VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
KOMPIUTERIJOS KATEDRA

Magistro baigiamasis darbas

# Programinio kodo palyginimo metodas

Atliko: Juozas Rimša . . . . . .
(parašas)

Darbo vadovas:
dr. L. Bukauskas . . . . . .
(parašas)

Vilnius
2012

Turinys

# Sutartiniai ženklai, vienetai ir terminų sutrumpinimai

- Medis – beciklis sujungtas grafas, kurio kiekviena viršūnė turi 0 arba daugiau vaikų ir daugiausiai 1 tėvą.
- Miškas – grafas, sudarytas iš vieno ar kelių medžių.
- Sutvarkytas žymėtas medis (angl. *ordered labeled tree*) – medis, kurio kiekviena viršūnė yra pažymėta ir viršūnės vaikų tarpusavio tvarka yra svarbi.
- Multiaibė (angl. *multiset* arba *bag*) – aibė, kurioje elementai gali kartotis.
- Linijinis grafas – medis, kuris neturi išsišakojimų; toks grafas turi dvi arba daugiau viršūnių; viršūnių laipsnis yra 1 arba 2.
- AST (angl. *abstract syntax tree*) – abstraktus sintaksės medis, t.y. abstrakti išeities kodo išraiška medžio pavidalu. Abstraktumas žymi tai, kad ne kiekviena kodo detalė atspindima medyje mazgu (pvz. operacijų grupavimo skliaustai).
- XML (angl. *extensible markup language*) – duomenų struktūrų apibrėžimo kalba, kuri paremta SGML.
- Maišos funkcija (angl. *hash function*) – funkcija duomenų aibę vienpusiškai transformuojanti į mažesnę raktų aibę.

# Anotacija

Darbe nagrinėjama programinio kodo palyginimo, siekiant rasti funkcinius dublikatus, problema. Tikslas – sukurti bei pasiūlyti universalų trivialiems pakeitimams atsparų programinio kodo palyginimo metodą. Darbe aprašomi du pasirinkti medžių palyginimo algoritmai: tikslus medžių redagavimo atstumo ir apytikslis pq-gram, pasižymintis geresniu našumu. Šie algoritmai praktiškai pritaikyti Java programinio kodo palyginimui: parašytas AST parseris bei programa, lyginanti XML dokumentus arba Java klases. Darbe įvertintos galimos AST transformacijos bei eksperimentiškai parinktos programinio kodo palyginimui tinkamiausios p ir q parametrų reikšmės pq-gram algoritmui. Pasiūlytas pq-gram algoritmo patobulinimas, pq-gram indeksuose saugant maišos funkcijų grąžintus įrašų atvaizdus.

# Summary

This paper investigates a problem of source code comparison with a goal of finding functional code duplicates. The goal of this paper is to create and present a universal source code comparison method, which would be resistant to trivial changes. Two algorithms were chosen to achieve this goal: exact tree edit distance and approximate pq-gram, which has better performance. Both algorithms were practically applied for Java code comparison: an AST parser and a program, which compares XML documents or Java classes, were created. This paper also evaluates AST transformations as well as presents experimental results of p and q values best suited for code comparison. Args4j library is used to evaluate the quality of source code comparison results

# Įvadas

Tobulinamos technologijos leidžia kurti vis sudėtingesnės ir didesnės apimties programų sistemas. Siekiant jas pritaikyti pastoviai kintantiems vartotojų poreikiams, programos būna tobulinamos, keičiamos, pritaikomos, atsižvelgiant į pasikeitusias problemas. Šis programų sistemos funkcionalumo ar jo pasikeitimų įgyvendinimas lemia nuolatinį programų sistemos išeities kodo apimties augimą. Toks kodo apimties augimas savo ruožtu kelia tam tikras problemas ir šalutinius efektus. Vienas iš jų – išeities kodo dublikatų atsiradimas.

Sintaksiškai skirtingi, bet funkcionaliai tapatūs išeities kodo dublikatai -– tai programos kodo fragmentai, kurių funkcionalumas yra vienodas arba labai panašus. Programos kode jie dažniausiai atsiranda dėl pakartotinio kodo panaudojimo kopijuojant panašaus ar identiško kodo fragmentą arba iš nežinojimo apie kitur egzistuojantį tokio funkcionalumo kodą, parašant jį iš naujo. Atlikti didelės apimties programų sistemų kodo vertinimai rodo, kad kodo dublikatų apimtis yra reikšminga ir gali siekti apie 6-13% viso sistemos kodo [LPM+97], [Bak95]. Taigi, jų aptikimas ir pašalinimas leistų paprasčiau ir greičiau palaikyti ir plėtoti sistemą, sumažintų šalutinių efektų kiekį sistemoje, pagerintų sitemos greitaveiką ir panašiai.

Programinio kodo dublikatai didina funkcionalumo įgyvendinimo kainą bei blogina programų sistemos palaikomumą ir plečiamumą. Kitaip sakant, naudojami papildomi ištekliai konkrečioje sistemoje kurti jau įgyvendintą funkcionalumą. Tuo tarpu vietoje vieno programinio kodo tenka skirti daugiau resursų palaikyti ir plėsti kelis panašaus funkcionalumo kodo variantus. Šie veiksniai lemia poreikį geriau suprasti bei valdyti (minimizuoti) funkcinių dublikatų skaičių, kai programinio kodo apimtis nuolat didėja. Tam reikalingi atitinkami programiniai įrankiai.

Taigi, darbo problema – kaip galima rasti besidubliuojančius išeities kodo fragmentus.

Praktikoje naudojama išeities kodo palyginimo programinė įranga dažniausia remiasi teksto lyginimo metodais, dėl to yra neatspari trivialiems pakeitimams (pvz. kintamųjų pervadinimui). Tai kelia kitokio pobūdžio metodų poreikį, kurie galėtų atpažinti nors ir skirtingai užrašytą, bet struktūriškai identišką programinį kodą.

Darbo tikslas -– sukurti bei pasiūlyti universalų(-ius) trivialiems pakeitimams atsparų(-ius) programinio kodo palyginimo metodą(-us), kurį(-iuos) būtų galima pritaikyti rasti funkcinius dublikatus. Tikslui pasiekti formuluojami uždaviniai: pasirinkti ir aprašyti medžių palyginimo metodus, tinkamus išeities kodo palyginimui, pasiūlyti galimas metodų optimizacijas, sukurti programinio kodo palyginimo sistemos prototipą ir parodyti praktišką sistemos naudingumą.

Darbe pristatyti du leksiniams pakitimams atsparūs išeities tekstų struktūros palyginimo metodai, pateikiantys santykinį kodo struktūrų panašumo įvertį. Šie metodai paremti pq-gram ir medžių redagavimo atstumo algoritmais. Abu algoritmus galima nesudėtingai pritaikyti bet kokiems duomenims, kuriuos galima išreikšti medžio struktūra. Darbe pristatyti ne vienas, o du metodai dėl iš esmės skirtingų jų savybių — konkretaus siūlomo metodo tinkamumas priklauso nuo programinio kodo apimties, priimtino analizės laiko bei tikslumo poreikių. Praktiškai įgyvendinant palyginimo metodus, suduriama su jų spartos problema: didelių sistemų lyginimas trunka ilgai. Sprendžiant

šią problemą, darbe siūlomos palyginimo metodų optimizacijos ir pristatomas pq-gram metodo patobulinimas, kuris pagreitina miškų lyginimą.

Pq-gram paremtas metodas programinį kodą palygina greitai, tačiau apytiksliai, o antrasis, paremtas medžių modifikavimo atstumu, pateikia tikslų palyginį, tačiau užtrunka ilgesnį laiką. Darbe taip pat pateikti pavyzdinių išeities tekstų palyginimų naudojantis sukurtais metodais rezultatai, praktiškai parodantys minėtų algoritmų specifiškumą.

# 1. Programų panašumas

Idealiu atveju funkciniais dublikatais laikytini kodo fragmentai, kurių vykdymo rezultatai yra vienodi. Teoriškai neįmanoma nustatyti, ar kodas bus apskritai baigtas vykdyti netgi vienam kodo fragmentui. Dėl šios priežasties taip pat neįmanoma nustatyti, ar dviejų kodo fragmentų vykdymo baigtis yra vienoda. Tai lemia ne tokių tikslių panašumo kriterijų pasirinkimą.

Teminėje literatūroje dažniausiai aprašomi kodo struktūros, semantinių arba išdėstymo panašumų aptikimo problematika bei algoritmai. Šie kriterijai labai priklauso nuo kodo analizės programai keliamų tikslų. Pavyzdžiui, plagiatų paieškos sistemos (pvz. MOSS, aprašytoje [SWA03]) dažniausiai taiko metodus, skirtus palyginti tekstus (t.y. aktualūs kodo išdėstymo panašumai). Šį pasirinkimą lemia jų tikslas – rasti plagiatą labai panašaus arba vienodo funkcionalumo išeities kode. Gali būti lyginami ir statistiniai kodo instrukcijų pasikartojimo duomenys, pvz. programos spartos esant tam tikrai techninės įrangos konfigūracijai prognozavimo tikslais (aprašyta [HPE+06]).

Norint rasti programinio kodo funkcinius dublikatus, kodo panašumų paieškai turėtų būti keliami tokie reikalavimai:

1. Atsparumas leksiniams pakeitimams (pvz. kintamųjų pervadinimui)

2. Nejautrumas informacijai, kuri neturi įtakos kodo vykdymui (komentarams bei matomiems tarpams (angl. white space))

Siekiant patenkinti šiuos reikalavimus, pateikiami tokie programinio kodo palyginimo metodai, kurie paremti ne teksto, o abstrakčių kodo sintaksės medžių analizavimu.

# 2. Medžių palyginimo modelis

## 2.1. Medžių redagavimo atstumas

Medžių redagavimo atstumas [ZS89] – tvarkingiems žymėtiems medžiams pritaikytas Damerau-Levenshtein atstumas (pastarasis pasiūlytas [Dam64]). Šis matas tinka, norint įvertinti medžių panašumą. Jis skirtumą tarp medžių išreiškia pagrindinių medžio redagavimo operacijų, kurias reikia atlikti, norint vieną medį transformuoti į kitą, skaičiumi. Autoriai Kaizhong Zhang ir Dennis Shasha apibrėžia tris pagrindines redagavimo operacijas:

1. Trynimas – viršūnė pašalinama, o visi jo vaikai priskiriami viršūnei protėvei.
2. Įterpimas – viršūnė įterpiama (jei reikia, dalis tėvinės viršūnės vaikų priskiriami naujai sukurtai viršūnei).
3. Pakeitimas – viršūnė pervadinama

Redagavimo operacijoms priskyrus skirtingus kaštus ir juos sumuojant vietoje operacijų kiekio, galima gauti tikslesnį skirtumą tarp medžių, jei operacijų tipas yra svarbesnis nei operacijų kiekis. Pavyzdžiu galėtų būti atnaujinamas XML dokumentas (RSS ar įrenginio grąžinami duomenys, OS atnaujinimų sąrašai), kur senų įrašų ištrynimas nėra svarbus.

Šio mato (t.y. medžių redagavimo atstumo) pasirinkimą lėmė jo tinkamumas programinio kodo medžių lyginimui. Jis yra tikslus ir atsparus pakeitimams, kaip parodyta Tobias Sager, Abraham Bernstein ir kt. straipsnyje „Panašių Java klasių radimas naudojantis medžių algoritmus“ [SBPK06]. Autoriai straipsnyje tiria kelių medžių palyginimo algoritmų pritaikymą AST lyginimui:

- Iš apačios generuojamo maksimalaus bendro pomedžio izomorfizmas;
- Iš viršaus generuojamo maksimalaus bendro pomedžio izomorfizmas;
- Medžių redagavimo atstumas.

Autoriai eksperimentiškai parodo, kad medžių redagavimo atstumo algoritmas yra tinkamiausias programinės įrangos išeities kodo palyginimui. Pagrindinė šio tinkamumo priežastis – atsparumas naujų mazgų atsiradimui ar senų mazgų pašalinimui.

## 2.2. Medžių redagavimo atstumo apskaičiavimas

Šiame darbe siūlomame kodo palyginimo metode medžių redagavimo atstumo apskaičiavimui naudojamas metodas, aprašytas Kaizhong Zhang ir Dennis Shasha [ZS89].

$S$ yra redagavimo operacijų seka $s_1, \ldots, s_n$, kurias taikant iš eilės medis $T_1$ transformuojamas į medį $T_2$.

$\gamma$ žymima kainos funkcija (atstumo metrika), funkcijos parametrui priskirianti neneigiamą skaičių. Išvedimo sekos kaina (atstumas) apibrėžiamas kaip $\gamma(S) = \sum_{i=1}^{|S|} \gamma(s_i)$, kur $s_i$ – redagavimo operacija.

Medžių redagavimo atstumas tarp medžių $T_1$ ir $T_2$ apibrėžiamas kaip $min(\gamma(S))$.

Autorių pateikiamas algoritmas paremtas tolesnėmis savybėmis. Atstumas tarp medžių $A$ ir $B$ išreiškiamas kaip

$$treedist(A,B) = min \begin{cases} forestdist(A, B') + \gamma(\emptyset \to B[j]) \\ forestdist(A', B) + \gamma(A[i] \to \emptyset) \\ forestdist(A', B') + \gamma(A[i] \to B[j]) \end{cases}$$

kur $A'$ ir $B'$ – miškai, gauti atitinkamai iš medžio $A$ arba $B$ pašalinus šakinį mazgą.

