CP468 Assignment 2

Sudoku CSP

Group 8

November 9, 2024

Team Members:

Jenish Bharucha, Romin Gandhi, Nakul Patel, Arsh Patel, Dhairya Patel
Paarth Bagga, Devarth Trivedi, Gleb Silin, Emmet Currie, Parker Riches

1 Problem Statement

This report demonstrates the application of AC-3 algorithims to enforce arc-consistency on an arbitrary 9 x 9 Sudoku puzzle. This 9 x 9 Sudoku puzzle will be structured as a Constraint Satisfaction Problem (CSP). The AC-3 algorithm implementation keeps track of and reports the length of the queue at every step taken by the AC-3 algorithm. After the code execution, it will report whether the equivalent arc-consistent CSP is found. In the instance that the problem is solved, the solution is reported. In the event that the puzzle is not solved, an additional algorithim (backtracking) will be used to solve the puzzle entirely and the solution is reported.

2 CSP Representation:

Basic Game Knowledge: Sudoku is played by assigning numbers ranging from 1-9 on a 9 x 9 grid. This grid is divided into nine 3x3 subgrids/units. The numbers must be assigned in a way that each number only appears once in every row, column, and one time in each of the 9 subgrids.

CSP Elements:

- **Variables (X):** Each one of the 81 squares, denoted as an x,y pair that corresponds to the coordinate of the square on the 9 x 9 Sudoku board, the values range from x: 0-8, and y: 0-8.
- **Domains (D):** Each variable has a domain which represents its possible values.
- **Constraints (C):** The rules discussed above are represented as constraints in an array, each variable is stored by its x,y coordinate pair and each number must be unique within its row, column and sub-grid. No two cells in the same row can have the same number.

Constraints (In Depth): As discussed in class, a 9x9 Sudoku has 27 AllDiff constraints, 1 for each column, 1 for each row and 1 for each box. For Example:
Row: AllDiff(A1,A2,A3,A4,A5,A6,A7,A8,A9)
Column: AllDiff(A1,B1,C1,D1,E1,F1,G1,H1,I1)
Box: AllDiff(A1,A2,A3,B1,B2,B3,C1,C2,C3)

These constraints can be decomposed into individual binary constraints for each cell. These will include constraints for every other cell in the row, column and box with respect to each cell. As a general example the cell A1 would have Binary constraints:

Row: A1 != A2
A1 != A3
A1 != A4
A1 != A5
A1 != A6
A1 != A7
A1 != A8
A1 != A9

Column: A1 != B1
A1 != C1
A1 != D1
A1 != E1
A1 != F1
A1 != G1
A1 != H1
A1 != I1

Box: A1 != A2 (Repeated for example)
A1 != A3 (Repeated for example)
A1 != B1 (Repeated for example)
A1 != B2
A1 != B3
A1 != C1(Repeated for example)
A1 != C2
A1 != C3

This process can be repeated for every cell to create the binary constraints. As noted in the example, some of the constraints are repeated since they are already included in the previous row or column constraints.

Based on the example shown in class we can reduce the domain of cells based on already filled in/confirmed values to ensure arc consistency. Since there is only 1 solution to a Sudoku any value that is either pre filled or confirmed based on a domain of size 1 is absolute and therefore can be used to reduce the domain of related cells to ensure arc consistency. For example, cell E4:

	1	2	3	4	5	6	7	8	9
A	-	-	3	-	2	-	6	-	-
B	9	-	-	3	-	5	-	-	1
C	-	-	1	8	-	6	4	-	-
D	-	-	8	1	-	2	9	-	-
E	7	-	-	-	-	-	-	-	8
F	-	-	6	7	-	8	2	-	-
G	-	-	2	6	-	9	5	-	-
H	8	-	-	2	-	3	-	-	9
I	-	-	5	-	1	-	3	-	-

Relevant filled-in cells: Row:
E1 = 7
E9 = 8

Column:
B4 = 3
C4 = 8
D4 = 1
F4 = 7
G4 = 6
H4 = 2

Since these values are filled-in to cells within either the Row, Column or Box of E4 they can be removed from the domain of E4.

