Artificial Intelligence Lab Report

Submited By
Vatsal Amruthling Mural
(1BM22CS323)

Course: Artificial Inteligence
Course Code:23CS5PCAIN
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 Number	Program Title	Page Number
1	Tic-Tac-Toe	3-6
2	Vacuum Cleaner	7-9
3	8-Puzzle BFS & DFS	10-15
4	A* Algorithm (8 Puzzle)	16-20
5	HILL CLIMBING(N-QUEENS)	21-22
6	SIMULATED ANNEALING	23-25
7	UNIFICATION IN FOL	26-28
8	FORWARD REASONING	29-31
9	ALPHA-BETA PRUNING	32-33
10	FOL To CNF	34-35
11	Proving Query Using Resolution	36-37
12	Proving Query Entails With KB or Not	38-39

Program 1 - Tic Tac toe

Algorithm:

4/10/23

AI

Define Tic Tac Toe

Pseudocode

Function Minimax (Nodes, depth, min, max)

if node is a terminal state:

return evaluate (node) -> check if the game has end.

if isMaximizing Player:

best Value = -infinity

for each child in node:

Value = minimax(child, depth +1, false)

best Value = max(bestvalue, value

return bestvalue

else:

bestValue = +infinity

for each child in node:

Value = minimax(child, depth+1, true)

bestValue = min (bestValue, value)

return bestValue

4/10/23

node : current game state

depth : current depth

isMaxplayer : boolean indicating where it is maximizing Player or not.

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):
    return board[pos] == ' '

def checkWin():
    win_conditions = [(1, 2, 3), (4, 5, 6), (7, 8, 9),  # Horizontal
                      (1, 4, 7), (2, 5, 8), (3, 6, 9),  # Vertical
                      (1, 5, 9), (3, 5, 7)]             # Diagonal
    for a, b, c in win_conditions:
        if board[a] == board[b] == board[c] and board[a] != ' ':
            return True
    return False

def checkMoveForWin(move):
    win_conditions = [(1, 2, 3), (4, 5, 6), (7, 8, 9),  # Horizontal
                      (1, 4, 7), (2, 5, 8), (3, 6, 9),  # Vertical
                      (1, 5, 9), (3, 5, 7)]             # Diagonal
    for a, b, c in win_conditions:
        if board[a] == board[b] == board[c] and board[a] == move:
            return True
    return False

def checkDraw():
    return all(board[key] != ' ' for key in board.keys())

def insertLetter(letter, position):
    if spaceFree(position):
        board[position] = letter
        printBoard(board)

        if checkWin():
            if letter == 'X':
                print('Bot wins!')
            else:
                print('You win!')
            return True  # Game over, no more moves
        elif checkDraw():
            print('Draw!')
            return True  # Game over, no more moves
    else:
        print('Position taken, please pick a different position.')
        position = int(input('Enter new position: '))
        return insertLetter(letter, position)

    return False  # Continue the game

player = 'O'
bot = 'X'
```

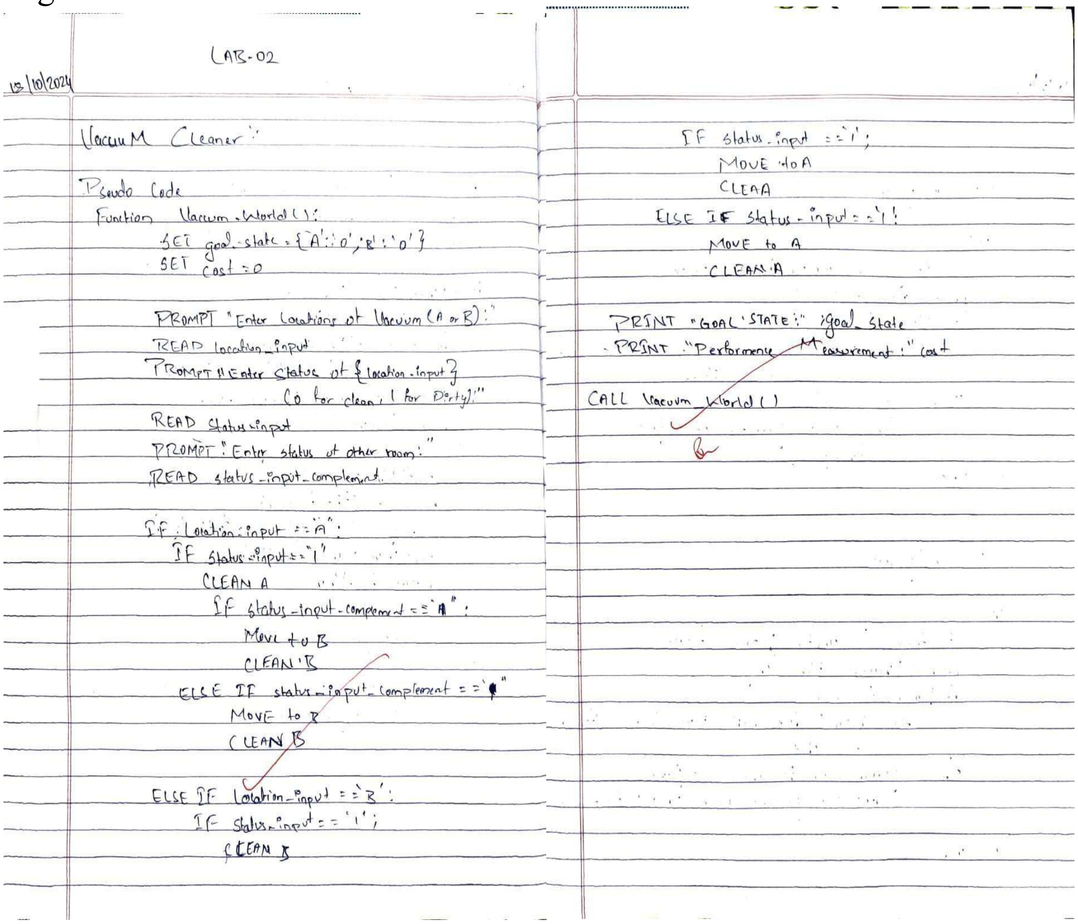
```
def playerMove():
    position = int(input('Enter position for O: '))
    return insertLetter(player, position)

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

    return insertLetter(bot, bestMove)

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] = ' '
                bestScore = max(score, bestScore)
        return bestScore
    else:
        bestScore = 1000
        for key in board.keys():
            if board[key] == ' ':
                board[key] = player
                score = minimax(board, True)
                board[key] = ' '
                bestScore = min(score, bestScore)
        return bestScore

