VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS INSTITUTAS
INFORMATIKOS KATEDRA

Kursinis projektas

Rikiavimo tobulinimas genetiniais algoritmais

(Improving sorting with genetic algorithms)

Atliko: 4 kurso 2 grupės studentas
Deividas Zaleskis (parašas)

Darbo vadovas:
Irmantas Radavičius (parašas)

Vilnius
2022

Turinys

Įvadas

Duomenų rikiavimas yra vienas pamatinių informatikos uždavinių. Matematiškai jis formuluojamas taip: duotai baigtinei palyginamų elementų sekai $S = (s_1, s_2, ..., s_n)$ pateikti tokį kėlinį, kad duotosios sekos elementai būtų išdėstyti monotonine (didėjančia arba mažėjančia) tvarka [RB13]. Efektyvus šio uždavinio sprendimas buvo svarbus, kai informatikos mokslo sąvoka dar neegzistavo - skaitmeninio rikiavimo algoritmas (angl. radix sort) buvo naudojamas perforuotų kortelių rikiavimui jau 1923 metais. Atsiradus kompiuteriams, rikiavimo uždavinys tapo dar aktualesnis ir buvo laikomas vienu pagrindinių diskrečių uždavinių, kuriuos turėtų gebėti spręsti kompiuteris [Knu70]. Informatikos mokslui vystantis, rikiavimas tapo viena iš intensyviausiai tiriamų sričių, o rikiavimo algoritmai dabar sutinkami beveik kiekvienoje informatikos studijų programoje. Efektyvus rikiavimo uždavinio sprendimas dažnai padeda pagrindą efektyviam kito uždavinio sprendimui, pavyzdžiui, atliekant paiešką sąraše, naivią paiešką tikrinant visus elementus iš eilės galima pakeisti dvejetaine paieška, jei sąrašas yra išrikiuotas ir taip sumažinti laiko sudėtingumą iš $O(n)$ į $O(log\, n)$. Nepaisant to, jog šis uždavinys yra nagrinėjamas nuo pat informatikos mokslo pradžios, nauji rikiavimo algoritmai ir įvairūs patobulinimai egzistuojantiems algoritmams yra kuriami ir dabar.

Rikiavimo uždaviniui spręsti egzistuoja labai įvairių algoritmų. Dažniausiai jie yra klasifikuojami pagal šiuos kriterijus: rėmimąsi palyginimu (palyginimu paremti algoritmai gauna informaciją apie duomenis tik remdamiesi palyginimo operacijomis), laiko sudėtingumą (optimalūs palyginimu paremti algoritmai blogiausiu atveju turi $O(n\, log\, n)$ laiko sudėtingumą), atminties sudėtingumą (optimaliu atveju - $O(1)$), stabilumą (stabilūs algoritmai nekeičia lygių elementų tvarkos). Tiesa, praktikoje labiausiai paplitę yra tik keli: rikiavimas sąlaja (angl. merge sort), rikiavimas įterpimu (angl. insertion sort) ir greitojo rikiavimo algoritmas (angl. quicksort). Visi jie turi savitų trūkūmų: rikiavimas sąlaja naudoja pakankamai daug papildomos atminties, rikiavimas įterpimu yra efektyvus tik kai elementų kiekis yra nedidelis, o greitojo rikiavimo algoritmas netinkamai parinkus slenkstį turi $O(n^2)$ sudėtingumą. Todėl pastaruoju metu plačiai naudojami hibridiniai rikiavimo algoritmai, kurie sujungia keletą klasikinių rikiavimo algoritmų į vieną ir panaudoja jų geriausias savybes. Nepaisant įvairovės ir naujų algoritmų gausos, klasikiniai rikiavimo algoritmai ir toliau išlieka aktualūs.

Šelo rikiavimo algoritmas (angl. Shellsort, toliau - Šelo algoritmas) [She59] yra paremtas palyginimu, nenaudojantis papildomos atminties ir nestabilus. Šelo algoritmą galima laikyti rikiavimo įterpimu optimizacija, kuri lygina ne gretimus, o toliau vienas nuo kito esančius elementus, taip paspartindama jų perkėlimą į galutinę poziciją. Pagrindinė algoritmo idėja - išskaidyti rikiuojamą seką S į posekius $S_1, S_2, ..., S_n$, kur kiekvienas posekis $S_i = (s_i, s_{i+h}, s_{i+2h}, ...)$ yra sekos S elementai, kurių pozicija skiriasi h. Išrikiavus visus sekos S posekius S_i su tarpu h, seka tampa h-išrikiuota. Remiantis tuo, jog sekai S esant h-išrikiuota ir ją k-išrikiavus, ji lieka h-išrikiuota [GK72], galima kiekvieną algoritmo iteraciją mažinti tarpą, taip vis didinant sekos S išrikiuotumą. Įprastai paskutinėje iteracijoje atliekamas rikiavimas su tarpu 1, kas užtikrina jog bus atliekamas

rikiavimas įterpimu ir seka bus pilnai išrikiuota.

Šelo algoritmo idėjas taip pat galima pritaikyti ir naujų rikiavimo algoritmų kūrimui. Ko gero žinomiausias to pavyzdys yra Dobosiewicz pateiktas algoritmas [Dob+80], kuris pritaiko Šelo algoritmo idėjas burbuliuko rikiavimo algoritmui optimizuoti. Perspektyvūs Šelo algoritmo variantai yra kuriami ir šiais laikais - M. Goodrich 2014 m. paskelbė Šelo algoritmo variantą, kurio asimptotinis sudėtingumas yra $O(n\,log\,n)$ [Goo14]. Tiesa, Šelo algoritmo variantai literatūroje nėra taip išsamiai ištirti, kaip klasikinė versija. Tuo pačiu tai galime matyti ir kaip galimybę - galbūt egzistuoja Šelo algoritmo versija, veikianti du kartus greičiau nei klasikinė, kurios dar niekas neatrado. Tokio algoritmo radimas galėtų suteikti tiek praktinės, tiek teorinės naudos.

Vienas iš metodų, kuriuos galima taikyti naujų Šelo algoritmo variantų radimui, yra genetinis algoritmas. Genetinis algoritmas (GA) yra metodas rasti euristikas, paremtas biologijos žiniomis apie natūralios atrankos procesą. J.H. Holland, GA pradininkas, savo knygoje [Hol92] apibrėžė genetinio algoritmo sąvoką ir su ja glaudžiai susijusias chromosomų, bei rekombinacijos, atrankos ir mutacijos operatorių koncepcijas. Genetinių algoritmų veikimas yra pagrįstas pradinės chromosomų populiacijos evoliucija, kiekvienos naujos chromosomų kartos gavimui naudojant rekombinacijos, atrankos ir mutacijos operatorius. Genetiniai algoritmai taikomi sprendžiant įvairius paieškos ir optimizavimo uždavinius, kuomet nesunku nustatyti, ar sprendinys tinkamas, tačiau tinkamo sprendinio radimas reikalauja daug resursų ar net pilno perrinkimo. Tokiu atveju apytikslio sprendinio radimas (euristika) gali būti daug patrauklesnis sprendimo būdas, kadangi tikslaus sprendinio radimas dažnai yra NP-sunkus uždavinys. Nesunku pastebėti, jog efektyvių Šelo algoritmo variantų radimas yra sunkus uždavinys atliekamų skaičiavimų prasme, tikėtinai reikalaujantis pilno potencialių sprendinių perrinkimo, tad šio uždavinio sprendimui taikyti GA atrodo prasminga. Kiek žinoma autoriui, genetiniai algoritmai kol kas nebuvo taikyti tokio tipo uždaviniui spręsti.

Darbo tikslas: pritaikyti genetinius algoritmus Šelo algoritmo variantų generavimui.

Darbo uždaviniai:

- Atlikti Šelo algoritmo ir jo variantų literatūros analizę.
- Nustatyti kriterijus Šelo algoritmo variantų efektyvumui įvertinti.
- Paruošti aplinką Šelo algoritmo variantų generavimui.
- Pasitelkiant genetinius algoritmus sugeneruoti Šelo algoritmo variantus.
- Paruošti aplinką Šelo algoritmo variantų efektyvumo tyrimui.
- Atliekant eksperimentus įvertinti sugeneruotų ir pateiktų literatūroje Šelo algoritmo variantų efektyvumą.

Šis darbas sudarytas iš 7 skyrių. Pirmame skyriuje atliekama Šelo algoritmo ir jo variantų literatūros analizė. Antrame skyriuje aptariami genetiniai algoritmai. Trečiame skyriuje nustatomi kriterijai Šelo algoritmo variantų efektyvumui įvertinti. Ketvirtame skyriuje paruošiama aplinka

Šelo algoritmo variantų generavimui. Penktame skyriuje pasitelkiant genetinius algoritmus generuojami Šelo algoritmo variantai. Šeštame skyriuje paruošiama aplinka Šelo algoritmo variantų efektyvumo tyrimui. Septintame skyriuje atliekant eksperimentus įvertinamas sugeneruotų ir pateiktų literatūroje Šelo algoritmo variantų efektyvumas.

1. Šelo algoritmas ir jo variantai

Šis skyrius sudarytas iš 2 poskyrių. Pirmame poskyryje nagrinėjamas Šelo algoritmas. Antrame poskyryje aptariami Šelo algoritmo variantai.

1.1. Šelo algoritmas

Vadovėlinis Šelo algoritmas [She59] (toliau - VŠA) yra vienas iš seniausių (D. L. Shell paskelbtas 1959 m.) ir geriausiai žinomų rikiavimo algoritmų. Taip pat jis yra ir vienas iš paprasčiausiai įgyvendinamų, ką galima pastebėti iš pseudokodo, pateikiamo 1 algoritme.

