

Sudoku Solver using Stacks

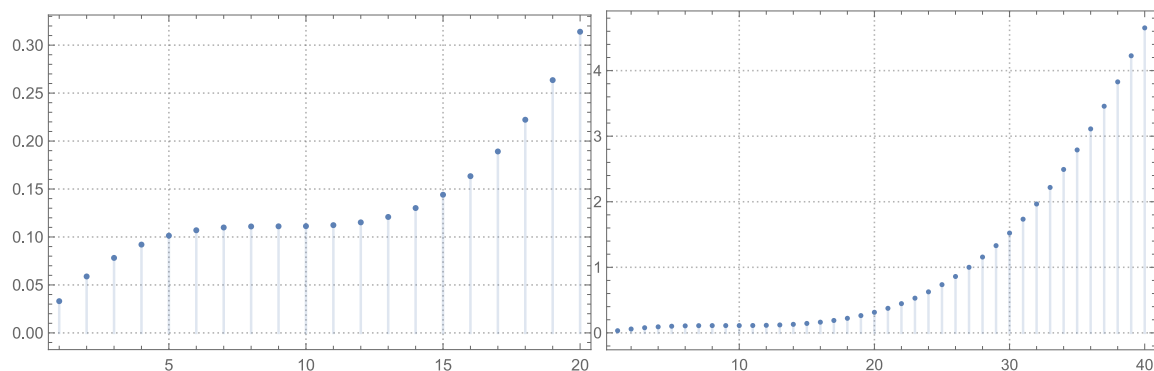
Abstract

In this project, we designed and implemented a sudoku solving algorithm and visualizer using Stacks, our modified LinkedLists, as a fundamental data structure. The core purpose of this project was to explore the implementation of a stack-based Sudoku solver using a depth-first search algorithm. Furthermore, we explored the relationship between the number of randomly selected initial constraint values and the likelihood of finding a solution. As well as how the number of initial conditions/constraints correlates to the time taken to solve the board. Taking this further I endeavored to create and implement a considerably faster algorithm, based off of concepts discussed in Donald Knuth's "Dancing Links" paper, but fell just short of doing so. The files will be included, and the ideas explained as I spent probably a minimum of 30 hours on the algorithm's implementation. In any case to put the results succinctly as the number of initial filled in cells increased it became considerably slower and more difficult to generate initial boards that were valid. However, these boards, once reached, by and large had no solution, and were incredibly quick to solve/deem solution-less. One exception to this being empty boards and near-empty boards which were trivial to both generate and solve. On the other hand, the easiest and quickest to generate boards were those with less than 10 entries. However, these were generally had the potential to be the most time and labor-intensive to solve, if solvable at all.

Results

Effect of Initial Constraints on Board Generation

As the quantity of initially filled in cells, or constraints as I will refer to them, increased the time required to generate a board increased substantially. The likelihood that at least one of "n" filled cells on the sudoku board will share a row, column or subgrid with a cell of the same value is given by the function (high chance this formula is wrong, probably not too wrong though, similarly pattern likely to hold due to inflection points being probably similar) $\frac{1}{9} \left(\frac{n}{3} - \frac{n^2}{27} + \frac{n^3}{729} \right)$, as such by the point that $n = 27$ and $\frac{1}{9} \left(\frac{n}{3} - \frac{n^2}{27} + \frac{n^3}{729} \right) = 1$. It is practically guaranteed that the board will have to be redrawn at least once to generate a valid sudoku. (To be honest I am horrible at probability, I've never taken a class or interest in the subject so I might be wrong here.)



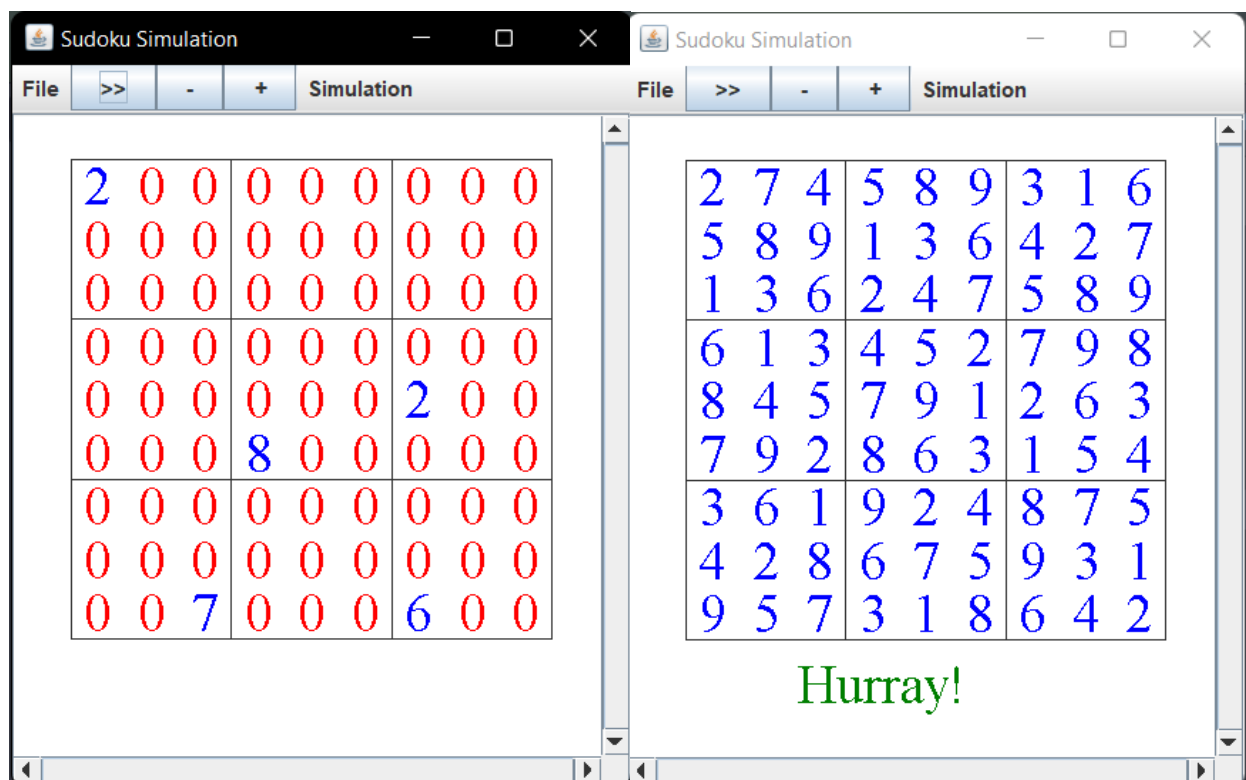
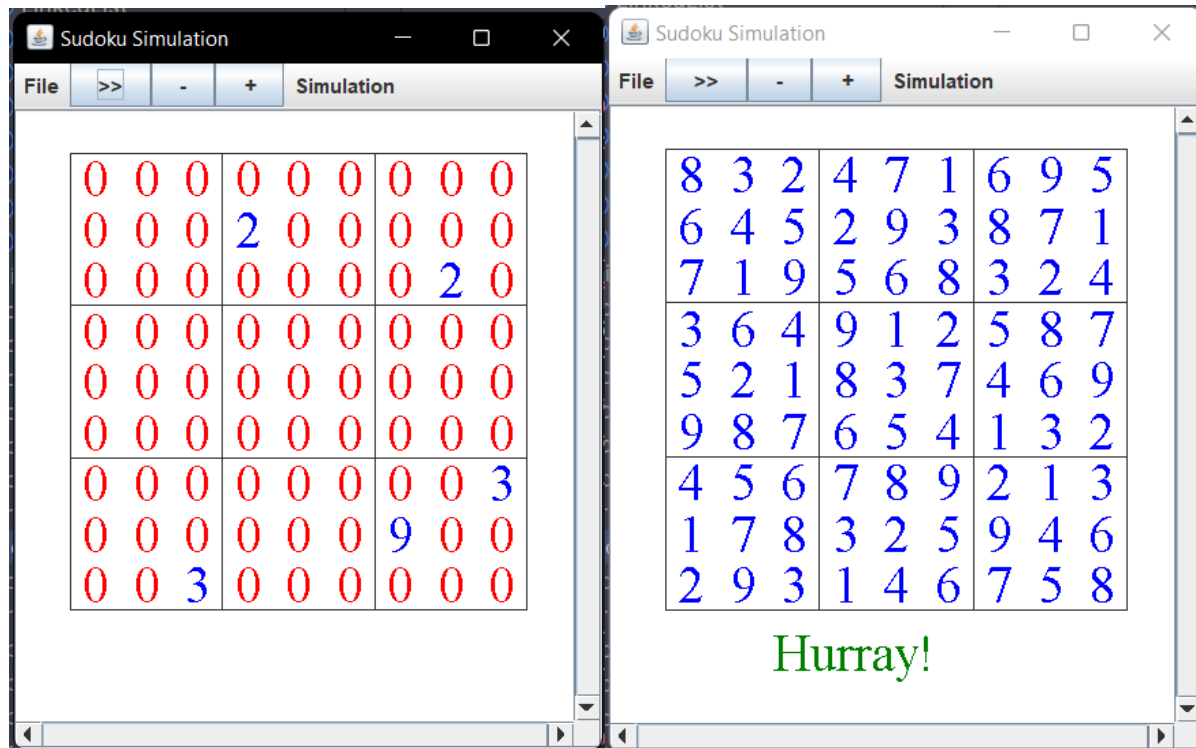
Effect of Initial Constraints on Board Solving

