Exploring Belief Propagation and Alternative Algorithms for Sudoku Solving

by

Harshit Suket Mutha

Bachelor Thesis in Computer Science

Submission: May 27, 2024

Supervisor: Dr. Fangning Hu

Constructor University | School of Computer Science and Engineering

Statutory Declaration

Family Name, Given/First Name	Mutha, Harshit
Matriculation number	30005096
Kind of thesis submitted	Bachelor Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

27/05/2024, Harshit Mutha

....

Date, Signature

Abstract

This thesis strives to create a user-friendly Sudoku solving program with belief propagation, a popular constraint satisfaction algorithm known for its effectiveness. Sudoku, a favorite puzzle game, is considered as an ideal place for applying belief propagation algorithm practically. In the beginning, many difficulties were encountered in implementing this algorithm especially with regard to the stopping sets which prevented it from completely solving sudoku puzzles.

In order to address these challenges, different algorithms were tried out and implemented. At first, a backtracking algorithm was created to build a basic solver. This straightforward method ensured that there was a reliable way to solve the problems. Then a constraint propagation algorithm was introduced which used constraint propagation for simplifying the complexity involved in solving puzzles. These undertakings yielded very useful information and helped in perfecting an efficient solver approach.

Afterward, the belief propagation algorithm was combined with the backtracking algorithm and modified, giving rise to a hybrid technique that was effective in solving Sudoku puzzles. This integration ensured that the puzzle solver dealt with intricacies of the game in a more efficient manner. This dissertation describes how these two algorithms were implemented, what problems were faced during their implementation process and what was learned from it. The thesis highlights the complexity of applying belief propagation on sudoku as well as stresses different methods should be tried so as come up with a strong solution.

Contents

1 Introduction

1.1 Background

Sudoku is a logic-based number placement puzzle that is often just as attractive to casual puzzle solvers as it is to algorithms. The classic Sudoku is a 9 × 9 grid broken into nine 3 × 3 subgrids. Every row, column, and subgrid must have each number from 1 to 9 without repeating any numbers. Though Sudoku rules are simple, the puzzles can be quite difficult to solve; therefore they fall under the areas of interest for computer science among others in artificial intelligence and combinatorial optimization.

The solving of Sudoku can be approached in various ways depending on whether it is being solved by humans or algorithms. Some common techniques used by people include heuristics based on experience through pattern recognition coupled with logical deductions while trying out different numbers sequentially until the correct one is found [1]. On the contrary, machines seek to completely automate this process thereby providing efficient solutions across all difficulty levels for any given puzzle instance. In fact, among these approaches lies belief propagation which has gained popularity due its grounding on probabilistic graphical models alongside successes achieved within related fields like error correcting codes systems design theory Bayesian networks analysis.

1.2 Problem Statement

The primary aim of this thesis was to develop a user-friendly Sudoku solver utilizing the belief propagation algorithm. Belief propagation is a message-passing algorithm [2] used for performing inference on graphical models, specifically factor graphs. The core idea is to approximate marginal probabilities by iteratively updating and passing messages between nodes in the graph. Given the structured constraints of Sudoku puzzles, belief propagation appeared to be a promising method for solving such problems efficiently.

1.3 Overview of Algorithms

In addition to belief propagation, this thesis explores and implements two other prominent algorithms for solving Sudoku puzzles:

1. **Constraint Propagation:** This method reduces the search space by iteratively applying constraints to eliminate possible values for each cell until a solution is found or no further progress can be made and can only solve sudokus which have only naked singles.
2. **Backtracking:** A classic algorithmic technique that uses depth-first search to explore possible solutions by recursively trying and eliminating values based on constraints until a solution is identified.

These algorithms were chosen to provide a comparative framework for evaluating the performance and effectiveness of belief propagation in solving Sudoku puzzles.

1.4 Difficulties

Many obstacles had to be overcome when implementing belief propagation for Sudoku solving. Firstly, the more loops a factor graph has, the more computationally complex belief propagation becomes, and this is a feature of the structure of the Sudoku puzzle [3].

It follows that achieving convergence and accurate marginal probabilities during practice sessions may therefore prove difficult.

Secondly, it was not easy to initialize the routine with proper messages or to ascertain their correctness after updating them. These are some of the things upon which the success of the algorithm largely depends since even a slight mistake in passing information could lead to wrong outputs being generated because such errors would be propagated or amplified along the way.

Again, Sudoku being a Constraint Satisfaction Problem (CSP) [4], some constraints may prematurely abort execution without arriving at any solution. Such conditions are said to create stopping sets which demonstrate how belief propagation cannot handle cyclic graphs that are common in many cases involving this type of puzzle.

In addition to that, my experience with working on different solvers which apply backtracks along constraints propagator besides belief propagator revealed that although each has its own merits and demerits when used singly or together they offer a platform from which best practices can be drawn towards solving sudoku puzzles using various algorithmic methods. This dissertation seeks to exhaustively examine these discoveries thereby adding to general knowledge concerning techniques employed in solving CSPs.

2 Statement and Motivation of Research

2.1 Research Question

The primary research question guiding this thesis is: **”Can belief propagation be effectively utilized to develop a user-friendly and efficient Sudoku solver, and how does it compare to traditional solving methods such as constraint propagation and backtracking?”**

This question drives the investigation into the application of belief propagation, a probabilistic graphical model algorithm, to the deterministic domain of Sudoku puzzles. The objective is to assess the feasibility, efficiency, and effectiveness of belief propagation in solving Sudoku, and to explore its advantages and limitations compared to more conventional algorithms.

2.2 Motivation

Sudoku puzzles, despite their seemingly simple structure, pose a significant computational challenge, making them an excellent testbed for evaluating algorithmic approaches to constraint satisfaction problems (CSPs) [5]. The motivation for this research includes:

1. **Educational Value:** Sudoku puzzles are widely understood and provide a clear, accessible example of CSPs, making them ideal for illustrating complex algorithms like belief propagation.

2. **Algorithmic Exploration:** While belief propagation has been applied successfully in fields such as error correction and Bayesian networks [6], its application to Sudoku represents an innovative exploration. Understanding how belief propagation performs in this context can provide insights into its applicability to other CSPs with similar structures.

3. **Comparative Analysis:** Implementing and comparing belief propagation with constraint propagation and backtracking aims to highlight the strengths and weaknesses of each approach. This comparative analysis can guide future research and practical applications in fields requiring efficient CSP solutions.

4. **Potential for Enhanced Solvers:** If belief propagation proves effective, it could lead to the development of more advanced and user-friendly Sudoku solvers with applications in educational tools, puzzle design, and broader CSP solutions.

