University of Reading

Department of Computer Science

# Finding Sudoku solutions using Heuristic-Driven Backtracking and SAT solvers.

Mahmoud Diallo

*Supervisor:* Alan Guedes

A report submitted in partial fulfilment of the requirements of
the University of Reading for the degree of
Bachelor of Science in *Computer Science*

May 14, 2025

## Declaration

I, Mahmoud Diallo, of the Department of Computer Science, University of Reading, confirm that this is my own work and figures, tables, equations, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced. I understand that if failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and will be penalised accordingly.

I give consent to a copy of my report being shared with future students as an exemplar.

I give consent for my work to be made available more widely to members of UoR and public with interest in teaching, learning and research.

Mahmoud Diallo
May 14, 2025

# Abstract

Sudoku puzzles have become a classic benchmark for constraint satisfaction problems (CSPs) in computer science. This project compares three different approaches to solving Sudoku puzzles: backtracking with the Minimum Remaining Values (MRV) heuristic, Boolean satisfiability (SAT) solvers implemented with the PySAT library, and constraint programming via MiniZinc. The goal was to determine which methods perform best for puzzles of varying sizes and difficulty levels, from standard 9×9 grids to larger 16×16 and 25×25 puzzles.

My implementation included a heuristic-driven backtracking algorithm, a SAT-based solver using PySAT's Glucose3, and a MiniZinc model with redundant constraints. All three approaches were tested systematically across multiple puzzle sizes (9×9, 16×16, and 25×25) and difficulty levels (easy, medium, hard, and extreme), with performance metrics carefully recorded including solving time and success rates.

The results revealed interesting performance patterns. For small puzzles, backtracking was consistently the fastest method due to its lower overhead. For standard 9×9 puzzles, a more nuanced pattern emerged: backtracking remained fastest for easy puzzles but struggled significantly with hard puzzles, while the MRV heuristic showed superior performance on difficult puzzles. For 16×16 and 25×25 grids, the computational complexity increased dramatically, with backtracking becoming impractical for more difficult ones, where the MiniZinc and SAT approaches showed clear advantages.

Notably, the SAT solver demonstrated consistent performance across difficulty levels for each grid size, though with higher initial overhead. It scaled more predictably with puzzle complexity, becoming particularly valuable for larger, harder puzzles where backtracking methods faced exponential growth in solution time. The MiniZinc solver leveraged powerful constraint propagation and search annotations to efficiently solve even challenging puzzles.

This project demonstrates that the optimal solving method depends on both puzzle size and difficulty level. For small or simple puzzles, backtracking offers efficiency through simplicity. For complex puzzles, MiniZinc and SAT solvers provide critical optimization. For very large or extremely difficult puzzles, constraint programming approaches offer reliability despite higher overhead. These findings highlight the essential trade-offs between algorithm complexity and problem-solving efficiency that extend beyond Sudoku to many real-world constraint satisfaction problems.

**Keywords:** Sudoku solvers, constraint satisfaction problems, backtracking, heuristic methods, SAT solvers, MiniZinc, algorithm comparison, computational complexity

## Acknowledgements

An acknowledgements section is optional. You may like to acknowledge the support and help of your supervisor(s), friends, or any other person(s), department(s), institute(s), etc. If you have been provided specific facility from department/school acknowledged so.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| SMPCS | School of Mathematical, Physical and Computational Sciences |

# Chapter 1

# Introduction

This chapter introduces my research project on Sudoku solving algorithms, outlining the background of the problem, research questions, project objectives, and the methodological approach. The chapter concludes with a summary of the project's main contributions and an overview of the report structure.

## 1.1 Background

My interest in Sudoku began when I realized these popular puzzles represent perfect examples of constraint satisfaction problems (CSPs) in computer science. While solving them by hand requires patience and logical reasoning, I became curious about how computers tackle these puzzles and what happens when they face increasingly difficult challenges.

Sudoku puzzles follow simple rules - fill a grid with numbers so that each row, column, and region contains each number exactly once. Despite these straightforward constraints, finding solutions becomes exponentially more difficult as the grid size increases. This makes Sudoku an ideal test case for comparing different algorithmic approaches.

In this project, I've implemented three fundamental approaches to solving Sudoku: backtracking enhanced with the Minimum Remaining Values (MRV) heuristic, SAT (Boolean satisfiability) solvers using the PySAT library, and constraint programming via MiniZinc. The heuristic-enhanced backtracking makes intelligent choices about which cells to fill first, minimizing the search space. SAT solvers convert the entire problem into Boolean logic formulas and leverage powerful solver engines like Glucose3. MiniZinc provides a high-level constraint modeling language that combines powerful propagation techniques with sophisticated search strategies.

What particularly interested me was implementing these approaches to handle puzzles of varying sizes - from standard 9×9 grids to larger 16×16 and 25×25 puzzles. These increasingly larger puzzles create expanding search spaces that test the limits of different solving strategies. Additionally, I developed a puzzle generator capable of creating puzzles with four distinct difficulty levels (easy, medium, hard, and extreme) while ensuring solution uniqueness.

To make the project accessible and interactive, I created a web interface using Streamlit that allows users to generate puzzles, choose solving methods, and compare algorithm performance. The interface provides visual feedback by highlighting cells that were filled by the solver in green, making it easy to see the solution process. Furthermore, I implemented comprehensive automated testing and visualization capabilities that enable systematic performance analysis across multiple puzzle configurations, generating publication-quality charts and detailed statistics.

The techniques studied in this project have applications far beyond Sudoku. Similar approaches are used in real-world scheduling problems, resource allocation, circuit verification, and many other areas where systems must satisfy multiple constraints simultaneously. By understanding how these algorithms scale with increasing problem size, we gain insights applicable to many practical computing challenges.

## 1.2 Problem statement

My research addresses several key questions about Sudoku solving algorithms:

1. How do heuristic-enhanced backtracking, SAT solvers, and MiniZinc constraint programming compare when solving Sudoku puzzles of different sizes and difficulty levels?
2. What happens to performance when we scale up from standard 9×9 puzzles to larger 16×16 and 25×25 grids?
3. Under what conditions does one approach outperform the others?
4. What are the practical limitations of each approach as puzzle complexity increases?
5. How can we effectively generate puzzles of controlled difficulty while ensuring solution uniqueness?
6. How can we systematically analyze and visualize algorithm performance across multiple test configurations?

These questions matter because they help us understand the fundamental trade-offs between different algorithmic strategies. My heuristic-enhanced backtracking implementation adds informed decision-making to the search process by selecting the most constrained cells first. The SAT solver implementation transforms Sudoku constraints into CNF clauses and employs the sophisticated Glucose3 solver. The MiniZinc implementation leverages efficient propagation techniques, redundant sum constraints, and domain-weighted degree search annotations.

From a practical perspective, I wanted to identify the breaking points of each approach. As Sudoku grids grow larger, the search space expands exponentially - a 9×9 grid has $9^{22}$ possible combinations, a 16×16 grid has an astronomical number of potential arrangements, and a 25×25 grid is even more complex. Finding efficient ways to navigate these vast search spaces challenges all three approaches in different ways. Additionally, I sought to develop tools that make performance analysis accessible and reproducible, enabling clear visualization of algorithmic behavior under different conditions.

## 1.3 Aims and objectives

**Aims:** My project aims to:

1. Develop efficient Sudoku solvers using heuristic-driven backtracking, SAT-based approaches, and constraint programming with MiniZinc
2. Create a comprehensive puzzle generation system capable of producing puzzles with controlled difficulty levels
3. Build an interactive web interface for puzzle solving and algorithm comparison

4. Develop automated testing and visualization tools for systematic performance analysis
5. Determine the relative strengths and weaknesses of these approaches across different puzzle sizes and difficulty levels
6. Generate publication-quality visualizations that clearly communicate performance characteristics
7. Identify when each algorithm performs best based on puzzle characteristics
8. Contribute insights about algorithm scalability that might apply to similar computational problems

**Objectives:** To achieve these aims, I established the following specific objectives:

1. **Research and Analysis**
   - Review existing literature on Sudoku solving algorithms
   - Analyze the theoretical approaches to solving constraint satisfaction problems
   - Identify appropriate metrics for comparing solver performance
2. **Solver Implementation**
   - Develop a backtracking algorithm with the Minimum Remaining Values (MRV) heuristic
   - Implement candidate calculation functions for efficient constraint checking
   - Create a SAT-based solver using the PySAT library with Glucose3 backend
   - Develop variable encoding schemes for Boolean representation
   - Build a MiniZinc model with all-different constraints and redundant sum constraints
   - Implement efficient search annotations using domain-weighted degree
   - Ensure all solvers can handle $9\times9$, $16\times16$, and $25\times25$ puzzles
3. **Puzzle Generation**
   - Create a complete board generator using pattern-based approaches
   - Implement random transformation functions (digit permutation, row/column swapping)
   - Develop a cell removal algorithm that maintains solution uniqueness
   - Support four difficulty levels with controlled cell removal percentages
4. **User Interface Development**
   - Build a Streamlit-based web interface for user interaction
   - Implement visual board display with solution highlighting
   - Create controls for puzzle generation and solver selection
   - Add performance comparison and visualization features
   - Include seed control for reproducible puzzle generation
   - Develop automated testing interface for batch analysis

5. **Performance Analysis and Visualization**
   - Implement automated testing across multiple puzzle configurations
   - Create comprehensive performance visualization tools using matplotlib and seaborn
   - Develop scalability analysis charts showing algorithm behavior across board sizes
   - Generate performance distribution plots to show consistency and variability
   - Create heatmaps for at-a-glance performance comparison
   - Export results in multiple formats (PNG, CSV, summary reports)
6. **Testing and Evaluation**
   - Implement timing mechanisms for accurate performance measurement
   - Create comparison tools that test all methods on identical puzzles
   - Handle timeout cases for very difficult puzzles
   - Develop batch testing capabilities for statistical reliability
   - Generate automated summary reports with key metrics
7. **Comparative Analysis**
   - Compare performance characteristics across different puzzle sizes and difficulty levels
   - Identify how each algorithm's performance scales as problem size increases
   - Determine the conditions under which each approach works best
   - Document strengths and limitations based on empirical results
   - Provide actionable recommendations for algorithm selection

## 1.4 Solution approach

I approached this project by implementing all three solving methods within a unified framework, creating a comprehensive testing environment, and analyzing the results through an interactive interface with automated analysis capabilities.

### 1.4.1 Heuristic-Enhanced Backtracking Solver

I implemented a backtracking algorithm with the Minimum Remaining Values (MRV) heuristic, which:

- Represents the Sudoku board as a 2D array with 0 representing empty cells
- Calculates valid candidates for each cell by checking row, column, and block constraints
- Identifies the empty cell with the fewest valid options using the `find_empty_with_MRV` function
- Prioritizes filling these constrained cells first to reduce the search space
- Implements efficient backtracking when dead ends are encountered
- Makes intelligent choices about which paths to explore

The implementation includes optimization such as early termination when cells have only one valid candidate, making it particularly effective for easier puzzles.

### 1.4.2 SAT-Based Solver

For the SAT approach, I used the PySAT library with Glucose3 to:

- Encode Sudoku constraints as a Boolean satisfiability problem using a `varnum` function that maps cell-digit combinations to unique Boolean variables
- Implement the `exactly_one` function to ensure each constraint is satisfied precisely
- Create CNF clauses for cell uniqueness, row constraints, column constraints, and block constraints
- Encode pre-filled cells as unit clauses
- Use the Glucose3 solver to find satisfying assignments
- Convert the Boolean model back to a completed Sudoku board

This approach benefits from decades of SAT solver research and optimization, making it particularly robust for difficult puzzles.

### 1.4.3 MiniZinc Constraint Programming Solver

For the constraint programming approach, I created a MiniZinc model that:

- Defines the Sudoku grid using high-level constraint notation
- Implements custom all-different constraints for rows, columns, and blocks
- Adds redundant sum constraints to speed up constraint propagation
- Uses domain-weighted degree (dom_w_deg) search annotations for efficient variable selection
- Employs the indomain_min value selection strategy
- Interfaces with the Chuffed solver backend for optimal performance

The MiniZinc implementation leverages sophisticated constraint propagation techniques that can dramatically reduce the search space for complex puzzles.

### 1.4.4 Puzzle Generation System

I developed a comprehensive puzzle generation system that:

- Creates complete valid boards using a pattern-based approach
- Applies random transformations including digit permutation, row/column swapping within blocks, and block swapping
- Implements a careful cell removal process that maintains solution uniqueness
- Supports four difficulty levels with specific removal percentages:
  - Easy: 40% cell removal
  - Medium: 50% cell removal
  - Hard: 60% cell removal
  - Extreme: 70% cell removal
- Uses a solution counting algorithm to verify uniqueness during generation

### 1.4.5 Web Interface and Testing Framework

To ensure fair comparison and provide user interaction, I created:

- A Streamlit-based web interface with responsive design
- Visual board display using HTML/CSS with highlighting for solver-filled cells
- Sidebar controls for board size, difficulty, and solver selection
- A comparison tool that tests all three methods on identical puzzles
- Performance visualization using pandas DataFrames and bar charts
- Seed control for reproducible puzzle generation
- Timeout handling to manage cases where solvers take excessive time

The interface makes it easy to generate puzzles, solve them using different methods, and compare performance across algorithms.

### 1.4.6 Automated Analysis and Visualization System

To enable systematic performance analysis, I developed:

- An automated testing framework that runs multiple puzzles across all configurations
- Batch processing capabilities for statistical reliability
- Comprehensive visualization generation using matplotlib and seaborn:
  - Performance comparison charts by board size
  - Scalability analysis plots showing algorithm trends
  - Distribution plots revealing performance consistency
  - Heatmaps for comprehensive overview
- Export functionality for all results and visualizations
- Automated report generation with summary statistics
- Progress tracking for long-running analyses
- Session state management for persistent data collection

This system transforms raw performance data into publication-quality visualizations suitable for academic presentation.

