CW1 Report

F29AI - Artificial Intelligence

Keqin ZHANG (H00460395) Yuwei ZHAO (H00460398)

OUC CW1 33 2025-11-10

Note: See Resources on git@github.com:ZhangKeqin0307/couresework1.git.

1 Part 1 - *Sudoku* Puzzels

1.1 Part 1A

1.1.1 Procedure

A CSP(constraint satisfaction problem) [1] 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:

$$\begin{cases} D_{i,j} = k, & \text{if } V_{i,j} \text{ is a given cell with value } k, \\ D_{i,j} = \{1, 2, \ldots, 9\}, & \text{if } V_{i,j} \text{ is an empty cell.} \end{cases}$$

- C: The set of 27 (9 rows + 9 columns + 9 subgrids) all-different constraints:

$$\begin{cases} C_{\text{row}}: & \text{Each row } i \text{ contains 9 different values,} \\ C_{\text{col}}: & \text{Each column } j \text{ contains 9 different values,} \\ C_{\text{subgrid}}: & \text{Each } 3 \times 3 \text{ subgrid contains 9 different values.} \end{cases}$$

1.1.2 Time Complexity Analysis

1. **Brute-force Search Algorithm** [2] :
   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 $O(9^k)$. The worst-case scenario is that algorithm needs to explore all possible combinations, leading to the $O(9^{81})$ time complexity.

2. **Backtracking Search Algorithm** [3] :
   This algorithm assigns a number to a cell and immediately checks constraint validity (row, column, and 3×3 sub-grid). If the partial assignment violates any constraint, it backtracks recursively to the previous step and tries a different value, thereby pruning large invalid sub-trees in the search space. Although the theoretical worst-case time complexity is still $O(9^k)$ (similar to brute-force), the practical performance is dramatically faster because the effective branching factor b is greatly reduced by constraint filtering ($b \ll 9$).

1.2 Part 1B

1.2.1 Procedure

1. **CSP Modelling**
   The Sudoku grid is modelled as a 9×9 matrix `sudoku_solver.board`. A candidate assignment is validated by `is_valid(row, col, num)`, which checks Row, Column, and Subgrid constraints.

2. **Input Handling**
   The solver accepts `.csv` files containing a 9×9 grid. `load_from_csv(filepath)` parses comma-separated rows, interprets 0/blank as empty cells, and rejects matrices not equal to 9×9.

```
self.board = []
with open(filepath, 'r', encoding='UTF-8') as f:
    reader = csv.reader(f)
    for row in reader:
        index = [int(num.strip()) for num in row if num.strip().isdigit()]
        if len(index) == 9:
            self.board.append(index)
```

3. **Backtracking Solver**
   Implemented as DFS in `solve_algorithm()`:

   (a) *Variable selection:* scan the grid to locate the next empty cell $V_{i,j}$.

   (b) *Value assignment:* try $d \in \{1, \ldots, 9\}$ sequentially.

   (c) *Pruning:* if `is_valid()` fails, skip deeper recursion.

