TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika eriala

Jaanus Jaggo

Sarnaste koodisegmentide põhjal soovitusmudeli loomine

Magistritöö (30 EAP)

Juhendaja: Siim Karus, PhD

Tartu 2016

Sarnaste koodisegmentide põhjal soovitusmudeli loomine

Lühikokkuvõte: Käesoleva töö eesmärgiks on luua töövoog soovitusmudeli koostamiseks, mida saaks rakendada staatilises koodianalüüsis kasutajale soovituste tegemisteks. Selleks otsib süsteem sarnaseid koodisegmente avatud lähtekoodiga projektidest ning klasterdab need sarnasuse alusel. See lähenemine põhineb hüpoteesil, et kui arendajad kirjutavad mitu korda sarnast koodi, siis sisaldab see programmeerimismustrit, mida võiks teistelegi soovitada. Töös kontrollitakse lähenemise sobivust ning hinnatakse loodud meetodi kasulikkust staatilises koodianalüüsis. Kokkuvõttes leiti, et sel viisil tehtud soovitused ei arvesta piisavalt analüüsitava projekti eripäradega ega pole rakendatavad selles projektis.

Võtmesõnad: Staatiline analüüs, koodikloonid

CERCS: P170 (Arvutiteadus, arvanalüüs, süsteemid, kontroll)

Creating a recommendation model based on similar code segments

Abstract: The goal of this work is to develop a workflow for the creation of recommendation model which could be used in static code analysis. The system searches for similar code segments from open source projects and clusters them based on their similarity. The hypothesis is that whenever developers write same code multiple times, it would contain an useful pattern that can be recommended to others. The aim of this work is to test this assumption and evaluate the usefulness of the developed process for static code analysis. In conclusion, it was found that the recommendations do not consider the specifics of the analysed project and are therefore not applicable in the project.

Keywords: Static analysis, code clones

CERCS: P170 (Computer science, numerical analysis, systems, control)

Sisukord

1 Sissejuhatus **4**

2 Koodikloonid **5**
2.1 Koodikloonide klassifikatsioon 5
2.2 Koodikloonide rakendamine staatilises analüüsis 5
2.3 Nõuded koodikloonide tuvastamise tehnikale 6

3 Koodikloonide tuvastmise tehnikad **7**
3.1 Lainikupõhine kloonide tuvastamine 7
3.2 Lekseemipõhine kloonide tuvastamine 8
3.3 Abstraktsel süntaksipuul põhinev kloonide tuvastamine 9
3.4 Süntaktilisel sufiksipuul põhinev kloonide tuvastamine 11
3.5 Semantikapõhine kloonide tuvastamine 12
3.6 Kloonituvastustehnikate kokkuvõte ja võrdlus 15

4 Soovitusmudeli koostamise töövoog **16**
4.1 Soovitusmudeli koostamise juhtimine 16
4.2 Koodisegmendi hoidmine andmebaasis 18

5 Lähtekoodi parsimine ja transleerimine **19**
5.1 Konreetse süntaksipuu transleerimine abstraktseks koodipuuks 19
5.2 Valikulause transleerimine 20
5.3 Avaldiste transleerimine 22
5.4 Abstraktse koodipuu tippude kontekst 23
5.5 Abstraktse koodipuu jagamine funktsioonideks 24

6 Koodisegmentide sarnasuse hindamine **25**
6.1 Abstraktse koodipuu teisendamine graafiks 26
6.2 Semantikat säilitav sarnasuse hindamine 27

7 Analüüsi töövoog **29**
7.1 Konteksti asendamine lähtekoodis 29
7.2 Rakenduse arhitektuur 30

8 Tulemused **31**
8.1 Algoritmi jõudluse hindamine 32
8.2 Migreeritud koodisegmentide jaotus 33
8.3 Tulemuste kvaliteedi hindamine 34
8.4 Autori hinnang töö tulemustele 37

9 Kokkuvõte **38**

1 Sissejuhatus

Tarkvaratööstuses on heaks arenduspraktikaks läbi viia korrapäraseid koodiülevaatuseid. Koodiülevaatuse käigus loevad projekti arendajad üksteise lähtekoodi ja jagavad soovitusi, kuidas seda paremaks muuta. Lisaks kasutatakse ka staatilist analüüsi, mis aitab koodiülevaatustele kuluvat aega vähendada. Staatlise koodianalüüsi tööriista loomine on aga kallis töö, sest vastava soovitusmudeli jaoks on vaja kirja panna palju veamustreid ning need on ka programmeerimiskeelte lõikes erinevad. Käesoleva töö eesmärgiks on luua meetod staatilise analüüsi jaoks sobiva soovitusmudeli koostamiseks, kasutades avatud lähtekoodiga projektides esinevaid sarnaseid koodisegmente. Samuti on eesmärgiks kontrollida, kas sellisel viisil õnnestub õppida pigem häid programmeerimise praktikaid, mida saaks kasutajale staatilise analüüsi käigus soovitada.

Sarnaste koodisegmentide põhjal soovitusmudeli koostamine põhineb hüpoteesil, et analüüsitavates projektides esineb sagedamini häid programmeerimise mustreid kui halbu. Luues efektiivse meetodi koodisegmentide klassifitseerimiseks saab kasutajale kõige sagedamini esinevaid mustreid soovitada. Näiteks kui JavaScriptis kirjutatud programmis on muutujate võrdlemiseks kasutatud operaatorit ==, kuid enam levinud praktikas kasutatakse operaatorit ===, võiks süsteem selle ära õppida ning staatilise analüüsi käigus viimast arendajale soovitada.

Kuna treeningandmeteks kasutatavad projektid sisaldavad paljude erinevate arendajate poolt kirjutatud koodi, on oluline, et koodisegmentide sarnasuse hindamisel oleks arvestatud ka erinevate programmeerimisstiilide ja -praktikatega. Kuna sama koodi on võimalik mitut moodi kirja panna, kirjeldatakse töös analüüsiprotsessi, mille käigus tõlgitakse lähtekood lihtsasse vahekeelde ja sarnasuse hindamisel võimaldatakse koodisegmentide liikumist lubatud konteksti piires.

Selliselt toimiva staatilise analüsaatori loomine mitte ainult ei vähenda arendajate tööd, vaid rajab tee mitmetele uutele võimalustele staatilise analüüsis. Esiteks võib automatiseeritud protsess tuvastada mustreid, mida inimene ei taipaks kirjeldada. Teiseks võimaldab see luua tööriistu, mis täiendavad ennast aja jooksul, näiteks migreerivad iga analüüsimiseks saadetud faili soovitusmudelisse juurde.

Peatükis 2 vaadeldakse erinevaid koodikloonide klassifikatsioone ning defineeritakse nõuded nende tuvastamise tehnikale. Peatükis 3 tutvustatakse enamlevinud koodikloonide tuvastamise tehnikaid ja vaadeldakse, kuidas need töötavad. Peatükis 4 käsitletakse töövoogu koodikloonide tuvastamiseks ning klassifitseerimiseks. Peatükis 5 kirjeldatakse abstraktse koodipuu struktuur ja meetodit selle koostamiseks. Peatükis 6 kirjeldatakse meetodit koodisegmentide sarnasuse hindamiseks, mis arvestab koodisegmentide liikumisega sama konteksti piires. Peatükis 7 kirjeldatakse analüüsi töövoogu ja rakenduse kasutamist. Peatükis 8 antakse ülevaade töö tulemustest.

2 Koodikloonid

Koodisegmentide kopeerimine ja mugandamine on levinud programmeerimise praktika. Sel viisil tekivad lähtekoodi sarnased koodilõigud, mida nimetatakse kloonideks. Enamasti on arendajad teadlikud oma programmis olevatest kloonidest ning need on ka programmi arhitektuuris olulisel kohal. Mõnikord võivad aga tekkida kloonid, mille olemasolust arendaja ei tea, või on selle unustanud. Sellised kloonid on potensiaalseks ohuallikaks, sest kui arendaja teeb hiljem muudatuse ühe klooni isendi juures, võib see teises kohas muutmata jääda. Selle vältimiseks peaksid kõik kavatsuslikud kloonid olema dokumenteeritud, kuid erinevad uuringud näitavad, et praktikas tehakse seda harva [KKI02]. Tahtmatute ja unustatud koodikloonide leidmiseks on loodud mitmeid meetodeid ning neil on ka oma roll teistes valdkondades, nagu plagiarismi tuvastamisel, intellektuaalse omandi kaitses ning staatilises analüüsis, millest viimast vaatleme käesolevas töös.

