Creating a Sudoku Solver Application with Java

Owen Sowatzke

Georgia Institute of Technology
School of Electrical and Computer Engineering

ECE 3005
Summer 2020

Table of Contents

1. Overview

This document provides instructions for how to create a sudoku solver application using Java. It details the solver's human-like algorithm, implementation of the algorithm, and application graphics.

2. Algorithm

To solve a given sudoku puzzle, all boxes, columns, and rows must be completely filled and must contain all numbers ranging from one to nine. The simplest algorithm to implement would be a brute force algorithm, which uses only guesses to find a solution. Mathematics Professor Frazer Jarvis estimates that there are approximately 6.671×10^{21} possible sudoku solutions [1], and as such, it is conclusive that a brute force algorithm is very inefficient. To minimize unnecessary guessing in the algorithm, a more human approach is taken. In particular, three solution techniques compromise the algorithm. They include placement of a single candidate, cross-hatching, and guessing (only when appropriate).

2.1 Placement of a Single Candidate

Placement of a single candidate is a solution technique that involves examining each box, column, and row as shown in Figure 1. The entries already in each structure are tallied, and if there are eight of them, the missing entry in the structure is filled with the number between one and nine that is not already in the set.

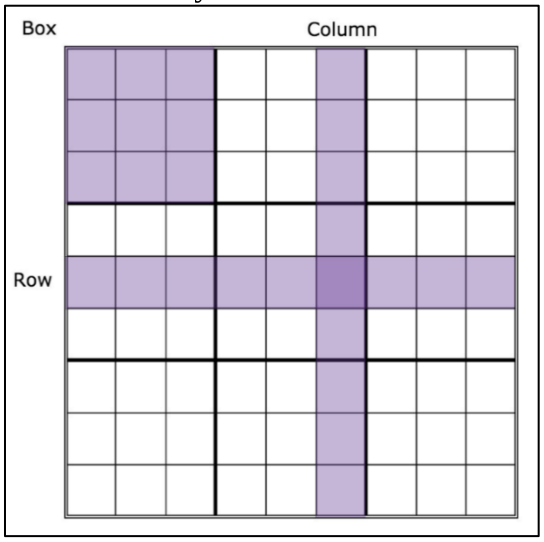


Figure 1. Empty sudoku puzzle highlighting a box, a column, and a row.

2.2 Cross-Hatching

In the technique of cross-hatching, each box, column, and row is checked, considering the missing numbers and empty tiles. In a given structure, every empty tile is a candidate for a missing number. When cross-hatching is applied to a box, rows and columns that contain a missing number cannot contain that missing number again in the box. A similar analysis can be performed on columns and rows. If only one empty tile in structure is still a candidate for a missing number following this analysis, then that empty tile must contain the missing number. Figure 2 showcases how this process works within a given box.

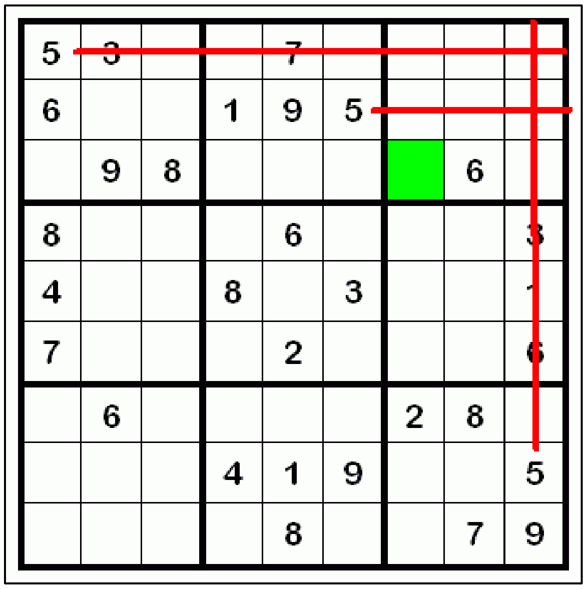


Figure 2. Using cross-hatching to place the number five in the upper rightmost box. Tiles with red lines across them have been eliminated, and the tile highlighted in green is the location of the missing five.

