AI ASSIGNMENT 2

HARIS AHMED

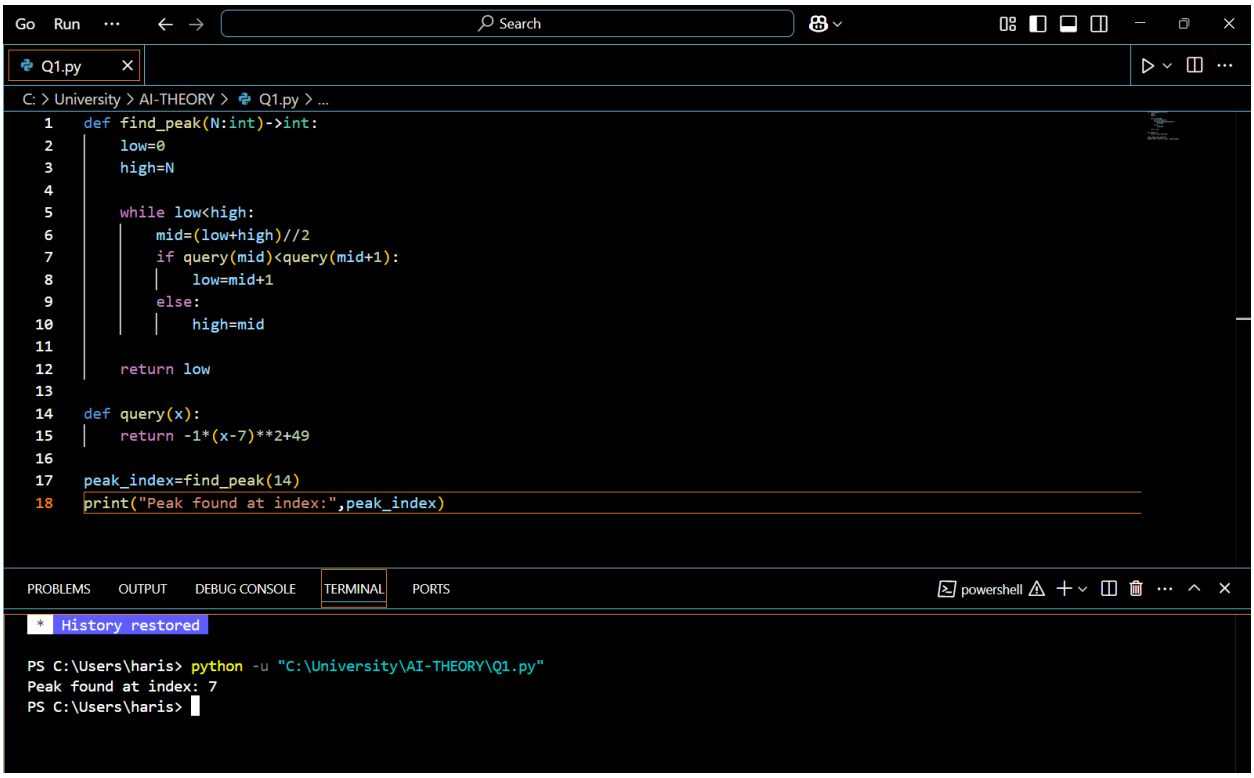
BSAI-4A

23K-6005

NOTE: I will first focus on completing the coding part of the assignment, ensuring all solvers are implemented and optimized, before moving on to the handwritten part later.

QUESTION 1:

```python
def find_peak(N:int)->int:
    low=0
    high=N

    while low<high:
        mid=(low+high)//2
        if query(mid)<query(mid+1):
            low=mid+1
        else:
            high=mid

    return low

def query(x):
    return -1*(x-7)**2+49

peak_index=find_peak(14)
print("Peak found at index:",peak_index)
```



QUESTION 2:

CODING PART:

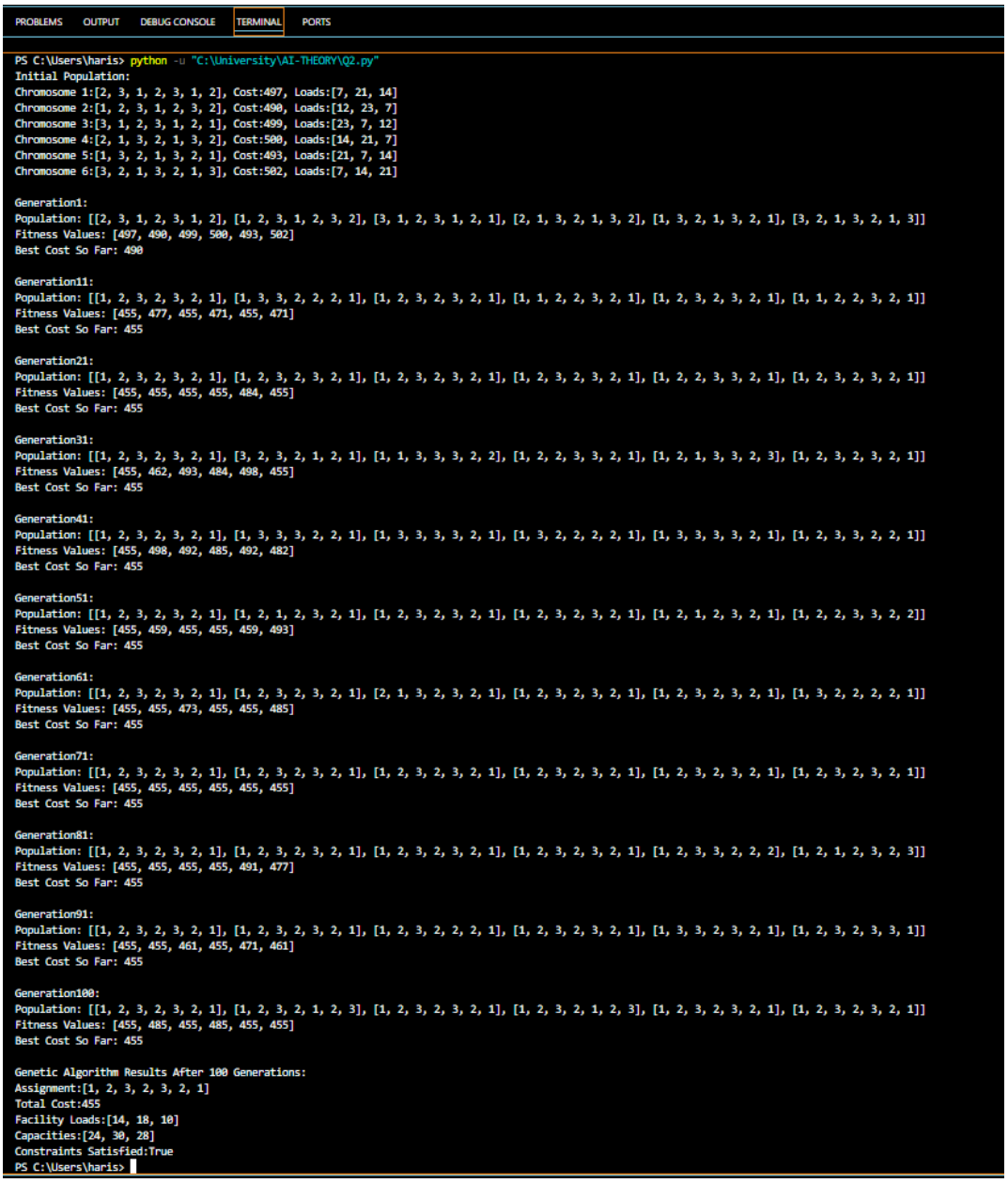
```
import random

#Data
task_times= [5,8,4,7,6,3,9]
facility_capacities =[24,30,28]
cost_matrix
=[[10,12,9],[15,14,16],[8,9,7],[12,10,13],[14,13,12],[9,8,10],[11,12,13]]

num_tasks=len(task_times)
num_facilities=len(facility_capacities)



population_size=6
crossover_rate= 0.8
mutation_rate= 0.2
generations =100
penalty_factor=1000
```

```
def calculate_fitness(assignment):
    total_cost =0
    facility_loads=[0] *num_facilities
    penalty=0
    for task, facility in enumerate(assignment):
        facility_idx=facility-1
        total_cost +=task_times[task]*cost_matrix[task][facility_idx]
        facility_loads[facility_idx]+=task_times[task]
    for load,capacity in zip(facility_loads,facility_capacities):
        if load > capacity:
            penalty +=penalty_factor*(load- capacity)
    return total_cost+penalty,facility_loads



def verify_assignment(assignment):
    fitness,loads=calculate_fitness(assignment)
    is_valid=all(load<=cap for load, cap in zip(loads,facility_capacities))
    print(f"Assignment:{assignment}")
    print(f"Total Cost:{fitness if penalty_factor ==0 else
fitness%penalty_factor}")
    print(f"Facility Loads:{loads}")
    print(f"Capacities:{facility_capacities}")
    print(f"Constraints Satisfied:{is_valid}")
    return fitness,is_valid



def create_population():
    return [
        [2,3,1,2,3,1,2],
        [1,2,3,1,2,3,2],
        [3,1,2,3,1,2,1],
        [2,1,3,2,1,3,2],
        [1,3,2,1,3,2,1],
        [3,2,1,3,2,1,3]
    ]



def select_parents(population,fitnesses):
    total_fitness=sum(1/f for f in fitnesses)
    probabilities=[(1/f)/total_fitness for f in fitnesses]
    return random.choices(population,weights=probabilities,k=2)
```

```
def crossover(parent1,parent2):
    if random.random()<crossover_rate:
        point=random.randint(1,num_tasks - 1)
        child1=parent1[:point]+parent2[point:]
        child2=parent2[:point]+parent1[point:]
        return child1,child2
    return parent1[:],parent2[:]



def mutate(assignment):
    if random.random() <mutation_rate:
        idx1, idx2=random.sample(range(num_tasks),2)
        assignment[idx1],assignment[idx2] =assignment[idx2],assignment[idx1]
    return assignment



def genetic_algorithm():
    population =create_population()
    best_solution =None
    best_fitness =float('inf')
    print("Initial Population:")
    for i,chrom in enumerate(population):
        fit, loads =calculate_fitness(chrom)
        print(f"Chromosome {i+1}:{chrom}, Cost:{fit}, Loads:{loads}")
    for gen in range(generations):
        fitnesses=[]
        for individual in population:
            fit, _ =calculate_fitness(individual)
            fitnesses.append(fit)
            if fit <best_fitness:
                best_fitness= fit
                best_solution= individual[:]
        if gen % 10 == 0 or gen ==generations - 1:
            print(f"\nGeneration{gen+1}:")
            print(f"Population: {population}")
            print(f"Fitness Values: {fitnesses}")
            print(f"Best Cost So Far: {best_fitness}")
        new_population =[best_solution[:]]  # Elitism: keep best
        while len(new_population)< population_size:
            parent1, parent2=select_parents(population, fitnesses)
            child1, child2=crossover(parent1, parent2)
            child1=mutate(child1)
            child2=mutate(child2)
```

```
            new_population.extend([child1, child2])
        population=new_population[:population_size]
    return best_solution,best_fitness



best_assignment,best_cost =genetic_algorithm()
print("\nGenetic Algorithm Results After 100 Generations: ")
verify_assignment(best_assignment)
```

OUTPUT:



QUESTION 3:

DIVIDED INTO 4 CODES:

1)mysolver1.py

```
import time

def make_board(puzzle):
    if len(puzzle) != 81:
        print("Error: puzzle string is wrog size")
        return None
    board=[]
    for i in range(9):
        row=[]
        for j in range(9):
            c=puzzle[i * 9 + j]
            if c=='.':
                row.append(0)
            elif c in '123456789':
                row.append(int(c))
            else:
                print("Error: bad charcter in puzzle")
                return None
        board.append(row)
    return board

# Checking if number fits
def is_ok(board,row,col,num):
    # Check row col
    for j in range(9):
        if board[row][j]==num and j!=col:
            return False
    for i in range(9):
        if board[i][col]==num and i!=row:
            return False

    # Checking 3x3
    box_row = (row // 3) * 3
    box_col = (col // 3) * 3
    for i in range(box_row, box_row + 3):
        for j in range(box_col, box_col + 3):
            if board[i][j] == num and (i, j) != (row, col):
                return False
```

```
    return True

# Get cells related to this one
def get_neighbors(row, col):
    neighbors=[]
    # Row and col neighbors
    for j in range(9):
        if j!=col:
            neighbors.append((row,j))
    for i in range(9):
        if i!=row:
            neighbors.append((i, col))

    # Box neighbors
    box_row= (row//3)*3
    box_col=(col//3)*3
    for i in range(box_row, box_row +3):
        for j in range(box_col,box_col +3):
            if (i,j)!=(row,col):
                neighbors.append((i,j))
    return neighbors



def ac3(board):
    # Seting possible vals for each cell
    possible = {}
    for i in range(9):
        for j in range(9):
            if board[i][j]==0:
                possible[(i,j)]=[1, 2, 3, 4, 5, 6, 7, 8, 9]
            else:
                possible[(i,j)] =[board[i][j]]

    # arc listss
    arcs =[]
    for i in range(9):
        for j in range(9):
            neighbors =get_neighbors(i, j)
            for n in neighbors:
                arcs.append(((i,j),n))

    # Process arcs slowly
    while arcs:
```

```
        (x_row,x_col), (y_row,y_col) = arcs.pop(0)  # Slow pop
        changed =False
        x_vals =possible[(x_row,x_col)].copy()
        for val in x_vals:
            y_vals=possible[(y_row,y_col)]
            if all(val==y_val for y_val in y_vals):
                possible[(x_row,x_col)].remove(val)
                changed=True
        if changed:
            if not possible[(x_row, x_col)]:
                return False,possible
            # Add arcs again
            neighbors=get_neighbors(x_row, x_col)
            for n in neighbors:
                if n !=(y_row,y_col):
                    arcs.append((n,(x_row, x_col)))

    return True, possible

#to find empty cells
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
def backtrack(board, possible):
    empty =find_empty(board)
    if not empty:
        return True
    row, col =empty
    for num in range(1, 10):
        if is_ok(board, row, col, num):
            board[row][col] = num
            if backtrack(board, possible):
                return True
            board[row][col] = 0
    return False



def solve_sudoku(puzzle):
```

```
    start = time.time()
    board = make_board(puzzle)
    if board is None:
        return None, time.time() -start
    for i in range(9):
        for j in range(9):
            if board[i][j]!=0:
                num = board[i][j]
                board[i][j]=0
                if not is_ok(board, i, j, num):
                    return None, time.time() - start
                board[i][j] =num
    #Running ac3
    ok, possible = ac3(board)
    if not ok:
        return None, time.time() - start
    #filling easy cells
    for i in range(9):
        for j in range(9):
            if board[i][j]==0 and len(possible[(i, j)])==1:
                board[i][j]=possible[(i, j)][0]
    #Backtracking
    backtrack(board, possible)
    return board, time.time()-start

#board to string
def board_to_string(board):
    if board is None:
        return "Nosolution"
    result=""
    for i in range(9):
        for j in range(9):
            result+=str(board[i][j])
    return result

# Main program
try:
    file =open('sudoku.txt', 'r')
    puzzles=[]
    for line in file:
        line=line.strip()
        if line:
            puzzles.append(line)
    file.close()
```

```
    for index in range(len(puzzles)):
        puzzle = puzzles[index]
        print("\n Puzzle", index + 1, ": ")
        print("Puzzle string :", puzzle)
        print("Solvingggg...")
        solution, elapsed = solve_sudoku(puzzle)
        print("Solution :", board_to_string(solution))
        print("Total Time :", "{:.5f}s".format(elapsed))
except:
    print("Error: something went wrong")
```

2) chatgpt_solver1.py

```
import time
from collections import deque

def make_board(s):
    """Convert puzzle string to 9x9 grid."""
    if len(s) != 81 or not all(c in '.123456789' for c in s):
        raise ValueError("Invalid puzzle string")
    board = [[int(c) if c != '.' else 0 for c in s[i*9:(i+1)*9]] for i in
range(9)]
    for i in range(9):
        for j in range(9):
            if board[i][j] != 0 and not is_valid(board, i, j, board[i][j]):
                raise ValueError("Initial board has conflicts")
    return board

def precompute_neighbors():
    """Precompute neighbors for each cell (row, column, box)."""
    neighbors = {}
    for i in range(9):
        for j in range(9):
            n = set()
            for y in range(9):
                if y != j:
                    n.add((i, y))
            for x in range(9):
                if x != i:
                    n.add((x, j))
            start_i, start_j = (i // 3) * 3, (j // 3) * 3
            for x in range(start_i, start_i + 3):
```

```
                for y in range(start_j, start_j + 3):
                    if (x, y) != (i, j):
                        n.add((x, y))
            neighbors[(i, j)] = list(n)
    return neighbors

NEIGHBORS = precompute_neighbors()

def ac3(board):
    """Enforce arc consistency using AC-3."""
    domains = {(i, j): [board[i][j]] if board[i][j] != 0 else list(range(1, 10))
               for i in range(9) for j in range(9)}
    queue = deque([((i, j), (ni, nj)) for i in range(9) for j in range(9)
                   for (ni, nj) in NEIGHBORS[(i, j)]])

    while queue:
        (xi, xj), (yi, yj) = queue.popleft()
        revised = False
        original = domains[(xi, xj)].copy()
        for val in original:
            if not any(val != d for d in domains[(yi, yj)]):
                domains[(xi, xj)].remove(val)
                revised = True
        if revised:
            if not domains[(xi, xj)]:
                return False, domains
            for (ni, nj) in NEIGHBORS[(xi, xj)]:
                if (ni, nj) != (yi, yj):
                    queue.append(((ni, nj), (xi, xj)))
    return True, domains

def is_valid(board, i, j, num):
    """Check if placing num at (i, j) is valid."""
    for y in range(9):
        if board[i][y] == num and y != j:
            return False
    for x in range(9):
        if board[x][j] == num and x != i:
            return False
    start_i, start_j = (i // 3) * 3, (j // 3) * 3
    for x in range(start_i, start_i + 3):
        for y in range(start_j, start_j + 3):
            if board[x][y] == num and (x, y) != (i, j):
```

```
                return False
    return True

def find_empty_mrv(board, domains):
    """Find the empty cell with the minimum remaining values (MRV)."""
    min_vals, best_pos = float('inf'), None
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                num_vals = len(domains[(i, j)])
                if num_vals < min_vals:
                    min_vals, best_pos = num_vals, (i, j)
    return best_pos

def backtrack(board, domains):
    """Solve using backtracking with MRV heuristic."""
    pos = find_empty_mrv(board, domains)
    if not pos:
        return True
    i, j = pos
    # Copy domains to avoid modifying during backtracking
    for num in domains[(i, j)].copy():
        if is_valid(board, i, j, num):
            board[i][j] = num
            # Update domains temporarily for neighbors
            old_domains = {k: v.copy() for k, v in domains.items()}
            for (ni, nj) in NEIGHBORS[(i, j)]:
                if num in domains[(ni, nj)]:
                    domains[(ni, nj)].remove(num)
            if backtrack(board, domains):
                return True
            # Restore board and domains
            board[i][j] = 0
            domains.update(old_domains)
    return False

def solve_sudoku(puzzle):
    """Solve the puzzle and return the solution grid and elapsed time."""
    start = time.time()
    try:
        board = make_board(puzzle)
        success, domains = ac3(board)
        if not success:
```

```
            return None, time.time() - start
        for i in range(9):
            for j in range(9):
                if board[i][j] == 0 and len(domains[(i, j)]) == 1:
                    board[i][j] = domains[(i, j)][0]
                    # Update neighbors' domains
                    for (ni, nj) in NEIGHBORS[(i, j)]:
                        if board[i][j] in domains[(ni, nj)]:
                            domains[(ni, nj)].remove(board[i][j])
        success = backtrack(board, domains)
        return board if success else None, time.time() - start
    except ValueError as e:
        return None, time.time() - start
    except Exception as e:
        print(f"Error in solving: {e}")
        return None, time.time() - start

def board_to_string(board):
    """Convert a 9x9 grid to an 81-character string."""
    if board is None:
        return "No solution"
    return ''.join(str(cell) for row in board for cell in row)

try:
    with open('sudoku.txt', 'r') as f:
        puzzles = [line.strip() for line in f]
    for idx, puzzle in enumerate(puzzles, 1):
        print(f"\n--- Puzzle {idx} ---")
        print(f"Puzzle string: {puzzle}")
        print("Solving...")
        solution, elapsed = solve_sudoku(puzzle)
        print(f"Solution: {board_to_string(solution)}")
        print(f"Total Time: {elapsed:.5f}s")
except FileNotFoundError:
    print("Error: sudoku.txt not found")
except ValueError as e:
    print(f"Error: {e}")
```

