1 Write the python program to solve 8-Puzzle problem

import heapq

# Goal state

goal\_state = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 0]

]

# Moves: Up, Down, Left, Right

moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def manhattan\_distance(state):

distance = 0

for i in range(3):

for j in range(3):

val = state[i][j]

if val != 0:

goal\_x = (val - 1) // 3

goal\_y = (val - 1) % 3

distance += abs(i - goal\_x) + abs(j - goal\_y)

return distance

def get\_blank\_position(state):

for i in range(3):

for j in range(3):

if state[i][j] == 0:

return i, j

def generate\_neighbors(state):

neighbors = []

x, y = get\_blank\_position(state)

for dx, dy in moves:

nx, ny = x + dx, y + dy

if 0 <= nx < 3 and 0 <= ny < 3:

new\_state = [row[:] for row in state]

new\_state[x][y], new\_state[nx][ny] = new\_state[nx][ny], new\_state[x][y]

neighbors.append(new\_state)

return neighbors

def is\_goal(state):

return state == goal\_state

def a\_star(start\_state):

visited = set()

heap = []

heapq.heappush(heap, (manhattan\_distance(start\_state), 0, start\_state, [])) # (f, g, state, path)

while heap:

f, g, current, path = heapq.heappop(heap)

state\_tuple = tuple(tuple(row) for row in current)

if state\_tuple in visited:

continue

visited.add(state\_tuple)

if is\_goal(current):

return path + [current]

for neighbor in generate\_neighbors(current):

heapq.heappush(heap, (

g + 1 + manhattan\_distance(neighbor), # f(n)

g + 1, # g(n)

neighbor,

path + [current]

))

return None

def print\_path(path):

for step, state in enumerate(path):

print(f"Step {step}:")

for row in state:

print(row)

print()

# Main

if \_\_name\_\_ == "\_\_main\_\_":

# Define initial state

start\_state = [

[1, 2, 3],

[4, 0, 6],

[7, 5, 8]

]

print("Initial State:")

for row in start\_state:

print(row)

print("\nSolving...\n")

solution = a\_star(start\_state)

if solution:

print(f"Solution found in {len(solution) - 1} moves:")

print\_path(solution)

else:

print("No solution found.")

2 Write the python program to solve 8-Queen problem

def solve(n=8, y=0, board=[]):

if y == n: return [board]

solutions = []

for x in range(n):

if all(x != c and abs(x - c) != y - r for r, c in enumerate(board)):

solutions += solve(n, y + 1, board + [x])

return solutions

def print\_solutions():

for sol in solve():

for row in sol:

print(" ".join(["\_" if i != row else "Q" for i in range(8)]))

print("\n")

print\_solutions()

3 Write the python program for Water Jug Problem

def solve(jug1, jug2, target):

q, visited, path = [(0, 0)], set(), []

while q:

a, b = q.pop(0)

if (a, b) in visited:

continue

visited.add((a, b))

path.append((a, b))

if a == target or b == target:

return path

q += [

(jug1, b), (a, jug2), (0, b), (a, 0),

(a - min(a, jug2 - b), b + min(a, jug2 - b)),

(a + min(b, jug1 - a), b - min(b, jug1 - a))

]

return "No Solution"

def print\_solution(jug1, jug2, target):

solution = solve(jug1, jug2, target)

if solution == "No Solution":

print(solution)

else:

for step in solution:

print(step)

print\_solution(4, 3, 2)

4 Write the python program for Cript-Arithmetic problem

def solve(words, result, letters, index, mapping, used):

if index == len(letters):

return mapping if sum(int("".join(str(mapping[c]) for c in w)) for w in words) == int("".join(str(mapping[c]) for c in result)) else None

for d in range(10):

if not used[d]:

mapping[letters[index]], used[d] = d, True

res = solve(words, result, letters, index + 1, mapping, used)

if res: return res

used[d] = False

return None

def crypt\_arithmetic(equation):

words, result = equation.replace(" ", "").split("=")

words = words.split("+")

letters = list(set("".join(words) + result))

return solve(words, result, letters, 0, {}, [False] \* 10) or "No solution"

print(crypt\_arithmetic("SEND + MORE = MONEY"))

