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 three heuristics—Minimum-Remaining-Value (MRV), Degree (DH), 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.
- **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, ord_dh, and val_lcv.

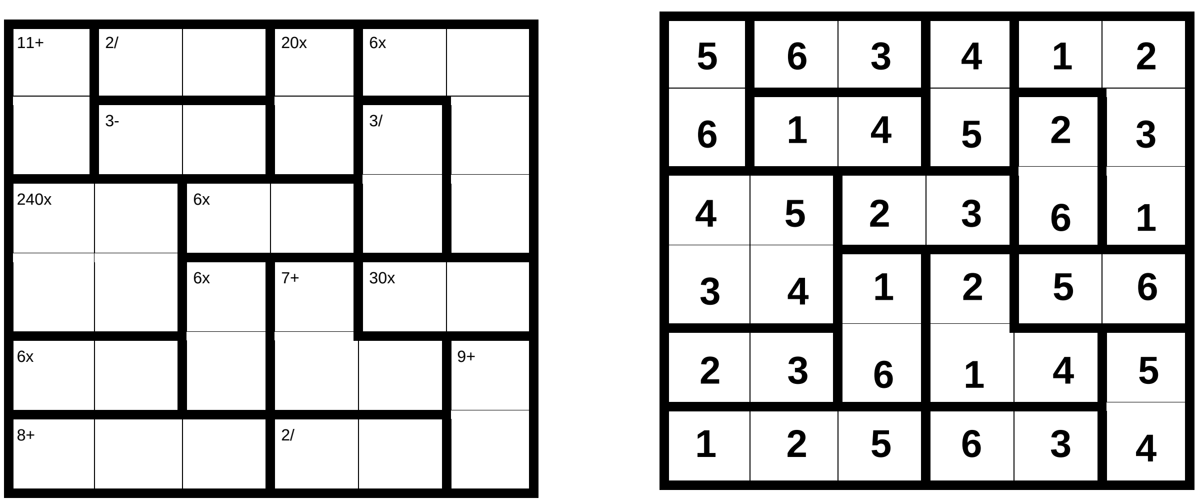

Figure 1: An example of a 6 × 6 KenKen grid with its start state (left) and solution (right).

- **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.
- **tests.py**. Sample test cases. Run the tests with “python3 tests.py”.

KenKen Formal Description

The KenKen puzzle 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.

Question 1: Propagators

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. val_lcv takes a CSP object csp and a Variable object var as input, and returns a value 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

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

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

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)

ord_dh

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 on other unassigned variables.

val_lcv

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

Question 2: Models

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 cells 21, 22, and 31 must be 2. That is (C21/C22)/C31 = 2, where C21, C22, and C31 are the assigned values of cells 21, 22, and 31 respectively.

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

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

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

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 CSP model). Also be mindful of the **time** complexity of your methods for identifying satisfying tuples, especially when coding the kenken_csp_model.