

# VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



## LAB REPORT on

### Artificial Intelligence (23CS5PCAIN)

*Submitted by*

B Vatsal

(1BM23CS061)

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



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

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



**CERTIFICATE**

This is to certify that the Lab work entitled “Artificial Intelligence (23CS5PCAIN)” carried out by **Name(1BM23CS)**, who is a 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.

Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE
---	--

## Index

<b>Sl. No.</b>	<b>Date</b>	<b>Experiment Title</b>	<b>Page No.</b>
1	20-8-2025	Implement Tic –Tac –Toe Game Implement vacuum cleaner agent	4-8
2	28-8-2025	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	9-13
3	3-9-2025	Implement A* search algorithm	14-18
4	10-9-2025	Implement Hill Climbing search algorithm to solve N-Queens problem	19-21
5	17-9-2025	Simulated Annealing to Solve 8-Queens problem	22-24
6	24-9-2025	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	25-27
7	8-10-2025	Implement unification in first order logic	28-32
8	15-10-2025	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	33-37
9	29-10-2025	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	38-43
10	12-11-2025	Implement Alpha-Beta Pruning.	44-46

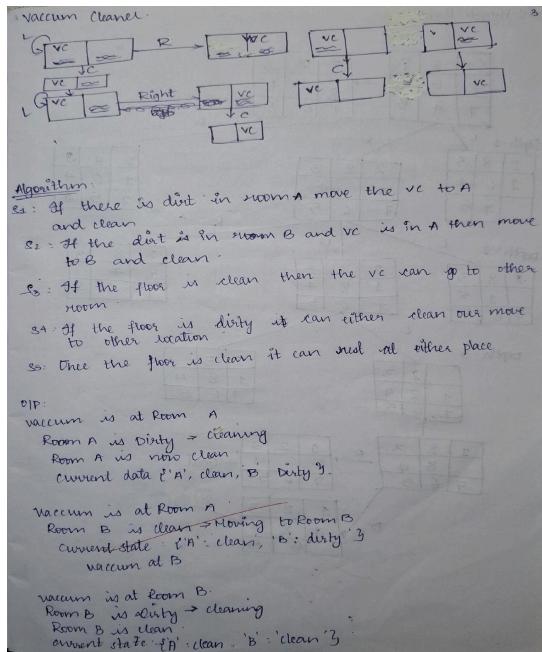
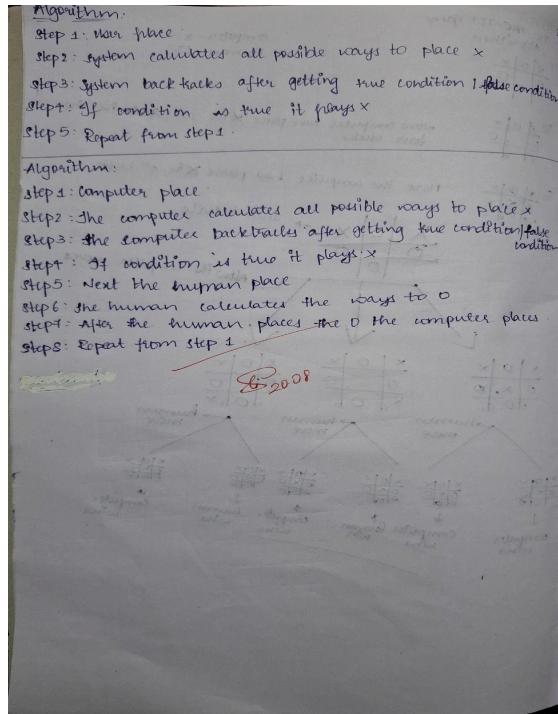
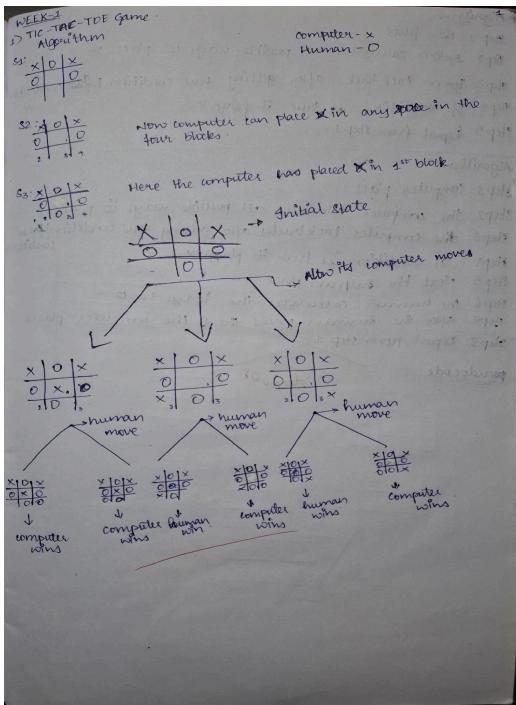
**Github Link:**

[https://github.com/Vatsalshetty/AI\\_LAB](https://github.com/Vatsalshetty/AI_LAB)

## PROGRAM 1:

Implement Tic – Tac – Toe Game  
Implement vacuum cleaner agent

## ALGORITHM:



## CODE:#T

icTacToe

import math

def check\_win(board)

wins = [(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 a,b,c in wins:

    if board[a] == board[b] == board[c] != 0:

        return board[a]

return 0 if 0 not in board else None

def selection(board, player):

    result = check\_win(board)

    if result is not None: return result

    scores = []

    for i in range(9): if

        board[i] == 0:

            board[i] = player

            scores.append(selection(board,

                -player)) board[i] = 0

    return max(scores) if player == 1 else min(scores)

def best\_move(board):

    return max((selection(board[:i]+[1]+board[i+1:], -1), i)

        for i in range(9) if board[i]==0)[1]

def print\_board(board):

    s = {1:'X', -1:'O', 0:' '}

    for i in range(0,9,3):

        print(f" {s[board[i]]}|{s[board[i+1]]}|{s[board[i+2]]}\n")

    ) if i<6: print(" ")

board=[0]\*9; player=1

while True:

    print\_board(board)

    res = check\_win(board)

    if res is not None:

        print("X wins!" if res==1 else "O wins!" if res== -1 else "Tie!")

        break

    if player==1:

        print("AI thinking. ")

```
    board[best_move(board)] = 1
else:
    move=int(input("Move (0-8): "))
    if 0<=move<=8 and board[move]==0: board[move]=-1
        else: print("Invalid"); continue
player*=-1
```

```
#Vaccum Cleaner:  
def isclean(list,place):  
    if(list[place]==1):  
        list[place] = 0  
    return False  
    else:  
        return True  
  
list = [1,1]  
place = 0  
count = 2  
  
while(count!=0):  
    if(place==0):  
        if(isclean(list,place)):  
            print("Location A is already cleaned")  
        else:  
            print("Location A is not clean")  
            print("cleaning Location A")  
            print("Locatin A has been cleaned")  
  
    count = count-1  
    place = 1  
  
    if(isclean(list,place)):  
        print("Location B is already cleaned")  
    else:  
        print("Location B is not clean")  
        print("cleaning Location B")  
        print("Locatin B has been cleaned")  
    count = count-1
```

OUTPUT:

```
0 | 1 | 2
---+---+
3 | 4 | 5
---+---+
6 | 7 | 8
AI thinking...
0 | 1 | 2
---+---+
3 | 4 | 5
---+---+
6 | 7 | X
Your move (0-8): 2
0 | 1 | 0
---+---+
3 | 4 | 5
---+---+
6 | 7 | X
AI thinking...
0 | 1 | 0
---+---+
3 | 4 | 5
---+---+
6 | X | X
Your move (0-8): 6
0 | 1 | 0
---+---+
3 | 4 | 5
---+---+
0 | X | X
AI thinking...
0 | 1 | 0
---+---+
3 | X | 5
---+---+
0 | X | X
Your move (0-8): 0
0 | 1 | 0
---+---+
3 | X | 5
```

```
Your move (0-8): 0
```

```
0 | 1 | 0
```

```
---+---+
```

```
3 | X | 5
```

```
---+---+
```

```
0 | X | X
```

```
AI thinking...
```

```
0 | X | 0
```

```
---+---+
```

```
3 | X | 5
```

```
---+---+
```

```
0 | X | X
```

```
X wins!
```

```
Location A is not clean
```

```
Cleaning Location A
```

```
Location A has been cleaned
```

```
Location B is not clean
```