2.1 Koodikloonide klassifikatsioon

Koodikloonid jagatakse formaalselt kolme klassi [BKA+07]:

1. Tüüp 1 kloonid on tekstilised koopiad, mis erinevad ainult formaadi, kommentaaride ja tühimärkide poolest.
2. Tüüp 2 kloonid võivad lisaks erineda ka parameetrite ja muutujate nimede poolest.
3. Tüüp 3 kloonid omavad sama tähendust, kuid lausete asukohad võivad olla vahetuses, samuti võib lauseid olla lisatud või eemaldatud.

Tarkvaratööstuses kasutatavad tööriistad peavad enamasti tuvastama vähemalt tüüp 2 kloone, sest koodi üks-ühele kopeerimist tuleb seal harva ette. Seevastu plagiaadi tuvastamisel piisab sageli ka esimest tüüpi kloonide leidmisest. Kolmandat tüüpi kloonipaar võib olla koguni kahe erineva inimese poolt kirjutatud lahendus sama ülesande jaoks ja seetõttu saab seda kasutada näiteks selleks, et välja selgitada, milliseid algoritme kasutasid tudengid sama ülesande lahendamiseks. Kuigi erinevaid kloonide tuvastamise tehnikaid on mitmeid, saab need üldjoontes jagada kolmeks:

1. Teksti- ja lekseemipõhised tehnikad tuvastavad enamasti vaid tüüp 1 kloone.
2. Süntaksipõhised tehnikad tuvastavad tüüp 1 ja tüüp 2 kloone.
3. Semantikapõhised tehnikad tuvastavad kõiki kolme tüüpi kloone.

Lisaks esineb ka tehnikaid, mis kasutavad kombinatsiooni mitmest lähenemisest, näiteks Koschke süntaktilise sufiksipuu (*syntax suffix tree*) algoritmis on kombineerinud lekseemi- ja süntaksipõhine lähenemine [KFF06].

2.2 Koodikloonide rakendamine staatilises analüüsis

Koodikloonid on kasulikuks tööriistaks arendusprotsessi mõistmisel. Kui arendaja teadlikult kopeerib või kirjutab sarnast koodi, siis ei tee ta seda üldjuhul mitte laiskusest, vaid kaalutletud disainiotsusena. Samuti vajab kopeeritud kood enamasti muutmist, et see uues kohas tööle panna. Yun Lin ja teised on loonud kontseptuaalse tööriista CCDemon

[LPX+15], mis aitab kasutajat koodi kopeerimisel, soovitades talle võimalikke muudatusi. Muudatusettepanekute tegemiseks otsib tööriist muudetava koodiga sarnaseid koodikloone ja soovitab kasutajale muudatusettepanekutena kloonides esinevaid süntaktilisi erinevuseid. CCDemon osutus kasulikuks tööriistaks, mis suutis 75% juhtudest soovitada kasutajale sobiva muudatuse.

Koodikloonide tuvastamise tehnikaid kasutavad ka paljud internetipõhised koodi otsingumootorid näiteks SearchCode [Sea16], mis otsib sisestatud lähtekoodile sarnaseid koodisegmente teistest koodifailidest. Tulemuseks kuvatakse kasutajale kõik sarnast segmenti sisaldavad koodifailid, milles on kattuvad võtmesõnad ära värvitud. Saadud otsingu tulemusi saab edukalt kasutada oma programmi korrigeerimiseks, kuid see pole kuigi intuitiivne protsess, sest kasutaja peab ise otsingutulemused läbi vaatama ja nende sobivust hindama.

JavaScripti jaoks on loodud mitmeid veebipõhiseid staatilisi analüsaatoreid, üheks selliseks on JSLint [JSL16]. See on kasulik tööriist, mille veebiliides võimaldab arendajatel oma lähtekoodi mugavalt kontrollida. Peamiselt kontrollib see lähtekoodi vastavust stiilinõuetele ning kasutab selleks mustrite sobitamisel põhinevat analüüsimeetodit.

Käesolevas töös on loodud tööprotsess, mis kogub koodisegmente paljudest projektidest ning klasterdab need sarnasuse põhjal soovitusmustriteks. Staatilise analüüsi käigus otsitakse sisestatud koodile sarnaseid mustreid ning pakutakse neid kasutajale muudatusettepanekutena välja. Erinevalt eelnevalt kirjeldatud tööriistades CCDemon ja SearchCode on käesolevas töös realiseeritud koodikloonide tuvastmise tehnika kohandatud eelkõige struktuursete kloonide leidmisele, mis võivad olla erinevalt kirja pandud. Näiteks leiab see klooni ka juhul, kui kõigi muutujate nimed on ära muudetud ning For-kordamislause on kirjutatud While-lausena.

2.3 Nõuded koodikloonide tuvastamise tehnikale

Kuna soovitusmustrite loomiseks on vaja klassifitseerida koodisegmente paljudest projektidest, on oluline eelkõige kolmandat tüüpi kloonide tuvastamine. Ainult nii saab hinnata sarnasust kahe erineva arendaja poolt kirjutatud koodisegmendi vahel. See parandab ka analüsaatori võimekust teha kasutajale keerulisemaid soovitusi, sest sarnasteks loetakse ka erineva "väljanägemisega", ehk hägusaid kloone. Kolmandat tüüpi kloonide leidmiseks peab tööriist tuvastama kloone, milles on koodisegmentide omavahelist järjekorda muudetud või on neid lisatud ja kustutatud.

Olulise lihtsustusena ei ole töös arvestatud muutujate skoobi ega tüüpidega. Kui tüüpide puudumine pole probleem, sest rakendus tüübipõhist analüüsi ei tee, siis skoobi puudumise tõttu võib rakendus mõnes olukorras sama nimega muutujaid valesti tõlgendada. Kuna rakendus teisendab koodikloonide tuvastamisel mõlemat isendit samamoodi, siis valenegatiivsete tulemuste arvu see ei suurenda, küll aga võib see kitsendus lisada valepositiivseid tulemusi.

Süsteem on disainitud selliselt, et see oleks laiendatav erinevatele programmeerimiskeeltele. Kuigi käesoleva töö raames on loodud tugi ainult JavaScripti jaoks, on töös kasutatav vahekeel sobilik paljude imperatiivsete programmeerimiskeelte jaoks. Uute keelte lisamiseks on vaja kirjutada vastava keele jaoks translaator. Samuti on tarvis teha väiksemaid muudatusi keele ülesehituse tõlgedamiseks. Näiteks kui JavaScript'is on põhiprogramm funktsioonidest väljaspool, siis Java's asub see alati main meetodis.

3 Koodikloonide tuvastmise tehnikad

Koodikloonide tuvastamise tehnikate ja tööriistade võrdlemisel kasutatakse mitmeid karateristikuid [RCK09]. Käesoleva töös vaatleme nendest lähemalt järgmiseid:

a) toetatud programmeerimiskeeled;

b) kloonide granulaarsus (piiritletud funktsiooni või süntaktilise osa täpsusega);

c) arvutuslik keerukus halvimal juhul;

d) eel- ja järeltöötluse tehnikad;

e) täpsus (*precision*), ehk valepositiivsete tulemuste arv;

f) tundlikkus (*recall*), ehk tuvastamata kloonide arv.

3.1 Lainikupõhine kloonide tuvastamine

Naiivne lahendus kahest tekstifailist koodikloonide tuvastamiseks on võrrelda nende failide tähemärke paarikaupa. See lahendus ei skaleeru. Probleemi lahendamiseks koostas Kilgi [Kil14] lainikutel (*wavelet*) põhineva kloonituvastusmeetodi suurte projektide jaoks. Lainik on kindlate omadustega matemaatiline funktsioon, mida kasutatakse enim signaalitöötluses. Lainik võimaldab antud rakenduses teisendada andmed kompaktsemale kujule ja seetõttu lahendada skaleeruvuse probleem. Töös kasutati meetodi efektiivsuse hindamiseks Linux'i kerneli lähtekoodi, mis sisaldab 666 796 rida lähtekoodi (SLOC), kuid mille saab teisendada kõigest 57 293 tähemärgiks, rakendades kuus korda lainikul põhinevat teisendust, iga korraga vähendades tähemärkide arvu kaks korda. Kilgi tööriist koosneb neljast etapist, millest igaüks on realiseeritud eraldiseisva programmina.

1. Esimeses etapis teisaldatakse andmed numbrilisele kujule. Selle käigus kogutakse kokku kõik projekti failid, eemaldatakse sealt tühimikud ja kodeeritakse tähemärgid ASCII koodideks. See protsess võimaldab andmete mahtu vähendada ligikaudu 33%. Erinevalt paljudest teistest tööriistadest säilitatakse aga kommentaarid, sest need võivad sisaldada kasulikku infot plagiaadi tuvastamisel.