Atstumas tarp dviejų miškų $A_1..A_n$ ir $B_1..B_m$ , atitinkamai sudarytų iš $n$ medžio $A$ ir $m$ medžio $B$ mazgų, išreiškiamas kaip

$$forestdist(A_1..A_n, B_1..B_m) = min \begin{cases} forestdist(A_1..A_n, B_1..B'_m) + \gamma(\emptyset \to B[j]) \\ forestdist(A_1..A'_n, B_1..B_m) + \gamma(A[i] \to \emptyset) \\ forestdist(A_1..A_{n-1}, B_1..B_{m-1}) + treedist(A_n, B_m) \end{cases}$$

Šie sąryšiai leidžia apskaičiuoti atstumą tarp dviejų medžių rekursiškai skaidant užduotį į vis paprastesnes použduotis.

Kairiausias mazgo $i$ palikuonis žymimas $l(i)$, o aibė, sudaryta iš paties mazgo bei visų jo protėvių žymima $anc(i)$.

Tada jei $i_1 \in anc(i)$ ir $j_1 \in anc(j)$, norint apskaičiuoti $treedist(i_1, j_1)$ reikalingos beveik visos $treedist(i, j)$ reikšmės, kai $i_1$ yra šaknis pomedžio, kuriam priklauso $i$ ir atitinkamai $j_1$ yra šaknis pomedžio, kuriam priklauso $j$. Dėl to visos pomedžių poros skaičiuojamos pagal principą „iš apačios į viršų“.

Kai $i$ yra kelyje tarp $l(i_1)$ ir $i_1$ o $j$ yra kelyje tarp $l(j_1)$ ir $j_1$, nereikia atskirai skaičiuoti $treedist(i, j)$, nes šios reikšmės gaunamos kaip šalutinis $treedist(i_1, j_1)$ skaičiavimo rezultatas.

Apibrėžiama aibė $LR\_keyroots(T) = \{k|\neg\exists k > x, l(k) = l(x)\}$, kurioje elementai (mazgų numeriai) išdėstyti didėjančia tvarka. Kitaip sakant, tai yra rinkinys mazgų, kurie yra šaknys visų pomedžių, kuriuos reikia palyginti atskirai.

Algoritmo principas: pirmiausia kiekvienam mazgui apskaičiuojama f-jos $l()$ reikšmė, tada sudaromos aibės $LR\_keyroots(T_1)$ ir $LR\_keyroots(T_2)$. Tada $\forall(i, j)$, kur $i \in LR\_keyroots(T_1)$ ir $j \in LR\_keyroots(T_2)$ skaičiuojamas atstumas $treedist(i, j)$.

$treedist(i, j)$ skaičiavimui naudojamas dinaminis algoritmas. Apskaičiuotos tarpinės $forestdist$ reikšmės saugomos laikinoje matricoje, kuri išvaloma suskaičiavus atitinkamą $treedist$. Skaičiuojant $treedist(i, j)$ atstumus tarp įvairių pomedžių porų palaipsniui užpildoma $treedist$ matrica, kurios kampinis elementas ir yra atstumas tarp dviejų lyginamų medžių.

Kadangi absoliutų atstumą tarp medžių naudoti nepatogu, metodų palyginimui naudojamas

normalizuotas $treedist$ atstumas tarp medžių $T_1$ ir $T_2$ [SBPK06]. Jis apibrėžiamas taip:

$$dist_{norm}^{treedist}(T_1, T_2) = \frac{treedist(T_1, T_2)}{|T_1| + |T_2|}$$

Šis matas išlaiko nenormalizuoto mato savybes – jis taip pat yra pseudometriškas.

## 2.3. Apytikslis medžių palyginimas

Tikslus medžių palyginimo rezultatas ne visada reikalingas, todėl dažnai yra naudojami apytiksliai medžių palyginimo algoritmai. Pastarieji veikia kur kas greičiau. Taigi jie gali būti naudojami didesnių bei sudėtingesnių medžių palyginimui. Apytikslio medžių palyginimo metodai taip pat gali būti naudojami pradinei didelės apimties duomenų analizei, siekiant atmesti mažai panašius medžius bei sumažinti pradinių duomenų kiekį tiksliam medžių palyginimui.

### 2.3.1. Pq-gram atstumas

Vienas iš būdų apytiksliai palyginti medžius siūlomas mokslininkų Nikolaus Augusten, Michael Böhlen ir Johann Gamper [ABG05], [ABG08]. Metodo taikymui būtinos kelios lyginamų medžių savybės:

1. Viršūnės turi turėti tarpusavio tvarką.

2. Kiekviena viršūnė turi turėti žymę (angl. *label*).

Šie apribojimai rodo, kad galima lyginti tvarkingus žymėtuosius medžius – taip pat kaip ir tiksliojo medžių palyginimo atveju.

Autoriai siūlo naują apytikslio medžių panašumo matą – pq-gram atstumą. Šis atstumas yra medžių redagavimo atstumo, modifikuoto didesnį svorį skirti išsiplėtusioms viršūnėms, aproksimacija. Todėl viršūnių, turinčių daug vaikų, keitimo kaina yra didesnė nei lapų, kurie jų neturi.

Pirmą kartą pq-gram ir pq-gram palyginimo algoritmas pristatyti 31-ojoje tarptautinėje labai didelių duomenų bazių konferencijoje [ABG05]. PQ-gram yra q-gram, skirtų palyginti teksto eilutes, apibendrinimas medžiams.

$T$ yra medis ir $p$ bei $q$ yra du natūralieji skaičiai. PQ-išplėstas medis $T^{pq}$ konstruojamas iš $T$ pridedant $p-1$ protėvį prie centrinės (angl. *anchor*) viršūnės, pridedant $q-1$ vaiką prieš pirmą ir po paskutinio vaiko kiekvienai viršūnei, kuri nėra lapas ir pridedant po $q$ vaikų kiekvienam lapui. Visos pridėtos viršūnės yra netikros: jos neegzistuoja pradiniame medyje $T$.

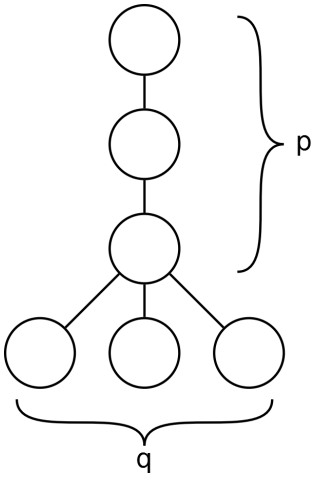
Kai $p$ ir $q$ yra natūralieji skaičiai ir medžio $T$ pq-išplėstasis medis yra $T^{p,q}$, $T^{p,q}$ pomedis yra pq-gram $G$ jei:

1. $G$ turi $q$ lapų ir $p$ viršūnių, kurios nėra lapai.

2. Visi $G$ lapai yra vienos $a \in G$ viršūnės, vadinamos centrine (angl. *anchor*), vaikai.

3. Viršūnės-lapai, priklausantys $G$ yra iš eilės einantys $T^{p,q}$ vienos viršūnės vaikai.

Šablonas, atitinkantis pq-gram formą pavaizduotas 1 pav.




1 pav. pq-gram šablonas

Jei $G$ yra pq-gram su viršūnėmis $V(G) = \{v_1, \ldots, v_p, v_{p+1}, \ldots, v_{p+q}\}$, kur $v_i$ yra i-toji viršūnė VKD (viršūnė, pomedžiai iš kairės į dešinę) tvarka.

Gretinys (angl. *tuple*) $l(G) = (l(v_1), \ldots, l(v_p), l(v_{p+1}), \ldots, l(v_{p+q}))$ vadinamas $G$ žymių rinkiniu.

Medžio pq-gram indeksas $I^{p,q}(T)$ medžiui $T$ apibrėžiamas kaip sąrašas žymių rinkinių $l(G_i)$ kiekvienai pq-gram $G_i$ sugeneruotai iš $T$. Indekse gali kartotis identiški elementai, todėl jis yra multiaibė (angl. *multiset* arba *bag*).

Dviejų medžių panašumas išreiškiamas pq-gram atstumu, kuris parodo skirtingų žymių rinkinių (ir atitinkamai pq-gram) kiekį tarp lyginamųjų medžių. Pq-gram atstumas $dist^{p,q}(T_1, T_2)$ tarp dviejų medžių $T_1$ ir $T_2$ (jų pq-gram indeksai atitinkamai $I_1 = I^{p,q}(T_1)$ ir $I_2 = I^{p,q}(T_2)$) apibrėžiamas [ABG08] taip:

$$dist^{p,q}(T_1, T_2) = |I_1 \cup I_2| - 2|I_1 \cap I_2|$$

Pq-gram atstumas yra pseudometriškas, t.y. nėra neigiamas, lygus 0 identiškiems medžiams ir yra išlaikoma trikampio nelygybė $dist^{p,q}(T_1,T_2) \leq dist^{p,q}(T_1,T_3) + dist^{p,q}(T_3,T_2)$. Jis nėra metriškas, nes skirtingų medžių pq-gram atstumas gali būti lygus 0. Kadangi vienodas pakeitimų kiekis mažiems medžiams gali reikšti didelį skirtumą, o dideliems – nedidelius pasikeitimus, minėtame darbe apibrėžiamas normalizuotas vienetas.

Normalizuotas pq-gram atstumas $dist^{p,q}_{norm}(T_1, T_2)$ tarp medžių $T_1$ ir $T_2$ apibrėžiamas taip:

$$dist^{p,q}_{norm}(T_1, T_2) = \frac{dist^{p,q}(T_1, T_2)}{|I_1 \cup I_2| - |I_1 \cap I_2|}$$

Šis matas išlaiko nenormalizuoto mato savybes – jis taip pat yra pseudometriškas.

Pq-gram algoritmas yra daug greitesnis už medžių redagavimo atstumą. Be to, galima saugoti tarpinius šio algoritmo rezultatus – pq-gram indeksus ir taip išvengti jų generavimo kiekvienam palyginimui. Tai turėtų paspartinti programinių sistemų lyginimą, kur tie patys medžiai lyginami daugelį kartų.

### 2.3.2. Pq-print atstumas

Pq-gram algoritmas leidžia greitai palyginti du medžius. Darbas orientuotas ne į dviejų unikalių medžių, bet į dviejų miškų palyginimą. Tai leidžia optimizuoti pq-gram atstumo skaičiavimo algortimą daugelio miškų lyginimui.

Lyginant daugelį medžių tarpusavyje ypač svarbi algoritmo greitaveika, todėl, taikant pq-gram algoritmą, pq-gram indeksai apskaičiuojami prieš pradedant palyginimus. Lyginimas supaprastėja iki dviejų surūšiuotų pq-gram indeksų palyginimo, kuris sudaro mažą algoritmo veikimo laiko dalį. Indekso įrašą sudaro $p + q$ viršūnių žymės, o pačių indekso elementų yra gerokai daugiau nei viršūnių, todėl indeksas yra gerokai didesnis už patį lyginamąjį medį, ypač jį rašant į rinkmenas. Taip pat, norint įsitikinti, jog du indekso elementai identiški, reikia iš eilės lyginti visus įrašų elementus.

Lyginant miškus, sudarytus iš sąlyginai didelių medžių, siekiant išsiaiškinti, kiek medžiai yra tarpusavyje panašūs, galima pašalinti šiuos pq-gram metodo trūkumus. Dėl didelių indeksų galima prarasti dalį indekso elemento palyginimo tikslumo, prarandant tik mažą dalį indeksų palyginimo tikslumo. Pq-gram algoritmas pagal apibrėžimą yra apytikslis, todėl pavienių kolizijų sukelti pq-print ir pq-gram algoritmų skirtumai nėra kritiniai. Taip pat, lyginant medį su daugeliu kitų medžių, verta minimalizuoti veiksmų, kartojamų kiekvienam palyginimui, kiekį, net jei tai padidins vieną kartą atliekamų veiksmų kiekį. Abu šiuos reikalavimus atitinka maišos (angl. *hash*) funkcijos.

Pq-gram algoritmas modifikuojamas taip, kad indeksuose būtų saugomi ne žymių rinkiniai, o jų kontrolinės sumos, grąžintos maišos funkcijos. Tokiu būdu ženkliai sumažinama indekso užimama vieta bei pagreitinamas elementų tarpusavio palyginimas. Atlikus šį pakeitimą, pagreitės indekso surūšiavimas. Tai sumažins maišos funkcijos vykdymo trukmę pradinio rinkmenos apdorojimo metu. Svarbiausia patobulinimo savybė – indeksų palyginimo pagreitėjimas, nes įrašas palyginamas vienu, o ne $p+q$ veiksmų. Dėl to pagreitėja miškų palyginimas, nes kartojamoje dalyje kelis kartus sumažėja operacijų kiekis. Vis dėlto lyginant du medžius tarpusavyje dėl papildomų veiksmų šis algoritmas veiks lėčiau nei pq-gram.

### 2.3.3. Metodas *Rabin fingerprint*

Pasiūlytas algoritmo patobulinimas bei jo greitaveika labai priklauso nuo pasirinktos maišos funkcijos. Ieškoma funkcija turi šias savybes: greitai apskaičiuojama, kolizijos turi būti retos. Mažą kolizijų kiekį garantuoja beveik visos plačiai naudojamos maišos funkcijos, todėl pagrindiniu pasirinkimo kriterijumi tampa veikimo greitis.

