CONSTRAINT SATISFACTION PROBLEMS

目录

CONSTRAINT SATISFACTION PROBLEMS(CSPS)

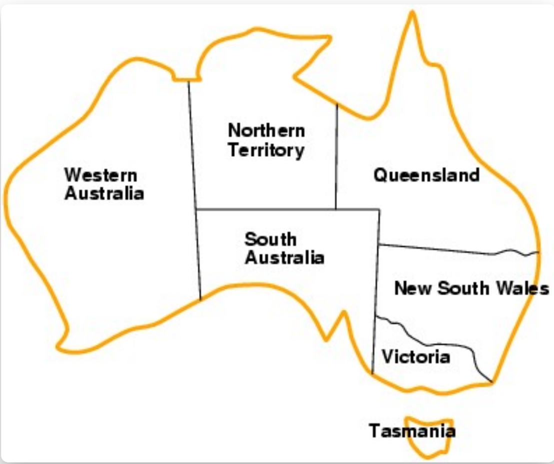
Standard Search Problem

State is a "black box" - any data structure that supports successor function, heuristic function, and goal test.

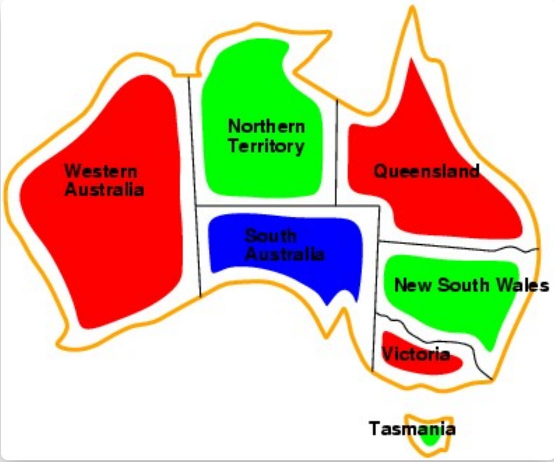
CSP

- **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

MAP-COLORING

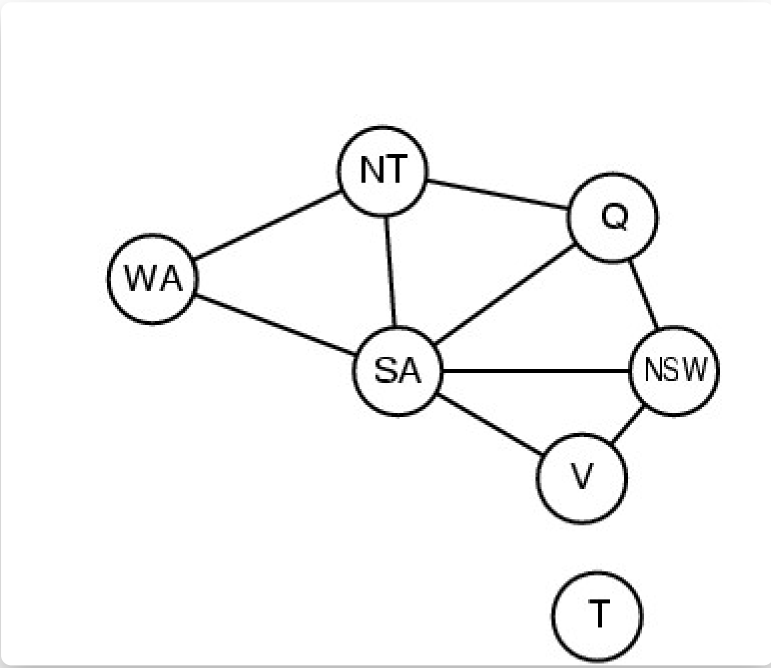


- Variables : WA, NT, Q, NSW, V, SA, T
- Domains : Di = {red,green,blue}
- Constraints: adjacent regions must have different colors
- Example : WA = red, NT = green, then SA = blue, Q = red, NSW = green, V = red, T = green （T can be red or green or blue for it's isolated.)



