****

**Kauno technologijos universitetas**

Informatikos fakultetas

**Inžinerinio projekto ataskaita**

individuali užduotis

|  |  |
| --- | --- |
|  |  |
| **Aistis Jakutonis**  Studentas | (parašas) (data) |
|  |  |
| **Dalius Makackas**  **Andrius Kriščiūnas**  **Tadas Kraujalis**  **Vidmantas Rimavičius**  Dėstytojai | (parašas) (data) |
|  |  |

**Kaunas, 2025**

Turinys

[1. Pirma dalis 3](#_Toc198728151)

[1.1. Asimptotinis programos vykdymo laiko sudėtingumas 3](#_Toc198728152)

[1.2. Programos abstraktus aprašas – pseudo kodas 4](#_Toc198728153)

[1.3. Metodų paaiškinimas ir analizė 5](#_Toc198728154)

[1.4. Grafikai pavaizduotas sprendimas 6](#_Toc198728155)

[1.5. Algoritmo išlygiagretinimas 6](#_Toc198728156)

[1.6. Išlygiagretinimo paaiškinimas 8](#_Toc198728157)

[2. Antra dalis 9](#_Toc198728158)

[2.1. Asimptotinis programos vykdymo laiko sudėtingumas 9](#_Toc198728159)

[2.2. Programos abstraktus aprašas – pseudo kodas 10](#_Toc198728160)

[2.3. Metodų paaiškinimas ir analizė 11](#_Toc198728161)

[2.4. Grafikai pavaizduotas sprendimas 11](#_Toc198728162)

[2.5. Algoritmo išlygiagretinimas 12](#_Toc198728163)

[2.6. Išlygiagretinimo paaiškinimas 13](#_Toc198728164)

[3. Trečia dalis 14](#_Toc198728165)

[3.1. Asimptotinis programos vykdymo laiko sudėtingumas 14](#_Toc198728166)

[3.2. Programos abstraktus aprašas – pseudo kodas 16](#_Toc198728167)

[3.3. Metodų paaiškinimas ir analizė 18](#_Toc198728168)

[3.4. Grafikai pavaizduotas sprendimas 18](#_Toc198728169)

[3.5. Diskretinio optimizavimo parametrai užduoties sprendimui 19](#_Toc198728170)

[3.6. Algoritmo išlygiagretinimas 19](#_Toc198728171)

[3.7. Išlygiagretinimo paaiškinimas 20](#_Toc198728172)

# Pirma dalis

## Asimptotinis programos vykdymo laiko sudėtingumas

|  |  |  |  |
| --- | --- | --- | --- |
| Nr. | Kodas | Laikas | Kartai |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | **static** Place **FindNearest**(Place **from**, List<Place> unvisited)  {  Place nearest = **null**;  **double** minDist = **double**.MaxValue;  **foreach** (**var** place **in** unvisited)  {  **double** dist = **from**.DistanceTo(place);  **if** (dist < minDist)  {  minDist = dist;  nearest = place;  }  }  **return** nearest;  } | C1  C2  C3  C4  C5  C6  C7  C8 | 1  1  1 |
|  | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Nr. | Kodas | Laikas | Kartai |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | **static** **void** **Nuoseklus**(List<Place> unvisited, Place current1,  Place current2, Place current3, List<**double**> route1,  List<**double**> route2, List<**double**> route3, Place start)  {  **while** (unvisited.Count > **0**)  {  Place nearest1 = FindNearest(current1, unvisited);  **if** (nearest1 != **null**)  {  route1.Add(current1.DistanceTo(nearest1));  current1 = nearest1;  unvisited.Remove(nearest1);  }  **if** (unvisited.Count == **0**)  {  **break**;  }  Place nearest2 = FindNearest(current2, unvisited);  **if** (nearest2 != **null**)  {  route2.Add(current2.DistanceTo(nearest2));  current2 = nearest2;  unvisited.Remove(nearest2);  }  **if** (unvisited.Count == **0**)  {  **break**;  }  Place nearest3 = FindNearest(current3, unvisited);  **if** (nearest3 != **null**)  {  route3.Add(current3.DistanceTo(nearest3));  current3 = nearest3;  unvisited.Remove(nearest3);  }  }  route1.Add(current1.DistanceTo(start));  route2.Add(current2.DistanceTo(start));  route3.Add(current3.DistanceTo(start));  } | C1  C2  C3  C4  C5  C6  C7  C8  C9  C10  C11  C12  C13  C14  C15  C16  C17  C18  C19  C20  C21  C22  C23 | 1  1  1 |
|  | | | |

Galutinis kodo sudėtingumas:

## Programos abstraktus aprašas – pseudo kodas

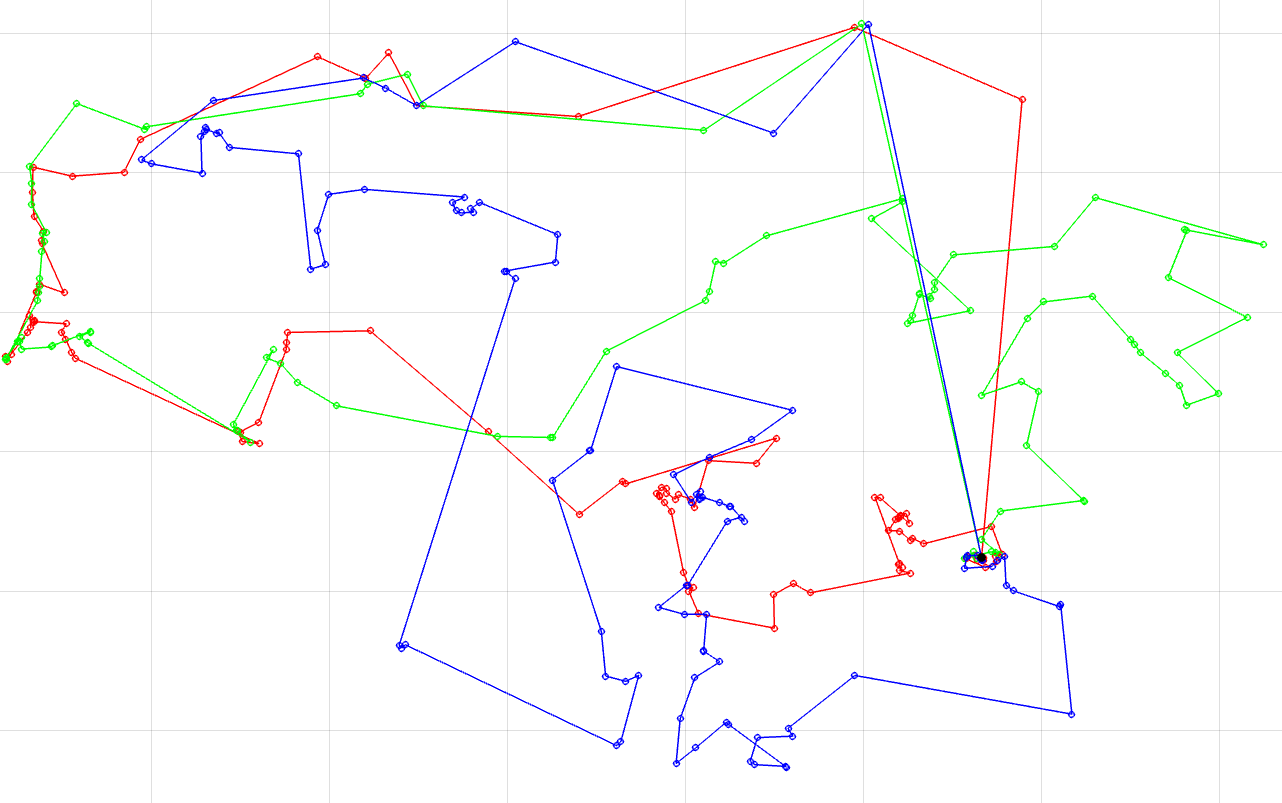
|  |
| --- |
| Funkcija Nuoseklus(unvisited, current1, current2, current3, route1, route2, route3, start):  Kol yra neaplankytų vietų:  // 1 autobusas ieško artimiausios vietos  Rask artimiausią vietą iš current1 tarp neaplankytų  Jei tokia vieta rasta:  Įrašyk atstumą į route1  current1 = ta vieta  Pašalink vietą iš neaplankytų  Jei nėra daugiau neaplankytų, nutrauk ciklą  // 2 autobusas ieško artimiausios vietos  Rask artimiausią vietą iš current2 tarp neaplankytų  Jei tokia vieta rasta:  Įrašyk atstumą į route2  current2 = ta vieta  Pašalink vietą iš neaplankytų  Jei nėra daugiau neaplankytų, nutrauk ciklą  // 3 autobusas ieško artimiausios vietos  Rask artimiausią vietą iš current3 tarp neaplankytų  Jei tokia vieta rasta:  Įrašyk atstumą į route3  current3 = ta vieta  Pašalink vietą iš neaplankytų  // Kai visos vietos aplankytos, grįžtama į pradinį tašką  Pridėk atstumą nuo current1 iki start į route1  Pridėk atstumą nuo current2 iki start į route2  Pridėk atstumą nuo current3 iki start į route3 |

## Metodų paaiškinimas ir analizė

Šiame nuosekliame sprendime yra naudojamas artimiausio kaimyno heuristinis metodas. Kol yra neaplankytų vietų algoritmas eina per kiekvieną autobusą paeiliui ir ieško tarp visų likusių vietų pačios artimiausios ir į ją važiuoja. Taip tęsiamas algoritmas iki kol nebelieka nepalankytų vietų. Šis metodas nėra visada optimalus, jis nesiekia rasti geriausio įmanomo sprendimo, o renkasi tiesiog artimiausią vietą.

