CS 4100/5100: Foundations of AI

Constraint satisfaction problems [1]

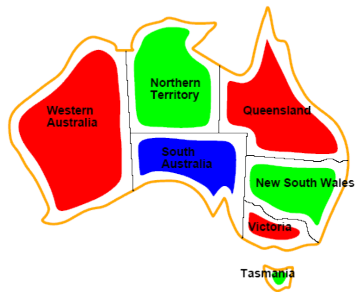
Instructor: Rob Platt
r.platt@neu.edu

College of Computer and information Science
Northeastern University

September 5, 2013

[1] These notes draw heavily from Hwee Tou Ng's slides (http://www.comp.nus.edu.sg/~nght/). I also draw from Gillian Smith's slides (http://sokath.com/main/).

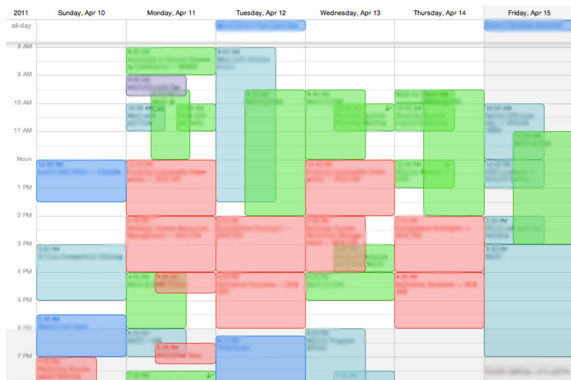
What is a constraint satisfaction problem (a CSP)?



You need to assign discrete values to a set of variables such that a set of constraints is satisfied:

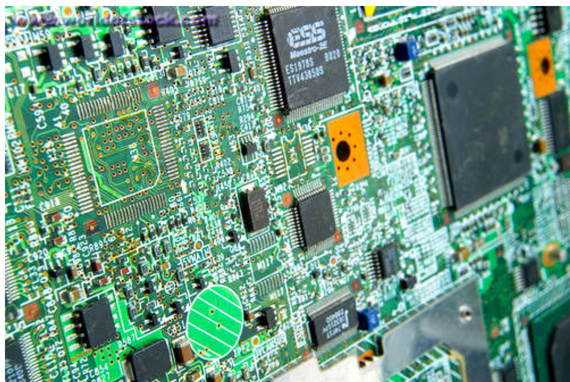
- each province gets a color assignment
- no two adjacent provinces can have the same color.

Examples of why CSPs are useful: scheduling



- what are the variables?
- what values do those variables take?
- what are the constraints?

Examples of why CSPs are useful: pcb board layout

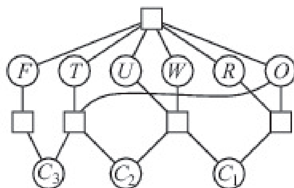


- what are the variables?
- what values do those variables take?
- what are the constraints?

Examples of why CSPs are useful: puzzles

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

(a)



(b)

- what are the variables?
- what values do those variables take?
- what are the constraints?

Examples of why CSPs are useful: Sudoku

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

(a)

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

(b)

- what are the variables?
- what values do those variables take?
- what are the constraints?

Types of CSPs

- discrete vs continuous
  - in this course, we assume CSPs are discrete.
  - linear programming, quadratic programming, *etc.* can be viewed as continuous CSPs
- constraints branching factor: unary constraints, binary constraints, *etc.*.
  - what type of constraint is in the coloring problem?
  - what type of constraint in Sudoku?
- any constraint graph with more than binary constraints can be converted into a binary constraint graph.
  - don't always do this, but is it often convenient
  - assume from here on out that we are dealing w/ binary constraint graphs

Solving CSPs

How do you think we might do it?

Solving CSPs

How do you think we might do it?

- DFS again...
- How would that work?

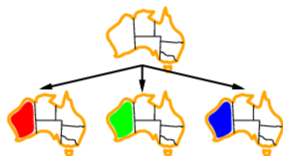
Backtracking search

Example:

Each ply of the tree corresponds to a single variable that will be assigned (because the tree is *commutative*).

Backtracking search

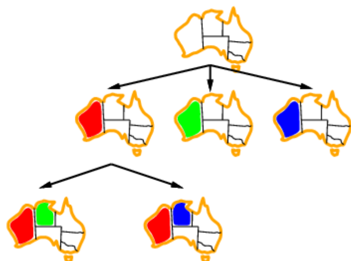
Example:



Each ply of the tree corresponds to a single variable that will be assigned (because the tree is *commutative*).

Backtracking search

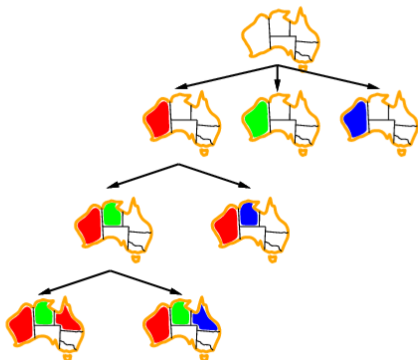
Example:



Each ply of the tree corresponds to a single variable that will be assigned (because the tree is *commutative*).