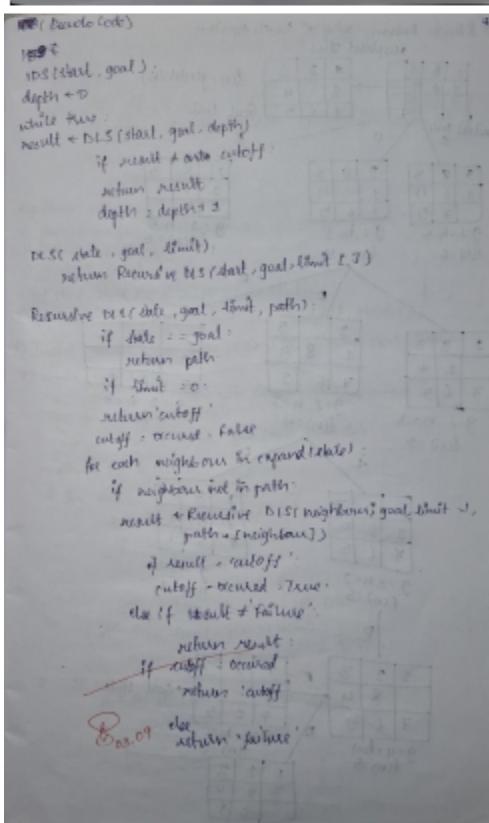
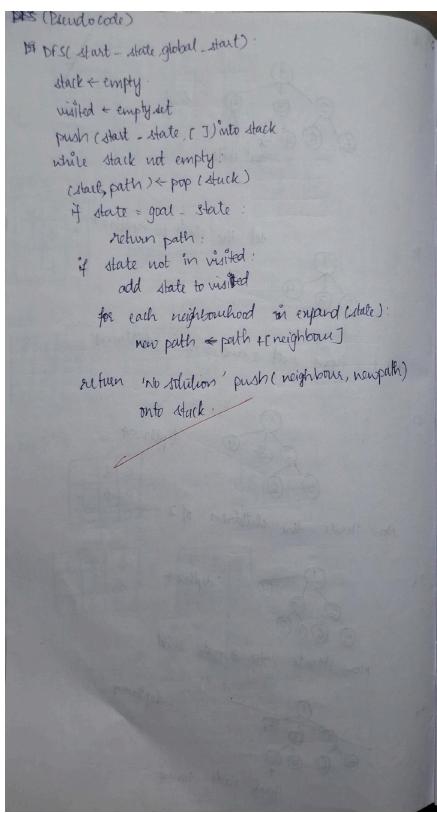
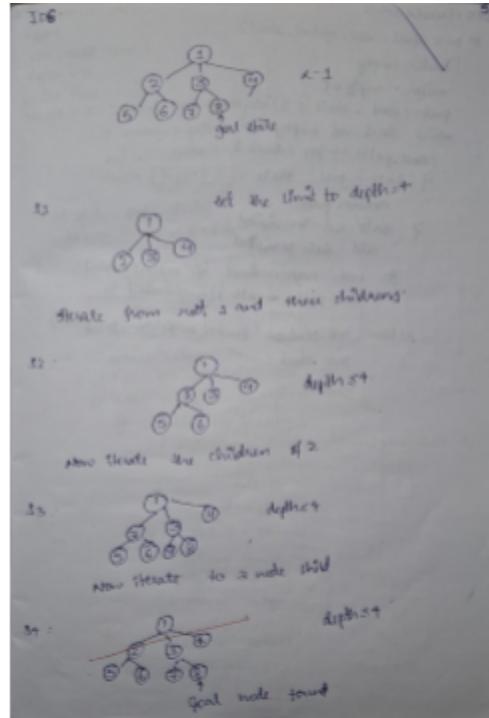
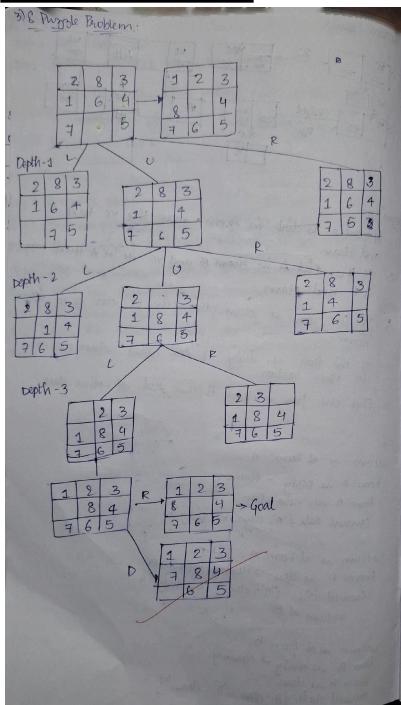
```
Cleaning Location B
```

```
Location B has been cleaned
```

## PROGRAM 2:

Implement 8 puzzle problems using Depth First Search (DFS)  
 Implement Iterative deepening search algorithm

## ALGORITHM:



## CODE:

```
goal = [[1,2,3],  
        [8,0,4],  
        [7,6,5]]  
  
moves = [(-1,0),(1,0),(0,-1),(0,1)]  
  
def find_zero(s):  
    for i in  
        range(3):  
            for j in range(3):  
                if s[i][j] == 0:  
                    return i, j  
  
def get_neighbors(s):  
    x, y = find_zero(s)  
    out = []  
    for dx, dy in moves:  
        nx, ny = x+dx,  
                 y+dy  
        if 0 <= nx < 3 and 0 <= ny < 3:  
            t = [r[:] for r in s]  
            t[x][y], t[nx][ny] = t[nx][ny],  
            t[x][y]  
            out.append(t)  
    return out  
  
def dfs(s, visited):  
    if s == goal:  
        return [s]  
    visited.add(str(s))  
    for nxt in get_neighbors(s):  
        if str(nxt) not in visited:  
            p = dfs(nxt, visited)  
            if p:  
                return [s] + p  
    return None  
  
def dls(s, goal, depth, path, visited):  
    if s == goal:  
        return path  
    if depth == 0:  
        return None  
    visited.add(str(s))
```

```
for nxt in get_neighbors(s):
    if str(nxt) not in visited:
        res = dls(nxt, goal, depth-1, path+[nxt], visited)
        if res:
            return res
return None
```

```
def ids(start, goal, limit=20):
    for d in range(limit+1):
        visited = set()
        r = dls(start, goal, d, [start], visited)
        if r:
            return r
    return None
```

```
if name == "main":
```

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

```
print("DFS:")
```

```
sol1 = dfs(start, set())
if sol1:
```

```
    print("Moves:", len(sol1)-1)
```

```
    for st in sol1:
```

```
        for r in st:
```

```
            print(r)
```

```
            print("-"*40)
```

```
else:
```

```
    print("No solution.")
```

```
print("\nIDS:")
```

```
sol2 = ids(start, goal, 20)
```

```
if sol2:  
    print("Moves:", len(sol2)-1)  
    for st in sol2:  
        for r in st:  
            print(r)  
            print("-"*40)  
  
else:  
    print("No solution.")
```

## Output:

```

DFS:
Moves: 28
[1, 2, 3]
[8, 6, 4]
[0, 7, 5]
-----
[1, 2, 3]
[0, 6, 4]
[8, 7, 5]
-----
[0, 2, 3]
[1, 6, 4]
[8, 7, 5]
-----
[2, 0, 3]
[1, 6, 4]
[8, 7, 5]
-----
[2, 6, 3]
[1, 0, 4]
[8, 7, 5]
-----
[2, 6, 3]
[1, 7, 4]
[8, 0, 5]
-----
[2, 6, 3]
[1, 7, 4]
[8, 0, 5]
-----
[2, 6, 3]
[1, 7, 4]
[8, 0, 5]

```

```

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

```

```

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

```

```

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

```

```

BS:
Moves: 2
[1, 2, 3]
[0, 6, 4]
[0, 7, 5]

