Week 8:
Constraint Satisfaction Problems

Dr Ammar Masood

Department of Cyber Security,

Air University Islamabad

Table of Contents

- Constraint Satisfaction Problem
- Types
- Map coloring Problem
- Job Shop scheduling
- Sudoku
- 8 Queens Problem
- Approaches

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
- Eliminate large portions of the search space, all at once by identifying variable/value combinations that violate the constraints.

Defining CSP

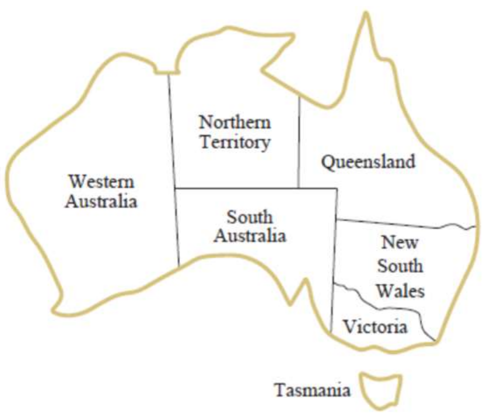
A constraint satisfaction problem consists of three components: X,D and C:

- X is a set of variables {X1,...,Xn} ,
- D is a set of domains, {D1,...,Dn} ,
- C is a set of constraints that specify allowable combinations of values.

Example: Sudoku, Crossword Puzzles, Timetable Scheduling.

Example Problem: Map coloring

Suppose that, having tired of Romania, we are looking at a map of Australia showing each of its states and territories. We are given the task of coloring each region either red, green, or blue in such a way that no two neighboring regions have the same color.



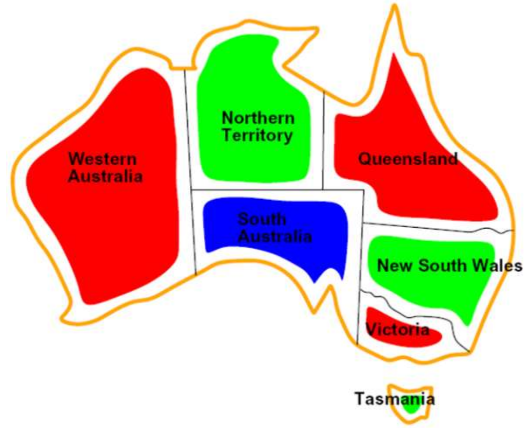
Example: Map Coloring

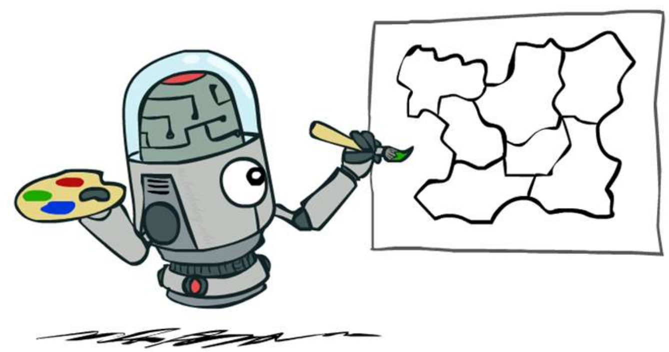
- Variables: $\mathrm{WA},\ \mathrm{NT},\ \mathrm{Q},\ \mathrm{NSW},\ \mathrm{V},\ \mathrm{SA},\ \mathrm{T}$
- Domains: $D = \{\mathrm{red}, \mathrm{green}, \mathrm{blue}\}$
- Constraints: adjacent regions must have different colors

  Implicit: $\mathrm{WA} \neq \mathrm{NT}$

  Explicit: $(\mathrm{WA}, \mathrm{NT}) \in \{(\mathrm{red}, \mathrm{green}), (\mathrm{red}, \mathrm{blue}), \ldots\}$
- Solutions are assignments satisfying all constraints, e.g.:

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





there are nine constraints:

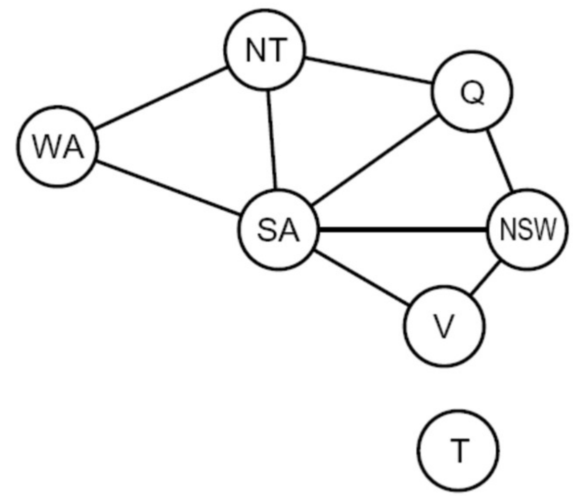
C = {SA≠WA,SA≠NT,SA≠Q,SA≠NSW,SA≠V, WA≠NT,NT ≠Q,Q≠NSW,NSW ≠V}.

Here we are using abbreviations; SA≠WA is a shortcut for ⟨(SA,WA),SA ≠ WA⟩ *where it is fully enumerated as:*

{(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green)}.

Constraint Graphs

- Binary CSP: each constraint relates (at most) two variables
- Nodes are variables, arcs show constraints
- General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!

