VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS INSTITUTAS
INFORMATIKOS KATEDRA

Baigiamasis bakalauro darbas

Rikiavimo tobulinimas genetiniais algoritmais

(Improving Sorting with Genetic Algorithms)

Atliko: 4 kurso 2 grupės studentas
Deividas Zaleskis (parašas)
Darbo vadovas:
Irmantas Radavičius (parašas)
Recenzentas:
Vardukas Pavardukas (parašas)

Vilnius
2022

Turinys

Sąvokų apibrėžimai

Šelo algoritmas GA OS WSL

Įvadas

Duomenų rikiavimas yra vienas aktyviausiai tiriamų uždavinių informatikos moksle. Iš dalies tai lemia rikiavimo uždavinio prieinamumas ir analizės paprastumas. Formaliai rikiavimo uždavinys 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. Kadangi rikiavimo uždavinio sąlyga yra gana paprasta, tai suteikia didelę galimų implementacijų įvairovę. Todėl nauji rikiavimo algoritmai ir įvairūs patobulinimai egzistuojantiems algoritmams yra kuriami ir dabar.

Rikiavimo uždavinys yra fundamentalus, kadangi rikiavimas padeda pagrindą efektyviam kitų uždavinių sprendimui. Kaip to pavyzdį galima pateikti dvejetainės paieškos algoritmą, kurio prielaida, jog duomenys yra išrikiuoti, leidžia sumažinti paieškos laiko sudėtingumą iki $O(log\, n)$. Rikiavimas taip pat svarbus duomenų normalizavimui bei pateikimui žmonėms lengvai suprantamu formatu. Kadangi duomenų rikiavimas yra fundamentalus uždavinys, net ir nežymūs patobulinimai žvelgiant bendrai gali atnešti didelę naudą.

Rikiavimo uždaviniui spręsti egzistuoja labai įvairių algoritmų. Plačiausiai žinomi yra klasikiniai rikiavimo algoritmai: rikiavimas sąlaja (angl. merge sort), rikiavimas įterpimu (angl. insertion sort) ir greitojo rikiavimo algoritmas (angl. quicksort). Tiesa, šie algoritmai turi įvairių trukūmų: rikiavimas sąlaja dažnai veikia lėčiau nei nestabilūs algoritmai ir naudoja $O(n)$ papildomos atminties, rikiavimas įterpimu yra efektyvus tik kai rikiuojamų duomenų dydis yra mažas, o greitojo rikiavimo algoritmas blogiausiu atveju turi $O(n^2)$ laiko sudėtingumą. Todėl šiuo metu praktikoje plačiausiai naudojami hibridiniai rikiavimo algoritmai, kurie apjungia kelis klasikinius algoritmus į vieną panaudodami jų geriausias savybes. Pavyzdžiui, C++ programavimo kalbos standartinėje bibliotekoje naudojamas introspektyvaus rikiavimo (angl. introsort) algoritmas įprastai naudoja greitojo rikiavimo algoritmą, pasiekus tam tikrą rekursijos gylį yra naudojamas rikiavimas krūva (angl. heapsort) siekiant išvengti $O(n^2)$ laiko sudėtingumo, o kai rikiuojamų duomenų dydis yra pakankamai mažas, pasitelkiamas rikiavimas įterpimu, kadangi su mažais duomenų dydžiais jis yra efektyvesnis. Apibendrinant, pasitelkiant įvairius klasikinius rikiavimo algoritmus ir jų unikalias savybes yra įmanoma rikiavimo uždavinį spręsti efektyviau.

Vienas iš teorine prasme įdomiausių klasikinių algoritmų yra Šelo rikiavimo algoritmas. Šelo algoritmą galima laikyti rikiavimo įterpimu optimizacija, kadangi atliekant rikiavimą yra lyginami ne tik gretimi elementai, kas leidžia kai kuriuos elementus perkelti į galutinę poziciją atliekant mažiau operacijų. Pagrindinė algoritmo idėja - išskaidyti rikiuojamą seką S į posekius $S_1, S_2, ..., S_n$, kur $S_i = (s_i, s_{i+h}, s_{i+2h}, ...)$ ir atskirai išrikiuoti kiekvieną posekį S_i. Įprastai tarpų rinkinys, kuriuo remiantis formuojami rikiuojami posekiai, vadinamas tarpų seka. Šelo algoritmo efektyvumas priklauso nuo pasirinktos tarpų sekos, todėl bendra teorinė šio algoritmo analizė yra labai sudėtinga. Taip pat reikia pastebėti, jog praktikoje efektyviausi yra eksperimentiškai gauti Šelo algoritmo variantai [Ciu01; Tok92].

Vienas iš eksperimentinių metodų, kuriuos galima taikyti efektyvių Šelo algoritmo variantų

radimui, yra genetinis algoritmas. Genetinis algoritmas yra metodas paremtas žiniomis apie natūralios atrankos procesą, leidžiantis efektyviai spręsti paieškos ir optimizavimo uždavinius. Pagrindinis genetinių algoritmų veikimo principas yra pradinės populiacijos generavimas ir iteratyvus jos atnaujinimas, siekiant, jog bendra populiacijos kokybė didėtų kiekvieną iteraciją. Populiacijos atnaujinimas yra vykdomas pasitelkiant biologijos įkvėptus atrankos, rekombinacijos ir mutacijos operatorius.

Literatūroje galima rasti darbų [RBH+02; SY99], kuriuose genetiniai algoritmai yra taikomi naujų Šelo algoritmo variantų radimui. Simpson-Yachavaram darbe daugiausia dėmesio skiriama atliekamiems palyginimams, teigiama, jog gauti variantai atlieka mažiausiai palyginimo operacijų, tačiau jie yra lyginami tik su Sedgewick ir Incerpi-Sedgewick variantais. Šiuo metu yra laikoma, jog vidutiniškai atliekamų palyginimų atžvilgiu optimaliausias yra Ciura [Ciu01] variantas. Roos et al. darbe daugiausia dėmesio skiriama veikimo laikui, teigiama, jog gautas variantas veikia greičiau nei kiti tirti variantai, tačiau pateikiamuose rezultatuose jis lyginamas tik su Simpson-Yachavaram variantu, tiriami tik keli duomenų dydžiai. Nepaisant aukščiau aprašytų darbų trūkumų, juose pateikiami rezultatai rodo, jog genetinių algoritmų taikymas Šelo algoritmo variantų konstravimui gali būti prasmingas ir duoti reikšmingų rezultatų.

Atliekant kursinį projektą buvo atlikti pradiniai eksperimentai, siekiant genetinių algoritmų pagalba sugeneruoti efektyvius Šelo algoritmo variantus. Viena iš problemų, su kuriomis buvo susidurta tame darbe, buvo pusiausvyros tarp atliekamų operacijų ir veikimo laiko išlaikymas, kadangi algoritmai, kurie atlieka labai mažai operacijų, tačiau veikia lėtai, nėra ypač naudingi praktiniams taikymams, o algoritmai, kurie veikia greitai, tačiau atlieka labai daug operacijų, rikiuojant sudėtingus duomenų tipus gali reikšmingai sulėtėti. Todėl prasmingiausia ieškoti naujų Šelo algoritmo variantų pasitelkiant tokį metodą, kuris išlaikytų pusiausvyrą tarp veikimo laiko ir atliekamų operacijų, neteikdamas prioriteto nei vienam iš šių kriterijų. Kursiniame projekte variantų generavimui buvo pasitelktas vienkriterinis genetinis algoritmas, veikimo laiko ir atliekamų operacijų pusiausvyros problemai spręsti naudotas svorinės sumos metodas. Retrospektyviai galima teigti, jog šis sprendimas nebuvo teisingas, kadangi skirtingų kriterijų pusiausvyros išlaikymą pasitelkiant šį metodą buvo sudėtinga įgyvendinti, o dėl pasirinktų svorių didžiausią įtaką turėjo atliekami palyginimai. Atsižvelgiant į kursinio projekto trūkumus, šiame darbe pasirinkta naudoti daugiakriterinį genetinį algoritmą, kuris padėtų sugeneruoti efektyvius Šelo algoritmo variantus bei išlaikytų tinkamą pusiausvyrą tarp atliekamų operacijų ir veikimo laiko.

Keliama tokia **hipotezė**:

Pasitelkiant genetinius algoritmus įmanoma sukonstruoti efektyvius Šelo algoritmo variantus, kurių vidutinis veikimo laikas būtų mažesnis nei šiuo metu žinomų variantų.

Siekiant patikrinti iškeltą hipotezę, reikia atlikti šiuos uždavinius:

- Išanalizuoti Šelo algoritmą ir jo variantus remiantis literatūra;
- Išanalizuoti Šelo algoritmo veikimą atliekant eksperimentus;
- Nustatyti kriterijus Šelo algoritmo variantų efektyvumui įvertinti;
- Realizuoti genetinį algoritmą Šelo algoritmo variantų generavimui;
- Pasitelkiant realizuotą genetinį algoritmą sugeneruoti Šelo algoritmo variantus;
- Eksperimentiškai palyginti sugeneruotų ir pateiktų literatūroje Šelo algoritmo variantų efektyvumą.

Šiame darbe atlikta:

- Atlikta Šelo algoritmo ir jo variantų literatūros analizė;
- Atlikti eksperimentai, kurie leido nustatyti šakų spėjimo įtaką Šelo algoritmo veikimo laikui su nedideliais duomenų dydžiais;
- Realizuotas genetinis algoritmas, leidžiantis generuoti Šelo algoritmo variantus, kurie išlaiko pusiausvyrą tarp veikimo laiko ir atliekamų operacijų;
- Pasitelkiant realizuotą genetinį algoritmą sugeneruoti Šelo algoritmo variantai, leidžiantys efektyviai rikiuoti įvairius duomenų dydžius;
- Atlikti eksperimentai, kuriais patvirtinta, jog sugeneruoti Šelo algoritmo variantai su pasirinktais duomenų dydžiais veikia greičiau, nei žinomi variantai;

Darbas remiasi tokiomis prielaidomis:

- Atliekant Šelo algoritmo variantų generavimą ir tarpusavio palyginimą rikiuojamų duomenų palyginimo ir priskyrimo sudėtingumas laiko atžvilgiu yra $O(1)$;
- Atliekant Šelo algoritmo variantų tarpusavio palyginimą yra pakankama aprėpti geriausiai žinomus variantus (visų literatūroje pateiktų variantų tarpusavio palyginimas reikalautų atskiro tyrimo);
- Šelo algoritmo variantų tarpusavio palyginimui pasirinkti duomenų rinkiniai pakankamai tiksliai atspindi dažniausiai praktikoje sutinkamus duomenų rinkinius.

1. Šelo algoritmas

1.1. Klasikinis Šelo algoritmas

Šelo algoritmas yra vienas iš seniausių ir geriausiai žinomų rikiavimo algoritmų. Šelo algoritmas yra paremtas palyginimu, adaptyvus, nestabilus ir nenaudojantis papildomos atminties. Yra įrodyta, kad Šelo algoritmo laiko sudėtingumo blogiausiu atveju apatinė riba yra $\Omega(\frac{n\,log^2\,n}{log\,log\,n^2})$ [PPS92], tad jis nėra asimptotiškai optimalus. Šio algoritmo laiko sudėtingumo analizė vidutiniu atveju yra labai sudėtinga ir lieka atvira problema [Ciu01; RB13]. Nepaisant sudėtingos analizės, Šelo algoritmas yra nesunkiai įgyvendinamas ir gana lengvai suprantamas. Tai įrodo ir pseudokodas, pateikiamas 1 algoritme.

