KAUNO TECHNOLOGIJOS UNIVERSITETAS

FAKULTETAS

Algoritmų sudarymas ir analizė  
Inžinerinio projekto ataskaita

Atliko:

Rokas Marcinkevičius

Priėmė:

Doc. Dr. Makackas Dalius

KAUNAS 2023

TURINYS

[1. Užduotis 3](#_Toc167878291)

[2. Pirma dalis – localiai geriausias 4](#_Toc167878292)

[2.1. Workflow 4](#_Toc167878293)

[2.2. Programos kodas ir asimptotinis sudėtingumas: 4](#_Toc167878294)

[2.3. Rezultatai 6](#_Toc167878295)

[3. Antra dalis – optimalus 8](#_Toc167878296)

[3.1. Workflow 8](#_Toc167878297)

[3.2. Programos kodas ir asimptotinis sudėtingumas: 8](#_Toc167878298)

[3.3. Grafikas: 10](#_Toc167878299)

[3.4. Rezultatai: 10](#_Toc167878300)

[4. Trečia dalis – genetinis algoritmas 11](#_Toc167878301)

[4.1. Workflow 11](#_Toc167878302)

[4.2. Asimptotinis sudėtingumas 11](#_Toc167878303)

[4.3. Rezultatai 12](#_Toc167878304)

[4.4. Rekomenduojami parametrai: 13](#_Toc167878305)

[4.5. Programinis kodas: 13](#_Toc167878306)

# Užduotis

**Programos užduotis:**

1. **Dalis.**  (~3 balai). Realizuoti programą, kuri individualiai problemai pateiktų optimalų sprendinį. Nustatyti, prie kokios duomenų apimties sprendinį pavyksta rasti jei programos vykdymo laikas negali būti ilgesnis nei 10 sek.
2. **Dalis.**  (~2 balai). Realizuoti programą, kuri individualios problemos rezultatą pateiktų priimant „lokaliai  geriausią“ sprendinį (pvz. esant keliems pasirinkimams keliauti į skirtingas viršūnes, visuomet renkamasi: pigiausias ar trumpiausias ar  ... kelias)
3. **Dalis.**  (~5 balai). Realizuoti programą, kuri pateiktų sprendinį taikant Genetinio Optimizavimo metodą. Programa pateikti rezultatą turi ne ilgiau, nei per 60 sek.

**Bendra:**

* Darbo ataskaitoje turi būti pateikta:
  + apskaičiuotas asimptotinis programos vykdymo laiko sudėtingumas (esant rekursiniams metodams, turi būti suformuojama rekurentinė lygtis);
  + realizuotų programų (sudaryto algoritmo konkrečiam uždaviniui) abstraktus aprašas („pseudo“ kodas ar „workflow“ diagrama);
  + skirtingų metodų rezultatų analizė uždavinio gerumo ir vykdymo laiko prasme esant kelioms skirtingoms pradinėms sąlygoms.
  + pateikti rekomenduojamus genetinio optimizavimo parametrus užduoties sprendimui (populiacijos dydis, elito kiekis, ...). Pateikti analizės rezultatus, kaip buvo parinkti rekomendaciniai parametrai, t. y. analizės rezultatuose turi pasimatyti, jog per maksimalų programos vykdymo laiką, tikėtina, jog tikslo funkcija įgaus didžiausią reikšmę.
  + marštutai, sudaryti analizuojant atksirus atvejus atvaizduojami grafiškai.
* Kelionės pradžią turi būti galima nurodyti bet kurią vietą ar miestą (priklausomai nuo individualios sąlygos) iš pateiktų duomenų.
* Programavimo kalba: bet kuri, išskyrus Python.

**Individuali užduotis:**

Faile places\_data.xlsx pateikta informacija apie miestus (įskaitant jų gerumo įvertį) ir kelius tarp jų (kelionės laikas ir kaina)  (2, 3 lentelė).

Tikslas: kaip galima geresnio (vertinama pagal aplankytų miestų gerumo įverčius) maršruto sudarymas kai:

* kelionės pradžios ir pabaigos vieta sutampa (su grįžimu atgal);
* bendras kelionės laikas negali viršyti 48 val.;
* tą patį miestą galima aplankyti kelis kartus, tačiau jo gerumo įvertis sumuojamas tik vieną kartą.

