Sudoku Solver Report

Benjamin Dunn

bed45@aber.ac.uk

Benjamin Dunn
bed45
November 20, 2020

Contents

1 Task 1: The Grid Class

1.1 Instance Variables and Constants

The grid class begins with initializing a 2D array to store the puzzle. Next the constant values are defined:

int MAX GRID SIZE - This constant specifies the dimensions of the sudoku puzzle. Using '9' for the constant will give us a 9x9 puzzle.

int MAX/MIN COORDINATE CALL - These constants specify the range that is being used for coordinates in the rest of the class. For 9x9 puzzles, the range of 0-8 is appropriate since rows/columns/3x3 grids will start at 0 and end at 8 (row[0], row[1], row[2]... etc).

int MAX/MIN VALUE - These constants specify the range the values in each cell can have. For a 9x9 puzzle values from 0-9 are appropriate since we need to fill the puzzle with values 1-9 and we can use 0 for blanks.

It's useful to note all of these constants could be altered in an appropriate fashion to allow our program to be used for larger or smaller puzzles.

Lastly we have initialized 3 different hashSets, one for rows, one for columns, and one for the 3x3 grids within the puzzle. The size of these sets have been defined by the MAX VALUE constant since they will be storing one of each of the possible values 1-9.

1.2 Method *get(int x, int y)*

This method takes two integers, one representing specified row, and one the specified column. It uses an if statement to ensure that the coordinates requested are valid and exist. If they do, it will return the value associated with the coordinates.

1.3 Method *set(int x, int y, int val)*

This method takes three integers. Two to specify the coordinates and a third to input a value at the selected coordinates. It uses two if statements, one to check that our coordinates are valid and exist and another to ensure the value to input is within the range of 1-9.

1.4 Method *isValid()*

This method will scan through the sudoku puzzle and check that the rules of sudoku have not been violated. These rules being:

- Each of the nine rows, nine columns and nine 3x3 grids that are contained within the puzzle must contain all numbers 1 through 9.
- Numbers can only appear once in a row, column or 3x3 grid.

Since we know that the values cannot be repeated, storing the values that are already in the puzzle given into hashSets is useful because they will only store unique values. Thus we have a ground work for quickly checking if any rules are broken. Additionally we will only need to add the values to these Hash Sets and this will have a run time of O(1) meaning we can check very quickly if a puzzle is valid.

To validate, we first create our hashSets for rows, columns and 3x3 grids. Now we will iterate through each cell in the puzzle and if the value is not 0 we can attempt to add this number to it's corresponding row, column and 3x3 grid hashSet. If for example the cell coordinate was (3,4) and the value was not 0, we would insert the value into hashSets row[3], column[4] and grid3x3[4]. If any of these .add() methods return a false because the value is already in the hashSet, we return false for the isValid() method and we now know the puzzle is not valid. If we iterate through the entire puzzle and we do not encounter any false returns we simple return true since this means non of the rules of sudoku have been violated and our puzzle is valid.

1.5 Review

This part of the assignment was relatively straightforward. From our weekly workshops I had already become familiar with hashSets and it's methods, so once I connected the similarities between a hashSet and a sudoku row/column/3x3 grid, it was quite a simple process.

2 Task 2: The Solver Class

2.1 Method *solve()*

This method follows the pseudocode provided to us in the brief. We iterate through the puzzle till we encounter an empty cell represented by '0'. Then we set the cell to '1' and check if it is valid. If it is, we recursively call solve() continuing to the next cell and repeat. If at any point the grid isn't valid we try the next value '2,3,4'...etc. If we find no value makes the puzzle valid for a cell we have made a mistake previously and we go back and change the previous cell - this is backtracking. We repeat this process until we reach the end of the puzzle. If we get to the end of the puzzle we have solved the puzzle.

2.2 Review

Thanks to the pseudocode given this was an easy task. All that was required was to simply follow the steps given and implement the correct methods. It compiled first time and passed all the tests.

The backtracking algorithm should have a worst case runtime of:

$$f(n) = O(g(n^m)) \tag{1}$$

Where '*n*' is the maximum in the range of values you can enter into a cell (in this case range is 1-9, so 9) and '*m*' is the number of empty cells in the puzzle.

Though I went to great lengths to improve the runtime of the program, I was unable to create a *solve()* method that would pass the given tests. However I do believe my thought process and logic would result in a significantly faster average runtime if implemented correctly.

