

A Study Of Sudoku Solving Algorithms: Backtracking and Heuristic

Apekshya Bhattacharai¹, Dinisha Uprety¹, Pooja Pathak¹, Safal Narshing Shrestha¹, Salina Narkarmi¹, Sanjog Sigdel^{1*}

¹Department of Computer Science, Kathmandu University, Dhulikhel, Nepal

*Corresponding author: sanjog.sigdel@ku.edu.np

Abstract

This paper presents a comparative analysis of Sudoku-solving strategies, focusing on recursive backtracking and a heuristic-based constraint propagation method. Using a dataset of 500 puzzles across five difficulty levels (Beginner to Expert), we evaluated performance based on average solving time. The heuristic approach consistently outperformed backtracking, achieving speedup ratios ranging from 1.27× in Beginner puzzles to 2.91× in Expert puzzles. These findings underscore the effectiveness of heuristic strategies, particularly in tackling complex puzzles across varying difficulty levels.

Keywords: Sudoku, backtracking, heuristic, constraint propagation

1 Introduction

Sudoku is a puzzle played on a partially filled 9x9 grid. The task is to complete the assignment using numbers from 1 to 9 such that the entries in each row, each column, and each major 3x3 block are pairwise different. Like for many logical puzzles, the challenge in Sudoku does not just lie in finding a solution [1]. Instead, it involves understanding the underlying strategies and techniques necessary for solving the puzzle efficiently. Players must navigate various constraints and make decisions based on limited information, often requiring critical thinking and pattern recognition. Moreover, the difficulty of Sudoku puzzles can vary significantly, demanding different levels of skill and problem-solving approaches. This multifaceted nature makes Sudoku not only an engaging activity but also a rich subject for algorithmic analysis and study. In the past few decades, distinct algorithms have been created for solving Sudoku puzzles. In this paper, we use Backtracking[2] and Heuristic[3] algorithms and compare their time to solve Sudoku puzzles.

We have evaluated our methods on different sets of difficulties of the puzzles to see how efficient both algorithms were in comparison to each other and themselves, varying the difficulty. This evaluation gives a fair comparison of the difficulty of different puzzle sources.

Backtracking

This is the easiest approach as it avoids computational complexities. This algorithm visits all the empty cells in a specific order and fills it with a digit from 1-9, and checks its validity. If no choice is available for a cell then it backtracks and changes the digit of the previous cell. Thus the algorithm continues until all the cells are filled with appropriate digits. [2][4]

Heuristic

This approach is an algorithm that combines human intuition and optimization to solve Sudoku puzzles [3]. It involves constraint propagation, where the algorithm eliminates impossible values for each cell based on existing constraints. It keeps track of possible candidates for each empty cell and uses techniques like naked singles or hidden pairs to progressively fill the grid. If no certain move is available, it may use backtracking with educated guesses, trying values and reverting if conflicts arise.

2 Related Works

The problem of solving Sudoku puzzles has been widely explored within the domain of Constraint Satisfaction Problems (CSPs) [5], particularly using backtracking combined with heuristics. These approaches aim to reduce the solution search space and improve the efficiency of solving time.

Simonis [1] proposed modeling Sudoku as a CSP using the all-different constraint, which ensures that each digit appears only once in every row, column, and 3×3 box. This approach laid the groundwork for constraint propagation techniques such as Arc Consistency (AC-3) and Forward Checking, which are widely used to prune inconsistent values early during the search.

Norvig [6] introduced a well-known Python-based Sudoku solver that integrates backtracking with constraint propagation. This method uses the Minimum Remaining Value (MRV) heuristic to prioritize variables with the fewest legal values, significantly improving search efficiency.

Rodrigues et al. improved heuristic performance by integrating both the Minimum Remaining Value (MRV) and Least Constraining Value (LCV) strategies [7]. Our solver adopts a similar strategy by implementing MRV, LCV, and other heuristics to optimize decision-making during solving.

Studies have experimented with hybrid techniques combining backtracking, constraint propagation, and domain-specific heuristics for enhanced performance [8]. Their work demonstrated the effectiveness of these methods in solving complex problems like Sudoku. Barker et al. [9] developed a method combining local search and constraint satisfaction, using backtracking along with a greedy approach to select promising variables, showing significant improvements for solving more challenging Sudoku puzzles.

