B.M.S. COLLEGE OF ENGINEERING
BENGALURU Autonomous Institute, Affiliated to VTU

Lab Record

Artificial Intelligence

(22CS5PCAIN)

Submitted in partial fulfillment for the 5th Semester Laboratory

Bachelor of Technology
in
Computer Science and Engineering

Submitted by:

Sneha Santhosh Bhat

1BM21CS213

Department of Computer Science and
Engineering B.M.S. College of Engineering
Bull Temple Road, Basavanagudi, Bangalore
560 November 2023-February 2024

B.M.S. COLLEGE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that the Artificial Intelligence (20CS5PCAIP) laboratory has been carried out by Sneha Santhosh Bhat (1BM21CS213) during the 5th Semester November 2023-February 2024.

Signature of the Faculty Incharge:

Sneha S Bagalkot
Assistant Professor
Department of Computer Science and Engineering
B.M.S. College of Engineering, Bangalore

Table of Contents

Sl. No.	Title	Page No.
1.	Tic Tac Toe	
2.	8 Puzzle Breadth First Search Algorithm	
3.	8 Puzzle Iterative Deepening Search Algorithm	
4.	8 Puzzle A* Search Algorithm	
5.	Vacuum Cleaner	
6.	Knowledge Base Entailment	
7.	Knowledge Base Resolution	
8.	Unification	
9.	FOL to CNF	
10.	Forward reasoning	

WEEK - 1

Analyze and implement Tic-tac-toe game.

Program :-

```
import math
import copy
X = "x"
O = "O"
EMPTY = None
def initial_state():
    return [[EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY]]
def player(board):
    countO = 0
    countX = 0
    for y in [0, 1, 2]:
        for x in board[y]:
            if x == "O":
                countO = countO + 1
            elif x == "x":
                countX = countX + 1
    if countO >= countX:
        return X
    elif countX > countO:
        return O
def actions(board):
    freeboxes = set()
    for i in [0, 1, 2]:
        for j in [0, 1, 2]:
            if board[i][j] == EMPTY:
                freeboxes.add((i, j))
    return freeboxes
```

WEEK - 1

Analyze and implement Tic-tac-toe game.

Program :-

```
import math
import copy
X = "x"
O = "O"
EMPTY = None
def initial_state():
    return [[EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY]]
def player(board):
    countO = 0
    countX = 0
    for y in [0, 1, 2]:
        for x in board[y]:
            if x == "O":
                countO = countO + 1
            elif x == "x":
                countX = countX + 1
    if countO >= countX:
        return X
    elif countX > countO:
        return O
def actions(board):
    freeboxes = set()
    for i in [0, 1, 2]:
        for j in [0, 1, 2]:
            if board[i][j] == EMPTY:
                freeboxes.add((i, j))
    return freeboxes
```

```
def result (board, action):
    i = action [0]
    j = action [1]
    if type(action) == list:
        action = (i, j)
    if action in actions (board):
        if player (board) == X:
            board [i][j] = X
        elif player (board) == 0:
            board [i][j] = 0
    return board
def winner (board):
    if (board [0][0] == board [0][1] == board [0][2] == X or board [1][0]
        == board (1)(1)== board [1][2] == X or board [2][0] ==
        board [2][1] == board [2][2] == X):
        return X
    if (board [0][0] == board [0][1]== board [0][2] ==0 or board [1][0]
        == board [1][1] == board (1)[2] == 0 or board [2][0] ==
        board [2][1] == board [2][2] = 0)
        return 0
    for i in [0, 1, 2]:
        s2 = []
        for j in [0, 1, 2]:
            s2. append (board [j][i])
        if (s2[0] == s2 [1] == s2 [2]):
            return s2 [0]
    strikeD = []
    for i in [0, 1, 2]:
        strikeD. append (board [i][i])
    if (strikeD [0] == strikeD [1] == strikeD [2]):
        return strikeD [0]
```

```
    if (board[0][2] == board[1][1] == board[2][0]):
        return board[0][2]
    return None
def terminal(board):
    Full = True
    for i in [0, 1, 2]:
        for j in board[i]:
            if j is None:
                Full = False
    if Full:
        return True
    if (winner(board) is not None):
        return True
    return False
def utility(board):
    if (winner(board) == X):
        return 1
    elif winner(board) == 0:
        return -1
    else:
        return 0
def minimax_helper(board):
    isMaxTurn = True if player(board) == X else False
    if terminal(board):
        return utility(board)
    scores = []
    for move in actions(board):
        result(board, move)
        scores.append(minimax_helper(board))
        board[move[0]][move[1]] = EMPTY
    return max(scores) if isMaxTurn else min(scores)
```

```
def minimax (board):
    isMaxTurn = True if player(board) == X else False
    bestMove = None
    if isMaxTurn:
        bestScore = -math.inf
        for move in actions(board):
            result(board, move)
            score = minimax_helper(board)
            board[move[0]][move[1]] = EMPTY
            if (score > bestScore):
                bestScore = score
                bestMove = move
        return bestMove
    else:
        bestScore = +math.inf
        for move in actions(board):
            result(board, move)
            score = minimax_helper(board)
            board[move[0]][move[1]] = EMPTY
            if (score < bestScore):
                bestScore = score
                bestMove = move
        return bestMove
def print_board(board):
    for row in board:
        print(row)
game_board = initial_state()
print("Initial Board:")
print_board(game_board)
while not terminal(game_board):
    if player(game_board) == X:
        user_input = input("\n Enter your move (row, column): ")
```

```
            row, col = map(int, user_input.split(','))
            result(game_board, (row, col))
    else:
            print("\n AI is making a move...")
            move = minimax(copy.deepcopy(game_board))
            result(game_board, move)
    print("\nCurrent Board:")
            print_board(game_board)
    if winner(game_board) is not None:
            printf"\n The winner is: {winner(game_board)}")
    else:
            print("\n It's a tie!")
```

Output :-

Initial Board :

[None, None, None]

[None, None, None]

[None, None, None]

Enter your move (row, column): 1, 1

Current Board:

[None, None, None]

[None, 'x', None]

[None, None, None]

AI is making a move...

Current Board:

['O', None, None]

[None, 'x', None]

[None, None, None]

Enter your move (row, column): 1, 2

Current Board:

['O', None, None]

[None, 'x', 'x']

[None, None, None]

AI is making a move...
Current Board:
['O', None, None]
['O', 'x', 'x']
[None, None, None]
Enter your move (row, column): 2, 0
Current Board:
['O', None, None]
['O', 'x', 'x']
['x', None, None]
AI is making a move...
Current Board:
['O', None, 'O']
['O', 'x', 'x']
['x', None, None]
Enter your move (row, column): 0, 1
Current Board:
['O', 'x', 'O']
['O', 'x', 'x']
['x', None, None]
AI is making a move...
Current Board:
['O', 'x', 'O']
['O', 'x', 'x']
['x', None, None]
Enter your move (row, column): 2, 2
Current Board:
['O', 'x', 'O']
['O', 'x', 'x']
['x', 'O', None]
Enter your move (row, column): 2, 2
Current Board:
['O', 'x', 'O']
['O', 'x', 'x']
['x', 'O', 'x']
It's a tie!

WEEK - 2

Analyze and implement vacuum cleaner agent.