2.3 Literature Review

The foundation of this research lies in the extensive body of work on Sudoku solving algorithms and probabilistic graphical models. Key studies include:

- **Belief Propagation:** Proposed by Judea Pearl in 1982, belief propagation was initially formulated as an exact inference algorithm [7] on trees and later extended to polytrees. It provides useful approximations even on general graphs and has applications ranging from decoding error-correcting codes to inference in Bayesian networks.

- **Constraint Propagation:** David Waltz initiated work on constraint propagation and consistency algorithms in 1975, introducing an arc-consistency algorithm [8] for three-dimensional cubic drawing interpretations, demonstrating the effectiveness of basic propagation algorithms for solving certain problems.

- **Backtracking Algorithms:** A fundamental approach to CSPs, backtracking has been extensively studied and optimized, using depth-first search to explore possible solutions and backtracking upon encountering conflicts.

While these studies provide a strong foundation, there is limited exploration of belief propagation specifically applied to Sudoku. This research seeks to fill this gap by providing a detailed implementation and analysis of belief propagation in this context.

2.4 Preliminary Experiments and Observation

Preliminary experiments with belief propagation revealed several challenges and insights:

1. **Initialization and Message Passing:** Initial attempts faced difficulties with message initialization and propagation, leading to convergence issues. Properly initializing messages and ensuring accurate updates are critical for the algorithm's success.
2. **Computational Complexity:** The complexity of belief propagation increased significantly with the number of loops in the Sudoku factor graph, making real-time solving challenging and highlighting the need for optimization or hybrid approaches.
3. **Stopping-Set Problem:** The algorithm often converged prematurely, failing to solve the entire puzzle, necessitating further investigation and potential integration with other solving methods.

Despite these challenges, the research provided valuable insights into applying belief propagation to Sudoku puzzles. These preliminary findings underscore the importance of a comprehensive comparative analysis with constraint propagation and backtracking.

2.5 Extending the State of the Art

This research seeks to extend the state of the art by:

1. **Refinement of Belief Propagation:** Addressing convergence issues and sensitivity to stopping sets to refine the belief propagation algorithm for better performance in Sudoku puzzles.
2. **Algorithmic Comparison:** Providing a detailed comparative analysis of belief propagation with backtracking and constraint propagation to contribute to the broader understanding of these methods in CSPs.
3. **Practical Applications:** Developing a robust and user-friendly Sudoku solver with practical implications for software development in recreational and educational contexts, demonstrating the practical utility of advanced algorithmic techniques.

2.6 Research Significance

This research extends the state of the art by exploring a novel application of belief propagation to Sudoku solving. By addressing computational challenges and comparing different algorithms, it aims to contribute to the broader field of CSP solutions. The findings can inform the development of more efficient and user-friendly solvers, with potential implications for education, entertainment, and various applications requiring CSP solutions.

In conclusion, this research aims to contribute to both the theoretical and practical aspects of algorithm development for Sudoku solvers. By exploring and refining belief propagation and comparing it with established methods, we hope to advance the understanding of constraint satisfaction problems and develop efficient, user-friendly solving techniques.

3 Requisite Knowledge and Skills

Developing the three Sudoku solving algorithms—Backtracking, Constraint Propagation, and Belief Propagation—required a diverse set of skills and a deep understanding of several programming concepts and tools. Here is an overview of the key areas of knowledge and skills that were necessary:

3.1 Initial Development with JavaScript

1. **Language Proficiency:** The project initially began in React JavaScript, which required a solid understanding of JavaScript, including its syntax, data structures, and asynchronous programming features.
2. **React Framework:** Knowledge of the React framework was crucial for building the initial user interface and managing the state of the application.

Although the initial development was in JavaScript, it became clear that the language did not meet the algorithmic needs of the project. This led to a strategic decision to transition to Python, which provided better support for the required numerical operations.

3.2 Python

1. **Language Proficiency:** The final implementation of the Sudoku solver was done in Python, necessitating a strong command of the language. This includes familiarity with Python's syntax, control structures, and standard libraries.
2. **Algorithm Implementation:** Implementing complex algorithms such as Backtracking, Constraint Propagation, and Belief Propagation required a deep understanding of Python's data structures and functions.

3.3 Libraries

1. **Tkinter:** Used for creating the graphical user interface (GUI) for the Sudoku solver, allowing for user interaction and display of the Sudoku grid.
2. **Numpy**
   - *Array Manipulations:* NumPy was extensively used for numerical computations, especially in the Belief Propagation algorithm. Proficiency in using NumPy for creating and manipulating arrays was critical.
   - *Linear Algebra Operations:* Many of the operations required for the algorithms involve linear algebra. NumPy provides a rich set of functions for performing these operations efficiently.
   - *Performance Optimization:* Leveraging NumPy's optimized functions helped in improving the performance of the algorithms, making the computations faster and more efficient.

3.4 Algorithm Knowledge

1. **Backtracking Algorithm:** Understanding the principles of recursive algorithms and depth-first search was essential for implementing the backtracking solver. This in-

cludes knowledge of how to systematically explore all possible solutions and backtrack when a solution path is not viable.

2. **Constraint Propagation:** Familiarity with constraint satisfaction problems and techniques for propagating constraints to reduce the search space was required. This involves iteratively narrowing down the possibilities for each cell until the puzzle is solved.

3. **Belief Propagation:** A more advanced algorithm, belief propagation required understanding probabilistic graphical models and message passing. This algorithm involves updating and normalizing probability distributions, which required a strong foundation in probability and statistics.

3.5 Additional Knowledge

1. **Problem-Solving Skills:** Developing these algorithms required strong analytical and problem-solving skills to devise efficient strategies for solving Sudoku puzzles.

2. **Debugging and Testing:** Proficiency in debugging and testing was crucial to ensure the correctness of the implementations. This includes writing test cases, using debugging tools, and systematically identifying and fixing issues in the code.

3. **Performance Analysis:** Evaluating the efficiency of the algorithms involved measuring the time complexity, number of iterations, and execution time. This required knowledge of algorithm analysis techniques and performance profiling tools.

In summary, the development of the Sudoku solver algorithms involved a comprehensive understanding of multiple programming languages and tools, with a strong emphasis on Python and NumPy. It also required deep algorithmic knowledge and problem-solving skills to implement and optimize the solving techniques effectively.

4 Description of the Investigation

4.1 Sudoku Puzzles for Testing

To ensure comprehensive evaluation of implemented Sudoku solvers, four different Sudoku puzzles with varying levels of difficulty were chosen: Default, Easy, Medium, and Hard. The Default puzzle acts as a control and is made to be a little more difficult than the Easy one but simpler than the Medium one. With this range of difficulty, it is possible to examine how well solvers perform at different complexities.

