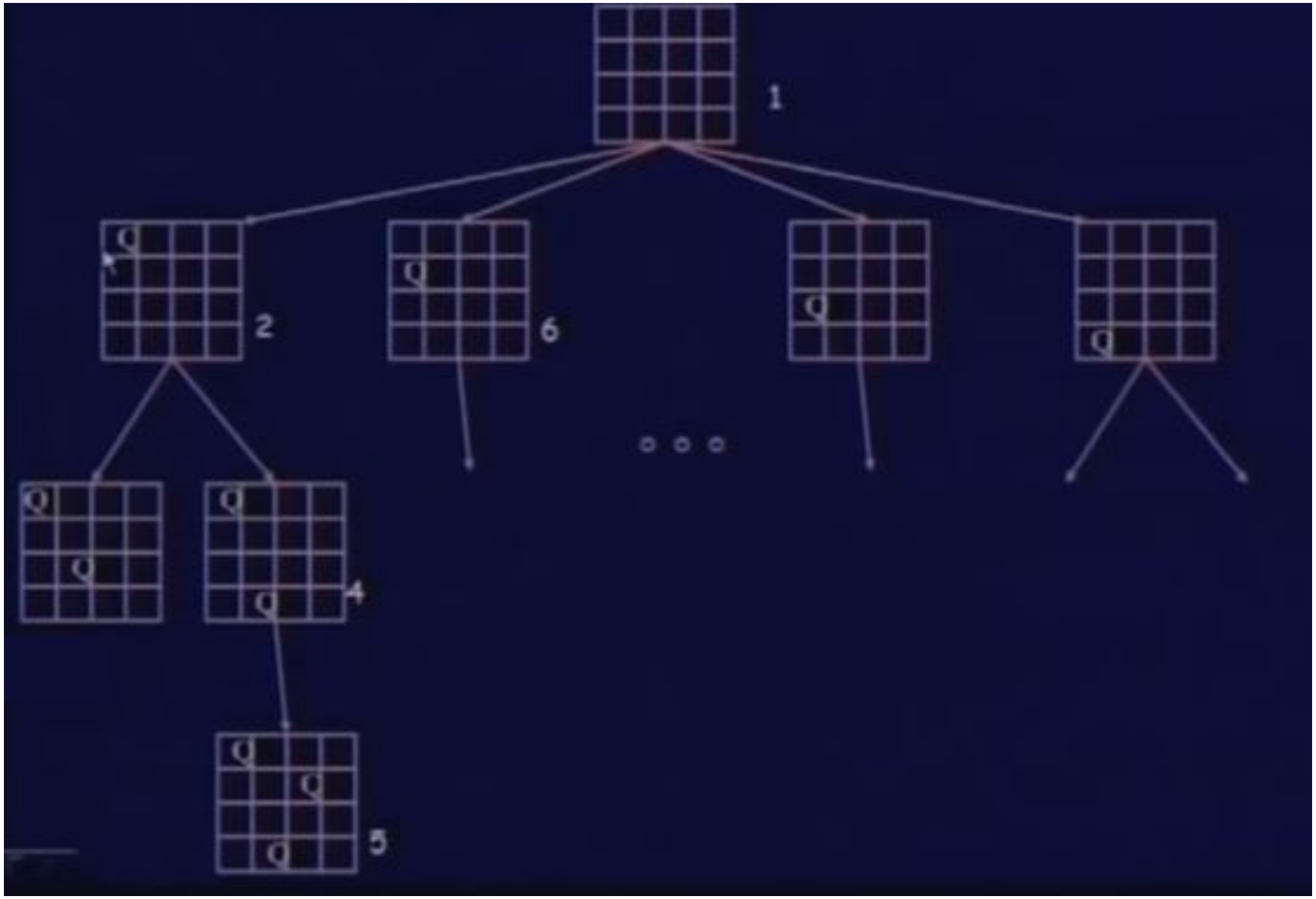
function BACKTRACKING-SEARCH(*csp*) **return** a solution or failure
    return RECURSIVE-BACKTRACKING(*{} , csp*)

function RECURSIVE-BACKTRACKING(*assignment, csp*) **return** 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 ← RRECURSIVE-BACTRACKING(*assignment, csp*)
            if *result* ≠ *failure* **then return** *result*
            remove *{var=value}* from *assignment*
    return *failure*

CSP as a search Problem

- What are states?
  - (nodes in graph)
- What are the operators?
  - (arcs between nodes)
- Initial state?
- Goal test?

CSP as a search Problem



CSP : map coloring

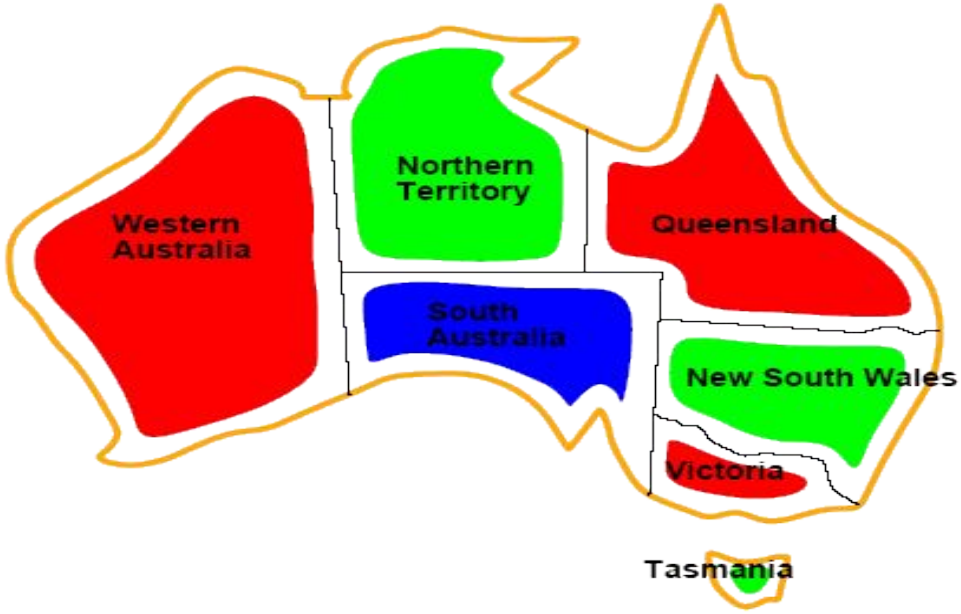
Variables: *WA, NT, Q, NSW, V, SA, T*

Domains: D_i=*{red,green,blue}*

Constraints:adjacent regions must have different colors.

- E.g. *WA* ≠ *NT* (if the language allows this)
- E.g. *(WA,NT)* ≠ *{(red,green),(red,blue),(green,red),…}*

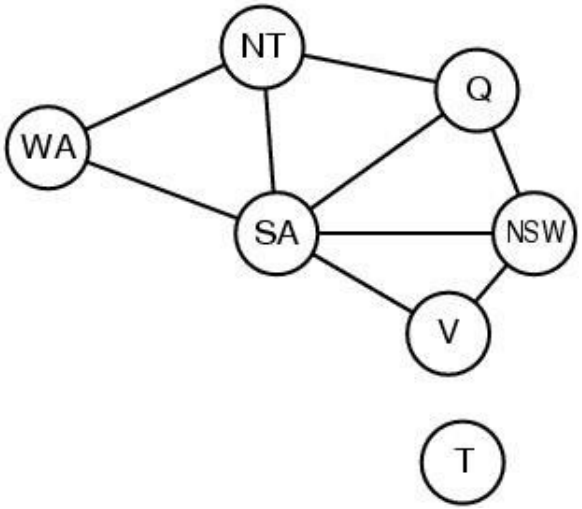
CSP example: map coloring



Solutions are assignments satisfying all constraints, e.g.

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

Constraint graph



CSP benefits

- **Standard representation pattern**
- **Generic goal and successor**
- **functions**

Generic heuristics (no domain specific expertise).

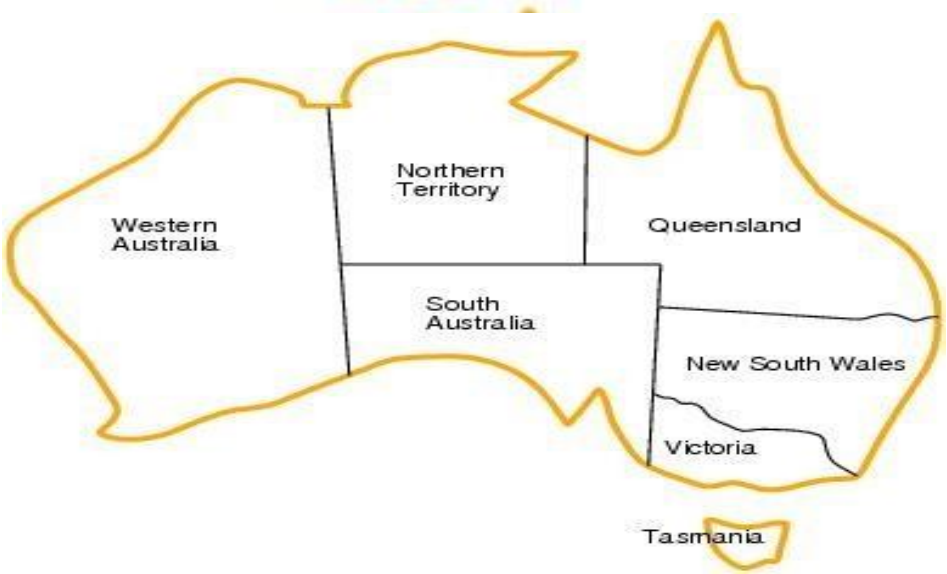
Binary CSP: each constraint relates two variables

Constraint graph = nodes are variables,arcs/edges show constraints.

