Artificial Intelligence Lab Report

Submitted by

Vinayak Halavoor(1BM22CS328)
Batch: 3

Course: Artificial Intelligence
Course Code: 23CS5PCAIP
Sem & Section: 5F

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B. M. S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
2023-2024

Table of contents

Program 1 - Tic Tac toe

Algorithm

Lab - 1
4/10/2024

Week - 1

Tic - Tac - Toe

```
function minimax (node, depth, isMaximizingPlayer):
if node is a terminal state:
  return evaluate (node)

if isMaximizingPlayer:
    best value = -infinity
    for each child in node:
        value = minimax (child, depth+1, false)
       bestvalue = max (best value, value)
       return best value

else:
    bestvalue = +infinity
    for each child in node:
        value = minimax (child, depth+1, true)
        bestvalue = min (bestvalue, value)
        return bestvalue
```

Code :

```
board={1:' ',2:' ',3:' ',
      4:' ',5:' ',6:' ',
      7:' ',8:' ',9:' '
}
```

```
def printBoard(board):
  print(board[1]+'|'+board[2]+'|'+board[3])
  print('-+-+-')
  print(board[4] + '|' + board[5] + '|' + board[6])
  print('-+-+-')
  print(board[7] + '|' + board[8] + '|' + board[9])
  print('\n')

def spaceFree(pos):
  if(board[pos]==' '):
     return True
  else:
     return False

def checkWin():
  if(board[1]==board[2] and board[1]==board[3] and board[1]!=' '):
     return True
  elif(board[4]==board[5] and board[4]==board[6] and board[4]!=' '):
     return True
  elif(board[7]==board[8] and board[7]==board[9] and board[7]!=' '):
     return True
  elif (board[1] == board[5] and board[1] == board[9] and board[1] != ' '):
     return True
  elif (board[3] == board[5] and board[3] == board[7] and board[3] != ' '):
     return True
  elif (board[1] == board[4] and board[1] == board[7] and board[1] != ' '):
     return True
  elif (board[2] == board[5] and board[2] == board[8] and board[2] != ' '):
     return True
  elif (board[3] == board[6] and board[3] == board[9] and board[3] != ' '):
     return True
  else:
     return False

def checkMoveForWin(move):
  if (board[1]==board[2] and board[1]==board[3] and board[1] ==move):
     return True
  elif (board[4]==board[5] and board[4]==board[6] and board[4] ==move):
     return True
  elif (board[7]==board[8] and board[7]==board[9] and board[7] ==move):
     return True
  elif (board[1]==board[5] and board[1]==board[9] and board[1] ==move):
     return True
  elif (board[3]==board[5] and board[3]==board[7] and board[3] ==move):
     return True
```

```
    elif (board[1]==board[4] and board[1]==board[7] and board[1] ==move):
        return True
    elif (board[2]==board[5] and board[2]==board[8] and board[2] ==move):
        return True
    elif (board[3]==board[6] and board[3]==board[9] and board[3] ==move):
        return True
    else:
        return False

def checkDraw():
    for key in board.keys():
        if (board[key]==' '):
            return False
    return True
def insertLetter(letter, position):
    if (spaceFree(position)):
        board[position] = letter
        printBoard(board)

        if (checkDraw()):
            print('Draw!')
        elif (checkWin()):
            if (letter == 'X'):
                print('Bot wins!')
            else:
                print('You win!')
        return

    else:
        print('Position taken, please pick a different position.')
        position = int(input('Enter new position: '))
        insertLetter(letter, position)
        return

player = 'O'
bot ='X'

def playerMove():
    position=int(input('Enter position for O:'))
    insertLetter(player, position)
    return

def compMove():
    bestScore=-1000
    bestMove=0
```

```
    for key in board.keys():
        if (board[key]==' '):
            board[key]=bot
            score = minimax(board, False)
            board[key] = ' '
            if (score > bestScore):
                bestScore = score
                bestMove = key

    insertLetter(bot, bestMove)
    return
def minimax(board, isMaximizing):
    if (checkMoveForWin(bot)):
        return 1
    elif (checkMoveForWin(player)):
        return -1
    elif (checkDraw()):
        return 0

    if isMaximizing:
        bestScore = -1000
        for key in board.keys():
            if board[key] == ' ':
                board[key] = bot
                score = minimax(board, False)
                board[key] = ' '
                if (score > bestScore):
                    bestScore = score
        return bestScore
    else:
        bestScore = 1000

        for key in board.keys():
            if board[key] == ' ':
                board[key] = player
                score = minimax(board, True)
                board[key] = ' '
                if (score < bestScore):
                    bestScore = score
        return bestScore

while not checkWin():
    compMove()
    playerMove()
```

Output :

```
V Dhanush Reddy
1BM22CS324
X| |
-+-+-
 | |
-+-+-
 | |


Enter position for O: 3
X| |O
-+-+-
 | |
-+-+-
 | |


X| |O
-+-+-
X| |
-+-+-
 | |


Enter position for O: 9
X| |O
-+-+-
X| |
-+-+-
 | |O


X| |O
-+-+-
X| |X
-+-+-
 | |O
```

```
X| |O
-+-+-
X| |X
-+-+-
 | |O


Enter position for O: 2
X|O|O
-+-+-
X| |X
-+-+-
 | |O


X|O|O
-+-+-
X|X|X
-+-+-
 | |O


Bot wins!
```

Program 2:Vacuum Cleaner

Algorithm

Week - 2
18/10/2024

Vaccum Cleaner Agent :

```
FUNCTION VACCUM_WORLD()
    INITIALIZE goal_state as {'A': '0', 'B': '0'}
    // 0: Clean , 1: Dirty.
    SET cost to 0

    PROMPT for vacuum location (A/B) as location_input
    PROMPT for status of location_input as status_input.
    SET other_location to 'B' if A else 'A'
    PROMPT for status of other_location as status_input_complement
    IF location_input is 'A':
        IF status_input is '1':
            SET goal_state ['A'] to '0' // clean A.
            INCREAMENT cost by 2
            IF status_input_complement is '1':
                set goal_state ['B'] to '0'
        ELSE IF status_input_complement is '1':
            INCREAMENT cost by 1
            SET goal_state ['B'] to '0'
    ELSE:
        IF status_input is 1 :
            SET goal_state ['B'] to '0'
            INCREAMENT cost by 2
            IF status_input_complement is '1':
                SET goal_state ['A'] to '0'
        ELSE_IF status_input_complement is '1':
            INCREAMENT cost by '1'
            SET goal_state ['A'] to '0'
    PRINT goal_state and cost.
END FUNCTION
call vaccum_~~agent~~.coould
```

18/10/2024

Code:

```
def vacuum_world():
    # Initializing goal_state
    # 0 indicates Clean and 1 indicates Dirty
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter Location of Vacuum (A or B): ").strip().upper()  #
User input for vacuum location
    status_input = input(f"Enter status of {location_input} (0 for Clean, 1 for Dirty):
").strip()  # Status of the current location
    other_location = 'B' if location_input == 'A' else 'A'
    status_input_complement = input(f"Enter status of {other_location} (0 for Clean, 1
for Dirty): ").strip()  # Status of the other room

    print("Initial Location Condition: " + str(goal_state))

    # Helper function to clean a location
    def clean(location):
        nonlocal cost
        goal_state[location] = '0'
        cost += 1  # Cost for sucking dirt
        print(f"Location {location} has been Cleaned. Cost: {cost}")

    # Main logic
    if location_input == 'A':
        print("Vacuum is placed in Location A.")
        if status_input == '1':
            print("Location A is Dirty.")
            clean('A')
            if status_input_complement == '1':
                print("Location B is Dirty.")
                print("Moving right to Location B.")
                cost += 1  # Cost for moving right
                print(f"COST for moving RIGHT: {cost}")
```

```
            clean('B')
        else:
            print("Location B is already clean.")
    else:
        print("Location A is already clean.")
        if status_input_complement == '1':
            print("Location B is Dirty.")
            print("Moving right to Location B.")
            cost += 1  # Cost for moving right
            print(f"COST for moving RIGHT: {cost}")
            clean('B')
        else:
            print("Location B is already clean.")

else:  # Vacuum is placed in Location B
    print("Vacuum is placed in Location B.")
    if status_input == '1':
        print("Location B is Dirty.")
        clean('B')
        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving left to Location A.")
            cost += 1  # Cost for moving left
            print(f"COST for moving LEFT: {cost}")
            clean('A')
        else:
            print("Location A is already clean.")
    else:
        print("Location B is already clean.")
        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving left to Location A.")
            cost += 1  # Cost for moving left
            print(f"COST for moving LEFT: {cost}")
            clean('A')
```