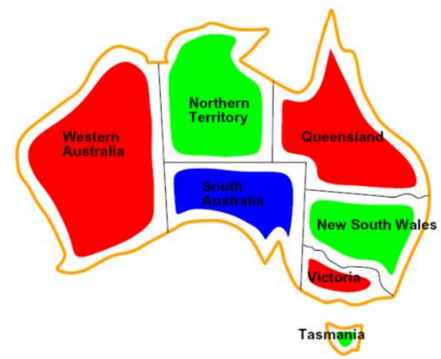


there are nine constraints:

C = {SA≠WA,SA≠NT,SA≠Q,SA≠NSW,SA≠V, WA≠NT,NT ≠Q,Q≠NSW,NSW ≠V}.

So nine edges above

Why formulate a problem?



1. CSPs yield a natural representation for a wide variety of problems; it is often easy to formulate a problem as a CSP.
2. Another is that years of development work have gone into making CSP solvers fast and efficient.
3. A CSP solver can quickly prune large swathes of the search space that an atomic state-space searcher cannot.

For example, once we have chosen {SA =blue} in the Australia problem, we can conclude that none of the five neighboring variables can take on the value blue. A search procedure that does not use constraints would have to consider $3^5 = 243$ assignments for the five neighboring variables; with constraints we have only $\mathbf{2^5 = 32}$ assignments to consider, a reduction of 87%

Problem#2: Job Shop Scheduling

- Factories have the problem of scheduling a day's worth of jobs, subject to various constraints. In practice, many of these problems are solved with CSP techniques.
- Consider the problem of scheduling the assembly of a car. The whole job is composed of tasks, and we can model each task as a variable,
  - The value of each variable is the time that the task starts, expressed as an integer number of minutes.
- Constraints can assert that one task must occur before another—for example, a wheel must be installed before the hubcap is put on—and that only so many tasks can go on at once.
- Constraints can also specify that a task takes a certain amount of time to complete.

Problem#2 cont..

We consider a small part of the car assembly, consisting of 15 tasks: install axles (front and back), affix all four wheels (right and left, front and back), tighten nuts for each wheel, affix hubcaps, and inspect the final assembly.

We can represent the tasks with 15 variables:

X = {AxleF, AxleB, WheelRF, WheelLF, WheelRB, WheelLB, NutsRF, NutsLF, NutsRB, NutsLB, CapRF, CapLF, CapRB, CapLB, Inspect}.

Precedence constraints

Next, we represent precedence constraints between individual tasks. Whenever a task T_1 must occur before task T_2 , and task T_1 takes duration d_1 to complete, we add an arithmetic constraint of form:

$$T_1 + d_1 \leq T_2.$$

In our example, the axles have to be in place before the wheels are put on, and it takes 10 minutes to install an axle, so we write

$$AxleF + 10 \leq WheelRF;\ AxleF + 10 \leq WheelLF;$$

$$AxleB + 10 \leq WheelRB;\ AxleB + 10 \leq WheelLB.$$

Working

Next we say that for each wheel, we must affix the wheel (which takes 1 minute), then tighten the nuts (2 minutes), and finally attach the hubcap (1 minute, but not represented yet):

$$WheelRF + 1 \leq NutsRF;\ NutsRF + 2 \leq CapRF;$$

$$WheelLF + 1 \leq NutsLF;\ NutsLF + 2 \leq CapLF;$$

$$WheelRB + 1 \leq NutsRB;\ NutsRB + 2 \leq CapRB;$$

$$WheelLB + 1 \leq NutsLB;\ NutsLB + 2 \leq CapLB.$$

Worker increased

Suppose we have four workers to install wheels, but they have to share one tool that helps put the axle in place. We need a disjunctive constraint to say that AxleF and AxleB must not overlap in time either one comes first or the other

$$(AxleF + 10 \leq AxleB) \text{ or } (AxleB + 10 \leq AxleF)$$

This looks like a more complicated constraint, combining arithmetic and logic. But it still reduces to a set of pairs of values that AxleF and AxleB can take on.

Inspection

We also need to assert that the inspection comes last and takes 3 minutes. For every variable except Inspect we add a constraint of the form $X+dX \leq Inspect$. Finally, suppose there is a requirement to get the whole assembly done in 30 minutes. We can achieve that by limiting the domain of all variables:

$$D_i=\{0,1,2,3,\ldots,30\}$$

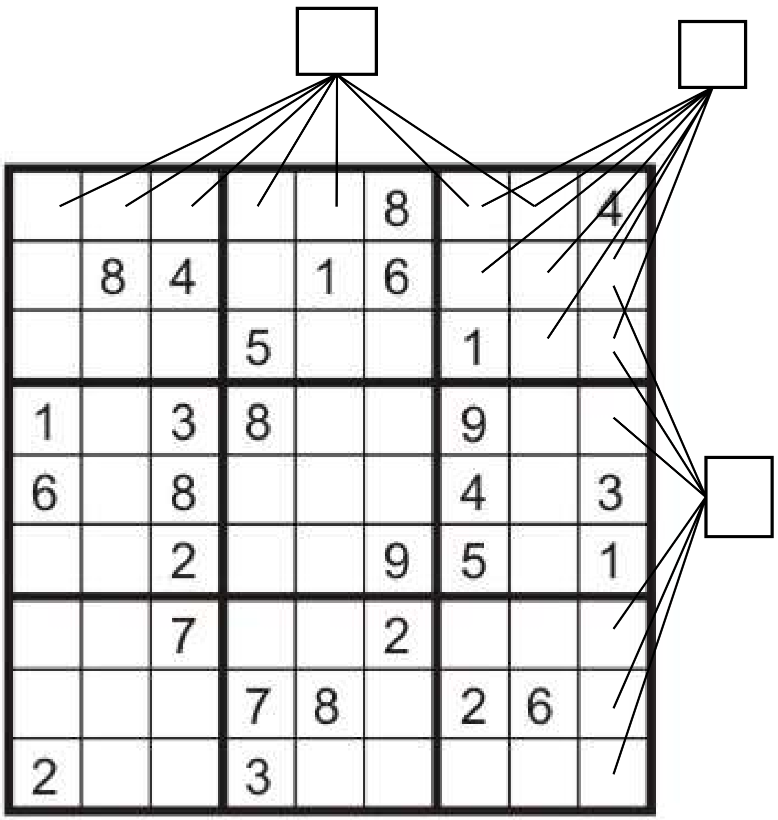
This particular problem is trivial to solve, but CSPs have been successfully applied to job shop scheduling problems like this with thousands of variables.

