VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

ROSHNI P (1BM22CS223)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **ROSHNI P (1BM22CS223),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Sneha P Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

LAB 1: Tic –Tac –Toe Game

Algorithm:

27/09/24 Lab-1

Algorithm

1) At first import a random to generate a random number.
2) And then define to print board function using '—' to generate 3×3 matrix
3) Now check the rows, columns and diagonals
4) And check the empty spots to the next move
5) To check the winner we will check the rows, colums, diagonals. And it is range from 3.

```
if board[i][0] == board[i][1] == board[i][2] != " ":
        return [i][0]
if board[0][i] == board[1][i] == board[2][i] != " ":
        return[0][i]
if board[0][0] == board[1][1] == board[2][2] != " ":
        return[0][0]
if board[0][2] == board[1][1] == board[2][0] != " ":
        return[0][2].
```

6) we have check the board is full or not. if it full it will show board is full there is no space
7) And computer will see for empty cell it will randomly choose a cell for next move.
8) Now define function tic tac toe. it will range from 3 and current player is 'X' and a computer player is 'O'
9) If the current player is equal to the 'X' then give input for rows and columns or else for computer player give input for rows & columns.
10) If they give a same row & column again it will show cell is already taken.
11) Later it will shows the who is the winner.

Code:

```
def print_board(board):
    for row in board:
```

```
        print(" | ".join(row))
        print("-" * 9)

def check_winner(board):
    # Check rows, columns, and diagonals for a winner
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != " ":
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != " ":
            return board[0][i]

    if board[0][0] == board[1][1] == board[2][2] != " ":
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != " ":
        return board[0][2]

    return None

def is_full(board):
    return all(cell != " " for row in board for cell in row)

def tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X"

    while True:
        print_board(board)
        row = int(input(f"Player {current_player}, enter the row (0-2): "))
        col = int(input(f"Player {current_player}, enter the column (0-2): "))

        if board[row][col] == " ":
            board[row][col] = current_player
        else:
            print("Cell is already taken! Try again.")
            continue

        winner = check_winner(board)
        if winner:
            print_board(board)
            print(f"Player {winner} wins!")
            break

        if is_full(board):
            print_board(board)
```

```
            print("It's a tie!")
            break

        current_player = "O" if current_player == "X" else "X"

if _name___== "_main_":
    tic_tac_toe()
```

OUTPUT:

```
Player X goes first.
  |   |
---------
  |   |
---------
  |   |
---------
Player X, enter the row (0-2): 2
Player X, enter the column (0-2): 1
  |   |
---------
  |   |
---------
  | X |
---------
Computer's turn...
Computer chooses row 1, column 1
  |   |
---------
  | O |
---------
  | X |
---------
Player X, enter the row (0-2):        1
Player X, enter the column (0-2): 3
Invalid input! Please enter numbers between 0 and 2.
Player X, enter the row (0-2): 1
Player X, enter the column (0-2): 2
  |   |
---------
  | O | X
---------
  | X |
---------
Computer's turn...
Computer chooses row 0, column 0
O |   |
---------
  | O | X
---------
  | X |
---------
Player X, enter the row (0-2): 2
Player X, enter the column (0-2): 1
Cell is already taken! Try again.
Player X, enter the row (0-2): 2
Player X, enter the column (0-2): 0
O |   |
---------
  | O | X
---------
X | X |
---------
Computer's turn...
Computer chooses row 2, column 2
O |   |
---------
  | O | X
---------
X | X | O
---------
Player O wins!
```

LAB 2: Vacuum Cleaner Agent

Algorithm:

1/10/24 Lab-2.

Algorithm:

1) Take a class vacuum cleaner Agent
2) Define a function init with self parameter.
3) Now select a location room A and room B
assign a room A and B are dirty.
4) Define a function move-right and move-left.
set a location move-right to room B and
move-left to room A.
5) Define a function suck if a rooms are clean
vacuum cleaner will suck.
6) Define a ~~sense and~~ function sense & act. In this
we will print a in which vacuum is cleaning
and what is the room state ~~and which room va~~
7) Now, check the condition.
if self.rooms[self.location]==Dirty
print ("Action : suck").
elif self.location == 'A'
print("Action: Move Right").
elif self.location == 'B'
print(" Action: Move left").
8) Define a function run: and take a range ~~to~~
and print a steps and it will show that the
room is cleaned (or) it is dirty.

Graced

Changes for 4 room

```
Self.rooms = { 'A': 'Dirty', 'B': 'Dirty', 'C': 'Dirty', 'D': 'Di
Self.neighbours = {'A': {'right': 'B', 'down': 'C'},
                   'B': {'left': 'A', 'down': 'D'}
                   'C': {'up': 'A', 'right': 'D'},
                   'D': {'up': 'B', 'left': 'C'}}.

def run(self, steps=10):
    for step in range(steps):
        print(f"Step{step+1}:")
        Self.sense_and_act().
        print(f"Room States: {self.rooms}")
        print("-"*40).
```

Code:

For 2 rooms:

```
class VacuumCleanerAgent:
    def _init_(self):
        # Start in room A, assume both rooms are dirty initially
        self.location = 'A'9i
        self.rooms = {'A': 'Dirty', 'B': 'Dirty'}

    def move_right(self):
        self.location = 'B'

    def move_left(self):
        self.location = 'A'

    def suck(self):
        self.rooms[self.location] = 'Clean'

    def sense_and_act(self):
        print(f"Vacuum is in room {self.location}, Room state: {self.rooms[self.location]}")
```

```
        # Perceive environment and take action
        if self.rooms[self.location] == 'Dirty':
            print("Action: Suck")
            self.suck()
        elif self.location == 'A':
            print("Action: Move Right")
            self.move_right()
        elif self.location == 'B':
            print("Action: Move Left")
            self.move_left()

    def run(self, steps=5):
        for step in range(steps):
            print(f"Step {step + 1}:")
            self.sense_and_act()
            print(f"Room states: {self.rooms}")
            print("-" * 20)

# Initialize the vacuum cleaner agent and run it
vacuum_agent = VacuumCleanerAgent()
vacuum_agent.run()
```

OUTPUT:

```
Step 1:
Vacuum is in room A, Room state: Dirty
Action: Suck
Room states: {'A': 'Clean', 'B': 'Dirty'}
--------------------
Step 2:
Vacuum is in room A, Room state: Clean
Action: Move Right
Room states: {'A': 'Clean', 'B': 'Dirty'}
--------------------
Step 3:
Vacuum is in room B, Room state: Dirty
Action: Suck
Room states: {'A': 'Clean', 'B': 'Clean'}
--------------------
Step 4:
Vacuum is in room B, Room state: Clean
Action: Move Left
Room states: {'A': 'Clean', 'B': 'Clean'}
--------------------
Step 5:
Vacuum is in room A, Room state: Clean
Action: Move Right
Room states: {'A': 'Clean', 'B': 'Clean'}
--------------------
```