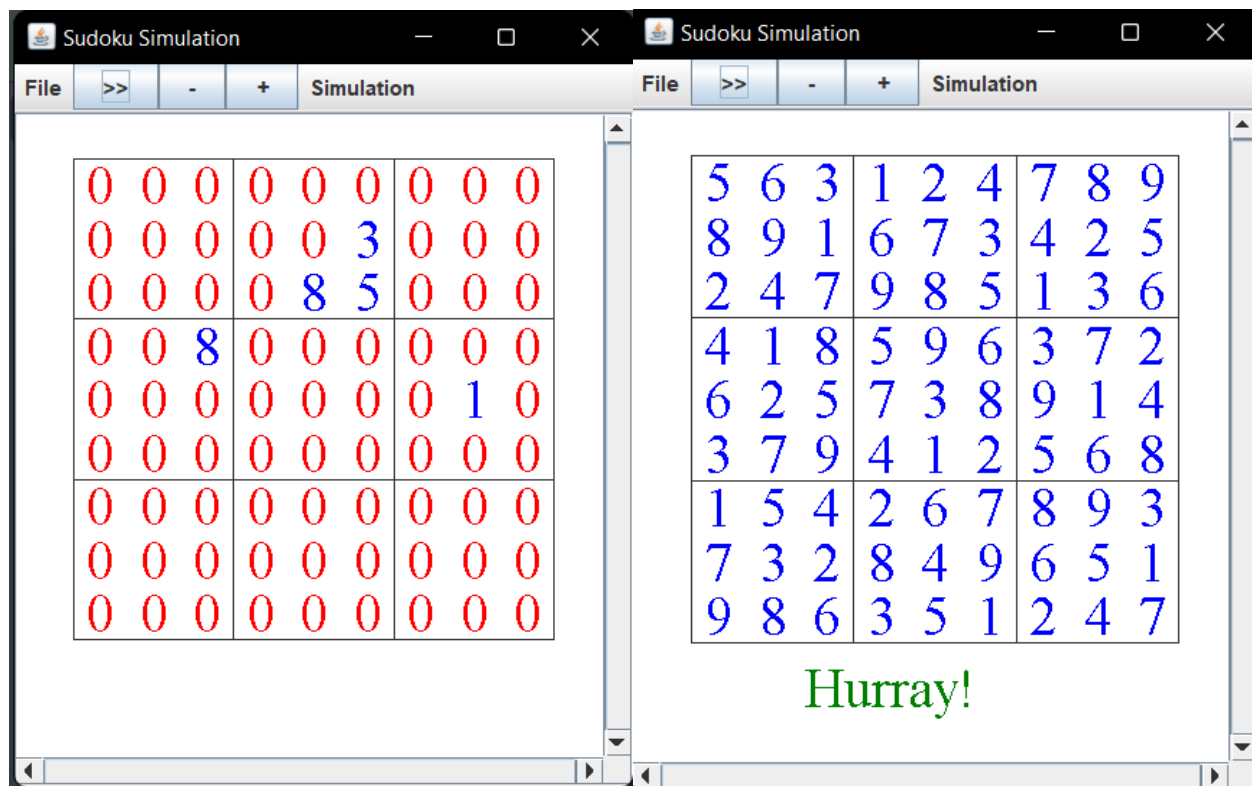
Conversely, the trend noted during board generation, as initial counts increased, the time to either solve the board or prove that no solution existed decreased. This is save for particularly small values of initial constraints where the quantity of free variables makes it such that if a solution is not reached early on during the algorithm, the solution space is so vast that the time required to extricate every other possible solution is tremendously long. One such 5-initial constraint sudoku ran continuously for 11 hours without reaching any solution.

On the other hand, Sudoku boards with many initial constraints generally lacked solutions, but were very quickly deemed as such. The time to reach a solution is essentially solely a function of the initial constraints. However, it is not the number of these constraints alone which determine this time. The nature of the constraints themselves determines computation time. Having a lesser number of initial constraints makes it such that there are more possible valid states of the board and many more possible combinations to check. This means that the algorithm is more likely to come across a possible solution, but in the worst case has to check many more possible states. As such, here is where the specific constraints come into play. Depending on the constraints the puzzle may have more or less solutions. The minimal number of filled cells to create a “no solution” board is 15, the minimal number of filled cells to ensure a single solution is 17. It is mathematically impossible to place more than 61 cells such that at least one cell is unconstrained (i.e. not in the same subgrid, row, or column as another cell). The likelihood of this extreme configuration arising naturally is equal to $\frac{61! 21!}{81!}$. Tremendously low odds. The likelihood of two randomly placed cells on an empty board being in the same row, column or subgrid is 1 in 4. The likelihood of each subsequent cell being constrained in some way, or another only goes up. As such the likelihood of reaching a solution quickly is a product directly proportional to the quantity of constraint cells and inversely proportional to the degree to which the constraints limit the system. What that means is that a quick solution is most likely to be met when the solution set is large, but the permutation set of the initial board is small. Regardless, statistically speaking, if a solution exists this algorithm is most likely to reach it quickly if the number of constraint cells is very small or very large, if no solution exists this algorithm will encounter a worst case run on very small constraint boards and may run forever (septillions of permutations), or be very quick on large constraint boards.

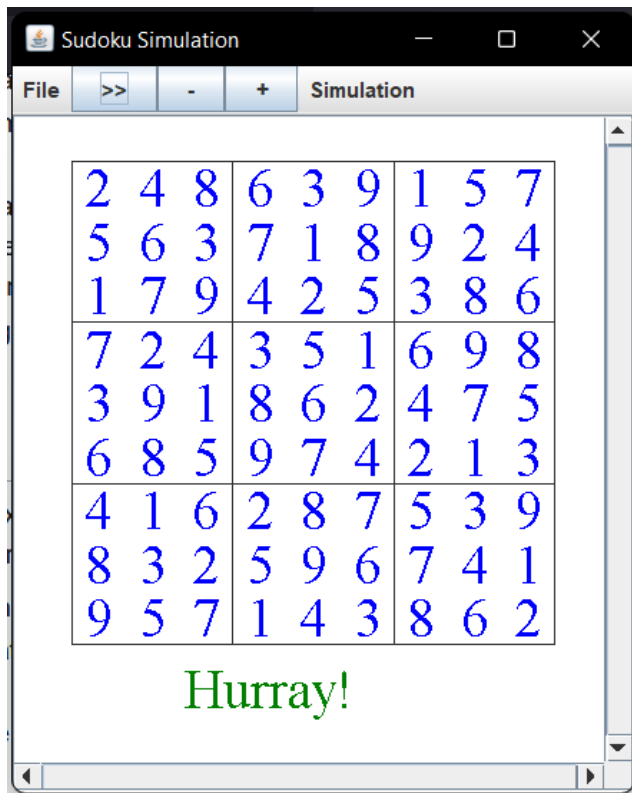
Here are the five cell solutions:

These first three were quickly generated.

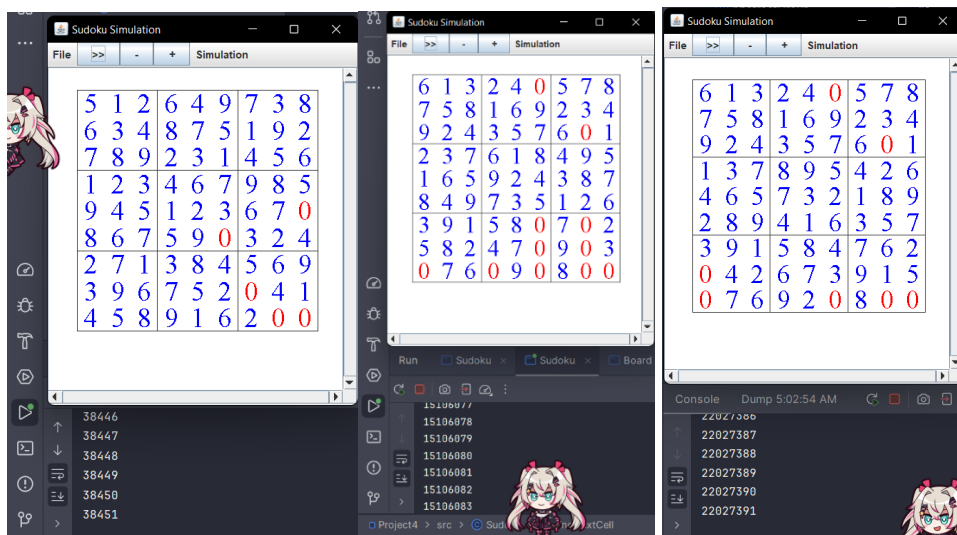




This solution took upwards of 10 minutes to reach:

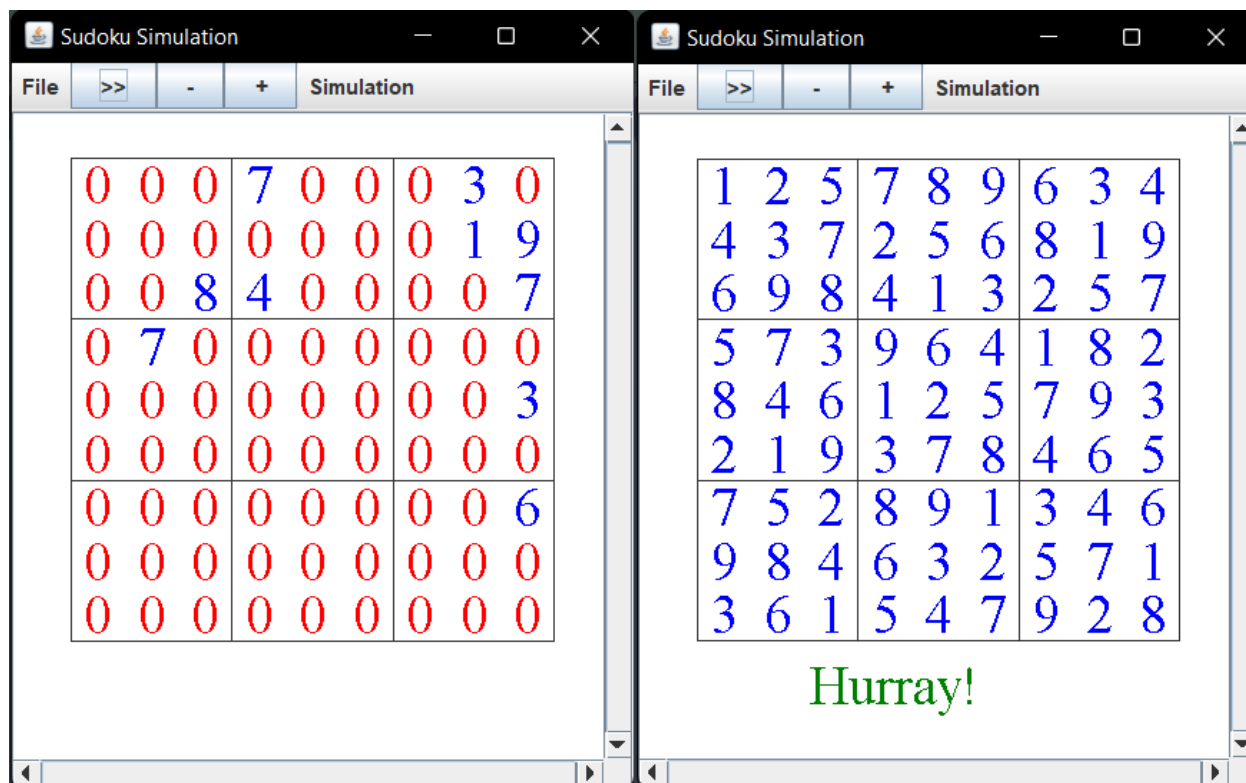
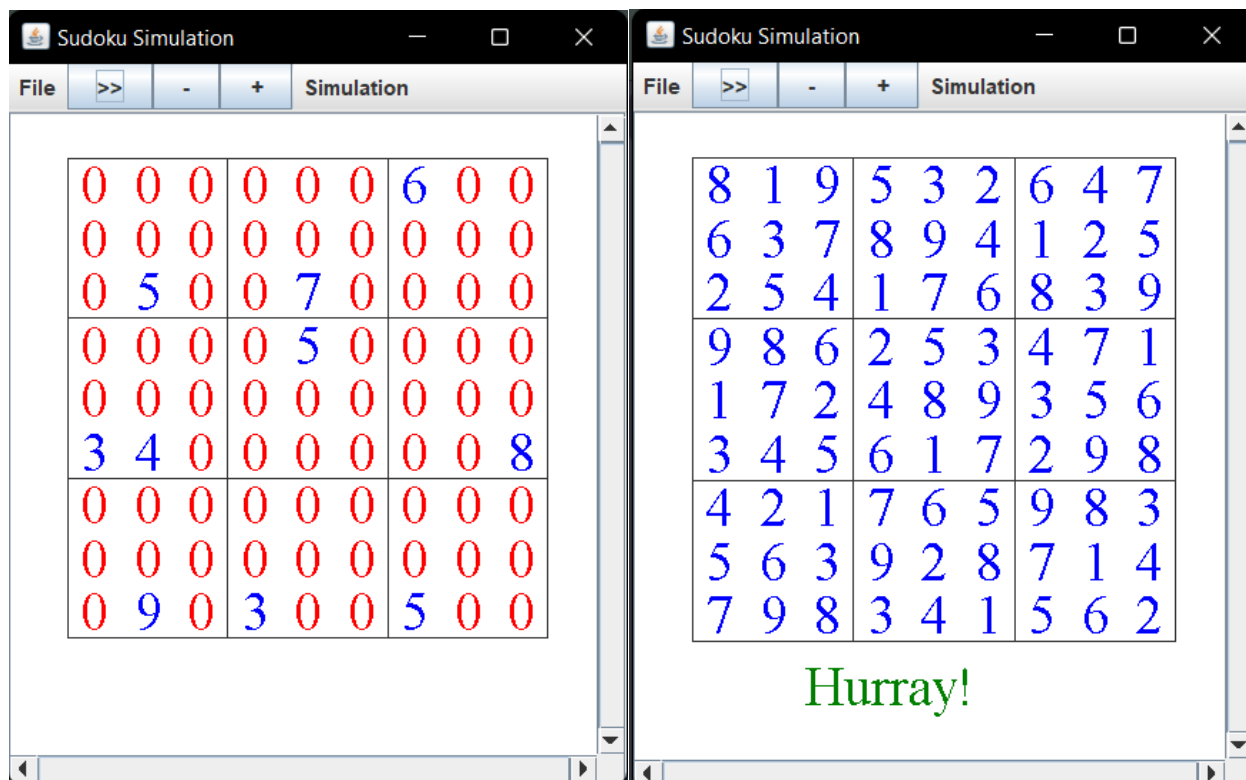


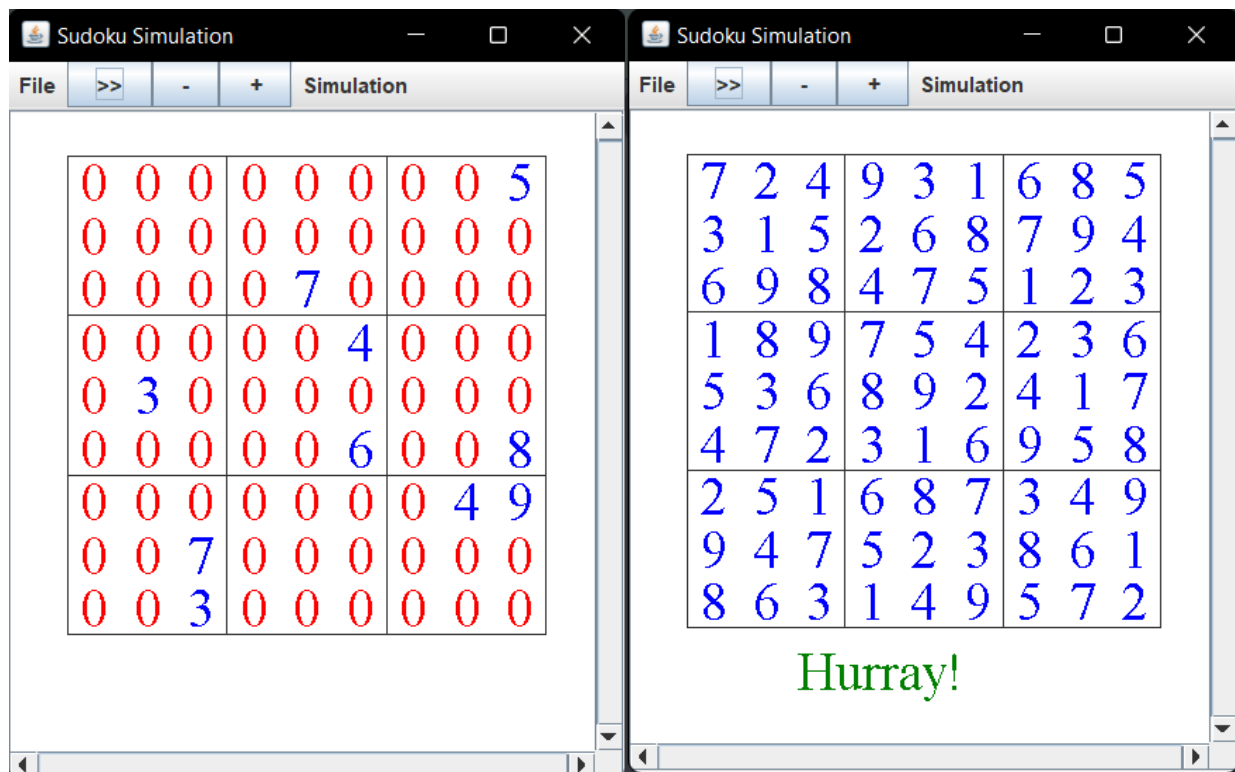
Whereas this solution is yet to be reached even after over eleven hours of continuous processing with over 40 million backtraces.



