Sudoku Solver Project Report

Part A:

In order to run the program, please have the input file in the same directory as the sudoku solver Python script. Then, import sys and copy in python, and use the actual input_file and output_file names in line 167 and 168. The program can be then runned by using
‘python3 sudoku.py’

Part B:

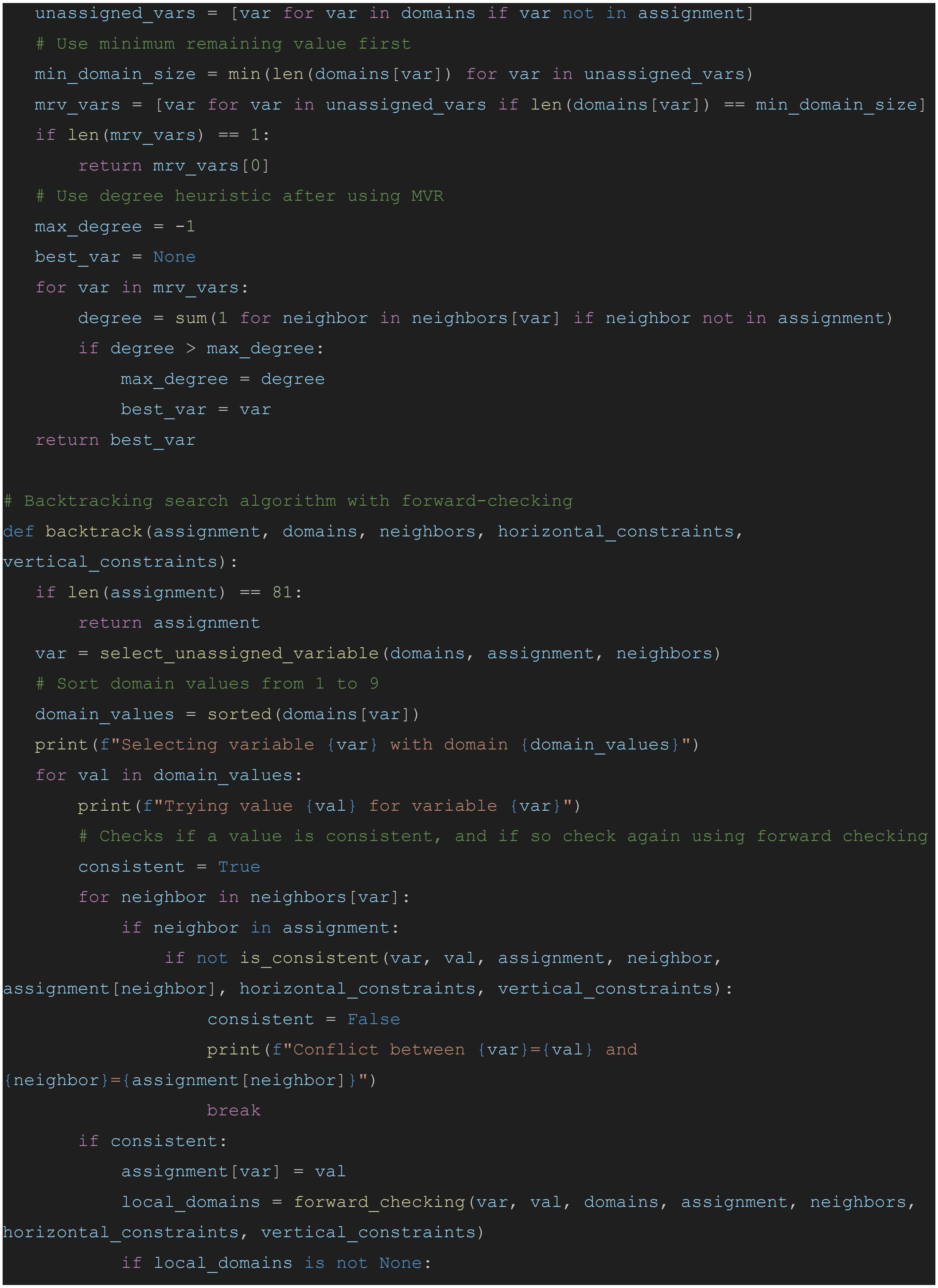
I formulate the sudoku as a constraint satisfaction problem by:

1. Variables:
    a. Each cell is represented as a variable, and they are defined as i, j where i is the row and j is the column index. The 9*9 sudoku grid contains a total of 81 variables.
2. Domains:
    a. The domain are the set of possible values that can be assigned to each cell. If a cell is pre defined in the input file, the there are only 1 possible value that can be assigned to that cell. Otherwise, it can be any value between 1 to 9, as long is it does not violates any constraints.
3. Constraints:
    a. Row constraints: Each value in 1 to 9 should be found in each row with no duplicates.
    b. Column constraints: Each value in 1 to 9 should be found in each column with no duplicates.
    c. Block constraints: Each value in 1 to 9 should be found in each independent 3*3 blocks with no duplicates.
    d. White dot constraints: All pairs of adjacent values connected by white dots, have one value that is 1 above the other value.
    e. Black dot constraints: All pairs of adjacent values connected by black dots, have one value that is 2 times the other value.
4. MRV:
    a. Select the unassignal variable with smallest domain size
5. Degree Heursitic:
    a. If MRV of two variables has a tie, select the variable with highest number of unassigned neighbor
6. INFERENCE: Forward Checking (Extra credit):
    a. After assigning a value to a variable, immediately update the domains of its unassigned neighbors by removing any values that violate the constraints and prunes it if violations are found.
7. Backtracking Algorithm:
    a. Recursively select an unassigned variable using MRV and degree heuristic, assign a possible value to it and use forward checking to propagate constraints. If consistent, then can move on to other nodes, otherwise backtrack.

