VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Arpith Gowda H S (1BM23CS053)

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

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Arpith Gowda H S (1BM23CS053),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

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

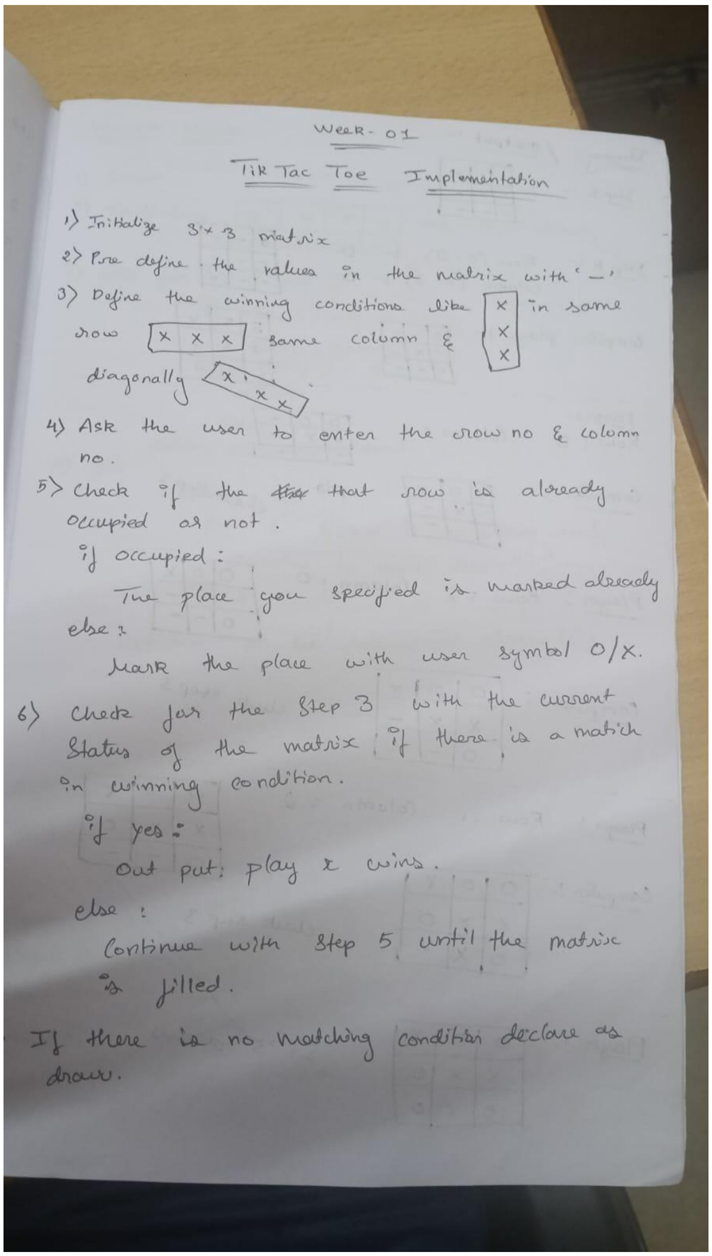
Index

Github Link:

https://github.com/Arpith261/AI-Lab

Program 1
Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:



Week- 01

Tik Tac Toe Implementation

1) Initialize 3×3 matrix
2) Pre define the values in the matrix with '–'
3) Define the winning conditions like x x x in same row x x x same column & x x x diagonally x x x
4) Ask the user to enter the row no & column no.
5) Check if the ~~that~~ that row is already occupied or not.
   if occupied :
   The place you specified is marked already
   else :
   Mark the place with user symbol O/X.
6) Check for the step 3 with the current status of the matrix if there is a match in winning condition.
   if yes :
   Out put: play x wins.
   else :
   Continue with step 5 until the matrix is filled.

If there is no matching condition declare as draw.

Tic Tac Toe code:

```python
import math

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

def check_winner(board, player):
    # Rows, columns, diagonals
    for row in board:
        if all(cell == player for cell in row):
            return True
    for col in range(3):
        if all(board[row][col] == player for row in range(3)):
            return True
    if all(board[i][i] == player for i in range(3)) or \
       all(board[i][2-i] == player for i in range(3)):
       return True
    return False

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

def minimax(board, depth, is_maximizing):
    if check_winner(board, "O"):  # Computer wins
        return 1
    if check_winner(board, "X"):  # Player wins
        return -1
```

```
    if is_full(board):
        return 0
    if is_maximizing:  # Computer's move
        best_score = -math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == " ":
                    board[i][j] = "O"
                    score = minimax(board, depth + 1, False)
                    board[i][j] = " "
                    best_score = max(score, best_score)
        return best_score
    else:  # Player's move
        best_score = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == " ":
                    board[i][j] = "X"
                    score = minimax(board, depth + 1, True)
                    board[i][j] = " "
                    best_score = min(score, best_score)
        return best_score
def best_move(board):
    best_score = -math.inf
```

```
    move = None
    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":
                board[i][j] = "O"
                score = minimax(board, 0, False)
                board[i][j] = " "
                if score > best_score:
                    best_score = score
                    move = (i, j)
    return move
def tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    print("Welcome to Tic-Tac-Toe! You are 'X' and computer is 'O'.")
    print_board(board)
    while True:
        # Player move
        while True:
            try:
                row = int(input("Enter row (0-2): "))
                col = int(input("Enter col (0-2): "))
                if board[row][col] == " ":
                    board[row][col] = "X"
                    break
```

```
            else:
                print("Cell already taken, try again.")
        except (ValueError, IndexError):
            print("Invalid input! Enter numbers 0-2.")
    print_board(board)
    if check_winner(board, "X"):
        print("□ You win!") break
    if is_full(board):
        print("It's a draw!")
        break
    # Computer move
    print("Computer's turn...")
    move = best_move(board)
    if move:
        board[move[0]][move[1]] = "O"
    print_board(board)
    if check_winner(board, "O"):
        print("□ Computer wins!")
        break
    if is_full(board):
        print("It's a draw!")
        break
if __name___== "__main__":
```

```
    tic_tac_toe()
```