# Main game loop
game_over = False
while not game_over:
    game_over = compMove()  # Bot's turn
    if not game_over:
        game_over = playerMove()  # Player's turn
print('Vatsal - 1BM22CS323')
```

Output:

```
Output
X| |
-+-+-
 | |
-+-+-
 | |


Enter position for O: 5
X| |
-+-+-
 |O|
-+-+-
 | |


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


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


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


Enter position for O: 4
X|X|O
-+-+-
O|O|
-+-+-
X| |


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


Enter position for O: 7
Position taken, please pick a different position.
Enter new position: 8
X|X|O
-+-+-
O|O|X
-+-+-
X|O|


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


Draw!
Vatsal - 1BM22CS323
```

Program-2 Vacuum Cleaner

Algorithm:



LAB-02

13/10/2024

Vacuum Cleaner:

Pseudo Code

```
Function Vacuum_World():
    SET goal_state = {'A': '0', 'B': '0'}
    SET cost = 0

    PROMPT "Enter Locations of Vacuum (A or B):"
    READ location_input
    PROMPT "Enter status of {location_input}
                (0 for clean, 1 for Dirty):"
    READ status_input
    PROMPT "Enter status of other room:"
    READ status_input_complement.

    IF Location_input == 'A':
        IF status_input == '1'
            CLEAN A
            IF status_input_complement == 'A':
                Move to B
                CLEAN B
        ELSE IF status_input_complement == '1'
            MOVE to B
            CLEAN B

    ELSE IF Location_input == 'B':
        IF status_input == '1';
            CLEAN B
        IF status_input == '1';
            MOVE to A
            CLEAN A
        ELSE IF status_input == '1':
            MOVE to A
            CLEAN A

    PRINT "GOAL STATE:" goal_state
    PRINT "Performance Measurement:" cost

CALL Vacuum_World()
```

Code:

```
def vacuum_world():

    goal_state = {'A': '0', 'B': '0'}
    cost = 0


    location_input = input("Enter Location of Vacuum (A or B): ").strip().upper()
    status_input = input(f"Enter status of A (0 for Clean, 1 for Dirty): ").strip()
    status_input_complement = input("Enter status of B (0 for Clean, 1 for Dirty): ").strip()

    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 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("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
            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("Location B is already clean.")

elif location_input == 'B':
    print("Vacuum is placed in Location B")
    if status_input == '1':
        print("Location B is Dirty.")
        goal_state['B'] = '0'  # Clean B
        cost += 1  # Cost for sucking
        print("Cost for cleaning B: " + str(cost))
        print("Location B has been Cleaned.")

        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving LEFT to Location A.")
            cost += 1  # Cost for moving left
            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("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
            print("Cost for moving LEFT: " + str(cost))

            goal_state['A'] = '0'
            cost += 1
            print("Cost for suck: " + str(cost))
```

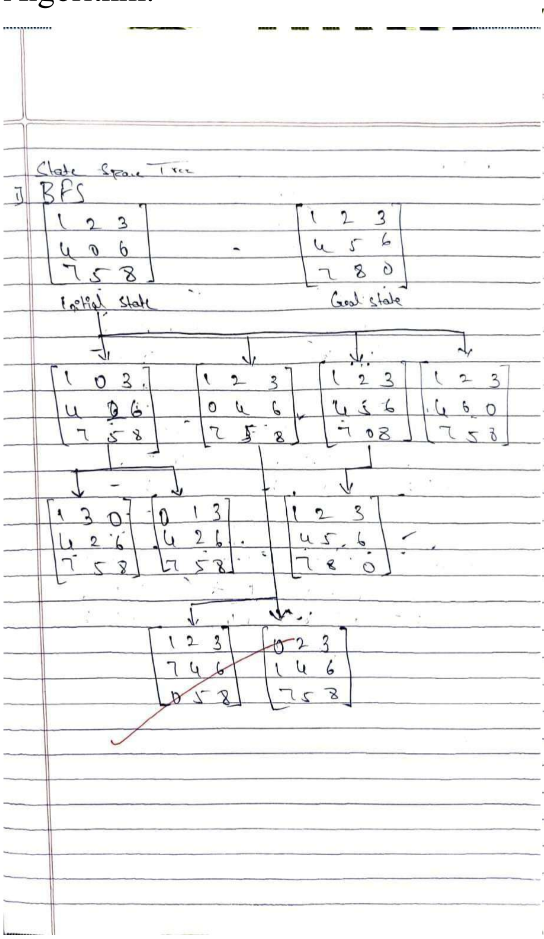
```
            print("Location A has been Cleaned.")
        else:
            print("Location A is already clean.")


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

# Output
vacuum_world()
print("Vatsal-1BM22CS323")
```

Output:

```
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): 0
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 already clean.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 1
Vatsal-1BM22CS323
```

Program-3 8-puzzle(BFS):

Algorithm:



18/10/24

8 Puzzle

BFS Algorithm

LOOP
if fringe is empty return failure
Node ← remove - first (fringe)
if Node is a goal
then return the path from initial state to final state
Else generate all successors of Node,
and add generated node to the back of fringe
End LOOP

DFS Algorithm

LOOP
if fringe is empty return failure
Node ← remove - first (fringe)
if Node is a goal
the return the path from initial state to final state
Else generate all successor of Node
And add generated node to the front of fringe
End LOOP

Code:

```
from collections import deque

class PuzzleState:
    def __init__(self, board, zero_position, path=[]):
        self.board = board
        self.zero_position = zero_position
        self.path = path

    def is_goal(self):
        return self.board == [1, 2, 3, 4, 5, 6, 7, 8, 0]

    def get_possible_moves(self):
        moves = []
        row, col = self.zero_position
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left, Up