The selected puzzles are as follows:

- **Default Puzzle:** This puzzle is designed to provide a baseline difficulty that sits between the Easy and Medium levels. It serves as a control to benchmark the performance of the solvers.
- **Easy Puzzle:** This puzzle is relatively straightforward, providing minimal complexity and ensuring that even the simplest solver algorithms can resolve it effectively.
- **Medium Puzzle:** This puzzle introduces a moderate level of complexity, requiring more advanced strategies and algorithms for efficient solving.
- **Hard Puzzle:** This puzzle presents the highest level of difficulty among the four, challenging the solvers with intricate constraints and requiring sophisticated solving techniques.

These puzzles were consistently used across all implemented algorithms—belief propagation, backtracking, and constraint propagation—to ensure a fair comparison of their performance and efficacy. By evaluating each solver with the same set of puzzles, a clear understanding of the strengths and limitations of each approach was obtained.

Default Puzzle

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

Easy Puzzle

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

Medium Puzzle

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

Hard Puzzle

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

4.2 Backtracking Algorithm

The backtracking algorithm is a fundamental technique for solving constraint satisfaction problems (CSPs) like Sudoku. Let's delve deeper into its implementation using the provided code snippet. We'll break down the code step-by-step and explain each relevant part.

4.2.1 Initialization and GUI Setup

```python
class SudokuSolverGUI:
    def __init__(self, master):
        self.master = master
        self.master.title("Sudoku Solver")

        # Create a frame to hold the Sudoku grid
        self.grid_frame = tk.Frame(master)
        self.grid_frame.pack()
```

Listing 1: Initialization

- **Explanation:**
  - A '**SudokuSolverGUI**' class is defined, which initializes the GUI application.
  - The '**__init__**' method sets the window title to "Sudoku Solver".
  - A frame is created to hold the Sudoku grid and added to the main window using '**pack()**'.

```python
# Create a 9x9 grid of Entry widgets to represent the Sudoku puzzle
        self.cells = [[None]*9 for _ in range(9)]
        for i in range(9):
            for j in range(9):
                self.cells[i][j] = tk.Entry(self.grid_frame, width=3,
                ↪   font=('Helvetica', 16))
                self.cells[i][j].grid(row=i, column=j)
```

Listing 2: Creating Grid

- **Explanation:**
  - A 9x9 grid of ‘**Entry**‘ widgets is created to represent the Sudoku puzzle.
  - ‘**self.cells**‘ is a 2D list where each element is an ‘**Entry**‘ widget.
  - Nested loops iterate over rows (‘**Entry**‘) and columns (‘**j**‘) to place each ‘**Entry**‘ widget in the grid.

4.2.2 Loading Predefined Puzzles

```python
# Load buttons to load different predefined puzzles
        self.load_default_button = tk.Button(master, text="Load
        ↪   Default Puzzle", command=self.load_default_puzzle)
        self.load_default_button.pack()
        self.load_easy_button = tk.Button(master, text="Load Easy
        ↪   Puzzle", command=self.load_easy_puzzle)
        self.load_easy_button.pack()
        self.load_medium_button = tk.Button(master, text="Load Medium
        ↪   Puzzle", command=self.load_medium_puzzle)
        self.load_medium_button.pack()
        self.load_hard_button = tk.Button(master, text="Load Hard
        ↪   Puzzle", command=self.load_hard_puzzle)
        self.load_hard_button.pack()
```

Listing 3: Buttons for Loading Puzzles

- **Explanation:**
  - Four buttons are created to load different predefined puzzles: default, easy, medium, and hard.
  - Each button is linked to a method (‘**load_default_puzzle**‘, ‘**load_easy_puzzle**‘, etc.) that loads a specific puzzle into the grid.

```python
# Function to load a puzzle into the Sudoku grid
    def load_puzzle(self, puzzle):
        for i in range(9):
            for j in range(9):
                self.cells[i][j].delete(0, tk.END)
                if puzzle[i][j] != 0:
                    self.cells[i][j].insert(0, puzzle[i][j])
```

Listing 4: Loading a Puzzle into the Grid

- **Explanation:**
  - The ‘**load_puzzle**‘ method takes a 2D list ‘**puzzle**‘ as input.
  - It iterates through each cell in the grid, clears the current content, and inserts the corresponding value from the ‘**puzzle**‘ list.
  - If the value is ‘**0**‘, the cell is left empty.

4.2.3 Solving the Puzzle

```
# Function to solve the Sudoku puzzle
    def solve_sudoku(self):
        # Extract the Sudoku puzzle from the grid
        puzzle = [[0]*9 for _ in range(9)]
        for i in range(9):
            for j in range(9):
                try:
                    value = self.cells[i][j].get().strip()
                    if value:
                        value = int(value)
                        if value < 1 or value > 9:
                            raise ValueError
                        puzzle[i][j] = value
                except ValueError:
                    messagebox.showerror("Error", "Invalid input in
                    ↪ cell ({}, {})".format(i+1, j+1))
                    return

        # Solve the Sudoku puzzle
        start_time = time.time()  # Start timing
        self.iteration_count = 0  # Initialize iteration count
        if self.solve(puzzle):
            end_time = time.time()  # End timing
            total_time = end_time - start_time  # Calculate the time
            ↪ taken
            messagebox.showinfo("Info", f"Sudoku Solved!\nTime taken:
            ↪ {total_time:.2f} seconds\nIterations:
            ↪ {self.iteration_count}")
        else:
            messagebox.showinfo("Info", "No solution found")
```

Listing 5: Solving the Sudoku Puzzle

- **Explanation:**
  - The '**solve_sudoku**' method starts by extracting the current '**puzzle**' from the grid into a 2D list puzzle.
  - It validates the input values, ensuring they are integers between 1 and 9.
  - If the input is invalid, an error message is displayed, and the solving process is aborted.
  - The solving process is timed, and the '**solve**' method is called to solve the puzzle.
  - If solved, the time taken and the number of iterations are displayed in a message box.

```python
# Function to find the next empty cell in the Sudoku puzzle
    def find_empty_cell(self, puzzle):
        for i in range(9):
            for j in range(9):
                if puzzle[i][j] == 0:
                    return (i, j)
        return None
```