## Grafikai pavaizduotas sprendimas

1 grafikas Artimiausio kaimyno metodu važiuojančių autobusų maršrutų grafikas



## Algoritmo išlygiagretinimas

|  |  |  |  |
| --- | --- | --- | --- |
| Nr. | Kodas | Laikas | Kartai |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113 | **static** **void** **Islygiagretintas**(List<Place> sharedUnvisited, List<**double**> route1, List<**double**> route2, List<**double**> route3, Place start, Place current1, Place current2, Place current3)  {  **object** lockObj = **new** **object**();  Parallel.Invoke(  () =>  {  **while** (**true**)  {  Place next = **null**;  **lock** (lockObj)  {  **if** (sharedUnvisited.Count == **0**)  {  **break**;  }  next = FindNearest(current1, sharedUnvisited);  **if** (next != **null**)  {  sharedUnvisited.Remove(next);  }  }  **if** (next != **null**)  {  route1.Add(current1.DistanceTo(next));  current1 = next;  }  }  route1.Add(current1.DistanceTo(start));  },  () =>  {  **while** (**true**)  {  Place next = **null**;  **lock** (lockObj)  {  **if** (sharedUnvisited.Count == **0**)  {  **break**;  }  next = FindNearest(current2, sharedUnvisited);  **if** (next != **null**)  {  sharedUnvisited.Remove(next);  }  }  **if** (next != **null**)  {  route2.Add(current2.DistanceTo(next));  current2 = next;  }  }  route2.Add(current2.DistanceTo(start));  },  () =>  {  **while** (**true**)  {  Place next = **null**;  **lock** (lockObj)  {  **if** (sharedUnvisited.Count == **0**)  {  **break**;  }  next = FindNearest(current3, sharedUnvisited);  **if** (next != **null**)  {  sharedUnvisited.Remove(next);  }  }  **if** (next != **null**)  {  route3.Add(current3.DistanceTo(next));  current3 = next;  }  }  route3.Add(current3.DistanceTo(start));  }  );  } | C1  C2  C3  C4  C5  C6  C7  C8  C9  C10  C11  C12  C13  C14  C15  C16  C17  C18  C19  C20  C21  C22  C23  C24  C25  C26  C27  C28  C29  C30  C31  C32  C33  C34  C35  C36  C37  C38 | 1  1  1  1 |
|  | | | |

Galutinis kodo sudėtingumas: . Veikia šiek tiek greičiau nei prieš tai buvęs nuoseklus.

## Išlygiagretinimo paaiškinimas

Visi trys autobusai dalijasi vienu vietų sąrašu. Todėl kiekvienas autobusas vykdo savo atskiras paralelines užduotis ir ieško artimiausių kelių iki kitų vietų. Kadangi visi autobusai turi prieigą prie bendro sąrašo ir renkasi duomenis iš jo, tai tam yra naudojamas užraktas, kad kol vienas autobusas renkasi duomenis kiti tų duomenų negalėtų pamodifikuoti. Tokiu metodu maršrutų skaičiavimas vyksta kiek greičiau, bet dėl autobusų lygiagrečių skaičiavimų rezultatai nesutampa su nuosekliais.

