# VISVESVARAYA TECHNOLOGICAL UNIVERSITY

**"JnanaSangama", Belgaum -590014, Karnataka.**

**LAB REPORT**
**on**

# Artificial Intelligence (23CS5PCAIN)

***Submitted by***

**Navanidhi D J (1BM23CS204)**

***in partial fulfillment for the award of the degree of***
**BACHELOR OF ENGINEERING**
***in***
**COMPUTER SCIENCE AND ENGINEERING**

**B.M.S. COLLEGE OF ENGINEERING**
**(Autonomous Institution under VTU)**
**BENGALURU-560019**
**Aug-2025 to Dec-2025**

**B.M.S. College of Engineering,**
**Bull Temple Road, Bangalore 560019**
(Affiliated To Visvesvaraya Technological University, Belgaum)
**Department of Computer Science and Engineering**

## CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Navanidhi D J(1BM23CS204),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

| | |
|---|---|
| Roopashree C S<br>Assistant Professor<br>Department of CSE, BMSCE | Dr. Kavitha Sooda<br>Professor & HOD<br>Department of CSE, BMSCE |

# Index



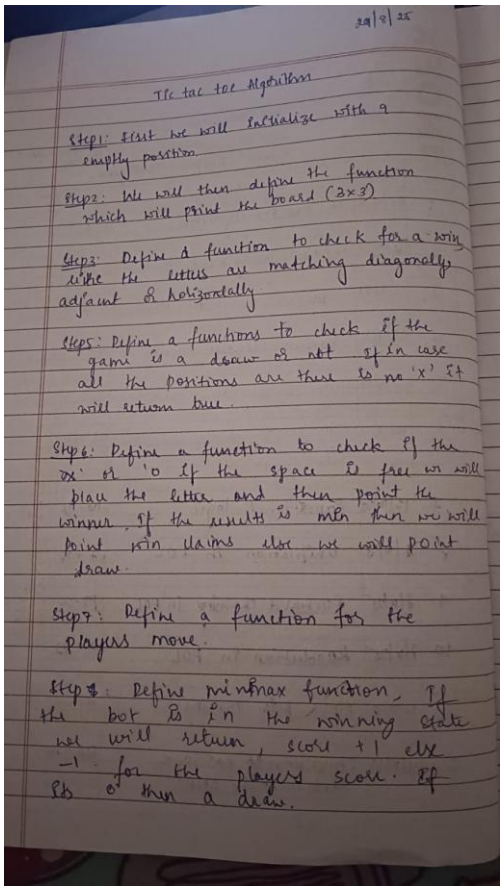
Github Link:
https://github.com/NavanidhiDJ/Artificial_intelligence-

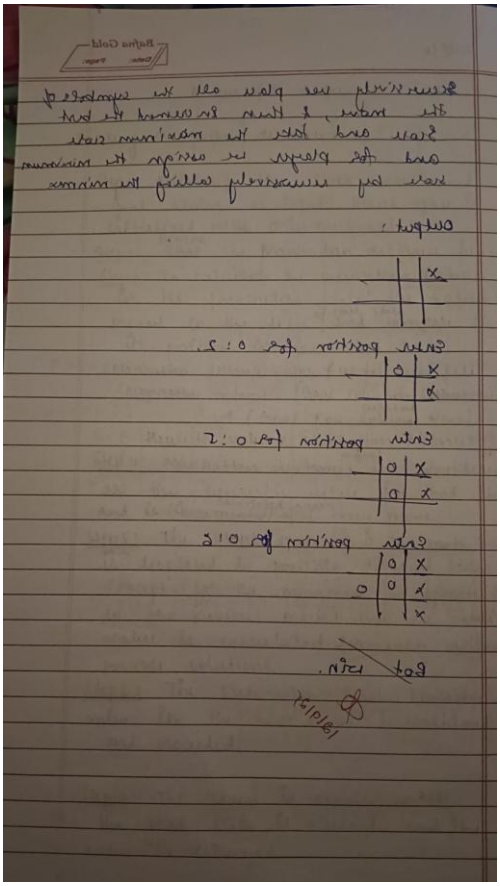
**Program 1**

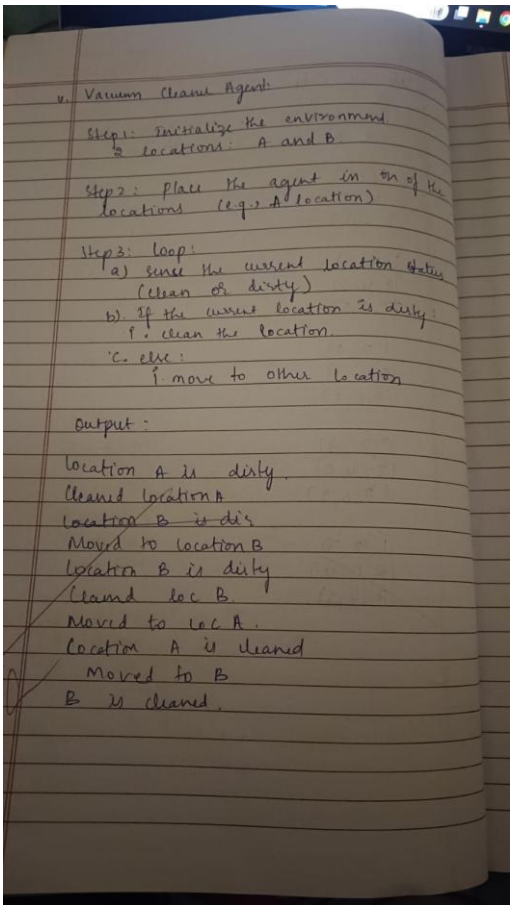
Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Algorithm:






CODE:

Tic Tac Toe:

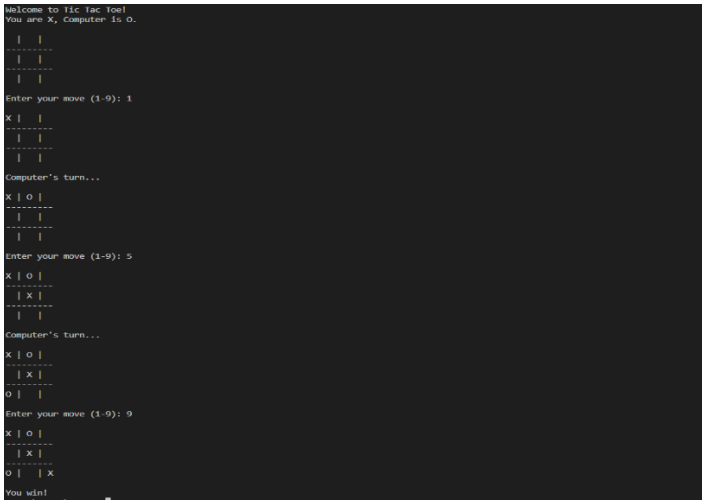
```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 5)

def check_winner(board, player):
    for row in board:
        if all(cell == player for cell in row):
            return True
    for col in range(3):
        if all(board[row][col] == player for row in range(3)):
            return True
    if all(board[i][i] == player for i in range(3)):
        return True
    if all(board[i][2 - i] == player for i in range(3)):
        return True
    return False
```

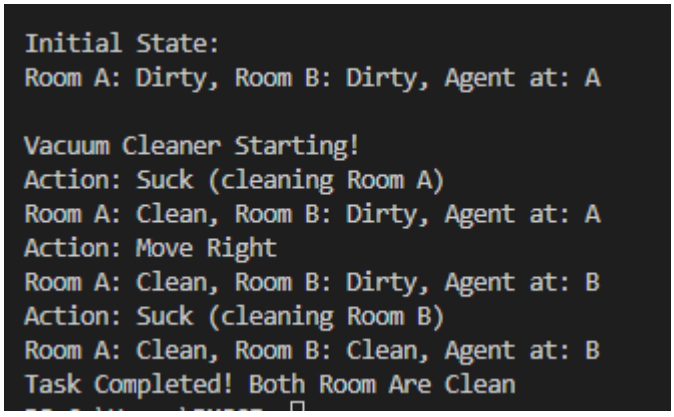
```
def is_draw(board):
    return all(cell != " " for row in board for cell in row)

def play_game():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X"
    while True:
        print_board(board)
        print(f"Player {current_player}'s turn.")
        try:
            row = int(input("Enter row (0, 1, 2): "))
            col = int(input("Enter col (0, 1, 2): "))
        except ValueError:
            print("Invalid input. Please enter numbers 0, 1, or 2.")
            continue
        if row not in [0,1,2] or col not in [0,1,2] or board[row][col] != " ":
            print("Invalid move. Try again.")
            continue
        board[row][col] = current_player
        if check_winner(board, current_player):
            print_board(board)
            print(f"🎉 Player {current_player} wins!")
            break
        if is_draw(board):
            print_board(board)
            print("It's a draw!")
            break
        current_player = "O" if current_player == "X" else "X"

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

VACUUM CLEANER AGENT :

```
class VacuumCleanerAgent:
    def __init__(self):
        self.location = 'A'

    def perceive_and_act(self, environment):
        current_status = environment[self.location]

        if current_status == 'Dirty':
            action = 'Suck'
            environment[self.location] = 'Clean'
        else:
            action = 'Move Right' if self.location == 'A' else 'Move Left'
            self.location = 'B' if self.location == 'A' else 'A'
```

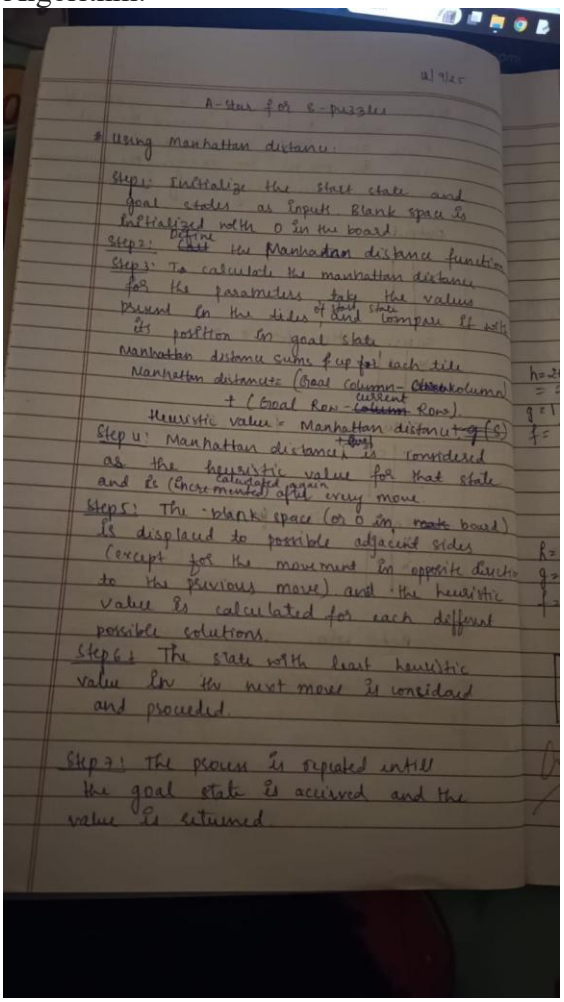
```
        return action

def run_vacuum_agent(environment):
    agent = VacuumCleanerAgent()
    steps = 0

    print(f"Initial environment: {environment}")
    while 'Dirty' in environment.values():
        action = agent.perceive_and_act(environment)
        print(f"Step {steps}: Agent at {agent.location}, Action: {action}, Environment:
{environment}")
        steps += 1

    print("Environment is clean!")

