Informatics 2D. Course work 1: Constraint Satisfaction Problems

Petros Papapanagiotou, Jacques Fleuriot

January 23, 2012

1 Introduction

The aim of this assignment is to familiarise you with Constraint Satisfaction Problems (CSPs) and the basic algorithms that are used to solve them. You will implement, use and evaluate CSPs and their algorithms using Haskell.

You should download the following file from the Inf2D web page:

http://www.inf.ed.ac.uk/teaching/courses/inf2d/coursework/Inf2dAssignment1.tar.gz

Use the command **tar -xvf Inf2dAssignment1.tar.gz** to extract the files contained within. This will create a directory named *Inf2dAssignment1/* containing three files: *CSPframework.hs*, *Examples.hs*, and *Inf2d.hs*.

You will submit a version of the *Inf2d.hs* file containing your implemented functions and a small report. Remember to add your **matriculation number** to the top of the file.

The deadline for the assignment is:
Monday, 27th of February 2012 at **2pm**.

Submission details can be found in Section 8.

There will be a lecture discussing this assignment on **Tuesday the 7th of February 2012** at the standard lecture time and place. There will also be weekly drop-in labs on **Wednesdays from 4pm until 6pm** in Appleton Tower 5.08 (Computer Lab South) where Petros Papapanagiotou will be available to answer your questions and provide appropriate help.

The assignment will build upon a framework for CSPs in Haskell. This framework consists of datatypes and functions that you will use in your implementation, as well as the simple Backtracking (BT) algorithm (R&N Fig. 6.5). The code for the framework can be found in the *CSPframework.hs* file. A reference manual for the framework is also available on the Inf2D web page:

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

The following CSPs are already implemented for this framework and are available to help you understand the CSP framework and test and evaluate your various implementations:

- The Map of Australia (R&N §6.1.1)
- The Map of Scotland
- The 3x3 Magic Square
- A Sudoku problem (R&N Fig: 6.4)

The description of these problems can be found in the CSP framework reference manual. You can use this code to gain a better understanding of the framework and as a starting point for your own implementation.

The assignment is divided into four sections:

1. In the first section you are asked to implement the "Cluedo" game as a CSP.
2. In the second section you will implement the Forward Chaining (FC) algorithm, the Minimum Remaining Values (MRV) heuristic, and the Least Constraint Value (LCV) heuristic.
3. In the third section you are asked to evaluate and compare the algorithms using the implemented CSPs.
4. Finally, in the fourth section you will implement the Arc Consistency (AC-3) algorithm and add it to your evaluation.

If your Haskell is quite rusty, you should revise the following topics:

- Recursion
- Currying
- Higher-order functions.
- List Processing functions such as map, filter, foldl, sortBy, etc.
- The Maybe monad

The following webpage has lots of useful information if you need to revise Haskell. There are also links to descriptions of the Haskell libraries that you might find useful:

Haskell resources: http://www.haskell.org/haskellwiki/Haskell

In particular, to read this assignment sheet, you should recall the Haskell type-definition syntax. For example, a function **foo** which takes an argument of type **Int** and an argument of type **String** , and returns an argument of type **Int**, has the type declaration **foo :: Int** → **String** → **Int**. Most of the functions you will write have their corresponding type-definitions already given.

2 Important Information

There are the following sources of help:

- Attend lab sessions
- Read Russell and Norvig (R&N) chapter 6.
- Email Petros Papapanagiotou (pe.p@ed.ac.uk)
- Read any emails sent to the Inf2D mailing list regarding the assignment.

3 The "Cluedo" CSP (10%)

Cluedo is a popular mystery board game created by Anthony E. Pratt. In this assignment, inspired by the classical "Zebra" CSP by Lewis Carrol, we model Cluedo and solve it using CSP algorithms.

The problem's description is as follows:

> The detectives investigating Dr.Black's mysterious murder in Tudor Mansion are putting together the evidence they have collected to solve the mystery. The primary suspects are neighbours living next to each other in five coloured houses (green, red, blue, yellow, and white). Their names are Miss Scarlett, Colonel Mustard, Reverend Green, Mrs. Peacock, and Professor Plum. There are also five possible murder weapons, each owned or found in the house of one of the suspects: a candlestick, a dagger, a lead pipe, a revolver, and a rope. Each of the suspects was in one of five different rooms of the Tudor Mansion and the time of the murder: the kitchen, the lounge, the billiard room, the library, dining room. Finally, of the five suspects, one is an accomplice, one is innocent, one was sleeping, one was studying, and one is the actual murderer.
>
> The problem is to find **who murdered Dr.Black, in which room, and with which weapon**.
>
> The detectives have gathered the following fourteen clues:
>
> 1. Miss Scarlett lives in the green house.
> 2. Colonel Mustard is innocent.
> 3. The owner of the red house was in the lounge.
> 4. Mrs. Peacock was in the kitchen.
> 5. The red house is on the right of the blue one.
> 6. The dagger was with whoever was sleeping.
> 7. The lead pipe was found in the yellow house.
> 8. The owner of the middle house was in the billiard room.
> 9. Professor Plum lives in the first house on the left.
> 10. The accomplice lives in a house next to that where the Rope was found.
> 11. The lead pipe was found in a house next to the person who was studying.
> 12. The revolver was found in the Library.
> 13. Reverend Green was holding the Candlestick.
> 14. Professor Plum's house is next to the white one.

Your task is to implement this problem as a CSP within the given framework.

A CSP is defined by a set of variables and a set of constraints. Each variable has a non-empty set of possible values (domain). Each constraint involves some subset of the variables –its *scope*– in a *relation* that specifies the allowable combinations of values for that subset.

Read the CSP framework reference manual given in the Introduction for details on the datatypes you must use for your implementation.

You can go through the four implemented examples of CSPs, namely the Map of Australia, the Map of Scotland, the 3x3 Magic Square, and the Sudoku[1]. This will help you understand the proper usage of the framework's datatypes so that you can implement your own CSP accordingly.

The following four tasks are required:

i. Your CSP must contain a set of variables and their respective domains that accurately represent the problem. Combine the variables and their domains to create a Domains list.

   (Hint: The five houses of the problem can be numbered 1-5. These numbers will be used as the possible values for your variables.)

ii. Moreover, you must implement the appropriate constraints that will properly reflect the restrictions given in the problem. For each constraint you must reuse or implement a Relation.

   The functions you must implement are the following:

   - **has_value :: Int → Relation** This unary relation ensures a variable has a given value. Note that the scope of the corresponding constraint is assumed to always contain a single variable. This relation will be needed for the constraints corresponding to Clues 8 and 9 of the problem.
   - **vars_same :: Relation** This binary relation ensures two variables are the same. It is always assumed the scope of the corresponding constraint is a list of two variables. This relation will be needed for the constraints corresponding to multiple Clues of the problem (eg. Clues 1-4).

   For the rest of the problem restrictions you can use the Relation functions that are already given with the framework. Note that you should only use unary or binary constraints. The only exception is the already implemented "all_diff" constraint.

   (Hint: You can use the "abs_diff_one" constraint for Clues 10, 11, and 14, and the "diff_one" constraint for Clue 5.)

iii. Once you have defined the variables, domains and constraints of the problem, you must combine them to form the definition of the Cluedo CSP:

   cluedo_csp :: CSP

iv. Use the implemented BT algorithm to check that you have a correctly defined CSP. If your representation is accurate, the result must be a valid assignment of *all* the variables that will lead to the solution of the problem.

4 CSP Algorithms (45%)

Having implemented a CSP and tried the BT algorithm on it, you are now asked to implement more complex CSP algorithms.

4.1 Forward Checking (FC) algorithm (20%)

The FC algorithm is a simple constraint propagation algorithm. Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is a neighbour of X and deletes from Y's domain any value that is inconsistent with the value chosen for X. X and Y are considered neighbours when they are connected by (ie. within the scope of) the same constraint. FC indicates failure if it detects that one of the variable domains becomes empty. For a more detailed explanation of the algorithm, refer to R&N §6.3.2.

Read the CSP framework reference manual mentioned in the Introduction for details on the datatypes and functions involved in retrieving the neighbours of a variable and the function to delete a value from a domain.

[1]Note that you should avoid trying to solve the Sudoku problem with any of the algorithms that do not use the MRV heuristic as these may take several minutes or even hours to compute the solution.

