VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Genetinis programavimas algoritmų paieškai

Genetic programming for algorithm search

Magistro baigiamasis darbas

Atliko: magistro 2 kurso, 1 grupės studentas

Gedmantas Varkalys (parašas)

Darbo vadovas:

Irmantas Radavičius (parašas)

Recenzentas:

Olga Kurasova (parašas)

Vilnius

2019

Santrauka

Genetinis programavimas - konkretus genetinio algoritmo pritaikymas programavimo uždaviniams, leidžiantis automatizuoti programinio kodo rašymą ir problemų sprendimą. Problemos sprendinys dažniausiai susideda iš tam tikro algoritmo pritaikymo, tačiau nėra aišku ar įmanoma sukurti genetiniu programavimu paremtą karkasą, kuris iš programinio kodo efektyviai išvysto tobulėjančius algoritmus. Literatūroje egzistuoja vos vienas detalus bandymas tokį procesą atlikti, tačiau šis tyrimas yra tik eksperimentinio pobūdžio, įgyvendintas prieš kelis dešimtmečius, o jo metu bandyta išvystyti tik paprasčiausius rikiavimo algoritmus, naudojant tų laikų technologijas. Šio darbo eigoje, genetinis algoritmas yra praplečiamas moderniomis modifikacijomis, pritaikant papildomus genetinius operatorius ir atliekant genetinių operatorių optimizaciją, eksperimentuojama su daug didesne individų populiacija rezultatų korektiškumui užtikrinti. Darbo procesas vyksta laipsniškai pildant rašomų gramatinių taisyklių sąrašus, kas leidžia sukurti vis sudėtingesnius informatikos mokslų algoritmus, o matuojant jų programinio kodo korektiškumą ir spartą, atliekamas palyginimas su viešoje literatūroje sutinkamais algoritmų variantais. Tikimasi, kad šio darbo rezultatai įrodys pasirinkto algoritmų paieškos automatizacijos būdo tinkamumą ir lankstumą įvairiems uždaviniams spręsti, taip didinant genetinio programavimo populiarumą ir susidomėjimą.

Raktiniai žodžiai: genetinis algoritmas, genetinis programavimas, algoritmų paieška, kodo generavimas

Summary

Genetic programming is a specific application of a genetic algorithm for programming tasks that enables automation of software code writing and problem solving. The solution to the problem usually consists of applying a particular algorithm, but it is not clear whether it is possible to create a framework based on genetic programming that efficiently develops progressively improving algorithms. There is only one detailed attempt in the literature to perform such a process, but this study is only experimental, implemented a few decades ago, and attempted to develop only the simplest sorting algorithms using the technology of those times. During this work, the genetic algorithm is expanded upon with modern modifications, applying additional genetic operators and optimizing them while at the same time experimenting with a much larger population of individuals to ensure correctness of results. The work process is done by gradually writing lists of grammatical rules, which allows creating more sophisticated algorithms, while comparing the correctness and speed of their code with the algorithm variants found in public literature. It is expected that the results of this work will prove the appropriateness and flexibility of the chosen algorithm search automation method for solving various tasks, thus increasing the popularity and interest of genetic programming.

Keywords: genetic algorithm, genetic programming, algorithm search, code generation

Turinys

Įvadas 5
1. Literatūros apžvalga ir teoriniai sprendimai 9
1.1. Susiję darbai 9
1.2. Programavimo kalba 11
1.2.1. Kalbos pasirinkimas 11
1.2.2. Kalbos gramatika 12
1.2.3. Automatizuotas kompiliavimas 16
1.3. Genetinis programavimas 18
1.3.1. Pradinis vienetas 18
1.3.2. Genetiniai elementai 20
1.3.3. Genetiniai operatoriai 29
1.3.4. Atnaujintas algoritmas 40
1.3.5. Genetinis tobulėjimas 45
1.4. Operatorių pritaikymo tvarka 47
2. Tiriamoji dalis 49
2.1. Karkasas 49
2.1.1. Vartotojo sąsaja 49
2.1.2. Klasių struktūra 52
2.1.3. Pradinė populiacija 55
2.1.4. Aplinkos sparta 56
2.2. C# gramatikos ypatumai 62
2.3. Išvystyti algoritmai 64
2.3.1. Burbulo algoritmas 65
2.3.2. Tiesinės paieškos algoritmas 70
2.3.3. Išrinkimo rikiavimo algoritmas 73
2.3.4. Įterpimo rikiavimo algoritmas 75
2.3.5. Dvejetainės paieškos algoritmas 78
2.4. Optimizuojami parametrai 81
2.4.1. Vėsinimo imitacijos algoritmas 86
2.5. Kodo vertinimas 88
Rezultatai ir išvados 91
Literatūros sąrašas 94

Įvadas

Algoritmai - veiksmų sekos, padedančios spręsti paprastus ir sudėtingus uždavinius. Sekos dydis dažniausiai priklauso nuo uždavinio sudėtingumo. Algoritmas gali būti pritaikytas ne tik vienam konkrečiam atvejui, bet ir visai uždavinių klasei, o kiekvienas uždavinys gali turėti net kelis algoritmus, gaunančius korektišką rezultatą. Tokių algoritmų veiksmų sekos gali drastiškai skirtis. Logiška pasirinkti tą, kuris norimą rezultatą gauna greičiausiai. Informatikos moksluose kompiuterinę programą taip pat galima vadinti algoritmu. Programa vykdo veiksmų sekas, kurių pabaigoje gaunamas tam tikras rezultatas – uždavinio sprendinys. Nors algoritmai ir padeda įveikti įvairias problemas, jų radimo būdai nėra visada aiškūs.

Algoritmo suradimas įrodo, kad uždavinys yra išsprendžiamas, tačiau tai negarantuoja jo efektyvumo. Efektyvumą labiausiai įtakoja tokie kriterijai kaip algoritmo greitis, kompiuterinių resursų sąnaudos, tikslumas. Nuo efektyvumo priklauso kaip dažnai sukurtas algoritmas bus naudojamas. Duomenų kiekis, kurį apdoroja algoritmas, gali drastiškai įtakoti veiksmų sekos sudarymą. Šiais laikais ypač populiaru dirbti su didžiaisiais duomenimis (angl. *big data*), ko pasėkoje net ir vienas perteklinis žingsnis gali neigiamai paveikti algoritmo darbo efektyvumo įvertinimą. Vis dėl to, net ir labai efektyvūs algoritmai gali užtrukti per ilgai.

Efektyvumo problemai spręsti sudaromi nauji algoritmai, kurie ieško ne pačio geriausio uždavinio sprendinio, o apsistoja ties patenkinamai geru. Tokie algoritmai vadinami euristiniais algoritmais. Jų pagalba išlošiama greitesnė darbo eiga, paaukojant tikslumą. Euristinis algoritmas, pagal įvairias euristines taisykles ar pastebėjimus sugeba atmesti ir netirti tam tikros uždavinio sprendinių aibės dalies. Algoritmus galima modifikuoti, keisti euristinių taisyklių sąrašus, plėsti ar mažinti tiriamą aibę. Euristinių algoritmų efektyvumas priklauso nuo korektiško pritaikymo tiriamajam uždaviniui, parametrų parinkimo, todėl tai ne visada yra geriausias būdas problemai spręsti.

Uždaviniuose, kurių tikslus ir korektiškas sprendimas yra kritiškai svarbus, euristinių algoritmų pagalba gauti rezultatai negarantuoja teisingo veikimo kiekvieną kartą kartojant tą patį tyrimą. Deterministinių algoritmų pranašumas yra tame, kad jie visada suras uždavinio optimumą, todėl specifinėse situacijose apsieiti be jų tiesiog neįmanoma. Iš to kyla griežti reikalavimai deterministinių algoritmų efektyvumui, ypač dirbant su dideliais duomenų kiekiais. Algoritmų paieškos metu surasti efektyviausią algoritmo variantą nėra paprasta. Kadangi tokį darbą dažniausiai atlieka žmogus, daugelis algoritmo variantų nėra visada ištiriami. Vis dėl to, deterministinių algoritmų paieškai įmanoma pritaikyti euristinius algoritmus.

Genetinis algoritmas – vienas iš euristinėmis taisyklėmis grįstų algoritmų. Pagrindinė genetinio algoritmo idėja yra dirbti su tam tikra tiriamų individų populiacija. Darbas vyksta modifikuojant individų genetinę informaciją, pritaikant tam tikrus operatorius veiksmams atlikti. Šis algoritmas yra grįstas biologinės evoliucijos sėkme ir gali būti pritaikomas įvairioms tyrimų sritims. Viena iš jų yra ir algoritmų paieška. Šiuo atveju, genetinio algoritmo individas yra vienas iš ieškomo algoritmo variantų, sugeneruotas iš anksto ir tam tikru būdu. Apie individų generavimą kalbama tolesniuose skyriuose. Individo genai – algoritmo veiksmų sekos elementai. Darbas vyksta, pritaikant standartinius genetinio algoritmo operatorius – atranką, kryžminimą ir mutaciją. Atrankos operatorius atsižvelgia į individo tinkamumo įvertinimą ir nustato geriausią populiacijos dalį. Kryžminimas leidžia iš dviejų individų sukurti vieną ar daugiau naujų individų, besiremiančių jų genetine informacija. Algoritmų paieškos atveju, kryžminimą galima įsivaizduoti kaip dviejų algoritmų veiksmų sekų sumaišymą, išlaikant logišką algoritmo veikimą. Mutacijos operatorius yra unikalus tuo, kad jo pritaikymas yra tikimybinis ir įvyksta sąlyginai retai. Mutacijos idėja – pakeisti vieną individo geną, taip išlaikant populiacijos genetinę įvairovę. Genetinis algoritmas turi ir keletą trūkumų, į kuriuos reikia atsižvelgti. Jo darbo sėkmė priklauso nuo parametrų parinkimo, reikia išvengti individų supanašėjimo, todėl populiacijos individų skaičiaus parinkimas bei šių individų sudarymas turi išlaikyti įvairovę. Tačiau yra ir teigiamų dalykų. Genetinį algoritmą galima modifikuoti. Svarbiausios modifikacijos, atliekant algoritmų paiešką, yra darbo lygiagretinimas, lokalios paieškos pritaikymas bei naujų operatorių pridėjimas. Genetinis algoritmas remiasi euristinėmis taisyklėmis ir tinka algoritmų paieškos uždaviniui, pritaikant jį darbui su programiniu kodu.

Genetinis programavimas – konkretus genetinio algoritmo pritaikymas programavimo uždaviniams. Kadangi kompiuterinę programą galima traktuoti kaip algoritmą, ją vadinsime vienu individu. Individo genas – programos eilutė. Tokiu būdu, genetinis algoritmas gali dirbti su kompiuterinėmis programomis tiesiogiai, atlikti manipuliaciją su eilutėmis, modifikuoti, pašalinti ar pridėti naujų eilučių. Tai yra automatinis, euristinėmis taisyklėmis besiremiantis programavimas. Genetinis programavimas leidžia automatizuoti algoritmų paiešką. Turint du skirtingus individus, jų kryžminimas vykdo kodo sintezę (angl. *synthesis*), taip maišant programinį kodą. Logiška uždrausti genetiniam algoritmui modifikuoti esminę programos dalį, susijusią su jos korektišku įvykdymu ir apsistoti tik ties eilutėmis, aprašančiomis patį algoritmo darbą. Atsižvelgiant į tai, pradinės populiacijos generavimas taip pat kuria tik patį algoritmą, o ne programai reikalingą informaciją. Algoritmui pateikiamas tuščios programos karkasas, kuriame yra visa reikalinga informacija jos paleidimui, tačiau nėra jokių papildomų veiksmų. Pradinės populiacijos generavimo metu ji yra užpildoma algoritmo veiksmais. Žinoma, konkretūs veiksmai priklauso nuo programavimo kalbos pasirinkimo. Tai yra esminis šio darbo žingsnis. Kadangi

algoritmo kūrimas nėra lengvai apibrėžiamas, verta įsigilinti į konkrečios programavimo kalbos gramatiką. Gramatinės kalbos taisyklės nurodo, kokius veiksmus bus įmanoma atlikti korektiškai. Tiriant tai, galima sukurti teoriškai grįstą kelią nuo tuščios pradinės programos iki programos, įgyvendinančios algoritmą. Jeigu tai pavyksta, tada galima tikėtis, kad atliekant genetinio programavimo žingsnius, bus artėjama link korektiško rezultato.

Genetinio programavimo pagalba gauto algoritmo įvertinimas taip pat kelia tam tikrų problemų. Neužtenka patikrinti ar sukurtas algoritmas gauna korektišką rezultatą. Vertinimas turi būti grįstas tam tikromis programavimo metrikomis. Kodo eilučių skaičius, kintamųjų skaičius, loginių išsišakojimų skaičius, visa tai leidžia palyginti dvi, tą patį rezultatą gaunančias programas. Šio tyrimo atveju, svarbu gauti kuo efektyvesnį algoritmą, kuris galėtų dirbti ir testavimo aplinkoje, ir įsivaizduojamoje didžiųjų duomenų tyrimo situacijoje. Taigi, remiantis įvertinimu, galutinis rezultatas leis pasakyti ar gautas algoritmas yra efektyvus.

Šis darbas susideda iš sudėtingo karkaso kūrimo. Karkaso idėja – pritaikyti genetinį algoritmą darbui su pasirinkta programavimo kalba, algoritmų paieškos uždaviniui spręsti. Tam pasiekti buvo įgyvendinti tam tikri esminiai elementai. Karkasas sugeba sukurti korektišką individų populiaciją bei atlikti genetinio algoritmo operatorių veiksmus su jais. Genetinio programavimo darbas išlygiagretinamas skaičiavimų spartinimui. Pritaikyta lokalios paieškos sistema, kuri imituoja programuotojo atliekamą reorganizavimo veiklą. Genetinio algoritmo darbas automatiškai optimizuojamas, pritaikant mašininio mokymosi idėjas, palaipsniui keičiant visus parametrus kuo geresniam rezultatui gauti.

Šio **darbo problema** – ar įmanoma genetinio programavimo pagalba efektyviai išvystyti algoritmą.

Šio **darbo tikslas** – sukurti genetiniu programavimu paremtą karkasą, kuris iš programinio kodo efektyviai išvysto tobulėjančius algoritmus.

Šiame darbe bus bandoma išvystyti paieškos ir rikiavimo algoritmus. Iki šiol panašių tyrimų atlikta labai nedaug ir jie yra daryti, kai kompiuteriai nebuvo pakankamai galingi, todėl tai dar nėra galutinai išspręstas uždavinys. Paieška ir rikiavimas – klasikiniai informatikos mokslų uždaviniai. Tokie uždaviniai turi ne vieną teisingą sprendimą, kurie buvo sukurti ir sugalvoti bėgant laikui. Galbūt egzistuoja dar vienas ar keli algoritmai, apie kuriuos iki šiol nežinoma.

Šio **darbo uždaviniai**:

1. Apžvelgti su tema susijusią literatūrą ir atlikti problemos analizę.
2. Apžvelgti programavimo kalbų gramatikas, pasirinkti konkrečią kalbą.
3. Pritaikyti genetinį algoritmą genetiniam programavimui, uždaviniams bei pasirinktai programavimo kalbai.
4. Apžvelgti kodo generavimo būdus.

5. Sukurti algoritmų mokymosi karkasą, reikalingą tyrimams atlikti.
6. Atlikti genetinių operatorių analizę ir derinimą.
7. Atlikti genetinių algoritmų optimizavimą, pritaikant modifikacijas ir mašininį mokymąsi.
8. Atlikti gautų algoritmų kodo sudėtingumo analizę ir su tuo susijusių matų apžvalgą.

Darbą sudaro du pagrindiniai skyriai, skirstomi į detalius poskyrius. Pirmame skyriuje pristatoma susijusi literatūra, jos analizė, bei priimti teoriniai sprendimai. Argumentuojami programavimo sprendimai ir genetinio algoritmo pritaikymas uždaviniui. Antrame skyriuje, pirmame poskyryje, pristatomas darbo metu sukurtas programinis karkasas, skirtas tyrimams atlikti. Antrame poskyryje aptariami programavimo kalbos ypatumai į kuriuos reikėjo atsižvelgti. Antro skyriaus trečiame poskyryje aptariami visi informatikos mokslų algoritmai, kurie buvo išvystyti sukurto programinio karkaso pagalba. Tai yra keturi pilnai išvystyti, idealų programinį kodą gaunantys algoritmai bei vienas nepilnai išvystytas algoritmas. Darbo procesas ir tyrimų rezultatai, kurie lydi link sėkmingo įvertinimo detaliai aptariami poskyrio skirsniuose. Ketvirtame antro skyriaus poskyryje pristatomas įgyvendintas vėsinimo imitacijos algoritmas bei karkaso parametrų optimizacija ir geriausios reikšmės. Penktame poskyryje pristatomas detalesnis galutinių, išvystytų programų programinio kodo vertinimo procesas, kurio pagalba buvo galima įvertinti gautą rezultatą ir padaryti teigiamas išvadas, kurios rodo, kad šio darbo tikslas buvo pasiektas.

1. Literatūros apžvalga ir teoriniai sprendimai

Šiame, teoriniame, skyriuje aptariama su tematika susijusi literatūra ir svarbiausi priimti sprendimai. Pirmiausia, detaliai apžvelgiami straipsniai, kurie glaudžiai susiję su darbo tikslu. Toliau pristatomas argumentuotas programavimo kalbos pasirinkimas ir aukšto lygio programavimo sprendimai. Sekančiu etapu aptariamas genetinis algoritmas bei genetinis programavimas, detaliai apžvelgiant visas esmines detales, daugelį iš jų pagrindžiant konkrečiais pavyzdžiais iš literatūros. Pateikiami žemo lygio programinio kodo pavyzdžiai. Analogiškai pristatomi genetiniai operatoriai, trumpai aptariant straipsnius iš kurių kilo jų pritaikymo idėjos bei pateikiamas supaprastintas programinis kodas, kuris atspindi teorinių sprendimų įgyvendinimo detales. Sekančiame skirsnyje aptariamas pilnas genetinis algoritmas su pasirinktomis modifikacijomis, trumpai pristatomas programinis kodas ir algoritmo veiksmų srautas. Pristatomas genetinis tobulėjimas, su tuo susijusi literatūra ir konkretūs teoriniai sprendimai, kurie buvo priimti darbo eigoje. Skyriaus pabaigoje aptariama genetinių operatorių pritaikymo tvarka ir teoriniai sprendimai, kurių pagalba ji buvo sudaryta.

1.1. Susiję darbai

Algoritmų paieška genetinio programavimo pagalba yra gana mažai ištirta mokslo šaka. Autorius Kenneth E. Kinnear Jr, 1993 metais pateikė straipsnį pavadinimu „Evolving a Sort: Lessons in Genetic Programming“[Kin93], kuriame genetinio programavimo paradigma buvo pritaikyta rikiavimo algoritmo sukūrimui. Autoriaus tikslas – išvystyti paprastą algoritmą genetinio programavimo pagalba. Straipsnyje aprašytas atliktas eksperimentas, technologijos koncepcijos įrodymas, o išmoktos pamokos dokumentuotos tolesniems tyrimams atlikti. Tyrimo metu dirbta su LISP programavimo kalbos funkcijomis. Funkcijų tinkamumo lygis matuojamas joms paduodant neišrikiuotus elementų masyvus. LISP funkcijos juos apdoroja, o rezultate matuojamas elementų išrikiavimo korektiškumas. Kad genetinis programavimas veiktų, autorius pirmiausia atliko tiriamosios srities pritaikymą evoliucionavimo paradigmai. Terminaliniais simboliais, kurių negalima keisti, buvo pažymėti LISP kalbos konstantos ir kintamieji. Gautas primityvų rinkinys, kuris sudarytas iš septinių funkcijų ir dviejų terminalų. Terminalai yra ***index***, kuris žymi ciklo skaitliuką ir ***len***, kuriuo nurodomas duomenų segmento ilgis. Funkcijos yra tokios:

(dobl start end work)

(swap x y)

(el+ x)

(el- x)

(wismaller x y)

(wibigger x y)

(e- x y)

Čia ***start***, ***end***, ***work***, ***x*** ir ***y*** yra neterminaliniai simboliai. Ką šios funkcijos atlieka, galima suprasti įsigilinus į autoriaus straipsnį, šio darbo kontekste tai nesvarbu. Tokio rinkinio pagalba genetinis algoritmas kūrė ir vystė įvairius rikiavimo algoritmus. Vienas iš geriausių rezultatų pateiktas 1-ame paveikslėlyje.

```
(dobl (wismaller (wismaller (el- len) len)
                 index)
      (dobl (wismaller index
                       (wismaller
                        (el- index)
                        (el+ (el- index))))
            (el- len)
            (swap (swap (el- len) index)
                  index))
      (dobl (swap (wibigger index (e- index len))
                  (e- index len))
            (el- len)
            (swap (wismaller (el+ index) index)
                  index))))
```

1 pav. *LISP programos kodas*

Matome, kad gautas kodas yra sunkiai skaitomas, tačiau jis veikia. Toks srities pritaikymas leidžia dirbti ne su bereikšmėmis simboliu eilutėmis, o su programavimo kalbos gramatikos elementais. Tai panaudota ir šio darbo metu. Apie pasirinktą programavimo kalbą ir kalbos gramatikos analizę detaliai kalbama sekančiame poskyryje.

Straipsnio rezultate sukurtos programos, kurios rikiavo duomenis, buvo gana sudėtingos ir sunkiai analizuojamos. Dėl daugybės skirtingų duomenų variantų, kuriuos būtų galima paduoti šioms programoms, autorius negali griežtai įrodyti, kad sukurti algoritmai yra korektiški bendruoju atveju. Tačiau, patikrinus juos su dideliais duomenų kiekiais, gaunami korektiški rezultatai. Tai parodo, kad nors programos yra sunkiai analizuojamos, jos pateikia teisingus rezultatus.

Autoriaus Kenneth E. Kinnear Jr, straipsnis yra bene vienintelis mokslinis tyrimas glaudžiai susijęs su šio darbo tema. Sekančiuose skyriuose prie straipsnio bus sugrįžta, atsižvelgiant į išmoktas pamokas ir gautus rezultatus.

1.2. Programavimo kalba

Genetinio programavimo pritaikymas algoritmų paieškai priklauso nuo konkrečios programavimo kalbos pasirinkimo. Reikia atsižvelgti ne tik į kalbos gramatiką ir jos paprastumą tiriamajam uždaviniui, bet ir į asmeninę praktiką. Šiame skyriuje pirmiausiai aptariamas ir argumentuojamas kalbos pasirinkimas, o toliau pristatoma ir išanalizuojama kalbos gramatika bei esminiai elementai, reikalingi genetinio programavimo pritaikymui algoritmų paieškos uždaviniui spręsti.

1.2.1. Kalbos pasirinkimas

Atliekant manipuliaciją su dinaminėmis duomenų struktūromis, grafinėmis vartotojo sąsajomis ar kitais intensyviais elementais, dažniausia linkstama link tų programavimo kalbų, kurios dirba arti kompiuterio architektūros, leidžia laisvai manipuliuoti atmintimi. Labiausiai paplitusios yra C ir C++ programavimo kalbos. Jos leidžia programuotojui laisvais dirbti su kompiuterio resursais. Tačiau šių kalbų sintaksės sudėtingumas ir kodo eilučių skaičius, reikalingas veiksmams atlikti, gali sukelti tam tikrų problemų. Verta apžvelgti ir kitas populiarias programavimo kalbas. Tam, kad genetinis programavimas būtų kuo paprastesnis, svarbu turėti aiškią ir suprantamą programavimo kalbos sintaksę, kuri leidžia nesunkiai manipuliuoti programos eilutėmis, keisti jų tvarką. Populiariausios programavimo kalbos, atitinkančios tokį reikalavimą, yra Java, C#, Python. Šios kalbos yra labai plačiai naudojamos, egzistuoja daug naudingų kodo bibliotekų, padedančių spręsti įvairias užduotis. Python programavimo kalba yra aukšto lygio. Didelė dalis programavimo logikos šioje kalboje yra užslėpta, ją sunku keisti, todėl šiam tyrimui Python tinka mažiausia. C# ir Java programavimo kalbos savo sintakse yra gana panašios. Atsižvelgiant į asmeninę patirtį, C# programavimo kalba yra tinkamiausia darbo tikslui pasiekti. Nors C# yra sąlyginai aukšto lygio, šios kalbos pasirinkimas leis atlikti geresnį genetinio programavimo pritaikymą darbo tikslui pasiekti, o skaičiavimų greitis nebus labai nutolęs nuo žemesnio lygio kalbų.

Peter Sestof, savo straipsnyje [Ses10] palygina C, C# ir Java programavimo kalbas. Lyginimas programų veikimo greitis, atliekant skaitinius skaičiavimus. Naudojama paprasta, ne specifinė techninė įranga – nešiojamas kompiuteris. Tyrimas atliktas tiriant keturias sritis: matricų dauginimas, ciklas su daug dalybos veiksmų, daugianario įvertinimas ir paskirstymo funkcijos skaičiavimas. Gauti rezultatai parodo, kad nei viena iš tirtų programavimo kalbų nebuvo dominuojanti. Tyrimo metu kiekviena iš programavimo kalbų buvo greičiausia ir lėčiausia

skirtingais atvejais. C# ir Java programavimo kalbos neatsilieka nuo C atliekant skaitinius skaičiavimus, o tam tikrais atvejais, naudojant nesaugų kodą (angl. *unsafe code*), leidžia C# kalbai dirbti žymiai efektyviau. Nors šis tyrimas nėra susijęs su algoritmų paieška, jo rezultatai leidžia lengviau įvertinti ir argumentuoti programavimo kalbos pasirinkimą.

Genetinio programavimo veiksmus galima apibendrinti iki darbo su masyvais. Kadangi Peter Sestof atlikti skaičiavimai taip pat remiasi gana paprastomis operacijomis, galima teigti, kad gauti veiklos rezultatai turėtų būti gana panašūs ir šio tyrimo atveju. Sukurto programinio karkaso veikimo laikas ir lėčiausių vietų analizė detaliai aprašyta šio darbo 2.1.4 skirsnyje. Anksčiau į kandidatus pasirinkta C# programavimo kalba neatsilieka nuo C skaičiavimų spartos atžvilgiu, todėl tuo dar kartą galima pagrįsti jos pasirinkimą.

Algoritmai dažniausiai susideda iš pakankamai paprastų programavimo konstruktų: ciklai, palyginimai, priskyrimo operacijos, paprastos matematinės operacijos. Atsižvelgiant į tai, nėra reikalingos naujausios ir tobuliausios C# programavimo kalbos ypatybės. Jos ne tik žymiai padidintų algoritmo sudėtingumą, bet ir nepasiūlytų to, ko negalima išreikšti paprastesnėmis funkcijomis. To pasėkoje nuspręsta algoritmų generavime apsistoti ties **C# 3.0** programavimo kalbos versija. Taip pat, visos aplinkos kūrimui pasirinkta **.Net Framework 4.6.1** versija, kuri palaiko **C# 7.0** kalbos variantą. Tai yra viena tobuliausių ir labiausiai palaikomų programavimo aplinkų, todėl šio darbo kontekste toks sprendimas duoda didžiausią naudą.

1.2.2. Kalbos gramatika

Programavimo kalbos gramatika yra svarbi šio darbo tyrimo sritis. Kadangi genetinis programavimas tiesiogiai dirba su programiniu kodu, geros kalbos žinios leis lengviau ir efektyviau jį pritaikyti. Kiekvienas sakinio sudarymas remiasi tam tikromis taisyklėmis. Programavimo kalba turi unikalius raktinius žodžius ir simbolius, kurių netaisyklingas vartojimas sukels papildomų problemų. Svarbu tokioms problemos iškart užkirsti kelią . Kenneth E. Kinnear, Jr atliktame tyrime [Kin93], kuris buvo pristatytas 1.1 poskyryje, kalbos gramatikos taisyklės yra panaudotos primityvų sudarymui. Tokį principą galima panaudoti ir šio darbo kontekste. Pirmiausia svarbu susipažinti su programavimo kalbos gramatika, tačiau dažniausiai ji nėra laisvai prieinama. Ne išimtis ir C# programavimo kalba – šaltiniuose egzistuoja tik pasenę, nepilni jos gramatikos variantai.

Autoriai R. Lammel ir C. Verhoef, straipsnyje [LV01] pasiūlo būdus ir technologijas kaip egzistuojančios kalbos gramatiką atkurti iš laisvai prieinamų resursų. Pagrindinis darbo procesas šiame tyrime vyksta, tiriant ir analizuojant su programavimo kalba susijusius šaltinius, procesų

aprašymus, dokumentaciją. Bandoma atkurti korektišką gramatikos taisyklių aprašymą, sudarant kalbos BNF (Backus-Naur forma). Tai jiems pavyksta padaryti. Tolimesniais etapais autoriai gautus rezultatus panaudoja vartotojo žinynų (angl. *user manual*) automatiniam generavimui bei kalbos analizatoriaus (angl. *parser*) kūrimui. Rezultate autoriams pavyko atkurti visiškai korektišką programavimo kalbos VS COBOL II gramatiką bei sukurti šios kalbos analizatorių. Straipsnyje pasiūlyti būdai kaip iš laisvai prieinamų programavimo kalbos resursų atkurti kalbos gramatikos specifikaciją yra naudingi. Tai panaudota tiriant pasirinktąją C# programavimo kalbą ir kuriant konkrečias genetinio programavimo taisykles tolimesniuose šio darbo etapuose. Šiame šaltinyje pasiūlytos idėjos leidžia lengviau spręsti kilusias sintaksės ir gramatinių ypatumų problemas.

Labai panašus ir Vadim Zaytsev parašytas straipsnis [Zay], kuris sutelkia dėmesį būtent į C# programavimo kalbos gramatikos atkūrimą. Autorius aprašo būdus kaip iš kalbos standarto išgauti naudingą informaciją, sugeneruoti kalbos gramatikos taisykles BNF ar EBNF (išplėsta BNF) formatu bei toliau su ja dirbti analizatoriaus kūrimui. Visas tyrimo procesas yra įdomus ir detalus, tačiau šiam darbui aktualus tik gramatikos taisyklių sudarymas. Verta paminėti, kad tyrimo rezultate buvo sukurtas ir patikrintas pilnas kalbos analizatorius, besiremiantis gauta gramatika. Kaip ir anksčiau minėto straipsnio aprašyme, Vadim Zaytsev pasinaudoja laisvai prieinamais programavimo kalbos resursais. Autorius siūlo naudoti EBNF gramatikos aprašymo formatą, nes jis yra laisvai gaunamas iš kitų formatų ir yra lengviau analizuojamas. Tuo remiamasi šio darbo eigoje.

Minėtų straipsnių idėjų panaudojimas atsispindi kuriant konkrečią programinę aplinką. Tai detaliai aprašyta 2.1 poskyryje.

Pilnas C# gramatikos aprašymas EBNF forma yra ilgas ir sudėtingas, todėl teoriškai aptariamos tik esminės jos dalys. Kadangi algoritmų paieškos darbo metu bus kuriamas tik algoritmas, o ne visa programa, svarbu suprasti tik kaip apibrėžiamos pagrindinės programos eilutės. Plačiau apie algoritmo programos rašymo detales kalbama 1.3.1 skirsnyje. Esant programos viduje, vienos eilutės užrašymas lygus vieno sakinio (angl. *statement*) užrašymui. EBNF formatu, C# sakinys aprašomas taip:

statement ::=
labeled_statement
declaration_statement
embedded_statement

Šis gramatikos sakinys sako, kad sutikus elementą ***statement***, jį galima pakeisti į vieną iš trijų naujų sakinių. Sakinys ***embedded_statement*** šifruoja tam tikro kodo segmento kūrimą. Šis sakinys detalizuojamas tokia taisykle:

embedded_statement ::=
block
empty_statement
expression_statement
selection_statement
iteration_statement
jump_statement
try_statement
checked_statement
unchecked_statement
lock_statement
using_statement
yield_statement
embedded_statement_unsafe

Kaip matome, galimi įvairūs tolimesni variantai. Paprasto ***block*** sakinio atveju prieinama iki štai tokios EBNF aprašytos gramatikos taisyklės:

block ::=
{ statement_list? }

Pristatomi du terminaliniai simboliai ***{*** ir ***}***, o simbolis ***statement_list?*** yra nebūtinas, nes pažymėtas klaustuku. Šiuo būdu aprašomas vienas C# kodo segmentas, atskirtas figūriniais skliaustais, kurio viduje yra nulis arba daugiau sakinių. Tyrimo metu ypač svarbu suprasti kaip gramatiškai apibrėžiamas loginis išsišakojimas. Iš anksčiau minėtos ***embedded_statement*** taisyklės, pereinant į ***selection_statement*** sakinį, vienas iš jos rezultatų bus loginis išsišakojimas ***IF***. EBNF bendrasis loginio ***IF*** atvejis atrodo taip:

if_statement ::=
if (boolean_expression) embedded_statement
if (boolean_expression) embedded_statement else embedded_statement

Matome, kad egzistuoja terminaliniai žodžiai ***IF, ELSE*** bei terminaliniai simboliai ***(*** ir ***)***. Atliekant genetinio programavimo darbą būtina atsižvelgti į šių ir anksčiau minėtų terminalinių simbolių sąrašą bei raktinių programavimo kalbos žodžių sąrašą. Visa tai yra detaliai apibrėžta kalbos gramatikoje, todėl šią informaciją gauti nesunku. Generuojant programinį kodą užtikrinta,

kad tokie simboliai nebūtų naudojami jiems netinkamose vietose, pavyzdžiui kintamųjų varduose. Tai ypač svarbu generuojant visiškai naujas kodo eilutes, atliekant genetinės mutacijos operaciją, apie kurią detaliai kalbama 1.3.3.3 straipsnyje.

Taigi, žinant kalbos gramatiką, svarbu išsiaiškinti ar įmanoma išvesti visiškai korektišką algoritmo kodą. Tai svarbu, nes genetinio programavimo metu karkasas išbandys visas įmanomas kodo eilučių kombinacijas, pradedant nuo pradinio vieneto. Jeigu to padaryti neįmanoma, sekančiame šio darbo etape genetinis algoritmas niekada nepasieks ieškomos problemos optimumo taško. Taigi, pabandykime gramatikos taisyklėmis apibrėžti paprastą Burbulo rikiavimo (angl. *Bubble sort*) algoritmą. Šio algoritmo kodas C# programavimo kalba pateiktas 2-ame paveikslėlyje.

```
int temp = 0;

for (int write = 0; write < arr.Length; write++)
{
    for (int sort = 0; sort < arr.Length - 1; sort++)
    {
        if (arr[sort] > arr[sort + 1])
        {
            temp = arr[sort + 1];
            arr[sort + 1] = arr[sort];
            arr[sort] = temp;
        }
    }
}
```

2 pav. *Burbulo rikiavimo algoritmas*

Žemiau pateikiamas algoritmas aprašytas nepilnai išskleistomis gramatikos taisyklėmis. Pirmąją kodo eilutę, kintamojo sukūrimą, galima užrašyti kaip tokių sakinių seką:

statement ::=
declaration_statement ::=
local_variable_declaration ::=
local_variable_type local_variable_declarators

Sekantis veiksmas yra sudėtingesnis, susidedantis iš kelių esminių dalių. Paprastesniam vaizdavimui, pirmiausia aprašykime vienos išraiškos sakinių eigą, tai atspindės viena iš trijų kodo eilučių, dirbančių su masyvo elementais.

expression_statement ::=
statement_expression ::=
assignment ::=
unary_expression assignment_operator expression

Tokios išraiškos sudarymo sakinių seka yra žymiai ilgesnė, bet detalizuoti sakiniai tiesiog atrenka operacijos narių tipus, operacijos ženklą ir taip toliau, teigiame, kad jie yra trivialūs. Šią seką sutrumpintai pažymėsime ***_priskyrimo_išraiška_***.

Sekančiu žingsniu aprašome likusį algoritmo kodą:

statement ::=
embedded_statement ::=
iteration_statement ::=
for_statement ::=
for (for_initializer? ; for_condition? ; for_iterator?) embedded_statement ::=
for_statement ::=
for (for_initializer? ; for_condition? ; for_iterator?) embedded_statement ::=
selection_statement ::=
if_statement ::=
if (boolean_expression) embedded_statement ::=
_priskyrimo_išraiška_
_priskyrimo_išraiška_
_priskyrimo_išraiška_

Kaip matome, pradedant sakinio simboliu ***statement***, logiška gramatikos elementu seka aprašomi visi algoritmo žingsniai. Tai parodo, kad ir genetinio programavimo metu, genetiniam algoritmui tiriant labai didelę visų įmanomų algoritmų aibę, jis tikrai turės tikimybę pasiekti optimumo tašką. Iš tuščio pradinio programos elemento sėkmingai gautas pilnas Burbulo algoritmas.

