Sudoku Generator & Solver

Final Project – Master's Degree, 1st. Semester

Mario Contreras
ITESO
Master's Degree in Computer Systems Student
Tlaquepaque, Jalisco, Mexico
ms705080@iteso.mx

Erick González
ITESO
Master's Degree in Computer Systems Student
Tlaquepaque, Jalisco, Mexico
ms705070@iteso.mx

***Abstract*—Sudoku, a mathematical game, has gain popularity over the years. Sudoku players can find it challenging, but software developers can also find challenging to generate a valid and soluble grid, same for solving a Sudoku. We present a simple Java application that using backtracking algorithms can generate a new grid or to solve it.**

Keywords—Sudoku; backtracking

I. Introduction

Sudoku is a game that uses a 9x9 grid divided in regions. A Sudoku grid is a type of Latin square, which is a square *nxn* matrix filled with *n* symbols and symbols are restricted to be different in each row and column. Figure 1 shows a Latin square of 5x5.

A full Sudoku grid, also known as solution, contains numbers from 1 to 9 on each row and column, and each subgrid or region cannot repeat a number. From the solution, some numbers are removed and the cell is leaved as a blank cell. The cells that contains numbers are typically called clues. Figure 2 and 3 are examples of a solution grid and a game grid with 43 clues. The game consists of filling the blank cells without breaking the Sudoku grid constraints.

1	2	3	4	5
2	5	4	1	3
5	3	1	2	4
4	1	5	3	2
3	4	2	5	1

Fig. 1. A Latin Square

Sudoku is not the original name for this game. It was published by *Dell Pencil Puzzles and Word Games* in 1979 as "Number Place" [1]. A Japanese magazine published this game in 1984 and it was named as "Sudoku" that can be translated as "single numbers". Today, Japanese use the original name while other regions use the word Sudoku.

Solving a Sudoku is a challenging task, and it is more challenging if the number of clues is too small. The challenge is not a direct relation with the number of clues in the initial grid, but if we have more numbers to place, there are more mental operations to perform. Mathematicians have calculated the minimum number of clues that are required in an initial Sudoku with a unique solution is 17. McGuire et al., created algorithms to search for at least one 16-clue Sudoku but they couldn't find it, therefore they concluded 17 is the minimum of clues required for a Sudoku [2].

3	7	2	4	8	9	5	1	6
6	9	8	5	1	7	3	4	2
4	5	1	6	3	2	7	8	9
7	2	6	9	5	8	4	3	1
9	4	3	7	6	1	8	2	5
1	8	5	2	4	3	6	9	7
5	6	9	8	2	4	1	7	3
2	1	4	3	7	6	9	5	8
8	3	7	1	9	5	2	6	4

Fig. 2. A Sudoku grid (solved)

	7		4				1	
6	9			1			4	2
	5			3	2	7	8	9
7	2	6	9		8		3	
		3	7	6				5
1		5		4		6	9	7
5	6	9		2		1		
	1	4				9	5	8
8		7		9				4

Fig. 3. A Sudoku grid with 44 clues

II. Math Background

In a *nxn* Sudoku, each cell can be 1..*n*. Therefore, a 9x9 Sudoku, which has 81 cells, could have 9^{81} possible combinations. However, that's if we don't consider any of the Sudoku rules. First of all, a Sudoku is a Latin square, therefore

we cannot use the same number twice in each row or column. In [1] we can find that there are 5,524,751,496,156,892,842,531,225,600 Latin squares of order 9. If all rules are considered, we reduce the number. We can also consider that if we take one row or column and swap it with another, it is essentially the same grid (an equivalent grid). If equivalent grids are removed, the number of grids according to Delahaye in [1] is 5,472,730,538, which is still too high.

Due to the restrictions, one way to use graph coloring. The graph is formed with 81 vertices, one per cell in the grid. Edges represents a connection between numbers that form part of the solution. Yildirim et al represent such graph as shown in figure 4. In Figure 4, B is the order of the Sudoku puzzle (typically 9).

$$
\begin{aligned}
G_B &= (V_B, E_B) \\
V_B &= \left\{ V_{i,j} \mid 1 \le i, j \le B^2 \right\} \\
E_B &= \{ (V_{i,j}, V_{t,k}) \mid i = t \vee j = k \vee \\
&\quad \left(\left\lceil \frac{i}{B} \right\rceil - 1 \right) * B + \left\lceil \frac{j}{B} \right\rceil = \left(\left\lceil \frac{t}{B} \right\rceil - 1 \right) * B + \left\lceil \frac{k}{B} \right\rceil \Big\}
\end{aligned}
$$

Fig. 4. Formal notation of a Sudoku graph by Yildirim et al.

