# Kropki Sudoku Solver

Tewoflos Girmay, Hariharan Janardhanan

December 16, 2024

## 1 Problem Description

The Kropki Sudoku is a variant of the traditional Sudoku puzzle with additional constraints introduced by black and white dots between cells:

- White Dot: Indicates that the two connected numbers differ by 1
- Black Dot: Indicates that one number is exactly double the other

The standard Sudoku rules still apply:

1. Each row must contain the numbers 1-9 without repetition
2. Each column must contain the numbers 1-9 without repetition
3. Each 3x3 sub-grid must contain the numbers 1-9 without repetition

## 2 Implementation Strategy

The solver employs a backtracking algorithm with several advanced techniques:

### 2.1 Constraint Propagation

- Minimum Remaining Values (MRV) heuristic for variable selection
- Forward checking to reduce search space
- Domain reduction based on Kropki dot constraints

### 2.2 Algorithm Outline

1. Select the most constrained unassigned cell
2. Try values within the cell's domain
3. Validate against Sudoku and Kropki constraints
4. Recursively solve the puzzle
5. Backtrack if no solution is found

## 3 Running the Program

To run the Kropki Sudoku solver, use the following command:

```
python3 kropki.py <input_file> <output_file>
```

Example:

```
python3 kropki.py Input1.txt solution1.txt
```

## 4 Input File Format

The input file consists of three sections:

1. First 9 lines: Initial Sudoku board (0 represents empty cells)
2. Blank line
3. Next 9 lines: Horizontal dot constraints
4. Blank line
5. Last 8 lines: Vertical dot constraints

## 5 Input 1

```
0 8 0 0 0 0 0 3 0
2 5 0 0 0 0 0 8 6
0 0 3 0 0 0 9 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 8 0 0 0 6 0 0
6 3 0 0 0 0 0 7 5
0 9 0 0 0 0 0 1 0

1 0 0 1 0 0 0 1
0 1 1 0 0 0 0 0
1 2 0 0 2 1 0 0
0 0 0 0 2 1 2 0
0 0 0 0 0 0 1 0
1 0 0 0 0 0 0 0
0 0 1 0 0 1 0 0
2 1 2 0 0 0 1 0
0 0 0 0 0 0 0 0

0 0 0 0 1 0 0 0 0
0 1 1 0 0 1 0 0 0
0 1 0 0 2 2 0 1 0
0 0 0 0 2 1 0 0 1
0 2 0 0 0 0 0 0 0
1 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0
1 0 0 2 1 0 2 0 0
```

## 6 Output 1

```
9 8 1 5 6 2 7 3 4
2 5 4 3 7 9 1 8 6
7 6 3 1 4 8 9 5 2
1 7 5 9 2 4 3 6 8
```

```
8 2 9 6 1 3 5 4 7
3 4 6 8 5 7 2 9 1
4 1 8 7 3 5 6 2 9
6 3 2 4 9 1 8 7 5
5 9 7 2 8 6 4 1 3
```

## 7 Input 2

```
0 0 0 1 0 0 8 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 3 0 6
4 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 4
7 0 2 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 9 0 0 7 0 0 0

0 2 0 0 0 0 0 0
0 0 0 1 0 2 2 0
0 0 2 0 0 1 0 0
1 0 0 0 0 0 1 0
0 0 0 0 1 2 0 1
0 0 0 0 0 0 0 1
0 2 0 0 0 0 0 0
2 0 2 2 0 0 0 1
0 1 0 0 0 1 0 2

0 0 1 0 0 0 2 0 0
0 0 0 1 1 0 1 1 0
0 0 0 0 0 0 0 0 0
2 0 0 0 1 0 0 0 0
0 0 1 1 0 0 0 2 1
0 0 0 0 0 0 0 0 1
0 0 2 0 2 0 0 0 0
0 0 0 1 0 0 1 0 0
```

## 8 Output 2

```
6 2 4 1 9 3 8 5 7
1 7 3 5 6 8 4 2 9
9 5 8 4 7 2 3 1 6
4 3 5 7 2 6 9 8 1
8 1 7 9 3 4 2 6 5
2 9 6 8 5 1 7 3 4
7 4 2 6 8 5 1 9 3
3 6 1 2 4 9 5 7 8
5 8 9 3 1 7 6 4 2
```

