VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

RAGHAVENDRA R(1BM22CS214)

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 **RAGHAVENDRA R (1BM22CS214),** 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.

Radhika A D Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

LAB 1: Tic –Tac –Toe Game

Algorithm:

tic-tac-toe: (LAB-1). 24/09/24

man = x : comp = o

O	X	O
O	X	X
X	O	X

Step 1: create a '2D' array. & initialize it with '-'. it is a 3×3 matrix.

Step 2: ⇒ [['-', '-', '-'], ['-', '-', '-'], ['-', '-', '-']]

Step 2: Use random to select the player to go first (human / computer). First move will be 'x'

Display the board after the each move

```
def print-board (board):
    for row in board:
        print ('|'.join(row))
        print('-' *5).
```

Step 3: After each move, check for the win of both the player and computer and if a win then return true for that particular player / for computer

```
def win(board):
    for i in range(3):
        if (board[i][0] == board[i][1] == board[i][2] != ''):
            return true. (board[i][0]).
        if (board[0][i] == board[1][i] == board[2][i] != '')
            return true (board[i][1]).
    // check for diagonals
    if (board[0][0] == board[1][1] == board[2][2] != '')
        return board[0][0]
    if (board[2][0] == board[1][1] == board[0][2] != '')
        return board[0][2]
```

Step 4: player must enter the index for the play and the particular place must be alloted to in the board if not occupied early

```
def player(board):
    row, col = input :
    if (board[row][col] == '-') {:
            board[row][col] = 'x'.
    }
```

Step 5: computer will analyze the moves like it will first prioritize the win moves of its and then avoid the winning possibilities of the player

```
// check winning move.
for (i in range(3):
      for j in range(3):
          if board[i][j] = '-':
              board[i][j] = 'O':
              if ~~check~~(winner()) == 'O':
              return    // returns if the computer
                                can win
```

// place at any random pos:

// get a random place and if it is free in the board place it over there..

Step 6! if ~~wins~~ any player wins, display it.

Code:

```
import random

def win(board):
    for row in board:
        if row[0] == row[1] == row[2] != "":
            return True
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] != "":
            return True
    if board[0][0] == board[1][1] == board[2][2] != "":
        return True
    if board[0][2] == board[1][1] == board[2][0] != "":
        return True
    return False

def printBoard(board):
    print("\n".join([" | ".join(row) for row in board]))

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

def user_move(board):
    while True:
        try:
            move = int(input("Enter your move (1-9): ")) - 1
            row, col = divmod(move, 3)
            if board[row][col] == "":
                board[row][col] = "X"
                break
            else:
                print("That space is already taken. Try again.")
        except (ValueError, IndexError):
            print("Invalid input. Please enter a number from 1 to 9.")

def computer_move(board):
    while True:
        move = random.randint(0, 8)
        row, col = divmod(move, 3)
        if board[row][col] == "":
            board[row][col] = "O"
            break

def _main():
    board = [["" for _ in range(3)] for _ in range(3)]
```

```python
    while True:
        printBoard(board)
        user_move(board)
        if win(board):
            printBoard(board)
            print("You win!")
            break
        if draw(board):
            printBoard(board)
            print("It's a draw!")
            break
        computer_move(board)
        if win(board):
            printBoard(board)
            print("Computer wins!")
            break
        if draw(board):
            printBoard(board)
            print("It's a draw!")
            break

if __name__ == "__main__":
    _main()
```

Output:

```
  |   |
  |   |
  |   |
Enter your move (1-9): 3
  |   | X
  | O |
  |   |
Enter your move (1-9): 2
  | X | X
  | O |
  |   | O
Enter your move (1-9): 1
X | X | X
  | O |
  |   | O
You win!
PS C:\Users\Dell\Desktop\BIS>
```

LAB 2: Vacuum cleaner agent

Algorithm:

LAB-2 : 1/10/2024

* write an algorithm and program for a AI controlled vacuum cleaner.

Step1: create two rooms using two variables A & B.

A	B

Step2: take user input from the user for room A & B as 0 - dirty, 1 - clean.

Step3: The agent is in "Room-A" and it checks if the room is dirty or not. and prints "A-cleaned" and moves to next room.

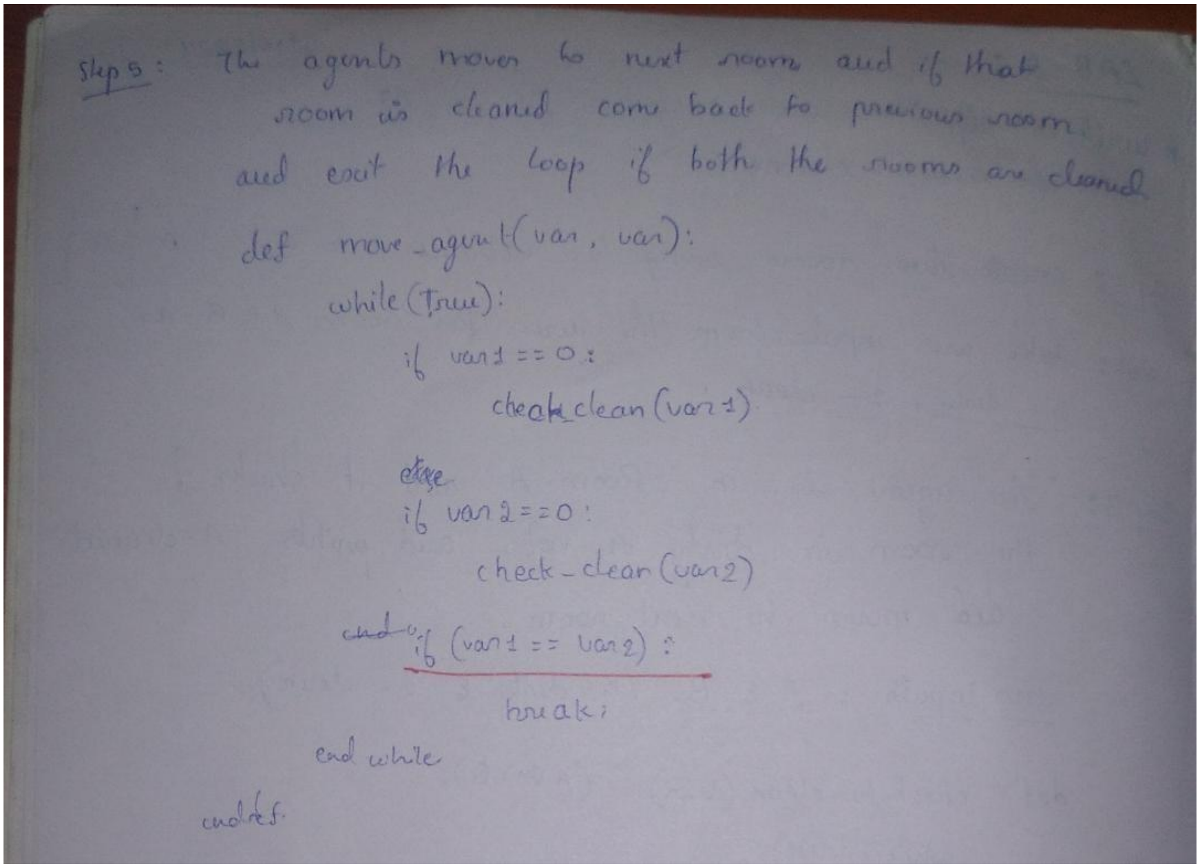
=> user inputs : A & B. (0-dirty & 1-clean)

```
def check_for_clean (var):  (A or B).
    while (true):
        if var == 0:
            clean(var);
        else:
            move(var);
    end while
end def
```

