

VISVESVARAYA TECHNOLOGICAL UNIVERSITY
“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Rani Aishwarya H S (1BM22CS217)

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 **Rani Aishwarya H S (1BM22CS217)**, 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. Kavitha Sooda Professor & HOD Department of CSE, BMSCE |
|--|--|

Index

| Sl. No. | Date | Experiment Title | Page No. |
|--------------------|-------------|---|-----------------|
| 1 | 24-09-2024 | Implement Tic –Tac –Toe Game | 4-8 |
| 2 | 01-10-2024 | Implement vacuum cleaner | 9-12 |
| 3 | 08-10-2024 | Implement 8 puzzle problems | 13-21 |
| 4 | 15-10-2024 | Implement Iterative deepening search algorithm Implement A* search algorithm | 22-30 |
| 5 | 22-10-2024 | Simulated Annealing to Solve 8-Queens problem | 31-35 |
| 6 | 29-10-2024 | Implement Hill Climbing search algorithm to solve N-Queens problem Implement A * search algorithm | 36-43 |
| 7 | 12-11-2024 | Implement propositional logic and show that the given query entails the knowledge base | 44-47 |
| 8 | 19-11-2024 | Implement unification in first order logic | 48-51 |
| 9 | 03-12-2024 | Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning. | 52-55 |
| 10 | 03-12-2024 | Implement Alpha-Beta Pruning. | 56-61 |
| | | | |

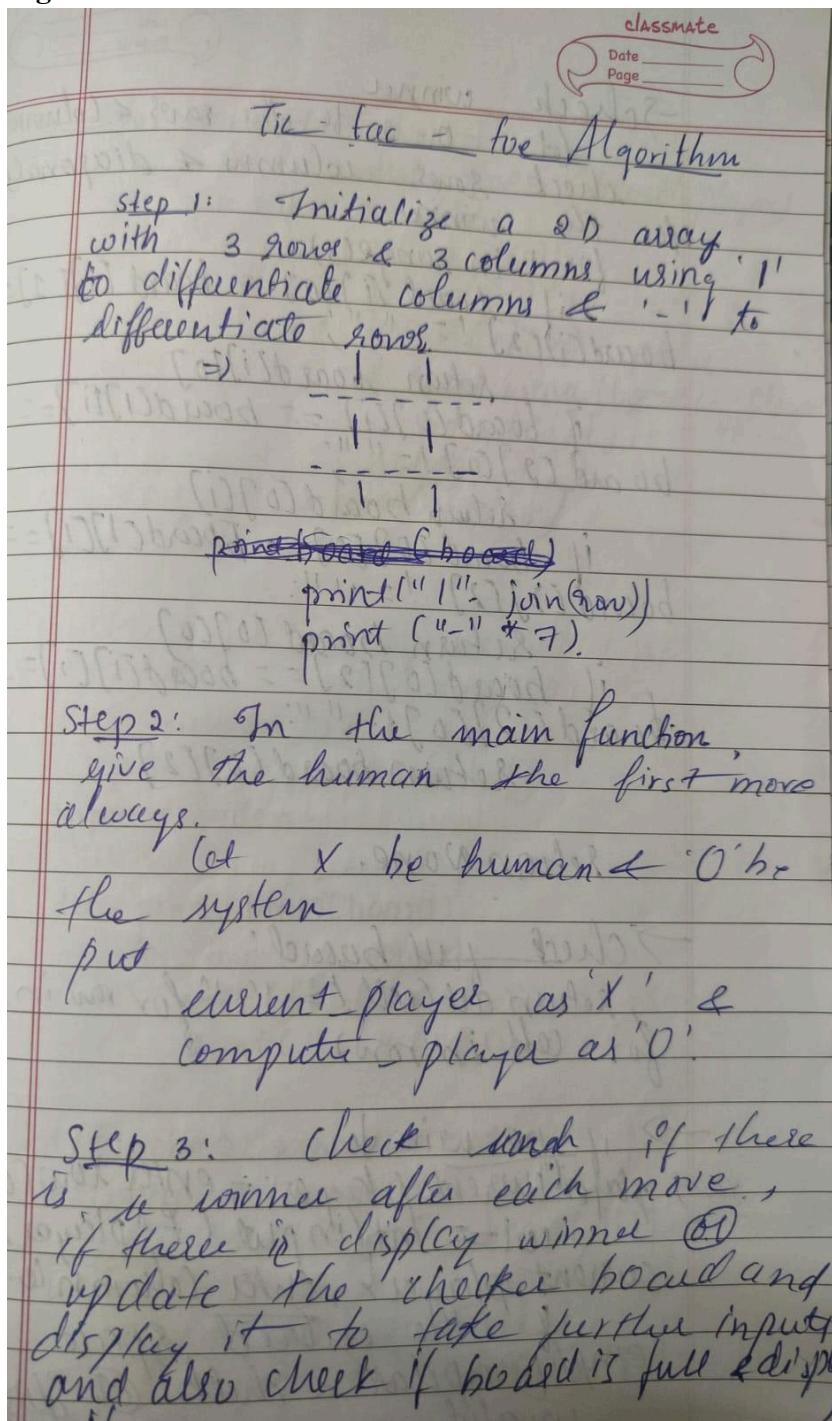
Github Link:

<https://github.com/likitha-zzie/AI-LAB>

Program 1

Implement Tic - Tac - Toe Game

Algorithm:



→ check winner
 let 0-2 be the rows & columns
 for check rows, columns & diagonally
 for the 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.

→ check full board:

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

→ if no win ↓

row player current player - enter row(0-2):
 col -> int(input("Player
current player, enter column(0-2):")
 empty update & check @ display
 invalid

Page

~~for computer move~~

find winning move (board, player)

```

for i in range(3):
    for j in range(3):
        if board[i][j] == " ":
            board[i][j] = player
            if check_winner(board) == player:
                board[i][j] = " "
                return board[i][j]

```

return None.

~~python code:~~

```

import random

def print_board(board):
    for row in board:
        print(" | ".join(row))
        print(" - " * 9)

def check_winner(board):
    for i in range(3):
        if board[0][i] == board[1][i] == board[2][i] == board[0][0]:
            return board[0][0]
        if board[0][i] == board[1][i] == board[2][i] == board[0][2]:
            return board[0][2]

```

Code:

```

# Initialize the board
board = [' ' for _ in range(9)]

# Function to draw the board
def draw_board():
    row1 = '| {} | {} | {} |'.format(board[0], board[1], board[2])
    row2 = '| {} | {} | {} |'.format(board[3], board[4], board[5])
    row3 = '| {} | {} | {} |'.format(board[6], board[7], board[8])
    print()
    print(row1)
    print(row2)
    print(row3)
    print()

```

```

# Function for player's move
def player_move(icon):
    if icon == 'X':
        number = 1
    elif icon == 'O':
        number = 2
    print("Your turn, player {}".format(number))
    choice = int(input("Enter your move (1-9): ").strip())
    if board[choice - 1] == '':
        board[choice - 1] = icon
    else:
        print()
        print("That space is taken!")

# Function to check for victory
def is_victory(icon):
    if (board[0] == icon and board[1] == icon and board[2] == icon) or \
       (board[3] == icon and board[4] == icon and board[5] == icon) or \
       (board[6] == icon and board[7] == icon and board[8] == icon) or \
       (board[0] == icon and board[3] == icon and board[6] == icon) or \
       (board[1] == icon and board[4] == icon and board[7] == icon) or \
       (board[2] == icon and board[5] == icon and board[8] == icon) or \
       (board[0] == icon and board[4] == icon and board[8] == icon) or \
       (board[2] == icon and board[4] == icon and board[6] == icon):
        return True
    else:
        return False

# Function to check for a draw
def is_draw():
    if '' not in board:
        return True
    else:
        return False

# Function to play the game
def play_game():
    draw_board()
    while True:
        player_move('X')
        draw_board()
        if is_victory('X'):
            print("Player 1 wins! Congratulations!")
            break
        elif is_draw():
            print("It's a draw!")

