**2. 8-PUZZLE**

from queue import PriorityQueue

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

class PuzzleState:

def \_\_init\_\_(self, puzzle, parent=None, move="Initial"):

self.puzzle = puzzle

self.parent = parent

self.move = move

self.h = self.calculate\_heuristic()

def \_\_lt\_\_(self, other):

return self.h < other.h

def \_\_eq\_\_(self, other):

return self.puzzle == other.puzzle

def \_\_hash\_\_(self):

return hash(str(self.puzzle))

def \_\_str\_\_(self):

return f"Move: {self.move}\n{self.print\_puzzle()}"

def print\_puzzle(self):

res = ""

for row in self.puzzle:

res += " ".join(map(str, row)) + "\n"

return res

def calculate\_heuristic(self):

h = 0

for i in range(3):

for j in range(3):

if self.puzzle[i][j] != goal\_state[i][j] and self.puzzle[i][j] != 0:

h += 1

return h

def get\_blank\_position(self):

for i in range(3):

for j in range(3):

if self.puzzle[i][j] == 0:

return i, j

def get\_possible\_moves(self):

i, j = self.get\_blank\_position()

possible\_moves = []

if i > 0:

possible\_moves.append((-1, 0))

if i < 2:

possible\_moves.append((1, 0))

if j > 0:

possible\_moves.append((0, -1))

if j < 2:

possible\_moves.append((0, 1))

return possible\_moves

def make\_move(self, move):

new\_puzzle = [row[:] for row in self.puzzle]

i, j = self.get\_blank\_position()

x, y = move

new\_puzzle[i][j], new\_puzzle[i + x][j + y] = new\_puzzle[i + x][j + y], new\_puzzle[i][j]

return PuzzleState(new\_puzzle, parent=self, move=f"Move: {new\_puzzle[i + x][j + y]}")

def best\_first\_search(initial\_state):

visited = set()

priority\_queue = PriorityQueue()

priority\_queue.put(initial\_state)

while not priority\_queue.empty():

current\_state = priority\_queue.get()

if current\_state.puzzle == goal\_state:

return current\_state

visited.add(current\_state)

possible\_moves = current\_state.get\_possible\_moves()

for move in possible\_moves:

new\_state = current\_state.make\_move(move)

if new\_state not in visited:

priority\_queue.put(new\_state)

def print\_solution(solution):

if solution is None:

print("No solution found.")

else:

path = []

current\_state = solution

while current\_state:

path.append(current\_state)

current\_state = current\_state.parent

path.reverse()

for state in path:

print(state)

initial\_state = PuzzleState([[1, 2, 3], [0, 5, 6], [4, 7, 8]])

solution = best\_first\_search(initial\_state)

print\_solution(solution)

**3.A\*SEARCH**

import math

import heapq

class Cell:

def \_init\_(self):

self.parent\_i = 0

self.parent\_j = 0

self.f = float('inf')

self.g = float('inf')

self.h = 0

ROW = 9

COL = 10

def is\_valid(row, col):

return (row >= 0) and (row < ROW) and (col >= 0) and (col < COL)

def is\_unblocked(grid, row, col):

return grid[row][col] == 1

def is\_destination(row, col, dest):

return row == dest[0] and col == dest[1]

def calculate\_h\_value(row, col, dest):

return ((row - dest[0]) \* 2 + (col - dest[1]) \* 2) \*\* 0.5

def trace\_path(cell\_details, dest):

print("The Path is:")

path = []

row = dest[0]

col = dest[1]

while not (cell\_details[row][col].parent\_i == row and cell\_details[row][col].parent\_j == col):

path.append((row, col))

temp\_row = cell\_details[row][col].parent\_i

temp\_col = cell\_details[row][col].parent\_j

row = temp\_row

col = temp\_col

path.append((row, col))

path.reverse()

for i in path:

print("->", i, end=" ")

print()

def a\_star\_search(grid, src, dest):

if not is\_valid(src[0], src[1]) or not is\_valid(dest[0], dest[1]):

print("Source or destination is invalid")

return

if not is\_unblocked(grid, src[0], src[1]) or not is\_unblocked(grid, dest[0], dest[1]):

print("Source or the destination is blocked")

return

if is\_destination(src[0], src[1], dest):

print("We are already at the destination")

