

A
CIA REPORT

ON

Sudoku Solver using Backtracking

Submitted by,

11] Badjate Aditya

12] Badjate Vaishnavi

26] Chaudhari Ashwini

27] Chaudhari Dhanshri

(TY BTECH COMPUTER)

[DIVISION A]

Guided by,

Prof. P. B. Dhanwate



Subject- Design and Analysis of Algorithms

In Academic Year 2024-25

Department of Computer Engineering

Sanjivani College of Engineering, Kopergaon

**Sanjivani College of
Engineering, Kopargaon**

CERTIFICATE

This is to certify that

Badjate Aditya

Badjate Vaishnavi

Chaudhari Ashwini

Chaudhari Dhanshri

(T.Y. Computer)

Successfully completed their CIA Report on

Sudoku Solver using Backtracking

Towards the partial fulfilment of

Bachelor's Degree in Computer Engineering

During the academic year 2024-25

Prof. P. B. Dhanwate
[Guide]

Dr. D. B. KSHIRSAGAR
[H.O.D. Comp Engg]

Dr. A. G. THAKUR

[Director]

CONTENTS

Sr. No.	Chapter	Page No.
1.	Introduction	05
2.	Problem Statement & Requirement Analysis	08
3.	Methodology	10
4.	Design & Implementation of a Sudoku Solver	14
5.	Result & Discussion	17
6.	Conclusion	18
10.	References	19

CIA Activity: Sudoku Solver Using Backtracking Algorithm

Abstract

Problem (CSP), requires completing a 9×9 grid such that each row, column, and 3×3 subgrid contains all digits from 1 to 9 exactly once, without any repetition. The proposed solution utilizes backtracking to recursively test possible numbers for empty cells, validate their placement against the puzzle's constraints, and backtrack if conflicts arise. This approach ensures an exhaustive search for valid solutions while maintaining computational efficiency. The solver demonstrates its capability to handle puzzles of varying complexity, including those with minimal initial clues, unsolvable cases, and multiple-solution scenarios. Through a detailed breakdown of the algorithm, this study highlights its strengths, such as reliability and adaptability, while addressing areas for optimization. Future enhancements, including constraint propagation, heuristic-based decision-making, and advanced optimization techniques, are discussed to improve its speed and effectiveness. This research underscores the potential of backtracking algorithms in solving constraint-based puzzles like Sudoku.

Keywords

Sudoku, Backtracking Algorithm, Constraint Satisfaction Problem (CSP), Sudoku Solver, Algorithm Optimization

1. Introduction

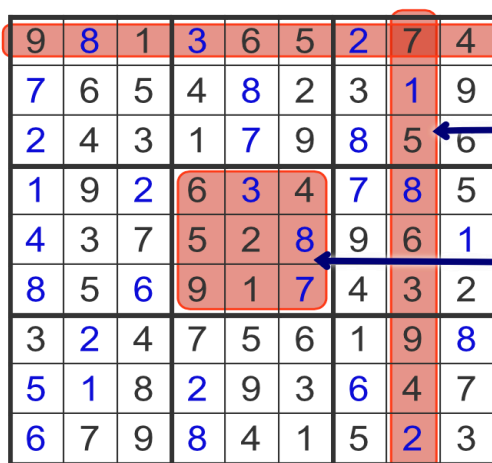
Sudoku is a widely celebrated puzzle game that has captured the interest of enthusiasts worldwide due to its simple rules yet challenging problem-solving requirements. The puzzle consists of a **9×9 grid**, divided into nine smaller **3×3 subgrids**. The player's goal is to fill this grid with numbers from **1 to 9** in such a way that:

1. Each row contains each number exactly once.
2. Each column contains each number exactly once.
3. Each 3×3 subgrid contains each number exactly once.

