**KAUNO TECHNOLOGIJOS UNIVERSITETAS**

**INFORMATIKOS FAKULTETAS**

**Modulio P170B400 „Algoritmų sudarymas ir analizė“**

Laboratorinio darbo aprašas (ataskaita)

**Inžinerinis projektas**

**Dėstytojas**

lekt. MAKACKAS Dalius

**Studentas**

Matas Palujanskas IFF-1/8

**KAUNAS, 2023**

**TURINYS**

[1. UŽDUOTIS 3](#_Toc135860134)

[2. PASIRINKTI KŪRIMO ĮRANKIAI 4](#_Toc135860135)

[3. PIRMA DALIS. 5](#_Toc135860136)

[3.1. Programos metodai 5](#_Toc135860137)

[3.2. Rezultatai ir gautas grafas 8](#_Toc135860138)

[3.3. Analizė 9](#_Toc135860139)

[3.4. Greitaveika 12](#_Toc135860140)

[4. ANTRA DALIS. 13](#_Toc135860141)

[4.1. Programinis kodas 14](#_Toc135860142)

[4.2. Rezultatai ir grafas 15](#_Toc135860143)

[4.3. Analizė 17](#_Toc135860144)

[4.4. Greitaveika 18](#_Toc135860145)

[5. TREČIA DALIS 19](#_Toc135860146)

[5.1. Programos metodai: 20](#_Toc135860147)

[5.2. Analizė 22](#_Toc135860148)

[5.3. Rezultatai ir grafas 24](#_Toc135860149)

# 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.

Užduotis:

Paveikslėlis, kuriame yra tekstas, Šriftas, ekrano kopija

Automatiškai sugeneruotas aprašymas

# PASIRINKTI KŪRIMO ĮRANKIAI

Inžineriniam projektui atlikti pasirinkau C# programavimo kalbą kadangi su ja turiu daugiausiai patirties bei ji turi biblioteką, kurios pagalba galima paprastai ir patogiai konvertuoti „Microsoft Excel“ dokumente esančią informaciją. Šią biblioteką naudojau nuskaitant duomenis iš duotojo pradinio failo.

Paveikslėlis, kuriame yra tekstas, ekrano kopija, programinė įranga, Operacinė sistema

Automatiškai sugeneruotas aprašymas

pav. 1 Duomenų nuskaitymas naudojant excel biblioteką

# PIRMA DALIS.

* 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.

Šis uždavinys taip pat atspindi klasikinį problemai "travelling salesman problem" (keliaujančio pardavėjo problema) koncepciją, kurioje siekiama rasti pigiausią maršrutą tarp kelių vietų, kurias reikia aplankyti tik vieną kartą.

Pirmoji dalis prašo sukurti programinį kodą, kuris individualiai išspręstų šią problemą naudojant

optimaliausią sprendimą. Naudojant dinaminį programavimą.

**Metodo idėja**

1. Klasėje "Place" apibrėžiamas vietos objektas, turintį savybes "Name" (pavadinimas), "Id" (identifikatorius), "X" ir "Y" koordinates.
2. Tuomet "LoadPlacesFromExcel" yra nuskaitomi vietų duomenis iš Excel failo į sąrašą.
3. Funkcija "FindOptimalRoute" randa optimalų maršrutą tarp vietų sąrašo, naudodama TSP sprendimo algoritmą. Ji generuoja visas galimas vietų permutacijas ir ieško mažiausios kainos.
4. Funkcija "GeneratePermutations" generuoja visas galimus vietų perstumdymus(variantus), naudodama rekursiją.
5. Funkcija "CalculateDistance" skaičiuoja atstumą tarp dviejų vietų, naudodama tiesioginę atstumo formulę.
6. Funkcija "GenerateTestData" generuoja sintetinius testinius duomenis, priklausomai nuo duomenų apimties.
7. Funkcija "GenerateRandomCoordinate" generuoja atsitiktines koordinates tam tikrame intervale.