Vaccum cleaner code:

```
import random
rooms=[1,1,1,1]
botpos =(int(input("Enter Initial Position: "))-1)
cleanedpos=[]
cost=0
def movebot(pos):
    while True:
        n= random.randint(0,3)
        if n != pos and n not in cleanedpos:
            pos = n
            break
    return pos
while True:
    print(str(rooms))
    print(botpos+1)
    if rooms[botpos]==1:
        rooms[botpos]=0
        cleanedpos.append(botpos)
        cost+=1
        if len(cleanedpos) == 4:
            break
```

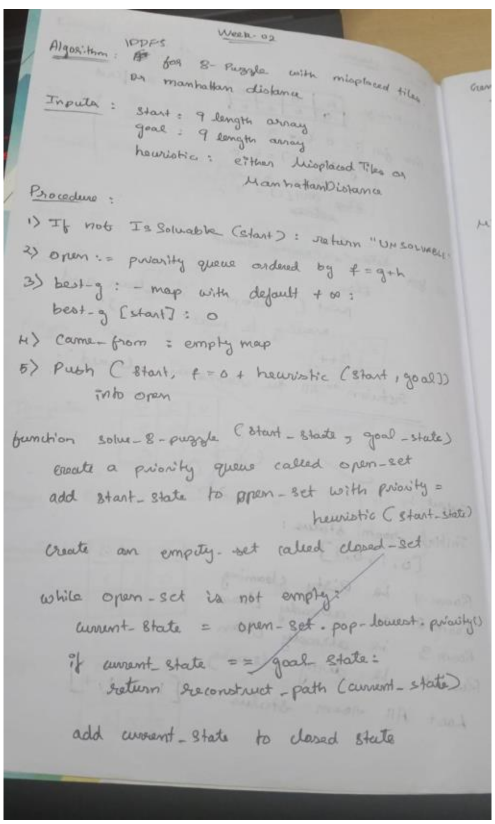
```
        botpos=movebot(botpos)
      elif rooms[botpos]==0:
        cleanedpos.append(botpos)
        if len(cleanedpos) == 4:
          break
        botpos = movebot(botpos)
    print("cost="+str(cost))
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

Implement Iterative deepening search algorithm

Algorithm:


Week-02

Algorithm: IDDFS for 8-Puzzle with misplaced tiles or manhattan distance

Inputs: start: 9 length array
goal: 9 length array
heuristic: either Misplaced Tiles or ManhattanDistance

Procedure:

1> If not IsSolvable (start): return "UNSOLVABLE"
2> open := priority queue ordered by f = g+h
3> best-g : - map with default + ∞ ;
best-g [start] : 0
4> came-from : empty map
5> Push (start, f = 0 + heuristic (start, goal)) into open

function solve-8-puzzle (start-state, goal-state)
  create a priority queue called open-set
  add start-state to open-set with priority = heuristic (start-state)

  Create an empty-set called closed-set

  while open-set is not empty:
    current-state = open-set.pop-lowest-priority()
    if current-state == goal-state:
      return reconstruct-path (current-state)

    add current-state to closed state

DFS code:

```
import time

def find_possible_moves(state):

    index = state.index('_')

    moves = {

        0: [1, 3],

        1: [0, 2, 4],

        2: [1, 5],

        3: [0, 4, 6],

        4: [1, 3, 5, 7],

        5: [2, 4, 8],

        6: [3, 7],

        7: [6, 8, 4],

        8: [5, 7],

    }

    return moves.get(index, [])

def dfs(initial_state, goal_state, max_depth=50):

    stack = [(initial_state, [], 0)]

    visited = {tuple(initial_state)}

    states_explored = 0

    printed_depths = set()

    while stack:

        current_state, path, depth = stack.pop()

        if depth > max_depth:
```

```
            continue

        if depth not in printed_depths:
            print(f"\n--- Depth {depth} ---")
            printed_depths.add(depth)

        states_explored += 1
        print(f"State #{states_explored}: {current_state}")

        if current_state == goal_state:
            print(f"\n Goal reached at depth {depth} after exploring {states_explored} states.\n")
            return path, states_explored

        possible_moves_indices = find_possible_moves(current_state)
        for move_index in reversed(possible_moves_indices):  # Reverse for DFS order
            next_state = list(current_state)
            blank_index = next_state.index('_')
            next_state[blank_index], next_state[move_index] = next_state[move_index],
next_state[blank_index]

            if tuple(next_state) not in visited:
                visited.add(tuple(next_state))
                stack.append((next_state, path + [next_state], depth + 1))

    print(f"\n Goal state not reachable within depth {max_depth}. Explored {states_explored} states.\n")
    return None, states_explored

# ----------- TEST -----------
initial_state = [1, 2, 3,
                 4, 8, '_',
                 7, 6, 5]

goal_state    = [1, 2, 3,
```

```python
        4, 5, 6,
        7, 8, '_']
# Measure execution time
start_time = time.time()
solution_path, explored = dfs(initial_state, goal_state, max_depth=50)
end_time = time.time()
if solution_path is None:
    print("No solution found.")
else:
    print("Solution path:")
    for step, state in enumerate(solution_path, start=1):
        print(f"Step {step}: {state}")
print("\nExecution time: {:.6f} seconds".format(end_time - start_time))
print("Total states explored:", explored)
```

IDDFS code:

```python
import time
# ----------- MOVE GENERATOR -----------
def find_possible_moves(state):
    index = state.index('_')
    if index == 0:
        return [1, 3]
    elif index == 1:
        return [0, 2, 4]
```

