**PYTHON CODES**

1. **8-Puzzle problem**

import heapq

def heuristic(state, goal):

return sum(abs(b % 3 - g % 3) + abs(b // 3 - g // 3) for b, g in zip(state, goal) if b)

def astar\_search(start, goal):

open\_list, closed = [(0, start)], set()

came\_from, cost = {}, {tuple(start): 0}

while open\_list:

\_, current = heapq.heappop(open\_list)

if current == goal:

break

closed.add(tuple(current))

for next\_state in get\_neighbors(current):

if tuple(next\_state) not in closed:

new\_cost = cost[tuple(current)] + 1

if tuple(next\_state) not in cost or new\_cost < cost[tuple(next\_state)]:

cost[tuple(next\_state)] = new\_cost

priority = new\_cost + heuristic(next\_state, goal)

heapq.heappush(open\_list, (priority, next\_state))

came\_from[tuple(next\_state)] = current

return came\_from, cost

def get\_neighbors(state):

neighbors, zero = [], state.index(0)

moves = [(-3, 0), (3, 0), (-1, -1), (1, 1)]

for move, col in moves:

new\_pos = zero + move

if 0 <= new\_pos < len(state) and (col == 0 or zero // 3 == new\_pos // 3):

neighbor = state[:]

neighbor[zero], neighbor[new\_pos] = neighbor[new\_pos], neighbor[zero]

neighbors.append(neighbor)

return neighbors

def reconstruct\_path(came\_from, start, goal):

current, path = goal, [goal]

while current != start:

current = came\_from[tuple(current)]

path.append(current)

return path[::-1]

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

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

came\_from, cost = astar\_search(start\_state, goal\_state)

path = reconstruct\_path(came\_from, start\_state, goal\_state)

print("Solution path:")

for state in path:

print(state)

print("Cost:", cost[tuple(goal\_state)])

2)**8-QUEEN PROBLEM:**

N = 8

def solveNQueens(board, col):

if col == N:

print(board)

return True

for i in range(N):

if isSafe(board, i, col):

board[i][col] = 1

if solveNQueens(board, col + 1):

return True

board[i][col] = 0

return False

def isSafe(board, row, col):

for x in range(col):

if board[row][x] == 1:

return False

for x, y in zip(range(row, -1, -1), range(col, -1, -1)):

if board[x][y] == 1:

return False

for x, y in zip(range(row, N, 1), range(col, -1, -1)):

if board[x][y] == 1:

return False

return True

board = [[0 for x in range(N)] for y in range(N)]

if not solveNQueens(board, 0):

print("No solution found")

**3)A STAR ALGORITHM:**

import heapq

def a\_star(start, goal, graph, heuristic):

open\_set = [(heuristic[start], start)]

g\_score = {node: float('inf') for node in graph}

g\_score[start] = 0

came\_from = {}

while open\_set:

\_, current = heapq.heappop(open\_set)

if current == goal:

path = []

while current in came\_from:

path.append(current)

current = came\_from[current]

path.append(start)

return path[::-1]

for neighbor, cost in graph[current].items():

tentative\_g\_score = g\_score[current] + cost

if tentative\_g\_score < g\_score[neighbor]:

came\_from[neighbor] = current

g\_score[neighbor] = tentative\_g\_score

heapq.heappush(open\_set, (tentative\_g\_score + heuristic[neighbor], neighbor))

return None

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

graph = {

'A': {'B': 1, 'C': 4},

'B': {'A': 1, 'C': 2, 'D': 5},

'C': {'A': 4, 'B': 2, 'D': 1},

'D': {'B': 5, 'C': 1}

}

heuristic = {

'A': 7,

'B': 6,

'C': 2,

'D': 0

}

start = 'A'

goal = 'D'

path = a\_star(start, goal, graph, heuristic)

if path:

print("Path found:", path)

else:

print("No path found")

**4)BFS:**

from collections import deque

def bfs(graph, start):

visited = set()

queue = deque([start])

visited.add(start)

while queue:

vertex = queue.popleft()

print(vertex, end=" ")

for neighbor in graph[vertex]:

if neighbor not in visited:

visited.add(neighbor)

queue.append(neighbor)

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

graph = {

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

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

'C': ['A', 'F'],

'D': ['B'],

'E': ['B', 'F'],

'F': ['C', 'E']

}

print("BFS Traversal starting from node 'A':")

bfs(graph, 'A')