Listing 6: Finding the Next Empty Cell

- **Explanation:**
  - The ‘**find_empty_cell**‘ method scans the puzzle for the next empty cell (value 0).
  - It returns the coordinates of the empty cell as a tuple ‘**(i, j)**‘.
  - If no empty cells are found, it returns ‘**None**‘.

```python
# Function to check if a move is valid in the Sudoku puzzle
    def is_valid_move(self, puzzle, row, col, num):
        if num in puzzle[row]:
            return False

        if num in [puzzle[i][col] for i in range(9)]:
            return False

        start_row, start_col = 3 * (row // 3), 3 * (col // 3)
        for i in range(start_row, start_row + 3):
            for j in range(start_col, start_col + 3):
                if puzzle[i][j] == num:
                    return False
        return True
```

Listing 7: Checking Validity of a Move

- **Explanation:**
  - The ‘**is_valid_move**‘ method checks if placing a number ‘**num**‘ in the cell at ‘**(row, col)**‘ is valid according to Sudoku rules.
  - It checks the row, column, and the 3x3 subgrid for the presence of ‘**num**‘.
  - If ‘**num**‘ is found in any of these, it returns ‘**False**‘; otherwise, it returns ‘**True**‘.

```python
# Recursive function to solve the Sudoku puzzle
   def solve(self, puzzle):
       self.iteration_count += 1  # Increment iteration count
       empty_cell = self.find_empty_cell(puzzle)
       if not empty_cell:
           return True  # Puzzle solved

       row, col = empty_cell
       for num in range(1, 10):
           if self.is_valid_move(puzzle, row, col, num):
               puzzle[row][col] = num
               self.cells[row][col].delete(0, tk.END)
               self.cells[row][col].insert(0, num)
               self.cells[row][col].update_idletasks()
               if self.solve(puzzle):
                   return True
               puzzle[row][col] = 0
               self.cells[row][col].delete(0, tk.END)
               self.cells[row][col].insert(0, "")
               self.cells[row][col].update_idletasks()
       return False
```

Listing 8: Recursive Backtracking Solver

- **Explanation:**
  - The ‘**solve**‘ method implements the backtracking algorithm.
  - It first increments the ‘**iteration_count**‘.
  - It finds the next empty cell using ‘**find_empty_cell**‘.
  - If no empty cells are left, the puzzle is solved, and the method returns ‘**True**‘.
  - It iterates through numbers 1 to 9, checking if placing each number in the empty cell is valid using ‘**is_valid_move**‘.
  - If a valid move is found, the number is placed in the cell, and the GUI is updated.
  - The method recursively calls itself to solve the updated puzzle.
  - If the recursive call returns ‘**True**‘, the puzzle is solved.
  - If no valid number can be placed in the cell, the method backtracks by resetting the cell to empty and tries the next number.
  - If no solution is found after trying all numbers, the method returns ‘**False**‘.

4.2.4 Running the Application

```python
def main():
    root = tk.Tk()
    app = SudokuSolverGUI(root)
    root.mainloop()

if __name__ == "__main__":
    main()
```

Listing 9: Main Function

- **Explanation:**
  - The ‘**main**‘ function creates the main window (‘**root**‘) and an instance of ‘**SudokuSolverGUI**‘.
  - The main event loop is started with ‘**root.mainloop()**‘, which keeps the application running and responsive to user input.

This detailed breakdown of the backtracking algorithm in the context of a Sudoku solver GUI illustrates how each part of the code contributes to the overall functionality. The integration of the backtracking approach with the Tkinter GUI allows for interactive problem-solving and visualization.

This implementation highlights the elegance of backtracking in solving constraint satisfaction problems efficiently. However, it also underscores the computational complexity associated with exhaustive search methods, especially for challenging Sudoku instances.

4.3 Constraint Propagation Algorithm

The constraint propagation algorithm is meant to continuously eliminate the potential values for each empty cell in a Sudoku puzzle by looking at the numbers present in its row, column or box [9]. It’s most effective in cases of solving puzzles with naked singles—those cells that can be directly filled when they have only one possible value. Now let’s take an overview on how this method works together with its implementation details as well as the main ideas behind such an approach.

4.3.1 Initialization

During the initialization phase, each empty cell is assigned a binary matrix representing the possibilities for that cell. Initially, all possibilities are allowed for empty cells.

```python
import time

class SudokuSolver:
    def __init__(self, puzzle):
        self.grid = puzzle
        self.possibilities = self.initialize_possibilities()
```

Listing 10: Creating the SudokuSolver Class and Initializing the Grid

- **Explanation:**
  - The ‘**SudokuSolver**‘ class is defined, and the “__**init**__ method initializes the solver with the given puzzle.
  - The ‘**initialize_possibilities**‘ method is called to set up the initial possibilities for each cell.

```python
def initialize_possibilities(self):
        possibilities = [[[1]*9 for _ in range(9)] for _ in range(9)]
        for i in range(9):
            for j in range(9):
                if self.grid[i][j] != 0:
                    possibilities[i][j] = [0]*9
                    possibilities[i][j][self.grid[i][j] - 1] = 1
        return possibilities
```

Listing 11: Initializing Possibilities

- **Explanation:**
  - A 3D list ‘**possibilities**‘ is created where each cell initially allows all numbers (1-9).
  - For cells already filled with a number, the possibilities matrix is updated to reflect the provided number.

    Considering the default sudoku puzzle:

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

    For example, consider a cell at position ‘**(1, 1)**‘ initially:

```
Cell (1, 1): [0, 0, 0, 0, 1, 0, 0, 0, 0]  # Cell is filled with
↪  5
```

    Whereas for cell at position ‘**(1, 3)**‘:

```
Cell (1, 3): [1, 1, 1, 1, 1, 1, 1, 1, 1]  # All possibilities are
↪  allowed
```

4.3.2 Propagating Constraints

Constraint propagation involves updating the possibilities matrix for each empty cell based on existing numbers in its row, column, and box.

```python
def propagate_constraints(self):
    for i in range(9):
      for j in range(9):
        if self.grid[i][j] == 0:
          for num in range(1, 10):
            if num in self.get_row(i) or num in self.get_col(j) or num in self.get_box(i, j):
              self.possibilities[i][j][num - 1] = 0
```

Listing 12: Propagating Constraints

- **Explanation:**
  - For each empty cell, the algorithm checks if placing a number from 1 to 9 violates constraints in its row, column, or box.
  - If a number is present in the row, column, or box, it marks the corresponding possibility as unavailable in the cell's matrix.
  - Example: After constraint propagation, the possibilities matrix for an empty cell at position (1, 3) will be updated as:

    ```
    Cell (1, 3): [1, 1, 0, 1, 0, 0, 0, 0, 0]  # Possibilities 3, 5, 6, 7, 8, and 9 are now unavailable
    ```

    Possibilities 3, 5, 6, 7, 8, and 9 are marked unavailable because they exist in the same row, column, or box.

```python
def get_row(self, i):
  return [self.grid[i][j] for j in range(9)]

def get_col(self, j):
  return [self.grid[i][j] for i in range(9)]

def get_box(self, i, j):
  box_i, box_j = (i // 3) * 3, (j // 3) * 3
  return [self.grid[box_i + x][box_j + y] for x in range(3) for y in range(3)]
```

Listing 13: Getting Row, Column, and Box Values

- **Explanation:**
  - **‘get_row(i)‘** returns the values in the specified row **‘i‘**.
  - **‘get_col(j)‘** returns the values in the specified column **‘j‘**.