if __name__ == "__main__":
    env = {'A': 'Dirty', 'B': 'Dirty'}
    run_vacuum_agent(env)
```



## 2. 8 Puzzle using A star(Manhatan)
## 8 Puzzle using A star(Misplaced Tiles)

Algorithm:


A-star for 8-puzzle

# Using Manhattan distance:

Step1: Initialize the start state and goal states as input. Blank space is initialized with 0 in the board.

Step2: Call the Manhattan distance function

Step3: To calculate the manhattan distance for the parameters, take the values present in the tiles of the current state and compare it with its position in goal state.

Manhattan distance sums up for each tile

Manhattan distance = (Goal Column - current column) + (Goal Row - current Row).

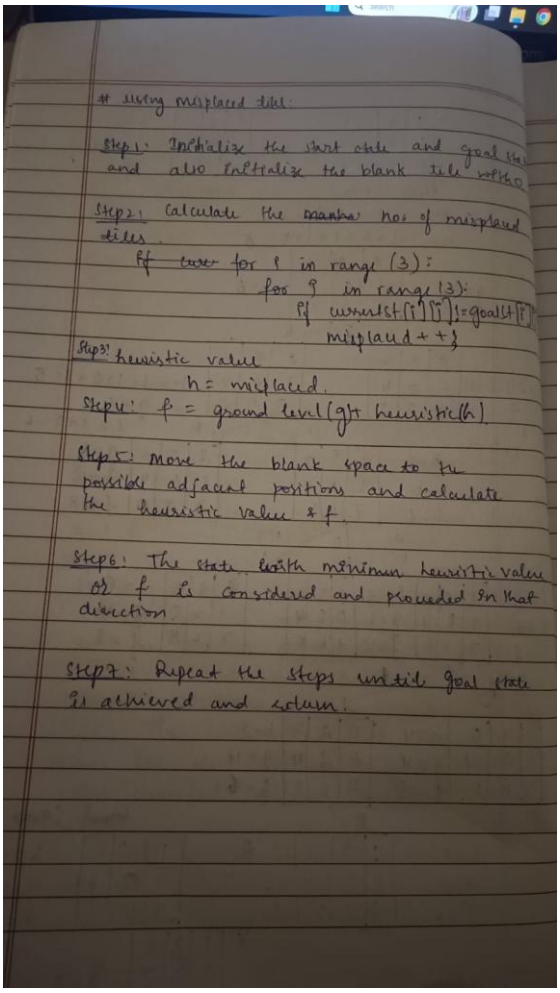
Heuristic value = Manhattan distance + g(s)

Step4: Manhattan distance is considered as the heuristic value for that state and is (incremented) calculated again after every move

Step5: The 'blank' space (or 0 in the board) is displaced to possible adjacent sides (except for the movement in opposite direction to the previous move) and the heuristic value is calculated for each different possible solutions.

Step6: The state with least heuristic value for the next move is considered and proceeded.

Step7: The process is repeated until the goal state is accessed and the value is returned.


# using misplaced tiles:

Step1: Initialize the start state and goal state and also initialize the blank tile with 0

Step2: Calculate the no: of misplaced tiles.

If user for i in range (3):
for j in range(3):
if currstate[i][j]!=goalstate[i][j]
misplaced++;

Step3: heuristic value
h = misplaced.

Step4: f = ground level(g) + heuristic(h).

Step5: Move the blank space to the possible adjacent positions and calculate the heuristic value & f.

Step6: The state with minimum heuristic value or f is considered and proceeded in that direction.

Step7: Repeat the steps until goal state is achieved and return.

Code:

```
import heapq

def misplaced_tiles(state, goal):
    return sum(1 for i in range(9) if state[i] != 0 and state[i] != goal[i])

def get_neighbors(state):
    neighbors = []
    zero_pos = state.index(0)
    x, y = divmod(zero_pos, 3)
    directions = [(-1,0), (1,0), (0,-1), (0,1)]

    for dx, dy in directions:
        nx, ny = x + dx, y + dy
```

```
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_pos = nx * 3 + ny
            new_state = list(state)
            new_state[zero_pos], new_state[new_pos] = new_state[new_pos], new_state[zero_pos]
            neighbors.append(tuple(new_state))
    return neighbors

def a_star(start, goal):
    open_set = []
    heapq.heappush(open_set, (0 + misplaced_tiles(start, goal), 0, start, []))
    closed_set = set()

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

        if current == goal:
            return path + [current]

        if current in closed_set:
            continue
        closed_set.add(current)

        for neighbor in get_neighbors(current):
            if neighbor in closed_set:
                continue
            new_g = g + 1
            new_f = new_g + misplaced_tiles(neighbor, goal)
            heapq.heappush(open_set, (new_f, new_g, neighbor, path + [current]))

    return None

def print_puzzle(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()

def input_state(prompt):
    while True:
        try:
            s = input(prompt)
            lst = tuple(int(x) for x in s.strip().split())
            if len(lst) != 9 or set(lst) != set(range(9)):
                raise ValueError
            return lst
        except ValueError:
            print("Invalid input! Please enter 9 numbers from 0 to 8 separated by spaces, no repeats.")
```

```
def main():
    print("Enter the start state (0 represents the blank):")
    start = input_state("Start state: ")
    print("Enter the goal state (0 represents the blank):")
    goal = input_state("Goal state: ")

    print("\nSolving...")
    path = a_star(start, goal)

    if path is None:
        print("No solution found.")
    else:
        print(f"Solution found in {len(path) - 1} moves:")
        for step in path:
            print_puzzle(step)

if __name__ == "__main__":
    main()


import heapq

def manhattan_distance(state, goal):
    distance = 0
    for num in range(1, 9):
        x1, y1 = divmod(state.index(num), 3)
        x2, y2 = divmod(goal.index(num), 3)
        distance += abs(x1 - x2) + abs(y1 - y2)
    return distance

def get_neighbors(state):
    neighbors = []
    zero_pos = state.index(0)
    x, y = divmod(zero_pos, 3)
    directions = [(-1,0), (1,0), (0,-1), (0,1)]

    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_pos = nx * 3 + ny
            new_state = list(state)
            new_state[zero_pos], new_state[new_pos] = new_state[new_pos], new_state[zero_pos]
            neighbors.append(tuple(new_state))
    return neighbors

def a_star(start, goal):
    open_set = []
```

```
    heapq.heappush(open_set, (0 + manhattan_distance(start, goal), 0, start, []))
    closed_set = set()

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

        if current == goal:
            return path + [current]

        if current in closed_set:
            continue
        closed_set.add(current)

        for neighbor in get_neighbors(current):
            if neighbor in closed_set:
                continue
            new_g = g + 1
            new_f = new_g + manhattan_distance(neighbor, goal)
            heapq.heappush(open_set, (new_f, new_g, neighbor, path + [current]))

    return None

def print_puzzle(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()

def input_state(prompt):
    while True:
        try:
            s = input(prompt)
            lst = tuple(int(x) for x in s.strip().split())
            if len(lst) != 9 or set(lst) != set(range(9)):
                raise ValueError
            return lst
        except ValueError:
            print("Invalid input! Please enter 9 numbers from 0 to 8 separated by spaces, no repeats.")

def main():
    print("Enter the start state (0 represents the blank):")
    start = input_state("Start state: ")
    print("Enter the goal state (0 represents the blank):")
    goal = input_state("Goal state: ")

    print("\nSolving...")
    path = a_star(start, goal)
```

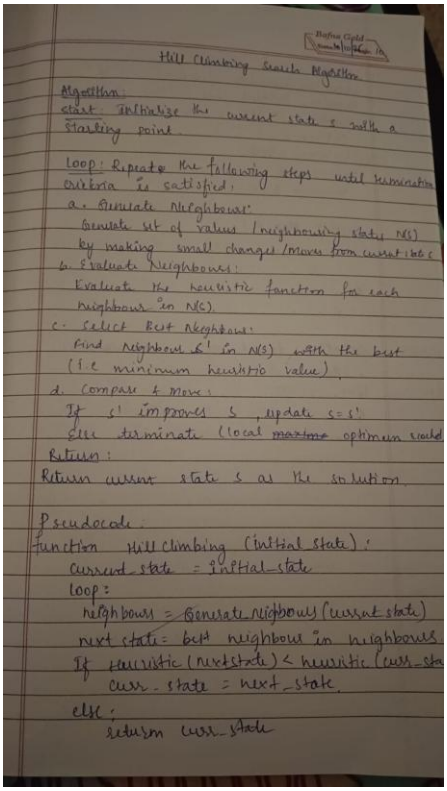
```
    if path is None:
        print("No solution found.")
    else:
        print(f"Solution found in {len(path) - 1} moves:")
        for step in path:
            print_puzzle(step)

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

```
--- Manhattan Distance Heuristic ---
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

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

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

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

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

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


[Done] exited with code=0 in 0.104 seconds
```

```
--- Misplaced Tiles Heuristic ---
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

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

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

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

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

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


[Done] exited with code=0 in 0.106 seconds
```

## 3. Hill Climbing Search Algorithm

Algirithm:




```
import random

def heuristic(state):
    h = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                h += 1
    return h

def get_neighbors(state):
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if state[col] != row:
                neighbor = list(state)
                neighbor[col] = row
                neighbors.append(neighbor)
    return neighbors
```