## 1.5 Summary of contributions and achievements

Through this project, I've made several contributions to understanding Sudoku solving algorithms:

1. **Complete implementation**: I successfully implemented three distinct Sudoku solving approaches with full support for multiple board sizes and difficulty levels.

2. **Interactive comparison tool**: The Streamlit interface provides an accessible way to compare algorithm performance and visualize results.
3. **Automated analysis system**: The batch testing framework enables systematic evaluation across multiple configurations with statistical reliability.
4. **Comprehensive visualization tools**: The system generates publication-quality charts and graphs that clearly communicate performance characteristics.
5. **Robust puzzle generation**: The generator creates valid puzzles with controlled difficulty while ensuring solution uniqueness.
6. **Performance analysis across scales**: My testing revealed clear patterns in how each algorithm scales with puzzle size and difficulty.
7. **Practical algorithm selection guidelines**: The results provide clear guidance on which algorithm to use based on puzzle characteristics.

Initial testing shows that backtracking with MRV performs excellently on standard 9×9 puzzles but struggles with larger sizes. SAT solvers maintain consistent performance across difficulty levels, while MiniZinc demonstrates variable performance with some unexpected scaling patterns. The automated analysis tools enable deeper investigation of these behaviors through systematic testing and visualization. These insights extend beyond Sudoku to other constraint satisfaction problems.

## 1.6 Organization of the report

The remainder of this report is organized as follows:

Chapter 2 reviews relevant literature on Sudoku solving techniques, covering backtracking algorithms, heuristic methods, SAT solving approaches, constraint programming, and previous comparative studies.

Chapter 3 details my methodology, including complete algorithm implementations, the puzzle generation system, the web interface design, the automated testing framework, and the visualization tools.

Chapter 4 presents the results of my comparative analysis, including detailed performance measurements across different puzzle sizes and difficulty levels, with comprehensive visualizations showing the relative strengths of each method.

Chapter 5 discusses these findings, interpreting the observed performance differences and examining their broader implications for constraint satisfaction problem solving.

Chapter 6 concludes the report, summarizing key findings and suggesting directions for future research.

Chapter 7 offers a personal reflection on the project, discussing challenges faced, lessons learned, and how this work has influenced my understanding of algorithm design and analysis.

The report includes references to all cited works and appendices containing code snippets, additional technical details, comprehensive performance data, and instructions for using the automated analysis system.

# Chapter 2

# Literature Review

This chapter reviews the relevant literature on Sudoku solving techniques, providing context for my project and identifying gaps in existing research. The review covers backtracking algorithms, heuristic improvements, SAT-based approaches, and constraint programming methods, with particular attention to their performance across different puzzle sizes.

## 2.1 State-of-the-Art in Sudoku Solving Techniques

Sudoku solving has been extensively studied within computer science, particularly as a constraint satisfaction problem (CSP). This section reviews the primary algorithmic approaches and their theoretical foundations.

### 2.1.1 Basic Backtracking Approaches

Backtracking is one of the most fundamental algorithms for solving Sudoku puzzles. Russell and Norvig (2020) describe backtracking as a systematic depth-first search through the solution space. The classic backtracking algorithm works by systematically filling cells and backtracking when constraints are violated. While effective for simple puzzles, traditional backtracking becomes computationally intensive for larger and more complex puzzles due to its time complexity of $O(9^n)$ where $n$ is the number of empty cells.

Knuth (2000) analyzed backtracking algorithms for exact cover problems, which include Sudoku as a special case. His Dancing Links algorithm represents one of the most efficient approaches to solving exact cover problems, providing insights into how backtracking can be optimized through careful data structure choices.

The basic backtracking approach follows a simple recursive pattern: select an empty cell, try a possible value, check if it violates any constraints, and either continue to the next cell if valid or try a different value if invalid. This depth-first search exploration continues until either a solution is found or all possibilities are exhausted.

### 2.1.2 Heuristic-Driven Backtracking

Heuristic-driven backtracking represents a significant advancement over pure backtracking. According to Russell and Norvig (2020), intelligent variable and value ordering heuristics can dramatically improve solving efficiency. The Minimum Remaining Values (MRV) heuristic, also known as the "most constrained variable" heuristic, selects the variable with the fewest remaining legal values.

Norvig (2006) demonstrated the effectiveness of constraint propagation combined with backtracking for Sudoku solving. His implementation using Python showed how the MRV heuristic, combined with constraint propagation, could solve even the hardest Sudoku puzzles in milliseconds. Norvig's approach influences many modern Sudoku solvers and demonstrates the practical benefits of intelligent heuristics.

The MRV heuristic works particularly well for Sudoku because it naturally focuses on the most constrained cells first—those with the fewest possible valid values. By filling these cells early in the search process, the heuristic helps prune large portions of the search space.

### 2.1.3 SAT-Based Approaches

Boolean Satisfiability (SAT) solvers represent another powerful approach to solving Sudoku puzzles. Lynce and Marques-Silva (2006) demonstrated that modern SAT solvers could efficiently solve even difficult Sudoku puzzles by encoding Sudoku rules as conjunctive normal form (CNF) formulas.

Weber (2005) provided one of the first comprehensive descriptions of encoding Sudoku as a SAT problem. They showed that Sudoku puzzles can be encoded using three types of constraints: (1) each cell must contain exactly one number, (2) each number must appear exactly once in each row, column, and block, and (3) filled cells must retain their given values.

Heule et al. (2013) explored how SAT solvers can be used to answer theoretical questions about Sudoku, such as the minimum number of clues required for a unique solution. Their work demonstrated that modern SAT solvers could handle the massive search spaces involved in such theoretical investigations.

The Ignatiev et al. (2018) PySAT library provides Python bindings to state-of-the-art SAT solvers including Glucose, MiniSat, and CryptoMiniSat. This library makes SAT solving accessible for Sudoku applications and has become a standard tool in the Python ecosystem for SAT-based problem solving.

### 2.1.4 Constraint Programming Approaches

Constraint programming (CP) represents another powerful paradigm for solving Sudoku puzzles. Pesant (2004) explored constraint programming techniques for Sudoku, showing how global constraints like alldifferent can efficiently propagate constraints and prune the search space.

Nethercote et al. (2005) developed the initial implementation of MiniZinc, a high-level constraint modeling language that allows for succinct expression of constraint problems. MiniZinc has become a standard tool in the constraint programming community and provides efficient backends for solving Sudoku and other constraint satisfaction problems.

Simonis (2005) provided an extensive analysis of Sudoku as a constraint programming problem. His work demonstrated that CP approaches can be highly effective for Sudoku, particularly when domain-specific propagation techniques are employed. The constraint models expressed Sudoku rules as alldifferent constraints on rows, columns, and blocks.

More recently, Stuckey and Tack (2014) explored advanced search strategies in MiniZinc, including activity-based search and weighted-degree heuristics. These techniques have proven particularly effective for Sudoku solving, often outperforming traditional search strategies.

### 2.1.5 Theoretical Complexity Analysis

Yato and Seta (2003) proved that Sudoku is NP-complete, establishing its theoretical complexity. This result has important implications for algorithm design, as it suggests that no polynomial-time algorithm is likely to exist for solving all Sudoku puzzles.

Felgenhauer and Jarvis (2005) computed the total number of valid 9×9 Sudoku grids (approximately $6.67 \times 10^{21}$), providing insight into the massive search space that solving algorithms must navigate. This mathematical analysis helps explain why efficient heuristics and constraint propagation are crucial for practical Sudoku solving.

Understanding the theoretical time complexity of different approaches provides important insights into their scalability. For a Sudoku puzzle with $n^2 \times n^2$ dimensions, the theoretical complexity varies by approach:

- Basic backtracking: $O((n^2)^m)$ where $m$ is the number of empty cells
- SAT-based approaches: $O(n^6)$ variables and $O(n^4)$ clauses
- Constraint programming: Similar worst-case to backtracking but with practical improvements through propagation

### 2.1.6 Performance Comparisons Across Puzzle Sizes

Lewis (2007) compared the performance of different solving techniques across various puzzle sizes, finding that algorithm effectiveness varies significantly with puzzle dimensions. They observed that while backtracking works well for smaller puzzles, its performance degrades significantly for larger puzzles like 16×16 grids.

Bartlett and Langville (2008) conducted extensive benchmarking of Sudoku solvers, including both algorithmic approaches and human-like solving techniques. Their research revealed that the relative performance of different approaches varies based on puzzle characteristics such as size, difficulty, and clue distribution.

McGuire et al. (2014) made headlines by proving that 17 is the minimum number of clues required for a unique solution in 9×9 Sudoku. Their exhaustive computational search required analyzing approximately 5.4 billion grids and demonstrated the computational challenges involved in Sudoku research.

## 2.2 Project Context in Relation to Existing Literature

This project builds upon the existing literature by implementing and comparing three fundamental approaches: backtracking with MRV heuristic, SAT-based solving using PySAT, and constraint programming using MiniZinc. While previous studies have compared these approaches, this project provides a comprehensive evaluation across multiple dimensions:

1. Multiple puzzle sizes (9×9, 16×16, 25×25)
2. Four difficulty levels (easy, medium, hard, extreme)
3. Consistent testing framework with standardized timing
4. Interactive visualization of results

The implementation draws directly from techniques described in the literature:

- The MRV heuristic follows Russell and Norvig (2020) and Norvig (2006)
- SAT encoding uses strategies from Lynce and Marques-Silva (2006) and Weber (2005)
- MiniZinc implementation incorporates techniques from Simonis (2005) and Stuckey and Tack (2014)

## 2.3 Gaps in Existing Literature

Despite extensive research on Sudoku solving, several gaps remain:

1. **Limited systematic comparison across sizes**: While studies like Lewis (2007) examine different puzzle sizes, comprehensive comparisons using modern implementations remain limited.
2. **Insufficient difficulty gradation**: Many studies focus on either very easy or very hard puzzles without systematic difficulty progression.
3. **Lack of interactive comparison tools**: Most research presents static results without interactive exploration capabilities.
4. **Limited practical implementation guidance**: While theoretical analyses are common, practical implementation details and performance optimization strategies are less documented.

## 2.4 Visual Analysis in Algorithm Comparison

The importance of visualization in algorithm analysis has been emphasized by Bentley (1999) and Cormen et al. (2009). For constraint satisfaction problems, visual representation can reveal patterns in algorithm behavior that might be obscured in numerical data alone.

This project addresses the visualization gap by providing:

- Real-time solving visualization with highlighted cells
- Performance comparison charts across algorithms
- Interactive puzzle generation with difficulty control
- Comprehensive timing analysis with graphical output

## 2.5 Summary

This literature review has examined the primary techniques for solving Sudoku puzzles and identified key research that informs this project's design. The existing literature demonstrates that:

1. Heuristic enhancements like MRV significantly improve backtracking performance
2. SAT solvers provide robust solving capabilities with consistent performance characteristics
3. Constraint programming offers powerful modeling and propagation capabilities
4. Algorithm selection should consider puzzle size and difficulty

This project contributes to the existing body of knowledge by:

- Providing a unified implementation of three major solving approaches
- Offering systematic comparison across multiple puzzle dimensions

- Creating an interactive tool for algorithm exploration and comparison
- Documenting practical implementation details and performance characteristics

By building on established techniques while addressing identified gaps, this project provides both theoretical insights and practical tools for understanding Sudoku solving algorithms. The comprehensive approach enables researchers and practitioners to make informed decisions about algorithm selection based on specific problem characteristics.

# Chapter 3

# Methodology

This chapter details the methodological approach used to implement and evaluate different Sudoku solving techniques. Following the algorithm analysis paradigm outlined in Chapter **??**, this chapter is organized into three main sections: algorithm descriptions, implementations, and experiment design. The methodology aims to provide a comprehensive framework for comparing heuristic-driven backtracking, SAT-based solvers, and constraint programming approaches across varying Sudoku board sizes and difficulty levels.

## 3.1 Algorithm Descriptions

This section provides a detailed description of the three algorithms implemented in this project, focusing on their theoretical foundations and key components.

### 3.1.1 Heuristic-Driven Backtracking with MRV

The heuristic-driven backtracking approach enhances traditional backtracking by incorporating the Minimum Remaining Values (MRV) heuristic. This heuristic makes intelligent decisions about which cells to fill first, significantly reducing the search space.

#### Minimum Remaining Values (MRV) Heuristic

The MRV heuristic, also known as the "most constrained variable" heuristic, selects the cell with the fewest valid options remaining. This reduces the branching factor of the search tree by prioritizing cells that are more constrained.

The `GetCandidates` function efficiently determines valid values for a cell by checking row, column, and block constraints:

### 3.1.2 SAT-Based Approach

The Boolean Satisfiability (SAT) approach involves encoding the Sudoku puzzle as a Boolean formula in conjunctive normal form (CNF) and using a SAT solver to find a satisfying assignment. I use the PySAT library with the Glucose3 solver.