Program :-

```
def vacuum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter Location of Vacuum")
    status_input = input("Enter status of " + location_input)
    status_input_complement = input("Enter status of other room")
    print("Initial Location Condition" + str(goal_state))
    if location_input == 'A':
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")
            goal_state['A'] = '0'
            cost += 1
            print("Cost for CLEANING A " + str(cost))
            print("Location A has been cleaned.")
        if status_input_complement == '1':
            print("Location B is Dirty.")
            print("Moving right to the location B.")
            cost += 1
            print("COST for moving RIGHT " + str(cost))
            goal_state['B'] = '0'
            cost += 1
            print("COST for SUCK " + str(cost))
            print("Location B has been cleaned.")
    else:
        print("No action" + str(cost))
        print("Location B is already clean.")
    if status_input == '0':
        print("Location A is already clean")
```

```
if status_input_complement == '1':
    print("Location B is Dirty.")
    print("Moving RIGHT to the Location B.")
    cost += 1
    print("COST for moving RIGHT" +str(cost))
    goal_state['B'] = '0'
    cost += 1
    print("Cost for SUCK" + str(cost))
    print("Location B has been Cleaned")
else:
    print("No action" +str(cost))
    print(cost)
    print("Location B is already clean")
else:
    print("Vacuum is placed in location B")
    if status_input == '1':
        print("Location B is Dirty.")
        goal_state['B'] = '0'
        cost += 1
        print("COST for CLEANING" + str(cost))
        print("Location B has been Cleaned.")
        if status_input_complement == '1':
            print("Location A is dirty.")
            print("Moving LEFT to the location A.")
            cost += 1
            print("COST for moving LEFT" +str(cost))
            goal_state['A'] = '0'
            cost += 1
            print("COST for SUCK" + str(cost))
            print("Location A has been Cleaned.")
```

```
else:
        print(cost)
        print("Location B is already clean.")
        if status_input_complement == '1':
            print("Location A is Dirty")
            print("Moving LEFT to the location A")
            cost += 1
            print("cost for moving LEFT" + str(cost))
            goal_state['A'] = '0'
            cost += 1
            print("cost for SUCK" + str(cost))
            print("Location A has been cleaned")
        else:
            print("No action" + str(cost))
            print("Location A is already clean.")
print("GOAL STATE:")
print(goal_state)
print("Performance Measurement:" + str(cost))
vacuum_world()
```

Output :-

① Enter location of Vacuum A
Enter status of A 1
Enter status of other room 1
Initial Location Condition {'A': '0', 'B', '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A 1
Location A has been Cleaned.
Location B is Dirty.
Moving right to the Location B

COST for moving RIGHT 2
COST for SUCK 3
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3

② Enter Location of Vacuum A
Enter status of A 0
Enter status of other room 0
Initial Location Condition {'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is already clean
No action 0
0
Location B is already clean.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 0

③ Enter Location of Vacuum B
Enter status of B 0
Enter status of other room 1
Initial Location Condition {'A': '0', 'B': '0'}
Vacuum is placed in location B
0
Location B is already clean
Location A is Dirty.
Moving LEFT to the Location A
COST for moving LEFT 1
Cost for SUCK 2
Location A has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}

WEEK - 3

Analyze 8 Puzzle problem and implement the same using Breadth First Search Algorithm

Program :-

```
def bfs (src, target):
    queue = []
    queue.append (src)
    exp = []
while len(queue) > 0:
    source = queue.pop(0)
    exp.append (source)
    print (source)
    if (source == target):
        print ("success")
        return
    poss_moves_to_do = []
    poss_moves_to_do = possible_moves (source, exp)
    for move in poss_moves_to_do:
        if move not in exp and move not in queue:
            queue.append (move)
def possible_moves (state, visited_states):
    b = state.index (0)
    d = []
    if b not in [0, 1, 2]:
        d.append ('u')
    if b not in [6, 7, 8]:
        d.append ('d'):
    if b not in [0, 3, 6]:
        d.append ('l')
    if b not in [2, 5, 8]:
        d.append ('r'
pos_moves_it_can = []
```

```
    for i in d:
        pos_moves_it_can.append(gen(state, i, b))
    return [move_it_can for move_it_can in pos_moves_it_can
            if move_it_can not in visited_states]
def gen(state, m, b):
    temp = state.copy()
    if m == 'd':
        temp[b+3], temp[b] = temp[b], temp[b+3]
    if m == 'u':
        temp[b-3], temp[b] = temp[b], temp[b-3]
    if m == 'l':
        temp[b-1], temp[b] = temp[b], temp[b-1]
    if m == 'r':
        temp[b+1], temp[b] = temp[b], temp[b+1]
    return temp
src = [1, 2, 3, 0, 4, 5, 6, 7, 8]
target = [1, 2, 3, 4, 5, 0, 6, 7, 8]
src = [2, 0, 3, 1, 8, 4, 7, 6, 5]
target = [1, 2, 3, 8, 0, 4, 7, 6, 5]
bfs(src, target)
```

Output :-

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

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

Sneha 12/12/23

WEEK - 4

Analyze Iterative Deepening Search Algorithm. Determine how 8 Puzzle problem could be solved using this algorithm. Implement the same.

Program:-

```
def get_path(node):
    path = []
    current = node
    while current:
        path.append((current.state, current.action))
        current = current.parent
    return path[::-1]
def get_neighbors(state):
    neighbors = []
    empty_index = state.index(0)
    row, col = divmod(empty_index, 3)
    for move in [(0,1), (1,0), (0,-1), (-1,0)]:
        new_row, new_col = row + move[0], col + move[1]
        if 0 <= new_row < 3 and 0 <= new_col < 3
            neighbor_state = list(state)
            neighbor_index = new_row * 3 + new_col
            neighbour_state[empty_index], neighbour_state[neighbour_index] = C
            neighbour_state[neighbour_index],
            neighbour_state[empty_index],
        )
                neighbors.append(tuple(neighbor_state))
    return neighbors
def depth_limited_search(state, goal_state, depth_limit,
    if state == goal_state:                    parent=None, action=None):
        return True
    elif depth_limit == 0:
        return False
```

```
    else:
        for neighbor_state in get_neighbors (state):
            result = depth_limited_search (
                neighbor_state, goal_state, depth_limit
                                                -1,
                    state, action
            )
        if result:
            return True
        return False
initial_state = eval (input (" src = "))
goal_state = eval (input (" target = "))
depth_limit = int (input (" Enter the depth limit : "))
result = depth_limited_search (initial_state, goal_state,
                                                depth_limit
print (result)
```

Output :

① src = (1, 2, 3, 0, 4, 5, 6, 7, 8)
target = (1, 2, 3, 4, 5, 0, 6, 7, 8)
Enter the depth limit : 1
False

② src = (3, 5, 2, 8, 7, 6, 4, 1, 0)
target = (0, 3, 7, 1, 5, 4, 6, 2)
Enter the depth limit : 1
False

③ src = (1, 2, 3, 0, 4, 5, 6, 7, 8)
target = (1, 2, 3, 6, 4, 5, 0, 7, 8)
Enter the depth limit : 1
True

Sneha
6/12/23

WEEK - 5

1. Analyze best first search algorithm. Implement 8 puzzle problem using the algorithm. Calculate complexity in the above and infer your answer.

Program :-

```
import heapq
class PuzzleNode:
    def __init__(self, state, parent = None):
        self.state = state
        self.parent = parent
        self.cost = 0
    def __lt__(self, other):
        return self.cost < other.cost
def manhattan_distance(state, goal_state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != goal_state[i][j]:
                x,y = divmod(goal_state[i][j], 3)
                distance += abs(x-i) + abs(y-j)
    return distance
def get_blank_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i,j
def get_neighbors(node):
    i,j = get_blank_position(node.state)
    neighbors = []
    for x,y in ((i+1,j),(i-1,j),(i,j+1),(i,j-1)):
        if 0<=x<3 and 0<=y<3:
            neighbor_state = [row[:] for row in node.state]
            neighbor_state[i][j], neighbor_state[x][y]
                = neighbor_state[x][y],
                neighbor_state[i][j]
```

```
                neighbors.append(PuzzleNode(neighbor_state, parent_node))
        return neighbors
def greedy_best_first_search(initial_state, goal_state, heuristic):
    initial_node = PuzzleNode(initial_state)
    goal_node = PuzzleNode(goal_state)
    if initial_state == goal_state:
        return [initial_state]
    priority_queue = [initial_node]
    visited_states = set()
    while priority_queue:
        current_node = heapq.heappop(priority_queue)
        if current_node.state == goal_state:
            path = [current_node.state]
            while current_node.parent:
                current_node = current_node.parent
                path.append(current_node.state)
            path.reverse()
            return path
        visited_states.add(tuple(map(tuple, current_node.state)))
        neighbors = get_neighbors(current_node)
        for neighbor in neighbors:
            if tuple(map(tuple, neighbor.state)) not in visited_states:
                neighbor.cost = heuristic(neighbor.state, goal_state)
                heapq.heappush(priority_queue, neighbor)
    return None
# Example usage:
initial_state = [
    [1, 2, 3],
    [4, 0, 5],
    [6, 7, 8],
]
```

```
goal_state = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0],
]
# Heuristic function (Manhattan distance)
heuristic_function = manhattan_distance
path = greedy_best_first_search(initial_state, goal_state,
                                heuristic_function)
if path:
    print("Solution found:")
    for state in path:
        for row in state:
            print(row)
        print()
else:
    print("No solution found")
```

Output:

Solution found:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2. Analyze ~~best first search~~ A* algorithm. Implement 8 puzzle problem using ~~both~~ the algorithm. Calculate complexity in ~~both~~ the above and infer your answer.