Buvo pasirinkta *Rabin fingerprint* funkcija [Rab81], priklausanti piršto antspaudų klasei. Ji garantuoja mažesnę kolizijų tikimybę nei kontrolinių sumų (angl. *checksum*) klasė. Pirštų antspaudų algoritmai veikia greičiau nei kriptografinės maišos funkcijos, kurios turi papildomas savybes: turint raktą, sunku rasti duomenis, atitinkančius tą raktą, ir sunku rasti įvesties duomenų porą, kurių raktai sutampa. *Rabin fingerprint* algoritmas naudojamas glaudinti informacinio pobūdžio duomenis, todėl kriptografinių maišos algoritmų savybės, galinčios apsaugoti nuo kolizijų sukūrimo, nėra reikalingos, todėl dėl kriptografinių maišos funkcijų lėtumo jų atsisakoma.

# 3. Medžių palyginimo modelio pritaikymas lyginti išeities kodą

## 3.1. Išeities kodo transformavimas į medžio struktūrą

Dėl plataus paplitimo pasirinkta analizuoti Java išeities kodą. Siekiant pritaikyti programinę įrangą, ji suskaidyta į dvi dalis:

1. AST parseris, galintis konvertuoti abstraktų sintaksės medį į XML formatą arba tiesiogiai į lyginimo programos vidinį formatą.

2. Lyginimo programa, lyginanti du XML dokumentus arba Java AST parserio grąžinamus medžius.

Modalinis požiūris į sistemą leidžia ją pritaikyti ir kitų programavimo kalbų išeities kodo lyginimui. Darbe analizuojama Java AST medžių struktūra, nekreipiant dėmesio į jos semantinę prasmę. Tai leidžia tikėtis analogiškų lyginimo rezultatų, lyginant kitų programavimo kalbų duomenis. Šią prielaidą pagrindžia XML formatu pateiktų C programų AST medžių palyginimas.

```
<?xml version="1.1" encoding="UTF-8"?>
<CompilationUnit>
  <TypeDeclaration interface="false">
    <Modifier keyword="public"/>
    <SimpleName identifier="MinimalClass"/>
    <MethodDeclaration constructor="false" extraDimensions="0">
      <Modifier keyword="public"/>
      <Modifier keyword="static"/>
      <PrimitiveType primitiveTypeCode="void"/>
      <SimpleName identifier="main"/>
      <SingleVariableDeclaration varargs="false" extraDimensions="0">
        <ArrayType>
          <SimpleType>
            <SimpleName identifier="String"/>
          </SimpleType>
        </ArrayType>
        <SimpleName identifier="args"/>
      </SingleVariableDeclaration>
      <Block/>
    </MethodDeclaration>
  </TypeDeclaration>
</CompilationUnit>
```

2 pav. XML dokumento pavyzdys

XML rinkmenos apibrėžia hierarchines struktūras, kurias galima interpretuoti kaip medžius. XML dokumentai sudaryti iš mazgų, kurie gali turėti tekstą ar kitų mazgų. Siekiant saugoti ir analizuoti hierarchinę struktūrą – AST medį – pasirenkamas XML formatas. Šiuo pasirinkimu

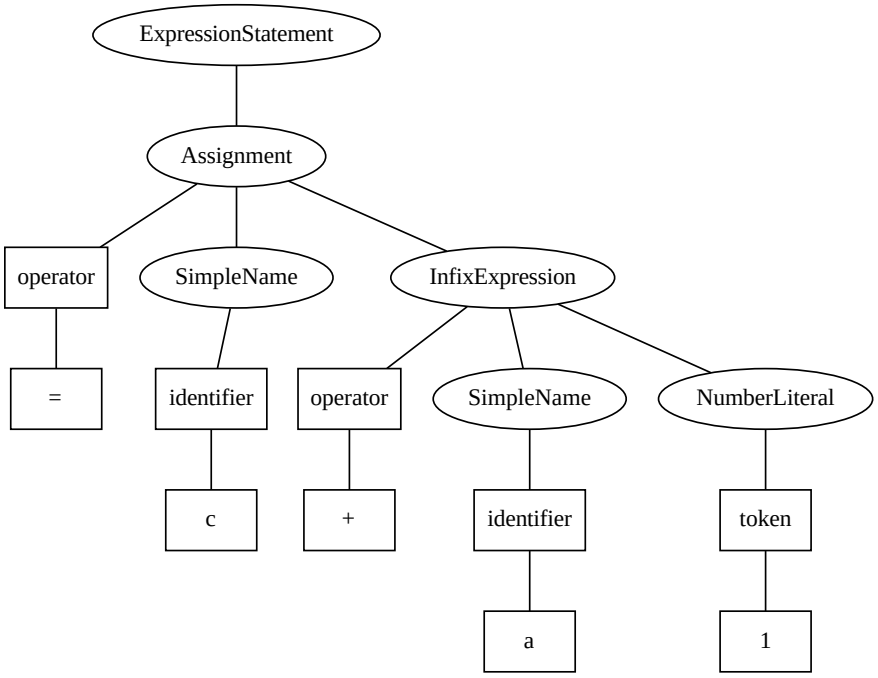
papildomai patikrinami tarpiniai duomenys – iš Java išeities kodo sugeneruoti medžiai. 2 paveikslėlyje pateiktas minimalią Java klasę atitinkantis XML dokumentas.

XML formatas apibrėžia galimybę saugoti informaciją taip pat ir mazgo atributuose. Tai sukelia papildomų sunkumų, nes susidaro dvilypė hierarchija – mazgai bei jų atributai, kurie paklūsta kiek kitokioms taisyklėms. Atributai sukeliami į mazgų medį kaip papildomi mazgai, pridedant juos į medį prieš tikruosius mazgus. Taip pasiekiamas atributo-mazgo ekvivalentumas, kuris panaudojamas kai kuriuose XML paremtuose dokumentų formatuose, pvz. *spring beans*. Šalutinis šio pakeitimo efektas – geresni rezultatai lyginant panašius XML dokumentus, kuriuose įvykdytas atributų pakeitimas mazgais.

AST parseris atlieka pradinį Java išeities kodo apdorojimą. Programa sugeneruoja AST medį iš duoto išeities kodo ir sugeneruoja vidinę reprezentaciją, atlikdama kelias transformacijas. Siekiant panaudoti egzistuojančius įrankius, AST parseris sukurtas naudojantis Eclipse JDT API.

Parseris turi tris darbo režimus:

1. Pilną (FULL) – kai į analizuojamą medį pridedami atributai atvaizduojami dviem viršūnėmis. Šiuo atveju kiekvienam atributui generuojami du mazgai: atributo pavadinimo, kuriam sukuriamas vaikas, ir atributo reikšmės. FULL režimo darbo rezultatas pavaizduotas 3 pav.




3 pav. Pilno parserio režimo rezultatas

2. Nepilną (BRIEF) – į analizuojamą medį pridedamos tik atributų reikšmės. Žinant mazgo tipą, žinomi ir jam priskiriami atributai, todėl saugoti atributų pavadinimus nėra tikslinga. AST medžių XML eksportas parseriui veikiant šiuo darbo režimu nepalaikomas dėl XML standarto apribojimų mazgų pavadinimams. BRIEF režimo darbo rezultatas pavaizduotas pirmame priede, 12 pav.

3. Struktūros (STRUCTURE) – atributai praleidžiami. Šiuo atveju išsaugoma tik bendra Java AST medžio struktūra, bet prarandama informacija apie detales – kintamųjų ir konstantų pavadinimai bei reikšmės, kviečiamų metodų pavadinimus bei atliekamas operacijas. Būtent šis metodas naudojamas Tobias Sager, Abraham Bernstein ir kitų straipsnyje „Panašių Java klasių radimas naudojantis medžių algoritmais“ [SBPK06]. STRUCTURE režimo darbo rezultatas pavaizduotas pirmame priede, 13 pav.

| | | Pakeista konstanta | Pervadintas kintamasis | Įterpta instrukcija | Sukeista instrukcijų tvarka | Iškeltas metodas | Greitasis rūšiavimas |
|---|---|---|---|---|---|---|---|
| Pilnas | Viršūnės | 456 | 456 | 441 | 456 | 440 | 622 |
| | Skirtumas | 1 | 9 | 16 | 6 | 98 | 270 |
| | Norm. skirt. | 0.1096% | 0.9868% | 1.7837% | 0.6578% | 10.9375% | 25.0463% |
| Nepilnas | Viršūnės | 322 | 322 | 311 | 322 | 310 | 435 |
| | Skirtumas | 1 | 9 | 12 | 4 | 74 | 208 |
| | Norm. skirt. | 0.1552% | 1.3975% | 1.8957% | 0.6211% | 11.7088% | 27.4768% |
| Struktūros | Viršūnės | 188 | 188 | 181 | 188 | 180 | 248 |
| | Skirtumas | 0 | 0 | 7 | 2 | 41 | 107 |
| | Norm. skirt. | 0% | 0% | 1.8970% | 0.5319% | 11.1413% | 24.5412% |

1 lentelė. Burbulo rūšiavimo metodo ir jo variacijų palyginimo rezultatai

Lentelėje nr. 1 pateikiami visų trijų režimų veikimo rezultatai, naudojant medžių redagavimo atstumą. Medžių redagavimo atstumo metodu lyginami nedideli kodo pakeitimai burbulo rūšiavimo programoje. Originalios programos viršūnių kiekis sutampa su pakeistos konstantos pavyzdžiu. Ši programa taip pat palyginama su greitojo rūšiavimo programa, kurioje skiriasi tik rūšiavimo metodas.

Lentelė nr. 1 parodomi visų trijų metodų privalumai bei trūkumai. Naudojant pilną (FULL) konvertavimą į lyginamąjį medį, gaunami didžiausi medžiai. Tai gali kelti problemų, analizuojant didesnės apimties išeities kodą. Struktūrinis (STRUCTURE) metodas grąžina daugiau nei dvigubai mažesnius medžius nei pilnas (FULL) metodas, tačiau šis žymus sumažinimas pasiekiamas, prarandant dalį AST medžio duomenų, todėl suprastėja palyginimo kokybė – pakeistos konstantos ir pervadinto kintamojo atvejai nebeišskiriami kaip besiskiriantys. Siekiant surasti kompromisą tarp šių dviejų polių, sukurtas nepilnas (BRIEF) filtravimo metodas, kuris leidžia prarasti tik labai nežymią informacijos dalį. Tai pasiekiama, iš medžio pašalinant atributų pavadinimų viršūnes. Informacija neturėtų būti prarandama, nes pagal tėvinės viršūnės tipą galima spręsti, kokius atributus ji turi. BRIEF metodas pašalina pusę struktūrinio metodo pašalinamų viršūnių. Eksperimento rezultatai parodo, kad jis rado visus skirtumus, rastus pilno metodo. Dėl sumažėjusio medžių dydžio skirtumų, kurie abiejuose medžiuose atsispindi vienodu mazgų kiekiu, nepilno (BRIEF) filtravimo metodo atveju atitinka didesnis normalizuotas skirtumas. Eksperimento rezultatai atskleidžia pagrindinį nepilno (BRIEF) metodo privalumą – nors viršūnių kiekis medyje yra ženkliai sumažintas,

išlaikoma palyginimo kokybė, nes pašalintos viršūnės savyje neneša informacijos Java AST medžių atveju. Šio teiginio negalima išplėsti kitiems formatams, nes kituose formatuose jis gali būti neteisingas. Pavyzdžiui, XML formate mazgo atributai neturi griežtai apibrėžtos tvarkos, todėl atributo vardas saugo struktūrinę informaciją.

## 3.2. Eksperimentinis p ir q parinkimas

Atliekant apytikslį medžių palyginimą, susiduriama su problema: reikia parinkti p ir q reikšmes, kurios tiktų palyginti programinės įrangos kodą. Mažos reikšmės gali saugoti per mažai konteksto panašumo palyginimui. Tuo tarpu didelės reikšmės reiškia saugomų ir lyginamų duomenų kiekio didėjimą.

### 3.2.1. Lyginamų duomenų kiekio priklausomybė nuo p ir q

Pq-gram algoritmo lyginamasis vienetas yra žymių rinkinys, kuriame yra $p + q$ žymių. Taigi, didėjant p ir q, tiesiškai auga ir įrašo dydis. Tačiau nuo p ir q priklauso ne tik lyginamojo elemento dydis.

Jei $q = 1$, tai yra lyginamas su $p + 1$ ilgio linijiniu grafu, lyginamųjų elementų kiekis lygus medžio lapų ir briaunų sumai. Medžio šaknis turi $p - 1$ protėvį, todėl ji gali būti centrine viršūne. Tokių pq-gram kiekis yra lygus jos vaikų kiekiui, nes $q = 1$ – ne lapams vaikai nepridedami. Šis teiginys teisingas kiekvienai viršūnei, kuri nėra lapas. Kiekvienam lapui pridedama po $q = 1$ vaikų, todėl kiekvieną lapą atitinka viena pq-gram. Kai $q = 1$, N yra viršūnių skaičius, o $N_l$ – lapų skaičius medyje, medžio indekso elementų kiekis yra:

$$|I^{p,q}(T)| = N_l + N - 1$$

Vienetas atimamas, nes medžio šaknis niekada nėra pq-gram vaikas.

Jei $p = 1$, lyginama su grafu, kuriame viena viršūnė turi $q$ vaikų. Kiekvienai medžio viršūnei $a \in T$, kuri nėra lapas, tampant centrine, iš originalaus medžio $T$ sukuriamų pq-gram, kurių pirmas elementas yra viršūnės $a$ vaikas, kiekis lygus viršūnės vaikų kiekiui. Be to, dėl $q - 1$ pridėtinės viršūnės išplėstame medyje $T^{pq}$, papildomai sukuriama $q - 1$ pq-gram, kurios pirmasis elementas – menama viršūnė. Medyje yra $N - -N_{lap}$ tokių viršūnių. Likusi lygties dalis yra identiška $q = 1$ atvejui, nes po viršūnės vaikų taip pat pridedamos $q - 1$ menamos viršūnės.

