Sudoku Vision

Foundations of AI: Multiagent Systems Project

Team Members

Jaidev Sanjay Khalane (22110103)

Vannsh Jani (22110279)

Aim

To develop an application that solves Sudoku puzzles from images using the most optimal method for different difficulty levels (by adjusting the proportion of the problem that will be solved by Constraint satisfaction and Backtracking).

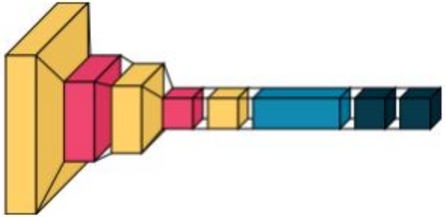
Achievements

- Developed Application that solves Sudoku directly from Images
- Explored the solving of Sudoku using:
  - Backtracking
  - Optimised Backtracking with MRV Heuristic
  - Forward Checking
  - AC3
- Developed applications for each of the algorithm for good visualisation
- Performed execution time and memory consumption analysis to find the most efficient algorithm (as a mixture of Backtracking and CSP solving based methods).

Methodology

Sudoku Image to Sudoku Array (Digit Recognition)

Dataset [1]



Model Architecture

Model: "sequential"

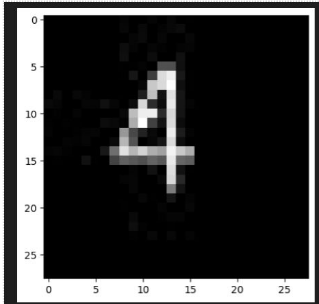
Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_2 (Conv2D)	(None, 3, 3, 64)	36,928
flatten (Flatten)	(None, 576)	0
dense (Dense)	(None, 64)	36,928
dense_1 (Dense)	(None, 10)	650

Total params: 93,322 (364.54 KB)

Trainable params: 93,322 (364.54 KB)

Non-trainable params: 0 (0.00 B)

Results




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step
[[ -0.89611906   1.3804021    6.136644     9.162972    37.61111
   -1.0218433    7.430334   -14.723456    -1.213322     6.000896  ]]
Prediction: 4
```

```
20/20 - 0s - 10ms/step - accuracy: 0.9873 - loss: 0.1744

Test accuracy: 0.9873015880584717

Test loss: 0.1743830144405365
```

Sudoku Image to Sudoku Array (Board Extraction)

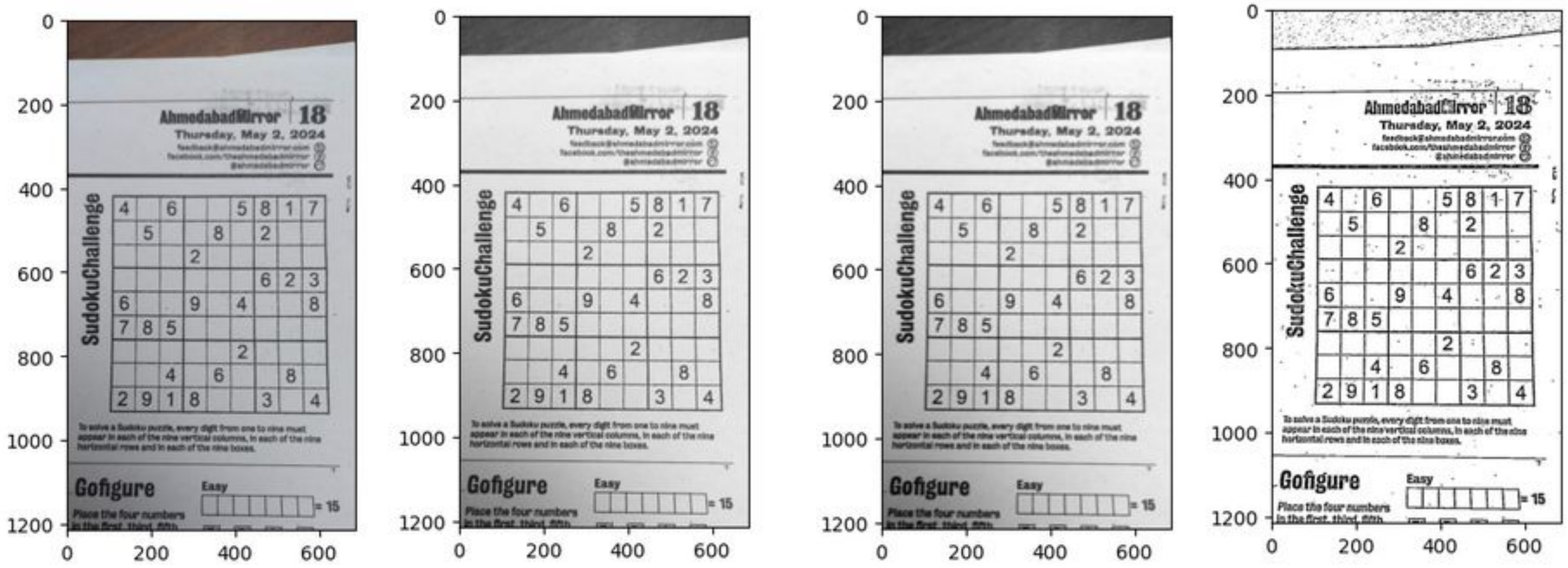
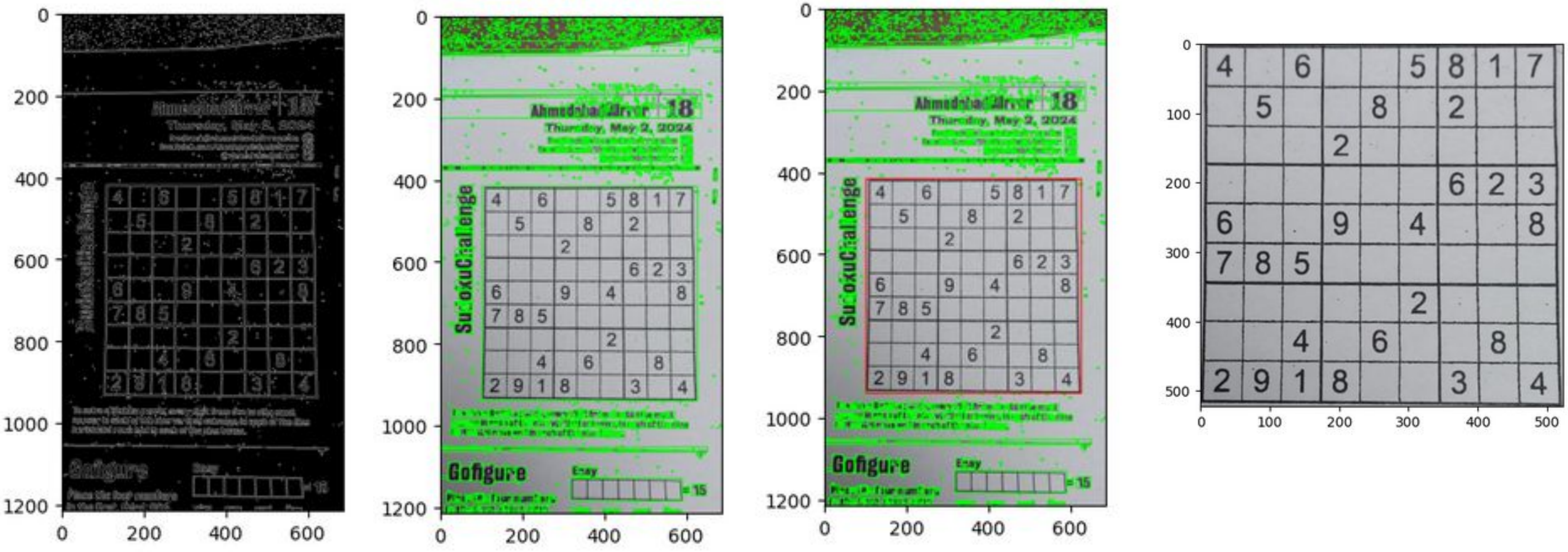


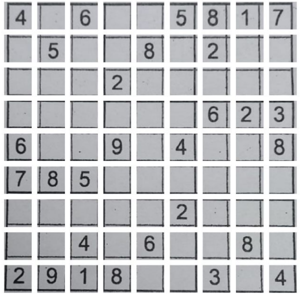
Image Capture - Grayscale Conversion - Denoising - Adaptive Thresholding

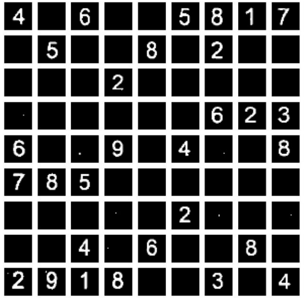
Sudoku Image to Sudoku Array (Board Extraction)



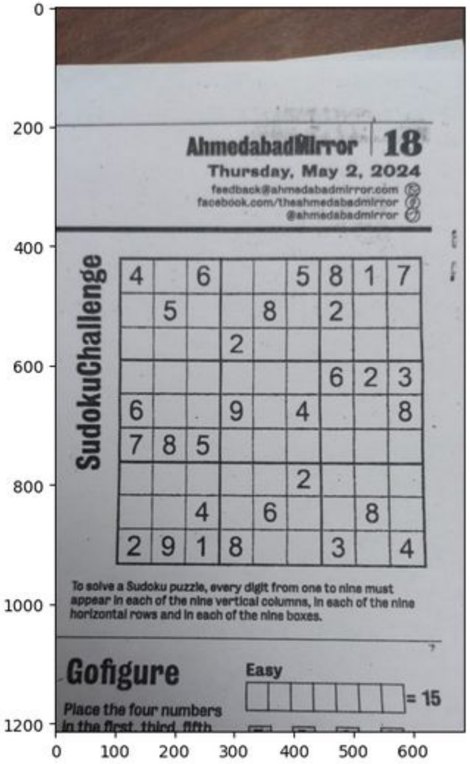
Edge Detection - Contour Analysis - Maximum Contour Extraction - Cropping

Sudoku Image to Sudoku Array (Board Extraction)





Cropping - Digit Classification



RESULTS

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

Dataset (Sudokus)

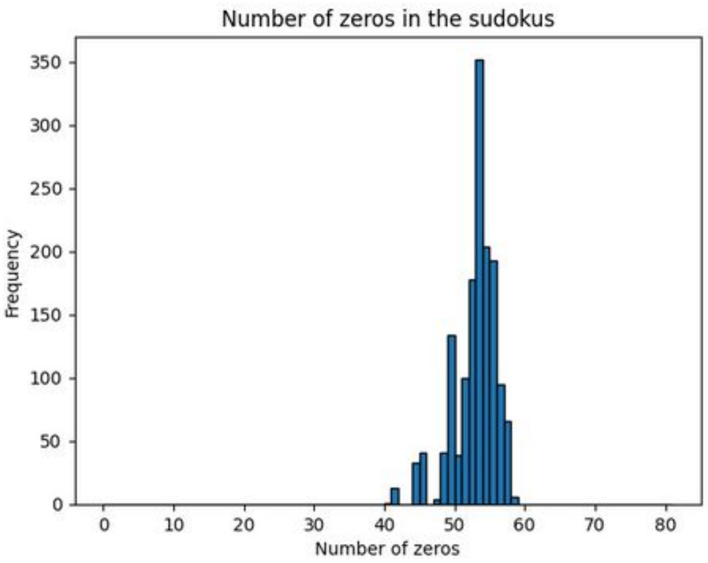
A total of 1500 sudokus were taken for analysis from [2]. They were classified into 3 levels of difficulties based on the number of zeros (empty locations):

- Hard: <50 zeros
- Medium: >50 and <55 zeros
- Easy: >55 zeros

Easy: 360 Puzzles

Medium: 873 Puzzles

Hard: 267 Puzzles



Backtracking for Solving Sudoku