Program :-

```
import heapq
class PuzzleNode:
    def __init__(self, state, parent = None, g=0, h=0):
        self.state = state
        self.parent = parent
        self.g = g
        self.h = h
        self.f = self.g + self.h
    def __lt__(self, other):
        return self.f < other.f
def manhattan_distance(state, goal_state):
    distance = 0
```

```
        for i in range(3):
            for j in range(3):
                if state[i][j] != goal_state[i][j] and
                                    state[i][j] != 0
                    x, y = divmod(goal_state
                                  [i][j], 3)
                    distance += abs(x-i) + abs(y-j)
        return distance
    def get_blank_position(state):
        for i in range(3):
            for j in range(3):
                if state[i][j] == 0:
                    return i, j
    def get_neighbors(node):
        i, j = get_blank_position(node.state)
        neighbors = []
        for x, y in ((i+1, j), (i-1, j), (i, j+1), (i, j-1)):
            if 0 <= x < 3 and 0 <= y < 3:
                neighbor_state = [row[:] for row in node.state]
                neighbor_state[i][j], neighbor_state[x][y]
                    = neighbor_state[x][y], neighbor_state[i][j]
                neighbors.append(PuzzleNode(neighbor_state,
                                            parent=node))
        return neighbors

    def a_star_search(initial_state, goal_state, hemictic):
        initial_node = PuzzleNode(initial_state, g=0,
                                  h=hemistic(initial_state,
                                             goal_state))
        goal_node = PuzzleNode
                    (goal_state)
        if initial_state == goal_state:
            return [initial_state]
        priority_queue = [initial_node]
        visited_states = set()
        while priority_queue:
            current_node = heapq.heappop(priority_queue)
```

```
if current_node.state == goal_state:
    path = [current_node.state]
    while current_node.parent:
        current_node = current_node.parent
        path.append(current_node.state)
    path.reverse()
    return path
visited_states.add(tuple(map(tuple, current_node.state)))
neighbors = get_neighbors(current_node)
for neighbor in neighbors:
    if tuple(map(tuple, neighbor.state)) not in visited_states:
        neighbor.g = current_node.g + 1
        neighbor.h = heuristic(neighbor.state, goal_state)
        neighbor.f = neighbor.g + neighbor.h
        heapq.heappush(priority_queue, neighbor)
return None
# Example usage:
initial_state = [
    [1, 2, 3],
    [4, 0, 5],
    [6, 7, 8]
]
goal_state = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]
# Heuristic function (Manhattan distance)
heuristic_function = manhattan_distance
path = a_star_search(initial_state, goal_state, heuristic_function)
```

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

Output:

Solution found:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

20/12/23

Date 20/12/2023
Page

WEEK - 6

Create a knowledgebase using prepositional logic and show that the given query entails the knowledge base or not

Program :-

```
# Define a function to evaluate logical expressions for the
  first input
def evaluate_first (premise, conclusion):
# Create all possible models for the variables p, q, and r
    models = [
        {'p': False, 'q': False, 'r': False},
        {'p': False, 'q': False, 'r': True},
        {'p': False, 'q': True, 'r': False},
        {'p': False, 'q': True, 'r': True},
        {'p': True, 'q': False, 'r': False},
        {'p': True, 'q': False, 'r': True},
        {'p': True, 'q': True, 'r': False},
        {'p': True, 'q': True, 'r': True}
    ]

    # Check if the premise logically entails the conclusion
    entails = True #initially assume the premise entails the conclusion
    for model in models:
        if evaluate_expression(premise, model) != evaluate_expression(conclusion, model):
            entails = False # if any model disagrees, premise does not entail conclusion
            break
    # Return the result
    return entails

# Define a function to evaluate logical expressions for the second input
def evaluate_second (premise, conclusion):
    # Generate all possible truth assignments to p, q and r
    models = [
        {'p': p, 'q': q, 'r': r}
```

```
        for p in [True, False]
        for q in [True, False]
        for r in [True, False]
    ]
    # Check if the conclusion holds in every model
    where the premise is true
    entails = all (evaluate_expression (premise, model) for
                model in models if evaluate_expression(conclusion,
                model)
                and evaluate_expression (premise, model))
    # Return the result
    return entails

# Define a function to evaluate logical expressions based on
  the given expression and model
def evaluate_expression (expression, model):
    # Evaluate the logical expression recursively using the
    given model
    if isinstance (expression, str):
        return model.get (expression)
    elif isinstance (expression, tuple):
        op = expression [0]
        if op == 'not':
            return not evaluate_expression (expression [1], model)
        elif op == 'and':
            return evaluate_expression (expression [1], model)
                and evaluate_expression (expression [2], model)
        elif op == 'if':
            return (not evaluate_expression (expression [1], model)
                or evaluate_expression (expression [2], model)
        elif op == 'or':
            return evaluate_expression (expression [1], model)
                or evaluate_expression (expression [2], model)
```

```
first_premise = ('and', ('or', 'p', 'q'), ('or', ('not', 'r'), 'p'))
first_conclusion = ('and', 'p', 'r')

second_premise = ('and', ('or', ('not', 'q'), ('not', 'p'), 'r'),
                  ('and', ('not', 'q'), 'p'), 'q')
second_conclusion = 'r'

result_first = evaluate_first(first_premise, first_conclusion)
result_second = evaluate_second(second_premise, second_conclusion)

if result_first:
    print("For the first input: The premise entails the conclusion.")
else:
    print("For the first input: The premise does not entail the conclusion.")

if result_second:
    print("For the second input: The premise entails the conclusion.")
else:
    print("For the second input: The premise does not entail the conclusion.")
```

Output :-

For the first input: The premise does not entail the conclusion

For the second input: The premise entails the conclusion.

17/1/24

classmate
Date 27/12/2023
Page

WEEK - 7

Create a knowledge base using propositional logic and prov
the given query using resolution.

Program :-

```
import re
def main (rules, goal):
    rules = rules.split(' ')
    steps = resolve (rules, goal)
    print('\n Step \t / Clause \t / Derivation \t ')
    print('-' * 30)
    i = 1
    for step in steps:
        print(f' {i}. \t / {step} \t | {steps [step]} \t')
        i += 1
def negate (term):
    return f'~{term}' if term [0] != '~' else term (1)
def reverse (clause):
    if len (clause) > 2:
        t = split_terms (clause)
        return f' {t [1]} v {t [0]} '
    return ''
def split_terms (rule):
    exp = '(~* [PQRS])'
    terms = re. findall (exp, rule)
    return terms
split_terms ('~P v R')
['~P', 'R']
def contradiction (goal, clause):
    contradictions = [f' {goal} v {negate (goal)}', f '{negate
                                (goal)} v {goal}']
    return clause in contradictions or reverse (clause) in
    contradictions
```

```
def resolve (rules, goal):
    temp = rules.copy()
    temp += [negate (goal)]
    steps = dict()
    for rule in temp:
        steps [rule] = 'Given'
    steps [negate (goal)] = 'Negated conclusion'
    i = 0
    while i < len (temp):
        n = len (temp)
        j = (i+1) % n
        clauses = []
        while j != i:
            terms1 = split_terms (temp [i])
            terms2 = split_terms (temp [j])
            for c in terms1:
                if negate (c) in terms2:
                    t1 = [t for t in terms1 if t != c]
                    t2 = [t for t in terms2 if t !=
                                        negate (c)]
                    gen = t1 + t2
                    if len (gen) == 2:
                        if gen [0] != negate (gen [1]):
                            clauses += [f'{gen [0]}
                                        v{gen [1]}']
                    else:
                        if contradiction (goal,
                        f'{gen [0]} v {gen [1]}'):
                            temp.append (f'{gen [0]}
                                        v{gen [1]}')
                            steps [''] = f'
                            Resolved {temp [i]}
                            and {temp [j]} to
                            {temp [-1]}, which is
                            in turn null
```