        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
```

```
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_board = self.board[:]
                # Swap zero with the adjacent tile
                new_board[row * 3 + col], new_board[new_row * 3 + new_col] = new_board[new_row * 3 +
new_col], new_board[row * 3 + col]
                moves.append(PuzzleState(new_board, (new_row, new_col), self.path + [new_board]))

        return moves

def bfs(initial_state):
    queue = deque([initial_state])
    visited = set()

    while queue:
        current_state = queue.popleft()
        if current_state.is_goal():
            return current_state.path

        visited.add(tuple(current_state.board))

        for next_state in current_state.get_possible_moves():
            if tuple(next_state.board) not in visited:
                queue.append(next_state)

    return None

def print_board(board):
    for i in range(3):
        print(board[i * 3:i * 3 + 3])

def main():
    print("Enter the initial state of the 8-puzzle (use 0 for the blank tile, e.g., '1 2 3 4 5 6 7 8 0'): ")
    user_input = input()
    initial_board = list(map(int, user_input.split()))

    if len(initial_board) != 9 or set(initial_board) != set(range(9)):
        print("Invalid input! Please enter 9 numbers from 0 to 8.")
        return

    zero_position = initial_board.index(0)
    initial_state = PuzzleState(initial_board, (zero_position // 3, zero_position % 3))

    solution_path = bfs(initial_state)

    if solution_path is None:
        print("No solution found.")
    else:
        print("Solution found in", len(solution_path), "steps.")
        for step in solution_path:
            print_board(step)
            print()

if __name__ == "__main__":
    main()
print("Vatsal-1BM22CS323")
```

Output:

```
Enter the initial state of the 8-puzzle (use 0 for the blank tile, e.g., '1
    2 3 4 5 6 7 8 0'):
1 2 3 4 5 6 0 7 8
Solution found in 2 steps.
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

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

Vatsal-1BM22CS323
```

Program 3- 8puzzle(DFS):

Algorithm:

18/10/24

8 Puzzle

BFS Algorithm

LOOP
- if fringe is empty return failure
- Node ← remove-first (fringe)
- if Node is a goal
  - then return the path from initial state to final state
- else, generate all successors of Node, and add generated node to the back of fringe

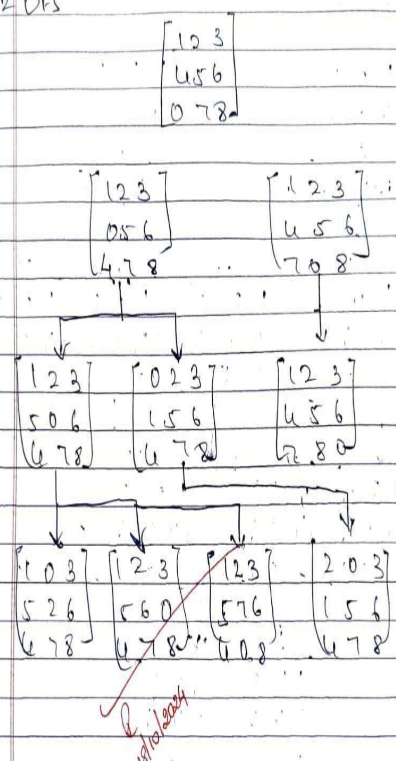
End LOOP

DFS Algorithm

LOOP
- if fringe is empty return failure
- Node ← remove-first (fringe)
- if Node is a goal
  - then return the path from initial state to final state
- else generate all successor of Node
  - And add generated node to the front of fringe

End LOOP

2 DFS



Code:

```
from collections import deque

class PuzzleState:
    def __init__(self, board, zero_position, path=[]):
        self.board = board
        self.zero_position = zero_position
        self.path = path
```

```
    def is_goal(self):
        return self.board == [1, 2, 3, 4, 5, 6, 7, 8, 0]

    def get_possible_moves(self):
        moves = []
        row, col = self.zero_position
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left, Up

        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_board = self.board[:]
                # Swap zero with the adjacent tile
                new_board[row * 3 + col], new_board[new_row * 3 + new_col] = new_board[new_row * 3 +
new_col], new_board[row * 3 + col]
                moves.append(PuzzleState(new_board, (new_row, new_col), self.path + [new_board]))

        return moves

def dfs(initial_state):
    stack = [initial_state]
    visited = set()

    while stack:
        current_state = stack.pop()
        if current_state.is_goal():
            return current_state.path

        visited.add(tuple(current_state.board))

        for next_state in reversed(current_state.get_possible_moves()):  # Reverse to simulate DFS
            if tuple(next_state.board) not in visited:
                stack.append(next_state)

    return None

def print_board(board):
    for i in range(3):
        print(board[i * 3:i * 3 + 3])

def main():
    print("Enter the initial state of the 8-puzzle (use 0 for the blank tile, e.g., '1 2 3 4 5 6 7 8 0'): ")
    user_input = input()
    initial_board = list(map(int, user_input.split()))

    if len(initial_board) != 9 or set(initial_board) != set(range(9)):
        print("Invalid input! Please enter 9 numbers from 0 to 8.")
        return

    zero_position = initial_board.index(0)
    initial_state = PuzzleState(initial_board, (zero_position // 3, zero_position % 3))

    solution_path = dfs(initial_state)

    if solution_path is None:
        print("No solution found.")
    else:
        print("Solution found in", len(solution_path), "steps.")
        for step in solution_path:
```

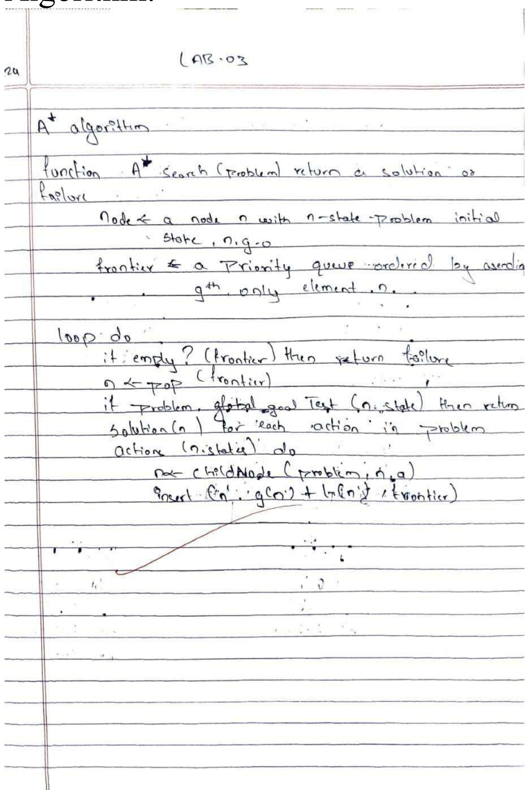
```
            print_board(step)
            print()

if __name__== "__main__":
    main()
print("Vatsal-1BM22CS323")
```

Output:

```
Enter the initial state of the 8-puzzle (use 0 for the blank tile, e.g., '1
    2 3 4 5 6 7 8 0'):
1 2 3 4 5 6 0 7 8
Solution found in 2 steps.
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