2.3 Guessing

The technique of guessing employed is an extension of the cross-hatching technique and is only used when the latter techniques fail to progress toward the solution. For example, consider a situation in which the end result of cross-hatching is more than one candidate. In this situation, one of the candidates is chosen and the puzzle is continued. If an error is found (two of any number in a row, column, or box), then the guess was incorrect. In this situation, the puzzle is reverted to its pre-guess state and a candidate not chosen at the time of the original guess is instead selected. This process can be extended to multiple tiers of guesses. Figure 3 demonstrates how wrong guesses are handled when there are multiple tiers of guesses.

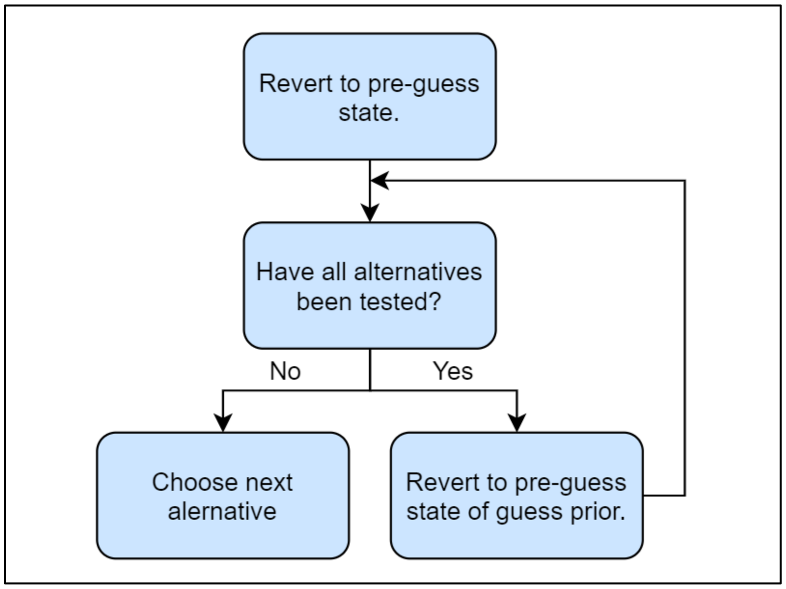

Revert to pre-guess state.
Have all alternatives been tested?
No
Yes
Choose next alernative
Revert to pre-guess state of guess prior.

Figure 3. Flowchart illustrating how wrong guesses are handled when there are multiple tiers of guesses.

3. Implementation of the Algorithm

The Algorithm is implemented after user data is input and validated. (A valid user-input contains only numbers one through nine and no identical numbers in a box, column, or row). Following validation, the program loops until there are no blank boxes in the puzzle. At every iteration of this loop, the first two algorithm techiniques, placement of single candidates and cross-hatching operations, are exectuted. The program also performs two checks during every loop iteration. The first check determines whether the first two algorithm techniques have changed the puzzle. If not, a guess subroutine is called. The second check determines if the puzzle contains an error. If there is an error, the last guess is incorrect, and an alternative guess is chosen. Figure 3 above shows how the case of a multiple tier guess is handled, while Figure 4 below displays the overall algorithm implementation. For this implementation to function properly, there must be efficient storage and access of not only sudoku puzzle entries but also guesses.