Algorithm 1 Vadovėlinis Šelo algoritmas

```
1:  foreach gap in H do
2:      for i ← gap to N − 1 do
3:          j ← i
4:          temp ← S[i]
5:          while j > gap and S[j − gap] > S[j] do
6:              S[j] ← S[j − gap]
7:              j ← j − gap
8:          end while
9:          S[j] ← temp
10:     end for
11: end for
```

Įdėmiau įsižiūrėjus, nesunku pastebėti Šelo algoritmo panašumą į paprastą rikiavimą įterpimu - pašalinus išorinį ciklą, iteruojantį tarpų sekos narius bei pakeitus kintamąjį gap į 1, gaunamas būtent šio algoritmo kodas. Didesnis įterpimo atstumas (tarpas) suteikia Šelo algoritmui pranašumą prieš rikiavimą įterpimu, kadangi labiausiai nuo galutinės pozicijos nutolę elementai gali greičiau patekti į reikiamą poziciją. Šelo algoritmo idėjos turi tvirtą matematinį pagrindą: rikiavimo įterpimu sudėtingumas yra $O(n^2)$ blogiausiu atveju (pavyzdžiui, kai rikiuojama seka yra išrikiuota atgaline tvarka), tačiau kai didžiausias atstumas tarp dviejų neišrikiuotų elementų yra k, rikiavimo įterpimu laiko sudėtingumas yra $O(kn)$. Tad seką h-išrikiavus, paskutinė iteracija su tarpu 1 turės $O(hn)$ laiko sudėtingumą. Iteracijos su didesniais tarpais taip pat yra pakankamai efektyvios, kadangi dirbama su mažais posekiais, o šiuo atveju rikiavimas įterpimu yra vienas tinkamiausių.

1.2. Šelo algoritmo variantai

Šelo algoritmas yra unikalus savo variantų gausa. Literatūroje Šelo algoritmo variantais vadinamos ir šio algoritmo implementacijos, kuriose naudojamos kitokios nei Šelo pasiūlytos tarpų

sekos, ir implementacijos, kurių kodas skiriasi nuo vadovėlinės versijos. Šiame darbe nagrinėsime tuos variantus, kurių kodas skiriasi nuo vadovėlinės versijos.

Kaip Šelo algoritmo varianto pavyzdį galime pateikti patobulintą Šelo algoritmą (toliau - PŠA) [RB13]. Šio algoritmo pseudokodas pateikiamas žemiau, 2 algoritme. Autorių teigimu [RB13], PŠA vidutiniškai atlieka 40-80% mažiau priskyrimų ir veikia 20% greičiau, nei VŠA. Šį skirtumą galima paaiškinti tuo, jog vykdant vidinį vadovėlinio Šelo algoritmo ciklą (1 algoritmo 5-8 eilutės), 5 eilutėje yra tikrinama, ar $S[j]$ jau yra tinkamoje pozicijoje. Jei $S[j]$ jau yra tinkamoje pozicijoje, vidinis ciklas nėra vykdomas ir jokių elementų pozicijos nėra keičiamos, tačiau 4 ir 9 eilutėse vis tiek yra veltui atliekami du priskyrimai. Tuo tarpu PŠA prieš vykdydamas bet kokius kitus žingsnius patikrina, ar elementas $S[j]$ jau yra tinkamoje pozicijoje (žr. 2 algoritmo 3 eil.) ir taip sumažina atliekamų priskyrimų skaičių.

Algorithm 2 Patobulintas Šelo algoritmas

```
1:  foreach gap in H do
2:      for i ← gap to N − 1 do
3:          if S[i − gap] > S[i] then
4:              j ← i
5:              temp ← S[i]
6:              repeat
7:                  S[j] ← S[j − gap]
8:                  j ← j − gap
9:              until j ≤ gap or S[j − gap] ≤ S[j]
10:             S[j] ← temp
11:         end if
12:     end for
13: end for
```

Nesunku pastebėti, jog VŠA ir PŠA struktūra yra tokia pati, ir algoritmai skiriasi tik tuo, kaip įgyvendinama elementų rikiavimo logika. Šelo algoritmo variantams būdingą struktūrą toliau vadinsime Šelo algoritmo karkasu, o karkaso viduje atliekamą rikiavimo logiką - perėjimu (angl. pass). VŠA taikomą perėjimą toliau vadinsime įterpimo perėjimu, o PŠA taikomą perėjimą - patobulintu įterpimo perėjimu. Šelo algoritmo karkaso apibrėžimas pateikiamas pseudokodu 3 algoritme.

Algorithm 3 Šelo algoritmo karkasas

```
1: foreach gap in H do
2:     perform pass with gap
3: end for
```

Dobosiewicz vienas pirmųjų pastebėjo, jog pasitelkiant Šelo algoritmo karkasą ir pakeitus ri-

kiavimo logiką (perėjimą) taip pat galima sukonstruoti pakankamai efektyvų algoritmą [Dob+80]. Dobosiewicz taikytas perėjimas yra labai panašus į burbuliuko rikiavimo algoritmo (angl. bubble sort) atliekamas operacijas: einama iš kairės į dešinę, palyginant ir (jei reikia) sukeičiant elementus vietomis. Todėl literatūroje šis perėjimas dažnai vadinamas burbuliuko perėjimu (angl. bubble pass) [Sed96]. Jo pseudokodas pateikiamas 4 algoritme.

Algorithm 4 Burbuliuko perėjimas

1: **for** $i \leftarrow 0$ **to** $N - gap - 1$ **do**
2: **if** $S[i] > S[i + gap]$ **then**
3: $swap(S[i], S[i + gap])$
4: **end if**
5: **end for**

Tiesa, burbuliuko metodą galima nežymiai patobulinti, suteikiant jam daugiau simetrijos ir atliekant perėjimą tiek iš kairės į dešinę, tiek iš dešinės į kairę. Tokiu būdu dešinėje esantys elementai greičiau pasieks savo galutinę poziciją. Šis metodas primena kokteilio purtymą, todėl literatūroje dažnai vadinamas kokteilio rikiavimu (angl. cocktail sort arba shaker sort). Šio algoritmo taikomą perėjimą, kurį toliau vadinsime supurtymo perėjimu (angl. shake pass), integravus į Šelo algoritmo karkasą taip pat gaunamas gana įdomus algoritmas [IS86]. Supurtymo perėjimo pseudokodas pateikiamas 5 algoritme.

Algorithm 5 Supurtymo perėjimas

1: **for** $i \leftarrow 0$ **to** $N - gap - 1$ **do**
2: **if** $S[i] > S[i + gap]$ **then**
3: $swap(S[i], S[i + gap])$
4: **end if**
5: **end for**
6: **for** $i \leftarrow N - gap - 1$ **to** 0 **do**
7: **if** $S[i] > S[i + gap]$ **then**
8: $swap(S[i], S[i + gap])$
9: **end if**
10: **end for**

Dar viena burbuliuko algoritmo modifikacija yra nelyginis-lyginis rikiavimas (angl. odd-even sort arba brick sort) [Hab72]. Šio algoritmo perėjimo idėja - išrikiuoti visas nelyginių/lyginių indeksų gretimų elementų poras, o tada atlikti tą patį visoms lyginių/nelyginių indeksų gretimų elementų poroms. Šį perėjimą, kurį toliau vadinsime mūrijimo perėjimu (angl. brick pass) [Sed96], nesunkiai galima pritaikyti ir Šelo algoritmo karkasui, kintamuoju pakeitus originaliame algoritme taikytą tarpą 1 [Lem94]. Mūrijimo perėjimo pseudokodas yra pateikiamas 6 algoritme.

Algorithm 6 Mūrijimo perėjimas

```
1: for i ← gap to N − gap − 1 step 2 ∗ gap do
2:     if S[i] > S[i + gap] then
3:         swap(S[i], S[i + gap])
4:     end if
5: end for
6: for i ← 0 to N − gap − 1 step 2 ∗ gap do
7:     if S[i] > S[i + gap] then
8:         swap(S[i], S[i + gap])
9:     end if
10: end for
```

2. Genetiniai algoritmai

Paprasčiausias genetinis algoritmas susideda iš chromosomų populiacijos bei atrankos, mutacijos ir rekombinacijos operatorių [SY99]. Šiame skyriuje bus nagrinėjamos šių terminų reikšmės ir genetinių algoritmų veikimo principai.

2.1. Chromosomų populiacija

Chromosoma GA kontekste vadiname potencialų uždavinio sprendinį. Projektuojant genetinį algoritmą tam tikro uždavinio sprendimui, svarbu tinkamai pasirinkti, kaip kompiuteriu modeliuoti galimus sprendinius. Įprastai siekiama sprendinio genus išreikšti kuo primityviau, siekiant palengvinti mutacijos ir rekombinacijos operatorių taikymą. Dažniausiai tai pasiekiama chromosomas išreiškiant bitų ar kitų primityvių duomenų tipų masyvais [Whi94]. Tada mutacija gali būti įgyvendinama tiesiog modifikuojant atsitiktinai pasirinktą masyvo elementą, o rekombinacijai pakanka remiantis tam tikra strategija perkopijuoti tėvinių chromosomų elementus į vaikinę chromosomą.

Sprendinio kokybę įvardijame kaip jo tinkamumą, kuris apibrėžiamas tinkamumo funkcijos reikšme, pateikus sprendinį arba tarpinį sprendinio kainos įvertį kaip parametrą. Tinkamumo funkcija yra viena svarbiausių genetinio algoritmo dalių, kadangi kai ji netinkamai parinkta, algoritmas nekonverguos į tinkamą sprendinį arba užtruks labai ilgai.

