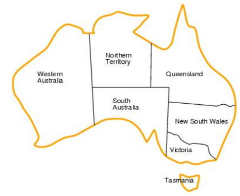
Constraint Satisfaction Problems (CSPs)

Russell and Norvig Chapter 6

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

CSP example: map coloring

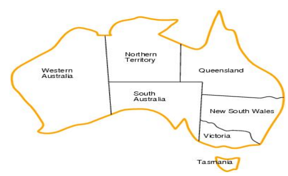


Given a map of Australia, color it using three colors such that no neighboring territories have the same color.

CSP example: map coloring

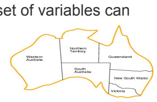
Constraint satisfaction problems

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



Constraint satisfaction problems

- A CSP is composed of:
  - A set of variables $X_1, X_2, \ldots, X_n$ with domains (possible values) $D_1, D_2, \ldots, D_n$
  - A set of constraints $C_1, C_2, \ldots, C_m$
  - 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 assignment: assignment that does not violate the constraints.
- Complete assignment: every variable is mentioned.
- Goal: a complete, legal assignment.

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

Constraint satisfaction problems

- Simple example of a formal representation language
- 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

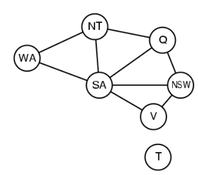
- Discrete variables
  - Finite domains of size $d \Rightarrow O(d^n)$ complete assignments.
    - The satisfiability problem: a Boolean CSP
  - Infinite domains (integers, strings, etc.)
- Continuous variables
  - Linear constraints solvable in poly time by linear programming methods (dealt with in the field of operations research).
- Our focus: discrete variables and finite domains

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.

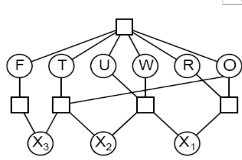
Constraint graph

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



Example: cryptarithmetic puzzles

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



Variables: $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$
Domains: $\{0,1,2,3,4,5,6,7,8,9\}$
Constraints
  alldiff(F,T,U,W,R,O)
  $O + O = R + 10 \cdot X_1$, etc.

The constraints are represented by a hypergraph

CSP as a standard search problem

- Incremental formulation
  - *Initial State*: the empty assignment {}.
  - *Successor*: Assign value to unassigned variable provided there is no 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?

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 (d: number of values per variable).
- 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

Sudoku solving

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

Sudoku solving

Constraints:

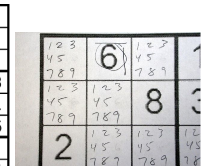
Alldiff(A1,A2,A3,A4,A5,A6,A7,A8,A9)
…
Alldiff(A1,B1,C1,D1,E1,F1,G1,H1,I1)
…
Alldiff(A1,A2,A3,B1,B2,B3,C1,C2,C3)
…

Can be translated into constraints between pairs of variables.

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

Sudoku solving

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



Let's see if we can figure the value of the center grid point.

Images from wikipedia and http://www.instructables.com/id/Solve-Sudoku-Without-even-thinking!/

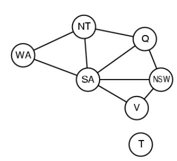
Solving Sudoku

"In this essay I tackle the problem of solving every Sudoku puzzle. It turns out to be quite easy (about one page of code for the main idea and two pages for embellishments) using two ideas: constraint propagation and search."

Peter Norvig

http://norvig.com/sudoku.html

Constraint propagation



- Enforce local consistency
- Propagate the implications of each constraint

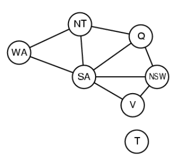
Arc consistency

- $X \rightarrow Y$ is arc-consistent iff
  for *every* value x of X there is some allowed value y of Y
- Example: X and Y can take on the values 0…9 with the constraint: $Y=X^2$. Can use arc consistency to reduce the domains of X and Y:
  - $X \rightarrow Y$ reduce X's domain to $\{0,1,2,3\}$
  - $Y \rightarrow X$ reduce Y's domain to $\{0,1,4,9\}$

The 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**
    $(X_i, X_j) \leftarrow$ 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*
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 *of* Y
- Consider mapping Australia with two colors. Each arc is consistent, and yet there is no solution to the CSP.
- So it doesn't help