For 4 rooms:

```
def printArr(arr):
    for row in arr:
        print(row)
    print()

def clean(arr, x, y):
    if arr[x][y] == 1:
        arr[x][y] = 0

def check(arr):
    for row in arr:
        if 1 in row:
            return True
    return False

# Directions: right (0,1), down (1,0), left (0,-1), up (-1,0)
directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
direction_index = 0  # Start moving right

# Get room status
print("Enter the status of the rooms (0 for clean; 1 for dirty):")
arr1 = []
for i in range(2):
    row = []
    for j in range(2):
        a = int(input(f"Status of room ({i}, {j}): "))
        row.append(a)
    arr1.append(row)

x, y = 0, 0  #Start cleaning from the first room

while True:
    printArr(arr1)
    if not check(arr1):
        break
    clean(arr1, x, y)

    #Move to the next room in the current direction
    dx, dy = directions[direction_index]
    new_x, new_y = x + dx, y + dy

    #Check bounds
    if 0 <= new_x < 2 and 0 <= new_y < 2:
        x, y = new_x, new_y
    else:
        #Change direction(turn right)
        direction_index = (direction_index + 1) % 4
        dx, dy = directions[direction_index]
        x, y = x + dx, y + dy  #Move in the new direction

print("All rooms are cleaned!")
```

OUTPUT:

```
Enter the status of the rooms (0 for clean; 1 for dirty):
Status of room (0, 0): 1
Status of room (0, 1): 0
Status of room (1, 0): 1
Status of room (1, 1): 0
[1, 0]
[1, 0]

[0, 0]
[1, 0]

[0, 0]
[1, 0]

[0, 0]
[1, 0]

[0, 0]
[0, 0]

All rooms are cleaned!
```

LAB 3: Implement 8 puzzle problems

Algorithm:

8/10/24

DATE: PAGE:

Lab - 3.

Algorithm:

1) import double ended queue from the collections.
2) initialize a final goal state in form of a matrix.
3) Now, initialize a moves like up, down, left, right.
4) we are taking 0 for the empty cells.
5) Define a function called manhattan distance with parameter state.

```
distance = 0
for i in range(3):
    for j in range(3):
        if state[i][j] != 0
            goal_i, goal_j = divmod(state[i][j]-1, 3).
```

6) Define a goal_state. to return the state of the goal.
7) Define a get_neighbours :

```
neighbours = [ ]
for i in range(3):
    for j in range(3):
        if state[i][j] == 0:
            for move in moves:
                new_i, new_j = i + move[0], j +
                                move[1].
```

it will check the row in the state and it will append the neighbours. return neighbours.

```
neighbours.append(new_state)
return neighbours.
```

7) define a function dfs with parameter state.
take a two queue one for dequeue and another visited.

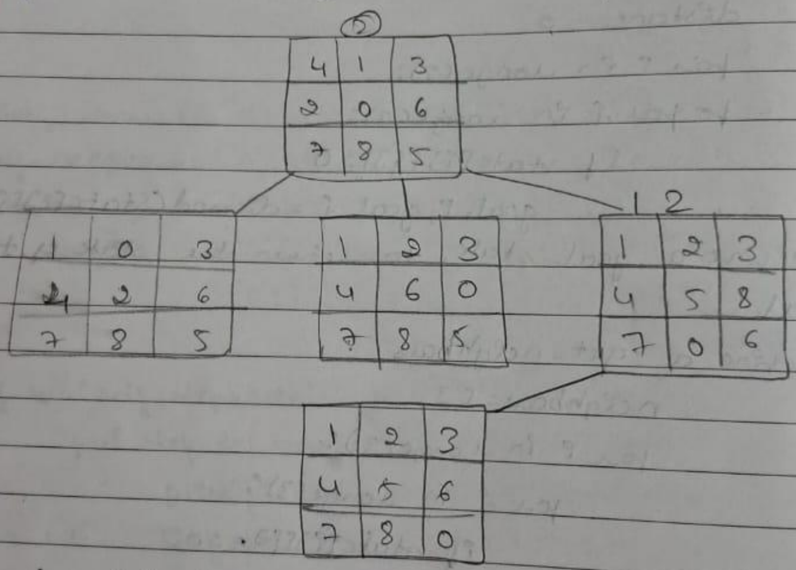
while queue():
 current_state, path = queue.popleft().

8) if goal state (current_state) return path.

9) ~~Take~~ using a map function we are converting tuple to state

10) Take a initial_state, assign a path = dfs (initial_state) to

11) check the path if path is in a state it will print the solution or else it won't print a solution.



i=2, j=1

new_i = i + move[0]
 = 2 + -1
 = 1

new_j = j + move[1]
 = 1 + 0
 = 1

Proceed

Code:

```
from collections import deque
GOAL_STATE = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]
MOVES = [
    (-1, 0),  # Up
    (1, 0),   # Down
    (0, -1),  # Left
    (0, 1)    # Right
]
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                goal_i, goal_j = divmod(state[i][j] - 1, 3)
                distance += abs(i - goal_i) + abs(j - goal_j)
    return distance
def is_goal_state(state):
    return state == GOAL_STATE
def get_neighbors(state):
    neighbors = []
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                for move in MOVES:
                    new_i, new_j = i + move[0], j + move[1]
                    if 0 <= new_i < 3 and 0 <= new_j < 3:
                        new_state = [row[:] for row in state]
                        new_state[i][j], new_state[new_i][new_j] = new_state[new_i][new_j],
new_state[i][j]
                        neighbors.append(new_state)
    return neighbors
def dfs(state):
    queue = deque([(state, [state])])
    visited = set()
    while queue:
        current_state, path = queue.popleft()
        if is_goal_state(current_state):
            return path
        if tuple(map(tuple, current_state)) in visited:
            continue
        visited.add(tuple(map(tuple, current_state)))
        for neighbor in get_neighbors(current_state):
            queue.append((neighbor, path + [neighbor]))
```

```
    return None
initial_state = [
    [4, 1, 3],
    [7, 2, 6],
    [5, 8, 0]
]
path = dfs(initial_state)
if path:
    print("Solution found in {len(path)} moves:")
    for state in path:
        for row in state:
            print(row)
        print()
else:
    print("No solution found.")
```

OUTPUT:

```
Solution found:
[1, 2, 3]
[4, 0, 5]
[7, 8, 6]

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

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

Total moves taken to reach the final state: 2
```

LAB 4: Iterative deepening search algorithm

Algorithm:

15/10/24.

Iterative Deepening Search:

1. Define a function called Iterative deepening search with a parameter graph, start node and a goal node
2. Define a function depth with a parameter node, goal and depth.

