Sudoku Solver Using Constraint Satisfaction Problem

Kerem Adalı

March 31, 2025

1 Introduction

This report presents a Sudoku solver implemented using the Constraint Satisfaction Problem (CSP) method. The implementation employs backtracking with forward checking to efficiently solve Sudoku puzzles of varying difficulty levels. This document outlines the formulation of the CSP for Sudoku, discusses the implementation details, analyzes the role of forward checking, and presents test cases to demonstrate the solver's performance.

2 Theoretical Background

A Constraint Satisfaction Problem consists of:

- A set of variables $X = \{X_1, X_2, ..., X_n\}$
- A domain for each variable $D_i = \{v_1, v_2, ..., v_m\}$
- A set of constraints C that restrict the values variables can take simultaneously

The goal is to find an assignment of values to variables that satisfies all constraints.

3 CSP Formulation for Sudoku

3.1 Variables

In our Sudoku implementation, the variables are defined as the cells in the 9×9 grid. Each cell is represented by a tuple (i, j) where i is the row index and j is the column index, both ranging from 0 to 8. This results in a total of 81 variables:

```
variables = [(i, j) for i in range(9) for j in range(9)]
```

3.2 Domains

The domain for each variable represents the possible values that can be assigned to the corresponding cell. The domain is defined based on the initial state of the puzzle:

- For empty cells (represented by 0), the domain includes all possible values from 1 to 9.
- For pre-filled cells, the domain contains only the pre-filled value.

This is implemented as follows:

```
domains = {}
for i in range(9):
    for j in range(9):
        if puzzle[i][j] == 0:  # Empty cell
            domains[(i, j)] = list(range(1, 10))
        else:  # Pre-filled cell
            domains[(i, j)] = [puzzle[i][j]]
```

3.3 Constraints

The constraints in Sudoku ensure that no two cells in the same row, column, or 3×3a box contain the same value. For each cell (i, j), we define the constraints as the set of all other cells that cannot have the same value as (i, j):

```
constraints = {}
for i in range(9):
    for j in range(9):
        constraints[(i, j)] = []

        # Same row constraints
        for jj in range(9):
            if jj != j:
                constraints[(i, j)].append((i, jj))

        # Same column constraints
        for ii in range(9):
            if ii != i:
                constraints[(i, j)].append((ii, j))

        # Same 3x3 box constraints
        box_i, box_j = 3 * (i // 3), 3 * (j // 3)
        for ii in range(box_i, box_i + 3):
            for jj in range(box_j, box_j + 3):
                if (ii, jj) != (i, j):
                    constraints[(i, j)].append((ii, jj))
```

The constraints map each variable to a list of variables that it constraints, which forms a constraint graph. In other words, each cell is connected to all other cells in the same row, column, and 3×3 box.

4 Solving Algorithm

The implementation uses backtracking search with forward checking to solve the Sudoku puzzle. The key components of the solution approach are discussed below.

4.1 Backtracking Search

Backtracking search is a depth-first search that recursively tries to extend a partial assignment to a complete solution. If the current assignment fails to satisfy a constraint, the algorithm backtracks by removing the most recently assigned variable and trying an alternative value.

The core of the backtracking algorithm is as follows:

```
def backtrack(self, assignment:dict):
    # If all variables are assigned, we're done
    if len(assignment) == len(self.variables):
        return assignment

    # Select an unassigned variable
    var = self.select_unassigned_variable(assignment)

    # Try assigning each value in var's domain
    for value in list(self.domains[var]):
        # Check if this assignment is consistent
        if self.is_consistent(var, value, assignment):
            # Assign the value
            assignment[var] = value

            # Apply forward checking
            pruned, is_consistent = self.forward_checking(var,
    value, assignment)

            # If assignment is consistent with forward checking
            if is_consistent:
                # Recursive call
                result = self.backtrack(assignment)
                if result is not None:
                    return result

            # If we get here, this assignment failed
            # Remove var from assignment
            del assignment[var]

            # Restore pruned values
            self.restore_pruned(pruned)

    # No solution found with current assignments
    return None
```