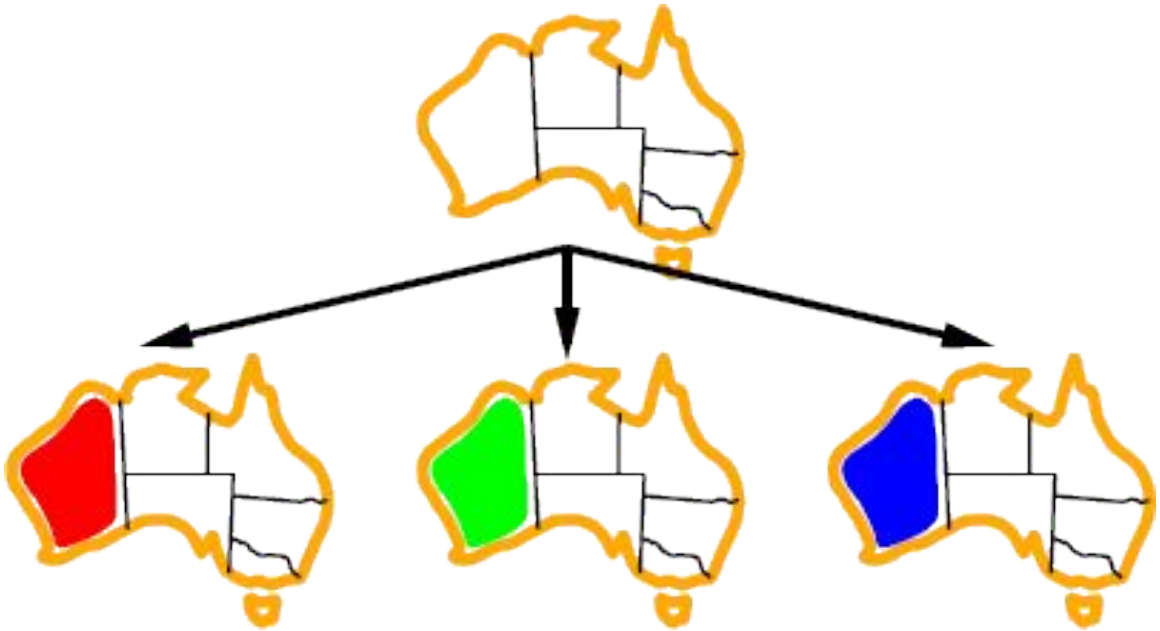
- **Graph can be used to simplify search.**

  e.g. Tasmania is an independent subproblem.

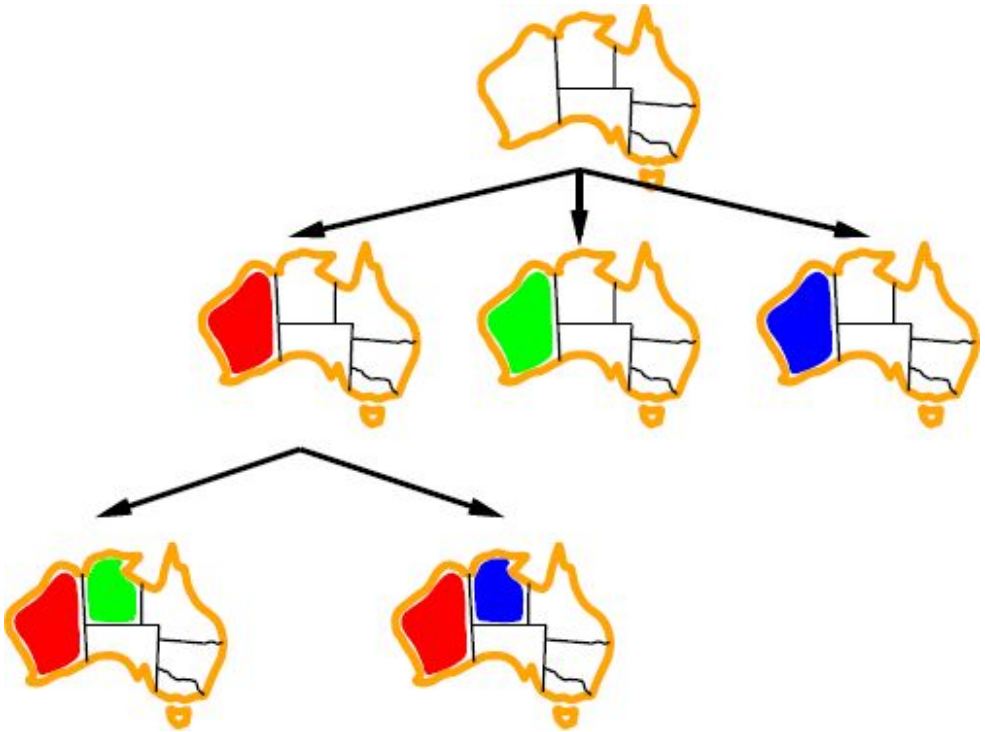
Backtracking example



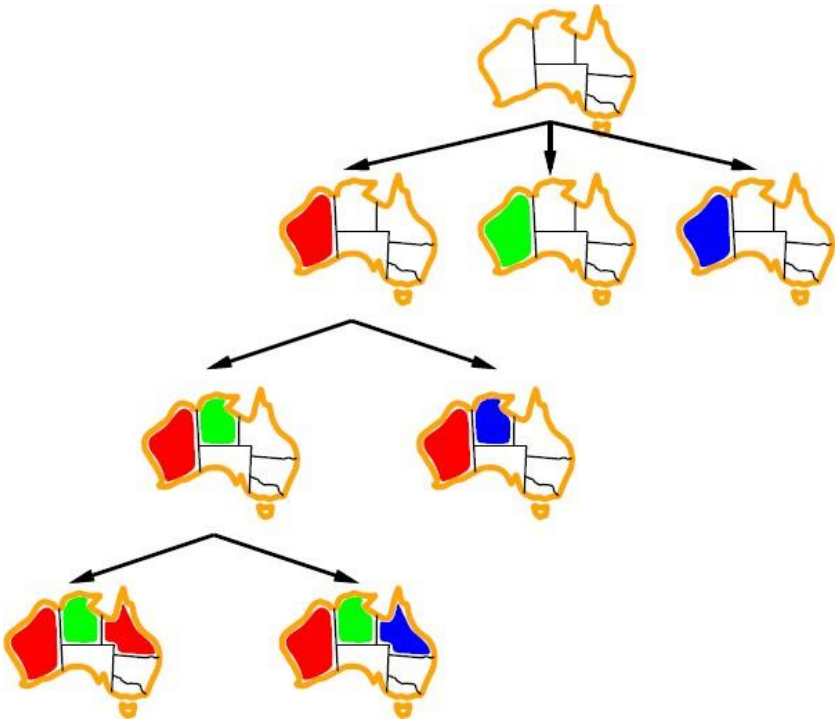
Backtracking example



Backtracking example



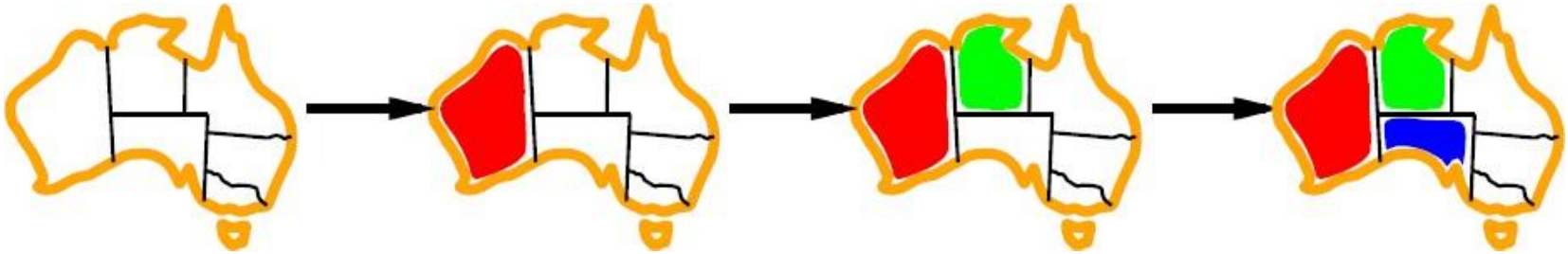
Backtracking example



Improving backtracking efficiency

- Previous improvements → introduce heuristics
- General-purpose methods can give huge gains in speed:
  - **Which variable should be assigned next? (MRV)**
  - **In what order should its values be tried? (LCV)**
  - **Can we detect inevitable failure early?**
  - **Can we take advantage of problem structure?**

Minimum remaining values (MRV)

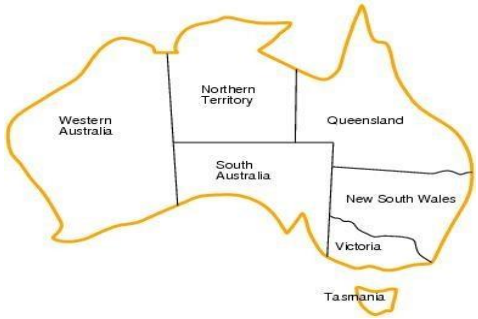


var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*],*assignment*,*csp*)

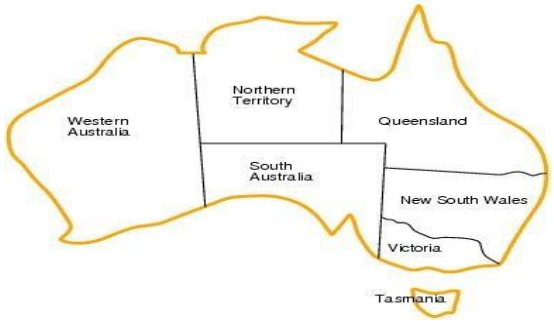
A.k.a. most constrained variable heuristic

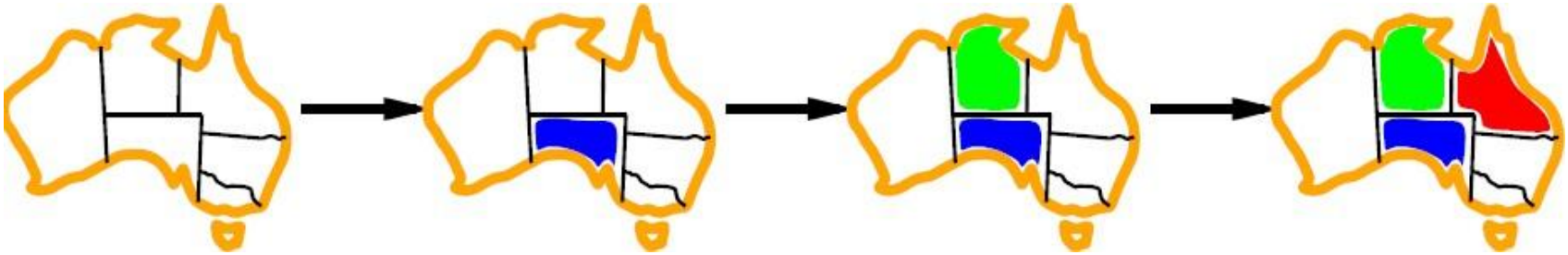
Rule: choose variable with the fewest legal moves

Which variable shall we try first?



