Computer Science 384
St. George Campus

February 7, 2016
University of Toronto

Homework Assignment #2: Constraint Satisfaction
Due: February 27, 2016 by 11:59 PM

Silent Policy: *A silent policy will take effect 24 hours before this assignment is due, i.e. no question about this assignment will be answered, whether it is asked on the discussion board, via email or in person.*

Late Policy: 10% per day after the use of 3 grace days.

Total Marks: This part of the assignment represents 10% of the course grade.

Handing in this Assignment

What to hand in on paper: Nothing.

What to hand in electronically: You must submit your assignment electronically. Download the assignment files from the A2 web page. Modify `propagators.py` and `tenner_csp.py` appropriately so that they solve the problems specified in this document. **Submit your modified files** `propagators.py` **and** `tenner_csp.py`**.**

How to submit: If you submit before you have used all of your grace days, you will submit your assignment using MarkUs. Your login to MarkUs is your teach.cs username and password. It is your responsibility to include all necessary files in your submission. You can submit a new version of any file at any time, though the lateness penalty applies if you submit after the deadline. For the purposes of determining the lateness penalty, the submission time is considered to be the time of your latest submission. More detailed instructions for using Markus are available at:

`http://www.cdf.toronto.edu/~csc384h/winter/markus.html`.

Warning: marks will be deducted for incorrect submissions.

We will test your code electronically. You will be supplied with a testing script that will run a **subset** of the tests. If your code fails all of the tests performed by the script (using Python version 3.5.2), you will receive zero marks. It's up to you to figure out further test cases to further test your code – that's part of the assignment!

When your code is submitted, we will run a more extensive set of tests which will include the tests run in the provided testing script and a number of other tests. You have to pass all of these more elaborate tests to obtain full marks on the assignment.

Your code will not be evaluated for partial correctness, it either works or it doesn't. It is your responsibility to hand in something that passes at least some of the tests in the provided testing script.

- *Make certain that your code runs on teach.cs using python3 (version 3.5.2) using only standard imports.* This version is installed as "python3" on teach.cs. Your code will be tested using this version and you will receive zero marks if it does not run using this version.
- *Do not add any non-standard imports from within the python file you submit (the imports that are already in the template files must remain).* Once again, non-standard imports will cause your code to fail the testing and you will receive zero marks.
- *Do not change the supplied starter code.* Your code will be tested using the original starter code, and if it relies on changes you made to the starter code, you will receive zero marks.

Evaluation Details: The details of the evaluation will be released as a separate document in approximately one week.

Clarification Page: Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Assignment 2 Clarification page, linked from the CSC384 A2 web page, also found at: `http://www.cdf.toronto.edu/~csc384h/winter/Assignments/A2/a2_faq.html`. You are responsible for monitoring the A2 Clarification page.

Questions: Questions about the assignment should be asked on Piazza:
`https://piazza.com/utoronto.ca/winter2017/csc384`.

If you have a question of a personal nature, please email the A1 TA, Yi Li, at liyi at cs dot toronto dot edu or the instructor, Sheila McIlraith, at csc384prof at cs dot toronto dot edu placing 384 and A2 in the subject line of your message.

Introduction

There are two parts to this assignment

1. the implementation of two constraint propagators – a Forward Checking constraint propagator, and a Generalized Arc Consistence (GAC) constraint propagator,

2. the encoding of two different CSP models to solve the logic puzzle, "Tenner Grid", as described below. In one model you will use binary not-equal constraints for row constraints, while in the other model you will use n-ary all-different constraints for them.

What is supplied:

- **cspbase.py** – class definitions for the python objects Constraint, Variable, and BT.
- **propagators.py** - starter code for the implementation of your two propagators. You will modify this file with the addition of two new procedures `prop_FC` and `prop_GAC`, to realize Forward Checking and GAC, respectively.
- **tenner_csp.py** – starter code for the two Tenner Grid CSP models.
- **Sample test cases** for testing your code. Test cases for Q1 will be released on the A2 web page on Friday, February 10; test cases for Q2 will be released on the A2 web page on Monday, February 13.

Tenner Grid Formal Description

The Tenner Grid puzzle[1] has the following formal description:

- Tenner Grid (also known as "From 1 to 10", "Zehnergitter", "Grid Ten") consists of a rectangular grid of width ten cells, i.e., the *grid* with dimensions n rows by 10 columns, and a special $(n+1)$-th row. The task is to fill in the first n rows so that every row contains the digits 0 through 9. In columns the numbers may be repeated.

