

Project report

Artificial Intelligence

# QUESTION-1:

import pygame

import heapq

import random

import time

# Define colors

BLACK = (0, 0, 0)

WHITE = (255, 255, 255)

GREEN = (0, 255, 0)

RED = (255, 0, 0)

BLUE = (0, 0, 255)

BROWN = (165, 42, 42)

GRAY = (128, 128, 128)

YELLOW = (255, 255, 0)

# Define grid size

GRID\_SIZE = 15

# Define window size

WINDOW\_SIZE = [600, 600]

# Define cell size

CELL\_SIZE = WINDOW\_SIZE[0] // GRID\_SIZE

# Define obstacle probability

OBSTACLE\_PROBABILITY = 0.3  # Increased probability

# Define delivery locations

DELIVERY\_LOCATIONS = []

for \_ in range(5):

    x = random.randint(0, GRID\_SIZE - 1)

    y = random.randint(0, GRID\_SIZE - 1)

    DELIVERY\_LOCATIONS.append((x, y))

# Initialize pygame

pygame.init()

# Set up the window

window = pygame.display.set\_mode(WINDOW\_SIZE)

pygame.display.set\_caption("Autonomous Delivery Robot")

# Define the grid

grid = [[0 for \_ in range(GRID\_SIZE)] for \_ in range(GRID\_SIZE)]

# Generate obstacles and cars

for i in range(GRID\_SIZE):

    for j in range(GRID\_SIZE):

        if random.random() < OBSTACLE\_PROBABILITY:

            grid[i][j] = random.choice([2, 3, 4])  # Representing different types of obstacles

        elif random.random() < 0.05:  # Adjust this probability as needed

            grid[i][j] = 5  # Representing cars

# Ensure robot and delivery locations are not blocked

grid[0][0] = 0  # Robot initial location

for loc in DELIVERY\_LOCATIONS:

    grid[loc[0]][loc[1]] = 0

# Define the heuristic function

def heuristic(a, b):

    (x1, y1) = a

    (x2, y2) = b

    return abs(x1 - x2) + abs(y1 - y2)

# Define the A\* algorithm

def astar(start, goal):

    frontier = []

    heapq.heappush(frontier, (0, start))

    came\_from = {}

    cost\_so\_far = {}

    came\_from[start] = None

    cost\_so\_far[start] = 0

    while frontier:

        current = heapq.heappop(frontier)[1]

        if current == goal:

            path = []

            while current != start:

                path.append(current)

                current = came\_from[current]

            path.append(start)

            path.reverse()

            return path

        for next in [(current[0] + 1, current[1]), (current[0] - 1, current[1]),

                     (current[0], current[1] + 1), (current[0], current[1] - 1)]:

            if 0 <= next[0] < GRID\_SIZE and 0 <= next[1] < GRID\_SIZE and grid[next[0]][next[1]] == 0:

                new\_cost = cost\_so\_far[current] + 1

                if next not in cost\_so\_far or new\_cost < cost\_so\_far[next]:

                    cost\_so\_far[next] = new\_cost

                    priority = new\_cost + heuristic(next, goal)

                    heapq.heappush(frontier, (priority, next))

                    came\_from[next] = current

    return []

def main():

    global DELIVERY\_LOCATIONS

    running = True

    path = []

    start = (0, 0)

    while running:

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                running = False

        # Clear the window

        window.fill(WHITE)

        # Draw the grid

        for i in range(GRID\_SIZE):

            for j in range(GRID\_SIZE):

                rect = pygame.Rect(j \* CELL\_SIZE, i \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE)

                if grid[i][j] == 1:  # Normal grid

                    pygame.draw.rect(window, WHITE, rect, 1)

                elif grid[i][j] == 2:  # Trees

                    pygame.draw.rect(window, GREEN, rect)

                elif grid[i][j] == 3:  # Buildings

                    pygame.draw.rect(window, GRAY, rect)

                elif grid[i][j] == 4:  # Houses

                    pygame.draw.rect(window, BROWN, rect)

                elif grid[i][j] == 5:  # Cars

                    pygame.draw.rect(window, YELLOW, rect)

        # Draw the delivery locations

        for loc in DELIVERY\_LOCATIONS:

            pygame.draw.circle(window, BLUE, (loc[1] \* CELL\_SIZE + CELL\_SIZE // 2, loc[0] \* CELL\_SIZE + CELL\_SIZE // 2), CELL\_SIZE // 4)

        # Draw the robot

        pygame.draw.circle(window, RED, (start[1] \* CELL\_SIZE + CELL\_SIZE // 2, start[0] \* CELL\_SIZE + CELL\_SIZE // 2), CELL\_SIZE // 3)

        # Draw the path

        if path:

            for i in range(len(path) - 1):

                pygame.draw.line(window, BLUE, (path[i][1] \* CELL\_SIZE + CELL\_SIZE // 2, path[i][0] \* CELL\_SIZE + CELL\_SIZE // 2),

