VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
STUDIJŲ PROGRAMA: INFORMATIKA

Automatinis programinio kodo testavimas

Automated Testing of Software Source Code

Baigiamasis magistro darbas

Atliko: Aurimas Petrėtis

VU e.paštas: aurimas.petretis@mif.stud.vu.lt

Darbo vadovas: lekt. Irmantas Radavičius

Recenzentas: prof. dr. Saulius Gražulis

Vilnius – 2023

Santrauka

Mutaciniu testavimu vadinamas baltosios dėžės testavimo metodas, kuris naudoja mutavimo atitikimo įverčio kriterijų. Mutavimo atitikimo įvertis aprašomas kaip sunaikintų ir neekvivalenčių kodų santykis. Naudojant mutacinį testavimą yra susiduriama su ekvivalenčių mutavusių kodų aptikimo problema, kuri yra laikoma neišsprendžiama. Todėl ekvivalenčių mutavusių kodų aptikimui yra taikomi įvairūs metodai, pateikiantys apytikslį atsakymą, įskaitant neuroninius tinklus. Literatūroje dažniau sprendžiamas kodų klonų aptikimo uždavinys, kurio tikslas – rasti kodų klonų poras. Kodų klonų aptikimo uždavinys yra klasifikuojamas į kelias pagrindines metodų grupes: tekstiniai metodai, leksiniai metodai, paremti medžiu metodai, paremti metrikomis metodai, semantiniai metodai ir hibridiniai metodai. Šiame darbe pasirinkta nagrinėti paremtą medžiu metodą, taikomą kartu su metrikų metodu. Šie modeliai taip pat pritaikyti ir ekvivalenčių mutavusių kodų aptikimo uždaviniui. Darbo tikslas: sukurti mutacinio testavimo prototipą naudojant kombinuotą ekvivalenčių mutavusių kodų aptikimo modelį. Modeliams mokyti buvo sukurtas mutavusių kodų duomenų rinkinys, kuris yra didžiausias viešai prieinamas mutavusių kodų rinkinys ir vienintelis, nagrinėjantis kodus metodų lygyje. Darbe suprojektuota modelių kombinacija, kodų poras pažyminčias kaip ekvivalenčias, jeigu nors vienas iš modelių ją taip identifikavo. Kombinuotas modelis buvo pritaikytas mutacinio testavimo procesui, kuris parodė, kad ekvivalenčių mutavusių kodų aptikimo modelio taikymas pagerina mutavimo atitikimo įvertį. Mutaciniam testavimui pritaikius genetinį algoritmą gautas tikslesnis mutavimo įvertis su daugiau identifikuotų neekvivalenčių kodų.

Raktiniai žodžiai: Mutacinis testavimas, neuroniniai tinklai, kodų klonų aptikimas, abstrakčios sintaksės medžio modelis, metrikų modelis, hibridinis modelis, ekvivalenčių mutavusių kodų aptikimas

Summary

Mutation testing is called white-box testing method, which uses the criterion of mutation adequacy. The mutation adequacy score is described as the ratio of killed mutants to non-equivalent code. Mutation testing has a problem of detecting equivalent mutants, which is considered unsolvable. Therefore, various methods, including neural networks, are used to detect equivalent mutants, providing an approximate result. The problem of detecting code clones is often addressed in literature, the purpose is find pairs of code clones. The problem of code clone detection is classified into several main groups of methods: textual methods, lexical methods, tree-based methods, metric-based methods, semantic methods, and hybrid methods. In this work, a tree-based method was chosen to be examined in combination with a metric-based method. These models are also suitable for the task of detecting equivalent mutants. The goal of this work is to create a prototype of mutation testing using a combined model for detecting equivalent mutants. A dataset of mutants was created to train the models, which is the largest publicly available dataset of mutants and the only one that examines codes at the method level. A combination of models was designed in this work, marking code pairs as equivalent if at least one of the models identified them as such. The combined model was applied to the mutation testing process and showed an improvement in mutation adequacy score. Genetic algorithm applied to mutation testing resulted in a more accurate mutation score with a greater number of identified non-equivalent code.

Keywords: Mutation Testing, Neural Networks, Code Clone Detection, Abstract Syntax Tree Model, Metrics Model, Hybrid Model, Equivalent Mutant Detection

Turinys

Įvadas

Programinio kodo testavimas yra procesas arba procesų seka, skirta įsitikinti, kad programinis kodas atlieka tai, ką turėtų atlikti, ir nedaro nieko, kas nėra numatyta. Sistema turėtų būti nuosekli ir nuspėjama, nesukelti netikėtumų vartotojui. Yra daug įvairių programavimo sričių ir daugeliui jų reikalingas testavimas, kad būtų užtikrinta realizuoto kodo kokybė. Į jas įeina svetainių programavimas (angl. *web programming*), socialinių tinklalapių kūrimas, mobiliųjų aplikacijų kūrimas, kurie yra kuriami naudojant servisų technologijas [MSB12].

Programinio kodo testavimą galima suvokti kaip rizikos valdymo veiklą. Didėjant testavimo atvejų skaičiui, didėja testavimui skiriamas laikas, tačiau mažinama klaidų rizika, todėl testuotojui tenka surasti optimalų tašką, kuriame klaidų užsilikimo poveikis tampa nebereikšmingu.

Testavimas vykdomas naudojantis baltosios dėžės (angl. *white box*), juodosios dėžės (angl. *black box*) arba pilkosios dėžės (angl. *grey box*) metodais. Baltosios dėžės testavimo metodai remiasi įgyvendintų funkcionalumų realizacija ir yra efektyvūs, nes jos kartu ištestuoja programos vidinę struktūrą, tačiau reikalauja ir gerų programavimo įgūdžių. Juodosios dėžės testavimo metodas yra paprasčiausias programų testavimo būdas, kurio metu testuojamas programos veikimas nežinant jos realizacijos. Pilkosios dėžės testavimo atveju yra kombinuojami baltosios dėžės ir juodosios dėžės testavimo principai, kai testuotojas atsižvelgia į programos vidinę struktūrą atlikdamas testavimą [JAA+16].

Egzistuoja daug testavimo įrankių. Juos galima klasifikuoti pagal paskirtį: tai gali būti įrankiai, skirti testuoti taikomąją programinę įrangą (angl. *application software*), duomenų bazes, žiniatinklio programas (angl. *web application*), ryšio protokolus ir t. t., iš kurių gausiausia yra žiniatinklio programų testavimo įrankių grupė [MAM09].

Kritiškoms sistemoms, kuriose klaidos kainuoja daug ir įprasto testavimo su testinių atvejų rinkiniu gali neužtekti klaidoms padengti, yra naudojamas modelių verifikavimas. Tačiau sistemai verifikuoti reikia skirti daug lėšų. Siekiant sumažinti resursų kainą, galima modelių verifikavimą ir testavimą kombinuoti siekiant geresnės programos kokybės ir sutaupyti lėšų [Car02].

Yra laikoma, kad siekiant užtikrinti aukštą programinio kodo kokybę, tipinio programavimo projekto testavimas užima nemažą projekto plėtojimo dalį, todėl yra ieškoma būdų automatizuoti šį procesą. Automatizuoti galima įvairius testavimo etapus: nuo testinių atvejų parinkimo, testavimo scenarijų aprašymo iki testuotojų informavimo apie testų rezultatus. Tačiau svarbu yra pasirinkti tinkamą kryptį, kurią norima automatizuoti siekiant, kad automatizavimas atsipirktų [GM16].

Automatizuojant testavimo atvejų rinkinių sudarymą, šis sąrašas gali būti generuojamas automatiškai. Yra įvairių metodų, kuriais šis generavimas pasiekiamas. Vienas iš testinių atvejų generavimo būdų yra mutacinis testavimas. Mutaciniu testavimu vadinamas baltosios dėžės testavimo metodas, kuris naudoja mutavimo atitikimo įverčio (angl. *mutation adequacy score*) kriterijų. Mutacinis testavimas vykdomas generuojant kodo mutavusius kodus, kur mutavusiu kodu yra vadinamas programinis kodas, gautas atlikus pradinio kodo pakeitimą imituojant klaidą ar klaidas. Čia mutavimo atitikimo įvertis apibrėžiamas kaip sunaikintų ir neekvivalenčių mutavusių kodų santykis, čia sunaikintu mutavusiu kodu vadinamas kodas, kuriam vienas iš testavimo atvejų grąžino neteisingą išvestį. Mutacinis testavimas gali būti naudojamas tikrinant testų rinkinio efektyvumą

testų rinkiniui apskaičiuojant mutavimo atitikimo įvertį. Tokiu būdu aptinkamos mutavusiame kode įveltos klaidos, kurios reprezentuoja potencialias programuotojo galimas padaryti klaidas [JH11].

Paieška paremtame programinio kodo testavime (angl. *search based software testing*) testavimo atvejų rinkiniams parinkti reikalingas ir pasirinktas metaeuristinis optimizavimo metodas, kuris randa pateiktos problemos euristinį sprendinį. Juos taikant svarbu pasirinkti efektyvią tinkamumo funkciją (angl. *fitness function*). Dažniausiai darbuose testavimo atvejams generuoti taikytas genetinio algoritmo metodas, tačiau yra ir daugiau metodų, kuriuos galima pritaikyti šiai sričiai [SRS17].

Mutacinis testavimas aktualus ir šiomis dienomis. Mutacinis testavimas naudojamas, kai neužtenka kitų naudojamų įrankių, pvz., kodo padengimas[1] (angl. *code coverage*), ir reikia griežtesnio kodo kokybės įvertinimo papildant kodo testavimą alternatyviais metodais. Tačiau naudojant mutacinį testavimą, susiduriama su problemomis, dėl kurių jis nėra plačiau naudojamas. Pirma, mutacinis testavimas yra brangus laiko ir atminties atžvilgiu. Turint projekto kodą su milijonais kodo eilučių, mutavusių kodų generavimas visam kodui be naudojamų mutavimo operatorių atrinkimo yra neįmanomas. Pavyzdžiui, *Google* programinio kodo bazė užima iš viso 2 milijardus eilučių. Kiekvieną dieną įvykdoma 40 000 kodo pakeitimo paraiškų, kuriems taikant tradicinį mutacinį testavimą prireikia prasukti 150 milijonų testų, kurie labai stabdo kodo naujinimo procesą [PIF+22]. Todėl mutacinis testavimas praktiškai dažniau pritaikomas naujai parašomam kodui atliekant jo testavimą arba kodo peržiūrą, kur naujai parašytas kodas ištestuojamas su nedideliu mutavimo operatorių skaičiumi.

Antra problema, su kuria susiduriama naudojant mutacinį testavimą, yra ekvivalenčių mutavusių kodų aptikimo problema, kuri yra laikoma neišsprendžiama, t. y. neegzistuoja algoritmas, galintis nustatyti, ar mutavusių kodų pora yra ekvivalenti [BA82]. Mutacinio testavimo tikslas yra įvertinti testavimo atvejų rinkinio efektyvumą bandant sunaikinti mutavusius kodus. Tačiau jei sugeneruojamas mutavęs kodas su visais testavimo atvejais grąžina tą patį rezultatą kaip originali programa, jis laikomas ekvivalenčiu mutavusiu kodu ir jo neįmanoma sunaikinti. Dėl šios priežasties mutacinis testavimas negali pateikti patikimų rezultatų ir yra ne taip dažnai naudojamas [MOT+14].

Kadangi ekvivalenčių mutavusių kodų aptikimo problema yra neišsprendžiama, jai kuriami įvairūs euristiniai algoritmai. Galimi ekvivalenčių mutavusių kodų aptikimo metodai yra kompiliatoriaus optimizavimo metodai, matematiniai apribojimai, programų pjaustymas į dalis lengvesniam mutavusių kodų aptikimui, semantinių skirtumų nagrinėjimas, Margravo (angl. *Margrave*) pokyčio poveikio analizė, Lesaro (angl. *Lesar*) modelio tikrinimas (angl. *model-checker*) [Kum15]. Taip pat yra taikomi ir mutavusių kodų generavimo metodai, siekiant išvengti ekvivalenčių mutavusių kodų sugeneravimo. Pagrindiniai tokie metodai yra pasirinktinas mutavimas (angl. *selective mutation*), programų priklausomybių analizė, koevoliucionuojančios paieškos analizė, semantinis trikių (angl. *exception*) hierarchijos nagrinėjimas, aukštesnio lygio mutavimas bei kt. [Kum15].

Taip pat ekvivalenčių mutavusių kodų aptikimo problemai yra taikomi ir mašininio mokymosi metodai. Taikant juos pateikiant duomenų rinkinį yra mokomas neuroninio tinklo modelis, kuriam

[1] Kodo padengimas apibrėžiamas kaip metrika, parodanti, kokia programinio kodo dalis yra padengta automatiniais testais

pateikus programų porą kaip įvestį jis gali išvesti rezultatą, ar tai ekvivalentūs kodai, ar ne.

Literatūroje dažniau sprendžiamas kodų klonų aptikimo (angl. *code clone detection*) uždavinys, kurio tikslas – rasti kodų klonų poras. Šio uždavinio tikslas – nustatyti, ar du kodo fragmentai yra klonai, ar ne, kur kodo klonas įvardijamas kaip kodo fragmentas, kuris yra toks pat arba panašus į kitą kodo fragmentą toje pačioje arba kitoje programinio kodo sistemoje [Ino21]. Kodų klonai yra skirstomi į keturis tipus: pirmojo, kuris žymi struktūriškai nepakitusius kodus (su pridėtais arba ištrintais tarpais arba komentarais), antrojo, kuris žymi kodus, kurie yra struktūriškai nepakitę, bet su pervadintais kintamųjų bei funkcijų vardais, trečiojo, kuriuose gali būti pridėta, atimta ar pakeista eilutė, bei ketvirtojo tipo, kuris žymi struktūriškai skirtingus kodus [GS16].

Pagal struktūrą ekvivalentūs mutavę kodai yra panašūs į trečiojo tipo kodų klonus, nes mutavusių kodų skirtumas būna pridėta, atimta arba pakeista eilutė, išskyrus tais atvejais, kai yra taikomi struktūriniai mutavimo operatoriai (pvz., sakinio pašalinimas). Tačiau skirtumas tarp šių uždavinių yra toks, kad kodų klonai nebūtinai turi pateikti identišką atsakymą. Uždavinys, kuriuo siekiama nustatyti, ar dvi atsitiktinės programos su ta pačia įvestimi visada grąžina tą patį rezultatą, yra vadinamas ekvivalenčių programų aptikimu.

Kodų klonų aptikimo uždavinys yra klasifikuojamas į kelias pagrindines metodų grupes [ABA+19]: tekstiniai metodai, leksiniai metodai, paremti medžiu metodai, paremti metrikomis metodai, semantiniai metodai ir hibridiniai metodai. Šiame darbe pasirinkta nagrinėti paremtą medžiu metodą, taikomą kartu su metrikų metodais, kadangi literatūroje nėra bandyta taikyti šių idėjų kartu [SRK+21]. Pirmu proceso etapu programinis kodas suvedama į abstraktaus sintaksės medžio struktūrą, kartu su iš šio medžio surinkta ir pateikta informacija apie programos požymius. Vėliau šie modeliai pritaikyti ekvivalenčių mutavusių kodų aptikimo uždaviniui. Parodyta, kad kombinuotas modelis padidina sunaikintų ekvivalenčių mutavusių kodų kiekį.

Darbo tikslas: sukurti mutacinio testavimo prototipą naudojant kombinuotą ekvivalenčių mutavusių kodų aptikimo modelį.

Darbo uždaviniai:

1. Atlikti testavimo srities analizę.
2. Apibrėžti mutacinio testavimo procesą.
3. Sukurti metrikų modelį kodų klonų aptikimo uždaviniui.
4. Apibrėžti kombinuoto modelio procesą kodų klonų aptikimo uždaviniui.
5. Sukurti naują mutavusių kodų duomenų rinkinį ekvivalenčių mutavusių kodų aptikimo modeliams mokyti.
6. Pritaikyti kodų klonų aptikimo modelius mutaciniam testavimui.
7. Įgyvendinti mutacinio testavimo prototipą.
8. Pritaikyti genetinį algoritmą testavimo atvejams generuoti siekiant gauti tikslesnį mutavimo įvertį.

Pirmajame skyriuje pateikta literatūros testavimo tematika apžvalga. Antrajame skyriuje aprašytas mutacinis testavimas, jam taikomi optimizavimo metodai, mutacinio testavimo tobulinimo galimybės ir ekvivalenčių mutavusių kodų problema. Trečiajame skyriuje pateikta neuroninių tinklų analizė. Ketvirtajame skyriuje aprašyta projekto realizacija. Penktajame skyriuje aprašytas kodų klonų aptikimo uždavinio sprendimas, jam taikyti modeliai ir jų kombinacija bei testavimo rezultatai. Šeštajame skyriuje pateiktas mutavusių kodų duomenų rinkinys ir modelių, mokytų ant šio duomenų rinkinio, testavimo rezultatai. Septintajame skyriuje pateiktas mutacinio testavimo prototipas ir ekvivalenčių mutavusių kodų uždavinio sprendimo pritaikymas.

1. Testavimas

Šiame skyriuje pateikta programinio kodo testavimo literatūros apžvalga. Aprašytos testavimo sritys.

1.1. Testavimo sritys

Dauguma testavimo veiklų yra susijusios su testavimo atvejų rašymu. Testavimo atvejis yra aibė išankstinių sąlygų, įvesčių ir laukiamų rezultatų, įgyvendintų siekiant patikrinti, ar testuojamas objektas (angl. *test item*) atitinka iškeltą testavimo tikslą (angl. *test objective*) [ISO22].

Siekiant sumažinti žmogaus testuotojo krūvį testuojant programas galima išskirti šešias testavimo veiklas, kuriose yra daug automatizavimo galimybių:

1. Testavimo atvejų projektavimas (angl. *test-case design*). Testavimo atvejų sąrašo arba testo reikalavimų sąrašo sudarymas kodo padengimo kriterijui tenkinti, kiti techniniai tikslai, kurie gali būti susieti ir su testavimu naudojantis testuotojo patirtimi, pvz., tiriamasis testavimas (angl. *exploratory testing*).

2. Testų scenarijų aprašymas (angl. *test scripting*). Testavimo atvejų aprašymas testavimo scenarijais, kurie vėliau vykdomi paties testuotojo be jokių automatizuotų žingsnių. Taip pat į šią dalį įeina ir automatinių testų generavimas ir automatizavimas siekiant sumažinti testuotojo ranka atliekamo darbo krūvį.

3. Testų vykdymas (angl. *test execution*). Testavimo atvejų vykdymas ir testų rezultatų sekimas.

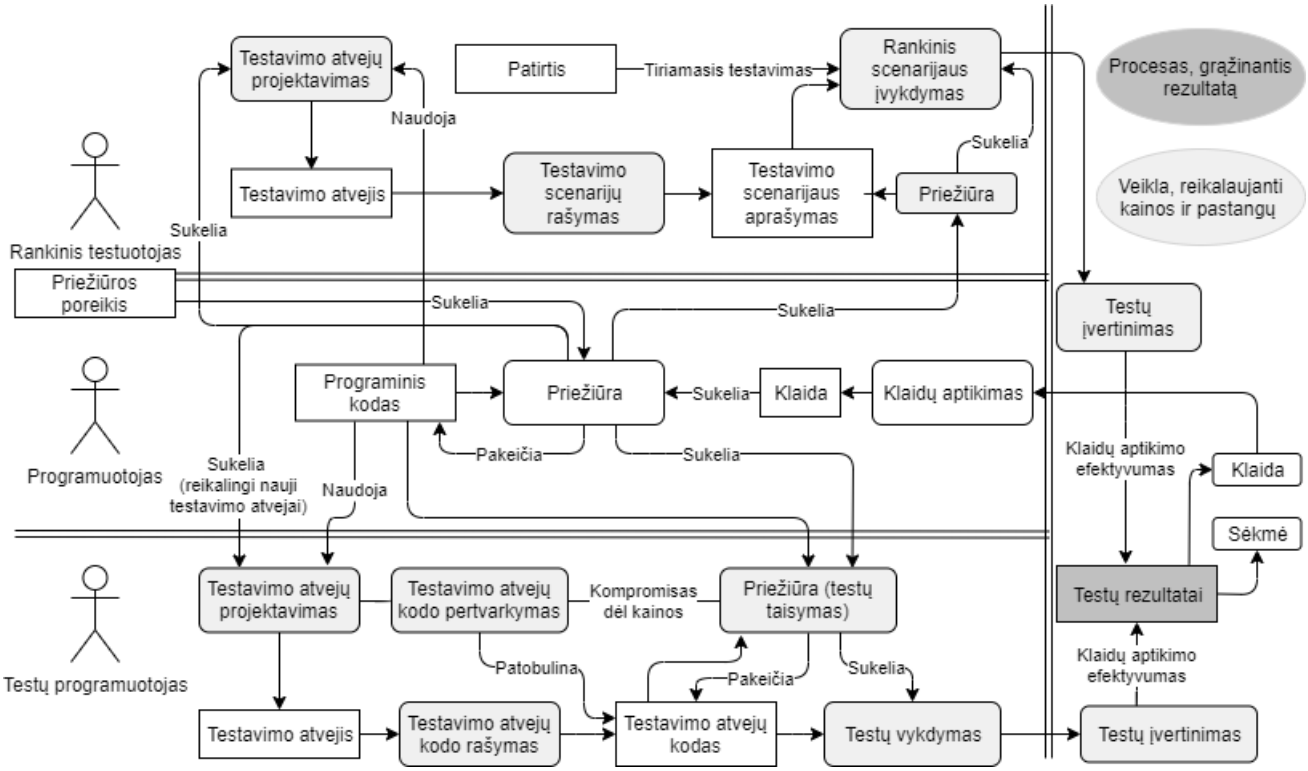
4. Testų rezultatų įvertinimas (angl. *test evaluation*): nustatymas, ar testai įvykdyti sėkmingai (angl. *pass*), ar ne (angl. *fail*).

5. Testų rezultatų perdavimas (angl. *test-result reporting*) programuotojams naudojantis klaidų sekimo sistemomis.

6. Testų valdymas (angl. *test management*) ir kitos techninės veiklos. Į testų valdymą įeina testavimo planavimas, kontrolė, sekimas bei pastangų įvertinimas. Kitos veiklos įtraukia testavimo atvejų rinkinio minimizavimą bei regresinių testų parinkimą [GM16].

1 pav. yra pavaizduota bendra testavimo proceso schema ir testuotojų bei programuotojų atliekamos veiklos, susijusios su testavimu, tokios kaip klaidų aptikimas (angl. *fault localization*) bei testuojamo programinio kodo (TPK) (angl. *software under test (SUT)*) priežiūra. Pirma, testavimo atvejų projektavimo etape yra pagaminami testavimo atvejai, taikant baltosios, juodosios arba pilkosios dėžės testavimo metodus. Šio etapo rezultatas yra testavimo atvejų rinkinys, kuris suformuoja įvestį kitam žingsniui – testų scenarijų aprašymui. Testų scenarijų aprašymo etapo išvestis yra testavimo scenarijai – instrukcijų rinkinys, kuris yra vykdomas testuojant programinį kodą. Kai yra įvykdomi testavimo scenarijai ir sugeneruojami ištestuoto kodo rezultatai, tada, atsižvelgiant į lauktą to testo rezultatą, yra sugeneruojamas verdiktas, ar testas įvykdytas sėkmingai, ar ne, ir pranešamas programuotojui. Taip pat yra reikalinga ir testavimo atvejų rinkinių priežiūros veikla.

Viena to priežasčių yra tai, kad galbūt pasikeitė programos reikalavimai ir atsiranda poreikis papildyti ar pakoreguoti testavimo atvejų rinkinį. Kita galima priežastis – testavimo atvejai pasirodė esantys nekorektiški ir atsiranda poreikis juos taisyti. Automatinio testavimo atveju taip pat dėl šių priežasčių gali tekti taisyti ar pertvarkyti testavimo programinį kodą.



1 pav. Testavimo proceso schema [SGP+14]

1.2. Testavimo atvejų projektavimas

ISO/IEC/IEEE 29119-1 standarte testavimo projektavimas (taip pat žinomas kaip testavimo atvejų projektavimas) apibrėžiamas kaip veikla, sąvokos, procesai ir šablonai, skirti testavimo modeliui sukonstruoti, kurie naudojami testuojamo objekto testų būsenai nustatyti, išvesti susijusius objektus, kurie padengia testinį atvejį ir išvesti pačius testavimo atvejus [ISO22]. Be to, ISO/IEC/IEEE 29119-4 standartas apibrėžia testavimo atvejų projektavimą, numato gaires testavimo sąlygoms, testų padengimo (angl. *test coverage*) objektus bei testavimo atvejus. Čia testavimo sąlyga yra testuojamas objekto aspektas, pvz., funkcija, ypatybė, kokybės atributas ar struktūros elementas, identifikuotas kaip pagrindinis testuojamas objektas. Testų padengimo objektai yra kiekvienos testų sąlygos atributai, kurie gali būti padengiami testavimo metu. Čia taikant testų padengimo kriterijų dažniausiai yra tikrinamas kodo eilučių padengimas testais. Testavimo atvejų rinkinys yra rinkinys išankstinių sąlygų, įvesčių ir laukiamų rezultatų, kuris skirtas nustatyti, ar padengta testuojamo objekto vieta yra suprogramuota teisingai [ISO21]. Taigi, kai yra pasirenkamas testavimo projektavimo sprendimas, testavimo veiklos sėkmė priklauso nuo tinkamo sprendimo pasirinkimo [SAR+17].

Būsenų sprogimo problema (angl. *state explosion problem*) – problema, atsirandanti programos testavimo ar verifikavimo metu, kai didėjant parametrų skaičiui eksponentiškai auga galimų

programos būsenų skaičius. Problema aktualiausia taikant modelio patikrinimo (angl. *model checking*) metodą, taikomą programoms, turinčioms baigtinį būsenų skaičių, verifikuoti [CKN+12]. Šis iššūkis aktualus ir testuojant nuolat kintančią informaciją, kai nuolatinis būsenų augimas ir jų galimų derinių kombinacijos staigiai viršija testavimo vietą. Todėl svarbu pasirinkti tinkamą testinių atvejų projektavimo strategiją, kuri atsirinktų tinkamiausius ir informatyviausius testinius atvejus.

1.3. Testavimo atvejų projektavimo metodai

Visus testavimo atvejų projektavimo metodus galima suskaidyti į tris pogrupius: baltosios dėžės testavimą, juodosios dėžės testavimą ir pilkosios dėžės testavimą.

1.3.1. Baltosios dėžės testavimas

Baltosios dėžės testavimas yra testavimas, kuris apima detalų programinio kodo vidinės logikos ir struktūros nagrinėjimą. Testuotojas turi visišką prieigą prie programinio kodo ir testuotojas turi nagrinėti programinį kodą siekdamas nustatyti klaidą. Taikant šį testavimo metodą testavimo atvejai yra sukuriami remiantis programinio kodo vidine struktūra. Baltosios dėžės testavimas gali būti naudojamas testuoti programinį kodą integracijos, vieneto, sistemos lygiu. Pagrindiniai baltosios dėžės testavimo metodo privalumai:

- Pasinaudojant testavimo atveju galima identifikuoti klaidą programiniame kode.
- Nagrinėjant programinį kodą, galima maksimizuoti kodo padengimo kriterijaus įvertį. Čia kodo padengimo kriterijus gali būti skaičiuojamas kodo eilučių padengimu, kodo sakinių padengimu (angl. *statement coverage*), kodo šakų padengimu (angl. *branch coverage*) ir kt.
- Testuotojas gali tiesiogiai nagrinėti programinį kodą nekomunikuodamas su programuotoju.

Pagrindiniai baltosios dėžės testavimo metodo trūkumai:

- Testavimas brangiai kainuoja, nes reikalauja gerų testuotojo testavimo įgūdžių.
- Jautrus programinio kodo pakeitimams. Atlikus programinio kodo pakeitimus, reikia atnaujinti ir testavimo rinkinį.

Pagrindinės baltosios dėžės testavimo rūšys:

1. Srauto valdymo testavimas (angl. *control flow testing*). Tai struktūrinė testavimo strategija, kuri naudoja programos valdymo srautą siekiant srauto valdymo valdymo grafe (angl. *control graph*) nagrinėti naudojamas briaunas.
2. Šakų testavimas (angl. *branch testing*). Šio testavimo tikslas: patikrinti, ar kiekviena kodo šaka yra įvykdyta bent vieną kartą, taip užtikrinant kodo pasiekiamumą. Testavimo išvestys visais atvejais turi atitikti lauktą rezultatą.

3. Pagrindinio programos veikimo kelio testavimas (angl. *basis path testing*). Šis testavimas leidžia testuotojui susikurti procedūrinio dizaino loginio sudėtingumo įvertį ir toliau naudoti šį įvertį siekiant įvardyti pagrindinius programos veikimo kelius.

4. Duomenų srauto testavimas (angl. *data flow testing*). Taikant šį testavimą, srauto valdymo grafas yra pažymėtas informacija apie programos kintamuosius ir jų naudojimą.

5. Ciklo testavimas (angl. *loop testing*). Šis testavimo metodas išskirtinai tikrina programinio kodo ciklų korektiškumą. Jo metu siekiama atpažinti begalinius ciklus, atpažinti neinicializuotus kintamuosius.

1.3.2. Juodosios dėžės testavimas

Juodosios dėžės testavimas yra testavimas, kurio metu yra žinoma sistemos architektūra, tačiau neturima prieigos prie vidinio programinio kodo ir yra koncentruojamasi į esminių sistemos aspektų testavimą. Pagrindiniai baltosios dėžės testavimo metodo privalumai:

- Efektyvu taikyti testuojant sistemą, turinčią daug programinio kodo.
- Sistema paprastai suprantama testuotojo.
- Atskirtos vartotojo ir programuotojo perspektyvos, naudojantis programa.
- Greita testavimo atvejų kūrimo veikla.

Pagrindiniai juodosios dėžės testavimo metodo trūkumai:

- Limituotas ištestuotas testavimo scenarijų skaičius.
- Testavimo atvejams kurti reikalinga aiški specifikacija.
- Neefektyvus klaidų identifikavimas testuojant.

Pagrindinės juodosios dėžės testavimo rūšys:

1. Ekvivalentus skaidymas (angl. *equivalence partitioning*). Naudojamas siekiant sumažinti testavimo atvejų skaičių, klasifikuojant testavimo atvejų grupes, iš kurių yra kuriami testavimo atvejai.

2. Kraštinių reikšmių analizė (angl. *boundary value analysis*). Testuojama koncentruojantis į kraštines reikšmes (minimumas, maksimumas, vidines ir išorines kraštines reikšmes, klaidų reikšmes ir tipines reikšmes (angl. *typical values*)).

3. Neteisingos įvesties testavimas (angl. *fuzzing*). Šis testavimas naudojamas ieškoti įgyvendinimo klaidų automatinėje arba pusiau automatinėje sesijoje įterpiant pažeistus duomenis.

4. Priežasties-pasekmės grafo metodas (angl. *cause-effect graph*). Šis testavimas vykdomas sukuriant grafą, kuriame atsispindi priežasčių ir pasekmių ryšiai naudojantis loginėmis operacijomis.

5. Ortogonalių masyvų testavimas (angl. *orthogonal array testing*). Šis testavimas gali būti naudojamas kaip alternatyva išsamiam testavimui ir imamas poaibis visų galimų įvesties kombinacijų ir naudojamas vietoj išsamaus testavimo būdo (angl. *exhaustive testing*), kai testavimo atvejų įvesties dydis būna nedidelis, tačiau per didelis išsamiam testavimui.

6. Visų porų testavimas (angl. *all pair testing*). Šio testavimo tikslas yra ištestuoti visas galimas kiekvienos įvesties poros kombinacijas, parenkant tokį testavimo atvejų rinkinį, kuris padengtų visas galimas poras.

7. Būsenų pasikeitimo testavimas (angl. *state transition testing*). Šis testavimas naudojamas būsenų pasikeitimui testuoti ir gali būti pritaikomas būsenų mašinai (angl. *state machine*) arba grafinio dizaino navigavimui testuoti.

1.3.3. Pilkosios dėžės testavimas

Pilkosios dėžės testavimas yra testavimas, kurio metu turima dalinė prieiga prie programinio kodo vidinės logikos ir struktūros ir kartu testuojami esminiai sistemos aspektai. Pagrindiniai pilkosios dėžės testavimo metodo privalumai:

- Sujungiami baltosios dėžės ir juodosios dėžės testavimo metodų privalumai.
- Testuotojas gali remtis sąsajos apibrėžimais ir funkcine specifikacija.
- Testuotojas gali sukurti aiškius testavimo scenarijus.
- Testavimo atvejai kuriami iš vartotojo perspektyvos.

Pagrindiniai pilkosios dėžės testavimo metodo trūkumai:

- Lieka daug neištestuotų programos veikimo kelių.
- Gali atsirasti pasikartojančių testavimo atvejų.

Pagrindinės pilkosios dėžės testavimo rūšys:

1. Ortogonalių masyvų testavimas (angl. *orthogonal array testing*).
2. Matricos testavimas (angl. *matrix testing*). Šio testavimo metu sugeneruojama programos būsenų ataskaita.
3. Regresinis testavimas (angl. *regression testing*). Testuojami programiniame kode atlikti pakeitimai, remiantis vykdomais testavimo atvejais.

1.4. Testavimo atvejų generavimas

Testavimas gali būti atliekamas rankiniu arba automatiniu būdu. Testavimas rankiniu būdu yra labiau tradicinis metodas, kurio metu testuotojas paruošia testavimo atvejų rinkinius. Automatizuota testavimo strategija labiau skirta nuobodžiam ir pasikartojančiam darbui automatizuoti pasinaudojant programiniais įrankiais, kurie generuoja testavimo atvejų rinkinius iš programos specifikacijų taikant juodosios dėžės testavimo metodą arba iš programinio kodo taikant baltosios dėžės testavimo metodą [LCM+07].

Penki dažniausiai taikomi testinių atvejų generavimo metodai:

1. Simbolinio vykdymo (angl. *symbolic execution*) ir programos struktūrinio kodo padengimo testavimas (angl. *program structural coverage testing*).
2. Modeliu paremtas (angl. *model-based*) testinių atvejų generavimas.
3. Kombinatorinis testavimas (angl. *combinatorial testing*).
4. Prisitaikantis atsitiktinis testavimas (angl. *adaptive random testing*) (atsitiktinio testavimo atšaka).
5. Paieška paremtas testavimas (angl. *search-based testing*).

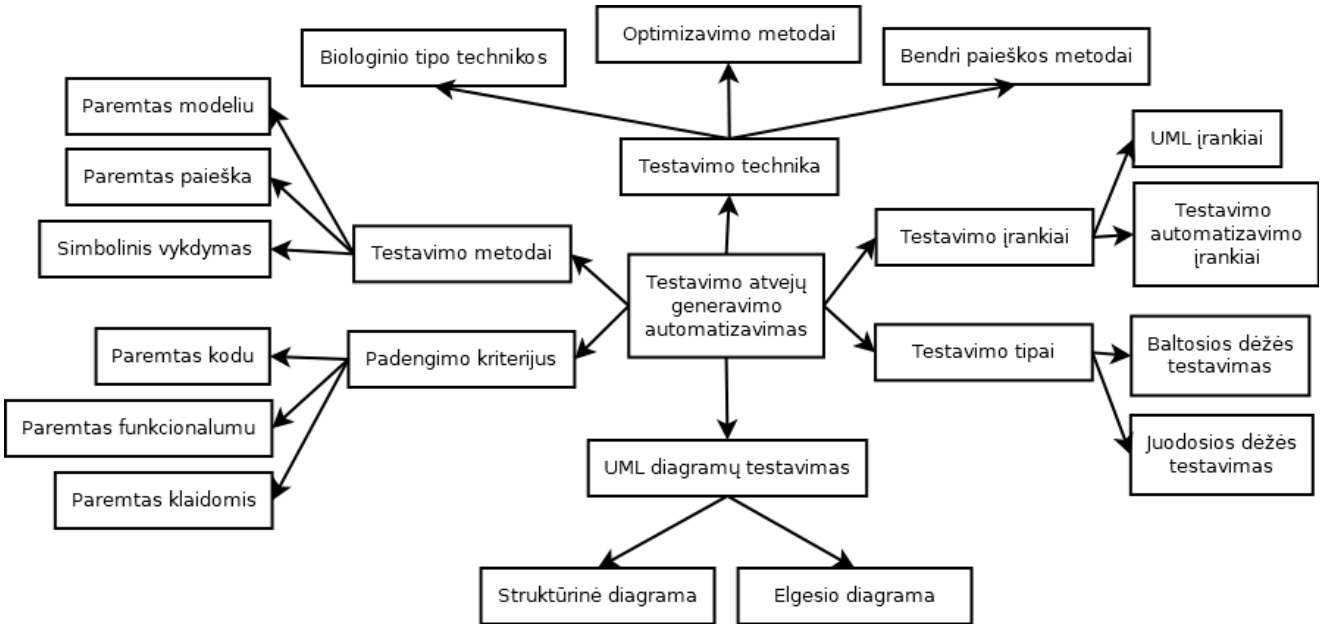
Kiti taikomi testinių atvejų generavimo metodai: mutacinis testavimas (angl. *mutation testing*), neteisingos įvesties testavimas (angl. *fuzzing*), kuriam duomenys testavimui gali būti mutuojami iš esamų duomenų arba generuojami atsitiktiniai nauji duomenys testavimui, specifikacija paremtas testavimas (angl. *specification-based testing*), metamorfinis testavimas (angl. *metamorphic testing*) [ABC+13].

Geriausiam sprendiniui rasti, taikant anksčiau įvardytus testavimo metodus, yra taikomi optimizavimo metodai, kur geriausias sprendinys yra tinkamiausias testavimo atvejų rinkinys. Pagrindiniai taikomi optimizavimo metodai:

- Tabu paieška (angl. *tabu search*).
- Vėsinimo imitacijos metodas (angl. *simulated annealing*).
- Dalelių spiečiaus optimizavimas (angl. *particle swarm optimization*).
- Skruzdžių kolonijos optimizavimas (angl. *ant colony optimization*).
- Genetinis algoritmas (angl. *genetic algorithm*).
- Gegutės paieška (angl. *cuckoo search*).
- Jonvabalių algoritmas (angl. *firefly*).

Be išvardytų optimizavimo metodų, yra ir daugiau metodų, kurie gali būti pritaikomi sprendžiant uždavinį [VP15].

