University of Alberta
Computing Science Department
CMPUT 366 - Fall 2022

Assignment 4
Due date: December 2
12 marks

# Search & Planning in AI (CMPUT 366)

## Submission Instructions

Submit on eClass your code as a zip file and the answer to the question of the assignment as a pdf. The pdf must be submitted as a separate file so we can more easily visualize it on eClass for marking. You shouldn't send the virtual environment in the zip file.

## Overview

In this assignment you will implement a Constraint Satisfaction solver for Sudoku. If you aren't familiar with Sudoku, please review the notes for Lecture 16. In the notes we describe a $4 \times 4$ puzzle with units of size $2 \times 2$ and variables with domain $\{1, 2, 3, 4\}$. In this assignment we will solve the traditional $9 \times 9$ Sudoku puzzles with units of size $3 \times 3$ and variable domains of $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

## How to Run Starter Code

Follow the steps below to run the starter code (instructions are for Mac OSx and Linux).

- Install Python 3.
- It is usually a good idea to create a virtual environment to install the libraries needed for the assignment. The virtual environment step is optional.
  - `virtualenv -p python3 venv`
  - `source venv/bin/activate`
  - When you are done working with the virtual environment you can deactivate it by typing `deactivate`.
- Run `pip install -r requirements.txt` to install the libraries specified in requirements.txt.

You are now ready to run the starter code which has two main files: `main.py` and `tutorial.py`. If everything goes as expected as you run `python3 tutorial.py`, you should see several messages, which are part of the tutorial of this assignment. The tutorial is described in detail below. You will implement the algorithms that are asked in the assignment in file `main.py`.

## 1 Tutorial (0 Marks)

A large portion of your CSP solver is already implemented in the starter code. This tutorial will teach you how to use the code that is given to you.

## Reading Puzzle

In this tutorial we will use the puzzle from the file `tutorial_problem.txt`, which is given by the string

```
4.....8.5.3..........7......2.....6.....8.4......1.......6.3.7.5..2.....1.4......
```

There are 81 characters in the line above, one for each variable of the puzzle. The dots represent the variables whose values the solver will need to find the values for; the values represent the cells that are filled in the puzzle.

If you want to read all puzzles from a file and iterate through them, you will use the following lines of code.

```
file = open('tutorial_problem.txt', 'r')
problems = file.readlines()

for p in problems:
    g = Grid()
    g.read_file(p)
```

Here, we will iterate over all problems in the file `tutorial_problem.txt`. Since there is only one puzzle in this file, the for loop will complete a single iteration. You will need to solve more instances later, so this for loop will be helpful. All instructions described in this tutorial are assumed to be in this for loop, as you can verify in `tutorial.py`. The code above creates an object in memory and stores the domains of all variables in the puzzle. For example, the domain of the variable at the top-left corner of the puzzle should be '4', while the domain of the second variable in the same row should be '123456789' because that variable wasn't assigned yet.

## Printing Puzzle

Let's start by printing `g` on the screen. The class Grid from the starter already comes with a function to print the puzzle on the screen, which can be quite helpful to debug your implementation.

```
g.print()
```

The code above will print the following on the screen.

```
- - - - - - - - - - - - -
| 4 . . | . . . | 8 . 5 |
| . 3 . | . . . | . . . |
| . . . | 7 . . | . . . |
- - - - - - - - - - - - -
| . 2 . | . . . | . 6 . |
| . . . | . 8 . | 4 . . |
| . . . | . 1 . | . . . |
- - - - - - - - - - - - -
| . . . | 6 . 3 | . 7 . |
| 5 . . | 2 . . | . . . |
| 1 . 4 | . . . | . . . |
- - - - - - - - - - - - -
```

Class Grid also has a method to print the domain of all variables, which can also be helpful for debugging.

```
g.print_domains()
```

