Kauno technologijos universitetas

Informatikos fakultetas

Individualus projektas

Laboratorinio darbo ataskaita

Algoritmų sudarymas ir analizė (P170B400)

Ataskaitą parengė

Kristupas Paulauskas

Laboratoriniam darbui vadovavo

dr. Dalius Makackas

Kaunas, 2026

Turinys

1. Pirma dalis 3
1.1. Programos vykdymo laiko sudėtingumo analizė 3
1.2. Algoritmo abstraktus aprašas 3
1.3. Rezultatų analizė 4
1.4. Grafinis maršrutų atvaizdavimas 5
2. Antra dalis 6
2.1. Programos vykdymo laiko sudėtingumo analizė 6
2.2. Algoritmo abstraktus aprašas 6
2.3. Rezultatų analizė 6
2.4. Grafinis atvaizdavimas 7
3. Trečia dalis 8
3.1. Programos vykdymo laiko sudėtingumo analizė 8
3.2. Algoritmo abstraktus aprašas 8
3.3. Rezultatų analizė 8
3.4. Grafinis maršrutų atvaizdavimas 9
4. Trečia dalis 11
4.1. „Greedy“ algoritmo benchmark analizė 11
4.2. Lygiagretaus genetinio algoritmo analizė 11
5. Galutinis algoritmų palyginimas 13

1. Pirma dalis

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).

1.1. Programos vykdymo laiko sudėtingumo analizė

Programos sudėtingumą lemia trys pagrindinės dalys:

- Rūšiavimas, naudojamas standartinis rūšiavimas, kurio sudėtingumas yra $O(n \log n)$, kur n yra vietovių skaičius.
- Maršruto sudarymas (BuildRoute): tai dvigubas ciklas. Išoriniame cikle einama per visus taškus n, o vidiniame ieškoma artimiausio kaimyno tarp dar neaplankytų taškų. Sudėtingumas $O(n^2)$.
- TwoOpt funkcija, kurią sudaro while ciklas, kuris vyksta tol, kol randamas pagerėjimas, ir du vidiniai for ciklai, kurie tikrina visas briaunų poras. Blogiausiu atveju sudėtingumas $O(i \cdot n^2)$, kur i iteracijų skaičius.

Iš šių funkcijų galime prieiti išvadą, jog bendras asimptotinis sudėtingumas yra vidutiniu atveju yra $O(n^2)$.

1.2. Algoritmo abstraktus aprašas

```
ALGORITMAS GreedySolve(vietovės, pradžia):
   1. Randa pradinį tašką S.
   2. Surūšiuoja visas kitas vietoves pagal kampą taško S atžvilgiu (Atan2).
   3. Padalina surūšiuotą sąrašą į dvi lygias dalis (Grupė A ir Grupė B).
   4. Kiekvienai grupei (A ir B) vykdo BuildRoute:
      a. Dabartinis taškas = S.
      b. Kol yra neaplankytų taškų:
         i. Randa artimiausią tašką T nuo dabartinio taško.
         ii. Prideda T į maršrutą, pažymi kaip aplankytą.
         iii. Dabartinis taškas = T.
      c. Grįžta į S.
   5. Kiekvienam maršrutui vykdo 2-opt optimizavimą:
      a. Kol randamas kelio sutrumpėjimas:
         i. Tikrina visas briaunų poras (i, j).
         ii. Jei sukeitus briaunų jungtis atstumas mažėja -> apverčia maršruto
segmentą.
 6. Grąžina geriausią sprendinį.
```

```
ALGORITMAS BuildRoute(pradžia, vietovės):
    1. Sukuria tuščią sąrašą 'maršrutas'.
    2. Į 'maršrutas' įtraukia 'pradžia'.
    3. Sukuria sąrašą 'likę_taškai' iš visų gautų vietovių.
    4. Kol 'likę_taškai' nėra tuščias:
       a. Nustato mažiausią_atstumą = begalybė.
       b. Peržiūri kiekvieną tašką T iš 'likę_taškai':
          i. Apskaičiuoja atstumą nuo paskutinio maršruto taško iki T.
          ii. Jei atstumas < mažiausias_atstumas:
              - mažiausias_atstumas = atstumas
              - geriausias_taškas = T.
       c. Pašalina geriausią_tašką iš 'likę_taškai'.
       d. Prideda geriausią_tašką į 'maršrutas'.
    5. Prideda 'pradžia' į 'maršrutas' (grįžimas namo).
 6. Grąžina 'maršrutas'.
```

```
ALGORITMAS TwoOpt(maršrutas):
    1. kintamasis 'ar_pagerėjo' = tiesa.
    2. Kol 'ar_pagerėjo' == tiesa:
       a. 'ar_pagerėjo' = melas.
       b. Kiekvienai briaunai i nuo 1 iki (n-2):
          i. Kiekvienai briaunai j nuo (i+1) iki (n-1):
             1. Apskaičiuoja dabartinį atstumą: Dist(i-1, i) + Dist(j, j+1).
             2. Apskaičiuoja potencialų atstumą: Dist(i-1, j) + Dist(i, j+1).
             3. Jei potencialus atstumas < dabartinis atstumas:
                - Apverčia maršruto dalį tarp i ir j.
                - 'ar_pagerėjo' = tiesa.
 3. Grąžina optimizuotą 'maršrutas'.
```

