

Report HW5: Sudoku Game with GUI

I will consider this as an independent assignment rather than just an improvement of a previous assignment. I have extended the functionalities with the implementation of GUI using swing. Along with all the things that I learned during the course I wanted to implement GUI right from the Tello Drone assignment as I had never done something like a UI before.

This application uses the Sudoku Solver algorithms from the last assignment and in addition to Sudoku solver; the following features have been added:

1. The application allows a user to solve the puzzle manually (user plays the game) by using the row column text fields and Set value button.
2. The Load Button allows a user to add puzzle from a text file. The dimensions can vary and can be extended to any puzzle size. The application opens with a default 9X9 puzzle which can be updated with a load file.
3. The help button extends the help requirement from last assignment.
4. The radio buttons have been added to allow a user select the algorithm which should be used to solve the puzzle although this feature does not match with the end experience but I wanted to keep this functionality from the last assignment intact and thus I implemented it this way.
5. The puzzle and solution validations are the same as last assignment.
6. Furthermore I tried to improve abstraction by using meaningful labels and identifiers. I've set the access of variables and objects as private where needed. The modularity has been improved and as per the discussion I've removed all the static methods from the last assignment since it is not an object oriented practice.
7. Code reuse is one attribute that I tried to use where ever possible.
8. The invalid puzzles are represented in the form of a message box.
9. I am removing the save output to text file from last assignment as it does not go well with GUI although it can be easily extended.

I started the GUI with a MVC pattern in mind as it appeared a better choice after some research and the lecture.

- The **Model** contains only the application data; it contains no logic describing how to present the data to a user.
- The **View** presents the model's data to the user. The view knows how to access the model's data, but it does not know what this data means or what the user can do to manipulate it.
- The **Controller** exists between the view and the model. It listens to events triggered by the view (or another external source) and executes the appropriate reaction to these events. In most cases, the reaction is to update, the result of this action is then automatically reflected in the view.

Although this appears simple but it took me more effort than the other assignments as I did not have any experience and knowledge with swing or designing a GUI. It took me a lot of time to study things on internet and implement.

I also took some references mainly tutorialspoint and GeeksforGeeks websites to look at the examples on how components like JPanel, JFrame, JFileChooser etc work. I've also used the code from this website as a boilerplate to use these components.

The thing that consumed most of my time was updating the GUI with new puzzles and updated solutions. But later on I figured out the solution to this issue and I found out using a couple of solutions like SwingUtilities.invokeLater() and repaint() method to update the GUI and thankfully it worked for me.

The algorithms have a strategy pattern applied and the execution of algorithms follows a set of steps for each algorithm, so a template method pattern fits well there. The algorithms follow an open-close principle.

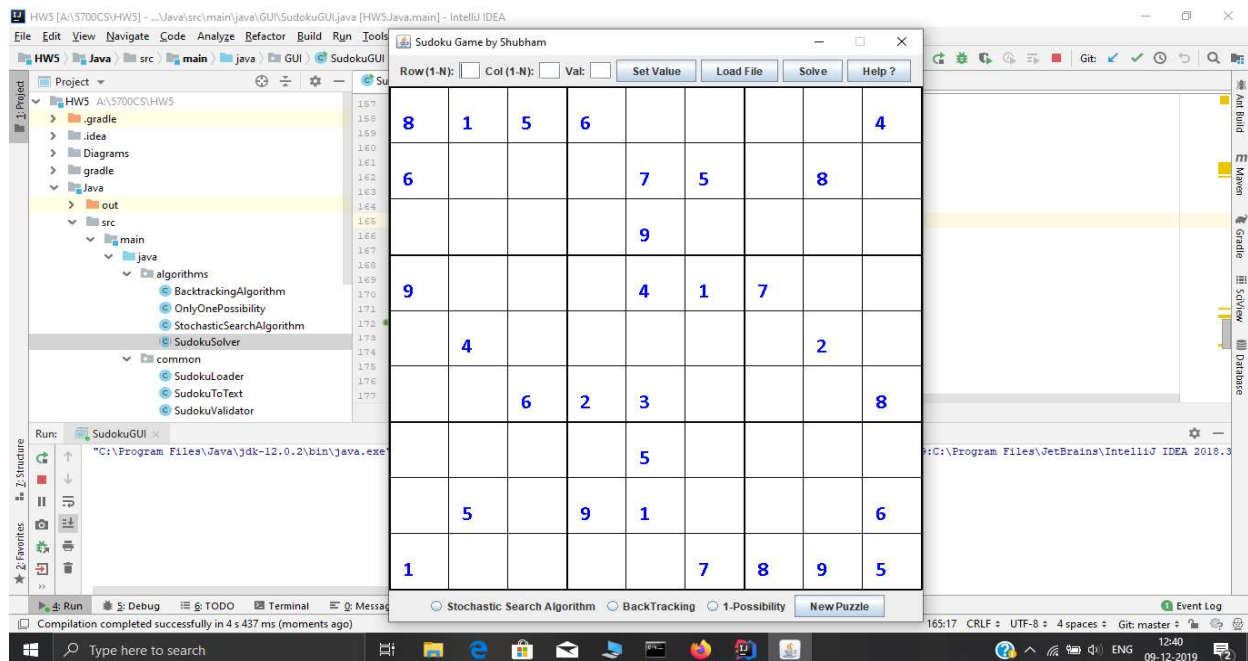
The GUI is implemented with a MVC pattern which also goes well with the design. The method patterns have not been forced into the design and are only applied where necessary.

