CSCI 3901 Assignment 4

Anqi Chen, B00838586

Overview

This program is going to accept a puzzle and ultimately solve it with both the basic rules for Sodoku and the math computation result of each grouping. Given a square n x n grid, each cell in the grid belongs to one grouping. Each grouping is marked with a letter (case sensitive), and it is a connected set of 1 or more cells with an operator and a result of the operator. To meet the requirement for each grouping, applying the operator to all the values in one grouping should has the result assigned to the operator.

Structure and flow

MainUI.java

- Main for the program that prompts the user to enter the proper commends via the interface. It is more accessible for users and testers to interact with the program in a unified format. Also, when users enter something unacceptable, this main function can catch the invalid input exceptions and provide feedback on the console table.

Mathdoku.java

- **loadPuzzle**

  Read a puzzle in from the given stream of data and store data in the designed data structure. For this assignment, the input stream should only be provided within a readable file. For the first part, using map to store each mark letter and the corresponding locations of the points. First part should end properly when finish loading the last point in the last row. For the second part, store each line into a defined piece structure with its operator, result number, and the corresponding locations of the points. Return true if the puzzle is read. Return false if some other error happened.

- **readyToSolve**

  Determine whether or not we have all the information needed to be able to try and solve the puzzle.

Occasions that the puzzle is obviously unsolvable:

- The puzzle is not in n×n size.
- Grouping letters that appeared in part 1 do not appear in part 2.
- Grouping letters that not appeared in part 1 appear in part 2.
- The grouping letter that only appears once in part 1, the corresponding operator in part 2 is not '='.
- Data in part 1 do not only consist of letters.
- One of the groupings does not have a constraint (operation or result).
- One of the groupings does not have a valid data type for each area (letter, number and operator).

Return false if there is something missing or amiss with the puzzle that is obvious (so not needing to solve the puzzle) and that would prevent a solution.

- **Solve**

Simulate the filling and matching of each cell and piece from the point (0,0) to find a solution to the puzzle using recursions. The detailed process will be explained later in key algorithms part. The solution of puzzle is stored within the class, using toBuff method to put a solution into the StringBuffer buff. If there are more than one solution, the use of solcopy method can guarantee all the valid solutions can be retrieved. Return true if you solved the puzzle and false if you could not solve the puzzle with the given set of words.

- **Print**

Print the solutions to the returned string object. In solve method, the result was already stored, so in this step, just convert buff into string to print out.

- **Choices**

Return the number of guesses that your program had to make and later undo while solving the puzzle. The choice time is already recorded while solving the puzzle in solve method. Every time a candidate is fail to finish the matching, the choice time is added up by 1, when the match_piece(cand, matchset[s][t].piece) returns false.

Key Algorithms

The main idea is to fill the whole puzzle properly from the first point at (0,0) to the last one at (n,n). The process can be divided into two main parts, the first step is the same as solving a sodoku and the second part is to check the math computations. For each cell, the status can record the available values for itself and also the availability from the same row and column directions. For the first part, the core is to inherit the column and row availabilities from other points to narrow down the possible candidates and fill the puzzle recursively. For the math operation part, the core is only when every cell in the grouping is filled, then check if applying the operations among all the cells, the computation result can meet the conditions.

Assumptions

1. If there are multiple possible results for the puzzle, the results will be printed in a consisting string with no special separator.
2. Once readyToSolve return false, the program will automatically quit.
3. Every input line in part 1 should be letters without space, every input line in part 2 should be letter, space, number, space, and operator.
4. The character for each cell grouping is case sensitive.

Test cases

Input Validation

The stream for loadPuzzle is null

The file path for loadPuzzle is nonexistent

Boundary Cases

Load a puzzle with only one cell

Load a puzzle in 15×15 size

Check if a puzzle with only one cell is ready to solve

Check if a puzzle in 15×15 size is ready to solve

Solve a puzzle with only one cell

Solve a puzzle in 15×15 size

Print the solution string for the one-cell puzzle

Print the solution string for the 15×15 puzzle

Print the choice time for the one-cell puzzle

Print the total choice time for the 15×15 puzzle

Control Flow

Load a puzzle that is not in n×n size

Check if a puzzle is ready to solve in any one of the obvious unsolvable conditions

Call solve if the puzzle is unsolvable

Call solve if the puzzle has more than one solution

Call print when there's no solution

Call print when there's more than one solution

Data Flow

Call readyToSolve before loadPuzzle

Call solve before calling readyToSolve

Call choices before calling print

Call solve twice in a row, then call print and choices

Call load, ready, solve, print for another puzzle after calling them for current puzzle

Limitations

1. The loadPuzzle method can only load data from a file.
2. If the input cannot be stored in designed data structure, the loadPuzzle may trigger the program to crash instead of returning false.
3. If the input is not perfect, the readyToSolve method is easy to return false. It is not robust enough to carry out common typos.
4. The data flow is not flexible for this program.
5. Many methods are static, calling a method twice in a row may trigger some errors.

6. If calling solve more than once, the result of print method will represent as a repeating solution string.

References

KenKen Puzzle Solver using Backtracking Algorithm (n.d.). Retrieved from: https://docplayer.net/45439751-Kenken-puzzle-solver-using-backtracking-algorithm.html

Kenken solver in java (n.d.). Retrieved from: http://freesourcecode.net/javaprojects/72583/kenken-solver#.Xl2qupP0kWo

Kulkarni, D. (2013, August 1). Enjoying Math: Learning Problem Solving with KenKen Puzzles. Wayback Machine. https://web.archive.org/web/20130801080339/

(N.d.). Retrieved from http://www.cs.ucf.edu/~dmarino/progcontests/cop4516/samplecode/kenken.java