5 Write the python program for Missionaries Cannibal problem

def valid(state):

m1, c1, boat, m2, c2 = state

return (0 <= m1 <= 3 and 0 <= c1 <= 3 and 0 <= m2 <= 3 and 0 <= c2 <= 3 and

(m1 == 0 or m1 >= c1) and (m2 == 0 or m2 >= c2))

def successors(state):

m1, c1, boat, m2, c2 = state

moves = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]

next\_states = []

for m, c in moves:

if boat:

next\_state = (m1 - m, c1 - c, 0, m2 + m, c2 + c)

else:

next\_state = (m1 + m, c1 + c, 1, m2 - m, c2 - c)

if valid(next\_state):

next\_states.append(next\_state)

return next\_states

def solve():

start, goal = (3, 3, 1, 0, 0), (0, 0, 0, 3, 3)

queue, visited = [(start, [])], set()

while queue:

state, path = queue.pop(0)

if state == goal:

return path + [state]

if state not in visited:

visited.add(state)

for next\_state in successors(state):

queue.append((next\_state, path + [state]))

return "No solution"

solution = solve()

if solution == "No solution":

print(solution)

else:

print("Steps to solve the Missionaries and Cannibals problem:")

for i, step in enumerate(solution):

m1, c1, boat, m2, c2 = step

print(f"Step {i + 1}: Left -> M:{m1} C:{c1} | Boat: {'Left' if boat else 'Right'} | Right -> M:{m2} C:{c2}")

6 Write the python program for Vacuum Cleaner problem

def vc(env, p):

for r in env:

print(f"VC at {p} - {p} is {'D' if env[p] == 1 else 'C'}")

if env[p]:

print(f"Cleans {p}...")

env[p] = 0

p = "A" if p == "B" else "B"

print("All Clean!")

env = {"A": 1, "B": 0}

vc(env, "A")

7 Write the python program to implement BFS.

def bfs(graph, start):

visited, queue = set(), [start]

while queue:

node = queue.pop(0)

if node not in visited:

print(node, end=" ")

visited.add(node)

queue.extend(sorted(graph[node] - visited))

graph = {

'A': {'B', 'C'},

'B': {'A', 'D', 'E'},

'C': {'A', 'F', 'G'},

'D': {'B'},

'E': {'B', 'H'},

'F': {'C'},

'G': {'C'},

'H': {'E'}

}

bfs(graph, 'A')

8 Write the python program to implement DFS.

def dfs(graph, node, visited):

if node not in visited:

print(node, end=" ") # Print the visited node

visited.add(node) # Mark node as visited

for neighbor in graph.get(node, []): # Visit all neighbors

dfs(graph, neighbor, visited)

# Example graph represented as an adjacency list

graph = {

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

'B': ['D', 'E'],

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

# Run DFS

visited = set()

print("DFS Traversal:")

dfs(graph, 'A', visited)

9 Write the python to implement Travelling Salesman Problem

def td(g, p):

d = sum(g[p[i]][p[i+1]] for i in range(len(p)-1))

return d + g[p[-1]][p[0]] # Return to start

# Generate all paths (backtracking)

def gt(c, s, p, v, bp, bd, g):

if len(p) == len(c):

d = td(g, p)

if d < bd[0]: bd[0], bp[:] = d, p[:]

return

for city in c:

if city not in v:

v.add(city); p.append(city)

gt(c, s, p, v, bp, bd, g)

p.pop(); v.remove(city)

# TSP function

def tsp(g):

c, s = list(g.keys()), list(g.keys())[0]

bp, bd = [], [float('inf')]

gt(c, s, [s], {s}, bp, bd, g)

return bp, bd[0]

# Graph (Adjacency matrix)

g = {

'A': {'A': 0, 'B': 10, 'C': 15, 'D': 20},

'B': {'A': 10, 'B': 0, 'C': 35, 'D': 25},

'C': {'A': 15, 'B': 35, 'C': 0, 'D': 30},

'D': {'A': 20, 'B': 25, 'C': 30, 'D': 0}

}

# Run TSP

bp, bd = tsp(g)

print("Best Path:", bp)

print("Min Distance:", bd)

