VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Santosh B (1BM22CS243)

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

<u>CERTIFICATE</u>

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Santosh B (1BM22CS243),** 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.

Seema Patil Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/Sanjeet-108/AI_Lab

Program 1

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:

A I

24/09/24

(1)

Implement a tic tak toe game

Algorithm

1. Step 1: Get player Names
   Ask for the name of two players

Step 2: Intialize the board
   create a 3x3 board (eg. using a numpy array) and fill it with "-".

Step 3: Game loop
   Start a loop that continues until game ends

Step 4: Display the board
   show current state of the board to the players

Step 5: check for Draw
   determine if the board is full (no "-" left). If so, declare a draw and end the

Step 6: player turn
   Indicate whose turn it is (player 1 or player 2)

Step 7: Get user input
   Ask the current player to enter their move (row and column number).

Step 8. validate Move.
   Ensure the chosen cell is empty ("-"). If not, ask for input again.

Step 9: update board
   place ~~the~~ the players symbol ('x' or 'O') for chosen cell

Step 10: see if the current player has ~~won~~ won by checking rows columns and diagonals. If there's a winner, declare the winner and end the game. otherwise, switch to the other players turn.

Lab 2

01/10/24

Vacuum cleaner:-

Step 1: Input initial states:

- Take user input for the initial status of rooms A and B. {0
- Take user input for the current location of the vacuum

Step 2: Base condition (Goal state check):

- check if both status_A and status_B are 0 (i.e both rooms are clean).
- If true, print("Goal reached") and stop program (return).

Step 3: check current location

- if the vacuum is in room A (curr_loc == 'A'), move to step 4.
- If the vacuum in room B (curr_loc == 'B') ~~move~~ to. step 7.

Step 4:- Room A dirty:

- If status_A == 1 print(" room is dirty, such operation done.
- Increment the cost of the operation.
- Ask the user to re-enter the updated status of room A after cleaning

Step 5:- check room B's status

- If room B's status is unknown, prompt the user to input the current status of room B.

Step 6: check Room A's new status:

- If room A is still dirty, recursively call the function with room A as the current location if room A is clean, print "moving right" and recursively call the function with room B as the current location.

7 Room B dirty
- if status ==1 , clean
- increment cost
- Ask user

8 check room A's status;

9 check room B new status;
- perform same operations

10 Repeat until goal is reached

1/10/24

Code:
(Tic –Tac –Toe)

```
import numpy as np

board=np.array([['-','-','-'],['-','-','-'],['-','-','-']])

current_player='X'
flag=0

def check_win():
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != '-':
            return True
    for i in range(3):
```

```
        if board[0][i] == board[1][i] == board[2][i] != '-':
            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 tic_tac_toe():
    n=0
    print(board)
    while n<9:
        if n%2==0:
            current_player='X'
        else :
            current_player='O'

        row = int(input("Enter row: "))
        col = int(input("Enter column: "))

        if(board[row][col]=='-'):
            board[row][col]=current_player
            print(board)
            flag=check_win();
            if flag==1:
                print(current_player+' wins')
                break
            else:
                n=n+1
        else :
            print("Invalid Position")

    if n==9:
        print("Draw")

tic_tac_toe()2
```

```
[['-' '-' '-']
 ['-' '-' '-']
 ['-' '-' '-']]
Enter row: 0
Enter column: 1
[['-' 'X' '-']
 ['-' '-' '-']
 ['-' '-' '-']]
Enter row: 0
Enter column: 0
[['O' 'X' '-']
 ['-' '-' '-']
 ['-' '-' '-']]
Enter row: 1
Enter column: 0
[['O' 'X' '-']
 ['X' '-' '-']
 ['-' '-' '-']]
Enter row: 1
Enter column: 1
[['O' 'X' '-']
 ['X' 'O' '-']
 ['-' '-' '-']]
Enter row: 1
Enter column: 2
[['O' 'X' '-']
 ['X' 'O' 'X']
 ['-' '-' '-']]
Enter row: 2
Enter column: 2
[['O' 'X' '-']
 ['X' 'O' 'X']
 ['-' '-' 'O']]
O wins
```

(vacuum cleaner agent)

```
cost =0
def vacuum_world(state, location):
  global cost
  if(state['A']==0 and state['B']==0):
    print('All rooms are clean')
    return

  if state[location]==1:
    state[location]=0
    cost+=1
    state[location]=(int(input('Is room '+ str(location) +' still dirty : ')))

    if state[location]==1:
      return vacuum_world(state, location)
```

```
    else:
      print('Room ' + str(location) + ' cleaned')

  next_location='B' if location=='A' else 'A'
  if state[next_location]==0:
    state[next_location]=(int(input('Is room '+ str(next_location) +' dirty : ')))
  print('Moving to room '+str(next_location))
  return vacuum_world(state, next_location)

state={}
state['A']=int(input('Enter status of room A : '))
state['B']=int(input('Enter status of room B : '))
location=input('Enter initial location of vacuum (A/B) : ')
vacuum_world(state,location)
print("Status = "+str(state))
print('Total cost: ' + str(cost))
```

```
Enter status of room A : 1
Enter status of room B : 1
Enter initial location of vacuum (A/B) : A
Is room A still dirty : 0
Room A cleaned
Moving to room B
Is room B still dirty : 0
Room B cleaned
Is room A dirty : 0
Moving to room A
All rooms are clean
Status = {'A': 0, 'B': 0}
Total cost: 2
```

Program 2
Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm
Algorithm:

Lab ②

08/10/24

Solve 8 puzzle problem using DFS and BFS → Lab ②

DFS:-

→Algorithm:-

1. Define the goal state
   → specify the goal configuration of the 8-puzzle
2. Initialize the initial state
   → Define the initial configuration of the 8 puzzle
3. Locate the Empty space
   → Identify the position of the empty space (blank) in the initial state.
4. create a visited matrix.
   → Initialize a visited matrix to keep track of visited states
5. Define movement directions.
   → set up arrays for row and column changes
6. Implement the recursive helper function
   → create a function helper (goal, in_array, row, col, vis) that
   - Marks the current position as visited
   - Prints the current state
   - Checks if the current state matches the goal state.
   - Explores all possible moves (up, down, left, right)
7. check bounds and visited states
   → In the helper function, for each possible move, check if the new position is within the bounds of the puzzle & if it has not been visited.