In addition to CSP-based strategies, Sudoku solving has also been explored through the lens of combinatorial optimization. Sudoku is a special case of the Exact Cover problem, which is NP-Complete, as proven by the general problem of solving $n \times n$ Sudoku grids [10]. Knuth's Dancing Links (DLX) algorithm [11], a technique for solving Exact Cover problems efficiently using sparse matrix manipulation, has been successfully applied to Sudoku solving. DLX performs particularly well due to its ability to efficiently undo and redo decisions during the backtracking process.

3 Methodology

We developed a web-based Sudoku application comprising two main sections: **Play** and **Solver**. The Play section features a custom puzzle generator that creates playable grids of varying difficulty. The Solver section allows users to input puzzles and compare two solving methods—recursive backtracking and heuristic-based constraint propagation—displaying their solving times side by side. This setup enabled real-time evaluation of algorithmic performance across multiple difficulty levels.

3.1 Application Framework

The architecture of our application adopts a modular design, with a clear separation between the frontend and backend components:

- **Frontend:** Built using HTML, CSS, and JavaScript, offering a responsive and user-friendly interface.
- **Backend:** Developed using Django (Python), with Postgres as the database. Core logic and algorithm execution are handled server-side via standard Django APIs.

3.2 Experimental Setup

To evaluate the performance of both the recursive backtracking and heuristic-based Sudoku solving algorithms, we conducted experiments using a dataset of 500 Sudoku puzzles across various difficulty

levels. These puzzles were generated through our own web-based Sudoku platform. The dataset was evenly categorized into five levels — Beginner, Easy, Medium, Hard, and Extreme — with 100 puzzles in each category. The difficulty classification was based on the number of initial clues (pre-filled cells) in each puzzle: **Beginner** puzzles contained 50 clues, **Easy** 40 clues, **Medium** 35 clues, **Hard** 27 clues, and **Extreme** 20 clues. The puzzles were collected using the solver section of our platform, and the solving time for both algorithms was recorded.

This experimental setup enabled a comprehensive analysis of algorithmic performance across varying levels of complexity, highlighting how each method scales with puzzle difficulty.

3.3 Puzzle Generator

To create a sudoku puzzle, we used backtracking method to completely fill an empty 9x9 grid. This process starts from the very first cell and proceeds cell by cell, ensuring that every placement complies with the rules of sudoku. Once a valid and fully populated grid was produced, we then applied the puzzle's difficulty by selectively removing numbers from the grid. The number of clues that remain corresponds to the chosen difficulty level; fewer clues generally mean a more challenging puzzle. By carefully "popping" values from the completed grid, we created a new playable puzzle, inviting solvers to engage with the logical challenge of filling in the missing numbers. The following pseudocode outlines the Sudoku puzzle generator that has been implemented:

Input: Empty 9x9 grid
Output: Sudoku Puzzle based on the chosen difficulty

```

1: function ISVALIDMOVE(grid, row, col, num)
2:   if num is in grid[row] then
3:     return False
4:   end if
5:   if num is in column col of grid then
6:     return False
7:   end if
8:   (start_row, start_col)  $\leftarrow 3 \cdot (\text{row} \div 3), 3 \cdot (\text{col} \div 3)$ 
9:   for i = 0 to 2 do
10:    for j = 0 to 2 do
11:      if grid[start_row + i][start_col + j] = num then
12:        return False
13:      end if
14:    end for
15:   end for
16:   return True
17: end function
```

Fig.1 Pseudocode of ISVALIDMOVE function

```

1: function FINDEMPTY(grid)
2:   for i = 0 to 8 do
3:     for j = 0 to 8 do
4:       if grid[i][j] = 0 then
5:         return (i, j)
6:       end if
7:     end for
8:   end for
9:   return None
10: end function