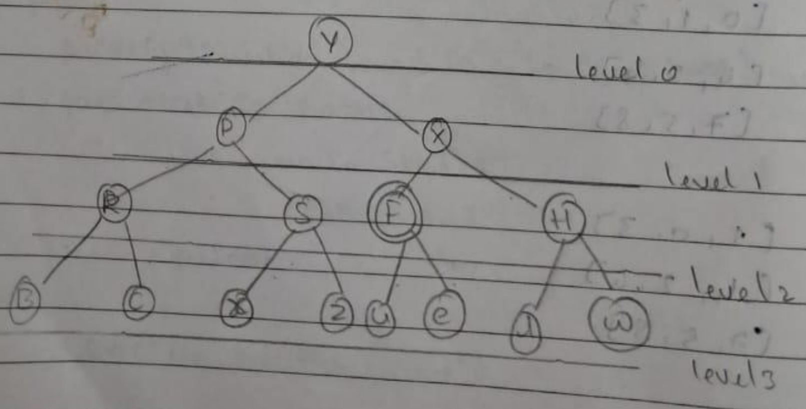
8. if depth = = 0:
    if node = = goal:
        return [node]
    else:
        return None.

or else it will go to the depth of the graph and it will check the child nodes.

3. while true:
    result = ids(start, goal, graph)
    if result is None.
        depth = depth + 1.

4. Define a function called user-input. take a user input for number of edges, and give a edges and give a start node and the goal node.

5. In the main function it will check the path is found or not.



Iteration	States
0	Y
1	Y -> P -> X
2	Y -> P -> X -> R -> S -> F.

8 Puzzle using A^*

1	2	3
8		4
7	6	5

initial state

2	8	1
	4	3
7	6	5

goal state

1. Define a heuristic function to calculate the counting number of misplaced state and target we are using a zip function to combine the state and the target ~~function~~.
2. Define a evaluation function state level contain the current state and we call a ~~functio~~ heuristic function to calculate then we are combining the h(n) and g(n)
3. then we will take a directions left, right, up and down.
4. Display the state in 3X3 grid format.
   it will range from 1 to 9 take a empty cells as 0.
5. Define a function A^* with parameter state, target.

arr = [[src, 0]]
visited = []
iterations = 0.

DATE: PAGE:

6. we will check the conditions and iteration will be incremented.

7. It will display the current state where the target is there.

8. Append a ~~to~~ visited states to the list and give a initial state and goal state. and call a function.

Code:

```
def iterative_deepening_search(graph, start, goal):
    def depth_limited_search(node, goal, depth):
        if depth == 0:
            if node == goal:
                return [node]
            else:
                return None
        elif depth > 0:
            for child in graph.get(node, []):
                result = depth_limited_search(child, goal, depth - 1)
                if result is not None:
                    return [node] + result
        return None
    depth = 0
    while True:
        result = depth_limited_search(start, goal, depth)
        if result is not None:
            return result
        depth += 1
def get_user_input_graph():
    graph = {}
    num_edges = int(input("Enter the number of edges: "))
    print("Enter each edge in the format 'node1 node2':")
    for _ in range(num_edges):
        node1, node2 = input().split()
        if node1 in graph:
            graph[node1].append(node2)
        else:
```

```
            graph[node1] = [node2]
        if node2 in graph:
            graph[node2].append(node1)
        else:
            graph[node2] = [node1]
    return graph
def main():
    graph = get_user_input_graph()
    start_node = input("Enter the starting node: ")
    goal_node = input("Enter the goal node: ")
    path = iterative_deepening_search(graph, start_node, goal_node)
    if path:
        print(f"Path found: {' -> '.join(path)}")
    else:
        print("No path found")
if _name___== "_main_":
    main()
```

```
Enter the number of edges: 14
Enter each edge in the format 'node1 node2':
Y P
Y X
P R
P S
X F
X H
R B
R C
S X
S Z
F U
F E
H L
H W
Enter the starting node: Y
Enter the goal node: F
Path found: Y -> X -> F
```

PART 2: Implement A* search algorithm

Code:

```
def H_n(state, target):
    return sum(x != y for x, y in zip(state, target))

# Evaluation function F(n) = H(n) + G(n)
```

```
def F_n(state_with_lvl, target):
    state, lvl = state_with_lvl
    return H_n(state, target) + lvl

# Function to generate possible moves
def possible_moves(state_with_lvl, visited_states):
    state, lvl = state_with_lvl
    b = state.index(0)  # Find index of the empty spot (0)
    directions = []  # Possible move directions ('d': down, 'u': up, 'l': left, 'r': right)
    pos_moves = []

    # Determine which moves are possible
    if b <= 5: directions.append('d')
    if b >= 3: directions.append('u')
    if b % 3 > 0: directions.append('l')
    if b % 3 < 2: directions.append('r')

    # Generate new states for each possible move
    for move in directions:
        temp = gen(state, move, b)
        if temp not in visited_states:
            pos_moves.append([temp, lvl + 1])  # Add new state with incremented level

    return pos_moves

# Generate new state based on move direction
def gen(state, move, b):
    temp = state.copy()
    if move == 'l': temp[b], temp[b - 1] = temp[b - 1], temp[b]
    if move == 'r': temp[b], temp[b + 1] = temp[b + 1], temp[b]
    if move == 'u': temp[b], temp[b - 3] = temp[b - 3], temp[b]
    if move == 'd': temp[b], temp[b + 3] = temp[b + 3], temp[b]
    return temp

# Display the state in a 3x3 grid format
def display_state(state):
    print("Current State:")
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()  # New line for better readability

# A* search algorithm with step display
def astar(src, target):
    arr = [[src, 0]]  # State, level
```

```
    visited_states = []
    iterations = 0

    while arr:
        iterations += 1
        current = min(arr, key=lambda x: F_n(x, target))  # Select state with minimum F(n)
        arr.remove(current)

        # Display the current state
        display_state(current[0])

        # If target is found
        if current[0] == target:
            return f'Found with {iterations} iterations'

        # Mark current state as visited
        visited_states.append(current[0])
        # Add possible moves to queue
        arr.extend(possible_moves(current, visited_states))

    return 'Not found'

# Test the A* algorithm
src = [4, 1, 3, 7, 2, 6, 5, 8, 0]  # Using 0 for the empty space
target = [1, 2, 3, 4, 5, 6, 7, 8, 0]  # Target state
print(astar(src, target))
```