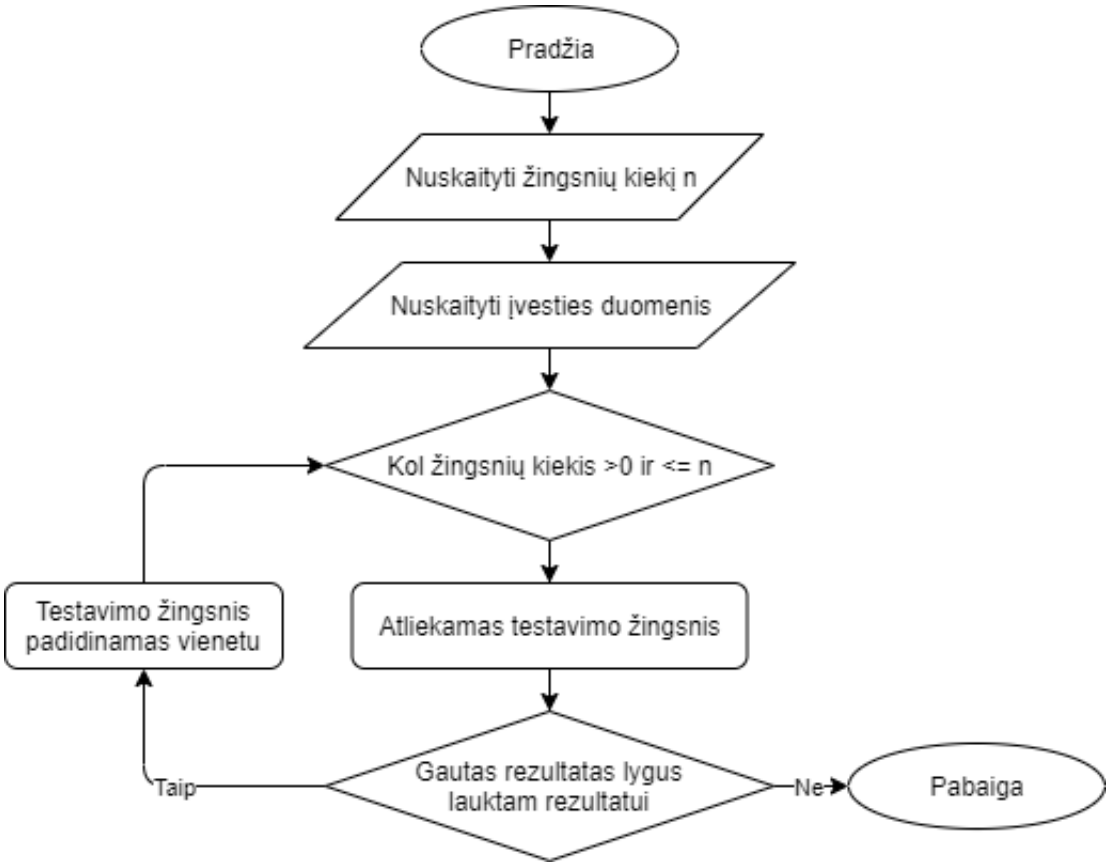
2 pav. yra pateiktas grafas, kokie testavimo atvejų generavimo elementai turi būti apsvarstyti prieš pradedant testavimą.



2 pav. Testavimo atvejų generavimo elementų grafas [VP15]

1.4.1. Testavimo atvejų generavimo struktūrinė schema

3 pav. yra pateiktas bendras testavimo atvejų generavimo procesas. Pirmajame proceso žingsnyje yra pateikiama tinkama įvestis testui vykdyti. Testo rezultatai, gauti vykdant testą, palyginami su laukiamais testo rezultatais ir jei jie sutampa testų vykdymo iteracija tęsiama, kitu atveju stabdoma radus klaidą.



3 pav. Testavimo atvejų generavimo procesas [VP15]

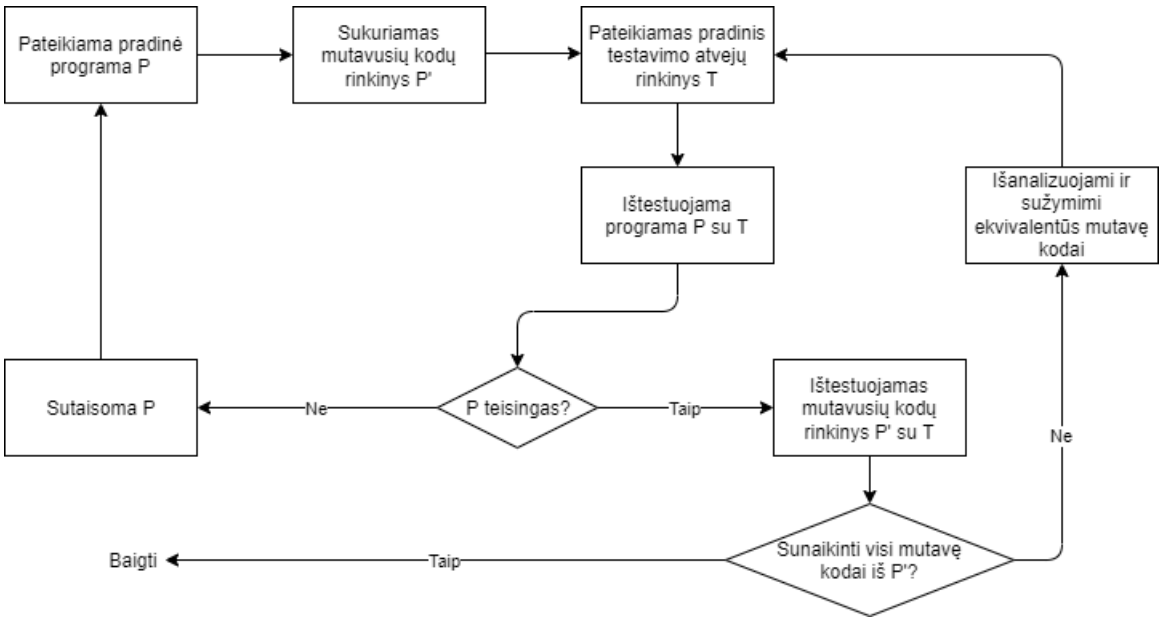
2. Mutacinis testavimas

Toliau darbe tyrinėjama mutacinio testavimo sritis. Šiame skyriuje aprašytas mutacinis testavimas, jam vykdyti taikomas procesas, jo iššūkiai ir ekvivalenčių mutavusių kodų problema.

2.1. Mutacinio testavimo procesas

Mutaciniu testavimu yra vadinama baltosios dėžės testavimo metodas, kuris naudoja mutavimo atitikimo įverčio kriterijų, kuris naudojamas tikrinant testavimo atvejų rinkinio efektyvumą aptinkant programiniame kode paliktas klaidas.

Tradicinis mutacinio testavimo analizės (angl. *mutation analysis*) procesas pateikiamas 4 pav.



4 pav. Mutacinio testavimo analizės procesas [JH11]

Pirmajame žingsnyje iš pradinės programos *p* atlikus sintaksinius pakeitimus yra sukuriamas mutavusių kodų (angl. *mutant*), žymimų *p'*, rinkinys. Transformavimo taisyklė, kuri iš pradinio kodo sugeneruoja mutavusį kodą, vadinama mutavimo operacija (angl. *mutation operator*). Tipinės mutavimo operacijos skirtos modifikuoti kintamojo reikšmę ir išraiškas taikant keitimo, įterpimo bei pašalinimo operatorius.

Kitame žingsnyje sistemai yra pateikiamas testavimo atvejų rinkinys *T* ir patikrinamas pradinio kodo korektiškumas su testavimo atvejais iš rinkinio *T*. Jeigu programa *p* nekorektiška, ji pataisoma ir grįžtama į pirmąjį žingsnį, kitu atveju su testavimo atvejų rinkiniu *T* testuojamas kiekvienas mutavęs kodas *p'*. Jeigu atsiranda testinis atvejis, su kuriuo programa *p* teisinga, o programa *p'* ne, tada laikoma, kad mutavęs kodas *p'* yra sunaikintas (angl. *killed*), kitu atveju laikoma, kad mutavęs kodas *p'* yra išgyvenęs (angl. *survived*).

Kai įvykdomi visi testavimo atvejai, gali būti, kad yra likę keli išgyvenę mutavę kodai. Tokiu atveju testuotojas gali papildyti testavimo atvejų rinkinį *T* savais testavimo atvejais siekdamas sunaikinti likusius mutavusius kodus. Tačiau gali atsirasti mutavusių kodų, kurie negali būti sunaikinti, nes jie grąžina tą patį rezultatą kaip ir pradinė programa su visais įmanomais testavimo

atvejais nepaisant to, kad jie yra sintaksiškai skirtingi. Tokie mutavę kodai vadinami ekvivalenčiais mutavusiais kodais (angl. *equivalent mutant*). Ši problema nėra automatiškai išsprendžiama ir tai pagrindinė priežastis, kodėl mutacinis testavimas nėra plačiai naudojamas.

Mutacinis testavimas užbaigiamas gaunant mutavimo atitikimo įvertį, žinomą kaip mutavimo įvertį (angl. *mutation score*), kuris nusako testavimo atvejo kokybę pateiktam pradiniam kodui. Mutavimo įvertis apibrėžiamas kaip sunaikintų mutavusių kodų ir visų neekvivalenčių mutavusių kodų santykis. Mutacinio testavimo analizės tikslas yra pasiekti mutacinio testavimo įvertį, lygų 1, kuris reikštų, kad testavimo atvejų rinkinys T gali aptikti visas klaidas, pažymėtas mutavusiais kodais [KO91].

2.2. Mutacinio testavimo teorija

Mutacinis testavimas yra efektyvus parenkant tinkamą testavimo atvejų rinkinį, kuris gali būti naudojamas tikroms klaidoms ieškoti. Tačiau galimas klaidų sąrašas yra per didelis, kad būtų galima sugeneruoti mutavusius kodus, padengiančius visas klaidas. Todėl tradiciškai mutacinis testavimas taikomas ieškoti tokio klaidų poaibio, kuris būtų pakankamas imituoti visas galimas klaidas. Ši praktika taikoma remiantis dviem hipotezėmis: kompetentingo programuotojo hipoteze CPH (angl. *competent programmer hypothesis*) ir jungties efektu (angl. *coupling effect*).

Pagal kompetentingo programuotojo hipotezę programuotojai yra kompetentingi, todėl jie geba sukurti programas, artimas teisingai versijai, ir klaidas galima pataisyti keliais nedideliais sintaksiniais pakeitimais. Todėl mutacinio testavimo atveju klaidos konstruojamos iš nedidelių sintaksinių pakeitimų, kurios reprezentuoja kompetentingų programuotojų paliktas klaidas.

Jungties efekto hipotezės esmė ta, kad testavimo duomenys, kurie išskiria teisingas programas iš visų programų su paprastomis klaidomis, yra jautrūs ir netiesiogiai išskiria ir sudėtingesnes klaidas. Offutt [Off89a; Off89b] praplėtė šį teiginį iki jungties efekto (angl. *coupling effect*) hipotezės ir mutavimo jungties efekto hipotezės pateikdamas tikslų paprastų ir sudėtingų klaidų apibrėžimą. Pagal šį apibrėžimą paprasta klaida reprezentuojama paprastu mutavusiu kodu, kuris gaunamas atlikus vieną pradinio kodo sintaksinį pakeitimą, kai sudėtinga klaida reprezentuojama sudėtingu mutavusiu kodu, kuris gaunamas atlikus daugiau nei vieną pradinio kodo sintaksinį pakeitimą. Dėl šios priežasties, kai atliekant mutacinį testavimą identifikuojami visi paprasti kodai, jie kartu padengia ir didelę dalį sudėtingų mutavusių kodų, todėl tradiciškai atliekant mutacinį testavimą yra naudojami tik paprasti mutavę kodai su vieno žingsnio pakeitimais [JH11].

2.3. Mutacinio testavimo optimizavimo metodai

Mutaciniam testavimui optimizuoti yra naudojami optimizavimo metodai, kitaip vadinami metaeuristikomis (angl. *meta-heuristics*). Tai yra paieška paremto programinio kodo testavimo atšaka (angl. *search based mutation testing*). Testavimo atvejų rinkinių generavimo uždavinys gali būti suformuluotas kaip paieškos problema, kurios tikslas yra rasti tinkamą testavimo atvejų rinkinį iš visos testavimo atvejų aibės duotai programai. Dažniausiai optimizavimo metodai pritaikomi testavimo atvejų rinkiniams generuoti, taip pat gali būti pritaikomi ir mutavimo operatoriams

parinkti, mutavusiems kodams generuoti, mutavusiems kodams bei testavimo atvejų rinkiniams generuoti vienu metu [SRS17].

Toliau pateikti pagrindiniai mutaciniam testavimui taikomi optimizavimo metodai.

2.3.1. Genetinis algoritmas

Genetinis algoritmas [MNZ+05] yra dažniausiai taikomas optimizavimo metodas mutaciniam testavimui (1 lentelė). Pagrindinė genetinio algoritmo idėja testavimo atvejams generuoti mutacinio testavimo kontekste yra surasti įvesčių rinkinį, kuris sunaikintų didžiausią mutavusių kodų kiekį. Tam maksimizuojama funkcija *f(x1, x2, ..., xm)*, kur *x1, x2, ..., xm* yra įvesčių kintamieji, kurie turi būti koreguojami siekiant pasiekti globalų maksimumą. Kintamieji yra tradiciškai pateikti kaip bitų eilutė ir ieškomas rezultatas taip pat yra bitų eilutė, kuri atitiktų galutinį įvesčių rinkinį.

Taikant genetinį algoritmą algoritmas įprastai yra aprašomas vartojant individo sąvoką. Atliekant mutacinį testavimą, individas aprašomas kaip įvesčių rinkinys, kuris atitinka konkretų testavimo atvejį. Taikant pirmąjį genetinio algoritmo žingsnį, reikia apibrėžti pradinius individus kaip baigtinį kiekį genų, kurie gali būti pateikti bitų eilutėmis, raidžių eilutėmis ir t. t.. Taip pat apibrėžiama tinkamumo funkcija (angl. *fitness function*), kuri kiekvienam individui grąžina reikšmę, žyminčią individo kokybę atsižvelgiant į sprendžiamą problemą. Mutacinio testavimo kontekste ši funkcija dažniausiai apibrėžiama atsižvelgiant į mutavusių kodų sunaikinimo įvertį, kuris pažymi, kokį skaičių mutavusių kodų sunaikina testavimo atvejų rinkinys.

Taip pat genetiniam algoritmui yra taikomos trys operacijos:

- Atranka (angl. *selection*). Šiame žingsnyje atrenkami testavimo atvejai sunaikinimo, mutacijos bei reprodukcijos operacijoms.
- Reprodukcija (angl. *reproduction*). Šiame žingsnyje taikomas kryžminimo (angl. *crossover*) operatorius, kurį taikant iš atrinktų testavimo atvejų porų yra konstruojami nauji testavimo atvejai, atrinktų testavimų atvejų poroms apsikeičiant tarpusavio informacija atsitiktinėje pozicijoje. Taip sukuriami nauji testavimo atvejai, kurie, laikui bėgant, artėja prie globalaus maksimumo skaičiuojant jų tinkamumo funkciją.
- Mutacija (angl. *mutation*). Mutacijos operatorius pakeičia atrinktų testavimų atvejų atsitiktinės pozicijos reikšmę.

Genetinis algoritmas aprašomas tokia seka:

1. Sukuriamas pradinis testavimo atvejų rinkinys.
2. Įvertinama kiekvieno testavimo atvejo tinkamumo funkcijos reikšmė.
3. Sukuriamas naujas testavimo atvejų rinkinys naudojantis atrankos, reprodukcijos (kryžminimo) bei mutacijos operatoriais.
4. Senas testavimo atvejų rinkinys pakeičiamas nauju testavimo atvejų rinkiniu.
5. Įvertinamas testavimo atvejų rinkinio tinkamumo kriterijus.

6. Jei įvertis nepakankamai geras, grįžtama į antrąjį žingsnį.

2.3.2. Bakteriologinis algoritmas

Bakteriologinis algoritmas (angl. *bacteriological algorithm*) [SRS17] yra genetinio algoritmo adaptacija. Pagrindiniai bakteriologinio algoritmo skirtumai nuo genetinio algoritmo:

- Turi atminties funkciją (angl. *memorization function*).
- Nuslopinta kryžminimo funkcija ir testavimo atvejų žymėjimas.

Eksperimentais įrodyta, kad bakteriologinis algoritmas greičiau konverguoja siekdamas gauti gerą rezultatą.

2.3.3. Lipimo į kalną metodas

Lipimo į kalną (angl. *hill climbing*) metodas [SPL+16] yra lokalus paieškos algoritmas, kuris juda link lokalaus ekstremumo taško iki kol randamas geriausias sprendinys. Pagrindinis lipimo į kalną metodo skirtumas nuo genetinio algoritmo yra, kad lipimo į kalną metodas visada randa lokalų ekstremumą, o genetinis algoritmas randa kažkokį (nebūtinai geresnį) ekstremumą iš visos aibės. Mutacinio testavimo kontekste šiame metode nurodoma tinkamumo funkcija tokia, kad būtų randamas testavimo atvejų rinkinys, sunaikinantis daugiausiai mutantų lokalioje aplinkoje.

Ateityje taikant lipimo į kalną metodą testavimo atvejams generuoti gali būti tobulinama tinkamumo funkcija, metodo kombinavimas su kitais optimizavimo metodais [SPL+16].

2.3.4. Skruzdžių kolonijos metodas

Skruzdžių kolonijos (angl. *Ant Colony*) metodas [ABA07] irgi gali būti taikomas mutaciniam testavimui. Šis metodas imituoja skruzdžių maisto paieškos elgesį. Iš pradžių sugeneruojami testavimo atvejai, o tolesniais žingsniais efektyvūs testavimo atvejai, kurie sunaikina mutavusius kodus, yra pažymimi ir išskiriami iš kitų. Algoritmas veikia grafe, kuriame viršūnės laikomos įvesties parametrais. Skruzdės, vaikščiodamos grafu, po kiekvienos iteracijos pažymi po briauną, taip išsaugant testavimo atvejį, kuris sunaikino mutavusį kodą. Algoritmui bėgant, sugeneruojama vis daugiau testavimo atvejų ir jis stabdomas, kai atitinkamas stabdymo kriterijus. [ABA07] straipsnyje gauti rezultatai, kad skruzdžių kolonijos algoritmas tiek mutavimo įverčio reikšme, tiek skaičiavimų trukme gauna geresnius rezultatus, nei tradiciniai genetinio algoritmo, kopimo į kalną, atsitiktinės paieškos metodai.

Tolesnė skruzdžių kolonijos metodo tyrimo kryptis išlieka testų tinkamumo funkcijos praplėtimas siekiant surasti būtinas ir pakankamas sąlygas mutavusiems kodams sunaikinti. Galima atlikti daugiau patobulinimų pritaikius daugiau kintamųjų tipų ir predikatų išraiškų naudojant tekstines eilutes, masyvus ir loginius kintamuosius.

2.3.5. Vėsinimo imitacijos metodas

Vėsinimo imitacijos metodas [WWZ14] modeliuoja fizikinį procesą, kada yra įkaitinamas objektas ir iš lėto vėsinamas taip minimizuojant sistemos energiją. Tai dar vienas metodas, kuris leidžia iššokti iš lokalios aplinkos ir ieškoti globalaus optimumo. Taikant šį algoritmą tinkamumo funkcijoje yra naudojamas metropolio kriterijus (angl. *metropolis criterion*), kurį taikant galimų sprendinių aibė pamažu mažėja ir susiaurėja iki ekstremumo taško.

Literatūroje šis metodas yra taikytas mutacinio testavimo sričiai. Straipsnyje [WWZ14] autoriai patobulino algoritmą naudodami dvi funkcijas. Pirma, sugeneruotam testavimo atvejų rinkiniui yra skaičiuojama klaidų padengimo funkcija ir jei padengiama daugiau klaidų, nei su ankstesniu testavimo atvejų rinkiniu, tada toliau taikoma sakinių padengimo funkcija. Jeigu sakinių padengimo funkcija grąžina didesnę reikšmę naujam testavimo atvejų rinkiniui, tada naujas testavimo rinkinys išsaugomas ir jis pakeičia seną sprendinį. Pagal straipsnio rezultatus patobulintas vėsinimo imitacijos algoritmas testuojant trikampių nelygybės programą buvo greitesnis už pradinę versiją.

2.3.6. Kiti taikomi optimizavimo metodai

Paieška paremto mutacinio testavimo ir testavimo atvejų generavimo uždaviniui spręsti galima pritaikyti ir kitus metaeuristinius metodus, tokius kaip dirbtinės imuninės sistemos metodas (angl. *artificial immune system*), sąrašo paieška (angl. *enumeration search*), godusis algoritmas (angl. *greedy algorithm*), atsitiktinė paieška (angl. *random search*), tabu paieška (angl. *tabu search*) ir kiti algoritmai (1 lentelė) [SRS17]. Taip pat mutacinio testavimo uždaviniui spręsti galima pritaikyti ir kitus dar nebandytus metodus, pvz., gegutės paieška (angl. *cuckoo search*). Atliekant tolesnius tyrimus, galima tirti mutacinį testavimą, taikant naujus, dar neišbandytus metaeuristinius metodus, taip pat galbūt kombinuoti skirtingus metodus norint pasiekti geresnių rezultatų šioje srityje.

2.3.7. Mutaciniam testavimui taikytų optimizavimo metodų santrauka

1 lentelėje yra pateiktas [SRS17] straipsnyje apžvelgtų paieška paremtų programinio kodo testavimo tyrimų pasiskirstymas nurodytiems taikomiems metaeuristiniams metodams. 2 lentelėje pateikti 1 lentelėje naudojamų metaeuristinių algoritmų sutrumpinimai.

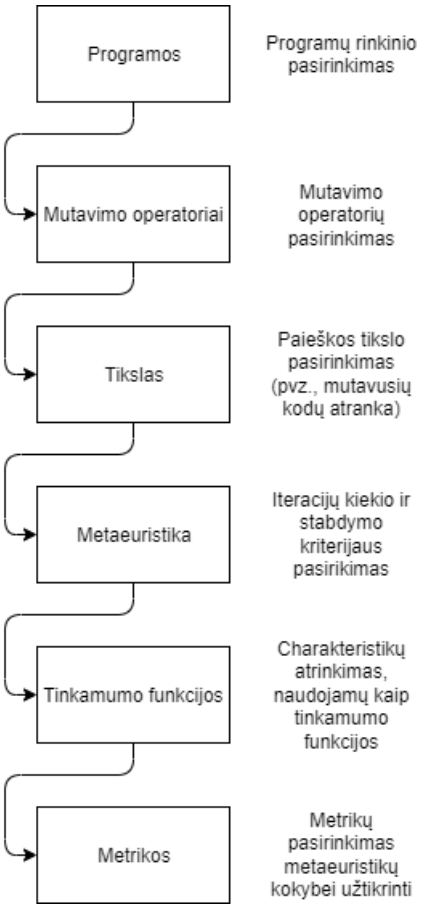
1 lentelė. Paieška paremtų programinio kodo testavimo tyrimų pasiskirstymas nurodytiems taikomiems metaeuristiniams metodams [SRS17]

Metaeuristikos	Testavimo atvejų generavimas	Mutavimo operatorių atrinkimas	Mutavusių kodų generavimas	Testavimo atvejų ir mutavusių kodų generavimas
AC	1	-	-	-
BA	1	-	-	-
BA, GA	2	-	-	-
EGA	2	-	-	-
GA	26	-	7	2
GA, HC, GrA, NSGA-II	-	-	1	-
GA, IA	2	-	-	-
GA, LS, RS	-	-	1	-
GP	2	-	-	-
GrA, GA, HC, RS	-	-	1	-
LS	2	-	-	-
HC	4	-	-	-
IA	1	-	-	2
NSGA-II	-	-	3	-
SA	3	-	-	-
TS, AC, NSGA-II	-	1	-	-
GA, SA, IA	1	-	-	-
GA, LS, RS, GLS, RRS, RES	-	-	1	-
GA, LS, RRS, GLS, ES	-	-	1	-
EA	1	-	-	-
GA, SA, HAIGA	1	-	-	-

2 lentelė. Paieška paremtų programinio kodo testavimo tyrimų pasiskirstymo lentelė nurodytiems taikomiems metaeuristiniams metodams, metaeuristinių metodų santrumpos [SRS17]

Santrumpos	Metodas	(angl.)
AC	Skruzdžių kolonijos metodas	*Ant Colony*
BA	Bakteriologinis algoritmas	*Bacteriological Algorithm*
EA	Evoliucionuojantis algoritmas	*Evoliutonary Algorithm*
EGA	Eltisto genetinis algoritmas	*Elltist Genetic Algorithm*
ES	Išvardijimo algoritmas	*Enumeration Search*
GA	Genetinis algoritmas	*Genetic Algorithm*
GLS	Orientuota lokali paieška	*Guided Local Search*
GP	Genetinis programavimas	*Genetic Programming*
GrA	Godusis algoritmas	*Greedy Algorithm*
HAIGA	Hibridinis dirbtinis imuninis genetinis algoritmas	*Hybrid Artificial Immune Genetic Algorithm*
HC	Lipimo į kalną metodas	*Hill Climbing*
IA	Imuninis algoritmas	*Immune Algorithm*
LS	Lokali paieška	*Local Search*
NSGA-II	Nedominuojantis rikiuojantis genetinis algoritmas II	*Non-dominated Sorting Genetic Algorithm-II*
RES	Apribota išvardijimo paieška	*Restricted Enumeration Search*
RRS	Apribota atsitiktinė paieška	*Restricted Random Search*
RS	Atsitiktinė paieška	*Random Search*
SA	Vėsinimo imitacijos metodas	*Simulated Annealing*
TS	Tabu paieška	*Tabu Search*

5 pav. yra pateiktos bendrosios charakteristikos, į kurias reikia atsižvelgti atliekant paieška paremto mutacinio testavimo eksperimentą.



5 pav. Paieška paremto mutacinio testavimo eksperimentų charakteristikų generalizavimas [SRS17]

Pagal 1 lentelę daugumoje darbų paieška paremto testavimo tema tirtos testavimo atvejų generavimo galimybės, nes ši mutacinio testavimo dalis išlieka aktuali. Daugumoje darbų koncentruotasi į genetinio algoritmo pritaikymą testavimo atvejams generuoti, tačiau šis algoritmas turi apribojimų. [SRS17] nagrinėtuose eksperimentuose rezultatui rasti genetinis algoritmas užtrukdavo ilgesnį laiką nei taikant kitus metodus. Taip pat jis pateikdavo gerus rezultatus, kai reikėdavo rasti vieną sprendinį, tačiau šis metodas nėra tinkamas, kai reikia rasti rezultatų aibę. Lyginant su genetinį algoritmą tirtais darbais, kiti metodai mokslinėje literatūroje nagrinėti gerokai rečiau. Todėl išlieka poreikis tirti kitus testavimo atvejų generavimo metodus.

5 pav. yra nurodytos pagrindinės charakteristikos, į kurias reikia atsižvelgti darant mutacinio testavimo tyrimų išvadas. Nepaisant atliktų eksperimentų negalima pateikti išvados apie geriausią metaeuristinį metodą mutacinio testavimo uždaviniui. Iš dalies taip yra dėl to, kad kai kuriuose eksperimentuose neatliktas bendras skirtingų metodų rezultatų palyginimas.

Pagrindiniai bendro optimizavimo metodų palyginimo iššūkiai atliekant darbą yra, kad eksperimentai atlikti skirtingomis programomis ir naudojant jų mutavusių kodų sąrašus, taip pat skirtingomis programavimo kalbomis, kurios turi skirtingus mutavimo operatorius. Atliekant eksperimentą yra svarbus mutavimo operatorių, mutavusių kodų generavimo būdo bei testavimo atvejų generavimo būdo pasirinkimas. Norint gauti lyginamuosius rezultatus, metodai turėtų būti testuojami su ta pačia programa ir jos mutavusių kodų aibe. Taip pat turi būti galimybė palyginti taikomą tinkamumo funkciją ir taikomas metrikas, tokias kaip mutavimo įvertis, kodo padengimo ir

pasiekiamumo kriterijus, būtinumo ir pakankamumo sąlygos. Kitos metrikos, tokios kaip vykdymo laikas bei iteracijų kiekis, gali būti naudojamos metaeuristinių metodų efektyvumui palyginti [SRS17].

2.4. Neišspręstos mutacinio testavimo problemos

Pagrindinės neišspręstos mutacinio testavimo problemos:

1. Viena pagrindinių neišspręstų problemų yra ekvivalenčių mutavusių kodų aptikimas. Yra atlikta nemažai darbų, kuriuose susitelkta ties ekvivalenčių mutavusių kodų aptikimu. Taip pat yra tyrinėjami metodai, kuriais būtų vengiama ekvivalenčių metodų generavimo arba mažinama tokių ekvivalenčių mutavusių kodų sugeneravimo tikimybė [JH11; SRS17].

2. Yra sukurta įrankių mutavimo operatoriams parinkti ir mutavusiems kodams generuoti, tačiau reikia naujų metodų ir tyrimų šioje srityje [SRS17].

3. Reikalingi naujų paieška paremtų testavimo metodų tyrimai specializuotose srityse, nes mutacinio testavimo srityje yra ištirta nedaug optimizavimo metodų. Taip pat mutacinis testavimas gali būti tiriamas lygiagrečių programų srityse [SRS17].

4. Paieška paremto testavimo strategijos identifikavimo sritis. Mutacinį testavimą būtų galima pritaikyti testavimo strategijoms vietoj testavimo atvejų paieškos. Tai padėtų pakelti abstrakcijos lygį ir sumažinti mutacinio testavimo kainą [SRS17].

5. Yra atlikta daug darbų mutacinio testavimo tema, tačiau jie atlikti naudojant skirtingas programas, programavimo kalbas, optimizavimo metodus ir nėra atlikto palyginimo tarp mutavimo operatorių parinkimo, ekvivalentumo problemos, testavimo atvejų generavimo darbų, todėl negalima išskirti tinkamiausių metodų šioms problemoms spręsti [SRS17]. Ateityje mutacinis testavimas gali būti vykdomas naudojant standartizuotą mutacinio testavimo įrankį [HO21], kurį naudojant galima palyginti ekvivalenčių mutavusių kodų aptikimo efektyvumą ir leis palyginti skirtingus taikomus metodus.

6. Yra atlikta daug mutacinio testavimo darbų, koncentruotų į mutavusių kodų generavimą, ir palyginti ne tiek daug darbų, koncentruotų į testavimo atvejų generavimą mutavusiems kodams sunaikinti. Yra egzistuojančių įrankių mutavusiems kodams generuoti, tačiau nėra panašaus pajėgumo testavimo atvejų generavimo įrankio, skirto mutavusiems kodams sunaikinti. Taip pat mutacinis testavimas suteikė galimybę įvertinti testavimo atvejų kokybę, tačiau atlikta nedaug darbų, skirtų patobulinti testavimo atvejų rinkinius atsižvelgiant į susijusią mutacinio testavimo analizę. Tikimasi, kad ateityje bus atlikta daugiau darbų, kuriuose bus siekiama naudoti aukštos kokybės mutavusius kodus kaip aukštos kokybės testavimo atvejų rinkinių generavimo pagrindą. Tačiau šiuo metu praktiškas testavimo atvejų generavimas siekiant pasiekti aukštą mutavimo įvertį lieka neišspręsta problema [KO91].

2.5. Ekvivalenčių mutavusių kodų problema

Ekvivalenčių mutavusių kodų atpažinimas yra kritinė užduotis optimizuojant mutacinio testavimo įgyvendinimo kainą. Kadangi mutavimo įvertis yra apibrėžiamas kaip sunaikintų mutantų ir neekvivalenčių mutavusių kodų santykis, siekiant užtikrinti proceso kokybę, svarbu sumažinti ekvivalenčių mutavusių kodų, naudojamų testavimo metu, skaičių.

Ekvivalenčių programų problema yra laikoma neišsprendžiama. Budd ir Angluin [BA82] įrodė, kad jeigu yra procedūra, nustatanti, ar dvi programos yra ekvivalenčios, taip pat yra ir procedūra adekvačiam testavimo rinkiniui generuoti, ir atvirkščiai. Čia testavimo rinkinys T programai P ir jos kaimynų aibei[2] ϕ yra laikomas adekvačiu, jei kiekvienai programai $Q \in \phi$ arba programa Q yra ekvivalenti programai P arba programa Q neįvykdo nors vieno testavimo atvejo iš rinkinio T. Budd ir Angluin parodė, kad nė viena iš šių procedūrų neegzistuoja. Todėl neegzistuoja algoritmas, nustatantis, ar dvi programos yra ekvivalenčios, ir todėl ekvivalenčių programų aptikimas laikomas neišsprendžiama problema.

Ekvivalenčių mutavusių kodų aptikimo problema yra ekvivalenčių programų aptikimo problemos poaibis. Šių uždavinių skirtumas yra tas, kad ekvivalenčių mutavusių kodų aptikimo uždavinyje vietoj dviejų atsitiktinių programų kaip įvestis yra pateikiami du kodai, kur vienas iš jų gaunamas atlikus kito kodo mutaciją. Nors iš ekvivalenčių programų problemos neišsprendžiamumo išplaukia, kad ekvivalenčių mutavusių kodų problema taip pat nėra išsprendžiama, ją galima spręsti specifiniais atvejais, pasinaudojant sintaksiniu programų panašumu.

Algoritmas 1 Programos, grąžinančios didesnį skaičių, pirmoji versija

```
1    int max(int a, int b) {
2        if(a > b) {
3            return a;
4        }
5        return b;
6    }
```

Algoritmas 2 Programos, grąžinančios didesnį skaičių, antroji versija

```
1    int max(int a, int b) {
2        if(a >= b) {
3            return a;
4        }
5        return b;
6    }
```

Kaip ekvivalenčių mutavusių kodų poros pavyzdys yra pateikti 1 ir 2 algoritmai, kurie gali būti mutavę pakeičiant vienas kito palyginimo operatorių. Jeigu $a \neq b$, tada šių programų tėkmė

[2] Čia programos P kaimynu vadinama programa, kuri priklauso nuo programos P

sutampa. Jeigu $a = b$, tada šių programų tėkmė skirtinga, nes 1 algoritmas kaip išvestį grąžina kintamąjį b, o 2 algoritmas grąžina kintamąjį a. Tačiau $a = b$, todėl grąžinamas rezultatas sutampa ir šiuo atveju. Todėl šie algoritmai yra ekvivalentūs. Šis pavyzdys rodo, kad ekvivalenčių mutavusių kodų problema nėra triviali.

Ekvivalenčių mutavusių kodų problemos variacijos yra ekvivalenčių mutavusių kodų aptikimo problema, ekvivalenčių mutavusių kodų siūlymo problema ir ekvivalenčių mutavusių kodų vengimo generuoti problema.

Ekvivalenčių mutavusių kodų aptikimo problema siekiama nustatyti, ar du kodai yra ekvivalentūs ir jos išvestis yra požymis *Taip*, jei ekvivalentūs, arba *Ne*, jei neekvivalentūs. Pagrindiniai ekvivalenčių mutavusių kodų aptikimo problemos metodai:

1. Kompiliatoriaus optimizavimo metodai. Šis metodas remiasi tuo, kad dauguma ekvivalenčių mutavusių kodų yra pradinio kodo optimizacijos arba deoptimizacijos. Todėl pritaikius kompiliatoriaus optimizatorius galima gauti dvi vienodas programas [OC94].

2. Matematiniai apribojimai. Šis metodas taikomas atliekant apribojimais paremtą testavimą ir programiniam kodui sukuriant apribojimų sistemą su pasiekiamumo (angl. *reachability*) ir būtinumo (angl. *necessity*) sąlygomis. Tada mutavusių kodų ekvivalentumas yra nustatomas jiems perskaičiuojant pradinio kodo apribojimus ir ieškant prieštaros, kur kodai skelbiami ekvivalenčiais tuo atveju, jeigu prieštaros nerandama [OP96].

3. Programų pjaustymas į dalis lengvesniam kodų aptikimui. Remiantis šiuo metodu, supaprastinamas kodo fragmentas, kurį testuotojas gali lengviau analizuoti ir nustatyti mutavusių kodų ekvivalentumą [HHG+00].

4. Semantinių skirtumų nagrinėjimas. Ellims et al. [EIP07] sprendė problemą kode pridedant papildomą požymį ir nagrinėjant jo pokyčius mutavusiame kode.

5. Margravo (angl. *Margrave*) pokyčio poveikio analizė. Margravo įrankis yra naudojamas automatiškai generuoti aukštos kokybės testavimo rinkinius su aukštu kodo padengimu. Sugeneruotas testavimo rinkinys gali būti naudojamas mutavusiems kodams naikinti, taip siekiant sumažinti ekvivalenčių mutavusių kodų kiekį [MX07].

6. Lesaro (angl. *Lesar*) modelio tikrinimas (angl. *model-checker*). Bousquet ir Delaunay [BD08] parodė, kad mutacinė analizė gali būti taikoma ir formaliai apibrėžtai deklaratyviai sinchroninei programavimo kalbai LUSTRE. Originaliai programai ir jos mutavusiems kodams gali būti pritaikomas Lesaro modelio tikrinimo įrankis, kuris kiekvienam mutavusiam kodui išveda išvestį, kuri vėliau palyginama su originalios programos išvestimi. Jeigu išvestys sutampa, programos pažymimos kaip ekvivalenčios.

Ekvivalenčių mutavusių kodų siūlymo problema siekiama nustatyti, ar du kodai yra ekvivalentūs ir jos išvestis yra tikimybė, ar mutavę kodai yra ekvivalentūs. Šia tikimybe programuotojas gali pasinaudoti nuspręsdamas, ar mutavęs kodas yra ekvivalentus, ar ne. Pagrindiniai ekvivalenčių mutavusių kodų siūlymo problemos metodai:

1. Bajeso mokymo (angl. *Bayesian-Learning*) gairių taikymas. Bajeso mokymas yra vienas iš mašininio mokymosi metodų, kuris išveda tikimybę remdamasis Bajeso išvada (angl. *Bayesian inference*). Vincenzi et al. [VNM+02] pritaikė šį metodą ekvivalenčių mutavusių kodų problemai.

2. Mutavusių kodų poveikio kodo padengimui analizė. Grün et al. [GSZ09] pasiūlė metodą, kur sekama mutavusių kodų programos veikimo tėkmė (angl. *control flow*) ir apskaičiuojamas skirtumas tarp originalaus kodo padengimo ir mutavusio kodo padengimo. Remiantis šiuo skirtumu išvedama tikimybė, ar kodai yra ekvivalentūs, ar ne.

