Computer and Information Science 352 January 18, 2023
Main Campus Queen's University

Homework Assignment #1: Constraint Satisfaction
Due: February 5, 2023 by 11:59 PM

Late Policy: 25% per day after the use of 3 grace days.

Total Marks: This assignment represents 6 % of the course grade.

Handing in this Assignment
You must submit your assignment electronically. Download the assignment files from the A1 assignment on onQ. Modify `propagators.py`, `heuristics.py`, and `cagey_csp.py` appropriately so that they solve the problems specified in this document. **Submit your modified** `propagators.py`**,** `heuristics.py`**, and** `cagey_csp.py` **files, and only those files, in a single .zip**.

How to submit: Submissions should include just the three files above in a zip file, and should be uploaded once for your group. It is your responsibility to include all necessary files in your submission. For the purposes of determining the lateness penalty, the submission time is considered to be the time of your latest submission.

Extra Information

Clarification Page: Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Assignment 1 description in onQ. You are responsible for monitoring the announcements in onQ for any updates.

Questions: **We recommend you to check the forum to see if others had already asked similar questions.** Then, post your question on the forum so that everyone can be benefited. Questions about the assignment are also welcomed to be asked during office hours. If you have a question of a personal nature, please email the A1 TA, Mengyang Liu, at **mengyang.liu@queensu.ca** with **CISC352: A1** in the subject line of your message.

Last Updated: January 19, 2023

Corrections Made

- (Jan 18) None so far

Evaluation Details

We will test your code electronically. **You will be supplied with a testing script that will run the same suite of tests**. If your code fails all of the tests performed by the script (using Python version 3.5.2 or higher), you will receive zero marks.

When your code is submitted, **we will run the same set of tests given to you**. If you passed all the given tests, you are on the right way 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 our cluster (where we will be testing). Do not panic on that as the boards for evaluation are not super hard to solve and a fair implementation can easily finish each board within 10 seconds.

Your code will **not be** evaluated for partial correctness on a particular test: 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 using Python3 (version 3.5.2 or above) using only standard imports.*. 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.
- All existing submissions will be tested by the official deadline. Any instance of a non-functional submission will be reported to its respective team, giving them the 3-day grace period to remedy the situation.

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 Consistency (GAC) constraint propagator—and two heuristics—Minimum-Remaining-Value (MRV) and Degree (DH).

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

What is supplied

- **API for cspbase.pdf**. A more or less detailed API of "cspbase.py" that introduces the objects and methods you will use in your implementation. **We strongly recommend to take a look at it.**
- **cspbase.py**. Class definitions for the python objects Constraint, Variable, and CSP.

- **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 heuristics, MRV and DH. You will modify this file with the addition of the new procedures `ord_mrv` and `ord_dh`.
- **cagey_csp.py**. Starter code for the CSP models. You will modify three procedures in this file: `binary_ne_grid`, `nary_ad_grid`, and `cagey_csp_model`.
- **autograder_stu.py**. An auto-grader that contains the same test cases we will use to grade your work. *Run it with "python3 autograder_stu.py"*. DO NOT THINK you can reverse-engineer the assignment from this file. If you do so, you will eventually see why it is impossible.
- **answer_set.py**. A script that "autograder_stu.py" depends on, containing solutions for all testing cases. It is a very large file and opening it can cause your computer to lag. We do not recommend you to open it unless you feel like you really need to.
- **csp_sample_run.py**. Example CSP problems to demonstrate usage of the API.

Extended Cagey Formal Description

The Cagey puzzle[1] has the following formal description:

- Cagey 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.
- Cagey 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*. In this extended variant of the Cagey Puzzle, the operation is not given, and must be determined.
- 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.
- For this assignment we are focusing on an extended version of the Cagey Puzzle where the operations in a given cell are not explicitly given, and must be determined as part of the puzzle.
- An example of a 6×6 grid following the extended ruleset 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.

[1] Some might recognize it as `http://thinkmath.edc.org/resource/introducing-kenken-puzzles`

