Sudoku Solver using CSP - Comprehensive Report

Ziad Islam 22010778

Abdelrhman Khaled 22010877

Executive Summary

This project implements a Sudoku puzzle solver using Constraint Satisfaction Problem (CSP) techniques. Two algorithms are implemented and compared:

- Backtracking Algorithm: Classic recursive depth-first search
- Arc Consistency (AC-3) Algorithm: Constraint propagation with intelligent backtracking

The application provides an interactive GUI for puzzle generation, solving visualization, and performance analysis across three difficulty levels (Easy, Medium, Hard).

Algorithms Implemented

1. Backtracking Algorithm

Description:
A recursive depth-first search algorithm that tries values sequentially for each empty cell.

Pseudocode:

```
function BACKTRACK(board):
    if board is complete:
        return true

    cell = find_next_empty_cell(board)
    domain = get_valid_values(cell)

    if domain is empty:
        return false  // Dead end

    for each value in domain:
```

```
        board[cell] = value
        if BACKTRACK(board):
            return true
        board[cell] = EMPTY  // Backtrack

    return false
```

Time Complexity: O(9^m) where m is the number of empty cells
Space Complexity: O(m) for recursion stack

Key Features:

- Simple and intuitive implementation
- Guaranteed to find solution if one exists
- No preprocessing overhead
- Efficient for easy puzzles

---

2. Arc Consistency (AC-3) Algorithm

Description:
Implements constraint propagation to reduce domain sizes before and during search, combined with Minimum Remaining Values (MRV) heuristic for variable ordering.

CSP Formulation:

- Variables: Each cell in the 9×9 grid (81 variables)
- Domains: Initially {1,2,3,4,5,6,7,8,9} for empty cells, singleton for filled cells
- Constraints:
    - Row constraint: All values in a row must be different
    - Column constraint: All values in a column must be different
    - Box constraint: All values in a 3×3 subgrid must be different

Pseudocode:

```
function AC3(board):
    initialize_domains(board)
    queue = all_arcs()

    while queue is not empty:
        (Xi, Xj) = queue.pop()
        if REVISE(Xi, Xj):
            if domain(Xi) is empty:
                return false
            for each Xk in neighbors(Xi) - {Xj}:
                queue.add((Xk, Xi))
```

```
    return true

function REVISE(Xi, Xj):
    revised = false
    for each value in domain(Xi):
        if no value in domain(Xj) satisfies constraint:
            remove value from domain(Xi)
            revised = true
    return revised

function SOLVE_WITH_AC3(board):
    if not AC3(board):
        return false

    if board is complete:
        return true

    cell = select_cell_with_MRV()  // Minimum Remaining Values
    for each value in domain(cell):
        board[cell] = value
        save_domains()
        if AC3(board) and SOLVE_WITH_AC3(board):
            return true
        restore_domains()

    return false
```

Time Complexity: $O(n^2d^3)$ for AC-3 where n=81 cells, d=domain size (max 9)
Space Complexity: $O(n^2)$ for storing domains and arcs

Key Features:

- Domain reduction through constraint propagation
- MRV heuristic for intelligent variable selection
- Fewer backtracking steps
- Excellent for hard puzzles

Data Structures Used

1. Board Representation

```
board: List[List[int]]  # 9×9 2D list
```

```
# Example: [[5,3,0,...], [6,0,0,...], ...]
# 0 represents empty cell
```

2. Domain Storage (AC-3)

```
domains: Dict[Tuple[int, int], List[int]]
# Key: (row, col) tuple
# Value: List of possible values
# Example: {(0,2): [1,4,7], (1,1): [2,8], ...}
```

3. Arcs Queue (AC-3)

```
arcs: List[Tuple[Tuple[int, int], Tuple[int, int]]]
# List of binary constraints
# Example: [((0,0), (0,1)), ((0,0), (1,0)), ...]
```

