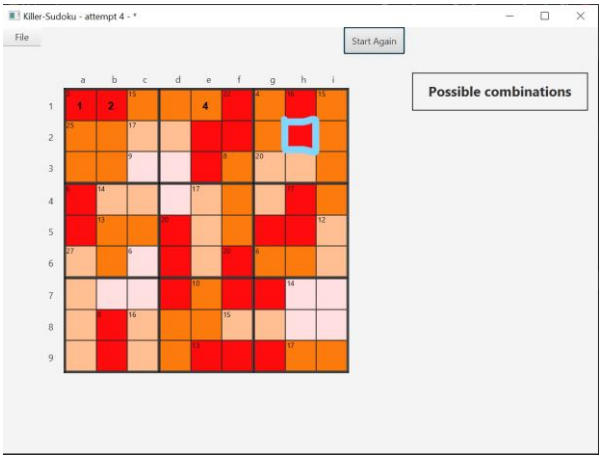
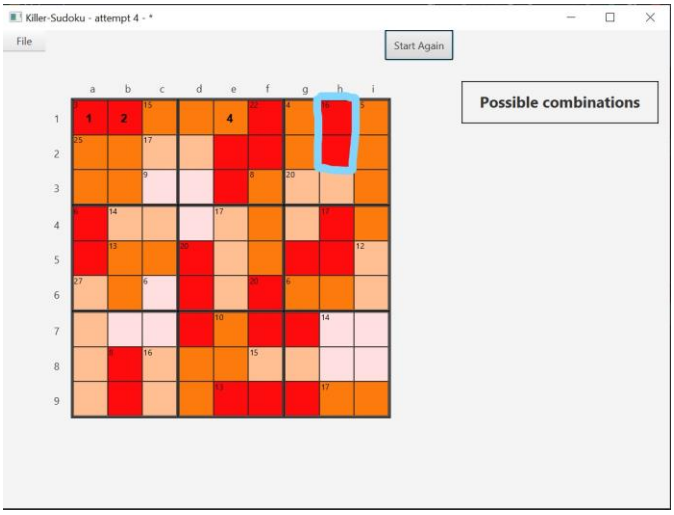
Killer Sudoku demanding task