# Pirma dalis – localiai geriausias

## Workflow

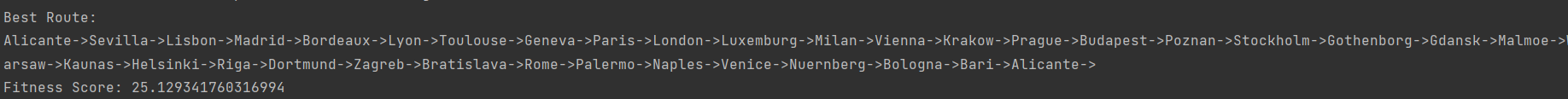
Paveikslėlis, kuriame yra tekstas, ekrano kopija, diagrama, eskizas

Automatiškai sugeneruotas aprašymas

## Programos kodas ir asimptotinis sudėtingumas:

|  |  |  |
| --- | --- | --- |
| Kodo eilutė | Kaina | Kiekis |
| private static void FindBestRouteLocal(string currentCity, int currentTime, double currentScore) |  |  |
| { |  |  |
| if (currentTime > maxTime) | C1 | 1 |
| { |  |  |
| return; | C2 | 1 |
| } |  |  |
| if (currentCity != "" && currentCity == visitedCities[0] && currentTime != 0) | C3 | 1 |
| { |  |  |
| if (currentScore > bestScore) | C4 | 1 |
| { |  |  |
| bestScore = currentScore; | C5 | 1 |
| bestRoute = new List<string>(visitedCities); | C6 | 1 |
| bestTimep = currentTime; | C7 | 1 |
| } |  |  |
| return; | C8 | 1 |
| } |  |  |
| string bestCityTo = ""; | C9 | 1 |
| int bestTime = 1; | C10 | 1 |
| double bestScorel = 0; | C11 | 1 |
| string backCityTo = ""; | C12 | 1 |
| int backTime = 1; | C13 | 1 |
| foreach (var route in routes) | C14 | n |
| { |  |  |
| if (route.StartCity == currentCity && (!visitedCities.Contains(route.EndCity) || visitedCities[0] == route.EndCity)) | N\*N | n\*m |
| { |  |  |
| if (GetCityScore(route.EndCity) / route.Time > bestScorel / bestTime) | C16 | n\*n |
| { |  |  |
| bestCityTo = route.EndCity; | C17 | n |
| bestTime = route.Time; | C18 | n |
| bestScorel = GetCityScore(route.EndCity); | C19 | n |
| } |  |  |
| if (route.EndCity == visitedCities[0]) | C21 | n |
| { |  |  |
| backCityTo = route.EndCity; | C22 | n |
| backTime = route.Time; | C23 | n |
| } |  |  |
| } |  |  |
| } |  |  |
| if (bestCityTo != "") | C24 | 1 |
| { |  |  |
| visitedCities.Add(bestCityTo); | C25 | 1 |
| FindBestRouteLocal(bestCityTo, currentTime + bestTime, currentScore + GetCityScore(bestCityTo)); | T( T+C, CS + N) | 1 |
| visitedCities.RemoveAt(visitedCities.Count - 1); | N-i-1 | 1 |
| } |  |  |
| if (backCityTo != "") | C27 | 1 |
| { |  |  |
| visitedCities.Add(backCityTo); | C28 | 1 |
| FindBestRouteLocal(backCityTo, currentTime + backTime, currentScore + GetCityScore(backCityTo)); | T(T+C, CS + N) | 1 |
| visitedCities.RemoveAt(visitedCities.Count - 1); | N-i-1 | 1 |
| } |  |  |
| } |  |  |

## Rezultatai

Geriausias kelias su duotais duomenimis, kai pradinis miestas Alicante:  