```
    elif index == 2:
        return [1, 5]
    elif index == 3:
        return [0, 4, 6]
    elif index == 4:
        return [1, 3, 5, 7]
    elif index == 5:
        return [2, 4, 8]
    elif index == 6:
        return [3, 7]
    elif index == 7:
        return [4, 6, 8]
    elif index == 8:
        return [5, 7]
    return []
# ----------------- DEPTH LIMITED SEARCH -----------------
def depth_limited_dfs(state, goal_state, limit, path, visited):
    if state == goal_state:
        return path
    if limit <= 0:
        return None
    visited.add(tuple(state))
    for move_index in find_possible_moves(state):
        next_state = list(state)
```

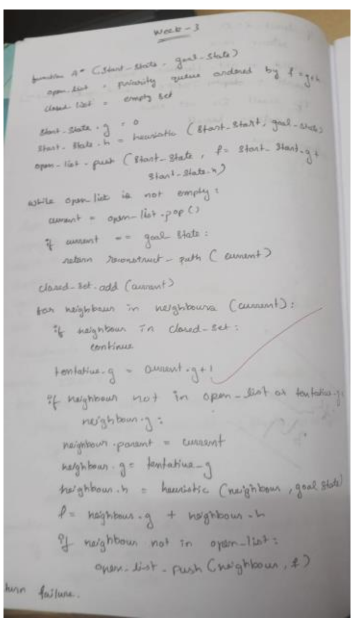
```
        blank_index = next_state.index('_')
        next_state[blank_index], next_state[move_index] = next_state[move_index], next_state[blank_index]
        if tuple(next_state) not in visited:
            result = depth_limited_dfs(next_state, goal_state, limit - 1, path + [next_state], visited)
            if result is not None:
                return result
    return None
# ---------------- ITERATIVE DEEPENING DFS ----------------
def iddfs(initial_state, goal_state, max_depth=30):
    for depth in range(max_depth):
        print(f"Searching at depth limit = {depth}")
        visited = set()
        result = depth_limited_dfs(initial_state, goal_state, depth, [initial_state], visited)
        if result is not None:
            return result, depth
    return None, max_depth
# ----------- TEST -----------
initial_state = [1, 2, 3,
                 4, 8, '_',
                 7, 6, 5]
goal_state    = [1, 2, 3,
                 4, 5, 6,
                 7, 8, '_']
# Measure execution time
```

```
start_time = time.time()

solution_path, depth_reached = iddfs(initial_state, goal_state, max_depth=30)

end_time = time.time()

if solution_path is None:

    print("Goal state is not reachable within given depth limit.")

else:

    print("\n\nSolution path found:")

    for step, state in enumerate(solution_path, start=0):

        print(f"Step {step}: {state}")

print("\nExecution time: {:.6f} seconds".format(end_time - start_time))

print("Depth reached:", depth_reached)
```

Program 3

Implement A* search algorithm

Algorithm:


Week - 3

function A* (start_state, goal_state)
open_list = priority queue ordered by f = g+h
closed_list = empty set

start_state.g = 0
start_state.h = heuristic (start_start, goal_state)
open_list.push (start_state, f= start_start.g + start_state.h)

while open list is not empty:
current = open_list.pop ()
if current == goal_state:
return reconstruct_path (current)

closed_set.add (current)
for neighbour in neighbours (current):
if neighbour in closed_set:
continue

tentative_g = current.g + 1
if neighbour not in open_list or tentative_g < neighbour.g :
neighbour.parent = current
neighbour.g = tentative_g
neighbour.h = heuristic (neighbour, goal state)
f = neighbour.g + neighbour.h
if neighbour not in open_list:
open_list.push (neighbour, f)
return failure.

A* code:

```
import heapq

def state_key(state):

    return ",".join(map(str, state))

def is_solvable(state):

    inversions = 0

    arr = [x for x in state if x != 0]

    for i in range(len(arr)):

        for j in range(i+1, len(arr)):

            if arr[i] > arr[j]:

                inversions += 1

    return inversions % 2 == 0

def manhattan(state):

    total = 0

    for index, val in enumerate(state):

        if val == 0:

            continue

        goal_idx = val - 1

        curr_row, curr_col = divmod(index, 3)

        goal_row, goal_col = divmod(goal_idx, 3)

        total += abs(curr_row - goal_row) + abs(curr_col - goal_col)

    return total

def get_neighbours(state):

    neighbours = []
```

```
    blank_idx = state.index(0)
    row, col = divmod(blank_idx, 3)
    moves = []
    if row > 0: moves.append(blank_idx - 3)
    if row < 2: moves.append(blank_idx + 3)
    if col > 0: moves.append(blank_idx - 1)
    if col < 2: moves.append(blank_idx + 1)
    for m in moves:
        new_state = list(state)
        new_state[blank_idx], new_state[m] = new_state[m], new_state[blank_idx]
        neighbours.append(tuple(new_state))
    return neighbours
def reconstruct_path(came_from, current_key):
    path = []
    while current_key in came_from:
        path.append(tuple(map(int, current_key.split(","))))
        current_key = came_from[current_key]
    path.append(tuple(map(int, current_key.split(","))))
    path.reverse()
    return path
def a_star(start_state, goal_state):
    if not is_solvable(start_state):
        return "UNSOLVABLE"
    start_key = state_key(start_state)
```