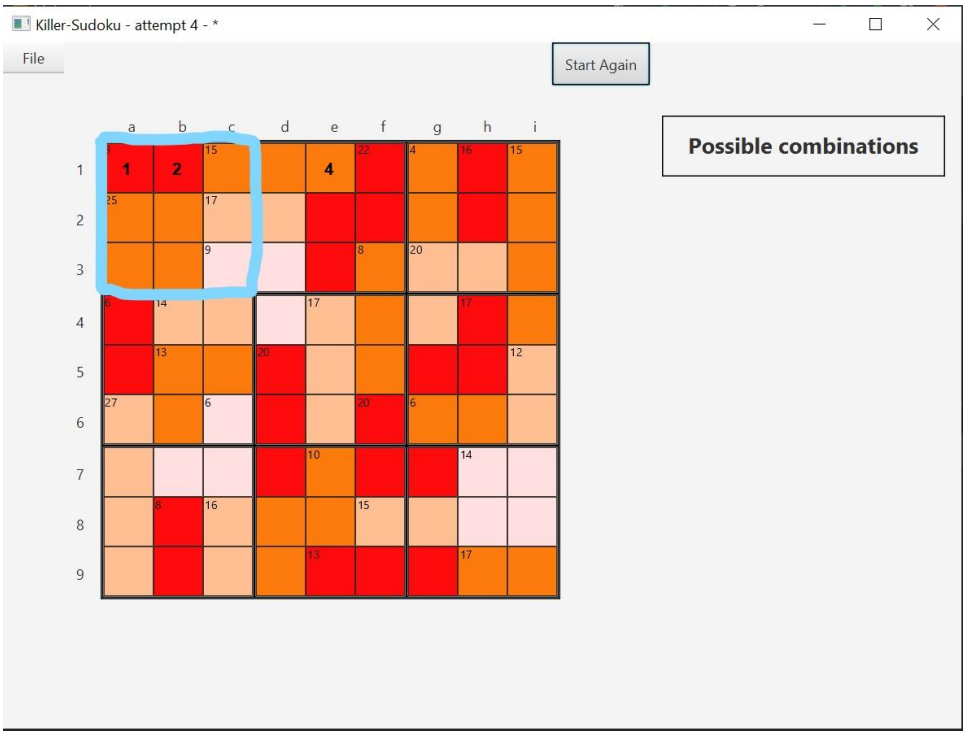
Date: 24.04.2023



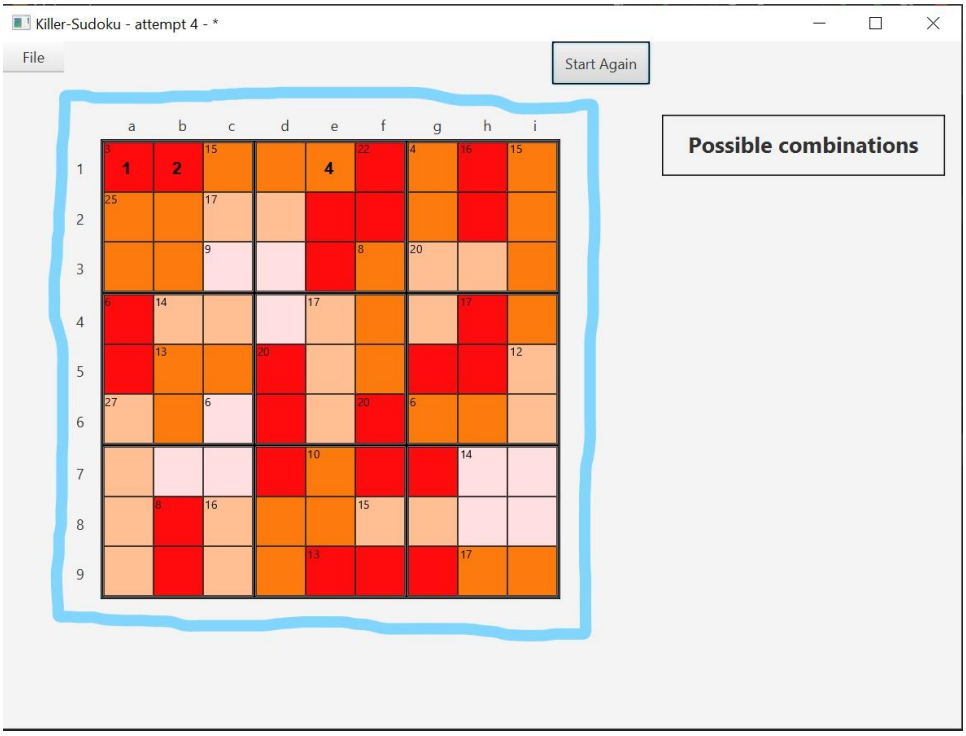
In the picture above the area highlighted with the light blue color is called Tile in my program.



In the picture above the area highlighted with the light blue color is called Subarea in my program. (It might contain from 2 to 4 tiles)



This area is called Square. (Contains 3 x 3 = 9 tiles)

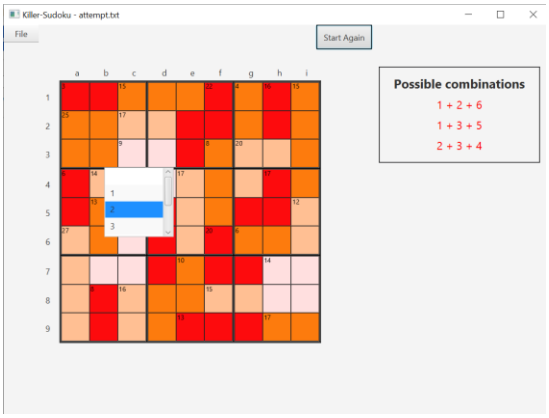


This area is called Puzzleboard (contains 9 x 9 = 81 tiles).

