

Constraint Satisfaction Problems

Petros Papapanagiotou

pe.p@ed.ac.uk

7 Feb 2012



Aim

- Familiarise yourselves with Constraint Satisfaction Problems (CSPs)
- Implement and evaluate basic algorithms using Haskell.

2 /35

Tools

- A CSP framework in Haskell.

<http://www.inf.ed.ac.uk/teaching/courses/inf2d/coursework/CSPReference.pdf>

3 /35

The framework

- Datatypes and functions for: *(CSPframework.hs)*
 - Variables
 - Assignments
 - Domains
 - Constraints
 - CSPs
- Examples of: *(Examples.hs)*
 - Constraints
 - CSPs

4 /35

A CSP

- Consists of:
 - **X** : a set of **Variables**
 - **D** : a set of **Domains**
 - **C** : a set of **Constraints**

5 /35

Datatypes

- **Var** : *String*
eg. "X" "Y"
- **Domain** : *List of Int's*
eg. [1, 2, 3]
 - Each value can have an implicit meaning
eg. colour, position, actual value, etc.
- **Domains** : *List associating each variable with a Domain*
eg. [("X", [1, 2, 3]), ("Y", [1, 2])]

6 /35

Assignment

- "State"
- "Assignment of values to some or all of the variables" - R&N §6.1
- Custom datatype - *List of assigned variables*
- Functions:
 - `assign` : Assign a value to a variable and add it to the list.
 - `lookup_var` : Lookup a value of a variable.
 - `is_unassigned` : Is a variable NOT yet assigned?

7 /35

Constraints

- "Each constraint C_i consists of a pair: $\langle \text{scope}, \text{rel} \rangle$ " - R&N §6.1
 - Custom datatype – *Pair of a scope and a Relation*
 - Scope as a list of variables:
eg. ["X", "Y"]
 - Relation as a function:
 - "Give me a scope and a state and I'll tell you if it's ok!"
- eg. `vars_diff` : The first two variables in the scope must have different values.
- `vars_diff` ["x", "y"] [x=1, y=2] → True
 - `vars_diff` ["x", "y"] [x=1, y=1] → False

8 /35

Constraint functions

- `check_constraint` : Check a constraint on a given state.
- `check_constraints` : Check multiple constraints on a given state.
- `scope` : Get the scope of a constraint.
- `is_constrained` : Is a variable within the scope?
- `neighbours_of` : List of other variables in the same scope.

9 /35

A CSP

- Consists of:
 - **X** : a set of **Variables**
 - **D** : a set of **Domains**
 - **C** : a set of **Constraints**
- Custom datatype
 - Includes a Domains list and a list of Constraints
- Functions:
 - `vars_of` : Returns **X**
 - `domains` : Returns **D**
 - `constraints` : Returns **C**

10 /35

CSP functions

- Domain functions: `domain_of`, `set_domain`, `add_domain_val`, etc.
- Variable functions: `get_unassigned_var`, `constraints_of`, etc.
- Assignment functions: `is_complete`, `is_consistent`, etc.

11 /35

The BACKTRACK algorithm

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

12 /35

The BACKTRACK algorithm

```
bt :: CSP -> Maybe Assignment
bt csp = bt_recursion [] csp

bt_recursion :: Assignment -> CSP -> Maybe Assignment
bt_recursion assignment csp =
  if (is_complete csp assignment) then Just assignment
  else find_consistent_value $ domain_of csp var
  where var = get_unassigned_var csp assignment
        find_consistent_value vals =
          case vals of -- recursion over the possible values
                    -- instead of for-each loop
          [] -> Nothing
          val:vs ->
            if (is_consistent_value csp assignment var val)
            then if (isNothing result)
                  then ret
                  else result
            else ret
            where result = bt_recursion (assign assignment var val) csp
                  ret = find_consistent_value vs
```

13 /35

Coursework 1

1. The “Cluedo” CSP (10%)
2. CSP Algorithms (45%)
 - 2.1. Forward Checking (20%)
 - 2.2. Minimum Remaining Values (MRV) (10%)
 - 2.3. Least Constraining Value (LCV) (15%)
3. Evaluation (25%)
4. Arc Consistency Algorithm AC-3 (20%)

14 /35

“Cluedo” CSP



15 /35

“Cluedo” CSP

- 5 houses: 
- 5 colours: **green, red, blue, yellow, white**
- 5 suspects: 
- 5 weapons: 

16 /35

“Cluedo” CSP

- 5 rooms:
the kitchen the lounge the billiard room
the library the dining room
- 5 people:
 - one accomplice, one innocent, one sleeping, one studying, one **murderer!**
- 14 Clues!