2. Teises etapis rakendatakse andmetele Haar lainikute põhist transformatsiooni [SF03]. Haar transformatsioon leiab paarikaupa kõrvuti asetsevate tähemärkide ASCII koodide keskmise väärtusega ja lisab saadud väärtuse väljundisse. Sisendist tulevaid väärtuseid läbitakse tagantpoolt ette ja saadud tulemused ümardatakse kahe komakoha täpsusega ujukomaarvuks. Täpselt sama protsessi korratakse uuesti tulemuseks saadud jadal, iga kord vähendades tulemuse mahtu poole võrra. Samuti korratakse kogu protsessi erinevate joondustega, mis vähendab hiljem valenegatiivsete kloonide arvu. Nii algoritmi maksimaalse sügavuse, kui ka joonduste arvu määrab programmi kasutaja parameetrina.

3. Kolmandas etapis otsitakse saadud failist pikimat ühist alamsõnet alustades võrdlemist kõrgeimast tasemest, milles on vähim numbreid. Niipea kui algoritm leiab vastavuse, jätkab see üks tase madalamalt, et veenduda, et tegemist pole vale positiivse tulemusega. Kõige madalamal tasemel, mille numbrid tähistavad esialgseid tähemärke, kontrollitud koodilõikudest eemaldatakse liiga lühikesed ja ülejäänud

valitakse kloonideks. Selleks, et lubada väiksemaid erinevuseid, näiteks muutujanimede vahetamiseid, võimaldab programm jätta väikese vahe kahe kattuva koodisegmendi vahele. Need segmendid seotakse seejärel üheks klooniks kokku.

4. Neljandas etapis koostatakse HTML aruanne inimkontrolliks.

Rakenduse testimiseks otsiti kloone Linuxi kernelist, mille 141 368 koodirida sisaldab 3,6 miljonit tähemärki. Kogu analüüsi protsess võttis aega 15 minutit, kasutades seitset erinevat joondust. Kuigi rakendus ise kasutab ainult ühte protsessori tuuma, siis annab lahendust lihtsasti mitmele tuumale skaleerida ning teoorias töötaks see oluliselt kiiremini graafikakaardi peal. Võrreldes teiste tekstipõhiste kloonide tuvastustööriistadega NiCad 3.5 [CR11] ja Simian 2.3.35 [Sim16] oli antud tööriist küll aeglasem, aga suutis leida rohkem kloone. Tabelis 1 on kajastatud nende kolme tööriista võrdlus, valimistulemuste programmi analüüsimisel, mis sisaldab 11460 rida koodi.

Tabel 1: Tekstipõhiste kloonituvastustööriistade võrdlus

Tööriist	**Kulunud aeg**	**Leitud kloonide arv**
Lainikud	896,7s	1449
NiCad	19,3s	152
Simian	5,5s	262

Võrreldes allpool kirjeldatud keerukamate kloonituvastusmeetoditega on puhtalt tekstipõhiste tööriistade täpsus ja tundlikkus madalam, samas on see tehnika täiesti keelest sõltumatu. Sel viisil õnnestub leida isegi kloone mitmekeelsetest programmidest, näiteks SQL lausetest, mis on kirjutatud Java koodi.

3.2 Lekseemipõhine kloonide tuvastamine

CCFinder (Code Clone Finder) [KKI02] on lekseemipõhine kloonituvastustööriist, mis on algselt disainitud väga suurtest ja vanast infosüsteemidest koodikloonide otsimiseks. See süsteem sisaldab ligikaudu miljon rida koodi ning on kirjutatud keeltes COBOL ja PL/I. CCFinder'i töövoog koosneb kolmest etapist:

1. Esimeses etapis teisendatakse sisendkood lekseemide jadaks.

2. Teises etapis tõlgitakse see jada ühtsele, sisendkeelest sõltumatule, kujule. Komponent koosneb keelepõhistest teisendusreeglitest, näiteks C++ keeles liigutatakse if-lause keha alati ploki sisse nagu näidatud joonisel 1. Samuti asendakase selle etapi käigus kõik identifikaatorid, muutujad ja konstandid spetsiaalsete lekseemidega, mis lubavad ka erinevate nimetuste korral koodisegmentidel kattuda. Hetkel toetab CC-

```
if (a == 1) b = 2; ⇒ if (a == 1) {b = 2};
```

Joonis 1: If-lause keha tõstmine ploki sisse

Finder keeli C, C++, Java ja Cobol. Vastav teisenduskomponent on küllaltki väike ja uue lähtekeele lisamiseks piisab ainult selle komponendi juurde kirjutamisest.

3. Kolmandas etapis otsitakse töödeldud lekseemisõne kõikidest alasõnedest kattuvaid kloonipaare. Selleks kasutatakse sufiksipuudel põhinevat algoritmi [Gus97], mille mälu ja ajaline keerukus on mõlemad $O(m*n)$, milles m on maksimaalse pikkusega klooni lekseemide arv ja n on kogu failis olevate lekseemide arv.

Lahenduse optimeerimiseks rakendas autor järgmiseid meetodeid:

1. Lekseemijadade joondamine - kuna programmi lausete keskel esinevad kloonid pole kuigi kasulikud, lubab süsteem kloonil alata ainult kindlatel lekseemidel. Sellisteks lekseemideks on näiteks plokki alustav lekseem, semikoolon, valikulause (if, else, switch) ja deklaratsioon (class, enum, typedef). Kuigi see piirang võib algoritmi tundlikkust vähendada, väheneb võimalike kloonide alguspunktide arv kahe kolmandiku võrra.

2. Korduva koodi eemaldamine - mõne programmeerimismustri, näiteks valikulausete korral esineb sama koodi mitu korda järjest. Niipea kui sama kood on teist korda kordunud, saab selle ära välja võtta ning eemaldada ka kõik ülejäänud kordused.

3. Suurte failide tükeldamine - niipea kui lähtefailid ei mahu enam arvuti mälusse ära, rakendab tööriist "jaga ja valluta"lähenemist. Kuigi antud meetod on ruutkeerukusega, on see praktikas efektiivne ka suurte koodifailide jaoks, kuni 10 miljonit koodirida.

Süsteemi testimine viidi läbi Pentium III 650MHz ja 640MB RAM arvuti peal analüüsides Java Development Kit versiooni 1.3.0, kus oli umbes 570 000 koodirida jaotatud 1877 faili vahel. Analüüs võttis aega umbes kolm minutit ja tuvastas 2333 klooniklassi, millest pikim sisaldas 627 rida koodi. Katse käigus testiti ka seda, kuidas teisendusreeglid, parameetrite asendamine ja korduva koodi eemaldamine tulemust mõjutavad ning selgus, et ilma nendeta suutis süsteem tuvastada ainult pooled esialgselt leitud kloonidest. CCFinder'it on kasutatud ka kohtuasja lahendamises, kus üks tarkvara ettevõte süüdistas teist illegaalses koodi varguses. Mõlemad süsteemid sisaldasid üle 100 rea C koodi, ning analüüsi tulemusena leiti, et rohkem kui 50% koodist kattub. Need tulemused esitati ka asitõenditena kohtusse.

3.3 Abstraktsel süntaksipuul põhinev kloonide tuvastamine

Kui lekseemipõhised kloonituvastustehnikad on küll kiired, siis vähene arusaamine programmi süntaksist võib tekitada palju valepositiivseid tulemusi. Seetõttu on arendatud mitmeid süntaksi põhiseid meetodeid. Võrreldes abstraktseid süntaksipuid (AST), saab lihtsasti välistada ainult leksilised kloonid, näiteks sellised, mis algavad ja lõppevad programmi lause keskelt.