Chromosomų rinkinys, literatūroje dažnai vadinamas populiacija, atspindi uždavinio sprendinių aibę, kuri kinta kiekvieną genetinio algoritmo iteraciją. Populiaciją dažnu atveju sudaro šimtai ar net tūkstančiai individų. Populiacijos dydis dažnai priklauso nuo sprendžiamo uždavinio, tačiau literatūroje nėra konsensuso, kokį populiacijos dydį rinktis bendru atveju.

2.2. Genetiniai operatoriai

Esminė GA dalis yra populiacijos genetinės įvairovės užtikrinimas, geriausių individų atranka ir kryžminimasis. Siekiant užtikrinti šių procesų išpildymą, genetinis algoritmas vykdymo metu iteratyviai atnaujina esamą populiaciją ir kuria naujas kartas taikydamas biologijos žiniomis paremtus atrankos, rekombinacijos ir mutacijos operatorius.

Atrankos operatorius grąžina tinkamiausius populiacijos individus, kuriems yra leidžiama susilaukti palikuonių taikant rekombinacijos operatorių. Dažniausiai atranka vykdoma atsižvelgiant į populiacijos individų tinkamumą, atrenkant ir pateikiant rekombinacijai tuos, kurių tinkamumas yra geriausias. Verta pastebėti, jog įprastai rekombinacijai yra pasirenkama tam tikra fiksuota einamosios populiacijos dalis ir daugelyje GA implementacijų šis dydis yra nurodomas kaip veikimo parametras.

Rekombinacijos operatorius įprastai veikia iš dviejų tėvinių chromosomų sukurdamas naują vaikinę chromosomą, kas dažniausiai pasiekiama tam tikru būdu perkopijuojant tėvų genų atkarpas į vaikinę chromosomą. Rekombinacijos strategijų yra įvairių, tačiau tinkamiausią strategiją galima pasirinkti tik atsižvelgiant į sprendžiamą uždavinį.

Mutacijos operatorius veikia modifikuojant pasirinktos chromosomos vieną ar kelis genus, kas dažniausiai įgyvendinama nežymiai pakeičiant pasirinktų genų reikšmes ar sukeičiant jas vietomis. Įprastai mutacija kiekvienai chromosomai taikoma su tam tikra tikimybe, kuri nurodoma kaip vienas iš GA veikimo parametrų. Tinkamas chromosomos mutacijos tikimybės parinkimas yra vienas iš svarbiausių sprendimų projektuojant GA, kadangi nuo mutacijos tikimybės dažnu atveju priklauso gaunamų sprendinių kokybė. Jei mutacijos tikimybė yra per didelė, GA išsigimsta į primityvią atsitiktinę paiešką [HAA+19] ir rizikuojama prarasti geriausius sprendinius. Jei mutacijos tikimybė per maža, tai gali vesti prie genetinio dreifo [Mas11], kas reiškia, jog populiacijos genetinė įvairovė palaipsniui mažės.

3. Šelo algoritmo variantų efektyvumo kriterijai

Rikiavimo algoritmai dažniausiai yra vertinami pagal atliekamų priskyrimų skaičių, kadangi daugelyje algoritmų priskyrimų skaičius uždaviniui augant greitai artėja prie palyginimų skaičiaus ir tokiu metodu gautas įvertis būna pakankamai tikslus. Vadovėlinio Šelo algoritmo atveju, atliekamų palyginimų ir priskyrimų skaičiaus santykis augant N nebūtinai artėja prie 1 [RB13]. Kaip parodo [Ciu01], vadovėliniame Šelo algoritme svarbiausia operacija yra palyginimas. Literatūroje sudėtinga rasti panašios informacijos apie Šelo algoritmo variantus, todėl laikysime, jog tinkamam efektyvumo įvertinimui būtina matuoti ir atliekamus palyginimus, ir atliekamus priskyrimus.

Algoritmo veikimo laikas taip pat gali duoti tinkamų įžvalgų įvertinant praktinį efektyvumą. Savaime suprantama, jog į veikimo laiką verta žvelgti kritiškai, kadangi jis priklauso nuo konkrečios algoritmo implementacijos, eksperimentams naudojamos mašinos architektūros ir techninių parametrų, operacinės sistemos, pasirinktos programavimo kalbos ir net kompiliatoriaus versijos. Reikia pastebėti, jog šiuolaikinių kompiuterių architektūros yra labai sudėtingos, kadangi gamintojai siekia pilnai išnaudoti mašinos galimybes. Šiam tikslui pasiekti yra pasitelkiamos įvarios strategijos: instrukcijos nėra vykdomos iš eilės (siekiant pilnai išnaudoti procesoriaus ciklus), duomenys saugomi kelių lygių talpykloje (siekiant panaikinti atminties delsą), šakos yra nuspėjamos (siekiant išlygiagretinti instrukcijų vykdymą). Todėl naudojant modernų kompiuterį labai sunku iš anksto nustatyti, kaip algoritmas veiks praktikoje. Remiantis kursiniame darbe gautais rezultatais taip pat galime teigti, jog Šelo algoritmo atliekamų operacijų skaičius ir veikimo laikas ne visada koreliuoja - tyrime geriausius veikimo laiko rezultatus pateikę Šelo algoritmo variantai atliko ženkliai daugiau operacijų, nei atliekamų operacijų operacijų atžvilgiu optimalūs variantai. Atsižvelgiant į aukščiau pateiktus argumentus, manome jog algoritmo veikimo laiko įvertis yra pakankamai geras įrankis juodos dėžės principu įvertinti praktinį algoritmo efektyvumą.

Remiantis aukščiau pateiktais argumentais, šiame darbe Šelo algoritmo variantai vertinami pagal atliekamų palyginimų skaičių, atliekamų priskyrimų skaičių ir veikimo laiką. Kadangi Šelo algoritmo variantai nėra taip išsamiai ištirti [Bre01], šio darbo kontekste sudėtinga pagrįstai nustatyti, kuri iš operacijų turėtų turėti didesnį svorį. Remdamiesi tuo, jog variantai veikia panašiu principu į vadovėlinę versiją, laikysime, jog didesnis svoris turėtų būti suteikiamas atliekamiems palyginimams. Veikimo laikui teiksime mažesnį svorį nei atliekamoms operacijoms, kadangi gauti veikimo laiko įverčiai gali priklausyti nuo išorinių veiksnių, dėl ko gauti rezultatai nebus tokie patikimi ir juos gali būti sudėtinga atkartoti.

4. Šelo algoritmo variantų generavimo aplinkos paruošimas

Šelo algoritmo variantus generuojantis genetinis algoritmas buvo įgyvendintas C++ programavimo kalba, pasitelkiant openGA biblioteką [MAM+17]. C++ buvo pasirinkta dėl praeitos patirties, galimybės kontroliuoti kodo našumą (ko dažnai nesuteikia aukštesnio lygio programavimo kalbos) ir tam tikrų kalbos aspektų palengvinančių projekto įgyvendinimą, tokių kaip operatorių perkrovimas. OpenGA biblioteka buvo pasirinkta dėl modernių C++ kalbos konstruktų ir lygiagretaus vykdymo palaikymo, išsamios dokumentacijos ir praeitos patirties (buvo naudojama ruošiant kursinį darbą).

Siekiant išmatuoti skirtingų Šelo algoritmo variantų atliekamų operacijų skaičių buvo suprojektuota bendrinė klasė (angl. generic class) Element. Klasė Element veikia kaip apvalkalas pasirinkto tipo duomenims, suteikdama galimybę automatiškai skaičiuoti atliekamus palyginimus ir priskyrimus. Šis funkcionalumas buvo įgyvendintas perkraunant Element klasės palyginimo ir priskyrimo operatorius - kviečiant kažkurį iš šių operatorių atitinkamai padidinamas atliktų palyginimų ar priskyrimų skaičius.

Atsižvelgiant į tai, jog dalis Šelo algoritmo variantų yra tikimybiniai (angl. probabilistic) ir pilnai išrikiuoja duomenis tik su tam tikra tikimybe, buvo pasirinkta skaičiuoti duomenų inversijas atlikus rikiavimą ir jas įtraukti į individų tinkamumo vertinimą. Atrodytų pakankamai intuityvu pasirinkti inversijas skaičiuoti burbuliuko metodu, kuris nėra labai efektyvus (sudėtingumas blogiausiu atveju $O(n^2)$), tačiau yra lengvai suprantamas ir gana paprastas įgyvendinti. Tačiau tarkime, jog GA bus vykdomas su 100 individų populiacija kai N yra 10000. Tada vienos kartos inversijų skaičiavimui blogiausiu atveju reikės apytiksliai $10000^2 * 100 = 10^{10}$ operacijų, kas yra visiškai nepriimtina žinant, jog GA kartų skaičius dažnai siekia šimtus ar net tūkstančius (priklausomai nuo implementacijos ir pasirinktų pabaigos sąlygų). Todėl buvo nuspręsta inversijų skaičiavimui pasitelkti modifikuotą rikiavimo sąlaja algoritmą, kurio įgyvendinimas yra kiek sunkesnis, tačiau sudėtingumas blogiausiu atveju siekia $O(n\,log\,n)$. Šio inversijų skaičiavimo metodo pseudokodas pateikiamas prieduose (7 ir 8 algoritmuose).

5. Šelo algoritmo variantų generavimas

Šis skyrius sudarytas iš 3 poskyrių. Pirmame poskyryje aptariamas Šelo algoritmo variantų generavimui suprojektuotas genetinis algoritmas. Antrame poskyryje generuojami Šelo algoritmo variantai, kai $N = 1000$. Trečiame poskyryje generuojami Šelo algoritmo variantai, kai $N = 100000$.