```

```

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

```

```

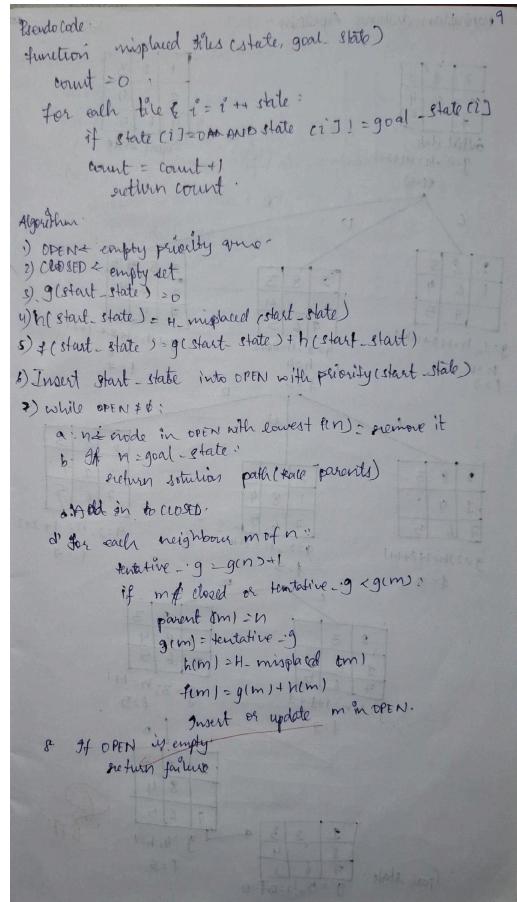
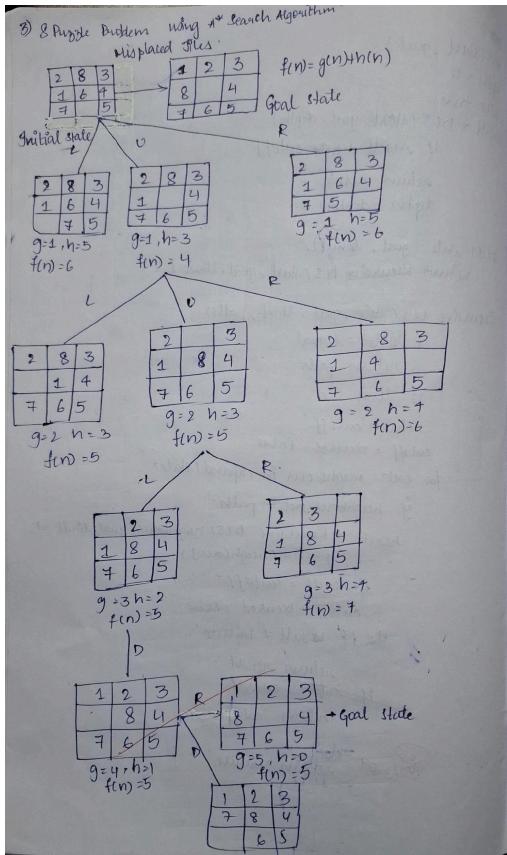
[1, 2, 3]
[0, 6, 4]
[7, 0, 5]

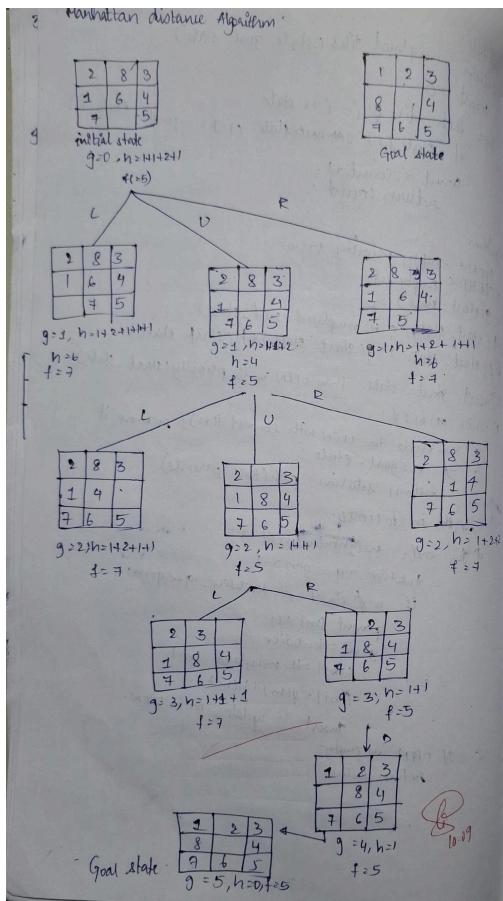
```

## PROGRAM 3:

Implement A\* search algorithm

## ALGORITHM:





Pseudo Code

```
function manhattan_distance(state, goal_state)
    distance = 0
    for file + 1 to 6:
        (x1,y1) = position_of(state, state)
        (x2,y2) = position_of(file, goal_state)
        distance = distance + (x1 - x2) + (y1 - y2)
```

return distance

States explored:

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

Final state: [1, 2, 3; 8, 4, 5; 7, 6, 5]

## CODE:

```
import heapq
class PuzzleState:
    def __init__(self, board, parent=None, move="", depth=0, cost=0):
        self.board = board
        self.parent = parent
        self.move = move
        self.depth = depth
        self.cost = cost

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

    def blank_pos(self):
        return self.board.index(0)

    def expand(self):
        b = self.blank_pos()
        row, col = divmod(b, 3)
        dirs = {
            "Up": (row - 1, col),
            "Down": (row + 1, col),
            "Left": (row, col - 1),
            "Right": (row, col + 1)
        }
        nxt = []
        for mv, (r, c) in dirs.items():
            if 0 <= r < 3 and 0 <= c < 3:
                idx = r * 3 + c
                nb = self.board[:]
                nb[b], nb[idx] = nb[idx], nb[b]
                nxt.append(PuzzleState(nb, self, mv, self.depth + 1))
        return nxt

    def build_path(self):
        p, node = [], self
        while node:
            p.append((node.move, node.board, node.depth))
            node = node.parent
        return list(reversed(p))

    def misplaced_tiles(state, goal):
        return sum(1 for i in range(9) if state.board[i] not in (0, goal[i]))
```

```

def manhattan_distance(state, goal):
    d = 0
    for i, v in enumerate(state.board):
        if v != 0:
            r1, c1 = divmod(i, 3)
            r2, c2 = divmod(goal.index(v), 3)
            d += abs(r1 - r2) + abs(c1 - c2)
    return d

def a_star(start, goal, h):
    opened = []
    closed = set()
    nodes = 0
    s = PuzzleState(start)
    s.cost = h(s, goal)
    heapq.heappush(opened, s)

    while opened:
        cur = heapq.heappop(opened)
        nodes += 1

        if cur.board == goal:
            return cur.build_path(),
            nodes

        closed.add(tuple(cur.board))

        for nxt in cur.expand():
            if tuple(nxt.board) in closed:
                continue
            nxt.cost = nxt.depth + h(nxt, goal)
            heapq.heappush(opened, nxt)

    return None, nodes

def print_solution(path, total_nodes):
    print("Steps:\n")
    for mv, st, d in path:
        label = "Start" if mv == "" else f"Move {mv}"
        print(f"{label} | Depth {d}")
        for i in range(0, 9, 3):
            print(" ".join(str(x) if x != 0 else " " for x in st[i:i+3]))
        print()
    print(f"Total Moves: {len(path)-1}")

```

```
print(f"Nodes Expanded: {total_nodes}")

if name_____ == "__main__":
    start = [1, 2, 3,
              4, 0, 6,
              7, 5, 8]

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

    print("A* (Misplaced Tiles)\n")
    sol1, n1 = a_star(start, goal, misplaced_tiles)
    if sol1:
        print_solution(sol1,
n1) else:
    print("No solution.")

    print("\nA* (Manhattan Distance)\n")
    sol2, n2 = a_star(start, goal, manhattan_distance)
    if sol2:
        print_solution(sol2,
n2) else:
    print("No solution.")
```

## OutPut:

```
A* (Manhattan Distance)
```

Steps:

Start | Depth 0

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

Move Down | Depth 1

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

Move Right | Depth 2

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

Total Moves: 2

Nodes Expanded: 3

```
A* (Misplaced Tiles)
```

Steps:

Start | Depth 0

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

Move Down | Depth 1

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

Move Right | Depth 2

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

Total Moves: 2

Nodes Expanded: 3

## PROGRAM 4:

Implement Hill Climbing search algorithm to solve N-Queens problem

## ALGORITHM:

4) Hill climbing search Algorithm (N-Queens)

swap	neighbour	cost
(0,1)	[1,3,2,0]	1
(0,2)	[2,1,3,0]	1
(0,3)	[0,1,2,3]	6
(1,2)	[3,2,1,0]	1
(1,3)	[3,0,2,1]	1
(2,3)	[3,1,0,2]	1

Chosen = (0,1) = [1,3,2,0]

swap	neighbour state	cost
(0,1)	[3,1,2,0]	2
(0,2)	[2,3,1,0]	2
(0,3)	[0,3,2,1]	4
(1,2)	[1,2,3,0]	4
(1,3)	[1,0,2,3]	2
(2,3)	[0,3,0,2]	0

Chosen = (2,3) = [1,3,0,2] cost = 0

Goal state = [1,3,0,2]

Algorithm:

```

function Hill - Climbing(problem) returns a state that is
    local maximum
    current ← make - node (problem, initial - state)
    loop do
        neighbour ← a best - valued successor of current if
            highest
        neighbour. value < current. value then return
            current state
        current ← neighbour
    end
    function Hill Climbing.(n)
        current = random state(n)
        while true:
            h = current = heuristic(current).
            neighbour = bestNeighbour (current)
            if h_neighbour >= h_current:
                if h_current = 0:
                    return current.
            else:
                current = random state(n)
            else:
                current = neighbour.
    output:
        solution = [1,3,0,2]
        [2,0,3,1]
        [0,2,1,3]
        [3,1,2,0]
    
```

## CODE:

```
import random
import time

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

def cost(b):
    n = len(b)
    c = 0
    for i in range(n):
        for j in range(i + 1, n):
            if b[i] == b[j] or abs(b[i] - b[j]) == abs(i - j):
                c += 1
    return c

def best_neighbor(b):
    n = len(b)
    best_b = list(b)
    best_c = cost(b)
    for r in range(n):
        for c in range(n):
            if b[r] != c:
                temp = list(b)
                temp[r] = c
                k = cost(temp)
                if k < best_c:
                    best_c = k
                    best_b = temp
    return best_b, best_c

def hill(n):
    state = [random.randint(0, n - 1) for _ in range(n)]
    c = cost(state)

    print("Initial Board:")
    print_board(state)
    print("Initial Cost:", c, "\n")

    step = 1
    while True:
        print("Step", step)
```

```
print("Current Board:")
print_board(state)
print("Current Cost:", c)
time.sleep(0.2)

nxt, nxt_c = best_neighbor(state)
print("Best Neighbor Cost:", nxt_c, "\n")
time.sleep(0.25)

if nxt_c >= c:
    break

state = nxt
c = nxt_c
step += 1

print("Final Board:")
print_board(state)
print("Final Cost:", c)

if c == 0:
    print("Solution
Found") else:
    print("Local Minimum Reached")

hill(6)
```

### OutPut:

Initial Board:

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

Initial Cost: 4

Step 1

Current Board:

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

Current Cost: 4

Best Neighbor Cost: 2

Step 2

Current Board:

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

Current Cost: 2

Best Neighbor Cost: 2

**Current Cost: 2**

**Best Neighbor Cost: 2**

**Final Board:**

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

**Final Cost: 2**

**Local Minimum Reached**

## PROGRAM 5:

Simulated Annealing to Solve 8-Queens problem

## ALGORITHM:

Pseudocode

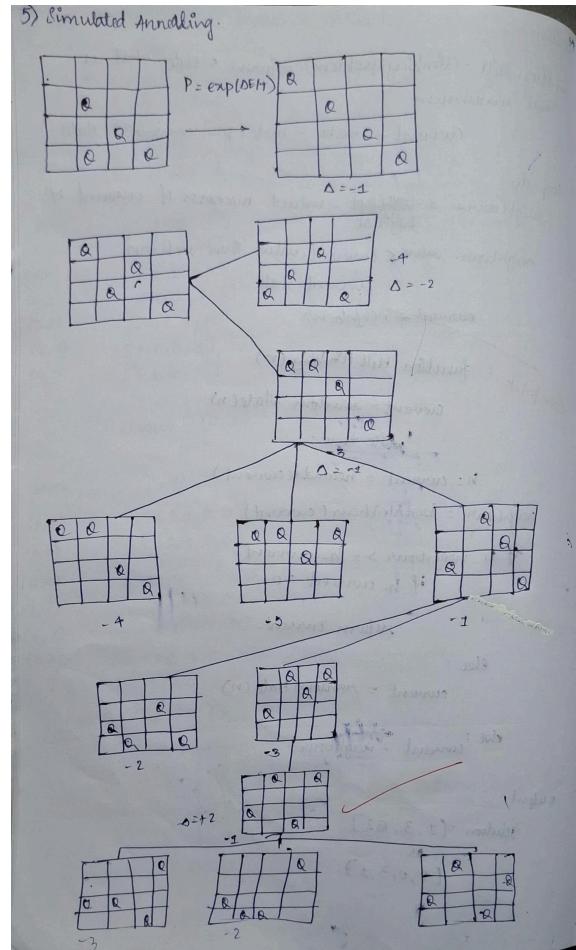
```

current <- initial state
do T < a large positive value
  while T > 0 do
    next <- a random neighbour of current
    DE <- current.cost - next.cost
    if DE > 0, then
      current <- next
    else
      current <- next with probability  $P = e^{\frac{DE}{T}}$ 
    end if
    decrease T
  end while
return current

```

Initial State:

Q							
	Q						
		Q					
			Q				
				Q			
					Q		
						Q	
							Q



## CODE:

```
import random
import math

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

def calculate_cost(board):
    n = len(board)
    cost = 0
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                cost += 1
    return cost

def random_neighbor(board):
    n = len(board)
    neighbor = list(board)
    row = random.randint(0, n - 1)
    col = random.randint(0, n - 1)
    neighbor[row] = col
    return neighbor

def simulated_annealing(n, initial_temp=100.0, cooling_rate=0.95, stopping_temp=1.0,
max_steps=20):
    current_board = [random.randint(0, n - 1) for _ in range(n)]
    current_cost = calculate_cost(current_board)
    temperature = initial_temp
    step = 1

    print("Initial Board:")
    print_board(current_board)
    print(f"Initial Cost: {current_cost}\n")

    while temperature > stopping_temp and current_cost > 0 and step <= max_steps:
        neighbor = random_neighbor(current_board)
        neighbor_cost = calculate_cost(neighbor)
        delta = neighbor_cost - current_cost

        if delta < 0 or random.random() < math.exp(-delta / temperature):
            current_board = neighbor
```

```
current_cost = neighbor_cost

print(f"Step {step}: Temp={temperature:.3f}, Cost={current_cost}")
step += 1
temperature *= cooling_rate

print("\nFinal Board:")
print_board(current_board)
print(f"Final Cost: {current_cost}")

if current_cost == 0:
    print("Goal State Reached!")
else:
    print("Terminated before reaching goal.")

if __name__ == "__main__":
    simulated_annealing(8, initial_temp=100.0, cooling_rate=0.95, stopping_temp=1.0,
max_steps=20)
```

## OutPut:

### Initial Board:

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

Initial Cost: 5

Step 1: Temp=100.000, Cost=5  
Step 2: Temp=95.000, Cost=5  
Step 3: Temp=90.250, Cost=5  
Step 4: Temp=85.737, Cost=4  
Step 5: Temp=81.451, Cost=4  
Step 6: Temp=77.378, Cost=5  
Step 7: Temp=73.509, Cost=5  
Step 8: Temp=69.834, Cost=8  
Step 9: Temp=66.342, Cost=9  
Step 10: Temp=63.025, Cost=10  
Step 11: Temp=59.874, Cost=10  
Step 12: Temp=56.880, Cost=10  
Step 13: Temp=54.036, Cost=10  
Step 14: Temp=51.334, Cost=10  
Step 15: Temp=48.767, Cost=10  
Step 16: Temp=46.329, Cost=12  
Step 17: Temp=44.013, Cost=8  
Step 18: Temp=41.812, Cost=8  
Step 19: Temp=39.721, Cost=9  
Step 20: Temp=37.735, Cost=9

### Final Board:

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

Final Cost: 9

Terminated before reaching goal.