```
goal_key = state_key(goal_state)
if start_key == goal_key:
    return [start_state]
open_heap = []
g_score = {start_key: 0}
f_score = {start_key: manhattan(start_state)}
came_from = {}
heapq.heappush(open_heap, (f_score[start_key], manhattan(start_state), start_state))
closed = set()
while open_heap:
    f_current, h_current, current_state = heapq.heappop(open_heap)
    current_key = state_key(current_state)
    if f_current > f_score.get(current_key, float("inf")):
        continue
    if current_key == goal_key:
        return reconstruct_path(came_from, current_key)
    closed.add(current_key)
    for neighbour in get_neighbours(current_state):
        neighbour_key = state_key(neighbour)
        tentative_g = g_score[current_key] + 1
        if neighbour_key in closed and tentative_g >= g_score.get(neighbour_key, float("inf")):
            continue
        if tentative_g < g_score.get(neighbour_key, float("inf")):
            came_from[neighbour_key] = current_key
```

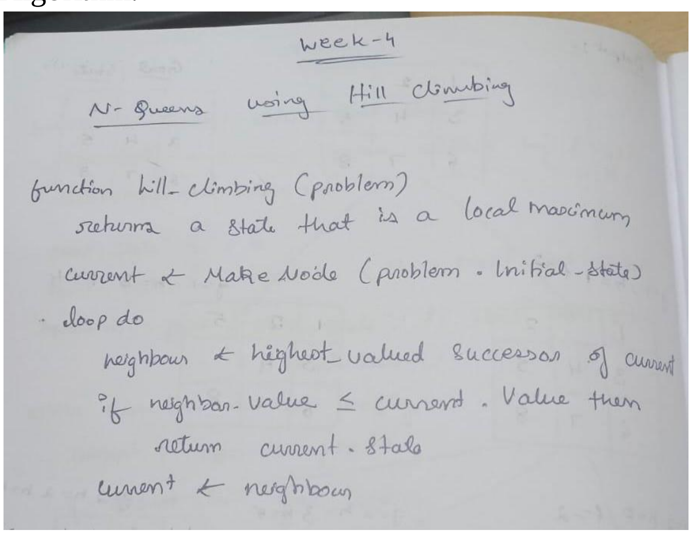
```
            g_score[neighbour_key] = tentative_g
            h = manhattan(neighbour)
            f = tentative_g + h
            f_score[neighbour_key] = f
            heapq.heappush(open_heap, (f, h, neighbour))
    return "FAILURE"
if __name__ == "__main__":
    print("Enter start state of the puzzle (9 numbers, 0 for blank space):")
    user_input = input().strip().split()
    if len(user_input) != 9:
        print("Invalid input! Please enter exactly 9 numbers")
        exit()
    try:
        start = tuple(map(int, user_input))
    except ValueError:
        print("Invalid input! Please enter integers only")
        exit()
    goal = (1, 2, 3,
            4, 5, 6,
            7, 8, 0)
    solution = a_star(start, goal)
    if solution == "UNSOLVABLE":
        print("Puzzle cannot be solved!")
    elif solution == "FAILURE":
```

```
        print("No solution found")
    else:
        print("Solution found in", len(solution) - 1, "moves:")
        for state in solution:
            for i in range(0, 9, 3):
                print(state[i:i+3])
            print("---- ")
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:


week-4

N-Queens using Hill climbing

function hill_climbing (problem)
returns a state that is a local maximum
current ← Make Node (problem . Initial_state)
loop do
neighbour ← highest_valued successor of current
if neighbor. value ≤ current . Value then
return current . state
current ← neighbour

Hill Climbing code:

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

```
            print("Q", end=" ")
        else:
            print(".", end=" ")
    print()
print()
def calculate_cost(state):
    """Returns number of attacking pairs of queens."""
    cost = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            # same row
            if state[i] == state[j]:
                cost += 1
            # same diagonal
            elif abs(state[i] - state[j]) == abs(i - j):
                cost += 1
    return cost
def get_neighbors(state):
    """Generates all neighbors by swapping two queen positions."""
    neighbors = []
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
```

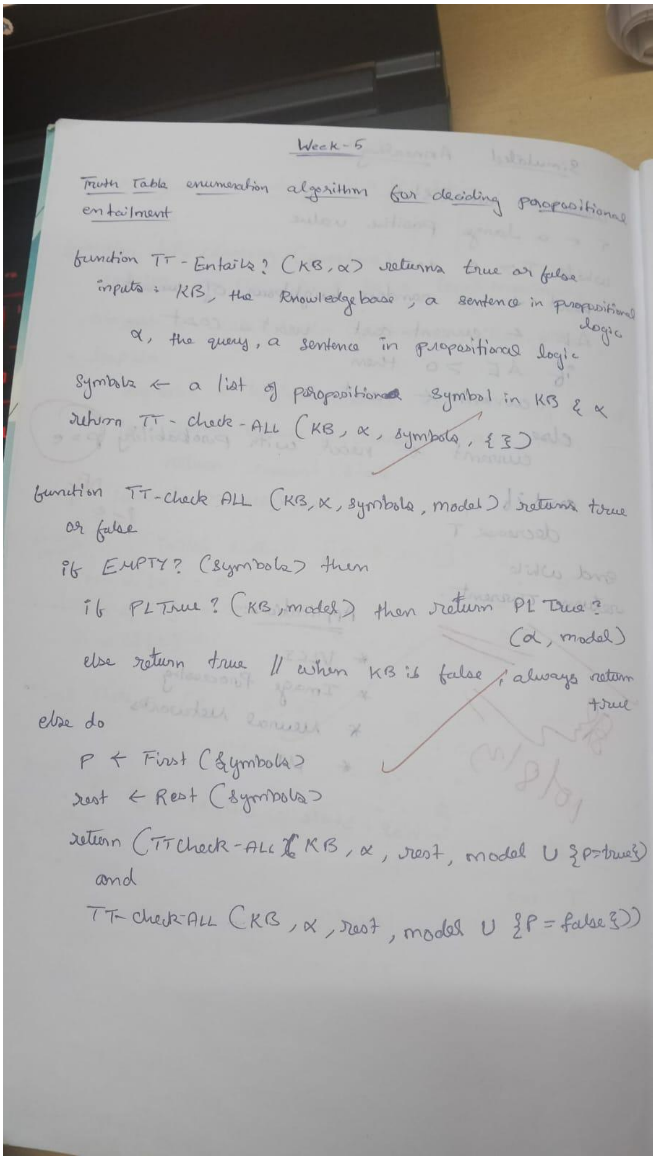
```
        neighbor = state.copy()
        neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
        neighbors.append((neighbor, (i, j)))
    return neighbors
def hill_climbing(state):
    print("\nInitial State:", state)
    print_board(state)
    current_cost = calculate_cost(state)
    step = 1
    while True:
        print(f"Step {step}: Current cost = {current_cost}")
        neighbors = get_neighbors(state)
        neighbor_costs = []
        # Calculate cost for all neighbors for
        neighbor, swapped in neighbors:
            cost          =          calculate_cost(neighbor)
            neighbor_costs.append((cost, neighbor, swapped))
        # Sort by cost and then by smallest column pair as per rules
        neighbor_costs.sort(key=lambda x: (x[0], x[2][0], x[2][1]))
        # Display neighbor costs
        print("Neighbor states and their costs:")
        for cost, neighbor, swapped in neighbor_costs:
            print(f"Swap x{swapped[0]} & x{swapped[1]} => {neighbor}, Cost = {cost}")
        best_cost, best_state, swap = neighbor_costs[0]
```

```
        print("\nBest Neighbor after swap", swap, "is", best_state, "with cost =", best_cost)

        print_board(best_state)

        if best_cost >= current_cost:  # No improvement (local minimum)

            print("No better neighbor found. Hill Climbing terminated.")

            print("Final state:", state)

            print_board(state)

            break

        else:

            state = best_state

            current_cost = best_cost

        if current_cost == 0:

            print("Goal state reached!")

            print_board(state)

            break

        step += 1

# ----------- MAIN -----------

if __name__== "__main__":

    print("Hill Climbing for 4-Queens Problem")

    print("Enter the row positions of 4 queens (each between 0 and 3):")

    state = list(map(int, input("Example (1 2 0 3): ").split()))

    hill_climbing(state)
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



Week-5

Truth Table enumeration algorithm for deciding propositional entailment

function TT-Entails? (KB, α) returns true or false
inputs : KB, the knowledgebase, a sentence in propositional logic
α, the query, a sentence in propositional logic

symbols ← a list of propositional symbol in KB & α
return TT-check-ALL (KB, α, symbols, { })

function TT-check ALL (KB, α, symbols, model) returns true or false
if EMPTY? (symbols) then
if PLTrue ? (KB, model) then return PL True? (α, model)
else return true // when KB is false, always return true
else do
P ← First (symbols)
rest ← Rest (symbols)
return (TTcheck-ALL (KB, α, rest, model ∪ {P=true})
and
TT-check-ALL (KB, α, rest, model ∪ {P = false}))

Simulated annealing code:

```
import random

import math

# ----- Helper functions -----

def random_state(n):

    """Generate a random state: one queen per column."""

    return [random.randint(0, n - 1) for _ in range(n)]

def cost(state):
```

```
    """Compute the number of attacking pairs of queens (lower is better)."""
    n = len(state)
    conflicts = 0
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts
def random_neighbour(state):
    """Generate a neighbour by moving one queen to another row."""
    n = len(state)
    neighbour = state.copy()
    col = random.randint(0, n - 1)      # random column
    new_row = random.randint(0, n - 1)  # new random row
    neighbour[col] = new_row
    return neighbour
# ----- Simulated Annealing -----
def simulated_annealing(n, initial_temp=1000, cooling_rate=0.95, stop_temp=1e-3, max_iterations=10000):
    current = random_state(n)
    current_cost = cost(current)
    T = initial_temp
    iteration = 0
    print("\nInitial state:", current, "Cost:", current_cost)
    while T > stop_temp and iteration < max_iterations:
```

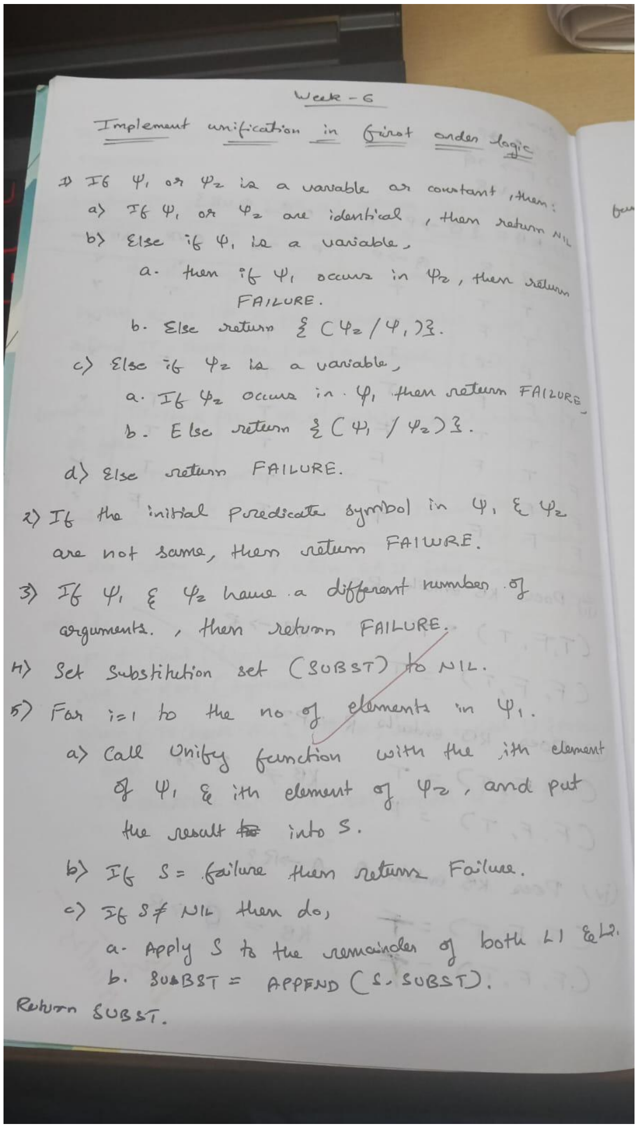
```
next_state = random_neighbour(current)

next_cost = cost(next_state)

deltaE = current_cost - next_cost

# Acceptance condition

if deltaE > 0:

    accepted = True

else:

    p = math.exp(deltaE / T)

    accepted = random.random() < p

# Print current step info

print(f"\nStep {iteration+1}:")

print(f"  Current: {current} (Cost={current_cost})")

print(f"  Next:    {next_state}  (Cost={next_cost})")

print(f"  ΔE = {deltaE:.3f}, T = {T:.3f}")

print(f"  Accepted: {accepted}")