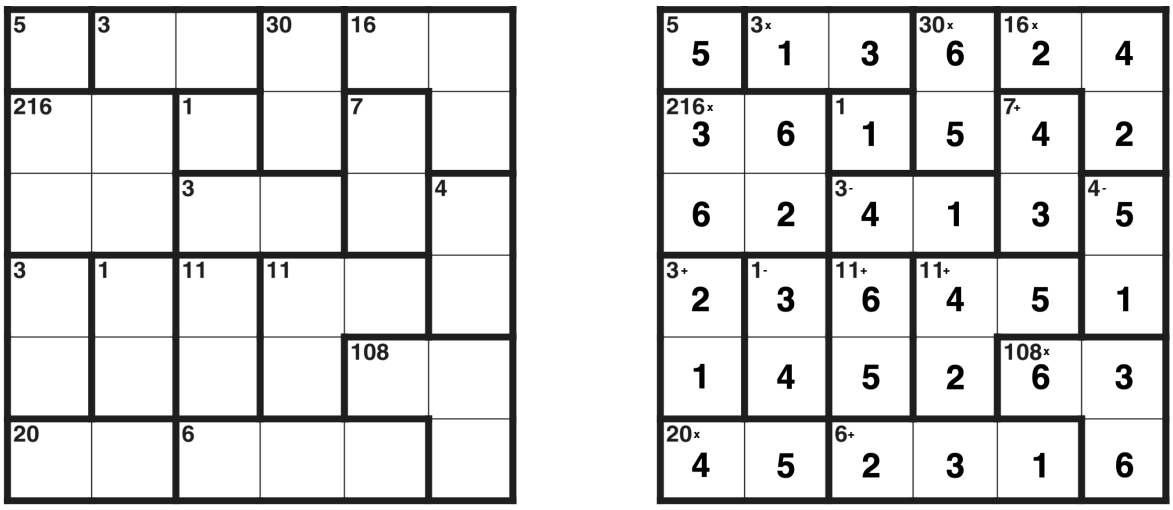

Figure 1: An example of a 6×6 Cagey grid with its start state (left) and solution (right).

Question 1: Propagators and Heuristics **(worth 3.5/6 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` and `ord_dh` take a CSP object `csp` as input, and return a Variable object `var`. In all cases, the CSP object is used to access variables and constraints of the problem, via methods found in `cspbase.py`.

Regarding heuristics: Information about heuristics for this assignments are *not on slides or lectures*, as they are quite straightforward. In this assignments, heuristic functions *take in a CSP and return one unassigned variable*, based on specifications. Heuristics chooses the next Variable to assign for the propagator, so that the solving can be faster in most cases.

You must implement:

`prop_FC` (worth 1/6 marks)

A propagator function that propagates according to the FC algorithm that check constraints that have *exactly one variable in their scope that has not assigned with a value*, and prune appropriately. **If `newVar` is `None`, forward check all constraints. Otherwise only check constraints containing `newVar`.**

`prop_GAC` (worth 1.5/6 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 0.5/6 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 remaining).

`ord_dh` (worth 0.5/6 marks)

A variable ordering heuristic that chooses the next variable to be assigned according to the Degree heuristic (DH). `ord_dh` returns the variable that is involved in the largest number of constraints, which have other unassigned variables.

Question 2: Models **(worth 2/6 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) *cage*. The three models are (a) binary grid-only Cagey; (b) *n*-ary grid-only Cagey; and (c) full Cagey. The CSP models you will build are described below. The file `cagey_csp.py` provides the **complete input/output specification**.

Brief implementation description: The three models take as input a valid Cagey grid, which is a formatted list of elements, where the first element N, gives the size of each dimension of the board. The second element is a list defining all *cages*. Each element in the *cages* list is a tuple representation of the grid *cages*. In this tuple, the first element is an integer which defines the *expected number* (a natural number) of the *cage*.
The second element in this tuple is a list containing indicies for all grid-cells enclosed by the *cage*. These indices are formatted as coordinate tuples (m,n), where m represents the row $(1...N)$, and n represents the column $(1...N)$.
The third and final element of this tuple is an operation identifier character. For a given cage, the enclosed values must be able to be arranged in some order such that the corresponding mathematical operation produces the *expected number* when applied.

Possible characters are:

- '+' - Addition
- '-' - Subtraction
- '*' - Multiplication
- '/' - Division, no fraction(such as 5/2) is allowed
- '?' - Unknown

The *unknown* operator can refer to any of the other operations, which can produce the expected numbers given the right grid-valuations. Your solution needs to solve for the operation that *unknown* is using (i.e. operators are also Variables). For cages that have only 1 cell, selecting any operator is fine, but leaving it as a "None" may cause issues for auto-graders.

For example, the model $(3,[(6,[(1,1),(1,2),(1,3)],'+'),(2,[(2,1),(2,2),(2,3)],'/'),...])$ corresponds to a 3x3 board[2] where

1. cells (1,1), (1,2) and (1,3) must sum to 6, and
2. the result of dividing some permutation of cells (2,1), (2,2), and (3,1) 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 Variable objects representing the board (i.e., the cell values and operand for a cage). The returned list is used to access the solution. The grid-only models do not need to encode the *cage* constraints.

[2]Note that cell indexing starts from 1, e.g. **the cell in the upper left corner is 1,1 not 0,0.**

You must implement:

`binary_ne_grid` (worth 0.5/6 marks)
: A model of a Cagey grid (without *cage* constraints) built using only binary not-equal constraints for both the row and column constraints.

`nary_ad_grid` (worth 0.5/6 marks)
: A model of a Cagey grid (without *cage* constraints) built using only *n*-ary all-different constraints for both the row and column constraints.

`cagey_csp_model` (worth 1/6 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) *cage* constraints. That is, you will choose one of the previous two grid models and expand it to include *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 `cagey_csp_model`.

Code Quality **(worth 0.5/6 marks)**

The overall code quality (readability, proper use of files, etc) will be worth 0.5 mark. Improper use of external libraries will result in far more than just 0.5 grade being deducted – you shouldn't need anything beyond standard Python libraries other than the following, and you can ask on the Forum if you'd like to use anything else.