[1] `http://www.cross-plus-a.com/html/cros7tng.htm`

6		1	5	7				3	
	9	7			2	1			
					0				1
	9		0	7		3	5	4	
6			5		0				
21	26	21	21	29	10	28	26	21	22

6	4	1	5	7	0	8	9	3	2
5	9	7	3	6	2	1	4	0	8
3	2	4	8	5	0	9	7	6	1
1	9	6	0	7	8	3	5	4	2
6	2	3	5	4	0	7	1	8	9
21	26	21	21	29	10	28	26	21	22

Figure 1: An example of a 5×10 grid with its start state (left) and solution (right).

- The $(n+1)$-th row contains numbers which give the sum of the numbers in their respective columns. The numbers in the $(n+1)$-th row are always given in the start state.
- The digits in adjacent cells (even diagonally adjacent cells) must be different. For example, $cell(0,0)$ is adjacent to $cell(0,1)$, $cell(1,0)$ and $cell(1,1)$.
- The start state of the puzzle has some spaces already filled in.
- A puzzle is *solved* if all empty cells are filled in with an integer from 0 to 9 and all above constraints are satisfied.
- An example of a 5×10 grid is shown in Figure 1. Note that your solution will be tested on $n \times 10$ grids where n can be from 3 to 8.

Question 1: Propagators **(worth 50/100 marks)**

You will implement python functions to realize two constraint propagators – a Forward Checking constraint propagator and a Generalized Arc Consistence (GAC) constraint propagator. These propagators are briefly described below. The files `cspbase.py` and `propagators.py` provide the **complete input/output specification** of the two functions you are to implement.

The correct implementation of each function is worth 25/100 marks.

Brief implementation description: A `Propagator Function` takes as input a CSP object `csp` and (optionally) a variable `newVar` representing a newly instantiated variable. The CSP object is used to access the variables and constraints of the problem (via methods found in `cspbase.py`). A propagator function returns a tuple of `(bool,list)` where `bool` is False if and only if a dead-end is found, and `list` is a list of `(Variable, value)` tuples that have been pruned by the propagator. You must implement:

`prop_FC` A propagator function that propagates according to the Forward Checking (FC) algorithm that check constraints that have *exactly one uninstantiated variable in their scope*, and prune appropriately.

If `newVar` is `None`, forward check all constraints. Else, if `newVar=var` only check constraints containing `newVar`.

`prop_GAC` A propagator function that propagates according to the Generalized Arc Consistency (GAC) algorithm, as covered in lecture. If `newVar` is `None`, run GAC on all constraints. Else, if `newVar=var` only check constraints containing `newVar`.

Question 2: Tenner Grid Models **(worth 50/100 marks)**

You will implement two different CSP encodings to solve the logic puzzle, Tenner Grid. In one model you will use only binary not-equal constraints apart from the arithmetic sum constraints, while in the other model you will use both binary not-equal and *n*-ary all-different constraints. These CSP models are briefly described below. The file `tenner_csp.py` provides the **complete input/output specification** for the two CSP encodings you are to implement.

The correct implementation of each encoding is worth 25/100 marks.

Brief implementation description: A `Tenner Grid Model` takes as input a valid Tenner Grid board, which is a tuple: (`n_grid`, `last_row`), and returns a `CSP object`, consisting of a variable corresponding to each cell of the grid. The `n_grid` variable consists of a list of *n* length-10 lists, representing the first *n* rows of the Tenner Grid. The `last_row` variable is a list of 10 integers, representing the $(n+1)$-th row of the grid. The variable domain of that cell is $\{0,...,9\}$ if the grid is unfilled at that position (denoted by -1 in the input), and equal to *i* if the grid has a fixed number *i* at that cell. All appropriate constraints will be added to the board as well. You must implement:

`tenner_csp_model_1` A model built using only binary not-equal constraints for the row and contiguous cells constraints, and *n*-ary sum constraints.

`tenner_csp_model_2` A model built using a combination of *n*-ary all-different constraints for the row and sum constraints, as well as binary not-equal constraints for the contiguous cells constraints.

Caveat: The Tenner Grid CSP models you will construct can be space expensive, especially for constraints over many variables, (e.g., for sum constraints and those contained in the second Tenner Grid CSP model). *HINT: Also be mindful of the **time** complexity of your methods for identifying satisfying tuples, especially when coding the second Tenner Grid CSP model.*

HAVE FUN and GOOD LUCK!