Assignment 2-Sudoku, CSE 537

Abhiroop Dabral Ankit Dave Rishi Josan

In this assignment we implemented two approaches to solve a given Sudoku problem.

Backtracking Approach: *(Python sudokuSolver.py backtracking)*

First the basic approach was implemented which uses simple backtracking to solve a given Sudoku.

This approach backtracks and corrects a previous assignment whenever it can't assign a valid value to the present cell it is working on. A valid assignment is defined as not having the same value in the corresponding row or column and the 3x3 box where this cell is positioned in the puzzle. Our input puzzle is always of size 9x9 but with varying number of cells filled for a particular problem.

To maintain the state of the puzzle which has been solved till now we are using the data structure Stack from the util library. ***To facilitate an easy condition check we have modified the actual Stack data structure to support and additional function called seek() which just returns the element at top of the stack but does not pop the element.***

We start with finding a cell whose value if 0 and then check incrementally for the validity of value within the range 1-9. If a given value passes all the restriction of row, column and the 3x3 matrix. We assign that value to that cell and push a list of [row, column, value] onto the stack and work on the next empty cell. While doing this if we reach a cell where none of the values in the range 1-9 satisfies the constrains then we *backtrack* and pop of the last successful cell assignment from the stack and reinitialize the cell to zero. Then we try to find the next valid value for this previous cell and continue in the same manner.

The Sudoku is considered solved if we have successfully assigned values in the range 1-9 to all the cell which had value 0 at the start of the execution.

Corresponding Functions: *SudokuBT(puzzle), IsValid(puzzle, row, column, value)*

Data for test runs using normal backtracking:

S.NO	Puzzle Size	Time Taken(Seconds)
1	5	0.096
2	10	0.164
3	20	0.032
4	30	0.101
5	40	0.083
6	45	0.028
7	50	0.29
8	60	0.39
9	70	0.045
10	80	0.029

Average Time taken = 0.1258 seconds, Variance = 0.01358196 seconds

MRV + FC + Arc Consistency: *(Python sudokuSolver.py MRV)*

In this approach we used the Minimum Remaining Values heuristic. To implement that we modified the input Sudoku puzzle before passing it to the Sudoku solver for MRV.

The modification done was to first change each cell of the puzzle into a list itself and then add a list of integer elements [1,2,3,4,5,6,7,8,9] to each cell which had initially a value zero. By doing this we are maintain a list of possible values at each cell which needs to be filled. This is done to maintain the state for Minimum Remaining Value. At each new assignment to a distinct empty cell we remove that possibility from other cells in that corresponding row column and 3x3 matrix.

At the same time we check if there is any unassigned cell which has its possible values list length set to zero as a result of this assignment if yes we return False which results into backtracking as we are using a recursive function to check for new unassigned cells. This implements our forward checking for the given puzzle in the clean() function.

Also with each assignment we check if there is any singleton cell as a result of this assignment. Singleton cell is defined as a cell which has a value which does not appears in any other cell in that row, column and 3x3 matrix. If we find such a cell we assign that value to that singleton cell and repeat the above mentioned process in a recursive way. This implements the Arc checking for the given Sudoku puzzle in the find singleton() function.

The solution keeps on finding and filling the unassigned cells by looking for cells with the minimum number of possibilities remaining and terminates when it can't find any new cells like these.

We pass the puzzle to each function as a deep copy of the current puzzle so that the changes made to the puzzle are only propagated if the assignments are valid otherwise the deep copy is discarded.

Corresponding Functions: *sudoku_solve(), clean(),find_singleton(),remove_possibilities()*

Data for runs:

S.NO	Puzzle Size	Time Taken(Seconds)
1	5	0.23
2	10	0.193
3	20	0.164
4	30	0.124
5	40	0.139
6	45	0.105
7	50	0.119
8	60	0.12
9	70	0.094
10	80	0.097

Average Time = 0.1385 Seconds, Variance = 0.00177505 Seconds.