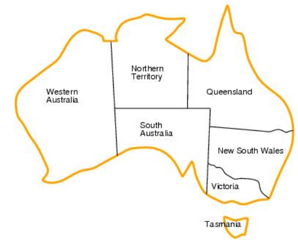
Constraint Satisfaction Problems (CSPs)

Russell and Norvig Chapter 6

CSP example: map coloring



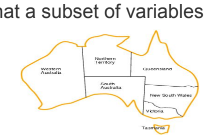
Given a map of Australia, color it using three colors such that no neighboring territories have the same color.

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 satisfaction problems

- A CSP is composed of:
  - A set of variables $X_1,X_2,\dots,X_n$ with domains (possible values) $D_1,D_2,\dots,D_n$
  - A set of constraints $C_1,C_2, \dots,C_m$
  - Each constraint C_i limits the values that a subset of variables can take, e.g., $V_1 \neq V_2$



In our example:

- 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) or
  - (WA,NT) in {(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green)}

Constraint satisfaction problems

- A state is defined by an assignment of values to some or all variables.
- Consistent (or legal) assignment: assignment that does not violate the constraints.
- Complete assignment: every variable is mentioned.
- Goal: a complete, consistent assignment.

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

Constraint satisfaction problems

- Simple example of a factored representation: splits each state into a fixed set of variables, each of which has a value
- CSP benefits
  - Standard representation language
  - Generic goal and successor functions
  - Useful general-purpose algorithms with more power than standard search algorithms, including generic heuristics
- Applications:
  - Time table problems (exam/teaching schedules)
  - Assignment problems (who teaches what)

Varieties of CSPs

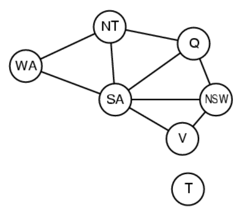
- Discrete variables
  - Finite domains of size $d \Rightarrow O(d^n)$ complete assignments.
    - The satisfiability problem: a Boolean CSP
      - (AvBvC)^(~BvCvD)^(~AvBv~D)…
  - Infinite domains (integers, strings, etc.)
    - e.g., job scheduling where variables are start/end times for each job.
    - Need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_2$.
- Continuous variables
  - e.g., start/end times for Hubble Telescope observations.
  - Linear constraints solvable in poly time by linear programming methods (dealt with in the field of operations research).

Varieties of constraints

- Unary constraints involve a single variable.
  - e.g., $SA \neq green$
- Binary constraints involve pairs of variables.
  - e.g., $SA \neq WA$
- Global constraints involve an arbitrary number of variables.
- Preference (soft constraints), e.g., *red* is better than *green;* often representable by a cost for each variable assignment; not considered here.

Constraint graph

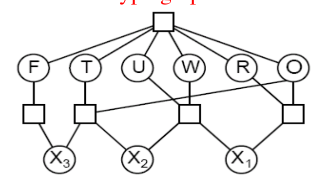
- Binary CSP: each constraint relates two variables
- Constraint graph: nodes are variables, edges are constraints



Example: cryptharithmetic puzzles

Hypergraph

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



Variables: $F, T, U, W, R, O, C_{10}, C_{100}, C_{1000}$

Domains: $\{0,1,2,3,4,5,6,7,8,9\}$

Constraints:

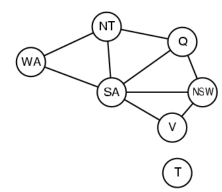
$alldiff(F, T, U, W, R, O)$

$O + O = R + 10 * C_{10}$

CSP as a standard search problem

- Incremental formulation
  - *Initial State*: the empty assignment {}.
  - *Successor function*: Assign value to unassigned variable provided that there is not conflict.
  - *Goal test*: the current assignment is complete.
- Same formulation for all CSPs !!!
- Solution is found at depth *n* (*n* variables).
  - What search method would you choose?