## PROGRAM 6:

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

## ALGORITHM:

Notes - Propositional logic

$$Q \Rightarrow P$$

$$P \Rightarrow Q$$

$$Q \vee R$$

D) Construct Truth Table

i) Does KB entail  $R \Rightarrow P$ ?  
ii) Does KB entail  $P \Rightarrow Q$ ?  
iii) Does KB entail  $R \Rightarrow Q$ ?  
iv) Does KB entail  $(KB \Rightarrow R) \Rightarrow P = N$   
v) Does KB entail  $(KB \Rightarrow Q) \Rightarrow R = Y$

Truth Table:

P	Q	R	$Q \Rightarrow P$	$P \Rightarrow R$	$Q \vee R$	KB
F	F	F	T	T	F	F
F	F	T	T	T	T	T
F	T	F	F	T	T	F
F	T	T	F	T	T	F
T	F	F	T	T	F	F
T	F	T	T	F	T	F
T	T	F	T	F	T	F
T	T	T	T	T	T	T

\* i)  $P = T, Q = F, R = T$   
\* ii)  $P = F, Q = F, R = T$   
iii)  $KB \Rightarrow R = Y$   
iv)  $(KB \Rightarrow R) \Rightarrow P = N$   
v)  $(KB \Rightarrow Q) \Rightarrow R = Y$

iii)

P	Q	R	KB	$R \Rightarrow P$
T	F	T	T	T
F	F	T	T	f → failed X

∴ No KB doesn't entail  $R \Rightarrow P$  as  $P: F \neq T$

Algorithm:

i)  $\text{TT\_entail}(KB, x)$ :  
Symbols  $\leftarrow$  all unique symbols in  $KB \cup \{x\}$   
return  $\text{TT\_check\_All}(KB, x, \text{symbols}, t^y)$

ii)  $\text{TT\_check\_All}(KB, x, \text{symbols}, \text{model})$ :  
if Symbols is empty:  
if evaluate( $KB, \text{model}$ ) then  
return evaluate( $x, \text{model}$ )  
else:  
return True

else:  
 $p \leftarrow$  first symbol.  
return  $\text{TT\_check\_All}(KB, x, \text{rest}, \text{model} \cup \{p = \text{true } y\})$

AND

iii)  $\text{TT\_check\_All}(KB, x, \text{rest}, \text{model} \cup \{p = \text{false } y\})$ :

✓ 8/10

## CODE:

```
import
itertools
import re

def interpret(expr, model):
    expr = expr.replace("<->", " == ")
    expr = expr.replace(">-", " <= ")
    expr = re.sub(r'~(\w+)', r'(not \1)', expr)
    expr = re.sub(r'~(([^)])+)', r'(not (\1))', expr)
    expr = expr.replace("^", " and ")
    expr = expr.replace("v", " or ")
    for s, v in model.items():
        expr = re.sub(r'\b' + re.escape(s) + r'\b', str(v), expr)
    return eval(expr)

def truth_table(kb, query, symbols):
    all_models = list(itertools.product([True, False], repeat=len(symbols)))
    ent = True

    print("Truth Table:\n")
    head = " | ".join(symbols) + " | KB | Q | KB⇒Q"
    print(head)
    print("-" * (len(head) + 10))

    for vals in all_models:
        model = dict(zip(symbols, vals))
        kb_v = interpret(kb, model)
        q_v = interpret(query, model)
        impl = (not kb_v) or q_v

        if kb_v and not q_v:
            ent = False

        row = " | ".join(["T" if v else "F" for v in vals])
        row += f" | {'T' if kb_v else 'F'} | {'T' if q_v else 'F'} | {'T' if impl else 'F'}"
        print(row)

    print("\nOutcome:")
    if ent:
        print("KB entails Query")
    else:
        print("KB does not entail Query")

def run_evaluation(kb, queries, symbols):
    print("\nKnowledge Base:", kb)
    print("Symbols:", ", ".join(symbols))
    print("\n" + "="*60 + "\n")
```

```

for q in queries:
    print("Evaluating Query:", q)
    print()
    truth_table(kb, q, symbols)
    print("\n" + "="*60 + "\n")

kb = "(Q -> P) ^ (P -> ~Q) ^ (Q v R)"
symbols = ["P", "Q", "R"]
queries = ["R", "R -> P", "Q -> R", "~P v (Q -> ~R)"]

run_evaluation(kb, queries, symbols)

```

## Output:

Knowledge base:  $(Q \rightarrow P) \wedge (P \rightarrow \neg Q) \wedge (Q \vee R)$   
 Symbols: P, Q, R

Evaluating Query: R

Truth Table:

P	Q	R	KB	Q	KB=Q
T	T	T	F	T	T
T	T	F	F	T	
T	F	T	T	T	T
T	F	F	F	T	
F	T	T	F	T	T
F	T	F	F	T	
F	F	T	T	T	T
F	F	F	F	T	

Outcome:

KB entails Query

Evaluating Query: R -> P

Truth Table:

P	Q	R	KB	Q	KB=Q
T	T	T	F	T	T
T	T	F	F	T	
T	F	T	T	T	T
T	F	F	F	T	
F	T	T	F	T	T
F	T	F	F	T	
F	F	T	T	T	T
F	F	F	F	T	

Outcome:  
 KB does not entail Query

Evaluating Query: Q -> R

Truth Table:

P	Q	R	KB	Q	KB=Q
T	T	T	F	T	T
T	T	F	F	T	
T	F	T	T	T	T
T	F	F	F	T	
F	T	T	F	T	T
F	T	F	F	T	
F	F	T	T	T	T
F	F	F	F	T	

Outcome:  
 KB entails Query

Evaluating Query: ~P v (Q -> ~R)

Truth Table:

P	Q	R	KB	Q	KB=Q
T	T	T	F	F	T
T	T	F	F	T	
T	F	T	T	T	T
T	F	F	F	T	
F	T	T	F	T	T
F	T	F	F	T	
F	F	T	T	T	T
F	F	F	F	T	

Outcome:

KB entails Query

## PROGRAM 7:

Implement unification in first order logic

## ALGORITHM:

Mark 4  
 First Order Logic  
 $\text{unify}(x, y, \theta)$   
 if  $\theta = \text{failure}$   
 return failure  
 else if  $x = y$   
 return  $\theta$   
 else if  $x$  is a variable:  
 return unify(var( $x$ ,  $y$ ),  $\theta$ )  
 else if  $y$  is a variable:  
 return unify(var( $y$ ,  $x$ ),  $\theta$ )  
 else if  $x$  &  $y$  are compound expressions:  
 if predicate( $x$ ) ≠ predicate( $y$ ) & length( $x$ ) ≠ length( $y$ )  
 return failure  
 else:  
 return unify(argument( $x$ ), argument( $y$ ),  $\theta$ )  
 else:  
 return failure  
 $\text{unify\_var}(\text{var}, x, \theta)$   
 if var is in  $\theta$ :  
 return unify( $\theta$ , [var],  $x, \theta$ )  
 else if  $x$  is in  $\theta$   
 return unify([var],  $x, \theta$ )  
 else if var occurs in  $x$ :  
 return failure  
 else:  
 add  $x = y$  to  $\theta$   
 return  $\theta$ .