Šis teorinis eksperimentas, įgyvendinant Burbulo algoritmą, taip pat realizuotas praktiškai. Eksperimento rezultatai ir išvados detaliai išdėstytos 2.3.1 skirsnyje.

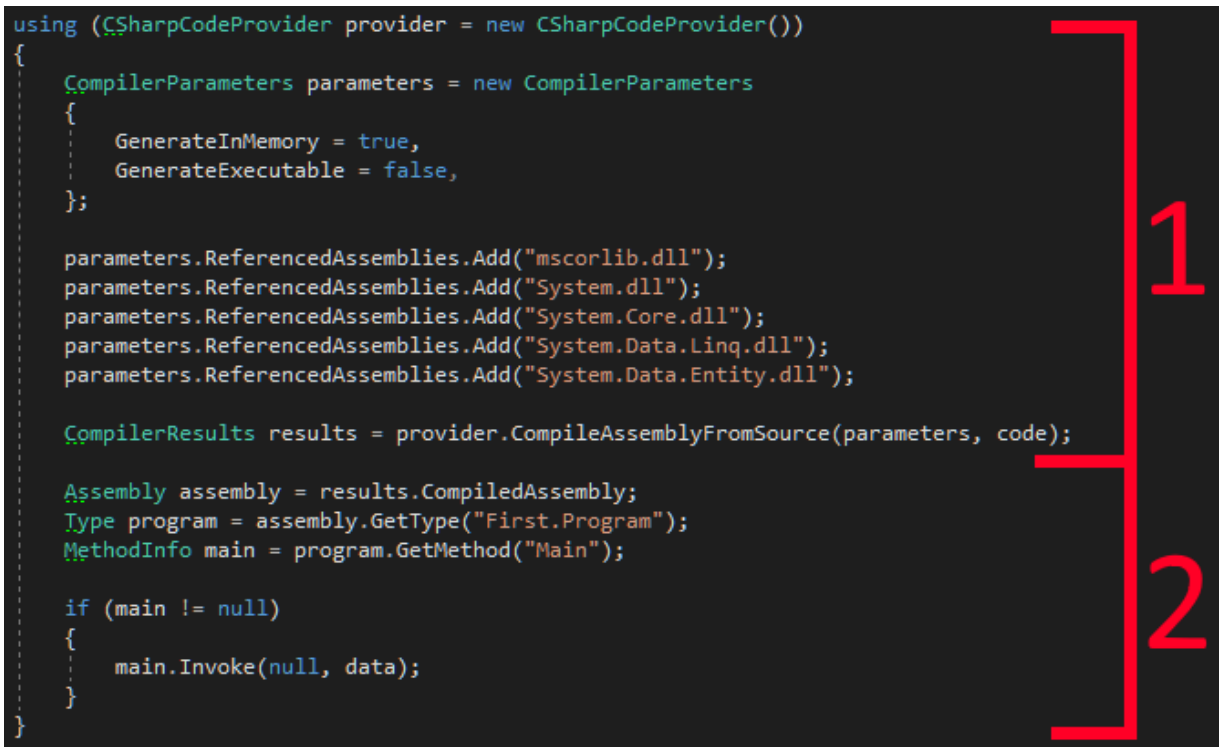
1.2.3. Automatizuotas kompiliavimas

Labai svarbi programavimo aplinkos dalis yra programinio kodo kompiliavimas. Atliekant algoritmų paiešką genetinis algoritmas kuria programinį kodą iš skirtingų elementų, todėl galutinis rezultatas ne visada yra teisinga ir korektiška programa. Taip pat, programos darbo rezultatas gali žymiai nukrypti nuo norimo, nes genetinis algoritmas turi atsitiktinumo. Sukurtą programinį kodą galima traktuoti kaip visiškai naują programą, ją išskirti, sukompiliuoti, paleisti ir pasižymėti gautą rezultatą, bet tai užimtų labai daug laiko.

Sukurto karkaso pagalba kuriamų ir vystomų algoritmų kodas įterpiamas į parengtą minimalią programą. Apie tai detaliai kalbama 1.3.1 skirsnyje. Programų korektiškam veikimui patikrinti reikalingas programinio kodo kompiliavimas. Atskirų vykdomųjų failų kūrimas ir

naudojimas užtrunka per ilgai ir kainuoja ne tik papildomą skaičių resursų, bet ir verčia sistemą rūpintis šių failų išvalymu. Problemai spręsti yra kitų būdų. Autorius Lumír Kojecký, internetiniame straipsnyje [Koj14] pateikia trumpą ir paprastą būdą kodui kompiliuoti dinamiškai, karkaso viduje, be atskirų pridėtinių išlaidų. Autorius savo trumpame straipsnyje pateikė idėjas ir būdus kaip atlikti daug kitų papildomų funkcijų bei pats atliko atskirus greičio įverčius šio metodo patikrinimui. Rezultate gauta, kad kodas, kompiliuotas šiuo būdu, greičiu visiškai neatsilieka nuo klasikinio kodo kompiliavimo. Lumír Kojecký straipsnis duoda esminę idėją kuriamo karkaso kodavimui, kurios pritaikymas leis visą procesą atlikti žymiai paprasčiau ir efektyviau. Pritaikant šį kodą, galima atsisakyti net ir duomenų perdavimo kaip failo ir tai tiesiog inkorporuoti į kompiliuojamą kodą.

Problemai spręsti pasirinkta naudotis C# programavimo kalbos funkcijomis, tiksliau, tai ***System.CodeDom.Compiler*** programinis paketas. Šio paketo esmė, tai automatizuotas programinio kodo kompiliavimas programos veikimo eigoje. Biblioteka leidžia programinį tekstą traktuoti kaip metodo įeities parametrą, o rezultate gražinamas sukompiliuotas kodas vis dar egzistuojantis atmintyje, todėl jį galima toliau apdoroti. Apdorojimas vyksta naudojant papildomą ***System.Reflection*** biblioteką, kuri leidžia C# programavimo kalbai dinamiškai perskaityti savo pačios kodą bei jį keisti. Ankstesniu etapu gautas sukompiliuotas kodas paimamas ir paleidžiamas, o rezultatai gali būti laisvai nukreipiami kur norima. Šios technologijos leidžia pagrindinio karkaso viduje greitai kurti, kompiliuoti, vykdyti programas bei atlaisvinti panaudotus resursus. Svarbiausias kodo segmentas, kuris tai atspindi, yra pateiktas 3-ame paveikslėlyje. Pirmu numeriu pažymėta kodo kompiliavimo dalis naudojasi ***System.CodeDom.Compiler*** bibliotekos pagalba. Čia taip pat matoma, kaip programiškai nurodomos visos papildomai reikalingos bibliotekos, pavyzdžiui ***Linq***, kuri naudojama generuojamo kodo viduje. Antruoju numeriu pažymėta ***System.Reflection*** dalis, paleidžianti programinį kodą atmintyje. Taip pat, šiuo momentu perduodami visi sukompiliuoto kodo darbui reikalingi parametrai. Gaunamas rezultatas yra nukreipiamas į standartinį išorinės sąsajos įrankį - konsolę.



3 pav. *Kodo kompiliavimo segmentas*

1.3. Genetinis programavimas

Šiame skyriuje detaliai aptariami pagrindiniai genetinio programavimo elementai. Pirmiausia pristatomas minimalus programinio kodo elementas, kuris yra kiekvieno kuriamo algoritmo pradžia. Toliau aptariami genetinio algoritmo pritaikymo ypatumai, svarbių genetinio algoritmo sąvokų apibrėžimai ir pritaikymai tiriamajai sričiai, programavimo sprendimai. Pristatoma genetinio algoritmo topologija bei aptariamas kiekvienas genetinis operatorius, kuris pritaikytas darbo eigoje. Pabaigoje aptariama genetinio tobulėjimo sąvoka bei pagrindiniai tokio proceso pritaikymo sprendimai ir panaudotos idėjos.

1.3.1. Pradinis vienetas

Genetinio algoritmo individas – tai C# kalba parašyta programa. Norint teisingai pritaikyti genetinio programavimo principus, verta apsibrėžti minimalų vienetą, kurio algoritmas nebandys modifikuoti. Tai reikalinga tam, kad būtu dirbama tik su darbui aktualia dalimi – algoritmo

paieška. Visiškai nesvarbu kaip sudaromos kitos esminės programos dalys: pradžia, pabaiga, bibliotekų aprašymai. Kaip šis pradinis vienetas atrodo pateikta 4-ame paveikslėlyje.

```
class Program
{
    static void Main(string[] args)
    {
        //algoritmo kodas
    }
}
```

4 pav. *Minimali C# programa*

Žinoma, reikia nepamiršti ir papildomų kintamųjų aprašymo. Kadangi bus vystomi algoritmai, jie turi dirbti su tam tikra duomenų seka. Tai reikalinga jų tinkamumo įrodymui. Įdomiausi tie algoritmai, kurie sugeba atlikti jiems paskirtą užduoti. Konkretus duomenų aprašymas gali priklausyti nuo algoritmo tipo, bet tai didelės įtakos pradiniam kodo segmentui nedaro. Programa testavimo duomenis gauna kaip programos paleidimo parametrą, kitaip vadinamu programos argumentu. Kadangi argumentai perduodami kaip tekstas, papildomai reikalinga juos konvertuoti į skaitinę išraišką. Kodėl duomenys bus generuojami atskirai, bus pristatoma sekančiame skyriuje, kalbant apie tinkamumo funkciją. Taip pat, svarbu gauti tam tikrą algoritmo darbo įvertinimą. Neužtektų tiesiog tikrinti ar algoritmas užduotį atliko visiškai gerai ar ne. Svarbūs ir tie atvejai, kada algoritmas veikia beveik gerai, tai parodo, kad jis turi naudingų savybių, kurios dar gali tobulėti. Todėl reikia skaičiuoti kokia dalis duomenų buvo teisingai apdorota, jei tai yra aktualu tiriamam uždaviniui. Apie tikslų algoritmo tinkamumo nustatymą kalbama 1.3.2.4 straipsnyje. Tinkamos duomenų dalies skaičiavimui reikia žinoti siektiną rezultatą. Rezultatas bus iš anksto sugeneruojamas ir įsimenamas. Tam, kad būtų sukurtas kuo paprastesnis algoritmas ir tyrimas vyktų sparčiai, kuriamo algoritmo programa iš anksto modifikuojama taip, kad rezultate būtų gražinama apdorota duomenų seka, kurią karkasas patikrina pats. Minimali programa su visais papildomais aprašymais pateikta 5-ame paveikslėlyje. Išlaikomas kodo paprastumas. Taip atrodys visos generuojamos algoritmų programos, skirsis tik pažymėta dalis, kurioje bus algoritmo kodas, kadangi tai aktualiausia šio tyrimo vieta. Verta pabrėžti, kad pirmąja eilute pateikiami skaitiniai duomenys, o paskutiniosios eilutės apdorotus duomenis palygina su norimu rezultatu, todėl šį segmentą galima pritaikyti įvairiems uždaviniams, išreikštinai nenurodant kokie veiksmai turi būti atliekami. Galima ieškoti rikiavimo, paieškos, skaičiavimo algoritmų, nes tai labai abstraktus kodo segmentas.

```
public class Program
{
    public static int Main(string[] args)
    {
        int[] data = args[0].Split(',')
                            .Select(int.Parse)
                            .ToArray();

        // algoritmo kodas

        var answer = args[1].Split(',')
            .Select(int.Parse)
            .ToArray();

        var matches = data.Where((t, i) => i < answer.Length && t == answer[i]).Count();

        return ConvertRange(0, answer.Length, 0, 100, matches);
    }

    public static int ConvertRange(
        int originalStart, int originalEnd,int newStart, int newEnd,
        int value)
    {
        var scale = (double)(newEnd - newStart) / (originalEnd - originalStart);
        return (int)(newStart + ((value - originalStart) * scale));
    }
}
```

5 pav. *Papildyta minimali C# programa*

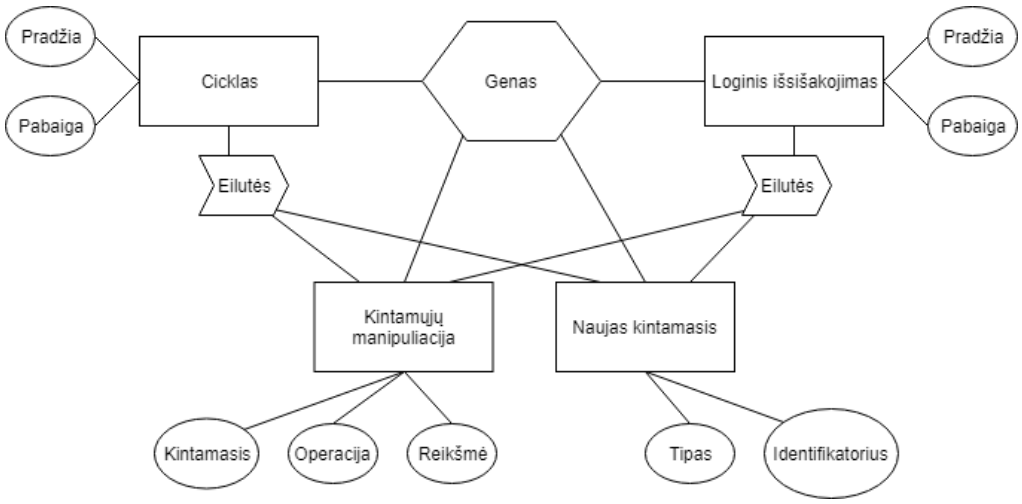
1.3.2. Genetiniai elementai

Svarbu tiksliai apsibrėžti visas genetinio algoritmo sąvokas ir kaip jos atitinka algoritmo pritaikymą genetiniam programavimui. Visu pirma, kaip jau buvo minėta ankstesniame skyriuje, individu laikoma viena programa. Programa aprašyta programos tekstu, o kai kurie jos elementai yra algoritmui neprieinami programos veikimo eigai užtikrinti.

1.3.2.1. Genas

Viena svarbiausių sąvokų – individo genas. Bendru atveju genas yra mažiausia individo dalis. Tuo atveju, kai individas yra programos tekstas, jo genas yra viena teksto eilutė. Tačiau tai pernelyg abstraktu, nes sudarkius eilučių seką, labai lengva gauti nekorektišką, gramatikos taisyklių neatitinkantį, programinį kodą. Autorių Abdelhalim Hiassat, Ali Diabat, Iyad Rahwan atliktame tyrime [HDR16] buvo susidurta su panašia problema. Šio tyrimo tikslas, sukurti modelį, kuris išsprendžia inventorizacijos maršrutų sudarymą su greitai gendančiais produktais. Tokia problema buvo įvertinta kaip NP-Sunkaus tipo uždavinys. Autoriai tyrimo metu pasirinko

genetinio algoritmo modifikaciją ir ją pritaikė kuriamam modeliui. Rezultate jiems pavyko per patenkinamą laiką gauti beveik idealius sprendinius. Nors šis tyrimas nėra glaudžiai susijęs su algoritmų paieška, jame iškeltos idėjos yra naudingos. Tyrimo metu autoriai susidūrė su individo chromosomų aprašymo problema. Daugumoje literatūros šaltinių nurodoma, kad jei chromosomų ilgiai yra vienodi, tai leidžia labai lengvai atlikti jų kryžminimo operacijas. Tačiau, šiuo atveju, autoriams prireikė kintančio ilgio chromosomų. Pagrindinė kilusi idėja – suskaidyti chromosomas į logiškus segmentus, kurių svarbumas nėra vienodas. Kadangi šio darbo atveju programos algoritmo kodas yra kintantis, t.y. jo gali padaugėti ar sumažėti, toks požiūris į chromosomų sudarymą tinka. Chromosoma laikysime ne vieną programinio teksto eilutę, o loginį vienetą. Tai naujo kintamojo sukūrimas, kintamųjų manipuliacija, loginis išsišakojimas arba ciklas. Loginio išsišakojimo ar ciklo atveju, chromosoma savyje saugo ne tik kodo eilutę, kurioje operacija pradedama, bet ir pabaigos eilutę bei visas vidines operacijų eilutes. Šių esybių supaprastinta ontologija pateikta 6-ame paveikslėlyje.



6 pav. *Geno ontologija*

Kvadratėliu žymimos atskiros, svarbiausios klasės. Genas gali įgauti vieną iš keturių skirtingų reikšmių. Apskritimais žymimos klasių savybės, kurios yra reikalingos kitų operacijų darbo metu. Taip pat, ciklo arba loginio išsišakojimo eilutės savybė yra sąrašo tipo, o kiekviena šio sąrašo eilutė gali būti vieno iš dviejų, anksčiau minėtų tipų – kintamųjų manipuliacija arba naujo kintamojo sukūrimas. Pradžia ir pabaiga žymi kodo eilutes, reikalingas programinei struktūrai užtikrinti, jos darbo metu nėra manipuliuojamos, išskyrus parametrus, esančius jų viduje, pavyzdžiui ciklo skaitliukas. Ciklo ***for*** kodo pradžia atrodo taip:

for (<u>inicializavimas</u> ; <u>sąlyga</u> ; <u>iteratorius</u>) {

čia **inicializavimas**, **sąlyga** ir **iteratorius** žymi neterminalinius simbolius, kurie yra papildomai generuojami. Kode tai atitinka atskiras klases. Visi likę simboliai yra privalomi korektiškam ciklo aprašymui, todėl jie nekeičiami. Ciklo pabaiga yra paprasta, tai simbolis **}** kodo bloko uždarymui. Kaip vyksta tokių genų kryžminimas aptarta 1.3.3.2 straipsnyje. Taigi, Abdelhalim Hiassat, Ali Diabat, Iyad Rahwan autorių atlikto tyrimo analizė suteikė įžvalgos į genetinio individo chromosomų aprašymo problemą.

1.3.2.2. Populiacija

Genetinio algoritmo populiacija – individų rinkinys. Populiacijos dydis dažniausiai priklauso nuo to, kiek resursų turi kompiuterinė sistema, kurioje algoritmas veikia. Autoriai X. H. Shi, L. M. Wan, H. P. Lee, X. W. Yang, L. M. Wang, Y. C. Liang, tyrime[SWL+03] pateikia įdomių idėjų, susijusių su populiacijos valdymu. Minėto straipsnio tikslas – pagerinti genetinio algoritmo darbą, atsižvelgiant į natūralų populiacijos dydžio kitimą. Tai pat, šis tyrimas pristato hibridinį PSO-GA algoritmą, kuris apjungia dalelių spiečiaus ir genetinį algoritmus. Autoriai kiekvienam populiacijos individui suteikia mirties tikimybę (angl. *dying probability*), priklausančią nuo kartos, kurią individas pasiekia. Genetinio algoritmo darbo metu individui pasiekus tam tikrą genetinės kartos skaičių, jo mirties tikimybė vis auga. Tai pat, kryžminimo operacijos metu, naujų individų pridėjimas į kartą nėra griežtai ribojamas. Tai reiškia, kad populiacija gali kisti – išaugti arba sumažėti. Sekami du slenksčiai (angl. *threshold*), minimalus individų skaičius populiacijoje ir maksimalus. Jeigu populiacija viršija maksimalų individų skaičių, autoriai siūlo sukurti karo (angl. *war*) operaciją, kuri tam tikrą dalį individų pašalintų, atliekant jų tinkamumo palyginimus. Tyrimo pabaigoje ši genetinio algoritmo modifikacija ir naujas PSO-GA algoritmas buvo patikrintas, o gauti rezultatai parodė, kad tokia modifikacija veikia efektyviau už tradicinį genetinį algoritmą. Ši populiacijos stebėjimo ir kontrolės modifikacija pritaikyta ir tiriamam algoritmų paieškos uždaviniui. Stebimi du minėtieji populiacijos skaičiai: apsibrėžtas maksimalus ir apsibrėžtas minimalus. Maksimalaus skaičiaus viršijimo atveju naudojama minėto straipsnio autorių siūloma karo operacija, o minimalaus skaičiaus atveju generuojami papildomi, atsitiktiniai populiacijos individai. Minėti slenksčiai yra optimizuojami parametrai, priklausantys nuo populiacijos dydžio. Karo operatorius savyje vykdo turnyro tipo atranką. Atrankos kodas pateiktas 7-ame paveikslėlyje. Šiuo atveju, naudojamas žymiai paprastesnis atrankos algoritmas, negu atrankos operatoriaus atveju, apie kurį detaliai kalbama 1.3.3.1 straipsnyje. Toks sprendimas priimtas dėl to, kad pasirinkimas nėra kritiškai svarbus, sumažinamas kodo sudėtingumas, taip išlošiant skaičiavimų greičio, nes ši operacija

vykdoma pakankamai dažnai. Resursų taupymas karkaso darbo metu, kur tai galima padaryti, yra vienas iš svarbiausių prioritetų, nes tai leis tirti didesnę individų aibę.

```
while (population.Individuals.Count > population.MaxThreshold)
{
    TournamentIndividuals.Clear();

    while (TournamentIndividuals.Count < TournamentSize)
    {
        TournamentIndividuals.Add(population.Individuals[Rnd.Next(population.Individuals.Count)]);
    }

    var strongestIndividual = new Individual(0); //Fitness = 0
    foreach (var individual in TournamentIndividuals)
    {
        if (individual.Fitness >= strongestIndividual.Fitness)
        {
            var previousStrongest = strongestIndividual;
            strongestIndividual = individual;
            population.Individuals.Remove(previousStrongest);
        }
        else
        {
            population.Individuals.Remove(individual);
        }
    }
}
```

7 pav. *Turnyro atrankos kodas*

Operacija vykdoma tol, kol populiacijos skaičius pasiekia slenkstį. Pirmiausia atsitiktinai pasirenkamas tam tikras skaičius individų. Tai vėl gi yra optimizuojamas parametras. Tada, iš atrinktų individų populiacijoje išlieka tik stipriausias, visi kiti yra pašalinami. Atnaujinta populiacija grąžinama tolesniam karkaso darbui.

1.3.2.3. Karta

Algoritmo karta apibrėžiama kaip iteracija, kurioje apdorojamas kiekvienas populiacijos individas, suskaičiuojamas jo tinkamumas ir pritaikomi visi genetiniai operatoriai. Tokių veiksmų pabaigoje algoritmas pareina į sekančią kartą. Individai, nepašalinti genetinių operatorių veikimo metu, yra laikomi senesni, išgyvenę bent kelias kartas. Pereinant iš vienos kartos į kitą, yra atliekami šie veiksmai:

- Genetinių operatorių pritaikymas.
- Individų tinkamumo skaičiavimas.
- Individų amžiaus padidinimas.
- Geriausio tos kartos individo suradimas.
- Vidutinio individo tinkamumo skaičiavimas.
- Pabaigos kriterijaus tikrinimas.

1.3.2.4. Tinkamumo funkcija

Tinkamumo funkcija (angl. *Fitness function*) apibrėžia procesą, kuriuo individų tinkamumui priskiriama skaitinė vertė. Algoritmų paieškos atveju individas yra programos tekstas, todėl pakankamai sunku patikrinti jo tinkamumo lygi. Kamaljit Kaur, Kirti Minhas, Neha Mehan, ir Namita Kakkar straipsnyje [KMM+09] aptaria kelias kodo įvertinimo metrikas. Straipsnio tikslas – įvertinti statinio ir dinaminio sudėtingumo metrikas. Straipsnyje autoriai aprašo du įvertinimo būdus – Halstead‘o metrika ir Mc Cabe‘o metrika. Abi metrikos naudojamos programinio kodo sudėtingumui vertinti. Atliekant palyginimą Halstead‘o metrika sulaukia kritikos. Rezultate autoriai teigia, kad kodo vertinimų metrikos padeda nuspręsti programos sudėtingumo lygį, o pritaikymas konkrečiam uždaviniui ir atitinkamų kriterijų pasirinkimas, kas garantuos geresnę kodo kokybę, priklauso nuo tos užduoties vykdytojo. Šio straipsnio metu siūlomos metrikos yra gana įdomios ir leidžia gauti skaitinį programos sudėtingumo įvertinimą. Nors Mc Cabe‘o siūlyta metrika nesulaukė autorių priekaištų, jos pritaikymas algoritmų paieškos uždaviniui yra netinkamas, nes ši metrika remiasi programinio kodo vertimu į srauto grafą, kuriame skaičiuojamos grafo savybės. Toks procesas reikalautų per daug programinių resursų, atsižvelgiant į tai, kad bus tiriamas labai didelis skaičius individų. Kitu atveju, Halstead‘o metrika, nors ir turi tam tikrų trūkumų, šiam uždaviniui tinka. Ją pritaikome individų tinkamumo įvertinime. Metrika seka keturis skaičius:

1. n_1 – skirtingų operatorių skaičius.
2. n_2 – skirtingų operandų skaičius.
3. N_1 – visų operatorių skaičius.
4. N_2 – visų operandų skaičius.

Pagal juos skaičiuojami tokie atributai:

1) Žodynas (n) = $n_1 + n_2$.
2) Ilgis (N) = $N_1 + N_2$.
3) Apimtis (V) = $N * \log_2 n$
4) Potenciali apimtis (V*) = $(2 + n_2) * \log_2(2 + n_2)$
5) Programos lygis (L) = $\frac{V^*}{V}$

Programą įvertina programos lygis. Kuo lygis arčiau vieneto, tuo programos kodas laikomas korektiškesniu. Pastebėsime, kad tik labai specifinis programinis kodas sugeba gauti idealų metrikos įvertinimą. Geriausias įvertinimas nėra kritiškai svarbus, nes metrika skirta programų lyginimui tarpusavyje. Didžiausias metrikos trūkumas atsispindi tame, kad išsigimusiais atvejais, pavyzdžiui, kai egzistuoja tik operandai, be operatorių, metrika įgauna reikšmes žymiai didesnes

už vienetą, todėl to korekcijai, visos reikšmės įgaunančios reikšmes didesnes nei 2 yra traktuojamos kaip nulinės, nes programos su tokiu įvertinimu tikrai nėra tinkamos. Kaip buvo minėta, reikšmė turi artėti link vieneto, todėl programos lygio įvertis turi rėžius nuo **0** iki **2**. Galutinis įvertinimas yra konvertuojamas į skalę nuo **0** iki **1**, pagal tai kiek gautas įvertinimas nutolęs nuo vieneto. Pavyzdžiui, metrikos įverčių **0,5** bei **1,5** atvejais, po konvertavimo gaunamas rezultatas yra vienodas, **0,5**, nes jie abu vienodai nutolę nuo trokštamo rezultato. Verta paminėti, kad eksperimentų metu, metrikos įvertis beveik niekada neviršijo vieneto. Šitai leidžia labai lengvai įvertinti visų individų tinkamumą, atsižvelgiant ne tik į maksimalų ar minimalų eilučių skaičių, bet ir į sudėtingesnę, programos lygio metriką. Kadangi skaičiuojami tik programos teksto ypatumai, šis procesas yra spartus.

Tinkamumo funkcija skaičiuoja individo pranašumą ne tik stebint kodo parametrus, bet atsižvelgia ir į tai, kad algoritmas atlieka jam paskirtą darbą. Žinoma, įdomūs tik tie algoritmai, kurie veikia gerai, bet didelio skaičiaus korektiškų algoritmų karkaso darbo pradžioje tikėtis negalima. Kaip tinkamumą vertinti tada? Idėja – įvertinti kokią duomenų dalį jie apskaičiavo gerai, o kokią ne, jei uždavinyje tai gali būti aktualu. Tai galima padaryti iš anksto sugeneruojant teisingą testo atsakymą. Karkaso darbo metu, algoritmui sugeneruojama ir pateikiama duomenų seka bei užduotis ją modifikuoti. Tuo pačiu metu, karkasas atmintyje saugo šios užduoties atsakymą. Vystomam algoritmui baigus darbą, karkasas patikrina gauto rezultato ir teisingo atsakymo atitikimo lygį ir tam suteikia skaitinį įvertinimą. Skaičiuojamas procentinis atitikimas, kuris konvertuojamas į galutinį įvertinimą skalėje nuo **0** iki **1**. Pavyzdžiui, jeigu algoritmas sugeba gauti rezultatą, kurio elementų išsidėstymas bei reikšmės atitinka **75%** elementų teisingame atsakyme, įvertinimas būtų lygus **0,75**. Galiausiai, turint du įverčius, vieną gautą pagal kodo metrikas, kuris yra skalėje nuo **0** iki **1**, o kitą pagal atliktos užduoties įvertinimą, kurio skalė taip pat išsidėsčiusi nuo **0** iki **1**, juos susumuojame į vieną. Galutinė formulė:

Tinkamumas = K(L) + D.

Čia **K(L)** – konvertuotas programos lygio įvertinimas rėžyje [0;1], o **D –** atsakymo atitikimo įvertinimas rėžyje [0;1]. Maksimalus tinkamumo funkcijos įvertinimas yra 2, o minimalus lygus nuliui. Šiuo tinkamumo funkcijos skaičiavimo atveju išvengiama daugiakriterinės užduoties problemos, nes karkaso tikslas gauti algoritmą, kuris veikia idealiai, t.y. atliktos užduoties rezultatas turi visiškai atitikti teisingą atsakymą. Ši bendra įverčių suma padeda greitai į priekį iškelti algoritmus, kurie veikia korektiškai, o tolimesnis jų palyginimas yra grįstas kodo metrikomis. Karkaso darbo metu vystomas kodo metrikas tenkinantis algoritmas, kuris gauna norimą rezultatą.

Verta pabrėžti, kad matuojamas individo programinio kodo veikimo laikas. Tai reikalinga tam, kad būtų užkirstas kelias begaliniams arba labai ilgiems ciklams. Kadangi programinis kodas

kuriamas su atsitiktinumu, įmanoma sukurti tokį ciklą, kuris neturi pabaigos. Tokiu atveju šio individo tinkamumo funkcijos darbas niekada nesibaigtų, nes nebūtų gaunamas rezultatas. Ši problema išspręsta matuojant uždelstą algoritmo laiką ir jo darbą stabdant peržengus maksimalų rėžį. Nuspręsta stabdyti visus individus, kurių programinis kodas užtrunka ilgiau nei 1000 milisekundžių. Antrame skyriuje aptarti eksperimentai vystė ganėtinai trumpus algoritmus, todėl tokio rėžio peržengimas reikštu, kad su labai didele tikimybe, sukurtas kodas nėra optimalus. Žinoma šis rėžis privalo būti apibrėžtas priklausomai nuo užduoties. Tokiais atvejais, kai individo sukurtas programinis kodas peržengia rėžį ir kodas yra stabdomas, individo tinkamumo įvertinimo antroji dalis, **D,** įgauna nulinę reikšme. Toks sprendimas pasirinktas tam, kad šių individų išlikimas populiacijoje būtų kuo mažesnis, bet ne visiškai nulinis. Net ir blogi individai gali turėti tam tikrus gerus genus, todėl jų visiškai sunaikinti negalima. Taigi, yra stebimas individų programinio kodo veikimo laikas ir atliekami korekciniai veiksmai, atvejais kai įsikišimas yra privalomas sukurto karkaso darbo eigos išsaugojimui.

Atsižvelgiant į išdėstytą teoriją ir aptartus straipsnius, tinkamumo funkcijos kodas susideda iš trijų esminių detalių. Tai atspindinti kodo dalis pateikta 8-ame paveikslėlyje.

```
var operandList = operandRegex.Matches(individual.RawCode).Cast<Match>()
    .Select(m => m.Value)
    .ToArray(); ;

var operatorList = operatorRegex.Matches(individual.RawCode).Cast<Match>()
    .Select(m => m.Value.Trim())
    .Where(m => m != string.Empty)
    .ToArray(); ;

var uniqueOperators = operatorList
    .GroupBy(w => w)
    .Select(g => g.Key)
    .Count(g => AllOperators.Contains(g));
var uniqueOperands = operandList
    .GroupBy(w => w)
    .Select(g => g.Key)
    .Count(g => !AllKeywords.Contains(g));
var totalOperators = operatorList
    .Count(g => AllOperators.Contains(g));
var totalOperands = operandList
    .Count(g => !AllKeywords.Contains(g));

var vocabulary = uniqueOperators + uniqueOperands;
var length = totalOperators + totalOperands;
var volume = length * Math.Log(vocabulary, 2);
var potentialVolume = (2 + uniqueOperands) * Math.Log(2 + uniqueOperands, 2);
var programLevel = Convert(potentialVolume / volume);

var result = compiler.Compile(individual.RawCode);
var compilerResult = result <= 1 ? result : 0;
var finalScore = programLevel + compilerResult;

individual.Fitness = finalScore;
```

8 pav. *Tinkamumo funkcijos kodas*

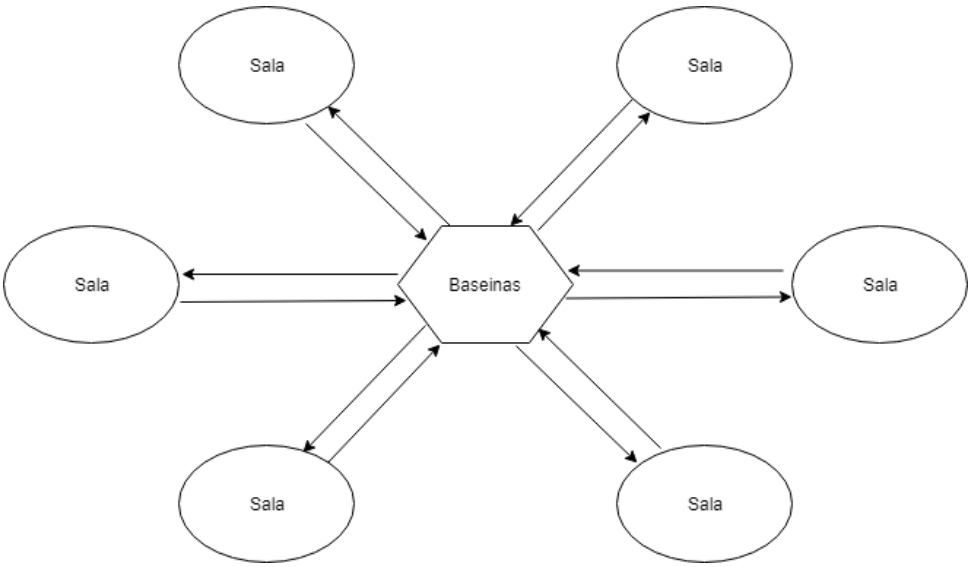
Esminės kodo dalys atskirtos raudonu brūkšniu. Pirmojoje, iš individo kodo, pateikto paprastu tekstiniu pavidalu, ***Regex*** bibliotekų pagalba išgaunami visi operandai bei operatoriai. Toks apdorojimo būdas pasirinktas dėl didesnės spartos. Kitu atveju kiekvienas individo kodas būtų kaip didelė klasių esybė, kurią karkasas saugotų kompiuterio atmintyje. Tai leistų greičiau gauti reikalingų elementų skaičių, bet papildoma resursų kaina struktūros palaikymui yra negatyviai veikiantis faktorius, kurio galima išvengti. Antrojoje dalyje, pagal anksčiau išdėstytas formules, suskaičiuojamos visos metrikos ir išvedamas galutinis programos lygis. Šie skaičiavimai yra pakankamai paprasti, todėl ši kodo dalis veikia greičiausiai. Trečiojoje, sudėtingiausioje, dalyje iškviečiamas automatinis kodo kompiliatorius, kuriame individo kodas yra įvykdomas. Apie kompiliatoriaus veikimo principus buvo kalbėta 1.2.3 skirsnyje, todėl detalės čia praleidžiamos. Patikrinamas gautas rezultatas ir pagal jį suskaičiuojamas galutinis tinkamumo lygis, atnaujinama individo tinkamumo reikšmė. Tokiu būdu apdorojamas vienas populiacijos individas.

