CSCI-630-02 (Foundations of Intelligent Systems)
Lab 1

Kunal Shitut(ks6807)

Input Parsing Finding a solution to the ripple effect puzzle effectively requires us to keep track of two major factors for every puzzle cell, the position of the cell in the puzzle and the region to which the puzzle cell belongs. This is why I decided to use two data structures to keep track of the puzzle. The first is 2 dimensional array where puzzle cells are stored in the position that they appear in the puzzle. This makes it easy to access the cells and their values using their position in the puzzle. Each element of the array is an object of the class `PuzzleCell`. We also need to keep track of the region to which a cell belongs and for this purpose I have used a python list which contains a sub-list for each region in the puzzle. This two data structures are used to whether a value can be assigned to a particular cell.

The input file provided by the user is processed by the method `processPuzzleArray` which outputs the array of puzzles called puzzle matrix and list of regions called region list. This method first initializes the matrix and the list. After this it starts from the second line of the puzzle in the input file. It then loops through every line and every character of the puzzle and uses variables `columnCounter` and `rowCounter` to keep track of the row and column to which a cell belongs. once a "." is encountered it reads the counters to check if a cell has already been added to the puzzle matrix. If it has not been added then the puzzle cell is added to the puzzle matrix and a new region is created for it in the region list. After this the method `getAllRegionCells` is called to parse the file and add all the cells that belong to the previous cell's region to its region and puzzle matrix. `getAllRegionCells` does this by recursively calling itself in every direction until it encounters a '|' or '-' which indicates the end of region in that direction. Once all cells for a particular region have been parsed, the method moves to the immediate next '.', where it checks the counters to see if the cell at this position has been added to the puzzle or not. If it has not been parsed then this means that it is part of a new region and thus that region is explored similarly to previous ones. Thus after the puzzle file has been parsed, we get the output in variable `puzzleMatrix` and `regionList`.

MRV and Forward Checking The implementation of MRV or Minimum Remaining Value heuristic and Forward Checking in the intelligent solver has drastically reduced the runtime of the puzzle solver for larger puzzles. For implementing the MRV, I have created inside each puzzle cell object the list of all values that are allowed for that cell. Initially this includes values from 1 to the length of the region, but then I eliminate MRV values for all cells based on fixed value cells near them or in their region. Once this has been done, I insert values for all cells whose region length is 1 and eliminate the value 1 from the MRV list of adjacent cells. This process is done using methods `fixedValueMRVAdjust` and `solveOneValMRV`. Both methods use the implementation of forward checking in the method `forwardCheck`. The method `forwardCheck` takes a puzzle cell, the region list and the puzzle matrix and then for the value assigned to that puzzle cell, eliminates that value from the MRV list of other puzzle cells that are in the region of that cell and adjacent to that cell in the range of that value. After this process is done, the modified puzzle matrix and region list is sent to the intelligent solver in the method `intelligentSolver`. The method `intelligentSolver` takes the region list and puzzle matrix, then scans through both of them to find the cell that has the least number of remaining values using the method `findMinMRV`, which simply checks all cells in a puzzle matrix to find the cell with minimum remaining values. If the row of cell is -1, this means that we haven't found any cell whose value has not been assigned and thus the puzzle has been solved. If the number of minimum remaining values is 0, this means we have failed and thus must go back and try a combination of other values. For each value in the MRV list, we assign it to the cell, forward check it from the nearby cells, and then again call the intelligent solver for solving the remaining puzzle. If the value returned by the recursive call to intelligent solver is false, then we check for other values in the list or having exhausted the values, simply return False. If it is true then, we have solved the puzzle and therefore return True. In case of False, we first remove the assigned value from that cell and then add that value to all the MRV of all

cells from which we had removed that value. The track of those cells is kept using the list variable `listOfCheckedCells`. This is how the intelligent solver has been implemented using **MRV** and **Forward Checking**

Empirical Reporting and Measurements The performance of every solver is measured on 2 factors: the actual time taken in milliseconds and the number of recursive calls to the solver. For most of the smaller and easier puzzles, the brute force solver and the intelligent solver is usually in the range of 10 milliseconds but for the larger ones there is a great amount of time difference between the two solvers. Sometimes for a simple puzzle, the brute force solver is faster than the intelligent solver as seen in case 2. Given below is a comparison of time and recursive calls for the puzzles that were given and an additional puzzle that I have provided along with my code.

Puzzle	**Bruteforce solver time in ms**	**Bruteforce solver recursive calls**	**Intelligent solver time in ms**	**Intelligent solver recursive calls**
7×6 puzzle from lab website	7.3	678	2.92	87
Sample 4 by Hammy	0.89	101	3.5	101
Sample 9 by Casty	12.74	1092	4.39	97
Competition puzzle by TKarino	391964	41901288	1684	16066
Sample problem 10 by T.Karino*	87.16	8987	10.1	173

* Please check the attached puzzle file for converted copy of the puzzle

As can be observed the intelligent solver shows great improvement when it comes to solving the difficult competition puzzle.

Conclusions So based on the measurements taken we observe that while small puzzles are solved easily by the brute force solver, it becomes exponentially slow for large puzzles. This means that a major factor in the improvement of solvers depends on the size of the puzzle. But, the other significant factor is the number of fixed values provided. For instance, the sample 4 and sample 9 puzzles though same in size differ greatly in number of recursive calls because of the fixed values for some cells.

This insight can be used for creating a program that creates Ripple Effect puzzle. The puzzle creator can be programmed to create puzzles of varying difficulty by varying the number of regions and fixed values. A harder puzzle would be created if it has more fixed values in large regions but the actual number of allowable values for each cell is less. This would provide a good challenge for the human solver as limited values for each cell in large regions would require thinking rather than guess work.

Instructions for running The program given in `re_solver.py` asks for a input file which is the text representation of the puzzle. The first line is taken as the number of rows and columns and the puzzle start is considered from the next line. After processing the input file, the program asks the user about the solver to be used which could be Bruteforce(Enter 'B') or the Intelligent(Enter 'I') solver. Once the puzzle produces the output, the user is then asked if he wants to continue with another puzzle or stop the program.