Varieties of CSPs

- Discrete variables
  - finite domains:
    - n variables, domain size $d \rightarrow O(d^n)$ complete assignments
    - e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)
  - infinite domains:
    - integers, strings, etc.
    - e.g., job scheduling, variables are start/end days for each job
    - need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$

- Continuous variables
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in polynomial time by LP

Example: Sudoku



- Variables:
  - Each (open) square
- Domains:
  - {1,2,…,9}
- Constraints:

9-way alldiff for each column

9-way alldiff for each row

9-way alldiff for each region

(or can have a bunch of pairwise inequality constraints)

Types of CSP

Unary Constraints

Constraints that involve a single variable.

Example: $X \neq 3$ (X cannot take the value 3).

Binary Constraints

Constraints involving pairs of variables.

Example: $X \neq Y$ (X and Y must have different values).

Higher-order Constraints

Constraints that involve more than two variables.

Example: $X + Y + Z = 10$.

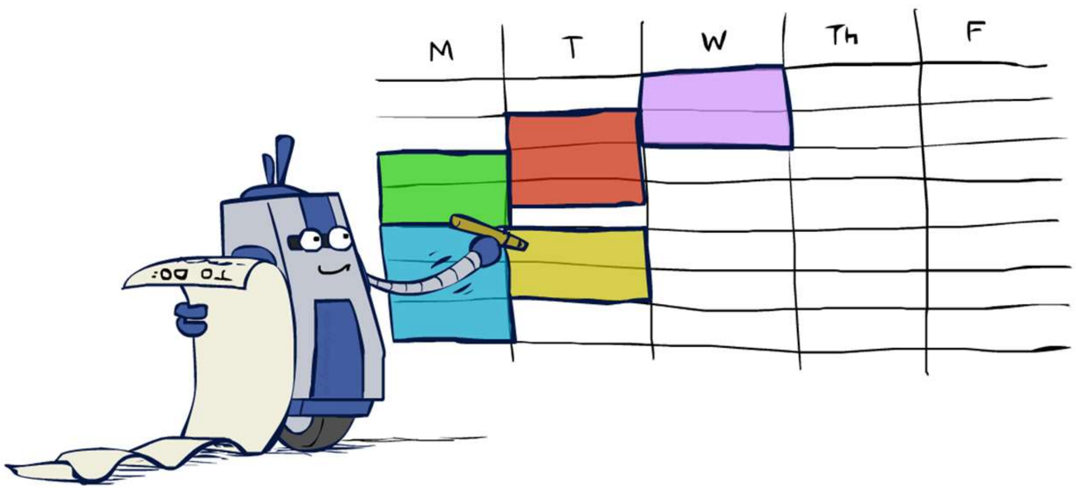
Soft vs Hard Constraints

Hard constraints must be strictly satisfied.

Soft constraints allow some violations but aim to minimize them.

Real-World CSPs

- Assignment problems: e.g., who teaches what class
- Timetabling problems: e.g., which class is offered when and where?
- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Circuit layout
- Fault diagnosis
- … lots more!



- Many real-world problems involve real-valued variables…

CSPs by Standard Search

- State
  - Defined by the values assigned so far
- Initial state
  - The empty assignment
- Successor function
  - Assign a value to a unassigned variable
- Goal test
  - All variables are assigned and no conflict

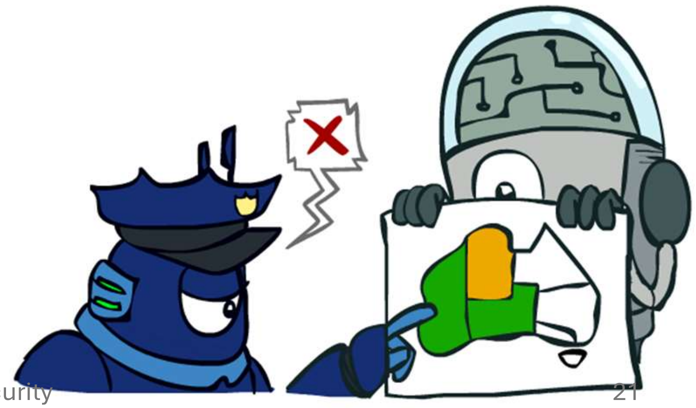
https://www.cs.cmu.edu/~15281/demos/csp_backtracking/