3. Invariantų[3] tikrinimas. Šis metodas remiasi idėja, kad mutavęs kodas pažeidžia nustatytus invariantus, yra mažiau tikėtinas kandidatas būti ekvivalentus.

4. Kodo padengimo pokyčio tyrimas. Schuler ir Zeller [SZ13] parodė, kad taikant kodo padengimo įvertį, duomenų įtakos įvertį bei kombinuotą įvertį, galima prognozuoti mutavusių kodų ekvivalentumą. Jie nustatė, kad didesnį pokytį padarę mutavę kodai koreliuoja su neekvivalenčiais kodais ir naudojantis aprašytais įverčiais galima prognozuoti kodų ekvivalentumą.

Ekvivalenčių mutavusių kodų vengimo generuoti metodas yra trečias būdas spręsti ekvivalenčių mutavusių kodų problemą. Šis metodas remiasi tuo, kad pritaikius įvairius metodus yra sugeneruojami mutavę kodai, tarp kurių yra mažiau ekvivalenčių mutavusių kodų nei įprastai. Pagrindiniai ekvivalenčių mutavusių kodų vengimo generuoti metodai:

1. Pasirinktinas mutavimas (angl. *selective mutation*). Šis metodas remiasi tuo, kad yra pasirenkami mutavimo operatoriai, kurie generuoja mažiau ekvivalenčių kodų. [MB99].

2. Programų priklausomybių analizė. Taikant šį metodą, sugeneravus mutavusį kodą yra sudaromas sekamų kintamųjų sąrašas ir leidžiant mutavusį kodą yra sekama jų būsena. Jeigu šių kintamųjų būsenos nepadaro įtakos programos veikimui lyginant su pradiniu kodu, mutavęs kodas laikomas ekvivalenčiu, kitu atveju kodas pašalinamas [HHD01].

3. Koevoliucionuojančios paieškos analizė. Taikant šį metodą naudojant genetinį algoritmą ar kurį kitą evoliucionuojantį paieška paremtą algoritmą, yra parenkamas mutavimo operatorių ir testavimo atvejų rinkinys. Naudojant parinktus mutavimo operatorius, yra sugeneruojamas mutavusių kodų rinkinys, kuris yra ištestuojamas pritaikant parinktą testavimo atvejų rinkinį. Metodui apibrėžta tinkamumo funkcija (angl. *fitness function*) – tokia, kuri žemai įvertina mutavusius kodus, kurie išgyveno visus testavimo atvejus. Tokiu būdu įmanoma pašalinti visus ekvivalenčius mutavusius kodus iš mutavusių kodų rinkinio ir palikti tik tokius kodus, kuriems pavyko surasti testavimo atvejį, su kuriuo jis pateikia nekorektišką rezultatą [AHH04].

4. Semantinis trikių (angl. *exception*) hierarchijos nagrinėjimas. Mutavusiems kodams aptikti ir pašalinti gali būti taikomi ir metodai, dirbantys su trikiais. Taikant mutavimo operatorius,

[3]Invariantas – savybė, kuri nesikeičia įvykstant pokyčiui

kurie keičia su trikiais susijusį kodą, sugeneruojamas mutavusių kodų rinkinys ir paruoštiems testavimo atvejams tikrinami programinio kodo pagaunami trikiai ir tikrinama jų hierarchija, siekiant nustatyti jų ekvivalentumą [JCX+09].

5. Aukštesnio lygio mutavimas. Siekiant išvengti ekvivalenčių mutavusių kodų, kodas gali būti mutuojamas daugiau nei vieną kartą. Tokiu būdu dėl didesnio skirtumo tarp pradinio kodo ir mutavusio kodo mažėja ekvivalentaus mutavusio kodo sugeneravimo tikimybė [JH09].

Ekvivalenčių mutavusių kodų problemai spręsti gali būti pritaikomas ir mašininis mokymasis. Taikant šį metodą, modelio architektūra būtų neuroninis tinklas, kuriam kaip įvestis būtų pateikiamas originalus kodas ir mutavęs kodas, o neuroninis tinklas pateiktų išvestį, ar tai ekvivalentūs kodai, ar ne. Išvestis gali būti pateikiama tiek tikimybės forma, tiek požymio forma, jeigu reikia skubiai, be žmogaus įsikišimo, įvertinti, ar kodų pora ekvivalenti, ar ne.

Numatoma, kad ateityje ekvivalenčių mutavusių kodų aptikimo srityje nauji sukurti įrankiai turėtų būti ištestuoti naudojant standartizuotą mutavusių kodų rinkinį, pasiūlytą [HO21] straipsnio autorių. Taip pat turėtų būti papildytas standartinis mutavimo operatorių sąrašas siekiant plėsti mutacinio testavimo galimybes.

Siekiant sukurti kokybiškus mašininio mokymosi įrankius ekvivalenčių mutavusių kodų problemai reikalingas gausus duomenų rinkinys. Didžiausias šiuo metu žinomas mutavusių kodų rinkinys yra *MutantBench* [HO21], kuris susideda iš 4400 mutavusių kodų, iš kurių 32 % yra ekvivalentūs. Taip pat Chung ir Yoo [CY22] pasiūlė įdėją, kad taikant simbolinį apdorojimą (angl. *symbolic execution*) galima praplėsti mutavusių kodų rinkinį. Jų pateikta idėja remiasi tuo, kad kode galima rasti kintamuosius, kurie nuo kažkurios eilutės nebekeičia programos tėkmės, ir vėlesnėje eilutėje galima įterpti vienetinius (angl. *short-cut*) operatorius, kurie keičia kintamojo reikšmę. Tačiau literatūroje trūksta duomenų rinkinio, kuris būtų pakankamai didelis ir padengtas įvairių mutavimo operatorių siekiant jį pritaikyti mašininio mokymosi metodams.

3. Mašininio mokymosi analizė

Šiame skyriuje aprašyta mašininio mokymosi būdai ir mašininio mokymosi pritaikymas kodų klonų uždaviniui ir ekvivalenčių mutavusių kodų uždaviniui.

3.1. Mašininio mokymosi metodai

Mašininis mokymasis (angl. *machine learning*) yra apibrėžiamas kaip sritis, kuri suteikia kompiuteriams galimybę mokytis neapibrėžiant konkrečių algoritmų [Mah20]. Mašininis mokymasis yra taikomas išmokyti mašinas efektyviai apdoroti duomenis. Kartais iš duomenų negalima tiksliai interpretuoti duomenų, todėl yra taikomas mašininis mokymasis. Mašininis mokymasis yra skaidomas į kelias sritis:

1. Prižiūrimasis mokymasis (angl. *supervised learning*). Tai mašininio mokymosi metodas, kurio metu yra išmokstama funkcija pateiktai įvesčiai grąžinti rezultatą pagal pavyzdinius įvesties ir išvesties duomenis. Šiam mokymo būdui taikomi metodai yra pasirinkimų medis (angl. *decision tree*), naivusis bajesas (angl. *naive bayes*), atraminių vektorių klasifikatorius (angl. *support vector machine*) ir kt.

2. Neprižiūrimasis mokymasis (angl. *unsupervised learning*). Tai mašininio mokymosi metodas, kurio metu, priešingai nei prižiūrimojo mokymosi metu, nėra mokytojo ir teisingų išvesčių. Mokymo metu algoritmas atrenka parametrus duomenims klasifikuoti. Išmokytam modeliui pateiktiems naujiems duomenims jis pagal šiuos parametrus priskiria mokymo metu išmoktas klases. Šiam mokymo būdui taikomi metodai yra principinių komponentų analizė (angl. *principal component analysis*), k-vidurkių metodas (angl. *k-means*) ir kt.

3. Pusiau prižiūrimasis mokymasis (angl. *semi-supervised learning*). Tai mašininio mokymosi metodas, kai dalis duomenų yra sužymimi laukiama išvestimi. Šiam mokymo būdui taikomi metodai yra generatyvūs modeliai (angl. *generative models*), savasis mokymasis (angl. *self training*), transduktyvus atraminių vektorių klasifikatorius (angl. *transductive support vector machine*) ir kt.

4. Mokymasis su pastiprinimu (angl. *reinforcement learning*). Tai mašininio mokymosi metodas, kuris apibrėžia, kaip virtualūs agentai priima sprendimus specialiai sukurtoje aplinkoje siekiant maksimizuoti visuotinę naudą.

5. Daugiafunkcis mokymasis (angl. *multi-task learning*). Tai mašininio mokymosi metodas, kuris skirtas spręsti kelias problemas vienu metu remiantis sprendžiamų problemų panašumais.

6. Kombinuotasis mokymasis (angl. *ensemble learning*). Tai mašininio mokymosi metodas, kurio metu yra kombinuojami skirtingi modeliai, siekiant pagerinti modelių tikslumą arba sumažinti tikimybę pasirinkti netinkamą atsakymą. Šiam mokymo būdui taikomi metodai yra skatinimas (angl. *boosting*), pakavimas (angl. *bagging*) ir kt.

7. Neuroniniais tinklais paremtas mokymasis (angl. *neural network*). Neuroninis tinklas yra algoritmų seka, kuri bando atpažinti priklausomybes tarp duomenų imituojant žmogaus smegenų darbo procesą. Todėl neuroninį tinklą galima apibrėžti kaip neuronų sistemą, tiek natūralią, tiek dirbtinę. Tipinė neuroninio tinklo struktūra susideda iš įvesties sluoksnio, kuriam yra pateikiami apdoroti įvesties duomenys, vieno ar daugiau vidinių sluoksnių ir išvesties sluoksnio, kuriame yra išvedamas uždavinio sprendimas. Dėl savo universalumo neuroniniai tinklai sparčiai populiarėja dirbtinio intelekto srityje. Tipiniai taikomi metodai šiam mokymuisi yra prižiūrimasis neuroninis tinklas (angl. *supervised neural network*), neprižiūrimasis neuroninis tinklas (angl. *unsupervised neural network*), pastiprintas neuroninis tinklas (angl. *reinforced neural network*).

8. Vienetu paremtas mokymasis (angl. *instance based learning*). Priešingai nei anksčiau aprašyti metodai, šis metodas nesistengia išvesti abstrakčių taisyklių vienetams identifikuoti. Šiam metodui taikomas artimiausio kaimyno metodas, kur pateikus testinį atvejį surandamas jo artimiausias kaimynas duomenų rinkinyje, pagal kurį nustatoma, kaip identifikuoti testinį atvejį.

3.2. Mašininio mokymosi pritaikymas kodų klonų uždaviniui

Kodų klonų aptikimo uždavinys yra klasifikuojamas į kelias pagrindines metodų grupes [ABA+19]:

1. Tekstiniai metodai (angl. *textual approaches*). Tai yra metodai, nagrinėjantys pradinį programos kodą.

2. Leksiniai metodai (angl. *lexical approaches*). Tai yra metodai, nagrinėjantys programinį kodą, suskaidytą į leksemas.

3. Paremti medžiu metodai (angl. *tree-based approaches*). Tai yra metodai, nagrinėjantys programinį kodą, pateiktą medžiu.

4. Paremti metrikomis metodai (angl. *metric based approaches*). Tai yra metodai, nagrinėjantys programinio kodo požymius, kurie yra vadinami metrikomis.

5. Semantiniai metodai (angl. *semantic approaches*). Tai yra metodai, kuriuose kodas nagrinėjamas kaip kodo priklausomybių grafas (angl. *program dependence graph*). Čia programų priklausomybių grafas yra duomenų ir programos veikimo sąryšių reprezentacija, pateikta grafu [FOW87].

6. Hibridiniai metodai (angl. *hybrid approaches*). Tai yra metodai, kuriuose yra naudojamos ir kombinuojamos įvairios idėjos iš prieš tai aprašytų metodų.

Toliau darbe pasirinkta nagrinėti paremtą medžiu metodą, taikomą kartu su metrikų metodais, kadangi literatūroje nėra bandyta taikyti šių idėjų kartu [SRK+21]. Metrikų sąrašas gaunamas

perskaičius iš programinio kodo suformuotą medį ir suskaičiuojant, kiek atskirų metrikose aprašytų požymių elementų turi ASM. Medžiu paremtiems ir metrikomis paremtiems neuroniniams tinklams įgyvendinti neuroniniai tinklai, sprendžiantys šį uždavinį. Šie modeliai vėliau pritaikyti ir ekvivalenčių mutavusių kodų problemai spręsti.

3.3. Mašininio mokymosi pritaikymas ekvivalenčių mutavusių kodų uždaviniui

Ekvivalenčių mutavusių kodų uždavinys yra panašus į kodų klonų aptikimo problemą. Kaip įvestis abiejų uždavinių modeliams yra kokiu nors metodu apdorota kodų pora, kuriai modelis pateikia atsakymą tikimybės pavidalu arba teigiama, arba neigiama reikšme. Todėl visus aprašytus kodų klonų aptikimo metodus galima pritaikyti ekvivalenčių mutavusių kodų aptikimo uždaviniui.

[NLN+19] straipsnyje pateiktas apribojimais paremto testavimo sprendimas ekvivalenčių mutavusių kodų aptikimo problemai, kuriam buvo taikytas mašininis mokymasis. Problemai spręsti buvo taikyti abstraktaus sintaksės medžio, semantinis, kuris remiasi programų priklausomybių grafu, ir metrikų metodai. Iš programų priklausomybių grafo būdavo surenkamos pasiekiamumo (angl. *reachability*), būtinumo (angl. *necessity*) ir pakankamumo (angl. *sufficiency*) metrikos. Remiantis šiomis metrikomis mutavę kodai suskaidyti į tris tipus: pirmojo tipo mutavusius kodus, kurie neatitinka pasiekiamumo ir būtinumo sąlygų, antrojo tipo, kurie tenkina pasiekiamumo ir būtinumo sąlygas, bet trūksta priklausomybių tarp kintamųjų, kurių reikšmės mutavus kodą pasikeitė, ir jų išvesčių, ir trečiojo tipo, kuris atitinka pirmų dviejų tipų mutavusių kodų sąlygas, tačiau mutavusių kodų būsenos nebūtinai sutampa programos veikimo metu. Kadangi trečiojo tipo ekvivalentūs mutavę kodai negali būti nustatomi taikant tradicinius mutavusių kodų aptikimo metodus (pvz., programų priklausomybių grafų lyginimas), todėl autoriai trečiojo tipo mutavusiems kodams aptikti taikė mašininio mokymosi metodus. Jiems kaip įvestis buvo apskaičiuojamas metrikų sąrašas. Autorių taikyti modeliai buvo atsitiktinio miško (angl. *random forest*), gradientu pagreitintais medžiais (angl. *gradient boosted trees*), atraminių vektorių mašinomis (angl. *support vector trees*) ir giliojo mokymosi modelis, paremtas H2O atgalinio grįžtamojo rezultato neuroniniu tinklu (BPNN (angl. *Backpropagation Neural Network*)). Autoriai pagamino savo duomenų rinkinį aprašę keletą pavyzdinių programų ir taikydami *Mujava* įrankį pasirinkę taikomų mutavimo operatorių poaibį pagamino mutavusių kodų rinkinį, kurį sužymėjo rankiniu būdu, iš dalies pasitelkiant testavimo atvejų vykdymą. Jų gautą duomenų rinkinį sudarė 1393 ekvivalentūs mutavę kodai, kurie sudarė 46 % viso duomenų rinkinio. Gauti rezultatai parodė, kad giliojo neuroninio tinklo modelis parodė 82 % tikslumo ir išsamumo įvertį, tačiau šiek tiek nusileido atsitiktinio miško ir gradientu paremtų medžių modeliams, kurių tikslumo ir išsamumo įverčiai atitinkamai buvo lygūs 85 % ir 86 %.

[PDD+21] straipsnyje pateiktas bandymas spręsti ekvivalenčių mutavusių kodų aptikimo uždavinį taikant giliojo mokymo metodą – medžio struktūra paremtą neuroninį tinklą, kuris originaliai buvo realizuotas spręsti kodų klonų aptikimo uždavinį [ZWZ+19]. Autoriai sudarė savo mutavusių kodų rinkinį, sudarytą iš mutavusių kodų, sugeneruotų taikant ABS (absoliučios reikšmės įterpimo (angl. *Absolute Value Insertion*)) ir OUI (vienetinio operatoriaus įterpimo (angl. *Unary Operation Insertion*)) mutavimo operatorių. Modeliai buvo mokyti atskirai šiems dviem mutavimo operato-

riams. Gauti rezultatai parodė neblogą modelių tikslumą (90 % operatoriui ABS ir 94 % operatoriui OUI).

[KMS+22] straipsnyje pateiktas panašus bandymas automatizuoti ekvivalenčių mutavusių kodų aptikimo problemą *Android* programoms. Autoriai sudarė savo mutavusių kodų rinkinį, sudarytą iš mutavusių kodų, sugeneruotų taikant ABS mutavimo operatorių, o ekvivalentiems mutavusiems kodams aptikti autoriai taip pat taikė medžio struktūra paremtą neuroninį tinklą pagal [ZWZ+19]. Mutavusiems kodams generuoti autoriai naudojo FAIR principą, kuriuo pažymima, kad duomenys turi būti randami (angl. *findable*), pasiekiami (angl. *accessible*), sąveikūs (angl. *interoperable*) ir pakartotinai panaudojami (angl. *reusable*), bei šį principą įgyvendinantį standartizuotą mutacinio testavimo įrankį, pasiūlytą [HO21] straipsnio autorių. Sukurtas modelis ekvivalentiems mutavusiems kodams aptikti pasiekė 94 procentų tikslumą ir 92 procentų validavimo tikslumą. Literatūros apžvalgos rezultatai rodo, kad taikant mašininį mokymąsi galima identifikuoti ekvivalenčius mutavusius kodus tam tikru tikslumu.

4. Projekto realizacija

Šiame skyrelyje aprašyta įgyvendinto projekto realizacija[4]. Kaip bazinis modelis iš pradžių buvo pasirinktas ir ASM strūkturos modeliui įgyvendinti pritaikytas Zhang et al. aprašytas modelis [ZWZ+19], kuris buvo praplėstas kitomis darbe nagrinėjamomis dalimis.

4.1. Projekto struktūra

Projektas susideda iš kelių pagrindinių dalių:

1. Aplanke *clone* paruošta eksperimentinė aplinka ir realizuotas kodų klonų uždavinio sprendimas, jam taikyti abstrakčios sintaksės medžio ir metrikų modeliai, jų kombinacija ir patys modeliai, išmokyti taikant mutavusių kodų rinkinį. Taip pat šiame aplanke aprašytas ir ekvivalenčių mutavusių kodų aptikimo sprendimas taikant abstrakčios sintaksės medžio ir metrikų modeliai bei jų kombinaciją.

2. Aplanke *mujava* realizuotas kodas, skirtas darbui su *Mujava* įrankiu. Jis buvo naudojamas tiek mutavusių kodų duomenų rinkiniui generuoti, tiek naujų testuojamų kodų mutavusiems kodams generuoti.

3. Pagrindiniame aplanke realizuota mutacinio testavimo aplinka. Šioje aplinkoje taip pat paruoštas mutacinio testavimo prototipas.

Detalus šių trijų dalių aprašymas yra pateiktas *readme* failuose atitinkamuose aplankuose.

4.2. Naudotų bibliotekų sąrašas

Modeliams realizuoti buvo pasirinkta *PyTorch* mašininio mokymosi biblioteka. Projektas buvo rašomas naudojantis 3.9 *Python* versija. Kadangi projektui pritaikytas *Zhang* ASM modelis [ZWZ+19] eksperimente buvo realizuotas naudojant natūralios kalbos apdorojimo uždaviniams spręsti skirtą *Gensim* biblioteką su 3.5.0 versija, reikėjo bibliotekos panaudojimus migruoti į 4.1.2 versiją.

Visos panaudotos *Python* bibliotekos išvardytos 3 lentelėje.

[4]Nuoroda į projektą: https://github.com/Aurimasjar/Mutation-Testing

3 lentelė. *Python* bibliotekos

Biblioteka	Versija
Click	8.0.4
Gensim	4.1.2
Javalang	0.13.0
JPype1	1.4.1
Matplotlib	3.5.2
Numpy	1.21.5
Pandas	1.4.4
Pycparser	2.21
Scikit learn	1.2.2
Torch	1.13.1
Tqdm	4.64.1

5. Kodų klonų aptikimo uždavinio sprendimas

Kodo klonas apibrėžiamas kaip kodo fragmentas, kuris yra toks pat arba panašus į kitą kodo fragmentą toje pačioje arba kitoje programinio kodo sistemoje [Ino21]. Šiame skyriuje aprašyti ir ištestuoti architektūriniai sprendimai kodų klonų uždaviniui. Vėliau šios struktūros pritaikytos ir ištestuotos ekvivalenčių kodų aptikimo uždaviniui.

5.1. Modelio architektūra

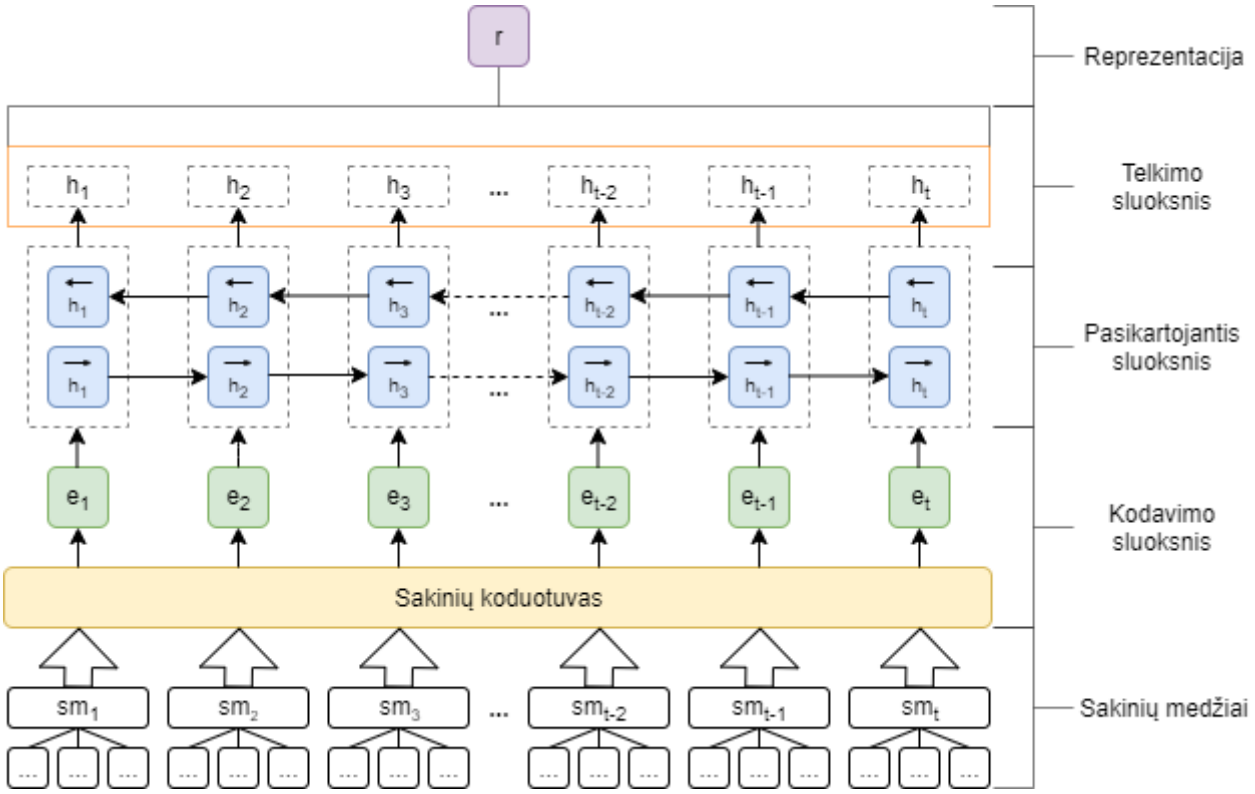
Šiame skyriuje aprašyta darbe įgyvendinta ASM ir metrikų modelių architektūra bei realizuota jų kombinacija.

5.1.1. ASM struktūros modelis

Abstraktus sintaksės medis (angl. *Abstract Syntax Tree*) (toliau – ASM) yra struktūra, reprezentuojanti programinio kodo abstrakčią sintaksę, nepriklausomą nuo programavimo kalbos. ASM yra taikomi programų klasifikavimo, kompiliavimo ir optimizavimo [HHB14], kodų klonų aptikimo [ZWZ+19], saugumo spragų aptikimo [YLR12] ir kitiems uždaviniams spręsti. ASM įprastai iš programinio kodo gaunamas panaudojus lekserį (angl. *lexer*), kuris simbolių rinkinį paverčia į leksemų (angl. *token*) rinkinį, ir analizatorių (angl. *parser*), kuris leksemų rinkinį paverčia abstrakčios sintaksės medžiu [OFS+19].

Darbe pritaikytas ASM struktūros neuroninio tinklo modelis kodų klonų aptikimo uždaviniui [ZWZ+19]. Modeliui panaudotas medžio struktūros konvoliucinis neuroninis tinklas (angl. *tree-based neural network*), kuris kaip įvestį priima programinių kodų abstrakčios sintaksės medžius. Gavęs įvestį, medžio struktūros neuroninis tinklas apskaičiuoja vektorinę išraišką, skaitydamas medį nuo lapų iki viršūnės, kurią naudoja tolesniems skaičiavimams. Straipsnyje [ZWZ+19] sprendžiama gradiento išnykimo (angl. *gradient vanishing*) problema, kai gradientas tampa nykstamai mažas modelio mokymo metu ir kuri ypač išryškėja mokymui naudojant didelius įvesties medžius. Todėl straipsnyje aprašyta proceso modifikacija, kuri iš didesnių programinių kodų pagamintus medžius suskaido į mažesnius.

Abstraktaus sintaksės medžio neuronų tinklo architektūra yra pateikta 6 pav. Čia rodyklės žymi duomenų judėjimą. Skaičius t žymi į tinklą pateikto paketo (angl. *batch*) medžių rinkinį, kurie buvo suskaidyti į pomedžius, siekiant išvengti gradiento išnykimo problemos. Vektoriai e_i žymi vektorių, reprezentuojantį vieną medžio elementą, o vektoriai h_i žymi šį vektorių dvikrypčiame pasikartojančių vartų mazge, kur jis būna perskaičiuojamas ir pateikiamas į telkimo sluoksnį.



6 pav. Abstraktaus sintaksės medžio neuroninio tinklo architektūra [ZWZ+19]

Pagal 6 pav. pradžioje kiekvienas ASM yra perrenkamas perėjiko (angl. *traverser*) ir konstruktoriaus. Perėjikas aplanko kiekvieną medžio viršūnę ėjimo į gylį (angl. *depth-first*) strategija pirmine (angl. *preorder*) medžio perėjimo tvarka. O konstruktorius kiekvienai viršūnei, kuri yra laikoma kaip sakinys, rekursyviai kuria sakinių medį ir prideda prie sakinių medžių sekos. Tokiu būdu gaunama sakinių medžių (angl. *ST-tree*) sekos, kaip abstraktaus sintaksės medžio neuroninio tinklo (angl. *abstract syntax tree neural network*) (toliau ASMNN), įvestis.

Kitas žingsnis yra gautą įvestį toliau apdoroti neuroninio tinklo mokymo etapui naudojant natūralios kalbos apdorojimo metodus. Šiam žingsniui yra naudojamas *word2vec* įrankis, kuris priima ištreniruotus simbolių įterpimus (angl. *embedding*) su sakinių koduotuvu ir grąžina vektorius, gautus taikant neprižiūrimąjį (angl. *unsupervised*) mokymą.

Toliau yra naudojamas dvikryptis pasikartojančių vartų mazgas (angl. *Bidirectional Gated Recurrent Unit*), skirtas apdoroti sakinių natūralumui. Į šį mazgą yra pateikiami užkoduoti vektoriai. Toliau vektoriai yra perduodami nuo pirmojo iki paskutinio, kur pirminiai vartai (angl. *reset gate*) kontroliuoja, kokia dalis ankstesnės būsenos turi įtakos naujai būsenai. Paskui vektoriai yra paskaičiuojami atgal nuo paskutinio iki pirmojo, kur atnaujinantys vartai (angl. *update gate*) sukombinuoja buvusio vektoriaus ir naujo vektoriaus informaciją.

Toliau naudojant telkimą atskiri vektoriai (angl. *pooling*) yra sukabinami į vientisą vektorių r, atrenkant svarbiausią informaciją, kuri reprezentuoja pradinį programinį kodą. Čia svarbiausia informacija laikoma didžiausia vektoriaus h_i reikšmė. Vektorius r yra laikomas programinio kodo vektorine išraiška.

Toliau turima dviejų programinių kodų $r1$ ir $r2$, pateiktų vektorine išraiška, pora, o atstumas tarp jų r yra apskaičiuojamas pagal formulę $r = |r1 - r2|$. Tada išvestis $y' = sigmoid(x')$ $[0, 1]$

gali būti laikoma jų panašumo indeksu, čia $x' = W_0 r + b_0$, $W_0 r$ yra svorių matrica, o b_0 yra bazė. Praradimų funkcija yra apibrėžta kaip binarinė kryžminė entropija (angl. *cross-entropy*), kuri toliau darbe žymima kaip *BCELoss*:

$$J(\Theta, y', y) = \sum(-(y \cdot log(y') + (1-y) \cdot log(1-y')))$$

Taip pat optimizavimui buvo naudojama *AdaMax* funkcija. Mokymosi greičiu (angl. *learning rate*) parinkta reikšmė, lygi 0,001. Modelis buvo mokomas taikant prižiūrimąjį mokymąsi (angl. *supervised learning*).

Galiausiai išmokyti modeliai yra išsaugomi. Norint pateikti naujas kodų poras ir gauti jų rezultatą, jie suformuojami į sakinių medžių seką ir teikiami modeliui grąžinti tikimybę. Tikimybė apibrėžiama kaip:

$$A = \begin{cases} Tiesa, & p > \delta \\ Netiesa, & p \leq \delta \end{cases}$$

Čia δ laikomas slenksčiu, o A yra laikomas modelio spėjimu. Jeigu $A = Tiesa$, tai programinių kodų pora laikoma ekvivalenčia, priešingu atveju programinių kodų pora laikoma neekvivalenčia.

5.1.2. Metrikų modelis

Kitas pasirinktas nagrinėti metodas buvo kodų klonų aptikimas, paremtas metrikomis. Programinio kodo metrika – tai parametras, gautas apskaičiuojant kokį nors pradinio kodo požymį. Metrikos yra naudojamos kodo kokybei pagerinti, taip pat klaidų aptikimo, testavimo, kodo pertvarkymo (angl. *refactoring*) uždaviniams [NPM+17].

Šiame darbe pasirinkta metrikas gauti nagrinėjant jau iš pradinio kodo gautą abstraktų sintaksės medį, grindžiant tuo, kad medis turi jau struktūrizuotą kontekstinę informaciją ir suformuos informatyvesnes metrikas. Pačių taikomų metrikų kiekis taip pat svarbus. Pasirinkus naudoti per mažai metrikų, neuroniniam tinklui būtų sudėtinga atpažinti požymius ir pasiekti gerų rezultatų. Tačiau modeliui patiekus per daug metrikų ir testuojant ant nedidelio duomenų rinkinio įvyksta modelio persimokymas (angl. *overfitting*), dėl to modelis per daug tiksliai klasifikuotų duomenis, ant kurių yra apmokomas, bet nebūtų toks korektiškas testuojant naujais duomenimis dėl pasitaikančių kraštinių atvejų. Iš viso aprašyta p leksemų parametrų, ieškantys tarp leksemų konkrečių žodžių, ir q medžio parametrai, pateikiantys informaciją apie patį medį. Visos metrikos yra skaitinio diskretaus tipo.

Apskaičiavus metrikas gaunamas duomenų vektorių rinkinys su metrikų skaitinėmis reikšmėmis. Tačiau neuroniniam tinklui pateikiamus duomenis reikia normalizuoti. Tam buvo galimi du būdai: normalizuoti visą duomenų rinkinį padalijant visus vektorius iš ilgiausio vektoriaus ilgio:

$$v = \frac{v}{max(||v||)}$$

Tačiau tokiu atveju modelis būtų nesubalansuotas, nes kai kurių metrikų reikšmės yra gerokai

didesnės už daugumos kitų ir modelis joms galimai skirtų nevienodą svorį. Todėl buvo pasirinktas duomenų standartizavimo būdas. Kiekvienai metrikai apskaičiuotas vidurkis ir standartinis nuokrypis (angl. *standard deviation*) ir vektoriai transformuoti iš jų atimant metrikų vidurkių vektorių ir padalijant iš metrikų standartinio nuokrypio vektoriaus:

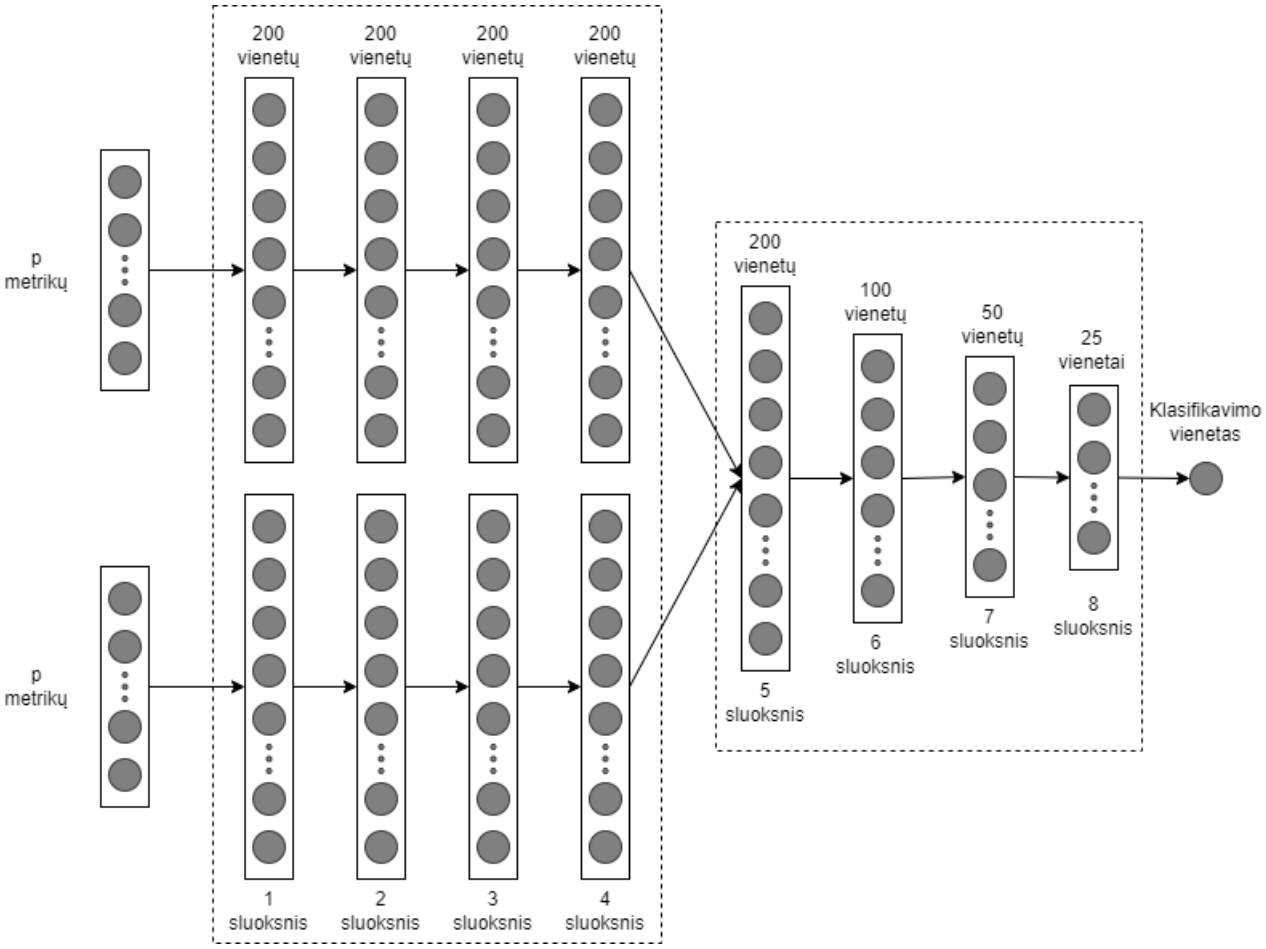
$$v = \frac{v - avg(v)}{std(v)}$$

Tokiu būdu gauti standartizuoti duomenys, kuriuos galima pateikti į neuroninį tinklą.

Metrikomis paremto neuroninio tinklo tikslas yra identifikuoti, ar pateikta programinių kodų pora yra ekvivalenti, ar ne. Todėl į neuroninį tinklą yra pateikiami ne atskiri programinius kodus reprezentuojantys metrikų vektorių vienetai, o metrikų vektorių pora. Tokiam duomenų pateikimui pagal [SFL+18] autorių atliktą kodų klonų aptikimo tyrimą pasirinktas siamietiškos architektūros modelis. Siamietiškas neuroninis tinklas (angl. *Siamese Neural Network*) yra dirbtinio neuroninio tinklo versija, kuri naudoja tuos pačius svorius dirbdama su dviem skirtingais įvesties vektoriais, kad būtų išvedami du palyginami vektoriai [PS22]. Tokio tipo tinklas, be kodų klonų aptikimo uždavinio, taip pat taikomas teksto panašumo palyginimo [NVR16] ir kitiems natūralios kalbos apdorojimo uždaviniams, kuriuose reikalingas dviejų įvesčių palyginimas.

7 pav. yra pateiktas realizuotas siamietiškas neuroninio tinklo modelis. Kaip įvestis yra pateikiama standartizuotų vektorių $r1$ ir $r2$ pora, atitinkanti programinius kodus. Jie nepriklausomai praeina pro 4 vidinius sluoksnius. Tada yra apskaičiuojamas skirtumas tarp jų $r = |r1 - r2|$. Gautas vektorius pateikiamas į tolimesnius sluoksnius su 200, 100, 50 ir 25 vienetais (neuronais). Paskutiniame sluoksnyje grąžinama viena konkreti reikšmė, kuriai pritaikoma sigmoidinė aktyvavimo funkcija. Galiausiai neuroninis tinklas šią reikšmę grąžina kaip klasifikavimo vienetą, kuris pažymi spėjimą, ar kodai yra ekvivalentūs, ar ne.

Šiam modeliui taip pat buvo naudojama praradimų funkcija, apibrėžta kaip binarinė kryžminė entropija ir optimizavimui skirta *AdaMax* funkcija. Mokymosi greičiu (angl. *learning rate*) parinkta reikšmė, lygi 0,001. Modelis buvo mokomas taikant prižiūrimąjį mokymąsi (angl. *supervised learning*).