**5)ALPHA BETA PRUNINBG:**

print("Alpha-Beta Pruning Algorithm")

MAX, MIN = 1000, -1000

def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):

if depth == 3:

return values[nodeIndex]

if maximizingPlayer:

best = MIN

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i, False, values, alpha, beta)

best = max(best, val)

alpha = max(alpha, best)

if beta <= alpha:

break

return best

else:

best = MAX

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i, True, values, alpha, beta)

best = min(best, val)

beta = min(beta, best)

if beta <= alpha:

break

return best

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

values = [3, 5, 6, 9, 1, 2, -1, 0]

print(values)

print("The optimal value is:", minimax(0, 0, True, values, MIN, MAX))

**6)CRYPTO ARTHMETIC:**

from itertools import permutations

def solve\_cryptarithmetic(puzzle):

parts = puzzle.split('+')

left = parts[0].strip()

right, result = parts[1].split('=')

right = right.strip()

result = result.strip()

letters = set(left + right + result)

for perm in permutations(range(10), len(letters)):

mapping = dict(zip(letters, perm))

if mapping[left[0]] == 0 or mapping[right[0]] == 0 or mapping[result[0]] == 0:

continue

left\_num = int(''.join(str(mapping[char]) for char in left))

right\_num = int(''.join(str(mapping[char]) for char in right))

result\_num = int(''.join(str(mapping[char]) for char in result))

if left\_num + right\_num == result\_num:

return mapping

return None # No solution found

# Example usage:

puzzle = "BASE + BALL = GAMES"

solution = solve\_cryptarithmetic(puzzle)

if solution:

print("Solution found:")

for letter, digit in solution.items():

print(f"{letter}: {digit}")

else:

print("No solution found.")

**7)DFS:**

def dfs(graph, start, visited=None):

if visited is None:

visited = set()

visited.add(start)

print(start, end=" ")

for neighbor in graph[start]:

if neighbor not in visited:

dfs(graph, neighbor, visited)

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

graph = {

1: [2, 3],

2: [1, 4],

3: [1, 5],

4: [2],

5: [3]

}

print("DFS Traversal starting from node 1:")

dfs(graph, 1)

**8)FEED FORWARD:**

**1)**import numpy as np

# Sigmoid activation function and its derivative

def sigmoid(x):

return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(x):

return x \* (1 - x)

# Initialize parameters

def initialize\_parameters(input\_dim, hidden\_dim, output\_dim):

np.random.seed(42) # For reproducibility

W1 = np.random.randn(input\_dim, hidden\_dim) # Weights for input to hidden layer

b1 = np.zeros((1, hidden\_dim)) # Biases for hidden layer

W2 = np.random.randn(hidden\_dim, output\_dim) # Weights for hidden to output layer

b2 = np.zeros((1, output\_dim)) # Biases for output layer

return W1, b1, W2, b2

# Forward propagation

def forward\_propagation(X, W1, b1, W2, b2):

Z1 = np.dot(X, W1) + b1

A1 = sigmoid(Z1)

Z2 = np.dot(A1, W2) + b2

A2 = sigmoid(Z2)

return A2, A1

# Compute cost (mean squared error)

def compute\_cost(A2, Y):

m = Y.shape[0]

cost = np.sum((A2 - Y) \*\* 2) / (2 \* m)

return cost

# Backward propagation

def backward\_propagation(X, Y, A2, A1, W2):

m = X.shape[0]

dA2 = A2 - Y

dZ2 = dA2 \* sigmoid\_derivative(A2)

dW2 = np.dot(A1.T, dZ2) / m

db2 = np.sum(dZ2, axis=0, keepdims=True) / m

dA1 = np.dot(dZ2, W2.T)

dZ1 = dA1 \* sigmoid\_derivative(A1)

dW1 = np.dot(X.T, dZ1) / m

db1 = np.sum(dZ1, axis=0, keepdims=True) / m

return dW1, db1, dW2, db2

# Update parameters using gradient descent

def update\_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, learning\_rate):

W1 -= learning\_rate \* dW1

b1 -= learning\_rate \* db1

W2 -= learning\_rate \* dW2

b2 -= learning\_rate \* db2

return W1, b1, W2, b2

# Training the neural network

def train(X, Y, hidden\_dim, epochs, learning\_rate):

input\_dim = X.shape[1]

output\_dim = Y.shape[1]