Path Consistency

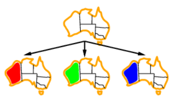
- Looks at triples of variables
  - The set $\{X_i, X_j\}$ is path-consistent with respect to X_m if for every assignment consistent with the constraints of X_i, X_j, there is an assignment to X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$
- The PC-2 algorithm achieves path consistency

K-consistency

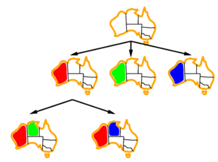
- Stronger forms of propagation can be defined using the notion of k-consistency.
- A CSP is k-consistent if for any set of k-1 variables and for any consistent assignment to those variables, a consistent value can always be assigned to any k-th variable.
- Not practical!

Backtracking example

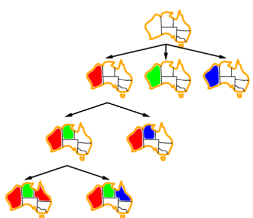
Backtracking example



Backtracking example



Backtracking example



Improving backtracking efficiency

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

Backtracking search

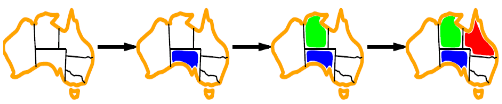
```
function BACKTRACKING-SEARCH(csp) return a solution or failure
    return RECURSIVE-BACKTRACKING({} , csp)

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


Most constrained variable

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

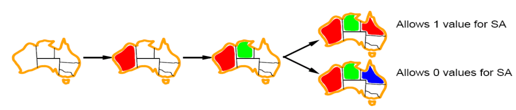
Choose the variable with the fewest legal values
(most constrained variable)
a.k.a. minimum remaining values (MRV) or “fail first” heuristic
- What is the intuition behind this choice?

Most constraining variable



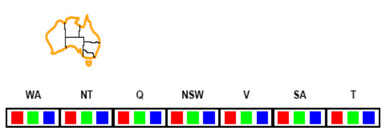
- Select the variable that is involved in the largest number of constraints on other unassigned variables.
- Also called the *degree* heuristic because that variable has the largest degree in the constraint graph.
- Often used as a tie breaker e.g. in conjunction with MRV.

Least constraining value heuristic



