KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS

Algoritmų sudarymas ir analizė (P175B124)

Inžinerinio projekto ataskaita

Atliko:

IFF-1/2 gr. studentas

Lukas Borinskij

2023 m. gegužės 24 d.

Priėmė:

Lekt. Dalius Makackas

Lekt. Vidmantas Rimavičius

Lekt. Tadas Kraujalis

KAUNAS 2023

TURINYS

1. Inžinerinio projekto 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.

Faile places_data.xlsx pateikta informacija apie lankytinas vietas (1 lentelė). Tikslas: kaip galima pigesnio maršruto sudarymas kai:

- priimama, kad kelionės tarp vietų kaina lygi kvadratinei šakniai iš kelionės atstumo;
- reikia aplankyti visas vietas;
- ta pati vieta negali būti aplankyta daugiau nei vieną kartą (tariama, kad vieta aplankyta, jei ją aplankė bet kuris iš keliautojų);
- kelionės pradžios ir pabaigos vieta sutampa (su grįžimu atgal);
- maršrutas planuojamas 3 keliautojam.

2. Pirma dalis

Pirmos dalies pseudokodas:

```
TSP_Implement(Adj_matrix, s):
    cities = []  // Store all cities apart from the source city
    for i = 0 to V-1:
        if i != s:
            cities.append(i)
    min_distance = INFINITY
    do:
        curr_distance = 0  // Initialize current path distance
        k = s
        for i = 0 to cities.size() - 1:
            curr_distance += Adj_matrix[k][cities[i]]
            k = cities[i]
        curr_distance += Adj_matrix[k][s]  // Add distance from last city back to source
        min_distance = min(min_distance, curr_distance)  // Update minimum distance
    while next_permutation(cities)  // Generate next permutation of cities

    return min_distance
```

Aprašymas:

TSP_Implement metodas yra skirtas spręsti keliaujančio prekeivio problemą (angl. Traveling Salesman Problem, TSP). Pagrindinė idėja yra rasti trumpiausią maršrutą, kuris aplankytų visus miestus tik kartą ir sugrįžtų į pradinį miestą.

Šis metodas naudoja permutacijas, kad sugeneruotų visus galimus maršrutus ir paskaičiuotų kiekvieno maršruto bendrą atstumą. Pradedant nuo pradinio miesto, jis eina per visus likusius miestus ir skaičiuoja atstumą tarp kiekvieno miesto ir jo sekančio miesto maršrute. Galiausiai pridedamas atstumas nuo paskutinio miesto iki pradinio miesto, kad būtų užbaigtas ciklas.

Metodas vykdo permutacijų operaciją, kol visi galimi maršrutai yra išbandyti.

Kodo analizė

	Kaina	Kartai
`public static List<Location> TSP_Implement(double[,] adjMatrix, int start, List<Location> locations, List<Location> original)`		
`{`		
`locations.Insert(0, original[start]);`	c	1
`int V = locations.Count;`	c	1
`var cities = new List<int>();`	c	n-1
`for (int i = 1; i < V; i++)`		
`{`		
`cities.Add(i);`	c	n
`}`		
`double minDistance = double.MaxValue;`	C	1
`List<Location> shortestPath = null;`	c	1
`do`	c	(n-1)!
`{`		
`double currDistance = 0;`	c	(n-1)!
`int k = start;`	c	(n-1)!
`var path = new List<Location> { locations[start] };`	c	(n-1)!
`for (int i = 0; i < cities.Count; i++)`	c	(n-1)!*(n-1)
`{`	c	(n-1)!*n
`currDistance += adjMatrix[k, cities[i]];`	c	(n-1)!*n
`k = cities[i];`	c	(n-1)!*n
`path.Add(locations[cities[i]]);`		
`}`		
`currDistance += adjMatrix[k, start];`	c	(n-1)!
`path.Add(locations[0]); // Add the starting location to the end of the path`	c	(n-1)!
`if (currDistance < minDistance)`	c	(n-1)!
`{`		
`minDistance = currDistance;`	c	(n-1)!
`shortestPath = path;`	c	(n-1)!
`}`		
`} while (NextPermutation(cities));`		
`return shortestPath;`	c	1
`}`		

$$T(n) = n * (n-1)! + C = O * (n!)$$

Bendras šio metodo vykdymo laikas priklauso nuo permutacijų skaičiaus, kuris yra n!, kur n yra miestų skaičius. Tai reiškia, kad laiko sudėtingumas yra O(n!). Per didelis miestų skaičius gali sukelti labai didelę laiko sąnaudą, nes permutacijų skaičius eksponentiškai greitai auga. Todėl TSP yra Nepolynominio sunkumo problema.