- **'get_box(i, j)'** returns the values in the 3x3 box containing the specified cell **'(i, j)'**.

4.3.3 Solving the Puzzle

```
def is_solved(self):
        return all(0 not in row for row in self.grid)
```

Listing 14: Getting Row, Column, and Box Values

- **Explanation:**
  - Checks if the Sudoku grid is completely filled by ensuring no row contains a **'0'**.

```python
def solve(self):
        start_time = time.time()  # Start timing
        unsolved_sudoku = [[self.grid[i][j] for j in range(9)] for i
        ↪  in range(9)]
        print("Unsolved Sudoku:")
        self.print_grid(unsolved_sudoku)

        print("Initialization:")
        self.print_possibilities()  # Print initial possibilities
        ↪  matrix

        iteration = 1
        while not self.is_solved():
            previous_grid = [row[:] for row in self.grid]
            self.propagate_constraints()
            self.fill_single_possibility()

            if previous_grid == self.grid:
                print(f"Stuck at iteration: {iteration}")
                break

            print(f"Iteration: {iteration}")
            self.print_possibilities()
            iteration += 1

        end_time = time.time()
        total_time = end_time - start_time

        if self.is_solved():
            solved_sudoku = [[self.grid[i][j] for j in range(9)] for
            ↪  i in range(9)]
            print("\nSolved Sudoku:")
            self.print_grid(solved_sudoku)
        else:
            print("\nUnable to solve Sudoku with current strategy.")

        print(f"Time taken: {total_time:.2f} seconds")
        print(f"Iterations: {iteration - 1}")
```

Listing 15: Getting Row, Column, and Box Values

- **Explanation:**
  - The '**solve**' method solves the Sudoku puzzle using constraint propagation.
  - It starts by printing the unsolved Sudoku and initial possibilities matrix.
  - In each iteration, it propagates constraints and fills cells with a single possibility.
  - If no progress is made in an iteration, the loop breaks.

  - The total time and number of iterations are printed, and the solved Sudoku is displayed if the puzzle is solved.

```
def fill_single_possibility(self):
        for i in range(9):
            for j in range(9):
                if self.grid[i][j] == 0 and
                 ↪  sum(self.possibilities[i][j]) == 1:
                    self.grid[i][j] =
                     ↪  self.possibilities[i][j].index(1) + 1
```

Listing 16: Filling Cells with Single Possibility

- **Explanation:**
  - This method fills cells where only one possibility exists.
  - If a cell is empty and has only one possible value, it is filled with that value.

4.3.4 Printing the Grid and Possibilities

```
def print_possibilities(self):
        for i in range(9):
            for j in range(9):
                print(f"Cell ({i}, {j}): {self.possibilities[i][j]}")
        print()
```

Listing 17: Filling Cells with Single Possibility

- **Explanation:**
  - Prints the possibilities matrix for each cell in the Sudoku grid.

The algorithm later prints the solved sudoku grid, time taken to solve the particular puzzle, and number of iteration before it got stuck due to its in-capabilities in case it did. By iteratively updating the possibilities matrix, the constraint propagation algorithm systematically reduces the possibilities for each empty cell in the Sudoku puzzle, efficiently progressing towards a solution.

4.4 Belief Propagation Algorithm

Belief propagation is a probabilistic method [10] for solving various kinds of constraint satisfaction problems, such as Sudoku puzzles. Unlike constraint propagation, which removes impossible values from each cell directly, belief propagation instead iteratively updates the probability (or “belief”) of each number being in every cell through a message-passing framework. In this section, we will describe how to implement belief propagation for Sudoku and give examples illustrating each step.

However, it should be noted that pure belief propagation alone cannot solve all Sudoku puzzles [11]. The algorithm presented here uses belief propagation to calculate probabilities and handle simple cases (naked singles). When belief propagation does not converge on more difficult puzzles, a backtracking algorithm is applied instead. The latter makes use of probabilities obtained by belief propagation to improve its efficiency.

4.4.1 Initialization

During initialization, each cell is assigned a belief vector representing the probabilities of each number (1-9) being in that cell. Initially, all possibilities are allowed for empty cells.

Here's the initialization process:

```python
def solve(self, puzzle):
    beliefs = np.ones((9, 9, 9))  # Initial belief: all numbers
    ↪ equally likely
    for i in range(9):
        for j in range(9):
            if puzzle[i][j] != 0:
                beliefs[i, j, :] = 0
                beliefs[i, j, int(puzzle[i][j]) - 1] = 1
```

Listing 18: Initialization Process for Belief Propagation

In this code:

- **'beliefs'** is a 3D array where **'beliefs[i, j, k]'** represents the probability of number **'k+1'** being in cell **'(i, j)'**.
- If a cell is already filled with a number, its corresponding belief vector is set to indicate certainty for that number and zero probability for others.

For example, consider the cell at position **'(2, 1)'** in the default puzzle which is filled with the number 6:

```
Cell (2, 1): [0, 0, 0, 0, 0, 1, 0, 0, 0]  # Certainty that the number is 6
```

For an empty cell at position (1, 3) initially:

```
Cell (1, 3): [1, 1, 1, 1, 1, 1, 1, 1, 1]  # All numbers are equally likely
```

4.4.2 Propagating Messages

The **'propagate_messages'** method updates messages based on the current beliefs. Messages are propagated to reflect constraints from the row, column, and box of each cell.

```python
def propagate_messages(self, beliefs):
    messages = np.ones((9, 9, 9, 4)) / 9  # Initialize messages

    # Iterate over cells
    for i in range(9):
        for j in range(9):
            if np.any(beliefs[i, j] > 1e-10):  # Check if the value
            ↪  is close to zero
                value = np.argmax(beliefs[i, j])
                messages[i, j, :, :] = 0
                messages[i, j, value, :] = 1
            else:
                # Update messages to reflect constraints
                row_values = beliefs[i, :].copy()
                row_values[j] = 0
                col_values = beliefs[:, j].copy()
                col_values[i] = 0
                box_values = beliefs[(i // 3) * 3:(i // 3 + 1) * 3,
                ↪  (j // 3) * 3:(j // 3 + 1) * 3].flatten()
                box_values[(i % 3) * 3 + (j % 3)] = 0

                # Normalize messages if not all zeros
                if row_values.sum() > 0:
                    row_values /= row_values.sum()
                if col_values.sum() > 0:
                    col_values /= col_values.sum()
                if box_values.sum() > 0:
                    box_values /= box_values.sum()

                # Set messages
                messages[i, j, :, 0] = row_values
                messages[i, j, :, 1] = col_values
                messages[i, j, :, 2] = box_values

    return messages
```