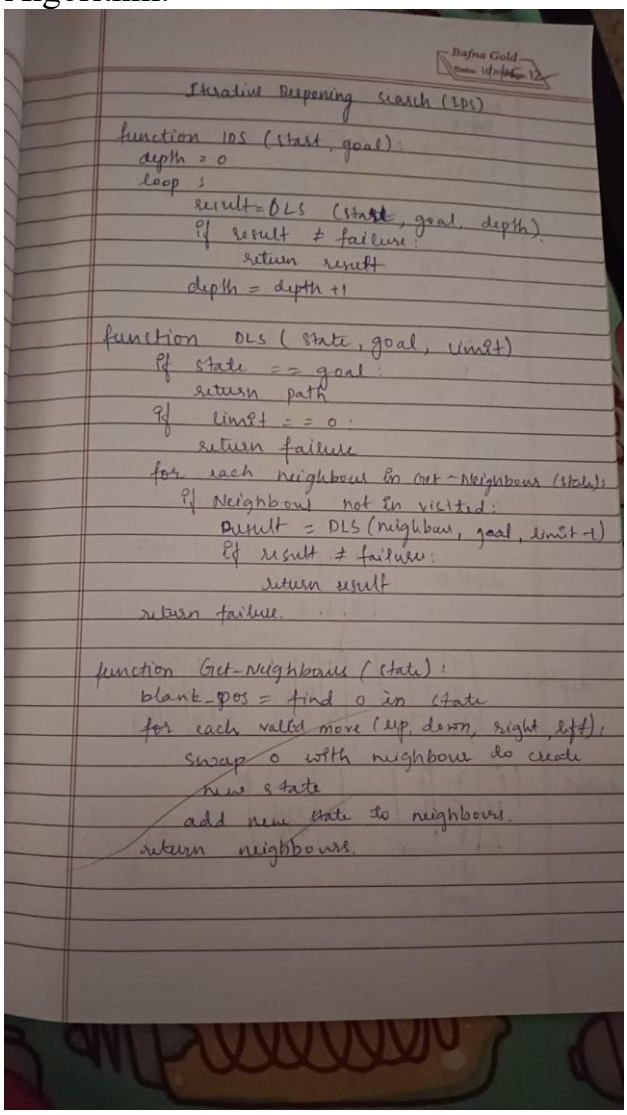
```
def hill_climbing(initial_state):
    current = list(initial_state)
    current_h = heuristic(current)

    while True:
        neighbors = get_neighbors(current)
        neighbor_h = [(heuristic(neighbor), neighbor) for neighbor in neighbors]

        neighbor_h.sort(key=lambda x: x[0])
        best_h, best_neighbor = neighbor_h[0]

        if best_h >= current_h:
            break  # No improvement

        current, current_h = best_neighbor, best_h

    return current, current_h

def random_restart_hill_climbing(initial_state, max_restarts=1000):
    n = len(initial_state)
    current_state = list(initial_state)

    for restart in range(max_restarts):
        solution, h = hill_climbing(current_state)
        if h == 0:
            return solution, h, restart

        current_state = [random.randint(0, n - 1) for _ in range(n)]
    return solution, h, max_restarts

def print_board(state):
    n = len(state)
    for row in range(n):
        line = ""
        for col in range(n):
            line += " Q " if state[col] == row else " . "
        print(line)
    print()

if __name__ == "__main__":
    while True:
        try:
            N = int(input("Enter the number of queens (N): "))
            if N > 3:
                break
            else:
```

```
            print("N must be at least 4.")
        except ValueError:
            print("Invalid input. Please enter an integer.")

    print(f"Enter the initial row positions for each of the {N} columns (0 to {N-1}):")
    initial_state = []
    for col in range(N):
        while True:
            try:
                row = int(input(f"Column {col}: "))
                if 0 <= row < N:
                    initial_state.append(row)
                    break
                else:
                    print(f"Please enter a number between 0 and {N-1}.")
            except ValueError:
                print("Invalid input. Enter an integer.")

    print("\nInitial board:")
    print_board(initial_state)

    solution, h, restarts = random_restart_hill_climbing(initial_state, max_restarts=1000)

    if h == 0:
        print(f"Solution found after {restarts} restarts:")
        print_board(solution)
    else:
        print(f"No solution found after {restarts} restarts. Local optimum with heuristic {h}:")
        print_board(solution)
```

```
Initial State: [3, 1, 2, 0] with 2 attacks
---------------------
| .. | .. | .. | Q  |
---------------------
| .. | Q  | .. | .. |
---------------------
| .. | .. | Q  | .. |
---------------------
| Q  | .. | .. | .. |
---------------------


=== Running Hill Climbing with Random Restarts ===
Solution found after 2 restart(s)!
Final State: [1, 3, 0, 2] with 0 attacks
---------------------
| .. | .. | Q  | .. |
---------------------
| Q  | .. | .. | .. |
---------------------
| .. | .. | .. | Q  |
---------------------
| .. | Q  | .. | .. |
---------------------
```

## 4. Iterative deepening search algorithm

Algorithm:



Iterative Deepening search (IDS)

function IDS (start, goal):
depth = 0
loop :
result = DLS (start, goal, depth)
if result ≠ failure:
return result
depth = depth + 1

function DLS ( state, goal, limit)
if state == goal:
return path
if limit == 0:
return failure
for each neighbour in Get-Neighbours (state):
if Neighbour not in visited:
Result = DLS (neighbour, goal, limit-1)
if result ≠ failure:
return result
return failure.

function Get-Neighbours ( state):
blank_pos = find 0 in state
for each valid move (up, down, right, left),
swap 0 with neighbour to create
new state
add new state to neighbours.
return neighbours.

Code:

```
from collections import deque

# Define the goal state
GOAL_STATE = ((1, 2, 3),
              (4, 5, 6),
              (7, 8, 0))

# Moves (up, down, left, right)
MOVES = [(-1, 0), (1, 0), (0, -1), (0, 1)]
```

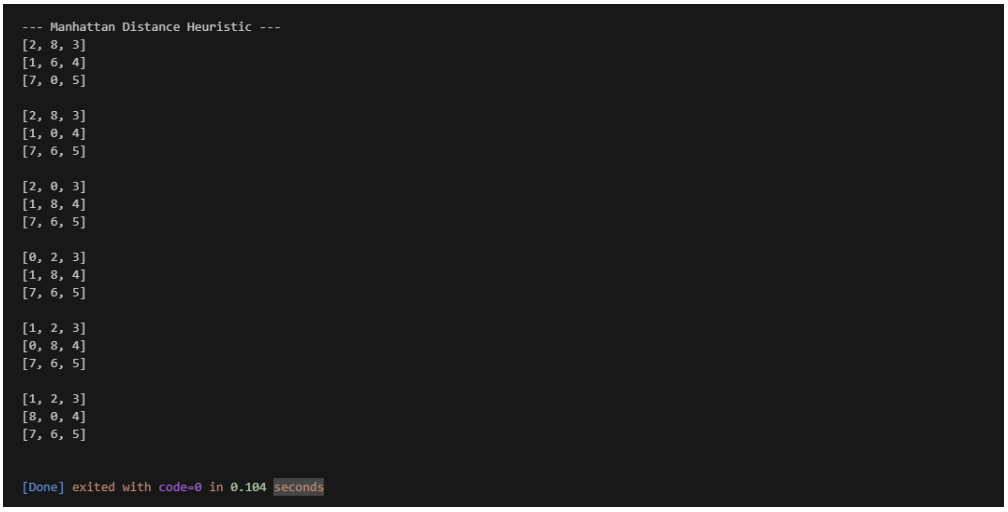
```
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None


def swap_positions(state, pos1, pos2):
    state_list = [list(row) for row in state]
    x1, y1 = pos1
    x2, y2 = pos2
    state_list[x1][y1], state_list[x2][y2] = state_list[x2][y2], state_list[x1][y1]
    return tuple(tuple(row) for row in state_list)


def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    for dx, dy in MOVES:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = swap_positions(state, (x, y), (nx, ny))
            neighbors.append(new_state)
    return neighbors


def depth_limited_search(state, goal, limit, path, visited):
    if state == goal:
        return path
    if limit <= 0:
        return None

    visited.add(state)

    for neighbor in get_neighbors(state):
        if neighbor not in visited:
            result = depth_limited_search(neighbor, goal, limit - 1, path + [neighbor], visited)
            if result is not None:
                return result

    visited.remove(state)
    return None


def iterative_deepening_search(start, goal):
    depth = 0
```

```
    while True:
        visited = set()
        print(f"Searching at depth: {depth}")
        result = depth_limited_search(start, goal, depth, [start], visited)
        if result is not None:
            return result
        depth += 1


def print_state(state):
    for row in state:
        print(' '.join(str(x) if x != 0 else ' ' for x in row))
    print()


if __name__ == "__main__":
    # Example start state
    start_state = ((1, 2, 3),
                   (4, 0, 6),
                   (7, 5, 8))

    solution_path = iterative_deepening_search(start_state, GOAL_STATE)

    print(f"Solution found in {len(solution_path) - 1} moves:\n")
    for step in solution_path:
        print_state(step)


import heapq
import copy

# Goal state
GOAL_STATE = [[1, 2, 3],
              [4, 5, 6],
              [7, 8, 0]]

DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right


def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            val = state[i][j]
            if val != 0:
                goal_x = (val - 1) // 3
                goal_y = (val - 1) % 3
```

```
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance


def misplaced_tiles(state):
    count = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != GOAL_STATE[i][j]:
                count += 1
    return count


def get_blank_pos(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j


def is_goal(state):
    return state == GOAL_STATE


def valid(x, y):
    return 0 <= x < 3 and 0 <= y < 3


def heuristic_fn(state, heuristic):
    if heuristic == "manhattan":
        return manhattan_distance(state)
    elif heuristic == "misplaced":
        return misplaced_tiles(state)
    else:
        raise ValueError("Unknown heuristic: choose 'manhattan' or 'misplaced'")


def a_star(start_state, heuristic="manhattan"):
    visited = set()
    heap = []

    g = 0
    h = heuristic_fn(start_state, heuristic)
    f = g + h

    heapq.heappush(heap, (f, g, start_state, []))  # (f, g, state, path)
```

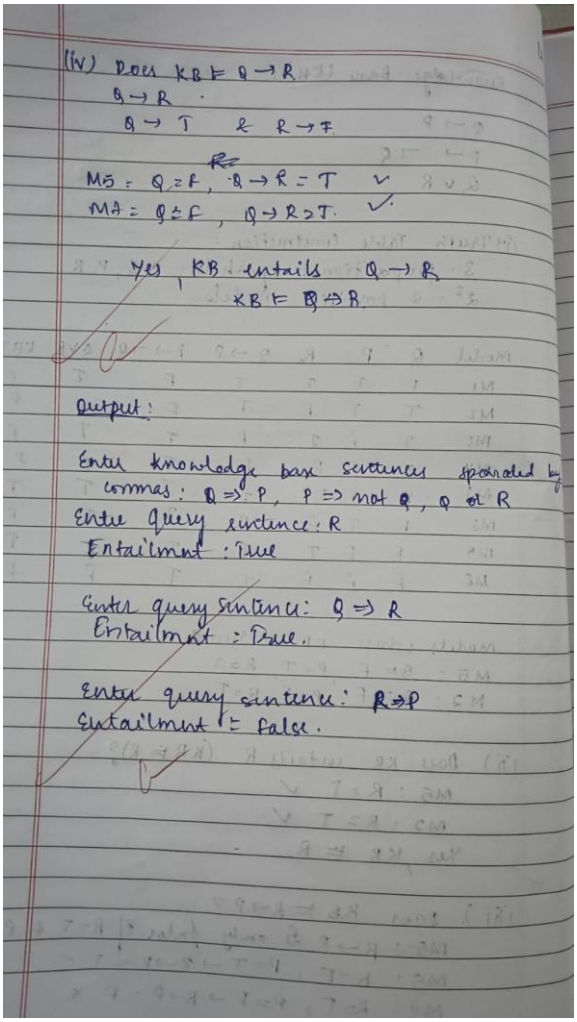
```
    while heap:
        f, g, state, path = heapq.heappop(heap)

        state_tuple = tuple(tuple(row) for row in state)
        if state_tuple in visited:
            continue

        visited.add(state_tuple)

        if is_goal(state):
            return path + [state]

        x, y = get_blank_pos(state)

        for dx, dy in DIRECTIONS:
            nx, ny = x + dx, y + dy
            if valid(nx, ny):
                new_state = copy.deepcopy(state)
                new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
                new_state_tuple = tuple(tuple(row) for row in new_state)
                if new_state_tuple not in visited:
                    new_g = g + 1
                    new_h = heuristic_fn(new_state, heuristic)
                    new_f = new_g + new_h
                    heapq.heappush(heap, (new_f, new_g, new_state, path + [state]))

    return None


def print_path(path):
    for step, state in enumerate(path):
        print(f"Step {step}:")
        for row in state:
            print(row)
        print("")


# Example usage:
if __name__ == "__main__":
    # Change this to test different starting positions
    start_state = [[1, 2, 3],
                   [4, 0, 6],
                   [7, 5, 8]]

    print("Using Manhattan Distance Heuristic:")
    path_manhattan = a_star(start_state, heuristic="manhattan")
    if path_manhattan:
```

