CW1 Report

F29AI - Artificial Intelligence

Keqin ZHANG (H00460395) Yuwei ZHAO (H00460398)

OUC CW1 33 2025-11-10

1 Part 1 - *Sudoku* Puzzels

1.1 Part 1A

1.1.1 Procedure

A CSP(constraint satisfaction problem) should involve the following three components: Variables, Domains and Constraints. Therefore, we can define the Sudoku problem as follows:

$$\text{Sudoku} = \langle V, D, C \rangle$$

where

- V: The set of 81 variables, $V = \{V_{i,j} \mid i, j \in \{1, 2, \ldots, 9\}\}$.
- D: The domain $D_{i,j}$ for each variable $V_{i,j}$ is defined as:
  - $D_{i,j} = \{k\}$, if $V_{i,j}$ is a given cell with value k.
  - $D_{i,j} = \{1, 2, \ldots, 9\}$, if $V_{i,j}$ is an empty cell.
- C: The set of 9×3 all-different constraints:
  - C_{row}: For each row i, all variables $V_{i,1}, V_{i,2}, \ldots, V_{i,9}$ must have different values.
  - C_{col}: For each column j, all variables $V_{1,j}, V_{2,j}, \ldots, V_{9,j}$ must have different values.
  - C_{subgrid}: For each 3×3 subgrid, all 9 variables within that subgrid must have different values.

1.1.2 Time Complexity Analysis

1. **Brute-force Search Algorithm:**
   For each of the k spaces, there are 9 possible choices of numbers. This results in a total of $9 \times 9 \times \ldots \times 9$ (k times) combinations. Therefore, the time complexity of the brute-force search algorithm is $O(9^k)$. When the worst-case scenario occurs, the algorithm needs to explore all possible combinations, leading to the $O(9^{81})$ time complexity.

2. **Backtracking Search Algorithm:**
   It checks the validity of constraints (row, column, and 3×3 sub-grid) immediately after assigning a number to a cell. If a conflict is detected, namely the current partial solution is invalid, the algorithm recursively backtracks to the previous step to try a different number. This process effectively prunes large sub-trees of the search space that are known to be invalid.

While the theoretical worst-case time complexity remains $O(9^k)$, which is similar to brute-force, the average-case performance is drastically faster. This is because the effective branching factor b becomes significantly smaller than 9 ($b \ll 9$) as the constraints restrict the number of valid choices for each subsequent cell.

1.2 Part 1B

1.2.1 Procedure

1. **Problem abstraction & CSP modelling**
   As stated in Part 1.1.1, model the puzzle as a CSP:

$$\textit{Sudoku} = \langle V, D, C \rangle$$

   where V is the 81 variables $V_{i,j}$; $D_{i,j}$ are domains and C are the row / column / 3×3 subgrid all-different constraints.

   In code this is corresponding to:

   - `sudoku_solver.board` : a 9×9 integer matrix (0 means empty).
   - `is_valid(row,col,num)` : enforces the three constraint types for a tentative assignment.

2. **Input format preset**
   Implement robust input routines that accept `.csv` formatted 9×9 grids. It needs to read rows with comma separation, convert each token to int, accept 0 or blank as empty.

   In our code base this is the method `load_from_csv(filepath)`. Validate that the parsed grid has 9 rows $\times$ 9 columns, otherwise throw / report an error.

3. **Core solver build**
   The core algorithm implements a Depth-First Search(DFS) based Backtracking strategy. The implementation logic within the method `solve_algorithm()` proceeds as follows:

   (a) **Select variables**
   The solver iterates through the grid coordinates to identify the first unassigned variable $V_{i,j}$.

   (b) **Assign values**
   For the selected empty cell, the algorithm sequentially attempts to assign values $d \in \{1, \ldots, 9\}$.

(c) **Carry out prune**
Before finalizing an assignment, the `is_valid()` function checks if the assignment violates any Row, Column, or Subgrid constraints. If a violation is detected, the branch is pruned immediately, which means there will be no further recursion occurs for that value.

(d) **Backtracking**

- If the assignment is valid, the state is updated and the function calls itself recursively to solve the rest of the board.
- If the recursive call returns `True`, the solution is propagated up the stack.
- If the recursive call returns `False`, the algorithm performs a backtrack - reset $V_{i,j}$ to 0 and proceed to try the next value in the domain.

4. **Relevant Metrics Print**

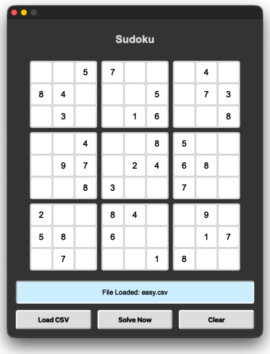
- **Execution Time:** Measured using `time.perf_counter()` in milliseconds to evaluate real-world speed.
- **Backtracks:** Counters incremented whenever the solver hits a dead end and must reverse an assignment. This metric serves as a proxy for the size of the search space explored.
- **Recursive Calls & Steps:** Tracks the depth and total operations of the search tree.

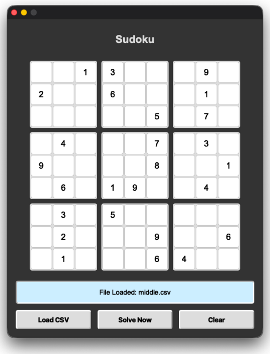
These metrics are encapsulated in the `run_solver()` wrapper method, separating the measurement logic from the core recursive algorithm.

