VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Harbakhshish Singh Arora (1BM23CS104)

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

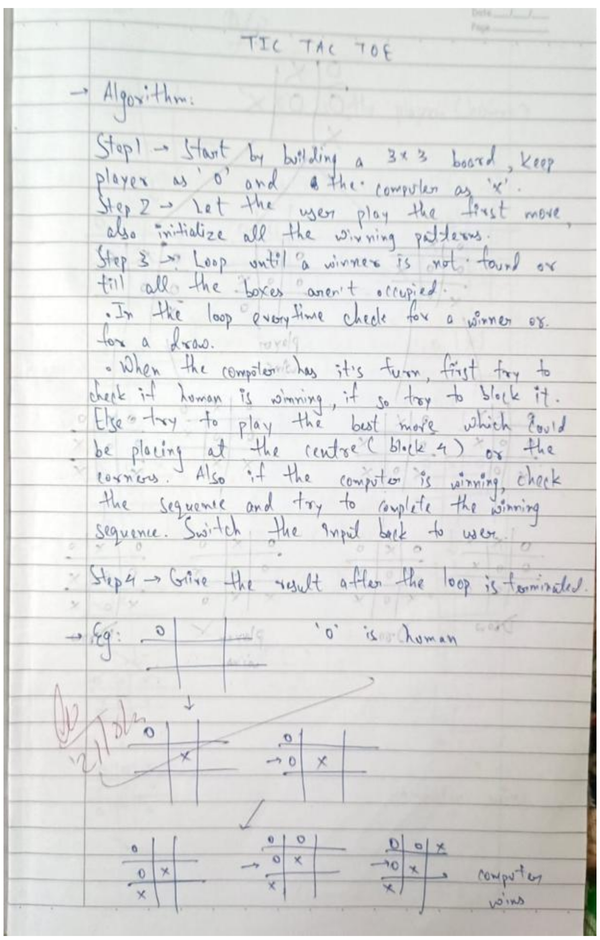
This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Harbakhshish Singh Arora (1BM23CS104),** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and 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.

Anusha S Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

INDEX

Program 1: Implement Tic –Tac –Toe Game & Implement vacuum cleaner agent.

ALGORITHM (Tic-Tac-Toe) :

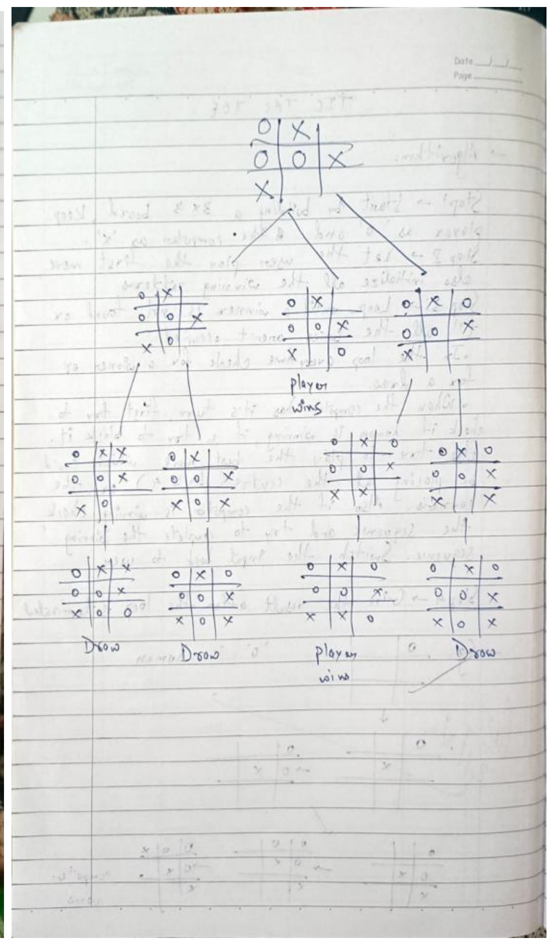

TIC TAC TOE

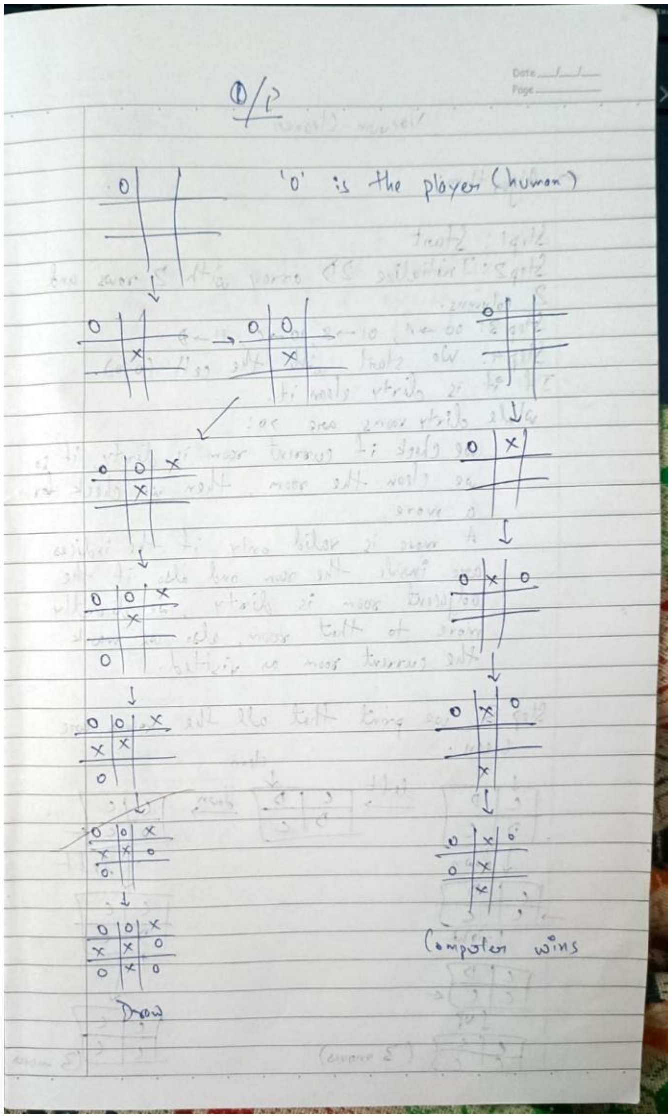
→ Algorithm:

Step1 → Start by building a 3×3 board, keep player as 'O' and the computer as 'x'.
Step 2 → Let the user play the first move, also initialize all the winning patterns.
Step 3 → Loop until a winner is not found or till all the boxes aren't occupied.
• In the loop everytime check for a winner or for a draw.
• When the computer has it's turn, first try to check if human is winning, if so try to block it. Else try to play the best move which could be placing at the centre (block 4) or the corners. Also if the computer is winning, check the sequence and try to complete the winning sequence. Switch the input back to user.

Step4 → Give the result after the loop is terminated.

→ Eg: 'O' is human





CODE (Tic-Tac-Toe) :

```
import math import
random def
print_board(board):
    """Prints the current state of the game board."""
for row in [board[i:i+3] for i in range(0, 9, 3)]:
        print('| ' + ' | '.join(row) + ' |')
def available_moves(board):
    """Returns a list of available spots on the board."""
return [i for i, spot in enumerate(board) if spot == ' '] def
check_winner(board, player):
    """Checks if the given player has won the game."""
```

```
    # Check for winning rows      for i in
range(0, 9, 3):          if all(s == player for s
in board[i:i+3]):
            return True

    # Check for winning columns      for i in range(3):
if board[i] == board[i+3] == board[i+6] == player:
            return True

    # Check for winning diagonals      if board[0]
== board[4] == board[8] == player:          return
True      if board[2] == board[4] == board[6] ==
player:
        return True

    return False

def minimax(board, is_maximizing):
    """
    Minimax algorithm to find the best move.
    'X' is the maximizing player (computer), 'O' is the minimizing player (human).
    """      ai_player =
'X'      human_player
= 'O'

    if check_winner(board, ai_player):
        return 1, None      if
check_winner(board, human_player):
        return -1, None      if not
available_moves(board):
        return 0, None
```

if is_maximizing: best_score = -
math.inf   best_move = None   for
move in available_moves(board):
board[move] = ai_player   score, _
= minimax(board, False)
board[move] = ' '   if score >
best_score: best_score = score
best_move = move   return
best_score, best_move   else:
best_score = math.inf
best_move = None   for move in
available_moves(board):
board[move] = human_player
score, _ = minimax(board, True)
board[move] = ' '   if score <
best_score: best_score = score
best_move = move
return best_score, best_move

def get_computer_move(board):
"""
Gets a beatable move for the computer. It finds all moves with a positive score and chooses one randomly. If no such moves exist, it chooses a random move to prolong the game.
"""   ai_player =
'X'   human_player
= 'O'

# If the board is empty, take the center.
if not available_moves(board):
return 4

# Check for an immediate win for the computer
for move in available_moves(board):
board[move] = ai_player if
check_winner(board, ai_player):
board[move] = ' '
return move board[move]
= ' '

Check for an immediate block of the human
player for move in available_moves(board):
board[move] = human_player if
check_winner(board, human_player):
board[move] = ' '
return move board[move]
= ' '

# If no immediate win or block, use a simplified minimax approach.
# Find all moves that result in a positive score.
positive_moves = [] for move in
available_moves(board):
board[move] = ai_player score, _
= minimax(board, False)
board[move] = ' ' if score > 0:
positive_moves.append(move)

# If there are any moves that lead to a win or a draw, pick one at random.
if positive_moves:
return random.choice(positive_moves)

If all remaining moves lead to a loss, choose a random available move to delay the loss.
return random.choice(available_moves(board))

