VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS INSTITUTAS
INFORMATIKOS KATEDRA

Kursinis darbas

Rikiavimo tobulinimas genetiniais algoritmais

(Improving sorting with genetic algorithms)

Atliko: 3 kurso 2 grupės studentas
Deividas Zaleskis (parašas)

Darbo vadovas:
Irmantas Radavičius (parašas)

Vilnius
2021

Turinys

Įvadas

Viena pagrindinių informatikos sąvokų yra algoritmas. Formaliai algoritmą galima apibūdinti kaip baigtinę seką instrukcijų, nurodančių kaip rasti nagrinėjamo uždavinio sprendinį. Algoritmo koncepcija egzistuoja nuo senovės laikų [Knu72], tačiau atsiradus kompiuteriams, tapo įmanoma algoritmų vykdymą automatizuoti, paverčiant juos mašininiu kodu suprantamu kompiuteriams [WWG51]. Taip informatikos mokslas nuo teorinių šaknų [Tur37] įgavo ir taikomąją pusę. Beveik visus algoritmus galima suskirstyti į dvi klases: kombinatorinius algoritmus ir skaitinius algoritmus. Skaitiniai algoritmai sprendžia tolydžius uždavinius: optimizuoti realaus argumento funkciją, išspręsti tiesinių lygčių sistemą su realiais koeficientais, etc. Kombinatoriniai algoritmai sprendžia diskrečius uždavinius ir operuoja diskrečiais objektais: skaičiais, sąrašais, grafais, etc. Vienas žinomiausių diskretaus uždavinio pavyzdžių yra duomenų rikiavimas.

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 pradinės sekos elementai būtų išdėstyti didėjančia (mažėjančia) tvarka [RB13]. Rikiavimo uždavinys yra aktualus nuo pat kompiuterių atsiradimo ir buvo laikomas vienu pagrindinių diskrečių uždavinių, kuriuos turėtų gebėti spręsti kompiuteris [Knu70]. Rikiavimo uždavinio sprendimas dažnai padeda pagrindą efektyviam kito uždavinio sprendimui, pavyzdžiui, atliekant paiešką sąraše, galima taikyti dvejetainės paieškos algoritmą tik tada, kai sąrašas yra išrikiuotas. Kadangi rikiavimo uždavinys yra fundamentalus, jam spręsti egzistuoja labai skirtingų algoritmų.

Rikiavimo algoritmų yra įvairių: paremtų palyginimu (elementų tvarką nustato naudojant palyginimo operatorius), stabilių (nekeičia lygių elementų tvarkos), nenaudojančių papildomos atminties (atminties sudėtingumas yra $O(1)$), etc. Asimptotiškai optimalūs palyginimu paremti algoritmai blogiausiu atveju turi $O(n \, log \, n)$ laiko sudėtingumą, o ne palyginimu paremti algoritmai gali veikti dar greičiau, tačiau nėra tokie universalūs, kadangi rikiuojama remiantis duomenų specifika. Tiesa, rikiuojant remtis vien algoritmo asimptotika nepakanka: rikiavimas įterpimu (angl. insertion sort) blogiausiu atveju turi $O(n^2)$ laiko sudėtingumą [BFM06], tačiau mažesnius elementų kiekius rikiuoja daug greičiau, nei asimptotiškai optimalūs algoritmai, pavyzdžiui, rikiavimas krūva (angl. heapsort) [For64]. Todėl pastaruoju metu plačiai naudojami hibridiniai rikiavimo algoritmai, kurie sujungia keletą rikiavimo algoritmų į vieną ir panaudoja jų geriausias savybes. Nepaisant įvairovės ir naujų algoritmų gausos, klasikiniai rikiavimo algoritmai išlieka aktualūs.

Šelo rikiavimo algoritmas (angl. Shellsort, toliau - Šelo algoritmas) [She59] yra paremtas palyginimu, nenaudojantis papildomos atminties ir nestabilus. Šelo algoritmą galima laikyti rikiavimo įterpimu 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.

Pritaikant šias idėjas ir rikiavimui naudojant mažėjančią tarpų seką su paskutiniu nariu 1, kuris garantuoja rikiavimą įterpimu paskutinėje iteracijoje, galima užtikrinti, jog algoritmo darbo pabaigoje seka S bus pilnai išrikiuota. Įvertinant Šelo algoritmo idėjas, nesunku pastebėti tarpų sekų įtaką jo veikimui.