## 9 Input 3

```
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0

0 2 0 2 0 0 1 0
1 0 1 0 2 1 0 0
1 2 0 0 0 1 0 0
1 2 0 2 0 2 0 0
0 2 0 0 1 0 0 1
0 0 0 0 0 1 0 0
0 0 0 1 0 0 0 1
0 0 0 0 2 0 0 1
1 0 1 0 0 1 0 0

0 0 1 0 0 0 1 0 1
0 0 1 1 0 1 1 0 0
0 2 2 0 1 1 2 1 0
1 2 0 0 1 0 0 0 0
```

```
2 0 0 0 0 0 1 0 2
0 0 0 1 0 0 1 0 1
0 1 0 0 1 0 0 2 0
0 0 1 0 0 1 1 0 1
```

## 10 Output 3

```
7 3 6 1 2 9 4 5 8
2 1 5 6 8 4 3 9 7
9 8 4 7 5 3 2 6 1
5 4 8 3 6 2 1 7 9
6 2 1 9 7 8 5 3 4
3 9 7 4 1 5 6 8 2
8 6 9 5 4 1 7 2 3
1 7 2 8 3 6 9 4 5
4 5 3 2 9 7 8 1 6
```

## 11 Constraint Satisfaction Problem Formulation

### 11.1 Problem Variables

The Kropki Sudoku is formulated as a Constraint Satisfaction Problem (CSP) with the following key components:

- **Variables:** $X = \{x_{ij} \mid 0 \leq i, j < 9\}$
  - Each $x_{ij}$ represents a cell in the 9x9 Sudoku grid
  - $x_{ij}$ corresponds to the value at row $i$, column $j$
- **Variable Domains:** $D = \{d_{ij} \mid 0 \leq i, j < 9\}$
  - Initially $d_{ij} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ for empty cells
  - Pre-filled cells have a domain of their specific value
  - Domains are dynamically reduced during solving

## 11.2 Constraint Types

The problem involves multiple constraint categories:

1. **Standard Sudoku Constraints:**
   - *Row Constraint:* $\forall i, k \in [0, 8], j_1 \neq j_2 : x_{ij_1} \neq x_{ij_2}$
   - *Column Constraint:* $\forall j, k \in [0, 8], i_1 \neq i_2 : x_{i_1 j} \neq x_{i_2 j}$
   - *3x3 Block Constraint:* Within each 3x3 sub-grid, no number can repeat
2. **Kropki Dot Constraints:**
   - *White Dot Constraint:* $|x_{ij} - x_{k,l}| = 1$
   - *Black Dot Constraint:* $x_{ij} = 2 \times x_{k,l}$ or $x_{k,l} = 2 \times x_{ij}$

## 11.3 Constraint Propagation Strategies

To efficiently solve the CSP:

- **Forward Checking:**
  - Immediately eliminate values from unassigned variables' domains
  - Prune domains based on current partial assignment
  - Detect and terminate inconsistent branches early
- **Domain Reduction Techniques:**
  - Dynamically update domains considering:
    1. Standard Sudoku rules
    2. Kropki dot constraints between adjacent cells
    3. Current partial solution state

## 11.4 Constraint Satisfaction Algorithm

The solving process follows these key steps:

1. Select most constrained unassigned variable
2. Try values from reduced domain

3. Apply forward checking
4. Recursively solve or backtrack
5. Restore domains on backtracking

### 11.5 Complexity Analysis

- **Worst-case Time Complexity:** $O(9^n)$, where $n$ is number of empty cells
- **Space Complexity:** $O(n)$ for recursion and domain tracking
- Practical performance improved by:
  - Minimum Remaining Values (MRV) heuristic
  - Aggressive domain pruning
  - Early constraint checking

## 12 Key Components

### 12.1 Constraint Validation

The `is_valid_assignment` method checks:

- Row constraint
- Column constraint
- 3x3 block constraint
- Horizontal and vertical Kropki dot constraints

### 12.2 Variable Selection

The `select_unassigned_variable` method uses:

- Minimum Remaining Values (MRV)
- Degree heuristic as a tie-breaker

## 13 Code Snippet: Constraint Validation

```
def is_valid_assignment(self, row: int, col: int, num: int) ->
    bool:
    # Check standard Sudoku constraints
    if num in self.board[row] or \
       num in [self.board[i][col] for i in range(9)]:
        return False

    # Check 3x3 block constraint
    start_row, start_col = 3 * (row // 3), 3 * (col // 3)
    for r in range(start_row, start_row + 3):
        for c in range(start_col, start_col + 3):
            if self.board[r][c] == num:
                return False

    # Check Kropki dot constraints
    # (implementation details...)

    return True
```

## 14 Conclusion

The Kropki Sudoku solver demonstrates an effective combination of backtracking and constraint propagation techniques to solve complex constraint satisfaction problems.