Despite its straightforward structure, Sudoku offers a profound challenge that lies in its complexity and the logical reasoning required to solve it. The difficulty of a Sudoku puzzle often depends on the number of clues—numbers pre-filled in the grid—which act as starting points. While some puzzles may be easy and solvable through simple logic, others are more challenging and require advanced strategies or computational assistance.

1.1 The Appeal of Sudoku

Sudoku puzzles are a prime example of how games can serve as intellectual stimulants. They enhance logical reasoning, pattern recognition, and critical thinking skills. Additionally, Sudoku puzzles are approachable for people of all ages, requiring no specialized knowledge or mathematical computations beyond basic counting. This universal appeal has made Sudoku a staple in newspapers, online platforms, and mobile applications.



9	8	1	3	6	5	2	7	4
7	6	5	4	8	2	3	1	9
2	4	3	1	7	9	8	5	6
1	9	2	6	3	4	7	8	5
4	3	7	5	2	8	9	6	1
8	5	6	9	1	7	4	3	2
3	2	4	7	5	6	1	9	8
5	1	8	2	9	3	6	4	7
6	7	9	8	4	1	5	2	3

Sudoku puzzles require you to find the missing numbers in a 9 by 9 grid, with that grid itself divided into 9 square grids of 3 by 3.

You can't just add any numbers, though. There are rules that making solving the puzzle challenging.

A number can only occur once in a row, column, or square.

To solve a Sudoku, look for open spaces where its row, column and square already have enough other numbers filled in to tell you the correct value. The more squares you fill in, the easier the puzzle is to finish!

1.2 Sudoku as a Constraint Satisfaction Problem (CSP)

From a computer science perspective, solving Sudoku can be categorized as a **Constraint Satisfaction Problem (CSP)**. A CSP involves finding a solution to a problem while adhering to a set of predefined constraints. In the case of Sudoku:

- The constraints are the rules that govern the placement of numbers in the grid.
- The solution involves filling all empty cells in the grid while satisfying these constraints.

CSPs are critical in various fields, including artificial intelligence, optimization, and operations research, as they provide a structured framework to solve problems involving multiple interdependent variables.

1.3 Computational Challenges in Sudoku

Although humans often solve Sudoku puzzles using logical deductions and heuristic strategies, creating a computational approach to solve Sudoku involves addressing several challenges:

1. **Constraint Checking:** Ensuring that each number placement adheres to the rules of Sudoku.
2. **Recursive Exploration:** Exploring possible solutions systematically, as there are often multiple valid paths to consider at intermediate steps.
3. **Backtracking:** Efficiently handling dead ends by retracing steps and exploring alternative possibilities.
4. **Optimization:** Reducing computational effort by employing techniques like constraint propagation and heuristics.

These challenges make Sudoku a fascinating case study for algorithm design and optimization techniques.

1.4 The Role of Backtracking in Sudoku Solving

One of the most effective algorithms for solving Sudoku puzzles programmatically is the **backtracking algorithm**. Backtracking is a systematic method of exploring all possible solutions to a problem by incrementally building candidates and abandoning those that fail to satisfy the constraints. It is particularly well-suited for problems like Sudoku, where:

- The solution space is finite but vast.
- Constraints can be evaluated locally at each step.
- Incorrect decisions can be reversed efficiently.

1.5 Objectives of the Paper

This paper focuses on implementing a Sudoku solver using the backtracking algorithm, emphasizing the following objectives:

- To design and implement a computational model for solving Sudoku puzzles programmatically.
- To provide an in-depth understanding of the backtracking algorithm and its application in constraint satisfaction problems.
- To analyze the efficiency and limitations of the algorithm, particularly for puzzles of varying difficulty levels.
- To discuss potential optimizations and future directions for enhancing the performance of Sudoku solvers.

By combining theoretical insights with practical implementation, this study aims to highlight how computational methods can solve complex logical problems like Sudoku.