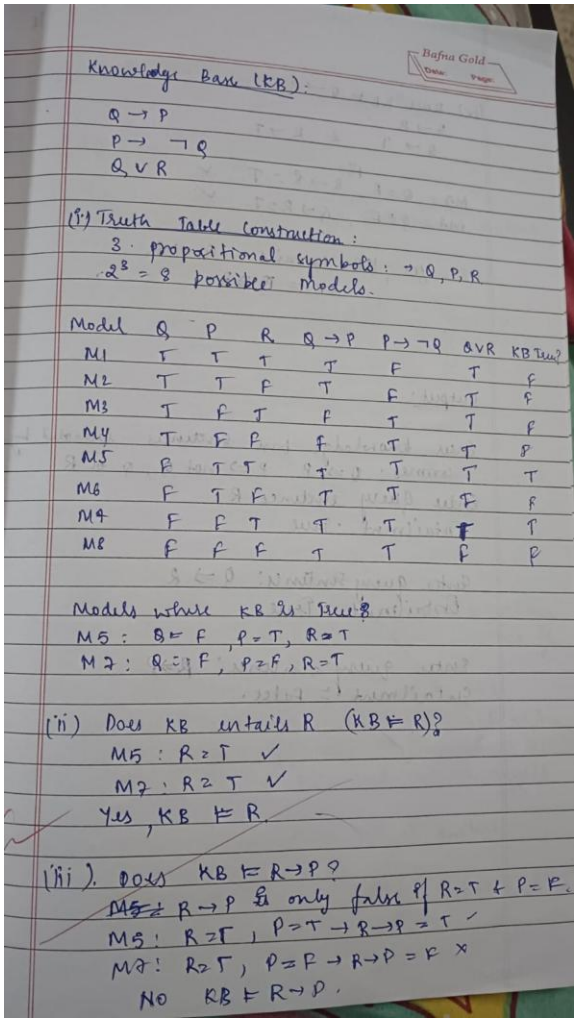
```
    print_path(path_manhattan)
    print(f"Total steps (Manhattan): {len(path_manhattan) - 1}")
else:
    print("No solution found with Manhattan.")

print("\nUsing Misplaced Tiles Heuristic:")
path_misplaced = a_star(start_state, heuristic="misplaced")
if path_misplaced:
    print_path(path_misplaced)
    print(f"Total steps (Misplaced Tiles): {len(path_misplaced) - 1}")
else:
    print("No solution found with Misplaced Tiles.")
```

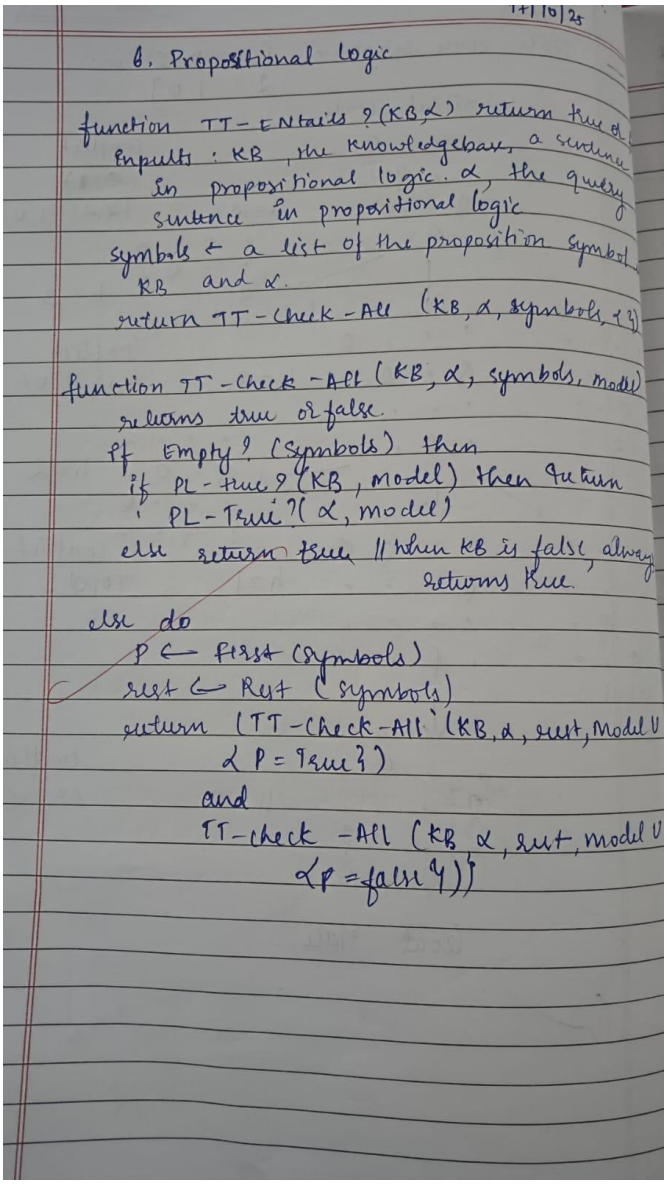


**5. Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.**

Algorithm:






## Code:

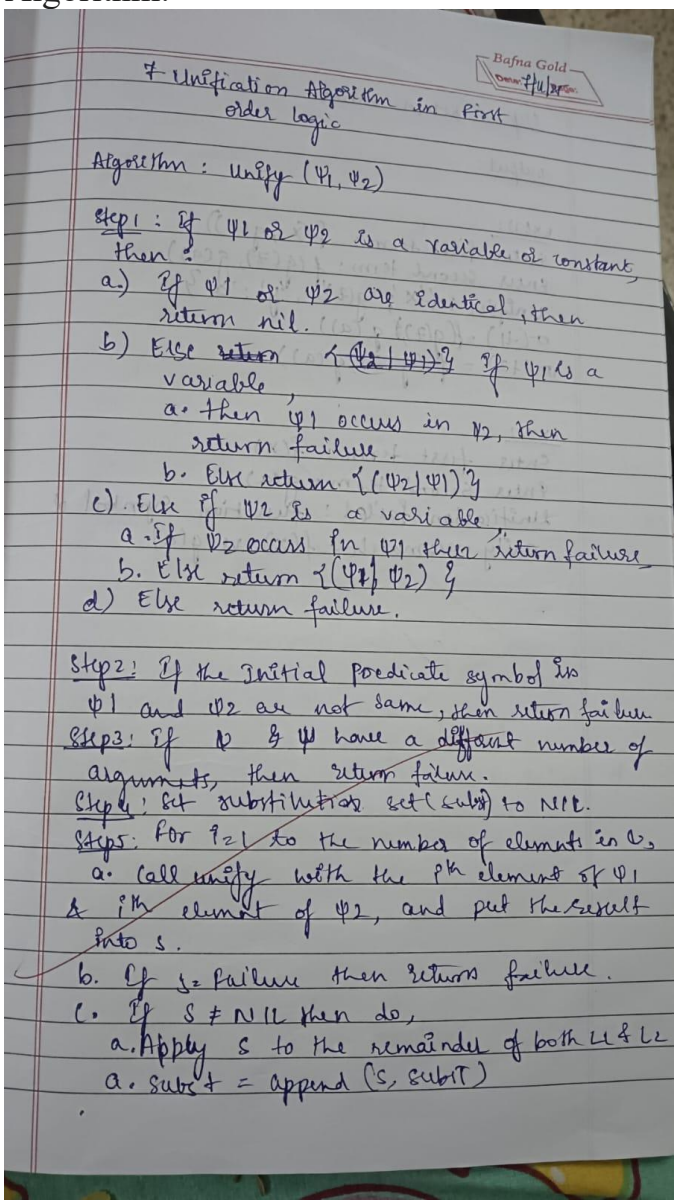
```
import itertools

def pl_true(expr, model):
    if expr == 'True': return True
    if expr == 'False': return False
    if expr in model: return model[expr]
    if expr.startswith('not '): return not pl_true(expr[4:], model)
    if ' and ' in expr:
        a, b = expr.split(' and ', 1)
        return pl_true(a, model) and pl_true(b, model)
    if ' or ' in expr:
```

```
        a, b = expr.split(' or ', 1)
        return pl_true(a, model) or pl_true(b, model)
    if '=>' in expr:
        a, b = expr.split('=>', 1)
        return (not pl_true(a, model)) or pl_true(b, model)
    if '<=>' in expr:
        a, b = expr.split('<=>', 1)
        return pl_true(a, model) == pl_true(b, model)
    return False

def tt_check_all(kb, alpha, symbols, model):
    if not symbols:
        if all(pl_true(s, model) for s in kb):
            return pl_true(alpha, model)
        else:
            return True
    rest = symbols[1:]
    p = symbols[0]
    model_true = model.copy()
    model_true[p] = True
    model_false = model.copy()
    model_false[p] = False
    return tt_check_all(kb, alpha, rest, model_true) and tt_check_all(kb, alpha, rest, model_false)

def tt_entails(kb, alpha):
    symbols = sorted({s for sentence in kb + [alpha] for s in sentence.replace('not
','').replace('(','').replace(')','').replace('and','').replace('or','').replace('=>','').replace('<=>','').split() if s
not in ['and','or','=>','<=>','not','True','False']})
    return tt_check_all(kb, alpha, symbols, {})

kb = input("Enter knowledge base sentences separated by commas: ").split(',')
alpha = input("Enter query sentence: ")
print("Entailment:", tt_entails([s.strip() for s in kb], alpha.strip()))
```

```
Evaluating Query: R

Truth Table Evaluation:

P | Q | R | KB | Query | KB ⇒ Query
------------------------------------------------------------------------
T | T | T | F  | T     | T
T | T | F | F  | F     | T
T | F | T | T  | T     | T
T | F | F | F  | F     | T
F | T | T | F  | T     | T
F | T | F | F  | F     | T
F | F | T | T  | T     | T
F | F | F | F  | F     | T

Result:
The Knowledge Base entails the Query (KB ⊨ Query)

================================================


Evaluating Query: R -> P

Truth Table Evaluation:

P | Q | R | KB | Query | KB ⇒ Query
------------------------------------------------------------------------
T | T | T | F  | T     | T
T | T | F | F  | T     | T
T | F | T | T  | T     | T
T | F | F | F  | T     | T
F | T | T | F  | F     | T
F | T | F | F  | T     | T
F | F | T | T  | F     | F
F | F | F | F  | T     | T

Result:
The Knowledge Base does NOT entail the Query (KB ⊭ Query)

================================================
```