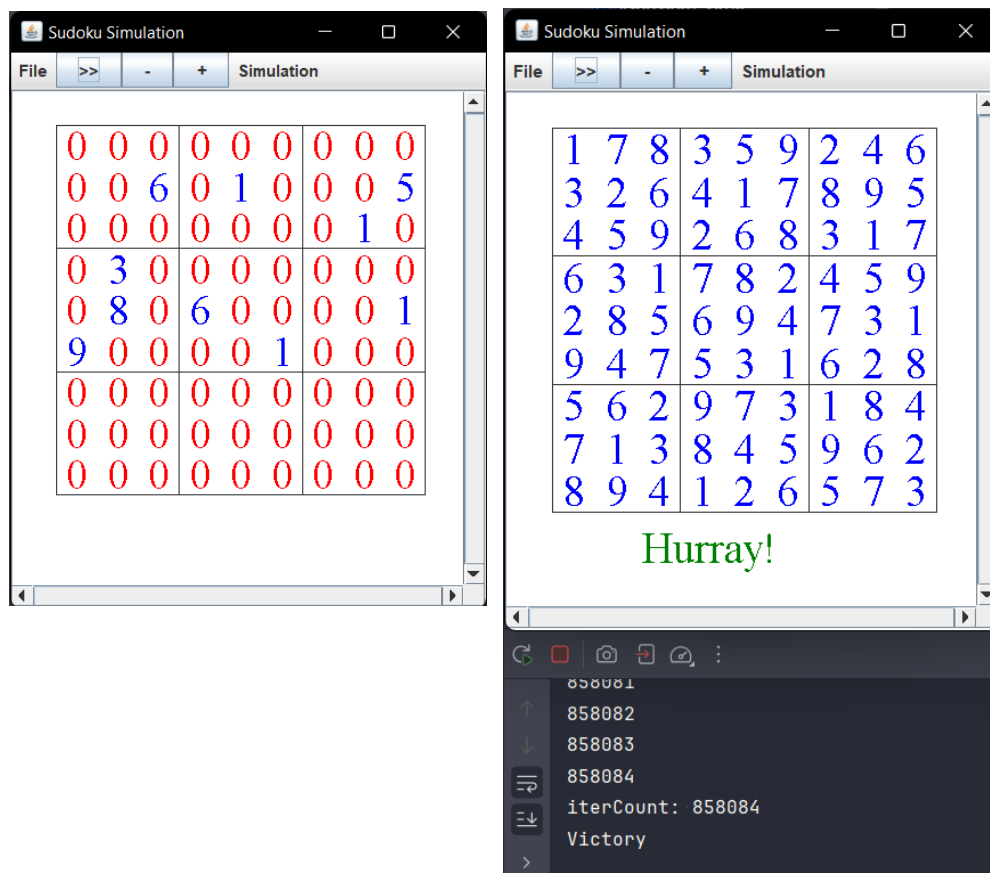
Here are the ten cell solutions:

These first three were quickly generated.

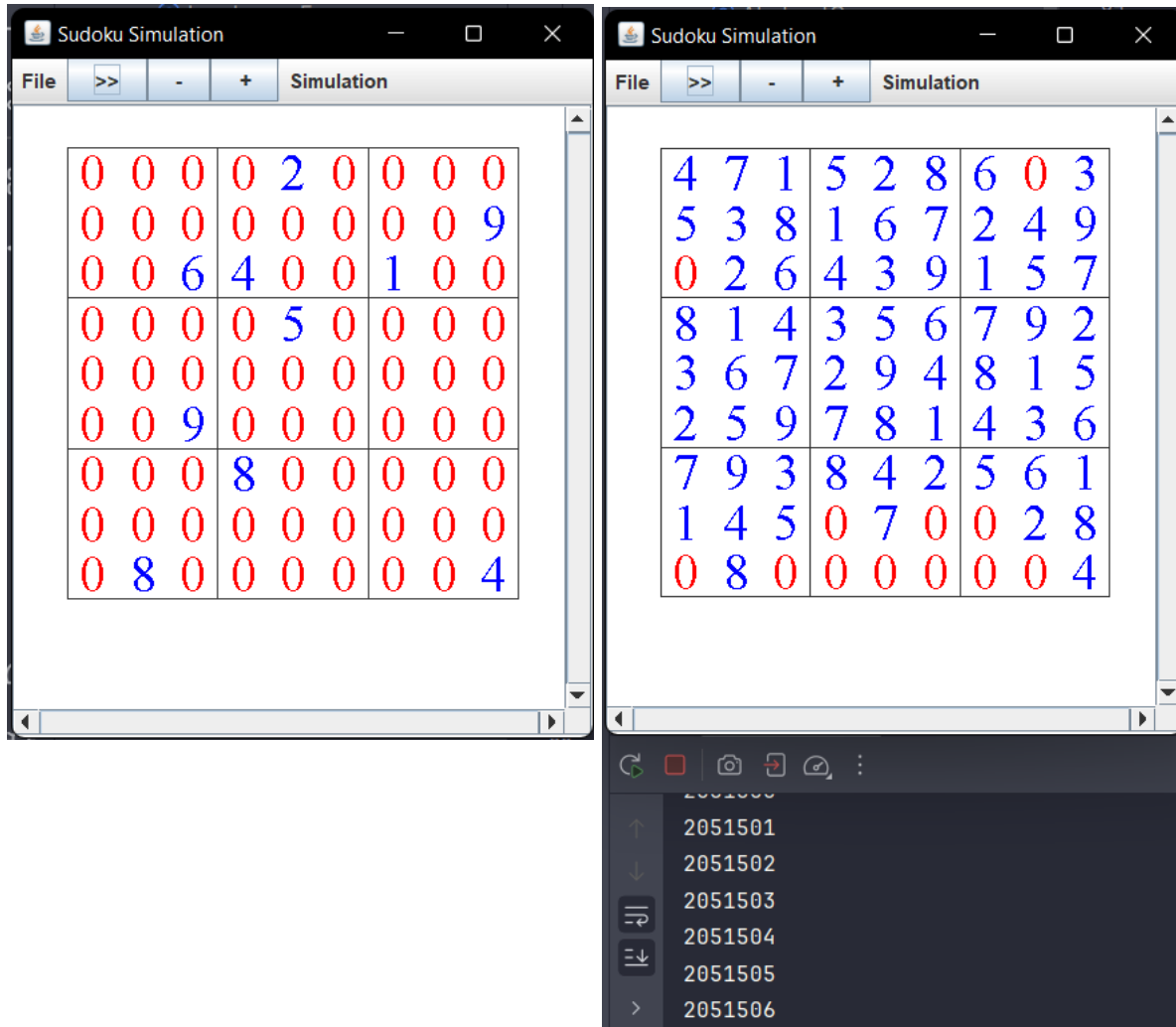




The following board was only able to reach a finished state after 858,084 backtraces.

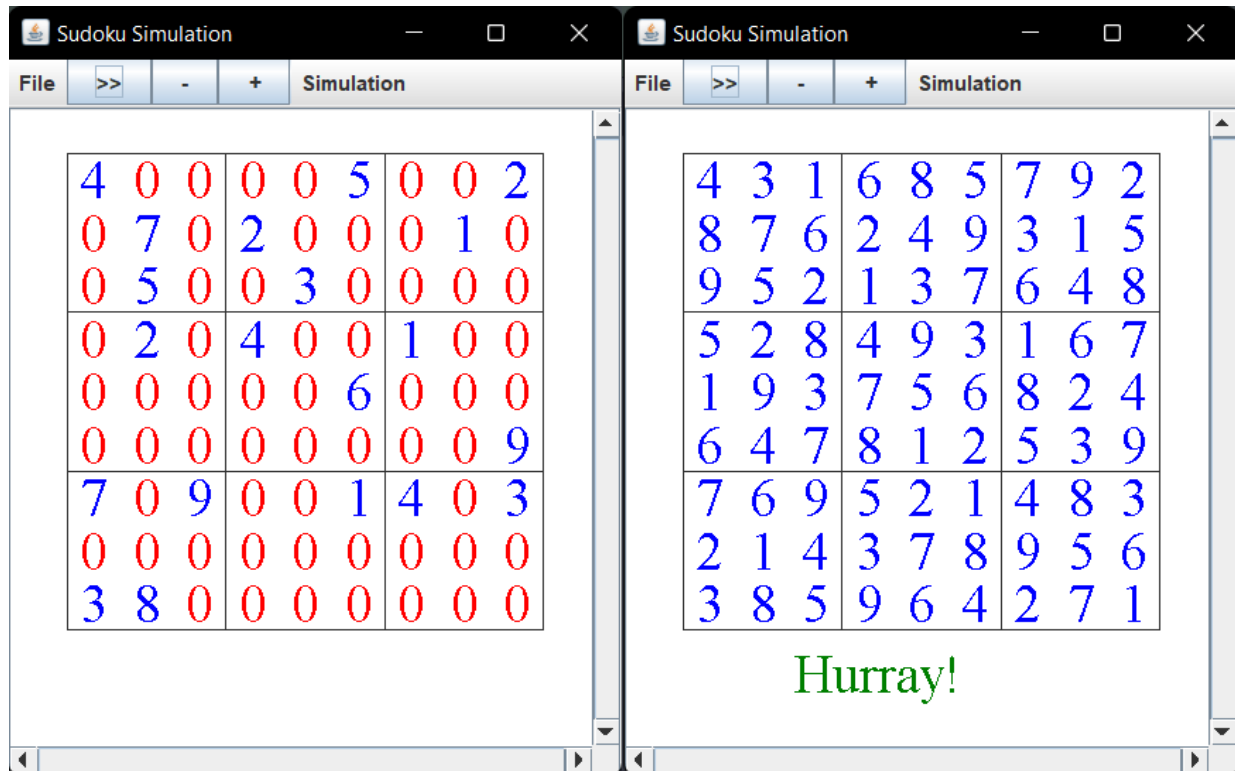


Finally, this last board was unable to reach a finished state even after over two million backtraces; however, to free up both threads and memory I halted its execution after backtraces reached this mark. (As an aside, between concurrent runs of this program and the many other windows I had open my memory utilization was at roughly 64 gigabytes. Using Process Monitor I managed to discern that each program run uses a maximum of 136 concurrent threads and about 1700 megabytes of memory. Luckily, CPU usage never exceeded 20% even with other programs running and concurrent simulation.)

