MANIPAL INSTITUTE OF TECHNOLOGY

MANIPAL

(A constituent unit of MAHE, Manipal)

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 - 220962021- AIML B

4.) Krishanu Banerjee - 220962290 – AIML B

Name & Signature of Evaluator

(Jan 2024 - May 2024)

Table of Contents

Chapter 1 – Introduction

Unit 1.1- Introduction:

For puzzle fans, the well-known logic-based number puzzle Sudoku offers an engaging challenge. Sudoku needs to fill in the grid with numbers from 1 to 9 so that each row, column, and sub grid has every number precisely once. The grid is made up of nine 3x3 sub grids separated into a grid of nine 9x9 cells.

Manual Sudoku problem-solving can be entertaining and challenging at the same time because it frequently calls for methodical approaches and conclusions from logic. However, automatic Sudoku solvers are an extremely helpful tool for those who enjoy solving challenging puzzles or are efficiency-driven fanatics.

In this project, we research the use of computational techniques to create a Sudoku solver. Using methods and algorithms like constraint propagation and backtracking, our goal is to develop a program that can solve Sudoku problems of different levels of difficulty quickly.

This solver will serve as an instructional tool for investigating the fields of artificial intelligence and logic-based problem-solving, in addition to showcasing the effectiveness of computer algorithms in puzzle solving.

Unit 1.2 – Problem Statement:

Create an AI-powered Sudoku solver that can effectively answer any Sudoku puzzle by utilizing sophisticated search strategies like constraint propagation, backtracking, and other methods. An incomplete Sudoku problem should be accepted as input by the solver, who should then represent it suitably and fill in the blank cells while adhering to the Sudoku rules to determine the answer. Include optimization techniques, deal with erroneous inputs, and produce an understandable output of the puzzle's solution. Create a modular and extendable solver for upcoming improvements.

Unit 1.3 – Objective:

Aim - Develop a program that can solve Sudoku puzzles automatically using backtracking algorithm and generate Sudoku problems.

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 is an arrangement of the Sudoku puzzle pieces, which consist of a 9x9 grid with some of the cells filled in with numbers and others left vacant.

Initial State: The partially completed Sudoku puzzle that is fed into the solver is the beginning state. It is composed of a 9x9 grid with zeros or another defined empty value in some cells and numbers ranging from 1 to 9 in others.

Actions: In this context, an action is any number that may be entered into an empty Sudoku grid cell, ranging from 1 to 9. In order to complete an action, you must first choose an empty cell and try to fill it with a number that complies with the Sudoku criteria (no repeats in rows, columns, or 3x3 sub grids).

Transition Model: Using appropriate actions, the transition model describes how the Sudoku solver moves from one state (grid configuration) to another. The solver changes the grid in response to an action (such as entering a number in an empty cell), which may result in a new state.

Goal Test: The purpose of the goal test is to determine if the Sudoku problem meets the completion requirements in its current state. When every cell in the 9x9 grid contains a number between 1 and 9, following the guidelines of Sudoku (i.e., no repetitions in rows, columns, or 3x3 sub grids), the goal is accomplished.

Path Cost: The number of actions (number placements) needed to travel from the starting state (a partially filled puzzle) to the destination state (a fully completed problem) is the path cost for a Sudoku solver. By applying proper number placements as effectively as possible to solve the puzzle, the objective is to minimize the path cost.

Unit 2.2 – Reference of the algorithm from textbook

The paper describes the methodology of using backtracking search for solving constraint satisfaction problems like Sudoku. In order to solve a Constraint Satisfaction Problem like sudoku the most common methodology used is backtracking search. At first the problem is modeled as a CSP by defining variables, their domains, and constraints. The technique of backtracking search, which is a depth-first search that builds candidates by assigning values to variables one at a time,is deployed. Constraint propagation techniques like forward checking and arc consistency is used to prune the search space by removing variable values that will violate constraints. Heuristics like the minimum remaining values (MRV) heuristic is defined to decide which variable to assign next, favoring the most constrained ones. Backtracking is performed when a constraint is violated, undoing the most recent variable assignment and trying a new value. The search terminates when all variables are assigned while satisfying all constraints, or when all possibilities are exhausted. The key ideas are systematically assigning values while using propagation to detect constraints violations early and avoid fruitless assignments. Heuristics guide the search order for better efficiency.

Chapter 3 – Methodology – Discussion of the algorithm

Unit 3.1 – Overview of backtracking:

The backtracking algorithm is an effective method for methodically looking for a solution to an issue, particularly when there are several options that might be taken. It works especially well for solving Sudoku-style constraint fulfilment problems, in which the objective is to come up with a workable value arrangement that complies with a set of restrictions.