## Programos metodai

Metodas apskaičiuojantis atstumą tarp vietovių:

|  |
| --- |
| public static double CalculateDistance(Place place1, Place place2)  {  double dx = place2.X - place1.X;  double dy = place2.Y - place1.Y;  double distance = Math.Sqrt(dx \* dx + dy \* dy);  return distance;  } |

Metodas rekursyviai generuoja visas vietų permutacijas(elementų sukeitimas/pertvarkymas):

|  |
| --- |
| public static void GeneratePermutationsRecursive(List<Place> places, int n, List<List<Place>> permutations)  {  if (n == 1)  {  List<Place> permutation = new List<Place>(places);  permutations.Add(permutation);  }  else  {  for (int i = 0; i < n - 1; i++)  {  GeneratePermutationsRecursive(places, n - 1, permutations);  if (n % 2 == 0)  {  Swap(places, i, n - 1);  }  else  {  Swap(places, 0, n - 1);  }  }  GeneratePermutationsRecursive(places, n - 1, permutations);  }  } |

|  |
| --- |
| public static List<List<Place>> GeneratePermutations(List<Place> places)  {  List<List<Place>> permutations = new List<List<Place>>();  GeneratePermutationsRecursive(places, places.Count, permutations);  return permutations;  } |

Metodas sukeičia du sąrašo elementus, nurodydamas jų indeksus:

|  |
| --- |
| public static void Swap(List<Place> places, int i, int j)  {  Place temp = places[i];  places[i] = places[j];  places[j] = temp;  } |

Metodas išsprendžia keliaujančio pirklio uždavinį(TSP), taikydamas grubios jėgos metodą. Jis sukuria visas galimas vietų permutacijas ir apskaičiuoja bendrą atstumą kiekvienai permutacijai. Metodas stebi mažiausią atstumą turinčią permutaciją, kuri yra optimalus maršrutas:

|  |
| --- |
| public static List<Place> FindOptimalRoute(List<Place> places)  {  List<Place> optimalRoute = new List<Place>();  int numPlaces = places.Count;  // Generate all possible permutations of the places  List<List<Place>> permutations = GeneratePermutations(places);  double minDistance = double.MaxValue;  // Iterate through each permutation to find the optimal route  foreach (List<Place> permutation in permutations)  {  double totalDistance = 0;  // Calculate the total distance for the current permutation  for (int i = 0; i < numPlaces - 1; i++)  {  totalDistance += CalculateDistance(permutation[i], permutation[i + 1]);  }  // Check if the total distance is smaller than the current minimum distance  if (totalDistance < minDistance)  {  minDistance = totalDistance;  optimalRoute = permutation;  }  }  // Complete the route by returning to the starting place  optimalRoute.Add(optimalRoute[0]);  return optimalRoute;  } |

Metodas priima optimalų maršrutą vaizduojančių objektų Place sąrašą ir, naudodamas ScottPlot biblioteką, sukuria brėžinį. Jis iš kiekvienos vietos (Place) ištraukia X ir Y koordinates ir nubraižo jas kaip sklaidos diagramą.

|  |
| --- |
| public static void PlotOptimalRoute(List<Place> optimalRoute)  {  int numPlaces = optimalRoute.Count;  double[] xCoords = new double[numPlaces];  double[] yCoords = new double[numPlaces];  for (int i = 0; i < numPlaces; i++)  {  xCoords[i] = optimalRoute[i].X;  yCoords[i] = optimalRoute[i].Y;  }  // Create a new ScottPlot figure  var plt = new ScottPlot.Plot(800, 600);  // Plot the optimal route as a line plot  plt.PlotScatter(xCoords, yCoords, markerSize: 5, lineWidth: 1);  // Customize the plot  plt.Title("Optimal Route");  plt.XLabel("X Coordinate");  plt.YLabel("Y Coordinate");  // Save the plot to a file  plt.SaveFig("optimal\_route.png");  } |

