1. Personal information

- Title: Killer Sudoku
- Student's name: Krzysztof Modrzyński
- Student number: 100586275
- Degree program: Bachelor of Science in Data Science
- Year of studies: 2022-2027
- Date: 15th of April 2023

2. General description

Sudokus are popular numerical puzzles that are constantly published in many newspapers, magazines and even have their own separate Sudoku magazines. The basic version of Sudoku has a 9*9 grid, in which player needs to fill in each square in the grid with a number from 1 to 9 so that:

a) in each row this number appears only once,

b) in each correspondingly column the number appears only once,

c) in each 3*3 sub-grid the number occurs only once.

Killer Sudoku differs from basic sudoku in that it does not contain any pre-filled squares. Instead, the grid of different sub-areas consists of 2-4 squares and only the sum of the numbers in the area is given; in other words, a sub-area can contain several different combinations of numbers. Otherwise, the solution requirements for placing numbers in a 9*9 grid are the same as in basic sudoku. The shape and placement of the sub-areas in the grid is not restricted in any way.

I completed the project with the demanding level of difficulty.

3. User interface

The program is launched by running the Main object in Sudoku.scala. After running it, the window containing the Killer Sudoku game opens. You can load and save board of Killer Sudoku games, as well as solve them.

Loading the board is done by clicking the LOAD button and then selecting a file containing a board.

Saving the board can be done by clicking the SAVE button and then selecting a place to save the file, as well as its name. The format of the saved file is JSON.

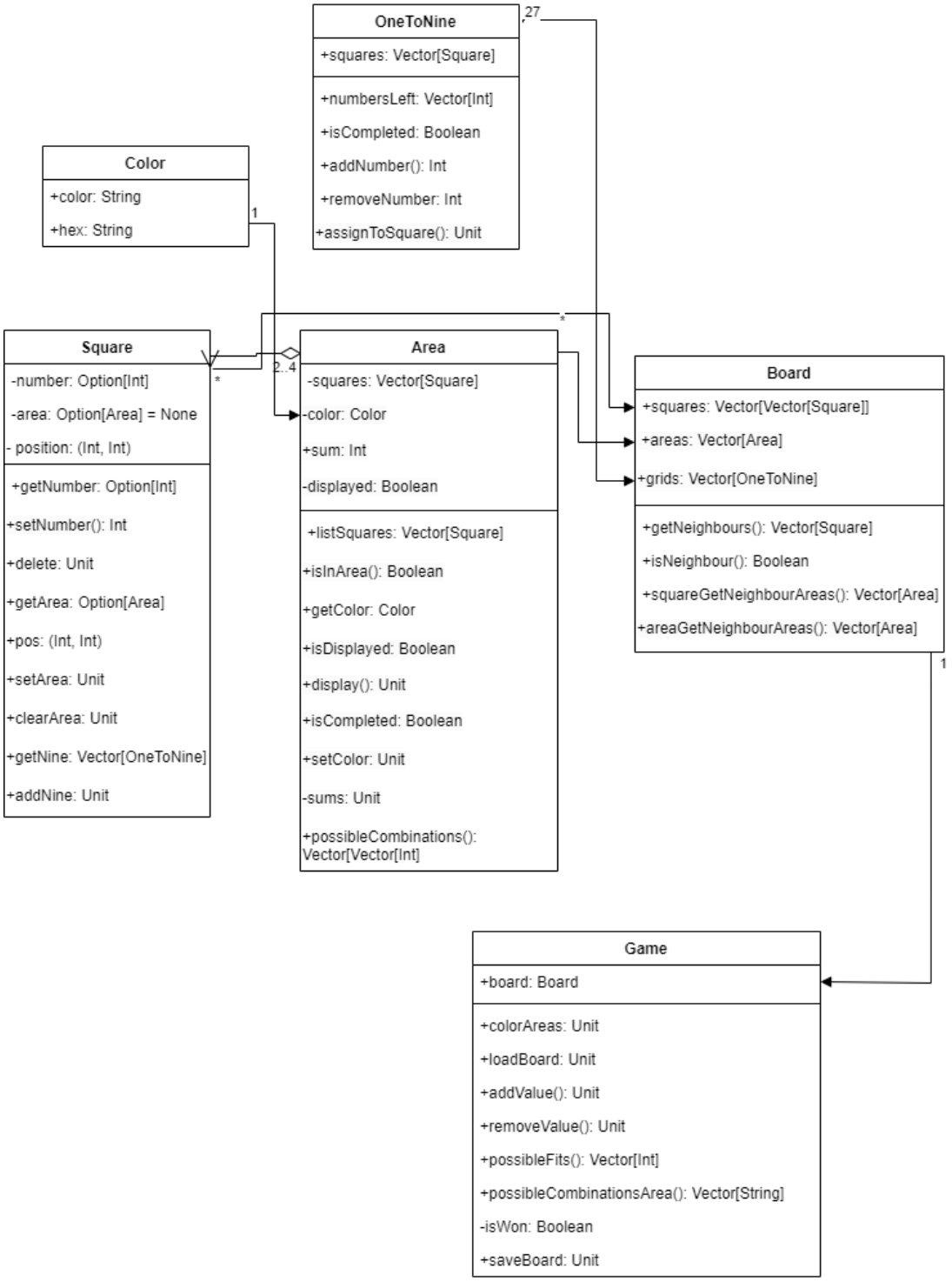
By hovering over a square, possible numbers that are allowed by the rules to be entered in that square are highlighted.