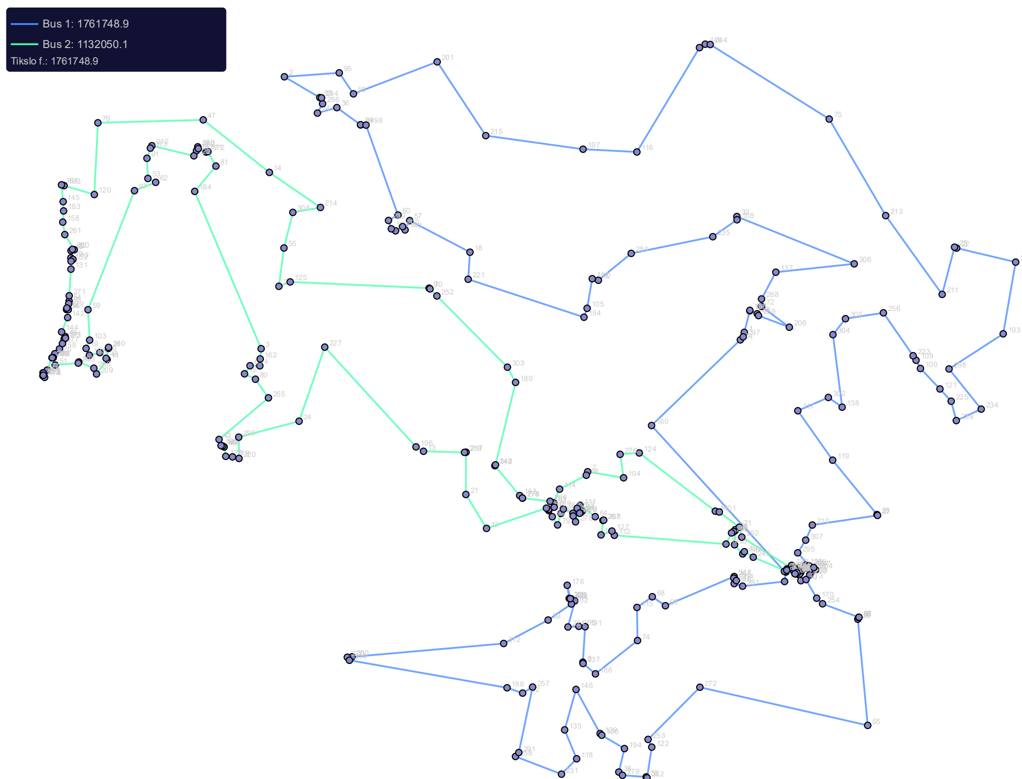
1.3. Rezultatų analizė

Kai pradinė vieta yra 67, o bendras vietų skaičius N = 314, gauname, jog algoritmo vykdymo laikas yra 12ms.

Maršrutas	Vietų skaičius	Atstumas
Bus 1	156	1761748
Bus 2	157	1132050

Galime pastebėti nemažą atstumų skirtumą tarp pirmo ir antro autobuso (skirtumas apie 629 tūkst. vnt.). Tai paaiškinama tuo, kad „Greedy“ algoritmas taškus skirsto pagal koordinates pradinio taško atžvilgiu, neatsižvelgiant į realų tankumą ar atstumus sekcijos viduje. „Bus 1“ maršrutas apima rečiau išsidėsčiusius taškus.

1.4. Grafinis maršrutų atvaizdavimas



2. Antra dalis

Realizuoti programą, kuri individualiai problemai pateiktų optimalų sprendinį (taikomas šakų ir rėžių metodas). Nustatyti, prie kokios duomenų apimties sprendinį pavyksta rasti jei programos vykdymo laikas negali būti ilgesnis nei 10 sek.

2.1. Programos vykdymo laiko sudėtingumo analizė

Šakų ir rėžių metodas naudojamas siekiant rasti tikslų (optimalų) sprendinį. Skirtingai nei „godusis“ algoritmas, šis metodas turi peržiūrėti visas galimas kombinacijas, tačiau jas atmeta naudodamas rėžius.

- Sudėtingumas: blogiausiu atveju šio algoritmo sudėtingumas yra eksponentinis $O(2^n \cdot n)$, arba $O(n!)$ priklausomai nuo šakojimosi.
- mTSP problemos versijoje kiekvienas miestas gali būti priskirtas arba pirmajam, arba antrajam maršrutui, todėl paieškos medis auga labai sparčiai.
- Atminties sudėtingumas: kadangi naudojamas (Stack<State>), atminties sudėtingumas priklauso nuo medžio gylio, t. y. $O(n)$.

2.2. Algoritmo abstraktus aprašas

```
ALGORITMAS BranchAndBound(vietovės, pradžia, laiko_riba, max_n):
   1. Atrenka 'max_n' vietovių ir pradinį tašką.
   2. Apskaičiuoja atstumų matricą D tarp visų taškų.
   3. Inicializuoja:
     - 'bestCost' = begalybė
     - 'stack' = įdeda pradinę būseną (abu maršrutai prasideda taške 0)
   4. Kol 'stack' ne tuščias IR laikas neviršytas:
     a. Išima būseną S.
     b. Jei visi taškai aplankyti:
       i. Prideda grįžimą į pradžią, apskaičiuoja Max(Cost1, Cost2).
       ii. Jei rezultatas < bestCost -> atnaujina geriausią sprendinį.
     c. Jei dabartinis Max(Cost1, Cost2) >= bestCost -> ATMESTI (Rėžis).
     d. Paima kitą neaplankytą tašką K:
       i. Bando priskirti K pirmam maršrutui:
         - Jei naujas Cost1 < bestCost -> sukuria būseną ir įdeda į dėklą.
       ii. Bando priskirti K antram maršrutui:
         - Jei naujas Cost2 < bestCost -> sukuria būseną ir įdeda į dėklą.
   5. Jei laikas baigėsi ir nerasta nieko -> kviečia Greedy algoritmą.
 6. Grąžina optimalų arba geriausią rastą sprendinį.
```

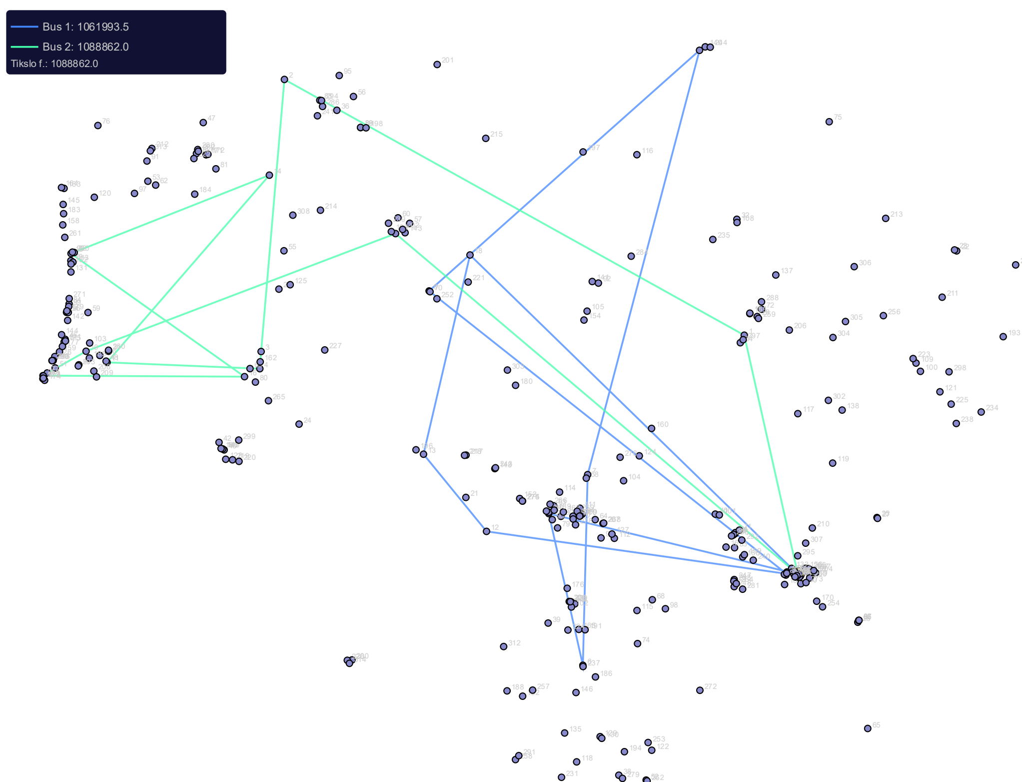
2.3. Rezultatų analizė

Siekiant rasti optimalų sprendinį, buvo apribota duomenų apimtis iki N = 20 (pirmi 20 mazgų), nustatant 10 sekundžių laiko limitą. Algoritmas sprendinį rado per 36 ms.

