Universiteit Utrecht

Sharing science, *shaping tomorrow*

Evaluating Our Iterated Local Search Implementation for Sudoku

Course Code: INFOB2CI

Q.M.C.G. Reef,

M.A. Chappin &

R. Schenkels

Prepared for:
Professor Dirk Thierens

December 15, 2023

Contents

1 Introduction

This paper presents an assessment of a Sudoku Solver implemented in C#
utilising a hill-climbing Iterated Local Search (ILS) algorithm.
The implementation is a key component of the Computational Intelligence course led by Professor D. Thierens. Our focus lies in providing an elaboration of our
implementation details, delving into the nuances of the hill-climbing ILS algorithm and its heuristic function. Additionally, we conduct experiments involving the manipulation of various parameters to ascertain a well-balanced configuration for the Sudoku solver.

1.1 Running the program

Running our program is trivial when utilising the initially supplied Sudoku

2 Our Implementation

Our implementation adopts an Object-Oriented Programming (OOP) paradigm, primarily utilizing four classes: '*SudokuCluster*', '*Sudoku*', '*SudokuSolver*' and '*Experiment*'. The '*Sudoku*' and '*SudokuCluster*' classes encapsulate the Sudoku board structure and its functions, while '*SudokuSolver*' hosts heuristic evaluation functions and implements the hill-climbing algorithm. '*Experiment*' contains all the function to generate the data for the reproducible results used for evaluation and experiments in this paper.

Efficiency improvements are achieved by efficiently using data types and data structures throughout the program; e.g., utilising the smaller 'ushort' data type instead of 'int', and hash sets instead of regular list structures. While loading the Sudoku from the input file, pre-processing occurs to reduce the processing of the hill-climbing algorithm, which involves tracking free variables and maintaining a list of to-be-swapped coordinates.

The hill-climbing algorithm iterates until a solution is found or the iteration limit is reached. The successors of a particular state are generated by selecting one of the nine Sudoku grids randomly and exploring all possible free variable swap permutations within this grid. When a board with a better fitness is found, the best board variable is replaced. The resulting best found board within this random search space, along with its fitness, is returned to the hill-climbing function to analyse.

Following successor generation, the algorithm compares the current active puzzle with the fitness of the best generated successor's fitness. If an improvement is detected, the loop continues with active Sudoku set to the best successor that contains the swapped values; otherwise, a consecutive attempt counter is incremented. When, during the generation of the next successors, a better option is found within the bounds of a predefined threshold, this counter resets. The predefined threshold is set to such a value that we almost certainly know that all clusters have been randomly selected upon reaching the threshold that we set to 30, to have a near 100% chance to select each cluster at least once, per: $E(n) = n \times \left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}\right)$ also known as the coupon collector problem. This allows us to assume that there are no better states deriving from the current Sudoku puzzle, signifying a local maximum or a plateau.

When this is the case, a random walk of variable size occurs within random grids. This random walk entails the swapping of two Sudoku elements without evaluating outcomes for a defined number of iterations. The resulting fitness of the board post-walk sets the new local best score in the hill-climbing algorithm. Subsequently, another hill-climbing operation commences from this point, seeking a better global maximum. If found, the global maximum is updated; otherwise, the algorithm reverts to the previous best score and initiates a random walk from that position. This iterative process continues until a satisfactory global maximum is attained.

The heuristic values are calculated using two different functions depending on the situation. The '*InitHeuristics*' function iterates over each column and row to assess the entire Sudoku, and is only used once for initializing the heuristic values of a randomly filled Sudoku. When updating the heuristic values with the '*UpdateHeuristics*' after two cells are swapped in the '*DetermineBestSuccessor*' function, at most two rows and two columns are evaluated reducing the cost of resources for the heuristic function.

3 Efficiency

3.1 Space Complexity

The space complexity of our Hill Climbing algorithm is generally considered to be relatively low. In our implementation, the algorithm maintains a two states, being current state and last local optimum. It explores its neighboring states. The amount of memory required is proportional to the depth of the search space, specifically the number of states stored in memory at any given time which in our case comes at around 2-3 (Current Sudoku, Best successor & Previous Local Optimum). Compared to more memory intensive search algorithms such as CSP the space complexity of the Hill Climbing algorithm is modest.