3)ortools_solver1.py

```
#!/usr/bin/env python3
# Copyright 2010-2025 Google LLC
# Licensed under the Apache License, Version 2.0 (the "License");
```

```
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""This model implements a sudoku solver using OR-Tools CP-SAT."""

from ortools.sat.python import cp_model
import time

def make_board(s):
    """Convert puzzle string to 9x9 grid."""
    if len(s) != 81 or not all(c in '.123456789' for c in s):
        raise ValueError("Invalid puzzle string")
    return [[int(c) if c != '.' else 0 for c in s[i*9:(i+1)*9]] for i in
range(9)]

def board_to_string(board):
    """Convert a 9x9 grid to an 81-character string."""
    if board is None:
        return "No solution"
    return ''.join(str(cell) for row in board for cell in row)

def solve_sudoku(puzzle):
    """Solve the puzzle using OR-Tools CP-SAT and return the solution grid and
elapsed time."""
    start = time.time()
    initial_grid = make_board(puzzle)
    model = cp_model.CpModel()
    cell_size = 3
    line_size = cell_size**2
    line = list(range(0, line_size))
    cell = list(range(0, cell_size))
    grid = {}
    for i in line:
        for j in line:
            grid[(i, j)] = model.new_int_var(1, line_size, f"grid {i} {j}")
```

```
    for i in line:
        model.add_all_different(grid[(i, j)] for j in line)
    for j in line:
        model.add_all_different(grid[(i, j)] for i in line)
    for i in cell:
        for j in cell:
            one_cell = []
            for di in cell:
                for dj in cell:
                    one_cell.append(grid[(i * cell_size + di, j * cell_size +
dj)])
            model.add_all_different(one_cell)
    for i in line:
        for j in line:
            if initial_grid[i][j]:
                model.add(grid[(i, j)] == initial_grid[i][j])
    solver = cp_model.CpSolver()
    status = solver.solve(model)
    elapsed = time.time() - start
    if status == cp_model.OPTIMAL or status == cp_model.FEASIBLE:
        solution = [[0] * 9 for _ in range(9)]
        for i in line:
            for j in line:
                solution[i][j] = int(solver.value(grid[(i, j)]))
        return solution, elapsed
    return None, elapsed

try:
    with open('sudoku.txt', 'r') as f:
        puzzles = [line.strip() for line in f]
    for idx, puzzle in enumerate(puzzles, 1):
        print(f"\n--- Puzzle {idx} ---")
        print(f"Puzzle string: {puzzle}")
        print("Solving...")
        solution, elapsed = solve_sudoku(puzzle)
        print(f"Solution: {board_to_string(solution)}")
        print(f"Total Time: {elapsed:.5f}s")
except FileNotFoundError:
    print("Error: sudoku.txt not found")
except ValueError as e:
    print(f"Error: {e}")
```

4)optimized_mysolver1

```
# OPTIMIZATIONSS:
# used collections.deque in ac3 for O(1) pop operations, speeding up constraint
propagation.
# applied MRV heuristic in find_empty to select the cell with the fewest possible
values, reducing backtracking steps.
# modified backtrack to use possible domains from ac3
# added dynamic domain updtes for neighbring cells in backtrack n restoring them
on failureand after filling easy cells in solve_sudoku.

import time
from collections import deque

# Make a board from the puzzle string
def make_board(puzzle):
    if len(puzzle)!=81:
        print("Error: puzzle string is wrog size")
        return None
    board=[]
    for i in range(9):
        row=[]
        for j in range(9):
            c=puzzle[i*9+j]
            if c=='.':
                row.append(0)
            elif c in '123456789':
                row.append(int(c))
            else:
                print("Error: bad charcter in puzzle")
                return None
        board.append(row)
    return board

# Checking if number fits
def is_ok(board,row,col,num):
    # Check row col
    for j in range(9):
        if board[row][j]==num and j!=col:
            return False
    for i in range(9):
        if board[i][col]==num and i!=row:
            return False

    # Checking 3x3
```

```
    box_row=(row//3)*3
    box_col=(col//3)*3
    for i in range(box_row,box_row+3):
        for j in range(box_col,box_col+3):
            if board[i][j]==num and (i,j)!=(row,col):
                return False
    return True

# Get cells related to this one
def get_neighbors(row,col):
    neighbors=[]
    # Row and col neighbors
    for j in range(9):
        if j!=col:
            neighbors.append((row,j))
    for i in range(9):
        if i!=row:
            neighbors.append((i,col))

    # Box neighbors
    box_row=(row//3)*3
    box_col=(col//3)*3
    for i in range(box_row,box_row+3):
        for j in range(box_col,box_col+3):
            if (i,j)!=(row,col):
                neighbors.append((i,j))
    return neighbors

def ac3(board):
    # Seting possible vals for each cell
    possible={}
    for i in range(9):
        for j in range(9):
            if board[i][j]==0:
                possible[(i,j)]=[1,2,3,4,5,6,7,8,9]
            else:
                possible[(i,j)]=[board[i][j]]

    # arc listss
    arcs=deque()
    for i in range(9):
        for j in range(9):
            neighbors=get_neighbors(i,j)
```

```
            for n in neighbors:
                arcs.append(((i,j),n))

    # Process arcs
    while arcs:
        (x_row,x_col),(y_row,y_col)=arcs.popleft()
        changed=False
        x_vals=possible[(x_row,x_col)].copy()
        for val in x_vals:
            y_vals=possible[(y_row,y_col)]
            if all(val==y_val for y_val in y_vals):
                possible[(x_row,x_col)].remove(val)
                changed=True
        if changed:
            if not possible[(x_row,x_col)]:
                return False,possible
            neighbors=get_neighbors(x_row,x_col)
            for n in neighbors:
                if n!=(y_row,y_col):
                    arcs.append((n,(x_row,x_col)))
    return True,possible

# to find empty cells with fewest options
def find_empty(board,possible):
    best=None
    fewest=10
    for i in range(9):
        for j in range(9):
            if board[i][j]==0:
                count=len(possible[(i,j)])
                if count<fewest:
                    fewest=count
                    best=(i,j)
    return best

# Backtracking
def backtrack(board,possible):
    empty=find_empty(board,possible)
    if not empty:
        return True
    row,col=empty
    for num in possible[(row,col)].copy():
        if is_ok(board,row,col,num):
```

```
            board[row][col]=num
            old_possible={}
            for i in range(9):
                for j in range(9):
                    old_possible[(i,j)]=possible[(i,j)].copy()
            for ni,nj in get_neighbors(row,col):
                if num in possible[(ni,nj)]:
                    possible[(ni,nj)].remove(num)
            if backtrack(board,possible):
                return True
            board[row][col]=0
            for i in range(9):
                for j in range(9):
                    possible[(i,j)]=old_possible[(i,j)]
    return False

def solve_sudoku(puzzle):
    start=time.time()
    board=make_board(puzzle)
    if board is None:
        return None,time.time()-start
    for i in range(9):
        for j in range(9):
            if board[i][j]!=0:
                num=board[i][j]
                board[i][j]=0
                if not is_ok(board,i,j,num):
                    return None,time.time()-start
                board[i][j]=num
    # Running ac3
    ok,possible=ac3(board)
    if not ok:
        return None,time.time()-start
    # filling easy cells
    for i in range(9):
        for j in range(9):
            if board[i][j]==0 and len(possible[(i,j)])==1:
                board[i][j]=possible[(i,j)][0]
                for ni,nj in get_neighbors(i,j):
                    if board[i][j] in possible[(ni,nj)]:
                        possible[(ni,nj)].remove(board[i][j])
    # Backtracking
    if not backtrack(board,possible):
        return None,time.time()-start
```

```
    return board,time.time()-start

# board to string
def board_to_string(board):
    if board is None:
        return "Nosolution"
    result=""
    for i in range(9):
        for j in range(9):
            result+=str(board[i][j])
    return result

# Main program
try:
    file=open('sudoku.txt','r')
    puzzles=[]
    for line in file:
        line=line.strip()
        if line:
            puzzles.append(line)
    file.close()

    for index in range(len(puzzles)):
        puzzle=puzzles[index]
        print("\n Puzzle",index+1,": ")
        print("Puzzle string :",puzzle)
        print("Solvingggg...")
        solution,elapsed=solve_sudoku(puzzle)
        print("Solution :",board_to_string(solution))
        print("Total Time :","{:.5f}s".format(elapsed))
except:
    print("Error: something went wrong")
```