Tikrinant programos veikimą, kad jis nevirsšytų 10 sekundžių buvo rasta, jog **duomenų apimtis maksimaliai yra mano kompiuteriui apie n=30.**

3. Antra dalis

Antros dalies pseudokodas:

```
GreedyTSP(adjMatrix, start):
   n = number of locations
   visited = array of size n to track visited locations
   path = empty list to store the final path
   current = start
   path.append(current)
   visited[current] = true

   while path size < n:
      nearestNeighbor = -1
      minDistance = INFINITY

      for i = 0 to n-1:
         if i != current and !visited[i] and adjMatrix[current][i] < minDistance:
            nearestNeighbor = i
            minDistance = adjMatrix[current][i]

      if nearestNeighbor = -1:
         return NULL  // No valid path found

      current = nearestNeighbor
      path.append(current)
      visited[current] = true

   return path
```

Aprašymas:

GreedyTSP implimentuoja godų algoritmą. Pagrindinė idėja yra pasirinkti artimiausią kaimyną kiekviename žingsnyje, siekiant sukurti trumpiausią galimą maršrutą.

Šis metodas pradedamas nuo pradinio miesto ir prideda jį į maršruto sąrašą. Tada jis pasirenka artimiausią kaimyną, prideda jį į maršruto sąrašą ir pažymi, kad šis kaimynas buvo aplankytas. Procesas kartojamas, kol visi miestai yra sudėti maršruto sąraše.

Metodas stengiasi rasti trumpiausią galimą maršrutą, pasirenkant artimiausius kaimynus, tačiau tai nėra garantuotas optimalus sprendimas šiai problemai. Greedy algoritmas gali duoti suboptimalų rezultatą, kuris yra trumpesnis nei tiesioginis linijinis maršrutas, bet ne trumpiausias galimas maršrutas.