4.2 Variable Selection: Minimum Remaining Values (MRV)

The variable selection strategy used is Minimum Remaining Values (MRV), which chooses the variable with the smallest domain. This heuristic helps to

identify the most constrained variables first, potentially reducing the search space:

```python
def select_unassigned_variable(self, assignment:dict):
    # Find variables that aren't assigned yet
    unassigned = [v for v in self.variables if v not in assignment]

    # Select the one with fewest remaining values in its domain
    return min(unassigned, key=lambda var: len(self.domains[var]))
```

4.3 Forward Checking

Forward checking is a constraint propagation technique that removes inconsistent values from the domains of unassigned variables when a variable is assigned. This helps to detect failures early in the search process:

```python
def forward_checking(self, var, value, assignment:dict):
    pruned = {}  # Track pruned values for backtracking

    # For each variable constrained by var
    for neighbor in self.constraints[var]:
        # Skip if already assigned
        if neighbor in assignment:
            continue

        # If the value is in the domain of the neighbor
        if value in self.domains[neighbor]:
            # Track pruning
            if neighbor not in pruned:
                pruned[neighbor] = []
            pruned[neighbor].append(value)

            # Remove the value from domain
            self.domains[neighbor].remove(value)

            # If domain becomes empty, this assignment fails
            if not self.domains[neighbor]:
                return pruned, False

    return pruned, True
```

4.4 Restoring Pruned Values

When backtracking, we need to restore the values that were pruned during forward checking:

```python
def restore_pruned(self, pruned):
    for var, values in pruned.items():
        for value in values:
            if value not in self.domains[var]:
                self.domains[var].append(value)
```

5 Analysis of Forward Checking

Forward checking is a fundamental technique in this Sudoku solver implementation, offering several significant advantages:

5.1 Early Detection of Failures

The most significant benefit of forward checking is the early detection of inconsistent assignments. When a value is assigned to a variable, forward checking immediately removes that value from the domains of all constrained variables. If any domain becomes empty, the algorithm knows that the current assignment cannot lead to a solution and backtracks immediately without exploring further. This prevents the algorithm from wasting time on paths that are guaranteed to fail.

5.2 Domain Reduction

By removing inconsistent values from domains, forward checking reduces the search space for future variable assignments. This is particularly effective in Sudoku, where each cell assignment typically affects 20 other cells (8 in the same row, 8 in the same column, and 4 in the same 3×3 box, excluding overlaps). The reduction in domain sizes can significantly decrease the number of values that need to be tried.

5.3 Effectiveness for Sudoku

Forward checking is especially effective for Sudoku because:

- Sudoku has a dense constraint graph, with each variable being constrained by many others
- Domain sizes are small (initially 9 or fewer values)
- Constraints are binary and relatively simple (all different)

5.4 Performance Improvement

Based on the implementation and test cases, forward checking significantly improves the performance of the Sudoku solver, especially for difficult puzzles. Without forward checking, the algorithm would need to explore a much larger search space, leading to significantly longer solving times.

To quantify this improvement, consider that a standard 9×9 Sudoku grid with no initial filled cells would have a search space of 9^{81} possible configurations if we used simple backtracking without any constraints or optimizations. With constraints and backtracking alone, this is reduced substantially, but forward checking reduces it even further by eliminating obviously inconsistent assignments early.

6 Test Cases

The implementation was tested with various Sudoku puzzles to evaluate its effectiveness and robustness. Here are the key test cases and their rationale:

6.1 Standard Test Case

The standard test case is a moderately difficult Sudoku puzzle:

```
[
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [0, 0, 0, 0, 0, 3, 0, 0, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 6],
    [0, 0, 0, 0, 0, 0, 2, 8, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 8],
    [0, 0, 0, 0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 0, 4, 0, 0]
]
```

This puzzle has 24 filled cells out of 81, providing enough constraints to make the solution deterministic but requiring substantial search.

6.2 Difficult Test Case

A more challenging puzzle with only 17 filled cells (the minimum number of clues for a unique solution):

```
[
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 3, 0, 8, 5],
    [0, 0, 1, 0, 2, 0, 0, 0, 0],
    [0, 0, 0, 5, 0, 7, 0, 0, 0],
    [0, 0, 4, 0, 0, 0, 1, 0, 0],
    [0, 9, 0, 0, 0, 0, 0, 0, 0],
    [5, 0, 0, 0, 0, 0, 0, 7, 3],
    [0, 0, 2, 0, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 4, 0, 0, 0, 9]
]
```

This test case was chosen to challenge the algorithm with a puzzle that has minimal constraints, requiring more search and relying heavily on the effectiveness of forward checking and the MRV heuristic.

