Sudoku - clicker

Andrii Kozlov

ČVUT–FIT

kozloan3@fit.cvut.cz

25.04.2025

1 Introduction

The goal of this project was to implement a local search-based solution for solving Sudoku using the **hill climbing algorithm**. Unlike systematic methods like backtracking or constraint propagation, local search explores the solution space through small mutations and improves candidates iteratively based on a scoring function.

This project explores multiple mutation strategies, compares their efficiency on various puzzle configurations, and evaluates performance on both standard (3x3 boxes) and extended (4x4 boxes) Sudoku boards.

2 Method of Solution

- **Board representation**

The Sudoku board is represented as a 2D NumPy array with an accompanying boolean matrix indicating fixed cells

- **Initialization**

The algorithm starts by randomly filling non-fixed positions in each row while maintaining uniqueness within that row.

- **Hill Climbing Algorithm**

For each restart:

1. Randomly initialize a complete Sudoku board.
2. Evaluate the board using a **mistake count**: duplicates in rows, columns, and boxes.
3. Apply local mutations (based on a chosen strategy).
4. Accept a new board if it has a lower score.
5. Restart the process after stagnation or local minima.

- **Evaluation Function**

The fitness function counts the number of conflicts in:

1. Each row
2. Each column
3. Each sub-box

- **Mutation Strategies**

Three local mutation strategies were tested:

- **swap_cells_in_row:** Swaps two non-fixed cells in a random row.
- **swap_cells_in_box:** Swaps two non-fixed cells in a random box.
- **change_value_in_cell:** Randomly changes a single non-fixed cell's value.

3 Implementation

Programming Language & Tools

- Python 3.10
- NumPy

Each strategy was run with:

- Maximum 1000 iterations
- Up to 200 restarts
- Strategy comparison based on final score, iterations, and success rate

Files and Modules

- app.py: Entry point of the application.
- board.py: Defines SudokuBoard class and board manipulation.
- algorithms.py: Contains mutation strategies.
- hill_climbing.py: Hill climbing logic and experimentation.
- evaluation.py: Score function used for optimization.
- sudoku: Contains txt files of give puzzles.

4 Results

On *9×9* boards with around *10* missing values, all three mutation strategies performed well, consistently finding solutions quickly. As the number of missing cells increased to *15 or 20*, **swap_cells_in_row** and **change_value_in_cell** remained effective, while **swap_cells_in_box** often failed to make meaningful progress and got stuck in local minima.

At *30* missing values, solving became significantly harder. **swap_cells_in_box** almost never succeeded, and even the better strategies required many restarts, sometimes up to *200* to find a solution.

On *16×16* boards, similar trends emerged. With *10–15* missing values, **swap_cells_in_row** and **change_value_in_cell** usually solved the puzzles, while **swap_cells_in_box** consistently failed. At *30* missing values, only **swap_cells_in_row** showed consistent success, with **change_value_in_cell** occasionally finding a solution.

5 Conclusion

The implemented hill climbing solver successfully solves most Sudoku puzzles with a low-to-moderate number of missing values, especially on standard *3x3* boards. Mutation strategy plays a crucial role in optimization effectiveness.

The best-performing strategy was **swap_cells_in_row**, followed by **change_value_in_cell**. **swap_cells_in_box** was often ineffective, likely due to limited scope of change and less impact on overall board fitness.

Larger boards (*4x4*) increase complexity, especially with higher number of missing values.

6 References

[1] Hill Climbing Algorithm In Artificial Intelligence | Artificial Intelligence Tutorial | Simplilearn

[2] How to override the copy/deepcopy operations for a Python object?

[3] NumPy Documentation