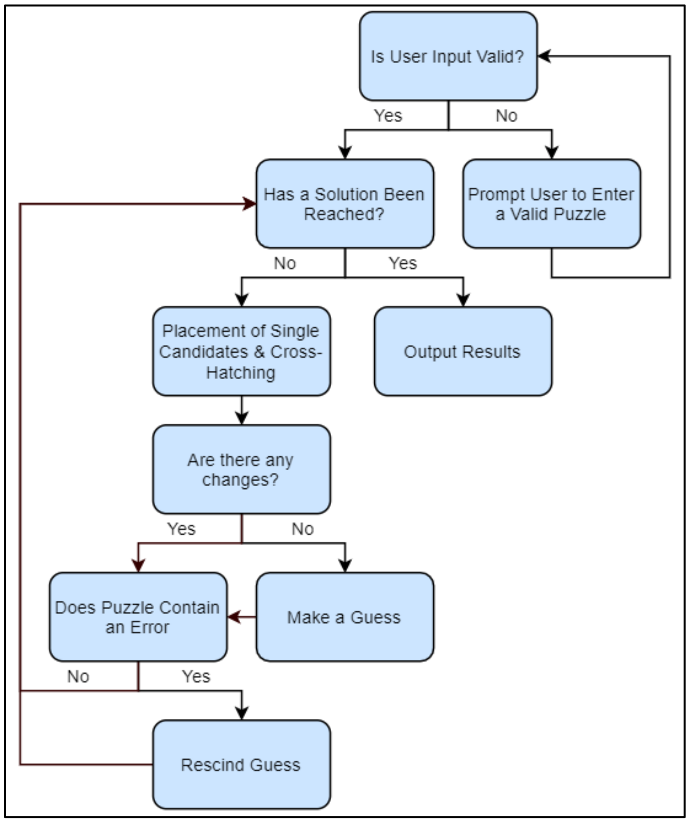

Is User Input Valid?
Yes
No
Has a Solution Been Reached?
Prompt User to Enter a Valid Puzzle
No
Yes
Placement of Single Candidates & Cross-Hatching
Output Results
Are there any changes?
Yes
No
Does Puzzle Contain an Error
Make a Guess
No
Yes
Rescind Guess

Figure 4. Flowchart displaying overall flow of program implementing sudoku solving algorithm.

3.1 Efficient Storage and Access of Sudoku Puzzle Entries

Sudoku puzzles are saved in a [9][9] array, where the first index refers to the box number and the second index refers to the index inside the box. Refer to Figure 5 below.

3.1.1 Accessing Items in a Row

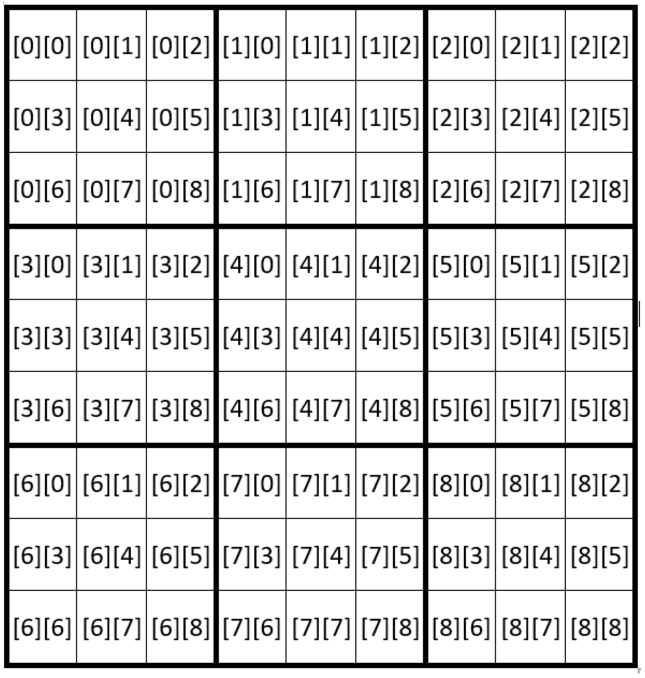
When the array is accessed element-by-element using two for loops with syntax array[i][j]. All items in a row can be accessed according to the following if statement:

```
if (((i/3)==(row/3))&&((j/3)==(row%3)))
```

3.1.2 Accessing Items in a Column

Using the same for loops and array indices, all items in a column can be accessed according to the following if statement:

```
if (((i%3)==(col/3))&&((j%3)==(col%3)))
```