```

```
break

player_move('O')
draw_board()
if is_victory('O'):
    print("Player 2 wins! Congratulations!")
    break
elif is_draw():
    print("It's a draw!")
    break

# Start the game
play_game()
```

Output:

```
[x] ⌂
  | | |
  | | |
  | | |

Your turn player 1
Enter your move (1-9): 1

| X |
|   |
|   |

Your turn player 2
Enter your move (1-9): 9

| X |   |
|   |   0
|   |

Your turn player 1
Enter your move (1-9): 2

| X | X |
|   |   |
|   |   0
|   |

Your turn player 2
Enter your move (1-9): 8

| X | X |
|   |   |
|   | 0 0
|   |

Your turn player 1
Enter your move (1-9): 3

| X | X | X |
|   |   |   |
|   | 0 0 |
|   |

Player 1 wins! Congratulations!
```

```
Your turn player 1
Enter your move (1-9): 1
| x |   |
|   |   |
|   |   |

Your turn player 2
Enter your move (1-9): 9
| x |   |
|   |   |
|   | o |

Your turn player 1
Enter your move (1-9): 3
| x |   | x |
|   | o |   |
|   |   | o |

Your turn player 2
Enter your move (1-9): 2
| x | o | x |
|   |   | o |
|   |   |   |

Your turn player 1
Enter your move (1-9): 7
| x | o | x |
| x |   | o |
|   |   |   |

Your turn player 2
Enter your move (1-9): 4
| x | o | x |
| x |   | o |
|   |   |   |

Your turn player 1
Enter your move (1-9): 6
| x | o | x |
| o |   | o |
| x |   |   |

Your turn player 2
Enter your move (1-9): 5
| x | o | x x |
| x |   | o |
| x |   |   |

Your turn player 1
Enter your move (1-9): 8
| x | o | x x |
| x | x | o |
| x | x | o |

It's a draw!
```

Program 2

Implement vacuum cleaner agent

Lab - 02 Date : 11/10/24

Write a program for implementation of a AI controlled vacuum cleaner.

Algorithm:

Step 1: Create two rooms A and B
Step 2: Take input from the user whether the room is clean or dirty. i.e., Clean → C & Dirty → D
Step 3: Initially both the rooms are dirty initially.

Step 4: The vacuum cleaner agent in room A checks if it is dirty if yes - cleans if not then A → cleaned and moves right i.e., next room B.

def check_if_clean(vac): (i.e., A@)
 while (true):
 if (roomval == D):
 clean(vac):
 else
 move(vac)
 end .
end def

→ If the room A is in status dirty, the loop executes implementing clean(A); if else the agent will move to the next room i.e. to its right and again checks the room's status
 if $B == D$:
 clean B;
 else
 move(A);
 → If both of them are cleaned then it breaks the loop
 ↗
 11/10/24

→ Program:
 print("0 - dirty and 1 - clean")
 roomA = int(input("Enter status for room A:"))
 roomB = int(input("Enter status for room B:"))
 def clean(room, RoomName):
 if cleanRoom == 0:
 room = 1
 print(RoomName + " cleaned")

Code:

```

# Reflex function for vacuum cleaner
def reflex(loc, status, cost):
    s = status # Track the current status of the location
    if status == 1: # If the location is dirty
        cost += 1
        print(f"SUCK at {loc}")
    s = 0 # The location is now clean
  
```

```

if loc == "A":
    print("Move RIGHT to B")
    loc = "B" # Move to B
elif loc == "B":
    print("Move LEFT to A")
    loc = "A" # Move to A
return cost, loc, s # Return updated cost, location, and status

# Function to check goal state
def goal(a_status, b_status):
    if a_status == 0 and b_status == 0:
        print("Goal reached")
    else:
        print("Goal not reached")

# Input for the starting location and statuses
loc = input("Enter the starting location of the vacuum (A or B): ").strip()
cost = 0
a_status = int(input("Enter the status of location A (0 for clean, 1 for dirty): "))
b_status = int(input("Enter the status of location B (0 for clean, 1 for dirty): "))

# Simulate cleaning process
if loc == "A":
    cost, loc, a_status = reflex("A", a_status, cost)
    cost, loc, b_status = reflex("B", b_status, cost)
elif loc == "B":
    cost, loc, b_status = reflex("B", b_status, cost)
    cost, loc, a_status = reflex("A", a_status, cost)

# Output the total cost and goal status
print(f"Total cost: {cost}")
goal(a_status, b_status)

```

Output:

```
Enter the starting location of the vacuum (A or B): A
Enter the status of location A (0 for clean, 1 for dirty): 0
Enter the status of location B (0 for clean, 1 for dirty): 0
Move RIGHT to B
Move LEFT to A
Total cost: 0
Goal reached
```

```
Enter the starting location of the vacuum (A or B): B
Enter the status of location A (0 for clean, 1 for dirty): 1
Enter the status of location B (0 for clean, 1 for dirty): 1
SUCK at A
Move RIGHT to B
SUCK at B
Move LEFT to A
Total cost: 2
Goal reached
```

```
Enter the starting location of the vacuum (A or B): A
Enter the status of location A (0 for clean, 1 for dirty): 1
Enter the status of location B (0 for clean, 1 for dirty): 0
SUCK at A
Move RIGHT to B
Move LEFT to A
Total cost: 1
Goal reached
```

```
Enter the starting location of the vacuum (A or B): A
Enter the status of location A (0 for clean, 1 for dirty): 0
Enter the status of location B (0 for clean, 1 for dirty): 1
Move RIGHT to B
SUCK at B
Move LEFT to A
Total cost: 1
Goal reached
```

Program 3

Implement 8 puzzle problems using DFS and BFS:

LAB - 03.

Date _____
Page _____
08/10/24

⇒ write an algorithm and a program
to solve 8-puzzle game.

→ Algorithm to find manhattan distance.

Step 1: A 2D array to represent the current state of the puzzle (3×3)
A 2D array to represent the goal state of the puzzle (3×3)

goal state =

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | |

= $[(1, 2, 3), (4, 5, 6), (7, 8, -)]$

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

Step 2: 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
 end for
 end for.
 return distance.

step 3: checking for the goal state in current state (if current state == goal state)
def check(state):
 return goalstate == current state

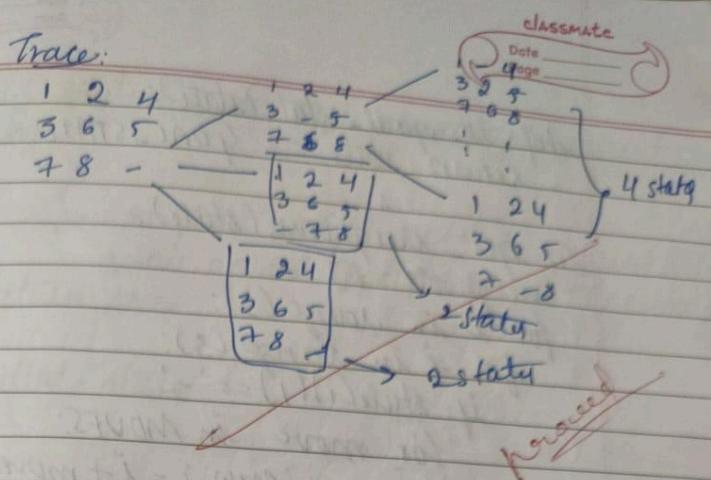
=> O+5 Algorithm.

step 4:

def neighbours(state):
 → Loop the matrix from i to 3
 → where if state[i][j] = '-' →
 then check all the four directions
 using the move matrix, we can do it.
 → add those to a new array.
 → Return the neighbours array

Step 5: DFS.

→ declare a queue
→ visited array to mark visited positions
→ loop it until the queue is empty
→ check if the current state is the goal state.
→ skip already visited state using visited array
→ if not visited - add to the visited array
→ Again get the neighbours
→ end. if no neighbours → none.



⇒ role implementation:

from collections import deque

GOAL-STATE = [[1, 2, 3],
[4, 5, 6],
[7, 8, -]]

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 i in range(3):~~

if state[i][j] != '-'

$\text{goal_i}, \text{goal_j} = \text{climod}(\text{state}[i][j]) - 1, 3$

distance += abs(i - goal_i) + abs(j - goal_j)

Code:

BFS:

```
from collections import deque

# Goal state for the 8-puzzle
goal_state = [[1, 2, 3],
              [4, 5, 6],
              [7, 8, 0]]

# Possible moves
moves = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1)}

# Function to find the position of the blank (0)
def find_blank(state):
    for i in range(len(state)):
        for j in range(len(state[i])):
            if state[i][j] == 0:
                return i, j

# Function to check if a state is the goal state
def is_goal(state):
    return state == goal_state

# Function to generate neighbors by moving the blank tile
def get_neighbors(state):
    neighbors = []
    blank_row, blank_col = find_blank(state)
    for move, (dr, dc) in moves.items():
        new_row, new_col = blank_row + dr, blank_col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            # Create a new state by swapping tiles
            new_state = [row[:] for row in state]
            new_state[blank_row][blank_col], new_state[new_row][new_col] =
            new_state[new_row][new_col], new_state[blank_row][blank_col]
            neighbors.append((new_state, move))
    return neighbors

# Function to print the puzzle state
def print_puzzle(state):
    for row in state:
        print(row)
    print()

# BFS algorithm to solve the puzzle
def bfs(start_state):
    queue = deque([(start_state, [])]) # Queue stores (current state, path to reach it)
```

```

visited = set() # To avoid revisiting states
visited.add(tuple(tuple(row) for row in start_state)) # Convert state to a tuple for
hashing

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

    # Check if the goal is reached
    if is_goal(current_state):
        return current_state, path

    # Explore neighbors
    for neighbor, move in get_neighbors(current_state):
        state_tuple = tuple(tuple(row) for row in neighbor)
        if state_tuple not in visited:
            visited.add(state_tuple)
            queue.append((neighbor, path + [move]))

return None, None

# Function to get user input for the initial state
def get_user_input():
    print("Enter the initial state of the 8-puzzle (row by row):")
    state = []
    for i in range(3):
        while True:
            try:
                row = list(map(int, input(f"Enter row {i+1} (3 integers, space-separated):").split()))
                if len(row) != 3 or any(x not in range(9) for x in row):
                    raise ValueError
                state.append(row)
                break
            except ValueError:
                print("Invalid input. Please enter 3 integers between 0 and 8.")
    return state

# Function to demonstrate the solution step by step
def demonstrate_solution(start_state, solution_path):
    current_state = start_state
    print("Initial state:")
    print_puzzle(current_state)
    for move in solution_path:
        print(f"Move: {move}")
        for neighbor, move_name in get_neighbors(current_state):
            if move_name == move:
                current_state = neighbor

```

```

print_puzzle(current_state)
break

# Main function
if __name__ == "__main__":
    start_state = get_user_input()
    final_state, solution_path = bfs(start_state)
    if solution_path:
        print("Solution found. Steps are demonstrated below:")
        demonstrate_solution(start_state, solution_path)
    else:
        print("No solution found.")