5.1. Genetinis algoritmas

Prieš pradedant generuoti Šelo algoritmo variantus, reikėtų aptarti kaip turėtų atrodyti sprendinys, atspindintis tam tikrą Šelo algoritmo variantą. Laikysime jog vieną varianto perėjimą sudaro pora $(type, gap)$, kur $type$ yra skaičius, atitinkantis vieną iš anksčiau darbe aptartų perėjimų tipų, o gap - tarpas, su kuriuo rikiuojama tame perėjime. Tadą Šelo algoritmo variantą galime modeliuoti sąrašu tokių perėjimų. Toliau darbe modeliuojamą Šelo algoritmo varianto sprendinį taip pat vadinsime chromosoma arba individu, o vieną jo perėjimą - genu. Kadangi genetinis algoritmas sprendinių evoliuciją vykdo taikant mutacijos ir rekombinacijos operatorius, siekėme, jog šių operatorių taikymas sprendiniams būtų kuo mažiau sudėtingas. Todėl sprendinio duomenų modelis yra pakankamai paprastas, kas taip pat palengvina serializaciją ir saugojimą. Tiesa, toks modelis neturi jokio funkcionalumo (juo duomenų rikiuoti negalime), tačiau tai nėra sunku išspręsti: pakanka kiekvienam perėjimo tipui paruošti atitinkamą funkciją, kuri kaip parametrus priima rikiuojamus duomenis ir tarpą su kuriuo rikiuojama. Tada tokį algoritmą galima vykdyti iteruojant jo perėjimų sąrašą ir kiekvienam perėjimui iškviečiant funkciją atitinkančią jo tipą.

Šelo algoritmo variantų generavimui buvo pasitelktas vienkriterinis genetinis algoritmas, kadangi tai supaprastina tiek paties GA vykdymą, tiek realizaciją. Projektuojant genetinį algoritmą buvo pasirinkta minimizuoti sprendinių atliekamų operacijų skaičių, tuo pačiu taikant baudas pagal pasirinktus kriterijus.

Kadangi openGA biblioteka nereikalauja įgyvendinti individų atrankos operatoriaus, lieka apibrėžti tik mutacijos ir rekombinacijos operatorius. Individų mutacija buvo įgyvendinta pasirenkant atsitiktinį sprendinio perėjimą ir pakeičiant jo tipą kuriuo nors kitu perėjimo tipu. Tarpų mutacija nebuvo vykdoma, kadangi šiame darbe tarpų sekos nėra pagrindinis tyrimo objektas. Tuo pačiu tai sumažina galimų variantų aibę, kas leidžia lengviau gauti kokybiškus rezultatus ir paspartina genetinio algoritmo vykdymą. Sprendinių rekombinacija buvo vykdoma tolygia strategija, kur dviejų tėvų genai turi vienodą tikimybę būti perduoti vaikinei chromosomai. Siekiant išvengti netinkamų sprendinių, rekombinacijos operatoriumi gauto naujo sprendinio perėjimai buvo išrikiuojami pagal tarpus. Reikia pastebėti, jog pasirinkta rekombinacijos operatoriaus implementacija leidžia tarpų sekų maišymąsi. Autoriaus nuomone, tai nėra blogas dalykas - netiesioginis šio darbo rezultatas gali būti ir nauja tarpų seka, gauta maišant individų inicializacijai naudotas tarpų sekas.

Atlikus keletą eksperimentų ir preliminariai įvertinus gautų sprendinių tinkamumą buvo pasirinkta GA taikyti tokius parametrus: populiacija - 200, mutacijos tikimybė - 0.1, rekombinuojama

populiacijos dalis - 0.2. Siekiant išlaikyti tinkamiausius sprendinius, taip pat buvo pasitelktas elitizmas ir 5 didžiausią tinkamumą turinčios chromosomos nekeistos patekdavo į kitą GA iteraciją. GA sustojimo kriterijumi buvo laikomas geriausio individo nepakitimas 100 iteracijų.

Abiejuose Šelo algoritmo variantų generavimo etapuose buvo naudojama ta pati tinkamumo funkcija, kurios apibrėžimas yra toks:
$f(c) = (c.inversions)^2 + (c.time)^{1.5} + (2 * c.comparisons) + c.assignments$. Kadangi siekėme sugeneruoti algoritmą, kuris visada išrikiuoja duomenis, buvo pasirinkta sprendiniams taikyti kvadratinę baudą priklausančią nuo inversijų skaičiaus. Taip pat sprendiniams buvo taikoma bauda, paremta veikimo laiku. Reikia pripažinti, jog šios baudos laipsnis neturi matematinio pagrindo ir buvo nustatytas eksperimentiškai. Algoritmo atliekamiems priskyrimams ir palyginimams tinkamumo funkcijoje buvo taikomi svoriai, paremti praeitame skyriuje apibrėžtais efektyvaus Šelo algoritmo varianto kriterijais ir eksperimentiškai gautais duomenimis apie tipiškai atliekamus palyginimus ir priskyrimus.

Genetinio algoritmo rezultatų saugojimui buvo pasirinkta naudoti PostgreSQL duomenų bazę. Tai buvo atlikta siekiant automatizuoti genetinio algoritmo rezultatų saugojimo procesą ir palengvinti geriausių rezultatų atranką. Rezultatų saugojimui buvo suprojektuota lentelė, kurios stulpelius sudaro sprendinys JSON formatu, sprendinio tinkamumas, rikiuojamų duomenų dydis, inversijų skaičius po rikiavimo, atliktų palyginimų skaičius, atliktų priskyrimų skaičius ir veikimo laikas. Genetiniam algoritmui baigus darbą gauti rezultatai buvo išsaugomi į failą, kurio turinys po to buvo automatiškai patalpinamas į duomenų bazę pasitelkiant Node.js skriptą.

5.2. Šelo algoritmo variantų generavimas, kai $N = 1000$

Kadangi šiame darbe siekiame ištirti Šelo algoritmo variantus, kurie skiriasi taikomo perėjimo tipu, šiame etape buvo pasirinkta sprendinių inicializacijai naudoti ne atsitiktinai sugeneruotas, o literatūroje rastas tarpų sekas. Tuo tikslu buvo pasirinktos šios tarpų sekos:

- Pratt: $1, 2, 3, 4, 6, 8, 9, 12, \ldots$ [Pra72]
- Sedgewick: $1, 5, 19, 41, 109, 209, 505, 929, \ldots$ [Sed86]
- Incerpi-Sedgewick: $1, 3, 7, 21, 48, 112, 336, 861, \ldots$ [IS85]
- Tokuda: $1, 4, 9, 20, 46, 103, 233, 525, \ldots$ [Tok92]
- Ciura: $1, 4, 10, 23, 57, 132, 301, 701$ [Ciu01]
- Geometrinės tarpų sekos, kur $q \in \{1.95, 2.05, 2.15, 2.25, 2.35, 2.45, 2.55, 2.65\}$ [RB13]

Pirmos 5 tarpų sekos buvo pasirinktos, kadangi literatūroje jos žinomos kaip vienos efektyviausių Šelo algoritmo variantams, kurie naudoja įterpimu paremtus perėjimus. Geometrinės tarpų sekos buvo pasirinktos atsižvelgiant į tai, jog jos įprastai yra naudojamos Šelo algoritmo variantuose,

kurių perėjimai nėra paremti įterpimu. Kai kurios iš pasirinktų tarpų sekų yra begalinės, tad buvo nuspręsta visas pasirinktas sekas sutrumpinti taip, jog jas sudarytų 8 elementai.

Modeliuojama chromosoma šiuo atveju buvo sudaryta iš 8 perėjimų, t.y. jos ilgis atitiko inicializacijai naudotų tarpų sekų ilgį. Chromosomos buvo inicializuojamos remiantis šia formule: $p_i = (rand\_type(), H_i)$, kur $rand\_type()$ yra funkcija grąžinanti atsitiktinį perėjimo tipą, H - atsitiktinai pasirinkta tarpų seka. Chromosomų tinkamumas buvo vertinamas rikiuojant du 1000 elementų dydžio masyvus. Pasitelkiant suprojektuotą GA buvo sugeneruota 50 unikalių sprendinių. Tada iš jų buvo atrinkti 5 didžiausią tinkamumą turintys sprendiniai A1, A2, A3, A4 ir A5, kurie glaustai aptariami žemiau (detalūs rezultatai JSON formatu pateikiami prieduose).

Algoritmą A1 pagrinde sudaro patobulinto įterpimo perėjimai (tik su didžiausiu tarpu naudojamas burbuliuko perėjimas), dažniausiai naudojami Ciura sekos tarpai, taip pat pastebime Sedgewick bei geometrinės tarpų sekos su $q = 2.65$ elementų. Algoritmas A2 naudoja tik Ciura sekos tarpus, pagrinde pasitelkiami patobulinto įterpimo perėjimai, nors pastebime ir burbuliuko perėjimą su tarpu 301. Algoritmo A3 mažiausi tarpai yra iš Ciura tarpų sekos, didesnius tarpus sudaro geometrinės tarpų sekos su $q = 2.55$ elementai, pagrinde naudojami patobulinto įterpimo perėjimai (burbuliuko perėjimas naudojamas tik su didžiausiu tarpu). Algoritmą A4 sudaro Tokuda, Ciura ir geometrinės tarpų sekos su $q = 2.55$ mišinys, didžiausiam tarpui naudojamas burbuliuko perėjimas, visiems kitiems - patobulinto įterpimo perėjimas. Algoritmą A5 sudaro Sedgewick, Ciura ir geometrinės tarpų sekos su $q = 2.65$ mišinys, su tarpais 929 ir 209 naudojami burbuliuko perėjimai, su likusiais tarpais pastebime patobulinto įterpimo perėjimus.