\nA contradiction is found when {negate(goal)}
is assumed as true. Hence, {goal} is true"
return steps
elif len(gen)==1:
clauses += [f'{gen[0]}']
else:
if contradiction(goal, f'{terms1[0]} v {terms2[0]}'):
temp.append(f"{terms1[0]} v {terms2[0]}")
steps[''] = f" Resolved {temp[i]} and {temp[j]}
to {temp[-1]}, which is in turn null."
\nA contradiction is found when {negate(goal)}
is assumed as true. Hence, {goal} is true
return steps

for clause in clauses:
if clause not in temp and clause[::-1] != reverse(clause)
and reverse(clause) not in temp:
temp.append(clause)
steps[clause] = f'Resolved from {temp[i]}
and {temp[j]}.'

j = (j+1) % n
i += 1

return steps
rules = 'Rv~P Rv~Q ~RvP ~RvQ'
goal = 'R'
main(rules, goal)
rules = 'PvQ ~PvR ~QvR'
goal = 'R'
main(rules, goal)
rules = 'PvQ PvR ~PvR RvS Rv~Q ~Sv~Q'
main(rules, 'R')

Output :-

Step	Clause	Derivation
1.	R v ~P	Given
2.	R v ~Q	Given
3.	~R v P	Given
4.	~R v Q	Given
5.	~R	Negated conclusion
6.		Resolved R v ~P and ~R v P to R v ~R, which is in turn null.

A contradiction is found when ~R is assumed as true. Hence, R is true.

Sneha
7/11/24

classmate
Date 10/1/24
Page

WEEK - 8

Implement unification in first order logic.

Program :-

```
import re
def getAttributes (expression):
    expression = expression.split ("(") [1:]
    expression = "(".join (expression)
    expression = expression [:-1]
    expression = re.split ("(?<!\(.),(?!.\))", expression)
    return expression
def getInitialPredicate (expression):
    return expression.split ("(") [0]
def isConstant (char):
    return char.isupper() and len (char) == 1
def isVariable (char):
    return char.islower() and len (char) == 1
def replaceAttributes (exp, old, new):
    attributes = getAttributes (exp)
    for index, val in enumerate (attributes):
        if val == old:
            attributes [index] = new
    predicate = getInitialPredicate (exp)
    return predicate + "(" + ",".join (attributes) + ")"
def apply (exp, substitutions):
    for substitution in substitutions:
        new, old = substitution
        exp = replaceAttributes (exp, old, new)
    return exp
def checkOccurs (var, exp):
    if exp.find (var) == -1:
        return False
    return True
```

```
def getFirstPart(expression):
    attributes = getAttributes(expression)
    return attributes[0]
def getRemainingPart(expression):
    predicate = getInitialPredicate(expression)
    attributes = getAttributes(expression)
    newExpression = predicate + "(" + ",".join(attributes[1:]) + ")"
    return newExpression
def unify(exp1, exp2):
    if exp1 == exp2:
        return []
    if isConstant(exp1) and isConstant(exp2):
        if exp1 != exp2:
            return False
    if isConstant(exp1):
        return [(exp1, exp2)]
    if isConstant(exp2):
        return [(exp2, exp1)]
    if isVariable(exp1):
        if checkOccurs(exp1, exp2):
            return False
        else:
            return [(exp2, exp1)]
    if isVariable(exp2):
        if checkOccurs(exp2, exp1):
            return False
        else:
            return [(exp1, exp2)]
    if getInitialPredicate(exp1) != getInitialPredicate(exp2):
        print("Predicates do not match. Cannot be unified")
        return False
```

```
attribute count1 = len (get Attributes (exp1))
attribute count2 = len (get Attributes (exp 2))
if attribute count1 != attribute count2 :
    return False
head 1 = get First Part (exp 1)
head 2 = get First Part (exp 2)
initial Substitution = unify (head 1, head 2)
if not initial Substitution :
    return False
if attribute Count1 = = 1 :
    return initial Substitution
tail 1 = get Remaining Part (exp 1)
tail 2 = get Remaining Part (exp 2)
if initial Substitution != [ ]:
    tail 1 = apply (tail 1, initial Substitution)
    tail 2 = apply (tail 2, initial Substitution)
remaining Substitution = unify (tail 1, tail 2)
if not remaining Substitution :
    return False
initial Substitution . extend (remaining Substitution)
return initial Substitution
exp 1 = " knows (f(x), y) "
exp 2 = " knows (J, John) "
substitutions = unify (exp 1, exp 2)
print (" Substitutions : ")
print (substitutions)
exp 1 = " Student (x) "
exp 2 = " Teacher (y) "
substitutions = unify (exp 1, exp 2)
print (" Substitutions : ")
print (substitutions)
```

```
expl = " knows (John, x) "
exp2 = " knows (y, Mother (y)) "
substitutions = unify (expl, exp 2)
print (" Substitutions : ")
print (substitutions)
expl = " like (A, y) "
exp 2 = " like (K, g(x)) "
substitutions = unify (expl, exp 2)
print (" Substitutions : ")
print (substitutions)
```

Output :-

```
Substitutions :
[('J', 'f(x)'), ('John', 'y')]
Predicates do not match. Cannot be unified.
Substitutions :
False
Substitutions :
[('John', 'y'), ('Mother (y)', 'x')]
Substitutions :
False
```

classmate
Date 17/01/24
Page

WEEK - 9

Convert given first order logic statement into Conjunctive Normal Form (CNF)

Program :-

```
import re
def fol_to_cnf (fol):
    statement = fol.replace ("<=>", "-")
    while '-' in statement:
        i = statement.index ('-')
        new_statement = '[' + statement [:i] + '=>' + statement[i+
                        + ']&[' + statement [i+1:] + '=>' +
                        statement [:i] + ']'
        statement = new_statement
    statement = statement.replace ("=>", "-")
    expr = '\[([^]]+)\]'
    statements = re.findall (expr, statement)
    for i, s in enumerate (statements):
        if '[' in s and ']' not in s:
            statements [i] += ']'
    for s in statements:
        statement = statement.replace (s, fol_to_cnf (s))
    while '-' in statement:
        i = statement.index ('-')
        br = statement.index ('[') if '[' in statement else 0
        new_statement = '~' + statement [br:i] + '|' + statement
                        [i+1:]
        statement = statement [:br] + new_statement if br > 0
                    else new_statement
    while '~∀' in statement:
        i = statement.index ('~∀')
        statement = list (statement)
        statement [i], statement [i+1], statement [i+2] = '∃',
        statement = ''.join (statement)  statement [i+2], '~'
    while '~∃' in statement:
        i = statement.index ('~∃')
```

```
        s = list(statement)
        s[i], s[i+1], s[i+2] = 'A', s[i+2], '~'
        statement = ''.join(s)
    statement = statement.replace('~[A', '[~A')
    statement = statement.replace('~[E', '[~E')
    expr = '(~[A|E].)'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))
    expr = '~\[[^]]+\]'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, DeMorgan(s))
    return statement
def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return [m for m in str(matches) if m.isalpha()]
def getPredicates(string):
    expr = '[a-z~]+\([A-Za-z,]+\)'
    return re.findall(expr, string)
def DeMorgan(sentence):
    string = ''.join(list(sentence).copy())
    string = string.replace('~~', '')
    flag = '[' in string
    string = string.replace('~[', '')
    string = string.strip(']')
    for predicate in getPredicates(string):
        string = string.replace(predicate, f'~{predicate}')
    s = list(string)
    for i, c in enumerate(string):
        if c == '|':
            s[i] = '&'
```

```
            elif c == 'f':
                s[i] = 'l'
        string = ''.join(s)
        string = string.replace('~~', '')
        return f'[{string}]' if flag else string
def Skolemization(sentence):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in
                        range(ord('A'), ord('Z')+1)]
    statement = ''.join(list(sentence).copy())
    matches = re.findall('[∀∃].', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, '')
        statements = re.findall('\[\[[^]]+\]]',
                                statement)
        for s in statements:
            statement = statement.replace(s, s[1:-1])
        for predicate in getPredicates(statement):
            attributes = getAttributes(predicate)
            if ''.join(attributes).islower():
                statement = statement.replace(match[1],
                                SKOLEM_CONSTANTS.pop(0))
            else:
                aL = [a for a in attributes if
                      a.islower()]
                aU = [a for a in attributes if
                      not a.islower()][0]
                statement = statement.replace
                (aU, f'{SKOLEM_CONSTANTS.
                        pop(0)}
                ([aL[0] if len(aL)
                else match[1]})')
    return statement
print(Skolemization(fol_to_cnf("animal(y)<=>loves(x,y)")))
print(Skolemization(fol_to_cnf("∀x[∀y[animal(y)=>loves
                (x,y)]]=>[∃z[loves(z,x)]]")))
print(fol_to_cnf("∀x food(x) => likes(john,x)"))
```