1.3.2.5. Salos

Papildoma genetinio algoritmo modifikacija vyksta pritaikant salų (angl. *island*) tipo topologiją. Autorius Jiri Jaros straipsnyje [Jar12] pristato tokio taikymo ypatumus. Tyrimo tikslas – išspręsti keliaujančio pirklio uždavinį, panaudojant genetinį algoritmą ir apjungiant keletą grafinių procesorių skaičiavimams atlikti. Autorius pasiūlo, jo manymu, unikalų genetinio algoritmo variantą, kuris darbą išskaido keliems grafiniams procesoriams. Straipsnyje detaliai aprašomos visos įgyvendinimo detalės, kompiuterio atminties valdymas, genetinio algoritmo operatorių pritaikymas. Pagrindinis dėmesys skiriamas skaičiavimų greičiui, darbu lygiagretinimui ir užduočių skaidymui. Rezultate gautas žymus sprendinių kokybės šuolius, kai naudojamas pakankamai didelis salų skaičius, o salos populiacija taip pat yra pakankamai plati. Tokie rezultatai nenustebina, nes maža populiacija neturi pakankamos genetinės įvairovės. Svarbiausias elementas, kuriuo buvo pasinaudota, tai skaičiavimų spartinimas. Jiri Jaros tyrimas, nors ir nėra susijęs su genetiniu programavimu, tačiau dirba labai panašiomis sąlygomis. Individų tinkamumo apskaičiavimas paskirtas grafiniam procesoriui, o visas likęs genetinio algoritmo darbas atliekamas su centriniu procesoriumi. Tokiu būdu, apjungus kelis kompiuterius, būtų gaunamas pakankamai geras darbo išlygiagretinimas. Spartinimo tikslas – sumažinti laiką, reikalingą vieno eksperimento įvertinimui. Tai ypač padeda atlikti parametrų optimizavimą, apie kurį plačiau kalbama 2.4 poskyryje.

Salos tipo genetinis algoritmas suteikia galimybę pridėti papildomų funkcijų. Viena iš jų – migracijos operatorius. Migracijos leidžia tam tikriems individas, tam tikru laiko momentu, peršokti iš vienos genetinio algoritmo salos į kitą, taip dalinantis gerais genais. Detaliau migracija aprašoma 1.3.3.5 straipsnyje.

Kiekvienoje saloje egzistuoja individuali, nuo kitų nepriklausanti genetinio algoritmo versija. Kaip jau buvo minėta tirtame literatūros šaltinyje, didesnis salų skaičius lydi link geresnio rezultato, todėl norima jų turėti kuo daugiau. Tai vėl kelia tam tikrų darbo spartos klausimų. Pirmiausia, vykdant migracijos operatoriaus darbą, paprasčiausiu atveju, kiekviena sala turėtų žinoti apie visas likusias, kad galėtų su jomis keistis individais. Kuo salų skaičius didesnis, tuo daugiau turi atsiminti kiekviena individuali sala, o tai sparčiai sunaudoja kompiuterio resursus. Problemai spręsti, įvedama papildoma baseino (angl. *pool*) sąvoka. Baseinas tai dirbtinis, salą primenantis elementas, kurio paskirtis yra kaupti migruojančius individus ir juos paskirstyti ten, kur reikia. Baseino vizualizacija pateikta 9-ame paveikslėlyje.



9 pav. *Baseino veikimo principas*

Po modifikacijos, kiekviena genetinio algoritmo sala žino tik apie baseiną, visos kitos salos tampa nematomos. Migracijos metu individai keliauja iš salos į baseiną. Baseinas, priklausomai nuo to kiek individų yra sukaupta, salai bando grąžinti tą patį skaičių naujų individų. Tokiu būdu, modifikacijos pagalba, žymiai sutaupomas naudojamų resursų kiekis. Baseinas savyje saugo individus bei salos, iš kurios jie kilo, identifikatorius. Tai reikalinga tam, kad individai negrįžtų į savo gimtąją salą. Egzistuoja tik dvi nepertraukiamos operacijos, viena individams priimti, o kita individams išimti, todėl baseino darbas yra labai spartus. Kadangi migracijos nevyksta dažnai, o operacijos paprastos ir greitos, užtenka jas apsaugoti C# programavimo kalbos ***Lock*** (liet. *Spyna*)

galimybėmis. Šis kalbos konstruktas tam tikrą kodo segmentą užrakina nuo pertraukimų, imituojant atominę operaciją. Šie specifiniai metodai pateikti 10-ame paveikslėlyje.

```
public void AddToPool(int callerID, Individual individual)
{
    lock (listLock)
    {
        _individualDictionary.Add(callerID, individual);
    }
}

public Individual TakeFromPool(int callerID)
{
    lock (listLock)
    {
        var values = _individualDictionary
            .Where(_ => _.Key != callerID)
            .Select(_ => _.Value)
            .ToList();

        return values.Count > 0 ? values[Rnd.Next(values.Count)] : null;
    }
}
```

10 pav. *Baseino metodai*

Kaip matoma paveikslėlyje, operacijos yra apsaugotos nuo nekorektiško pertraukimo kelių gijų atveju. Kai sala kreipiasi į baseiną ir iškviečia individo paėmimo metodą *TakeFromPool*, pirmiausia gaunamas visų individų sąrašas, kurie kilę iš kitų salų. Toliau iš jų atrenkamas vienas atsitiktinis arba gražinama tuščia reikšmė tuo atveju, kai baseinas yra tuščias arba nėra individų, galinčių pereiti į salą, kuri dabar bendrauja su baseinu.

1.3.3. Genetiniai operatoriai

Šiame skyriuje detaliai aprašomi visi genetinio algoritmo operatoriai, pristatant jų įgyvendinimo sprendimus bei panaudojant literatūroje sutiktus pritaikymo variantus ir idėjas.

1.3.3.1. Atranka

Genetinio algoritmo atrankos operatorius yra vienas iš įtakingiausių genetinių operatorių. Atrankos metu algoritmas, pagal individų įvertinimą, atrenka dalį tinkamiausių individų, kurie bus panaudoti, kuriant sekančią populiaciją. Dažniausiai, ši individų dalis tiesiogiai perkeliama į naują populiaciją, kuri pilnai užpildoma kitų operatorių darbo metu. Nesunku pastebėti, kad atrankos rezultatai lemia visų likusių operatorių pritaikymo kokybę. Atrankos procesas gali būti atliekamas

įvairiais būdais. Logiška manyti, kad atrinkus tik pačius geriausius individus, bendras visos populiacijos tinkamumo lygis sparčiai išaugtų. Deja, toks atrankos būdas negarantuoja gerų rezultatų dėl įvairių priežasčių. Viena iš jų, tai individų supanašėjimas. Jeigu atrenkami tik tie patys geri individai, jie dalyvaus kryžminime ir perduos savo genetinę informaciją, kuriant gana identiškus individus. Po didelio genetinių kartų skaičiaus gaunama populiacija, kurioje yra tik tokių individų variantai, nes jų tinkamumo lygis buvo geriausias. Tai lydi prie genetinės įvairovės problemos. Jeigu populiacijoje nėra genų, kurie leistų individams toliau tobulėti, genetinis algoritmas užsistovės lokaliame optimume. Vienintelė galimybė iš tokios situacijos ištrūkti – kelių sėkmingų mutacijų pritaikymas, ko tikimybė yra labai maža. Tuo remtis tikrame tyrime nėra prasmės. Taigi, privalu užtikrinti, kad į sekančią kartą pateks ne tik dalis labai gerų individų, bet, su tam tikra tikimybe bus atrinkti ir keli atsitiktiniai individai iš visos buvusios populiacijos. Įvedamas atsitiktinumas (angl. *randomization*), kas prisideda prie genetinio algoritmo sudėtingumo. Vėl gi, tokio tipo atrankai atlikti egzistuoja keletas praktikoje patikrintų būdų. Vienas iš populiariausių ir dažniausiai naudojamų – turnyro atranka (angl. *tournament selection*). Šios atrankos metu vyksta mažos individų kovos. Visiškai atsitiktinai parenkami keli populiacijos individai, iš kurių geriausias pereina į sekančią kartą. Tokiu būdu, atsitiktinumo dėka, individų grupelės gali būti net ir labai žemo tinkamumo, kas leis vienam iš tokių individų išgyventi iki kitos iteracijos. Procesas imituoja turnyrą, kurio nugalėtojas yra tinkamiausias individas. Autoriai Noraini Mohd Razali ir John Geraghty savo straipsnyje [RG11] aprašo palyginimą tarp turnyro ir dviejų atskirų ruletės tipo atrankos būdų. Jų tyrimo tikslas – patikrinti genetinio algoritmo darbą, sprendžiant keliaujančio pirklio uždavinį, kai pritaikomi skirtingi atrankos būdai. Tyrime daug dėmesio skiriama šių metodų analizei ir korektiškam pritaikymui. Rezultate gauta, kad turnyro tipo atranka žymiai greičiau gauna gerus rezultatus prie pakankamai mažo populiacijos dydžio. Autoriai teigia, kad didėjant populiacijos dydžiui, šio tipo atranka lydi prie lokalaus, o ne globalaus optimumo radimo. Remiantis bendra praktika ir straipsnio rezultatais, galima teigti, kad turnyro tipo atranka turi gerų savybių.

Turnyro atranka, nors ir plačiai naudojama, turi ir tam tikrų trūkumų, kuriuos galima išspręsti. Vladimir Filipovic tyrime pavadinimu „Fine-Grained Tournament Selection Operator in Genetic Algorithms“[Fil12] pristato šios atrankos modifikaciją. Autorius pateikia motyvaciją, patobulinto algoritmo pseudokodą, teorišką jo pagrindimą ir realaus pritaikymo rezultatus. Pagrindinė šio geresnio algoritmo argumentacija yra ta, kad tradicinis turnyras remiasi konkrečiu turnyro dydžiu, kuris nekinta viso proceso metu. Išankstinės konvergencijos problemai spręsti, Vladimir Filipovic pritaiko tam tikrą turnyro dydžio atsitiktinumą. Sekamas vidutinis turnyro dydis, žymimas F_{tour}. Autorius išbandė įvairius variantus, bet apsistojo ties turnyrais, kurių dydis daugiausia skiriasi tik per vienetą. Visas pagerintos atrankos pseudokodas, kuris buvo pateiktas

straipsnyje, yra matomas 11-ame paveikslėlyje. Verta pabrėžti, kad ši modifikacija yra gana paprasta ir didelių pridėtinių išlaidų viso algoritmo darbo metu nesukelia.

```
Input: Population a (size of array a is n), desired average tournament size F_tour, F_tour ∈ R
Output: Population after selection a' (size of array a' is n)
Fine_Grained_Tournament_Selection:
begin
   F_tour^- := trunc( F_tour);
   F_tour^+ := trunc( F_tour) + 1;
   n^- := trunc ( n * (F_tour^+ - F_tour) );
   n^+ := n - n^-;
   /* tournaments with size F_tour^- */
   for i := 1 to n^- do
      a[i]' := best fitted among F_tour^- individuals randomly selected from population a;
   /* tournaments with size F_tour^+ */
   for i := n^- + 1 to n do
      a[i]' := best fitted among F_tour^+ individuals randomly selected from population a;
   return a'
end
```

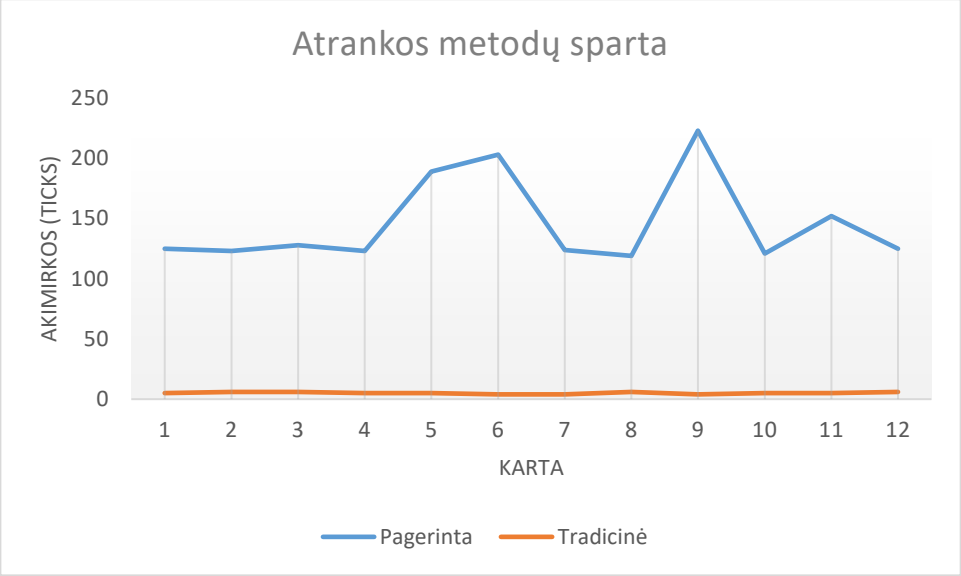
11 pav. *Pagerintos turnyro tipo atrankos pseudokodas*

Svarbu atkreipti dėmesį į tai, kad algoritmo darbas priklauso nuo pasirinkto vidutinio turnyro dydžio F_{tour}. Tais atvejais, kai pasirinktas dydis yra sveikas skaičius, šis algoritmas vyks analogiškai paprastai turnyro atrankai. Į tai atsižvelgta atliekant genetinio algoritmo parametrų optimizaciją. Tam, kad būtų naudojama tik patobulinta atranka, vidutinio turnyro dydžio įvertis gali būti tik skaičius su trupmenine dalimi. Apie tai plačiau kalbama 2.4 poskyryje. Autorius matematiškai įrodo kelias svarbias pagerintos atrankos algoritmo savybes, kas leidžia su didesniu pasitikėjimu jį taikyti kuriamoje sistemoje. Gauti tyrimo rezultatai taip pat parodė, kad patobulintos turnyrinės atrankos dėka buvo gauti geri rezultatai tiriant NP-Sunkus klasės uždavinius. Autorius netgi atliko tam tikrą vidutinio turnyro dydžio tyrimą ir tirtuose uždaviniuose sugebėjo rasti optimumo reikšmę, $F_{tour} = 5{,}6$. Nors algoritmų paieška yra visiškai kitoks uždavinys, ši gauta reikšmė panaudota kaip pradinis parametras, kuris toliau optimizuojamas. Taigi, genetinio algoritmo atranka atliekama panaudojant ypatingą turnyro tipo atrankos modifikaciją.

```
var n = population.Individuals.Count;

var FtourMinus = Math.Truncate(_desiredAverageTournamentSize);
var FtourPlus = Math.Truncate(_desiredAverageTournamentSize) + 1;

var nMinus = (int)Math.Truncate(n * (FtourPlus - _desiredAverageTournamentSize));

for (int i = 0; i < nMinus; i++)
{
    newPopulation.Individuals[i] = Tournament(population, FtourMinus);
}

for (int i = nMinus; i < n-1; i++)
{
    newPopulation.Individuals[i] = Tournament(population, FtourPlus);
}

return newPopulation;
```

12 pav. *Atrankos algoritmo kodas*

Atrankos operatoriaus kodas yra gana paprastas. Jis pateiktas 12-ame paveikslėlyje. Kaip ir buvo išdėstyta minėtame straipsnyje, kiekvienas naujos populiacijos individas, pereinant iš vienos kartos į kitą, yra atrenkamas kuriant turnyro tipo kovą. Yra sekami du skirtingi turnyrų dydžiai, kurie skiriasi labai nežymiai. Pirma populiacijos dalis sudaroma remiantis vienu turnyro dydžiu, o likusi kitu. Tai, programiniame kode, atitinka dvejų ***for*** ciklų naudojimą. Turnyro metu atsitiktinai atrenkamas reikalingas skaičius individų, iš kurių geriausias tampa naujos populiacijos individu. Šis kodas yra trivialus, todėl jis detaliau netiriamas. Darbo pabaigoje, C# kalbos automatizuotomis priemonėmis, senoji populiacija yra įsimenama tik iki paskutinio genetinio operatoriaus pritaikymo, o į sekančią algoritmo kartą pereina naujoji individų populiacija. Labiausiai algoritmo spartą įtakoja naujų turnyrų kūrimas, kas sulėtina atrankos operatoriaus darbą. Jei būtų naudojama paprastesnė atrankos versija, tada būtų gaunamas spartos padidėjimas, bet remiantis literatūra ir ankstesne patirtimi su genetiniu algoritmu, labai tikėtina, kad dėl to taip pat sumažės populiacijos vidutinio tinkamumo įvertinimo augimas. Tam pagrįsti atliktas trumpas eksperimentas. Eksperimento rezultatas pateiktas 13-ame paveikslėlyje. Abscisių ašyje žymimos algoritmo genetinės kartos, o ordinačių ašyje pateikiamas procesoriaus akimirkų (angl. *ticks*) skaičius. Karkasas paleidžiamas su paprastais testavimo duomenimis ir stebimos akimirkos, kurias užtrunka kiekvienas atrankos algoritmas. Kaip matoma iš rezultatų, tradicinė atranka yra žymiai greitesnė. Prireikus algoritmo spartos, atrankos metodo pakeitimas į paprastesnį galėtų būti vienas iš esminių žingsnių. Šiuo atveju, kadangi tai pirmasis genetinio algoritmo operatorius, jis labiausiai įtakoja bendrą algoritmo rezultatą, todėl verta paaukoti spartą, kad būtų gaunamas geresnis individų sąrašas.



13 pav. *Atrankos metodų spartos palyginimas*

1.3.3.2. Kryžminimas

Kryžminimo operacija taip pat yra labai svarbi genetinio algoritmo darbo dalis. Būtent čia vyksta naujų individų evoliucija, genų perdavimas iš tėvinių individų į vaikinius. Detalias kryžminimo idėjas ir bendrojo proceso aprašymą galima rasti viešai prieinamoje literatūroje. Kaip buvo minėta 1.3.2.1 straipsnyje, šio tyrimo atveju individo chromosomos nebus lygios. Tai kryžminimą skaidys į du atskirus atvejus. Pirmiausia, kai kryžminamos chromosomos yra vienetinio tipo eilutės, t.y. priskyrimo sakiniai, naujų kintamųjų kūrimas. Kadangi jos nesiremia kontekstine informacija, tokių eilučių kryžminimą galima apibrėžti tiesiog mainais. Antrasis variantas, kai kryžminamos chromosomos turinčius ciklo ar loginio išsišakojimo sakinius. Logiškas sprendimas – leisti kryžminti tik to paties tipo chromosomas, apkeičiant jų vidinius sakinius. Tačiau iš karto galima pastebėti, kad tai yra pernelyg griežtas reikalavimas, kryžminimas pavyks tik kai abu individai savyje turės tokio tipo chromosomas. Nevyks unikalių genų plytimas. Tai išsprendžiama šiam procesui suteikiant atsitiktinumą (angl. *randomization*). Su tam tikra, nedidele tikimybe kryžminimo procese leidžiama apkeisti vienos eilutės tipo chromosomą su visa kelių eilučių tipo chromosoma. Ši tikimybė taip pat yra vienas iš optimizuojamų algoritmo parametrų. Apie parametrų optimizavimą detaliau kalbama 2.4 poskyryje.

Kadangi kryžminimas yra viena iš bazinių genetinio algoritmo sąvokų, įvairūs jo variantai buvo ištirti ir įvertinti jau prieš kelis dešimtmečius. Dažniausiai literatūroje susiduriama su tolyginiu kryžminimu (angl. *uniform crossover*), kurio metu kiekviena individų chromosomų pora yra svarstoma atskirai. Su 50% tikimybe, naujai kuriamas individas gauna arba vieno arba kito tėvinio individo chromosomą. Tais atvejais, kai vieno iš tėvinių individų chromosomų sąrašas yra

ilgesnis, tiesiog svarstoma ar ši chromosoma pateks ar ne su ta pačia 50% tikimybe. Autoriai William M. Spears ir Vic Anand straipsnyje [SA91] palygino šį tolydų bei vieno ir dviejų taškų kryžminimo variantus. Tyrimo tikslas – persvarstyti ir papildomai ištirti genetinio algoritmo kryžminimo pritaikymą darbuose, susijusiuose su neuroninių tinklų modulių ir jų valdymo blokų projektavimu. Tyrimo rezultatai leido autoriams pateikti konkrečius teiginius kodėl buvo gaunami tam tikri rezultatai. Pašalinė šio tyrimo detalė, kuri yra labiau svarbi kalbant apie kryžminimą, tai kryžminimo tipų palyginimas. Gauta, kad tolydus kryžminimas labai aiškiai dominavo savo tinkamumu būtent tai uždavinių sričiai. Remiantis bendra praktika, asmenine patirtimi dirbant su genetiniu algoritmu ir savotiškai įdomiu tyrimu, galima sutikti su tolydaus kryžminimo nuopelnais. Algoritmų paieškos metu, toks kryžminimo būdas taip pat pritaikytas ir panaudotas.

Kryžminimas užima didžiausią kodo dalį iš visų operatorių, nes reikia apžvelgti visus įmanomus variantus, todėl detaliai kiekvienos eilutės aptarti nepavyks. Žemiau pateikiama apibendrinta kryžminimo kodo logikos seka, iš kurios matomi visi atliekami lyginimai ir veiksmai:

1. Sukuriama populiacijos kopija, į kurią bus dedami nauji individai.
2. Dalis populiacijos iš anksto užpildoma individais gautais iš atrankos operatoriaus.
3. Kryžminimo operatorius pradeda darbą ir kartoja veiksmus 4-11 iki kol pasiekiamas reikalingas populiacijos individų skaičius, o tokiu atveju pereinama į 12 žingsnį.
4. Sukuriami du tušti individai, vadinami vaikais, kurių genų sąrašas bus pildomas. Jiems priskiriamas nulinis tinkamumo įvertinimas.
5. Atsitiktinai, iš geriausių individų sąrašo, išrenkami du, vadinami tėvais bei perkopijuojamas jų genų sąrašas tolimesniems darbams atlikti.
6. Kol bent viename iš tėvinių individų genų sąrašų yra nepanaudotų genų, kartojami veiksmai 7-10. Kai tikslas pasiektas, pereinama į 11 žingsnį.
7. Paimami pirmi genai iš abiejų tėvų.
8. Jei bent vieno iš tėvų genų sąrašas yra tuščias, likęs genas pereina į vieną iš vaikų, arba pranyksta su 50% tikimybe.
9. Kai abu genai gauti, priklausomai nuo genų tipo, vyksta genų mainai. Jeigu tėvų genai yra paprastos eilutės tipo, jie pereina į vaikus, o ciklo arba loginio išsišakojimo atveju, genai taip pat pereina, bet sumaišomos jų vidinės eilutės, su ta pačia 50% tikimybe.
10. Iš tėvų genų sąrašo pašalinami apdoroti genai, o į vaikų genų sąrašus įdedami tėvų genai.
11. Į naują populiaciją įdedami du nauji vaikinai individai, o tėviniai individai išnyksta kartu su senąja populiacija.
12. Kryžminimo operatorius baigia darbą.

Dar viena svarbi kryžminimo pasekmė – labai spartus programinio teksto šakojimasis. Atliekant chromosomų maišymą, procesas vyksta su tam tikru atsitiktinumu. Galimi tokie atvejai, kai vienas individas gauna tik labai plačias ir sudėtingas chromosomas, taip greitai didinant jo programinį tekstą ir sudėtingumą, ko pasėkoje auga algoritmo darbo laikas. Tai apribojama sekant vidutinį populiacijos programinio kodo ilgį, užkertant kelią labai ryškiems nuokrypiams. Apie šį procesą detaliau kalbama 2.1.4.1 straipsnyje.

1.3.3.3. Mutacija

Mutacija – nedidelė tikimybė pertvarkyti vieną iš individo genų. Kodėl šis operatorius toks svarbus bei jo bendrą pritaikymą galima rasti viešai prieinamoje literatūroje. Šio tyrimo metu svarbu apsibrėžti geno pakeitimo operacijos variantus. Binarinės mutacijos atveju, tiesiog yra vykdoma bito inversija. Deja, atliekant algoritmų paiešką, chromosoma bus žymiai sudėtingesnis objektas. Idėjų, kaip tai pritaikyti būtent genetinio programavimo uždaviniui, galima rasti 1.3.5 skirsnyje aptartame William B. Langdon ir Mark Harman straipsnyje [LH13]. Autoriai mutaciją apibrėžė kaip trijų skirtingų operacijų ruletę: ištrynimas, apkeitimas ir įterpimas. Vykstant chromosomos mutacijai viena iš šių operacijų parenkama atsitiktinai, su vienodomis galimybėmis. Paprasčiausia yra ištrynimo operacija, kurios metu chromosoma yra tiesiog pašalinama. Apkeitimo ir įterpimo atveju, reikalingas naujas kodo segmentas. Jis sukuriamas remiantis turima programavimo kalbos gramatika. Aktualios C# kalbos gramatikos dalys buvo aptartos 1.2.2 skirsnyje. Abiem atvejais papildomai generuojamas operacijos tikslo indeksas. Tai vieta programiniame tekste, kurioje mutacija suveiks. Apkeitimo atveju, tai privalo būti ne triviali kodo eilutė, o įterpimui apribojimų nėra. Toks mutacijos apibrėžimas puikiai tinka ir šio darbo kontekste, todėl jis pasirinktas ir naudojamas visuose etapuose. Žemiau išdėstomi ir pateikiami visi priimti programavimo sprendimai.

Kiekvieno individo atžvilgiu mutacija veikia su maža tikimybe. Mutacijos tikimybė yra optimizuojamas parametras. Jei mutacija įvyksta, tai atsitiktinai pasirenkama viena iš trijų operacijų ir priklausomai nuo pasirinkimo, vykdomas skirtingas programinio kodo segmentas.

Kaip ir buvo minėta, pašalinimo operacija yra paprasčiausia, tiesiog įvykdomos tokios kodo eilutės:

var randomGeneToRemove = individual.GenePool[Rnd.Next(geneCount)];

individual.GenePool.Remove(randomGeneToRemove);

Čia, pirmąja eilute, iš visų individo genų visiškai atsitiktinai parenkamas vienas. Antrąja eilute jis yra pašalinamas.

Antroji operacija, eilutės modifikacija, yra žymiai sunkiau užkoduojama. Pirmiausia, kaip ir praėjusios operacijos metu, atsitiktinai pasirenkamas vienas individo genas. Tada, priklausomai nuo tipo, jis yra modifikuojamas. Šiuo atveju svarbus algoritmo kontekstas, nes negalima kurti reikšmių visiškai atsitiktinai. Atsitiktinis generavimas neaprėpiamai padidintų tiriamąją uždavinių aibę. Turint begalinę programos variantų aibę, bet baigtinius kompiuterio resursus, negalima atlikti tinkamo tyrimo. Konteksto pagalba, generuojant kintamojo reikšmės priskyrimo ar modifikacijos sakinius, reikšmės keičiamos tik parenkant jau žinomus identifikatorius, o skaitinės reikšmės aprėžiamos rėžiais. Tokiu atveju, nėra kuriami nauji kintamieji. Modifikacijos operacija leidžia žymiai pakeisti vienos eilutės darbą, išlaikant kodo struktūrą.

```
foreach (var individual in population.Individuals)
{
    if (Rnd.Next(100) < _mutationRate) // 100/rate = mutation %
    {
        switch (Rnd.Next(3)) // 33%
        {
            case 0: // delete
                var randomGeneToRemove = individual.GenePool[Rnd.Next(individual.GenePool.Count)];
                individual.GenePool.Remove(randomGeneToRemove);
                break;
            case 1: // modify
                var randomGeneToModify = individual.GenePool[Rnd.Next(individual.GenePool.Count)];
                ModifyGene(randomGeneToModify, population.Context);
                break;
            case 2: // insert
                var newGene = GenerateRandomGene(population.Context);
                var newIndex = GenerateIndex(individual.GenePool.Count);
                individual.GenePool.Insert(newIndex, newGene);
                break;
        }
    }
}
```

14 pav. *Mutacijos algoritmo kodas*

Trečioji operacija yra sudėtingiausia. Jos veikimo atveju generuojamas visiškai naujas genas, kuris neturi apribojimų bei indeksas, kuris žymi kurioje kodo eilutėje šis genas bus įterpiamas. Kadangi kompiuterio resursai yra riboti, negalima leisti šiam algoritmui dirbti visiškai atsitiktinai, todėl, kaip ir ankstesniu atveju, perduodamas tam tikras konteksto objektas, kuris praneša apie visus esamus kintamuosius, reikšmes ir reikšmių rėžius. Šiuo atveju, skirtingai nuo modifikacijos operacijos, leidžiama sukurti naują kintamąjį, taip atnaujinant kontekstą. Eilučių generavimo procesas remiasi užkoduotomis gramatikos taisyklėmis, apie kurias detaliai kalbėta 1.2.2 skirsnyje bei 2.2 poskyryje.

Kaip atrodo minimalus mutacijos operacijos kodas, pateikta 14-ame paveikslėlyje. Kodas atitinka anksčiau aprašytą veikimo logiką: operatorius iškviečiamas kiekvienam individui populiacijoje; suveikia tik su tam tikra, optimizuojama, tikimybe; vienodais šansais parenka vieną iš trijų mutacijos operacijų; iškviečia pakeitimus atliekantį kodą. Kodas yra pakankamai platus, todėl neverta peržvelgti visų trivialių detalių. Įdomiausia dalis, tai naujo kodo generavimas. 15-

ame paveikslėlyje pateiktas kodas atlieka naujo geno sukūrimą. Paprastumo dėlei buvo pasirinktas naujos reikšmės priskyrimo kintamajam variantas. Kodas remiasi aprašytomis C# programavimo kalbos gramatikos taisyklėmis. Naujos reikšmės priskyrimo eilutė susideda iš trijų pagrindinių gramatinių elementų, tai kintamojo identifikatoriaus, priskyrimo operatoriaus bei reikšmės. Kodo segmente sukuriami reikalingų gramatikos taisyklių klasių objektai ir su tam tikru atsitiktinumu bei apribojimais, perduodant kontekstą, sugeneruojama paprasta kodo eilutė.

```
var expression = new Expression();
var unaryExpression = new UnaryExpression();
var assignmentOperator = new AssignmentOperator();

var newGene = new Gene(3)
{
    VariableManipulation =
    {
        Variable = unaryExpression.Form(rnd, context),
        Operation = assignmentOperator.Form(rnd, context),
        Value = expression.Form(rnd, context)
    }
};

return newGene;
```

15 pav. *Kintamojo reikšmės priskyrimo geno generavimas*

1.3.3.4. Individų mirtis

Individų mirtis – naujas genetinis operatorius, kurio pridėjimas buvo argumentuotas 1.3.2.2 straipsnyje, kalbant apie populiacijos mirties tikimybę. Kiekvienam populiacijos individui suteikiama papildoma mirties tikimybė, kuri auga šiam individui sėkmingai pereinant iš vienos kartos į kitą. Remiantis anksčiau aptartu X. H. SHI ir kitų autorių straipsniu [SWL+03], nutarta individams leisti gyventi daugiausia tris algoritmo iteracijas, t.y. kol bus pasiekta trečia karta. Mirties tikimybė lygi **0** pirmai kartai, **0,3** antrajai kartai, **0,7** trečiajai kartai ir **1** ketvirtajai kartai. Kaip buvo minėta, sekamas populiacijos individų skaičius, kuris gali sparčiai kristi ir augti, o pasiekus kritinius etapus šis operatorius pritaiko vieną iš dviejų papildomų operacijų – karą arba individų generavimą. Tuo atveju, kai populiacija pernelyg išauga, pasirenkama karo operacija, kuri apibrėžiama kaip papildoma turnyro tipo atranka. Atrankos metu iš populiacijos atsitiktinai paimama grupė individų, iš kurių į populiaciją grįžta tik stipriausias. Karo operatoriaus algoritmo kodas ir principai detaliau aptarti 1.3.2.2 straipsnyje, kuriame kalbama apie populiacijos valdymą. Veiksmai kartojami tol, kol pasiekiamas norimas populiacijos dydis arba pritaikoma papildoma individų generavimo operacija, jei populiacija yra pernelyg maža. Šis procesas tiesiogiai remiasi

pradinės populiacijos kūrimo operacija. Taigi, individų mirties operatorius atliks kelis nesudėtingus veiksmus genetinio algoritmo rezultatų gerinimui užtikrinti.

```
for (int i = population.Individuals.Count - 1; i >= 0; i--)
{
    switch (population.Individuals[i].DeathProbability)
    {
        case 0:
        case 1:
            break;
        case 2:
            if (Rnd.Next(100) < 30)
                population.Individuals.RemoveAt(i);
            break;
        case 3:
            if (Rnd.Next(100) < 70)
                population.Individuals.RemoveAt(i);
            break;
        case 4:
            population.Individuals.RemoveAt(i);
            break;
    }
}
```

16 pav. *Individų mirties operatoriaus kodas*

Individų mirties operatoriaus kodas yra pateiktas 16-ame paveikslėlyje. Kodas yra nesudėtingas, o veiksmai paprasti. Verta pabrėžti, kad šiuo atveju per individų sąrašą yra einama atvirkščiai, nuo paskutinio, link pirmojo. Šio sprendimo priežastis yra C# programavimo kalbos trūkumas, neleidžiantis vienu metu eiti per sąrašą nuo pradžių bei keisti jo narių skaičių. Tai gana įdomus akcentas, kuri verta paminėti, nes su tuo susidurta ne vienoje kodo vietoje, kuriant genetinio programavimo karkasą algoritmų paieškai.

1.3.3.5. Migracija

Migracija – individo perėjimas iš vienos genetinio algoritmo salos į kitą. Šis operatorius yra sąlyginai paprastas. Jo proceso metu, geriausias individas iš vienos salos išimamas ir perkeliamas į kitą. Išėmimas nereiškia pašalinimo, nes šie individai išlieka savo salose, tik yra padaromos jų kopijos. Fred´ eric Lardeux ir Adrien Go´effon, straipsnyje, pavadinimu „A Dynamic Island-Based Genetic Algorithms Framework“, į šio operatoriaus pritaikymą pasigilino detaliau. Tyrimo tikslas – pristatyti dinaminį salomis grįstą genetinio algoritmo modelį. Autoriai apibrėžia kelis svarbius migracijos operatoriaus parametrus, iš kurių labiausiai tinkantys šio darbo kontekste yra:

1. Migruojančių individų skaičius.
2. Migracijos dažnumas.
3. Migruojančių individų atrankos kriterijus.
4. Migruojančių individų pakeitimo kriterijus.

