VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Aryan Gowda (1BM23CS054)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

Sep-2024 to Jan-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Aryan Gowda (1BM23CS054),** 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.

Sandhya A Kulkarni Associate Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Table Of Contents

Infosys Springboard Certificates

Infosys
Navigate your next

CERTIFICATE OF ACHIEVEMENT

The certificate is awarded to

Aryan Gowda

for successfully completing

Applied Generative AI Certification

on November 25, 2025

Infosys | Springboard

Congratulations! You make us proud!

Issued on: Tuesday, November 25, 2025
To verify, scan the QR code at https://verify.onwingspan.com

Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Infosys
Navigate your next

COURSE COMPLETION CERTIFICATE

The certificate is awarded to

Aryan Gowda

for successfully completing the course

OpenAI Generative Pre-trained Transformer 3 (GPT-3) for developers

on November 25, 2025

Infosys | Springboard

Congratulations! You make us proud!

Issued on: Wednesday, November 26, 2025
To verify, scan the QR code at https://verify.onwingspan.com

Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Infosys
Navigate your next

COURSE COMPLETION CERTIFICATE

The certificate is awarded to

Aryan Gowda

for successfully completing the course

Introduction to OpenAI GPT Models

on November 25, 2025

Infosys | Springboard

Congratulations! You make us proud!

Issued on: Wednesday, November 26, 2025
To verify, scan the QR code at https://verify.onwingspan.com

Satheesha B.N.
Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Infosys®
Navigate your next

COURSE COMPLETION CERTIFICATE

The certificate is awarded to

Aryan Gowda

for successfully completing the course

Prompt Engineering

on November 25, 2025

Infosys | Springboard

Congratulations! You make us proud!

Issued on: Wednesday, November 26, 2025
To verify, scan the QR code at https://verify.onwingspan.com

Satheesha B.N.
Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Infosys®
Navigate your next

COURSE COMPLETION CERTIFICATE

The certificate is awarded to

Aryan Gowda

for successfully completing the course

AI-first Software Engineering

on November 25, 2025

Infosys | Springboard

Congratulations! You make us proud!

Issued on: Wednesday, November 26, 2025
To verify, scan the QR code at https://verify.onwingspan.com

Satheesha B.N.
Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Tic Tac Toe

Tic-Tac Toe

board [9] = {-1}
player_turn=0
minimax (board, player_turn):
    while (i!=9):
        player_turn ← ~player_turn
        ~~list~~ I
        c_board = copy (board) // copying board
        c-board [i] = player turn
        board_c, n = minimax (c-board, playerturn)
        all_outcomes. extend ([board_c, n])
        ~~for~~
    ~~f~~ if all_outcomes == NULL:
        n= check-outcome (board)
    else:
        for b, n in all_outcomes:-
            if n > max&:
                n= max
                b_max = b
        return b, n

def check-outcome (board):
    ~~for (int i=0;~~
    for i in range 3:
        if board [i] == board [i+3] and board[i]== board[i+6]
            ~~return~~
            if board[i] == 1:
                return 1
            ~~else~~ else return -1
        if board ~~ret~~ [i] == board[i+1] and board[i]== b [i+2]
            if board (i)==1:
            return 1 else return -1;

```
if
if   board[0] == board[4]  and board[4]==board[8]
     if  board[0] == 1:
           return 1
     else
           return -1:
if   board[2] == board[4]  and  board[4]== board[6]
     if  board[0] == 1:
           return 1
     else
           return -1:

return   0
while(game !finished){
   player_input = int(input("Enter square"))
   board [player_input] = 0
   board =  minimax(board, 10)
   game_finished = input ("Enter finished"))
```

Code:

```
 import math

class TicTacToe:
def __init__(self):
self.board = []

    def create_board(self):
        self.board = [['-' for _ in range(3)] for _ in range(3)]

    def fix_spot(self, position, player):
        row = position // 3         col =
position % 3
self.board[row][col] = player

    def is_spot_empty(self, position):
        row = position // 3         col =
position % 3         return
self.board[row][col] == '-'

    def is_player_win(self, player):
        n = len(self.board)

        # Check rows and columns
for i in range(n):
            if all(self.board[i][j] == player for j in range(n)) or \
all(self.board[j][i] == player for j in range(n)):
                return True

        # Check diagonals         if all(self.board[i][i] == player
for i in range(n)) or \          all(self.board[i][n - 1 - i] ==
player for i in range(n)):
            return True

        return False

    def is_board_filled(self):
for row in self.board:
for item in row:
if item == '-':
return False         return
True
```

```
    def show_board(self):
print("\nCurrent Board:")
for row in self.board:
print(" ".join(row))        print()

    def show_positions(self):
print("\nPosition Map:")        positions
= [str(i) for i in range(9)]        for i in
range(0, 9, 3):
            print(" ".join(positions[i:i + 3]))
print()

    def evaluate(self):        if
self.is_player_win('O'):
return 1        elif
self.is_player_win('X'):
            return -1
else:
return 0

    def minimax(self, depth, is_maximizing):
        score = self.evaluate()

        # Base conditions
if score == 1:            return
score        if score == -1:
return score        if
self.is_board_filled():
            return 0

        if is_maximizing:
best = -math.inf            for i in
range(9):                if
self.is_spot_empty(i):
self.fix_spot(i, 'O')
best = max(best,
self.minimax(depth + 1, False))
self.fix_spot(i, '-')  # Undo move
return best        else:
            best = math.inf            for i in range(9):                if
self.is_spot_empty(i):                    self.fix_spot(i, 'X')
```

```
best = min(best, self.minimax(depth + 1, True))
self.fix_spot(i, '-')  # Undo move             return best

    def find_best_move(self):
best_val = -math.inf         best_move
= -1

        for i in range(9):             if
self.is_spot_empty(i):
self.fix_spot(i, 'O')                 move_val =
self.minimax(0, False)
self.fix_spot(i, '-')                 if move_val >
best_val:                    best_move = i
best_val = move_val

        return best_move

    def start(self):
self.create_board()
player = 'X'         computer
= 'O'

        print("Welcome to Tic Tac Toe!")
print("You are 'X' and the computer is 'O'.")
self.show_positions()

        while True:
            self.show_board()
try:
                position = int(input("Enter position (0–8): "))             except
ValueError:
                print("Invalid input! Enter a number between 0 and 8.")
continue

            if position < 0 or position > 8:
                print("Invalid position! Enter between 0 and 8.")
continue

            if not self.is_spot_empty(position):
                print("That spot is already taken! Choose another.")
continue

            self.fix_spot(position, player)
```

```
            if self.is_player_win(player):
                self.show_board()
print("You win!")                break

            if self.is_board_filled():
self.show_board()
print("It's a draw!")                break

            print("Computer is making its move...")
best_move = self.find_best_move()
self.fix_spot(best_move, computer)
print(f"Computer chose position {best_move}")

            if self.is_player_win(computer):
                self.show_board()
print("Computer wins!")
break            if
self.is_board_filled():
self.show_board()
print("It's a draw!")
break
# Start the game tic_tac_toe
= TicTacToe()
tic_tac_toe.start()
```

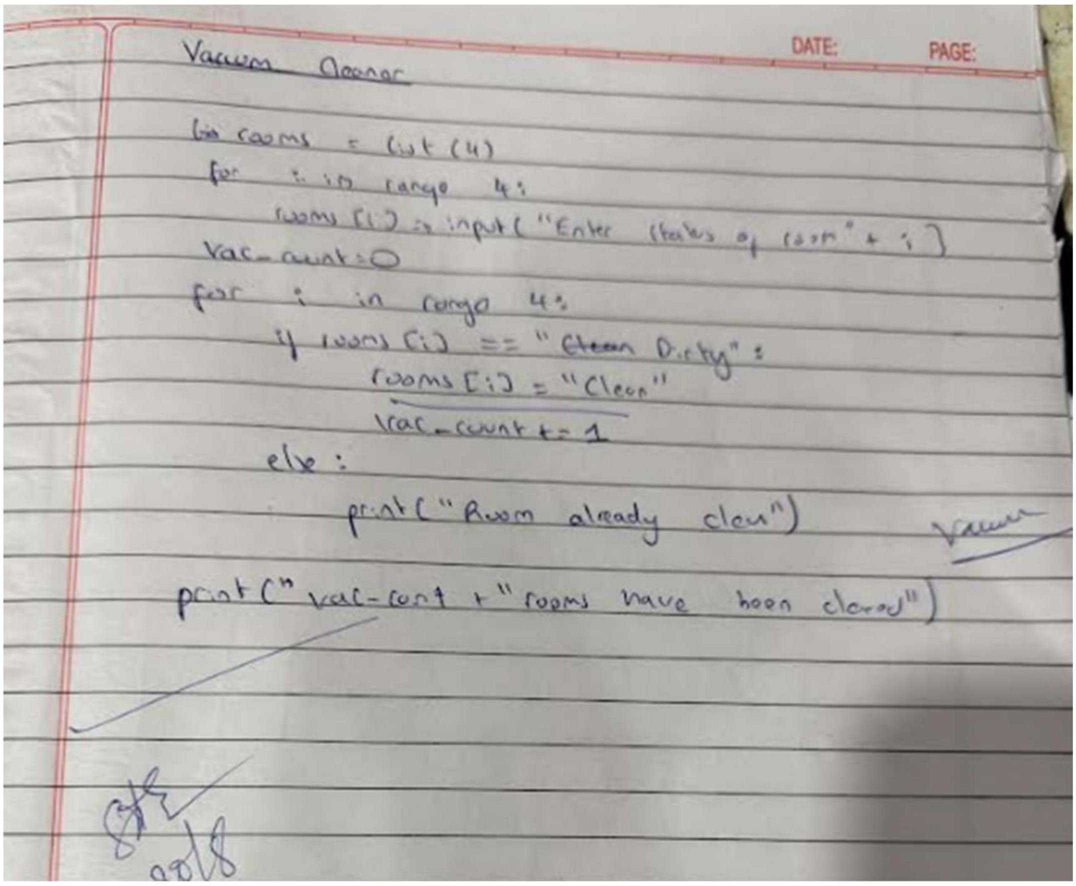
Output:

```
Welcome to Tic Tac Toe!
You are 'X' and the computer is 'O'.

Position Map:
0 1 2
3 4 5
6 7 8


Current Board:
- - -
- - -
- - -

Enter position (0-8): 4
Computer is making its move...
Computer chose position 0

Current Board:
O - -
- X -
- - -

Enter position (0-8): 5
Computer is making its move...
Computer chose position 3

Current Board:
O - -
O X X
- - -

Enter position (0-8): 6
Computer is making its move...
Computer chose position 2

Current Board:
O - O
O X X
X - -

Enter position (0-8): 1
Computer is making its move...
Computer chose position 7
```

```
Current Board:
O X O
O X X
X O -

Enter position (0-8): 8

Current Board:
O X O
O X X
X O X

It's a draw!
```