Step4: The room gets cleaned by the agent (vacuum cleaner) and updates the status to 1 from 0 and prints as "cleaned"

```
def clean (var):
    if (var == 0) {
        var == 1.
    end if
        move(var);
end def
```

1/10/24


Step 5: The agents moves to next room and if that room is cleaned come back to previous room and exit the loop if both the rooms are cleaned

```
def move_agent(var, var):
    while(True):
        if var1 == 0:
            check_clean(var1)
        ~~else~~
        if var2 == 0:
            check_clean(var2)
    end if
        if (var1 == var2):
            break;
    end while
enddef.
```

Code: (2 rooms)

```
def printArr(arr):
    n=len(arr)
    print(arr[0],arr[1])

def clean(arr,vac):
    if(arr[vac] == 1):
        arr[vac]=0
    if(arr[vac] == 0):
        return

def check(arr):
    if(arr[0]==0 and arr[1]==0):
        return False
    else:
        return True

print("Enter the status of the room(0 for clean; 1 for dirty):")
arr1 = []
for i in range(0,2):
    a=int(input("Status of the room %d:" %i))
```

```
    arr1.append(a)

vac=0
while(True):
    printArr(arr1)
    if(check(arr1) == False):
        break
    clean(arr1,vac)
    if(vac==0):
        vac=1
    else:
        vac=0
print("Rooms are cleaned!")
```

Output:

```
i/2room.py
Enter the status of the room(0 for clean; 1 for dirty):
Status of the room 0:1
Status of the room 1:1
1 1
0 1
0 0
Rooms are cleaned!
```

Code: (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 = [(0, 1), (1, 0), (0, -1), (-1, 0)]
```

```python
direction_index = 0  # Start moving right
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)

    dx, dy = directions[direction_index]
    new_x, new_y = x + dx, y + dy

    if 0 <= new_x < 2 and 0 <= new_y < 2:
        x, y = new_x, new_y
    else:
        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): 0
Status of room (0, 1): 0
Status of room (1, 0): 0
Status of room (1, 1): 1
[0, 0]
[0, 1]

[0, 0]
[0, 1]

[0, 0]
[0, 1]

[0, 0]
[0, 0]

All rooms are cleaned!
```

LAB 3: 8 Puzzle game

Algorithm:

LAB-3:

08/10/2024

→ write an algorithm & a program to solve 8 puzzle game.

Step1: Initialize goal state and possible moves.

goal_state = [[1,2,3], [4,5,6], [7,8,_]].

moves = [(-1,0), (1,0), (0,-1), (0,1)]

Step2: Function to calculate "Manhattan distance".

~~for i in~~

```
def manhattan(state):
    for i in range(3):
        for j in range(3):
            if (state[i][j] != '_'):
                goal_i, goal_j = divmod(state[i][j] - 1, 3).
                distance += (abs(i - goal_i) - abs(j - goal_j))
            end if
        endfor
    endfor
    return distance.
```

Step 3: check if current_state is goal_state:

```
def check(state):
    return goal_state == current_state
```

Step4: ~~DFS~~ New neighbours

def neighbours(state):

* Loop the matrix from i to 3 & j to 3
* for where, if state[i][j] = '_' → them check all the four directions
* using the moves matrix. you can do that
* add those to a new array
* return the neighbours array

step 5: DFS:

* declare a queue [(state, [state])
* visited array to mark the visited pos.
* loop it untill queue is empty
* check if current is the goal state
* skip already visited states.
  → u can check it using visited array
* if not visited
  → add to the visited array
* again, get the neighbours.
  if no → none

end:

Code:

```
class PuzzleState:
    def __init__(self, board, moves=0, previous=None):
        self.board = board
        self.moves = moves
        self.previous = previous
        self.empty_pos = self.find_empty()

    def find_empty(self):
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return (i, j)

    def manhattan_distance(self):
        dist = 0
        for i in range(3):
            for j in range(3):
                tile = self.board[i][j]
                if tile != 0:
                    target_x = (tile - 1) // 3
                    target_y = (tile - 1) % 3
                    dist += abs(i - target_x) + abs(j - target_y)
        return dist
```