[0][0]	[0][1]	[0][2]	[1][0]	[1][1]	[1][2]	[2][0]	[2][1]	[2][2]
[0][3]	[0][4]	[0][5]	[1][3]	[1][4]	[1][5]	[2][3]	[2][4]	[2][5]
[0][6]	[0][7]	[0][8]	[1][6]	[1][7]	[1][8]	[2][6]	[2][7]	[2][8]
[3][0]	[3][1]	[3][2]	[4][0]	[4][1]	[4][2]	[5][0]	[5][1]	[5][2]
[3][3]	[3][4]	[3][5]	[4][3]	[4][4]	[4][5]	[5][3]	[5][4]	[5][5]
[3][6]	[3][7]	[3][8]	[4][6]	[4][7]	[4][8]	[5][6]	[5][7]	[5][8]
[6][0]	[6][1]	[6][2]	[7][0]	[7][1]	[7][2]	[8][0]	[8][1]	[8][2]
[6][3]	[6][4]	[6][5]	[7][3]	[7][4]	[7][5]	[8][3]	[8][4]	[8][5]
[6][6]	[6][7]	[6][8]	[7][6]	[7][7]	[7][8]	[8][6]	[8][7]	[8][8]

Figure 5. Sudoku puzzle grid correlating array indices to tiles within the puzzle.

3.2 Efficient Storage and Access of Guesses

To track guesses made when solving a given a puzzle, a guess class is defined. This class defines a data type that holds a guess and all the alternative guesses. Figure 6 defines this class pictorially. Within the class there is an integer “num” (which whose location is being guessed). The class also includes three ArrayLists (variable length lists) which define the location of the guess (“box” and “index”) and the validity of each possibility (“valid”).

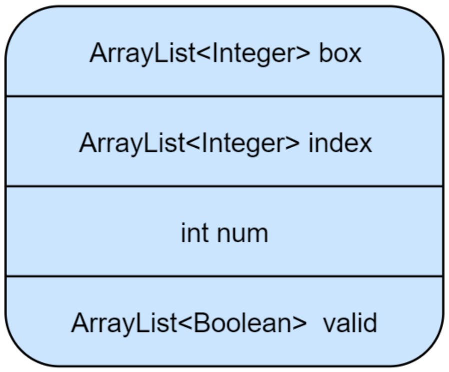


Figure 6. Data included within the Guess class.

Within the implementation, a list of guesses with type ArrayList<guess> holds a list of all the guesses made, while another list with type ArrayList<int[][]> holds screenshots of the pre-guess puzzles. Entries are added to these lists when additional guesses are made and are removed from these lists when a guess and its alternatives are all invalid. As already stated, this is only the case when a previous guess is incorrect.

4. Application Graphics

An integral part to a sudoku solver is the application graphics. These graphics are composed of a JFrame, a JLabel, JButtons, and JTextFields as shown below in Figure 7. The most fundamental parts of these graphics are its resizeability and its ability to collect user input.