```

Fig.2 Pseudocode of FINDEMPTY function

```

1: function SOLVESUDOKU(grid)
2:   (row, col)  $\leftarrow$  FINDEMPTY(grid)
3:   if no empty cell found then
4:     return True                                 $\triangleright$  Puzzle is solved
5:   end if
6:   for num in shuffled 1 to 9 do
7:     if ISVALIDMOVE(grid, row, col, num) then
8:       grid[row][col]  $\leftarrow$  num
9:       if SOLVESUDOKU(grid) then
10:        return True
11:      end if
12:      grid[row][col]  $\leftarrow$  0                 $\triangleright$  Backtrack
13:    end if
14:   end for
15:   return False
16: end function

```

Fig.3 Pseudocode of SOLVESUDOKU function

```

1: function GENERATESUDOKU(clues)
2:   Initialize grid as  $9 \times 9$  zeros
3:   for i in  $\{0, 3, 6\}$  do
4:     nums  $\leftarrow$  shuffled list from 1 to 9
5:     for j = 0 to 2 do
6:       for k = 0 to 2 do
7:         grid[i + j][i + k]  $\leftarrow$  POP(nums)
8:       end for
9:     end for
10:   end for
11:   SOLVESUDOKU(grid)
12:   solution  $\leftarrow$  copy of grid
13:   cells  $\leftarrow$  all  $(i, j)$  in grid
14:   Shuffle cells
15:   for first  $81 - \text{clues}$  cells do
16:     (row, col)  $\leftarrow$  cell
17:     grid[row][col]  $\leftarrow$  0
18:   end for
19:   return (grid, solution)
20: end function

```

Fig.4 Pseudocode of GENERATESUDOKU function

3.4 Algorithm Design

This research implemented two algorithms: heuristic and backtracking, to solve the puzzles generated by our generator. Below is an overview of each algorithm:

3.4.1 Recursive Backtracking

The backtracking algorithm systematically explores all possible configurations of the puzzle. It incrementally builds candidates for solutions and abandons a candidate as soon as it is determined that it cannot lead to a valid solution.

Input: 9×9 grid *G* with some pre-filled values and zeros representing empty cells.

Output: Solved grid or "No solution"

```

1: function SOLVE( $G$ )
2:   for  $r \leftarrow 0$  to 8 do
3:     for  $c \leftarrow 0$  to 8 do
4:       if  $G[r][c] = 0$  then
5:         for  $num \leftarrow 1$  to 9 do
6:           if ISVALID( $G, r, c, num$ ) then
7:              $G[r][c] \leftarrow num$ 
8:             if SOLVE( $G$ ) then
9:               return True
10:            end if
11:             $G[r][c] \leftarrow 0$ 
12:          end if
13:        end for
14:        return False
15:      end if
16:    end for
17:  end for
18:  return True
19: end function

```

Fig.5 Pseudocode of SOLVE function

Time Complexity :

The results of calculations of the time complexity of the backtracking algorithm using the big theta notation is(average case)[12]

$$\theta(n^3)$$

3.4.2 Heuristic-Based Solver (MRV + LCV)

The heuristic-based solver prioritizes selecting the variable with the fewest legal values remaining (MRV) and then chooses values that impose the least restriction on other variables (LCV), which significantly speeds up the solving process by reducing the search space and employs backtracking when a chosen value proves incorrect.

Input: 9×9 grid G with some pre-filled values and zeros representing empty cells.

Output: A completely filled valid Sudoku grid if solvable; otherwise, the message “No solution”.

```

1: function SolveSudokuWithHeuristics(board)
2:   results  $\leftarrow \{\text{solved: false, solution: null, error: null}\}$ 
3:   possibilities  $\leftarrow \text{INITIALIZEPOSSIBILITIES}(\textit{board})$ 
4:   HEURISTICS(board, possibilities)
5:   return results
6: end function