Output :-

[~ animal (y) | loves (x, y)] & [~ loves (x, y) | animal (y)]

[animal (G(x)) & ~ loves (x, G(x))] | [loves (F(x), x)]

∃ x ~ food (x) | likes (john, x)

classmate
Date 24/01/24
Page

WEEK - 10

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

Program :-

```
import re
def isVariable(x):
    return len(x) == 1 and x.islower() and x.isalpha()
def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return matches
def getPredicates(string):
    expr = '([a-zA-Z~]+)\([^&|]+\)'
    return re.findall(expr, string)
class Fact:
    def __init__(self, expression):
        self.expression = expression
        predicate, params = self.splitExpression(expression)
        self.predicate = predicate
        self.params = params
        self.result = any(self.getConstants())
    def splitExpression(self, expression):
        predicate = getPredicate(expression)[0]
        params = getAttributes(expression)[0].strip('()').split(',')
        return [predicate, params]
    def getResult(self):
        return self.result
    def getConstants(self):
        return [None if isVariable(c) else c for c in self.params]
    def getVariables(self):
        return [v if isVariable(v) else None for v in self.params]
```

```
    def substitute(self, constants):
        c = constants.copy()
        f = f"{self.predicate}({','.join([constants.pop(p) if isVariable(p) else p for p in self.params])})"
        return Fact(f)
class Implication:
    def __init__(self, expression):
        self.expression = expression
        l = expression.split('=>')
        self.lhs = [Fact(f) for f in l[0].split('&')]
        self.rhs = Fact(l[1])
    def evaluate(self, facts):
        constants = {}
        new_lhs = []
        for fact in facts:
            for val in self.lhs:
                if val.predicate == fact.predicate:
                    for i,v in enumerate(val.getVariables()):
                        if v:
                            constants[v] = fact.getConstants()[i]
                    new_lhs.append(fact)
        predicate, attributes = getPredicates(self.rhs.expression)[0], str(getAttributes(self.rhs.expression)[0])
        for key in constants:
            if constants[key]:
                attributes = attributes.replace(key, constants[key])
        expr = f'{predicate}({attributes})'
        return Fact(expr) if len(new_lhs) and all([f.getResult() for f in new_lhs]) else None
```

```
class KB:
    def __init__(self):
        self.facts = set()
        self.implications = set()
    def tell(self, e):
        if '=>' in e:
            self.implications.add(Implication(e))
        else:
            self.facts.add(Fact(e))
        for i in self.implications:
            res = i.evaluate(self.facts)
            if res:
                self.facts.add(res)
    def query(self, e):
        facts = set([f.expression for f in self.facts])
        i = 1
        print(f'Querying {e}:')
        for f in facts:
            if Fact(f).predicate == Fact(e).predicate:
                print(f'\t{i+1}. {f}')

kb_ = KB()
kb_.tell('king(x) & greedy(x) => evil(x)')
kb_.tell('king(John)')
kb_.tell('greedy(John)')
kb_.tell('king(Richard)')
kb_.query('evil(x)')
```

Output :-

Querying evil(x):

1. evil(John)

1. Implement Tic –Tac –Toe Game

```
import math

def print_board(board):
    for i in range(len(board)):
        for j in range(len(board[i])):
            print(board[i][j], end='')
            if j < len(board[i]) - 1:
                print('|', end='')
        print()
        if i < len(board) - 1:
            print('-'*5)
    print()

def check_winner(board):
    # Check rows, columns, and diagonals for a winner
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != ' ':
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != ' ':
            return board[0][i]
    if board[0][0] == board[1][1] == board[2][2] != ' ':
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != ' ':
        return board[0][2]
    return None

def get_empty_cells(board):
    # Returns a list of empty cells in the board
    return [(i, j) for i in range(3) for j in range(3) if board[i][j] == '
']

def minimax(board, depth, is_maximizing):
    winner = check_winner(board)
    if winner:
        return 10 - depth if winner == 'X' else -10 + depth
    elif not get_empty_cells(board):
        return 0

    if is_maximizing:
        best_score = -math.inf
        for i, j in get_empty_cells(board):
            board[i][j] = 'X'
            score = minimax(board, depth + 1, False)
            board[i][j] = ' '
```

```
            best_score = max(score, best_score)
        return best_score
    else:
        best_score = math.inf
        for i, j in get_empty_cells(board):
            board[i][j] = 'O'
            score = minimax(board, depth + 1, True)
            board[i][j] = ' '
            best_score = min(score, best_score)
        return best_score

def best_move(board):
    best_score = -math.inf
    move = None
    for i, j in get_empty_cells(board):
        board[i][j] = 'X'
        score = minimax(board, 0, False)
        board[i][j] = ' '
        if score > best_score:
            best_score = score
            move = (i, j)
    return move

def play_game():
    board = [[' ' for _ in range(3)] for _ in range(3)]
    print("Welcome to Tic Tac Toe!")
    print_board(board)

    while not check_winner(board) and get_empty_cells(board):
        user_move = input("Enter your move (row and column separated by a
space): ")
        x, y = map(int, user_move.split())
        if board[x][y] == ' ':
            board[x][y] = 'O'
            print_board(board)
        else:
            print("Invalid move. Try again.")
            continue

        if not get_empty_cells(board):
            break

        computer_move = best_move(board)
        board[computer_move[0]][computer_move[1]] = 'X'
        print("Computer's move:")
        print_board(board)
```

```
    winner = check_winner(board)
    if winner:
        print(f"Player {winner} wins!")
    else:
        print("It's a tie!")

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

OUTPUT

```
Welcome to Tic Tac Toe!
 | |
-----
 | |
-----
 | |

Enter your move (row and column separated by a space): 2 2
 | |
-----
 | |
-----
 | |O

Computer's move:
 | |
-----
 |X|
-----
 | |O

Enter your move (row and column separated by a space): 1 2
 | |
-----
 |X|O
-----
 | |O

Computer's move:
 | |X
-----
 |X|O
-----
 | |O

Enter your move (row and column separated by a space): 2 0
 | |X
-----
 |X|O
-----
O| |O

Computer's move:
 | |X
-----
 |X|O
-----
O|X|O

Enter your move (row and column separated by a space): 1 1
Invalid move. Try again.
Enter your move (row and column separated by a space): 0 1
 |O|X
-----
 |X|O
-----
O|X|O
```

```
Computer's move:
X|O|X
-----
 |X|O
-----
O|X|O

Enter your move (row and column separated by a space): 1 0
X|O|X
-----
O|X|O
-----
O|X|O

It's a tie!
```

2. Implement vaccum cleaner agent

```
def printInformation(location):
    print("Location " + location + " is Dirty.")
    print("Cost for CLEANING " + location + ": 1")
    print("Location " + location + " has been Cleaned.")


def vacuumCleaner(goalState, currentState, location):
    # printing necessary data
    print("Goal State Required:", goalState)
    print("Vacuum is placed in Location " + location)

    # cleaning locations
    totalCost = 0

    while (currentState != goalState):
        if (location == "A"):
            # cleaning
            if (currentState["A"] == 1):
                currentState["A"] = 0
                totalCost += 1
                printInformation("A")
            # moving
            elif (currentState["B"] == 1 ):
                print("Moving right to the location B.\nCost for moving
RIGHT: 1")
                location = "B"
                totalCost += 1

        elif (location == "B"):
            # cleaning
            if (currentState["B"] == 1):
                currentState["B"] = 0
                totalCost += 1
                printInformation("B")
            # moving
            elif (currentState["A"] == 1):
                print("Moving left to the location A.\nCost for moving LEFT:
1")
                location = "A"
                totalCost += 1

    print("GOAL STATE:", currentState)
    return totalCost
```

```
# declaring dictionaries
goalState = {"A": 0, "B": 0}
currentState = {"A": -1, "B": -1}