2. General description

The idea of the Killer Sudoku program is to help a player to solve a puzzle. It won't solve the puzzle automatically, but instead it will provide some useful tools to the player, which will help her/him to solve the puzzle. Let's take a closer look at the functionality behind the program.

While solving the puzzle the program with the GUI will allow the user to select the digit, which could be placed in the tile on the Sudoku board. User should only click on the tile, where she/he wants to place a new digit and possible digits will appear as drop-down menu (list view) as shown in the GUI picture below.



(In the picture the cursor is on list view item "2").

When the user clicks on some tile, the program will show all possible digits (candidates in the program) that can be placed in the square according to the 9 x 9 standard Sudoku rules and the sum of the area which the square belongs to (an area is from 2 to 4 tiles). If the player later decides to change the digit in some tile to another one it will be also possible, and the program will automatically update the list of the possible digits in other tiles. However, this action is only possible if there are candidates left. Also, when the user moves the cursor over a candidate number without clicking it, the program highlights the squares in the grid (changes border color to yellow) that already have the number in question.

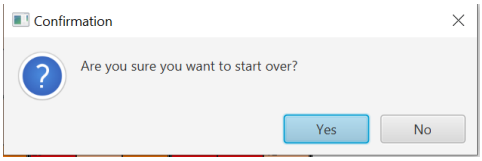
I decided not to implement the following feature, because I thought that it will be harder for the user to understand, which tiles belong to which sub-area after tile's color changing:

When all instances of a number 1-9 have been placed in the grid, the corresponding candidate number's square changes color and becomes passive.

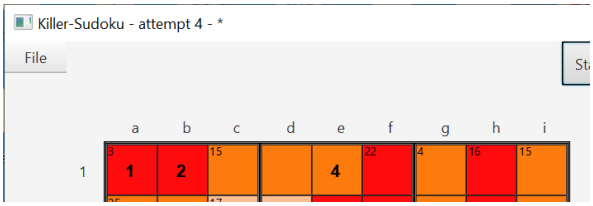
When the program starts running the GUI window will show the File menu in the top left corner, possible combinations title in vertical box and "start again"-button. A new Sudoku task can be made by providing a new file, which contains all the information about subareas and tiles, such as sum and placement on the board or by reading saved file. If user wants to download a new Killer Sudoku puzzle, it can easily be done by clicking "File"-menu and selecting "Add file"-button. A File Chooser will appear asking to select the .txt file that user wants to open. After providing this information, the program will display a board and, if the user hovers cursor over one of the tiles on the board, possible combinations will appear. The user can also save a file and open it later by using "Save"-menu item or "Save as"-menu item. If the file was saved earlier,

then the user doesn't need to give a new name of the file. It can be done just by clicking on "Save"-menu item and the file will be saved to the same place as specified earlier.

The program won't store all the previous moves of the player as was mentioned earlier in the General plan. Although, the user can open saved files. If at some point the player wants to start from scratch, it will be also possible, just by clicking the button in the GUI "start again". In case the user accidentally clicks the button, the alert will appear:



If the board was modified (some digits were placed or removed from tile) then the GUI window will show asterisk '*' in the title of the window. The asterisk will disappear as soon as the user saves the file.



3. User interface

The program starts running by clicking the "Run" arrow button in the IntelliJ IDEA. When the building is finished the window will appear, which will contain file menu, "start again"-button and vertical box with the title "Possible combinations". The user now can click on the" File"-menu and the drop-down menu should appear. It will contain three buttons (menu items) "New file", "Save" and "Save as". At this point "Save" and "Save as" menu items should be disabled, because there is no board given yet.

