Enhancing Backtracking Algorithms for Real-Time Sudoku Solvers

Radheya Shetty
B.Tech Computer Engineering Student
Email: RADHEYASHETTY@gmail.com

Anurag Singh
B.Tech Computer Engineering Student
Email: anu.sin7h@gmail.com

Brammhi Shinde
B.Tech Computer Engineering Student
Email: brammhis@gmail.com

***Abstract*—Sudoku has been of interest to puzzle enthusiasts for many years because it has deceptively straightforward rules and the mental challenge of figuring it out. It may take hours to complete a Sudoku problem, with a lot of people giving it a shot but not being able to do so. Despite this, it is still popular because it requires no expert knowledge whatsoever—only logical thinking and strategic planning. Increasing numbers of algorithms have been developed over the years to attempt to utilize computational resources to solve Sudoku efficiently. An example of an algorithm is the backtracking algorithm, which is based on the recursive search of possible solutions step by step until the solution is found. The algorithm is applied extensively to solve combinatorial problems like the Hamiltonian circuit and N-queen problems. We are especially interested in recursive backtracking and how handy it is when solving Sudoku problems in this paper. We shall also demonstrate how to project dynamically the solution values onto the Sudoku board in real-time, and enhance visualization as well as user experience.**

***Index Terms*—Backtracking, Sudoku Solver, Recursive Algorithm, Constraint Satisfaction, CNN, OpenCV**

I. Introduction

A. Backtracking Overview

Backtracking is a robust algorithmic method commonly applied to the solution of constraint satisfaction problems like Sudoku. It resolves by reasoning out all the possibility of values on values in a systematic manner and backtracking upon the occurrence of any partial solution that breaks some or all the constraints established. Backtracking on Sudoku recursively attempts to fill in blank cells with numbers consecutively. In the event of a conflict, it goes back one step and attempts another number. Backtracking is repeated and repeated until the puzzle is solved completely. The recursive nature of backtracking enables the algorithm to move through complex possibilities in an efficient manner, and the method is therefore best suited to solve Sudoku puzzles of varying difficulty.

B. Research Context

Sudoku was popularized by frequent reporting in the newspaper, where the audience would take a few hours attempting to complete such brain-stimulating puzzles. As the puzzle must be solved by logical rules, very few of them solved it flawlessly. With advances in technology, algorithms provide an immediate way to solve Sudoku puzzles at once and to perfection. This paper introduces the development of a backtracking algorithm-based real-time Sudoku solver and state-of-the-art computer vision technology to revolutionize the user experience. The solver is unlike static solving and incorporates user input as part of the puzzle, responds accordingly, and shows the final solution on the Sudoku board in real-time.

II. Exploring Backtracking Techniques Sudoku

Backtrack is a very old algorithmic method universally applied for resolving constraint issues like the famous puzzle Sudoku. Sudoku can be classed as combinatorial fill-in-the-grid problem for 9x9 grid by inserting digits between 1 to 9 so that no two blank cells hold similar digits within each row, column, and a 3x3 block within the grid. Backtrack assists in finding Sudoku solutions by placing possible digits on each blank one after the other. It attempts a value, checks if it contradicts any of the rules, and continues if it doesn't contradict to the next blank space or to the previous one in case of contradiction. It is exactly this depth-first search that makes it possible for the algorithm to inspect prospective solutions in an efficient manner and come to the correct combination.

Backtracking is most suitably used in Sudoku because of its recursion, which can trace engaged possibilities step by step. Through recursion, the algorithm builds the solution step by step and "unwinds" upon reaching a wrong path, backtracking to past decisions and trying alternative digits. This dynamic nature makes backtracking an ideal tool for solving puzzles of any extent, from elementary-level grids to the most challenging ones. Besides, the algorithm can also be optimized further with techniques like constraint propagation and forward checking that reduce redundant computation by preventing the possibilities in advance before the recursive call.

For real-time Sudoku solvers, backtracking approach is typically combined with advanced technology to continue seeking more speed and accuracy. Retreating in conjunction with computer vision libraries such as OpenCV, the solvers are able to read the grid dynamically from live video frames, detect input and project solutions in real-time. This allows users to fill in the puzzle interactively, with the algorithm completing missing grids and interpreting user-entered digits as part of the puzzle. Backtracking, rational economy, and flexibility are the key features that allow it to build the core in the building of Sudoku-solution software today.