- Using *unsigned short* data types, we achieve greater space efficiency than using full *integers* to represent most of our program, as most values will never be negative or exceed 65,535. This is especially beneficial when working with large arrays, as is the case with Sudoku grids. Another, possibly more space efficient option would have been the *char* representation; however, this solution leaves little overhead space and brings another problem with them, namely, if we have to make a calculation with a *ushort* and a *char*, we will need to explicitly cast one of the values; this adds unnecessary code complexity for the small space complexity improvement.
- The use of *HashSet* for storing and accessing unique elements (e.g. checking the numbers in a row or column) is also a space-efficient choice. *HashSet*s prevent duplication and thus prevent the storing values we will not need, instead counting the fails of the *Add* function.
- By choosing an Object Oriented approach, we unfortunately need to keep a lot of deep copies of Sudoku boards around, resulting in worse overall space complexity, but aiding in productivity and splitting problems up to smaller bite-sized problems to solve.

3.2 Time Complexity

Because Hill Climbing algorithms are very dependent on the landscape of the solution space, determining the time complexity can be divided into three cases, namely, the best, worst and average case. We will compare these complexities with brute forcing a Sudoku, which will explore the entire search space and thus have an exponential complexity.

- **Best Case:** If the algorithm finds the right path quickly, and without many obstacles (like local minima or plateaus) the time complexity will be closer to polynomial. In theory, linear is possible if the hill climbing algorithm finds the right path right away, and runs into no local minima on the way, this however, is not feasible and thus will not be taken into consideration for the complexity analysis.
- **Worst Case:** In complex puzzles, where there is ample opportunity to run into local optima or plateaus, the algorithm may frequently get stuck or will need to perform many random walks, this can lead to exponential time complexity, similar to brute force.
- **Average Case:** Commonly, the time complexity will lean towards polynomial when solving a Sudoku with a hill climbing algorithm, this can, however, vary as the complexity of the Sudoku increases or decreases.

In conclusion, our implementation of the *Evaluate , HillClimbing, RandomWalk and GetSuccessorsOrderedByScore* always looks for the optimal path to take, but takes into consideration that we may get stuck on local minima or plateaus with the *consecutiveIterationsWithoutImprovement* variable. On average, our algorithm boasts exponential time complexity, as seen in Figure 5.3.

4 Solution Completeness

The completeness of a Sudoku solver algorithm is for effectively solving a given problem. In the context of our Sudoku solver algorithm, the Hill-Climbing approach doesn't guarantee finding a solution in all instances. Standard Hill-Climbing algorithms, even when augmented with random walks, may struggle to escape certain local optima. These optima, as well as plateaus, pose challenges, persisting even after attempts to diverge with random walks, potentially leading to sub-optimal solutions that approach a global maximum, but can never be escaped.

The algorithm's greedy exploration of neighboring solutions may result in overlooking more promising areas in the solution space. To mitigate this, introducing biased successor selection could enhance the algorithm's ability to discover globally optimal solutions. Additionally, the algorithm's sensitivity to the random initial state is notable; if the starting state is far from the globally optimal solution, the random walk size may not be sufficient to navigate to more favorable regions, thus, again, getting stuck in local optima.

4.1 Edge cases

Edge cases can pose challenges to the completeness and effectiveness of the solver. In some scenarios the algorithm struggles to converge to a solution that ultimately approaches the global maximum as it has not sufficiently traversed the search space, potentially missing more optimal solutions that exist in unexplored regions. Another scenario is that the algorithm converges to 1D local optima. In our implementation, it is impossible to gather if either the row values or mainly the column values had contributed to a lower overall fitness. This may cause incorrect convergence of the search area.

5 Experiments

We employed various methodologies to investigate our performance over various problem states. Our focus is on understanding the dynamics of sudoku puzzle solving and the implications of varying various variables.

5.1 Sample Size

Every Sudoku was solved a variable amount of times with 22 different random step sizes. To determine the effect of the sample size on our experiments, resulting in the following graph. The best fitting and least resource impacting sample size was chosen to perform the remaining experiments efficiently.