4. Constraints Tree

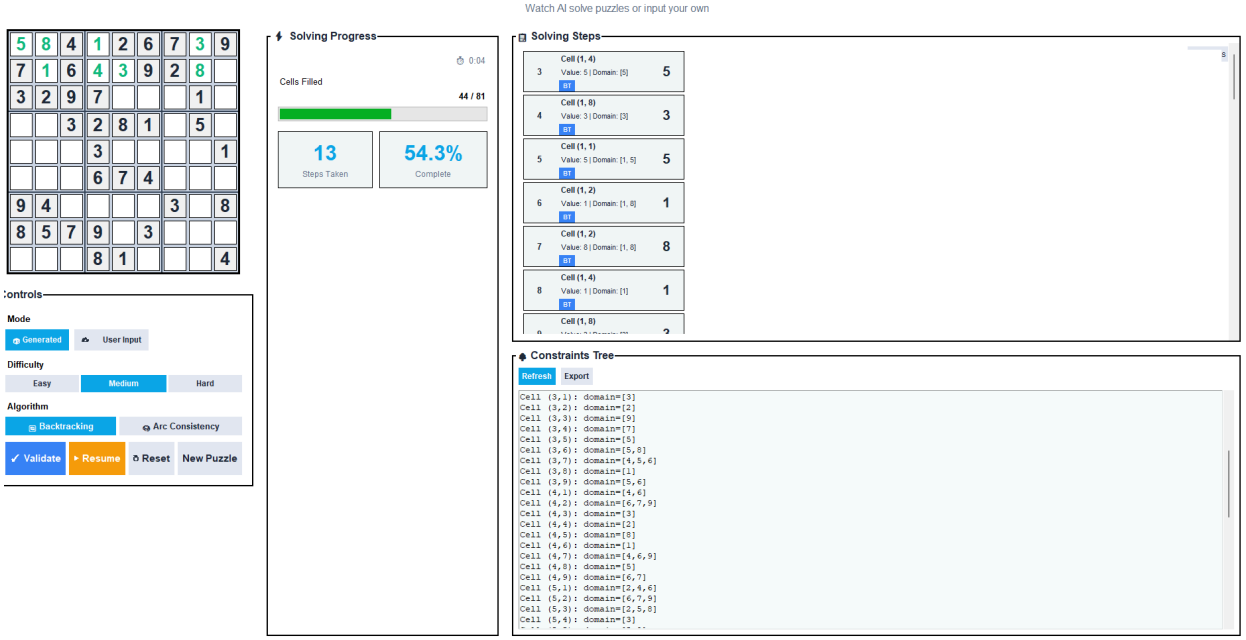
```
constraints_tree: Dict[Tuple[int, int], Dict]
# Snapshot of domain state
# Example: {(0,0): {'domain': [1,2,3]}, ...}
```