Süntaksipuust kloonide otsimine on oma olemuselt sama, mis iga alampuu võrdlemine kõigi teiste alampuudega sellest süntaksipuus. Naiivne lahendus ilmselgelt ei skaleeru, sest vastavate võrdluste tegemine on $O(N^2)$ keerukusega, kus N on abstraktse koodipuu tippude arv. Baxter [BYdM+98] on lahendanud oma süsteemis selle probleemi jagades alampuud B ämbrisse selliselt, et võrrelda oleks vaja ainult sama ämbri sees olevaid alampuid. Kuigi ka sellisel juhul on ühe ämbri piires nende võrdluste tegemine $O(N^2)$ keerukusega, siis praktikas läheneb see konstandile, kui B on valitud N-ga samas suurusjärgus. Seda silmas pidades on praktikas kogu analüüsi ajaliseks keerukuseks hinnanguliselt $O(N)$. Peamine küsimus seisneb selles, et kuidas valida ämbritesse jaotamise

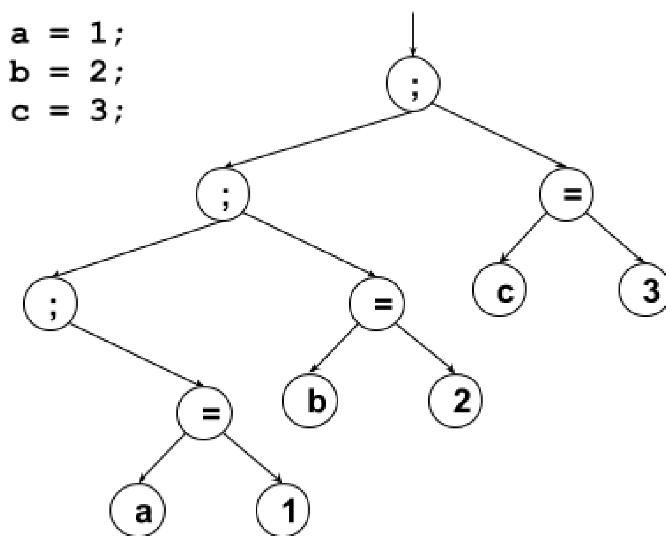
funktsioon selliselt, et ei jääks liiga palju kloone avastamata, sest tavapärased räsifunktsioonid võimaldaksid leida ainult üks-üheseid vastavusi. Baxter ja teised kasutas selleks tehislikult halba räsifunktsiooni. Nad panid tähele, et hägusad kloonid esinevad enamasti vaid abstraktse koodipuu lehtedes tehtud muudatuste poolest. Sellele tuginedes koostasid nad räsifunktsiooni, mis ignoreerib väikeseid alampuid ja kui muude tippude poolest puud kattuvad, siis paneb need samasse ämbrisse. Sama ämbri sees olevate alampuude võrdlemiseks defineeris Baxter sarnasuse järgnevalt:

$$Sarnasus = \frac{2 * S}{2 * S + L + R}$$

Milles:

- S - ühiste tippude arv;
- L - erinevate tippude arv esimeses alampuus;
- R - erinevate tippude arv teises alampuus.

Kuigi antud meetod töötab hästi puude hulgast kloonide leidmiseks, on programmeerimiskeeltes palju jadastruktuure, näiteks programmi laused ja funktsiooni argumendid. Sellised struktuurid väljenduvad abstraktses süntaksipuus enamasti vasakule või paremale kalduvate puudena, mis tekivad parsimise käigus vasak- või paremrekursiivsete produktsioonireeglite tõttu. Iga omistamisoperatsioon joonisel 2 on sõltuv eelmisest, seetõttu on tarvis teistsugust tehnikat nende sarnasuse hindamiseks



Joonis 2: Kolmele omistamislausele vastav abstraktse süntaksipuu struktuur

Baxteri jadastruktuuridest kloonide tuvastamise algoritm võrdleb omavahel kõiki jadatippe sisaldavaid alampuu paare, otsides maksimaalse pikkusega jada, mis moodustab kloonipaari. Kui tavaline algoritm raporteeriks iga kattuva alampuu eraldi kloonipaarina, siis jadastruktuurides neelavad suuremad kloonid väiksemad alla. Algoritm ei suuda aga tuvastada kloone jadadest, millesse on lauseid lisatud või eemaldatud.

Peale esialgset kloonide tuvastamist vaadatakse tuvastatud kloonide vanemaid ning kui vanem osutub samuti klooniks, siis lisatakse kloonide kompositsioon uueks klooniks ning alamkloonid kustutatakse. Sel viisil laiendab algoritm kloonide kattuvust veel kaugemale.

Testimiseks rakendati süsteemi 7 aastat vanale protsessi juhtimissüsteemile, milles oli umbes 400 000 rida C koodi. Analüüsi tulemusena leiti, et keskmiselt 12,7% kogu koodist moodustasid kloonid. Enamus kloonidest olid väikesed, suurusjärgus 10 koodirida, ning kloone, mis sisaldasid rohkem kui 25 rida koodi esines väga vähe. Kuigi tööriist suutis analüüsida 100 000 koodirida alla kahe tunni, osutus peamiseks pudelikaelaks mälu kasutus, sest algoritm hoiab kõiki ämbreid mälus. Probleemi lahendamiseks jagati projekti kood 100 000 realisteks juppideks, mis mahtusid kenasti 600Mb vaba mälu sisse ära.

3.4 Süntaktilisel sufiksipuul põhinev kloonide tuvastamine

Räsifunktsiooni kasutamine aitab küll koodisegmentide võrdlemiste arvu vähendada, kuid samas lubab see hägusates koodikloonides erinevuseid ainult väikestes alampuudes ning samasse ämbrisse võib sattuda palju kloonikandidaate, mis tähendab, et halvimal juhul on algoritmil ikkagi ruutkeerukus. Teisest küljest lekseemipõhised tööriistad, nagu CCFinder töötavad pea-aegu alati lineaarse kiirusega, aga raporteerivad sageli ebavajalikke kloone, näiteks selliseid, mis algavad ja lõppevad erinevates funktsioonides. Rainer [KFF06] täiendas abstraktsel süntaksipuul baseeruvat kloonituvastusalgoritmi kombineerides seda sufiksipuud kasutava algoritmiga, mida kasutatakse enamasti lekseemipõhiste lähenemiste puhul. See meetod koosneb neljast etapist:

a) programmi parsimine ja abstraktse süntaksipuu genereerimine;

b) abstraktse süntaksipuu serialiseerimine lekseemide jadaks;

c) sufiksipuul baseeruva kloonituvastusalgoritmi rakendamine;

d) tulemuseks saadud lekseemide jada taastamine süntaktilisteks osadeks.

Kui esimene etapp on sarnane teistele süntaksipõhistele tööriistadele, siis teises etapis läbitakse see puu eesjärjestuses ning koostatakse uus lekseemide jada, mida on täiendatud selliselt, et see hoiaks puu süntaktilist struktuuri. Näiteks joonisel 3 kujutatud näiteprogrammi põhjal koostatakse joonisel 4 kujutatud serialiseeritud lekseemijada. Alanumbrid iga lekseemi järel näitavad, mitu järglast on vastavas alampuus kokku.

```
if c + y then a := j; else foo; end if;
if p     then a := j; else foo; end if;
if q     then x := k; else bar; end if;
```

Joonis 3: Näitekood if lausetest keeles Ada

Kloonide tuvastamiseks on tööriistas kasutatud sufiksipuul baseeruvat Ukkoneni algoritmi [Ukk95]. Selles etapis ignoreeritakse identifikaatorite nimesid, neid kasutatakse hiljem, et eristada esimest ja teist tüüpi kloone.

Viimases etapis koostatakse kloonidele vastavate lekseemide jadadest uuesti süntaktilised ühikud. Analüüsides joonisel 3 olevat programmi tuvastab tööriist kaks kloonigruppi mida on kujutatud joonistel 5 ja 6. Neist esimene tuvastati lähtekoodi ridadest 2 ja 3 ning

```
seq23
   if8 +2 id0 id0 =2 id0 id0 call1 id0
   if6 id0 =2 id0 id0 call1 id0
   if6 id0 =2 id0 id0 call1 id0
```

Joonis 4: Joonisel 3 kujutatud näitekoodile vastav serialiseeritud lekseemide jada

on juba ise süntaktiline ühik. Teine kloon leiti kõigist kolmest koodireast, aga tegemist pole süntaktilise ühikuga, mistõttu rakendatakse sellele dekompositsiooni, millega jagatakse see kolmeks süntaktiliseks ühikuks mida on kujutatud joonisel 7. Neid ühikuid kasutades saab klooni vastandada abstraktse koodipuuga, et sealt täpne kloon üles leida.

```
   if6 id0 =2 id0 id0 call1 id0
```

Joonis 5: Joonisel 3 kujutatud näitekoodile vastav kloonigrupp 1

```
id0 =2 id0 id0 call1 id0
```

Joonis 6: Joonisel 3 kujutatud näitekoodile vastav kloonigrupp 2

Tööriista testimiseks kasutati Belloni testi [Bel02], mis on tuntud kvantitatiivne meetod koodiklooni tuvastamise tööriistade hindamiseks. Süntaktilise sufiksipuul baseeruv meetod leidis 71% rohkem kloone võrreldes CCFinder'iga ja umbes 50% rohkem kloone võrreldes teiste abstraktse koodipuul baseeruvate meetoditega. Jõudlustest viidi läbi Inteli 64-bitise ja kahetuumalise (3.0GHz) protsessori ning 16 GB mäluga arvutil. Tööriist, mis kasutab ainult ühte tuuma analüüsis PostgreSQL andmebaasi, mis sisaldas 253 000 koodirida, 2 tundi ja 45 minutit. Kokkuvõttes on see analüüsimeetod jõudluse poolest võrreldav lekseemipõhiste lahendustega, kuid leiab paremini tüüp 2 kloone.

