Artificial Intelligence Lab Report

Submitted by

Ariz Ejaz Khan(1BM23CS051)
Batch: A3

Course: Artificial Intelligence
Course Code: 23CS5PCAIN
Sem & Section: 5A

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B. M. S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
2025-2026

B.M.S. COLLEGE OF ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that the Artificial Intelligence(23CS5PCAIN) laboratory has been carried out by Ariz Ejaz Khan (1BM23CS051) during the 5^{th} Semester August 2025-December 2025

Signature of the Faculty In charge:

Ms. Sandhya A Kulkarni
Assistant Professor

Department of Computer Science and Engineering
B.M.S. College of Engineering, Bangalore

Table of contents

Certificates

Infosys
Navigate your next

CERTIFICATE OF ACHIEVEMENT

The certificate is awarded to

Ariz Ejaz Khan

for successfully completing

Artificial Intelligence Foundation Certification

on November 21, 2025

image not available

Infosys | Springboard

Congratulations! You make us proud!

Issued on: Friday, November 21, 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

Ariz Ejaz Khan

for successfully completing the course

Introduction to Artificial Intelligence

on November 12, 2025

Infosys | Springboard

Congratulations! You make us proud!

Issued on: Wednesday, November 12, 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

Ariz Ejaz Khan

for successfully completing the course

Introduction to Deep Learning

on November 15, 2025

Infosys | Springboard

Congratulations! You make us proud!

Issued on: Saturday, November 15, 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

Ariz Ejaz Khan

for successfully completing the course

Introduction to Natural Language Processing

on November 18, 2025

Infosys | Springboard

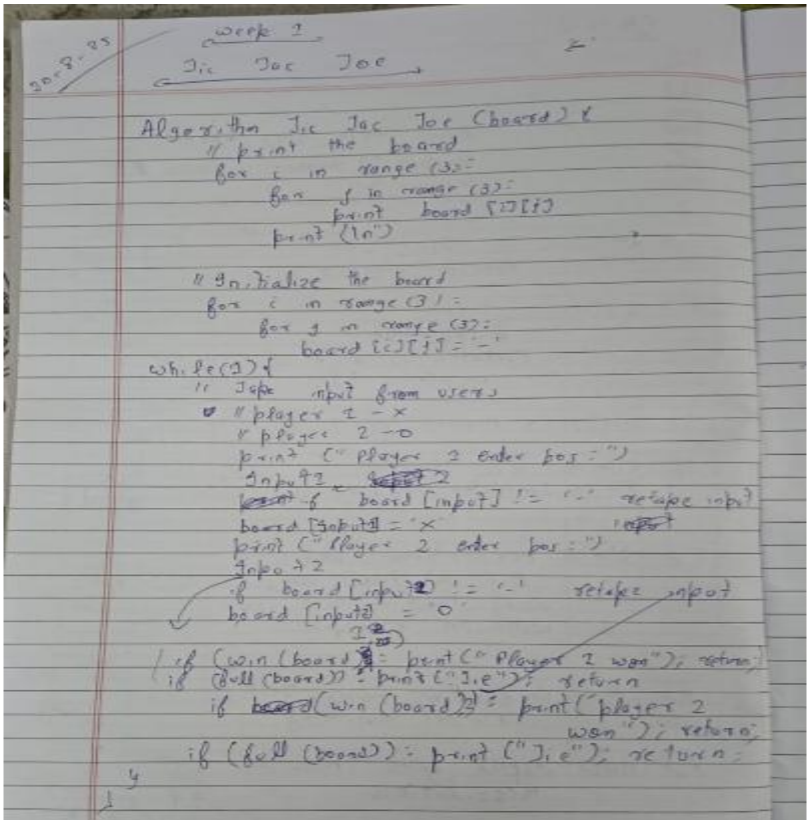
Congratulations! You make us proud!

Issued on: Tuesday, November 18, 2025
To verify, scan the QR code at https://verify.onwingspan.com

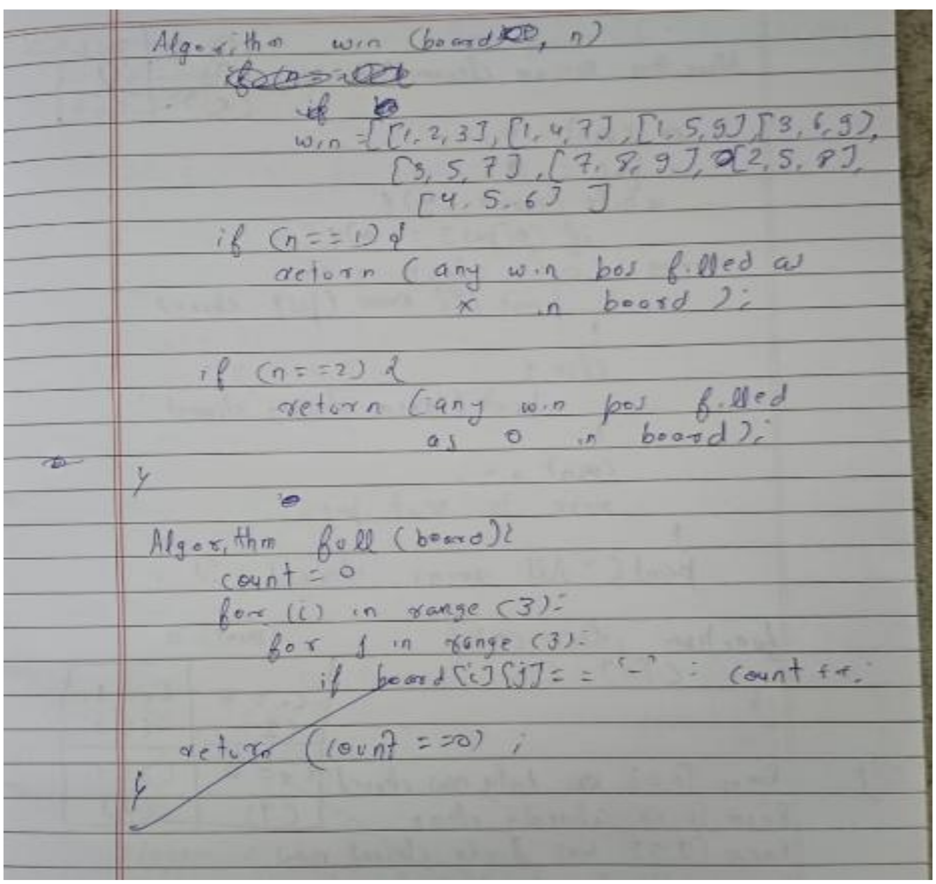
Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Program 1 - Tic Tac toe

Algorithm



```
20-3-25          week 1
        Tic Tac Toe

Algorithm Tic Tac Toe (board) {
    // print the board
    for i in range (3):
        for j in range (3):
            print board [i][j]
        print ("\n")

    // Initialize the board
    for i in range (3):
        for j in range (3):
            board [i][j] = '-'
while(1){
    // Take input from user
    // player 1 - x
    // player 2 - o
    print ("Player 1 enter pos:")
    input1
    if board [input] != '-' retake input
    board [input1] = 'x'
    print ("Player 2 enter pos:")
    input2
    if board [input2] != '-' retake input
    board [input2] = 'o'
    if (win (board, 1)): print ("Player 1 won"); return
    if (full (board)): print ("Tie"); return
    if (win (board, 2)): print ("player 2 won"); return;
    if (full (board)): print ("Tie"); return;
}
}
```



```
Algorithm win (board, n)
    win = [[1,2,3],[1,4,7],[1,5,9],[3,6,9],
           [3,5,7],[7,8,9],[2,5,8],
           [4,5,6]]
    if (n==1){
        return (any win pos filled as
                x in board);
    }
    if (n==2){
        return (any win pos filled
                as o in board);
    }
}

Algorithm full (board){
    count = 0
    for (i) in range (3):
        for j in range (3):
            if board[i][j] == '-' : count++;
    return (count == 0);
}
```

Tic Tac Toe

O/P :- Tic Tac Toe. Two player (1-9 version)
['-', '-', '-']
['-', '-', '-']
['-', '-', '-']
Player X position : 1
[X, -, -]
[-, -, -]
[-, -, -]

Player O, position : 2
[X, O, -]
[-, -, -]
[-, -, -]

Player X, position : 4
[X, O, -]
[X, -, -]
[-, -, -]

Player O, position : 3
[X, O, O]
[X, -, -]
[-, -, -]

Player X, position : 4
That cell is already taken. Try again

Player X, position : 7
[X, O, O]
[X, -, -]
[X, -, -]
Player X wins!