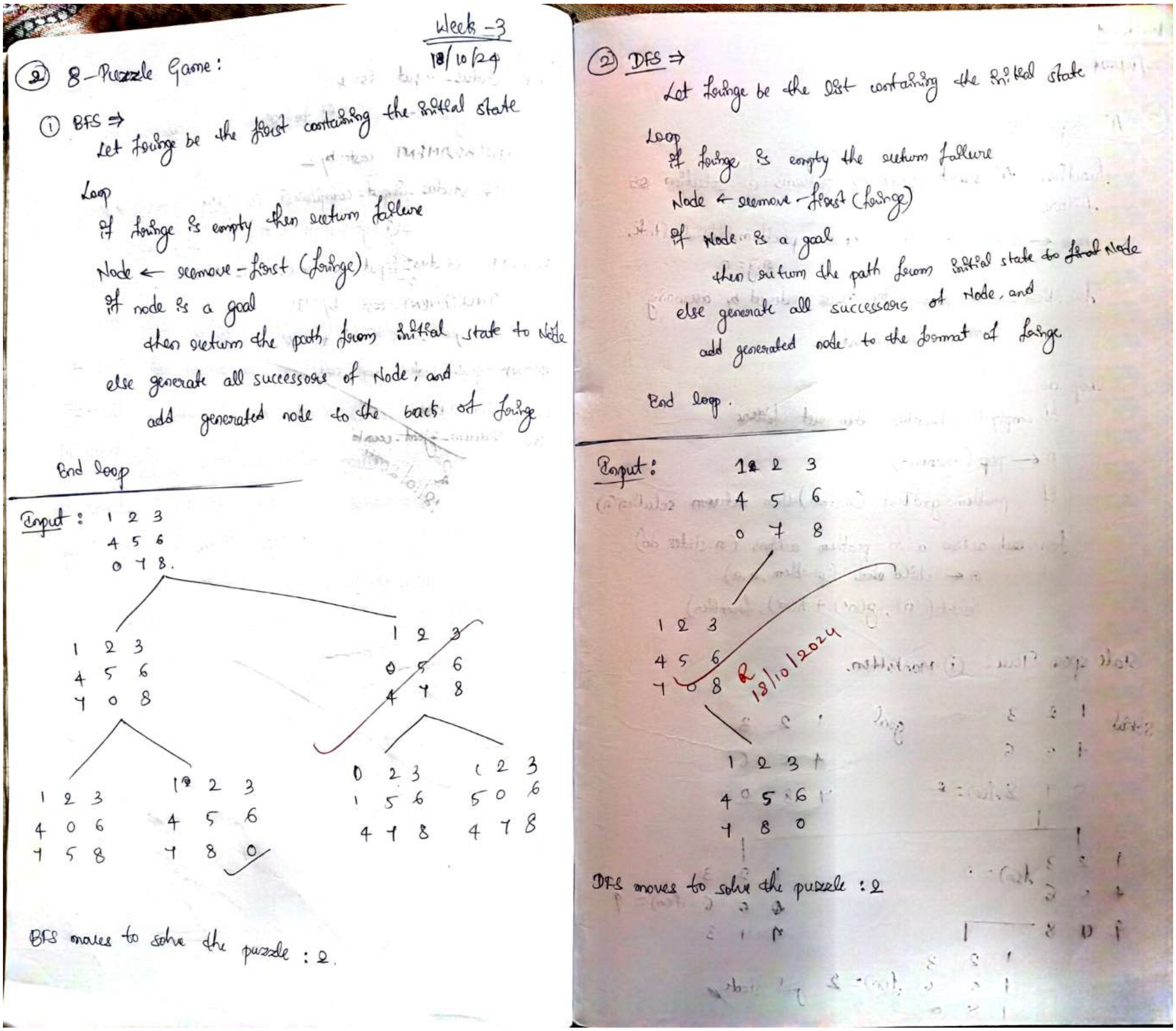
```
        else:
            print("Location A is already clean.")

    # Done cleaning
    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))

# Output
print('OUTPUT:')
print(' Vinayak H (1BM22CS328):')
vacuum_world()
```

OUTPUT:

```
Vinayak H (1BM22CS328):
Enter Location of Vacuum (A or B): A
Enter status of A (0 for Clean, 1 for Dirty): 1
Enter status of B (0 for Clean, 1 for Dirty): 1
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in Location A.
Location A is Dirty.
Location A has been Cleaned. Cost: 1
Location B is Dirty.
Moving right to Location B.
COST for moving RIGHT: 2
Location B has been Cleaned. Cost: 3
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

Program 3 - 8 Puzzle Using BFS and DFS

Algorithm



Week -3
18/10/24

(2) 8-Puzzle Game:

(1) BFS ⇒
Let fringe be the first containing the initial state
Loop
if fringe is empty then return failure
Node ← remove-first (fringe)
if node is a goal
then return the path from initial state to Node
else generate all successors of Node, and
add generated node to the back of fringe
End loop

Input:
1 2 3
4 5 6
0 7 8.

1 2 3
4 5 6
7 0 8

1 2 3
0 5 6
4 7 8

1 2 3
4 0 6
7 5 8

1 2 3
4 5 6
7 8 0

0 2 3
1 5 6
4 7 8

1 2 3
5 0 6
4 7 8

BFS moves to solve the puzzle : 2.

(2) DFS ⇒
Let fringe be the list containing the initial state
Loop
if fringe is empty the return failure
Node ← remove-first (fringe)
if Node is a goal
then return the path from initial state to final Node
else generate all successors of Node, and
add generated node to the front of fringe
End loop.

Input:
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

DFS moves to solve the puzzle : 2

18/10/2024

Code(BFS) :

```
# 8 puzzle problem using BFS technique
# prompt: solve 8-puzzle problem using BFS

from collections import deque

def solve_8puzzle_bfs(initial_state):

    def find_blank(state):
        """Finds the row and column of the blank tile."""
        for row in range(3):
            for col in range(3):
                if state[row][col] == 0:
                    return row, col

    def get_neighbors(state):
        """Generates possible neighbor states by moving the blank tile."""
        row, col = find_blank(state)
        neighbors = []
        if row > 0:
            new_state = [row[:] for row in state]
            new_state[row][col], new_state[row - 1][col] = new_state[row - 1][col],
new_state[row][col]
            neighbors.append(new_state)
        if row < 2:
            new_state = [row[:] for row in state]
            new_state[row][col], new_state[row + 1][col] = new_state[row + 1][col],
new_state[row][col]
            neighbors.append(new_state)
        if col > 0:
            new_state = [row[:] for row in state]
            new_state[row][col], new_state[row][col - 1] = new_state[row][col - 1],
new_state[row][col]
```

```
            neighbors.append(new_state)
        if col < 2:
            new_state = [row[:] for row in state]
            new_state[row][col], new_state[row][col + 1] = new_state[row][col + 1],
new_state[row][col]
            neighbors.append(new_state)
        return neighbors

    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    queue = deque([(initial_state, [])])
    visited = set()

    while queue:
        current_state, path = queue.popleft()
        if current_state == goal_state:
            return path + [current_state]

        visited.add(tuple(map(tuple, current_state)))
        for neighbor in get_neighbors(current_state):
            if tuple(map(tuple, neighbor)) not in visited:
                queue.append((neighbor, path + [current_state]))

    return None  # No solution found
# Example usage:
initial_state = [[1, 2, 3], [4, 0, 6], [7, 5, 8]]
solution = solve_8puzzle_bfs(initial_state)