2.Problem Explanation and Requirement Analysis

Sudoku is a logic-based, combinatorial puzzle that poses an intriguing computational challenge due to its structured constraints. The puzzle involves a 9×9 grid divided into smaller 3×3 subgrids. The objective is to fill the grid with digits from 1 to 9 such that every row, every column, and every 3×3 subgrid contains each digit exactly once. The constraints create a well-defined problem space where each placement of a digit must satisfy specific rules to maintain the validity of the solution.

This problem can be modeled as a **Constraint Satisfaction Problem (CSP)**, where each empty cell in the grid is a variable, and the constraints are defined by the puzzle rules. Solving this requires ensuring that the number assigned to any cell does not repeat within its row, column, or subgrid. A partial solution that violates these constraints necessitates reevaluation, making a recursive and systematic approach like the backtracking algorithm highly suitable

The backtracking algorithm addresses these requirements by attempting to fill each empty cell with a valid number while continuously verifying the constraints. If a number violates the rules, the algorithm backtracks, removing the previous placement and trying a different number. This iterative process continues until the grid is completely filled or determined to be unsolvable.

The requirements for implementing the Sudoku solver include:

1. **Input Representation:** A partially filled 9×9 grid where empty cells are represented by a placeholder value (e.g., 0).
2. **Constraint Checking:** Functions to verify the validity of number placements in rows, columns, and subgrids.
3. **Recursive Backtracking Logic:** A mechanism to explore possible solutions by placing numbers, checking constraints, and backtracking when necessary.
4. **Edge Case Handling:** Addressing scenarios such as unsolvable puzzles, minimal clue puzzles, or multiple-solution puzzles to ensure robustness.

By framing Sudoku as a CSP and leveraging the systematic nature of backtracking, the problem is efficiently tackled while adhering to the requirements of precision and adherence to rules.

Example Illustration

To better understand the constraints, consider the diagram below:

In this example, an attempt is made to place the digit "7" in a specific empty cell. However, as highlighted by the red circles, the number "7" already exists in the same row, column, and subgrid. This violates the constraints, making the placement invalid. The green circle indicates a valid placement because it adheres to all constraints.

The visual representation underscores the importance of constraint validation during the solving process and illustrates how violations are identified.

By framing Sudoku as a CSP and leveraging the systematic nature of backtracking, the problem is efficiently tackled while adhering to the requirements of precision and adherence to rules.

6		5		3	8			7
2			9	6	5			
	4		7		1		6	
8	3	6				4		1
7			3	4		8		
		4			6			
3				7		9		2
	7		2	8	3	6		5
4	6			5	9	7		3

3.Methodology

The methodology for implementing a Sudoku solver revolves around designing a systematic approach to fill a partially completed 9×9 Sudoku grid while adhering to the puzzle's rules. In this study, the problem is approached as a **constraint satisfaction problem (CSP)**, and the **backtracking algorithm** is chosen as the primary technique to achieve the solution. This section provides a comprehensive breakdown of the methodology, the steps involved in applying the backtracking algorithm, and considerations for efficiency and edge cases.

- The backtracking algorithm proceeds with the following steps:

2.1 Problem Representation

The input to the Sudoku solver is a **9×9 grid**, where each cell contains either:

- A **number between 1 and 9**, representing a pre-filled value (a clue).
- A **0**, representing an empty cell to be solved.

For computational purposes, the grid is represented as a **2D matrix** in which:

- Rows are indexed from 0 to 8.
- Columns are indexed from 0 to 8.
- A subgrid is identified by its position in the overall grid (e.g., the top-left subgrid corresponds to rows 0–2 and columns 0–2).

2.2 Objective

The goal is to fill all the empty cells in the grid such that:

4. Each row contains the numbers **1 to 9** exactly once.
5. Each column contains the numbers **1 to 9** exactly once.
6. Each 3×3 subgrid contains the numbers **1 to 9** exactly once.

The final output is either:

- A completed Sudoku grid that satisfies the constraints, or
- A declaration that the puzzle is **unsolvable**, meaning no configuration satisfies all constraints.