# Antra dalis

## Asimptotinis programos vykdymo laiko sudėtingumas

|  |  |  |  |
| --- | --- | --- | --- |
| Nr. | Kodas | Laikas | Kartai |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60 | **static** **void** **Branch**(State state, List<Place> remaining, Place start, **ref** **double** bestTime, **ref** State bestState, Stopwatch timer)  {  **if** (timer.Elapsed.TotalSeconds > **10**)  {  Console.WriteLine("timeend");  **return**;  }  **if** (state.Visited.Count == remaining.Count)  {  **var** cloned = state.Clone();  **for** (**int** i = **0**; i < **3**; i++)  {  **var** last = cloned.Routes[i].Last();  cloned.Distances[i] += last.DistanceTo(start);  cloned.Routes[i].Add(start);  }  **double** maxTime = cloned.Distances.Max();  **if** (maxTime < bestTime)  {  bestTime = maxTime;  bestState = cloned;  }  **return**;  }  **foreach** (**var** place **in** remaining)  {  **if** (state.Visited.Contains(place.Id))  {  **continue**;  }  **for** (**int** bus = **0**; bus < **3**; bus++)  {  **var** newState = state.Clone();  **var** lastPlace = newState.Routes[bus].Last();  newState.Routes[bus].Add(place);  newState.Distances[bus] += lastPlace.DistanceTo(place);  newState.Visited.Add(place.Id);  **if** (newState.Bound() < bestTime)  Branch(newState, remaining, start, **ref** bestTime, **ref** bestState, timer);  }  }  } | C1  C2  C3  C4  C5  C6  C7  C8  C9  C10  C11  C12  C13  C14  C15  C16  C17  C18  C19  C20  C21  C22  C23  C24  C25 | 1  1  1  1  1  4  3  3  3  1  1  1  1  1 |
|  | | | |

Galutinis kodo sudėtingumas:

## Programos abstraktus aprašas – pseudo kodas

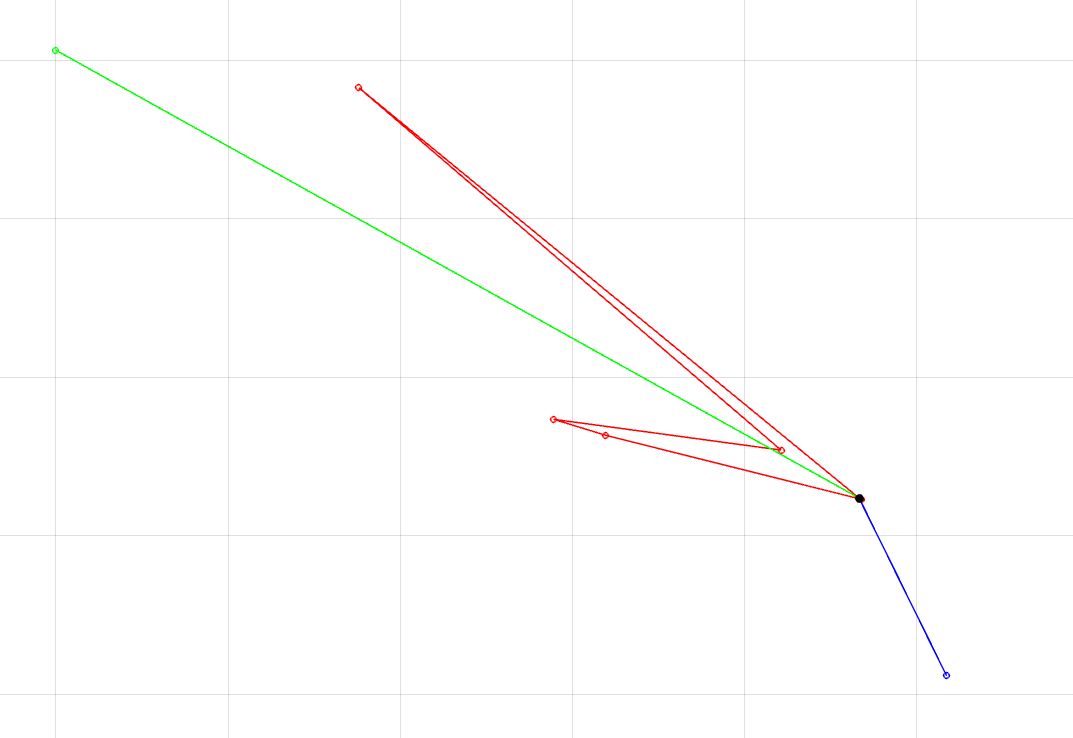
|  |
| --- |
| Funkcija Branch(būsena, likusiosVietos, startoVieta, geriausiasLaikas, geriausiaBūsena, laikmatis):  Jei praeita daugiau nei 10 sekundžių:  Baigti (laikas baigėsi)  Jei visos vietos jau aplankytos:  Sukurti kopiją iš esamos būsenos  Kiekvienam autobusui:  Pridėti kelią nuo paskutinės vietos iki starto  Jei maksimalus maršruto ilgis < geriausiasLaikas:  Atnaujinti geriausią laiką  Išsaugoti esamą būseną kaip geriausią  Grįžti  Kiekvienai neaplankytai vietai:  Jei vieta jau aplankyta – praleisti  Kiekvienam autobusui:  Sukurti naują būsenos kopiją  Gauti paskutinę autobuso maršruto vietą  Pridėti pasirinktą vietą į autobuso maršrutą  Atnaujinti maršruto atstumą  Pažymėti vietą kaip aplankytą  Jei naujos būsenos maksimalus atstumas (Bound) < geriausiasLaikas:  Rekursyviai iškviesti Branch su nauja būsena |

