

Artificial Intelligence (CSE 2225) MINI PROJECT REPORT ON

**Sudoku Solver**

*SUBMITTED TO*

**Department of Computer Science & Engineering**

*by*

1.) Arnav Karnik 220962021 – AIML B

2.) Nikhil Nair 220962013- AIML B

3.) Yashas Sagilli 220962012- AIML B

4.) Krishanu Banerjee 220962290 – AIML B

Name & Signature of Evaluator

(Jan 2024 - May 2024)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Table of Contents** | | | | |
|  | | | | Page No |
|  | | | | |
| **Chapter 1** | | | **INTRODUCTION** | 1 |
|  | **1.1** | Introduction | | 1 |
|  | **1.2** | Problem Statement | | 1 |
|  | **1.3** | Objective | | 1 |
|  | | | | |
| **Chapter 2** | | | LITERATURE REVIEW – Mention the topic from syllabus | **3** |
|  | **2.1** | Formulating the problem | | 3 |
|  | **2.2** | References of the algorithm from the textbook | | 4 |
|  | | | | |
| **Chapter 3** | | | **METHODOLOGY – Discussion of algorithm and technique used** | **6** |
|  | **3.1** | Overview of backtracking | | 6 |
|  | **3.2** | How it is used in the Sudoku Solver? | | 7 |
|  | | | | |
| **Chapter 4** | | | **RESULTS AND DISCUSSION** | **10** |
|  | **4.1** | Screenshots of Output | | 10 |
|  | **4.2** | Features present | | 11 |
|  | | | | |
| **Chapter 5** | | | **CONCLUSIONS AND FUTURE ENHANCEMENTS** |  |
|  | **5.1** | Conclusions | |  |
|  | **5.2** | Future Enhancements | |  |
|  | | | | |
| **REFERENCES** | | | |  |
|  | | | |  |

**Rules for report making**

**Font – Times New Roman**

**Fontsize Title – 14 pt Bold**

**Subtitle – 12 pt Bold**

**Content – 12 pt.**

**All figures and tables should be labelled.**

**Organization of Report:**

CONTENTS

1. Introduction

2. Literature survey, with brief descriptions of the contributions in each of the paper referred. / Course content from the syllabus

3. Methodology

4. Results and discussion

5. Conclusions and future enhancements, if any

6. Reference(s)

**Chapter 1 – Introduction**

**Unit 1.1- Introduction:**

Sudoku, the popular logic-based number puzzle, presents a captivating challenge for puzzle enthusiasts. With its grid of 9x9 cells divided into nine 3x3 sub grids, Sudoku requires filling in the grid with numbers from 1 to 9 such that each row, column, and sub grid contains every number exactly once.

Solving Sudoku puzzles manually can be both enjoyable and demanding, often requiring systematic strategies and logical deductions. However, for complex puzzles or enthusiasts seeking efficiency, automated Sudoku solvers provide an invaluable tool.

In this project, we explore the development of a Sudoku solver using computational methods. By leveraging algorithms and techniques such as backtracking and constraint propagation, we aim to create a program capable of swiftly solving Sudoku puzzles of varying difficulty levels.

This solver will not only demonstrate the power of computer algorithms in puzzle solving but also serve as an educational exploration into the realms of artificial intelligence and logic-based problem-solving.

**Unit 1.2 – Problem Statement:**

Sudoku is a popular number puzzle consisting of a 9x9 grid divided into nine 3x3 sub grids. The puzzle starts with some cells filled with numbers from 1 to 9, and the goal is to fill the empty cells in such a way that every row, column, and 3x3 sub grid contains all the numbers from 1 to 9 without repetition.

Develop a program that can solve Sudoku puzzles automatically using computational techniques.

**Unit 1.3 – Objective:**

Aim - Develop a program that can solve Sudoku puzzles automatically using backtracking algorithm

Input -

1. A partially filled Sudoku grid represented as a 2D array (9x9).

2. The grid contains numbers from 1 to 9 in cells that are already filled, and empty cells are represented by zeros or another designated empty value.

Output -

The solved Sudoku grid, where all empty cells have been filled according to the Sudoku rules.

Constraints -

1. The input Sudoku grid will be a valid partially filled Sudoku puzzle.

2. Each row, column, and 3x3 sub grid of the input grid will contain numbers from 1 to 9 with no repetitions in the filled cells.

Expectations:

1. The Sudoku solver should implement efficient algorithms to solve Sudoku puzzles of varying difficulty levels.

2. The solution should correctly fill in all empty cells of the input grid while adhering to Sudoku rules.

3. The solver should handle edge cases and invalid inputs gracefully, providing informative outputs or error messages when necessary.