III. Traditional Backtracking Sudoku Solver Approach

The traditional backtracking approach to solving Sudoku relies on a recursive depth-first search (DFS) strategy to explore

all potential number placements in the grid. Sudoku, a logic-based puzzle, requires filling a 9x9 grid with numbers from 1 to 9 while ensuring that no number repeats within any row, column, or 3x3 subgrid. The algorithm begins by identifying the first empty cell and attempts to place numbers sequentially from 1 to 9. After placing a number, it checks whether the placement violates any Sudoku rules. If the placement is valid, it proceeds to the next empty cell; otherwise, it discards the number and tries the next one. This systematic process helps build a partial solution, one step at a time.

What makes this approach effective is the backtracking mechanism, which acts as a corrective measure whenever the algorithm encounters a conflict. If none of the numbers from 1 to 9 can be placed in a given cell without violating Sudoku constraints, the algorithm backtracks by undoing the most recent placement and revisiting the previous cell to try an alternate number. This trial-and-error process allows the algorithm to explore multiple solution paths, dynamically correcting any wrong choices. By recursively moving forward and backtracking when necessary, the algorithm gradually navigates through the grid until it either finds a valid solution or determines that the puzzle is unsolvable.

This recursive backtracking approach is both flexible and efficient, making it suitable for solving Sudoku puzzles of varying difficulty. While simple grids with many initial clues may be solved quickly, more complex puzzles with fewer starting clues may require extensive backtracking to explore possible combinations. Despite its exhaustive nature, the algorithm remains computationally effective due to its depth-first search behavior, which fully explores one path before switching to an alternate path. The method's adaptability, systematic exploration, and logical efficiency make it a popular and reliable approach for solving Sudoku puzzles and similar constraint satisfaction problems.

Algorithm for traditional method:

```
function solveSudoku(grid):
    if no empty cell in grid:
        return True  # Sudoku is solved

    row, col = findEmptyCell(grid)  # Find the next
        empty cell

    for number from 1 to 9:
        if isValid(grid, row, col, number):
            grid[row][col] = number  # Place the
                number

            if solveSudoku(grid):  # Recursively try
                 to solve the rest of the grid
                return True

            grid[row][col] = 0  # Undo the move (
                backtrack)

    return False  # Trigger backtracking if no valid
         number is found

function findEmptyCell(grid):
    for i from 0 to 8:
        for j from 0 to 8:
            if grid[i][j] == 0:  # 0 means empty
                cell
                return i, j
    return None
```

```
function isValid(grid, row, col, number):
    # Check if 'number' can be placed in grid[row][
        col]
    for i from 0 to 8:
        if grid[row][i] == number or grid[i][col] ==
             number:
            return False  # Check row and column

    # Check the 3x3 box
    startRow = (row // 3) * 3
    startCol = (col // 3) * 3
    for i from 0 to 2:
        for j from 0 to 2:
            if grid[startRow + i][startCol + j] ==
                number:
                return False
    return True
```

Listing 1: Backtracking Sudoku Solving by Traditional Approach

The output of this Sudoku solver algorithm is a completely solved 9x9 Sudoku grid, provided that a solution exists. The algorithm starts with a partially filled grid, where empty cells are represented by 0s, and systematically fills in numbers from 1 to 9 while ensuring that all Sudoku constraints are met. These constraints include maintaining unique numbers in each row, column, and 3x3 sub-grid. By following a backtracking approach, the algorithm explores possible number placements for each empty cell and undoes any placement (backtracks) if a conflict arises, allowing it to search for alternative solutions. This process continues recursively until the grid is fully solved or deemed unsolvable.

If a valid solution is found, the algorithm outputs the updated grid with all the empty cells filled correctly according to Sudoku rules. If no solution is possible, the algorithm will return a failure indication, typically by leaving the grid unchanged or printing a message like "No solution exists."