Vacuum Cleaner

Pseudocode:



Vacuum Cleaner

bin rooms = list (4)
for i in range 4:
rooms [i] = input ("Enter status of room" + i]
vac_count = 0
for i in range 4:
if rooms [i] == "Clean Dirty":
rooms [i] = "Clean"
vac_count += 1
else:
print ("Room already clean")

print (" vac_cont + " rooms have been cleaned")

Code:

```
def vacuum_world():
    goal_state = {'A': '0', 'B': '0'}      cost = 0      location_input =
input("Enter Location of Vacuum: ")      status_input = input("Enter
status of " + location_input + " : ")      status_input_complement =
input("Enter status of other room : ")

    print("Initial Location Condition {A : " + str(status_input_complement) +
", B : " + str(status_input) + " }")

    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  # cost for suck              print("Cost for
```

```
CLEANING A " + str(cost))
print("Location A has been Cleaned.")

            if status_input_complement == '1':
print("Location B is Dirty.")                 print("Moving
right to the Location B.")               cost += 1
print("COST for moving RIGHT " + str(cost))
goal_state['B'] = '0'                cost += 1
print("COST for SUCK " + str(cost))
print("Location B has been Cleaned.")            else:
                print("No action " + str(cost))
print("Location B is already clean.")

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

else:
        print("Vacuum is placed in Location B")
if status_input == '1':            print("Location B is
Dirty.")            goal_state['B'] = '0'            cost +=
1            print("COST for CLEANING " +
str(cost))            print("Location B has been
Cleaned.")

            if status_input_complement == '1':
print("Location A is Dirty.")                 print("Moving
LEFT to the Location A.")               cost += 1
print("COST for moving LEFT " + str(cost))
goal_state['A'] = '0'                cost += 1
print("COST for SUCK " + str(cost))
print("Location A has been Cleaned.")            else:
                print(cost)
                print("Location A is already clean.")
```

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

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


vacuum_world()
```

Output:

```
Python 3.11.9 (tags/v3.11.9:de54cf5, Apr  2 2024
Type "help", "copyright", "credits" or "license(

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

Iterative Deepening Depth First Search

8 Puzzle using IDDFS Pseudocode:

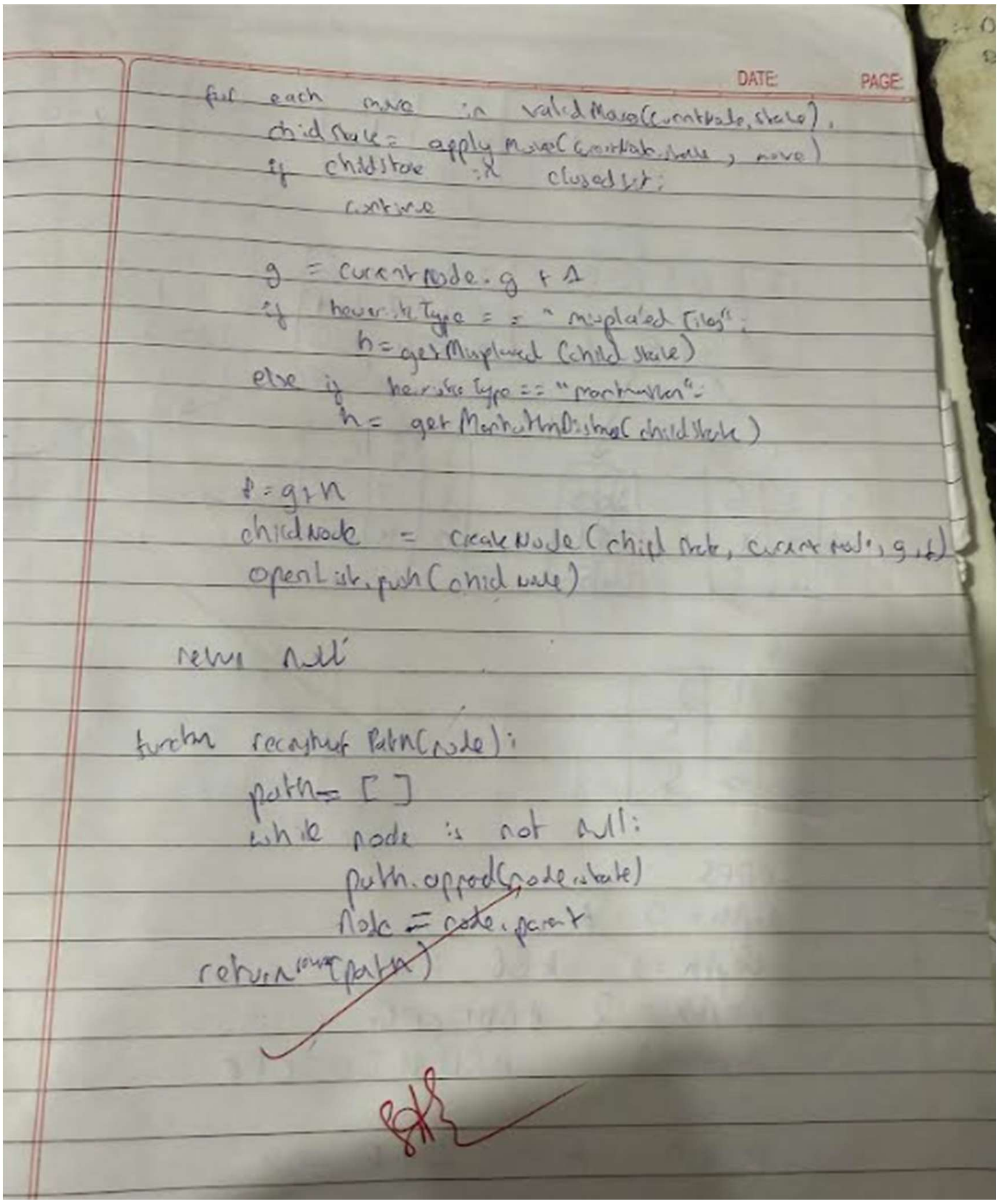
IDDFS

Pseudocode

```
function IDDFS(root) is
    for depth from 0 to ∞ do
        found, remaining ← DLS(root, depth)
        if found ≠ null then
            return found
        else if not remaining then
            return null

function DLS(node, depth) is
    if depth = 0 then
        if node is goal then
            return (node, true)
        else
            return (null, true)

    else if depth > 0 then
        any_remaining ← false
        foreach child of node do
            found, remaining ← DLS(child, depth-1)
            if found ≠ null then
                return (found, true)
            if remaining then
                any_remaining ← true
        return (null, any_remaining)
```

fore
Manhattan Distance
```
function getManhattanDistance(state):
    distance <- 0
    for each tile in state:
        if tile is not 0:
            target_position <- getTargetPosition(state)
            current_position <- getCurrentPosition(state, tile)
        distance += abs(target_position.row - current_position.row)
                  + abs(target_position.col - current_position.col)
return distance
```