5.3. Šelo algoritmo variantų generavimas, kai $N = 100000$

Šiame etape buvo pasirinkta sprendinių inicializacijai dalinai naudoti sprendinius, gautus praeitame etape. Kadangi tokio dydžio duomenų rikiavimui 8 elementų tarpų seka yra per trumpa jog būtų efektyvi, sprendinių perėjimų sąrašas buvo pratęsiamas iki 14 elementų, pirmuosius 8 perėjimus perkopijuojant iš praeitame etape gautų sprendinių, o likusius užpildant remiantis šia rekursyvia formule: $p_i = (rand\_type(), q * gap(p_{i-1}))$, kur $rand\_type()$ yra funkcija grąžinanti atsitiktinį perėjimo tipą, $gap(x)$ yra funkcija grąžinanti duoto perėjimo tarpą, $q \in [2.2, 2.25]$.

Chromosomų tinkamumas buvo vertinamas rikiuojant vieną 100000 elementų dydžio masyvą. Pasitelkiant suprojektuotą GA buvo sugeneruota 50 unikalių sprendinių. Tada iš jų buvo atrinkti 5 didžiausią tinkamumą turintys sprendiniai B1, B2, B3, B4 ir B5, kurie glaustai aptariami žemiau (detalūs rezultatai JSON formatu pateikiami prieduose).

Algoritmą B1 yra sudaro algoritmo A2 naudoti tarpai, su visais tarpais naudojamas patobulinto įterpimo perėjimas. Algoritmas B2 yra paremtas algoritmų A2 ir A3 naudotais tarpais, pagrinde naudojamas patobulinto įterpimo perėjimas, su tarpu 36335 taip pat pasitelkiamas burbuliuko perėjimas. Algoritmas B3 naudoja algoritmų A2 ir A3 tarpus, dažniausiai naudojami patobulinto įterpimo perėjimai, dviems didžiausiems tarpams taikomi burbuliuko ir mūrijimo perėjimai. Algoritmas B4 yra pagrįstas A2 ir A3 algoritmų tarpais, daugiausia naudojami patobulinto įterpimo

perėjimai, dviems didžiausiems tarpams taikomas burbuliuko perėjimas. Algoritmas B5 pasitelkia algoritmų A2 ir A3 tarpus, su visais tarpais naudojamas patobulinto įterpimo perėjimas.

6. Šelo algoritmo variantų efektyvumo tyrimo aplinkos paruošimas

Eksperimentų vykdymui buvo naudojamas kompiuteris su 2.70 GHz Intel(R) Core(TM) i7-10850H procesoriumi, 32 GB operatyviosios atminties ir Windows 10 operacine sistema. Tyrimas buvo įgyvendintas C++ kalba su MSVC 19.16.27043 kompiliatoriumi.

Vykdant eksperimentus buvo pasirinkta naudoti sveikaskaitinius pradinius duomenis. Tai buvo įgyvendinta naudojant MT19937 pseudoatsitiktinių skaičių generatorių, inicializuotą sisteminio laiko reikšme. Generuojami duomenys buvo tolygiai paskirstyti nuo INT_MIN iki INT_MAX, kas leido žymiai sumažinti duomenų duplikacijos tikimybę. Visi algoritmai buvo vertinami rikiuojant tuos pačius pradinius duomenis.

Algoritmų atliekamų operacijų skaičiavimui buvo pasitelkta anksčiau darbe minėta Element klasė. Veikimo laiko matavimui buvo pasitelktas monotoninis laikrodis std::chrono::steady_clock, kuris nepriklauso nuo sisteminio laiko reikšmės ir užtikrina tikslų laiko intervalų matavimą. Siekiant neiškreipti gautų rezultatų, veikimo laiko ir atliekamų operacijų matavimai buvo vykdomi atskirai.

7. Šelo algoritmo variantų efektyvumo tyrimas

Šis skyrius sudarytas iš 3 poskyrių. Pirmame poskyryje aptariama efektyvumo tyrimo metodika. Antrame poskyryje tiriamas Šelo algoritmo variantų efektyvumas, kai $N \leq 1000$. Trečiame poskyryje tiriamas Šelo algoritmo variantų efektyvumas, kai $N \leq 100000$.

7.1. Efektyvumo tyrimo metodika

Kadangi ne įterpimo perėjimu paremti Šelo algoritmo variantai nėra išsamiai ištirti [Bre01], buvo pasirinkta GA sugeneruotus variantus lyginti su įterpimu paremtais VŠA ir PŠA naudojant keletą skirtingų tarpų sekų. Remiantis anksčiau darbe apibrėžtais Šelo algoritmo variantų efektyvumo kriterijais, efektyvumo vertinimui buvo pasirinkta matuoti vidutiniškai atliekamus palyginimus ir priskyrimus bei vidutinį veikimo laiką.

Tyrimui buvo pasirinktos šios tarpų sekos:

- Sedgewick: $1, 5, 19, 41, 109, 209, 505, 929, \ldots$ [Sed86]
- Tokuda: $1, 4, 9, 20, 46, 103, 233, 525, \ldots$ [Tok92]
- Ciura: $1, 4, 10, 23, 57, 132, 301, 701$ [Ciu01]

Kadangi Ciura seka yra baigtinė, didesnių duomenų dydžių rikiavimui ji buvo pratęsta pasitelkiant rekursyvią formulę $h_k = \lfloor 2.25 h_{k-1} \rfloor$. Sedgewick ir Tokuda sekos yra begalinės, tad jos buvo sutrumpintos taip, jog gauta seka būtų ilgiausia įmanoma seka, kurios visi elementai mažesni už maksimalų tiriamą N.

Atliekamų matavimų skaičius buvo pasirinktas atsižvelgiant į tame etape tiriamų duomenų dydį: jei N pakankamai mažas, tikimybė jog gautas vidutinio veikimo laiko įvertis bus netikslus yra didesnė. Todėl matavimai su mažesniais duomenų dydžiais buvo atliekami daugiau kartų.

Vidutinis veikimo laikas su nedideliais duomenų dydžiais pateikiamas suapvalinus iki šimtosios dalies, atsižvelgiant į tai, jog gautų rezultatų absoliutiniai skirtumai nėra dideli. Visi kiti rezultatai pateikiami suapvalinus iki sveikosios dalies. 3 geriausi vienos kategorijos rezultatai su tam tikru N pateikiami pajuodintu šriftu ir žalia fono spalva.

7.2. Šelo algoritmo variantų efektyvumo tyrimas, kai $N \leq 1000$

Šiame efektyvumo tyrimo etape buvo tiriami šie Šelo algoritmo variantai:

- VŠA su Ciura tarpų seka
- VŠA su Tokuda tarpų seka
- VŠA su Sedgewick tarpų seka
- PŠA su Ciura tarpų seka

- PŠA su Tokuda tarpų seka
- PŠA su Sedgewick tarpų seka
- GA sugeneruoti algoritmai (A1, A2, A3, A4, A5)

7.2.1. Tyrimo rezultatai

1 lentelė. Efektyvumo tyrimo rezultatai, kai $N = 125$

algoritmas	vid. palyginimai	vid. priskyrimai	vid. laikas (µs)
VŠA (Ciura)	**978**	1568	4.04
VŠA (Tokuda)	991	1615	4.90
VŠA (Sedgewick)	1005	1508	3.90
PŠA (Ciura)	**978**	1068	3.94
PŠA (Tokuda)	991	**1042**	3.75
PŠA (Sedgewick)	1005	1125	**3.07**
A1	**978**	1068	**3.63**
A2	**978**	1068	3.68
A3	984	**1053**	3.71
A4	992	**1049**	4.01
A5	**978**	1068	**3.39**

Vertinant rezultatus pateikiamus 1 lentelėje, galime pastebėti, jog mažiausiai palyginimų atliko A1, A5 ir Ciura tarpų seka paremti variantai. Tokuda seka paremtas PŠA šiuo atveju atlieka mažiausiai priskyrimų, nedaug atsilieka A3 ir A4. Vertinant veikimo laiką pirmauja PŠA su Sedgewick tarpų seka.

2 lentelė. Efektyvumo tyrimo rezultatai, kai $N = 250$

algoritmas	vid. palyginimai	vid. priskyrimai	vid. laikas (µs)
VŠA (Ciura)	**2377**	3776	10.71
VŠA (Tokuda)	2401	3853	10.00
VŠA (Sedgewick)	2453	3685	9.63
PŠA (Ciura)	**2377**	2571	**9.08**
PŠA (Tokuda)	2401	**2533**	9.78
PŠA (Sedgewick)	2453	2696	**8.40**
A1	**2377**	2571	9.23
A2	**2377**	2571	9.44
A3	2390	**2558**	**9.09**
A4	2400	**2542**	9.27
A5	**2367**	2578	9.15

Nagrinėjant rezultatus pateikiamus 2 lentelėje, galime pastebėti, jog mažiausiai palyginimų atlikę algoritmai pradeda atsiskirti ir A5 algoritmas jų atlieka mažiausiai. A3 ir A4 algoritmai bei Tokuda seka paremtas PŠA pateikia geriausius rezultatus vertintant atliktus priskyrimus. Vertinant veikimo laiką pirmauja PŠA su Sedgewick tarpų seka.