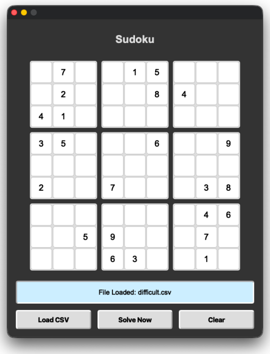
1.2.2 Testing Results

- **Methodology**
In order to test the efficiency of the method "Backtracking with pruning", I seperate the puzzels into three levels - Easy, Middle and Difficult. Specifically, puzzles and its corresponding solutions from each level are provided by *Sudoku Name*.

- **Result Display**







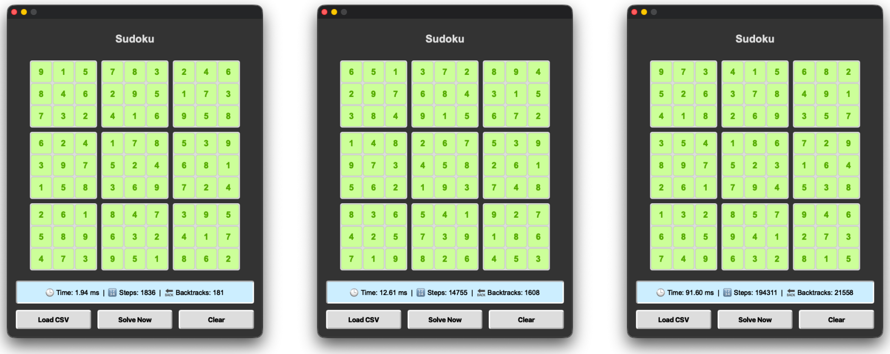


Figure 1: Puzzles with increasing levels

1.2.3 Theoretical Comparison: Backtracking vs. A* Search

1.

1.3 Part 2 - Automated Planning

1.3.1 Part 2A: Modelling the Domain

1.3.2 Part 2B: Modelling the Problems

1.3.3 Part 2C: Extension

2 Reflection and Analysis

3 Conclusion

4 Source Code

- **Part1 - *sudoku_solver.py***

```python
import csv
import time

class SudokuSolver:
    def __init__(self):
        self.board = []
        # Relevant metrics
        self.steps = 0              # Total number of steps
        self.recursive_calls = 0  # Recursive calls
        self.backtracks = 0         # Backtracks
        self.start_time = 0
        self.execution_time = 0

    def load_from_csv(self, filepath):
        self.board = []
        try:
            with open(filepath, 'r', encoding='UTF-8') as f:
                reader = csv.reader(f)
                for row in reader:
                    # Converts the string to an integer, handling possible whitespace
                    index = [int(num.strip()) for num in row if num.strip().isdigit()]
                    if len(index) == 9:
                        self.board.append(index)

            if len(self.board) != 9:
                raise ValueError("Invalid row count in CSV")

            print(f"[Succeed] File loaded: {filepath}")
            return True
        except Exception as e:
            print(f"[Error] File loading failed: {e}")
            return False

    def is_valid(self, row, col, num):
        # Row check
        for x in range(9):
            if self.board[row][x] == num:
                return False

        # Column check
        for x in range(9):
            if self.board[x][col] == num:
                return False

        # 3x3 box check
        start_row = row - row % 3
        start_col = col - col % 3
        for i in range(3):
            for j in range(3):
                if self.board[i + start_row][j + start_col] == num:
                    return False

```

```python
        return True

    def solve_algorithm(self):
        self.recursive_calls += 1

        for i in range(9):
            for j in range(9):
                if self.board[i][j] == 0:
                    for num in range(1, 10):
                        self.steps += 1
                        # If invalid, prune it
                        if self.is_valid(i, j, num):
                            self.board[i][j] = num

                            if self.solve_algorithm():
                                return True

                            # Backtracking
                            self.board[i][j] = 0
                            self.backtracks += 1

                    return False
        return True

    def run_solver(self):
        self.steps = 0
        self.recursive_calls = 0
        self.backtracks = 0

        self.start_time = time.perf_counter()

        success = self.solve_algorithm()

        end_time = time.perf_counter()
        self.execution_time = (end_time - self.start_time) * 1000

        return success

    def print_metrics(self):
        print("\n")
        print(f"Execution Time: {self.execution_time:.4f} ms")
        print(f"Total Steps (Attempts): {self.steps}")
        print(f"Recursive Calls: {self.recursive_calls}")
        print(f"Backtracks: {self.backtracks}")
        print("\n")
```

5 Reference

@ CSP
<https://en.wikipedia.org/wiki/Constraint_satisfaction_problem>
<https://www.geeksforgeeks.org/artificial-intelligence/constraint-satisfaction-probler

@ Brute-force
<https://en.wikipedia.org/wiki/Brute-force_search>
<https://www.geeksforgeeks.org/dsa/brute-force-approach-and-its-pros-and-cons/>

@ Backtracking
<https://en.wikipedia.org/wiki/Backtracking>
<https://www.geeksforgeeks.org/dsa/backtracking-algorithms/>

@ pyqt
1. <https://www.riverbankcomputing.com/static/Docs/PyQt6/>
2. <https://www.runoob.com/python3/python-pyqt.html>
3. <https://blog.csdn.net/u012117917/article/details/41604711>
4. <https://zhuanlan.zhihu.com/p/390192953>

@

See Resources on \href{https://github.com/ZhangKeqin0307/couresework1.git}{git@github