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

Output Snapshot

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

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

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

Code(DFS):

```
from collections import deque


def solve_8puzzle_dfs(initial_state):



    def find_blank(state):
        """Finds the row and column of the blank tile."""
        for row in range(3):
            for col in range(3):
                if state[row][col] == 0:
                    return row, col

    def get_neighbors(state):
        """Generates possible neighbor states by moving the blank tile."""
        row, col = find_blank(state)
        neighbors = []
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_state = [r[:] for r in state]
                new_state[row][col], new_state[new_row][new_col] =
new_state[new_row][new_col], new_state[row][col]
```

```
            neighbors.append(new_state)
        return neighbors

    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    stack = [(initial_state, [])]
    visited = set()

    while stack:
        current_state, path = stack.pop()
        state_tuple = tuple(map(tuple, current_state))  # Convert to tuple for set
        if state_tuple in visited:
            continue
        visited.add(state_tuple)

        if current_state == goal_state:
            return path + [current_state]

        for neighbor in get_neighbors(current_state):
            stack.append((neighbor, path + [current_state]))

    return None  # No solution found

# Example usage:
initial_state = [[1, 2, 3], [4, 5, 6], [0, 7, 8]]
solution = solve_8puzzle_dfs(initial_state)

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

OUTPUT:

Solution found:

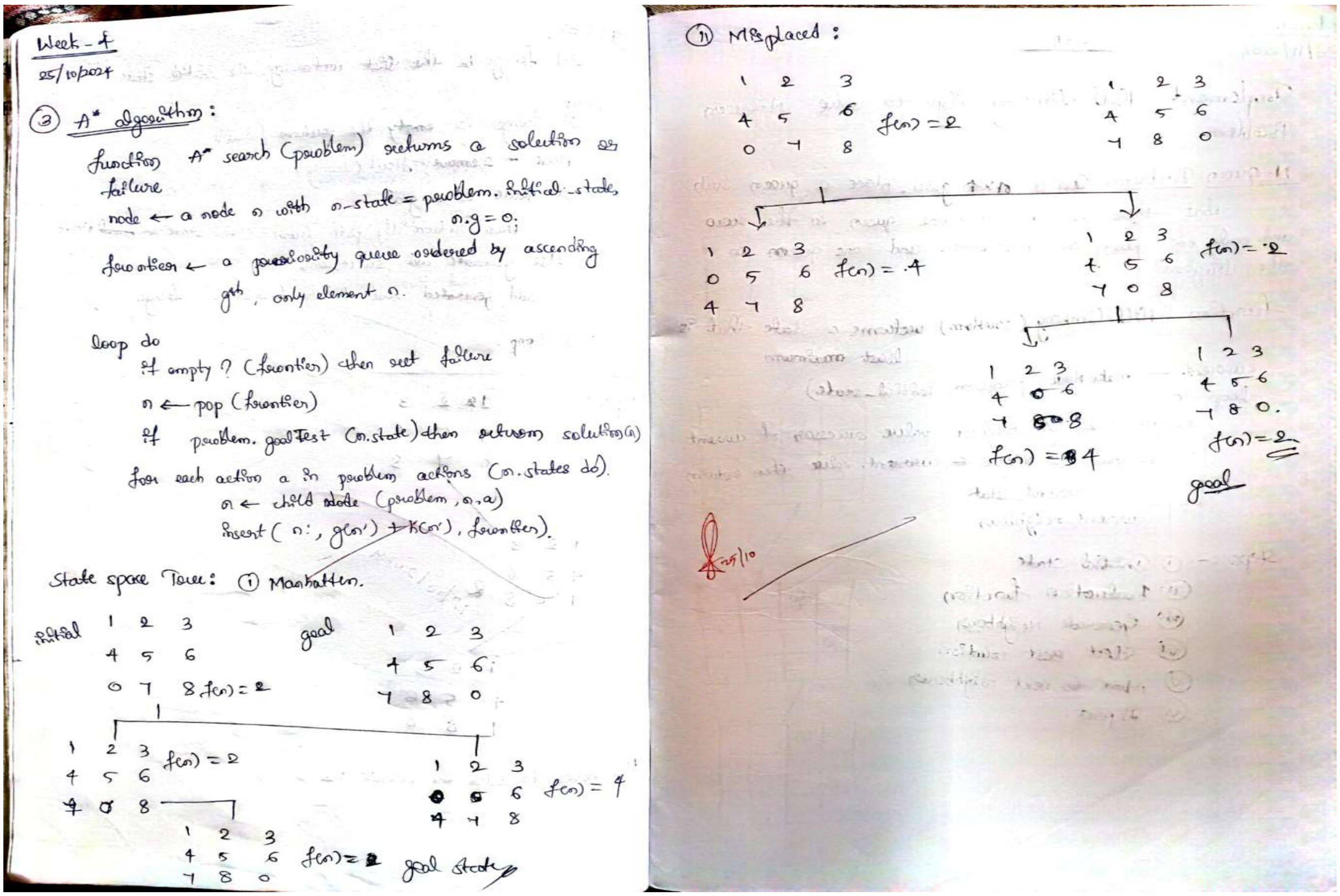
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]

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

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

Program 4 -A* Algorithm

Algorithm



Week-4
25/10/2024

③ A* algorithm:

function A* search (problem) returns a solution or failure

node ← a node n with n-state = problem.initial-state, n.g = 0;

frontier ← a priority queue ordered by ascending g+h, only element n.

loop do

if empty ? (frontier) then return failure

n ← pop (frontier)

if problem.goalTest (n.state) then return solution(n)

for each action a in problem actions (n.states do).

n' ← child node (problem, n, a)

insert (n', g(n') + h(n'), frontier).

State space Tree: ① Manhattan.

initial: 1 2 3 / 4 5 6 / 0 7 8 f(n) = 2 goal: 1 2 3 / 4 5 6 / 7 8 0

1 2 3 / 4 5 6 / 7 0 8 f(n) = 2 1 2 3 / 0 5 6 / 4 7 8 f(n) = 4

1 2 3 / 4 5 6 / 7 8 0 f(n) = 2 goal state

② Misplaced:

1 2 3 / 4 5 6 / 0 7 8 f(n) = 2 1 2 3 / 4 5 6 / 7 8 0

1 2 3 / 0 5 6 / 4 7 8 f(n) = 4 1 2 3 / 4 5 6 / 7 0 8 f(n) = 2

1 2 3 / 4 0 6 / 7 5 8 f(n) = 4 1 2 3 / 4 5 6 / 7 8 0 f(n) = 2 goal

Code:

Manhattan Distance

```
#Manhattan Distance
import heapq


class PuzzleState:
    def __init__(self, board, g=0):
        self.board = board
        self.g = g  # Cost from start to this state
        self.zero_pos = board.index(0)  # Position of the empty space
```

```
    def h(self):
        # Calculate the Manhattan distance
        distance = 0
        for i in range(9):
            if self.board[i] != 0:
                target_x, target_y = divmod(self.board[i] - 1, 3)
                current_x, current_y = divmod(i, 3)
                distance += abs(target_x - current_x) + abs(target_y - current_y)
        return distance

    def f(self):
        return self.g + self.h()

    def get_neighbors(self):
        neighbors = []
        x, y = divmod(self.zero_pos, 3)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_zero_pos = new_x * 3 + new_y
                new_board = self.board[:]
                # Swap zero with the neighboring tile
                new_board[self.zero_pos], new_board[new_zero_pos] =
new_board[new_zero_pos], new_board[self.zero_pos]
                neighbors.append(PuzzleState(new_board, self.g + 1))
        return neighbors

def a_star(initial_state, goal_state):
    open_set = []
    heapq.heappush(open_set, (initial_state.f(), 0, initial_state))  # Push (f, unique_id, state)
    came_from = {}
    g_score = {tuple(initial_state.board): 0}

    while open_set:
```

```
        current_f, _, current = heapq.heappop(open_set)

        if current.board == goal_state:
            return reconstruct_path(came_from, current)

        for neighbor in current.get_neighbors():
            neighbor_tuple = tuple(neighbor.board)
            tentative_g_score = g_score[tuple(current.board)] + 1

            if neighbor_tuple not in g_score or tentative_g_score < g_score[neighbor_tuple]:
                came_from[neighbor_tuple] = current
                g_score[neighbor_tuple] = tentative_g_score
                # Push (f, unique_id, state)
                heapq.heappush(open_set, (neighbor.f(), neighbor.g, neighbor))  # Using g as a
tie-breaker

    return None  # If no solution is found

def reconstruct_path(came_from, current):
    path = []
    while current is not None:
        path.append(current.board)
        current = came_from.get(tuple(current.board), None)
    return path[::-1]