8. Make a move:

→ swap the empty space with the adjacent tile when making a move. print the move taken (U, D, R, L).

9. Backtrack if necessary:

→ If the move does not lead a solution, swap back the tiles and mark current position as unvisited

10. Initiate the search

→ call the helper function with the initial state and the position of the empty space to start the search.

Code:
(8 puzzle problems using Depth First Search (DFS))

```
cnt = 0;
def print_state(in_array):
global cnt
cnt += 1
for row in in_array:
print(' '.join(str(num) for num in row))
print() # Print a blank line for better readability


def helper(goal, in_array, row, col, vis):
# Mark the current position as visited
vis[row][col] = 1
drow = [-1, 0, 1, 0] # Directions for row movements: up, right, down, left
```

```
dcol = [0, 1, 0, -1] # Directions for column movements
dchange = ['U', 'R', 'D', 'L']


# Print the current state
print("Current state:")
print_state(in_array)


# Check if the current state is the goal state
if in_array == goal:
print_state(in_array)
print(f"Number of states : {cnt}")
return True


# Explore all possible directions
for i in range(4):
nrow = row + drow[i]
ncol = col + dcol[i]


# Check if the new position is within bounds and not visited
if 0 <= nrow < len(in_array) and 0 <= ncol < len(in_array[0]) and not vis[nrow][ncol]:
# Make the move (swap the empty space with the adjacent tile)
print(f"Took a {dchange[i]} move")
in_array[row][col], in_array[nrow][ncol] = in_array[nrow][ncol], in_array[row][col]


# Recursive call
if helper(goal, in_array, nrow, ncol, vis):
return True


# Backtrack (undo the move)
in_array[row][col], in_array[nrow][ncol] = in_array[nrow][ncol], in_array[row][col]
# Mark the position as unvisited before
returning vis[row][col] = 0
return False

# Example usage
initial_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]] # 0 represents the empty space
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
visited = [[0] * 3 for _ in range(3)] # 3x3 visited matrix
empty_row, empty_col = 1, 0 # Initial position of the
empty space
found_solution = helper(goal_state, initial_state, empty_row, empty_col,
```

visited) print("Solution found:", found_solution)

```
Took a L move
Current state:
1 2 3
4 6 8
0 7 5

Took a D move
Current state:
1 2 3
4 5 6
7 0 8

Took a R move
Current state:
1 2 3
4 5 6
7 8 0

1 2 3
4 5 6
7 8 0

Number of states : 42
Solution found: True
```

(Iterative deepening search algorithm)

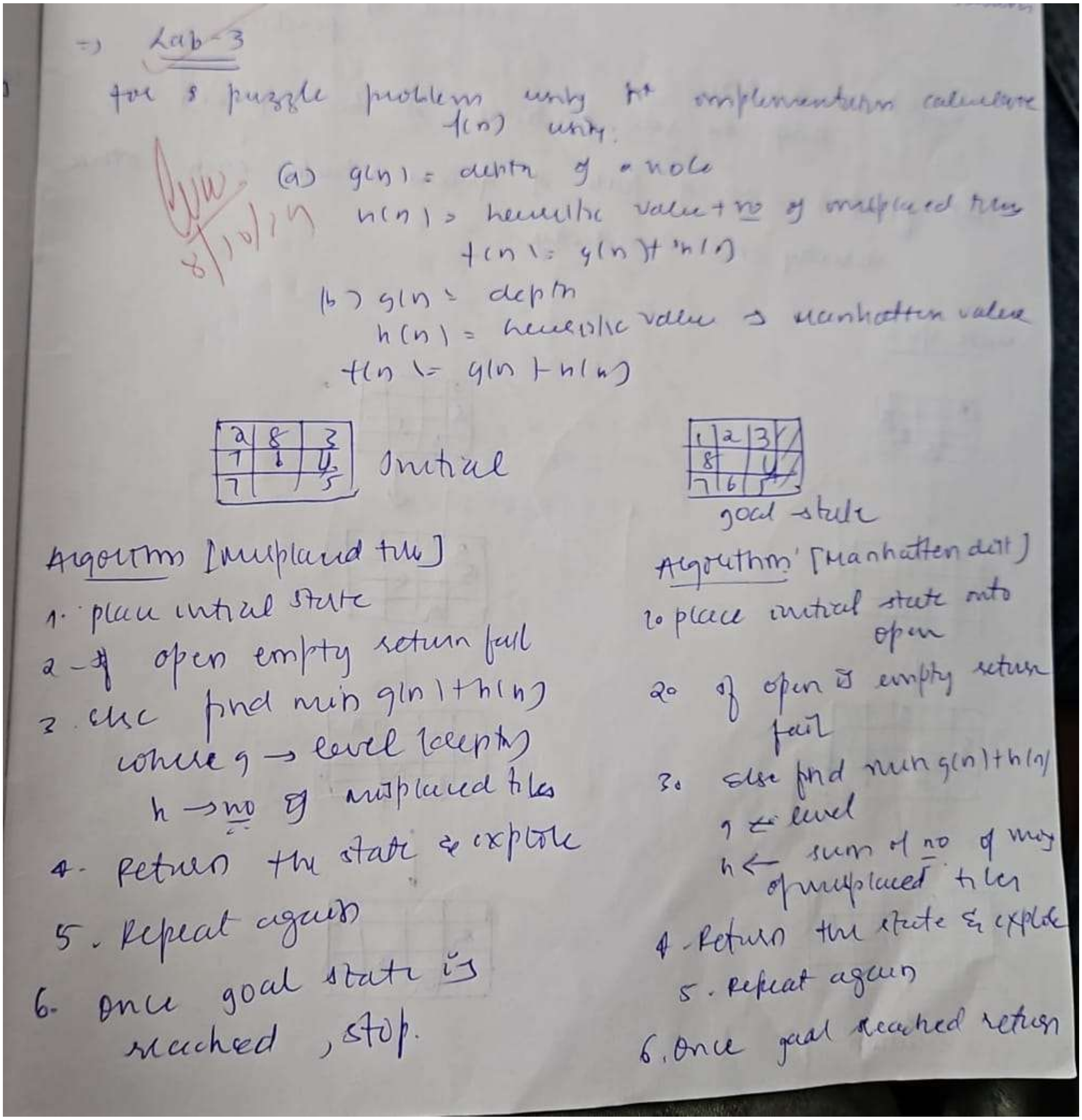
```
class PuzzleState:
    def __init__(self, board, empty_tile_pos, depth=0, path=[]):
        self.board = board
        self.empty_tile_pos = empty_tile_pos  # (row, col)
        self.depth = depth
        self.path = path  # Keep track of the path taken to reach this state

    def is_goal(self, goal):
        return self.board == goal

    def generate_moves(self):
        row, col = self.empty_tile_pos
        moves = []
        directions = [(-1, 0, 'Up'), (1, 0, 'Down'), (0, -1, 'Left'), (0, 1, 'Right')]  # up, down, left, right
        for dr, dc, move_name in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_board = self.board[:]
                new_board[row * 3 + col], new_board[new_row * 3 + new_col] = new_board[new_row *
3 + new_col], new_board[row * 3 + col]
                new_path = self.path + [move_name]  # Update the path with the new move
                moves.append(PuzzleState(new_board, (new_row, new_col), self.depth + 1, new_path))
        return moves

    def display(self):
        # Display the board in a matrix form
        for i in range(0, 9, 3):
            print(self.board[i:i + 3])
        print(f"Moves: {self.path}")  # Display the moves taken to reach this state
        print()  # Newline for better readability

def iddfs(initial_state, goal, max_depth):
    for depth in range(max_depth + 1):
        print(f"Searching at depth: {depth}")
        found = dls(initial_state, goal, depth)
        if found:
            print(f"Goal found at depth: {found.depth}")
            found.display()
            return found
    print("Goal not found within max depth.")
    return None

def dls(state, goal, depth):
```

```
    if state.is_goal(goal):
        return state

    if depth <= 0:
        return None

    for move in state.generate_moves():
        print("Current state:")
        move.display()  # Display the current state
        result = dls(move, goal, depth - 1)
        if result is not None:
            return result
    return None

def main():
    # User input for initial state, goal state, and maximum depth
    initial_state_input = input("Enter initial state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8
0'): ")
    goal_state_input = input("Enter goal state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8
0'): ")
    max_depth = int(input("Enter maximum depth: "))

    initial_board = list(map(int, initial_state_input.split()))
    goal_board = list(map(int, goal_state_input.split()))
    empty_tile_pos = initial_board.index(0) // 3, initial_board.index(0) % 3  # Calculate the position of
the empty tile

    initial_state = PuzzleState(initial_board, empty_tile_pos)

    solution = iddfs(initial_state, goal_board, max_depth)

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