6.2.1 Invalid Puzzle (Case 4)

```
[1, 1, 1, 1, 1, 1, 1, 1, 1],
[2, 2, 2, 2, 2, 2, 2, 2, 2],
...
```

This test case is designed to verify that the solver correctly identifies puzzles without valid solutions. The rows filled with identical digits violate the column constraints, making the puzzle unsolvable. This tests the algorithm's ability to detect impossibility rather than running indefinitely.

6.2.2 Already Solved Puzzle (Case 5)

A completely filled, valid Sudoku grid tests whether the algorithm can recognize a solution without any additional processing. This verifies the base case of the recursion and ensures that trivial cases are handled efficiently.

6.2.3 Minimal Clues Puzzle (Case 6)

A puzzle with very few given values tests the algorithm's ability to solve highly underconstrained problems. This is useful for:

- Stress testing the performance of the backtracking algorithm
- Evaluating how well forward checking reduces the search space in sparse puzzles
- Testing the effectiveness of the MRV heuristic when many variables have large domains

6.2.4 Empty Grid (Case 7)

An entirely empty grid represents the most unconstrained scenario. This tests whether the algorithm can generate a valid Sudoku from scratch. While numerous valid solutions exist, this case tests the algorithm's ability to find at least one of them efficiently.

6.2.5 Single Missing Cell (Case 8)

A puzzle with only one empty cell tests the algorithm's efficiency on nearly-constrained problems. With only one variable to assign and its domain potentially reduced to a single value by constraints, this tests the best-case scenario and verifies that the algorithm doesn't unnecessarily explore the search space.

6.2.6 Multiple Solutions Possible (Case 9)

A puzzle with multiple valid solutions tests whether the algorithm returns the first solution it finds rather than exploring the entire search space unnecessarily.

7 Results and Visualization

The implementation includes visualization capabilities that allow us to observe the solving process step by step. This is particularly useful for understanding the algorithm's behavior and debugging.

The visualization shows the state of the grid after each variable assignment, providing insight into how the backtracking and forward checking algorithms progress through the solution space.

7.1 Visualization Approach

The visualization is implemented by creating a snapshot of the current grid state at each step of the backtracking algorithm:

```
def create_grid_snapshot(self, assignment:dict):
    # Create empty grid
    grid = [[0 for _ in range(9)] for _ in range(9)]

    # Fill in values from the assignment
    for (i, j), value in assignment.items():
        grid[i][j] = value

    return grid
```

There is also an option to create grid snapshots of cell domains.

```
     def create_grid_snapshot_with_domains(self, domains:dict):
        """
        Creates a grid snapshot for visualization with domains
        """
        # Create empty grid
        grid = [[0 for _ in range(9)] for _ in range(9)]

        # Fill in values from the domains
        for (i, j), values in domains.items():
            if len(values) == 1:
                grid[i][j] = values[0]
            else:
                grid[i][j] = values
        return grid
```

These snapshots are stored in a list and displayed after the solution is found, showing the progression of the algorithm:

```
def visualize(self,viz_domains=False):
        """Visualizes the steps of the solving algorithm"""
        if viz_domains:
            print("\n" + "*" * 7 + " Visualization with Domains " +
    "*" * 7)
            for step, grid in enumerate(self.viz_domains):
                print(f"\nStep {step+1}:")
                self.print_sudoku(grid)
                print("-" * 30)
        else:
            print("\n" + "*" * 7 + " Visualization " + "*" * 7)
            for step, grid in enumerate(self.viz):
                print(f"\nStep {step+1}:")
                self.print_sudoku(grid)
                print("-" * 30)
```

8 Conclusion

The Sudoku solver implemented using CSP with backtracking and forward checking demonstrates the effectiveness of these techniques for solving constraint satisfaction problems. The key findings from this implementation are:

- The CSP formulation provides a natural and efficient representation of the Sudoku problem.
- Forward checking significantly improves the performance of the solver by detecting inconsistencies early and reducing the search space.
- The MRV heuristic for variable selection further enhances efficiency by focusing on the most constrained variables first.
- The solver successfully handles puzzles of varying difficulty, including those with minimal clues.