To improve the basic backtracking algorithm lets first assess it's weaknesses. The first thing would be to recognize it will attempt to validate all values until it finds a valid one. This is unnecessary. To work around this in my own implementation I made a separate class '*Cell*' and had it store the information associated with an empty cell. This information included it's x y coordinates, how many possible values could be entered based on it's corresponding rows, columns and 3x3 grid, and what those possible values were. Inside the *Solver* class I created an instance variable for an ArrayList for '*Cell*' objects and then created a method that would scan the puzzle, locate every empty cell and then create a '*Cell*' for that empty cell and give it it's x y coordinates, how many values could be entered into it, and what those values were. This was done again like in validation with hashSets for each row, column and 3x3 grid. Using a coordinates corresponding hashSets will eliminate possible values and the remaining values that aren't in those corresponding hashSets must be the only possible values to try.

The second thing to recognise is that the backtracking algorithm will attempt each cell one by one iterating down the columns one by one. When puzzles start with cells that have many possible values that can be entered, we are going to have instances where we have to backtrack all the way to the start many more times than is necessary. When I first ran through all 400 puzzles it took several hours on one puzzle because I was going through the puzzle row by row instead of column by column. This meant that all the cells on the first few rows that had 9/8/7...etc. possible values greatly increased our runtime. Fixing that was simple as all I had to do was scan by columns instead of rows, however there will be puzzles where using column major will encounter the same problem. To work around this we can solve cells that have the least amount of possible solutions first, thereby drastically reducing the possible solution trees we will run down. In some puzzles you will have many cells with only one possible value. Entering these and updating the grid will then change other cells to only having one possible value and so on. This compounding affect would have a large impact on the runtime.

3 Task 3: Solving Puzzles

3.1 Method *main()*

This is the programs main method and is very straight forward. We create a for loop to run through and solve each of the 400 puzzles and timers within the loop to time how long it takes to solve each one.

3.2 Review

After running the program we can see that the puzzles that have only 20 gaps, and the puzzles that have 69 gaps can often times have a very similar solve time. This is due to how the backtracking solves each cell in sequence and how the puzzles are set up. It would be very easy to create puzzles that would take the program an exceedingly long time to solve regardless of the gap count, simply by understanding how the algorithm goes though the puzzle and where to place gaps with 9 possible valid values. Likewise you can have the program solve puzzles with 69 gaps instantaneously by having all of it's troublesome gaps at the end of the puzzle. For this algorithm these are the main factors that will affect it's runtime. Having said that, given a totally random puzzle we should still expect a worst case runtime of $f(n) = O(g(n^m))$.

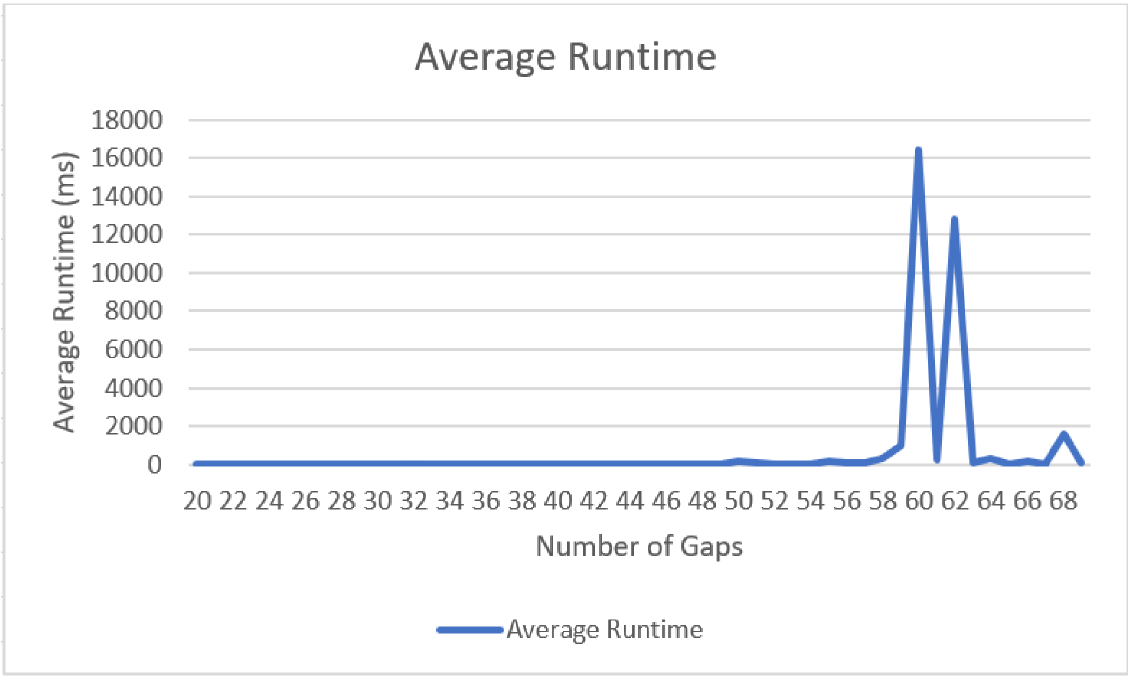
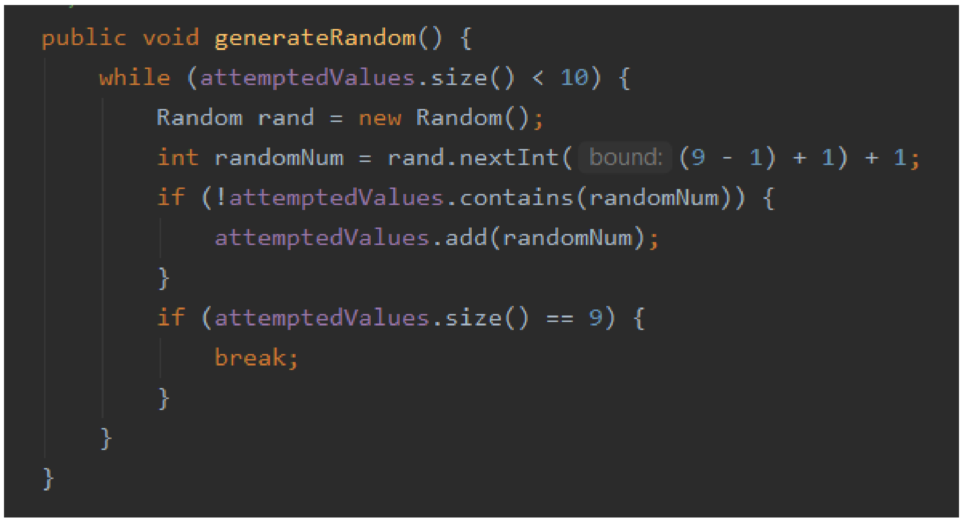