I wanted to extract v listeners from the Controller in MVC but since GUI wants the same object and components to be updated the action listeners need their components. For example the isSelected() in radio buttons uses the instantiation of the button to check if it is selected or not. Reset button (previously named as new puzzle reset spuzzle.)

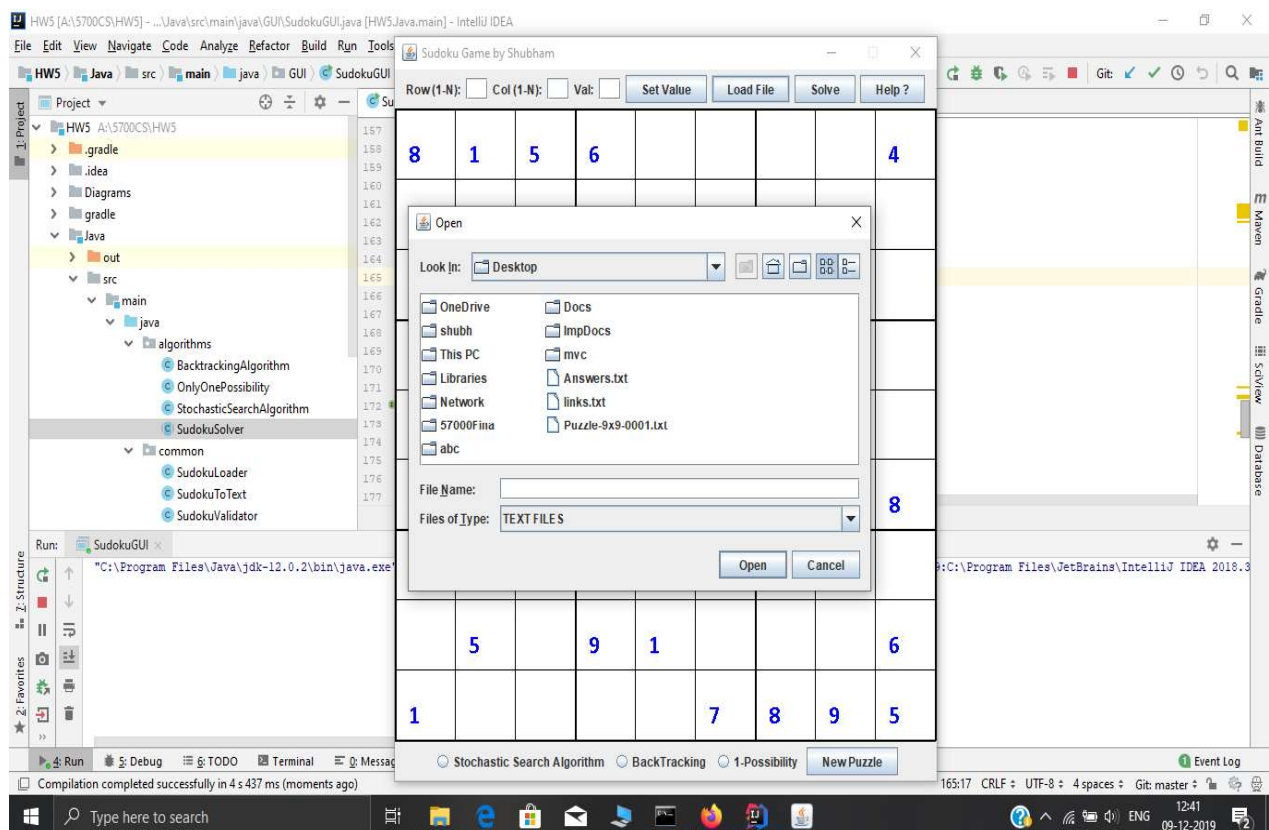
The test cases have been written for all non GUI components with almost full method coverage.