A picture containing line, diagram, text

Description automatically generated

# Antra dalis – optimalus

## Workflow

Paveikslėlis, kuriame yra tekstas, diagrama, ekrano kopija, dizainas

Automatiškai sugeneruotas aprašymas

## Programos kodas ir asimptotinis sudėtingumas:

Rekurentinė lygtis

|  |  |  |
| --- | --- | --- |
| Kodo eilutė | Kaina | Kiekis |
| private static void FindBestRouteBranchAndCut(string currentCity, int currentTime, double currentScore) |  |  |
| { |  |  |
| if (currentTime > maxTime && stopwatch.Elapsed.TotalSeconds > 10) | c1 | 1 |
| { |  |  |
| return; | c2 | 1 |
| } |  |  |
|  |  |  |
| double maxPossibleAdditionalScore = GetMaxPossibleScoreForRemainingTime(maxTime - currentTime); | T(n\*n) | 1 |
| if (currentScore + maxPossibleAdditionalScore <= bestScore) | c3 | 1 |
| { |  |  |
| return; | c4 | 1 |
| } |  |  |
|  |  |  |
| if (currentCity != "" && currentCity == visitedCities[0] && currentTime != 0) | c5 | 1 |
| { |  |  |
| if (currentScore > bestScore) | c6 | 1 |
| { |  |  |
| bestScore = currentScore; | c7 | 1 |
| bestRoute = new List<string>(visitedCities); | c8 | 1 |
| bestTimep = currentTime; | c9 | 1 |
| } |  |  |
| return; | c10 | 1 |
| } |  |  |
|  |  |  |
| foreach (var route in routes) | n | 1 |
| { |  |  |
| if (route.StartCity == currentCity && (!visitedCities.Contains(route.EndCity) || visitedCities[0] == route.EndCity)) | c11 | n |
| { |  |  |
| visitedCities.Add(route.EndCity); | c12 | n |
| FindBestRouteBranchAndCut(route.EndCity, currentTime + route.Time, currentScore + GetCityScore(route.EndCity)); | T(t+c,cs+n) | n |
| visitedCities.RemoveAt(visitedCities.Count - 1); | m-i-1 | n |
| } |  |  |
| } |  |  |
| } |  |  |

, kur

n – maršrutų kiekis, t – laikas, cs - įvertis

|  |  |  |
| --- | --- | --- |
| Kodo eilutė | Kaina | Kiekis |
| private static double GetMaxPossibleScoreForRemainingTime(int remainingTime) |  |  |
| { |  |  |
| double maxScore = 0; | c1 | 1 |
| foreach (var route in routes) | n | 1 |
| { |  |  |
| if (route.Time <= remainingTime) | c2 | n |
| { |  |  |
| maxScore += GetCityScore(route.EndCity); | n | n |
| remainingTime -= route.Time; | c3 | n |
| } |  |  |
| } | c4 | n |
| return maxScore; |  |  |
| } |  |  |

T(n) = n\*n, kur n – maršrutų kiekis

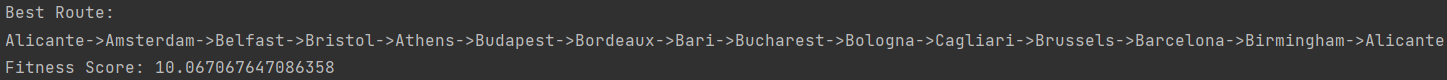
## Grafikas:

A picture containing line, text, diagram, plot

Description automatically generated

## Rezultatai:

Geriausias kelias su 20 pirmųjų miestų iš duotų duomenų, kai pradinis miestas Alicante:



A picture containing line, diagram, sketch, drawing

Description automatically generated

# Trečia dalis – genetinis algoritmas

## Workflow

Paveikslėlis, kuriame yra tekstas, diagrama, ekrano kopija, dizainas

Automatiškai sugeneruotas aprašymas

## Asimptotinis sudėtingumas

GetCityByName metodas yra O(n), kur n yra miestai.

GetShuffledCityNames metodas yra O(m), kur m yra keliai.

TournamentSelection metodas yra O(P/10), kur P yra populiacija.

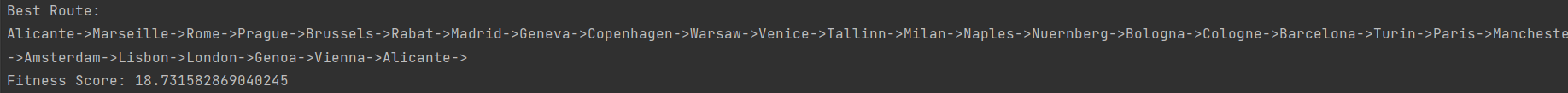
Crossover metodas yra O(c), kur c yra vaiko miestai.

Mutate metodas yra O(m), kur m yra keliai.

GeneticAlgorithm metodas yra O(G\*(P+m)), kur G yra generacija, P yra populiacija, o m yra keliai.

## Rezultatai

Geriausias kelias su duotais duomenimis, kai pradinis miestas yra Alicante:



A picture containing line, diagram

Description automatically generated

## Rekomenduojami parametrai:

Atlikus skirtingus bandymus:  
A picture containing text, menu, screenshot, number

Description automatically generated

Rekomenduojami parametrai yra:  
Population = 300

Generation = 500

Mutation = 0.1

## Programinis kodas:

using System;

using System.Drawing;

using System.ComponentModel;

using System.Collections.Generic;

using System.Linq;

using System.Diagnostics;

using SkiaSharp;

using OfficeOpenXml;

using Microsoft.Office.Interop.Excel;

namespace IP\_1;

class Program

{

private static List<City> cities; // List of available cities

private static List<Route> routes; // List of available routes

private static List<string> visitedCities; // List of visited cities in the current route

private static double bestScore; // Best score found so far

private static Stopwatch stopwatch; // Stopwatch to track execution time

private static List<string> bestRoute; // Best route found so far

private const int maxTime = 172800; // Maximum travel time allowed

private static int bestTimep;

static void Main(string[] args)