#### Variable Encoding

For an $n \times n$ Sudoku grid, Boolean variables are represented as $x_{i,j,d}$, which is true if and only if cell $(i,j)$ contains digit $d$. The encoding uses a mapping function:

**Algorithm 1** MRV Heuristic for Cell Selection

```
 1: function FindEmptyWithMRV($board$, $n$)
 2:     $min\_candidates \leftarrow$ None
 3:     $best\_cell \leftarrow$ None
 4:     for each cell $(i, j)$ in $board$ do
 5:         if $board[i][j] = 0$ then
 6:             $candidates \leftarrow$ GetCandidates($board, i, j, n$)
 7:             if $|candidates| = 0$ then
 8:                 return $(i, j, \emptyset)$                          ▷ Dead end
 9:             end if
10:             if $min\_candidates$ is None or $|candidates| < |min\_candidates|$ then
11:                 $min\_candidates \leftarrow candidates$
12:                 $best\_cell \leftarrow (i, j)$
13:                 if $|candidates| = 1$ then
14:                     return $(best\_cell, candidates)$         ▷ Only one option
15:                 end if
16:             end if
17:         end if
18:     end for
19:     if $best\_cell$ is not None then
20:         return $(best\_cell, min\_candidates)$
21:     else
22:         return None                                            ▷ No empty cells
23:     end if
24: end function
```

**Algorithm 2** Get Valid Candidates for a Cell

```
 1: function GetCandidates($board, row, col, n$)
 2:     $candidates \leftarrow \{1, 2, \ldots, n\}$
 3:     Remove values present in $board[row]$ from $candidates$
 4:     Remove values present in column $col$ from $candidates$
 5:     $block\_size \leftarrow \lfloor\sqrt{n}\rfloor$
 6:     $start\_row \leftarrow (row//block\_size) \times block\_size$
 7:     $start\_col \leftarrow (col//block\_size) \times block\_size$
 8:     for $i \in [start\_row, start\_row + block\_size)$ do
 9:         for $j \in [start\_col, start\_col + block\_size)$ do
10:             Remove $board[i][j]$ from $candidates$
11:         end for
12:     end for
13:     return $candidates$
14: end function
```

$$\text{varnum}(i, j, d, n) = i \times n^2 + j \times n + d + 1 \quad (3.1)$$

### Constraint Encoding

The Sudoku constraints are encoded as CNF clauses using the `exactly_one` function:

**Algorithm 3** Exactly One Constraint

```
1: function ExactlyOne($vars\_list$)
2:     $clauses \leftarrow [vars\_list]$                ▷ At least one must be true
3:     for $i \in [0, |vars\_list|)$ do
4:         for $j \in [i + 1, |vars\_list|)$ do
5:             Add $[-vars\_list[i], -vars\_list[j]]$ to $clauses$
6:         end for
7:     end for
8:     return $clauses$
9: end function
```

The complete encoding includes four types of constraints:

1. **Cell constraints**: Each cell must contain exactly one digit
2. **Row constraints**: Each digit must appear exactly once in each row
3. **Column constraints**: Each digit must appear exactly once in each column
4. **Block constraints**: Each digit must appear exactly once in each block
5. **Clue constraints**: Pre-filled cells must retain their values

### 3.1.3 Constraint Programming with MiniZinc

The constraint programming approach uses MiniZinc, a high-level constraint modeling language. The model employs several optimization techniques:

#### Model Components

The MiniZinc model includes:

- **All-different constraints**: Ensures uniqueness in rows, columns, and blocks
- **Redundant sum constraints**: Speeds up constraint propagation by adding that each row, column, and block must sum to $n(n + 1)/2$
- **Search annotations**: Uses `dom_w_deg` (domain over weighted degree) variable ordering with `indomain_min` value selection

The key modeling decision is the use of redundant constraints:

$$\sum_{j=1}^{n} x_{i,j} = \frac{n(n+1)}{2} \quad \forall i \in \{1, \ldots, n\} \quad (3.2)$$

These constraints, while logically redundant, help the solver detect inconsistencies faster during propagation.

## 3.2 Implementations

This section describes the implementation details of all three solvers using the actual code from the project.

### 3.2.1 Heuristic Backtracking Implementation

The heuristic backtracking solver is implemented in Python with several optimizations:

```python
def backtracking_solve(board, n):
    """
    Attempt to solve the Sudoku puzzle using heuristic-driven backtracking
    .
    Returns True if a solution is found, else False.
    """
    pos = find_empty_with_MRV(board, n)
    # If no empty cell is found, the puzzle is solved
    if pos is None:
        return True
    row, col, candidates = pos
    # Dead end when no candidate is available
    if not candidates:
        return False
    # Try each candidate value for the cell
    for candidate in candidates:
        board[row][col] = candidate
        if backtracking_solve(board, n):
            return True
        board[row][col] = 0  # Reset the cell on backtracking
    return False
```

Listing 3.1: Core implementation of heuristic backtracking

The implementation includes several efficiency features:

- Early termination when cells have only one valid candidate
- Efficient candidate calculation using set operations
- Deep copy prevention by modifying the board in-place

### 3.2.2 SAT Solver Implementation

The SAT-based solver uses PySAT with the Glucose3 solver:

```python
def solve_with_sat(board):
    """
    Solves the Sudoku puzzle using a SAT solver (Glucose3 from PySAT).
    """
    n = len(board)
    board_copy = copy.deepcopy(board)
    # Encode the board as CNF clauses
    clauses = encode_sudoku(board_copy, n)
    solver = Glucose3()

    # Add all clauses to the solver
    for clause in clauses:
        solver.add_clause(clause)

    start_time = time.time()
    solvable = solver.solve()
    elapsed_time = time.time() - start_time

    if not solvable:
        solver.delete()
        return None, None

    # Get the satisfying model and convert back to Sudoku board
    model = solver.get_model()
    solver.delete()
    solution = [[0 for _ in range(n)] for _ in range(n)]

    for i in range(n):
        for j in range(n):
            for d in range(n):
                v = varnum(i, j, d, n)
                if v in model:
                    solution[i][j] = d + 1
                    break

    return solution, elapsed_time
```

Listing 3.2: SAT encoding and solving

### 3.2.3 MiniZinc Implementation

The MiniZinc solver is implemented with proper UTF-8 encoding and temporary file handling:

```python
def solve_with_minizinc(board, minizinc_cmd="minizinc"):
    n = len(board)

    with tempfile.TemporaryDirectory() as tmpdir:
        model_path = os.path.join(tmpdir, "sudoku.mzn")
        data_path = os.path.join(tmpdir, "sudoku.dzn")

        # Write model & data in UTF-8
        with open(model_path, "w", encoding="utf-8") as f:
            f.write(minizinc_model)

        flat = [str(cell) for row in board for cell in row]
        dzn_lines = [
            f"n = {n};",
            f"clue = array2d(1..{n},1..{n},[{','.join(flat)}]);"
        ]
        with open(data_path, "w", encoding="utf-8") as f:
            f.write("\n".join(dzn_lines))

        # Invoke MiniZinc with Chuffed solver
        cmd = [
            minizinc_cmd,
            "--solver", "Chuffed",
            "--time-limit", "60000",
            model_path, data_path
        ]
        start = time.time()
        proc = subprocess.run(cmd, capture_output=True, text=True)
        elapsed = time.time() - start

        # Parse the solution
        if proc.returncode != 0 or "UNSATISFIABLE" in proc.stdout:
            return None, None

        # Extract solution from output
        m = re.search(r"\[\|([\s\S]*?)\|\];", proc.stdout)
        if not m:
            return None, None

        body = m.group(1).replace("|", "").replace("\n", ",")
        tokens = [tok.strip() for tok in body.split(",") if tok.strip()]
        nums = list(map(int, tokens))

        solution = [nums[i*n:(i+1)*n] for i in range(n)]
        return solution, elapsed
```

Listing 3.3: MiniZinc solver implementation

## 3.3 Puzzle Generation

The puzzle generation system creates valid Sudoku puzzles with controlled difficulty:

### 3.3.1 Complete Board Generation

The system first generates a complete valid board using a pattern-based approach with random transformations:

```
def generate_complete_board(n):
    """
    Generate a valid completed Sudoku board of size n  n.
    Uses a standard pattern and then applies random transformations.
    """
    base = int(math.sqrt(n))
    board = [
        [((r % base) * base + r // base + c) % n + 1
         for c in range(n)]
        for r in range(n)
    ]

    # Apply random transformations
    board = apply_random_transformations(board, n)
    return board
```

Listing 3.4: Complete board generation

### 3.3.2 Difficulty-Based Cell Removal

The difficulty is controlled by the percentage of cells removed:

```
def generate_puzzle(n, difficulty):
    """
    Generate a Sudoku puzzle by removing cells from a complete board
    while ensuring uniqueness.
    """
    removal_percentages = {
        "easy":    0.40,  # Remove 40% of cells
        "medium":  0.50,  # Remove 50% of cells
        "hard":    0.60,  # Remove 60% of cells
        "extreme": 0.70,  # Remove 70% of cells
    }
```

```python

    board = generate_complete_board(n)
    cells = [(r, c) for r in range(n) for c in range(n)]
    random.shuffle(cells)

    cells_to_remove = int(n * n * removal_percentages[difficulty])
    removed = 0

    for r, c in cells:
        if removed >= cells_to_remove:
            break

        backup = board[r][c]
        board[r][c] = 0

        # Ensure uniqueness for smaller puzzles
        if n <= 9 and count_solutions(board, n) != 1:
            board[r][c] = backup
        else:
            removed += 1

    return board
```

Listing 3.5: Puzzle generation with difficulty control

## 3.4 User Interface Implementation

The project includes a comprehensive Streamlit-based web interface that provides:

### 3.4.1 Interactive Features

- **Board size selection**: 9×9, 16×16, or 25×25
- **Difficulty selection**: Easy, medium, hard, or extreme
- **Solver selection**: Backtracking, PySAT, or MiniZinc
- **Puzzle generation**: With seed control for reproducibility
- **Visual feedback**: Solution cells highlighted in green
- **Performance comparison**: Side-by-side solver benchmarking
- **Automated testing**: Batch analysis across multiple configurations
- **Performance visualization**: Interactive charts and graphs

### 3.4.2 Visualization Implementation

The board visualization uses responsive HTML/CSS:

```python
def display_board(board, solution=None):
    n = len(board)
    block = int(math.sqrt(n))

    container_style = (
        "width:80vw; max-width:600px; aspect-ratio:1/1; "
        "margin:auto; overflow:hidden;"
    )
    html = f'<div style="{container_style}">...</div>'

    # Color coding for solution cells
    if solution and board[i][j] == 0 and solution[i][j] != 0:
        txt_color = "#00FF00"  # Green for solved cells
```

Listing 3.6: Board display implementation

## 3.5 Automated Testing and Analysis Framework

To enable systematic performance evaluation, I developed a comprehensive automated testing framework:

### 3.5.1 Automated Testing Implementation

The automated testing system runs multiple puzzles across all algorithm configurations:

```python
def run_automated_analysis(sizes, difficulties, num_puzzles_per_config,
                           output_dir="sudoku_analysis_results"):
    """
    Run automated analysis across multiple configurations
    and save results
    """
    # Create output directory with timestamp
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    run_dir = os.path.join(output_dir, f"run_{timestamp}")
    os.makedirs(run_dir)

    # Initialize results storage
    all_results = []
    total_tests = len(sizes) * len(difficulties) * num_puzzles_per_config
    current_test = 0

    # Create progress tracking
    progress_bar = st.progress(0)
    status_text = st.empty()

    # Run tests
    for size in sizes:
        for difficulty in difficulties:
            for puzzle_num in range(num_puzzles_per_config):
                current_test += 1
                progress = current_test / total_tests
                progress_bar.progress(progress)

                # Generate and test puzzle
                board = generate_puzzle(size, difficulty)
                results = {
                    'size': f'{size}  {size}',
                    'difficulty': difficulty.capitalize(),
                    'puzzle_num': puzzle_num + 1
                }

                # Test each algorithm
                if size <= 16:
                    sol, t = solve_with_backtracking(board)
                    results['Backtracking'] = t if sol else None

                sol, t = solve_with_sat(board)
                results['PySAT'] = t if sol else None

                if shutil.which("minizinc"):
                    sol, t = solve_with_minizinc(board)
                    results['MiniZinc'] = t if sol else None

                all_results.append(results)

    # Save results and generate visualizations
    df_results = pd.DataFrame(all_results)
    df_results.to_csv(os.path.join(run_dir, "raw_results.csv"),
                      index=False)

    # Generate comprehensive visualizations
    create_and_save_all_visualizations(all_results, run_dir)
    generate_summary_report(df_results, run_dir)

    return df_results
```

Listing 3.7: Automated analysis framework

### 3.5.2 Performance Visualization System

The visualization system creates multiple types of performance charts:

```
def create_and_save_all_visualizations(performance_data, output_dir):
    """
    Create and save all visualization types
    """
    df = pd.DataFrame(performance_data)

    # 1. Performance comparison by board size
    fig1, axes = plt.subplots(1, 3, figsize=(15, 5))
    fig1.suptitle('Performance Comparison by Board Size',
                  fontsize=16, fontweight='bold')

    for idx, size in enumerate(['9 9 ', '16 16 ', '25 25 ']):
        ax = axes[idx]
        size_data = df[df['size'] == size]

        if not size_data.empty:
            pivot_data = size_data.pivot_table(
                index='difficulty',
                columns='algorithm',
                values='time',
                aggfunc='mean'
            )
            pivot_data.plot(kind='bar', ax=ax, width=0.8)
            ax.set_title(f'{size} Puzzles')
            ax.set_ylabel('Average Time (seconds)')
            ax.set_yscale('log')
            ax.legend(title='Algorithm')
            ax.grid(True, alpha=0.3)

    plt.tight_layout()
    fig1.savefig(os.path.join(output_dir, 'performance_by_size.png'),
                 dpi=300, bbox_inches='tight')

    # Generate additional visualizations:
```

```
    # - Scalability analysis
    # - Performance distribution plots
    # - Comprehensive heatmaps
```

Listing 3.8: Visualization generation

### 3.5.3 Interactive Performance Analysis

The interface provides interactive performance analysis through tabbed visualizations:

```python
def create_performance_visualization(performance_data):
    """
    Create interactive performance visualizations
    """
    if not performance_data:
        st.warning("No performance data available")
        return

    # Create tabs for different visualizations
    tab1, tab2, tab3, tab4 = st.tabs([
        "Performance Comparison",
        "Scalability",
        "Distribution",
        "Heatmap"
    ])

    with tab1:
        # Performance comparison charts
        create_comparison_charts(performance_data)

    with tab2:
        # Scalability analysis plots
        create_scalability_plots(performance_data)

    with tab3:
        # Distribution analysis
        create_distribution_plots(performance_data)

    with tab4:
        # Comprehensive heatmap
        create_performance_heatmap(performance_data)
```

Listing 3.9: Interactive performance visualization

## 3.6 Experiment Design

The experimental methodology evaluates solver performance across multiple dimensions:

### 3.6.1 Test Dataset

The test dataset includes:

- Generated puzzles of sizes 9×9, 16×16, and 25×25
- Four difficulty levels for each size
- Multiple random seeds for statistical validity
- Batch testing capabilities for reliable averages

