Computer Science 384 June 17, 2025
St. George Campus University of Toronto

Homework Assignment #2: Constraint Satisfaction
Due: July 9, 2025 by 11:00 PM

Handing in this Assignment

What to hand in on paper: Nothing.

What to hand in electronically: You must submit your assignment electronically. Download `cspbase.py`, `propagators.py`, `puzzle_csp.py` and `sample_csp_run.py` from Quercus. These are in the zip file that is called `csp.zip`. Modify `propagators.py` and `puzzle_csp.py` so that they solve the problems specified in this document. Then, submit your modified `propagators.py` using MarkUs. Your login to MarkUs is your teach.cs username and password. 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.

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 3.10), you will receive a failing grade on 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.

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 3.10) 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 mark 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.

Clarifications: Important corrections (hopefully few or none) and clarifications to the assignment will be announced on Quercus and/or Piazza
You are responsible for monitoring Quercus announcements for any clarifications or corrections.

Help Sessions: There will be several help sessions for this assignment. Dates and times for these sessions will be posted on Quercus.

Questions: Questions about the assignment should be posed on Piazza.

Evaluation Details

We will test your code electronically. **You will be supplied with a testing script that will run a subset** of the tests (`tests.py`). If your code fails all of the tests performed by the script (using Python version 3.10), you will receive zero marks. **It's up to you to create 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**. You have to pass all of these more elaborate tests to obtain full marks on the assignment.

We will set a timeout of 120 seconds per board during marking. This 120 second time limit includes the time to create a CSP object 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 3.10) 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 General Arc Consistency constraint propagator
2. **CSP Encodings**. You will implement three different CSP encodings: two grid-only puzzle encodings, and one full FunPuzz encoding (adding *cage* constraints to grid).

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`.
- **puzzle_csp.py**. Starter code for the CSP encodings. You will modify three procedures in this file: `binary_ne_grid`, `nary_ad_grid`, and `caged_csp`.
- **csp_sample_run.py**. This file contains a sample implementation of two CSP problems.
- **tests.py**. Sample test cases. Run the tests with "python3 tests.py". **This file will be released some time after the assignment is posted.**

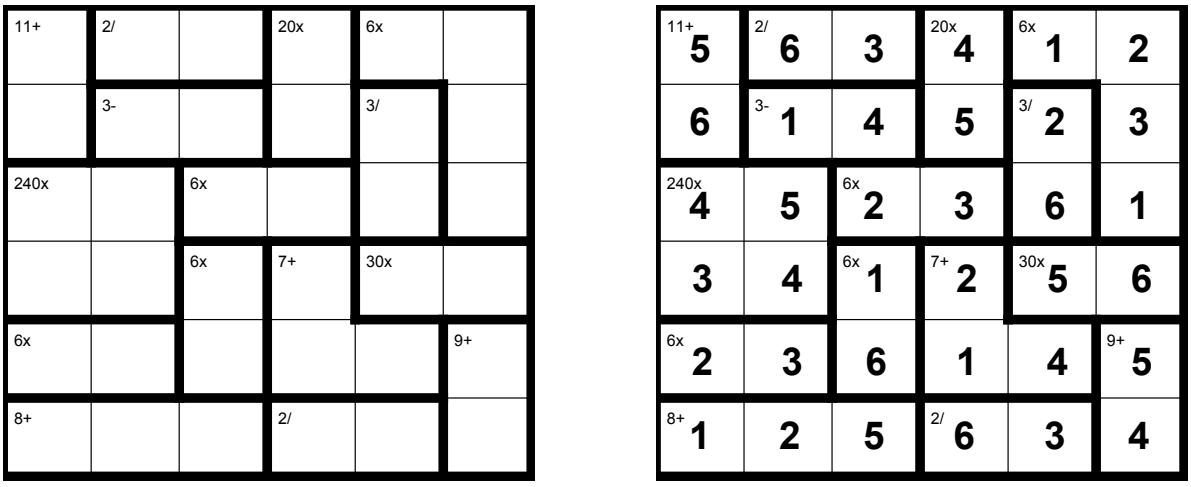

Figure 1: An example of a 6×6 FunPuzz grid with its start state (left) and solution (right).

FunPuzz Formal Description

The FunPuzz puzzle has the following formal description:

- FunPuzz 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.
- FunPuzz 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.

Additional notes:

- It is possible for a given cell to not participate in any cage constraints. That is still a valid FunPuzz board.
- For division, we are only concerned with divisions producing integer results throughout the operation.

Question 1: Propagators **(worth 55/100 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` and `propagators.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. 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 25/100 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 30/100 marks)
: 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: CSP Encodings **(worth 45/100 marks)**

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

Brief implementation description: The three encodings take as input a valid FunPuzz 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 grid $((3),(11,12,13,6,0),(21,22,31,2,2),....)$ corresponds to a 3x3 board[1] 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...

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

All encodings 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 encodings do not need to encode the *cage* constraints.

You must implement:

`binary_ne_grid` (worth 10/100 marks)

An encoding of a FunPuzz 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)

An encoding of a FunPuzz grid (without *cage* constraints) built using only *n*-ary all-different constraints for both the row and column constraints.

`caged_csp` (worth 25/100 marks)

An encoding built using your choice of (1) binary binary not-equal, or (2) *n*-ary all-different constraints for the grid, together with (3) FunPuzz *cage* constraints. That is, you will choose one of the previous two grid encodings and expand it to include FunPuzz *cage* constraints.

Notes: The CSP encodings 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 encoding). Also be mindful of the **time** complexity of your methods for identifying satisfying tuples, especially when coding the `caged_csp`.

HAVE FUN and GOOD LUCK!