Figure 1: Runtime Graph using standard backtracking algorithm. As you can see, the greater the gaps the greater the chance of hitting a puzzle that has a large runtime. It's key to note that more gaps do not not necessarily mean greater runtime.

4 Task 4: Generating Puzzles

To generate a solvable sudoku puzzle I first created a new puzzle in Examples and set all it's cells values to 0. Now we have an empty grid. Next I made a method to fill an ArrayList with random values from 1-9. This method will only add values if the list doesn't already contain said value.



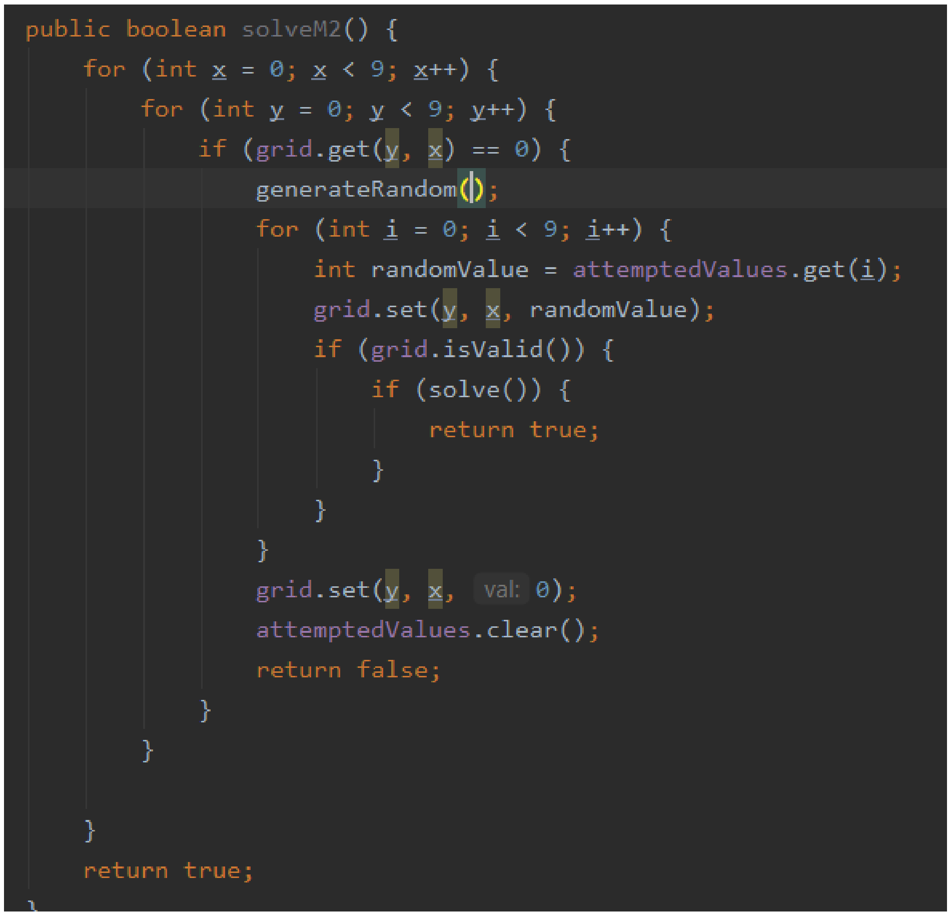
```java
public void generateRandom() {
    while (attemptedValues.size() < 10) {
        Random rand = new Random();
        int randomNum = rand.nextInt( bound: (9 - 1) + 1) + 1;
        if (!attemptedValues.contains(randomNum)) {
            attemptedValues.add(randomNum);
        }
        if (attemptedValues.size() == 9) {
            break;
        }
    }
}
```