Code

```
def create_board():
    return [["-" for _ in range(3)] for _ in range(3)]
def show_board(board):
    for row in board:
        # print(" | ".join(row))
        print(row," ")

# Map cell number (1-9) to row, col
def cell_to_coords(cell):
    cell -= 1 # shift to 0–8
    return cell // 3, cell % 3
def is_full(board):
    return all(board[r][c] != "-" for r in range(3) for c in range(3))

def check_winner(board, player):
    # rows
    for r in range(3):
        if all(board[r][c] == player for c in range(3)):
            return True
    # cols
    for c in range(3):
        if all(board[r][c] == player for r in range(3)):
            return True
    # diagonals
    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

# Player move
def get_move(board, player):
    while True:
        try:
            cell = int(input(f"Player {player}, position: "))
            if cell < 1 or cell > 9:
                print("Please enter a number from 1 to 9.\n")
                continue
            r, c = cell_to_coords(cell)
            if board[r][c] != "-":
                print("That cell is already taken. Try again.\n")
                continue
            return r, c
        except ValueError:
            print("Please enter numbers only.\n")

# Main game loop
def play_game():
    board = create_board()
    current = "X" # Player X starts
    print("Tic Tac Toe — Two Players (1-9 version)\n")
    show_board(board)
```

```
    while True:
        r, c = get_move(board, current)
        board[r][c] = current
        show_board(board)

        if check_winner(board, current):
            print(f" Player {current} wins!")
            break
        if is_full(board):
            print("It's a draw!")
            break

        current = "O" if current == "X" else "X"

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

Output Snapshot

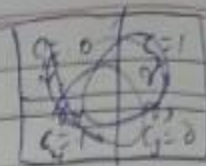
```
Tic Tac Toe — Two Players (1-9 version)

['-', '-', '-']
['-', '-', '-']
['-', '-', '-']
Player X, position: 1
['X', '-', '-']
['-', '-', '-']
['-', '-', '-']
Player O, position: 2
['X', 'O', '-']
['-', '-', '-']
['-', '-', '-']
Player X, position: 4
['X', 'O', '-']
['X', '-', '-']
['-', '-', '-']
Player O, position: 3
['X', 'O', 'O']
['X', '-', '-']
['-', '-', '-']
Player X, position: 4
That cell is already taken. Try again.

Player X, position: 7
['X', 'O', 'O']
['X', '-', '-']
['X', '-', '-']
 Player X wins!
```

Program 2 - Vacuum Cleaner

Algorithm

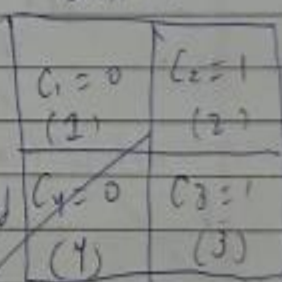
20-8-2025

Vaccum Cleaner,




```
Algorithm Vaccum cleaner (room){

    count = 0;
    pos = 1;
    while (count < 4) {
        if (c[pos] == 0){
room        clean room;
            print ("Room {pos} cleaned");
        }
        else {
            print ("Room already cleaned");
        }
        count ++;
        move to next pos;
    }
    print (" All rooms cleaned !");

function clean (n) {
    c[n] = 1;
}
```

Count = 0



O/P

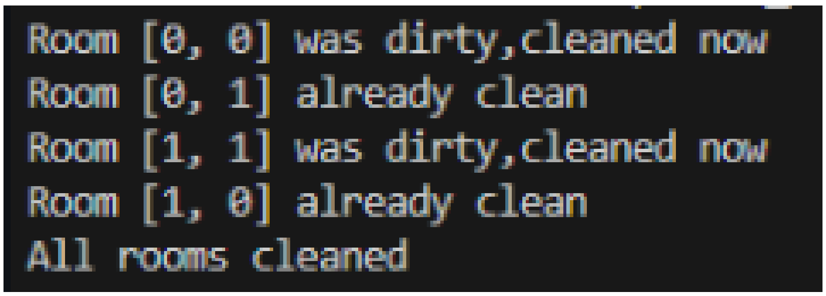
Room [0,0] was dirty, now cleaned
Room [0,1] already clean
Room [1,1] was dirty, cleaned now
Room [1,0] already cleaned
All rooms cleaned

20/8/25

Code

```
pos= [[0,0], [0,1], [1,1], [1,0]]
clean= [[0,1], [1,0]]

for i in pos:
    row=i[0]
    col=i[1]
    if(clean[row][col]):
        print(f"Room {i} already clean")
    else:
        print(f"Room {i} was dirty,cleaned now")
        clean[row][col]=1
print("All rooms cleaned")25
```

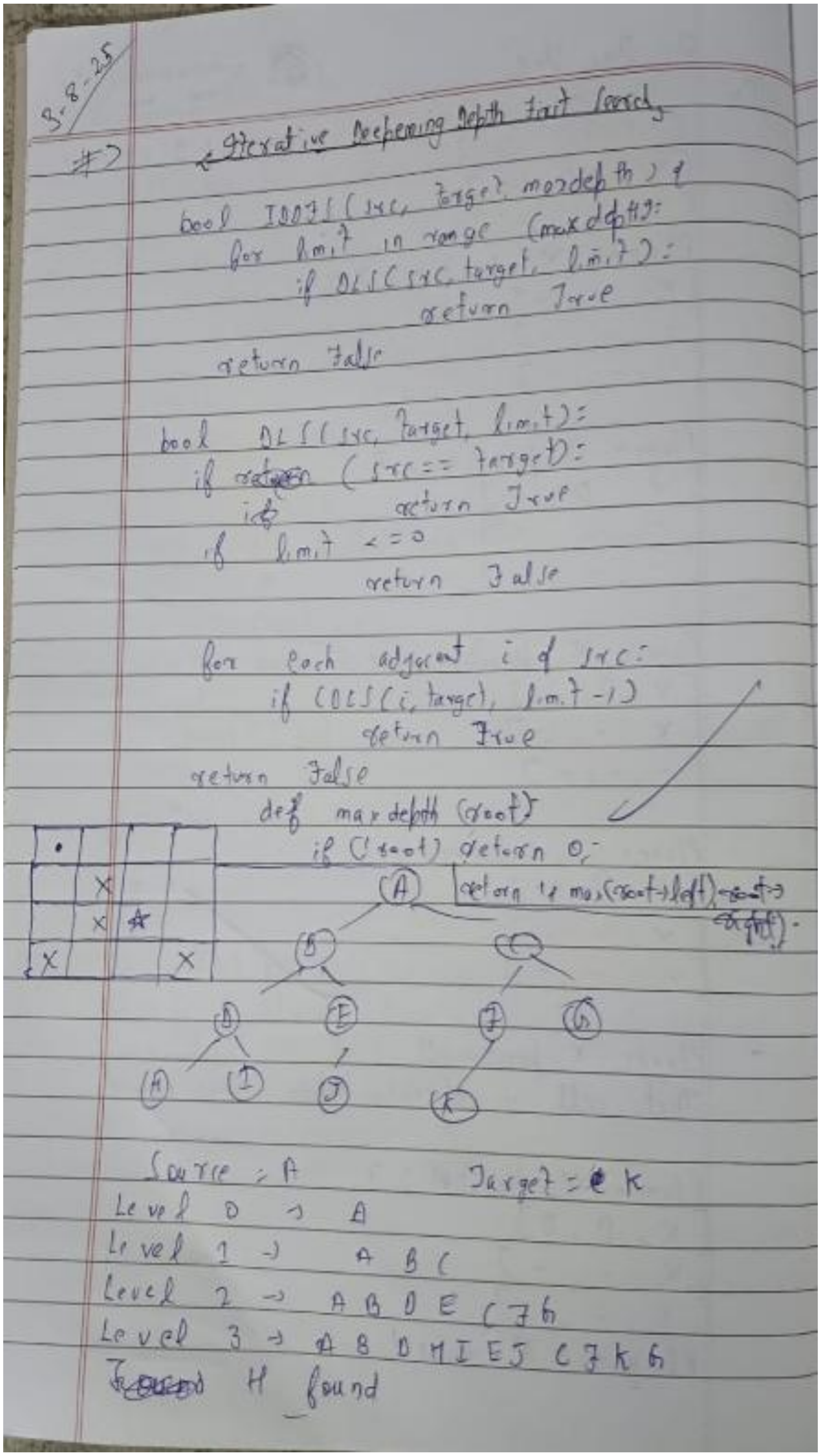
Output Snapshot