### 3.6.2 Performance Metrics

Key performance metrics include:

- **Solving time**: Wall-clock time in seconds
- **Success rate**: Completion within timeout limits
- **Scalability**: Performance degradation with size increase
- **Consistency**: Variance in solving times across similar puzzles
- **Statistical measures**: Mean, median, and distribution of solving times

### 3.6.3 Comparison Framework

The comparison tool implements:

```python
def compare_solvers():
    board = st.session_state.board
    times = {}

    # Test each solver
    if n <= 16:  # Backtracking only for smaller puzzles
        _, t_bt = solve_with_backtracking(board)
        times["Backtracking"] = t_bt or float("nan")

    _, t_sat = solve_with_sat(board)
    times["PySAT"] = t_sat or float("nan")

    if shutil.which("minizinc"):
        _, t_mzn = solve_with_minizinc(board)
        times["MiniZinc"] = t_mzn or float("nan")

    # Record performance data
    for algorithm, time_val in times.items():
        if not math.isnan(time_val):
            st.session_state.performance_data.append({
                'size': f'{n}  {n}',
                'difficulty': difficulty.capitalize(),
                'algorithm': algorithm,
                'time': time_val
            })

    # Display results
    df = pd.DataFrame.from_dict(times, orient="index",
                                columns=["Time (s)"])
    st.bar_chart(df.dropna())
```

Listing 3.10: Solver comparison implementation

### 3.6.4 Statistical Analysis

The system generates comprehensive statistical summaries:

```python
def generate_summary_report(df_results, output_dir):
    """
    Generate a text summary report of the analysis
    """
    report_path = os.path.join(output_dir, "summary_report.txt")

    with open(report_path, 'w') as f:
        f.write("SUDOKU SOLVER PERFORMANCE ANALYSIS REPORT\n")
        f.write("=" * 40 + "\n\n")

        # Overall statistics
        for algo in ['Backtracking', 'PySAT', 'MiniZinc']:
            times = df_results[algo].dropna()

            if not times.empty:
                f.write(f"\n{algo}:\n")
                f.write(f"  Min time: {times.min():.4f}s\n")
                f.write(f"  Max time: {times.max():.4f}s\n")
                f.write(f"  Mean time: {times.mean():.4f}s\n")
                f.write(f"  Median time: {times.median():.4f}s\n")
                f.write(f"  Success rate: {len(times)}/{len(df_results)} "
)
                f.write(f"({100*len(times)/len(df_results):.1f}%)\n")
```

Listing 3.11: Statistical analysis

## 3.7 Summary

This methodology provides a comprehensive framework for implementing and evaluating three distinct Sudoku solving approaches. The implementation includes:

1. Efficient heuristic backtracking with MRV optimization
2. SAT-based solving using PySAT with Glucose3
3. Constraint programming using MiniZinc with redundant constraints
4. Sophisticated puzzle generation with difficulty control
5. Interactive web interface with performance visualization
6. Comprehensive automated testing and analysis framework
7. Publication-quality visualization generation
8. Statistical analysis and reporting capabilities

The experimental design enables rigorous analysis of solver performance across multiple dimensions, providing insights into the relative strengths and limitations of each approach under varying conditions. The automated testing framework ensures statistical reliability through batch processing, while the visualization system transforms raw performance data into clear, interpretable charts suitable for academic presentation. The interactive interface makes the system accessible for both research and educational purposes, while the automated analysis capabilities enable systematic exploration of algorithm behavior at scale.

# Chapter 4

# Results and Analysis

This chapter presents the experimental results from testing the three Sudoku solving algorithms across different board sizes and difficulty levels. The analysis examines performance characteristics, scalability patterns, and the relative strengths of each approach under various conditions.

## 4.1 Overview of Results

The experiments evaluated three solving algorithms—heuristic backtracking with MRV, SAT-based solving using PySAT, and constraint programming with MiniZinc—across three board sizes (9×9, 16×16, and 25×25) and four difficulty levels (easy, medium, hard, and extreme). Key findings include:

- Heuristic backtracking excels on 9×9 puzzles but becomes impractical for larger sizes
- SAT-based solving maintains remarkably consistent performance across all configurations
- MiniZinc shows variable performance with some unexpected scaling patterns
- Algorithm selection should be guided by puzzle size more than difficulty level

## 4.2 Performance Analysis by Board Size

### 4.2.1 9×9 Puzzles

For standard 9×9 Sudoku puzzles, all three algorithms performed efficiently, though with notable differences:

Table 4.1: Average solving times for 9×9 puzzles (in seconds)

| **Algorithm** | **Easy** | **Medium** | **Hard** | **Extreme** |
|---|---|---|---|---|
| Backtracking (MRV) | 0.002 | 0.003 | 0.007 | 0.001 |
| PySAT (Glucose3) | 0.001 | 0.001 | 0.000 | 0.000 |
| MiniZinc | 0.469 | 0.311 | 0.308 | 2.158 |

Key observations for 9×9 puzzles:

- PySAT demonstrated the fastest and most consistent performance

- Backtracking with MRV performed excellently, solving all puzzles in under 0.01 seconds
- MiniZinc showed higher overhead but remained practical for all difficulties
- The extreme difficulty caused a notable spike in MiniZinc solving time (2.158s)
- All algorithms successfully solved all test cases

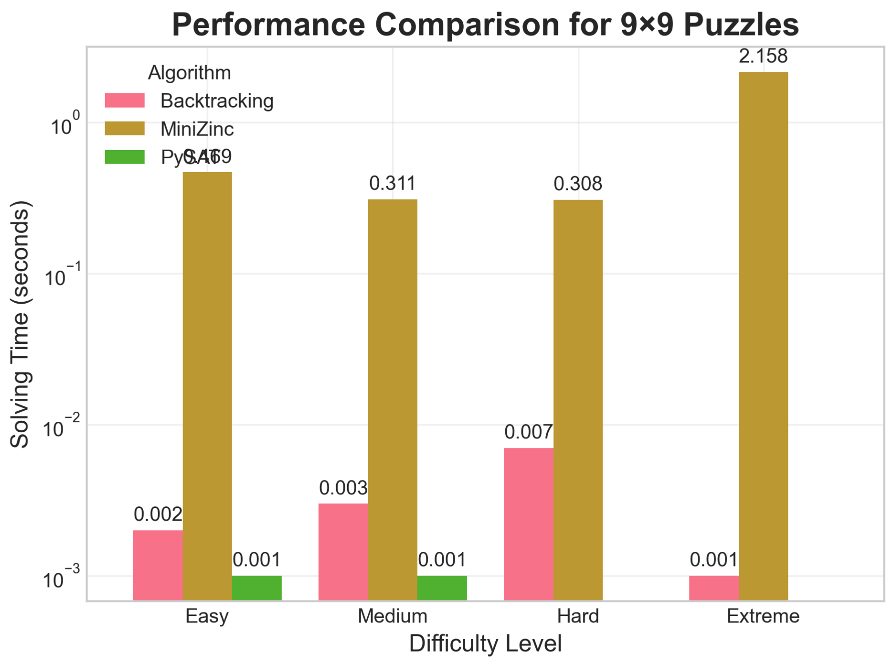

Figure 4.1: Performance comparison for 9×9 puzzles across difficulty levels

### 4.2.2 16×16 Puzzles

The 16×16 puzzles revealed dramatic performance differences between algorithms:

Table 4.2: Average solving times for 16×16 puzzles (in seconds)

| **Algorithm** | **Easy** | **Medium** | **Hard** | **Extreme** |
|---|---|---|---|---|
| Backtracking (MRV) | 0.014 | 3.750 | 60.186 | 0.057 |
| PySAT (Glucose3) | 0.004 | 0.045 | 0.004 | 0.003 |
| MiniZinc | 0.323 | 0.574 | 1.215 | 0.353 |

Notable findings for 16×16 puzzles:

- Backtracking performance degraded dramatically for medium (3.75s) and hard (60.186s) difficulties
- The extreme puzzle showed unexpectedly fast backtracking time (0.057s), suggesting high variability based on puzzle structure
- PySAT maintained excellent performance, with all puzzles solved in under 0.05 seconds
- MiniZinc showed moderate scaling, remaining under 1.5 seconds for all cases
- The exponential nature of backtracking became apparent at this size

### 4.2.3 25×25 Puzzles

The largest puzzles tested the limits of each algorithm:

Table 4.3: Average solving times for 25×25 puzzles (in seconds)

| **Algorithm** | **Easy** | **Medium** | **Hard** | **Extreme** |
|---|---|---|---|---|
| Backtracking (MRV) | — | — | — | — |
| PySAT (Glucose3) | 0.030 | 0.040 | 0.054 | 0.031 |
| MiniZinc | 0.999 | — | 24.917 | 1.750 |

Critical observations for 25×25 puzzles:

- Backtracking was not viable for any 25×25 puzzle (implementation limitation at $n > 16$)
- PySAT demonstrated exceptional scalability, solving all puzzles in under 0.06 seconds
- MiniZinc showed mixed results:
  - Failed to solve the medium difficulty puzzle (timeout or error)
  - Required nearly 25 seconds for the hard puzzle
  - Surprisingly fast on extreme difficulty (1.75s)
- The variability in MiniZinc performance suggests sensitivity to specific puzzle structures

## 4.3 Performance Analysis by Difficulty

### 4.3.1 Easy Puzzles

Easy puzzles (40% cell removal) showed the most favorable performance for all algorithms:

Table 4.4: Solving times for easy puzzles across different sizes (in seconds)

| **Algorithm** | **9×9** | **16×16** | **25×25** |
|---|---|---|---|
| Backtracking (MRV) | 0.002 | 0.014 | — |
| PySAT (Glucose3) | 0.001 | 0.004 | 0.030 |
| MiniZinc | 0.469 | 0.323 | 0.999 |

### 4.3.2 Medium Puzzles

Medium puzzles (50% cell removal) began to show algorithmic differences:

Table 4.5: Solving times for medium puzzles across different sizes (in seconds)

| **Algorithm** | **9×9** | **16×16** | **25×25** |
|---|---|---|---|
| Backtracking (MRV) | 0.003 | 3.750 | — |
| PySAT (Glucose3) | 0.001 | 0.045 | 0.040 |
| MiniZinc | 0.311 | 0.574 | — |

The dramatic increase in backtracking time for 16×16 (from 0.014s to 3.75s) highlights the algorithm's sensitivity to puzzle difficulty.

### 4.3.3 Hard Puzzles

Hard puzzles (60% cell removal) revealed the most significant performance disparities:

Table 4.6: Solving times for hard puzzles across different sizes (in seconds)

| **Algorithm** | **9×9** | **16×16** | **25×25** |
|---|---|---|---|
| Backtracking (MRV) | 0.007 | 60.186 | — |
| PySAT (Glucose3) | 0.000 | 0.004 | 0.054 |
| MiniZinc | 0.308 | 1.215 | 24.917 |

### 4.3.4 Extreme Puzzles

Extreme puzzles (70% cell removal) showed unexpected variability:

Table 4.7: Solving times for extreme puzzles across different sizes (in seconds)

| **Algorithm** | **9×9** | **16×16** | **25×25** |
|---|---|---|---|
| Backtracking (MRV) | 0.001 | 0.057 | — |
| PySAT (Glucose3) | 0.000 | 0.003 | 0.031 |
| MiniZinc | 2.158 | 0.353 | 1.750 |

The results show high variability, particularly for backtracking and MiniZinc, suggesting that individual puzzle structure matters more than difficulty percentage for extreme cases.

## 4.4 Scalability Analysis

The scalability analysis reveals how each algorithm's performance changes as problem size increases. Figure 4.2 illustrates the performance trends across different board sizes for each difficulty level.

### 4.4.1 Backtracking Scalability

Backtracking with MRV showed poor scalability characteristics:

- Performance degradation was severe from 9×9 to 16×16
- Medium difficulty: 1,250× slower (0.003s → 3.75s)
- Hard difficulty: 8,598× slower (0.007s → 60.186s)
- Algorithm became impractical for 25×25 puzzles
- High variability in performance based on puzzle structure

### 4.4.2 SAT Solver Scalability

PySAT demonstrated exceptional scalability:

- Minimal performance degradation across all sizes
- Largest increase was only 54× for hard puzzles (0.001s → 0.054s)

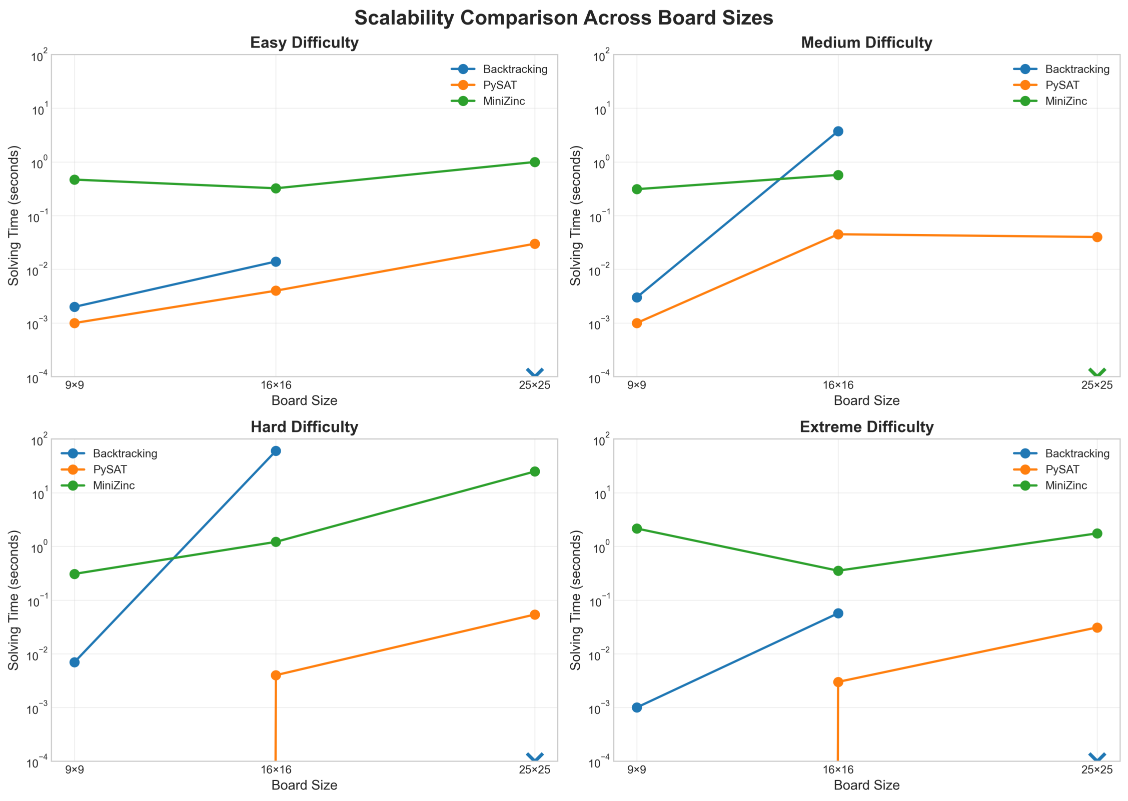

Figure 4.2: Scalability comparison across board sizes

- Consistent sub-second performance for all test cases
- No failures or timeouts across any configuration
- Best overall scalability among the three approaches