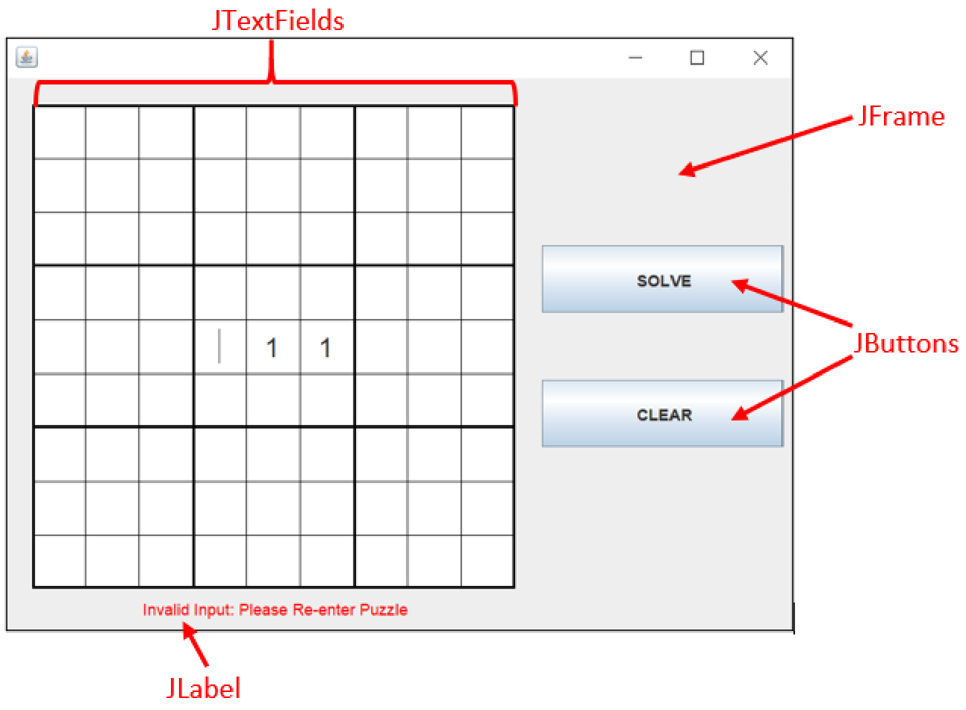


Figure 7. Sudoku Solver GUI with component types labeled.

4.1 Implementing Resizeability

Usually a layout manager ensures placement of items and sizing of items within the GUI (defined by the JFrame); however, a layout manager also restricts the placement of items. As a result of this, a layout manager is not used in this implementation. Instead, the GUI is given an initial size, and the components are assigned a starting location and size.

A component listener [2] listens for resizing events. When the window is resized, the height and width of the GUI are taken into consideration. In order to ensure the components within the window retain their original shape, the components are resized based on the smallest dimension of the GUI relative to its original dimensions.

4.2 Collecting User Input

To facilitate transfer of information between a separate graphics class and main class, entries within the graphics class are defined publicly. This allows textbox entries to be gathered and results placed within the textboxes.

An action listener [3] then waits for a button press. If the solve is pressed, the textbox entries are stored and evaluated for correctness. First, the graphics section of the program verifies that there are no entries other than the numbers one through nine. Once this has been verified, the sudoku puzzle is evaluated for duplicate entries in boxes, columns, and rows. If neither of these conditions occurs, then the puzzle is ready to be solved, and its values are stored within the 9x9 puzzle array. The solution is then added to this array and transferred back into the textboxes. If the provided puzzle was entered with an error, the JLabel (error message) is made visible and a solution is not attempted.

5. Summary

This wiki provides information useful in creating a sudoku solver in Java. Within this sudoku solver, a human-like algorithm prioritizes placement of a single candidate and cross-hatching to minimize unnecessary guessing. Arrays and a user-defined guess class are structures that help to facilitate the implementation of this algorithm. Finally, resizable graphics defined without a layout manager help graphics components to maintain their shape during resizing events. Components of the GUI are defined publicly to allow other classes to allow gather and place information within the GUI. Overall, the ideas presented within this wiki can be extended to other coding languages.

References

[1] F. Jarvis, B. Felgenhauer, “Enumerating possible Sudoku grids,” June 20, 2005. Accessed on: July 12, 2020. [Online]. http://www.afjarvis.staff.shef.ac.uk/sudoku/felgenhauer_jarvis_spec1.pdf

[2] Oracle, *How to Write a Component Listener*, Oracle. Accessed on: July 12, 2020. [Online]. Available: https://docs.oracle.com/javase/tutorial/uiswing/events/componentlistener.html.

[3] Oracle, *How to Write an Action Listener,* Oracle. Accessed on: July 12, 2020. [Online]. Available: https://docs.oracle.com/javase/tutorial/uiswing/events/actionlistener.html.