Who murdered Dr.Black, in which room, and with which weapon?

17 /35

Clues:

- Miss Scarlett lives in the green house.
- Colonel Mustard is innocent.
- The owner of the red house was in the lounge.
- Mrs. Peacock was in the kitchen.
- The red house is on the right of the blue one.
- The dagger was with whoever was sleeping.
- The lead pipe was found in the yellow house.
- The owner of the middle house was in the billiard room.
- Professor Plum lives in the first house on the left.
- The witness lives in a house next to that where the Rope was found.
- The lead pipe was found in a house next to the person who was studying.
- The revolver was found in the Library.
- Reverend Green was holding the Candlestick.
- Professor Plum's house is next to the white one.

18 /35

Task 1 (10%)

- Define the “Cluedo” CSP within the framework.
- Tip: Check examples in framework: Map of Australia, Map of Scotland, 3x3 Magic Square, Sudoku
- Define the variables and their domains.
- Define the constraints using available Relations plus 2 of your own:
 1. `has_value` : A variable must have the given value.
 2. `vars_same` : Two variables must have the same value.
- Test using BT.

19 /35

Forwardcheck (20%)

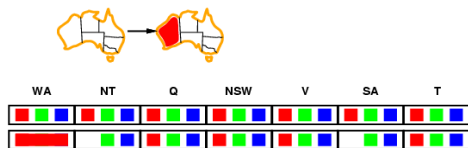
- Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is a neighbour of X .
- Deletes from Y 's domain any value that is inconsistent with the value chosen for X .



20 /35

Forwardcheck (20%)

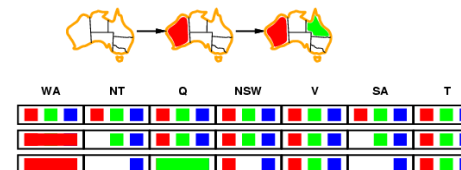
- Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is a neighbour of X .
- Deletes from Y 's domain any value that is inconsistent with the value chosen for X .



21 /35

Forwardcheck (20%)

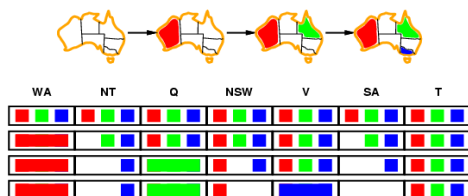
- Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is a neighbour of X .
- Deletes from Y 's domain any value that is inconsistent with the value chosen for X .



22 /35

Forwardcheck (20%)

- Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is a neighbour of X .
- Deletes from Y 's domain any value that is inconsistent with the value chosen for X .



23 /35

Forwardcheck (20%)

- Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is a neighbour of X .
- Deletes from Y 's domain any value that is inconsistent with the value chosen for X .
- Implement:
 - `forwardcheck`: Given X and the current state, apply forwardchecking.
 - `fc_recursion`: Same as `bt_recursion` but with added forwardchecking.

24 /35

Minimum Remaining Values (MRV) (10%)

- **Variable ordering** heuristic.
- Selects the *variable* that has the *least available values*.
- Implement:
 1. `mrv_compare`: Given 2 variables returns the ordering of the sizes of their domains: LT / EQ / GT
eg. "X" ∈ [1, 2, 3] - "Y" ∈ [1, 2]
`mrv_compare` returns GT
 2. `get_mrv_variable`: Returns the unassigned variable that has the smallest domain.
 - Tip: Use `sortBy`!
 3. `fc_mrv_recursion`: Same as `fc_recursion` but with MRV!