The class diagrams, interaction diagrams and state diagram have been uploaded in Diagrams folder.

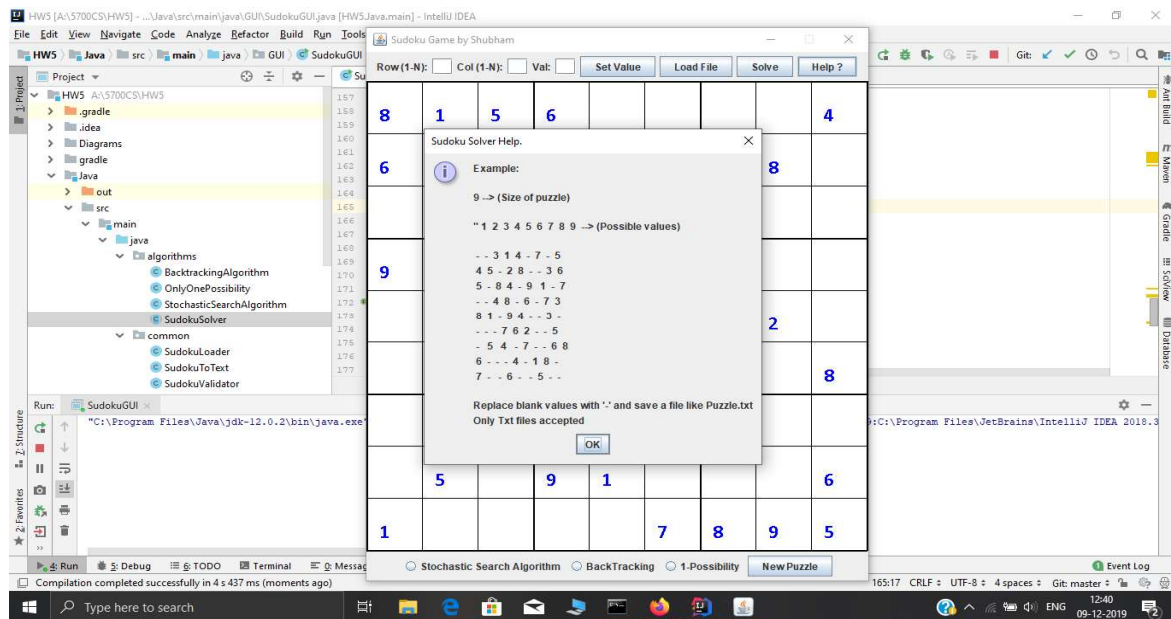
How the application opens with a default 9X9 Sudoku.