```
    def generate_moves(self):
        moves = []
        x, y = self.empty_pos
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = [row[:] for row in self.board]
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y], 
new_board[x][y]
                moves.append(PuzzleState(new_board, self.moves + 1, self))

        return moves

def dfs(start_board, max_depth):
    stack = [PuzzleState(start_board)]
    visited = set()
    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

    while stack:
        current_state = stack.pop()

        if current_state.board == goal_state:
            return current_state

        visited.add(tuple(map(tuple, current_state.board)))

        if current_state.moves < max_depth:
            for next_state in current_state.generate_moves():
                if tuple(map(tuple, next_state.board)) not in visited:
                    if next_state.manhattan_distance() < 10:
                        stack.append(next_state)

    return None

def print_solution(solution):
    path = []
    while solution:
        path.append(solution.board)
        solution = solution.previous
    for step in reversed(path):
        for row in step:
            print(row)
        print()
```

```python
    print(f"Total moves taken to reach the final state: {len(path) - 1}")


initial_board = [[1, 2, 3], [4, 0, 5], [7, 8, 6]]
max_depth = 10
solution = dfs(initial_board, max_depth)
if solution:
    print("Solution found:")
    print_solution(solution)
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
```

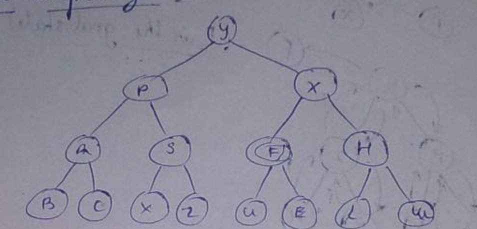
LAB 4: Iterative deepening search and A* algorithm

Algorithm:

LAB - 04:

write an algorithm & code for Iterative deepening depth first search and Solve 8 puzzle using A* Algorithm.

→ Iterative deepening DFS:



* Iterative deepening DFS is a combination of DFS & BFS. it goes to each level and then does DFS till the level.

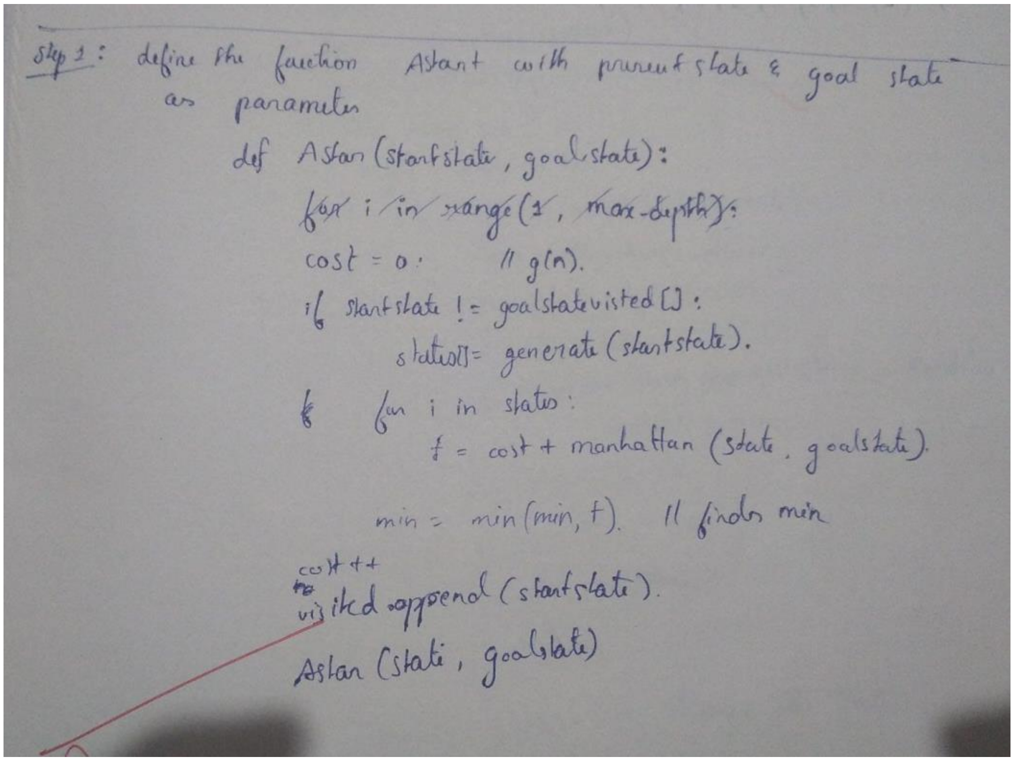
* Step 1: call the limit search function from range(1, max-size)
    * take the goal as i/p.
    * take the graph as i/p.

* Step 2:

```
def IDDFS(graph, limit, head):
    for max-depth = 0 to limit:
        result = DFS(start, demone. head, max-depth).
        if result:
            return result
        else:
            return 0.
        end if
    end for
```

* Step 3:

```
def DepthFirstSearch(head, max-depth, limit):
    if root head == goal:
        return head.
    if depmax-depth == limit + 1:
        return
    for child in root:
        dfs(child, max-depth, limit).
```


Step 1: define the function Astart with present state & goal state as parameters

def Astar(startstate, goalstate):
for i in range(1, max-depth):
cost = 0; // g(n).
if startstate != goalstate visited []:
states[] = generate(startstate).
for i in states:
f = cost + manhattan(state, goalstate).
min = min(min, f). // finds min
cost ++
visited.append(startstate).
Astar(state, goalstate)

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: "))
```

```python
    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()
```

```python
def H_n(state, target):
    return sum(x != y for x, y in zip(state, target))
def F_n(state_with_lvl, target):
    state, lvl = state_with_lvl
    return H_n(state, target) + lvl
def possible_moves(state_with_lvl, visited_states):
    state, lvl = state_with_lvl
    b = state.index(0)
    directions = []
    pos_moves = []
    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')
    for move in directions:
        temp = gen(state, move, b)
        if temp not in visited_states:
            pos_moves.append([temp, lvl + 1])
    return pos_moves
def gen(state, move, b):
    temp = state.copy()
```

```python
    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
def display_state(state):
    print("Current State:")
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()
def astar(src, target):
    arr = [[src, 0]]
    visited_states = []
    iterations = 0
    while arr:
        iterations += 1
        current = min(arr, key=lambda x: F_n(x, target))
        arr.remove(current)
        display_state(current[0])
        if current[0] == target:
            return f'Found with {iterations} iterations'
        visited_states.append(current[0])
        arr.extend(possible_moves(current, visited_states))
    return 'Not found'
src = [1, 2, 3, 8, 0, 4, 7, 6, 5]
target = [2, 8, 1, 0, 4, 3, 7, 6, 5]
print(astar(src, target))
```