```

Output:

```

Enter the initial state of the 8-puzzle (row by row):
Enter row 1 (3 integers, space-separated): 1 2 3
Enter row 2 (3 integers, space-separated): 0 4 6
Enter row 3 (3 integers, space-separated): 7 5 8
Solution found. Steps are demonstrated below:
Initial state:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]

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

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

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

```

```

Enter the initial state of the 8-puzzle (row by row):
Enter row 1 (3 integers, space-separated): 1 3 2
Enter row 2 (3 integers, space-separated): 4 6 0
Enter row 3 (3 integers, space-separated): 7 5 8
No solution found.

```

DFS:

```
from copy import deepcopy

# Directions for moving the blank space (0): up, down, left, right
DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]

class PuzzleState:
    def __init__(self, board, parent=None, move=""):
        self.board = board
        self.parent = parent
        self.move = move

    # Find the position of the blank (0)
    def get_blank_position(self):
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return i, j

    # Generate successor states by moving the blank space
    def generate_successors(self):
        successors = []
        x, y = self.get_blank_position()
        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 = deepcopy(self.board)
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y], new_board[x][y]
                successors.append(PuzzleState(new_board, parent=self))
        return successors

    # Check if the current state matches the goal state
    def is_goal(self, goal_state):
        return self.board == goal_state

    # String representation of the puzzle state
    def __str__(self):
        return "\n".join([" ".join(map(str, row)) for row in self.board])

    # Depth-limited search (DLS)
    def depth_limited_search(current_state, goal_state, depth):
        if depth == 0 and current_state.is_goal(goal_state):
            return current_state
```

```

if depth > 0:
    for successor in current_state.generate_successors():
        found = depth_limited_search(successor, goal_state, depth - 1)
        if found:
            return found
    return None

# Iterative deepening search (IDS)
def iterative_deepening_search(start_state, goal_state):
    depth = 0
    while True:
        print(f"\nSearching at depth level: {depth}")
        result = depth_limited_search(start_state, goal_state, depth)
        if result:
            return result
        depth += 1

# Get user input for start and goal states
def get_user_input():
    print("Enter the start state (use 0 for the blank):")
    start_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        start_state.append(row)
    print("Enter the goal state (use 0 for the blank):")
    goal_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        goal_state.append(row)
    return start_state, goal_state

# Main function
def main():
    start_board, goal_board = get_user_input()
    start_state = PuzzleState(start_board)
    goal_state = goal_board
    result = iterative_deepening_search(start_state, goal_state)
    if result:
        print("\nGoal reached!")
        path = []
        while result:
            path.append(result)
            result = result.parent
        path.reverse()
        for state in path:
            print(state, "\n")
    else:

```

```
print("Goal state not found.")

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

Output:

```
Enter the start state (use 0 for the blank):
2 8 3
1 6 4
7 0 5
Enter the goal state (use 0 for the blank):
1 2 3
8 0 4
7 6 5

Searching at depth level: 0
Searching at depth level: 1
Searching at depth level: 2
Searching at depth level: 3
Searching at depth level: 4
Searching at depth level: 5