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

Vatsal-1BM22CS323
```

Program 4-A* Search:

Algorithm:



LAB-03

A* algorithm

function A* Search (problem) return 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 with only element n.

loop do

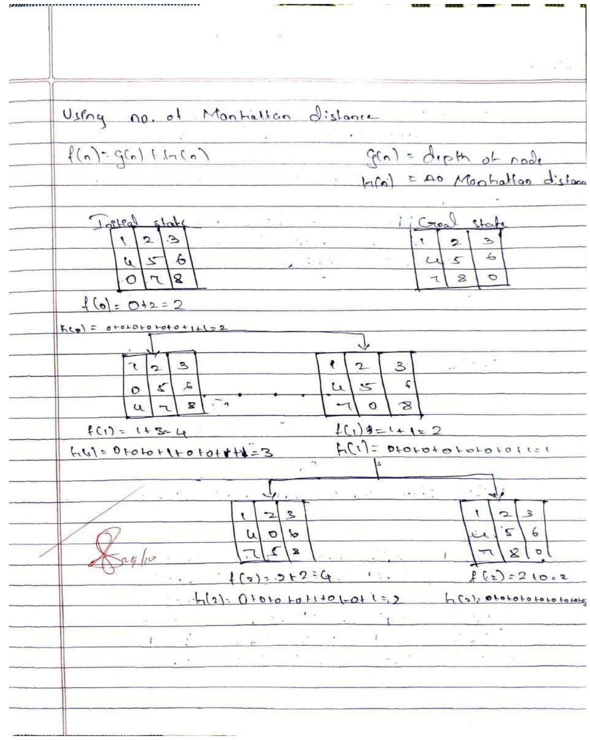
if empty? (frontier) then return failure

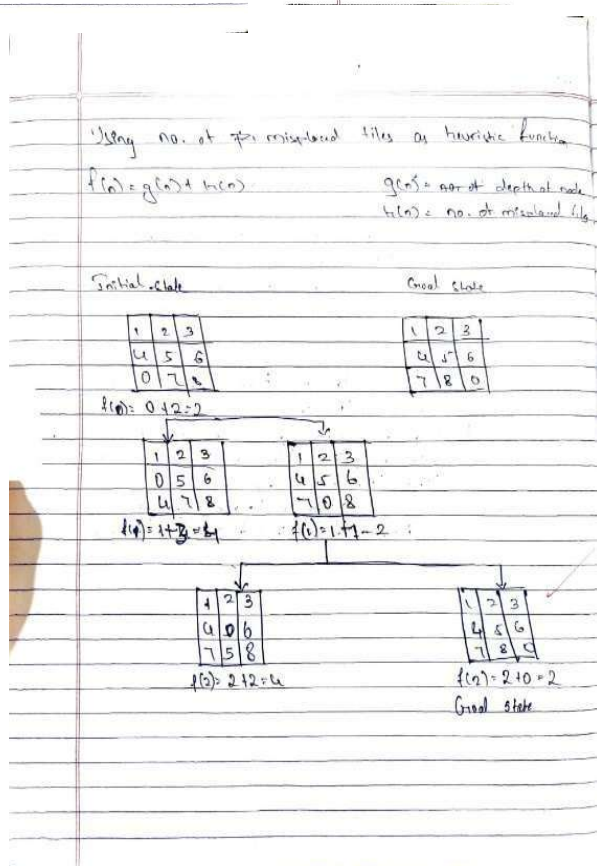
n ← pop (frontier)

if problem. goal Test (n.state) then return solution(n) for each action in problem action (n.state) do

n' ← childNode (problem, n, a)

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





Code (No. of Misplaced Tiles):

```
import heapq

class PuzzleNode:
    def __init__(self, state, parent=None, action=None, depth=0, cost=0):
        self.state = state          # Current state of the board
        self.parent = parent        # Parent node for path tracking
        self.action = action        # Move taken to reach this state
        self.depth = depth          # Depth of the node in the search tree
        self.cost = cost            # Total cost (f = g + h) for A* search

    def __lt__(self, other):
        return self.cost < other.cost

def get_misplaced_tiles(state, goal):
    """Calculate the number of misplaced tiles (excluding the blank)."""
    return sum(1 for i in range(3) for j in range(3) if state[i][j] != 0 and state[i][j] != goal[i][j])

def generate_successors(state):
    moves = []
    x, y = [(i, row.index(0)) for i, row in enumerate(state) if 0 in row][0]

    directions = {
        "UP": (x - 1, y),
        "DOWN": (x + 1, y),
        "LEFT": (x, y - 1),
        "RIGHT": (x, y + 1)
    }

    for action, (new_x, new_y) in directions.items():
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[x][y]
            moves.append((action, new_state))
    return moves

def a_star_search(initial, goal):
    start = PuzzleNode(initial, cost=get_misplaced_tiles(initial, goal))
    frontier = []
    heapq.heappush(frontier, start)
    explored = set()

    while frontier:
        current = heapq.heappop(frontier)

        # Display current state and cost
        print("Expanding node with state:")
        for row in current.state:
            print(row)
        print(f"Cost (f = g + h): {current.cost} (Depth: {current.depth}, Heuristic: {current.cost -
current.depth})\n")

        if current.state == goal:
            print("Goal reached!\n")
            return # Stop the search when the goal is reached

        explored.add(tuple(map(tuple, current.state)))

        for action, state in generate_successors(current.state):
            if tuple(map(tuple, state)) in explored:
```

```
                continue

            depth = current.depth + 1
            cost = depth + get_misplaced_tiles(state, goal)
            child = PuzzleNode(state, current, action, depth, cost)

            # Display successor info
            print(f"Generated successor by moving {action}:")
            for row in state:
                print(row)
            print(f"Successor cost (f = g + h): {cost} (Depth: {depth}, Heuristic: {cost - depth})\n")

            heapq.heappush(frontier, child)

    print("No solution found.")
    return None  # Return None if no solution is found

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

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

a_star_search(initial_state, goal_state)
print("Vatsal-1BM2CS323")
```

Output:

```
Expanding node with state:
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
Cost (f = g + h): 2 (Depth: 0, Heuristic: 2)

Generated successor by moving UP:
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]
Successor cost (f = g + h): 4 (Depth: 1, Heuristic: 3)

Generated successor by moving RIGHT:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
Successor cost (f = g + h): 2 (Depth: 1, Heuristic: 1)

Expanding node with state:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
Cost (f = g + h): 2 (Depth: 1, Heuristic: 1)

Generated successor by moving UP:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
Successor cost (f = g + h): 4 (Depth: 2, Heuristic: 2)

Generated successor by moving RIGHT:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Successor cost (f = g + h): 2 (Depth: 2, Heuristic: 0)

Expanding node with state:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Cost (f = g + h): 2 (Depth: 2, Heuristic: 0)

Goal reached!