Degree heuristic





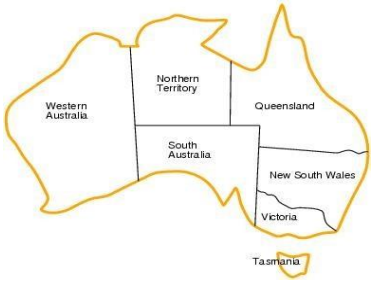
Use degree heuristic

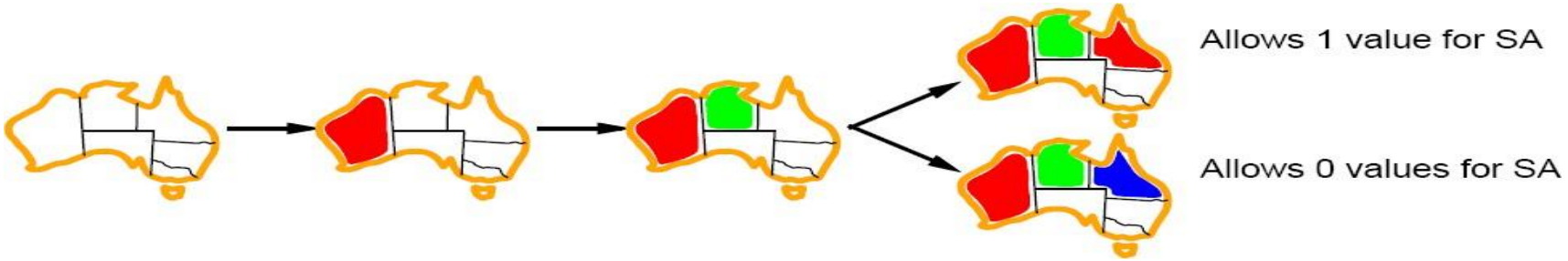
Rule: select variable that is involved in the largest number of constraints on other unassigned variables.(most edges in the graph)

Degree heuristic is very useful as a tie breaker.

In what order should its values be tried?

Least constraining value





Least constraining value heuristic

Rule: given a variable choose the least constraing value i.e. the one that leaves the maximum flexibility for subsequent variable assignments.

Combining these heuristics makes 1000 queens feasible

Rationale for MRV, DH, LCV

In all cases we want to enter the most promising branch, but we also want to detect inevitable failure as soon as possible.

- MRV+DH: the variable that is most likely to cause failure in a branch is assigned first. The variable must be assigned at some point, so if it is doomed to fail, we'd better found out soon. E.g X1-X2-X3, values 0,1, neighbors cannot be the same.
- LCV: tries to avoid failure by assigning values that leave maximal flexibility for the remaining variables. We want our search to succeed as soon as possible, so given some ordering, we want to find the successful branch.

Forward checking

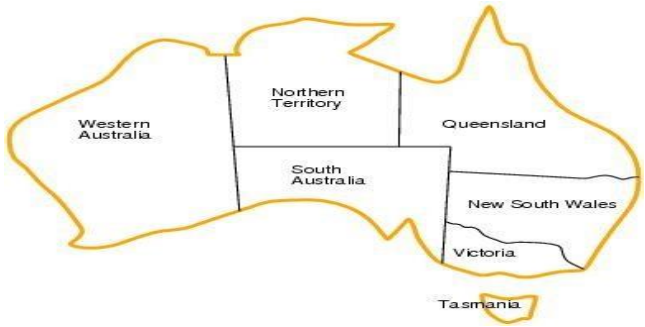
Can we detect inevitable failure early?

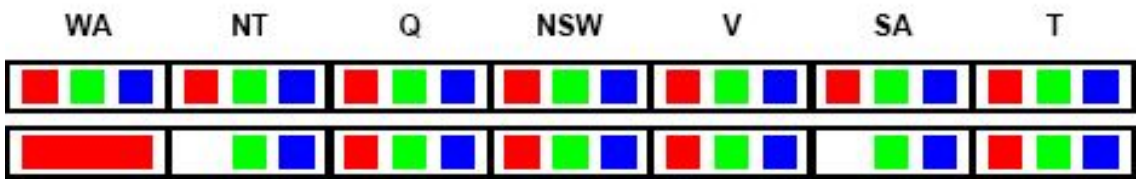
- ***And avoid it later?***

Forward checking idea: keep track of remaining legal values for unassigned variables (rather than just for assigned variables).

Terminate search when any variable has no legal values.

Forward checking



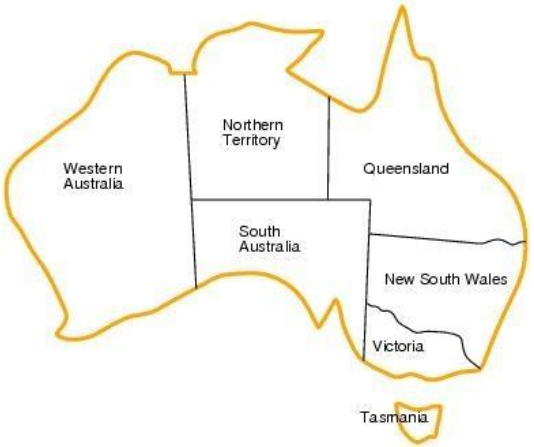


Assign *{WA=red}*

Effects on other variables connected by constraints with WA

- ***NT can no longer be***
- ***red SA can no longer be red***

Forward checking



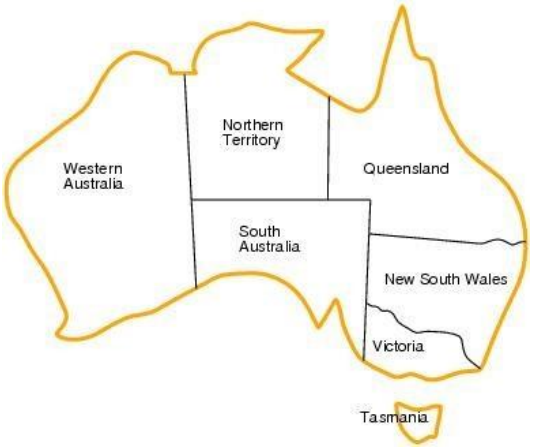
Assign *{Q=green}*

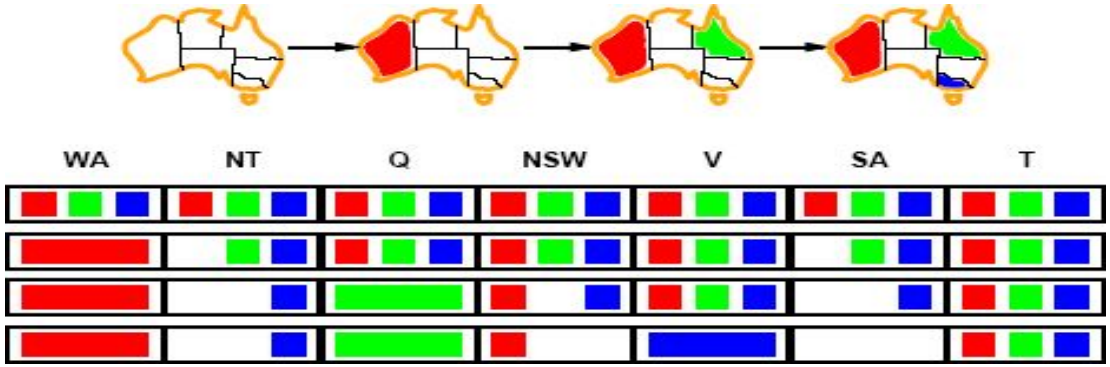
Effects on other variables connected by constraints with WA

- ***NT can no longer be green***
- ***NSW can no longer be***
- ***green SA can no longer be green***

MRV heuristic will automatically select NT and SA next, why?