III. SUDOKUSOLVER GUI

Our application was written in Java (target JRE 8u66) using NetBeans 8.1 IDE. We created a Cell class which inherits from JPanel. It represents a single cell in a Sudoku grid. Cell class contains a JTextField attribute named valueField that represents the number cell and a JLabel attribute named letterLabel to show the letter clue (S, M, L which stands for small, medium and large). Figure 5 is the Cell class diagram. Our Sudoku implementation is a variant that uses a letter plus the number. Letters add additional restrictions to the grid.

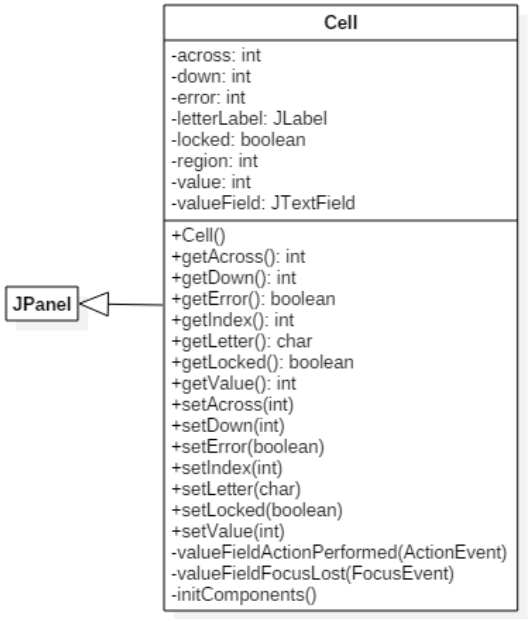

Fig. 5. Cell class diagram

A cell has a value (1..9) and a letter (S, M, L), but it also has a reference to the column (across) and row (down) in the complete Sudoku grid. The subgrid or region is also stored. If a cell represents a clue, it is considered to be locked.

Two valueField events were handled (actionPerformed and focusLost) to prevent invalid values (letters, symbols, etc.). We use getter and setter methods to control the cell appearance and to validate input. When a cell is locked (it's a clue), the setter method named setLocked() change the text color and it also disables the text field. setError() is used to mark a cell as an invalid entry from the user and it will be rendered with red text. If the value is 0, setValue() will store 0 in value field, but the textfield will be empty. Figure 6 includes the setLocked() and setError() methods code.

```
public void setLocked(boolean locked) {
  this.locked = locked;
  this.valueField.setEditable(!locked);
  valueField.setBackground(

  javax.swing.UIManager.getDefaults().getColor(
    "TextField.background"));
  if(locked)
    valueField.setForeground(Color.BLUE);
  else if(error)
    valueField.setForeground(Color.RED);
  else
    valueField.setForeground(Color.BLACK);
}

public void setError(boolean error) {
  this.error = error;
  if(this.error)
    valueField.setForeground(Color.RED);
  else if(locked)
    valueField.setForeground(Color.BLUE);
  else
    valueField.setForeground(Color.BLACK);
}
```

Fig. 6. setLocked() and setError() methods

Cell class, and AboutDialog, a JDialog subclass, are used by SudokuSolver class. Figure 7 is a partial class diagram of SudokuSolver. SudokuSolver class inherits from JFrame and represents the main form. It contains 81 attributes named cell*xy* where *x* and *y* goes from 0 to 8. Those attributes represent the actual cells that are rendered in the form. The name property of those attributes were configured in design time with a 2-character string that represents their relative position. The SudokuSolver constructor uses the name property to populate the cells array which is used as a "shortcut" to manipulate the actual cells. It is easier to have an array of references and access the cells by index (cells[i]) instead of using the variable name (cell24).

We wrote a Difficulty enum (Easy, Medium, Hard) to represent the difficulty level. By default, the application starts with easy difficulty selected. Difficulty just represents how many clues the grid will have when a new game is created. Easy will contain 51 clues, medium 36 and hard will have 21.

The game's basic options are new, hint, evaluate and solve. In order to demonstrate our algorithms, we split new in "New", "New (Blank)" and "New (Solved)" and solve was split in "Solve" and "Solved (Backtracking)". "New (Blank)" creates a new empty grid without any values. "New" generates a new full and valid Sudoku grid and it will be stored in solution array, but the player will show just a few clues. The cells array will contain some of the values (clues) and the rest will be cleared randomly. Due to the algorithm is too fast to see the progress, a copy of the algorithm was used in "New (Solved)" option which contains a delay between each number is selected so the player can see the progress when selecting numbers This copy was implemented in doInBackground method of NewGridTask class, which inherits from SwingWorker. This allows to work with the UI without locking so the user won't think it was hung. Due to the backtracking logic used, numbers will be shown on screen then it will go back a few cells if a number has a conflict with the rest of the values. Due to this option will show all the values, it doesn't really present a playable grid (all values will be filled) and it is for demonstration proposes only.