Algorithm 1 Šelo algoritmas

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

1.2. Optimizuotas Šelo algoritmas

Reikia pastebėti, jog klasikinis Šelo algoritmas nėra optimizuotas ir tam tikrais atvejais atlieka nereikalingas operacijas. Jei vykdant vidinį Šelo algoritmo ciklą esamas elementas $S[j]$ nėra mažesnis už elementą $S[j-h]$, $while$ ciklas kuriame atliekama esminė rikiavimo logika nebus vykdomas, tad du priskyrimai bus atlikti veltui. Optimizuota Šelo algoritmo versija [RB13] patikrina ar esamas elementas yra mažesnis už elementą $S[j-h]$ ir tik tada vykdo vidinį ciklą. Tai leidžia sumažinti atliekamų priskyrimų skaičių 40-80% ir sumažinti veikimo laiką apytiksliai 20%, lyginant su klasikine implementacija. Toliau darbe naudojama ir analizuojama tik optimizuota Šelo algoritmo implementacija.

Algorithm 2 Optimizuotas Šelo algoritmas

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

1.3. Šelo algoritmo variantai

Šelo algoritmas pasižymi variantų gausa. Šelo algoritmą iš esmės galima laikyti tarpų sekos iteravimu, su kiekvienu iš tarpų atliekant tam tikro posekio rikiavimą. Tad norint patobulinti Šelo algoritmą galima keisti tiek posekių formavimą (naudojant kitokią tarpų seką), tiek posekių rikiavimo logiką. Šioje darbo dalyje nagrinėjami tie variantai, kurie nuo originalios versijos skiriasi taikoma posekių rikiavimo logika.

Siekiant supaprastinti skirtingų Šelo algoritmo variantų palyginimą, galima išskirti dvi sudėtines kiekvieno varianto dalis: visiems variantams bendrą struktūrą ir konkrečiam variantui būdingą posekių rikiavimo logiką. Šelo algoritmo variantams būdinga struktūra, kurioje iteruojant per tarpus rikiuojami posekiai, toliau vadinama Šelo algoritmo karkasu, o karkaso viduje atliekama rikiavimo logika - perėjimu (angl. pass). Optimizuoto Šelo algoritmo taikomas perėjimas toliau vadinamas įterpimo perėjimu.

Dobosiewicz vienas pirmųjų pastebėjo, jog pasitelkiant Šelo algoritmo karkasą ir pakeitus rikiavimo logiką 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 šis perėjimas dažniausiai vadinamas burbuliuko perėjimu (angl. bubble pass) [Sed96]. Jo pseudokodas pateikiamas 3 algoritme.

Algorithm 3 Burbuliuko perėjimas

```
1: for i ← 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
```

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). Šio algoritmo idėjas integravus į Šelo algoritmo karkasą taip pat gaunamas gana efektyvus algoritmas [IS86]. Jo taikomas perėjimas toliau vadinamas supurtymo perėjimu (angl. shake pass), o šio perėjimo pseudokodas yra pateikiamas 4 algoritme.