Constraint propagation



- Is a type of inference
  - Enforce local consistency
  - Propagate the implications of each constraint

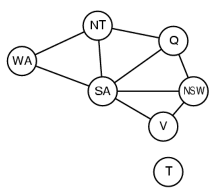
Arc consistency

- $X \rightarrow Y$ is arc-consistent iff
  for *every* value x of X there is some allowed y
- Constraint: $Y=X^2$ or $((X,Y), \{(0,0), (1,1), (2,4), (3,9)\}$
  - $X \rightarrow Y$ reduce X's domain to $\{0,1,2,3\}$
  - $Y \rightarrow X$ reduce Y's domain to $\{0,1,4,9\}$

Arc Consistency Algorithm

function AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise
  inputs: *csp*, a binary csp with components *{X, D, C}*
  local variables: *queue,* a queue of arcs initially the arcs in *csp*
  while *queue* is not empty **do**
    (X_i, X_j) ← REMOVE-FIRST(*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 ← *false*
  for each *x* **in** D_i **do**
    if no value *y* in D_j allows (x,y) to satisfy the constraints between X_i and X_j
    then delete x from D_i
    revised ← *true*
  return *revised*

Arc consistency limitations

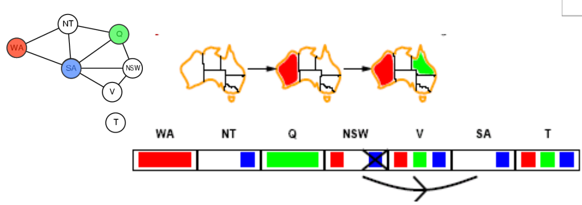


- $X \rightarrow Y$ is arc-consistent iff
  for *every* value x of X there is some allowed y
- *Yet SA* → *WA* is consistent under all of the following:
  - {(red, green), (red, blue), (green, red), (green, blue), (blue, red)}
- So it doesn't help

Path Consistency

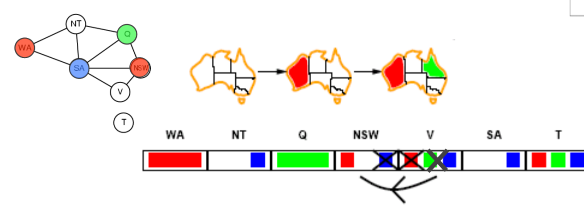
- Looks at triples of variables
  - The set $\{X_i, X_j\}$ is path-consistent with respect to X_m if for every assignment consistent with the constraints of X_i, X_j, there is an assignment to X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$

Path consistency



- If *SA=blue* and *NSW=red* is a consistent assignment wrt Q, then *SA* → *Q* → *NSW* is consistent.
- Arc can be made consistent by removing *blue* from *NSW*

Path consistency



- But need to RECHECK neighbors !!
  - Remove red and blue from *V* to ensure path-consistency for *SA* → *V* → *NSW*

K-consistency

- 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.
- Not practical!

Backtracking search

- Observation: the order of assignment doesn't matter

  ⇒ can consider assignment of a single variable at a time. Results in d^n leaves.
- Backtracking search: DFS for CSPs with single-variable assignments (backtracks when a variable has no value that can be assigned)
- The basic uninformed algorithm for CSP

Backtracking search

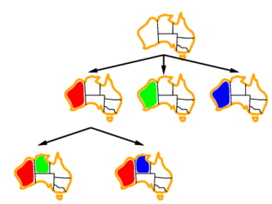
function BACKTRACKING-SEARCH(*csp*) **returns** a solution or failure
  return BACKTRACK({} , *csp*)

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

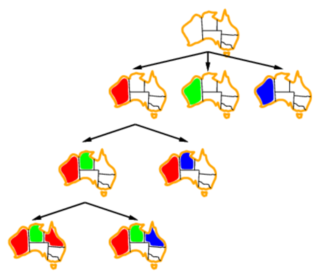
Backtracking example

Backtracking example

Backtracking example



Backtracking example



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?

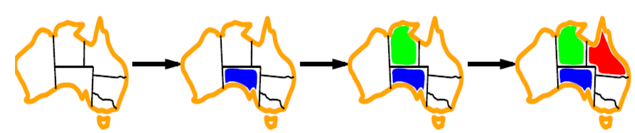
Most constrained variable

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

Choose the variable with the fewest legal values
(most constrained variable)
a.k.a. minimum remaining values (MRV) or "fail first" heuristic

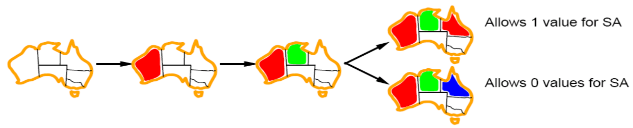
- What is the intuition behind this choice?