3 lentelė. Efektyvumo tyrimo rezultatai, kai $N = 500$

algoritmas	vid. palyginimai	vid. priskyrimai	vid. laikas (µs)
VŠA (Ciura)	**5621**	8835	24.83
VŠA (Tokuda)	5666	9001	24.52
VŠA (Sedgewick)	5783	8642	24.56
PŠA (Ciura)	**5621**	**5994**	22.96
PŠA (Tokuda)	5666	5999	23.03
PŠA (Sedgewick)	5783	6360	**21.30**
A1	**5612**	**5991**	24.24
A2	**5621**	**5994**	**21.71**
A3	5655	6027	23.16
A4	5669	6016	23.26
A5	**5607**	6089	**22.66**

3 lentelėje matome, jog A1 ir A5 bei Ciura tarpų seka paremti variantai pasirodė geriausiai vertinant atliktus palyginimus. Vertinant atliktus priskyrimus, geriausi rezultatai gauti su A1, A2 ir PŠA su Ciura tarpų seka. Pagal veikimo laiką pirmauja PŠA su Sedgewick tarpų seka.

4 lentelė. Efektyvumo tyrimo rezultatai, kai $N = 1000$

algoritmas	vid. palyginimai	vid. priskyrimai	vid. laikas (µs)
VŠA (Ciura)	13041	20270	55.57
VŠA (Tokuda)	13129	20708	56.69
VŠA (Sedgewick)	13395	20089	52.55
PŠA (Ciura)	13041	**13816**	55.05
PŠA (Tokuda)	13129	**13863**	56.55
PŠA (Sedgewick)	13395	14525	**50.50**
A1	**12961**	13905	**52.21**
A2	**12990**	14048	52.34
A3	13070	**13817**	53.18
A4	13121	13889	54.02
A5	**13023**	14366	**51.56**

Pagal 4 lentelės rezultatus mažiausiai palyginimų atliko GA sugeneruoti algoritmai: A1, A2 ir A5. Mažiausiai priskyrimų atliko PŠA su Ciura ir Tokuda tarpų sekomis bei A3. PŠA su Sedgewick tarpų seka išlieka geriausiu vertinant veikimo laiką.

7.3. Šelo algoritmo variantų efektyvumo tyrimas, kai $N \leq 100000$

Šiame efektyvumo tyrimo etape buvo tiriami šie Šelo algoritmo variantai:

- VŠA su Ciura tarpų seka
- VŠA su Tokuda tarpų seka
- VŠA su Sedgewick tarpų seka
- PŠA su Ciura tarpų seka
- PŠA su Tokuda tarpų seka
- PŠA su Sedgewick tarpų seka
- GA sugeneruoti algoritmai (B1, B2, B3, B4, B5)

7.3.1. Tyrimo rezultatai

5 lentelė. Efektyvumo tyrimo rezultatai, kai $N = 12500$

algoritmas	vid. palyginimai	vid. priskyrimai	vid. laikas (µs)
VŠA (Ciura)	**247436**	376596	1009
VŠA (Tokuda)	248135	381261	1025
VŠA (Sedgewick)	254437	381187	984
PŠA (Ciura)	**247436**	262114	**936**
PŠA (Tokuda)	248135	**261766**	973
PŠA (Sedgewick)	254437	268334	**929**
B1	**247107**	**261734**	949
B2	**247317**	**261825**	**948**
B3	248879	263458	972
B4	248598	263534	951
B5	247540	262044	956

5 lentelėje pateikiamuose rezultatuose matome, kad pagal mažiausiai palyginimų atliko B1 ir B2 algoritmai, nedaug atsiliko Ciura tarpų seka paremti variantai. Mažiausiai priskyrimų atliko B1 ir PŠA su Tokuda tarpų seka. Mažiausias veikimo laikas pastebimas su PŠA naudojant Sedgewick tarpų seką.

6 lentelė. Efektyvumo tyrimo rezultatai, kai $N = 25000$

algoritmas	vid. palyginimai	vid. priskyrimai	vid. laikas (µs)
VŠA (Ciura)	**542838**	822347	2323
VŠA (Tokuda)	544133	832851	2379
VŠA (Sedgewick)	558162	838206	2267
PŠA (Ciura)	**542838**	574642	**2162**
PŠA (Tokuda)	544133	**573608**	2219
PŠA (Sedgewick)	558162	586459	**2140**
B1	**541866**	**573477**	**2157**
B2	**541303**	574561	2183
B3	546239	579532	2208
B4	545417	576050	2225
B5	543056	**573435**	2214

Vertinant rezultatus pateikiamus 6 lentelėje, pastebime, jog mažiausiai palyginimų atliko B2, nedaug atsiliko B1 ir Ciura tarpų seka paremti variantai. Pagal atliktus priskyrimus geriausi rezul-

tatai gauti su B5, B1 ir PŠA su Tokuda tarpų seka. PŠA su Sedgewick tarpų seka veikimo laiko rezultatai šiuo atveju yra geriausi.

7 lentelė. Efektyvumo tyrimo rezultatai, kai $N = 50000$

algoritmas	vid. palyginimai	vid. priskyrimai	vid. laikas (µs)
VŠA (Ciura)	**1180477**	1781235	5203
VŠA (Tokuda)	1184476	1804441	5335
VŠA (Sedgewick)	1211998	1817385	5140
PŠA (Ciura)	**1180477**	1250232	4816
PŠA (Tokuda)	1184476	**1244335**	4866
PŠA (Sedgewick)	1211998	1269569	**4723**
B1	**1178027**	**1247428**	**4816**
B2	**1180429**	1249592	**4795**
B3	1189048	1261191	4822
B4	1184607	1252540	4819
B5	1180783	**1246917**	4832

Nagrinėjant 7 lentelės rezultatus, pastebime, jog mažiausiai palyginimų atliko B1, taip pat geri rezultatai gauti su B2 ir Ciura tarpų seka paremtais variantais. Vertinant atliktus priskyrimus, geriausi rezultatai gauti su PŠA naudojant Tokuda tarpų seką ir B5 bei B1 algoritmais. PŠA su Sedgewick tarpų seka rodo geriausius veikimo laiko rezultatus.

8 lentelė. Efektyvumo tyrimo rezultatai, kai $N = 100000$

algoritmas	vid. palyginimai	vid. priskyrimai	vid. laikas (µs)
VŠA (Ciura)	**2552009**	3835896	11305
VŠA (Tokuda)	2564127	3888498	11853
VŠA (Sedgewick)	2624030	3941772	11980
PŠA (Ciura)	**2552009**	2710178	10765
PŠA (Tokuda)	2564127	**2691417**	10671
PŠA (Sedgewick)	2624030	2741613	**10424**
B1	**2545884**	**2703088**	**10475**
B2	**2549224**	2735020	10568
B3	2569991	2768514	10481
B4	2556929	2718989	**10434**
B5	2552700	**2697404**	10542

8 lentelės rezultatuose matome, jog mažiausiai palyginimų atliko B1 ir B2 algoritmai, taip pat neblogus rezultatus rodo Ciura tarpų seka paremti variantai. Analizuojant atliktų priskyrimų

rezultatus, pirmauja PŠA naudojant Tokuda tarpų seką ir B5 bei B1 algoritmai. PŠA su Sedgewick tarpų seka pateikia geriausius veikimo laiko rezultatus.

7.4. Tyrimo rezultatų apibendrinimas

Apibendrinant rezultatus, gautus tiriant Šelo algoritmo variantus kai $N \leq 1000$, galime teigti, jog PŠA su Ciura tarpų seka ir algoritmas A1 šiame etape buvo efektyviausi, kadangi jie atliko mažiausiai palyginimų ir priskyrimų bei pateikė pakankamai gerus veikimo laikus.

Įdomu pastebėti, jog didelė dalis GA sugeneruotų algoritmų mažesniems duomenų dydžiams naudoja visiškai tokius pat perėjimus ir tarpus, kaip ir PŠA su Ciura tarpų seka. Tuo pačiu galime konstatuoti, jog veikimo laiko įverčiai nėra ypač tikslūs - jei algoritmai atlieka tas pačias operacijas ir rikiuoja tuos pačius duomenis, natūralu būtų tikėtis labai panašių veikimo laikų. Taip pat matome, jog su didesniais tarpais taikant burbuliuko perėjimą gaunamas algoritmas (A2), kuris atlieka mažiau palyginimų ir daugiau priskyrimų nei analogišką tarpų seką naudojantis algoritmas paremtas tik patobulinto įterpimo perėjimais (PŠA su Ciura tarpų seka). Ši skirtumą galime paaiškinti tuo, jog burbuliuko perėjimas mažiau aprikiuoja duomenis nei įterpimo perėjimai, taigi sutaupoma palyginimų, tačiau sekantys perėjimai turi atlikti daugiau darbo, dėl ko pastebimas padidėjęs priskyrimų skaičius. Vertinant tik veikimo laikus, panašu jog šis kompromisas duoda teigiamą efektą, tačiau reikia atsižvelgti ir į faktą, jog tyrimui buvo naudojami tik sveikaskaitiniai duomenys, kurių priskyrimas yra labai pigus atliekamų skaičiavimų prasme.

Apibendrinant rezultatus, gautus tiriant Šelo algoritmo variantus kai $N \leq 100000$, galime teigti, jog efektyviausias buvo B1 algoritmas, kadangi jo rezultatai su visais tame etape tirtais duomenų dydžiais buvo vieni geriausių tiek vertinant atliktas operacijas, tiek veikimo laiką. Reikia pastebėti, jog pirmi 8 B1 algoritmo perėjimai sutampa su PŠA naudojant Ciura tarpų seką. Šis algoritmas taip pat naudoja tik patobulintą iterpimo perėjimą, taigi iš tiesų buvo sugeneruota efektyvi tarpų seka $S_1 = 1, 4, 10, 23, 57, 132, 301, 701, 1577, 3552, 7983, 17982, 40459, 91032$, kuri yra Ciura tarpų sekos tęsinys.

