INDUSTRIAL TRAINING

A PROJECT REPORT

Submitted by

Ujjwal Saini (22BCS10863)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE & ENGINEERING

Chandigarh University

July, 2023

BONAFIDE CERTIFICATE

Certified that this project report **“ SUDOKU SOLVER ”** is the bonafide work of “ **Ujjwal Saini(22BCS10863) ”** **who carried out the project work under my/our supervision.**

SIGNATURE

Ms. Jyoti Arora

SUPERVISOR

Department of Computer Science

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER

1.
INTRODUCTION

1.1. Introduction to Project:

Sudoku, a popular number puzzle that originated in Japan, has become a favourite pastime for millions of people worldwide. Its simple rules and challenging gameplay have led to the creation of countless Sudoku puzzles, ranging from easy to fiendishly difficult. The goal is to fill a 9x9 grid with numbers from 1 to 9, ensuring that each row, each column, and each of the nine 3x3 sub-grids (referred to as "regions") contains all the digits from 1 to 9, with no repetitions. While the rules of Sudoku are straightforward, solving the puzzles can be a complex and intriguing task.

In this project, we delve into the world of Sudoku and tackle the problem of solving Sudoku puzzles using a powerful algorithm known as "backtracking." Backtracking is a widely used technique in computer science and mathematics for solving combinatorial problems, and it's particularly effective for Sudoku, thanks to the puzzle's constraint-based nature.

Currently, Sudoku puzzles are becoming increasingly popular among the people all over the world. The game has become popular now in a large number of countries and many developers have tried to generate even more complicated and more interesting puzzles. Today, the game appears in almost every newspaper, in books and in many websites.

In this project we present a Sudoku Solver algorithm that is used to solve the puzzles. The algorithm is formulated based on DSA concept i.e. Backtracking with the help of C++.

Throughout this project, we aim to provide a comprehensive understanding of how backtracking can be applied to solve Sudoku puzzles and offer a practical, working solution that anyone can use to conquer even the most challenging Sudoku grids. By the end of this project, you'll not only have a Sudoku solver at your disposal but also gain insights into the broader world of backtracking algorithms and problem-solving techniques in computer science.

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

Fig 1: A 9x9 Sudoku

1.2. Identification of Problem

The core problem we aim to address in this project is the automated solution of Sudoku puzzles using the backtracking algorithm. Solving Sudoku puzzles manually can be an enjoyable and mentally stimulating exercise for many, but it can also be time-consuming and frustrating when faced with more challenging grids. Therefore, the need for a computer program capable of swiftly and accurately solving Sudoku puzzles is evident.

One key aspect of the problem is the need for a systematic approach to explore potential solutions. Traditional computer algorithms often fall short in tackling Sudoku puzzles due to their constraint-based nature. This problem is not easily reducible to standard search or optimization techniques.

The complexity of Sudoku puzzles varies, and the difficulty level can range from very easy to highly intricate. As a result, the problem we need to address extends to handling puzzles of varying complexities. Our Sudoku solver should be able to cope with all difficulty levels, making it a versatile tool for Sudoku enthusiasts and a valuable educational resource for understanding backtracking as a problem-solving technique.

CHAPTER

2.

LITERATURE REVIEW/BACKGROUND STUDY

2.1. Existing solutions

- **Human Solving Techniques**: Sudoku puzzles can be solved using the techniques employed by human players. These techniques include identifying naked and hidden singles, pairs, triples, and other patterns. Human-like solving strategies can provide insight into developing non-backtracking solving algorithms.

- **Guess and Check**: In this approach, you make educated guesses about the values of certain cells and continue solving the puzzle. If a contradiction is reached, the solver backtracks to a previous state and tries an alternative guess. This method is different from backtracking in that it doesn't explore all possible values simultaneously but rather sequentially.

- **Brute-Force Enumeration**: Similar to backtracking, this approach exhaustively enumerates all possible combinations of numbers within the constraints of the puzzle. It's not as efficient as backtracking, as it doesn't incorporate the notion of intelligent backtracking to avoid exploring unnecessary branches. Still, it can solve simpler puzzles.

- **X-Wing, Swordfish, and Jellyfish**: These are advanced Sudoku-solving techniques that look for specific patterns of candidate numbers across rows and columns. They are more complex strategies but can be quite powerful in solving challenging puzzles.

