Computer Science 384
St. George Campus

June 24, 2019
University of Toronto

Homework Assignment #3: Constraint Satisfaction
Due: July 19, 2019 by 10:00 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 11% 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 A3 web page. Modify `propagators.py`, `heuristics.py`, and `kenken_csp.py` appropriately so that they solve the problems specified in this document. **Submit your modified** `propagators.py`**,** `heuristics.py`**,** `kenken_csp.py` **files.** You must also fill in and submit the acknowledgment form, `acknowledgment_form.pdf`.

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.teach.cs.toronto.edu/~csc384h/summer/markus.html`.

Extra Information

Clarification Page: Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Assignment 3 Clarification page.[1] You are responsible for monitoring the A3 Clarification page.

Questions: Questions about the assignment should be asked on Piazza.[2] If you have a question of a personal nature, please email the A3 TA, Chris Karavasilis, at ckar at cs dot toronto dot edu or the instructor placing 384 and A3 in the subject line of your message.

---

[1] `http://www.teach.cs.toronto.edu/~csc384h/summer/Assignments/A3/a3_faq.html`.

[2] `https://piazza.com/utoronto.ca/summer2019/csc384`.

Evaluation Details

We will test your code electronically.

When your code is submitted, **we will run an extensive set of tests**. You have to pass all of these tests to obtain full marks on the assignment.

We will set a timeout of 60 seconds per board during marking. This 60 second time limit includes the time to create a model and find a solution. Solutions that time-out are considered incorrect. Ensure that your implementations perform well under this limit, and be aware that your computer may be quicker than teach.cs (where we will be testing). We will not conduct end to end tests on boards with constraints whose domain is larger than six.

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 3.7) 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.

Introduction

There are two parts to this assignment.

1. **Propagators**. You will implement two constraint propagators—a Forward Checking constraint propagator, and a Generalized Arc Consistence (GAC) constraint propagator—and two heuristics—Minimum-Remaining-Value (MRV), and Least-Constraining-Value (LCV).

2. **Models**. You will implement three different CSP models: two grid-only KenKen models, and one full KenKen puzzle model (adding *cage* constraints to grid).

What is supplied

- **cspbase.py**. Class definitions for the python objects Constraint, Variable, and BT.
- **csp_sample_run.py**. This file contains a sample implementation of two CSP problems.
- **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`.
- **heuristics.py**. Starter code for the implementation of the variable ordering heuristic, MRV; and the value heuristic, LCV. You will modify this file with the addition of the new procedures `ord_mrv`, and `val_lcv`.
- **kenken_csp.py**. Starter code for the CSP models. You will modify three procedures in this file: `binary_ne_grid`, `nary_ad_grid`, and `kenken_csp_model`.

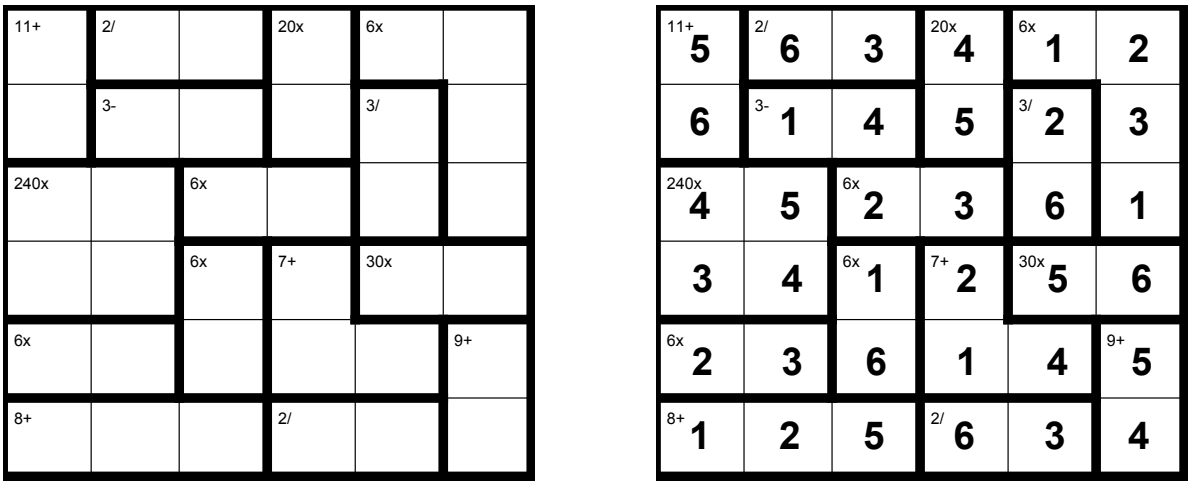

Figure 1: An example of a 6×6 KenKen grid with its start state (left) and solution (right).