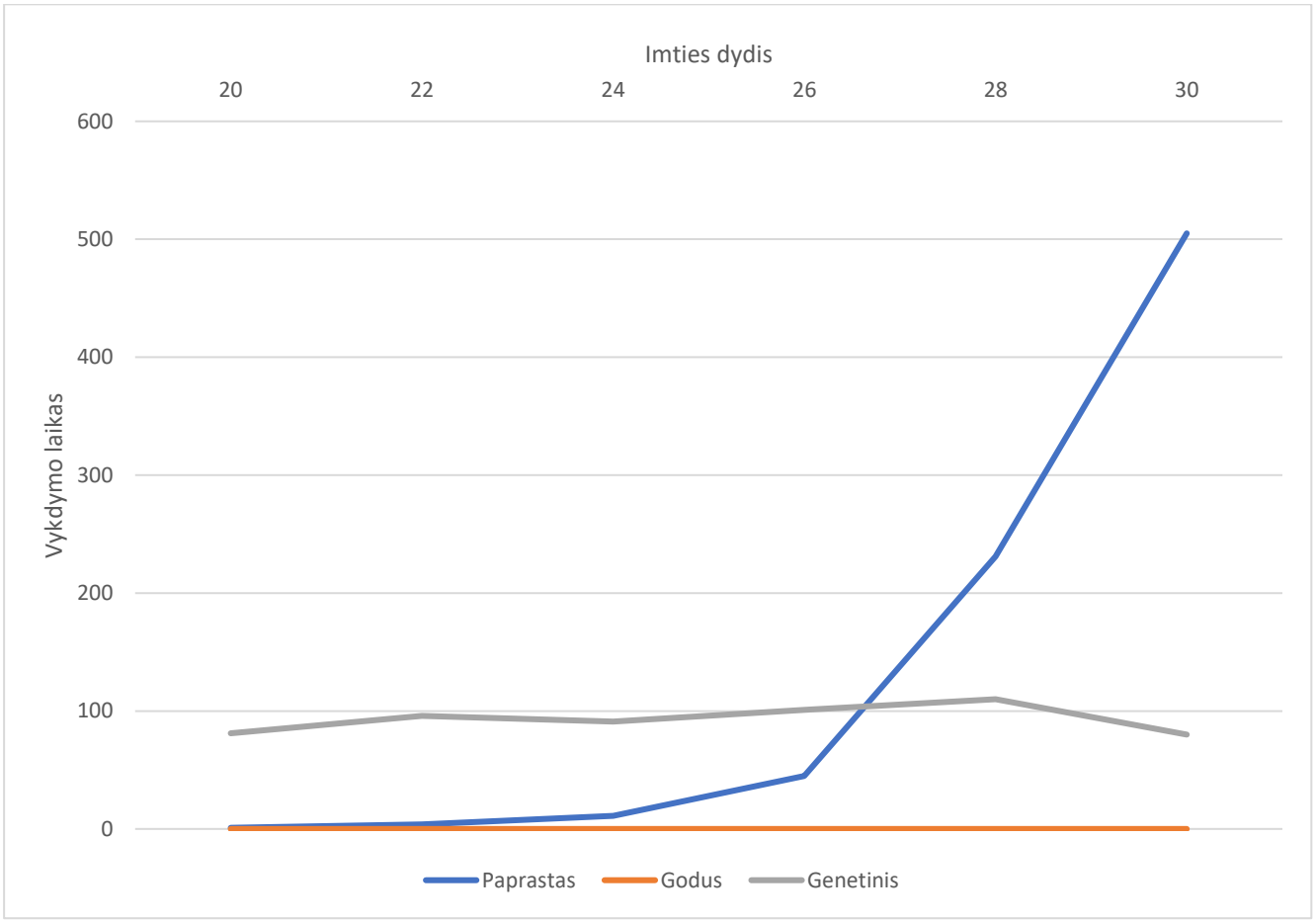
Kodo analizė

	Kaina	Kartai
`public static List<Location> GreedyTSP(double[,] adjMatrix, int start, List<Location> locations, List<Location> original)`		
`{`		
`int n = locations.Count;`	c	1
`bool[] visited = new bool[n];`	c	1
`List<Location> path = new List<Location>();`	c	1
`locations.Insert(0, original[start]);`	c	1
`path.Add(locations[0]);`	c	1
`visited[0] = true;`	c	1
`while (path.Count < n)`	c	n-1
`{`	c	n
`int lastCityIndex = path.Count - 1;`	c	n
`double minDistance = double.MaxValue;`	c	n
`int closestCityIndex = -1;`	c	n
`for (int i = 0; i < n; i++)`	c	(n-1)n
`{`		
`if (!visited[i] && adjMatrix[lastCityIndex, i] < minDistance)`	c	n^2
`{`		
`minDistance = adjMatrix[lastCityIndex, i];`	c	n^2
`closestCityIndex = i;`	c	n^2
`}`		
`}`		
`if (closestCityIndex != -1)`	c	n
`{`		
`path.Add(locations[closestCityIndex]);`	c	n
`visited[closestCityIndex] = true;`	c	n
`}`		
`}`		
`path.Add(original[start]); // Add the starting location to complete the cycle`	c	1
`return path;`	c	1
`}`		

$$T(n) = n^2 + C = O * (n^2)$$

Bendras šio metodo vykdymo laikas yra priklausomas nuo miestų skaičiaus ir skaičiavimo sąnaudų. Naudojant artimiausio kaimyno paiešką kiekviename žingsnyje, bendras laiko sudėtingumas yra O(n^2), kur n yra miestų skaičius. Tai reiškia, kad metodas yra efektyvesnis nei eksponentinis O(n!). Tačiau, nepaisant to, algoritmas gali būti nepakankamai efektyvus dideliems miestų skaičiams.