OUTPUT:

```
Current State:
[0, 1, 3]
[4, 2, 6]
[7, 5, 8]

Current State:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]

Current State:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Current State:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Current State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

LAB 5: Simulated Annealing

Algorithm:

22/10/24.

DATE: PAGE:

Algorithm.

1) First import math and import random.
2) Define a objective function(x) after that take a function for minimum $x=3$

   return. $(x-3)^2$.
3) Define a function simulated annealing with parameters objective_function, initial_solution, initial_temperature, cooling rate, stopping_temperature, max_iterations.
4) Set the current_solution to the initial_solution.

   current_solution = initial_solution

   current_value = objective_function(current_solution)

   track a best solution found so far.

   best_solution = current_solution

   best_value = current_value.

   temperature = initial_temperature.

   iteration = 0.

   continue until the temperature stops belows the stopping condition @ max iterations.

   while temperature > stopping_temperature and iterations < max iterations.
5) generate a new solution by randomly changing the current solution. this allows the exploration of solution space.
6) calculate the difference of objective function value to check how much better ~~the~~ @ worse the new solution.
7) Decide whether to accept the new solution. if the new solution is better we accept it, if the new solution is worse we accept the probability based on the temperature.

current_solution = new_solution
current_value = new_value.

8) Highest temperature allow worse solutions to accept more easily.

9) update a best solution found so far and cool down the temperature, if temp decreases the algorithm become less likely to accept.

10) temperature *= cooling_rate.

10) Initialize a values for initial_solution, initial_temperature, max iterations, cooling_rate, stopping_temperatures.

best_solution, best_value = objective_function(initial_solution, initial_temperature, max_iterations, cooling_rate, stopping_temperature).

11) print the temperature, iterations, best_solution, value.
at last write print best_solution & value after all the iterations.

Code:

```
import math
import random

# Define the objective function: our goal is to minimize this function
def objective_function(x):
    # The function we are minimizing is f(x) = (x - 3)^2, which has a minimum at x = 3
    return (x - 3) ** 2

# Simulated Annealing algorithm
def simulated_annealing(objective_function, initial_solution, initial_temperature,
cooling_rate, stopping_temperature, max_iterations):
    """
    Parameters:
    objective_function - The function we are trying to minimize
    initial_solution - Starting point for the algorithm
    initial_temperature - Initial temperature of the system
```

```
    cooling_rate - Rate at which the temperature decreases
    stopping_temperature - Minimum temperature at which the algorithm stops
    max_iterations - Maximum number of iterations to avoid infinite loops
    """

    # Step 1: Set the current solution to the initial solution
    current_solution = initial_solution
    current_value = objective_function(current_solution)

    # Track the best solution found so far
    best_solution = current_solution
    best_value = current_value

    # Initialize the temperature to the initial temperature
    temperature = initial_temperature

    # Initialize iteration counter
    iteration = 0

    # Main loop of the Simulated Annealing algorithm
    # Continue until the temperature drops below the stopping condition or max iterations is
reached
    while temperature > stopping_temperature and iteration < max_iterations:

        # Step 2: Generate a new solution by perturbing (randomly changing) the current
solution
        # This allows exploration of the solution space
        new_solution = current_solution + random.uniform(-1, 1)  # Take a small random step
from the current solution
        new_value = objective_function(new_solution)

        # Calculate the difference in objective function values (how much better or worse is the
new solution?)
        delta_value = new_value - current_value

        # Step 3: Decide whether to accept the new solution:
        # If the new solution is better, we always accept it
        if delta_value < 0:
            current_solution = new_solution
            current_value = new_value
        else:
            # If the new solution is worse, accept it with a probability based on the temperature
            # Higher temperatures allow worse solutions to be accepted more easily, facilitating
exploration
            probability = math.exp(-delta_value / temperature)
            if random.random() < probability:
                current_solution = new_solution
                current_value = new_value

        # Step 4: Update the best solution found so far
```

```
        if current_value < best_value:
            best_solution = current_solution
            best_value = current_value

        # Step 5: Cool down the temperature (gradually reduce temperature to "freeze" the
system)
        # As temperature decreases, the algorithm becomes less likely to accept worse solutions
        temperature *= cooling_rate

        # Increment the iteration counter
        iteration += 1

        # Print the current state of the algorithm for monitoring
        # This helps visualize how the temperature changes and how the algorithm progresses
        print(f"Iteration: {iteration}, Temperature: {temperature:.4f}, Current Solution:
{current_solution:.4f}, Best Solution: {best_solution:.4f}")

    # Return the best solution found and its value after the algorithm has finished
    return best_solution, best_value

# Parameters for the Simulated Annealing algorithm
initial_solution = 10  # Starting point for the algorithm
initial_temperature = 1000  # High starting temperature
cooling_rate = 0.95  # Rate at which the temperature is decreased (reduce by 5% each
iteration)
stopping_temperature = 1e-8  # Algorithm stops when the temperature is very low
max_iterations = 10 # Limit the number of iterations to prevent infinite loops

# Execute the Simulated Annealing algorithm
best_solution, best_value = simulated_annealing(objective_function, initial_solution,
initial_temperature, cooling_rate, stopping_temperature, max_iterations)

# Output the final result: best solution and value
print(f"Best solution found: x = {best_solution:.4f}, f(x) = {best_value:.4f}")
```

OUTPUT:

```
Best solution found: -0.7323104061658242
```

LAB 6: Implement Hill Climbing

Algorithm:

29/10/24.

A* algorithm for 8 queens

1) import heap queue
2) Create class Node, define a function init with a parameter state, g, h.
3) state: current configuration, g: cost from the start node to current node. (number of moves mades). h: heuristic estimate of the cost to reach the goal (no of attacks).
f: total estimated cost (g+h)
4) define a function less than operator for priority.
    def __lt__(self, other):
        return self.f < other.f.
5) define a function heuristic with parameter state
    attacks = 0.
    for i in range(len(state)):
        for j in range(i+1, len(state)):
            if state[i] == state[j] or abs(state[i] - state[j]) == j - i:
                attacks += 1
6) define a function A_star_8_queens
    initial_state = tuple([-1] * 8)
    open_set = []. push a openset, initial_set, 0, heuristic to the queue.
    visited = set()
it will pop the current_node from the priority_queue. if node visited it will do the further processing iterates through all columns for each new state calculate g and h.
7) Define a function display board.
    for i in range(8):
        line = " "
        for col in range(8):

if state[row] = = col.
line += "Q"
else line += ".".
print (line).
solution = a_star_8_queens().
print the display_board and it will print the solution if it found (or) else it will print solution not found.

<u>Hill climbing for 8 queens:</u>

1) import random.
2) Define a function calculate attacks.
attacks = 0.
for i in range(len(state)):
for j in range(i+1, len(state)):
if state[i] == state[j] or abs(state[i]-state[j]) == j-i:
attacks += 1.
3) Define a function hill_climbing
state = [random.randint(0,7) for _ in range(8)].
current_attacks = calculate_attacks(state)
for _ in range(10).
neighbours = [] take a range for rows and columns.
if state[row] != col:
neighbour = state[:].
neighbor[row] = col.
neighbor.append(neighbor).
calculate the new state & take a min of the neighbours.

if next_attacks >= current_attacks
break
State = next_state.
Current_attacks = new_attacks
Define a function display_board.
for i in range(8):
line = " "
for j in range(8):
· if state[row] == col
line += "Q"
else line += "."
print(line).
for in range(attempts)
solution = hill_climbing
It will check for the best solution and it will print the best solution.

Code:

```
import random

def calculate_attacks(state):
    attacks = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] == state[j] or abs(state[i] - state[j]) == j - i:
                attacks += 1
    return attacks

def hill_climbing_8_queens():
    state = [random.randint(0, 7) for _ in range(8)]
    current_attacks = calculate_attacks(state)

    for _ in range(100):  # Limit the number of iterations
        neighbors = []
        for row in range(8):
```

```
            for col in range(8):
                if state[row] != col:
                    neighbor = state[:]
                    neighbor[row] = col
                    neighbors.append(neighbor)

        next_state = min(neighbors, key=calculate_attacks)
        next_attacks = calculate_attacks(next_state)

        if next_attacks >= current_attacks:
            break

        state = next_state
        current_attacks = next_attacks

    return state, current_attacks

def display_board(state):
    for row in range(8):
        line = ""
        for col in range(8):
            if state[row] == col:
                line += "Q "
            else:
                line += ". "
        print(line)
    print()

# Run multiple attempts
best_solution = None
best_attacks = float('inf')
attempts = 10

for _ in range(attempts):
    solution, attacks = hill_climbing_8_queens()
    if attacks < best_attacks:
        best_solution = solution
        best_attacks = attacks

    if best_attacks == 0:
        break

if best_solution:
    print(f"Best solution found (with {best_attacks} attacking pairs):")
```

```
        display_board(best_solution)
    else:
        print("No solution found.")
```

OUTPUT:

```
Best solution found (with 1 attacking pairs):
. . . . . Q . .
. . . . . . . Q
. . Q . . . . .
. . . . . . Q .
. . . Q . . . .
. Q . . . . . .
. . . . Q . . .
Q . . . . . . .
```

PART 2: Implement A* search algorithm

Code:

```
import heapq

class Node:
    def __init__(self, state, g, h):
        self.state = state
        self.g = g
        self.h = h
        self.f = g + h

    def __lt__(self, other):
        return self.f < other.f

def heuristic(state):
    attacks = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] == state[j] or abs(state[i] - state[j]) == j - i:
                attacks += 1
    return attacks

def a_star_8_queens():
    initial_state = tuple([-1] * 8)
    open_set = []
    heapq.heappush(open_set, Node(initial_state, 0, heuristic(initial_state)))
    visited = set()

    while open_set:
        current_node = heapq.heappop(open_set)
        current_state = current_node.state

        if current_node.h == 0 and -1 not in current_state:
```

```
            return current_state

        if current_state in visited:
            continue
        visited.add(current_state)

        next_row = current_state.index(-1) if -1 in current_state else len(current_state)
        if next_row < 8:
            for col in range(8):
                new_state = list(current_state)
                new_state[next_row] = col
                new_state = tuple(new_state)
                if new_state not in visited:
                    g = current_node.g + 1
                    h = heuristic(new_state)
                    heapq.heappush(open_set, Node(new_state, g, h))

    return None

def display_board(state):
    for row in range(8):
        line = ""
        for col in range(8):
            if state[row] == col:
                line += "Q "
            else:
                line += ". "
        print(line)
    print()

solution = a_star_8_queens()
if solution:
    print("A* Solution:")
    display_board(solution)
else:
    print("No solution found.")
```

OUTPUT:

```
A* Solution:
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . Q . . . . .
. . . . . Q . .
```

LAB 7: Propositional Logic

Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

Algorithm:

12/11/24

Lab-7.

Knowledge Base:

1. Alice is the mother of Bob P
2. Bob is the father of charlie Q
3. A father is a parent R
4. A mother is a parent S
5. All parents have childern T
6. If someone is a parent, their children are siblings U
7. Alice is married to David. V

Hypotheses:

"charlie is a sibling of Bob".

Entailment Process:

From 1 and 2:

Alice is the mother of Bob, and Bob is the father of charlie.

From 3 and 4:

father and mother are considered as a parent.

From 5:

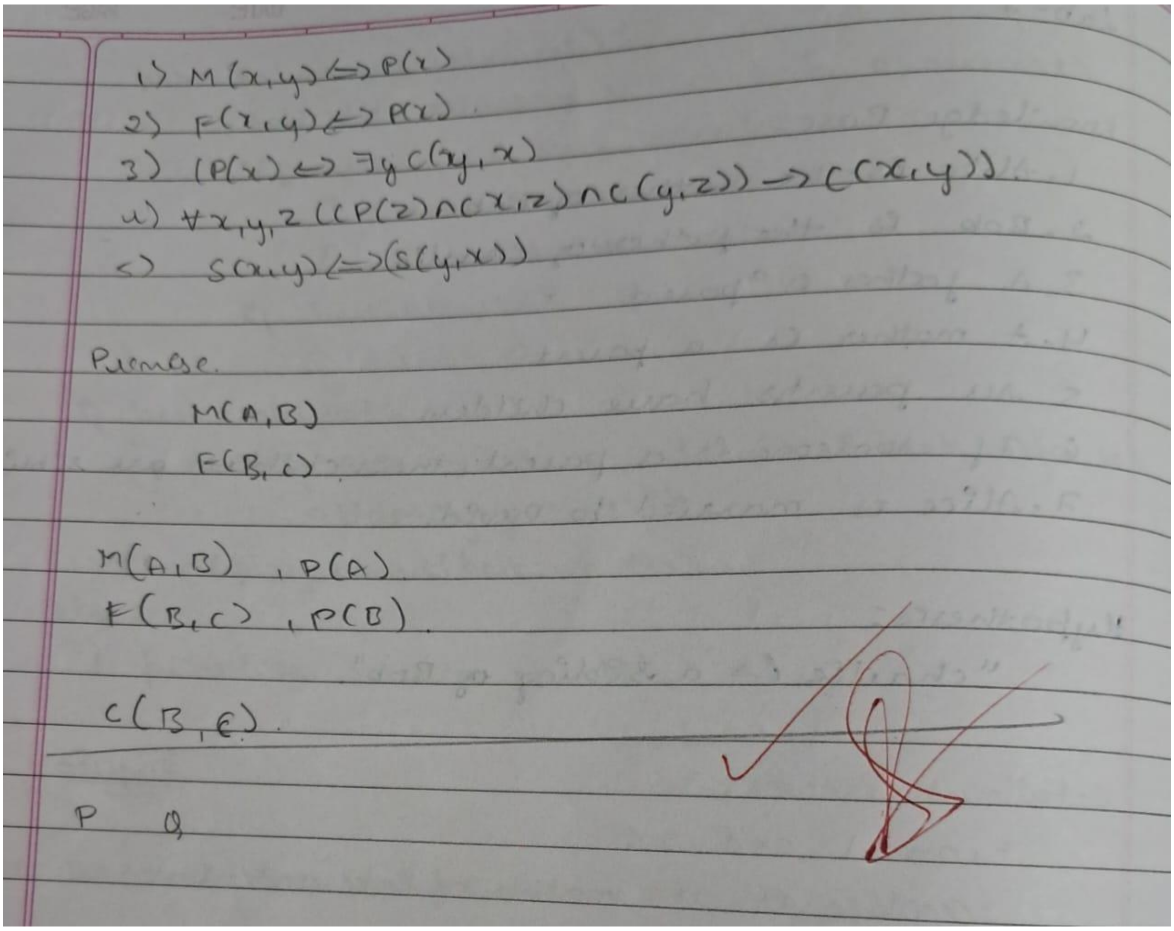
All the parents have a children

From 6 & 7:

Since bob is a parent of the charlie & potent Alice, due to her marriage to David.

Conclusion:

charlie can be a sibling of Bob because Bob is parent of charlie.



1) $M(x,y) \leftrightarrow P(x)$
2) $F(x,y) \leftrightarrow P(x)$
3) $(P(x) \leftrightarrow \exists y\, C(y,x)$
4) $\forall x,y,z\, ((P(z) \wedge C(x,z) \wedge C(y,z)) \rightarrow C(x,y))$
5) $S(x,y) \leftrightarrow (S(y,x))$

Premise

$M(A,B)$

$F(B,C)$

$M(A,B)$, $P(A)$

$F(B,C)$, $P(B)$

$C(B,C)$

P Q

Code:

```
class KnowledgeBase:
    def __init__(self):
        self.facts = []
        self.rules = []

    def add_fact(self, fact):
        if fact not in self.facts:
            self.facts.append(fact)

    def add_rule(self, rule):
        self.rules.append(rule)

    def infer(self):
        inferred_facts = []
        for rule in self.rules:
```

```
            new_facts = rule(self.facts)
            for fact in new_facts:
                if fact not in self.facts and fact not in inferred_facts:
                    inferred_facts.append(fact)
        self.facts.extend(inferred_facts)

    def check_hypothesis(self, query):
        self.infer()  # Apply all rules to infer new facts
        return query in self.facts



# Define the rules
def rule_parent_relationship(facts):
    """If someone is a father or mother, they are also a parent."""
    inferred_facts = []
    for fact in facts:
        if fact[0] == "father" or fact[0] == "mother":
            inferred_facts.append(("parent", fact[1], fact[2]))
    return inferred_facts



def rule_sibling_relationship(facts):
    """If two people share a parent, they are siblings."""
    inferred_facts = []
    for fact1 in facts:
        if fact1[0] == "parent":
            for fact2 in facts:
                if (
                    fact2[0] == "parent"
                    and fact1[1] == fact2[1]
                    and fact1[2] != fact2[2]
                ):
                    inferred_facts.append(("sibling", fact1[2], fact2[2]))
                    inferred_facts.append(("sibling", fact2[2], fact1[2]))  # Symmetry
    return inferred_facts
```

```
# Instantiate the knowledge base
kb = KnowledgeBase()

# Add facts to the knowledge base
kb.add_fact(("mother", "Alice", "Bob"))     # Alice is Bob's mother
kb.add_fact(("father", "Bob", "Charlie"))   # Bob is Charlie's father

# Add rules to the knowledge base
kb.add_rule(rule_parent_relationship)
kb.add_rule(rule_sibling_relationship)

# Check the hypothesis
hypothesis = ("sibling", "Charlie", "Bob")
kb.infer()  # Apply all rules and infer facts

if kb.check_hypothesis(hypothesis):
    print(f"The hypothesis '{hypothesis}' is TRUE.")
else:
    print(f"The hypothesis '{hypothesis}' is FALSE.")
```

OUTPUT:

```
PS C:\Users\pbs82\Downloads\AI> & C:/Users/pbs82/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/pbs82/Downloads/AI/aii.py
The hypothesis '('sibling', 'Charlie', 'Bob')' is TRUE.
```

LAB 8: Unification in first order logic

Algorithm:

Lab 8.

John is a human
Humans can speak English
John can speak English.

Problem:

1. All humans can speak English.
   $\forall x(H(x) \rightarrow S(x))$.
2. John is a human
   $H(John)$
3. John can speak English
   $S(John)$.

Proof.

1. $\forall x(H(x) \rightarrow S(x))$
2. $H(John)$.
3. $\forall x(H(x) \rightarrow S(x))$, we conclude $H(John) \rightarrow S(John)$.
4. Modus Ponens: From $H(John)$ and $H(John) \rightarrow S(John)$ we conclude $S(John)$.

Conclusion:

Proved that John can speak English.

Output:

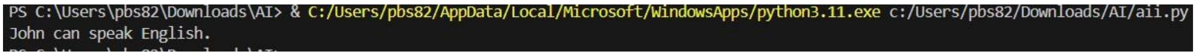
John can speak English.

Code:

```
# Logical system implementation to prove "John can speak English"

class LogicSystem:
    def __init__(self):
        self.knowledge_base = []

    def add_statement(self, statement):
        """Add a statement or rule to the knowledge base."""
        self.knowledge_base.append(statement)

    def infer(self, entity, predicate):
        """
        Infer whether a given predicate is true for a specific entity.
        Returns True if proven, False otherwise.
        """
        for rule in self.knowledge_base:
            if callable(rule):  # If it's a rule (function), try applying it
                if rule(entity, predicate):
                    return True
            elif rule == (entity, predicate):  # Direct match in the knowledge base
                return True
        return False


# Define predicates
def is_human(entity):
    """Returns True if the entity is human."""
    return entity == "John"  # John is human


def universal_rule(entity, predicate):
    """
    Implements the universal rule: Human(x) → CanSpeakEnglish(x).
    Returns True if the rule infers the predicate for the entity.
    """
    if predicate == "CanSpeakEnglish" and is_human(entity):
        return True
    return False


# Initialize the logical system
logic_system = LogicSystem()

# Add premises to the knowledge base
logic_system.add_statement(("John", "Human"))  # Premise 1: John is human
logic_system.add_statement(universal_rule)       # Premise 2: All humans can speak English

# Prove the statement: John can speak English
entity = "John"
predicate = "CanSpeakEnglish"

# Check inference
if logic_system.infer(entity, predicate):
```

```
    print(f"{entity} can speak English.")
else:
    print(f"{entity} cannot speak English.")
```

OUTPUT:



```
PS C:\Users\pbs82\Downloads\AI> & C:/Users/pbs82/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/pbs82/Downloads/AI/aii.py
John can speak English.
```

LAB 9: Forward Chaining

Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning

Algorithm:

DATE: PAGE:

3/12/24.

First order logic: Forward chaining.

Solve using forward chaining.

Problem: As per the law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robet, who is an American citizen.

Prove that "Robert is criminal"

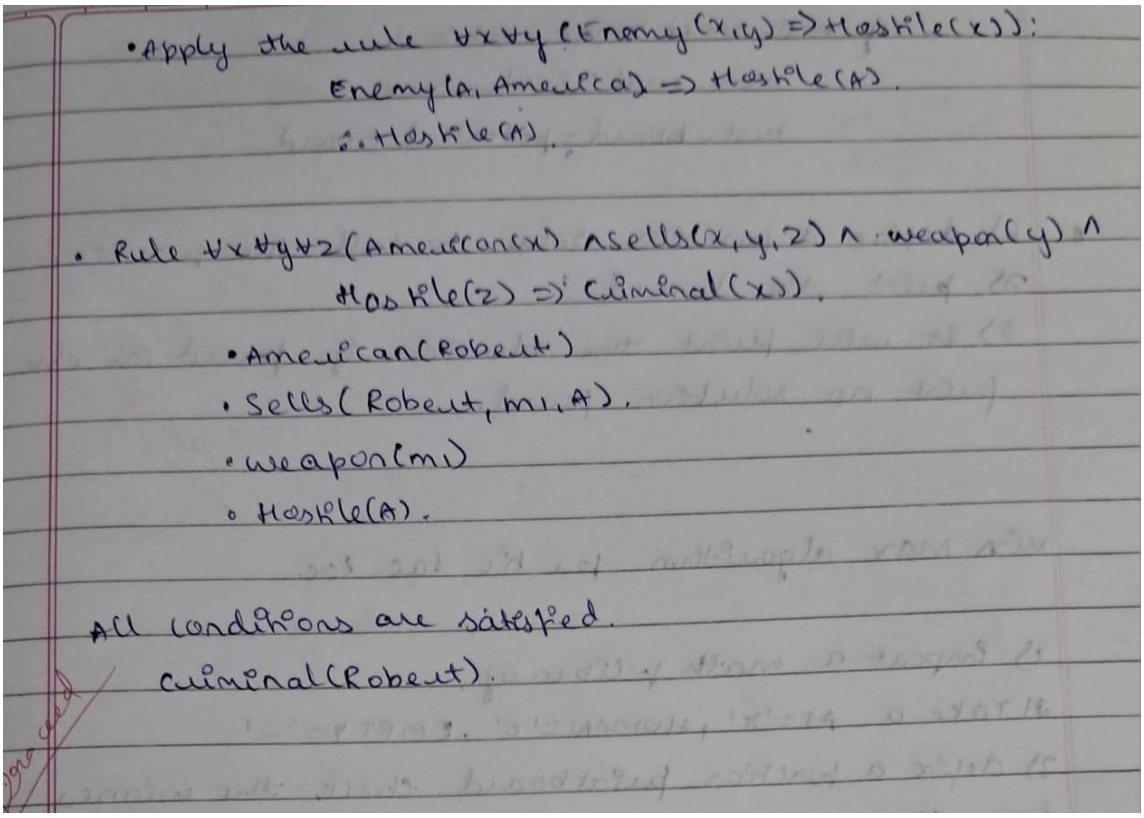
Facts:

Missile(m1).
Owns(A, m1)
Enemy(A, America)
∀x (Missile(x) ⇒ Weapon(x)).
American(Robert).
Sells(Robert, m1, A).

Rule:

Hostility Rule: ∀x∀y(Enemy(x,y) ⇒ Hostile(x)).
Crime Rule:
∀x∀y∀z (American(x) ∧ Sells(x,y,z) ∧ Weapon(y) ∧ Hostile(z) ⇒ Criminal(x)).

• Apply the rule ∀x(Missile(x) ⇒ Weapon(x)):
• Missile(m1) ⇒ Weapon(m1).
∴ Weapon(m1).



• Apply the rule $\forall x \forall y (Enemy(x,y) \Rightarrow Hostile(x))$:

$Enemy(A, America) \Rightarrow Hostile(A)$.

$\therefore Hostile(A)$.

• Rule $\forall x \forall y \forall z (American(x) \land Sells(x,y,z) \land weapon(y) \land Hostile(z) \Rightarrow Criminal(x))$.

• American(Robert)

• Sells(Robert, m1, A).

• weapon(m1)

• Hostile(A).

All conditions are satisfied.

Criminal(Robert).

Code:

```
class ForwardChaining:
    def _init_(self):
        self.facts = set()
        self.rules = []

    def add_fact(self, fact):
        """Add a fact to the knowledge base."""
        self.facts.add(fact)

    def add_rule(self, conditions, conclusion):
        """Add a rule to the knowledge base."""
        self.rules.append((conditions, conclusion))

    def infer(self):
        """Apply forward chaining to infer new facts."""
        inferred = True
        while inferred:  # Continue until no new facts are inferred
            inferred = False
```

```
            for conditions, conclusion in self.rules:
                if conclusion not in self.facts and all(condition in self.facts for condition in conditions):
                    self.facts.add(conclusion)
                    inferred = True

    def prove(self, goal):
        """Check if the goal can be proved."""
        self.infer()
        return goal in self.facts
if _name___== "_main_":
    # Initialize the knowledge base
    fc = ForwardChaining()

    # Add facts
    fc.add_fact("American(Robert)")
    fc.add_fact("Hostile(CountryA)")
    fc.add_fact("OwnsMissiles(CountryA)")
    fc.add_fact("Sells(Robert, Missiles, CountryA)")

    # Add rules
    fc.add_rule(["American(X)", "Sells(X, Weapons, Y)", "Hostile(Y)"], "Criminal(X)")
    fc.add_rule(["OwnsMissiles(Y)"], "Weapons(Y)")  # Missiles are weapons

    # Prove the goal
    goal = "Criminal(Robert)"
    if fc.prove(goal):
        print(f"The goal '{goal}' is TRUE. Robert is a criminal.")
    else:
        print(f"The goal '{goal}' is FALSE. Robert is not a criminal.")
```

OUTPUT:

```
PS C:\Users\pbs82\Downloads\AI> & C:/Users/pbs82/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/pbs82/Downloads/AI/aii.py
The goal 'Criminal(Robert)' is TRUE. Robert is  a criminal.
```

LAB 10: Implement Tic Tac Toe using Min Max

Algorithm:

8 queens using alpha Beta Pruning search algorithm.

Algorithm

1) Define a function init with self and size.

2) Define a another function is_safe with parameters, with self, board, row and column.

3) using a zip combine a row and column

4) Define a function a alpha beta search with parameter self, board, col, alpha, beta, maxi_player.

    if col >= self.size.

        return 0, [row[:] for row in board]

5) check the condition maximizing_player. in that check the is_safe call a function alpha_beta_search function.

if eval_score > max_eval:
max_eval = eval_score
best_board = potential_board.

6) define a function solve to call the function
7) print the board.
8) it will print the solution if found or else print no solution found.

Done

min max algorithm for tic tac toe.

1) import a math library.
2) Take a AI = 'x', HUMAN = 'o', EMPTY = '_'
3) define a function print board. check the winner player X or O has won.
4) checks the board is full and no winner exists.
5) Define a function minmax. with parameter board. depth
6) AI : positive score (10-depth)
Human = negative score (depth-10)
draw: Return 0
7) AI attempts to maximize its score by choosing moves that lead to better outcomes.
simulates all possible moves and recursively evaluates them using min max
Human: player minimizes the score, assuming they play optimally.
8) loops through all empty cells. places the AI's move in an empty cell & evaluates it using the minimax function.

track the moves with the highest score
a) Print the solutions.

AI = 10
Human = -10

→ Output

Current board:

X O X

O X O

— — —.

→ Output:

Solution found.

. Q . . . . . .

Q

Q

Q

Q

Q

Q

Q

→ Output:

→ Robert is a criminal.

Proceed
9/12/24

Code:

```
import math

# Constants for the players
AI = 'X'
HUMAN = 'O'
EMPTY = '_'

# Function to print the board
def print_board(board):
    for row in board:
        print(" ".join(row))
    print()

# Function to check if a player has won
def check_winner(board, player):
    # Check rows, columns, and diagonals
    for row in board:
        if all(cell == player for cell in row):
            return True
    for col in range(3):
        if all(row[col] == player for row in board):
            return True
    if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3)):
        return True
    return False

# Function to check if the game is a draw
def is_draw(board):
    return all(cell != EMPTY for row in board for cell in row)

# Minimax algorithm
def minimax(board, depth, is_maximizing):
    if check_winner(board, AI):
        return 10 - depth
    if check_winner(board, HUMAN):
        return depth - 10
    if is_draw(board):
        return 0

    if is_maximizing:
        best_score = -math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == EMPTY:
                    board[i][j] = AI
                    score = minimax(board, depth + 1, False)
                    board[i][j] = EMPTY
                    best_score = max(best_score, score)
        return best_score
    else:
        best_score = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == EMPTY:
                    board[i][j] = HUMAN
                    score = minimax(board, depth + 1, True)
                    board[i][j] = EMPTY
                    best_score = min(best_score, score)
        return best_score
```

```
# Function to find the best move for AI
def find_best_move(board):
    best_score = -math.inf
    move = (-1, -1)
    for i in range(3):
        for j in range(3):
            if board[i][j] == EMPTY:
                board[i][j] = AI
                score = minimax(board, 0, False)
                board[i][j] = EMPTY
                if score > best_score:
                    best_score = score
                    move = (i, j)
    return move

# Example usage
if __name__== "__main__":
    # Initialize a sample board
    board = [
        ['X', 'O', 'X'],
        ['O', 'X', 'O'],
        ['_', '_', '_']
    ]
    print("Current Board:")
    print_board(board)

    best_move = find_best_move(board)
    print(f"The best move for AI is: {best_move}")
```

OUTPUT:

```
Current Board:
X O X
O X O

_ _ _

The best move for AI is: (2, 0)
```

PART 2: Implement Alpha-Beta Pruning

Code:

```
class EightQueens:
    def __init__(self, size=8):
        self.size = size

    def is_safe(self, board, row, col):
        """Check if placing a queen at board[row][col] is safe."""
        for i in range(col):
            if board[row][i] == 1:  # Check this row on the left
```

```
            return False

        for i, j in zip(range(row, -1, -1), range(col, -1, -1)):  # Check upper diagonal
            if board[i][j] == 1:
                return False

        for i, j in zip(range(row, self.size), range(col, -1, -1)):  # Check lower diagonal
            if board[i][j] == 1:
                return False

        return True

    def evaluate(self, board):
        """Simple heuristic to minimize conflicts."""
        conflicts = 0
        for row in range(self.size):
            for col in range(self.size):
                if board[row][col] == 1:
                    # Count conflicts for current queen
                    conflicts += sum(board[row][:col])  # Same row to the left
                    conflicts += sum(board[i][col] for i in range(row))  # Same column above
                    conflicts += sum(board[row - k][col - k] for k in range(1, min(row, col) + 1))
# Upper diagonal
                    conflicts += sum(board[row + k][col - k] for k in range(1, min(self.size - row,
col) + 1))  # Lower diagonal
        return -conflicts  # Less conflict is better

    def alpha_beta_search(self, board, col, alpha, beta, maximizing_player):
        """Alpha-Beta Pruning Search."""
        if col >= self.size:  # If all queens are placed
            return self.evaluate(board), board

        if maximizing_player:
            max_eval = float('-inf')
            best_board = None
            for row in range(self.size):
```

```
            if self.is_safe(board, row, col):
                board[row][col] = 1
                eval_score, _ = self.alpha_beta_search(board, col + 1, alpha, beta, False)
                board[row][col] = 0
                if eval_score > max_eval:
                    max_eval = eval_score
                    best_board = [row[:] for row in board]
                alpha = max(alpha, eval_score)
                if beta <= alpha:  # Beta cutoff
                   break
        return max_eval, best_board
    else:
        min_eval = float('inf')
        best_board = None
        for row in range(self.size):
            if self.is_safe(board, row, col):
                board[row][col] = 1
                eval_score, _ = self.alpha_beta_search(board, col + 1, alpha, beta, True)
                board[row][col] = 0
                if eval_score < min_eval:
                    min_eval = eval_score
                    best_board = [row[:] for row in board]
                beta = min(beta, eval_score)
                if beta <= alpha:  # Alpha cutoff
                    break
        return min_eval, best_board

def solve(self):
    """Solve the 8-Queens problem."""
    board = [[0] * self.size for _ in range(self.size)]
    _, solution = self.alpha_beta_search(board, 0, float('-inf'), float('inf'), True)
    return solution

def print_board(self, board):
    """Print the chessboard."""
```

```
        for row in board:
            print(" ".join("Q" if col else "." for col in row))
        print()


if _name___== "_main_":
    game = EightQueens()
    solution = game.solve()
    if solution:
        print("Solution found:")
        game.print_board(solution)
    else:
        print("No solution exists.")
```

OUTPUT:

```
Solution found:
. Q . . . . . .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . Q . . . . .
. . . . . . . Q
. . . . . Q . .
. . . Q . . . .
```