Šelo algoritmo efektyvumas tiesiogiai priklauso nuo pasirinktos tarpų sekos. Weiss atlikto tyrimo [Wei91] rezultatai rodo, jog su Sedgewick pasiūlyta seka šis algoritmas veikia beveik dvigubai greičiau nei Šelo pradinis variantas, kai $n = 1000000$. 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 [Pra72], tačiau praktikoje ji veikia lėčiau už Ciura [Ciu01] ar Tokuda [Tok92] pasiūlytas sekas. Daugelio praktikoje efektyvių sekų asimptotinis sudėtingumas laiko atžvilgiu lieka atvira problema, nes jos yra rastos eksperimentiškai. Vienas iš metodų, kuriuos galima taikyti efektyvių tarpų sekų radimui, yra genetinis algoritmas.

Genetinis algoritmas (GA) yra metodas rasti euristikas, paremtas biologijos žiniomis apie natūralios atrankos procesą. Kartu su genetiniu programavimu, evoliuciniais algoritmais ir kitais metodais, genetiniai algoritmai sudaro evoliucinių skaičiavimų šeimą. Visi šios šeimos atstovai yra paremti pradinės populiacijos generavimu ir iteraciniu populiacijos atnaujinimu naudojant biologijos įkvėptas strategijas. J.H. Holland, GA pradininkas, savo knygoje [Hol92] apibrėžė genetinio algoritmo sąvoką ir su ja glaudžiai susijusias chromosomų (potencialių uždavinio sprendinių, išreikštų genų rinkiniu), bei rekombinacijos (tėvinių chromosomų genų perdavimo palikuonims), atrankos (tinkamiausių chromosomų atrinkimo) ir mutacijos (savaiminio chromosomos genų kitimo) operatorių koncepcijas. Genetinių algoritmų veikimo strategija pagrįsta pradinės chromosomų populiacijos evoliucija, kiekvienos naujos chromosomų kartos gavimui naudojant rekombinacijos, atrankos ir mutacijos operatorius. Toliau bus aptariamos genetinių algoritmų taikymo galimybės.

Genetiniai algoritmai taikomi sprendžiant įvairius paieškos ir optimizavimo uždavinius, kuomet nesunku nustatyti, ar sprendinys tinkamas, tačiau tinkamo sprendinio radimas reikalauja daug resursų ar net pilno perrinkimo. Tokiu atveju apytikslio sprendinio radimas (euristika) gali būti daug patrauklesnis sprendimo būdas, kadangi tikslaus sprendinio radimas dažnai yra NP-sunkus uždavinys. Todėl GA yra pritaikomi sudarant grafikus ir tvarkaraščius, sprendžiant globalaus optimizavimo uždavinius ir net projektuojant NASA mikrosatelitų antenas [HGL+06]. Nesunku pastebėti, jog efektyvių Šelo algoritmo tarpų sekų radimas yra sunkus uždavinys atliekamų skaičiavimų prasme, tikėtinai reikalaujantis pilno potencialių sprendinių perrinkimo, tad šio uždavinio sprendimui taikyti GA yra prasminga. Kiek žinoma autoriui, kol kas yra buvę du bandymai taikyti genetinius algoritmus efektyvių Šelo algoritmo tarpų sekų radimui [RBH+02; SY99]. Abiejuose darbuose teigiama, jog genetiniais algoritmais gautos tarpų sekos veikia greičiau už Sedgewick seką, kuri literatūroje laikoma viena efektyviausių.

Darbo **tikslas**: pritaikyti genetinius algoritmus Šelo algoritmo tarpų sekoms generuoti.
Darbo uždaviniai:

- Atlikti Šelo rikiavimo algoritmo literatūros analizę.
- Nustatyti kriterijus tarpų sekų efektyvumui įvertinti.
- Paruošti aplinką eksperimentų vykdymui.
- Naudojant genetinius algoritmus sugeneruoti tarpų sekas.
- Atliekant eksperimentus įvertinti sugeneruotų ir pateiktų literatūroje tarpų sekų efektyvumą.