Gaunama tokia lygtis:

$$|I^{p,q}(T)| = (q - 1) * (N - N_{lap}) + N_{lap} + N - 1$$

Nors tarp medžio elementų kiekio ir indekso dydžio egzistuoja tiesinė priklausomybė, parametras q lemia daugiklį. Tuo tarpu parametras p įtakoja tik pq-gram indekso elemento dydį.

### 3.2.2. Eksperimentinis p ir q parinkimas

Siekiant parinkti standartinius p ir q dydžius, kurie naudojami programų išeities kodo palyginimui, lyginami keli paprasti atvejai, t.y. lyginami šie pakeitimai pavyzdinėje burbulo rūšiavimo programoje:

1. Konstantos pakeitimas.
2. Kintamojo pervadinimas.
3. Naujos instrukcijos pridėjimas.
4. Instrukcijų vykdymo tvarkos pakeitimas.
5. Metodo iškėlimas.

Lyginimui naudojamas pq-gram algoritmas su skirtingomis p ir q reikšmėmis. Parseris naudoja nepilną režimą, kuris generuoja mažesnius medžius.

Siekiant įrodyti parinktų parametrų tinkamumą, testas kartojamas lyginant pavyzdines programas:

1. Burbulo rūšiavimo (*bubble sort*) pavyzdys.
2. Greitojo rūšiavimo (*quick sort*) pavyzdys.

Siekiant automatizuoti eksperimentinių duomenų rinkimą, parašyti skriptai, renkantys ir apdorojantys duomenis. Kiekvienam parametrų rinkiniui (p reikšmė, q reikšmė, programų pora) programa leidžiama po 10 kartų siekiant gauti pakankamai tikslų vykdymo trukmės vidurkį.

Eksperimentas parodo, kad vykdymo laiko skirtumas tarp $p = q = 1$ ir $p = q = 10$ analizuotiems pavyzdžiams yra mažesnis nei 5%, todėl vykdymo laikas didelės svarbos p ir q parinkimui neturi. Eksperimento duomenys pateikti priede Nr. 2.

Rezultatai rodo, kad testiniams duomenims (lentelės 4, 5, 6, 7, 8) p pasikeitimas visiškai neturėjo įtakos, tuo tarpu realiame palyginime (pav. 9) didėjant p skirtumas tarp programų didėjo. Tai paaiškinti galima tuo, kad testinėse programose pakeitimai buvo vykdyti medžių lapuose arba arti jų, o realioje programoje keitėsi ir viršūnės, esančios arti medžio šaknies. Tai lėmė vis didesnio pasikeitusių pq-gram kiekio radimą didėjant p. Siekiant apriboti skirtumo augimą tikslinga parinkti tokią p reikšmę, kad būtų lyginamos lokalios viršūnės, o pakeitimai arti šaknies neiškreiptų palyginimo rezultatų. Šie pastebėjimai kartu su AST medžio analize lemia standartinės p reikšmės parinkimą: $p = 3$.

Tuo tarpu q pasikeitimas visais atvejais įtakojo rezultatų keitimąsi. Augant q didėjo ir normalizuotas pq-gram atstumas. Tai lemia $q - 1$ menamų mazgų pridėjimas visiems lapams, taigi ir $q - 1$ pq-gram atsiradimas. Išskirtinis $q = 1$ atvejis pateikiamas 7 lentelėje, kai dėl išsišakojimo nebuvimo nerandama instrukcijų tvarkos sukeitimo. Taigi, siekiant rasti kaimyninių viršūnių pasikeitimą, nesukeliant atstumo augimo, parenkama ir $q = 3$.

# 4. Eksperimentinis įgyvendinimo vertinimas

## 4.1. Eksperimento duomenys (aplinka)

Programinio kodo palyginimo metodų veikimui įvertinti pasirinkta *args4j* biblioteka, skirta apdoroti komandinės eilutės parametrus Java programose. Projekto interneto puslapis – `http://args4j.kohsuke.org`, išeities kodas – `https://github.com/kohsuke/args4j`. Palyginimams bus naudojamos 2.0.10 ir 2.0.16 bibliotekos versijos.

Biblioteka palyginimui pasirinkta dėl savo dydžio – atmetus testus, jos dydis yra 40-50 Java klasių. Toks kiekis yra pakankamas, norint įvertinti palyginimo įrankių darbo rezultatus, bet ne toks didelis, kad eksperimento laikas taptų nepraktiškas.

## 4.2. Eksperimentų metodika

Prieš aprašant konkrečius eksperimentus, trumpai apie bendrą eksperimentų kūrimo metodiką. Visuose eksperimentuose lyginamos dvi *args4j* bibliotekos versijos. Siekiant lyginti prasmingas išeities kodo dalis, testai nelyginami. Visos vienos versijos rinkmenos lyginamos su visomis kitos versijos rinkmenomis. Visi palyginimo įrankiai grąžina normalizuotus rezultatus intervale $[0; 1]$. Šioje skalėje 0 reiškia identiškumą, 1 – visišką skirtumą.

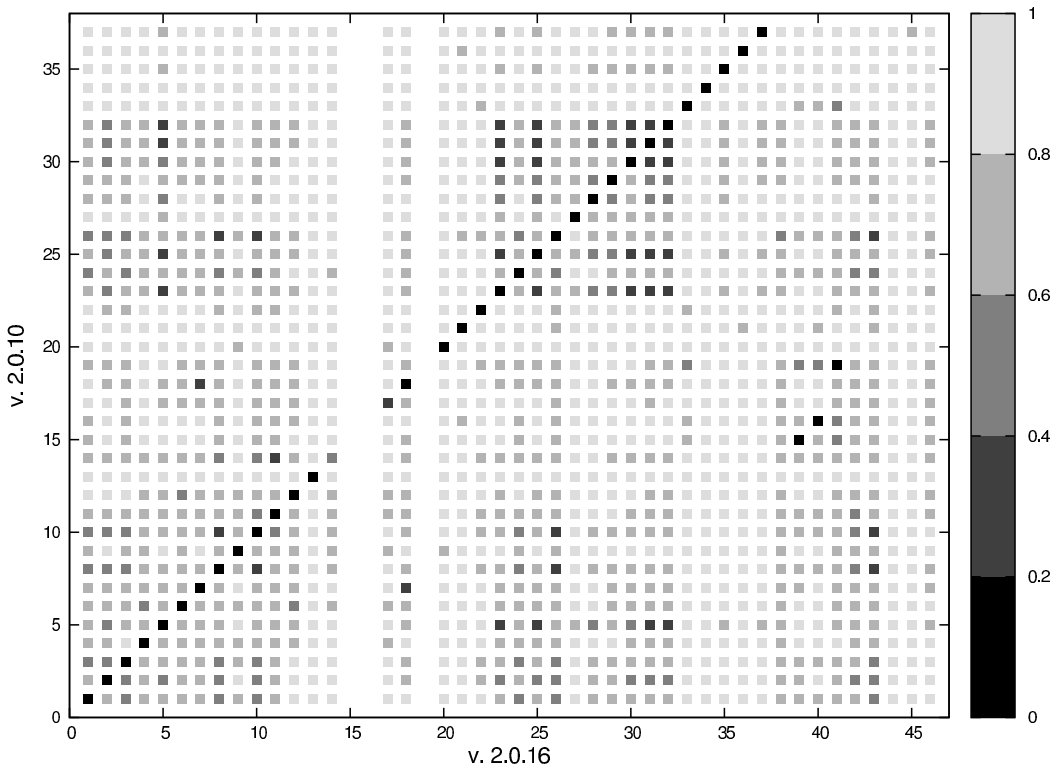
Siekiant rezultatus pavaizduoti grafiškai, dėl aiškumo pasirinkta Java klasių (ir rinkmenų) pavadinimams priskirti natūraliuosius skaičius. Lygintų klasių rodyklė pateikiama priede Nr. 3. Rodyklė sudaryta klasių naujumo principu: sunumeruotos 2.0.3 versijos klasės, tuomet numeruotos klasės, atsiradusios iki 2.0.10 versijos ir galiausiai – naujos 2.0.16 versijos klasės. Lentelėse galima pastebėti, kad *args4j* 2.0.16 versijoje pašalintos 3 klasės (15, 16 ir 19 numeriai rodyklėje), bei pridėtos 9 klasės (38-46 rodyklėje).

Jei nenurodyta kitaip, palyginimui pq-gram metodu naudojamos $p = 3$ ir $q = 3$ parametrų rekšmės.

## 4.3. Kontrolinis palyginimo įrankis

Siekiant įvertinti rezultatus, reikalingas nepriklausomas palyginimo įrankis. De-facto standartinis įrankis programinio kodo palyginimui yra *diff*, kuris randamas kiekvienoje versijų kontrolės sitemoje. Šis įrankis grąžina besiskiriančias rinkmenų eilutes vienu iš *diff* formatų. Nors įrankis turi daugybę skirtingų implementacijų, visos jos atlieka tą pačią funkciją. Eksperimentams naudojama GNU *diff* atmaina, randama daugumoje Linux distribucijų.

Norint gauti normalizuotą palyginimo rezultatą, naudojamas skriptas, kuris standartinių UNIX įrankių pagalba apdoroja *diff* grąžinamus rezultatus. Iš unifikuoto *diff* formato pašalinamos eilutės, nerodančios skirtumų tarp rinkmenų. Likusios eilutės suskaičiuojamos ir, rezultatą padalinus iš lyginamųjų rinkmenų eilučių kiekio sumos, gaunamas santykinis rinkmenų skirtumas. Kontrolinio palyginimo rezultatas pateikiamas pav. 4.




4 pav. *args4j* v2.0.10 ir v2.0.16 palyginimo rezultatai, naudojant *diff*

## 4.4. Parserio darbo režimų įtaka palyginimo rezultatams

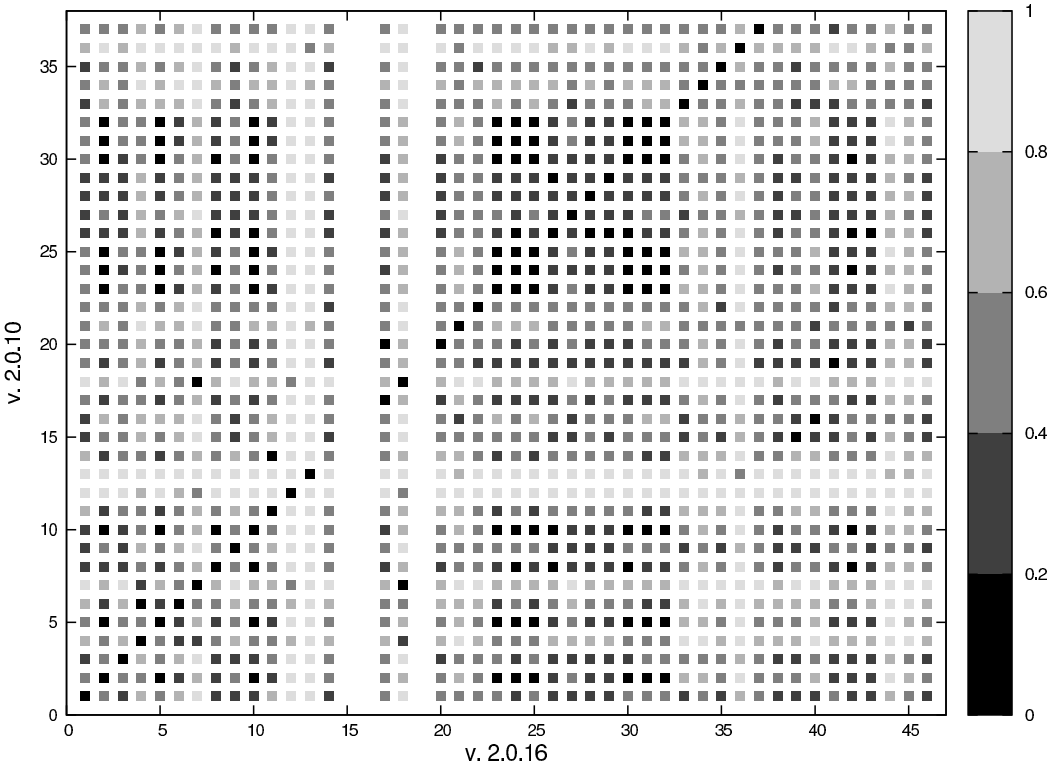
Siekiant rasti geriausią metodą, skirtą atpažinti programinės įrangos išeities kodo panašumus, būtina įvertinti parserio darbo režimų įtaką palyginimo rezultatams.