```
def backtracking(sudoku_grid):
    def solve_sudoku(grid):
        for row in range(9):
            for col in range(9):
                if grid[row][col] == 0:
                    for num in range(1, 10):
                        if is_valid(grid, row, col, num):
                            grid[row][col] = num
                            if solve_sudoku(grid):
                                return True
                            grid[row][col] = 0
                    return False
        return True

    solve_sudoku(sudoku_grid)

    return sudoku_grid
```

Optimised Backtracking for Solving Sudokus

```python
# mrv = minimum remaining values herustic
def find_mrv(board):
    min_count = 10
    mrv_position = (-1, -1)
    for row in range(9):
        for col in range(9):
            if board[row][col] == 0:
                count = sum(is_valid(board, row, col, num) for num in range(1, 10))
                if count < min_count:
                    min_count = count
                    mrv_position = (row, col)
    return mrv_position
```

```python
def solve_sudoku(board,all_boards):
    all_boards.append([row[:] for row in board])
    empty = find_mrv(board)
    if empty == (-1, -1):
        return True, all_boards

    row, col = empty
    for num in range(1, 10):
        if is_valid(board, row, col, num):
            board[row][col] = num
            if solve_sudoku(board,all_boards):
                return True, all_boards
            board[row][col] = 0

    return False, all_boards
```

Forward Checking (Constraint Development)

```python
def get_constraints(grid):
    constraints = {}
    for row in range(9):
        for col in range(9):
            if grid[row][col] == 0:
                allowed_values = set(range(1, 10))
                for x in range(9):
                    if grid[row][x] in allowed_values:
                        allowed_values.remove(grid[row][x])
                    if grid[x][col] in allowed_values:
                        allowed_values.remove(grid[x][col])
                start_row, start_col = 3 * (row // 3), 3 * (col // 3)
                for i in range(3):
                    for j in range(3):
                        if grid[start_row + i][start_col + j] in allowed_values:
                            allowed_values.remove(grid[start_row + i][start_col + j])
                constraints[(row, col)] = allowed_values
    return constraints
```

```
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
```

```
Constraints for each unassigned variable:
Cell (0, 2): {1, 2, 4}
Cell (0, 3): {2, 6}
Cell (0, 5): {2, 4, 6, 8}
Cell (0, 6): {1, 4, 8, 9}
Cell (0, 7): {1, 2, 4, 9}
Cell (0, 8): {2, 4, 8}
Cell (1, 1): {2, 4, 7}
Cell (1, 2): {2, 4, 7}
Cell (1, 6): {3, 4, 7, 8}
Cell (1, 7): {2, 3, 4}
Cell (1, 8): {2, 4, 7, 8}
Cell (2, 0): {1, 2}
Cell (2, 3): {2, 3}
Cell (2, 4): {3, 4}
Cell (2, 5): {2, 4}
Cell (2, 6): {1, 3, 4, 5, 7}
Cell (2, 8): {2, 4, 7}
Cell (3, 1): {1, 2, 5}
Cell (3, 2): {1, 2, 5, 9}
Cell (3, 3): {5, 7, 9}
Cell (3, 5): {1, 4, 7}
Cell (3, 6): {4, 5, 7, 9}
Cell (3, 7): {2, 4, 5, 9}
Cell (4, 1): {2, 5}
Cell (4, 2): {2, 5, 6, 9}
Cell (4, 4): {5}
Cell (4, 6): {5, 7, 9}
Cell (4, 7): {2, 5, 9}
Cell (5, 1): {1, 5}
Cell (5, 2): {1, 3, 5, 9}
Cell (5, 3): {5, 9}
Cell (5, 5): {1, 4}
Cell (5, 6): {4, 5, 8, 9}
Cell (5, 7): {4, 5, 9}
Cell (6, 0): {1, 3, 9}
Cell (6, 2): {1, 3, 4, 5, 7, 9}
Cell (6, 3): {3, 5, 7}
Cell (6, 4): {3, 5}
Cell (6, 5): {7}
Cell (6, 8): {4}
Cell (7, 0): {2, 3}
```

Forward Checking