1. Fundamental Concept: When a given path cannot lead to a workable solution, retracing is a depth-first search strategy that investigates potential solutions progressively.

2. Recursive Approach: To investigate every decision point, the backtracking algorithm makes use of a recursive function:

- It tests a selection (for example, entering a number in a cell).
- Proceeds to study that option in a recursive manner.
- 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 prunes branches early if they lead to invalid states because it investigates the solution space depth-first.
- The branching factor, or the number of options at each decision point, and the depth of the recursion have a significant impact on the backtracking algorithm's temporal complexity.
- Sudoku's 81 cells and restricted number of options make the worst-case time complexity doable.

Implementation Considerations:

- To optimise the solution, efficiently implement validity checks (row, column, and 3x3 subgrid restrictions).
- Within the backtracking algorithm, appropriately handle edge cases and invalid inputs.

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

3.2.1-Algorithm:

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or k-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

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.

```python
def solve(self):
    # Make the other buttons unclickable
    self._toggle_buttons()

    puzzle = [[0 for _ in range(9)] for _ in range(9)]
    for i in range(9):
        for j in range(9):
            try:
                puzzle[i][j] = int(self.grid[i][j].get())
            except ValueError:
                pass
    # print(puzzle)
    if is_valid_sudoku(puzzle):
        if self.backtrack(puzzle):
            for i in range(9):
                for j in range(9):
                    self.grid[i][j].delete(0, tk.END)
                    self.grid[i][j].insert(0, puzzle[i][j])
                    # sleep for 0.05 seconds
                    time.sleep(randint(1, 10) / 100)
                    self.master.update()
        else:
            messagebox.showerror("Error", "No solution exists")
    else:
        messagebox.showerror("Error", "Invalid Sudoku")

    # make the other buttons clickable
    self._toggle_buttons()
```