Forward checking





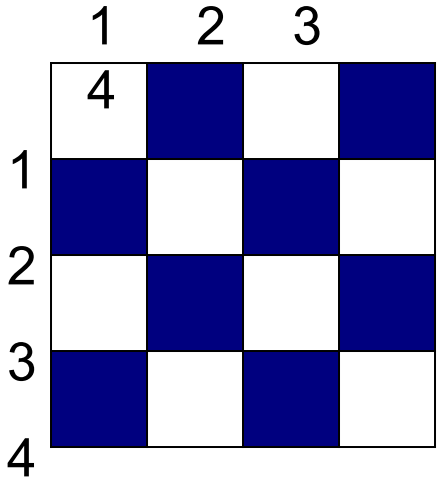
If *V* is assigned *blue*

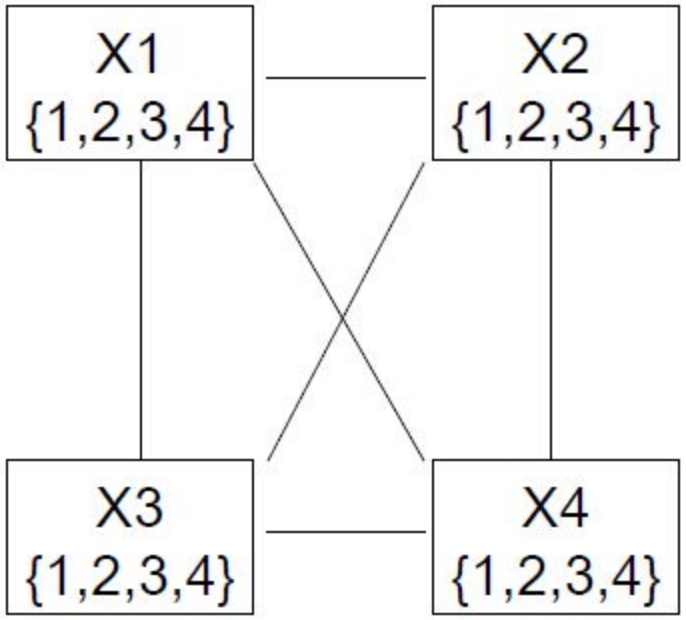
Effects on other variables connected by constraints with WA

- ***SA is empty***
- ***NSW can no longer be blue***

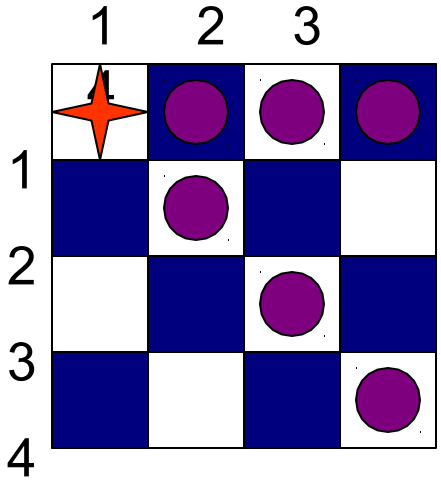
FC has detected that partial assignment is *inconsistent* with the constraints and backtracking can occur.