10 Write the python program to implement A\* algorithm

class N:

def \_\_init\_\_(s, n, g=0, h=0):

s.n, s.g, s.h, s.f, s.p = n, g, h, g + h, None

def a\_star(g, h, s, e):

o, c = {s: N(s, 0, h[s])}, {}

while o:

cur = min(o.values(), key=lambda x: x.f)

del o[cur.n]; c[cur.n] = cur

if cur.n == e:

p = []

while cur: p.append(cur.n); cur = cur.p

return p[::-1]

for nb, cost in g[cur.n].items():

if nb in c: continue

g\_new = cur.g + cost

if nb not in o or g\_new < o[nb].g:

o[nb] = N(nb, g\_new, h[nb])

o[nb].p = cur

return None

g = {'A': {'B': 4, 'C': 3}, 'B': {'D': 5, 'E': 12}, 'C': {'E': 10}, 'D': {'F': 8}, 'E': {'F': 6}, 'F': {}}

h = {'A': 14, 'B': 12, 'C': 11, 'D': 6, 'E': 4, 'F': 0}

print("Shortest Path:", a\_star(g, h, 'A', 'F'))

11 Write the python program for Map Coloring to implement CSP.

neighbors = {

"A": ["B", "C"],

"B": ["A", "C", "D"],

"C": ["A", "B", "D", "E"],

"D": ["B", "C", "E"],

"E": ["C", "D"]

}

# Available colors

colors = ["Red", "Green", "Blue"]

# Dictionary to store the assigned colors

color\_assignment = {}

def is\_valid(region, color):

"""Check if the color assignment is valid for the given region."""

for neighbor in neighbors.get(region, []):

if neighbor in color\_assignment and color\_assignment[neighbor] == color:

return False

return True

def solve(region\_list):

"""Backtracking function to assign colors."""

if not region\_list: # If all regions are assigned colors, return True

return True

region = region\_list[0]

for color in colors:

if is\_valid(region, color):

color\_assignment[region] = color # Assign color

if solve(region\_list[1:]): # Recur for the remaining regions

return True

del color\_assignment[region] # Backtrack if assignment fails

return False # No valid assignment found

# Start solving

if solve(list(neighbors.keys())):

print("Color Assignment:", color\_assignment)

else:

print("No solution found.")

12 Write the python program for Tic Tac Toe game

# Tic Tac Toe Game in Python (2 Player)

def print\_board(board):

for row in board:

print(" | ".join(row))

print("-" \* 5)

def check\_winner(board, player):

# Check rows

for row in board:

if all(cell == player for cell in row):

return True

# Check columns

for col in range(3):

if all(board[row][col] == player for row in range(3)):

return True

# Check diagonals

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\_draw(board):

return all(cell in ['X', 'O'] for row in board for cell in row)

def main():

board = [["1", "2", "3"],

["4", "5", "6"],

["7", "8", "9"]]

current\_player = "X"

while True:

print\_board(board)

move = input(f"Player {current\_player}, choose a position (1-9): ")

if not move.isdigit() or not (1 <= int(move) <= 9):

print("Invalid input! Choose a number from 1 to 9.")

continue

move = int(move) - 1

row, col = divmod(move, 3)

if board[row][col] in ['X', 'O']:

print("That position is already taken. Try again.")

continue

board[row][col] = current\_player

if check\_winner(board, current\_player):

print\_board(board)

print(f"🎉 Player {current\_player} wins!")

break

if is\_draw(board):

print\_board(board)

print("It's a draw!")

break

current\_player = "O" if current\_player == "X" else "X"

if \_\_name\_\_ == "\_\_main\_\_":

main()

13 Write the python program to implement Minimax algorithm for gaming

def eval\_bd(bd):

for r in range(3):

if bd[r][0] == bd[r][1] == bd[r][2] != '\_': return 10 if bd[r][0] == 'X' else -10

for c in range(3):

if bd[0][c] == bd[1][c] == bd[2][c] != '\_': return 10 if bd[0][c] == 'X' else -10

if bd[0][0] == bd[1][1] == bd[2][2] != '\_' or bd[0][2] == bd[1][1] == bd[2][0] != '\_':

return 10 if bd[1][1] == 'X' else -10

return 0

def minimax(bd, is\_max):

score = eval\_bd(bd)