7 pav. Siamietiškos architektūros modelis, paremtas [SFL+18]

5.1.3. ASM ir metrikų modelių kombinacija

Šiame darbe pritaikyta ASM ir metrikų modelių kombinacija [SRK+21], kuri nebuvo aptikta literatūroje. Pasirinkta ASM struktūros ir metrikų modelius mokyti nepriklausomai. Modelių testavimas vykdomas ant to paties testavimo atvejų rinkinio abiem modeliams, siekiant gauti lyginamuosius rezultatus. Modeliams kombinuoti galimi keli skirtingi metodai:

1. Modelio skatinimas (angl. *boosting*). Taikant šį metodą, vieno modelio išvestis galėtų būti naudojama kaip kito modelio įvestis taip siekiant pagerinti galutinę išvestį.

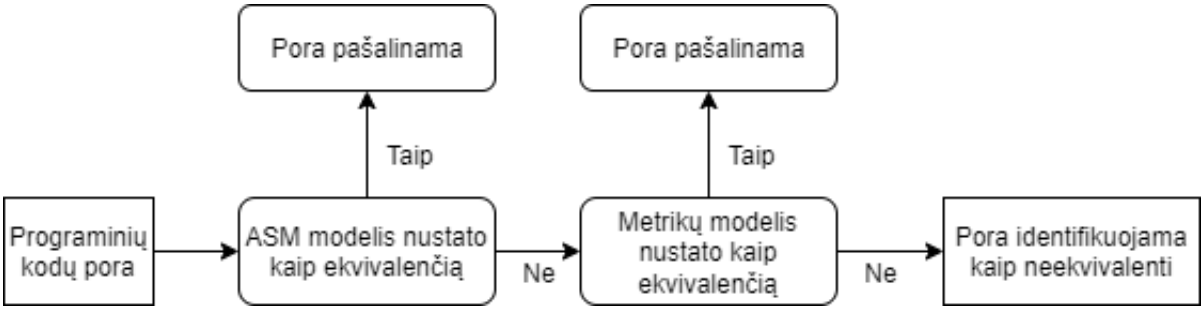
2. Modelio sudėjimas (angl. *stacking*), kuris aprašomas meta algoritmu (angl. *meta algorithm*), kuris priima skirtingų modelių pateiktas išvestis ir išmoksta, kaip pasiekti geriausią tikslumą kombinuojant skirtingų modelių pateiktas tikimybes.

3. Modelių rezultatų vidurkių vedimas (angl. *averaging*). Taikant šį metodą, iš skirtingų modelių pateiktų rezultatų yra vedami vidurkiai ir pateikiami susumuoti galutiniai įverčiai.

Modelių kombinavimo tikslas yra padidinti išsamumo įvertį, kad kodų klonai būtų kuo tiksliau aptinkami, kadangi kodų klonų aptikimo uždavinys yra sprendžiamas siekiant išspręsti mutaciniame testavime atsirandančią ekvivalenčių mutavusių kodų problemą, kur atsiradę ekvivalentūs

mutavę kodai neleidžia aklai pasikliauti mutacinio testavimo rezultatais. Todėl darbui pritaikytas kombinuotas modelis, kuriame skiriamas dėmesys padidinti galutinio modelio išsamumo įvertį sumažinant po testavimo išgyvenusių ekvivalenčių mutavusių kodų kiekį.

Modelių testavimo procesas pateiktas 8 pav. Šis testavimo procesas apibrėžia realizuotą kombinuotą modelį.



8 pav. Kodų klonų testavimo procesas

Čia kiekviena programinių kodų pora iš testavimo rinkinio pateikiama į abu modelius ir identifikuojama kaip neekvivalenti, jeigu nė vienas modelis jų tokia nenustato.

5.2. Modelio realizacija

Šiame poskyryje aprašyti kodų klonų aptikimo uždavinio sprendimo mokymų duomenys ir rezultatai. Tyrimai buvo atliekami su abstrakčios sintaksės medžio ir metrikų modeliais taikant C ir *Java* kalbos duomenų rinkinius. Taip pat palyginti kombinuoti modelių rezultatai.

5.2.1. Naudojamas duomenų rinkinys

C/C++ programavimo kalbai kodų klonų aptikimo uždaviniui naudotas *OJClone* duomenų rinkinys [MLJ+14]. Duomenų rinkinys gautas problemoms gavus studentų pateiktus sprendimus programinio kodo formatu, išsaugotus atskiruose failuose. Autoriai [MLJ+14] gautus sprendimus suporavo tarpusavyje ir sužymėjo kodus, ar jie yra klonai, ar ne.

Duomenų rinkinys suskaidytas į mokymo, validavimo ir testavimo duomenų rinkinius santykiu 60 %, 20 %, 20 %, o jo dydis yra 50 000 programinių kodų porų. Tačiau pats duomenų rinkinys nėra subalansuotas, nes dauguma kodų porų nėra klonai. Pavyzdžiui, eksperimentuose naudoto testavimo duomenų rinkinyje iš 10 000 programinių kodų porų tik 668 buvo klonai. Toks nesubalansuotas duomenų rinkinys gali atsiliepti modelio mokymo ir testavimo rezultatams.

Java programavimo kalbai kodų klonų aptikimo uždaviniui buvo naudojamas *BigCloneBench* duomenų rinkinys [SR15]. Šis duomenų rinkinys yra praktikoje rašomų programinių kodų etalonas, turintis per 8 milijonų programinių kodų porų, kur kiekvienas programinis kodas atitinka vieną metodą. Šį duomenų rinkinį galima suskaidyti į skirtingų tipų kodų klonų rinkinius taikant funkcinį panašumą [WL17]. Kadangi šis modelis vėliau pritaikytas ekvivalenčių mutavusių kodų aptikimo uždaviniui, nuspręsta iš *BigCloneBench* duomenų rinkinio eksperimente naudoti tik 3 tipo klonus, kurių panašumas papuola į intervalą $[0.7, 1)$. Čia panašumo metrika yra lygi kodo eilučių

daliai, kurią turi abu tiriami kodai atlikus kodo normalizaciją, kuri atliekama iš kodų pašalinant komentarus, spausdinimo į ekraną eilutes ir pervadinant visus identifikatorius (angl. *identifier*) ir absoliučias reikšmes į iš anksto nustatytą reikšmę [SIK+14]. Gauto duomenų rinkinio dydis yra 38 294 programinių kodų poros. Šis duomenų rinkinys suskaidytas į mokymo, validavimo ir testavimo duomenų rinkinius santykiu 60 %, 20 %, 20 %. Šis duomenų rinkinys yra subalansuotas, nes lyginant testavimo duomenų rinkinį, iš 7636 programinių kodų porų 3650 yra klonai. Tikimasi, kad išskirtinai ryškaus trečiojo tipo klonų ir subalansuoto duomenų rinkinio naudojimas leis *Java* programavimo kalbos klonus nagrinėjantiems modeliams pasiekti neblogus rezultatus.

5.2.2. Modelių mokymų informacija

Neuroninių tinklų modelių mokymai vykdyti naudojant *HP ProBook 455 G8 Notebook PC* nešiojamąjį kompiuterį su *AMD Ryzen 5 5600U* 6 branduolių procesoriumi ir *Windows 10* operacine sistema. Kompiuteris neturi integruotos vaizdo plokštės, todėl skaičiavimai buvo atliekami naudojant procesorių neišnaudojant efektyvios mašininio mokymosi skaičiavimo bibliotekos *Cuda*.

Medžio struktūros modelį C programavimo kalbos kodų klonų uždaviniui mokyti prireikė 10 h 37 min. Metrikų modeliams C programavimo kalbos kodų klonų uždaviniui mokyti 250 epochų reikdavo apie 30 min, o 80-čiai epochų – 10 min.

Medžio struktūros modelį *Java* programavimo kalbos kodų klonų uždaviniui mokyti prireikė apie 10 h. Metrikų modeliams *Java* programavimo kalbos kodų klonų uždaviniui mokyti 250 epochų reikdavo apie 23 min, o 80-čiai epochų – 7 min.

5.2.3. Modelių, taikomų C kalbai, testavimo rezultatai

Suprogramavus metrikų modelio struktūros neuroninį tinklą, jis buvo mokomas ir lyginami gauti rezultatai. Modeliai palyginami tikslumo (angl. *precision*), išsamumo (angl. *recall*) ir $f1$ įverčiais. Tikslumo įvertis p pažymi, kokia identifikuota pradinio duomenų rinkinio teigiamų atvejų dalis, išsamumo įvertis r pažymi, kokia duomenų rinkinio teigiamai pažymėtų atvejų dalis yra identifikuota klaidingai, o $f1$ įvertis, kuris įvertina bendrą modelio tikslumą, yra žymimas tokia formule:

$$f1 = 2\frac{pr}{p+r}$$

5.2.3.1. Metrikų modelio, taikomo C kalbai, testavimo rezultatai

Iš pradžių kodų klonų aptikimo uždaviniui spręsti suformuotas 57 metrikų sąrašas, kuriuo bandyta mokyti metrikų modelį. Modelio praradimo ir tikslumo funkcijų grafikai (41 ir 42 pav.) rodo, kad modelio mokymosi ir validavimo kreivės išsiskiria, o tai pažymi, kad įvyko modelio permokymas (angl. *model overfitting*). Kreivės seka viena kitą iki maždaug 20–25 epochos, toliau iki 50–60 epochos validavimo kreivės pradeda nusistovėti, o nuo 60 epochos pradeda didėti kreivės pokyčio amplitudė ir nebepateikia stabilių rezultatų. Idealiu atveju treniravimo ir validavimo kreivės turėtų sekti viena paskui kitą, kad išmokytas modelis atpažintų realaus pasaulio atvejus. Tačiau mokymosi ir validavimo kreivės išsiskiria. Tai pažymi, kad modelis, išmokytas pagal treniravimo

duomenų rinkinį, nėra gabus atpažinti realių ir jam dar nematytų scenarijų. Kad ši problema būtų išspręsta, reikia išspręsti modelio permokymo problemą.

Pagrindiniai galimi modelio permokymo sprendimo būdai:

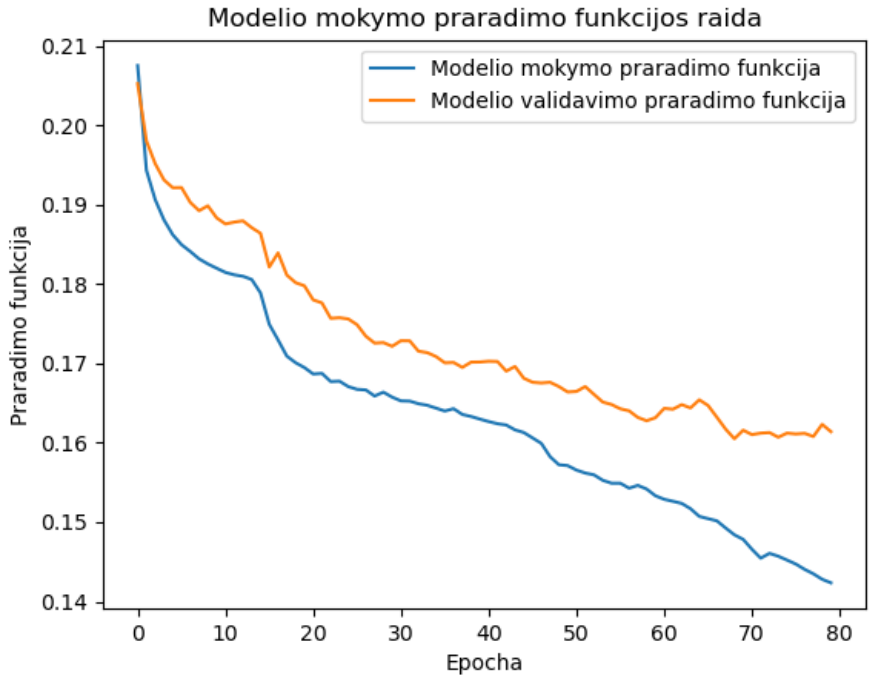
1. Ankstyvasis stabdymas (angl. *early stopping*). Tai dažnai taikomas metodas giliajame neuroninių tinklų mokyme. Jis paremtas tuo, kad iki tam tikro slenksčio modelio tikslumas gerėja, bet po kurio laiko dingsta bendrų bruožų atpažinimas ir modelis persimoko mokymo duomenų atžvilgiu. Todėl modelio mokymą galima stabdyti, kai jis pasiekia didžiausią tikslumą. Šiuo atveju mokymas turėtų būti stabdomas apie 50–60 epochą, kai pradeda kilti modelio validavimo praradimo funkcija (41 pav.) ir kristi modelio validavimo tikslumas (42 pav.).

2. Parametrų prastinimas. Problema galėtų būti išsprendžiama pateikiant didesnį duomenų rinkinį modeliui mokyti, kuris padengtų daugiau galimų parametrų kombinacijų. Turimame duomenų rinkinyje yra daug kodų porų, kurie turi retą parametrą ar parametrų kombinaciją, dėl kurios modelis išmoksta atpažinti parametrų kombinaciją kaip darančią įtaką įverčiui, nors tie parametrai neturėtų turėti jokios įtakos įverčiui.

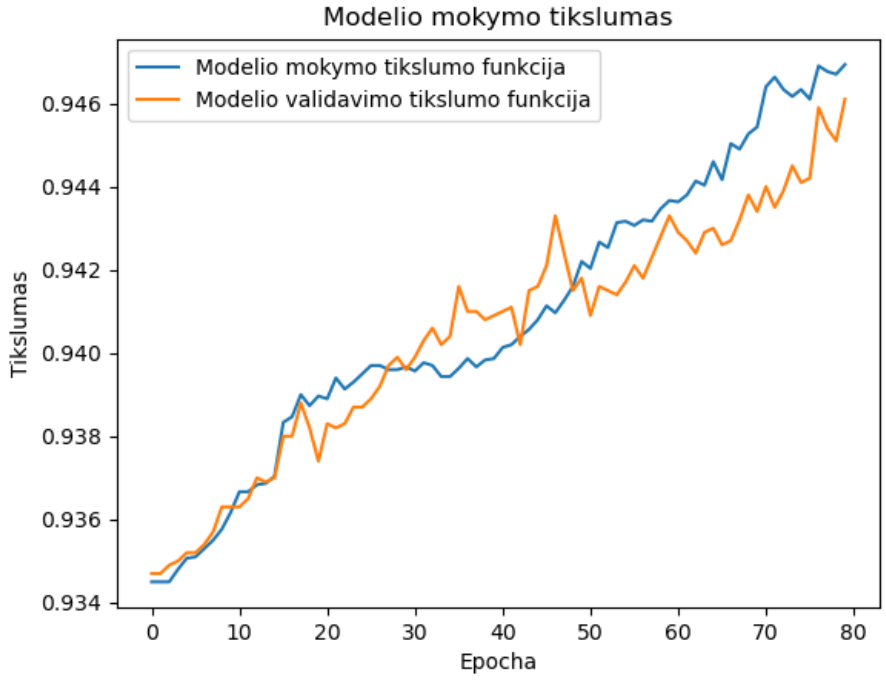
Suformuotas naujas 44 metrikų sąrašas, pašalinant nišines ir retai pasitaikančius elementus fiksuojančias metrikas, tačiau paliekant parametrus, pažyminčius palyginimo ir aritmetinius operatorius.

Pagal 46 ir 47 pav., kai modelis išmokytas esant 44 metrikų sąrašui, validavimo tikslumo ir praradimo funkcijos ilgiausiai sekė mokymo tikslumo ir validavimo funkcijas (iki maždaug 80 epochos). Siekiant gauti tikslesnį modelį, reikėtų pritaikyti ankstyvųjį stabdymą, kad gauto modelio mokymo rezultatai atitiktų validavimo ir testavimo rezultatus.

Modelio mokymo rezultatai taikant ankstyvųjį stabdymą pateikti 9 ir 10 pav., modelis mokytas iki 80-tos epochos.

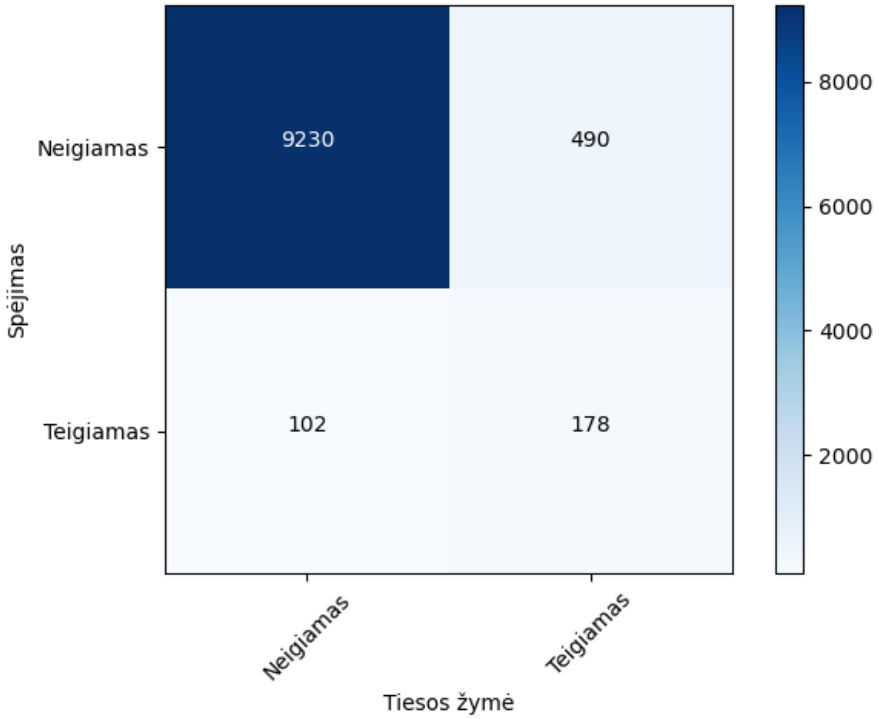


9 pav. Trečiosios versijos metrikų modelio praradimo funkcija *BCELoss*



10 pav. Trečiosios versijos metrikų modelio tikslumo funkcija

Modelio klaidų matrica pateikta 11 pav.

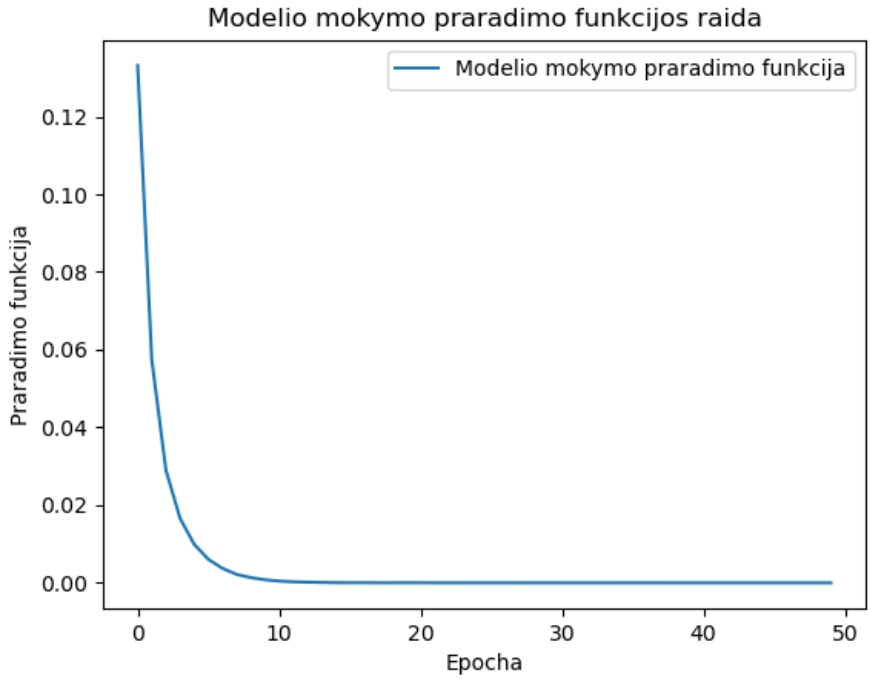


11 pav. Kodų klonų metrikų modelio klaidų matrica

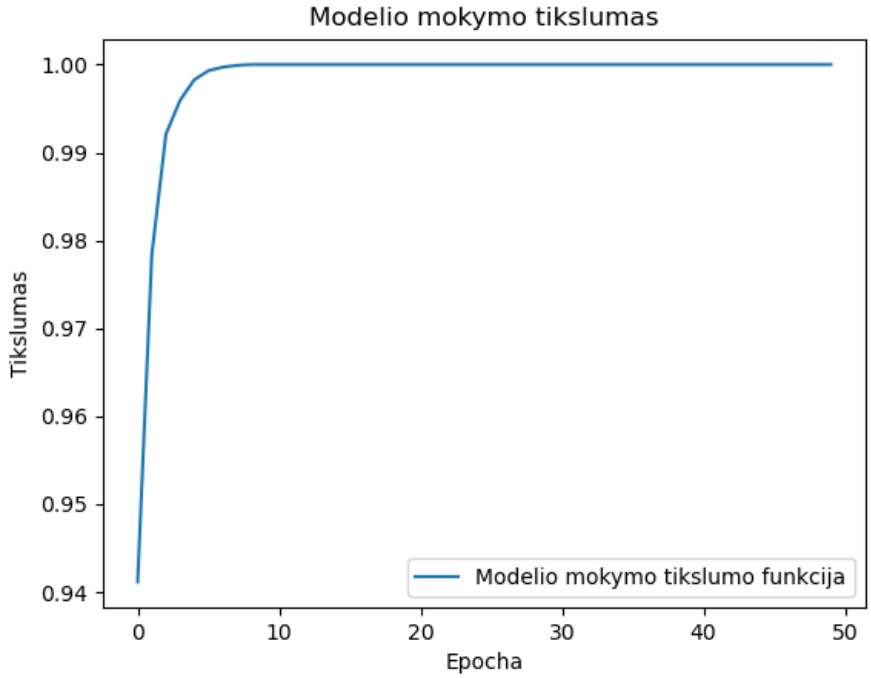
Šis modelis pasiekė 94,08 % bendrą tikslumą. Identifikuotų elementų tikslumas pasiekė 63,6 %, nes iš 280 teigiamai identifikuotų porų 178 buvo klonai. Tuo tarpu identifikuotų elementų išsamumas siekia 26,6 %, nes iš 668 klonų duomenų rinkinyje 178 buvo pažymėti teigiamai. Šio modelio f1 įvertis lygus 0,376.

5.2.3.2. Medžio struktūros modelio, taikomo C kalbai, testavimo rezultatai

Medžio struktūros modelio mokymo rezultatai pateikti 12 ir 13 pav. Modelis mokytas taikant [ZWZ+19] autorių pateiktą eksperimento kodą su 50 epochų.

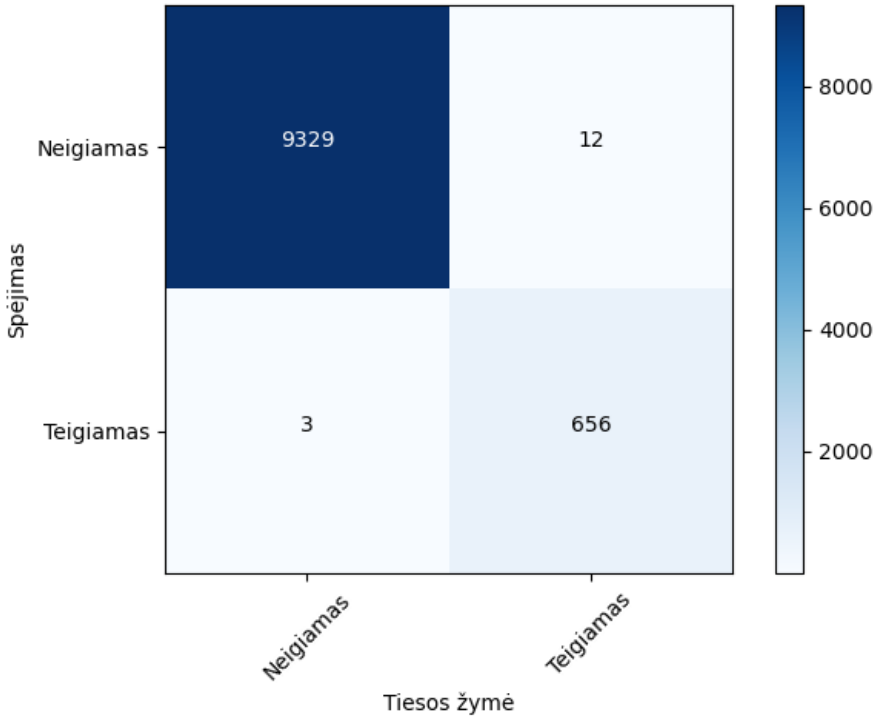


12 pav. Kodų klonų medžio struktūros modelio praradimo funkcija *BCELoss*



13 pav. Kodų klonų medžio struktūros modelio tikslumo funkcija

Pagal 12 ir 13 pav. modelio mokymas rodo stabilius rezultatus, nes tiek praradimo, tiek tikslumo funkcijos po kurio laiko nusistovi ir rodo gerus rezultatus.



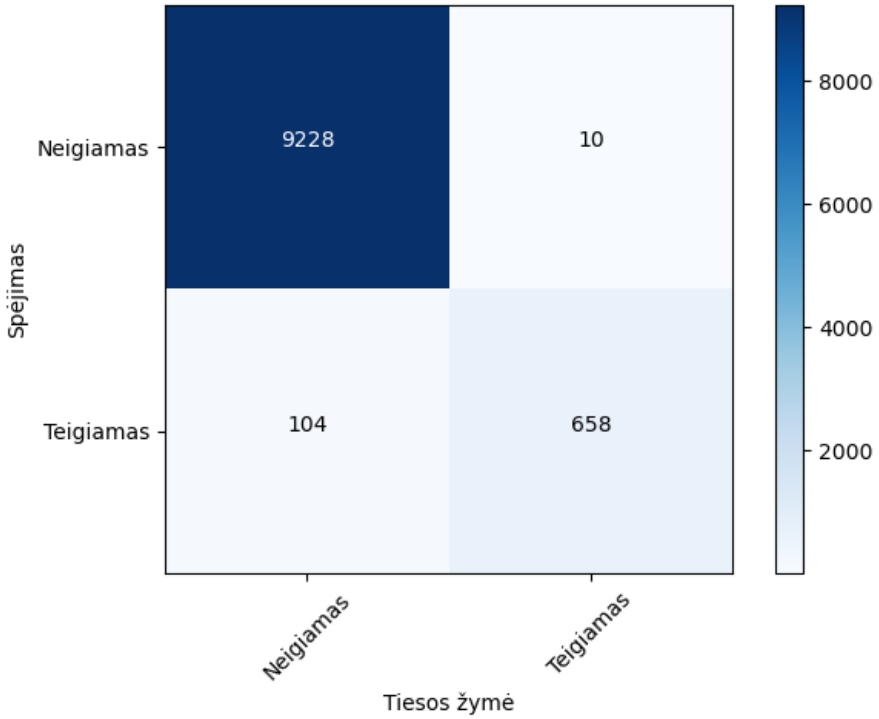
14 pav. Kodų klonų medžio struktūros modelio klaidų matrica

14 pav. pateikti abstraktaus sintaksės medžio struktūros modelio testavimo rezultatai klaidų matricos forma. Bendras modelio tikslumas yra 99,9 %, nes iš 10 000 testavimo rinkinio atvejų 9985 buvo identifikuoti teisingai. Identifikuotų elementų tikslumas pasiekė 99,5 %, nes iš 659 teigiamai identifikuotų porų 656 buvo klonai, o identifikuotų elementų išsamumas siekia 98,2 %, nes iš 668 klonų duomenų rinkinyje 656 buvo pažymėti teigiamai. Šio modelio f1 įvertis lygus 0,989.

Medžio struktūros modelio rezultatai gauti geresni nei originalaus straipsnio autorių [ZWZ+19] (tikslumas 98,9 %, išsamumas 92,7 %, f1 įvertis 95,5 %) greičiausiai dėl kitokio pasirinkto elementų paketo dydžio (angl. *batch size*) (32 vietoj 64), epochų skaičiaus (50 vietoj 5) ar dėl tiesiog palankiau sukritusio testavimų atvejų rinkinio dalijant visą *OJClone* duomenų rinkinį į mokymo, validavimo ir testavimo duomenų rinkinius.

5.2.3.3. Medžio struktūros ir metrikų modelių. taikomų C kalbai, kombinuoti rezultatai

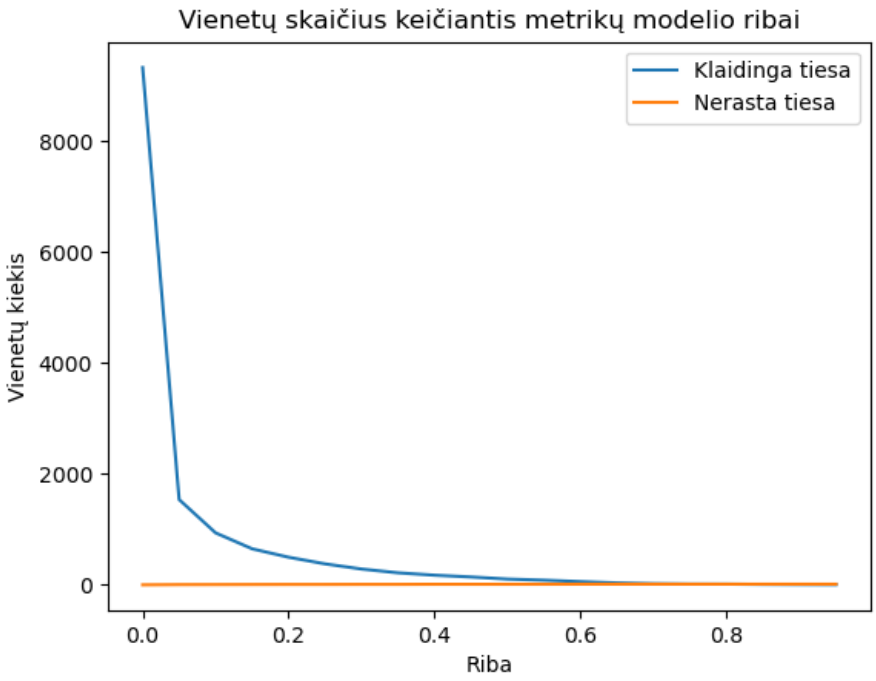
Metrikų ir medžio struktūros modeliams palyginti 15 pav. pateikta kombinuoto modelio (8 pav.) klaidų matrica. Čia kodų pora pažymima kaip klonai tuo atveju, jeigu nors vienas iš modelių taip pažymėjo tą kodų porą.



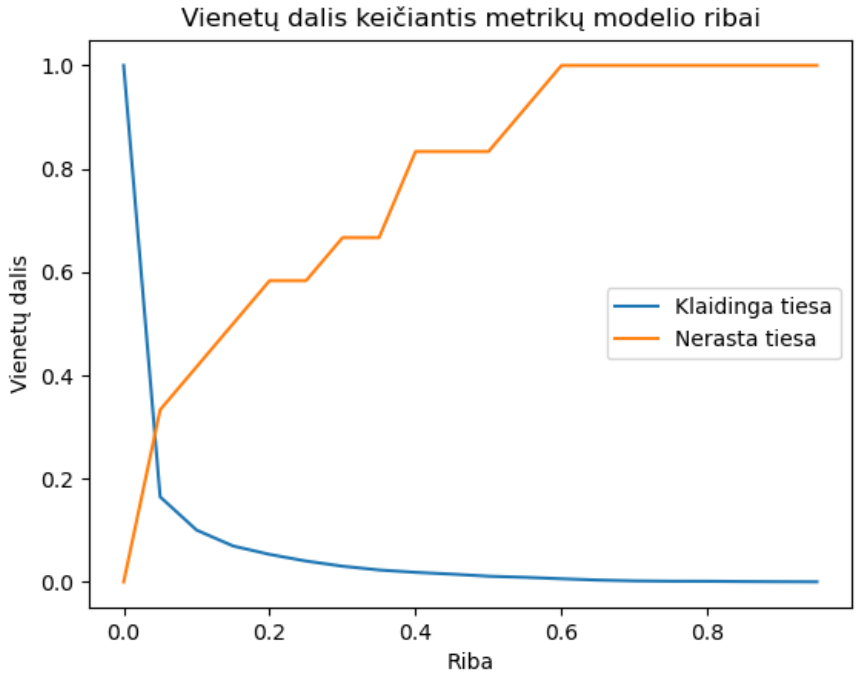
15 pav. Kodų klonų kombinuoto modelio klaidų funkcija

Pagal 15 pav. iš 12 medžio struktūros neidentifikuotų kodų klonų porų 2 aptiko metrikų modelis ir taikant kombinuotą modelį liko 10 neidentifikuotų kodų klonų porų, taip pagerinant modelio išsamumą iki 98,5 %. Tačiau suprastėjo identifikuotų kodų klonų porų tikslumas, nes iš 762 identifikuotų kodų klonų porų 104 buvo identifikuoti klaidingai, taip sumažinant tikslumą iki 86,4 %. Todėl galutinio modelio f1 įvertis yra lygus 92 %.

Kadangi metrikų modelis grąžina tikimybę, ar kodų pora yra ekvivalenti, ar ne, galima palyginti rezultatus keičiantis nustatytai modelio ribai δ (angl. *threshold*). 16 ir 17 pav. yra pateikta, kokia kombinuoto modelio kodų porų dalis buvo identifikuota klaidingai keičiantis metrikų modelio ribai. Kadangi pradinis medžio struktūros modelis nerado tik 12 vienetų, kuriuos buvo bandoma aptikti kombinuojant modelius, nerastos tiesos grafikai yra kampuoti ir reikėtų atlikti daugiau tyrimų norint pateikti apie juos tikslesnes išvadas.



16 pav. Kodų klonų klaidingai identifikuotų vienetų kiekis keičiantis metrikų modelio identifikavimo ribai



17 pav. Kodų klonų klaidingai identifikuotų vienetų dalis keičiantis metrikų modelio identifikavimo ribai

Pagal 16 pav., kai riba lygi 0, visos kodų poros identifikuojamos kaip klonai, todėl klaidingos tiesos įvertis lygus visos aibės dydžiui, atmetus tikrų kodų klonų poras. O kai riba lygi 1, klaidingos tiesos ir nerastos tiesos įverčiai tampa lygūs 3 ir 12, o tai atitinka medžio struktūros modelio

testavimo rezultatus. Ribai esant tarp 0 ir 1, klaidingos tiesos ir nerastos tiesos įverčiai pamažu keičiasi. Pagal 17 pav. nerastos tiesos kreivė rodo, kokia yra išlikusi medžio struktūros modelio nerastų kodų klonų porų dalis. Kai riba lygi 0,05, pažymėta, kad neidentifikuota 4 iš 12 kodų klonų porų, o kai riba lygi 0,5, neidentifikuota 10 iš 12 kodų klonų porų. Klaidingos tiesos kreivė, kai riba lygi 0,05, klaidingai pažymi 1648 poras klonais, o ties riba, lygia 0,5, šis skaičius nukrinta iki 104. Remiantis šiais kombinuoto modelio testavimo rezultatais priklausomai nuo poreikio galima pasirinkti metrikų modelio ribą ir taikyti modelį įvairiems kodų klonų aptikimo uždavinio sprendimams.

Bendru atveju kombinuotas modelis suprastina medžio struktūros modelio rezultatus ir nepateisina naudojimo dėl tikslumo sumažėjimo. Tačiau ekvivalenčių mutavusių kodų aptikimo uždavinyje galima argumentuoti, kad aukštą modelio išsamumą yra pasiekti svarbiau nei tikslumą. Taip yra todėl, kad dėl prastesnio modelio tikslumo įverčio mutacinio testavimo procese būtų sunaikinama daugiau kodų porų, tačiau tai atperka ekvivalenčių mutavusių kodų kiekio įverčio sumažinimas. Neaptikti ekvivalentūs mutavę kodai sumažina galutinio testavimų atvejų rinkinio įvertį ir yra pagrindinė problema, kuri trukdo pasitikėti mutacinio testavimo rezultatais. Todėl mutaciniame testavime sprendžiant ekvivalenčių mutavusių kodų problemą galima pateisinti orientavimąsi į modelio išsamumo įvertį šiek tiek susilpninant modelio tikslumą.

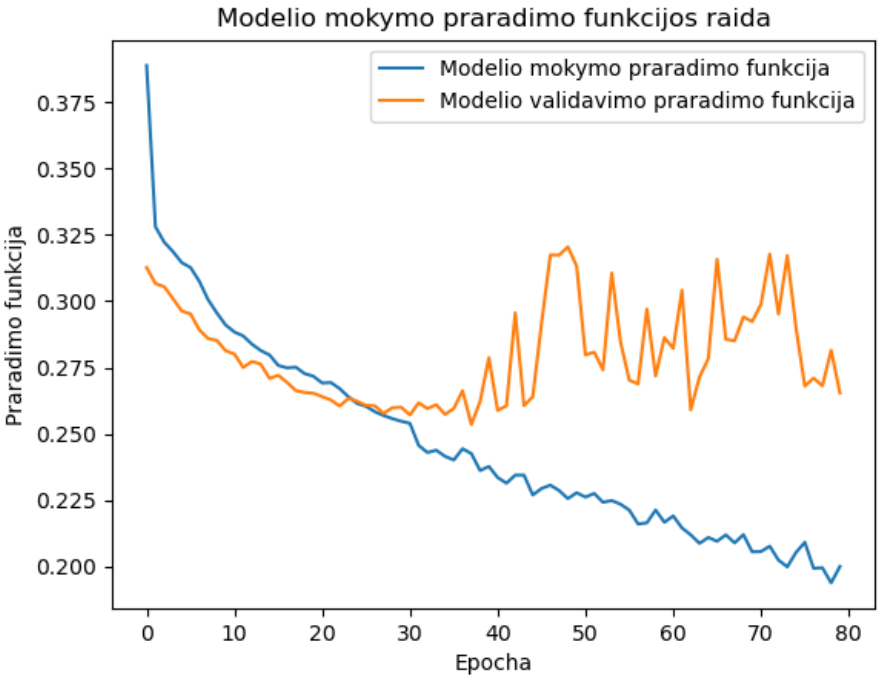
5.2.4. Metrikų modelio, taikomo *Java* kalbai, testavimo rezultatai

Šiame poskyryje aprašyti gauti galutinio metrikų modelio, skirto *Java* programavimo kalbos klonams atpažinti, rezultatai.

