Importing Libraries:

python

import random

import math

These lines import the random and math modules, providing functions for generating random numbers and mathematical operations.

Reading City Names:

python

def read\_city\_names(file\_path):

with open(file\_path, 'r') as file:

return [line.strip() for line in file if not line.startswith("#")]

The read\_city\_names function reads a file containing city names and returns a list of names. It ignores lines starting with '#' (comments).

Reading Distances:

python

def read\_distances(file\_path):

distances = []

with open(file\_path, 'r') as file:

for line in file:

if not line.startswith("#"):

distances.append(list(map(int, line.strip().split())))

return distances

The read\_distances function reads a file containing distance data between cities and returns a 2D list of distances.

Individual Class:

python

class Individual:

def \_\_init\_\_(self, genes):

self.genes = genes

self.result = 0

This class represents an individual in the population. An individual has a set of genes representing the order of cities to visit and a result representing the total distance of the route.

find\_result Method:

python

def find\_result(self, distances):

for i in range(len(self.genes) - 1):

a = self.genes[i]

b = self.genes[i + 1]

self.result += distances[a][b]

The find\_result method calculates the total distance of the route for an individual based on the provided distance matrix.

mutate Function:

python

def mutate(individual):

rand\_gene1 = random.randint(0, len(individual.genes) - 1)

rand\_gene2 = random.randint(0, len(individual.genes) - 1)

individual.genes[rand\_gene1], individual.genes[rand\_gene2] = individual.genes[rand\_gene2], individual.genes[rand\_gene1]

The mutate function randomly swaps two genes in an individual's route to introduce variation.

cross Function:

python

def cross(p1, p2, next\_gen):

stopper = random.randint(0, len(p1.genes) - 1)

c1\_genes = p1.genes[:stopper] + [g for g in p2.genes if g not in p1.genes[:stopper]]

c2\_genes = p2.genes[:stopper] + [g for g in p1.genes if g not in p2.genes[:stopper]]

```

The `cross` function performs crossover between two parents (`p1` and `p2`) to generate two children (`c1` and `c2`).

reproduce Function:

python

def reproduce(population, next\_gen):

init\_size = len(population)

while len(population) > init\_size // 2:

p1 = min(population, key=lambda x: x.result)

population.remove(p1)

p2 = min(population, key=lambda x: x.result)

population.remove(p2)

cross(p1, p2, next\_gen)

The reproduce function creates the next generation through selection, crossover, and mutation.

init\_next\_gen Function:

python

def init\_next\_gen(population, next\_gen):

population.extend(next\_gen)

next\_gen.clear()

The init\_next\_gen function initializes the next generation by replacing the current population with the new one.

print\_individual Function:

python

def print\_individual(individual, city\_names):

print("The shortest path is:")

path = " -> ".join([city\_names[city] for city in individual.genes])

print(path)

reversed\_path = " -> ".join([city\_names[city] for city in reversed(individual.genes)])

print("\nReversed path is:")

print(reversed\_path)

print("\nThe total distance is:")

print(individual.result)

print("\n|{:<20}|{:<20}|".format(city\_names[individual.genes[0]], city\_names[individual.genes[-1]]))

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

start\_city = city\_names[individual.genes[i]]

end\_city = city\_names[individual.genes[i + 1]]

print("|{:<20}|{:<20}|".format(start\_city, end\_city))

print("|{:<20}|{:<20}|".format(city\_names[individual.genes[-1]], city\_names[individual.genes[0]]))

print("|{:<20}|{:<20}|".format(individual.result, individual.result))

The print\_individual function prints the details of the current best individual, including the path, reversed path, total distance, and a table of cities and distances.

main Function:

python

def main(N, max\_iterations):

global distances

city\_names = read\_city\_names("uk12\_name.txt")

distances = read\_distances("uk12\_dist.txt")

initial\_city = city\_names[0]

population = [Individual(random.sample(range(N), N)) for \_ in range(10)]

next\_gen = []

iter = 0

best = None

while iter <= max\_iterations:

if iter == 1 or iter == 8 or iter == 100 or iter == 1000 or iter == 5000 or iter == max\_iterations:

print("Iteration", iter, end=": ")

best = min(population, key=lambda x: x.result)

print\_individual(best, city\_names)

reproduce(population, next\_gen)

init\_next\_gen(population, next\_gen)

iter += 1

return best.result

The main function initializes the algorithm, runs the evolution process, and returns the result of the shortest path.

Program Execution:

python

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

N = 12

max\_iterations = 5000

result = main(N, max\_iterations)

print("Shortest path:", result)

This block ensures that the program runs when executed and prints the shortest path found by the algorithm.