# Example usage
initial_state = PuzzleState([1, 2, 3, 4, 5, 6, 0, 7, 8])
goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]

solution = a_star(initial_state, goal_state)
if solution:
    for step in solution:
        print(step)
else:
```

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

Output:

[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]

CODE:

Number of Misplaced tiles

```
import heapq

def misplaced_tiles(state, goal):
    return sum(1 for i in range(len(state)) if state[i] != 0 and state[i] != goal[i])

def get_neighbors(state):
    neighbors = []
    zero_idx = state.index(0)  # Find the empty tile (represented as 0)
    row, col = divmod(zero_idx, 3)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right

    for dr, dc in directions:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_idx = new_row * 3 + new_col
            new_state = state[:]
            # Swap 0 with the neighbor
            new_state[zero_idx], new_state[new_idx] = new_state[new_idx],
new_state[zero_idx]
            neighbors.append(new_state)
    return neighbors

def a_star(initial_state, goal_state):
    # Priority queue for A* (min-heap)
    open_set = []
```

```
    heapq.heappush(open_set, (0, initial_state))  # (priority, state)

    # Dictionaries to store the cost and parent of each state
    g_cost = {tuple(initial_state): 0}  # Cost from start to current state
    parent = {tuple(initial_state): None}  # To reconstruct the path

    while open_set:
        # Get the state with the lowest f(n) = g(n) + h(n)
        _, current = heapq.heappop(open_set)

        # If we reach the goal, reconstruct the path
        if current == goal_state:
            return reconstruct_path(parent, current)

        for neighbor in get_neighbors(current):
            neighbor_tuple = tuple(neighbor)
            tentative_g_cost = g_cost[tuple(current)] + 1

            # If this path is better, update costs and add to open set
            if neighbor_tuple not in g_cost or tentative_g_cost < g_cost[neighbor_tuple]:
                g_cost[neighbor_tuple] = tentative_g_cost
                f_cost = tentative_g_cost + misplaced_tiles(neighbor, goal_state)
                heapq.heappush(open_set, (f_cost, neighbor))
                parent[neighbor_tuple] = current

    return None  # No solution found

# Helper function to reconstruct the path from start to goal
def reconstruct_path(parent, state):
    path = []
    while state is not None:
        path.append(state)
        state = parent[tuple(state)]
    return path[::-1]  # Reverse the path
# Main function
```

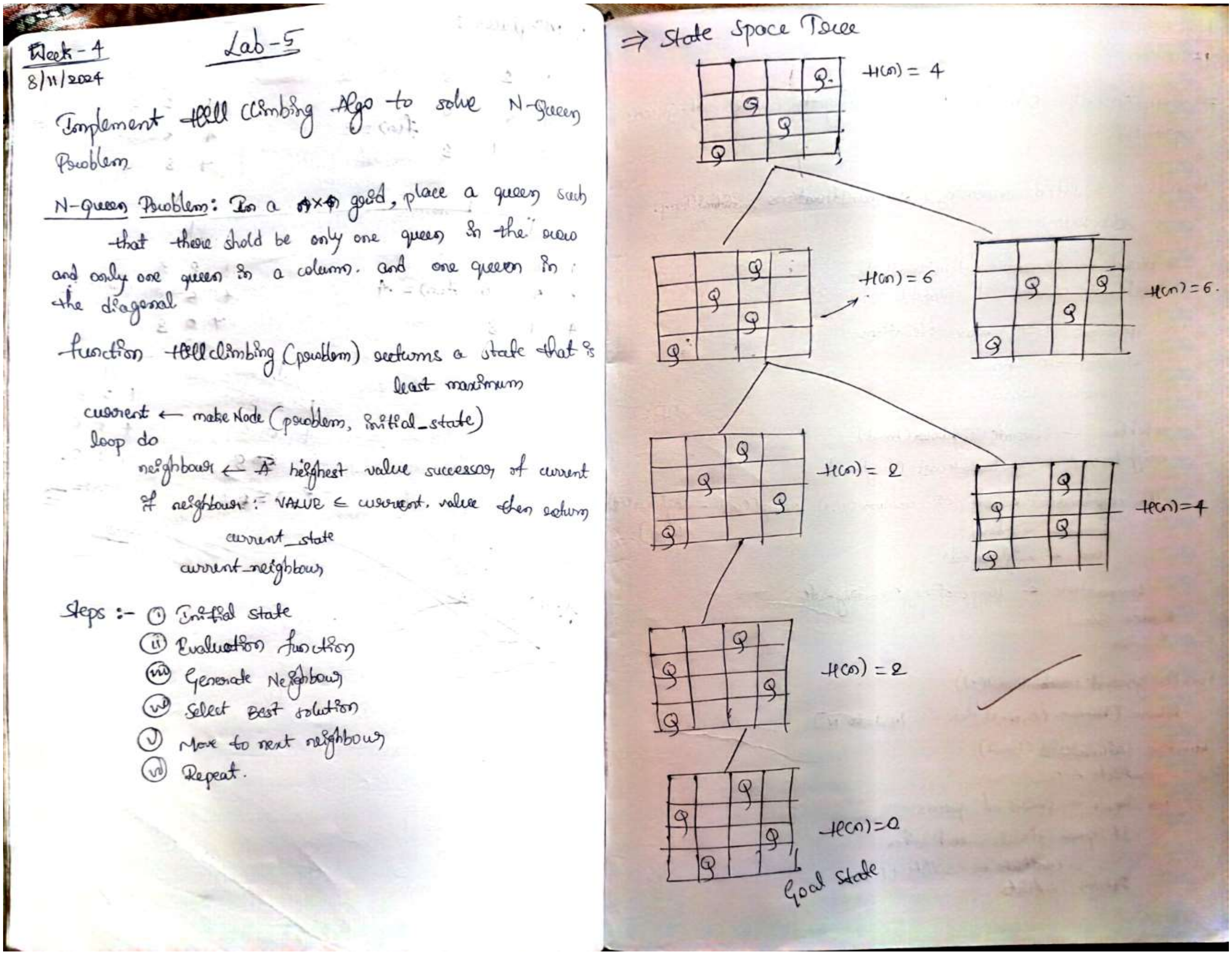
```
if __name__ == "__main__":
    # Define the initial state and goal state
    initial_state = [1, 2, 3, 0, 4, 6, 7, 5, 8]  # 0 represents the empty tile
    goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
    solution = a_star(initial_state, goal_state)

    if solution:
        print("Solution found:")
        for step in solution:
            print_board(step)
            print()
    else:
        print("No solution exists.")
# Helper function to print the 8-puzzle board
def print_board(state):
    for i in range(0, 9, 3):
        print(state[i:i + 3])
```

OUTPUT:

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

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

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

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

PROGRAM 5:HILL CLIMBING(N-QUEENS)

Algorithm


Week-4
8/11/2024
Lab-5

Implement Hill Climbing Algo to solve N-Queen Problem

N-Queen Problem: In a n×n grid, place a queen such that there should be only one queen in the row and only one queen in a column. and one queen in the diagonal

function HillClimbing(problem) returns a state that is least maximum
current ← makeNode(problem, initial_state)
loop do
neighbour ← A highest value successor of current
if neighbour.VALUE ≤ current.value then return current_state
current_neighbour

Steps :- (i) Initial state
(ii) Evaluation function
(iii) Generate Neighbour
(iv) Select Best solution
(v) Move to next neighbour
(vi) Repeat.

CODE:

```
def count_conflicts(state):
    conflicts = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j]:
                conflicts += 1
            if abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
```

```
    return conflicts

def generate_neighbors(state):
    neighbors = []
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            neighbor = state[:]
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]  # Swap positions of queens i
and j
            neighbors.append(neighbor)
    return neighbors

def hill_climbing(n, initial_state):
    state = initial_state
    while True:
        current_conflicts = count_conflicts(state)
        if current_conflicts == 0:
            return state
        neighbors = generate_neighbors(state)
        best_neighbor = None
        best_conflicts = float('inf')
        for neighbor in neighbors:
            conflicts = count_conflicts(neighbor)
            if conflicts < best_conflicts:
                best_conflicts = conflicts
                best_neighbor = neighbor
        if best_conflicts < current_conflicts:
            state = best_neighbor
        else:
            return None
print('Vinayak H (1BM22CS328):')

def get_user_input(n):
    while True:
```

```
        try:
            user_input = input(f"Enter the row positions for the queens (space-separated
integers between 0 and {n-1}): ")
            initial_state = list(map(int, user_input.split()))
            if len(initial_state) != n or any(x < 0 or x >= n for x in initial_state):
                print(f"Invalid input. Please enter exactly {n} integers between 0 and {n-1}.")
                continue
            return initial_state
        except ValueError:
            print(f"Invalid input. Please enter a list of {n} integers.")