```

Fig.6 Pseudocode of SolveSudokuWithHeuristics function

```

1: function HEURISTICS(board, possibilities)
2:   row, col, values  $\leftarrow \text{FINDMOSTCONSTRAINEDCELL}(\textit{possibilities})$ 
3:   if row = -1 and col = -1 then
4:     results.solved  $\leftarrow \text{true}$ 
5:     results.solution  $\leftarrow \text{copy of board}$ 
6:     return true
7:   end if
8:   if useRandomization then
9:     Shuffle values
10:   end if
11:   cellCoord  $\leftarrow (\textit{row}, \textit{col})$ 
12:   for each num in values do
13:     if IsVALIDMOVE(board, row, col, num) then
14:       board[row][col]  $\leftarrow \textit{num}$ 
15:       updatedPossibilities  $\leftarrow \text{deep copy of possibilities}$ 
16:       Remove cellCoord from updatedPossibilities
17:       for each (r, c) in GETAFFECTEDCELLS(row, col) do
18:         if (r, c) exists in updatedPossibilities then
19:           Remove num from updatedPossibilities[(r, c)]
20:         end if
21:       end for
22:       if HEURISTICS(board, updatedPossibilities) then
23:         return true
24:       end if
25:       board[row][col]  $\leftarrow 0$                                  $\triangleright \text{Backtrack}$ 
26:     end if
27:   end for
28:   possibilities[cellCoord]  $\leftarrow \text{set of values}$ 
29:   return false
30: end function

```

Fig.7 Pseudocode of Heuristics function

```

1: function GETAFFECTEDCELLS((row, col))
2:   Initialize empty set affected
3:   for c  $\leftarrow$  0 to 8 do
4:     if c  $\neq$  col then
5:       Add (row, c) to affected
6:     end if
7:   end for
8:   for r  $\leftarrow$  0 to 8 do
9:     if r  $\neq$  row then
10:      Add (r, col) to affected
11:    end if
12:  end for
13:  box_row  $\leftarrow$  3  $\times$   $\lfloor$ row/3 $\rfloor$ 
14:  box_col  $\leftarrow$  3  $\times$   $\lfloor$ col/3 $\rfloor$ 
15:  for r  $\leftarrow$  box_row to box_row + 2 do
16:    for c  $\leftarrow$  box_col to box_col + 2 do
17:      if (r, c)  $\neq$  (row, col) then
18:        Add (r, c) to affected
19:      end if
20:    end for
21:  end for
22:  return affected
23: end function

```

Fig.8 Pseudocode of GETAFFECTEDCELLS function

```

1: function ISVALIDMOVE(board, row, col, num)
2:   for x  $\leftarrow$  0 to 8 do
3:     if board[row][x] = num or board[x][col] = num then
4:       return False
5:     end if
6:   end for
7:   Compute box start indices
8:   for each cell in the 3x3 box do
9:     if cell has value = num then
10:      return False
11:    end if
12:   end for
13:   return True
14: end function

```

Fig.9 Pseudocode of IsValidMove function

```

1: function INITIALIZEPOSSIBILITIES((board))
2:   Initialize empty dictionary possibilities
3:   for each (row, col) in board do
4:     if board[row][col] = 0 then
5:       possibilities[(row, col)]  $\leftarrow \{1, \dots, 9\}$ 
6:     end if
7:   end for
8:   for each (row, col) in board do
9:     if board[row][col]  $\neq 0$  then
10:      value  $\leftarrow$  board[row][col]
11:      for each (r, c) in GetAffectedCells(row, col) do
12:        if (r, c)  $\in$  possibilities then
13:          Remove value from possibilities[(r, c)]
14:        end if
15:      end for
16:    end if
17:  end for
18:  return possibilities
19: end function

```

Fig.10 Pseudocode of INITIALIZEPOSSIBILITIES function

```

1: function FINDMOSTCONSTRAINEDCELL((possibilities))
2:   if possibilities is empty then
3:     return  $(-1, -1, [])$ 
4:   end if
5:   min  $\leftarrow 10, best_cell  $\leftarrow (-1, -1)$ 
6:   for each (row, col), values in possibilities do
7:     if  $|values| < min$  then
8:       min  $\leftarrow |values|$ 
9:       best_cell  $\leftarrow (row, col)$ 
10:      best_values  $\leftarrow values$ 
11:    end if
12:   end for
13:   return  $(row, col, list(best\_values))$ 
14: end function$ 
```

Fig.11 Pseudocode of FINDMOSTCONSTRAINEDCELL function

Time Complexity :