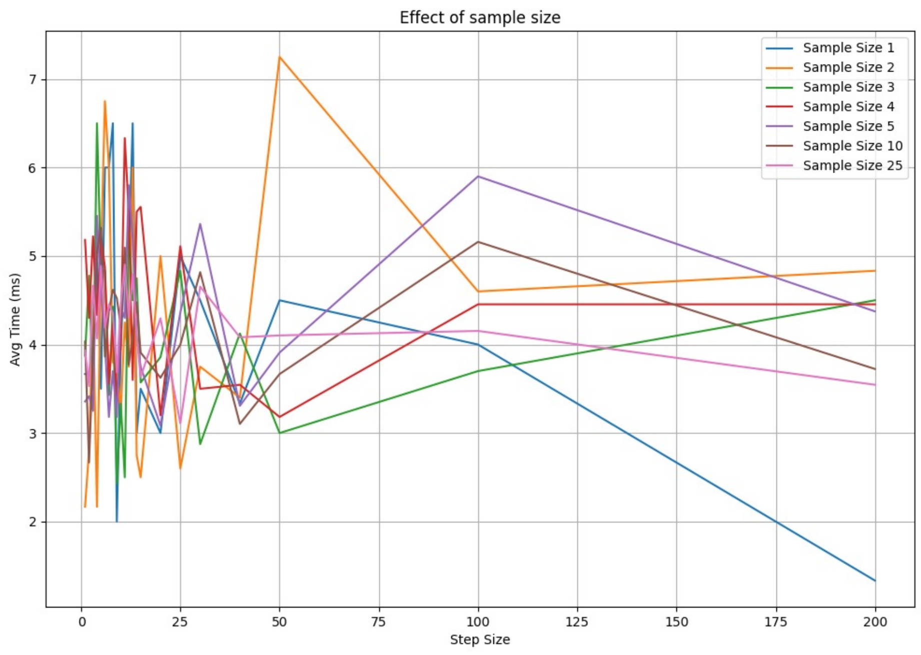

Figure 1: Illustrates optimal sample size for further experiments, indicating that using a sample size of 5 returns relative most reliable results.

5.2 Random Walk Size Variations

In order to determine a fitting parameter for our random walk step size, we conducted an experiment in which we tried to solve all Sudoku's with all permutations of the variable walk step size. Afterwards we combined these results of the different Sudoku's to use it in a single metric. This metric is the total median processing time along with it's interquartile range (IQR) and the suspected outliers.

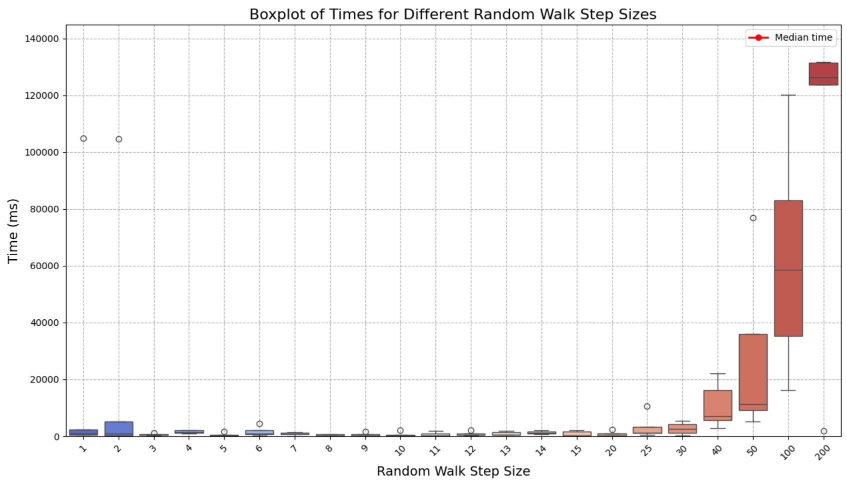

Figure 2: Illustrates median processing time in ms for all five Sudokus for all the different random walk variation, indicating that starting from a random walk step size of 40 and upwards the algorithm no longer efficiently solves the Sudoku's.