Maršrutas	Vietų skaičius	Atstumas
Bus 1	9	1061993
Bus 2	11	1088862

Atlikus tikslią paiešką, matome gerokai geresnį balansą: skirtumas tarp maršrutų yra tik apie 26868. Tai įrodo, kad Šakų ir rėžių metodas sugeba subalansuoti abiejų agentų krūvį, tačiau dėl eksponentinio sudėtingumo jis praktiškai pritaikomas tik mažoms duomenų imtims.

2.4. Grafinis atvaizdavimas



3. Trečia dalis

Realizuoti programą, kuri pateiktų sprendinį taikant pasirinktą diskretinio optimizavimo metodą (pvz. atkaitinimo algoritmą ar genetinio optimizavimo metodą). Programa pateikti rezultatą turi ne ilgiau, nei per 60 sek.

3.1. Programos vykdymo laiko sudėtingumo analizė

Genetinio algoritmo sudėtingumas priklauso nuo populiacijos dydžio (P), generacijų skaičiaus (G) ir individo maršruto ilgio (N).

- Vienos generacijos sudėtingumas: $O(P \cdot N)$, nes kiekvienam individui reikia apskaičiuoti tinkamumo funkciją (fitness), atlikti kryžminimą ir mutaciją.
- Bendra asimptotė: $O(G \cdot P \cdot N)$
- Papildoma dalis: Algoritmo pabaigoje vykdomas „TwoOpt“ segmentų optimizavimas prideda $O(N^2)$, sudėtingumą, tačiau jis vykdomas tik vieną kartą geriausiam individui.

Kadangi algoritmas turi griežtą laiko ribojimą (60 s), realus generacijų skaičius G yra kintamas ir priklauso nuo procesoriaus spartos.

3.2. Algoritmo abstraktus aprašas

```
ALGORITMAS GeneticAlgorithmSolve(vietovės, pradžia, P, elite, mutRate, G, laikas):
   1. Apskaičiuoja atstumų matricą D visoms vietovėms.
   2. Inicializuoja populiaciją (P):
     a. Vienas individas sukuriamas naudojant „Greedy“ algoritmą (seeding).
     b. Likę P-1 individai sukuriami atsitiktine tvarka.
   3. Kol nepasiekta generacijų riba G IR laikas neviršija 60s:
     a.Kiekvienam chromosomos maršrutui apskaičiuoja Fitness=Max(Route1_Cost, Route2_Cost).
     b. Surūšiuoja populiaciją didėjimo tvarka pagal Fitness.
     c. Išsaugo geriausią visų laikų sprendinį.
     d. Sukuria naują populiaciją:
       i. Nukopijuoja 'elite' geriausių individų tiesiogiai.
       ii. Likusią dalį užpildo kryžmindamas (OX Crossover) elito atstovus.
       iii. Taiko mutaciją (Swap mutate ir Split point shift).
   4. Geriausiam rastam individui taiko „TwoOpt“ lokalią paiešką abiems maršrutams atskirai.
  5. Grąžina galutinį optimizuotą sprendinį.
```