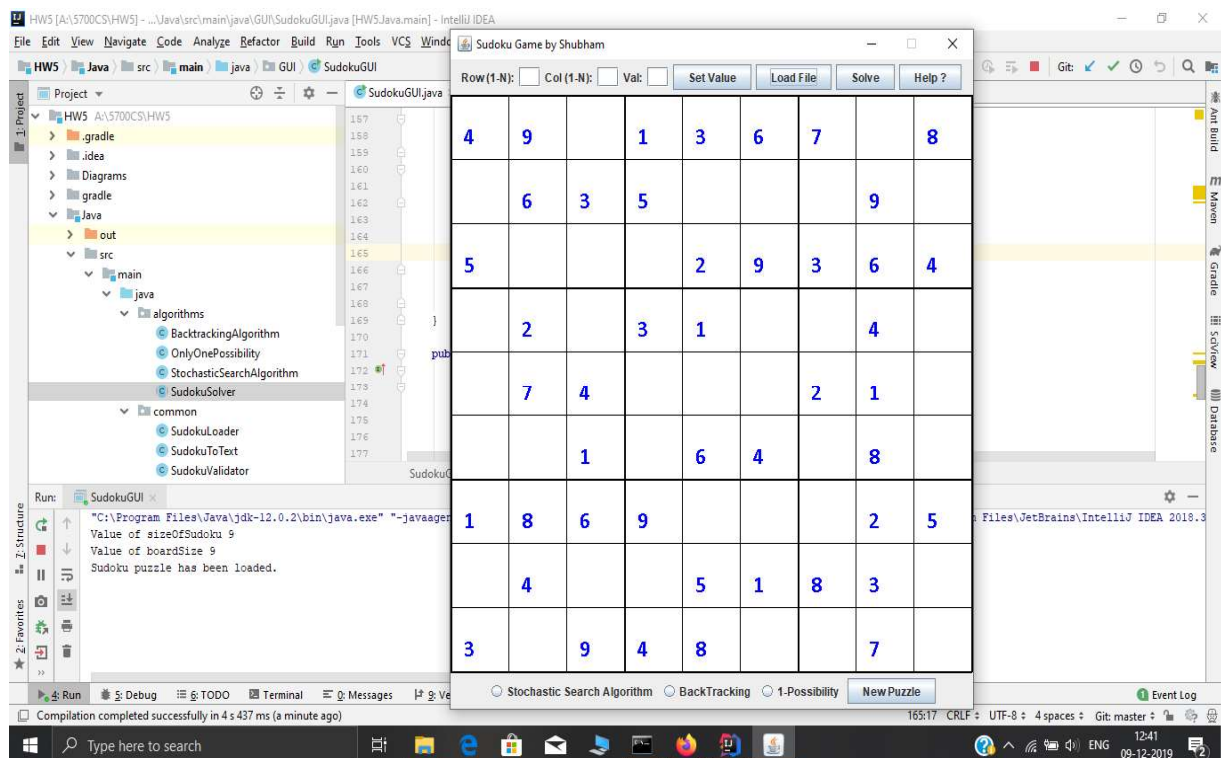
Used Load Button to update the puzzle from a text file.