Pirmieji du parametrai yra skaitiniai, jų konkrečios reikšmės turės būti tiesiogiai optimizuojamos. Migracijos dažnumas apibrėžiamas genetinių kartų skaičiumi, t.y. migracija vykdoma prabėgus tam tikram genetinio algoritmo iteracijų skaičiui. Skaitinių parametrų optimizavimo būdas aptariamas 2.4 poskyryje. Migruojančių individų atrankos kriterijus apibrėžia pagal ką migruojantys individai bus renkami. Galima apibrėžti kelis variantus: geriausio individo atranka, vidutinio populiacijos individo atranka, turnyro tipo atranka. Konkretaus pasirinkimo pasekmės nėra lengvai prognozuojamos, todėl karkaso optimizacijos metu leidžiama rinktis kiekvieną kombinaciją, ieškant tinkamiausios. Analogiškai, svarbus ir migruojančių individų pakeitimo kriterijus. Jis atsako į klausimą - kokius naujos salos individus pakeis migruojančių individų grupė. Vėl galima įžvelgti kelis individų atrankos variantus: silpniausius individus, vidutinius populiacijos individus, visiškai atsitiktinai atrinktus individus. Karkaso optimizacijos metu patikrinamas kiekvienas variantas. Taigi, atsižvelgiant į literatūroje detaliai aptartus genetinio algoritmo salos būdus, sudarytas tam tikras migracijos operatoriaus parametrų sąrašas, kuris optimizuojamas karkaso darbo metu. Operatoriaus kodas yra pakankamai paprastas. Pagrindinis kodo segmentas pateiktas 17-ame paveikslėlyje.

```
if (population.CurrentGeneration % _migrationRate == 0)
{
    var migratingIndividuals = new List<Individual>();
    for (int i = 0; i < _migrationCount; i++)
    {
        migratingIndividuals.Add(Select(population.Individuals));
    }

    foreach(var individual in migratingIndividuals)
        _migrationPool.AddToPool(_islandID, individual);

    var takenIndividuals = new List<Individual>();
    for (int i = 0; i < _migrationCount; i++)
    {
        takenIndividuals.Add(_migrationPool.TakeFromPool(_islandID));
    }

    AddToPopulation(takenIndividuals, population);
}
```

17 pav. *Migracijos operatoriaus kodas*

Pirmiausia, pirma eilute yra tikrinamas migracijos operatoriaus pritaikymo kriterijus. Kodas:

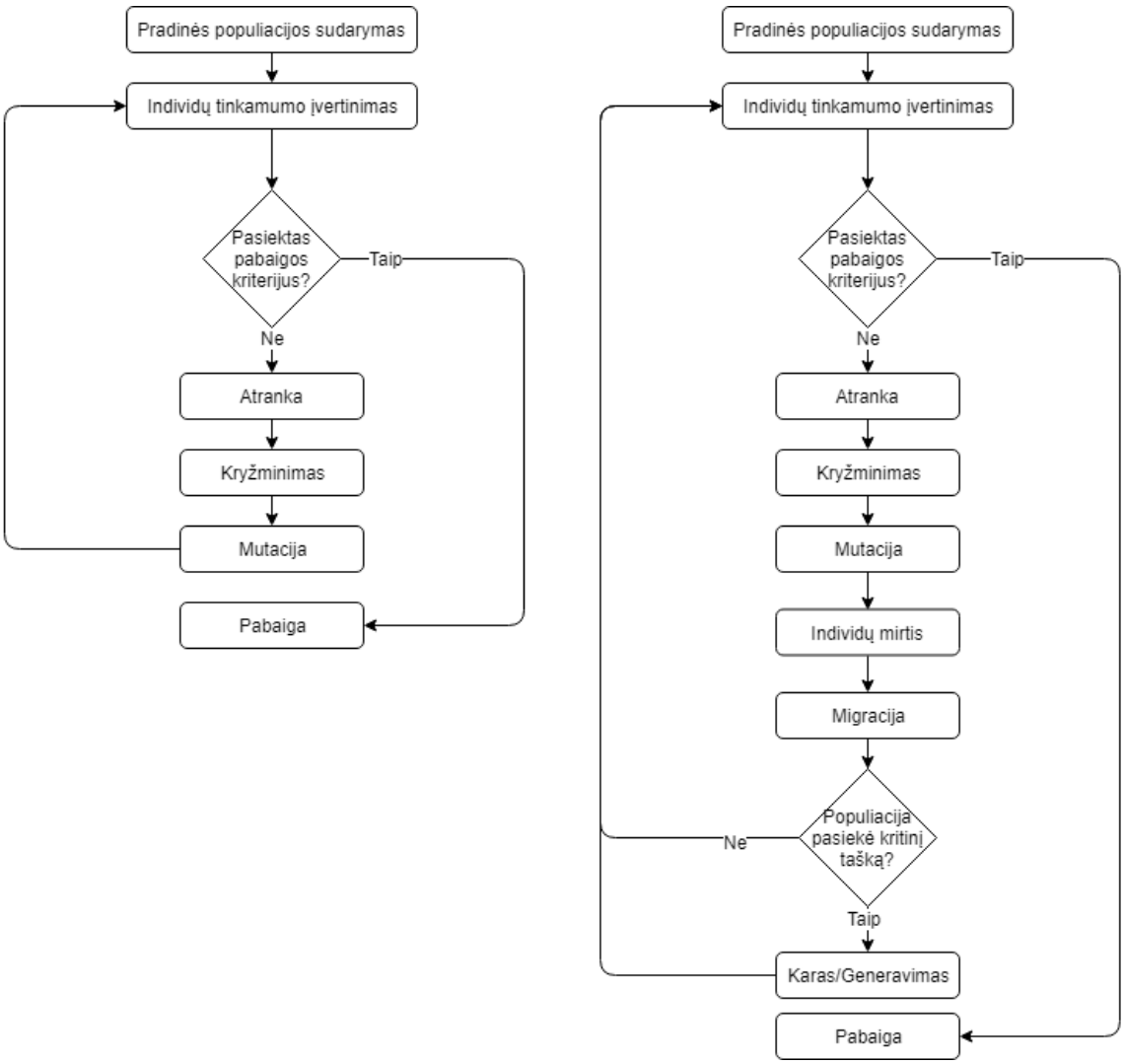
population.CurrentGeneration % _migrarionRate == 0,

nurodo, kad migracija suveiks tik kas tam tikrą, iš anksto nurodytą, genetinių kartų skaičių. Pavyzdžiui, jeigu migracijos dažnumas yra kas aštuonias kartas, tai šis kodas suveiktų ties 8, 16, 24, 32 ir t.t. algoritmo karta. Sekančiais žingsniais iš esamos populiacijos, pagal nustatytus parametrus, paimamas tam tikras skaičius individų, tada jie nusiunčiami į baseiną ir galiausiai iš

baseino paimamas toks pat skaičius individų. Paimti individai atitinkamai įterpiami į salos genetinio algoritmo populiaciją. Tokiu būdu vyksta migracijos procesas.

1.3.4. Atnaujintas algoritmas

Šiame genetinio programavimo skyriuje pristatoma kaip išvardintos modifikacijos pakeičia genetinį algoritmą. Tradicinėje literatūroje genetinis algoritmas yra apibrėžiamas gana abstrakčiai. Jis susideda tik iš pagrindinių, esminių žingsnių ir paprasčiausių operatorių, tačiau to dažniausiai neužtenka sudėtingesniuose uždaviniuose. Šio darbo metu genetinis algoritmas yra žymiai platesnis, pritaikomi nauji operatoriai, migracija ir individų mirtis, apie kuriuos detaliau buvo aprašyta ankstesniuose skyriuose. Taip pat, vyksta karo arba generavimo funkcija, jei algoritmo populiacijos skaičius žymiai nukrypsta nuo normos. Kaip atrodo paprastas ir modifikuotas algoritmas pateikta 18-ame paveikslėlyje. Šių modifikacijų programinio kodo įtaka algoritmų vystymo procese detaliai aprašyta 2.1 poskyryje, kalbant apie esminius kuriamo karkaso segmentus.



18 pav. *Paprastas genetinis algoritmas (kairėje) ir modifikuotas genetinis algoritmas (dešinėje)*

Remiantis sukurta genetinio algoritmo schema, parašytas ją atitinkantis programinis kodas. Pagrindinis kodo segmentas parodytas 19-ame paveikslėlyje. Pateiktas ciklas, kuriame įvykdomi visi genetinio algoritmo žingsniai, o pabaigoje pereinama į sekančią kartą. Paprastumo dėlei, kodas suskirstytas į penkis darbo vienetus. Pirmajame vienete vykdoma intensyvi, daug programinių resursų reikalaujanti tinkamumo skaičiavimo funkcija, taip pat populiacijos objektui perduodamas dabartinės kartos skaičius, kuris naudojamas migracijos operatoriuje. Antrame darbo vienete vyksta pabaigos kriterijaus tikrinimas. Algoritmo ciklas ***while*** vykdomas tol, kol pasiekiama apsibrėžta paskutinioji karta arba surastas individas, kurio tinkamumo lygis patenkina baigties sąlygą. Antruoju atveju ciklas yra dirbtinai stabdomas, nes skaičiavimų tęsti nebereikia. Trečiajame darbo vienete, kaip ir buvo apsibrėžta algoritmo schemoje, pritaikomi visi genetiniai operatoriai. Kiekvienas operatorius paima populiaciją ir su ja atlieka atitinkamus veiksmus bei atnaujina individus. Populiacijos objektas, pereinant iš vieno operatoriaus į kitą, nepraranda atliktų pakeitimų, todėl procesas yra priklausomas nuo operatorių iškvietimo tvarkos. Pasirinkta tvarka

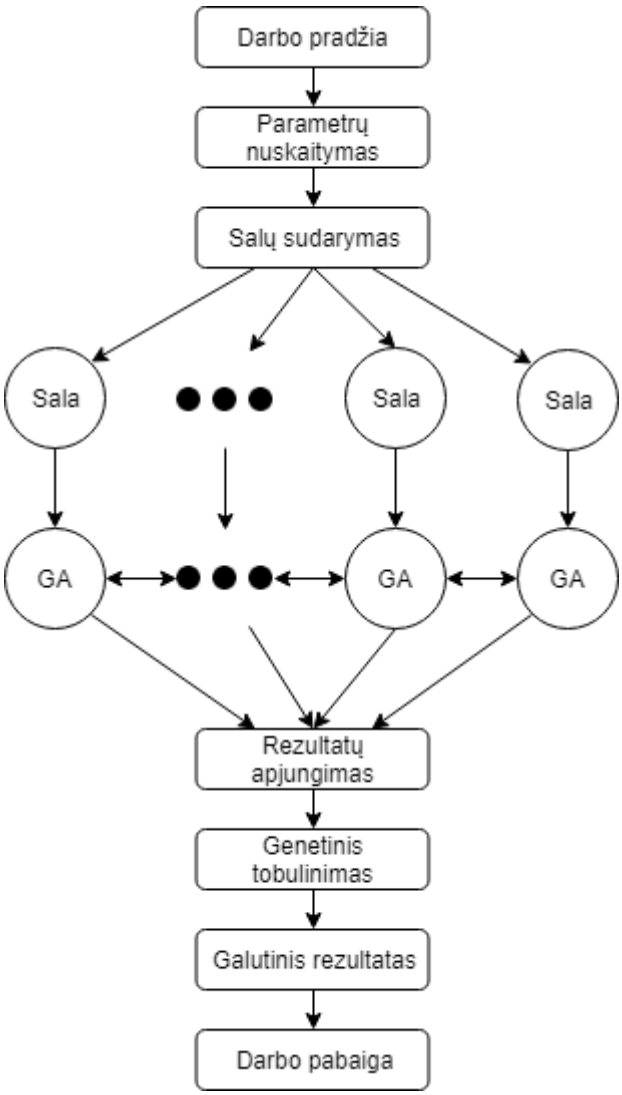
atitinka operatorių darbo įtaką galutiniam rezultatui, apie ką detaliai kalbama 1.4 poskyryje. Toliau, ketvirtuoju darbo elementu, atliekamas populiacijos skaičiaus įvertinimas. Jeigu pasiekta maksimali reikšmė, iškviečiamas karo operatorius, o jeigu populiacija per daug sumažėjo – papildomų individų generavimo operatorius. Verta pabrėžti, kad šie operatoriai veikia spontaniškai, jie gali būti niekada nepritaikomi, jeigu populiacijos skaičius išlieka pastovus ir veikia labiau kaip apsauga nuo išskirtinių atvejų.

```
while (currentGeneration < _param.GenerationLimit)
{
    population.CurrentGeneration = currentGeneration;
    _fitnessFunction.CalculateFitnessForPopulation(population);

    foreach (var member in population.Individuals)
    {
        if (member.Fitness == _endCriteria)
            break;
    }

    _selectionOperator.ApplyOperator(population);
    _crossoverOperator.ApplyOperator(population);
    _mutationOperator.ApplyOperator(population);
    _deathOperator.ApplyOperator(population);
    _migrationOperator.ApplyOperator(population);

    if (population.Individuals.Count >= population.MaxThreshold)
        _warOperator.ApplyOperator(population);

    if (population.Individuals.Count <= population.MinThreshold)
        _birthOperator.ApplyOperator(population);

    foreach (var individual in population.Individuals)
        individual.DeathProbability++;

    bestPerGeneration.Add(currentGeneration,population.Individuals
        .OrderByDescending(_ => _.Fitness)
        .First());
    averagePerGeneration.Add(currentGeneration, population.Individuals.Average(_ => _.Fitness));
    currentGeneration++;
}
```

1 2 3 4 5

19 pav. *Genetinio algoritmo kodas*

Anksčiau minėti žingsniai žymi tik pačio algoritmo darbo eigą, tačiau bendras procesas yra daug sudėtingesnis. Kaip tai galima įsivaizduoti pateikta 20-ame paveikslėlyje. Čia apskritimuose pažymėtuose simboliais *GA* būtų įterpiama visa 18-ame paveikslėlyje esanti genetinio algoritmo schema, nes kiekviena sala vykdo savo algoritmo versiją.



20 pav. *Genetinio programavimo proceso schema*

Toliau pristatoma bendrojo proceso kodo struktūra. Pirmiausia, algoritmo parametrai nuskaitomi iš įvedimo laukų, esančių vartotojo sąsajoje. Vartotojo sąsaja detaliai apibrėžta 2.1.1 skirsnyje. Reikšmių paėmimo kodas yra trivialus, todėl jis detaliai neanalizuojamas. Sekančiu žingsniu, pagal nuskaitytą norimų salų skaičiaus reikšmę, generuojamos naujos genetinio algoritmo salos. Genetinio proceso schemos kodas pateiktas 21-ame paveikslėlyje. Naujų salų kūrimas, kiekvienai iš jų suteikiant identifikacinį numerį, vyksta segmente pažymėtu pirmu numeriu. Kiekvienoje iš salų sukuriamas ir inicializuojamas naujas genetinio algoritmo variantas. Taip pat, šiuo metu sukuriamas ir inicializuojamas individų baseinas, kuris reikalingas migracijos operatoriaus darbo metu. Tai atspindima antrajame žingsnyje. Verta pabrėžti, kad genetiniam algoritmui perduodamas didelis skaičius įvairių parametrų. Vienas iš jų, tai atsitiktinių skaičių generatorius. Dėl C# programavimo kalbos apribojimų, didžiausias atsitiktinumas gaunamas tik tada, jei visos programos kontekste egzistuoja tik vienas atsitiktinių skaičių generatoriaus klasės egzempliorius. Jis yra sukuriamas paleidžiant programą ir klasių konstruktorių pagalba perduodamas visiems to reikalaujantiems objektams, kas šiek tiek apsunkina programinio kodo

rašymą ir analizę. Be generatoriaus ir baseino, genetinis algoritmas taip pat gauna uždavinio kontekstą, savo salos identifikacinį numerį bei visus genetinio algoritmo parametrus ***Parameters*** tipo, ***param*** objekte, kurio funkcija yra apjungti visus atskirus nustatymus į vieną ***Parameters*** klasės konteinerį.

```
var islandList = new List<Island>();
for (var i = 0; i < param.IslandCount; i++)
{
    islandList.Add(new Island(i));
}
                                                                    1
_pool = new Pool(Rnd);

foreach (var island in islandList)
{
    island.AddGA(new GA.GA(_pool, context, island.ID, Rnd, param));
}
                                                                    2
Dispatcher.Invoke(() => lbl_TaskStatus.Text = "In Progress...");

var resultDictionary = new Dictionary<int, GAResult>();
var progress = 0;
                                                                    3
Parallel.ForEach(islandList, island =>
{
    var result = island.RunGA();
    resultDictionary.Add(island.ID, result);

    Dispatcher.Invoke(() =>
    {
        pb_LengthyTaskProgress.Value = progress;
        lbl_CountDownTimer.Text = progress.ToString();
    });

    progress++;                                                     4
});

Dispatcher.Invoke(() => lbl_TaskStatus.Text = "Done!");
```

21 pav. *Pagrindinės karkaso veiklos kodas*

Trečiajame žingsnyje atnaujinama vartotojo sąsaja bei sukuriamas rezultatą kaupiantis objektų sąrašas. Šiuo atveju, šis sąrašas atitinka žodyno (angl. *dictionary*) objektą, nes kaupiant statistiką svarbu žinoti iš kokios salos buvo gautas kuris rezultatas, todėl rezultatai susiejami su salos identifikaciniu numeriu. Ketvirtuoju žingsniu pradedamas genetinio algoritmo darbas. Naudojama paralelizacijos biblioteka, kuri yra sisteminio C# bibliotekų rinkinio dalis. ***Parallel.ForEach*** funkcija leidžia vienu metu eiti per visus masyvo elementus, atliekant su jais tam tikras komandas. Šiuo atveju, einant per visų salų sąrašą, kiekvienoje saloje yra iškviečiamas ir paleidžiamas vidinis genetinis algoritmas. Gautas rezultatas įdedamas į rezultatų masyvą bei atnaujinama vartotojo sąsaja, rodanti progresą. Paskutinėje eilutėje vartotojo sąsaja dar karta atnaujinama, pranešant apie baigtą darbą. Čia pateikta tik supaprastinta viso kodo versija, išmetant trivialiausias dalis, tokias kaip vartotojo sąsajos elementų manipuliacija bei kintamųjų sukūrimas

ir rezultatų saugojimas. Būtent ši, viso proceso dalis, yra svarbiausia, nes joje iškviečiamas beveik visas sukurtas programinis kodas. Gilinantis į kiekvieną genetinio programavimo proceso kodo elementą būtų pereinama prie visų 1.3 poskyryje išdėstytų kodo eilučių. Tai yra genetinio algoritmo apžvalga, analizuojant jį iš aukšto lygmens.

Taigi, šiame skyriuje aptarti patys svarbiausi programavimo sprendimai, įgyvendinant praplėstą genetinį algoritmą bei bendrą karkaso veiklos schemą.

1.3.5. Genetinis tobulėjimas

Atliekant plačią sprendinių aibės paiešką genetinio algoritmo pagalba galima greitai pastebėti vieną iš jo trūkumų. Genetinis algoritmas negali labai lengvai rasti lokalių sprendinių. Genai, reikalingi pastūmėti lokalų optimumą iki globalaus, gali tiesiog neegzistuoti toje populiacijoje. Tokiu atveju, algoritmas veiks iki kol visi individai supanašės arba mutacijos pagalba šis genas atsiras ir padės pasiekti geriausią rezultatą. Tai nėra palanku dideliems ir sudėtingiems tyrimams, todėl šią problemą reikia spręsti. Vienas iš būdų – genetinis tobulėjimas (angl. *genetic improvement*). Autoriai Justyna Petke, Mark Harman, William B. Langdon, ir Westley Weimer, straipsnyje [PHL+17] jį puikiai panaudoja. Šio straipsnio tikslas – pagerinti C++ programavimo kalba parašytą programą, pritaikant automatinę kodo transplantaciją. Tyrimo metu autoriai kodo reorganizavimui (angl. *refactoring*) atlikti naudojo genetinį programavimą. Reorganizavimas – operacija, kuria programuotojai optimizuoja programos kodą, šalinant perteklinius ar nereikalingus veiksmus, nekeičiant funkcionalumo. Pagal aprašymą straipsnyje, autoriai genetinį tobulėjimą atlieka naudodami kodo donorus. Viso proceso metu tobulinama viena programa, panaudojant segmentus iš kitų programų. Kodo modifikacija vyksta eilučių principu, t.y. keičiamos tik kodo eilutės, o ne tam tikri loginiai segmentai. Straipsnyje detaliai aprašomos mutacijos operatoriaus modifikacijos. Šio genetinio tobulinimo metu, kiekviena donorinė programa yra aprašoma kaip mutacijų rinkinys. Kadangi jos nuo tobulinamosios programos skiriasi nedaug, tokiu būdu gaunamas resursų sutaupymas. Likę genetiniai operatoriai veikia pakankamai paprastai. Rezultate autoriai pateikia konkrečius skaitinius įverčius, įrodančius jų darbo sėkmę. Genetinio tobulinimo pagalba jiems pavyko sukurti net 17% greitesnę programos versiją, kuri pasirodė pranašesnė net už žmogaus rankomis optimizuotą programą. Šis tyrimas duoda daug idėjų. Galima ne tik pakartoti tokį genetinio tobulėjimo panaudojimą, bet ir pasinaudoti sukurtomis idėjomis. Algoritmų paieškos darbe genetinis programavimas kuria veikiančius algoritmus, ieškodamas jų labai plačioje aibėje. Kai gaunami pakankamai geri rezultatai ir turima tam tikra grupė veikiančių algoritmų, galima apriboti tyrimo aibę, paliekant tik

šiuos gerus egzempliorius. Tada pradedamas genetinio tobulinimo procesas. Kiekviena programa tobulinama tam tikrą laiką, o donorinėmis programomis laikomos visos kitos. Tobulinimo procesas vyksta analogiškai algoritmų paieškai: kuriami nauji individai, lyginamas jų tinkamumo lygis, pritaikomi genetiniai operatoriai. Tokiu būdu atliekama lokali paieška, kuri yra žymiai greitesnė, nes tiriamoji aibė sumažėja daug kartų.

Autoriai William B. Langdon ir Mark Harman straipsnyje „Optimising Existing Software with Genetic Programming“[LH13] genetinį tobulinimą pritaiko praktiškai. Straipsnio tikslas – parodyti, kad genetinis tobulinimas yra pritaikomas tobulinti net ir sudėtingoms, 50000 kodo eilučių turinčioms programoms. Šis tyrimas labai tiksliai parodo ne tik genetinio tobulinimo, bet ir bendro genetinio programavimo taikymą realioje užduotyje. Autoriai detaliai aptaria visas genetinio algoritmo modifikacijas ir sprendimus, kuriuos priėmė. Autorių naudojamas genetinis algoritmas remiasi programavimo kalbos gramatika. Tobulinimas vyksta, modifikuojant programinį kodą mutacijų pagalba bei stebint tam tikras kodo naudojimo metrikas, kaip, pavyzdžiui, dažniausiai iškviečiamos kodo eilutės programos darbo metu. Atsižvelgiant į metrikas, autorių algoritmas bando sukurti konkrečių programos kodo dalių supaprastinimus. Naujų programų tinkamumas matuojamas, pritaikant testavimo atvejus bei lyginant juos su programos originalo versija. Rezultate autoriams pavyksta sukurti rinkinį pakeitimų, kuriuos įgyvendinus programa paspartėja net 70 kartų. Tyrimas, kurį atliko William B. Langdon ir Mark Harman yra labai detalus ir apibrėžia daug naudingų genetinio programavimo elementų, susijusių ne tik su genetiniu tobulinimu. Pagrindinė mintis, kuri iš šio straipsnio pritaikyta algoritmų paieškoje yra viso tobulinimo traktavimas kaip atskiro genetinio programavimo uždavinio vykdymas.

Šių dviejų šaltinių pasiūlytas idėjas galima apjungti į vieną. Genetinio programavimo algoritmų paieškos metu genetinis tobulinimas taikomas algoritmo darbo pabaigoje. Turint grupę geriausių, per tam tikrą iteracijų skaičių sukurtų individų, kiekvienam iš jų pritaikomas atskiras genetinio programavimo uždavinys. Šio proceso metu kuriamos naujos programų-klonų populiacijos, kurios nuo originalaus individo skiriasi tik tam tikrais elementais. Šie klonai gaunami modifikuojant tam tikras programinio kodo vietas, pritaikant mutacijos operatorių, apribojant jį taip, kad nebūtų kuriamas naujas kodas, o tik modifikuojamas arba panaikinamas esamas. Tinkamumo lygis vertinamas tikrinant ne šių programų funkcionalumą, kas pagal kodo reorganizavimo ideologiją turi nepasikeisti, o stebint kitas kodo metrikas: kodo ilgį, vykdymo greitį, loginių išsišakojimų gylį, kintamųjų skaičių. Tokiu būdu tikimasi, kad genetinio tobulinimo pabaigoje kiekvienas individas pasieks aukščiausią savo lygį. Individas atliks korektiškus veiksmus pagal jam apibrėžtą uždavinį, o jo programinis kodas bus supaprastintas iki minimalaus. Žinoma, idealių rezultatų tikėtis iš karto negalima, genetinis tobulinimas taip pat reikalauja atskiro parametrų optimizavimo ir derinimo. Geriausių šio proceso rezultatų galima tikėtis pirmiausia

pilnai optimizavus patį genetinį algoritmą ir tik tada atlikus genetinio tobulinimo parametrų optimizavimą.

Genetinio tobulėjimo kodas remiasi jau 1.3.4 skirsnyje sukurtu genetiniu algoritmu. Skirtumas tame, kad pradinė individų populiacija yra sudaroma pagal anksčiau minėtas taisykles. Kiekvienas individas praleidžiamas per mutacijos operatorių, o gautas naujas individas tampa populiacijos dalimi. Populiacijos dydis, mutacijų skaičius vienam naujam individui bei algoritmų genetinių kartų skaičius yra optimizuojami parametrai. Genetinio tobulėjimo kodas pateiktas 22-ame paveikslėlyje. Esminiai kodo žingsniai yra pakankamai paprasti. Pirmiausia, iš rezultatų sąrašo, pasirenkamas paskutinės kartos geriausias individas. Tada, pagal jį yra sukuriama nauja populiacija, kuri užpildoma pasirinkto individo mutacijomis. Sukuriamas naujas, supaprastintas genetinis algoritmas, kuriame nėra salų. Nuspręsta salų nenaudoti, nes tobulėjimas imituoja lokalią paiešką, o migracijos operatorius padeda globalios paieškos atveju. Žinoma, tai žymiai sumažina reikalingų resursų kiekį. Genetinio algoritmo darbo pabaigoje geriausias individas yra palyginamas su prieš tobulinimo procesą gautu individu ir geriausias iš jų išsaugomas rezultate.

```
foreach (KeyValuePair<int, GAResult> entry in resultDictionary)
{
    var bestIndividual = entry.Value.BestIndividualsPerGeneration.Values.Last();

    var newPopulation = new Population(_context);
    FillPopulationWithMutations(
        bestIndividual,
        newPopulation,
        _param.RefactoringMutationsPerIndividual,
        _param.RefactoringPopulationSize);
    var ga = new GA(_context, Rnd, _param);
    var gaResult = ga.GeneticAlgorithmSimplified(newPopulation, _param.RefactoringGenerationCount);

    if (gaResult.BestIndividualsPerGeneration.Last().Value.Fitness > bestIndividual.Fitness)
        resultDictionary[entry.Key] = gaResult;
}
```

22 pav. *Genetinio tobulėjimo kodas*

1.4. Operatorių pritaikymo tvarka

Kaip jau buvo minėta ankstesniuose skyriuose, genetinis algoritmas buvo modifikuotas, pridedant papildomų operatorių. Kyla klausimas, kokia operatorių pritaikymo tvarka yra tinkamiausia. Tam nuspręsti, pirmiausia reikia įvertinti kiekvieno operatoriaus poveikį bendram algoritmo darbui. Atrankos operatorius, tirtoje literatūroje, visada pritaikomas pirmasis. To priežastis yra pakankamai paprasta – atrankos metu išsaugoma dalis geriausių individų, o tai tiesiogiai įtakoja geriausią bei vidutinį algoritmo kartos įvertį. Vien dėl to, verta apsistoti ties šiuo operatoriumi kaip pirmuoju ir pasikliauti literatūros šaltiniais. Antruoju operatoriumi dažniausiai

yra skelbiamas kryžminimas. Kryžminimas užpildo tuščią populiacijos dalį, atsiradusią po atrankos, todėl tai taip pat yra neatsiejamas operatorius, kurio negalima vykdyti bet kuriuo kitu metu. Taigi, atranka ir kryžminimas tarpusavyje susiję ir yra vykdomi algoritmo pradžioje.

Visi likę genetiniai operatoriai, tai mutacija, migracija bei individų mirtis neturi tokio aiškaus tvarkos apibrėžimo. Mutacijos operatorius pritaikomas su nedidele tikimybe, o migracija ir individų mirtis aktyvuojasi tik kas tam tikrą iteracijų skaičių. Kad galėtume įvertinti operatorių įtaką, verta apskaičiuoti vidutinį skaičių individų, kurį modifikuotų kiekvienas operatorius. Tarkime tiriama ***P*** individų aibė, o algoritmas vykdomas tol, kol pasiekiama ***G*** genetinė karta. Mutacija, kuri, tarkime, veikia su ***M*** tikimybe, vidutiniškai pakeistų ***P*G*M*** individų. Migracija suveiktų tik kas tam tikrą skaičių iteracijų, potencialiai algoritme pakeičiant ***G/p*N*K*** individų, kur ***N*** žymi migruojančių individų skaičių, ***K*** žymi salų skaičių, o ***p*** žymi migracijos dažnumą, t.y. kas kiek iteracijų ji vyksta. Individų mirties operatoriaus vidutinį paveiktų individų skaičių apskaičiuoti sunkiausia, nes jis visiškai priklauso nuo kitų operatorių darbo rezultato. Geriausiu atveju, individų mirties operatorius kas ***T*** skaičių iteracijų sunaikina visą populiaciją, t.y. paveikia ***P*G/T*** skaičių individų. Blogiausiu atveju, kai kas bent per ***T-1*** skaičių algoritmo iteracijų, visi individai yra pergeneruojami prieš tai buvusių operatorių, šis operatorius nepaveiktų nei vieno. Atsižvelgiant į tai, keičiant parametrus, galima kiekvieną iš minėtų operatorių padaryti labiausiai įtakojančiu. Svarbiausiais ir didžiausią įvertį turinčiais parametrais yra laikomi individų skaičius ***P*** bei iteracijų skaičius ***G***. Kadangi mutacija tiesiogiai nuo jų priklauso, galima teigti, kad ji ir bus įtakingiausia, plečiant karkaso darbą link didesnės populiacijos. Taigi, liko apsispręsti tarp migracijos ir individų mirties operatorių, o tai yra gana paprasta. Kadangi migracija priklauso tik nuo iteracijų skaičiaus, ji paveiks mažiausią skaičių individų. Parametrai ***N*** ir ***K*** pagal dizainą, tikrai nepasieks pakankamai aukštų reikšmių. Taigi, galutinė operatorių pritaikymo tvarka yra atranka, kryžminimas, mutacija, individų mirtis bei migracija.

2. Tiriamoji dalis

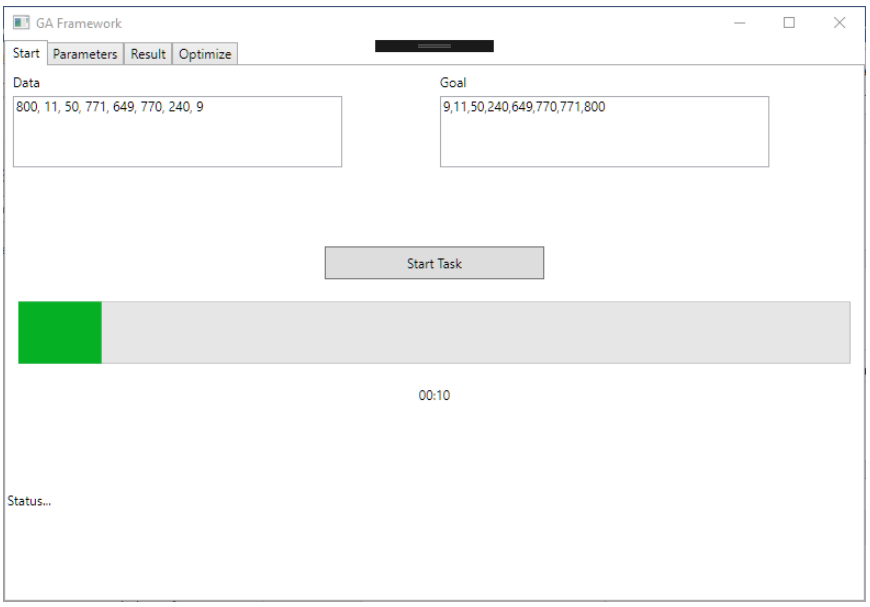
Šiame skyriuje detaliai pristatoma darbo tiriamoji dalis. Pirmiausia, aptariamas sukurtas programinis karkasas, vartotojo sąsaja, klasių struktūra. Išdėstomas svarbus pradinės populiacijos kūrimo procesas bei viso karkaso veikimo spartos aspektai, greitinimo metodai. Toliau aptariami sudėtingesni gramatikos pritaikymo aspektai. Pereinama prie konkrečių algoritmų paieškos proceso pristatymo, kaip jie buvo gauti, kokie žingsniai buvo įgyvendinti ir su kokiomis problemomis susidurta. Taip pat, išdėstomi visi sukurto programinio įrankio parametrai bei jų automatizuoto optimizavimo būdas, sprendimai ir įgyvendinimas. Galiausiai, aptariamas gautų algoritmų kodo vertinimo būdas, procesas ir rezultatų analizė.

2.1. Karkasas

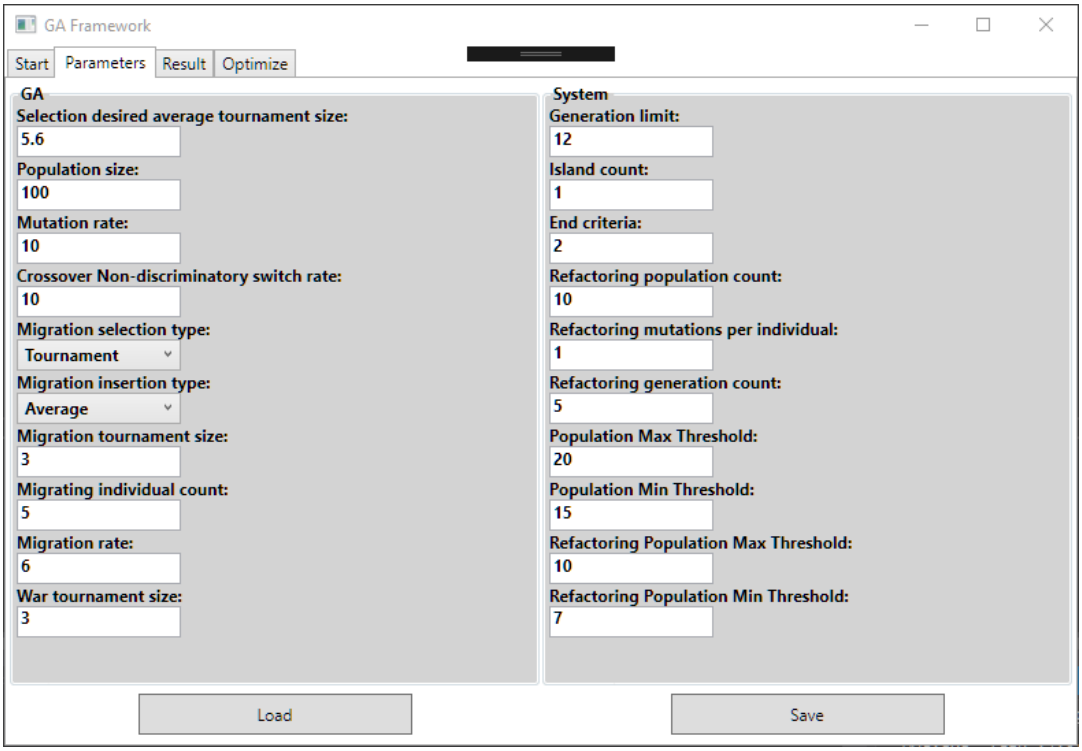
Šiame skyriuje aprašomas sukurtas karkasas, kuriame vykdomi visi genetinio programavimo, tobulinimo ir paieškos procesai. Karkasas taip pat aprėpia parametrų nustatymą, rezultato pateikimą bei parametrų optimizavimą. Karkasas, kaip visuma, yra kompiuterinė programa turinti daug funkcijų, kurias galima keisti vartotojo sąsajos pagalba, todėl skyriuje detaliai apžvelgiama sukurta sąsaja bei parametrų nustatymo funkcijos. Išdėstomi esminiai kodavimo sprendimai, kurie dar nebuvo aptarti ankstesniuose skyriuose, taip pat, giliau apžvelgiami svarbesni įgyvendinimo sprendimai, gauti rezultatai ir problemos, su kuriomis buvo susidurta bei kaip jos buvo išspręstos.

2.1.1. Vartotojo sąsaja

Sukurto programinio karkaso vartotojo sąsaja susideda iš keturių pagrindinių langų. Navigacija tarp jų realizuota skirtukų (angl. *tabs*) pagalba. Pirmojo skirtuko vaizdas pateiktas 23-ame paveikslėlyje. Čia matomas genetinio algoritmo darbo pradžios mygtukas, progreso juosta, laikas, kuris praėjo nuo darbo pradžios bei informacinių žinučių etiketė. Pagrindinė šio vartotojo sąsajos lango paskirtis, tai paleisti karkaso darbą bei stebėti progresą ar iškilusias problemas.