Ex 1:  
 $\text{unify}[\text{knows}(\text{John}, x), \text{knows}(y, \text{Elizabeth})]$   
 $\theta \rightarrow y = \text{John}, x = \text{Elizabeth}$   
 $\theta \rightarrow \text{knows}(\text{John}, \text{Elizabeth})$   
 $\therefore \text{successful}$   
Ex 2:  $\text{unify}[\text{pred}(x, \text{hey}), \text{pred}, f(z)]$   
 $\theta \neq f$   
 $\therefore \text{failure}$   
Ex 3:  $\text{unify}[\text{Loves}(x, \text{ay}), \text{Loves}(\text{John}, f(\text{Mary}))]$   
 $\theta = \{x = \text{John}, y = \text{Mary}\}$   
 $\therefore \text{successful}$

## CODE:

```
import
time class
Term:
    def __init__(self, value):
        self.value = value
    def __eq__(self, other):
        return isinstance(other, Term) and self.value == other.value
    def __hash__(self):
        return hash(self.value)
    def __repr__(self):
        return str(self.value)

class Constant(Term):
    pass

class Variable(Term):
    pass

class Function(Term):
    def __init__(self, symbol, args):
        self.value = symbol
        self.arguments = list(args)
    def __eq__(self, other):
        return isinstance(other, Function) and self.value == other.value and self.arguments == other.arguments
    def __hash__(self):
        return hash((self.value, tuple(self.arguments)))
    def __repr__(self):
        return f'{self.value}({", ".join(map(str, self.arguments))})'

FAILURE = "FAILURE"

def occurs_in(v, t):
    if v == t:
        return True
    if isinstance(t, Function):
        return any(occurs_in(v, a) for a in t.arguments)
    return False

def substitute(subst, t):
    if not subst:
        return t
    if isinstance(t, Variable):
        return subst.get(t, t)
    if isinstance(t, Function):
```

```

        return Function(t.value, [substitute(subst, a) for a in t.arguments])
    return t

def compose(s1, s2):
    if not s1:
        return s2 or
    {} if not s2:
        return dict(s1)
    r = {v: substitute(s1, t) for v, t in s2.items()}
    for v, t in s1.items():
        if v not in r:
            r[v] = t
    return r

def unify_core(a, b):
    if a == b:
        return {}
    if isinstance(a, Variable):
        if occurs_in(a, b):
            return FAILURE
        return {a: b}
    if isinstance(b, Variable):
        if occurs_in(b, a):
            return FAILURE
        return {b: a}
    if isinstance(a, Function) and isinstance(b, Function) and a.value == b.value and len(a.arguments)
    == len(b.arguments):
        s = {}
        for x, y in zip(a.arguments, b.arguments):
            x2 = substitute(s, x)
            y2 = substitute(s, y)
            r = unify_core(x2, y2)
            if r == FAILURE:
                return FAILURE
            if r:
                s = compose(r, s)
        return s
    return FAILURE

def unify(a, b):
    return unify_core(a, b)

def pretty(s):
    if not s:
        return "{}"
    return "{" + ", ".join(["f'{k}'->{v}" for k, v in s.items()]) + "}"

```

```

def trace_unify(a, b):
    print("Unify:")
    print("A =", a)
    print("B =", b)
    print()
    steps = [(a,
              b)] subst = {}
    step = 1
    while steps:
        x, y = steps.pop(0)
        x = substitute(subst, x)
        y = substitute(subst, y)
        print("Step", step)
        print("Compare:", x, "and",
              y) if x == y:
            print("Equal\n")
            step += 1
            time.sleep(0.1)
            continue
        if isinstance(x, Variable):
            if occurs_in(x, y):
                print("Failure\n")
                return FAILURE
            s = {x: y}
            subst = compose(s, subst)
            print("Bind:", pretty(s))
            print("Now:", pretty(subst), "\n")
            step += 1
            time.sleep(0.1)
            continue
        if isinstance(y, Variable):
            if occurs_in(y, x):
                print("Failure\n")
                return FAILURE
            s = {y: x}
            subst = compose(s, subst)
            print("Bind:", pretty(s))
            print("Now:", pretty(subst), "\n")
            step += 1
            time.sleep(0.1)
            continue
        if isinstance(x, Function) and isinstance(y, Function) and x.value == y.value:
            for a1, a2 in zip(x.arguments, y.arguments):
                steps.insert(0, (a1, a2))
            print("Expand arguments\n")
            step += 1
            time.sleep(0.1)

```

```

        continue
    print("Failure\n")
    return FAILURE
print("Final substitution:", pretty(subst), "\n")
return subst

X = Variable("X")
Y = Variable("Y")
A = Constant("a")
B = Constant("b")

t1 = Function("P", [X, A])
t2 = Function("P", [B, Y])

t3 = Function("R", [X, Function("f", [X])])
t4 = Function("R", [A, Function("f", [B])])

print("\n==== Example 1 ====\n")
trace_unify(t1, t2)

print("\n==== Example 2 ====\n")
trace_unify(t3, t4)

```

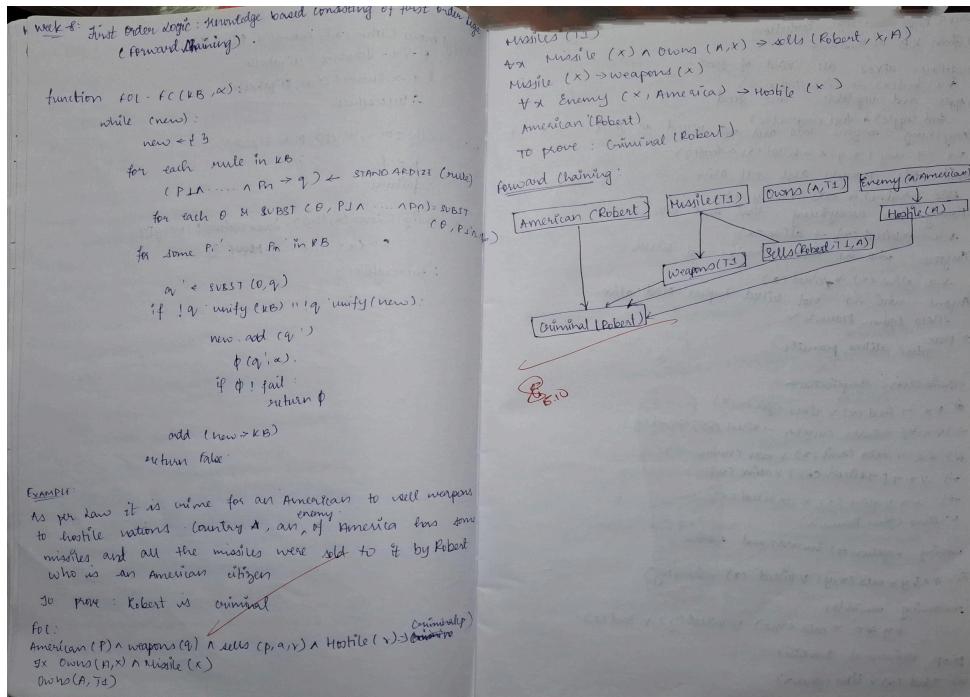
## OutPut:

<pre> ==== Example 1 ====  unify: A = P(X, a) B = P(b, Y)  Step 1 Compare: P(X, a) and P(b, Y) Expand arguments  Step 2 Compare: a and Y Bind: {Y-&gt;a} Now: {Y-&gt;a}  Step 3 Compare: X and b Bind: {X-&gt;b} Now: {Y-&gt;a, X-&gt;b}  Final substitution: {Y-&gt;a, X-&gt;b} </pre>	<pre> ==== Example 2 ====  Unify: A = R(X, f(X)) B = R(a, f(b))  Step 1 Compare: R(X, f(X)) and R(a, f(b)) Expand arguments  Step 2 Compare: f(X) and f(b) Expand arguments  Step 3 Compare: X and b Bind: {X-&gt;b} Now: {X-&gt;b}  Step 4 Compare: b and a Failure </pre>
---	---