Degree heuristic



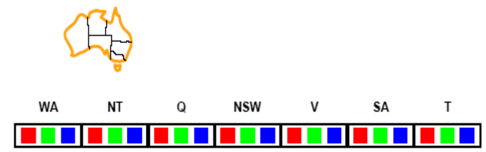
- Select the variable that is involved in the largest number of constraints on other unassigned variables.
- Often used as a tie breaker, e.g., in conjunction with MRV.

Least constraining value heuristic



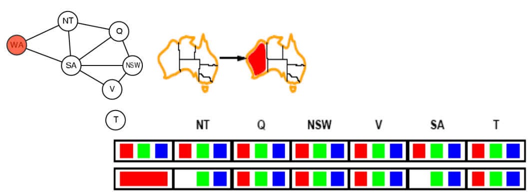
- Guides the choice of which value to assign next.
- Given a variable, choose the least constraining value:
  - the one that rules out the fewest values in the remaining variables
  - why?

Forward checking



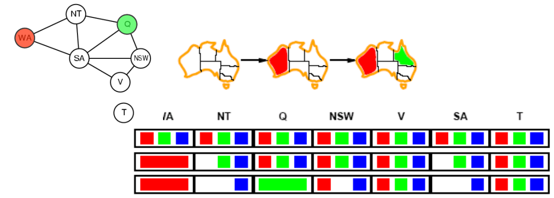
- Can we detect inevitable failure early?
  - *And avoid it later?*
- *Forward checking:* keep track of remaining legal values for unassigned variables.
- Terminate search direction when a variable has no legal values.

Forward checking



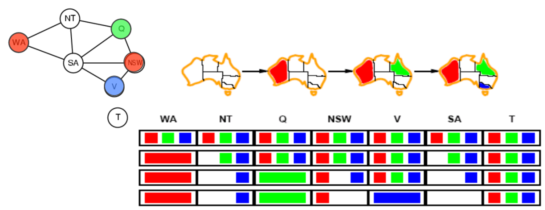
- Assign *{WA=red}*
- Effects on other variables connected by constraints with WA
  - *NT can no longer be red*
  - *SA can no longer be red*

Forward checking



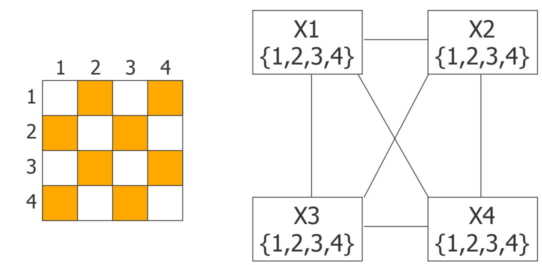
- Assign *{Q=green}*
- Effects on other variables connected by constraints with WA
  - *NT can no longer be green*
  - *NSW can no longer be green*
  - *SA can no longer be green*

Forward checking

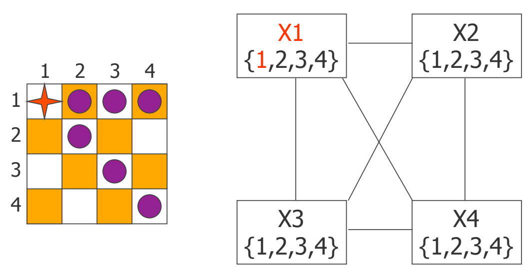


- If *V* is assigned *blue*
- Effects on other variables connected by constraints with WA
  - *SA is empty*
  - *NSW can no longer be blue*
- FC has detected that partial assignment is *inconsistent* with the constraints and backtracking can occur.