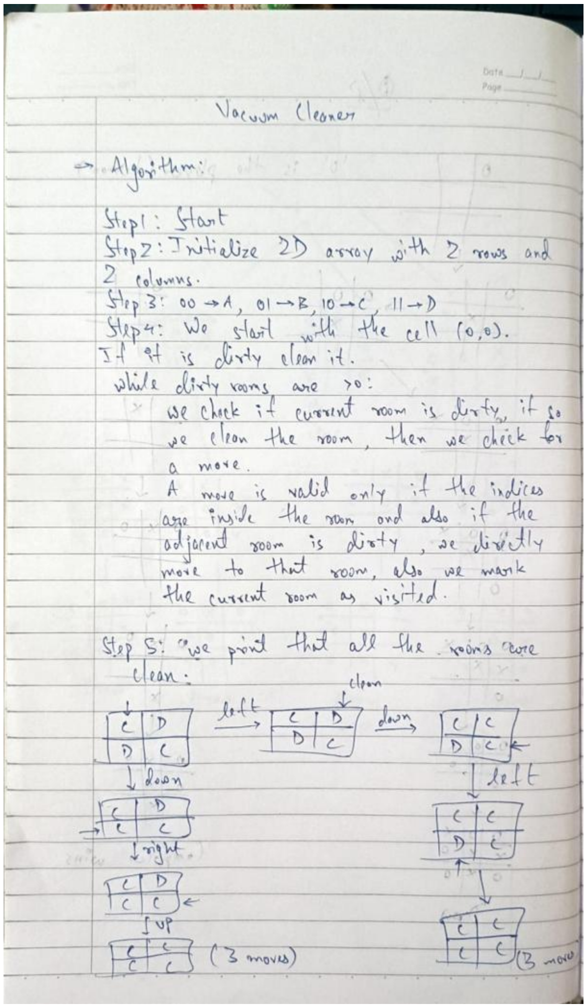
```
def get_player_move(board):
   """Gets a valid move from the human
player."""     while True:         try:
         move = int(input("Enter your move (1-9): ")) - 1
if move not in available_moves(board):
            print("Invalid move. Please enter a number from 1-9 that is not taken.")
else:
            return move
except ValueError:
         print("Invalid input. Please enter a number.")

def play_game():
   """The main function to run the Tic-Tac-Toe game loop."""
board = [' '] * 9
   print("Welcome to Tic-Tac-Toe!") print_board(board)

   # Let the human player ('O') go first.
   turn = 'O'

   while True:        if turn == 'O':
# Human's turn            move =
get_player_move(board)
board[move] = 'O'        else:
         # Computer's turn
print("Computer is thinking...")
move = get_computer_move(board)
board[move] = 'X'

      print(f"Player {turn} makes a move to square {move + 1}")
print_board(board)
```

```
        # Check for a winner after the
move        if check_winner(board, turn):
print(f"{turn} wins!")            break

        # Check for a tie        if not
available_moves(board):
            print("It's a draw!")
break

        # Switch turns        turn = 'X'
if turn == 'O' else 'O'

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

OUTPUT:

```
Welcome to Tic-Tac-Toe!
|   |   |   |
|   |   |   |
|   |   |   |
Enter your move (1-9): 5
Player O makes a move to square 5
|   |   |   |
|   | O |   |
|   |   |   |
Computer is thinking...
Player X makes a move to square 6
|   |   |   |
|   | O | X |
|   |   |   |
Enter your move (1-9): 1
Player O makes a move to square 1
| O |   |   |
|   | O | X |
|   |   |   |
Computer is thinking...
Player X makes a move to square 9
| O |   |   |
|   | O | X |
|   |   | X |
Enter your move (1-9): 3
Player O makes a move to square 3
| O |   | O |
|   | O | X |
|   |   | X |
Computer is thinking...
Player X makes a move to square 2
| O | X | O |
|   | O | X |
|   |   | X |
Enter your move (1-9): 7
Player O makes a move to square 7
| O | X | O |
|   | O | X |
| O |   | X |
O wins!
```

ALGORITHM (Vacuum Cleaner):

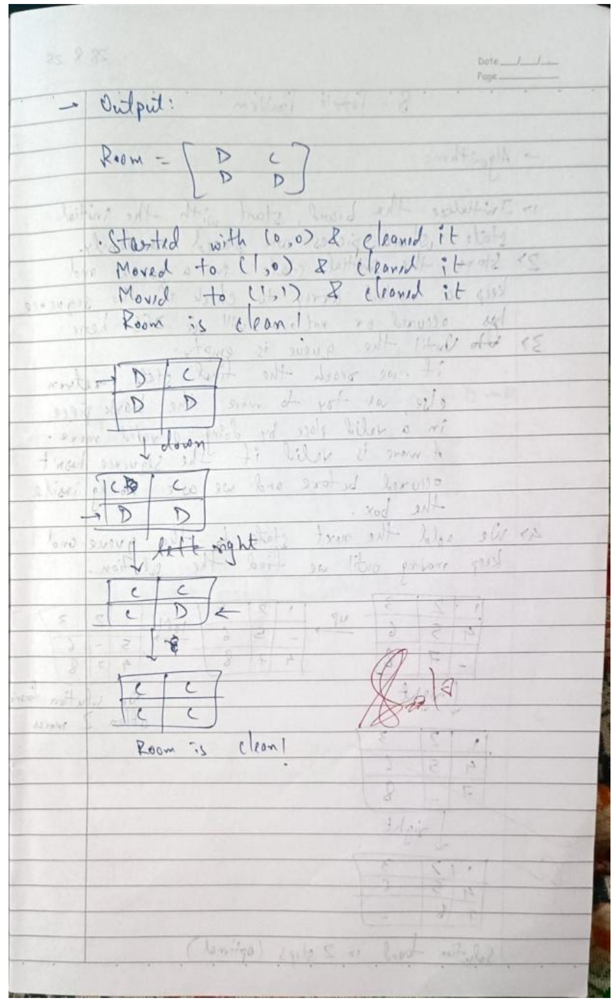


Vacuum Cleaner

→ Algorithm:

Step1: Start
Step2: Initialize 2D array with 2 rows and 2 columns.
Step 3: 00 → A, 01 → B, 10 → C, 11 → D
Step4: We start with the cell (0,0).
If it is dirty clean it.
while dirty rooms are > 0:
we check if current room is dirty, if so we clean the room, then we check for a move.
A move is valid only if the indices are inside the room and also if the adjacent room is dirty, we directly move to that room, also we mark the current room as visited.

Step 5: we print that all the rooms are clean.



→ Output:

Room = $\begin{bmatrix} D & C \\ D & D \end{bmatrix}$

· Started with (0,0) & cleaned it
Moved to (1,0) & cleaned it
Moved to (1,1) & cleaned it
Room is clean!

CODE (Vacuum cleaner) :

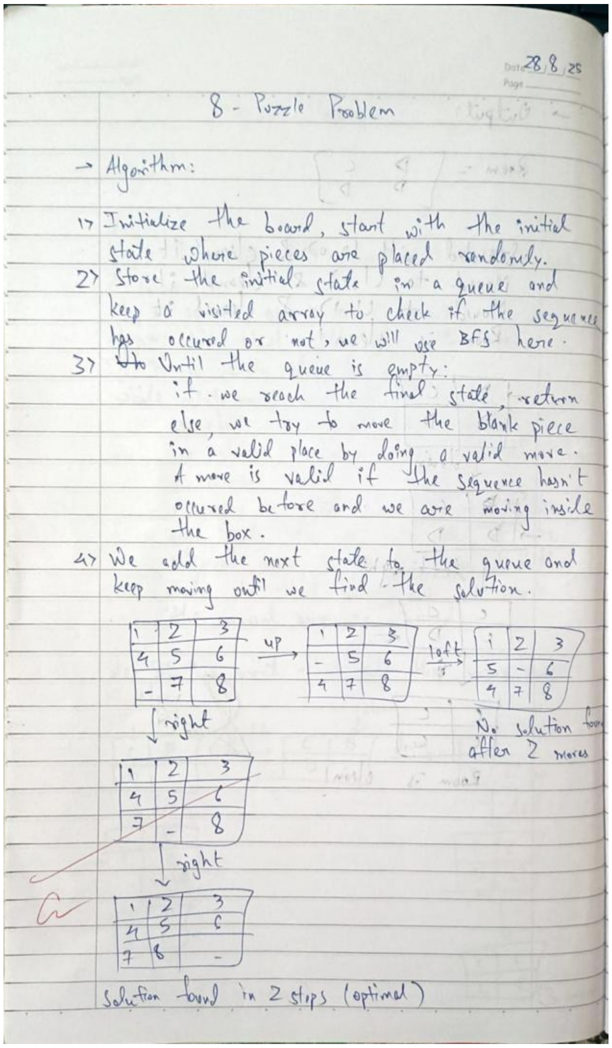
```
# Initialize room (2x2 grid), each cell could be 'dirty' or 'clean'
room = [
    ['dirty', 'clean'],
    ['dirty', 'dirty']
]
# Starting position
x, y = 0, 0

# Function to check if the current cell is dirty
def is_dirty(x, y):
    return room[x][y] == 'dirty'
```

```
# Function to clean the current cell
def clean(x, y):
   room[x][y] = 'clean'
print(f"Cleaned cell ({x},{y})")

# Function to move to a new position (up, down, left, or right)
def move_to(new_x, new_y):
   global x, y     x, y = new_x,
new_y     print(f"Moved to
({x},{y})")

# Function to check if all cells are
clean def all_clean():     for i in
range(2):         for j in range(2):
if room[i][j] == 'dirty':
            return False
return True

# Cleaning path: Visit each cell and clean (left to right, top to bottom)
# The allowed moves are up, down, left, right
cleaning_path = [(0, 0), (0, 1), (1, 1), (1, 0)]

# Start cleaning the room following the
path for cell in cleaning_path:
move_to(cell[0], cell[1])     if
is_dirty(cell[0], cell[1]):         clean(cell[0],
cell[1])

# Final check to see if the room is clean
if all_clean():
   print("Room is clean!") else:
print("Some cells are still dirty.")
```

OUTPUT :

```
Moved to (0,0)
Cleaned cell (0,0)
Moved to (0,1)
Moved to (1,1)
Cleaned cell (1,1)
Moved to (1,0)
Cleaned cell (1,0)
Room is clean!
```

Program 2: Implement 8-puzzle **ALGORITHM:**



Date 28/8/25

8 - Puzzle Problem

→ Algorithm:

1> Initialize the board, start with the initial state where pieces are placed randomly.
2> Store the initial state in a queue and keep a visited array to check if the sequence has occured or not, we will use BFS here.
3> ~~to~~ Until the queue is empty:
if we reach the final state, return
else, we try to move the blank piece in a valid place by doing a valid move.
A move is valid if the sequence hasn't occured before and we are moving inside the box.
4> We add the next state to the queue and keep moving until we find the solution.

1	2	3
4	5	6
-	7	8

up →

1	2	3
-	5	6
4	7	8

left →

1	2	3
5	-	6
4	7	8

No solution found after 2 moves

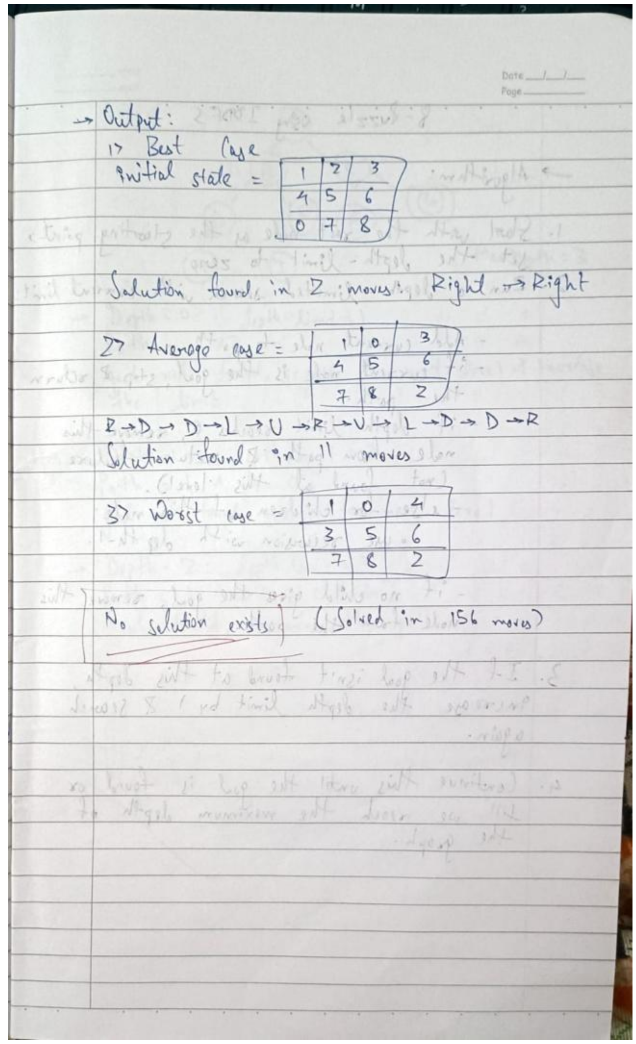
↓ right

1	2	3
4	5	6
7	-	8

↓ right

1	2	3
4	5	6
7	8	-

Solution found in 2 steps (optimal)



Date __/__/__

→ Output:
1> Best Case
initial state =

1	2	3
4	5	6
0	7	8

Solution found in 2 moves: Right → Right

2> Average case =

1	0	3
4	5	6
7	8	2

R → D → D → L → U → R → U → L → D → D → R
Solution found in 11 moves

3> Worst case =

1	0	4
3	5	6
7	8	2

No solution exists (Solved in 156 moves)

CODE:

```
from collections import deque import
copy

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

# Directions for moving the blank tile
DIRECTIONS = {
  'Up': (-1, 0),
  'Down': (1, 0),
  'Left': (0, -1),
'Right': (0, 1)
```

```
}

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

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

def get_neighbors(state):
neighbors = []
   x, y = find_blank(state)

   for move, (dx, dy) in DIRECTIONS.items():
      new_x, new_y = x + dx, y + dy
if is_valid(new_x, new_y):
         new_state = copy.deepcopy(state)          # Swap blank with adjacent tile
new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[x][y]
neighbors.append((move, new_state))     return neighbors

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

   while queue:
      current_state, path = queue.popleft()        state_tuple
= tuple(tuple(row) for row in current_state)

      if current_state == GOAL_STATE:
         return path  # Return the solution if we reach the goal state

      if state_tuple in visited:
continue        visited.add(state_tuple)
for move, neighbor in
get_neighbors(current_state):
         queue.append((neighbor, path + [move]))

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

```
solution = bfs(start_state)

# Since we removed "no solution" handling, it will keep exploring until it finds the solution
if solution:
    print("Solution found in", len(solution), "moves:")
print(" -> ".join(solution))
```

OUTPUT:

```
Solution found 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 3: Implement IDDFS

ALGORITHM :

8-Puzzle using IDDFS

→ Algorithm:

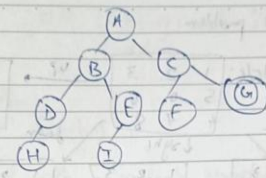
1. Start with the root node as the starting point, set the depth-limit to zero
2. Run a depth-limited search with current limit

   - Add current node to path
   - if current node is the goal, stop & return the path.
   - if depth limit reaches 0, remove this node from path & return failure. (not found at this level).
   - else for children of this node:
     - use recursion with depth-1.

   - if no child gives the goal, remove this node from the path.

3. If the goal isn't found at this depth, increase the depth limit by 1 & search again.

4. Continue this until the goal is found or till we reach the maximum depth of the graph.



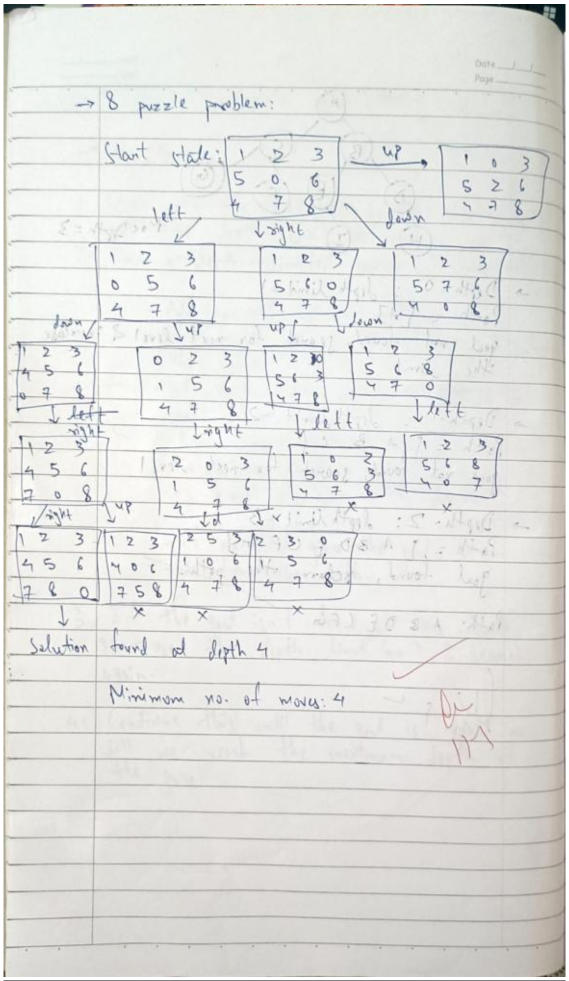
Max Depth = 3

→ Depth = 0 : depth limit = 1
Path = [A]
goal not found, search for next level & increase the limit

→ Depth-1 : depth limit = 2
Path = [A B C]
goal not found, search for next level

→ Depth-2 : depth limit = 3
Path = [A B D E C F G]
goal found, return the path.

Path: A B D E C F G



CODE :

```
from collections import deque

# Directions for movement
MOVES = {
   'Up': -3,
   'Down': 3,
   'Left': -1,
'Right': 1
}

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

# Valid indices for moves def
valid_moves(index):
```

```
moves = []     row, col =
divmod(index, 3)

    if row > 0: moves.append('Up')
if row < 2: moves.append('Down')
if col > 0: moves.append('Left')     if
col < 2: moves.append('Right')

    return moves

# Apply move to a state def
apply_move(state, move):
    idx = state.index(0)
    new_idx = idx + MOVES[move]

    # Special case for left/right edge wrapping
if move == 'Left' and idx % 3 == 0:
        return None     if move == 'Right'
and idx % 3 == 2:
        return None

    state = list(state)     state[idx], state[new_idx] =
state[new_idx], state[idx]     return tuple(state)

# DFS with depth limit def
dls(state, depth, visited, path):
if state == GOAL_STATE:
        return path

    if depth == 0:
return None

    visited.add(state)     for move in
valid_moves(state.index(0)):        next_state =
apply_move(state, move)        if next_state
and next_state not in visited:
            result = dls(next_state, depth - 1, visited.copy(), path + [(move, next_state)])
if result:                return result     return None

# IDDFS main function def
iddfs(start_state, max_depth=50):
for depth in range(max_depth):
```

```
        print(f"\n--- Iteration {depth + 1}: Depth Limit = {depth} ---")
visited = set()         path = dls(start_state, depth, visited, [])         if
path is not None:
            return path
return None

# Function to print the puzzle state in a readable format
def print_state(state):     for i in range(0, 9, 3):
print(state[i:i+3])     print()

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

    print("Initial State:")
print_state(start)

    solution = iddfs(start)
if solution:
        print(f"Solution found in {len(solution)} moves:")
current_state = start         for move, state in solution:
print(f"Move: {move}")             print_state(state)
current_state = state     else:
        print("No solution found.")
```

<u>OUTPUT :</u>

ALGORITHM:

```
Initial State:
(1, 2, 3)
(4, 5, 6)
(0, 7, 8)


--- Iteration 1: Depth Limit = 0 ---

--- Iteration 2: Depth Limit = 1 ---

--- Iteration 3: Depth Limit = 2 ---
Solution found in 2 moves:
Move: Right
(1, 2, 3)
(4, 5, 6)
(7, 0, 8)

Move: Right
(1, 2, 3)
(4, 5, 6)
(7, 8, 0)
```

Program 4: Implement Hill Climbing & Simulated Annealing

ALGORITHM:

Date 9/10/25

Page

Hill Climbing and Simulated Annealing

→ Algorithm:

- Hill Climbing

i> Start with a random'y generated solution (initial solution).

ii> Repeat until termination:

- Generate the neighbouring solution using the objective function.
- If any neighbour has a better solution than the current solution, move to that neighbour.
- Keep repeating until no better solution is found.

iii> Finally return the best found solution.

Pseudocode