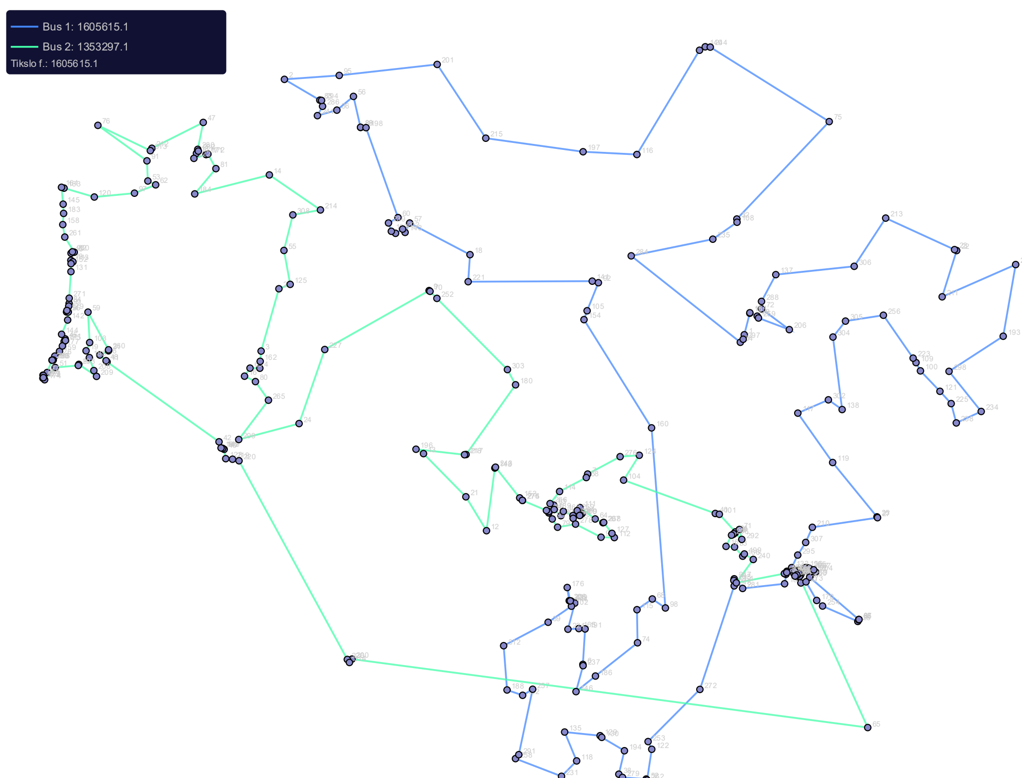
3.3. Rezultatų analizė

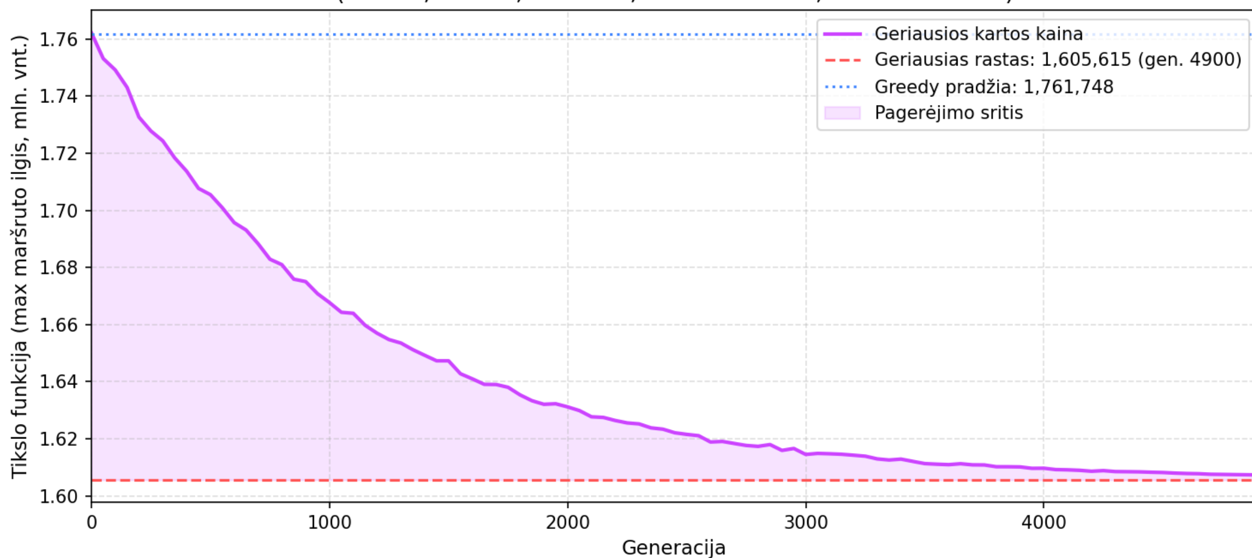
Atlikus optimizavimą su pilna duomenų imtimi (N = 314) ir 60 sekundžių laiko limitu, algoritmas pasiekė geriausią rezultatą 4900-ojoje generacijoje.

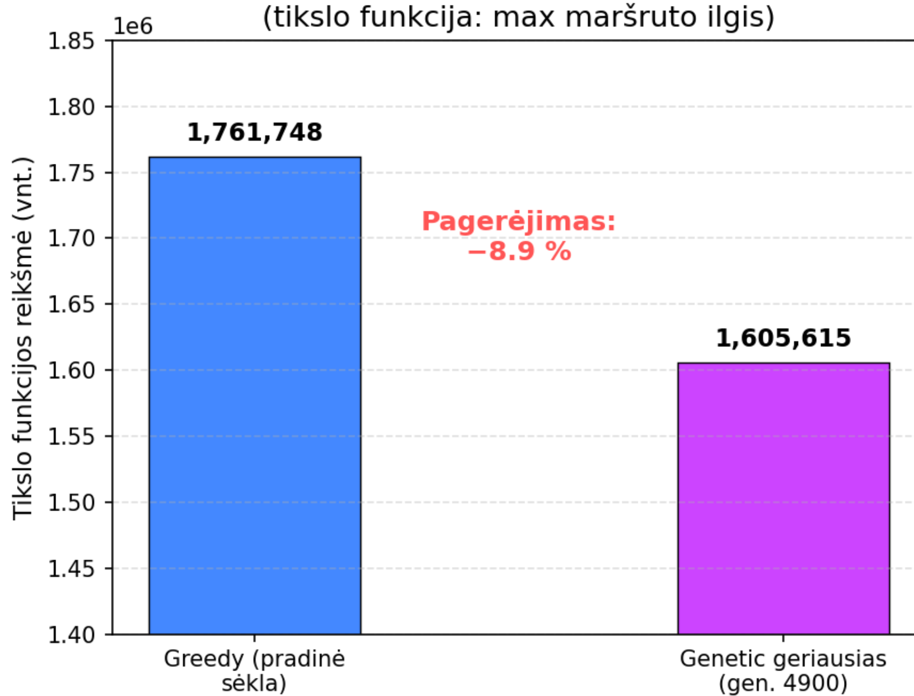
Maršrutas	Vietų skaičius	Atstumas
Bus 1	152	1605615
Bus 2	161	1353297

Genetinis algoritmas pagerino pradinį „Greedy“ sprendinį. Tikslo funkcijos reikšmė sumažėjo nuo 1761748 (Greedy) iki 1605615 (GA). Geresnis balansas: atstumų skirtumas tarp autobusų sumažėjo nuo 629 tūkst. iki 252 tūkst. vienetų.

3.4. Grafinis maršrutų atvaizdavimas




3 dalis – Genetinis algoritmas: tikslo funkcijos konvergencija
(n=314, P=200, elite=20, mutRate=15%, laiko riba=60 s)
Geriausios kartos kaina
Geriausias rastas: 1,605,615 (gen. 4900)
Greedy pradžia: 1,761,748
Pagerėjimo sritis
Tikslo funkcija (max maršruto ilgis, mln. vnt.)
1.76
1.74
1.72
1.70
1.68
1.66
1.64
1.62
1.60
0
1000
2000
3000
4000
Generacija


3 dalis – Greedy pradinis vs. Genetinis galutinis
(tikslo funkcija: max maršruto ilgis)
1e6
1.85
1.80
1.75
1.70
1.65
1.60
1.55
1.50
1.45
1.40
Tikslo funkcijos reikšmė (vnt.)
1,761,748
Pagerėjimas:
−8.9 %
1,605,615
Greedy (pradinė sėkla)
Genetic geriausias (gen. 4900)

4. Trečia dalis

Atlikti realizuotų algoritmų išlygiagretinimą. Paskaičiuoti jų išlygiagretinimo koeficientą, bei jį palyginti su praktiškai gaunamais rezultatais.