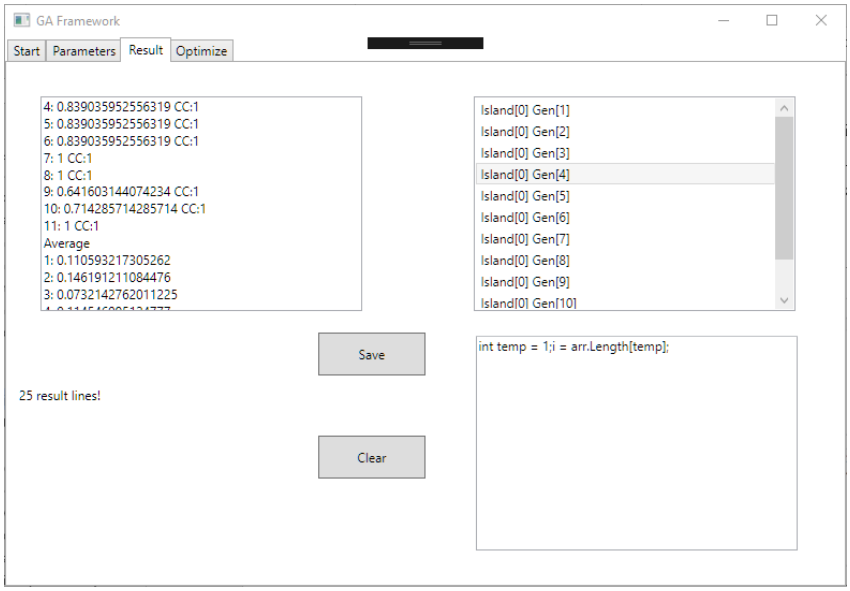
23 pav. *Vartotojo sąsajos pirmasis langas*



24 pav. *Vartotojo sąsajos antrasis langas*

Antrasis vartotojo sąsajos skirtukas pateikia visus karkaso sekamus parametrus. Sąsajos vaizdas pateiktas 24-ame paveikslėlyje. Skirtuke pateikiamos dvi pagrindinės parametrų grupės: parametrai susiję su genetinio algoritmo darbu kairėje ir parametrai susiję su viso proceso darbu dešinėje. Parametrus galima laisvai keisti prieš pradedant skaičiavimus. Beveik visi sekami parametrai yra skaitinio tipo. Detalus parametrų aprašymas pateiktas 2.4 poskyryje. Du parametrai, migracijos atrankos tipas bei migracijos individų įterpimo tipas yra neskaitiniai, o pasirenkami

parametrai. Programiniame kode jie atitinka ***enum*** tipo kintamuosius. Jiems nustatyti užtenka praskleisti parinkčių dėžutę (angl. *combo box*) ir pasirinkti vieną iš galimų reikšmių. Algoritmo darbo metu visos reikšmės yra nuskaitomos ir perduodamos reikalingiems elementams. Skaičiavimams prasidėjus šis vartotojo sąsajos skirtukas yra išjungiamas ir kol darbas nepasiekė pabaigos, tolimesnių pakeitimų atlikti negalima.

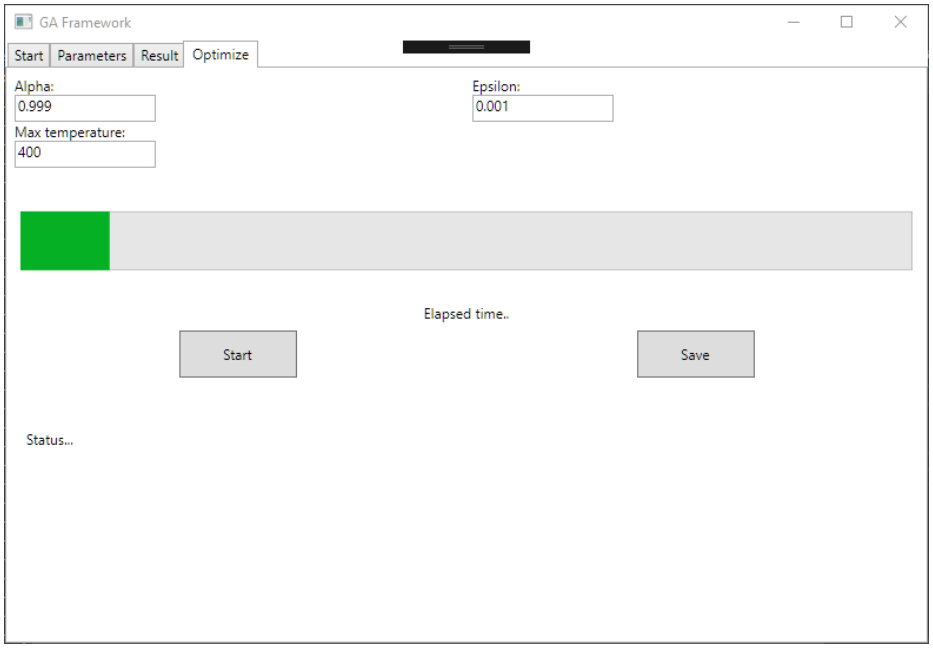


25 pav. *Vartotojo sąsajos trečiasis langas*

Trečiasis vartotojo sąsajos langas, pateiktas 25-ame paveikslėlyje, yra skirtas rezultatų peržiūrai. Algoritmo darbo pabaigoje, į tekstinį langą atspausdinama informacija apie kiekvienos salos ir kiekvienos iteracijos, kurias toje saloje vykdė genetinis algoritmas, geriausią bei vidutinį individų tinkamumo įvertinimą. Duomenys pateikiami laisvai kopijuojamu formatu lengvam perkėlimui į kitas sistemas, pavyzdžiui *Microsoft Excel*. Taip pat, pridėta informacinė etiketė, pranešanti apie rezultato eilučių skaičių. Šiame sąsajos lange pridėti du papildomi mygtukai bazinėms funkcijoms atlikti. Vienas iš jų išsaugo rezultatą į tekstinį failą, o kitas išvalo rezultato langą. Lango dešinėje pusėje pateikiamas geriausių individų, gautų tam tikroje saloje, sąrašas. Jis užpildomas darbo pabaigoje. Paspaudus ant sąrašo eilutės, tekstinėje dėžutėje esančioje po sąrašu parodomas geriausio individo programinis kodas. Tai leidžia greitai analizuoti koks programinis kodas yra kuriamas.

Paskutinysis vartotojo sąsajos langas leidžia reguliuoti automatizuotą karkaso parametrų optimizacijos procesą. Sąsajos lango pavyzdys pateiktas 26-ame paveikslėlyje. Šis sąsajos langas yra pakankamai paprastas. Tiesiog nurodomi optimizacijos parametrai ir paspaudžiamas mygtukas pradėti. Darbui pasibaigus rezultatą galima išsaugoti kaip atskirą failą, kurį galima panaudoti vėliau, nurodant karkaso parametrus. Optimizacijos procesas vyksta simuliuojant algoritmų paieškos procesą. Procesas pilnai įvykdomas ir gaunami geriausi individai. Šis rezultatas yra

įsimenamas, o eksperimentas kartojamas, pakeičiant karkaso parametrus. Sąsajos langas parodo optimizacijos proceso užimtą laiką bei progresą link pabaigos.



26 pav. *Vartotojo sąsajos ketvirtasis langas*

Sukurta vartotojo sąsaja yra paprasta ir minimali, siūlanti tik bazines funkcijas. Šiame tyrimo etape sudėtingesnių sąsajos elementų neprireikė.

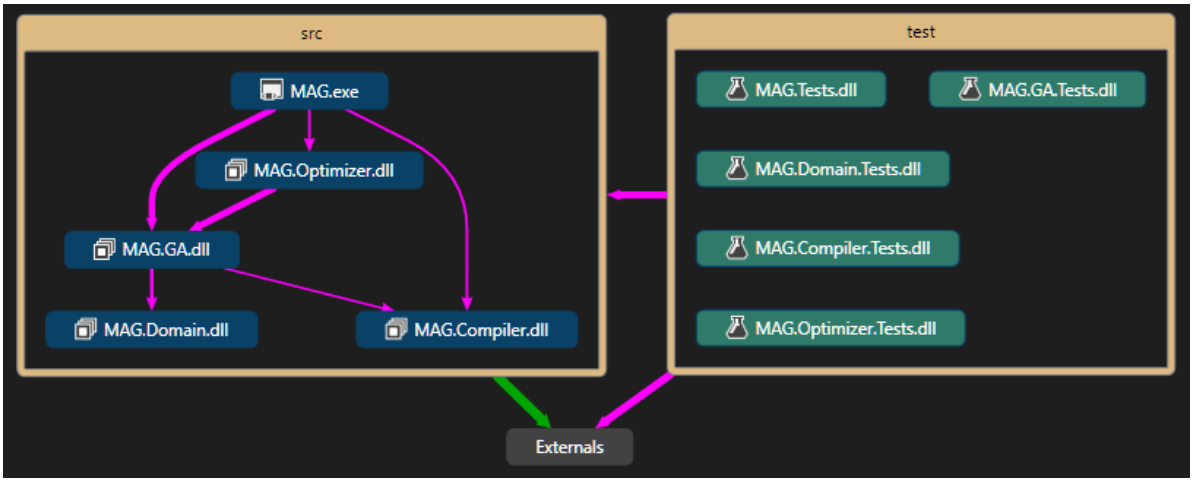
2.1.2. Klasių struktūra

Kuriamas karkasas susideda iš labai didelio skaičiaus skirtingų programinių elementų, todėl apžvelgti kiekvieną iš jų individualiai nėra prasmės. Visas programinis kodas yra suskirstytas į šešis atskirus projektus:

- *Compiler* – projekte egzistuoja kodas, susijęs su 1.2.3 skirsnyje detaliai aprašytu automatiniu kodo kompiliatoriumi.
- *Domain* – projekte laikomas kodas, susijęs su C# programavimo kalbos gramatika ir susideda iš didžiausio skaičiaus klasių.
- *GA* – projekte laikomas kodas, susijęs su genetiniu algoritmu bei genetiniu programavimu. Tai visas kodas aprašytas 1.3 poskyryje. Individualių klasių skaičius yra taip pat didelis bei klasės savyje saugo daug logikos.
- *UI* – projektas, skirtas vartotojo sąsajos kodui. Čia egzistuoja tik sąsajos elementų aprašymo kodas, parametrų nuskaitymo logika bei algoritmo darbo startavimo procesai, reaguojantys į mygtukų paspaudimus.

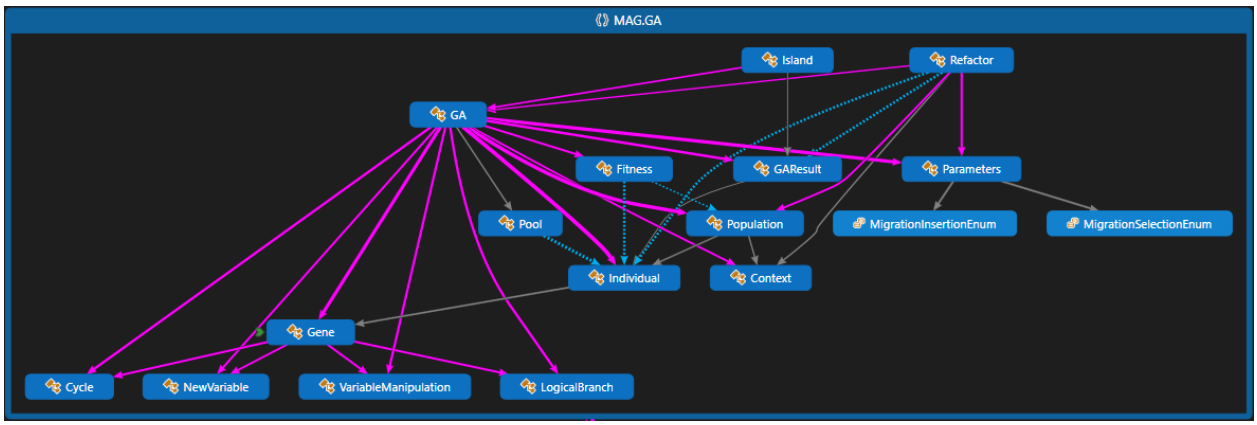
- *Tests* – projektų grupė, skirta automatizuotų testų kūrimui. Šio darbo metu sukurti esminiai programinio kodo logiką patikrinantys testai. Testai padengia beveik visą programinį kodą, kuris yra svarbus šio darbo kontekste.
- *Optimizer* – projektas, skirtas karkaso parametrų automatiniam optimizavimui. Projekte saugomos klasės įgyvendinančios vėsinimo imitacijos (angl. s*imulated annealing*) optimizacijos algoritmą, apie kurį detaliai kalbama 2.4.1 skirsnyje.

Šių projektų sąsaja tarpusavyje pateikta 27-ame paveikslėlyje. Kadangi projektų skaičius nedidelis, ši schema yra pakankamai aiški. Pagrindinis programinis kodas slypi ***src*** grupėje, kurioje egzistuoja penki iš anksčiau išvardintų projektų. Jų tarpusavio ryšiai nesudėtingi. Čia ***MAG.exe*** atitinka *UI* projektą, o likusių paskirtis aiški iš jų pavadinimų. Testavimui sukurta visiškai atskira grupė, kuri naudoja išorines testavimo bibliotekas. Testuojamas kodas kviečiamas iš visų projektų esančių ***src*** grupėje, todėl yra svarbus ryšys tarp jų. Abi grupės, kaip ir buvo minėta, naudoja tam tikrą skaičių išorinių, pagalbinių bibliotekų, čia pažymėtų ***Externals***, tačiau tai nėra svarbu šio darbo kontekste. Taigi, bendra karkaso struktūra yra gana paprasta.



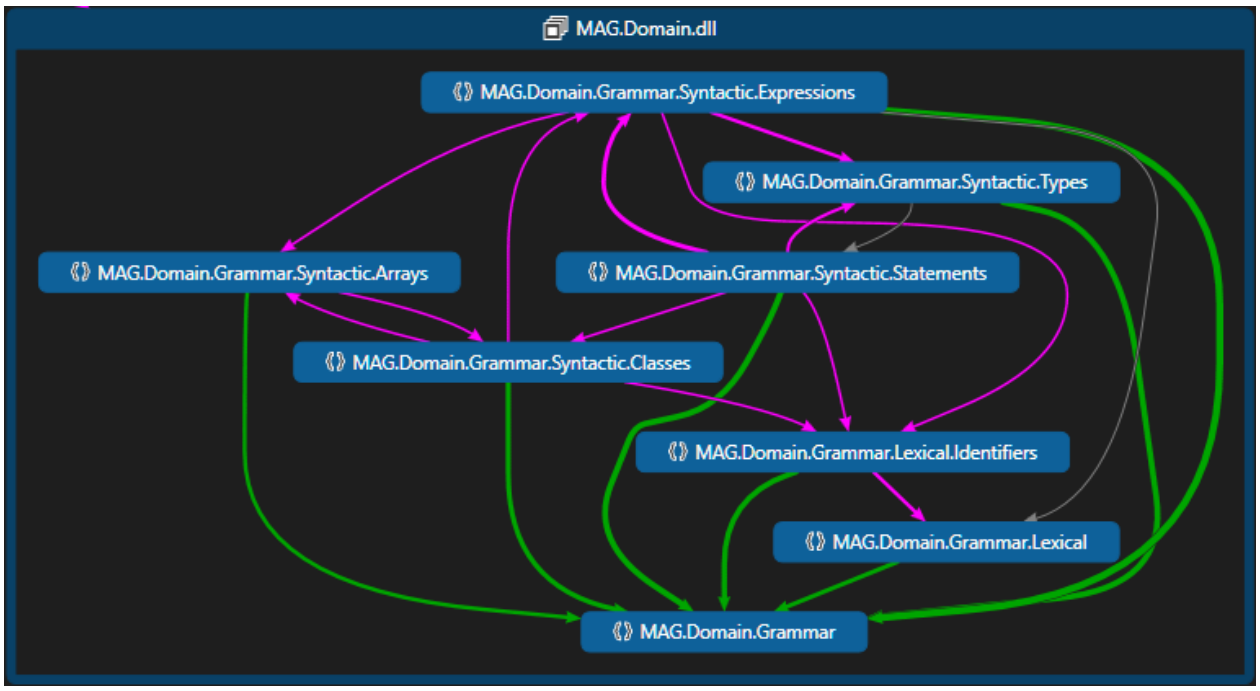
27 pav. *Projektų sąsajos tarpusavyje*

Verta giliau apžvelgti klasių ryšius *GA* projekte, kuris saugo genetinio algoritmo klases. Klasių ryšiai tarpusavyje bei jų sąrašas pateiktas 28-ame paveikslėlyje. Nesunku pastebėti, kad svarbiausia klasė, kuri apjungia beveik visas likusias, yra ***GA***, atitinkanti genetinį algoritmą, kaip objektą. Šioje klasėje vyksta genetinio algoritmo vykdymas, kurio detali schema buvo pateikta 1.3.4 skirsnyje. Taip pat, paveikslėlio apačioje pateikta atskira genetinių operatorių grupė. Ši grupė susideda iš visų klasių atitinkančių genetinius operatorius. Ryšiai tarp projekto *GA* klasių yra svarbūs, nes nuo jų darbo veikimo priklauso gaunamų rezultatų kokybė. Klasių projektavimas ir vidinių esybių suskirstymas buvo svarbi šio darbo dalis.



28 pav. *GA projekto klasių ryšiai*

Galiausiai, projekte *Domain*, kaip ir buvo minėta, saugomos visos klasės atspindinčios C# programavimo kalbos gramatiką. 29-ame paveikslėlyje pateikta klasių grupių ryšių diagrama. Kiekvienoje grupėje yra didelis skaičius skirtingų klasių, todėl diagrama, kurioje jos visos matomos, yra pernelyg sudėtinga, o ryšiai painūs, todėl pasirinktas šis vaizdavimo būdas. Kiekviena klasių grupė yra glaudžiai susijusi su bent keliomis kitomis grupėmis. Tokį ryšių persidengimą galima paaiškinti dar kartą trumpai apžvelgiant 1.2.2 skirsnyje aptartas gramatikos taisykles. Sudarant naują sakinį programavimo kalboje, įmanomas labai didelis galimų elementų skaičius. Elementai, analogiškai, gali taip pat susidėti iš skirtingų elementų savo viduje, todėl vykstant kodo generavimui, labai greitai sukuriamas gilus išsišakojimų medis, kurio paskutinis lapas yra terminalinis simbolis. Tik pasiekus terminalinį simbolį galima grįžti vienu žingsniu atgal ir analizuoti sekantį sakinio elementą. To pasėkoje, kiekviena klasių grupė turi ryšius su beveik visomis kitomis grupėmis, nes labai tikėtina, kad bent vienas jose esantis elementas generuoja kodą iš kitos grupės. *Domain* projektas saugo sudėtingą ir didžiausią klasių rinkinį, kurio sukūrimas buvo viena iš sunkiausių šio darbo dalių. Sukurtas klasių sąrašas nėra pilnas, remiantis pirmame šio darbo skyriuje išvardintais sprendimais bei 2.1.3 skirsnyje išdėstytomis mintimis, dalis gramatikos taisyklių buvo atmesta, nes sudėtingi programavimo kalbos dariniai nėra reikalingi algoritmų kūrime. Taigi, *Domain* projekte patalpinta C# programavimo kalbos gramatikos imitacija.



29 pav. *Domain projekto klasių ryšiai*

2.1.3. Pradinė populiacija

Pradinės algoritmo populiacijos sudarymas – vienas iš sudėtingiausių šio karkaso etapų, nes kodo generavimas remiasi programavimo kalbos gramatikos taisyklėmis. Tam, kad būtų sukurtas taisyklingas kodas, reikia pilnai perrašyti visų gramatikos taisyklių elementus, kurių kiekvienas turi begales skirtingų variacijų. Be to, kad skaičiavimai būtų baigtiniai, negalima kurti visiškai atsitiktinio kodo, jis turi būti apribotas tam tikrame kontekste. Įsivaizduokime kaip savo ruožtu algoritmo paieškos uždavinį atliktų žmogus. Pirmiausia, gavus tam tikrą duomenų rinkinį, labai tikėtina, kad būtent į šį rinkinį bus kreipiamasi bent kelis kartus, norint iš jo paimti duomenis. Dėl to, sakinių generavimas, kuriuose identifikatoriai yra visiškai atsitiktiniai, yra nelogiškas veiksmas. Atsižvelgiant į tai, būtų verta apriboti naujų identifikatorių kūrimo veiksmus ir tai leisti daryti tik išskirtiniais atvejais, o kitais – rinktis iš jau esamų identifikatorių sąrašo kontekste. Einant toliau taip pat tikėtina, kad žmogus, kurdamas algoritmą, nenaudos sudėtingų programavimo kalbos elementų, tokių kaip naujos klasės kūrimo, paveldėjimo, darbo su nesaugiu kodu ir panašiai. Tokie elementai, nors ir gali būti panaudoti, tikrai neprives prie optimalaus, greito ir paprasto algoritmo, kurio tikimės iš kuriamo įrankio. Dėl to, galima stipriai apriboti naudojamą C# programavimo kalbos gramatiką, išimant taisykles ir elementus, kurių tikrai neprireiks generuojant algoritmo kodą. Visi minėti sprendimai yra labai svarbūs, kuriant pradinę algoritmo populiaciją, nes taip sumažinama tiriamų variacijų aibė, dėl ko siektinas rezultatas randamas žymiai greičiau.

Pirmajame šio darbo skyriuje, analizuojant su tema susijusius darbus, buvo aptartas autoriaus Kenneth E. Kinnear Jr. straipsnis, kuriame nurodyti tam tikri pradinės populiacijos bei pačio algoritmo parametrai. Autorius pasirinko tirti 1000 individų populiaciją bei algoritmui leisti veikti 49 iteracijas. Šio tyrimo atveju, pirmiausia atliekamas tyrimas atkartojantis minėto straipsnio apimtį. Tai atlikus, populiacija palaipsniui didinama stebint rezultatus. Rezultatams užsistovėjus, didinamas iteracijų skaičius. Tai tęsiama, kol išvystomas pakankamai geras individas, arba skaičiavimų laikas tampa nepatenkinamu. Detalesnis skaičiavimo laiko bei tiriamos populiacijos dydžio aprašymas pateikiamas sekančiame skyriuje. Teoriniai individų populiacijos sudarymo ypatumai buvo trumpai aptarti 1.3.2.2 straipsnyje.

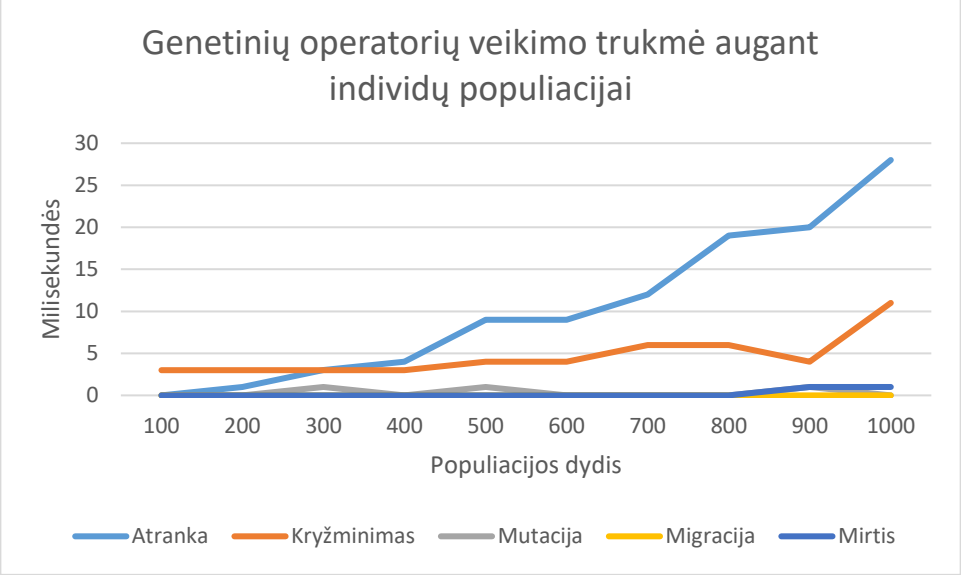
2.1.4. Aplinkos sparta

Aplinkos sparta – kaip greit atliekamos karkaso funkcijos. Tai svarbus optimizacijos etapas, nes kuo trumpesnį laiką vyks skaičiavimai, tuo daugiau individų bus galima tirti. Spartai didinti yra įvairių būdų. Vienas paprasčiausių - galingesnės kompiuterinės technikos naudojimas, tačiau tai ne visada padeda, tam tikras programinis kodas gali tiesiog nesugebėti pilnai išnaudoti esamų resursų. Taigi, svarbu stebėti programinio kodo vietas, kuriose vyksta daugiausia skaičiavimų, ieškoti būdų joms supaprastinti.

Kodo greičiui matuoti naudojama C# programavimo kalbos biblioteka ***Stopwatch***. Šios bibliotekos pagalba sukuriamas virtualus chronometras, kurį galima automatiškai paleisti ir sustabdyti tam tikrose kodo vietose. Pats kodo naudojimas yra trivialus, todėl į tai nebus gilinamasi. Lėtų funkcijų įtaka matoma atliekant pilnus skaičiavimus su optimaliu parametrų sąrašu. 1.3.3.1 straipsnyje gautas spartos įvertinimas taip pat parengtas ***Stopwatch*** bibliotekos pagalba.

Viena lėčiausių karkaso vietų yra lengvai pastebima – tai individo tinkamumo skaičiavimas. Kadangi kiekvieno individo kodas yra tikrinamas realioje programoje, ji turi būti kompiliuojama ir paleidžiama. Kuo individų daugiau, tuo ilgiau šis procesas užtruks. Deja, nėra paprastų būdų išvengti tokio sulėtėjimo. Nepaisant to, galima bandyti greitinti likusias karkaso dalis. Plečiant individų populiacija, net ir paprasčiausi veiksmai reikalauja daugiau laiko. Atkreipkime dėmesį į genetinių operatorių darbą. Genetinių operatorių darbo trukmė augant populiacijos dydžiui pateikta 30-ame paveikslėlyje. Galima pastebėti, kad įtakingiausi yra tik du pagrindiniai operatoriai, atranka bei kryžminimas. Kryžminimo atveju didėjanti trukmė yra nekontroliuojama. Kadangi vis auganti populiacija turi būti užpildoma vaikais, sukurtais kryžminant tėvų genus, daug spartos išlošti čia nėra šansų. Nepaisant to, kryžminimas veikia pakankamai greitai net ir prie didelio populiacijos skaičiaus. Sekančiu atveju, atranka, nors ir užtrunka ilgiau, gali būti

kontroliuojama. Atranka sukuria mažus turnyrus tarp nedidelių individų grupių. Kuo populiacija didesnė, tuo daugiau turnyrų yra kuriama. Vienas iš karkaso optimizuojamų parametrų yra tokio turnyro dydis, kuo jis mažesnis, tuo greičiau bus įvykdomas turnyras ir tuo greičiau bus pabaigtas operatoriaus darbas. Modifikuojant šį parametrą galima kontroliuoti operatoriaus spartą. Taigi, lėčiausia karkaso dalis yra neišvengiama, bet darbo trukmė gali būti spartinama keičiant kitus, nedidelę įtaką turinčius algoritmo elementus.

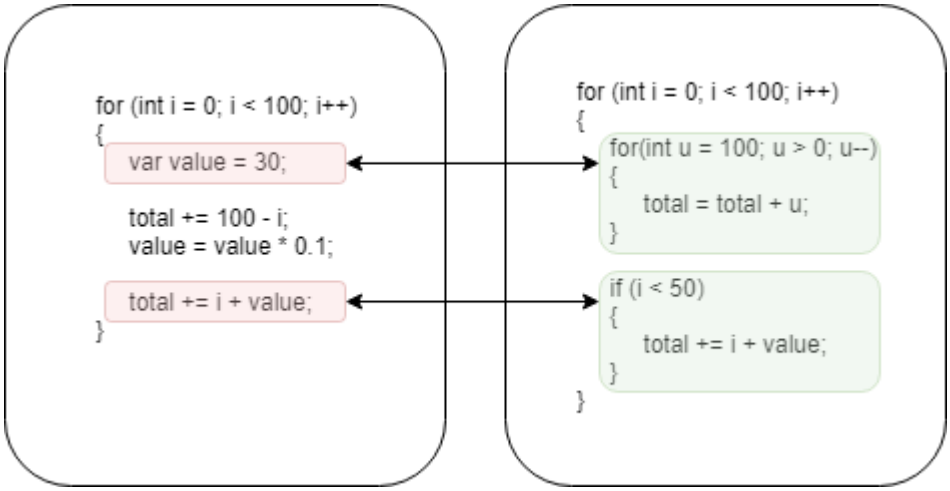


30 pav. *Genetinių operatorių trukmė augant individų populiacijai*

Vienas iš papildomų spartos klausimų iškyla atliekant karkaso optimizaciją. Kadangi pagrindinė idėja – pakartotinai atlikti viso karkaso darbą vis pakeičiant pradinių nustatymų reikšmes, tai užtruks žymiai ilgiau nei paprastas tyrimas. Atsižvelgiant į tai, optimizacija atliekama dirbant su žymiai mažesniu individų skaičiumi, o algoritmas kartojamas su mažiau iteracijų. Detaliau apie optimizavimo procesą aprašyta 2.4.1 skirsnyje. Optimizacijos algoritmas taip pat turi tam tikrus parametrus, nuo kurių priklauso kiek kartų kartojamas optimizacijos procesas. Pasirinkta darbą atlikti su šimto individų populiaciją, o individams leisti evoliucionuoti iki dešimtos kartos, ko pabaigoje geriausias individas pamatuojamas ir pereinama prie sekančio optimizacijos etapo. Prie tokių parametrų, vienas karkaso paieškos procesas užtrunka 143 sekundes, o optimizacija vykdoma 122 kartus. Naujo parametrų rinkinio sudarymas užtrunka vos kelias sekundės dalis, todėl tai nėra spartos klausimas. Iš esmės, visas proceso laikas yra sunaudojamas karkaso darbui atlikti.

2.1.4.1. Programinio kodo augimas

Viena iš problemų, su kuria buvo susidurta darbo metu, tai staigus programinio kodo augimas individuose, vykstant kryžminimo operacijai. Apie šią problema trumpai užsiminta 1.3.3.3 straipsnyje. Kryžminimo metu, individai keičiasi savo genais. Tikėtina, kad tam tikri genai gali būti labai sudėtingi, o juos atspindintis programinis kodas yra pakankamai ilgas. Egzistuoja tikimybė, kad tokių sudėtingesnių genų rinkinys gali užsibūti viename iš individų, ko pasėkoje išauga bendras šio individo kodas. Karkaso darbo metu, populiacijoje toks individas gali išlikti ir plisti, perduodant savo sudėtingus genus kitiems individams. Įmanoma pasiekti tokią situaciją, kada beveik visas individų kodas yra ilgas ir sudėtingas. Tai sukelia papildomą apkrovą keliose vietose. Pirmiausia, skaičiuojant tinkamumą, programinis kodas yra vertinamas, todėl ilgesnis kodas lydi link ilgesnio vertinimo proceso. Sudėtingi genai taip pat turi savo aspektų, pavyzdžiui, jų mutacijos operacija turi papildomų veiksmų, kuriuos reikia atlikti. Ilgas programinis kodas bei genų sąrašas ir visos tai atspindinčios programinės klasės ir konstruktai turi būti saugomi kompiuterio atmintyje. Nors tai ir nėra labai aukšto prioriteto problema, tačiau plečiant šiame darbe sukurtas idėjas, į tai reikia atsižvelgti. Kuo mažesni individai, tuo mažiau resursų prireiks jiems apdoroti. Kaip ir buvo minėta, ši problema yra gana aktuali, o jos sprendimas lydi link greitesnio kuriamos aplinkos darbo.



31 pav. *Programinio kodo augimas*

Problemos vizualizacija pateikta 31-ame paveikslėlyje. Kryžminimo metu maišantis genams, kurie atitinka programinio kodo eilutes bei sudėtingesnius elementus kaip ciklas ir loginis išsišakojimas, jų mainai nėra lygus. Pateiktu atveju kairysis individas perima du, kodo eilučių atžvilgiu stambesnius elementus, taip sukuriant žymiai didesnį individą. Pavyzdyje kodo eilutės

neturi logiškos prasmės ir pateiktos tik problemai paaiškinti, tačiau realūs pavyzdžiai yra pakankamai panašūs.

Problemos sprendimo būdas yra sąlyginai paprastas. Vykstant genetinio algoritmo darbui, pritaikius kryžminimo operatorių, yra patikrinamas kiekvieno individo programinio kodo ilgis. Jeigu individo kodas peržengia maksimalų pasirinktą kodo eilučių kiekį, jis yra pašalinamas iš populiacijos. Tokį sprendimą galima argumentuoti atsižvelgiant į tai kad ieškomi ne tik korektišką atsakymą gaunantys algoritmai, bet ir tie algoritmai, kurie veikia greitai. Tikėtina, kad net jei ir daug perteklinio kodo turintis individas vis dar sugeba gauti korektiška atsakymą, nukenčia jo darbo sparta, todėl tokio individo pašalinimas tik pagerina visos populiacijos potencialą. Pašalinus individą populiacija sumažėja, bet kaip ir buvo minėta ankstesniuose skyriuose, populiacijos augimas ir kritimas yra dažnas reiškinys šiame modifikuotame genetiniame algoritme. Populiacija niekada nenukris žemiau tam tikro, pasirinkto, rėžio, nes suveiks naujų individų generacijos operatorius. Taigi, užkirtus kelią šiai problemai, išvengiama potencialaus sukurto įrankio sulėtėjimo, kompiuteriniai resursai yra valdomi efektyviau.

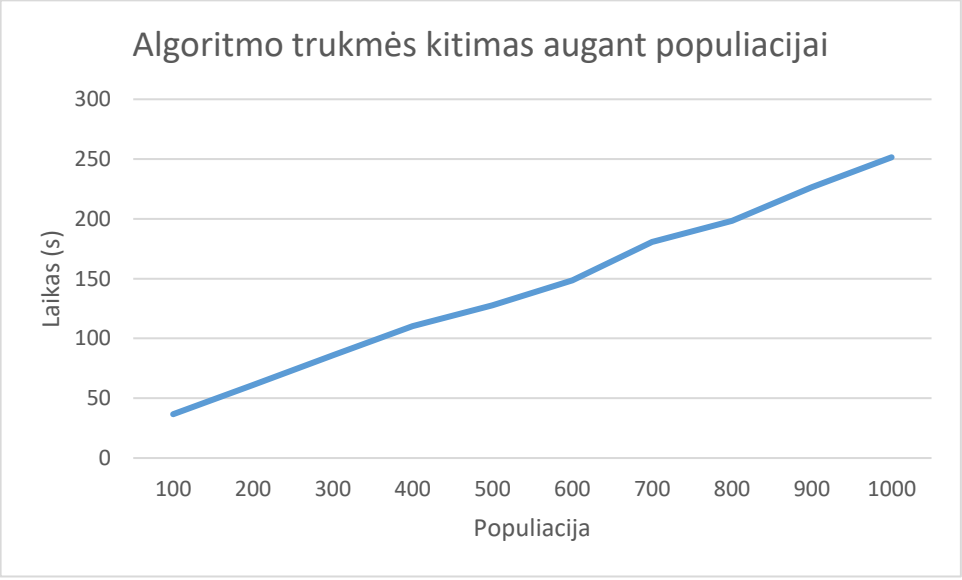
2.1.4.2. Aplinkos parametrai

Kuriamo karkaso sparta priklauso nuo tam tikrų parametrų. Didžiausią įtaką turintys yra populiacijos dydis bei genetinio algoritmo kartų skaičius. Kuo didesnė individų populiacija ir kuo daugiau iteracijų atliekama, tuo daugiau skaičiavimų yra įvykdoma. Spartai įtakos turi ir kiti, mažiau susiję parametrai, kaip atrankos turnyro dydis, migracijos turnyro dydis, migruojančių individų skaičius, migracijos tikimybė, mutacijos tikimybė, salų skaičius bei analogiški parametrai, susiję su genetiniu tobulėjimu. Dalis iš šių parametrų turi aiškiai apibrėžtas maksimalias reikšmes, pavyzdžiui mutacijos tikimybė, todėl optimizacijos metu bus išbandomos beveik visos šių parametrų reikšmės. Deja, dalis parametrų maksimalios reikšmės neturi ir gali augti link begalybės. Tokios didelės aibės negalima ištirti per baigtini laiką, todėl reikia šiuos parametrus apriboti. Pasirinktos maksimalios begalinių parametrų reikšmės yra pateiktos 1-oje lentelėje. Tai yra maksimalus rėžis, kuris gali būti pasiekiamas karkaso darbo metu. Optimizuojant, tikėtina, kad didelė dalis galimų reikšmių bus išbandytos, tačiau ne pilnai, bet tai yra patenkinamas rezultatas.