CSP by Standard Search

- Every solution appears at depth *d* with *n* variables
  - Use depth-first search
- Path is irrelevant
- Number of leaves
  - $n!d^n$
    - Too many

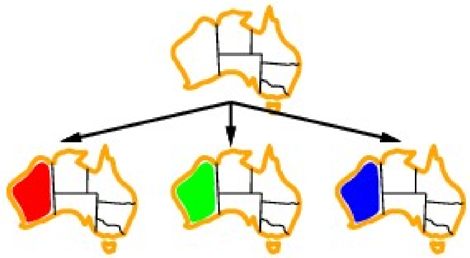
Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
  - Variable assignments are commutative, so fix ordering
  - 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 step
- Idea 2: Check constraints as you go
  - I.e. consider only values which do not conflict previous assignments
  - Might have to do some computation to check the constraints
  - "Incremental goal test"
- Depth-first search with these two improvements is called *backtracking search* (not the best name)
- Can solve n-queens for n $\approx$ 25

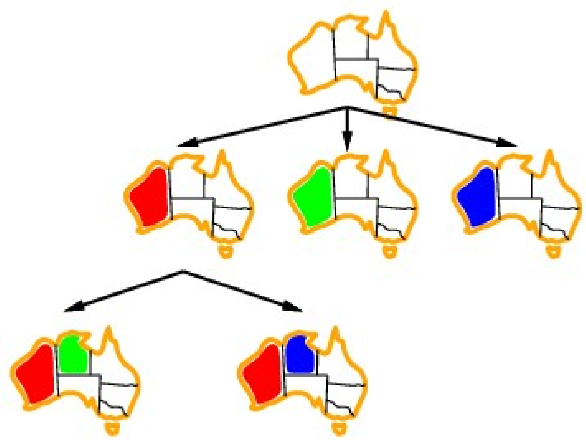


Backtracking example

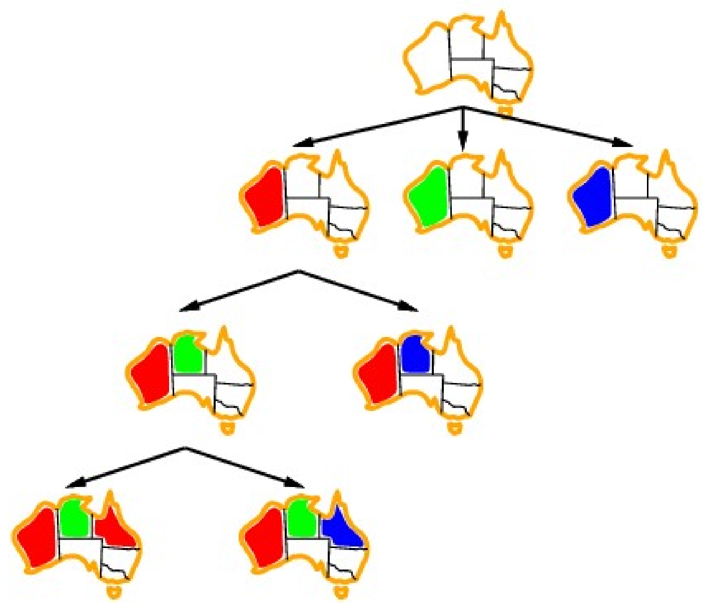
Backtracking example



Backtracking example



Backtracking example



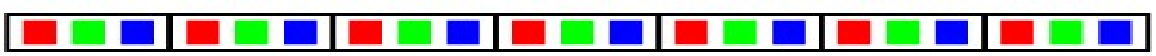
https://www.cs.cmu.edu/~15281/demos/csp_backtracking/

Filtering: Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Forward checking: Cross off values that violate a constraint when added to the existing assignment
  - Terminate search when any variable has no legal values

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

function BACKTRACK(csp, assignment) returns a solution or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
  for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
     if value is consistent with assignment then
        add {var = value} to assignment
        inferences ← INFERENCE(csp, var, assignment)
        if inferences ≠ failure then
           add inferences to csp
           result ← BACKTRACK(csp, assignment)
           if result ≠ failure then return result
           remove inferences from csp
        remove {var = value} from assignment
  return failure
```

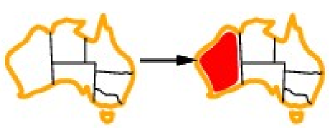
WA NT Q NSW V SA T

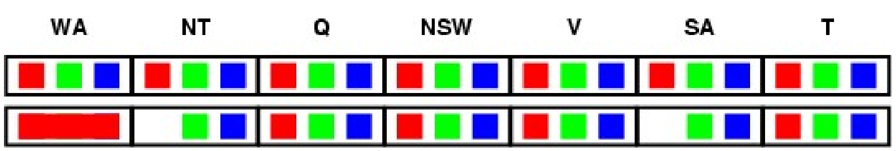


The functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, implement the heuristics discussed later

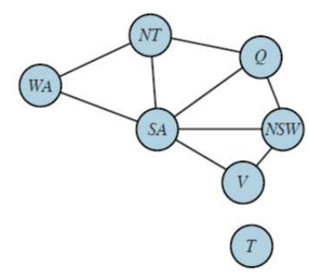
Forward checking

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



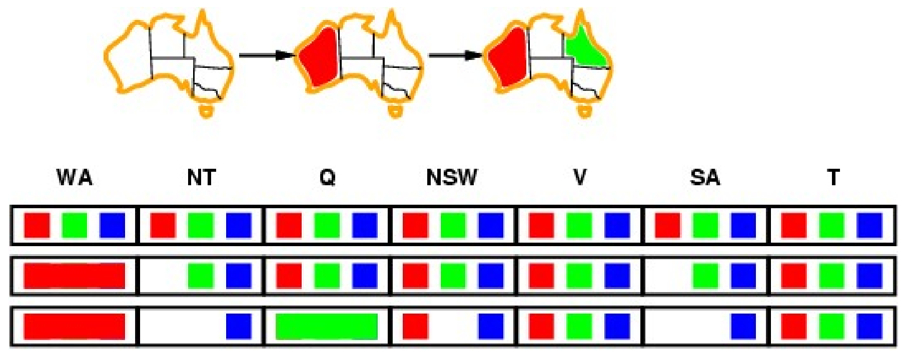


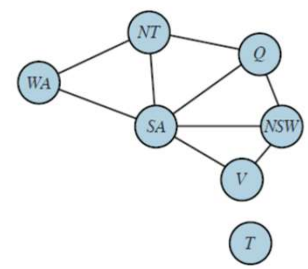

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
  return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
  for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
     if value is consistent with assignment then
        add {var = value} to assignment
        inferences ← INFERENCE(csp, var, assignment)
        if inferences ≠ failure then
           add inferences to csp
           result ← BACKTRACK(csp, assignment)
           if result ≠ failure then return result
           remove inferences from csp
        remove {var = value} from assignment
   rn failure
```