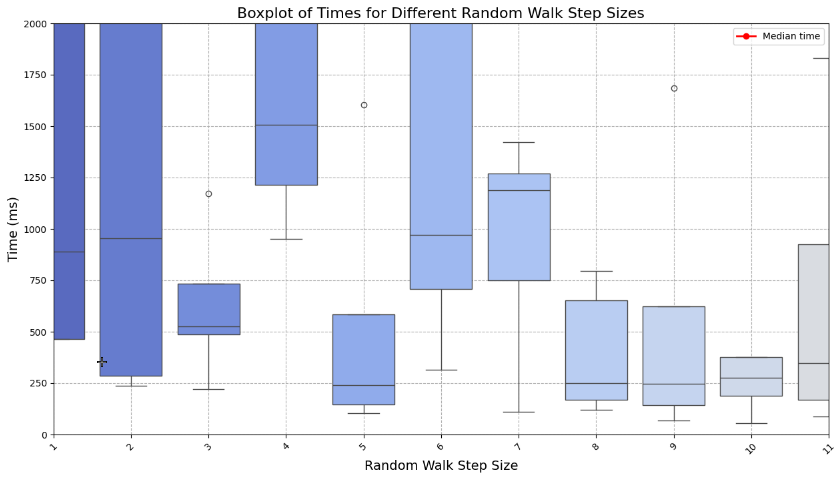

Figure 3: Illustrates median processing time in ms for all five Sudokus for the different random walk variation ranging from 0 11, indicating that a random walk of 5 for our algorithm appears to be the most efficient.

5.3 Easy vs. Hard Sudoku's

Using the previous found optimal random walk step size we solved the individual Sudoku's 22 times, as by enlarging the sample sizes we could make an accurate median prediction of the solving times. We used a logarithmic scale as the processing time results for some Sudoku's varied several orders in magnitude.

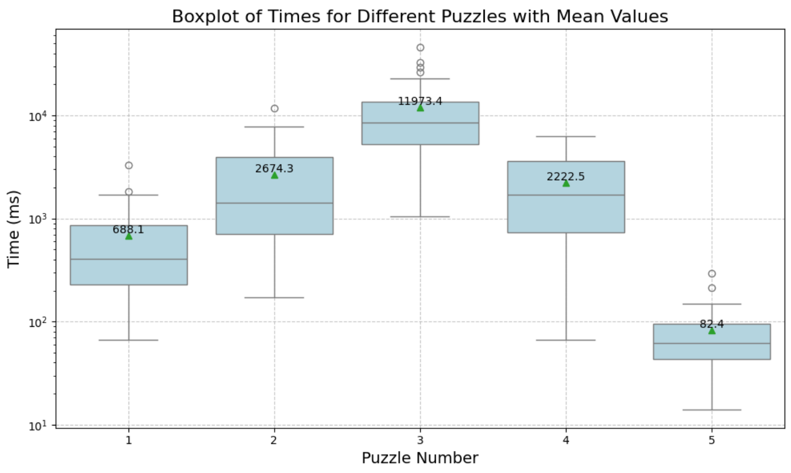

Figure 4: Illustrates median processing time in ms for each puzzle on log scale, indicating that Sudoku 3 is the hardest (med. ms ∼12000) and Sudoku 5 is the simplest (med. ms ∼ 82) to solve.

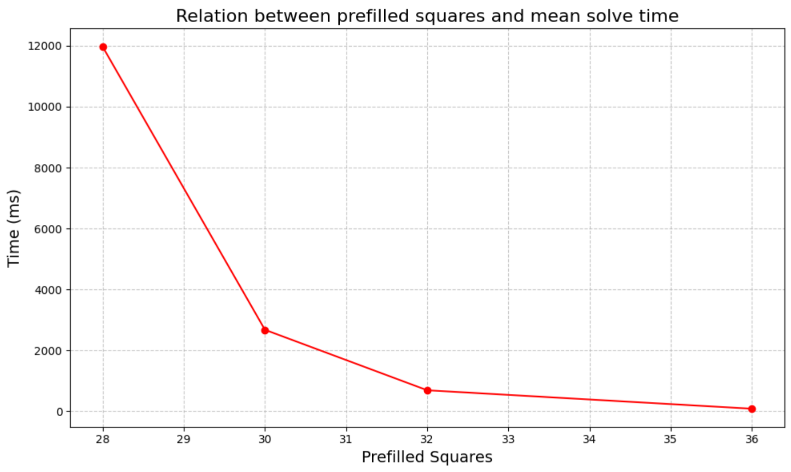

Figure 5: Illustrates the fixed tiles vs the median processing time in ms for each puzzle, indicating that more free variables result in a larger search area and thus a higher processing time.