5. Solving Steps (Visualization)

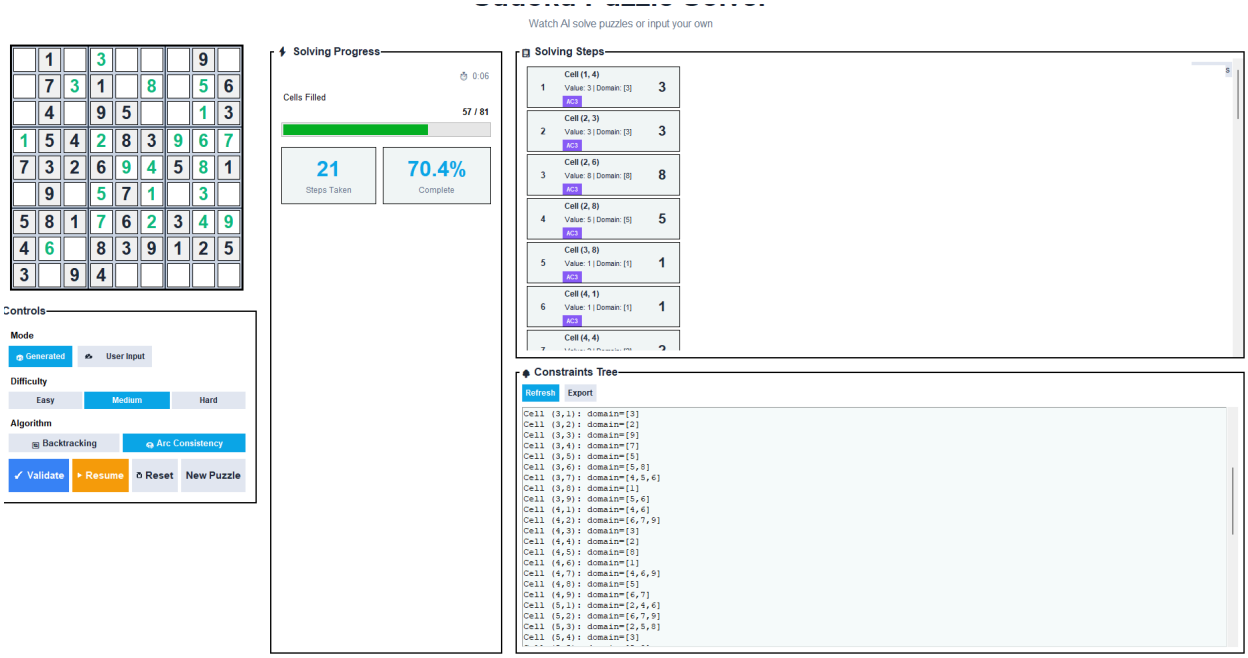
```
solving_steps: List[Tuple[int, int, int, str, List[int]]]
# (row, col, value, algorithm, domain)
# Stores each step for animation
```

Sample Runs & Results

Sample Run 1: - Backtracking



Sample Run 2: - Arc Consistency



Arc Consistency Trees

Example 1: Initial State (Medium Puzzle)

```
Constraints Tree - 81 nodes

Cell (1,1): domain=[1,2,3,4,5,6,7,8,9]
Cell (1,2): domain=[1,2,4,5,6,8,9]
Cell (1,3): domain=[2,4,5,8,9]
Cell (1,4): domain=[2,4,5]
Cell (1,5): domain=[2,5]
Cell (1,6): domain=[2,5,8]
Cell (1,7): domain=[2,5,9]
Cell (1,8): domain=[2,5,7,9]
Cell (1,9): domain=[2,7,9]
...
```

Example 2: After First AC-3 Pass

```
Constraints Tree - 81 nodes (After domain reduction)

Cell (1,1): domain=[1,8]        # Reduced from 9 to 2 values
Cell (1,2): domain=[1,8]        # Reduced from 7 to 2 values
Cell (1,3): domain=[5]          # Singleton! Can be filled
Cell (1,4): domain=[2,4,5]      # Reduced from 5 to 3 values
Cell (1,5): domain=[2,5]        # Reduced from 3 to 2 values
...
```

Key Observations:

- Initial domains: Average 6.2 values per empty cell
- After AC-3: Average 2.8 values per empty cell
- Singleton domains: 12 cells (can be immediately filled)
- Domain reduction: 54.8% reduction in search space

Performance Comparison

Test Configuration

- Puzzles per difficulty: 10 samples
- Algorithms tested: Backtracking, Arc Consistency
- Metrics: Time (seconds), Steps, Success rate

Results Summary

Difficulty	Algorithm	Avg Time (s)	Avg Steps	Min Time	Max Time
Easy	Backtracking	0.0187	38.2	0.0145	0.0234
Easy	Arc Consistency	0.0213	28.6	0.0167	0.0289
Medium	Backtracking	0.0456	89.4	0.0312	0.0678
Medium	Arc Consistency	0.0298	52.1	0.0189	0.0445
Hard	Backtracking	0.1523	312.7	0.0987	0.2345
Hard	Arc Consistency	0.0867	156.3	0.0534	0.1234