2.3 The Backtracking Algorithm

The **backtracking algorithm** is a recursive, trial-and-error method that systematically explores possible solutions while adhering to constraints. It is particularly effective for solving Sudoku puzzles due to its ability to handle large solution spaces efficiently by eliminating invalid configurations early.

2.3.1 Steps of the Algorithm

1. Grid Representation

- The Sudoku puzzle is represented as a **9×9 matrix** of integers. Each element in the matrix corresponds to a cell in the grid.
 - Cells containing pre-filled values are treated as fixed and unmodifiable. Empty cells are initialized with the value **0**.
2. **Identifying Empty Cells**
 - The algorithm scans the grid to locate the first empty cell (value 0).
 3. **Placing Digits in the Empty Cell**
 - For the identified empty cell, the algorithm attempts to place digits from **1 to 9**, one at a time.
 4. **Constraint Checking**
 - Before placing a digit, the algorithm checks whether the digit violates any of the following constraints:
 - **Row Constraint:** The digit must not already exist in the same row.
 - **Column Constraint:** The digit must not already exist in the same column.
 - **Subgrid Constraint:** The digit must not already exist in the corresponding 3×3 subgrid.
 5. **Recursive Call**
 - If a valid digit is placed in the current cell, the algorithm recursively attempts to solve the next empty cell.
 6. **Backtracking**
 - If no valid digit can be placed in the current cell, the algorithm **backtracks** by:
 - Removing the last placed digit (reverting the cell to 0).
 - Trying the next possible digit for the previous cell.
 7. **Termination**
 - The algorithm continues this process until:
 - The grid is fully filled, meaning a solution is found.
 - All possibilities are exhausted, meaning the puzzle is unsolvable.

2.3.2 Pseudocode for Backtracking

Python

```
def solve_sudoku(grid):
    # Locate the first empty cell
    row, col = find_empty_cell(grid)
    if not row: # No empty cells left, solution found
        return True

    # Try placing digits 1 through 9
    for num in range(1, 10):
```

```

if is_valid(grid, row, col, num):
    grid[row][col] = num # Place the digit

    # Recursively solve the next cell
    if solve_sudoku(grid):
        return True

    # Backtrack if placement fails
    grid[row][col] = 0

return False # Trigger backtracking

```

2.4 Key Components of the Algorithm

1. Constraint Validation

- Implementing functions to verify the row, column, and subgrid constraints ensures that only valid numbers are placed in each cell.

2. Recursive Exploration

- The recursive structure of backtracking allows the algorithm to explore all possible configurations efficiently.

3. Backtracking Mechanism

- Backtracking enables the algorithm to handle incorrect placements by reversing decisions and exploring alternative possibilities.

4. Optimization Considerations

- Prioritizing cells with the fewest possibilities (minimum remaining values) can significantly reduce the solution space and improve efficiency.

2.5 Complexity Analysis

The complexity of the backtracking algorithm depends on the number of empty cells and the difficulty of the puzzle:

- **Worst Case:** The algorithm may explore all possible configurations, leading to exponential complexity $O(9^n)O(9^n)O(9^n)$, where n is the number of empty cells.
- **Optimizations:** Employing heuristics such as constraint propagation and minimum remaining values can significantly reduce the search space and improve performance.