if score: return score

if not any('\_' in row for row in bd): return 0

best = -1000 if is\_max else 1000

for i in range(3):

for j in range(3):

if bd[i][j] == '\_':

bd[i][j] = 'X' if is\_max else 'O'

best = max(best, minimax(bd, not is\_max)) if is\_max else min(best, minimax(bd, not is\_max))

bd[i][j] = '\_'

return best

def best\_move(bd):

move, best\_val = (-1, -1), -1000

for i in range(3):

for j in range(3):

if bd[i][j] == '\_':

bd[i][j] = 'X'

val = minimax(bd, False)

bd[i][j] = '\_'

if val > best\_val: move, best\_val = (i, j), val

return move

grid = [['X', 'O', 'X'], ['O', 'O', 'X'], ['\_', '\_', '\_']]

print("Best Move:", best\_move(grid))

14 Write the python program to implement Apha & Beta pruning algorithm for gaming

def alpha\_beta(node, d, a, b, max\_p):

if d == 0 or isinstance(node, int): return node

val = -999999 if max\_p else 999999

for c in node:

v = alpha\_beta(c, d - 1, a, b, not max\_p)

val = max(val, v) if max\_p else min(val, v)

a, b = (max(a, val), b) if max\_p else (a, min(b, val))

if b <= a: break

return val

tree = [[3, 5, 6], [2, 9, -1], [4, 7, 8]]

print("Best outcome:", alpha\_beta(tree, 3, -999999, 999999, True))

15 Write the python program to implement Decision Tree

class Node:

def \_\_init\_\_(self, question=None, left=None, right=None, label=None):

self.question = question

self.left = left

self.right = right

self.label = label

def build\_tree():

return Node("Is it raining?",

Node("Do you have an umbrella?",

Node(label="Go outside"),

Node(label="Stay inside")),

Node(label="Go outside"))

def classify(node):

while node.label is None:

ans = input(node.question + " (yes/no): ").strip().lower()

node = node.left if ans == "yes" else node.right

return node.label

tree = build\_tree()

print("Decision:", classify(tree))

16 Write the python program to implement Feed forward neural Network

import random, math

def sigmoid(x): return 1 / (1 + math.exp(-x))

def d\_sigmoid(x): return x \* (1 - x)

def init(n, h, o):

return [[random.uniform(-1, 1) for \_ in range(h)] for \_ in range(n)], [[random.uniform(-1, 1) for \_ in range(o)] for \_ in range(h)], [random.uniform(-1, 1) for \_ in range(h)], [random.uniform(-1, 1) for \_ in range(o)]

def forward(inp, w1, w2, b1, b2):

h = [sigmoid(sum(i \* w + b for i, w, b in zip(inp, ws, b1))) for ws in zip(\*w1)]

o = [sigmoid(sum(h[i] \* w + b for i, w, b in zip(range(len(h)), ws, b2))) for ws in zip(\*w2)]

return h, o

def train(data, tar, e=1000, lr=0.5):

w1, w2, b1, b2 = init(len(data[0]), 2, len(tar[0]))

for \_ in range(e):

for i, inp in enumerate(data):

h, o = forward(inp, w1, w2, b1, b2)

d\_o = [(tar[i][j] - o[j]) \* d\_sigmoid(o[j]) for j in range(len(o))]

d\_h = [sum(w2[k][j] \* d\_o[j] for j in range(len(o))) \* d\_sigmoid(h[k]) for k in range(len(h))]

for j in range(len(o)): b2[j] += lr \* d\_o[j]

for k in range(len(h)): b1[k] += lr \* d\_h[k]

for k in range(len(h)): w2[k] = [w + lr \* h[k] \* d\_o[j] for j, w in enumerate(w2[k])]

for j in range(len(inp)): w1[j] = [w + lr \* inp[j] \* d\_h[k] for k, w in enumerate(w1[j])]

return w1, w2, b1, b2

X, Y = [[0,0], [0,1], [1,0], [1,1]], [[0], [1], [1], [0]]

w1, w2, b1, b2 = train(X, Y)

for x in X: print(f"Input: {x}, Output: {forward(x, w1, w2, b1, b2)[1]}")

17 Write a Prolog Program to Sum the Integers from 1 to n.

sum of integers sum(0, 0).

sum(N, S) :-

N > 0,

N1 is N - 1,

sum(N1, S1),

S is N + S1.