{

int printRow = 2;

//for (int row = 25; row <= 25; row++)

var bitmap = new SKBitmap(4434086 / 1000, 4405311 / 1000);

InOut inOut = new InOut();

cities = inOut.ReadCities(104);

var points = new List<SKPoint>();

List<string> citiesNames = new List<string>();

foreach (var city in cities)

{

points.Add(new SKPoint((float)(city.X + 1017080) / 1000, (float)(city.Y - 4031810) / 1000));

citiesNames.Add(city.Name);

}

routes = inOut.ReadRoutes(citiesNames);

string startingCity = "Alicante";

// Start the recursive method to find the best route

visitedCities = new List<string>();

visitedCities.Add(startingCity);

bestScore = 0;

bestRoute = new List<string>();

bestTimep = 0;

stopwatch = new Stopwatch();

// Start the algorithm and track the execution time

stopwatch.Start();

FindBestRouteLocal(startingCity, 0, GetCityScore(startingCity));

//FindBestRouteOptimal(startingCity, 0, GetCityScore(startingCity));

stopwatch.Stop();

// Print the best route and its score score

Console.WriteLine("Best Route:");

foreach (var city in bestRoute)

{

Console.Write(city + "->");

}

Console.WriteLine();

Console.WriteLine("Total city Score: " + bestScore);

Console.WriteLine("Execution Time: " + stopwatch.Elapsed.TotalSeconds + " seconds");

InOut.PaintCanvas(bitmap, points, citiesNames, bestRoute);

// Printing to result file

string filePath = "Results.xlsx";

InOut.PrintResults(filePath, printRow, bestScore, stopwatch);

printRow++;

}

private static void FindBestRouteBranchAndCut(string currentCity, int currentTime, double currentScore)

{

if (currentTime > maxTime && stopwatch.Elapsed.TotalSeconds > 10)

{

return;

}

double maxPossibleAdditionalScore = GetMaxPossibleScoreForRemainingTime(maxTime - currentTime);

if (currentScore + maxPossibleAdditionalScore <= bestScore)

{

return;

}

if (currentCity != "" && currentCity == visitedCities[0] && currentTime != 0)

{

if (currentScore > bestScore)

{

bestScore = currentScore;

bestRoute = new List<string>(visitedCities);

bestTimep = currentTime;

}

return;

}

foreach (var route in routes)

{

if (route.StartCity == currentCity && (!visitedCities.Contains(route.EndCity) || visitedCities[0] == route.EndCity))

{

visitedCities.Add(route.EndCity);

FindBestRouteBranchAndCut(route.EndCity, currentTime + route.Time, currentScore + GetCityScore(route.EndCity));

visitedCities.RemoveAt(visitedCities.Count - 1);

}

}

}

private static double GetMaxPossibleScoreForRemainingTime(int remainingTime)

{

double maxScore = 0;

foreach (var route in routes)

{

if (route.Time <= remainingTime)

{

maxScore += GetCityScore(route.EndCity);

remainingTime -= route.Time;

}

}

return maxScore;

}

private static void FindBestRouteLocal(string currentCity, int currentTime, double currentScore)

{

if (currentTime > maxTime)

{

return;

}

if (currentCity != "" && currentCity == visitedCities[0] && currentTime != 0)

{

if (currentScore > bestScore)

{

bestScore = currentScore;

bestRoute = new List<string>(visitedCities);

bestTimep = currentTime;

}

return;

}

string bestCityTo = "";

int bestTime = 1;

double bestScorel = 0;

string backCityTo = "";

int backTime = 1;

foreach (var route in routes)

{

if (route.StartCity == currentCity && (!visitedCities.Contains(route.EndCity) || visitedCities[0] == route.EndCity))

{

if (GetCityScore(route.EndCity) / route.Time > bestScorel / bestTime)

{

bestCityTo = route.EndCity;

bestTime = route.Time;

bestScorel = GetCityScore(route.EndCity);

}

if (route.EndCity == visitedCities[0])

{

backCityTo = route.EndCity;

backTime = route.Time;

}

}

}

if (bestCityTo != "")

{

visitedCities.Add(bestCityTo);