return

closed\_list = [[False for \_ in range(COL)] for \_ in range(ROW)]

cell\_details = [[Cell() for \_ in range(COL)] for \_ in range(ROW)]

i, j = src

cell\_details[i][j].f = 0

cell\_details[i][j].g = 0

cell\_details[i][j].h = 0

cell\_details[i][j].parent\_i = i

cell\_details[i][j].parent\_j = j

open\_list = []

heapq.heappush(open\_list, (0.0, i, j))

found\_dest = False

while open\_list:

p = heapq.heappop(open\_list)

i, j = p[1], p[2]

closed\_list[i][j] = True

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

for dir in directions:

new\_i, new\_j = i + dir[0], j + dir[1]

if is\_valid(new\_i, new\_j) and is\_unblocked(grid, new\_i, new\_j) and not closed\_list[new\_i][new\_j]:

if is\_destination(new\_i, new\_j, dest):

cell\_details[new\_i][new\_j].parent\_i = i

cell\_details[new\_i][new\_j].parent\_j = j

print("The destination cell is found")

trace\_path(cell\_details, dest)

found\_dest = True

return

else:

g\_new = cell\_details[i][j].g + 1.0

h\_new = calculate\_h\_value(new\_i, new\_j, dest)

f\_new = g\_new + h\_new

if cell\_details[new\_i][new\_j].f == float('inf') or cell\_details[new\_i][new\_j].f > f\_new:

heapq.heappush(open\_list, (f\_new, new\_i, new\_j))

cell\_details[new\_i][new\_j].f = f\_new

cell\_details[new\_i][new\_j].g = g\_new

cell\_details[new\_i][new\_j].h = h\_new

cell\_details[new\_i][new\_j].parent\_i = i

cell\_details[new\_i][new\_j].parent\_j = j

if not found\_dest:

print("Failed to find the destination cell")

def main():

grid = [

[1, 0, 1, 1, 1, 1, 0, 1, 1, 1],

[1, 1, 1, 0, 1, 1, 1, 0, 1, 1],

[1, 1, 1, 0, 1, 1, 0, 1, 0, 1],

[0, 0, 1, 0, 1, 0, 0, 0, 0, 1],

[1, 1, 1, 0, 1, 1, 1, 0, 1, 0],

[1, 0, 1, 1, 1, 1, 0, 1, 0, 0],

[1, 0, 0, 0, 0, 1, 0, 0, 0, 1],

[1, 0, 1, 1, 1, 1, 0, 1, 1, 1],

[1, 1, 1, 0, 0, 0, 1, 0, 0, 1]

]

src = [8, 0]

dest = [0, 0]

a\_star\_search(grid, src, dest)

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

main()

**5. GENETIC**

import random

def initialize\_population(population\_size, num\_cities):

population = []

for \_ in range(population\_size):

individual = list(range(num\_cities))

random.shuffle(individual)

population.append(individual)

return population

def evaluate\_fitness(individual, distances):

total\_distance = 0

for i in range(len(individual) - 1):

city1, city2 = individual[i], individual[i+1]

total\_distance += distances[city1][city2]

total\_distance += distances[individual[-1]][individual[0]] # Return to the starting city

return total\_distance

def crossover(parent1, parent2):

crossover\_point = random.randint(1, len(parent1) - 1)

child = parent1[:crossover\_point]

for gene in parent2:

if gene not in child:

child.append(gene)

return child

def mutate(individual, mutation\_rate):

if random.random() < mutation\_rate:

idx1, idx2 = random.sample(range(len(individual)), 2)

individual[idx1], individual[idx2] = individual[idx2], individual[idx1]

return individual

def genetic\_algorithm(distances, population\_size, mutation\_rate, generations):

num\_cities = len(distances)

population = initialize\_population(population\_size, num\_cities)

for \_ in range(generations):

new\_population = []

for \_ in range(population\_size):

parent1, parent2 = random.sample(population, 2)

child = crossover(parent1, parent2)

child = mutate(child, mutation\_rate)

new\_population.append(child)

population = new\_population

best\_individual = min(population, key=lambda x: evaluate\_fitness(x, distances))

best\_distance = evaluate\_fitness(best\_individual, distances)

return best\_individual, best\_distance

# Example usage:

distances = [

[0, 10, 15, 20],

[10, 0, 35, 25],

[15, 35, 0, 30],

[20, 25, 30, 0]