**Chapter 2 – Literature Review**

**Unit 2.1 – Formulating the Problem -**

States: Each state represents a configuration of the Sudoku puzzle, which includes a 9x9 grid with numbers filled in some cells and other cells left empty.

Initial State: The initial state is the partially filled Sudoku puzzle provided as input to the solver. It consists of a 9x9 grid where some cells contain numbers from 1 to 9, and others are empty (represented by zeros or another designated empty value).

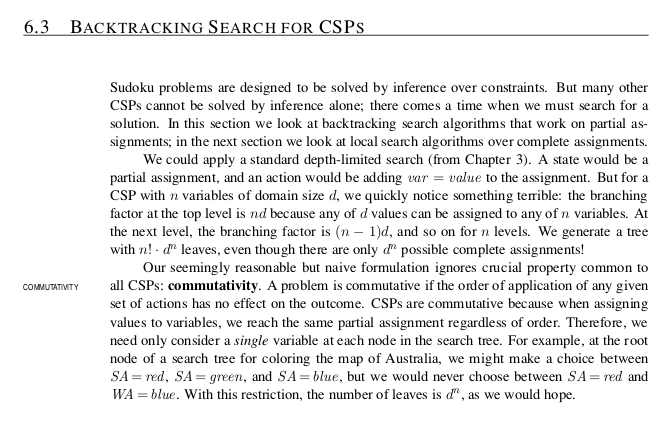
Actions: An action in this context refers to placing a valid number (from 1 to 9) into an empty cell of the Sudoku grid. Each action involves selecting an empty cell and attempting to fill it with a valid number that adheres to Sudoku rules (no repetitions in rows, columns, or 3x3 sub grids).

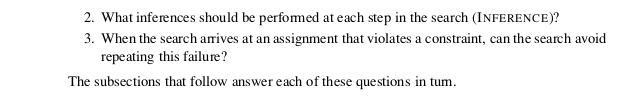
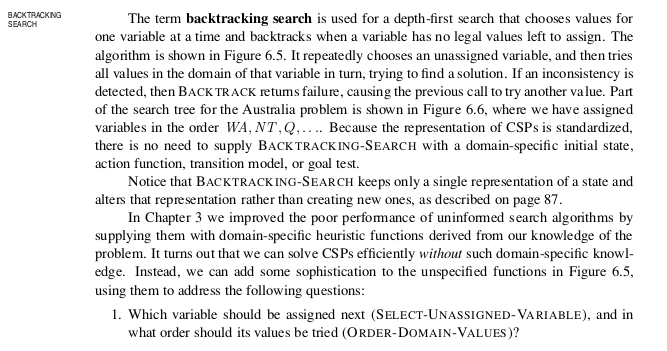
Transition Model: The transition model defines how the Sudoku solver transitions from one state (grid configuration) to another by applying valid actions. When an action is executed (e.g., placing a number in an empty cell), the solver modifies the grid accordingly, potentially leading to a new state.

Goal Test: The goal test checks whether the current state of the Sudoku puzzle satisfies the completion criteria. The goal is achieved when all cells of the 9x9 grid are filled with numbers from 1 to 9, adhering to Sudoku rules (no repetitions in rows, columns, or 3x3 sub grids).

Path Cost: For a Sudoku solver, the path cost can be defined as the number of actions (number placements) required to transition from the initial state (partially filled puzzle) to the goal state (fully solved puzzle). The goal is to minimize the path cost by efficiently applying valid number placements to solve the puzzle.

**Unit 2.2 – Reference of the algorithm from textbook**





**Chapter 3 – Methodology – Discussion of the algorithm**

**Unit 3.1 – Overview of backtracking:**

The backtracking algorithm is a powerful technique used to systematically search for a solution to a problem, especially in cases where there are multiple decision points leading to a solution. It is particularly effective for solving constraint satisfaction problems like Sudoku, where the goal is to find a valid arrangement of values that satisfies certain constraints.

1. Basic Idea: Backtracking is a depth-first search technique that explores potential solutions incrementally, backtracking as soon as it determines that a particular path cannot lead to a valid solution.

2. Recursive Approach: The backtracking algorithm uses a recursive function to explore each decision point:

* + It tries out a choice (e.g., placing a number in a cell).
  + Recursively explores further with that choice.
  + If the choice leads to a dead end (i.e., violates constraints or doesn't lead to a solution), it backtracks and tries a different choice.

