1. Assignment

Task: Study the existing methods for solving a selected problem. Choose one of them or propose your own, implement it, and verify it through experiments. Describe the process and results in a report.

Problem and Method Selection: You may either use the article processed in Assignment 1 or select any problem described in Chapters 13–18 of book [1], with the exception of subsections 13.3, 13.6, 13.10, 14.1 to 14.5, 15.3, 15.4. From Chapter 18, topics related to text and strings are not allowed, but set-related topics are permitted. These restrictions are intended to guide you towards topics not covered in the PRO or PT courses.

1.1 Introduction: Rules

Classic Sudoku has a 9x9 grid with at least 17 prefilled cells. The goal of the game is to fill all empty cells so that the entire grid satisfies three rules:

1. No repeating numbers in any column,
2. No repeating numbers in any row,
3. No repeating numbers in any 3x3 block.

There are many Sudoku variants, such as multisudoku, 12x12, 16x16, etc., but they all follow these three rules without exception.

1.2 Quadrants

We will focus on Sudoku close to the classic version, without additional rules. The block is always a square with sides 3x3 or 4x4, and the number of blocks corresponds to the square of the block side. Puzzles not matching this criteria are excluded from this research.

2. Problem Nature

Sudoku puzzles can be conditionally divided into two categories: Simple or Complex.

Simple puzzles: Have enough prefilled squares so that there is always at least one cell with a single possible candidate.

Complex puzzles: Require much more analysis, since only a few dozen numbers are initially placed on the board, often resulting in 2–3 candidates per cell. This classification affects algorithm choice, though ideally, we want a reliable universal algorithm that can solve puzzles of any difficulty.

3. Existing Methods

Solving strategies consist of several parts. First, we must decide how to select the next cell to fill.

1. Selecting the first, last, or a random empty cell. All are equivalent in the sense that there is no evidence one heuristic is better than another.
2. Selecting the most constrained cell: For each open cell (i, j), check how many candidates remain based on the used values in row i, column j, and the sector containing (i, j). Choose the cell with the fewest candidates. This method is significantly better. Often,

there are open cells with only one remaining candidate, making the choice unavoidable. Assigning such values helps reduce options for other cells. Although more time is needed to choose a cell, it avoids unnecessary backtracking in simple puzzles.

If the most constrained cell has two options, the chance of guessing correctly is 1/2 compared to 1/9 in an unrestricted cell. Reducing average choices per cell from 3 to 2 is a huge win. For 20 cells, 2^20 = 1,048,576 vs. 3^20 > 3 billion operations.

Candidate Value Strategies:

- **Local count**: Backtracking works correctly if the candidate generation routine allows all values (1–9) not already in the row, column, or block.
- **Lookahead**: Sometimes, a partially filled grid leads to a cell with no valid candidates under local rules. It is better to detect this early (lookahead) and backtrack immediately to avoid wasted effort.

Early pruning using lookahead is essential for hard puzzles. Even simple pruning strategies can reduce time from "never" to "instant".

Smart cell selection effectively reduces branching in the solution tree. Each fixed cell imposes constraints that reduce options for future cells.

4. Chosen Solution

The selected strategy:

1. Find the empty cell with the fewest candidates.
2. If a value is valid, assign it and proceed recursively.
3. If recursion returns true, the puzzle is solved.
4. If not, backtrack and try another value.
5. If no value works, backtrack one level further.

Included implementations:

- A simple naive algorithm unable to solve puzzles with fewer than 27 filled cells.
- A custom implementation.
- An algorithm from book [1].

5. Experiments and Results

Puzzle Type	Method 1	Method 2	Method 3
4x4 (125 filled)	-	182.01 ms	41.64 ms
4x4 (126 filled)	-	113.65 ms	33.25 ms
4x4 (127 filled)	-	17.25 ms	11.03 ms
4x4 (128 filled)	-	18.44 ms	11.00 ms
4x4 (129 filled)	0.14 ms	14.46 ms	0.58 ms
3x3 (17 filled)	-	0.98 ms	0.77 ms
3x3 (27 or more filled)	~0.01 ms	< 0.1 ms	< 0.1 ms

Result Explanation:

- 4x4 means a 16x16 puzzle. The number in parentheses is the number of prefilled cells.
- Dashes ("-") indicate failure or exceeded time.

Reducing just one prefilled cell in a 16x16 puzzle dramatically increases solving time, as discussed in [1].

Other key observations:

1. The idea "fewer prefilled cells = easier" is misleading. An empty grid allows multiple solutions.
2. Performance depends more on the **placement** of filled cells than on their **quantity**. Cells that eliminate many candidates are more informative.

In classic 9x9 Sudoku, the minimum solvable configuration has 17 filled cells. All methods except the first showed reliable results across configurations, solving them in under a second. Each method was tested at least 5 times; the table shows average execution time.

5.1 Evaluation of Results

The first method showed no strong results. It was fast but only worked on easy puzzles (≥27 filled). It failed when no cell had a unique candidate.

The custom algorithm performed better and handled harder puzzles, though solving time increased rapidly for 16x16 puzzles with fewer clues.

The third method was clearly superior: faster and reliable across all difficulties.

All programs were written in Java.

Test Environment:

- CPU: Intel(R) Core(TM) i5-9300H @ 2.40 GHz
- RAM: 16.0 GB

6. Conclusion

Numerous tests were conducted to determine which algorithm was fastest and in which scenarios. The simple algorithm was quickest for easy puzzles but failed on complex ones. The custom method showed consistent performance but struggled with large sparse puzzles. The third algorithm combined reliability and speed, outperforming the others in most cases.

7. References

[1] Crook, J. F. "A Pencil-and-Paper Algorithm for Solving Sudoku Puzzles." Notices of the AMS, 2009. Available at: https://www.ams.org/notices/200904/tx090400460p.pdf

[2] Skiena, Steven S. *The Algorithm Design Manual*. 2nd ed., Springer, 2008. Available at: http://cad6.csie.fju.edu.tw/ecp100/Books/Springer.The%20Algorithm%20Design%20Manual.pdf