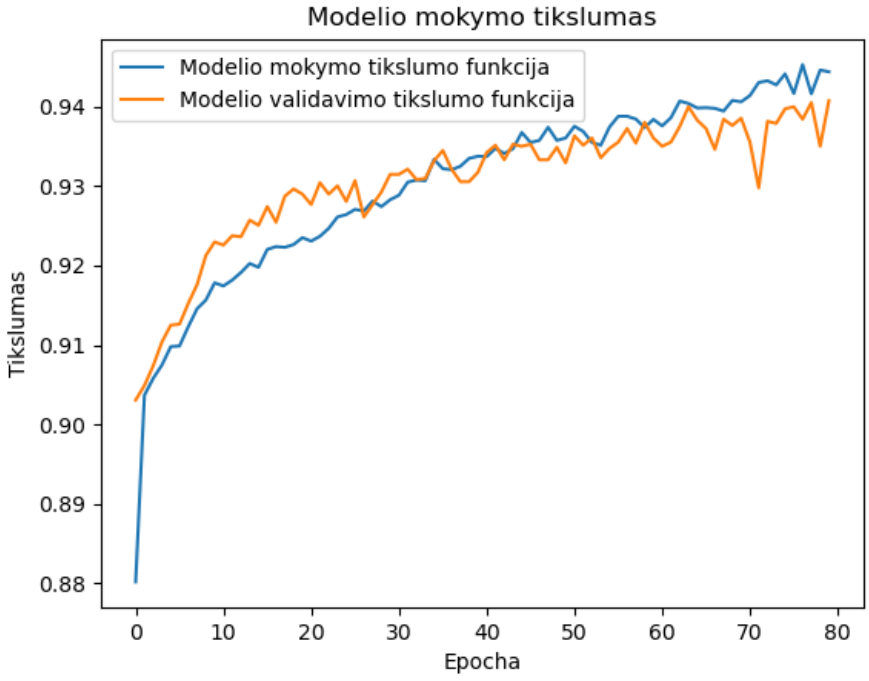
5.2.4.1. Metrikų modelio versijos rezultatai

Suformuotas 44 *Java* kodo metrikų sąrašas, sudarytas panašiu detalumu kaip ir galutinis C kodo metrikų sąrašas, kuriame palikti parametrai, žymintys palyginimo ir aritmetinius operatorius. Pagal 48 ir 49 pav., kai modelis mokytas naudojant 44 *Java* kodo metrikų sąrašą, validavimo tikslumo ir praradimo funkcijos, panašiai kaip ir C kodo metrikų modelyje, ilgiausiai sekė mokymo tikslumo ir validavimo funkcijas (iki maždaug 80 epochos). Siekiant gauti tikslesnį modelį, reikėtų pritaikyti ankstyvųjį stabdymą, kad gauto modelio mokymo rezultatai atitiktų validavimo ir testavimo rezultatus.

Modelio mokymo rezultatai, taikant ankstyvųjį stabdymą, pateikti 18 ir 19 pav., modelis mokytas iki 80-tos epochos.

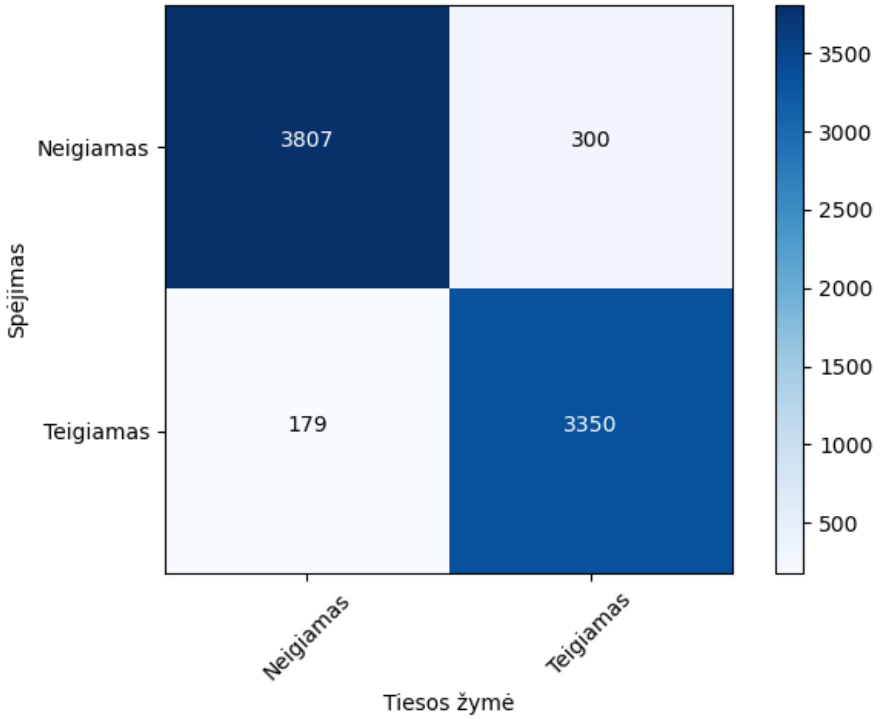


18 pav. Kodų klonų metrikų modelio praradimo funkcija *BCELoss*



19 pav. Kodų klonų metrikų modelio tikslumo funkcija

Modelio klaidų matrica pateikta 20 pav.

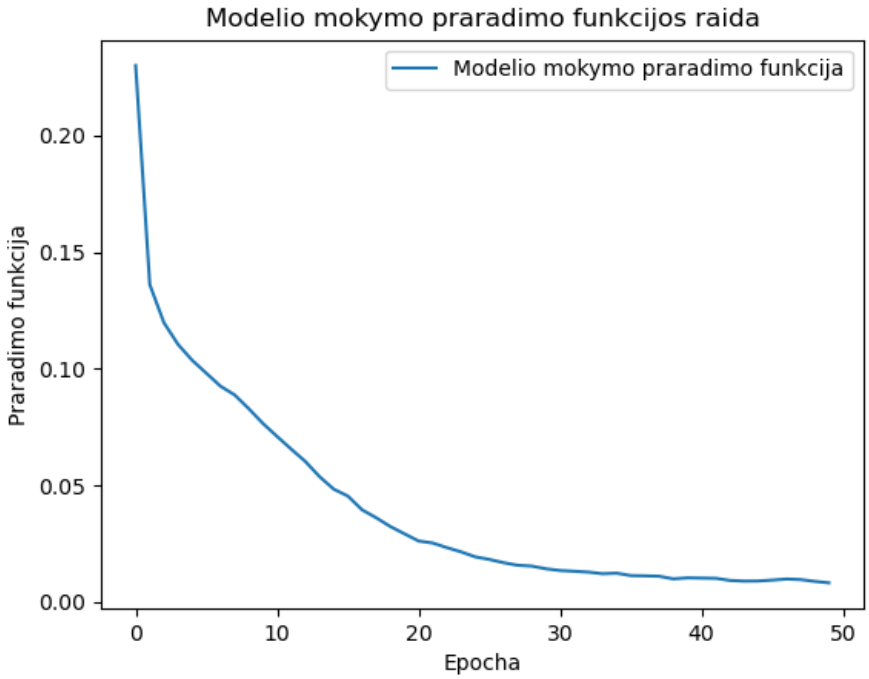


20 pav. Kodų klonų metrikų modelio klaidų matrica

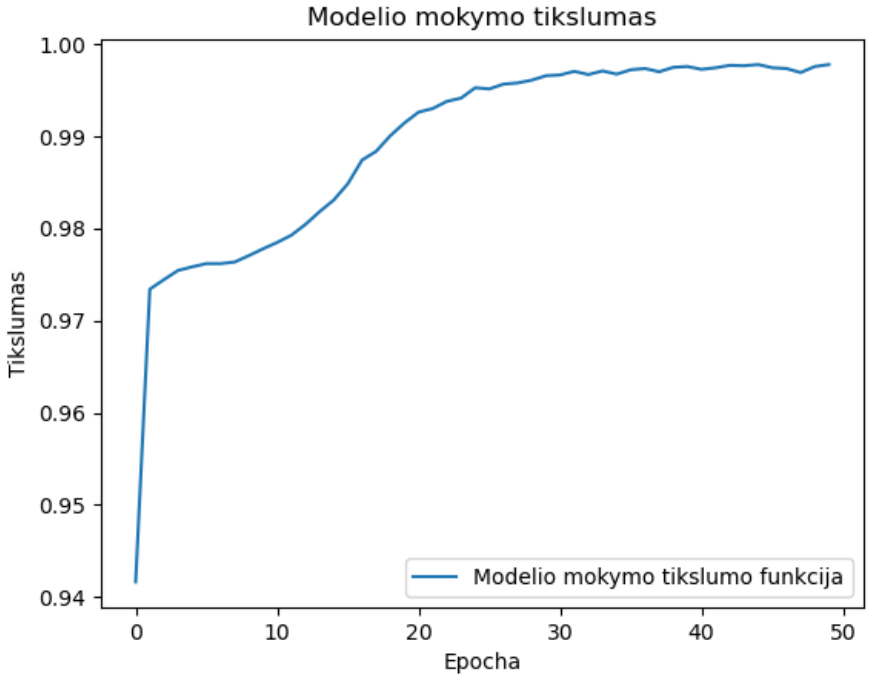
Šis modelis pasiekė 93,73 % bendrą tikslumą. Identifikuotų elementų tikslumas pasiekė 94,9 %, nes iš 3529 teigiamai identifikuotų porų 3350 buvo klonai, o identifikuotų elementų išsamumas siekia 91,8 %, nes iš 3650 klonų duomenų rinkinyje 3350 buvo pažymėti teigiamai. Šio modelio f1 įvertis lygus 0,933.

5.2.4.2. Medžio struktūros modelio, taikomo *Java* kalbai, rezultatai

Medžio struktūros modelio mokymo rezultatai pateikti 21 ir 22 pav. Modelis mokytas taikant [ZWZ+19] autorių pateiktą eksperimento kodą su 50 epochų.

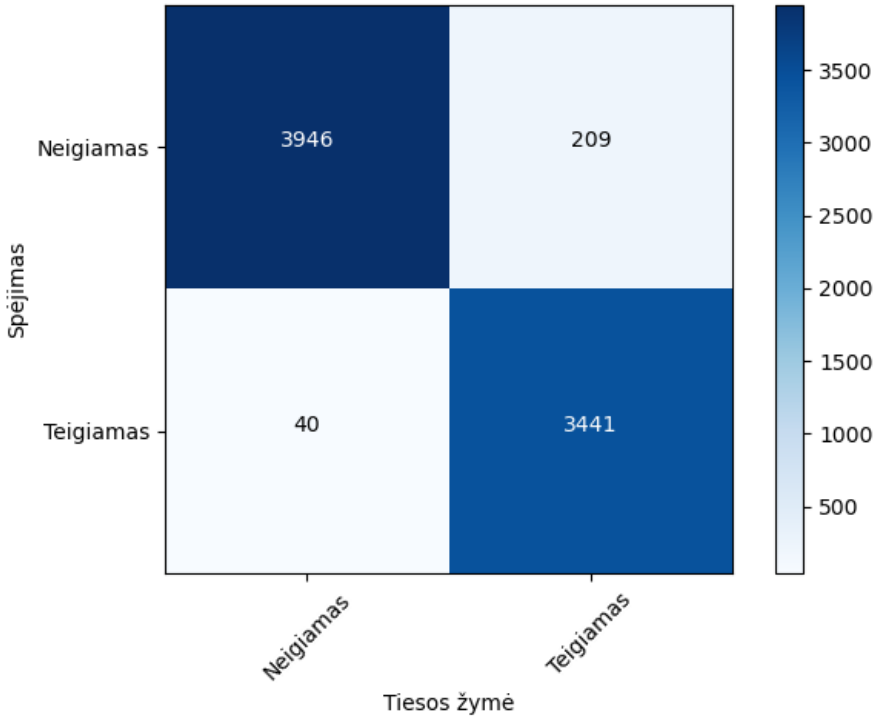


21 pav. Kodų klonų medžio struktūros modelio praradimo funkcija *BCELoss*



22 pav. Kodų klonų medžio struktūros modelio tikslumo funkcija

Pagal 21 ir 22 pav. modelio mokymas rodo stabilius rezultatus, nes tiek praradimo, tiek tikslumo funkcijos po kurio laiko nusistovi ir rodo gerus rezultatus. Tačiau modeliui pasiekti stabilius rezultatus prireikė daugiau epochų nei mokant C kalbos medžio struktūros modelį.



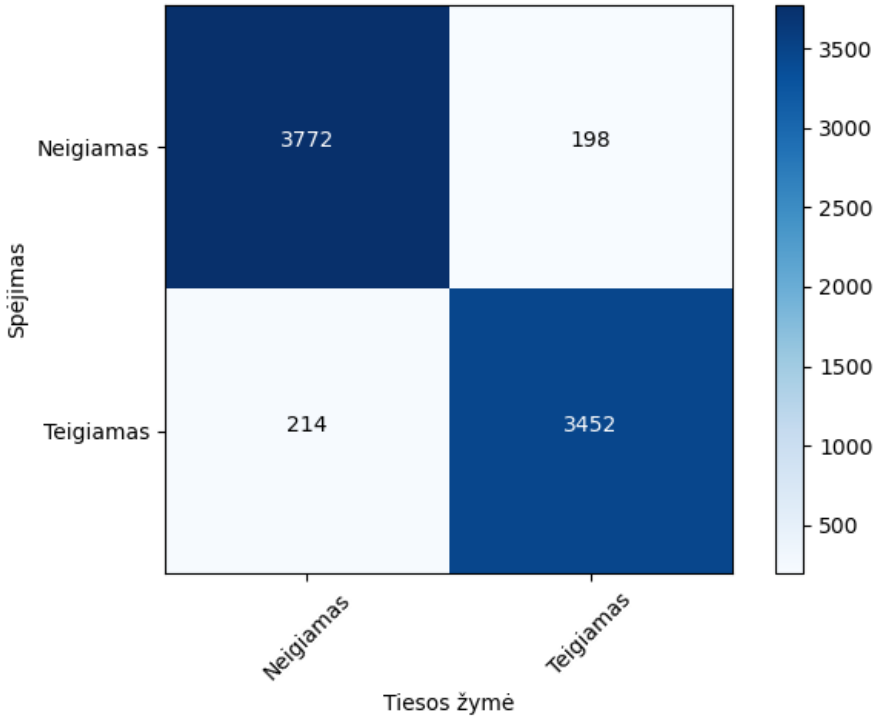
23 pav. Kodų klonų medžio struktūros modelio klaidų matrica

23 pav. pateikti *Java* kalbos abstraktaus sintaksės medžio struktūros modelio testavimo rezultatai klaidų matricos forma. Bendras modelio tikslumas yra 96,7 %, nes iš 7636 testavimo rinkinio atvejų 7387 buvo identifikuoti teisingai. Identifikuotų elementų tikslumas pasiekė 98,9 %, nes iš 3481 teigiamai identifikuotų porų 3441 buvo klonai. O identifikuotų elementų išsamumas siekia 94,3 %, nes iš 3650 klonų duomenų rinkinyje 3441 buvo pažymėti teigiamai. Šio modelio f1 įvertis lygus 0,965.

Medžio struktūros modelio ryškaus trečiojo tipo klonams atpažinti gauti rezultatai panašūs, nei originalaus straipsnio autorių [ZWZ+19] (tikslumas 99,9 %, išsamumas 94,2 %, f1 įvertis 97,0 %) nepaisant kitokio pasirinkto elementų paketo dydžio (angl. *batch size*) (32 vietoj 64) ir epochų skaičiaus (50 vietoj 5). Kadangi modelio mokymo tikslumas viršijo 99 %, nors testavimo tikslumas buvo tik 96,7 %, galima nedidelio modelio persimokymo galimybė. Kita vertus, taip pat įtakos galėjo turėti ir kitaip sukritęs duomenų rinkinys.

5.2.4.3. Medžio struktūros ir metrikų modelių, taikomų *Java* kalbai, kombinuoti rezultatai

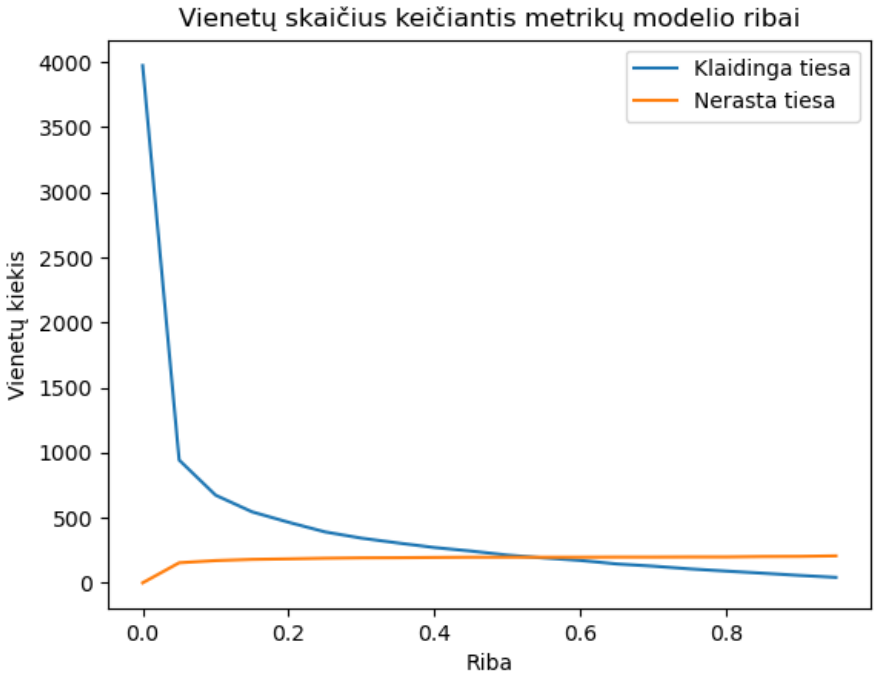
Metrikų ir medžio struktūros modeliams palyginti 24 pav. pateikta kombinuoto modelio (8 pav.) klaidų matrica. Čia kodų pora pažymima kaip klonai tuo atveju, jeigu nors vienas iš modelių taip pažymėjo tą kodų porą.



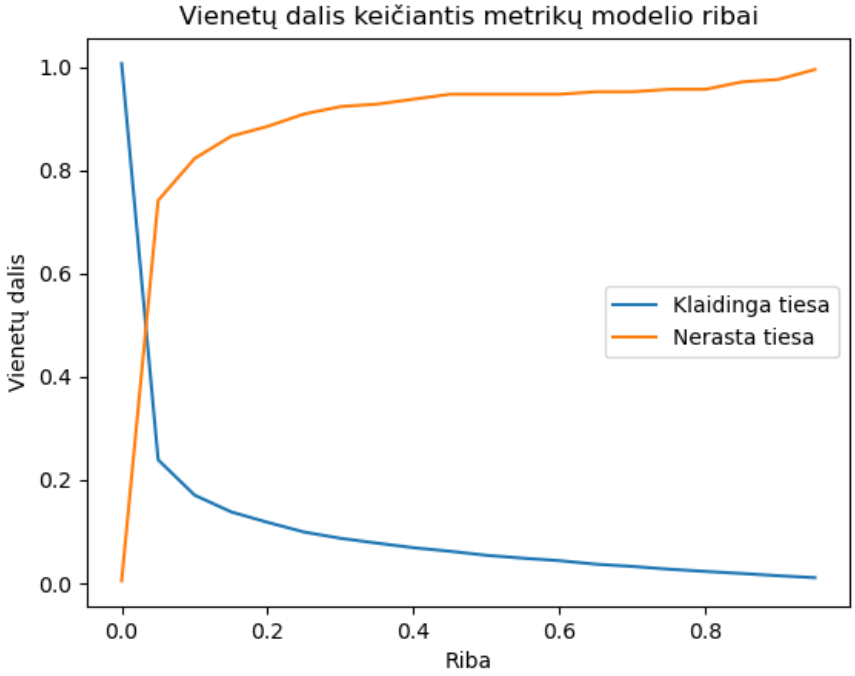
24 pav. Kombinuoto modelio klaidų funkcija

Pagal 24 pav. iš 209 medžio struktūros neidentifikuotų kodų klonų porų 6 aptiko metrikų modelis ir taikant kombinuotą modelį liko 198 neidentifikuotų kodų klonų porų, taip pagerinant modelio išsamumą iki 94,6 %. Tačiau suprastėjo identifikuotų kodų klonų porų tikslumas, nes iš 3666 identifikuotų kodų klonų porų 214 buvo identifikuoti klaidingai, taip sumažinant tikslumą iki 94,16 %. Todėl galutinio modelio f1 įvertis yra lygus 94,4 %.

Kadangi metrikų modelis grąžina tikimybę, ar kodų pora yra ekvivalenti, ar ne, galima palyginti rezultatus keičiantis nustatytai modelio ribai δ (angl. *threshold*). 25 ir 26 pav. yra pateikta, kokia kombinuoto modelio kodų porų dalis buvo identifikuota klaidingai keičiantis metrikų modelio ribai. Kadangi pradinis medžio struktūros modelis nerado 209 *Java* kodo porų, nerastos tiesos grafikai yra tikslesni, nei C kodui ir apie juos galima daryti tikslesnes išvadas.



25 pav. Klaidingai identifikuotų vienetų kiekis keičiantis metrikų modelio identifikavimo ribai



26 pav. Klaidingai identifikuotų vienetų dalis keičiantis metrikų modelio identifikavimo ribai

Pagal 25 pav., kai riba lygi 0, visos kodų poros identifikuojamos kaip klonai, todėl klaidingos tiesos įvertis lygus visos aibės dydžiui, atmetus tikrų kodų klonų poras. O kai riba lygi 1, klaidingos tiesos ir nerastos tiesos įverčiai tampa lygūs 40 ir 209, tai atitinka medžio struktūros modelio testavimo rezultatus. Ribai esant tarp 0 ir 1, klaidingos tiesos ir nerastos tiesos įverčiai pamažu keičiasi. Pagal 26 pav. nerastos tiesos kreivė rodo, kokia yra išlikusi medžio struktūros modelio nerastų

kodų klonų porų dalis. Kai riba lygi 0,05, pažymėta, kad neidentifikuota 155 iš 209 kodų klonų porų, o kai riba lygi 0,5, neidentifikuota 198 iš 209 kodų klonų porų. Klaidingos tiesos kreivė, kai riba lygi 0,05, klaidingai pažymi 944 poras klonais, o ties riba, lygia 0,5, šis skaičius nukrinta iki 214. Remiantis šiais kombinuoto modelio testavimo rezultatais priklausomai nuo poreikio atitinkamai kaip ir C kodo modelyje galima pasirinkti metrikų modelio ribą ir taikyti modelį įvairiems kodų klonų aptikimo uždavinio sprendimams. Šį modelį taip pat galima pritaikyti ekvivalenčių mutavusių kodų aptikimo uždavinyje, orientuojantis į išsamumo įverčio sumažinimą.

5.2.5. Modelių rezultatų suvestinė

Modelių rezultatų suvestinė pateikta 4 lentelėje. Čia *OJClone* duomenų rinkinys buvo naudojamas mokyti atpažinti C programavimo kalbos klonus, o *BigCloneBench* buvo naudojamas *Java* programavimo kalbos klonams atpažinti. Visi kodų klonų tipai apima kodų klonus nuo pirmo iki ketvirto tipo imtinai. O ryškaus trečio tipo klonai apima kodų poras, kurių panašumo indeksas papuola į intervalą $[0.7, 1)$.

4 lentelė. Modelių rezultatų suvestinė

Modelis	Duomenų rinkinys	Klonų tipas	Tikslumas	Išsamumas	f1
Medžio struktūros modelis	*OJClone*	Visi	0,995	0,982	0,989
Metrikų modelis	*OJClone*	Visi	0,636	0,266	0,376
Kombinuotas modelis	*OJClone*	Visi	0,864	0,985	0,920
Medžio struktūros modelis	*BigCloneBench*	Ryškus trečias tipas	0,989	0,943	0,965
Metrikų modelis	*BigCloneBench*	Ryškus trečias tipas	0,949	0,918	0,933
Kombinuotas modelis	*BigCloneBench*	Ryškus trečias tipas	0,941	0,946	0,944
Medžio struktūros modelis [ZWZ+19]	*OJClone*	Visi	0,989	0,927	0,955
Medžio struktūros modelis [ZWZ+19]	*BigCloneBench*	Ryškus trečias tipas	0,999	0,942	0,970
Medžio struktūros modelis [ZWZ+19]	*BigCloneBench*	Visi	0,998	0,884	0,938
Oreo metrikų modelis [SFL+18]	*BigCloneBench*	Ryškus trečias tipas	-	0,890	-
Oreo metrikų modelis [SFL+18]	*BigCloneBench*	Visi	0,895	-	-
CloneWorks indeksuotų leksemų modelis [SFL+18]	*BigCloneBench*	Ryškus trečias tipas	-	0,930	-
CloneWorks indeksuotų leksemų modelis [SFL+18]	*BigCloneBench*	Visi	0,987	-	-
NiCad tekstinis modelis [SFL+18]	*BigCloneBench*	Ryškus trečias tipas	-	0,930	-
NiCad tekstinis modelis [SFL+18]	*BigCloneBench*	Visi	0,990	-	-

4 lentelės pirmoje dalyje pateikti šio tyrimo metu gauti rezultatai, o antroje lentelės dalyje pateikti kitų tyrimų, surinktų iš darbų, kuriais buvo remtasi, geriausi rezultatai. Iš šio tyrimo metu

gautų rezultatų medžio struktūros modelis pateikė daug geresnius rezultatus taikant C kalbos modelius ir šiek tiek geresnius rezultatus taikant *Java* kalbos modelius. Kombinuotų modelių $f1$ įvertis yra kiek mažesnis nei medžio struktūros modelių, tačiau išsamumo įvertis yra šiek tiek didesnis.

Rezultatams palyginti papildomai pateikti abstrakčios sintaksės medžio [ZWZ+19], *Oreo* metrikų modelio [SFL+18] ir kitų dviejų bazinių modelių *Cloneworks* [SR17] ir *NiCad* [RC08] rezultatai. Tačiau kitų autorių gauti rezultatai yra gauti pritaikius testavimo rinkinį iš kitų kodų porų, nepaisant to, kad modeliams testuoti naudoti tie patys duomenų rinkiniai, todėl negalima aklai lyginti modelių tarpusavio rezultatų. Taip pat *Oreo*, *Cloneworks* ir *NiCad* modeliai mokyti naudojant kitą duomenų rinkinį vietoj *BigCloneBench* duomenų rinkinio, todėl šių modelių testavimo rezultatų įverčiai gali būti kiek silpnesni nei kiti rezultatai. Didžiausią tikslumą pateikė medžio struktūros modelis tiek šio tyrimo metu, tiek [ZWZ+19] tyrimo metu taikant *Java* ir C kalbas (tarp 0,989 ir 0,999) ir *NiCad* modelis taikant *Java* kalbą (0,990). Didžiausią išsamumą C kalbai pateikė šio tyrimo metu išmokytas medžio struktūros modelis (0,982), kurį nežymiai patobulino kombinuotas modelis (0,985). Didžiausią f1 įvertį taip pat pateikė medžio struktūros modeliai (0,989 C kalbai ir 0,965–0,970 *Java* kalbai). O didžiausią išsamumą *Java* kalbai pateikė medžio struktūros modeliai (0,942–0,943), kur darbe įgyvendintą versiją nežymiai patobulino kombinuotas modelis (0,946). Taip parodyta, kad taikant modelių kombinacijas sumažinant modelio tikslumą galima pagerinti išsamumo įvertį.

6. Ekvivalenčių mutavusių kodų aptikimo uždavinio sprendimas

Kodų klonų aptikimo uždaviniui kaip įvestis gali būti pateikiami bet kokie du kodai, kuriems yra pateikiamas atsakymas, ar kodai yra klonai, ar ne. Ekvivalenčių mutavusių kodų atpažinimo uždavinys yra panašus į kodų klonų aptikimo uždavinį, nes įvestis abiejuose uždaviniuose yra du programiniai kodai, pateikti kokia nors forma. Todėl kodų klonų uždavinio modelių struktūras galima pritaikyti ir ekvivalenčių mutavusių kodų aptikimo problemai. Skirtumas tarp šių uždavinių yra toks, kad ekvivalenčių mutavusių kodų aptikimo modeliui pateikiami du kodai, kurie skiriasi vos per vieną mutaciją.

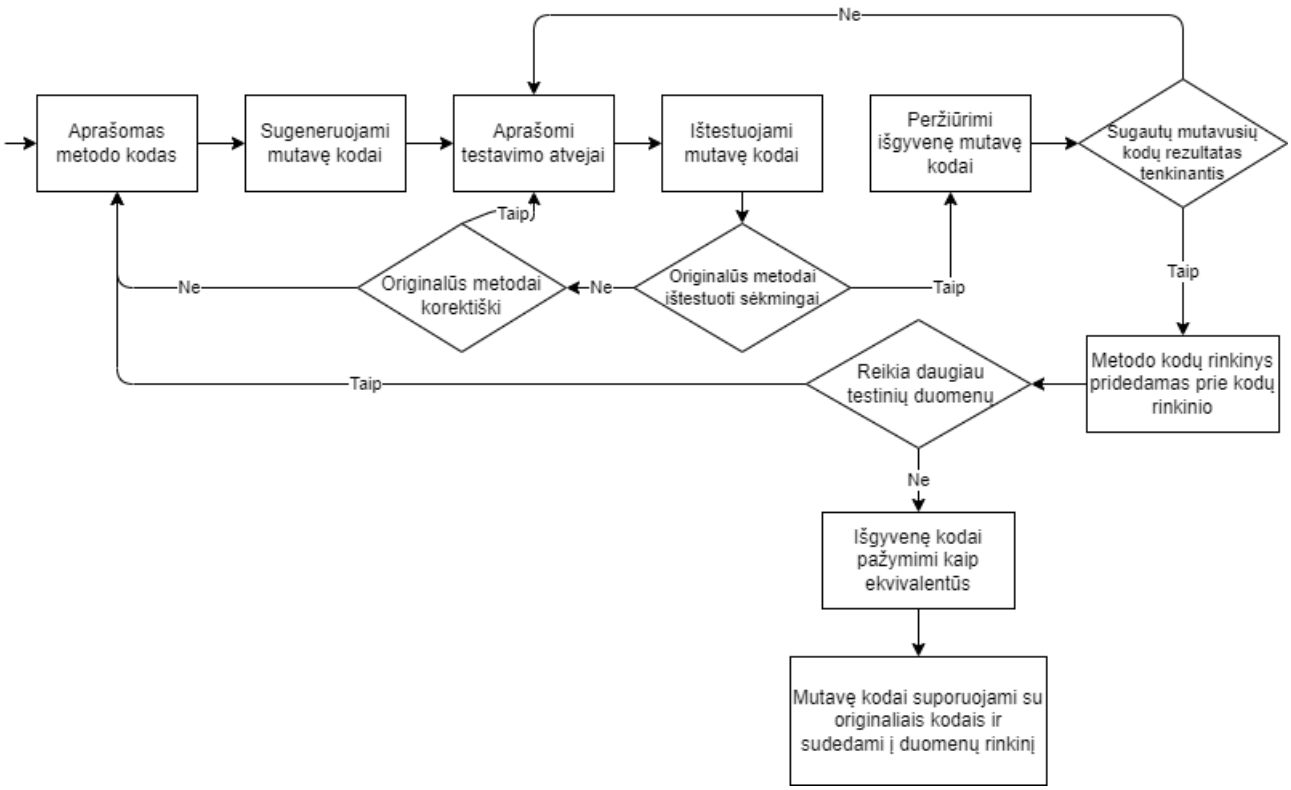
6.1. Mutavusių kodų duomenų rinkinys

Šiame poskyryje pateiktas mutavusių kodų duomenų rinkinio gamybos procesas, taikyti metodai ir rezultatai. Duomenų rinkinio gamybos tikslas buvo gauti mutavusių kodų rinkinį, kurį būtų galima taikyti ekvivalenčių mutavusių kodų aptikimo mašininio mokymosi modeliams.

6.1.1. Mutavusių kodų duomenų rinkinio gamybos procesas

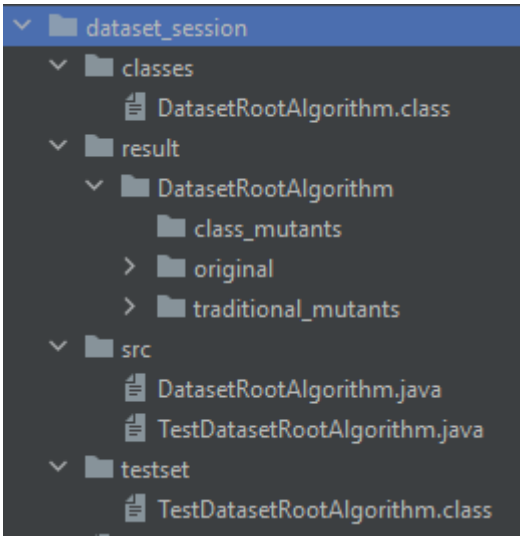
Siekiant pagaminti tikslesnius modelius, sprendžiančius ekvivalenčių mutavusių kodų problemą, buvo pagamintas naujas duomenų rinkinys. Mutavusių kodų duomenų rinkiniui gaminti naudotas *Mujava* įrankis [MOK06]. *Java* kodas buvo rašomas naudojant 1.8 versiją, kadangi *Mujava* įrankis nepalaiko naujesnės *Java* kodo sukompiliuotos versijos.

Iš pradžių taikant *Mujava* metodą *makeMuJavaStructure* duomenų rinkinio gamybos aplinkoje *dataset session* aplanke sugeneruota *Mujava* įrankio darbo aplinkos struktūra, kuri susideda iš *classes*, *result*, *src* ir *testset* aplankų. Toliau aplanke *classes* sukurta klasė *DatasetRootAlgorithm*, kurioje buvo aprašomi testuojami metodai, ir klasė *TestDatasetRootAlgorithm*, kurioje buvo aprašomi testavimo atvejai. Tolesnis duomenų rinkinio gamybos procesas parodytas 27 pav.



27 pav. Duomenų rinkinio gamybos procesas

Aprašius naujus testuojamus algoritmus *DatasetRootAlgorithm.java* failas būdavo sukompiliuojamas ir baitų kodų failas perkeliamas į aplanką *classes*. Toliau taikant *Mujava* įrankio metodą *GenMutantsMain* aplanke *results* pateikiami sugeneruoti rezultatai: originalaus kodo baitų kodas *bytecode* ir dekompiliuotas originalus kodas aplanke *original* ir sugeneruoti mutavę kodai su metodų mutacijomis, pateikti baitų kodų formatu ir dekompiliuotu kodu, pateikti aplanke *traditional mutants*. Baitų kodų formatu programos išsaugomos tam, kad vėliau testavimo proceso metu jie galėtų būti ištestuojami ant aprašytų testavimo atvejų. Visa darbo aplinka pateikta 28 pav.



28 pav. *Mujava* įrankio struktūra

Toliau klasėje *TestDatasetRootAlgorithm* testuojamiems metodams aprašomi testavimo at-

vejai. Testavimo atvejus buvo stengiamasi parinkti tokius, kad jie išgaudytų visus neekvivalenčius mutavusius kodus. Tada testavimo kodas būdavo sukompiliuojamas ir baitų failas perkeliamas į aplanką *testset*. Tada naudojantis *Mujava* funkcija *RunTestMain* buvo testuojami originalūs ir mutavę kodai ir išvedami rezultatai į funkcijos kvietimo metu komandinėje eilutėje nurodytą failą. Jeigu testavimo atvejai neatitikdavo originalių metodų kodų išvesties, būdavo tikrinama, ar metodai ir jų testavimo atvejai realizuoti korektiškai. Jeigu buvo randama klaida testavimo atvejų aprašyme, būdavo grįžtama taisyti testavimo atvejų, kitu atveju, jeigu buvo randama klaida originaliame metode, būdavo grįžtama taisyti originalių metodų, kad jie išvestų korektišką rezultatą.

Jeigu originalūs metodai būdavo ištestuoti sėkmingai, buvo nagrinėjami mutavusių kodų testavimo rezultatai. Jeigu būdavo randamas išgyvenęs mutavęs kodas, kuris, autoriaus manymu, turėtų būti sunaikintas, būdavo grįžtama į testavimo atvejų aprašymo žingsnį, siekiant parinkti tokį testavimo atvejį, kuris sunaikintų šį mutavusį kodą. Kitu atveju, jei testavimo rezultatai būdavo tenkinantys, metodo kodų rinkinys buvo laikomas pridėtu prie kodų rinkinio ir einama prie kitų metodų mutavusių kodų rinkinio generavimo.

Kai buvo laikoma, kad sugeneruotas pakankamas duomenų rinkinio dydis, visiems originaliems kodams ir jų mutavusiems kodams buvo dar kartą įvykdytas testavimo procesas, išgyvenę kodai pažymėti kaip ekvivalentūs, sunaikinti kodai pažymėti kaip neekvivalentūs, mutavę kodai suporuoti su originaliais kodais ir sudėti į duomenų rinkinį. Gautas duomenų rinkinys, susidedantis iš originalių ir mutavusių kodų, sudėtų į CSV failą *mut funcs all* ir sužymėtų kodų porų bei jų ekvivalentumo būsenų, sudėtų į CSV failą *mut pair ids*.

6.1.2. Mutavusio kodo gaudymo pavyzdys

Toliau pateiktas paieškos į plotį algoritmo mutavusio kodo, gauto pritaikius aritmetinio operatoriaus pridėjimo AOIS operatorių. Kaip įvestis yra pateikiamas orientuotas svorinis grafas *rGraph*, pradinė viršūnė *s*, galutinė viršūnė *t*, šiuo metu žinomas viršūnių tėvų sąrašas *parent*. Metodo išvestimi laikomas *boolean* požymis, ar grafe *rGraph* egzistuoja kelias nuo viršūnės *s* iki viršūnės *t* ir atnaujintas viršūnių tėvų sąrašas *parent*, kur pažymėta, iš kokios viršūnės yra pasiektos konkrečios viršūnės. Čia grafo viršūnių indeksai skaičiuojami nuo 0.