## 6. Implement unification in first order logic

Algorithm:



7 Unification Algorithm in First order logic

Algorithm : Unify ($\psi_1, \psi_2$)

Step 1 : If $\psi_1$ or $\psi_2$ is a variable or constant, then :
a.) If $\psi_1$ or $\psi_2$ are identical, then return nil.
b) Else ~~return {($\psi_2$/$\psi_1$)}~~ If $\psi_1$ is a variable,
a. then $\psi_1$ occurs in $\psi_2$, then return failure
b. Else return {($\psi_2$/$\psi_1$)}
c) Else if $\psi_2$ is a variable,
a. If $\psi_2$ occurs in $\psi_1$ then return failure
b. Else return {($\psi_1$/$\psi_2$)}
d) Else return failure.

Step 2 : If the initial predicate symbol in $\psi_1$ and $\psi_2$ are not same, then return failure
Step 3 : If $\psi$ & $\psi$ have a ~~different~~ number of arguments, then return failure.
Step 4 : Set substitution set (subst) to NIL.
Step 5 : For $i = 1$ to the number of elements in $\psi_1$,
a. Call unify with the $i^{th}$ element of $\psi_1$ & $i^{th}$ element of $\psi_2$, and put the result into S.
b. If S = failure then return failure.
c. If $S \neq$ NIL then do,
a. Apply S to the remainder of both L1 & L2
a. subst = append (S, subst)

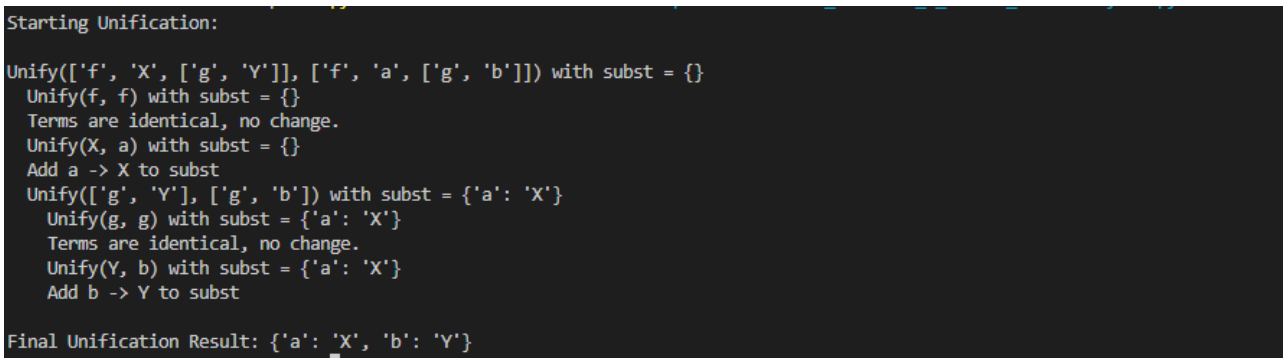
Code:

```
from dataclasses import dataclass
from typing import Tuple, Dict, Union, List, Optional

@dataclass(frozen=True)
class Var:
    name: str
    def __repr__(self):
        return self.name
```

```
@dataclass(frozen=True)
class Func:
    name: str
    args: Tuple['Term', ...]
    def __repr__(self):
        if not self.args:
            return self.name
        return f"{self.name}({', '.join(map(str, self.args))})"

Term = Union[Var, Func]
Subst = Dict[Var, Term]

def is_variable(t: Term) -> bool:
    return isinstance(t, Var)

def apply_subst(subst: Subst, term: Term) -> Term:
    if is_variable(term):
        while term in subst:
            term = subst[term]
        return term
    return Func(term.name, tuple(apply_subst(subst, a) for a in term.args))

def occurs_in(var: Var, term: Term, subst: Subst) -> bool:
    term = apply_subst(subst, term)
    if term == var:
        return True
    if isinstance(term, Func):
        return any(occurs_in(var, a, subst) for a in term.args)
    return False

class UnificationFailure(Exception):
    pass

def unify_var(var: Var, x: Term, subst: Subst) -> Subst:
    x = apply_subst(subst, x)
    if var == x:
        return subst
    if occurs_in(var, x, subst):
        raise UnificationFailure(f"Occurs-check failed: {var} occurs in {x}")
    new_subst = {v: apply_subst({var: x}, t) for v, t in subst.items()}
    new_subst[var] = x
    return new_subst

def unify(t1: Term, t2: Term, subst: Optional[Subst] = None) -> Subst:
    if subst is None:
        subst = {}
```

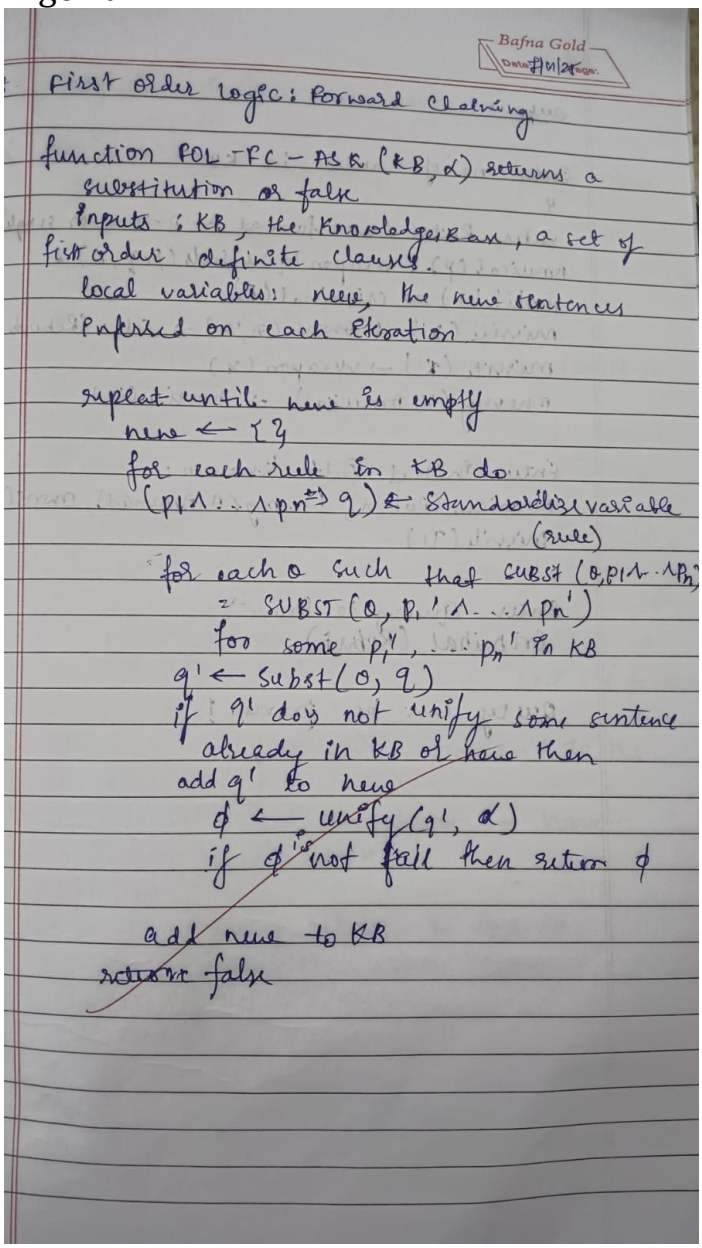
```
    t1 = apply_subst(subst, t1)
    t2 = apply_subst(subst, t2)
    if t1 == t2:
        return subst
    if is_variable(t1):
        return unify_var(t1, t2, subst)
    if is_variable(t2):
        return unify_var(t2, t1, subst)
    if isinstance(t1, Func) and isinstance(t2, Func):
        if t1.name != t2.name or len(t1.args) != len(t2.args):
            raise UnificationFailure(f"Function symbol or arity mismatch: {t1} vs {t2}")
        for a1, a2 in zip(t1.args, t2.args):
            subst = unify(a1, a2, subst)
        return subst
    raise UnificationFailure(f"Cannot unify {t1} and {t2}")

def parse_term(s: str) -> Term:
    s = s.strip()
    if '(' not in s:
        if s[0].islower():
            return Func(s, ())
        else:
            return Var(s)
    name, rest = s.split('(', 1)
    rest = rest[:-1]
    args = split_args(rest)
    return Func(name, tuple(parse_term(arg) for arg in args))

def split_args(s: str) -> List[str]:
    args, depth, current = [], 0, ''
    for c in s:
        if c == ',' and depth == 0:
            args.append(current.strip())
            current = ''
        else:
            if c == '(':
                depth += 1
            elif c == ')':
                depth -= 1
            current += c
    if current:
        args.append(current.strip())
    return args

if __name__ == "__main__":
    expr1 = input("Enter first term: ").strip()
    expr2 = input("Enter second term: ").strip()
```

```
    try:
        t1 = parse_term(expr1)
        t2 = parse_term(expr2)
        sigma = unify(t1, t2)
        print("Unifier:", {str(k): str(v) for k, v in sigma.items()})
        print("σ(t1) =", apply_subst(sigma, t1))
        print("σ(t2) =", apply_subst(sigma, t2))
    except UnificationFailure as e:
        print("Unification failed:", e)
    except Exception as e:
        print("Error:", e)
```



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

Algorithm:




Code:

```
import re

def parse_predicate(pred):
    pred = pred.strip()
    if "(" not in pred:
        # Treat constant symbols or atomic facts with no arguments
```

```
        return pred, []
    name, args = pred.split("(", 1)
    args = args.strip(")").split(",")
    return name.strip(), [a.strip() for a in args]

def is_variable(x):
    return x[0].islower()  # lowercase = variable

def unify(x, y, theta=None):
    if theta is None:
        theta = {}
    if x == y:
        return theta
    if is_variable(x):
        return unify_var(x, y, theta)
    if is_variable(y):
        return unify_var(y, x, theta)
    if isinstance(x, tuple) and isinstance(y, tuple):
        if x[0] != y[0] or len(x[1]) != len(y[1]):
            return None
        for a, b in zip(x[1], y[1]):
            theta = unify(a, b, theta)
            if theta is None:
                return None
        return theta
    return None

def unify_var(var, x, theta):
    if var in theta:
        return unify(theta[var], x, theta)
    if x in theta:
        return unify(var, theta[x], theta)
    if occurs_check(var, x, theta):
        return None
    theta[var] = x
    return theta

def occurs_check(var, x, theta):
    if var == x:
        return True
    if isinstance(x, tuple):
        return any(occurs_check(var, arg, theta) for arg in x[1])
    if x in theta:
        return occurs_check(var, theta[x], theta)
    return False

def substitute(pred, theta):
```