```
Room [0, 0] was dirty,cleaned now
Room [0, 1] already clean
Room [1, 1] was dirty,cleaned now
Room [1, 0] already clean
All rooms cleaned
```

program-03 IDDFS

Algorithm



3-8-25

#7 Iterative Deepening Depth First Search

bool IDDFS(src, target, maxdepth) {
for limit in range (maxdepth):
if DLS(src, target, limit):
return True
return False

bool DLS(src, target, limit):
if (src == target):
return True
if limit <= 0
return False

for each adjacent i of src:
if (DLS(i, target), limit-1)
return True
return False

def maxdepth (root)
if (!root) return 0;
return 1 + max(root->left, root->right).

Source : A   Target : K
Level 0 -> A
Level 1 -> A B C
Level 2 -> A B D E C F G
Level 3 -> A B D H I E J C F K G
H found

Code

```
class Node:
    def __init__(self, value):
```

```
        self.value = value
        self.children = []

# Build the tree
A = Node('A')
B = Node('B')
C = Node('C')
D = Node('D')
E = Node('E')
F = Node('F')
G = Node('G')
H = Node('H')

A.children = [B, C, D]
B.children = [E, F]
C.children = [G]
G.children = [H]

def dls(node, target, depth, visited_at_level):
    if node is None:
        return None
    visited_at_level.append(node.value)
    if node.value == target:
        return node
    if depth == 0:
        return None
    for child in node.children:
        found = dls(child, target, depth - 1, visited_at_level)
        if found:
            return found
    return None

def iddfs(root, target, max_depth):
    found_node = None
    for depth in range(max_depth + 1):
        visited_at_level = []
        dls(root, target, depth, visited_at_level)
        print(f"Depth {depth}: Visited nodes -> {', '.join(visited_at_level)}")
        if target in visited_at_level:
            found_node = target
            break
    if found_node:
        print(f"\nResult: Found '{target}' at depth {depth}")
    else:
        print(f"\nResult: '{target}' not found within depth {max_depth}")
```

```
# Example usage
target = 'H'
iddfs(A, target, max_depth=4)
```

Output Snapshot

```
Depth 0: Visited nodes -> A
Depth 1: Visited nodes -> A, B, C, D
Depth 2: Visited nodes -> A, B, E, F, C, G, D
Depth 3: Visited nodes -> A, B, E, F, C, G, H

Result: Found 'H' at depth 3
```

Program 04 - 8 Puzzle Using A*
Algorithm

8- Puzzle problem by Hueristic

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 2 & 3 \\ 1 & 0 & 4 & 6 \\ 2 & 7 & 5 & 8 \end{array}$$

```
goal_state = [[1, 2, 3],
              [4, 5, 6],
              [7, 8, 0]]
moves = [(1,0), (-1,0), (0,1), (0,-1)]

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

def manhattan(state):
    d = 0
    for i in range(3):
        for j in range(3):
            v = state[i][j]
            if v != 0:
                gx, gy = divmod(v-1, 3)
                d += abs(i-gx) + abs(j-gy)
    return d
                d = |row - goalrow| + |col - goal_col|;

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
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 6 \\ 7 & 5 & 8 \end{bmatrix} \xleftrightarrow{\text{End}} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$$

initial   Final

Step 1

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 0 & 6 \\ 7 & 5 & 8 \end{bmatrix}$$

Step 2

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 0 & 8 \end{bmatrix}$$

Step 3

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$$

3/7/3

10-9-25

Lab 3

8 Puzzle problem using A* method

1	2	3
4	5	6
7	8	_

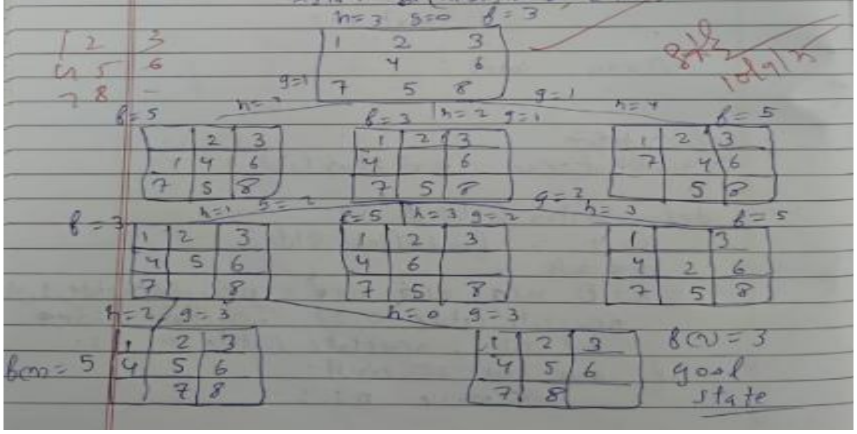
goal state

1	2	3
_	4	6
7	5	8

Start state

$$f(n) = g(n) + h(n)$$

```
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 ++;

    return count;

i, j = findblank(state)

def Astar(state, d):
    i, j = findblank(state)
    ns1, ns2, ns3, ns4 = newstate(state, 1, 0),
    newstate(state, 0, 1), newstate(state,
    -1, 0), newstate(state, 0, -1);
    if any ns == Null:
        remove ns;

gn = [d, d, d, d]
hn = [ for i in
misplaced_tiles(ns[i]) range(4)]

for (i) in range 4:
    if ns_i == NULL
        fn[i] = NULL

fn[i] = hn[i] + gn[i]

nextstate = ns[min(fn)]

if checkfinal(nextstate):
    print(final state reached)
else:
    Astar(nextstate, d+1)
```