The user can open a new file by clicking "New file"-menu item. After this action, the file chooser will appear, and the user should provide some file with .txt extension, which contains the information about the board. The application checks for multiple types of configuration file format violations. If the file is corrupted, for example, some information is missing or there are duplicates, the alert should be displayed with the error message. Otherwise, after opening the file, the board, which was specified in the file should appear and the sub-areas should be colored.

Now, when the user hovers cursor over one of the tiles, the possible combinations should appear as a list on the right-hand side of the screen from the user perspective. When the user clicks on one of the tiles on the board, a drop-down list view should be displayed, and the user can select the candidate that she/he wants to place in that tile. If one of the candidates is already on the board in other tiles, then when the user hovers cursor over that candidate number in the list view, the borders of the tiles containing the same candidate should change from black to yellow. At

some point the user may encounter the situation when there are no candidates to be placed, but there are still free tiles left. In this scenario, the alert is displayed, which will inform the user that there are no candidates left. The user can either remove some placed numbers from other tiles by clicking on them and selecting the first item, which is empty item, or click “Start again”-button. In the latter case, the alert will appear, and the user should click on “Yes” option, if she/he wants to start over. After this action all the numbers placed will disappear and the board should look as if it was open just now.

At some point, the user may want to save the current state of board (either completed or half-done). This action can be done by clicking on the “File”-menu and then “Save” and “Save as”-menu items should be chosen. After clicking one of them the file chooser will open, and the user can select to which directory she/he wants to save the file. After the file is saved, the title of the GUI window should also be changed to the name of the recently saved file. After modifying the board, the user now can just click the “Save”-menu item if one wants to rewrite the previous version with the new one. It should be also possible to save the file using a different name and in other directory, if the user doesn’t want to rewrite the previous files.

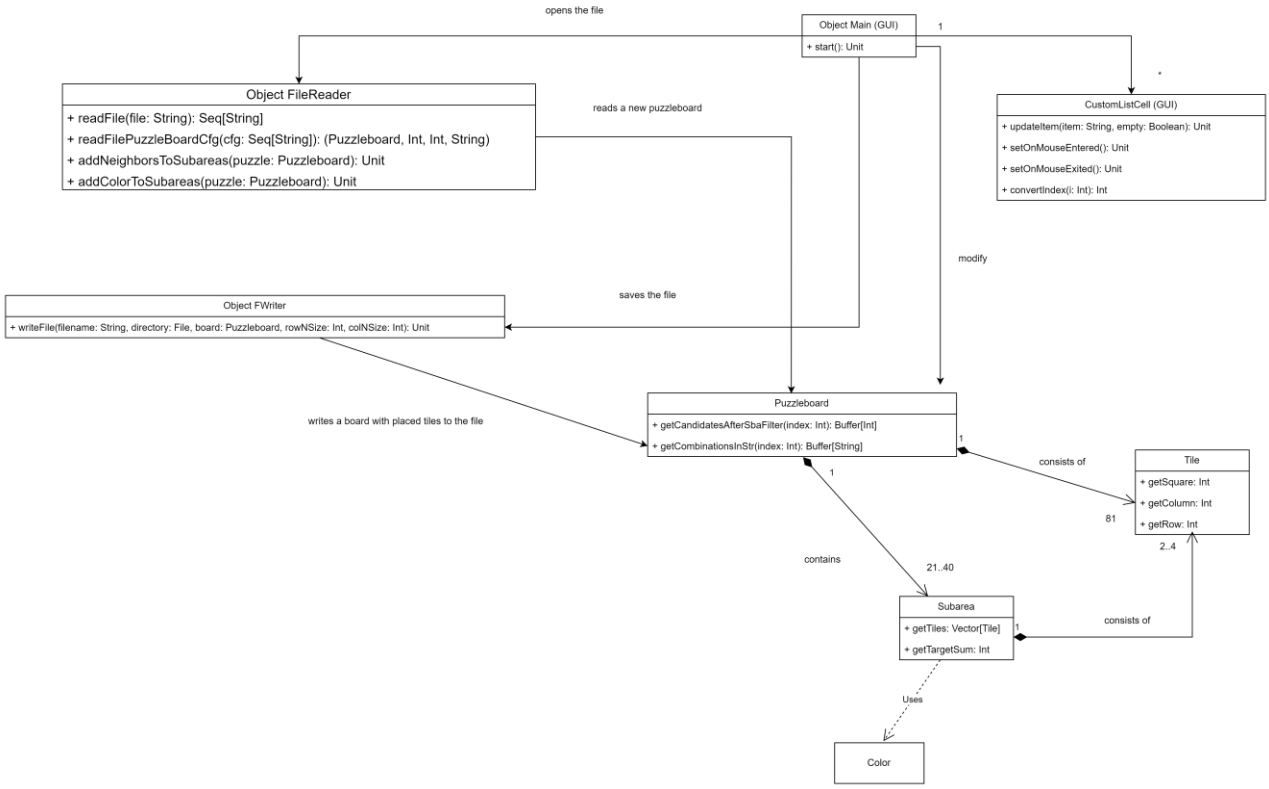
After saving a file the user can close the application and return to the assignment later. The user can open the saved file and the board should be displayed according to the content of the file. If there are unsaved changes in the file and the user wants to close the application, the popup window will appear asking to save file, if the user declines, then the changes will be lost. The user can also open a new file with another board by clicking on “New file”-menu item. Unfortunately, it will open it even if there are unsaved changes, and the progress will be lost if it wasn’t saved earlier. This a known issue to be fixed in a next version of the application.