### Efficiency and Complexity:

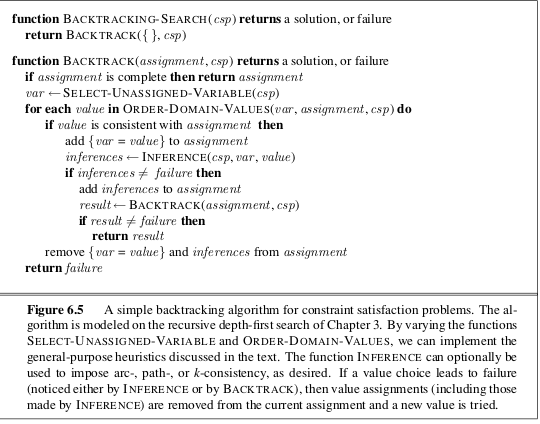
* Backtracking explores the solution space in a depth-first manner, which can lead to pruning branches early if they lead to invalid states.
* The time complexity of the backtracking algorithm heavily depends on the branching factor (number of choices at each decision point) and the depth of the recursion.
* For Sudoku, the worst-case time complexity is typically manageable due to the constraints of the puzzle (81 cells with limited number choices).

### Implementation Considerations

* Implement checks for validity (row, column, and 3x3 subgrid constraints) efficiently to optimize the solver.
* Handle edge cases and invalid inputs gracefully within the backtracking algorithm.

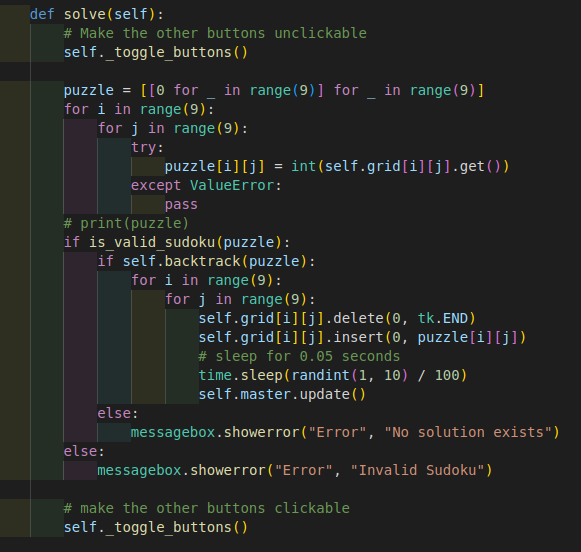
**Unit 3.2 – How Backtracking is used in the sudoku solver:**

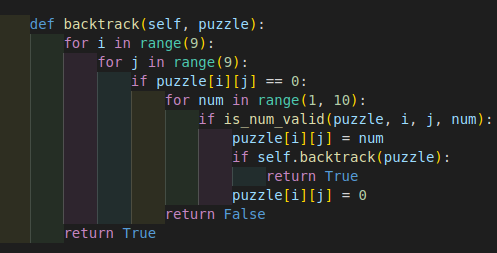
3.2.1-Algorithm:

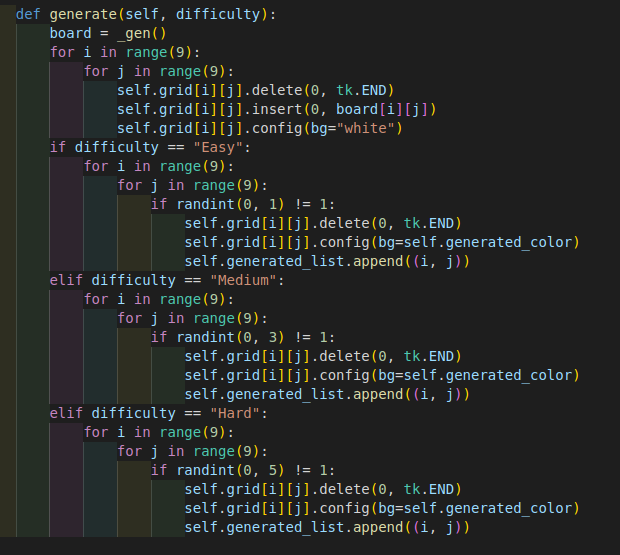


### 3.2.2- Steps for Backtracking Sudoku Solver:

1. Initialization:
   * Start with the partially filled Sudoku grid.
   * Identify an empty cell to begin with (a cell with value 0 or another designated empty value).
2. Recursive Backtracking Function:
   * Define a recursive function (solveSudoku):
     + Base Case: If the grid is fully filled (no empty cells left), return True (solution found).
     + Recursive Case:
       - Choose an empty cell.
       - Try placing numbers (1 to 9) in the chosen cell.
       - For each number:
         * Check if the number placement is valid (no conflicts with row, column, or 3x3 subgrid).
         * If valid, recursively call solveSudoku on the updated grid.
         * If recursion returns True (solution found), propagate True up the call stack.
         * If recursion returns False, backtrack (undo the number placement) and try the next number.
3. Backtracking Mechanism:
   * If no valid number can be placed in the current empty cell (all numbers lead to conflicts), return False to backtrack.
   * Backtracking involves undoing the last number placement (resetting the cell value) and trying the next possible number.
4. Solution Checking:
   * After trying all possible numbers for a given cell, return the result (solution found or not) to the calling function.