## Metodų paaiškinimas ir analizė

Čia yra naudojamas šakų ir rėžių metodas. Jis ieško geriausio maršruto visiems trims autobusams kartu. Kiekviename žingsnyje bandoma priskirti naują neaplankytą vietą vienam iš autobusų ir taip eina gilyn rekursiškai. Tuomet, jei dabartinė būsena jau yra blogesnė nei geriausias rastas sprendinys, tai tada ta šaka yra nutraukiama. Grįžtama šiek tiek atgal ir bandoma naudoti kitą vietą ir taip toliau. Galiausiai yra randamas optimalus sprendinys. Tačiau šis algoritmas tikrindamas labai daug kombinacijų užtrunka gan ilgai.

## Grafikai pavaizduotas sprendimas

2 grafikas Šakų ir rėžių metodu važiuojančių autobusų optimaliausi maršrutai



## Algoritmo išlygiagretinimas

|  |  |  |  |
| --- | --- | --- | --- |
| Nr. | Kodas | Laikas | Kartai |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95 | **static** (**double** bestTime, State bestState) BranchParallel(State state, List<Place> remaining, Place start, Stopwatch timer, **int** depth = **0**)  {  **if** (timer.Elapsed.TotalSeconds > **10**)  **return** (**double**.MaxValue, **null**);  **if** (state.Visited.Count == remaining.Count)  {  **var** cloned = state.Clone();  **for** (**int** i = **0**; i < **3**; i++)  {  **var** last = cloned.Routes[i].Last();  cloned.Distances[i] += last.DistanceTo(start);  cloned.Routes[i].Add(start);  }  **double** maxTime = cloned.Distances.Max();  **return** (maxTime, cloned);  }  **double** bestTime = **double**.MaxValue;  State bestState = **null**;  **var** tasks = **new** List<Task<(**double**, State)>>();  **foreach** (**var** place **in** remaining)  {  **if** (state.Visited.Contains(place.Id))  **continue**;  **for** (**int** bus = **0**; bus < **3**; bus++)  {  **var** newState = state.Clone();  **var** lastPlace = newState.Routes[bus].Last();  newState.Routes[bus].Add(place);  newState.Distances[bus] += lastPlace.DistanceTo(place);  newState.Visited.Add(place.Id);  **if** (newState.Bound() >= bestTime)  **continue**;  **if** (depth < **2**)  {  tasks.Add(Task.Run(() =>  {  **return** **BranchParallel**(newState, remaining, start, timer, depth + **1**);  }));  }  **else**  {  **var** (bTime, bState) = BranchParallel(newState, remaining, start, timer, depth + **1**);  **if** (bState != **null** && bTime < bestTime)  {  bestTime = bTime;  bestState = bState;  }  }  }  }  **if** (tasks.Count > **0**)  {  Task.WaitAll(tasks.ToArray());  **foreach** (**var** t **in** tasks)  {  **var** (tTime, tState) = t.Result;  **if** (tState != **null** && tTime < bestTime)  {  bestTime = tTime;  bestState = tState;  }  }  }  **return** (bestTime, bestState);  } | C1  C2  C3  C4  C5  C6  C7  C8  C9  C10  C11  C12  C13  C14  C15  C16  C17  C18  C19  C20  C21  C22  C23  C24  C25  C26  C27  C28  C29  C30  C31  C32  C33  C34  C35  C36  C37  C38  C39  C40 | 1  1  1  1  4  3  3  3  3  3  1  1  1  1  1  1 |
|  | | | |

Galutinis kodo sudėtingumas:

## Išlygiagretinimo paaiškinimas

Šiame išlygiagretinime pradžioje kiekviena vieta bandoma priskirti visiems trims autobusams lygiagrečiai. Vėliau, jei gylis nėra didelis, kiekviena šaka taip pat yra vykdoma atskirai. Čia kaip ir praeitame atvejyje duomenys yra apsaugojami juos užrakinant.