If the user clicks on the square, possible combinations of numbers that can create a given area (sum to number) are displayed.

4. Program structure

I split the project into distinct components of the Killer Sudoku game:

- Graphical user interface (Sudoku.scala)
- Game class (Game.scala) - responsible for state of the game and internal functions that allow users to influence board state.
- Board class (Board.scala) – responsible for keeping track of the state of the board, keeps information about individual squares, areas and grids.
- Area class (Area.scala) – structure that keeps information about areas and functions regarding them.
- Grid class (OneToNine.scala) – structure that keeps information about rows, columns, 3x3 grids and functions regarding them.
- Square class (Square.scala) - structure that keeps information about single square
- File containing Colour objects (Color.scala) – file defining colour objects used to colour areas on the board



GUI has a Game object to keep track of game status and just displays it.

5. Algorithms

- Greedy colouring algorithm:
  First, this algorithm assigns a colour to the first area. Then, for each area it loops through its neighbours to see which colours have been already assigned and assigns the first free one. If all have been assigned, a new colour is added.
- Algorithm showing possible sums:
  Recursive function producing all possible results of different integers summing to the sum of selected sub-area. Returns all vectors of numbers that sum up to the given sum

6. Data Structures

- Graph – only for executing the colouring algorithm. Program treats areas as vertices and shows with edges which areas are bordering each other.
- Vectors – to store information about squares that belong to an area, for the board to remember the areas it consists of and as returns of certain functions.
- Buffers – to store possible combinations of integers that create an Area
- 2 dimensional tables (Vector of vectors) – to store information about squares in the board and displaying the table in the GUI.

7. Files and Internet access

The program saves and reads the board to and from a JSON file. Already filled squares are stored as a Vector of position and number inputted in “filledSquares”. File also stores information about the areas: their sum and squares that they are composed of in “areas”.

8. Testing

I did the unit testing of functions in classes Area, OneToNine and Board. They are tested in their separate testing classes: areaSpec, oneToNineSpec and boardSpec. In boardSpec also tested compatibility of classes Square, Area, Board. The whole game logic is tested in gameSpec which tests the Game class.

The functions tested are:

- areaSpec:
  - isInArea – checks if function return correct squares from area
  - possibleCombinations – checks if function returns correct number of combinations and if those combinations are correct
- boardSpec:
  - getNeighbours – check if square returns its neighbours correctly
  - isNeighbour – check if the square can identify its neighbours correctly
  - squareGetNeighbouringAreas – checks if the function returns correct neighbouring areas given a square and if the function returns an empty vectors when squares are not assign to areas
  - areaGetNEighbouringAreas – checks if the function returns correct neighbouring areas given a area
- oneToNineSpec:
  - given a series of commands OneToNine should return correct state of completion
- gameSpec:
  - loadBoard – check if the function creates a right board from the given file
  - addValue – check if function adds correct value to the right square
  - removeValue – check if function removes value from the right square
  - possibleFits – check if function returns correct numbers
  - possibleCombinationsArea – check if function returns correct combinations of integers summing up to sum of the area
  - colorAreas – check if neighbouring areas do not have the same colour

I wrote classes and their corresponding test classes almost at the same time. The program passes all the tests and I think I have tested all the functions that were important to test except from saveBoard function which I tested manually along with the GUI.