Misplaced Tiles
```
function getMisplacedTiles(state):
    misplaced <- 0
    for each tile in state:
        if tile is not in targetposition:
            misplaced += 1
    return misplaced

function AstarSearch(startState, goalState, heuristicType):
    openList = priorityQueue()
    closedList = set()
    startNode = createNode(startState, null, 0, 0)
    openList.push(startNode)

    while openList is not empty:
        currentNode = openList.pop()
        if currentNode.state == goalState:
            return reconstructPath(currentNode)
        closedList.add(currentNode.state)
```


DATE: PAGE:

```
for each move in validMoves(currentNode.state):
    childState = applyMove(currentNode.state, move)
    if childState in closedList:
        continue

    g = currentNode.g + 1
    if heuristicType == "misplaced tiles":
        h = getMisplaced(childState)
    else if heuristicType == "manhattan":
        h = getManhattanDistance(childState)

    f = g + h
    childNode = createNode(childState, currentNode, g, f)
    openList.push(childNode)

return null

function reconstructPath(node):
    path = [ ]
    while node is not null:
        path.append(node.state)
        node = node.parent
    return reverse(path)
```

Code:

```
from copy import deepcopy

class Puzzle:     def __init__(self,
board, goal):
```

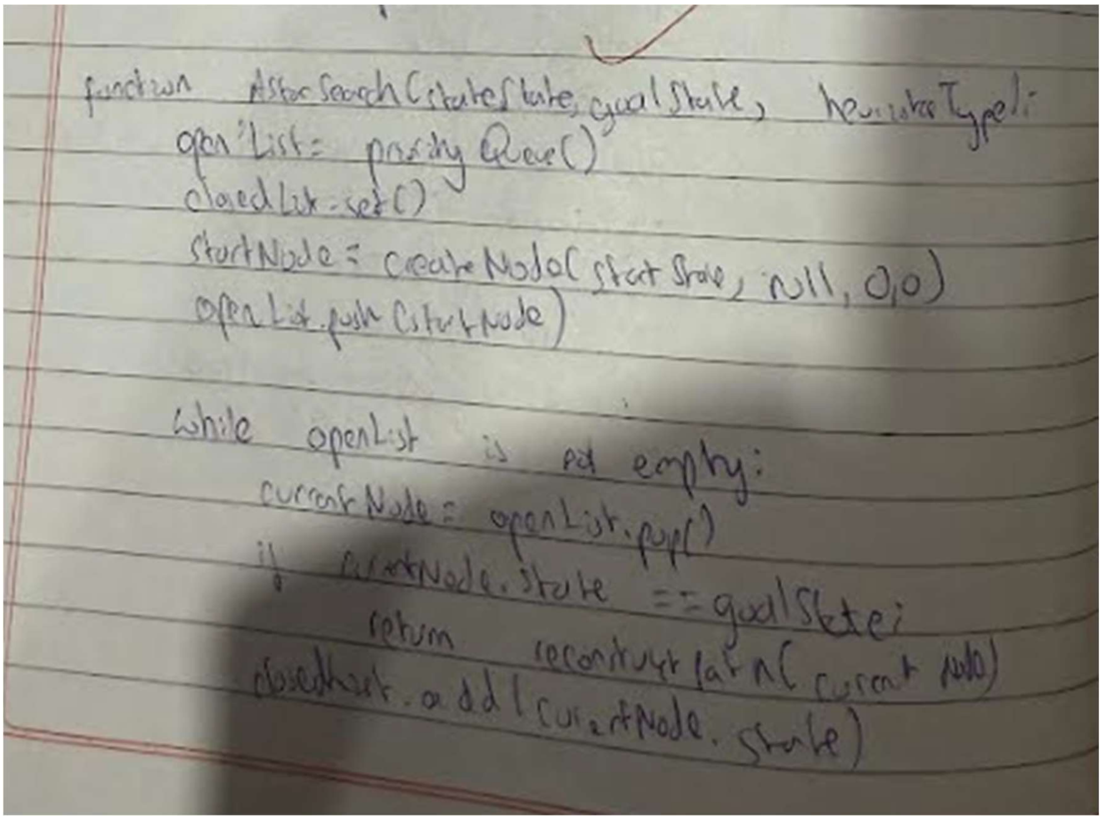
```
        self.board = board
self.goal = goal         self.size
= 3

    def find_blank(self, state):
for i in range(self.size):
for j in range(self.size):
if state[i][j] == 0:
                    return i, j

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

    def get_moves(self, state):         x, y =
self.find_blank(state)         moves = []
directions = [(-1,0),(1,0),(0,-1),(0,1)]         for dx, dy
in directions:             nx, ny = x+dx, y+dy             if
0 <= nx < self.size and 0 <= ny < self.size:
                new_state = deepcopy(state)                 new_state[x][y],
new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
moves.append(new_state)         return moves

    def dls(self, state, depth, path, visited):
if self.is_goal(state):             return path
+ [state]         if depth == 0:
return None         visited.append(state)
for move in self.get_moves(state):
if move not in visited:
                new_path = self.dls(move, depth - 1, path + [state], visited)
if new_path:
                    return new_path
return None

    def iddfs(self, start):
        depth = 0         while True:             visited
= []             result = self.dls(start, depth, [],
visited)             if result:                 return
result             depth += 1


def input_state(prompt):
    print(prompt)     state = []     for _ in
range(3):         row = list(map(int,
```

```
input().split()))        state.append(row)
return state


start = input_state("Enter the initial state row-wise (use 0 for blank):")
goal = input_state("Enter the goal state row-wise (use 0 for blank):")

puzzle = Puzzle(start, goal) path
= puzzle.iddfs(start)

print("\nTotal moves:", len(path)-1) for
step, state in enumerate(path):
    print(f"\nMove {step}:")
for row in state:
print(*row)
```

Output:

```
Enter the initial state row-wise (use 0 for blank):
1 2 3
0 4 6
7 5 8
Enter the goal state row-wise (use 0 for blank):
1 2 3
4 5 6
7 8 0

Total moves: 3

Move 0:
1 2 3
0 4 6
7 5 8

Move 1:
1 2 3
4 0 6
7 5 8

Move 2:
1 2 3
4 5 6
7 0 8

Move 3:
1 2 3
4 5 6
7 8 0
```

8 Puzzle problem using A* Pseudocode:




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

    def generate_child(self):
        x, y = self.find(self.data, '_')        directions = [[x - 1,
y], [x + 1, y], [x, y - 1], [x, y + 1]]        children = []
for i, j in directions:
            child = self.shuffle(self.data, x, y, i, j)
if child is not None:
                child_node = Node(child, self.level + 1, 0)
children.append(child_node)        return children
```

```
    def shuffle(self, puz, x1, y1, x2, y2):         if 0 <= x2 <
len(self.data) and 0 <= y2 < len(self.data):
            temp_puz = self.copy(puz)
            temp_puz[x1][y1], temp_puz[x2][y2] = temp_puz[x2][y2], temp_puz[x1][y1]
return temp_puz         return None

    def copy(self, root):
        return [row[:] for row in root]

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


class Puzzle:     def
__init__(self, size=3):
        self.n = size
self.open = []
self.closed = []

    def accept(self):
        puz = []         for i in
range(self.n):             row =
input().split()
puz.append(row)
return puz

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

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

    def process(self):
        print("Enter the start state matrix (use _ for blank):")
        start = self.accept()         print("Enter the goal state
matrix (use _ for blank):")         goal = self.accept()
```

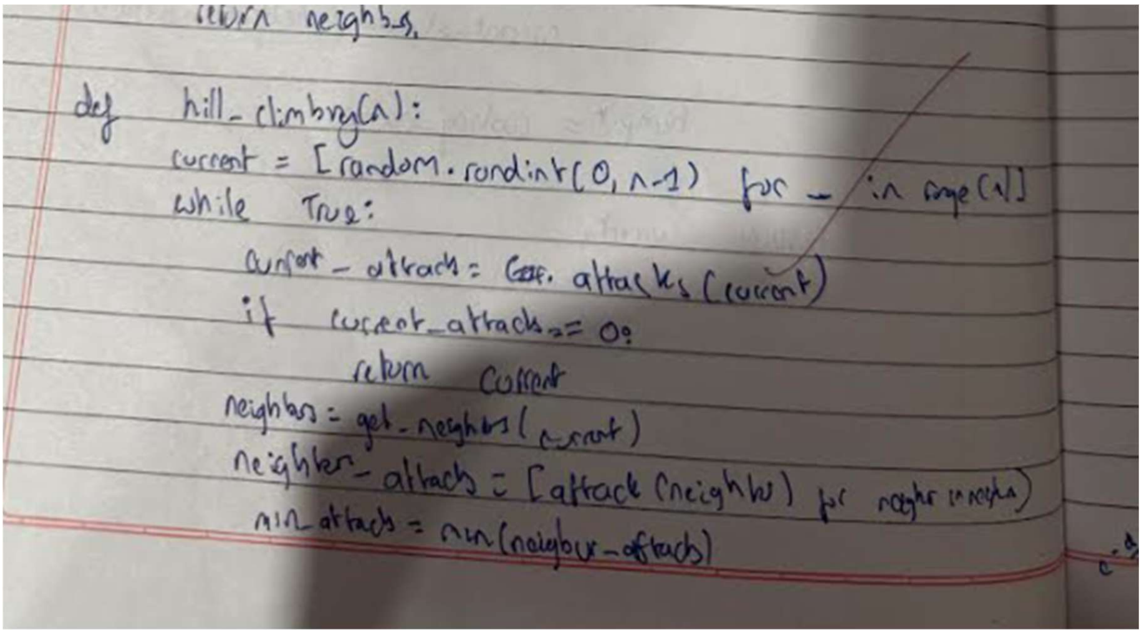
```
        start_node = Node(start, 0, 0)
start_node.fval = self.f(start_node, goal)
self.open.append(start_node)

        print("\nSolving...\n")
move = 0         while True:
            cur = self.open[0]
print(f"\nMove {move}:")             for
row in cur.data:                 print('
'.join(row))            if self.h(cur.data,
goal) == 0:                print("\nGoal
state reached!")                break

            for child in cur.generate_child():
child.fval = self.f(child, goal)
self.open.append(child)

            self.closed.append(cur)            del
self.open[0]
self.open.sort(key=lambda x: x.fval)
move += 1


if __name__ == "__main__":
    Puzzle().process()
```

Output:

```
Enter the start state matrix (use _ for blank):
1 2 3
_ 4 6
7 5 8
Enter the goal state matrix (use _ for blank):
1 2 3
4 5 6
7 8 _

Solving...


Move 0:
1 2 3
_ 4 6
7 5 8

Move 1:
1 2 3
4 _ 6
7 5 8

Move 2:
1 2 3
4 5 6
7 _ 8

Move 3:
1 2 3
4 5 6
7 8 _

Goal state reached!
```

Hill Climb Searching

Pseudocode:



```
return neighbors

def hill_climbing(n):
    current = [random.randint(0, n-1) for _ in range(n)]
    while True:
        current_attack = cal. attacks(current)
        if current_attack == 0:
            return current
        neighbors = get_neighbors(current)
        neighbor_attacks = [attack(neighbor) for neighbor in neighbors]
        min_attacks = min(neighbor_attacks)
```

N queens using Hill Climb Searching:

Code:

```
def print_board(state):
    """Prints the 4x4 board representation with 'Q' and '.'."""
n = len(state)     for row in range(n):         for col in
range(n):             if state[col] == row:                 print("Q",
end=" ")             else:                 print(".", end=" ")
print()     print()


def calculate_cost(state):
    """Returns number of attacking pairs of queens."""
    cost = 0     n = len(state)
for i in range(n):         for j
in range(i + 1, n):             #
same row             if state[i]
== state[j]:
                cost += 1             # same diagonal
elif abs(state[i] - state[j]) == abs(i - j):
                cost += 1
return cost
```

```
def get_neighbors(state):
   """Generates all neighbors by swapping two queen positions."""
neighbors = []     n = len(state)     for i in range(n):         for j in
range(i + 1, n):             neighbor = state.copy()             neighbor[i],
neighbor[j] = neighbor[j], neighbor[i]
neighbors.append((neighbor, (i, j)))     return neighbors


def hill_climbing(state):
print("\nInitial State:", state)
print_board(state)     current_cost =
calculate_cost(state)     step = 1

   while True:
       print(f"Step {step}: Current cost = {current_cost}")
neighbors = get_neighbors(state)         neighbor_costs =
[]

       # Calculate cost for all neighbors         for neighbor,
swapped in neighbors:             cost =
calculate_cost(neighbor)
neighbor_costs.append((cost, neighbor, swapped))         #
Sort by cost and then by smallest column pair as per rules
neighbor_costs.sort(key=lambda x: (x[0], x[2][0],
x[2][1]))

       # Display neighbor costs         print("Neighbor
states and their costs:")         for cost, neighbor,
swapped in neighbor_costs:
           print(f"Swap x{swapped[0]} & x{swapped[1]} => {neighbor}, Cost = {cost}")

       best_cost, best_state, swap = neighbor_costs[0]         print("\nBest Neighbor after
swap", swap, "is", best_state, "with cost =", best_cost)         print_board(best_state)

       if best_cost >= current_cost:  # No improvement (local
minimum)             print("No better neighbor found. Hill Climbing
terminated.")             print("Final state:", state)
print_board(state)             break         else:
           state = best_state
current_cost = best_cost
```

```
        if current_cost == 0:
print("Goal state reached!")
print_board(state)             break

        step += 1


# ---------------- MAIN ---------------- if
__name__ == "__main__":
    print("Hill Climbing for 4-Queens Problem")     print("Enter the row
positions of 4 queens (each between 0 and 3):")     state = list(map(int,
input("Example (1 2 0 3): ").split()))     hill_climbing(state)
```

Output:

```
Hill Climbing for 4-Queens Problem
Enter the row positions of 4 queens (each between 0 and 3):
Example (1 2 0 3): 0 1 2 3

Initial State: [0, 1, 2, 3]
Q . . .
. Q . .
. . Q .
. . . Q

Step 1: Current cost = 6
Neighbor states and their costs:
Swap x0 & x1 => [1, 0, 2, 3], Cost = 2
Swap x0 & x3 => [3, 1, 2, 0], Cost = 2
Swap x1 & x2 => [0, 2, 1, 3], Cost = 2
Swap x2 & x3 => [0, 1, 3, 2], Cost = 2
Swap x0 & x2 => [2, 1, 0, 3], Cost = 4
Swap x1 & x3 => [0, 3, 2, 1], Cost = 4

Best Neighbor after swap (0, 1) is [1, 0, 2, 3] with cost = 2
. Q . .
Q . . .
. . Q .
. . . Q

Step 2: Current cost = 2
Neighbor states and their costs:
Swap x0 & x2 => [2, 0, 1, 3], Cost = 1
Swap x0 & x3 => [3, 0, 2, 1], Cost = 1
Swap x1 & x2 => [1, 2, 0, 3], Cost = 1
Swap x1 & x3 => [1, 3, 2, 0], Cost = 1
Swap x2 & x3 => [1, 0, 3, 2], Cost = 4
Swap x0 & x1 => [0, 1, 2, 3], Cost = 6

Best Neighbor after swap (0, 2) is [2, 0, 1, 3] with cost = 1
. Q . .
. . Q .
Q . . .
. . . Q

Step 3: Current cost = 1
Neighbor states and their costs:
Swap x2 & x3 => [2, 0, 3, 1], Cost = 0
Swap x0 & x1 => [0, 2, 1, 3], Cost = 2
Swap x0 & x2 => [1, 0, 2, 3], Cost = 2
Swap x1 & x3 => [2, 3, 1, 0], Cost = 2
Swap x0 & x3 => [3, 0, 1, 2], Cost = 4
Swap x1 & x2 => [2, 1, 0, 3], Cost = 4

Best Neighbor after swap (2, 3) is [2, 0, 3, 1] with cost = 0
```

```
Best Neighbor after swap (2, 3) is [2, 0, 3, 1] with cost = 0
. Q . .
. . . Q
Q . . .
. . Q .

Goal state reached!
. Q . .
. . . Q
Q . . .
. . Q .
```

Simulated Annealing

Pseudocode:

10/9/25 LAB

```
import math
import random

def simulated_annealing(n, max_iter = 10000, initial_temp=100,
                        cooling_rate = 0.99):
    current = [random.randint(0, n-1) for _ in range(n)]
    current_attacks = attacks(current)
    temp= initial_temp

    for _ in range(max_iter):
        if current_attacks == 0:
            return current
        col = random.randint(0, n-1)
        row= random.randint(0, n-1)
        neighbor = list(current)
        neighbor[col] = row
        neighbor_attacks = attack(neighbor)

        delta = neighbor_attacks - current_attacks
        if delta <0 or random.random() < math.exp(-delta/temp):
            current = neighbor
            current_attacks = neighbor_attacks