```
Enter initial state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0'): 1 2 3 0 4 6 7 5 8
Enter goal state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0'): 1 2 3 4 5 6 7 8 0
Enter maximum depth: 2
Searching at depth: 0
Searching at depth: 1
Current state:
[0, 2, 3]
[1, 4, 6]
[7, 5, 8]
Moves: ['Up']

Current state:
[1, 2, 3]
[7, 4, 6]
[0, 5, 8]
Moves: ['Down']

Current state:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
Moves: ['Right']

Searching at depth: 2
Current state:
[0, 2, 3]
[1, 4, 6]
[7, 5, 8]
Moves: ['Up']


Current state:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
Moves: ['Right']

Current state:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]
Moves: ['Right', 'Up']

Current state:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
Moves: ['Right', 'Down']

Current state:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
Moves: ['Right', 'Left']

Current state:
[1, 2, 3]
[4, 6, 0]
[7, 5, 8]
Moves: ['Right', 'Right']

Goal not found within max depth.
```

Program 3

Implement A* search algorithm Algorithm:



=> Lab-3

for 8 puzzle problem using A* implementation calculate f(n) using:

(a) g(n) = depth of a node
h(n) = heuristic value + no of misplaced tiles
f(n) = g(n) + h(n)

(b) g(n) = depth
h(n) = heuristic value → manhattan value
f(n) = g(n) + h(n)

2	8	3
1	6	4
7		5

Initial

1	2	3
8		4
7	6	5

goal state

Algorithm [misplaced tile]

1. place initial state
2. If open empty return fail
3. else find min g(n)+h(n)
   where g → level (depth)
   h → no of misplaced tiles
4. Return the state & explore
5. Repeat again
6. Once goal state is reached, stop.

Algorithm [Manhattan dist]

1. place initial state onto open
2. if open is empty return fail
3. else find min g(n)+h(n)
   g ← level
   h ← sum of no of moves of misplaced tiles
4. Return the state & explore
5. Repeat again
6. Once goal reached return

Code:

```
class Node:
    def __init__(self, state, parent=None, move=None, cost=0):
        self.state = state
        self.parent = parent
        self.move = move
        self.cost = cost

    def heuristic(self):
        goal_state = [[1,2,3], [8,0,4], [7,6,5]]
        count = 0
        for i in range(len(self.state)):
            for j in range(len(self.state[i])):
                if self.state[i][j] != 0 and self.state[i][j] != goal_state[i][j]:
                    count += 1
        return count

def get_blank_position(state):
    for i in range(len(state)):
        for j in range(len(state[i])):
            if state[i][j] == 0:
                return i, j

def get_possible_moves(position):
    x, y = position
    moves = []
    if x > 0: moves.append((x - 1, y, 'Down'))
    if x < 2: moves.append((x + 1, y, 'Up'))
    if y > 0: moves.append((x, y - 1, 'Right'))
    if y < 2: moves.append((x, y + 1, 'Left'))
    return moves

def generate_new_state(state, blank_pos, new_blank_pos):
    new_state = [row[:] for row in state]
    new_state[blank_pos[0]][blank_pos[1]], new_state[new_blank_pos[0]][new_blank_pos[1]] = \
        new_state[new_blank_pos[0]][new_blank_pos[1]], new_state[blank_pos[0]][blank_pos[1]]
    return new_state

def a_star_search(initial_state):
    open_list = []
    closed_list = set()