Forward checking

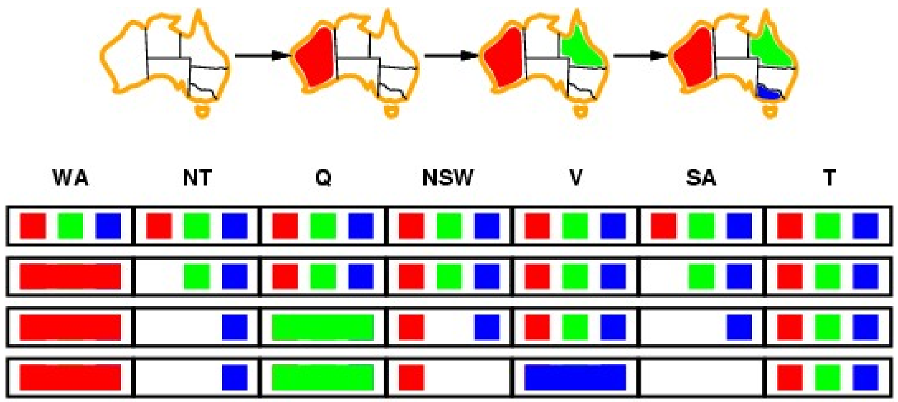
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values

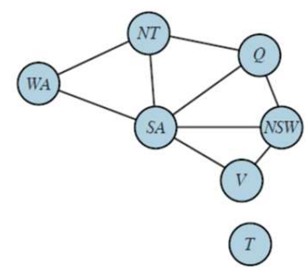




Forward checking

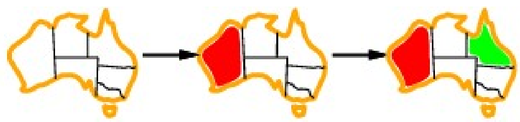
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values

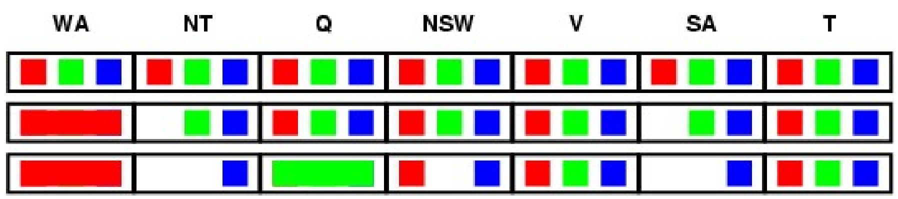


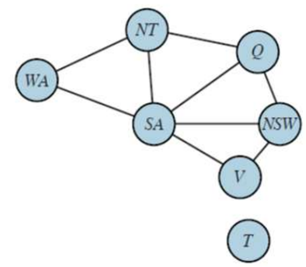


Constraint propagation

- 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 algorithms repeatedly enforce constraints locally...

Improving Backtracking

- General-purpose ideas give huge gains in speed
  - … but it's all still NP-hard

- Filtering (Forward Checking): Can we detect inevitable failure early?

- Ordering:
  - Which variable should be assigned next? (MRV)
  - In what order should its values be tried? (LCV)

- Structure: Can we exploit the problem structure?