Code

```
import heapq
goal_state = [[1, 2, 3],
              [4, 5, 6],
              [7, 8, 0]]
moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # up, down, left, right
def is_valid(x, y):
    return 0 <= x < 3 and 0 <= y < 3

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

def state_to_tuple(state):
    return tuple(tuple(row) for row in state)

# --------- Heuristics ---------
def manhattan_distance(state):
    d = 0
    for i in range(3):
        for j in range(3):
            v = state[i][j]
            if v != 0:
                gx, gy = divmod(v - 1, 3)
                d += abs(i - gx) + abs(j - gy)
    return d

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

# --------- A* Search ---------
def a_star(start, heuristic):
    visited = set()
    pq = []
    heapq.heappush(pq, (heuristic(start), 0, start, []))
    while pq:
        f, g, state, path = heapq.heappop(pq)
        if state == goal_state:
            return path + [state]
        visited.add(state_to_tuple(state))
        x, y = find_blank(state)
        for dx, dy in moves:
            nx, ny = x + dx, y + dy
            if is_valid(nx, ny):
                new_state = [row[:] for row in state]
```

```
                new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
                if state_to_tuple(new_state) not in visited:
                    g2 = g + 1
                    f2 = g2 + heuristic(new_state)
                    heapq.heappush(pq, (f2, g2, new_state, path + [state]))
    return None

# --------- Solver ---------
def solve(start, method="astar_manhattan"):
    if method == "astar_misplaced":
        return a_star(start, misplaced_tiles)
    else: # default: A* Manhattan
        return a_star(start, manhattan_distance)

# --------- Example Run ---------
start_state = [[1, 2, 3],
               [0, 4, 6],
               [7, 5, 8]]

for algo in ["astar_misplaced", "astar_manhattan"]:
    print("\nMethod:", algo)
    path = solve(start_state, algo)
    if path:
        print("Solved in", len(path) - 1, "moves")
        for step in path:
            for row in step: print(row)
            print("-----")
    else:
        print("No solution found")
```

Output Snapshot

```
Method: astar_misplaced
Solved in 3 moves
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
-----
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
-----
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
-----
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
-----

Method: astar_manhattan
Solved in 3 moves
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
-----
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
-----
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
-----
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
-----
```

Program-05 Hill Climbing

Algorithm

8-10-25

Hill Climbing - N Queens

			Q
	Q		
		Q	
Q			

Initial state
Queens at col 0 row 3,
col 1 row 1,
col 2 row 2,
col 3 row 0.

board = [3, 1, 2, 0]

Finally
A state where no queens intersect

cost function
|row diff| = |col diff| or $\boxed{\text{row 1} == \text{row2}}$

Check pairs
Q_1 (3,0) & Q_2 (1,1) → |3-1| ≠ |0-1|
3 ≠ 1 — count = 0

Q_1 & Q_3 → |3-2| ≠ |0-2|
3 ≠ 2 — count = 0

Q_1 & Q_4
|0-3| == |3-0| — count = 1

Q_2 & Q_3
|1-1| = |2-1| — count = 2

Q_2 & Q_4
|1-0| ≠ |3-1| &
1 ≠ 0 — count = 2

Q_3 & Q_4
|2-0| ≠ |2-3| & 2 ≠ 0 — count = 2

initial cost = 2

Algorithm
```
def random_state (N, max):
    for i in 1 to max:
        current ← random_board (n)
        while true:
            neighbours ← generate_neighbour (current)
            best_neighbour ← neighbour in neighbours
                                with min cost
            with min_conflict (neighbours)
                if evaluate (current) <=
                        evaluate ( best_neighbour)
                    break:

            else
                current ← best_neighbour
            if evaluate (current) == 0
                return current

    return failure.

def evaluate (state):
    int cost = 0
    for row, col in state:
        if (row_i == row_{i+1}     or
            (row_i - row_{i+1}) == (col_i - col_{i+1}))
                cost ++;

    return cost.
```

Code

```
import random

def conflicts(state):
    count = 0
    for i in range(4):
        for j in range(i + 1, 4):
            # same column or same diagonal
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                count += 1
    return count

def print_board(state):
    for i in range(4):
        row = ["Q" if state[i] == j else "." for j in range(4)]
        print(" ".join(row))
    print()

def hill_climbing():
    state = [random.randint(0, 3) for _ in range(4)]
    current_conflicts = conflicts(state)

    while True:
        best_state = state
        best_conflicts = current_conflicts

        # Try moving each queen to every other column
        for row in range(4):
            for col in range(4):
                if col == state[row]:
                    continue
                new_state = state.copy()
                new_state[row] = col
                new_conflicts = conflicts(new_state)

                if new_conflicts < best_conflicts:
                    best_conflicts = new_conflicts
                    best_state = new_state

        # If no improvement, stop
        print("State space now:")
        print_board(state)
        print("no of conflicts=",best_conflicts,"\n")
        if best_conflicts == current_conflicts:
            break

        # Update to new better state
        state = best_state
        current_conflicts = best_conflicts

        # Stop early if solved
        if current_conflicts == 0:
```

```
            break

    return state, current_conflicts

# --- Run the 4-Queens Hill Climbing ---
solution, c = hill_climbing()
print_board(solution)

if c == 0:
    print("Found a valid 4-Queens solution!")
else:
    print("Stuck in local minimum (", c, "conflicts )")
```

OUTPUT

```
State space now:
Q . . .
. . . Q
. . Q .
. . Q .

no of conflicts= 1

State space now:
Q . . .
. . . Q
Q . . .
. . Q .

no of conflicts= 0

. Q . .
. . . Q
Q . . .
. . Q .

Found a valid 4-Queens solution!
```

Program-06 Simulated Annealing

Algorithm

8-10-25

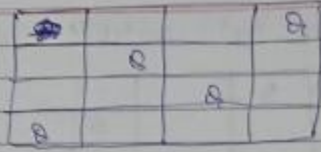
Simulated annealing
N-Queens

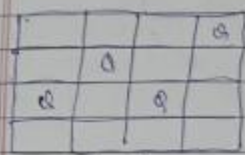
Formula: $P = e^{-\Delta E/kT}$

where
k = cooling factor
T = Temperature
ΔE = change cost
P = probability of change

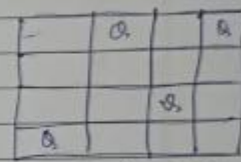
```
algorithm simulated-annealing(initial_state):
    current = initial_state
    T = initial_temp (=100)
    while T>0:
        neighbours = getNeighbours(current)
        if (neighbours == empty):
            return current

        for ne in neighbours:
            ΔE = cost(neighbours)
            if ΔE > 0 or random < e^(-ΔE/T):
                current = neighbour
    T = T * k
```

Initially cost = 2



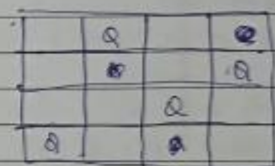


cost = 2



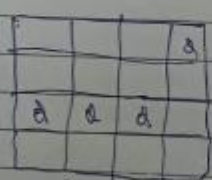
cost = 1

cost = 4

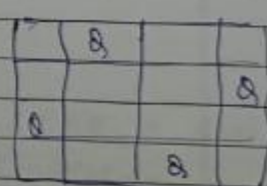


cost = 1

~~selecting best suitable~~

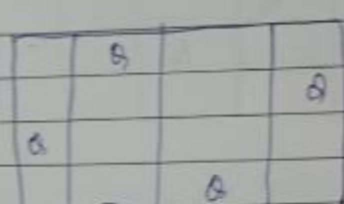


cost = 3



cost = 0

Selecting best suitable



8/10/25

Code

```
import random
import math

N = 4

def cost(state):
    """Compute number of attacking queen pairs."""
    conflicts = 0
    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):
                conflicts += 1
    return conflicts

def random_neighbor(state):
    """Generate a neighbor by moving one queen to another row."""
    neighbor = state.copy()
    col = random.randrange(N)
    new_row = random.randrange(N-1)
    if new_row >= neighbor[col]:
        new_row += 1
    neighbor[col] = new_row
    return neighbor

def simulated_annealing(
    T0=5.0, alpha=0.995, Tmin=1e-6, max_iters=50000
):
    # Start with a random placement of queens
    state = [random.randrange(N) for _ in range(N)]
    current_cost = cost(state)
    T = T0
    it = 0

    while T > Tmin and it < max_iters and current_cost != 0:
        neighbor = random_neighbor(state)
        neighbor_cost = cost(neighbor)
        delta = neighbor_cost - current_cost

        if delta <= 0 or random.random() < math.exp(-delta / T):
            state, current_cost = neighbor, neighbor_cost
            print(f"Iteration {it}, Cost {current_cost}, State: {state}")

        T *= alpha
```

```
        it += 1

    return state, current_cost

def print_board(state):
    """Pretty-print the board."""
    for r in range(N):
        row = ""
        for c in range(N):
            row += "Q " if state[c] == r else ". "
        print(row)
    print()

# Run SA for 4-queens
solution, c = simulated_annealing()

print("Final state (col -> row):", solution)
print("Cost:", c)
print("\nBoard:")
print_board(solution)
```