Algoritmas 3 Paieškos į plotį algoritmas su AOIS operatoriaus mutacija 16 eilutėje

```
1  public  boolean bfs1( int[][] rGraph, int s, int t, int[] parent )
2      {
3          int V = rGraph.length;
4          boolean[] visited = new boolean[V];
5          for (int i = 0; i < V; ++i) {
6              visited[i] = false;
7          }
8          int[] queue = new int[V];
9          queue[0] = s;
10         int queueLength = 1;
11         visited[s] = true;
12         parent[s] = -1;
13         while (queueLength != 0) {
14             int u = queue[0];
15             for (int i = 1; i < queueLength; i++) {
16                 queue[i - 1] = queue[i++]; // mutavusi eilutė
17             }
18             queueLength--;
19             for (int v = 0; v < V; v++) {
20                 if (!visited[v] && rGraph[u][v] > 0) {
21                     if (v == t) {
22                         parent[v] = u;
23                         return true;
24                     }
25                     queue[queueLength] = v;
26                     queueLength++;
27                     parent[v] = u;
28                     visited[v] = true;
29                 }
30             }
31         }
32         return false;
33     }
```

3 algoritme pateiktas mutavęs paieškos į plotį kodas, kur 16 eilutėje vietoj įprastinio masyvo elemento perskyrimo $queue[i-1] = queue[i]$ yra įvykdoma eilutė $queue[i-1] = queue[i++]$, kuriai yra pritaikytas AOIS operatorius, dėl to eilutėje yra pridėtas padidinimo vienetu operatorius. Todėl vykdant šią eilutę yra įvykdomas ir kintamojo i padidinimas vienetu, dėl to elemento trynimo metu yra perskiriamas tik kas antras elementas. Tai implikuoja, kad algoritmas neper-

rinks tvarkingai eilėje laukiančių viršūnių, kol jos galėtų būti perrinktos. Tačiau pradinis testavimo atvejų rinkinys nesunaikino šio mutavusio kodo, todėl reikia surasti naują testavimo atvejį, kuris sunaikintų šį kodą.

4 algoritme pateiktas testavimo atvejis, kuris sunaikina 1 algoritme mutavusį kodą.

Algoritmas 4 Testavimo atvejis, sunaikinantis 3 algoritme pateiktą mutavusį kodą

```
int[][] graph = new int[][] {
    {0, 16, 0, 0, 0, 0, 0, 0 },
    {0, 0, 10, 12, 0, 0, 0, 6 },
    {13, 4, 0, 0, 1, 0, 0, 0 },
    {0, 0, 9, 0, 0, 0, 0, 1 },
    {0, 0, 13, 7, 0, 4, 2, 0 },
    {0, 0, 0, 0, 0, 0, 0, 0 },
    {0, 0, 0, 0, 0, 0, 0, 3 },
    {0, 0, 0, 0, 0, 0, 4, 0 },
};
int[] parent = {0, 0, 0, 0, 0, 0, 0, 0};
assertEquals (true, alg.bfs1(graph, 0, 5, parent));
assertArrayEquals (new int[] {-1, 0, 1, 1, 2, 4, 7, 1}, parent);
```

Šiam pateiktam testavimo atvejui paieškos į plotį algoritmas grąžina rezultatą *false* (nes nebuvo rastas kelias nuo nulinės viršūnės iki penktos) ir kaip viršūnių tėvų sąrašą grąžino rinkinį $\{-1, 0, 1, 1, 2, 0, 0, 1\}$, nors buvo tikimasi, kad bus rastas takas ir kad viršūnių tėvų sąrašas bus lygus $\{-1, 0, 1, 1, 2, 4, 7, 1\}$. Vykdant mutavusį kodą su tokia įvestimi septinta viršūnė, atsidūrusi eilėje, prašokama, dėl to neatrandama tako iki penktos viršūnės per septintą viršūnę. Tokiu būdu po papildomos peržiūros mutavęs kodas buvo sunaikintas. Taikant šį procesą, buvo naikinami ir kiti mutavę kodai, kurie pasirodė nesantys ekvivalentūs.

6.1.3. Duomenų rinkinio informacija

Duomenų rinkinys buvo sudaromas *Java* programavimo kalba, kuri atitiko 1.8 *Java* versiją. Duomenų rinkinio gamybai buvo realizuoti 23 unikalūs metodai. Įskaitant įvairias šių metodų variacijas iš viso realizuoti 77 metodai. Aprašant metodus, buvo pasirinktas duomenų tipų poaibis iš duomenų tipų *int*, *long*, *double*, *boolean*, *char*, *String*, laikant, kad jie padengia pagrindinį funkcionalumą. Todėl likę *Java* duomenų tipai *byte*, *short*, *float*, taip pat visos aptraukiančios (angl. *wrapper*) klasės duomenų tipai, kaip *Integer* ir kt., nebuvo naudoti. Taip pat kode nebuvo naudojamos klasės iš *java.util* bibliotekos, siekiant sumažinti aprašomo kodo sudėtingumą. Norint, kad išmokyti modeliai galėtų sėkmingai veikti ant kodo, kuris turi šiuos tipus, reikėtų plėsti duomenų rinkinį. Visi realizuoti metodai yra išvardyti 6 lentelėje.

Aprašytiems metodams buvo pritaikyta 19 mutavimo operatorių, aprašytų 5 ir sugeneruota 10 027 mutavusių kodų. Mutavusių kodų testavimo metu *Mujava* įrankis ne visiems mutavusiems

kodams sugebėjo pateikti atsakymą, ar kodas ekvivalentus, ar ne, dėl kodo vykdymo metu įvykusios klaidos (pvz., masyvo kūrimas su neigiamu ilgiu) ar kitų su *Mujava* įrankiu susijusių nežinomų priežasčių. Dėl testavimo išvesties neaiškumo tokie kodai buvo praleidžiami. Todėl iš visų 10 027 mutavusių kodų sėkmingai buvo ištestuoti 9692, kurie nugulė į galutinį duomenų rinkinį.

Siekiant išvengti begalinio ciklo problemos mutavusiuose koduose kiekvieno mutavusio kodo veikimui buvo nustatytas 3 sekundžių limitas. Mutavę kodai, vykdymo metu viršiję 3 sekundžių limitą, buvo pažymėti kaip neekvivalentūs, nes realizuoti algoritmai ir paruošti testavimo atvejai nebuvo sudėtingi ir toks ilgas programos veikimas indikuoja, kad programoje atsirado begalinis ciklas.

Iš 9692 mutavusių kodų 1011 pažymėti kaip ekvivalentūs, o 8681 pažymėti kaip neekvivalentūs. Ekvivalenčių mutavusių kodų kiekis sudaro 10,43 % viso duomenų rinkinio. Tačiau nors ir ekvivalentūs mutavę kodai buvo peržiūrimi rankiniu būdu, siekiant įvertinti, ar mutavęs kodas yra ekvivalentus, yra tikimybė, kad neekvivalentus mutavęs kodas yra nepastebėtas, dažniausiai tais atvejais, kai sunku įvertinti, ar kodo mutacija galėjo turėti įtakos išvedamam rezultatui, ar ne. Todėl, autoriaus manymu, suformuotame duomenų rinkinyje egzistuoja maža dalis neekvivalenčių mutavusių kodų, kurie yra pažymėti kaip ekvivalentūs. Tačiau nėra geresnio būdo sužymėti mutavusių kodų ekvivalentumą, nes visi automatiniai įrankiai pateikia rezultatus su tam tikra paklaida, o rankinis procesas užima daug laiko ir reikalauja kruopštaus darbo. Šį procesą būtų galima optimizuoti nebent kombinuojant rankinį ir automatinį mutavusių kodų žymėjimo procesą, skiriant daugiau dėmesio mutavusiems kodams, kuriuos automatinis įrankis pažymėjo kaip neekvivalenčius.

Galutinį duomenų rinkinį sudaro 9769 kodai, kurį sudaro originalios programos ir mutavę kodai ir jų poros, kurių iš viso yra 9692, čia mutavę kodai suporuoti su jų originalia programa. Duomenų rinkinio dydis savo dydžiu daugiau nei dvigubai lenkia *MutantBench* mutavusių kodų rinkinį, kurį sudaro 4400 mutavusių kodų [HO21]. Todėl tai šiuo metu yra didžiausias įvairiems mutavimo operatoriams padengtas mutavusių kodų rinkinys.

Kiekvieną programą duomenų rinkinyje sudaro metodo aprašas ir jo realizacija. Kodai čia laikomi ekvivalentūs tuo atveju, jeigu su kiekviena įvestimi kodams yra gaunamas tas pats rezultatas. O *MutantBench* duomenų rinkinį sudaro kodai, apibrėžti klasės lygmeniu, kur kodą sudaro klasė su savo laukais ir metodais, o tokios struktūros kodų pora laikoma ekvivalenti, jeigu visų viešų metodų su visomis įvestimis išvestys sutampa. Tačiau literatūroje nėra jokio viešai prieinamo duomenų rinkinio, kurio kodus sudaro metodai. Kadangi buvo nagrinėjamas kodų ekvivalentumas metodų lygiu, buvo poreikis sukurti savo duomenų rinkinį, sudarytą iš metodų.

6.1.4. Mutavimo operatoriai

Mutavusiems kodams generuoti buvo naudojami visi mutavimo operatoriai, aprašyti [MOK16]. Mutavimo operatoriai ir jų operatorių naudojamumas aprašyti 5 lentelėje.

5 lentelė. *Mujava* įrankio mutavimo operatoriai [MOK16] ir jų naudojamumas

Mutavimo operatorius	Aprašymas	Taikyti operatoriai	Netaikyti operatoriai	Mutavusių kodų kiekis	Sėkmingai ištestuotų mutavusių kodų kiekis
AORB	Dvinario aritmetinio operatoriaus pakeitimas	+, -, *, /, %	Nė vienas	736	732
AORS	Vienetinio aritmetinio operatoriaus pakeitimas	op++, ++op, op–, –op	Nė vienas	150	149
AOIU	Dvinario aritmetinio operatoriaus pridėjimas	-	Nė vienas	856	850
AOIS	Vienetinio aritmetinio operatoriaus pridėjimas	op++, ++op, op–, –op	Nė vienas	3440	3398
AODU	Dvinario aritmetinio operatoriaus pašalinimas	+, -, *, /, %	Nė vienas	19	17
AODS	Vienetinio aritmetinio operatoriaus pašalinimas	Nė vienas	op++, ++op, op–, –op	0	0
ROR	Santykinio (angl. *relational*) operatoriaus pakeitimas	&&, ||, ^, *true*, *false*	|, &	1466	1404
COR	Dvinario sąlyginio (angl. *conditional*) operatoriaus pakeitimas	&&, ||, ^,	|, &	84	82
COI	Vienetinio sąlyginio (angl. *conditional*) operatoriaus pridėjimas	!	Nė vienas	291	289
COD	Vienetinio sąlyginio (angl. *conditional*) operatoriaus pašalinimas	!	Nė vienas	12	12
SOR	Pastūmimo (angl. *shift*) operatoriaus pašalinimas	Nė vienas	», «, »»	0	0
LOR	Dvinario loginio operatoriaus pakeitimas	Nė vienas	|, &, ^	0	0
LOI	Vienetinio loginio operatoriaus pridėjimas	~	Nė vienas	1154	1119
LOD	Vienetinio loginio operatoriaus pašalinimas	Nė vienas	~	0	0
ASRS	Priskyrimo operatoriaus pakeitimas	+=, -=, *=, /=, %=	&=, |=, ^=, «=, »=, »>=	56	53
SDL	Sakinio pašalinimas	Taikyta		766	611
VDL	Kintamojo pašalinimas	Taikyta		367	356
CDL	Konstantos pašalinimas	Taikyta		129	129
ODL	Operatoriaus pašalinimas	Taikyta		501	491

5 lentelėje yra išvardyti visi *Mujava* įrankio mutavimo operatoriai. Taikytų operatorių stulpelyje išvardyti sintaksiniai operatoriai, kurie papuolė į galutinį duomenų rinkinį, prie netaikytų operatorių išvardyti operatoriai, kurie nepapuolė į galutinį duomenų rinkinį. Originaliuose koduose nebuvo naudojami binariniai operatoriai, todėl galutiniame rinkinyje atsidūrė tik tokie binariniai operatoriai, kurie galėjo būti gaunami mutavus pradinį kodą (*xor* operatorius ^ ROR ir COR mutavimo operatoriuose ir invertavimo operatorius ~ mutavimo operatoriuje LOI). Taip pat į galutinį rinkinį nepapuolė AODS mutavimo operatoriaus operatoriai, nepaisant to, kad inkrementavimo ir

dekrementavimo operatoriai buvo plačiai naudojami pradiniame kode. Siekiant gauti išsamesnį duomenų rinkinį, jį galima papildyti kodais, kurie naudoja šiuos mutavimo operatorius.

Taip pat buvo plačiai naudojami ir šalinimo operatoriai SDL, VDL, CDL ir ODL. SDL mutavimo operatorius pašalina vykdomą sakinį iš pradinio kodo, taip pat gali pašalinti ir sąlygos *if* ar ciklų *for* ir *while* struktūras. VDL mutavimo operatorius ištrina visas pasirinkto kintamojo nuorodas, taip pat ištrinant ir operatorius, kad būtų išlaikytas kompiliuojamas kodas. CDL mutavimo operatorius ištrina pasirinktą konstantą, taip pat ištrinant ir operatorių, kad būtų išlaikytas kompiliuojamas kodas. ODL mutavimo operatorius ištrina pasirinktą operatorių, kuris gali būti aritmetinis, sąlyginis, loginis, binarinis, postūminis ar priskyrimo tipo. Šalinant dvinarį operatorių, yra paliekamas tik vienas iš šalinamo operatoriaus operandų ir yra sugeneruojami du mutantai: kur paliekamas kairysis operandas ir kur paliekamas dešinysis operandas.

Pagal 5 lentelę iš viso sugeneruotų mutavusių kodų rinkinį sudaro 10 027 mutavę kodai. Didžiausią jų dalį (3440) sudaro mutavę kodai, gauti pritaikius AOIS mutavimo operatorių, kur mutavimo kodas yra gaunamas prie kintamojo pridėjus inkrementavimo arba dekrementavimo operatorių priešdėliniu (angl. *prefix*) arba galūniniu (angl. *postfix*) būdu. Taip pat nemažą dalį sudaro mutavę kodai, gauti pritaikius santykinio operatoriaus pakeitimo mutavimo operatorių ROR (1466) ar vienetinio loginio operatoriaus pridėjimo mutavimo operatorių LOI (1154). Sudėtingesnės struktūros aprašytame programiniame kode pasitaiko rečiau, todėl ir mutavimo operatoriai, tokie kaip sakinio ar kito elemento pašalinimas, pasitaiko rečiau.

6.1.5. Įgyvendinti metodai

Duomenų rinkiniui paruošti buvo realizuota grupė klasikinių problemų algoritmų. Realizuoti metodai aprašyti 6 lentelėje.

Mutavusių kodų duomenų rinkinio gamybai buvo įgyvendinti 23 skirtingi metodai su įvairiomis jų variacijomis. Pasirinkti tradiciniai uždaviniai, tokie kaip tiesinė paieška, dvejetainė paieška ir kt. Įgyvendinti keli metodai darbui su tekstinėmis eilutėmis, tokie kaip *charCount* ir kt. Įgyvendinti penki rikiavimo algoritmai ir du pagalbiniai metodai, atliekantys tarpinius skaičiavimus rikiavimo metodams. Taip pat įgyvendinti du grafų algoritmai su dviem pagalbiniais metodais, atliekančiais tarpinius skaičiavimus. Iš viso rinkinyje yra 4 metodai, kurie kreipiasi į kitą šio algoritmų rinkinio metodą, atliekantį tarpinius skaičiavimus: algoritmas *mergeSort* kviečia metodą *merge*, algoritmas *quickSort* kviečia metodą *partition*, algoritmas *dijkstra* kviečia metodą *minDistance* ir algoritmas *fordFulkerson* kviečia algoritmą *bfs*. Čia kiekviena šių metodų variacija kreipiasi į tokį pagalbinį metodą, kuris atitinka jų duomenų tipus. Taip pat rinkinyje yra rekursyvių metodų: *binarySearch*, *mergeSort* ir *quickSort*.

Skirtingos metodų variacijos pažymi skirtingą metodų įvestį, kai metodai atlieka tą patį funkcionalumą. Pavyzdžiui, tiesinės paieškos metodas *linearSearch* pažymi, kad elementas gali būti ieškomas bet kokių tipų elementų sąraše (*int*, *long*, ...) ir yra realizuota kiekviena šio metodo versija. Skirtingos rikiavimo algoritmų versijos gali išrikiuoti skirtingų tipų elementus. O skirtingos algoritmų versijos, dirbančios su grafais, pažymi skirtingus grafų tipus: *int*, *long* ir *double* duomenų tipai pažymi svorinį grafą, o *boolean* duomenų tipas žymi, kad grafas yra nesvorinis.

6 lentelė. Duomenų rinkiniui paruošti realizuoti metodai

Uždavinys	Aprašymas	Realizuotų versijų skaičius	Realizuotos versijos	Mutavusių kodų kiekis
linearSearch	Elemento paieška neišrikiuotame sąraše	6	*int*, *long*, *double*, *char*, *String*, *boolean*	271
binarySearch	Dvejetainė elemento paieška išrikiuotame sąraše	6	*int*, *long*, *double*, *char*, *String*, *boolean*	1176
max	Didžiausio elemento paieška neišrikiuotame sąraše	3	*int*, *long*, *double*	154
isEven	Patikrinimas, ar elementas lyginis	2	*int*, *long*	48
power	$f(x, y) = x^y, y \epsilon int$	2	*long*, *double*	165
prime	Patikrinimas, ar skaičius pirminis	2	*int*, *long*	135
solveLinearEq	Tiesinės lygčių sistemos sprendimas	1	*double*	448
charCount	Simbolio pasikartojimų tekstinėje eilutėje kiekis	1	*char*	46
wordCount	Žodžio pasikartojimų tekstinėje eilutėje kiekis	1	*String*	32
allWordCount	Žodžių tekstinėje eilutėje kiekis	1	*String*	3
palindrome	Patikrinimas, ar tekstinė eilutė yra palindromas	1	*String*	58
firstNonRepeatingChar	Pirmo nepasikartojančio simbolio paieška tekstinėje eilutėje	1	*String*	57
bubbleSort	Burbulo rikiavimo algoritmas	5	*int*, *long*, *double*, *char*, *String*	422
insertionSort	Įterpimo rikiavimo algoritmas	5	*int*, *long*, *double*, *char*, *String*	606
selectionSort	Parinkimo rikiavimo algoritmas	5	*int*, *long*, *double*, *char*, *String*	581
merge	Dviejų sąrašų sujungimas elementų didėjimo tvarka	5	*int*, *long*, *double*, *char*, *String*	1485
mergeSort	Sujungimo rikiavimo algoritmas	5	*int*, *long*, *double*, *char*, *String*	467
partition	Sąrašo elementų grupavimas į mažesnius už ašį ir didesnius	5	*int*, *long*, *double*, *char*, *String*	680
quickSort	Greitasis rikiavimo algoritmas	5	*int*, *long*, *double*, *char*, *String*	335
minDistance	Trumpiausio atstumo tarp neapdorotų viršūnių paieška	3	*int*, *long*, *double*	214
dijkstra	Dijkstros trumpiausio kelio nuo konkrečios viršūnės iki kiekvienos viršūnės paieška	4	*int*, *long*, *double*, *boolean*	791
bfs	Paieškos į plotį algoritmas	3	*int*, *long*, *double*, *boolean*	694
fordFulkerson	Didžiausio srauto paieška	4	*int*, *long*, *double*, *boolean*	824

Pagal 6 iš aprašytų metodų daugiausia mutavusių kodų sėkmingai ištestuota metodui *merge* (1485), todėl kad šiam metodui yra sukurtos 5 skirtingos variacijos ir visos jos užima 43 eilutes. Toliau daug mutavusių kodų sėkmingai ištestuota metodui *binarySearch* (1176), nes jam realizuotos 6 skirtingos variacijos ir jų realizuotas kodas užima 13 eilučių su daug skirtingų mutacijų galimybių. Taip pat nemažai mutavusių kodų ištestuota ir rikiavimo bei grafų apdorojimo metodams. Mažiausiai mutavusių kodų sugeneruota ir ištestuota metodams, dirbantiems su tekstinėmis eilutėmis, nes jų yra vos viena realizuota versija su palyginti nedaug logikos.

6.2. Modelių mokymų informacija

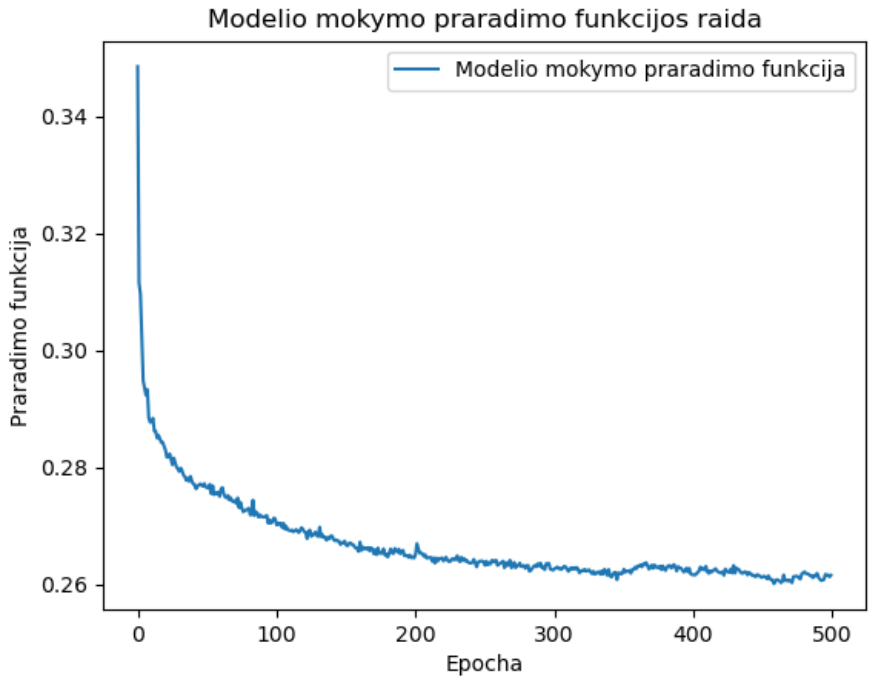
Visų kodų testavimo rezultatų generavimo procesas truko apie 4 valandas. Šį procesą galima optimizuoti suskaidžius tarpusavyje nesusijusius metodus į atskiras klases.

Mokyti metrikų modelį taikant mutavusių kodų rinkinį 300 epochų prireikė 13 minučių. Mokyti abstrakčios sintaksės medžio modelį taikant mutavusių kodų rinkinį prireikė 3 h 33 min.

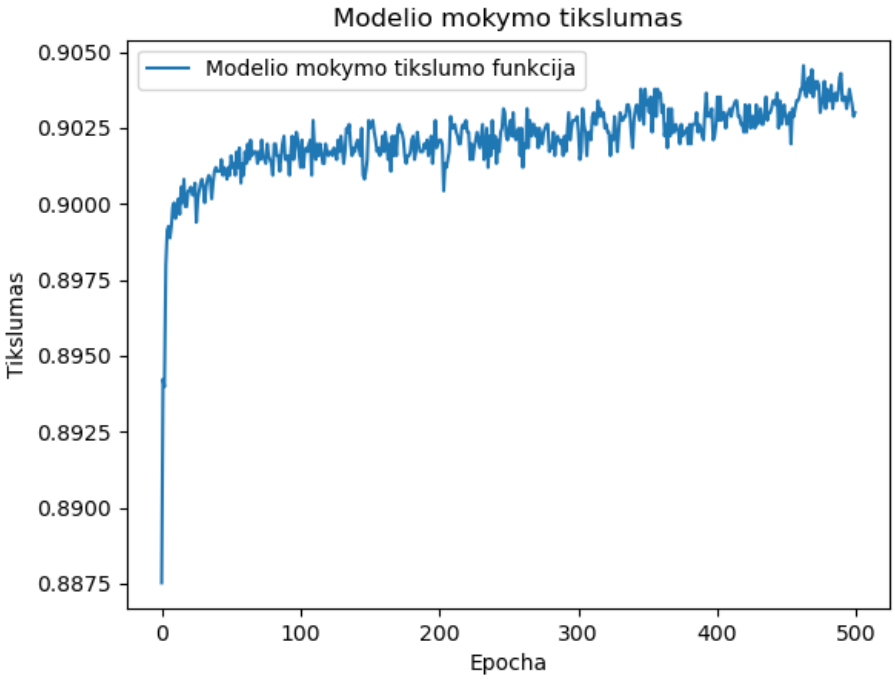
6.3. Metrikų modelio, taikomų *Java* ekvivalenčių mutavusių kodų duomenų rinkiniui, testavimo rezultatai

Ekvivalenčių mutavusių kodų aptikimo metrikų modeliui suformuotas 58 *Java* kodo metrikų sąrašas, kuris yra pateiktas 10 ir 11 lentelėse. Kodo metrikos parinktos tokios, kad jos atskleistų informaciją apie kodus bendrai, arba kad atskleistų pasikeitimą pritaikius mutavusį operatorių tarp originalaus kodo ir mutavusio kodo.

Modelio mokymo rezultatai pateikti 29 ir 30 pav.



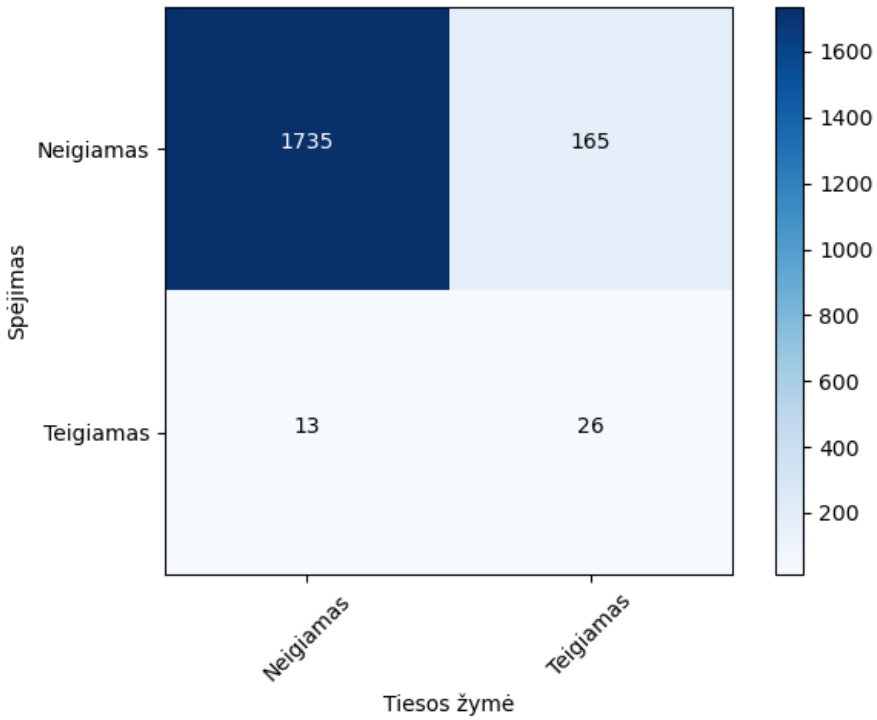
29 pav. *Java* mutavusių kodų metrikų modelio praradimo funkcija *BCELoss*



30 pav. *Java* mutavusių kodų metrikų modelio tikslumo funkcija

Pagal 29 ir 30 pav., kai modelis mokytas naudojant 58 *Java* mutavusių kodų metrikų sąrašą, praradimo funkcijos įvertis po 500 epochų nukrito iki maždaug 0,26, o tikslumas svyruoja apie 0,903.

Modelio klaidų matrica pateikta 31 pav.



31 pav. *Java* mutavusių kodų metrikų modelio klaidų matrica

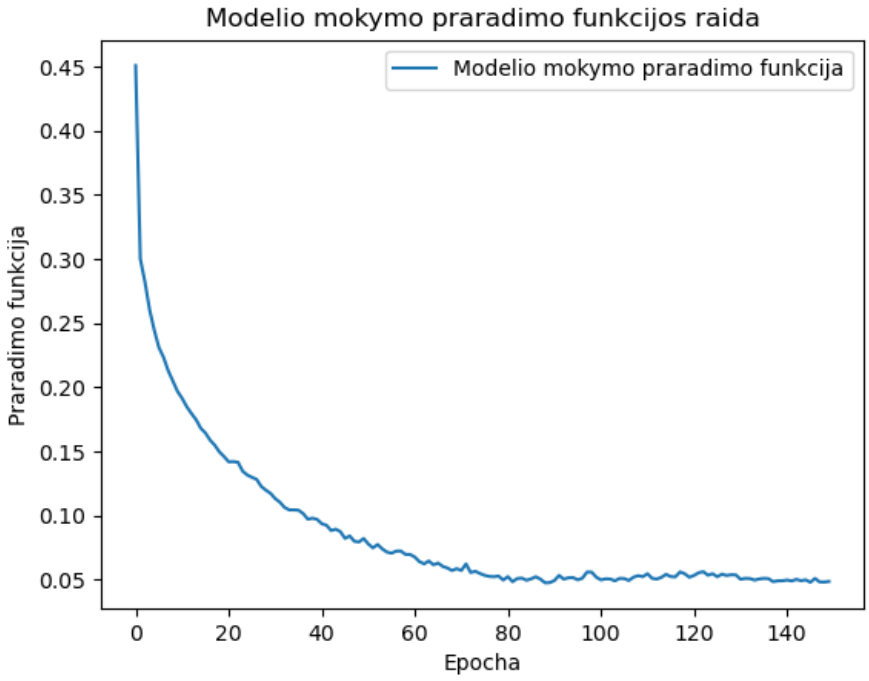
Šis modelis pasiekė 90,82 % bendrą tikslumą. Identifikuotų elementų tikslumas pasiekė 66,7

%, nes iš 39 teigiamai identifikuotų porų 26 buvo ekvivalenčios, o identifikuotų elementų išsamumas siekia 13,61 %, nes iš 191 ekvivalenčių mutavusių kodų testavimo duomenų rinkinyje 26 buvo pažymėti teigiamai. Šio modelio f1 įvertis lygus 0,226.

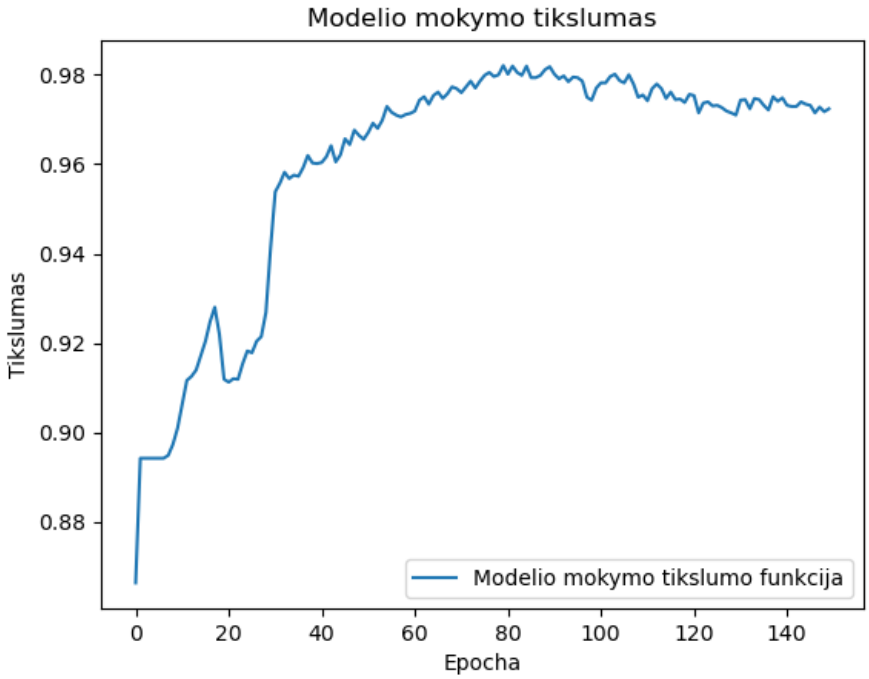
Esant modelio ribai $\delta 1 = 0{,}5$ modelis aptinka nedaug ekvivalenčių mutavusių kodų (13,61 %). Norint aptikti didesnį skaičių ekvivalenčių mutavusių kodų galima ribą mažinti $\delta 1$ ribą, tačiau tada yra aukojamas modelio tikslumas. Modelį galima optimizuoti pritaikius *Lime* arba *Shap* technologijas, siekiant nustatyti ir atrinkti metrikas, kurios labiausiai padėjo identifikuoti ekvivalenčius mutavusius kodus. Taip pat modelį galima optimizuoti jam pritaikius hiperparametrų derinimą (angl. *hyperparameter tuning*). Taip pat siekiant gauti tikslesnių rezultatų reikėtų plėsti duomenų rinkinį, kad būtų sukurta kuo didesnė kodų įvairovė.

6.4. Medžio struktūros modelio, taikomo *Java* ekvivalenčių mutavusių kodų duomenų rinkiniui, testavimo rezultatai

Medžio struktūros modelio mokymo rezultatai pateikti 32 ir 33 pav. Modelis mokytas taikant [ZWZ+19] autorių pateiktą eksperimento kodą su 50 epochų.

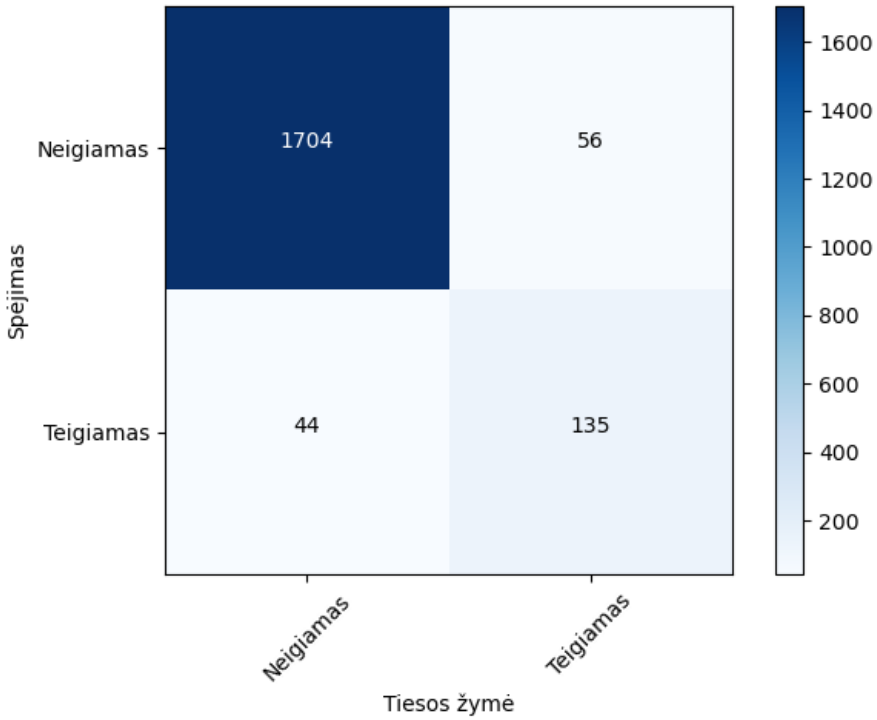


32 pav. Medžio struktūros modelio praradimo funkcija *BCELoss*



33 pav. Medžio struktūros modelio tikslumo funkcija

Pagal 32 ir 33 pav. modelio mokymas rodo stabilius rezultatus, o praradimo ir tikslumo funkcijų kreivės svyruoja minimaliai. Modelio mokymo praradimo funkcijos įvertis nusistovi ties 0,05, o tikslumas pasiekia 0,97–0,98 ribą.



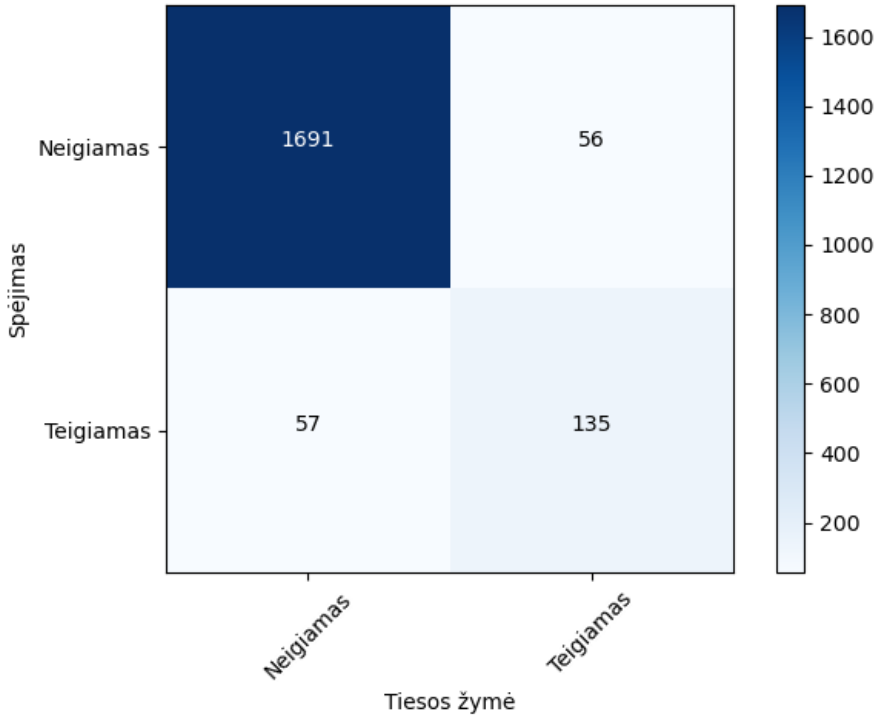
34 pav. Medžio struktūros modelio klaidų matrica

34 pav. pateikti *Java* mutavusių kodų abstraktaus sintaksės medžio struktūros modelio testavimo rezultatai klaidų matricos forma. Bendras modelio tikslumas yra 94,84 %, nes iš 1939

testavimo rinkinio atvejų 1839 buvo identifikuoti teisingai. Identifikuotų elementų tikslumas pasiekė 75,4 %, nes iš 179 teigiamai identifikuotų porų 135 buvo ekvivalenčios. O identifikuotų elementų išsamumas siekia 70,7 %, nes iš 191 mutavusių kodų testavimo duomenų rinkinyje 135 buvo pažymėti teigiamai. Šio modelio f1 įvertis lygus 0,730.