## Rezultatai ir gautas grafas

Paveikslėlis, kuriame yra tekstas, ekrano kopija, Šriftas, juodas ir baltas

Automatiškai sugeneruotas aprašymas

pav. 2 Rezultatai

**Grafas:**

Paveikslėlis, kuriame yra diagrama, linija, Grafikas, tekstas

Automatiškai sugeneruotas aprašymas

pav. 3 Grafas

## Analizė

**Pagalbiniai metodai:**

|  |  |  |
| --- | --- | --- |
| public static double CalculateDistance(Place place1, Place place2) | Laikas | Kiekis |
| { |  |  |
| double dx = place2.X - place1.X; | c1 | 1 |
| double dy = place2.Y - place1.Y; | c2 | 1 |
| double distance = Math.Sqrt(dx \* dx + dy \* dy); | c3 | 1 |
| return distance; | c4 | 1 |
| } |  |  |

T(n) = c1+c2+c3 = Ѳ(1).

|  |  |  |
| --- | --- | --- |
| public static void GeneratePermutationsRecursive(List<Place> places, int n, List<List<Place>> permutations) | Laikas | Kiekis |
| { |  |  |
| if (n == 1) | c1 | 1 |
| { |  |  |
| List<Place> permutation = new List<Place>(places); | c2 | 1 |
| permutations.Add(permutation); | c3 | 1 |
| } |  |  |
| else | c4 | n |
| { |  |  |
| for (int i = 0; i < n - 1; i++) | c5 | n |
| { |  |  |
| GeneratePermutationsRecursive(places, n - 1, permutations); | c6 | n-1 |
|  |  |  |
| if (n % 2 == 0) | c7 | n |
| { |  |  |
| Swap(places, i, n - 1); | c8 | n |
| } |  |  |
| else |  |  |
| { |  |  |
| Swap(places, 0, n - 1); | c9 | n |
| } |  |  |
| } |  |  |
|  |  |  |
| GeneratePermutationsRecursive(places, n - 1, permutations); | c10 | 1 |
| } |  |  |
| } |  |  |

T(n) = c1+c2+c3+c6\*(n-1)+(c4+c5+c7+c8+c9)\*n+c10 = Ѳ(n).

|  |  |  |
| --- | --- | --- |
| public static List<List<Place>> GeneratePermutations(List<Place> places) | Laikas | Kiekis |
| { |  |  |
| List<List<Place>> permutations = new List<List<Place>>(); | c1 | 1 |
| GeneratePermutationsRecursive(places, places.Count, permutations); |  |  |
| return permutations; | c3 | 1 |
| } |  |  |

T(n) = c1+c2+c3 = Ѳ(1).

|  |  |  |
| --- | --- | --- |
| public static void Swap(List<Place> places, int i, int j) | Laikas | Kiekis |
| { |  |  |
| Place temp = places[i]; | c1 | 1 |
| places[i] = places[j]; | c2 | 1 |
| places[j] = temp; | c3 | 1 |
| } |  |  |

T(n) = c1+c2+c3 = Ѳ(1).

**Pagrindinio skaičiavimo metodo analizė:**

|  |  |  |
| --- | --- | --- |
| public static List<Place> FindOptimalRoute(List<Place> places) | Laikas | Kiekis |
| { |  |  |
| List<Place> optimalRoute = new List<Place>(); | c1 | 1 |
| int numPlaces = places.Count; | c2 | 1 |
|  |  |  |
| // Generate all possible permutations of the places |  |  |
| List<List<Place>> permutations = GeneratePermutations(places); | c3 | 1 |
|  |  |  |
| double minDistance = double.MaxValue; | c4 | 1 |
|  |  |  |
| // Iterate through each permutation to find the optimal route |  |  |
| foreach (List<Place> permutation in permutations) | c5 | n |
| { |  |  |
| double totalDistance = 0; | c6 | n |
|  |  |  |
| // Calculate the total distance for the current permutation |  |  |
| for (int i = 0; i < numPlaces - 1; i++) | c7 | n^2 |
| { |  |  |
| totalDistance += CalculateDistance(permutation[i], permutation[i + 1]); | c8 | n^2 |
| } |  |  |
|  |  |  |
| // Check if the total distance is smaller than the current minimum distance |  |  |
| if (totalDistance < minDistance) | c9 | n^2 |
| { |  |  |
| minDistance = totalDistance; | c10 | n^2 |
| optimalRoute = permutation; | c11 | n^2 |
| } |  |  |
| } |  |  |
|  |  |  |
| // Complete the route by returning to the starting place |  |  |
| optimalRoute.Add(optimalRoute[0]); | c12 | 1 |
|  |  |  |
| return optimalRoute; | c13 | 1 |
| } |  |  |