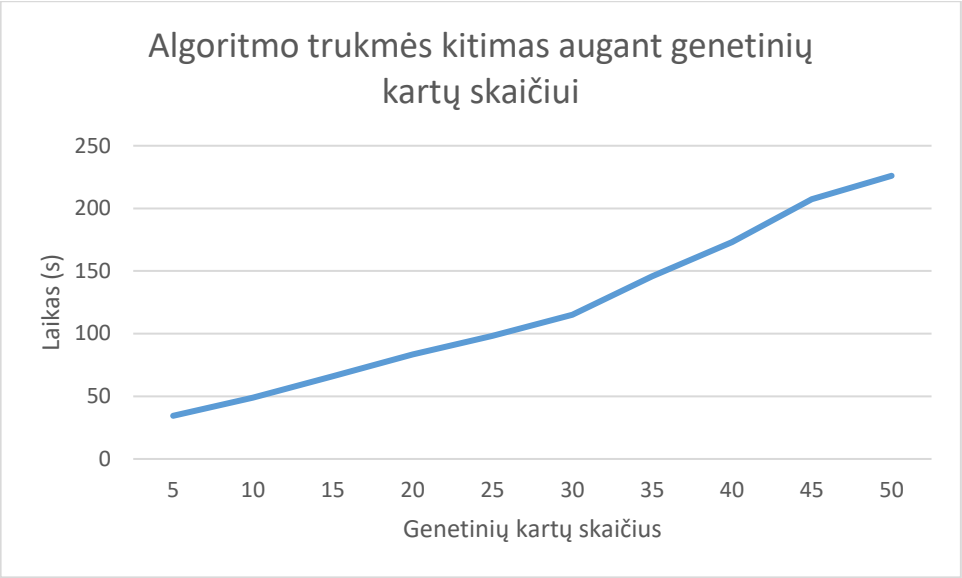
1 lentelė. *Maksimalios begalinių parametrų reikšmės*

Parametro pavadinimas	**Maksimali reikšmė**
Vidutinis atrankos operatoriaus turnyro dydis	10
Populiacijos dydis	10000
Maksimalus algoritmo iteracijų skaičius.	50
Salų skaičius	10
Genetinio tobulėjimo populiacijos dydis.	1000
Genetinio tobulėjimo maksimalus iteracijų skaičius	20

Kaip jau buvo minėta, populiacijos dydis ir iteracijų skaičius yra įtakingiausi parametrai. Didinant jų reikšmes viso karkaso darbo laikas sparčiai išauga. Nėra visiškai aišku koks parametrų sąryšis tarpusavyje. Galbūt vieną iš jų galima didinti žymiai sparčiau. Tam įvertinti matuojamas karkaso darbo laikas. 32-ame paveikslėlyje pateikiamas darbo laiko augimas didinant tik individų populiaciją, o 33-ame paveikslėlyje analogiškas eksperimentas didinant tik algoritmo iteracijų skaičių, t.y. genetinių kartų skaičių. Paveikslėliuose abscisių ašyje žymimas populiacijos dydis arba genetinių kartų skaičius, o ordinačių ašyje pateikta trukmė, žymima sekundėmis. Palaipsniui didinant parametrus, užimto laiko trukmė auga tiesiškai.

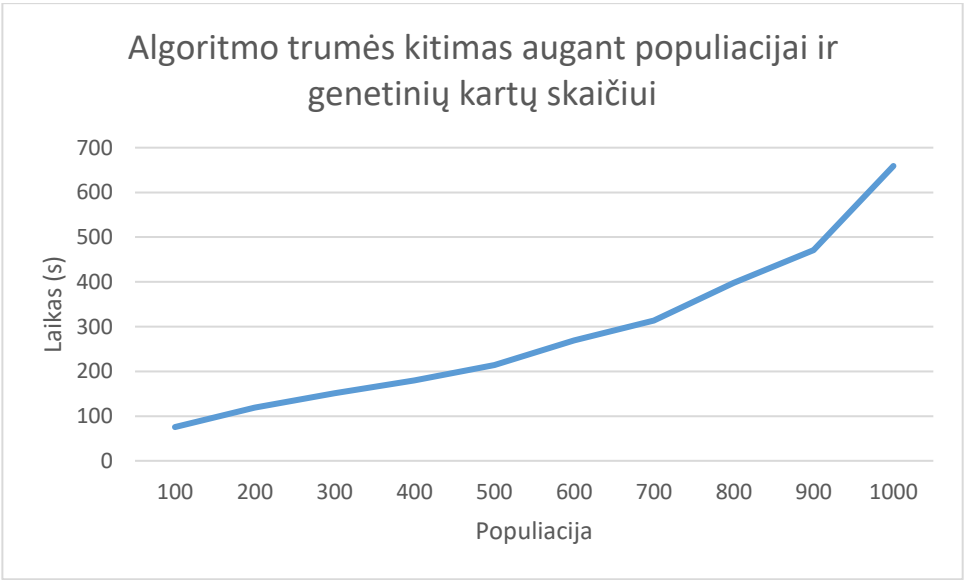


32 pav. *Algoritmo trukmės kitimas augant populiacijai*



33 pav. *Algoritmo trukmės kitimas augant genetinių kartų skaičiui*

Eksperimentuojant pastebėta, kad individų populiaciją galima didinti gana sparčiai. Galima teigti, kad efektyviausia didinti individų populiacijos skaičių, o iteracijų skaičių palikti sąlyginai mažą. Didinant ir populiaciją, ir iteracijų skaičių, algoritmas reikalauja eksponentiškai daugiau laiko, tą pavaizduojantis eksperimento rezultatas pateiktas 34-ame paveikslėlyje. Paveikslėlyje abscisių ašyje pateiktas populiacijos dydis, o ordinačių ašyje vaizduojama trukmė sekundėmis. Galima daryti išvadą, kad optimaliausiam resursų išnaudojimui svarbu didinti populiaciją, o iteracijų skaičių išlaikyti pakankamai mažą ir pastovų.



34 pav. *Algoritmo trukmės kitimas augant populiacijai ir genetinių kartų skaičiui*

2.1.4.3. Kompiuteriniai resursai

Programinis karkasas buvo sukurtas ir testuojamas ant asmeninio kompiuterio. Šio kompiuterio techninė specifikacija pateikta 2-oje lentelėje. Kompiuteris yra pakankamai galingas eksperimentams atlikti. Optimaliai išnaudojamas galingas procesorius, o operatyviosios atminties pakanka didelėms individų populiacijoms tirti. Išnaudojamas paralelizavimo aspektas, leidžiantis karkaso darbo metu atlikti kitas užduotis ir stebėti progresą. Kompiuterio resursų pakanka eksperimentams atlikti. Didinant karkaso parametrus, auga sunaudojamo laiko kiekis, bet jis išlieka pakankamai žemas reikalingiems rezultatams gauti.

2 lentelė. *Kompiuterio specifikacija*

Kompiuterio specifikacija	
Operacinė sistema	Windows 10 Home 64-bit
Procesorius	Intel(R) Core(TM) i7-3520M CPU @ 2.90GHz (4 CPUs), ~2.9GHz
Operatyvioji atmintis	8192MB
Atminties disko tipas	SSD

2.2. C# gramatikos ypatumai

Kaip jau buvo minėta skyriuose 1.2.2, 2.1.2 bei 2.1.3, C# programavimo kalbos gramatikos perkėlimas į programinį pavidalą yra sudėtingas. Kiekvienas loginis elementas turi būti užkoduotas atitinkama klase. Tiesa, šio darbo kontekste, to neužtenka, nes vykstant kodo generacijai, gramatikos elementai turi sugebėti atsitiktinai generuoti save, pasirenkant vieną iš galimų išsišakojimo variantų. Kaip jau buvo minėta 1.3.4 skirsnyje, C# programavimo kalbos atsitiktinis skaičių generatorius yra ribotas ir tinkamiausiai veikia, kai programoje egzistuoja tik vienas jo egzempliorius. Tai reiškia, kad kuriant ir generuojant kiekvieną gramatikos elementą, turi būti perduodama nuoroda į šį atsitiktinių skaičių generatoriaus vienetą, ko pasėkoje pasunkinamas greitas gramatikos elementų generavimas. Nepaisant to, veiksmas yra privalomas ir jo negalima nevykdyti.

Kiekviena gramatikos taisyklės klasė turi gana panašią struktūrą ir susideda iš šių esminių elementų:

- Terminalinių simbolių sąrašas. Tai yra nekintami eilutės tipo laukai kodo atžvilgiu.
- Neterminalinių simbolių sąrašas. Kodo atžvilgiu, tai nuoroda į kitą klasę, atvaizduojančią gramatikos taisyklę.
- Visų įmanomų išsišakojimų sąrašas. Kuriant kodą, vienas elementas gali įgauti daug reikšmių. Pavyzdžiui, norint sukurti naują eilutės objektą, jo išsišakojimai gali būti kodo segmentas, naujo kintamojo sukūrimas, veiksmas su masyvu ir daug daugiau. Šie išsišakojimai taip pat šakojasi ir gali sugrįžti atgal link naujos eilutės kūrimo, todėl įmanomas begalinis ciklas. Tai buvo vienas iš C# gramatikos ypatumų, dėl kurių reikėjo atlikti papildomų kodo modifikacijų. Generuojant kodą, šakojimasis turi vykti su tam tikra, pakankamai nedidele tikimybe. Tokiu atveju procesas bus ne tik baigtinis, bet ir neužtruks pernelyg ilgai. Taigi, šis išsišakojimų sąrašas kode yra sudaromas tiesiog paprasto sąrašo atžvilgiu, kuris užpildomas, sukuriant gramatikos taisyklės klasės objektą.
- Galutinio programinio kodo generavimo metodas. Kiekviena gramatikos taisyklių klasė savo ruožtu įgyvendina bazinę klasę, kuri joms nurodo pateikti generacijos metodo įgyvendinimą. Tai užtikrina, kad visos sukurtos taisyklės bus generuojamos. Generacijos metodas priklauso nuo taisyklės sudėtingumo ir susideda iš šių esminių žingsnių:
    - Išsišakojimo pasirinkimas. Atsitiktinai iš sąrašo pasirenkamas vienas išsišakojimo kelias.
    - Neterminalinių simbolių, naudojamų išsišakojime, generavimas. Kiekvienas neterminalinio simbolio generavimas atitinka viso išvardinto proceso vykdymą iš naujo, naujos klasės kontekste.
    - Rezultato suformavimas ir grąžinimas.

Dar vienas ypatumas, kurį verta paminėti, tai raktinių žodžių bei konstantų saugojimas. Analogiškai gramatikos taisyklėms, C# programavimo kalbos raktiniai žodžiai ir konstantos taip pat turi būti pasiekiamos, kaip objektai. Generuojant kodą, dažnai prireikia sukurti naują skaitinę reikšmę arba naują identifikatoriaus pavadinimą. Šio proceso metu atsitiktinai generuojamos reikšmės, kurios turi būti kaskart tikrinamos su raktinių žodžių sąrašu. Tai yra esminis etapas, kurio praleisti negalima. Tikimybė, kad atsitiktinai sugeneruotas identifikatorius atitiks raktinį programavimo kalbos žodį yra labai maža, tačiau tokio įvykio neigiamos pasekmės yra didelės. Sukūrus netinkamą identifikatorių, programinis kodas tampa netaisyklingu ir kompiliacijos procesas pateiks programinę klaidą. Šios rizikos sumažinimui racionalizuojama papildoma, sąlyginai daug laiko užimanti iteracija per visus C# programavimo kalbos raktinius žodžius.

2.3. Išvystyti algoritmai

Šiame skyriuje pristatomi visi tradiciniai informatikos mokslų algoritmai, kurie buvo išvystyti sukurto karkaso pagalba. Buvo siekta įrodyti, kad teoriniai sprendimai yra teisingi, o programinis karkaso įgyvendinimas veikia ir duoda tam tikrus rezultatus. Pagrindinė karkaso esmė yra leisti vartotojui nustatyti tik pradinius duomenis ir galutinį tikslą, o visas kitas darbas, tai programavimas, algoritmo kūrimas ir įvertinimas yra atliekami automatiškai. Kiekvienas iš išvystytų algoritmų yra trumpai pristatomas, aptariamas literatūroje prieinamas programinis kodas bei išdėstomi aspektai, į kuriuos reikėjo atsižvelgti karkaso kūrimo metu. Papildomai, pateikiama algoritmo grafinė evoliucijos proceso interpretacija. Svarbu pabrėžti, kad visada tiriamas vidutinis genetinės populiacijos individas, todėl grafikuose egzistuoja staigūs šuoliai žemyn. Tai nėra išskirtiniai atvejai, o tiesiog genetinio algoritmo darbo aspektas. Taigi, pristatomi keli esminiai, nesudėtingi algoritmai.

Karkaso darbo metu, generuojant naujas kodo eilutes pagal programavimo kalbos taisykles, naudojamas svorinis atsitiktinumas. Tai reiškia, kad renkantis vieną iš kelių galimų variantų, jų pasirinkimo tikimybės nėra lygios. Pavyzdžiui, generuojant eilutę, tikimybė, kad tai bus ***for*** ciklas, arba priskyrimo sakinys, yra skirtingos. Toks svorinis atsitiktinumas reikalingas tam, kad būtų eliminuoti begaliniai programinio kodo generavimo atvejai, nes daugelis programavimo kalbos gramatinių taisyklių yra tarpusavyje susijusios ir sukuria ciklus. Kuriant sakinio eilutę ir parenkant vieną iš galimų reikšmių, ši reikšmė gali sukurti daugiau nei vieną naują reikšmę, o jos savo ruožtu gali vėl pasiekti paprasto sakinio kūrimo atvejį. Pavyzdžiui, sukuriant ciklą ***for***, jo viduje gali būti begalinis skaičius naujų eilučių. Šiose eilutėse vėl gali atsirasti naujas ciklas ***for*** ir procesas kartojasi. Šiai problemai spręsti buvo įgyvendinti svoriai. Generuojant programinį kodą, tam tikrais atvejais sekamas išsišakojimo gylis. Priklausomai nuo gylio, sakinių, kurie gali plačiai šakotis, tikimybės yra mažinamos. Nekeičiamos tik baigtinių sakinių, kurie neturi išsišakojimų, tikimybės. Kiekvienai šakai suteikiamas tam tikras skaitinis svoris. Šis svoris yra dalinamas iš išsišakojimo gylio, taip sparčiai mažinant tikimybę, kad šaka bus pasirinkta. Toks svorinis atsitiktinumo įgyvendinimas leidžia individams plėstis, bet su vis mažėjančia tikimybe, iki kol begalinis šakojimasis tampa praktiškai neįmanomas. Svorinis atsitiktinumas taip pat leidžia padėti genetiniam algoritmui išvystyti tam tikrą algoritmą, keičiant programinio kodo elementų sukūrimo tikimybes. Karkaso darbo metu, algoritmas, kuris bus sukurtas, nėra aiškus, tačiau įmanoma sukurti sąlygas, kurios yra palankios konkrečių algoritmų atsiradimui. Keičiant kuriamų individų dydį, programinio kodo elementų tikimybes, bei genetinio algoritmo parametrus, karkasas yra pakreipiamas link norimo algoritmo. Žinoma šis papildomas žingsnis yra reikalingas tik

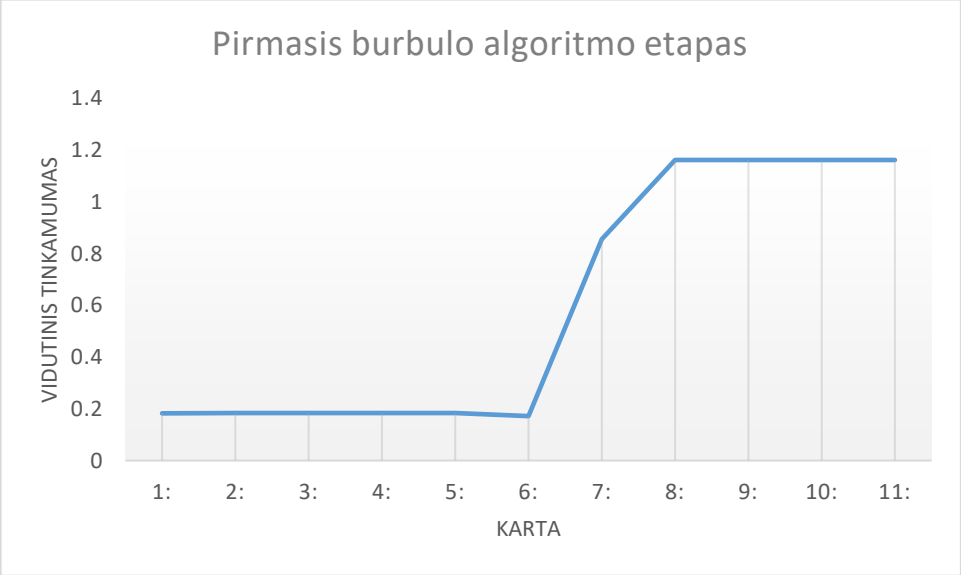
eksperimentams atlikti, norint įsitikinti, kad vienas ar kitas algoritmas tikrai gali būti sukuriamas atsitiktinio kodo generavimo procesu. Taigi, svorinis atsitiktinumas leidžia ne tik išvengti problemų, bet ir palengvinti sukurto karkaso įvertinimą.

Visų eksperimentų metu darbas vyksta atsižvelgiant į 2.1.4 skirsnyje aptartas problemas ir jų sprendimus. Atliekant tyrimus, didelę įtaką turintiems parametrams priskiriama iš anksto apsibrėžta maksimali reikšmė. Šios reikšmės yra matomos 1-oje lentelėje. Eksperimentuojant, genetinis algoritmas kuria 10000 individų populiaciją ir ją vysto iki kol pasiekiama 50 genetinių kartų. Burbulo algoritmo atveju, pirmieji eksperimentai atlikti su mažesniais parametrais, nes sukurtas karkasas dar nebuvo pilnai optimizuotas, apie ką detaliau kalbama sekančiame skirsnyje. Darbo metu, pradiniai duomenys visais atvejais yra vienodi ir susideda iš sveikų skaičių masyvo, kurį galima nurodyti vartojo sąsajos pagalba. Rezultatai, kurie išdėstyti šiame poskyryje, yra gauti naudojant skirtingus sveikų skaičių masyvus iki šimto elementų. Darbo metu tirti du skirtingi uždaviniai, tai paieška ir rikiavimas. Paieškos atveju, kontekstinėje informacijoje taip pat nurodomas skaičius, kurio ieškoma. Rezultatas, kurį norima gauti, yra taip pat nustatomas atsižvelgiant į užduotį. Rikiavimo atveju, tai duomenų masyvas, kuris išrikiuotas norima tvarka, o paieškos atveju, tai yra vienas skaičius atspindintis indeksą. Karkasas sukurtas atsižvelgiant į tai, kad bus sprendžiami ne tik paieškos ar rikiavimo uždaviniai. Įrankis sugeba grąžinti sveiką skaičių, o kaip šis skaičius yra interpretuojamas priklauso nuo pradinio programos vieneto aprašymo, apie ką detaliai kalbėta 1.3.1 skirsnyje. Kodas, kuris, priklausomai nuo užduoties, įvertina algoritmo darbą, yra įterpiamas į pradinį vienetą. Pavyzdžiui, rikiavimo užduoties atveju, įterpiamas programinis kodas, kuris įvertina pradinių duomenų masyvo atitikimą rezultatui, o gražinama reikšmė atspindi rezultato atitikimo lygį. Paieškos atveju, įterpiamas kodas, kuris patikrina ar algoritmo grąžinama reikšmė yra lygi norimai ir ją gražina. Atsakymo patikrinimas, kaip ir buvo minėta ankstesniuose skyriuose, yra tinkamumo funkcijos pritaikymo užduočiai dalis, kurią reikia atlikti jeigu tai yra reikalinga arba neįmanoma norimo funkcionalumo gauti su jau egzistuojančiu sprendimu.

2.3.1. Burbulo algoritmas

Šio tyrimo pradžioje, 1.2.2 skirsnyje, buvo pristatytas teorinis burbulo rikiavimo algoritmo išvedimas. Tam, kad būtų patikrintas sukurto karkaso funkcionalumas, verta tą patį tyrimą atkartoti gaunant realius rezultatus. Į burbulo algoritmo vystymą buvo pažvelgta detaliau, aptariant vystymo proceso eigą ir kilusias kliūtis bei jų sprendimus.

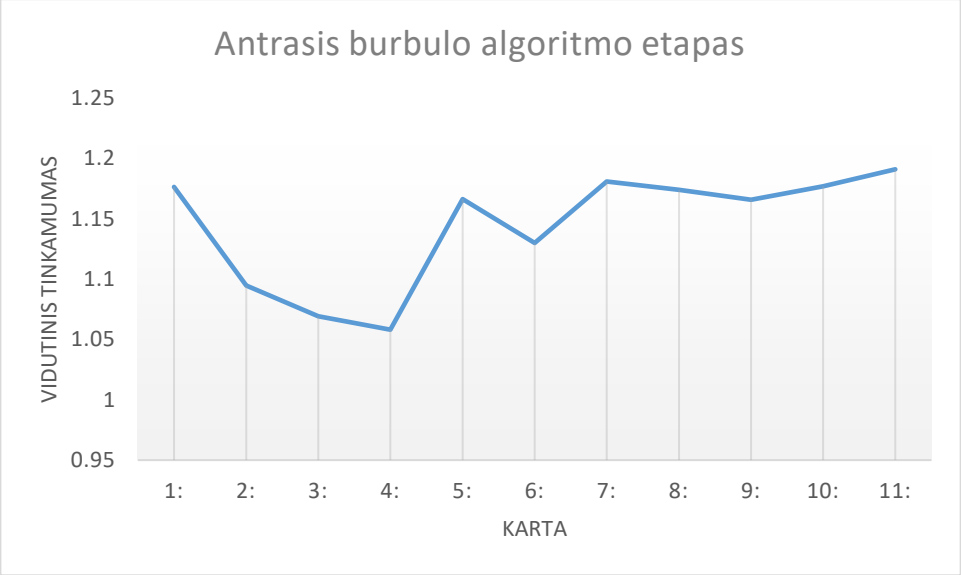
Tyrimas vyko keliais etapais, palaipsniui pildant programinį kodą ir plečiant karkaso galimybes. Pirmiausia, buvo sukurtas tik trijų esminių algoritmo eilučių išvystymas, kurios atlieka duomenų masyvo elemento apkeitimo funkciją. Eilutės yra generuojamos C# programavimo kalbos gramatikos taisyklėmis, pateikiant kontekstą, todėl galimų identifikatorių reikšmės, šiuo atveju, masyvo identifikatorius, laikinas kintamasis bei masyvo skaitliukas yra iš anksto žinomos. Vidutinis tinkamumo kitimas, bėgant laikui, pateiktas 35-ame paveikslėlyje. Abscisių ašyje žymimos algoritmo kartos, o ordinačių ašyje pateikiamas vienos iteracijos vidutinis individų tinkamumo įvertis, apskaičiuotas pagal 1.3.2.4 straipsnyje aprašyta tinkamumo funkciją. Galima pastebėti, kad pirmosiomis genetinio algoritmo iteracijomis kitimas yra labai nežymus. Po tam tikro laiko, mutacijų ir kryžminimo pagalba išvystomas individas, kuris sugeba teisingai išspręsti pateiktą užduotį ir tai žymiai padidina jo tinkamumo lygį. Šiam individui maišantis su likusiais populiacijos individais, vidutinis tinkamumo lygis kiekvienos kartos metu kyla, kol pasiekiama viršūnė. Viršūnėje algoritmas arba pasiekia globalų optimumo tašką, arba supanašėja, t.y. patenka į lokalų optimumo tašką. Kadangi šis etapas yra pakankamai paprastas, labiausiai tikėtinas pirmasis variantas, todėl ties šiuo tyrimu neapsistojama.



35 pav. *Pirmojo burbulo algoritmo etapo grafikas*

Antrame etape, buvo atsitiktinai generuojamos ne tik trys pagrindinės eilutės, bet ir loginio išsišakojimo sąlygos parametrai. Tokiu būdu atsitiktinio kodo generavimas padidinamas viena eilute. Išlaikoma loginio išsišakojimo struktūra, nekeičiant esminių programos kodo reikšmių, o tik kintamuosius. 36-ame paveikslėlyje pateikiamas vidutinio tinkamumo lygio kitimas, bėgant laikui. Abscisių ašyje žymimos algoritmo genetinės kartos, o ordinačių ašyje pateikiamas vienos iteracijos vidutinis individų tinkamumo įvertis, apskaičiuotas pagal 1.3.2.4 straipsnyje aprašyta tinkamumo funkciją. Šio tyrimo atveju, tiriamų individų populiacijos dydis padidintas nuo šimto

iki dviejų šimtų individų, todėl net ir pirmosios iteracijos metu atsiranda individas, sugebantis teisingai išspręsti uždavinį. Generuojamo kodo yra sąlyginai mažai, todėl, dėl mutacijų, jis lengvai nukrypsta ir vidutinis kartos tinkamumas tam tikrais etapais krenta. Taip pat, pirmasis šuolis žemyn yra labai dažnas darbo su genetiniu algoritmu metu. Tik pradėjus darbą, sugeneruoti individai yra labai įvairūs. Pradinėje populiacijoje gali egzistuoti keletas labai aukštą tinkamumo lygį turinčių individų, kas kelia vidutinį tos iteracijos tinkamumo lygį. Tačiau, pereinant į sekančias, vis dar pradines iteracijas, šie keli išskirtiniai individai susimaišo su likusiais populiacijos individais, perduodant savo gerus genus. Dėl to matomas tinkamumo kritimas, nes, nors geri genai egzistuoja, jie dar nėra tinkamai išnaudojami. Laikui bėgant, populiacijos tinkamumas palaipsniui auga, individai su gerais genais išgyvena ir perduoda savo informaciją naujiems individams. Populiacija po truputį panašėja, atsiranda vis daugiau individų su naujais genais, o tada atsiranda ir tinkamumo šuoliai, kai sukuriamas dar geresnis individas, besiremiantis išgyvenusiais naudingais genais.



36 pav. *Antrojo burbulo algoritmo etapo grafikas*

Galiausiai, pasirinkta atsitiktinai generuoti visus įmanomus burbulo paieškos metodo parametrus, išlaikant ciklo bei loginio išsišakojimo struktūrą. Dėl burbulo algoritmo paprastumo, šis galutinis eksperimentas pasunkėja nežymiai, tačiau po kelių pakartotinių bandymų, pavyko gauti vieną iš geriausių vidutinio tinkamumo įvertinimų. Šio eksperimento grafikas pateiktas 37-ame paveikslėlyje. Abscisių ašyje žymimos algoritmo genetinės kartos, o ordinačių ašyje pateikiamas vienos iteracijos vidutinis individų tinkamumo įvertis, apskaičiuotas pagal 1.3.2.4 straipsnyje aprašytą tinkamumo funkciją. Verta pabrėžti, kad tyrimo metu nebuvo gautas geriausias įmanomas tinkamumo įvertinimas, kuris gaunamas vertinant literatūroje aprašytą optimalų burbulo algoritmą. Geriausias įmanomas įvertis lygus **1,26**, o geriausias pasiektas **1,22**.

Tai gauta, apskaičiavus individų tinkamumo įvertinimą, pagal 1.3.2.4 straipsnyje išdėstytą procesą. Literatūroje pripažintas burbulo rikiavimo algoritmo kodas, kuris buvo vertintas, matomas 2-ame paveikslėlyje, o geriausias išvystytas individas prieš karkaso optimizaciją, pateiktas 38-ame paveikslėlyje.

Analizuojant geriausio individo kodą, matoma, kad dauguma kodo eilučių buvo sukurtos kodo bloko pavidalu, nes jos prasideda ir baigiasi figūriniais skliaustais. Tai galima paaiškinti, atsižvelgus į tai, kad supaprastintoje C# kalbos gramatikoje, paprastos eilutės ir kodo bloko sakinių panaudojimo tikimybės yra vienodos, kas neatitinka realaus programavimo, bet nedaro didelės neigiamos įtakos, todėl eksperimento dėlei tikimybės nebuvo keičiamos. Kodo blokas iš vienos eilutės yra ekvivalentus paprastai kodo eilutei. Taip pat matyti, kad kodo eilutės

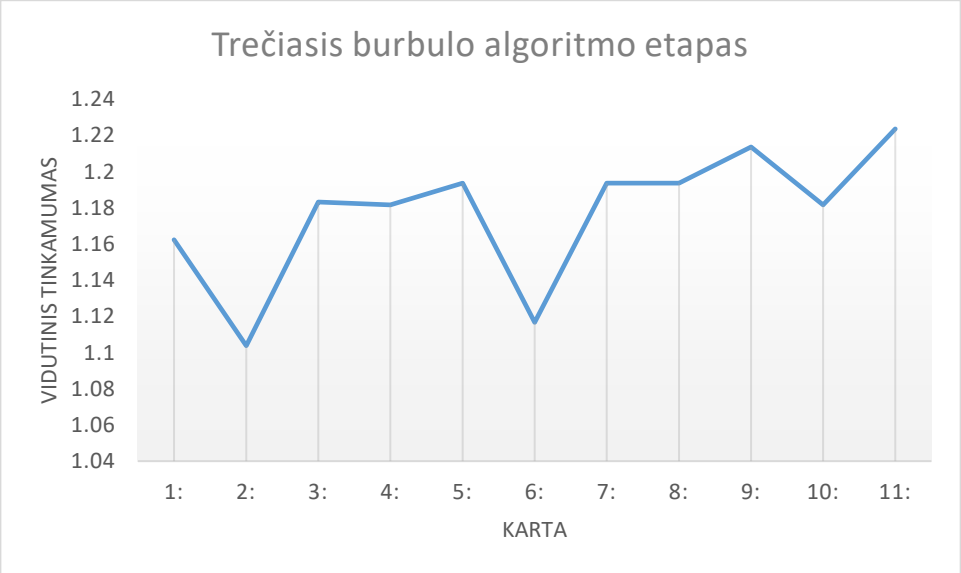
{ (write)++; };

--write;

viena kitą anuliuoja. Tikėtina, kad jos nebuvo pašalintos dėl priklausomybės vienai nuo kitos, nes naujas individas, kuriame būtų tik viena iš šių eilučių, neišspręstų užduoties. Taip pat, kodo eilutė

{ temp = (temp – 1); };

neatlieka jokio naudingo veiksmo užduoties kontekste. Tikėtina, kad genetinio tobulinimo metu, nebuvo sukurtas individas, kuris gavo geresnį įvertinimą neturėdamas šios eilutės. Iš to galima daryti išvadą, kad karkaso parametrų optimizavimo metu reikia pasirinkti pakankamai didelį genetinio tobulėjimo populiacijos dydį, dėl ko bus sugeneruojami įvairesni individų variantai.



37 pav. *Trečiojo burbulo algoritmo etapo grafikas*

Grįžtant prie 37-o paveikslėlio, kaip ir ankstesniais atvejais, pastebimas tinkamumo šuolis darbo pradžioje. Taip pat, žymus tinkamumo šuolis žemyn šeštoje kartoje. Šuolį žemyn galima paaiškinti genetinio algoritmo atsitiktinumu. Galbūt šioje kartoje didesnė dalis naudingų individų buvo sudarkyti, o naujoje kartoje atsirado individų, kurie uždavinį sprendė neteisingai, kas labai žymiai sumažino tinkamumo įvertinimą. Apie tinkamumo vertinimą, priklausomai nuo korektiško uždavinio sprendimo, buvo kalbėta 1.3.2.4 straipsnyje.

Sukurto karkaso pagalba buvo sėkmingai gautas burbulo rikiavimo algoritmo kodas, palaipsniui tobulinant pritaikomas C# programavimo kalbos gramatikos taisykles. Kodas sėkmingai rikiuoja jam pateiktą duomenų masyvą ir gauna visiškai korektišką atsakymą. Geriausio gauto individo įvertinimas nėra optimalus dėl perteklinių kodo eilučių, bet tai ne esminis trūkumas.

```
int temp = 0;

for (int write = +0; write < arr.Length; write++)
{
    {
        for (int sort = 0; sort < (arr.Length - 1); sort++)
        {
            if (arr[(sort - 1 + 1)] > arr[sort + 1])
            {
                { (write)++; };
                { temp = (temp - 1); };
                temp = arr[sort + 1];
                --write;
                arr[sort + 1] = arr[sort + 0];
                { arr[sort] = temp; ; };
            }
        }
    }
}
```

38 pav. *Geriausio individo kodas prieš karkaso optimizaciją*

Atlikus pilną sukurto karkaso optimizaciją, burbulo algoritmo kodo defektai buvo pašalinti. Optimizuoto karkaso darbo metu, eilutės su pertekliniais kodo bloko simboliais buvo pilnai panaikintos. Tai atlikta sugriežtinus naudojamas gramatikos taisykles. Taip pat, pašalinta galimybė sukurti tuščią kodo eilutę, nes ji yra visiškai nereikalinga algoritmo veikimui, tokios generuojamo kodo šakos egzistavimas tik lėtina karkaso darbą. Eilutės anuliuojančios viena kitą pranyko, padidinus karkaso parametrus. Esant daugiau individų ir ilgesniam vystymo procesui, eilutės nedarančios įtakos algoritmo darbui palaipsniui pranyksta, nes individai be jų gauna geresnį tinkamumo įvertinimą. Tai ypač pasireiškia genetinio tobulėjimo metu. Eksperimentuojant, programinis kodas, kuris visiškai atitinka literatūroje sutinkamą burbulo algoritmą, buvo gautas ne iš karto. Kaip ir tradicinių uždavinių atveju, genetinio algoritmo rezultatas stipriai priklauso nuo pradinių populiacijos genų sąrašo. Pakartotinis karkaso darbo veikimas konkrečiam uždaviniui yra dažnas procesas, galbūt net privalomas, norint įsitikinti, kad tiriama uždavinio aibė ištirta maksimaliai pilnai. Nepaisant to, vienu iš eksperimentų buvo gautas geriausias burbulo

algoritmo programinis kodas, turintis aukščiausią tinkamumo įvertinimą, **1,26**. Galima daryti išvadą, kad optimizacijos procesas padėjo atsikratyti kodo defektų.

2.3.2. Tiesinės paieškos algoritmas

Tiesinės paieškos algoritmas yra vienas iš paprasčiausių ir primityviausių algoritmų. Nepaisant to, jis yra gana dažnai naudojamas ten, kur sparta nėra aktuali ar ne toks svarbus programinio kodo efektyvumas, arba kodas rašomas nepatyrusio programuotojo, nes algoritmo programinis kodas yra labai paprastas. Tiesinės paieškos algoritmo metu duomenų masyvu yra keliaujama nuosekliai, pereinant kiekvieną elementą ir tikrinant ar jis yra ieškomasis. Algoritmas darbo pabaigoje gražina indeksą, kuris nurodo kurioje duomenų aibės vietoje yra tikslo elementas. Darbas baigiamas iš karto radus atsakymą, t.y., duomenų masyvas nėra tikrinamas iki pabaigos, dėl to įmanomi atvejai, kai algoritmas darbą baigia labai greitai, o kitais atvejais labai lėtai – tik pasiekus paskutinįjį elementą. Tiesinės paieškos algoritmas nėra sudėtingas, bet jo išvystymas nuo nulio yra pakankamai įdomi užduotis, kuri buvo įvykdyta šio darbo metu.