        temp *= cooling_rate

    return current
```

N Queens using Simulated Annealing

Pseudocode: import random import math

def print_board(state): n = len(state) print("Board State:") for row in

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

def calculate_cost(state):
   cost = 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):
            cost += 1
return cost

def get_all_neighbors(state):
   neighbors = []     n = len(state)     for i in range(n):
for j in range(i + 1, n):             neighbor = state.copy()
neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
neighbors.append((neighbor, (i, j)))     return neighbors

def hill_climbing(state):
   print("\n--- Starting Hill Climbing ---")
current_state = state.copy()     current_cost =
calculate_cost(current_state)     step = 1

   print("Initial State:", current_state, "with Cost:", current_cost)
print_board(current_state)

   while True:
      print(f"--- Step {step} ---")        neighbors =
get_all_neighbors(current_state)
neighbor_costs = []

      for neighbor, swapped in neighbors:            cost =
calculate_cost(neighbor)
neighbor_costs.append((cost, neighbor, swapped))

      neighbor_costs.sort(key=lambda x: (x[0], x[2][0], x[2][1]))
best_cost, best_neighbor, best_swap = neighbor_costs[0]
```

```
        print(f"Current State: {current_state}, Cost: {current_cost}")        print(f"Best
neighbor is {best_neighbor} with cost {best_cost} (Swap columns {best_swap[0]}
& {best_swap[1]})")

        if best_cost >= current_cost:
            print("\nNo better neighbor found. Reached a local minimum or plateau.")
print("Final State:", current_state)            print_board(current_state)
print("Final Cost:", current_cost)            break

        current_state = best_neighbor
current_cost = best_cost
print_board(current_state)

        if current_cost == 0:
            print("\nGoal state reached!")
print("Final State:", current_state)
print_board(current_state)            print("Final
Cost:", current_cost)            break

        step += 1

def get_random_neighbor(state):     n = len(state)
neighbor = state.copy()     i, j =
random.sample(range(n), 2)     neighbor[i],
neighbor[j] = neighbor[j], neighbor[i]     return
neighbor

def simulated_annealing(state, initial_temp=100.0, cooling_rate=0.95, max_steps=1000):
    print("\n--- Starting Simulated Annealing ---")
current_state = state.copy()     current_cost =
calculate_cost(current_state)     temp =
initial_temp

    print("Initial State:", current_state, "with Cost:", current_cost)
print_board(current_state)

    for step in range(max_steps):
if temp < 0.01:
            print("\nTemperature cooled down. Algorithm terminated.")
break

        if current_cost == 0:
```

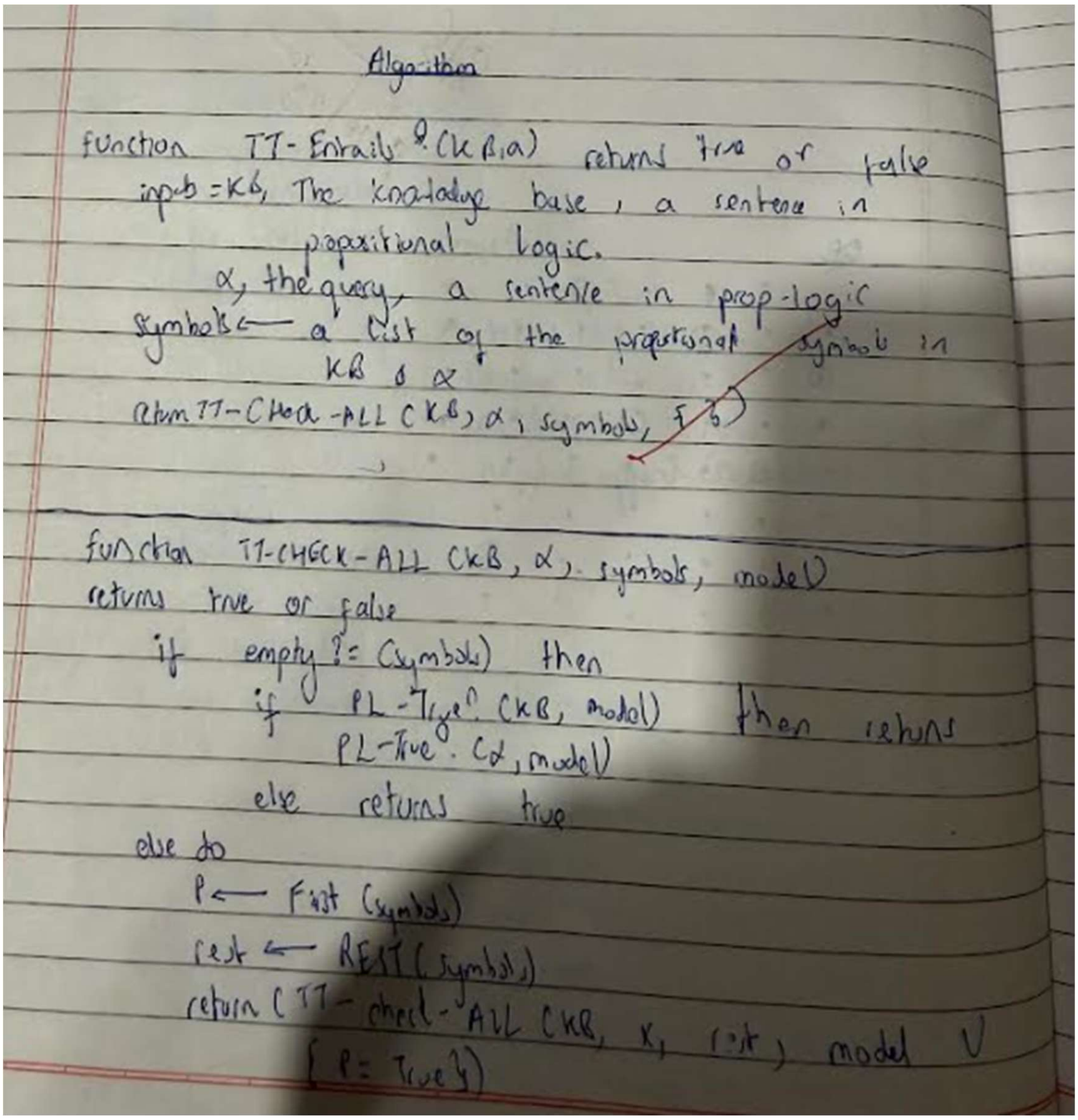
```
            print(f"\nGoal state reached in {step+1} steps!")
break

        neighbor_state = get_random_neighbor(current_state)
neighbor_cost = calculate_cost(neighbor_state)         delta_cost
= neighbor_cost - current_cost

        if delta_cost < 0:
            print(f"Step {step+1}: Good move. Moved to {neighbor_state} (Cost: {neighbor_cost})")
current_state = neighbor_state             current_cost = neighbor_cost          else:
            acceptance_prob = math.exp(-delta_cost / temp)             if random.random() <
acceptance_prob:                 print(f"Step {step+1}: Bad move accepted! Moved to
{neighbor_state} (Cost:
{neighbor_cost}) with prob {acceptance_prob:.2f}")                 current_state =
neighbor_state                 current_cost = neighbor_cost             else:
print(f"Step {step+1}: Bad move rejected. Staying at {current_state} (Cost:
{current_cost})")

        temp *= cooling_rate

    print("\nFinal State:", current_state)
print_board(current_state)     print("Final
Cost:", current_cost)

if __name__ == "__main__":
    print("--- 4-Queens Problem Solver ---")     print("The state is a list of 4 numbers
representing the row of the queen in each column.")     print("Example: 1 3 0 2 means -> Q
in col 0 row 1, col 1 row 3, col 2 row 0, col 3 row 2\n")

try:
        initial_state = list(map(int, input("Enter the initial state (e.g., 2 0 3 1): ").split()))
if len(initial_state) != 4 or not all(0 <= x <= 3 for x in initial_state):
print("Invalid input. Please enter 4 numbers between 0 and 3.")         else:
            simulated_annealing(initial_state)
except ValueError:
        print("Invalid input format. Please enter numbers separated by spaces.")
```

Output:

```
Enter the initial state (e.g., 2 0 3 1): 0 1 2 3

--- Starting Simulated Annealing ---
Initial State: [0, 1, 2, 3] with Cost: 6
Board State:
Q . . .
. Q . .
. . Q .
. . . Q

Step 1: Good move. Moved to [0, 1, 3, 2] (Cost: 2)
Step 2: Bad move accepted! Moved to [1, 0, 3, 2] (Cost: 4) with prob 0.98
Step 3: Bad move accepted! Moved to [3, 0, 1, 2] (Cost: 4) with prob 1.00
Step 4: Good move. Moved to [3, 1, 0, 2] (Cost: 1)
Step 5: Bad move accepted! Moved to [3, 2, 0, 1] (Cost: 2) with prob 0.99
Step 6: Good move. Moved to [3, 1, 0, 2] (Cost: 1)
Step 7: Bad move accepted! Moved to [3, 2, 0, 1] (Cost: 2) with prob 0.99
Step 8: Good move. Moved to [0, 2, 3, 1] (Cost: 1)
Step 9: Bad move accepted! Moved to [3, 2, 0, 1] (Cost: 2) with prob 0.99
Step 10: Good move. Moved to [3, 1, 0, 2] (Cost: 1)
Step 11: Bad move accepted! Moved to [3, 0, 1, 2] (Cost: 4) with prob 0.95
Step 12: Bad move accepted! Moved to [3, 2, 1, 0] (Cost: 6) with prob 0.97
Step 13: Good move. Moved to [3, 0, 1, 2] (Cost: 4)
Step 14: Good move. Moved to [3, 1, 0, 2] (Cost: 1)
Step 15: Bad move accepted! Moved to [3, 1, 2, 0] (Cost: 2) with prob 0.98
Step 16: Bad move accepted! Moved to [3, 2, 1, 0] (Cost: 6) with prob 0.92
Step 17: Good move. Moved to [3, 2, 0, 1] (Cost: 2)
Step 18: Bad move accepted! Moved to [2, 3, 0, 1] (Cost: 4) with prob 0.95
Step 19: Bad move accepted! Moved to [2, 1, 0, 3] (Cost: 4) with prob 1.00
Step 20: Good move. Moved to [1, 2, 0, 3] (Cost: 1)
Step 21: Bad move accepted! Moved to [1, 0, 2, 3] (Cost: 2) with prob 0.97
Step 22: Bad move accepted! Moved to [0, 1, 2, 3] (Cost: 6) with prob 0.89
Step 23: Good move. Moved to [1, 0, 2, 3] (Cost: 2)
Step 24: Good move. Moved to [1, 2, 0, 3] (Cost: 1)
Step 25: Good move. Moved to [1, 3, 0, 2] (Cost: 0)

Goal state reached in 26 steps!

Final State: [1, 3, 0, 2]
Board State:
. . Q .
Q . . .
. . . Q
. Q . .

Final Cost: 0
```