]

population\_size = 100

mutation\_rate = 0.01

generations = 1000

best\_solution, best\_distance = genetic\_algorithm(distances, population\_size, mutation\_rate, generations)

print("Best solution:", best\_solution)

print("Best distance:", best\_distance)

**5.SIMULATED**

import random

import math

def generate\_initial\_solution(num\_cities):

return list(range(num\_cities))

def evaluate\_cost(solution, distances):

total\_distance = 0

for i in range(len(solution) - 1):

city1, city2 = solution[i], solution[i+1]

total\_distance += distances[city1][city2]

total\_distance += distances[solution[-1]][solution[0]] # Return to the starting city

return total\_distance

def generate\_neighbor\_solution(solution):

neighbor\_solution = solution[:]

idx1, idx2 = random.sample(range(len(solution)), 2)

neighbor\_solution[idx1], neighbor\_solution[idx2] = neighbor\_solution[idx2], neighbor\_solution[idx1]

return neighbor\_solution

def acceptance\_probability(new\_cost, old\_cost, temperature):

if new\_cost < old\_cost:

return 1.0

return math.exp((old\_cost - new\_cost) / temperature)

def cool(temperature, cooling\_rate):

return temperature \* cooling\_rate

def simulated\_annealing(distances, initial\_temperature, cooling\_rate, max\_iterations):

num\_cities = len(distances)

current\_solution = generate\_initial\_solution(num\_cities)

current\_cost = evaluate\_cost(current\_solution, distances)

best\_solution = current\_solution[:]

best\_cost = current\_cost

temperature = initial\_temperature

for \_ in range(max\_iterations):

new\_solution = generate\_neighbor\_solution(current\_solution)

new\_cost = evaluate\_cost(new\_solution, distances)

if new\_cost < current\_cost or random.random() < acceptance\_probability(new\_cost, current\_cost, temperature):

current\_solution = new\_solution

current\_cost = new\_cost

if current\_cost < best\_cost:

best\_solution = current\_solution[:]

best\_cost = current\_cost

temperature = cool(temperature, cooling\_rate)

return best\_solution, best\_cost

# Example usage:

distances = [

[0, 10, 15, 20],

[10, 0, 35, 25],

[15, 35, 0, 30],

[20, 25, 30, 0]

]

initial\_temperature = 1000

cooling\_rate = 0.95

max\_iterations = 1000

best\_solution, best\_distance = simulated\_annealing(distances, initial\_temperature, cooling\_rate, max\_iterations)

print("Best solution:", best\_solution)

print("Best distance:", best\_distance)

**6.COLORING MAPPING**

def map\_coloring(map\_data, colors):

def is\_valid(region, color, coloring):

for neighbor in map\_data[region]:

if neighbor in coloring and coloring[neighbor] == color:

return False

return True

def backtrack(region\_index, coloring):

if region\_index == len(map\_data):

return coloring

region = list(map\_data.keys())[region\_index]

for color in colors:

if is\_valid(region, color, coloring):

coloring[region] = color

result = backtrack(region\_index + 1, coloring)

if result is not None:

return result

coloring.pop(region)

return None

return backtrack(0, {})

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

# Example usage:

# Define the map data (regions and their neighbors)

map\_data = {

"WA": ["NT", "SA"],

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

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

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

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

"V": ["SA", "NSW"]

}

# Define available colors

colors = ["red", "green", "blue"]

# Solve the Map Coloring Problem

solution = map\_coloring(map\_data, colors)

if solution:

print("Map Coloring Solution:")

for region, color in solution.items():

print(f"{region}: {color}")

else:

print("No solution found")

**7.TIC TAC TOE**

import math

def print\_board(board):

for row in board:

print(" | ".join(row))

print("-" \* 5)

def is\_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\_full(board):