FindBestRouteLocal(bestCityTo, currentTime + bestTime, currentScore + GetCityScore(bestCityTo));

visitedCities.RemoveAt(visitedCities.Count - 1);

}

if (backCityTo != "")

{

visitedCities.Add(backCityTo);

FindBestRouteLocal(backCityTo, currentTime + backTime, currentScore + GetCityScore(backCityTo));

visitedCities.RemoveAt(visitedCities.Count - 1);

}

}

private static double GetCityScore(string cityName)

{

foreach (var city in cities)

{

if (city.Name == cityName)

{

return city.Score;

}

}

return 0;

}

}

namespace IP\_1

{

public class InOut

{

private ExcelPackage \_package;

private ExcelWorksheet \_worksheet;

public InOut()

{

ExcelPackage.LicenseContext = OfficeOpenXml.LicenseContext.NonCommercial;

\_package = new ExcelPackage(new System.IO.FileInfo(@"./places\_data.xlsx"));

\_worksheet = \_package.Workbook.Worksheets[0];

}

public List<City> ReadCities(int maxrow)

{

List<City> cities = new List<City>();

for (int row = 6; row <= maxrow; row++) ///104

{

string name = \_worksheet.Cells[row, 8].Value?.ToString();

double score = Convert.ToDouble(\_worksheet.Cells[row, 9].Value?.ToString());

double x = Convert.ToDouble(\_worksheet.Cells[row, 10].Value?.ToString());

double y = Convert.ToDouble(\_worksheet.Cells[row, 11].Value?.ToString());

cities.Add(new City(name, score, x, y));

}

return cities;

}

public List<Route> ReadRoutes(List<string> cities)

{

List<Route> routes = new List<Route>();

for (int row = 6; row <= 2275; row++) ///2275 ///2282

{

string cityFrom = \_worksheet.Cells[row, 13].Value?.ToString();

string cityTo = \_worksheet.Cells[row, 14].Value?.ToString();

int time = Convert.ToInt32(\_worksheet.Cells[row, 15].Value?.ToString());

double price = Convert.ToDouble(\_worksheet.Cells[row, 16].Value?.ToString());

if(cities.Contains(cityFrom) && cities.Contains(cityTo))

{

routes.Add(new Route(cityFrom, cityTo, time, price));

}

}

\_package.Dispose();

return routes;

}

public static void PaintCanvas(SKBitmap bitmap, List<SKPoint> points, List<string> cities, List<string> bestRoute)

{

using (var canvas = new SKCanvas(bitmap))

{

canvas.Clear(SKColors.White);

using var paint = new SKPaint

{

Color = SKColors.Red,

IsAntialias = true,

StrokeWidth = 6,

StrokeCap = SKStrokeCap.Round,

Style = SKPaintStyle.Stroke

};

foreach (var point in points)

{

canvas.DrawCircle(point.X, point.Y, 5, paint);

}

using var path = new SKPath();

path.MoveTo(points[0].X, points[0].Y);

for (int i = 1; i < points.Count; i++)

{

path.LineTo(points[i].X, points[i].Y);

}

using var paint2 = new SKPaint

{

Color = SKColors.Blue,

IsAntialias = true,

StrokeWidth = 3,

StrokeCap = SKStrokeCap.Round,

Style = SKPaintStyle.Stroke

};

for (int i = 0; i < bestRoute.Count - 1; i++)

{

canvas.DrawLine(points[cities.IndexOf(bestRoute[i])], points[cities.IndexOf(bestRoute[i + 1])], paint2);

}

}

using (var stream = new SKFileWStream("image.png"))

{

bitmap.Encode(stream, SKEncodedImageFormat.Png, 10000);

}

}

public static void PrintResults(string file, int printRow, double bestScore, Stopwatch stopwatch)

{

using (var package = new ExcelPackage(new FileInfo(file)))

{

// Get the existing worksheet or create a new one

ExcelWorksheet worksheet = package.Workbook.Worksheets.FirstOrDefault();

if (worksheet == null)

worksheet = package.Workbook.Worksheets.Add("Results");

worksheet.Cells[printRow, 1].Value = bestScore;

worksheet.Cells[printRow, 3].Value = stopwatch.Elapsed.TotalSeconds;

package.Save();

}

}

}

}

public class Route

{

public string StartCity { get; set; }

public string EndCity { get; set; }

public int Time { get; set; }

public double Price { get; set; }

public Route(string startCity, string endCity, int time, double price)

{

StartCity = startCity;

EndCity = endCity;

Time = time;

Price = price;

}

}

public class City

{

public string Name { get; set; }

public double Score { get; set; }

public double X { get; set; }

public double Y { get; set; }

public City(string name, double score, double x, double y)