   (d) *Backtracking:* on failure, reset $V_{i,j}$ to 0 and attempt the next candidate.

```
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 self.is_valid(i, j, num):
                    self.board[i][j] = num

                    if self.solve_algorithm():
                        return True

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

4. **Performance Metrics**
   `run_solver()` records:
   - execution time (via `time.perf_counter()`).
   - number of backtracks.
   - number of recursive calls.

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

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"* [4].

- **Result Display**
  To facilitate user interaction and clearly visualize the solving process, a graphical interface was implemented using the *PyQt6* framework [5]. The figures below demonstrate the artifact's capability to handle puzzles of varying difficulties, displaying both the initial board configuration and the final calculated solution in a clean grid format.

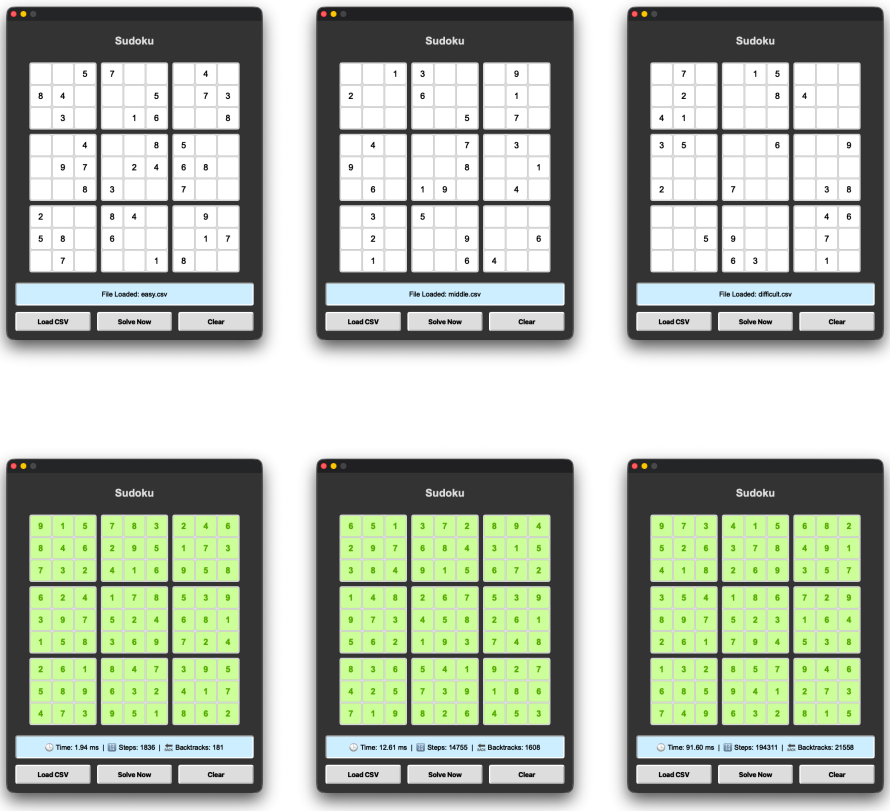

Figure 1: Visualisation of the Sudoku Solver Artefact on different difficulty levels

- **Quantitative Analysis**

Difficulty	Time (ms)	Recursive Calls	Backtracks
Easy	1.94 ms	1,836	181
Middle	12.61 ms	14,755	1,608
Difficult	91.60 ms	194,311	21,558

Table 1: Performance Metrics by Difficulty Level

1.2.3 Theoretical Comparison: Backtracking vs. A* Search

1. ***A* Search for Sudoku***
   Sudoku puzzle can be defined by *A** cost function $f(n) = g(n) + h(n)$:

$$\begin{cases} g(n) \text{ (path cost)} : \text{The number of cells filled so far} \\ h(n) \text{ (heuristic)} : \text{The sum of domain sizes of all empty cells} \end{cases}$$

2. **Comparison table**

Metrics	Backtracking	*A** Search
Data Structure	DFS (stack)	BFS (priority queue)
Time Taken	Faster. Low overhead per step allows checking millions of nodes quickly.	Slower. High overhead due to calculating heuristics $h(n)$ and sorting the queue at every step.
Search Steps	High. May explore many redundant branches before backtracking.	Low. Heuristics guide the search effectively, visiting fewer total nodes.
Memory Usage	Linear $O(D)$	Exponential $O(b^d)$
Failure Case	Can get stuck in a wrong branch for too long on malicious puzzles.	Likely to crash due to running out of RAM before finding a solution.

Table 2: Backtracking vs. A* Search for *Sudoku* Puzzles

3. **Analysis of performance**

   - **Efficiency:** Though *A** visits fewer nodes, the computational cost per node is high due to heuristic calculations and priority queue maintenance. Backtracking checks millions of nodes in milliseconds due to its minimal overhead, resulting in faster overall execution time.
   - **Nature of the Problem:** *A** is optimized for finding the *shortest path* in a graph. However, *Sudoku* is a typical CSP where the solution depth is fixed. We require any valid solution, not the shortest one, making *A**'s path-optimizing features redundant.
   - **Space Complexity:** The exponential memory usage of *A** makes it impractical for hard puzzles, whereas Backtracking's linear space complexity ensures robustness on any environments.

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

References

[1] Wikipedia contributors. ”Constraint satisfaction problem.” Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Constraint_satisfaction_problem

[2] GeeksforGeeks. ”Brute Force Approach and its Pros and Cons.” [Online]. Available: https://www.geeksforgeeks.org/brute-force-approach-and-its-pros-and-cons/

[3] GeeksforGeeks. ”Backtracking Algorithms.” [Online]. Available: https://www.geeksforgeeks.org/backtracking-algorithms/

[4] Sudoku Name. ”Online Sudoku Puzzles.” [Online]. Available: https://www.sudoku.name/

[5] Riverbank Computing. ”PyQt6 Reference Guide.” [Online]. Available: https://www.riverbankcomputing.com/static/Docs/PyQt6/