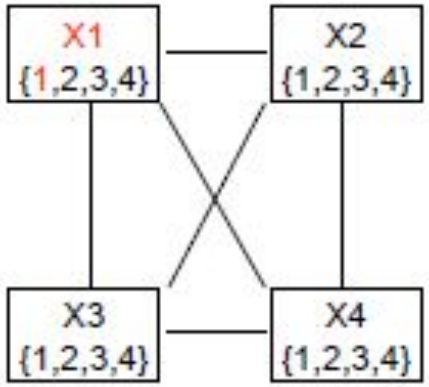
Example: 4-Queens Problem



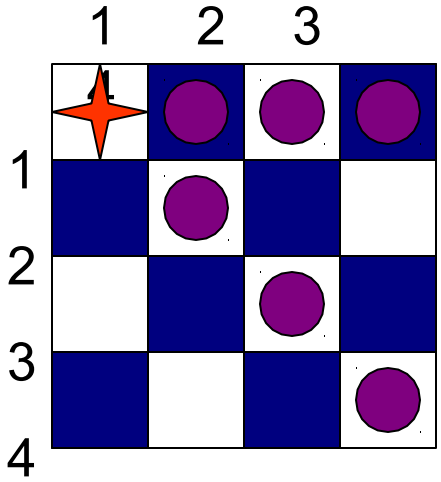


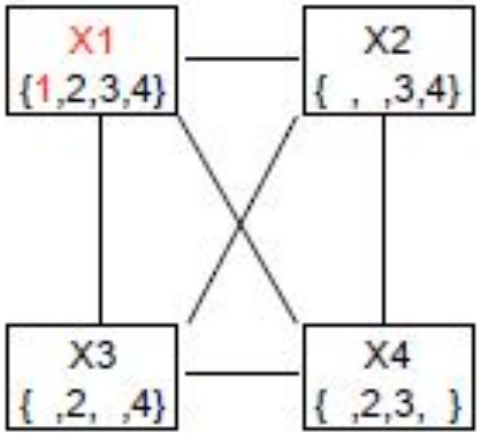
Example: 4-Queens Problem



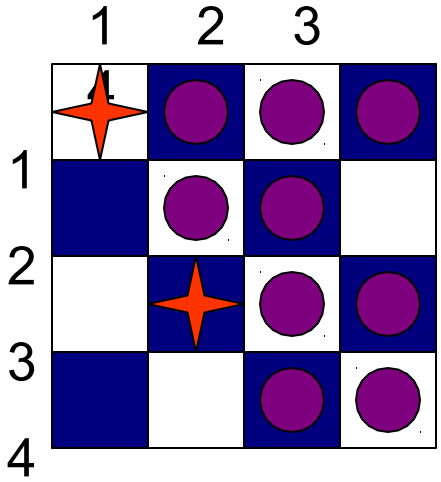


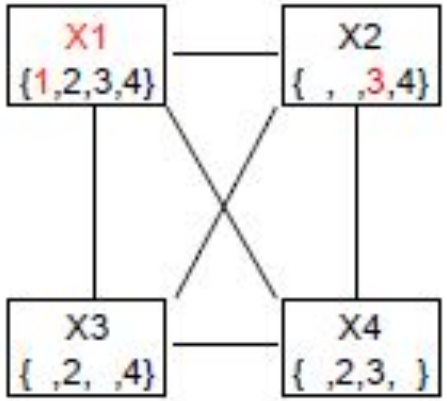
Example: 4-Queens Problem



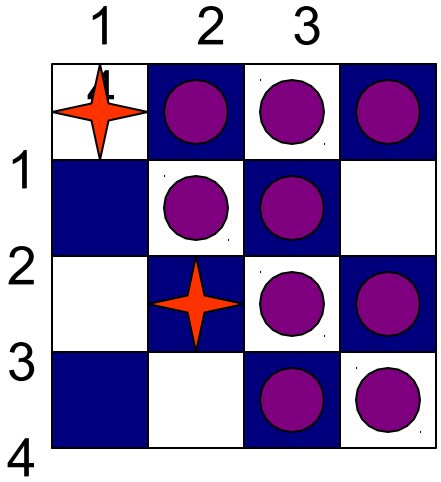


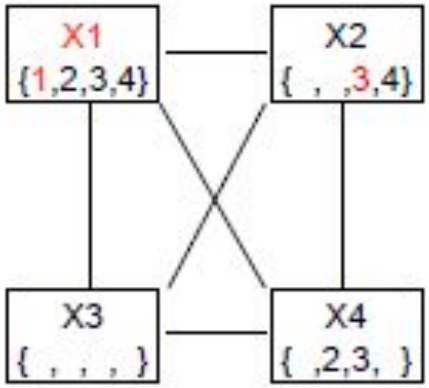
Example: 4-Queens Problem



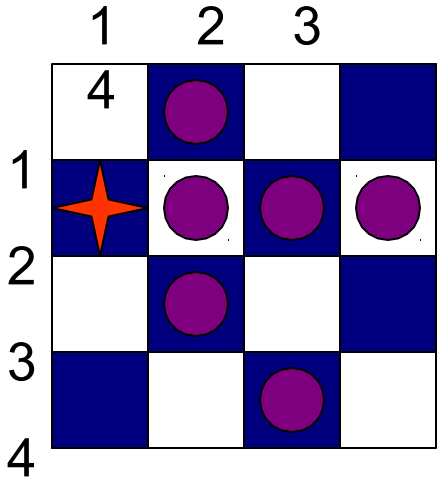


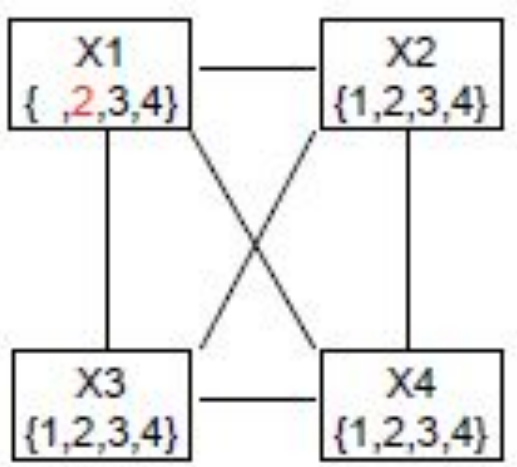
Example: 4-Queens Problem



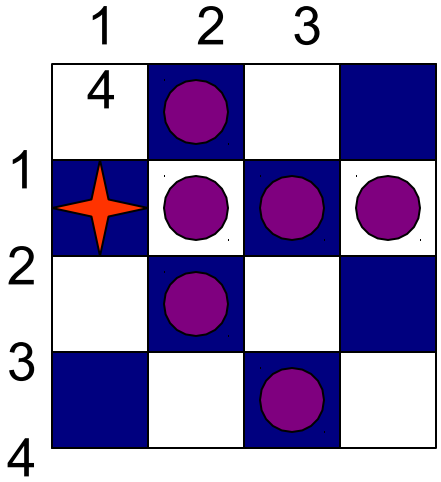


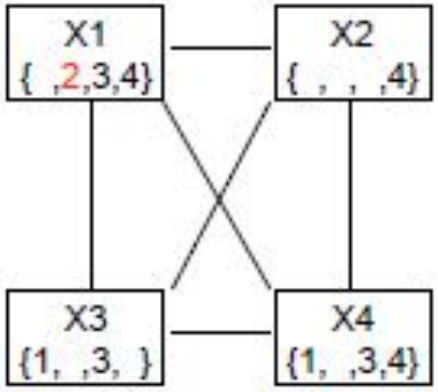
Example: 4-Queens Problem



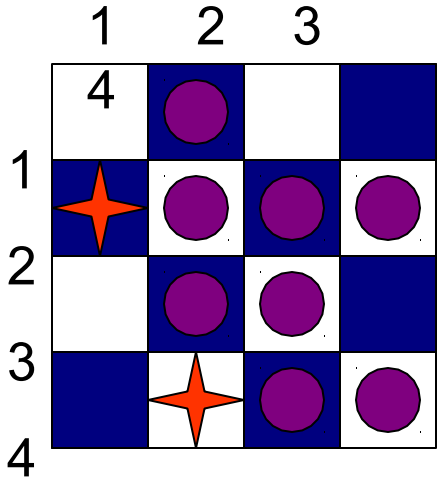


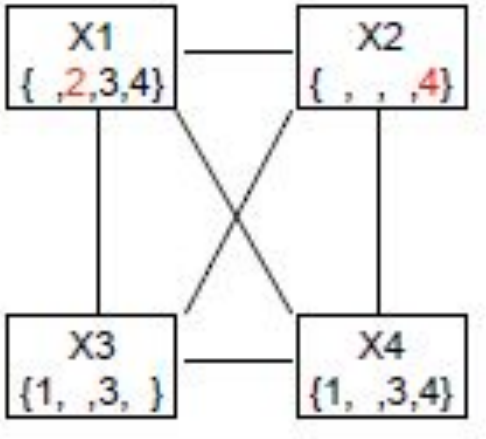
Example: 4-Queens Problem



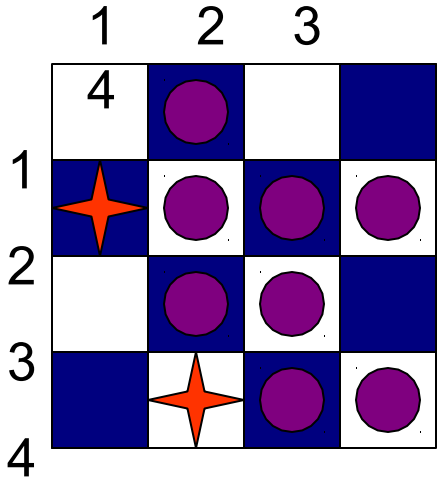


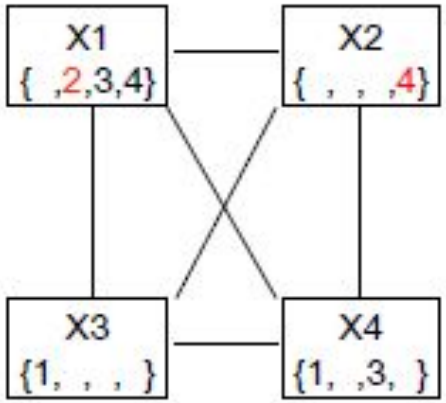
Example: 4-Queens Problem



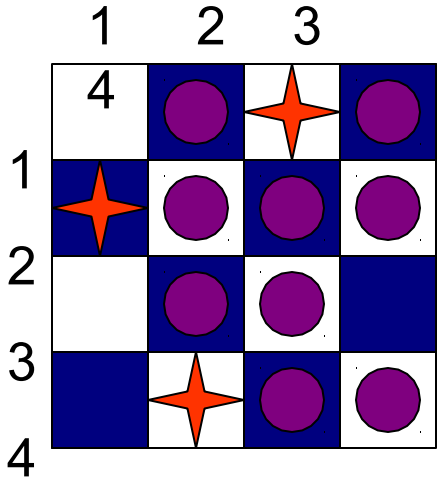


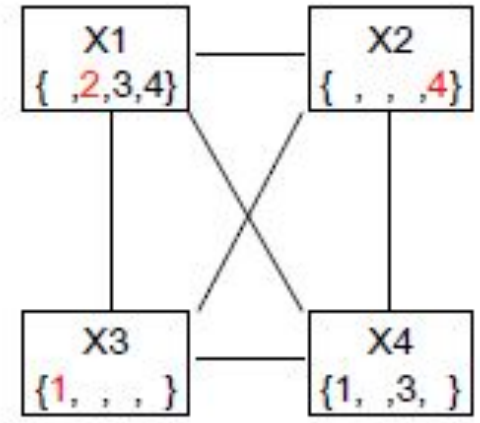
Example: 4-Queens Problem



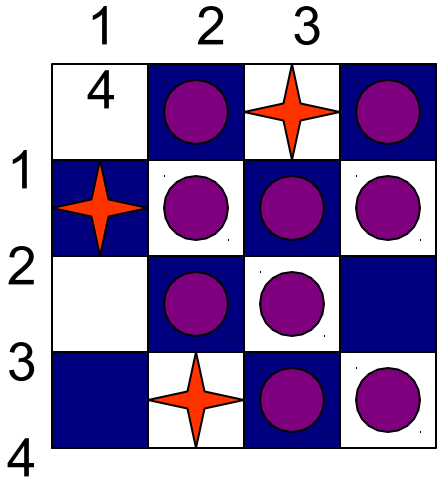


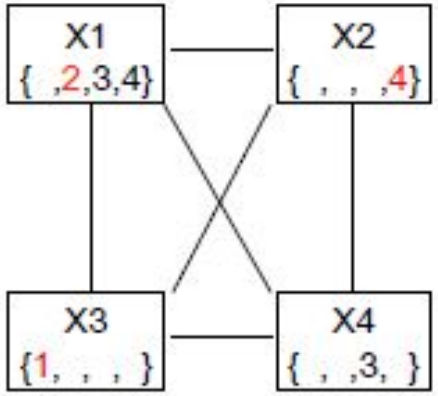
Example: 4-Queens Problem



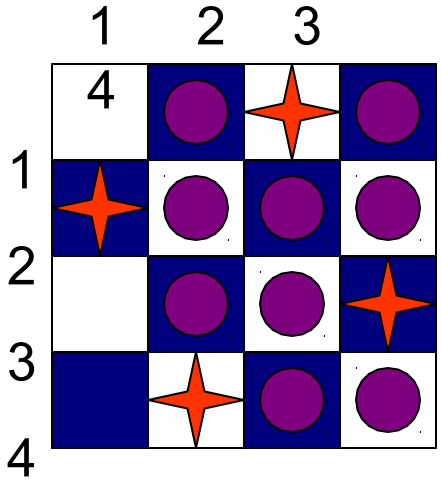


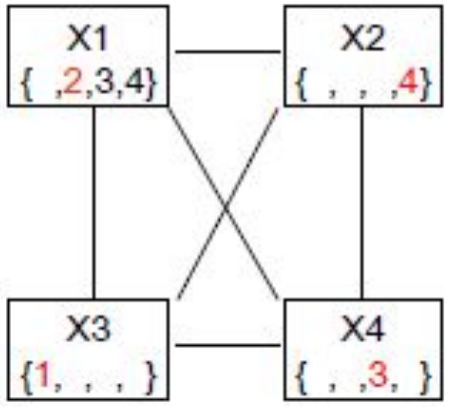
Example: 4-Queens Problem



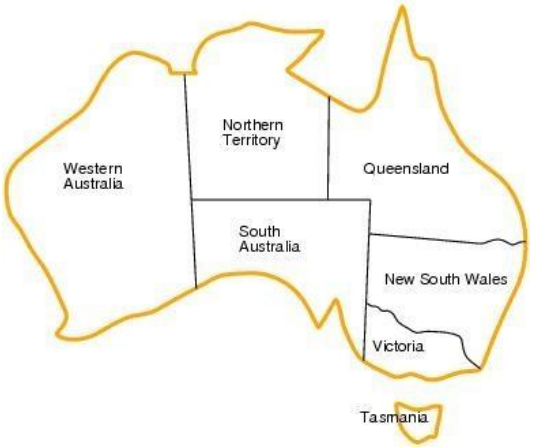


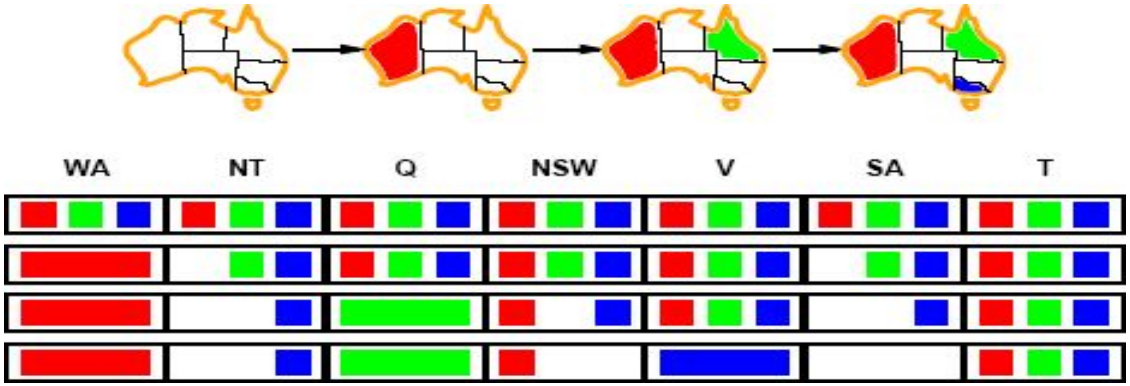
Example: 4-Queens Problem





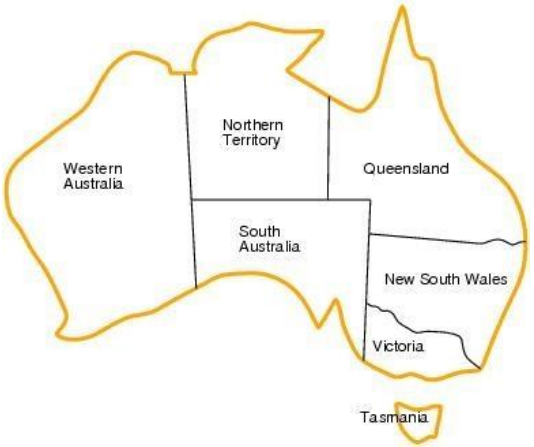
Constraint propagation

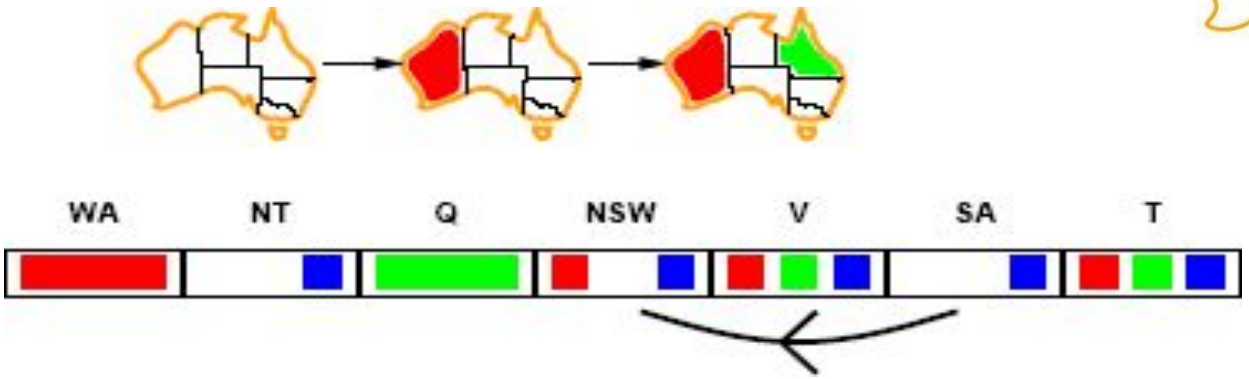




- Solving CSPs with combination of heuristics plus forward checking is more efficient than either approach alone.
- FC checking propagates information from assigned to unassigned variables but does not provide detection for all failures.
  - **NT and SA cannot be blue!**
- Constraint propagation repeatedly enforces constraints locally

Arc consistency





Simplest form of propagation makes each arc consistent

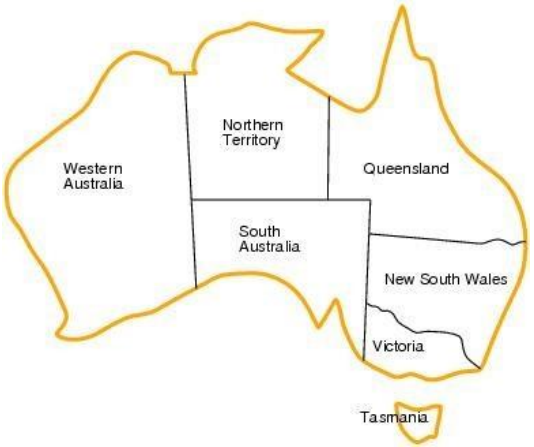
$X \rightarrow Y$ is consistent iff

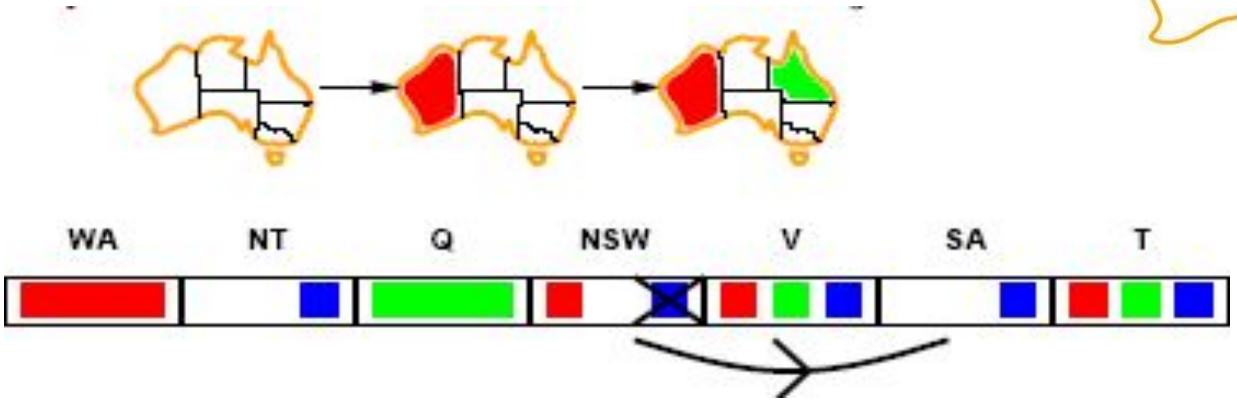
for *every* value x of X there is some allowed y

$SA \rightarrow NSW$ is consistent iff

SA=*blue* and NSW=*red*

Arc consistency