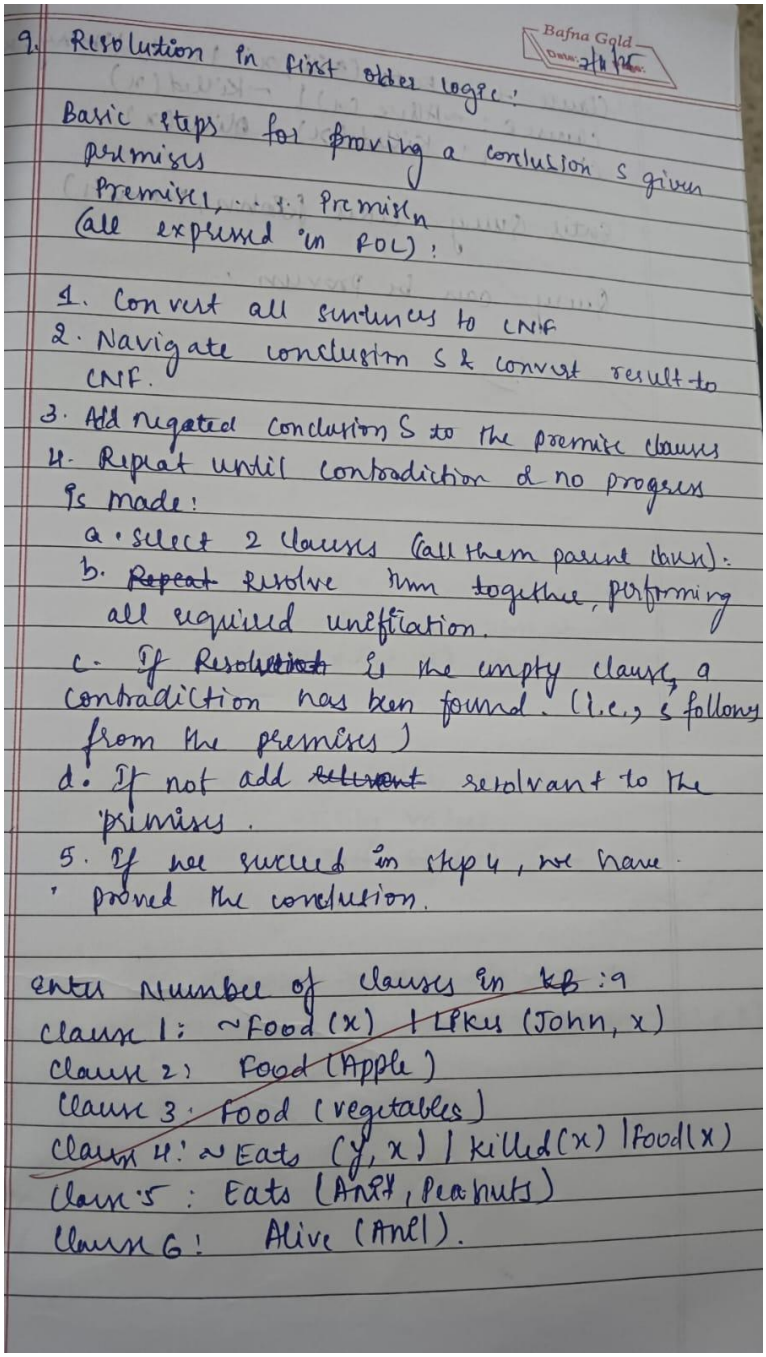
```
    name, args = pred
    new_args = [theta.get(a, a) for a in args]
    return (name, new_args)

def parse_clause(clause):
    if "=>" in clause:
        premises, conclusion = clause.split("=>")
        premises = [parse_predicate(p.strip()) for p in premises.split("&")]
        conclusion = parse_predicate(conclusion.strip())
    else:
        premises = []
        conclusion = parse_predicate(clause.strip())
    return premises, conclusion

def forward_chaining(kb, facts, query):
    query = parse_predicate(query)
    known_facts = [parse_predicate(f) for f in facts]
    added = True
    while added:
        added = False
        for premises, conclusion in kb:
            possible_subs = [{}]
            for prem in premises:
                new_subs = []
                for f in known_facts:
                    theta = unify(prem, f)
                    if theta is not None:
                        for ps in possible_subs:
                            merged = {**ps, **theta}
                            new_subs.append(merged)
                possible_subs = new_subs
            for s in possible_subs:
                new_fact = substitute(conclusion, s)
                if new_fact not in known_facts:
                    known_facts.append(new_fact)
                    added = True
                    if unify(new_fact, query) is not None:
                        return True
    return False

n = int(input("Enter number of rules in Knowledge Base:\n"))
kb = []
print("\nEnter rules (use format: A & B => C or single fact):")
for _ in range(n):
    clause = input().strip()
    kb.append(parse_clause(clause))
```

```
# Fix: Use re.findall to correctly parse facts that might contain commas in arguments
facts_input_string = input("\nEnter initial known facts (comma separated):\n").strip()
facts = re.findall(r'[A-Za-z_]+\(.*?\)', facts_input_string)
query = input("\nEnter query to prove:\n").strip()

result = forward_chaining(kb, facts, query)

print("\n----------------------------")
print("Knowledge Base:")
for p, c in kb:
    print(f"{' & '.join([f'{x[0]}({','.join(x[1])})' for x in p])} => {c[0]}({','.join(c[1])})" if p else
f"{c[0]}({','.join(c[1])})")
print("\nInitial Facts:", facts)
print("Query:", query)
print("----------------------------")
print("Query can be proven!" if result else "Query cannot be proven.")
```

```
--- Forward Chaining (FOL-FC-ASK) ---

Initial known facts:
   Missile(T1)
   Owns(A, T1)
   Enemy(A, America)
   American(Robert)

Inferred: Weapon(T1)
Inferred: Hostile(A)
Inferred: Sells(Robert, T1, A)
Inferred: Criminal(Robert)

 Query satisfied: Criminal(Robert)
```

## 8. Resolution in First order logic:

Algorithm:




Code:

```
import re

def standardize_variables(clause, index):
    vars_in_clause = re.findall(r'[a-z]\w*', clause)
```

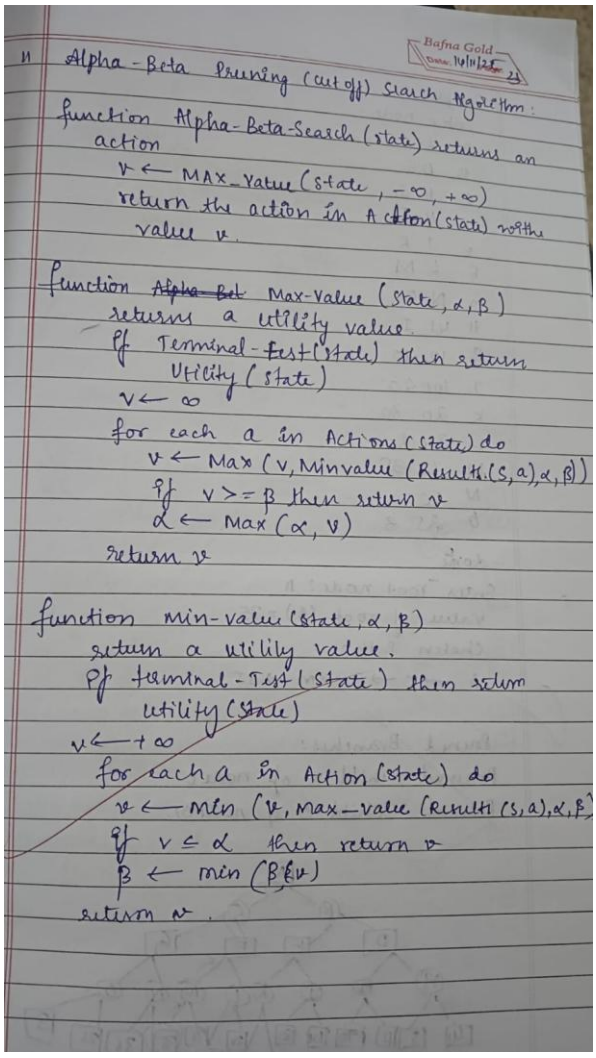
```
    for v in vars_in_clause:
        clause = re.sub(r'\b' + v + r'\b', v + str(index), clause)
    return clause

def negate_literal(literal):
    if literal.startswith('~'):
        return literal[1:]
    else:
        return '~' + literal

def parse_clause(clause):
    return [lit.strip() for lit in clause.split('|')]

def unify(x, y, subs):
    if subs is None:
        return None
    elif x == y:
        return subs
    elif isinstance(x, str) and x[0].islower():
        return unify_var(x, y, subs)
    elif isinstance(y, str) and y[0].islower():
        return unify_var(y, x, subs)
    elif '(' in x and '(' in y:
        fx, argsx = x.split('(', 1)
        fy, argsy = y.split('(', 1)
        if fx != fy:
            return None
        argsx = argsx[:-1].split(',')
        argsy = argsy[:-1].split(',')
        for a, b in zip(argsx, argsy):
            subs = unify(a.strip(), b.strip(), subs)
        return subs
    else:
        return None

def unify_var(var, x, subs):
    if var in subs:
        return unify(subs[var], x, subs)
    elif x in subs:
        return unify(var, subs[x], subs)
    else:
        subs[var] = x
        return subs

def apply_subs(literal, subs):
    for var, val in subs.items():
        literal = re.sub(r'\b' + var + r'\b', val, literal)
```

```
    return literal

def resolve(ci, cj):
    for di in ci:
        for dj in cj:
            if di == negate_literal(dj):
                new_clause = list(set(ci + cj))
                new_clause.remove(di)
                new_clause.remove(dj)
                return new_clause
    return None

n = int(input("Enter number of clauses in KB: "))
kb = []
for i in range(n):
    kb.append(standardize_variables(input(f"Clause {i+1}: ").strip(), i))

query = input("Enter query: ").strip()
negated_query = negate_literal(query)
kb.append(negated_query)

new = set()
resolved = False

while True:
    pairs = [(kb[i], kb[j]) for i in range(len(kb)) for j in range(i + 1, len(kb))]
    for (ci, cj) in pairs:
        resolvent = resolve(parse_clause(ci), parse_clause(cj))
        if resolvent == []:
            resolved = True
            break
        if resolvent:
            new.add(' | '.join(resolvent))
    if resolved:
        print("\nQuery can be proven by resolution.")
        break
    if new.issubset(set(kb)):
        print("\nQuery can be proven.")
        break
    for c in new:
        if c not in kb:
            kb.append(c)
```

```
FIRST-ORDER LOGIC RESOLUTION PROOF
Proving: John likes peanuts
================================================================================

================================================================================
FIRST-ORDER LOGIC REPRESENTATIONS:
================================================================================
1. John likes all kinds of food
   ForAll x (Food(x) -> Likes(John, x))
2. Apple is food
   Food(Apple)
3. Vegetables are food
   Food(Vegetables)
4. Anything anyone eats and is not killed is food
   ForAll y ForAll z ((Eats(y,z) AND ~Killed(y)) -> Food(z))
5. Anil eats peanuts
   Eats(Anil, Peanuts)
6. Anil is alive
   Alive(Anil)
7. Harry eats everything that Anil eats
   ForAll x (Eats(Anil,x) -> Eats(Harry,x))
8. Anyone who is alive is not killed
   ForAll x (Alive(x) -> ~Killed(x))
9. Anyone who is not killed is alive
   ForAll x (~Killed(x) -> Alive(x))

Query: Likes(John, Peanuts)
Negated Query (for refutation): ~Likes(John, Peanuts)

================================================================================
CNF CLAUSES (after conversion):
================================================================================
C1: {Likes(John, x) V ~Food(x)}
C2: {Food(Apple)}
C3: {Food(Vegetables)}
C4: {Food(z) V Killed(y) V ~Eats(y, z)}
C5: {Eats(Anil, Peanuts)}
C6: {Alive(Anil)}
C7: {Eats(Harry, x) V ~Eats(Anil, x)}
C8: {~Alive(x) V ~Killed(x)}
C9: {Alive(x) V Killed(x)}
C10: {~Likes(John, Peanuts)}

================================================================================
RESOLUTION PROOF:
================================================================================
```