Šis darbas sudarytas iš 6 skyrių. Pirmame skyriuje atliekama Šelo rikiavimo algoritmo literatūros analizė. Antrame skyriuje apžvelgiama genetinio algoritmo sąvoka. Trečiame skyriuje nustatomi kriterijai tarpų sekų efektyvumui įvertinti. Ketvirtame skyriuje paruošiama eksperimentų vykdymo aplinka. Penktame skyriuje generuojamos tarpų sekos, naudojant genetinius algoritmus. Šeštame skyriuje atliekant eksperimentus įvertinamas sugeneruotų ir pateiktų literatūroje tarpų sekų efektyvumas.

1. Šelo rikiavimo algoritmas

1.1. Šelo rikiavimo algoritmo efektyvumas

1.1.1. Šelo rikiavimo algoritmo asimptotinė analizė

1.1.2. Šelo rikiavimo algoritmo efektyvumas priklausomai nuo duomenų dydžio

Renkantis kokį algoritmą naudoti, labai svarbu įvertinti tikėtiną duomenų dydį. Šelo rikiavimo algoritmas yra efektyviausias, kai duomenų dydis yra ganėtinai mažas [Ciu01]. Kaip teigia [SY99], šis algoritmas veikia geriausiai, kai N < 150. Tokiu atveju, jis lenkia net ir vienu greičiausių laikomą greitojo rikiavimo algoritmą. Todėl Šelo rikiavimo algoritmas dažnai naudojamas hibridiniuose rikiavimo algoritmuose, kai pasiekus tam tikrą rekursijos lygį rikiuojamų duomenų dalis tampa pakankamai maža ir tolimesnė rekursija asimptotiškai optimaliu algoritmu nebeturi prasmės, kadangi jos tęsimas reikalautų per daug mašinos resursų. Tokių šio algoritmo naudojimo pavyzdžių galima rasti Go programavimo kalbos standartinėje bibliotekoje [Aut09] bei bzip2 failų glaudinimo programoje [Sew10]. Tiesa, net ir rikiuojant didesnius duomenų dydžius, Šelo algoritmas nėra labai lėtas [Ciu01]. Todėl jis yra vertingas įrankis programuojant įterptinėms sistemoms, kadangi dėl ribotų atminties išteklių asimptotiškai optimalūs algoritmai, kurie dažniausia yra rekursyvūs ir naudoja daugiau atminties, tokioms sistemoms netinka. Taigi, nors Šelo rikiavimo algoritmas nėra vienas greičiausių kai uždavinys didelis, yra scenarijų, kuriais šį algoritmą rinktis tikrai verta.

1.1.3. Šelo rikiavimo algoritmo efektyvumas priklausomai nuo duomenų specifikos

Viena palankiausių Šelo algoritmo savybių yra jo adaptyvumas. Adaptyvumas rikiavimo algoritmų kontekste reiškia, jog algoritmas atlieka mažiau operacijų, jei duomenys dalinai išrikiuoti. Moksliniuose tyrimuose ši savybė retai turi įtakos, kadangi dažniausia siekiama ištirti algoritmo veikimą, kai duomenys nėra tvarkingai išdėstyti ir tipiškai renkamasi atsitiktinai generuoti pradinius duomenis šiam tikslui pasiekti. Savaime aišku, jog praktikoje retai pavyks sutikti visiškai atsitiktinai išsidėsčiusius duomenis, neturinčius nei vieno išrikiuoto posekio didesnio už 1. Kaip pavyzdį galima pateikti vieną populiariausių šiuo metu naudojamų algoritmų, Timsort **??**. Esminė šio algoritmo idėja yra ta, jog realiame pasaulyje sutinkami duomenys labai dažnai turi išrikiuotų posekių, o juos aptikus rikiavimą galima atlikti greičiau, kadangi šių posekių rikiuoti nebereikia. Deja, literatūroje sudėtinga rasti Šelo algoritmo adaptyvumo tyrimų, tad sunku tiksliai įvertinti, kokią didelę įtaką duomenų tvarka daro šio algoritmo efektyvumui. Tačiau remiantis faktu, jog viena pagrindinių Šelo algoritmo idėjų yra aprikiuoti duomenis ir tik tada taikyti rikiavimą įterpimu, kuris veikia labai greitai (šiuo atveju sudėtingumas laiko atžvilgiu yra $O(n)$), kai duomenys išrikiuoti, galima daryti prielaidą, jog su dalinai išrikiuotais pradiniai duomenimis šis algoritmas veiktų ženkliai greičiau.