3.5 Semantikapõhine kloonide tuvastamine

Koodi struktuuri muutvad muudatused teevad kloonide tuvastamise keerulisemaks. Sellisteks muudatusteks on peamiselt:

- muudatused, mis on tehtud avaldistes ja muutujates;
- osa koodi on lisatud või eemaldatud;
- osa koodist on liigutatud teise kohta.

Selleks, et leida koodikloone, milles on neid muudatusi tehtud, peavad süntaksi põhised kloonide tuvastajad tegema kompromissi täpsuse ja tundlikkuse vahel. Mitteidentsete koodikloonide leidmiseks on nendes tööriistades ignoreeritud teatud tunnuseid, näiteks muutujate nimesid. See aga võib anda valepositiivseid tulemusi. Krinke on koostanud programmi semantikal põhineva lähenemise [Kri01], mis sedasorti järeleandmisi tegema ei pea. Tööriist kajastab programmi täiendatud programmi sõltuvusgraafi abil, mis leiab sellest sarnased alamgraafid ja vastandab need hiljem esialgsele koodile, et seda kasutajale presenteerida.

```
id0, =2 id0 id0 and call1 id0
```

Joonis 7: Süntaktilised ühikud, mis saadi joonisel 6 kujutatud klooni dekompositsiooni tulemusena

Traditsiooniline programmi sõltuvusgraaf (PDG) [HRB90] on suunatud märgendatud graaf. Graafi tipud väljendavad väärtustamisoperatsioone ja kontroll predikaate. Servad väljendavad seoseid programmi komponentide vahel ning on jaotatud kontrolli ja andmete servadeks. Krinke täiendatud sõltuvusgraafi tipud saab peaaegu üksüheses vastavuses tavalise sõltuvusgraafi tippudega, kuid sellele on lisatud eriotstarbelised kohesed kontrolli servad, mis valideeritakse enne teisi. Samuti jagunevad seal andmete servad omakorda väärtustele vastavateks servadeks ja viidetele vastavateks servadeks ning need erinevad selle poolest, et viidetele vastavate servade väärtuseid hoitakse muutujates.

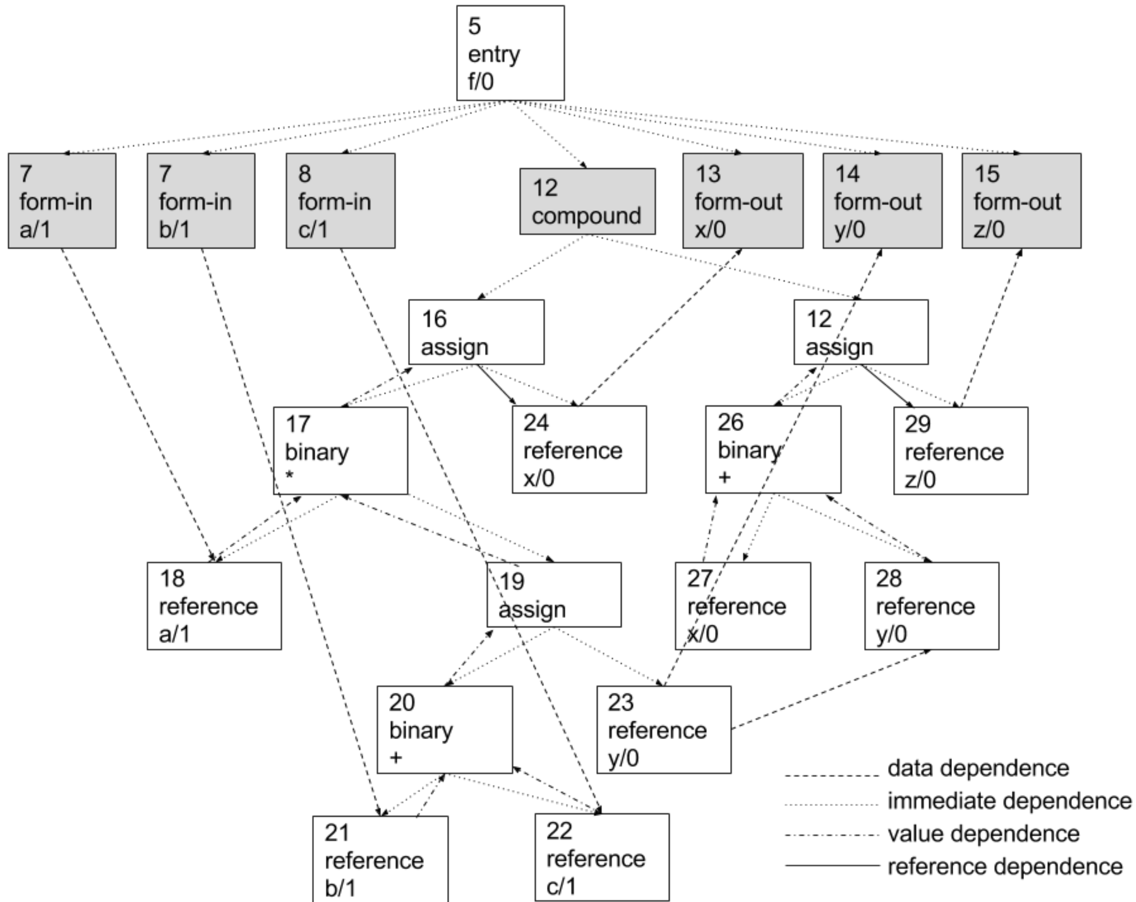
```
void f (int a, int b, int c) {
  x = a * (y = b + c);
  z = x + y;
}
```

Joonis 8: Näitekood sõltuvusgraafi jaoks

Joonisel 9 on kujutatud joonisel 8 olevale lähtekoodile vastavalt täiendatud sõltuvusgraafi. Graafis on märgitud sisenemistipp 5. Tipud 6, 7 ja 8 on formaalsed parameetrite sisenemise tipud ja tipud 13, 14 ja 15 on formaalsed parameetrite väljundtipud. Tipp 12 on liigendtipp, mis grupeerib programmi alamgraafid kokku. Kõik ülejäänud tipud moodustavad programmi avaldised. Sellise kujutuse põhjal saab lihtsasti leida sarnaseid või identseid tippe ja kaari.

Kahte graafi loetakse isomorfseks, kui saab leida bijektiivse vastavuse nende graafide servadel ning servade tüübid ja vastavate tippude märgendid on samad. Graafide isomorfsuse kontrollimine on tuntud NP-täielik probleem, mis tähendab, et ei leidu polünoomse keerukusega algoritmi selle lahendamiseks. Veelgi enam, antud olukorras on tarvis hinnata graafe mitte isomorfsuse vaid sarnasuse põhjal. Krinke definitsiooni järgi on kaks graafi G ja G' sarnased, kui igale teekonnale $v_0, e_1, v_1, e_2, \ldots, e_n, v_n$ ühes graafis leidub teekond $v'_0, e'_1, v'_1, e'_2, \ldots, e'_n, v'_n$ teises graafis nii, et servade tüübid ja tippude märgendid oleksid nende vastandamisel samad. Samuti peavad kõik need teekonnad algama samast tipust $v \in G$ ja $v' \in G'$, nii et kõigi teekondade puhul kehtib $v_0 = v$ ja $v'_0 = v'$.

Naiivne algoritm arvutaks kõik tsükliteta teekonnad, mis algavad tippudest v ja v' ning kontrolliks paarikaupa nende kokkulangemist. Krinke algoritm leiab maksimaalsed sarnased alamgraafid induktsiooni abil alustades algtippudest v ja v' ning piirates induktsiooni sügavuse maksimaalse teekkonna pikkusega k. Algoritmi teeb mõistlikuks see, et see oskab proovida kõiki võimalikke servi samaaegselt. Sellegi poolest on isegi optimeeritud kujul selle algoritmi keerukuseks $O(|V|^2)$. Selleks, et algoritm töötaks mõistliku ajaga, tuleb proovimiseks valida ainult alamhulk kõikidest tippudest eeldades, et ülejäänud tipud läbitakse teekonna ehitamise käigus. See, kuidas need tipud valida, on aga rakenduse spetsiifiline, näiteks üheks võimaluseks on valida nendeks tippudeks ainult funktsioonide sisenemistipud, joonisel 9 tähistatud numbriga 5, mis aga võimaldab leida ainult sarnaseid funktsioone. Krinke otsustas aga kasutada predikaattippe, mis annavad parema granulaarsuse ning seega leitakse iga predikaattipu kohta maksimaalse pikkusega



Joonis 9: Joonisele 8 vastav täiendatud sõltuvusgraaf

sarnased alamgraafid.