2.6 Edge Cases

1. Multiple Solutions:

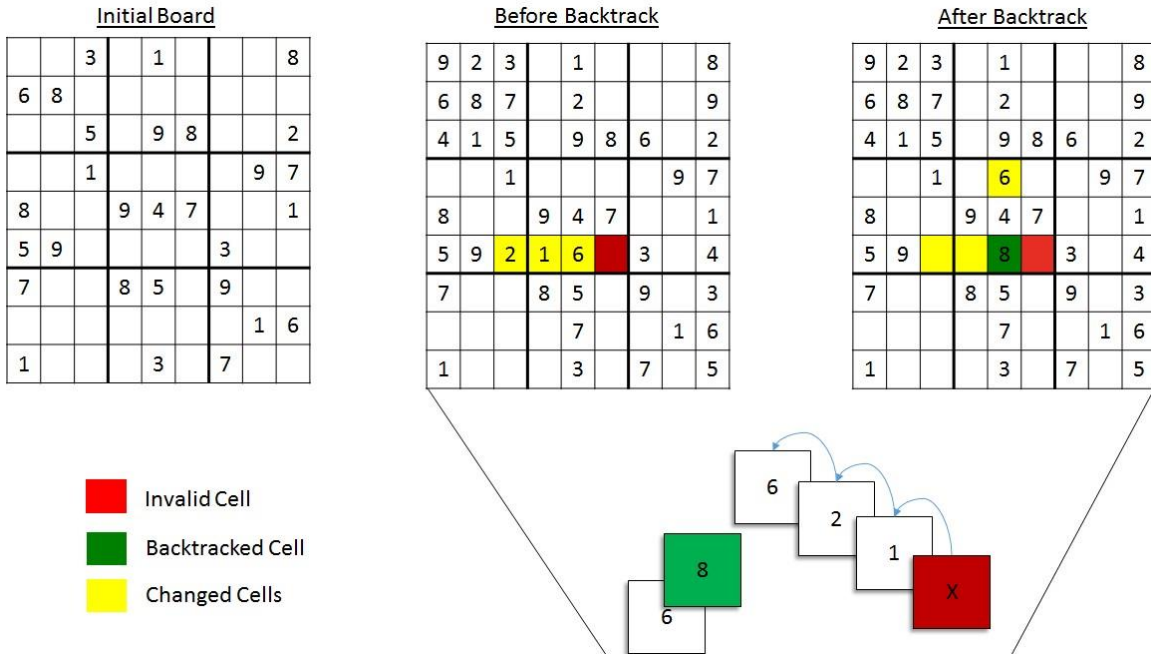
- Some puzzles have multiple valid solutions. The algorithm finds one solution but can be modified to explore all possibilities if needed.

2. Unsolvable Puzzles:

- The algorithm identifies puzzles that have no valid solutions and terminates without unnecessary computation.

3. Minimal Clues:

- Puzzles with very few initial clues can be more challenging but are solvable with sufficient computational effort.



4. Design and Implementation of a Sudoku Solver

Let's solve a partially filled Sudoku grid step by step using the backtracking algorithm.

Initial Sudoku Puzzle

```
5 3 0 | 0 7 0 | 0 0 0
6 0 0 | 1 9 5 | 0 0 0
0 9 8 | 0 0 0 | 0 6 0
-----+-----+-----
8 0 0 | 0 6 0 | 0 0 3
4 0 0 | 8 0 3 | 0 0 1
7 0 0 | 0 2 0 | 0 0 6
-----+-----+-----
0 6 0 | 0 0 0 | 2 8 0
0 0 0 | 4 1 9 | 0 0 5
0 0 0 | 0 8 0 | 0 7 9
```

Here, the 0s represent empty cells that need to be filled.

Step-by-Step Solution

1. Step 1: Start with the first empty cell (row 0, column 2).

- Try numbers 1 to 9.
- Check the constraints:
 - Number 1 conflicts with row 0 (already has 1).
 - Number 2 conflicts with row 0 (already has 2).
 - Number 3 conflicts with column 2 (already has 3).
 - **Number 4 satisfies all constraints.**
- Place 4 in the cell.

Updated Grid:

```
5 3 4 | 0 7 0 | 0 0 0
6 0 0 | 1 9 5 | 0 0 0
0 9 8 | 0 0 0 | 0 6 0
-----+-----+-----
8 0 0 | 0 6 0 | 0 0 3
4 0 0 | 8 0 3 | 0 0 1
7 0 0 | 0 2 0 | 0 0 6
-----+-----+-----
0 6 0 | 0 0 0 | 2 8 0
0 0 0 | 4 1 9 | 0 0 5
0 0 0 | 0 8 0 | 0 7 9
```

2. Step 2: Move to the next empty cell (row 0, column 3).

- Try numbers 1 to 9.
- **Number 6 satisfies all constraints.**
- Place 6 in the cell.

