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:
doc. dr. Vardauskas Pavardauskas (parašas)

Vilnius
2022

Turinys

Sąvokų apibrėžimai

Šelo algoritmas GA

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 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].

Literatūroje galima rasti darbų [RBH+02; SY99], kuriuose genetiniai algoritmai yra taikomi

naujų Šelo algoritmo tarpų sekų radimui. Simpson-Yachavaram darbe daugiausia dėmesio skiriama atliekamiems palyginimams, teigiama, jog gautos tarpų sekos atlieka mažiausiai palyginimo operacijų, tačiau jos yra lyginamos tik su Sedgewick ir Incerpi-Sedgewick tarpų sekomis. Tačiau šiuo metu yra laikoma, jog vidutiniškai atliekamų palyginimų atžvilgiu optimaliausia yra Ciura [Ciu01] tarpų seka. Kursinio darbo metu buvo atliktas Ciura ir Simpson-Yachavaram tarpų sekų tarpusavio palyginimas vidutiniškai atliekamų priskyrimų atžvilgiu, gauti rezultatai tam neprieštaravo. Roos et al. darbe daugiausia dėmesio skiriama veikimo laikui, teigiama, jog gauta tarpų seka veikia greičiau nei kitos tirtos tarpų sekos, tačiau pateikiamuose rezultatuose ji lyginama tik su Simpson-Yachavaram tarpų seka, tiriami tik keli duomenų dydžiai. Kursinio darbo metu buvo atliktas platesnio masto Roos et al. ir kitų tarpų sekų tarpusavio palyginimas vidutinio veikimo laiko atžvilgiu, pagal gautus rezultatus Roos et al. sekos veikimo laikas buvo mažiausias, tačiau ji taip pat atliko žymiai daugiau palyginimo ir priskyrimo operacijų nei kitos tirtos tarpų sekos. Nepaisant aukščiau aprašytų darbų trūkumų, juose gauti rezultatai rodo, jog genetinių algoritmų taikymas Šelo algoritmo variantų konstravimui yra prasmingas ir gali duoti reikšmingų rezultatų.

Viena iš pagrindinių problemų konstruojant rikiavimo algoritmą yra pusiausvyros tarp atliekamų operacijų ir veikimo laiko išlaikymas. Algoritmas, kuris atlieka labai mažai operacijų, tačiau veikia lėtai, yra įdomus tik teorine prasme ir sunkiai panaudojamas praktikoje. Tas pats galioja ir algoritmams, kurie prioritetizuoja tik veikimo laiką, tačiau atlieka labai daug operacijų, kadangi tai apsunkina sudėtingesnių duomenų tipų rikiavimą ir apriboja duoto algoritmo panaudojamumą. Todėl prasmingiausia būtų ieškoti naujų Šelo algoritmo variantų pasitelkiant tokį metodą, kuris išlaikytų pusiausvyrą tarp veikimo laiko ir atliekamų operacijų. Kursiniame projekte buvo atliktas tyrimas, kuriame pasitelkiant vienkriterinį genetinį algoritmą buvo konstruojami Šelo algoritmo variantai. Siekiant išlaikyti balansą tarp skirtingų kriterijų, buvo pasitelktas svorinės sumos metodas, tačiau tyrimo metu buvo pastebėtas jo ribotas efektyvumas ir svorių teikimo skirtingiems kriterijams nepagrįstumas. Atsižvelgiant į kursiniame projekte atlikto tyrimo trūkumus, šiame darbe pasirinkta naudoti daugiakriterinį genetinį algoritmą, kuris padėtų sukonstruoti algoritmą išlaikantį 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 ir eksperimentiškai gautais duomenimis;
- 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:

- kol kas nieko

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

Šis skyrius sudarytas iš 3 poskyrių. Pirmame poskyryje remiantis literatūra atliekama Šelo algoritmo analizė. Antrame poskyryje remiantis literatūra atliekama Šelo algoritmo variantų analizė. Trečiame poskyryje atliekama eksperimentinė Šelo algoritmo analizė.