The time complexity of the heuristic algorithm for solving a Sudoku puzzle is almost in polynomial time. [13]

Heuristic Cost Function :

A heuristic cost function is a mathematical function used to estimate the cost or effort required to reach a solution from a given state in a problem space.

$$f(s) = g(s) + h(s)$$

- $f(s)$: Total estimated cost
- $g(s)$ = Cost value of the trajectory performed from the root to the current (s) state;
- $h(s)$ = Heuristic function that estimates the cost of the path from the current state to the destination state. [14]

In our approach, these are defined as:

A heuristic cost function $f(s)$ to evaluate the quality of a partially completed Sudoku state s . This function guides the backtracking algorithm by combining both the progress made and the difficulty of remaining decisions.

- $g(s) = 81 - |\text{possibilities}|$
where $|\text{possibilities}|$ is the number of currently empty cells.
- $h(s) = \sum_{(i,j) \in \text{possibilities}} \frac{1}{|P(i,j)|}$
where $P(i, j)$ is the set of possible values for cell (i, j) .

Therefore, the full heuristic becomes:

$$f(s) = 81 - |\text{possibilities}| + \sum_{(i,j)} \frac{1}{|P(i,j)|}$$

4 Result

To evaluate the effectiveness of the two solving approaches, we ran both algorithms on 500 Sudoku puzzles spanning five difficulty levels: Beginner, Easy, Medium, Hard, and Expert. The table below summarizes the average solving time (in milliseconds) for each method.

Table 1: Average Solving Time for 500 Sudoku Puzzles

Difficulty	Backtracking (ms)	Heuristic (ms)
Beginner	0.93	0.73
Easy	4.56	3.27
Medium	14.78	8.26
Hard	173.45	78.88
Expert	360.72	123.80

4.1 Speedup Analysis

To better understand the relative improvement of the heuristic solver, we computed the speedup ratio[15] as the time taken by the backtracking approach divided by the time taken by the heuristic approach. The heuristic solver consistently outperforms backtracking, with its advantage becoming more significant at higher difficulty levels.

Table 2: Speedup Ratio of Heuristic Solver Over Backtracking

Difficulty	Backtracking (ms)	Heuristic (ms)	Ratio (y/x)	Interpretation
Beginner	0.93	0.73	1.27	~1.27x faster
Easy	4.56	3.27	1.39	~1.39x faster
Medium	14.78	8.26	1.79	~1.79x faster
Hard	173.45	78.88	2.20	~2.20x faster
Expert	360.72	123.80	2.91	~2.91x faster