Algorithm 4 Supurtymo perėjimas

```
1: for i ← 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 ← 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 mūrijimo rikiavimas (angl. 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. Šią idėją nesunkiai galima pritaikyti ir Šelo algoritmo karkaso viduje, kintamuoju pakeitus originaliame algoritme taikytą tarpą 1 [Lem94]. Šis perėjimas toliau vadinamas mūrijimo perėjimu (angl. brick pass), o jo pseudokodas yra pateikiamas 5 algoritme.

Algorithm 5 Mūrijimo perėjimas

1: **for** $i \leftarrow gap$ **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** $i \leftarrow 0$ **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. Eksperimentinė Šelo algoritmo analizė

Norint paruošti greitai veikiantį Šelo algoritmo variantą arba taisykles tokiam algoritmui gauti, vertinga žinoti, kodėl vieni variantai veikia greičiau nei kiti. Kursiniame projekte buvo pastebėta, jog Šelo algoritmo variantų atliekamų operacijų skaičius ne visada koreliuoja su veikimo laiku. Dėl šiuolaikinių kompiuterių veikimo subtilybių algoritmai, kurie atlieka daugiau palyginimo ir priskyrimo operacijų, kai kuriais atvejais gali veikti greičiau, nei operacijų atžvilgiu optimalesni algoritmai. Šiame skyriuje siekiama nustatyti, kokie veiksniai labiausiai įtakoja Šelo algoritmo veikimo laiką.

Šiuolaikinių kompiuterių architektūros yra labai sudėtingos ir tai apsunkina teorines veikimo laiko prognozes. Tai lemia įvairios šiuolaikinių kompiuterių architektūrose pasitelkiamos strategijos: instrukcijų vykdymas ne iš eilės (siekiant pilnai išnaudoti procesoriaus ciklus), duomenų saugojimas kelių lygių talpykloje (siekiant panaikinti atminties delsą) ir šakų nuspėjimas (siekiant tęsti intrukcijų vykdymą nelaukiant, kol sąlygos rezultatas bus žinomas). Jei duotas algoritmas netinkamai išnaudoja šias mašinos galimybes, praktinis jo efektyvumas gali stipriai nukentėti. Todėl bene geriausias būdas nustatyti tikėtiną praktinį algoritmo efektyvumą yra atlikti eksperimentus ir stebėti mašinos veikimą. Žinoma, tokiu būdu gauti rezultatai priklauso nuo duotos mašinos techninių parametrų ir įvairių kitų veiksnių, tačiau didžioji dalis šiuolaikinių kompiuterių pasitelkia tas pačias strategijas, tad atliekant kelių algoritmų veikimo palyginimą skirtinguose kompiuteriuose, santykiniai rezultatų skirtumai neturėtų reikšmingai skirtis.

Pirmiausia buvo nutarta pamatuoti kelių skirtingų Šelo algoritmo variantų veikimo laiką ir atliekamas operacijas, siekiant nustatyti kaip tarpų sekos įtakoja šiuos kriterijus. Vykdant eksperimentus buvo nuspręsta tirti Šelo algoritmo veikimą rikiuojant 64 elementus. Tai pakankamai svarbus scenarijus, kadangi jis tokiu principu dažnai pasitelkiamas hibridiniuose rikiavimo algoritmuose [Aut09; Sew10]. Šiam eksperimentų etapui buvo pasirinkti du Šelo algoritmo variantai, kur pirmasis naudoja tarpų seką $\{1\}$ kas prilygsta rikiavimui įterpimu, o antrasis naudoja sutrumpintą Ciura tarpų seką $\{57,23,10,4,1\}$.

Eksperimentai buvo atliekami naudojant kompiuterį su 2.70 GHz Intel(R) Core(TM) i7-10850H procesoriumi, 32 GB operatyviosios atminties ir Windows 11 OS. Tyrimui reikalinga programinė įranga yra prieinama tik Linux OS vartotojams, tad eksperimentai buvo vykdomi WSL aplinkoje su g++ 9.3 kompiliatoriumi naudojant `-O2` optimizacijos lygmenį. Reikia pastebėti, jog WSL veikia virtualioje mašinoje, tad tai galėjo paveikti gautus rezultatus. Kadangi praktinis efektyvumas reikšmingai priklauso nuo konkrečios implementacijos, prieduose yra pateikiama eksperimentams naudota Šelo algoritmo implementacija C++ kalba. Visi eksperimentai buvo vykdomi 5000000 kartų rikiuojant atsitiktinius sveikaskaitinius duomenis.

Pagal 1 lentelėje matomus rezultatus galima pastebėti, jog Šelo algoritmas su Ciura tarpų seka atliko žymiai mažiau operacijų, tačiau taip pat veikė reikšmingai lėčiau nei variantas su tarpų seka $\{1\}$. Šiuo atveju rezultatai yra visiškai neintuityvūs, tačiau aiškiai pastebima, jog atliekamos operacijos ir veikimo laikas nekoreliuoja. Norint juos paaiškinti, reikia surinkti daugiau duomenų.

1 lentelė. Eksperimentų rezultatai matuojant veikimo laiką ir operacijas, kai $N = 64$

Tarpų seka	palyginimai	priskyrimai	laikas (ciklais)
$\{1\}$	1067	1126	2544
Ciura	403	439	3873

Šiuolaikiniai kompiuteriai dažniausiai suteikia galimybes stebėti rodiklius, kurie labiausiai įtakoja veikimo laiką. Tam yra pasitelkiami atskiri registrai, kuriuose saugomas pvz. neatspėtų šakų skaičius. Kadangi rankiniu būdu tikrinti registrų reikšmes yra gana sudėtinga, buvo pasirinkta esminių rodiklių stebėjimui naudoti `perf` profiliuotoją. Kadangi `perf` stebi visą programos veikimo laikotarpį, atliekant matavimus pirmiausia buvo išmatuojama kiek operacijų, ciklų ir pan. reikia vien duomenų generavimui, vėliau juos atimant iš rodiklių, gautų atliekant rikiavimą.

Naudojant `perf` surinkti duomenys pateikiami 2 lentelėje. Kai kurie šiuo atveju neesminiai kriterijai nėra pateikiami, kadangi juos geriau atspindi kombinuoti kriterijai (pvz. neverta pateikti viso atliktų šakų ir neatspėtų šakų, kai galima pateikti neatspėtų šakų santykį). Reikia pastebėti, jog instrukcijos įvykdomos per ciklą iš dalies priklauso nuo procesoriaus gebėjimo atspėti šaką, kadangi jos neatspėjus tenka atstatyti prieš spėjimą buvusią būseną, kas priklausomai nuo procesoriaus užtrunka nuo 10 iki 30 ciklų. Taip pat skirtingų variantų atliekamos instrukcijos nėra vienodai prasmingos - su tarpu 1 norint perstumti elementą per 3 pozicijas reikės atlikti daugiau instrukcijų, nei tą patį atliekant su tarpu 3, nors galutinis rezultatas bus toks pats. Tad vien instrukcijos per ciklą neduoda pilno vaizdo apie rikiavimo spartą ir į tai būtina atsižvelgti. Analizuojant gautus rezultatus galima pastebėti, kad variantas su tarpų seka $\{1\}$ turi apytiksliai 6 kartus mažesnį neatspėtų šakų santykį bei įvykdo apytiksliai 3 kartus daugiau instrukcijų per ciklą. Todėl, nors jis ir nėra efektyvus teorine prasme (atlieka daug palyginimų ir priskyrimų), praktikoje jis veikia reikšmingai greičiau, kadangi geriau išnaudoja mašinos galimybes. Tiesa, rikiuojant sudėtingus duomenų tipus variantas su Ciura tarpų seka turėtų gauti palankesnius veikimo laiko rezultatus, kadangi atlieka mažiau operacijų.

2 lentelė. Rezultatai gauti naudojant perf, kai $N = 64$

Tarpų seka	instrukcijos/ciklai	neatspėtos šakos
$\{1\}$	2.35	3.44%
$\{57,23,10,4,1\}$	0.71	21.30%

Pirmiausia reikėtų nustatyti, nuo ko priklauso šakų atspėjamumas naudojant šias tarpų sekas. Reikia pastebėti, jog rikiavimas įterpimu yra vienas palankiausių $O(n^2)$ algoritmų šakų nuspėjimui ir rikiuojant neatspėjama $O(n)$ šakų [BG05]. Tai lemia kelios priežastys: išorinio ciklo sąlyga yra lengvai nuspėjama, kadangi ji bus patenkinta tik perėjus visus elementus; vidiniame cikle pirmiausia tikrinama, ar nepasiekta rikiuojamų duomenų pradžia (tai teisinga tik blogiausiu atveju), o kita šaka priklauso nuo elementų palyginimo rezultato (sunkiausiai nuspėjama, kadangi priklauso

nuo duomenų). Reikia paminėti, jog rikiavimo įterpimu vidinis ciklas įprastai yra vykdomas keletą iteracijų (norint perstumti elementą per n pozicijų, reikia atlikti tiek pat iteracijų) ir todėl elementų palyginimo šaka yra pakankamai lengvai nuspėjama. Šelo algoritmo šakų nuspėjamumas priklauso nuo naudojamos tarpų sekos, tačiau jis įprastai būna reikšmingai prastesnis nei rikiavimo įterpimu [BG05]. Biggar et al. pastebi, jog su Gonnet tarpų seka Šelo algoritme per vieną iteraciją kiekvienas elementas yra pastumiamas apytiksliai per $0.9744 * h$ [BNW+08]. Tai reiškia, kad vidinis ciklas, kuriame vykdomas rikiavimas įterpimu su tam tikru tarpu, vidutiniškai atlieka mažiau nei vieną iteraciją, kas lemia, jog vidiniame cikle vykdomo elementų palyginimo šaką yra sudėtinga nuspėti.

Taip pat verta pastebėti, jog kiekvienas papildomas tarpas padidina neatspėtų šakų santykį. Kaip pastebima [BNW+08], Šelo algoritmo neatspėtų šakų skaičius priklauso nuo atliekamų perėjimų skaičiaus. To paaiškinimas yra gana paprastas - rikiavimas įterpimu sukelia $O(n)$ neatspėtų šakų, tad vienas perėjimas, kuriame atliekamas posekio rikiavimas įterpimu taip pat sukels $O(n)$ neatspėtų šakų [BNW+08]. Tad p perėjimų atliekantis Šelo algoritmas viso sukels $O(np)$ neatspėtų šakų.

Kadangi prieš tai nagrinėtos tarpų sekos pastebimai skiriasi, buvo pasirinkta ištirti tarpų sekas, kurios yra gana paprastos ir tokio paties ilgio: $\{1, 16\}$, $\{1, 32\}$, $\{1, 48\}$. Atliekant eksperimentus gauti rezultatai pateikiami 3 lentelėje. Remiantis gautais rezultatais, geriausią šakų nuspėjamumą turi tarpų seka $\{1, 48\}$. Taip pat galima pastebėti, kaip mažėjant neatspėtų šakų skaičiui auga instrukcijų per ciklą skaičius bei didėja atliekamų operacijų skaičius. Nors su tarpų seka $\{1, 48\}$ šakos nuspėjamos tiksliau bei atliekama daugiau instrukcijų per ciklą, su tarpų seka $\{1, 32\}$ veikimo laikas yra geresnis. Tad vien šakų nuspėjimo tikslumas ir instrukcijų įvykdomų per ciklą kiekis pilnai nenusako veikimo spartos.

3 lentelė. Eksperimentų rezultatai, kai $N = 64$

Tarpų seka	instrukcijos/ciklai	neatspėtos šakos	palyginimai	priskyrimai	laikas (ciklais)
$\{1, 16\}$	1.29	9.94%	578	656	2822
$\{1, 32\}$	1.74	6.19%	753	823	2579
$\{1, 48\}$	1.93	5.20%	825	888	2618

Vertinant gautus rezultatus, taip pat pastebima, jog variantai su tarpų sekomis $\{1, 48\}$, $\{1, 32\}$, $\{1, 16\}$ lenkia variantą su Ciura tarpų seka vertinant veikimo laiką ir beveik prilygsta variantui su tarpų seka $\{1\}$. Žinoma, tam pasiekti jie atlieka daugiau operacijų, nei variantas su Ciura tarpų seka, tačiau taip pat atlieka reikšmingai mažiau operacijų nei variantas su tarpų seka $\{1\}$. Tad šiuo atveju pasiekiamas tam tikras balansas tarp atliekamų operacijų ir veikimo laiko.

Nagrinėjamu atveju galima bendrai įvertinti pasirinktų tarpų sekų veikimą. Pavyzdžiui, tarpų seka $\{1, 48\}$, pirmame perėjime atliks 16 iteracijų, kadangi vykdymas pradedamas nuo indekso 48. Panašia logika, tarpų seka $\{1, 32\}$ pirmame perėjime atliks 32 iteracijas. Šios tarpų sekos vidinį ciklą visada vykdys maksimaliai vieną kartą, kadangi pirmo perėjimo tarpai yra pakankamai

dideli ir rikiuojamo posekio ilgis visada bus 2. Tarpų seka $\{1, 16\}$ pirmame perėjime atliks 48 iteracijas, tačiau rikiuojamo posekio ilgis didės kas 16 iteracijų. Remiantis tuo, įmanoma apskaičiuoti maksimaliai pirmame perėjime atliekamų palyginimų skaičių kiekvienai tarpų sekai: $\{1, 48\}$ - $16 * 1 = 16$, $\{1, 32\}$ - $32 * 1 = 32$, $\{1, 16\}$ - $16 * 1 + 16 * 2 + 16 * 3 = 96$. Tad tarpų seka $\{1, 16\}$ pirmame perėjime gali atlikti 6 kartus daugiau elementų palyginimų, nei tarpų seka $\{1, 48\}$.

Tiesa, iš pateiktų rezultatų sunku nuspręsti, kaip priklauso elementų palyginimo šakos nuspėjamumas priklausomai nuo tarpo dydžio. Atlikus preliminarius matavimus rikiuojant vien su tarpais 16, 32 ir 48 nebuvo pastebėta reikšmingų skirtumų (visais atvejais neatspėtų šakų buvo apie 17%). Tad galima teigti, jog tarpų sekų $\{1, 48\}$ ir $\{1, 32\}$ šakų nuspėjamumo rezultatai yra geresni, nes jos pirmame perėjime mažiau aprikiuoja duomenis ir taip atlieka mažiau elementų palyginimo šakų, kas lemia, jog daugiau rikiavimo atliekama su tarpu 1, kur nuspėjamumas yra geresnis. Bendrai vertinant šakų nuspėjamumą Šelo algoritmu rikiuojant nedidelius duomenų dydžius, galima teigti, jog bendras šakų nuspėjamumas priklauso ne tik nuo naudojamų perėjimų kiekio, tačiau ir nuo pasirinktų tarpų dydžio, kadangi nuo jo priklauso, kiek elementų palyginimų bus atlikta su tam tikru tarpu.

3. Šelo algoritmo variantų efektyvumo kriterijai

Įprastai praktinis rikiavimo algoritmų efektyvumas yra vertinamas matuojant jų atliekamų palyginimų ar priskyrimų skaičių. Tai yra pakankamai geri kriterijai norint praktiškai įvertinti duoto algoritmo efektyvumą su tam tikrais duomenų dydžiais, kadangi teorinis laiko sudėtingumas nurodo tik algoritmo sudėtingumo funkcijos augimo greitį ir neatsižvelgia į konstantas, kurios praktikoje taip pat įtakoja veikimo greitį [BG05].

Tiksliam efektyvumo įvertinimui tinkama matuoti tiek atliekamus palyginimus, tiek atliekamus priskyrimus. To priežastis yra gana paprasta - rikiuojant duomenis, kurių palyginimas yra sudėtingas (pvz. simbolių eilutes), algoritmo veikimo laiką stipriau įtakoja jo atliekamų palyginimų skaičiaus. Analogiškas efektas pastebimas ir rikiuojant duomenis, kurių priskyrimas yra sudėtingas. Vertinant Šelo algoritmo variantų efektyvumą būtina matuoti tiek atliekamus palyginimus, tiek atliekamus priskyrimus, kadangi atliekamų palyginimų ir priskyrimų skaičiaus santykis priklauso nuo implementacijos ir augant N nebūtinai artėja prie 1 [RB13].

Vertinant Šelo algoritmo variantų efektyvumą taip pat būtina atsižvelgti ir į veikimo laiką. Savaime suprantama, jog į šį kriterijų verta žvelgti pakankamai kritiškai, kadangi jis priklauso nuo konkrečios algoritmo implementacijos, eksperimentams naudojamos mašinos techninių parametrų ir kompiliatoriaus taikomų optimizacijų lygio. Tačiau tai bene vienintelis kriterijus, leidžiantis įvertinti realų algoritmo praktinį efektyvumą, kas yra ypač svarbu, kadangi naudojant šiuolaikinį kompiuterį labai sunku iš anksto nustatyti, kaip greitai algoritmas veiks praktikoje. Atsižvelgiant į aukščiau pateiktus argumentus, galima teigti, jog algoritmo veikimo laiko įvertis yra svarbus kriterijus įvertinant praktinį Šelo algoritmo varianto efektyvumą.

Remiantis aukščiau pateiktais argumentais, šiame darbe Šelo algoritmo variantai vertinami pagal atliekamų palyginimų skaičių, atliekamų priskyrimų skaičių ir veikimo laiką. Kiekvienas iš šių kriterijų yra vienodai svarbus įvertinant duoto algoritmo efektyvumą, tad svoriai šiems kriterijams nėra taikomi.

4. 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.

4.1. 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 įvertį kaip parametrą. Sprendžiant minimizavimo uždavinį, tinkamumo funkcija taip pat vadinama kainos funkcija. 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.

4.2. 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.

5. Genetinis algoritmas Šelo algoritmo variantų generavimui

Pirmiausia reikėtų aptarti kaip turėtų būti modeliuojamas sprendinys, atspindintis tam tikrą Šelo algoritmo variantą. Laikysime jog Šelo algoritmo variantą sudaro sąrašas porų (p, h), kur p yra skaičius, atitinkantis vieną iš anksčiau darbe aptartų perėjimų tipų, o h - tarpas, su kuriuo rikiuojama tame perėjime. Toliau darbe tokiu būdu modeliuojamą Šelo algoritmo variantą vadinsime chromosoma arba individu, o porą (p, h) - genu. 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 modeliuojamą Šelo algoritmo variantą galima nesunkiai vykdyti iteruojant jo genų sąrašą ir kiekvienai porai (p, h) iškviečiant funkciją atitinkančią jos tipą ir nurodant tarpą su kuriuo rikiuoti.

Taip pat labai svarbu apibrėžti kaip bus inicializuojami pradiniai sprendiniai, kurie bus naudojami genetinio algoritmo vykdymui. Parinkti atsitiktinį perėjimo tipą yra nesudėtinga, kadangi pakanka sugeneruoti atsitiktinį skaičių, atitinkantį vieną iš apibrėžtų tipų. Tuo tarpu greitai sugeneruoti pakankamai efektyvias tarpų sekas yra gana sudėtinga. Tuo tikslu buvo pasirinkta individų inializacijai naudoti geometrines tarpų sekas, kadangi jas gana nesudėtinga generuoti ir jos bendru atveju yra pakankamai efektyvios (didelė dalis efektyvių eksperimentiškai gautų tarpų sekų yra geometrinės). Geometrinių tarpų sekų generavimo eiga yra pateikiama 6 algoritme. Siekiant padidinti individų genetinę įvairovę, taip pat buvo pasirinkta dalį individų inicializuoti tarpų sekomis, kurios generuojamos metodu panašiu į geometrinių tarpų sekų generavimą, koeficientą q parenkant atsitiktinai kiekvienam tarpui. Generuojant geometrines tarpų sekas buvo nuspręsta koeficientą q parinkti atsitiktinai, užtikrinant, jog $1.5 \leq q \leq 7$. Generuojant atsitiktines tarpų sekas taip pat buvo taikomas panašus principas, tačiau šiuo atveju naudotas toks q, kad $1.5 \leq q \leq 12.5$.

Algorithm 6 Geometrinių tarpų sekų generavimas