## PROGRAM 8:

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

## ALGORITHM:



## CODE:

```
class Predicate:  
    def __init__(self, name, args):  
        self.name = name  
        self.args = args  
  
    def __repr__(self):  
        return f'{self.name}({", ".join(map(str, self.args))})'  
  
    def __eq__(self, other):  
        return isinstance(other, Predicate) and self.name == other.name and self.args == other.args  
  
    def __hash__(self):  
        return hash((self.name, tuple(self.args)))  
  
class Var:  
    def __init__(self, name):  
        self.name = name  
  
    def __repr__(self): return  
        self.name  
  
    def __eq__(self, other):  
        return isinstance(other, Var) and self.name == other.name  
  
    def __hash__(self): return  
        hash(self.name)  
  
class Const:  
    def __init__(self, name):  
        self.name = name  
  
    def __repr__(self): return  
        self.name  
  
    def __eq__(self, other):  
        return isinstance(other, Const) and self.name == other.name  
  
    def __hash__(self): return  
        hash(self.name)  
  
def is_variable(x):  
    return isinstance(x, Var)  
  
def unify(x, y, subst={}):  
    if subst is None:
```

```

        return None
    elif x == y:
        return subst
    elif is_variable(x):
        return unify_var(x, y, subst)
    elif is_variable(y):
        return unify_var(y, x, subst)
    elif isinstance(x, Predicate) and isinstance(y, Predicate):
        if x.name != y.name or len(x.args) != len(y.args):
            return None
        for a, b in zip(x.args, y.args):
            subst = unify(a, b, subst)
            if subst is None:
                return None
        return subst
    else:
        return None

def unify_var(var, x, subst):
    if var in subst:
        return unify(subst[var], x, subst)
    elif is_variable(x) and x in subst:
        return unify(var, subst[x], subst)
    elif occurs_check(var, x, subst):
        return None
    else:
        subst_copy = subst.copy()
        subst_copy[var] = x
        return subst_copy

def occurs_check(var, x, subst):
    if var == x:
        return True
    elif is_variable(x) and x in subst:
        return occurs_check(var, subst[x], subst)
    elif isinstance(x, Predicate):
        return any(occurs_check(var, arg, subst) for arg in x.args)
    else:
        return False

def substitute(predicate, subst):
    new_args = []
    for arg in predicate.args:
        val = arg
        while is_variable(val) and val in subst:
            val = subst[val]
        new_args.append(val)

```

```

return Predicate(predicate.name, new_args)

class Rule:
    def __init__(self, premises, conclusion):
        self.premises = premises
        self.conclusion = conclusion

    def __repr__(self):
        return f'{self.premises} => {self.conclusion}'

def forward_chain(kb_facts, kb_rules, query):
    inferred = set(kb_facts)
    print("Initial Facts:")
    for f in inferred:
        print(f'{f}')
    print("\nStarting inference steps:\n")

    new_inferred = True

    while new_inferred:
        new_inferred = False
        for rule in kb_rules:
            possible_substs = [{}]

            for premise in rule.premises:
                temp_substs = []
                for fact in inferred:
                    for subst in possible_substs:
                        subst_try = unify(premise, fact, subst)
                        if subst_try is not None:
                            temp_substs.append(subst_try)
                possible_substs = temp_substs

            for subst in possible_substs:
                concluded_fact = substitute(rule.conclusion, subst)
                if concluded_fact not in inferred:
                    print(f'Inferred: {concluded_fact} from rule {rule} using substitution {subst}')
                    inferred.add(concluded_fact)
                    new_inferred = True
                if unify(concluded_fact, query) is not None:
                    print(f'\nQuery {query} proved!')
                    return True

    print(f'\nQuery {query} not proved.')
    return False

if __name__ == "__main__":

```

```

a = Const('a')
b = Const('b')
c = Const('c')
x = Var('x')
y = Var('y')
z = Var('z')

kb_facts = {
    Predicate('Parent', [a, b]),
    Predicate('Parent', [b, c]),
}

kb_rules = [
    Rule([Predicate('Parent', [x, y])], Predicate('Ancestor', [x, y])),
    Rule([Predicate('Parent', [x, y]), Predicate('Ancestor', [y, z])], Predicate('Ancestor', [x, z])),
]

query = Predicate('Ancestor', [a, c])

print("Running forward chaining...\n")
forward_chain(kb_facts, kb_rules, query)

```

## Output:

```

Running forward chaining...

Initial Facts:
Parent(a, b)
Parent(b, c)

Starting inference steps:

Inferred: Ancestor(a, b) from rule Parent(x, y) => Ancestor(x, y) using substitution {x: a, y: b}
Inferred: Ancestor(b, c) from rule Parent(x, y) => Ancestor(x, y) using substitution {x: b, y: c}
Inferred: Ancestor(a, c) from rule Parent(x, y) ∧ Ancestor(y, z) => Ancestor(x, z) using substitution {x: a, y: b, z: c}

Query Ancestor(a, c) proved!

```

## **PROGRAM 9:**

Create a knowledge base consisting of first order logic statements and prove the given query using Resolution

## ALGORITHM:

Week 9: First Order Logic: Resolution

D) Given KB on Business:

- John likes all kind of food.
- $\forall x : \text{food}(x) \rightarrow \text{likes}(\text{John}, x)$
- Apple and vegetable are food.
- Food (apple) & food (vegetable)  $\wedge$  not mixed is food.
- Anything anyone eats and not mixed is food.
- $\forall x \forall y : \text{eats}(x, y) \wedge \neg \text{mixed}(x, y) \rightarrow \text{food}(y)$
- Hammy eats peanuts and still alive.
- $\text{eats}(\text{Hammy}, \text{peanuts}) \wedge \text{alive}(\text{Hammy})$
- Hammy eats everything that Aril eats.
- $\forall x : \forall y : \text{eats}(x, y) \rightarrow \text{alive}(y)$
- Anyone who is alive implies not killed.
- $\forall x : \text{alive}(x) \rightarrow \neg \text{killed}(x)$
- Anyone who is not killed implies dead alive.
- Likes (John, Peanuts)

To Prove: John likes peanuts.

Proof by Resolution:

```

graph TD
    A[likes(John, Peanuts)] --> B[food(x) v likes(John, x)]
    A --> C[Peanuts != y]
    B --> D[eats(y, z) v mixed(y, z)]
    B --> E[food(z)]
    C --> F[eats(y, z) v mixed(y, z)]
    C --> G[eats(Aril, Peanuts)]
    D --> H[alive(z) v -killed(z)]
    E --> H
    F --> I[alive(y) v -killed(y)]
    G --> I
    H --> J[alive(Aril, y) v -killed(y)]
    I --> J
    J[Hence Proven]
  
```

Elimination Implication:

- $\forall x : \text{food}(x) \vee \text{likes}(\text{John}, x)$
- $\forall x \forall y : \text{eats}(x, y) \wedge \neg \text{mixed}(x, y) \rightarrow \text{food}(y)$
- $\forall x : \forall y : \text{eats}(\text{Aril}, x) \vee \text{eats}(\text{Hammy}, x)$
- $\forall x : \forall y : [\neg \text{killed}(x)] \vee \text{alive}(x)$
- $\forall x : \forall y : \text{alive}(x) \vee \neg \text{killed}(x)$
- Likes (John, Peanuts)

Moving negation ( $\neg$ ) inwards and rewrite.

- $\forall x \forall y : \text{eats}(x, y) \vee \text{killed}(x) \vee \text{food}(y)$

remaining variables:

- $\forall y \forall z : \text{eats}(y, z) \vee \text{killed}(y) \vee \text{food}(z)$ .

Drop universal quantifiers:

- $\text{food}(x) \vee \text{likes}(\text{John}, x)$

## CODE:

```

def is_variable(x):
    return x[0].islower() and len(x) == 1

def unify(x, y, subs=None):
    if subs is None:
        subs = {}
    if x == y:
        return subs
    if is_variable(x):
        return unify_var(x, y, subs)
    if is_variable(y):
        return unify_var(y, x, subs)
    if isinstance(x, list) and isinstance(y, list) and len(x) == len(y):
        for a, b in zip(x, y):
            subs = unify(a, b, subs)
        if subs is None:
            return None
        return subs
    return None

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

def occurs_check(var, x, subs):
    if var == x:
        return True
    if isinstance(x, list):
        return any(occurs_check(var, xi, subs) for xi in x)
    if x in subs:
        return occurs_check(var, subs[x], subs)
    return False

def negate(literal):
    return literal[1:] if literal.startswith("¬") else "¬" + literal

def parse_predicate(pred):
    name, args = pred.split("(")
    args = args[:-1].split(",")
    args = [a.strip() for a in args]

```

```

return name.strip(), args

def substitute(literal, subs):
    name, args = parse_predicate(literal.replace("¬", ""))
    new_args = [subs.get(a, a) for a in args]
    new_lit = name + "(" + ", ".join(new_args) + ")"
    return "¬" + new_lit if literal.startswith("¬") else new_lit

def unify_predicates(p1, p2):
    p1_clean = p1.replace("¬", "")
    p2_clean = p2.replace("¬", "")
    n1, a1 = parse_predicate(p1_clean)
    n2, a2 = parse_predicate(p2_clean)
    if n1 != n2 or len(a1) != len(a2):
        return None
    return unify(a1, a2)

def resolve(ci, cj):
    for li in ci:
        for lj in cj:
            if lj == negate(li):
                subs = unify_predicates(li, lj)
                if subs is not None:
                    new_clause = set()
                    for l in ci.union(cj):
                        if l != li and l != lj:
                            new_clause.add(substitute(l, subs))
                    return new_clause, (li, lj), subs
    return None, None, None