4.1. „Greedy“ algoritmo benchmark analizė

Atlikus nuoseklų ir lygiagretų „Greedy“ algoritmo vykdymą, gauti šie rezultatai:

- Nuoseklus laikas ($T1$): 9.2 ms
- Lygiagretus laikas (Tp): 74.7 ms
- Praktinis pagreitis (S): 0.12x
- Teorinis Amdahl limitas: 7.74x

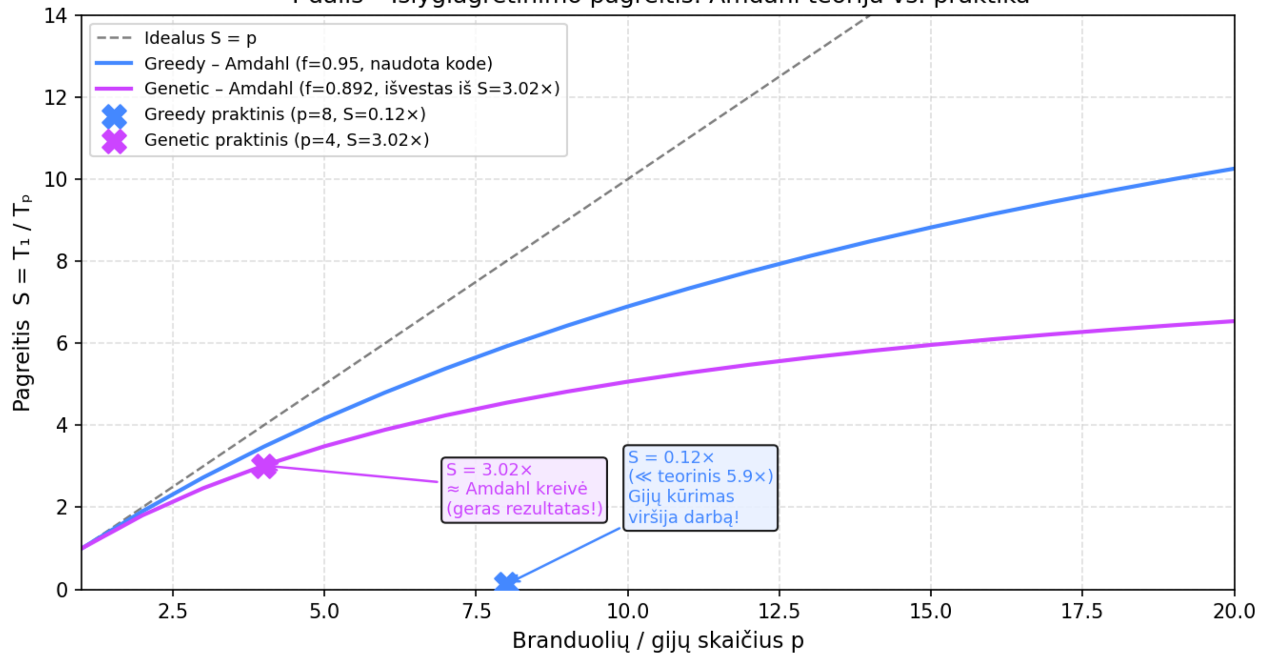
Praktinis pagreitis yra mažesnis už 1, o tai reiškia, kad lygiagretus vykdymas užtruko ilgiau nei nuoseklus. Taip nutiko todėl, kad „Greedy“ algoritmas yra per greitas (tik 9.2 ms). Sistemos sąnaudos, reikalingos gijų sukūrimui, valdymui ir sinchronizavimui per Parallel.For, viršijo paties skaičiavimo laiką. Tai įrodo, kad lygiagretinti apsimoka tik tas operacijas, kurios trunka bent keliasdešimt ar šimtus milisekundžių.

4.2. Lygiagretaus genetinio algoritmo analizė

Genetinis algoritmas buvo lygiagretinamas naudojant 4 gijas.

- Nuoseklus vykdymas (1 gija): 283 ms, kaina = 1 661 780,51
- Lygiagretus vykdymas (4 gijos): 375 ms, geriausia kaina = 1 761 748,95
- Skaičiuojamas pagreitis: 3.03x (teorinis limitas ≤4x)

Galime teigti, jog čia lygiagretinimas pasiteisino kur kas geriau. Pasiektas 3.03x pagreitis yra labai aukštas rezultatas naudojant 4 branduolius. Nedidelis nuokrypis nuo teorinio 4x pagreičio atsirado dėl procesoriaus resursų dalijimosi tarp gijų.


4 dalis – Išlygiagretinimo pagreitis: Amdahl teorija vs. praktika
Idealus S = p
Greedy – Amdahl (f=0.95, naudota kode)
Genetic – Amdahl (f=0.892, išvestas iš S=3.02×)
Greedy praktinis (p=8, S=0.12×)
Genetic praktinis (p=4, S=3.02×)
S = 3.02×
≈ Amdahl kreivė
(geras rezultatas!)
S = 0.12×
(≪ teorinis 5.9×)
Gijų kūrimas
viršija darbą!
Pagreitis S = T₁ / Tₚ
Branduolių / gijų skaičius p

5. Galutinis algoritmų palyginimas

Apibendrinant visus eksperimentus, gauti šie tikslo funkcijos (maksimalaus maršruto ilgio) rezultatai:

Metodas	Atstumas	Laikas	Pastebėjimai
Greedy	1761748,95	12ms	Greičiausias, bet neoptimalus.
B&B (20 vietų)	1088862,00	36ms	Tiksliausias, bet veikia tik su mažais duomenimis
Genetic	1605615,06	9444ms	Geriausias balansas tarp laiko ir kokybės.
Parallel Genetic	1761748,95	375ms	Greitas, bet šiuo atveju sustojo su prastesniu rezultatu.