6.5. Medžio struktūros ir metrikų modelių, taikomo *Java* mutavusių kodų duomenų rinkiniui, kombinuoti rezultatai

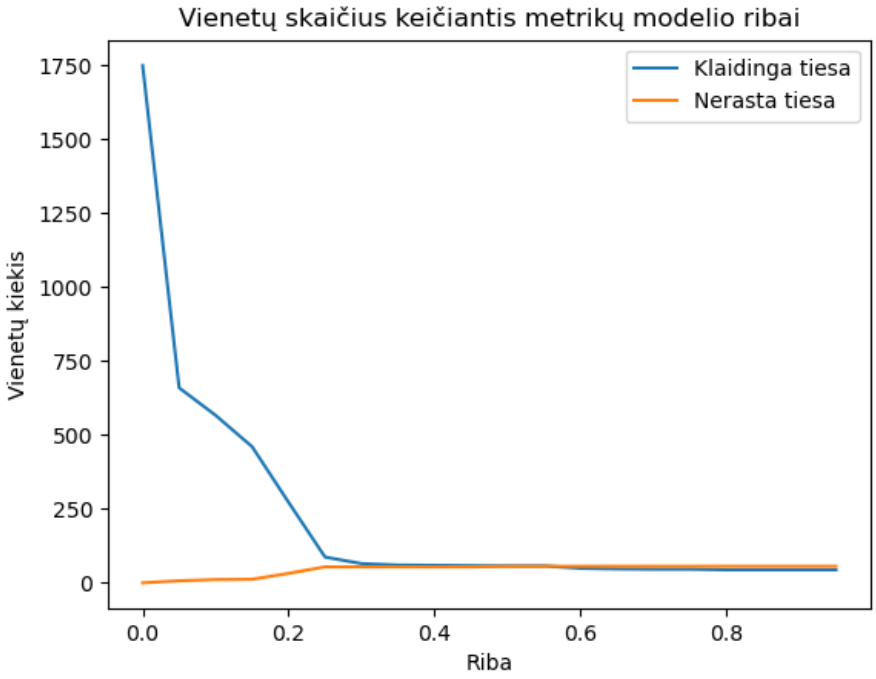
Metrikų ir medžio struktūros modeliams palyginti 35 pav. pateikta mutavusių kodų kombinuoto modelio (8 pav.) klaidų matrica. Čia kodų pora pažymima kaip ekvivalenti tuo atveju, jeigu nors vienas iš modelių taip pažymėjo tą kodų porą, identiškai kaip sprendžiant kodų klonų aptikimo uždavinį. Abiejų modelių slenksčiai $\delta 1$ ir $\delta 2$ yra lygūs 0,5.



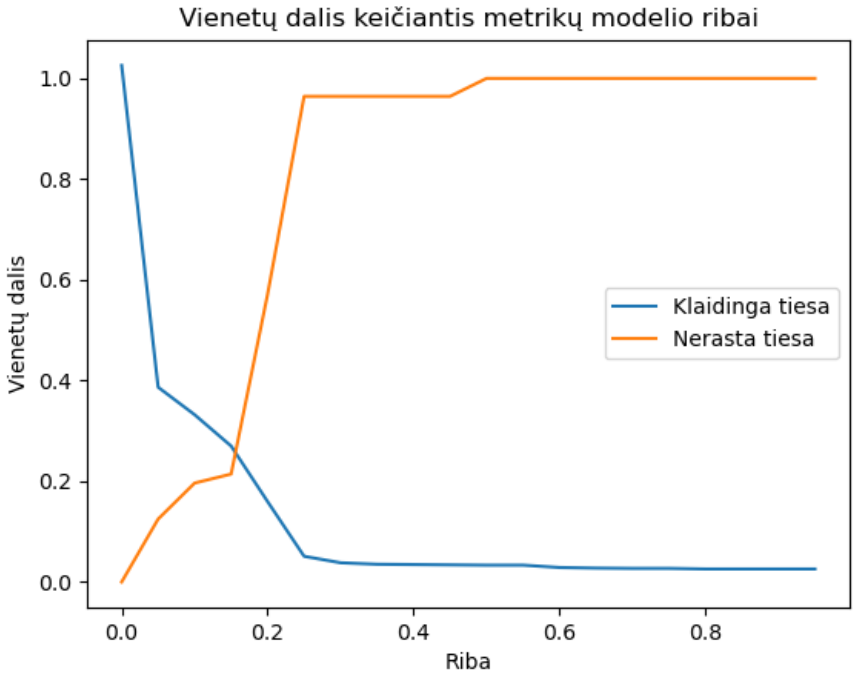
35 pav. Mutavusių kodų kombinuoto modelio klaidų funkcija

Pagal 35 pav. iš 56 medžio struktūros neidentifikuotų ekvivalenčių mutavusių kodų porų kombinuotas modelis neaptiko nė vieno ir modelio išsamumo įvertis išliko lygus 70,68 %. O identifikuotų ekvivalenčių mutavusių kodų tikslumas nukrito iki 70,31 %, nes iš 192 identifikuotų kodų klonų porų 57 buvo identifikuoti klaidingai. Todėl galutinio modelio f1 įvertis yra lygus 70,5 %.

Kadangi metrikų modelis grąžina tikimybę, ar kodų pora yra ekvivalenti, ar ne, galima palyginti rezultatus keičiantis nustatytai modelio ribai (angl. *threshold*). 36 ir 37 pav. yra pateikta, kokia kombinuoto modelio kodų porų dalis buvo identifikuota klaidingai keičiantis metrikų modelio ribai $\delta 2$.



36 pav. Klaidingai identifikuotų ekvivalenčių mutavusių kodų porų kiekis keičiantis metrikų modelio identifikavimo ribai

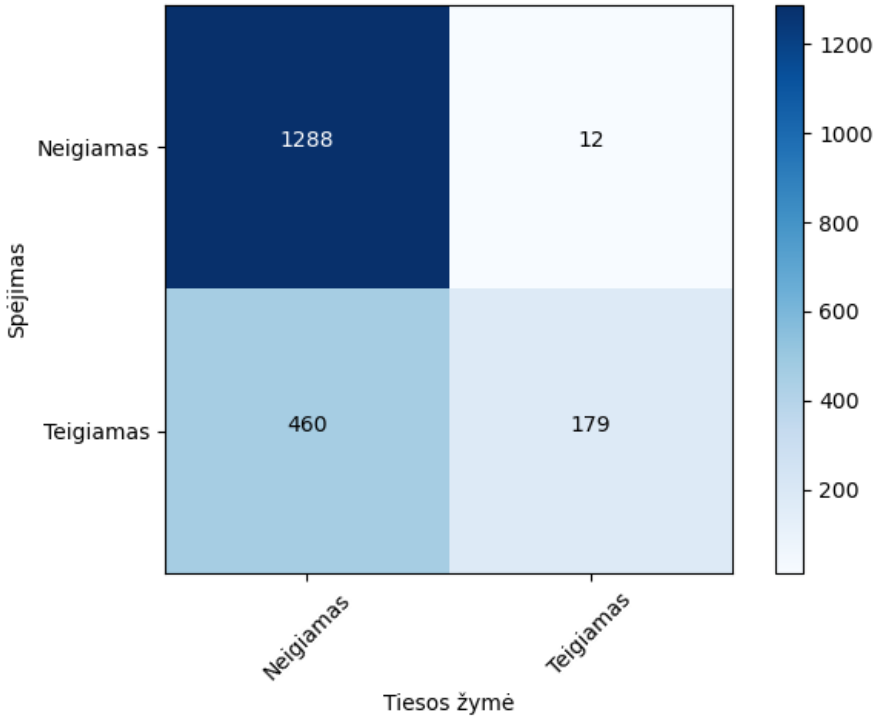


37 pav. Klaidingai identifikuotų ekvivalenčių mutavusių kodų porų dalis keičiantis metrikų modelio identifikavimo ribai

Pagal 36 pav., kai riba lygi 0, visos kodų poros identifikuojamos kaip ekvivalenčios, todėl klaidingos tiesos įvertis lygus visos aibės dydžiui, atmetus tikrų ekvivalenčių kodų poras. O kai riba lygi 1, klaidingos tiesos ir nerastos tiesos įverčiai tampa lygūs 57 ir 56, tai atitinka medžio struktūros

modelio testavimo rezultatus. Ribai esant tarp 0 ir 1, klaidingos tiesos ir nerastos tiesos įverčiai pamažu keičiasi. Pagal 37 pav. nerastos tiesos kreivė rodo, kokia yra išlikusi medžio struktūros modelio nerastų kodų klonų porų dalis. Kai riba lygi 0,05, pažymėta, kad neidentifikuota 7 iš 56 ekvivalenčių kodų porų, ties riba, lygia 0,15, pažymėta, kad neidentifikuota 12 iš 56 ekvivalenčių kodų porų, o kai riba lygi 0,5, neidentifikuotos visos 56 ekvivalenčios kodų poros. Klaidingos tiesos kreivė, kai riba lygi 0,05, klaidingai pažymi 659 poras ekvivalenčiomis, ties riba, lygia 0,15, klaidingai pažymimos 460 poros, o ties riba, lygia 0,5, šis skaičius nukrinta iki 57. Remiantis šiais kombinuoto modelio testavimo rezultatais priklausomai nuo poreikio galima pasirinkti metrikų modelio ribą ir taikyti modelį įvairiems kodų klonų aptikimo uždavinio sprendimams.

Kadangi modelių riboms $\delta 1$ ir $\delta 2$ parinkus reikšmę, lygią 0,5, kombinuotas modelis nepagerino išsamumo įverčio, šias ribas galima keisti norint pasiekti šį tikslą. 38 pav. pateikta kombinuoto modelio klaidų matrica, metrikų modelio ribai $\delta 2$ parinkus reikšmę, lygią 0,15.



38 pav. Mutavusių kodų kombinuoto modelio klaidų funkcija, kai $\delta 2 = 0{,}15$

Pagal 38 pav. iš 56 medžio struktūros neidentifikuotų ekvivalenčių mutavusių kodų porų ši kombinuoto modelio versija aptiko 44 ir pagerino išsamumo įvertį iki 93,72 %. Tačiau dėl ypač žemos pasirinktos ribos metrikų modeliui identifikuotų ekvivalenčių mutavusių kodų tikslumas nukrito iki 28,01 %, nes iš 639 identifikuotų kodų klonų porų 460 buvo identifikuoti klaidingai. Galutinio šios kombinuotos versijos modelio f1 įvertis yra lygus 75,66 %. Tačiau mutacinio testavimo metu ši kombinuota ekvivalenčių mutavusių kodų aptikimo modelio versija yra prasmingesnė naudoti norint pasiekti stipresnį mutavimo įvertį.

7. Mutacinio testavimo proceso taikymas

Šiame skyriuje aprašyta mutacinio testavimo proceso realizacija ir parodyta, kaip pritaikomas ekvivalenčių mutavusių kodų uždavinys.

7.1. Mutacinio testavimo prototipo realizacija

Mutacinio testavimo prototipas buvo realizuotas *Python* programavimo kalba. Testuojamas kodas pateiktas *Java* programavimo kalba, siekiant parodyti, kad procesą galima pritaikyti įvairioms programavimo kalboms. Siekiant paleisti *Java* kodą iš *Python* aplinkos buvo naudojama *Jpype* biblioteka.

Funkcija *applyMutationTesting* apibrėžtas mutacinio testavimo procesas. Šio proceso pseudokodas yra pateiktas 5 algoritmu:

Algoritmas 5 Mutacinio testavimo proceso pseudokodas

```
1  def apply_mutation_testing(method_name, set_initial_data):
2      if set_initial_data:
3          test_set = set_initial_data(method_name)
4      else:
5          test_set = generate_initial_test_set(method_name)
6      mutants = get_mutants(method_name)
7      mutants = mark_equivalent_mutants(method_name, mutants)
8      test_set = convert_to_bits(test_set)
9      score = eval_mutation_score(method_name, mutants, test_set)
10     for i in range(0, iter_count):
11         apply_genetic_operations(score, test_set, method_name)
12         test_set = convert_from_bits(test_set)
13         recalculate_outputs(test_set, method_name)
14         score = eval_mutation_score(method_name, mutants, test_set)
15     return score
```

Pagal 5 algoritmą, iš pradžių pagal testuojamo metodo pavadinimą yra sugeneruojamas mutavusių kodų rinkinys (6 eilutė) ir testavimo atvejų rinkinys (5 eilutė). Arba jeigu pasirenkama pradinį testavimo atvejų rinkinį suformuoti pačiam, testavimo atvejų rinkinys 3 eilutėje yra įkraunamas iš failo. Tada septintoje eilutėje yra pažymimi mutavę kodai, kuriuos kombinuotas modelis nustatė kaip ekvivalenčius. Kodai laikomi ekvivalenčiais, tačiau ir toliau testuojami siekiant surasti testavimo atvejį, kuriam jis pateikia neteisingą rezultatą. Jeigu pavyksta rasti tokį testavimo atvejį, kodas nebelaikomas neekvivalenčiu. Aštuntoje eilutėje testavimo atvejų rinkinys yra užkoduojamas bitų eile, kiekvieną kiekvieno testavimo atvejo parametrą konvertuojant į bitus ir gautas bitų eiles sujungus kiekvienam testavimo atvejui. Devintoje eilutėje yra apskaičiuojamas pradinis mutavimo įvertis (visų ir neekvivalenčių mutavusių kodų). Tada yra pradedamas ciklas ir pradedamas vykdyti genetinis algoritmas. Vienuoliktoje eilutėje atliekamos genetinės operacijos, iš kurių

gaunamas naujas testavimo atvejų rinkinys, kuriam, naudojantis *Jpype* biblioteka, perskaičiuojamos originalaus *Java* kodo išvestys su naujais testavimo atvejais ir perskaičiuojamas mutavimo įvertis. Galiausiai kaip rezultatas yra grąžinamas mutacinio testavimo įvertis, kuris pažymi, kiek mutavusių kodų sunaikino galutinis testavimo atvejų rinkinys.

Funkcija *evalMutationScore* aprašytas mutavimo įverčio radimas.

Algoritmas 6 Mutavimo įverčio radimo pseudokodas

```
1  def eval_mutation_score(method_name, mutants, test_set):
2      considered_non_eq_mutants_length =
3          len(list(filter(lambda m: not m.is_equivalent or m.is_killed, mutants)))
4      inverted_mutation_table = []
5      for i, mutant in enumerate(mutants):
6          score_line = []
7          for test_case in test_set:
8              output = run_java_method(method_name, test_case)
9              is_correct = test_case.output != output
10             score_line.append(is_correct)
11         inverted_mutation_table.append(score_line)
12     mutation_table = list(map(list, zip(*inverted_mutation_table)))
13     for i, test_case in enumerate(test_set):
14         test_case.mutant_proportion =
15         mutation_table[i].count(True) / len(mutation_table[i])
16     count_score =
17     [all(col) for col in inverted_mutation_table].count(False)
18     score = count_score / len(mutants)
19     non_eq_score = count_score / considered_non_eq_mutants_length
20     return [score, non_eq_score]
```

Pagal 6 algoritmą, siekiant apskaičiuoti testavimo rinkinio mutavimo įvertį, iš pradžių 2–3 eilutėse randamas mutavusių kodų, laikomų neekvivalenčiais, skaičius. 4–12 eilutėse apskaičiuojama mutavimo lentelė, kurioje kiekvienai mutavusio kodo ir testavimo atvejo porai sužymima, ar gauta išvestis atitinka originalaus kodo pateiktą išvestį, ar ne. Dėl egzistuojančios galimybės sugeneruoti mutavusius kodus, kurie gali turėti amžiną ciklą, vykdant *Java* kodą uždėtas 1 sekundės terminas, kiek laiko jis gali būti vykdomas. Jeigu programa negrąžina išvesties per nustatytą laiką arba jeigu programa grąžina klaidą (pvz. dalyba iš 0), pažymima, kad išvestis nesutampa su laukta išvestimi ir kodas pažymimas kaip sunaikintas. Jeigu išvestys nesutampa, laikoma, kad kodas sunaikintas, kitu atveju laikoma, kad jis išgyveno. Toliau 13–15 eilutėse kiekvienam testavimo atvejui yra apskaičiuojama, kokią dalį mutavusių kodų šis testavimo atvejis sunaikino. Galiausiai 16–20 eilutėse yra apskaičiuojama, kokią dalį visų mutavusių kodų ir kokią dalį laikomų neekvivalenčiais mutavusių kodų nesunaikino nė vienas testavimo atvejis ir tai pateikiama kaip šios funkcijos išvestis.

7.2. Genetinio algoritmo realizacija testavimo atvejų rinkiniui generuoti

Funkcija *applyGeneticOperations* aprašytas genetinio algoritmo operacijų taikymas naujam testavimo duomenų rinkiniui formuoti. Genetinio algoritmo struktūra aprašyta pagal Khan ir Amjad aprašytą procesą [KA15].

Algoritmas 7 Genetinių operatorių taikymo naujam testavimo rinkiniui formuoti pseudokodas

```
1  def apply_genetic_operations(test_set, method_name):
2      removal_threshold = 0.2
3      threshold = 0.5
4      amount_of_test_cases = len(test_set)
5
6      # selection
7      for test_case in test_set:
8          if test_case.mutant_proportion <= removal_threshold:
9              test_set.remove(test_case)
10
11     # mutation
12     for test_case in test_set:
13         if test_case.mutant_proportion <= threshold:
14             while True:
15                 new_test_case = mutate(test_case.bit_input)
16                 if test_case_exists(test_set, new_test_case):
17                     test_case.bit_input = new_test_case
18                     break
19
20     # new data generation if less than 2 test cases are alive
21     while len(test_set) < 2:
22         new_test_case = generate_bit_input(8 * len(method_params))
23                 if test_case_exists(test_set, new_test_case):
24             test_set.append(TestCase(None, 8, new_test_case, method_name))
25
26     # crossover
27     while len(test_set) < amount_of_test_cases:
28         new_test_case = crossover(test_set)
29                 if test_case_exists(test_set, new_test_case):
30             test_set.append(TestCase(None, 8, new_test_case, method_name))
```

Pagal 7 algoritmą, iš pradžių 2 eilutėje yra parenkama testavimo atvejo pašalinimo riba, kuri parenkama lygi 0,2, ir 3 eilutėje parenkama testavimo atvejo mutavimo riba, kuri parenkama lygi 0,5. Toliau 6–9 eilutėse vykdomas testavimo atvejų atrinkimo procesas, kurio metu pašalinami

testavimo atvejai, kurie sunaikino mažiau nei 20 % mutavusių kodų. 11–18 eilutėse vykdomas testavimo atvejų mutavimo procesas, kur testavimo atvejams su mutavimo proporcijos įverčiu, lygiu tarp 0,2 ir 0,5, yra invertuojamas atsitiktinis bitas testavimo atvejo bitų eilutės išraiškoje. 20–24 eilutėse yra sugeneruojami nauji testavimo atvejai, jeigu rinkinyje jų liko mažiau, nei 2. 26–30 eilutėse yra vykdomas kryžminimo procesas, kur iš išgyvenusių testavimo atvejų yra sugeneruojami nauji testavimo atvejai. Kryžminimo metu yra parenkami du egzistuojantys testavimo atvejai ir jiems parenkamos dvi atsitiktinės pozicijos $pos1$ ir $pos2$. Tada naujas testavimo atvejis yra gaunamas kaip junginys paėmus pirmo testavimo atvejo bitų eilutės bitus nuo nulinio iki pos1, antro testavimo atvejo bitų eilutės bitus nuo $pos1 + 1$ iki $pos2$, pirmo testavimo atvejo bitų eilutės bitus nuo $pos2 + 1$ iki paskutinio bito ir juos sujungus į vieną eilutę. Testavimo atvejų yra sugeneruojama tiek, kad būtų pasiektas pradinis testavimo atvejų rinkinio dydis. Gautas naujas testavimo atvejų rinkinys naudojamas tolesniame mutacinio testavimo procese.

7.3. Mutacinio testavimo taikymo pavyzdys

Mutacinio testavimo procesui pavaizduoti yra pateiktas pavyzdys su trikampio nelygybės programa, kuri yra viena iš klasikinių programavimo problemų, pateikta 8 algoritme. Šios programos įvestis yra 3 skaičiai, o programos išvestimi yra laikomas požymis, ar skaičiai tenkina trikampio nelygybę ir iš tokio ilgio kraštinių galima sudaryti trikampį, ar ne.

Algoritmas 8 Trikampio nelygybės programa

```
1    public boolean triangle(int a, int b, int c) {
2        if(a + b > c && a + c > b && b + c > a) {
3            return true;
4        }
5        return false;
6    }
```

Toliau darbas vykdomas *program session* aplanke. Trikampio nelygybės programa pateikiama į *src* aplanką, jos sukompiliuotas kodas pateikiamas į *classes* aplanką. Toliau, taip pat kaip ir duomenų rinkinio paruošimo metu, taikant *Mujava* įrankio metodą *GenMutantsMain* aplanke *results* pateikiami sugeneruoti rezultatai: originalaus kodo baitų kodas *bytecode* ir dekompiliuotas originalus kodas aplanke *original* ir sugeneruoti mutavę kodai su metodų mutacijomis, pateikti baitų kodų formatu ir dekompiliuotu kodu, pateikti aplanke *traditional mutants*. Iš viso trikampio nelygybės programai gauti 101 mutavęs kodas.

Toliau sutvarkyti kodai *results* aplanke yra sutvarkomi, kad jie būtų galimi paleisti naudojant *Java* virtualią mašiną JVM. Šiam tikslui kiekvienam mutavusiam kodui virš klasės apibrėžimo yra pridedama *package* nuoroda su aplankų tvarka, kokia šis kodas yra pasiekiamas. Tokiu būdu yra užtikrinamas mutavusių kodų klasių unikalumas, kad JVM žinotų, kokią klasę reikia iškviesti.

Toliau 101 mutavusiam kodui yra pritaikomas 6 skyriuje aprašytas modelis. Čia riba $\delta 1$ medžio struktūros modeliui parinkta 0,1, o riba $\delta 2$ metrikų struktūros modeliui parinkta 0,2. Medžio

struktūros modelis 7 kodus pažymėjo kaip ekvivalenčius, metrikų struktūros modelis 15 kodų pažymėjo kaip ekvivalenčius, o kombinuotas modelis 22 kodus pažymėjo kaip ekvivalenčius. Todėl prieš pradedant genetinio algoritmo vykdymą iš 101 kodų 79 yra laikomi neekvivalenčiais.

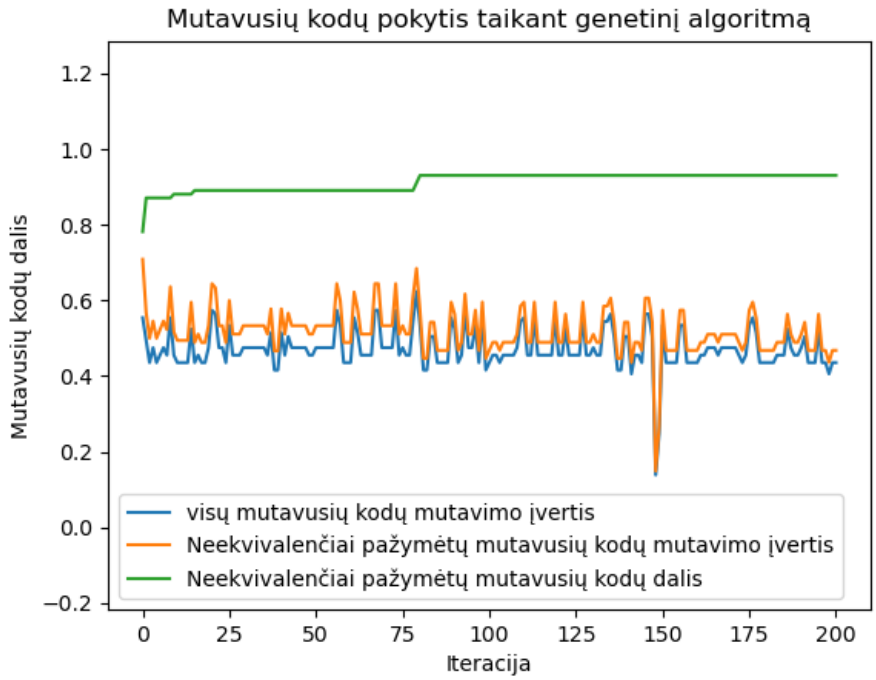
Toliau yra pateikiamas pradinis testavimo atvejų rinkinys (7 lentelė).

7 lentelė. Trikampio nelygybės programos pradinis testavimo atvejų rinkinys

a	b	c	Išvestis
10	20	30	Netiesa
2	3	4	Tiesa
0	9	10	Netiesa

Iš pradžių patikrinamas testavimo atvejų rinkinio korektiškumas, palyginant gautus rezultatus jį įvykdžius taikant pradinį kodą. Toliau yra šiam testavimo atvejų rinkiniui yra gaunamos visų mutavusių kodų išvestys. Šis testavimo rinkinys 56 kodus pažymėjo kaip sunaikintus, iš jų 9 kombinuoto modelio buvo pažymėti kaip ekvivalentūs. Kadangi 9 kodams, kurie buvo laikomi ekvivalenčiais, buvo surastas testavimo atvejis, su kuriuo jis grąžina neteisingą rezultatą, jis nebėra laikomas ekvivalenčiu.

Toliau yra vykdomas genetinis algoritmas testavimo atvejų rinkiniui generuoti. Mutavimo įverčio pokytis visiems ir tik neekvivalentiems kodams bei neekvivalenčiais laikomų mutavusių kodų dalis yra pateikta 39 pav.



39 pav. Visų mutavusių kodų ir neekvivalenčių mutavusių kodų įverčiai, bei neekvivalenčių mutavusių kodų dalis taikant genetinį algoritmą naudojant pradinį testavimo atvejų rinkinį

Genetinis algoritmas buvo vykdomas 200 iteracijų. Vykdant genetinį algoritmą iš visų kombinuoto ekvivalenčių mutavusių kodų aptikimo modelio 22 pažymėtų kodų, penkiems nebuvo rastas

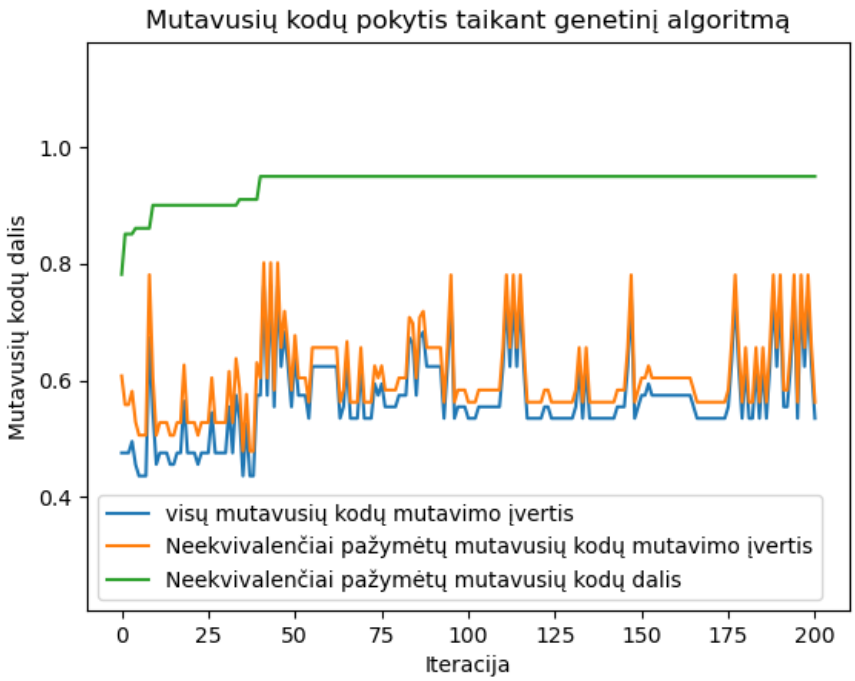
joks testavimo atvejis, kuriems mutavęs kodas pateikia nekorektišką rezultatą. Tačiau genetinio algoritmo metu neekvivalenčiais laikomų mutavusių kodų skaičius pakilo iki 96, todėl suartėjo visų mutavusių kodų mutavimo įvertis su neekvivalenčiai pažymėtų kodų mutavimo įverčiu. Genetinio algoritmo vykdymo metu buvo nepaliekami testavimo atvejai, tikrinantys metodo kraštinius atvejus, todėl galutinis testavimo atvejų rinkinys papuola į lokalaus maksimumo tašką ties 0,436 visiems mutavusiems kodams ar 0,468 neekvivalenčiais laikomiems mutavusiems kodams.

Toliau yra pateikiamas atsitiktinai sugeneruotas pradinis testavimo atvejų rinkinys (8 lentelė). Atsitiktinai generuojant pradinį testavimo atvejų rinkinį galima paspartinti mutacinio testavimo procesą, nes testuotojui nereikia galvoti pradinio testavimo atvejų rinkinio. Testavimo atvejų kiekvienas parametras generuotas 1 baito erdvėje, t. y. generuojamas įvesties parametras skaičiaus pavidalu papuola į intervalą 0–255.

8 lentelė. Atsitiktinai sugeneruotas trikampio nelygybės programos pradinis testavimo atvejų rinkinys

a	b	c	Išvestis
170	234	99	Tiesa
37	24	41	Tiesa
202	230	114	Tiesa

Iš pradžių patikrinamas testavimo atvejų rinkinio korektiškumas, palyginant gautus rezultatus jį įvykdžius taikant pradinį kodą. Toliau yra šiam testavimo atvejų rinkiniui yra gaunamos visų mutavusių kodų išvestys. Šis testavimo rinkinys 48 kodus pažymėjo kaip sunaikintus, iš jų 7 kombinuoto modelio buvo pažymėti kaip ekvivalentūs. Toliau yra vykdomas genetinis algoritmas testavimo atvejų rinkiniui generuoti. Mutavimo įverčio pokytis visiems ir tik neekvivalentiems kodams bei neekvivalenčiais laikomų mutavusių kodų dalis yra pateikta 40 pav.



40 pav. Visų mutavusių kodų ir neekvivalenčių mutavusių kodų įverčiai, bei neekvivalenčių mutavusių kodų dalis taikant genetinį algoritmą naudojant atsitikinai sugeneruotą pradinį testavimo atvejų rinkinį

Genetinis algoritmas buvo vykdomas 200 iteracijų. Galutinis mutavimo įvertis, panašiai kaip ir 39 pav., papuola į lokalų maksimumą, čia visų mutavusių kodų įvertis po 200 iteracijų yra lygus 0,535, o mutavusių kodų įvertis neekvivalenčiais laikomiems mutavusiems kodams lygus 0,563. Todėl viena iš galimų tolesnio tyrimo krypčių yra tinkamas genetinio algoritmo tinkamumo funkcijos (angl. *fitness function*) parinkimas, siekiant išvengti lokalaus maksimumo sprendinio ir pateikti testavimo atvejų rinkinį, maksimaliai padengiantį mutavusius kodus.

Rezultatai ir išvados

Darbe buvo nagrinėjama mutacinio testavimo tema ir sėkmingai įgyvendintas tikslas: sukurti mutacinio testavimo prototipą naudojant kombinuotą ekvivalenčių mutavusių kodų aptikimo modelį. Taip pat pasiekti šie rezultatai:

1. Įgyvendintas metrikų modelis ekvivalenčių mutavusių kodų aptikimo uždaviniui pagal metrikas, surinktas iš abstraktaus sintaksės medžio.
2. Įgyvendinta metrikų ir abstraktaus sintaksės medžio kombinacija, padidinanti tikimybę identifikuoti ekvivalenčius mutavusius kodus.
3. Sukurtas naujas ekvivalenčių mutavusių kodų rinkinys, sudarytas iš *Java* kalbos metodų.
4. Ekvivalenčių mutavusių kodų kombinuotas modelis sėkmingai pritaikytas mutacinio testavimo procesui taip pagerinant mutavimo atitikimo įvertį.
5. Genetinis algoritmas sėkmingai pritaikytas mutaciniam testavimui, taip patikslinant mutavimo atitikimo įvertį.

Prieš sprendžiant ekvivalenčių mutavusių kodų aptikimo uždavinį, siekiant gauti korektiškus modelius iš pradžių buvo sprendžiamas kodų klonų aptikimo uždavinys. Šiam uždaviniui taikomus modelius galima pritaikyti ekvivalenčių mutavusių kodų aptikimo problemai spręsti, koncentruojantis į kodą, kurį geba mutuoti ir sugeneruoti mutacinio testavimo įrankiai.

Šiame darbe buvo spręstas kodų klonų aptikimo uždavinys pasitelkiant abstrakčios sintaksės medžio struktūros modelį ir metrikų modelį. Darbe realizuota šių modelių kombinacija yra šio darbo rezultatas, kuris literatūros apžvalgos metu nebuvo aptiktas. Abstrakčios sintaksės medžio struktūros modelis pateikė stiprius rezultatus nagrinėjant C kodą ir šiek tiek silpnesnius rezultatus nagrinėjant *Java* kodą. Tuo tarpu metrikų modelis pateikė silpnus rezultatus nagrinėjant C kodą ir kiek stipresnius rezultatus nagrinėjant *Java* kodą. Iš esmės medžio struktūros modelis yra geresnis pasirinkimas šiam uždaviniui dėl sugebėjimo užfiksuoti kontekstinę informaciją, kur kode atsiranda pokyčių ir dėl gebėjimo atsiminti taikomą informaciją taikant uždaromąjį pasikartojantį bloką (angl. *gated reccurent unit*) ar kitus metodus.

Darbe kombinuoti gauti modeliai siekiant pagerinti rezultatus. Kombinacija įgyvendinta panaudojant testavimo rinkinį abiems modeliams testuoti ir lyginant jų rezultatus. Kadangi atliekant mutacinį testavimą ypač svarbu yra sumažinti ekvivalenčių mutavusių kodų kiekį, pasirinkta suprojektuoti galutinį modelį taip, kad jis sužymėtų kodus klonais tuo atveju, jeigu nors vienas iš modelių jį tokį identifikavo. Gauto galutinio modelio tikslumas dėl to šiek tiek sumažėjo, tačiau nežymiai padidino modelio išsamumą, nes aptikta daugiau klonų. Tokiu būdu taikant modelių kombinacijas galima pagerinti ir paties geriausio modelio išsamumo įvertį.

Kodų klonų uždavinio sprendimo metu aprašytos struktūros buvo pritaikytos ekvivalenčių mutavusių kodų aptikimo uždaviniui. Modeliams mokyti buvo kuriamas naujas ekvivalenčių mutavusių kodų duomenų rinkinys, susidedantis iš kodų[5] ir jų porų[6]. Naujo duomenų rinkinio gamyba

[5]https://github.com/Aurimasjar/Mutation-Testing/blob/main/clone/data/javamut/mut_funcs_all.csv
[6]https://github.com/Aurimasjar/Mutation-Testing/blob/main/clone/data/javamut/mut_pair_ids.csv

reikalauja daug laiko (siekiant gauti gausiai padengtą duomenų rinkinį su užtektinai duomenų) ir kruopštumo (siekiant gauti duomenų rinkinį, kuriame ekvivalentūs mutavę kodai yra sužymėti teisingai). Gautą duomenų rinkinį sudaro 9769 programos ir 9692 poros. Lyginant pagal porų skaičių šis duomenų rinkinys yra didesnis už bet kurį šiuo metu viešai prieinamą mutavusių kodų rinkinį. Taip pat tai vienintelis viešai prieinamas duomenų rinkinys, nagrinėjantis mutavusius kodus metodų lygiu. Metrikų modelis šiam duomenų rinkiniui pateikia neužtikrintus rezultatus dėl per reto duomenų rinkinio, o medžio struktūros modelis šiam duomenų rinkiniui pateikia kiek tikslesnius rezultatus, tačiau rezultatai vis tiek prastesni, nei sprendžiant kodų klonų aptikimo uždavinį. Kombinuojant modelius, galima parinkti žemas porų identifikavimo ribas, tačiau tokiu būdu aukojamas modelio tikslumas. Norint gauti tikslesnius rezultatus, reikėtų plėsti mutavusių kodų duomenų rinkinį. Tai yra viena iš galimų tolesnio tyrimo krypčių.

Tolesnė galima ekvivalenčių mutavusių kodų aptikimo uždavinio tyrimo kryptis galėtų būti kitų modelių kombinacijų išbandymas siekiant pagerinti galutinio modelio įvertį. Vienas iš galimų metodų yra modelio skatinimas (angl. *boosting*), kada vieno modelio išvestis galėtų būti naudojama kaip kito modelio įvestis taip siekiant pagerinti galutinę išvestį. Kitas galimas modelių kombinacijos metodas yra sudėjimas (angl. *stacking*), kuris aprašomas meta algoritmu (angl. *meta algorithm*), kuris priima skirtingų modelių pateiktas išvestis ir išmoksta, kaip pasiekti geriausią tikslumą kombinuojant skirtingų modelių pateiktas tikimybes. Taip pat pravartu būtų pritaikyti kombinacijas taikant modelius, kurios taiko kitas metodikas, pasitelkiančias programos priklausomybių grafus, taip pat tekstinę ar sintaksinę kodo informaciją.

Kitos galimos ekvivalenčių mutavusių kodų aptikimo uždavinio tyrimo kryptys galėtų būti *Lime* ir *Shap* technologijų pritaikymas, siekiant pagrįsti modelį aprašant jų spėjimų paaiškinimus. Taip pat galima optimizuoti metrikų modelį modelio mokymo metu validavimo sluoksnyje vykdant hiperparametrų derinimą (angl. *hyperparameter tuning*). Šis procesas leistų optimizuoti taikomą modelį ir leistų gauti geresnius modelio rezultatus. Šį procesą būtų aktualu realizuoti metrikų modeliui, kadangi jis yra paremtas [SFL+18] straipsniu ir nėra bandytas optimizuoti.

Darbe pateiktas mutacinio testavimo prototipas taikant ekvivalenčių mutavusių kodų aptikimo uždavinį. Parodyta, kad galima įvertinti pasirinkto testavimo atvejų rinkinio korektiškumą patikrinant jo padengimą mutavusių kodų rinkiniui ir jo pagerėjimą pritaikius ekvivalenčių mutavusių kodų aptikimo modelius. Taip pat parodyta, kad kombinuojant genetinį algoritmą testavimo atvejams generuoti ir ekvivalenčių mutavusių kodų aptikimo modelius galima pagerinti mutavimo įvertį iš mutavusių kodų rinkinio pašalinant mutavusius kodus, kurių nesunaikino joks testavimo atvejis ir modelio yra pažymėtas kaip ekvivalentus. Genetinis algoritmas gali būti tobulinamas siekiant sugeneruoti adekvatų testavimo atvejų rinkinį, padengiantį visus neekvivalenčius kodus. Parodyta, kad pritaikius genetinį algoritmą ir ekvivalenčių mutavusių kodų aptikimo modelį galima gauti galutinį kodų rinkinį mutavimo įverčių skaičiavimui, kurį sudaro tik neekvivalentūs kodai.

Literatūra

[ABA+19] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam ir B. Maqbool. A Systematic Review on Code Clone Detection. *IEEE Access*, 7:86121–86144, 2019. DOI: 10.1109/ACCESS.2019.2918202.

