159.271 Computational Thinking

Assignment 1: Sudoku Solver

This assignment is worth 15% of your final mark. It will be marked out of a total of 20 marks. The due date for this assignment is Monday April 26th 11:55pm China time. You are expected to work on this assignment *individually* and all work that you hand in is expected to be your *own* work.

In this assignment your task is to write a Sudoku solver. You will develop an animated application that uses recursive backtracking as the basis of the implementation. I'm going to assume that you know how to solve Sudoku already, since, if not, then you must have been hiding under a rock for the past 10 years!

Go to Stream and download the folder

`Assignment1 code template`

This folder contains a template program that implements the code needed to read Sudoku puzzles from files and to display them, and to run the solver in animation mode. Set up a new project in your IDE and add the folder **`src`** to it. The main game module is **`SudokuApp.py`**. If you run this, a screen will appear and clicking on either the 'e' or the 'h' key will display a randomly chosen puzzle (either easy or hard) from one of the two folders of puzzles included.

You need to complete the module **`Solver.py`**, so that a call to the **`solve`** function will do a bit more than just displaying the puzzle. You need to turn this function into a recursive backtracking solver, which, for animation purposes, displays a snapshot of the current state of the puzzle at the start of each recursive call.

The program uses two classes to represent a Sudoku – **`Cell`** and **`Snapshot`**. **`Cell`** represents a single cell in a Sudoku grid, and has methods to get and set the position of the cell and the value of the cell. In any complete solution, the value must be between 1..9. If the value is 0 this means that the cell is still empty. **`Snapshot`** describes a state of the Sudoku at a certain point in time – some cells have values (other than zero), some may not. The **`Snapshot`** class has methods that allow to clone a snapshot (this is useful to produce the next level of snapshots in the recursion tree), to query the cells in various ways, and to set cells.

`Sudoku_IO.py` contains the code used to read Sudoku puzzles from puzzle files and to display them. The puzzle files consist of 9 lines containing 9 digits each, encoding the initial values for the board cells.

In general, to develop a recursive backtracking algorithm you need to consider:

1. What question should we ask? Thinking only about the top level of recursion, you need to formulate one small question about the solution that is being searched for. Remember that, in

order for the final algorithm to be efficient, it is important that the number of possible answers is small.
2. How should we construct a subinstance? How to express the problem that we want to solve as a smaller instance of the same search problem?

1. An input instance to the problem will be a **`snapshot`** specifying the current state of the Sudoku. A solution is a valid way of filling the remaining empty cells.
2. The simple question to ask is "What number can go in the next empty cell?"

The brute force strategy is simply to find the next empty cell in the snapshot, working left to right, top to bottom, and then to try all possible numbers in that cell. Initial pruning of the recursion tree should include two strategies - 1. we don't continue on any branch that has already produced an inconsistent (invalid) solution and 2. we stop once a complete solution has been found. Python code examples that implement this basic strategy for two straightforward problems have been given in the lecture slides.

A smarter approach is to look for 'singletons', i.e., those cells that can only have one possible number in them. So, for each cell you need to keep a list of which numbers are allowed within it, and then check the relevant row, column, and 3 ◎x 3 block to remove those numbers that have already been used. Of course, if you end up in a situation where there are no possible numbers that can go in a cell then you've done something wrong, so your algorithm should stop and backtrack. You will need to add attributes and methods to the **`Cell`** class to implement this strategy.

Once you have run out of singletons you need another strategy. Perhaps you keep a list of the cells in the current snapshot, sorted in order of the lengths of their 'possibles' lists. There are other intelligent things that can be done. You can 'cross-reference' between the 'missing' values in a row, column, or block and the 'possibles' lists for its empty cells. If a 'missing' value appears in only one of the 'possibles' lists, then this value must go in the cell.

Marking scheme

1. 10 marks for a correct program that uses a recursive backtracking strategy with basic pruning.
2. 5 marks for choosing 'singletons' ahead of other cells. With this approach, for easy puzzles, your recursion tree will become simply a path, since at each stage you will find at least one cell that can only have one value.
3. 5 marks for more sophisticated pruning strategies, either those described here or others, that allow for fast backtracking over hard puzzles. You will need to submit a readMe document briefly describing the more sophisticated strategies that you have employed.

We will test your program using puzzles similar to those provided, for both easy and hard categories.

Submission

Submit your project in Stream as a single zipped file containing your completed code, contained in the folder **`Sudoku_YourID`**, and your completed readMe document.

Plagiarism

There are many, many Sudoku solvers available on the internet. I am sure that many of these are recursive backtracking solutions implemented in Python. For this reason, and also so that the concepts taught in lectures are reinforced, you **must** use the template provided as the basis of your implementation. You can improve the display or the animation however you like, you can add more classes or improve those provided (this part you probably actually need to do), you can make the solver as sophisticated as you like. **What you cannot do is download a solution from the internet and submit it as your own work.** If you try this, **you will get no marks**. Make an attempt to read the course notes, to follow the lecture slides and to apply the concepts discussed, in completing this assignment.

As stated at the beginning of this document, you are expected to work on this assignment *individually* and all work that you hand in is expected to be your *own* work. You are allowed to discuss any aspect of this assignment with other class mates, in person, or electronically. **What you are not allowed to do is to submit someone else's solution as your own work.** If it appears that this is the case, **you will get zero marks**.

Late submission:

Late assignments will be penalised 10% for each weekday past the deadline, for up to five (5) days; after this no marks will be gained. In special circumstances an extension may be obtained from the paper co-ordinator, and these penalties will not apply. Workload will not be considered a special circumstance – you must budget your time.