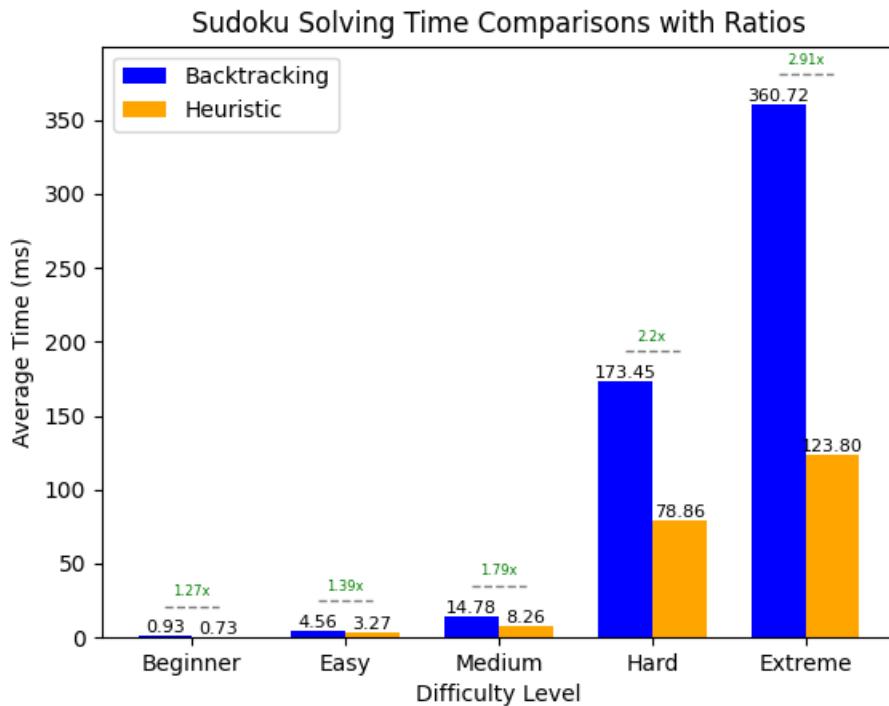


Fig.12 Barchart Solving Time Comparison Across Difficulties

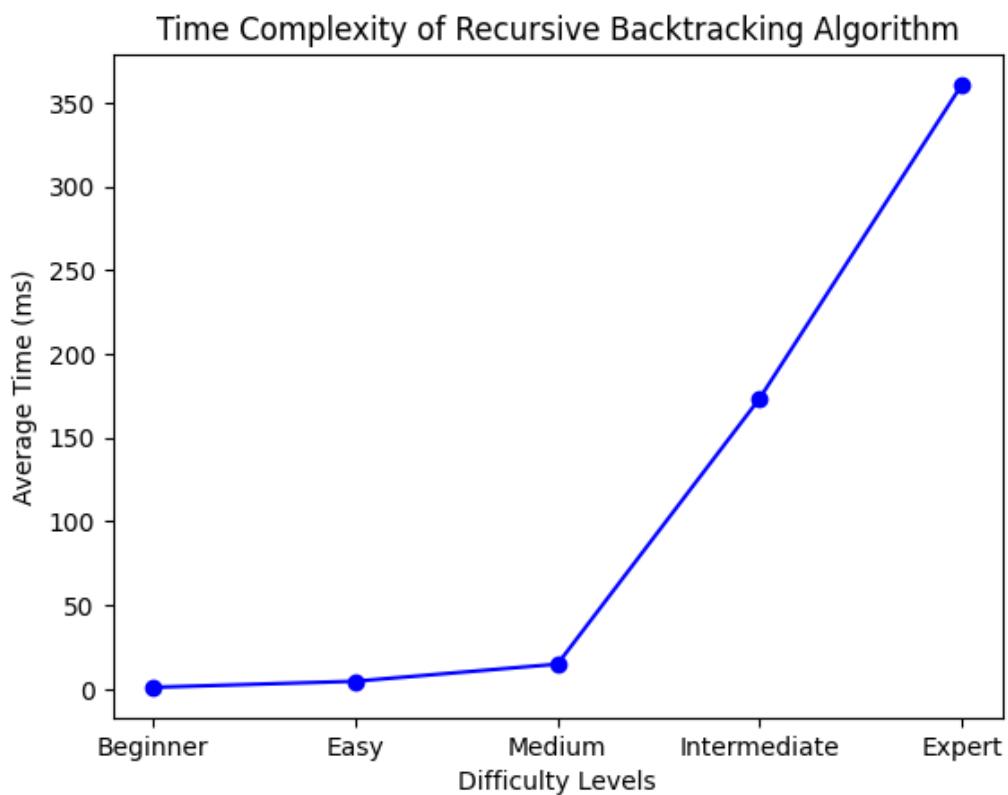


Fig.13 Line Chart Solving Time of Backtracking Across Difficulties

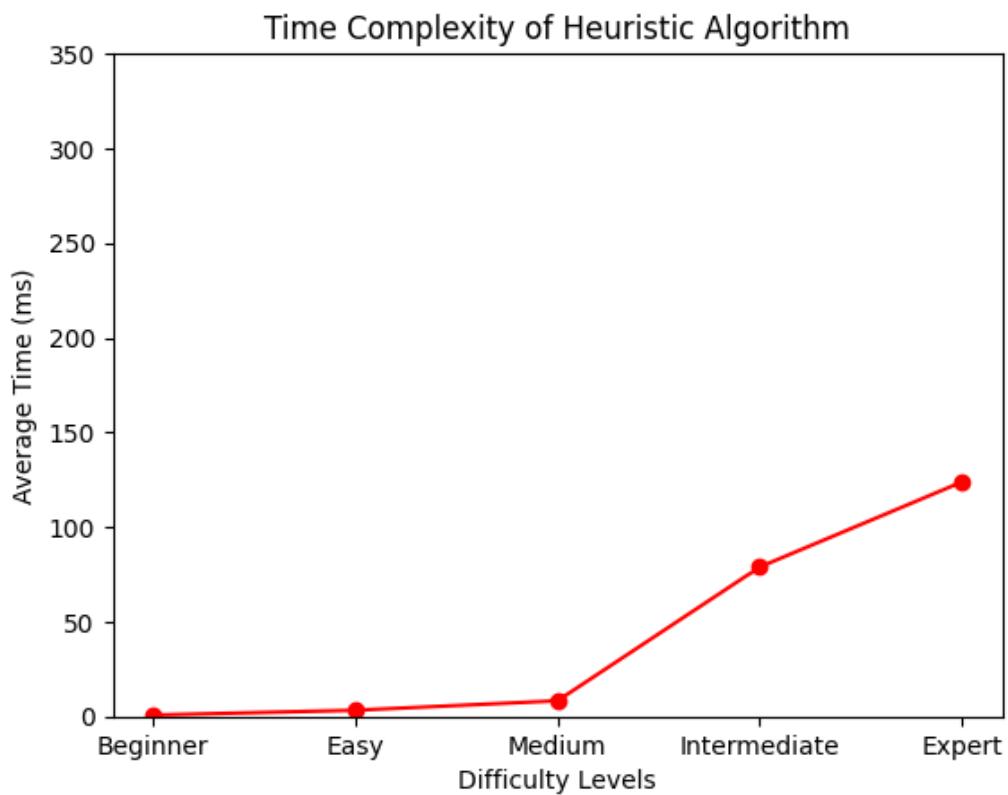


Fig.13 Line Chart Solving Time of Heuristic Across Difficulties

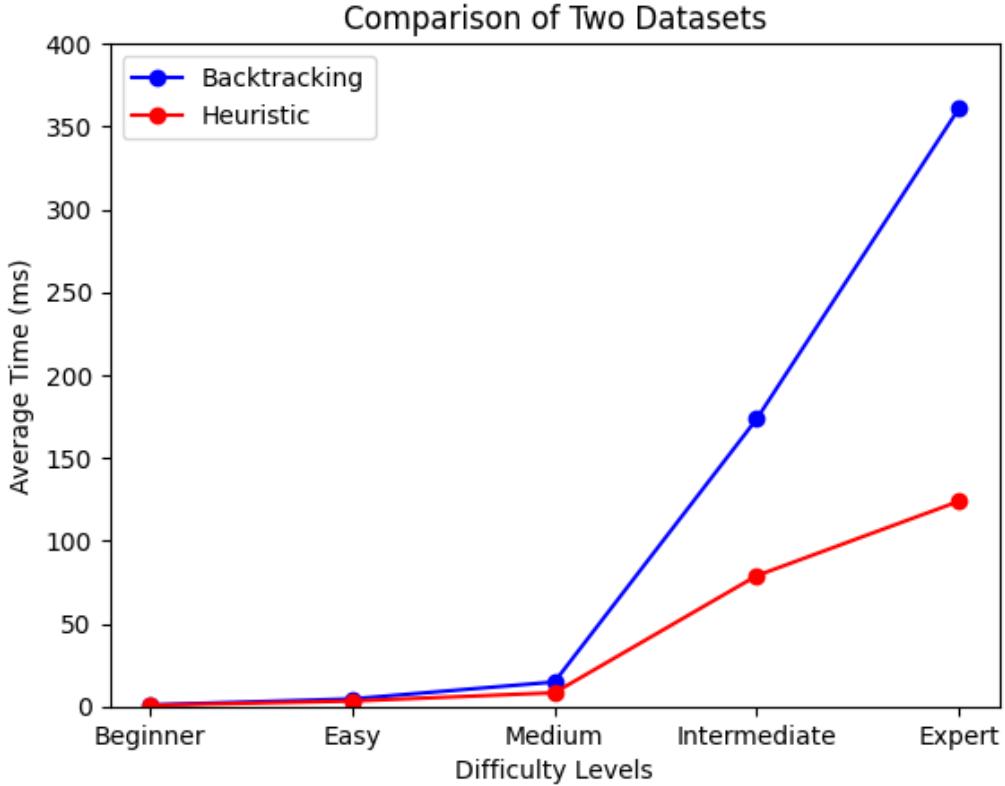


Fig.13 Line Chart Solving Time Comparison Across Difficulties

The analysis shows that as puzzle difficulty increases, the gap in performance between the two methods widens. The **speedup ratios** reveal that the heuristic method becomes more advantageous for harder puzzles. Additionally, the heuristic solver maintains **consistency**, performing efficiently across all difficulty levels. Overall, while backtracking serves as a simple baseline, the heuristic method significantly improves performance, particularly with more complex puzzles.

5 Conclusion

Heuristic-based methods greatly improved the effectiveness of Sudoku solvers, especially when dealing with more challenging puzzles. Although recursive backtracking is a straightforward and efficient technique, the heuristic approach consistently surpasses it, demonstrating significant speed advantages at every difficulty level. As noted in section 2, there are additional methods that can provide various performance attributes