Goal reached!
2 8 3
1 6 4
7 0 5

2 8 3
1 0 4
7 6 5

2 0 3
1 8 4
7 6 5

0 2 3
1 8 4
7 6 5

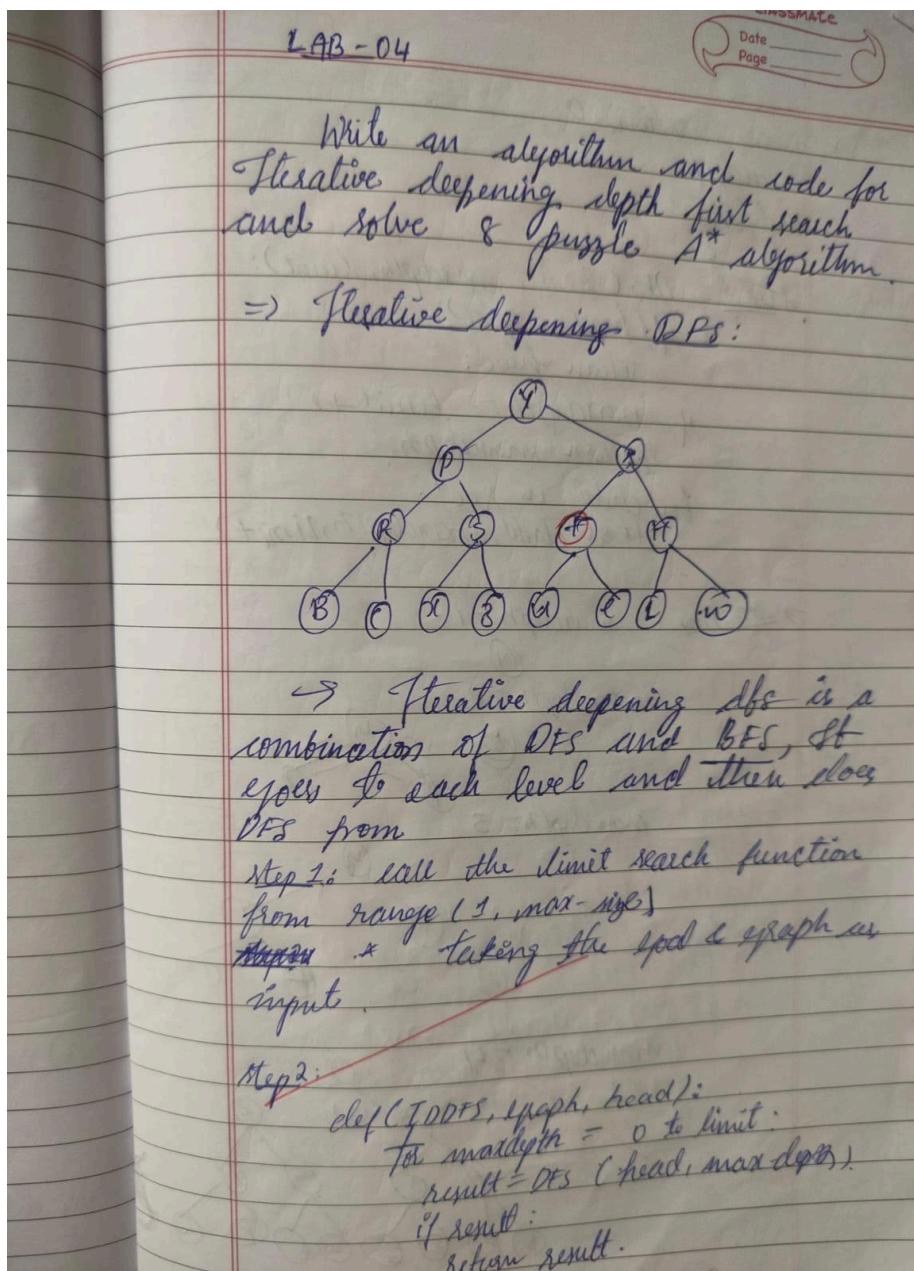
1 2 3
0 8 4
7 6 5

1 2 3
8 0 4
7 6 5
```

Program 4

Implement A* search algorithm

Algorithm:

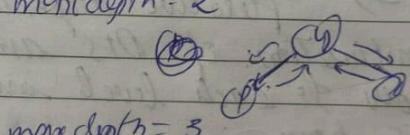


return 0
end if
end for.

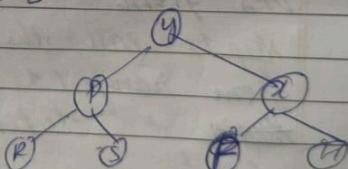
Step 3: DFS(head, maxdepth, limit):
 if (head == goal):
 return head:
 if maxdepth == limit + 1
 return maxdepth.
 for child in root:
 DFS(child, maxdepth, limit).

Iteration: maxdepth = 1

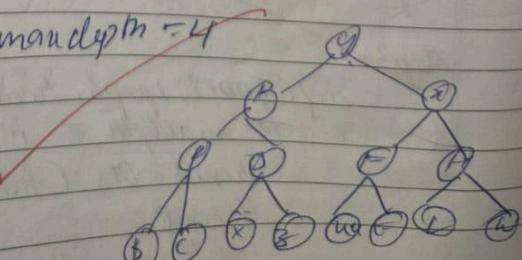
maxdepth = 2



maxdepth = 3



maxdepth = 4



Algorithm for 8 puzzle using A*.

Initial state

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | | 4 |
| 7 | 6 | 5 |

| | | |
|---|---|---|
| 2 | 8 | 1 |
| | 4 | 3 |
| 7 | 6 | 5 |

goal state

| | |
|---|---|
| 1 | 3 |
| 8 | 2 |
| 7 | 6 |

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | 4 | |
| 7 | 6 | 5 |

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | 4 | |
| 7 | 6 | 5 |

| | | |
|---|---|---|
| 4 | 2 | 3 |
| 8 | 6 | 4 |
| 7 | 5 | |

$$h(n) = 8$$

$$f(n) = 9$$

$$h(n) = 6$$

$$f(n) = 7$$

$$h(n) = 6$$

$$f(n) = 7$$

$$h(n) = 8$$

$$f(n) = 9$$

$$h(n) = 7$$

$$f(n) = 9$$

$$h(n) = 7$$

$$f(n) = 9$$

$$h(n) = 5$$

$$f(n) = 7$$

$$h(n) = 7$$

$$f(n) = 9$$

$$h(n) = 9$$

$$f(n) = 9$$

$$h(n) = 7$$

$$f(n) = 7$$

$$h(n) = 5$$

$$f(n) = 7$$

$$h(n) = 7$$

$$f(n) = 9$$



| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | 4 | 3 |
| 7 | 6 | 5 |

$$h(n) = 6$$

$$f(n) = 9$$

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | 1 | 4 |
| 7 | 6 | 5 |

$$h(n) = 4$$

$$f(n) = 9$$

Code:
Misplaced Tiles:

```
import heapq

# Define the goal state as a tuple of tuples
GOAL_STATE = ((1, 2, 3),
               (8, 0, 4),
               (7, 6, 5))

# Heuristic: Count the number of misplaced tiles
def misplaced_tile(state):
    misplaced = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != GOAL_STATE[i][j]:
                misplaced += 1
    return misplaced

# Find the position of the blank tile (0)
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

# Generate neighbors by moving the blank tile
def generate_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)] # Right, Left, Down, Up
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            # Create a new state by swapping tiles
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))
    return neighbors

# Reconstruct the path from the start state to the goal state
def reconstruct_path(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
```

```

path.reverse()
return path

# A* search algorithm
def a_star(start):
    open_list = []
    heapq.heappush(open_list, (0 + misplaced_tile(start), 0, start)) # (f(n), g(n), state)
    g_score = {start: 0} # Cost from start to the current state
    came_from = {}
    visited = set()

    while open_list:
        _, g, current = heapq.heappop(open_list)

        # Check if we have reached the goal
        if current == GOAL_STATE:
            path = reconstruct_path(came_from, current)
            return path, g

        visited.add(current)

        # Explore neighbors
        for neighbor in generate_neighbors(current):
            if neighbor in visited:
                continue

            tentative_g = g_score[current] + 1 # Each move has a cost of 1

            # If this path is better, update scores and add to the queue
            if tentative_g < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score = tentative_g + misplaced_tile(neighbor) # f(n) = g(n) + h(n)
                heapq.heappush(open_list, (f_score, tentative_g, neighbor))

    return None, None # No solution found

# Print a given puzzle state
def print_state(state):
    for row in state:
        print(row)
    print()

# Main function
if __name__ == "__main__":
    start_state = ((2, 8, 3),
                  (1, 6, 4),

```

```

(7, 0, 5))

print("Initial State:")
print_state(start_state)
print("Goal State:")
print_state(GOAL_STATE)

solution, cost = a_star(start_state)
if solution:
    print(f"Solution found with cost: {cost}")
    print("Steps:")
    for step in solution:
        print_state(step)
else:
    print("No solution found.")

```

Output:

```

▶ Initial State:
⇒ (2, 8, 3)
⇒ (1, 6, 4)
⇒ (7, 0, 5)

Goal State:
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

Solution found with cost: 5
Steps:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

(2, 8, 3)
(1, 0, 4)
(7, 6, 5)

(2, 0, 3)
(1, 8, 4)
(7, 6, 5)

(0, 2, 3)
(1, 8, 4)
(7, 6, 5)

(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

```

Manhattan:

```
import heapq

# Define the goal state as a tuple of tuples
GOAL_STATE = ((1, 2, 3),
               (8, 0, 4),
               (7, 6, 5))

# Heuristic: Calculate the Manhattan distance for each tile
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:
                goal_x, goal_y = divmod(value - 1, 3) # Find the goal position of the current tile
                distance += abs(goal_x - i) + abs(goal_y - j)
    return distance

# Find the position of the blank tile (0)
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

# Generate neighbors by moving the blank tile
def generate_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)] # Right, Left, Down, Up
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            # Create a new state by swapping tiles
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))
    return neighbors

# Reconstruct the path from the start state to the goal state
def reconstruct_path(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
    path.append(current)
    return path
```

```

    path.append(current)
    path.reverse()
    return path

# A* search algorithm
def a_star(start):
    open_list = []
    heapq.heappush(open_list, (manhattan_distance(start), 0, start)) # (f(n), g(n), state)
    g_score = {start: 0} # Cost from start to the current state
    came_from = {}
    visited = set()

    while open_list:
        f, g, current = heapq.heappop(open_list)

        # Check if we have reached the goal
        if current == GOAL_STATE:
            path = reconstruct_path(came_from, current)
            return path, g

        visited.add(current)

        # Explore neighbors
        for neighbor in generate_neighbors(current):
            if neighbor in visited:
                continue

            tentative_g = g_score[current] + 1 # Each move has a cost of 1

            # If this path is better, update scores and add to the queue
            if tentative_g < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score = tentative_g + manhattan_distance(neighbor) # f(n) = g(n) + h(n)
                heapq.heappush(open_list, (f_score, tentative_g, neighbor))

    return None, None # No solution found

# Print a given puzzle state
def print_state(state):
    for row in state:
        print(row)
    print()

# Main function
if __name__ == "__main__":
    start_state = ((2, 8, 3),

```

```

(1, 6, 4),
(7, 0, 5))

print("Initial State:")
print_state(start_state)
print("Goal State:")
print_state(GOAL_STATE)

solution, cost = a_star(start_state)
if solution:
    print(f"Solution found with cost: {cost}")
    print("Steps:")
    for step in solution:
        print_state(step)
else:
    print("No solution found.")

```

Output:

```

Initial State:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

Goal State:
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

Solution found with cost: 5
Steps:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

(2, 8, 3)
(1, 0, 4)
(7, 6, 5)

(2, 0, 3)
(1, 8, 4)
(7, 6, 5)

(0, 2, 3)
(1, 8, 4)
(7, 6, 5)

(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:

LAB-05

Simulation of Annealing.

Objective function : $x^2 + 5 \sin x$.

Step 1: Define a function called "Simulation-Annealing".

```
def simulationAnnealing(initialState,
                         initialTemp, coolingRate,
                         current = initialState,
                         best = current,
                         best = objective(current),
                         temp = initialTemp,
                         while temp > 1:
                             for i ← 1 to it:
                                 new = neighbours(current)
                                 current = objective(new)
                                 newCost = object(new)
                                 if function(curr, newCost,
                                             temp) > Rand(0, 1):
                                     current = new
                                 if new < best:
                                     best = new
                                     temp * = cooling
                                     retain(best, best - cost).
```

Step 2: Now define a objective function to change the state

```

def objective(state):
    cost = 0
    for element in state:
        cost += ele * ( + circle)
    return cost

```

Step 3: Next function is to check / search for neighbours

```

def neighbour(state):
    new = state.copy()
    index = rand(0, len(state)-1)
    new[index] += Rand(-1,1)
    return new

```

Step 4: A function for acceptance probability

```

def AP(current_cost, new_cost, temp):
    if (new_cost < current_cost):
        return 1
    else:
        return e^(new_cost - current_cost) / T

```

circle:

```

def main():
    initial_temp = 1000
    cooling_rate = 0.9
    iterations = 1000