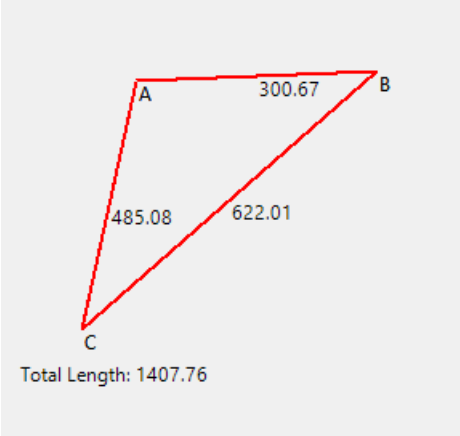
Našumo palyginimas

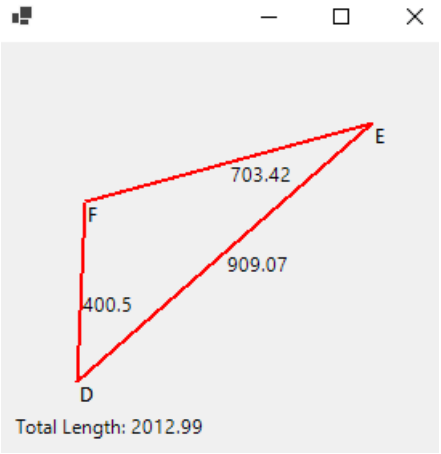


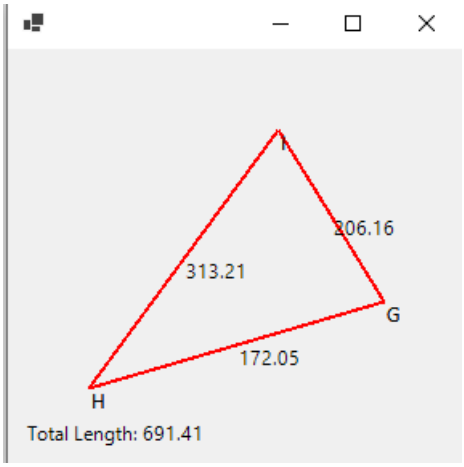
4. Pavyzdiniai duomenys faile data.csv :

```
A,4314994364,100,20
B,3713676194,400,0
C,5028700202,30,500
D,4829246478,0,600
E,1323938974,700,20
F,3296721649,20,200
G,1416904471,150,400
H,2837707104,10,500
I,5028700985,100,200
```