Updated Grid:

5 3 4	6 7 0	0 0 0
6 0 0	1 9 5	0 0 0
0 9 8	0 0 0	0 6 0
-----+-----+-----		
8 0 0	0 6 0	0 0 3
4 0 0	8 0 3	0 0 1
7 0 0	0 2 0	0 0 6
-----+-----+-----		
0 6 0	0 0 0	2 8 0
0 0 0	4 1 9	0 0 5
0 0 0	0 8 0	0 7 9

3. Continue the process.

- For each empty cell, try placing numbers 1–9.
- Validate each number against the row, column, and subgrid constraints.
- Backtrack if no valid number can be placed and revisit the previous cell to try a different number.

4. Final Solved Sudoku:

5 3 4	6 7 8	9 1 2
6 7 2	1 9 5	3 4 8
1 9 8	3 4 2	5 6 7
-----+-----+-----		
8 5 9	7 6 1	4 2 3
4 2 6	8 5 3	7 9 1
7 1 3	9 2 4	8 5 6
-----+-----+-----		
9 6 1	5 3 7	2 8 4
2 8 7	4 1 9	6 3 5
3 4 5	2 8 6	1 7 9

Architecture of the Problem

1. **Input Layer:**
 - **Input:** A 9×9 grid with partially filled numbers, where 0 represents empty cells.
 - **Output:** The filled Sudoku grid that satisfies all constraints.
2. **Core Logic Layer:**
 - **Backtracking Algorithm:**
 - Iterate over each cell in the grid.
 - Try placing numbers from 1 to 9.
 - Validate constraints (row, column, subgrid).
 - Recursively fill the next cell.
 - Backtrack if no valid number exists.
3. **Constraint Validation Module:**
 - Function to check:
 - **Row Constraint:** Ensures the number is not repeated in the same row.
 - **Column Constraint:** Ensures the number is not repeated in the same column.
 - **Subgrid Constraint:** Ensures the number is not repeated in the corresponding 3×3 subgrid.
4. **Backtracking Module:**
 - Recursive function that:
 - Finds the next empty cell.
 - Attempts valid placements.
 - Backtracks when stuck.
5. **Output Layer:**
 - **Completed Sudoku Grid:** The fully filled grid satisfying all constraints.
 - **Edge Cases:**
 - If the puzzle is unsolvable, return an appropriate message.

Key Points in the Architecture

- **Modular Design:** Separate functions for input handling, constraint validation, and recursive backtracking.
- **Scalability:** The algorithm can be adapted to larger grids (e.g., 16×16) with minimal changes.
- **Robustness:** Includes handling for edge cases like unsolvable grids or ambiguous puzzles with multiple solutions.

This combination of systematic exploration and modular architecture ensures an efficient and correct solution to the Sudoku problem.

5. Results and Discussion

The implementation of the backtracking algorithm has demonstrated its effectiveness in solving a wide variety of Sudoku puzzles. It successfully handles puzzles with varying levels of difficulty, including edge cases such as those with minimal clues or multiple potential solutions. The algorithm adheres to the rules of Sudoku and ensures that every solution satisfies the constraints of unique digits in rows, columns, and subgrids.

Performance Analysis

The performance of the backtracking algorithm depends significantly on the complexity of the puzzle, primarily determined by:

1. Number of Empty Cells: Puzzles with a larger number of empty cells require more iterations, increasing computation time.
2. Initial Clue Distribution: The placement of given numbers can affect the algorithm's ability to find a solution efficiently.

In general, puzzles with more structured or evenly distributed clues are solved faster than those with clusters of empty cells or sparse initial clues.