1.1. Šelo algoritmo literatūros analizė

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
```

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
```

1.2. Šelo algoritmo variantų literatūros analizė

Šelo algoritmas pasižymi variantų gausa. Šelo algoritmą iš esmės galima laikyti tarpų iteravimu, su kiekvienu iš jų 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 nagrinėsime tuos variantus, 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 būdingą struktūrą toliau vadinsime Šelo algoritmo karkasu, o karkaso viduje atliekamą rikiavimo logiką - perėjimu (angl. pass). Šelo algoritmo karkaso apibrėžimas pateikiamas pseudokodu 3 algoritme. Standartinį Šelo algoritmo taikomą perėjimą toliau vadinsime įterpimo perėjimu.

Algorithm 3 Šelo algoritmo karkasas

```
1: foreach h in H do
2:     perform pass with gap h
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 šis perėjimas dažniausiai 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 rikiavimu (angl. cocktail sort arba shaker sort). Šio algoritmo taikomą perėjimą, kurį toliau vadinsime supurtymo perėjimu (angl. shake pass), integravus į Šelo algoritmo karkasą taip pat gaunamas gana įdomus algoritmas [IS86]. Supurtymo perėjimo pseudokodas pateikiamas 5 algoritme.

Algorithm 5 Supurtymo perėjimas

```
1: for i ← 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 karkasui, kintamuoju pakeitus originaliame algoritme taikytą tarpą 1 [Lem94]. Šį perėjimą toliau vadinsime mūrijimo perėjimu (angl. brick pass). Jo pseudokodas yra pateikiamas 6 algoritme.

Algorithm 6 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**

1.3. Eksperimentinė Šelo algoritmo analizė

2. 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.

Pastebėsime, jog 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. 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 išlygiagretinti instrukcijų vykdymą). Dėl šiuolaikinių kompiuterių veikimo subtilybių algoritmai, kurie atlieka daugiau palyginimo ir priskyrimo operacijų, kai kuriais atvejais gali veikti greičiau, nei operacijų atžvilgiu optimalesni algoritmai [Pet21]. Rezultatai, gauti kursiniame darbe ir kursiniame projekte taip pat tam neprieštarauja - tyrimuose geriausius veikimo laiko rezultatus parodę Šelo algoritmo variantai atliko ženkliai daugiau operacijų, nei atliekamų operacijų operacijų atžvilgiu optimalūs variantai. 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.

3. 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.

3.1. 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.

3.2. 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.

4. 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. Sudėtingiau yra greitai sugeneruoti pakankamai efektyvias tarpų sekas. Tuo tikslu buvo pasirinkta individų inializacijai naudoti geometrines tarpų sekas. Kaip galima pastebėti 7 algoritme, jas gana nesudėtinga generuoti. Taip pat geometrinės tarpų sekos yra pakankamai efektyvios bendru atveju. 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.

Algorithm 7 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, lygiagretaus vykdymo palaikymo ir patirties sukauptos ruošiant kursinį darbą ir kursinį projektą.

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 pasirinkta 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:

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

Individų mutacija buvo įgyvendinta pasirenkant atsitiktinį individo geną ir atsitiktinai pakeičiant jo perėjimo tipą kuriuo nors kitu. Sprendinių 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 - 500, mutacijos tikimybė - 0.1, rekombinuojama populiacijos dalis - 0.4. GA buvo nurodyta sustoti įvykdžius 100 iteracijų.

5. Šelo algoritmo variantų generavimas

Išvados

Išvadose ir pasiūlymuose, nekartojant atskirų dalių apibendrinimų, suformuluojamos svarbiausios darbo išvados, rekomendacijos bei pasiūlymai.

Conclusions

Šiame skyriuje pateikiamos išvados (reziume) anglų kalba.

Literatūra

[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.

[Pet21] Orson RL Peters. Pattern-defeating quicksort. *arXiv preprint arXiv:2106.05123*, 2021.

[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.

[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.

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

[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.