```
1: procedure GEOMETRIC_GAPS(n, q)
2:     let gaps be an empty set
3:     current ← 1
4:     while current < n do
5:         insert ⌈current⌉ into gaps
6:         current ← current * q
7:     end while
8:     return gaps
9: end procedure
```

Genetinio algoritmo įgyvendinimui buvo pasirinkta naudoti C++ programavimo kalbą ir openGA biblioteką [MAM+17]. C++ buvo pasirinkta dėl veikimo spartos ir patirties sukauptos ruošiant kursinį darbą ir kursinį projektą. OpenGA biblioteka buvo pasirinkta dėl modernumo ir lygiagretaus vykdymo palaikymo.

Nors kriterijai Šelo algoritmo varianto efektyvumui įvertinti buvo apibrėžti praeitame skyriuje, reikalinga plačiau aptarti kaip bus nustatoma duoto individo kaina. Pirmiausia, siekiama, jog gautas algoritmas būtų deterministinis ir visada išrikiuotų duomenis, kadangi tikimybinių rikiavimo algoritmų panaudojamumas yra ribotas. Todėl būtina po rikiavimo suskaičiuoti rikiuojamų duomenų inversijas ir šį kriterijų minimizuoti. Taip pat reikia, jog pats inversijų skaičiavimas neužtruktų per ilgai, kadangi tai apsunkintų genetinio algoritmo vykdymą. Šiuo tikslu buvo nuspręsta inversijų skaičiavimui pasitelkti modifikuotą rikiavimo sąlaja algoritmą, kurio sudėtingumas blogiausiu atveju yra $O(n\,log\,n)$. Remiantis darbe iškelta hipoteze taip pat siekiama, jog sugeneruoti algoritmai veiktų greitai, todėl vienas iš esminių kriterijų yra veikimo laikas. Veikimo laiko matavimams buvo nuspręsta naudoti rikiavimo metu procesoriaus atliekamų ciklų skaičių pasitelkiant `RDTSC` instrukciją. Taip pat labai svarbu, jog algoritmo veikimo laikas stipriai nepriklausytų nuo rikiuojamų duomenų specifikos, tad būtina užtikrinti, jog atliekamų palyginimo ir priskyrimo operacijų skaičius būtų pakankamai mažas. Operacijų skaičiavimui buvo pasitelkta projektiniame darbe naudota Element klasė, kuri naudojant palyginimo ir priskyrimo operatorių perkrovimą seka šių operacijų skaičių rikiavimo metu.

Kaip buvo minėta įvade, Šelo algoritmo variantų generavimui buvo pasirinkta naudoti daugiakriterinį genetinį algoritmą. Kadangi daugiakriteriniame GA svoriai netaikomi, buvo nuspręsta kai kuriems kriterijams taikyti prioritetus. Tai reiškia, jog vienam iš kriterijų neatitinkant nustatyto apribojimo, kai kurie kiti kriterijai gali būti ignoruojami tam, kad prioritetinis kriterijus atitiktų apribojimus. Remiantis aukščiau aptartais tikslais, buvo nuspręsta taikyti prioritetus duomenų inversijoms bei veikimo laikui. Laikant jog i yra inversijų skaičius, t - veikimo laikas (ciklais), c - palyginimų skaičius, a - priskyrimų skaičius, o T - tam tikra konstanta, individo kainos funkcija yra apibrėžiama tokia forma:

$$cost(i,t,c,a) = \begin{cases} (i,\infty,\infty,\infty), & \text{if } i > 0 \\ (i,t,\infty,\infty), & \text{if } t > T \\ (i,t,c,a), & \text{otherwise} \end{cases}$$

Algoritmai buvo vertinami rikiuojant atsitiktinius sveikaskaitinius duomenis. Atliekamų matavimų skaičius buvo nustatomas pagal duomenų dydį, kuriam generuojami algoritmai (jei duomenų dydis didelis, atliekama mažiau matavimų). Pirmiausia dėl to, jog su didesniais duomenų dydžiais GA neužtruktų per ilgai, o tuo pačiu ir tam, jog rezultatai gauti su mažesniais duomenų dydžiais būtų tikslesni.

Siekiant pagerinti gautų rezultatų kokybę buvo įgyvendinta individų atranka. Šiuo tikslu buvo atmetami individai, kurių vidutinis duomenų inversijų skaičius po rikiavimo didesnis nei $fracN10$. Taip pat buvo atmetami individai, kurių vidutinis veikimo laikas buvo didesnis nei $2\,T$.

Individų mutacija buvo įgyvendinta pasirenkant atsitiktinį individo geną ir atsitiktinai pakeičiant jo perėjimo tipą kuriuo nors kitu. Individų rekombinacija buvo vykdoma tolygia strategija, kur dviejų tėvų genai turi vienodą tikimybę būti perduoti vaikiniam individui. Siekiant išvengti

netinkamų sprendinių, rekombinacijos operatoriumi gauto naujo individo genai buvo išrikiuojami pagal tarpą naudojamą rikiavimui.

Atlikus pradinius eksperimentus ir preliminariai įvertinus gautų sprendinių tinkamumą buvo pasirinkta GA taikyti tokius parametrus: populiacija - 250, mutacijos tikimybė - 0.1, rekombinuojama populiacijos dalis - 0.4. GA buvo nurodyta sustoti įvykdžius 100 iteracijų.

6. Šelo algoritmo variantų generavimas

Šiame skyriuje siekiama sugeneruoti Šelo algoritmo variantus, kurie leistų rikiuoti duomenis greičiau nei žinomi variantai. Genetinio algoritmo esminiai veikimo principai buvo apibrėžti praeitame skyriuje, tačiau konstantos T, apibrėžiančios apribojimus veikimo laikui, reikšmė nebuvo apibrėžta. Iš esmės T reikšmė priklauso nuo duomenų dydžio - jei rikiuojama daugiau elementų, reikėtų pasirinkti didesnę T reikšmę. Natūralu būtų siekti apibrėžti funkciją f, kuri atitinkamam duomenų dydžiui grąžintų reikiamą T. Tačiau tam reikėtų funkcijos, nusakančios apytikslį įvairių Šelo algoritmo variantų vidutinį veikimo laiką. Kadangi tai įgyvendinti yra gana sudėtinga, buvo pasirinktas paprastesnis metodas - kiekvienam naudojamam duomenų dydžiui ištirti Šelo algoritmo su Ciura tarpų seka veikimo laiką ir laikyti, jog tai yra T.

6.1. Šelo algoritmo variantų generavimas, kai $N = 128$

Šiame poskyryje siekiama sugeneruoti Šelo algoritmo variantus, kurie leistų rikiuoti nedidelius duomenų dydžius greičiau nei žinomi variantai. Algoritmų generavimui pasirinkta naudoti $T = 9000$. Genetinis algoritmas buvo įvykdytas 50 kartų, viso buvo sugeneruota 890 skirtingų variantų.

Atsižvelgiant į surinktus duomenis apie veikimo laiką ir atliekamas operacijas buvo atrinkti 3 tinkamiausi variantai: A1, A2 ir A3. Kadangi gauti algoritmai dažniausiai naudoja įterpimo perėjimą, toliau glaustumo dėlei bus nusakomos tik tarpų sekos ir tarpai, su kuriais naudojami kitokie perėjimai.