#sum(5, Result).

18 Write a Prolog Program for A DB WITH NAME, DOB.

dob(john, '1995-06-15').

dob(alice, '2000-12-01').

dob(bob, '1988-03-23').

dob(eve, '1992-07-19').

find\_dob(Name, DOB) :- dob(Name, DOB).

#find\_dob(john, DOB).

19 Write a Prolog Program for STUDENT-TEACHER-SUB-CODE.

teaches(mr\_smith, math, 101).

teaches(ms\_jones, physics, 102).

student(john, math, 101).

student(alice, physics, 102).

find\_teacher(Student, Teacher) :-

student(Student, Subject, Code),

teaches(Teacher, Subject, Code).

#find\_teacher(john, Teacher).

20 Write a Prolog Program for PLANETS DB.

planet(mercury, terrestrial, 57).

planet(venus, terrestrial, 108).

planet(uranus, ice\_giant, 2871).

planet(neptune, ice\_giant, 4495).

find\_planet\_info(Name, Type, Distance) :- planet(Name, Type, Distance).

#find\_planet\_info(mercury, Type, Distance).

21 Write a Prolog Program to implement Towers of Hanoi.

hanoi(1, Source, Target, \_) :-

write('Move disk 1 from '), write(Source), write(' to '), write(Target), nl.

hanoi(N, Source, Target, Auxiliary) :-

N > 1,

N1 is N - 1,

hanoi(N1, Source, Auxiliary, Target),

write('Move disk '), write(N), write(' from '), write(Source), write(' to '), write(Target), nl,

hanoi(N1, Auxiliary, Target, Source).

#hanoi(3, 'A', 'C', 'B').

22 Write a Prolog Program to print particular bird can fly or not. Incorporate required queries.

can\_fly(sparrow).

can\_fly(eagle).

cannot\_fly(penguin).

cannot\_fly(kiwi).

bird\_flight(Bird, 'can fly') :- can\_fly(Bird).

bird\_flight(Bird, 'cannot fly') :- cannot\_fly(Bird).

#bird\_flight(sparrow, Result).

23 Write the prolog program to implement family tree.

parent(john, mary).

parent(john, mike).

parent(susan, mary).

parent(susan, mike).

parent(mary, alice).

parent(mary, bob).

parent(mike, charlie).

father(X, Y) :- parent(X, Y), male(X).

mother(X, Y) :- parent(X, Y), female(X).

sibling(X, Y) :- parent(P, X), parent(P, Y), X \= Y.

grandparent(X, Y) :- parent(X, Z), parent(Z, Y).

grandchild(X, Y) :- grandparent(Y, X).

ancestor(X, Y) :- parent(X, Y).

ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

male(john).

male(mike).

male(bob).

male(charlie).

female(susan).

female(mary).

female(alice).

#sibling(mary, Sibling).

#grandparent(Grandfather, alice), male(Grandfather).

24 Write a Prolog Program to suggest Dieting System based on Disease.

diet(diabetes, 'Avoid sugar and high carbs. Eat fiber-rich food.').  
diet(hypertension, 'Avoid salt. Eat potassium-rich food.').  
diet(obesity, 'Eat low-calorie food and exercise regularly.').

suggest\_diet(Disease, Advice) :-  
diet(Disease, Advice).

?- suggest\_diet(diabetes, Advice).

25 Write a Prolog program to implement Monkey Banana Problem

initial\_state(state(at\_door, on\_floor, at\_window, has\_not\_eaten)).

final\_state(state(\_, \_, \_, has\_eaten)).

% Define the possible actions and their effects

% Note: Action effects are expressed as changes to the state of the monkey

action(state(at\_door, on\_floor, at\_window, has\_not\_eaten), climb, state(at\_window,

on\_window, at\_window, has\_not\_eaten)).

action(state(at\_window, on\_floor, at\_window, has\_not\_eaten), grasp, state(at\_window,

on\_floor, at\_window, has\_eaten)).

action(state(at\_window, on\_window, at\_window, has\_not\_eaten), climb, state(at\_door,

on\_window, at\_door, has\_not\_eaten)).

action(state(at\_door, on\_window, at\_door, has\_not\_eaten), walk, state(at\_middle,

on\_floor, at\_middle, has\_not\_eaten)).

action(state(at\_middle, on\_floor, at\_middle, has\_not\_eaten), grasp, state(at\_middle,

on\_floor, at\_middle, has\_eaten)).

action(state(at\_middle, on\_floor, at\_middle, has\_eaten), walk, state(at\_door, on\_floor,

at\_door, has\_eaten)).