Propositional Logic

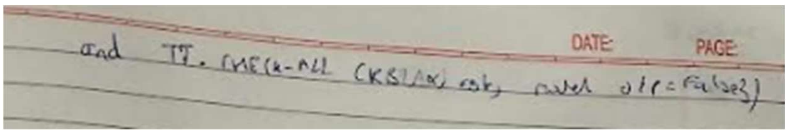
Pseudocode:



Algorithm

function TT-Entails? (KB, α) returns true or false
inputs = KB, The knowledge base, a sentence in propositional logic.
α, the query, a sentence in prop-logic
symbols ← a list of the propositional symbols in KB & α
return TT-CHECK-ALL (KB, α, symbols, { })

function TT-CHECK-ALL (KB, α, symbols, model)
returns true or false
if empty? (symbols) then
if PL-True? (KB, model) then returns PL-True? (α, model)
else returns true
else do
P ← First (symbols)
rest ← REST (symbols)
return (TT-check-ALL (KB, α, rest, model ∪ {P = True})



DATE: PAGE:

and TT-CHECK-ALL (KB, α, rest, model ∪ {P = False}))

Code:
import itertools import
pandas as pd

```
# Define the propositional logic sentences variables
= ['P', 'Q', 'R']

def q_implies_p(p, q, r):
  return not q or p

def p_implies_not_q(p, q, r):
  return not p or not q

def q_or_r(p, q, r):
  return q or r

sentences = {
    'Q implies P': q_implies_p,
    'P implies not(Q)': p_implies_not_q,
    'Q or R': q_or_r
}

# Generate a truth table truth_values = [True, False] combinations =
list(itertools.product(truth_values, repeat=len(variables)))

truth_table_data = [] for
combination in combinations:
  p, q, r = combination   row = {'P': p, 'Q': q, 'R': r}    for
sentence_name, sentence_func in sentences.items():
    row[sentence_name] = sentence_func(p, q, r)
truth_table_data.append(row)

# Convert the list of dictionaries to a pandas DataFrame for better visualization truth_table_df
= pd.DataFrame(truth_table_data)

# Evaluate the knowledge base (KB) sentence_columns =
list(sentences.keys()) truth_table_df['KB is True'] =
truth_table_df[sentence_columns].all(axis=1)

models = truth_table_df[truth_table_df['KB is True']]

# Check for entailment kb_entails_r = models['R'].all() kb_entails_r_implies_p =
(models['R'].apply(lambda x: not x) | models['P']).all() kb_entails_q_implies_r =
(models['Q'].apply(lambda x: not x) | models['R']).all()
```

Display the results print("Truth
Table:") display(truth_table_df)

print("\nModels where KB is True:") display(models)

print("\nEntailment Results:") print(f"KB entails R:
{kb_entails_r}") print(f"KB entails R implies P:
{kb_entails_r_implies_p}") print(f"KB entails Q implies R:
{kb_entails_q_implies_r}")

Output:

Truth Table:

	P	Q	R	Q implies P	P implies not(Q)	Q or R	KB is True
0	True	True	True	True	False	True	False
1	True	True	False	True	False	True	False
2	True	False	True	True	True	True	True
3	True	False	False	True	True	False	False
4	False	True	True	False	True	True	False
5	False	True	False	False	True	True	False
6	False	False	True	True	True	True	True
7	False	False	False	True	True	False	False

Models where KB is True:

	P	Q	R	Q implies P	P implies not(Q)	Q or R	KB is True
2	True	False	True	True	True	True	True
6	False	False	True	True	True	True	True

```
Entailment Results:
KB entails R: True
KB entails R implies P: False
KB entails Q implies R: True
```

Unification

Pseudocode:

Bafna Gold
Date: Page:

Implementing Unification in First order Logic

Unify (Ψ_1, Ψ_2)

Step 1: If Ψ_1 or Ψ_2 is a variable or constant then:
(a) If Ψ_1, Ψ_2 are identical, then return NIL
(b) Else if Ψ_1 is a variable,
(a) then if Ψ_1 occurs in Ψ_2, then return Failure
(b) Else return $\{(\Psi_2/\Psi_1)\}$.
(c) Else if Ψ_2 is a variable,
(a) If Ψ_2 occurs in Ψ_1 then return failure
(b) Else return $\{(\Psi_1/\Psi_2)\}$
(d) Else return Failure

Step 2: If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE
Step 3: IF Ψ_1 and Ψ_2 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 Ψ_1
(a) Call Unify function with the ith element of Ψ_1 with ith element of Ψ_2 and put the result into S.
(b) If S = failure then returns Failure.
(c) If S ≠ NIL then do,
(a) apply S to the remainder of both L1 and L2.
b) SUBST = APPEND(S, SUBST)
Step 6: Return SUBST

Code: import
re

```
# Utility: parse the expression into function/operator and arguments
def parse(expr):     expr = expr.strip()     if '(' not in expr:
return expr, []     func = expr[:expr.index('(')].strip()     args =
expr[expr.index('(')+1:-1]     args = [a.strip() for a in
split_args(args)]     return func, args

# Split arguments correctly (handles nested brackets)
def split_args(args_str):     args, level, start = [], 0, 0
for i, ch in enumerate(args_str):         if ch == ',' and
level == 0:
            args.append(args_str[start:i].strip())
start = i + 1         elif ch == '(':             level
+= 1         elif ch == ')':             level -= 1
args.append(args_str[start:].strip())
return args

# Apply substitution to an expression
def substitute(expr, subs):     for var,
val in subs.items():
        expr = re.sub(rf'\b{var}\b', val, expr)
return expr

# Check if variable occurs inside term (Occurs check)
def occurs_check(var, term):     if var == term:
return True     if '(' not in term:         return False
    _, args = parse(term)     return
any(occurs_check(var, arg) for arg in args)

# Unification algorithm def
unify(e1, e2, subs=None):
if subs is None:         subs =
{}

    e1 = substitute(e1, subs)
e2 = substitute(e2, subs)

    if e1 == e2:
return subs

    f1, args1 = parse(e1)
f2, args2 = parse(e2)
```