Backtracking search

Example:



Each ply of the tree corresponds to a single variable that will be assigned (because the tree is *commutative*).

Backtracking search

Backtracking search

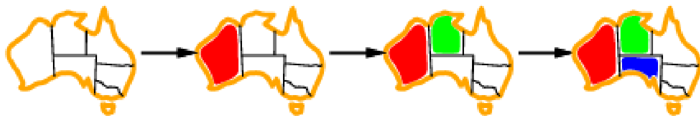
```
function BACKTRACKING-SEARCH( csp) returns a solution, or failure
   return RECURSIVE-BACKTRACKING({}, csp)

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
```

Improving backtracking efficiency

There are a few heuristic strategies that generally speed up backtracking.

Minimum remaining values (MRV) heuristic:

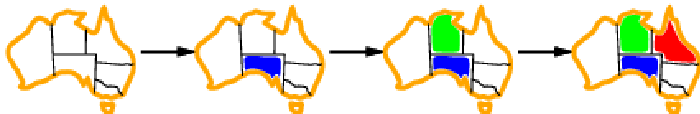
- Which variable does the algorithm expand next?
  - Plain backtracking allows you to expand any unassigned variable...
- choose the variable with the fewest legal values



Improving backtracking efficiency

Degree heuristic:

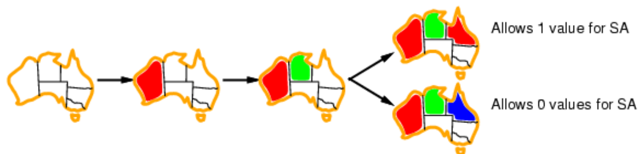
- choose the variable with the highest branching factor.
  - MRV is typically more useful, but Degree can help if multiple variables have the same MRV.



Improving backtracking efficiency

Least constraining value (LCV) heuristic:

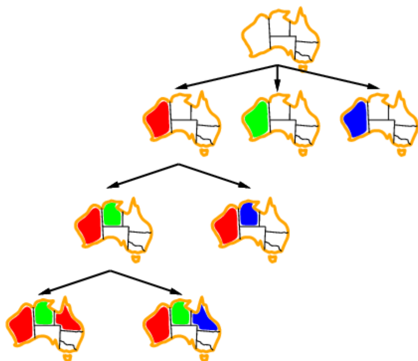
- this heuristic chooses the order in which *values* (not *variables*) are assigned
- assign values in the least constaining order



What else can we do to improve backtracking efficiency?

Notice that as we expand the search tree, our prior value choices constrain future choices.

For example, what should be the branching factor for the leftmost leaf nodes?

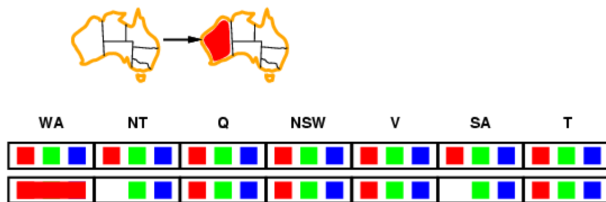


Forward checking

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

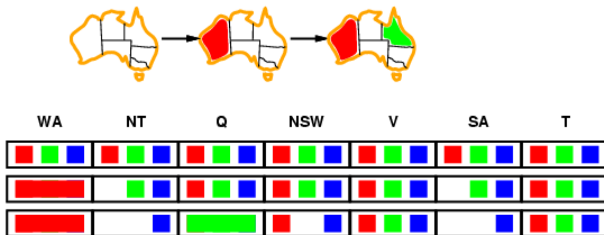
Forward checking

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



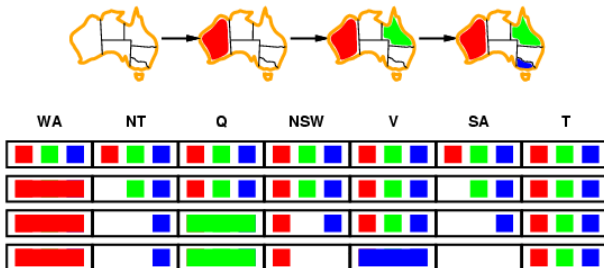
Forward checking

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



Forward checking

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



Arc consistency

Given a pair of variables, x and y, that are connected by a binary constraint: variable x is *arc consistent* with variable y if for every value in the domain D_x, there is a value in D_y that satisfies the binary constraint, (x, y).