Your task is to implement the FC algorithm for the given framework according to the description of R&N §6.3.2. In particular you will need to implement the following two functions:

i. **forwardcheck :: CSP → Assignment → Var → (CSP, Bool)** This is the forward checking constraint propagation function. Its arguments are: the current CSP, the current assignment/state of the problem and the variable X that was last assigned by the algorithm. It must find all variables Y that are neighbours with X and delete from their domains any values that are inconsistent with the given assignment. It returns the CSP with an updated Domains list paired with a boolean value that indicates whether the assignment is consistent or not (ie. returns False if one of the variable domains becomes empty).

ii. **fc_recursion :: Assignment → CSP → (Maybe Assignment, Int)** The recursion for the FC algorithm. This is the forward checking counterpart of the "bt_recursion" function. Every time a value is assigned to a variable it must use "forwardcheck" and then apply a recursion over the returned CSP rather than the original one. It must return "Just" a consistent and complete assignment for the given CSP or "Nothing" if there is no solution, paired with the number of nodes that were visited during the search. Examine the implementation of "bt_recursion" to understand how the counting of the nodes is accomplished. The algorithm should also backtrack if "forwardcheck" indicates failure (returns False).

4.2 Minimum Remaining Values (MRV) (10%)

The MRV heuristic is a variable ordering heuristic. It affects the order in which the algorithm selects the variables for assignment. The aim is to have the algorithm select the variables with less available values (therefore less branching) earlier on in the execution.

Your task is to implement the MRV heuristic and use it in the FC algorithm in the given framework, resulting in the FC+MRV algorithm. In particular you will need to implement the following functions:

i. **mrv_compare :: CSP → Var → Var → Ordering** An ordering function that given a CSP and two variables X and Y as arguments, returns the ordering of the sizes of the domains of the variables. The Ordering type allows three possible values: GT (greater), EQ (equal), and LT (less than).

   For example, *mrv_compare cluedo_csp "X" "Y"* should return "GT" if the domain of "*X*" is strictly greater than that of "*Y*".

ii. **get_mrv_variable :: CSP → Assignment → Var** This is the implementation of the MRV heuristic for the FC algorithm. Given a CSP and an assignment, this function returns the first unassigned variable that has the *smallest domain* in the CSP. You are advised to use the list function **sortBy :: (a → a → Ordering) → [a] → [a]** combined with "mrv_compare".

iii. **fc_mrv_recursion :: Assignment → CSP → (Maybe Assignment, Int)** This is the same as "fc_recursion" from Section 4.1 *but* with the addition of the MRV heuristic implemented as "get_mrv_variable".

4.3 Least Constraining Value (LCV) (15%)

The LCV heuristic is a value ordering heuristic. It affects the order in which the algorithm selects the values to assign to the selected variable. The aim is to have the algorithm select the value that allows the most choices for the neighbours of the variable.

Your task is to implement the LCV heuristic and use it in the FC anf FC+MRV algorithms (resulting in the FC+LCV and FC+MRV+LCV algorithms respectively) in the given framework. In particular you will need to implement the following functions:

i. **num_choices :: CSP → Assignment → Var → Int** Given a CSP, an assignment, and a variable X, this function returns the number of values (choices) that are allowed for all the neighbours of X. Note that it is assumed that X is an assigned variable in the given assignment.

ii. **lcv_sort :: CSP → Assignment → Var → [Int]** This function returns the domain of a given variable, sorted according to the LCV heuristic. You should use the "num_choices" function to help you implement the heuristic.

iii. **fc_lcv_recursion :: Assignment → CSP → (Maybe Assignment, Int)** This is the same as "fc_recursion" from Section 4.1 *but* with the addition of the LCV heuristic implemented as "lcv_sort".

iv. **fc_mrv_lcv_recursion :: Assignment → CSP → (Maybe Assignment, Int)** This is the same as "fc_mrv_recursion" from Section 4.2 *but* with the addition of the LCV heuristic implemented as "lcv_sort".

5 Evaluation and Discussion (25%)

For this task you are asked to run a small evaluation on your implemented algorithms and briefly discuss the results.

The evaluation will be based on **the number of visited search nodes** by each of the algorithms when used to solve each of the following CSPs: Map of Australia, Map of Scotland, 3x3 Magic Square, Cluedo.

We consider that a new search node has been visited when a new value has been assigned to a variable, followed by the respective recursive call of the algorithm. You should check the implementation of the "bt_recursion" function to understand how this is accomplished.

i. According to the specification given in 4.1 and 4.2, your implemented algorithms must return the solution to the problem paired with the number of visited search nodes. You must record this number for each algorithm on each of the four CSPs.

ii. You must complete the table provided in the given Haskell file with the number of visited nodes by the algorithms BT, FC, FC+MRV, FC+LCV, and FC+MRV+LCV for each of the CSPs Map of Australia, Map of Scotland, 3x3 Magic Square, and Cluedo.

iii. You must also write a brief report on your implementation and evaluation. Give a brief explanation of the values in the table and the differences observed between the algorithms. Discuss whether these were expected or not, and why. In particular, compare and explain your results of 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

iv. Is there a simple yet effective optimisation that you can apply to your implemented FC algorithm to improve its performance (esp. when considering much bigger and harder CSPs)? Give a suggestions and your arguments as to how your optimisation would improve the algorithm.