Kodas:

```
 class Solution
{
    public List<Place> Route1;
    public List<Place> Route2;
    public double Cost;

    public Solution()
    {
        Route1 = new List<Place>();
        Route2 = new List<Place>();
        Cost = double.MaxValue;
    }

    public static double Dist(Place a, Place b)
    {
        double dx = a.X - b.X, dy = a.Y - b.Y;
        return Math.Sqrt(dx * dx + dy * dy);
    }

    public static double RouteCost(List<Place> r)
    {
        double c = 0;
        for (int i = 1; i < r.Count; i++)
            c += Dist(r[i - 1], r[i]);
        return c;
    }

    public void Print()
    {
        Console.WriteLine("  Bus 1 ({0} vietu, dist={1:F2}):", Route1.Count - 2, RouteCost(Route1));
        Console.WriteLine("    " + string.Join(" -> ", Route1.Select(p => p.Id.ToString()).ToArray()));
        Console.WriteLine("  Bus 2 ({0} vietu, dist={1:F2}):", Route2.Count - 2, RouteCost(Route2));
        Console.WriteLine("    " + string.Join(" -> ", Route2.Select(p => p.Id.ToString()).ToArray()));
        Console.WriteLine("  Tikslo f. (max route): {0:F2}", Cost);
    }
}

// PART 1 - GREEDY  O(n^2)
static class Greedy
{
    public static Solution Solve(List<Place> places, int startId)
    {
        Place start = null;
        foreach (Place p in places) if (p.Id == startId) { start = p; break; }

        List<Place> cities = places.Where(p => p.Id != startId).ToList();
```

```
            cities.Sort((a, b) =>
                Math.Atan2(a.Y - start.Y, a.X - start.X)
                    .CompareTo(Math.Atan2(b.Y - start.Y, b.X - start.X)));

            int half = cities.Count / 2;
            List<Place> cities1 = cities.Take(half).ToList();
            List<Place> cities2 = cities.Skip(half).ToList();

            List<Place> r1 = BuildRoute(start, cities1);
            List<Place> r2 = BuildRoute(start, cities2);

            Solution sol = new Solution();
            sol.Route1 = r1;
            sol.Route2 = r2;
            sol.Cost = Math.Max(Solution.RouteCost(r1), Solution.RouteCost(r2));
            return sol;
        }

        static List<Place> BuildRoute(Place start, List<Place> cities)
        {
            List<Place> route = new List<Place>();
            route.Add(start);
            List<Place> remaining = new List<Place>(cities);
            Place cur = start;
            while (remaining.Count > 0)
            {
                int bestIdx = 0;
                double bestDist = double.MaxValue;
                for (int i = 0; i < remaining.Count; i++)
                {
                    double d = Solution.Dist(cur, remaining[i]);
                    if (d < bestDist) { bestDist = d; bestIdx = i; }
                }
                cur = remaining[bestIdx];
                remaining.RemoveAt(bestIdx);
                route.Add(cur);
            }
            route.Add(start);
            return route;
        }

        public static List<Place> TwoOpt(List<Place> route)
        {
            bool improved = true;
            while (improved)
            {
                improved = false;
                for (int i = 1; i < route.Count - 2; i++)
                {
                    for (int j = i + 1; j < route.Count - 1; j++)
                    {
                        double d1 = Solution.Dist(route[i - 1], route[i]) + Solution.Dist(route[j], route[j +
1]);
                        double d2 = Solution.Dist(route[i - 1], route[j]) + Solution.Dist(route[i], route[j +
1]);
                        if (d2 < d1 - 1e-9)
                        {
                            route.Reverse(i, j - i + 1);
                            improved = true;
                        }
                    }
                }
            }
            return route;
        }
    }

    //BRANCH & BOUND  O(2^n) worst, pruned
    static class BranchAndBound
    {
        class State
        {
            public long Visited;
            public int[] Route1;
            public int[] Route2;
            public double Cost1;
            public double Cost2;
            public int Last1;
            public int Last2;

            public State Clone()
            {
                State s = new State();
                s.Visited = Visited;
                s.Route1 = (int[])Route1.Clone();
                s.Route2 = (int[])Route2.Clone();
                s.Cost1 = Cost1;
                s.Cost2 = Cost2;
                s.Last1 = Last1;
```

```
            s.Last2 = Last2;
            return s;
        }
    }

    static int[] Append(int[] arr, int val)
    {
        int[] next = new int[arr.Length + 1];
        arr.CopyTo(next, 0);
        next[arr.Length] = val;
        return next;
    }

    public static Solution Solve(List<Place> places, int startId, int timeLimitMs, int maxN)
    {
        Place start = null;
        foreach (Place p in places) if (p.Id == startId) { start = p; break; }

        List<Place> nodes = places.Where(p => p.Id != startId).Take(maxN).ToList();
        List<Place> allNodes = new List<Place>();
        allNodes.Add(start);
        allNodes.AddRange(nodes);
        int n = nodes.Count;

        double[,] D = new double[allNodes.Count, allNodes.Count];
        for (int i = 0; i < allNodes.Count; i++)
            for (int j = 0; j < allNodes.Count; j++)
                D[i, j] = Solution.Dist(allNodes[i], allNodes[j]);

        double bestCost = double.MaxValue;
        int[] best1 = null;
        int[] best2 = null;

        long deadline = Stopwatch.GetTimestamp() + (long)(timeLimitMs / 1000.0 * Stopwatch.Frequency);
        long allVisited = (1L << n) - 1;

        Stack<State> stack = new Stack<State>();
        State init = new State();
        init.Visited = 0;
        init.Route1 = new int[] { 0 };
        init.Route2 = new int[] { 0 };
        init.Cost1 = 0;
        init.Cost2 = 0;
        init.Last1 = 0;
        init.Last2 = 0;
        stack.Push(init);

        while (stack.Count > 0 && Stopwatch.GetTimestamp() < deadline)
        {
            State s = stack.Pop();

            if (s.Visited == allVisited)
            {
                double c = Math.Max(s.Cost1 + D[s.Last1, 0], s.Cost2 + D[s.Last2, 0]);
                if (c < bestCost) { bestCost = c; best1 = s.Route1; best2 = s.Route2; }
                continue;
            }

            if (Math.Max(s.Cost1, s.Cost2) >= bestCost) continue;

            int nextNode = -1;
            for (int k = 0; k < n; k++)
                if ((s.Visited & (1L << k)) == 0) { nextNode = k; break; }

            int ni = nextNode + 1;

            double nc1 = s.Cost1 + D[s.Last1, ni];
            if (nc1 < bestCost)
            {
                State s1 = s.Clone();
                s1.Visited |= (1L << nextNode);
                s1.Route1 = Append(s1.Route1, ni);
                s1.Cost1 = nc1;
                s1.Last1 = ni;
                stack.Push(s1);
            }

            double nc2 = s.Cost2 + D[s.Last2, ni];