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

def get\_empty\_cells(board):

return [(i, j) for i in range(3) for j in range(3) if board[i][j] == " "]

def minimax(board, depth, maximizing):

if is\_winner(board, "X"):

return 1

elif is\_winner(board, "O"):

return -1

elif is\_full(board):

return 0

if maximizing:

max\_eval = -math.inf

for i, j in get\_empty\_cells(board):

board[i][j] = "X"

eval = minimax(board, depth+1, False)

board[i][j] = " "

max\_eval = max(max\_eval, eval)

return max\_eval

else:

min\_eval = math.inf

for i, j in get\_empty\_cells(board):

board[i][j] = "O"

eval = minimax(board, depth+1, True)

board[i][j] = " "

min\_eval = min(min\_eval, eval)

return min\_eval

def get\_best\_move(board):

best\_eval = -math.inf

best\_move = None

for i, j in get\_empty\_cells(board):

board[i][j] = "X"

eval = minimax(board, 0, False)

board[i][j] = " "

if eval > best\_eval:

best\_eval = eval

best\_move = (i, j)

return best\_move

def main():

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

current\_player = "X"

while True:

print\_board(board)

if current\_player == "X":

i, j = map(int, input("Enter row and column (0-2) for your move: ").split())

if board[i][j] != " ":

print("Cell already occupied. Try again.")

continue

board[i][j] = current\_player

else:

print("AI is thinking...")

i, j = get\_best\_move(board)

board[i][j] = current\_player

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

print\_board(board)

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

break

if is\_full(board):

print\_board(board)

print("It's a draw!")

break

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

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

main()

**8.KINSHIP**

% Define relationships

parent(john, lisa).

parent(john, kate).

parent(mary, lisa).

parent(mary, kate).

parent(lisa, anna).

parent(lisa, jack).

% Define rules for other relationships

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

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

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

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

% Define genders

male(john).

male(jack).

female(mary).

female(lisa).

female(kate).

female(anna).

% Query examples

% Is John the father of Anna?

% ?- father(john, anna).

% This will return false, as John is not the father of Anna.

% Who are the children of Mary?

% ?- child(X, mary).

% This will return X = lisa; X = kate.

% Who are the siblings of Lisa?

% ?- sibling(X, lisa).

% This will return X = kate; X = jack.

% Who are the parents of Anna?

% ?- parent(X, anna).

% This will return X = lisa.

**9.K-MEANS**

import numpy as np

class KMeans:

def \_init\_(self, n\_clusters, max\_iter=300):

self.n\_clusters = n\_clusters

self.max\_iter = max\_iter

def fit(self, X):

self.centroids = X[np.random.choice(X.shape[0], self.n\_clusters, replace=False)]

for \_ in range(self.max\_iter):

# Assign each data point to the nearest centroid

labels = self.\_assign\_clusters(X)

# Update centroids based on the mean of data points assigned to each cluster

new\_centroids = self.\_update\_centroids(X, labels)

# Check for convergence

if np.allclose(new\_centroids, self.centroids):

break

self.centroids = new\_centroids

return self

def \_assign\_clusters(self, X):

distances = np.linalg.norm(X[:, np.newaxis] - self.centroids, axis=2)

return np.argmin(distances, axis=1)

def \_update\_centroids(self, X, labels):

new\_centroids = np.empty\_like(self.centroids)

for i in range(self.n\_clusters):

new\_centroids[i] = np.mean(X[labels == i], axis=0)

return new\_centroids

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

# Example usage:

# Generate some random data points

np.random.seed(42)

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

# Number of clusters

k = 3

# Instantiate and fit KMeans model

kmeans = KMeans(n\_clusters=k)

kmeans.fit(X)

# Get cluster centroids and labels

centroids = kmeans.centroids

labels = kmeans.\_assign\_clusters(X)

print("Cluster Centroids:")

print(centroids)

print("\nCluster Labels:")

print(labels)

**10. DECISION MAKING**

% Define decision tree rules

play\_tennis(outlook, temperature, humidity, wind, yes) :-

outlook(sunny),

temperature(high),

humidity(normal),

wind(weak).

play\_tennis(outlook, temperature, humidity, wind, yes) :-

outlook(sunny),

temperature(high),

humidity(high),

wind(strong).

play\_tennis(outlook, temperature, humidity, wind, no) :-

outlook(overcast).

play\_tennis(outlook, temperature, humidity, wind, yes) :-

outlook(rainy),

wind(weak).

play\_tennis(outlook, temperature, humidity, wind, no) :-

outlook(rainy),

wind(strong).

% Define facts about weather conditions

outlook(sunny).

outlook(overcast).

outlook(rainy).

temperature(high).

temperature(normal).

humidity(high).

humidity(normal).

wind(weak).

wind(strong).