                                 (path[i + 1][1] \* CELL\_SIZE + CELL\_SIZE // 2, path[i + 1][0] \* CELL\_SIZE + CELL\_SIZE // 2), 3)

        # Update the display

        pygame.display.flip()

        # Deliver items

        if DELIVERY\_LOCATIONS:

            current\_goal = min(DELIVERY\_LOCATIONS, key=lambda loc: heuristic(start, loc))

            path = astar(start, current\_goal)

            if path:

                while path:

                    next\_step = path.pop(0)

                    pygame.draw.line(window, BLUE, (start[1] \* CELL\_SIZE + CELL\_SIZE // 2, start[0] \* CELL\_SIZE + CELL\_SIZE // 2),

                                     (next\_step[1] \* CELL\_SIZE + CELL\_SIZE // 2, next\_step[0] \* CELL\_SIZE + CELL\_SIZE // 2), 3)

                    start = next\_step

                    pygame.draw.circle(window, RED, (start[1] \* CELL\_SIZE + CELL\_SIZE // 2, start[0] \* CELL\_SIZE + CELL\_SIZE // 2), CELL\_SIZE // 3)

                    pygame.display.flip()

                    time.sleep(0.3)

                DELIVERY\_LOCATIONS.remove(current\_goal)

        else:

            # Return to initial location

            path\_back = astar(start, (0, 0))

            if path\_back:

                while path\_back:

                    next\_step = path\_back.pop(0)

                    pygame.draw.line(window, BLUE, (start[1] \* CELL\_SIZE + CELL\_SIZE // 2, start[0] \* CELL\_SIZE + CELL\_SIZE // 2),

                                     (next\_step[1] \* CELL\_SIZE + CELL\_SIZE // 2, next\_step[0] \* CELL\_SIZE + CELL\_SIZE // 2), 3)

                    start = next\_step

                    pygame.draw.circle(window, RED, (start[1] \* CELL\_SIZE + CELL\_SIZE // 2, start[0] \* CELL\_SIZE + CELL\_SIZE // 2), CELL\_SIZE // 3)

                    pygame.display.flip()

                    time.sleep(0.3)

                    if not path\_back:

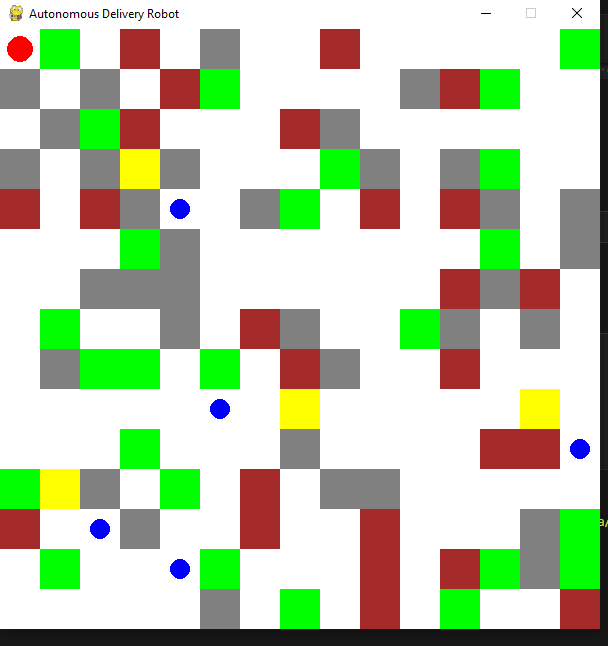
                        DELIVERY\_LOCATIONS.clear()  # All deliveries completed, clear the delivery locations list

            running = False  # Stop the loop once all deliveries are completed

    pygame.quit()

if \_\_name\_\_ == "\_\_main\_\_":

    main()



# QUESTION-2:

import tkinter as tk

from tkinter import ttk, messagebox, simpledialog

import time

class SudokuCSP:

    def \_\_init\_\_(self, board):

        if not all(len(row) == 9 for row in board) or len(board) != 9:

            raise ValueError("Board must be a 9x9 grid.")

        self.board = [row[:] for row in board]

        self.variables = [(i, j) for i in range(9) for j in range(9)]  # All cells are variables

        self.domains = {(i, j): list(range(1, 10)) if board[i][j] == 0 else [board[i][j]]