```python
def forward_checking(grid, constraints, iterations=81):
    def assign_value(row, col, num):
        grid[row][col] = num
        for x in range(9):
            if (row, x) in constraints:
                constraints[(row, x)].discard(num)
            if (x, col) in constraints:
                constraints[(x, col)].discard(num)
        start_row, start_col = 3 * (row // 3), 3 * (col // 3)
        for i in range(3):
            for j in range(3):
                if (start_row + i, start_col + j) in constraints:
                    constraints[(start_row + i, start_col + j)].discard(num)

    for _ in range(iterations):
        if not constraints:
            break
        min_cell = min(constraints, key=lambda k: len(constraints[k]))
        if len(constraints[min_cell]) == 0:
            break
        num = constraints[min_cell].pop()
        print(f"Assigning {num} to cell {min_cell} with {len(constraints[min_cell])} remaining values")
        assign_value(min_cell[0], min_cell[1], num)
        del constraints[min_cell]

    return grid
```

```
forward_checking(sudoku_grid, constraints, 25)
```

```
Assigning 5 to cell (4, 4) with 0 remaining values
Assigning 2 to cell (4, 1) with 0 remaining values
Assigning 9 to cell (4, 7) with 0 remaining values
Assigning 6 to cell (4, 2) with 0 remaining values
Assigning 7 to cell (4, 6) with 0 remaining values
Assigning 9 to cell (5, 3) with 0 remaining values
Assigning 7 to cell (3, 3) with 0 remaining values
Assigning 3 to cell (6, 4) with 0 remaining values
Assigning 4 to cell (2, 4) with 0 remaining values
Assigning 2 to cell (2, 5) with 0 remaining values
Assigning 6 to cell (0, 3) with 0 remaining values
Assigning 8 to cell (0, 5) with 0 remaining values
Assigning 1 to cell (2, 0) with 0 remaining values
Assigning 3 to cell (2, 3) with 0 remaining values
Assigning 5 to cell (2, 6) with 0 remaining values
Assigning 7 to cell (2, 8) with 0 remaining values
Assigning 4 to cell (3, 6) with 0 remaining values
Assigning 1 to cell (3, 5) with 0 remaining values
Assigning 5 to cell (3, 1) with 0 remaining values
Assigning 9 to cell (3, 2) with 0 remaining values
Assigning 2 to cell (3, 7) with 0 remaining values
Assigning 1 to cell (5, 1) with 0 remaining values
Assigning 3 to cell (5, 2) with 0 remaining values
Assigning 4 to cell (5, 5) with 0 remaining values
Assigning 8 to cell (5, 6) with 0 remaining values

Sudoku after forward checking:
5 3 . 6 7 8 . . .
6 . . 1 9 5 . . .
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 . 6
. 6 . . 3 . 2 8 .
. . . 4 1 9 . . 5
. . . . 8 . . 7 9
```

AC3

```python
def arc_consistency_check(domains, x1, y1, x2, y2):

    domX1 = domains[x1][y1]
    domX2 = domains[x2][y2]
    if len(domX1) == 0:
        return True, []
    revised = False
    val = []
    for x in domX1:
        isconsistent = False
        for y in domX2:
            if x != y:
                isconsistent = True
                break
        if not isconsistent:
            val.append(x)
            revised = True
    for x in val:
        domX1.discard(x)
    return revised, val
```

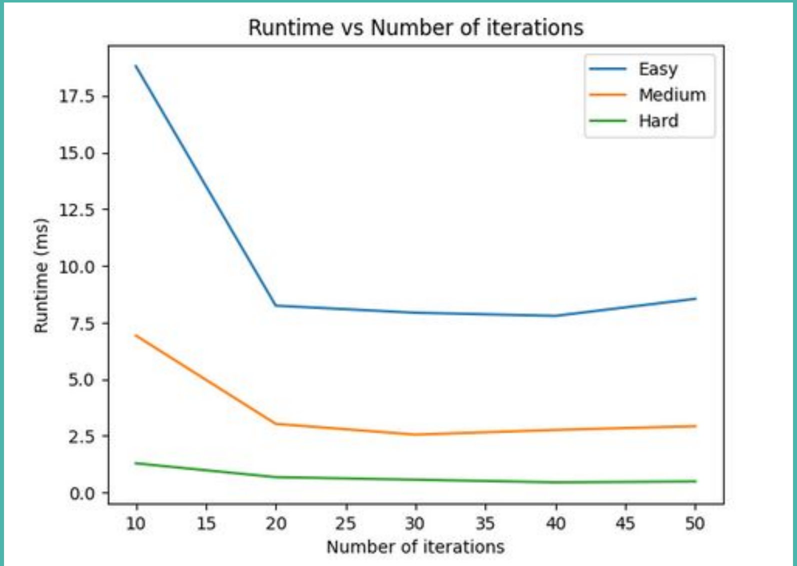
```python
def AC3(domains, arcs,constraints):
    is_consistent = True
    updated_domains_every_step = []
    all_arcs_ac3 = []
    while arcs:
        (Xi, Xj) = arcs.pop(0)
        revised, val = arc_consistency_check(domains, Xi[0],Xi[1], Xj[0],Xj[1])
        updated_domains_every_step.append(deepcopy(domains))
        all_arcs_ac3.append(((Xi[0],Xi[1]),(Xj[0],Xj[1])))
        if revised:
            if len(domains[Xi[0]][Xi[1]]) == 0:
                is_consistent = False
                return is_consistent, updated_domains_every_step, all_arcs_ac3, Xi,
            for Xk in constraints:
                if Xi in constraints[Xk]:
                    arcs.append(((int(Xk[1]),int(Xk[4])), Xi))