Note: The report should be **no** longer than **one** A4 page of plain text written within your version of the Haskell file, in the section indicated in the document.

6 Arc Consistency Algorithm AC-3 (20%)

Arc consistency is a fast method for constraint propagation based on checking the consistency of the arcs in the constraint graph. 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 left for Y then we remove x' from X's possible values (domain).

The AC-3 algorithm is an implementation of arc consistency. You can find the pseudocode of the algorithm in Figure 6.3 of R&N §6.2. Your task is to implement the AC-3 algorithm and evaluate it.

In particular, you will need to implement the following functions:

i. **exists_consistent_value :: CSP → Var → Int → Var → [Int] → Bool** The arguments for this function are a CSP, a variable X, a value x that is being checked for variable X, a variable Y and the list of possible values (current domain) for Y. X and Y are assumed to be neighbours. The function checks if there exists a value y for Y such that if x is assigned to X and y to Y, then their common constraints (ie. the constraints that involve both X and Y) are satisfied. It returns False if there is no such value.

ii. **revise :: CSP → Var → Var → (CSP, Bool)** This corresponds to the "REVISE" function of the algorithm in R&N Fig. 6.3. It uses "exists_consistent_value" to check for arc consistency between the two given variables. It returns a pair consisting of the CSP with a new domain (without the inconsistent arcs) and a Bool value that is True only if the domain of the first variable was revised.

iii. **ac3_check :: CSP → [(Var,Var)] → (CSP, Bool)** This corresponds to the AC-3 algorithm in the pseudocode in R&N Figure 6.3. It returns an updated, arc-consistent CSP and a boolean value which is False if an inconsistency is found or True otherwise.

iv. **ac3_recursion :: Assignment → CSP → (Maybe Assignment, Int)** This is the AC-3 checking counterpart of the "bt_recursion" function. The standard backtracking algorithm should be used, with the addition of the call to the "ac3_check" function to check for arc consistency and update the CSP domains.

v. **ac3_mrv_lcv_recursion :: Assignment → CSP → (Maybe Assignment, Int)** This function should be the *same* as the "ac3_recursion" function *but* with the addition of the MRV heuristic for the selection of variables and the LCV heuristic for the selection of values.

Note that the AC-3 algorithm is designed to work for **binary** CSPs whereas we allow n-ary constraints such as "all_diff". This limitation does not affect this implementation. All the constraints in this assignment are designed to ignore unassigned variables. Therefore, if the assignment being checked contains only two assigned variables, the corresponding constraints are treated as binary.

For example, checking the *all_diff ["x", "y", "z"]* constraint against the assignment $[x = 1, y = 2]$ will return True, even though z is not assigned.

You should keep this in mind when implementing your AC-3 algorithm, so that you treat all constraints as binary.

You should add evaluation results for, AC3 and AC3+MRV+LCV on the CSPs Map of Australia, Map of Scotland, and Cluedo to the table from Section 5 and *briefly* discuss how they compare with the results from their forwardchecking counterparts FC and FC+MRV+LCV, in a maximum of half a page in your report.

Since the 3x3 Magic Square problem contains n-ary constraints that cannot be reduced to binary ones, it should not be included in the evaluation. You may, however, run the AC-3 algorithms on the guven Sudoku problem for your own amusement.

7 Notes

Please note the following:

- To ensure maximum credit, make sure you have commented your code and that it can be properly loaded on ghci before submitting.
- All the algorithms are expected to terminate within **5 minutes or less** for all the CSPs in this assignment **excluding the Sudoku problem**. You may lose marks if your algorithms do not terminate in under 5 minutes of runtime.
- You are free to implement any number of custom functions to help you in your implementation. However, you **must** comment these functions explaining their functionality and why you needed them.
- Do **not** change the names or type definitions of the functions you are asked to implement. Moreover, you may **not** alter any part of the code given as part of the CSP framework. You may only edit and submit your version of the *Inf2d.hs* file.
- You are strongly encouraged to create your own unit tests to test your individual functions (these do not need to be included in your submitted file).

8 Submission

You should submit your version of the file *Inf2d.hs*, that contains your implemented functions, evaluation table and report, using the following command:

```
submit inf2d 1 Inf2d.hs
```

Your file must be submitted by **2pm** on the **27th of February 2012**.