n = 4
initial_state = get_user_input(n)

solution = hill_climbing(n, initial_state)

if solution:
    print("Solution found!")
    for row in range(n):
        board = ['Q' if col == solution[row] else '.' for col in range(n)]
        print(' '.join(board))
else:
    print("No solution found (stuck in local minimum).")
```

OUTPUT:

```
Vinayak H (1BM22CS328):
Enter the row positions for the queens (space-separated integers between 0 and 3): 3 1 2 0
Solution found!
. Q . .
. . . Q
Q . . .
. . Q .
```

PROGRAM 6:SIMULATED ANNEALING ALGORITHM

Week - 5
13/11/2024

Lab - 6

⇒ Implement simulated Annealing to solve N-Queens problem.

Function simulated annealing (N, maxIterations, initialTemp, coolingRate)

board ← GenerateRandomBoard(N).
cost ← calculatecost (board).

FOR iterations 1 to max iterations
  IF cost = 0 then
    return board

  neighbour ← generateNeighbour (board).
  neighbourcost ← Calculatecost (neighbour).

  IF neighbourCost < cost OR random (0,1) < exp((cost - neighbourcost) /temp)
    board ← neighbour
    cost ← neighbour cost.

  temperature ← temperature * coolingRate
Return Board
End function

Function generaterandomBoard(N).
  Return [Random (0,N-1) for i in 1 to N].

Function CalculateCost (board).
  conflicts ← 0.
  For i, j in pairs of queens
    If queen threatens each other.
      conflicts ← conflicts + 1
  Return conflicts.

Function GenerateNeighbour (board).
  i, j ← Random (0, N-1)
  swap (Board [i], board [j]).
  Return board.

Output:

No. of queens = 4
Enter positions of queen 1 (0 to 3) : 0
queen 2 (0 to 3) : 1
queen 3 (0 to 3) : 2
queen 4 (0 to 4) : 3.

Iteration 0 : cost = 3, Temp = 1000.00.
[2, 1, 2, 0]
Iteration 1 : cost = 3, Temp = 990.00
[2, 1, 2, 0].
It 2 : cost = 2, Temp = 980.10.
[2, 0, 2, 0].

Solution → [1, 3, 0, 2]

CODE:

```
import random
import math

def calculate_conflicts(board):

    conflicts = 0
    n = len(board)
```

```
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j]:
                conflicts += 1
            elif abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def generate_neighbor(board):

    n = len(board)
    new_board = board[:]

    col = random.randint(0, n - 1)

    current_row = new_board[col]
    possible_rows = set(range(n)) - {current_row}

    valid_rows = set()
    for row in possible_rows:
        valid = True
        for c in range(n):
            if c != col and abs(row - new_board[c]) == abs(col - c):
                valid = False
                break
        if valid:
            valid_rows.add(row)

    if valid_rows:

        new_board[col] = random.choice(list(valid_rows))
    return new_board

def simulated_annealing(n, initial_state, max_iterations=10000, initial_temp=1000,
cooling_rate=0.99):
```

```
    """
    Simulated Annealing to solve the N-Queens problem.
    """
    current_state = initial_state
    current_conflicts = calculate_conflicts(current_state)
    temperature = initial_temp

    for iteration in range(max_iterations):
        if current_conflicts == 0:
            return current_state

        neighbor = generate_neighbor(current_state)
        neighbor_conflicts = calculate_conflicts(neighbor)

        delta = current_conflicts - neighbor_conflicts

        if delta > 0 or random.random() < math.exp(delta / temperature):
            current_state = neighbor
            current_conflicts = neighbor_conflicts

        temperature *= cooling_rate

    return None

def print_solution(board):
    """
    Prints the solution board in a human-readable format.
    """
    n = len(board)
    for row in range(n):
        board_row = ['Q' if col == board[row] else '.' for col in range(n)]
        print(' '.join(board_row))

print('Vinayak H (1BM22CS328):
')
```

```
n = int(input("Enter the number of queens: "))
initial_state_input = input(f"Enter the initial state (a list of {n} integers representing the row positions of queens in each column): ")

initial_state = list(map(int, initial_state_input.strip('[]').split(',')))


if len(initial_state) != n or any(queen < 0 or queen >= n for queen in initial_state):
    print("Invalid initial state! Please make sure it's a list of integers between 0 and n-1.")
else:
    solution = simulated_annealing(n, initial_state)

    if solution:
        print("Solution found:")
        print_solution(solution)
    else:
        print("No solution found.")
```

OUTPUT:

```
Vinayak H (1BM22CS328):
Enter the number of queens: 8
Enter the initial state (a list of 8 integers representing the row positions of queens in each column): 0,1,2,3,4,5,6,7
Solution found:
. . . . . . Q . .
. . . Q . . . .
. . . . . . . Q .
Q . . . . . . .
. . . . . . . . Q
. Q . . . . . .
. . . . Q . . .
. . Q . . . . .
```

//Output:
for 8 queens by taking random inputs

Initial state Board:

```
. . . . . Q . .
. . . . . . Q .
. . . . Q . . .
. . . Q . . . .
. Q . . . . . .
Q . . . . . . .
. . . . . . . Q
. . Q . . . . .
```

Date ___
Page ___

solution found!

```
. . Q . . . . .
. . . . . Q . .
. Q . . . . . .
. . . . . . Q .
. . . . Q . . .
Q . . . . . . .
. . . . . . . Q
. . . Q . . . .
```

PROGRAM 7:UNIFICATION IN FOL

ALGORITHM

Week-6
22/11/24

Lab-7

Unification Algorithm

Algorithm: Unify (Ψ_1, Ψ_2).

Step 1: If Ψ_1 & Ψ_2 ~~are~~ is a variable or constant, then:

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

b) ~~else return {(Ψ_2/Ψ_1)}~~.

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.

Step 2: If the predicate symbol in Ψ_1 and Ψ_2 are not same, then return Failure

Step 3: If Ψ_1 & Ψ_2 have different no. of arguments then return failure

Step 4: set substitutions set (SUBST) to nil.

Step 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 ≠ NIL then do.

a. Apply S to the remainder of both L1 and L2

b. SUBST = APPEND (S, SUBST).

Step 6: Return SUBST.

Output:

Expression_a = "Eats (x, Apple)"
Expression_b = "Eats (Riya, y)"
} I/p

Unification Successful:

{'x' : 'Riya', 'y' : 'Apple'}

22/11/24

CODE:

```
#Implement unification in First Order Logic
def is_variable(x):
    """Checks if x is a variable (assuming variables are single lowercase letters)."""
    return isinstance(x, str) and x.islower() and len(x) == 1

def occurs_check(var, term):
    """Checks if a variable occurs in a term (used to avoid circular unification)."""
    if var == term:
        return True
    if isinstance(term, tuple):  # If term is a function (tuple), check its arguments.
        return any(occurs_check(var, t) for t in term)
    return False

def unify(x, y, substitution=None):
    """Unifies two terms x and y, applying substitutions."""
    if substitution is None:
        substitution = {}