for solving a Sudoku puzzle. Our results emphasize the scalability and effectiveness of heuristic methods, particularly in tackling more complex puzzles. This study provides important understanding of the relative strengths of two different techniques and indicates that heuristic methods are particularly effective compared to backtracking for efficiently solving challenging Sudoku puzzles.

References

1. Simonis, H. (2005, October). Sudoku as a constraint problem. In *CP Workshop on modeling and reformulating Constraint Satisfaction Problems* (Vol. 12, pp. 13-27). Sitges, Spain: Citeseer.
2. Christopher, J. (2017). Solving Sudoku Puzzles using Backtracking Algorithms.
3. Pillay, Nelishia. (2012). Finding Solutions to Sudoku Puzzles Using Human Intuitive Heuristics. *SACJ*. 49. 25-34. 10.18489/sacj.v49i0.111.

4. Chatterjee, S., Paladhi, S., & Chakraborty, R. A Comparative Study On The Performance Characteristics Of Sudoku Solving Algorithms. *IOSR Journals (IOSR Journal of Computer Engineering)*, 1(16), 69-77.
5. B. Crawford, M. Aranda, C. Castro and E. Monfroy, "Using Constraint Programming to solve Sudoku Puzzles," *2008 Third International Conference on Convergence and Hybrid Information Technology*, Busan, Korea (South), 2008, pp. 926-931, doi: 10.1109/ICCIT.2008.154.
6. Norvig, P. (2006). Solving Every Sudoku Puzzle. <https://norvig.com/sudoku.html>