# Accept or reject

if accepted:

    current = next_state

    current_cost = next_cost

# Cooling

T *= cooling_rate

iteration += 1

# Stop if solved

if current_cost == 0:
```

```
            break
    return current, current_cost, iteration
# ----- Main -----
if __name__ == "__main__":
    n = int(input("Enter the number of queens (N): "))
    solution, cost_val, iterations = simulated_annealing(n)
    print("\nFinal state:", solution)
    print("Conflicts:", cost_val)
    print("Iterations:", iterations)
    if cost_val == 0:
        print("\nSolution found:\n")
        for row in range(n):
            print(" ".join("Q" if solution[col] == row else "." for col in range(n)))
    else:
        print("\nNo perfect solution found (try rerunning; SA is stochastic).")
```

Program 6

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

Algorithm:



Week - 6

Implement unification in first order logic

1) If Ψ_1 or Ψ_2 is a variable or constant, then:
   a) If Ψ_1 or Ψ_2 are identical, then return NIL
   b) Else if Ψ_1 is a variable,
      a. then if Ψ_1 occurs in Ψ_2, then return FAILURE.
      b. Else return $\{(\Psi_2/\Psi_1)\}$.
   c) Else if Ψ_2 is a variable,
      a. If Ψ_2 occurs in Ψ_1 then return FAILURE
      b. Else return $\{(\Psi_1/\Psi_2)\}$.
   d) Else return FAILURE.
2) If the initial Predicate symbol in Ψ_1 & Ψ_2 are not same, then return FAILURE.
3) If Ψ_1 & Ψ_2 have a different number of arguments, then return FAILURE.
4) Set Substitution set (SUBST) to NIL.
5) For i=1 to the no of elements in Ψ_1.
   a) Call Unify function with the ith element of Ψ_1 & ith element of Ψ_2, and put the result into S.
   b) If S = failure then returns Failure.
   c) If S ≠ NIL then do,
      a. Apply S to the remainder of both L1 & L2.
      b. SUBST = APPEND (S, SUBST).

Return SUBST.

Propositional logic code:

```
from itertools import product

# Function to safely evaluate logical expressions from user input
def eval_expr(expr, model):
    # Replace logical symbols for Python syntax
    expr = expr.replace('∨', 'or').replace('∧', 'and').replace('¬', 'not ')
    return eval(expr, {}, model)

# Generate all possible truth assignments (models)
```

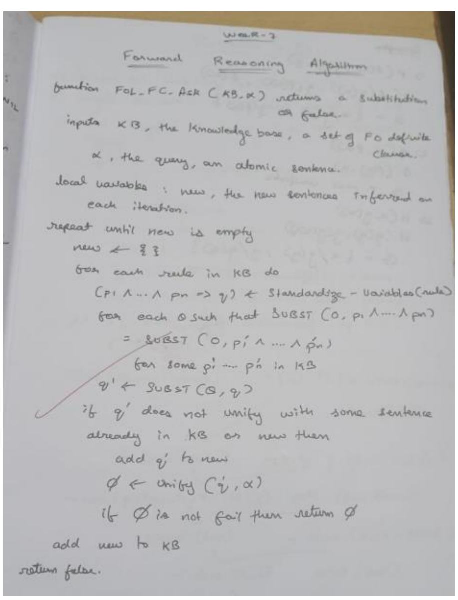
```
def all_models(symbols):
    for values in product([False, True], repeat=len(symbols)):
        yield dict(zip(symbols, values))
# Check entailment: KB ⊨ α
def entails(KB_expr, alpha_expr, symbols):
    for model in all_models(symbols):
        kb_val = eval_expr(KB_expr, model)
        alpha_val = eval_expr(alpha_expr, model)
        if kb_val and not alpha_val:
            print(" Counterexample found:", model)
            return False
    return True
# Display truth table
def truth_table(KB_expr, alpha_expr, symbols):
    headers = "  ".join(f"{s:^6}" for s in symbols)
    print(f"{headers}   {'KB':^8}   {'α':^8}")
    print("-" * (10 * len(symbols) + 20))
    for model in all_models(symbols):
        values = "  ".join(f"{str(model[s]):^6}" for s in symbols)
        kb_val = eval_expr(KB_expr, model)
        alpha_val = eval_expr(alpha_expr, model)
        print(f"{values}   {str(kb_val):^8}   {str(alpha_val):^8}")
# === Main Program ===
print("=== Propositional Entailment using Truth Table Enumeration ===")
```

```
# Input propositional variables

symbols = input("Enter propositional symbols (comma separated, e.g., A,B,C): ").replace(" ", "").split(",")

# Input Knowledge Base (KB) and Query (α)

KB_expr = input("Enter Knowledge Base (use and/or/not or ∧/∨/¬): ")

alpha_expr = input("Enter Query α (use and/or/not or ∧/∨/¬): ")

# Display truth table

print("\n--- Truth Table ---")

truth_table(KB_expr, alpha_expr, symbols)

# Check entailment

result = entails(KB_expr, alpha_expr, symbols)

print("\nResult:")

print(" KB entails α" if result else " KB does NOT entail α")
```

Program 7

Implement unification in first order logic

Algorithm:



Week-7

Forward Reasoning Algorithm

function FOL_FC_Ask (KB, α) returns a substitution or false.

inputs KB, the knowledge base, a set of FO definite clauses

α, the query, an atomic sentence.

local variables : new, the new sentences inferred on each iteration.

repeat until new is empty

new ← { }

for each rule in KB do

($p_1 \wedge \dots \wedge p_n \Rightarrow q$) ← Standardize - Variables(rule)

for each θ such that SUBST (θ, $p_1 \wedge \dots \wedge p_n$)

= SUBST (θ, $p_1' \wedge \dots \wedge p_n'$)

for some $p_1' \dots p_n'$ in KB

q' ← SUBST (θ, q)

if q' does not unify with some sentence already in KB or new then

add q' to new

ϕ ← unify (q', α)

if ϕ is not fail then return ϕ

add new to KB

return false.

Unification code:

```
import re

# Utility: parse the expression into function/operator and arguments

def parse(expr):

    expr = expr.strip()

    if '(' not in expr:

        return expr, []

    func = expr[:expr.index('(')].strip()

    args = expr[expr.index('(')+1:-1]

    args = [a.strip() for a in split_args(args)]

    return func, args

# Split arguments correctly (handles nested brackets)

def split_args(args_str):

    args, level, start = [], 0, 0

    for i, ch in enumerate(args_str):

        if ch == ',' and level == 0:

            args.append(args_str[start:i].strip())

            start = i + 1

        elif ch == '(':

            level += 1

        elif ch == ')':

            level -= 1

    args.append(args_str[start:].strip())

    return args
```

```
# Apply substitution to an expression
def substitute(expr, subs):
    for var, val in subs.items():
        expr = re.sub(rf'\b{var}\b', val, expr)
    return expr
# Check if variable occurs inside term (Occurs check)
def occurs_check(var, term):
    if var == term:
        return True
    if '(' not in term:
        return False
    _, args = parse(term)
    return any(occurs_check(var, arg) for arg in args)
# Unification algorithm
def unify(e1, e2, subs=None):
    if subs is None:
        subs = {}
    e1 = substitute(e1, subs)
    e2 = substitute(e2, subs)
    if e1 == e2:
        return subs
    f1, args1 = parse(e1)
    f2, args2 = parse(e2)
    # Case 1: Both are compound terms
```

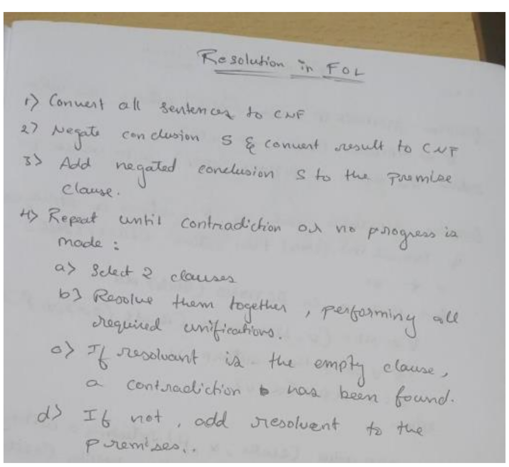
```
if args1 and args2:
    if f1 != f2 or len(args1) != len(args2):
        print(f"❌Function symbols or arity mismatch: {f1} vs {f2}")
        return None
    for a1, a2 in zip(args1, args2):
        subs = unify(a1, a2, subs)
        if subs is None:
            return None
    return subs
# Case 2: Variable binding
elif e1.islower() and e1.isalpha(): # e1 is variable
    if occurs_check(e1, e2):
        print(f"❌Occurs check failed: {e1} occurs in {e2}")
        return None
    subs[e1] = e2
    return subs
elif e2.islower() and e2.isalpha(): # e2 is variable
    if occurs_check(e2, e1):
        print(f"❌Occurs check failed: {e2} occurs in {e1}")
        return None
    subs[e2] = e1
    return subs
# Otherwise mismatch
else:
```

```
        print(f"❌Cannot unify {e1} with {e2}")
        return None

# --- MAIN PROGRAM ---
print("=== Unification Algorithm ===")
expr1 = input("Enter first expression: ").strip()
expr2 = input("Enter second expression: ").strip()
result = unify(expr1, expr2)
if result:
    print("\n✅Unification Successful!")
    print("Substitutions:")
    for k, v in result.items():
        print(f" {k} / {v}")
else:
    print("\n❌Unification Failed.")
```

Program 8

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

Algorithm:



Resolution in FOL

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

Forward reasoning code:

```
import re

def isVariable(x):

    return len(x) == 1 and x.islower() and x.isalpha()

def getAttributes(string):

    expr = r'\([^)]+\)'

    matches = re.findall(expr, string)

    return matches

def getPredicates(string):

    expr = r'([a-z~]+)\([^&|]+\)'

    return re.findall(expr, string)

class Fact:

    def __init__(self, expression):

        self.expression = expression

        predicate, params = self.splitExpression(expression)

        self.predicate = predicate

        self.params = params

        self.result = any(self.getConstants())

    def splitExpression(self, expression):

        predicate = getPredicates(expression)[0]

        params = getAttributes(expression)[0].strip('()').split(',')

        return [predicate, params]

    def getResult(self):

        return self.result
```

```
    def getConstants(self):

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

    def getVariables(self):

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

    def substitute(self, constants):

        constants_copy = constants.copy()

        expr = f"{self.predicate}({','.join([constants_copy.pop(0) if isVariable(p) else p for p in
self.params])})"

        return Fact(expr)

class Implication:

    def  __init__(self,  expression):

        self.expression = expression

        l = expression.split('=>')

        self.lhs = [Fact(f) for f in l[0].split('&')]

        self.rhs = Fact(l[1])

    def evaluate(self, facts):

        constants = {}

        new_lhs = []

        for fact in facts:

            for val in self.lhs:

                if val.predicate == fact.predicate:

                    for i, v in enumerate(val.getVariables()):

                        if v:

                            constants[v] = fact.getConstants()[i]

                    new_lhs.append(fact)
```

```
        predicate = getPredicates(self.rhs.expression)[0]
        attributes = str(getAttributes(self.rhs.expression)[0])
        for key in constants:
            if constants[key]:
                attributes = attributes.replace(key, constants[key])
        expr = f'{predicate}{attributes}'
        return Fact(expr) if len(new_lhs) and all([f.getResult() for f in new_lhs]) else None
class KB:
    def __init__(self):
        self.facts = set()
        self.implications = set()
    def tell(self, e):
        if '=>' in e:
            self.implications.add(Implication(e))
        else:
            self.facts.add(Fact(e))
        for i in self.implications:
            res = i.evaluate(self.facts)
            if res:
                self.facts.add(res)
    def ask(self, e):
        facts = set([f.expression for f in self.facts])
        print(f'\nQuerying {e}:')
        i = 1
```

```
        found = False
        for f in facts:
            if Fact(f).predicate == Fact(e).predicate:
                print(f'\t{i}. {f}')
                i += 1
                found = True
        if not found:
            print("\tNo matching facts found.")
    def display(self):
        print("\nAll facts:")
        for i, f in enumerate(set([f.expression for f in self.facts])):
            print(f'\t{i+1}. {f}')
def main():
    kb = KB()
    print("Enter the number of FOL expressions present in KB:")
    n = int(input())
    print("Enter the expressions:")
    for i in range(n):
        fact = input().strip()
        kb.tell(fact)
    print("Enter the query:")
    query = input().strip()
    kb.ask(query)
    kb.display()
```

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

Program 9

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

Algorithm:

Resolution code:

```
def parse_clause(clause_str):
    return set(clause_str.split('v'))
def get_complement(literal):
    return literal[1:] if literal.startswith('~') else '~' + literal
def resolve(ci, cj):
    resolvents = set()
    for literal in ci:
        complement = get_complement(literal)
        if complement in cj:
            new_clause = (ci - {literal}) | (cj - {complement})
            resolvents.add(frozenset(new_clause))
    return resolvents
def resolution(kb_clauses, query):
    negated_query = get_complement(query)
    kb = [parse_clause(clause) for clause in kb_clauses] + [parse_clause(negated_query)]
    print("\n---------------------------- ")
    print("KnowledgeBase - Resolution")
```

```
    print("----------------------------")
    print(f"\nKnowledge Base Clauses: {kb_clauses}")
    print(f"Query: {query}")
    print(f"Negated Query Added: {negated_query}")
    print("\nResolution Steps:\n")
    new = set()
    while True:
        pairs = [(kb[i], kb[j]) for i in range(len(kb)) for j in range(i + 1, len(kb))]
        for (ci, cj) in pairs:
            resolvents = resolve(ci, cj)
            for resolvent in resolvents:
                print(f"Resolving {set(ci)} and {set(cj)} => {set(resolvent)}")
                if not resolvent:
                    print("\n Knowledge Base entails the query (empty clause derived).")
                    return True
                new.add(resolvent)
        if new.issubset(set(map(frozenset, kb))):
            print("\n Knowledge Base does NOT entail the query (no empty clause derived).")
            return False
        for clause in new:
            if clause not in kb:
                kb.append(clause)
print("KnowledgeBase - Resolution")
print("----------------------------")
```

```
print("Enter clauses for the Knowledge Base.")

print("Use 'v' for OR between literals (e.g., '~qv~pvr'), and separate each clause with a space.\n")

kb_input = input("Enter clauses: ").split()

query_input = input("Enter the query: ")

resolution(kb_input, query_input)
```

Program 10

Implement Alpha-Beta Pruning.

Algorithm:

Adversarial Search

function ALPHABETA SEARCH (state) returns an action
  v ← MAXvalue (state, −∞, +∞)
return the action in ACTIONS (state) with value v

function MAXvalue (state, α, β) returns a utility value
  if Terminal Test (state) then return UTILITY (state)
   v ← −∞
   for each a in ACTIONS (state) do
     v ← MAX (v, MIN-value (Result (s,a), α, β))
     if v ≥ β then return v
     α ← MAX (α, v)
   return v

function MIN value (state, α, β) returns a utility value
  if terminal Test (state) then return Utility (state)
  v ← +∞
  for each a in Actions (state) do
    v ← MIN (v, MAX-VALUE (Result (s, a), α, β))
    if v ≤ α then return v
    β ← MIN (β, v)
  return v

Alpha Beta Pruning code:

```
import math

import random

# Use an external "real" board only for the main game loop; recursive functions use state parameters.

board = [" " for _ in range(9)]  # 3x3 board

def print_board(state):

    print("\n")

    for i in range(3):

        print(" " + " | ".join(state[i*3:(i+1)*3]))

        if i < 2:

            print("---+---+---")

    print("\n")

def is_winner(state, player):

    win_combinations = [

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

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

        [0, 4, 8], [2, 4, 6]

    ]

    return any(all(state[i] == player for i in combo) for combo in win_combinations)

def is_full(state):

    return " " not in state

def actions(state):

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

def result(state, action, player):
```

```
    new_state = state.copy()
    new_state[action] = player
    return new_state
def utility(state):
    if is_winner(state, "O"):
        return +1
    elif is_winner(state, "X"):
        return -1
    else:
        return 0
def terminal_test(state):
    return is_winner(state, "X") or is_winner(state, "O") or is_full(state)
# --- Alpha-Beta Functions ---
def max_value(state, alpha, beta):
    if terminal_test(state):
        return utility(state)
    v = -math.inf
    for a in actions(state):
        v = max(v, min_value(result(state, a, "O"), alpha, beta))
        if v >= beta:
            return v
        alpha = max(alpha, v)
    return v
def min_value(state, alpha, beta):
```

```
    if terminal_test(state):
        return utility(state)
    v = math.inf
    for a in actions(state):
        v = min(v, max_value(result(state, a, "X"), alpha, beta))
        if v <= alpha:
            return v
        beta = min(beta, v)
    return v
def alpha_beta_search(state):
    best_score = -math.inf
    best_action = None
    if not actions(state):
        return None
    for a in actions(state):
        value = min_value(result(state, a, "O"), -math.inf, math.inf)
        if value > best_score:
            best_score = value
            best_action = a
    # Fallback: if something goes wrong, return a random legal move
    if best_action is None:
        legal = actions(state)
        return random.choice(legal) if legal else None
    return best_action
```

```
# --- Game Loop ---
def human_move():
    while True:
        try:
            move = int(input("Enter your move (1-9): ")) - 1
        except ValueError:
            print("Please enter a number 1-9.")
            continue
        if move < 0 or move > 8:
            print("Move out of range. Choose 1-9.")
            continue
        if board[move] != " ":
            print("Cell already taken. Try another.")
            continue
        return move
def choose_first():
    while True:
        ans = input("Who goes first? (me/ai) [me]: ").strip().lower()
        if ans == "" or ans.startswith("m"):
            return "me"
        if ans.startswith("a"):
            return "ai"
        print("Type 'me' or 'ai' (or press Enter for me).")
def main():
```

```
global board
board = [" " for _ in range(9)]
print("Welcome to Tic-Tac-Toe! You are X, AI is O.")
first = choose_first()
print_board(board)
while True:
    if first == "me":
        # Human turn
        move = human_move()
        board[move] = "X"
        print_board(board)
        if is_winner(board, "X"):
            print("You win!")
            break
        if is_full(board):
            print("It's a draw!")
            break
        # AI turn
        print("AI is thinking...")
        ai_move = alpha_beta_search(board)
        if ai_move is None:
            print("AI could not find a move — it's a draw.")
            break
        board[ai_move] = "O"
```

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

```
        print_board(board)
        if is_winner(board, "X"):
            print("You win!")
            break
        if is_full(board):
            print("It's a draw!")
            break
if __name___== "__main__":
    main()
```