- A1: $1, 6, 52$, su didžiausiu tarpu naudojamas burbuliuko perėjimas
- A2: $1, 7, 92$, su didžiausiu tarpu naudojamas supurtymo perėjimas
- A3: $1, 10, 76$, su didžiausiu tarpu naudojamas burbuliuko perėjimas

Galima pastebėti tendenciją, jog visi gauti algoritmai naudoja ne įterpimo perėjimą su didžiausiu tarpu. Kaip buvo minėta viename iš praeitų skyrių, su dideliais (santykinai duomenų dydžiui) tarpais atlikti įterpimo perėjimą nėra prasminga, tad panašu, jog kainos funkcija taip pat privedė individus prie šios optimizacijos taikymo.

6.2. Šelo algoritmo variantų generavimas, kai $N = 1024$

Šiame poskyryje siekiama sugeneruoti Šelo algoritmo variantus, kurie leistų rikiuoti vidutinius duomenų dydžius greičiau nei žinomi variantai. Algoritmų generavimui pasirinkta naudoti $T = 125000$. Genetinis algoritmas buvo įvykdytas 50 kartų, viso buvo sugeneruota 690 skirtingų variantų.

Atsižvelgiant į surinktus duomenis apie veikimo laiką ir atliekamas operacijas buvo atrinkti 3 tinkamiausi variantai: B1, B2 ir B3. Kadangi gauti algoritmai naudoja tik įterpimo perėjimus, toliau glaustumo dėlei bus nusakomos tik jų tarpų sekos.

- B1: $1, 4, 17, 40, 162$
- B2: $1, 5, 19, 155$
- B3: $1, 8, 26, 97$

Galima pastebėti, jog gautų algoritmų tarpų sekos yra žymiai trumpesnės, nei įprastai sutinkamų Šelo algoritmo variantų. Kaip pastebima [RB13], vertinant veikimo laiką dažnai tinkamesnės būna geometrinės tarpų sekos, kur q yra didesnis. Reikia pastebėti, kad kuo didesnis q, tuo mažiau perėjimų bus atliekama tam tikro duomenų dydžio išrikiavimui, o kiekvienas papildomas perėjimas didina neatspėtų šakų santykį bei reikalauja papildomo darbo.

6.3. Šelo algoritmo variantų generavimas, kai $N = 8192$

Šiame poskyryje siekiama sugeneruoti Šelo algoritmo variantus, kurie leistų rikiuoti didelius duomenų dydžius greičiau nei žinomi variantai. Algoritmų generavimui pasirinkta naudoti $T = 1500000$. Genetinis algoritmas buvo įvykdytas 20 kartų, viso buvo sugeneruoti 322 skirtingi variantai.

Atsižvelgiant į surinktus duomenis apie veikimo laiką ir atliekamas operacijas buvo atrinkti 3 tinkamiausi variantai: C1, C2 ir C3. Kadangi gauti algoritmai dažniausiai naudoja įterpimo perėjimą, toliau glaustumo dėlei bus nusakomos tik tarpų sekos ir tarpai, su kuriais naudojami kitokie perėjimai.

- C1: $1, 4, 11, 25, 104, 264, 615, 1794$, su antru pagal dydį tarpu naudojamas supurtymo perėjimas
- C2: $1, 4, 11, 25, 104, 356, 1152, 1794$, su antru pagal dydį tarpu naudojamas burbuliuko perėjimas
- C3: $1, 6, 19, 53, 187, 760$

7. Šelo algoritmo variantų efektyvumo tyrimo aplinka

7.1. Duomenų rinkiniai

Norint tinkamai įvertinti įvairių Šelo algoritmo variantų efektyvumą, yra būtina atlikti eksperimentus naudojant įvairius duomenų rinkinius. Idealiu atveju, tyrime taip pat naudojami skirtingi duomenų tipai (skaičiai, simbolių eilutės, etc.), kas leidžia įvertinti atliekamų palyginimų ir priskyrimų įtaką rikiavimo laikui. Šiame darbe buvo pasirinkta naudoti sveikaskaitinius duomenis, generuojamus juos išdėstant 6 skirtingais būdais. Šie duomenų išsidėstymo būdai yra apibrėžiami žemiau.

Dažniausiai literatūroje tiriant rikiavimo algoritmų efektyvumą yra sutinkami atsitiktiniai išsidėstę duomenys. Šiame darbe buvo pasirinkta naudoti kelis šio duomenų išsidėstymo variantus. Variantas `shuffled_int` pirmiausia sugeneruoja sąrašą pavidalo $\{0, \ldots, size-1\}$, tada naudojant `std::shuffle` funkciją šis sąrašas yra išmaišomas. Reikia paminėti, jog `std::shuffle` garantuoja, kad ją įvykdžius elementų tvarka yra atsitiktinė ir kiekvienas skirtingas įmanomas elementų derinys turi vienodą tikimybę pasirodyti galutiniame sąraše.

Variantas `shuffled_mod_sqrt` pirmiausia sugeneruoja sąrašą, kur kiekviena reikšmė yra formos $a \pmod{\lfloor\sqrt{size}\rfloor}$ ir $a \in \{0, \ldots, size-1\}$, o tada naudojant `std::shuffle` šis sąrašas yra išmaišomas. Šis duomenų rinkinys leidžia įvertinti, kaip veikia rikiavimo algoritmai, kai yra pakankamai daug dublikatų.

Variantas `descending` sugeneruoja sąrašą pavidalo $\{size-1, \ldots, 0\}$, t.y. elementai yra išdėstomi mažėjimo tvarka. Šiuo atveju laikome, jog algoritmai rikiuoja didėjimo tvarka, tad toks duomenų rinkinys leidžia įvertinti, kaip veikia tiriami rikiavimo algoritmai, kai duomenys yra išdėstyti nepalankia (atvirkščia) tvarka. Tuo pačiu reikia pridurti, jog rinkinys, kur elementai išdėstyti didėjimo tvarka, šio tyrimo kontekste neturėtų prasmės ir todėl nėra įtraukiamas, kadangi visiems Šelo algoritmo variantams tai yra geriausias atvejis (nereikia vykdyti jokių sukeitimų). Lygiai tuo pačiu principu buvo pasirinkta neįtraukti ir duomenų rinkinio, kur visi elementai yra lygūs.

Variantas `partially_sorted` sugeneruoja sąrašą pavidalo $\{0, \ldots, size-1\}$, tada naudojant `std::shuffle` jį išmaišo, o po to dalinai jį išrikiuoja pasitelkiant `std::partial_sort` funkciją. Šiuo atveju dalinis rikiavimas atliekamas tokiu būdu, kad pirma pusė elementų yra išrikiuoti, o antra pusė elementų yra išdėstyti atsitiktine tvarka. Šis duomenų rinkinys leidžia įvertinti, kaip veikia rikiavimo algoritmai, kai į vieną yra sujungiami du sąrašai, kur pirmas sąrašas yra išrikiuotas, o antras ne. Galima nesunkiai įsivaizduoti tokio duomenų išsidėstymo pasitaikymo atvejį praktikoje - tarkime, programa turi duomenis kurie yra išrikiuoti, gauna daugiau duomenų, šiuos sąrašus sujungia ir vėl išrikiuoja, siekiant išlaikyti jų tvarką.