1.2. Šelo rikiavimo algoritmo versijos

1.2.1. Vadovėlinis Šelo rikiavimo algoritmas

Kaip ir daugelis algoritmų, Šelo rikiavimo algoritmas turi keletą galimų implementacijų. Žinomiausia iš jų, be abejo, yra vadovėlinė šio algoritmo versija, daugeliui pasaulio informatikos studentų žinoma iš duomenų struktūrų ir algoritmų kurso. Jos pseudokodas pateikiamas žemiau.

Algorithm 1 Vadovėlinis Šelo rikiavimo algoritmas

```
1:  foreach gap in H do
2:      for i ← gap + 1 to N 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
```

1.2.2. Patobulintas Šelo rikiavimo algoritmas

Verta pastebėti, jog vadovėlinė Šelo algoritmo implementacija nėra naši [RB13]. Pradedant vidinį vadovėlinio Šelo algoritmo ciklą 5 eilutėje yra tikrinama, ar $S[j]$ jau yra tinkamoje pozicijoje. Tai švaisto resursus, kadangi prieš tai 3 ir 4 eilutėse yra atliekami du priskyrimai, kurie atliekami veltui, jei $S[j]$ jau yra tinkamoje pozicijoje (o taip nutinka dažnai). Siekiant tai ištaisyti, [RB13] pateikė patobulintą Šelo algoritmo versiją, kuri prieš vykdydama bet kokias kitas instrukcijas patikrina, ar elementas $S[j]$ jau yra tinkamoje pozicijoje. Patobulinto Šelo algoritmo pseudokodas pateikiamas žemiau.

Algorithm 2 Patobulintas Šelo rikiavimo algoritmas

```
1: foreach gap in H do
2:     for i ← gap + 1 to N 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
```

2. Genetinis algoritmas

Prieš pradedant generuoti sekas, reikalinga plačiau apžvelgti genetinio algoritmo sąvoką. Paprasčiausias genetinis algoritmas susideda iš chromosomų populiacijos bei atrankos, mutacijos ir rekombinacijos operatorių. Projektuojant genetinį algoritmą tam tikro uždavinio sprendimui, svarbu tinkamai pasirinkti, kaip kompiuteriu modeliuoti galimus sprendinius (chromosomas). Chromosomos kompiuterio atmintyje standartiškai išreiškiamos bitų eilutėmis, kadangi tai palengvina tiek mutaciją (pakanka apversti kurio nors atsitiktinio bito reikšmę), tiek rekombinaciją (pakanka perkopijuoti pasirinktus tėvinių chromosomų bitus į vaikinę chromosomą). Tiesa, tai nėra vienintelis įmanomas būdas, ir kai kurių uždavinių sprendiniai modeliuojami pvz. grafu ar simbolių eilute. Chromosomų rinkinys, literatūroje dažnai vadinamas populiacija, atspindi uždavinio sprendinių aibę, kuri kinta kiekvieną algoritmo iteraciją. Sprendinio kokybę įvardijame kaip jo tinkamumą, kuris apibrėžiamas tinkamumo funkcijos reikšme, pateikus sprendinį kaip parametrą. Nesunku pastebėti, jog tinkamumo funkcija tėra tikslo funkcijos specializacija, kuri naudojama chromosomų vertinimui. Tinkamumo funkcija yra viena svarbiausių genetinio algoritmo dalių, kadangi kai ji netinkamai parinkta, algoritmas nekonverguos į tinkamą sprendinį arba užtruks labai ilgai. Genetinis algoritmas vykdymo metu iteratyviai atnaujina esamą populiaciją, kurdamas naujas kartas taikant atrankos, rekombinacijos ir mutacijos operatorius. Atrankos operatorius grąžina tinkamiausius populiacijos individus, kuriems yra leidžiama susilaukti palikuonių taikant rekombinacijos operatorių. Rekombinacijos operatorius 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 ne kiekvienam uždaviniui visos jos tinka, kadangi kai kuriais atvejais netinkamai parinkta rekombinacijos strategija pagamina neatitinkančią uždavinio apribojimų vaikinę chromosomą. Pavyzdžiui, jei modeliuojama chromosoma yra sąrašas, turintis susidėti iš tam tikrų elementų, neįmanoma garantuoti, jog atsitiktinai perkopijavus tėvinių chromosomų genus į vaikinę chromosomą šis apribojimas bus išlaikytas. Galiausiai tam tikrai populiacijos daliai yra pritaikomas mutacijos operatorius. Jo veikimo principas yra gana paprastas: pasirinktos chromosomos vienas ar keli genai yra modifikuojami, nežymiai pakeičiant jų reikšmes ar sukeičiant kelių genų reikšmes vietomis. Mutacijos operatorius praplečia vykdomos paieškos erdvę, o tai labai svarbu, kadangi kitaip algoritmas gali konverguoti į lokaliojo minimumo taškus, taip ir nepasiekdamas globaliojo minimumo. Atsižvelgiant į tai, yra laikoma, jog mutacijos operatorius yra kertinė genetinio algoritmo dalis, kuri palaiko genetinę individų įvairovę ir padeda rasti tinkamiausius sprendinius. Jei algoritmas suprojektuotas tinkamai, dažnu atveju vidutinis populiacijos tinkamumas gerės kiekvieną iteraciją. Kad algoritmas neveiktų amžinai, yra pasirenkama tam tikra sustojimo sąlyga, pvz. pasiektas maksimalus kartų skaičius, vidutinio tinkamumo pokytis tarp paskutinių dviejų populiacijos kartų pakankamai mažas.