Palyginimui naudojami abu algoritmai, nes jų darbo rezultatai ženkliai skiriasi ir reikia įsitikinti, ar vienas režimas grąžina geriausius rezultatus visiems algoritmams. Palyginimo rezultatai naudojant medžių redagavimo atstumo algoritmą pateikti šiuose grafikuose: 5, 7, 6. Pq-gram algoritmo rezultatai pateikti šiuose grafikuose: 8, 9, 10.

| Algoritmas | Režimas | Vidutinis elementų kiekis | | Vidutinis skirtumas | Vidutinis skirtumas % |
|---|---|---|---|---|---|
| | | v. 2.0.10 | v. 2.0.16 | | |
| Redagavimo atstumas | Pilnas | 534.9 | 562.8 | 698.7 | 52.154% |
| | Nepilnas | 381.6 | 401.1 | 514.7 | 54.526% |
| | Struktūrinis | 228.3 | 239.4 | 299.2 | 52.075% |
| pq-gram | Pilnas | 1446.0 | 1520.6 | 2195.0 | 76.043% |
| | Nepilnas | 986.1 | 1035.5 | 1582.6 | 79.736% |
| | Struktūrinis | 566.6 | 593.7 | 863.6 | 75.700% |
| pq-print | Pilnas | 1446.0 | 1520.6 | 2194.9 | 76.041% |
| | Nepilnas | 986.1 | 1035.5 | 1582.5 | 79.734% |
| | Struktūrinis | 566.6 | 593.7 | 863.6 | 75.700% |
| Diff | – | 67.0 | 69.7 | 115.0 | 77.412% |

2 lentelė. Parserio režimų įtaka lyginamų objektų kiekiui ir palyginimo rezultatams

2 lentelėje pateikti elementų kiekių skirtumai atitinka aprašytus 3.1 skyriuje. Dėl skirtingo lyginamųjų elementų kiekio neišeina tiesiogiai lyginti skirtumų, bet galima lyginti normalizuotus vidutinius skirtumus, analizuojant kiekvieną algoritmą atskirai. Pastebėtina, kad pilno bei struktūrinio parserio veikimo režimų rezultatai panašūs. Pastarasis režimas randa truputį daugiau panašumų, kai tuo tarpu nepilnas režimas randa kiek daugiau skirtumų. Ši situacija susidaro todėl, kad, lyginant su pilnu darbo režimu, nepilnas didesnį svorį priskiria kintamųjų, metodų bei klasių pavadinimams, pagrindinėms aritmetinėms operacijoms, konstantoms bei metodų iškvietimams, o struktūrinis lygina tik kodo struktūrą – visų anksčiau paminėtų objektų pavadinimai ir / ar reikšmės nėra lyginami.



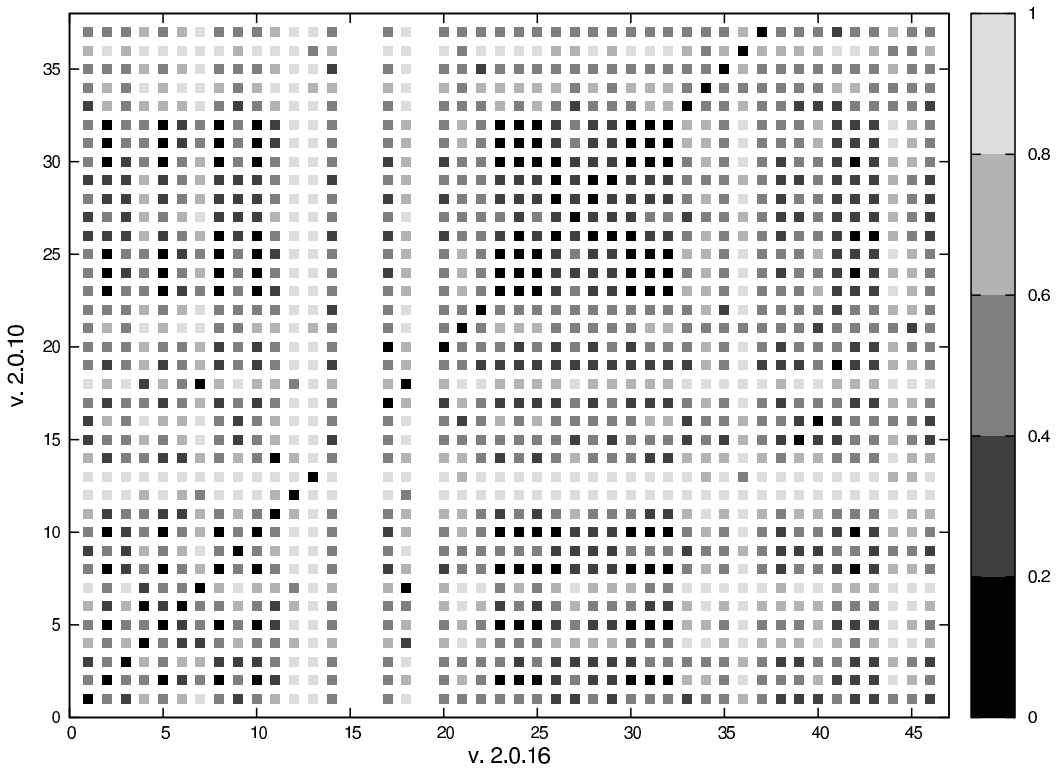

5 pav. *args4j* v2.0.10 ir v2.0.16 palyginimo rezultatai, naudojant medžių redagavimo atstumo algoritmą ir FULL parserio režimą.

Nors algoritmų grąžinami rezultatai ženkliai skiriasi, parserio darbo režimų įtaka normalizuotiems skirtumams yra labai maža. Normalizuotų skirtumų normalizuoto skirtumo vidurkiai skirtingiems algoritmams skiriasi mažiau nei 0,4%, lyginant pilną ir nepilną režimus, mažiau nei 0,2%, lyginant pilną ir struktūrinį. Šis skaičius buvo gautas, naudojant formulę

$$dist_{norm}^{full:mode} = \frac{|dist_{norm}^{full} - dist_{norm}^{mode}|}{dist_{norm}^{full}}$$

kur $dist_{norm}^{full}$ – normalizuotas skirtumas, gautas naudojant pilną parserio režimą, o $dist_{norm}^{mode}$ – normalizuotas skirtumas, gautas naudojant analizuojamą parserio režimą.

Struktūrinis režimas gerai tinka, kai ieškoma kodo struktūros dublikatų – klasių, turinčių pa-

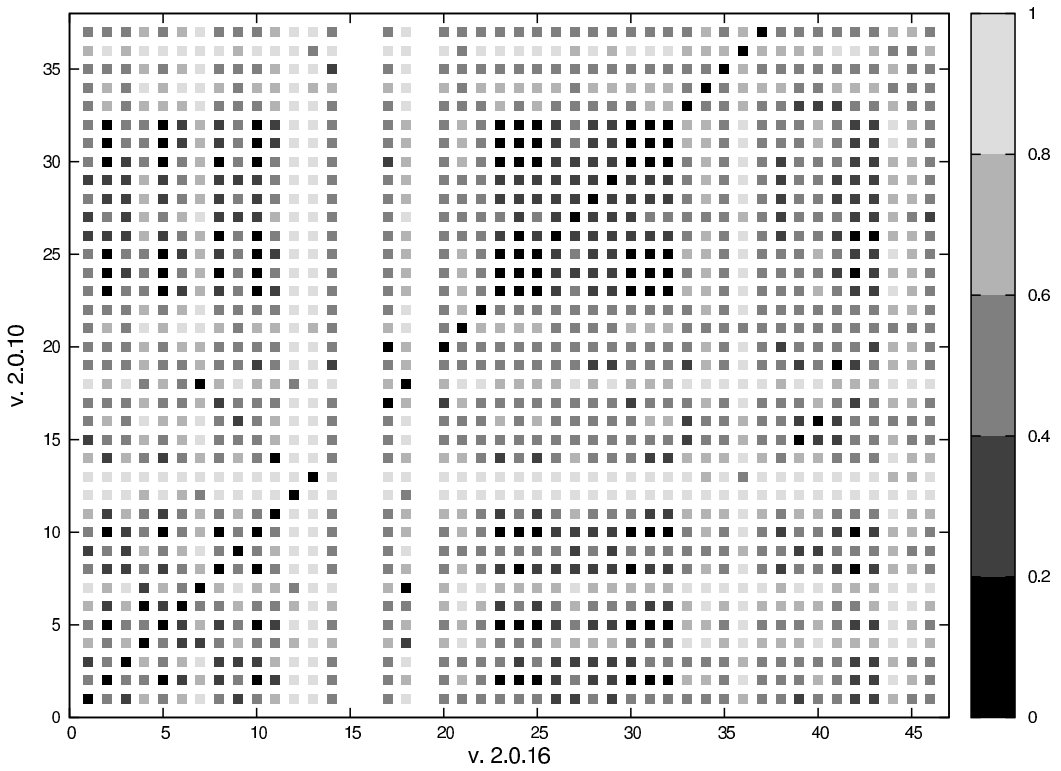



6 pav. *args4j* v2.0.10 ir v2.0.16 palyginimo rezultatai, naudojant medžių redagavimo atstumo algoritmą ir STRUCTURE parserio režimą.

našų metodų kiekį, kuriuose kodas išdėstytas panašiai. Pagrindinis šio metodo pranašumas – jis gerokai greitesnis už kitus, nes tarpusavyje lyginama 2-3 kartus mažiau elementų. Šį režimą galima taikyti, siekiant rasti vienodo funkcionalumo klases, jei šis functionalumas yra standartinis (t.y. beveik identiškas), tačiau kurtas ir plėtotas atskirai ir todėl kintamųjų bei metodų pavadinimai skiriasi. Šis režimas praranda daug semantinės informacijos, todėl, naudojant santykinį skirtumą kaip panašummo kriterijų, reikia nusibrėžti slenkstį tarp panašių ir nepanašių klasių žemiau nei pilno darbo režimo atveju. Net pasirinkus santykinai žemą panašumo slenkstį, atsiranda didelė klaidingai teigiamų (angl. *false positive*) rezultatų tikimybė, nes daug skirtingų operacijų generuoja identiškus sintaksės medžius. Pavyzdžiui `x = a + 1;` bei `circumference = diameter * 3.1415;` struktūros režimu sugeneruoti medžiai bus identiški.

Nepilnas režimas pasižymi priešingomis savybėmis: lyginant su pilnu režimu jis didesnį svorį suteikia semantinei kodo daliai – klasių, metodų ir kintamųjų pavadinimams, konstantų reikšmėms ir operacijoms. Tai pasiekiama iš medžio pašalinant atributų pavadinimus atitinkančias viršūnes, kurios sudaro apie 1/3 medžio viršūnių. Algoritmų darbas pagreitėja, sumažinus lyginamų medžių apimtis. AST medį apdorojus šiuo režimu, palyginimo metodų veikimo greitis turėtų būti tarp pilno ir struktūrinio režimo. Žvelgiant iš praktinės pusės, šis algoritmas yra pranašus tuo, kad jis išskiria semantinius skirtumus – būtent tai, kas domina sistemose, ieškant dublikatų ir kopijuotų programinio kodo fragmentų.

Apibendrinus gautus rezultatus, galima teigti, jog parserio darbo režimo pasirinkimą lemia



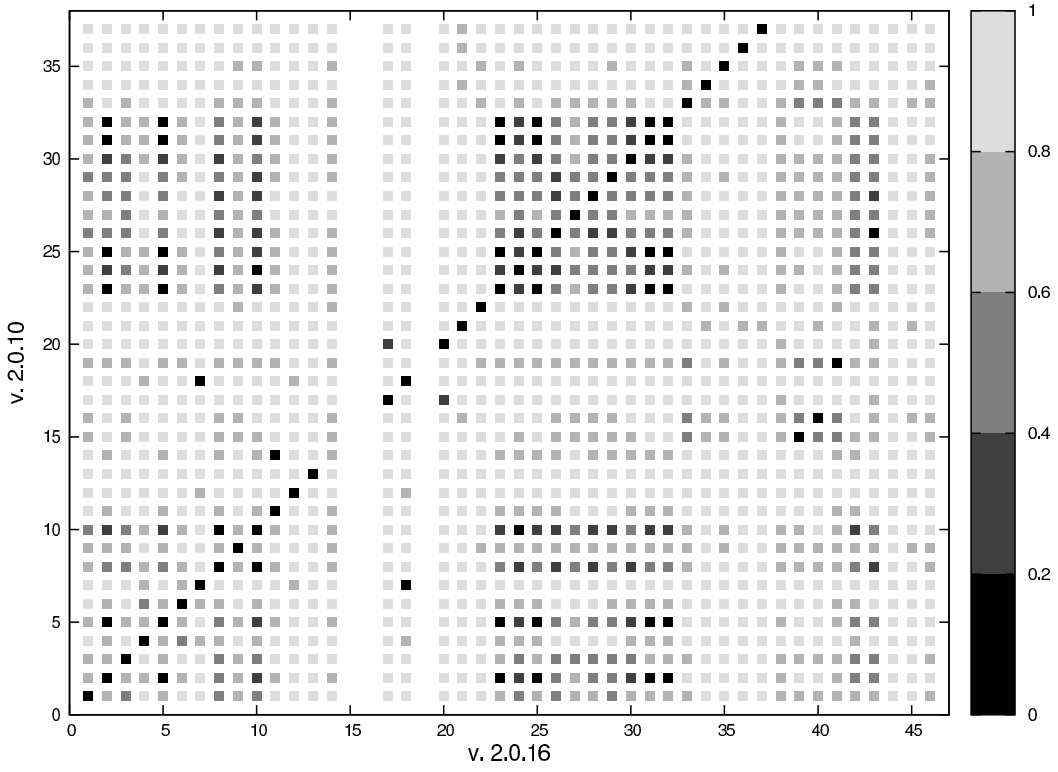

7 pav. *args4j* v2.0.10 ir v2.0.16 palyginimo rezultatai, naudojant medžių redagavimo atstumo algoritmą ir BRIEF parserio režimą.

sprendžiamos problemos pobūdis. Jei ieškoma identiškų kodo fragmentų, verta pasirinkti nepilną parserio režimą. Jei ieškoma programinio kodo, turinčio panašią struktūrą, pavyzdžiui siekiant apibendrinti panašias klases, verta pasirinkti struktūrinį režimą. Naudoti pilną parserio veikimo režimą nerekomenduojama, nes kiti režimai veikia daug greičiau, ir grąžina tikslesnius rezultatus. Jis naudojamas tik kaip neutralus palyginimo taškas sudėtingesniems palyginimo režimams – tiksliausiai atitinka Java AST medį.