The backtracking algorithms to generate a new solution and to solve an existing grid, will be discussed in the next sections.

IV. New Grid and New Game – Generate a New Sudoku Using Backtracking

When the application starts, or when the user selects new game option, newGame() method is called. The general algorithm for creating a new solution is to create two 81-elements arrays, one array of Cell objects that represents the new solution and an array of ArrayList objects that will be used to store the available numbers we can use for each cell. A visual representation of the blank grid with a list of available numbers (for cells 1, 2 and 3) is listed as figure 8. Initially, each element of each list in available array will have the same values: 1..9. The available array allows us to keep track of which numbers were already used and must not be used again.

After the initialization phase, a while loop is used to iterate all the cells in the grid (1..81). We select from available array one random number and verify if this number can be used without any conflict, in other words, without breaking the game rules. If the conflict() method confirms the number can be used, it will be removed from the available list and the cell will be copied to the solution, however, if it does have a conflict we just remove it from the list, but the counter variable won't be increment so we know we haven't found a number for that cell index.

It is possible that we ran out of options when selecting a number, and in that case, we have to add all the numbers from 1 to 9 to the available list for that cell and move back to the previous cell. Because we have a list of unused numbers, the selection of number for the previous cell will be different of the one it had, so if it doesn't have a conflict and we move to the next cell, it might have removed the previous conflict and allowed us to move forward. In case we move back and we find that the previous cell doesn't have more available numbers, we will move another cell back and so on until we find one cell that can use a new number.

Fig. 7. SudokuSolver class diagram

After we have successfully found one number for each of the 81 cells without creating a single conflict, we have the solution. After finding the solution, we need to calculate the letters for each cell that will be used in the backtracking algorithm for solving the game without reviewing the solution array, just using the starting clues. Calculating which letter will be used in a cell is easy due to each 3 cells will represent the tree available letters: S, M and L. The newGame() method reads 3 cells at the time and assign a letter based with this rule: S<M<L.

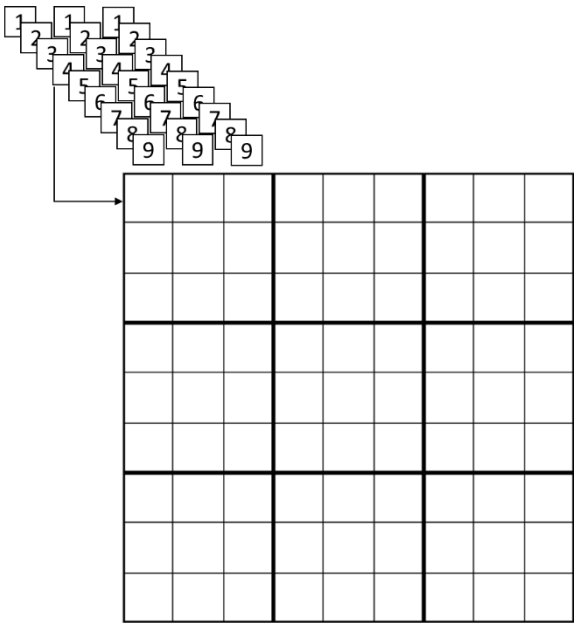

Fig. 8. The available lists for a new grid.

When we have in the solution array all the cells with the final value and letter, we check the difficulty level so we create an ArrayList of the values we are going to remove which are selected randomly. Those values will be removed from the cells array (the array used as a fast way to modify the cells in the form) and the other values will be copied and marked as locked. The algorithm in pseudocode is listed in figure 9.

The conflict() method reviews the new selected cell for each cell that have the same across value, down value or region value. If those conditions are met, then we check if the value is the same (there is a conflict). After reviewing the cells, if no conflict was found method returns false. A visual representation of the conflict verification is shown in figure 10.

V. Solve (Backtracking) Option – Calculate the Solution Using Backtracking

Due to we already have calculated a complete Sudoku grid (which is the solution) when the application starts (or the user selects new game option), we just need to compare the cells that are not locked with the values in the solution array. Solve option of the game menu uses this alternative.

However, if we implemented an algorithm that doesn't rely on the solution array, instead, it calculates the missing values using a backtracking algorithm.