[ABA07] K. Ayari, S. Bouktif ir G. Antoniol. Automatic Mutation Test Input Data Generation via Ant Colony ABSTRACT. In p. 1074–1081, 2007. DOI: 10.1145/1276958.1277172.

[ABC+13] S. Anand, E. K. Burke, T. Y. Chen, J. Clark ir kt. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013. DOI: 10.1016/j.jss.2013.02.061.

[AHH04] K. Adamopoulos, M. Harman ir R. Hierons. How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution. In t. 3103, p. 1338–1349, 2004. DOI: 10.1007/978-3-540-24855-2_155.

[BA82] T. A. Budd ir D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*:31–45, 1982. DOI: 10.1007/BF00625279.

[BD08] du L. Bousquet ir M. Delaunay. Towards Mutation Analysis for Lustre Programs. *Electronic Notes in Theoretical Computer Science*, 203(4):35–48, 2008. DOI: 10.1016/j.entcs.2008.05.009.

[Car02] J. S. Carson. Model verification and validation. In *Proceedings of the Winter Simulation Conference*, t. 1, 52–58 vol.1, 2002. DOI: 10.1109/WSC.2002.1172868.

[CY22] S. Chung ir S. Yoo. Augmenting Equivalent Mutant Dataset Using Symbolic Execution. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, p. 150–159, 2022. DOI: 10.1109/ICSTW55395.2022.00038.

[CKN+12] E. M. Clarke, W. Klieber, M. Nováček ir P. Zuliani. *Model Checking and the State Explosion Problem*. In *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. 2012, p. 1–30. DOI: 10.1007/978-3-642-35746-6_1.

[EIP07] M. Ellims, D. Ince ir M. Petre. The Csaw C Mutation Tool: Initial Results. In *Testing: Academic and Industrial Conference - Practice and Research Techniques (TAICPART)*, p. 185–192, 2007. DOI: 10.1109/TAIC.PART.2007.28.

[FOW87] J. Ferrante, K. J. Ottenstein ir J. D. Warren. The Program Dependence Graph and Its Use in Optimization. 9:319–349, 1987. DOI: 10.1145/24039.24041.

[GM16] V. Garousi ir M. V. Mäntylä. When and what to automate in software testing? A multivocal literature review. *Information and Software Technology*, 76:92–117, 2016. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2016.04.015.

[GS16] P. Gautam ir H. Saini. Various Code Clone Detection Techniques and Tools: A Comprehensive Survey. In p. 655–667, 2016. ISBN: 978-981-10-3432-9. DOI: 10.1007/978-981-10-3433-6_79.

[GSZ09] B. J. M. Grün, D. Schuler ir A. Zeller. The Impact of Equivalent Mutants. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, p. 192–199, 2009. DOI: 10.1109/ICSTW.2009.37.

[HHB14] N. Hung-Cuong, T. Huynh Quyet ir T. Ba-Vuong. Rule-Based Techniques Using Abstract Syntax Tree for Code Optimization and Secure Programming in Java. In *International Conference on Context-Aware Systems and Applications*, p. 168–177, 2014. DOI: 10.1007/978-3-319-05939-6_17.

[HHD01] M. Harman, R. Hierons ir S. Danicic. *The Relationship between Program Dependence and Mutation Analysis*. In *Mutation Testing for the New Century*. Kluwer Academic Publishers, 2001, p. 5–13. DOI: 10.5555/571305.571310.

[HHG+00] R. Hierons, M. Harman, E. Grove ir N. Db. Using Program Slicing to Assist in the Detection of Equivalent Mutants. *Software Testing Verification and Reliability*, 9:232–262, 2000. DOI: 10.1002/(SICI)1099-1689(199912)9:43.0.CO;2-3.

[HO21] L. Hijfte ir A. Oprescu. MutantBench: an Equivalent Mutant Problem Comparison Framework. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, p. 7–12, 2021. DOI: 10.1109/ICSTW52544.2021.00015.

[YLR12] F. Yamaguchi, M. Lottmann ir K. Rieck. Generalized Vulnerability Extrapolation Using Abstract Syntax Trees. In p. 359–368, New York, NY, USA. Association for Computing Machinery, 2012. DOI: 10.1145/2420950.2421003.

[Ino21] K. Inoue. *Introduction to Code Clone Analysis*. In *Code Clone Analysis*. 2021, p. 3–27. DOI: 10.1007/978-981-16-1927-4_1.

[ISO21] ISO. Software and systems engineering — Software testing — Part 4: Test techniques. Techn. atask., International Organization for Standardization, Geneva, CH, 2021.

[ISO22] ISO. Software and systems engineering — Software testing — Part 1: General concepts. Techn. atask., International Organization for Standardization, Geneva, CH, 2022.

[JAA+16] A. Jamil, M. Arif, N. Abubakar ir A. Ahmad. Software Testing Techniques: A Literature Review. In p. 177–182, 2016. DOI: 10.1109/ICT4M.2016.045.

[JCX+09] C. Ji, Z. Chen, B. Xu ir Z. Wang. A New Mutation Analysis Method for Testing Java Exception Handling. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, t. 2, p. 556–561, 2009. DOI: 10.1109/COMPSAC.2009.192.

[JH09] Y. Jia ir M. Harman. Higher Order Mutation Testing. *Information and Software Technology*, 51:1379–1393, 2009. DOI: 10.1016/j.infsof.2009.04.016.

[JH11] Y. Jia ir M. Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011. DOI: 10.1109/TSE.2010.62.

[KA15] R. Khan ir M. Amjad. Automatic test case generation for unit software testing using genetic algorithm and mutation analysis. In *2015 IEEE UP Section Conference on Electrical Computer and Electronics (UPCON)*, p. 1–5, 2015. DOI: 10.1109/UPCON.2015.7456734.

[KMS+22] M. B. Kusharki, B. Misra S. M., I. A. Salihu ir B. Suri. Automatic Classification of Equivalent Mutants in Mutation Testing of Android Applications. *Symmetry*, 14(4), 2022. DOI: 10.3390/sym14040820.

[KO91] K. N. King ir A. J. Offutt. A fortran language system for mutation-based software testing. *Software: Practice and Experience*, 21(7):685–718, 1991. DOI: https://doi.org/10.1002/spe.4380210704.

[Kum15] R. Kumar. Heuristic Approaches for Equivalent Mutant Problem. *International Journal for Scientific Research and Development*, 3:345–349, 2015. DOI: nežinomas.

[LCM+07] A. Leitner, I. Ciupa, B. Meyer ir M. Howard. Reconciling Manual and Automated Testing: The AutoTest Experience. In *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, 261a–261a, 2007. DOI: 10.1109/HICSS.2007.462.

[Mah20] B. Mahesh. Machine learning algorithms-a review. *International Journal of Science and Research (IJSR).[Internet]*, 9:381–386, 2020. DOI: 10.21275/ART20203995.

[MAM09] K. Musa, R. Al-Qutaish ir M. Muhairat. Classification of Software Testing Tools Based on the Software Testing Methods. In t. 1, 2009-12. DOI: 10.1109/ICCEE.2009.9.

[MB99] E. Mresa ir L. Bottaci. Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study. *Software Testing Verification and Reliability*, 9:205–232, 1999. DOI: 10.1002/(SICI)1099-1689(199912)9:43.0.CO;2-X.

[MLJ+14] L. Mou, G. Li, Z. Jin, L. Zhang ir T. Wang. Convolutional Neural Network over Tree Structures for Programming Language Processing. In *The 30th AAAI Conference on Artificial Intelligence (AAAI)*, 2014. DOI: 10.13140/RG.2.1.2912.2966.

[MNZ+05] M. Masud, A. Nayak, M. Zaman ir N. Bansal. Strategy for mutation testing using genetic algorithms. In *Canadian Conference on Electrical and Computer Engineering, 2005.* P. 1049–1052, 2005. DOI: 10.1109/CCECE.2005.1557156.

[MOK06] Y. Ma, J. Offutt ir Y. Kwon. MuJava: A Mutation System for Java. In p. 827–830. Association for Computing Machinery, 2006. DOI: 10.1145/1134285.1134425.

[MOK16] Y. Ma, J. Offutt ir Y. Kwon. Description of muJava's Method-level Mutation Operators, 2016. DOI: nežinomas. URL: https://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf. 152 KB, tikrinta 2023-05-06.

[MOT+14] L. Madeyski, W. Orzeszyna, R. Torkar ir M. Józala. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Transactions on Software Engineering*, 40(1):23–42, 2014. DOI: 10.1109/TSE.2013.44.

[MSB12] G. J. Myers, C. Sandler ir T. Badgett. *The Art of Software Testing*. 2012. DOI: 10.1002/stvr.322.

[MX07] E. Martin ir T. Xie. A fault model and mutation testing of access control policies. In p. 667–676, 2007. DOI: 10.1145/1242572.1242663.

[NLN+19] M. R. Naeem, T. Lin, H. Naeem ir H. Liu. A machine learning approach for classification of equivalent mutants. *Journal of Software: Evolution and Process*:1–32, 2019. DOI: 10.1002/smr.2238.

[NPM+17] A. S. Nuñez-Varela, H. G. Pérez-Gonzalez, F. E. Martínez-Perez ir C. Soubervielle-Montalvo. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128:164–197, 2017. DOI: 10.1016/j.jss.2017.03.044.

[NVR16] P. Neculoiu, M. Versteegh ir M. Rotaru. Learning Text Similarity with Siamese Recurrent Networks. In *Proceedings of the 1st Workshop on Representation Learning for NLP*, p. 148–157. Association for Computational Linguistics, 2016. DOI: 10.18653/v1/W16-1617.

[OC94] A. J. Offutt ir W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing*, 4, 1994. DOI: 10.1002/STVR.4370040303.

[Off89a] A. Offutt. The Coupling Effect: Fact or Fiction. In *TAV3: Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification*, p. 131–140, 1989. DOI: 10.1145/75308.75324.

[Off89b] A. Offutt. The Coupling Effect: Fact or Fiction. 14(8):131–140, 1989. DOI: 10.1145/75309.75324.

[OFS+19] S. Owens, M. Flatt, O. Shivers ir B. McMullan. Lexer and Parser Generators in Scheme. In p. 783–794, 2019. DOI: 10.1109/ICSE.2019.00086.

[OP96] A. J. Offutt ir Jie P. Detecting equivalent mutants and the feasible path problem. In *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS '96*, p. 224–236, 1996. DOI: 10.1109/CMPASS.1996.507890.

[PDD+21] S. Peacock, L. Deng, J. Dehlinger ir S. Chakraborty. Automatic Equivalent Mutants Classification Using Abstract Syntax Tree Neural Networks. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, p. 13–18, 2021. DOI: 10.1109/ICSTW52544.2021.00016.

[PIF+22] G. Petrović, M. Ivanković, G. Fraser ir R. Just. Practical Mutation Testing at Scale: A view from Google. *IEEE Transactions on Software Engineering*, 48(10):3900–3912, 2022. DOI: 10.1109/TSE.2021.3107634.

[PS22] S. Patel ir R. Sinha. Combining Holistic Source Code Representation with Siamese Neural Networks for Detecting Code Clones. In *Testing Software and Systems*, p. 148–159. Springer International Publishing, 2022. DOI: 10.1007/978-3-031-04673-5_12.

[RC08] C. K. Roy ir J. R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, p. 172–181. IEEE Computer Society, 2008. DOI: 10.1109/ICPC.2008.41.

[SAR+17] I. S. Santos, R. M. C. Andrade, L. Souza R., S. Matalonga, K. M. Oliveira ir G. H. Travassos. Test case design for context-aware applications: Are we there yet? *Information and Software Technology*, 88:1–16, 2017. DOI: https://doi.org/10.1016/j.infsof.2017.03.008.

[SFL+18] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi ir C. Lopes. Oreo: Detection of Clones in the Twilight Zone. *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (2018)*:354–365, 2018. DOI: 10.1145/3236024.3236026.

[SGP+14] Z. Sahaf, V. Garousi, D. Pfahl, R. Irving ir Y. Amannejad. When to automate software testing? decision support based on system dynamics: An industrial case study. In 2014-05. ISBN: 978-1-4503-2754-1. DOI: 10.1145/2600821.2600832.

[SIK+14] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy ir M. M. Mia. Towards a Big Data Curated Benchmark of Inter-Project Code Clones. In *ICSME '14: Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, p. 476–480, 2014. DOI: 10.1109/ICSME.2014.77.

[SPL+16] F. C. M. Souza, M. Papadakis, Y. Le Traon ir M. E. Delamaro. Strong Mutation-Based Test Data Generation Using Hill Climbing. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, p. 45–54, 2016. DOI: 10.1145/2897010.2897012.

[SR15] J. Svajlenko ir C. K. Roy. Evaluating clone detection tools with BigCloneBench. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, p. 131–140, 2015. DOI: 10.1109/ICSM.2015.7332459.

[SR17] J. Svajlenko ir C. K. Roy. Fast and Flexible Large-Scale Clone Detection with CloneWorks. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, p. 27–30, 2017. DOI: 10.1109/ICSE-C.2017.3.

[SRK+21] G. Shobha, A. Rana, V. Kansal ir S. Tanwar. Code Clone Detection - A Systematic Review. In *Emerging Technologies in Data Mining and Information Security*, p. 645–655. Springer Nature Singapore, 2021.

[SRS17] R. A. Silva, S. Rocio ir P. Sérgio. A systematic review on search based mutation testing. *Information and Software Technology*, 81:19–35, 2017. DOI: https://doi.org/10.1016/j.infsof.2016.01.017.

[SZ13] D. Schuler ir A. Zeller. Covering and Uncovering Equivalent Mutants. *Software Testing, Verification and Reliability*, 23, 2013. DOI: 10.1002/stvr.1473.

[VNM+02] A. Vincenzi, E. Nakagawa, J. Maldonado, M. Delamaro ir R. Romero. Bayesian-Learning Based Guidelines to Determine Equivalent Mutants. *International Journal of Software Engineering and Knowledge Engineering*, 12:675–689, 2002. DOI: 10.1142/S021819400200113X.

[VP15] M. Venkat ir M. Prasanna. Generation of Test Case using Automation in Software Systems – A Review. *Indian Journal of Science and Technology*, 8:1–9, 2015. DOI: 10.17485/ijst/2015/v8i35/72881.

[WL17] H. Wei ir M. Li. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, p. 3034–3040, 2017. DOI: 10.5555/3172077.3172312.

[WWZ14] K. Wang, Y. Wang ir L. Zhang. Software testing method based on improved simulated annealing algorithm. In *2014 10th International Conference on Reliability, Maintainability and Safety (ICRMS)*, p. 418–421, 2014. DOI: 10.1109/ICRMS.2014.7107215.

[ZWZ+19] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang ir X. Liu. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, p. 783–794. IEEE Press, 2019. DOI: 10.1109/ICSE.2019.00086.

A. Papildomi tyrimai metrikų modelio C kalbos kodo klonų uždaviniui

Šiame priede pateikti papildomi tyrimai, skirti C kalbos kodo klonų uždaviniui spręsti. Pagrindiniame tekste pateikti tik trečiosios ir galutinės metrikų modelio versijos rezultatai.

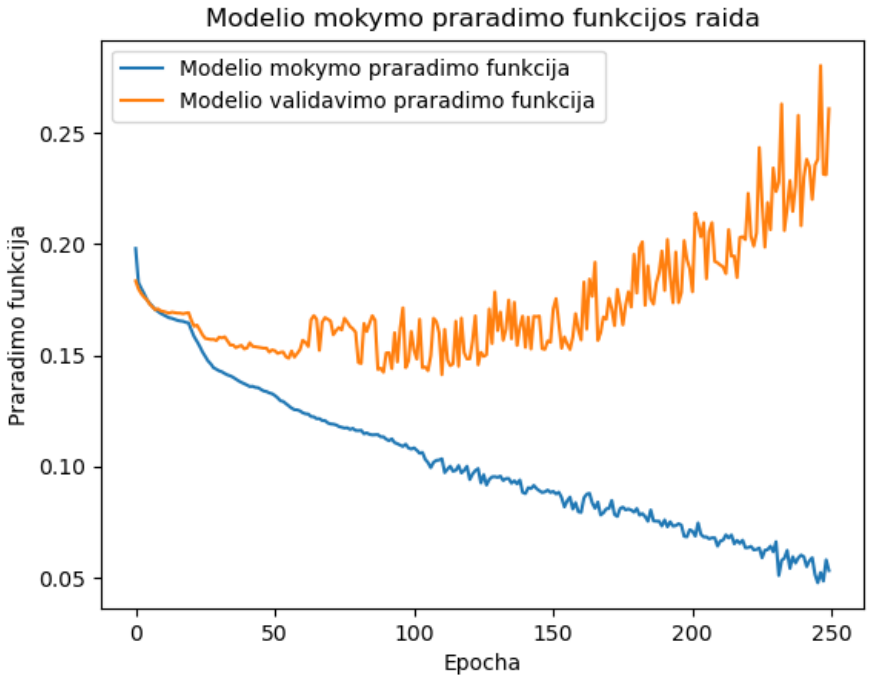
A.1. Pirmosios metrikų modelio versijos rezultatai

Gauti rezultatai naudojant duomenis, apibrėžtus 75 metrikomis, pateikti 9 lentelėje.

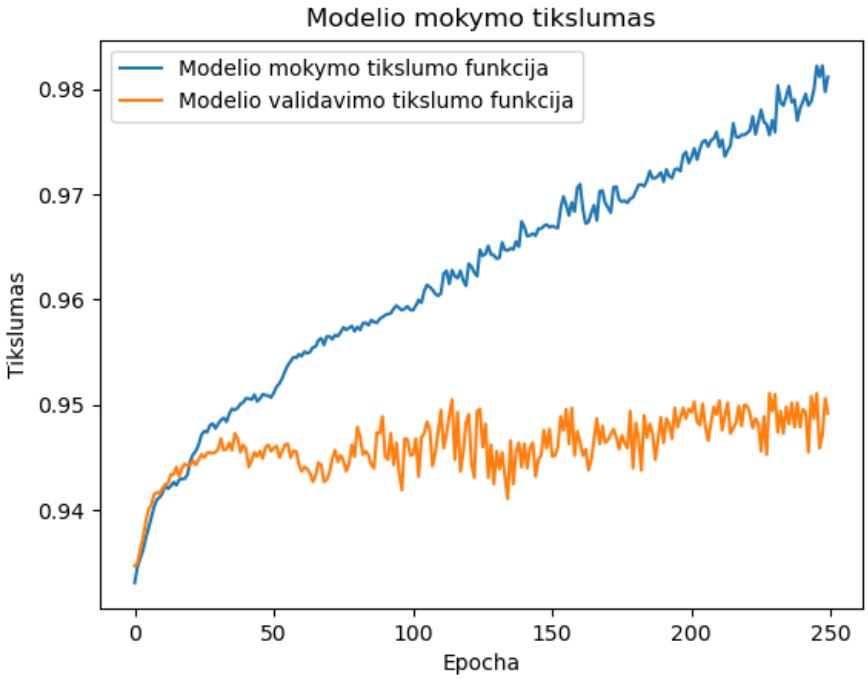
9 lentelė. Pradinio metrikų modelio testavimo rezultatai

Modelis	Kalba	Epochų skaičius	Tikslumas	Išsamumas	f1
Metrikų modelis	C	150	0,688	0,403	0,508
Metrikų modelis	C	250	0,614	0,467	0,531

Iš pradžių buvo bandoma mokyti tinklą ant didelio epochų skaičiaus, pasirenkant epochų skaičių, lygų 150 ir 250, tačiau pastebėta, kad esant didesniam epochų skaičiui krinta tikslumas (9 lentelė). Todėl prie modelio buvo pridėtas validavimo sluoksnis, kuris patikrina modelio tikslumą modelio mokymo metu. Modelio mokymo rezultatai, modelius apmokant iki 250-tos epochos, pateikti 41 ir 42 pav.

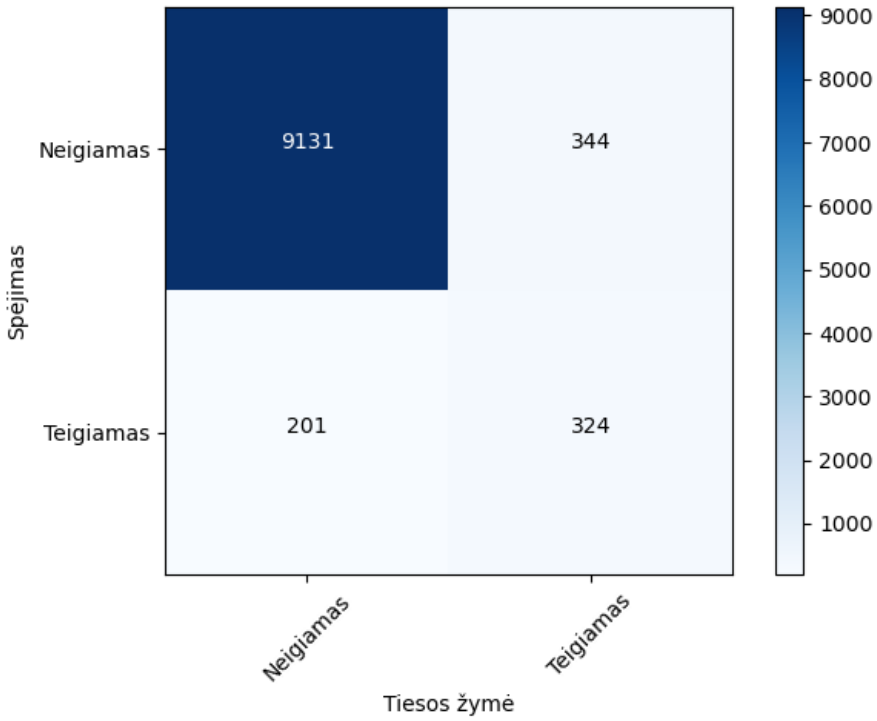


41 pav. Pirmosios versijos metrikų modelio praradimo funkcija *BCELoss*



42 pav. Pirmosios versijos metrikų modelio tikslumo funkcija

43 pav. yra pateikta modelio klaidų matrica (angl. *confusion matrix*).



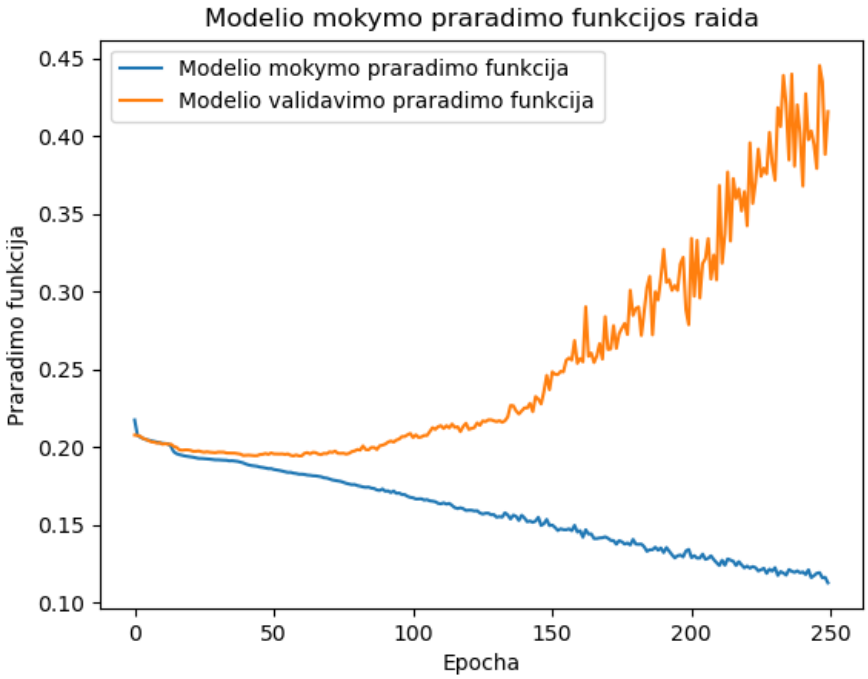
43 pav. Pirmosios versijos metrikų modelio klaidų matrica

Pagal 43 pav. iš 9332 neigiamų atvejų 201 buvo klaidingai klasifikuoti teigiamai. Taip pat iš 525 teigiamai identifikuotų atvejų 324 identifikuoti teisingai, tai pažymi 61,7 % tikslumą. O iš 668 teigiamų atvejų 324 identifikuoti teisingai, tai pažymi 48,5 % išsamumą. Šio modelio f1

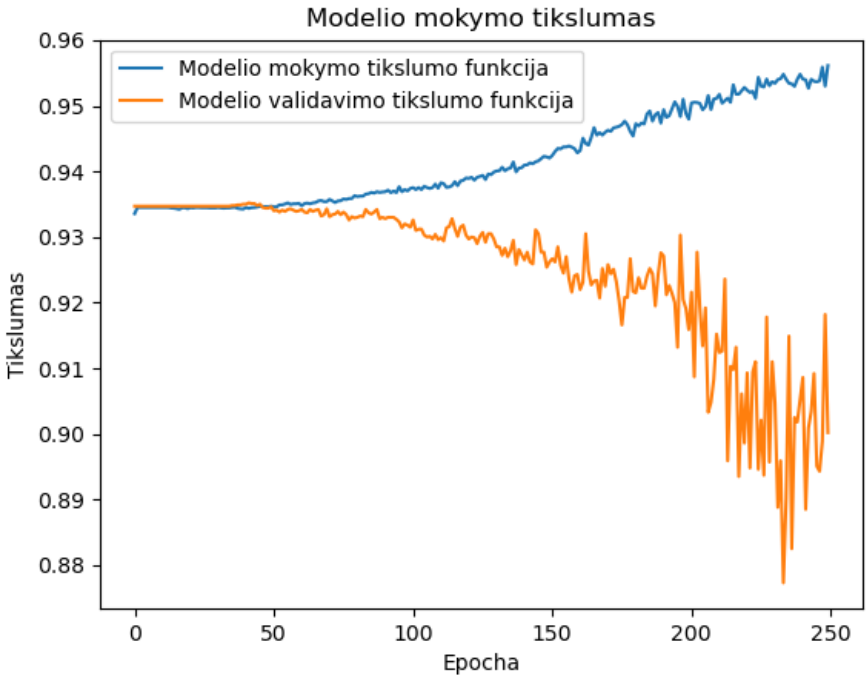
įvertis lygus 0,543. Šie rezultatai palyginti su gautais kito modelio rezultatais, kur išspręsta modelio permokymo problema.

A.2. Antrosios metrikų modelio versijos rezultatai

Siekiant išvengti modelio persimokymo problemos, metrikų sąrašas sumažintas iki 21, parenkant metrikas, kurios bendrais bruožais apibūdina kodą. Modelio rezultatai, modelius apmokant iki 250-tos epochos, pateikti 44 ir 45 pav.



44 pav. Antrosios versijos metrikų modelio praradimo funkcija *BCELoss*

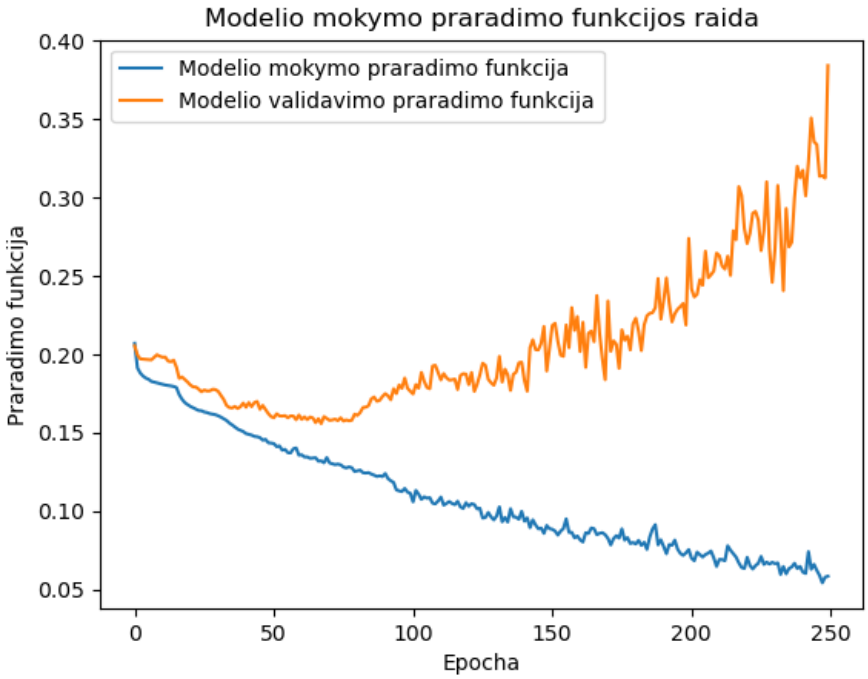


45 pav. Antrosios versijos metrikų modelio tikslumo funkcija

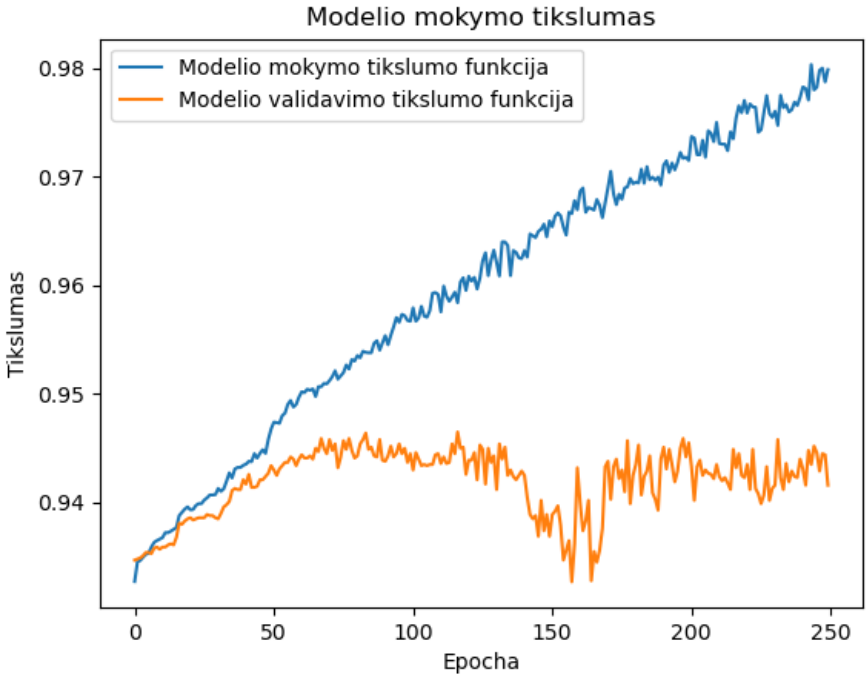
Pastebėta, kad tinklą mokant taikant bendrąsias metrikas modelio tikslumas nedidėja. Po kurio laiko tikslumas pradeda prastėti dėl modelio permokymo todėl, kad modelis pagauna konkrečias metrikų reikšmes ir pagal jas daro išvadas, nors tai yra per mažo duomenų rinkinio problema. Matoma, kad reikia palikti metrikas, nusakančias pradinių kodų kontekstinę informaciją (kiekvieno palyginimo, aritmetinio operatoriaus skaičiaus vietoj vienos bendros metrikos, skaičiuojančios visus).

A.3. Trečiosios metrikų modelio versijos papildomi rezultatai

Trečiosios modelio versijos mokymo rezultatai, modelius apmokant iki 250-tos epochos, pateikti 46 ir 47 pav.



46 pav. Trečiosios versijos metrikų modelio praradimo funkcija *BCELoss*

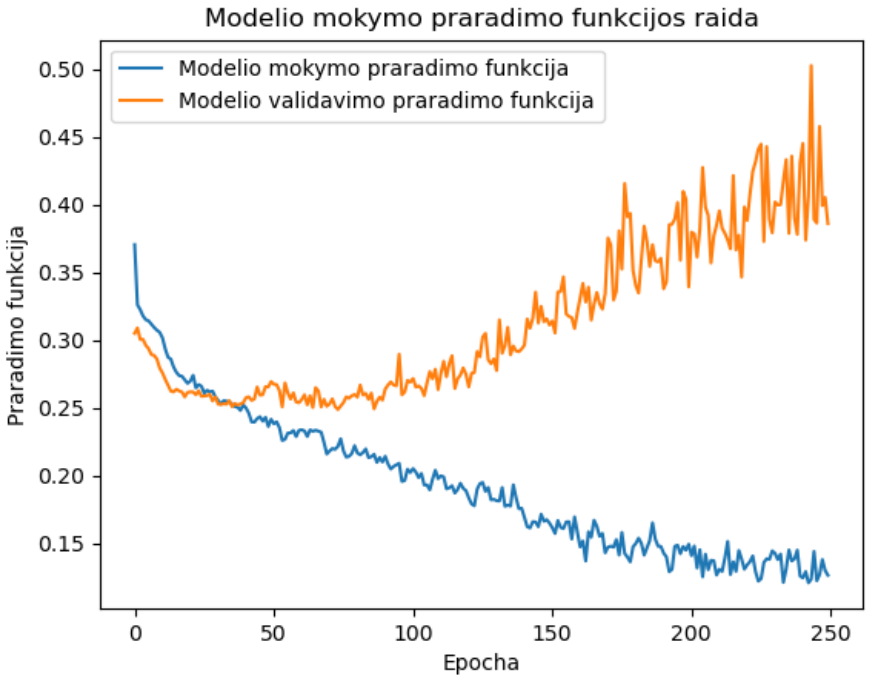


47 pav. Trečiosios versijos metrikų modelio tikslumo funkcija

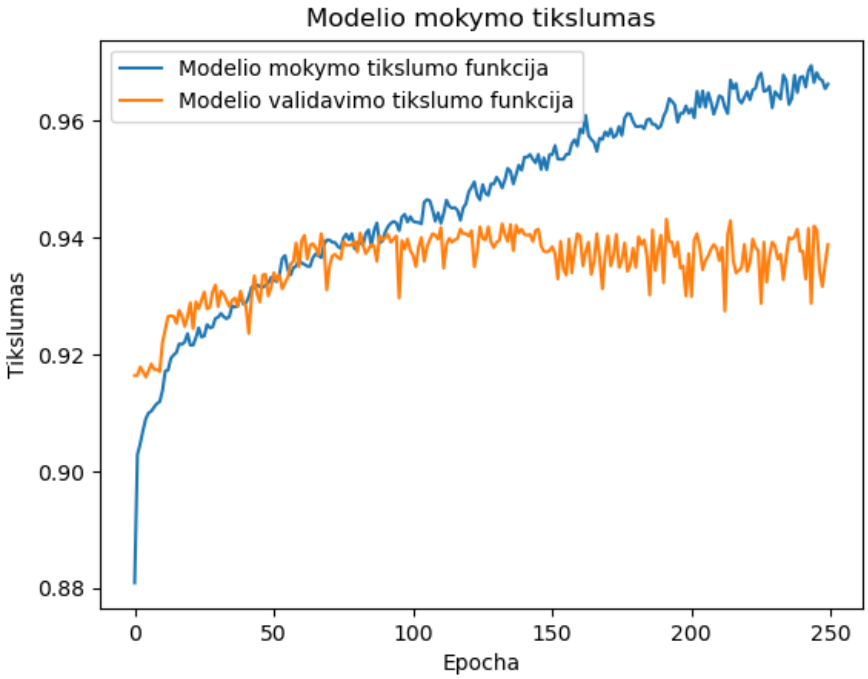
B. Papildomi tyrimai metrikų modelio *Java* kalbos kodo klonų uždaviniui

Šiame priede pateikti papildomi grafikai, gauti *Java* kalbos kodo klonų uždaviniui spręsti, kurie iliustruoja modelio permokymą.

Modelio mokymo rezultatai, modelius apmokant iki 250-tos epochos, pateikti 48 ir 49 pav.



48 pav. *Java* kodo metrikų modelio praradimo funkcija *BCELoss*



49 pav. *Java* kodo metrikų modelio tikslumo funkcija

C. Ekvivalenčių mutavusių kodų metrikų sąrašas

10 ir 11 lentelėse pateiktas *Java* kodų metrikų sąrašas, taikytų ekvivalenčių mutavusių kodų aptikimo uždaviniui. Taip pat visam mutavusių kodų rinkiniui apskaičiuoti metrikų vidurkiai ir vidutiniai nuokrypiai. Šie duomenys atskleidžia, kad dėl skirtingo metrikų pasikartojimo koduose pateikiant metrikų sąrašą į neuroninį tinklą reikėjo duomenis normalizuoti.

10 lentelė. Mutavusių kodų modelio metrikų sąrašo pirma dalis

Metrika	Vidurkis	Vidutinis nuokrypis
if_count	1.222	0.859
for_count	1.809	1.53
while_count	0.696	1.059
break_count	0.0	0.0
continue_count	0.0	0.0
method_inv_count	1.045	1.18
return_count	1.53	1.013
cast_count	0.011	0.145
variable_decl_count	5.166	3.378
local_variable_decl_count	3.694	2.454
member_ref_count	35.23	20.485
ref_type_count	1.308	3.802
literal_count	5.725	3.383
arr_count	8.838	6.94
block_count	3.879	2.113
end_count	3.728	1.911
null_count	0.0	0.0
true_count	0.335	0.811
false_count	0.269	0.61
basic_type_count	0.128	0.336
char_count	0.001	0.036
string_count	0.001	0.036
int_count	9.264	4.603
long_count	0.427	1.264
double_count	0.436	1.267
bool_count	6.552	3.664
assignment_count	0.708	1.539
sum_assignment_count	1.056	2.153
min_assignment_count	0.521	1.013

11 lentelė. Mutavusių kodų modelio metrikų sąrašo antra dalis

Metrika	Vidurkis	Vidutinis nuokrypis
mul_assignment_count	3.91	3.219
div_assignment_count	0.209	0.571
is_lower_count	2.483	1.878
is_upper_count	0.233	0.433
is_lower_or_equal_count	0.282	0.46
is_upper_or_equal_count	0.338	0.648
is_equal_count	0.367	0.672
is_not_equal_count	0.459	0.733
and_count	0.667	0.846
or_count	0.12	0.327
not_count	0.186	0.39
bitwise_not_count	0.115	0.318
bitwise_and_count	0.0	0.0
bitwise_or_count	0.0	0.0
bitwise_xor_count	0.004	0.065
plus_count	1.732	1.511
minus_count	1.51	1.254
mul_count	0.123	0.446
div_count	0.273	0.546
rem_count	0.033	0.178
binary_op_count	16.839	11.186
ternary_op_count	8.781	7.488
increment_count	2.972	2.934
decrement_count	0.304	0.509
prefix_count	1.214	1.063
postfix_count	2.687	2.442
leaf_count	79.979	39.301
node_count	185.974	94.777
max_depth	10.579	1.924