W1, b1, W2, b2 = initialize\_parameters(input\_dim, hidden\_dim, output\_dim)

for epoch in range(epochs):

A2, A1 = forward\_propagation(X, W1, b1, W2, b2)

cost = compute\_cost(A2, Y)

dW1, db1, dW2, db2 = backward\_propagation(X, Y, A2, A1, W2)

W1, b1, W2, b2 = update\_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, learning\_rate)

if epoch % 100 == 0:

print(f"Epoch {epoch}: Cost {cost}")

return W1, b1, W2, b2

# Predict function

def predict(X, W1, b1, W2, b2):

A2, \_ = forward\_propagation(X, W1, b1, W2, b2)

return A2

# Sample data (for demonstration purposes)

X = np.array([

[0.1, 0.2],

[0.9, 0.8],

[0.8, 0.9],

[0.2, 0.1]

])

Y = np.array([

[0.1],

[0.9],

[0.8],

[0.2]

])

# Training parameters

hidden\_dim = 5

epochs = 1000

learning\_rate = 0.01

# Train the model

W1, b1, W2, b2 = train(X, Y, hidden\_dim, epochs, learning\_rate)

# Predict

predictions = predict(X, W1, b1, W2, b2)

print("Predicted Output after training:")

print(predictions)

**2)** from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.neural\_network import MLPClassifier

from sklearn.metrics import accuracy\_score

# Load the Iris dataset

iris = load\_iris()

X, y = iris.data, iris.target

# Normalize features

scaler = StandardScaler()

X = scaler.fit\_transform(X)

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25, random\_state=42)

# Create and train the feedforward neural network

model = MLPClassifier(hidden\_layer\_sizes=(10,), max\_iter=50, random\_state=42)

model.fit(X\_train, y\_train)

# Evaluate the model

y\_pred = model.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Test Accuracy: {accuracy:.4f}")

# Make a prediction on a new sample

new\_sample = scaler.transform([[5.1, 3.5, 1.4, 0.2]])

prediction = model.predict(new\_sample)

print(f"Prediction for the new sample: {iris.target\_names[prediction[0]]}")

**9)MAP COLORING:**

class CSP:

def \_\_init\_\_(self, variables, domains): # Corrected the method name

self.variables = variables

self.domains = domains

def is\_consistent(self, variable, assignment):

return all(assignment[neighbor] != assignment[variable] for neighbor in self.variables[variable] if neighbor in assignment)

def backtracking\_search(self, assignment={}):

if len(assignment) == len(self.variables):

return assignment

unassigned = [var for var in self.variables if var not in assignment]

first\_unassigned = unassigned[0]

for value in self.domains[first\_unassigned]:

assignment[first\_unassigned] = value

if self.is\_consistent(first\_unassigned, assignment):

result = self.backtracking\_search(assignment)

if result is not None:

return result

assignment.pop(first\_unassigned)

return None

def main():

# Define the variables and domains for the Map Coloring problem

variables = {

'WA': ['NT', 'SA'],

'NT': ['WA', 'SA', 'Q'],

'SA': ['WA', 'NT', 'Q', 'NSW', 'V'],

'Q': ['NT', 'SA', 'NSW'],

'NSW': ['Q', 'SA', 'V'],

'V': ['SA', 'NSW']

}

domains = {

'WA': ['red', 'green', 'blue'],

'NT': ['red', 'green', 'blue'],

'SA': ['red', 'green', 'blue'],

'Q': ['red', 'green', 'blue'],

'NSW': ['red', 'green', 'blue'],

'V': ['red', 'green', 'blue']

}

csp = CSP(variables, domains)

solution = csp.backtracking\_search()

if solution is not None:

print("Solution found:")

for var, val in solution.items():

print(f"{var}: {val}")

else:

print("No solution found.")

if \_\_name\_\_ == "\_\_main\_\_": # Corrected the variable names

main()

**10)MIN MAX ALGORITHM:**

import math

def minimax (curDepth, nodeIndex,

maxTurn, scores,

targetDepth):

# base case : targetDepth reached

if (curDepth == targetDepth):

return scores[nodeIndex]

if (maxTurn):

return max(minimax(curDepth + 1, nodeIndex \* 2,

False, scores, targetDepth),

minimax(curDepth + 1, nodeIndex \* 2 + 1,

False, scores, targetDepth))

else:

return min(minimax(curDepth + 1, nodeIndex \* 2,

True, scores, targetDepth),

minimax(curDepth + 1, nodeIndex \* 2 + 1,

True, scores, targetDepth))