2.2. Problem Definition

The problem at hand is to design and implement a Sudoku solver that efficiently finds valid solutions for Sudoku puzzles of varying complexities, from easy to extremely challenging, using the backtracking algorithm. The solver should not only be capable of solving relatively straightforward puzzles but also handle highly complex ones by leveraging the backtracking algorithm. It should provide a solution in a reasonable amount of time, regardless of the puzzle's difficulty. It should detect and reject puzzles that do not adhere to the basic Sudoku rules, such as duplicate numbers in the same row, column, or sub-grid. The objective is to create a user-friendly, scalable, efficient, less time-consuming program that allows users to input Sudoku puzzles, view the solutions generated by the solver, and gain insights into the backtracking algorithm's operation.

2.3. Goals/Objectives

The primary goal of the project "Sudoku Solver Using Backtracking" is to create an efficient and user-friendly software solution for solving Sudoku puzzles. The project's objectives encompass various aspects, from algorithmic efficiency to user experience and educational value:

- **Implement Backtracking Algorithm**: Develop a robust and efficient implementation of the backtracking algorithm to systematically explore and solve Sudoku puzzles. The solver should be capable of handling puzzles of varying complexities.

- **Efficiency and Optimization**: Optimize the backtracking algorithm to minimize unnecessary exploration of invalid solutions, ensuring that the solver can efficiently find solutions even for highly complex puzzles.

- **Educational Value**: Provide insights into the inner workings of the backtracking algorithm, making the project a valuable educational resource for those interested in learning about problem-solving techniques in computer science and Sudoku puzzle strategies.

- **Handling Various Sudoku Grid Sizes**: Design the solver to handle different grid sizes, including the standard 9x9 Sudoku grids, as well as other variants like 6x6 or 12x12 grids. This versatility ensures a broader range of puzzle-solving capabilities.

- **Scalability**: Consider the scalability of the project, allowing for potential integration into larger software systems, applications, games, or educational tools that require Sudoku-solving capabilities.

- **Performance Testing**: Conduct rigorous testing and performance evaluations to ensure the solver's reliability, accuracy, and efficiency across a diverse range of Sudoku puzzles.

CHAPTER

3.

DESIGN FLOW/PROCESS

3.1. Evaluation & Selection of Specifications/Features

The process of evaluation and selection of specifications and features for the "Sudoku Solver Using Backtracking" project is a critical phase in defining the scope and functionality of the software. It involves assessing the requirements of the project, considering potential user needs, and making informed decisions about which features to include. Here, we outline the key steps and considerations in this process:

1. **Requirement Analysis:**

   - **Input and Output**: Evaluate the input methods for Sudoku puzzles. Consider options for manual entry, import from external files, and generating random puzzles for solving. Define the output format for presenting solved puzzles.

   - **Grid Size**: Determine the grid sizes to support, such as the standard 9x9 grid and potential variants like 6x6 or 12x12. Evaluate the complexity of adapting the solver to various grid sizes.

   - **Solving Methods**: Explore backtracking algorithm to solve the sudoku in an efficient way.

2. **User Interface Design:**

   - **Input Interface**: Create a user-friendly interface for inputting Sudoku puzzles.

   - **Usability**: Ensure that the interface is intuitive and accessible to users, including those with varying levels of experience in solving Sudoku puzzles.

3. **Algorithm Selection and Implementation:**

   - **Backtracking Algorithm**: Assess the suitability of the backtracking algorithm for solving Sudoku puzzles. Explore different variations and optimization techniques to improve solver efficiency.

4. **Complexity Testing**: Perform rigorous testing on the solver to ensure it can efficiently handle a broad range of Sudoku puzzles, from easy to extremely challenging.

5. **Platform Compatibility**: Ensure the software is compatible with common operating systems and devices to maximize its accessibility.

3.2. Analysis of Features and finalization subject to constraints

As the development of the "Sudoku Solver Using Backtracking" project progresses, it is essential to conduct a thorough analysis of the proposed features and finalize them while considering the project's constraints. This process involves evaluating the practicality of incorporating certain features and ensuring that the project remains within scope, schedule, and resource limitations. Here are the key steps in this phase:

- **Essential Features**: Identify the core features that are fundamental to the project's purpose, such as the backtracking algorithm implementation, user-friendly interface, and input validation.

- **Desirable Features**: Distinguish between features that would enhance the user experience and those that may be desirable but not critical. For example, advanced solving techniques, educational components, or support for various grid sizes.

- **Resource Limitations**: Consider the availability of resources, including development time, programming expertise, and hardware/software constraints.