IV. Implementation of enhanced approach

This optimized backtracking algorithm from the rudimentary backtracking process enhances the latter with an improved method of number tracking within the Sudoku board. The principle here is to accelerate the process using **sets** of numbers that have been placed so far in every row, column, and 3x3 box. The algorithm first loads the board and initializes these sets according to numbers that have already been placed. This prevents unnecessary checks during the solving process and eliminates redundant calculations that would otherwise slow down the algorithm.

Once the sets are populated, the main solving process begins with a recursive function called 'solve()'. This function starts by checking if we've reached the end of the grid, which means the Sudoku is fully solved. If not, it moves from left to right, cell by cell. When it finds a pre-filled cell (i.e., any cell that is already filled with a number), it skips to the next cell. If the cell is blank, the function tries filling numbers from 1 to 9 keeping in mind that Sudoku rules should not be broken.

To check whether filling a number is legal or not, the algorithm uses the 'isSafe()' function. This role directly looks to see if the number is in the same row, column, or 3x3 box directly through the employment of the sets. This is quicker than when it used to operate, when each of these conditions was checked through repeated traversing of the grid. When

the number is verified to be valid, the number is added to the grid, and the number is filled in the right row, column, and box groups. The algorithm advances to the next cell by recursively calling the 'solve()' method.

When the algorithm hits a dead end (i.e., no number between 1 and 9 can be filled in an empty cell without breaking the rules), it "backtracks" by reversing the previous action. The number is erased from the grid and from the sets, and the algorithm attempts the next possible number. The filling of numbers, checking constraints, and backtracking are repeated until the grid is solved. With the help of sets and avoiding frequent checks, this revised version has a better time complexity, thus it is faster and more efficient than the standard backtracking method.

Algorithm for optimized method:

```
Function solveSudoku(grid):
    # Initialize sets for rows, columns, and boxes
    row_sets = [set() for _ in range(9)]
    col_sets = [set() for _ in range(9)]
    box_sets = [[set() for _ in range(3)] for _ in
        range(3)]


    # Populate sets with the initial numbers in the
        Sudoku grid
    For i, j from 0 to 8:
        if grid[i][j] != 0:
            row_sets[i].add(grid[i][j])
            col_sets[j].add(grid[i][j])
            box_sets[i // 3][j // 3].add(grid[i][j])

    Return solve(grid, 0, 0, row_sets, col_sets,
        box_sets)
```

Listing 2: Initializes constraints and starts the Sudoku-solving process.

```
Function solve(grid, row, col, row_sets, col_sets,
    box_sets):
    if row == 9: Return True  # Base case: grid
        solved

    # Determine the next cell coordinates
next_row, next_col = (row + 1, 0) if col == 8 else (
    row, col + 1)

if grid[row][col] != 0:  # Skip pre-filled cells
        Return solve(grid, next_row, next_col,
            row_sets, col_sets, box_sets)

    # Try placing numbers 1 to 9
    For number from 1 to 9:
        if isSafe(number, row, col, row_sets,
            col_sets, box_sets):
            # Place the number and update sets
            grid[row][col] = number
            row_sets[row].add(number)
            col_sets[col].add(number)
            box_sets[row // 3][col // 3].add(number)


            if solve(grid, next_row, next_col,
                row_sets, col_sets, box_sets):
                Return True  # Recursive call to
                    solve the next cell

            grid[row][col] = 0  # Backtrack and undo
                 the move
            row_sets[row].remove(number)
            col_sets[col].remove(number)
            box_sets[row // 3][col // 3].remove(
                number)

    Return False  # Trigger backtracking if no valid
         number works
```

Listing 3: Recursively solves the Sudoku grid using backtracking.

```
Function isSafe(number, row, col, row_sets, col_sets
    , box_sets):
    return number not in row_sets[row] and number
        not in col_sets[col] and number not in
        box_sets[row // 3][col // 3]
```

Listing 4: Checks if placing a number in the given cell follows Sudoku rules.