Figure 2: generateRandom() method

Next we can slightly modify the *solve()* method to enter values from our ArrayList. This continues until the puzzle is filled. This is a valid puzzle.

Finally, to remove values I used another new method *removeValuesAtRandom(int gaps)* that takes an integer for the amount of gaps you want in your puzzle. Two random numbers are generated in this method to be entered as our random coordinates and then we set that cells value to 0. This loops until we have removed the requested amount of values.

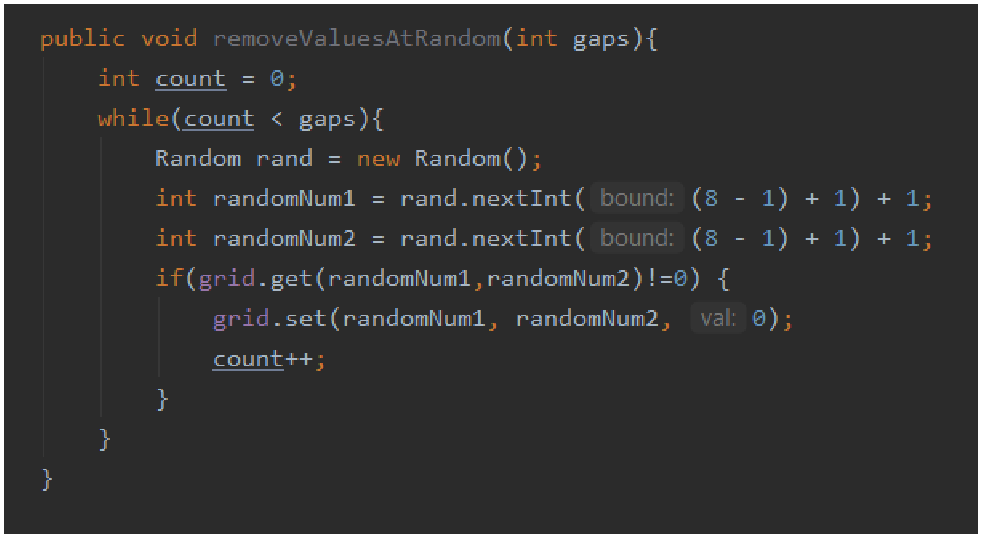
I have not created a class for this as requested but the methods exist within the Solver class and have been commented out in main. Make sure there is an empty puzzle in 'Examples' and set the above for loop to that puzzle if you wish to see my monstrosity in action.



```java
public boolean solveM2() {
    for (int x = 0; x < 9; x++) {
        for (int y = 0; y < 9; y++) {
            if (grid.get(y, x) == 0) {
                generateRandom();
                for (int i = 0; i < 9; i++) {
                    int randomValue = attemptedValues.get(i);
                    grid.set(y, x, randomValue);
                    if (grid.isValid()) {
                        if (solve()) {
                            return true;
                        }
                    }
                }
                grid.set(y, x,  val: 0);
                attemptedValues.clear();
                return false;
            }
        }

    }
    return true;
```

Figure 3: SolveM2() method



```java
public void removeValuesAtRandom(int gaps){
    int count = 0;
    while(count < gaps){
        Random rand = new Random();
        int randomNum1 = rand.nextInt( bound: (8 - 1) + 1) + 1;
        int randomNum2 = rand.nextInt( bound: (8 - 1) + 1) + 1;
        if(grid.get(randomNum1,randomNum2)!=0) {
            grid.set(randomNum1, randomNum2,  val: 0);
            count++;
        }
    }
}
```

Figure 4: removeValuesAtRandom(int gaps) method

5 Evaluation

For this assignment the biggest hurdle I found was trying to improve the algorithm for solving. While I believe all my logic on how to improve it is sound, I as someone only a year into programming had difficulty translating my thoughts into code.

That being said the code I do have is clean, readable and well commented. All tasks do function as requested, (aside from Task 4 since it doesn't have it's own class, though it does complete the requested function with some adjustments to the main method and Examples class). Furthermore my report is clean and concise and does a good job explaining why I made certain decisions and what each part of the program does.

I believe a grade of at least 70 would be fair.