Listing 19: propagate_messages

In this method:

- ‘**messages**‘ is a 4D array where ‘**messages[i, j, k, d]**‘ represents the message about number ‘**k+1**‘ for cell ‘**(i, j)**‘ from direction ‘**d**‘ (0: row, 1: column, 2: box).
- If a cell already has a determined value, the messages are set to reflect that certainty.
- For empty cells, messages are updated to reflect constraints from row, column, and box.
- Messages are normalized to ensure they sum to 1.

Explanation of ‘argmax‘:
The function **‘np.argmax‘** is used to find the index of the maximum value in an array. In this context, **‘np.argmax(beliefs[i, j])‘** returns the index of the highest probability (belief) for cell (i, j). This indicates the number most likely to be in that cell based on current beliefs. For example, if **‘beliefs[i, j] = [0.1, 0.3, 0.05, 0.05, 0.1, 0.1, 0.05, 0.1, 0.15]‘**, **‘np.argmax(beliefs[i, j])‘** would return **‘1‘** because the second element (index 1) has the highest value (0.3).

Normalization:
Normalization involves dividing each probability by the sum of all probabilities in that message vector to ensure the sum equals 1. This step ensures that the messages remain valid probability distributions. For example, if the initial row messages for a cell are **‘[0.2, 0.3, 0.5]‘** (summing to 1), and updating based on constraints results in **‘[0.2, 0.1, 0.2]‘** (summing to 0.5), normalization would adjust it to **‘[0.4, 0.2, 0.4]‘**.

For example, consider the cell at position **‘(1, 3)‘** after propagation:

```
Row messages: [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]  # Reflects
↪  equal distribution
Column messages: [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]  # Reflects
↪  equal distribution
Box messages: [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]  # Reflects
↪  equal distribution
```

4.4.3 Updating Beliefs

The **‘update_beliefs‘** method updates the beliefs based on the received messages.

```python
def update_beliefs(self, beliefs, messages):
    updated_beliefs = beliefs.copy()

    # Iterate over cells
    for i in range(9):
        for j in range(9):
            # If cell has a value, skip
            if not np.any(np.isclose(beliefs[i, j], 0)):
                continue

            # Update beliefs based on received messages
            belief_product = np.prod(messages[i, j], axis=1)
            updated_beliefs[i, j] *= belief_product

            # Normalize beliefs if not all zeros
            if np.sum(updated_beliefs[i, j]) > 0:
                updated_beliefs[i, j] /= np.sum(updated_beliefs[i,
                ↪  j])

    return updated_beliefs
```

Listing 20: update_beliefs

In this method:

- **'update_beliefs'** is a copy of the current beliefs.
- For each cell, the beliefs are updated by multiplying with the product of the received messages from row, column, and box constraints. This step combines the constraints from these three sources to refine the belief for each number in the cell.
- Beliefs are normalized to ensure they sum to 1.

For example, consider an empty cell at position **'(1, 3)'** with the initial beliefs:

```
Initial beliefs: [1, 1, 1, 1, 1, 1, 1, 1, 1]  # All numbers are equally
↪  likely
```

After propagating messages and updating beliefs, suppose the row, column, and box messages for number 5 are lower due to constraints:

```
Updated row messages: [0.1, 0.1, 0.1, 0.1, 0.05, 0.1, 0.1, 0.1, 0.1]
Updated column messages: [0.1, 0.1, 0.1, 0.1, 0.05, 0.1, 0.1, 0.1, 0.1]
Updated box messages: [0.1, 0.1, 0.1, 0.1, 0.05, 0.1, 0.1, 0.1, 0.1]
```

The belief product for number 5 would be lower, leading to:

```
Beliefs after update: [0.1 * 0.1 * 0.1, ..., 0.05 * 0.05 * 0.05, ...]
```

Normalization ensures these beliefs sum to 1:

```
Normalized beliefs: [0.1, 0.1, 0.2, 0.2, 0.1, 0.2, 0.2, 0.2, 0.2]  #
↪  Reflects changes based on constraints
```

Why Calculate the Belief Product?
The belief product integrates the influences from the row, column, and box constraints into a single updated belief vector for each cell. By multiplying the messages, we combine the information from all three sources to refine the probability distribution for each number in the cell. This step ensures that the updated beliefs reflect all constraints, leading to more accurate probability estimates.

4.4.4 Solving the Sudoku

The belief propagation process iterates until the beliefs converge or the maximum number of iterations is reached.

```python
def solve(self, puzzle):
    beliefs = np.ones((9, 9, 9))
    for i in range(9):
        for j in range(9):
            if puzzle[i][j] != 0:
                beliefs[i, j, :] = 0
                beliefs[i, j, int(puzzle[i][j]) - 1] = 1

    for _ in range(10):  # Limit iterations to avoid infinite loop
        messages = self.propagate_messages(beliefs)
        beliefs = self.update_beliefs(beliefs, messages)
        if np.all(np.sum(beliefs, axis=2) == 1):  # Check for
        ↪   convergence
            break

    if np.any(np.sum(beliefs, axis=2) != 1):
        # Belief propagation did not converge to a solution, use
        ↪   backtracking
        solution = self.backtrack(puzzle, beliefs)
        return solution
    else:
        solution = np.argmax(beliefs, axis=2) + 1
        return solution
```

Listing 21: Solving Sudoku

In this code:

1. **Initialization:**
    - The beliefs for each cell are initialized to represent equal probabilities for all numbers (1-9).
    - If a cell already contains a number, the belief for that number is set to 1, and the beliefs for all other numbers are set to 0.
2. **Iterative Belief Propagation:**
    - The algorithm iterates up to a maximum of 10 times. In each iteration:
        - Messages are propagated based on current beliefs using **'propagate_messages'**.
        - Beliefs are updated based on the propagated messages using **'update_beliefs'**.
        - Convergence is checked by verifying if the sum of probabilities for each cell equals 1.
    - If convergence is achieved, the loop breaks early.