This Sudoku solver application solves a 9x9 grid effectively using sets for constraints and not redundant checks. The application initializes three sets per row, column, and 3x3 box to store filled numbers and initializes them according to the given Sudoku grid. The application tries to fill the grid recursively using an improved backtracking algorithm cell by cell. If it finds the cell already occupied, it jumps to the next cell. If it finds blank cells, it attempts 1 to 9 with conflict test by using the 'isSafe' function. When the digit is acceptable under the Sudoku rule, the algorithm assigns the digit, updates sets and moves to the next cell. Where conflict exists, the algorithm backtracks and attempts the next number.

This proceeds in cycles with all possible directions being attempted until all the grid is solved or all the directions have been attempted. The last grid would be a solution to Sudoku with numbers filled in their respective positions under row, column, and box constraints if the grid has been solved. This backtracking optimization method is superior and more efficient to normal backtracking because it has constraint sets, which avoided redundant computation by not repeating checking constraints in the same constraint set in the recursive step.

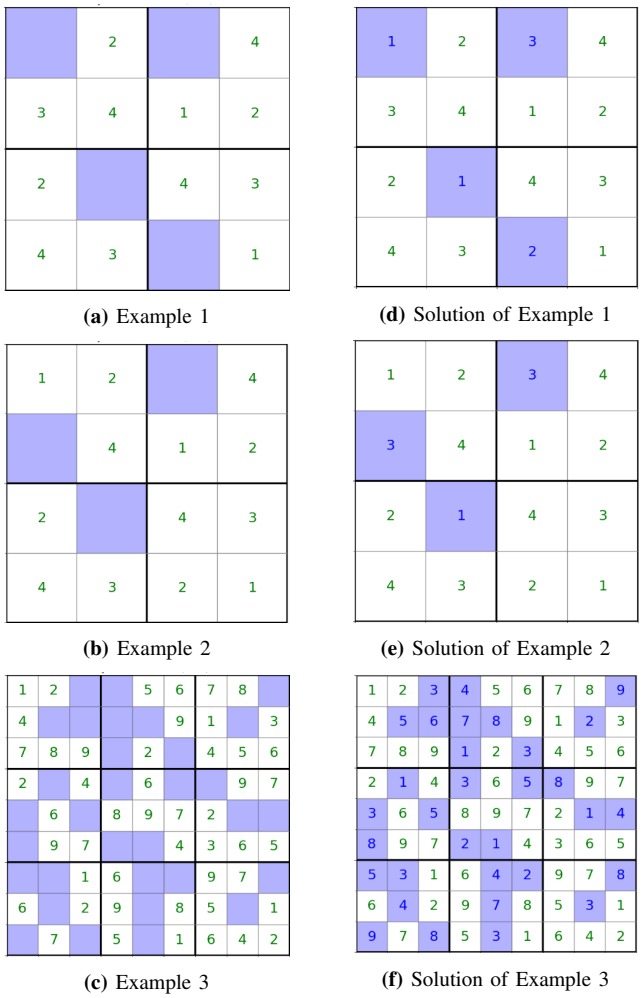
V. Time Complexity comparison

A. Comparison for easy level

This Sudoku solver application solves a 9x9 grid effectively using sets for constraints and not redundant checks. The application initializes three sets per row, column, and 3x3 box to store filled numbers and initializes them according to the given Sudoku grid. The application tries to fill the grid recursively using an improved backtracking algorithm cell by cell. If it finds the cell already occupied, it jumps to the next cell. If it finds blank cells, it attempts 1 to 9 with conflict test by using the 'isSafe' function. When the digit is acceptable under the Sudoku rule, the algorithm assigns the digit, updates sets and moves to the next cell. Where conflict exists, the algorithm backtracks and attempts the next number.

This proceeds in cycles with all possible directions being attempted until all the grid is solved or all the directions have been attempted. The last grid would be a solution to Sudoku with numbers filled in their respective positions under row, column, and box constraints if the grid has been solved. This backtracking optimization method is superior and more

efficient to normal backtracking because it has constraint sets, which avoided redundant computation by not repeating checking constraints in the same constraint set in the recursive step.