    initial_node = Node(state=initial_state, cost=0)
    open_list.append(initial_node)

    while open_list:

        open_list.sort(key=lambda node: node.cost + node.heuristic())
```

```
        current_node = open_list.pop(0)


        move_description = current_node.move if current_node.move else "Start"
        print("Current state:")
        for row in current_node.state:
            print(row)
        print(f"Move: {move_description}")
        print(f"Heuristic value (misplaced tiles): {current_node.heuristic()}")
        print(f"Cost to reach this node: {current_node.cost}\n")

        if current_node.heuristic() == 0:

            path = []
            while current_node:
                path.append(current_node)
                current_node = current_node.parent
            return path[::-1]
        closed_list.add(tuple(map(tuple, current_node.state)))

        blank_pos = get_blank_position(current_node.state)
        for new_blank_pos in get_possible_moves(blank_pos):
            new_state = generate_new_state(current_node.state, blank_pos, (new_blank_pos[0],
new_blank_pos[1]))

            if tuple(map(tuple, new_state)) in closed_list:
                continue

            cost = current_node.cost + 1
            move_direction = new_blank_pos[2]
            new_node = Node(state=new_state, parent=current_node, move=move_direction, cost=cost)

            if new_node not in open_list:
                open_list.append(new_node)

    return None

initial_state = [[2,8,3], [1,6,4], [7,0,5]]
solution_path = a_star_search(initial_state)

if solution_path:
    print("Solution path:")
    for step in solution_path:
        for row in step.state:
            print(row)
        print()
else:
```

```
    print("No solution found.")
```

```
Current state:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
Move: Left
Heuristic value (misplaced tiles): 0
Cost to reach this node: 5

Solution path:
[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]
```

```
class Node:
    def __init__(self, state, parent=None, move=None, cost=0):
        self.state = state
        self.parent = parent
        self.move = move
        self.cost = cost

    def heuristic(self):
        goal_positions = {
            1: (0, 0), 2: (0, 1), 3: (0, 2),
            8: (1, 0), 0: (1, 1), 4: (1, 2),
            7: (2, 0), 6: (2, 1), 5: (2, 2)
        }
        manhattan_distance = 0
```

```
        for i in range(len(self.state)):
            for j in range(len(self.state[i])):
                value = self.state[i][j]
                if value != 0:
                    goal_i, goal_j = goal_positions[value]
                    manhattan_distance += abs(i - goal_i) + abs(j - goal_j)
        return manhattan_distance

def get_blank_position(state):
    for i in range(len(state)):
        for j in range(len(state[i])):
            if state[i][j] == 0:
                return i, j

def get_possible_moves(position):
    x, y = position
    moves = []
    if x > 0: moves.append((x - 1, y, 'Down'))
    if x < 2: moves.append((x + 1, y, 'Up'))
    if y > 0: moves.append((x, y - 1, 'Right'))
    if y < 2: moves.append((x, y + 1, 'Left'))
    return moves

def generate_new_state(state, blank_pos, new_blank_pos):
    new_state = [row[:] for row in state]
    new_state[blank_pos[0]][blank_pos[1]], new_state[new_blank_pos[0]][new_blank_pos[1]] = \
        new_state[new_blank_pos[0]][new_blank_pos[1]], new_state[blank_pos[0]][blank_pos[1]]
    return new_state

def a_star_search(initial_state):
    open_list = []
    closed_list = set()

    initial_node = Node(state=initial_state, cost=0)
    open_list.append(initial_node)

    while open_list:

        open_list.sort(key=lambda node: node.cost + node.heuristic())
        current_node = open_list.pop(0)


        move_description = current_node.move if current_node.move else "Start"
        print("Current state:")
        for row in current_node.state:
            print(row)
        print(f"Move: {move_description}")
```

```
        print(f"Heuristic value (Manhattan distance): {current_node.heuristic()}")
        print(f"Cost to reach this node: {current_node.cost}\n")

        if current_node.heuristic() == 0:

            path = []
            while current_node:
                path.append(current_node)
                current_node = current_node.parent
            return path[::-1]
        closed_list.add(tuple(map(tuple, current_node.state)))

        blank_pos = get_blank_position(current_node.state)
        for new_blank_pos in get_possible_moves(blank_pos):
            new_state = generate_new_state(current_node.state, blank_pos, (new_blank_pos[0],
new_blank_pos[1]))

            if tuple(map(tuple, new_state)) in closed_list:
                continue

            cost = current_node.cost + 1
            move_direction = new_blank_pos[2]
            new_node = Node(state=new_state, parent=current_node, move=move_direction, cost=cost)

            if new_node not in open_list:
                open_list.append(new_node)

    return None

initial_state = [[2,8,3], [1,6,4], [7,0,5]]
solution_path = a_star_search(initial_state)

if solution_path:
    print("Solution path:")
    for step in solution_path:
        for row in step.state:
            print(row)
        print()
else:
    print("No solution found.")
```

```
Current state:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
Move: Left
Heuristic value (Manhattan distance): 0
Cost to reach this node: 5

Solution path:
[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 Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

Lab 4

22/10/24

Implement hill climbing search algorithm using N-Queens

function Hill_climbing(problem) returns a state that is a local maxima

current ← Make_node(problem Initial_state)