Output:

```
i/iter.py
Enter the number of edges: 4
Enter each edge in the format 'node1 node2':
a b
a c
b d
c e
Enter the starting node: a
Enter the goal node: e
Path found: a -> c -> e
```

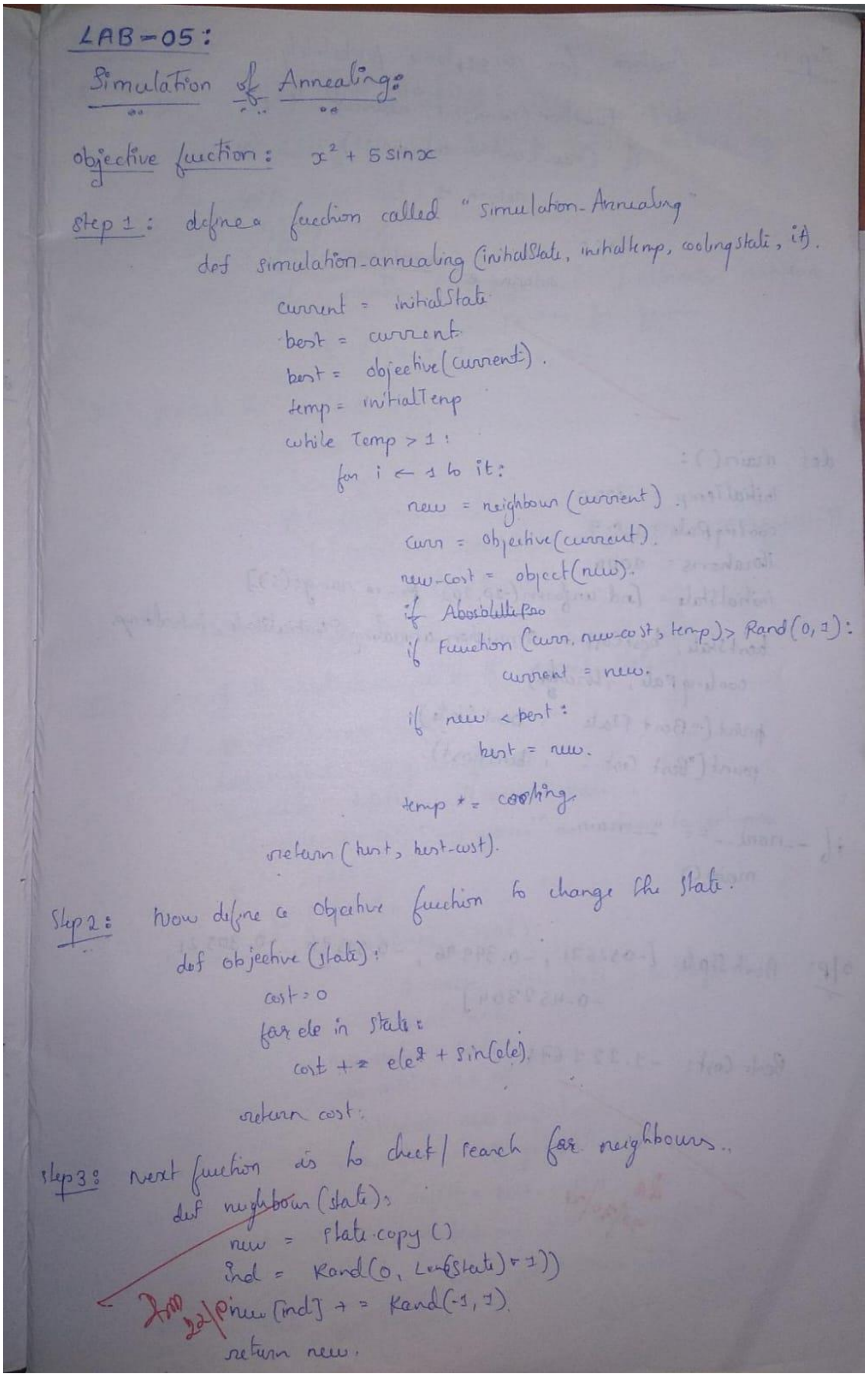
```
Current State:
[0, 8, 1]
[2, 4, 3]
[7, 6, 5]

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

Found with 40 iterations
```

LAB 5: Simulated annealing.

Algorithm:



LAB-05:

Simulation of Annealing:

objective fuction: $x^2 + 5\sin x$

step 1: define a fuction called "simulation-Annealing"

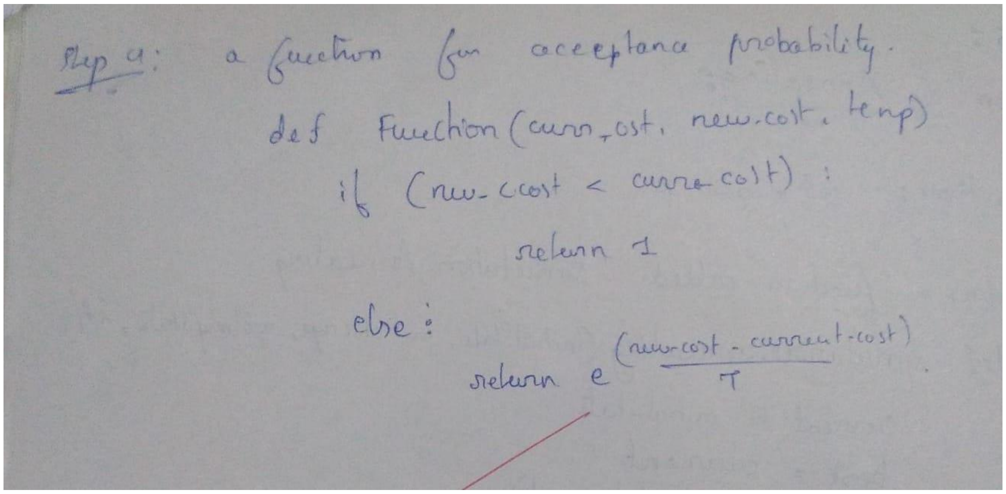
```
def simulation-annealing (initialState, initialtemp, coolingstate, it).
        current = initialState
        best = current
        best = objective(current).
        temp = initialTemp
        while Temp > 1 :
            for i <- 1 to it:
                new = neighbour (current).
                curr = objective(current).
                new-cost = object(new).
                if Absolutilepao
                if Function (curr, new-cost, temp) > Rand(0, 1):
                        current = new.
                if new < best :
                        best = new.
                temp *= cooling
        return (best, best-cost).
```