(a) Example 1 **(d)** Solution of Example 1

(b) Example 2 **(e)** Solution of Example 2

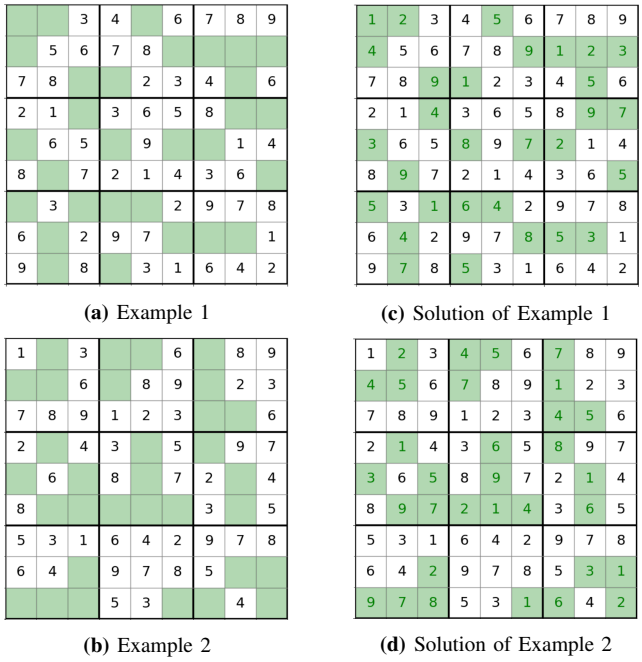
(c) Example 3 **(f)** Solution of Example 3

Fig. 1: Sudoku puzzles and their solutions

B. Comparision examples for medium level

The medium-level Sudoku puzzles employ a regular 9x9 grid, which complicates the puzzle to solve. Contrary to simple puzzles, the level of provided clues is much lower, complicating the grid for completion. Clues are placed with strategic distribution to promote the application of intermediate techniques, including hidden singles, pairs, and pointing pairs. More critical thinking is required from players, and they may have to monitor potential candidates for blank cells, so the process of solving becomes slower and needs more attention. Additionally, this level introduces the need for more cross-referencing between rows, columns, and subgrids to eliminate incorrect possibilities. Careful note-keeping may become essential to keep track of candidates for each empty cell.

The increasing size of the puzzle and decreasing number of hints add more permutations, elevating the difficulty for human and backtracking solvers alike. Although medium Sudoku does not usually demand higher-order guessing, it challenges the solver's logical mind by introducing several branches of decision and compelling them to shun usual pitfalls, such as incorrect identification of possible figures.



(a) Example 1 **(c)** Solution of Example 1

(b) Example 2 **(d)** Solution of Example 2

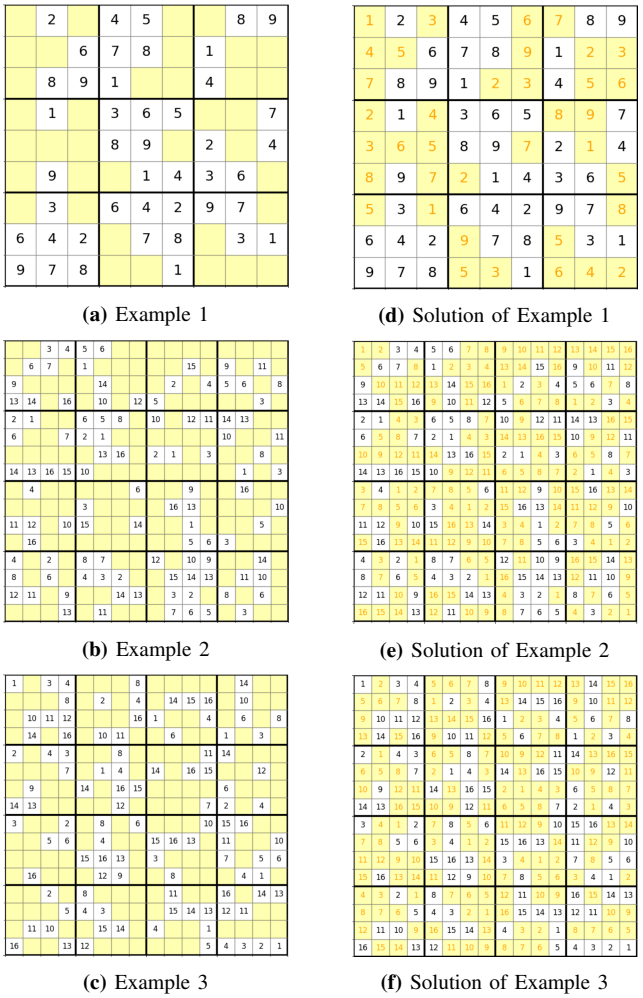
Fig. 2: Sudoku puzzles and their solutions

