#include <iostream>

#include <vector>

#include <algorithm>

#include <ctime>

#include <cstdlib>

#include<cmath>

using namespace std;

#define N 5 // Number of cities

#define POP\_SIZE 10 // Population size

#define GENERATIONS 1000 // Number of generations

#define MUTATION\_RATE 0.1 // Mutation rate

// Structure to represent a city

struct City {

int x, y;

};

// Function to calculate the Euclidean distance between two cities

double distance(const City& city1, const City& city2) {

int dx = city1.x - city2.x;

int dy = city1.y - city2.y;

return sqrt(dx \* dx + dy \* dy);

}

// Function to generate random permutation of cities

vector<int> generateRandomPermutation() {

vector<int> permutation(N);

for (int i = 0; i < N; ++i)

permutation[i] = i;

random\_shuffle(permutation.begin(), permutation.end());

return permutation;

}

// Function to calculate the fitness (total distance) of an individual (path)

double calculateFitness(const vector<int>& path, const vector<City>& cities) {

double totalDistance = 0.0;

for (int i = 0; i < N - 1; ++i)

totalDistance += distance(cities[path[i]], cities[path[i + 1]]);

totalDistance += distance(cities[path[N - 1]], cities[path[0]]); // Return to the starting city

return totalDistance;

}

// Function to perform crossover (order crossover)

vector<int> crossover(const vector<int>& parent1, const vector<int>& parent2) {

int pos1 = rand() % N;

int pos2 = rand() % N;

if (pos1 > pos2)

swap(pos1, pos2);

vector<int> child(N, -1);

for (int i = pos1; i <= pos2; ++i)

child[i] = parent1[i];

int j = 0;

for (int i = 0; i < N; ++i) {

if (j == pos1) j = pos2 + 1;

if (find(child.begin(), child.end(), parent2[i]) == child.end()) {

child[j++] = parent2[i];

}

}

return child;

}

// Function to perform mutation (swap mutation)

void mutate(vector<int>& path) {

for (int i = 0; i < N; ++i) {

if ((rand() / (double)RAND\_MAX) < MUTATION\_RATE) {

int j = rand() % N;

swap(path[i], path[j]);

}

}

}

int main() {

srand(time(0));

// Define the cities

vector<City> cities = {{0, 0}, {1, 1}, {2, 3}, {5, 2}, {3, 4}};

// Initialize population

vector<vector<int>> population(POP\_SIZE);

for (int i = 0; i < POP\_SIZE; ++i)

population[i] = generateRandomPermutation();

// Main loop

for (int generation = 0; generation < GENERATIONS; ++generation) {

// Calculate fitness for each individual

vector<pair<double, int>> fitnessIndex;

for (int i = 0; i < POP\_SIZE; ++i)

fitnessIndex.push\_back({calculateFitness(population[i], cities), i});

sort(fitnessIndex.begin(), fitnessIndex.end());

// Create new population through selection, crossover, and mutation

vector<vector<int>> newPopulation(POP\_SIZE);

for (int i = 0; i < POP\_SIZE; ++i) {

int parent1 = fitnessIndex[rand() % (POP\_SIZE / 2)].second;

int parent2 = fitnessIndex[rand() % (POP\_SIZE / 2)].second;

newPopulation[i] = crossover(population[parent1], population[parent2]);

mutate(newPopulation[i]);

}

population = newPopulation;

}

// Print the best solution (shortest path)

double minDistance = calculateFitness(population[0], cities);

vector<int> bestPath = population[0];

for (int i = 1; i < POP\_SIZE; ++i) {

double dist = calculateFitness(population[i], cities);

if (dist < minDistance) {

minDistance = dist;

bestPath = population[i];

}

}

cout << "Shortest path distance: " << minDistance << endl;

cout << "Best path: ";

for (int i : bestPath)

cout << i << " ";

cout << endl;

return 0;

}