```

```

KB = [
    {"¬food(x)", "likes(John,x)"},  

    {"food(Apple)"},  

    {"food(vegetables)"},  

    {"¬eats(y,z)", "killed(y)", "food(z)"},  

    {"eats(Anil,Peanuts)"},  

    {"alive(Anil)"},  

    {"¬eats(Anil,w)", "eats(Harry,w)"},  

    {"killed(g)", "alive(g)"},  

    {"¬alive(k)", "¬killed(k)"},  

    {"likes(John,Peanuts)"}
]

```

```

def resolution_tree(KB, query):
    print("      PROOF BY RESOLUTION (FOL)      ")

```

```

print(f'Goal: Prove that {query}\n')
print("Converting to CNF and negating the query...\n")

clauses = [c.copy() for c in KB]
negated_query = negate(query)
clauses.append({negated_query})
print(f'Negated query added to KB: {negated_query}\n')
new = []
step = 1
proof_steps = []
tree_nodes = []

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:
        resolvent, used, subs = resolve(ci, cj)
        if resolvent is not None:
            proof_steps.append({
                "step": step,
                "parents": (ci, cj),
                "used": used,
                "subs": subs,
                "resolvent": resolvent
            })

            print(f"Step {step}: Resolving {ci} and {cj}")
            print(f"  Used literals: {used}")
            if subs:
                print(f"  Substitution: {subs}")
            print(f"  => New Clause: {resolvent}\n")
            tree_nodes.append((ci, cj, resolvent))

        if not resolvent:
            print("  Empty clause derived — Query is PROVED!\n")
            print("\n===== PROOF TREE =====\n")
            print_tree(tree_nodes)
            return True

    if resolvent not in clauses:
        new.append(resolvent)
        step += 1

```

```

if not new:
    print(" No new clauses can be derived — Query cannot be proven.\n")
    print("\n===== PROOF TREE (Partial) =====\n")
    print_tree(tree_nodes)

return False

clauses.extend(new)
new = []

def print_tree(nodes):
    def print_branch(node, level=0):
        indent = " " * level
        c1, c2, result = node
        print(f'{indent} |— Derived {result}')
        from:") print(f'{indent} |   {c1}"')
        print(f'{indent} |   {c2}"')

    for n in nodes:
        print_branch(n)
        print("")

if name____ == "_main_": query
    = "likes(John,Peanuts)"
    result = resolution_tree(KB, query)
    print("Result:", " PROVED" if result else " NOT PROVED")

```

## OutPut:

PROOF BY RESOLUTION (FOL)

Goal: Prove that likes(John,Peanuts)

Converting to CNF and negating the query...

Negated query added to KB:  $\neg$ likes(John,Peanuts)

Step 1: Resolving {'likes(John,Peanuts)'} and {' $\neg$ likes(John,Peanuts)'}

Used literals: ('likes(John,Peanuts)'), ' $\neg$ likes(John,Peanuts)')

→ New Clause: set()

Empty clause derived – Query is PROVED!

===== PROOF TREE =====

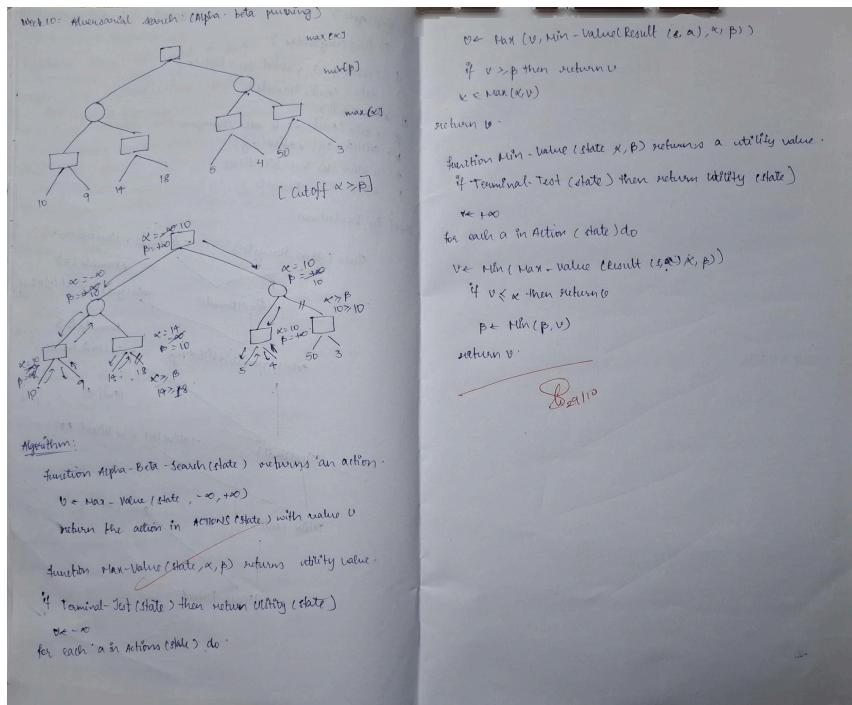
|- Derived set() from:  
  {'likes(John,Peanuts)' }  
  {' $\neg$ likes(John,Peanuts)' }

Result: PROVED

**PROGRAM 10:**

Implement Alpha-Beta Pruning.

## ALGORITHM:



## CODE:

```
import math
```

```
tree = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': ['H', 'I'],
    'E': ['J', 'K'],
    'F': ['L', 'M'],
    'G': ['N', 'O'],
    'H': [], 'T': [], 'J': [], 'K': [],
    'L': [], 'M': [], 'N': [], 'O': []
}
```

```
values = {
    'H': 3, 'T':
    5,
```

```

'J': 6, 'K': 9,
'L': 1, 'M': 2,
'N': 0, 'O': -1
}

def get_children(node):
    return tree.get(node, [])

def evaluate(node):
    return values.get(node, 0)

def alphabeta(node, depth, alpha, beta, maximizingPlayer):
    if not get_children(node) or depth == 0:
        return evaluate(node)

    if maximizingPlayer:
        value = -math.inf
        print(f"MAX Node {node}: α={alpha}, β={beta}")
        for child in get_children(node):
            value = max(value, alphabeta(child, depth - 1, alpha, beta, False))
            alpha = max(alpha, value)
            print(f" MAX updating α={alpha} after visiting {child}")
            if alpha >= beta:
                print(f" Pruned remaining children of {node} (α={alpha}, β={beta})")
                break
        return value
    else:
        value = math.inf
        print(f"MIN Node {node}: α={alpha}, β={beta}")
        for child in get_children(node):
            value = min(value, alphabeta(child, depth - 1, alpha, beta, True))
            beta = min(beta, value)
            print(f" MIN updating β={beta} after visiting {child}")
            if beta <= alpha:
                print(f" Pruned remaining children of {node} (α={alpha}, β={beta})")
                break
        return value

print("\n--- Running Alpha-Beta Pruning on Minimax Tree ---\n")
optimal_value = alphabeta('A', depth=4, alpha=-math.inf, beta=math.inf, maximizingPlayer=True)
print("\nOptimal value at the root (A):", optimal_value)

```

## **OutPut:**

--- Running Alpha-Beta Pruning on Minimax Tree ---

```
MAX Node A: α=-inf, β=-inf
MIN Node B: α=-inf, β=-inf
MAX Node D: α=-inf, β=-inf
    MAX updating α=3 after visiting H
    MAX updating α=5 after visiting I
    MIN updating β=5 after visiting D
MAX Node E: α=-inf, β=5
    MAX updating α=6 after visiting J
        Pruned remaining children of E (α=6, β=5)
    MIN updating β=5 after visiting E
    MAX updating α=5 after visiting B
MIN Node C: α=5, β=-inf
MAX Node F: α=5, β=-inf
    MAX updating α=5 after visiting L
    MAX updating α=5 after visiting M
    MIN updating β=2 after visiting F
        Pruned remaining children of C (α=5, β=2)
    MAX updating α=5 after visiting C
```

Optimal value at the root (A): 5