The user can figure out if there are unsaved changes by looking at the title of the window. If it has an asterisk, then there are unsaved changes. This could be helpful, when cracking a nut for a long time and one wants to be sure that progress is saved.

4. Program structure

The high level UML diagram of the project is presented below.

h



What follows is the description of major classes comprising the diagram above.

Tile(column: Int, row: Int, square: Int):

Tile-class is a simple class that describes a single tile on the Sudoku board. It keeps information about its coordinates (row, column), neighbors (tiles in the four main directions) and square it belongs to. Square is 3 x 3 square on the board and sub-area is an area that consists of 2 to 4 tiles. Tile-instance also has a variable inUse, which tells whether the tile was already placed on the board. Some of the instances have also targetSum, which is the sum of the sub-area. In addition, every Tile knows its top left corner coordinates in the GUI. This class serves as a data container and doesn't have any algorithms in it.

Subarea(targetSum: Int, tiles: Vector[Tile], tileWithTargetSum: Tile):

The Subarea class's instance describes a single-colored area that has from 2 to 4 tiles in it. This class serves as a data container and doesn't have any algorithms in it. It has only methods that give some of the parameters. After the file is read and Puzzleboard is initialized all of the class Subarea instances should contain the information of their neighbors (other sub-areas), tiles which are placed in this sub-area, and target sum, which is the configured sum of the tiles. For coloring I use Greedy algorithm, which is described later in the Algorithms section.

Puzzleboard(allTiles: Vector[Tile], subareas: Vector[Subarea]):

This class describes a 9 x 9 – Killer Sudoku board, which consists of 9 squares and 9 x 9 = 81 tiles. It knows all the tiles and sub-areas that are placed on the board. Let's have a closer look at its methods.

GetCandidatesAfterSbaFilter(index: Int): Buffer[Int]

This method is used for calculating the possible digits that can be placed in the tile with the given tile index. Candidates are first calculated using only standard Sudoku rules, which state that the same number shouldn't be placed in the same row, column or square.

Returns candidates after filtering them depending on the number of tiles in the sub-area, already placed digits in sub-area and the sum of the sub-area.

GetCombinationsInStr(index: Int): Buffer[String]

Finds combinations that can be applied to the sub-area, depending on how many digits are already placed in this sub-area and how many tiles are in this sub-area. Buffer consists of Vector[Int], where Vector[Int] describes a single possible combination.