Output Snapshot

```
Iteration 0, Cost 2, State: [1, 3, 1, 2]
Iteration 1, Cost 4, State: [1, 1, 1, 2]
Iteration 2, Cost 3, State: [1, 1, 2, 2]
Iteration 3, Cost 5, State: [2, 1, 2, 2]
Iteration 4, Cost 4, State: [0, 1, 2, 2]
Iteration 5, Cost 3, State: [1, 1, 2, 2]
Iteration 6, Cost 2, State: [1, 3, 2, 2]
Iteration 7, Cost 0, State: [1, 3, 0, 2]
Final state (col -> row): [1, 3, 0, 2]
Cost: 0

Board:
. . Q .
Q . . .
. . . Q
. Q . .
```

Program-07 Entailment in prepositional logic

Algorithm

15-10-25

Propositional Logic

Truth Table for connectivities

P	Q	¬P	P ∧ Q	P ∨ Q	P ⇔ Q
F	F	T	F	F	T
F	T	T	F	T	F
T	F	F	F	T	F
T	T	F	T	T	T

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 proposition
            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 return true // when KB false, ret True
    else do
        P ← First(symbols)
        rest ← REST (symbols)
        return (TT-check-ALL (KB, α, rest,
            model ∪ {P = True})
        and TT-CHECK-ALL (KB, α, rest,
            model ∪ {P = False}))

Q. Consider a KB that contains the following propositional logic sentences:

$Q \to P$

$P \to \neg Q$

$Q \vee R$

i) Construct a truth table that shows the truth value of each sentence in KB & indicate the models in which the KB is true.

ii) Does KB entail R?

iii) Does KB entail $R \to P$?

iv) Does KB entail $Q \to R$?

Truth Table

$KB = (Q \to P) \wedge (P \to \neg Q) \wedge (Q \vee R)$

i)

P	Q	R	$Q \to P$	$P \to \neg Q$	$Q \vee R$	$R \to P$	$Q \to R$	KB=T
F	F	F	T	T	F	T	T	F
F	F	T	T	T	T	F	T	T
F	T	F	F	T	T	T	F	F
F	T	T	F	T	T	F	T	F
T	F	F	T	T	F	T	T	F
T	F	T	T	T	T	T	T	T
T	T	F	T	F	T	T	F	F
T	T	T	T	F	T	T	T	F

ii) Yes KB entail R

iii) No KB entail $R \to P$

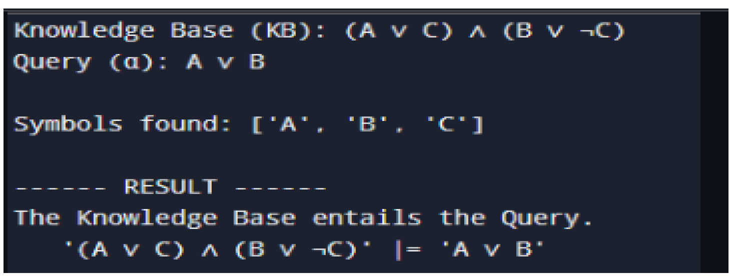
iv) Yes KB entails $Q \to R$

Code

```
import re
from itertools import product

def pl_true(sentence, model):
    try:
        return eval(sentence, model)
    except NameError:
        return False

def tt_entails(kb, alpha):
    kb = kb.replace('¬', 'not ').replace('∧', ' and ').replace('∨', ' or ')
    alpha = alpha.replace('¬', 'not ').replace('∧', ' and ').replace('∨', ' or ')
    symbols = sorted(list(set(re.findall(r'[A-Z]', kb + alpha))))
```

```
    print(f"Symbols found: {symbols}")
    for values in product([True, False], repeat=len(symbols)):
        model = dict(zip(symbols, values))
        if pl_true(kb, model):
            if not pl_true(alpha, model):
                print(f"Counterexample found: {model}")
                return False
    return True
if __name__ == "__main__":
    kb_formula = "(A ∨ C) ∧ (B ∨ ¬C)"
    alpha_formula = "A ∨ B"
    print(f"Knowledge Base (KB): {kb_formula}")
    print(f"Query (α): {alpha_formula}\n")
    result = tt_entails(kb_formula, alpha_formula)
    print("\n------ RESULT ------")
    if result:
        print(f"The Knowledge Base entails the Query.")
        print(f"   '{kb_formula}' |= '{alpha_formula}'")
    else:
        print(f"The Knowledge Base does NOT entail the Query.")
        print(f"   '{kb_formula}' |/= '{alpha_formula}'")
```

Output Snapshot



```
Knowledge Base (KB): (A v C) ∧ (B v ¬C)
Query (α): A v B

Symbols found: ['A', 'B', 'C']

------ RESULT ------
The Knowledge Base entails the Query.
   '(A v C) ∧ (B v ¬C)' |= 'A v B'
```

Program-8 Unification in FOL

Algorithm

23-10-25

First Order Logic

Unification : It deals with finding a common substitution for variables in different terms to make them match.

Unification Algorithm

unify (Ψ_1, Ψ_2)

1> If Ψ_1 or Ψ_2 is a variable then:
   a> If Ψ_1 or Ψ_2 are identical, return NIL
   b> Else if Ψ_1 is a variable
      i> then if Ψ_1 occurs in Ψ_2, return FAIL
      ii> else return {Ψ_2/Ψ_1}
   c> Else if Ψ_2 is a variable
      i> if Ψ_2 occurs in Ψ_1, return FAIL
      ii> else return {(Ψ_1/Ψ_2)}
   d> else return FAIL