2.1. Galima paprasto genetinio algoritmo implementacija

Apjungiant visas aukščiau aptartas genetinio algoritmo dalis, galima suformuluoti paprasto genetinio algoritmo pseudokodą.

Algorithm 3 Paprastas GA

Let $current\_population$ be a random population of chromosomes.
Let $mutation\_rate$ be a real value between 0 and 1.
Let $recombination\_fraction$ be a real value between 0 and 1.
repeat
    Apply the fitness function to each chromosome in $current\_population$.
    Sort the chromosomes in $current\_population$ based on their fitness.
    Let $new\_population$ be an empty set.
    repeat
        Let $(parent1, parent2)$ be a pair of parent chromosomes selected randomly from the first $[N * recombination\_fraction]$ elements of $current\_population$.
        Let $child\_chromosome$ be the result of applying the recombination operator to $parent1$ and $parent2$.
        Let $rand$ be a random real value between 0 and 1.
        if $rand < mutation\_rate$ **then**
            Apply the mutation operator to $child\_chromosome$
        end if
        Add $child\_chromosome$ to $new\_population$
    until N offspring have been created.
    Assign $new\_population$ to $current\_population$.
until termination criteria are satisfied.

3. Tarpų sekų efektyvumo kriterijų nustatymas

Šelo algoritmo tarpų sekų efektyvumo įvertinimas nėra trivialus. Rikiavimo algoritmai dažniausiai yra vertinami pagal atliekamų priskyrimų skaičių. Skaičiuojant algoritmo atliekamus priskyrimus, gana paprasta jais išreikšti inversijų skaičių. Daugelyje algoritmų priskyrimų skaičius uždaviniui augant greitai artėja prie palyginimų skaičiaus, tad tokiu metodu gautas įvertis būna pakankamai tikslus. Šelo algoritmo atveju, vien priskyrimų skaičius nėra pakankamai tikslus kriterijus tarpų sekų efektyvumui įvertinti, kadangi remiantis tik juo, gautas įvertis neatspindi praktinio efektyvumo. Kaip parodo [Ciu01], šiame algoritme dominuojanti operacija yra palyginimas. Tad galima daryti išvadą, jog atliekamų palyginimų skaičius yra tinkamesnis kriterijus efektyvumui įvertinti.