T(n) = c1+c2+c3+c4+(c5+c6)\*n+(c7+c8+c9+c10+c11)\*n^2+c12+c13 = Ѳ(n^2).

## Greitaveika

Atlikime greitaveikos testus ir atvaizduokime gautus rezultatus diagramoje, kurioje matysis programos vykdymo laiko priklausomybė nuo duomenų kiekio:

Paveikslėlis, kuriame yra tekstas, linija, Grafikas, skaičius

Automatiškai sugeneruotas aprašymas

pav. 4 Laiko priklausomybė nuo duomenų kiekio

**Rezultatai**: Iš rezultatų galime matyti, kad programos vykdymo laikas yra didesnis nei 10 sekundžių kai duomenų kiekis yra didesnis nei 9.

# ANTRA DALIS.

* **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)

Antrame etape norima sukurti programą, kuri naudoja "lokaliai geriausią" sprendimą, taikant godų (angl. greedy) algoritmą. Šis algoritmas veikia taip, kad iš dabartinio mazgo ieškoma kaimyninių mazgų, į kuriuos galima pereiti, ir pasirenkamas tas, kuris yra artimiausias. Šis procesas kartojamas, kol bus aplankyti visi mazgai. Godus algoritmas yra efektyvesnis nei dinaminis programavimas arba rekursinis sprendimas, tačiau jis ne visuomet užtikrina optimalų rezultatą.

**Metodo idėja**

1. Pradžioje jis atsitiktinai pasirenka pradžios vietą iš sąrašo (places) ir prideda ją prie optimalaus maršruto sąrašo (optimalRoute). Taip pat pašalina šią vietą iš sąrašo (places).
2. Tada jis ieško artimiausios dar nepalankintos vietos, naudodamas "CalculateCost" metodą, kuris apskaičiuoja atstumą tarp dviejų vietų.
3. Jis pasirenka vietą su mažiausiomis sąnaudomis ir ją prideda prie optimalaus maršruto sąrašo (optimalRoute). Taip pat pašalina šią vietą iš sąrašo (places).
4. Šis procesas kartojamas, kol optimalus maršruto sąrašas (optimalRoute) pasiekia maksimalų leistiną vietų skaičių arba kol nebeliko nepalankintų vietų sąraše (places).
5. Galiausiai pridedama pradinė vieta iš optimalaus maršruto sąrašo į galą, užbaigiant ciklą.

## Programinis kodas

|  |
| --- |
| public static List<Place> FindOptimalRoute2(List<Place> places)  {  List<Place> optimalRoute = new List<Place>();  int numPlaces = places.Count;  // Randomly select the starting place  Random random = new Random();  int startIndex = random.Next(numPlaces);  Place currentPlace = places[startIndex];  places.RemoveAt(startIndex);  optimalRoute.Add(currentPlace); // Add the first place with the place number  // Find the nearest unvisited place with the cheapest path and add it to the optimal route  while (optimalRoute.Count < 150 && places.Count > 0)  {  double minCost = double.MaxValue;  int minIndex = -1;  for (int i = 0; i < places.Count; i++)  {  double cost = CalculateCost(currentPlace, places[i]);  if (cost < minCost)  {  minCost = cost;  minIndex = i;  }  }  currentPlace = places[minIndex];  places.RemoveAt(minIndex);  optimalRoute.Add(currentPlace); // Add the place with the place number  }  // Complete the route by returning to the starting place  optimalRoute.Add(optimalRoute[0]);  return optimalRoute;  } |