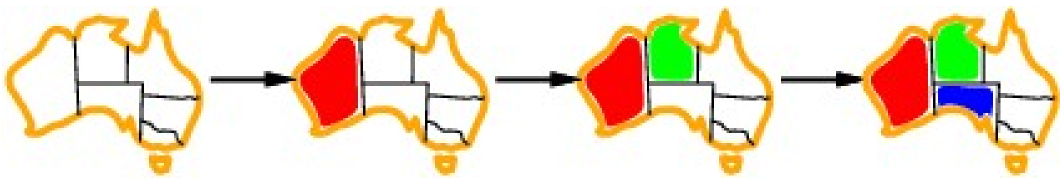
Ordering: Improving backtracking efficiency

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

Most Constrained Variable or Minimum Remaining Values (MRV) Heuristic

- Most constrained variable:

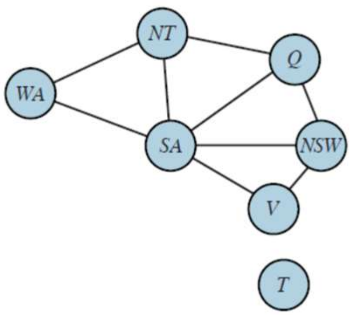
  choose the variable with the fewest legal values

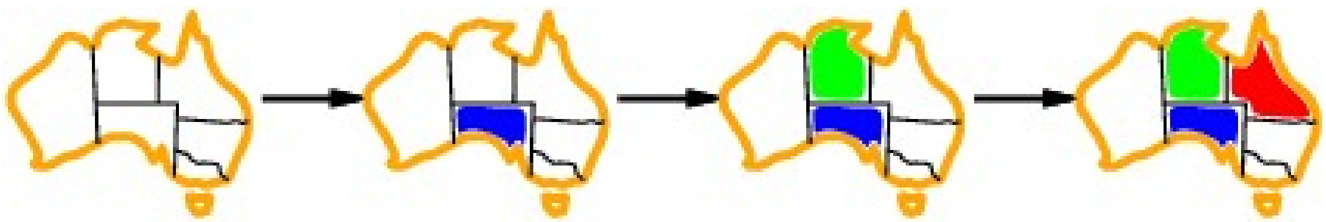


- a.k.a. minimum remaining values (MRV) heuristic

Most Constraining Variable or Degree Heuristic

- A good idea is to use it as a tie-breaker among most constrained variables
- The MRV heuristic doesn't help at all in choosing the first region to color in Australia, because initially every region has three legal colors.
- Most constraining variable:
  - choose the variable with the most constraints on remaining variables or highest degree in constraint graph





Least Constraining Value

- Once a variable has been selected, the algorithm must decide on the order in which to examine its values
- Given a variable to assign, choose the least constraining value:
  - the one that rules out the fewest values in the remaining variables
  - Combining these heuristics makes 1000 queens feasible

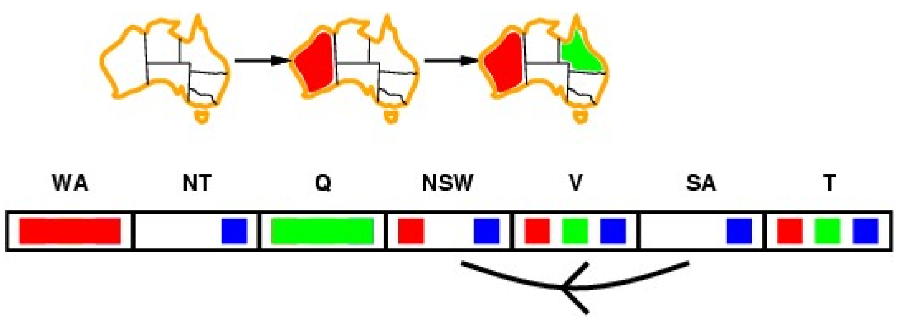
```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
  return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
  for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, assignment)
      if inferences ≠ failure then
        add inferences to csp
        result ← BACKTRACK(csp, assignment)
        if result ≠ failure then return result
        remove inferences from csp
      remove {var = value} from assignment
  return failure
```

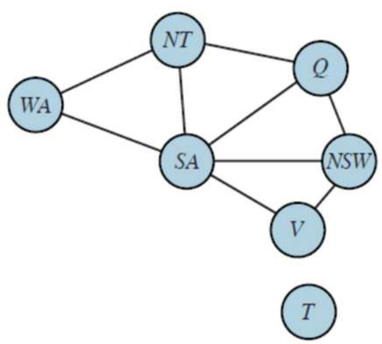
Arc consistency

- Simplest form of propagation makes each arc consistent. Variable in a CSP is arc-consistent if every value in its domain satisfies the variable's binary constraints.

- $X \rightarrow Y$ is consistent iff

  for every value x of X there is some allowed y



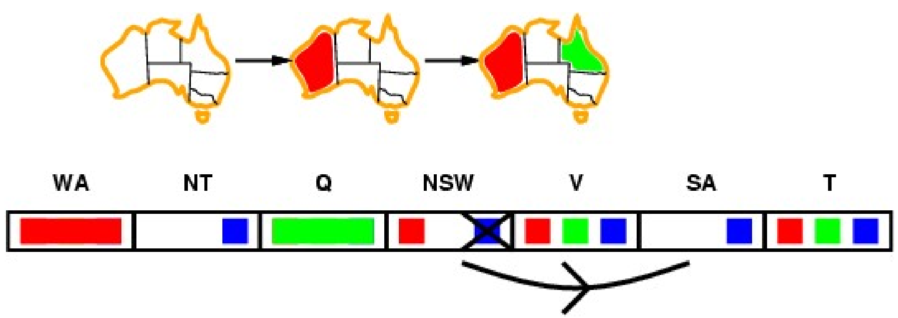


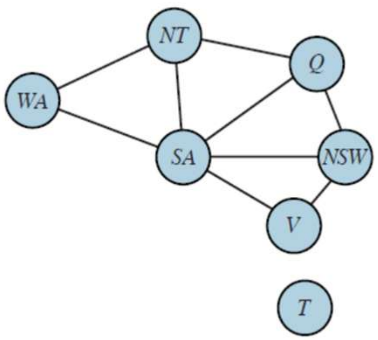
Arc consistency

- Simplest form of propagation makes each arc consistent. Variable in a CSP is arc-consistent if every value in its domain satisfies the variable's binary constraints.