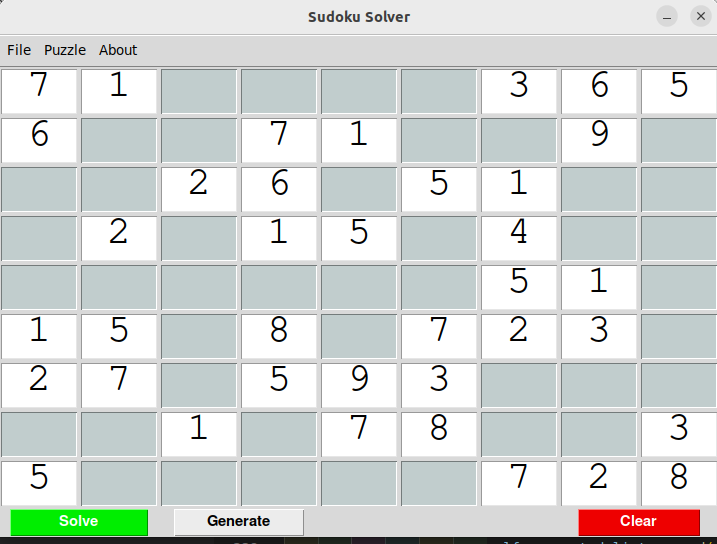




**Chapter 4 – Results and Discussion**

**Unit 4.1 – Screenshots of output:**

Initial Output -



Final Output -

**Unit 4.2 – Features Present:**

1.) Users have the option to input numbers manually to solve the Sudoku puzzle.

2.) The Solve function uses the backtracking algorithm to completely solve the Sudoku puzzle.

3.) The Generate feature randomly assigns numbers to certain squares, creating a new Sudoku puzzle.

4.) By using the Clear function, all squares on the board are reset, allowing for a new state and the start of a new puzzle.

**Chapter 5 - Conclusions and Future Enhancements**

**Unit 5.1 – Conclusions:**

In conclusion, the backtracking algorithm provides an effective and elegant solution to solving Sudoku puzzles. By leveraging a systematic depth-first search approach with recursive backtracking, the algorithm efficiently explores potential solutions while gracefully handling constraints and conflicts inherent in the puzzle.

The key strengths of the backtracking approach lie in its ability to:

* Navigate through the solution space by trying different possibilities at each decision point.
* Prune invalid branches early, minimizing unnecessary exploration of non-viable paths.
* Backtrack and undo choices when a dead-end is encountered, allowing the algorithm to explore alternative paths.

Through this methodical process, the backtracking algorithm can successfully solve even complex Sudoku puzzles, leveraging logical deductions and constraint satisfaction principles. While the algorithm's performance can be influenced by factors such as the puzzle's complexity and the efficiency of constraint-checking mechanisms, its fundamental design makes it a versatile and widely applicable technique for solving various types of constraint satisfaction problems.

Overall, the backtracking algorithm stands as a testament to the power of systematic exploration and logical deduction in problem-solving, offering a robust framework for tackling Sudoku and similar puzzles with clarity and efficiency.

**Unit 5.2 – Future Enhancements:**

For a Sudoku solver program, there are several potential future enhancements that can improve functionality, performance, and user experience. Here are some ideas for enhancing a Sudoku solver:

1. Optimized Algorithm Implementations:
   * Implement advanced algorithms like "Constraint Propagation" (e.g., using techniques like Naked Pairs/Triples) to reduce the search space and solve puzzles more efficiently.
   * Explore techniques beyond basic backtracking, such as heuristic-based search algorithms like A\* with suitable heuristics for Sudoku.

2. Performance Optimization:

* Profile and optimize critical parts of the solver code for improved performance, especially for solving complex or large puzzles.
* Explore parallel processing techniques (multi-threading or distributed computing) to leverage multiple CPU cores for faster solving.

3. Accessibility and Localization:

* Improve accessibility features such as supporting screen readers or high contrast modes for visually impaired users.
* Localize the solver to support multiple languages and regional preferences for a broader user base.

4. Data Persistence and Sharing:

* Add functionality to save and load Sudoku puzzles from files or databases, enabling users to resume solving or share puzzles with others.
* Implement puzzle sharing features where users can exchange puzzles and solutions within a community

**References**