# taking input from user
location = input("Enter Location of Vacuum (A/B): ");
currentState["A"] = int(input("Enter status of A (0/1): "))
currentState["B"] = int(input("Enter status of B (0/1): "))

# calling function
totalCost = vacuumCleaner(goalState, currentState, location)
print("Performance Measurement:", totalCost)
```

OUTPUT

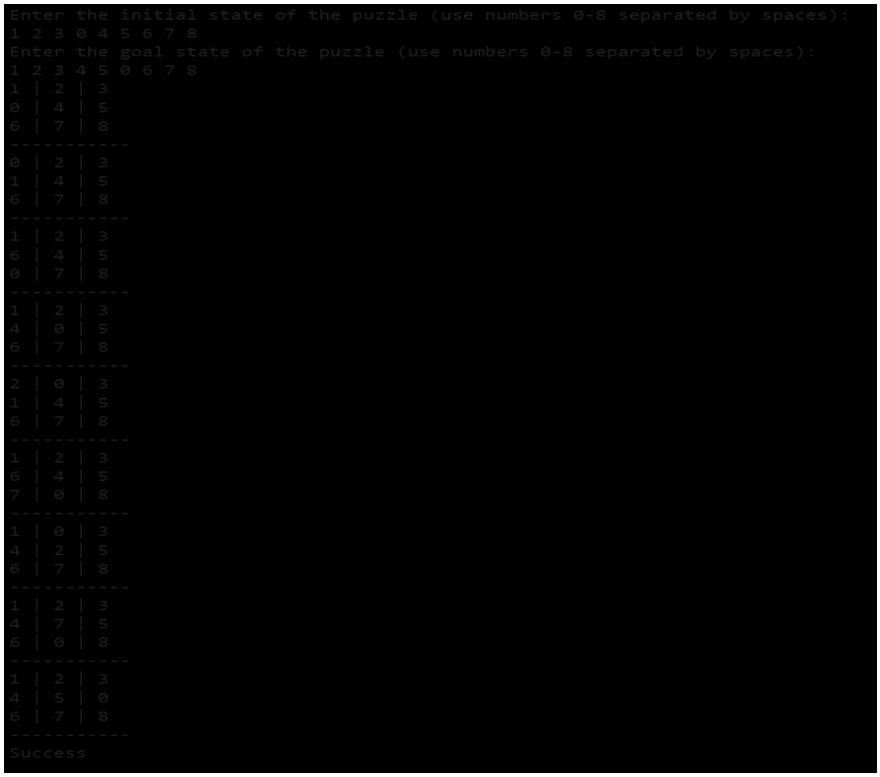
```
Enter Location of Vacuum (A/B): B
Enter status of A (0/1): 1
Enter status of B (0/1): 1
Goal State Required: {'A': 0, 'B': 0}
Vacuum is placed in Location B
Location B is Dirty.
Cost for CLEANING B: 1
Location B has been Cleaned.
Moving left to the location A.
Cost for moving LEFT: 1
Location A is Dirty.
Cost for CLEANING A: 1
Location A has been Cleaned.
GOAL STATE: {'A': 0, 'B': 0}
Performance Measurement: 3
```

3. Analyse 8 Puzzle problem and implement the same using Breadth First Search Algorithm

```
def bfs(src, target):
    queue = []
    queue.append(src)
    visited = set()

    while queue:
        source = queue.pop(0)
        visited.add(tuple(source))  # Store visited states as tuples for
faster lookup

        print(source[0], '|', source[1], '|', source[2])
        print(source[3], '|', source[4], '|', source[5])
        print(source[6], '|', source[7], '|', source[8])
        print("-----------")

        if source == target:
            print("Success")
            return

        poss_moves_to_do = possible_moves(source, visited)
        for move in poss_moves_to_do:
            queue.append(move)

def possible_moves(state, visited_states):
    b = state.index(0)
    d = []

    # Add possible directions to move based on the position of the empty
cell
    if b not in [0, 1, 2]:
        d.append('u')
    if b not in [6, 7, 8]:
        d.append('d')
    if b not in [0, 3, 6]:
        d.append('l')
    if b not in [2, 5, 8]:
        d.append('r')

    pos_moves_it_can = []

    for i in d:
        pos_moves_it_can.append(gen(state, i, b))
```

```
    # Return possible moves that have not been visited yet
    return [move_it_can for move_it_can in pos_moves_it_can if
tuple(move_it_can) not in visited_states]

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

# Taking input for initial and goal states
print("Enter the initial state of the puzzle (use numbers 0-8 separated by
spaces):")
src = list(map(int, input().split()))

print("Enter the goal state of the puzzle (use numbers 0-8 separated by
spaces):")
target = list(map(int, input().split()))

bfs(src, target)
```

OUTPUT


Enter the initial state of the puzzle (use numbers 0-8 separated by spaces):
1 2 3 0 4 5 6 7 8
Enter the goal state of the puzzle (use numbers 0-8 separated by spaces):
1 2 3 4 5 0 6 7 8
1 | 2 | 3
0 | 4 | 5
6 | 7 | 8
---
0 | 2 | 3
1 | 4 | 5
6 | 7 | 8
---
1 | 2 | 3
6 | 4 | 5
0 | 7 | 8
---
1 | 2 | 3
4 | 0 | 5
6 | 7 | 8
---
2 | 0 | 3
1 | 4 | 5
6 | 7 | 8
---
1 | 2 | 3
6 | 4 | 5
7 | 0 | 8
---
1 | 0 | 3
4 | 2 | 5
6 | 7 | 8
---
1 | 2 | 3
4 | 7 | 5
6 | 0 | 8
---
1 | 2 | 3
4 | 5 | 0
6 | 7 | 8

Success

4. Analyse Iterative Deepening Search Algorithm. Demonstrate how 8 Puzzle problem could be solved using this algorithm

```
def dfs(src,target,limit,visited_states):
    if src == target:
        return True
    if limit <= 0:
        return False
    visited_states.append(src)
    moves = possible_moves(src,visited_states)
    for move in moves:
        if dfs(move, target, limit-1, visited_states):
            return True
    return False

def possible_moves(state,visited_states):
    b = state.index(-1)
    d = []
    if b not in [0,1,2]:
        d += 'u'
    if b not in [6,7,8]:
        d += 'd'
    if b not in [2,5,8]:
        d += 'r'
    if b not in [0,3,6]:
        d += 'l'
    pos_moves = []
    for move in d:
        pos_moves.append(gen(state,move,b))
    return [move for move in pos_moves if move not in visited_states]

def gen(state, move, blank):
    temp = state.copy()
    if move == 'u':
        temp[blank-3], temp[blank] = temp[blank], temp[blank-3]
    if move == 'd':
        temp[blank+3], temp[blank] = temp[blank], temp[blank+3]
    if move == 'r':
        temp[blank+1], temp[blank] = temp[blank], temp[blank+1]
    if move == 'l':
        temp[blank-1], temp[blank] = temp[blank], temp[blank-1]
    return temp

def iddfs(src,target,depth):
    for i in range(depth):
        visited_states = []
        if dfs(src,target,i+1,visited_states):
```

```
            return True, i+1
    return False

print("Enter the initial state of the puzzle (use numbers 0-8 separated by spaces):")
src = list(map(int, input().split()))

print("Enter the goal state of the puzzle (use numbers 0-8 separated by spaces):")
target = list(map(int, input().split()))
depth = 8
iddfs(src, target, depth)
```

OUTPUT

```
Enter the initial state of the puzzle (use numbers 0-8 separated by spaces):
1 2 3 -1 4 5 6 7 8
Enter the goal state of the puzzle (use numbers 0-8 separated by spaces):
1 2 3 6 4 5 7 8 -1
(True, 3)
```

5. Implement A* search algorithm

```
class Node:
    def __init__(self,data,level,fval):

        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):

        x,y = self.find(self.data,'_')

        val_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data,x,y,i[0],i[1])
            if child is not None:
                child_node = Node(child,self.level+1,0)
                children.append(child_node)
        return children

    def shuffle(self,puz,x1,y1,x2,y2):

        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 <
len(self.data):
            temp_puz = []
            temp_puz = self.copy(puz)
            temp = temp_puz[x2][y2]
            temp_puz[x2][y2] = temp_puz[x1][y1]
            temp_puz[x1][y1] = temp
            return temp_puz
        else:
            return None


    def copy(self,root):