Part C:
Source Code:

```
Import sys
import copy

# Reads the input file and return the arrays
def read_input_file(input_file):
   grid = []
   horiz_dots = []
   vert_dots = []

   with open(input_file, 'r') as f:
       for _ in range(9):
           line = f.readline().strip()
           grid.append([int(x) for x in line.split()])
       f.readline()

       for _ in range(9):
           line = f.readline().strip()
           horiz_dots.append([int(x) for x in line.split()])
       f.readline()

       for _ in range(8):
           line = f.readline().strip()
           vert_dots.append([int(x) for x in line.split()])

   return grid, horiz_dots, vert_dots


# Initialize variables, domains, and constraints using the grid and constraints
received from the input file
def initialize_variables(sudoku_grid, horizontal_constraints, vertical_constraints):
   variables = []
   domains = {}
   neighbors = {}

   # Initialize variables and domains
   for i in range(9):
       for j in range(9):
           if sudoku_grid[i][j] == 0:
               variables.append((i, j))
               domains[(i, j)] = set(range(1, 10))
```

```
            else:
                domains[(i, j)] = set([sudoku_grid[i][j]])

    # Initialize neighbors
    for i in range(9):
        for j in range(9):
            var = (i, j)
            cell_neighbors = set()
            for k in range(9):
                if k != j:
                    cell_neighbors.add((i, k))
                if k != i:
                    cell_neighbors.add((k, j))
            # Check block neighbors
            block_row = (i // 3) * 3
            block_col = (j // 3) * 3
            for m in range(block_row, block_row + 3):
                for n in range(block_col, block_col + 3):
                    if (m, n) != (i, j):
                        cell_neighbors.add((m, n))
            # Checks horizontal rows neighbors
            if j < 8 and horizontal_constraints[i][j] != 0:
                cell_neighbors.add((i, j + 1))
            if j > 0 and horizontal_constraints[i][j - 1] != 0:
                cell_neighbors.add((i, j - 1))
            # Checks vertical column neightbors
            if i < 8 and vertical_constraints[i][j] != 0:
                cell_neighbors.add((i + 1, j))
            if i > 0 and vertical_constraints[i - 1][j] != 0:
                cell_neighbors.add((i - 1, j))
            neighbors[var] = cell_neighbors

    return variables, domains, neighbors

# Checks if the all the constraints are satisfied
def is_consistent(var, val, assignment, var2, val2, horizontal_constraints,
vertical_constraints):
    i1, j1 = var
    i2, j2 = var2
    # Return false if they are the same
    if val == val2:
        return False
    # Return false if horizontal constraint is not satisfied
    if i1 == i2 and abs(j1 - j2) == 1:
        h_index = min(j1, j2)
        h_constraint = horizontal_constraints[i1][h_index]
        if h_constraint == 1:
            if abs(val - val2) != 1:
                return False
        elif h_constraint == 2:
            if val != 2 * val2 and val2 != 2 * val:
                return False
    # Return false if vertical constraint is not satisfied
    if j1 == j2 and abs(i1 - i2) == 1:
        v_index = min(i1, i2)
        v_constraint = vertical_constraints[v_index][j1]
        if v_constraint == 1:
            if abs(val - val2) != 1:
                return False
        elif v_constraint == 2:
            if val != 2 * val2 and val2 != 2 * val:
                return False
    return True

#  Optional Forward-checking by copying the domains using deep copy and checks the
consistency
def forward_checking(var, val, domains, assignment, neighbors, horizontal_constraints,
vertical_constraints):
    local_domains = copy.deepcopy(domains)
    for neighbor in neighbors[var]:
        if neighbor not in assignment:
            to_remove = set()
            for val2 in local_domains[neighbor]:
                if not is_consistent(var, val, assignment, neighbor, val2,
horizontal_constraints, vertical_constraints):
                    to_remove.add(val2)
            if to_remove:
                local_domains[neighbor] -= to_remove
                if not local_domains[neighbor]:
                    return None
    return local_domains

# Select an available variable using minimum remaining value and degree heuristic
def select_unassigned_variable(domains, assignment, neighbors):
```



```
    unassigned_vars = [var for var in domains if var not in assignment]
    # Use minimum remaining value first
    min_domain_size = min(len(domains[var]) for var in unassigned_vars)
    mrv_vars = [var for var in unassigned_vars if len(domains[var]) == min_domain_size]
    if len(mrv_vars) == 1:
        return mrv_vars[0]
    # Use degree heuristic after using MVR
    max_degree = -1
    best_var = None
    for var in mrv_vars:
        degree = sum(1 for neighbor in neighbors[var] if neighbor not in assignment)
        if degree > max_degree:
            max_degree = degree
            best_var = var
    return best_var

# Backtracking search algorithm with forward-checking
def backtrack(assignment, domains, neighbors, horizontal_constraints,
vertical_constraints):
    if len(assignment) == 81:
        return assignment
    var = select_unassigned_variable(domains, assignment, neighbors)
    # Sort domain values from 1 to 9
    domain_values = sorted(domains[var])
    print(f"Selecting variable {var} with domain {domain_values}")
    for val in domain_values:
        print(f"Trying value {val} for variable {var}")
        # Checks if a value is consistent, and if so check again using forward checking
        consistent = True
        for neighbor in neighbors[var]:
            if neighbor in assignment:
                if not is_consistent(var, val, assignment, neighbor,
assignment[neighbor], horizontal_constraints, vertical_constraints):
                    consistent = False
                    print(f"Conflict between {var}={val} and
{neighbor}={assignment[neighbor]}")
                    break
        if consistent:
            assignment[var] = val
            local_domains = forward_checking(var, val, domains, assignment, neighbors,
horizontal_constraints, vertical_constraints)
            if local_domains is not None:
```

```python
                result = backtrack(assignment, local_domains, neighbors,
horizontal_constraints, vertical_constraints)
                if result is not None:
                    return result
            print(f"Backtracking on variable {var}")
            del assignment[var]
    return None

# Start to solve the sudoku problem
if __name__ == '__main__':
    input_file = 'Input3.txt'
    output_file = 'Output3.txt'

    sudoku_grid, horizontal_constraints, vertical_constraints =
read_input_file(input_file)

    variables, domains, neighbors = initialize_variables(sudoku_grid,
horizontal_constraints, vertical_constraints)
    assignment = {}
    # Assign pre-filled cells
    for i in range(9):
        for j in range(9):
            if sudoku_grid[i][j] != 0:
                assignment[(i, j)] = sudoku_grid[i][j]

    result = backtrack(assignment, domains, neighbors, horizontal_constraints,
vertical_constraints)
    # If a correct solution is found that satisfies all the constraints
    if result:
        solution_grid = [[0 for _ in range(9)] for _ in range(9)]
        for (i, j), val in result.items():
            solution_grid[i][j] = val

        # Print out the grid before writing to file
        print("\nSolved Sudoku:")
        for row in solution_grid:
            print(' '.join(str(num) for num in row))
        print()

        # Write the solution to the output file
        with open(output_file, 'w') as f:
            for row in solution_grid:
```

```
            f.write(' '.join(str(num) for num in row) + '\n')
    print(f"The solution was written to {output_file}")

else:
    print("No solution found.")
    with open(output_file, 'w') as f:
        f.write("No solution found.\n")
```

Output1.txt:

```
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
```

Output2.txt:

```
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
```

Output3.txt:

```
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
```