                        for i in range(9) for j in range(9)}

        self.neighbors = {v: self.find\_neighbors(v) for v in self.variables}

    def find\_neighbors(self, var):

        r, c = var

        neighbors = [(r, cc) for cc in range(9) if cc != c] + \

                    [(rr, c) for rr in range(9) if rr != r] + \

                    [(rr, cc) for rr in range((r // 3) \* 3, (r // 3) \* 3 + 3) for cc in

                     range((c // 3) \* 3, (c // 3) \* 3 + 3)

                     if (rr, cc) != var]

        return set(neighbors)

    def solve\_with\_backtracking(self):

        start\_time = time.time()

        if self.backtrack():

            end\_time = time.time()

            self.time\_complexity = end\_time - start\_time

            return True

        return False

    def backtrack(self):

        unassigned = self.find\_unassigned\_location()

        if not unassigned:

            return True  # Puzzle solved

        row, col = unassigned

        for num in range(1, 10):  # Try all possible numbers

            if self.is\_valid\_assignment(row, col, num):

                self.board[row][col] = num

                if self.backtrack():

                    return True

                self.board[row][col] = 0  # Undo assignment (backtrack)

        return False  # Trigger backtracking

    def solve\_with\_ac3(self):

        start\_time = time.time()

        if self.ac3():

            # If AC-3 prunes the domains successfully, try to solve the puzzle with backtracking

            if self.solve\_with\_backtracking():

                end\_time = time.time()

                self.time\_complexity = end\_time - start\_time

                return True

        else:

            # If AC-3 fails to prune the domains, return False indicating no solution

            return False

    def ac3(self):

        queue = [(xi, xj) for xi in self.variables for xj in self.neighbors[xi]]

        while queue:

            xi, xj = queue.pop(0)

            if self.revise(xi, xj):

                if not self.domains[xi]:

                    return False

                for xk in self.neighbors[xi]:

                    if xk != xj:

                        queue.append((xk, xi))

        return True

    def revise(self, xi, xj):

        revised = False

        domain\_xi = self.domains.get(xi, [])

        domain\_xj = self.domains.get(xj, [])

        for x in domain\_xi[:]:

            if all(x != y for y in domain\_xj) and self.check\_sudoku\_constraints(xi, x):

                self.domains[xi].remove(x)

                revised = True

        return revised

    def check\_sudoku\_constraints(self, var, value):

        row, col = var

        if any(value == self.board[row][j] for j in range(9)):

            return False

        if any(value == self.board[i][col] for i in range(9)):

            return False

        start\_row, start\_col = 3 \* (row // 3), 3 \* (col // 3)

        if any(value == self.board[i][j] for i in range(start\_row, start\_row + 3)

                                         for j in range(start\_col, start\_col + 3)):

            return False

        return True

    def find\_unassigned\_location(self):

        for i in range(9):

            for j in range(9):

                if self.board[i][j] == 0:

                    return i, j

        return None

    def is\_valid\_assignment(self, row, col, num):

        for k in range(9):

            if self.board[row][k] == num or self.board[k][col] == num:

                return False

        box\_start\_row, box\_start\_col = 3 \* (row // 3), 3 \* (col // 3)

        for i in range(box\_start\_row, box\_start\_row + 3):

            for j in range(box\_start\_col, box\_start\_col + 3):

                if self.board[i][j] == num:

                    return False

        return True

class SudokuGUI:

    def \_\_init\_\_(self, master):

        self.master = master

        master.title("Sudoku Solver")

        self.cells = {}

        # Initialize entry cells for the Sudoku grid

        for row in range(9):

            for col in range(9):

                entry = tk.Entry(master, width=3, font=('Arial', 18), borderwidth=2, justify='center')

                entry.grid(row=row, column=col, sticky='nsew', padx=1, pady=1)

                self.cells[(row, col)] = entry

        # Initialize control frame and other components after all entries are created

        self.initialize\_controls()

        self.puzzles = {

            'Easy': [

                # Puzzle #1

                [

                    [0, 7, 0, 3, 5, 0, 8, 0, 0],

                    [0, 3, 8, 7, 1, 4, 0, 6, 9],

                    [6, 4, 5, 0, 0, 0, 7, 1, 3],

                    [5, 8, 0, 1, 0, 0, 4, 0, 0],

                    [0, 0, 2, 0, 0, 9, 3, 0, 7],

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

                    [9, 5, 0, 2, 4, 1, 0, 0, 0],

                    [0, 6, 0, 8, 9, 5, 1, 0, 2],

                    [8, 2, 1, 6, 3, 7, 0, 0, 5]