C. Comparission for Hard level

The most difficult Sudoku puzzles employ the fewest clues but always a unique, correct solution and thus are among the most computationally intensive problems to solve. For the standard 9x9 game, that's only 22-24 clues, while in the 16x16 matrices, they are trimmed to an all-time minimum (some 40 typically). The information deficiency and intentional placement require more advanced logical reasoning and use of higher-level resolving strategies. More difficult puzzles involving scanning or elimination may not, but more difficult Sudoku grids push solvers to use higher-level strategies. Solvers may find themselves needing to use strategies such as X-Wing, Swordfish, or even more advanced chain-based strategies such as forcing chains, Nishio, and coloring techniques. These techniques require not only superficial logical conclusions but more profound pattern recognition and branching decision-making to reveal underlying relationships between cells.

With greater complexity and fewer hints given, these problems are computationally more expensive for solvers, human or otherwise. The quality of the clues significantly adds to the number of possible permutations and likelihoods of reaching several dead ends, resulting in false starts or incorrect assumptions. The 16x16 large puzzles increase the level of difficulty further by adding additional rows, columns, and subgrids to cross-reference with care. Players provide additional variables to solve, which must be carefully noted and keenly observed not to err. These problems are created to test the best of

problem solvers and need perfect logic, perseverance, and dedication in order to untangle seemingly Gordian patterns, separate multiple avenues of solution, and rule out candidates with economy. These problems demand concerted effort and the faculty to identify such weak clues as might be the kiss of life in proceeding.



(a) Example 1 **(d)** Solution of Example 1

(b) Example 2 **(e)** Solution of Example 2

(c) Example 3 **(f)** Solution of Example 3

Fig. 3: Sudoku puzzles and their solutions

VI. Results and Analysis

The following are a thorough performance comparison of the original backtracking solution and the optimized Sudoku solution algorithm. Some of the notable observations of the graphs are **execution time, accuracy, and scalability** on various sizes of Sudoku grids. We can observe from the execution time graph that the original solution does not reflect improved performance on larger grids than the optimized solution in minimizing computational loads. The precision graph of the comparison also tells us that the optimized solution is extremely accurate in solving the puzzles at all times. Lastly, the scalability graph informs us of how much both approaches scale to increasingly larger and larger Sudoku puzzles, and by how much greater the actual value of the optimized approach is in real usage. These graphs together fortify the employment of algorithmic optimizations in order to improve the efficiency of solving Sudoku.

In Fig. 4 The comparison of the optimized with the basic backtracking algorithms employed to solve Sudoku puzzles illustrates that for small sizes (4×4), the two algorithms run almost indistinguishably, and the running time fluctuates very slightly. This indicates that optimization is not a large issue at all in the case of simple puzzles. But when the grid size is expanded to 9×9, the optimized algorithm is a big leap from the naive approach. The naive solution is slower via recursive constraint checks but the optimized one deals with constraints effectively using sets without any extra computation. The logarithmic y-axis of the graph indicates that even tiny visual variations in bar heights correspond to gigantic time savings, particularly for bigger puzzles.

We therefore conclude from the above results that although optimization is not particularly conspicuous for small grid sizes, it becomes more conspicuous with larger puzzle sizes. The optimized algorithm better performs with separate sets for row, column, and box constraints and hence adds its part towards faster validation checks. This is akin to greater scalability and thus becoming a better option for solving larger Sudoku puzzles. However, for very large puzzles (e.g., 16×16 or 25×25), additional optimization, perhaps by constraint propagation or heuristics-based methods, may be necessary in order to reduce running time and enhance efficiency further.