Remiantis gautais rezultatais, galime daryti išvadą, jog nėra objektyvių priežasčių rinktis VŠA - PŠA naudojant tas pačias tarpų sekas su visais duomenų dydžiais atlieka 29-43% mažiau priskyrimų ir veikia greičiau. Ypač teigiamas PŠA naudojimo efektas pastebimas su Sedgewick tarpų seka, kur veikimo laikas su visais duomenų dydžiais yra 4-24% mažesnis lyginant su VŠA paremtu variantu. Tuo pačiu PŠA su Sedgewick tarpų seka yra nenugįnčijamai sparčiausias tirtas algoritmas.

Bendrai vertinant GA sugeneruotus algoritmus galime teigti, jog gauti algoritmai yra gana efektyvūs. Taip pat galime pastebėti, jog gautuose algoritmuose dominuoja patobulinto įterpimo perėjimai. Su didesniais tarpais taip pat pastebime ir burbuliuko perėjimų. Tiesa, generuojant algoritmus buvo pasirinktas gana siauras q rėžis geometrinėms tarpų sekoms, kuris galėjo įtakoti šį rezultatą - galbūt kitokių tipų perėjimai būtų pastebimi dažniau, jei inicializacijai parinktos tarpų sekos jiems būtų palankesnės.

Taip pat įdomu pastebėti, jog gauti algoritmai sugebėjo atliekamų palyginimų atžvilgiu aplenkti

Ciura tarpų seka paremtus Šelo algoritmo variantus. Tiesa, su nedideliais N tai buvo pasiekta tik paaukojant atliekamus priskyrimus ir naudojant burbuliuko perėjimą didesniems tarpams. Autoriaus nuomone, algoritmai kombinuojantys keletą skirtingų perėjimų nėra tinkami naudoti praktikoje, kadangi daug paprasčiau įgyvendinti algoritmą, kuris rikiuoja naudodamas tik vieno tipo perėjimą.

Išvados

Renkantis kurį Šelo algoritmo variantą naudoti, nesunkiai galime rekomenduoti patobulintą Šelo algoritmą. Remiantis gautais rezultatais, jis atlieka mažiau priskyrimo operacijų ir veikia greičiau už vadovėlinę versiją.

Nedideliems duomenų dydžiams ($N \leq 1000$) rikiuoti bendru atveju galime rekomenduoti patobulintą Šelo algoritmą su Ciura tarpų seka. Remiantis puikiais veikimo laiko rezultatais taip pat galime rekomenduoti ir patobulintą Šelo algoritmą su Sedgewick tarpų seka. Nors darbo metu gautas algoritmas A1 taip pat yra gana efektyvus, jį sunku rekomenduoti dėl kelių skirtingų perėjimų taikymo, kas stipriai apsunkina implementaciją.

Vidutiniams duomenų dydžiams ($1000 \leq N \leq 100000$) rikiuoti bendru atveju galime rekomenduoti patobulintą Šelo algoritmą su S_1 tarpų seka. Remiantis puikiais veikimo laiko rezultatais taip pat galime rekomenduoti ir patobulintą Šelo algoritmą su Sedgewick tarpų seka.

Reikia pastebėti, jog šiame darbe visi algoritmai buvo tiriami naudojant tik atsitiktinius duomenis. Atliekant išsamesnį Šelo algoritmo variantų tyrimą, reikėtų panaudoti daugiau skirtingų duomenų rinkinių: išrikiuotų (didėjančia/mažėjančia tvarka), dalinai išrikiuotų, sutinkamų realybėje. Taip pat reikėtų panaudoti ir skirtingus duomenų tipus, kadangi palyginimo ir priskyrimo operacijų kaina priklauso nuo rikiuojamų duomenų tipo. Tai galėtų duoti naudingų įžvalgų į operacijų kainų įtaką veikimo laikui.

Įvertinant gautus rezultatus, galima teigti, jog genetiniai algoritmai yra tinkamas metodas efektyvių Šelo algoritmo variantų radimui. Tiesa, suprojektuotas genetinis algoritmas galėtų būti patobulintas siekiant sugeneruoti dar geresnių variantų. Remiantis tuo, jog darbo metu sugeneruoti algoritmai veikimo greičiu atsiliko nuo PŠA su Sedgewick tarpų seka, turime priežasčių manyti, jog tinkamumo funkcijoje taikyti svoriai nėra visiškai teisingi ir galėjo atmesti kokybiškus variantus, kurie veikė greitai, bet atliko santykinai daug palyginimų. Ateityje būtų įdomu panagrinėti, kokia tinkamumo funkcija tiksliausiai išreiškia praktinį algoritmo efektyvumą. Pavertus suprojektuotą vienkriterinį genetinį algoritmą į daugiakriterinį, taip pat būtų įmanoma gauti daugiau kokybiškų variantų, kadangi veikimo laikas ir atliekamos operacijos dažnai nekoreliuoja ir šiuos kriterijus būtų prasminga atskirti.

Dar viena potenciali naujų darbų kryptis galėtų būti ir geometrinių tarpų sekų generavimas pasitelkiant genetinius algoritmus. Šiame darbe buvo gautas Ciura tarpų sekos tęsinys S_1, kuris yra efektyvesnis už literatūroje pateikiamą tęsinį gautą naudojant rekursyvią formulę $h_k = \lfloor 2.25h_{k-1} \rfloor$. Sukonstravus genetinį algoritmą Ciura tarpų sekos tęsinių generavimui naudojant skirtingus q galėtų būti gauta ir dar efektyvesnė tarpų seka. Žinoma, nėra būtina apsiriboti vien tęsinių generavimu - vienodai prasmingas uždavinys būtų ir pilnų geometrinių tarpų sekų generavimas pasitelkiant genetinius algoritmus.

Literatūra

[Bre01] Bronislava Brejová. Analyzing variants of Shellsort. *Information Processing Letters*, 79(5):223–227, 2001.

[Ciu01] Marcin Ciura. Best increments for the average case of shellsort. *International Symposium on Fundamentals of Computation Theory*, p.p. 106–117. Springer, 2001.

[Dob+80] Wlodzimierz Dobosiewicz ir k.t. An efficient variation of bubble sort, 1980.

[GK72] David Gale ir Richard M. Karp. A phenomenon in the theory of sorting. *Journal of Computer and System Sciences*, 6(2):103–115, 1972. ISSN: 0022-0000. DOI: https://doi.org/10.1016/S0022-0000(72)80016-3. URL: https://www.sciencedirect.com/science/article/pii/S0022000072800163.

[Goo14] Michael T Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in o (n log n) time. *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, p.p. 684–693, 2014.

[HAA+19] Ahmad Hassanat, Khalid Almohammadi, Esra' Alkafaween, Eman Abunawas, Awni Hammouri ir VB Prasath. Choosing mutation and crossover ratios for genetic algorithms—a review with a new dynamic approach. *Information*, 10(12):390, 2019.

[Hab72] A Nico Habermann. Parallel neighbor-sort (or the glory of the induction principle), 1972.

[Hol92] John Henry Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[IS85] Janet Incerpi ir Robert Sedgewick. Improved upper bounds on Shellsort. *Journal of Computer and System Sciences*, 31(2):210–224, 1985.

[IS86] Janet Incerpi ir Robert Sedgewick. *Practical variations of shellsort*. Disertacija, INRIA, 1986.

[Knu70] Donald E. Knuth. Von Neumann's First Computer Program. *ACM Comput. Surv.*, 2(4):247–260, 1970-12. ISSN: 0360-0300. DOI: 10.1145/356580.356581. URL: https://doi.org/10.1145/356580.356581.

[Lem94] P Lemke. The performance of randomized Shellsort-like network sorting algorithms. *SCAMP working paper P18/94*. Institute for Defense Analysis, 1994.

[MAM+17] Arash Mohammadi, Houshyar Asadi, Shady Mohamed, Kyle Nelson ir Saeid Nahavandi. OpenGA, a C++ Genetic Algorithm Library. *Systems, Man, and Cybernetics (SMC), 2017 IEEE International Conference on*, p.p. 2051–2056. IEEE, 2017.

[Mas11] Joanna Masel. Genetic drift. *Current Biology*, 21(20):R837–R838, 2011.

[Pra72] Vaughan R Pratt. Shellsort and sorting networks. Tech. atask., STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1972.

[RB13] Irmantas Radavičius ir Mykolas Baranauskas. An empirical study of the gap sequences for Shell sort. *Lietuvos matematikos rinkinys*, 54(A):61–66, 2013-12. DOI: 10.15388/LMR.A.2013.14. URL: https://www.journals.vu.lt/LMR/article/view/14899.

[Sed86] Robert Sedgewick. A new upper bound for Shellsort. *Journal of Algorithms*, 7(2):159–173, 1986. ISSN: 0196-6774. DOI: https://doi.org/10.1016/0196-6774(86)90001-5. URL: https://www.sciencedirect.com/science/article/pii/0196677486900015.

[Sed96] Robert Sedgewick. Analysis of Shellsort and related algorithms. *European Symposium on Algorithms*, p.p. 1–11. Springer, 1996.

[She59] D. L. Shell. A High-Speed Sorting Procedure. *Commun. ACM*, 2(7):30–32, 1959-07. ISSN: 0001-0782. DOI: 10.1145/368370.368387. URL: https://doi.org/10.1145/368370.368387.

[SY99] Richard Simpson ir Shashidhar Yachavaram. Faster shellsort sequences: A genetic algorithm application. *Computers and Their Applications*, p.p. 384–387, 1999.

[Tok92] Naoyuki Tokuda. An Improved Shellsort. *Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing '92, Volume 1 - Volume I*, p.p. 449–457, NLD. North-Holland Publishing Co., 1992. ISBN: 044489747X.