{

Name = name;

Score = score;

X = x;

Y = y;

}

}

using SkiaSharp;

using System.Diagnostics;

namespace IP\_3;

public class Program

{

private static List<City> cities; // List of available cities

private static List<Route> routes; // List of available routes

private static Random random; // Random number generator

private static Stopwatch stopwatch; // Stopwatch to track execution time

private static int populationSize = 300; // Size of the population

private static int maxGenerations = 300; // Maximum number of generations

private static double mutationRate = 0.1; // Rate of mutation

private const int maxTime = 48 \* 3600; // Maximum travel time allowed

private const int maxExecutionTime = 60; // Maximum execution time in seconds

public static void Main(string[] args)

{

var bitmap = new SKBitmap(4434086 / 1000, 4405311 / 1000);

InOut inOut = new InOut();

cities = inOut.ReadCities();

var points = new List<SKPoint>();

List<string> citiesNames = new List<string>();

foreach (var city in cities)

{

points.Add(new SKPoint((float)(city.X + 1017080) / 1000, (float)(city.Y - 4031810) / 1000));

citiesNames.Add(city.Name);

}

routes = inOut.ReadRoutes(citiesNames);

string startingCity = "Alicante";

random = new Random();

stopwatch = new Stopwatch();

stopwatch.Start();

TestRoute bestRoute = GeneticAlgorithm(startingCity);

stopwatch.Stop();

Console.WriteLine("Best Route:");

foreach (var city in bestRoute.Cities)

{

Console.Write(city + "->");

}

Console.WriteLine();

Console.WriteLine("City Score: " + bestRoute.Score);

Console.WriteLine("Execution Time: " + stopwatch.Elapsed.TotalSeconds + " seconds");

InOut.PaintCanvas(bitmap, points, bestRoute, citiesNames);

}

private static TestRoute TournamentSelection(List<TestRoute> population)

{

List<TestRoute> tournamentParticapants = new List<TestRoute>();

int tournamentSize = populationSize / 10;

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

{

int randomIndex = random.Next(population.Count);

tournamentParticapants.Add(population[randomIndex]);

}

TestRoute bestIndividual = tournamentParticapants.OrderBy(r => r.Score).ThenByDescending(r => r.TotalTravelTime).ThenByDescending(r => r.Cities.Count).Last();

return bestIndividual;

}

static TestRoute GetShuffledCityNames(string startingCity)

{

TestRoute route = new TestRoute(new List<string>(), 0, 0);

route.Cities.Add(startingCity);

route.Score = GetCityByName(startingCity).Score;

string currentCity = startingCity;

while (true)

{

List<Route> availableRoutes = routes.FindAll(r => r.CityFrom == currentCity);

List<Route> validRoutes = availableRoutes.FindAll(r => r.Time + route.TotalTravelTime < maxTime);

if (validRoutes.Count == 0)

break;

Route randomRoute = validRoutes[random.Next(validRoutes.Count)];

currentCity = randomRoute.CityTo;

route.TotalTravelTime += randomRoute.Time;

if (!route.Cities.Contains(currentCity))

{

route.Score += GetCityByName(currentCity).Score;

}

route.Cities.Add(currentCity);

if (currentCity == startingCity)

break;

}

if (route.Cities[0] == route.Cities[route.Cities.Count - 1])

{

return route;

}

return null;

}

static City GetCityByName(string name)

{

return cities.FirstOrDefault(c => c.Name == name);

}

private static TestRoute GeneticAlgorithm(string startingCity)

{

List<TestRoute> population = InitializePopulation(startingCity);

TestRoute bestRoute = new TestRoute(new List<string>(), 0, 0);

int generation = 0;

while (generation < maxGenerations && stopwatch.Elapsed.TotalSeconds < maxExecutionTime)

{

TestRoute elite = GetBestIndividual(population);

if (elite.Score > bestRoute.Score || (elite.Score == bestRoute.Score && elite.TotalTravelTime < bestRoute.TotalTravelTime) || (elite.Score == bestRoute.Score && elite.TotalTravelTime == bestRoute.TotalTravelTime && elite.Cities.Count < bestRoute.Cities.Count))

{

bestRoute = elite.Copy();

Console.WriteLine("New best route found with score: {0}", bestRoute.Score);

}

Console.WriteLine("Generation: {0}", generation + 1);

List<TestRoute> nextGeneration = new List<TestRoute>();

nextGeneration.Add(elite);

while (nextGeneration.Count < populationSize)

{

TestRoute parent1 = TournamentSelection(population);