    # Case 1: If both terms are the same, no unification needed
    if x == y:
        return substitution

    # Case 2: If x is a variable, try to unify
    elif is_variable(x):
        if x in substitution:
            return unify(substitution[x], y, substitution)
        elif occurs_check(x, y):
            raise ValueError(f"Unification fails due to occurs check for {x} in {y}")
```

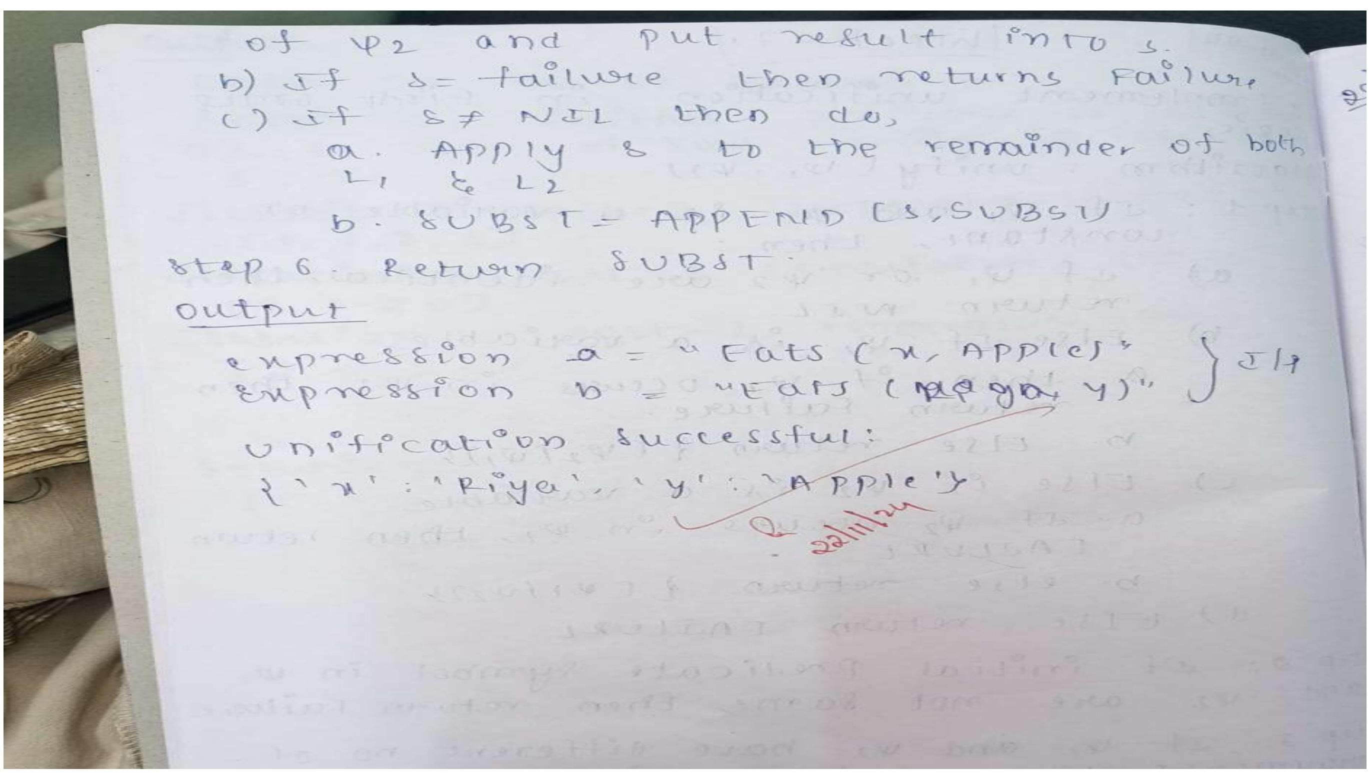
```
        else:
            substitution[x] = y
            return substitution

    # Case 3: If y is a variable, try to unify
    elif is_variable(y):
        return unify(y, x, substitution)

    # Case 4: If both terms are compound (functions), unify their components
    elif isinstance(x, tuple) and isinstance(y, tuple):
        if x[0] != y[0]:
            raise ValueError(f"Unification fails: {x[0]} != {y[0]}")
        # Recursively unify arguments
        for a, b in zip(x[1:], y[1:]):
            substitution = unify(a, b, substitution)
        return substitution

    # Case 5: Unification fails if x and y have no other cases
    else:
        raise ValueError(f"Unification fails: {x} cannot be unified with {y}")

def apply_substitution(term, substitution):
    """Applies the substitution to the term."""
    if isinstance(term, str):
        return substitution.get(term, term)
    elif isinstance(term, tuple):
        return (term[0], *[apply_substitution(t, substitution) for t in term[1:]])
    return term

def parse_term(term_str):
    """Parses a string representation of a term into a Python data structure."""
    term_str = term_str.strip()

    # Case 1: If it's a variable (single lowercase letter)
    if term_str.islower() and len(term_str) == 1:
```

```
        return term_str

    # Case 2: If it's a constant (any non-empty string, for example 'apple')
    if term_str.isalpha():
        return term_str

    # Case 3: If it's a function, e.g., 'f(x, y)'
    if term_str.startswith('f(') and term_str.endswith(')'):
        func_str = term_str[2:-1]  # Remove 'f(' and ')'
        parts = func_str.split(',')
        return ('f', *[parse_term(p.strip()) for p in parts])  # Function name, arguments

    # If none of these, raise an error
    raise ValueError(f"Invalid term format: {term_str}")
print(Vinayak H (1BM22CS328):')

def main():
    print("Enter two terms to unify (e.g., f(x, y), f(a, b)):")
    term1_str = input("Enter first term: ")
    term2_str = input("Enter second term: ")

    try:
        term1 = parse_term(term1_str)
        term2 = parse_term(term2_str)

        print(f"Unifying terms: {term1} and {term2}")
        substitution = unify(term1, term2)

        # Apply substitution to both terms to get the unified expression
        unified_term1 = apply_substitution(term1, substitution)
        unified_term2 = apply_substitution(term2, substitution)

        print("Unification successful!")
        print("Substitution:", substitution)
        print("Unified expression:")
```

```
        print(f"Term 1 after substitution: {unified_term1}")
        print(f"Term 2 after substitution: {unified_term2}")

    except ValueError as e:
        print("Unification failed:", e)

# Run the program
if __name__ == "__main__":
    main()
```

OUTPUT:

Vinayak H(1BM22CS328):
Enter two terms to unify (e.g., f(x, y), f(a, b)):
Enter first term: f(x,car)
Enter second term: f(bike,y)
Unifying terms: ('f', 'x', 'car') and ('f', 'bike', 'y')
Unification successful!
Substitution: {'x': 'bike', 'y': 'car'}
Unified expression:
Term 1 after substitution: ('f', 'bike', 'car')
Term 2 after substitution: ('f', 'bike', 'car')


of ψ_2 and put result into s.
b) If s = failure then returns Failure
c) If S ≠ NIL then do,
a. Apply S to the remainder of both L_1 & L_2
b. SUBST = APPEND (S, SUBST)
Step 6. Return SUBST.
Output
expression a = "Eats (x, Apple)"
expression b = "Eats (Riya, y)"
unification successful:
{'x': 'Riya', 'y': 'Apple'}
22/11/24

PROGRAM 8:FORWARD CHAINING ALGORITHM

Week - 7
29/11/2024

Lab - 8

Forward Reasoning Algorithm :

function FOL-FC-Ask (KB, α) returns a substitution or false
inputs : KB, the knowledgebase, 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
($p_1 \wedge \ldots \wedge p_n \Rightarrow q$) ← standardize -variables (rule)
for each θ such that SUBST (θ, $p_1 \wedge \ldots \wedge p_n$) = SUBST ($\theta$, $p_1' \ldots \wedge p_n'$)
for some $p_1' \ldots p_n'$ in KB.
q' ← SUBST (θ, q).
if q' does not unify with some sentence already in KB or new then add q' to new.
ϕ ← Unify (q', α)
if ϕ is not fail then return ϕ.
add new to KB.
~~false~~
return false

Output :

Rule applied : {'conditions' : {'cough', 'fever'}
'conclusion' : 'flu'}

New fact inferred : flu

Final facts = {'cough', 'flu', 'fever'}

Inferred facts = {'flu'}.

a) Occupation (Emily, Surgeon) v Occupation (Emily, Lawyer)

b) Occupation (Joe, Actor) $\wedge \exists$ o (o $\neq$ Actor $\wedge$ occupation (Joe, o)).

c) $\forall$ P (Occupation (P, surgeon) → ∃ (Occupation (P, Doctor))

d) ¬ ∃ P.(occupation (P, Layer) ∧ customer (Joe, P)).

e) ∃ P (Occupation (P, Lawyer) ∧ Boss (P, Emily)).

f) ∃ P (Occupation (P, Lawyer) ∧ ∀ C (customer (C, P) → occupation (C, Doctor)).

g) ∀ P (Occupation (P, surgeon) → ∃ C (occupation (C, Lawyer) ∧ customer (P, C))).

29/11/24

CODE:

```
knowledge_base = {
    "facts": {
        "American(Robert)": True,
        "Enemy(A, America)": True,
        "Owns(A, T1)": True,
        "Missile(T1)": True,
    },
    "rules": [
        {"if": ["Missile(x)"], "then": ["Weapon(x)"]},
        {"if": ["Enemy(x, America)"], "then": ["Hostile(x)"]},
        {"if": ["Missile(x)", "Owns(A, x)"], "then": ["Sells(Robert, x, A)"]},
        {
            "if": ["American(p)", "Weapon(q)", "Sells(p, q, r)", "Hostile(r)"],
            "then": ["Criminal(p)"],
        },
    ],
}

def forward_chaining(kb):
    facts = kb["facts"].copy()
    rules = kb["rules"]

    inferred = set()

    while True:
```

```
new_inferences = set()

for rule in rules:
    if_conditions = rule["if"]
    then_conditions = rule["then"]

    substitutions = {}
    all_conditions_met = True

    for condition in if_conditions:
        predicate, args = condition.split("(")
        args = args[:-1].split(",")

        matched = False
        for fact in facts:
            fact_predicate, fact_args = fact.split("(")
            fact_args = fact_args[:-1].split(",")

            if predicate == fact_predicate and len(args) == len(fact_args):
                temp_subs = {}
                for var, val in zip(args, fact_args):
                    if var.islower():
                        if var in temp_subs and temp_subs[var] != val:
                            break
                        temp_subs[var] = val
                    elif var != val:
                        break
                else:
                    matched = True
                    substitutions.update(temp_subs)
                    break

        if not matched:
            all_conditions_met = False
            break
```

```
            if all_conditions_met:
                for condition in then_conditions:
                    predicate, args = condition.split("(")
                    args = args[:-1].split(",")
                    new_fact = predicate + "(" + ",".join(substitutions.get(arg, arg) for arg in args)
+ ")"
                    new_inferences.add(new_fact)

        if new_inferences - inferred:
            inferred.update(new_inferences)
            facts.update({fact: True for fact in new_inferences})
        else:
            break

    return inferred

result = forward_chaining(knowledge_base)
print(' (1BM22CS328):')

if "Criminal(Robert)" in result:
    print("Proved: Robert is a criminal.")
else:
    print("Could not prove that Robert is a criminal.")
```

OUTPUT:

(1BM22CS328):

Proved: Robert is a criminal.

PROGRAM 9:ALPHA BETA PRUNING

ALGORITHM

Week-8

Lab - 10

Alpha - Beta Pruning :

Algorithm :

1) Alpha - Beta proposes to compute ~~find~~ the optimal solution without ~~finding~~ looking at the every node in the given tree

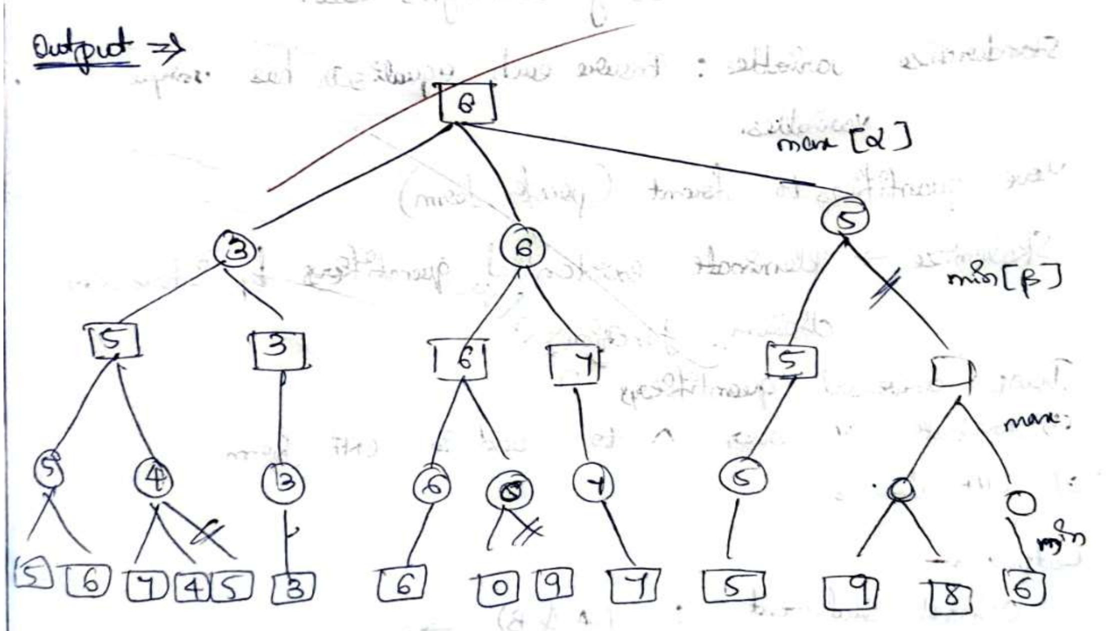
2) Max contains (α) alpha , min contains (β). bound during calculation.

3) In both min and max node we return when α ≥ β which computes. with its parent node only.

4) Both min-max and α-β cut off gives same path.

5) α-β - gives optimal path as it takes less time to get the value of the root node.

Output ⇒



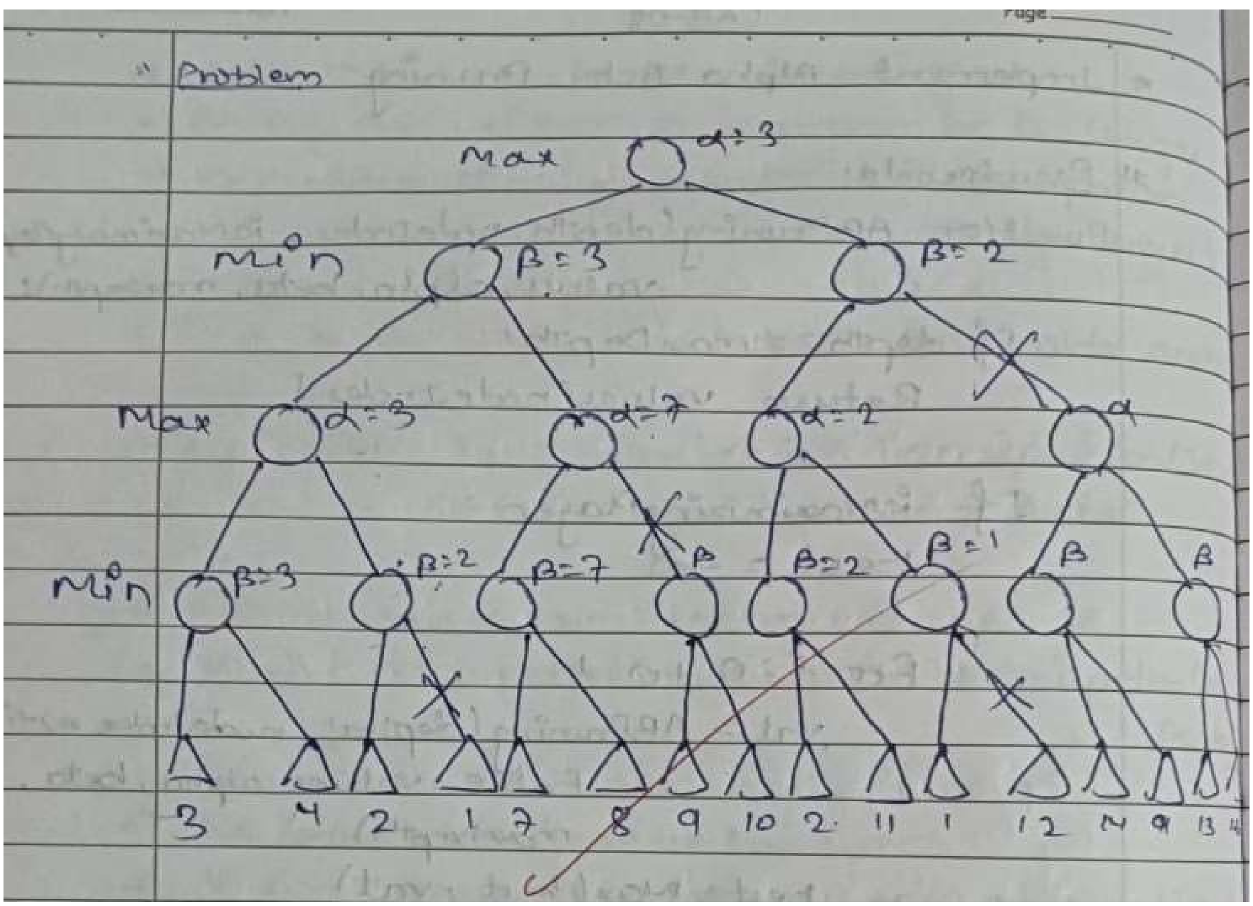
CODE:

```
import math

def alpha_beta_pruning(depth, node_index, is_maximizing_player, values, alpha, beta,
max_depth):
    # Base case: when the maximum depth is reached
    if depth == max_depth:
        return values[node_index]

    if is_maximizing_player:
        best = -math.inf

        # Recur for left and right children
        for i in range(2):
            val = alpha_beta_pruning(depth + 1, node_index * 2 + i, False, values, alpha, beta,
max_depth)
            best = max(best, val)
            alpha = max(alpha, best)

            # Prune the remaining nodes
            if beta <= alpha:
                break
        return best

    else:
        best = math.inf

        # Recur for left and right children
        for i in range(2):
            val = alpha_beta_pruning(depth + 1, node_index * 2 + i, True, values, alpha, beta,
max_depth)
            best = min(best, val)
            beta = min(beta, best)
```

```
            # Prune the remaining nodes
            if beta <= alpha:
                break
        return best


# Example usage
if __name__ == "__main__":
    # Example tree represented as a list of leaf node values
    values = [3, 5, 6, 9, 1, 2, 0, -1]
    max_depth = 3  # Height of the tree

    result = alpha_beta_pruning(0, 0, True, values, -math.inf, math.inf, max_depth)
    print("The optimal value is:", result)
```

OUTPUT:

The optimal value is: 5



PROGRAM 10:PROVE A QUERY USING RESOLUTION ALGORITHM

Week -8
20/12/24

Lab - 9

Convert FOL to Resolution:

Algorithm -

1) Convert all sentences to ~~FOL~~ CNF

2) Negate conclusion S and convert result to CNF

3) Add negated conclusion S to the premise clauses.

4) Repeat until contradiction or no progress is made
   a) Select two clauses (call them parent clauses).
   b) Resolve them together, performing all required unification
   c) If resolvent is the empty clause a contradiction has been found (i.e S follows from the premises)
   d) If not - add resolvent to the premises.

If we succeed in step 4 we have proved the conclusion.

Eg → Given KB or premises.

a: John likes all kinds of food

b: Apple and vegetables are food

c: Anything anyone eats and not dies is a food

d: Anil eats peanuts & is still alive

e: Harry eats everything that anil eats.

f: Prove by resolution that John likes food.

⇒ The conclusion can be proved by Resolution
Hence proved

CODE:

```
# Example propositional logic statements in CNF
kb = [{"¬B", "¬C", "A"},  # ¬B ∨ ¬C ∨ A
```

```
    {"B"},  # B
    {"¬D", "¬E", "C"},  # ¬D ∨ ¬E ∨ C
    {"E", "F"},  # E ∨ F
    {"D"}, #D
     {"¬F"}, #¬F
]
# Negate the query: If the query is "A", we negate it to "¬A"
def negate_query(query):
    if "¬" in query:
        return query.replace("¬", "")  # If it's negated, remove the negation
    else:
        return f"¬{query}"  # Otherwise, add negation in front

# Function to perform resolution on two clauses
def resolve(clause1, clause2):
    resolved_clauses = []

    # Try to find complementary literals
    for literal1 in clause1:
        for literal2 in clause2:
            # If literals are complementary (e.g., "A" and "¬A"), resolve them
            if literal1 == f"¬{literal2}" or f"¬{literal1}" == literal2:
                new_clause = (clause1 | clause2) - {literal1, literal2}
                resolved_clauses.append(new_clause)

    return resolved_clauses

# Perform resolution-based proof
def resolution(kb, query):
    # Step 1: Negate the query and add it to the knowledge base
    negated_query = negate_query(query)
    kb.append({negated_query})

    # Step 2: Initialize the set of clauses
    new_clauses = set(frozenset(clause) for clause in kb)
```

```
    while True:
        resolved_this_round = set()
        clauses_list = list(new_clauses)

        # Try to resolve every pair of clauses
        for i in range(len(clauses_list)):
            for j in range(i + 1, len(clauses_list)):
                clause1 = clauses_list[i]
                clause2 = clauses_list[j]

                # Apply resolution to the two clauses
                resolved = resolve(clause1, clause2)
                if frozenset() in resolved:
                    return True  # Found an empty clause (contradiction), query is provable
                resolved_this_round.update(resolved)

        # If no new clauses were added, stop
        if resolved_this_round.issubset(new_clauses):
            return False  # No new clauses, query is not provable
        # Add new resolved clauses to the set
        new_clauses.update(resolved_this_round)

# Query to prove: "A"
query = (input("Enter the query:"))
result = resolution(kb, query)
print("OUTPUT:(1BM22CS328)")
print("Using Resolution to prove a query")
print(f"Is the query '{query}' provable? {'Yes' if result else 'No'}")
```

OUTPUT:

```
Enter the query:A
OUTPUT:(1BM22CS328)
Using Resolution to prove a query
Is the query 'A' provable? Yes
```

PROGRAM 11:FOL TO CNF

ALGORITHM

~~Propositional Logic :~~

~~Objective → Create a KB using propositional logic and show that the given query entails the knowledge or not~~

~~Algorithm:~~

~~function TT-Entails (KB, α)~~

Lab - 11

Convert FOL to CNF.

Input first-order-logic-statement.

Eliminate Implications : Replace (A → B) with (¬A ∨ B).

Move ¬ inwards using Demorgan's Law.

Standardize variables : Ensure each quantifier has unique variables.

Move quantifiers to front (prenex form)

Skolemize → Eliminate existential quantifiers by introducing skolem functions

Drop universal quantifiers

Distribute ∨ over ∧ to add in CNF form

o/p CNF clauses

Output :-

Original Statement : (A & B) → C

CNF form : ~A/~B/C.

20/12/24

CODE:

```
from sympy import symbols, Not, Or, And, Implies, Equivalent
from sympy.logic.boolalg import to_cnf

def fol_to_cnf(fol_expr):
```

```
    """
    Converts a First-Order Logic (FOL) statement to Conjunctive Normal Form (CNF).
    Arguments:
      fol_expr: A sympy logical expression representing the FOL statement.
    Returns:
      The CNF equivalent of the input expression.
    """
    # Step 1: Eliminate equivalences (A ↔ B) using (A → B) ∧ (B → A)
    fol_expr = fol_expr.replace(Equivalent, lambda a, b: And(Implies(a, b), Implies(b, a)))

    # Step 2: Eliminate implications (A → B) using (¬A ∨ B) fol_expr
    = fol_expr.replace(Implies, lambda a, b: Or(Not(a), b))

    # Step 3: Convert to CNF
    cnf_form = to_cnf(fol_expr, simplify=True)

    return cnf_form

def main():
    # Define propositional symbols instead of first-order predicates
    P = symbols("P")
    Q = symbols("Q")
    R = symbols("R")

    # Example 1: P → Q
    fol_expr1 = Implies(P, Q)
    print("Example 1: P → Q")
    print("Original FOL Expression:")
    print(fol_expr1)

    # Convert to CNF
    cnf1 = fol_to_cnf(fol_expr1)
    print("\nCNF Form:")
    print(cnf1)
```

```
    # Example 2: (P ∨ ¬Q) → (Q ∨ R) fol_expr2
    = Implies(Or(P, Not(Q)), Or(Q, R))
    print("\nExample 2: (P ∨ ¬Q) → (Q ∨ R)")
    print("Original FOL Expression:")
    print(fol_expr2)

    # Convert to CNF
    cnf2 = fol_to_cnf(fol_expr2)
    print("\nCNF Form:")
    print(cnf2)

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

OUTPUT:

```
Example 1: P → Q
Original FOL Expression:
Implies(P, Q)

CNF Form:
Q | ~P

Example 2: (P ∨ ¬Q) → (Q ∨ R)
Original FOL Expression:
Implies(P | ~Q, Q | R)

CNF Form:
Q | R
```