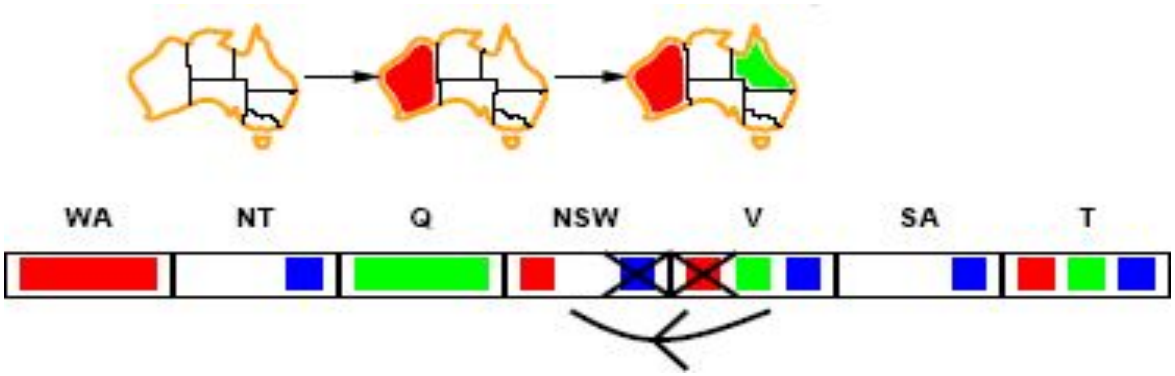


$X \rightarrow Y$ is consistent iff
for *every* value x of X there is some allowed y

$NSW \rightarrow SA$ is consistent iff
NSW=red and *SA=blue*
NSW=blue and SA=???

Arc can be made consistent by removing *blue* from *NSW*

Arc consistency

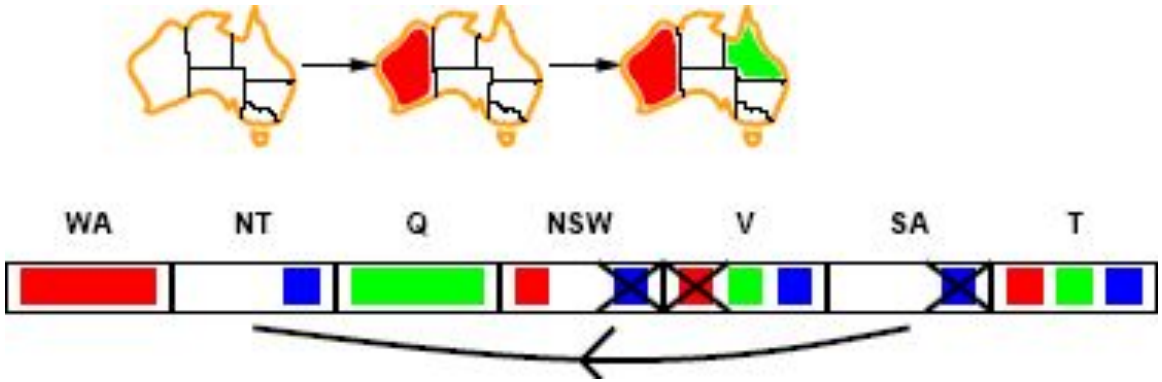


Arc can be made consistent by removing *blue* from *NSW*

RECHECK neighbours !!

- **Remove red from *V***

Arc consistency



Arc can be made consistent by removing *blue* from *NSW*

RECHECK neighbours !!

- **Remove red from *V***

Arc consistency detects failure earlier than FC

Can be run as a preprocessor or after each assignment.

- **Repeated until no inconsistency remains**

Arc consistency algorithm

function AC-3(*csp*) **return** 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 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_i, X_j) to queue

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **return** *true* iff we remove a value

removed $\leftarrow$ *false*

for each *x* **in** DOMAIN[X_i] **do**

if no value *y* in DOMAIN[X_i] allows (x,y) to satisfy the constraints between X_i and X_j

then delete x from DOMAIN[X_i]; *removed* $\leftarrow$ *true*

return *removed*

K-consistency

Arc consistency does not detect all inconsistencies:

- **Partial assignment *{WA=red, NSW=red}* is inconsistent.**

Stronger forms of propagation can be defined using the notion of k-consistency.

A CSP is k-consistent if for any set of k-1 variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable.

- **E.g. 1-consistency or node-consistency**
- **E.g. 2-consistency or arc-consistency**
- **E.g. 3-consistency or path-consistency**

K-consistency

- A graph is strongly k-consistent if
  - **It is k-consistent and**
  - **Is also (k-1) consistent, (k-2) consistent, … all the way down to 1-consistent.**
- This is ideal since a solution can be found in time $O(nd)$
  - instead of $O(n^2d^3)$
- YET *no free lunch*: any algorithm for establishing n-consistency must take time exponential in n, in the worst case.