Variantas `merge` sugeneruoja du sąrašus pavidalo $\{0, \ldots, \lfloor size/2 \rfloor\}$ ir juos sujungia į vieną. Šis duomenų rinkinys leidžia įvertinti, kaip veikia rikiavimo algoritmai, kai vieną sąrašą sudaro du išrikiuoti nuoseklūs posekiai. Praktinis šio duomenų rinkinio naudojimo atvejis yra gana panašus į `partially_sorted` duomenų rinkinio. Reikia pastebėti, jog šis atvejis su praktikoje sutinkamais

duomenimis yra gana dažnas, dėl to pvz. Timsort algoritmas, kuris sugeba išnaudoti duomenyse pasitaikančius išrikiuotus posekius efektyvesniam rikiavimui, yra taip plačiai naudojamas [ANP15].

Variantas `push_min` sugeneruoja sąrašą pavidalo $\{1, \ldots, size - 1, 0\}$, t.y. visi elementai, išskyrus paskutinį, yra išrikiuoti. Šis duomenų rinkinys leidžia įvertinti, kaip veikia rikiavimo algoritmai, kai į išrikiuoto sąrašo galą yra pridedamas dar vienas elementas ir siekiama vėl užtikrinti duomenų tvarką. Šis atvejis yra gana nepalankus, kadangi neišrikiuotas yra tik vienas elementas, tačiau jį yra būtina perstumti į kitą sąrašo galą. Galima nesunkiai įsivaizduoti tokio duomenų išsidėstymo pasitaikymo atvejį praktikoje, ypač jei siekiant išlaikyti duomenų tvarką dėl ribotos standartinės bibliotekos ar žinių trūkumo pasirenkama naudoti ne prioritetinę eilę, o duomenis kaskart išrikiuoti.

7.2. Tyrimo metodika

Atliekant tyrimą buvo pasirinkta įvertinti sugeneruotų ir geriausiai žinomų Šelo algoritmo variantų efektyvumą. Tuo tikslu buvo pasirinkta gautus variantus lyginti su optimizuotu Šelo algoritmu naudojant šias tarpų sekas:

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

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 8192 (maksimalų tirtą duomenų dydį).

Kadangi darbe buvo generuojami algoritmai, leidžiantys efektyviai rikiuoti tris skirtingus duomenų dydžius, tyrimą sudaro trys algoritmų rinkiniai. Pirmą rinkinį sudaro A1, A2, A3 algoritmai ir literatūroje pateikti variantai, antrą rinkinį sudaro B1, B2, B3 algoritmai ir literatūroje pateikti variantai, trečią rinkinį C1, C2, C3 algoritmai ir literatūroje pateikti variantai. Su kiekvienu iš šių rinkinių eksperimentai atliekami naudojant tris skirtingus duomenų dydžius: $\frac{N}{2^2}$, $\frac{N}{2}$, N. Čia N yra duomenų dydis, su kuriuo pasitelkiant GA buvo sugeneruota dalis to rinkinio algoritmų.

Atliekant efektyvumo tyrimą buvo pasirinkta kiekvienai algoritmo ir duomenų rinkinio porai atlikti matavimus tol, kol praeis 10 sekundžių. Tokiu būdu atliekamų matavimų skaičius automatiškai prisitaiko prie duomenų dydžio ir rezultatai gauti su mažesniais duomenų dydžiais yra tikslesni. Siekiant padidinti rezultatų tikslumą, kiekviena algoritmo ir duomenų rinkinio pora prieš atliekant matavimus buvo 2 sekundes apšildoma atliekant rikiavimą ir nefiksuojant rezultatų. Tai leido apšildyti procesoriaus talpyklas ir sumažinti tikimybę, jog procesoriaus veikimo dažnis ženkliai pasikeis atliekant matavimus.

Kiekvienai algoritmo ir atsitiktinio duomenų rinkinio porai taip pat buvo naudojama ta pati pradinė reikšmė, kuri inicializuoja `std::shuffle` naudojamą atsitiktinių skaičių generatorių. Tad iš esmės kiekvienas algoritmas tam tikrame duomenų rinkinyje turėtų rikiuoti tokius pat pradinius duomenis.

Siekiant jog gauti rezultatai būtų statistiškai reikšmingi, buvo pasirinkta naudoti tarpkvartilinio diapazono (angl. interquartile range) metodą. Šis metodas leidžia aptikti nuokrypius, kas leidžia nustatyti nenumatytų veiksnių įtaką gautiems rezultatams. Tai ypač svarbu matuojant veikimo laiką, kadangi jį labiausiai paveikia išorinių veiksnių įtaka. Taip pat tarpkvartilinis diapazonas nurodo rezultatų sklaidą aplink medianą, kas leidžia nustatyti atliktų matavimų paklaidą.

7.3. Tyrimo aplinka

Eksperimentų vykdymui buvo naudojamas kompiuteris su 2.70 GHz Intel(R) Core(TM) i7-10850H procesoriumi, 32 GB operatyviosios atminties ir Windows 11 operacine sistema. Efektyvumo tyrimas buvo įgyvendintas C++ kalba su MSVC 19.16.27043 kompiliatoriumi.

8. Šelo algoritmo variantų efektyvumo tyrimas

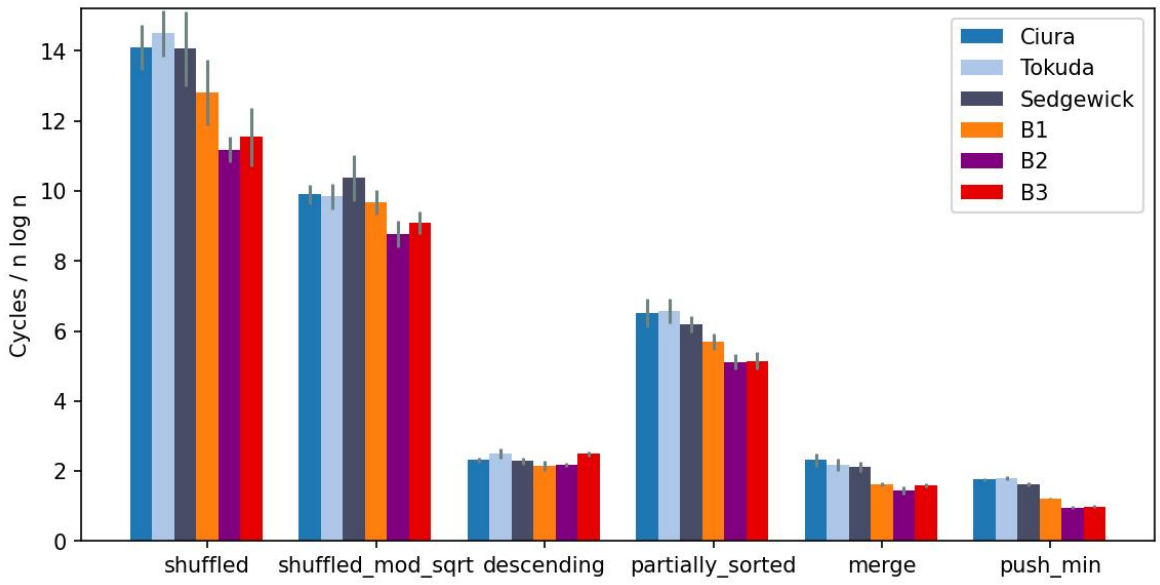
8.1. Efektyvumas priklausomai nuo duomenų rinkinio

8.1.1. Veikimo laikas

Remiantis 1 pav. rezultatais, mažiausiai palankus veikimo laiko prasme yra duomenų rinkinys `shuffled`. Duomenų rinkinys `shuffled` iš tiriamų rinkinių turi didžiausią entropiją, kadangi jį sudaro atsitiktine tvarka išdėstyti ir neturintys dublikatų duomenys. Tokį duomenų rinkinį sudėtinga rikiuoti, kadangi jau išrikiuotų posekių egzistavimo tikimybė yra gana maža ir kiekvienas elementas turi vienodą tikimybę būti bet kurioje iš galimų pozicijų. Rinkinys `shuffled_mod_sqrt` taip pat yra atsitiktinai išdėstomas ir todėl gana nepalankus, nors pastebimi veikimo laikai yra geresni nei su rinkiniu `shuffled`.

Veikimo laiko atžvilgiu labiausiai palankūs yra rinkiniai `merge` ir `push_min`. Šie rinkiniai pasižymi tuo, jog yra dalinai išrikiuoti (`merge` sudaro du išrikiuoti sąrašai, `push_min` turi tik vieną neišrikiuotą elementą). Šelo algoritmas (o tuo pačiu ir tiriami variantai) yra adaptyvus (dalinai išrikiuotus duomenis rikiuoja greičiau), tad šiuo atveju galima pastebėti šios savybės teikiamą naudą.

Su visais duomenų rinkiniais yra pastebimas geresnis GA sugeneruotų variantų veikimo laikas lyginant juos su literatūroje pateiktais variantais. Bene didžiausi rezultatų skirtumai pastebimi su `shuffled` duomenų rinkiniu. Tai galima paaiškinti faktu, jog GA optimizavo variantus rikiuojant atsitiktinius, dublikatų neturinčius duomenis, tad sugeneruoti variantai geriausiai prisitaikė rikiuoti būtent tokius duomenų rinkinius.



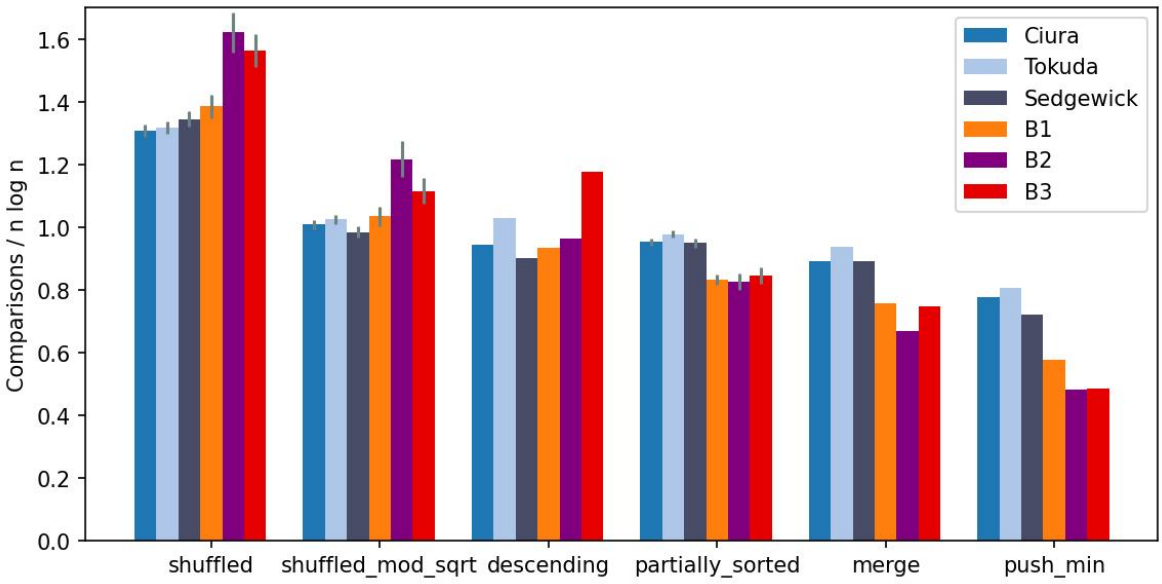
1 pav. Veikimo laikas priklausomai nuo duomenų rinkinio su $N = 1024$

8.1.2. Palyginimai

Vertinant 2 pav. pateikiamus atliekamų palyginimų rezultatus, pastebimos panašios tendencijos, kaip ir analizuojant veikimo laiko rezultatus - mažiausiai palankus yra `shuffled` duomenų rinkinys, labiausiai palankus yra `push_min` duomenų rinkinys.