                ],

                # Puzzle #2

                [

                    [0, 7, 1, 0, 8, 0, 0, 3, 0],

                    [3, 0, 9, 6, 5, 7, 0, 1, 0],

                    [0, 0, 2, 0, 1, 9, 6, 8, 0],

                    [0, 6, 0, 2, 3, 5, 8, 0, 1],

                    [0, 2, 3, 0, 9, 0, 7, 5, 0],

                    [1, 0, 8, 0, 0, 6, 3, 9, 2],

                    [8, 1, 6, 0, 0, 3, 0, 2, 5],

                    [7, 3, 5, 0, 0, 4, 9, 0, 0],

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

                ],

                # Puzzle #3

                [

                    [7, 6, 1, 9, 3, 4, 0, 0, 2],

                    [5, 0, 0, 8, 2, 1, 0, 6, 0],

                    [0, 0, 2, 6, 7, 0, 0, 1, 4],

                    [0, 1, 0, 3, 0, 6, 0, 0, 0],

                    [9, 3, 0, 0, 0, 7, 0, 8, 5],

                    [0, 5, 7, 2, 8, 9, 0, 3, 0],

                    [0, 2, 9, 5, 0, 8, 4, 0, 0],

                    [0, 0, 5, 7, 9, 3, 0, 2, 6],

                    [6, 7, 3, 4, 0, 2, 0, 9, 0]

                ],

                # Puzzle #4

                [

                    [2, 7, 3, 0, 0, 0, 0, 8, 5],

                    [0, 0, 1, 8, 0, 0, 0, 7, 0],

                    [5, 0, 0, 0, 0, 0, 0, 0, 1],

                    [0, 0, 0, 0, 8, 9, 0, 4, 0],

                    [0, 0, 8, 0, 6, 5, 0, 3, 7],

                    [4, 0, 7, 0, 0, 2, 8, 5, 0],

                    [3, 5, 0, 1, 7, 0, 0, 2, 4],

                    [0, 0, 0, 0, 0, 0, 7, 1, 0],

                    [7, 0, 0, 9, 0, 3, 0, 6, 8]

                ]

            ],

             # Medium Level Puzzles...

            'Medium': [

                      # Puzzle #1

                [

                    [1, 0, 0, 0, 0, 0, 3, 5, 6],

                    [9, 0, 0, 4, 3, 5, 1, 2, 0],

                    [0, 0, 0, 1, 2, 0, 0, 0, 4],

                    [8, 2, 7, 3, 6, 0, 0, 0, 1],

                    [0, 0, 1, 0, 5, 0, 0, 6, 0],

                    [0, 3, 6, 0, 1, 9, 8, 4, 0],

                    [0, 5, 4, 2, 9, 3, 6, 0, 8],

                    [0, 0, 9, 6, 8, 7, 4, 3, 0],

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

                ],

                      # Puzzle #2

                [

                    [3, 5, 6, 0, 8, 0, 9, 0, 1],

                    [2, 0, 0, 1, 0, 3, 7, 5, 6],

                    [9, 1, 7, 0, 0, 0, 0, 0, 8],

                    [4, 8, 0, 6, 0, 0, 3, 0, 0],

                    [0, 0, 0, 8, 3, 0, 1, 0, 0],

                    [0, 3, 0, 0, 0, 2, 6, 8, 5],

                    [5, 6, 4, 0, 0, 9, 8, 1, 0],

                    [0, 0, 0, 0, 1, 0, 0, 0, 0],

                    [0, 0, 0, 4, 0, 0, 0, 0, 7]

                ],

                      # Puzzle #3

                [

                    [9, 6, 2, 0, 4, 0, 0, 7, 0],

                    [7, 0, 0, 1, 0, 0, 0, 2, 0],

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

                    [0, 0, 3, 7, 8, 4, 0, 1, 0],

                    [8, 1, 0, 0, 0, 6, 0, 0, 7],

                    [0, 0, 7, 0, 0, 5, 0, 8, 2],

                    [0, 0, 0, 9, 0, 0, 0, 4, 0],

                    [4, 0, 0, 0, 0, 0, 0, 0, 5],

                    [0, 7, 5, 0, 0, 0, 0, 0, 0]

                ],      # Puzzle #4

                [

                    [5, 0, 0, 0, 0, 0, 7, 8, 0],

                    [0, 0, 0, 8, 0, 0, 0, 1, 0],

                    [0, 3, 7, 0, 1, 0, 9, 0, 0],

                    [0, 0, 0, 1, 0, 0, 6, 0, 8],

                    [9, 8, 0, 3, 2, 6, 0, 0, 0],

                    [0, 0, 3, 0, 0, 0, 0, 9, 0],

                    [3, 0, 0, 0, 6, 4, 0, 5, 9],

                    [0, 7, 9, 2, 5, 0, 3, 4, 0],

                    [6, 0, 4, 9, 3, 0, 0, 2, 0]

                ]

            ],

            'Hard': [

                      # Puzzle #1

                [

                    [7, 0, 0, 0, 0, 5, 0, 0, 0],

                    [0, 0, 0, 1, 0, 4, 6, 5, 0],

                    [0, 0, 0, 0, 6, 0, 3, 0, 1],

                    [3, 0, 6, 4, 0, 0, 0, 0, 0],

                    [0, 0, 4, 8, 0, 0, 1, 0, 9],

                    [9, 0, 0, 0, 7, 0, 4, 6, 0],

                    [0, 0, 0, 0, 0, 3, 0, 0, 2],

                    [5, 0, 0, 0, 0, 0, 9, 0, 0],

                    [1, 0, 8, 0, 4, 0, 5, 0, 0]