```
function hill climb(problem):
    current = initial_state(problem)
    loop:
        neighbours = generate(current)
        b = best(neighbour)
        if value(b) > value(current):
                current = b
        else:
            break
    return current
```

Date __/__/__

Page

- N Queens using Hill Climb

1. Start with a random board.
2. Generate neighbours by moving a queen to every other row.
3. Choose the neighbour with the least no. of pair of queens attacking each other.
4. Move to the best neighbour if it exists else stop.
5. Continue until we reach optimal solution.
6. Return the best found solution.

→ Output:

Initial board:

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

Move to board with 1 attacks:

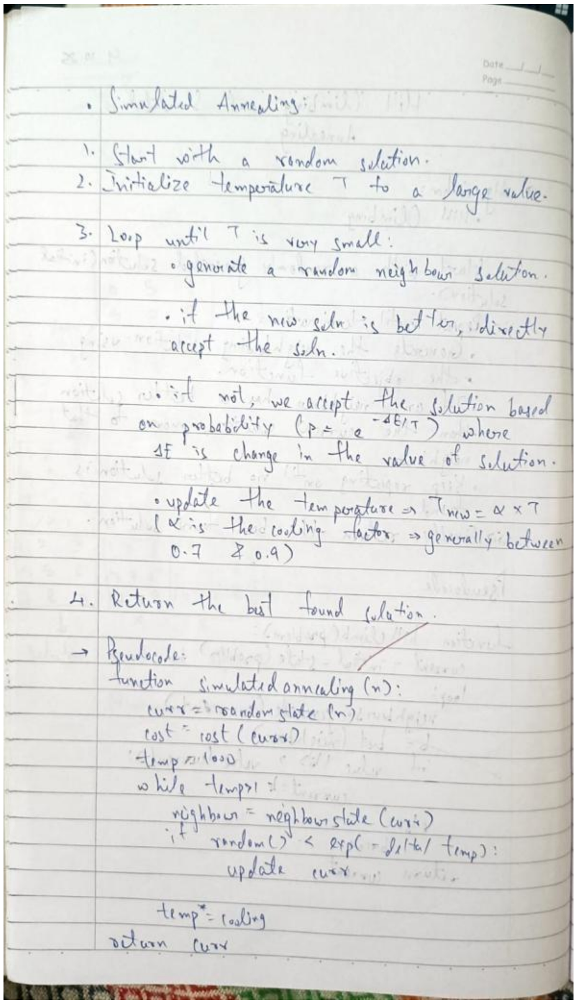
```
Q . Q .
. . . .
. . . Q
. Q . .
```

Move to board with 0 attacks:

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

Solution found:

Steps taken = 2



• Simulated Annealing:

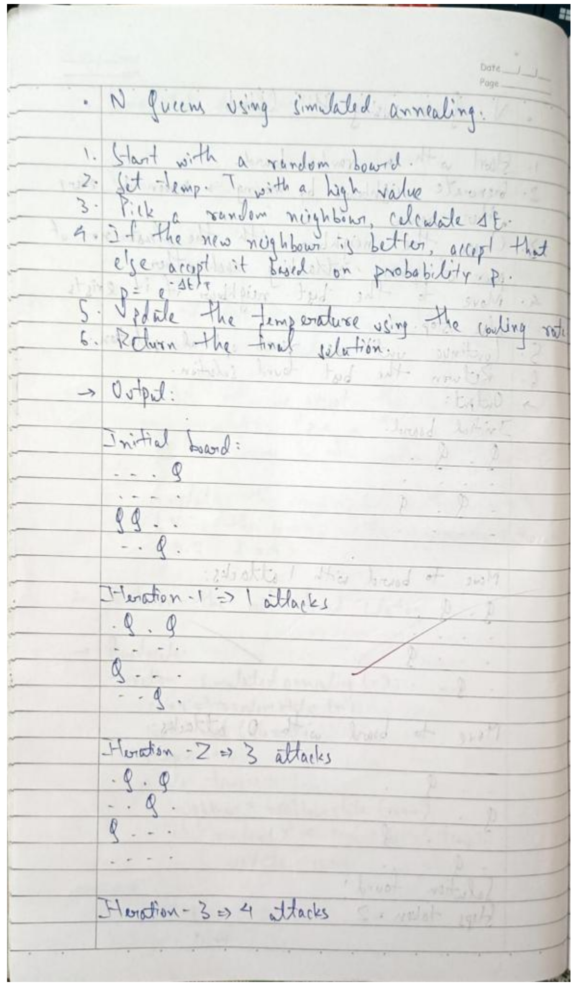
1. Start with a random solution.
2. Initialize temperature T to a large value.

3. Loop until T is very small:
   - generate a random neighbour solution.
   - if the new soln is better, directly accept the soln.
   - if not, we accept the solution based on probability ($p = e^{-\Delta E/T}$) where ΔE is change in the value of solution.
   - update the temperature ⇒ $T_{new} = \alpha \times T$ (α is the cooling factor ⇒ generally between 0.7 & 0.9)

4. Return the best found solution.

→ Pseudocode:

```
function simulatedannealing (n):
    curr = random state (n)
    cost = cost (curr)
    temp = 1000
    while temp>1 :
        neighbour = neighbourstate (curr)
        if random() < exp( - delta/ temp):
            update curr

        temp *= cooling
    return curr
```



• N Queens using simulated annealing:

1. Start with a random board.
2. Set temp. T with a high value
3. Pick a random neighbour, calculate ΔE.
4. If the new neighbour is better, accept that else accept it based on probability p. $p = e^{-\Delta E/T}$
5. Update the temperature using the cooling rate
6. Return the final solution.

→ Output:

Initial board:

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

Iteration - 1 ⇒ 1 attacks

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

Iteration - 2 ⇒ 3 attacks

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

Iteration - 3 ⇒ 4 attacks

CODE (Hill Climbing) :

```
import random def
heuristic(board):
    h = 0     n = len(board)     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):
                h += 1
return h

def hill_climbing_restart(initial_board,
max_restarts=100):     N = len(initial_board)     board = [x-
1 for x in initial_board]  # 0-based      h = heuristic(board)

    restart_count = 0
    while h != 0 and restart_count < max_restarts:
        steps = 0          while True:
best_board = board[:]             best_h = h
```

```
for col in range(N):                for row in
range(N):                   if row != board[col]:
neighbor = board[:]
neighbor[col] = row                         h_neighbor
= heuristic(neighbor)                        if
h_neighbor < best_h:
best_board = neighbor                            best_h
= h_neighbor            steps += 1            if
best_h >= h:  # stuck                 break
board = best_board            h = best_h            if
h == 0:                break         if h == 0:
          print(f"Solution found after {restart_count} restarts and {steps} steps.")
break
       # Random restart         board = [random.randint(0,
N-1) for _ in range(N)]        h = heuristic(board)
restart_count += 1

    return [x+1 for x in board], h

# User input
N = int(input("Enter number of queens (N): ")) print(f"Enter the initial
positions of {N} queens (row numbers 1 to {N}):") initial_board =
list(map(int, input().split()))

solution, h_val =
hill_climbing_restart(initial_board) print("Final
board:", solution) print("Heuristic H =", h_val)
```

OUTPUT (Hill Climbing) :

```
Enter number of queens (N): 4
Enter the initial positions of 4 queens (row numbers 1 to 4):
3 4 1 2
Solution found after 0 restarts and 3 steps.
Final board: [2, 4, 1, 3]
Heuristic H = 0

=== Code Execution Successful ===
```

CODE (Simulated Annealing):

```
from datetime import datetime
import random, time, math from
copy import deepcopy, copy
import decimal
```

```
class Board:     def __init__(self,
queen_count=4):
self.queen_count = queen_count
        self.reset()

    def reset(self):
        self.queens = [-1 for i in range(0, self.queen_count)]

        for i in range(0, self.queen_count):
            self.queens[i] = random.randint(0, self.queen_count - 1)
# self.queens[row] = column


    def calculateCost(self):
        threat = 0

        for queen in range(0, self.queen_count):             for
next_queen in range(queen+1, self.queen_count):
                if self.queens[queen] == self.queens[next_queen] or abs(queen - next_queen) ==
abs(self.queens[queen] - self.queens[next_queen]):
threat += 1

        return threat

    @staticmethod     def
calculateCostWithQueens(queens):
threat = 0
        queen_count = len(queens)

        for queen in range(0, queen_count):             for
next_queen in range(queen+1, queen_count):
                if queens[queen] == queens[next_queen] or abs(queen - next_queen) ==
abs(queens[queen] - queens[next_queen]):
                    threat += 1

        return threat

    @staticmethod
def toString(queens):
board_string = ""

        for row, col in enumerate(queens):
            board_string += "(%s, %s)\n" % (row, col)
```

```
        return board_string

    def getLowerCostBoard(self):
displacement_count = 0         temp_queens =
self.queens         lowest_cost =
self.calculateCost(temp_queens)

        for i in range(0, self.queen_count):
            temp_queens[i] = (temp_queens[i] + 1) % (self.queen_count - 1)

            for j in range(queen+1, self.queen_count):
                temp_queens[j] = (temp_queens[j] + 1) % (self.queen_count - 1)

    def __str__(self):
board_string = ""

        for row, col in enumerate(self.queens):
            board_string += "(%s, %s)\n" % (row, col)

        return board_string

class SimulatedAnnealing:
def __init__(self, board):
self.elapsedTime = 0;
self.board = board
self.temperature = 4000
self.sch = 0.99
self.startTime =
datetime.now()


    def run(self):
        board = self.board
board_queens = self.board.queens[:]
solutionFound = False

        for k in range(0, 170000):
            self.temperature        *=        self.sch
board.reset()                successor_queens =
board.queens[:]
            dw = Board.calculateCostWithQueens(successor_queens) -
Board.calculateCostWithQueens(board_queens)             exp =
decimal.Decimal(decimal.Decimal(math.e) ** (decimal.Decimal(-dw) *
```

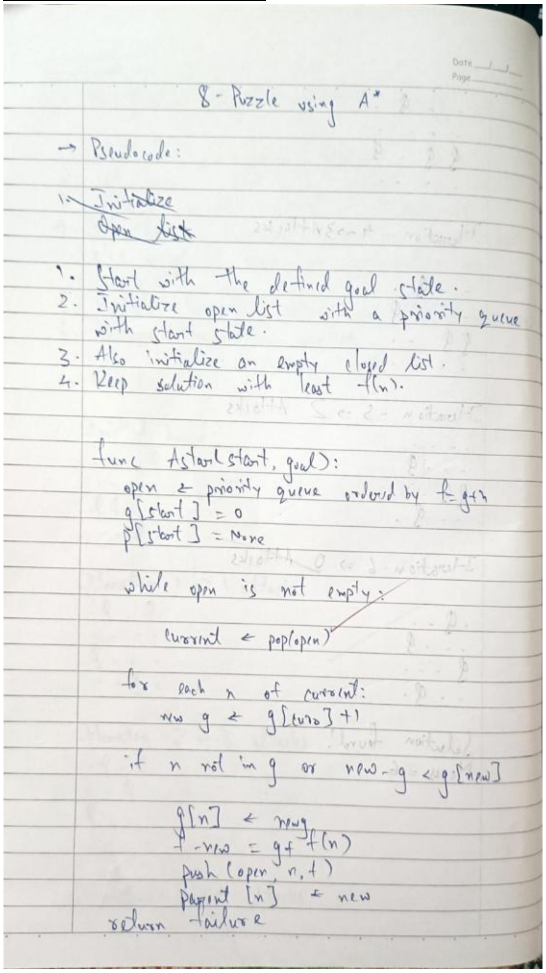
```
decimal.Decimal(self.temperature)))

            if dw > 0 or random.uniform(0, 1) < exp:
board_queens = successor_queens[:]

            if Board.calculateCostWithQueens(board_queens) == 0:
print("Solution:")                print(Board.toString(board_queens))
self.elapsedTime = self.getElapsedTime()                print("Success,
Elapsed Time: %sms" % (str(self.elapsedTime)))                solutionFound
= True                break

        if solutionFound == False:
            self.elapsedTime = self.getElapsedTime()            print("Unsuccessful,
Elapsed Time: %sms" % (str(self.elapsedTime)))

        return self.elapsedTime

    def getElapsedTime(self):        endTime = datetime.now()
elapsedTime = (endTime - self.startTime).microseconds / 1000
return elapsedTime


if __name__ == '__main__':
    board = Board()
print("Board:")    print(board)
SimulatedAnnealing(board).r
un()
```

OUTPUT (Simulated Annealing):

```
Board:
(0, 1)
(1, 0)
(2, 1)
(3, 3)

Solution:
(0, 2)
(1, 0)
(2, 3)
(3, 1)