7. Rodrigues, C., Galvão, E., & Azevedo, R. LSVF: A heuristic search to reduce the backtracking calls when solving Constraint Satisfaction Problems. *SI nforme*, 19.
8. Kumar, V. (1992). Algorithms for Constraint-Satisfaction Problems: A Survey. *AI Magazine*, 13(1), 32–44.
9. C. Barker, J. Bridge, and P. Nightingale, “Combining Heuristics, Local Search and Backtracking for Sudoku,” in UK Workshop on Computational Intelligence, 2013.
10. Yato, T., & Seta, T. (2003). Complexity and completeness of finding another solution amand its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E86-A(5), 1052–1060.
11. Knuth, D.E. (2000). Dancing Links. *Journal of Algorithms*, 45(2), 123-145.
12. Sitorus, P., & Zamzami, E. M. (2020, June). An implementation of backtracking algorithm for solving a Sudoku-Puzzle based on Android. In *Journal of Physics: Conference Series* (Vol. 1566, No. 1, p. 012038). IOP Publishing.
13. Chen, Z. (2009). Heuristic reasoning on graph and game complexity of sudoku. *arXiv preprint arXiv:0903.1659*.
14. Silva, J. B. B., Siebra, C. A., & Nascimento, T. P. (2016). A simplified cost function heuristic applied to the A*-based path planning. *International Journal of Computer Applications in Technology*, 54(2), 96.
15. Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving* (pp. 112–115). Addison-Wesley.