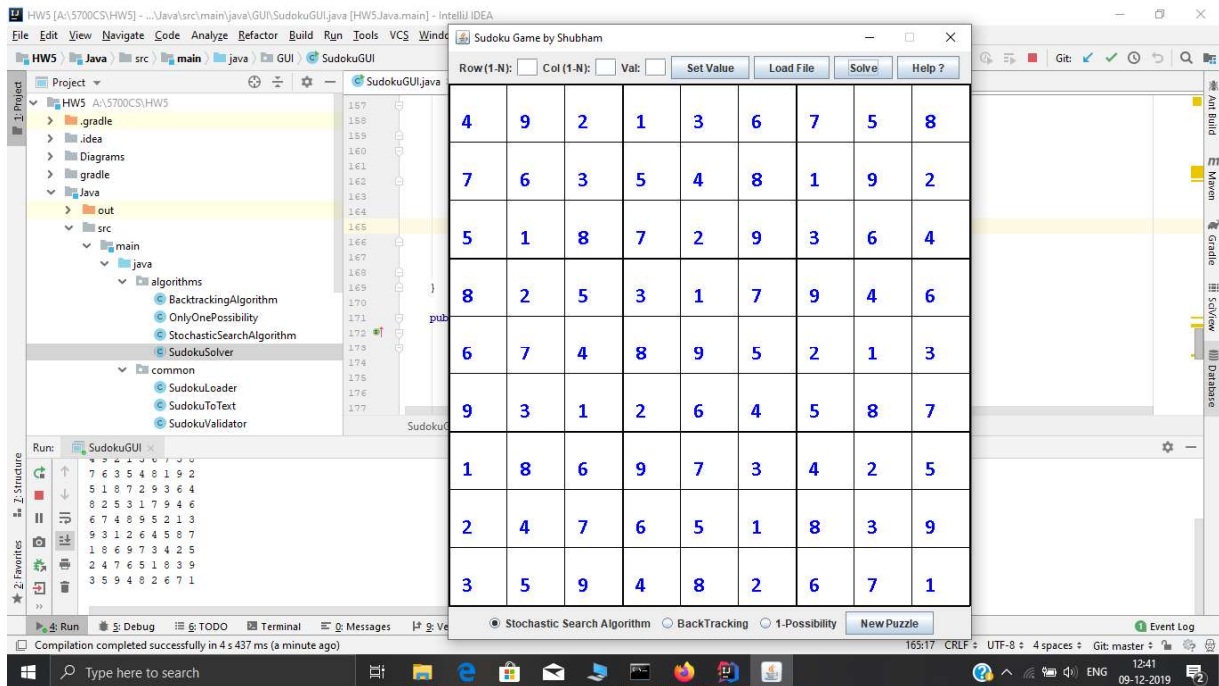
Help button opens a dialogue to offer help to user.



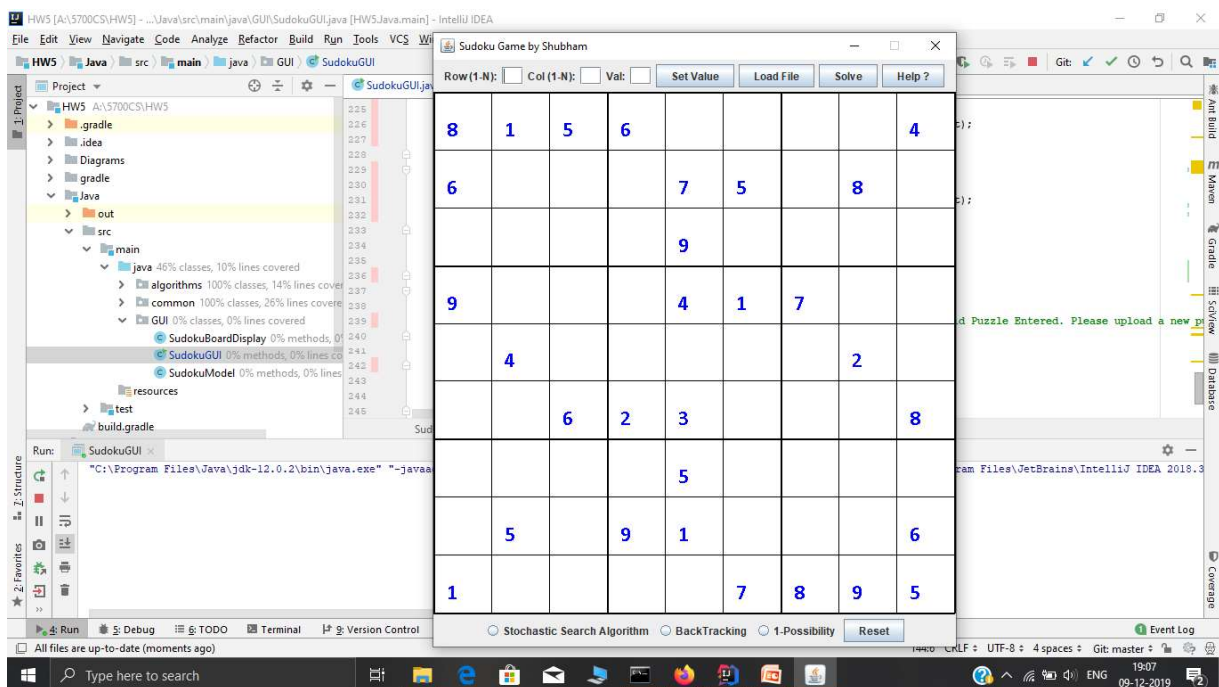
Updated puzzle after loading a text file.