```

Don't worry about this

Code:

```

import numpy as np
from scipy.optimize import dual_annealing

```

```
def queens_max(position):
```

This function calculates the number of pairs of queens that are not attacking each other

```

position = np.round(position).astype(int)
n = len(position)
queen_not_attacking = 0

for i in range(n - 1):
    for j in range(i + 1, n):
        # Check if queens i and j are not attacking each other
        if position[i] != position[j] and abs(position[i] - position[j]) != (j - i):
            queen_not_attacking += 1

return -queen_not_attacking # Return negative for maximization

# Bounds for each queen's position (0 to 7 for an 8x8 chessboard)
bounds = [(0, 7) for _ in range(8)]

# Use dual_annealing for simulated annealing optimization
result = dual_annealing(queens_max, bounds)

# Display the results
best_position = np.round(result.x).astype(int)
best_objective = -result.fun

print("The best position found is:", best_position)
print('The number of queens that are not attacking each other is:', best_objective)

```

Output:

→ The best position found is: [3 5 7 2 0 6 4 1]
 The number of queens that are not attacking each other is: 28

Example: SUDOKU PROBLEM

```

import numpy as np
import random
import math

def is_valid(puzzle, row, col, num):
    """Check if a number can be placed at a specific position."""
    if num in puzzle[row] or num in puzzle[:, col]:
        return False
    box_x, box_y = row // 3 * 3, col // 3 * 3
    if num in puzzle[box_x:box_x + 3, box_y:box_y + 3]:
        return False
    return True

```

```

def initial_fill(puzzle):
    """Fill the empty cells in the puzzle with valid random values."""
    filled = puzzle.copy()
    for row in range(9):
        for col in range(9):
            if filled[row][col] == 0:
                possible_values = [num for num in range(1, 10) if is_valid(filled, row, col, num)]
                if possible_values:
                    filled[row][col] = random.choice(possible_values)
    return filled

def objective(puzzle):
    """Calculate the number of conflicts in the Sudoku puzzle."""
    conflicts = 0
    # Row conflicts
    for row in range(9):
        conflicts += 9 - len(set(puzzle[row]))
    # Column conflicts
    for col in range(9):
        conflicts += 9 - len(set(puzzle[:, col]))
    # Box conflicts
    for box_x in range(0, 9, 3):
        for box_y in range(0, 9, 3):
            box = puzzle[box_x:box_x+3, box_y:box_y+3].flatten()
            conflicts += 9 - len(set(box))
    return conflicts

def simulated_annealing(puzzle, max_iter=100000, start_temp=1.0, end_temp=0.01, alpha=0.99):
    """Solve the Sudoku puzzle using simulated annealing."""
    current_state = initial_fill(puzzle)
    current_score = objective(current_state)
    temp = start_temp

    for iteration in range(max_iter):
        if current_score == 0: # Solution found
            break
        # Randomly pick an empty cell
        row, col = random.randint(0, 8), random.randint(0, 8)
        while puzzle[row][col] != 0:
            row, col = random.randint(0, 8), random.randint(0, 8)

        # Create a new state with a random value in the chosen cell
        new_state = current_state.copy()
        new_value = random.randint(1, 9)
        if is_valid(new_state, row, col, new_value):
            new_state[row][col] = new_value

```

```

new_score = objective(new_state)
delta_score = new_score - current_score

# Accept new state based on simulated annealing criteria
if delta_score < 0 or random.uniform(0, 1) < math.exp(-delta_score / temp):
    current_state, current_score = new_state, new_score

# Decrease temperature
temp *= alpha
if temp < end_temp:
    break

return current_state

# Example usage:
puzzle = np.array([
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 9, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [8, 0, 0, 0, 6, 0, 0, 0, 3],
    [4, 0, 0, 8, 0, 3, 0, 0, 1],
    [7, 0, 0, 0, 2, 0, 0, 0, 6],
    [0, 6, 0, 0, 0, 0, 2, 8, 0],
    [0, 0, 0, 4, 1, 9, 0, 0, 5],
    [0, 0, 0, 0, 8, 0, 0, 7, 9]
])
solved_puzzle = simulated_annealing(puzzle)
print("Solved Sudoku:\n", solved_puzzle)

```

Output:

```

Solved Sudoku:
[[5 3 4 2 7 6 9 1 0]
[6 2 7 1 9 5 4 3 8]
[1 9 8 3 4 0 5 6 2]
[8 1 2 7 6 4 0 9 3]
[4 0 9 8 5 3 7 2 1]
[7 0 3 9 2 1 8 5 6]
[3 6 5 0 0 7 2 8 4]
[2 8 0 4 1 9 3 0 5]
[0 4 1 5 8 2 6 7 9]]

```

Program 6

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

29/10/24.

Hill climbing to implement 8-queens.

Step 1: Initialize a random configured 8×8 chessboard with ~~each row~~ ~~one~~ a queen placed in each row - call it start state.

Step 2: calculate all possible conflicts of all the respective states. i.e no two queens should be in same row & same diagonal.

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

Step 3: Initialize current state
→ set current_state to initial_state.
→ calculate current_attacks by counting conflict in current_state.

Function hill_climbing():
 current_state = randomly place queens in each row
 current_attacks = calculate_attacks(current_state)

step 4: Heuristic to find improved state

for i in range(100):

 neighbors = []

 for row in range(8):

 for col in range(8):

 if state[row] != col:

 neighbor = state[i]

 neighbor[row] = col

 neighbors.append(neighbor)

step 5: choose the best neighbor and return result.

current_state = next_state

current_attacks = next_attacks

Return current_state, current_attacks

output:

Best solution found (with 0 attacks) :

.....
..Q....
.....
.....Q..
.....
.....Q.
.....
..Q....

Code:

```
import random
```

```
def get_attacking_pairs(state):
```

```
    """Calculates the number of attacking pairs of queens."""
```

```

attacks = 0
n = len(state)
for i in range(n):
    for j in range(i + 1, n):
        # Check if queens are in the same column or diagonals
        if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
            attacks += 1
return attacks

def generate_successors(state):
    """Generates all possible successors by moving each queen to every other column in
    its row."""
    n = len(state)
    successors = []
    for row in range(n):
        for col in range(n):
            if col != state[row]: # Only generate new states with different columns
                new_state = state[:]
                new_state[row] = col
                successors.append(new_state)
    return successors

def hill_climbing(n):
    """Hill climbing algorithm for N-Queens problem."""
    # Start with a random state
    current = [random.randint(0, n - 1) for _ in range(n)]
    steps = 0

    while True:
        current_attacks = get_attacking_pairs(current)
        successors = generate_successors(current)

        # Find the neighbor with the minimum attacks
        neighbor = min(successors, key=get_attacking_pairs)
        neighbor_attacks = get_attacking_pairs(neighbor)
        steps += 1

        print(f"Step {steps}: Current State: {current}, Attacks: {current_attacks}")

        # If no better neighbor is found, return the current state
        if neighbor_attacks >= current_attacks:
            return current, current_attacks

        # Move to the better neighbor
        current = neighbor

def print_board(state):

```

```

"""Prints the board with queens placed."""
n = len(state)
board = [".." for _ in range(n)] for _ in range(n)]
for row in range(n):
    board[row][state[row]] = "Q"
for row in board:
    print(" ".join(row))
print("\n")

# Set the size of the board
n = 8 # Change this value to test with different board sizes

solution, attacks = hill_climbing(n)
print("Final State (Solution):", solution)
print("Number of Attacking Pairs:", attacks)
print_board(solution)

```

Output:

```

Step 1: Current State: [0, 0, 0, 2], Attacks: 4, Cost: 4
Step 2: Current State: [0, 3, 0, 2], Attacks: 1, Cost: 1
Step 3: Current State: [1, 3, 0, 2], Attacks: 0, Cost: 0
Final State (Solution): [1, 3, 0, 2]
Number of Attacking Pairs: 0
. Q .
. . . Q
Q . . .
. . Q .

```

Implement A-star algorithm to solve N-Queens problem

Algorithm:

⇒ A-star algorithm to implement
8. Queens

Step 1: Initialize a starting state where 8x8 board is empty.

→ Define it where each row is set to -1 i.e., no queens are placed

→ Add this as starting node in a priority queue i.e., priority-set with $f=0$ and $h=\text{number of conflicts}$.

~~Step 2~~

Step 2: Define heuristic function which gives the computation of attacking pairs for given state.

Def 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.

Step 3: search for solution by removing node with lowest f (heuristic function)
if $h = 0$:
return solution.