    return is_consistent, updated_domains_every_step, all_arcs_ac3,None,None
```

Algorithmic Analysis (Optimal Combination)

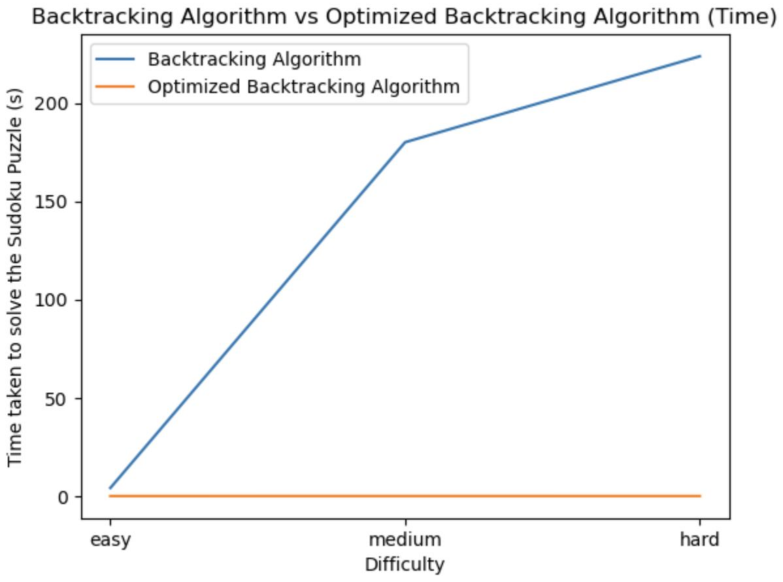
- In order to determine the optimal combination of the algorithms (Backtracking and Forward Checking), a detailed algorithmic analysis was carried out:
    - For different difficulty levels of Sudokus
    - For different number of iterations (of running Forward Checking Algorithm)
- The optimality of the algorithm was considered in terms of the “Mean Execution Time” and “Mean Memory Consumption” for 3 executions.