9. Known bugs and missing features

I have no knowledge of any bugs or missing features. I have not found any throughout my testing and I think my program's functionality matches that of task description.

10. 3 best sides and 3 weaknesses

I am quite happy with my test classes as I think they are well made and test all the important features of my program. I am also satisfied with how clean my code looks for

the most part. Moreover, I think that the whole game mechanic is implemented quite nicely.

I am mostly displeased with the GUI side of the project. It is not as concise and good looking as I would like it to be. Probably the GUI could be implemented more efficiently in terms of space and time complexity.

11.Deviations from the plan, realized process and schedule

Schedule:

- Implementing classes Square, Area, OneToNine (1 week)
- Implementing Board along with Game and tests (2 weeks)
- Implementing GUI (2 weeks)
- Adjusting GUI and Game to be more compatible with each other, fixing errors, further testing (2 weeks)

This was my schedule before starting the project. I did almost everything as I planned with small exceptions. I had problems with testing classes as they would not work no matter what I did, so I had to postpone writing them slightly as I tried to figure out what was wrong with them. This also slightly delayed the GUI part of the project but I still managed to fit in the time frames that I set.

However, I think I should have started implementing the GUI earlier as I knew very little about scalafx. Except for that I think I chose the right order of implementation of the project.

12.Final evaluation

I think that the final project is quite well done. I have some slight issues with the GUI but I consider this to be a decent program, especially that it was my first time working with scalafx.

Definitely the program could be improved, mostly the GUI, but also I think that loadBoard function in Game class could be done more concisely.

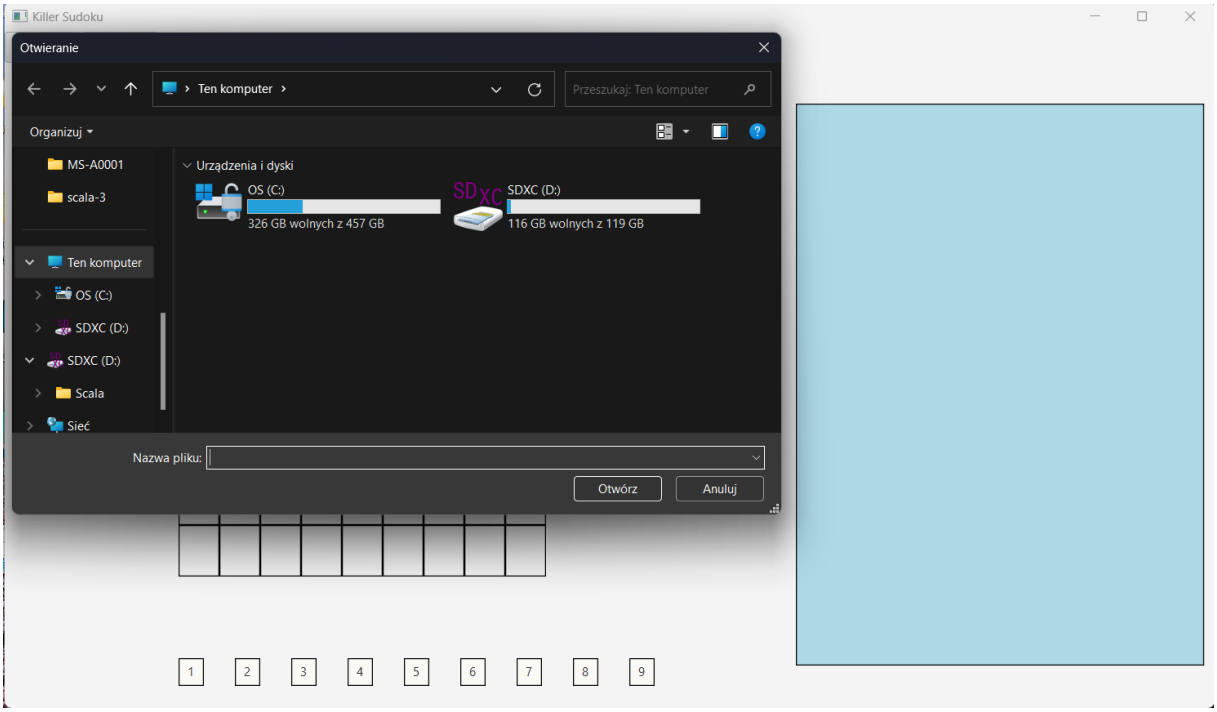
I also think that finding combinations of integers that fit into the area could have been done better as now it finds all combinations with repetitions and integers bigger than 9 and then I filter through those solutions. So I am sure that this could be done directly in searching for solutions without filters.