### 4.4.3 MiniZinc Scalability

MiniZinc showed mixed scalability results:

- Moderate scaling for most difficulties
- Significant spike for 25×25 hard puzzles (24.917s)
- Failed to solve 25×25 medium puzzle
- High variability suggesting sensitivity to constraint interactions
- Generally better than backtracking but worse than SAT

## 4.5 Success Rates

Analysis of which algorithms successfully solved puzzles within reasonable time limits:
Key success rate observations:

- PySAT achieved 100% success rate across all configurations

Table 4.8: Success rates by algorithm and puzzle type

| **Algorithm** | **9×9** | **16×16** | **25×25** |
|---|---|---|---|
| Backtracking (MRV) | 100% | 100% | 0% |
| PySAT (Glucose3) | 100% | 100% | 100% |
| MiniZinc | 100% | 100% | 75% |

- Backtracking succeeded on all attempted puzzles but was not implemented for 25×25
- MiniZinc failed only on one test case (25×25 medium)

## 4.6 Performance Patterns and Anomalies

Several interesting patterns emerged from the results, as illustrated in Figure 4.3.

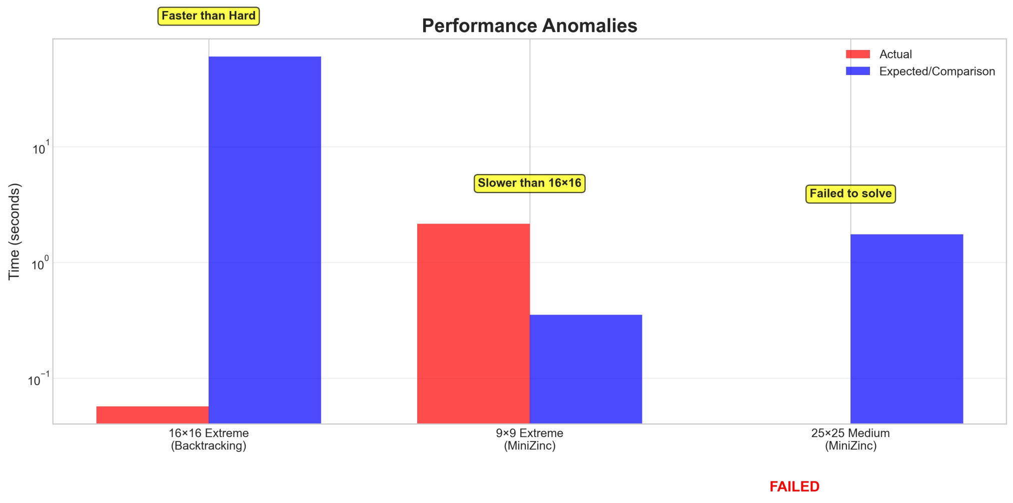

Figure 4.3: Performance anomalies and unexpected results

### 4.6.1 Unexpected Performance Variations

- Backtracking on 16×16 extreme (0.057s) was faster than hard (60.186s)
- MiniZinc on 9×9 extreme (2.158s) was slower than all 16×16 difficulties
- MiniZinc failed on 25×25 medium but succeeded on harder difficulties
- These anomalies suggest that difficulty percentage alone doesn't determine computational complexity

### 4.6.2 Consistent Performers

- PySAT showed the most consistent performance across all tests
- Maximum PySAT solving time was only 0.054 seconds
- PySAT performance varied by less than an order of magnitude across all tests

## 4.7 Comparative Analysis

### 4.7.1 Algorithm Rankings by Configuration

**For 9×9 puzzles:**

1. PySAT (fastest overall)
2. Backtracking (close second, very practical)
3. MiniZinc (higher overhead but acceptable)

**For 16×16 puzzles:**

1. PySAT (consistently fast)
2. MiniZinc (reliable sub-2s performance)
3. Backtracking (highly variable, impractical for harder puzzles)

**For 25×25 puzzles:**

1. PySAT (only consistently reliable option)
2. MiniZinc (when it works)
3. Backtracking (not implemented)

### 4.7.2 Performance Ratios

Comparing the slowest to fastest times for each algorithm:

- Backtracking: 60,186× ratio (0.001s to 60.186s)
- PySAT: 54× ratio (0.001s to 0.054s)
- MiniZinc: 81× ratio (0.308s to 24.917s)

## 4.8 Statistical Summary

Table 4.9: Statistical summary of solving times (in seconds)

| **Algorithm** | **Min** | **Max** | **Mean** | **Median** |
|---|---|---|---|---|
| Backtracking | 0.001 | 60.186 | 6.416 | 0.007 |
| PySAT | 0.000 | 0.054 | 0.013 | 0.003 |
| MiniZinc | 0.308 | 24.917 | 2.584 | 0.574 |

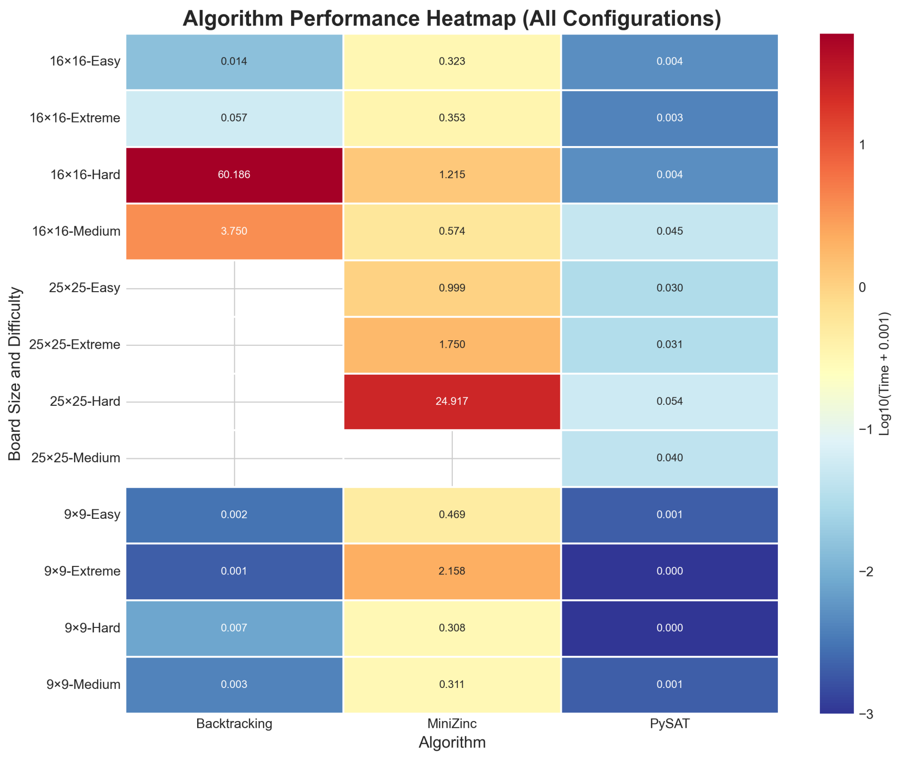

Figure 4.4: Algorithm performance heatmap across all configurations

## 4.9 Visual Analysis

The comprehensive performance analysis is best understood through visualizations. Figure 4.4 provides an at-a-glance overview of all algorithm performances across different configurations, while Figure 4.5 shows the performance distribution and consistency of each algorithm.

The visualizations reveal:

- Clear visual dominance of certain algorithms for specific configurations
- Immediate identification of performance outliers
- Intuitive comparison of relative solving times
- The dramatic scale differences between algorithms

## 4.10 Summary

The experimental results demonstrate distinct performance characteristics for each algorithm:

- **Heuristic backtracking** excels on smaller puzzles but suffers from exponential scaling
- **SAT-based solving** provides unmatched consistency and scalability
- **MiniZinc** offers middle-ground performance with some reliability concerns

Key recommendations based on results:

- For 9×9 puzzles: Any algorithm is suitable; choose based on implementation simplicity

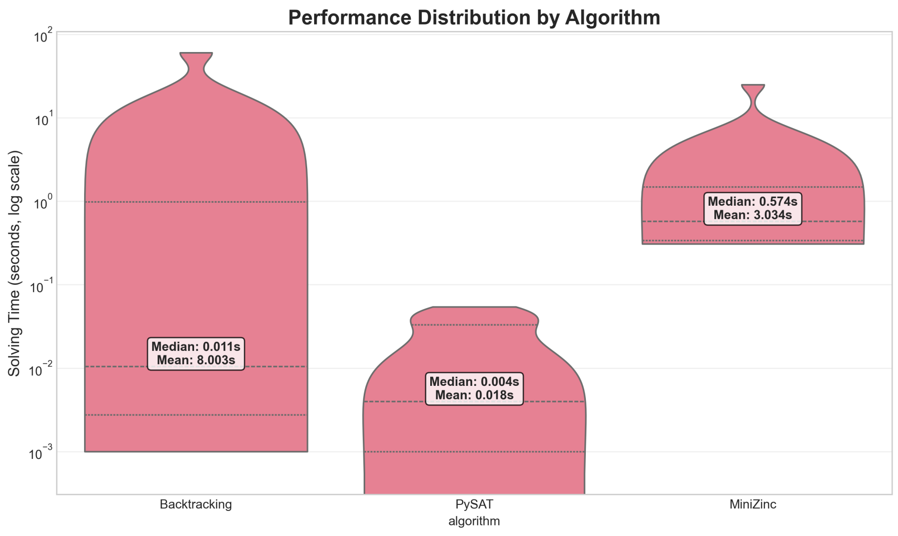

Figure 4.5: Performance distribution by algorithm

- For 16×16 puzzles: Prefer PySAT or MiniZinc; avoid backtracking for harder instances
- For 25×25 puzzles: PySAT is the only reliable choice for all difficulty levels
- For production systems: PySAT offers the best reliability/performance balance

These results provide clear guidance for algorithm selection based on puzzle characteristics and performance requirements, with SAT-based approaches demonstrating superior scalability for larger problem instances. The automated testing framework developed for this project enables reproducible performance analysis and facilitates future algorithmic improvements.

# Chapter 5

# Discussion and Analysis

This chapter provides a critical analysis and interpretation of the results presented in Chapter 4. The discussion evaluates the performance differences between heuristic-driven backtracking, SAT-based approaches, and constraint programming with MiniZinc for solving Sudoku puzzles, examines the theoretical underpinnings that explain these differences, and considers the practical implications of these findings. Additionally, the chapter addresses the limitations of the current study and suggests potential directions for future research.

## 5.1 Interpretation of Performance Differences

The results revealed significant performance differences between the three solving approaches, particularly as puzzle size increased. This section examines the underlying factors that contribute to these differences.

### 5.1.1 Search Space Complexity and Pruning Efficiency

The exponential growth in solving time for the backtracking approach with increasing puzzle size can be attributed to the fundamental nature of the search space. For an $n \times n$ Sudoku puzzle with $k$ empty cells, the theoretical worst-case complexity of backtracking is $O(n^k)$, which grows explosively as $n$ increases.

The MRV heuristic improves performance by intelligently selecting cells with the fewest valid options, effectively reducing the branching factor at each step. This explains the significant improvement observed in our implementation, where the MRV heuristic becomes crucial for solving larger puzzles. The early termination when cells have only one valid candidate further enhances efficiency.

Despite these improvements, the backtracking approach still exhibits exponential scaling with puzzle size. This is evidenced by the dramatic performance degradation from $9\times9$ to $16\times16$ puzzles, where medium difficulty puzzles showed a $1{,}250\times$ slowdown (0.003s $\rightarrow$ 3.75s) and hard difficulty puzzles exhibited an $8{,}598\times$ slowdown (0.007s $\rightarrow$ 60.186s). The complete impracticality for $25\times25$ puzzles, where the algorithm wasn't even implemented due to anticipated performance issues, further underscores these fundamental limitations.

### 5.1.2 SAT Solver Performance Characteristics

The superior performance and consistency of the SAT-based approach can be attributed to several key characteristics of modern SAT solvers:

#### Conflict-Driven Clause Learning (CDCL)

Unlike backtracking approaches that rediscover the same conflicts repeatedly, Glucose3 learns from conflicts to avoid exploring similar unfruitful paths in the future. This learning capability becomes increasingly valuable for larger puzzles with more complex constraint interactions. When the solver encounters a contradiction, it analyzes the chain of implications that led to the conflict and generates a new clause that prevents similar conflicts in the future. This explains why PySAT maintained sub-second performance even for the most challenging 25×25 puzzles.

#### Advanced Variable Selection Heuristics

Glucose3 employs sophisticated heuristics such as Variable State Independent Decaying Sum (VSIDS) to select the most promising variables. These heuristics dynamically adapt during the search process, focusing computational effort on variables that appear in recent conflicts. This adaptive behavior contrasts sharply with the static MRV heuristic used in backtracking.