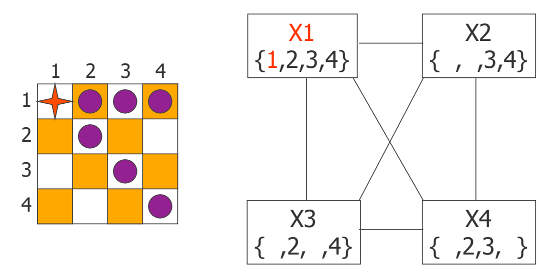
Example: 4-Queens Problem



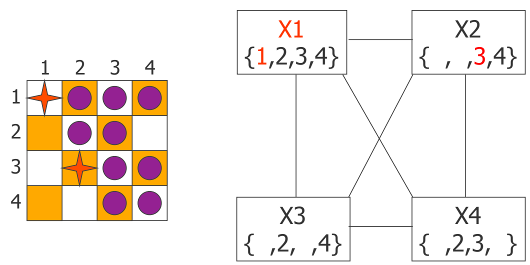
Example: 4-Queens Problem



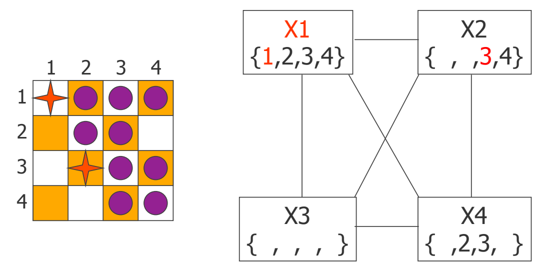
Example: 4-Queens Problem



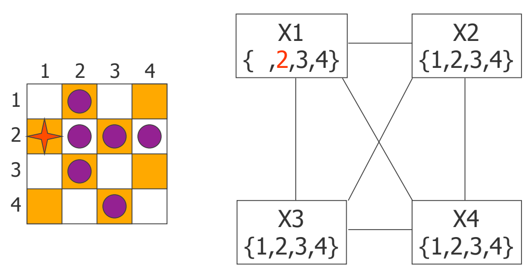
Example: 4-Queens Problem



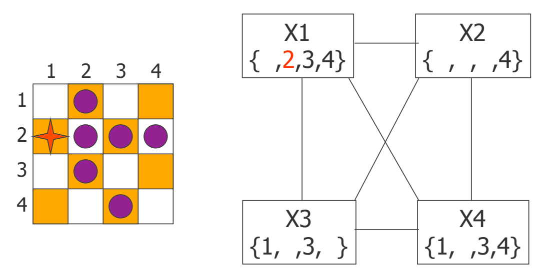
Example: 4-Queens Problem



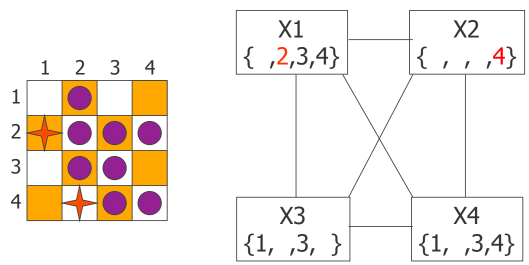
Example: 4-Queens Problem



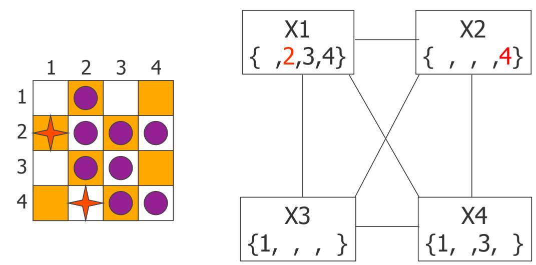
Example: 4-Queens Problem



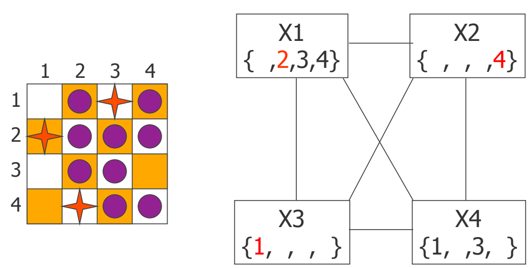
Example: 4-Queens Problem



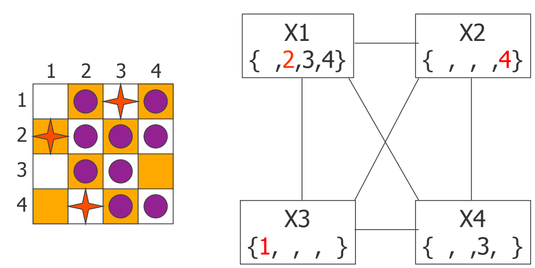
Example: 4-Queens Problem



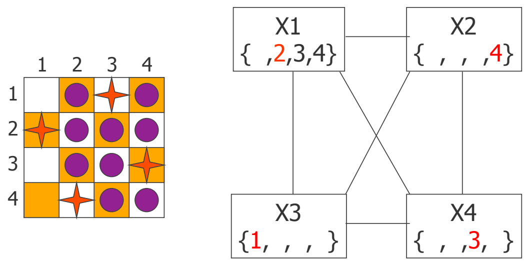
Example: 4-Queens Problem



Example: 4-Queens Problem



Example: 4-Queens Problem



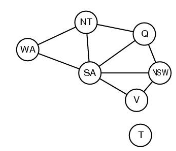
Local search for CSP

- Local search methods use a “complete” state representation, i.e., all variables assigned.
- To apply to CSPs
  - Allow states with unsatisfied constraints
  - operators reassign variable values
- Select a variable: random conflicted variable
- Select a value: *min-conflicts heuristic*
  - Value that violates the fewest constraints
  - Hill-climbing like algorithm with the objective function being the number of violated constraints
- Works surprisingly well in problem like n-Queens