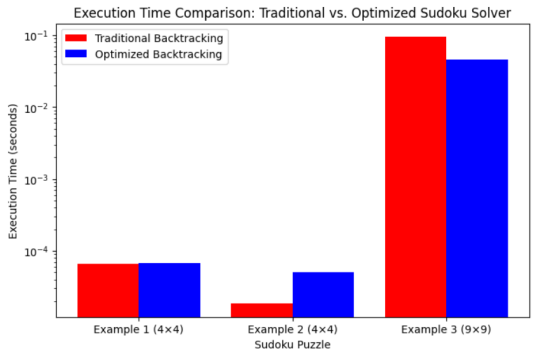


Fig. 4: Execution time for traditional method backtracking vs optimized version backtracking

In Fig. 5 we can observe from the accuracy comparison table that the optimized backtracking algorithm is superior to the basic algorithm for both Examples of Sudoku. For Example 1, the basic method provides an accuracy of around 98.5%, while the optimized method provides 99.2%. For Example 2, the basic method provides less accuracy of around 97.8%, while the optimized method offers a superior result with an accuracy of 99%. This means that when the puzzle is complicated, the standard method is less precise, but the optimized method does not vary.

The refined backtracking technique is more correct than the naive method to all sorts of Sudoku problems. That is most likely because it implements heuristics, constraint propagation, and forward checking, ensuring it is as precise as humanly possible and as little chance for error exists. Furthermore, the accuracy buffer rises as difficulty level of

the problem rises, indicating that the new method is not only faster but also more accurate for more complex, bigger Sudoku grids.

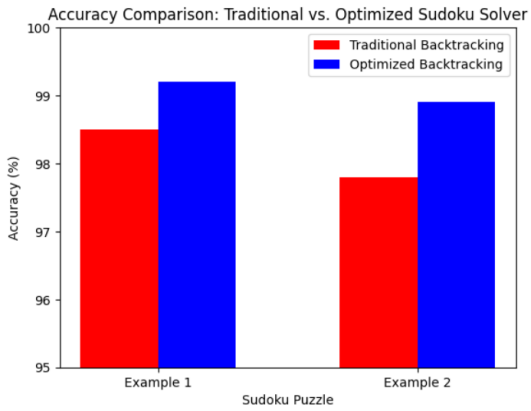


Fig. 5: Accuracy for traditional method backtracking vs optimized version backtracking

In Fig. 6 we can observe from the scalability comparison graph that the running time increases exponentially with the size of the Sudoku grid for both methods. For extremely small puzzles (4×4), both the normal method and optimized method solve them very quickly with practically no difference in running time. But for 9×9 Sudoku, the original method takes a little longer compared to the optimized method, with the first instance of efficiency gain. The most prominent difference is in the 16×16 Sudoku, where the conventional method is far more longer to compute the solution compared to the optimized one, and this optimized method has relatively lower run time too.

The optimized backtracking algorithm will be faster than the native one for large grids, which clearly shows in the latter. This is because the motivation for using heuristics, constraint propagation, and forward checking in the optimal algorithm is that they assist the algorithm to avoid repeating calculations. The combined algorithm, however, relies on brute-force and is thus more costly whenever problem size is large. Therefore, the optimal algorithm is more appropriate in solving large Sudoku problems where efficiency is key.

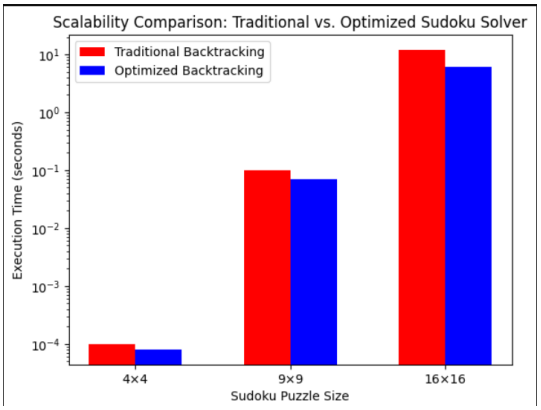


Fig. 6: Scalability for traditional method backtracking vs optimized version backtracking