Matuojant rikiavimo algoritmo efektyvumą tik naudojant sveikaskaitinius pradinius duomenis, gauti rezultatai gali būti netikslūs, kadangi šių operacijų sparta priklauso nuo rikiuojamų duomenų tipo. Rikiuojant simbolių eilutes, priskyrimas atliekamas naudojant rodykles, kas yra $O(1)$ operacija, tačiau palyginimas yra $O(n)$ blogiausiu atveju. Ir atvirkščiai, rikiuojant įrašus kurie saugomi steke, priskyrimas reikalauja perkopijuoti visą įrašą, o palyginimas gali būti atliekamas naudojant tam tikrą raktą. Todėl apsiriboti vien palyginimų ar priskyrimų skaičiumi nepakanka, kadangi tiksliausia praktinio algoritmo veikimo laiko aproksimacija (tai, kam ir skaičiuojamos operacijos), bus gauta tik įvertinant abu šiuos rodiklius.

Algoritmo veikimo laikas, nors ir priklausomas nuo platformos, kurioje vykdomas tyrimas, detalių, taip pat gali duoti tinkamų įžvalgų įvertinant praktinį efektyvumą. Kadangi algoritmo atliekamos operacijos skaičiuojamos tam, jog gauti praktinio veikimo laiko aproksimaciją, tai realus veikimo laikas yra konkretus įvertis, leidžiantis praktiškai įvertinti duotos sekos efektyvumą. Taigi, įvertinant tarpų sekų efektyvumą bus remiamasi visomis atliekamomis operacijomis bei realiais veikimo laikais.

4. Eksperimentų vykdymo aplinkos paruošimas

4.1. Techninės detalės

Tyrimui buvo naudotas kompiuteris su 2.70 GHz Intel(R) Core(TM) i7-10850H procesoriumi, 32 GB operatyviosios atminties ir Windows 10 operacine sistema. Tyrimas buvo įgyvendintas C++ kalba su GNU g++ 8.1.0 kompiliatoriumi. Genetinio algoritmo implementacijai buvo pasirinkta OpenGA biblioteka [MAM+17] dėl suteikiamos laisvės pasirinkti, kaip įgyvendinti genetinius operatorius bei modernių kalbos konstruktų ir lygiagretaus vykdymo palaikymo.

4.2. Pasiruošimas matavimų atlikimui

4.2.1. Operacijas skaičiuojantis vadovėlinis Šelo algoritmas

Siekant išmatuoti vadovėlinio Šelo algoritmo atliekamų operacijų skaičių, buvo parengtas 4 algoritmas, kuris kaip rezultatą grąžina atliktus palyginimus bei priskyrimus.

4.2.2. Operacijas skaičiuojantis patobulintas Šelo algoritmas

Siekant išmatuoti patobulinto Šelo algoritmo atliekamų operacijų skaičių, buvo parengtas 5 algoritmas, kuris kaip rezultatą grąžina atliktus palyginimus bei priskyrimus.

4.2.3. Veikimo laiko matavimas

Verta pastebėti, jog matuoti veikimo laiką naudojant 4 ir 5 algoritmus nėra tinkama, kadangi jie skaičiuodami atliekamas operacijas vykdo papildomus žingsnius, o tai gali iškreipti gautus rezultatus. Atsižvelgiant į tai, veikimo laiko matavimui naudoti 1 ir 2 algoritmai.

5. Tarpų sekų generavimas

5.1. Galinių tarpų sekų generavimas

Pirmame etape buvo generuojamos tarpų sekos, kurios efektyvios kai $N = 1000$. Toliau šios sekos vadinamos galinėmis sekomis. Remiantis [SY99], galinės sekos chromosoma buvo modeliuojama kaip septynių sveikų skaičių masyvas. Galinių sekų generavimui buvo paruoštas vienkriterinis genetinis algoritmas. Chromosomos buvo vertinamos jas naudojant 20-ties atsitiktinai sugeneruotų sveikų skaičių masyvų rikiavimui vadovėline Šelo rikiavimo algoritmo versija ir skaičiuojant atliktas palyginimo operacijas. Chromosomos tinkamumo funkcija buvo apibrėžta kaip atliktų palyginimų skaičiaus aritmetinis vidurkis. Rekombinacijos operatorius buvo įgyvendintas tolygia strategija, kur abiejų tėvų genai turi vienodą tikimybę būti perduoti vaikinei chromosomai. Mutacijos operatorius buvo įgyvendintas su $\frac{1}{2}$ tikimybe keičiant kiekvieną chromosomos geną, pridedant prie jo atsitiktinį skaičių iš intervalo $[-100, 100]$ su apribojimu, jog pakeistas genas turi priklausyti intervalui $[1, 1000]$. Siekiant išvengti netinkamų sprendinių, kiekviena seka po rekombinacijos ar mutacijos operatorių taikymo buvo išrikiuojama, taip pat užtikrinant, jog pirmas sekos narys yra 1.