3. **Convergence Check and Backtracking:**
    - If belief propagation does not converge within the set number of iterations, the backtracking algorithm is invoked.
    - The backtracking function leverages the computed probabilities to guide its search, making it more efficient than traditional backtracking.
4. **Return Solution:**
    - If belief propagation converges, the solution is determined by selecting the number with the highest probability for each cell.
    - If backtracking is used, it returns the solved puzzle from the backtracking algorithm.

4.4.5 Backtracking Algorithm

The backtracking algorithm is modified to use probabilities from belief propagation to guide its search, enhancing efficiency.

```python
def backtrack(self, puzzle, beliefs):
    empty_cells = [(i, j) for i in range(9) for j in range(9) if
    ↪   puzzle[i][j] == 0]

    def backtrack_solve(index):
        if index == len(empty_cells):
            return True

        i, j = empty_cells[index]
        probabilities = beliefs[i, j]
        numbers = np.argsort(probabilities)[::-1] + 1  # Sort numbers
        ↪  by their probabilities

        for num in numbers:
            if self.is_valid(puzzle, i, j, num):
                puzzle[i][j] = num
                if backtrack_solve(index + 1):
                    return True
                puzzle[i][j] = 0

        return False

    if backtrack_solve(0):
        return puzzle
    else:
        return None
```

Listing 22: Solving Sudoku

In this code:

1. **Identify Empty Cells:**
    - Collects the coordinates of all empty cells in the puzzle.
2. **Validity Check Function:**
    - Defines **‘is_valid‘** to check if placing a number in a specific cell violates Sudoku rules (row, column, and box constraints).
3. **Backtracking Search Function:**
    - Defines **‘solve_backtrack‘** to recursively attempt to fill the puzzle:
        - If all empty cells are filled (base case), the puzzle is solved.
        - For each empty cell, retrieves possible values sorted by descending probabilities from beliefs.
        - Attempts to place each possible value in the cell, checks its validity, and proceeds recursively.
        - If placing a value leads to a dead end, undoes the move (backtracks) and tries the next value.

Why This is Different from Traditional Backtracking?

- Traditional backtracking tries all possible values in a fixed or arbitrary order, leading to inefficiency.
- This modified backtracking uses belief propagation probabilities to prioritize more likely values, reducing wrong guesses and backtracking steps, leading to a more efficient search.

4.4.6 Conclusion

This Sudoku solver uses the belief propagation’s probabilistic framework to solve easy cases and narrow possibilities, making the following backtracking algorithm more efficient. By this method, we combine belief propagation’s probabilistic guidance with backtracking to ensure a solution while balancing between probabilistic and deterministic methods. We limit the number of iterations for belief propagation so that it doesn’t become infinite. Unlike conventional methods, we optimize our search in the backtracking stage using probability estimations.

5 Evaluation of the Investigation

In this section, we evaluate the effectiveness and performance of three different algorithms for solving Sudoku puzzles: backtracking, constraint propagation, and belief propagation. Each of these algorithms has its own strengths and weaknesses, which are discussed in the context of their implementation, efficiency, and suitability for solving various difficulty levels of Sudoku puzzles.

5.1 Backtracking Algorithm

The Sudoku puzzle-solving program employs the backtracking technique, generating and evaluating potential solutions using a trial and error approach till either the correct answer is located or all options have been depleted. Although uncomplicated, the method may be time-consuming especially when dealing with more complex puzzles.

5.1.1 Code Analysis

The backtracking solver is implemented within a graphical user interface (GUI) using Tkinter. The code initializes a 9x9 grid of Entry widgets for user input and includes buttons to load predefined puzzles of varying difficulty levels. The solving process is initiated by a button press, which extracts the current state of the grid and uses a recursive function to find a solution.

Key features of the backtracking code:

- **Initializaton:** The GUI setup is user-friendly, allowing users to input their own puzzles or load predefined ones.
- **Validation:** The solver checks for valid inputs and ensures that each value placed in a cell adheres to Sudoku rules.
- **Iteration and Timing:** The code measures the time taken to solve the puzzle and counts the number of iterations required.

Performance
This traditional method guarantees a solution for any Sudoku puzzle, regardless of complexity. On further experiment the data shows that for the Default puzzle, it required 4209 iterations and took an average of 2.83 seconds to solve. The Easy puzzle was solved much more efficiently, with only 155 iterations in an average time of 0.09 seconds. For the Medium puzzle, the iteration count rose significantly to 2842, with a corresponding solve time of 1.94 seconds. The Hard puzzle presented the greatest challenge, necessitating 45671 iterations and taking an average of 36.52 seconds to find the solution. This demonstrates that while Backtracking is reliable in solving any puzzle, its performance diminishes considerably with increasing puzzle difficulty, resulting in longer solve times and higher iteration counts.

5.2 Constraint Propagation Algorithm

The constraint propagation algorithm reduces the search space by eliminating possibilities that violate Sudoku rules based on the current state of the grid. This method is more sophisticated than simple backtracking, as it incorporates logical inference to prune the search space.

5.2.1 Code Analysis

The constraint propagation solver initializes the possibilities for each cell based on the given puzzle and iteratively applies constraints to reduce these possibilities. The main components of the code include:

- **Possibility Initialization:** Sets up a 3D list to store possible values for each cell.
- **Constraint Propagation:** Iterates through the grid to remove impossible values from the possibilities list based on the current state of the grid.
- **Single Possibility Filling:** Automatically fills cells where only one possibility remains.

Key features of the constraint propagation code:

- **Efficiency:** By systematically eliminating impossible values, the algorithm reduces the need for extensive searching.
- **Iteration:** The code prints the state of possibilities at each iteration, providing insight into the solving process.
- **Stuck Detection:** If the algorithm cannot make further progress, it stops to prevent infinite loops.

Performance
Constraint Propagation algorithm showed remarkable efficiency for simpler puzzles but struggled with more complex ones. For the Default puzzle, it required only 10 iterations and took an average of 0.09 seconds to solve. The Easy puzzle was even faster, needing just 5 iterations and 0.04 seconds. However, its performance declined sharply with the Medium and Hard puzzles. The Medium puzzle was partially processed in 2 iterations, taking 0.02 seconds, but the algorithm got stuck and failed to solve it. Similarly, the Hard puzzle saw no progress, with 0 iterations in 0.01 seconds, indicating that Constraint Propagation was unable to handle the increased complexity. This algorithm excels in speed for puzzles it can solve but fails with medium and hard difficulties. This is because the algorithm is designed to solve naked singles only.