            if (nc2 < bestCost)
            {
                State s2 = s.Clone();
                s2.Visited |= (1L << nextNode);
                s2.Route2 = Append(s2.Route2, ni);
                s2.Cost2 = nc2;
                s2.Last2 = ni;
                stack.Push(s2);
            }
        }
```

```
            if (best1 == null)
            {
                Console.WriteLine("  B&B: laikas baigesi, grazinama Greedy.");
                return Greedy.Solve(places, startId);
            }

            List<Place> BuildRoute(int[] indices)
            {
                List<Place> r = new List<Place>();
                foreach (int i in indices) r.Add(allNodes[i]);
                r.Add(start);
                return r;
            }

            Solution sol = new Solution();
            sol.Route1 = BuildRoute(best1);
            sol.Route2 = BuildRoute(best2);
            sol.Cost = bestCost;
            return sol;
        }
    }

//GENETIC ALGORITHM  O(G * P * n)
    static class GeneticAlgorithm
    {
        class Chromosome
        {
            public int[] Perm;
            public int Split;
            public double Fitness;

            public Chromosome(int[] perm, int split)
            {
                Perm = perm;
                Split = split;
                Fitness = double.MaxValue;
            }
        }

        static Random _rng = new Random(42);

        static double Evaluate(Chromosome chr, double[,] D, int n, int startIdx)
        {
            double c1 = 0, c2 = 0;
            int prev = startIdx;
            for (int i = 0; i < chr.Split; i++) { c1 += D[prev, chr.Perm[i]]; prev = chr.Perm[i]; }
            c1 += D[prev, startIdx];

            prev = startIdx;
            for (int i = chr.Split; i < n; i++) { c2 += D[prev, chr.Perm[i]]; prev = chr.Perm[i]; }
            c2 += D[prev, startIdx];

            return Math.Max(c1, c2);
        }

        static Chromosome OXCrossover(Chromosome a, Chromosome b, int n)
        {
            int p1 = _rng.Next(n), p2 = _rng.Next(n);
            if (p1 > p2) { int t = p1; p1 = p2; p2 = t; }

            int[] child = new int[n];
            for (int i = 0; i < n; i++) child[i] = -1;

            HashSet<int> used = new HashSet<int>();
            for (int i = p1; i <= p2; i++) { child[i] = a.Perm[i]; used.Add(a.Perm[i]); }

            int bi = 0;
            for (int i = 0; i < n; i++)
            {
                if (child[i] != -1) continue;
                while (used.Contains(b.Perm[bi])) bi++;
                child[i] = b.Perm[bi];
                used.Add(b.Perm[bi]);
                bi++;
            }

            int split = _rng.Next(2) == 0 ? a.Split : b.Split;
            split = Math.Max(1, Math.Min(n - 1, split));
            return new Chromosome(child, split);
        }

        static void Mutate(Chromosome chr, double mutRate, int n)
        {
            if (_rng.NextDouble() < mutRate)
            {
                int i = _rng.Next(n), j = _rng.Next(n);
                int t = chr.Perm[i]; chr.Perm[i] = chr.Perm[j]; chr.Perm[j] = t;
            }
            if (_rng.NextDouble() < mutRate)
```

```
        {
            int delta = _rng.Next(2) == 0 ? 1 : -1;
            chr.Split = Math.Max(1, Math.Min(n - 1, chr.Split + delta));
        }
    }

    static void TwoOptSegment(int[] perm, int lo, int hi, double[,] D, int startIdx)
    {
        if (hi - lo < 2) return;
        bool improved = true;
        while (improved)
        {
            improved = false;
            for (int i = lo; i <= hi - 1; i++)
            {
                int a = (i == lo) ? startIdx : perm[i - 1];
                for (int j = i + 1; j <= hi; j++)
                {
                    int b = (j == hi) ? startIdx : perm[j + 1];
                    double d1 = D[a, perm[i]] + D[perm[j], b];
                    double d2 = D[a, perm[j]] + D[perm[i], b];
                    if (d2 < d1 - 1e-9)
                    {
                        Array.Reverse(perm, i, j - i + 1);
                        improved = true;
                    }
                }
            }
        }
    }

    public static Solution Solve(List<Place> places, int startId,
        int popSize, int eliteCount, double mutRate, int maxGen, int timeLimitMs)
    {
        Place start = null;
        foreach (Place p in places) if (p.Id == startId) { start = p; break; }

        Place[] cities = places.Where(p => p.Id != startId).ToArray();
        int n = cities.Length;
        int startIdx = n;

        double[,] D = new double[n + 1, n + 1];
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                D[i, j] = Solution.Dist(cities[i], cities[j]);
        for (int i = 0; i < n; i++) { D[i, n] = Solution.Dist(cities[i], start); D[n, i] = D[i, n]; }

        int[] baseArr = new int[n];
        for (int i = 0; i < n; i++) baseArr[i] = i;

        List<Chromosome> pop = new List<Chromosome>(popSize);

        Solution gs = Greedy.Solve(places, startId);
        List<int> c1ids = new List<int>();
        List<int> c2ids = new List<int>();
        for (int i = 1; i < gs.Route1.Count - 1; i++)
        {
            int idx = Array.FindIndex(cities, c => c.Id == gs.Route1[i].Id);
            if (idx >= 0) c1ids.Add(idx);
        }
        for (int i = 1; i < gs.Route2.Count - 1; i++)
        {
            int idx = Array.FindIndex(cities, c => c.Id == gs.Route2[i].Id);
            if (idx >= 0) c2ids.Add(idx);
        }
        int[] seedPerm = c1ids.Concat(c2ids).ToArray();
        if (seedPerm.Length == n)
            pop.Add(new Chromosome(seedPerm, c1ids.Count));

        while (pop.Count < popSize)
        {
            int[] perm = baseArr.OrderBy(_ => _rng.Next()).ToArray();
            pop.Add(new Chromosome(perm, _rng.Next(1, n)));
        }

        Chromosome bestEver = null;
        double bestFit = double.MaxValue;
        long deadline = Stopwatch.GetTimestamp() + (long)(timeLimitMs / 1000.0 * Stopwatch.Frequency);
        int gen = 0;

        while (gen < maxGen && Stopwatch.GetTimestamp() < deadline)
        {
            foreach (Chromosome chr in pop)
                chr.Fitness = Evaluate(chr, D, n, startIdx);

            pop.Sort((a, b) => a.Fitness.CompareTo(b.Fitness));

            if (pop[0].Fitness < bestFit)
            {
```

```
                    bestFit = pop[0].Fitness;
                    bestEver = new Chromosome((int[])pop[0].Perm.Clone(), pop[0].Split);
                    bestEver.Fitness = bestFit;
                }

                if (gen % 100 == 0)
                    Console.Write("\r  Gen {0}/{1} | Best: {2:F2}        ", gen, maxGen, bestFit);

                List<Chromosome> newPop = new List<Chromosome>(popSize);
                for (int i = 0; i < eliteCount; i++) newPop.Add(pop[i]);

                while (newPop.Count < popSize)
                {