TestRoute offspring = null;

int tries = 0;

while (offspring == null && tries < 5)

{

TestRoute parent2 = TournamentSelection(population);

if (parent1 == parent2)

{

offspring = null;

tries++;

continue;

}

offspring = Crossover(parent1, parent2);

tries++;

}

nextGeneration.Add(offspring == null ? parent1 : offspring);

}

foreach (var individual in nextGeneration)

{

Mutate(individual);

}

population = nextGeneration;

generation++;

}

return bestRoute;

}

static List<TestRoute> InitializePopulation(string startingCity)

{

List<TestRoute> population = new List<TestRoute>();

while (population.Count < populationSize)

{

TestRoute route = GetShuffledCityNames(startingCity);

if (route != null)

{

population.Add(route);

}

}

return population;

}

private static double CalculateScore(List<string> cities2)

{

double totalScore = 0;

int time = 0;

List<string> visitedCities = new List<string>();

string prevCity = null;

foreach (string city in cities2)

{

if (!visitedCities.Contains(city))

{

City currentCity = GetCityByName(city);

totalScore += currentCity.Score;

visitedCities.Add(city);

}

if (prevCity != null)

{

var route = routes.FirstOrDefault(r => (r.CityFrom == prevCity && r.CityTo == city));

if (route != default)

{

time += route.Time;

}

else

{

return 0;

}

}

prevCity = city;

}

if (time > 48 \* 3600)

{

return 0;

}

return totalScore;

}

private static TestRoute Crossover(TestRoute parent1, TestRoute parent2)

{

bool flag = false;

List<string> childCities = new List<string>();

int length = Math.Min(parent1.Cities.Count, parent2.Cities.Count);

childCities.Add(parent1.Cities[0]);

for (int i = 1; i < length - 1; i++)

{

if (!flag && parent1.Cities[i] != parent2.Cities[i])

{

flag = true;

}

childCities.Add(flag ? parent2.Cities[i] : parent1.Cities[i]);

}

for (int i = length - 1; i < parent2.Cities.Count; i++)

{

childCities.Add(parent2.Cities[i]);

}

double score = CalculateScore(childCities);

int totalTravelTime = CalculateTotalTravelTime(childCities);

if (flag && totalTravelTime <= maxTime)

{

return new TestRoute(childCities, score, totalTravelTime);

}

else

{

return null;

}

}

static int CalculateTotalTravelTime(List<string> cities2)

{

double score = 0;

HashSet<string> visitedCities = new HashSet<string>();

int totalTravelTime = 0;

for (int i = 0; i < cities2.Count - 1; i++)

{

string cityA = cities2[i];

string cityB = cities2[i + 1];

visitedCities.Add(cityA);

var route = routes.FirstOrDefault(r => (r.CityFrom == cityA && r.CityTo == cityB) || (r.CityFrom == cityB && r.CityTo == cityA));

if (route != default)

{

City cityAObject = GetCityByName(cityA);

if (cityAObject != default)

{

score += cityAObject.Score;

totalTravelTime += route.Time;

}

else

{

Console.WriteLine($"City object not found: {cityA}");

}

}

}

string lastCity = cities2.Last();

visitedCities.Add(lastCity);

var returnTrip = routes.FirstOrDefault(r => (r.CityFrom == lastCity && r.CityTo == cities2[0]) || (r.CityFrom == cities2[0] && r.CityTo == lastCity));

if (returnTrip != default)

{

totalTravelTime += returnTrip.Time;

}

if (totalTravelTime > maxTime)

{

score = 0;

}

return totalTravelTime;

}

private static void Mutate(TestRoute route)

{

for (int i = 1; i < route.Cities.Count - 1; i++)

{

if (random.NextDouble() < mutationRate)

{

if (random.NextDouble() >= 0.005)

{

List<Route> validRoutesFrom = routes.Where(r => r.CityFrom == route.Cities[i - 1]).ToList();

List<Route> validRoutesTo = routes.Where(r => r.CityTo == route.Cities[i + 1]).ToList();

List<string> validCities = (from fromRoute in validRoutesFrom

join toRoute in validRoutesTo on fromRoute.CityTo equals toRoute.CityFrom

select fromRoute.CityTo).Distinct().ToList();

if (validCities.Count == 0)

{

continue;

}

string city = ShuffleList(validCities).First();

route.Cities[i] = city;

}

else

{

List<Route> validRoutesFrom = routes.Where(r => r.CityFrom == route.Cities[i]).ToList();

List<Route> validRoutesTo = routes.Where(r => r.CityTo == route.Cities[i + 1]).ToList();