## 9. Implement Alpha-Beta Pruning:

Algorithm:

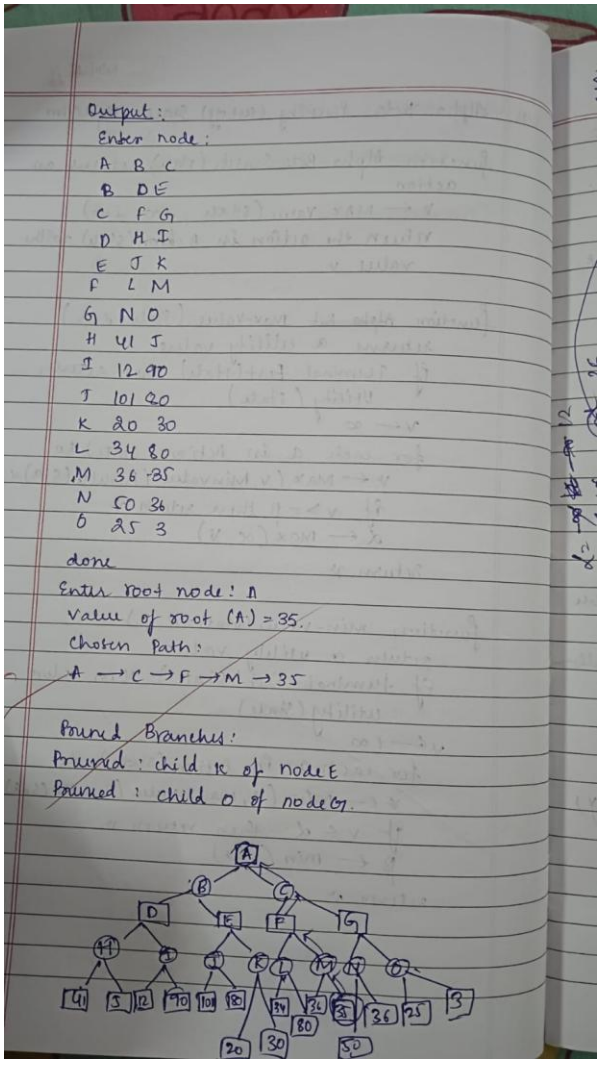


Alpha - Beta Pruning (cut off) Search Algorithm:

function Alpha-Beta-Search (state) returns an action
v ← MAX_Value (state, −∞, +∞)
return the action in Action(state) with the value v.

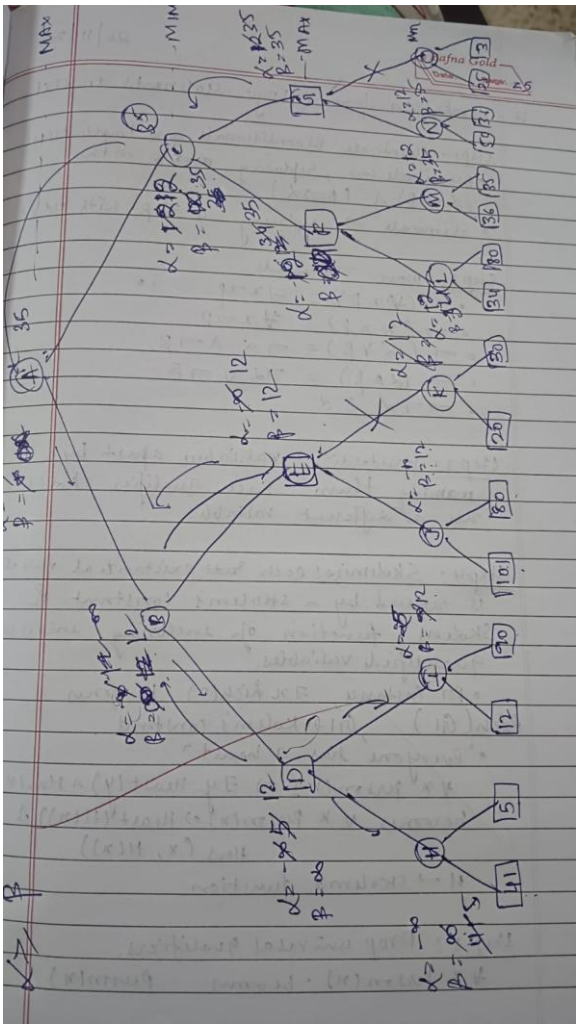
function ~~Alpha-Beta~~ Max-Value (State, α, β)
returns a utility value
if Terminal-~~Test~~(State) then return Utility (State)
v ← ∞
for each a in Actions (State) do
v ← Max (v, Minvalue (Result(S,a), α, β))
if v >= β then return v
α ← Max (α, v)
return v

function Min-value (State, α, β)
return a utility value.
if terminal-Test (State) then return utility (State)
v ← +∞
for each a in Action (state) do
v ← min (v, Max-value (Result(S,a), α, β))
if v ≤ α then return v
β ← min (β, v)
return v.



Output:
Enter node:
A B C
B D E
C F G
D H I
E J K
F L M
G N O
H 41 5
I 12 90
J 101 40
K 20 30
L 34 80
M 36 35
N 50 36
O 25 3
done
Enter root node: A
Value of root (A) = 35.
Chosen Path:
A → C → F → M → 35

Pruned Branches:
Pruned : child 2 of node E
Pruned : child 0 of node G.



Code:

```
import math

pruned = []   # store pruned branches

def alphabeta(node, isMax, alpha, beta, tree):
    global pruned

    # Leaf node (integer)
    if isinstance(node, int):
        return node

    children = tree[node]

    if isMax:      # MAX NODE
        value = -math.inf
        for child in children:
```

```
            if value >= beta:
                pruned.append((node, child))
                break
            val = alphabeta(child, False, alpha, beta, tree)
            value = max(value, val)
            alpha = max(alpha, value)
        return value

    else:          # MIN NODE
        value = math.inf
        for child in children:
            if value <= alpha:
                pruned.append((node, child))
                break
            val = alphabeta(child, True, alpha, beta, tree)
            value = min(value, val)
            beta = min(beta, value)
        return value

def compute_value(node, isMax, tree):
    """Compute subtree value without pruning."""
    if isinstance(node, int):
        return node
    if isMax:
        return max(compute_value(c, False, tree) for c in tree[node])
    else:
        return min(compute_value(c, True, tree) for c in tree[node])


def find_path(node, target, isMax, tree):
    if isinstance(node, int):
        return [node]

    children = tree[node]

    if isMax:
        for c in children:
            if compute_value(c, False, tree) == target:
                return [node] + find_path(c, target, False, tree)
    else:
        for c in children:
            if compute_value(c, True, tree) == target:
                return [node] + find_path(c, target, True, tree)

    return [node]

print("\nEnter your game tree.")
```

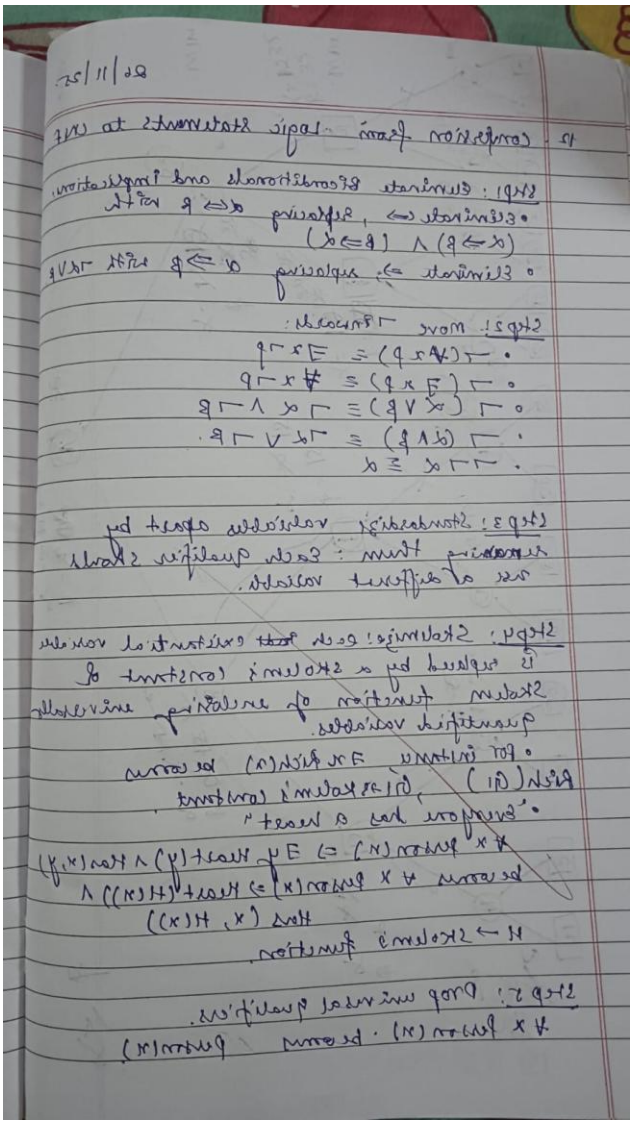
```
print("Format for each line:")
print("  Parent Child1 Child2 Child3 ...")
print("Use integers for leaves.")
print("Example:")
print("  A B C")
print("  B D E")
print("  D 41 5")
print("Type 'done' to stop\n")

tree = {}

while True:
    line = input("Enter node: ")
    if line.lower().strip() == "done":
        break

    parts = line.split()
    parent = parts[0]
    children = []

    for ch in parts[1:]:
        # convert leaf to int
        try:
            children.append(int(ch))
        except:
            children.append(ch)

    tree[parent] = children

root = input("\nEnter root node: ").strip()

pruned = []
root_value = alphabeta(root, True, -math.inf, math.inf, tree)
path = find_path(root, root_value, True, tree)

print("\n==============================")
print("      ALPHA–BETA RESULT")
print("==============================\n")

print(f"Value of Root ({root}) = {root_value}\n")

print("Chosen Path:")
print(" → ".join(map(str, path)), "\n")

print("Pruned Branches:")
if pruned:
    for p in pruned:
```