## Rezultatai ir grafas

Paveikslėlis, kuriame yra tekstas, ekrano kopija, Šriftas, juodas ir baltas

Automatiškai sugeneruotas aprašymas

pav. 5 Suformuoto kelio pradžia

**...**

Paveikslėlis, kuriame yra tekstas, ekrano kopija, Šriftas, juodas ir baltas

Automatiškai sugeneruotas aprašymas

pav. 6 Suformuoto kelio pabaiga

|  |  |
| --- | --- |
|  |  |

**Rezultatai**: buvo aplankyta 150 lankytinų vietų ir grįžta į pradinę vietą.

**Gautas grafas**

Paveikslėlis, kuriame yra tekstas, diagrama, linija, Grafikas

Automatiškai sugeneruotas aprašymas

## Analizė

|  |  |  |
| --- | --- | --- |
| public static List<Place> FindOptimalRoute2(List<Place> places) | Laikas | Kiekis |
| { |  |  |
| List<Place> optimalRoute = new List<Place>(); | c1 | 1 |
| int numPlaces = places.Count; | c2 | 1 |
|  |  |  |
| // Randomly select the starting place |  |  |
| Random random = new Random(); | c3 | 1 |
| int startIndex = random.Next(numPlaces); | c4 | 1 |
| Place currentPlace = places[startIndex]; | c5 | 1 |
| places.RemoveAt(startIndex); | c6 | 1 |
| optimalRoute.Add(currentPlace); // Add the first place with the place number | c7 | 1 |
|  |  |  |
| // Find the nearest unvisited place with the cheapest path and add it to the optimal route |  |  |
| while (optimalRoute.Count < 150 && places.Count > 0) | c8 | n=optimalRoute.Count |
| { |  |  |
| double minCost = double.MaxValue; | c9 | n |
| int minIndex = -1; | c10 | n |
|  |  |  |
| for (int i = 0; i < places.Count; i++) | c11 | n^2 |
| { |  |  |
| double cost = CalculateCost(currentPlace, places[i]); | c12 | n^2 |
| if (cost < minCost) | c13 | n^2 |
| { |  |  |
| minCost = cost; | c14 | n^2 |
| minIndex = i; | c15 | n^2 |
| } |  |  |
| } |  |  |
|  |  |  |
| currentPlace = places[minIndex]; | c16 | n |
| places.RemoveAt(minIndex); | c17 | n |
| optimalRoute.Add(currentPlace); // Add the place with the place number | c18 | n |
| } |  |  |
|  |  |  |
| // Complete the route by returning to the starting place |  |  |
| optimalRoute.Add(optimalRoute[0]); | c19 | 1 |
|  |  |  |
| return optimalRoute; | c20 | 1 |
| } |  |  |

T(n) = c1+c2+c3+c4+c5+c6+c7+(c8+c9+c10+c16+c17+c18)\*n+

+(c11+c12+c13+c14+c15)\*n^2+c19+c20 = Ѳ(n^2).

|  |  |  |
| --- | --- | --- |
| public static double CalculateCost(Place place1, Place place2) | Laikas | Kiekis |
| { |  |  |
| double dx = place2.X - place1.X; | c1 | 1 |
| double dy = place2.Y - place1.Y; | c2 | 1 |
| double distance = Math.Sqrt(dx \* dx + dy \* dy); | c3 | 1 |

T(n) = c1+c2+c3 = Ѳ(1)

## Greitaveika

Atlikime greitaveikos testus ir įsitikinkime, kad apskaičiuotas asimptotinis sudėtingumas yra teisingas:

Paveikslėlis, kuriame yra tekstas, ekrano kopija, programinė įranga, Šriftas

Automatiškai sugeneruotas aprašymas

pav. 7 Testinių duomenų generavimas

Paveikslėlis, kuriame yra tekstas, Šriftas, ekrano kopija

Automatiškai sugeneruotas aprašymas

pav. 8 Greitaveika 1

Paveikslėlis, kuriame yra tekstas, Šriftas, ekrano kopija, juodas

Automatiškai sugeneruotas aprašymas

pav. 9 Greitaveika 2

Paveikslėlis, kuriame yra tekstas, Šriftas, ekrano kopija, tipografija

Automatiškai sugeneruotas aprašymas

pav. 10 Greitaveika 3

Paveikslėlis, kuriame yra tekstas, Šriftas, ekrano kopija, juodas

Automatiškai sugeneruotas aprašymas

pav. 11 Greitaveika 4

**Rezultatai**: Iš gautų rezultatų galime teigti, kad apskaičiuotas asimptotinis sudėtingumas atitinka kodo greitaveiką.

# TREČIA DALIS

* **Dalis**. (~5 balai). Realizuoti programą, kuri pateiktų sprendinį taikant Genetinio Optimizavimo metodą. Programa pateikti rezultatą turi ne ilgiau, nei per 60 sek.

**Algoritmo įgyvendinimas:**

1. Programa pradžioje nuskaito duomenis iš "places\_data.xlsx" failo ir užkrauna vietų informaciją į sąrašą. Iš visų vietų sąrašo pasirenkama pirmų 150 vietų sąrašo dalis, kurios bus naudojamos optimizavimui.
2. Sukuriamas genetinis optimizavimo objektas, perduodant pasirinktas vietas, populiacijos dydį (150 individų), mutacijos ir kryžminimo rodiklius, bei maksimalų kartų skaičių (1000). Naudojant genetinio optimizavimo metodą (FindCheaperRoute), ieškomas pigesnis maršrutas tarp pasirinktų vietų. Tai vyksta per nurodytą maksimalų kartų skaičių (maxGenerations).
3. Kiekvienoje kartotėje (generacijoje) vyksta individų vertinimas, t.y. skaičiuojama kiekvieno individų tinkamumo (fitness) reikšmė, remiantis kainomis (atstumais) tarp vietų. Individai yra atrinkti reprodukcijai pagal jų tinkamumo reikšmes (Selection). Čia naudojamas turnyrinio atrankos metodas, kai iš atsitiktinai pasirinktų individų parenkamas geriausias.
4. Po atrankos vyksta kryžminimas (Crossover) ir mutacija (Mutation) kiekvienam palikuoniui (offspring), kad būtų sukurta nauja populiacija (offspringPopulation). Sukeitus senąją populiaciją su nauja palikuonių populiacija, eina nauja kartotė (generacija).
5. Baigus nustatytą maksimalų kartų skaičių, randamas geriausias individas (maršrutas) iš galutinės populiacijos, naudojant CalculateTotalDistance funkciją.
6. Pateikiamas geriausias maršrutas, išvedami kiekvienos vietos pavadinimas, ID ir kaina (atstumas), apskaičiuojama bendra kelionės kaina (totalCost).Taip pat yra sukuriamas grafinis vaizdas naudojant ScottPlot biblioteką. Pavaizduojamos vietų koordinatės grafike, o pažymimosios taškai vaizduoja vietų padėtis.

Programa baigia vykdymą ir išveda praleistą laiką (execution time) sekundėmis.