        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
            temp.append(t)
        return temp

    def find(self,puz,x):
```

```
        for i in range(0,len(self.data)):
            for j in range(0,len(self.data)):
                if puz[i][j] == x:
                    return i,j


class Puzzle:
    def __init__(self,size):

        self.n = size
        self.open = []
        self.closed = []

    def accept(self):

        puz = []
        for i in range(0,self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz

    def f(self,start,goal):

        return self.h(start.data,goal)+start.level

    def h(self,start,goal):

        temp = 0
        for i in range(0,self.n):
            for j in range(0,self.n):
                if start[i][j] != goal[i][j] and start[i][j] != '_':
                    temp += 1
        return temp


    def process(self):

        print("Enter the start state matrix \n")
        start = self.accept()
        print("Enter the goal state matrix \n")
        goal = self.accept()

        start = Node(start,0,0)
        start.fval = self.f(start,goal)

        self.open.append(start)
```

```
        print("\n\n")
        while True:
            cur = self.open[0]
            print("")
            print("   | ")
            print("   | ")
            print(" \\\'/ \n")
            for i in cur.data:
                for j in i:
                    print(j,end=" ")
                print("")

            if(self.h(cur.data,goal) == 0):
                break
            for i in cur.generate_child():
                i.fval = self.f(i,goal)
                self.open.append(i)
            self.closed.append(cur)
            del self.open[0]


            self.open.sort(key = lambda x:x.fval,reverse=False)


puz = Puzzle(3)
puz.process()
```

OUTPUT


Enter the start state matrix
1 2 3
_ 4 6
7 5 8
Enter the goal state matrix
1 2 3
4 5 6
7 8 _
1 2 3
_ 4 6
7 5 8
1 2 3
4 _ 6
7 5 8
1 2 3
4 5 6
7 _ 8
1 2 3
4 5 6
7 8 _

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

```
def tell(kb, rule):
    kb.append(rule)

combinations = [(True, True, True), (True, True, False),
                (True, False, True), (True, False, False),
                (False, True, True), (False, True, False),
                (False, False, True), (False, False, False)]


def ask(kb, q):
    for c in combinations:
        s = r1(c)
        f = q(c)
        print(s, f)
        if s != f and s != False:
            return 'Does not entail'
    return 'Entails'


kb = []

rule_str = input("Enter Rule 1 as a lambda function (e.g., lambda x: x[0] or
x[1] and (x[0] and x[1]): ")
r1 = eval(rule_str)
tell(kb, r1)

query_str = input("Enter Query as a lambda function (e.g., lambda x: x[0]
and x[1] and (x[0] or x[1]): ")
q = eval(query_str)


result = ask(kb, q)
print(result)
```

OUTPUT 1

```
Enter Rule 1 as a lambda function (e.g., lambda x: x[0] or x[1] and (x[0] and x[1]): lambda x: (not x[1] or not x[0] or x[2]) and (not x[1] and
Enter Query as a lambda function (e.g., lambda x: x[0] and x[1] and (x[0] or x[1]): lambda x: x[2]
False True
False False
False True
False False
False True
False False
False True
False False
Entails
```

OUTPUT 2

```
Enter Rule 1 as a lambda function (e.g., lambda x: x[0] or x[1] and (x[0] and x[1]): lambda x: (x[0] or x[1]) and (not x[2] or x[0])
Enter Query as a lambda function (e.g., lambda x: x[0] and x[1] and (x[0] or x[1]): lambda x: x[0] and x[2]
True True
True False
Does not entail
```

7. Create a knowledge base using prepositional logic and prove the given query using resolution

```
import re

def main():
    rules = input("Enter the rules (space-separated): ")
    goal = input("Enter the goal: ")
    rules = rules.split(' ')
    steps = resolve(rules, goal)
    print('\nStep\t|Clause\t|Derivation\t')
    print('-' * 30)
    i = 1
    for step in steps:
        print(f' {i}.\t| {step}\t| {steps[step]}\t')
        i += 1


def negate(term):
    return f'~{term}' if term[0] != '~' else term[1]


def split_terms(rule):
    exp = '(~*[PQRS])'
    terms = re.findall(exp, rule)
    return terms


def contradiction(goal, clause):
    contradictions = [ f'{goal}v{negate(goal)}', f'{negate(goal)}v{goal}']
    return clause in contradictions
def resolve(rules, goal):
    temp = rules.copy()
    temp += [negate(goal)]
    steps = dict()
    for rule in temp:
        steps[rule] = 'Given.'
    steps[negate(goal)] = 'Negated conclusion.'
    i = 0
    while i < len(temp):
        n = len(temp)
        j = (i + 1) % n
        clauses = []
        while j != i:
            terms1 = split_terms(temp[i])
            terms2 = split_terms(temp[j])
            for c in terms1:
                if negate(c) in terms2:
                    t1 = [t for t in terms1 if t != c]
                    t2 = [t for t in terms2 if t != negate(c)]
                    gen = t1 + t2
                    if len(gen) == 2:
                        if gen[0] != negate(gen[1]):
                            clauses += [f'{gen[0]}v{gen[1]}']
```

```
                            if contradiction(goal,f'{gen[0]}v{gen[1]}'):
                                temp.append(f'{gen[0]}v{gen[1]}')
                                steps[''] = f"Resolved {temp[i]} and {temp[j]}
to {temp[-1]}, which is in turn null. \
                                \nA contradiction is found when {negate(goal)}
is assumed as true. Hence, {goal} is true."
                                return steps
                    elif len(gen) == 1:
                        clauses += [f'{gen[0]}']
                    else:
                        if contradiction(goal,f'{terms1[0]}v{terms2[0]}'):
                            temp.append(f'{terms1[0]}v{terms2[0]}')
                            steps[''] = f"Resolved {temp[i]} and {temp[j]} to
{temp[-1]}, which is in turn null. \
                            \nA contradiction is found when {negate(goal)} is
assumed as true. Hence, {goal} is true."
                            return steps
            for clause in clauses:
                if clause not in temp :
                    temp.append(clause)
                    steps[clause] = f'Resolved from {temp[i]} and {temp[j]}.'
            j = (j + 1) % n
        i += 1
    return steps
if __name__ == "__main__":
    main()
```

OUTPUT

```
Enter the rules (space-separated): Rv~P Rv~Q ~RvP ~RvQ
Enter the goal: R

Step    |Clause |Derivation
------------------------------
 1.     | Rv~P  | Given.
 2.     | Rv~Q  | Given.
 3.     | ~RvP  | Given.
 4.     | ~RvQ  | Given.
 5.     | ~R    | Negated conclusion.
 6.     |       | Resolved Rv~P and ~RvP to Rv~R, which is in turn null.
A contradiction is found when ~R is assumed as true. Hence, R is true.
```

8. Implement unification in first order logic

```
import re

def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression[:-1]
    expression = re.split("(?<!\(.),(?!.\))", expression)
    return expression

def getInitialPredicate(expression):
    return expression.split("(")[0]

def isConstant(char):
    return char.isupper() and len(char) == 1

def isVariable(char):
    return char.islower() and len(char) == 1

def replaceAttributes(exp, old, new):
    attributes = getAttributes(exp)
    for index, val in enumerate(attributes):
        if val == old:
            attributes[index] = new
    predicate = getInitialPredicate(exp)
    return predicate + "(" + ",".join(attributes) + ")"

def apply(exp, substitutions):
    for substitution in substitutions:
        new, old = substitution
        exp = replaceAttributes(exp, old, new)
    return exp

def checkOccurs(var, exp):
    if exp.find(var) == -1:
        return False
    return True


def getFirstPart(expression):
    attributes = getAttributes(expression)
    return attributes[0]


def getRemainingPart(expression):
    predicate = getInitialPredicate(expression)
    attributes = getAttributes(expression)
    newExpression = predicate + "(" + ",".join(attributes[1:]) + ")"
    return newExpression

def unify(exp1, exp2):
    if exp1 == exp2:
```

```
        return []

    if isConstant(exp1) and isConstant(exp2):
        if exp1 != exp2:
            return False

    if isConstant(exp1):
        return [(exp1, exp2)]

    if isConstant(exp2):
        return [(exp2, exp1)]

    if isVariable(exp1):
        if checkOccurs(exp1, exp2):
            return False
        else:
            return [(exp2, exp1)]

    if isVariable(exp2):
        if checkOccurs(exp2, exp1):
            return False
        else:
            return [(exp1, exp2)]

    if getInitialPredicate(exp1) != getInitialPredicate(exp2):
        print("Predicates do not match. Cannot be unified")
        return False

    attributeCount1 = len(getAttributes(exp1))
    attributeCount2 = len(getAttributes(exp2))
    if attributeCount1 != attributeCount2:
        return False

    head1 = getFirstPart(exp1)
    head2 = getFirstPart(exp2)
    initialSubstitution = unify(head1, head2)
    if not initialSubstitution:
        return False
    if attributeCount1 == 1:
        return initialSubstitution

    tail1 = getRemainingPart(exp1)
    tail2 = getRemainingPart(exp2)

    if initialSubstitution != []:
        tail1 = apply(tail1, initialSubstitution)
        tail2 = apply(tail2, initialSubstitution)

    remainingSubstitution = unify(tail1, tail2)
    if not remainingSubstitution:
        return False
```