# Driver code

scores = [3, 5, 2, 9, 12, 5, 23, 23]

treeDepth = math.log(len(scores), 2)

print("The optimal value is : ", end = "")

print(minimax(0, 0, True, scores, treeDepth))

**11)MISSINORIES-CALIBERS:**

from collections import deque

def is\_valid\_state(m, c):

return 0 <= m <= 3 and 0 <= c <= 3 and (m == 0 or m >= c) and ((3 - m) == 0 or (3 - m) >= (3 - c))

def get\_next\_states(state):

m\_left, c\_left, b\_left, m\_right, c\_right, b\_right = state

next\_states = []

for i in range(3):

for j in range(3):

if 1 <= i + j <= 2:

if b\_left:

new\_state = (m\_left - i, c\_left - j, 0, m\_right + i, c\_right + j, 1)

else:

new\_state = (m\_left + i, c\_left + j, 1, m\_right - i, c\_right - j, 0)

if is\_valid\_state(new\_state[0], new\_state[1]) and is\_valid\_state(new\_state[3], new\_state[4]):

next\_states.append(new\_state)

return next\_states

def bfs():

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

queue = deque([(start, [])])

visited = {start}

while queue:

state, path = queue.popleft()

if state == goal:

return path + [goal]

for next\_state in get\_next\_states(state):

if next\_state not in visited:

visited.add(next\_state)

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

return None

def print\_solution(solution):

if solution:

print("Solution found!")

for i, state in enumerate(solution):

print(f"Step {i + 1}: {state[:3]} || {state[3:]}")

else:

print("No solution found.")

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

print\_solution(bfs())

**12)TIC TAC TOE:**

def print\_board(x\_state, z\_state):

board = [str(i) if x\_state[i] == z\_state[i] == 0 else ('X' if x\_state[i] else 'O') for i in range(9)]

print(f" {board[0]} | {board[1]} | {board[2]} ")

print("---|---|---")

print(f" {board[3]} | {board[4]} | {board[5]} ")

print("---|---|---")

print(f" {board[6]} | {board[7]} | {board[8]} ")

def check\_win(x\_state, z\_state):

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

for win in wins:

if sum(x\_state[i] for i in win) == 3:

print("X won the game")

return True

elif sum(z\_state[i] for i in win) == 3:

print("O won the game")

return True

return False

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

total\_turns = 9

x\_state = [0] \* 9

z\_state = [0] \* 9

turn = 1 # 1 for X and 0 for O

print("Welcome to TIC-TAC-TOE")

while True:

print\_board(x\_state, z\_state)

player = 'X' if turn == 1 else 'O'

print(f"{player}'s Chance")

value = int(input("Please enter a value (0-8): "))

if not (0 <= value <= 8 and x\_state[value] == z\_state[value] == 0):

print("Invalid move! Please choose an empty cell (0-8).")

continue

if turn == 1:

x\_state[value] = 1

else:

z\_state[value] = 1

total\_turns -= 1

if check\_win(x\_state, z\_state) or total\_turns == 0:

print("GAME OVER")

print\_board(x\_state, z\_state)

break

turn = 1 – turn

**13)TSP:**

import itertools

import math

def calculate\_distance(p1, p2):

return math.hypot(p1[0] - p2[0], p1[1] - p2[1])

def total\_distance(points, order):

return sum(calculate\_distance(points[order[i]], points[order[(i + 1) % len(order)]]) for i in range(len(order)))

def tsp\_bruteforce(points):

return min(

(total\_distance(points, perm), perm)

for perm in itertools.permutations(range(len(points)))

)

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

points = [(0, 0), (1, 5), (5, 2), (6, 6)]

min\_distance, optimal\_order = tsp\_bruteforce(points)

print("Minimum Distance:", min\_distance)

print("Optimal Order:", optimal\_order)

**14)VACCUME CLEANER:**

a=[[1,0,1,0],[1,1,1,1],[1,0,1,1],[1,0,1,1]]

print("Room With dust are represented as 1 and Room With NO Dust represented as 0\nRoom Structure with and without Dirt\n",a)

print("AGENT is Cleaning")

for i in range(4):

for j in range(4):

if(a[i][j]==1):

print("Agent Cleaned Location",i,j)

a[i][j]=0

print("Agent Cleaned Room",i+1)

print("Room After Cleaning \n",a)