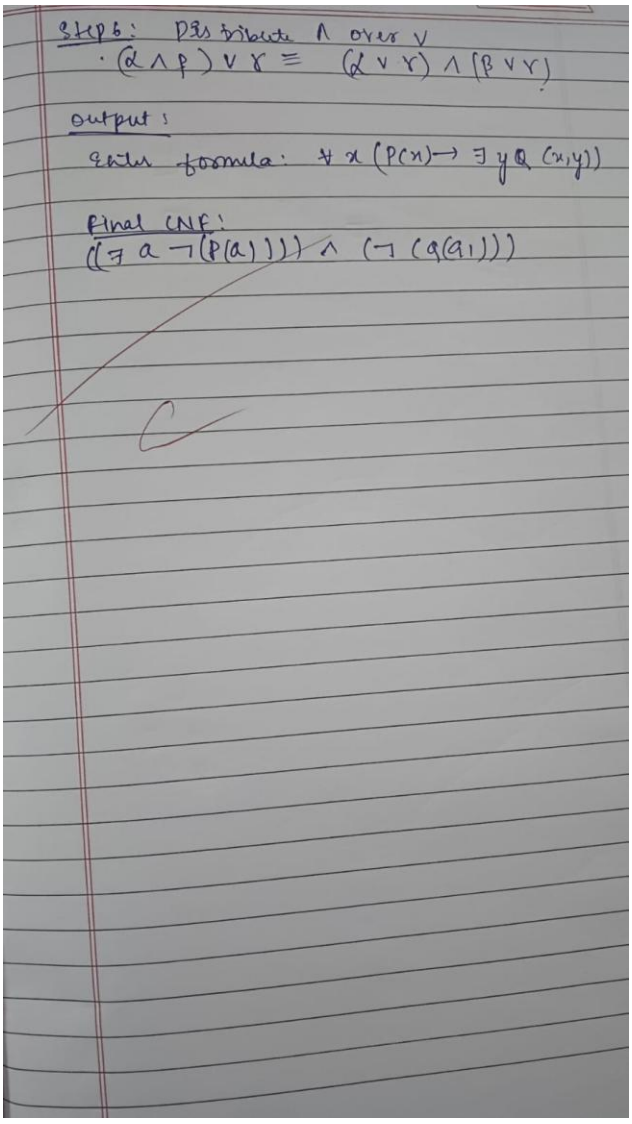
```
        print(f"  Pruned: child {p[1]} of node {p[0]}")
else:
    print("  No branches pruned.")
```

```
Game Tree Structure:

                A (MAX)
               /       \
          B (MIN)       C (MIN)
          /     \       /     \
      D (MAX)  E (MAX)  F (MAX)  G (MAX)
      /   \     /   \     /   \     /   \
    10    9   14   18   5    4   50    3

================================================================================

Starting Alpha-Beta Pruning...

================================================================================

Exploring MAX node A (depth=0), alpha=-inf, beta=inf
--> Exploring child B of A
  Exploring MIN node B (depth=1), alpha=-inf, beta=inf
  --> Exploring child D of B
    Exploring MAX node D (depth=2), alpha=-inf, beta=inf
    --> Exploring child L1 of D
      Reached leaf L1 with value 10
    Updated MAX node D: value=10, alpha=10, beta=inf
    --> Exploring child L2 of D
      Reached leaf L2 with value 9
    Updated MAX node D: value=10, alpha=10, beta=inf
  Updated MIN node B: value=10, alpha=-inf, beta=10
  --> Exploring child E of B
    Exploring MAX node E (depth=2), alpha=-inf, beta=10
    --> Exploring child L3 of E
      Reached leaf L3 with value 14
    Updated MAX node E: value=14, alpha=14, beta=10
    !!! PRUNING at MAX node E (beta=10 <= alpha=14)
  Updated MIN node B: value=10, alpha=-inf, beta=10
Updated MAX node A: value=10, alpha=10, beta=inf
--> Exploring child C of A
  Exploring MIN node C (depth=1), alpha=10, beta=inf
  --> Exploring child F of C
    Exploring MAX node F (depth=2), alpha=10, beta=inf
    --> Exploring child L5 of F
      Reached leaf L5 with value 5
    Updated MAX node F: value=5, alpha=10, beta=inf
    --> Exploring child L6 of F
      Reached leaf L6 with value 4
    Updated MAX node F: value=5, alpha=10, beta=inf
  Updated MIN node C: value=5, alpha=10, beta=5
```

## 10.Conversion of logic statement to CNF

Algorithm:






Code:

```
import re
import itertools

def remove_implications(expr):
    """Eliminate ↔ and → using logical equivalences."""
    # Eliminate ↔ : (A ↔ B) = (A → B) ∧ (B → A)
    while "↔" in expr:
        expr = re.sub(r"(.*?)(↔)(.*)", r"(\1 → \3) ∧ (\3 → \1)", expr)
```

```
    # Eliminate → : (A → B) = ¬A ∨ B
    expr = re.sub(r"(.*?)→(.*)", r"(¬(\1)) ∨ (\2)", expr)
    return expr


def push_negation(expr):
    """Move negations inward using De Morgan rules & quantifier rules."""

    # Double negation
    expr = re.sub(r"¬¬", "", expr)

    # ¬(A ∧ B) = (¬A ∨ ¬B)
    expr = re.sub(r"¬\((.*?) ∧ (.*?)\)", r"(¬\1) ∨ (¬\2)", expr)

    # ¬(A ∨ B) = (¬A ∧ ¬B)
    expr = re.sub(r"¬\((.*?) ∨ (.*?)\)", r"(¬\1) ∧ (¬\2)", expr)

    # Quantifiers:
    expr = re.sub(r"¬∀([a-z])", r"∃\1¬", expr)
    expr = re.sub(r"¬∃([a-z])", r"∀\1¬", expr)

    return expr


def standardize_variables(expr):
    """Rename variables so all quantifiers use different names."""
    variables = "abcdefghijklmnopqrstuvwxyz"
    mapping = {}
    new_expr = ""

    counter = 0
    for ch in expr:
        if ch in variables:
            if ch not in mapping:
                mapping[ch] = variables[counter]
                counter += 1
            new_expr += mapping[ch]
        else:
            new_expr += ch
    return new_expr


skolem_counter = 1

def skolemize(expr):
    """Remove existential quantifiers replacing them with Skolem constants/functions."""
    global skolem_counter
```

```
    # Replace ∃x P(x) with P(G1)
    pattern = r"∃([a-z])\s*(\w+)\((.*?)\)"
    match = re.search(pattern, expr)

    while match:
        var = match.group(1)
        pred = match.group(2)
        args = match.group(3)

        new_const = f"G{skolem_counter}"
        skolem_counter += 1

        expr = re.sub(pattern, f"{pred}({new_const})", expr)
        match = re.search(pattern, expr)

    return expr


def drop_universal(expr):
    """Remove ∀ quantifiers (they are implicit in CNF)."""
    expr = expr.replace("∀", "")
    return expr


def distribute_and_over_or(expr):
    """Distribute ∧ over ∨ (very simplified CNF converter)."""
    # This function is intentionally simple and works for standard textbook examples.

    # (A ∧ B) ∨ C -> (A ∨ C) ∧ (B ∨ C)
    pattern = r"\((.*?) ∧ (.*?)\) ∨ (.*)"
    if re.search(pattern, expr):
        A, B, C = re.findall(pattern, expr)[0]
        return f"({A} ∨ {C}) ∧ ({B} ∨ {C})"

    return expr

def convert_to_cnf(expr):
    print("\n---- Step 1: Remove ↔ and → ----")
    expr = remove_implications(expr)
    print(expr)

    print("\n---- Step 2: Push ¬ inward ----")
    expr = push_negation(expr)
    print(expr)

    print("\n---- Step 3: Standardize variables ----")
```

```
    expr = standardize_variables(expr)
    print(expr)

    print("\n---- Step 4: Skolemize ----")
    expr = skolemize(expr)
    print(expr)

    print("\n---- Step 5: Drop universal quantifier ----")
    expr = drop_universal(expr)
    print(expr)

    print("\n---- Step 6: Distribute ∧ over ∨ ----")
    expr = distribute_and_over_or(expr)
    print(expr)

    return expr

print("Enter a First-Order Logic Statement to Convert to CNF:")
print("Example input:")
print("   ∀x (P(x) → ∃y Q(x,y))")
print("   (A ↔ B)")
print("------------------------------------------------------------")

expr = input("\nEnter formula: ")

print("\n============================================================")
print("                CNF CONVERSION STEPS")
print("============================================================")

cnf = convert_to_cnf(expr)

print("\n============================================================")
print("Final CNF:")
print(cnf)
print("============================================================")
```

# Output:

```
Enter a First-Order Logic Statement to Convert to CNF:
Example input:
   ∀x (P(x) → ∃y Q(x,y))
   (A ↔ B)
----------------------------------------------------------------

Enter formula: ∀x (P(x) → ∃y Q(x,y))

================================================================
                    CNF CONVERSION STEPS
================================================================

---- Step 1: Remove ↔ and → ----
(¬(∀x (P(x) )) ∨ ( ∃y Q(x,y)))

---- Step 2: Push ¬ inward ----
((∃x¬ (P(x) ))) ∧ (¬( ∃y Q(x,y)))

---- Step 3: Standardize variables ----
((∃a¬ (P(a) ))) ∧ (¬( ∃b Q(a,b)))

---- Step 4: Skolemize ----
((∃a¬ (P(a) ))) ∧ (¬( Q(G1)))

---- Step 5: Drop universal quantifier ----
((∃a¬ (P(a) ))) ∧ (¬( Q(G1)))

---- Step 6: Distribute ∧ over ∨ ----
((∃a¬ (P(a) ))) ∧ (¬( Q(G1)))

================================================================
Final CNF:
((∃a¬ (P(a) ))) ∧ (¬( Q(G1)))
================================================================
```