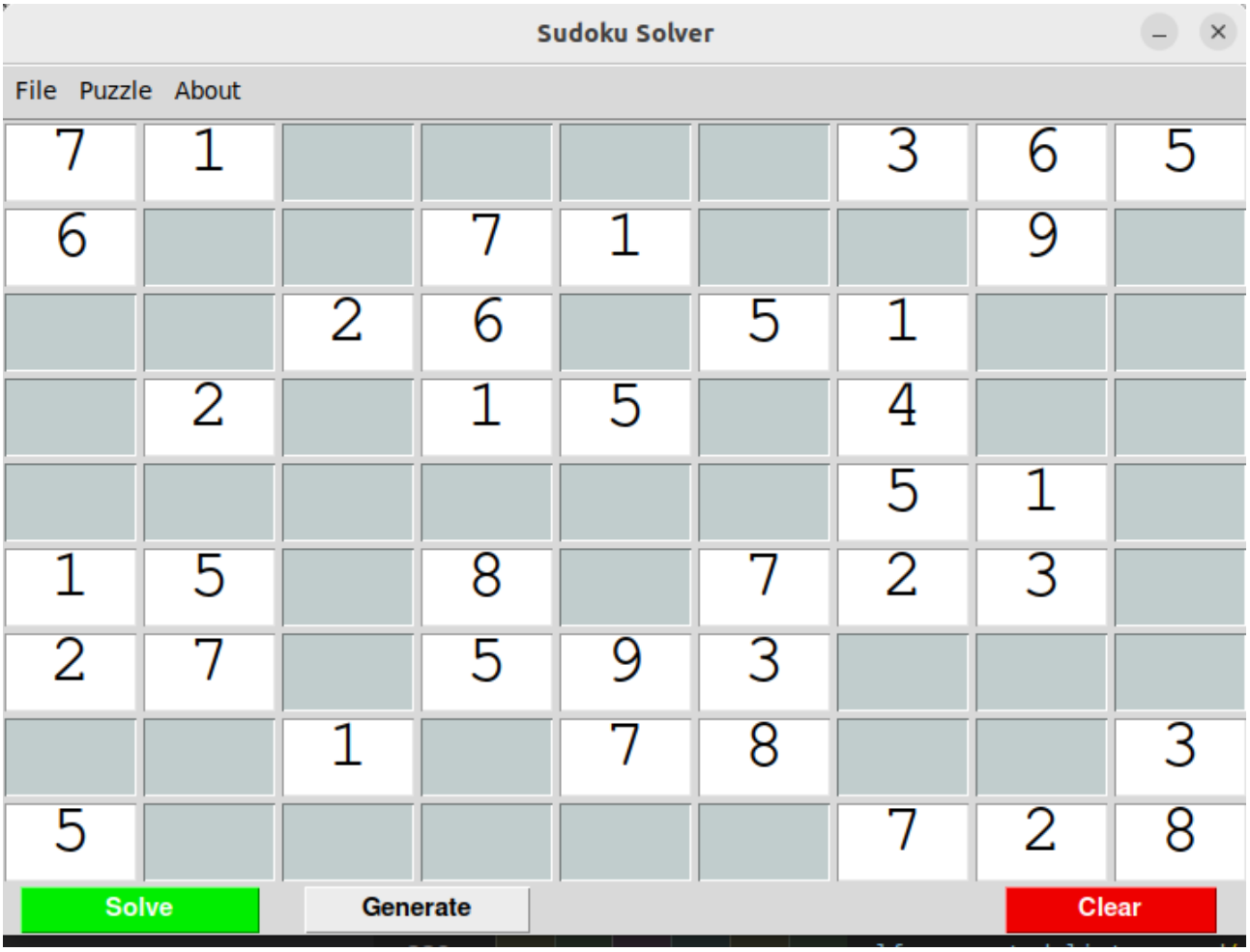
```
    def backtrack(self, puzzle):
        for i in range(9):
            for j in range(9):
                if puzzle[i][j] == 0:
                    for num in range(1, 10):
                        if is_num_valid(puzzle, i, j, num):
                            puzzle[i][j] = num
                            if self.backtrack(puzzle):
                                return True
                            puzzle[i][j] = 0
                    return False
        return True
```

```
def generate(self, difficulty):
    board = _gen()
    for i in range(9):
        for j in range(9):
            self.grid[i][j].delete(0, tk.END)
            self.grid[i][j].insert(0, board[i][j])
            self.grid[i][j].config(bg="white")
    if difficulty == "Easy":
        for i in range(9):
            for j in range(9):
                if randint(0, 1) != 1:
                    self.grid[i][j].delete(0, tk.END)
                    self.grid[i][j].config(bg=self.generated_color)
                    self.generated_list.append((i, j))
    elif difficulty == "Medium":
        for i in range(9):
            for j in range(9):
                if randint(0, 3) != 1:
                    self.grid[i][j].delete(0, tk.END)
                    self.grid[i][j].config(bg=self.generated_color)
                    self.generated_list.append((i, j))
    elif difficulty == "Hard":
        for i in range(9):
            for j in range(9):
                if randint(0, 5) != 1:
                    self.grid[i][j].delete(0, tk.END)
                    self.grid[i][j].config(bg=self.generated_color)
                    self.generated_list.append((i, j))
```