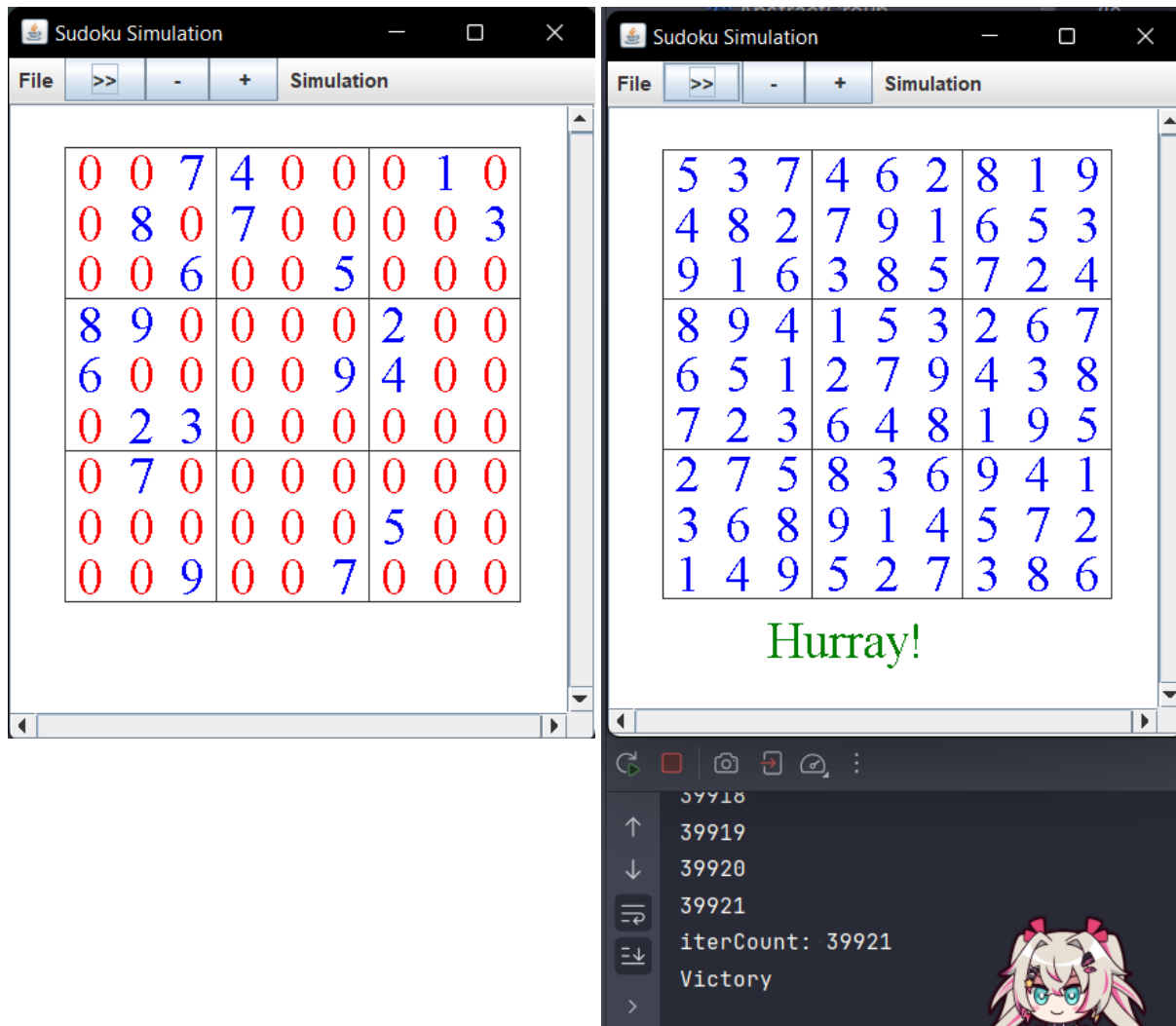


Here are the twenty cell solutions:

The first was quickly generated.



The following two were generated moderately quickly, sub 50,000 backtraces.



The left window shows a 9x9 grid with the following values (red numbers are in the original image, blue numbers are in the original image):

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 7 | 4 | 0 | 0 | 0 | 1 | 0 |
| 0 | 8 | 0 | 7 | 0 | 0 | 0 | 0 | 3 |
| 0 | 0 | 6 | 0 | 0 | 5 | 0 | 0 | 0 |
| 8 | 9 | 0 | 0 | 0 | 0 | 2 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 9 | 4 | 0 | 0 |
| 0 | 2 | 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 |
| 0 | 0 | 9 | 0 | 0 | 7 | 0 | 0 | 0 |


The right window shows a completed 9x9 grid with the following values (all blue numbers in the original image):

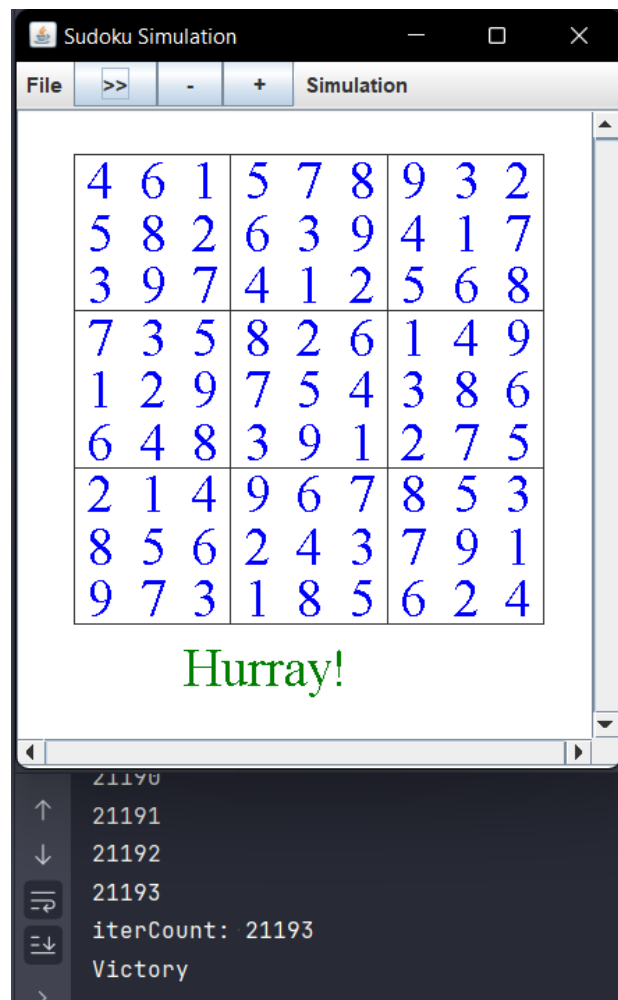
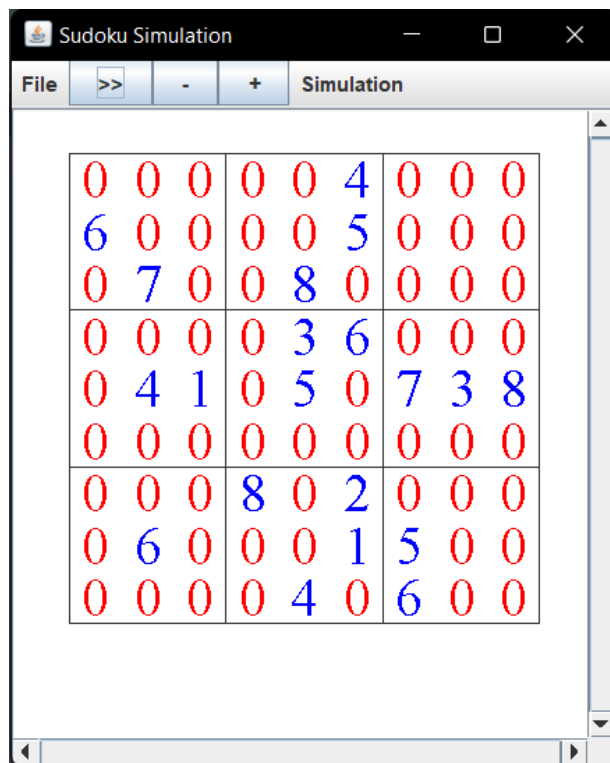
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | 7 | 4 | 6 | 2 | 8 | 1 | 9 |
| 4 | 8 | 2 | 7 | 9 | 1 | 6 | 5 | 3 |
| 9 | 1 | 6 | 3 | 8 | 5 | 7 | 2 | 4 |
| 8 | 9 | 4 | 1 | 5 | 3 | 2 | 6 | 7 |
| 6 | 5 | 1 | 2 | 7 | 9 | 4 | 3 | 8 |
| 7 | 2 | 3 | 6 | 4 | 8 | 1 | 9 | 5 |
| 2 | 7 | 5 | 8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

Hurray!

Terminal output:

```
39918
↑ 39919
↓ 39920
↺ 39921
iterCount: 39921
Victory
```





This next board generated very slowly, over two million backtraces; however, a solution was finally reached.

The image displays two side-by-side windows of a 'Sudoku Simulation' application, showing the progression from an initial puzzle to a solved state.