                ],

                      # Puzzle #2

                [

                    [0, 0, 9, 0, 0, 0, 3, 0, 0],

                    [0, 0, 1, 0, 0, 4, 9, 6, 0],

                    [0, 6, 0, 0, 8, 0, 0, 0, 4],

                    [8, 0, 6, 4, 5, 3, 0, 9, 0],

                    [0, 0, 0, 0, 0, 0, 0, 8, 0],

                    [5, 0, 0, 0, 6, 0, 0, 3, 0],

                    [6, 0, 2, 0, 4, 0, 0, 0, 0],

                    [0, 1, 3, 5, 0, 0, 0, 0, 0],

                    [0, 0, 0, 0, 2, 0, 1, 0, 0]

                ],

                      # Puzzle #3

                [

                    [8, 0, 0, 0, 0, 0, 0, 0, 3],

                    [0, 0, 3, 0, 0, 4, 7, 0, 0],

                    [0, 0, 2, 0, 0, 0, 0, 0, 0],

                    [0, 0, 1, 5, 8, 0, 9, 7, 0],

                    [3, 0, 0, 0, 0, 0, 8, 5, 1],

                    [0, 0, 5, 0, 0, 0, 0, 0, 4],

                    [9, 0, 0, 0, 0, 7, 0, 0, 0],

                    [0, 0, 0, 4, 0, 6, 2, 0, 0],

                    [0, 0, 0, 8, 0, 0, 6, 4, 9]

                ],

                      # Puzzle #4

                [

                    [5, 7, 9, 0, 0, 2, 8, 0, 0],

                    [2, 0, 0, 8, 0, 0, 7, 0, 9],

                    [0, 0, 8, 0, 0, 0, 2, 6, 0],

                    [0, 0, 7, 0, 0, 0, 0, 0, 6],

                    [0, 5, 4, 0, 9, 6, 0, 0, 0],

                    [0, 1, 0, 0, 7, 0, 0, 0, 0],

                    [0, 6, 0, 7, 2, 0, 4, 0, 0],

                    [0, 0, 3, 6, 0, 0, 0, 0, 7],

                    [0, 2, 0, 4, 0, 0, 6, 0, 0]

                ]

            ]

        }

        self.current\_puzzle = 0  # Start with the first puzzle by default

        self.load\_puzzle(self.current\_puzzle)

        self.update\_gui\_from\_board()

    def initialize\_controls(self):

        # Set up controls frame and other GUI elements

        self.controls\_frame = tk.Frame(self.master)

        self.controls\_frame.grid(row=0, column=9, rowspan=9, sticky='ns')

        self.level\_var = tk.StringVar(value='Easy')

        ttk.Label(self.controls\_frame, text="Level").grid(row=0, column=0, padx=5, pady=5)

        self.level\_menu = ttk.Combobox(self.controls\_frame, textvariable=self.level\_var, values=['Easy', 'Medium', 'Hard'])

        self.level\_menu.grid(row=1, column=0, padx=5, pady=5)

        self.algorithm\_var = tk.StringVar(value='Backtracking')

        ttk.Label(self.controls\_frame, text="Algorithms").grid(row=2, column=0, padx=5, pady=5)

        ttk.Radiobutton(self.controls\_frame, text='Arc Consistency-3', variable=self.algorithm\_var, value='AC-3').grid(row=3, column=0, padx=5, pady=5)

        ttk.Radiobutton(self.controls\_frame, text='Backtracking', variable=self.algorithm\_var, value='Backtracking').grid(row=4, column=0, padx=5, pady=5)

        self.puzzle\_var = tk.IntVar(value=1)

        ttk.Label(self.controls\_frame, text="Choose Puzzle").grid(row=5, column=0, padx=5, pady=5)

        for i in range(4):  # Assuming there are 4 puzzles

            ttk.Radiobutton(self.controls\_frame, text=f'Puzzle {i+1}', variable=self.puzzle\_var, value=i, command=self.change\_puzzle).grid(row=6+i, column=0, padx=5, pady=5)

        ttk.Button(self.controls\_frame, text="Solve", command=self.solve).grid(row=10, column=0, padx=5, pady=5)

        ttk.Button(self.controls\_frame, text="Reset", command=self.reset).grid(row=11, column=0, padx=5, pady=5)

        self.status = ttk.Label(self.master, text="Load a puzzle and select a solving method.", font=('Arial', 14))

        self.status.grid(row=10, column=0, columnspan=9)

    def change\_puzzle(self):

        """ Change the currently loaded puzzle based on user selection """

        index = self.puzzle\_var.get()

        self.current\_puzzle = index

        self.load\_puzzle(self.current\_puzzle)

        self.update\_gui\_from\_board()

    def load\_puzzle(self, index):