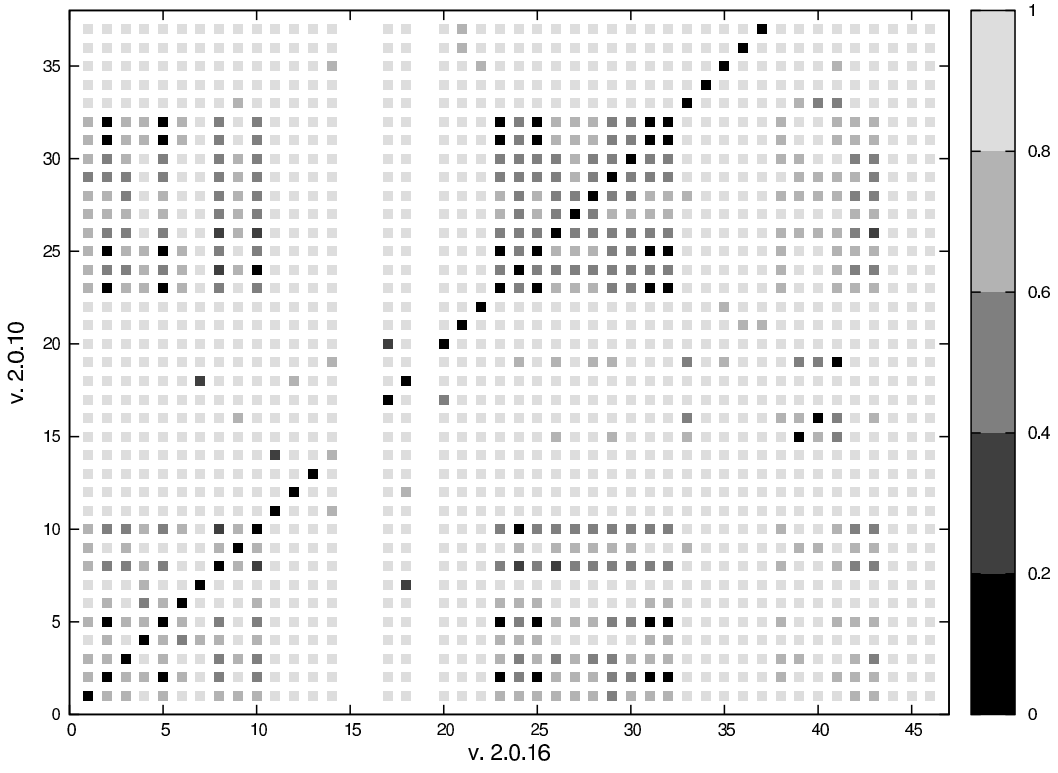
## 4.5. Išeities kodo palyginimo rezultatai

Eksperimento metu pq-gram ir medžio redagavimo atstumo metodais palygintos dvi *args4j* versijos. Palyginimo rezultatai pateikti šiuose grafikuose: medžių redagavimo atstumo – 5, 7 ir 6 pav., pq-gram ir pq-print – 8, 9 ir 10 pav. Kontrolinio metodo, paremto *diff*, rezultatai pateikiami 4 grafike.

Visi grafikai atspindi kelis bendrus dviejų *args4j* bibliotekos versijų bruožus: klasės buvo mažai keistos (beveik identiškų klasių įstrižainė), akivazidžios pašalintos (tušti 15, 16 ir 19 stulpeliai), bei pridėtos (38-46 stulpeliai) klasės. Taip pat visuose grafikuose išsiskiria grupė panašių klasių, išplečiančių bendrinę (angl. *generic*) *OptionHandler* klasę skirtingiems parametrų tipams. Grupę sudaro šios klasės: 1, 2, 3, 5, 8, 9, 10, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 42, 43. Šiuo požiūriu visi grafikai yra panašūs.

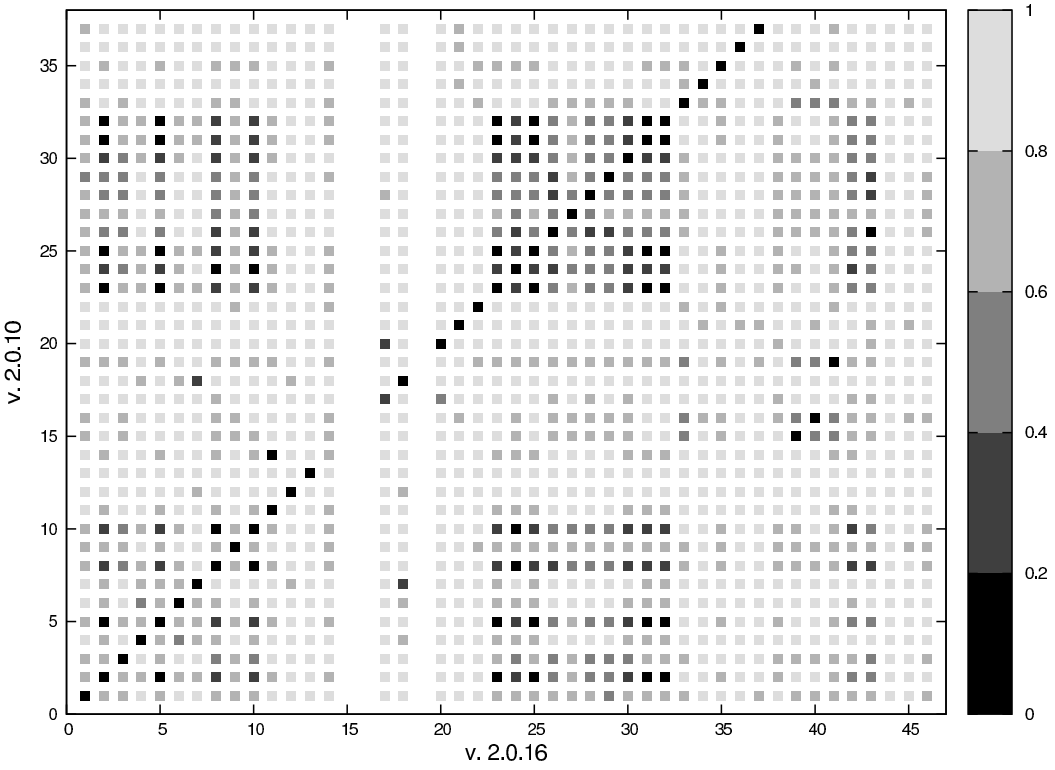



8 pav. *args4j* v2.0.10 ir v2.0.16 palyginimo rezultatai, naudojant pq-gram algoritmą ir FULL parserio režimą.




9 pav. *args4j* v2.0.10 ir v2.0.16 palyginimo rezultatai, naudojant pq-gram algoritmą ir BRIEF parserio režimą.

Lyginamuose rezultatų grafikuose matyti, kad medžių palyginimo metodais paremti algoritmai aptiko daugiau panašių klasių, lyginant su *diff* metodu paremtu algoritmu. Iš dalies tai paaiškintina tuo, kad *diff* lygina išeities kodo eilutes, tuo tarpu analizuojamieji metodai lygina pačius Java kodą atitinkančius AST medžius. Dėl šių ypatybių *diff* taip pat lygina kodo formatavimą, komentarus, *JavaDoc* įrašus ir tarpus (angl. *whitespace*) – programinio kodo vykdymui įtakos neturinčią informaciją. Kitą rezultatų skirtumo dalį sudaro elementų smulkumas – pasikeitus nors vienam elementui eilutėje, visa eilutė žymima kaip besiskirianti, tuo tarpu AST medžius analizuojantys metodai randa, kad pasikeitė tik dalis elementų – AST medžių palyginimo metodai daug tiksliau įvertina pasikeitimų dydį.




10 pav. *args4j* v2.0.10 ir v2.0.16 palyginimo rezultatai, naudojant pq-gram algoritmą ir STRUCTURE parserio režimą.

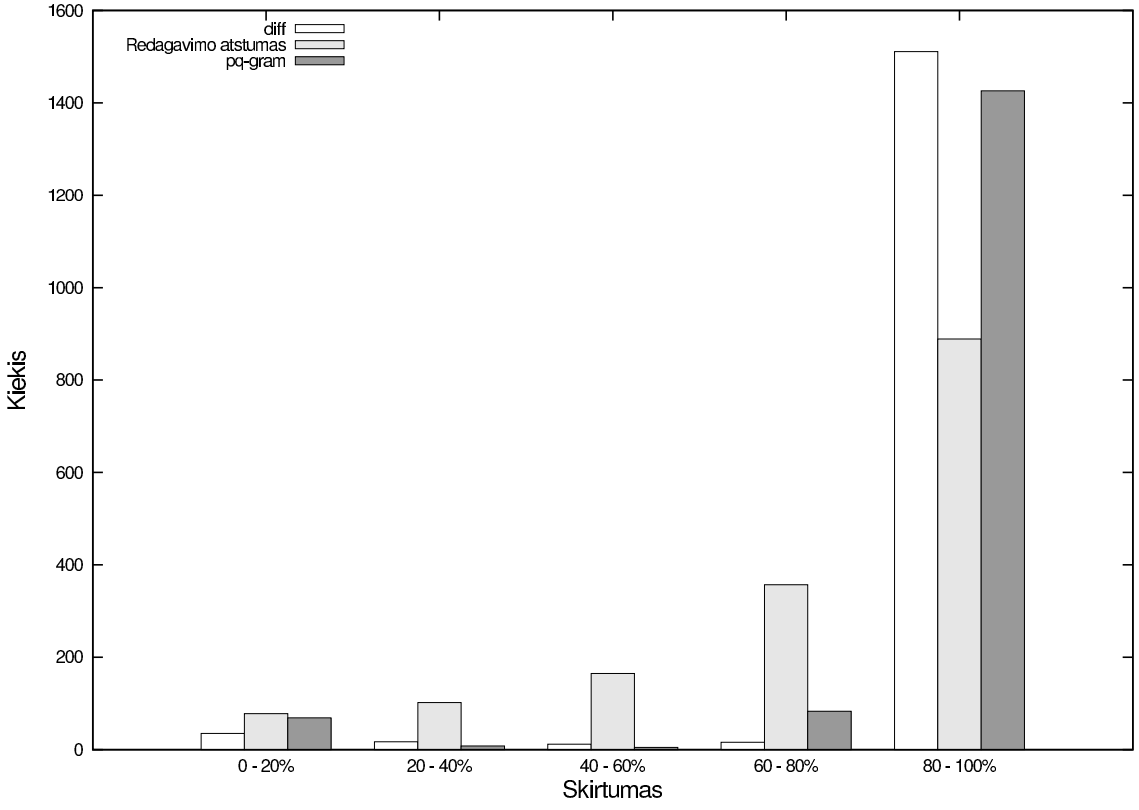
Taip pat ryškus ir pq-gram bei medžių redagavimo atstumo palyginimo rezultatų skirtumas. Pritaikius pq-gram medžių palyginimo algoritmą, gaunama daug mažiau panašių klasių. Pq-print reikšmių pasiskirstymas iš esmės yra identiškas pq-gram, nepaisant poros kolizijų. Dėl to abiejų algoritmų rezultatų grafikai sutampa ir pq-print algoritmo rezultatų grafikai nepateikiami.

Pq-gram ir diff rezultatų grafikai tarpusavyje panašūs: dauguma rinkmenų įvertintos kaip labai besiskiriančios, bet pq-gram randa daugiau panašumų, kurie visai neatspindimi diff rezultatuose (pavyzdžiui 17-to *Argument* ir 20-to *Option* interfeisų, kurie aprašo būtinus ir nebūtinus argumentus ir yra labai panašūs). Diff randa daug mažiau panašumų ir *OptionHandler* klasę išplečiančių klasių grupėje, kur daugeliui klasių randamas tik mažas panašumas, tuo tarpu pq-gram metodas jas pažymi kaip labai panašias, nepaisant pasirinkto parserio darbo režimo.

Medžių redagavimo atstumo metodas daugelį klasių vertina kaip panašias, todėl, norint naudoti šį metodą rasti klasių bei kodo dublikatams, norint pažymėti klases panašiomis, reikia gauti daug mažesnį skirtumą nei pq-gram metodo atveju. Tai turėtų padėti išvengti atsitiktinio struktūros panašumo.

Structure parserio režimas leidžia įvertinti failų struktūros panašumą, bet jis pateikia nemažai triukšmo, nes atsiranda neteisingai panašūs įvertinimai, pvz. visi 46-tai *ClassParser* klasei panašumą rodantys rezultatai. Visais didesnio panašumo atvejais panaši tik kodo forma, pavyzdžiui, vietoje *for* ciklų randami *try* blokai. Toks palyginimas gali būti naudingas, bet jis reikalauja papildomų žmogiškų pastangų įvertinti realų klasių panašumą.

Visų algoritmų reikšmių pasiskirstymą vizualiai parodo 11 pav. pateikta visų metodų darbo rezultatų histograma. Joje atsiskleidžia grafikuose sunkiau pastebimas diff bei pq-gram algoritmų rezultatų skirtumas – pq-gram rezultatas randa žymiai daugiau panašių klasių. Histogramoje matomas ir medžių redagavimo atstumo poslinkis į panašumo pusę – šio metodo rezultatai daug tolygiau pasiskirstę per galimų reikšmių skalę.