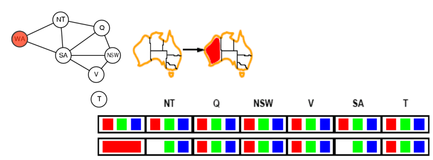
- Guides the choice of which value to assign next.
- Given a variable, choose the least constraining value:
  - the one that rules out the fewest values in the remaining variables
  - why?

Forward checking



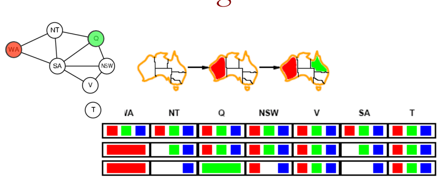
- Can we detect inevitable failure early?
  - *And avoid it later?*
- *Forward checking:* keep track of remaining legal values for unassigned variables.
- Terminate search direction when a variable has no legal values.

Forward checking



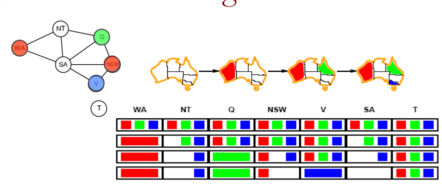
- Assign *{WA=red}*
- Effects on other variables connected by constraints with WA
  - *NT can no longer be red*
  - *SA can no longer be red*

Forward checking

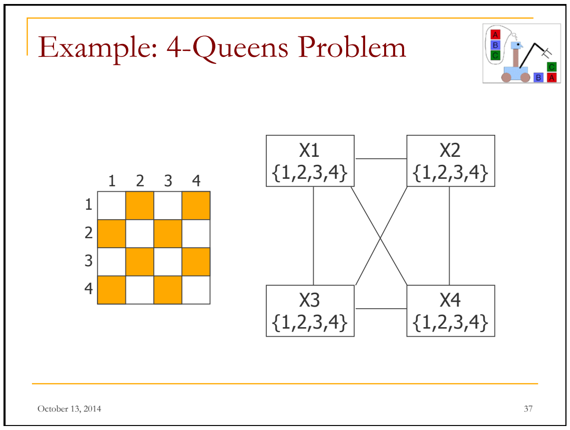


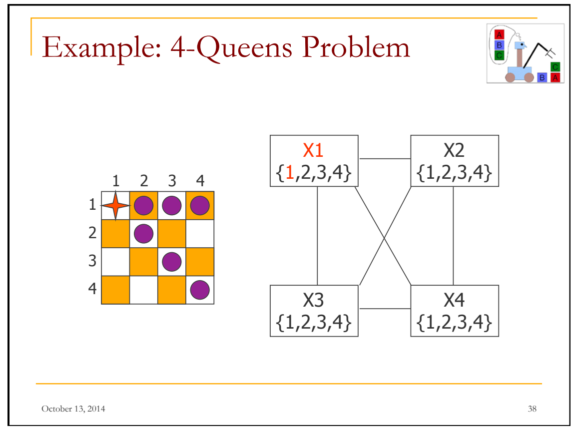
- Assign *{Q=green}*
- Effects on other variables connected by constraints with WA
  - *NT can no longer be green*
  - *NSW can no longer be green*
  - *SA can no longer be green*

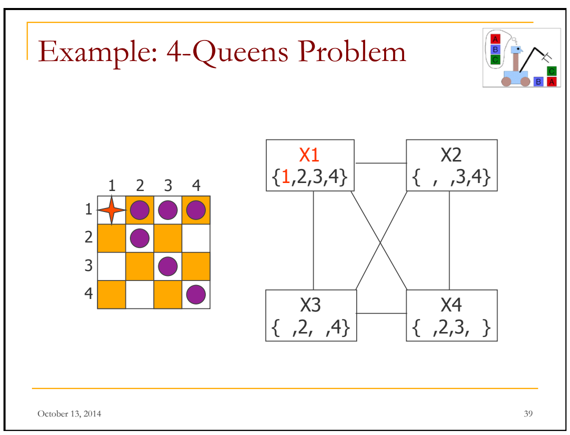
Forward checking

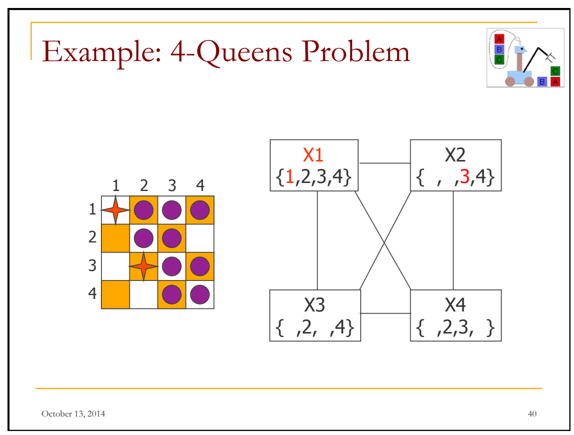


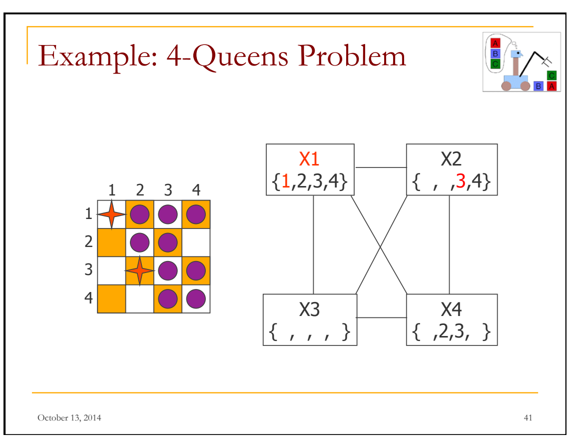
- If *V* is assigned *blue*
- Effects on other variables connected by constraints with WA
  - *SA is empty*
  - *NSW can no longer be blue*
- FC has detected that partial assignment is *inconsistent* with the constraints and backtracking can occur.

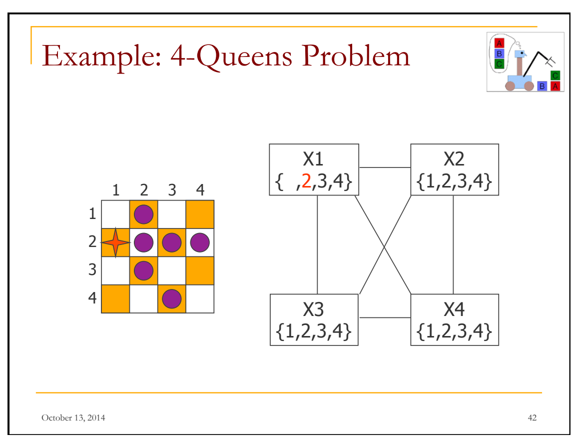

Example: 4-Queens Problem
1 2 3 4
1
2
3
4
X1
{1,2,3,4}
X2
{1,2,3,4}
X3
{1,2,3,4}
X4
{1,2,3,4}
October 13, 2014
37


Example: 4-Queens Problem
1 2 3 4
1
2
3
4
X1
{1,2,3,4}
X2
{1,2,3,4}
X3
{1,2,3,4}
X4
{1,2,3,4}
October 13, 2014
38

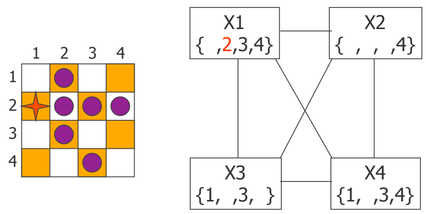

Example: 4-Queens Problem
1 2 3 4
1
2
3
4
X1
{1,2,3,4}
X2
{ , ,3,4}
X3
{ ,2, ,4}
X4
{ ,2,3, }
October 13, 2014
39