I am quite satisfied with data structure and class structure in the game functionality but there is probably room for improvement. On the other hand, in the GUI, the data structure could definitely be built better as I forgot about adding the labels displaying sums of the

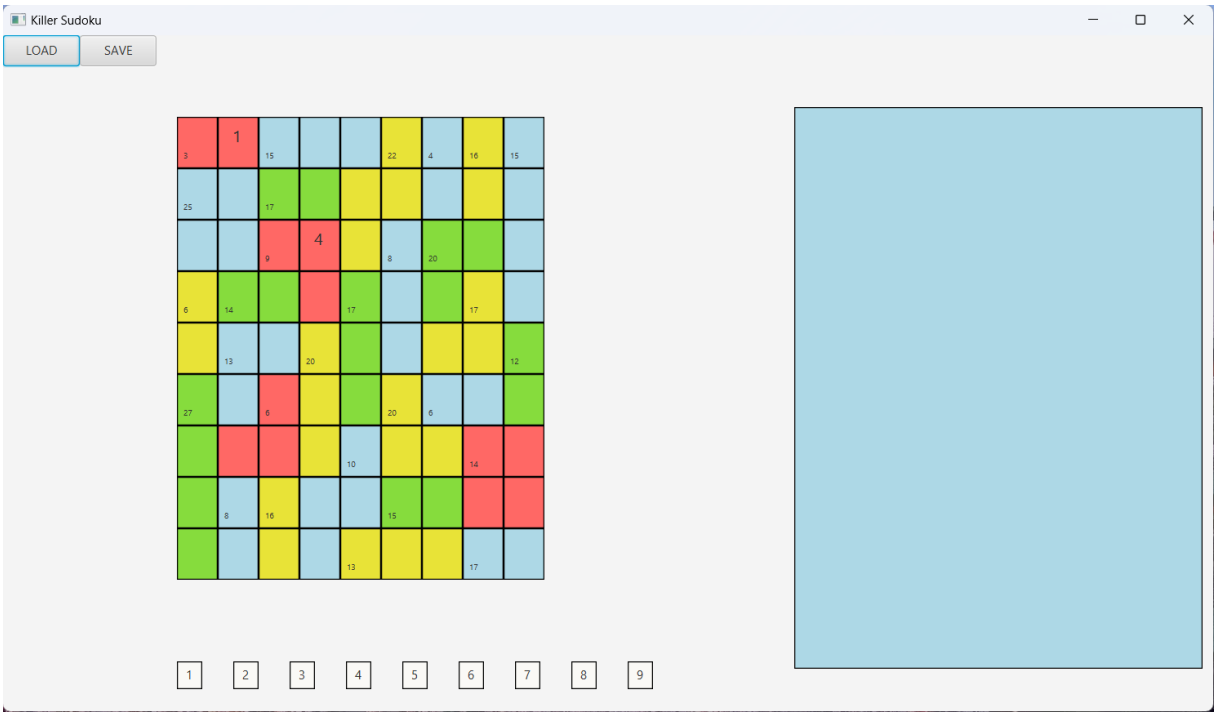
areas on the board. This forced me to fit into an already done program which left me with very little flexibility. Therefore, I consider the game logic is suitable for changes and extensions but not so much the GUI.