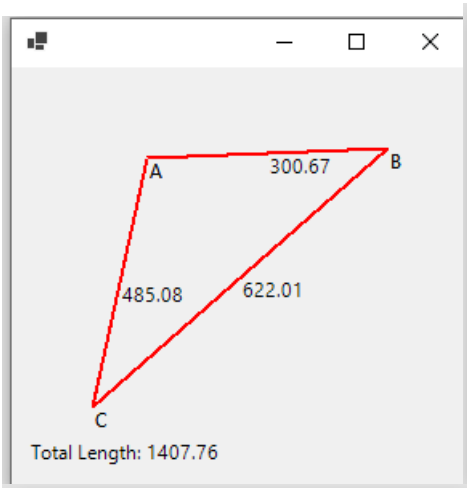
5. Paprastas algoritmas

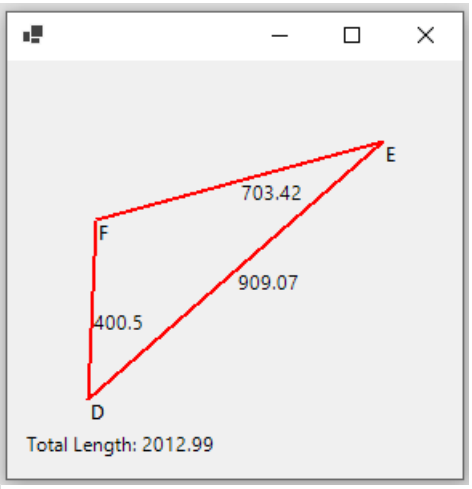


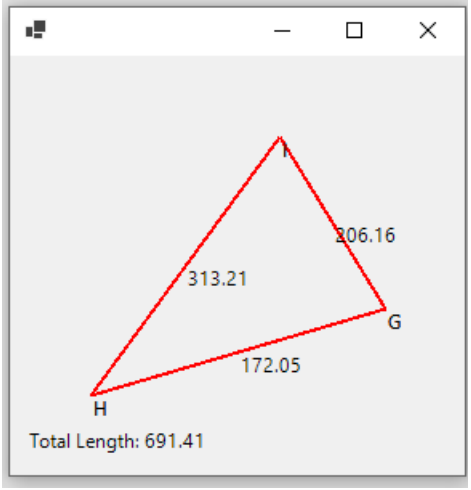




6. Godus algoritmas







7. Trečia dalis

***Aprašymas*:**

geneticTSP metodas (genetinis algoritmas TSP sprendimui) naudoja genetikos algoritmo principus, siekiant rasti artimą optimaliam maršrutą keliautojo. Šio metodo idėja yra simuliuoti evoliucijos procesą, kurio metu generuojamos ir kombinuojamos galimos sprendimų populiacijos, siekiant rasti geriausią maršrutą.

Šio metodo pagrindiniai žingsniai yra:

Pradinė populiacija: Sugeneruojama pradinė atsitiktinė maršrutų populiacija. Kiekvienas maršrutas yra koduojamas kaip chromosoma, kurioje užrašytas miestų lankymo tvarka.

Atpažinimas (fitness): Kiekvienam maršrutui apskaičiuojamas tinkamumo (fitness) įvertis, atsižvelgiant į jo nuotolį arba atstumą. Tinkamumo įvertinimas gali būti atvirkštinis atstumo, kurį norima minimizuoti, arba tiesioginis maršruto kokybės matas.

Selekcija: Atliekama selekcija, kurioje pagal tinkamumo įvertinimus parenkami geriausi maršrutai iš populiacijos, kurie bus paveldėti ir dalyvaus ateities kartose.

Kryžminimas (crossover): Pasirinktiems tėvams atliekamas kryžminimas, perkeliant jų chromosomų dalis vienas į kitą. Tai leidžia sukurti naujus individus, kombinuojant geriausius tėvų savybių.

Mutacija: Naujai sukurta populiacija keičiama mutacijos operacija, kuri atsitiktinai keičia arba pertvarko genetinę informaciją maršrutuose. Tai padeda išvengti pasiekto lokalaus optimumo ir leidžia plėtotis įvairioms galimybėms.

Pakartoti: Šie žingsniai kartojami kelis kartus (kartų skaičius yra parametras), siekiant evoliucionuoti ir gerinti populiacijos maršrutus.

Geriausio sprendimo pasirinkimas: Baigus iteracijas, iš populiacijos parenkamas geriausias individas (maršrutas) pagal tinkamumo įvertinimą. Tai yra suboptimalus arba artimas optimaliam TSP maršrutas.

Genetinis algoritmas naudojamas TSP yra heuristinis metodas, kuris negali garantuotai rasti optimalaus sprendimo, bet gali rasti artimus optimaliam sprendimus.

