1 Write the python program to solve 8-Puzzle problem

class Puzzle:

def \_\_init\_\_(self, b, p="", g=0):

self.b, self.p, self.g, self.h = b, p, g, sum(abs((v - 1) % 3 - i % 3) + abs((v - 1) // 3 - i // 3) for i, v in enumerate(b) if v)

def moves(self):

i = self.b.index(0)

for d, p in zip("UDLR", (-3, 3, -1, 1)):

if 0 <= i + p < 9 and not (i % 3 == 0 and p == -1) and not (i % 3 == 2 and p == 1):

nb = self.b[:]

nb[i], nb[i + p] = nb[i + p], nb[i]

yield Puzzle(nb, self.p + d, self.g + 1)

def solve(s):

q, v = [Puzzle(s)], {tuple(s)}

while q:

n = min(q, key=lambda x: x.g + x.h)

q.remove(n)

if n.b == [1, 2, 3, 4, 5, 6, 7, 8, 0]: return n.p

for m in n.moves():

if tuple(m.b) not in v: q.append(m); v.add(tuple(m.b))

return "No Solution"

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

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}")s

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

def print\_board(board):

for row in board:

print(" | ".join(row))

print("-" \* 9)

# Function to check for a win

def check\_win(board, player):

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

# Function to check if the board is full

def is\_full(board):

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

# Function to play the game

def play\_game():

board = [[" " for \_ in range(3)] for \_ in range(3)]

players = ["X", "O"]

turn = 0

while True:

print\_board(board)

row = int(input(f"Player {players[turn]}, enter row (0-2): "))

col = int(input(f"Player {players[turn]}, enter col (0-2): "))

if board[row][col] == " ":

board[row][col] = players[turn]

if check\_win(board, players[turn]):

print\_board(board)

print(f"Player {players[turn]} wins!")

break

if is\_full(board):

print\_board(board)

print("It's a tie!")

break

turn = 1 - turn # Switch turn

else:

print("Cell already taken, try again!")

# Start the game

play\_game()

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.

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.

25 Write a Prolog program to implement Monkey Banana Problem

% Initial state: (MonkeyPos, ChairPos, HasBanana)

state(at\_door, on\_floor, no).

state(at\_window, on\_floor, no).

state(at\_banana, on\_floor, no).

state(at\_banana, on\_chair, yes). % Goal state

% Actions

move(state(M, on\_floor, no), walk(M, NewM), state(NewM, on\_floor, no)).

move(state(M, on\_floor, no), push\_chair(M, NewM), state(NewM, on\_floor, no)).

move(state(M, on\_floor, no), climb\_chair, state(M, on\_chair, no)).

move(state(M, on\_chair, no), grab\_banana, state(M, on\_chair, yes)).

% Solve the problem

solve(State, []) :- State = state(\_, \_, yes). % Goal state

solve(State, [Action | Actions]) :-

move(State, Action, NewState),

solve(NewState, Actions).

% Example Query:

% ?- solve(state(at\_door, on\_floor, no), Actions).

% Output: Actions = [walk(at\_door, at\_banana), climb\_chair, grab\_banana].

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

% Define the graph with heuristic values

edge(a, b, 4).

edge(a, c, 3).

edge(b, d, 5).

edge(b, e, 12).

edge(c, f, 10).

edge(c, g, 8).

edge(e, h, 7).

edge(f, i, 6).

edge(g, j, 9).

% Define heuristic values for nodes

heuristic(a, 7).

heuristic(b, 6).

heuristic(c, 4).

heuristic(d, 5).

heuristic(e, 3).

heuristic(f, 2).

heuristic(g, 6).

heuristic(h, 5).

heuristic(i, 1).

heuristic(j, 4).

% Best First Search Algorithm

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

best\_first([[Start]], Goal, Path).

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

best\_first([CurrentPath | OtherPaths], Goal, Solution) :-

CurrentPath = [CurrentNode | \_],

findall([Next, CurrentNode | CurrentPath],

(edge(CurrentNode, Next, \_), \+ member(Next, CurrentPath)),

NewPaths),

sort\_by\_heuristic(NewPaths, SortedPaths),

append(SortedPaths, OtherPaths, UpdatedQueue),

best\_first(UpdatedQueue, Goal, Solution).

% Sorting paths based on heuristic values

sort\_by\_heuristic(Paths, SortedPaths) :-

map\_list\_to\_pairs(evaluate\_path, Paths, Paired),

keysort(Paired, SortedPaired),

pairs\_values(SortedPaired, SortedPaths).

evaluate\_path([Node | \_], H) :-

heuristic(Node, H).

28 Write the prolog program for Medical Diagnosis

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

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

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