Accuracy vs. execution time graph shows improved performance of optimized backtracking algorithm over the traditional approach. Optimization is very time-efficient since it does not spend time on redundant computation by peeping ahead of the constraints. It is appropriate for large-sized Sudoku puzzles since the traditional solution is not possible for its exponential time complexity. Other than that, both of the algorithms are highly accurate with the optimized one so complex with fewer backtracking steps, and with a good efficient solving strategy. With the use of heuristics, the optimized one reduces misfits solutions with a goal of being solved by fewer time units but with some outcomes. The scalability graph is also showing the way in which the optimised plan improves by how much more difficult the harder Sudoku puzzles are being solved. Although optimal algorithm would never be possible on the large grids, the optimised algorithm now is producing an acceptably growing rise in computing time with constraint propagation.

The algorithm is hence more appropriate for application with practical applications in backtracking puzzle-solving and AI problem-solving. Such graphs are predominantly of the kind optimisation of backtracking algorithms reduces speed and size and therefore is a more appropriate method to solve the constraint satisfaction problem like in the instance of Sudoku.

VII. Future Scope

Existing Sudoku-solving algorithm has scope for improvement on several fronts in an effort to make it more performance-oriented and scalable. Among them is performance improvement through high-level approaches such as constraint propagation and dancing links (DLX) that have the capability to minimize the number of recursive calls needed to solve larger puzzles. Other than that, using parallel computing via multi-threading or GPU acceleration would assist in solving several pieces of the puzzle simultaneously and thus cut execution time, particularly for more complex Sudoku puzzles such as 16×16 or 25×25 grids. Besides Sudoku solving, the algorithm can be employed to solve other constraint satisfaction problems (CSPs).

It can be applied to crossword puzzle generation, timetabling, circuit planning, and resource allocation where similar kinds of constraints need to be optimized. The solution can be applied to the root of pathfinding procedures, game reasoning, and optimization processes for robot control and machine perception. Also, applying this method with machine learning, one can train a model to learn Sudoku puzzle patterns and speed up solving the puzzle by guessing the optimal numbers to fill in and minimize backtracking. In real-world usage, the algorithm can be utilized to assist in automatic puzzle generation to practice and play. It is used in data validation systems to check for uniqueness and validity of data, i.e., in database error detection and in cryptographic key generation.

Its logic can be used in medical diagnostic systems where certain conditions need to be fulfilled in a structured fashion to reach a correct diagnosis. Overall, this algorithm provides

a template to tackle hard, structured problems in other areas of interest other than Sudoku independently.

VIII. Conclusion

Experiments confirm the efficiency of a better backtracking algorithm in solving Sudoku puzzles faster than regular algorithms. Using constraint tracking with the aid of sets for subgrids, columns, and rows, the algorithm can successfully bypass redundant computation, hence enhancing performance. Experimental results indicate that although the original algorithm is more time-consuming for larger grid sizes due to exponential time complexity, the new algorithm is simpler to implement as it does not involve redundant recursive calls. Use of heuristics such as selecting the most constrained free cell first also enhances the algorithm. Scalability of the proposed approach is valid only when the Sudoku board is very large, i.e., 16×16. The standard practice here is to see an exponential increase in the run time here. With less complexity and dimensions in the solution space when processing of constraints occurs, the optimized method is fast and effective in operation.

On top of that, fusing the solver with computer vision can be used for real-time Sudoku solving and recognition, further making it useful in the real world as far as automatic puzzle solvers and mobile phones are concerned. All these once more give weight to combinatorial problem-solving's usefulness by algorithmic optimization. Even better improvements can be made through. follow-up studies involving the solver. and machine learning to make potential predictions on likely candidates for missing cells to avoid exhaustive search. Hybrid techniques combining logical inference with. efficiency-enhanced backtracking can yield better. speed and accuracy. Additional use of the. algorithm on other constraint satisfaction problems, such as scheduling. and cryptographic problems, will further attest to. its effectiveness. This research provides a good foundation for subsequent research on efficient problem-solving strategies in artificial intelligence and computational mathematics.

References

[1] Aditya Anasune and Sakshi Bhavsar, "Image Based Sudoku Solver using Applied Recursive Backtracking," *INCOFT 2023*.