5.3 Belief Propagation Algorithm

Belief propagation is a probabilistic inference technique used in graphical models and is adapted here for solving Sudoku puzzles. This approach iteratively updates beliefs (probabilities) about the values of cells based on the constraints imposed by the puzzle, however it falls back to efficient backtracking if it fails to converge.

5.3.1 Code Analysis

The belief propagation solver uses a GUI similar to the backtracking solver for user interaction. The main components of the code include:

- **Belief Initialization:** Sets up initial probabilities for each cell based on the given puzzle.
- **Message Passing:** Iteratively updates the probabilities based on the constraints from neighboring cells.

- **Decision Making:** Determines the most probable value for each cell based on the updated beliefs.

Key features of the constraint propagation code:

- **Probabilistic Approach:** Uses probabilities to guide the solving process, making it potentially more flexible in handling uncertainty.
- **GUI Integration:** Allows users to input and solve puzzles through a graphical interface.
- **Performance Monitoring:** Tracks the time taken and iterations required to solve the puzzle.

Performance
Belief Propagation algorithm, which includes a fallback to Backtracking if convergence is not achieved, emerged as the most balanced and effective approach. It processed all puzzles in a fixed number of iterations (10), which was set as a threshold before invoking backtracking if necessary. For the Default puzzle, it took an average of 0.03 seconds, and the Easy puzzle was solved in the same time frame. The Medium puzzle took slightly longer, averaging 0.09 seconds, while the Hard puzzle required 4.67 seconds on average. The use of belief propagation for initial iterations and switching to backtracking only if convergence was not reached allowed this algorithm to combine the speed of belief propagation with the reliability of backtracking.

5.4 Additional Comparison Factors

In addition to the factors mentioned above, we can also consider other aspects for comparison, such as:

- **Robustness:** How well does each algorithm handle variations in Sudoku puzzles, including different levels of complexity and patterns?
- **Resource Utilization:** How efficiently does each algorithm utilize computational resources, including memory and processing power?
- **Scalability** How well does each algorithm scale with larger Sudoku grids or more complex puzzles?

By considering these additional factors, we can gain a more comprehensive understanding of the strengths and weaknesses of each algorithm. However, it's important to note that the performance of each algorithm may vary depending on the specific implementation and the characteristics of the Sudoku puzzles being solved.

5.5 Summary

To sum up, though guaranteed any Sudoku puzzle could be solved by the backtracking algorithm, they are slow on difficult puzzles. The constraint propagation algorithm is very fast and efficient on easy puzzles but cannot handle more complicated ones whereas the belief propagation algorithm offers the best overall performance with backtracking fallback. The hybrid approach of belief propagation algorithm plus backtracking fallback—makes it possible to solve wide ranges in difficulty level for puzzles by using both strengths from these techniques. But in comparison among these three analyzed methods this one is superior because it can quickly deal with simpler cases while keeping them reliable enough for harder ones too.

Table 1: Performance Comparison of Sudoku Solving Algorithms

Algorithm	**Puzzle Difficulty**	**Iterations**	**Average Time (seconds)**
Backtracking	Default	4209	2.83
	Easy	155	0.09
	Medium	2842	1.94
	Hard	45671	36.52
Constraint Propagation	Default	10	0.09
	Easy	5	0.04
	Medium	2	0.02
	Hard	0	0.01
Belief Propagation	Default	10	0.03
	Easy	10	0.03
	Medium	10	0.09
	Hard	10	4.67

6 Conclusions

Based on our experience and the analysis of three different ways to solve sudoku (Backtracking, Constraints Propagation, and Belief Propagation), we can make several conclusions.

Firstly, Backtracking proves itself as a reliable method for solving sudoku puzzles of any difficulty because it always finds a solution. Nevertheless, it takes too much time to solve hard puzzles than easy ones.

Secondly, Constraints Propagation copes well with simple puzzles due to the quick elimination of naked singles. At the same time, it cannot solve medium and hard puzzles which implies that this approach has some limitations when dealing with more complicated configurations of numbers in sudoku.

Finally, Belief Propagation seems to be the best general-purpose algorithm. It guarantees the answer to a puzzle and is faster for medium and hard sudokus if compared with ordinary backtracking only methods. The reason is that it integrates backtracking by beliefs gathered during propagation.

To sum up, each algorithm has its own strong and weak points, but Belief Propagation appears to be the most flexible and effective among them. Furthermore, further improvements may be applied into the technique making it more powerful tool not only for sudoku fans but also for computer scientists working on algorithms development in general.

References

[1] Radek Pelánek. “Human problem solving: Sudoku case study”. In: *Faculty of Informatics, Masaryk University* (2011).

[2] Jonathan S Yedidia, William Freeman, and Yair Weiss. “Generalized belief propagation”. In: *Advances in neural information processing systems* 13 (2000).

[3] Kevin Murphy, Yair Weiss, and Michael I Jordan. “Loopy belief propagation for approximate inference: An empirical study”. In: *arXiv preprint arXiv:1301.6725* (2013).

[4] Todd K Moon and Jacob H Gunther. “Multiple constraint satisfaction by belief propagation: An example using sudoku”. In: *2006 IEEE mountain workshop on adaptive and learning systems*. IEEE. 2006, pp. 122–126.

[5] Helmut Simonis. “Sudoku as a constraint problem”. In: *CP Workshop on modeling and reformulating Constraint Satisfaction Problems*. Vol. 12. Citeseer. 2005, pp. 13–27.

[6] Christopher Fogelberg, Vasile Palade, and Phil Assheton. “Belief propagation in fuzzy bayesian networks”. In: *1st International Workshop on Combinations of Intelligent Methods and Applications (CIMA) at ECAI’08*. 2008, pp. 19–24.

[7] Kristian Kersting, Babak Ahmadi, and Sriraam Natarajan. “Counting belief propagation”. In: *arXiv preprint arXiv:1205.2637* (2012).

[8] Martin Cooper and Thomas Schiex. “Arc consistency for soft constraints”. In: *Artificial Intelligence* 154.1-2 (2004), pp. 199–227.

[9] Christopher G Reeson. “Using Constraint Processing to Model, Solve, and Support Interactive Solving of Sudoku Puzzles”. In: *Undergraduate Thesis and Undergraduate Honors Thesis. Department of Computer Science and Engineering, University of Nebraska-Lincoln* (2007).

[10] Alec Kirkley, George T Cantwell, and MEJ Newman. “Belief propagation for networks with loops”. In: *Science Advances* 7.17 (2021), eabf1211.

[11] Johannes Josef Schneider and Scott Kirkpatrick. “Belief Propagation and Survey Propagation”. In: *Stochastic Optimization* (2006), pp. 529–536.