The code above will print the following on the screen.

```
['4', '123456789', '123456789', '123456789', '123456789', '123456789', '8', '123456789', '5']
['123456789', '3', '123456789', '123456789', '123456789', '123456789', '123456789', '123456789', '123456789']
['123456789', '123456789', '123456789', '7', '123456789', '123456789', '123456789', '123456789', '123456789']
['123456789', '2', '123456789', '123456789', '123456789', '123456789', '123456789', '6', '123456789']
['123456789', '123456789', '123456789', '123456789', '8', '123456789', '4', '123456789', '123456789']
['123456789', '123456789', '123456789', '123456789', '1', '123456789', '123456789', '123456789', '123456789']
['123456789', '123456789', '123456789', '6', '123456789', '3', '123456789', '7', '123456789']
['5', '123456789', '123456789', '2', '123456789', '123456789', '123456789', '123456789', '123456789']
['1', '123456789', '4', '123456789', '123456789', '123456789', '123456789', '123456789', '123456789']
```

Here, each list contains the domains of each variable in a row of the puzzle. For example, the first element of the first row is the string '4' because the grid starts with the number 4 in that position. The second element of the same list is the string '123456789', because any of these values can be used in that cell.

## Going Through Variables and Domains

The Grid class has an attribute for the size of the grid (`_width`), which is always 9 in this assignment. You can either hardcode the number 9 when you need to go through the variables or use the function `get_width()`, as we do in the code below.

```
for i in range(g.get_width()):
    for j in range(g.get_width()):

        print('Domain of ', i, j, ': ', g.get_cells()[i][j])

        for d in g.get_cells()[i][j]:
            print(d, end=' ')
        print()
```

In this code we iterate through every cell of the grid, which are accessed with the operation `g.get_cells()[i][j]`. The method `get_cells()` returns the grid, and the part `[i][j]` accesses the string representing the domain of the i-th row and j-th column of the puzzle. The innermost for-loop goes through all values in the domain of the $(i, j)$ variable.

## Making Copies of the Grid

The Backtracking search is easier to implement if we make a copy of the grid for each recursive call of the algorithm. That way we make sure that the search in a subtree won't affect the grid of the root of the subtree. Here is how to create a copy of a grid.

```
copy_g = g.copy()

print('Copy (copy_g): ')
copy_g.print()

print('Original (g): ')
g.print()
```

The code above should print exactly the same grid. Despite being identical, variables `copy_g` and `g` refer to different objects in memory. If we modify the domain of one of the variables in `copy_g`, that shouldn't affect the domains of `g`. This is illustrated in the code below, where we remove '2' from the domain of variable $(0, 1)$ of the grid `copy_g`, but not of the grid `g`.

```
copy_g.get_cells()[0][1] = copy_g.get_cells()[0][1].replace('2', '')
copy_g.print_domains()
g.print_domains()
```

### Arc Consistency Functions

The code starter also comes with three functions you will use to implement Forward Checking. The functions receive a variable $v$ that makes all variables in the $v$'s row (`remove_domain_row`), column (`remove_domain_column`), and unit (`remove_domain_unit`) arc-consistent with $v$. The following code excerpt removes '4' from the domain of all variables in the row of variable $(0, 0)$.

```
failure = g.remove_domain_row(0, 0)
```

The variable `failure` indicates whether any variable had their domain reduced to the empty set during the operation. If `failure` is true, then the search should backtrack immediately as the current assignment renders the problem unsolvable. We can perform similar operations with the row and the unit of $(0, 0)$.

```
failure = g.remove_domain_column(0, 0)
failure = g.remove_domain_unit(0, 0)
```

All three functions assume that the value of the variable $(i, j)$ passed as input was already set (i.e., the domain of the variable is of size 1). In our example, while it makes sense to pass variable $(0, 0)$ as input, it doesn't make sense to pass variable $(0, 1)$. This is because the domain of $(0, 1)$ is larger than 1.

## 2 Implement Backtracking Search (4.5 Marks)

Considering the partial implementation provided in the starter code, implement the following functions.

1. (0.5 Mark) Implement a function `select_variable_fa`. This function must receive an instance of a Grid object and return a tuple $(i, j)$ with the index of the first variable on the grid whose domain is greater than 1. The function can iterate through the grid in whichever order you prefer (e.g., from left to right and top to bottom). This is a naïve variable selection heuristic we will use in our experiments.