Success, Elapsed Time: 7.9ms
```

Program 5: A* Algorithm

ALGORITHM :



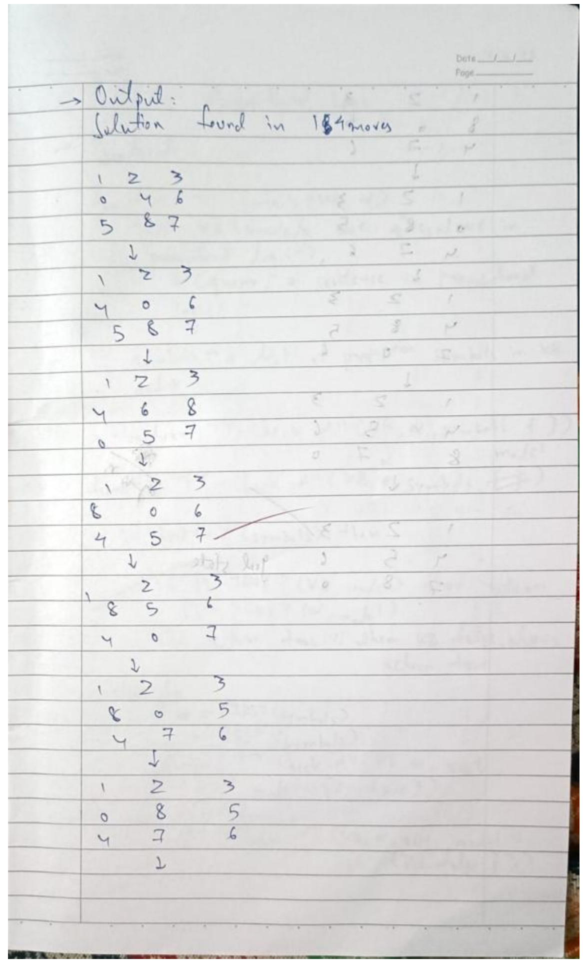
8 - Puzzle using A*

→ Pseudocode:

~~1. Initialize Open list~~

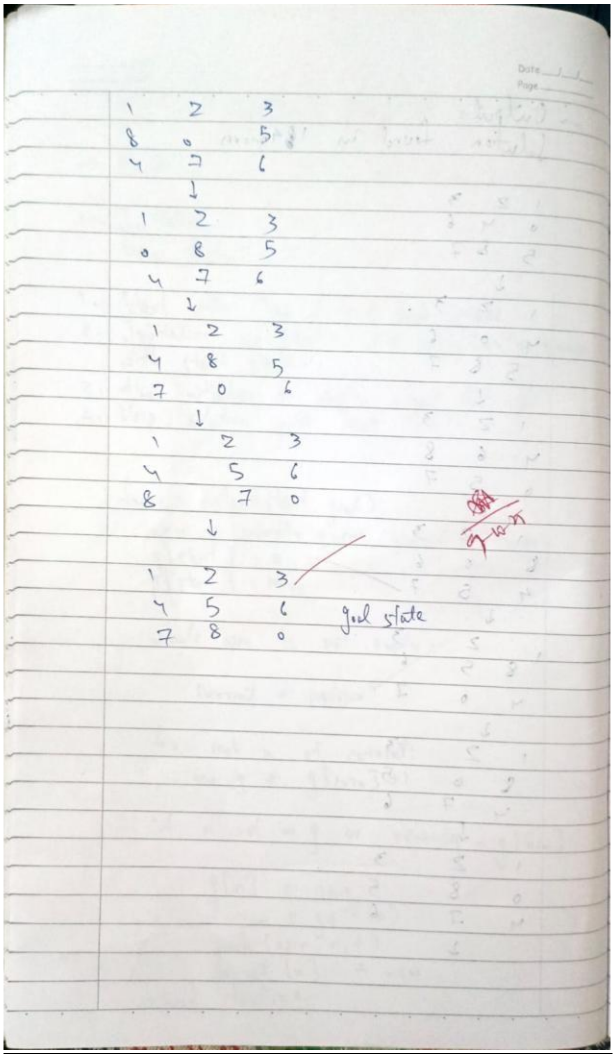
1. Start with the defined goal state.
2. Initialize open list with a priority queue with start state.
3. Also initialize an empty closed list.
4. Keep solution with least f(n).

```
func Astar(start, goal):
    open ← priority queue ordered by f=g+h
    g[start] = 0
    p[start] = None

    while open is not empty:

        current ← pop(open)

    for each n of current:
        new g ← g[curr] + 1

    if n not in g or new-g < g[new]

        g[n] ← newg
        f-new = g + f(n)
        push (open, n, f)
        parent [n] ← new
  return failure
```



→ Output:
Solution found in 184 moves

```
1 2 3
0 4 6
5 8 7
  ↓
1 2 3
4 0 6
5 8 7
  ↓
1 2 3
4 6 8
0 5 7
  ↓
1 2 3
8 0 6
4 5 7
  ↓
1 2 3
8 5 6
4 0 7
  ↓
1 2 3
8 0 5
4 7 6
  ↓
1 2 3
0 8 5
4 7 6
  ↓
```


1 2 3
8 0 5
4 7 6
↓
1 2 3
0 8 5
4 7 6
↓
1 2 3
4 8 5
7 0 6
↓
1 2 3
4 5 6
8 7 0
↓
1 2 3
4 5 6 goal state
7 8 0

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(opene
d, 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* (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

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
```

Program 6: Propositional Logic

ALGORITHM :

Date 16/10/25

Prepositional Logic

→ Pseudocode

```
function TT-Entails (KB, α)
    inputs: KB (knowledge base, a sentence in
      propositional logic),
        α (query, a sentence in propositional
        logic)

    symbols → a list of proposition symbols in KB
    and α

    return TT-Check-All (KB, α, symbols, { })

function TT-Check-All (KB, α, symbols, model)

  if EMPTY? (symbols) then

      if PL-TRUE? (KB, model) then return
          PL-TRUE? (α, model)
      else return true // when KB false, always
                            return 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. i) $Q \to P$
ii) $P \to \neg Q$
iii) $Q \vee R$

- Truth Table

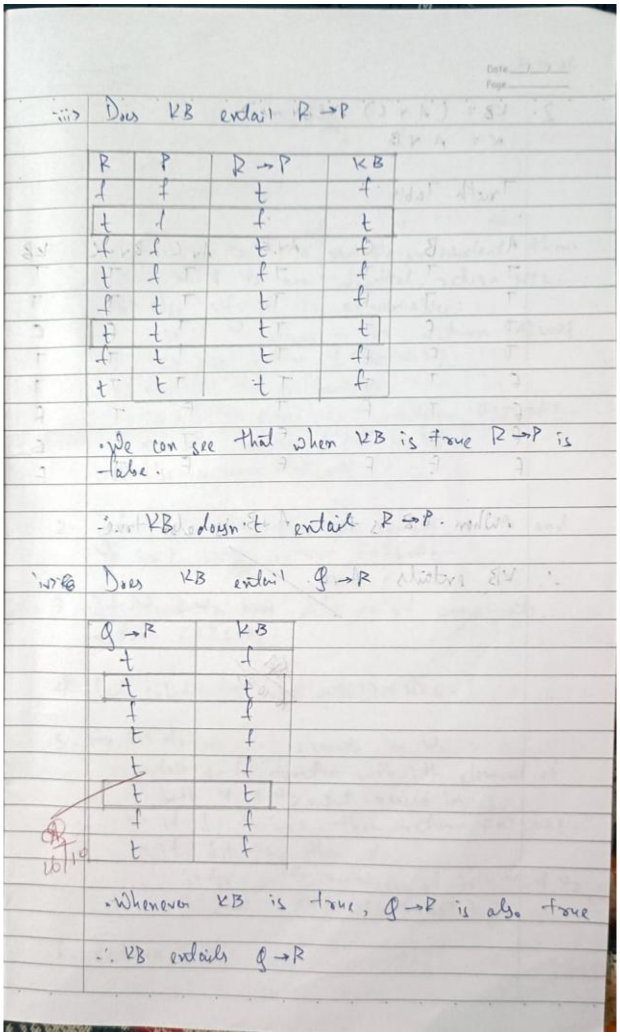
P	Q	R	$\neg Q$	$Q \to P$	$P \to \neg Q$	$Q \vee R$	KB
f	f	f	t	t	t	f	f
f	f	t	t	t	t	t	t
f	t	f	f	f	t	t	f
f	t	t	f	f	t	t	f
t	f	f	t	t	t	f	f
t	f	t	t	t	t	t	t
t	t	f	f	t	f	t	f
t	t	t	f	t	f	t	f

ii) Does KB entail R

R	KB
f	f
t	t
f	f
t	f
f	f
t	t
f	f
t	f

- Whenever KB is true, R is also true.

∴ KB entails R



iii) Does KB entail R → P

R	P	R → P	KB
f	f	t	f
t	f	f	t
f	f	t	f
t	f	f	f
f	t	t	f
t	t	t	t
f	t	t	f
t	t	t	f

• We can see that when KB is true R → P is false.

∴ KB doesn't entail R → P.

iv) Does KB entail Q → R

Q → R	KB
t	f
t	t
f	f
t	f
t	f
t	t
f	f
t	f

10/10

• Whenever KB is true, Q → R is also true

∴ KB entails Q → R

CODE :

```
import itertools

# Logical operations
def implies(a, b):
return not a or b

def or_operator(a, b):
    return a or b

def not_operator(a):
    return not a

# Constructing the truth table
def construct_truth_table():
```

```
truth_values = [True, False]
truth_table = []

   # Generate all combinations for Q, P, R     for values
in itertools.product(truth_values, repeat=3):
      Q, P, R = values

      # Evaluate KB sentences        q_implies_p =
implies(Q, P)        p_implies_not_q = implies(P,
not_operator(Q))        q_or_r = or_operator(Q, R)

      # KB = (Q → P) ∧ (P → ¬Q) ∧ (Q ∨ R)
      kb_is_true = q_implies_p and p_implies_not_q and q_or_r

      # Entailment expressions
entail_r = R        entail_r_implies_p =
implies(R, P)        entail_q_implies_r =
implies(Q, R)

      # Add row to truth table        truth_table.append((
Q, P, R,            q_implies_p, p_implies_not_q, q_or_r,
kb_is_true,            entail_r, entail_r_implies_p,
entail_q_implies_r
      ))
   return truth_table

# Print the truth table nicely def
print_truth_table(truth_table):
   header = [
      "Q", "P", "R",
      "Q → P", "P → ¬Q", "Q ∨ R",
      "KB (all true)",
      "R", "R → P", "Q → R"
   ]
   print(" | ".join(header))
print("-" * 85)

   for row in truth_table:
      # Format True/False as T/F for compactness
formatted_row = [("T" if val else "F") for val in row]
print(" | ".join(formatted_row))
```

```
# Generate and print truth table
truth_table = construct_truth_table()
print_truth_table(truth_table)

# Additionally, check entailment by verifying if for all models where KB is true, entailment is true

def check_entailment(truth_table,
entailment_index):     for row in truth_table:
kb_true = row[6]         entailment_val =
row[entailment_index]         if kb_true and not
entailment_val:
            return False
return True

print("\nEntailment Results:") print(f"Does KB entail R?
{check_entailment(truth_table, 7)}") print(f"Does KB entail R → P?
{check_entailment(truth_table, 8)}") print(f"Does KB entail Q →
R? {check_entailment(truth_table, 9)}")
```

OUTPUT :