OUTPUT FOR ALL 4 IN QUESTION 3:

```
PS C:\Users\haris> python -u "C:\University\AI-THEORY\mysolver1.py"

 Puzzle 1 :
Puzzle
string : ..3.2.6..9..3.5..1..18.64....81.29..7.......8..67.82....26.95..8..2.3..9
..5.1.3..
Solvingggg...
```

```
Solution :
483921657967345821251876493548132976729564138136798245372689514814253769695417382
Total Time : 0.07096s

 Puzzle 2 :
Puzzle string :
8.........36......7..9.2...5...7.......457.....1...3...1....68..85...1..9....4..
Solvingggg...
Solution :
812753649943682175675491283154237896369845721287169534521974368438526917796318452
Total Time : 0.30538s

 Puzzle 3 :
Puzzle string :
53..7....6..195....98....6.8...6...34..8.3..17...2...6.6....28....419..5....8..79
Solvingggg...
Solution :
534678912672195348198342567859761423426853791713924856961537284287419635345286179
Total Time : 0.08658s

PS C:\Users\haris> python -u "C:\University\AI-THEORY\chatgpt_solver1.py"

--- Puzzle 1 ---
Puzzle
string: ..3.2.6..9..3.5..1..18.64....81.29..7.......8..67.82....26.95..8..2.3..9.
.5.1.3..
Solving...
Solution:
483921657967345821251876493548132976729564138136798245372689514814253769695417382
Total Time: 0.01212s

--- Puzzle 2 ---
Puzzle string:
8.........36......7..9.2...5...7.......457.....1...3...1....68..85...1..9....4..
Solving...
Solution:
812753649943682175675491283154237896369845721287169534521974368438526917796318452
Total Time: 0.16507s

--- Puzzle 3 ---
Puzzle string:
53..7....6..195....98....6.8...6...34..8.3..17...2...6.6....28....419..5....8..79
Solving...
```

```
Solution:
534678912672195348198342567859761423426853791713924856961537284287419635345286179
Total Time: 0.02292s

PS C:\Users\haris> python -u "C:\University\AI-THEORY\ortools_solver1.py"

load C:\Users\haris\AppData\Local\Programs\Python\Python312\Lib\site-
packages\ortools\.libs\zlib1.dll...
load C:\Users\haris\AppData\Local\Programs\Python\Python312\Lib\site-
packages\ortools\.libs\abseil_dll.dll...
load C:\Users\haris\AppData\Local\Programs\Python\Python312\Lib\site-
packages\ortools\.libs\utf8_validity.dll...
load C:\Users\haris\AppData\Local\Programs\Python\Python312\Lib\site-
packages\ortools\.libs\re2.dll...
load C:\Users\haris\AppData\Local\Programs\Python\Python312\Lib\site-
packages\ortools\.libs\libprotobuf.dll...
load C:\Users\haris\AppData\Local\Programs\Python\Python312\Lib\site-
packages\ortools\.libs\highs.dll...
load C:\Users\haris\AppData\Local\Programs\Python\Python312\Lib\site-
packages\ortools\.libs\ortools.dll...

--- Puzzle 1 ---
Puzzle
string: ..3.2.6..9..3.5..1..18.64....81.29..7.......8..67.82....26.95..8..2.3..9.
.5.1.3..
Solving...
Solution:
483921657967345821251876493548132976729564138136798245372689514814253769695417382
Total Time: 0.01867s

--- Puzzle 2 ---
Puzzle string:
8.........36......7..9.2...5...7.......457.....1...3...1....68..85...1..9....4..
Solving...
Solution:
812753649943682175675491283154237896369845721287169534521974368438526917796318452
Total Time: 0.07543s

--- Puzzle 3 ---
Puzzle string:
53..7....6..195....98....6.8...6...34..8.3..17...2...6.6....28....419..5....8..79
Solving...
```

```
Solution:
534678912672195348198342567859761423426853791713924856961537284287419635345286179
Total Time: 0.01644s

PS C:\Users\haris> python -u "C:\University\AI-THEORY\optimizedmysolver1.py"

 Puzzle 1 :
Puzzle
string : ..3.2.6..9..3.5..1..18.64....81.29..7.......8..67.82....26.95..8..2.3..9
..5.1.3..
Solvingggg...
Solution :
483921657967345821251876493548132976729564138136798245372689514814253769695417382
Total Time : 0.01682s

 Puzzle 2 :
Puzzle string :
8.........36......7..9.2...5...7.......457.....1...3...1....68..85...1..9....4..
Solvingggg...
Solution :
812753649943682175675491283154237896369845721287169534521974368438526917796318452
Total Time : 0.29064s

 Puzzle 3 :
Puzzle string :
53..7....6..195....98....6.8...6...34..8.3..17...2...6.6....28....419..5....8..79
Solvingggg...
Solution :
534678912672195348198342567859761423426853791713924856961537284287419635345286179
Total Time : 0.02285s

PS C:\Users\haris>
```