                    Chromosome p1 = pop[_rng.Next(eliteCount)];
                    Chromosome p2 = pop[_rng.Next(Math.Min(eliteCount * 3, popSize))];
                    Chromosome child = OXCrossover(p1, p2, n);
                    Mutate(child, mutRate, n);
                    newPop.Add(child);
                }

                pop = newPop;
                gen++;
            }

            Console.WriteLine();
            if (bestEver == null) bestEver = pop[0];

            TwoOptSegment(bestEver.Perm, 0, bestEver.Split - 1, D, startIdx);
            TwoOptSegment(bestEver.Perm, bestEver.Split, n - 1, D, startIdx);

            List<Place> r1 = new List<Place> { start };
            for (int i = 0; i < bestEver.Split; i++) r1.Add(cities[bestEver.Perm[i]]);
            r1.Add(start);

            List<Place> r2 = new List<Place> { start };
            for (int i = bestEver.Split; i < n; i++) r2.Add(cities[bestEver.Perm[i]]);
            r2.Add(start);

            Solution sol = new Solution();
            sol.Route1 = r1;
            sol.Route2 = r2;
            sol.Cost = Math.Max(Solution.RouteCost(r1), Solution.RouteCost(r2));
            return sol;
        }
    }

// PARALLELISM  S = T1/Tp
    static class ParallelSolver
    {
        public static void RunGreedyBenchmark(List<Place> places, int startId)
        {
            int workerCount = Environment.ProcessorCount;
            Console.WriteLine("  Processoriu: {0}", workerCount);

            Stopwatch sw = Stopwatch.StartNew();
            for (int i = 0; i < workerCount; i++)
                Greedy.Solve(places, startId);
            sw.Stop();
            double seqMs = sw.Elapsed.TotalMilliseconds;

            Solution[] results = new Solution[workerCount];
            Stopwatch sw2 = Stopwatch.StartNew();
            Parallel.For(0, workerCount, new ParallelOptions { MaxDegreeOfParallelism = workerCount }, i =>
            {
                results[i] = Greedy.Solve(places, startId);
            });
            sw2.Stop();
            double parMs = sw2.Elapsed.TotalMilliseconds;

            double speedup = seqMs / parMs;
            double theoretical = 1.0 / (0.05 + 0.95 / workerCount);
            Console.WriteLine("  Nuoseklusis laikas  T1 = {0:F1} ms", seqMs);
            Console.WriteLine("  Lygiagretusis laikas Tp = {0:F1} ms", parMs);
            Console.WriteLine("  Praktinis S = T1/Tp = {0:F2}x", speedup);
            Console.WriteLine("  Teorinis Amdahl (f=0.95, p={0}): S <= {1:F2}x", workerCount, theoretical);
        }

        public static Solution RunParallelGenetic(List<Place> places, int startId, int timeLimitMs)
        {
            int workers = Math.Min(4, Environment.ProcessorCount);

            Stopwatch sw = Stopwatch.StartNew();
            Solution seqSol = GeneticAlgorithm.Solve(places, startId, 100, 10, 0.15, 300, timeLimitMs /
workers);
            sw.Stop();
            double seqMs = sw.Elapsed.TotalMilliseconds;
            Console.WriteLine("\n  [SEQ 1 worker] {0:F0}ms, cost={1:F2}", seqMs, seqSol.Cost);

            Solution[] pgResults = new Solution[workers];
```

```
            Stopwatch sw2 = Stopwatch.StartNew();
            Parallel.For(0, workers, new ParallelOptions { MaxDegreeOfParallelism = workers }, i =>
            {
                pgResults[i] = GeneticAlgorithm.Solve(places, startId, 100, 10,
                    0.10 + i * 0.05, 300, timeLimitMs);
            });
            sw2.Stop();
            double parMs = sw2.Elapsed.TotalMilliseconds;

            Solution best = pgResults.OrderBy(s => s.Cost).First();
            double speedup = (seqMs * workers) / parMs;
            Console.WriteLine("  [PAR {0} workers] {1:F0}ms, best cost={2:F2}", workers, parMs, best.Cost);
            Console.WriteLine("  Pagreitis S = {0:F2}x (teorinis <= {1}x)", speedup, workers);
            return best;
        }
    }

class Program
    {
        static void Main(string[] args)
        {
            Console.OutputEncoding = Encoding.UTF8;
            List<Place> places = Data.GetPlaces();
            Console.WriteLine("Ikelta {0} vietu.", places.Count);

            int startId = 67;
            if (args.Length > 0 && int.TryParse(args[0], out int sid))
                startId = sid;

            Place startPlace = null;
            foreach (Place p in places) if (p.Id == startId) { startPlace = p; break; }

            if (startPlace == null) { Console.WriteLine("Vieta {0} nerasta.", startId); return; }
            Console.WriteLine("Pradzia: {0} - {1}\n", startPlace.Id, startPlace.Name);

            string outDir = AppDomain.CurrentDomain.BaseDirectory;

            //   1. GREEDY
            Console.WriteLine("=== 1 DALIS: GREEDY (Nearest Neighbor + 2-opt) ===");
            Stopwatch sw = Stopwatch.StartNew();
            Solution greedySol = Greedy.Solve(places, startId);
            greedySol.Route1 = Greedy.TwoOpt(greedySol.Route1);
            greedySol.Route2 = Greedy.TwoOpt(greedySol.Route2);
            greedySol.Cost = Math.Max(Solution.RouteCost(greedySol.Route1),
Solution.RouteCost(greedySol.Route2));
            sw.Stop();
            Console.WriteLine("Laikas: {0} ms", sw.ElapsedMilliseconds);
            greedySol.Print();
            SvgExport.Export(places, greedySol, startId, Path.Combine(outDir, "greedy.svg"));
            Console.WriteLine("  -> greedy.svg\n");

            //   2. BRANCH & BOUND
            Console.WriteLine("=== 2 DALIS: BRANCH & BOUND (pirmi 20 mazgai, 10s riba) ===");
            sw.Restart();
            Solution bnbSol = BranchAndBound.Solve(places, startId, 10000, 20);
            sw.Stop();
            Console.WriteLine("Laikas: {0} ms", sw.ElapsedMilliseconds);
            bnbSol.Print();
            SvgExport.Export(places, bnbSol, startId, Path.Combine(outDir, "bnb.svg"));
            Console.WriteLine("  -> bnb.svg\n");

            //   3. GENETIC
            Console.WriteLine("=== 3 DALIS: GENETINIS ALGORITMAS (pop=200, elite=20, mut=15%, 60s) ===");
            sw.Restart();
            Solution gaSol = GeneticAlgorithm.Solve(places, startId,
                200, 20, 0.15, 5000, 60000);
            sw.Stop();
            Console.WriteLine("Laikas: {0} ms", sw.ElapsedMilliseconds);
            gaSol.Print();
            SvgExport.Export(places, gaSol, startId, Path.Combine(outDir, "genetic.svg"));
            Console.WriteLine("  -> genetic.svg\n");

            //   4. PARALLEL
            Console.WriteLine("=== 4 DALIS: LYGIAGRETINIMAS ===");
            ParallelSolver.RunGreedyBenchmark(places, startId);
            Console.WriteLine();
            Console.WriteLine("  Lygiagretus genetinis:");
            Solution pgBest = ParallelSolver.RunParallelGenetic(places, startId, 20000);
            SvgExport.Export(places, pgBest, startId, Path.Combine(outDir, "parallel_genetic.svg"));
            Console.WriteLine("  -> parallel_genetic.svg\n");
        }
    }
}
```