I would definitely try doing the GUI differently if I have started the project again from the beginning as now I have better knowledge of scalafx and the problem itself. So I could definitely improve the GUI by a lot. I would also make changes to possibleCombinations() in Area class as I described above. Outside of that I am quite happy with the outcome of the project.

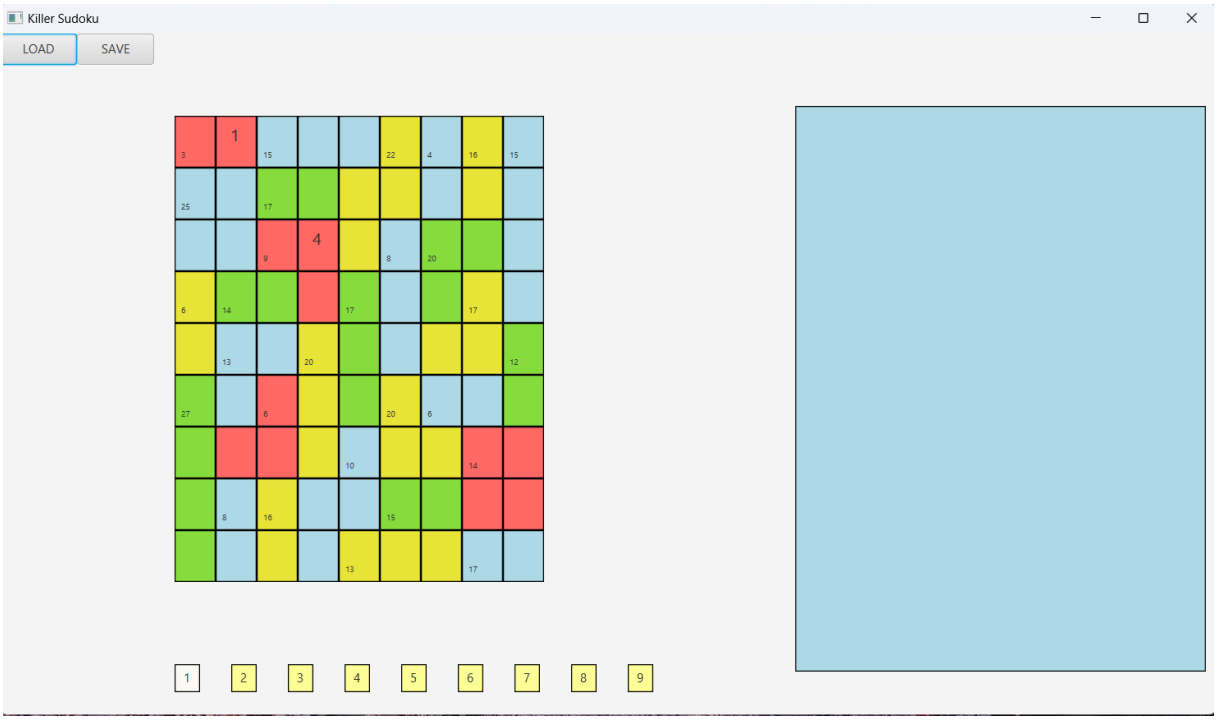
13.Execution examples



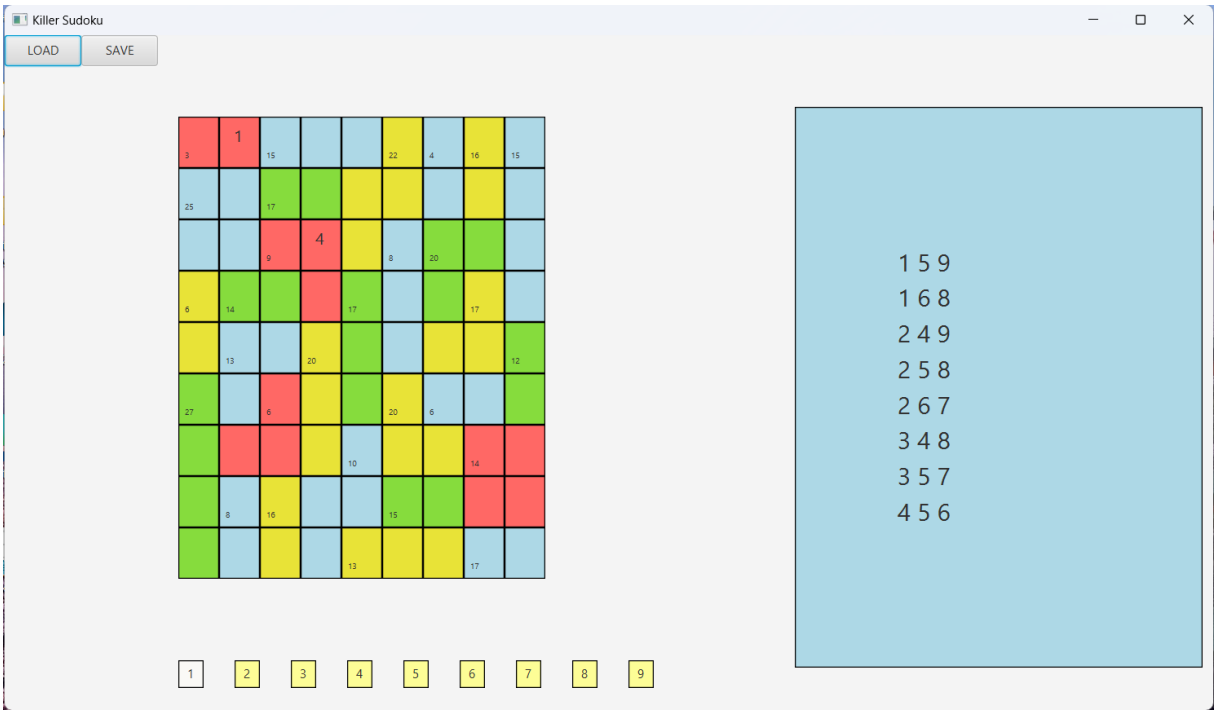
Program after clicking LOAD button.



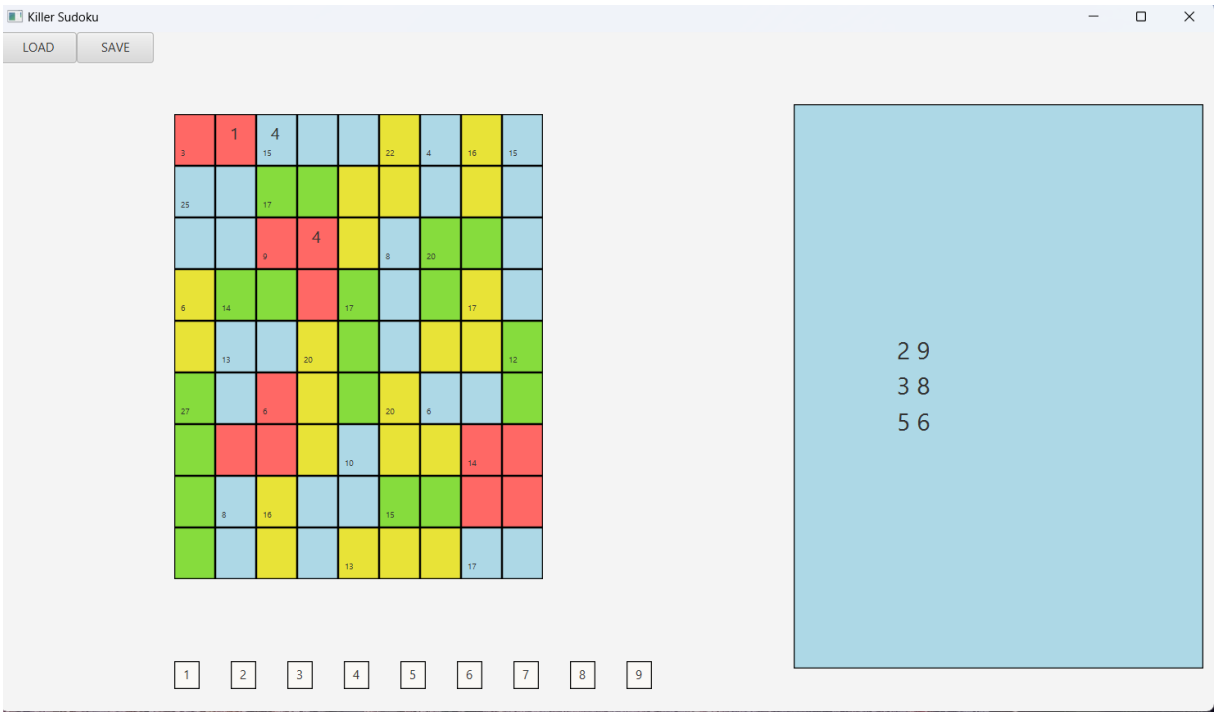
Program after selecting file with an example board and opening it.



