Borovlev Petr

Sudoku Puzzle Generator and Solver

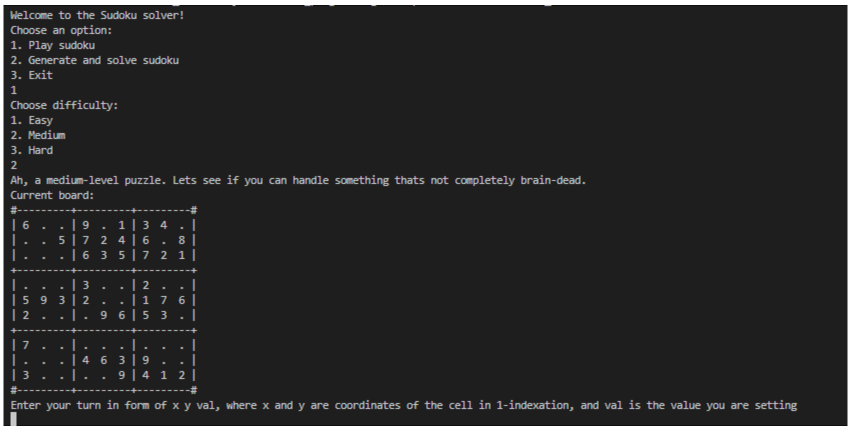
What the task was?

My project was to implement sudoku generator and solver. I had to:

- Make console (or other user) interface
- Implement the solver that can solve any valid puzzle
- Create generator that generates puzzles, allowing to vary difficulty
- Analyze the performance and make it as efficient as possible
- Handle invalid inputs
- Implement property-based tests

Architecture of my solution:

- src/Board.hs:
  - Type Board is defined
  - Several helper functions that are used throughout whole project are implemented here
  - Also function `printBoard` is defined, which nicely prints the grid into the console
- src/Solver.hs:
  - Main algorithm of solving the sudoku is implemented here. It works using backtracking
  - Some other helpful functions are implemented here:
    - `findEmpty`: returns the position of the first empty cell, if there are none then Nothing is returned
    - `isValid`: returns Bool representing whether specific move is valid or not
- src/Generator.hs:
  The logic of generation is to at first generate a full valid board, and then removing x random cells, where x depends on the difficulty. I supposed that easy puzzle misses 30 cells, for middle this number is equal to 40, and for hard level it is 50.
  - So there are functions `fillRandomly` and `generateSolved` that generate a full grid
  - And there is `removeRandomly` function that takes board and number of cells to be removed and removes them
  - Lastly, for each difficulty there is `getXXXSudoku` where instead of `XXX` is the level of difficulty, like `getEasySudoku`.
- src/SampleBoards.hs:
  Basically it is just a file I used while testing. It contains several hardcoded boards.
- app/Main.hs:
  User interface is implemented in it. I could've done it with any user interface, so I've chosen REPL style console interface:



```
Welcome to the Sudoku solver!
Choose an option:
1. Play sudoku
2. Generate and solve sudoku
3. Exit
1
Choose difficulty:
1. Easy
2. Medium
3. Hard
2
Ah, a medium-level puzzle. Lets see if you can handle something thats not completely brain-dead.
Current board:
#---------+---------+---------#
| 6  .  . | 9  .  1 | 3  4  . |
| .  .  5 | 7  2  4 | 6  .  8 |
| .  .  . | 6  3  5 | 7  2  1 |
+---------+---------+---------+
| .  .  . | 3  .  . | 2  .  . |
| 5  9  3 | 2  .  . | 1  7  6 |
| 2  .  . | .  9  6 | 5  3  . |
+---------+---------+---------+
| 7  .  . | .  .  . | .  .  . |
| .  .  . | 4  6  3 | 9  .  . |
| 3  .  . | .  .  9 | 4  1  2 |
#---------+---------+---------#
Enter your turn in form of x y val, where x and y are coordinates of the cell in 1-indexation, and val is the value you are setting
```

Why certain architecture decisions were done

There are not many choices of how to solve sudoku, and main one is backtracking. It is not really hard to implement, and also for an average puzzle really efficient since there are quite a few numbers that a suitable for each particular cell, and therefore the tree of recursion will not have a lot of deep paths. Moreover it will not be too broad, again, due to limited amount of numbers suitable for cell.

It made sense to separate different logical parts into different files so thats why I have file for Generator and Solver separately. Same as main logic of interaction with user is implemented in Main.hs file, and Board.hs for basic stuff related to the board.

Why certain libraries were chosen

- `vector` - this was used to be able to store the board in a data structure that allows efficient random access to data. At first I was using `Data.Array` but changed it to Vector because of the speed (I'll take about it later)
- `random` and `random-shuffle` - those are the 2 libraries I use when randomly generating the board. First one is used to generated seeds that are passed into `shuffle'` function from `random-shuffle`
- `HUnit` - used for tests of course

Investigation of the performance

- As mentioned above I changed Array into Vector, because it is more efficient. Array copies all elements when it comes to changing one element. But Vector doesn't do that, due to its internal implementation it *usually* copies only a few elements. Well, the worst case scenario is still O(N) but it doesn't happen that often, since the changes we make are distributed evenly throughout the 2d vector, because of random.