```
Q | P | R | Q → P | P → ¬Q | Q ∨ R | KB (all true) | R | R → P | Q → R
-----------------------------------------------------------------------------------
T | T | T | T | F | T | F | T | T | T
T | T | F | T | F | T | F | F | T | F
T | F | T | F | T | T | F | T | F | T
T | F | F | F | T | T | F | F | T | F
F | T | T | T | T | T | T | T | T | T
F | T | F | T | T | F | F | F | T | T
F | F | T | T | T | T | T | T | F | T
F | F | F | T | T | F | F | F | T | T

Entailment Results:
Does KB entail R? True
Does KB entail R → P? False
Does KB entail Q → R? True
```

Program 7: Unification **Algorithm:**

Date 30.10.25

Unification

→ Algorithm
Unify (Ψ_1, Ψ_2)

1. If Ψ_1 or Ψ_2 is a variable or constant, then:
   a) if Ψ_1 & Ψ_2 are identical, return NIL.
   b) else if Ψ_1 is a variable:
      i) if Ψ_1 occurs in Ψ_2, return FAILURE.
      ii) else return $\{(\Psi_2/\Psi_1)\}$
   c) else if Ψ_2 is variable:
      i) if Ψ_2 occurs in Ψ_1, return FAILURE.
      ii) else return $\{(\Psi_1/\Psi_2)\}$
   d) else return FAILURE.

2. If the initial predicate symbol in Ψ_1 and Ψ_2 isn't same, return FAILURE.

3. If Ψ_1 & Ψ_2 have diff no. of arguements, return FAILURE.

4. Set substitution set(SUBST) = NIL.

5. For i=1 to no. of elements in Ψ_1
   a) Call unify function with ith element of both Ψ_1 & Ψ_2, put result in S.
   b) if S = failure then return FAILURE.
   c) if S ≠ NIL, then do
      i) Apply S to remainder of both Ψ_1 & Ψ_2
      ii) SUBST = APPEND(S, SUBST)

6. Return SUBST

Date __/__/__
Page ____

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

i) $f(x)$ and $f(g(z))$
$x \rightarrow g(z)$ (as both have same predicate)

∴ we replaced x with $g(z)$

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

ii) $g(y)$ and $g(f(a))$

$y \rightarrow f(a)$ (as both have same predicate)

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

$P(f(g(z)), g(f(a)), f(a))$
(substitute y here as well)
$P(f(g(z)), g(f(a)), f(a))$

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

2. $g(x, f(x))$ and $g(f(y), y)$

i) x and $f(y)$

∴ $\theta = \{x/f(y)\}$

$g(f(y), f(f(y)))$ and $g(f(y), y)$

ii) $f(f(y))$ and y
∵ y occurs on both sides

∴ Not unifiable

3. $P(x, g(x))$ and $P(g(y), g(g(z)))$

i) x and $g(y)$

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

$P(g(y), g(g(y)))$ & $P(g(y), g(g(z)))$

ii) $g(g(y))$ and $g(g(z))$

replace y with z

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

CODE :

```
 class Variable:     def
__init__(self, name):
self.name = name

    def __eq__(self, other):
        return isinstance(other, Variable) and self.name == other.name

    def __hash__(self):
        return hash(self.name)

    def __repr__(self):
return self.name

class Constant:     def
__init__(self, value):
self.value = value
```

```
    def __eq__(self, other):
        return isinstance(other, Constant) and self.value == other.value

    def __hash__(self):
        return hash(self.value)

    def __repr__(self):
return str(self.value)

class Function:     def
__init__(self, name, args):
        self.name = name
self.args = args

    def __eq__(self, other):
        return (isinstance(other, Function) and
self.name == other.name and                len(self.args) ==
len(other.args) and               all(a == b for a, b in
zip(self.args, other.args)))

    def __hash__(self):
        return hash((self.name, tuple(self.args)))

    def __repr__(self):
        return f"{self.name}({', '.join(map(str, self.args))})"

def unify(term1, term2, substitution=None):
    """
    Unifies two first-order logic terms and returns the MGU (substitution)
or None if unification is not possible.
    """     if substitution is
None:        substitution
= {}

    # Apply existing substitutions
term1 = substitute(term1, substitution)
term2 = substitute(term2, substitution)

    if term1 == term2:        return
substitution     elif
isinstance(term1, Variable):
        return unify_var(term1, term2, substitution)
elif isinstance(term2, Variable):
```

```
        return unify_var(term2, term1, substitution)     elif
isinstance(term1, Function) and isinstance(term2, Function):         if
term1.name != term2.name or len(term1.args) != len(term2.args):
            return None  # Function symbols or arity don't
match         for arg1, arg2 in zip(term1.args, term2.args):
substitution = unify(arg1, arg2, substitution)             if
substitution is None:
                return None  # Sub-unification
failed         return substitution     else:
        return None  # Cannot unify different types (e.g., Constant and Function)

def unify_var(var, x, substitution):
    """Handles unification when one of the terms is a variable."""
if var in substitution:
        return unify(substitution[var], x, substitution)
elif x in substitution:
        return unify(var, substitution[x],
substitution)     elif occurs_check(var, x,
substitution):         return None  # Occurs check
fails     else:
        substitution[var] = x
return substitution

def occurs_check(var, term, substitution):
    """Checks if a variable occurs within a term, preventing infinite
substitutions."""     term = substitute(term, substitution)  # Apply current
substitutions     if var == term:         return True     elif isinstance(term, Function):
        return any(occurs_check(var, arg, substitution) for arg in term.args)
return False

def substitute(term, substitution):
    """Applies a given substitution to a term."""
if isinstance(term, Variable):
        return substitution.get(term, term)
elif isinstance(term, Function):
        return Function(term.name, [substitute(arg, substitution) for arg in term.args])
return term

# Example Usage: if
__name__ == "__main__":
    # Define terms
    x, y, z = Variable('x'), Variable('y'), Variable('z')
a, b = Constant('a'), Constant('b')     f =
```

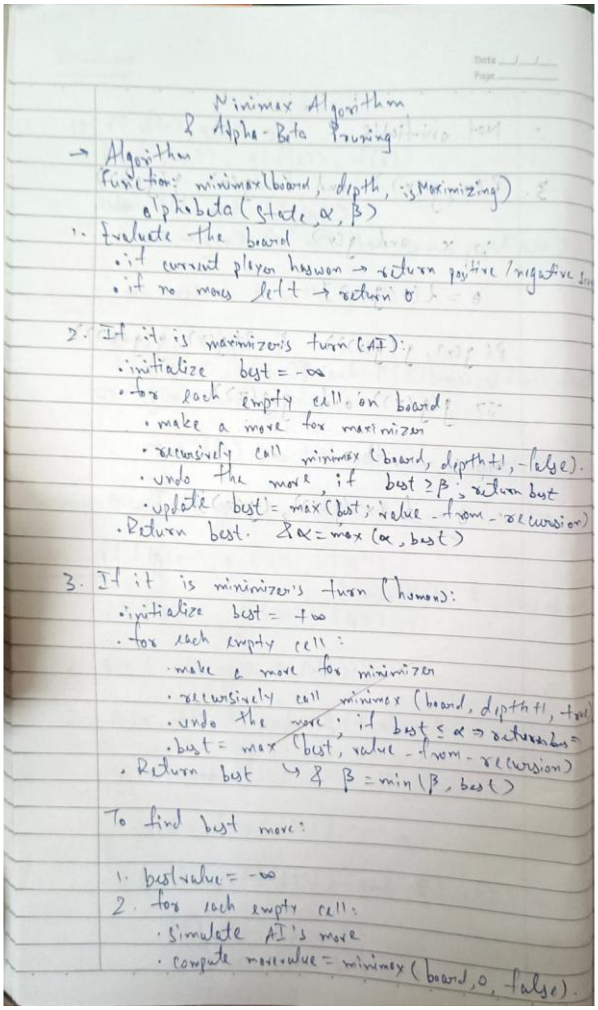
```
Function('f', [x, Constant('b')])     g = Function('g',
[Constant('a'), y])     h = Function('h', [z])

    print(f"Unify(f(x, b), f(a, y)): {unify(Function('f', [x, b]), Function('f', [a, y]))}")
print(f"Unify(g(a, y), g(a, b)): {unify(Function('g', [a, y]), Function('g', [a, b]))}")
print(f"Unify(x, f(x, b)): {unify(x, Function('f', [x, b]))}") # Occurs check failure
print(f"Unify(f(x, y), f(a, g(z))): {unify(Function('f', [x, y]), Function('f', [a, Function('g',
[z])]))}")     print(f"Unify(P(x, A), P(B, y)): {unify(Function('P', [x, Constant('A')]),
Function('P', [Constant('B'), y]))}")

    print("\n--- Your Requested Tests ---")

    # 1. p(f(x), g(y), y) and p(f(g(z)), g(f(a)), f(a))     term1_1 =
Function('p', [Function('f', [x]), Function('g', [y]), y])
    term1_2 = Function('p', [Function('f', [Function('g', [z])]), Function('g', [Function('f', [a])]),
Function('f', [a])])     print(f"Unify(p(f(x), g(y), y), p(f(g(z)), g(f(a)), f(a))): {unify(term1_1,
term1_2)}")

    # 2. q(x, f(x)) and q(f(y), y)     term2_1 = Function('q', [x,
Function('f', [x])])     term2_2 = Function('q', [Function('f', [y]), y])
print(f"Unify(q(x, f(x)), q(f(y), y)): {unify(term2_1, term2_2)}")

    # 3. p(x, g(x)) and p(g(y), g(g(z)))     term3_1 = Function('p', [x,
Function('g', [x])])     term3_2 = Function('p', [Function('g', [y]), Function('g',
[Function('g', [z])])])     print(f"Unify(p(x, g(x)), p(g(y), g(g(z)))):
{unify(term3_1, term3_2)}")
```

OUTPUT :

```
Unify(f(x, b), f(a, y)): {x: a, y: b}
Unify(g(a, y), g(a, b)): {y: b}
Unify(x, f(x, b)): None
Unify(f(x, y), f(a, g(z))): {x: a, y: g(z)}
Unify(P(x, A), P(B, y)): {x: B, y: A}