Sellise lähenemise probleemiks on see, et mõnikord võivad leitud alamgraafid sisaldada ainult struktuurseid omadus, mida süntaksipõhised tööriistad oskavad juba hästi lahendada. See juhtub siis, kui andmete kaared ei ole vastavuses ning peamised vastavused leitakse vaid kontrollkaarte hulgast. Selle parandamiseks saab määrata igale alamgraafile kaalud sõltuvalt sellest, kui palju on seal andmete kaari.

Tööriista testiti erinevate C-keelsete projektide peal, mille suurused jäid vahemikku 2000 kuni 25 000 koodirida. Kuna süsteem tugineb paarikaupa võrdlusel, siis on sellel hinnanguliselt ruutkeerukus, täpne töötamise aeg sõltub nii sarnaste alamgraafide arvust kui ka suurusest selles programmis. See on ka põhjuseks, miks teekondade võrdlemise algoritmis on maksimaalse teekonna pikkus konstandi k kaudu piiratud. Selle puuduseks on asjaolu, et vastavalt valitud k väärtusele ei saa suuremaid kloone tuvastada. Testimise käigus selgus, et kui väiksemate programmide analüüs töötas kenasti k väärtustega kuni 100, siis ühe programmi puhul, mis sisaldas väga palju kloone, oli piiriks 25 ning programmi analüüs võttis juba aega 46 tundi. Positiivsest küljest ei tuvastanud süsteem ühtegi valepositiivset klooni. Samuti on $k = 20$ enamasti piisav hea tundlikkuse saavutamiseks ning näiteks parserigeneraatori Bison lähtekoodi analüüsimiseks kulus selle korral kõigest 47 sekundit.

3.6 Kloonituvastustehnikate kokkuvõte ja võrdlus

Kuigi eelnevalt sai tutvustatud kõigest viit erinevat kloonituvastustehnikat on erinevaid tehnikaid ja tööriistu veel teisigi, kuid need baseeruvad enamasti samadel alustel. Kuigi erinevaid tehnikaid on päris üks-üheselt raske võrrelda, sest paljud omadused sõltuvad konkreetsest realisatsioonist ja tööriista lisavõimalustest, siis kindlad iseärasused joonistuvad iga klassi kohta välja. Eelkõige seisneb see selles, et enamasti mida keerukam on meetod, seda täpsemini ja tundlikumalt ta kloone tuvastab, kuid on aeglasem praktikas. Tabelis 2 on võrreldud eelpool kirjeldatud koodikloonide tuvastamise tehnikatele vastavaid tööriistu. Tabel on koostatud vastavaid tööriistu kajastanud artiklite põhjal ning seetõttu võib neid olla vahepeal täiendatud. Veerus toetatud keeled on toodud programmeerimisekeeled, millest tööriist oskab kloone tuvastada. Veerus granulaarsus on toodud vähim programmeerimiskeele ühik mille täpsusega tööriist kloone raporteerib. Veerus klooni tüübid on kloonide tüübid, mida tööriist tuvastab ja veerus keerukus on toodud algoritmi ajaline keerukus, milles n tähistab koodi pikkust ja m maksimaalse klooni pikkust. Samuti on tabelis 3 võrreldud nende tööriistade peamiseid eeliseid ja tabelis 4 on toodud problemaatilised kloonid, mille tuvastamisega nad toime ei tule.

Kuigi eelnevalt kirjeldatud tehnikatest ainult süntaktilise sufiksipuul baseeruv tehnoloogia kombineeris omavahel kahte erinevat tehnikat, osutus see üsna kasulikuks lähenemiseks, sest sel moel õnnestus vähendada kummagi lähenemise puuduseid. Tulevikus on oodata rohkem hübriidlahendusi, näiteks on välja pakutud analüüsimeetod [BS03], mis võimaldab abstraktsel süntaksipuul põhinevaid kloonituvastustehnikaid andmevoo graafile üle kanda.

Tabel 2: Kloonituvastustööriistade võrdlus

Meetod	**Toetatud keeled**	**Granulaarsus**	**Klooni tüübid**	**keerukus**
lainikud	kõik keeled	tähemärk	tüüp 1 ja 2	$O(n * log(n))$
lekseemid	C, C++, Java, Cobol	süntaktiline ühik	tüüp 1 ja 2	$O(m * n)$
AST	C, C++	süntaktiline ühik	tüüp 1 ja 2	$O(n^2)$
AST sufiks	C	süntaktiline ühik	tüüp 1 ja 2	$O(m * n)$
sõltuvusgraaf	C	funktsioon	tüüp 1, 2 ja 3	$O(n^2)$

Tabel 3: Kloonituvastustööriistade peamised eelised

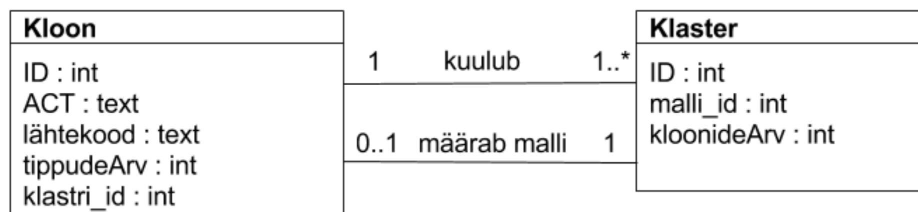
Meetod	**Peamine eelis**
lainikud	keeleneutraalne
lekseemid	sobilik suurte süsteemide analüüsimiseks
AST	leitud kloonid on süntaktiliselt mõistlikud
AST sufiks	AST ja lekseemi kombineeritud eelised
sõltuvusgraaf	leiab kõik kloonid

Tabel 4: Kloonituvastustööriistade jaoks probleemsete kloonide tunnused

Meetod	**Probleemsete kloonide tunnused**
lainikud	pikad tekstilised erinevused
lekseemid	tippe on lisatud, kustutatud või järjekorda muudetud
AST	tippe on lisatud või kustutatud
AST sufiks	tippe on lisatud, kustutatud või järjekorda muudetud
sõltuvusgraaf	pikad kloonid suurtes projektides

4 Soovitusmudeli koostamise töövoog

Käesolevas töös koostatav soovitusmudel koosneb klastritest. Igasse klastrisse kuulub üks või mitu koodisegmenti ning sama koodisegment saab kuuluda ainult ühte klastrisse korraga. Lisaks on igal klastril määratud üks koodisegment malliks, mis tähendab, et see iseloomustab seda klastrit kõige paremini. Klastris olevaid koodisegmente nimetame edaspidi kloonideks, kuna kõik klastris olevad koodisegmendid on üksteisega sarnased. Joonisel 10 on kujutatud vastava andmestruktuuri UML diagrammi.



Joonis 10: Soovitusmudeli andmestruktuur

Soovitusmudeli koostamise moodul saab sisendiks lähtekoodi tekstifaili, mis parsitakse ning transleeritakse abstraktseks koodipuuks, täpsemalt on seda protsessi kirjeldatud järgmises peatükis 5. Saadud puustruktuuri alampuid võrreldakse paarikaupa soovitusmudeli klastritega andmebaasis. Alampuu määratakse klastrisse, mille mall on antud alampuule kõige sarnasem. Juhul kui ükski klaster ei ületa sarnasuse künnist σ, luuakse antud alampuu jaoks uus klaster ning alampuu määratakse selle klastri malliks. Vastav töövoog on kujutatud joonisel 11.

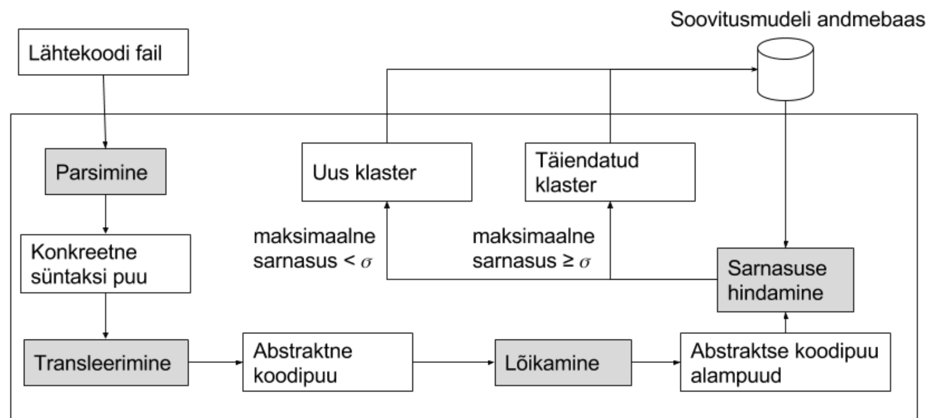
Kuna algoritm hõlmab kõikide alampuude võrdlemist kõikide klastritega andmebaasis, on see ruutkeerukusega. Selleks, et seda protsessi kiirendada, saab välistada klastrid, mille malli tippude arv m on oluliselt erinev võrreldava alampuu tippude arvuga m. Seega on tarvis võrrelda ainult klastritega, mille korral kehtib $min(m, n)/max(m, n) > \sigma$.