Pirmiausia algoritmas buvo vykdomas su atsitiktinai sugeneruota 10000 individų populiacija, kiekvieną iteraciją pritaikant mutacijos ir rekombinacijos operatorius atitinkamai $\frac{8}{10}$ ir $\frac{1}{10}$ populiacijos. Tokiu būdu buvo sugeneruota ir atrinkta 10 sekų. Po to algoritmas buvo vykdomas su atsitiktinai sugeneruota 15000 individų populiacija, dalį jos inicializuojant geriausiomis prieš tai gautomis sekomis ir kiekvieną iteraciją pritaikant mutacijos ir rekombinacijos operatorius atitinkamai $\frac{9}{10}$ ir $\frac{1}{100}$ populiacijos. Tokiu būdu buvo sugeneruota ir atrinkta dar 10 sekų.

Galiausiai iš visų sugeneruotų sekų buvo atrinkta tinkamiausia. Tai buvo atlikta sugeneruotas sekas naudojant 100000 atsitiktinai sugeneruotų masyvų rikiavimui ir matuojant atliktų palyginimo operacijų aritmetinį vidurkį. Matavimai buvo atlikti 5 kartus, siekiant atmesti sekas, gerai pasirodžiusias dėl palankių pradinių duomenų. Žemiau pateikiami atliktų matavimų rezultatai.

Matavimo nr.	Geriausia seka	Vidutinis atliktų palyginimų skaičius
1	974, 669, 137, 40, 13, 5, 1	32204.4
2	974, 669, 137, 40, 13, 5, 1	32205.7
3	974, 669, 137, 40, 13, 5, 1	32201.6
4	974, 669, 137, 40, 13, 5, 1	32202.9
5	974, 669, 137, 40, 13, 5, 1	32203.5

Iš pateiktų duomenų galima daryti išvadą, jog tinkamiausia seka yra $974, 669, 137, 40, 13, 5, 1$. Taip pat verta paminėti, jog ir kitos bandymo metu tirtos sekos pasirodė pakankamai gerai ir vidutiniškai atlikdavo mažiau nei 33000 palyginimų.

5.2. Vidutinio ilgio tarpų sekų generavimas

Antrame etape buvo generuojamos tarpų sekos, kurios efektyvios kai $N = 1000000$. Remiantis [SY99], tarpų sekos chromosoma buvo modeliuojama kaip sveikų skaičių masyvas ilgio 15, kurio paskutiniai septyni elementai yra $974, 669, 137, 40, 13, 5, 1$, t.y. sekos, gautos pirmame etape, nariai. Sekų generavimui teko iš pagrindų pakeisti genetinį algoritmą, naudotą pirmame etape, kadangi žymiai išaugo uždavinio dydis. Buvo atsisakyta chromosomų vertinimo, naudojant 20 skirtingų masyvų, kadangi tai neigiamai įtakojo algoritmo veikimo laiką. Vietoje to, kiekvienos sekos vertinimui buvo naudojamas vienas atsitiktinai sugeneruotas masyvas ilgio 1000000. Populiacijos dydis buvo sumažintas iki 2500, kadangi ties šia riba galutinių sprendinių tinkamumas priklausomai nuo populiacijos dydžio nebuvo pakankamai didelis, jog atsvertų išaugusį veikimo laiką. Taip pat buvo pasirinkta atsisakyti tarpų sekų rikiavimo po mutacijos ir rekombinacijos operatorių taikymo, siekiant sumažinti atliekamų operacijų skaičių. Atlikus keletą bandomųjų algoritmo iteracijų ir iššiaiškinus apytikslius optimalios tarpų sekos narių reikšmių rėžius, chromosomos genų inicializacija buvo perdaryta taip, jog kiekvieno iš chromosomos genų reikšmė priklausytų tam tikram rėžiui, nesikertančiam su kitų chromosomos genų reikšmių rėžiais. Analogiški pakeitimai buvo pritaikyti ir chromosomų mutacijai. Šie pakeitimai taip pat pagerino gaunamų sprendinių kokybę, kadangi dėl jų sumažėjo paieškos erdvė. Pirmiausia buvo sugeneruotos 5 sekos naudojant atsitiktinai sugeneruotą individų populiaciją. Tada buvo sugeneruotos dar 5 sekos, dalį populiacijos inicializuojant prieš tai gautomis sekomis. Iš šių 10 sekų buvo atrinktos 3 tinkamiausios sekos, jas naudojant 100 atsitiktinai sugeneruotų masyvų ilgio 1000000 rikiavimui ir matuojant atliktų palyginimo operacijų aritmetinį vidurkį.