- Results are presented in the Results Section

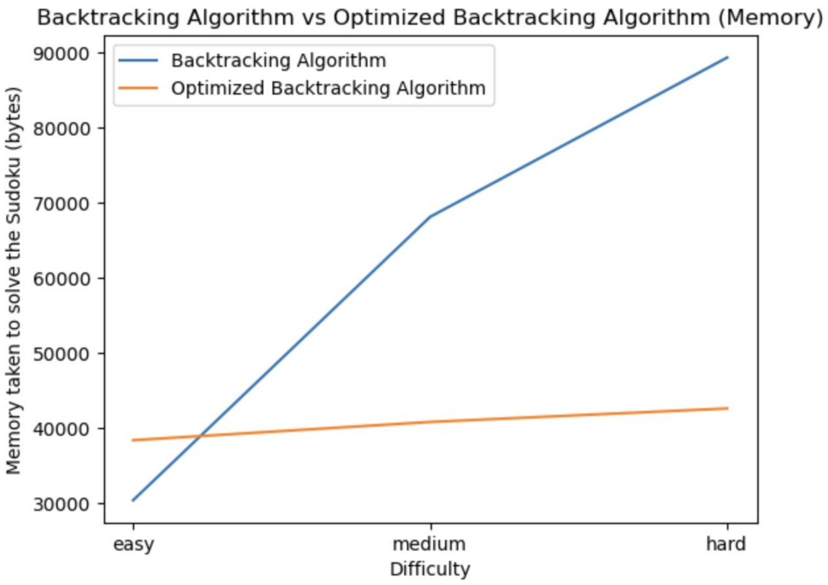
Results



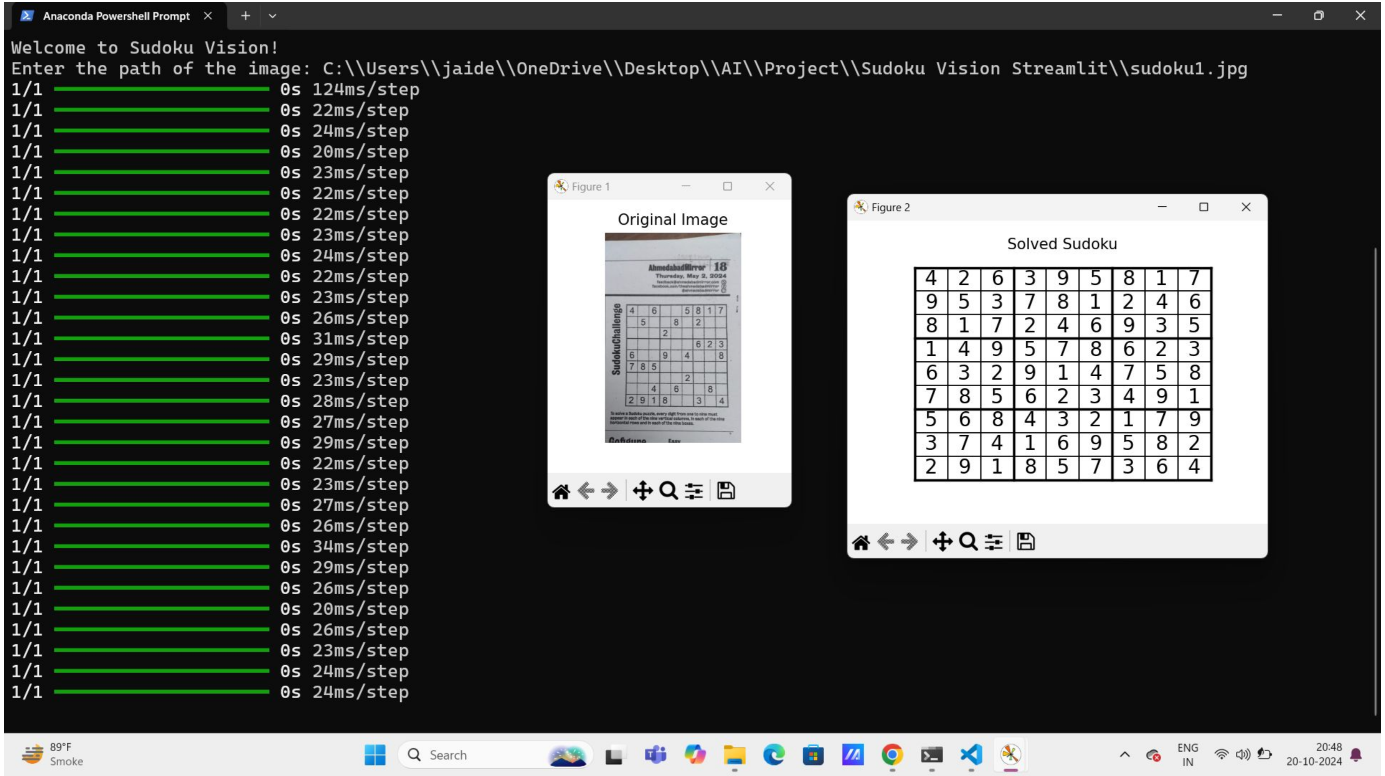
Runtime vs Number of iterations
Easy
Medium
Hard
Runtime (ms)
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
10
15
20
25
30
35
40
45
50
Number of iterations

Backtracking vs Optimized Backtracking (with MRV)





CLI Applications

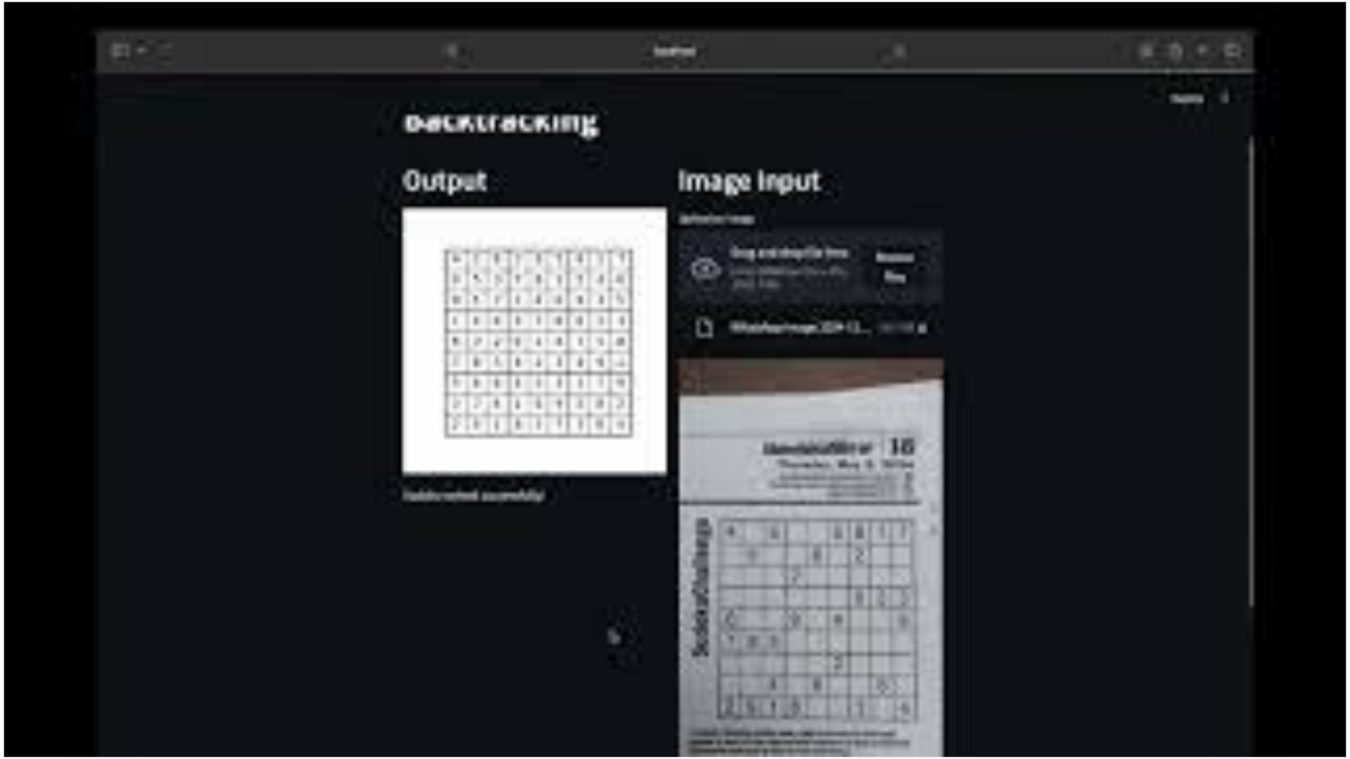


Backtracking Application

Backtracking Animation

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

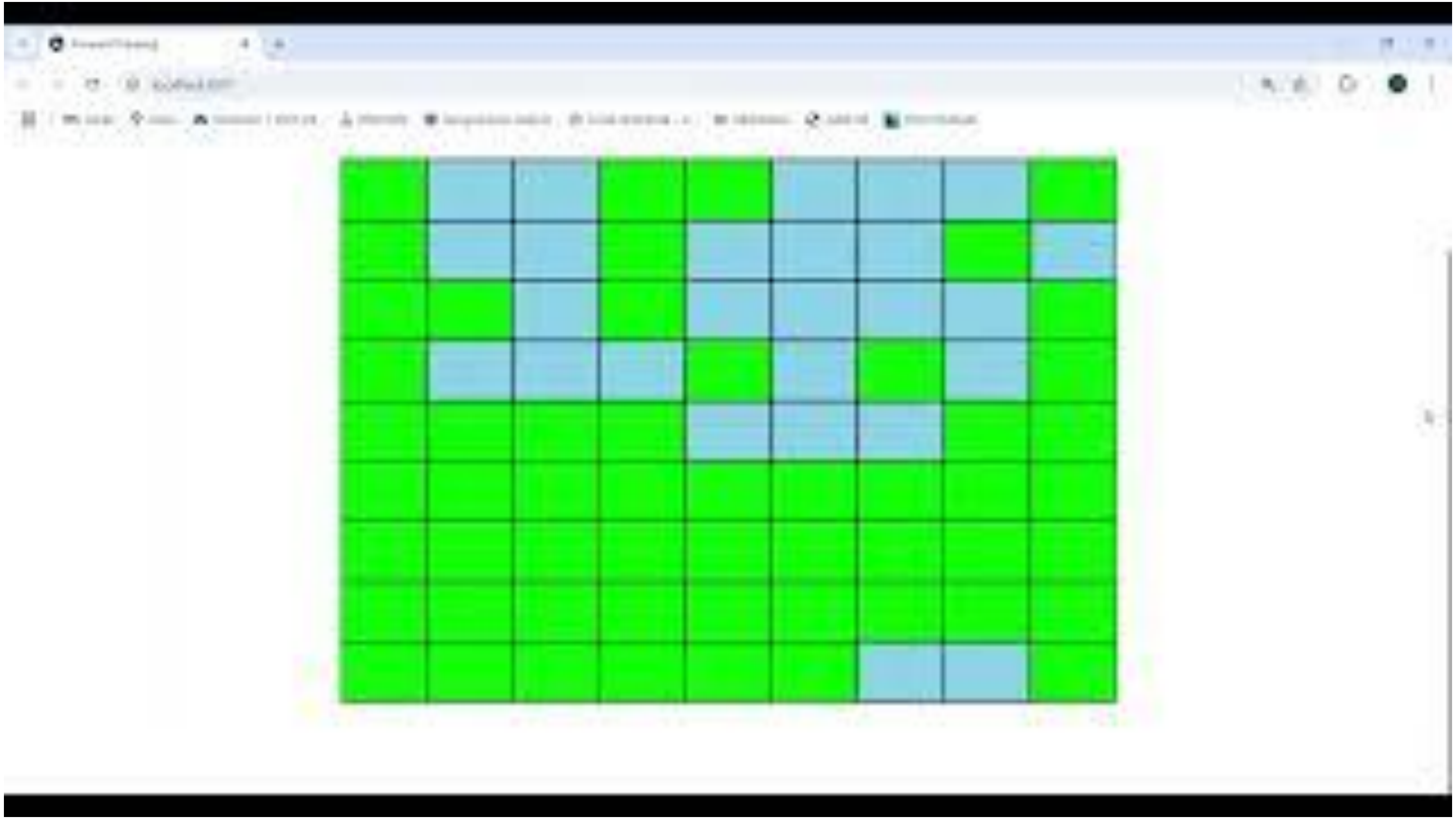
Optimised Backtracking Application