        """Load a puzzle based on the selected level."""

        level = self.level\_var.get()

        self.board = self.puzzles[level][index]

        self.csp = SudokuCSP(self.board)

    def solve(self):

        """Solve the currently loaded puzzle using the selected algorithm."""

        solution\_found = False

        if self.algorithm\_var.get() == 'AC-3':

            if self.csp.solve\_with\_ac3():

                self.status['text'] = f"Puzzle solved using AC-3! Time complexity: {self.csp.time\_complexity:.6f} seconds."

                solution\_found = True  # Set solution\_found to True here

            else:

                self.status['text'] = "AC-3 could not completely solve the puzzle."

        elif self.algorithm\_var.get() == 'Backtracking':

            if self.csp.solve\_with\_backtracking():

                self.status['text'] = f"Puzzle solved using Backtracking! Time complexity: {self.csp.time\_complexity:.6f} seconds."

                solution\_found = True

            else:

                self.status['text'] = "No solution exists using Backtracking."

        if solution\_found:

            self.board = [row[:] for row in self.csp.board]  # Copy the solved board to the GUI

            self.update\_gui\_from\_board()  # Update GUI only if a solution was found

    def reset(self):

        """Reset the board to the initial state of the current puzzle."""

        self.load\_puzzle(self.current\_puzzle)

        self.update\_gui\_from\_board()

        self.status['text'] = "Puzzle reset to initial state."

    def update\_gui\_from\_board(self):

        # Update the GUI to reflect the current state of the board.

        for row in range(9):

            for col in range(9):

                self.cells[(row, col)].delete(0, tk.END)

                if self.board[row][col] != 0:

                    self.cells[(row, col)].insert(0, str(self.board[row][col]))

def main():

    root = tk.Tk()

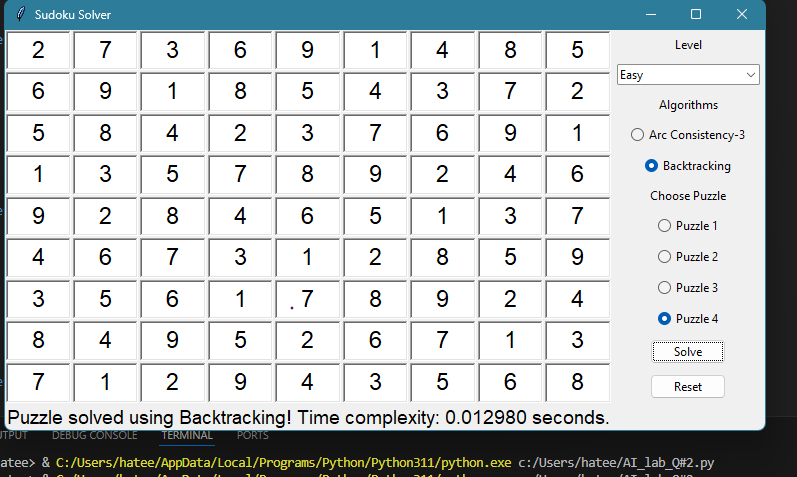
    gui = SudokuGUI(root)

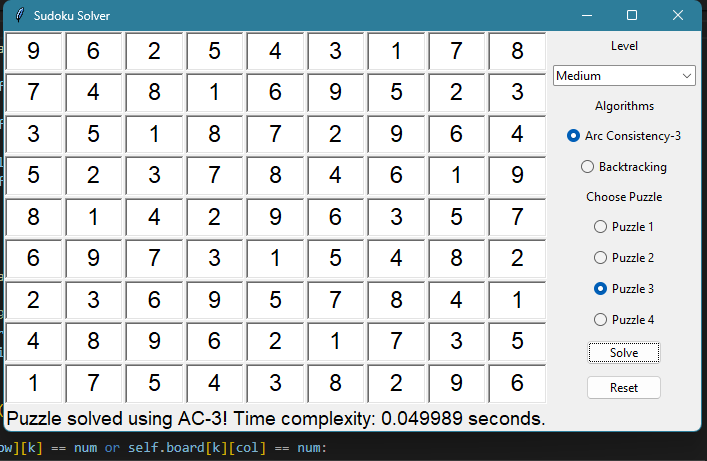
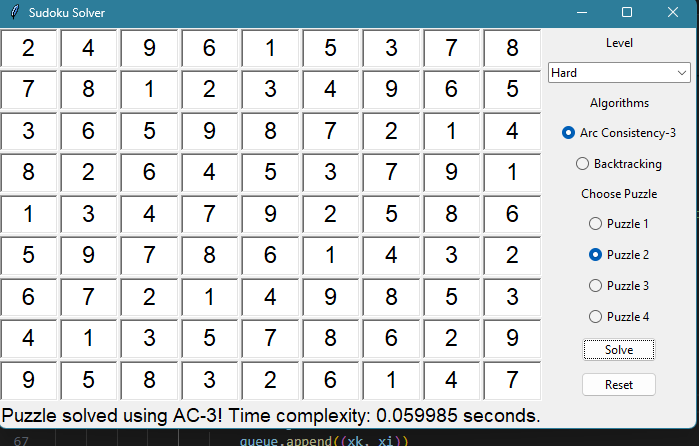
    root.mainloop()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