% Define a predicate to execute a sequence of actions

% Note: The last argument of the predicate is the final state of the sequence of actions

execute\_actions(\_, [], FinalState) :- final\_state(FinalState).

execute\_actions(CurrentState, [Action|Rest], FinalState) :-

action(CurrentState, Action, NextState),

execute\_actions(NextState, Rest, FinalState).

% Define a predicate to solve the problem

% Note: The solution is expressed as a sequence of actions

solve\_problem(ActionList) :-

initial\_state(InitialState),

execute\_actions(InitialState, ActionList, FinalState),

final\_state(FinalState).

ActionList = [walk, grasp].

ActionList = [climb, climb, walk, grasp].

26 Write a Prolog Program for fruit and its color using Back Tracking.

% Facts: Fruit and its corresponding color

fruit\_color(apple, red).

fruit\_color(banana, yellow).

fruit\_color(grape, purple).

fruit\_color(orange, orange).

fruit\_color(lemon, yellow).

fruit\_color(blueberry, blue).

fruit\_color(strawberry, red).

fruit\_color(kiwi, green).

% Query Examples:

% Find all fruits with a specific color

% ?- fruit\_color(Fruit, red).

% Find the color of a specific fruit

% ?- fruit\_color(apple, Color).

27 Write a Prolog Program to implement Best First Search algorithm

% Example Graph

edge(a, b, 1).

edge(a, c, 3).

edge(b, d, 1).

edge(c, d, 1).

edge(d, e, 6).

edge(b, e, 4).

edge(c, f, 5).

% Heuristic Values (for Best-First Search)

heuristic(a, 10).

heuristic(b, 8).

heuristic(c, 6).

heuristic(d, 4).

heuristic(e, 0).

heuristic(f, 7).

% Best-First Search Main Entry Point

best\_first(Start, Goal, Path) :-

heuristic(Start, H),

best\_first\_search([[H, Start]], Goal, [], RevPath),

reverse(RevPath, Path).

% Base Case: Goal node found

best\_first\_search([[\_, Goal]|\_], Goal, Visited, [Goal|Visited]).

% Recursive Case

best\_first\_search([[\_, CurrentNode]|RestQueue], Goal, Visited, Path) :-

findall(

[H, NextNode],

(

edge(CurrentNode, NextNode, \_),

\+ member(NextNode, Visited),

heuristic(NextNode, H)

),

Children

),

append(RestQueue, Children, NewQueue),

sort(NewQueue, SortedQueue), % sort by heuristic value

best\_first\_search(SortedQueue, Goal, [CurrentNode|Visited], Path).

?- best\_first(a, e, Path).

Path = [a, b, e].

28 Write the prolog program for Medical Diagnosis

% Symptoms for diseases

disease(flu) :-

has(fever),

has(cough),

has(sore\_throat),

has(body\_ache).

disease(common\_cold) :-

has(sneezing),

has(runny\_nose),

has(cough).

disease(covid19) :-

has(fever),

has(cough),

has(loss\_of\_taste),

has(shortness\_of\_breath).

disease(malaria) :-

has(fever),

has(chills),

has(sweating),

has(headache).

disease(typhoid) :-

has(fever),

has(abdominal\_pain),

has(weakness),

has(loss\_of\_appetite).

% Dynamic declaration so facts can be added at runtime

:- dynamic(has/1).

ask(Symptom) :-

format('Do you have ~w? (yes/no): ', [Symptom]),

read(Response),

( Response == yes ->

assertz(has(Symptom))

; true).

diagnose :-

retractall(has(\_)), % Clear previous session

symptoms, % Ask user symptoms

( disease(D) ->

format('~nYou may have: ~w~n', [D])

; format('~nSorry, diagnosis is inconclusive.~n')

).

symptoms :-

ask(fever),

ask(cough),

ask(sore\_throat),

ask(body\_ache),

ask(sneezing),

ask(runny\_nose),

ask(loss\_of\_taste),

ask(shortness\_of\_breath),

ask(chills),

ask(sweating),

ask(headache),

ask(abdominal\_pain),

ask(weakness),

ask(loss\_of\_appetite).

