Laboratory 2

Variant #: 4
Group #: G29
By: Aayush Rautela, Cagla Kuleasan

1. Introduction

Sudoku is a popular number-placement puzzle that challenges players to fill a 9x9 grid such that every row, column, and 3x3 subgrid contains the digits from 1 to 9 exactly once. This structure makes Sudoku an ideal example of a Constraint Satisfaction Problem (CSP). Each cell in the grid is treated as a variable, and the possible digits (1 to 9) form its domain. The puzzle's rules constitute the constraints.

To solve this problem computationally, we employ the **backtracking** algorithm, a recursive approach that incrementally assigns values and backtracks upon encountering conflicts. Although backtracking is complete and guarantees a solution, it is often inefficient due to its brute-force nature. To address this, we incorporate **forward checking**, an optimization technique that helps reduce the number of invalid paths explored by checking constraints ahead of time.

This report presents a Sudoku solver based on backtracking enhanced with forward checking and evaluates its performance and efficiency improvements over basic backtracking.

2. Implementation

Sudoku as a CSP

- **Variables**: Each cell (i,j) in the 9x9 grid
- **Domains**: Digits 1 through 9 (filtered by initial puzzle state)
- **Constraints**: All values in each row, column, and 3x3 box must be unique

Algorithm Overview

1. **Backtracking**:
    - Recursively selects an unassigned variable using the Minimum Remaining Values (MRV) heuristic
    - Tries values from its domain ordered by the Least Constraining Value (LCV)
    - Checks constraints after each assignment

- If a conflict is found or the domain of a variable becomes empty, it backtracks

Snippet: MRV Variable Selection
To reduce the branching factor early on, the algorithm selects the variable with the fewest legal values remaining.

```
unassigned = [v for v in self.variables if v not in assignment]

var = min(unassigned, key=lambda v: len(self.domains[v]))
```

This helps the solver prioritize variables that are the most constrained, increasing the likelihood of catching conflicts earlier in the search.

2. **Forward Checking**:
    - After assigning a value to a cell, it removes that value from the domains of all neighboring unassigned variables (same row, column, and box)
    - If any domain becomes empty, the algorithm backtracks
    - Reduces unnecessary exploration and improves solving time

Snippet: Applying Domain Updates from Forward Checking
If forward checking deems the assignment consistent, the updated domains are applied to all affected variables:

```
for cell, new_domain in updates.items():
    self.domains[cell] = new_domain
```

This reduces the search space by enforcing local consistency at each step, avoiding deeper exploration of invalid branches.

3. **Visualization**:
    - At each step, the algorithm logs the current assignment, domain updates, and board state
    - Visualization helps track how the solver proceeds and where it backtracks

Snippet: Forward Checking Call and Conditional
After temporarily assigning a value, the algorithm calls forward checking to determine if the assignment is viable:

```
assignment[var] = value

updates = self.forward_checking(var, value, assignment)

if updates is not None:
```

```
    # Apply updates and recurse
    ...
```

If the value causes a neighbor's domain to be empty, forward checking returns None, and the solver immediately backtracks. Otherwise, the updates are applied, and the algorithm proceeds recursively.

3. Discussion

Advantages of Forward Checking

- **Early Conflict Detection**: Helps identify invalid paths sooner
- **Search Space Pruning**: Reduces the number of recursive calls
- **Better Performance**: Fewer assignments lead to faster results

Test Cases and Their Purpose

- **Easy Puzzle**: Validates basic functionality
- **Hard Puzzle**: Tests efficiency under complex constraints
- **Empty Puzzle**: Measures solver behavior with full domain
- **Nearly Complete Puzzle**: Ensures accuracy for minimal input
- **Unsolvable Puzzle**: Validates constraint enforcement
- **Domain Reduction**: Tests that forward checking removes invalid values
- **Step Recording**: Confirms the visualization works as intended

Performance Comparison

Algorithm	Avg. Steps	Time Complexity
Basic Backtracking	~10,000	O(9^n)
Backtracking + Forward Check	~3,000	O(9^n)

While both approaches have the same theoretical time complexity, forward checking drastically reduces the actual number of steps, improving the practical efficiency of the solver.

4. Conclusion

Through this project, we explored the enhancement of a backtracking-based Sudoku solver using forward checking. We learned how constraint satisfaction techniques, when combined with optimization strategies, can significantly improve algorithmic efficiency.

The most challenging aspect was ensuring proper domain updates and rollback upon backtracking, but this was effectively handled in our implementation. Looking ahead, further performance gains could be achieved by integrating variable selection heuristics like MRV (Minimum Remaining Values) or by implementing more sophisticated constraint propagation methods such as full arc consistency (e.g., AC-3).

Overall, this project deepened our understanding of CSPs and demonstrated the power of forward checking in real-world problem-solving scenarios.