2> If initial predicate in Ψ_1, Ψ_2 don't match, return FAIL

3> If Ψ_1 and Ψ_2 don't have identical no of elements, return FAIL

4> Set substitution set (SUBST) to NIL

5> for i=1 to no of elements in Ψ_1:
   a> Call unify with i^{th} ele of Ψ_1 & Ψ_2 & put result in S.
   b> If S = failure then returns FAIL
   c> If S ≠ NIL then do
      i> apply S to remainder of both L_1 & L_2
      ii> SUBST = APPEND(S, SUBST)

return SUBST

Unification examples

1> $P(f(x), g(y), y)$, $P(f(g(z)), g(f(a)), f(a))$

2> $Q(x, f(x))$, $Q(f(y), y)$

3> $H(x, g(x))$, $H(g(y), g(g(z)))$

1> $\{g(z) / x\}$

$P(f(x), g(y), y)$, $P(f(x), g(f(a)), f(a))$

1> $\{g(z)/x\}$

$P(f(x), g(y), y)$, $P(f(x), g(f(a)), f(a))$

$\{f(a)/y\}$

$P(f(x), g(y), y)$, $P(f(x), g(y), y)$

$\theta = \{g(z)/x, f(a)/y\}$

2> $Q(x, f(x))$

1> $\{x / g(z)\}$

$P(f(g(z)), g(y), y)$, $P(f(g(z)), g(f(a)), f(a))$

$\{y / f(a)\}$

$P(f(g(z)), g(f(a)), f(a))$, $P(f(g(z)), g(f(a)), f(a))$

$\theta = \{x / g(z), y / f(a)\}$

2> $x / f(y)$

$\theta(f(y), f(f(y)))$, $\theta(f(y), y)$

Fails

3> $H(x, g(x))$, $H(g(y), g(g(z)))$

$\{x / g(y)\}$

$H(g(y), g(g(y)))$, $H(g(y), g(g(z)))$

$\{y / z\}$

$H(g(y), g(g(z)))$, $H(g(y), g(g(z)))$

$\theta = \{x / g(y), y/(z)\}$

30/10/23

Code

```
import re

def is_variable(x):
    return x[0].islower() and x.isalpha()

def parse(term):
    term = term.strip()
    if '(' not in term:
        return term
    name, args = term.split('(', 1)
    args = args[:-1]  # remove closing parenthesis
    return name.strip(), [parse(a.strip()) for a in args.split(',')]

def occurs_check(var, expr):
    if var == expr:
        return True
    if isinstance(expr, tuple):
        _, args = expr
        return any(occurs_check(var, a) for a in args)
    return False

def substitute(subs, expr):
    if isinstance(expr, str):
        if expr in subs:
            return substitute(subs, subs[expr])
        return expr
    else:
        func, args = expr
        return (func, [substitute(subs, a) for a in args])

def unify(x, y, subs=None):
    if subs is None:
        subs = {}

    x = substitute(subs, x)
    y = substitute(subs, y)

    if x == y:
        return subs

    if isinstance(x, str) and is_variable(x):
        if occurs_check(x, y):
```

```
            return None
        subs[x] = y
        return subs

    if isinstance(y, str) and is_variable(y):
        if occurs_check(y, x):
            return None
        subs[y] = x
        return subs

    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]):
            subs = unify(a, b, subs)
            if subs is None:
                return None
        return subs

    return None


def term_to_str(t):
    if isinstance(t, str):
        return t
    func, args = t
    return f"{func}({', '.join(term_to_str(a) for a in args)})"

def pretty_print(subs):
    return ', '.join(f"{v} : {term_to_str(t)}" for v, t in subs.items())


pairs = [
    ("P(f(x),g(y),y)", "P(f(g(z)),g(f(a)),f(a))"),
    ("Q(x,f(x))", "Q(f(y),y)"),
    ("H(x,g(x))", "H(g(y),g(g(z)))")
]

for s1, s2 in pairs:
    print(f"\nUnifying: {s1}  and  {s2}")
    result = unify(parse(s1), parse(s2))
    if result:
        print("=> Substitution:", pretty_print(result))
    else:
```

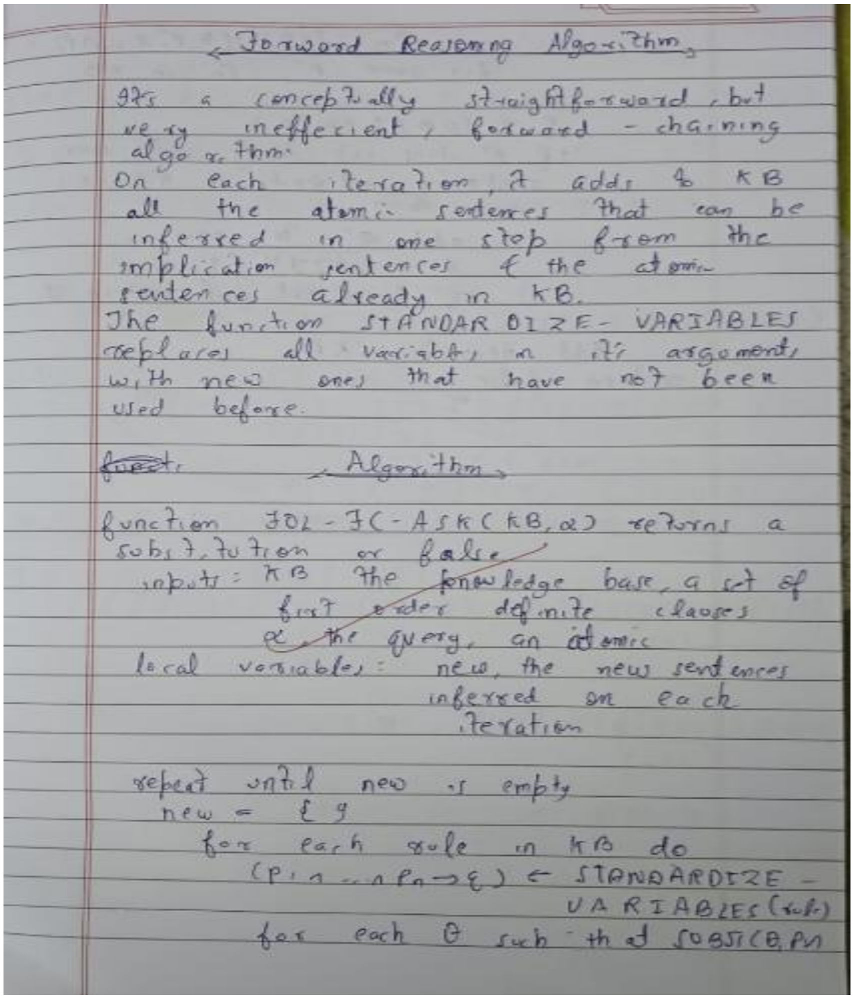
```
print("=> Not unifiable.")
```

Output Snapshot

```
Unifying: P(f(x),g(y),y)  and  P(f(g(z)),g(f(a)),f(a))
=> Substitution: x : g(z), y : f(a)

Unifying: Q(x,f(x))  and  Q(f(y),y)
=> Not unifiable.

Unifying: H(x,g(x))  and  H(g(y),g(g(z)))
=> Substitution: x : g(y), y : z
```

Program-9 Forward Reasoning in FOL

Algorithm



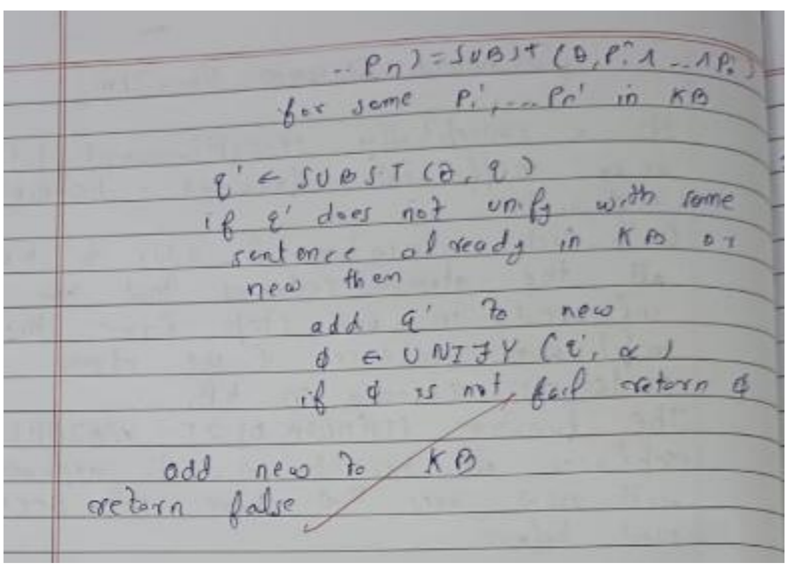
Forward Reasoning Algorithm

It's a conceptually straightforward, but very inefficient, forward-chaining algorithm.
On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences & the atomic sentences already in KB.
The function STANDARDIZE-VARIABLES replaces all variables in its arguments with new ones that have not been used before.