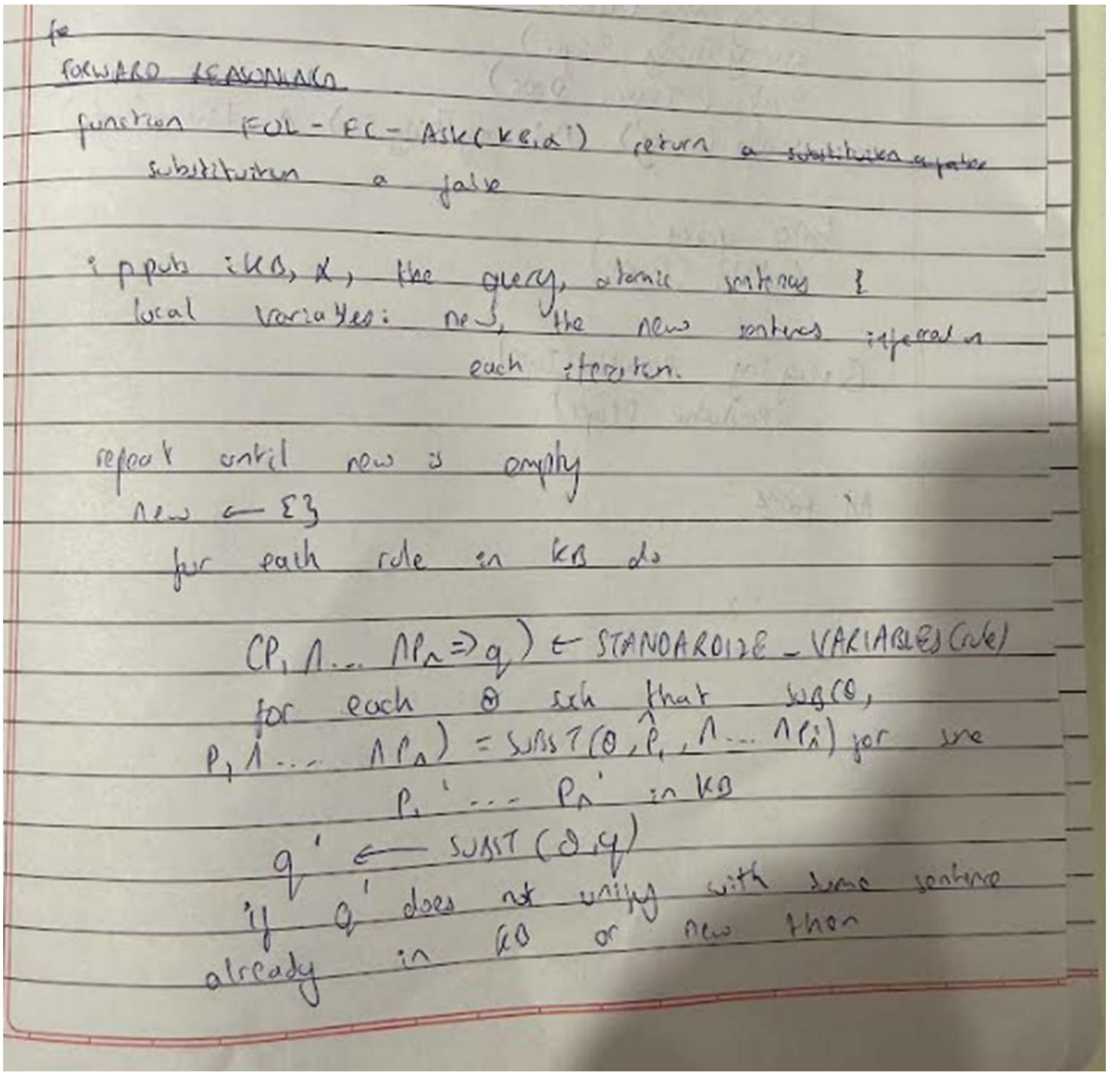
```
    # Case 1: Both are compound terms
if args1 and args2:          if f1 != f2 or
len(args1) != len(args2):
            print(f" Function symbols or arity mismatch: {f1} vs {f2}")
return None          for a1, a2 in zip(args1, args2):             subs =
unify(a1, a2, subs)             if subs is None:                 return None
return subs

    # Case 2: Variable binding     elif e1.islower()
and e1.isalpha(): # e1 is variable          if
occurs_check(e1, e2):
            print(f" Occurs check failed: {e1} occurs in {e2}")
return None          subs[e1] = e2          return subs     elif
e2.islower() and e2.isalpha(): # e2 is variable          if
occurs_check(e2, e1):
            print(f" Occurs check failed: {e2} occurs in {e1}")
return None          subs[e2] = e1
        return subs

    # Otherwise mismatch
else:
        print(f" Cannot unify {e1} with {e2}")
return None

# --- MAIN PROGRAM --- print("===
Unification Algorithm ===") expr1 =
input("Enter first expression: ").strip() expr2 =
input("Enter second expression: ").strip()

result = unify(expr1, expr2)

if result:
    print("\n Unification Successful!")
print("Substitutions:")     for k, v in
result.items():          print(f" {k} /
{v}") else:     print("\n Unification
Failed.")
```

Output:

```
Python 3.11.9 (tags/v3.11.9:de54cf5, Apr  2 2024, 10:12:12) [MSC v.1938 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: D:\desktop data\AI LAb programs.py
=== Unification Algorithm ===
Enter first expression: P(f(x), g(y), y)
Enter second expression: P(f(g(z)), g(f(a)), f(a))

☑ Unification Successful!
Substitutions:
 x / g(z)
 y / f(a)

================== RESTART: D:\desktop data\AI LAb programs.py =================
=== Unification Algorithm ===
Enter first expression: Q(x, f(x))
Enter second expression: Q(f(y), y)
✖ Occurs check failed: y occurs in f(f(y))

✖ Unification Failed.

================== RESTART: D:\desktop data\AI LAb programs.py =================
=== Unification Algorithm ===
Enter first expression: H(x, g(x))
Enter second expression: H(g(y), g(g(z)))

☑ Unification Successful!
Substitutions:
```

Forward Reasoning

Pseudocode:




```
Code: import
re def
isVariable(x)
:
    return len(x) == 1 and x.islower() and x.isalpha()

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

def getPredicates(string):
```

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

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

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

    def getResult(self):
return self.result

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

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

    def substitute(self, constants):
        constants_copy = constants.copy()         expr =
f"{self.predicate}({','.join([constants_copy.pop(0) if isVariable(p) else p for p in
self.params])})"         return Fact(expr)

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

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

```
        predicate = getPredicates(self.rhs.expression)[0]
attributes = str(getAttributes(self.rhs.expression)[0])

        for key in constants:
if constants[key]:
                attributes = attributes.replace(key, constants[key])

        expr = f'{predicate}{attributes}'         return Fact(expr) if len(new_lhs) and
all([f.getResult() for f in new_lhs]) else None

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

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

    def ask(self, e):
        facts = set([f.expression for f in self.facts])
print(f'\nQuerying {e}:')        i = 1         found
= False         for f in facts:             if
Fact(f).predicate == Fact(e).predicate:
                print(f'\t{i}. {f}')
i += 1                 found =
True         if not found:
            print("\tNo matching facts found.")

    def display(self):         print("\nAll facts:")         for i, f in
enumerate(set([f.expression for f in self.facts])):
            print(f'\t{i+1}. {f}')

def main():     kb = KB()     print("Enter the number of FOL
expressions present in KB:")     n = int(input())     print("Enter
the expressions:")     for i in range(n):         fact = input().strip()
kb.tell(fact)
```

```
    print("Enter the query:")
query = input().strip()
kb.ask(query)     kb.display()

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

Output:

```
Enter the number of FOL expressions present in KB:
4
Enter the expressions:
LivingThing(Deer)
LivingThing(Tiger)
Eats(Tiger, Deer)
Eats(x, y) ^ LivingThing(x) ^ LivingThing(y) => Predator(x)
Enter the query:
Predator(Tiger)

Querying Predator(Tiger):
        1. redator(Tiger)

All facts:
        1. LivingThing(Tiger)
        2. LivingThing(Deer)
        3. redator(Tiger)
        4. Eats(Tiger, Deer)
|
```

Resolution

Pseudocode:

Resolution FOL and α-β pruning

Converting logic statement to CNF:

1. Eliminate biconditionals and implications
   - Eliminate $\longleftrightarrow$, replacing $\alpha \longleftrightarrow \beta$ with $(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$
   - Eliminate $\longrightarrow$, replacing $\alpha \rightarrow \beta$ with $\neg\alpha \vee \beta$

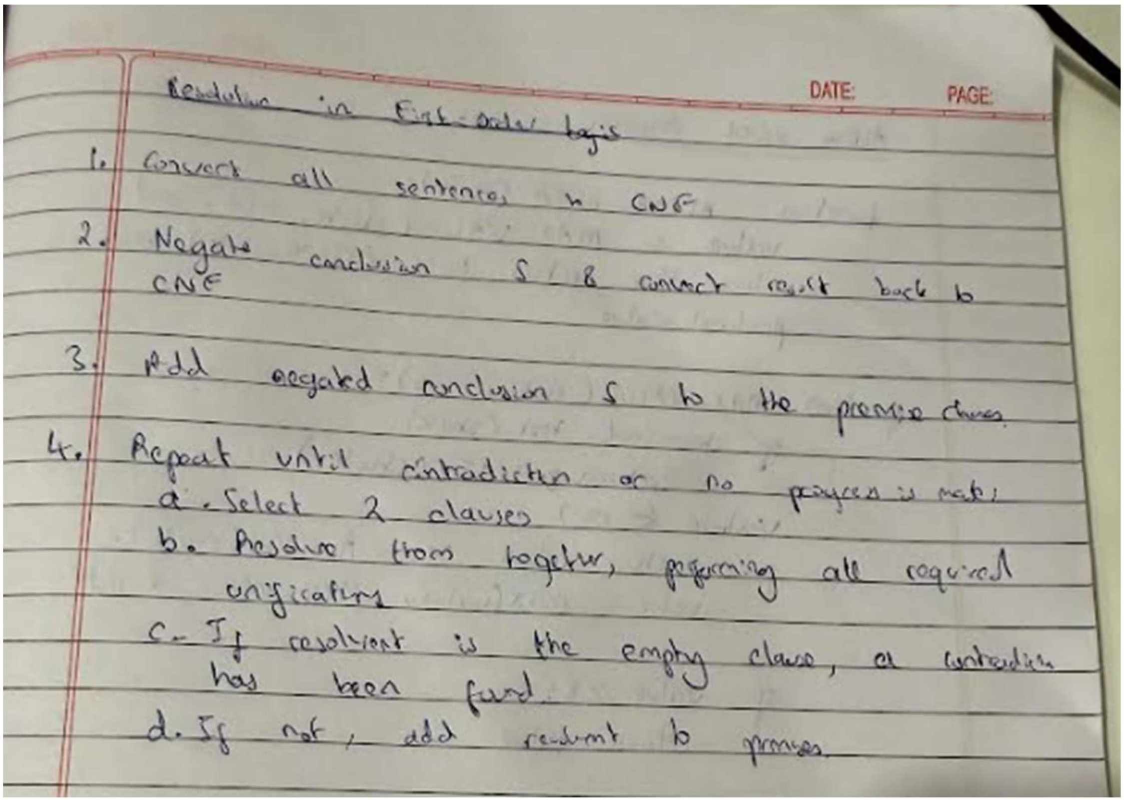
2. Move $\neg$ inwards:
   - $\neg(\forall x\ p) \equiv \exists x\ \neg p$
   - $\neg(\exists x\ p) \equiv \forall x\ \neg p$
   - $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$
   - $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$
   - $\neg\neg\alpha \equiv \alpha$

3. Standardize variables apart by renaming them

4. Skolemize: each existential variable is replaced by a Skolem constant or Skolem function.
   - $\exists x\ Rich(x)$ becomes $Rich(G1)$ where $G1$ is a new Skolem constant.

Drop universal quantifiers
   - For instance, $\forall x\ Person(x)$ becomes $Person(x)$

Distribute $\wedge$ over $\vee$.
   - $(\alpha \wedge \beta) \vee \gamma \equiv (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$



Resolution in First-order logic

1. Convert all sentences to CNF
2. Negate conclusion S & convert result back to CNF
3. Add negated conclusion S to the premise clauses
4. Repeat until contradiction or no progress is made:
   a. Select 2 clauses
   b. Resolve them together, performing all required unifications
   c. If resolvent is the empty clause, a contradiction has been found
   d. If not, add resultant to premises

Code:

```
def parse_clause(clause_str):
return set(clause_str.split('v'))


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


def resolve(ci, cj):
resolvents = set()      for
literal in ci:
      complement = get_complement(literal)
if complement in cj:
         new_clause = (ci - {literal}) | (cj - {complement})
resolvents.add(frozenset(new_clause))      return
resolvents


def resolution(kb_clauses, query):
   negated_query = get_complement(query)      kb = [parse_clause(clause) for clause in
kb_clauses] + [parse_clause(negated_query)]
```

```
   print("\n----------------------------")
print("KnowledgeBase - Resolution")     print("-------
---------------------")     print(f"\nKnowledge Base
Clauses: {kb_clauses}")     print(f"Query: {query}")
print(f"Negated Query Added: {negated_query}")
print("\nResolution Steps:\n")

   new = set()

   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:
           resolvents = resolve(ci, cj)
for resolvent in resolvents:
               print(f"Resolving {set(ci)} and {set(cj)} => {set(resolvent)}")
if not resolvent:
                   print("\n Knowledge Base entails the query (empty clause derived).")
return True                 new.add(resolvent)

       if new.issubset(set(map(frozenset, kb))):
           print("\n Knowledge Base does NOT entail the query (no empty clause derived).")
return False

       for clause in new:
if clause not in kb:
kb.append(clause)


print("KnowledgeBase - Resolution") print("----------------------------") print("Enter clauses for
the Knowledge Base.") print("Use 'v' for OR between literals (e.g., '~qv~pvr'), and separate each
clause with a space.\n")

kb_input = input("Enter clauses: ").split() query_input
= input("Enter the query: ")

resolution(kb_input, query_input)
```

Output:

```
KnowledgeBase - Resolution
-----------------------------
Enter clauses for the Knowledge Base.
Use 'v' for OR between literals (e.g., '~qv~pvr'), and separate each clause with
 a space.

Enter clauses: (~q v ~p v r) ^ (~q ^ p) ^ q
Enter the query: r

-----------------------------
KnowledgeBase - Resolution
-----------------------------

Knowledge Base Clauses: ['(~q', 'v', '~p', 'v', 'r)', '^', '(~q', '^', 'p)', '^'
, 'q']
Query: r
Negated Query Added: ~r

Resolution Steps:


 Knowledge Base does NOT entail the query (no empty clause derived).
```

Alpha Beta Pruning

Pseudocode:

```
ALPHA BETA PRUNING

function   ALPHA_BETA (state):
        value  =  MAX_VALUE(state, -∞, ∞)
        return  the  action  from  ACTIONS(state) that
        produced value

function  MAX_VALUE( state, α, β):
        if  terminal_ test (state):
                return UTILITY(state)
        value ← ∞)
        for  each  action  in  ACTIONS(state):
              value = max(value,  MIN_VALUE, α, β))

        if   value ≥ β:
              return  value

        α = max(α, value)

  return   Value

function    MIN_VALUE( state, α, β):
      if    TERMINAL_TEST(state):
              return   UTILITY(state)

value  = +∞

for   each   action   in   Actions(state):
     value = min Value,   MAX_VALUE
     if   value  ≤ α :
             return    value

             β = min(β, value)
return   value
```

Tic Tac Toe using Alpha Beta Code:

```
import math import
random

# Use an external "real" board only for the main game loop; recursive functions use state parameters.
board = [" " for _ in range(9)]  # 3x3 board

def print_board(state):
print("\n")     for i in
range(3):
        print(" " + " | ".join(state[i*3:(i+1)*3]))
if i < 2:            print("---+---+---")
print("\n")

def is_winner(state, player):
win_combinations = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],
        [0, 3, 6], [1, 4, 7], [2, 5, 8],
        [0, 4, 8], [2, 4, 6]
    ]
    return any(all(state[i] == player for i in combo) for combo in win_combinations)

def is_full(state):
    return " " not in state

def actions(state):
    return [i for i in range(9) if state[i] == " "]

def result(state, action, player):
new_state = state.copy()
new_state[action] = player     return
new_state

def utility(state):     if
is_winner(state, "O"):
        return +1     elif
is_winner(state, "X"):
return -1     else:
        return 0

def terminal_test(state):
    return is_winner(state, "X") or is_winner(state, "O") or is_full(state)
```

```
# --- Alpha-Beta Functions --- def
max_value(state, alpha, beta):
if terminal_test(state):        return
utility(state)     v = -math.inf
for a in actions(state):
        v = max(v, min_value(result(state, a, "O"), alpha, beta))
if v >= beta:           return v        alpha = max(alpha, v)
return v

def min_value(state, alpha, beta):
if terminal_test(state):
return utility(state)     v =
math.inf     for a in actions(state):
        v = min(v, max_value(result(state, a, "X"), alpha, beta))
if v <= alpha:           return v        beta = min(beta, v)
return v

def alpha_beta_search(state):
best_score = -math.inf
best_action = None     if not
actions(state):
        return None     for
a in actions(state):
        value = min_value(result(state, a, "O"), -math.inf, math.inf)
if value > best_score:           best_score = value
best_action = a
    # Fallback: if something goes wrong, return a random legal move
if best_action is None:        legal = actions(state)        return
random.choice(legal) if legal else None     return best_action

# --- Game Loop ---
def human_move():
while True:
try:
            move = int(input("Enter your move (1-9): ")) - 1
except ValueError:
            print("Please enter a number 1-9.")
continue        if move < 0 or move > 8:
            print("Move out of range. Choose 1-9.")
continue        if board[move] != " ":
            print("Cell already taken. Try another.")
continue        return move
```

```
def choose_first():
while True:
        ans = input("Who goes first? (me/ai) [me]: ").strip().lower()
if ans == "" or ans.startswith("m"):
            return "me"         if
ans.startswith("a"):
            return "ai"         print("Type 'me' or 'ai' (or
press Enter for me).")

def main():     global board     board = [" " for _ in
range(9)]     print("Welcome to Tic-Tac-Toe! You are X,
AI is O.")     first = choose_first()     print_board(board)

    while True:         if
first == "me":
# Human turn
move =
human_move()
board[move] = "X"
print_board(board)
if is_winner(board,
"X"):
print("You win!")
break            if
is_full(board):
                print("It's a draw!")
break

            # AI turn            print("AI is
thinking...")            ai_move =
alpha_beta_search(board)            if
ai_move is None:
                print("AI could not find a move — it's a draw.")
break            board[ai_move] = "O"
print_board(board)            if is_winner(board, "O"):
print("AI wins!")                break            if is_full(board):
                print("It's a draw!")
break

        else:  # AI first            print("AI is
thinking...")            ai_move =
```

```
alpha_beta_search(board)              if
ai_move is None:
              print("AI could not find a move — it's a draw.")
break             board[ai_move] = "O"
print_board(board)             if is_winner(board, "O"):
print("AI wins!")                 break             if is_full(board):
              print("It's a draw!")
break

          # Human turn
move = human_move()
board[move] = "X"
print_board(board)             if
is_winner(board, "X"):
print("You win!")
break             if is_full(board):
print("It's a draw!")
break

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

Output:

```
Welcome to Tic-Tac-Toe! You are X, AI is O.
Who goes first? (me/ai) [me]: me


   |   |   
---+---+---
   |   |   
---+---+---
   |   |   


Enter your move (1-9): 5


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


AI is thinking...


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


Enter your move (1-9): 7


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


AI is thinking...
```

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

Enter your move (1-9): 6

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

AI is thinking...

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

Enter your move (1-9): 9

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

It's a draw!
```