25 /35

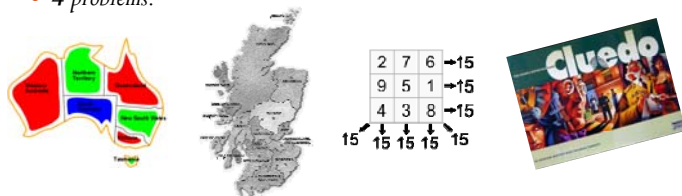
Least Constraining Value (LCV) (15%)

- **Value ordering** heuristic.
- Selects the *value* that has the value that allows the most choices for the neighbours.
- Implement:
 1. `num_choices`: Given a variable X, returns the (total) number of values (choices) allowed for all neighbours of X.
 - Assumes X is already assigned!
 2. `lcv_sort`: Returns domain of a variable sorted with LCV.
 3. `fc_lcv_recursion`
 4. `fc_mrv_lcv_recursion`

26 /35

Evaluation & Discussion (25%)

- 5 algorithms:
 - BT FC FC+MRV FC+LCV FC+MRV+LCV
- 4 problems:



- Evaluation: 4x5 Table of *visited nodes*.
 - A search node is considered *visited* when a value has been assigned to a variable.

27 /35

Evaluation & Discussion (25%)

- Report:
 1. Explain *briefly* the values on the table.
 2. Compare the following pairs of algorithms:
 - BT vs. FC
 - FC vs. FC+MRV
 - FC vs. FC+LCV
 - FC+MRV vs. FC+LCV
 - FC+MRV vs. FC+MRV+LCV
 3. Compare values, and explain the differences.
 - Expected or not?
 4. Is there a **simple, effective** optimisation for FC?
 - Consider large scale CSPs
 - Give suggestions and arguments
- No longer than one A4 page of plain text.

28 /35

Arc Consistency Algorithm AC-3 (20%)

- Constraint propagation algorithm.
- An arc between two variables X and Y is *consistent* if
 - for every value x of variable X
 - there is a *possible* value of Y that is consistent with x .
- If there is a value x' such that if it is assigned to X and there is **no consistent value** for Y then we *remove* x' from X 's possible values (domain).
- eg. " X " $\in [1, 2]$ - " Y " $\in [1]$ - Constraint: $X \neq Y$
 - To make $X \rightarrow Y$ consistent, AC-3 will *revise* the domain of X by *removing* 1.

29 /35

Arc Consistency Algorithm AC-3 (20%)

- R&N - Figure 6.3

```
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 all 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
  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
```

30 /35

Arc Consistency Algorithm AC-3 (20%)

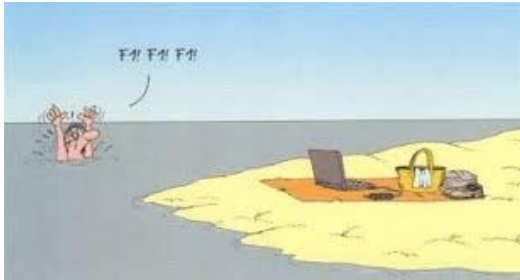
- Implement:
 1. `exists_consistent_value`: Given two variables X and Y and a value x for X , checks if there exists a value in the domain of Y such that all constraints between X and Y are satisfied.
 - Assumes X and Y are neighbours.
 2. `revise`: The REVISE function of Fig 6.3. Returns a possibly revised CSP and True if the domain of X_i was revised (False if not).
 3. `ac3_check`: The AC-3 function of Fig 6.3. Returns an updated CSP and True if no inconsistencies were found (False otherwise).
 4. `ac3_recursion`
 5. `ac3_mrv_lcv_recursion`

31 /35

Arc Consistency Algorithm AC-3 (20%)

- Report:
 - Add visited nodes to the table.
 - Compare:
 - FC vs. AC3
 - FC+MRV+LCV vs. AC3+MRV+LCV
 - **No longer than an additional half A4 page of plain text.**








32 /35



- Theory: *Lecture 6 – R&N § 6.1-6.3*
- Framework: *Reference Manual & code*
- Drop-in lab: *Every Wed 16:00-18:00 South Lab*
- Me : pe.p@ed.ac.uk - Mailing list: inf2d-students@inf.ed.ac.uk
 - Haskell: www.haskell.org
- Haskell refresher lecture for Coursework 1: *Thu 09/02/2011*

33 /35



- **Comment** your code! 
- Make sure it **loads** on *ghci*! 
- **Max runtime**: 5 minutes 
 - (NOT for Sudoku)
- **Any** custom functions you want **BUT** comment and explain! 
- Do **NOT** change names or type definitions!
- Create **unit tests**. 
- No plagiarism!  

34 /35

Deadline: 
27th February 2012 – 2pm

- *Start early!*
- *Good luck!*

35 /35