Function a-star

initial-state = [-1, -1, -1, -1, -1, 1, -1, -1]

priority-set = []

visited = empty set

while priority-set is not empty.

current-node = heap.heap.pop(priority-set)

current-state = current-node.state

If heuristic(current-state) == 0

return current-state

If current-state is in visited.

continue

Add current-state to visited

next-row = index of first empty row
in current-state

if next-row > 8:

for column in range(8):

new-state = list(current-state)

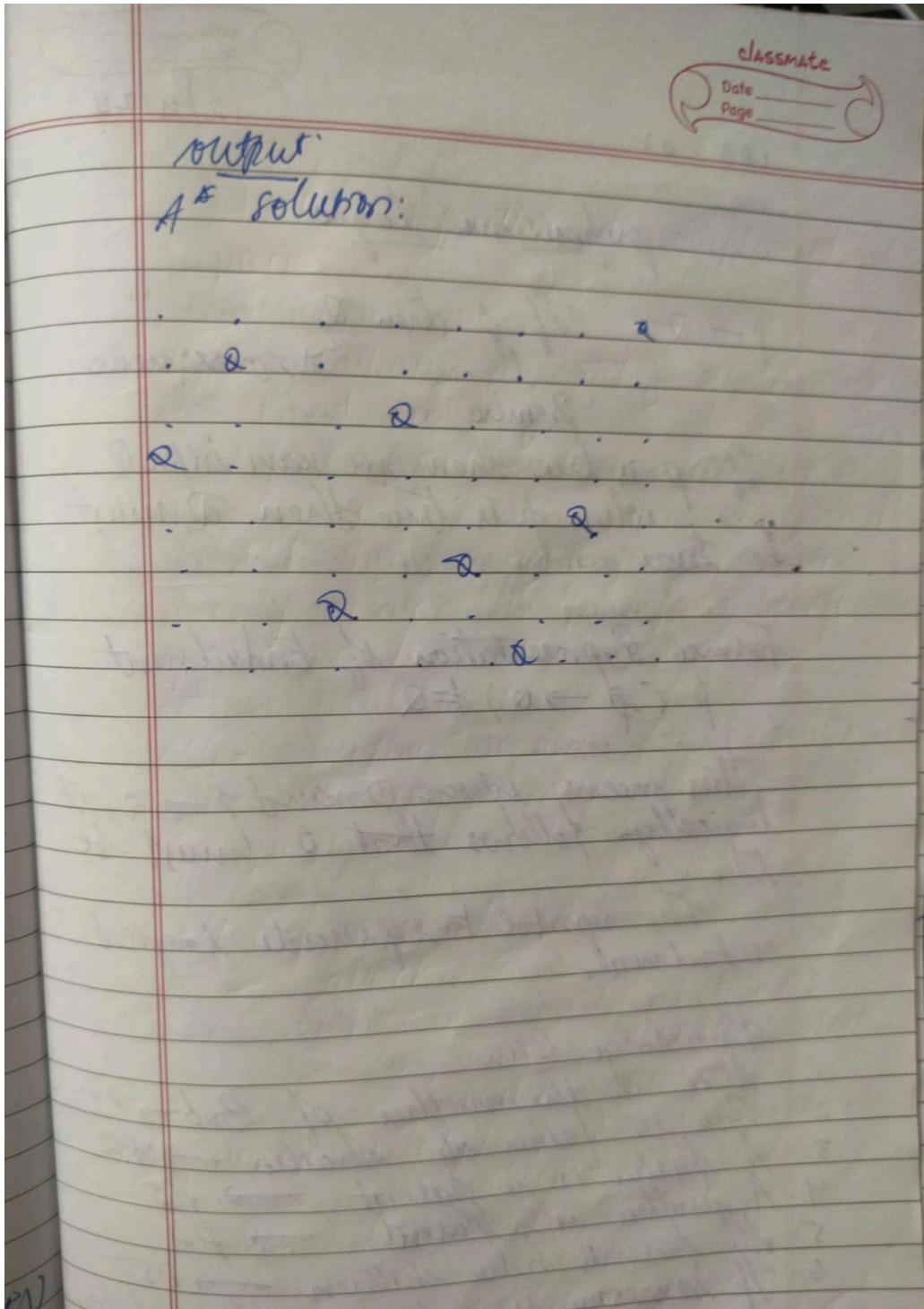
new-state[next-row][column] = 0

new-state = tuple(new-state)

if new-state is not in visited:

h = heuristic(new-state)

return heap.push(priority-set, h, new-state)



code:

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():
    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:

```
Solution board (column positions for each row): [0, 4, 7, 5, 2, 6, 1, 3]
```

Program 7

Implement propositional logic

Algorithm:

LAB - 02 12/11/24

propositional logic.

$p \rightarrow q$ If p then q
 ↑ ↓ consequence
 Premise

If p is true then we can infer q .
when p is true then q must be true.

Formal Representation of Entailment
 $p, (p \rightarrow q) \models q$

This means, given p and $p \rightarrow q$, it logically follows that q must be true.

The symbol \models represents logical entailment.

Knowledge Base:

1. Alice is the mother of Bob $\rightarrow p_1$
2. Bob is father of Charlie $\rightarrow p_2$
3. A father is a parent $\rightarrow p_3$
4. A mother is a parent $\rightarrow p_4$
5. All parents have children $\rightarrow p_5$
6. If someone is a parent, their children are siblings $\rightarrow p_6$
7. Alice is married to David $\rightarrow p_7$

Hypothesis:

- Charlie is a sibling of bob

\Rightarrow Implications:

1. From P1: if Alice is the mother of bob then bob is a child of Alice.
 $P1 \rightarrow Q1$: Bob is a child of Alice

2. From P2: Bob is father of Charlie
 $P2 \rightarrow Q2$: Charlie is a child of Bob

3. From P3 & P4
 $P3 \rightarrow P4$: Bob is a parent and Alice is a parent.

4. From P5:
 $P5 \rightarrow Q3$: Bob & Alice have children

5. From P6:
 ~~$P6 \rightarrow H$: Charlie & Bob are siblings~~

\Rightarrow Conclusion: $P1 \wedge P2 \wedge P3 \wedge P4 \wedge P5 \wedge P6 \rightarrow H$.

Done
10/11/19

code:

```
class KnowledgeBase:
```

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

```
    def add_fact(self, fact):
```

```

    self.facts.append(fact)

def add_rule(self, premise, conclusion):
    self.rules.append((premise, conclusion))

def infer(self):
    new_inferences = True

    while new_inferences:
        new_inferences = False

        for premise, conclusion in self.rules:
            if all(fact in self.facts for fact in premise) and conclusion not in self.facts:
                self.facts.append(conclusion)
                new_inferences = True

def entails(self, hypothesis):
    return hypothesis in self.facts

# Example Usage
kb = KnowledgeBase()

# Adding facts
kb.add_fact("Alice is mother of Bob")
kb.add_fact("Bob is father of Charlie")
kb.add_fact("A father is a parent")
kb.add_fact("A mother is a parent")
kb.add_fact("All parents have children")
kb.add_fact("Alice is married to Davis")

# Adding rules
kb.add_rule(["Bob is father of Charlie", "A father is a parent"], "Bob is parent")
kb.add_rule(["Alice is mother of Bob", "A mother is a parent"], "Alice is parent")
kb.add_rule(["Bob is parent", "All parents have children"], "Charlie and Bob are siblings")

# Perform inference
kb.infer()

# Hypothesis
hypothesis = "Charlie and Bob are siblings"

if kb.entails(hypothesis):
    print(f"The hypothesis '{hypothesis}' is entailed by the knowledge base.")
else:
    print(f"The hypothesis '{hypothesis}' is not entailed by the knowledge base.")