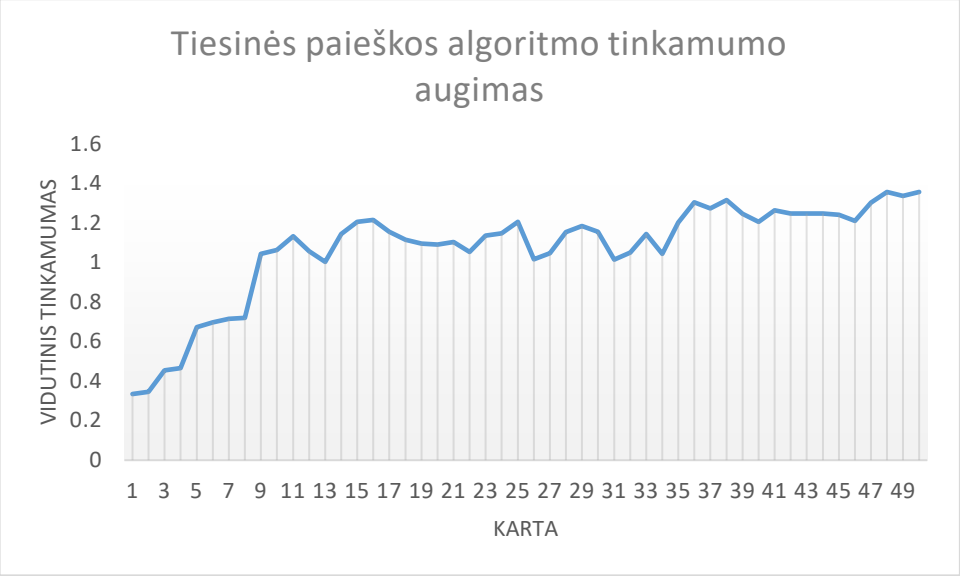
Teoriškai, šis algoritmas mažai kuo skiriasi nuo paprastų maksimalios ar minimalios reikšmės paieškos algoritmų. Šių algoritmų atveju, visada einama per visą masyvą, o elementų reikšmės lyginamos tarpusavyje. Vienintelis skirtumas yra tai, kad tiesinės paieškos atveju, kaip ir buvo minėta, darbas baigiamas iškart radus atsakymą. Maksimalios ar minimalios reikšmės paieškos metu duomenų aibė kiekvienu atveju yra peržvelgiama iki galo. Atsakymas taip pat truputi skiriasi, vietoje indekso yra gražinama tam tikro elemento reikšmė. Nepaisant to, programinio kodo prasme, skirtumas tarp šių algoritmų yra minimalus, todėl jie nėra vystomi šio darbo metu.

Tiesinės paieškos algoritmo kodas yra laisvai prieinamas viešoje literatūroje. Tradicinis įgyvendinimas C# programavimo kalba yra pateiktas 39-ame paveikslėlyje. Nesunku pastebėti, kad tai yra tiesiog perėjimas per masyvą su palyginimu. Paveikslėlyje ***x*** žymi tikslą, elementą, kurio ieškoma. Algoritmas susideda iš vieno ciklo. Rezultatas yra sveikasis skaičius, žymintis elemento indeksą masyve. Jeigu elementas, kurio ieškoma, masyve neegzistuoja, algoritmas grąžina neigiamą reikšmę. Tai yra būtinas veiksmas, norint greitai identifikuoti problemą. Algoritmo programinis kodas yra pakankamai paprastas, kas leidžia tikėtis greito algoritmo išvystymo karkaso darbo metu.

```
public static int LinearSearch(int[] arr, int x)
{
    for (int i = 0; i < arr.Length; i++)
    {
        if (arr[i] == x)
            return i;
    }

    return -1;
}
```

39 pav. *Tradicinis tiesinės paieškos programinis kodas C# kalba*

Programuojant karkasą reikėjo atsižvelgti ir į šio paprasto algoritmo aspektus. Pirmiausia, ciklo sukūrimas, kuris yra pagrindinė algoritmo dalis, jau buvo sėkmingai įgyvendintas burbulo algoritmo kūrimo metu, todėl į tai papildomai gilintis nereikėjo. Analogiškai anksčiau jau buvo apžvelgtas ir loginio išsišakojimo sakinys. Šiuo atveju, prisidėjo papildoma loginio išsišakojimo sąlyga, elementų palyginimas, ko nereikėjo rikiavimo metu. Sudėtingiausia šio algoritmo dalis, atliekant algoritmo kūrimo procesą, buvo kontrolės šuolio sakinys ***return***. Rikiavimo metu galutinis rezultatas buvo duomenų masyvas, kuris jau gautas kaip pradiniai duomenys, todėl nereikėjo papildomo rezultato grąžinimo. Šiuo atveju, algoritmas privalo grąžinti indeksą, pagal kurį nustatoma norimo elemento vieta arba neigiamas skaičius, pranešantis apie nesėkmę. Darbo metu turėjo būti apžvelgtos C# programavimo kalbos kontrolės šuolių taisyklės. Daugumą šių taisyklių nėra aktualios algoritmų vystymo metu, pavyzdžiui taisyklės, aprašančios programavimo kalbos kritinių klaidų apdorojimą su raktiniu žodžiu ***throw*** arba kontrolinius šuolius su retai naudojamu operatoriumi ***goto***. Todėl į supaprastintą gramatiką, kuri naudojama karkaso viduje, tokios taisyklės nebuvo įtrauktos, taip sumažinant tiriamąją aibę ir atmetant nereikalingus sprendinius. Taigi, atsižvelgiant į minėtus aspektus ir atnaujinus kuriamą karkasą, pereita prie praktinio algoritmo vystymo, nes teorinis kelias nuo tuščios programos link tiesinės paieškos algoritmo yra įmanomas.



40 pav. *Tiesinės paieškos algoritmo tinkamumo augimas*

Tiesinės paieškos algoritmo tinkamumo augimas karkaso darbo metu pateiktas 40-ame paveikslėlyje. Algoritmo vystymo pradžioje iš karto sugeneruojama bent keletas pakankamai gerų individų, nes šio algoritmo kodas yra nesudėtingas. Tačiau, kadangi didelė dalis kitų algoritmų yra blogi, vidutinis tinkamumas karkaso darbo pradžioje yra mažas. Bėgant laikui, vidutinis individų populiacijos tinkamumas greitai išauga, nes blogi individai neišgyvena, o geri išlieka ir toliau perduoda savo genus. Jau po dešimties genetinio algoritmo kartų matyti nusistovėjimas, nebėra staigių tinkamumo šuolių aukštyn. Tai įvyksta, nes geri individai turi sąlyginai trumpą programinį kodą, kurio pagerinti neišeina. Galiausiai, artėjant link pabaigos, vienoje iš genetinių kartų sukuriamas idealus individas, kurio programinis kodas pilnai atitinka tiesinės paieškos algoritmą. Nors geriausias individas buvo gautas jau ir ankstesnėje kartoje, karkaso darbas buvo tęsiamas iki galo. Gautas rezultatas panašus į tradicinį genetinio algoritmo darbą kituose uždaviniuose, kai rezultatas palaipsniui išvystomas, nereikalaujant pernelyg daug iteracijų. Galima teigti, kad to pasekmė yra užduoties paprastumas. Taigi, eksperimentuojant, pavyko išvystyti tiesinės paieškos algoritmą, kuris įgavo aukščiausią galimą tinkamumo įvertinimą, kas reiškia, kad vieno iš populiacijos individų programinis kodas pilnai atitiko literatūroje prieinamą versiją.

```
for (int AX11 = 0; AX11 < arr.Length; AX11++)
{
    if (arr[AX11] == 770)
        return AX11;
}

return -1;
```

41 pav. *Išvystytas tiesinės paieškos algoritmo kodas*

Rezultatai parodė, kad karkaso darbo metu pavyko išvystyti idealų tiesinės paieškos algoritmą. Išvystyto algoritmo kodas pateiktas 41-ame paveikslėlyje. To buvo galima tikėtis darbo pradžioje. Kadangi algoritmo programinis kodas yra labai paprastas ir susideda tik iš ciklo ir loginio išsišakojimo, tai nesukėlė kokių nors sunkumų. Programinis kodas išvystomas sąlyginai greitai, gaunamas tikslus algoritmas, kuris puikiai atlieka jam paskirtą užduotį. Net ir genetinio tobulinimo pagalba šis programinis kodas negali būti pagerintas. Taigi, sukurto karkaso pagalba buvo pilnai išvystytas vienas iš tradicinių paieškos algoritmų, ko pasėkoje galima pereiti link sudėtingesnių algoritmų vystymo.

2.3.3. Išrinkimo rikiavimo algoritmas

Išrinkimo rikiavimo (angl. *selection sort*) algoritmas taip pat yra vienas iš tradicinių informatikos mokslų algoritmų. Jis yra sąlyginai paprastas ir primena burbulo algoritmą. Pagrindinė algoritmo esmė yra kiekvienos iteracijos metu į duomenų aibės pradžią iškelti mažiausios ar didžiausios reikšmės elementą, priklausomai nuo rikiavimo tvarkos. Tada veiksmai kartojami su duomenų aibe be pirmojo elemento, taip pamažu pereinant per visus duomenis. Darbo pabaigoje gaunama pilnai surikiuota duomenų aibė. Algoritmas yra nesudėtingas, bet jo išvystymas karkaso pagalba yra įdomus.

```
static void SelectionSort(int[] arr)
{
    for (int i = 0; i < arr.Length - 1; i++)
    {
        int minId = i;

        for (int j = i + 1; j < arr.Length; j++)
            if (arr[j] < arr[minId])
                minId = j;

        int temp = arr[minId];
        arr[minId] = arr[i];
        arr[i] = temp;
    }
}
```

42 pav. *Išrinkimo rikiavimo algoritmo programinis kodas C# kalba*

Išrinkimo rikiavimo algoritmo kodas C# programavimo kalba pateiktas 42-ame paveikslėlyje. Kodo segmentas yra sąlyginai paprastas, du ciklai ir vienas loginis išsišakojimas, o pabaigoje - reikšmės apkeitimas. Šiuo atveju, sudėtingiausia dalis slypi antrojo ciklo sąlygos sakinyje. Norint, kad išrinkimo rikiavimo algoritmas veiktų korektiškai, antrasis ciklas turi būti

pasistūmėjęs nuo pirmojo. Taip pat, verta paminėti, kad šiame algoritme daug kartų naudojami specifiniai masyvo indeksai, tai irgi yra problematinė sritis, atsižvelgiant į tai, kad kiekvienas iš jų bus parenkamas atsitiktinai ir korektiškas išdėstymas nėra dažnas įvykis. Taigi, nors algoritmas ir nesudėtingas, į tam tikras programinio kodo dalis privalu atsižvelgti, bandant algoritmą sukurti nuo pradžių.

Pritaikant karkasą šio algoritmo vystymui pirmiausia teko apriboti jau aptarto burbulo algoritmo potencialą. Tai yra visiškai dirbtinis eksperimentas, norint patikrinti vystomų gramatinių taisyklių galimybes. Realios užduoties atveju apribojimai yra nereikalingi. Atlikus minėtas modifikacijas, pereita prie karkaso pritaikymo šiam algoritmui. Ciklas, loginis išsišakojimas bei masyvo elementų reikšmės apkeitimas jau buvo pilnai įgyvendinti ankstesnių algoritmų vystymo metu, todėl į tai papildomai gilintis nereikėjo. Teoriškai, visi šie elementai gali būti sukurti karkaso darbo metu. Sudėtingiausia šio algoritmo dalis yra specifiniai ciklo sakiniai, kurie buvo aptarti burbulo algoritmo darbo metu, todėl gramatinių taisyklių modifikacijos neprireikė. Išrinkimo rikiavimo algoritmas reikalauja tik daugiau programinių resursų, nes jo kodas yra sudėtingesnis. Nerasta jokių kliūčių, kurios neleistų iš tuščios programos, gramatinių taisyklių pagalba, išvystyti norimo algoritmo.

Kaip ir buvo minėta, šis algoritmas nereikalavo naujų modifikacijų ir nedaug kuo skyrėsi nuo jau aptarto burbulo metodo. 43-ame paveikslėlyje pateikta, kaip kito šį algoritmą atspindinčių individų tinkamumas, vykstant darbui. Kaip ir ankstesnių algoritmų atveju, pirmaisiais karkaso žingsniais geras algoritmo kodas dar neegzistuoja, darbas vyksta su individais, turinčiais daug įvairių genų, iš kurių tik keli yra naudingi. Paveikslėlyje atvaizduoto tyrimo metu, dalis šių gerų genų perėjo į vieną individą vienuoliktoje genetinio algoritmo kartoje ir sparčiai padidino vidutinį populiacijos tinkamumo įvertį. Tęsiant darbą pamažu išsivystė pilnas išrinkimo atrankos algoritmas. Eksperimentuojant, pavyko gauti kelis atskirus atvejus, kai bendras algoritmo kodas buvo gautas, tačiau juose egzistavo kodo defektai, kurie neišnyko net ir genetinio tobulėjimo metu. Galiausiai toks individas buvo rastas ir pasiektas didžiausias tinkamumo įvertinimas, **1,19**. Individo programinis kodas pateiktas 44-ame paveikslėlyje. Tai atitinka literatūroje prieinamo kodo tinkamumo įvertį, todėl galima teigti, kad karkasas sėkmingai išvystė išrinkimo rikiavimo algoritmą.

43 pav. *Išrinkimo rikiavimo algoritmo tinkamumo augimas*

```
for (int I00Z = 0; I00Z < arr.Length - 1; I00Z++)
{
    int _RO0 = I00Z;

    for (int T111F = I00Z + 1; T111F < arr.Length; T111F++)
    {
        if (arr[T111F] < arr[_RO0])
            _RO0 = T111F;
    }

    int _G00 = arr[_RO0];
    arr[_RO0] = arr[I00Z];
    arr[I00Z] = _G00;
}
```

44 pav. *Išvystytas išrinkimo rikiavimo algoritmo kodas*

2.3.4. Įterpimo rikiavimo algoritmas

Įterpimo rikiavimo (angl. *insertion sort*) algoritmas taip pat labai panašus į burbulo ir išrinkimo rikiavimo algoritmus. Algoritmo esmė yra duomenų aibę padalinti į dvi atskiras dalis, kairiąją ir dešiniąją. Kairiojoje, darbo pradžioje yra tik vienas elementas. Tada, iš dešiniosios palaipsniui imama po vieną elementą ir jis įterpiamas į jam priklausančią vietą kairiojoje dalyje. Tai kartojama iki kol dešinioji pusė tampa tuščia. Nors algoritmas ir nėra labai išskirtinis, įterpimo funkcijos vystymas yra pakankamai unikalus uždavinys sukurto karkaso darbui patikrinti.

```
private static void InsertionSort(int[] arr)
{
    int n = arr.Length;

    for (int i = 1; i < n; ++i)
    {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }

        arr[j + 1] = key;
    }
}
```

45 pav. *Įterpimo rikiavimo algoritmo programinis kodas C# kalba*

Įterpimo rikiavimo algoritmo programinis kodas, įgyvendintas C# programavimo kalba, yra pateiktas 45-ame paveikslėlyje. Šiuo atveju, programinis kodas neturi loginių išsišakojimų, tačiau susideda iš dviejų ciklų. Pirmasis ciklas yra sąlyginai paprastas, tačiau antrasis panaudoja dar nenagrinėtą loginį operatorių **&&**. Šio ciklo sąlygos sakinys yra pakankamai sudėtingas, todėl galima tikėtis ilgo paieškos laiko. Įterpimo rikiavimo algoritme naudojama pakankamai daug reikšmių padidinimo ar sumažinimo per vienetą funkcijų. Nepaisant išvardintų sunkumų, šio algoritmo išvystymas sukurto karkaso pagalba yra įdomus uždavinys.

Vystant įterpimo rikiavimo algoritmą, viena iš programinio kodo detalių, į kurią reikėjo atsižvelgti, tai papildomo loginio operatoriaus pridėjimas. Taip pat, įgyvendintos gramatikos taisyklės, leidžiančios atlikti ciklą ***while*** su tam tikra sąlyga. Svarbus momentas yra tai, kad šis ciklas gali labai lengvai tapti begaliniu, kada jo baigties sąlyga yra niekada nepatenkinama, o ciklo viduje nėra kontrolės šuolį įvykdančių elementų. Dėl to, karkasą reikėjo papildomai modifikuoti, stabdant individus, kurie uždavinį spręsdavo per ilgai. Stabdymo procesas detaliai aptartas 1.3.2.4 straipsnyje. Žinoma, tokių individų įvertinimas žymiai nukenčia ir tikėtina, kad jie ilgai neišgyvens. Kaip ir praeitame skyriuje apžvelgto algoritmo metu, atliekant šį tyrimą irgi reikėjo užkirsti kelią kitų, panašių algoritmų vystymui. Žinoma, tai nėra reikalinga bandant gauti bet kokį, uždavinį atliekantį, algoritmą ir yra atlikta tik eksperimento dėlei.

Įterpimo rikiavimo algoritmas nereikalavo daug naujų modifikacijų ir nedaug kuo skyrėsi nuo jau aptartų rikiavimo metodų. Individų tinkamumo kitimas, vykstant darbui, pateiktas 46-ame paveikslėlyje. Tinkamumo augimas neparodo jokių netikėtumų, staigių šuolių ar kritinių reikšmių, genetinio algoritmo darbas vyksta sąlyginai paprastai ir nuspėjamai. Pirmiausia, tinkamumas auga nežymiai, kol ties aštunta karta gaunami palankūs genai ir įvertinimas per kelias kartas šoka į viršų.

Darbo eigai tęsiantis algoritmas po truputį optimizuojamas, pranyksta defektai. Nors įterpimo rikiavimo algoritmas nereikalavo papildomų modifikacijų, sėkmingai išvystyti jo programinį kodą buvo ganėtinai sunku. Kad būtų sukurtas uždavinį išsprendžiantis algoritmas, sudėtingas ciklo sąlygos sakinys ir daug reikšmių didinimo operacijų turėjo atsirasti beveik vienu metu. Dėl to, įterpimo rikiavimo algoritmo vystymas reikalavo daug pakartotinių eksperimentų. Atliekant tyrimą, vieno iš eksperimentų metu buvo sukurtas reikalingas individas. Karkasui tęsiant darbą, pašalinti visi kodo defektai, atsiradę dėl atsitiktinumo ir mutacijų, taip keliant gauto gero individo tinkamumą. Rezultate pasiektas didžiausias tinkamumo įvertinimas, atitinkantis literatūroje prieinamo įterpimo rikiavimo algoritmo kodo tinkamumą, **1,17**. Individo programinis kodas pateiktas 47-ame paveikslėlyje. Taigi, sukurto programinio karkaso pagalba, buvo pilnai išvystytas įterpimo rikiavimo algoritmo programinis kodas.

46 pav. *Įterpimo rikiavimo algoritmo tinkamumo augimas*

```
for (int JBB = 1; JBB < arr.Length; ++JBB)
{
    int D00A = arr[JBB];
    int _0JL = JBB - 1;

    while (_0JL >= 0 && arr[_0JL] > D00A)
    {
        arr[_0JL + 1] = arr[_0JL];
        _0JL = _0JL - 1;
    }

    arr[_0JL + 1] = D00A;
}
```

47 pav. *Išvystytas įterpimo rikiavimo algoritmo kodas*

2.3.5. Dvejetainės paieškos algoritmas

Vienas sudėtingiausių algoritmų, kurį buvo bandoma išvystyti šio darbo metu, yra dvejetainės paieškos algoritmas. Algoritmas yra sąlyginai populiarus, o jo įgyvendinimas pakankamai paprastas. Dvejetainės paieškos algoritmas turi tam tikrą trūkumą – jis gali dirbti tik su išrūšiuotu masyvu, todėl tai sumažina jo panaudos atvejus realiose sistemose. Nepaisant to, algoritmas yra labai įdomus. Pagrindinė šio algoritmo esmė, tai tiriamos duomenų aibės dalinimas pusiau, per vidurį. Kadangi žinoma, kad duomenys išrūšiuoti, padalinus juos per pusę ir palyginus vidurio reikšmę su ieškomąja, galima iš karto atmesti vieną iš pusių. Tada toliau tiriama tik likusi duomenų pusė, kuri analogiškai dalinama per pusę ir veiksmai kartojami, tol, kol randamas ieškomas elementas arba baigiasi duomenys. Pabaigoje, grąžinamas ieškomojo elemento indeksas arba neigiama reikšmė, atspindinti nesėkmę. Dvejetainės paieškos algoritmo išvystymas yra pakankamai sudėtingas ir įdomus uždavinys, kuris išspręstas sukurto karkaso pagalba.

Dvejetainės paieškos algoritmo įgyvendinimas įmanomas dviem būdais. Pirmiausia, tai galima atlikti rekursinių veiksmų pagalba. Tokiu atveju, algoritmas pirmiausia patikrina ar vidurinė reikšmė yra lygi ieškomajai, jei ne – vienoje iš pusių atliekami identiški veiksmai. Randama vidurinė reikšmė ir ji palyginama, tada pasirenkama sekanti pusė. Rekursinių veiksmų naudojimas yra pakankamai sudėtingas procesas programinėje kalboje, dėl ko prireiktų žymiai praplėsti naudojamų gramatinių taisyklių sąrašą, kas žymiai padidintų galimų sprendinių aibę ir sulėtintų sukurto karkaso darbo eigą. Egzistuoja paprastesnė dvejetainės paieškos algoritmo realizacija, besiremianti iteraciniu procesu. Šio metodo realizacija C# programavimo kalbos aplinkoje yra pateikta 48-ame paveikslėlyje. Vienintelis esminis skirtumas nuo rekursinio įgyvendinimo yra ciklo naudojimas. Ciklas sukasi tol, kol nepatenkinama jo sąlyga arba įvyksta kontrolės perdavimas. Darbo metu vis atnaujinamos reikšmės, žyminčios tiriamųjų duomenų dalį. Nors dvejetainės paieškos iteracinio metodo programinis kodas atrodo sąlyginai paprastas, yra keletas aspektų į kuriuos reikia atsižvelgti, norint tokį algoritmą išvystyti nuo nulio.

Kadangi tai yra antrasis paieškos algoritmas, kuris vystomas sukurto karkaso pagalba, reikia atlikti tam tikrų papildomų karkaso modifikacijų. Anksčiau apžvelgtas tiesinės paieškos algoritmas yra labai paprastas, todėl jo kodo tinkamumo įvertinimas yra geresnis nei dvejetainės paieškos algoritmo. Tikėtina, kad karkaso darbo metu, dėl visiško atsitiktinumo generuojant eilutes ir jas vertinant, algoritmas, kuris bus sukurtas, priklausys tik nuo programinio kodo paprastumo. Reikia atlikti papildomų karkaso modifikacijų tam pakeisti. Atliekant užduotį buvo įgyvendintas programinio kodo spartos matavimas. Kadangi dvejetainės paieškos algoritmas bendruoju atveju yra greitesnis, labiau tikėtina, kad jis rezultatą gaus greičiau. Pagal tai, vertinant individo

tinkamumą ir remiantis 1.3.2.4 straipsnyje aptarta tinkamumo funkcija, galutinis rezultatas yra modifikuojamas, pranašumą suteikiant greitesniam individui. Papildomai, kadangi darbas vyksta su sąlyginai mažomis duomenų aibėmis ir skirtumas tarp algoritmų spartos yra labai mažas, matuojamas tam tikrų programinių elementų egzistavimas, leidžiantis identifikuoti kuriamą algoritmą. Ši modifikacija yra dirbtinė, jos nereikia normalaus tyrimo metu, kai nėra bandoma išvystyti tam tikrą algoritmą, todėl ji laikina, egzistuojanti tik eksperimento metu ir detaliai nebus analizuojama. Aptartos modifikacijos leidžia karkasą pakreipti link tam tikro algoritmo išvystymo, norint patikrinti pagal gramatikos taisykles kuriamą naują programinį kodą.

```
private static int BinarySearch(int[] arr, int x)
{
    int left = 0;
    int right = arr.Length - 1;

    while (left <= right)
    {
        int middle = left + ((right - left) / 2);

        if (arr[middle] == x)
            return middle;

        if (arr[middle] < x)
            left = middle + 1;
        else
            right = middle - 1;
    }

    return -1;
}
```

48 pav. *Iteracinis dvejetainės paieškos programinis kodas C# kalba*

Karkaso pritaikymo dvejetainės paieškos algoritmui metu teko susidurti su papildomais programavimo iššūkiais. Svarbu leisti tam tikriems sakiniams šakotis ir įgauti sudėtingesnius pavidalus, nes iteracinis dvejetainės paieškos programinis kodas, skaičiuodamas vidurinės reikšmės indeksą, panaudoja ganėtinai sudėtingą matematinę operaciją. Užtikrinimui, kad šis sakinys galės būti sugeneruojamas, svarbu pakeisti svorius pagal kuriuos šakojimasis yra ribojamas. Taip pat, papildytos loginio išsišakojimo taisyklės, leidžiančios apžvelgti ***else*** programinio kodo dalį. Iki šiol, toks funkcionalumas nebuvo reikalingas, tačiau šis pakeitimas yra nedidelis ir jį atlikti nebuvo sunku. Kontrolinių šuolių su raktiniu žodžiu ***return*** įgyvendinimas buvo atliktas tiesinės paieškos algoritmo vystymo metu, todėl ši dalis problemų nesukėlė. Taip pat, dvejetainės paieškos algoritmas naudoja bent kelis lokalius parametrus reikšmėms saugoti. Lokalaus parametro kūrimas buvo trumpai apžvelgtas burbulo algoritmo metu, todėl šiuo atveju tai nereikalavo papildomų pastangų. Atlikus išvardintus papildomus pakeitimus C# programavimo kalbos gramatikoje, vienintelė keblumų kelianti dalis, kuri jau buvo trumpai paminėta, yra

vidurinės reikšmės skaičiavimas. Intuityviai, atliekami veiksmai yra aiškūs ir paprasti, tačiau labai svarbi jų korektiška interpretacija. Karkasas turi sukurti tokią reikšmę modifikuojančią eilutę, kuri pirmiausia atliktų atimties veiksmą tarp dviejų elementų, tada rezultatą padalintų iš skaitinės reikšmės ir dar kartą gautą rezultatą susumuotų su papildomu elementu. Kadangi visos eilutės yra generuojamos atsitiktinai, tikimybė, kad visi šie veiksmai įvyks vienu metu yra labai maža. Algoritmas savo sudėtimi yra pats sudėtingiausias, nes jo korektiškam išvystymui turi būti sukuriamas didelis skaičius tarpusavyje bendraujančių kodo eilučių. Tai reiškia, kad dvejetainės paieškos algoritmo vystymas reikalaus ypatingai daug programinių resursų ir tikėtina, užtruks žymiai ilgiau negu visi kiti tirti algoritmai.

Dvejetainės paieškos algoritmo vystymas reikalavo daug programinių resursų, o gauti rezultatai kito bėgant laikui, atliekant papildomas karkaso modifikacijas. Eksperimentų metu, nepavyko gauti visiškai idealus dvejetainės paieškos algoritmo programinio kodo, tačiau tai nereiškia, kad to padaryti neįmanoma. Darbo proceso metu išmoktos naudingos pamokos. Eksperimentuojant, keičiami tam tikri karkaso parametrai, bandant padėti karkasui gauti šį sudėtingesnį programinį kodą, tačiau su sėkmingu, visų reikalingų genų sugeneravimo vienu metu, atveju nebuvo susidurta. Taip pat, gelbstint karkaso darbui, buvo keičiami gramatinių taisyklių atsitiktinio generavimo svoriai, leidžiant matematinėms operacijoms šakotis plačiau. Tai buvo daroma bandant išgauti vidurinės reikšmės skaičiavimo sakinį, kas buvo viena iš sudėtingiausių vietų šio tyrimo metu. Tikrinant generuojamą programinį kodą, visos reikalingos programinės eilutės buvo gautos vienu ar kitu atveju, tačiau jų atsiradimas viename individe kryžminimo pagalba neįvyko. To priežastis yra ganėtinai didelis atsitiktinumas ir labai ilgas skaičiavimo laikas. Kiekvienas eksperimentas, tyrimą atliekantis iki maksimalios pasirinktos kartos, užtrukdavo maždaug 37 valandas, todėl buvo atlikta tik keletas tokių tyrimų. Kiekvieno iš jų pabaigoje atliktos karkaso modifikacijos, siekiant paspartinti skaičiavimų greitį ir pakeisti gramatinių taisyklių pritaikymo aspektus. Tikėtina, kad turint daugiau programinių resursų arba atliekant eksperimentus ilgiau, individas, pilnai išvystantis dvejetainės paieškos algoritmą, būtų rastas, nes teorinis kelias iki to yra įmanomas. Taigi, nors dvejetainės paieškos algoritmo pilnai išvystyti nepavyko, proceso metu buvo praplėstas supratimas apie karkaso darbą ir padidintas gramatinių taisyklių sąrašas, kas leidžia ateityje daromus tyrimus atlikti žymiai detaliau.

Taigi, šio darbo metu pabandyta išvystyti dvejetainės paieškos algoritmą, įgyvendintą iteraciniu būdu. Šio algoritmo programinis kodas, iš pirmo žvilgsnio, yra sąlyginai paprastas, tačiau gilinantis į gramatikos taisykles, kurios yra reikalingos, norint tokį kodą gauti, nesunku pastebėti, kad tai vis dėl to yra sudėtingas procesas. Algoritmo darbo metu turi būti sukuriami kritiškai svarbūs individo genai, atspindintys sudėtingas programinio kodo eilutes. Šių genų bendravimas tarpusavyje taip pat privalo būti išlaikomas, ko pasėkoje, tikėtis tokio atsitiktinio

įvykio korektiškos baigties per trumpą laiko tarpą, negalima. Darbo metu patikrintas labai didelis skaičius galimų individų, kurie buvo atitinkamai vertinami. Buvo svarbu apsibrėžti algoritmą, kurio ieškoma, nes karkasas linkęs užduotį spręsti paprasčiausiu būdu, todėl labiausiai tikėtina, kad paieškos metu būtų gaunamas tiesinis algoritmas, aptartas ankstesniame skyriuje. Realaus tyrimo metu, kai nėra bandoma gauti tam tikrą rezultatą, tokie apribojimai yra pašalinami ir karkasas darbą atlieką korektiškai, ieškodamas bet kokio algoritmo, sugebančio išspręsti pateiktą užduotį efektyviausiai. Vis dėl to, šio mokslinio tyrimo metu sukurtomis taisyklėmis yra įmanoma pilnai išvystyti tokį algoritmą, teorinis kelias nuo tuščio vieneto iki dvejetainės paieškos algoritmo yra aiškus ir patikrinamas 1.2.2 skirsnyje aptartu būdu. Visa tai parodo, kad papildytas gramatinių taisyklių sąrašas atvėrė duris link daug sudėtingesnių algoritmų kūrimo, o proceso sėkmė ir sparta priklauso tik nuo užduočiai paskirtų resursų kiekio.

2.4. Optimizuojami parametrai

Genetinis algoritmas turi daug svarbių parametrų. Operatorių veikimo dažnumas, populiacijos dydis ir genetinių kartų skaičius yra keletas iš jų. Tai įvairios modifikacijos, apie kurias buvo kalbama ankstesniuose skyriuose. Jos taip pat prideda papildomų, reikalingų parametrų, kurie bus sekami: minimalus ir maksimalus populiacijos skaičius, individo mirties tikimybė, migracijos tikimybė ir t.t. Kiekvieno iš šių parametrų pokytis gali įvairiai paveikti galutinį genetinio programavimo rezultatą. Susiduriama su atskira problema - kaip patikrinti visus parametrų variantus ir atsirinkti geriausią? Vienas iš būdų, kurį netiesiogiai pasiūlo autorius Kenneth De Jong [Jon88], tai leisti genetiniam algoritmui optimizuoti save. Šio straipsnio tikslas – suprasti kada ir kaip genetinis algoritmas gali sukelti tikslinius ir struktūrinius pokyčius besimokančiose sistemose. Tyrimo metu atskiriamos dvi posistemės: užduoties atlikimo ir mokymosi. Genetinis algoritmas pritaikomas parametrų optimizacijai. Jis perima mokymosi sistemos vaidmenį ir bando gerinti užduoties atlikimo sistemos darbą. Autorius išbando ir daugiau būdų, kuriuos būtų galima panaudoti, kuriant genetinį algoritmą kitos sistemos optimizavimui, bet jie šiuo atveju nėra svarbūs. Tyrimo rezultate nustatoma, kad genetinis algoritmas nėra pats geriausias sistemų optimizavimo būdas, tačiau pasiūlo efektyvų sprendimą tam tikrose situacijose. Tokias idėjas, apmąstymus bei tyrimus galima pritaikyti ir šio darbo kontekste. Kadangi kuriamas sudėtingas karkasas, jį galima traktuoti kaip užduoties atlikimo posistemę, turinčią daug parametrų. Sukurkime atskirą, paprastą genetinį algoritmą, kuris optimizuotų šios sudėtingos posistemės darbą. Kad tai būtų galima atlikti, vėlgi reikia apsibrėžti esmines genetinio algoritmo sąvokas. Šiuo atveju, individas – tai vienas karkaso parametrų rinkinys. Individo genai – atskirų

parametrų skaitinės vertės. Populiacija taptu tokių individų masyvu. Kadangi visi parametrai yra skaitiniai, labai paprasta juos užkoduoti bei pritaikyti genetinius operatorius. Tereikia sekti šių parametrų rėžius. Vienos iteracijos etape būtų įvykdomas visas vidinis karkaso darbas ir gaunamas vienas rezultatas – geriausio išvystyto algoritmo skaitinis įvertinimas. Dabar jau turime viską ko reikia mokymo procesui įvykdyti. Šis procesas reikalaus didelio kiekio skirtingų eksperimentų vykdymo. Sunku iš anksto nuspėti ar toks optimizavimo būdas pasiteisins.

Kitas būdas, kurio literatūroje detaliai aprašyto nerasta, tai dirbtinio neuroninio tinklo pritaikymas genetinio algoritmo parametrų optimizavimo uždaviniui. Savo idėjomis jis panašus į anksčiau minėtą būdą, t.y. atskiriamos užduoties atlikimo bei mokymosi sistemos. Autoriai Laurent Magnier ir Fariborz Haghighat savo darbe [MF10] neuroninį tinklą panaudojo genetinio algoritmo tinkamumo funkcijos optimizavimui. Tokio tyrimo tikslas – pasiūlyti būdą gyvenamųjų namų energijos suvartojimo ir šilumos pasiskirstymo optimizacijai. Vietoje sudėtingų tinkamumo skaičiavimų autoriai pirmiausia neuroninį tinklą apmokė atpažinti palankius objektus, o tada jį naudojo jų tinkamumo lygiui įvertinti. Rezultatai parodė, kad sukurtas dvilypis daugiakriterinis optimizacijos algoritmas padėjo pasiekti žymiai geresnius rezultatus. Nors šis tyrimas nėra susijęs su algoritmų paieška, autorių pasiūlytas būdas apjungti dirbtinį neuroninį tinklą ir genetinį algoritmą yra įdomus. Algoritmų paieškos darbo metu tinkamumo skaičiavimas nėra labai sudėtingas, todėl atskiro neuroninio tinklo naudojimas nepadės išlošti tikslumo ar greičio. Vis dėl to, analogiškai, tinklą galima pritaikyti parametrų optimizavimo uždaviniui. Tinklo įeitys būtų kuriamo karkaso parametrų vidutinės reikšmės. Neuroninis tinklas jiems priskirtų svorius, taip modifikuojant jų konkrečias reikšmes ir užtikrinant, kad nebus peržengti specifiniai rėžiai. Tada, karkasas atliktų skaičiavimus su duotaisiais parametrais, o rezultate butu gaunamas geriausio išvystyto algoritmo įvertinimas. Žinoma, toks procesas reikalautų atskiro neuroninio tinklo apmokymo, kam reiktų papildomų resursų, tačiau tai dar vienas būdas parametrų optimizacijai atlikti.

Atlikti karkaso parametrų optimizaciją įmanoma pasitelkiant ne tik genetinį algoritmą ar neuroninius tinklus, bet ir kitus mašininio mokymosi algoritmus. Vienas iš jų, kurį galima aptarti detaliau, tai vėsinimo imitacijos (angl. *simulated annealing*) algoritmas. Šis algoritmas remiasi metalurgijos principais, kai vėstant įkaitintam metalui pašalinami tam tikri defektai. Algoritmas yra sąlyginai paprastas ir detalų jo aprašymą galima rasti laisvai prieinamoje literatūroje. Tokio optimizacijos būdo pranašumas yra žymiai paprastesnis programinio kodo pritaikymas. Vienintelė algoritmo vieta, į kurią reikia atkreipti dėmesį, yra susijusi su atsitiktiniu sekančio tiriamo elemento sudarymu. Tyrimo atveju algoritmas generuoja parametrų sąrašo perstatą. Kiekvienas iš parametrų turi tam tikras, iš anksto nustatymas minimalias ir maksimalias reikšmes, kurios pasirenkamos atsitiktinai, su vienoda tikimybe. Gautas parametrų sąrašas yra pritaikomas ir

vėsinimo imitacijos algoritmas per vieną iteraciją simuliuoja viso karkaso darbą. Sekančioje iteracijoje parenkamas naujas parametrų sąrašas ir procesas kartojamas. Išvardinti veiksmai yra pakankamai paprasti. Taigi, vėsinimo imitacijos algoritmas leistų nesunkiai atlikti kuriamo karkaso parametrų optimizavimą.