Example: 4-Queens Problem
1 2 3 4
1
2
3
4
X1
{1,2,3,4}
X2
{ , ,3,4}
X3
{ ,2, ,4}
X4
{ ,2,3, }
October 13, 2014
40

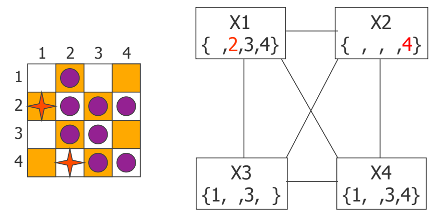

Example: 4-Queens Problem
1 2 3 4
1
2
3
4
X1
{1,2,3,4}
X2
{ , ,3,4}
X3
{ , , , }
X4
{ ,2,3, }
October 13, 2014
41


Example: 4-Queens Problem
1 2 3 4
1
2
3
4
X1
{ ,2,3,4}
X2
{1,2,3,4}
X3
{1,2,3,4}
X4
{1,2,3,4}
October 13, 2014
42

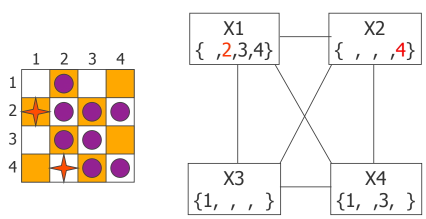
Example: 4-Queens Problem



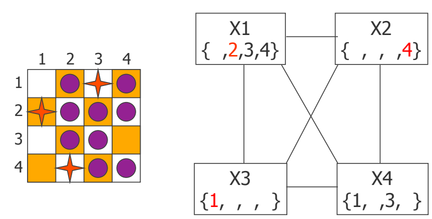
Example: 4-Queens Problem



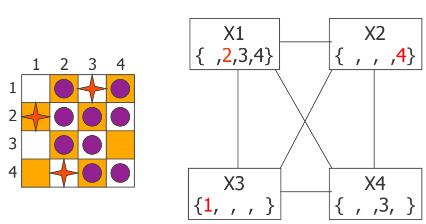
Example: 4-Queens Problem



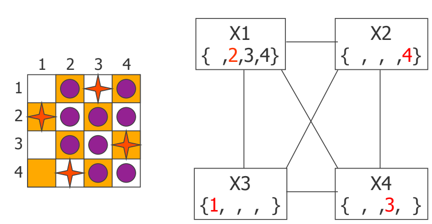
Example: 4-Queens Problem



Example: 4-Queens Problem



Example: 4-Queens Problem



Forward checking

- Solving CSPs with combination of heuristics plus forward checking is more efficient than either approach alone.
- FC does not provide early detection of all failures.
  - Once WA=red and Q=green: NT and SA cannot both be blue!
- MAC (maintaining arc consistency): calls AC-3 after assigning a value (but only deals with the neighbors of a node that has been assigned a value).

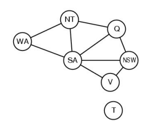
Local search for CSP

- Local search methods use a "complete" state representation, i.e., all variables assigned.
- To apply to CSPs
  - Allow states with unsatisfied constraints
  - 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 problems like n-Queens

Min-Conflicts

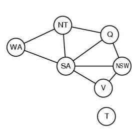
```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
    inputs: csp, a constraint satisfaction problem
            max_steps, the number of steps allowed before giving up
    current ← an initial complete assignment for csp
    for i = 1 to max_steps do
        if current is a solution 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