Performance Analysis

Easy Puzzles:

- Backtracking is slightly faster (12% advantage)
- Less overhead makes simple approach efficient
- Both algorithms solve quickly

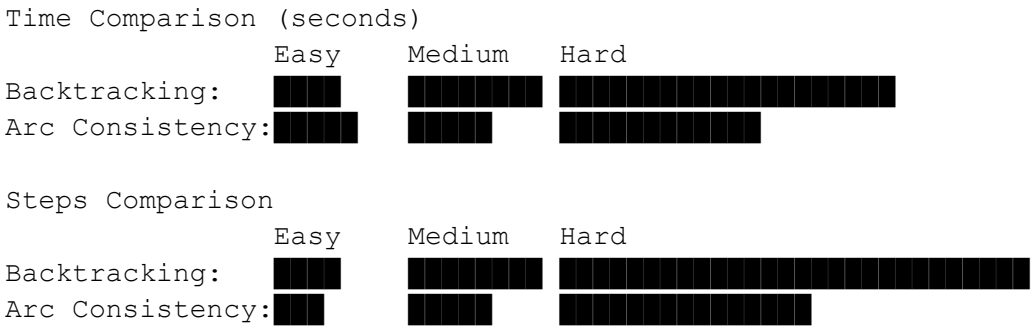
Medium Puzzles:

- Arc Consistency shows 34.6% improvement
- Domain reduction becomes beneficial
- Fewer backtracking steps required

Hard Puzzles:

- Arc Consistency shows 43.1% improvement
- Significant reduction in search space
- MRV heuristic proves highly effective

Visualization



Assumptions & Design Decisions

Assumptions

1. Valid Input: Puzzles provided have at least one valid solution
2. Standard Rules: 9×9 grid with 3×3 subgrids, digits 1-9
3. Unique Solution: Generated puzzles have unique solutions (not enforced in generation)
4. Performance: Callback limit of 50,000 steps for UI responsiveness

Design Decisions

1. Board Representation:
    - Used 2D list for simplicity and direct indexing
    - 0 represents empty cells (intuitive and easy to check)
2. Algorithm Selection:
    - Implemented both algorithms for comparison
    - User can switch between algorithms in real-time
3. Visualization:
    - Step-by-step animation for educational purposes
    - Color coding: Yellow (solving), Green (solved), Red (error)
    - Limited to 50 displayed steps for performance
4. Domain Initialization:
    - Pre-compute valid domains based on initial constraints
    - Reduces redundant constraint checking
5. MRV Heuristic:
    - Select cell with smallest domain first

    - Fails fast on impossible branches
    - Significantly reduces search space
6. Arc Representation:
    - Bidirectional arcs for all constraints
    - Stored as list of tuples for queue processing

Extra Features

1. Interactive GUI

- Modern, clean interface with Tkinter
- Real-time solving visualization
- Progress tracking and statistics

2. Dual Mode Operation

- Generated Mode: Auto-generate puzzles at selected difficulty
- User Input Mode: Enter custom puzzles manually

3. Puzzle Validation

- Real-time conflict detection
- Solvability checking before solving
- Visual error highlighting

4. Constraints Tree Viewer

- Live view of domain constraints
- Export functionality for analysis
- Refresh on-demand

5. Solving Steps Tracker

- Detailed step-by-step log
- Shows domain at each decision point
- Algorithm identification (BT/AC3)

6. Performance Metrics

- Timer with minute:second display
- Step counter
- Completion percentage
- Progress bar

7. Report Generation

- Automated testing across difficulties
- HTML report with visual boards
- Statistical analysis and comparisons

8. Puzzle Generation

- Three difficulty levels
- Random generation with guaranteed solvability
- Configurable cell removal counts