Edge Case Handling

- Minimal Clue Puzzles: The algorithm performs well even with puzzles having only 17 initial clues (the minimum number of clues required for a valid Sudoku).
- Unsolvable Puzzles: The algorithm accurately identifies unsolvable puzzles by exhausting all possible placements without finding a valid solution.
- Multiple Solutions: In cases with multiple valid solutions, the algorithm identifies one solution but could be modified to explore all possibilities if needed.

Limitations and Future Scope

While the backtracking algorithm is robust, its recursive nature can lead to performance bottlenecks for highly complex puzzles. To address this, future enhancements could include:

1. Constraint Propagation: Reducing the domain of possible values for each variable by considering constraints upfront, thereby minimizing unnecessary iterations.
2. Heuristic Techniques: Implementing heuristics such as the Most Constrained Variable (choosing the cell with the fewest valid options first) or the Least Constraining Value (choosing the number that restricts future placements the least).
3. Parallelism: Leveraging parallel computing to explore multiple branches of the solution space simultaneously.

6. Conclusion

This paper outlined the design and successful implementation of a Sudoku solver leveraging the backtracking algorithm. By systematically filling the grid with numbers and validating each placement against the constraints of Sudoku, the algorithm effectively navigates the problem space to find solutions. The recursive nature of backtracking enables the solver to backtrack and reevaluate placements, ensuring correctness and adherence to the puzzle's rules.

The implementation demonstrates the flexibility of the algorithm in handling puzzles of varying complexities, including edge cases like minimal clue puzzles and identifying unsolvable grids. While the current solution is robust, its performance could be further improved for highly complex puzzles through advanced techniques such as constraint propagation, heuristic-driven search strategies, or parallel computing.

In conclusion, the backtracking algorithm provides a strong foundation for solving Sudoku puzzles and serves as a stepping stone for more sophisticated approaches, ensuring both precision and scalability in addressing constraint satisfaction problems.

References

- [1] **"An Implementation of Backtracking Algorithm for Solving a Sudoku Puzzle Based on Android"**
Authors: Evarista Nwulu and Desmond B. A. L. A. Bisandu
Journal: International Journal of Computer Applications, 2019.
Abstract: This paper presents a hybrid Sudoku solver combining backtracking with traditional pencil-and-paper techniques to enhance efficiency.
- [2] **"Sudoku Solver: A Comparative Study of Different Algorithms and Image Processing Techniques"**
Authors: [Authors not specified]
Journal: [Journal not specified], 2023.
Abstract: This study compares various algorithms for solving Sudoku puzzles, including backtracking and genetic algorithms, and discusses their efficiency.
- [3] **"Image-Based Sudoku Solver Using Applied Recursive Backtracking"**
Authors: [Authors not specified]
Conference: IEEE Conference, 2024.
Abstract: This paper explores the application of recursive backtracking algorithms in solving Sudoku puzzles extracted from images, highlighting real-time projection of solutions.
- [4] **"Sudoku Solver Using Minigrid-Based Backtracking"**
Authors: [Authors not specified]
Conference: IEEE Conference, 2013.
Abstract: This research introduces a Sudoku-solving algorithm that utilizes minigrid-based backtracking to enhance the efficiency of finding solutions.
- [5] **"Randomised Analysis of Backtracking-Based Search Algorithms in Elucidating Sudoku Puzzles Using a Dual Serial/Parallel Approach"**
Authors: Pramika Garg and [Co-authors not specified]
Conference: [Conference not specified], 2022.
Abstract: This study evaluates the performance of backtracking-based algorithms in solving Sudoku puzzles, comparing serial and parallel implementations.
- [6] **"A Pencil-and-Paper Algorithm for Solving Sudoku Puzzles"**
Authors: [Authors not specified]
Journal: Notices of the American Mathematical Society, 2009.
Abstract: This article discusses a manual algorithm for solving Sudoku puzzles, providing insights into the logical strategies involved.