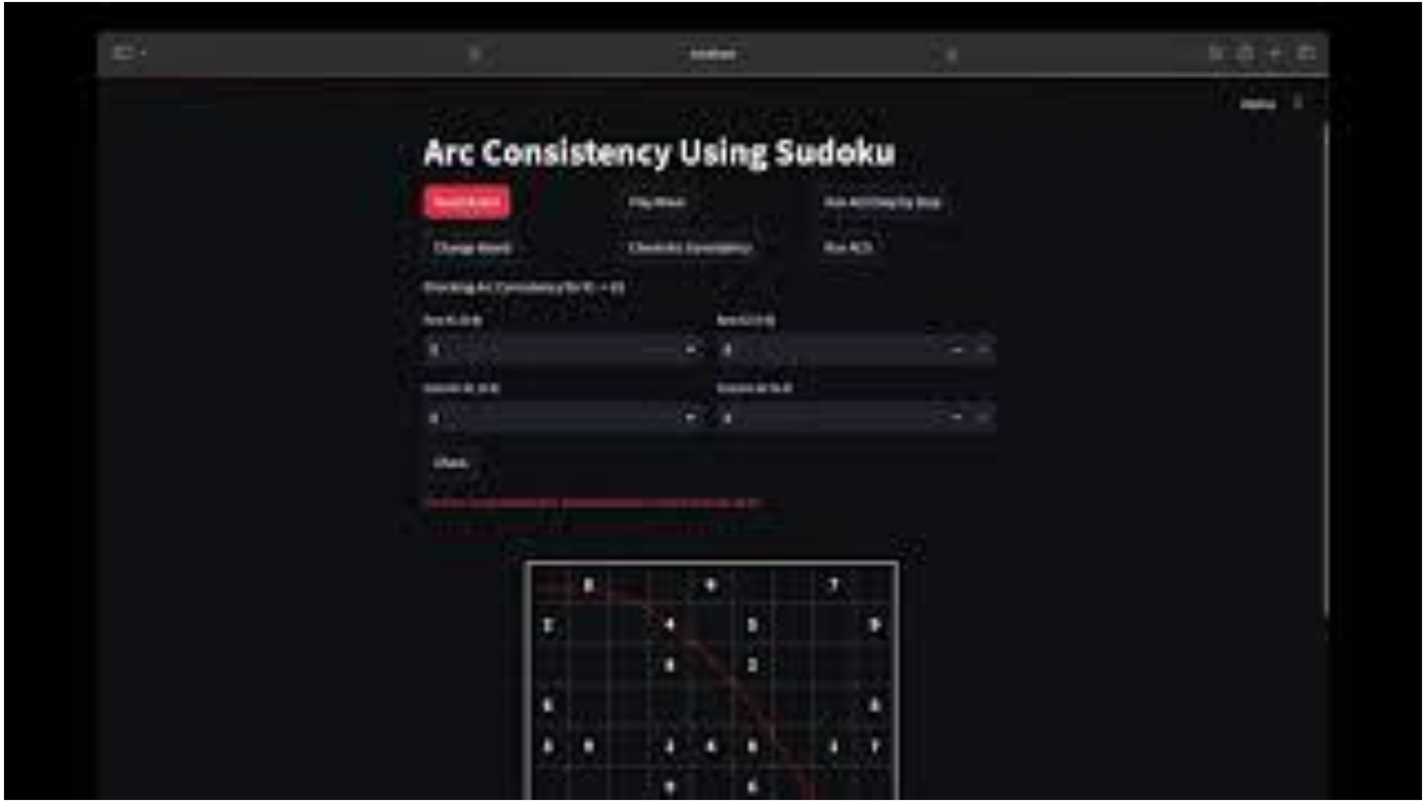
Optimized Backtracking Animation

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

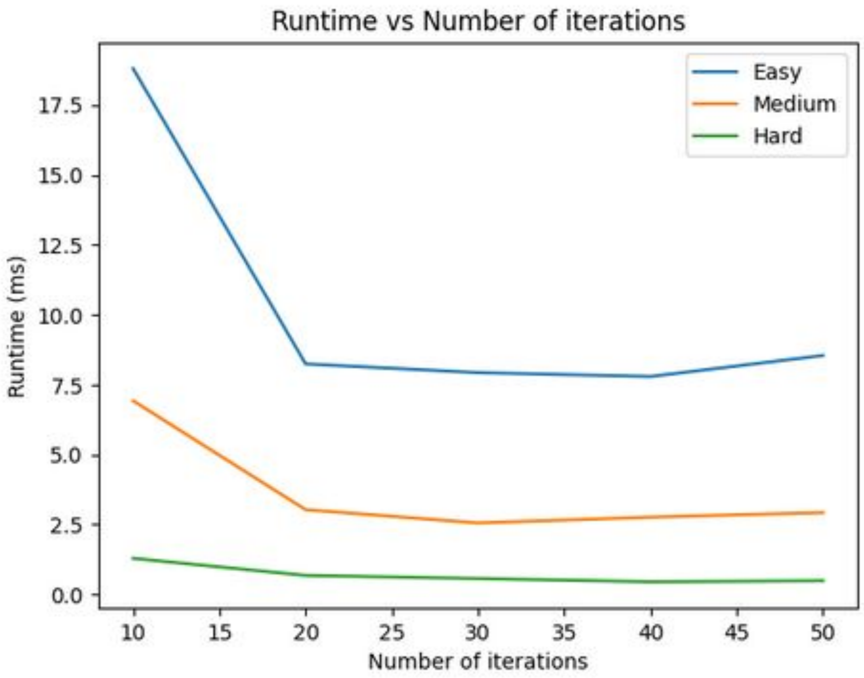
Forward Checking Application

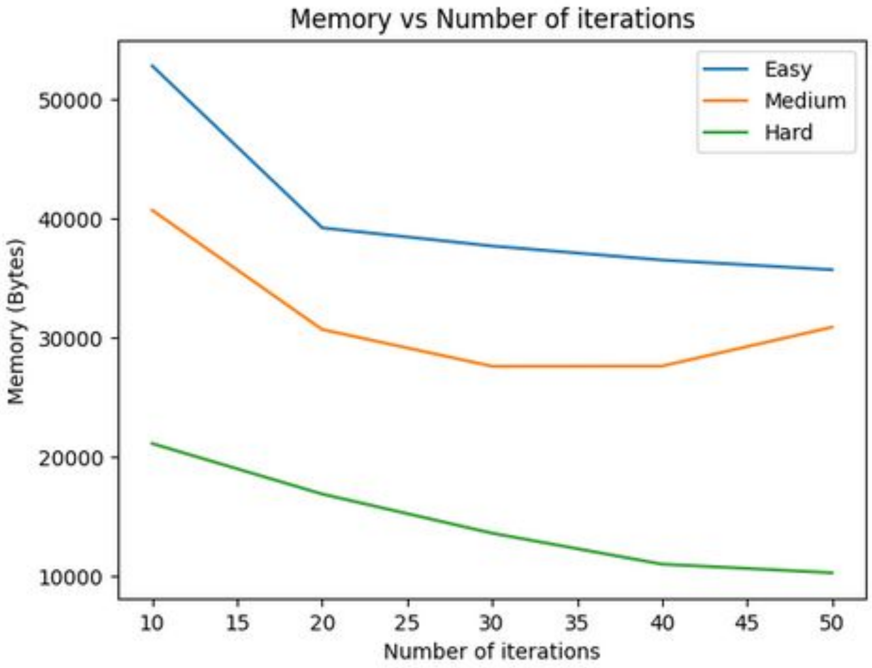


AC3 Application



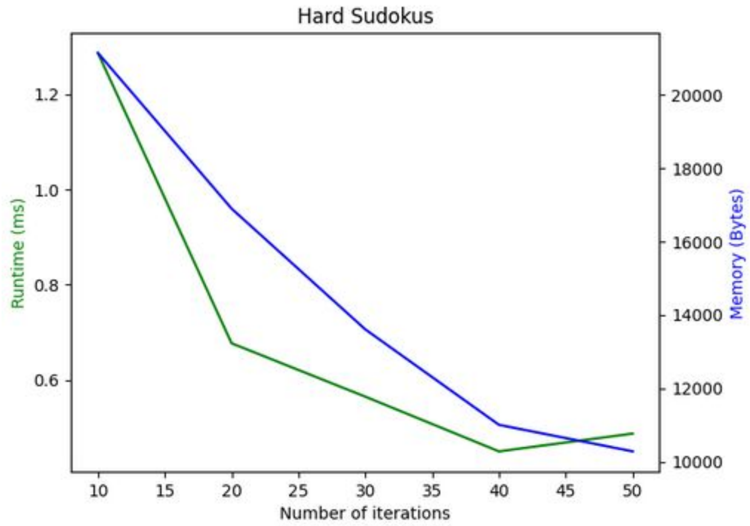
Algorithm Analysis



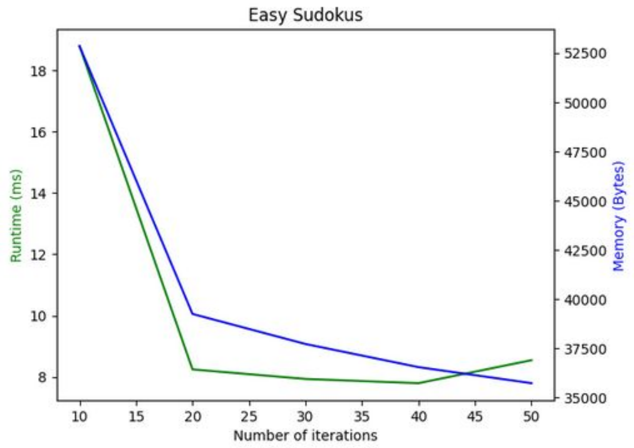


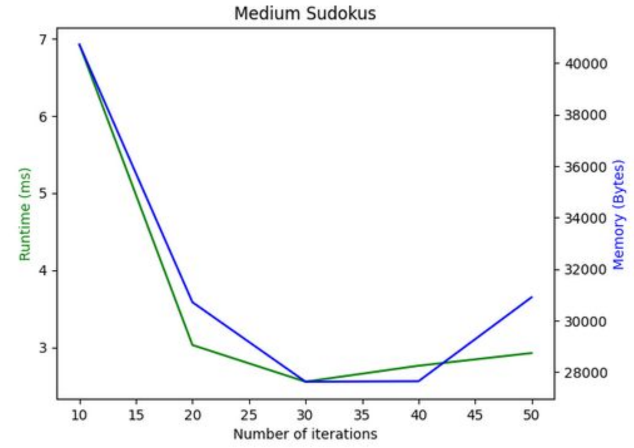
Number of Iterations of Forward Checking

Algorithm Analysis

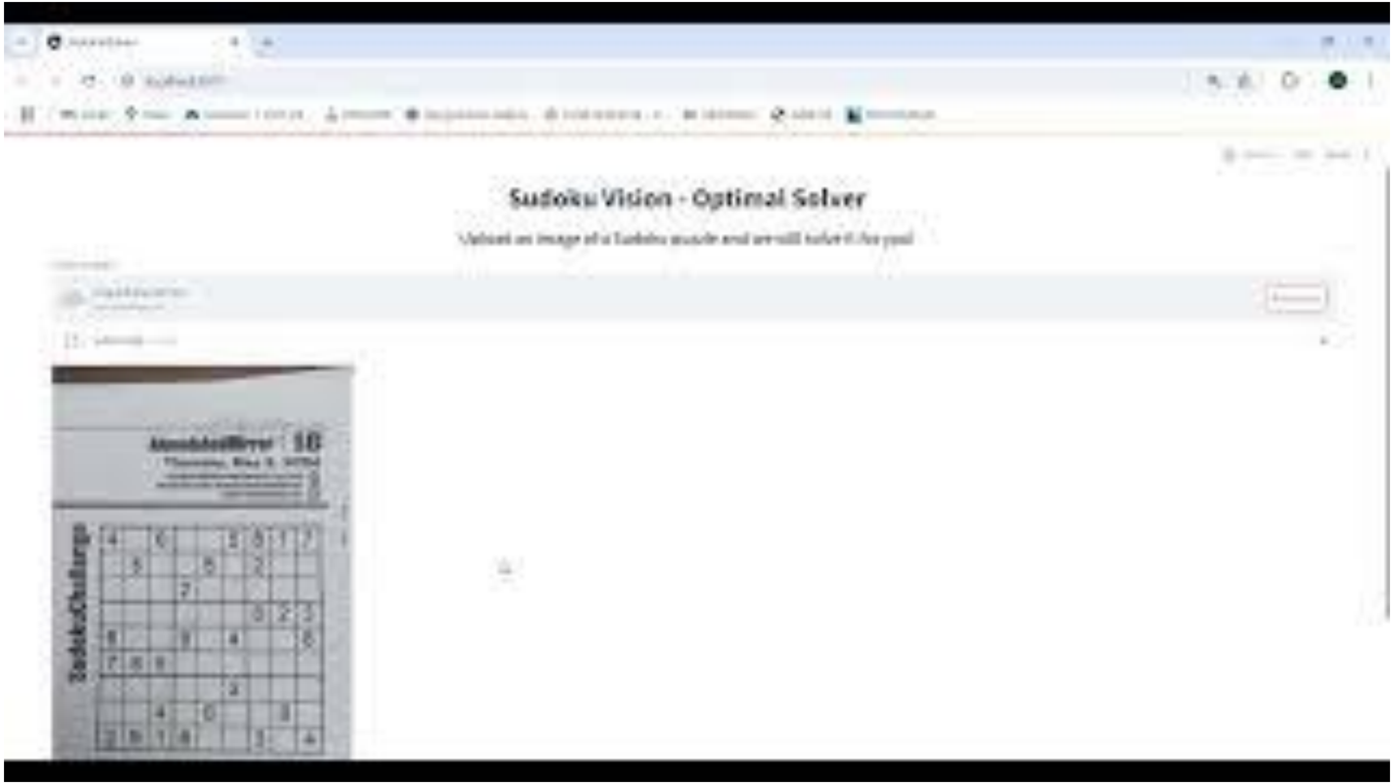


Difficulty	No. of Iterations of Forward Checking
Easy	~45
Medium	~30
Hard	~45





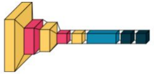
Optimal Sudoku Vision Solver



Learnings and Conclusion

Sudoku Image to Sudoku Array (Digit Recognition)

Dataset [1]



Model Architecture

Model: "sequential"

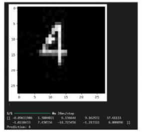
Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_2 (Conv2D)	(None, 3, 3, 64)	36,928
flatten (Flatten)	(None, 576)	0
dense (Dense)	(None, 64)	36,928
dense_1 (Dense)	(None, 10)	650

Total params: 93,322 (364.54 KB)

Trainable params: 93,322 (364.54 KB)

Non-trainable params: 0 (0.00 B)

Results

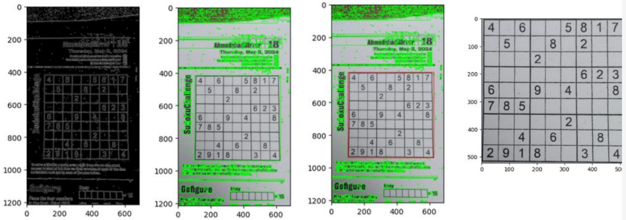


