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]. 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 sumažinti uždavinio sprendimo 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).

Šelo rikiavimo algoritmas (angl. Shellsort, toliau - Šelo algoritmas) [She59] yra paremtas palyginimu, nenaudojantis papildomos atminties ir nestabilus. Šelo algoritmą galima laikyti rikiavimo įterpimu modifikacija, 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 efektyvumas tiesiogiai priklauso nuo pasirinktos tarpų sekos [Ciu01; Sed96]. Yra įrodyta, kad Šelo algoritmo laiko sudėtingumo blogiausiu atveju apatinė riba yra $\Omega(\frac{n\, log^2\, n}{log\, log\, n^2})$ [PPS92], taigi jis nėra asimptotiškai optimalus. Tiesa, kol kas nėra rasta seka, su kuria Šelo algoritmas pasiektų šią apatinę ribą. Kiek žinoma autoriui, asimptotiškai geriausia tarpų seka yra rasta Pratt, kuri yra formos 2^p3^p ir turi $\Theta(n\, log^2\, n)$ laiko sudėtingumą blogiausiu atveju [Pra72], tačiau praktikoje ji veikia lėčiau už Ciura [Ciu01] ar Tokuda [Tok92] eksperimentiškai rastas sekas.

Darbo **tikslas**: pritaikyti genetinius algoritmus Šelo algoritmo variantų generavimui.

Darbo uždaviniai:

- Atlikti Šelo algoritmo ir jo variantų literatūros analizę.

- Atlikti genetinių algoritmų 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 atliekama genetinių algoritmų literatūros analizė. 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
```

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 rikiavimo 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 ← 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 plaktuvo rikiavimu (angl. cocktail shaker sort). Šio algoritmo taikomą perėjimą, kurį toliau vadinsime purtymo perėjimu (angl. shake pass), integravus į Šelo algoritmo karkasą taip pat gaunamas gana įdomus algoritmas [IS86]. Purtymo perėjimo pseudokodas pateikiamas 5 algoritme.

Algorithm 5 Purtymo 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). 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 plytos perėjimu (angl. brick pass) [Sed96], nesunkiai galima pritaikyti ir Šelo algoritmo karkasui, kintamuoju pakeitus originaliame algoritme taikytą tarpą 1 [Lem94]. Plytos perėjimo pseudokodas yra pateikiamas 6 algoritme.

Algorithm 6 Plytos 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. Genetiniai algoritmai

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 dominuojanti operacija yra palyginimas. Tiesa, Šelo algoritmo variantai nėra taip išsamiai ištirti [Bre01]. Todėl šio darbo kontekste sudėtinga pagrįstai nustatyti, kuri iš operacijų turėtų turėti didesnį svorį.

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. Taip pat reikia pastebėti, kad Šelo algoritmo atliekamų operacijų skaičius ir veikimo laikas nėra tiesiogiai susiję, t.y. vien tai, jog duotas Šelo algoritmo variantas atlieka sąlyginai mažai operacijų, dar nereiškia jog praktikoje jis veiks greičiau už tuos, kurie operacijų atlieka daugiau. Atsižvelgiant į tai, 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.

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 "saugesnės" programavimo kalbos) ir tam tikrų kalbos aspektų, tokių kaip operatorių perkrovimas. OpenGA biblioteka buvo pasirinkta dėl modernių C++ kalbos konstruktų 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 7 ir 8 algoritmuose.

5. Šelo algoritmo variantų generavimas

Šis skyrius sudarytas iš 3 poskyrių. Antrame poskyryje generuojami Šelo algoritmo variantai, kai rikiuojamų duomenų dydis yra 1000. Trečiame poskyryje generuojami Šelo algoritmo variantai, kai rikiuojamų duomenų dydis yra 100000.

5.1. Bendra info

Prieš pradedant generuoti Šelo algoritmo variantus, reikėtų aptarti kaip turėtų atrodyti sprendinys, atspindintis tam tikrą Šelo algoritmo variantą. Kadangi genetinis algoritmas sprendinių evoliuciją vykdo taikant mutacijos ir rekombinacijos operatorius, sieksime, jog šių operatorių taikymas sprendiniams būtų kuo paprastesnis. Tuo tikslu sprendinio duomenų modelis turėtų būti kuo paprastesnis, o jo elementai sudaryti iš primityvių duomenų tipų. 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. Tiesa, toks algoritmas 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.

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, siekiant jog baudos dydis neviršytų $max(c.comparisons, c.assignments)$ su pasirinktais duomenų dydžiais. Algoritmo atliekamiems priskyrimams ir palyginimams tinkamumo funkcijoje buvo taikomi svoriai, paremti praeitame skyriuje apibrėžtais efektyvaus Šelo algoritmo varianto kriterijais.

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 sprendinio tinkamumas, rikiuojamų duomenų dydis, inversijų skaičius po rikiavimo, atliktų palyginimų skaičius, atliktų priskyrimų skaičius ir veikimo laikas.

5.2. Šelo algoritmo variantų generavimas, kai N = 1000

5.3. Šelo algoritmo variantų generavimas, kai N = 100000

6. Šelo algoritmo variantų efektyvumo tyrimo aplinkos paruošimas

7. Šelo algoritmo variantų efektyvumo tyrimas

Šis skyrius sudarytas iš 2 poskyrių. Pirmame poskyryje tiriamas Šelo algoritmo variantų efektyvumas, kai maksimalus rikiuojamų duomenų dydis yra 1000. Antrame poskyryje tiriamas Šelo algoritmo variantų efektyvumas, kai maksimalus rikiuojamų duomenų dydis yra 100000.

7.1. Šelo algoritmo variantų efektyvumo tyrimas, kai N <= 1000

7.2. Šelo algoritmo variantų efektyvumo tyrimas, kai N <= 100000

Išvados

Išvadose visą darbą sukišam į porą puslapių, tad čia gana svarbi dalis.

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.

[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.

[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.

[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.

[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.

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
```