**15)WATER JUG:**

from collections import deque

def solve\_water\_jug\_problem(cap1, cap2, target):

queue = deque([(0, 0, [])])

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

while queue:

jug1, jug2, path = queue.popleft()

if jug1 == target or jug2 == target:

path.append((jug1, jug2))

for step in path:

print(f"Jug1: {step[0]} liters, Jug2: {step[1]} liters")

return

for next\_jug1, next\_jug2 in [(cap1, jug2), (jug1, cap2), (0, jug2), (jug1, 0),

(jug1 - min(jug1, cap2 - jug2), jug2 + min(jug1, cap2 - jug2)),

(jug1 + min(jug2, cap1 - jug1), jug2 - min(jug2, cap1 - jug1))]:

if (next\_jug1, next\_jug2) not in visited and 0 <= next\_jug1 <= cap1 and 0 <= next\_jug2 <= cap2:

visited.add((next\_jug1, next\_jug2))

queue.append((next\_jug1, next\_jug2, path + [(jug1, jug2)]))

print("No solution found.")

# Example usage

solve\_water\_jug\_problem(4, 3, 2)

**16)DECISION TREE:**

**1)**import numpy as np

class TreeNode:

def \_\_init\_\_(self, feature\_index=None, threshold=None, left=None, right=None, value=None):

self.feature\_index = feature\_index

self.threshold = threshold

self.left = left

self.right = right

self.value = value

class DecisionTreeClassifier:

def \_\_init\_\_(self, max\_depth=None):

self.max\_depth = max\_depth

self.tree = None

def fit(self, X, y):

self.tree = self.\_build\_tree(X, y, depth=0)

def \_build\_tree(self, X, y, depth):

if len(y) == 0:

return None

if depth == self.max\_depth or len(np.unique(y)) == 1:

return TreeNode(value=np.bincount(y).argmax())

best\_split = self.\_find\_best\_split(X, y)

if not best\_split:

return TreeNode(value=np.bincount(y).argmax())

X\_left, y\_left, X\_right, y\_right = self.\_split\_data(X, y, best\_split['feature\_index'], best\_split['threshold'])

if len(y\_left) == 0 or len(y\_right) == 0:

return TreeNode(value=np.bincount(y).argmax())

left\_subtree = self.\_build\_tree(X\_left, y\_left, depth + 1)

right\_subtree = self.\_build\_tree(X\_right, y\_right, depth + 1)

return TreeNode(feature\_index=best\_split['feature\_index'], threshold=best\_split['threshold'], left=left\_subtree, right=right\_subtree)

def \_find\_best\_split(self, X, y):

best\_split = {}

best\_gini = float('inf')

for feature\_index in range(X.shape[1]):

thresholds = np.unique(X[:, feature\_index])

for threshold in thresholds:

X\_left, y\_left, X\_right, y\_right = self.\_split\_data(X, y, feature\_index, threshold)

gini = self.\_gini\_index(y\_left, y\_right)

if gini < best\_gini:

best\_gini = gini

best\_split = {'feature\_index': feature\_index, 'threshold': threshold}

return best\_split

def \_split\_data(self, X, y, feature\_index, threshold):

left\_mask = X[:, feature\_index] <= threshold

return X[left\_mask], y[left\_mask], X[~left\_mask], y[~left\_mask]

def \_gini\_index(self, y\_left, y\_right):

n\_left, n\_right = len(y\_left), len(y\_right)

if n\_left == 0 or n\_right == 0:

return 0

gini\_left = 1.0 - sum((np.sum(y\_left == c) / n\_left) \*\* 2 for c in np.unique(y\_left))

gini\_right = 1.0 - sum((np.sum(y\_right == c) / n\_right) \*\* 2 for c in np.unique(y\_right))

return (n\_left \* gini\_left + n\_right \* gini\_right) / (n\_left + n\_right)

def predict(self, X):

return np.array([self.\_predict\_sample(x, self.tree) for x in X])

def \_predict\_sample(self, x, node):

if node.value is not None:

return node.value

if x[node.feature\_index] <= node.threshold:

return self.\_predict\_sample(x, node.left)

else:

return self.\_predict\_sample(x, node.right)

def create\_synthetic\_data():

np.random.seed(42)