# Trečia dalis

## Asimptotinis programos vykdymo laiko sudėtingumas

|  |  |  |  |
| --- | --- | --- | --- |
| Nr. | Kodas | Laikas | Kartai |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **static** **double** **RouteDistance**(List<Place> route, Place start)  {  **double** dist = **0.0**;  **var** current = start;  **foreach** (**var** p **in** route)  {  dist += current.DistanceTo(p);  current = p;  }  dist += current.DistanceTo(start);  **return** dist;  } | C1  C2  C3  C4  C5  C6  C7 | 1  1  1  1 |
|  | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Nr. | Kodas | Laikas | Kartai |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | **static** BusSolution **MutateBusSolution**(BusSolution solution)  {  **var** newSol = solution.Clone();  **if** (rand.NextDouble() < **0.5**)  {  **int** **from** = rand.Next(**3**);  **int** to = rand.Next(**3**);  **if** (**from** != to && newSol.Routes[**from**].Count > **0**)  {  **int** index = rand.Next(newSol.Routes[**from**].Count);  **var** place = newSol.Routes[**from**][index];  newSol.Routes[**from**].RemoveAt(index);  newSol.Routes[to].Add(place);  }  }  **else**  {  **int** bus = rand.Next(**3**);  **if** (newSol.Routes[bus].Count > **1**)  {  **int** i = rand.Next(newSol.Routes[bus].Count);  **int** j = rand.Next(newSol.Routes[bus].Count);  (newSol.Routes[bus][i], newSol.Routes[bus][j]) = (newSol.Routes[bus][j], newSol.Routes[bus][i]);  }  }  **return** newSol;  } | C1  C2  C3  C4  C5  C6  C7  C8  C9  C10  C11  C12  C13  C14  C15  C16 | 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1 |
|  | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Nr. | Kodas | Laikas | Kartai |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **static** BusSolution **CreateInitialBusSolution**(List<Place> places)  {  **var** solution = **new** BusSolution();  **var** shuffled = places.OrderBy(x => rand.Next()).ToList();  **for** (**int** i = **0**; i < shuffled.Count; i++)  solution.Routes[i % **3**].Add(shuffled[i]);  **return** solution;  } | C1  C2  C3  C4  C5 | 1  1  1 |
|  | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Nr. | Kodas | Laikas | Kartai |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | **static** BusSolution **SimulatedAnnealingNuoseklus**(List<Place> allPlaces, Place start, Stopwatch timer)  {  **double** temp = **1000**;  **double** coolingRate = **0.99999**;  **double** minTemp = **1e-4**;  **int** noImprovement = **0**;  **int** maxNoImprovement = **10000**;  **var** current = CreateInitialBusSolution(allPlaces);  **var** best = current.Clone();  **double** bestDist = best.Routes.Max(r => RouteDistance(r, start));  **int** i = **0**;  **while** (temp > minTemp && noImprovement < maxNoImprovement)  {  **if** (timer.Elapsed.TotalSeconds > **10**)  {  Console.WriteLine("timeend");  **break**;  }  **var** next = MutateBusSolution(current);  **double** nextDist = next.Routes.Max(r => RouteDistance(r, start));  **double** delta = nextDist - bestDist;  i++;  **if** (delta < **0** || Math.Exp(-delta / temp) > rand.NextDouble())  {  current = next;  **if** (nextDist < bestDist)  {  best = next.Clone();  bestDist = nextDist;  noImprovement = **0**;  Console.WriteLine($"{i} = {bestDist}");  }  **else** noImprovement++;  }  temp \*= coolingRate;  }  **return** best;  } | C1  C2  C2  C4  C5  C6  C7  C8  C9  C10  C11  C12  C13  C14  C15  C16  C17  C18  C19  C20  C21  C22  C23  C24  C25  C26  C27 | 1  1  1  1  1  n  1  n  1  1  1  1  1  n  1  1  1  1  1  1  1  1  1  1  1  1 |
|  | | | |