```

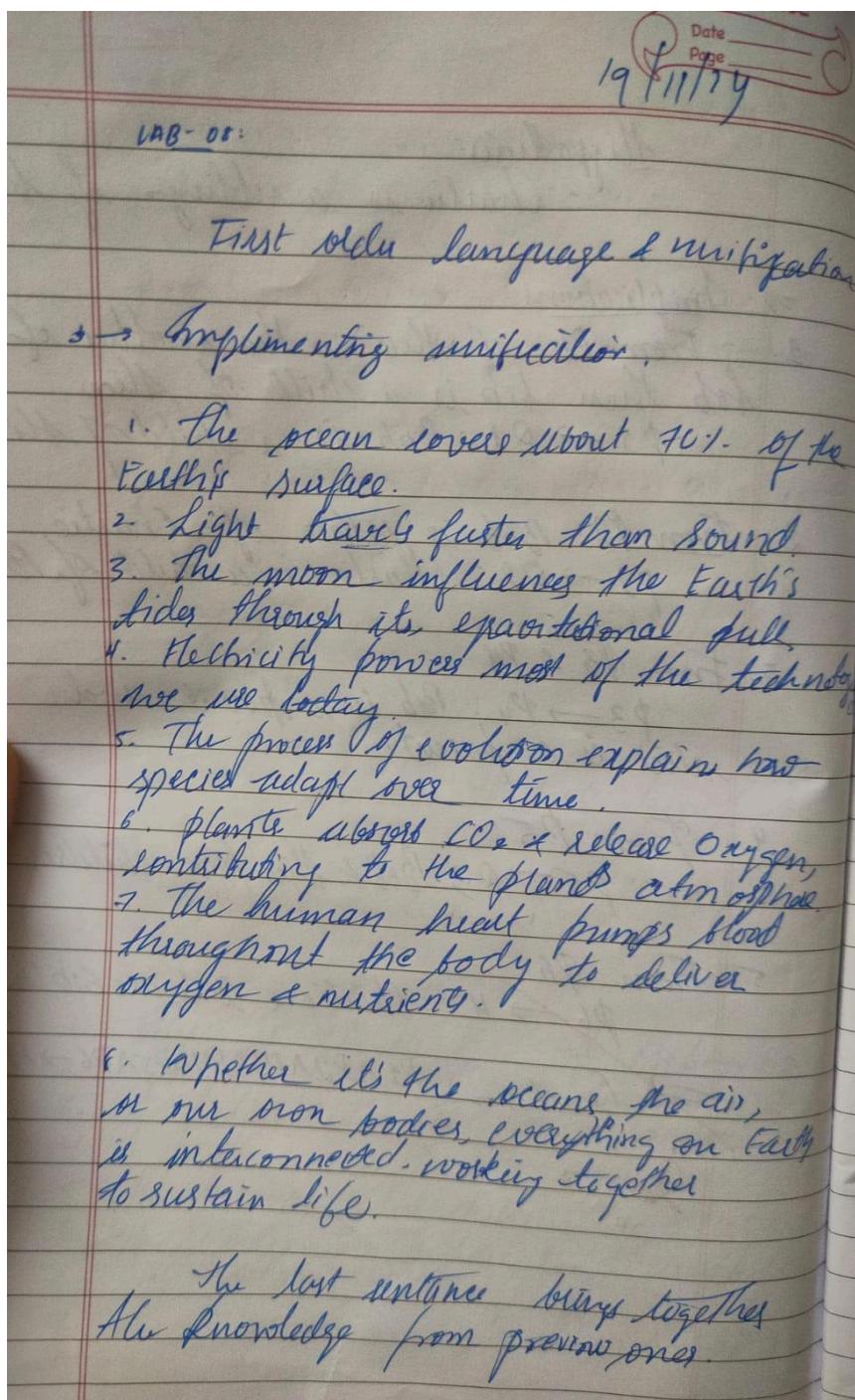
output:

| Output | Clear |
|--|-------|
| The hypothesis 'Charlie and Bob are siblings' is entailed by the knowledge base. | |

Program 8

Implement unification in first order logic

Algorithm:



1. $\rightarrow \forall x (\text{ocean}(x) \rightarrow \text{covers}(x, \text{Earth})) \wedge \forall y (\text{Marine Life}(y) \rightarrow \text{sustains}(x, y))$
2. $\rightarrow \forall x \forall y (\text{Lightning}(x) \wedge \text{Thunder}(y) \rightarrow (\text{speed}(x) > \text{speed}(y)))$
3. $\rightarrow \forall x \forall y (\text{Moon}(x) \wedge \text{Tides}(y) \rightarrow \text{Influence}(x, y))$
4. $\rightarrow \exists x (\text{Electricity}(x) \wedge \forall y (\text{Technology}(y) \rightarrow \text{powers}(x, y)))$
5. $\rightarrow \forall x (\text{Species}(x) \exists y (\text{adaptation}(y) \wedge \text{leadsTo}(x, y)))$
6. $\rightarrow \forall x (\text{plants}(x) \rightarrow \exists y (\text{photosynthesis}(y) \wedge \text{contributesTo}(y, \text{oxygen}) \wedge \text{regulates}(y, \text{atmosphere})))$
7. $\rightarrow \forall x (\text{heat}(x) \rightarrow \forall y (\text{organism}(y) \rightarrow (\text{circulates}(x, y) \wedge \text{delivers}(x, y, \text{oxygen}))))$

8. $\rightarrow \text{unified}$

$\rightarrow \forall x \forall y \forall z (\text{Earth}(x) \wedge \text{air}(y) \wedge \text{organisms}(z) \rightarrow (\text{interconnected}(x, y, z) \wedge \text{worksTogether}(x, y, z)))$

$\rightarrow \text{ocean}$

$\rightarrow \text{organisms}$ *Dom
idulim*

2

Code:

```
class UnificationError(Exception):
    """Custom exception for unification errors."""
    pass
```

```
def occurs_check(var, term, subst):
```

```

"""Check if `var` occurs in `term` (to prevent circular substitutions)."""
if var == term:
    return True
elif isinstance(term, (list, tuple)):
    return any(occurs_check(var, t, subst) for t in term)
elif isinstance(term, str) and term in subst:
    return occurs_check(var, subst[term], subst)
return False

def is_variable(term):
    """Check if `term` is a variable (starting with '?')."""
    return isinstance(term, str) and term.startswith('?')

def unify(psi1, psi2, subst=None):
    """Attempt to unify two terms, `psi1` and `psi2`, under the given substitution."""
    if subst is None:
        subst = {}
    if psi1 == psi2:
        return subst
    elif is_variable(psi1):
        if psi1 in subst:
            return unify(subst[psi1], psi2, subst)
        elif occurs_check(psi1, psi2, subst):
            raise UnificationError(f"Occurs check failed: {psi1} in {psi2}")
        else:
            subst[psi1] = psi2
            return subst
    elif is_variable(psi2):
        if psi2 in subst:
            return unify(psi1, subst[psi2], subst)
        elif occurs_check(psi2, psi1, subst):
            raise UnificationError(f"Occurs check failed: {psi2} in {psi1}")
        else:
            subst[psi2] = psi1
            return subst
    elif isinstance(psi1, list) and isinstance(psi2, list):
        if psi1[0] != psi2[0]:
            raise UnificationError(f"Predicate symbols don't match: {psi1[0]} != {psi2[0]}")
        if len(psi1) != len(psi2):
            raise UnificationError(f"Argument lengths don't match: {len(psi1)} != {len(psi2)}")
        for arg1, arg2 in zip(psi1[1:], psi2[1:]): # Skip the predicate symbol (first element)
            subst = unify(arg1, arg2, subst)
        return subst
    else:
        raise UnificationError(f"Cannot unify {psi1} with {psi2}")

```

```

def get_input():
    """Get input from the user and perform unification."""
    try:
        term1 = eval(input("Enter the first term (e.g., ['P', 'b', 'x', ['f', ['g', 'z']]]): "))
        term2 = eval(input("Enter the second term (e.g., ['P', 'z', ['f', 'y'], ['f', 'y']]): "))
        substitution = unify(term1, term2)
        print("Unification successful!")
        print("Substitution:", substitution)
    except UnificationError as e:
        print("Unification failed:", e)
    except Exception as e:
        print("Invalid input or error:", e)

# Run the unification input prompt
get_input()

```

Output:

Enter the first term (e.g., ['P', 'b', 'x', ['f', ['g', 'z']]]): ['P', ['f', ['a']], ['g', ['?y']]]
Enter the second term (e.g., ['P', 'z', ['f', 'y'], ['f', 'y']]): ['P', '?x', '?x']
Unification failed: Predicate symbols don't match: g != f

+ Code + Text

Enter the first term (e.g., ['P', 'b', 'x', ['f', ['g', 'z']]]): ['P', 'b', '?x', ['f', ['g', '?z']]]
Enter the second term (e.g., ['P', 'z', ['f', 'y'], ['f', 'y']]): ['P', '?z', ['f', '?y'], ['f', '?y']]
Unification successful!
Substitution: {'?z': 'b', '?x': ['f', 'y'], '?y': ['g', '?z']}

Program 9

Forward Chaining:

Algorithm:

03/12/2024 Page
LAB - 09

1) Using first order logic - prove the given query using forward chaining or forward reasoning.

→ It starts with a knowledge base. To that base we put logic and using the inference rules we extract new information from the existing base.

→ Using this forward chaining method we will extract the goal step by step given query.

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

To prove: Robert is criminal.

Representation in FOL

It is a crime for an American to sell weapons to hostile nations

American - p $\exists p \text{ American}$
Weapons - v $\exists v \text{ Weapons}$
Hostile - r $\exists r \text{ Hostile}$

$\exists p \text{ American} \wedge \exists v \text{ Weapons} \wedge \exists r \text{ Hostile} \wedge p \text{ sells}(p, v, r) \Rightarrow \text{Criminal}(p)$

2 → Country A has some missiles.
 $\exists x \text{Owns}(A, x) \wedge \text{Missile}(x)$

anew Constant T1:
 $\text{Owns}(A, T1)$
 $\text{Missile}(T1)$

3 → All of the missiles were sold to
Country by Robert.