Returns combinations as strings, for example "2 + 4 + 6". This method will be used in the Main object in GUI, when displaying possible combinations on the right side of the screen.

FileReader-object

This object is used for reading a configuration file. FileReader has many methods and some of them are private. Below is a description of public methods.

ReadFile(file: String): Seq[String]

Reads all lines in the given file and returns a sequence, which contains all those lines.

ReadFilePuzzleBoardCfg(cfg: Seq[String]): (Puzzleboard, Int, Int, String)

Takes return value of the previous method as a parameter and tries to create a Puzzleboard according to the configuration information. If some information is absent, then the method throws an exception with some error message. This method returns a new Puzzleboard, number of rows, number of columns and the title of the file.

In the technical plan that was written earlier I mentioned that the program will also contain the history of the moves that were made by the user. However, I realized that it cannot be done without using some database, which will allow me to maintain the user's moves and return previous moves. Unfortunately, I'm not yet quite familiar with databases and queries and I also wasn't quite sure how to connect the program with them, so I decided not to implement that functionality. I also removed time and date tags from the file configuration, because I was planning to use them for previous moves in the first place. After I gave up the idea of storing the previous moves I got rid of date and time tags.

FWriter-object

This object is used to write the Puzzleboard information to the file, so that the user can save the file and open it later. This object has one public method and few private methods. I will present only public method.

WriteFile(filename: String, directory: File, board: Puzzleboard, rowNSize: Int, colNSize: Int): Unit

Creates a new file in the given directory and writes all the information that board contains (Subarea, colsize, rowsize, title, and placed digits). If this file already has some text, it will just remove the previous one and write a new one.

Gui

For GUI I created one class and one object.

Main-object

This object represents almost every functionality that the GUI has. It has one public method start(): Unit, which contains a big number of other methods which are used for drawing and event handling. Some of the methods are responsible for creating a sudoku board. For this task I used GridPane class. I found it quite useful, because adding nodes was done in a quite convenient way.

CustomListCell(tiles: Vector[Rectangle], board: Puzzleboard, colNSize: Int)

This class represents a ListView class, which is made for easier cursor handling. For example, changing the background color of the ListView items, when the cursor hovers them and changing them back, when the cursor exits them. Also, I used this class's methods, when I wanted to change the border color of the tiles, in which the same digits are already placed.

5. Algorithms

Possible combinations:

When we calculate possible combinations that can be applied to the selected sub-area, we use the knowledge about how many tiles are in the current sub-area. We also need the information about the digits already placed in tiles in the current sub-area. Also, we need the knowledge of candidates that can be placed in these tiles according to standard 9 x 9 sudoku rules. Then having this information, we use one of the three different algorithms, that calculates possible combinations using a brute force approach.

For example, when we have two tiles with no digits placed in them, we select the first candidate in the first tile and go through all the other tile’s candidates. If the sum of these candidates' addition gives us the sum of the subarea, then we store it in Buffer-structure so that we can easily get it later. A similar algorithm is used when we have a sub-area of three or four tiles. Brute forcing is usually a slow method of getting all the combinations, but because in this case we have only digits from 1 to 9, then it should not take long, and that is why we use it in this project.

If there are more candidates, we should of course try to optimize these algorithms, if there are ways of doing it.

Candidates:

For finding candidate digits that can be placed in the tile I created the algorithm which consists of several parts.

First, I find candidates that can be placed according to the row where the tile is. If some digits are placed in tiles that are in the same row as the current one, then placed digits should be removed from the list of candidates. A similar approach is used to find the candidates that can be placed according to the square and column. Then, if we want to find candidates according to standard Sudoku rules, we need to find the intersection of these candidate lists.

Then for calculating candidates that can be placed in the tile according to the sum of the sub-area, we use possible combinations for finding all the digits that can be placed and then we again use intersection between current candidates and candidates that we got from the standard Sudoku rules candidates.