[Whi94] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

Priedas Nr. 1
Inversijų skaičiavimas pasitelkiant rikiavimą sąlaja

Algorithm 7 Inversijas skaičiuojantis rikiavimas sąlaja

1: **procedure** MERGESORT(a)
2: $n \leftarrow size(a)$
3: $inv \leftarrow 0$
4: **if** $n \leq 1$ **then**
5: **return** 0
6: **end if**
7: let l and r be the result of splitting a at $n/2$
8: $inv \leftarrow inv + mergesort(l)$
9: $inv \leftarrow inv + mergesort(r)$
10: $inv \leftarrow inv + merge(l,r,a)$
11: **return** inv
12: **end procedure**

Algorithm 8 Inversijas skaičiuojantis sąlajos algoritmas

```
1:  procedure MERGE(a,b,c)
2:      inv ← 0
3:      i ← 0
4:      j ← 0
5:      n ← size(a)
6:      m ← size(b)
7:      for k ← 0 to size(c) − 1 do
8:          if i < n then
9:              if j < m then
10:                 if a[i] ≤ b[j] then
11:                     c[k] ← a[i])
12:                     i ← i + 1
13:                 else
14:                     c[k] ← b[j])
15:                     j ← j + 1
16:                     inv ← inv + (n − i)
17:                 end if
18:             else
19:                 c[k] ← a[i])
20:                 i ← i + 1
21:             end if
22:         else
23:             c[k] ← b[j])
24:             j ← j + 1
25:         end if
26:     end for
27:     return inv
28: end procedure
```

Priedas Nr. 2

Algoritmas A1


```
[
  {
    "gap": 861,
    "passType": "bubble"
  },
  {
    "gap": 347,
    "passType": "insertion_improved"
  },
  {
    "gap": 132,
    "passType": "insertion_improved"
  },
  {
    "gap": 57,
    "passType": "insertion_improved"
  },
  {
    "gap": 23,
    "passType": "insertion_improved"
  },
  {
    "gap": 10,
    "passType": "insertion_improved"
  },
  {
    "gap": 4,
    "passType": "insertion_improved"
  },
  {
    "gap": 1,
    "passType": "insertion_improved"
  }
]
```

Priedas Nr. 3

Algoritmas A2


```
[
  {
    "gap": 701,
    "passType": "insertion_improved"
  },
  {
    "gap": 301,
    "passType": "bubble"
  },
  {
    "gap": 132,
    "passType": "insertion_improved"
  },
  {
    "gap": 57,
    "passType": "insertion_improved"
  },
  {
    "gap": 23,
    "passType": "insertion_improved"
  },
  {
    "gap": 10,
    "passType": "insertion_improved"
  },
  {
    "gap": 4,
    "passType": "insertion_improved"
  },
  {
    "gap": 1,
    "passType": "insertion_improved"
  }
]
```

Priedas Nr. 4
Algoritmas A3


```
[
  {
    "gap": 702,
    "passType": "bubble"
  },
  {
    "gap": 275,
    "passType": "insertion_improved"
  },
  {
    "gap": 108,
    "passType": "insertion_improved"
  },
  {
    "gap": 57,
    "passType": "insertion_improved"
  },
  {
    "gap": 23,
    "passType": "insertion_improved"
  },
  {
    "gap": 10,
    "passType": "insertion_improved"
  },
  {
    "gap": 4,
    "passType": "insertion_improved"
  },
  {
    "gap": 1,
    "passType": "insertion_improved"
  }
]
```

Priedas Nr. 5
Algoritmas A4


```
[
  {
    "gap": 701,
    "passType": "bubble"
  },
  {
    "gap": 233,
    "passType": "insertion_improved"
  },
  {
    "gap": 108,
    "passType": "insertion_improved"
  },
  {
    "gap": 43,
    "passType": "insertion_improved"
  },
  {
    "gap": 20,
    "passType": "insertion_improved"
  },
  {
    "gap": 9,
    "passType": "insertion_improved"
  },
  {
    "gap": 4,
    "passType": "insertion_improved"
  },
  {
    "gap": 1,
    "passType": "insertion_improved"
  }
]
```

Priedas Nr. 6
Algoritmas A5


```
[
  {
    "gap": 929,
    "passType": "bubble"
  },
  {
    "gap": 347,
    "passType": "insertion_improved"
  },
  {
    "gap": 209,
    "passType": "bubble"
  },
  {
    "gap": 57,
    "passType": "insertion_improved"
  },
  {
    "gap": 23,
    "passType": "insertion_improved"
  },
  {
    "gap": 10,
    "passType": "insertion_improved"
  },
  {
    "gap": 4,
    "passType": "insertion_improved"
  },
  {
    "gap": 1,
    "passType": "insertion_improved"
  }
]
```

Priedas Nr. 7

Algoritmas B1


```
[{
  "gap": 91032, "passType": "insertion_improved"
},
{
  "gap": 40459, "passType": "insertion_improved"
},
{
  "gap": 17982, "passType": "insertion_improved"
},
{
  "gap": 7983, "passType": "insertion_improved"
},
{
  "gap": 3552, "passType": "insertion_improved"
},
{
  "gap": 1577, "passType": "insertion_improved"
},
{
  "gap": 701, "passType": "insertion_improved"
},
{
  "gap": 301, "passType": "insertion_improved"
},
{
  "gap": 132, "passType": "insertion_improved"
},
{
  "gap": 57, "passType": "insertion_improved"
},
{
  "gap": 23, "passType": "insertion_improved"
},
{
  "gap": 10, "passType": "insertion_improved"
},
{
  "gap": 4, "passType": "insertion_improved"
},
{
  "gap": 1, "passType": "insertion_improved"
}]
```

Priedas Nr. 8
Algoritmas B2


```
[{
  "gap": 84685, "passType": "insertion"
},
{
  "gap": 36335, "passType": "bubble"
},
{
  "gap": 20724, "passType": "insertion_improved"
},
{
  "gap": 7692, "passType": "insertion_improved"
},
{
  "gap": 3459, "passType": "insertion_improved"
},
{
  "gap": 1549, "passType": "insertion_improved"
},
{
  "gap": 702, "passType": "insertion_improved"
},
{
  "gap": 301, "passType": "insertion_improved"
},
{
  "gap": 132, "passType": "insertion_improved"
},
{
  "gap": 57, "passType": "insertion_improved"
},
{
  "gap": 23, "passType": "insertion_improved"
},
{
  "gap": 10, "passType": "insertion_improved"
},
{
  "gap": 4, "passType": "insertion_improved"
},
{
  "gap": 1, "passType": "insertion_improved"
}]
```

Priedas Nr. 9
Algoritmas B3


```
[{
  "gap": 64769, "passType": "brick"
},
{
  "gap": 40412, "passType": "bubble"
},
{
  "gap": 17961, "passType": "insertion_improved"
},
{
  "gap": 8929, "passType": "insertion_improved"
},
{
  "gap": 2660, "passType": "insertion_improved"
},
{
  "gap": 1577, "passType": "insertion_improved"
},
{
  "gap": 702, "passType": "insertion_improved"
},
{
  "gap": 275, "passType": "insertion_improved"
},
{
  "gap": 132, "passType": "insertion_improved"
},
{
  "gap": 57, "passType": "insertion_improved"
},
{
  "gap": 23, "passType": "insertion_improved"
},
{
  "gap": 10, "passType": "insertion_improved"
},
{
  "gap": 4, "passType": "insertion_improved"
},
{
  "gap": 1, "passType": "insertion_improved"
}]
```

Priedas Nr. 10
Algoritmas B4


```
[{
  "gap": 84718, "passType": "bubble"
},
{
  "gap": 40238, "passType": "bubble"
},
{
  "gap": 17002, "passType": "insertion_improved"
},
{
  "gap": 7594, "passType": "insertion_improved"
},
{
  "gap": 3450, "passType": "insertion_improved"
},
{
  "gap": 2044, "passType": "insertion_improved"
},
{
  "gap": 702, "passType": "insertion_improved"
},
{
  "gap": 301, "passType": "insertion_improved"
},
{
  "gap": 132, "passType": "insertion_improved"
},
{
  "gap": 57, "passType": "insertion_improved"
},
{
  "gap": 23, "passType": "insertion_improved"
},
{
  "gap": 10, "passType": "insertion_improved"
},
{
  "gap": 4, "passType": "insertion_improved"
},
{
  "gap": 1, "passType": "insertion_improved"
}]
```

Priedas Nr. 11

Algoritmas B5


```
[{
  "gap": 83276, "passType": "insertion_improved"
},
{
  "gap": 37567, "passType": "insertion_improved"
},
{
  "gap": 16947, "passType": "insertion_improved"
},
{
  "gap": 7504, "passType": "insertion_improved"
},
{
  "gap": 3475, "passType": "insertion_improved"
},
{
  "gap": 1556, "passType": "insertion_improved"
},
{
  "gap": 702, "passType": "insertion_improved"
},
{
  "gap": 301, "passType": "insertion_improved"
},
{
  "gap": 132, "passType": "insertion_improved"
},
{
  "gap": 57, "passType": "insertion_improved"
},
{
  "gap": 23, "passType": "insertion_improved"
},
{
  "gap": 10, "passType": "insertion_improved"
},
{
  "gap": 4, "passType": "insertion_improved"
},
{
  "gap": 1, "passType": "insertion_improved"
}]
```

Priedas Nr. 12
Projekto programinis kodas

Kursinio projekto praktinės dalies programinis kodas pasiekiamas GitHub platformoje šiuo adresu: `https://github.com/dzaleskis/course_project_lab`