X = np.random.rand(100, 2)

y = (X[:, 0] + X[:, 1] > 1).astype(int)

return X, y

def train\_test\_split(X, y, test\_size=0.2):

indices = np.arange(X.shape[0])

np.random.shuffle(indices)

test\_size = int(len(y) \* test\_size)

train\_indices, test\_indices = indices[:-test\_size], indices[-test\_size:]

return X[train\_indices], X[test\_indices], y[train\_indices], y[test\_indices]

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

X, y = create\_synthetic\_data()

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y)

model = DecisionTreeClassifier(max\_depth=3)

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

accuracy = np.mean(y\_pred == y\_test)

print(f"Accuracy: {accuracy}")

**2)** import numpy as np

from sklearn.tree import DecisionTreeClassifier

from sklearn import tree

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.datasets import load\_iris

# Load the Iris dataset

iris = load\_iris()

X = iris.data

y = iris.target

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25, random\_state=42)

# Create Decision Tree Classifier

clf = DecisionTreeClassifier()

# Train the classifier

clf.fit(X\_train, y\_train)

# Predicting the test set

y\_pred = clf.predict(X\_test)

# Predicting a new sample

new\_sample = np.array([[5.1, 3.5, 1.4, 0.2]]) # Sample from the Iris dataset

prediction = clf.predict(new\_sample)

target\_names = iris.target\_names

print(f"Prediction for the new sample {new\_sample}: {target\_names[prediction[0]]}")

# Visualizing the Decision Tree

plt.figure(figsize=(12, 8))

tree.plot\_tree(clf, feature\_names=iris.feature\_names, class\_names=target\_names, filled=True)

plt.title("Decision Tree")

plt.show()

**PROLOG**

1)**SUM**

sum(0,0).

sum(N,Total):-

N>0,

N1 is N-1,

sum(N1,Result),

Total is N+Result.

**OUT PUT:**

%sum(3,Total).

Total = 6 .

2)**DOB**

person(seetha,"05-11-2005").

person(navya,"28-04-2004").

person(kavya,"05-09-2005").

get\_dob(Name,DOB):-

person(Name,DOB).

get\_name(DOB,Name):-

person(Name,DOB).

**OUT PUT:**

% get\_name("05-11-2005", Name).

Name = seetha

%get\_dob(seetha,DOB).

DOB = "05-11-2005".

3)**STUDENT-TEACHER**

studies(annya,ai).

studies(divya,ai).

studies(anjali,csbs).

studies(seetha,maths).

teaches(jk,ai).

teaches(jimin,csbs).

teaches(tae,maths).

teaches(jin,ai).

lecturer(Professor,Student):-

teaches(Professor,Course),

studies(Student,Course).

**OUT PUT:**

%

?- lecturer(jk,Student).

Student = annya .

%?- teaches(jimin,Student).

Student = csbs.

%?- teaches(jimin,Course).

Course = csbs.

%?- studies(annya,Course).

Course = ai.

%?- studies(seetha,Course).

Course = maths.

4)**PLANETS DB**

planet(jupiter,30000).

planet(earth,40000).

planet(satrun,7000).

planet(venus,8999).

planet(neptune,90000).

list\_of(Planets,Distance):-

findall(Planets,planet(Planets,\_),Planets).

list\_distance(Distance):-

findall(Distance,planet(\_,Distance),Planets).

list\_distance(Distance):-

findall(Distance,planet(\_,Distance),Distance).

**OUTPUT:**

%planet(earth,Distance).

Distance = 40000.

%planet(Planet,8999).

Planet = venus.

5)**TOWERS OF HANOI:**

% Move a single disk from Source peg to Destination peg

move\_disk(1, Source, Destination, \_) :-

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

% Move N disks from Source peg to Destination peg using Auxiliary peg

move\_disk(N, Source, Destination, Auxiliary) :-

N > 1,

N1 is N - 1,

move\_disk(N1, Source, Auxiliary, Destination),

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

move\_disk(N1, Auxiliary, Destination, Source).

% Solve the Towers of Hanoi puzzle with N disks

towers\_of\_hanoi(N) :-

move\_disk(N, 'Source', 'Destination', 'Auxiliary').

**OUTPUT:**

towers\_of\_hanoi(3).