OUTPUT:



A screenshot of a math game

Description automatically generated

# QUESTION-3

import numpy as np

import pandas as pd

from sklearn.cluster import KMeans

import tkinter as tk

from tkinter import ttk, scrolledtext, filedialog

# Function to collect room capacities

def collect\_room\_capacities():

    # Simulated room capacities

    room\_capacities = [30] \* 25 + [35] \* 4 + [25]

    return room\_capacities

# Room Assignment Heuristic

def assign\_clusters\_to\_rooms(df, capacities):

    cluster\_counts = df['Cluster'].value\_counts().sort\_index()

    room\_assignments = {}

    room\_id = 0

    for cluster, count in cluster\_counts.items():

        while count > 0:

            available\_capacity = capacities[room\_id]

            room\_assignments.setdefault(room\_id, []).append(cluster)

            count -= available\_capacity

            room\_id = (room\_id + 1) % len(capacities)

    return room\_assignments

# Faculty Assignment to Rooms

def match\_faculty\_to\_rooms(assignments, faculties):

    faculty\_assignments = {}

    for room, clusters in assignments.items():

        common\_domain = students\_df[students\_df['Cluster'].isin(clusters)]['Domain'].mode()

        if not common\_domain.empty:

            faculty = faculties[faculties['Expertise\_Domain'] == common\_domain.iloc[0]]

            if not faculty.empty:

                faculty\_assignments[room] = faculty.iloc[0]['Faculty\_ID']

            else:

                faculty\_assignments[room] = "No faculty available for Domain"

        else:

            faculty\_assignments[room] = "No common domain found"

    return faculty\_assignments

# Reporting Function

def generate\_report(room\_assignments, faculty\_room\_assignments):

    report = "Seating Plan and Faculty Assignments Report\n\n"

    report += "Room Assignments:\n"

    for room, clusters in room\_assignments.items():

        report += f"Room {room}: Clusters {clusters}\n"

    report += "\nFaculty Assignments:\n"

    for room, faculty in faculty\_room\_assignments.items():

        report += f"Room {room}: Faculty ID {faculty}\n"

    return report

# Save Report to File

def save\_report(report):

    file\_path = filedialog.asksaveasfilename(defaultextension=".txt", filetypes=[("Text files", ".txt"), ("All files", ".\*")])

    if file\_path:

        with open(file\_path, "w") as file:

            file.write(report)

# Tkinter Application Setup

root = tk.Tk()

root.title("Exam Management System")

# Simulating Student Data

data = {

    'Student\_ID': range(1, 2501),

    'Domain': np.random.choice([0, 1, 2, 3, 4], 2500),  # CS=0, AI=1, BA=2, SE=3, EE=4

    'Batch': np.random.choice([19, 20, 21, 22, 23], 2500)

}

students\_df = pd.DataFrame(data)

# K-Means Clustering

kmeans = KMeans(n\_clusters=30, random\_state=0)

students\_df['Cluster'] = kmeans.fit\_predict(students\_df[['Domain', 'Batch']])

# Simulating Faculty Data

faculty\_data = {

    'Faculty\_ID': range(1,31),

    'Expertise\_Domain': np.random.choice([0, 1, 2, 3, 4], 30)

}

faculty\_df = pd.DataFrame(faculty\_data)

# Room capacities

room\_capacities = collect\_room\_capacities()

# Assign clusters to rooms

room\_assignments = assign\_clusters\_to\_rooms(students\_df, room\_capacities)

# Faculty assignment to rooms

faculty\_room\_assignments = match\_faculty\_to\_rooms(room\_assignments, faculty\_df)

# Generate report

report = generate\_report(room\_assignments, faculty\_room\_assignments)

# Seating Plan and Faculty Assignment Display

report\_text = scrolledtext.ScrolledText(root, width=80, height=20)

report\_text.pack()