~~funct.~~ Algorithm

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

repeat until new is empty
new = { }
for each rule in KB do
(P₁ ∧ ... ∧ Pn → Q) ← STANDARDIZE-VARIABLES(rule)
for each θ such that SUBST(θ, P₁



... Pn) = SUBST(θ, P₁' ∧ ... ∧ Pn')
for some P₁', ... Pn' in KB

q' ← SUBST(θ, q)
if q' does not unify with some sentence already in KB or new then
add q' to new
φ ← UNIFY(q', α)
if φ is not fail return φ

add new to KB
return false

Code

from copy import deepcopy

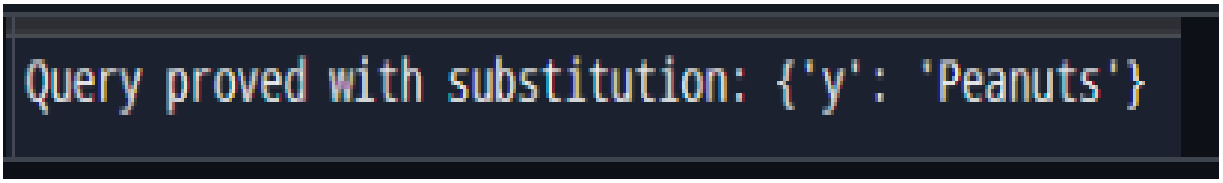
```
def standardize_variables(rule, var_count):
    """
    Standardize variables by renaming them with a unique suffix to avoid clashes.
    """
    new_rule = []
    for literal in rule:
        new_literal = []
        for token in literal:
            if token.islower() and token.isalpha():
                new_literal.append(token + str(var_count))
            else:
                new_literal.append(token)
        new_rule.append(tuple(new_literal))
    return new_rule

def unify(x, y, subst={}):
    """
    Unify two literals (tuples) x and y with current substitution subst.
    Returns updated substitution or None if cannot unify.
    """
    if subst is None:
        return None
    elif x == y:
        return subst
    elif isinstance(x, str) and x.islower():
        return unify_var(x, y, subst)
    elif isinstance(y, str) and y.islower():
        return unify_var(y, x, subst)
    elif isinstance(x, tuple) and isinstance(y, tuple) and len(x) == len(y):
        for a, b in zip(x, y):
            subst = unify(a, b, subst)
        return subst
    else:
        return None

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

def substitute(sentence, subst):
```

```
    """
    Apply substitution subst to a sentence (list of literals).
    """
    new_sentence = []
    for literal in sentence:
        new_literal = []
        for token in literal:
            if token in subst:
                new_literal.append(subst[token])
            else:
                new_literal.append(token)
        new_sentence.append(tuple(new_literal))
    return new_sentence

def fol_fc_ask(KB, alpha):
    """
    Forward chaining algorithm for First Order Logic.

    KB: list of rules in the form [ (premises), conclusion ]
    alpha: query, atomic sentence as a tuple
    """

    var_count = 0
    new = []
    while True:
        new = []
        for (premises, conclusion) in KB:
            # Standardize variables in rule
            var_count += 1
            standardized_premises = standardize_variables(premises, var_count)
            standardized_conclusion = standardize_variables([conclusion], var_count)[0]

            # Try to unify each premise with some sentence in KB or new
            for fact in KB:
                subst = {}
                for premise in standardized_premises:
                    subst = unify(premise, fact[1], subst)
                    if subst is None:
                        break
                if subst is not None:
                    # Apply substitution
                    inferred = substitute([standardized_conclusion], subst)[0]
                    if inferred not in KB and inferred not in new:
                        new.append(inferred)
                    # Check if query is proven
```

```
            subst_alpha = unify(alpha, inferred, {})
            if subst_alpha is not None:
                return subst_alpha

    if not new:
        return False
    KB.extend([((), fact) if isinstance(fact, tuple) else fact for fact in new])

# Example usage:
# Represent atomic sentences as tuples, e.g. Likes(John, Food) -> ("Likes", "John", "Food")

if __name__ == "__main__":
    # KB: List of rules: (premises, conclusion)
    # premises and conclusion are list/tuple of literals (predicates as tuples)

    KB = [
        # John likes all food: Food(x) => Likes(John, x)
        ( [("Food", "x")], ("Likes", "John", "x") ),

        # Apple and Vegetable are food
        ( [], ("Food", "Apple") ),
        ( [], ("Food", "Vegetable") ),

        # Anything eaten and not killed is food: Eats(x,y) ^ ¬Killed(x) => Food(y)
        ( [("Eats", "x", "y"), ("NotKilled", "x")], ("Food", "y") ),

        # Anil eats peanuts and not killed
        ( [], ("Eats", "Anil", "Peanuts") ),
        ( [], ("NotKilled", "Anil") ),

        # Harry eats everything Anil eats: Eats(Anil,y) => Eats(Harry,y)
        ( [("Eats", "Anil", "y")], ("Eats", "Harry", "y") ),

        # Alive implies not killed: Alive(x) => NotKilled(x)
        ( [("Alive", "x")], ("NotKilled", "x") ),

        # Not killed implies alive: NotKilled(x) => Alive(x)
        ( [("NotKilled", "x")], ("Alive", "x") )
    ]

    # Query: Likes(John, Peanuts)
    query = ("Likes", "John", "Peanuts")

    result = fol_fc_ask(KB, query)
    if result:
```

```
        print("Query proved with substitution:", result)
    else:
        print("Query cannot be proved.")
```

OutputSnapshot



Program-10 Resolution in FOL.

Algorithm

12-11-25

Resolution in FOL

Algorithm FOL-Resolution (KB, Q)

Input :
KB → Knowledge Base (a set of FOL sentences)
Q – Query (sentence to be proven)

Output :
True if KB ⊨ Q
False otherwise

→ Convert all sentences in KB into CNF :
- Eliminate ⇔ & →
- Move ¬ inwards
- Standardize variables apart by renaming them each quantifier should use a different variable
- Skolemize each existential variable
- ~~Drop universal quantifier~~
- Distribute ∧ over ∨

→ Negate Query & to ¬Q & add to KB
→ Initialize Resolvents {}
→ Repeat until contradiction is found or no new clause :
a.) Select two clauses C1 & C2 from KB.
b) Resolve them together, performing all required unifications
c) If resolvent is empty clause, a contradiction has been found
d) If not, add the resolvent to the premises

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

O/P

Initial Clauses :
- Food (x) ∨ Likes (John, x)
Food (Apple)
Food (Vegetable)
- Eats (x, y) ∨ ¬ Killed (x) ∨ Food (y)
- Eats (Anil, Peanuts)
- Killed (Anil)
- Eats (x, y) ∨ Eats (Harry, y)
- Alive (x) ∨ - Killed (x)
- - Killed (x) ∨ Alive (x)
- Likes (John, Peanuts)

Resolution steps

Contradiction found between - Killed (x) and - - Killed (x)
So query is proved.