Move disk 1 from Source to Destination

Move disk 2 from Source to Auxiliary

Move disk 1 from Destination to Auxiliary

Move disk 3 from Source to Destination

Move disk 1 from Auxiliary to Source

Move disk 2 from Auxiliary to Destination

Move disk 1 from Source to Destination

True

6)**BIRD CAN FLY OR NOT:**

% Define birds and their ability to fly

bird(sparrow, fly).

bird(pigeon, fly).

bird(squirrel, cannotfly).

bird(tan, cannotfly).

% Predicate to check if a bird can fly

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

% Query to find all birds that can fly

fly\_of(Birds) :- findall(Bird, can\_fly(Bird), Birds).

**OUTPUT:**

can\_fly(sparrow).

true.

?- fly\_of(Birds).

Birds = [sparrow, pigeon].

?- bird(sparrow,Fly).

Fly = fly.

?- bird(sparrow,fly).

True

7)**FAMILY TREE:**

% Facts: parent relationships

parent(john, mary).

parent(john, joe).

parent(susan, mary).

parent(susan, joe).

parent(mary, ann).

parent(mary, tom).

parent(david, ann).

parent(david, tom).

% Rules: defining relationships based on parent relationships

% X is a child of Y if Y is a parent of X

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

% X is a grandparent of Z if X is a parent of Y and Y is a parent of Z

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

% X is a grandchild of Z if Z is a grandparent of X

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

% X is a sibling of Y if they share at least one parent

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

% X is a cousin of Y if their parents are siblings

cousin(X, Y) :-

parent(A, X),

parent(B, Y),

sibling(A, B).

% X is an aunt or uncle of Y if X is a sibling of Y's parent

aunt\_or\_uncle(X, Y) :-

parent(Z, Y),

sibling(X, Z).

% X is a niece or nephew of Y if Y is a sibling of X's parent

niece\_or\_nephew(X, Y) :-

parent(Z, X),

sibling(Z, Y).

% X is a descendant of Y if Y is a parent of X or Y is an ancestor of Z who is a parent of X

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

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

% X is an ancestor of Y if X is a parent of Y or X is an ancestor of Z who is a parent of Y

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

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

**OUT PUT:**

% Find all children of john

% ?- child(X, john).

% Find all siblings of mary

% ?- sibling(X, mary).

% Find all grandparents of ann

% ?- grandparent(X, ann).

% Find all cousins of tom

% ?- cousin(X, tom).

% Find all descendants of john

% ?- descendant(X, john).

% Find all ancestors of tom

% ?- ancestor(X, tom).

8) **DIETING SYSTEM BASED ON DIEASEASE:**

% Define diseases and their symptoms

dise(heartatack,"heart").

dise(fever,"hot").

dise(cold,"runn").

% Define dietary recommendations based on symptoms

diet(heart,"Avoid oil food").

diet(hot,"Avoid Cool food").

diet(runn,"Avoid cool food").

food(Dise,Diet):-

dise(Dise,Fa),

diet(Fa,Diet).

**OUT PUT:**

%Fa=symptom

%dise(fever,Fa).

Fa = "hot".

%?- diet(runn,Diet).

Diet = "Avoid cood food".

9)**MONKEY-BANANA:**

% Initial state

initial\_state(on\_ground).

% Actions and their effects

action(on\_ground, push\_box, box\_under\_banana).

action(box\_under\_banana, climb, on\_box).

action(on\_box, reach, banana\_reached).

% Plan generation

plan(State, [], State).

plan(State1, [Action | Rest], Goal) :-

action(State1, Action, State2),

plan(State2, Rest, Goal).

**OUTPUT:**

% ?- initial\_state(State), plan(State, Plan, banana\_reached).

%State = on\_ground,

%Plan = [push\_box, climb, reach]

?- initial\_state(State).

State = on\_ground.

?- action(on\_ground, Action, NextState).

Action = push\_box,

NextState = box\_under\_banana.

?- initial\_state(State), plan(State, Plan, on\_box).

State = on\_ground,

Plan = [push\_box, climb] .

?- initial\_state(State), action(State, \_, NextState).

State = on\_ground,

NextState = box\_under\_banana.

10)**FRUIT COLOR:**

% Define fruits and their colors

fruit\_color(apple, red).

fruit\_color(banana, yellow).

fruit\_color(grape, purple).

fruit\_color(orange, orange).

fruit\_color(lemon, yellow).

fruit\_color(cherry, red).

% Query to find fruits by color using backtracking

find\_fruits\_by\_color(Color, Fruit) :-

fruit\_color(Fruit, Color).

**OUT PUT:**

% ?- find\_fruits\_by\_color(yellow, Fruit).