Further improvements

Checking special constraints

- **Checking Alldif(…) constraint**

  E.g. {WA=red, NSW=red}

- **Checking Atmost(…) constraint**

  Bounds propagation for larger value domains

Intelligent backtracking

- **Standard form is chronological backtracking i.e. try different value for preceding variable.**
- **More intelligent, backtrack to conflict set.**

  Set of variables that caused the failure or set of previously assigned variables that are connected to X by constraints.

  Backjumping moves back to most recent element of the conflict set.

  Forward checking can be used to determine conflict set.

Local search for CSP

- Use complete-state representation
- For CSPs
  - **allow states with unsatisfied constraints**
  - **operators reassign variable values**
- Variable selection: randomly select any conflicted variable
- Value selection: *min-conflicts heuristic*
  - **Select new value that results in a minimum number of conflicts with the other variables**

Local search for CSP

function MIN-CONFLICTS(*csp, max_steps*) **return** solution or failure

inputs: *csp*, a constraint satisfaction problem

max_steps, the number of steps allowed before giving up

current ← an initial complete assignment for *csp*

for *i* = 1 to *max_steps* **do**

if *current* is a solution for *csp* then return *current*

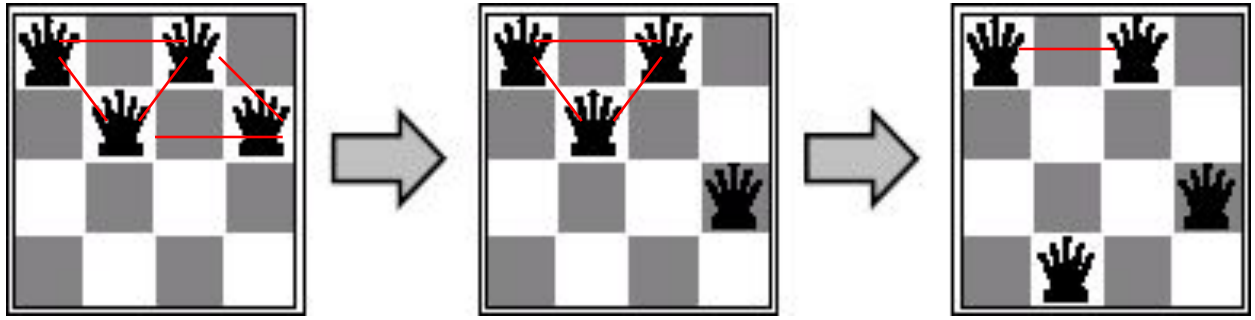
var ← a randomly chosen, conflicted variable from VARIABLES[*csp*]

value ← the value *v* for *var* that minimizes CONFLICTS(*var,v,current,csp*)

set *var* = *value* in *current*

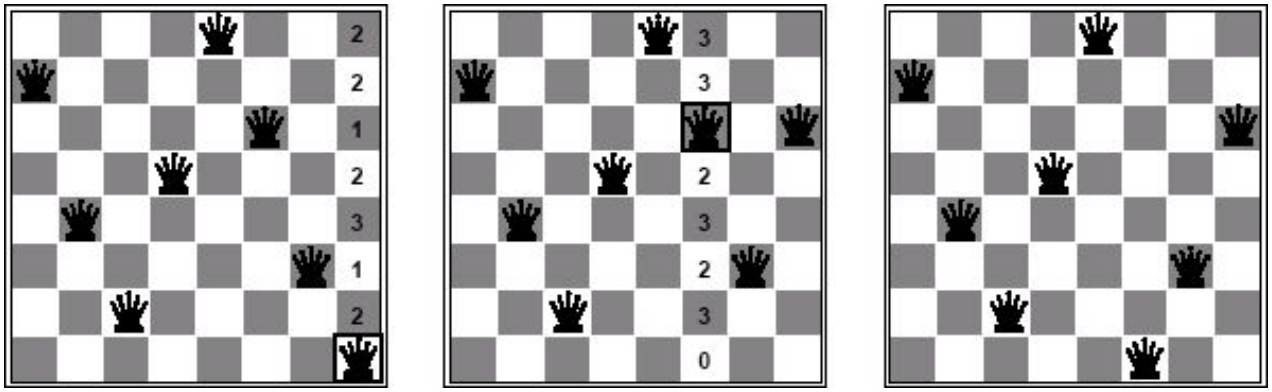
return *faiilure*

Min-conflicts example 1



Use of min-conflicts heuristic in hill-climbing.

Min-conflicts example 2

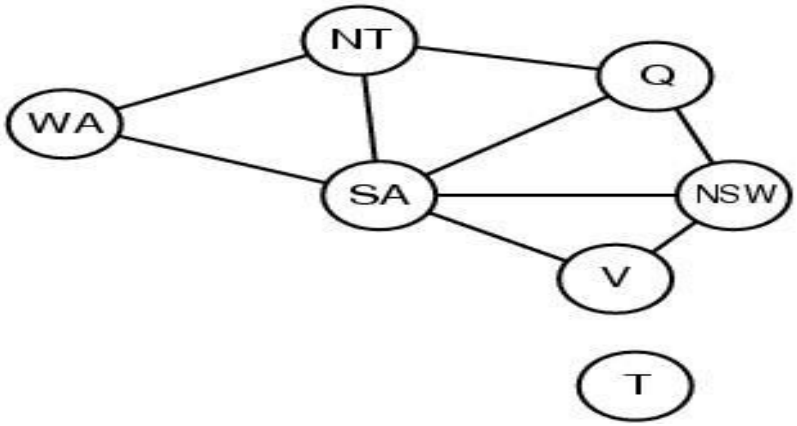


A two-step solution for an 8-queens problem using min-conflicts heuristic.

At each stage a queen is chosen for reassignment in its column.

The algorithm moves the queen to the min-conflict square breaking ties randomly.

Problem structure



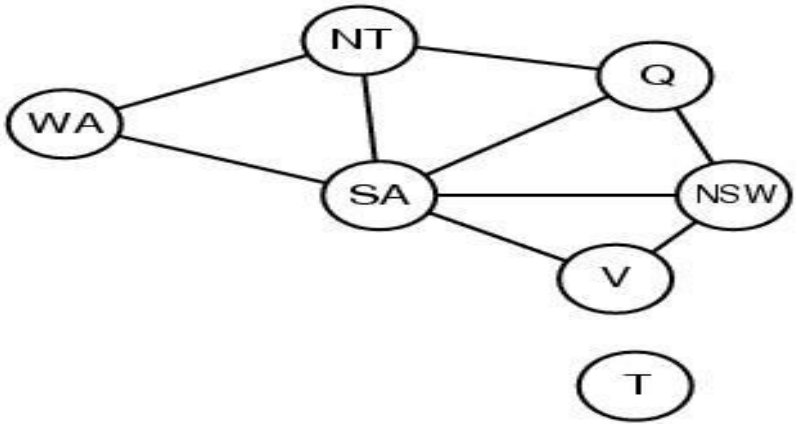
How can the problem structure help to find a solution quickly?

Subproblem identification is important:

- **Coloring Tasmania and mainland are independent subproblems**
- **Identifiable as connected components of constrained graph.**

Improves performance

Problem structure



Suppose each problem has *c* variables out of a total of *n*.

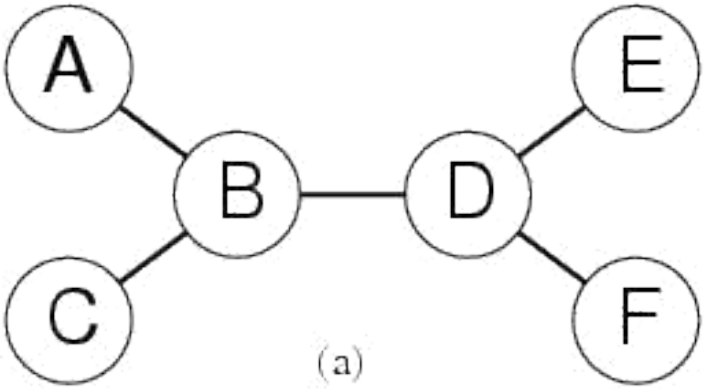
Worst case solution cost is $O(n/c\ d^c)$, i.e. linear in *n*

- **Instead of $O(d^n)$, exponential in *n***

E.g. *n= 80, c= 20, d=2*

- **2^{80} = 4 billion years at 1 million nodes/sec.**
- **$4 * 2^{20}$= .4 second at 1 million nodes/sec**

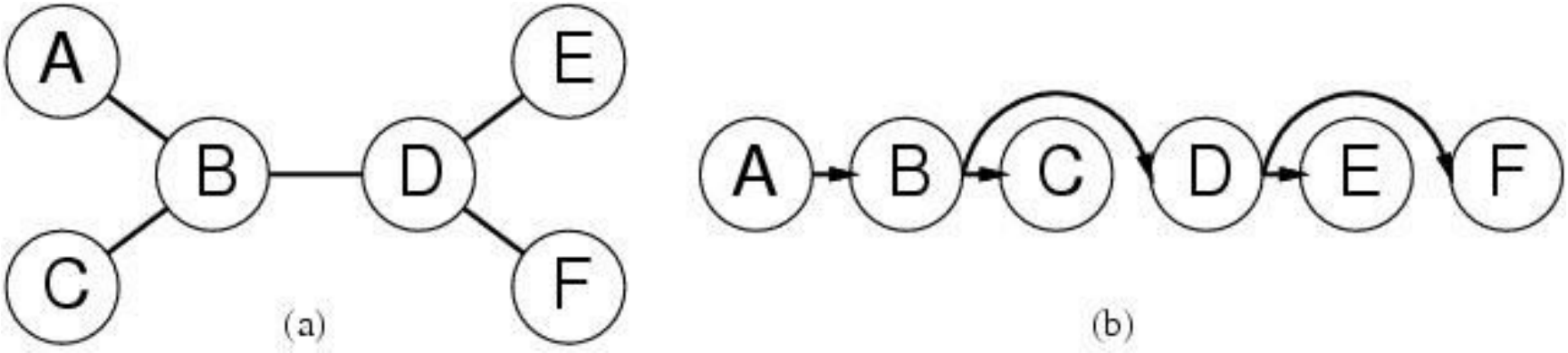
Tree-structured CSPs



Theorem: if the constraint graph has no loops then CSP can be solved in $O(nd^2)$ time