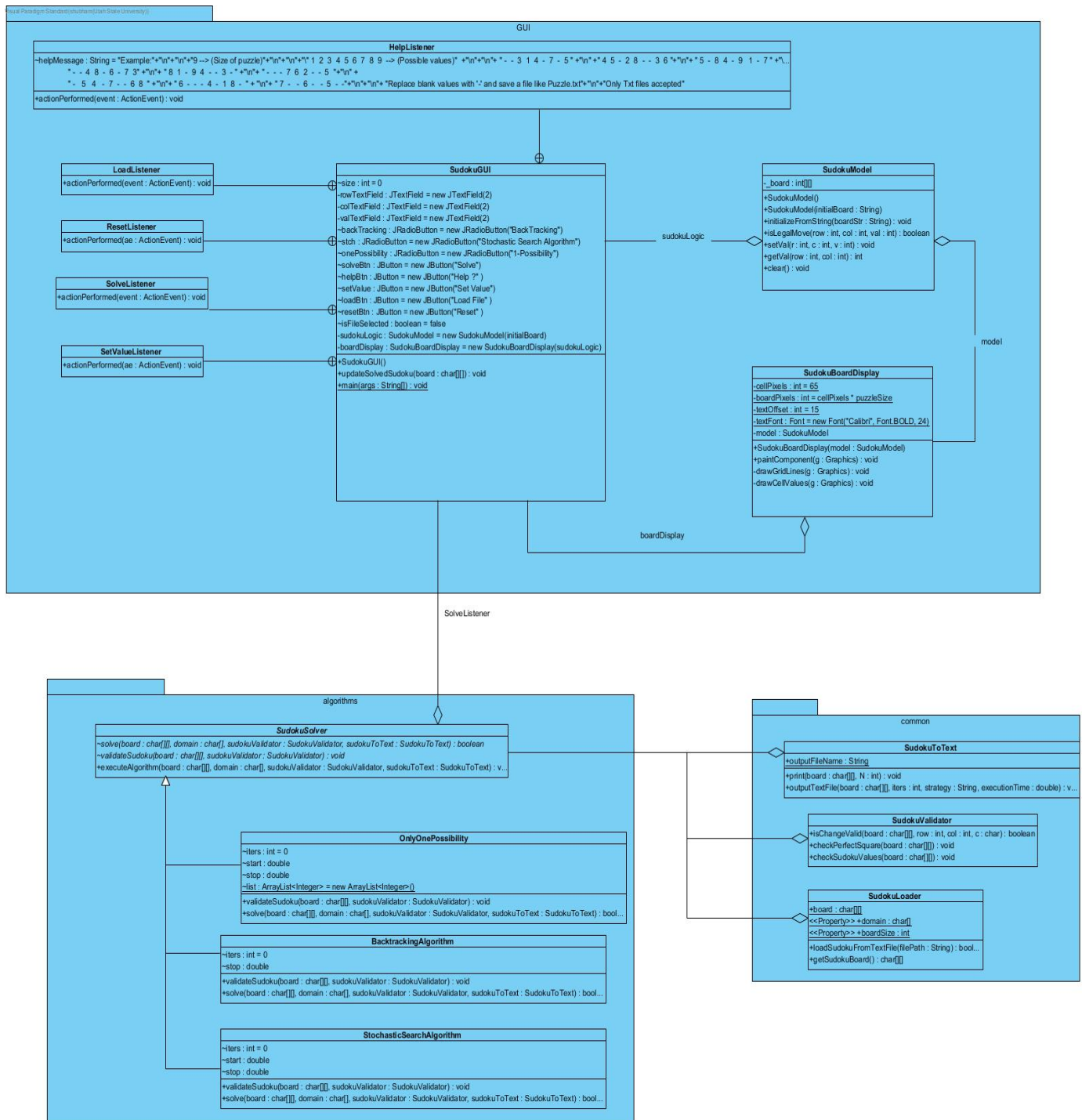
Click on a radio button to select the algorithm and hit the solve button.