## Programos metodai:

Šis metodas sukuria pradinę populiaciją, kurioje yra sąrašai su atsitiktinai išmaišytomis vietomis. Metodas kiekvieną kartą sukuria naują maršrutą, kurio elementai yra išmaišyti. Visi šie maršrutai yra pridėti į populiacijos sąrašą:

|  |
| --- |
| private List<List<Place>> InitializePopulation()  {  List<List<Place>> population = new List<List<Place>>();  for (int i = 0; i < populationSize; i++)  {  List<Place> route = new List<Place>(places);  Shuffle(route); // Shuffle the route randomly  population.Add(route);  }  return population;  } |

Šis metodas įvertina kiekvieno individo tinkamumo reikšmę populiacijoje. Tai daroma skaičiuojant kiekvieno maršruto visą atstumą ir apskaičiuojant tinkamumo reikšmę:

|  |
| --- |
| private List<double> EvaluateFitness(List<List<Place>> population)  {  List<double> fitnessScores = new List<double>();  foreach (List<Place> route in population)  {  double totalDistance = CalculateTotalDistance(route);  double fitness = 1 / Math.Sqrt(totalDistance); // Fitness is inversely proportional to the square root of the total distance  fitnessScores.Add(fitness);  }  return fitnessScores;  } |

Individai yra atrinkti reprodukcijai pagal jų tinkamumo reikšmes:

|  |
| --- |
| private List<List<Place>> Selection(List<List<Place>> population, List<double> fitnessScores)  {  List<List<Place>> selectedPopulation = new List<List<Place>>();  for (int i = 0; i < populationSize; i++)  {  List<Place> selectedIndividual = SelectIndividual(population, fitnessScores);  selectedPopulation.Add(selectedIndividual);  }  return selectedPopulation;  } |

Reprodukcijos metodu reprodukcija atliekama kuriant palikuonis kryžminimo ir mutacijos būdu. Kaip įvestis imama dabartinė populiacija ir atitinkami jos tinkamumo balai, ir grąžinama nauja populiacija:

|  |
| --- |
| private List<List<Place>> Reproduction(List<List<Place>> population, List<double> fitnessScores)  {  List<List<Place>> offspringPopulation = new List<List<Place>>();  while (offspringPopulation.Count < populationSize)  {  List<Place> parent1 = SelectIndividual(population, fitnessScores);  List<Place> parent2 = SelectIndividual(population, fitnessScores);  List<Place> offspring = Crossover(parent1, parent2);  // Apply mutation  if (new Random().NextDouble() < mutationRate)  {  Mutate(offspring);  }  offspringPopulation.Add(offspring);  }  return offspringPopulation;  } |

Naudojant metodą FindBestIndividual (rasti geriausią individą (maršrutą) iš populiacijos, remiantis jų tinkamumo balais, surandamas geriausias individas (maršrutas). Jis įveda dabartinę populiaciją ir grąžina geriausią individą (maršrutą), kurio tinkamumo balai yra didžiausi:

|  |
| --- |
| private List<Place> FindBestIndividual(List<List<Place>> population)  {  List<Place> bestIndividual = null;  double bestFitness = 0;  foreach (List <Place> individual in population)  {  double fitness = 1 / CalculateTotalDistance(individual);  if (bestIndividual == null || fitness > bestFitness)  {  bestIndividual = individual;  bestFitness = fitness;  }  }  return bestIndividual;  } |

## Analizė

|  |  |  |
| --- | --- | --- |
| private List<double> EvaluateFitness(List<List<Place>> population) | Laikas | Kiekis |
| { |  |  |
| List<double> fitnessScores = new List<double>(); | c1 | 1 |
|  |  |  |
| foreach (List<Place> route in population) | c2 | n=route |
| { |  |  |
| double totalDistance = CalculateTotalDistance(route); | c3 | n |
| double fitness = 1 / Math.Sqrt(totalDistance); // Fitness is inversely proportional to the square root of the total distance | c4 | n |
| fitnessScores.Add(fitness); | c5 | n |
| } |  |  |
|  |  |  |
| return fitnessScores; | c6 | 1 |
| } |  |  |