Step 2: Now define a objective fuction to change the state.

```
def objective (state):
        cost = 0
        for ele in state:
            cost += ele^2 + sin(ele).
        return cost.
```

step 3: Next fuction is to check/search for neighbours..

```
def neighbour (state):
        new = state.copy ()
        ind = Rand(0, Len(state) - 1))
        new [ind] += Rand(-1, 1).
        return new.
```



Step 4: a function for acceptance probability.

def Function (curr_cost, new_cost, temp)

if (new_cost < curr_cost) :

return 1

else :

return $e^{\frac{(new\_cost - current\_cost)}{T}}$

Code:

```
import random
import math

def energy(x):
    return x ** 2 + 5 * math.sin(x) + math.exp(-x)

def adaptive_simulated_annealing(start, temp, cooling_rate, lower_limit,
upper_limit):
    current = start
    current_energy = energy(current)

    while temp > 1:
        # Adaptive step size based on temperature (larger steps when hot)
        step_size = random.uniform(-1, 1) * temp
        new = current + step_size

        # Ensure new solution is within bounds
        if new < lower_limit or new > upper_limit:
            continue

        new_energy = energy(new)

        # If the new spot is better, move there
        if new_energy < current_energy:
            current = new
            current_energy = new_energy
        else:
            # Acceptance probability (explore worse spots)
            probability = math.exp((current_energy - new_energy) / temp)
            if random.uniform(0, 1) < probability:
```

```
                current = new
                current_energy = new_energy

        # Adaptive cooling based on progress
        if abs(new_energy - current_energy) < 0.01:
            temp *= 0.98  # Slow cooling near solution
        else:
            temp *= cooling_rate

    return current

# Run the simulation multiple times from different starting points
best_solution = None
for _ in range(10):  # 10 runs
    result = adaptive_simulated_annealing(start=random.uniform(-10, 10), temp=100,
cooling_rate=0.99, lower_limit=-10, upper_limit=10)
    if best_solution is None or energy(result) < energy(best_solution):
        best_solution = result

print(f"Best solution found: {best_solution}")
```

Output:

```
i/anneal.py
Best solution found: -0.7390095302681031
```

LAB 6: 8 queens using Hill Climb and A* Algorithm.

Algorithm:

* LAB -6.

1. 8 queens using Hill climbing :

Step 1 : define the init function / method. to initialize table & size

```
def __init__(size=8, state).
        self.size = size
        self.state = state.
        state = random(). // generates / allocates random
                                 position for queens
             eg: [2, 0, 4, 7, 1, 3, 6, 5].
```

Step

Step 2 : Generate the conflicts.

```
fun i -> 0 to size :
    fun j -> 0 to size :
        if conflicts = 0 :
          if (state[i] == state[j] && abs(state[i]-state[j]) == abs(i-j))
                conflict++.
return conflicts
```

Step 3 : Generate the best successor for the current state :

```
def get-best-successor(self):
    best-conflict = self.get-conflict().
    best.state = state[i].    // current state
    if (state[row] != col). : // queen is not in target pos
    for i in row:
        for j in col :
            if (state[row] != col):
                    // create a new state & initialize it to
                       that state
               new-state = state
               new-state[row] = col
               // calculate new conflict
               new-conflict = self. get-conflict().
               if new-conflict < best-conflict :
                      best-conflict = new-conflict
                          best-state = new-state
return best-state, best conflict
```

Step 4 : def hill climbing (self):
current-conflict = get-~~self~~conflict() get_best_successor
new-state, new-conflict = self. get_successor().
if (new-conflict > current-conflict)
return self.state : if current-conflict == 0 else none
~~current~~ self state = new-state.

O/P:

2. 8 queens using A* algorithm:

Step 1: initialize:
def _init_ (self, state=none):
self.size = size // 8
self.state = [~~random~~] // allocates random positions for queen

Step 2: is_goal(self):
return self. getconflict == 0

Step 3: get-conflict (self):
for i -> 0 to size:
for j -> 0 to size:
conflicts = 0
if (check for same col. & diagonal)
conflict ++.
return conflict.

Step 4: def_ get-cost (~~self~~ state): // calls the conflict function
return get- conflict().

~~state~~

step 5: def get_successor(state, size)

// run two loops to find the successors

// store them in successor array

successor = board(new_state)

- successors.append(successor.getcost, successor)

step 6: // initialize board

initial board = board()

open_set = []

heapq.heappush(~~initial board~~, openset, (initial board.getcost(), initial))

while ~~state~~ openset:

current_cost, board = heapq.heappop(openset)

if board.isgoal():

return board.state

for cost, successors in board.get_successors():

heapq.heappush(openset, (cost + 1), successor).

o/p:

Code:

```
import random

# Helper function to calculate the heuristic (number of conflicts)
def heuristic(board):
    conflicts = 0
    for i in range(len(board)):
        for j in range(i + 1, len(board)):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                conflicts += 1
    return conflicts

# Hill climbing for 8-queens
def hill_climbing_8_queens():
    n = 8
    # Generate a random initial state
    board = [random.randint(0, n - 1) for _ in range(n)]
```

```
    while True:
        current_h = heuristic(board)
        if current_h == 0:
            return board  # Solution found

        # Find the best neighbor by moving each queen to every other column in its
row
        best_board = board[:]
        best_h = current_h
        for row in range(n):
            for col in range(n):
                if col == board[row]:
                    continue
                new_board = board[:]
                new_board[row] = col
                new_h = heuristic(new_board)

                # If the new board has fewer conflicts, update the best board
                if new_h < best_h:
                    best_h = new_h
                    best_board = new_board

        # If no improvement, we're stuck in a local minimum; restart
        if best_h >= current_h:
            board = [random.randint(0, n - 1) for _ in range(n)]
        else:
            board = best_board

# Run hill climbing search
solution = hill_climbing_8_queens()
print("Solution board (column positions for each row):", solution)
```

```
import heapq

# Helper function to calculate the heuristic (number of conflicts)
def heuristic(board):
    conflicts = 0
    for i in range(len(board)):
        for j in range(i + 1, len(board)):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                conflicts += 1
    return conflicts

# A* Search for 8-queens
def a_star_8_queens():
```

```python
    n = 8
    open_set = []
    # Initial state: empty board
    heapq.heappush(open_set, (0, []))  # (f, board)

    while open_set:
        f, board = heapq.heappop(open_set)

        # Goal check
        if len(board) == n and heuristic(board) == 0:
            return board

        # Generate successors
        row = len(board)
        for col in range(n):
            new_board = board + [col]
            if heuristic(new_board) == 0:  # No conflicts so far
                g = row + 1
                h = heuristic(new_board)
                heapq.heappush(open_set, (g + h, new_board))


    return None  # No solution found

# Run A* search
solution = a_star_8_queens()
print("Solution board (column positions for each row):", solution)
```

Output:

```
i/Hill.py
Solution board (column positions for each row): [3, 1, 6, 4, 0, 7, 5, 2]
PS C:\Users\Dell\Desktop\BIS>
```

```
i/AA.py
Solution board (column positions for each row): [0, 4, 7, 5, 2, 6, 1, 3]
PS C:\Users\Dell\Desktop\BIS>
```

LAB 7: knowledge base using propositional logic

Algorithm:

LAB-07:

→ Propositional Logic:

1) $P \rightarrow \varphi$ (if p is true, then φ must be true).
2) P (we know P is true).

→ we can infer φ.

→ Formal Representations of Entailment:

$$P, (P \rightarrow \varphi) \models \varphi$$

"given P & $P \rightarrow \varphi$, it logically follows that φ must be true". ($\models$ represents Logical entailment)

Step 1:

Knowledge Base:

1) Alice is the mother of Bob
2) Bob is the father of charlie
3) A father is a parent
4) A mother is a parent
5) All parents have children
6) if someone is a parent, their children are siblings
7) Alice is married to David

Hypothesis:

- "charlie is a sibling of Bob"

) M(Alice, Bob)

) F(Bob, charlie)

) Father(x) → Parent(x)

4) Mother(x) → parent(x)

5) Parent(x) → Haschildren(x)

6) parent(x) ∧ haschildren(x) → sibling(children(x))

7) ~~Married(Alice, David)~~

Logical reasoning:

1) 1 & 4:

Alice is mother of Bob & all mothers are parents,
So, Alice is a parent

2) From 2 & 3:

Bob is the father of charlie & all fathers are parent,
So, Bob is a parent

3) From 5:

Alice & Bob are both parents & have children.

4) From 6:

Since, Alice & Bob are parents & their children
Bob & charlie are siblings

Sibling(charlie, Bob) can be logically concluded

11/11/24

Code:

```python
# Define the knowledge base
knowledge_base = {
    "parent": {
        "Alice": ["Bob"],  # Alice is the parent of Bob
        "Bob": ["Charlie"]  # Bob is the parent of Charlie
    },
    "married": {
        "Alice": "David"  # Alice is married to David
    }
}

# Define helper functions
def is_parent(parent, child):
    """Check if a person is a parent of a given child."""
    return child in knowledge_base["parent"].get(parent, [])

def get_children(parent):
    """Retrieve all children of a given parent."""
    return knowledge_base["parent"].get(parent, [])
```

```python
def are_siblings(person1, person2):
    """Check if two people are siblings."""
    for parent, children in knowledge_base["parent"].items():
        if person1 in children and person2 in children:
            return True
    return False

# Hypothesis: "Charlie is a sibling of Bob"
hypothesis = are_siblings("Charlie", "Bob")

# Solve and print the result
if hypothesis:
    print("The hypothesis 'Charlie is a sibling of Bob' is TRUE.")
else:
    print("The hypothesis 'Charlie is a sibling of Bob' is FALSE.")
```

Output:

```
i/entail.py
The hypothesis 'Charlie is a sibling of Bob' is FALSE.
```

LAB 8: Unification in First Order Logic.

Algorithm:

Statements:

1) Every farmer grows crops to feed their village
2) if it rains, crops grow together
3) Mohan is a farmer
4) The village has large field where farmer grows crops
5) if someone grows crops and the crops grow faster, village prospers
6) Today, it rained heavily in the village

1) $\forall x\,(\text{Farmer}(x) \longrightarrow \text{Growscrops}(x))$
2) $\forall y\,(\text{Rains}(y) \longrightarrow \text{GrowsFaster}(y))$
3) Farmer(Mohan) $\land$ Livesin (Mohan).
4) $\exists z\,(\text{Field}(z) \land \text{UsedForCrops}(z))$
5) $\forall w\,(\text{GrowsCrops}(w) \land \text{GrowsFaster}(w) \rightarrow \text{village prospers})$
6) Rains(Today)

1) (3) $\rightarrow$ ~~Mohan~~ Farmer(Mohan)

(1) $\rightarrow$ if Mohan is a farmer, he grows crops ( ).