Tile neighbors:

I build neighbor lists after tile objects are created according to number of rows and columns in the board. When creating tiles, the algorithm checks tile's coordinates (row, column) and tries to find neighboring tiles' coordinates based on this information. After creating a new tile, the algorithm runs through all the tiles that have been already created and adds the newly created tile as neighbor to neighboring tiles. This algorithm has a high time complexity, when dealing with big sudoku boards. But as I said earlier, when explaining another algorithm, in this project it should be fine.

If we want to optimize this method, we should decrease the number of iterations the program takes to find neighbors. This can be achieved, for example by creating a loop, which iterates only four times. When a new tile is created, we check its column and row and find the coordinates of the tiles in the four main directions. If some tiles are not defined at that point, we just jump over them. Unfortunately, I haven't had so much time to implement this using a more optimal and efficient algorithm of calculating neighbors of the tiles.

Coloring:

According to the Killer Sudoku rules two neighboring sub-areas cannot have the same color. A single tile's neighbors are located in four main compass directions: North, East, South and West. So, diagonally placed tile is not interpreted as a neighbor. Every tile has a list of neighbor tiles. Every subarea has a list of neighbor subareas. We then compare index of sub-areas between the current tile and its neighbors. If index differs, then we add sub-area to the current sub-area as a neighbor. Same action is done with the other sub-area.

For coloring I used graph coloring algorithm, Greedy algorithm. This algorithm states that we should create a graph of nodes, in which some of the nodes are connected. Then we select one node (doesn't matter which) from where we start coloring our graph. We give the first node some color and then continue coloring the graph. At the next node we select the most optimal color option at that moment and if there are no already used colors left, then we define a new one.

In our case, nodes are sub-areas and instead of curves we have neighbors of each sub-area. So, we start from the first sub-area and give it some color and also keep all previously used colors in colors list. Then we continue with the next sub-area and if its neighbors have some colors, we should select from the colors list the one that isn't used by neighbors. If there are no colors left, we define a new one and add it to the data structure.

6. Data structures

I used scala.collection.mutable.Buffer data structure, when I had to add some information to the end of the data structure, or I had to get some elements from a particular index. This is because Buffer data structure is very convenient and efficient, when we have for example for-loop, and we add elements at the end of the structure. Every time the memory slot becomes full, the program creates a new Buffer, which has two times more memory as before. So, it will not create a new Buffer often, which will reduce the time. Also, Buffers are known to be very good at returning the element at the given index.

I also used immutable.Vector data structure to store information that will be initialized only once, when the program starts working. This data structure was used because elements of the data structure won't be modified and that's why it's better to store them in immutable data structures.

In some places I also used Option data structure. It's convenient, when we don't know in which type the information is given, but we are expecting a particular type. If the value has something else that we are expecting, then Option will return None and if the expected value, then Some(). This is very convenient, when trying to handle possible errors that might occur because of the corrupted file or other possibilities, when the received value wasn't same as expected one.

7. Files and Internet access

The program should handle text files with .txt format.

I modified the format of the configuration file a bit compared with the one given in the "Technical plan" and examples will be given in the testingData folder in the project. It should contain the following tags: #Subarea:, #Title:, #colsize: and #rowsize:. Also, if some digits are already placed in some of the tiles, then the user can specify them using #placedNums:.

8. Testing

I created a “TestGame” class that has a number of unit tests, which are mainly testing “Puzzleboard”-class's and “FileReader”-object’s methods. GUI part of the application was tested manually.

The project successfully passes all the developed unit tests.

9. Known bugs and missing features

As I mentioned earlier, when file is not saved yet, the user still can open a new file and remove previous progress. Also, the drop-down list of possible candidates that can be placed in the tile will not disappear, when the user clicks outside the board. I think this can be fixed by using some cursor handling method in the Main.scala file.