Updated GUI:

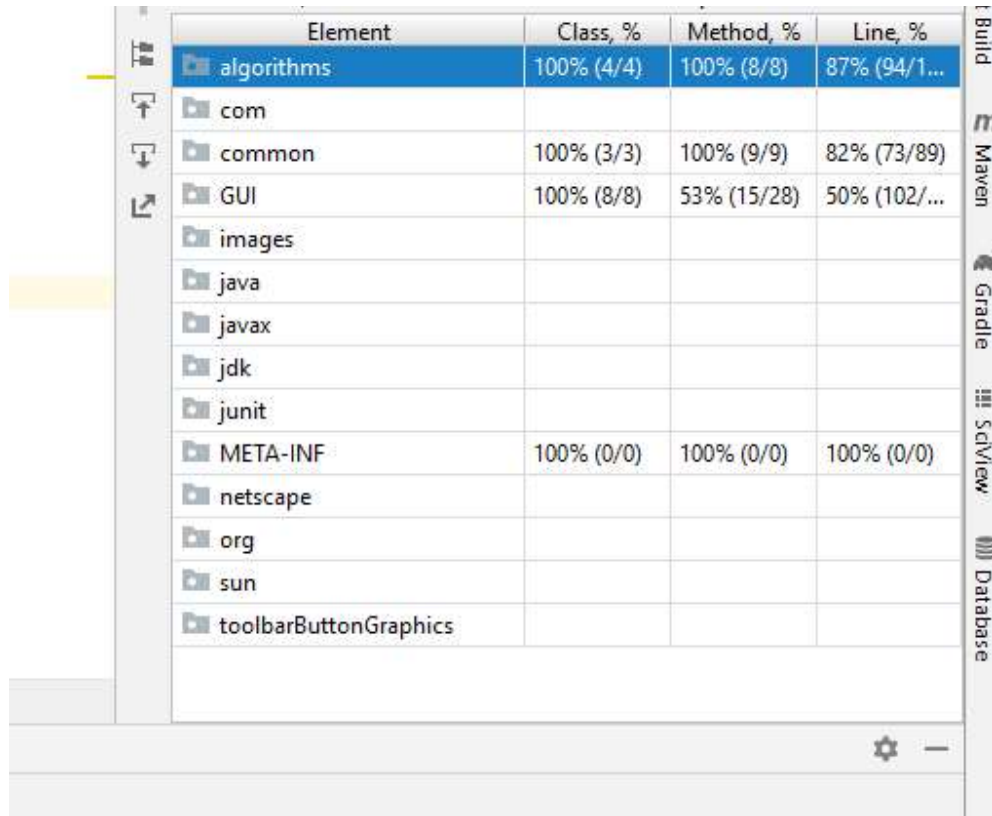


Class Diagram:



Unit Testing:

I've tested all the classes the coverage of GUI was low because of Action Listeners and initialization of GUI components.



The screenshot shows an IDE's coverage tool. On the left is a directory tree with folders like algorithms, com, common, GUI, images, java, javax, jdk, junit, META-INF, netscape, org, sun, and toolbarButtonGraphics. The 'algorithms' folder is selected. On the right is a table with four columns: Element, Class, %, Method, %, and Line, %. The table lists coverage data for various elements, including 'algorithms' (100% class, 100% method, 87% line) and 'META-INF' (100% class, 100% method, 100% line). The IDE's right sidebar shows tabs for Build, Maven, Gradle, ScView, and Database.

| Element | Class, % | Method, % | Line, % |
|-----------------------|------------|-------------|---------------|
| algorithms | 100% (4/4) | 100% (8/8) | 87% (94/108) |
| com | | | |
| common | 100% (3/3) | 100% (9/9) | 82% (73/89) |
| GUI | 100% (8/8) | 53% (15/28) | 50% (102/204) |
| images | | | |
| java | | | |
| javax | | | |
| jdk | | | |
| junit | | | |
| META-INF | 100% (0/0) | 100% (0/0) | 100% (0/0) |
| netscape | | | |
| org | | | |
| sun | | | |
| toolbarButtonGraphics | | | |