University of Antwerp
Faculty of Science

Artificial Intelligence

Assignment 4: Constraint Satisfaction Problems

Deadline: November 12, 2023

General Remarks

- You are free to consult with other students. The assignment is, however, individual, meaning that each submitted assignment has to be sufficiently unique.
- Always motivate your answers. Detail the steps needed to come to your conclusions.
- You are free to provide answers in either English or Dutch.
- If anything is unclear or if you experience technical problems with the assignment, contact me at joey.depauw@uantwerpen.be.

Assignment (15 points)

For this assignment you will implement a CSP solver that can solve **any** constraints satisfaction problem (with binary constraints). A template to get you started is already provided. There are five files in the template, two of which you need to complete. However, it can be interesting to first look at all of them to get a better understanding of how everything works:

CSP.py
: Contains the definition of CSP and three solving algorithms which you need to implement: brute force, forward checking and arc consistency.

Sudoku.py
: An example CSP is Sudoku. Define the constraints in this file and see whether your solver can find a solution to even the hardest of sudoku puzzles.

NQueens.py
: Another CSP is the NQueens problem (discussed in the theory). This one is already implemented for you.

solver.py
: Use this file to call your solutions from the command line.

util.py
: Contains miscellaneous code used throughout the template.

You will need python3.8 with the typer and tqdm[1] modules to run the code. Verify that your environment is working by using the following command:
`>python solver.py queens`
It should report that no solution is found.

Problem 1 - Backtracking (3pt)

Your first task is to implement a basic backtracking algorithm to solve CSP problems in general. The NQueens problem can be used as an example to test your code. Implement the functions:

CSP.py (2pt)

- `CSP::isComplete`
- `CSP::isValid`
- `CSP::_solveBruteForce`

After doing this, you should already be able to solve easy versions of the puzzle. You can also try it for higher values of n:
`>python solver.py queens --n 10`

Questions

1. How many calls does your algorithm need (on average) for n=10? Is there a lot of variation in the amount of calls when you try this multiple times? (Hint: report a few runs in a table and calculate the mean and standard deviation.) (1pt)

Problem 2 - Forward Checking (5pt)

A more efficient implementation takes the impact of an assignment on the domains of other variables into account. Implement the forward checking algorithm from the theory lectures and adapt your backtracking algorithm from problem 1 using the following templates:

CSP.py (2pt)

- `CSP::_solveForwardChecking`
- `CSP::forwardChecking`

You can try solving a more difficult variant of the puzzle with forward checking:
`>python solver.py queens --method fc --n 50 --no-mrv --no-lcv`
Try a few different values of n to see whether this method works better than brute force already.

[1]Install with: `pip install typer tqdm`

Now find a better implementation for the following functions (see slides) and inspect the impact on solving the NQueens problem:

CSP.py (1pt)

- `CSP::selectVariable`
- `CSP::orderDomain`

Questions

2. Report on the average amount of calls required for n=50 with and without `CSP::selectVariable` and `CSP::orderDomain` (you can use the flags `--no-mrv` and `--no-lcv`). Can you explain why these two functions make the algorithm faster/slower? (2pt)

Problem 3 - AC3 (4pt)

Since all our constraints are binary, we also want to implement the AC3 algorithm for checking binary constraints. Implement the following functions:

CSP.py (2pt)

- `CSP::_solveAC3`
- `CSP::ac3`

Use this command to test your implementation:

```
>python solver.py queens --method ac3 --n 50
```

Questions

3. Report on the average calls required for n=50 with and without the improved `CSP::selectVariable` and `CSP::orderDomain`. Discuss their impact and relate your answer to question 2. (2pt)

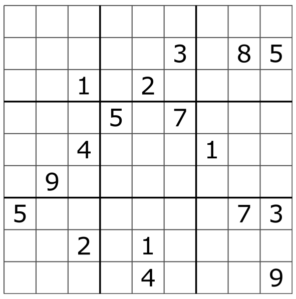
Problem 4 - Sudoku (3pt)

Sudoku is a popular game based on logic and, indeed, constraint satisfaction. The goal is to fill the board with the numbers 1 through 9, such that every column, every row, and the 3-by-3 squares that cover the field all contain exactly the numbers 1 through 9. An example started puzzle is shown in Figure 1.

Your solver should now be able to solve this CSP as well, provided you define its variables and constraints. Complete the following functions:

Sudoku.py (2pt)

- `Sudoku::__init__`
- `Sudoku::variables`
- `Sudoku::getCell`



					3		8	5
		1		2				
			5		7			
		4				1		
	9							
5							7	3
		2		1				
				4				9

Figure 1: Example of a Sudoku instance to be solved.

- `Sudoku::neighbors`
- `Sudoku::isValidPairwise`
- `Cell::__init__`
- `Cell::startDomain`

If you have trouble implementing these functions, you can take a look at the `NQueens` problem definition first to get an idea of how they should work.

When this is done, you can test your implementation with the following command:

```
>python solver.py sudoku puzzles/easy.txt --method ac3
```

Questions

4. Can you solve the hard puzzle with any of your algorithms? Report on the runtime and amount of calls required to solve it with forward checking and with AC3. (1pt)

Feedback (0pt)

Finally, feel free to include some feedback on the assignment in your report, specifically remarks on the workload, difficulty and relevance of the assignment. Suggestions on how to improve it towards next year are also welcome. Naturally, what you write here will not affect your grades in any way.

Questions

5. (Optional) Evaluate the assignment.

Submit

Complete the provided template code. Upload a **zip** file containing **only** the files `report.pdf`, `Sudoku.py` and `CSP.py` via Blackboard. Always be formal *and* provide explanation where appropriate. Good Luck!