- $X \rightarrow Y$ is consistent iff

  for every value x of X there is some allowed y

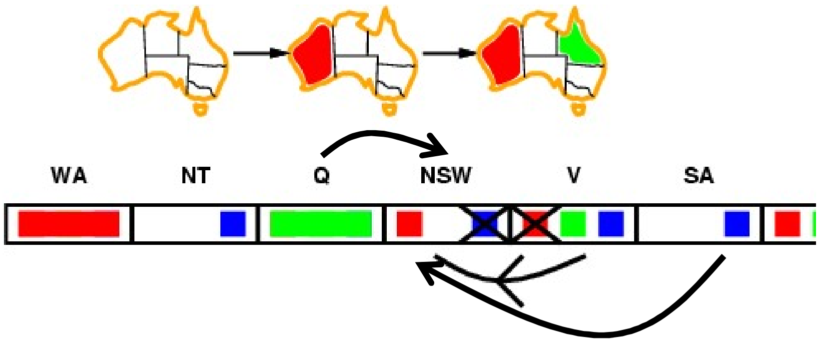


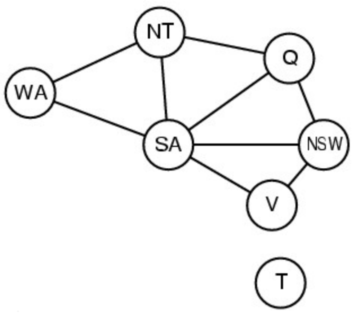


Arc consistency

- 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





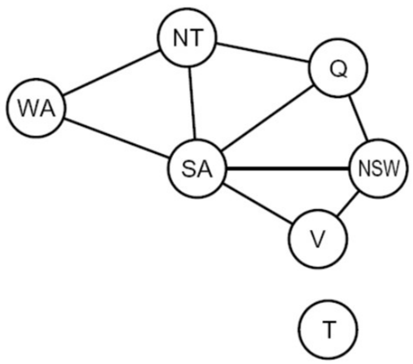
- If X loses a value, neighbors of X need to be rechecked

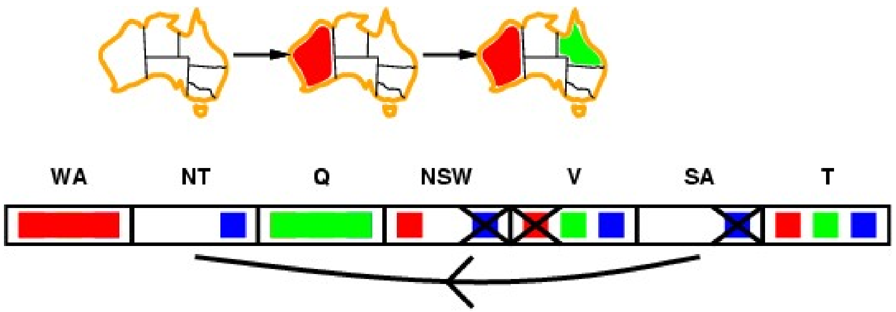
Arc consistency

- 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** false if an inconsistency is found and true otherwise
  $queue \leftarrow$ a queue of arcs, initially all the arcs in csp

  while $queue$ is not empty **do**
    $(X_i, X_j) \leftarrow$ POP($queue$)
    if REVISE(csp, X_i, X_j) **then**
      if size of $D_i = 0$ **then return** $false$
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add (X_k, X_i) to $queue$
  return $true$

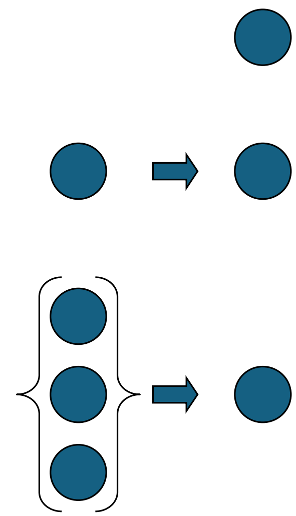
function REVISE(csp, X_i, X_j) **returns** true iff we revise the domain of X_i
  $revised \leftarrow false$
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      $revised \leftarrow true$
  return $revised$

- Time complexity: O(#constraints $|\text{domain}|^3$)

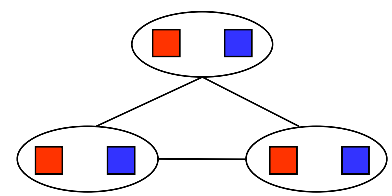
Checking consistency of an arc is O($|\text{domain}|^2$)

K-Consistency

- Increasing degrees of consistency
  - 1-Consistency (Node Consistency): Each single node's domain has a value which meets that node's unary constraints
  - 2-Consistency (Arc Consistency): For each pair of nodes, any consistent assignment to one can be extended to the other
  - K-Consistency: For each k nodes, any consistent assignment to k-1 can be extended to the k^{th} node.