$\forall x \text{Missile}(x) \wedge \text{Owns}(A, x) \Rightarrow \text{Sells}(\text{Robert}, x, A)$

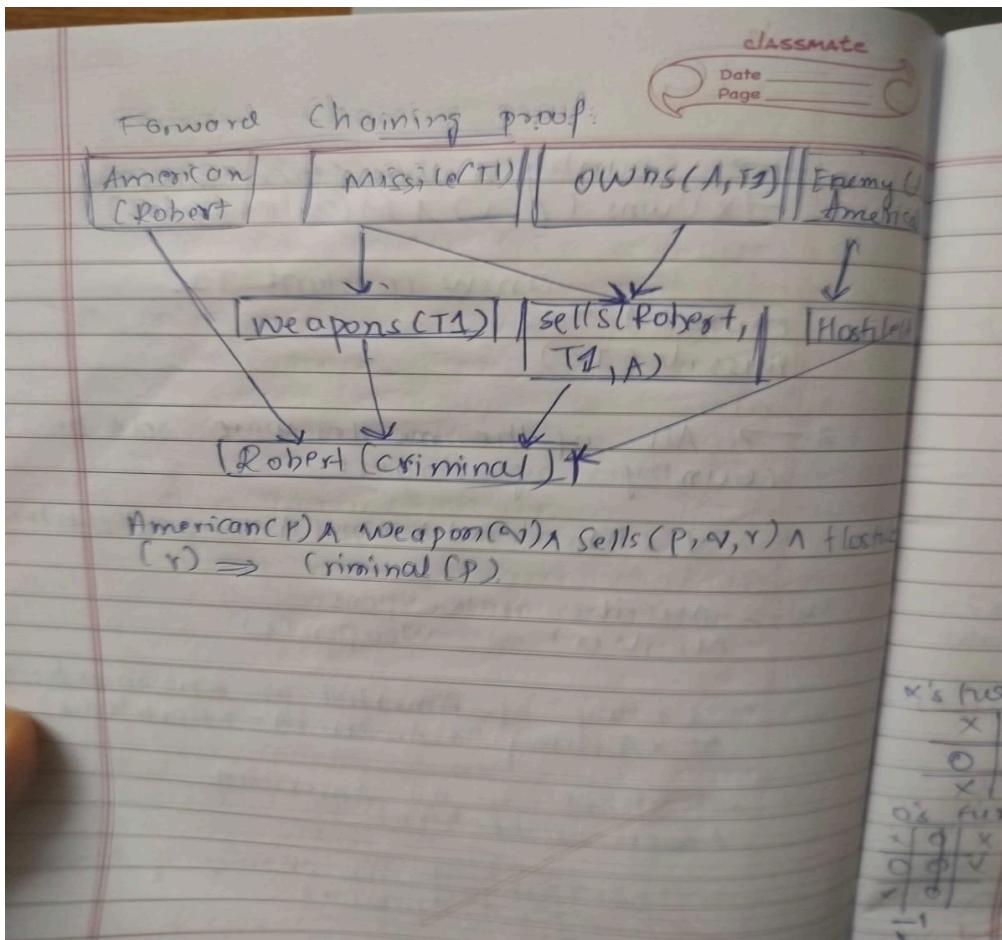
4 → Missiles are weapons.
 $\text{Missile}(x) \Rightarrow \text{Weapon}(x)$

5 → Enemy of America is known as hostile
 $\forall x \text{Enemy}(x, \text{America}) \Rightarrow \text{Hostile}(x)$

6 → Robert is an American
American (Robert)

7 → The country A, an enemy of America
Enemy (A, America).

To prove:
Robert is Criminal
Criminal (Robert).



Code:

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)
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:

Output

```

Inferred: Weapon(T1)
Inferred: Sells(Robert, T1, A)
Inferred: Hostile(A)
Inferred: Criminal(Robert)
Robert is a criminal!

```

Program 10:

Alpha Beta Pruning(Tic-Tac-Toe):

Algorithm:

(AB TD :-)

→ Solving minimax Tic-Tac-Toe game using minimax ~~with~~ alpha beta pruning.

In this pruning happens when we can confidently say that a given path will not affect the final result (because the opponent will avoid it)

Example:-

X's turn :-

| | | |
|---|---|---|
| X | 0 | X |
| 0 | X | |
| 0 | | |

O's turn :-

| | | |
|---|---|---|
| X | 0 | X |
| 0 | X | |
| X | 0 | X |

| | | |
|---|---|---|
| X | 0 | X |
| 0 | X | |
| X | 0 | X |

| | | |
|---|---|---|
| X | 0 | X |
| 0 | X | |
| X | 0 | X |

* X

Y 0.

Code:

```
import math
```

```
def minimax(node, depth, is_maximizing):
    """
```

Implement the Minimax algorithm to solve the decision tree.

Parameters:

node (dict): The current node in the decision tree, with the following structure:

```
{  
    'value': int,  
    'left': dict or None,  
    'right': dict or None  
}
```

depth (int): The current depth in the decision tree.

is_maximizing (bool): Flag to indicate whether the current player is the maximizing player.

Returns:

int: The utility value of the current node.

"""

Base case: Leaf node

if node['left'] is None and node['right'] is None:

```
    return node['value']
```

Recursive case

if is_maximizing:

```
    best_value = -math.inf
```

if node['left']:

```
    best_value = max(best_value, minimax(node['left'], depth + 1, False))
```

if node['right']:

```
    best_value = max(best_value, minimax(node['right'], depth + 1, False))
```

```
    return best_value
```

else:

```
    best_value = math.inf
```

if node['left']:

```
    best_value = min(best_value, minimax(node['left'], depth + 1, True))
```

if node['right']:

```
    best_value = min(best_value, minimax(node['right'], depth + 1, True))
```

```
return best_value
```

Example usage

decision_tree = {

```
    'value': 5,
```

```
    'left': {
```

```
        'value': 6,
```

```
        'left': {
```

```
            'value': 7,
```

```
            'left': {
```

```
                'value': 4,
```

```
                'left': None,
```

```
                'right': None
```

```
            },
```

```

    'right': {
      'value': 5,
      'left': None,
      'right': None
    }
  },
  'right': {
    'value': 3,
    'left': {
      'value': 6,
      'left': None,
      'right': None
    },
    'right': {
      'value': 9,
      'left': None,
      'right': None
    }
  }
},
'right': {
  'value': 8,
  'left': {
    'value': 7,
    'left': {
      'value': 6,
      'left': None,
      'right': None
    },
    'right': {
      'value': 9,
      'left': None,
      'right': None
    }
  },
  'right': {
    'value': 8,
    'left': {
      'value': 6,
      'left': None,
      'right': None
    },
    'right': None
  }
}

```

```
# Find the best move for the maximizing player
best_value = minimax(decision_tree, 0, True)
print(f"The best value for the maximizing player is: {best_value}")
```

Output:

→ The best value for the maximizing player is: 6

Alpha Beta Pruning(8-queens):

Algorithm:

Algorithm for 8-Queens :

```
def alphabeta(board, row, alpha, beta):
    if row == 8:
        if is_valid(board):
            return True
        else:
            return False
    # try placing a queen in each col of current row
    for col in range(8):
        if is_safe(board, row, col):
            board[row] = col
            if alpha >= beta:
                break
            if alphabeta(board, row+1, alpha, beta):
                return True
            board[row] = -1
    return False

conflict checking :-
```

~~if~~

```
def is_safe(board, row, col):
    for r in range(row):
        c = board[r]
        if c == col or abs(c - col) == abs(r - row):
            return False
    return True
```

Date _____
Page _____

```

def is_valid(board):
    return True

def solve_8queens():
    board = [-1] * 8
    alpha = -float('inf')
    beta = float('inf')

    if alpha_beta(board, 0, alpha, beta):
        print("solution found.")
        print(board)
    else:
        print("no sol found")

solution:-
    . Q . . . . .
    . . . . Q . . .
    . . . . . . Q .
    Q . . . . . .
    . . Q . . . .
    . . . . . . Q .
    . . . . Q . .
    . . . . . . .
  
```

code:

```

def is_valid(board, row, col):

    for i in range(row):
        if board[i] == col or \
           abs(board[i] - col) == abs(i - row):
            return False
    return True

def alpha_beta(board, row, alpha, beta, isMaximizing):
  
```

```

if row == len(board):
    return 1

if isMaximizing:
    max_score = 0
    for col in range(len(board)):
        if is_valid(board, row, col):
            board[row] = col
            max_score += alpha_beta(board, row + 1, alpha, beta, False)
            board[row] = -1
            alpha = max(alpha, max_score)
            if beta <= alpha:
                break
    return max_score
else:
    min_score = float('inf')
    for col in range(len(board)):
        if is_valid(board, row, col):
            board[row] = col
            min_score = min(min_score, alpha_beta(board, row + 1, alpha, beta, True))
            board[row] = -1
            beta = min(beta, min_score)
            if beta <= alpha:
                break
    return min_score

def solve_8_queens():

    board = [-1] * 8
    alpha = -float('inf')
    beta = float('inf')
    return alpha_beta(board, 0, alpha, beta, True)

solutions = solve_8_queens()
print(f'Number of solutions for the 8 Queens problem: {solutions}')

```

output:

| Output |
|---|
| Number of solutions for the 8 Queens problem: 6 |