University of Alberta
Computing Science Department
CMPUT 366 - Fall 2024

Assignment 3
Due date: November 29
8 marks

Search & Planning in AI (CMPUT 366)

Submission Instructions

Submit on Canvas your code as a zip file. You should not 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 Section 5.1.2 of the lecture notes. In the notes, we describe a 4×4 puzzle with units of size 2×2 and variables with domain $\{1, 2, 3, 4\}$. In this assignment, we will solve the traditional 9×9 Sudoku puzzles with units of size 3×3 and variable domains of $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

How to Run Starter Code

You will need Python 3, Numpy, and Matplotlib installed to run the starter code. The starter can be run with: `python3 main.py`. If everything goes as expected, you should see several messages, which are part of the tutorial of this assignment. The tutorial is described in detail below.

1 Tutorial (0 Marks)

A large portion of your CSP solver is already implemented in the code starter. This tutorial will teach you how to use the code that is given to you. The tutorial code is available in `main.py` in the folder of the code starter. **Please remove the tutorial code entirely before submitting your solution on Canvas.**

Reading Puzzle

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

```
4..5..7....1..2.8......79...36.4...2...2......8..3...6...9.85..1....58..3..6...1.
```

There are 81 characters in the line above, one for each variable of the puzzle. The dots represent the variables whose values the solver needs to find; 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 loop, as you can verify in `main.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 on the same row should be '123456789' because that variable isn't assigned.

Printing Puzzle

Let's start by printing `g` on the screen. The class Grid from the code 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 . . | 5 . . | 7 . . |
| . . 1 | . . 2 | . 8 . |
| . . . | . . 7 | 9 . . |
- - - - - - - - - - - - - -
| . 3 6 | . 4 . | . . 2 |
| . . . | 2 . . | . . . |
| . 8 . | . 3 . | . . 6 |
- - - - - - - - - - - - - -
| . . . | 9 . 8 | 5 . . |
| 1 . . | . . 5 | 8 . . |
| 3 . . | 6 . . | . 1 . |
- - - - - - - - - - - - - -
```

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', '5', '123456789', '123456789', '7', '123456789', '123456789']
['123456789', '123456789', '1', '123456789', '123456789', '2', '123456789', '8', '123456789']
['123456789', '123456789', '123456789', '123456789', '123456789', '7', '9', '123456789', '123456789']
['123456789', '3', '6', '123456789', '4', '123456789', '123456789', '123456789', '2']
['123456789', '123456789', '123456789', '2', '123456789', '123456789', '123456789', '123456789', '123456789']
['123456789', '8', '123456789', '123456789', '3', '123456789', '123456789', '123456789', '6']
['123456789', '123456789', '123456789', '9', '123456789', '8', '5', '123456789', '123456789']
['1', '123456789', '123456789', '123456789', '123456789', '5', '8', '123456789', '123456789']
['3', '123456789', '123456789', '6', '123456789', '123456789', '123456789', '1', '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

Class Grid has an attribute for the size of the grid (`_width`), which is set to 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 AC3. 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 $(0, 0)$.

```
    ac3 = AC3()
    variables_assigned, failure = ac3.remove_domain_row(g, 0, 0)
```

The variable `variables_assigned` contains the variables that had their domains reduced to size 1 while removing the value of $(0, 0)$ from their domain. The information in `variables_assigned` is important because AC3 will add all incoming arcs related to these variables in its set. See more information about AC3 below. 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 as the current assignment renders the problem unsolvable. We can perform similar operations with the row and the unit of $(0, 0)$.

```
variables_assigned, failure = ac3.remove_domain_column(g, 0, 0)
variables_assigned, failure = ac3.remove_domain_unit(g, 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.

Verifying the Correctness of Solutions

The base case of backtracking requires one to check whether the current variable assignment represents a solution to the problem. The solution of a grid can be verified with method `is_solved` from the Grid class.

```
print('Is the current grid a solution? ', g.is_solved())
```

2 Implement Backtracking Search (4 Marks)

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

1. (1 Mark) Implement method `select_variable` from class `FirstAvailable`. This method receives an instance of a Grid object and it returns a tuple (i, j) with the index of the first variable on the table whose domain is greater than 1. This is a naïve variable selection heuristic we will use in our experiments.

2. (1 Mark) Implement method `select_variable` from class `MRV`. This method receives an instance of a Grid object and it returns a tuple (i, j) according to the MRV heuristic for variable selection we studied in class.

3. (2 Marks) Implement method `search(self, grid, var_selector)` in the code starter. This method should perform Backtracking search as described in the code below. The variable `var_selector` is either an instance of `FirstAvailable` or `MRV`, as you have implemented above. 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 would violate 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.

```
1  def Backtracking(A):
2    if A is complete: return A
3    var = select-unassigned-var(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 Arc Consistency (4 Marks)

You will implement a domain-specific version of AC3. Instead of creating a constraint graph and follow the AC3 steps we discussed in class, 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.

In our domain-dependent implementation of AC3, instead of keeping a set with the arcs, we will simply keep a set with the variables (i, j) that need to be processed. If the domain of a variable (k, p) is reduced to the size of 1 while removing the value of (i, j) from its neighbors, then we add (k, p) to the set to be processed. The pseudocode for our domain-dependent AC3 is shown on the next page.

AC3 receives a partial assignment (instance of class Grid in the code starter) and a set Q containing the variables that need to be processed. For example, if using the procedure above in a pre-processing step, then Q should contain all variables that are assigned in the puzzle. If used during search, then Q should contain only the variable for which the Backtracking search has assigned a value.

Considering the partial implementation provided in the starter, what you have already implemented, and the discussion above, implement the following functions.

1. (2 Marks) Implement the method `consistency` from the starter. This method should implement our domain-specific AC3 pseudocode.

```
def ac3(A, Q):
  while Q is not empty:
    var = Q.pop()

    remove_rows(A, var)
    remove_columns(A, var)
    remove_units(A, var)

    if any removal returned failure:
      return failure

    add to Q all variables that had their domains reduced to size 1
  return sucesss
```

2. (1.0 Mark) Implement the method `pre_process_consistency`. This method should be called just once, before the search starts. Here, you will initialize the set Q with all variables whose values were already set in the initial grid of the puzzle and then call the method `consistency` you have already implemented.

3. (1.0 Mark) Modify your Backtracking implementation to run the `consistency` method you have implemented for each value assigned during the Backtracking search. If the inference returns failure, the search should backtrack (i.e., try another value assignment for the current variable). Backtracking should also call `pre_process_consistency` before starting the search.

4 Questions to Consider (0 Marks)

You will generate a scatter plot for the 95 problems in file `top95.txt` where the x-axis will show 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.

Do the results you observe make sense? Would you be able to explain them?