## Programos abstraktus aprašas – pseudo kodas

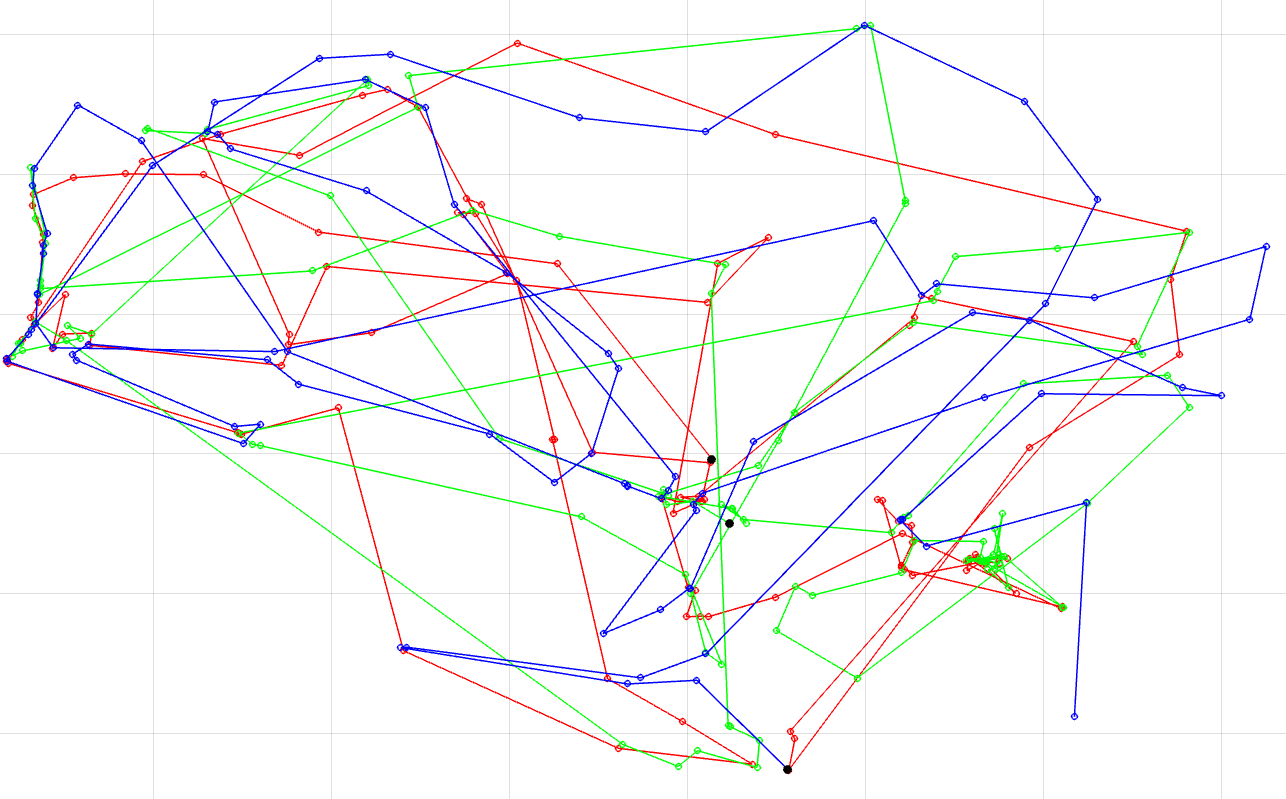
|  |
| --- |
| Funkcija SimulatedAnnealingNuoseklus(vietos, starto\_taskas, laikmatis):  Inicializuoti pradine temperatura = 1000  Nustatyti vėsinimo tempą (pvz., 0.99999)  Nustatyti minimalią temperatūrą ir maksimalių iteracijų be pagerėjimo skaičių  Sukurti pradine atsitiktinę sprendimo versiją (trys autobusų maršrutai)  Apskaičiuoti maksimalų vieno iš trijų autobusų maršruto ilgį – tai bus "geriausias iki šiol"  Kol temperatūra > minimali ir nepagerinta per daug iteracijų:  Jei praejo daugiau nei 10 sekundžių – išeiti  // Sukuriamas naujas kandidatinis sprendimas:  Kopijuoti dabartinį sprendimą kaip naują  Atsitiktinai pasirinkti vieną iš dviejų mutacijų:  1) Perkelti atsitiktinę vietą iš vieno autobuso į kitą:  - Pasirinkti atsitiktinius du skirtingus autobusus (FROM ir TO)  - Jei FROM turi bent vieną vietą:  - Pašalinti atsitiktinę vietą iš FROM  - Pridėti ją į TO  ARBA  2) Sukeisti dvi vietas vieno autobuso maršrute:  - Pasirinkti atsitiktinį autobusą  - Jei jis turi bent 2 vietas:  - Atsitiktinai parinkti dvi pozicijas ir sukeisti jas vietomis  Apskaičiuoti šio kandidato maksimalų vieno autobuso maršruto ilgį  Skirtumas = kandidato ilgis - geriausias ilgis  Jei skirtumas < 0 (geresnis) ARBA priimamas pagal tikimybę (Math.Exp(-delta / temp)):  Priimti kandidatą kaip naują dabartinį sprendimą  Jei jis taip pat yra geresnis nei geriausias iki šiol:  Atnaujinti geriausią sprendimą  Nustatyti pagerėjimų skaitiklį į 0  Kitu atveju:  Padidinti be pagerėjimo skaitiklį  Temperatūra mažinama (temp \*= coolingRate)  Grąžinti geriausią rastą sprendimą |

## Metodų paaiškinimas ir analizė

Čia buvo naudojamas atvėsinimo algoritmas, kuris leidžia ieškoti optimalaus sprendimo. Šiame algoritme kiekvienas galimas būsimos būsenos variantas yra gaunamas taikant atsitiktinę mutaciją – tai gali būti viename autobuse vietų sukeitimas ar tarp dviejų autobusų vietų keitimas. Kiekviena nauja būsena yra įvertinama ir jei ji yra geresnė tai ji yra priimama. Jei blogesnė ji taip pat gali būti priimama su tam tikra tikimybe, priklausančia nuo temperatūros ir sprendinių skirtumo. Temperatūra vis mažinama, taip ribojant blogesnių sprendinių priėmimą vėlesnėse iteracijose.