Time Comparison Table

Solver	Puzzle 1	Puzzle 2	Puzzle 3	Total Time
mysolver1.py	0.07096s	0.30538s	0.08658s	0.46292s
optimizedmysolver1.py	0.01682s	0.29064s	0.02285s	0.33031s
chatgpt_solver1.py	0.01212s	0.16507s	0.02292s	0.20011s

ortools_solver1.py	0.01867s	0.07543s	0.01644s	0.11054s

Ranking:

1. **ortools_solver1.py**: 0.11054s - Fastest overall, especially on Puzzle 2 (0.07543s).
2. **chatgpt_solver1.py**: 0.20011s - Second fastest, strong on Puzzle 1 (0.01212s).
3. **optimizedmysolver1.py**: 0.33031s - Third, improved over mysolver1.py but slower on Puzzle 2.
4. **mysolver1.py**: 0.46292s - Slowest, as expected due to inefficiencies.

Haris Ahmed
23k-6005
BSAI-4A

ARTIFICIAL INTELLIGENCE ASSIGNMENT Date: _____

HANDWRITTEN TASKS

QUESTION 2: DRY RUN OF GENETIC ALGORITHM

Performing 2 iterations:

Step 1: Initialize Population (6 chromosomes)

C1: [2,3,1,2,3,1,2]

C2: [1,2,3,1,2,3,2]

C3: [3,1,2,3,1,2,1]

C4: [2,1,3,2,1,3,2]

C5: [1,3,2,1,3,2,1]

C6: [3,2,1,3,2,1,3]

Step 2: Evaluate fitness

C1: 2,3,1,2,3,1,2	C2: 1,2,3,1,2,3,2	C3: 3,1,2,3,1,2,1
Costs:	Costs:	Costs:
Task 1, F2 = 5*12 = 60	Task 1, F1 = 5×10 = 50	Task 2, F3 = 5×9 = 45
Task 2, F3 = 8×16 = 128	Task 2, F2 = 8×14 = 112	Task 2, F1 = 8×15 = 120
Task 3, F1 = 4×8 = 32	Task 3, F1 = 4×7 = 28	Task 3, F2 = 4×9 = 36
Task 4, F2 = 7×10 = 70	Task 4, F1 = 7×12 = 84	Task 4, F3 = 7×13 = 91
Task 5, F3 = 6×12 = 72	Task 5, F2 = 6×13 = 78	Task 5, F2 = 6×14 = 84
Task 6, F1 = 3×9 = 27	Task 6, F3 = 3×10 = 30	Task 6, F2 = 3×8 = 24
Task 7, F2 = 9×12 = 108	Task 7, F2 = 9×12 = 108	Task 7, F1 = 9×11 = 99
Total cost: 497	Total cost: 490	Total cost: 499
Loads:	Loads:	Loads:
F1: 4+3 = 7	F1: 5+7 = 12	F1: 8+6+9 = 23
F2: 5+7+9 = 21	F2: 8+6+9 = 23	F2: 4+3 = 7
F3: 8+6 = 14	F3: 4+3 = 7	F3: 5+7 = 12
Capacities: [24,30,28]	Capacities: [24,30,28]	Capacities: [24,30,28]
All within capacities.	All within capacities	All within capacities
Fitness: 497	Fitness: 490	Fitness: 499

Date: _____

C_4: [2,1,3,2,1,3,2]	C_5: [1,3,2,1,3,2,1]	C_6: [3,2,1,3,2,1,3]
Costs:	Costs:	Costs:
Task 2, f_2 : 5×12 = 60	Task 1, f_1 = 5 × 10 = 50	Task 1, f_3 = 5 × 9 = 45
Task 2, f_1 : 8×15 = 120	Task 2, f_3 = 8 × 16 = 128	Task 2, f_2 = 8 × 14 = 112
Task 3, f_3 : 4×7 = 28	Task 3, f_2 = 4 × 9 = 36	Task 3, f_1 = 4 × 8 = 32
Task 4, f_2 : 7×10 = 70	Task 4, f_1 = 7 × 12 = 84	Task 4, f_3 = 7 × 13 = 91
Task 5, f_1 : 6×14 = 84	Task 5, f_3 = 6 × 12 = 72	Task 5, f_2 = 6 × 13 = 78
Task 6, f_3 : 3×10 = 30	Task 6, f_2 = 3 × 8 = 24	Task 6, f_1 = 3 × 9 = 27
Task 7, f_2 : 9×12 = 108	Task 7, f_1 = 9 × 11 = 99	Task 7, f_3 = 9 × 13 = 117
Total cost = 500	Total cost = 493	Total cost = 502
f_1 = 8+6 = 14	loads: f_1: 5+7+9 = 21	loads: f_1: 4+3 = 7
f_2 = 5+7+9 = 21	f_2 = 4+3 = 7	f_2 = 8+6 = 14
f_3 = 4+3 = 7	f_3 = 8+6 = 14	f_3 = 5+7+9 = 21
Capacities: [24, 30, 28]	Capacities: [24, 30, 28]	Capacities: [24, 30, 28]
All within capacities	All within capacities	All within capacities
Fitness = 500	Fitness = 493.	Fitness = 502

Fitness values: [493, 490, 499, 500, 493, 502]

Step #3: Selection (Roulette wheel) $1/cost$

Inverses:	Probabilities:
C_1: $1/493$ = 0.002012	C_1: 0.002012 / 0.012077 = 0.167
C_2: $1/490$ = 0.002041	C_2: 0.002041 / 0.012077 = 0.169
C_3: $1/499$ = 0.002004	C_3: 0.002004 / 0.012077 = 0.166
C_4: $1/500$ = 0.002000	C_4: 0.002000 / 0.012077 = 0.166
C_5: $1/493$ = 0.002028	C_5: 0.002028 / 0.012077 = 0.168
C_6: $1/502$ = 0.001992	C_6: 0.001992 / 0.012077 = 0.165
Total: 0.012077	

Selecting $C_2, C_5, C_1, C_3, C_2, C_5$ (C_2 and C_5 favoured due to slightly lower cost)

Date: _____

Step #4: Crossover (80% chance)

Pair 1: C_2: [1,2,3,1,2,3,2] C_5: [1,3,2,1,3,2,1]

Crossover at Pos 2:

child 1: [1,2,2,1,3,2,1]

child 2: [1,3,3,1,2,3,2]