loop do

neighbour ← a highest_valued successor of current

if neighbour.value ≤ current.value then return current.state

current ← neighbour

~~cost~~

$x_0, x_1, x_2, x_3 \longrightarrow x_i \longrightarrow i$ column

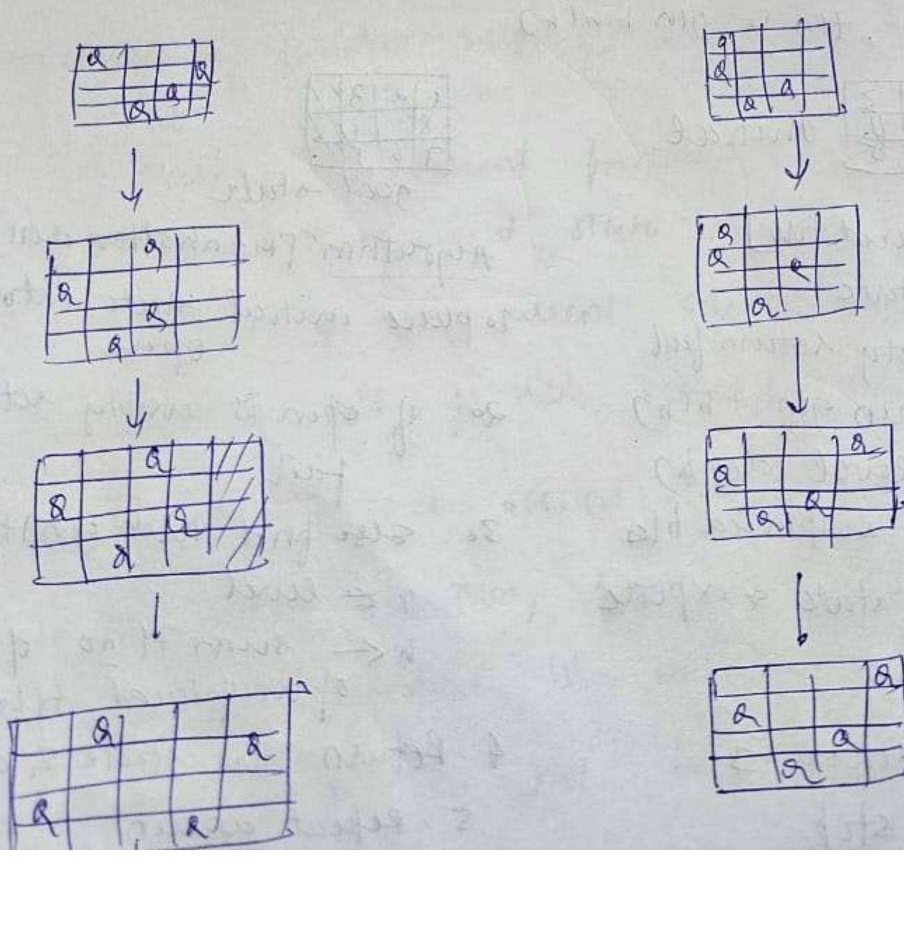
↳ value of row or position

neighbour relation:

swap the row positions of two queens

cost function: no of pairs of queens attack each other directly or indirectly

state space:



Code:

```
import random
def calculate_cost(board):

    n = len(board)

    attacks = 0

    for i in range(n):

        for j in range(i + 1, n):

            if board[i] == board[j]:  # Same column

                attacks += 1

            if abs(board[i] - board[j]) == abs(i - j):  # Same diagonal

                attacks += 1

    return attacks


def get_neighbors(board):

    neighbors = []

    n = len(board)

    for col in range(n):

        for row in range(n):

            if row != board[col]:  # Only change the row of the queen

                new_board = board[:]

                new_board[col] = row

                neighbors.append(new_board)

    return neighbors
```

```
def hill_climb(board, max_restarts=100):
    current_cost = calculate_cost(board)
    print("Initial board configuration:")
    print_board(board, current_cost)
    iteration = 0
    restarts = 0

    while restarts < max_restarts:  # Add limit to the number of restarts
        while current_cost != 0:  # Continue until cost is zero
            neighbors = get_neighbors(board)
            best_neighbor = None
            best_cost = current_cost

            for neighbor in neighbors:
                cost = calculate_cost(neighbor)
                if cost < best_cost:  # Looking for a lower cost
                    best_cost = cost
                    best_neighbor = neighbor

            if best_neighbor is None:  # No better neighbor found
                break  # Break the loop if we are stuck at a local minimum

            board = best_neighbor
```

```
            current_cost = best_cost
            iteration += 1
            print(f"Iteration {iteration}:")
            print_board(board, current_cost)

        if current_cost == 0:
            break  # We found the solution, no need for further restarts
        else:
            # Restart with a new random configuration
            board = [random.randint(0, len(board)-1) for _ in range(len(board))]
            current_cost = calculate_cost(board)
            restarts += 1
            print(f"Restart {restarts}:")
            print_board(board, current_cost)

    return board, current_cost

def print_board(board, cost):
    n = len(board)
    display_board = [['.'] * n for _ in range(n)]  # Create an empty board
    for col in range(n):
        display_board[board[col]][col] = 'Q'  # Place queens on the board
    for row in range(n):
        print(' '.join(display_board[row]))  # Print the board
```

```
    print(f"Cost: {cost}\n")


if __name__ == "__main__":
    n = int(input("Enter the number of queens (N): "))  # User input for N
    initial_state = list(map(int, input(f"Enter the initial state (row numbers for each column, space-separated): ").split()))
    if len(initial_state) != n or any(r < 0 or r >= n for r in initial_state):
        print("Invalid initial state. Please ensure it has N elements with values from 0 to N-1.")
    else:
        solution, cost = hill_climb(initial_state)
        if cost == 0:
            print(f"Solution found with no conflicts:")
        else:
            print(f"No solution found within the restart limit:")
        print_board(solution, cost)
```

```
Enter the number of queens (N): 4
Enter the initial state (row numbers for each column, space-separated): 0 1 2 3
Initial board configuration:
Q . . .
. Q . .
. . Q .
. . . Q
Cost: 6
```

```
Iteration 10:
. Q . .
. . . Q
Q . . .
. . Q .
Cost: 0

Solution found with no conflicts:
. Q . .
. . . Q
Q . . .
. . Q .
Cost: 0
```

Program 5
Simulated Annealing to Solve 8-Queens problem:
Algorithm:

Q) To implement simulated Annealing algorithm. 29/10/24

function SIMULATED-ANNEALING (problem, schedule) return a solution

inputs : problem, a problem.
schedule, a mapping from time to "temperature".

current ← MAKE-NODE (problem, INITIAL-STATE)

for t=1 to ∞ do.

T ← schedule (t).

if T = 0 then return a current.

next ← a randomly selected successor of current

ΔE ← next.value − current.value.

if ΔE > 0 then current ← next

else current ← next only with probability $e^{\Delta E/T}$

output:-

The best position found is [2 6 1 7 4 0 3 5]

The number of queens that are not attacking each other: 8

output:-

solved solution.