Taip pat galima pastebėti, jog rinkinys `descending` yra palankesnis veikimo laikui, nei būtų galima tikėtis vertinant atliekamus palyginimus. Nors rikiuojant šį duomenų rinkinį reikia atlikti gana daug operacijų, jis yra lengvai nuspėjamas, kadangi visi elementai yra išrikiuoti (nors ir nepalankia tvarka). Tad beveik visada galima spėti, jog elementas ties pozicija i yra mažesnis už elementą esantį ties pozicija $i - h$ ir atliekant tokius spėjimus neatspėtų šakų skaičius turėtų būti santykinai žemas.

Taip pat galima pastebėti, jog genetinio algoritmo sugeneruoti Šelo algoritmo variantai su duomenų rinkiniais `shuffled`, `shuffled_mod_sqrt`, `descending` atlieka ženkliai daugiau palyginimo operacijų, nei literatūroje pateikti variantai. Iš esmės to buvo galima tikėtis, kadangi gautų variantų veikimo laikas buvo patobulintas paaukojant atliekamų operacijų kiekį. Tačiau su likusiais duomenų rinkiniais šie variantai atlieka pastebimai mažiau palyginimo operacijų. Tai galima paaiškinti tuo, jog sugeneruoti variantai turi mažiau perėjimų, nei literatūroje pateikti variantai. Todėl kai duomenys yra dalinai išrikiuoti, šie algoritmai atlieka mažiau nereikalingų palyginimų, nei literatūroje pateikti variantai.

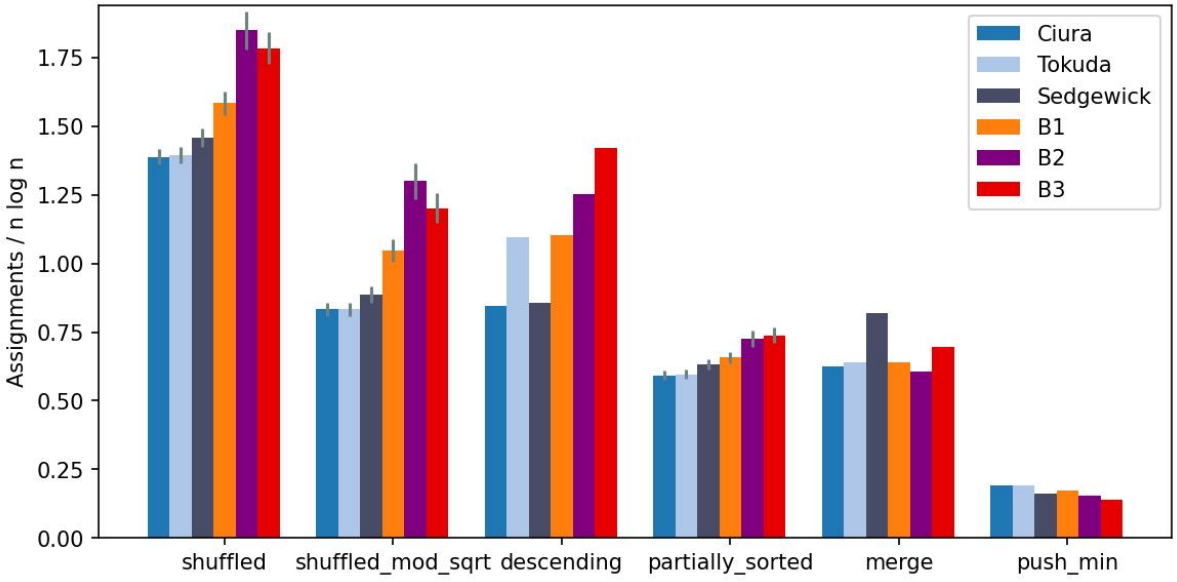


2 pav. Atliekami palyginimai priklausomai nuo duomenų rinkinio su $N = 1024$

8.1.3. Priskyrimai

3 pav. pateikiami atliekamų priskyrimų rezultatai. Kaip ir vertinant atliekamus palyginimus, nepalankiausias yra duomenų rinkinys `shuffled`, labiausiai palankus - rinkinys `push_min`.

Genetinio algoritmo pagalba gauti Šelo algoritmo variantai, kaip ir analizuojant atliekamus palyginimus, su duomenų rinkiniais `shuffled`, `shuffled_mod_sqrt`, `descending` atlieka ženkliai daugiau priskyrimo operacijų, nei literatūroje pateikti variantai. Tiesa, su duomenų rinkiniais `partially_sorted` ir `merge` atliekamų priskyrimų rezultatai yra kiek palankesni, o su duomenų rinkiniu `push_min` sugeneruoti variantai lenkia literatūroje pateikiamus variantus.

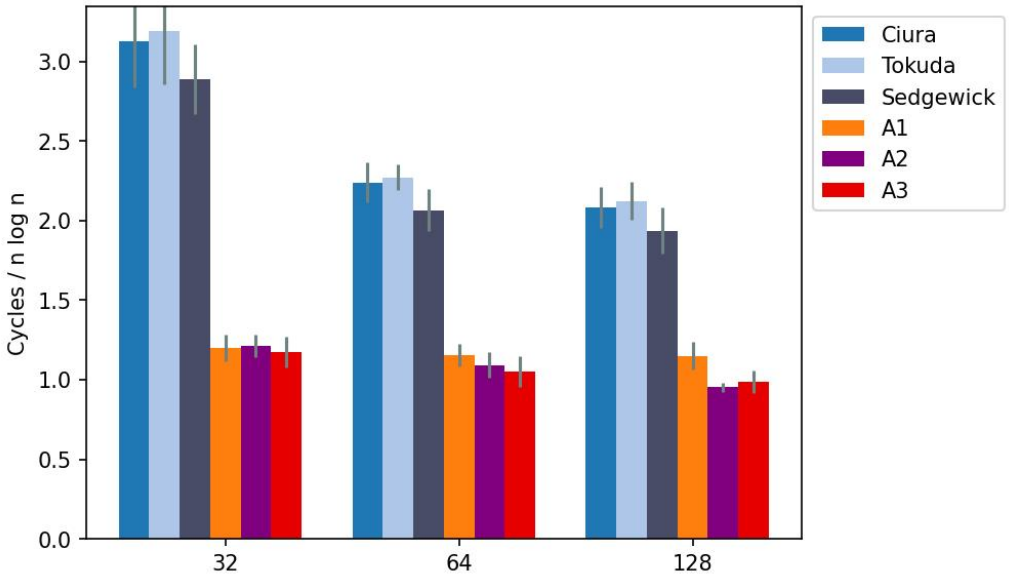


3 pav. Atliekami priskyrimai priklausomai nuo duomenų rinkinio su $N = 1024$

8.2. Efektyvumas priklausomai nuo duomenų dydžio

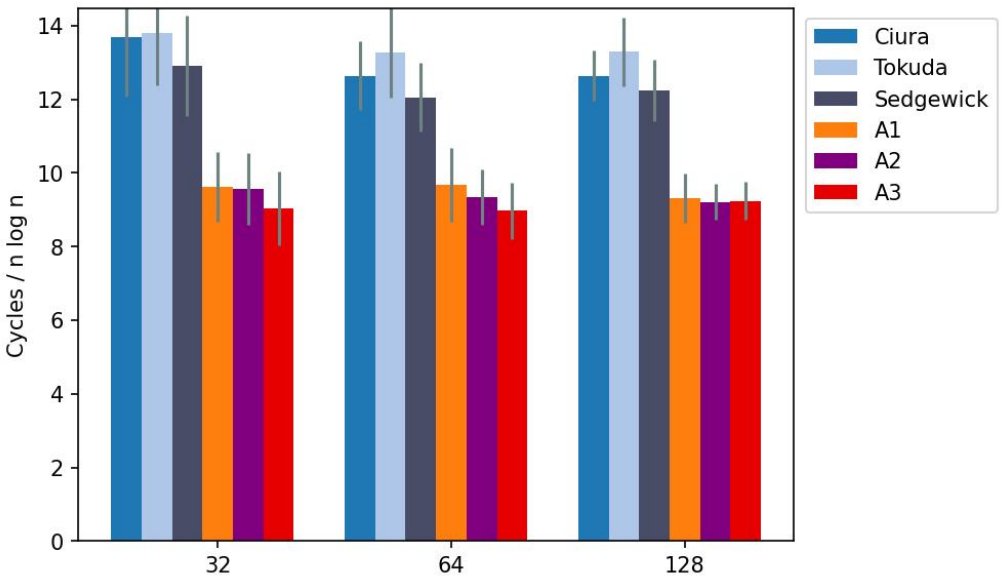
8.2.1. Veikimo laikas

4 pav. pateikiami veikimo laiko rezultatai su duomenų rinkiniu `push_min` ir nedideliais duomenų dydžiais. Galima pastebėti, jog augant duomenų dydžiui sugeneruotų algoritmų veikimo laikas beveik nesikeičia, tuo tarpu literatūroje pateikti variantai su mažiausiu duomenų dydžiu veikia daugiau nei 2 kartus lėčiau, augant duomenų dydžiui veikimo laikas gerėja. Tai galima paaiškinti tuo, jog sugeneruoti algoritmai su $N = 32$ atliks vos du perėjimus, tuo tarpu literatūroje pateikiami variantai atliks 4. Didžiausią įtaką veikimo laikui šiuo atveju turi mažiausi tarpai, kadangi su didesniais tarpais yra atliekama mažiau iteracijų. Kadangi šis duomenų rinkinys yra beveik išrikiuotas, papildomi du perėjimai šiuo atveju pastebimai sulėtina veikimą.



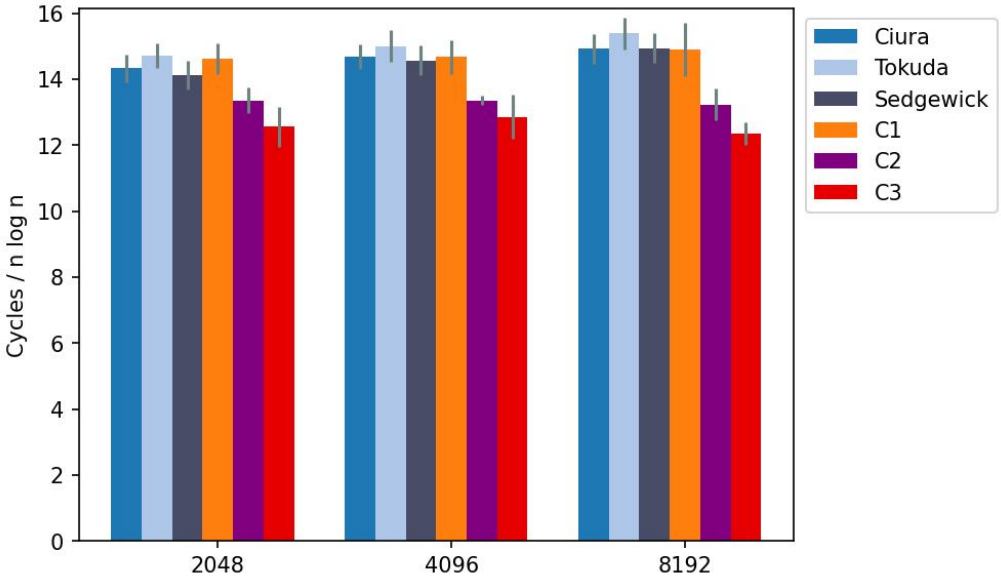
4 pav. Veikimo laikas su duomenų rinkiniu push_min ir $N \leq 128$

5 pav. pateikiami veikimo laiko rezultatai su duomenų rinkiniu `shuffled` ir nedideliais duomenų dydžiais. Galima pastebėti, jog sugeneruotų ir literatūroje pateikiamų variantų rezultatai šiuo atveju skiriasi mažiau. Tai paaiškina `shuffled` duomenų rinkinio specifika - kadangi duomenys yra atsitiktiniai, tenka atlikti žymiai daugiau elementų perstūmimų, tad didesnis naudojamų tarpų kiekis nesukelia problemų. Ir toliau galima pastebėti tendenciją, jog su nedideliais duomenų dydžiais literatūroje pateikiamų variantų veikimo laikas gerėja augant duomenų dydžiui. Sugeneruoti algoritmai su visais pateikiamais duomenų dydžiais lenkia literatūroje pateikiamus variantus. Iš esmės tai galima paaiškinti tuo, jog sugeneruoti algoritmai naudoja mažiau perėjimų, jų santykinis tarpų atstumas yra didesnis, tad su visais duomenų dydžiais daugiau rikiavimo jie atlieka naudodami rikiavimą įterpimu, kuris veikia greičiau nei Šelo algoritmas nedideliems duomenų dydžiams.



5 pav. Veikimo laikas su duomenų rinkiniu shuffled ir $N \leq 128$

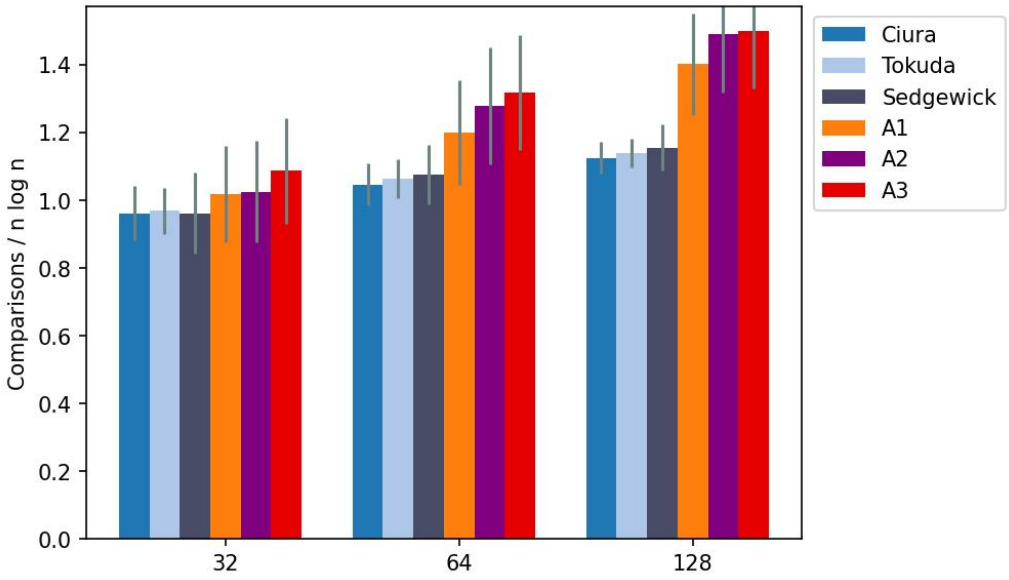
6 pav. yra pateikiami veikimo laiko rezultatai su duomenų rinkiniu `shuffled` ir dideliais duomenų dydžiais. Šiuo atveju galima pastebėti, jog literatūroje pateikiamų variantų veikimo laikas nežymiai didėja augant duomenų dydžiui. Tuo tarpu sugeneruotų algoritmų C2 ir C3 veikimo laikas išlieka pakankamai stabilus, net ir augant duomenų dydžiui, taip pat bene geriausi rezultatai pastebimi su didžiausiu duomenų dydžiu. Tiesa, algoritmo C1 rezultatai yra artimesni literatūroje pateikiamiems variantams, kas yra pakankamai netikėta, kadangi jis yra labai panašus į algoritmą C2 (skiriasi tik du priešpaskutiniai perėjimai). Tad galima daryti išvadą, jog perėjimai su didesniais tarpais rikiuojant pakankamai didelius duomenų dydžius turi gana didelę įtaką duoto varianto efektyvumui.



6 pav. Veikimo laikas su duomenų rinkiniu shuffled ir $N \leq 8192$

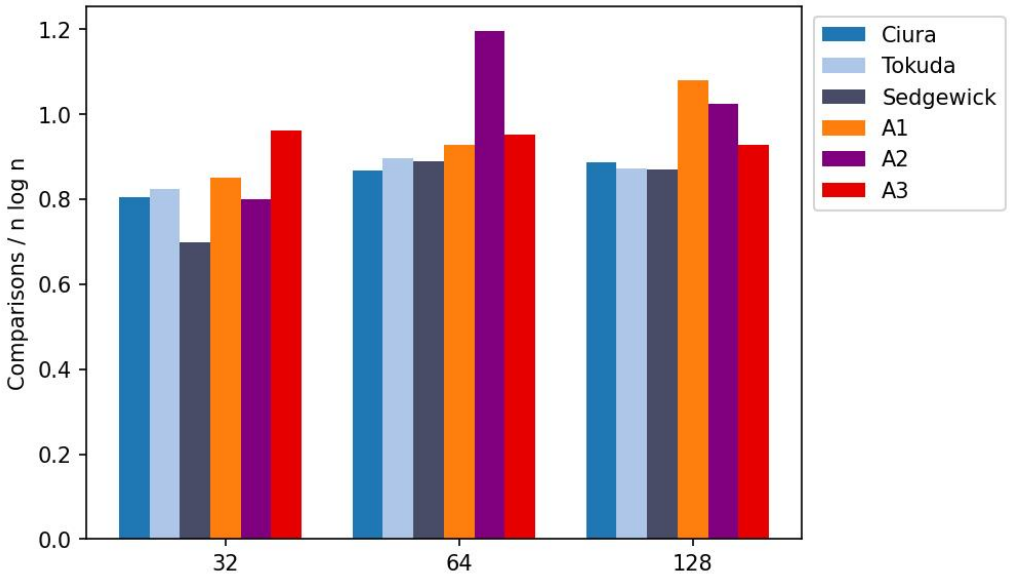
8.2.2. Palyginimai

7 pav. yra pateikiami atliekami palyginimai su duomenų rinkiniu `shuffled` ir nedideliais duomenų dydžiais. Galima pastebėti, jog sugeneruotų variantų atliekamų palyginimų skaičius didėjant duomenų dydžiui gana sparčiai auga. Literatūrinių variantų atliekamų palyginimų skaičius auga nuosaikiau. Su $N = 32$ sugeneruotų variantų atliekamų palyginimų skaičius yra pakankamai mažas ir nedaug atsilieka nuo literatūrinių variantų. Tai galima paaiškinti tuo, šie variantai naudoja mažiau tarpų ir daugiau rikiavimo atlieka su tarpu 1, o rikiavimas įterpimu yra pakankamai efektyvus atliekamų palyginimų prasme, kai duomenų dydis yra pakankamai mažas. Tačiau mažesnis perėjimų kiekis sukelia sunkumų su didesniais duomenų dydžiais ir duomenų dydžiui augant sugeneruoti variantai atlieka vis daugiau palyginimų.