4.1 Soovitusmudeli koostamise juhtimine

Selleks, et kasutaja saaks juhtida soovitusmudeli koostamise protsessi, on töö raames loodud rakendusse lisatud järgmised parameetrid:

1. Klastri sarnasus - sarnasuse künnis σ, mis määrab minimaalse sarnasuse väärtuse, mille korral alampuu klastrisse lisatakse.

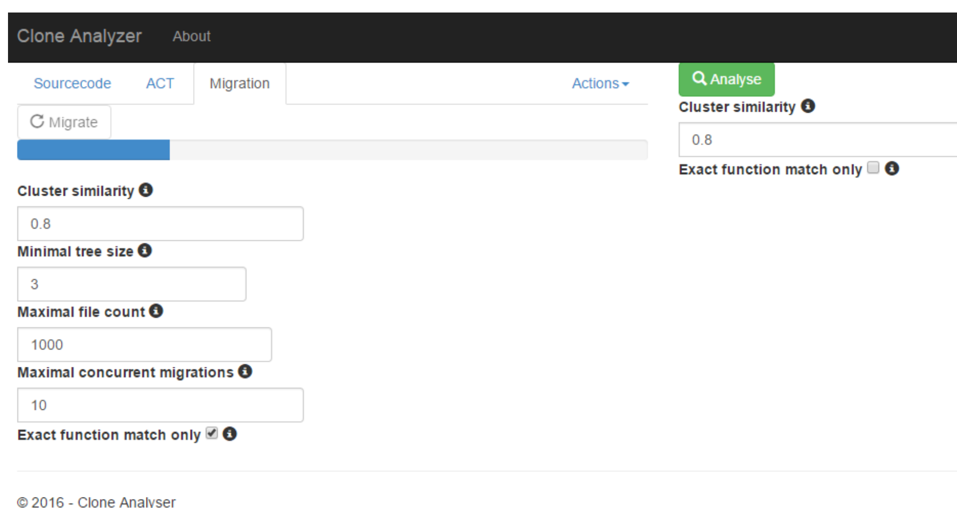
2. Minimaalne puu suurus - minimaalne tippude arv alampuus, mida klastritega võrreldakse.



Joonis 11: Soovitusmudeli koostamise töövoog

3. Maksimaalne failide arv - määrab mitu esimest faili repositooriumite andmebaasist analüüsimiseks võetakse.
4. Maksimaalne samaaegsete migratsioonide arv - samaaegsete migreerimispäringute arv, mille klient korraga välja saadab. Enne uue päringu saatmist ootab klient ära eelneva päringu vastuse.
5. Ainult täpsete funktsioonikattuvuste vastandamine - kui see parameeter on sisse lülitatud, siis loetakse sarnasuse hindamise algoritmis sarnasteks ainult neid sisseehitatud funktsioonide kattuvusi, millel on täpselt sama nimi. Näiteks ei loeta siis sarnaseks funktsioone == ja ===.

Kuna migreerimisele kuuluvaid faile on palju, siis kuvatakse migreerimist alustades kasutajale progressiriba, mis näitab, kui palju faile on jäänud veel migreerida. Rakenduses olevat migreerimiskuva on kujutatud joonisel 12.



Joonis 12: Soovitusmudeli koostamise juhtimise kuva

4.2 Koodisegmendi hoidmine andmebaasis

Selleks, et andmebaasis hoitavat koodisegmenti saaks mõne teise koodisegmendi abstraktse koodipuuga võrrelda, hoitakse klooni tabelis sõneks kodeeritud abstraktset koodipuud. Puu kodeerimisfunktsioon *ToSuccinct* läbib sisendina saadud puu eesjärjestuses ning iga tipu järglased lisatakse selle tipu tunnuse järele sulgudesse. Kui tegemist on mitteterminaliga, millel puudub väärtus, siis lisatakse sõnesse ainult seda tippu tähistav sümbol. Kui tegemist on identifikaatori või mitteterminaliga, millel on väärtus, näiteks funktsiooni deklaratsiooni väärtuses hoitakse selle funktsiooni nime, siis lisatakse sõnesse seda tippu tähistav sümbol, seejärel selle väärtus ja lõpuks semikoolon, mis seda järgnevatest tippudest eraldab. Lisaks on realiseeritud ka funktsioon *FromSuccinct*, mis etteantud sõne põhjal uuesti abstraktse koodipuu koostab.

Andmebaasis hoitakse ka kloonile vastavat lähtekoodi, mida kasutatakse hiljem staatilise analüüsi käigus soovituse tegemiseks. See kood on aga rekonstrueeritud konkreetsest koodipuust kahel põhjusel. Esiteks saab niimoodi konstrueerida lähtekoodi mistahes abstraktse koodipuu alampuu jaoks. Teiseks pole andmebaasis vaja hoida liigseid tühimikke ja kommentaare, mis moodustavad umbes kolmandiku lähtekoodi pikkusest.

Samuti hoitakse iga klooni kohta tema abstraktse koodipuu tippude arvu, see võimaldab kiiresti hinnata kas võrreldavad kloonid on samas suurusjärgus ning vältida asjatut puu taastamist. Ülejäänud abstraktse koodipuu kohta käiv info, nagu alampuude sügavused ja muutujate kontekstid genereeritakse abstraktse koodipuu pealt ning seda infot pole vaja andmebaasis eraldi hoida.

5 Lähtekoodi parsimine ja transleerimine

Lähtekoodi süntaktilise analüüsi ehk parsimise jaoks on töös koostatud JavaScripti parser kasutades parserigeneraatori ANTLR (ANother Tool for Language Recognition) neljandat versiooni [ANT16a]. ANTLR võimaldab arendajal spetsifitseerida keele lekseri ja parseri grammatikad ning genereerib kiire $LL(*)$ parsimismeetodit kasutavad lekseri ja parseri.

Võrreldes varasemate ANTLR-i versioonidega on ANTLR nelja erinevuseks see, et tema poolt genereeritud parserid tagastavad programmi konkreetse süntaksipuu, mis vastab üks-üheselt keele grammatika reeglitele. See tähendab, et kui ANTLR-i kolmandas versioonis oli tarvis abstraktse süntaksipuu saamiseks grammatikat vastavate puu struktureerimise reeglitega märgendada, siis versioon neljas seda võimalust ei ole. Selle asemel on ligipääs konkreetsele süntaksipuule ning selle modifitseerimiseks saab arendaja ise kirjutada puu läbimise algoritmi. Sellel lähenemisel on kolm olulist plussi:

1. Esiteks jääb grammatika puhtam ja loetavam. Seetõttu on ka sobiva keele grammatikat lihtsam leida, näiteks on ANTLR-i GitHub'is [ANT16b] üle 80 erineva keele grammatika, millest enamik on vabavaralised ja neid saab oma programmides kasutada. Käesolevas töös on kasutatud ECMAScript'i grammatikat [Ecm16b] JavaScripti parsimiseks, mis põhineb ECMA-262 5.1 standardil [ecm16a] ja on MIT litsentsi alusel kasutatav.

2. Teiseks on konkreetsel süntaksipuul olemas vastavus esialgsesse lähtekoodi, seetõttu saab sealt kätte koodilõikude rea- ja sümbolinumbrid ning vajadusel saab süntaksipuu ka tagasi koodiks muuta.

3. Kolmandaks on eraldi kirjutatud transleerimise algoritm tunduvalt võimekam kui märgendatud grammatika põhjal genereeritud algoritm. Näiteks saab konkreetses süntaksipuus mitu tippu edasi vaadata ning seeläbi targemaid otsuseid teha.

Töö käigus loodud analüsaator eeldab, et rakenduse sisendiks antud kood on süntaktiliselt korrektne. JavaScripti parser oskab küll süntaksi vigasid tuvastada, kuid nende parandamisega süsteem ei tegele. Kui andmete migratsiooni käigus leitakse süntaksiviga, jäetakse selle faili võrdlemine pooleli ja analüüsi jätkatakse järgmisest failist, mida õnnestub parsida. Kui kasutaja sisestab süntaksiveaga lähtekoodi, siis jäetakse see analüüsimata.

5.1 Konreetse süntaksipuu transleerimine abstraktseks koodipuuks

EcmaScripti grammatikas on kokku 76 erinevat parsimise produktsioonireeglit. Kuna paljusid neist saab väljendada läbi teiste reeglite, näiteks for-tsükli asemel saab kasutada while-tsüklit, siis transleeritakse parsimise järel konkreetne süntaksipuu abstraktseks koodipuuks. Sellisel lähenemisel on mitu eelist:

1. Edasist andmevoo analüüsi on märgatavalt lihtsam teha, sest erinevaid konstruktsioone ja erandjuhte on vähem.

2. See lähenemine vähendab erinevate programmeerimise stiilide mõju, kuna vahekeeles puudub palju süntaktilist suhkrut.

3. Ühtne vahekeel võimaldab tööriista kohandada lähtekeelele sobivaks, selleks on vaja koostada ainult uus transleerimise komponent.