KenKen Formal Description

The KenKen puzzle[3] has the following formal description:

- KenKen consists of an $n \times n$ grid where each cell of the grid can be assigned a number 1 to n. No digit appears more than once in any row or column. Grids range in size from 3×3 to 9×9.
- KenKen grids are divided into heavily outlined groups of cells called *cages*. These *cages* come with a *target* and an *operation*. The numbers in the cells of each *cage* must produce the *target* value when combined using the *operation*.
- For any given *cage*, the *operation* is one of addition, subtraction, multiplication or division. Values in a *cage* can be combined in any order: the first number in a *cage* may be used to divide the second, for example, or vice versa. Note that the four operators are "left associative" e.g., 16/4/4 is interpreted as (16/4)/4 = 1 rather than 16/(4/4) = 16.
- A puzzle is *solved* if all empty cells are filled in with an integer from 1 to n and all above constraints are satisfied.
- An example of a 6×6 grid is shown in Figure 1. Note that your solution will be tested on $n \times n$ grids where n can be from 3 to 9.

[3] `http://thinkmath.edc.org/resource/introducing-kenken-puzzles`

Question 1: Propagators **(worth 60/100 marks)**

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

Brief implementation description: The propagator functions take as input a CSP object `csp` and (optionally) a Variable `newVar` representing a newly instantiated Variable, and return 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. `ord_mrv` takes a CSP object `csp` as input, and return a Variable object `var`. `val_lcv` takes a CSP object `csp` and a Variable object `var` as input, and returns a list of values in the domain of that variable. In all cases, the CSP object is used to access variables and constraints of the problem, via methods found in `cspbase.py`.

You must implement:

`prop_FC` (worth 20/100 marks)

A propagator function that propagates according to the 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. Otherwise only check constraints containing `newVar`.

`prop_GAC` (worth 20/100 marks)

A propagator function that propagates according to the GAC algorithm, as covered in lecture. If `newVar` is `None`, run GAC on all constraints. Otherwise, only check constraints containing `newVar`.

`ord_mrv` (worth 10/100 marks)

A variable ordering heuristic that chooses the next variable to be assigned according to the Minimum-Remaining-Value (MRV) heuristic. `ord_mrv` returns the variable with the most constrained current domain (i.e., the variable with the fewest legal values).

`val_lcv` (worth 10/100 marks)

A value heuristic that, given a variable, chooses the value to be assigned according to the Least-Constraining-Value (LCV) heuristic. `val_lcv` returns a list of values. The list is ordered by the value that rules out the fewest values in the remaining variables (i.e., the variable that gives the most flexibility later on) to the value that rules out the most.

Question 2: Models **(worth 40/100 marks)**

You will implement three different CSP models using three different constraint types. The three different constraint types are (1) binary not-equal; (2) *n*-ary all-different; and (3) KenKen *cage*. The three models are (a) binary grid-only KenKen; (b) *n*-ary grid-only KenKen; and (c) full KenKen. The CSP models you will build are described below. The file `kenken_csp.py` provides the **complete input/output specification**.

Brief implementation description: The three models take as input a valid KenKen grid, which is a list of lists, where the first list has a single element, *N*, which is the size of each dimension of the board, and each following list represents a *cage* in the grid. Cell names are encoded as integers in the range $11, ..., nn$ and each inner list contains the numbers of the cells that are included in the corresponding *cage*, followed by the *target* value for that *cage* and the *operation* (0=+, 1=-, 2=/, 3=*). If a list has two elements, the first element corresponds to a cell, and the second one—the *target*—is the value enforced on that cell.

For example, the model $((3),(11,12,13,6,0),(21,22,31,2,2),....)$ corresponds to a 3x3 board[4] where

1. cells 11, 12 and 13 must sum to 6, and
2. the result of dividing some permutation of cells 21, 22, and 31 must be 2. That is, (C21/C22)/C23 = 2 or (C21/C23)/C22 = 2, or (C22/C21)/C23 = 2, etc...

All models need to return a CSP object, and a list of lists of Variable objects representing the board. The returned list of lists is used to access the solution. The grid-only models do not need to encode the *cage* constraints.

You must implement:

`binary_ne_grid` (worth 10/100 marks)
: A model of a KenKen grid (without *cage* constraints) built using only binary not-equal constraints for both the row and column constraints.

`nary_ad_grid` (worth 10/100 marks)
: A model of a KenKen grid (without *cage* constraints) built using only *n*-ary all-different constraints for both the row and column constraints.

`kenken_csp_model` (worth 20/100 marks)
: A model built using your choice of (1) binary binary not-equal, or (2) *n*-ary all-different constraints for the grid, together with (3) KenKen *cage* constraints. That is, you will choose one of the previous two grid models and expand it to include KenKen *cage* constraints.

Notes: The CSP models you will construct can be space expensive, especially for constraints over many variables, (e.g., for *cage* constraints and those contained in the first `binary_ne_grid` grid CSP model). Also be mindful of the **time** complexity of your methods for identifying satisfying tuples, especially when coding the `kenken_csp_model`.

HAVE FUN and GOOD LUCK!

[4]Note that cell indexing starts from 1, e.g. 11 is the cell in the upper left corner.