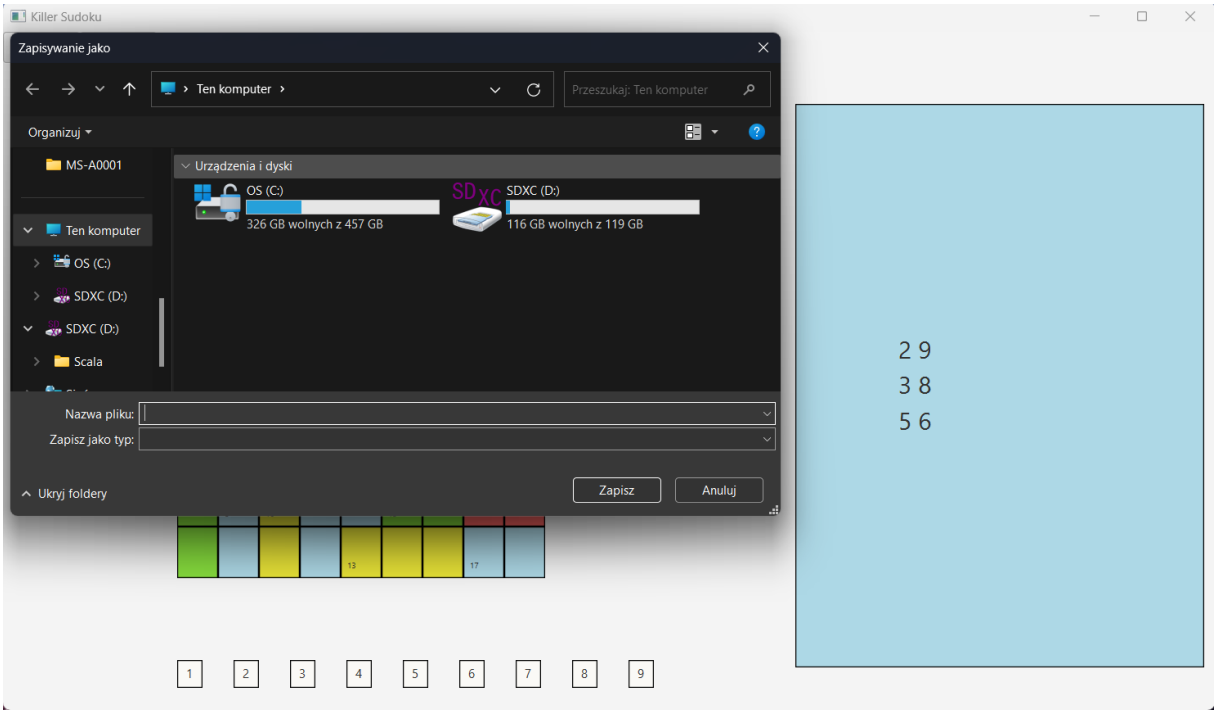
Program when hovering over one of the tiles (unfortunately you cannot see a cursor in the screenshot).



Program after clicking one of the tiles.



Program after clicking the box with number 4 after selecting a tile to input it in.



Program after clicking SAVE button, after selecting location and naming the file, board will be saved there.