--- Your Requested Tests ---
Unify(p(f(x), g(y), y), p(f(g(z)), g(f(a)), f(a))): {x: g(z), y: f(a)}
Unify(q(x, f(x)), q(f(y), y)): None
Unify(p(x, g(x)), p(g(y), g(g(z)))): {x: g(y), y: z}
```

Program 8: MinMax & AlphaBeta

ALGORITHM :



Minimax Algorithm
& Alpha-Beta Pruning

→ Algorithm

Function: minimax(board, depth, isMaximizing)
alphabeta(state, α, β)

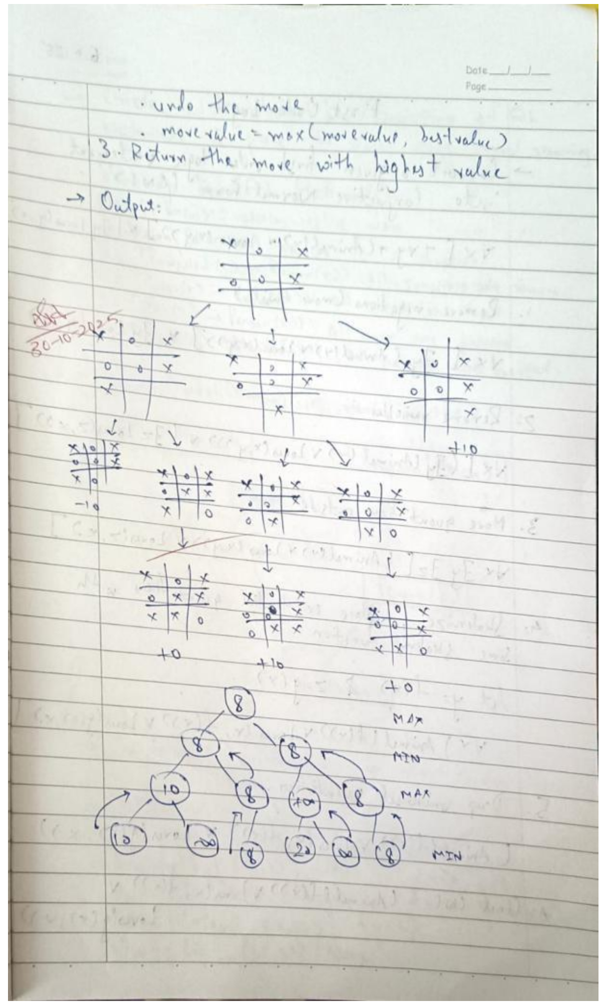
1. Evaluate the board
   - if current player has won → return positive / negative score
   - if no moves left → return 0

2. If it is maximizer's turn (AI):
   - initialize best = -∞
   - for each empty cell on board:
     - make a move for maximizer
     - recursively call minimax (board, depth+1, false).
     - undo the move, if best ≥ β, return best
     - update best = max(best, value_from_recursion)
   - Return best. & α = max (α, best)

3. If it is minimizer's turn (human):
   - initialize best = +∞
   - for each empty cell:
     - make a move for minimizer
     - recursively call minimax (board, depth+1, true)
     - undo the move; if best ≤ α ⇒ return best
     - best = max (best, value_from_recursion)
   - Return best & β = min(β, best)

To find best move:

1. bestvalue = -∞
2. for each empty cell:
   - Simulate AI's move
   - compute movevalue = minimax (board, 0, false).



CODE :

```
def alpha_beta(node, depth, alpha, beta, maximizing_player, path):
# Base case: if node is a leaf (integer), return its value and path
if isinstance(node, int):
        return node, path

    if maximizing_player:
value = float('-inf')
best_path = None          for i,
child in enumerate(node):
            child_value, child_path = alpha_beta(
                child, depth + 1, alpha, beta, False, path + [i]
            )
```

```
            if child_value > value:
value = child_value
best_path = child_path

            # Artifact '62' removed here
alpha = max(alpha, value)

            if alpha >= beta:
                print(f"  [PRUNE] MAX (Depth {depth}): Alpha ({alpha}) >= Beta ({beta})")
break

        return value, best_path

else:
        value = float('inf')
best_path = None

        for i, child in enumerate(node):
            child_value, child_path = alpha_beta(
child, depth + 1, alpha, beta, True, path + [i]
            )

            if child_value < value:
value = child_value
best_path = child_path

            beta = min(beta, value)

            if beta <= alpha:
                print(f"  [PRUNE] MIN (Depth {depth}): Beta ({beta}) <= Alpha ({alpha})")
break

        return value, best_path

if __name__ == "__main__":
    # Tree structure with artifact '63' removed
tree = [
        [
            [
                [10, 11],
                [9, 12]
            ],
            [
                [14, 15],
```

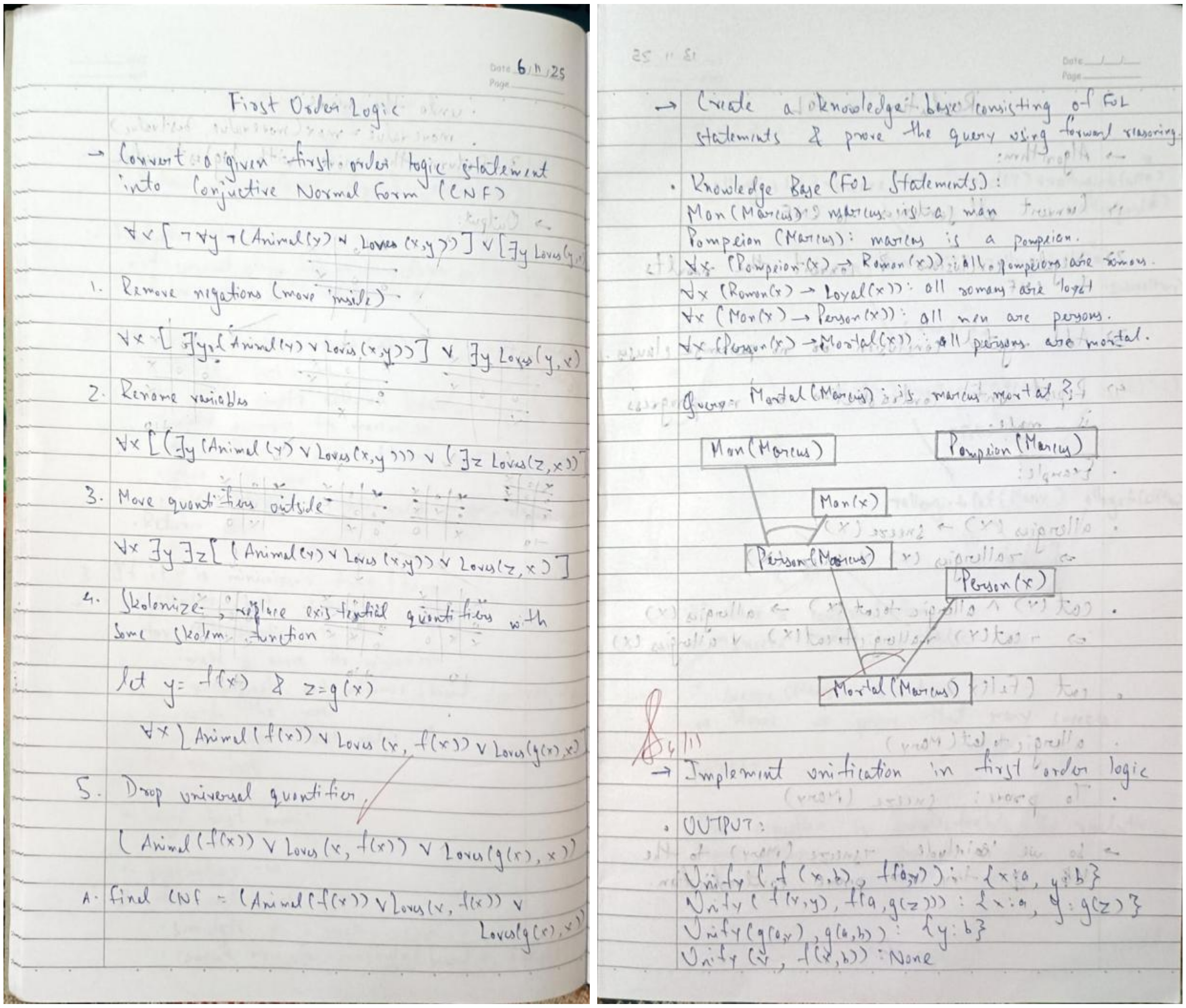
```
            [13, 14]
        ],
    ],
    [
        [
            [5, 2],
            [4, 1]
        ],
        [
            [3, 22],
            [20, 21]
        ],
    ]
]

print("Starting Alpha-Beta Pruning...\n" + "-"*30)

value, best_path = alpha_beta(tree, 0, float('-inf'), float('inf'), True, [])

print("-" * 30)
print(f"FINAL MINIMAX VALUE AT ROOT: {value}")
print(f"BEST PATH INDICES: {best_path}")
```

OUTPUT :

```
Starting Alpha-Beta Pruning...
------------------------------
  [PRUNE] MIN (Depth 3): Beta (9) <= Alpha (10)
  [PRUNE] MAX (Depth 2): Alpha (14) >= Beta (10)
  [PRUNE] MIN (Depth 3): Beta (5) <= Alpha (10)
  [PRUNE] MIN (Depth 3): Beta (4) <= Alpha (10)
  [PRUNE] MIN (Depth 1): Beta (5) <= Alpha (10)
------------------------------
FINAL MINIMAX VALUE AT ROOT: 10
BEST PATH INDICES: [0, 0, 0, 0]
```

Program 9: Forward Chaining & Conversion to CNF

ALGORITHM :


Date 6/11/25

First Order Logic

→ Convert a given first order logic statement into Conjunctive Normal Form (CNF)

∀x [¬∀y ¬(Animal(y) ∧ Loves(x,y))] ∨ [∃y Loves(y,x)]

1. Remove negations (move inside)

∀x [∃y(Animal(y) ∨ Loves(x,y))] ∨ ∃y Loves(y,x)

2. Rename variables

∀x [(∃y (Animal(y) ∨ Loves(x,y))) ∨ (∃z Loves(z,x))]

3. Move quantifiers outside

∀x ∃y ∃z [(Animal(y) ∨ Loves(x,y)) ∨ Loves(z,x)]

4. Skolemize, replace existential quantifiers with some skolem function

let y = f(x) & z = g(x)

∀x [Animal(f(x)) ∨ Loves(x, f(x)) ∨ Loves(g(x),x)]

5. Drop universal quantifier

(Animal(f(x)) ∨ Loves(x, f(x)) ∨ Loves(g(x), x))

A. Final CNF = (Animal(f(x)) ∨ Loves(x, f(x)) ∨ Loves(g(x),x))

→ Create a knowledge base consisting of FOL statements & prove the query using forward reasoning.

• Knowledge Base (FOL Statements):

Man(Marcus): marcus is a man.

Pompeian(Marcus): marcus is a pompeian.

∀x (Pompeian(x) → Roman(x)): all pompeians are romans.

∀x (Roman(x) → Loyal(x)): all romans are loyal.

∀x (Man(x) → Person(x)): all men are persons.

∀x (Person(x) → Mortal(x)): all persons are mortal.

Query: Mortal(Marcus) i.e. is marcus mortal?

8/11

→ Implement unification in first order logic

• OUTPUT:

Unify(f(x,b), f(a,y)): {x: a, y: b}

Unify(f(x,y), f(a,g(z))): {x: a, y: g(z)}

Unify(g(a,x), g(a,b)): {y: b}

Unify(x, f(x,b)): None

CODE (Forward Chaining) :

```
import re

def isVariable(x):
    return len(x) == 1 and x.islower() and x.isalpha()