Constraint Graph

- **Binary CSP** : each constraint relates two variables
- **Constraint Graph** : nodes are variables, arcs are constraints



VARIETIES OF CSPS

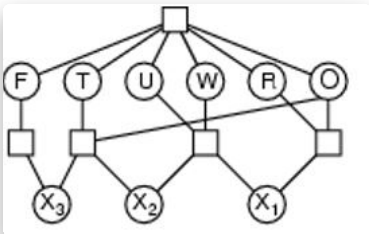
- Discrete Variables
  - Finite Domains:
    - n variables, domain size d -> O(d^n) complete assignments
    - e.g., Boolean CSPs, incl.~Boolean satisfiability (NP-complete)
  - Infinite Domains:
    - integers, strings, etc.
    - e.g., job scheduling, variables are start/end days for each job

- need a constraint language, e.g., StartJob1 + 5 ≤ StartJob3

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

VARIETIES OF CONSTRAINTS

- **Unary** constraints involve a single variable, e.g. $SA \neq green$.
- **Binary** constraints involve pairs of variables, e.g. $SA \neq WA$.
- **Higher-Order** constraints involve 3 or more variables, e.g. cryptarithmetic column constraints.
  - Example : Cryptarithmetic



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

  - Variables : F, T, U, W, R, O, X_1, X_2, X_3, X_3
  - Domains : {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
  - Constraints : Alldiff(F, T, U, W, R, O)
    - $O + O = R + 10 \times X_1$
    - $W_1 + W + W = U + 10 \times X_2$
    - $X_2 + T + T = O + 10 \times X_3$
    - $X_3 = F_1, T \neq 0, F \neq 0$

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

STANDARD SEARCH FORMULATION (INCREMENTAL)

Let's start with the straightforward approach, then fix it

States are defined by the values assigned so far

- **Initial State**: the empty assignment { }
- **Successor function**: assign a value to an unassigned variable that does not conflict with current assignment
  -> fail if no legal assignments

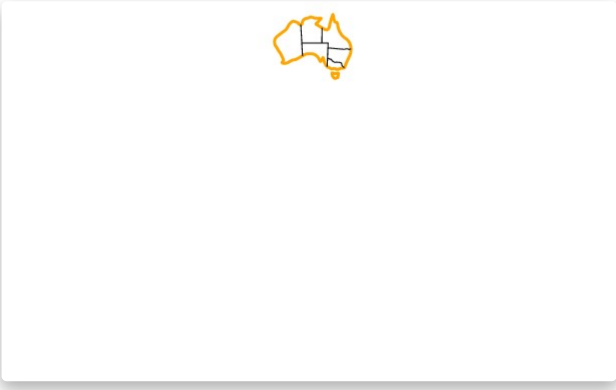
- **Goal test**: the current assignment is complete
- This is the same for all CSPs
- Every solution appears at depth n with n variables -> use depth-first search
- Path is irrelevant, so can also use complete-state formulation
  $b = (n - l)d$ at depth l, hence $n! \cdot d^n$ leaves