11 pav. *args4j* palyginimo rezultatų pasiskirstymas, naudojant pilną (FULL) režimą. Pq-gram ir pq-print histogramos sutampa, todėl pateikta tik viena iš jų.

## 4.6. Palyginimo metodų sparta

Lyginant palyginimo metodus, būtina įvertinti ir jų spartą. Sparta įvertinta, lyginant *args4j* bibliotekos 2.0.10 ir 2.0.16 versijas. Šį palyginimą sudaro 1591 atskirų klasių palyginimas. Spartos matavimo rezultatai pateikiami 3 lentelėje.

| | Filtras | Diff | Redagavimo atstumas | Pq-gram | Pq-print |
|---|---|---|---|---|---|
| Java kodo palyginimas | Pilnas | 0:36 | 57:05 | 38:08 | 38:45 |
| | Nepilnas | – | 51:05 | 37:43 | 37:49 |
| | Struktūrinis | – | 44:08 | 38:11 | 37:49 |
| Prieš lyginimą išparsinus visus medžius į XML ir lyginant XML | Pilnas | – | 25:14 | 5:55 | 6:09 |
| | Nepilnas | – | 18:44 | 5:39 | 5:52 |
| | Struktūrinis | – | 12:47 | 5:17 | 5:24 |
| Prieš lyginimą išparsinus visus medžius į AST ir sugeneravus indeksus | Pilnas | – | – | 5:04 | 4:49 |
| | Nepilnas | – | – | 5:05 | 4:49 |
| | Struktūrinis | – | – | 5:03 | 4:48 |

3 lentelė. *args4j* 2.0.10 ir 2.0.16 versijų palyginimo sparta.

Atliktas paprastas palyginimas parodo, jog dideli medžio dydžio pasikeitimai turi sąlyginai mažą įtaką palyginimo laikui. Medžių redagavimo atstumo algoritmas yra gerokai lėtesnis už pq-gram algoritmą. Pq-print algoritmo našumas beveik identiškas pq-gram atstumo algoritmui, kuriuo jis paremtas. Profiliavimas parodo, kad Java kodo parsinimas į AST medį, atliekamas kiekvienam palyginimui, užima apie pusę sugaišto laiko. Eksperimentas buvo patobulintas: prieš pradedant lyginti versijas, kiekvienai Java rinkmenai sugeneruojamas ją atitinkantis XML dokumentas. Tuomet dokumentai lyginami tarpusavyje kaip pirmu atveju. Tai leidžia išvengti pakartotinio Java išeities kodo parsinimo bei AST medžių generavimo kiekvienam palyginimui. Patobulinus eksperimentą, gaunami duomenys, geriau atitinkantys teorinę algoritmų spartą. Palyginimo laikas medžio redagavimo atstumo algoritmu, taikant pilną (FULL) ir struktūrinį (STRUCTURE) parserio darbo režimus, skiriasi 50%. Tuo tarpu pq-gram bei pq-print algoritmams didesnis lyginamų duomenų kiekis turi gerokai mažesnę įtaką. Atmetus Java parserio sugaištą laiką, žymesniu tampa pq-print algoritmo užimamo laiko skirtumas lyginant su pq-gram algoritmu. Pq-gram paremtus algoritmus galima dar pagreitinti, prieš lyginimą apskaičiuojant atitinkamai pq-gram ir pq-gram antspaudų indeksus. Atlikus šį pakeitimą, abiejų algoritmų darbo laikas sutrumpėja, bei darbo laiko skirtumas tarp pilno (FULL) ir struktūrinio (STRUCTURE) parserio režimų nukrenta iki poros sekundžių. Pq-print algoritmas pradeda veikti greičiau už pq-gram, nes indeksus, kurie yra lyginami, sudaro daug paprastesni elementai. Tai lemia tiek indeko elementų palyginimo greitį, tiek rinkmenoje išsaugoto indekso dydį.

Pradinis lyginamo išeities kodo apdorojimas, jį parsinant į AST medį bei pastarąjį išsaugant XML formatu, iš viso užtrunka 1:16 apdoroti abi bibliotekos versijas. Pridėjus pq-gram indekso apskaičiavimą ir saugant jį, pq-gram ir pq-print pradinis apdorojimas trunka po 1:20. Likęs vykdymo laikas sugaištamas visų Java klasių tarpusavio palyginimui.

Pasiektas paspartėjimas ir maža medžių dydžio įtaka palyginimo spartai suponuoja, kad pq-gram paremti metodai tinka lyginti dideles programinio kodo sistemas. Optimizuotų pq-gram ir pq-print metodų rezultatai rodo, kad palyginimo metodų sparta, didėjant kodo apimčiai, lėtėja pakankamai negreitai, lyginant su neoptimizuota metodų versija. Didėjant kodo apimčiai, pradinio apdorojimo trukmė didės tiesiškai, priklausomai nuo lyginamų klasių kiekio, o sistemų palyginimo trukmė augs prikausomai nuo palyginimų kiekio. Taigi, pq-print algoritmu galima lyginti didesnes sistemas, nes jis palyginimą atlieka gerokai greičiau už pq-gram algoritmą. Tuo tarpu dėl mažos spartos ir didelio jautrumo lyginamų medžių dydžiui, medžių redagavimo atstumo algoritmo programų sistemų išeities kodo lyginimui naudoti netikslinga.

# Išvados

Medžių palyginimo metodai yra tinkami lyginti XML dokumentams, programiniam kodui bei kitokiam struktūrizuotam tekstui. Sąlyginai mažos apimties medžiams lyginti verta naudoti tikslų medžių redagavimo atstumo algoritmą, tačiau, išaugus duomenų apimčiai, praktiška rinktis apytikslio palyginimo algoritmus.

Pq-gram apytikslio medžių palyginimo algoritmas veikia gerokai greičiau už tikslius algoritmus bei išlaiko kodo blokų palyginimui reikalingą palyginimo tikslumą. Programinio kodo palyginimui optimalių p ir q reikšmių parinkimas nėra akivaizdus net ir turint pakankamai eksperimentinių duomenų, nes, išskyrus kelis ribinius atvejus, reikšmingo skirtumo tarp rezultatų, pasirinkus skirtingas tiriamų parametrų reikšmes, nėra. Vis dėlto priimtini rezultatai gaunami, pasirinkus beveik bet kokias parametrų p ir q reikšmes, todėl šis algoritmas yra tinkamas palyginti programinį kodą.

Pasiūlyta pq-gram algoritmo optimizacija miškams lyginti supaprastina palyginimų cikle atliekamas operacijas bei sumažina jų kiekį. Tai pasiekiama išaugusio pradinio apdorojimo operacijų kiekio kaina. Šalutinis šio algoritmo poveikis ženkliai sumažina pq-gram indekso dydį, kas supaprastina jo saugojimą. Optimizuotas algoritmas leidžia lyginti didenės apimties programinę įrangą, išlaikant pq-gram metodo rezultatų kokybę.

Kitas būdas pagreitinti kodo palyginimo algoritmų našumą paremtas AST medžio transformacijomis. Žinant galimus medžio elementus, įmanoma ženkliai sumažinti medžio mazgų skaičių: pašalinus dalį savyje informacijos nenešančių mazgų, pagreitinamas palyginimo algoritmų veikimas. Vis dėlto tokiomis transformacijomis galima pasinaudoti tik tada, kai mazgų seka yra žinoma (pvz. fiksuota atributų eilės tvarka), kitu atveju negrįžtamai prarandama struktūrinė informacija.

Šiame darbe pristatyti programinio kodo palyginimo metodai gali būti patobulinti, specializuotai pritaikant AST parserį Java programavimo kalbai. Patys palyginimo metodai gali būti patobulinti, pridedant semantinės analizės galimybes, pvz. analizuoti pakitusių medžių elementų prasmę ir ieškoti susijusių pakitimų, tokių kaip kintamųjų pervadinimas. Tuo atveju susiję, kelis mazgus apimantys pakitimai galėtų būti fiksuojami kaip viena redagavimo operacija ir taip sumažinama trivialių pakitimų reikšmė palyginiui.

# Literatūros sąrašas

[ABG05] Nikolaus Augsten, Michael Böhlen, and Johann Gamper. Approximate matching of hierarchical data using pq-grams. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 301–312. VLDB Endowment, 2005.

[ABG08] Nikolaus Augsten, Michael Böhlen, and Johann Gamper. The pq-gram distance between ordered labeled trees. *ACM Trans. Database Syst.*, 35:4:1–4:36, February 2008.

[Bak95] B. Baker. On finding duplication and near-duplication in large software systems. In *IEEEE Proceedings of the Working Conference on Reverse Engineering*, 1995.

[Dam64] Fred J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, March 1964.

[HPE+06] Kenneth Hoste, Aashish Phansalkar, Lieven Eeckhout, Andy Georges, Lizy K. John, and Koen De Bosschere. Performance prediction based on inherent program similarity. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 114–122, New York, NY, USA, 2006. ACM.

[LPM+97] Bruno Lague, Daniel Proulx, Jean Mayrand, Ettore M. Merlo, and John Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. *Software Maintenance, IEEE International Conference on*, 0:314, 1997.

[Rab81] Michael O. Rabin. Fingerprinting by random polynomials. Technical report, Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[SBPK06] Tobias Sager, Abraham Bernstein, Martin Pinzger, and Christoph Kiefer. Detecting similar java classes using tree algorithms. In *In Proceedings of the 2006 international Workshop on Mining Software Repositories*, pages 65–71. ACM Press, 2006.

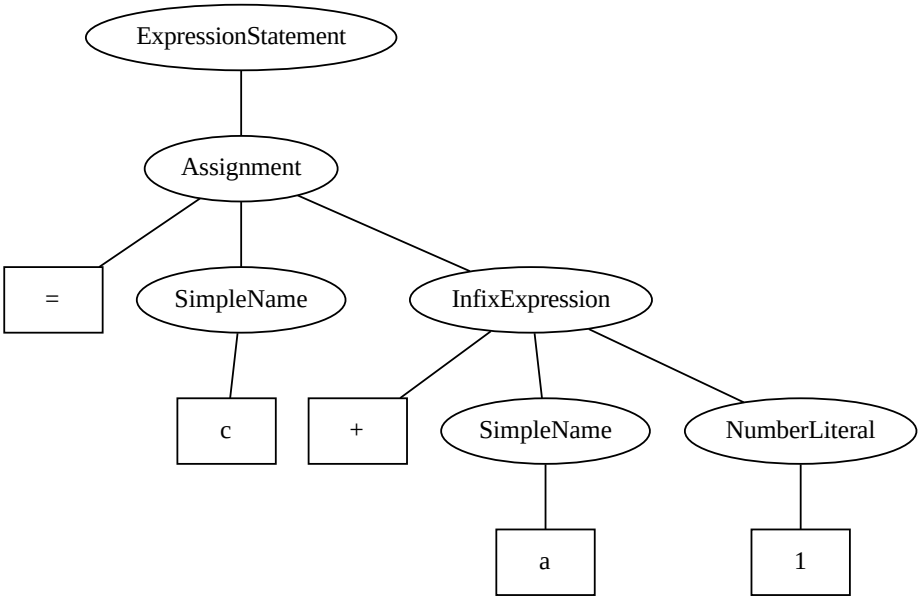
[SWA03] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 76–85, New York, NY, USA, 2003. ACM.

[ZS89] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18:1245–1262, December 1989.

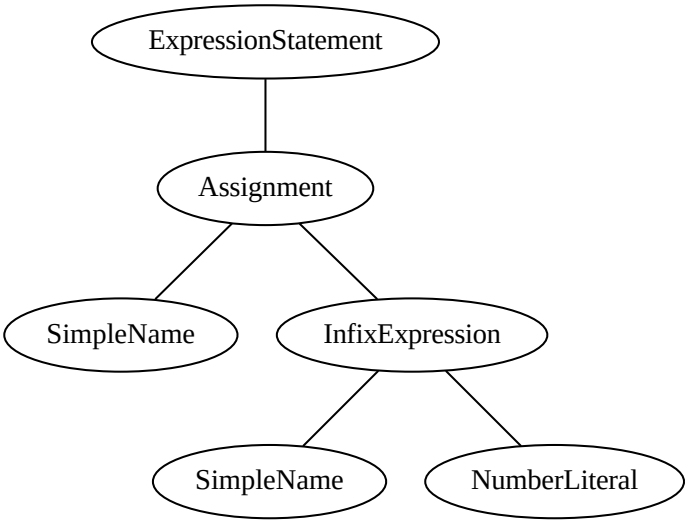
# Priedas Nr. 1.
# Java AST medžių parserio darbo režimų pavyzdys

Žemiau pateikiami Java AST parserio sugeneruoti medžių fragmentai. Visi fragmentai vaizduoja vieną Java sakinį:

```
c = a + 1;
```