Left Window (Initial Puzzle):

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 8 | 3 | 0 | 1 | 0 | 0 | 6 |
| 0 | 9 | 3 | 0 | 0 | 0 | 8 | 0 | 0 |
| 5 | 0 | 0 | 9 | 0 | 0 | 7 | 0 | 0 |
| 0 | 0 | 0 | 2 | 9 | 4 | 3 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 |
| 3 | 0 | 0 | 6 | 0 | 0 | 4 | 0 | 0 |

Right Window (Solved Puzzle):

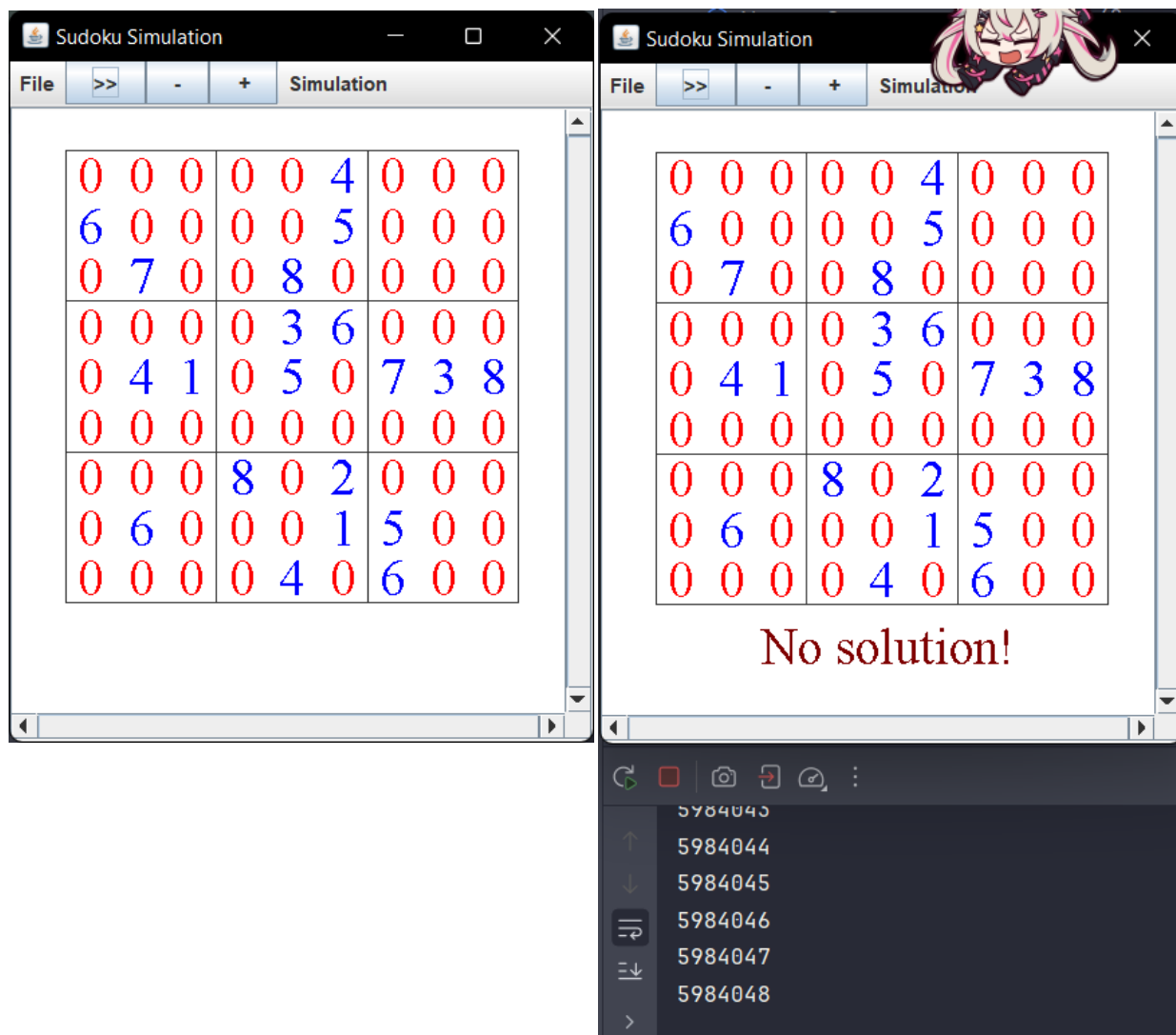
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | 5 | 2 | 1 | 3 | 6 | 9 | 4 | 8 |
| 8 | 3 | 1 | 4 | 2 | 9 | 6 | 7 | 5 |
| 9 | 4 | 6 | 7 | 8 | 5 | 1 | 3 | 2 |
| 2 | 7 | 8 | 3 | 4 | 1 | 5 | 9 | 6 |
| 6 | 9 | 3 | 5 | 7 | 2 | 8 | 1 | 4 |
| 5 | 1 | 4 | 9 | 6 | 8 | 7 | 2 | 3 |
| 1 | 8 | 5 | 2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

Hurray!

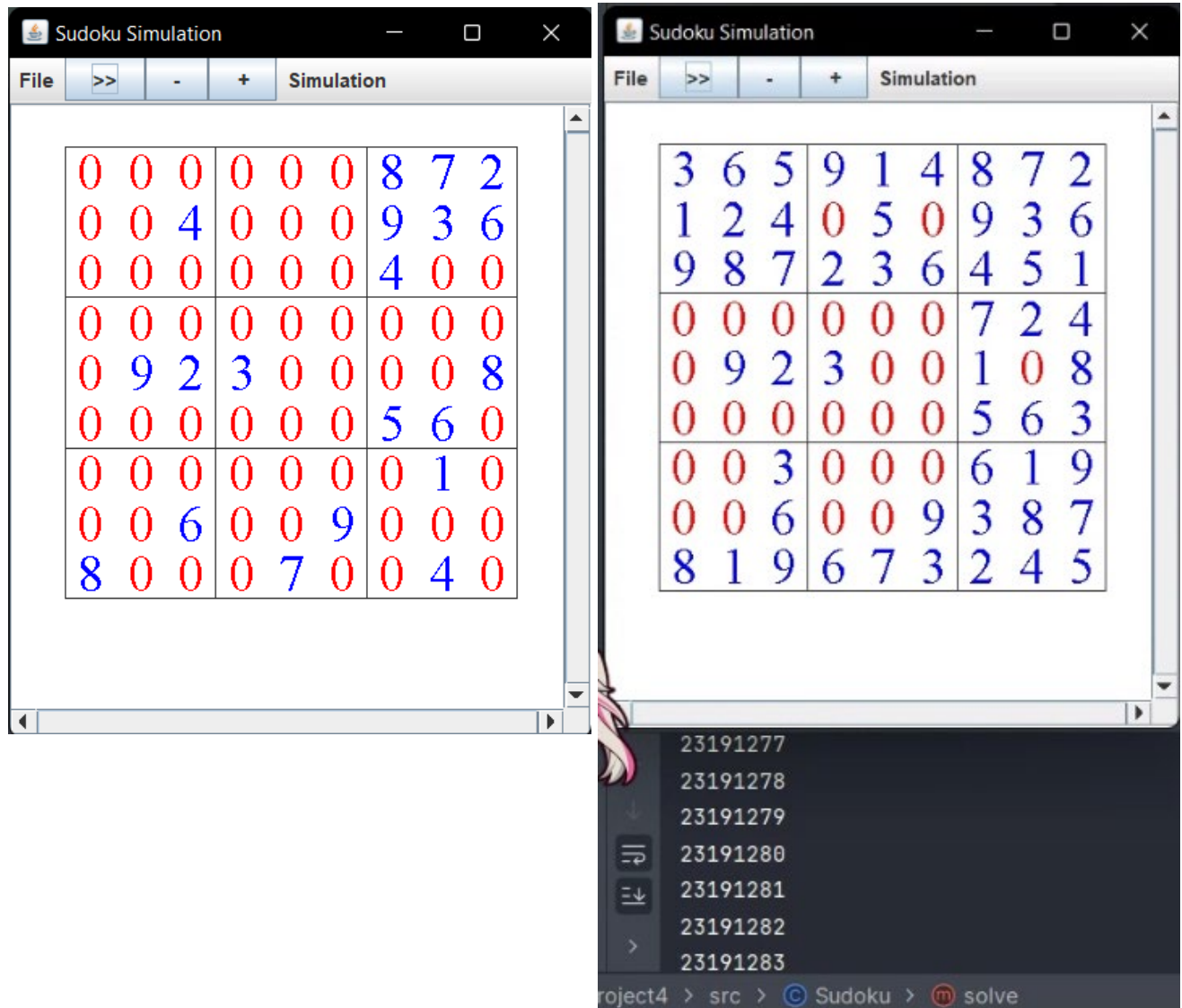
Terminal Output:

```
2025144
2025145
2025146
2025147
iterCount: 2025147
Victory
```

The next board required an inordinate amount of time but, finally after upwards of 5.5 million backtraces the algorithm proved that no solution existed.



Finally, this last board was unable to generate a solution over upwards of 22 million back traces.



The image displays two instances of the 'Sudoku Simulation' application and a terminal window. The left application window shows a 9x9 grid with the following values (red numbers are in red, blue numbers are in blue):

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 8 | 7 | 2 |
| 0 | 0 | 4 | 0 | 0 | 0 | 9 | 3 | 6 |
| 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 9 | 2 | 3 | 0 | 0 | 0 | 0 | 8 |
| 0 | 0 | 0 | 0 | 0 | 0 | 5 | 6 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 6 | 0 | 0 | 9 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 7 | 0 | 0 | 4 | 0 |

The right application window shows a 9x9 grid with the following values:

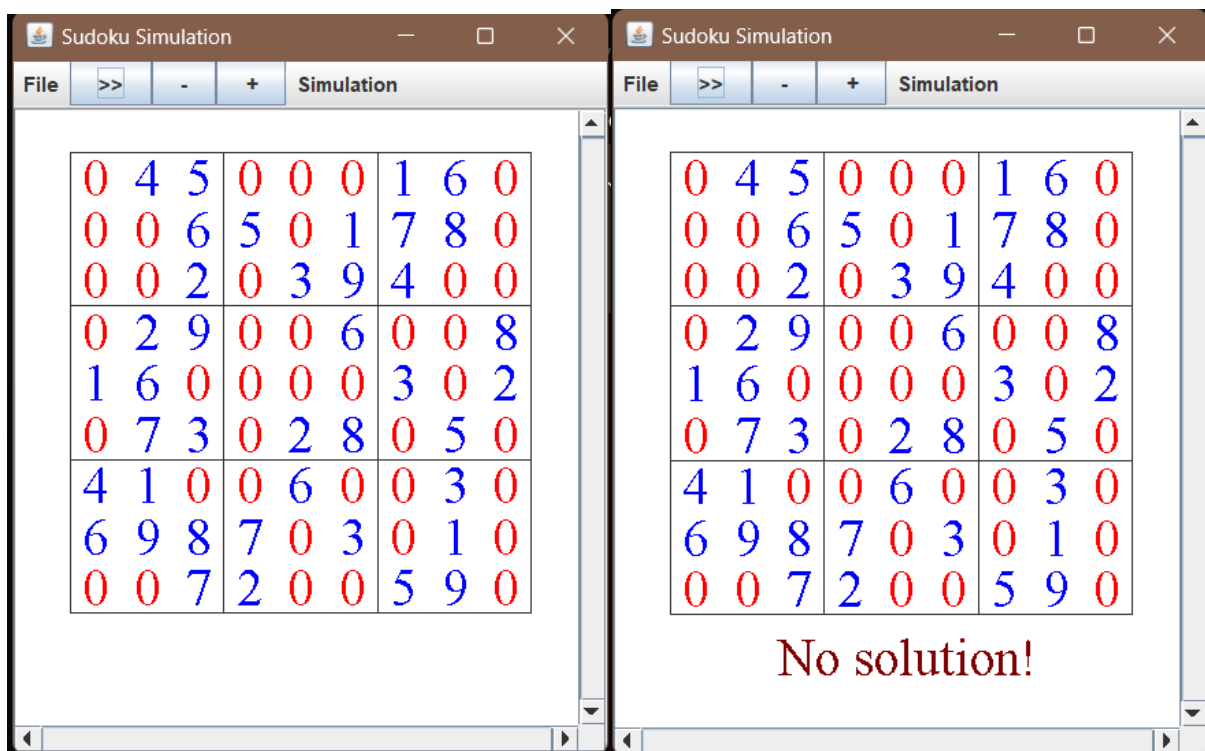
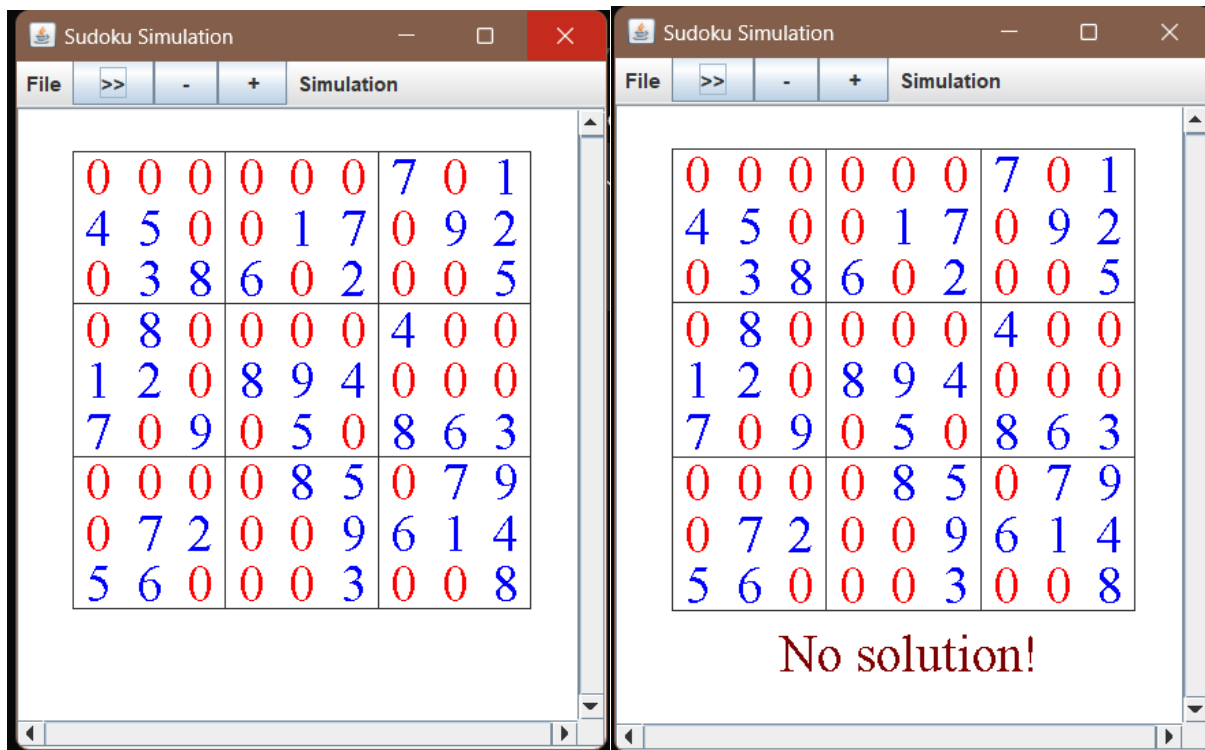
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 3 | 6 | 5 | 9 | 1 | 4 | 8 | 7 | 2 |
| 1 | 2 | 4 | 0 | 5 | 0 | 9 | 3 | 6 |
| 9 | 8 | 7 | 2 | 3 | 6 | 4 | 5 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 7 | 2 | 4 |
| 0 | 9 | 2 | 3 | 0 | 0 | 1 | 0 | 8 |
| 0 | 0 | 0 | 0 | 0 | 0 | 5 | 6 | 3 |
| 0 | 0 | 3 | 0 | 0 | 0 | 6 | 1 | 9 |
| 0 | 0 | 6 | 0 | 0 | 9 | 3 | 8 | 7 |
| 8 | 1 | 9 | 6 | 7 | 3 | 2 | 4 | 5 |

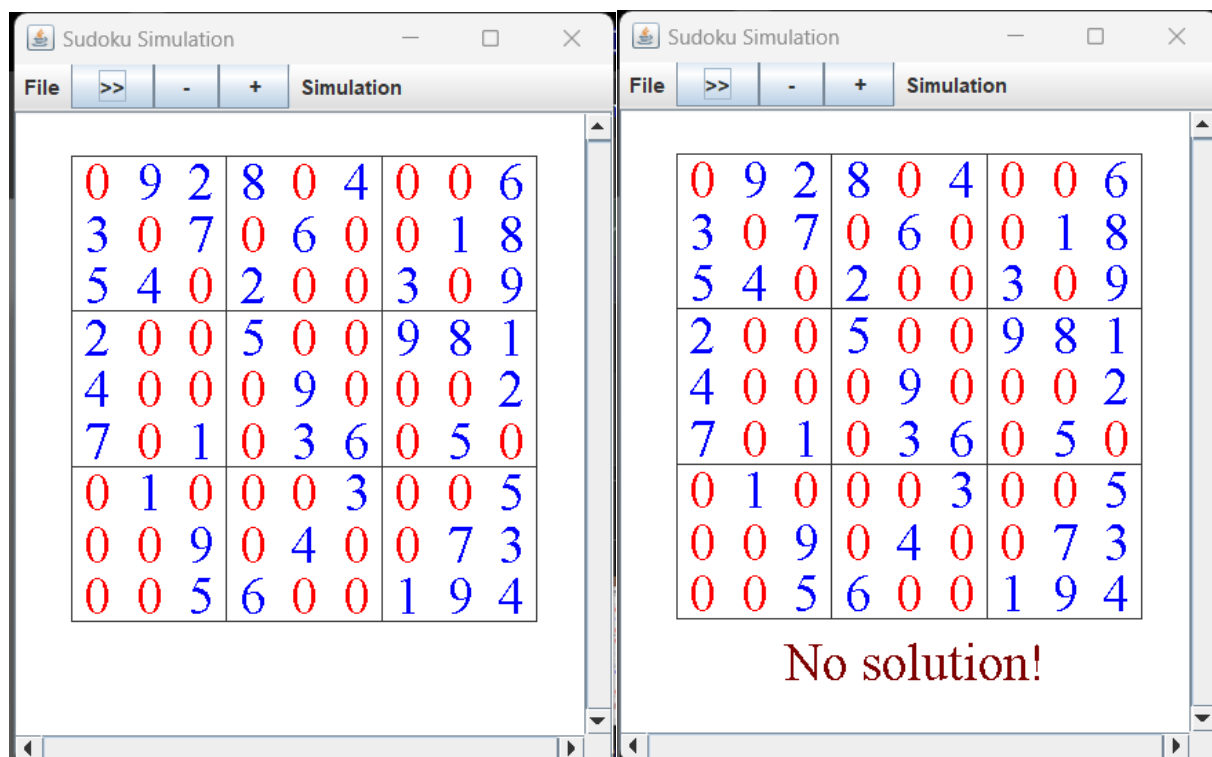
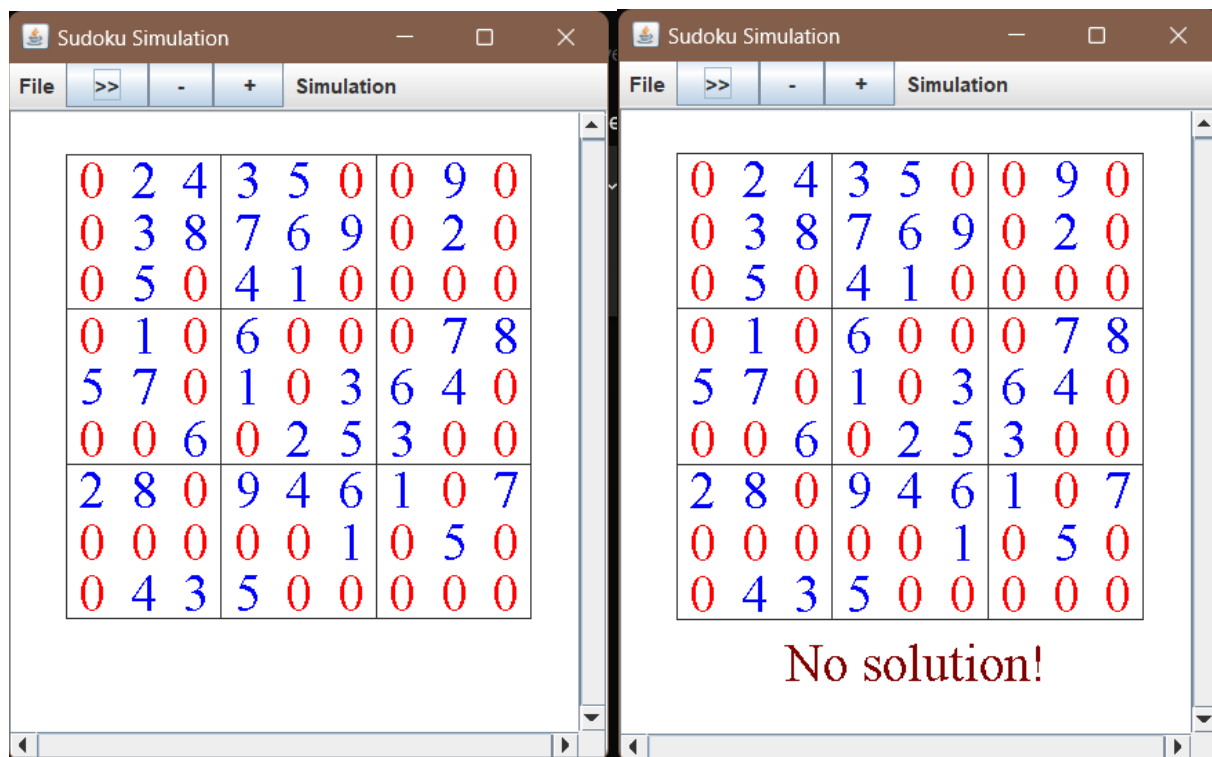
Below the right application window is a terminal window showing a list of numbers and a file path:

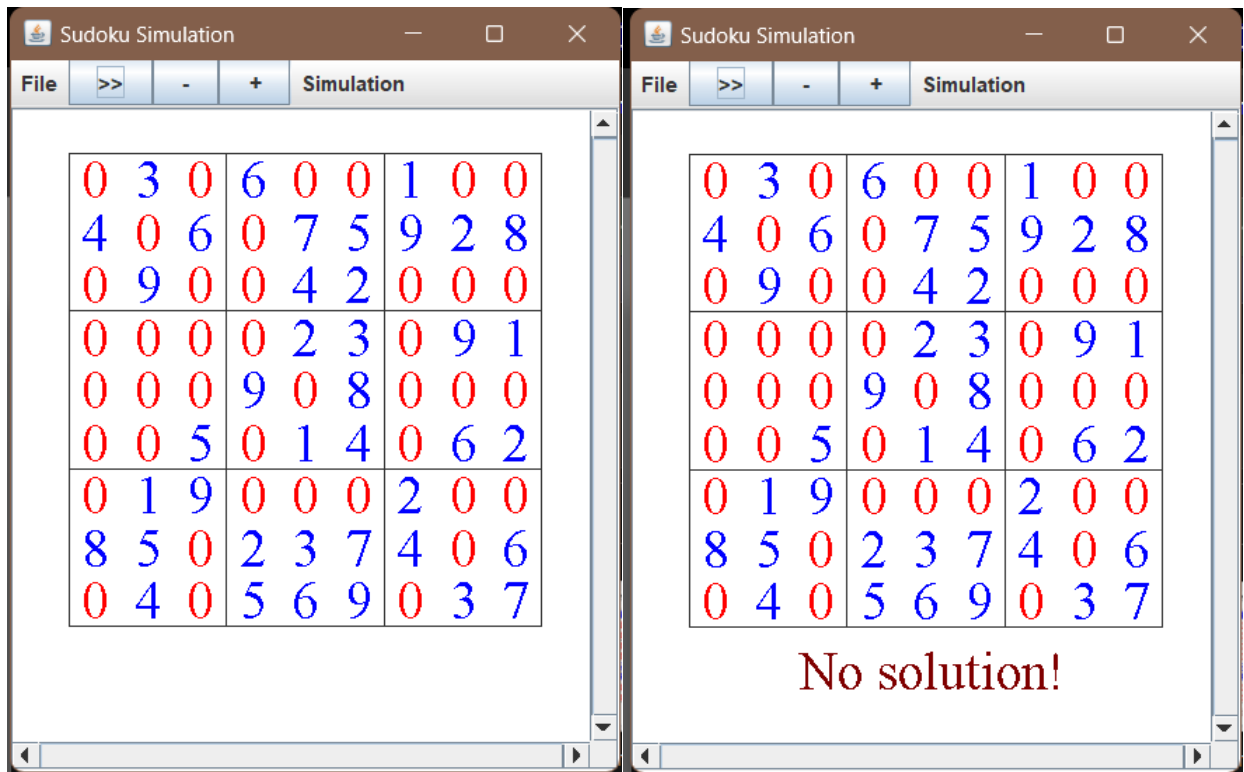
```
23191277
23191278
23191279
23191280
23191281
23191282
23191283
project4 > src > Sudoku > solve
```

Here are the 40 cell solutions:

All these boards terminated execution very quickly, none of which had a solution.







Extensions

1. Basic Graphical User Interface (GUI) with Optional Settings:

- **Implementation:** I added a graphical user interface (GUI) to my simulation, providing users with a more interactive and user-friendly experience. This GUI includes a couple optional settings that allow users to customize the simulation's behavior. Some of the optional settings could include parameters like selecting the file to generate from, the sleep time between updates, and a step-by-step solution.
- **Effect:** The GUI enhances the usability of the simulation. Users can slow down or speed up the simulation, see each guess step, and observe how custom boards are resolved. This feature makes the simulation more accessible and provides a way to experiment with different scenarios without altering the code. It also facilitates experimentation and analysis by enabling appreciable solve speeds.

2. Partial Dancing Links/Exact Cover Solver:

- **Implementation:** I coded an almost complete implementation of Donald Knuth's Algorithm X. This implementation is through the 'SudokuAlt' class. This class is my own version of an algorithm solving the exact cover problem based loosely on Donald Knuth's Dancing Links implementation of his Algorithm X as described by his paper on the subject. This class generates an incidence matrix, if one is not provided, and then is an almost functioning implementation of the

algorithm. It aims to solve Sudoku puzzles of any number of clues and of any complexity in a fraction of a second. Unfortunately, because of some pesky null pointers resultant from oversight in cloning during the implementation of the cover method this does not work perfectly. The fixes required to get this class functioning are minimal and realistically could be implemented in a day. Regardless, the underlying `'ToroidalDoublyLinkedList'` class functions perfectly and has several test functions to demonstrate this. The same goes for the `'CircularLinkedList'` class.

- **Effect:** Sudoku is an example of the exact cover problem, as such it is NP-complete and while solutions to the problem are trivial to check, they are near impossible to generate via brute-force alone. This is largely a result of exploding/recursive combinatorics. NP-complete problems such as these are best suited by algorithms specifically suited for the problem. This algorithm is an attempt at that. Unfortunately, it doesn't work, I was a bit too ambitious. This algorithm capitalizes on the fact that the incidence matrix for the cover problem implementation of sudoku is sparse and pattern based to remove many options at once.

Reflection

In this project, we embarked on a journey to develop a Sudoku solver and gain deeper insights into the problem-solving algorithms. Our primary objective was to implement a stack-based solver for Sudoku using a depth-first search algorithm. Throughout this endeavor, we encountered several significant findings and made notable progress.

The impact of the number of initial constraints on Sudoku generation and solving times became evident as we delved into the intricacies of the puzzle. Generating Sudoku boards with a higher quantity of initial filled cells was a time-consuming task due to the challenge of avoiding conflicts. As the constraints increased, generating valid boards required frequent retries.

Conversely, the trend reversed when it came to solving Sudoku boards. With a higher number of initial constraints, the solving time decreased significantly. However, it was intriguing to observe that as the number of initial constraints decreased, the time taken to find a solution could increase, especially when the constraints were too low.

Our exploration further uncovered that the nature of constraints on the Sudoku board was a crucial factor in determining the solving time. A minimal number of filled cells was necessary to ensure a unique solution, but if the constraints limited the puzzle too much, the solving process became faster. This dichotomy demonstrates the delicate balance between puzzle complexity and solution efficiency.

The extensions to our project included a graphical user interface (GUI) to make the simulation more interactive and user-friendly. The GUI introduced optional settings, such as controlling the sleep time between updates and offering a step-by-step solution, enhancing the user experience and promoting experimentation.

Additionally, I tried to implement an alternative algorithm, inspired by Donald Knuth's "Dancing Links." While this work-in-progress implementation has potential, I encountered technical challenges related to null pointers and optimization.

In conclusion, the Sudoku solver project has been an interesting journey into the world of problem-solving, and algorithmic efficiency. As I continue to refine my solver and explore alternative algorithms I hope to make a functioning implementation soon.

References/Acknowledgements

I worked on this project individually; all code is my own. Most JavaDoc is handwritten, some is trimmed down versions of migrated Javadoc (ie in implementing LinkedList I copied in the declarations and Javadoc of things I needed to make/implement and then wrote up the code myself before trimming down and rewriting JavaDoc.) Also, in the linked list implementation I looked at Javas own implementation to make sure I implemented some methods I might need as I wanted to make my code able to function on either implementation. Specifically, I originally wrote it for Javas's implementation and coded in my implementation afterwards. However, all code is mine and written by me at the end of the day. My SudokuAlt class is inspired by my understanding of topology in data science, and by the pseudo-code for algorithm X in Donald Knuth's paper on algorithm x accessed from arxiv.