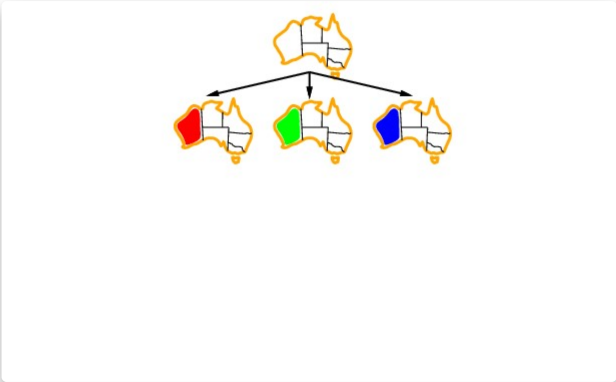
Backtracking Search

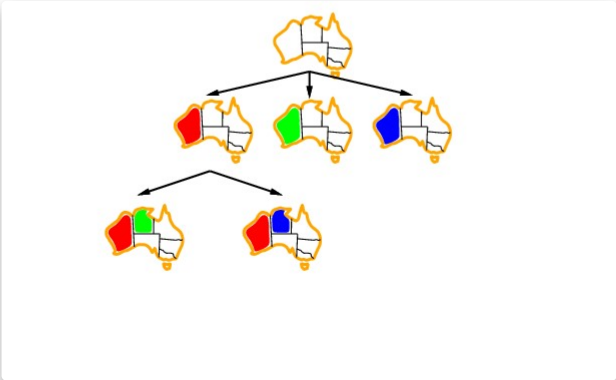
- DFS Search

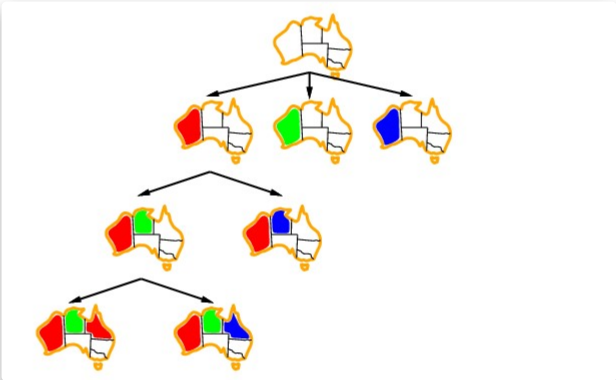
```
function BACKTRACKING-SEARCH( csp) returns a solution, or failure
   return RECURSIVE-BACKTRACKING({}, csp)
function RECURSIVE-BACKTRACKING( assignment, csp) returns a solution, or
failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE( Variables[csp], assignment, 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
```

Example: DFS Search









Improving BackTracking Efficiency

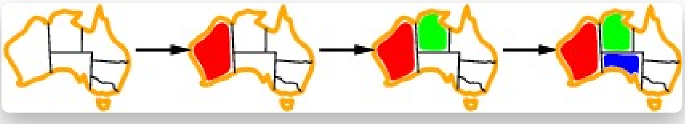
General-purpose methods can give huge gains in speed:

- Which variable should be assigned next?
- In what order should its values be tried?
- Can we detect inevitable failure early?

Most Constrained Variable

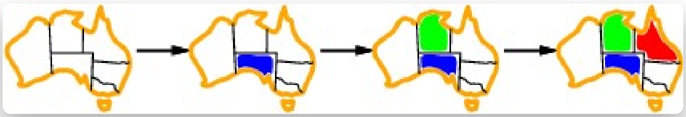
- Most Constrained Variable:

  Choose the variable with the fewest legal values.



- Tie-breaker among most constrained variables

  Choose the variable with the most constraints on remaining variables



Least Constraining Value

- Give a variable, choose the least constraining value:

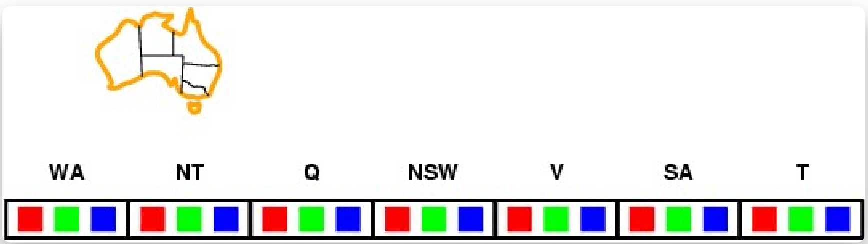
  the one that rules out the fewest values in the remaining variables