Min-Conflicts

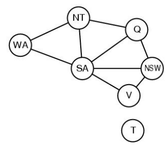
function MIN-CONFLICTS(*csp*, max_steps) **returns** a solution or failure
  inputs: *csp*, a constraint satisfaction problem
    max_steps, the number of steps allowed before giving up
  current ← an initial complete assignment for *csp*
  for *I* = 1 to *max_steps* **do**
    if *current* is a solution for *csp* **then return** *current*
    var← a randomly chosen conflicted variable from *csp*.VARIABLES
    value← the value *v* for *var* that minimizes CONFLICTS(*var, v, current, csp*)
    set *var*=*value* in *current*
  return *failure*

Problem structure



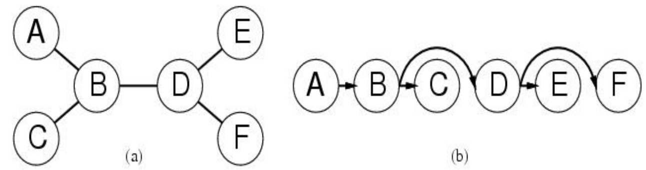
- *How can the problem structure help to find a solution quickly?*
- Subproblem identification is important:
  - Coloring Tasmania and mainland are independent subproblems
  - Identifiable as connected components of constraint graph.
- Improves performance

Problem structure



- Suppose each problem has *c* variables out of a total of *n*.
- Worst case solution cost is $O(n/c\ d^c)$ instead of $O(d^n)$
- Suppose *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



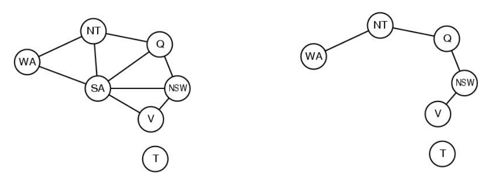
- Perform a topological sort of the variables
- Theorem: if the constraint graph has no loops then CSP can be solved in $O(nd^2)$ time
- Compare with general CSP, where worst case is $O(d^n)$

Tree-structured CSPs

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

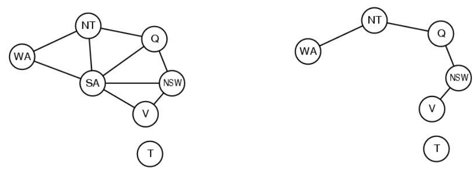
Function TREE-CSP-SOLVER(*csp)* **returns** a solution or failure
  inputs: *csp,* a CSP with components *X, D, C*
  n ← number of variables in *X*
  assignment ← an empty assignment
  root ← any variable in *X*
  X ← TOPOLOGICALSORT(*X, root)*
  for *j* = *n* **down to** 2 **do**
    MAKE-ARC-CONSISTENT(PARENT(X_j),X_j)
    if it cannot be made consistent **then return** *failure*
  for *i* = 1 **to** *n* **do**
    assignment[X_i] ← any consistent value from D_i
    if there is no consistent value **then return** *failure*
  ***return** assignment*

Nearly tree-structured CSPs



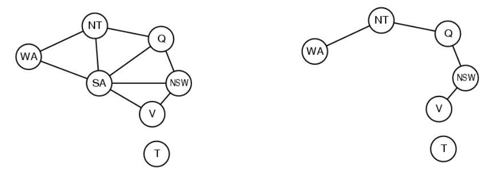
- *Can more general constraint graphs be reduced to trees?*
- Two approaches:
  - Remove certain nodes
  - Collapse certain nodes

Nearly tree-structured CSPs



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

Nearly tree-structured CSPs



- This approach is effective if cycle cutset is small.
- Finding the smallest cycle cutset is NP-hard
  - Approximation algorithms exist
- This approach is called *cutset conditioning*.