Compare difference with general CSP, where worst case is $O(d^n)$

Tree-structured CSPs

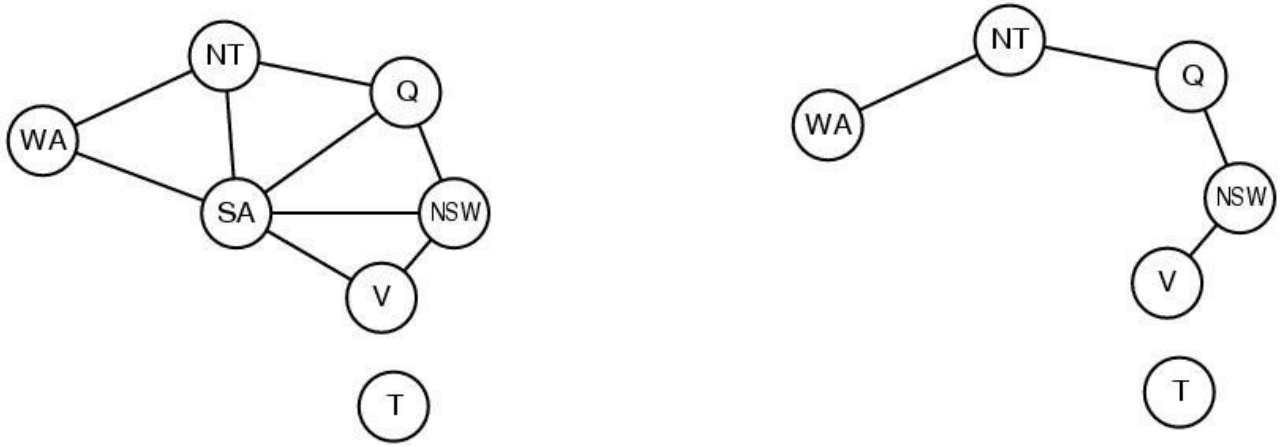


In most cases subproblems of a CSP are connected as a tree

Any tree-structured CSP can be solved in time linear in the number of variables.

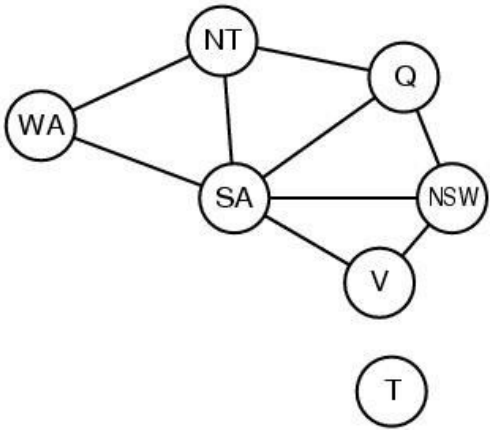
- **Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering.**
- **For *j* from *n* down to 2, apply** REMOVE-INCONSISTENT-VALUES(Parent(X_j),X_j)
- **For *j* from 1 to *n* assign X_j consistently with Parent(X_j)**

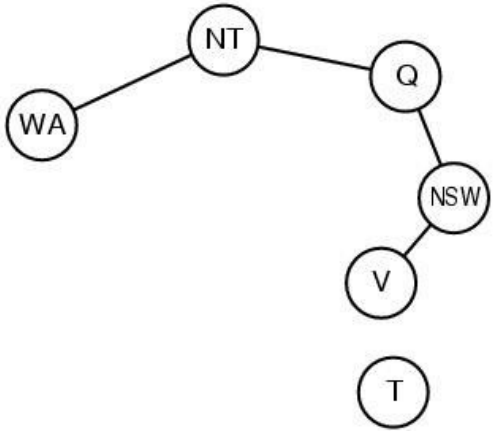
Nearly tree-structured CSPs



- *Can more general constraint graphs be reduced to trees?*
- Two approaches:
  - **Remove certain nodes**
  - **Collapse certain nodes**

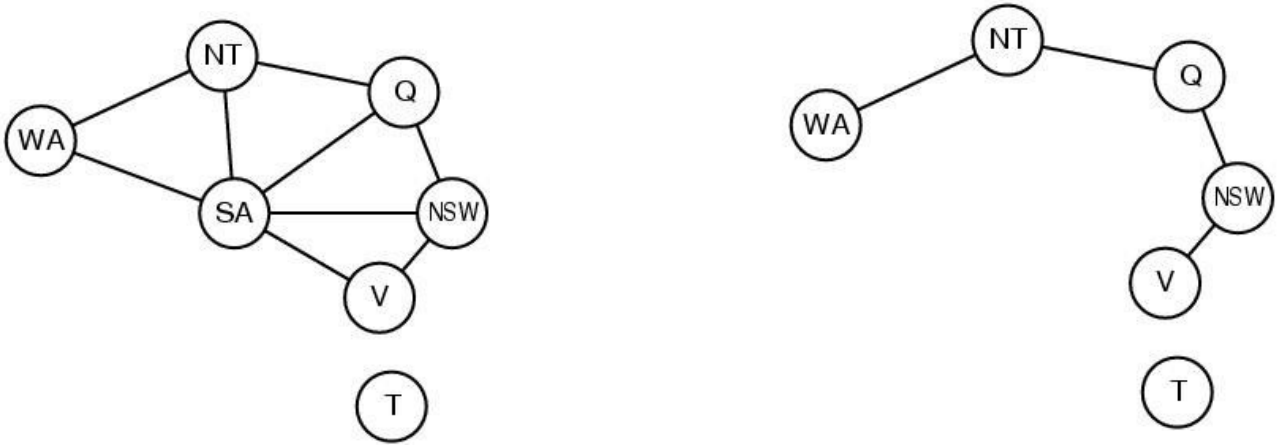
Nearly tree-structured CSPs





- Idea: assign values to some variables so that the remaining variables form a tree.
- Assume that we assign *{SA=x}* ← *cycle cutset*
  - **And remove any values from the other variables that are inconsistent.**
  - **The selected value for SA could be the wrong one so we have to try all of them**

Nearly tree-structured CSPs



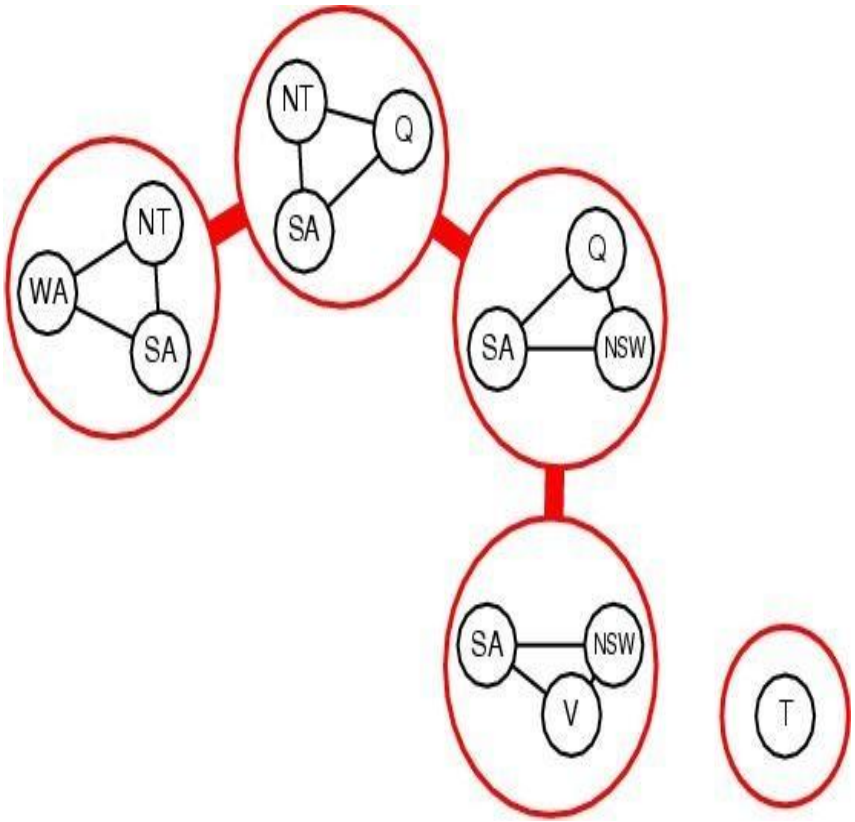
This approach is worthwhile if cycle cutset is small.

Finding the smallest cycle cutset is NP-hard

- **Approximation algorithms exist**

This approach is called *cutset conditioning*.

Nearly tree-structured CSPs



Tree decomposition of the constraint graph in a set of connected subproblems.

Each subproblem is solved independently

Resulting solutions are combined.

Necessary requirements:

- **Every variable appears in ar least one of the subproblems.**
- **If two variables are connected in the original problem, they must appear together in at least one subproblem.**
- **If a variable appears in two subproblems, it must appear in eacht node on the path.**

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 lead to failure.

Constraint propagation does additional work to constrain values and detect inconsistencies.

The CSP representation allows analysis of problem structure.

Tree structured CSPs can be solved in linear time.

Iterative min-conflicts is usually effective in practice.