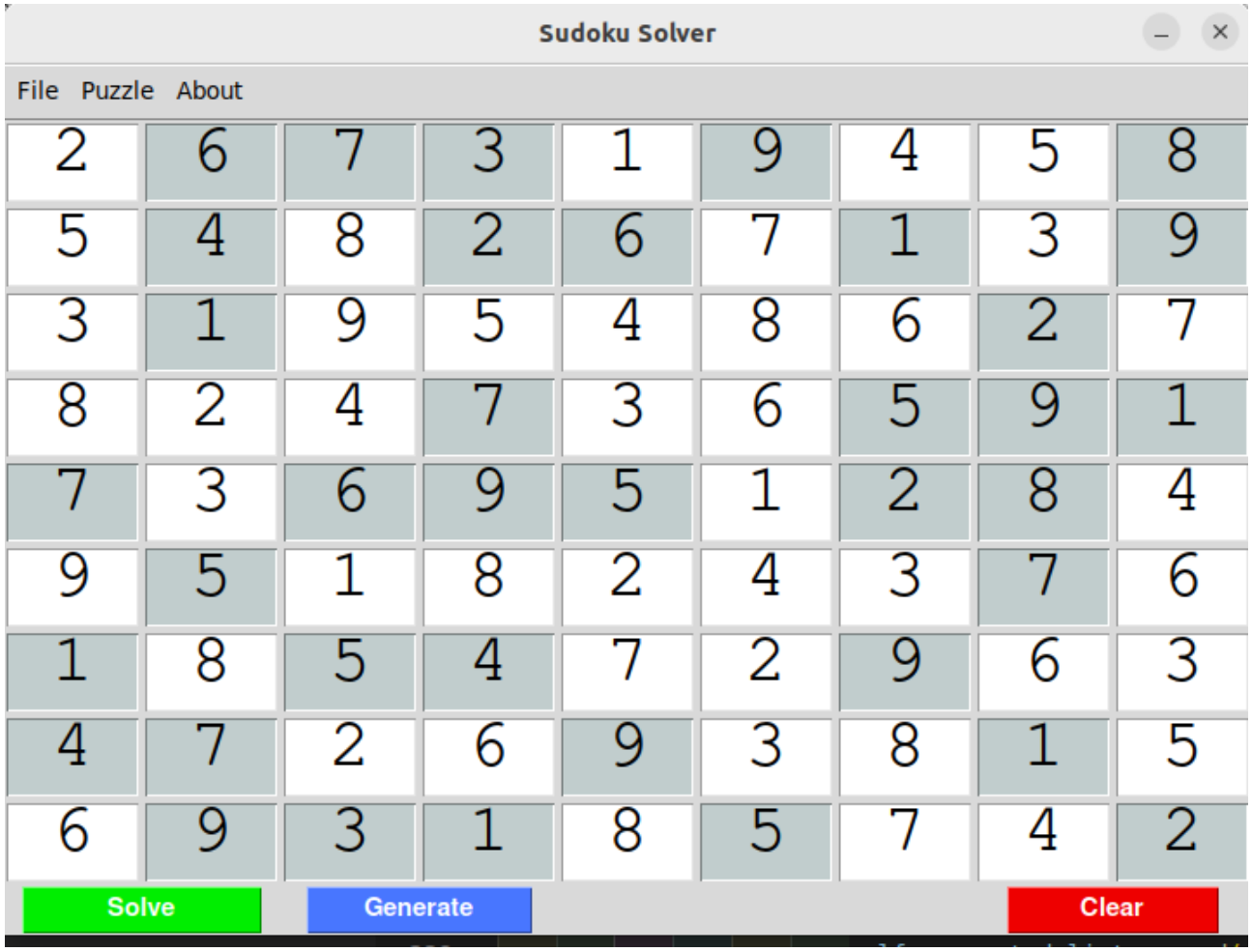
Chapter 4 – Results and Discussion

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:

To sum up, the backtracking method offers a clever and efficient way to solve Sudoku puzzles. Using a methodical depth-first search strategy combined with recursive backtracking, the program effectively investigates possible solutions while tactfully managing the limitations and contradictions included in the puzzle.

The ability to:

- Prune invalid branches early, minimizing needless investigation of non-viable paths.
- Navigate across the solution space by trying multiple possibilities at each decision point.
- When an algorithm encounters a dead end, go back and undo decisions so that the program can investigate other options.

Utilising logical reasoning and constraint fulfilment principles, the backtracking algorithm may effectively answer even the most challenging Sudoku problems through this meticulous procedure. Although the intricacy of the puzzle and the effectiveness of constraint-checking systems might affect the algorithm's performance, its basic design makes it a flexible and broadly applicable method for solving many kinds of constraint satisfaction issues.

Overall, the backtracking technique provides a solid framework for approaching Sudoku and related puzzles with clarity and efficiency, serving as a monument to the value of methodical investigation and logical reasoning in problem-solving.

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. Enhanced Algorithm Applications:

    - Reduce the search space and solve problems more quickly by putting sophisticated algorithms like "Constraint Propagation" (e.g., employing approaches like Naked Pairs/Triples) into practice.

- Investigate methods other than simple backtracking, such as search algorithms based on heuristics, such as A*, which have heuristics appropriate for Sudoku.

2. Performance Optimization:

    - To enhance performance, particularly when solving complicated or large puzzles, profile and optimize key components of the solver code.
    - Investigate distributed computing and multi-threading parallel processing strategies to take advantage of numerous CPU cores for quicker problem solutions.

3. Localization and Accessibility:

    - Enhance features that support screen readers or high contrast modes for users with visual impairments.
    - Localize the solver to accommodate a wider range of regional preferences and languages.

4. Data Persistence and Sharing:

    - Provide the ability to load and store Sudoku puzzles from databases or files, so that users can continue working on the puzzles or share them with others.
    - Provide puzzle sharing tools so that members of a community can trade riddles and solutions.

References

- http://www.cs.toronto.edu/~torsten/csc384-f11/lectures/csc384f11-Lecture04-BacktrackingSearch.pdf

- https://www.researchgate.net/publication/2625821_Backtracking_Algorithms_for_Constraint_Satisfaction_Problems_-_a_Tutorial_Survey