Kodo analizė

	Kaina	Kartai

```
 static Random random = new Random();

        public static List<Location> GeneticTSP(double[,] adjMatrix,
int start, List<Location> locations, int populationSize, int
maxGenerations)
        {
            int n = locations.Count;
            List<int[]> population = InitializePopulation(n,
populationSize);
            List<Location> bestPath = null;
            double bestFitness = double.MaxValue;

            for (int generation = 0; generation < maxGenerations;
generation++)
            {
                List<double> fitnessValues =
CalculateFitness(adjMatrix, population, locations);
                int eliteIndex = GetEliteIndex(fitnessValues);
                double eliteFitness = fitnessValues[eliteIndex];

                if (eliteFitness < bestFitness)
                {
                    bestFitness = eliteFitness;
                    bestPath = ConvertToPath(population[eliteIndex],
locations);
                }

                List<int[]> newPopulation = new List<int[]>();

                while (newPopulation.Count < populationSize)
                {
                    int[] parent1 = SelectParent(population,
fitnessValues);
                    int[] parent2 = SelectParent(population,
fitnessValues);

                    int[] child = Crossover(parent1, parent2);
                    child = Mutate(child);
                    newPopulation.Add(child);
                }

                population = newPopulation;
            }

            bestPath.Add(locations[start]); // Add the starting
location to complete the cycle

            return bestPath;
        }

        static List<int[]> InitializePopulation(int n, int
populationSize)
        {
            List<int[]> population = new List<int[]>();

            for (int i = 0; i < populationSize; i++)
            {
                int[] individual = Enumerable.Range(0, n).ToArray();
                ShuffleArray(individual);
                population.Add(individual);
            }

            return population;
        }
```

```
        static void ShuffleArray(int[] array)
        {
            int n = array.Length;

            for (int i = 0; i < n - 1; i++)
            {
                int j = random.Next(i, n);
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }

        static List<double> CalculateFitness(double[,] adjMatrix,
List<int[]> population, List<Location> locations)
        {
            List<double> fitnessValues = new List<double>();

            foreach (int[] individual in population)
            {
                double distance = 0;

                for (int i = 0; i < individual.Length - 1; i++)
                {
                    int city1 = individual[i];
                    int city2 = individual[i + 1];
                    distance += adjMatrix[city1, city2];
                }

                distance += adjMatrix[individual[individual.Length -
1], individual[0]];
                fitnessValues.Add(1 / distance);
            }

            return fitnessValues;
        }

        static int GetEliteIndex(List<double> fitnessValues)
        {
            int eliteIndex = 0;
            double maxFitness = double.MinValue;

            for (int i = 0; i < fitnessValues.Count; i++)
            {
                if (fitnessValues[i] > maxFitness)
                {
                    maxFitness = fitnessValues[i];
                    eliteIndex = i;
                }
            }

            return eliteIndex;
        }

        static List<Location> ConvertToPath(int[] individual,
List<Location> locations)
        {
            List<Location> path = new List<Location>();
            foreach (int cityIndex in individual)
            {
                path.Add(locations[cityIndex]);
            }

            return path;
        }
```

```
        static int[] SelectParent(List<int[]> population,
List<double> fitnessValues)
        {
            int index = RouletteWheelSelection(fitnessValues);
            return population[index];
        }

        static int RouletteWheelSelection(List<double> fitnessValues)
        {
            double totalFitness = fitnessValues.Sum();
            double randomValue = random.NextDouble() * totalFitness;
            double partialSum = 0;
            for (int i = 0; i < fitnessValues.Count; i++)
            {
                partialSum += fitnessValues[i];

                if (partialSum >= randomValue)
                {
                    return i;
                }
            }

            return fitnessValues.Count - 1;
        }

        static int[] Crossover(int[] parent1, int[] parent2)
        {
            int n = parent1.Length;
            int[] child = new int[n];
            int startPos = random.Next(n);
            int endPos = random.Next(n);
            for (int i = 0; i < n; i++)
            {
                if (startPos < endPos && i > startPos && i < endPos)
                {
                    child[i] = parent1[i];
                }
                else if (startPos > endPos && !(i < startPos && i >
endPos))
                {
                    child[i] = parent1[i];
                }
            }

            for (int i = 0; i < n; i++)
            {
                if (!child.Contains(parent2[i]))
                {
                    for (int j = 0; j < n; j++)
                    {
                        if (child[j] == 0)
                        {
                            child[j] = parent2[i];
                            break;
                        }
                    }
                }
            }

            return child;
        }
        static int[] Mutate(int[] individual)
        {
            int n = individual.Length;
            int mutationPoint1 = random.Next(n);
            int mutationPoint2 = random.Next(n);
            int temp = individual[mutationPoint1];
            individual[mutationPoint1] = individual[mutationPoint2];
```

```
            individual[mutationPoint2] = temp;
            return individual;
        }
```

$$T(n) = O(kartos * populaticija * n))$$