6. Tarpų sekų efektyvumo įvertinimas

Išvados

Gavome tą ir aną, rekomenduojame šitai.

Literatūra

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

[BFM06] Michael A Bender, Martin Farach-Colton ir Miguel A Mosteiro. Insertion sort is O (n log n). *Theory of Computing Systems*, 39(3):391–397, 2006.

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

[For64] G. E. Forsythe. Algorithms. *Commun. ACM*, 7(6):347–349, 1964-06. ISSN: 0001-0782. DOI: 10.1145/512274.512284. URL: https://doi.org/10.1145/512274.512284.

[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.

[HGL+06] Gregory Hornby, Al Globus, Derek Linden ir Jason Lohn. Automated antenna design with evolutionary algorithms. *Space 2006*, p. 7242. 2006.

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

[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.

[Knu72] Donald E Knuth. Ancient babylonian algorithms. *Communications of the ACM*, 15(7):671–677, 1972.

[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.

[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.

[Sew10] Julian Seward. Sort implementation in bzip2. 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).

[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.

[Tur37] Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[Wei91] Mark Allen Weiss. Short Note: Empirical study of the expected running time of Shellsort. *The Computer Journal*, 34(1):88–91, 1991.

[WWG51] Maurice V Wilkes, David J Wheeler ir Stanley Gill. *The Preparation of Programs for an Electronic Digital Computer: With special reference to the EDSAC and the Use of a Library of Subroutines*. Addison-Wesley, 1951.

Priedas Nr. 1

Operacijas skaičiuojantis vadovėlinis Šelo algoritmas

Algorithm 4 Operacijas skaičiuojantis vadovėlinis Šelo algoritmas

```
1:  cmp ← 0; asn ← 0;
2:  foreach gap in H do
3:      for i ← gap + 1 to N do
4:          j ← i; temp ← S[i];
5:          cmp ← cmp + 1; asn ← asn + 2;
6:          while j > gap and S[j − gap] > S[j] do
7:              S[j] ← S[j − gap];
8:              j ← j − gap;
9:              asn ← asn + 2; cmp ← cmp + 2;
10:         end while
11:         if j < gap then
12:             cmp ← cmp + 1;
13:         else
14:             cmp ← cmp + 2;
15:         end if
16:         S[j] ← temp;
17:         asn ← asn + 1;
18:     end for
19: end for
20: return (asn, cmp);
```

Priedas Nr. 2

Operacijas skaičiuojantis patobulintas Šelo algoritmas

Algorithm 5 Operacijas skaičiuojantis patobulintas Šelo algoritmas

```
1:  cmp ← 0; asn ← 0;
2:  foreach gap in H do
3:      for i ← gap + 1 to N do
4:          j ← i; temp ← S[i];
5:          cmp ← cmp + 1; asn ← asn + 2;
6:          while j > gap and S[j − gap] > S[j] do
7:              S[j] ← S[j − gap];
8:              j ← j − gap;
9:              asn ← asn + 2; cmp ← cmp + 2;
10:         end while
11:         if j < gap then
12:             cmp ← cmp + 1;
13:         else
14:             cmp ← cmp + 2;
15:         end if
16:         S[j] ← temp;
17:         asn ← asn + 1;
18:     end for
19: end for
20: return (asn, cmp);
```