List<string> validCities = (from fromRoute in validRoutesFrom

join toRoute in validRoutesTo on fromRoute.CityTo equals toRoute.CityFrom

select fromRoute.CityTo).Distinct().ToList();

if (validCities.Count == 0)

{

continue;

}

string city = ShuffleList(validCities).First();

route.Cities.Insert(i, city);

continue;

}

}

}

route.Score = CalculateScore(route.Cities);

}

private static TestRoute GetBestIndividual(List<TestRoute> population)

{

TestRoute bestIndividual = population.OrderBy(r => r.Score).ThenByDescending(r => r.TotalTravelTime).ThenByDescending(r => r.Cities.Count).Last();

return bestIndividual;

}

private static List<T> ShuffleList<T>(List<T> list)

{

int n = list.Count;

while (n > 1)

{

n--;

int k = random.Next(n + 1);

T value = list[k];

list[k] = list[n];

list[n] = value;

}

return list;

}

}

namespace IP\_3;

public class TestRoute

{

public List<string> Cities { get; set; }

public double Score { get; set; }

public int TotalTravelTime { get; set; }

public TestRoute(List<string> cities, double score, int totalTravelTime)

{

Cities = cities;

Score = score;

TotalTravelTime = totalTravelTime;

}

public TestRoute Copy()

{

return new TestRoute(this.Cities, this.Score, this.TotalTravelTime);

}

}

using OfficeOpenXml;

using SkiaSharp;

namespace IP\_3

{

public class InOut

{

private ExcelPackage \_package;

private ExcelWorksheet \_worksheet;

public InOut()

{

ExcelPackage.LicenseContext = OfficeOpenXml.LicenseContext.NonCommercial;

\_package = new ExcelPackage(new System.IO.FileInfo(@"./places\_data.xlsx"));

\_worksheet = \_package.Workbook.Worksheets[0];

}

public List<City> ReadCities()

{

List<City> cities = new List<City>();

for (int row = 6; row <= 104; row++) ///104 ////109

{

string name = \_worksheet.Cells[row, 8].Value?.ToString();

double fitness = Convert.ToDouble(\_worksheet.Cells[row, 9].Value?.ToString());

double x = Convert.ToDouble(\_worksheet.Cells[row, 10].Value?.ToString());

double y = Convert.ToDouble(\_worksheet.Cells[row, 11].Value?.ToString());

cities.Add(new City(name, fitness, x, y));

}

return cities;

}

public List<Route> ReadRoutes(List<string> cities)

{

List<Route> routes = new List<Route>();

for (int row = 6; row <= 2275; row++) ///2275 ///2282

{

string cityFrom = \_worksheet.Cells[row, 13].Value?.ToString();

string cityTo = \_worksheet.Cells[row, 14].Value?.ToString();

int time = Convert.ToInt32(\_worksheet.Cells[row, 15].Value?.ToString());

double price = Convert.ToDouble(\_worksheet.Cells[row, 16].Value?.ToString());

if (cities.Contains(cityFrom) && cities.Contains(cityTo))

{

routes.Add(new Route(cityFrom, cityTo, time, price));

}

}

\_package.Dispose();

return routes;

}

public static void PaintCanvas(SKBitmap bitmap, List<SKPoint> points, TestRoute bestRoute, List<string> cities)

{

using (var canvas = new SKCanvas(bitmap))

{

canvas.Clear(SKColors.White);

using var paint = new SKPaint

{

Color = SKColors.Red,

IsAntialias = true,

StrokeWidth = 6,

StrokeCap = SKStrokeCap.Round,

Style = SKPaintStyle.Stroke

};

foreach (var point in points)

{

canvas.DrawCircle(point.X, point.Y, 5, paint);

}

using var path = new SKPath();

path.MoveTo(points[0].X, points[0].Y);

for (int i = 1; i < points.Count; i++)

{

path.LineTo(points[i].X, points[i].Y);

}

using var paint2 = new SKPaint

{

Color = SKColors.Blue,

IsAntialias = true,

StrokeWidth = 3,

StrokeCap = SKStrokeCap.Round,

Style = SKPaintStyle.Stroke

};

for (int i = 0; i < bestRoute.Cities.Count - 1; i++)

{

canvas.DrawLine(points[cities.IndexOf(bestRoute.Cities[i])],

points[cities.IndexOf(bestRoute.Cities[i + 1])], paint2);

}

}

using (var stream = new SKFileWStream("image.png"))

{

bitmap.Encode(stream, SKEncodedImageFormat.Png, 10000);

}

}

}

}