```
20/20 - 0s - 10ms/step - accuracy: 0.9873 - loss: 0.1744

Test accuracy: 0.9873015880584717

Test loss: 0.1743830144405365
```

Sudoku Image to Sudoku Array (Board Extraction)



Edge Detection - Contour Analysis - Maximum Contour Extraction - Cropping

```python
def backtracking(sudoku_grid):
    def solve_sudoku(grid):
        for row in range(9):
            for col in range(9):
                if grid[row][col] == 0:
                    for num in range(1, 10):
                        if is_valid(grid, row, col, num):
                            grid[row][col] = num
                            if solve_sudoku(grid):
                                return True
                            grid[row][col] = 0
                    return False
        return True

    solve_sudoku(sudoku_grid)

    return sudoku_grid
```

```python
# mrv = minimum remaining values herustic
def find_mrv(board):
    min_count = 10
    mrv_position = (-1, -1)
    for row in range(9):
        for col in range(9):
            if board[row][col] == 0:
                count = sum(is_valid(board, row, col, num) for num in range(1, 10)
                if count < min_count:
                    min_count = count
                    mrv_position = (row, col)
    return mrv_position
```

```python
def solve_sudoku(board,all_boards):
    all_boards.append([row[:] for row in board])
    empty = find_mrv(board)
    if empty == (-1, -1):
        return True, all_boards

    row, col = empty
    for num in range(1, 10):
        if is_valid(board, row, col, num):
            board[row][col] = num
            if solve_sudoku(board,all_boards):
                return True, all_boards
            board[row][col] = 0

    return False, all_boards
```

Learnings and Conclusion

```python
def get_constraints(grid):
    constraints = {}
    for row in range(9):
        for col in range(9):
            if grid[row][col] == 0:
                allowed_values = set(range(1, 10))
    return
```

```python
def forward_checking(grid, constraints, iterations=81):
    def assign_value(row, col, num):
        grid[row][col] = num
        for x in range(9):
            if (row, x) in constraints:
                constraints[(row, x)].discard(num)
            if (x, col) in constraints:
                constraints[(x, col)].discard(num)
        start_row, start_col = 3 * (row // 3), 3 * (col // 3)
        for i in range(3):
            for j in range(3):
```

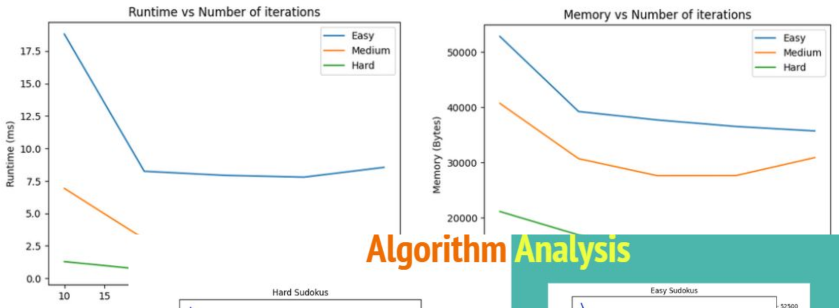
```python
def AC3(domains, arcs,constraints):
    is_consistent = True
    updated_domains_every_step = [
    all_arcs_ac3 = []
    while arcs:
        (Xi, Xj) = arcs.pop(0)
        revised, val = arc_consiste
        updated_domains_every_step.
        all_arcs_ac3.append(((Xi[0]
        if revised:
            if len(domains[Xi[0]][X
                is_consistent = Fa
                return is_consister
            for Xk in constraints:
                if Xi in constraint
                    arcs.append(((

    return is_consistent, updated_
```

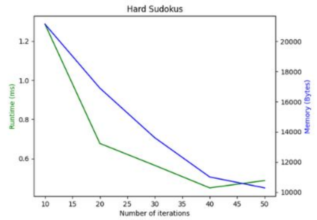
```python
def arc_consistency_check(domains, x1, y1, x2, y2):

    domX1 = domains[x1][y1]
    domX2 = domains[x2][y2]
    if len(domX1) == 0:
        return True, []
    revised = False
    val = []
    for x in domX1:
        isconsistent = False
        for y in domX2:
            if x != y:
                isconsistent = True
                break
        if not isconsistent:
            val.append(x)
            revised = True
    for x in val:
        domX1.discard(x)
    return revised, val
```

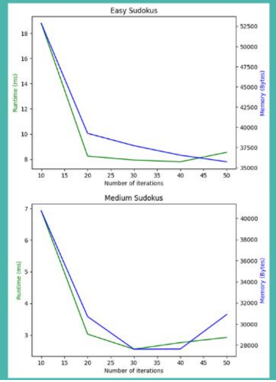
Algorithm Analysis



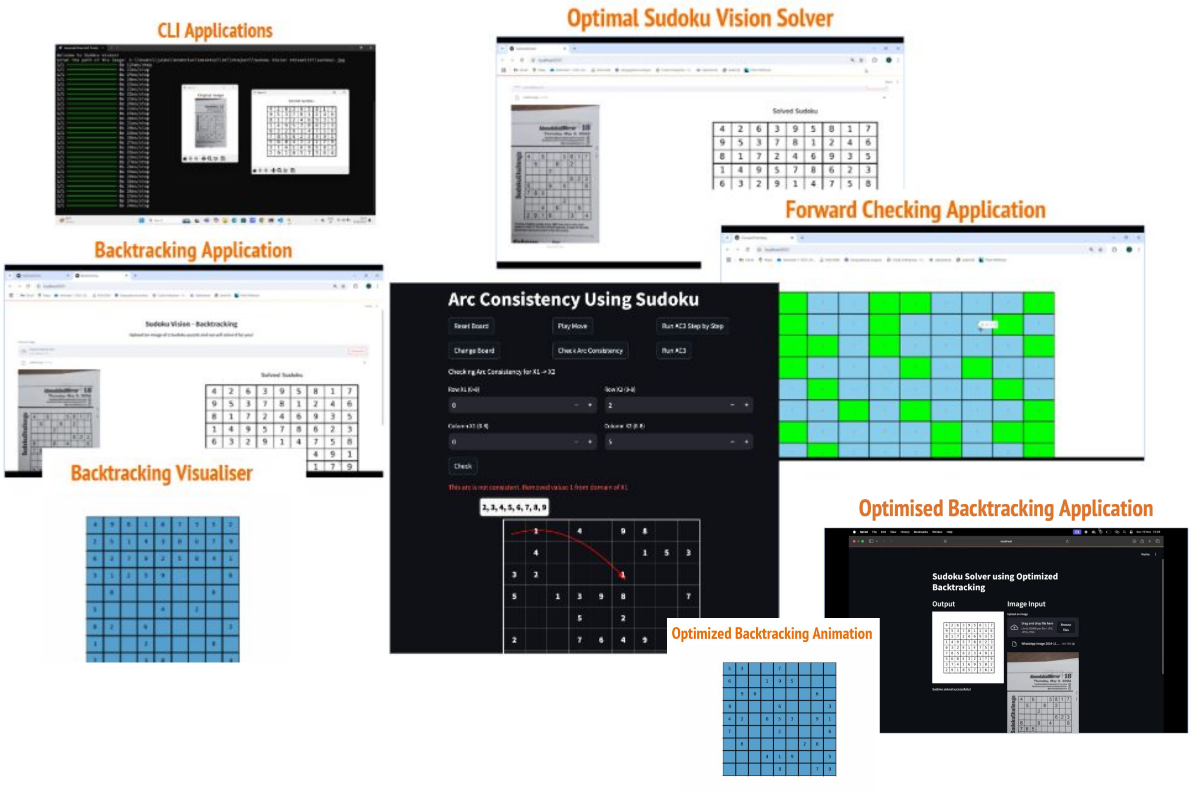
Algorithm Analysis



Difficulty	No. of Iterations of Forward Checking
Easy	~45
Medium	~30
Hard	~45



Learnings and Conclusion



References and Dataset Credits

[1] https://www.kaggle.com/datasets/kshitijdhama/printed-digits-dataset

[2] https://github.com/grantm/sudoku-exchange-puzzle-bank