Pair 2: C_1: [2,3,1,2,3,1,2] C_3: [3,1,2,3,1,2,1]

Crossover at Pos 2:

child 3: [2,3,2,3,1,2,1]

child 4: [3,1,1,2,3,1,2]

Pair 3: C_4: [1,2,3,1,2,3,2] C_6: [1,3,2,1,3,3,1]

Crossover at Pos 5

child 5: [1,2,3,1,2,1]

child 6: [1,3,2,1,3,3,1]

Step #5: Mutation: (20% chance)

Child 1: [1,2,2,1,3,2,1] → No mutation

Child 2: [1,3,3,1,2,3,2] → 20% chance. Swapping pos 3 and 6: [1,3,3,1,2,2,3]

Child 3: [2,3,2,3,1,2,1] → No mutation

Child 4: [3,1,1,2,3,1,2] → No mutation

Child 5: [1,2,3,1,2,2,1] → 20% chance. Swapping pos 2 and 4: [1,1,3,2,2,2,1]

Child 6: [1,3,2,1,3,3,2] → No mutation

New Population:

[1,2,2,1,3,2,1] [3,1,2,3,1,2]

[1,3,3,1,2,2,3] [1,1,3,2,2,2,1]

[2,3,2,3,1,2,1] [1,3,2,1,3,3,2]

Evaluating fitness of next Population

C_1: [1,2,2,1,3,2,1]

Total cost: 50+112+36+84+72+24+99: 477

Loads: F_1: 5+7+9=21, F_2: 8+4+3=15, F_3: 6

All within capacities, Fitness: 477

C_2: [1,3,3,1,2,2,3]

Total cost: 50+128+38+84+78+24+117 = 509

Loads: F_1: 5+7=12, F_2: 8+6+3=17, F_3: 4+9=13

All within capacities, Fitness: 509

C_3: [2,3,2,3,1,2,1]

Total cost: 60+128+36+91+84+24+99=522

Loads: F_1: 6+9=15, F_2: 5+4+3=12, F_3: 8+7=15

All within capacities, Fitness: 522

C_4: [3,1,1,2,3,1,2]

Total cost: 45+120+32+70+92+27+108 = 494

Loads: F_1: 8+4+3=15, F_2: 7+9=16, F_3: 5+6=11

All within capacities, Fitness: 494.

C_5: [1,1,3,2,2,2,1]

Total cost: 50+120+28+70+78+24+99: 469

Loads: F_1: 5+8+9=22, F_2: 7+6+3=16, F_3: 4

All within capacities, Fitness: 469

C_6: [1,2,2,1,3,3,2]

Total cost: 50+128+36+84+72+30+108: 508

Loads: F_1: 5+7=12, F_2: 8+9=17, F_3: 4+6+3=13

All within capacities, Fitness: 508

Date: _____

Fitness: [471, 505, 522, 477, 469, 508]

Conclusion: After 1 iteration the GA improved the best cost from 490 to 469 with the Assignment [1, 1, 3, 2, 2, 2, 1] meeting all constraints.

QUESTION 4:

	X 2 3	
	X 5 6	
	0 0 9	
Three X: +1000		
Three O: -1000		
Two X's, one empty: +100		
One X, Two empty: +10		
Two O's, one empty: -100		
One O, Two empty: -10		
Else: 0		
V_sum = Sum_R + Sum_C + Sum_D		

STATES FOR X AT POS 2

	X X 3	
	X 5 6	
	0 0 9	

State 1: O at 3

R1: 0	X X O	
R2: +10	X 5 6	
R3: -100	0 0 9	
C1: 0		
C2: 0		
C3: -10		
D1: +10		
D2: -100		
Sum_R: -90	Sum_C: -10	Sum_D: -90
V_Sum: -190.		

State 2: O at 5

R1: +100	X X 3	
R2: 0	X O 6	
R3: -100	0 0 9	
C1: 0		
C2: 0		
C3: 0		
D1: 0		
D2: -100		
Sum_R: 0	Sum_C: 0	Sum_D: -100
V_Sum: -100		

State 3: O at 6

R1: +100	X X 3	
R2: 0	X 5 O	
R3: -100	0 0 9	
C1: 0		
C2: 0		
C3: -10		
D1: +10		
D2: -10		
Sum_R: 0	Sum_C: -10	Sum_D: 0
V_Sum: -10		

Date: _____

State 4: 0 at 9

R_1: +100 X X 3

R_2: +10 X 5 6

R_3: -1000 0 0 0

C_1: 0

C_2: 0

C_3: -10

D_1: 0

D_2: -10

Sum_R: -890, Sum_C: -10, Sum_D: -10

V_Sum: -910

STATES FOR X AT Pos 9

X 2 3

X 5 6

0 0 X

STATE 5: 0 at 2

R_1: 0 X 0 3

R_2: +10 X 5 6

R_3: 0 0 0 X

C_1: 0

C_2: -100

C_3: +10

D_1: +100

D_2: -10

Sum_R: +10, Sum_C: -90, Sum_D: 90

V_Sum: 10

STATE 6: 0 at 3

R_1: 0 X 2 0

R_2: +10 X 5 6

R_3: 0 0 0 X

C_1: 0

C_2: -100

C_3: +10

D_1: +100

D_2: -10

Sum_R: +10, Sum_C: -90, Sum_D: 90

V_Sum: 90

STATE 7: 0 at 5

R_1: +10 X 2 3

R_2: 0 X 0 6

R_3: 0 0 0 X

C_1: 0

C_2: -100

C_3: +10

D_1: 0

D_2: -100

Sum_R: +10, Sum_C: -90, Sum_D: -100

V_Sum: -180

STATE 8: 0 at 6

R_1: +10 X 2 3

R_2: 0 X 5 0

R_3: 0 0 0 X

C_1: 0

C_2: -10

C_3: 0

D_1: +100

D_2: -10

Sum_R: +10, Sum_C: -10, Sum_D: 90

V_Sum: 90

Date: _____

Minimax Decision:

X at 2: V. Sum. {-150, -100, -10, -910} → minimum: -910

X at 9: V. Sum: {10, 0, -180, 90} → minimum: -180

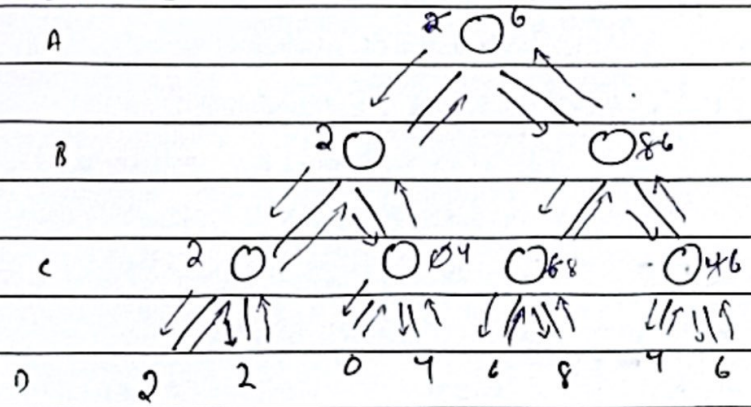
X chooses max(minimum): max(-910, -180) : -180

BEST MOVE:

Place x at Position 9, as it yields a minimum V. sum of -180, better than -910 for position 2.