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

29/10/24

output:-

edges in the MST:

0 - 2 (weight = 1)

8 - 3 (weight : 3)

Code:

```
#!pip install mlrose-hiive joblib
#!pip install --upgrade joblib
#!pip install joblib==1.1.0
import mlrose_hiive as mlrose
import numpy as np

def queens_max(position):
    no_attack_on_j = 0
    queen_not_attacking = 0
    for i in range(len(position) - 1):
        no_attack_on_j = 0
        for j in range(i + 1, len(position)):
            if (position[j] != position[i]) and (position[j] != position[i] + (j - i)) and (position[j] !=
position[i] - (j - i)):
                no_attack_on_j += 1
        if (no_attack_on_j == len(position) - 1 - i):
            queen_not_attacking += 1
    if (queen_not_attacking == 7):
        queen_not_attacking += 1
    return queen_not_attacking

objective = mlrose.CustomFitness(queens_max)

problem = mlrose.DiscreteOpt(length=8, fitness_fn=objective, maximize=True, max_val=8)
T = mlrose.ExpDecay()

initial_position = np.array([4, 6, 1, 5, 2, 0, 3, 7])

#The simulated_annealing function returns 3 values, we need to capture all 3
best_position, best_objective, fitness_curve = mlrose.simulated_annealing(problem=problem,
schedule=T, max_attempts=500,
                                                  init_state=initial_position)

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

```
The best position found is: [4 0 7 5 2 6 1 3]
The number of queens that are not attacking each other is: 8.0
```

Program 6

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

Algorithm:

Date: 12/11/24

·) Wumpus world using propositional logic

$R_1: \neg P_{1,1}$

$R_2: B_{1,1} \Leftrightarrow P_{1,2} \vee P_{2,1}$ [R → resolution]

$R_3: B_{21} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$

$P_{x,y}$

$W_{x,y}$

$B_{x,y}$

$S_{x,y}$.

Create a KB using propositional logic & show that given queries entails the knowledge base or not.

function TT-entails? (KB, α) returns true or false

input: KB, the knowledge base

α, a query

symbols ← a list of all the propositional symbols in KB and α

return TT-check-ALL (KB, α, symbols, {})

function TT-check-All (KB, α, symbol, model) returns true or false

if empty? (symbols then)

if PL-true? (KB, model) then return PL-true

else return true when KB is false always return the

else do.

P ← first (symbols)

rest ← rest (symbols)

return [TT-check-All(KB, α, rest, model ∪ {P=true})

and

[TT-check-ALL (KB, α, rest, model ∪ {P=false}))

α = A ∨ B   KB = (A ∨ C) ∧ (B ∨ ¬C)

A	B	C	A ∨ C	B ∨ ¬C	KB	α
F	F	F	F	T	F	F
F	F	T	T	F	F	F
F	T	F	F	T	F	T

3-12-24

Code:

```
import pandas as pd


# Define the truth table for all combinations of A, B, C
truth_values = [(False, False, False),
                (False, False, True),
                (False, True, False),
                (False, True, True),
                (True, False, False),
                (True, False, True),
                (True, True, False),
                (True, True, True)]



# Columns: A, B, C
table = pd.DataFrame(truth_values, columns=["A", "B", "C"])


# Calculate intermediate columns
table["A or C"] = table["A"] | table["C"]          # A ∨ C
table["B or not C"] = table["B"] | ~table["C"]     # B ∨ ¬C


# Knowledge Base (KB): (A ∨ C) ∧ (B ∨ ¬C)
table["KB"] = table["A or C"] & table["B or not C"]



# Alpha (α): A ∨ B
table["Alpha (α)"] = table["A"] | table["B"]





# Define a highlighting function
def highlight_rows(row):
    if row["KB"] and row["Alpha (α)"]:
        return ['background-color: blue'] * len(row)
    else:
        return [''] * len(row)
```

```
# Apply the highlighting function
styled_table = table.style.apply(highlight_rows, axis=1)

# Display the styled table
styled_table
```

	A	B	C	A or C	B or not C	KB	Alpha (α)
0	False	False	False	False	True	False	False
1	False	False	True	True	False	False	False
2	False	True	False	False	True	False	True
3	False	True	True	True	True	True	True
4	True	False	False	True	True	True	True
5	True	False	True	True	False	False	True
6	True	True	False	True	True	True	True
7	True	True	True	True	True	True	True

Program 7

Implement unification in first order logic

Algorithm:

Lab - 7

First order logic → unification

1) If Ψ_1 or Ψ_2 is a variable or constant then:

a) If Ψ_1 or Ψ_2 are identical, return NIL.

b) else if Ψ_1 is a variable,

(a) then if Ψ_1 occurs in Ψ_2, then return failure

(b) else return $\{(\Psi_2 | \Psi_1)\}$

(c) else if Ψ_2 is a variable

(a) if Ψ_2 occurs in Ψ_1 then return failure

(b) else return $\{(\Psi_1 | \Psi_2)\}$

(d) else return failure

2) If the initial predicate symbol in Ψ_1 and Ψ_2 are not same same, then return failure

3) If Ψ_1 & Ψ_2 have a diff no of arguments, return failure

4) set substitution set (subset) to NIL.

5) for i = 1 to no of elements in Ψ_1.

(a) call unify function with the ith element of Ψ_1 & ith element of Ψ_2, and put the result into S

b) If S = failure then return failure

(c) If S ≠ NILL then do,

(a) Apply S to the remainder of both L & L

(b) SUBST = Append (S, SUBST)

(6) Return SUBST

Q(a, g(x, a), f(y)) → ①

Q(a, g(f(b), a), x) → ②

replace x in ① with f(b) ⇒ Q(a, g(f(b), a), f(y))

replace f(y) in ② with x ⇒ Q(a, g(f(b), a), x)

Code:

```
import re

def occurs_check(var, x):
    """Checks if var occurs in x (to prevent circular substitutions)."""
    if var == x:
        return True
    elif isinstance(x, list):  # If x is a compound expression (like a function or predicate)
        return any(occurs_check(var, xi) for xi in x)
    return False


def unify_var(var, x, subst):
    """Handles unification of a variable with another term."""
    if var in subst:  # If var is already substituted
        return unify(subst[var], x, subst)
    elif isinstance(x, (list, tuple)) and tuple(x) in subst:  # Handle compound expressions
        return unify(var, subst[tuple(x)], subst)
    elif occurs_check(var, x):  # Check for circular references
        return "FAILURE"
    else:
        # Add the substitution to the set (convert list to tuple for hashability)
        subst[var] = tuple(x) if isinstance(x, list) else x
        return subst


def unify(x, y, subst=None):
    """
    Unifies two expressions x and y and returns the substitution set if they can be unified.
    Returns 'FAILURE' if unification is not possible.
    """
    if subst is None:
        subst = {}  # Initialize an empty substitution set

    # Step 1: Handle cases where x or y is a variable or constant
    if x == y:  # If x and y are identical
        return subst
    elif isinstance(x, str) and x.islower():  # If x is a variable
        return unify_var(x, y, subst)
    elif isinstance(y, str) and y.islower():  # If y is a variable
        return unify_var(y, x, subst)
    elif isinstance(x, list) and isinstance(y, list):  # If x and y are compound expressions (lists)
        if len(x) != len(y):  # Step 3: Different number of arguments
            return "FAILURE"
```

```
        # Step 2: Check if the predicate symbols (the first element) match
        if x[0] != y[0]:  # If the predicates/functions are different
            return "FAILURE"

        # Step 5: Recursively unify each argument
        for xi, yi in zip(x[1:], y[1:]):  # Skip the predicate (first element)
            subst = unify(xi, yi, subst)
            if subst == "FAILURE":
                return "FAILURE"
        return subst
    else:  # If x and y are different constants or non-unifiable structures
        return "FAILURE"


def unify_and_check(expr1, expr2):
    """
    Attempts to unify two expressions and returns a tuple:
    (is_unified: bool, substitutions: dict or None)
    """
    result = unify(expr1, expr2)
    if result == "FAILURE":
        return False, None
    return True, result


def display_result(expr1, expr2, is_unified, subst):
    print("Expression 1:", expr1)
    print("Expression 2:", expr2)
    if not is_unified:
        print("Result: Unification Failed")
    else:
        print("Result: Unification Successful")
        print("Substitutions:", {k: list(v) if isinstance(v, tuple) else v for k, v in subst.items()})


def parse_input(input_str):
    """Parses a string input into a structure that can be processed by the unification algorithm."""
    # Remove spaces and handle parentheses
    input_str = input_str.replace(" ", "")

    # Handle compound terms (like p(x, f(y)) -> ['p', 'x', ['f', 'y']])
    def parse_term(term):
        # Handle the compound term
        if '(' in term:
            match = re.match(r'([a-zA-Z0-9_]+)
', term)
            if match:
```

```
            predicate = match.group(1)
            arguments_str = match.group(2)
            arguments = [parse_term(arg.strip()) for arg in arguments_str.split(',')]
            return [predicate] + arguments
        return term

    return parse_term(input_str)


# Main function to interact with the user
def main():
    while True:
        # Get the first and second terms from the user
        expr1_input = input("Enter the first expression (e.g., p(x, f(y))): ")
        expr2_input = input("Enter the second expression (e.g., p(a, f(z))): ")

        # Parse the input strings into the appropriate structures
        expr1 = parse_input(expr1_input)
        expr2 = parse_input(expr2_input)

        # Perform unification
        is_unified, result = unify_and_check(expr1, expr2)

        # Display the results
        display_result(expr1, expr2, is_unified, result)

        # Ask the user if they want to run another test
        another_test = input("Do you want to test another pair of expressions? (yes/no): ").strip().lower()
        if another_test != 'yes':
            break


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

```
Enter the first expression (e.g., p(x, f(y))): p(b,x,f(g(z)))
Enter the second expression (e.g., p(a, f(z))): p(z,f(y),f(y))
Expression 1: ['p', 'b', 'x', ['f', ['g', 'z']]]
Expression 2: ['p', 'z', ['f', 'y'], ['f', 'y']]
Result: Unification Successful
Substitutions: {'b': 'z', 'x': ['f', 'y'], 'y': ['g', 'z']}
Do you want to test another pair of expressions? (yes/no): yes
Enter the first expression (e.g., p(x, f(y))): p(x,h(y))
Enter the second expression (e.g., p(a, f(z))): p(a,f(z))
Expression 1: ['p', 'x', ['h', 'y']]
Expression 2: ['p', 'a', ['f', 'z']]
Result: Unification Failed
Do you want to test another pair of expressions? (yes/no): yes
Enter the first expression (e.g., p(x, f(y))): p(f(a),g(y))
Enter the second expression (e.g., p(a, f(z))): p(x,x)
Expression 1: ['p', ['f', 'a'], ['g', 'y']]
Expression 2: ['p', 'x', 'x']
Result: Unification Failed
Do you want to test another pair of expressions? (yes/no): no
```

Program 8

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

Lab-8

Forward reasoning algorithm

function FOL-FC-ASK(KB, α) returns a substitution or false

inputs. KB, the knowledge base, a set of first-order definite clauses

α, the query, an atomic sentence

local variables: new, the new sentences inferred on each iteration

repeat until new is empty

new ← [ ]

for each rule in KB do

$(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES(rule)

for each θ such that SUBST(θ, $p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta$, $p_1' \wedge \ldots \wedge p_n'$)

for some $p_1', \ldots p_n'$ in KB

$q' \leftarrow$ SUBST(θ, q)

if q' does not unify with some sentence already in KB or new then

add q' to new

$\phi \leftarrow$ UNIFY(q', α)