6 Discussion

Our research into the performance of the Sudoku solver involved systematic experiments, shedding light on the algorithm's behaviour under several conditions. In this section we discuss the key aspects of our methodologies and our findings derived from the conducted experiments.

6.1 Sample Size Optimization

The first experiment focused on determining an optimal sample size for conducting the future Sudoku puzzle experiments. Through 22 iterations with different random step sizes, we identified that by maintaining a sample size of 5 yielded reliable results while. When compared to for example a sample size of 10, it produced similar data without to much variation (As can be seen by the spikes in Figure 1). Using 5 instead of 10 as a sample size reduced the processing time of each experiment in half, which in our case would be approximately 30 minutes instead of 1 hour (for experiment 2).

6.2 Random Walk size variation

The variation in random walk step sizes was explored to pinpoint a global optimal parameter for the algorithm. The results indicated that a random walk step size of 5 appears to be the most efficient, as illustrated by the processing time box plots. Starting from a step size of 40 and upwards, the algorithm's efficiency notably declines, highlighting the importance of selecting an appropriate step size for effective Sudoku solving.

6.3 Easy vs. Hard Sudoku Puzzles

Employing the optimal random walk step size, we delved into solving individual Sudoku puzzles 22 times each, allowing for a robust analysis of solving times. The logarithmic scale effectively shows the vast differences in processing times across puzzles. Notably as seen in Figure 4, Sudoku 3 emerged as the most challenging, with a median processing time of approximately 12,000 milliseconds, while Sudoku 5 proved to be the simplest with a median processing time of around 82 milliseconds.
Sudoku 3, with fewer fixed tiles (28 out of 81), offers a more extensive problem space for the algorithm to explore, leading to an extensively larger solving time compared to Sudoku 5 (with 36 fixed tiles out of 81). Figure 5 visually underscores the exponential correlation between the number of fixed tiles and the median solving times.

7 Reflection

7.1 Experiments

While our experiments provided valuable insights, we recognize the possible need for an investigation into Sudoku puzzle-specific parameter optimization. Although constrained by time in this phase, we wanted to delve deeper into refining these parameters to enhance the algorithm's Sudoku's specific performance. Also the implementation of random restarts or biased successor selection has been left for a future project due to a time lack of time.

7.2 Programming

Throughout the development of our algorithm for solving Sudoku puzzles, we encountered several challenges that significantly impeded our progress. Persistent underlying pointer issues due to a late discovered critical error in the clone function of our Sudoku class and its sub-classes were amongst the most impeding hurdles. Because of our OOP approach we extended our classes with an instance of IClonable. Instead of a deep-copy we made a shallow copy, resulting in inexplicable side effects during the evaluation and assignment to the '*_activeSudoku*'. It took a lot of time to figure out that this issues existed in the first place. We rectified this issue by a collective comprehensive review of our class implementation, with a focus on ensuring the proper replication of puzzle states.

Furthermore, we encountered some issues with getting stuck in local optima, which we resolved by altering the random step size. There also persisted an issue were the state extremely diverged from the solution, to resolve this we kept track of the previous best local optimum. After a completing a random walk we check if the random walk had found a resulted in a worse fitness and reverted the current active board to the local optimum we found before commencing the hill climb after the random walk. This way we hope to keep the current state near the global optimum. Overall implementing the actual algorithm without the reference issues took us about 6 hours each. This does not include the time determining the program structure and resolving issues and refactoring the code base.

Despite these challenges, the programming experience provided valuable lessons for us the keep us alert in future projects.

8 Conclusion

Utilizing C# to implement a Sudoku Solver with a hill-climbing ILS algorithm, showed effective puzzle-solving capabilities. Leveraging Object-Oriented Programming and efficient data structures, we optimized parameters like sample size and random walk step size by conducting several experiments. The resulting refined algorithm performs reasonably well, though it still closely resembles an exponential curve, as it is a probabilistic algorithm. It remains sensitive to initial states and potential local optima. Despite encountering programming hurdles, the project enlightened our understanding of algorithmic design.