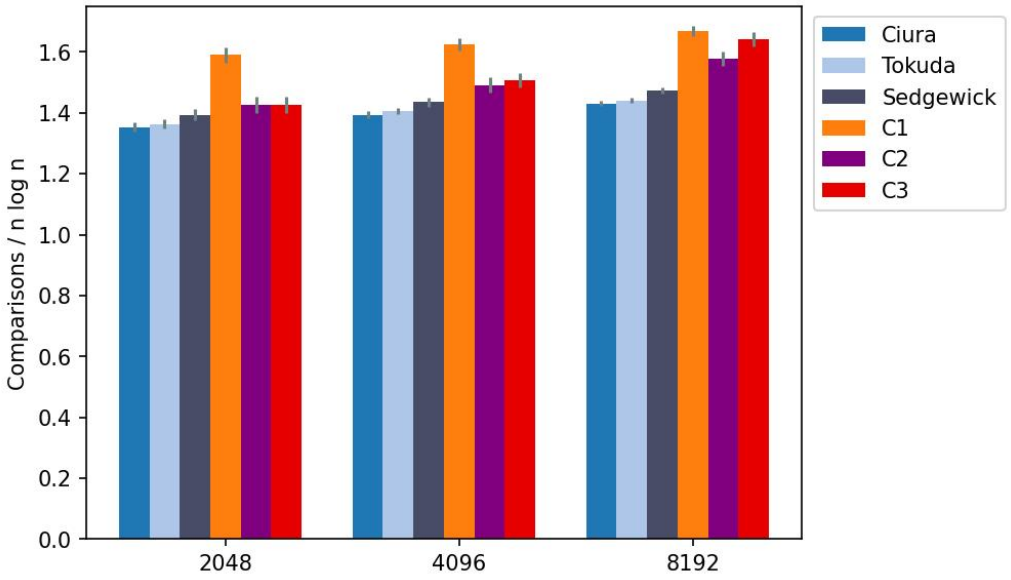
7 pav. Palyginimai su duomenų rinkiniu shuffled ir $N \leq 128$

8 pav. yra pateikiami atliekami palyginimai su duomenų rinkiniu `descending` ir nedideliais duomenų dydžiais. Šiuo atveju literatūrinų variantų rezultatai yra gana stabilūs ir auga nežymiai didėjant duomenų dydžiui. Tuo tarpu sugeneruoti variantai veikia mažiau stabiliai, galima pastebėti nereguliarius rezultatus priklausomai nuo duomenų dydžio: su $N = 32$ pastebimi apytiksliai 30% prastesni A3 varianto rezultatai lyginant su geriausius rezultatus pateikusiu variantu (Sedgewick), su $N = 64$ pastebimas didelis A2 varianto atliekamų palyginimų šuolis, o su $N = 128$ pastebimas gana didelis A1 varianto skirtumas nuo geriausio rezultato. Tai galima paaiškinti tuo, jog sugeneruoti variantai naudoja ganėtinai mažai tarpų, ir pasitaikius nepalankiam duomenų rinkiniui ir/ar dydžiui, šių variantų efektyvumas gali būti stipriau paveiktas, kadangi papildomi tarpai duoda tam tikrą garantiją, jog blogiausias atvejis nebus ypač prastas. Kaip pavyzdį galima pateikti Pratt tarpų seką [Pra72], kuri naudoja labai daug tarpų, tad praktikoje nėra efektyvi, tačiau šiuo metu yra geriausia žinoma seka pagal laiko sudėtingumą blogiausiu atveju.



8 pav. Palyginimai su duomenų rinkiniu descending ir $N \leq 128$

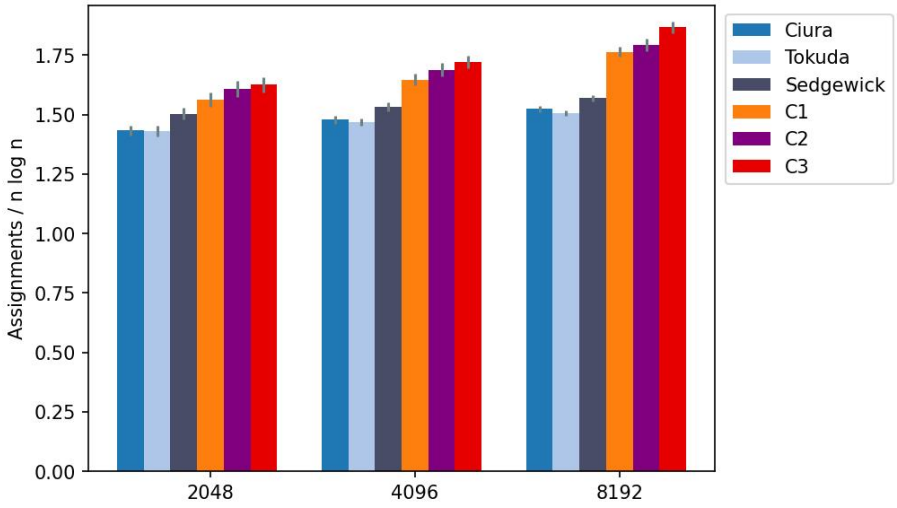
9 pav. yra pateikiami atliekami palyginimai su duomenų rinkiniu `shuffled` ir dideliais duomenų dydžiais. Galima pastebėti, jog sugeneruotų variantų rezultatai su duomenų dydžiais 2048 ir 4096 yra pakankamai geri ir nežymiai atsilieka nuo literatūrinių variantų. Tiesa, su didžiausiu tiriamu duomenų dydžiu atliekamų palyginimų rezultatai yra prastesni, tačiau veikimo laiko rezultatai su šiuo duomenų dydžiu buvo pastebimai geresni. Tad iš esmės pastebimas sugeneruotų variantų daromas kompromisas, kur atliekamų operacijų skaičius paaukojamas vardan palankesnio veikimo laiko.



9 pav. Palyginimai su duomenų rinkiniu shuffled ir $N \leq 8192$

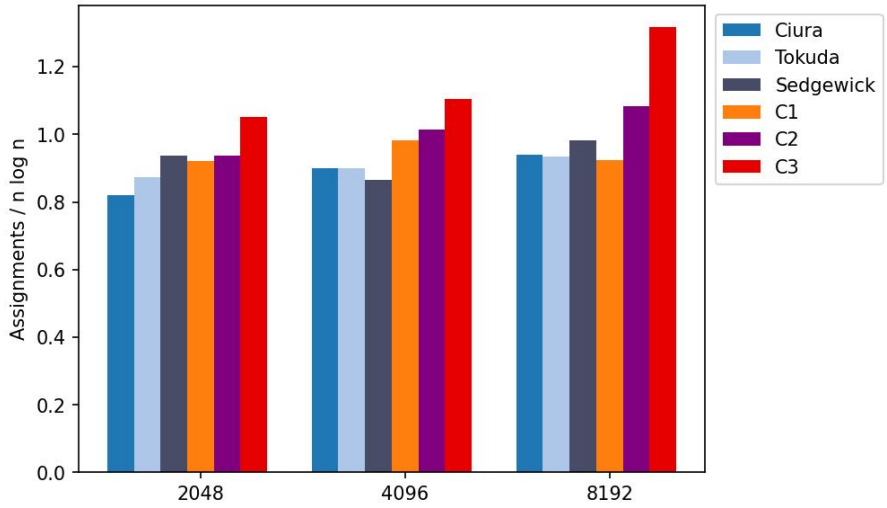
8.2.3. Priskyrimai

Bendrai atliekamų priskyrimų rezultatai yra gana panašūs į atliekamų palyginimų, nors yra pastebima skirtumų. To pavyzdys yra 10 pav. esantys priskyrimų rezultatai su duomenų rinkiniu `shuffled` ir dideliais duomenų dydžiais. Šiuo atveju variantas C1, kuris su tirtais duomenų dydžiais atliko daugiausia palyginimų, iš sugeneruotų variantų atliko mažiausiai priskyrimų. Labiausiai tikėtina, jog generavimo metu žemas priskyrimų skaičius atsvėrė kitus kriterijus ir taip sumažino varianto bendrą kainą.



10 pav. Priskyrimai su duomenų rinkiniu shuffled ir $N \leq 8192$

Taip pat skirtumai pastebimi su `descending` duomenų rinkiniu ir dideliais duomenų dydžiais (11 pav.). Šiuo atveju varianto C3 atliekamų priskyrimų skaičius vis auga ir ties $N = 8192$ nuo geriausio rezultato skiriasi apytiksliai 30%. Kadangi variantas C3 turi 2 perėjimais mažiau nei variantai C1 ir C2 ir didžiausi jo tarpai yra santykinai nedideli, galima pastebėti naudojamų perėjimų skaičiaus ir didesnių tarpų įtaką efektyvumui kai duomenų rinkinys yra nepalankus.



11 pav. Priskyrimai su duomenų rinkiniu shuffled ir $N \leq 8192$

Išvados

Conclusions

Literatūra

[ANP15] Nicolas Auger, Cyril Nicaud ir Carine Pivoteau. Merge strategies: from merge sort to Timsort, 2015.

[Aut09] The Go Authors. sort/sort.go. 2009. URL: https://golang.org/src/sort/sort.go (tikrinta 2021-05-24).

[BG05] Paul Biggar ir David Gregg. Sorting in the presence of branch prediction and caches, 2005.

[BNW+08] Paul Biggar, Nicholas Nash, Kevin Williams ir David Gregg. An experimental study of sorting and branch prediction. *Journal of Experimental Algorithmics (JEA)*, 12:1–39, 2008.

[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.

[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.

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

[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.

[PPS92] C. G. Plaxton, B. Poonen ir T. Suel. Improved lower bounds for Shellsort. *Proceedings., 33rd Annual Symposium on Foundations of Computer Science*, p.p. 226–235, 1992. DOI: 10.1109/SFCS.1992.267769.

[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.

[RBH+02] Robert S Roos, Tiffany Bennett, Jennifer Hannon ir Elizabeth Zehner. A genetic algorithm for improved shellsort sequences. *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, p.p. 694–694, 2002.

[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.

[Sew10] Julian Seward. bzip2/blocksort.c. 2010. URL: https://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/util/compress/bzip2/blocksort.c#L519 (tikrinta 2021-05-24).

[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

Eksperimentams naudota Šelo algoritmo implementacija

```
template <typename T>
inline void test_shellsort(T & data) {
    const int gaps[] = { 1 };
    const int size = data.size();

    for (int gap: gaps) {
        for (int i = gap; i < size; i++) {
            if (data[i - gap] > data[i]) {
                auto temp = data[i];
                int j = i;

                do {
                    data[j] = data[j - gap];
                    j -= gap;
                } while (j >= gap && data[j - gap] > temp);

                data[j] = temp;
            }
        }
    }
}
```