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 2
1. Šelo algoritmas ir jo variantai 4
    1.1. Šelo algoritmas 4
    1.2. Šelo algoritmo variantai 4
2. Genetiniai algoritmai 7
    2.1. Chromosomų populiacija 7
    2.2. Genetiniai operatoriai 7
3. Šelo algoritmo variantų efektyvumo kriterijai 9
4. Šelo algoritmo variantų generavimo aplinkos paruošimas 10
5. Šelo algoritmo variantų generavimas 11
    5.1. Genetinis algoritmas 11
    5.2. Šelo algoritmo variantų generavimas, kai N = 1000 12
    5.3. Šelo algoritmo variantų generavimas, kai N = 100000 13
6. Šelo algoritmo variantų efektyvumo tyrimo aplinkos paruošimas 14
7. Šelo algoritmo variantų efektyvumo tyrimas 15
    7.1. Šelo algoritmo variantų efektyvumo tyrimas rikiuojant nedidelius duomenų dydžius 15
    7.2. Šelo algoritmo variantų efektyvumo tyrimas rikiuojant vidutinius duomenų dydžius 17
Išvados 19
Literatūra 20
Priedas Nr.1
Priedas Nr.2

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

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

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 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 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 rikiuojamų duomenų dydis yra 1000. Trečiame poskyryje generuojami Šelo algoritmo variantai, kai rikiuojamų duomenų dydis yra 100000.

5.1. Genetinis algoritmas

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. Tai tuo pačiu palengvina ir sprendinių serializaciją, kas leidžia sprendinius serializuoti pvz. JSON formatu ir išsaugoti į failą ar duomenų bazę. 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. 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.

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.

Esminė genetinio algoritmo dalis yra genetiniai operatoriai. 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.

Genetinio algoritmo rezultatų kokybė taip pat labai priklauso nuo pasirinktų vykdymo parametrų. 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, 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 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, kurie JSON formatu pateikiami prieduose.

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: $p_i = (rand\_type(), gap(p_{i-1}) * q)$, 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, kurie JSON formatu pateikiami prieduose.

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 rikiuojant nedidelius duomenų dydžius

1 lentelė. Efektyvumo tyrimo rezultatai, kai N = 125

algoritmas	vid. palyginimai	vid. priskyrimai	vid. laikas (µs)
S_C	**978**	1568	4.04
S_T	991	1615	4.90
S_S	1005	1508	3.90
SI_C	**978**	1068	3.94
SI_T	991	**1042**	3.75
SI_S	1005	1125	**3.07**
GS_1	**978**	1068	**3.63**
GS_2	**978**	1068	3.68
GS_3	984	**1053**	3.71
GS_4	992	**1049**	4.01
GS_5	**978**	1068	**3.39**

2 lentelė. Efektyvumo tyrimo rezultatai, kai N = 250

algoritmas	vid. palyginimai	vid. priskyrimai	vid. laikas (µs)
S_C	**2377**	3776	10.71
S_T	2401	3853	10.00
S_S	2453	3685	9.63
SI_C	**2377**	2571	**9.08**
SI_T	2401	**2533**	9.78
SI_S	2453	2696	**8.40**
GS_1	**2377**	2571	9.23
GS_2	**2377**	2571	9.44
GS_3	2390	**2558**	**9.09**
GS_4	2400	**2542**	9.27
GS_5	**2367**	2578	9.15

3 lentelė. Efektyvumo tyrimo rezultatai, kai N = 500

algoritmas	vid. palyginimai	vid. priskyrimai	vid. laikas (µs)
S_C	**5621**	8835	24.83
S_T	5666	9001	24.52
S_S	5783	8642	24.56
SI_C	**5621**	**5994**	22.96
SI_T	5666	5999	23.03
SI_S	5783	6360	**21.30**
GS_1	**5612**	**5991**	24.24
GS_2	**5621**	**5994**	**21.71**
GS_3	5655	6027	23.16
GS_4	5669	6016	23.26
GS_5	**5607**	6089	**22.66**