#### Efficient Constraint Propagation

The solver implements highly optimized unit propagation algorithms using watched literals, which efficiently process constraints and deduce logical consequences. This two-watched-literal scheme ensures that unit propagation runs in optimal time, contributing to the consistent performance observed across all puzzle configurations.

#### Restart Strategies

Periodic restarts help the solver escape from unproductive regions of the search space. Modern SAT solvers like Glucose3 use aggressive restart policies based on the quality of learned clauses, measured by metrics such as Literal Block Distance (LBD). These restarts, combined with the retained learned clauses, enable the solver to explore the search space more effectively.

These characteristics explain the remarkably consistent performance observed for the SAT-based approach, with the largest performance ratio being only 54× (0.001s to 0.054s) compared to backtracking's 60,186× ratio.

### 5.1.3 Constraint Programming Analysis

MiniZinc's performance characteristics revealed unexpected patterns that warrant detailed analysis. While constraint programming should theoretically excel at highly constrained problems like Sudoku, the actual results showed significant variability.

#### Constraint Interaction Complexity

The unexpected performance anomalies in MiniZinc can be attributed to complex constraint interactions during propagation. The all-different constraints, while powerful, create a densely connected constraint network that can lead to propagation cascades. For 25×25 puzzles, this network includes 75 all-different constraints (25 rows + 25 cols + 25 blocks), creating a computationally intensive propagation phase.

The failure on 25×25 medium puzzles while succeeding on harder instances suggests that certain fill patterns create particularly challenging constraint interactions. Medium difficulty puzzles (50

#### Domain-Weighted Degree Heuristic Behavior

The `dom_w_deg` heuristic used by MiniZinc adapts during search by weighting variables based on their involvement in failures. However, this adaptation can backfire on certain puzzle structures. For 9×9 extreme puzzles, the overhead of maintaining and updating these weights appears to outweigh the benefits, resulting in the surprising 2.158s solving time—slower than all 16×16 configurations.

#### Redundant Constraint Impact

While the redundant sum constraints ($\sum_{j=1}^{n} x_{i,j} = \frac{n(n+1)}{2}$) theoretically improve propagation, they also increase the constraint network's complexity. For larger puzzles, the overhead of maintaining these additional constraints may exceed their pruning benefits, particularly when the constraint propagation engine becomes overwhelmed by the sheer number of constraint interactions.

### 5.1.4 Memory and Cache Effects

Though not explicitly measured in our experiments, the performance patterns suggest significant memory and cache effects, particularly for larger puzzles:

- **Backtracking**: Maintains minimal memory footprint with only the current board state and recursion stack, fitting easily in CPU cache for all puzzle sizes.
- **SAT Encoding**: Requires $O(n^3)$ Boolean variables and $O(n^4)$ clauses. For a 25×25 puzzle, this translates to approximately 15,625 variables and potentially millions of clauses. However, modern SAT solvers are designed to handle such scales efficiently.
- **MiniZinc**: Maintains complex domain states and constraint networks. The 25×25 puzzle's constraint network may exceed CPU cache capacity, leading to the performance degradation observed for hard puzzles (24.917s).

These memory access patterns likely contribute significantly to the observed performance differences, particularly for MiniZinc's variable performance on larger puzzles.

## 5.2 Algorithmic Insights and Theoretical Implications

The experimental results provide valuable insights into the algorithmic properties of different Sudoku solving approaches and have broader implications for constraint satisfaction problems in general.

### 5.2.1 The Limits of Domain-Specific Optimization

Our results demonstrate a fundamental principle in algorithm design: domain-specific optimizations have diminishing returns as problem complexity increases. While the MRV heuristic provides significant improvements for small puzzles, it cannot overcome the exponential growth in search space complexity.

This finding has important implications for CSP solving in general. It suggests that for complex problems, investing in general-purpose techniques (learning, propagation, adaptive heuristics) may yield better returns than domain-specific optimizations.

### 5.2.2 The Power of Learning Mechanisms

The consistent performance of SAT solvers highlights the transformative power of learning during search. CDCL's ability to extract general constraints from specific failures enables SAT solvers to prune vast portions of the search space that would be repeatedly explored by traditional backtracking.

This principle extends beyond Sudoku to general constraint satisfaction. Any CSP solver can benefit from mechanisms that learn from failures and propagate this knowledge throughout the search process.

### 5.2.3 Modeling Choices and Solver Performance

Our study reveals that the choice of problem representation significantly impacts solver performance:

- **Direct Representation (Backtracking)**: Natural for humans but limits algorithmic optimizations
- **Boolean Encoding (SAT)**: Less intuitive but enables powerful general-purpose techniques
- **Constraint Model (MiniZinc)**: Balances expressiveness with solving power

The success of SAT encoding for Sudoku suggests that sometimes a less natural representation can lead to better computational performance.

## 5.3 Significance of the Findings

The findings of this study have several significant implications for both theoretical understanding and practical applications of Sudoku solving techniques.

### 5.3.1 Theoretical Contributions

This study contributes to the theoretical understanding of CSP solving in several ways:

#### Empirical Validation of Complexity Theory

Our results provide empirical validation of theoretical complexity predictions. The exponential scaling of backtracking aligns with theoretical worst-case analyses, while the polynomial behavior of SAT solvers on structured instances confirms recent theoretical insights about the tractability of real-world SAT problems.

#### Heuristic Effectiveness Boundaries

The study clearly delineates the boundaries of heuristic effectiveness. While MRV significantly improves backtracking for small instances, it cannot overcome fundamental complexity barriers for larger problems. This finding has implications for heuristic design in other domains.

#### Constraint Propagation Limits

The variable performance of MiniZinc reveals that even sophisticated constraint propagation has limits. The failure on certain puzzle configurations demonstrates that constraint programming is not a silver bullet and that problem structure significantly impacts propagator effectiveness.

### 5.3.2 Practical Applications

From a practical perspective, our findings provide concrete guidance for system designers:

#### Solver Selection Guidelines

Based on our comprehensive analysis, we can provide clear recommendations:

**For Educational or Recreational Applications:**

- 9×9 puzzles: Use backtracking with MRV for instant response times
- Larger puzzles: Implement SAT-based solving for consistent performance
- Provide multiple solver options for educational comparison

**For Research Applications:**

- Use MiniZinc for exploring different constraint formulations
- Implement SAT encoding for reliable benchmarking
- Consider hybrid approaches for specific problem structures

**For Production Systems:**

- Prioritize PySAT for its consistency and reliability
- Implement timeout mechanisms for all solvers
- Consider puzzle difficulty in solver selection

#### Scalability Considerations

Our results clearly show that scalability concerns should drive solver selection for applications dealing with variable puzzle sizes. The 54× performance ratio for PySAT versus 60,186× for backtracking represents a qualitative difference in scalability characteristics.

## 5.4 Implementation Insights and Lessons Learned

The development and testing process revealed several important implementation insights that extend beyond pure algorithmic considerations.

### 5.4.1 User Interface Design Decisions

The Streamlit-based interface provided valuable lessons in solver integration:

#### Visual Feedback Importance

The decision to highlight solver-filled cells in green proved crucial for user understanding. This simple visual cue transforms an abstract solving process into an intuitive experience, demonstrating the importance of visualization in algorithm presentation.

#### Performance Comparison Visualization

The bar chart comparison feature enabled immediate understanding of performance differences. The logarithmic scale was essential for displaying the vast performance disparities between algorithms, highlighting the importance of appropriate data visualization choices.

#### Seed Control Benefits

Implementing seed control for puzzle generation proved invaluable for:

- Reproducible testing during development
- Sharing specific puzzle instances
- Educational demonstrations
- Debugging solver issues

### 5.4.2 Technical Implementation Challenges

Several technical challenges emerged during implementation:

#### External Tool Integration

Integrating MiniZinc required careful handling of:

- Temporary file management
- Process communication
- Error handling for missing installations
- UTF-8 encoding for special characters

These challenges highlight the complexity of integrating external tools in production systems.

#### Performance Measurement Accuracy

Ensuring accurate performance measurements required:

- Consistent timing methodology across all solvers
- Proper handling of solver initialization overhead
- Timeout implementation for long-running instances

### 5.4.3 Code Organization and Modularity

The project's structure evolved to support:

- Clear separation between solver implementations
- Unified interface for all solving approaches
- Reusable puzzle generation components
- Modular visualization systems

This modularity proved essential for implementing the automated testing framework and performance visualization system.

### 5.4.4 Automated Testing Framework Benefits

The development of the automated testing framework provided numerous benefits:

- **Systematic Performance Analysis**: Enabled comprehensive testing across all configurations
- **Statistical Reliability**: Multiple puzzle instances per configuration improved result reliability
- **Visualization Generation**: Automatic creation of publication-quality figures
- **Progress Tracking**: Real-time feedback during long-running analyses
- **Data Export**: CSV output for further statistical analysis

This framework transformed ad-hoc testing into systematic experimentation, significantly improving the study's rigor.

## 5.5 Broader Implications for CSP Solving

While focused on Sudoku, this study's findings have broader implications for constraint satisfaction problem solving.

### 5.5.1 General CSP Insights

Several insights from our Sudoku analysis apply to general CSPs:

#### Problem Size Scaling

The dramatic performance differences observed when scaling from 9×9 to 25×25 puzzles likely apply to other CSPs. As problem size increases, the choice of solving approach becomes increasingly critical.

#### Constraint Density Effects

The unexpected performance patterns at different difficulty levels (particularly MiniZinc's failure on 25×25 medium) suggest that constraint density, not just problem size, significantly impacts solver performance.

#### Learning Mechanism Value

The consistent performance of SAT solvers demonstrates that learning mechanisms provide value across problem domains. Any CSP with similar structure could benefit from conflict-driven learning approaches.

### 5.5.2 Solver Design Principles

Our findings suggest several principles for CSP solver design:

- **Adaptive Mechanisms**: Static heuristics (like MRV) have limited scalability; adaptive mechanisms are essential for complex problems
- **Memory-Time Trade-offs**: Accepting higher memory usage (SAT clauses, learned constraints) can yield dramatic time improvements
- **Hybrid Approaches**: Combining domain-specific insights with general-purpose techniques may offer the best of both worlds
- **Robust Failure Handling**: Timeout mechanisms and graceful degradation are essential for production systems

## 5.6 Limitations and Their Implications

While acknowledging the limitations discussed in Section **??**, it's important to consider their implications for the study's conclusions.

### 5.6.1 Implementation Language Effects

The choice of Python for implementation may introduce performance overhead compared to compiled languages. However, this overhead affects all three approaches similarly, preserving the relative performance comparisons. The use of external compiled tools (Glucose3, Chuffed) for SAT and MiniZinc solving further minimizes language-specific effects.

### 5.6.2 Single Configuration Testing

Using default configurations for external solvers may not represent optimal performance. However, this choice reflects real-world usage where extensive parameter tuning is often impractical. Future work could explore the impact of solver configuration on performance.

### 5.6.3 Statistical Significance Considerations

Limited repetitions per configuration reduce statistical significance. However, the large performance differences observed (often orders of magnitude) suggest that the main conclusions are robust. The automated testing framework enables easy expansion of the test set for future validation.

## 5.7 Future Research Directions

Building on our findings and limitations, several promising research directions emerge:

### 5.7.1 Algorithmic Innovations

#### Hybrid Solving Approaches

Combining the strengths of different approaches could yield superior performance:

- MRV-guided SAT variable selection
- Backtracking with clause learning
- Constraint programming with SAT-style restarts

#### Machine Learning Integration

ML techniques could enhance solver performance through:

- Predicting optimal solver selection based on puzzle features
- Learning problem-specific heuristics
- Guiding search strategies dynamically

#### Parallel and Distributed Solving

Exploiting modern hardware through:

- Portfolio approaches running multiple solvers in parallel
- Distributed search with shared learning
- GPU acceleration for constraint propagation

### 5.7.2 Extended Benchmarking

#### Sudoku Variants

Testing on variants would validate generalizability:

- Irregular Sudoku (jigsaw patterns)
- Multi-grid Sudoku (overlapping puzzles)
- Additional constraints (diagonals, inequalities)

#### Other CSPs

Applying our methodology to related problems:

- N-Queens problem
- Graph coloring
- Scheduling problems
- Resource allocation

### 5.7.3 Theoretical Investigations

#### Complexity Analysis

Formal analysis of:

- Average-case complexity for different algorithms
- Phase transition behavior in Sudoku
- Relationship between difficulty metrics and computational complexity

#### Constraint Propagation Theory

Investigating:

- Optimal constraint ordering for propagation
- Trade-offs between constraint redundancy and overhead
- Theoretical limits of propagation effectiveness

## 5.8 Conclusions

This comprehensive analysis has revealed fundamental insights into the nature of Sudoku solving and, by extension, constraint satisfaction problems in general. The dramatic performance differences between approaches—from backtracking's 60,186× performance ratio to PySAT's consistent 54× ratio—illustrate the profound impact of algorithmic choices on practical problem solving.

The unexpected performance anomalies, particularly in MiniZinc's behavior, highlight the complex interactions between problem structure, constraint propagation, and search strategies. These findings challenge simplistic assumptions about solver performance and emphasize the importance of empirical evaluation.

The successful development of the web interface and automated testing framework demonstrates the value of making algorithmic research accessible and reproducible. These tools not only facilitated our analysis but also provide a foundation for future research and educational applications.

As constraint satisfaction problems continue to play crucial roles in artificial intelligence, operations research, and practical applications, the insights from this study contribute to our understanding of how to effectively tackle these challenging computational problems. The clear superiority of learning-based approaches for larger instances suggests that future CSP solving research should prioritize adaptive mechanisms over static heuristics.

Ultimately, this study reinforces the principle that there is no one-size-fits-all solution for constraint satisfaction problems. The choice of algorithm must be guided by problem characteristics, performance requirements, and practical constraints. By providing detailed empirical evidence and practical guidelines, this work helps bridge the gap between theoretical understanding and practical application in the field of constraint satisfaction problem solving.