Vatsal-1BM2CS323
```

Code(Manhattan Distance):

```
import heapq

class PuzzleNode:
    def __init__(self, state, parent=None, action=None, depth=0, cost=0):
        self.state = state          # Current state of the board
        self.parent = parent        # Parent node for path tracking
        self.action = action        # Move taken to reach this state
        self.depth = depth          # Depth of the node in the search tree
        self.cost = cost            # Total cost (f = g + h) for A* search

    def __lt__(self, other):
        return self.cost < other.cost

def  get_manhattan_distance(state,  goal):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                x, y = divmod(goal.index(state[i][j]), 3)
                distance += abs(x - i) + abs(y - j)
    return distance

def generate_successors(state):
    moves = []
    x, y = [(i, row.index(0)) for i, row in enumerate(state) if 0 in row][0]

    directions = {
        "UP": (x - 1, y),
        "DOWN": (x + 1, y),
        "LEFT": (x, y - 1),
        "RIGHT": (x, y + 1)
    }

    for action, (new_x, new_y) in directions.items():
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[x][y]
            moves.append((action, new_state))
    return moves

def  a_star_search(initial,  goal):
    goal_flat = sum(goal, [])
    start = PuzzleNode(initial, cost=get_manhattan_distance(initial, goal_flat))
    frontier = []
    heapq.heappush(frontier, start)
    explored = set()

    while frontier:
        current = heapq.heappop(frontier)

        # Display current state and cost
        print("Expanding node with state:")
        for row in current.state:
            print(row)
        print(f"Cost (f = g + h): {current.cost} (Depth: {current.depth}, Heuristic: {current.cost -
current.depth})\n")

        if current.state == goal:
            print("Goal reached!\n")
```

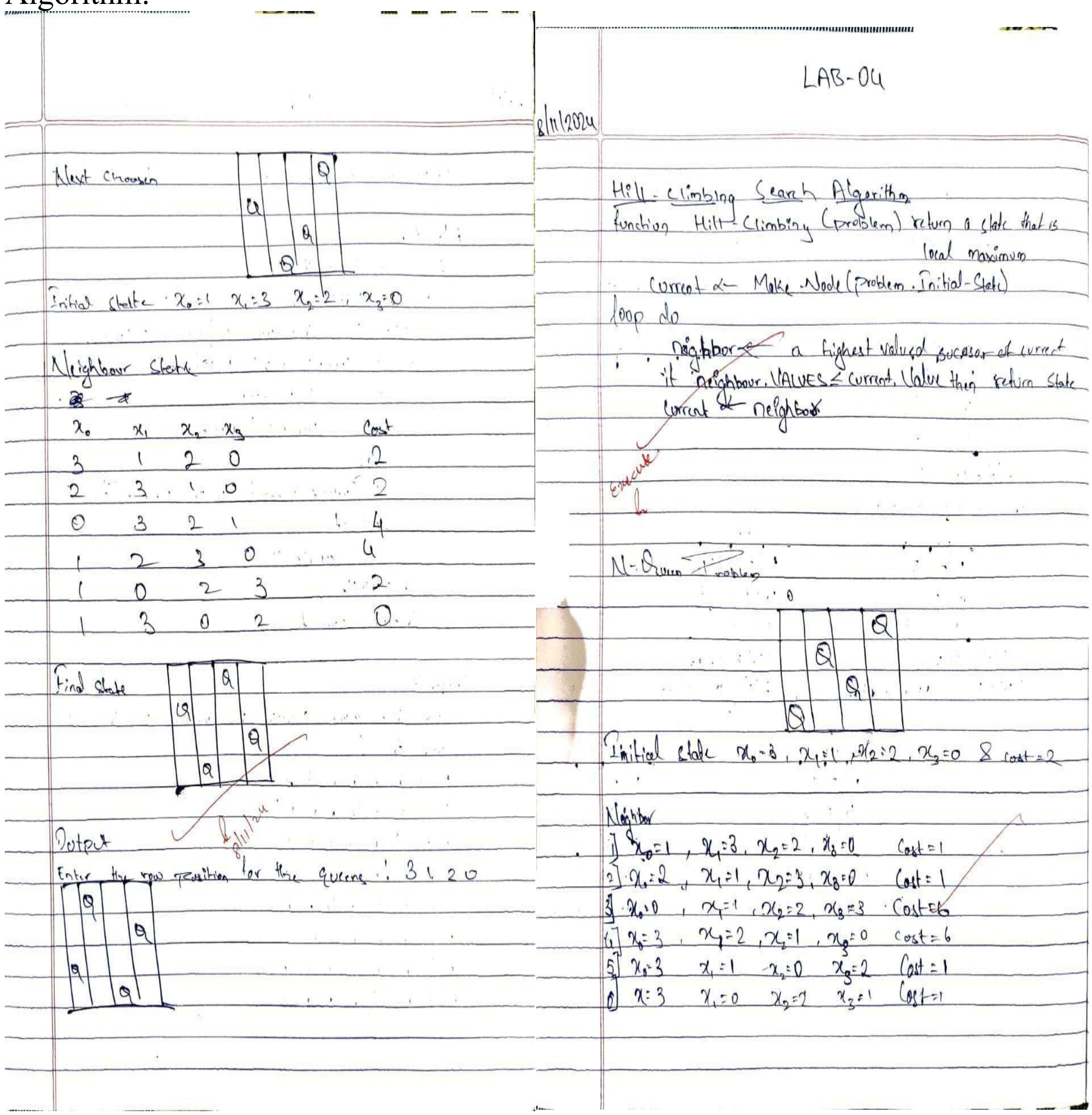
```
            return  # Stop the search when the goal is reached

        explored.add(tuple(map(tuple, current.state)))

        for action, state in generate_successors(current.state):
            if tuple(map(tuple, state)) in explored:
                continue

            depth = current.depth + 1
            cost = depth + get_manhattan_distance(state, goal_flat)
            child = PuzzleNode(state, current, action, depth, cost)

            # Display successor info
            print(f"Generated successor by moving {action}:")
            for row in state:
                print(row)
            print(f"Successor cost (f = g + h): {cost} (Depth: {depth}, Heuristic: {cost - depth})\n")

            heapq.heappush(frontier, child)

    print("No solution found.")
    return None  # Return None if no solution is found

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

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

a_star_search(initial_state, goal_state)
print("Vatsal-1BM2CS323")
```

Output:

```
Expanding node with state:
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
Cost (f = g + h): 2 (Depth: 0, Heuristic: 2)

Generated successor by moving UP:
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]
Successor cost (f = g + h): 4 (Depth: 1, Heuristic: 3)

Generated successor by moving RIGHT:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
Successor cost (f = g + h): 2 (Depth: 1, Heuristic: 1)

Expanding node with state:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
Cost (f = g + h): 2 (Depth: 1, Heuristic: 1)

Generated successor by moving UP:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
Successor cost (f = g + h): 4 (Depth: 2, Heuristic: 2)

Generated successor by moving RIGHT:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Successor cost (f = g + h): 2 (Depth: 2, Heuristic: 0)

Expanding node with state:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Cost (f = g + h): 2 (Depth: 2, Heuristic: 0)

Goal reached!