(6) $\rightarrow$ it rained today (Rains (Today))

(2) $\rightarrow$ CropsGrowFaster (Today)),

(5) $\rightarrow$ if Mohan grows crops.,

(5) By combining above

VillageProspers(Mohan, Today) : TRUE

Code:

```
def unify(x, y, subst=None):
    """
    Unification Algorithm: Unifies two terms, X and Y.
    """
    if subst is None:
        subst = {}

    if x == y:  # Step 1(a): If X and Y are identical
        return subst
    elif isinstance(x, str) and x.islower():  # Step 1(b): If X is a variable
        return unify_variable(x, y, subst)
    elif isinstance(y, str) and y.islower():  # Step 1(c): If Y is a variable
        return unify_variable(y, x, subst)
    elif isinstance(x, tuple) and isinstance(y, tuple):  # Step 2: Check predicates and arguments
        if x[0] != y[0] or len(x) != len(y):  # Predicate symbol or argument count mismatch
            return None
        for x_i, y_i in zip(x[1:], y[1:]):  # Step 5: Recurse through arguments
            subst = unify(x_i, y_i, subst)
            if subst is None:
                return None
        return subst
    else:
        return None  # Step 1(d): Failure case


def unify_variable(var, x, subst):
    """
    Unify variable with another term.
    """
    if var in subst:
        return unify(subst[var], x, subst)
    elif occurs_check(var, x, subst):  # Check if var occurs in x
        return None
    else:
        subst[var] = x
        return subst


def occurs_check(var, x, subst):
    """
    Check if a variable occurs in a term.
    """
```

```python
    if var == x:
        return True
    elif isinstance(x, tuple):
        return any(occurs_check(var, xi, subst) for xi in x)
    elif isinstance(x, str) and x in subst:
        return occurs_check(var, subst[x], subst)
    return False


# Test cases for unification
x1 = ("P", "a", "x")
y1 = ("P", "a", "b")

x2 = ("Q", "x", ("R", "x"))
y2 = ("Q", "a", ("R", "a"))

print("Unifying", x1, "and", y1, "=>", unify(x1, y1))  #
print("Unifying", x2, "and", y2, "=>", unify(x2, y2))
```

Output:

```
i/unify.py
Unifying ('P', 'a', 'x') and ('P', 'a', 'b') => {'x': 'b'}
Unifying ('Q', 'x', ('R', 'x')) and ('Q', 'a', ('R', 'a')) => {'x': 'a'}
```

LAB 9: knowledge base consisting of first order logic.

Algorithm:

LAB-09

First order logic → Prove the given query using forward reasoning
→ starts with a base state and uses the inference rules and available knowledge in the forward direction till it reaches the goal state

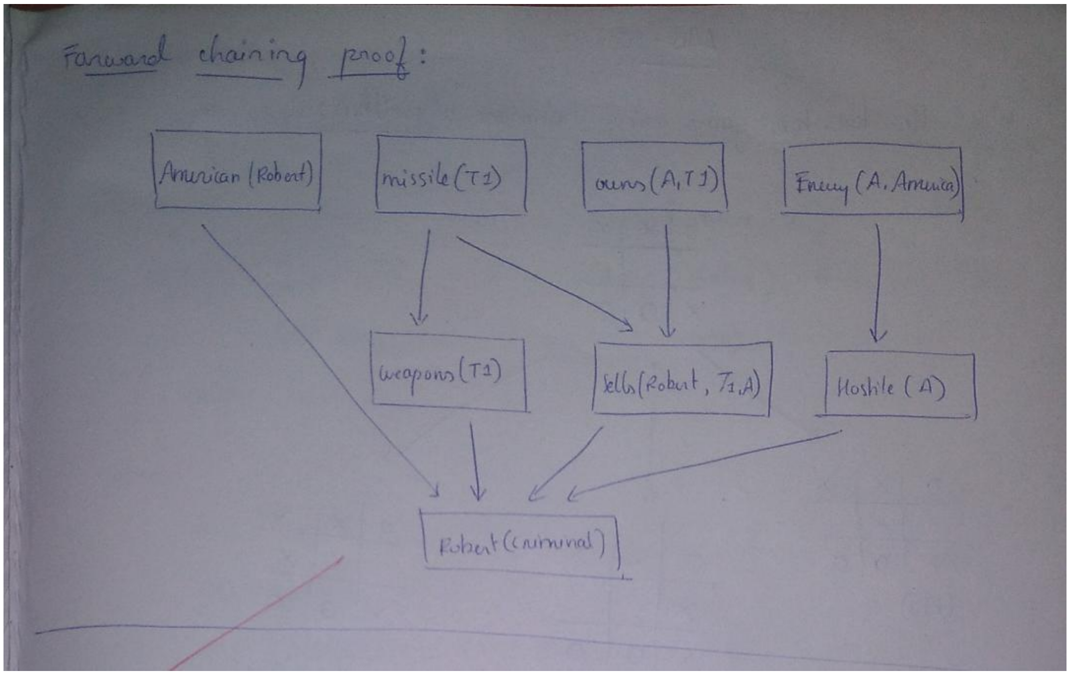
Question:
As per the law, it is a crime for an american to sell weapons to hostile nation. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robert, who is an american citizen"
→ Prove that - "Robert is criminal."

Given facts:
→ Robert is a american ⇒ American(Robert)
→ country A is enemy of america. ⇒ Enemy(A, america).
→ It is a crime to sell weapons to hostile nations ⇒.
$American(P) \wedge weapons(q) \wedge Sells(P, q, r) \wedge Hostile(r) \Rightarrow criminal(P)$

→ country A has some missiles
$\exists x \; owns(A, x) \wedge missile(x)$.
→ $owns(A, T1)$
→ $Missile(T1)$.
→ All of missiles were sold to country A by Robert
$\forall x \; Missiles(x) \wedge owns(A, x) \Rightarrow Sells(Robert, x, A)$

→ Missiles are weapons.
$Missiles(x) \rightarrow weapons(x)$

→ Enemy of america is known as hostile.
$\forall x \; Enemy(x, America) \Rightarrow Hostile(x)$.