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

def getPredicates(string):
    expr = r'([a-zA-Z~]+)\([^&|]+\)'
    return re.findall(expr, string)
```

```
class Fact:     def __init__(self,
expression):
        self.expression = expression.strip()         predicate,
params = self.splitExpression(expression)
self.predicate = predicate         self.params = params

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

    def substitute(self, var_map):
        params = [var_map.get(p, p) for p in self.params]
return Fact(f"{self.predicate}({','.join(params)})")

    def __repr__(self):
return self.expression

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

    def infer(self, known_facts):
        substitutions = {}

        for fact in self.lhs:             matched =
False             for known in known_facts:
if known.predicate == fact.predicate:
                    mapping = {}                     for i, param
in enumerate(fact.params):                         if
isVariable(param):
                            mapping[param] =
known.params[i]                         elif param !=
known.params[i]:                             break
else:
                        substitutions.update(mapping)
                        matched = True
break             if not matched:
return None

        return self.rhs.substitute(substitutions)
```

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

    def tell(self, expr):
if '=>' in expr:
            self.implications.add(Implication(expr))
else:
            self.facts.add(Fact(expr))

    def infer_all(self):         added =
True         while added:
added = False             for rule in
self.implications:
                new_fact = rule.infer(self.facts)                 if new_fact and
new_fact.expression not in [f.expression for f in self.facts]:
                    print(f"Derived:
{new_fact.expression}")
self.facts.add(new_fact)                     added = True

    def ask(self, query):
print(f"\nQuerying {query}:")
self.infer_all()         facts = [f.expression
for f in self.facts]         if query in facts:
            print(f"Yes, {query.split('(')[1].strip(')')} is {query.split('(')[0]}.")
else:
            print(f"No, cannot infer {query}.")

    def display(self):
        print("\nAll facts in Knowledge Base:")         for i, f in
enumerate(sorted([f.expression for f in self.facts])):
            print(f"\t{i+1}. {f}")

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

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

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

OUTPUT (Forward Chaining) :

```
Enter number of FOL expressions: 6
Enter expressions:
Man(Marcus)
Pompeian(Marcus)
Pompeian(x) => Roman(x)
Roman(x) => Loyal(x)
Man(x) => Person(x)
Person(x) => Mortal(x)
Enter query: Mortal(Marcus)

Querying Mortal(Marcus):
Derived: Person(Marcus)
Derived: Roman(Marcus)
Derived: Mortal(Marcus)
Derived: Loyal(Marcus)
Yes, Marcus is Mortal.

All facts in Knowledge Base:
    1. Loyal(Marcus)
    2. Man(Marcus)
    3. Mortal(Marcus)
    4. Person(Marcus)
    5. Pompeian(Marcus)
    6. Roman(Marcus)
```

CODE (CNF) :

```
import re def getAttributes(string):     expr =
r'\([^)]+\)'     matches = re.findall(expr, string)
return [m for m in str(matches) if m.isalpha()]


def getPredicates(string):
    expr = r'[A-Za-z~]+\([A-Za-z,]+\)'
    return re.findall(expr, string)
```

```
def DeMorgan(sentence):
    string =
''.join(list(sentence).copy())     string
= string.replace('~~', '')     flag = '[' in
string     string = string.replace('~[', '')
string = string.strip(']')

    for predicate in getPredicates(string):
        string = string.replace(predicate, f'~{predicate}')

    s = list(string)     for i,
c in enumerate(s):         if
c == 'V':             s[i] = '^'
elif c == '^':             s[i] =
'V'

    string = ''.join(s)     string =
string.replace('~~', '')     return
f'[{string}]' if flag else string

def Skolemization(sentence):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)]
statement = ''.join(list(sentence).copy())
   matches = re.findall('[∀∃].', statement)

    for match in matches[::-1]:
        statement = statement.replace(match, '')

    statements = re.findall(r'\[\[[^]]+\]\]', statement)
for s in statements:
        statement = statement.replace(s, s[1:-1])
```

```
    for predicate in
getPredicates(statement):         attributes =
getAttributes(predicate)          if
''.join(attributes).islower():
            statement = statement.replace(predicate, predicate)
else:
            aL = [a for a in attributes if a.islower()]
            aU = [a for a in attributes if not a.islower()][0] if attributes else ''
if aU:
                statement = statement.replace(aU, f'{SKOLEM_CONSTANTS.pop(0)}({aL[0] if
len(aL) else match[1]})')     return statement

def clean_output(expr):
    # Remove multiple brackets and redundant
negations     expr = expr.replace('~~', '')     while '[[' in
expr or ']]' in expr:
        expr = expr.replace('[[', '[').replace(']]', ']')

    expr = expr.strip('[] ')

    # Remove redundant outer brackets like [(p | q)] -> p | q
if expr.startswith('(') and expr.endswith(')'):
        expr = expr[1:-1]

    # Replace internal redundant
patterns     expr = re.sub(r'\s+', ' ', expr)
return expr

def fol_to_cnf(fol):
    statement = fol.replace("<=>", "_")
while '_' in statement:
```

```
        i = statement.index('_')
        new_statement = '[' + statement[:i] + '=>' + statement[i+1:] + ']^[' + statement[i+1:] + '=>' +
statement[:i] + ']'
        statement = new_statement

    statement = statement.replace("=>", "-")

    expr = r'\[([^]]+)\]'
    statements = re.findall(expr,
statement)     for i, s in
enumerate(statements):         if '[' in s and
']' not in s:             statements[i] += ']'
for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))

    while '-' in statement:
        i = statement.index('-')
        br = statement.index('[') if '[' in statement else 0         new_statement =
'~' + statement[br:i] + 'V' + statement[i+1:]         statement = statement[:br] +
new_statement if br > 0 else new_statement

   while '~∀' in statement:        i =
statement.index('~∀')
        statement = list(statement)
       statement[i], statement[i+1], statement[i+2] = '∃', statement[i+2], '~'
        statement = ''.join(statement)

   while '~∃' in statement:        i =
statement.index('~∃')
        s = list(statement)
       s[i], s[i+1], s[i+2] = '∀', s[i+2], '~'
```

```
        statement = ''.join(s)

   statement = statement.replace('~[∀', '[~∀')     statement =
statement.replace('~[∃', '[~∃')

   expr = r'(~[∀V∃].)'
    statements = re.findall(expr, statement)
for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))

    expr = r'~\[[^]]+\]'
    statements = re.findall(expr, statement)
for s in statements:
        statement = statement.replace(s, DeMorgan(s))

    return statement

def main():
    print("\n" + "="*50)
    print("   FOL to CNF Converter (Simplified Output)")
print("="*50)
   print("Supports: ∀, ∃, ~, &, |, >>, <=>, brackets [] () {}")
    print("NOTE: Use 'V' for OR inside the formula if needed.")
print("-" * 50)
    fol = input("Enter FOL formula: ").strip()
print("-" * 50)
    try:
        raw_cnf  =  fol_to_cnf(fol)                result  =
Skolemization(raw_cnf)                     cleaned   =
clean_output(result)        print(f"Original:   {fol}")
print(f"CNF   Form:       {cleaned}")           except
```

Exception as e: print("\nError: Could not parse the formula.") print("Details:", e) print("="*50 + "\n") if __name__ == "__main__": main()

OUTPUT (CNF);

```
==================================================
   FOL to CNF Converter (Simplified Output)
==================================================
Supports: ∀, ∃, ~, &, |, >>, <=>, brackets [] () {}
NOTE: Use 'V' for OR inside the formula if needed.
--------------------------------------------------
Enter FOL formula: ∀x[~∀y~(Animal(y)VLoves(x,y))]V[∃y Loves(y,x)]
--------------------------------------------------
Original:   ∀x[~∀y~(Animal(y)VLoves(x,y))]V[∃y Loves(y,x)]
CNF Form:   Animal(y)VLoves(x,y))]V[ Loves(y,x
==================================================
```

Program 10: Resolution ALGORITHM

:

Date 13/11/25

Resolution in FOL

→ Algorithm:

1) Convert all sentences to CNF.

2) Negate conclusions & convert the results to CNF.

3) Add negated conclusion to the premise clauses.

4) Repeat until contradiction or no progress is made.

- Example:

- allergies (X) → sneeze (X)
  ⇒ ¬allergies (X) ∨ sneeze (X)

- cat (Y) ∧ allergic to cat (X) → allergies (X)
  ⇒ ¬cat(Y) ∨ ¬allergic to cat(X) ∨ allergies (X)

- cat (Felix)

- allergic to cat (Mary)

- To prove: sneeze (Mary)

→ So we include ¬sneeze (Mary) to the KB & try to prove contradiction.

→ Resolution:

¬allergies (X) ∨ sneeze (X)   ¬cat(Y) ∨ ¬allergic to cat(Z) ∨ allergies(Z)

{X/Z}

sneeze (X) ∨ ¬cat(Y) ∨ ¬allergic to cat(X)   ¬sneeze(Mary)

{Mary/X}

¬cat (Y) ∨ ¬allergic to cat(Mary)   cat(Felix)

{Felix/Y}

¬allergic to cat (Mary)   allergic to cat(Mary)

{ }

13/11

∴ We have a contradiction
⇒ ¬sneeze (Mary) is false

∴ sneeze (Mary) is true
⇒ Hence we prove that Mary sneezes.

→ OUTPUT:

The negation is unsatisfiable. By resolution, Mary sneezes is proven.

CODE :

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

# Define symbols
Food = symbols('Food')
Apple = symbols('Apple')
Vegetables = symbols('Vegetables')
Peanuts = symbols('Peanuts')
John_likes_x = symbols('John_likes_x')
Anil_eats_x = symbols('Anil_eats_x')
Harry_eats_x = symbols('Harry_eats_x')
```

```
Alive_x = symbols('Alive_x')
Killed_x = symbols('Killed_x')

# Knowledge Base in propositional logic
# a. John likes all kind of food -> For each food x, John_likes_x if Food(x)
kb = And(
    Implies(Food, John_likes_x),  # a
    Implies(Or(Apple, Vegetables), Food),  # b
    Implies(And(Anil_eats_x, Not(Killed_x)), Food),  # c
    And(Anil_eats_x, Alive_x),  # d
    Implies(Anil_eats_x, Harry_eats_x),  # e
    Implies(Alive_x, Not(Killed_x)),  # f
    Implies(Not(Killed_x), Alive_x)  # g
)

# We want to prove: John likes peanuts -> John_likes_x for Peanuts
# Assume the negation of the goal for resolution goal_negation
= Not(John_likes_x)

# Combine KB with negated goal
combined = And(kb, goal_negation)

# Check satisfiability
sat_result = satisfiable(combined)

if sat_result:
    print("The negation is satisfiable. Cannot prove John likes peanuts from KB.") else:
    print("The negation is unsatisfiable. By resolution, John likes peanuts is proven!")
```

OUTPUT :

```
The negation is unsatisfiable. By resolution, John likes peanuts is proven!
```