?- diagnose.

29 Write a Prolog Program for forward Chaining. Incorporate required queries.

:- dynamic(fact/1).

% Initial Facts

fact(raining).

fact(cloudy).

fact(have\_umbrella).

% Rules

infer(wet\_ground) :-

fact(raining),

\+ fact(wet\_ground),

assertz(fact(wet\_ground)).

infer(take\_umbrella) :-

fact(cloudy),

fact(have\_umbrella),

\+ fact(take\_umbrella),

assertz(fact(take\_umbrella)).

infer(stay\_dry) :-

fact(take\_umbrella),

\+ fact(stay\_dry),

assertz(fact(stay\_dry)).

% Forward chaining engine

forward :-

infer(\_),

!,

forward.

forward.

% Show all derived facts

show\_facts :-

fact(X),

write('Fact: '), write(X), nl,

fail.

show\_facts.

% Sample query to run forward chaining and display results

run :-

forward,

show\_facts.?- run.

30 Write a Prolog Program for backward Chaining. Incorporate required queries.

% Knowledge Base (Facts)

raining.

cloudy.

have\_umbrella.

% Rules (Backward Chaining)

take\_umbrella :-

cloudy,

have\_umbrella.

stay\_dry :-

take\_umbrella.

wet\_ground :-

raining.

% Query interface

ask(Query) :-

(call(Query) ->

write('Yes, '), write(Query), write(' is true.'), nl

;

write('No, '), write(Query), write(' is not provable.'), nl

).

?- ask(stay\_dry).

Yes, stay\_dry is true.

?- ask(wet\_ground).

Yes, wet\_ground is true.

?- ask(take\_umbrella).

Yes, take\_umbrella is true.

?- ask(snowing).

No, snowing is not provable.

31. Write a Prolog program to find the number of vowels

% Define vowels

is\_vowel(a).

is\_vowel(e).

is\_vowel(i).

is\_vowel(o).

is\_vowel(u).

% Base case: empty list has 0 vowels

count\_vowels([], 0).

% Recursive case: head is a vowel

count\_vowels([H|T], Count) :-

is\_vowel(H),

count\_vowels(T, RestCount),

Count is RestCount + 1.

% Recursive case: head is not a vowel

count\_vowels([H|T], Count) :-

\+ is\_vowel(H),

count\_vowels(T, Count).

?- count\_vowels([h,e,l,l,o], Count).

Count = 2.

?- count\_vowels([p,r,o,l,o,g], Count).

Count = 2.

?- count\_vowels([], Count).

Count = 0.

32.Write a Prolog program to implement pattern matching

% match(Pattern, Target) succeeds if Pattern matches the start of Target

match([], \_). % An empty pattern always matches

match([H1|T1], [H1|T2]) :-

match(T1, T2).

% pattern\_match(Pattern, Target) succeeds if Pattern appears anywhere in Target

pattern\_match(Pattern, Target) :-

match(Pattern, Target). % Try matching at the beginning

pattern\_match(Pattern, [\_|T]) :-

pattern\_match(Pattern, T). % Skip one element and try again

**?- pattern\_match([2,3], [1,2,3,4]).**

**true.**

**?- pattern\_match([3,5], [1,2,3,4,5]).**

**false.**

**?- pattern\_match([], [1,2,3]).**

**true.**

33. Create a Web Blog using Word press to demonstrate Anchor Tag, Title Tag, etc.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Learn Anchor Tags and HTML in WordPress | My Blog</title>

</head>

<body>

<article>

<h1>Demonstrating HTML Elements in WordPress</h1>

… <p>Anchor tags can wrap other elements, like images. Here's an example:</p>

<a href="#top"><img src="https://via.placeholder.com/150" alt="Sample Image"></a>

<p>Click the image to return to the top.</p>

<p id="top"><a href="#top">Back to Top</a></p>

</article>

</body>

</html>