The algorithm creates a list for each cell that is not locked. In an easy game, 51 cells are locked (clues) and 30 cells are cleared. Hence, when calculating the solution using our backtracking algorithm, the initial solution list contains 30 elements.

Each element of the solution list is populated using the cells letters. Each region contains 3 horizontal groups of cells with S, M or L letters. Because letters have indicated a relation between cells, we can eliminate the values that cannot be used due to this restriction. If we have a group of 3 cells with blank, 8 and 3 and M, L and S letters, we know that the blank value must be greater than 3 and less than 8, therefore we add 4, 5, 6 and 7 to the solutions list. If the new game generator leaves three cells in the same sub region without clues, and with letters S, M and L, the

```
newGame() {
  solution = new Cell[81];
  ArrayList<Integer>[] available =
    new ArrayList[81];
  c = 0;
  while(c < 81) {
    if(!available[c].isEmpty()) {
      i = Math.random() *
      (available[c].size() - 1);
      n = available[c].get(i);
      cell = newCell(c, n);
      if(!conflict(solution, cell)) {
        copyCell(cell, solution[c]);
        available[c].remove(i);
        c++;
      }
      else {
        available[c].remove(i);
      }
    }
    else {
      for(x = 1; x < 10; x++) {
        available[c].add(x);
      }
      c--;
      clear(solution[c]);
    }
  }
  top = 0;
  ArrayList<Integer> values =
    new ArrayList<>();
  switch(difficulty) {
      case Easy:
          top = 30;
      case Medium:
          top = 45;
      case Hard:
          top = 60;
  }
  // Copy solution to current grid
  for(i = 0;i < solution.length;i++) {
      copyCell(solution[i], cells[i]);
      cells[i].setLocked(true);
  }
  c = 0;
  values = randomValues(top);
  for(int v : values) {
      cells[v].setValue(0);
      cells[v].setLocked(false);
  }
}
```

Fig 9. newGame() algorithm.

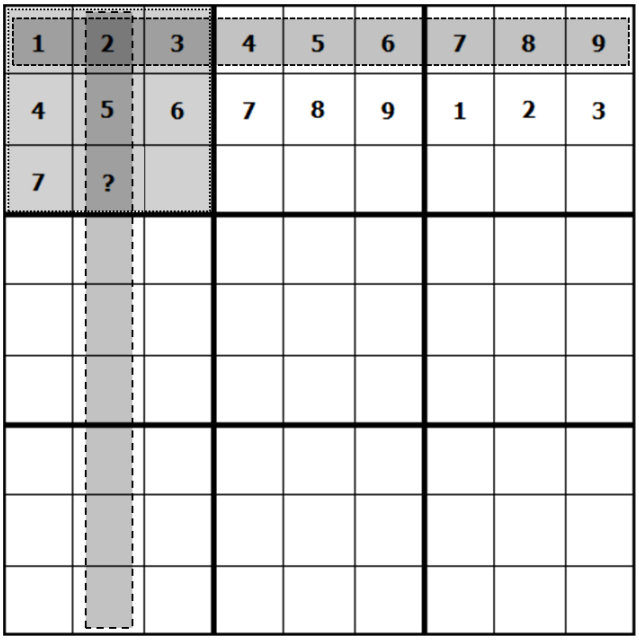

Fig. 10. Visual representation of conflict verification.

first cell can only contain 1..7 due to in worst case, 8 will be M and L will be L. For the second cell, possible values are 1..8 leaving 1 to the first cell because S is less than M, and finally we assign as possible solutions from 3 to 9 for the third cell.

If the solutions list contains lists with one element only, we have a solution. However, this is scenario is uncommon and it's quite rare for a game with hard difficulty.

Once we have the valid numbers for each cell in the solution list, we select one number for the first cell from its corresponding solution list and review if there is no conflict. If it doesn't have a conflict with other values in the same row, column or region, we proceed to validate the next cell. If there is a conflict, we remove the selected value and add it to the errors list. If we ran out of possible solutions, we go back and select another value from the solutions list.

VI. Conclusions

Sudoku is a math game that represents a real challenge for creating algorithms that search for solutions for a combinatory problem. In order to find a new valid grid or to find a solution of an existing grid, we use backtracking techniques that allow us to avoid testing values that were already discarded leading to poor performance. We didn't analyze the performance of our algorithms nor compare them with another alternatives, which could be part of a future work.

References

[1] J.-P. Delahaye, "The Science behind Sudoku," *Sci. Am.*, vol. 294, no. 6, pp. 80–87, 2006.

[2] G. Mcguire, B. Tugemann, and G. Civario, "There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem via Hitting Set Enumeration," 2013.