Code:

```
# Define the knowledge base (KB) as a set of facts
KB = set()

# Premises based on the provided FOL problem
KB.add('American(Robert)')
KB.add('Enemy(America, A)')
KB.add('Missile(T1)')
KB.add('Owns(A, T1)')

# Define inference rules
def modus_ponens(fact1, fact2, conclusion):
    """ Apply modus ponens inference rule: if fact1 and fact2 are true, then
conclude conclusion """
    if fact1 in KB and fact2 in KB:
        KB.add(conclusion)
        print(f"Inferred: {conclusion}")

def forward_chaining():
    """ Perform forward chaining to infer new facts until no more inferences can be
made """
    # 1. Apply: Missile(x) → Weapon(x)
    if 'Missile(T1)' in KB:
        KB.add('Weapon(T1)')
        print(f"Inferred: Weapon(T1)")

    # 2. Apply: Sells(Robert, T1, A) from Owns(A, T1) and Weapon(T1)
```

```python
    if 'Owns(A, T1)' in KB and 'Weapon(T1)' in KB:
        KB.add('Sells(Robert, T1, A)')
        print(f"Inferred: Sells(Robert, T1, A)")

    # 3. Apply: Hostile(A) from Enemy(A, America)
    if 'Enemy(America, A)' in KB:
        KB.add('Hostile(A)')
        print(f"Inferred: Hostile(A)")

    # 4. Now, check if the goal is reached (i.e., if 'Criminal(Robert)' can be
inferred)
    if ('American(Robert)' in KB and
        'Weapon(T1)' in KB and
        'Sells(Robert, T1, A)' in KB and
        'Hostile(A)' in KB):
        KB.add('Criminal(Robert)')
        print("Inferred: Criminal(Robert)")

    # Check if we've reached our goal
    if 'Criminal(Robert)' in KB:
        print("Robert is a criminal!")
    else:
        print("No more inferences can be made.")

# Run forward chaining to attempt to derive the conclusion
forward_chaining()
```

Output:

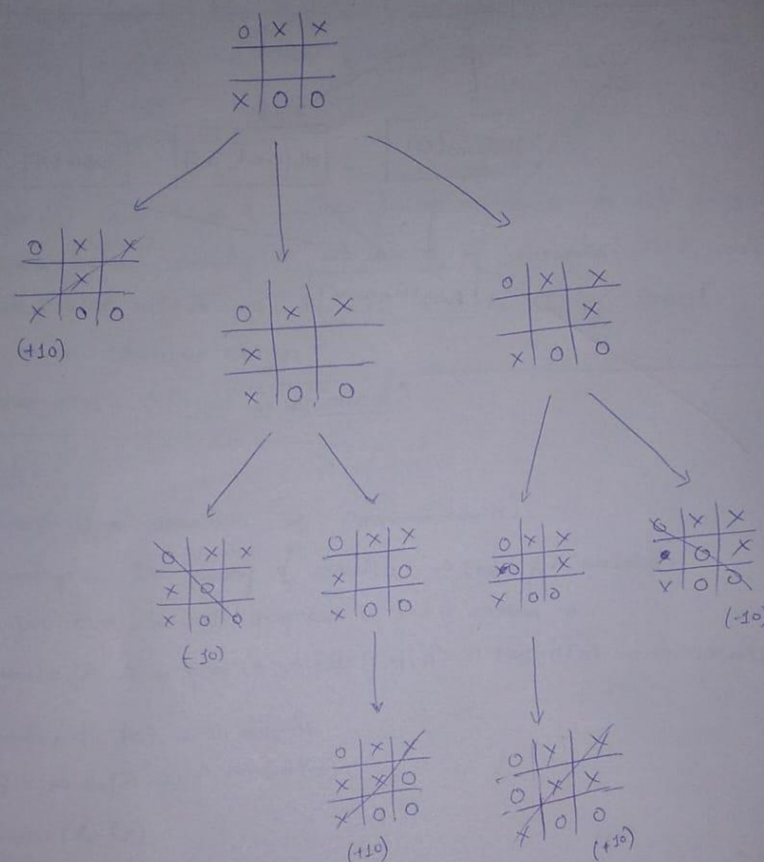
```
i/forward.py
Inferred: Weapon(T1)
Inferred: Sells(Robert, T1, A)
Inferred: Hostile(A)
Inferred: Criminal(Robert)
Robert is a criminal!
```

LAB 10: Min-max algorithm and alpha-beta pruning.

Algorithm:

LAB-10

=> Solve tic-tac-toe game using minmax algorithm:



Algorithm:

```
def minimax(board, is_maximizing):
    if is_winner(board, 'x'):
        return 10
    if is_winner(board, 'o'):
        return -10;
    if is_draw(board):
        return 0;
```

```
if is maximizing:
    best_score = -float('inf')
    for move in possible_moves(board):
        new_board = make_move(board, move, 'x')
        score = minimax(new_board, False)  # 'o' turn
        best_score = max(best_score, score)
    return best_score

else:
    best_score = float('inf')
    for move in possible_moves(board):
        new_board = make_move(board, move, 'o')
        ~~new_board~~ =
         score = minimax(new_board, True)  # 'x' turn
        best_score = min(bestscore, score)
    return best_score

def make_move(board, move, player):
    new_board = board[:]
    new_board[move] = player.
return new_board
```

LAB - 11

⇒ Solve 8 queens problems using alpha-beta pruning.

Algorithm:

```
def alpha-beta (self.board, col, alpha, beta, maximizing.player)
    if col >= self.size:
        return 0. [row[:] for row in board]

    if maximizing player:
        max-eval = float('inf')
        best-board = None
        for row in range (self. size):
            if self. issafe (board, row, col):
                board[row][col] = 1
                eval-score, potential-board = 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 = potential-board.
                alpha = max (alpha, eval_score)
                if beta <= alpha
                 break
        return max.eval, best-board

    else:
        _ " __
        beta = min (beta, eval-score)
        if beta <= alpha:
            break;
        return min-eval, best-board.
```

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):
```

```python
            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}")
```

```python
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 alpha_beta_search(self, board, col, alpha, beta, maximizing_player):
        """Alpha-Beta Pruning Search."""
        if col >= self.size:  # If all queens are placed
            return 0, [row[:] for row in board]  # Return 0 as heuristic since it's a valid solution

        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, potential_board = 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 = potential_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, potential_board = 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 = potential_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:

```
i/minmax.py
Current Board:
X O X
O X O
_ _ _

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

```
i/alpha.py
Solution found:
. Q . . . . . .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . Q . . . . .
. . . . . . . Q
. . . . . Q . .
. . . Q . . . .
```