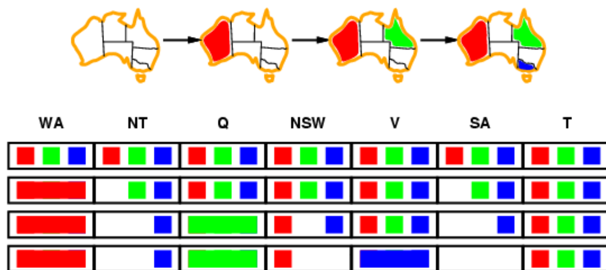
Example: $y = x^2$, $x = \{0, \ldots, 10\}$, $y = \{0, \ldots, 10\}$

- x is not AC wrt y. y is not AC wrt x.
- if $x = \{0, 1, 2, 3\}$, then x is AC wrt y (but not vice versa).
- if $y = \{0, 1, 4, 9\}$, then y is also AC wrt x.

You can make a variable AC wrt to its neighbors by deleting values which are inconsistent w/ the neighbors.

- this is what we're doing w/ forward checking.
- each time we assign a variable, we do ONE step of AC w/ all of that variable's neighbors (make the neighbors AC wrt to it)

Forward checking



- Notice forward checking must get to the very end before it recognizes that there is a conflict.
- Constraint propagation can detect this kind of failure earlier.

Constraint propagation: AC3

AC3: makes every variable in the graph arc consistent w/ every other variable.

- there is also such a thing as *node consistency*. Guess what that does...

Algorithm: AC3

- initialize a queue w/ the set of all constraint arcs in the graph (both directions)
- pop an arc (x, y) off of the queue and make x AC wrt y.
- if D_x is unchanged, then move on to the next arc in the queue
- if D_x gets smaller, then add all arcs GOING INTO x to the end of the queue. Then, move on to the next arc in the queue.
- if the domain of any variable reaches zero, then the graph is no longer AC!

Constraint propagation: AC3

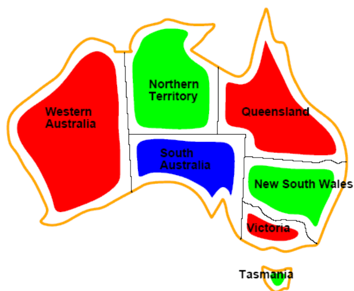
You can use AC3 before doing any search to reduce the amount of search that needs to be done.

You can also use AC3 after each step of search to "update" the values to be searched with the latest information: this is MAC (maintaining arc consistency).

- suppose that backtracking search has just assigned a value to x
- create a queue comprised only of arcs going from the neighbors of x to x.
- do AC3 on that queue.

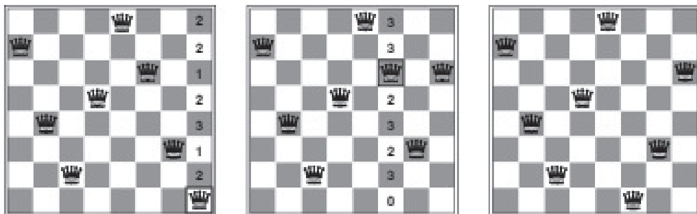
Constraint propagation is strictly more powerful than forward checking.

Constraint propagation: AC3



WA	NT	Q	NSW	V	SA	T

Local search



Min conflicts heuristic:

- choose a random variable that is in conflict
- assign that variable the value which minimizes the number of pairwise conflicts
- repeat

Slightly different from hill climbing:

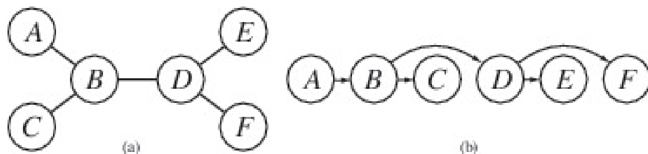
- remember in HC, we had a *complete assignment* to the variables and made incremental changes in queen locations.
- here, we are adding queens in a least-constrained way.
- works well for n-queens because solutions are fairly dense in the search space...

Problem structure

So far, we've talked about search for general CSP problems. But, some problems have special structure...

- disconnected componenents in the constraint graph
- tree struture.

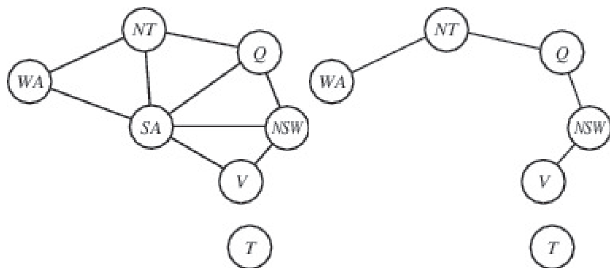
Problem structure: trees



Trees can be solved in time linear in the number of variables

- First, do a *topological sort* of the nodes.
- Starting w/ the leaf node (last node in the sorted list), make the parent AC w/ it.
- Continue w/ the next node in the list...
- complexity?

Problem structure: almost trees



Suppose you have a problem that is *almost* a tree, except for one or two nodes. Then:

- Exhaustively set all possible values for the "exception nodes".
- For each value setting, do CSP-Trees.

Value symmetry

Many CSPs have value symmetries.

- for example, the australia problem has symmetry in the colors
- you can avoid searching the extra variables by imposing an arbitrary symmetry-breaking constraint on the variables; for example: $NT < SA < WA$.