Vatsal-1BM2CS323
```

Program 5-Hill Climbing:

Algorithm:



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
print('Vatsal -1BM22CS323')
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

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:

```
Vatsal -1BM22CS323
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-Stimulated Annealing:

Algorithm:

15/11/24

LAB-5

N-queen implementation using Stimulated Annealing

Algorithm
Current ← initial state
while T>0 do
next ← a random neighbor of current
ΔE ← current.cost − next.cost
if $\Delta E > 0$ then
Current ← next
else
Current ← next with probability $P = e^{\Delta E/T}$
end if
decrease T
endwhile
return current

Output:-
Enter the column position for queens: 2 5 7 6 3 4 1 0
Solution found!

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

15/11/24

Code:

```
import random
import math

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  # Same column (vertical conflict)
            if abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1  # Diagonal conflict
    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  acceptance_probability(old_cost,  new_cost,  temperature):
    if new_cost < old_cost:
        return 1.0  # Always accept if the new state is better
    return math.exp((old_cost - new_cost) / temperature)

def simulated_annealing(n, initial_state, initial_temp, cooling_rate, max_iterations):
    state = initial_state
    current_cost = count_conflicts(state)
    temperature = initial_temp

    for iteration in range(max_iterations):
        neighbors = generate_neighbors(state)
        random_neighbor = random.choice(neighbors)
        new_cost =  count_conflicts(random_neighbor)

        if acceptance_probability(current_cost, new_cost, temperature) > random.random():
            state = random_neighbor
            current_cost = new_cost

        temperature *= cooling_rate  # Reduce the temperature

        if current_cost == 0:
            return state  # Solution found with no conflicts

    return None  # No solution found within the given iterations

def get_user_input(n):
    while True:
        try:
            user_input = input(f"Enter the column 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 = 8
initial_state = get_user_input(n)

# Parameters for simulated annealing
initial_temp = 1000  # Initial temperature
cooling_rate = 0.99  # Cooling rate
max_iterations = 10000  # Maximum number of iterations

# Run simulated annealing to find the solution
solution  =  simulated_annealing(n,  initial_state,  initial_temp,  cooling_rate,  max_iterations)

# Output the solution
if solution:
    print("Solution found!")
    # Printing the board
    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 within the given iterations.")
print('Vatsal - 1BM22CS323')
```

Output:

```
Enter the column positions for the queens (space-separated integers between 0 and 7): 3 6 2 7 1 4 0
    5
Solution found!
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
. . . . . Q . .
Vatsal - 1BM22CS323
```

Program-7 Unification:

Algorithm:

2/11/24

LAB-06

Unification :-

Algorithm Unify (Ψ_1, Ψ_2

Step 1 :- If Ψ_1 or Ψ_2 are variable or constant then :
    a) if Ψ_1 or Ψ_2 are identical, then return NIL :
    b) else if Ψ_1 is a variable,
        i] then if Ψ_1 occurs in Ψ_2 then return Failure
        ii] else return {(Ψ_1 / Ψ_2)}
    c) else if Ψ_2 is a variable,
        i] If Ψ_2 occur in Ψ_1, then return Failure   ii] else return {(Ψ_2/Ψ_1)}
    d] else return failure

Step 2 :- If the initial ~~predict~~ predicate symbol in Ψ_1 & Ψ_2 are not same, the return failure

Step 3 :- If Ψ_1 and Ψ_2 have different number of argument then return failure

Step 4 :- Set substitution set (SUBST) to NIL

Step 5 :- For i = 1 to the number of elements in Ψ_1
    a) call unify function with the ith element of Ψ_1 & ith element of Ψ_2 & 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 L_1 & L_2
        b] SUBST = APPEND (S, SUBST)

Step 6 - Return SUBST

Output
Enters two term to unify (eg., f(x,y); f(a,b)):
Enter first term : f(x,apple)
Enter first term : f(eats, y)
Unifying terms : ('f','x','apple') & ('f','eats','y')
Unification successful !
Substitution {'x':'eats', 'y':'apple'}
Unified expression :
Term 1 after substitution : ('f','eats','apple')
Term 2 after substitution : ('f', 'eats', 'apple')

22/11/24

Code:

```
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):
        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 = {}
    if x == y:
        return substitution
```

```
    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

    elif is_variable(y):
        return unify(y, x, substitution)

    elif isinstance(x, tuple) and isinstance(y, tuple):
        if x[0] != y[0]:
            raise ValueError(f"Unification fails: {x[0]} != {y[0]}")

        for a, b in zip(x[1:], y[1:]):
            substitution = unify(a, b, substitution)
        return substitution

    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()


    if term_str.islower() and len(term_str) == 1:
        return term_str


    if term_str.isalpha():
        return term_str


    if term_str.startswith('f(') and term_str.endswith(')'):
        func_str = term_str[2:-1]
        parts = func_str.split(',')
        return ('f', *[parse_term(p.strip()) for p in parts])


    raise ValueError(f"Invalid term format: {term_str}")

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)

        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)


if _name__== "_main_":
    main()
print("Vatsal - 1BM22CS323")
```

Output:

```
Enter two terms to unify (e.g., f(x, y), f(a, b)):
Enter first term: f(x,apple)
Enter second term: f(eats,y)
Unifying terms: ('f', 'x', 'apple') and ('f', 'eats', 'y')
Unification successful!
Substitution: {'x': 'eats', 'y': 'apple'}
Unified expression:
Term 1 after substitution: ('f', 'eats', 'apple')
Term 2 after substitution: ('f', 'eats', 'apple')
Vatsal - 1BM22CS323
```

Program-8 Forward Reasoning Algorithm:

Algorithm:

LAB-07

Forward Reasoning Algorithm

Algorithm

```
function FOL-FC-ASK(KB, α) returns a substitution or false
    input: KB, the knowledge base, a set of first-order definite
           α, the query, an atomic sentence
    local variable: new, the new sentence inferred on each iteration

  repeat until new is empty
        new ← { }
        for each rule in KB do
            (P1 ∧ ... ∧ Pn ⇒ q) ← STANDARDIZE-VARIABLE(rule)
            for each θ such that SUBST(θ, P1 ∧ ... ∧ Pn)
                                = SUBST(θ, P1' ∧ ... ∧ Pn')
                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
    return false
```

24/11/24

Output

Initial

It is raining

The ground is wet

Final Fact deduced:

It is raining

The ground is wet

People might slip

a] Emily is either a surgeon or lawyer:

Occupation (Emily, surgeon) ∨ Occupation (Emily, lawyer)

b] Joe is an actor, but also holds another Job:

Occupation (Joe, Actor) ∧ (∃o ≠ Actor Occupation (Joe, o))

c] All surgeon are doctors

∀p (Occupation (p, surgeon) → Occupation (p, Doctor))

d] Joe doesn't have a lawyer

∀p (Customer (Joe, p) → ¬Occupation (p, lawyer))

e] Emily has boss who is lawyer

∃b (Boss (b, Emily) ∧ Occupation (b, lawyer))

f] There exist a lawyer all of whose customer are doctors

∃p (Occupations (p, lawyer) ∧ ∀c (Customer (c, p) ⇒ Occupation (c, Doctor))

g] Every surgeon has a lawyer

∀p (Occupation (p, surgeon) → ∃c (Customer (p, c) ∧ Occupation (c, lawyer)))

24/11/24

Code:

```
class ForwardReasoningSystem:
    def __init__(self):
        self.facts = []  # List to store known facts
        self.rules = []  # List to store rules

    def add_fact(self, fact):
        """Add a new fact to the system."""
        if fact not in self.facts:
            self.facts.append(fact)

    def add_rule(self, premise, conclusion):
        """Add a rule to the system (IF premise THEN conclusion)."""
        self.rules.append((premise, conclusion))

    def apply_rule(self, rule):
        """Apply a single rule to the current facts."""
        premise, conclusion = rule
        if all(p in self.facts for p in premise) and conclusion not in self.facts:
            self.facts.append(conclusion)
            return True  # A new fact was added
        return False  # No new facts were added

    def forward_reasoning(self):
        """Iterate through rules and apply forward reasoning."""
        new_facts = True
        while new_facts:
            new_facts = False
            for rule in self.rules:
                if self.apply_rule(rule):
                    new_facts = True  # If a new fact was added, continue reasoning
        return self.facts


# Example usage

# Create the forward reasoning system
system = ForwardReasoningSystem()

# Add initial facts
system.add_fact("It is raining")
system.add_fact("The ground is wet")

# Add inference rules
system.add_rule(["It is raining"], "The ground is wet")  # IF raining THEN wet ground
system.add_rule(["The ground is wet"], "People might slip")  # IF wet ground THEN slip

# Run forward reasoning
all_facts = system.forward_reasoning()

# Display the results
print("Final Facts deduced:")
for fact in all_facts:
    print(fact)
print("-----------------------")
print("Vatsal - 1BM22CS323")
```

Output:

```
Final Facts deduced:
It is raining
The ground is wet
People might slip
---------------------
Vatsal - 1BM22CS323
```

Program-9 Alpha Beta Pruning:

Algorithm:

LAB-08

Alpha - Beta pruning
Function alpha-beta-Pruning (node, depth, alpha, Beta)
maximizing (player):
If depth == 0 or node is terminal:
Return Evaluate (node)
If maximizing-Player:
max-eval = -Infinity
for each child of node:
eval = alpha-beta-Pruning (child, depth-1, alpha, Beta, false)
max-eval = max (max-eval, eval)
alpha = max (alpha, eval)
If beta <= alpha:
Break
return max-eval
Else:
min-eval = Infinity
For each child node:
eval = alpha-beta-pruning (child, depth-1, alpha, Beta, true)
min-eval = min (min-eval, eval)
beta = min (beta, eval)
If Beta <= alpha:
Break
return min-eval

Output:
For tree = [[3,5,6], [9,1,2], [0,7,4]]
Optimal value: 6

20/12/24

Code:

```
def alpha_beta_pruning(node, depth, alpha, beta, maximizing_player):
    if depth == 0 or isinstance(node, int):
        return node

    if maximizing_player:
        max_eval = float('-inf')
        for child in node:
            eval = alpha_beta_pruning(child, depth - 1, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval
```

```
    else:
        min_eval = float('inf')
        for child in node:
            eval = alpha_beta_pruning(child, depth - 1, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval


# Proper tree structure for alpha-beta pruning
tree = [
    [[3, 5, 6], [9, 1, 2], [0, 7, 4]],

]

print("Optimal Value:", alpha_beta_pruning(tree, 3, float('-inf'), float('inf'), True))
print(“Vatsal – 1BM22CS323”)
```

Output:

```
Optimal Value: 6
Vatsal - 1BM22CS323
```

Program-10 FOL to CNF:

Algorithm:

(AI - '24)

Converting FOL into CNF

Input first order logic statement
Eliminate implication : Replace (A → B) using (¬A ∨ B)
Move ¬ (negation) inwards using De morgan's law
~~Standardize variables to the front (P — form)~~
Standardize variable : Ensure each quantifiers has unique variable
Move quantifiers to the front (Prenex form)
Skolemize : Eliminate existential quantifiers by introducing Skolem function
Drop universal quantifiers.
Distribute ∨ ~~over~~ over ∧ to obtain CNF form
Output CNF

Output :-
Original statement :- (A ∧ B) → C
CNF Form :- ¬A | ¬B | ¬C

20/11/24

Code:

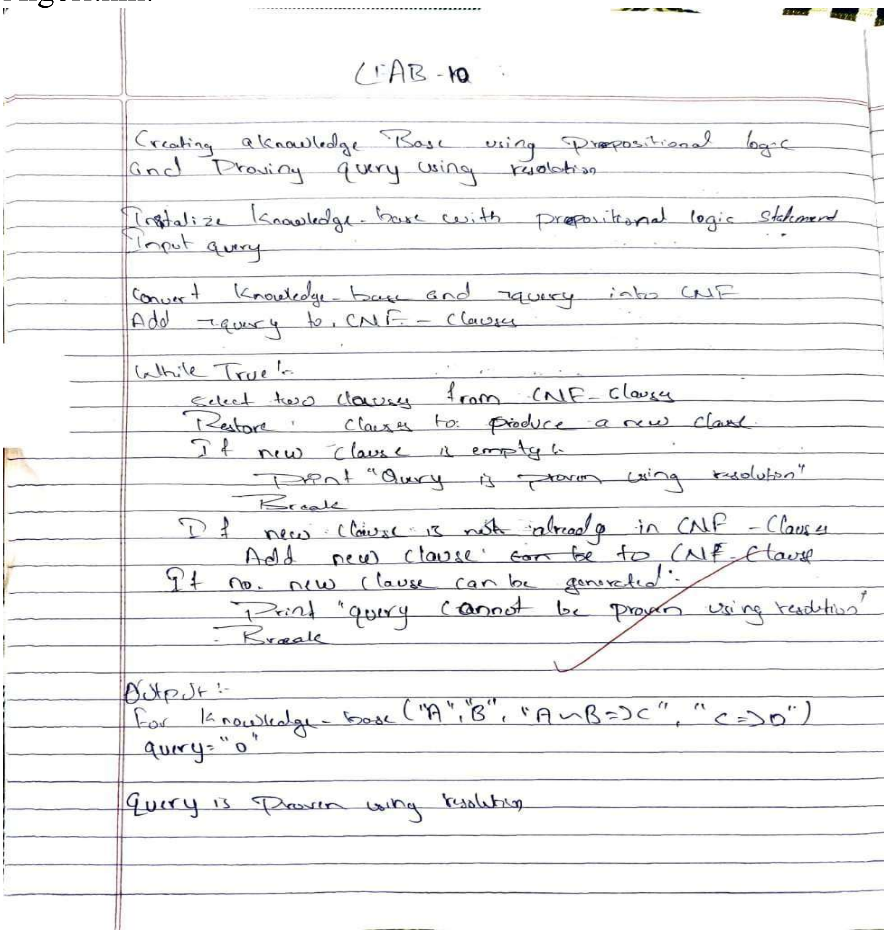
```
from sympy import symbols, Not, Or, And, Implies
from sympy.logic.boolalg import to_cnf

def convert_to_cnf(statement):
   return to_cnf(statement, simplify=True)


A, B, C = symbols("A B C")
fol_statement = Implies(A & B, C)

print("Original Statement:", fol_statement)
print("CNF Form:", convert_to_cnf(fol_statement))
```

Output:

```
Original Statement: (A & B) >> C
CNF Form: (~A | ~B | C)
```

Program-11 Proving Query using Resolution:

Algorithm:



LAB-10

Creating a knowledge Base using Propositional logic and Proving query using resolution

Initialize knowledge base with propositional logic statement
Input query

Convert knowledge base and query into CNF
Add ¬query to CNF-clauses

While True:
  Select two clauses from CNF-clauses
  Resolve clauses to produce a new clause
  If new clause is empty:
    Print "Query is proven using resolution"
    Break
  If new clause is not already in CNF-clauses
    Add new clause ~~can be~~ to CNF-clause
  If no new clause can be generated:
    Print "query cannot be proven using resolution"
    Break

Output:
For knowledge-base ("A", "B", "A∧B=>C", "C=>D")
query = "D"

Query is proven using resolution

Code:

```
from itertools import combinations


def resolve(clause1, clause2):
    for literal in clause1:
        complementary_literal = f"~{literal}" if not literal.startswith("~") else literal[1:]
        if complementary_literal in clause2:
            new_clause = (clause1 | clause2) - {literal, complementary_literal}
            return new_clause
    return None


def resolution(knowledge_base, query):
    # Convert the knowledge base into a set of clauses
    clauses = [set(clause.split()) for clause in knowledge_base]
    # Add the negation of the query
    clauses.append(set(f"~{query}".split()))

    # Perform resolution
    while True:
        new_clauses = []
```

```
        for (ci, cj) in combinations(clauses, 2):
            resolvent = resolve(ci, cj)
            if resolvent is not None:
                if not resolvent:  # Empty set means the query is proven
                    return True
                new_clauses.append(resolvent)

        # Check if new clauses introduce anything novel
        if not any(new_clause not in clauses for new_clause in new_clauses):
            return False  # No new information, query cannot be proven

        # Add new clauses to the knowledge base
        for new_clause in new_clauses:
            if new_clause not in clauses:
                clauses.append(new_clause)


# Example usage
knowledge_base = [
    "A",
    "~A B",
    "~B C",
    "~C D"
]
query = "D"

if resolution(knowledge_base, query):
    print("Query is proven using resolution.")
else:
    print("Query cannot be proven using resolution.")
print("Vatsal - 1BM22CS323")
```

Output:

```
Query is proven using resolution.
Vatsal - 1BM22CS323
```

Program-12 Proving Query Entails With KB or Not:

Algorithm:

LAB-10

Knowledge Base using propositional logic

Initalize knowledge base with propositional logic statement
Input query
If ~~fordw~~ forward-chaining(knowledge base, query):
    Print "Query is entailed by the knowledge Base"
Else:
    Print "Query is not entailed by knowledge Base"

Function forward-chaining(knowledge-base, query):
    Initalize agenda with known facts from knowledge base
    while agenda is not empty:
        Pop a fact from agenda
        If fact matches query
            Return True
        For each rule in knowledge-base
            In fact satisfy a rule's premise:
                Add the rule's conclusion to agenda
    Return False

20/12/24

Output:
For the knowledge-Base = ["A", "B", "A&B" => "C", "C => D"]
Query = "D"

Query is entailed by the knowledge Base

Code:

```
def   KB_entails(knowledge_base,  query):
    agenda = [fact for fact in knowledge_base if "=>" not in fact]
    inferred = set()

    while agenda:
        fact = agenda.pop(0)
        if fact == query:
            return True
        inferred.add(fact)
        for rule in knowledge_base:
            if "=>" in rule:
                premise, conclusion = rule.split("=>")
                premises = premise.split("&")
                if all(p.strip() in inferred for p in premises) and conclusion.strip() not in inferred:
                    agenda.append(conclusion.strip())
    return False
```

```
knowledge_base = [
    "A",
    "B",
    "A & B => C",
    "C => D"
]
query = "D"

if forward_chaining(knowledge_base, query):
    print("Query is entailed by the knowledge base.")
else:
    print("Query is not entailed by the knowledge base.")
print("Vatsal - 1BM22CS323")
```

Output:

```
Query is entailed by the knowledge base.
Vatsal - 1BM22CS323
```