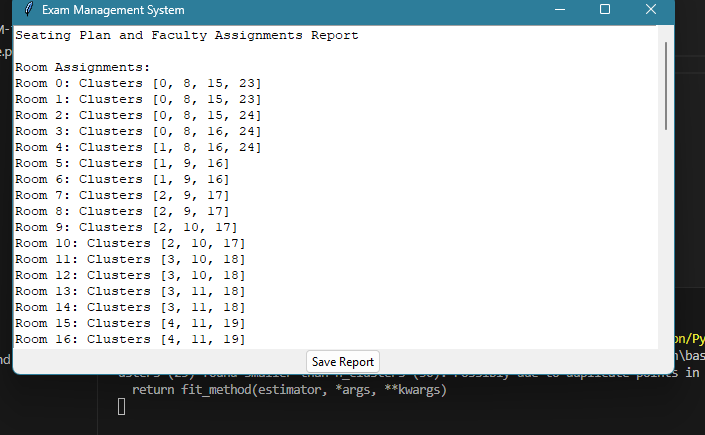
report\_text.insert(tk.INSERT, report)

# Button to Save Report

save\_button = ttk.Button(root, text="Save Report", command=lambda: save\_report(report))

save\_button.pack()

root.mainloop()



# QUESTION-4

import numpy as np

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

# Load Iris dataset

iris\_data = load\_iris()

X = iris\_data.data

y = iris\_data.target

# Divide the dataset into training and testing sets using a 70/30 ratio

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Standardize features

scaler = StandardScaler()

X\_train\_std = scaler.fit\_transform(X\_train)

X\_test\_std = scaler.transform(X\_test)

# Add bias term to input features

X\_train\_bias = np.c\_[np.ones((X\_train\_std.shape[0], 1)), X\_train\_std]

X\_test\_bias = np.c\_[np.ones((X\_test\_std.shape[0], 1)), X\_test\_std]

# Implement the Perceptron Learning Rule

class PerceptronLearningRule:

    def \_\_init\_\_(self, num\_features, learning\_rate=0.01):

        self.weights = np.zeros(num\_features + 1)

        self.learning\_rate = learning\_rate

    def \_activate(self, x):

        return np.where(x >= 0, 1, 0)

    def \_predict(self, x):

        return self.\_activate(np.dot(x, self.weights[1:]) + self.weights[0])

    def train(self, X, y, epochs=100):

        for \_ in range(epochs):

            for xi, yi in zip(X, y):

                prediction = self.\_predict(xi)

                error = yi - prediction

                self.weights[1:] += self.learning\_rate \* error \* xi

                self.weights[0] += self.learning\_rate \* error

    def evaluate(self, X, y):

        predictions = self.\_predict(X)

        accuracy = np.mean(predictions == y)

        return accuracy

# Implement sigmoid activation function for Gradient Descent Delta Rule

def sigmoid(z):

    return 1 / (1 + np.exp(-z))

class GradientDescentDeltaRule:

    def \_\_init\_\_(self, num\_features, learning\_rate=0.01):

        self.weights = np.zeros(num\_features + 1)

        self.learning\_rate = learning\_rate

    def \_predict(self, X):

        z = np.dot(X, self.weights[1:]) + self.weights[0]

        return sigmoid(z)

    def train(self, X, y, epochs=100):

        for \_ in range(epochs):

            predictions = self.\_predict(X)

            error = y - predictions

            gradient = np.dot(X.T, error) / len(X)

            self.weights[1:] += self.learning\_rate \* gradient

            self.weights[0] += self.learning\_rate \* np.mean(error)

    def evaluate(self, X, y):

        predictions = self.\_predict(X)

        predictions = np.where(predictions >= 0.5, 1, 0)

        accuracy = np.mean(predictions == y)

        return accuracy

# Adjust the learning rates of the models

learning\_rate\_perceptron = 0.1

learning\_rate\_gradient\_descent = 0.01

# Perceptron Learning Rule

plr = PerceptronLearningRule(X\_train\_bias.shape[1], learning\_rate\_perceptron)

plr.train(X\_train\_bias, (y\_train == 2).astype(int))

accuracy\_perceptron\_train = plr.evaluate(X\_train\_bias, (y\_train == 2).astype(int))

accuracy\_perceptron\_test = plr.evaluate(X\_test\_bias, (y\_test == 2).astype(int))

print("Perceptron Learning Rule:")

print("Accuracy (Train):", accuracy\_perceptron\_train)

print("Accuracy (Test):", accuracy\_perceptron\_test)

# Gradient Descent Delta Rule with sigmoid activation function

gddr = GradientDescentDeltaRule(X\_train\_bias.shape[1], learning\_rate\_gradient\_descent)

gddr.train(X\_train\_bias, (y\_train == 2).astype(int))

accuracy\_gradient\_descent\_train = gddr.evaluate(X\_train\_bias, (y\_train == 2).astype(int))

accuracy\_gradient\_descent\_test = gddr.evaluate(X\_test\_bias, (y\_test == 2).astype(int))

print("\nGradient Descent Delta Rule with Sigmoid Activation:")

print("Accuracy (Train):", accuracy\_gradient\_descent\_train)

print("Accuracy (Test):", accuracy\_gradient\_descent\_test)