## Grafikai pavaizduotas sprendimas

3 grafikas Atkaitinimo metodu važiuojančių autobusų grafikas



## Diskretinio optimizavimo parametrai užduoties sprendimui

Sprendžiant užduotį buvo taikytas diskretinis optimizavimas, kurio paieškos erdvę sudarė visos įmanomos vietų kombinacijos paskirstant jas į 3 autobusus. Sprendinys – tai konkretus vietų išdėstymas tarp autobusų, o tikslo funkcija buvo maksimalaus vieno autobuso maršruto ilgis, kurį siekta minimizuoti. Apribojimų tiesiogiai nebuvo taikoma, išskyrus tai, kad kiekviena vieta gali būti paskirta tik vienam autobusui. Pagrindiniai optimizavimo algoritmo parametrai:

* **Pradinė temperatūra**: 1000 – leido pradžioje laisvai priimti ir blogesnius sprendimus.
* **Temperatūros mažinimo koeficientas**: 0.99999 – užtikrino laipsnišką perėjimą prie mažesnės sprendimų priėmimo rizikos.
* **Minimalios temperatūros riba**: 0.0001 – sustabdymo kriterijus.
* **Maksimalus negerėjančių iteracijų skaičius**: 10 000 – taip pat naudotas kaip sustabdymo sąlyga.

## Algoritmo išlygiagretinimas

|  |  |  |  |
| --- | --- | --- | --- |
| Nr. | Kodas | Laikas | Kartai |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80 | **static** BusSolution **SimulatedAnnealingLygiagretus**(List<Place> allPlaces, Place start, Stopwatch timer)  {  **double** temp = **1000**;  **double** coolingRate = **0.99999**;  **double** minTemp = **1e-4**;  **int** noImprovement = **0**;  **int** maxNoImprovement = **10000**;  **int** candidatesPerIter = Environment.ProcessorCount;  **var** current = CreateInitialBusSolution(allPlaces);  **var** best = current.Clone();  **double** bestDist = best.Routes.Max(r => RouteDistance(r, start));  **while** (temp > minTemp && noImprovement < maxNoImprovement)  {  **if** (timer.Elapsed.TotalSeconds > **10**)  {  Console.WriteLine("timeend");  **break**;  }  **var** candidates = **new** BusSolution[candidatesPerIter];  Parallel.For(**0**, candidatesPerIter, i =>  {  candidates[i] = MutateBusSolution(current);  });  BusSolution bestCandidate = **null**;  **double** bestCandidateDist = **double**.MaxValue;  **foreach** (**var** candidate **in** candidates)  {  **double** dist = candidate.Routes.Max(r => RouteDistance(r, start));  **if** (dist < bestCandidateDist)  {  bestCandidate = candidate;  bestCandidateDist = dist;  }  }  **double** delta = bestCandidateDist - bestDist;  **if** (delta < **0** || Math.Exp(-delta / temp) > rand.NextDouble())  {  current = bestCandidate;  **if** (bestCandidateDist < bestDist)  {  best = bestCandidate.Clone();  bestDist = bestCandidateDist;  noImprovement = **0**;  }  **else**  {  noImprovement++;  }  }  temp \*= coolingRate;  }  **return** best;  } | C1  C2  C3  C4  C5  C6  C7  C8  C9  C10  C11  C12  C13  C14  C15  C16  C17  C18  C19  C20  C21  C22  C23  C24  C25  C26  C27  C28  C29  C30  C31  C32  C33  C34 | 1  1  1  1  1  1  n  1  n  1  1  1  1  1  1  1  1  1  n  1  1  1  1  1  1  1  1  1  1  1  1  1  1 |
|  | | | |

Galutinis kodo sudėtingumas:

## Išlygiagretinimo paaiškinimas

Kiekvienoje iteracijoje vietoje vieno sprendinio mutacijos, lygiagrečiai sukuriama tiek kandidatų sprendinių, kiek yra procesorių branduolių. Šie sprendiniai generuojami naudojant tą pačią MutateBusSolution funkciją, bet vykdomi lygiagrečiai. Iš sugeneruotų variantų išrenkamas geriausias ir jis lyginamas su dabartiniu sprendimu pagal atvėsinimo kriterijų. Šis metodas leidžia vienu metu išbandyti kelis galimus patobulinimus.