Initial Domain:
DE4 = (1,2,3,4,5,6,7,8,9)

Rows:
Within Row E the values 7 and 8 are already filled and can therefore be removed from the domain of E4. Now the domain of E4 is:

DE4 = (1,2,3,4,5,6,9)

Column:
The same concept can be applied to the column 4, which already has 3,8,1,7,6,2 filled in, 7 and 8 have already been removed from the domain but the other values can now be removed as well: Now the domain of E4 is:

DE4 = (4,5,9)

Box:
The same concept can also be applied to the box E4 is in. In this case the box contains 1,2,7,8. However, these values have already been removed from the domain of E4 in previous steps, meaning the domain remains:

DE4 = (4,5,9)

3 Implementation

```python
from collections import deque

# Helper function to read Sudoku from a file
def read_puzzle(file_path):
    with open(file_path, 'r') as f:
        return [[int(num) for num in line.split()] for line in f]

# Helper function to print Sudoku grid
def print_board(board):
    for row in board:
        print(" ".join(str(num) if num != 0 else '.' for num in row))
    print()

# Get all peers for a given cell (row, col)
def get_peers(row, col):
    peers = set()
    for i in range(9):
        peers.add((row, i))  # Same row
        peers.add((i, col))  # Same column
    # Same 3x3 box
    box_row, box_col = 3 * (row // 3), 3 * (col // 3)
    for i in range(box_row, box_row + 3):
        for j in range(box_col, box_col + 3):
            peers.add((i, j))
    peers.discard((row, col))  # Remove the cell itself
    return peers

# Initialize domains for all cells
def initialize_domains(board):
    domains = {}
    for row in range(9):
        for col in range(9):
            if board[row][col] == 0:
                domains[(row, col)] = set(range(1, 10))
            else:
                domains[(row, col)] = {board[row][col]}
    return domains

# AC-3 algorithm implementation
def ac3(board, domains):
    queue = deque([(cell, peer) for cell in domains for peer in get_peers(*cell)])
    step = 1  # Step counter

    while queue:
        print(f"Step {step}: Queue length: {len(queue)}")  # Print the step and queue length
        cell, peer = queue.popleft()
```

```python
        if revise(domains, cell, peer):
            if not domains[cell]:  # Domain is empty -> failure
                return False
            if len(domains[cell]) == 1:
                board[cell[0]][cell[1]] = next(iter(domains[cell]))  # Update board
            for neighbor in get_peers(*cell) - {peer}:
                queue.append((neighbor, cell))
        step += 1  # Increment step counter
    return True

# Revise function to enforce arc-consistency
def revise(domains, cell, peer):
    revised = False
    if len(domains[peer]) == 1:
        peer_value = next(iter(domains[peer]))
        if peer_value in domains[cell]:
            domains[cell].remove(peer_value)
            revised = True
    return revised

# Function to find an empty cell
def find_empty(board):
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                return i, j
    return None

# Backtracking algorithm to solve Sudoku if AC-3 doesn't fully solve
def backtrack(board, domains):
    empty = find_empty(board)
    if not empty:
        return True  # No empty cells left, puzzle solved
    row, col = empty
    for value in sorted(domains[(row, col)]):
        if is_consistent(board, row, col, value):
            board[row][col] = value
            # Create a copy of domains for backtracking
            new_domains = {key: domains[key].copy() for key in domains}
            new_domains[(row, col)] = {value}

            if backtrack(board, new_domains):
                return True
            board[row][col] = 0  # Undo assignment
    return False

# Function to check if a value can be placed in a cell without conflicts
def is_consistent(board, row, col, val):
    for peer_row, peer_col in get_peers(row, col):
        if board[peer_row][peer_col] == val:
            return False
    return True
```

```python
# Solve function
def solve_sudoku(file_path):
    board = read_puzzle(file_path)
    print("Original Sudoku:")
    print_board(board)

    domains = initialize_domains(board)
    if ac3(board, domains):
        print("Sudoku after AC-3:")
        print_board(board)
        if find_empty(board):
            print("AC-3 did not completely solve the puzzle.
                Applying backtracking...")
            if not backtrack(board, domains):
                print("No solution exists.")
        else:
            print("Solved using AC-3!")
    else:
        print("AC-3 failed to achieve arc-consistency.")

    print("Final Solution:")
    print_board(board)

# Example usage
solve_sudoku('test_file.txt')
```

4 Input and Outputs

Scenario 1: Can be solved by AC-3 NOTE: The top 10 and bottom 10 steps are shown due to the sheer quantity of steps needed. A better example will be provided during live demo of code.

```
    Original Sudoku:
5 3 . . 7 . . . .
6 . . 1 9 5 . . .
. 9 8 . . . . 6 .
8 . . . 6 . . . 3
4 . . 8 . 3 . . 1
7 . . . 2 . . . 6
. 6 . . . . 2 8 .
. . . 4 1 9 . . 5
. . . . 8 . . 7 9

Step 1: Queue length: 1620
Step 2: Queue length: 1619
Step 3: Queue length: 1618
Step 4: Queue length: 1617
Step 5: Queue length: 1616
Step 6: Queue length: 1615
Step 7: Queue length: 1614
Step 8: Queue length: 1613
Step 9: Queue length: 1612
Step 10: Queue length: 1611
...
Step 9363: Queue length: 10
```

```
Step 9364: Queue length: 9
Step 9365: Queue length: 8
Step 9366: Queue length: 7
Step 9367: Queue length: 6
Step 9368: Queue length: 5
Step 9369: Queue length: 4
Step 9370: Queue length: 3
Step 9371: Queue length: 2
Step 9372: Queue length: 1

Sudoku after AC-3:
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9

Solved using AC-3!
Final Solution:
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9
```