- **Technical Challenges**: Identify potential technical challenges, such as optimizing the backtracking algorithm, ensuring compatibility with different platforms, and handling input validation efficiently.

- **Modularity**: Assess the design's modularity and flexibility. Features should be integrated in a way that allows for easier future expansion and maintenance.

- **Scalability Planning**: If additional features or scalability options are considered, evaluate their impact on the project's overall complexity and resource requirements.

- **Platform Considerations**: Address any compatibility challenges associated with specific operating systems, devices, or software dependencies.

- **Feature List**: Based on the analysis and considerations, finalize the list of features to be implemented in the project. Consider the optimal balance between essential, desirable, and educational features.

- **Risk Assessment**: Identify potential risks or challenges that may arise during feature implementation and create contingency plans to address them.

3.3. Design Flow (Algorithm)

❖ **<u>Step 1</u>:- Initialize the Sudoku Grid:**

- Create a 9x9 grid to represent the Sudoku puzzle.
- Use '0' to represent empty cells.

```
int grid[N][N] = {
    {5, 3, 0, 0, 7, 0, 0, 0, 0},
    {6, 0, 0, 1, 9, 5, 0, 0, 0},
    {0, 9, 8, 0, 0, 0, 0, 6, 0},
    {8, 0, 0, 0, 6, 0, 0, 0, 3},
    {4, 0, 0, 8, 0, 3, 0, 0, 1},
    {7, 0, 0, 0, 2, 0, 0, 0, 6},
    {0, 6, 0, 0, 0, 0, 2, 8, 0},
    {0, 0, 0, 4, 1, 9, 0, 0, 5},
    {0, 0, 0, 0, 0, 0, 0, 7, 9}
```

Fig 2: Input of 9x9 Sudoku

❖ **Step 2:- Define the "isSafe" Function:**

- Create a function bool isSafe(grid, row, col, num) to check if it's safe to place a number num in a given cell (row, col) of the grid.
- Check if the number is not present in the same row, same column, or the 3x3 subgrid that contains the cell. ➢ Return true if it's safe, and false otherwise.

```
bool isSafe(int grid[N][N], int row, int col, int num) {
    // Check if the number is already present in the row or column
    for (int i = 0; i < N; i++) {
        if (grid[row][i] == num || grid[i][col] == num) {
            return false;
        }
    }

    // Check if the number is already present in the 3x3 subgrid
    int startRow = row - row % 3;
    int startCol = col - col % 3;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (grid[i + startRow][j + startCol] == num) {
                return false;
            }
        }
    }

    return true;
}
```

Fig 3: isSafe Function of Code

❖ **<u>Step 3</u>:- Define the "solveSudoku" Function :**

- Create a recursive function bool solveSudoku(grid) to solve the Sudoku puzzle.
- Check if there are any empty cells left in the grid. If not, the puzzle is solved, so return true.
- If there are empty cells,try out all the possiblemcombinations by Recursively calling solveSudoku to solve the remaining cells.
- If a solution is found, return true.
- If no solution is found, backtrack by resetting the cell to '0'. ➢ If no solution is found for any number, return false.

```
bool solveSudoku(int grid[N][N]) {
    int row, col;

    // Check if there are no empty cells left
    bool isEmpty = true;
    for (row = 0; row < N; row++) {
        for (col = 0; col < N; col++) {
            if (grid[row][col] == 0) {
                isEmpty = false;
                break;
            }
        }
        if (!isEmpty) {
            break;
        }
    }

    // If there are no empty cells, the Sudoku puzzle is solved
    if (isEmpty) {
        return true;
    }

    // Try placing a number from 1 to 9 in the empty cell
    for (int num = 1; num <= 9; num++) {
        if (isSafe(grid, row, col, num)) {
            grid[row][col] = num;

            // Recursively solve for the remaining cells
            if (solveSudoku(grid)) {
                return true;
            }

            // If no solution is found, backtrack and reset the cell to 0
            grid[row][col] = 0;
        }
    }

    return false;
}
```

Fig 4: solveSudoku Function of Code

❖

Step 4:- Define the Main Function :

- In the main function:
- Initialize the Sudoku grid with your puzzle and Call the solveSudoku function to solve the puzzle.
- If a solution is found, print the solved Sudoku grid and If no solution exists, print "No solution exists."

```
if (solveSudoku(grid)) {

cout<<"------------------------SOLVED SUDOKU------------------------"<<endl;
cout<<endl;
    // Print the solved Sudoku grid

    for (int i = 0; i < N; i++) {
        cout<<"                    ";
        for (int j = 0; j < N; j++) {
            cout << grid[i][j] << "  ";
        }

        cout << endl;

    }
} else {
    cout << "No solution exists." << endl;
}
cout<<endl;
cout<<"---------------------------THANK YOU---------------------------"<<endl;
```

Fig 5: Main Function of Code

CHAPTER 4.