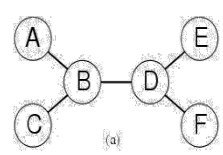
- *How can the problem structure help to find a solution quickly?*
- Subproblem identification is important:
  - Coloring Tasmania and mainland are independent subproblems
  - Identifiable as connected components of constraint graph.
- Improves performance

Problem structure



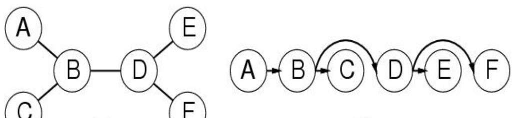
- Suppose each problem has *c* variables out of a total of *n*.
- Worst case solution cost is $O(n/c\ d^c)$ instead of $O(d^n)$
- Suppose *n=80, c=20, d=2*
  - 2^{80} = 4 billion years at 1 million nodes/sec.
  - 4 * 2^{20}= .4 second at 1 million nodes/sec

Tree-structured CSPs



- 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



(a) (b)

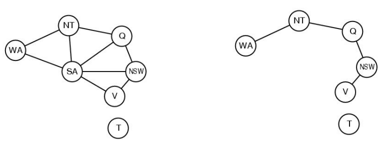
- Any tree-structured CSP can be solved in time linear in the number of variables.
  - Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering. (label var from X_1 to X_n)
  - For *j* from *n* down to 2, apply REMOVE-INCONSISTENT-VALUES(Parent(X_j), X_j)
  - For *j* from 1 to *n* assign X_j consistently with Parent(X_j)

Tree-structured CSPs

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

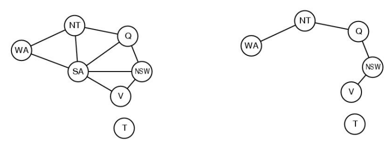
Function TREE-CSP-SOLVER(*csp)* **returns** a solution or failure
    inputs: *csp,* a CSP with components *X, D, C*
    n ← number of variables in *X*
    assignment ← an empty assignment
    root ← any variable in *X*
    X ← TOPOLOGICALSORT(*X, root)*
    for *j* = *n* **down to** 2 **do**
        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



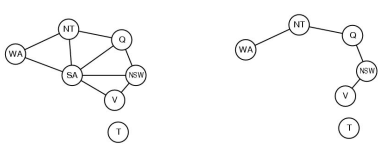
- *Can more general constraint graphs be reduced to trees?*
- Two approaches:
  - Remove certain nodes
  - Collapse certain nodes

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

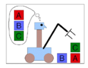


- This approach is effective if cycle cutset is small.
- Finding the smallest cycle cutset is NP-hard
  - Approximation algorithms exist
- This approach is called *cutset conditioning*.

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.
- Structure of CSP affects its complexity. Tree structured CSPs can be solved in linear time.

Interim class summary



- We have been studying ways for agents to solve problems.
- Search
  - Uninformed search
    - Easy solution for simple problems
    - Basis for more sophisticated solutions
  - Informed search
    - Information = problem solving power
  - Adversarial search
    - $\alpha\beta$-search for play against optimal opponent
    - Early cut-off when necessary
  - Constraint satisfaction problems
- What's next?
  - Logic
  - Machine learning