if ϕ is not fail then return ϕ

add new to KB

return false

26/11/21

Code:

```
class KnowledgeBase:
    def __init__(self):
        self.facts = set()   # Set of known facts
        self.rules = []      # List of rules

    def add_fact(self, fact):
        self.facts.add(fact)

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

    def infer(self):
        inferred = True
        while inferred:
            inferred = False
            for rule in self.rules:
                if rule.apply(self.facts):
                    inferred = True

# Define the Rule class
class Rule:
    def __init__(self, premises, conclusion):
        self.premises = premises  # List of conditions
        self.conclusion = conclusion  # Conclusion to add if premises are met

    def apply(self, facts):
        if all(premise in facts for premise in self.premises):
            if self.conclusion not in facts:
                facts.add(self.conclusion)
                print(f"Inferred: {self.conclusion}")
                return True
        return False

# Initialize the knowledge base
kb = KnowledgeBase()

# Facts in the problem
kb.add_fact("American(Robert)")
kb.add_fact("Missile(T1)")
kb.add_fact("Owns(A, T1)")
kb.add_fact("Enemy(A, America)")
```

```
# Rules based on the problem
# 1. Missile(x) implies Weapon(x)
kb.add_rule(Rule(["Missile(T1)"], "Weapon(T1)"))

# 2. Enemy(x, America) implies Hostile(x)
kb.add_rule(Rule(["Enemy(A, America)"], "Hostile(A)"))

# 3. Missile(x) and Owns(A, x) imply Sells(Robert, x, A)
kb.add_rule(Rule(["Missile(T1)", "Owns(A, T1)"], "Sells(Robert, T1, A)"))

# 4. American(p) and Weapon(q) and Sells(p, q, r) and Hostile(r) imply Criminal(p)
kb.add_rule(Rule(["American(Robert)", "Weapon(T1)", "Sells(Robert, T1, A)", "Hostile(A)"],
"Criminal(Robert)"))

# Infer new facts based on the rules
kb.infer()

# Check if Robert is a criminal
if "Criminal(Robert)" in kb.facts:
    print("Conclusion: Robert is a criminal.")
else:
    print("Conclusion: Unable to prove Robert is a criminal.")
```

```
Inferred: Weapon(T1)
Inferred: Hostile(A)
Inferred: Sells(Robert, T1, A)
Inferred: Criminal(Robert)
Conclusion: Robert is a criminal.
```

Program 9

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

Lab program_9 | 06/11/2…

convert a given first order logic statement into resolut…

Basic steps for proving a conclusion S given premises

(all expressed in FOL):

1. convert all sentences to CNF
2. Negate conclusion S & convert result to CNF.
3. Add negated conclusion S to the premise clauses.
4. Repeat until contradiction or no progress is made:
   a. Select 2 clauses (call them parent clauses)
   b. Resolve them together, performing all required unif…
   c. If the resolvent is empty clause, a contradiction ha… been found (i.e S follows from the premises)
   d. If not, add resolvent to the premises

If we succeeded in step 4, we have proved the conclusi…

Given KB

John likes all kinds of food

Apple and vegetable are food

Any thing anyone eats and not killed is food

Anil eats peanuts and still alive

Harry eats everything that Anil eats

Anyone who is alive implies alive

¬food(x) ∨ likes(John, x)

food(apple)

food(vegetables)

¬eats(y, z) ∨ killed(y) ∨ food(z)

eats(Anil, peanuts)

alive(Anil)

¬eats(Anil, w) ∨ eats(Harry, w)

killed(g) ∨ alive(g)

¬alive(k) ∨ ¬killed(k)

likes(John, peanuts)

¬likes(John, peanuts) ¬food(x) ∨ likes(John, x)

¬food(peanuts) ¬eat(y, z) ∨ killed(y) ∨ food(z)

¬eats(y, peanuts) ∨ killed(y) eats(Anil, peanuts)

killed(Anil) ¬alive(k) ∨ ¬killed(x)

¬alive(Anil) alive(Anil)

{ }

Code:

```
KB = {
    "food(Apple)": True,
    "food(vegetables)": True,
    "eats(Anil, Peanuts)": True,
    "alive(Anil)": True,
    "likes(John, X)": "food(X)",  # Rule: John likes all food
    "food(X)": "eats(Y, X) and not killed(Y)",  # Rule: Anything eaten and not killed is food
    "eats(Harry, X)": "eats(Anil, X)",  # Rule: Harry eats what Anil eats
    "alive(X)": "not killed(X)",  # Rule: Alive implies not killed
    "not killed(X)": "alive(X)",  # Rule: Not killed implies alive
}

# Function to evaluate if a predicate is true based on the KB
def resolve(predicate):
    # If it's a direct fact in KB
    if predicate in KB and isinstance(KB[predicate], bool):
        return KB[predicate]

    # If it's a derived rule
    if predicate in KB:
        rule = KB[predicate]
        if " and " in rule:  # Handle conjunction
            sub_preds = rule.split(" and ")
            return all(resolve(sub.strip()) for sub in sub_preds)
        elif " or " in rule:  # Handle disjunction
            sub_preds = rule.split(" or ")
            return any(resolve(sub.strip()) for sub in sub_preds)
        elif "not " in rule:  # Handle negation
            sub_pred = rule[4:]  # Remove "not "
            return not resolve(sub_pred.strip())
        else:  # Handle single predicate
            return resolve(rule.strip())

    # If the predicate is a specific query (e.g., likes(John, Peanuts))
    if "(" in predicate:
        func, args = predicate.split("(")
        args = args.strip(")").split(", ")
        if func == "food" and args[0] == "Peanuts":
            return resolve("eats(Anil, Peanuts)") and not resolve("killed(Anil)")
```

```
        if func == "likes" and args[0] == "John" and args[1] == "Peanuts":
            return resolve("food(Peanuts)")

    # Default to False if no rule or fact applies
    return False

# Query to prove: John likes Peanuts
query = "likes(John, Peanuts)"
result = resolve(query)

# Print the result
print(f"Does John like peanuts? {'Yes' if result else 'No'}")
```

Does John like peanuts? Yes

Program 10

Implement Alpha-Beta Pruning.

Implement alpha-beta pruning

function ALPHA-BETA-SEARCH(state) returns an action
v ← max-value (state, -∞, +∞)
return the action in Actions(state) with v

function max-value (state, α, β) return a utility value
if terminal-test (state) then return utility (state)
v ← -∞
for each a in actions(state) do
v ← max (v, min-value(Result (s,a), α, β))
if v ≥ β then return v
α ← max(α, v)
return v

function min-value (state, α, β) return a utility value
if terminal-test (state) then return utility
v ← +∞
for each a in actions (states) do
v ← min (v, max-value (Result(S,a), α, β))
if v ≤ α then return v
β ← min(β, v)

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}")
```

The best value for the maximizing player is: 6