If the file opening isn’t successful, then the black dot on the screen appears. I think it is there, because I didn’t remember to remove some object from the grid, and it is displayed as a very tiny object. The title of the application doesn’t show the file name with capital letters if it contains them and displays also .txt after the file name, which is not so nice. I think the latter one can be done by using dropRight method on the file name string. The capital letter issue is a bit harder, and I guess, the fix should be done in the FileReader object, when reading a configation file.

The drop-down list of possible candidates has sometimes empty slots. This doesn’t affect the program logic but is not so nice from the usability perspective. I think this can be fixed by finding the method or some property on ListView object in Scalafx-library, which will allow to set the size of the drop-down list to be same as the number of candidates it contains.

When finding the placement of the tiles top left corner in the GUI, so that the sub-area sum and placed numbers will appear in the grid, I calculated each border of every object on the board. I end up doing this, because I didn’t find the proper way of doing it, like some method or property, which will tell me, the proper coordinates of each tile.

10. 3 best sides and 3 weaknesses

I think the biggest weaknesses of my program are in error handling of ill-defined configuration files. There are quite a lot of way a configuration file can be ill-formed, and it was a bit hard for me to find all those places where my program could crash because of the corrupted file. Also, I didn’t divide the Main.scala object into smaller ones, which led to harder bug fixing and slow building. As a result the readability of the code became sub-optimal. My GUI window is static,

which means that, when resizing the window, there might be situations when all objects are not visible. In addition to these I didn't test the GUI automatically, which means that it can have some bugs.

I think one of the best parts of the code base is the clean separation between front-end (GUI) and back-end, which does all heavy lifting of application logic. This should simplify the transition to a different type of GUI or GUI framework.

Another useful feature is the asterisk in the title of the screen, when there are some unsaved changes on the board. I believe that this improves user experience. I also think that saving a file is done in a quite convenient way. As a further improvement, the Ctrl + S key combination could be handled, to save the file.

11. Deviations from the plan, realized process and schedule

First, I created a couple of main classes "Puzzleboard", "Subarea" and "Tile" almost without methods. I also started studying how to create a GUI window and tried to create it. After the GUI window looked a bit better, I switched again on the back end of the program. I started implementing FileReader-object methods, such as readFile and readFilePuzzleboardCfg. Then I tried to improve both sides of the program back- and front-end. I created "FWriter"-object and wrote some methods to it so that the program could save the file from the GUI. At this point I also colored tiles depending on the sub-area they are placed in. It wasn't Greedy algorithm initially, and every sub-area contained a different color.

12. Final evaluation

In my opinion, the program works fine, and is able to do the basic things that were mentioned in the instructions for this project. The main weak aspects that I found are described in "9. Known bugs and missing features" section of this document. Also, as I mentioned in "5. Algorithms" section, some of the algorithms are not so efficient, if more iterations are required. However, because this doesn't affect the program, we don't need to do so many calculations.

The structure of GUI could be better, for example, dividing the Main.scala object into smaller ones. As a result, the modification of the code would be easier if we want to add some features to the program. Data structures in some places could also be better, for example, when storing the colors of the sub-areas while applying Greedy algorithm, we could use Set. This is because colors should always be different and Set data structure never contains the duplicates.

If I start this project from the beginning, I would try to spend more time developing it. As a result, I guess some parts could be done better, for example more descriptive and comprehensive error messages and error handling, when reading a file. I would also probably try to create automated GUI test, so that it would be easier and faster for me to test it.

13. References

I used StackOverFlow and Geeksforgeeks.org for problem solving, scalafx.org to get familiar with Scalafx and I also used Wikipedia, when I tried to understand how the Greedy algorithm works. I also found ChatGPT to be quite instrumental to get an inspiration and to unblock me in answering some of the questions about GUI API wokings.

I also checked the solution of the example sudoku board that was given in the task description from Wikipedia “Killer sudoku”. I also used this website for generating a new killer sudoku puzzle with solution: https://www.killersudokugenerator.de/