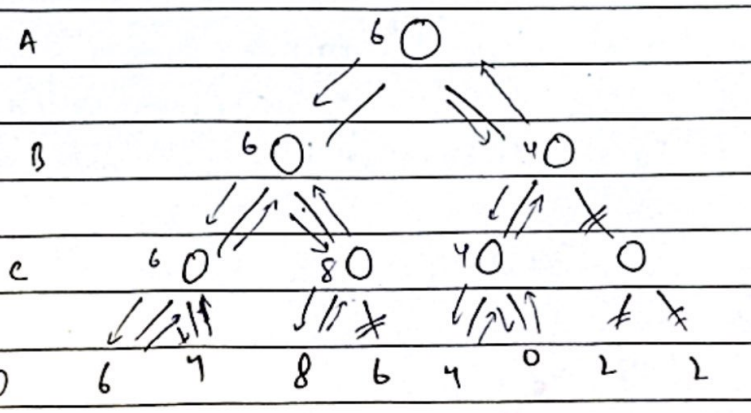
QUESTION 5:

A



All Branches evaluated, none of them pruned, No parts cut, winning Path is A → B2 → C4 → D8. As All are computed, none are struked out.

B



All Branches under B_1 except $D_4(6)$ is evaluated, $D_4(6)$ pruned when C2's 8 exceeds B_1 6. Under B_2, C_3 yields 4, setting B_2 4, Since A already has 6 from B_1, and C_4 max is less than 6, C_4 including D_7 and D_8 is pruned, as it can't improve A's value. The winning path is $A \to B_1 \to C_1 \to D_1(6)$; struck out values are $6(D_4)$, $8(D_7)$, $2(D_8)$

QUESTION 6.

Part (a)

1) PLAYERS:

max (Defenders): An AI driven IDS, tasked with protecting the network from cyber threats.

min (Attacker): Aims to penetrate the network through various attack methods.

2 DECISION MAKING:

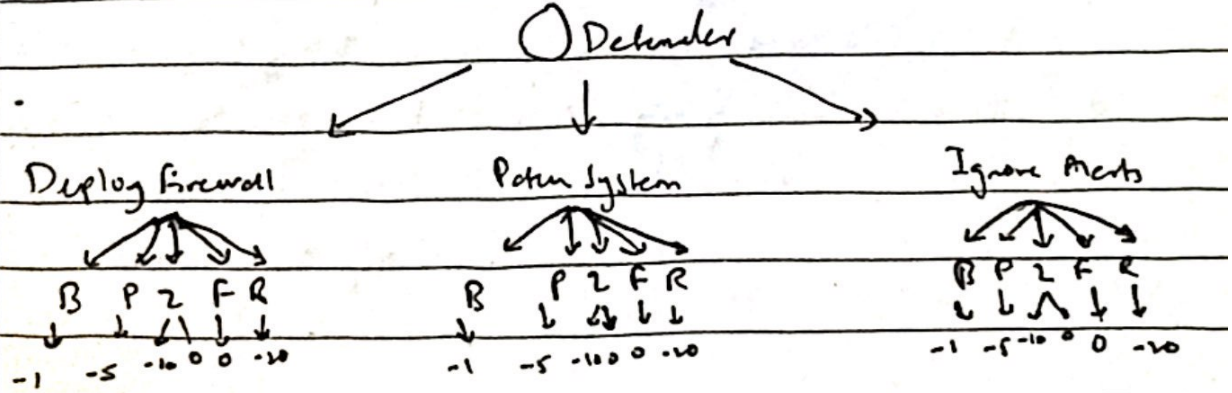
max (Defender): Decides on actions such as setting up firewalls, applying patches, or dismissing alerts to reduce damage while balancing resource use.

Min (Attacker): Picks Attacks like Brute force, Phishing, Zero-Day Exploit, Fake, or Bad to inflict maximum harm on the network.

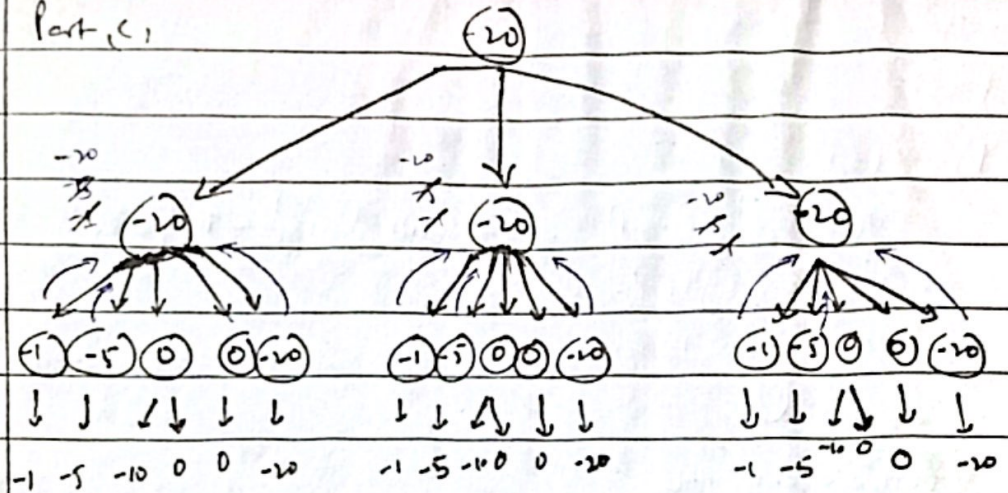
3 Stochastic Elements.

Attacks with probabilities, such as zero-Day Exploits (eg 50% success chance), create unpredictability, pushing the defender to adjust from worst case planning to strategies based on expected value (eg using Expectimax)

Part (b)



Part (c)



Part (d)

(1) Success (50%) = Damage -10

Fail (50%) = Damage = 0

Expected value = $(0.5 \times -10) + (0.5 \times 0) = \boxed{5}$

Zero-Day Exploit, on average, causes -5 damage to the system

(2) Minimax: The Defender prepares for the worst outcome (eg Zero-Day exploit succeeding at -10) typically choosing to deploy a firewall for strong defence.

Expectimax: The Defender considers the expected value (-5) for zero Day Exploits, viewing them as less severe. It might prefer Patching the system or even ignoring Alerts if fake Attacks are probable, taking a more calculated approach rather than always overprotecting.