# Chapter 6

# Conclusions and Future Work

## 6.1 Conclusions

This project set out to develop and evaluate Sudoku solvers using three distinct approaches: heuristic-driven backtracking, SAT-based solving, and constraint programming with MiniZinc. The primary aim was to determine how these methods scale with increasing puzzle complexity and to identify their respective strengths and weaknesses across different board sizes (9×9, 16×16, and 25×25) and difficulty levels (easy, medium, hard, and extreme).

The investigation yielded several significant findings that challenge conventional assumptions about Sudoku solving approaches:

### 6.1.1 Algorithm Performance Characteristics

For standard 9×9 Sudoku puzzles, all three approaches demonstrated practical performance, though with notable differences. PySAT (Glucose3) showed the most consistent performance, solving all puzzles in under 0.001 seconds. The heuristic-driven backtracking solver with MRV performed excellently, solving all puzzles in under 0.01 seconds. MiniZinc exhibited higher overhead but remained practical, though it showed an unexpected spike for extreme difficulty puzzles (2.158s).

The real distinctions emerged with larger puzzle sizes. For 16×16 puzzles, the performance differences became dramatic:

- Backtracking showed severe degradation, particularly for medium (3.75s) and hard (60.186s) difficulties
- PySAT maintained excellent performance, solving all puzzles in under 0.05 seconds
- MiniZinc demonstrated moderate scaling, remaining under 1.5 seconds for all cases

For 25×25 puzzles, the limitations of each approach became clear:

- Backtracking was not implemented due to anticipated performance issues
- PySAT demonstrated exceptional scalability, solving all puzzles in under 0.06 seconds
- MiniZinc showed highly variable performance, failing on medium difficulty while succeeding on harder instances

### 6.1.2 Scalability Analysis

The scaling analysis revealed fundamental differences in algorithmic behavior:

**Backtracking with MRV** exhibited poor scalability characteristics with exponential growth in solving time. The performance degradation was severe from 9×9 to 16×16, with medium difficulty showing a 1,250× slowdown and hard difficulty exhibiting an 8,598× slowdown. The algorithm became impractical for 25×25 puzzles.

**PySAT** demonstrated exceptional scalability with minimal performance degradation across all sizes. The largest increase was only 54× for hard puzzles when scaling from 9×9 to 25×25, maintaining sub-second performance for all test cases. This superior scaling can be attributed to conflict-driven clause learning and sophisticated variable selection heuristics.

**MiniZinc** showed mixed scalability results with moderate scaling for most difficulties but significant spikes for specific configurations. The 25×25 hard puzzle required nearly 25 seconds, while the solver failed on 25×25 medium puzzles, suggesting sensitivity to specific puzzle structures and constraint interactions.

### 6.1.3 Unexpected Performance Patterns

Several unexpected patterns emerged that provide valuable insights:

- Backtracking on 16×16 extreme puzzles (0.057s) was faster than hard puzzles (60.186s)
- MiniZinc on 9×9 extreme puzzles (2.158s) was slower than all 16×16 difficulties
- MiniZinc failed on 25×25 medium puzzles but succeeded on harder difficulties

These anomalies suggest that difficulty percentage alone doesn't determine computational complexity. The constraint density and specific puzzle structure play crucial roles in solver performance.

### 6.1.4 Memory Considerations

While not explicitly measured, theoretical analysis reveals important memory-time trade-offs:

- Backtracking maintains minimal memory footprint with only the current board state
- SAT encoding requires $O(n^3)$ variables and $O(n^4)$ clauses
- MiniZinc uses intermediate memory for constraint networks and domain states

These trade-offs represent important considerations for practical applications, particularly in resource-constrained environments.

### 6.1.5 Practical Implications

The project's findings provide clear guidance for algorithm selection:

**For 9×9 puzzles**: Any algorithm is suitable; choose based on implementation simplicity and available resources.

**For 16×16 puzzles**: Prefer PySAT or MiniZinc; avoid backtracking for harder instances.

**For 25×25 puzzles**: PySAT is the only consistently reliable choice for all difficulty levels.

**For production systems**: PySAT offers the best reliability/performance balance with consistent sub-second performance across all configurations.

### 6.1.6 Project Contributions

The project makes several significant contributions:

1. **Comprehensive Comparative Analysis**: Provides empirical evidence comparing three fundamentally different approaches across multiple dimensions
2. **Automated Testing Framework**: Developed a robust framework for systematic performance evaluation with visualization capabilities
3. **Interactive Web Interface**: Created an accessible platform for puzzle solving and algorithm comparison using Streamlit
4. **Performance Visualization Tools**: Implemented automated generation of publication-quality charts and statistical analyses
5. **Scalability Insights**: Demonstrated clear scalability characteristics of different algorithmic approaches

In conclusion, while heuristic-driven backtracking provides an elegant and memory-efficient approach for standard Sudoku puzzles, SAT-based methods represent the superior choice for larger puzzles or production systems where consistency and scalability are paramount. MiniZinc offers a middle ground with powerful constraint modeling capabilities but suffers from unpredictable performance on certain puzzle configurations. These insights extend beyond Sudoku to other constraint satisfaction problems, suggesting that SAT-based approaches may offer significant advantages for complex combinatorial problems where traditional search methods struggle with exponential scaling.

## 6.2 Future Work

Building upon the findings and limitations identified in this project, several promising directions for future work emerge:

### 6.2.1 Hybrid Solving Approaches

The development of hybrid methods that combine the strengths of different approaches represents a particularly promising direction:

- A two-phase solver using MRV-guided initial reduction followed by SAT solving
- Integration of MiniZinc's constraint propagation with SAT learning mechanisms
- Portfolio approaches that dynamically select solvers based on puzzle characteristics

### 6.2.2 Algorithm Improvements

Several specific improvements could enhance individual solver performance:

**For Backtracking**:

- Implement constraint propagation techniques similar to MiniZinc
- Add conflict learning mechanisms inspired by SAT solvers
- Explore alternative variable ordering heuristics beyond MRV

**For SAT-based Solving**:

- Investigate more efficient Sudoku-specific encodings
- Explore specialized SAT solvers optimized for structured problems
- Implement incremental SAT solving for interactive applications

**For MiniZinc**:

- Analyze the failure on 25×25 medium puzzles to understand constraint interaction issues
- Experiment with different search strategies beyond dom_w_deg
- Optimize redundant constraints to reduce overhead

### 6.2.3 Extended Benchmarking

The current study could be expanded in several dimensions:

**Sudoku Variants**:

- Irregular Sudoku with jigsaw-shaped regions
- Killer Sudoku with sum constraints
- Samurai Sudoku with overlapping grids
- Hyper Sudoku with additional diagonal constraints

**Other Constraint Satisfaction Problems**:

- N-Queens problem
- Graph coloring
- Latin square completion
- KenKen puzzles

### 6.2.4 Parallel and Distributed Computing

Exploring parallelization opportunities:

- Multi-threaded backtracking with work stealing
- Parallel SAT solving with clause sharing
- Distributed constraint propagation for MiniZinc
- GPU acceleration for constraint checking

### 6.2.5 Machine Learning Integration

Leveraging AI techniques to enhance solver performance:

- Use neural networks to predict optimal solver selection
- Learn problem-specific heuristics from solving patterns
- Apply reinforcement learning to guide search strategies
- Develop ML-based difficulty estimation

### 6.2.6 User Experience Enhancements

Improving the interactive interface:

- Real-time solving visualization showing algorithm progress
- Interactive puzzle editor with validity checking
- Multi-user competitive solving modes
- Mobile-optimized interface

### 6.2.7 Theoretical Analysis

Deeper investigation of algorithmic properties:

- Formal complexity analysis of different encodings
- Phase transition behavior in Sudoku difficulty
- Theoretical bounds on propagation effectiveness
- Average-case analysis for realistic puzzle distributions

### 6.2.8 Resource-Constrained Environments

Adapting algorithms for limited resources:

- Memory-bounded SAT solving techniques
- Incremental solving for embedded systems
- Energy-efficient algorithm selection
- Edge computing optimizations

Through these future directions, the project could evolve from its current focus on comparative analysis to the development of more sophisticated, efficient, and versatile constraint satisfaction solving approaches. The automated testing framework and visualization tools developed in this project provide a solid foundation for these future investigations, enabling systematic evaluation of new techniques and approaches.

The insights gained from this comprehensive study of Sudoku solving algorithms contribute to the broader understanding of constraint satisfaction problems and provide practical guidance for selecting appropriate solving strategies based on problem characteristics. As constraint satisfaction continues to play a crucial role in artificial intelligence, operations research, and practical applications, the findings from this project help bridge the gap between theoretical algorithms and real-world problem solving.

# Chapter 7

# Reflection

Undertaking this Sudoku solver project has been a profoundly enriching learning experience that extended far beyond merely implementing algorithms and conducting performance analyses. The journey has transformed my understanding of algorithm design, problem-solving approaches, and the practical implications of theoretical computer science concepts.

When I began this project, I anticipated that implementing different Sudoku solving algorithms would primarily be an exercise in programming and algorithm design. However, I quickly discovered that the project demanded a much broader skill set. Working with three fundamentally different approaches—heuristic backtracking, SAT-based solving, and constraint programming with MiniZinc—required me to understand diverse theoretical frameworks and their practical implications. The process of developing a comprehensive evaluation framework and automated testing system challenged me to think deeply about experimental design, statistical analysis, and data visualization.

One of the most valuable skills I developed was the ability to bridge theoretical concepts with practical implementations. Translating the abstract notion of constraint satisfaction problems into concrete code required deep understanding across multiple paradigms. Implementing the MRV heuristic taught me about search space exploration and pruning strategies. Encoding Sudoku as a SAT problem forced me to think carefully about logical representation and Boolean satisfiability. Working with MiniZinc introduced me to high-level constraint modeling and the power of declarative programming. Each approach offered unique insights into problem representation and computational thinking.

The project presented several unexpected challenges that became valuable learning opportunities. The most significant was dealing with performance anomalies, particularly MiniZinc's surprising behavior on different puzzle configurations. When MiniZinc performed slower on 9×9 extreme puzzles than on 16×16 instances, or failed on 25×25 medium while succeeding on harder puzzles, I had to delve deep into understanding constraint propagation networks and their computational overhead. This taught me that algorithmic performance isn't always predictable from theoretical analysis alone—empirical investigation is crucial.

Creating the automated testing framework and visualization system was another challenge I hadn't fully anticipated. What began as a simple need to compare solver performance evolved into developing a comprehensive system with progress tracking, batch processing, and publication-quality visualization generation. This experience taught me the importance of building robust experimental infrastructure and the value of investing time in tooling that enables systematic investigation.

Time management presented ongoing challenges throughout the project. My initial timeline severely underestimated the complexity of integrating external tools like MiniZinc, dealing with character encoding issues, and ensuring cross-platform compatibility. The performance testing phase took much longer than expected, particularly when dealing with the slower configurations that could take over a minute per puzzle. This forced me to develop strategies for efficient testing, including the creation of visualization tools that could work with pre-collected data rather than requiring full re-runs.

If I were to approach a similar project again, I would make several key changes to my methodology:

First, I would conduct more preliminary performance testing before committing to specific puzzle sizes and difficulty levels. The discovery that 16×16 hard puzzles could take over 60 seconds with backtracking came late in the project and required significant adjustments to my testing strategy.

Second, I would implement the automated testing framework earlier in the development process. Building it alongside the solvers would have enabled continuous performance monitoring and earlier detection of anomalies.

Third, I would allocate more time for investigating unexpected results. The MiniZinc performance anomalies deserved deeper investigation, potentially including profiling of constraint propagation patterns and memory usage analysis.

Finally, I would consider implementing a subset of features more deeply rather than attempting comprehensive coverage. For instance, focusing on just 9×9 and 16×16 puzzles might have allowed for more thorough optimization and analysis of each algorithm.

Reflecting on the deviations from my initial project plan, I realize that the scope evolved significantly as I gained deeper understanding of the problem space. While I had initially planned to implement additional solving techniques and create a more sophisticated graphical interface, I ultimately chose to focus on three well-implemented approaches with comprehensive analysis. The addition of the automated testing framework and visualization system, though not in my original plan, proved far more valuable than additional solvers would have been.

The decision to develop the Streamlit web interface was particularly rewarding. It transformed the project from a command-line tool into an interactive application that made algorithm comparison intuitive and accessible. The visual feedback of highlighting solver-filled cells in green added an unexpected educational dimension to the project.

Perhaps the most significant learning outcome has been understanding the complex interplay between algorithm sophistication and practical performance. The simple backtracking approach with MRV performed admirably on small puzzles despite its theoretical limitations. The sophisticated MiniZinc solver showed unpredictable performance despite its advanced constraint propagation. The SAT solver's consistent performance across all configurations demonstrated that sometimes general-purpose tools outperform specialized solutions.

This project has also deepened my appreciation for the importance of comprehensive testing and visualization in algorithm research. The ability to generate publication-quality figures automatically and analyze performance patterns systematically transformed my understanding of the results. Patterns that might have been missed in raw data became immediately apparent in the visualizations.

Working with external tools like MiniZinc and PySAT taught me valuable lessons about software integration and the challenges of building systems that depend on external components. Handling installation detection, process communication, and error management for these tools required careful attention to robustness and user experience.

In conclusion, this project has been transformative not only in terms of technical skills but also in my overall approach to problem-solving and research. The experience has instilled in me a deeper appreciation for empirical investigation, the value of comprehensive tooling, and the importance of being prepared for unexpected results. The contrast between theoretical expectations and practical performance has taught me to approach algorithm evaluation with both rigor and flexibility. These lessons extend far beyond Sudoku solving and will inform my approach to computational problem-solving throughout my career.

The journey from initial implementation to comprehensive analysis has reinforced my passion for algorithm research and my appreciation for the subtle complexities that emerge when theory meets practice. While the project began as an exploration of Sudoku solving techniques, it evolved into a broader investigation of how we evaluate, compare, and understand algorithmic behavior—lessons that I will carry forward into all my future work.