2. (1 Mark) Implement function `select_variable_mrv`. This method must receive an instance of a Grid object and return a tuple $(i, j)$ according to the MRV heuristic we studied in class. The implementation can break possible ties whichever way you prefer.

3. (3 Marks) Implement function `search(self, grid, var_selector)`. This function should perform Backtracking search as described in the pseudocode below. The variable `var_selector` is either the function `select_variable_fa` or `select_variable_mrv`. The order in which we iterate through the domain values (see line 4) is arbitrary. The conditional check in line 5 should verify if the value $d$ violates a constraint in the puzzle. For example, we can't set the value of 4 to the second variable in the first row of our example because the first value is already 4. It is helpful to implement Backtracking such that it returns two values: a grid with a possible solution and a Boolean value indicating if a call to Backtracking was successful or not; the pseudocode below is simplified and it returns a single value.

```
1  def Backtracking(A, var_selector):
2    if A is complete: return A
3    var = var_selector(A)
4    for d in domain(var):
5      if d is consistent with A:
6        copy_A = A.copy()
7        {var = d} in copy_A
8        rb = Backtracking(copy_A)
9        if rb is not failure:
10         return rb
11   return failure
```

Use the instance from file `tutorial_problem.txt` to test your implementation. Backtracking without inference takes too long to solve some of the puzzles from `top95.txt`, so we will save those for later.

## 3 Implement Forward Checking (4.5 Marks)

You will implement a domain-specific version of Forward Checking. Instead of creating a constraint graph to apply Forward Checking, we will implement a version specific for Sudoku that is much simpler. In Sudoku we only change the domain of a variable when a neighbor variable has its value assigned. For example, we if assign the value of 2 to a variable $v$, then we can remove the value of 2 from the domain of all variables in the same row, column, and unit of $v$.

The pseudocode for our domain-dependent Forward Checking can be written as follows. This algorithm should be called for after assigning a value to a variable `var` during search, so that it simplifies the domain of the variables connected to `var`.

Forward Checking receives a partial assignment `A` (instance of class Grid in the starter code) and a variable `var` to be processed. Forward Checking returns 'failure' if one of the variables in `A` was reduced to an empty domain; it returns 'success' otherwise. Considering the partial implementation provided in the code starter, what you have already implemented, and the discussion above, implement the following functions.

```
def forward_checking(A, var):
  remove_rows(A, var)
  remove_columns(A, var)
  remove_units(A, var)

  return success or failure
```

1. (2 Marks) Implement function `forward_checking`. This function should implement the Forward Checking pseudocode shown above.
2. (1 Mark) Implement function `pre_process_forward_checking`. This method should be called just once, before the search starts. In this function, you will call `forward_checking` for each variable whose value was already set in initial grid of the puzzle.
3. (1.5 Marks) Modify your Backtracking implementation to run `forward_checking` for each value assigned during the Backtracking search. If the inference returns failure, the search should backtrack immediately. Backtracking should also call `pre_process_forward_checking` before the search starts.

## 4 Plot and Discuss Results (3 Marks)

Like in Assignments 1 and 2, generate a scatter plot for the 95 problems in file `top95.txt` where the x-axis shows the running time in seconds of Backtracking with the MRV heuristics and the y-axis the running time of Backtracking with the "first available" heuristic. The code required to plot the results is available in the code starter. Here is an example of how to use the plotting function.

```
plotter = PlotResults()
plotter.plot_results(running_time_mrv, running_time_first_available,
"Running Time Backtracking (MRV)",
"Running Time Backtracking (FA)", "running_time")
```

In the code above, `running_time_mrv` and `running_time_first_available` are lists containing the running time in seconds for each of the 95 Sudoku puzzles. The first entry of each list is the running time of the two approaches for the first instance, the second for the second instance and so on.

Explain and discuss the results you observe in the scatter plot; please include the plot in your answer. Your answer should not be long. All you need to do is to describe the results you observe and explain them (i.e., why the points in the scatter plot are distributed the way they are?).