RESULTS ANALYSIS AND VALIDATION

4.1. Implementation of solution

The implementation phase of the "Sudoku Solver Using Backtracking" project is a critical step where the theoretical concepts and design come to life as a functional software application. This phase involves coding the solver, creating the user interface, and integrating all the essential components.

Implementing the backtracking algorithm, which systematically explores empty cells in the grid, tries different values, and backtracks when inconsistencies are encountered. Ensure that the algorithm follows Sudoku rules by checking for duplicate values in rows, columns, and regions.

The implementation phase is a dynamic process that involves coding, iterative testing, and refinement. It requires close collaboration between developers, designers, and potential end-users to ensure that the Sudoku solver meets its objectives, functions effectively, and provides an excellent user experience.

Fig 6: Output (Solved Sudoku)

CHAPTER 5.
CONCLUSION AND FUTURE WORK

5.1. Conclusion

The "Sudoku Solver Using Backtracking" project has successfully come to fruition, providing an efficient and user-friendly solution for solving Sudoku puzzles of various complexities. The journey from conception to completion has been marked by dedication, meticulous planning, and collaborative efforts. In this conclusion, we reflect on the achievements of the project and the impact it has on Sudoku enthusiasts and learners.

Algorithmic Excellence: The heart of this project lies in the implementation of the backtracking algorithm, a robust and efficient approach to solving Sudoku puzzles. The algorithm has been meticulously designed and optimized to handle a wide range of puzzle complexities. It systematically explores puzzle spaces, finds valid solutions, and elegantly backtracks when inconsistencies arise. The project's algorithmic foundation embodies the essence of problem-solving in computer science.

In conclusion, the "Sudoku Solver Using Backtracking" project has achieved its primary goal of delivering an efficient, educational, and user-friendly solution for Sudoku enthusiasts and learners. The project not only stands as a valuable tool for solving Sudoku puzzles but also offers an educational journey into the world of problem-solving algorithms. It is a testament to the potential of computer science to tackle real-world challenges and provide accessible solutions.

As the project reaches its completion, the possibilities for further development and integration into diverse applications are apparent. Whether used for personal enjoyment, educational purposes, or as a building block for larger software systems, this Sudoku solver project serves as a testament to the power of creative problem-solving in the realm of computer science.

5.2. Future Work

- **Advanced Variants**: Expanding to solve more complex Sudoku variants, such as irregular, diagonal, and overlapping puzzles.
- **Machine Learning Integration**: Incorporating machine learning for improved solving and generating personalized puzzles.
- **AI Guidance:** Developing AI agents that guide users through puzzle solving and provide hints.
- **Cross-Platform Compatibility**: Extending the solver to various platforms, including mobile and web applications.
- **Community Engagement**: Encouraging further community contributions, bug fixes, and feature enhancements to maintain and improve the project's quality and usability.

REFERENCES

- Gent, I. P., Harvey, W., & Smith, B. M. (2006). Constraint propagation in a solver for sudoku. Principles and Practice of Constraint Programming - CP 2006, 130-144.
- Sivaraj, R., & Ranjani, S. (2015). Parallel Sudoku Solver using Multithreading. Procedia Computer Science, 45, 419-428.
- Sudoku Solver using Backtracking in C++ by Gaurav Sen (2021).
- Sudoku Solver using Backtracking in Java by Abhishek Kumar (2021)
- Tomasek, A. (2014). Sudoku as a project for teaching programming and collaboration. Proceedings of the 2014 conference on Innovation & technology in computer science education, 203-208.
- Weikpedia.org. "Sudoku solving algorithms." [Online] Available: https://en.wikipedia.org/wiki/Sudoku_solving_algorithms. Accessed: October 2023.
- Russell, Stuart, and Peter Norvig. "Artificial Intelligence: A Modern Approach." Pearson, 2021. ISBN: 978-0134610996.
- Berthold, Michael R., and Jussi Rasku. "The Backtracking Solver." Springer, 2016. ISBN: 978-3-319-25549-1.