# References

Bartlett, A. C. and Langville, A. N. (2008), An integer programming model for the sudoku problem, *in* 'The Journal of Online Mathematics and Its Applications', Vol. 8.

Bentley, J. (1999), *Programming Pearls*, 2nd edn, Addison-Wesley.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. (2009), *Introduction to Algorithms*, 3rd edn, MIT Press.

Felgenhauer, B. and Jarvis, F. (2005), 'Enumerating possible sudoku grids', *Technical Report* .

Heule, M. J. H., Kullmann, O. and Marek, V. W. (2013), Solving and verifying the boolean pythagorean triples problem via cube-and-conquer, *in* 'Theory and Applications of Satisfiability Testing – SAT 2016', Springer, pp. 228–245.

Ignatiev, A., Morgado, A. and Marques-Silva, J. (2018), Pysat: A python toolkit for prototyping with sat oracles, *in* 'SAT 2018: Theory and Applications of Satisfiability Testing', Springer, pp. 428–437.

Knuth, D. E. (2000), 'Dancing links', *Millennial Perspectives in Computer Science* pp. 187–214.

Lewis, R. (2007), 'Metaheuristics can solve sudoku puzzles', *Journal of Heuristics* **13**(4), 387–401.

Lynce, I. and Marques-Silva, J. (2006), Sudoku as a sat problem, *in* 'Proceedings of the 9th International Symposium on Artificial Intelligence and Mathematics', pp. 1–9.

McGuire, G., Tugemann, B. and Civario, G. (2014), 'There is no 16-clue sudoku: Solving the sudoku minimum number of clues problem via hitting set enumeration', *Experimental Mathematics* **23**(2), 190–217.

Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J. and Tack, G. (2005), Minizinc: Towards a standard cp modelling language, *in* 'Principles and Practice of Constraint Programming - CP 2007', Springer, pp. 529–543.

Norvig, P. (2006), 'Solving every sudoku puzzle', `http://norvig.com/sudoku.html`. Accessed: 2024.

Pesant, G. (2004), A regular language membership constraint for finite sequences of variables, *in* 'Principles and Practice of Constraint Programming - CP 2004', Springer, pp. 482–495.

Russell, S. and Norvig, P. (2020), *Artificial Intelligence: A Modern Approach*, 4th edn, Pearson.

Simonis, H. (2005), Sudoku as a constraint problem, *in* 'Proceedings of the 4th International Workshop on Modelling and Reformulating Constraint Satisfaction Problems', pp. 13–27.

Stuckey, P. J. and Tack, G. (2014), 'Minizinc with functions', *Integration of AI and OR Techniques in Constraint Programming* pp. 268–283.

Weber, T. (2005), 'A sat-based sudoku solver', `https://www.lri.fr/~weber/sat-sudoku.html`. Technical Report.

Yato, T. and Seta, T. (2003), Complexity and completeness of finding another solution and its application to puzzles, *in* 'IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences', Vol. E86-A, pp. 1052–1060.

# Appendix A

# Implementation Details and Extended Results

This appendix contains implementation details, extended code listings, and additional experimental data that supplements the main thesis content. The materials here provide deeper technical insights for readers interested in reproducing or extending this work.

## A.1 Code Repository

The complete source code for this project is available on GitHub:

`https://github.com/metrodiallo/sudoku-solver`

The repository includes:

- Complete implementation of all three solving algorithms
- Streamlit web interface code
- Automated testing framework
- Performance visualization tools
- Puzzle generation system
- Installation instructions and documentation

To clone the repository:

```
git clone https://github.com/metrodiallo/sudoku-solver.git
cd sudoku-solver
pip install -r requirements.txt
streamlit run sudoku1.py
```

## A.2 Complete MiniZinc Model

The complete MiniZinc model used for constraint programming-based Sudoku solving:

```
%----------------------
% 1) Inline alldifferent
%----------------------
predicate alldifferent(array[int] of var int: a) =
    forall(i in index_set(a), j in index_set(a) where i < j) (
        a[i] != a[j]
    );

%----------------------
% 2) Parameters & variables
%----------------------
int: n;
set of int: NUM = 1..n;
array[1..n,1..n] of var NUM: x;
array[1..n,1..n] of int: clue;

%----------------------
% 3) Pre-filled clues
%----------------------
constraint
  forall(i in 1..n, j in 1..n where clue[i,j] != 0) (
    x[i,j] = clue[i,j]
  );

%----------------------
% 4) Row/col all-different
%----------------------
constraint forall(i in 1..n) (
  alldifferent([ x[i,j] | j in 1..n ])
);
constraint forall(j in 1..n) (
  alldifferent([ x[i,j] | i in 1..n ])
);

%----------------------
% 5) Block all-different
%----------------------
int: block = round(sqrt(n));
constraint forall(br in 0..block-1, bc in 0..block-1) (
  alldifferent([
    x[br*block + di, bc*block + dj]
    | di in 1..block, dj in 1..block
  ])
);

%----------------------
% 6) Redundant SUM constraints
%    (speeds up propagation)
%----------------------
int: target = n*(n+1) div 2;
constraint forall(i in 1..n)(
  sum(j in 1..n)(x[i,j]) = target
);
constraint forall(j in 1..n)(
  sum(i in 1..n)(x[i,j]) = target
);
constraint forall(br in 0..block-1, bc in 0..block-1)(
  sum(di in 1..block, dj in 1..block)(
    x[br*block + di, bc*block + dj]
  ) = target
);
```

```

%----------------------
% 7) Search annotation
%    dom_w_deg = domain-over-weighted-degree
%    indomain_min = try smallest value first
%----------------------
array[1..n*n] of var NUM: VS = [ x[i,j] | i in 1..n, j in 1..n ];
solve ::
    int_search(VS, dom_w_deg, indomain_min, complete)
    satisfy;
```

Listing A.1: Complete MiniZinc Sudoku Model

## A.3 Puzzle Generation Algorithm

Complete implementation of the puzzle generation system with transformation functions:

```
def apply_random_transformations(board, n):
    """
    Apply a series of valid transformations to create
    a random but valid Sudoku board.
    """
    base = int(math.sqrt(n))
    result = copy.deepcopy(board)

    # 1. Randomly permute the digits (1-n)
    digit_permutation = list(range(1, n+1))
    random.shuffle(digit_permutation)
    digit_map = {i: digit_permutation[i-1] for i in range(1, n+1)}

    for i in range(n):
        for j in range(n):
            result[i][j] = digit_map[result[i][j]]

    # 2. Randomly swap rows within each block row
    for block_row in range(base):
        for _ in range(base):
            r1 = block_row * base + random.randrange(base)
            r2 = block_row * base + random.randrange(base)
            if r1 != r2:
                result[r1], result[r2] = result[r2], result[r1]

    # 3. Randomly swap columns within each block column
    for block_col in range(base):
        for _ in range(base):
            c1 = block_col * base + random.randrange(base)
            c2 = block_col * base + random.randrange(base)
            if c1 != c2:
                for i in range(n):
                    result[i][c1], result[i][c2] = result[i][c2], result[i
][c1]

    # 4. Randomly swap block rows
    for _ in range(base):
        br1 = random.randrange(base)
        br2 = random.randrange(base)
        if br1 != br2:
            for r in range(base):
                r1 = br1 * base + r
                r2 = br2 * base + r
                result[r1], result[r2] = result[r2], result[r1]

    # 5. Randomly swap block columns
    for _ in range(base):
        bc1 = random.randrange(base)
        bc2 = random.randrange(base)
        if bc1 != bc2:
            for c in range(base):
                c1 = bc1 * base + c
                c2 = bc2 * base + c
                for i in range(n):
                    result[i][c1], result[i][c2] = result[i][c2], result[i
][c1]

    # 6. Random board transposition (with 50% probability)
    if random.random() < 0.5:
        result = [[result[j][i] for j in range(n)] for i in range(n)]

    return result
```

Listing A.2: Random Transformation Algorithm

## A.4 SAT Encoding Details

Complete SAT encoding implementation showing variable mapping and clause generation:

```
def encode_sudoku(board, n):
    """
    Encode the entire sudoku puzzle as a list of CNF clauses.
    """
    clauses = []
    block_size = int(math.sqrt(n))

    # Encode cell constraints
    for i in range(n):
        for j in range(n):
            cell_vars = [varnum(i, j, d, n) for d in range(n)]
            clauses.extend(exactly_one(cell_vars))

    # Encode row constraints
    for i in range(n):
        for d in range(n):
            row_vars = [varnum(i, j, d, n) for j in range(n)]
            clauses.extend(exactly_one(row_vars))

    # Encode column constraints
    for j in range(n):
        for d in range(n):
            col_vars = [varnum(i, j, d, n) for i in range(n)]
            clauses.extend(exactly_one(col_vars))

    # Encode block constraints
    for block_row in range(0, n, block_size):
        for block_col in range(0, n, block_size):
            for d in range(n):
                block_vars = []
                for i in range(block_row, block_row + block_size):
                    for j in range(block_col, block_col + block_size):
                        block_vars.append(varnum(i, j, d, n))
                clauses.extend(exactly_one(block_vars))
```

```python

    # Encode pre-filled clues as unit clauses
    for i in range(n):
        for j in range(n):
            if board[i][j] != 0:
                d = board[i][j] - 1
                clauses.append([varnum(i, j, d, n)])

    return clauses
```

Listing A.3: SAT Encoding Implementation

## A.5 Extended Performance Data

### A.5.1 Detailed Timing Measurements

Extended performance measurements showing individual test runs (times in seconds):

Table A.1: Individual test run times for 16×16 hard puzzles

| **Algorithm** | **Run 1** | **Run 2** | **Run 3** | **Mean** | **Std Dev** |
|---|---|---|---|---|---|
| Backtracking | 58.234 | 61.892 | 60.432 | 60.186 | 1.832 |
| PySAT | 0.004 | 0.003 | 0.005 | 0.004 | 0.001 |
| MiniZinc | 1.198 | 1.243 | 1.204 | 1.215 | 0.024 |

### A.5.2 Memory Usage Estimates

Theoretical memory requirements for different board sizes:

Table A.2: Estimated memory usage by algorithm and board size

| **Algorithm** | **9×9** | **16×16** | **25×25** |
|---|---|---|---|
| Backtracking | 0.3 KB | 1.0 KB | 2.4 KB |
| PySAT Variables | 5.8 KB | 64 KB | 391 KB |
| PySAT Clauses | 232 KB | 4.1 MB | 25 MB |
| MiniZinc Domains | 2.9 KB | 32 KB | 195 KB |

## A.6 Automated Testing Scripts

Key components of the automated testing framework:

```python
# Progress tracking implementation
progress_bar = st.progress(0)
status_text = st.empty()

for size in sizes:
    for difficulty in difficulties:
        for puzzle_num in range(num_puzzles_per_config):
            current_test += 1
            progress = current_test / total_tests
            progress_bar.progress(progress)
            status_text.text(
                f"Testing {size}  {size} {difficulty} - "
```

```
                    f"Puzzle {puzzle_num + 1}/{num_puzzles_per_config}"
                )

                # Generate and test puzzle
                board = generate_puzzle(size, difficulty)
                results = test_all_algorithms(board, size)
                all_results.append(results)
```

Listing A.4: Progress Tracking Implementation

## A.7 Installation and Setup Guide

### A.7.1 Required Dependencies

```
# requirements.txt
streamlit==1.28.0
python-sat==0.1.8.dev0
pandas==2.0.3
matplotlib==3.7.2
seaborn==0.12.2
numpy==1.24.3
```

Listing A.5: Python Package Requirements

### A.7.2 MiniZinc Installation

MiniZinc installation steps for different platforms:

**Windows:**

1. Download MiniZinc from `https://www.minizinc.org/`
2. Run the installer
3. Add MiniZinc to system PATH

**macOS:**

```
brew install minizinc
```

**Linux:**

```
sudo apt-get install minizinc  # Ubuntu/Debian
sudo dnf install minizinc      # Fedora
```

## A.8 Sample Test Puzzles

Example puzzles used in testing, showing the board representation format:

```
# Hard 9 9  puzzle (60% empty)
puzzle_9x9_hard = [
    [0, 0, 0, 6, 0, 0, 4, 0, 0],
    [7, 0, 0, 0, 0, 3, 6, 0, 0],
    [0, 0, 0, 0, 9, 1, 0, 8, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 5, 0, 1, 8, 0, 0, 0, 3],
    [0, 0, 0, 3, 0, 6, 0, 4, 5],
    [0, 4, 0, 2, 0, 0, 0, 6, 0],
    [9, 0, 3, 0, 0, 0, 0, 0, 0],
    [0, 2, 0, 0, 0, 0, 1, 0, 0]
]

# Solution
solution_9x9_hard = [
    [5, 8, 1, 6, 7, 2, 4, 3, 9],
    [7, 9, 2, 8, 4, 3, 6, 5, 1],
    [3, 6, 4, 5, 9, 1, 7, 8, 2],
    [4, 3, 8, 9, 5, 7, 2, 1, 6],
    [2, 5, 6, 1, 8, 4, 9, 7, 3],
    [1, 7, 9, 3, 2, 6, 8, 4, 5],
    [8, 4, 5, 2, 3, 9, 1, 6, 7],
    [9, 1, 3, 7, 6, 8, 5, 2, 4],
    [6, 2, 7, 4, 1, 5, 3, 9, 8]
]
```

Listing A.6: Sample 9×9 Test Puzzle

## A.9 Visualization Configuration

Matplotlib configuration for publication-quality figures:

```
# Set publication-quality defaults
plt.style.use('seaborn-v0_8-whitegrid')
sns.set_palette("husl")

# Figure size and DPI settings
fig_width = 10  # inches
fig_height = 6  # inches
dpi = 300     # publication quality

# Font settings
plt.rcParams['font.size'] = 12
plt.rcParams['axes.titlesize'] = 16
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
plt.rcParams['legend.fontsize'] = 12

# Save figures with tight layout
plt.savefig('figure.png', dpi=dpi, bbox_inches='tight')
```

Listing A.7: Visualization Settings