12 pav. Nepilno parserio režimo rezultatas




13 pav. Struktūrinio parserio režimo rezultatas

# Priedas Nr. 2.
# Eksperimentinio p ir q parinkimo duomenys

Šiame priede pateikta dalis eksperimentų metu gautų duomenų, lėmusių p ir q parinkimą. Visos programos lygintos su burbulo rūšiavimo algoritmu, kurio išeities kodas naudojamas kitų programų kūrimui. Normalizuotas pq-gram atstumas išreikštas procentais, mažesnės reikšmės reiškia didesnį panašumą. Visose lentelėse p kinta eilutėse, q – stulpeliuose.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.875 | 0.929 | 0.958 | 0.977 | 0.989 | 0.999 | 1.006 | 1.011 | 1.016 | 1.019 |
| 2 | 0.875 | 0.929 | 0.958 | 0.977 | 0.989 | 0.999 | 1.006 | 1.011 | 1.016 | 1.019 |
| 3 | 0.875 | 0.929 | 0.958 | 0.977 | 0.989 | 0.999 | 1.006 | 1.011 | 1.016 | 1.019 |
| 4 | 0.875 | 0.929 | 0.958 | 0.977 | 0.989 | 0.999 | 1.006 | 1.011 | 1.016 | 1.019 |
| 5 | 0.875 | 0.929 | 0.958 | 0.977 | 0.989 | 0.999 | 1.006 | 1.011 | 1.016 | 1.019 |
| 6 | 0.875 | 0.929 | 0.958 | 0.977 | 0.989 | 0.999 | 1.006 | 1.011 | 1.016 | 1.019 |
| 7 | 0.875 | 0.929 | 0.958 | 0.977 | 0.989 | 0.999 | 1.006 | 1.011 | 1.016 | 1.019 |
| 8 | 0.875 | 0.929 | 0.958 | 0.977 | 0.989 | 0.999 | 1.006 | 1.011 | 1.016 | 1.019 |
| 9 | 0.875 | 0.929 | 0.958 | 0.977 | 0.989 | 0.999 | 1.006 | 1.011 | 1.016 | 1.019 |
| 10 | 0.875 | 0.929 | 0.958 | 0.977 | 0.989 | 0.999 | 1.006 | 1.011 | 1.016 | 1.019 |

4 lentelė. Normalizuotas pq-gram atstumas lyginant su programa, kurioje pakeista konstantos reikšmė. Visos reikšmės išreikštos procentais.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 7.611 | 8.060 | 8.304 | 8.459 | 8.565 | 8.642 | 8.701 | 8.747 | 8.785 | 8.816 |
| 2 | 7.611 | 8.060 | 8.304 | 8.459 | 8.565 | 8.642 | 8.701 | 8.747 | 8.785 | 8.816 |
| 3 | 7.611 | 8.060 | 8.304 | 8.459 | 8.565 | 8.642 | 8.701 | 8.747 | 8.785 | 8.816 |
| 4 | 7.611 | 8.060 | 8.304 | 8.459 | 8.565 | 8.642 | 8.701 | 8.747 | 8.785 | 8.816 |
| 5 | 7.611 | 8.060 | 8.304 | 8.459 | 8.565 | 8.642 | 8.701 | 8.747 | 8.785 | 8.816 |
| 6 | 7.611 | 8.060 | 8.304 | 8.459 | 8.565 | 8.642 | 8.701 | 8.747 | 8.785 | 8.816 |
| 7 | 7.611 | 8.060 | 8.304 | 8.459 | 8.565 | 8.642 | 8.701 | 8.747 | 8.785 | 8.816 |
| 8 | 7.611 | 8.060 | 8.304 | 8.459 | 8.565 | 8.642 | 8.701 | 8.747 | 8.785 | 8.816 |
| 9 | 7.611 | 8.060 | 8.304 | 8.459 | 8.565 | 8.642 | 8.701 | 8.747 | 8.785 | 8.816 |
| 10 | 7.611 | 8.060 | 8.304 | 8.459 | 8.565 | 8.642 | 8.701 | 8.747 | 8.785 | 8.816 |

5 lentelė. Normalizuotas pq-gram atstumas lyginant su programa, kurioje pervadintas kintamasis. Visos reikšmės išreikštos procentais.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 4.158 | 4.334 | 4.431 | 4.683 | 4.856 | 4.982 | 5.078 | 5.154 | 5.215 | 5.266 |
| 2 | 4.158 | 4.334 | 4.431 | 4.683 | 4.856 | 4.982 | 5.078 | 5.154 | 5.215 | 5.266 |
| 3 | 4.158 | 4.334 | 4.431 | 4.683 | 4.856 | 4.982 | 5.078 | 5.154 | 5.215 | 5.266 |
| 4 | 4.158 | 4.334 | 4.431 | 4.683 | 4.856 | 4.982 | 5.078 | 5.154 | 5.215 | 5.266 |
| 5 | 4.158 | 4.334 | 4.431 | 4.683 | 4.856 | 4.982 | 5.078 | 5.154 | 5.215 | 5.266 |
| 6 | 4.158 | 4.334 | 4.431 | 4.683 | 4.856 | 4.982 | 5.078 | 5.154 | 5.215 | 5.266 |
| 7 | 4.158 | 4.334 | 4.431 | 4.683 | 4.856 | 4.982 | 5.078 | 5.154 | 5.215 | 5.266 |
| 8 | 4.158 | 4.334 | 4.431 | 4.683 | 4.856 | 4.982 | 5.078 | 5.154 | 5.215 | 5.266 |
| 9 | 4.158 | 4.334 | 4.431 | 4.683 | 4.856 | 4.982 | 5.078 | 5.154 | 5.215 | 5.266 |
| 10 | 4.158 | 4.334 | 4.431 | 4.683 | 4.856 | 4.982 | 5.078 | 5.154 | 5.215 | 5.266 |

6 lentelė. Normalizuotas pq-gram atstumas lyginant su programa, kurioje įterptas priskyrimo instrukcija. Visos reikšmės išreikštos procentais.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.000 | 0.929 | 0.958 | 0.977 | 0.989 | 0.999 | 1.006 | 1.011 | 1.016 | 1.019 |
| 2 | 0.000 | 0.929 | 0.958 | 0.977 | 0.989 | 0.999 | 1.006 | 1.011 | 1.016 | 1.019 |
| 3 | 0.000 | 0.929 | 0.958 | 0.977 | 0.989 | 0.999 | 1.006 | 1.011 | 1.016 | 1.019 |
| 4 | 0.000 | 0.929 | 0.958 | 0.977 | 0.989 | 0.999 | 1.006 | 1.011 | 1.016 | 1.019 |
| 5 | 0.000 | 0.929 | 0.958 | 0.977 | 0.989 | 0.999 | 1.006 | 1.011 | 1.016 | 1.019 |
| 6 | 0.000 | 0.929 | 0.958 | 0.977 | 0.989 | 0.999 | 1.006 | 1.011 | 1.016 | 1.019 |
| 7 | 0.000 | 0.929 | 0.958 | 0.977 | 0.989 | 0.999 | 1.006 | 1.011 | 1.016 | 1.019 |
| 8 | 0.000 | 0.929 | 0.958 | 0.977 | 0.989 | 0.999 | 1.006 | 1.011 | 1.016 | 1.019 |
| 9 | 0.000 | 0.929 | 0.958 | 0.977 | 0.989 | 0.999 | 1.006 | 1.011 | 1.016 | 1.019 |
| 10 | 0.000 | 0.929 | 0.958 | 0.977 | 0.989 | 0.999 | 1.006 | 1.011 | 1.016 | 1.019 |

7 lentelė. Normalizuotas pq-gram atstumas lyginant su programa, kurioje sukeista instrukcijų tvarka. Visos reikšmės išreikštos procentais.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 19.028 | 20.228 | 20.879 | 21.288 | 21.434 | 21.540 | 21.622 | 21.686 | 21.737 | 21.780 |
| 2 | 21.200 | 22.504 | 23.210 | 23.654 | 23.827 | 23.954 | 24.050 | 24.126 | 24.187 | 24.237 |
| 3 | 22.619 | 23.743 | 24.353 | 24.737 | 24.870 | 24.968 | 25.042 | 25.101 | 25.148 | 25.187 |
| 4 | 22.970 | 23.743 | 24.353 | 24.737 | 24.870 | 24.968 | 25.042 | 25.101 | 25.148 | 25.187 |
| 5 | 22.970 | 23.743 | 24.353 | 24.737 | 24.870 | 24.968 | 25.042 | 25.101 | 25.148 | 25.187 |
| 6 | 22.970 | 23.743 | 24.353 | 24.737 | 24.870 | 24.968 | 25.042 | 25.101 | 25.148 | 25.187 |
| 7 | 22.970 | 23.743 | 24.353 | 24.737 | 24.870 | 24.968 | 25.042 | 25.101 | 25.148 | 25.187 |
| 8 | 22.970 | 23.743 | 24.353 | 24.737 | 24.870 | 24.968 | 25.042 | 25.101 | 25.148 | 25.187 |
| 9 | 22.970 | 23.743 | 24.353 | 24.737 | 24.870 | 24.968 | 25.042 | 25.101 | 25.148 | 25.187 |
| 10 | 22.970 | 23.743 | 24.353 | 24.737 | 24.870 | 24.968 | 25.042 | 25.101 | 25.148 | 25.187 |

8 lentelė. Normalizuotas pq-gram atstumas lyginant su programa, kurioje dalis kodo iškelta į atskirą metodą. Visos reikšmės išreikštos procentais.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 43.913 | 46.029 | 46.934 | 47.601 | 48.140 | 48.533 | 48.831 | 49.066 | 49.256 | 49.412 |
| 2 | 51.243 | 52.199 | 52.834 | 53.325 | 53.738 | 54.039 | 54.269 | 54.449 | 54.595 | 54.715 |
| 3 | 57.162 | 58.161 | 58.811 | 59.303 | 59.711 | 60.009 | 60.235 | 60.413 | 60.557 | 60.675 |
| 4 | 60.130 | 60.517 | 60.829 | 61.108 | 61.300 | 61.440 | 61.547 | 61.631 | 61.700 | 61.756 |
| 5 | 60.853 | 61.412 | 61.817 | 62.153 | 62.384 | 62.553 | 62.682 | 62.783 | 62.865 | 62.932 |
| 6 | 62.452 | 62.919 | 63.272 | 63.574 | 63.781 | 63.933 | 64.049 | 64.140 | 64.213 | 64.274 |
| 7 | 64.015 | 64.273 | 64.509 | 64.736 | 64.892 | 65.006 | 65.093 | 65.162 | 65.217 | 65.263 |
| 8 | 64.358 | 64.758 | 65.073 | 65.348 | 65.537 | 65.675 | 65.781 | 65.864 | 65.931 | 65.987 |
| 9 | 64.698 | 65.000 | 65.260 | 65.500 | 65.665 | 65.786 | 65.879 | 65.951 | 66.010 | 66.058 |
| 10 | 64.698 | 65.000 | 65.260 | 65.500 | 65.665 | 65.786 | 65.879 | 65.951 | 66.010 | 66.058 |

9 lentelė. Normalizuotas pq-gram atstumas lyginant su greitojo rūšiavimo (*quicksort*) programa. Visos reikšmės išreikštos procentais.

# Priedas Nr. 3.
# args4j Java rinkmenų numeravimas grafikuose

Visos klasės saugomos args4j/src kataloge, išskyrus pavyzdžius, kurie saugomi args4j/examples kataloge. Pavyzdžių klasėms paketo (angl. *package*) pavadinimas nepateikiamas.

1. org/kohsuke/args4j/spi/EnumOptionHandler.java
2. org/kohsuke/args4j/spi/DoubleOptionHandler.java
3. org/kohsuke/args4j/spi/BooleanOptionHandler.java
4. org/kohsuke/args4j/spi/Parameters.java
5. org/kohsuke/args4j/spi/IntOptionHandler.java
6. org/kohsuke/args4j/spi/Setter.java
7. org/kohsuke/args4j/spi/Messages.java
8. org/kohsuke/args4j/spi/FileOptionHandler.java
9. org/kohsuke/args4j/spi/OptionHandler.java
10. org/kohsuke/args4j/spi/StringOptionHandler.java
11. org/kohsuke/args4j/IllegalAnnotationError.java
12. org/kohsuke/args4j/ExampleMode.java
13. org/kohsuke/args4j/CmdLineParser.java
14. org/kohsuke/args4j/CmdLineException.java
15. org/kohsuke/args4j/MethodSetter.java
16. org/kohsuke/args4j/MultiValueFieldSetter.java
17. org/kohsuke/args4j/Argument.java
18. org/kohsuke/args4j/Messages.java
19. org/kohsuke/args4j/FieldSetter.java
20. org/kohsuke/args4j/Option.java
21. SampleMain.java
22. org/kohsuke/args4j/NamedOptionDef.java
23. org/kohsuke/args4j/spi/LongOptionHandler.java
24. org/kohsuke/args4j/spi/StopOptionHandler.java
25. org/kohsuke/args4j/spi/ShortOptionHandler.java
26. org/kohsuke/args4j/spi/URLOptionHandler.java
27. org/kohsuke/args4j/spi/StringArrayOptionHandler.java
28. org/kohsuke/args4j/spi/MapOptionHandler.java
29. org/kohsuke/args4j/spi/OneArgumentOptionHandler.java
30. org/kohsuke/args4j/spi/CharOptionHandler.java
31. org/kohsuke/args4j/spi/FloatOptionHandler.java
32. org/kohsuke/args4j/spi/ByteOptionHandler.java
33. org/kohsuke/args4j/MapSetter.java
34. org/kohsuke/args4j/Starter.java
35. org/kohsuke/args4j/OptionDef.java
36. SampleAnt.java
37. SampleStarter.java
38. org/kohsuke/args4j/spi/Setters.java
39. org/kohsuke/args4j/spi/MethodSetter.java
40. org/kohsuke/args4j/spi/MultiValueFieldSetter.java
41. org/kohsuke/args4j/spi/FieldSetter.java
42. org/kohsuke/args4j/spi/RestOfArgumentsHandler.java
43. org/kohsuke/args4j/spi/URIOptionHandler.java
44. org/kohsuke/args4j/XmlParser.java
45. org/kohsuke/args4j/Config.java
46. org/kohsuke/args4j/ClassParser.java