Code

```
from itertools import product

# Helper function: check if a literal is negated
def is_negative(literal):
    return literal.startswith("¬")

# Negate a literal
def negate(literal):
    return literal[1:] if is_negative(literal) else "¬" + literal

# Unification function (basic variable substitution)
def unify(x, y, substitution={}):
    if substitution is None:
        return None
    elif x == y:
        return substitution
    elif isinstance(x, str) and x.islower():  # variable
        return unify_var(x, y, substitution)
    elif isinstance(y, str) and y.islower():
        return unify_var(y, x, substitution)
    elif isinstance(x, tuple) and isinstance(y, tuple) and len(x) == len(y):
        for a, b in zip(x, y):
            substitution = unify(a, b, substitution)
        return substitution
    else:
        return None

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

# Resolution algorithm
def resolution(kb, query):
```

```
    clauses = kb + [negate(query)]
    new = set()

    print("Initial clauses:")
    for c in clauses:
        print(" -", c)
    print("\nResolution steps:")

    while True:
        pairs = [(clauses[i], clauses[j]) for i in range(len(clauses)) for j in range(i + 1, len(clauses))]
        for (ci, cj) in pairs:
            resolvents = resolve(ci, cj)
            if "" in resolvents:
                print(f"\nContradiction found between {ci} and {cj}")
                print(" Therefore, the query is PROVED true.")
                return True
            new = new.union(resolvents)

        if new.issubset(set(clauses)):
            print("\nNo new clauses can be added.")
            print( Query cannot be proved.")
            return False
        for c in new:
            if c not in clauses:
                clauses.append(c)

# Resolve two clauses
def resolve(ci, cj):
    resolvents = set()
    ci_literals = ci.split(" ∨ ")
    cj_literals = cj.split(" ∨ ")

    for di in ci_literals:
        for dj in cj_literals:
            if di == negate(dj):
                new_clause = list(set(ci_literals + cj_literals))
                new_clause.remove(di)
                new_clause.remove(dj)
                resolvent = " ∨ ".join(sorted(set(new_clause)))
```

```
            resolvents.add(resolvent)
    return resolvents

if __name__ == "__main__":
    KB = [
        "¬Food(x) ∨ Likes(John, x)",                # John likes all food
        "Food(Apple)",                              # Apple is food
        "Food(Vegetable)",                          # Vegetable is food
        "¬Eats(x, y) ∨ ¬¬Killed(x) ∨ Food(y)",      # Anything anyone eats and not killed is food
        "Eats(Anil, Peanuts)",                      # Anil eats peanuts
        "¬Killed(Anil)",                            # Anil is still alive
        "¬Eats(x, y) ∨ Eats(Harry, y)",             # Harry eats everything Anil eats
        "¬Alive(x) ∨ ¬Killed(x)",                   # Alive ⇒ not killed
        "¬¬Killed(x) ∨ Alive(x)"                    # not killed ⇒ alive
    ]
    QUERY = "Likes(John, Peanuts)"
    resolution(KB, QUERY)
```

<u>OUTPUT:</u>

```
Initial clauses:
 - ¬Food(x) v Likes(John, x)
 - Food(Apple)
 - Food(Vegetable)
 - ¬Eats(x, y) v ¬¬Killed(x) v Food(y)
 - Eats(Anil, Peanuts)
 - ¬Killed(Anil)
 - ¬Eats(x, y) v Eats(Harry, y)
 - ¬Alive(x) v ¬Killed(x)
 - ¬¬Killed(x) v Alive(x)
 - ¬Likes(John, Peanuts)

Resolution steps:

Contradiction found between ¬Killed(x) v ¬¬Killed(x) and ¬¬Killed(x)
Therefore, the query is PROVED true.
```

Program-11 Alpha-Beta Pruning

Algorithm

12-11-25

Alpha Beta Pruning,

function Alpha-Beta Search (state) returns an action

$v \leftarrow$ MAX-VALUE (state, $-\infty$, $+\infty$)
return an action in ACTIONS (state) with value v

function MAX-VALUE (state, α, β) returns a utility value
if leaf-Node (state)
return utility (state)

$v \leftarrow -\infty$
for each a in ACTIONS (state) do
$v \leftarrow$ MAX (v, MIN-VALUE (RESULT (S, a), α, β))

if $v \geq \beta$ return v

$\alpha \leftarrow$ MAX (α, v)

return v

function MIN-VALUE (state, α, β) returns a utility value
if leaf (state) then
return utility (state)

$v \leftarrow +\infty$

for each a in Actions (state) do
$v \leftarrow$ MIN (v, MA-VALUE (RESULT (s, a), α, β))

if $v \leq \alpha$ then return v
$\beta \leftarrow$ MIN (β, v)

return v

o/p Initially
User → X AI → O

```
1 | 2 | 3
4 | 5 | 6
7 | 8 | 9
```

Enter move : 1 AI moves

```
x |   |
  | o |
  |   |
```

Enter your move : 9 AI moves

```
x | o |
  | o |
  |   | x
```

Enter your move : 3 AI moves

```
x | o | x
  | o |
  | o | x
```

O wins ;

Code

```
import math

# Initialize board
board = [" " for _ in range(9)]

# Function to print the board
def print_board(board):
    print()
    for i in range(3):
        print(" | ".join(board[i*3:(i+1)*3]))
        if i < 2:
            print("---------")
    print()

# Check winner or draw
def check_winner(board):
    win_positions = [
        (0, 1, 2), (3, 4, 5), (6, 7, 8),
        (0, 3, 6), (1, 4, 7), (2, 5, 8),
        (0, 4, 8), (2, 4, 6)
    ]
    for (x, y, z) in win_positions:
        if board[x] == board[y] == board[z] and board[x] != " ":
            return board[x]
    if " " not in board:
        return "Draw"
    return None

def available_moves(board):
    return [i for i in range(9) if board[i] == " "]

# Alpha-Beta Pruning (Minimax)
def minimax(board, is_maximizing, alpha, beta):
    winner = check_winner(board)
    if winner == "O":
        return 1
    elif winner == "X":
        return -1
    elif winner == "Draw":
        return 0

    if is_maximizing:
        best_val = -math.inf
        for move in available_moves(board):
            board[move] = "O"
            val = minimax(board, False, alpha, beta)
            board[move] = " "
            best_val = max(best_val, val)
            alpha = max(alpha, val)
            if beta <= alpha:
                break
```

```
        return best_val
    else:
        best_val = math.inf
        for move in available_moves(board):
            board[move] = "X"
            val = minimax(board, True, alpha, beta)
            board[move] = " "
            best_val = min(best_val, val)
            beta = min(beta, val)
            if beta <= alpha:
                break
        return best_val

# AI selects best move
def best_move(board):
    best_val = -math.inf
    move = None
    for i in available_moves(board):
        board[i] = "O"
        move_val = minimax(board, False, -math.inf, math.inf)
        board[i] = " "
        if move_val > best_val:
            best_val = move_val
            move = i
    return move

# Game loop
def play_game():
    print("Welcome to Tic-Tac-Toe! You are X, and AI is O.")
    print_board(board)

    while True:
        # Human move
        try:
            human_move = int(input("Enter your move (1-9): ")) - 1
            if human_move not in range(9):
                print("Invalid input! Choose 1–9.")
                continue
            if board[human_move] != " ":
                print("That spot is taken! Try again.")
                continue
        except ValueError:
            print("Enter a number between 1–9.")
            continue

        board[human_move] = "X"

        # Check if player wins before AI moves
        if check_winner(board):
            break

        # AI move
```

```
        ai_move = best_move(board)
        board[ai_move] = "O"

        # Print board only after both have played
        print_board(board)

        # Check winner after AI move
        if check_winner(board):
            break

    winner = check_winner(board)
    if winner == "Draw":
        print("It's a draw!")
    else:
        print(f"{winner} wins!")
if __name__ == "__main__":
    play_game()
```

OUTPUT:

```
Welcome to Tic-Tac-Toe! You are X, and AI is O.

  |   |
---------
  |   |
---------
  |   |

Enter your move (1-9): 1

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

Enter your move (1-9): 9

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

Enter your move (1-9): 3

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

Enter your move (1-9): 4

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

O wins!
```