4 lentelė. Efektyvumo tyrimo rezultatai, kai N = 1000

algoritmas	vid. palyginimai	vid. priskyrimai	vid. laikas (µs)
S_C	13041	20270	55.57
S_T	13129	20708	56.69
S_S	13395	20089	52.55
SI_C	13041	**13816**	55.05
SI_T	13129	**13863**	56.55
SI_S	13395	14525	**50.50**
GS_1	**12961**	13905	**52.21**
GS_2	**12990**	14048	52.34
GS_3	13070	**13817**	53.18
GS_4	13121	13889	54.02
GS_5	**13023**	14366	**51.56**

7.2. Šelo algoritmo variantų efektyvumo tyrimas rikiuojant vidutinius duomenų dydžius

5 lentelė. Efektyvumo tyrimo rezultatai, kai N = 12500

algoritmas	vid. palyginimai	vid. priskyrimai	vid. laikas (µs)
S_C	**247436**	376596	1009
S_T	248135	381261	1025
S_S	254437	381187	984
SI_C	**247436**	262114	**936**
SI_T	248135	**261766**	973
SI_S	254437	268334	**929**
GL_1	**247107**	**261734**	949
GL_2	**247317**	**261825**	**948**
GL_3	248879	263458	972
GL_4	248598	263534	951
GL_5	247540	262044	956

6 lentelė. Efektyvumo tyrimo rezultatai, kai N = 25000

algoritmas	vid. palyginimai	vid. priskyrimai	vid. laikas (µs)
S_C	**542838**	822347	2323
S_T	544133	832851	2379
S_S	558162	838206	2267
SI_C	**542838**	574642	**2162**
SI_T	544133	**573608**	2219
SI_S	558162	586459	**2140**
GL_1	**541866**	**573477**	**2157**
GL_2	**541303**	574561	2183
GL_3	546239	579532	2208
GL_4	545417	576050	2225
GL_5	543056	**573435**	2214

7 lentelė. Efektyvumo tyrimo rezultatai, kai N = 50000

algoritmas	vid. palyginimai	vid. priskyrimai	vid. laikas (µs)
S_C	**1180477**	1781235	5203
S_T	1184476	1804441	5335
S_S	1211998	1817385	5140
SI_C	**1180477**	1250232	4816
SI_T	1184476	**1244335**	4866
SI_S	1211998	1269569	**4723**
GL_1	**1178027**	**1247428**	**4816**
GL_2	**1180429**	1249592	**4795**
GL_3	1189048	1261191	4822
GL_4	1184607	1252540	4819
GL_5	1180783	**1246917**	4832

8 lentelė. Efektyvumo tyrimo rezultatai, kai N = 100000

algoritmas	vid. palyginimai	vid. priskyrimai	vid. laikas (µs)
S_C	**2552009**	3835896	11305
S_T	2564127	3888498	11853
S_S	2624030	3941772	11980
SI_C	**2552009**	2710178	10765
SI_T	2564127	**2691417**	10671
SI_S	2624030	2741613	**10424**
GL_1	**2545884**	**2703088**	**10475**
GL_2	**2549224**	2735020	10568
GL_3	2569991	2768514	10481
GL_4	2556929	2718989	**10434**
GL_5	2552700	**2697404**	10542

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.

[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.

[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.

[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.

[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

procedure MERGESORT(a)
    $n \leftarrow size(a)$
    $inv \leftarrow 0$
    if $n \leq 1$ **then**
        return 0
    end if
    let l and r be the result of splitting a at $n/2$
    $inv \leftarrow inv + mergesort(l)$
    $inv \leftarrow inv + mergesort(r)$
    $inv \leftarrow inv + merge(l,r,a)$
    return inv
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
Šelo algoritmo variantų generavimo rezultatai