Mathdoku – Assignment_4

Overview:

A mathdoku puzzle needs to be solved for the given pattern of data.

Classes:

Mathdoku : Includes the main functionality of creating the puzzle and solving it.

Data : Includes group name, outcome and operator.

Functions:

1. Public boolean loadPuzzle(BufferedReader stream) : reads the input stream given by the user, where the first part of input contains the pattern and grouping of the grid and the second part contains the group name, outcome and operator.
- Set<Character> set_data (): reads the arraylist containing the input data and finds the unique groups so that the groups that are not stated in the pattern are not given in the second part of input in loadPuzzle.

2. Public boolean readyToSolve() : checks if all the loaded inputs are sufficient for solving the puzzle.
- Char[][] temp() : stores the pattern given as input.
- Boolean checkGroup(char grp): Finds the group in the given pattern, checks its adjacent values and looks for a broken cage.

3. Public boolean solve(): solves the loaded puzzle.
- ArrayList<Integer> data (char grp): finds the group from the pattern and returns the position of the group from the matrix.
- SearchRow(int row,int value): checks if the assumed number is present in that particular row of the puzzle.
- SearchCol(int col,int value): checks if the assumed number is present in that particular column of the puzzle.
- CalGrid(ArrayList<Integer>,int row, int col, int value): Computes the values based on the operator and returns true if the outcome is matched for the given input.

4. Public String print() : prints the solved mathdoku in string format.
5. Public int choices(): returns the number of choices made in the program before finding the exact value.

Approach to the Problem/ Strategy:

- The puzzle is solved using backtracking algorithm where the hit and trail method is followed.
- Based on the size of the grid, each cell will be holding a value, less than the grid size.
- Before placing any value into the cell, the basic rule of mathdoku is checked if that number is already present in that particular row or column.
- Also, until the cell value is matching the outcome as per the operator, the pointer does not move to the next cell. If the cell is not able to assume any of the value, then the pointer is moved backward (one step to change the previous cell value).
- This continues until all the cells in the puzzle are filled satisfying the basic rule of mathdoku.
- The whole problem is broken into modules and after the implementation of each module, a unit testing is done to check if the requirement is satisfied.
- Also, required exception handling is done for appropriate functions.
- Code is completely modularized so that it will be easy to understand how exactly the functionality is implemented.

The same problem can be implemented using partitioning where, based on the outcome and operator and the grid size, each cell would be holding a set of possibilities that can fit into it.

A stack can be used for holding these possibilities and push, pop operations can be performed for adding the possibility and deleting the possibility satisfying the basic rule of mathdoku.

This also decreases the guesses but it would be difficult to maintain as it changes from operator to operator.

Efficiency:

- Since we are using the hit and trail method, the time taken for computation would be less when compared to the partition strategy.
- There would be less maintenance of the code as we do not need to keep track of the possibilities for a particular cell.
- Also, it uses less amount of memory since there is less data to be stored.

Test Cases:

Input Validation:

Public boolean loadPuzzle(BufferedReader stream) :

- Value is null, return false.
- Value is empty, return false.
- Spaces between the alphabets while giving the pattern to form the grid, return false.
- First part of input are not alphabets(a-z)(A-Z), return false.
- Second part of the input contains other than a-z/A-Z alphabets or 0-9 digits or +,-,*,/,= operators, return false.
- Second part of the input contains multiple alphabets, multiple digits, multiple operators, return false.

Boundary Cases:

Public boolean loadPuzzle(BufferedReader stream) :

- Load a single grid.
- Load a largest grid.

Public boolean readyToSolve():

- Check if the grid is ready to solve with a single group value.
- Check if the grid is ready to solve with a multiple group values.

Public boolean solve():

- Solve a 1x1 grid.
- Solve a largest grid.

Public String print():

- Print a 1x1 grid.
- Print a largest grid.

Control Flow:

Public boolean loadPuzzle(BufferedReader stream) :

- While loading the puzzle, the alphabet used in the pattern is not present while giving second part of input(group, outcome, operator) and vice versa.
- Grouped division and subtraction into more than 2 cells.

Public boolean readyToSolve()

- Broken patterns like “abaa” in the row and in the column.
- Group with no operator.
- Group with no outcome.
- Division and Subtraction having more than two grids.
- Same group given diagonally.
- Same operator for all the groups.
- Larger group, less target value.
- Smallest group, more target value.

Public boolean solve():

- Solve a puzzle with single operator value.
- Solve a puzzle with multiple operator values.
- Solve a puzzle with broken group values.

Data Flow:

- Call solve before loading the puzzle.
- Call solve before calling readyToSolve.
- Call solve after readytoSolve and before loading the puzzle.
- Call print before loading the puzzle.
- Call print before solving the puzzle.
- Call print after solving the puzzle.
- Call ReadyToSolve before loading the puzzle.
- Call ReadytoSolve after solving the puzzle.
- Call ReadyToSolve before printing the puzzle.
- Call ReadytoSolve after printing the puzzle.
- Call choices before solving the puzzle.
- Call choices before readyToSolve.
- Call choices before loading the puzzle.

Assumptions:

- Subtraction and Division occupy only two cells in the grid.
- Group, outcome and operator order is not changing.
- Square grid is formed based on the first input length.
- Values stores inside a cell are not negative.
- Every cell is categorized into a group.
- Cannot have broken groups.
- Cannot have “=” for all the groups.

Constraints:

- Cannot use existing library to solve the puzzle.
- Use of buffered reader while loading the puzzle.

References:

For understanding backtracking approach, https://cs.lmu.edu/~ray/notes/backtracking/

For handling regular expressions,
https://www.vogella.com/tutorials/JavaRegularExpressions/article.html

For understanding the puzzle, http://www.mathdoku.com/