% Fruit = banana ;

% Fruit = lemon.

%fruit\_color(lemon,Color).

%Color = yellow.

%fruit\_color(apple, red).

%true

11)**BFS:**

% Define the graph using edge facts

edge(a, b).

edge(a, c).

edge(b, d).

edge(b, e).

edge(c, f).

edge(d, g).

edge(e, g).

edge(f, g).

% BFS Algorithm

bfs(Start, Goal, Path) :-

bfs\_helper([[Start]], Goal, RevPath),

reverse(RevPath, Path).

% Helper predicate for BFS

bfs\_helper([[Goal|Rest] | \_], Goal, [Goal|Rest]).

bfs\_helper([[Node|Path] | Paths], Goal, Result) :-

findall([NextNode, Node | Path],

(edge(Node, NextNode), \+ member(NextNode, [Node | Path])),

NewPaths),

append(Paths, NewPaths, UpdatedPaths),

bfs\_helper(UpdatedPaths, Goal, Result).

**OUT PUT:**

% ?- bfs(a, g, Path).

% Path = [a, b, d, g] ;

% Path = [a, b, e, g] ;

% Path = [a, c, f, g] ;

12)**MEDICAL DIAGNOSIS:**

% Define diseases and their symptoms

dise(heart\_attack, "heart").

dise(fever, "hot").

dise(cold, "runny\_nose").

dise(diabetes, "sugar").

dise(allergy, "itchy").

dise(migraine, "headache").

dise(stomach\_ache, "pain").

dise(arthritis, "joint\_pain").

% Define dietary recommendations based on symptoms

diet("heart", "Avoid oily food").

diet("hot", "Avoid cool food").

diet("runny\_nose", "Avoid cold food").

diet("sugar", "Avoid sugary food").

diet("itchy", "Avoid allergens").

diet("headache", "Stay hydrated and avoid caffeine").

diet("pain", "Avoid spicy food").

diet("joint\_pain", "Avoid heavy lifting").

% Match disease to dietary recommendation

food(Dise, Diet) :-

dise(Dise, Symptom),

diet(Symptom, Diet).

**OUT PUT:**

% Query to get the dietary recommendation for 'arthritis'

% ?- food(arthritis, Diet).

% Diet = "Avoid heavy lifting".

% Query to get the symptom for 'diabetes'

% ?- dise(diabetes, Symptom).

% Symptom = "sugar".

% food(arthritis,"Avoid heavy lifting" ).

%true.

13)**FORWARD CHAINING:**

% Facts

likes(john, pizza).

likes(mary, sushi).

% Rule

eats\_out(X) :- likes(X, \_).

% Forward chaining process

forward\_chaining :-

% Collect all facts

findall(X, likes(X, \_), People),

% Apply the rule to each person

forall(member(Person, People), (eats\_out(Person), assert(fact(eats\_out(Person))))).

% Query to list all facts

list\_facts :-

fact(Fact),

write(Fact), nl,

fail.

list\_facts.

**OUT PUT:**

% forward\_chaining, list\_facts.

%eats\_out(john)

%eats\_out(mary)

%true.

14)**BACKWARD CHAINING:**

% Facts

likes(john, pizza).

likes(mary, sushi).

% Rule

eats\_out(X) :- likes(X, \_).

% Backward chaining process

% Check if the goal can be satisfied

backward\_chaining(Goal) :-

call(Goal), !. % Use call/1 to execute the goal

**OUTPUT:**

% Query to check if john eats out

% ?- backward\_chaining(eats\_out(john)).

% Query to check if mary eats out

% ?- backward\_chaining(eats\_out(mary)).

**14)2forward chaining:**

% Initial facts

fact(likes(john, pizza)).

fact(likes(mary, sushi)).

% Rule for forward chaining

derive\_fact :-

fact(likes(Person, \_)),

\+ fact(eats\_out(Person)),

assert(fact(eats\_out(Person))).

% Forward chaining process

forward\_chaining :-

repeat,

( derive\_fact -> fail ; ! ),

write('Forward chaining completed'), nl.

% Query to list all facts

list\_facts :-

fact(Fact),

write(Fact), nl,

fail.

list\_facts.

**Output:**

?- list\_facts.

likes(john,pizza)

likes(mary,sushi)

true.

fact(Fact).

Fact = likes(john, pizza) ;

Fact = likes(mary, sushi).