Accepted libraries/methods so far:

- `itertools.products`
- `itertools.permutations`

Do not use the internal methods in `cspbase.py`, they are for internal calls, not for your implementation. Also, please avoid using "match-case"(switch-case in Python) and use multiple "if-else" instead if needed. Your assignment is randomly run by one of the graders and we cannot guarantee the Python running has version above 3.10 (the version first supports "match-case"). We are sorry for that but it should not bring too much extra work.

HAVE FUN and GOOD LUCK!

API for `cspbase.py`

Table of Contents:

`Variable` class:

```
cspbase.Variable(
    name, domain=[]
)
```

- Arguments:
  - **name**: `string` type. The name of this variable.
    - Ex: Variable cell (1,1) should have name `Cell(1,1)`.
    - Ex: A cage contains cell (1,1) and cell (1,2), with operation '?' and expected number 12. The operand variable should have name:
      - `Cage_op(12:?:[Var-Cell(1,1), Var-Cell(1,2)])`
  - **domain**: `[int]` or `[string]` type. A list of `int` or `string` representing the PERMANENT domain of this variable.
    - PERMANENT domain: never changes during filtering.
    - CURRENT domain(will see later): can be changed by pruning/unpruning values during filtering.
    - Hint: `int` for cell variables and `string` for operand variables.

- Methods:
  - `domain()`:
    - Return the variable's PERMANENT domain.
  - `domain_size()`:
    - Return the size of the PERMANENT domain.

- `add_domain_values(values)`:
  - Add additional domain values to the PERMANENT domain.
  - `values`: a collection of `int` or `string` to add.
- `prune_value(value)`:
  - Remove `value` from CURRENT domain.
- `unprune_value(value)`:
  - Restore `value` to CURRENT domain.
- `restore_curdom()`:
  - Restore all values back into CURRENT domain.
  - Now CURRENT domain is the same as PERMANENT domain.
- `cur_domain()`:
  - Return list of values in CURRENT domain.
  - If assigned, only assigned value is viewed as being in current domain.
- `in_cur_domain(value)`:
  - Check if `value` is in CURRENT domain (without constructing lists).
  - If assigned, only the assigned value is viewed as being in current domain.
  - *Implemented by searching and indexing in domain, so this method is cheap*.
- `cur_domain_size()`:
  - Return the size of the variables in CURRENT domain, (without constructing lists).
  - *Implemented by traversing once in domain, so this method is cheap*.
- `is_assigned()`:
  - Return `True` if this variable is assigned with a value.
- `get_assigned_value()`:
  - Return the assigned value to this variable.
  - If this variable is not assigned, `None` is returned.

`Constraint` class:

```
cspbase.Constraint(
    name, scope
)
```

- Arguments:
  - **name**: `string` type. The name of this constraint.
    - Can be any descriptive and unique name among other constraints.
  - **scope**: `[Variable]` type. The list of all variables involved in this constraint.

- Methods:
  - `add_satisfying_tuples(tuples)`:
    - Specify the constraint by adding its complete list of satisfying tuples.

        - `tuples`: a list of tuples of satisfying values.
    - `get_scope()`:
        - Return a list of variables that are involved in this constraint.
    - `check_tuple(t)`:
        - Return `True` if the given tuple is a satisfying tuple for this constraint. `False` otherwise.
    - `get_n_unasgn()`:
        - Return the number of unassigned variables in the constraint's scope.
    - `get_unasgn_vars()`:
        - Return list of unassigned variables in constraint's scope.
        - Caution: this method is computationally expensive. See if `get_n_unasgn()` is enough to do the job.
    - `check_var_val(var, val)`:
        - Return `True` if:
            - Suppose we want to assign variable `var` with value `val`, there are still satisfying tuples in this constraint (in the CURRENT domain of all variables in the scope).
        - Return `False` otherwise.

`CSP` class:

```
cspbase.CSP(
    name, vars=[]
)
```

- Arguments:
    - **name**: `string` type. The name of this CSP object.
        - Can be any descriptive and unique name among other CSP objects.
    - **vars**: `[Variables]` type. The list of all variables in this CSP.

- Methods:
    - `add_var(var)`:
        - Add variable `var` to CSP.
    - `add_constraint(con)`:
        - Add constraint `con` to CSP.
        - All variables in the constraint's scope must already have been added to the CSP.
    - `get_all_vars()`:
        - Return a list of all variables in the CSP
    - `get_all_unasgn_vars()`:
        - Return a list of unassigned variables in the CSP
    - `get_all_cons()`:
        - Return a list of all constraints in the CSP.

    - `get_cons_with_var(var)`:
        - Return a list of constraints that include variable `var` in their scope.
    - `get_all_nary_cons(n)`:
        - Return a list of all constraints that have exactly `n` variables in its scope.
    - `print_all()`:
        - Debugging method. Prints all the variables and constraints in the CSP.
    - `print_soln()`:
        - Debugging method. Prints all the variables and their assigned values in the CSP.

Misc:

- If you have any concerns about this file, feel free to post on the forum: (https://discourse.caslab.queensu.ca/c/cisc-352-w23/).