Detaliai aptarti bent trys unikalūs karkaso optimizacijos būdai, tačiau šiame tyrime aktualus tik vieno iš jų pritaikymas. Svarbu nepamiršti šio darbo tikslo, todėl atlikti papildomų tyrimų, susijusių vien su karkaso optimizacija, nėra produktyvu. Genetinio algoritmo naudojimas optimizuoti kitą genetinį algoritmą yra įdomus sprendimas, bet tai dvigubina jau atliekamą darbą. Proceso metu, reiktų papildomai gilintis į optimizacijos algoritmo genetinius operatorius, individus ir populiaciją, kas reikalauja daug laiko ir pastangų. Dėl to, šis būdas netinka darbo kontekste. Sekantis – dirbtinių neuroninių tinklų naudojimas yra taip pat labai įdomus ir didelį potencialą turintis metodas. Šio metodo pritaikymas yra labai sudėtingas ir reikalautų daug papildomų pastangų dirbtinio neuroninio tinklo kodo paruošimui. Deja, šio būdo taip pat atsisakoma. Tai nereiškia, kad minėtų būtų pritaikyti negalima, galbūt jų pagalba būtų galima gauti analogiškus ar net geresnius rezultatus, bet toks eksperimentavimas gali būti atliekamas, esant atskiram suinteresuotų šalių poreikiui. Taigi, karkaso parametrų optimizacijai atlikti bus taikomas vėsinimo imitacijos algoritmas. Algoritmo įgyvendinimas yra greitas, o kodo pritaikymas paprastas, todėl tai nereikalauja papildomų pastangų ar tyrimų. Algoritmas yra vienas iš tradicinių mašininio mokymosi algoritmų, dažnai minimas literatūroje, todėl jo pliusų ir minusų papildomai nagrinėti nebūtina. Detaliau apie vėsinimo imitacijos kodavimą kalbama 2.4.1 skirsnyje. Galiausiai, pasirinkus optimizacijos procesą, verta apsibrėžti visus turimus karkaso parametrus bei juos sugrupuoti. Pilnas karkaso parametrų sąrašas pateiktas 3-oje lentelėje. Lentelėje, antrasis stulpelis žymi kode esantį parametro identifikatorių.

3 lentelė. *Karkaso parametrai*

Pavadinimas	**Kodo žymėjimas**	**Tipas**	**Komentarai**
Vidutinis atrankos operatoriaus turnyro dydis.	*SelectionDesiredAverage TournamentSize*	**Double** – skaičius su trupmenine dalimi.	Negali būti sveikas skaičius.
Populiacijos dydis.	*PopulationSize*	**Int** – sveikas skaičius.	
Populiacijos maksimalus rėžis.	*PopulationMaxThreshold*	**Int** – sveikas skaičius.	Tik teigiamas ir didesnis už pradinės populiacijos skaičių.
Populiacijos minimalus rėžis.	*PopulationMinThreshold*	**Int** – sveikas skaičius.	Tik teigiamas ir tarp nulio bei pradinio populiacijos skaičiaus.
Maksimalus algoritmo iteracijų skaičius.	*GenerationLimit*	**Int** – sveikas skaičius.	Tik teigiamas ir didesnis už vienetą.
Mutacijos dažnumas.	*MutationRate*	**Int** – sveikas skaičius.	Apribotas rėžiuose 0-100. Išreikšta kaip tikimybė iš šimto.
Kryžminimo nediskriminuojančio eilučių apkeitimo tikimybė	*CrossoverNonDiscriminant SwitchRate*	**Int** – sveikas skaičius.	Apribotas rėžiuose 0-100. Išreikšta kaip tikimybė iš šimto.
Migracijos atrankos tipas.	*MigrationSelectionType*	**Enum** – išvardinimo tipas.	Apribotas pasirinkimu iš jau žinomų reikšmių.
Migracijos įterpimo tipas.	*MigrationInsertionType*	**Enum** – išvardinimo tipas.	Apribotas pasirinkimu iš jau žinomų reikšmių.
Migracijos atrankos turnyro dydis.	*MigrationTournamentSize*	**Int** – sveikas skaičius.	Tik teigiamas ir didesnis už vienetą.

Migracijos operatoriaus migruojančių individų skaičius.	*MigrationMigrating IndividualsCount*	**Int** – sveikas skaičius.	Tik teigiamas ir didesnis arba lygus vienetui.
Migracijos dažnumas.	*MigrationRate*	**Int** – sveikas skaičius.	Tik teigiamas ir didesnis už vienetą.
Karo operacijos turnyro dydis.	*WarTournamentSize*	**Int** – sveikas skaičius.	Tik teigiamas ir didesnis už vienetą.
Salų skaičius.	*IslandCount*	**Int** – sveikas skaičius.	Tik teigiamas ir didesnis už vienetą.
Pabaigos kriterijus.	*EndCriteria*	**Float** – skaičius su trupmenine dalimi.	Tik teigiamas.
Genetinio tobulėjimo populiacijos dydis.	*RefactoringPopulationSize*	**Int** – sveikas skaičius.	Tik teigiamas ir didesnis už vienetą.
Genetinio tobulėjimo mutacijų vienam individui.	*RefactoringMutations PerIndividual*	**Int** – sveikas skaičius.	Tik teigiamas ir didesnis už vienetą.
Genetinio tobulėjimo maksimalus iteracijų skaičius.	*RefactoringGenerationCount*	**Int** – sveikas skaičius.	Tik teigiamas ir didesnis už vienetą.
Genetinio tobulėjimo populiacijos maksimalus rėžis.	*RefactoringPopulation MaxThreshold*	**Int** – sveikas skaičius.	Tik teigiamas ir didesnis už pradinės genetinio tobulėjimo populiacijos skaičių.
Genetinio tobulėjimo populiacijos minimalus rėžis.	*RefactoringPopulation MinThreshold*	**Int** – sveikas skaičius.	Tik teigiamas ir tarp nulio bei pradinio genetinio tobulėjimo populiacijos skaičiaus.

Kad būtų gauti geriausi rezultatai, reikia surasti optimaliausią parametrų rinkinį. Dalį iš jų galima parinkti, atsižvelgiant į patirtį ir literatūros šaltinius, tačiau likusius parametrus reikia tirti eksperimentų pagalba. Galimų parametrų rinkinių aibė yra labai didelė. Pilnas aibės tyrimas reikalauja daug resursų, bet, nepaisant to, galima surasti tam tikrą rinkinį, kurio pagalba gaunami pastovūs, geri rezultatai. Parametrų rinkinys, kuris šio darbo metu padėjo gauti geriausius rezultatus yra pateiktas 4-oje lentelėje.

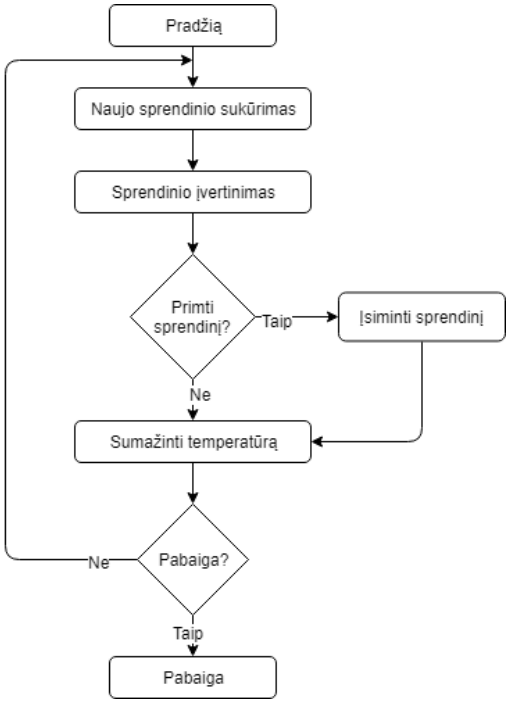
4 lentelė. *Geriausias rastas parametrų rinkinys*

Parametro pavadinimas anglų kalba	Optimali parametro reikšmė
SelectionDesiredAverageTournamentSize	5,6
PopulationSize	10000
PopulationMaxThreshold	10000
PopulationMinThreshold	9000
GenerationLimit	50
MutationRate	8%
CrossoverNonDiscriminantSwitchRate	7%
MigrationSelectionType	Tournament
MigrationInsertionType	Weakest
MigrationTournamentSize	5
MigrationMigratingIndividualsCount	3
MigrationRate	6%
WarTournamentSize	3
IslandCount	3
EndCriteria	2
RefactoringPopulationSize	20
RefactoringMutationsPerIndividual	3
RefactoringGenerationCount	5
RefactoringPopulationMaxThreshold	20
RefactoringPopulationMinThreshold	15

2.4.1. Vėsinimo imitacijos algoritmas

Mašininis mokymasis, tai automatizuotas parametrų optimizavimas. Vėsinimo imitacija yra vienas iš mašininio mokymosi algoritmų, kurio pagalba tokį darbą galima lengvai atlikti. Sukurto karkaso aplinka suprojektuota taip, kad programinį kodą būtų galima lengvai paleisti ant atskirų

programinių gijų. Naujo funkcionalumo pridėjimas naujų klasių ir jų metodų pavidalu, lengvai integruojamas į jau esamus projektus. Egzistuoja du pagrindiniai mazgai, tai genetinio algoritmo ***GA*** klasė ir vartotojo sąsajos kodas, paleidžiantis bendrą procesą. Modifikacijos atliekamos tik šiuose dviejuose telkiniuose, kas leidžia pridėti papildomą funkcionalumą, nekeičiant viso karkaso darbo.



49 pav. *Modeliuojamo atkaitinimo algoritmo schema*

Kaip jau buvo minėta ankstesniuose skyriuose, vėsinimo imitacijos algoritmas remiasi metalurgijos principais. Sekama fiktyvi tiriamojo objekto temperatūra, kuri darbo metu pamažu mažėja. Mažėjant temperatūrai, algoritmas linkęs apsistoti ties geriausiais sprendiniais. Priešingu atveju, kai temperatūra yra aukšta, vėsinimo imitacijos algoritmas visiškai atsitiktinai renkasi potencialius kandidatus iš sprendinių erdvės. Vėsinimo imitacijos algoritmo apibendrinta schema pateikta 49-ame paveikslėlyje. Techniškai sudėtingiausia algoritmo vieta yra potencialaus sprendinio pasirinkimas tam tikru algoritmo darbo metu. Jeigu aklai visada pasirenkami tik geriausi gauti sprendiniai, įmanoma užsistovėti lokaliame optimumo taške, todėl reikia su tam tikra tikimybe priimti ir prastus sprendinius. Tikimybė proporcinga temperatūrai, nes kuo įsivaizduojamas objektas labiau aušta, tuo labiau tikėtina, kad pasirenkami tik geriausi sprendiniai. Taigi, toks algoritmo aprašymas yra nesudėtingas, o jo pritaikymo pavyzdžių konkrečioje programavimo kalboje rasti yra nesunku. Autoriaus Assaad Chalhoub internetiniame straipsnyje[Cha06] yra pateiktas vėsinimo imitacijos algoritmo įgyvendinimas C# programavimo kalba, sprendžiantis keliaujančio pirklio uždavinį. Šį kodą nesunku pritaikyti algoritmų paieškos

uždaviniui, tereikia pakeisti sprendinių generavimo procesą. Tai atlikta šio darbo metu. Sukurtame vėsinimo imitacijos algoritme darbo uždaviniui spręsti, vyksta viso karkaso darbo simuliacija kiekvienoje algoritmo iteracijoje. Gautas rezultatas yra įsimenamas, o veiksmai kartojami su nauju sprendiniu. Sprendiniai yra palyginami ir pagal apskaičiuotą tikimybę naujasis iš jų yra atmetamas arba priimamas kaip geriausias sprendinys. Optimizacijos pabaigoje gaunamas vienas parametrų rinkinys, kurį galima išsaugoti tolimesniems tyrimams atlikti. Akivaizdu, kad šis procesas reikalauja daug laiko, todėl jį atlikti dažnai yra pakankamai sunku. Nepaisant to, šiuo būdu pavyko gauti optimizuotą karkaso parametrų rinkinį, su kuriuo gaunami geriausi rezultatai.

Nauja mašininio mokymosi logika, apie kurią teoriškai aptarta anksčiau, įgyvendinta naujame projekte. Projekto sukūrimas leido izoliuoti programinį kodą nuo viso karkaso logikos. Apie programinio kodo paskirstymą detaliai aptarta 2.1.2 skirsnyje. Bendravimas tarp projektų vykta tik paprastais funkcijų kvietimais, kas yra smulkus pakeitimas bendroje kodo visumoje. Vėsinimo imitacijos algoritmas įgyvendintas, remiantis viešai prieinamais programinio kodo pavyzdžiais. Aptartame internetiniame straipsnyje pateikta pakankamai kodo pavyzdžių nesudėtingam algoritmo pritaikymui bet kokiam uždaviniui, todėl tai atlikti šio darbo metu nebuvo sunku.

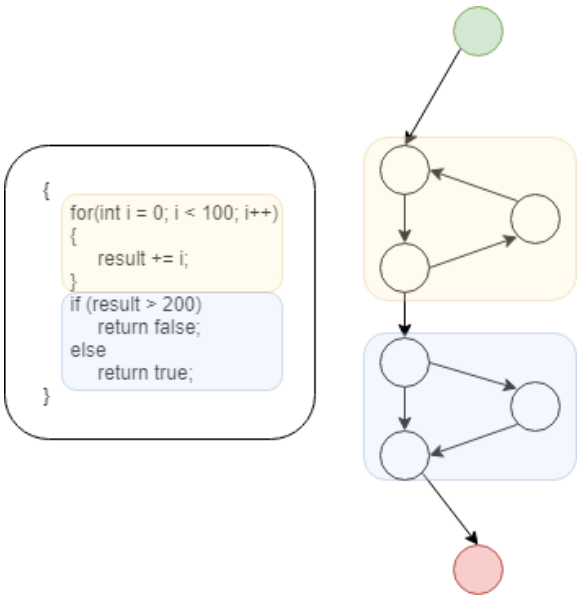
2.5. Kodo vertinimas

Sukurto karkaso veikimo metu yra vystomi įvairūs algoritmai. Tam, kad būtų galima įvertinti jų tinkamumą, atliekamas programinio kodo vertinimas. Tai vyksta kiekvienos iteracijos metu. Detaliau šis procesas buvo išdėstytas 1.3.2 skirsnyje. Tinkamumo vertinimo atveju, pasirinkta metrika yra labai paprasta, nes šis veiksmas kartojamas tūkstančius kartų. Vis dėl to, gavus jau pakankamai gerus algoritmus, t.y., kai genetinis algoritmas pasiekė tam tikrą genetinių kartų skaičių ar tinkamumo kriterijų, būtų pranašu dar kartą įvertinti gautus algoritmus. Pakartotinis įvertinimas leidžia griežčiau pasakyti ar sukurtas algoritmas yra geras. Kadangi toks procesas vyksta tik genetinio algoritmo pabaigoje, su keliais geriausiais individais, galima pritaikyti sudėtingesnes kodo vertinimo metrikas.

Kaip jau buvo minėta, kalbant apie tinkamumo vertinimą ankstesniuose skyriuose, viena iš populiarių kodo vertinimo metrikų yra Mc Cabe‘o metrika, dar kitaip vadinama tiesiog ciklomatinio sudėtingumo metrika. Straipsnyje [SW08] autoriai Yonghee Shin ir Laurie Williams sėkmingai panaudojo net kelias šios metrikos versijas. Jų straipsnio tikslas – nustatyti kodo sudėtingumo metrikas, kurios atskiria pažeidžiamas funkcijas nuo nepažeidžiamų bei ištirti ar kodo metrikos gali padėti nuspėti silpnąsias kodo vietas. Tyrimo metu buvo patikrintos devynios

skirtingos metrikos bei atsakyta į keletą svarbių, su jų tema susijusių klausimų. Rezultate, jiems pavyko atskirti funkcijų tipus šių metrikų pagalba. Mc Cabe‘o metrika remiasi grafo generavimu bei grafo savybių sekimu, kai matuojamas ciklomatinis sudėtingumas. Kaip buvo minėta, kadangi bus vertinama tik geriausių individų grupė, toks procesas nereikalaus pernelyg ilgo laiko tarpo. Autorių Volker Gruhn ir Ralf Laue [GL08] straipsnyje taip pat užsimenama apie Mc Cabe‘o metrikos pranašumus, pritaikant ją verslo procesų modelių analizėje. Autorius Thomas J. Mccabe savo metriką pristatė straipsnyje „A Complexity Measure“[Mcc76], išleistame dar 1976 metais. Straipsnis sulaukė didelio dėmesio ir yra dažnai cituojamas įvairiuose tyrimuose, kur Mc Cabe‘o metrika ar jos variacijos yra pritaikomos. Į originalaus autoriaus šaltinį atsižvelgta, pritaikant ir koduojant kodo įvertinimo mechanizmą. Taigi, remiantis grafais apibrėžta, kad su Mc Cabe‘o kodo sudėtingumo metrika galima įvertinti kuriamų algoritmų efektyvumą, o tai padeda pasiekti užsibrėžtą viso darbo tikslą – surasti geresnius algoritmus.

Mc Cabe‘s metrika įgyvendinta sąlyginai paprastu būdu. Kadangi skaičiuojamas kodo išsišakojimas, jį galima įsivaizduoti, kaip grafą, kurio viršūnės yra programos stadijos, o briaunos žymi perėjimus iš jų. Pavyzdys pateiktas 50-ame paveikslėlyje. Programinio kodo ciklo atveju analogiškai gaunamas ciklas grafe, o loginis išsišakojimas primena medžio grafą. Ciklomatinį sudėtingumą galima skaičiuoti pagal formulę $E - N + 2P$, kur E yra grafo briaunų skaičius, N yra grafo viršūnių skaičius, o P – jungių komponenčių skaičius. Šio darbo atveju, visi kuriami algoritmai turės vieną jungią komponentę, todėl formulę galima supaprastinti iki $E - N + 2$. Paveikslėlyje pateiktame pavyzdyje yra 9 briaunos ir 8 viršūnės, todėl šio paprasto kodo ciklomatinis sudėtingumas yra lygus trims. Nesunku pastebėti, kad turint grafą, metriką apskaičiuoti yra pakankamai paprasta. Tą patį principą galima pritaikyti ir kuriamame karkase. Programinio kodo vertimas į grafą yra perteklinis veiksmas, kurio nereikia pilnai atlikti. Tą patį rezultatą galima gauti, skaičiuojant tam tikrus programinio kodo raktinius žodžius. Pavyzdžiui, raktiniai žodžiai ***for*** bei ***if*** padidintų sudėtingumą per vienetą. Gaunama nauja metrikos formulė, *1+{raktinių žodžių rastų programiniame tekste skaičius}*. Grįžtant prie pavyzdžio 50-ame paveikslėlyje, matomi minėti raktiniai žodžiai, kurių yra du. Pagal formulę, gaunamas toks pat rezultatas, tačiau šiuo atveju jo apskaičiavimą galima labai paprastai užkoduoti. Tai ir buvo atlikta šio darbo metu. Karkasui baigus darbą ir gavus geriausių individų sąrašą, jų programinis tekstas yra papildomai tiriamas ciklomatinio sudėtingumo metrikos pagalba, o gautas rezultatas pateikiamas vartotojo sąsajoje.



50 pav. *Ciklomatinio sudėtingumo skaičiavimas*

Rezultatai ir išvados

Iš atliktos literatūros apžvalgos bei karkaso kūrimo metu buvo detaliai susipažinta su visais esminiais genetinio programavimo pritaikymo algoritmų paieškai klausimais. Ištirtas su darbo tema glaudžiai susijęs Kenneth E. Kinnear Jr tyrimas, kurio metu gautos žinios padėjo ne tik tiksliau atlikti su programavimu susijusius darbus, bet ir leido palyginti gaunamus rezultatus. Deja, daugiau panašia tema aprašytų šaltinių rasti nepavyko, iš ko galima daryti išvadą, kad ši mokslo šaka dar mažai ištirta. Darbo rezultate sukurtas karkasas padės jai tapti populiaresne ir suteiks daugiau informacijos naujiems tyrimams atlikti ateityje.

Darbo metu sukurtas programinis karkasas, kuris leidžia iš tuščio programinio kodo gauti pilnai veikiantį algoritmą iškeltai užduočiai išspręsti. Genetinio programavimo procesas, kuris tai įgalina, priklauso nuo programavimo kalbos pasirinkimo. Šiam darbui atlikti panaudota C# programavimo kalba, o pasirinkimas pagrįstas atliktais programavimo kalbų palyginimais bei asmenine patirtimi. Papildomai atliktas programavimo kalbos gramatikos tyrimas. Sukurtas detalus C# programavimo kalbos klasių sąrašas, imituojantis esmines gramatikos taisykles. Trumpo tyrimo metu buvo surinktos C# kalbos gramatikos taisyklės ir jų pagalba aprašytas paprastas rikiavimo algoritmas. Atliktų darbų rezultatai leido iškelti teigiamą prielaidą - kadangi iš tuščio programinio kodo gramatinių taisyklių pagalba įmanoma sukurti pilnai veikiantį algoritmą, galima patvirtinti, kad atsitiktinumu grįstos genetinio programavimo paieškos metu, algoritmas tikrai turi tam tikrą tikimybę rasti sprendinį. Sukurto karkaso pagalba pavyko sėkmingai patvirtinti iškeltą prielaidą – buvo **gauti standartiniai paieškos ir rikiavimo algoritmai** bei jų programinis kodas, kuris sėkmingai atlieka pateiktas užduotis ir tinkamumo įvertinimu neatsilieka nuo literatūroje pateiktų algoritmų. Taigi, genetinio programavimo metodais gauti algoritmai sugeba efektyviai ir teisingai išspręsti pateiktas užduotis. Jų programinis kodas buvo gautas genetinių operatorių pagalba plečiant minimalų programos vienetą.

Sukurtas programinis karkasas yra lankstus ir gali būti pritaikomas kitų algoritmų paieškai. Automatizuotas kodo kompiliavimas, duomenų perdavimas bei atsakymo tikrinimas yra nesusietas su konkrečiu algoritmu, todėl uždavinys gali būti keičiamas pagal poreikį, o sudėtingesnių algoritmų paieškai užtenka koreguoti tinkamumo funkcijos pritaikymą. Darbo metu buvo ištirti keli esminiai aspektai, susiję su genetinio programavimo taikymu įvairioms užduotims. Pirmiausia, apibrėžtas pradinis minimalios programos vienetas. Po to detalizuotos pagrindinės genetinio algoritmo sąvokos – genas, populiacija, karta, tinkamumas. Kiekvienai iš jų pritaikytas literatūra grįstas projektavimo sprendimas bei įgyvendintas programinio kodo kūrimas. Detaliai aptarti genetiniai operatoriai bei jų veikimo principai, esminiai sprendimai ir jų pritaikymo sėkmė

tyrimuose, kur genetinis programavimas ar genetinis algoritmas leido išspręsti sudėtingas problemas. Kiekvienas operatorius sukurtas remiantis gautais sprendimais, o parašytas kodas išanalizuotas ir patikrintas automatinių testų pagalba. Galiausiai, remiantis tuo grįstais tyrimais, detalizuota ir įgyvendinta genetinio tobulėjimo sąvoka. Įgyvendinimas remiasi šio darbo metu sukurtu genetiniu algoritmu, supaprastinant kelias perteklines funkcijas. Taip pat padaryti šie principiniai darbo sprendimai:

- Remiantis literatūra ir atliktu tyrimu, darbams atlikti pasirinkta C# programavimo kalba.
- Karkaso kūrimui pasirinktas **.Net Framework 4.6.1** programinis paketas, kuris palaiko C# 7.0 programavimo kalbos versiją.
- Įgyvendinta automatizuoto kompiliavimo sistema, naudojanti ***System.CodeDom.Compiler*** bibliotekas.
- Pasirinktas minimalus pradinės programos, kurioje bus vystomas algoritmas, vienetas.
- Apibrėžta geno struktūra.
- Pasirinkta ir įgyvendinta tinkamumo funkcija.
- Apibrėžtos salos bei individų migracijų tarp jų procesas.
- Nustatyta genetinių operatorių pritaikymo tvarka.
- Apibrėžtas galutinis genetinis algoritmas su visais darbe reikalingais plėtiniais.
- Pasirinktas ir pritaikytas genetinio tobulėjimo procesas.
- Sudaryta lengvai plečiama programinių klasių ir projektų struktūra.
- Pasirinktas mašininio mokymosi procesas karkaso parametrams optimizuoti.

Pilnai optimizuotas karkaso darbas visų parametrų atžvilgiu. Sukurtas genetinio programavimu paremtas karkasas įneša didelį skaičių optimizuojamų parametrų. Aptartas ir įgyvendintas tokios problemos sprendimo būdas, kuris buvo pritaikytas praktikoje. Sudarytas pilnas optimizuojamų parametrų sąrašas bei gautas geriausią rezultatą duodantis reikšmių rinkinys. Karkaso optimizavimas atliktas, pritaikant mašininio mokymosi principus, kurių įgyvendinimui iš anksto pasiruošta, projektuojant klasių diagramas bei projektų sąsajas. Surastas ir pritaikytas pagalbinis C# programavimo kalbos kodas, įgyvendinantis vėsinimo imitacijos algoritmą. Taip pat, aptarti galutinių algoritmų kodo vertinimo būdai ir kokios metrikos literatūroje dažniausiai naudojamos, kuo jos ypatingos. Viena iš pasirinktų metrikų, ciklomatinis sudėtingumas, buvo pritaikyta šio darbo kontekste. Darbo metu panaudoti visi įmanomi skaičiavimo resursai kuo didesnei individų aibei ištirti bei atlikta pilna gautų algoritmų analizė, pritaikant jau naudotas programinio kodo teksto metrikas bei aptartą ciklomatinio sudėtingumo metriką. Atlikto optimizacijos proceso rezultatai leidžia teigti, kad sukurtas karkasas dirba optimaliai.

Pasiektas šio darbo tikslas – įgyvendintas karkasas galintis rasti ne tik gerėjančius, bet ir naujus ar geresnius už dabar žinomus, algoritmus. Ištirti literatūros šaltiniai įnešė daug naudingų žinių ir idėjų kuriamo karkaso projektavimui. Turimo programinio projekto pagalba, buvo sukurtas pilnai veikiantis algoritmų paieškos karkasas. Atlikti visi šio darbo uždaviniai. Apžvelgti su tema susiję darbai, atlikta programavimo kalbų analizė ir pasirinkta konkreti programavimo kalba. Genetinis algoritmas pritaikytas konkrečiai problemai spręsti bei ištirti ir įgyvendinti vidiniai kodo generavimo būdai. Atliekant karkaso kūrimą, buvo atsižvelgta į visus aprašytus ir literatūra grįstus sprendimus. Sukurtame karkase įgyvendinti visi genetiniai operatoriai bei atliktas jų derinimas. Optimizuotas karkaso parametrų sąrašas bei rasti keli esminiai informatikoje dažnai naudojami algoritmai, įrodant sukurto karkaso korektiškumą bei efektyvumą. Literatūroje išdėstyti ir pagrįstai pasirinkti sprendimai padėjo sukurti genetiniu programavimu paremtą algoritmų paieškos procesą, kuris išvysto ne tik korektiškus, bet ir su kiekviena iteracija tobulėjančius algoritmus.

Tęsiant darbus ateityje, sukurtas programinis karkasas gali būti pritaikytas kitoms uždavinių sritims. Vystomo algoritmo kodo sudėtingumą galima pakeisti pridedant daugiau gramatikos taisyklių ir atnaujinant pradinį programos vienetą su reikalingomis bibliotekomis. Karkasas vykdo mašininį mokymąsi ir kuria algoritmus, kurių tinkamumas priklauso nuo iškelto uždavinio sprendimo. Geriausiai užduotį atliekantis programinis kodas kyla į viršų. Išvardintos funkcijos leidžia suinteresuotoms šalims laisvai panaudoti šiame darbe sukurtą produktą ir idėjas įvairiems tyrimams atlikti, toliau tęsti algoritmų paiešką, plėtoti genetinį programavimą**. Tikimasi, kad šio darbo rezultatai įrodė pasirinkto sprendimo būdo tinkamumą ir lankstumą įvairiems algoritmų paieškos uždaviniams spręsti.**

Literatūros sąrašas

[Cha06] Assaad Chalhoub. *Simulated Annealing Example in C#*. Code Project, Birželio 30 d., 2006 [žiūrėta 2019m. Balandžio 27 d.]. Prieiga per internetą < https://www.codeproject.com/Articles/13789/Simulated-Annealing-Example-in-C>

[Fil12] Vladimir Filipović. *Fine-grained tournament selection operator in genetic algorithms*. Computing and Informatics 22.2, 2012, p. 143-161 [žiūrėta 2019m. Balandžio 27 d.]. Prieiga per internetą <http://www.cai.sk/ojs/index.php/cai/article/viewFile/452/360>

[GL08] Volker Gruhn, Ralf Laue. *Complexity Metrics for Business Process Models*. Computer Science Faculty, University of Leipzig, Vokietija, 2006 [žiūrėta 2019m. Balandžio 27 d.]. Prieiga per internetą < https://www.researchgate.net/publication/221281564_Complexity_Metrics_for_business_Process_Models>

[HDR16] Abdelhalim Hiassat, Ali Diabat, Iyad Rahwan. *A genetic algorithm approach for location-inventory-routing problem with perishable products*. Journal of Manufacturing Systems, 2016, p. 93–103 [žiūrėta 2019m. Balandžio 27 d.]. Prieiga per internetą: < http://www.ttcenter.ir/ArticleFiles/ENARTICLE/3388.pdf>

[Jar12] Jiri Jaros. *Multi-GPU Island-Based Genetic Algorithm for Solving the Knapsack Problem*. WCCI 2012 IEEE World Congress on Computational Intelligence. Canberra, ACT, Australia, 2012, p. 217-224 [žiūrėta 2019m. Balandžio 27 d.]. Prieiga per internetą: <http://www.fit.vutbr.cz/research/pubs/index.php.en?file=%2Fpub%2F9860%2F328.pdf&id=9860>

[Jon88] Kenneth De Jong. *Learning with Genetic Algorithms: An Overview*. Machine Learning · October 1988, 3. Computer Science Department, George Mason University. Fairfax, VA 22030. U,S,A., 1988, p. 121-138 [žiūrėta 2019m. Balandžio 27 d.]. Prieiga per internetą: < https://www.researchgate.net/profile/Kenneth_De_Jong/publication/226376117_Learning_with_Genetic_Algorithms_An_Overview/links/0fcfd5109557343353000000.pdf>

[Kin93] Kenneth E. Kinnear, Jr. *Evolving a Sort: Lessons in Genetic Programming*. Proceedings of the 1993 International Conference on Neural Networks, New York. Boxboro, MA 01719 USA, 1993 [žiūrėta 2019m. Balandžio 27 d.]. Prieiga per internetą: < https://pdfs.semanticscholar.org/18f8/5a84f64ba3427b449df9297422ef7ea6b7dc.pdf >

[KMM+09] Kamaljit Kaur, Kirti Minhas, Neha Mehan, Namita Kakkar. *Static and Dynamic Complexity Analysis of Software Metrics*. International Journal of Computer, Electrical, Automation, Control and Information Engineering Vol:3, No:8, 2009, p. 1936-1938 [žiūrėta 2019m. Balandžio 27 d.]. Prieiga per internetą: < https://pdfs.semanticscholar.org/ec8b/cd8fc681bad42bcc88d657b0e24dbb46c40c.pdf >

[Koj14] Lumír Kojecký. *Compiling C# Code at Runtime*. 2014 [žiūrėta 2019m. Balandžio 27 d.]. Prieiga per internetą <https://www.codeproject.com/Tips/715891/Compiling-Csharp-Code-at-Runtime>

[LH13] William B. Langdon, Mark Harman. *Optimising Existing Software with Genetic Programming*. IEEE Transactions on Evolutionary Computation, 2013 [žiūrėta 2019m. Balandžio 27 d.]. Prieiga per internetą: <http://discovery.ucl.ac.uk/1413298/3/Langdon_2013_ieeeTEC_1.pdf>

[LV01] Ralf Lämmel, Chris Verhoef. Semi-automatic grammar recovery. Software: Practice and Experience 31.15, 2001, p. 1395-1438 [žiūrėta 2019m. Balandžio 27 d.]. Prieiga per internetą <https://pdfs.semanticscholar.org/05ea/a5bab12bb06b08a4e7c9e57d60db3d85094f.pdf>

[MF10] Laurent Magnier, Fariborz Haghighat. *Multiobjective optimization of building design using TRNSYS simulations, genetic algorithm, and Artificial Neural Network*. Department of Building, Civil and Environmental Engineering, Concordia University, Montrealis, Kvebekas, Kanada, 2010, p. 739-746 [žiūrėta 2019m. Balandžio 27 d.]. Prieiga per internetą < https://www.sciencedirect.com/science/article/pii/S0360132309002091>

[Mcc76] Thomas J. McCabe. *A complexity measure*. IEEE Transactions on software Engineering 4, 1976, p. 308-320 [žiūrėta 2019m. Balandžio 27 d.]. Prieiga per internetą <http://juacompe.mrchoke.com/natty/thesis/FrameworkComparison/A%20complexity%20measure.pdf>

[PHL+17] Justyna Petke, Mark Harman, William B. Langdon, Westley Weimer. *Using Genetic Improvement & Code Transplants to Specialise a C++ Program to a Problem Class*. Proceedings of the 17th European Conference on Genetic Programming, EuroGP, Berlin/Heidelberg, Vokietija, 2014, p. 137-149 [žiūrėta 2019m. Balandžio 27 d.]. Prieiga per internetą < http://discovery.ucl.ac.uk/1419638/1/Petke_2014_EuroGP.pdf>

[RG11] Noraini Mohd Razali, John Geraghty. *Genetic algorithm performance with different selection strategies in solving TSP*. Proceedings of the world congress on engineering. Vol. 2. Honkongas, 2011 [žiūrėta 2019m. Balandžio 27 d.]. Prieiga per internetą <https://pdfs.semanticscholar.org/010b/545848cfd29fe6e83987d494fdd00b486229.pdf>

[SA91] William M. Spears, Vic Anand. *A study of crossover operators in genetic programming*. International Symposium on Methodologies for Intelligent Systems. Springer, Berlynas, Vokietija, 1991 [žiūrėta 2019m. Balandžio 27 d.]. Prieiga per internetą <http://www.dtic.mil/get-tr-doc/pdf?AD=ADA294071>

[Ses10] Peter Sestoft. *Numeric performance in C, C# and Java*. IT University of Copenhagen Denmark, Versija 0.9.1, 2010 [žiūrėta 2019m. Balandžio 27 d.]. Prieiga per internetą: <http://www.itu.dk/~sestoft/papers/numericperformance.pdf>

[SW08] Yonghee Shin, Laurie Williams. *An Empirical Model to Predict Security Vulnerabilities using Code Complexity Metrics*. ESEM'08, Kaiserslautern, Vokietija, 2008, p. 315-317 [žiūrėta 2019m. Balandžio 27 d.]. Prieiga per internetą <https://collaboration.csc.ncsu.edu/laurie/Papers/p315-shin.pdf>

[SWL+03] X. H. SHI, L. M. WAN, H. P. LEE, X. W. YANG, L. M. WANG, Y. C. LIANG. *AN IMPROVED GENETIC ALGORITHM WITH VARIABLE POPULATION SIZE AND A PSO-GA BASED HYBRID EVOLUTIONARY ALGORITHM*. Proceedings of the Second International Conference on Machine Learning and Cybernetics, Wan, 2-5, 2003, p. 1735-1740 [žiūrėta 2019m. Balandžio 27 d.]. Prieiga per internetą: < https://www.researchgate.net/profile/Xiaowei_Yang2/publication/260286132_An_improved_genetic_algorithm_with_variable_population-size_and_a_PSO-GA_based_hybrid_evolutionary_algorithm/data/02e7e5307f504d1ae6000000/An-improved-genetic-algorithm-with-variable-population-size-and-a-PSO-GA-based-hybrid-evolutionary-algorithm.pdf>

[Zay] Vadim Zaytsev. *Semi-automatic Recovery of a C# Grammar*. Department of Information Management and Software Engineering, Amsterdamas, Nyderlandai [žiūrėta 2019m. Balandžio 27 d.]. Prieiga per internetą <http://www.academia.edu/2861835/Semi-automatic_Recovery_of_a_C_Grammar>