- Higher k more expensive to compute
- (You need to know the k=2 case: arc consistency)



Strong K-Consistency

- Strong k-consistency: also k-1, k-2, … 1 consistent
- Claim: strong n-consistency means we can solve without backtracking!
- Why?
  - Choose any assignment to any variable
  - Choose a new variable
  - By 2-consistency, there is a choice consistent with the first
  - Choose a new variable
  - By 3-consistency, there is a choice consistent with the first 2
  - …
- Lots of middle ground between arc consistency and n-consistency! (e.g. k=3, called path consistency)

Other techniques for CSPs

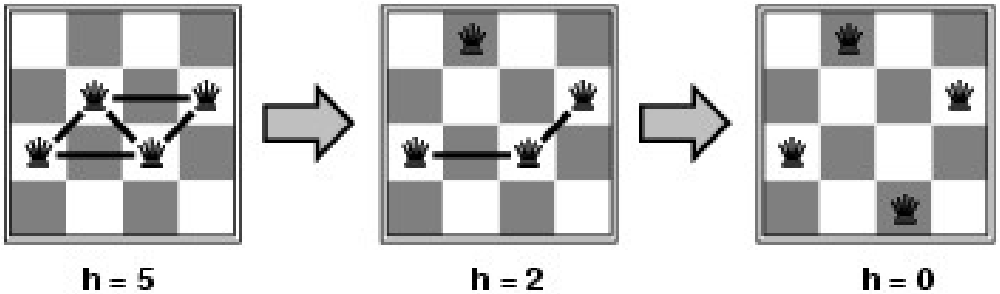
- Global constraints
  - E.g., Alldiff
  - E.g., Atmost(10,P1,P2,P3), i.e., sum of the 3 vars $\leq$ 10
  - Special propagation algorithms
    - Bounds propagation
      - E.g., number of people on two flight D1 = [0, 165] and D2 = [0, 385]
      - Constraint that the total number of people has to be at least 420
      - Propagating bounds constraints yields D1 = [35, 165] and D2 = [255, 385]
    - …
- Symmetry breaking

Local search for CSPs

- Hill-climbing, 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

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

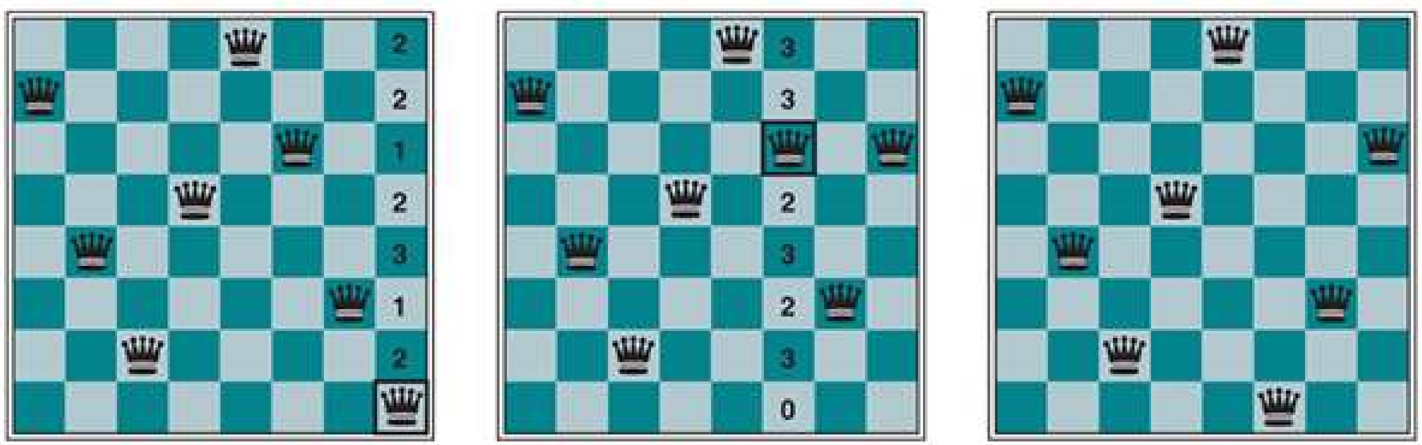


- Given random initial state, can solve *n*-queens in almost constant time for arbitrary *n* with high probability (e.g., *n* = 10,000,000)

8 Queens

- We start on the left with a complete assignment to the 8 variables; typically this will violate several constraints.
- Randomly choose a conflicted variable, which turns out to be Q8 , the rightmost column.
- Change the value to something that brings us closer to a solution; select the value that results in the minimum number of conflicts with other variables—**the min-conflicts heuristic**

```
function MIN-CONFLICTS(csp, max_steps) returns a 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 csp.VARIABLES
     value ← the value v for var that minimizes CONFLICTS(csp, var, v, current)
     set var = value in current
  return failure
```



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

CSP Solution approaches

Backtracking Search

A brute-force approach that assigns values to variables step-by-step and backtracks when a constraint is violated.

Algorithm:

- Assign a value to a variable.
- Check constraints.
- If constraints are satisfied, move to the next variable.
- If a constraint is violated, backtrack and try another value.
- Repeat until all variables are assigned valid values.

Other Approaches

Forward Checking

- Improves backtracking by eliminating values that would violate constraints before making assignments.

Constraint Propagation

- Uses techniques like **Arc Consistency (AC-3)** to reduce the search space by eliminating inconsistent values.

Local Search with Min-Conflicts Heuristic

- Starts with a random assignment and iteratively moves to a less conflicting assignment.
- Used in large CSPs where traditional search methods are inefficient.