Scenario 2: Can't be solved NOTE: The top 10 and bottom 10 steps are shown due to the sheer quantity of steps needed. A better example will be provided during live demo of code.

```
Original Sudoku:
. . . . . 6 . . .
2 . . . 3 . . . 1
. . 9 . . . 8 . .
. . 5 . . . 3 . .
. . 4 9 . 2 5 . .
. . 3 . . . 7 . .
. . 7 . . . 4 . .
8 . . . 5 . . . 9
. . . 7 . . . . .

Step 1: Queue length: 1620
Step 2: Queue length: 1619
Step 3: Queue length: 1618
Step 4: Queue length: 1617
Step 5: Queue length: 1635
Step 6: Queue length: 1653
Step 7: Queue length: 1671
Step 8: Queue length: 1670
```

```
Step 9: Queue length: 1669
Step 10: Queue length: 1668
...
Step 6798: Queue length: 10
Step 6799: Queue length: 9
Step 6800: Queue length: 8
Step 6801: Queue length: 7
Step 6802: Queue length: 6
Step 6803: Queue length: 5
Step 6804: Queue length: 4
Step 6805: Queue length: 3
Step 6806: Queue length: 2
Step 6807: Queue length: 1

Sudoku after AC-3:
. . . . . 6 . . .
2 . . . 3 . . . 1
. . 9 . . . 8 . .
. . 5 . . . 3 . .
. . 4 9 . 2 5 . .
. . 3 . . . 7 . .
. . 7 . . . 4 . .
8 . . . 5 . . . 9
. . . 7 . . . . .

AC-3 did not completely solve the puzzle. Applying backtracking...
No solution exists.
Final Solution:
. . . . . 6 . . .
2 . . . 3 . . . 1
. . 9 . . . 8 . .
. . 5 . . . 3 . .
. . 4 9 . 2 5 . .
. . 3 . . . 7 . .
. . 7 . . . 4 . .
8 . . . 5 . . . 9
. . . 7 . . . . .
```

Scenario 3: Can be solved by backtracking NOTE: The top 10 and bottom 10 steps are shown due to the sheer quantity of steps needed. A better example will be provided during live demo of code.

```
Original Sudoku:
1 . 5 . 7 . . . .
. . 4 . . 3 9 . .
. . . . . . . 6 8
. 9 . . . 7 . 2 .
. . . . . 2 8 . 7
. . 3 . 4 . . . .
. . 8 . . . . 3 .
9 4 . 5 . . . . .
. . . . . 4 2 . .

Step 1: Queue length: 1620
Step 2: Queue length: 1619
Step 3: Queue length: 1618
```

```
Step 4: Queue length: 1617
Step 5: Queue length: 1616
Step 6: Queue length: 1615
Step 7: Queue length: 1614
Step 8: Queue length: 1613
Step 9: Queue length: 1612
Step 10: Queue length: 1611
...
Step 7482: Queue length: 10
Step 7483: Queue length: 9
Step 7484: Queue length: 8
Step 7485: Queue length: 7
Step 7486: Queue length: 6
Step 7487: Queue length: 5
Step 7488: Queue length: 4
Step 7489: Queue length: 3
Step 7490: Queue length: 2
Step 7491: Queue length: 1

Sudoku after AC-3:
1 . 5 . 7 . 3 4 2
. . 4 . . 3 9 . .
. . . . . . . 6 8
. 9 . . . 7 . 2 .
. . . . . 2 8 . 7
. . 3 . 4 . . . .
. . 8 . . . . 3 .
9 4 . 5 . . . . .
. . . . . 4 2 . .

AC-3 did not completely solve the puzzle. Applying backtracking...
Final Solution:
1 8 5 9 7 6 3 4 2
7 6 4 2 8 3 9 5 1
3 2 9 4 1 5 7 6 8
8 9 1 6 5 7 4 2 3
4 5 6 3 9 2 8 1 7
2 7 3 8 4 1 6 9 5
6 1 8 7 2 9 5 3 4
9 4 2 5 3 8 1 7 6
5 3 7 1 6 4 2 8 9
```