T(n) = c1+(c2+c3+c4+c5)\*n+c6 =Ѳ(n)

|  |  |  |
| --- | --- | --- |
| private List<List<Place>> Selection(List<List<Place>> population, List<double> fitnessScores) | Laikas | Kiekis |
| { |  |  |
| List<List<Place>> selectedPopulation = new List<List<Place>>(); | c1 | 1 |
|  |  |  |
| for (int i = 0; i < populationSize; i++) | c2 | n |
| { |  |  |
| List<Place> selectedIndividual = SelectIndividual(population, fitnessScores); | c3 | n |
| selectedPopulation.Add(selectedIndividual); | c4 | n |
| } |  |  |
|  |  |  |
| return selectedPopulation; | c5 | 1 |
| } |  |  |

T(n) = c1+(c2+c3+c4)\*n+c5 =Ѳ(n)

|  |  |  |
| --- | --- | --- |
| private List<List<Place>> Reproduction(List<List<Place>> population, List<double> fitnessScores) | Laikas | Kiekis |
| { |  |  |
| List<List<Place>> offspringPopulation = new List<List<Place>>(); | c1 | 1 |
|  |  |  |
| while (offspringPopulation.Count < populationSize) | c2 | n |
| { |  |  |
| List<Place> parent1 = SelectIndividual(population, fitnessScores); | c3 | n |
| List<Place> parent2 = SelectIndividual(population, fitnessScores); | c4 | n |
|  |  |  |
| List<Place> offspring = Crossover(parent1, parent2); | c5 | n |
|  |  |  |
| // Apply mutation |  |  |
| if (new Random().NextDouble() < mutationRate) | c6 | n |
| { |  |  |
| Mutate(offspring); | c7 | n |
| } |  |  |
|  |  |  |
| offspringPopulation.Add(offspring); | c8 | n |
| } |  |  |
|  |  |  |
| return offspringPopulation; | c9 | 1 |
| } |  |  |

T(n) = c1+(c2+c3+c4+c5+c6+c7+c8)\*n+c9 =Ѳ(n)

|  |  |  |
| --- | --- | --- |
| private List<Place> FindBestIndividual(List<List<Place>> population) | Laikas | Kiekis |
| { |  |  |
| List<Place> bestIndividual = null; | c1 | 1 |
| double bestFitness = 0; | c2 | 1 |
|  |  |  |
| foreach (List <Place> individual in population) | c3 | n=individual |
| { |  |  |
| double fitness = 1 / CalculateTotalDistance(individual); | c4 | n |
|  |  |  |
| if (bestIndividual == null || fitness > bestFitness) | c5 | n |
| { |  |  |
| bestIndividual = individual; | c6 | n |
| bestFitness = fitness; | c7 | n |
| } |  |  |
| } |  |  |
|  |  |  |
| return bestIndividual; | c8 | 1 |
| } |  |  |

T(n) = c1+c2+(c3+c4+c5+c6+c7)\*n+c8 =Ѳ(n)

## Rezultatai ir grafas

Genetinio optimizavimo metodo asimptotinis sudėtingumas priklauso ne tik nuo duomenų kiekio, bet ir nuo įvedamų parametrų – populiacijos dydžio, kryžminimo dažnio, kartų kiekio. Norint gauti kuo optimalesnį sprendinį, reikia eksperimentuoti su metodo parametrais. Rankiniu būdu ištestavęs metodą, pastebėjau, kad optimaliausi rezultatai gauti su tokiais parametrais:

* Duomenų kiekis – 150;
* Populiacijos dydis - 299;
* Mutacijos dažnis – 0.01;
* Kryžminimo dažnis – 0.8;
* Generacijos (kartai) – 1500;

Sugeneruotas geriausias variantas aplankyti 150 vietų per mažiau nei 60s:

Paveikslėlis, kuriame yra tekstas, ekrano kopija, Šriftas, juodas ir baltas

Automatiškai sugeneruotas aprašymas

pav. 12 Rezultatai

Paveikslėlis, kuriame yra linija, diagrama, ekrano kopija, Paralelė

Automatiškai sugeneruotas aprašymas

pav. 13 Gautas grafas

**Išvados**: programos vykdymo laikas nėra didesnis nei 60 sekundžių, programa rado optimaliausią sprendimą su turimais duomenimis.