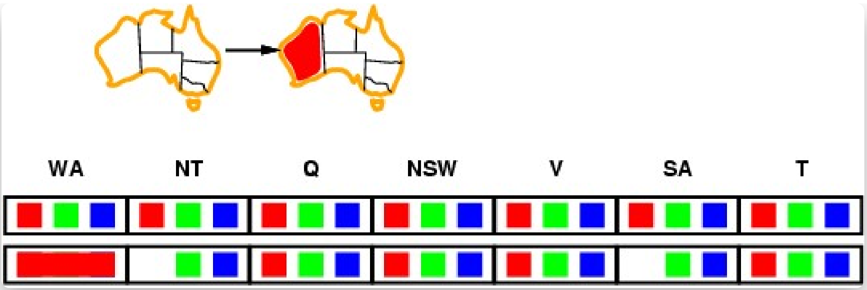
- Combining these heuristics makes 1000 queens feasible

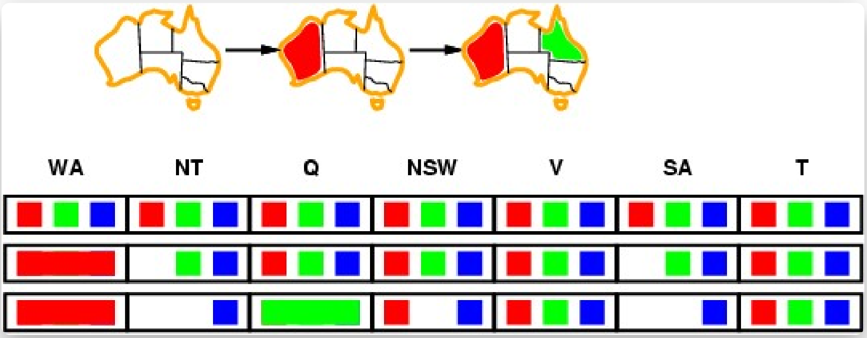
FORWARD CHECKING

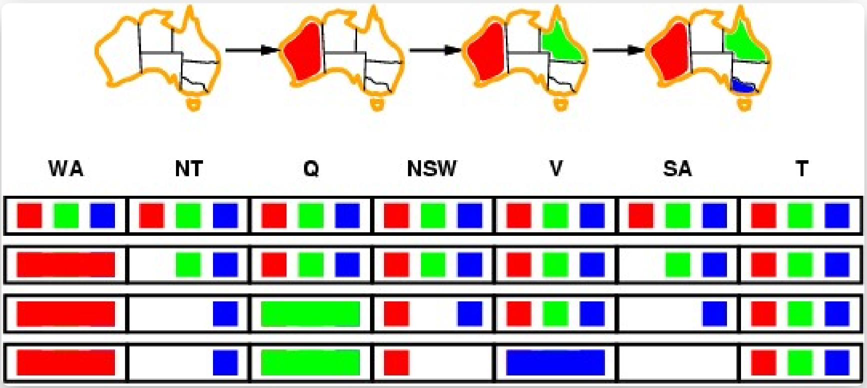
Idea

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



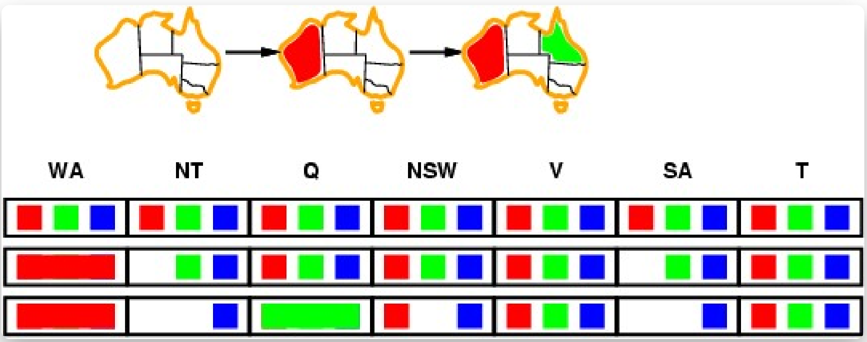






Constraint Propagation

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

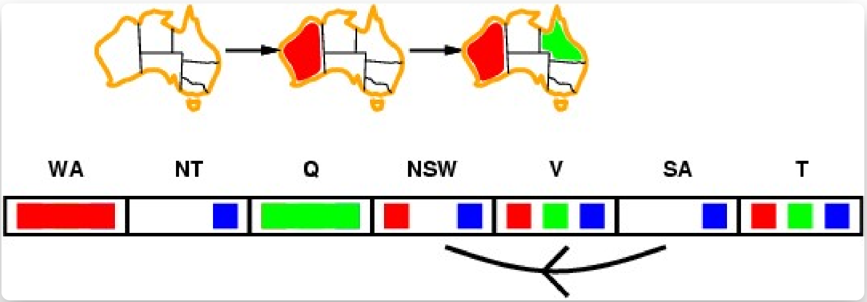


NT and SA cannot both be blue!

Constraint propagation repeatedly enforces constraints locally

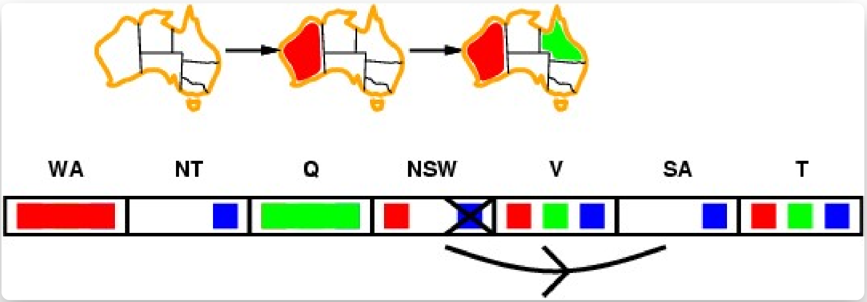
Arc consistency

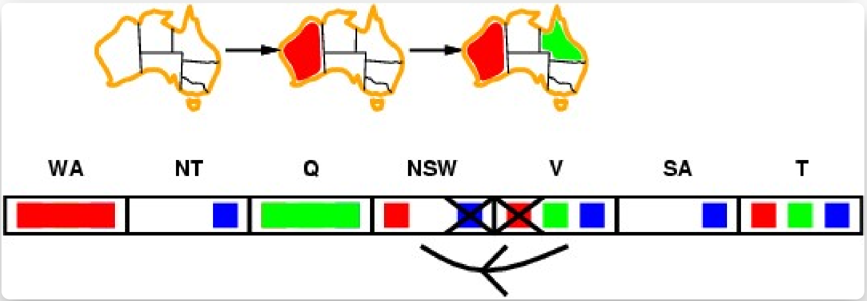
- Directed arc(Zi, Zj) is arc-consistent iff for any value of Zi there is a Zj value that is consistent with respect to Czizj



- Simplest form of propagation makes each arc consistent
  X -> Y is consistent iff for every value x of X there is some allowed y

  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 {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (Xi, Xj) ← REMOVE-FIRST(queue)
      if RM-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] do
            add (Xk, Xi) to queue
```

```
function RM-INCONSISTENT-VALUES( Xi, Xj) returns true iff remove a value
   removed ← false
   for each x in DOMAIN[Xi] do
      if no value y in DOMAIN[Xj] allows (x,y) to satisfy constraint(Xi, Xj)
         then delete x from DOMAIN[Xi];   removed ← true
   return removed
```

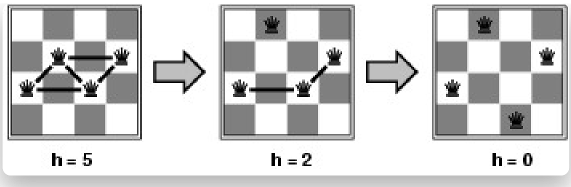
- Time complexity: $O(n^2 d^3)$

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 (44 = 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)

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