```
    initialSubstitution.extend(remainingSubstitution)
    return initialSubstitution


exp1 = input("Enter the first expression: ")
exp2 = input("Enter the second expression: ")


substitutions = unify(exp1, exp2)


print("Substitutions:")
print(substitutions)
```

OUTPUT

```
Enter the first expression: knows(f(x),y)
Enter the second expression: knows(J,John)
Substitutions:
[('J', 'f(x)'), ('John', 'y')]
```

9. Convert a given first order logic statement into Conjunctive Normal Form (CNF)

```
import re

def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return [m for m in str(matches) if m.isalpha()]

def getPredicates(string):
    expr = '[a-z~]+\([A-Za-z,]+\)'
    return re.findall(expr, string)

def DeMorgan(sentence):
    string = ''.join(list(sentence).copy())
    string = string.replace('~~','')
    flag = '[' in string
    string = string.replace('~[','')
    string = string.strip(']')
    for predicate in getPredicates(string):
        string = string.replace(predicate, f'~{predicate}')
    s = list(string)
    for i, c in enumerate(string):
        if c == '|':
            s[i] = '&'
        elif c == '&':
            s[i] = '|'
    string = ''.join(s)
    string = string.replace('~~','')
    return f'[{string}]' if flag else string

def Skolemization(sentence):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)]
    statement = ''.join(list(sentence).copy())
    matches = re.findall('[∀∃].', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, '')
        statements = re.findall('\[\[^]]+\]]', statement)
        for s in statements:
            statement = statement.replace(s, s[1:-1])
        for predicate in getPredicates(statement):
            attributes = getAttributes(predicate)
            if ''.join(attributes).islower():
                statement = statement.replace(match[1],SKOLEM_CONSTANTS.pop(0))
            else:
                aL = [a for a in attributes if a.islower()]
                aU = [a for a in attributes if not a.islower()][0]
                statement = statement.replace(aU,
f'{SKOLEM_CONSTANTS.pop(0)}({aL[0] if len(aL) else match[1]})')
    return statement

def fol_to_cnf(fol):
```

```
    statement = fol.replace("<=>", "_")
    while '_' in statement:
        i = statement.index('_')
        new_statement = '[' + statement[:i] + '=>' + statement[i+1:] + ']&['+
statement[i+1:] + '=>' + statement[:i] + ']'
        statement = new_statement
    statement = statement.replace("=>", "-")
    expr = '\[([^]]+)\]'
    statements = re.findall(expr, statement)
    for i, s in enumerate(statements):
        if '[' in s and ']' not in s:
            statements[i] += ']'
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))
    while '-' in statement:
        i = statement.index('-')
        br = statement.index('[') if '[' in statement else 0
        new_statement = '~' + statement[br:i] + '|' + statement[i+1:]
        statement = statement[:br] + new_statement if br > 0 else new_statement
    while '~∀' in statement:
        i = statement.index('~∀')
        statement = list(statement)
        statement[i], statement[i+1], statement[i+2] = '∃', statement[i+2], '~'
        statement = ''.join(statement)
    while '~∃' in statement:
        i = statement.index('~∃')
        s = list(statement)
        s[i], s[i+1], s[i+2] = '∀', s[i+2], '~'
        statement = ''.join(s)
    statement = statement.replace('~[∀','[~∀')
    statement = statement.replace('~[∃','[~∃')
    expr = '(~[∀|∃].)'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))
    expr = '~\[[^]]+\]'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, DeMorgan(s))
    return statement



print(Skolemization(fol_to_cnf("animal(y)<=>loves(x,y)")))
print(Skolemization(fol_to_cnf("∀x[∀y[animal(y)=>loves(x,y)]]=>[∃z[loves(z,x)]]"
)))
print(fol_to_cnf("[american(x)&weapon(y)&sells(x,y,z)&hostile(z)]=>criminal(x)")
)
```

OUTPUT

```
[~animal(y)|loves(x,y)]&[~loves(x,y)|animal(y)]
[animal(G(x))&~loves(x,G(x))]|[loves(F(x),x)]
[~american(x)|~weapon(y)|~sells(x,y,z)|~hostile(z)]|criminal(x)
```

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

```
import re

def isVariable(x):
    return len(x) == 1 and x.islower() and x.isalpha()

def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return matches

def getPredicates(string):
    expr = '([a-z~]+)\([^&|]+\)'
    return re.findall(expr, string)

class Fact:
    def __init__(self, expression):
        self.expression = expression
        predicate, params = self.splitExpression(expression)
        self.predicate = predicate
        self.params = params
        self.result = any(self.getConstants())

    def splitExpression(self, expression):
        predicate = getPredicates(expression)[0]
        params = getAttributes(expression)[0].strip('()').split(',')
        return [predicate, params]

    def getResult(self):
        return self.result

    def getConstants(self):
        return [None if isVariable(c) else c for c in self.params]

    def getVariables(self):
        return [v if isVariable(v) else None for v in self.params]

class Implication:
    def __init__(self, expression):
        self.expression = expression
        l = expression.split('=>')
        self.lhs = [Fact(f) for f in l[0].split('&')]
        self.rhs = Fact(l[1])

    def evaluate(self, facts):
        constants = {}
        new_lhs = []
        for fact in facts:
            for val in self.lhs:
                if val.predicate == fact.predicate:
                    for i, v in enumerate(val.getVariables()):
```

```
                        if v:
                            constants[v] = fact.getConstants()[i]
                    new_lhs.append(fact)
        predicate, attributes = getPredicates(self.rhs.expression)[0],
str(getAttributes(self.rhs.expression)[0])
        for key in constants:
            if constants[key]:
                attributes = attributes.replace(key, constants[key])
        expr = f'{predicate}{attributes}'
        return Fact(expr) if len(new_lhs) and all([f.getResult() for f in
new_lhs]) else None

class KB:
    def __init__(self):
        self.facts = set()
        self.implications = set()

    def tell(self, e):
        if '=>' in e:
            self.implications.add(Implication(e))
        else:
            self.facts.add(Fact(e))
        for i in self.implications:
            res = i.evaluate(self.facts)
            if res:
                self.facts.add(res)

    def query(self, e):
        facts = set([f.expression for f in self.facts])
        i = 1
        print(f'Querying {e}:')
        for f in facts:
            if Fact(f).predicate == Fact(e).predicate:
                print(f'\t{i}. {f}')
                i += 1

    def display(self):
        print("All facts: ")
        for i, f in enumerate(set([f.expression for f in self.facts])):
            print(f'\t{i+1}. {f}')
kb = KB()
kb.tell('missile(x)=>weapon(x)')
kb.tell('missile(M1)')
kb.tell('enemy(x,America)=>hostile(x)')
kb.tell('american(West)')
kb.tell('enemy(Nono,America)')
kb.tell('owns(Nono,M1)')
kb.tell('missile(x)&owns(Nono,x)=>sells(West,x,Nono)')
kb.tell('american(x)&weapon(y)&sells(x,y,z)&hostile(z)=>criminal(x)')
kb.query('criminal(x)')
kb.display()
```

OUTPUT

```
Querying criminal(x):
        1. criminal(West)
All facts:
        1. enemy(Nono,America)
        2. weapon(M1)
        3. owns(Nono,M1)
        4. missile(M1)
        5. criminal(West)
        6. hostile(Nono)
        7. sells(West,M1,Nono)
        8. american(West)
```