Töös kasutatav abstraktse koodipuu struktuur on inspireeritud Nielson'i While keelest [NN07] ning selle grammatikat on kujutatud joonisel 13. Kuigi grammatikale vastav koodipuu on tunduvalt kompaktsem, kui konkreetne süntaksipuu, sisaldab see siiski mitteterminale, mida on tarvis grammatika kirjeldamiseks, kuid ei ole programmi seisukohalt olulised. Seetõttu ei lisata parsimise ajal abstraktsesse koodipuusse mitteterminale "lause", vaid selle järglased lisatakse kohe eelneva tipu alla. Samamoodi eemaldatakse hiljem plokid, mis asuvad vahetult mõne teise ploki sees, millel järglased puuduvad, või on ainult üks järglane. Need optimeering vähendab puu tippude arvu veel ligikaudu kaks korda.

```
<program> ::= <stmt>*
<stmt> ::= <block> | <if> | <while> | <assign> | <call>
| <trycatch> | <throw> | <return> | <continue>
| <break> | <function> | label | term
<block> ::= <stmt>*
<if> ::= <stmt> <block> <block>
<while> ::= <stmt> <block>
<assign> ::= term <stmt>
<call> ::= term <block>
<trycatch> ::= <block> <block>
<throw> ::= <stmt>
<return> ::= <stmt>?
<continue> ::= label?
<break> ::= label?
<function> ::= <block> <block>
```

Joonis 13: Abstraktse koodipuu EBNF grammatika

Tipud "label", "term"ja "function"saavad endale väärtuseks vastavalt konstandi, muutuja või funktsiooni nime. See tipu sisse ja kasutatakse hiljem vastandamisel.

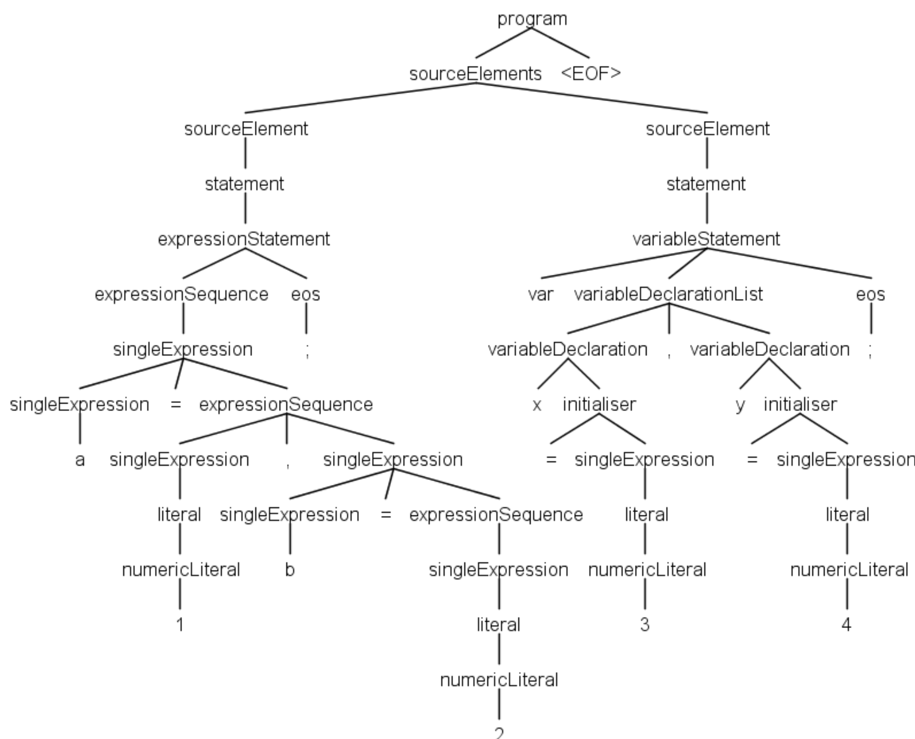
Lisaks suurele mitteterminalide arvule on konkreetse süntaksi grammatikas ka erinevaid konstruktsioone, mis on vajalikud parsimiseks ja süntaksivigade tuvastamiseks, kuid teevad süntaksipuu analüüsimise keeruliseks. Listingul 14 on kujutatud JavaScripti programmi, mis algväärtustab neli muutujat: a ja b globaalselt ning x ja y lokaalselt. Kuigi süntaks ja tähendus on mõlemat tüüpi väärtustamisel peaaegu sama, on tekkivad alampuud hoopis erineva struktuuriga, mida on kujutatud joonisel 15. Võtmesõna *var* kasutamine viitab grammatikas reeglile *variableDeclarationList* ning kõik järgnevad väärtustamised lisatakse selle reegli võrdseteks järglasteks. Seevastu võtmesõna *var* puudumise korral on tegemist reegliga *expressionSequence* ning selle struktuur määrab iga järgneva väärtustamise uuele sügavusele. Joonisel 16 on kujutatud samale sisendile vastav abstraktne koodipuu ning sealt on näha, et kõik neli väärtustamist on viidud samale tasandile ning alles on jäänud vaid minimaalne arv tippe, millega saab seda süntaksit edasi anda.

```
a = 1, b = 2;
var x = 3, y = 4;
```

Joonis 14: Muutujate väärtustamine JavaScriptis

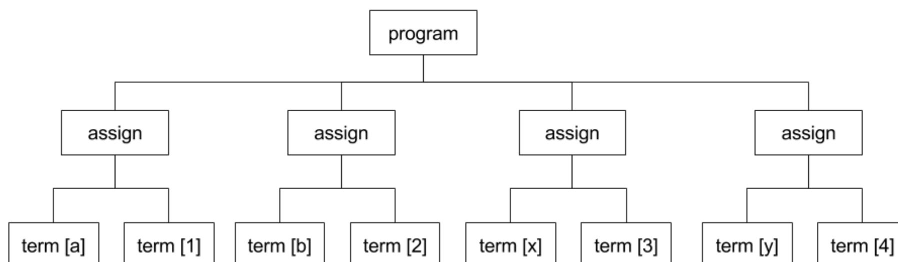
5.2 Valikulause transleerimine

Kui for-tsüklit saab teisendada while-tsükliks ilma semantikat muutmata, siis pole see kõigi programmeerimise konstruktsioonide puhul nii lihtne. Üheks selliseks on valikulau-



Joonis 15: Joonisele 14 vastav konkreetne süntaksipuu

se (*switch*), mis teisendatakse *if* konstruktsioonideks. Enamikel juhtudel on kõik valikud (*case*) teineteist välistavad ning ning seetõttu teisendab süsteem iga valiku eraldi *if* konstruktsiooniks ning *default* valiku eelnevate eituseks. Kuna erinevad arendajad võivad teineteisest sõltumatuid valikuid kirjutada erinevas järjekorras, võimaldab see lahendus neid hiljem samasugustena käsitleda. Antud lahenduse puuduseks on see, et semantiliselt ei ole need representatsioonid enam alati samaväärsed. Juhul kui mõni valikutest kontrollitavat tingimust muudab, siis tingimuslausetena kirjutades võib eelnev tingimus aktiveerida endale järgneva. Siinkohal ei ole lahenduseks ka *if else* konstruktsiooni kasutamine, sest see välistab ühe valiku võimaluse järgmisesse edasi kukkuda, kui puudub lõpetav käsk *break*. Kuigi ka seda lahendust annab täiustada, näiteks kopeerides kõik järjestikused lõpetamata valikud sama tingimuse alla, ei lahenda ka see täielikult probleemi. Joonisel 17 on toodud näide korrektsest JavaScript'i valikulausest, mida ei saa *if*



Joonis 16: Joonisele 14 vastav abstraktne koodipuu

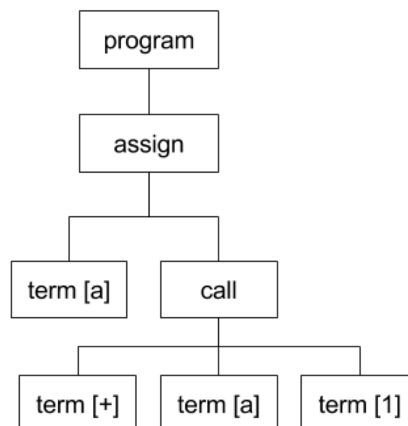
ja *else* konstruktsooni kasutades simuleerida, sest *default* haru ei lõppe ära, vaid kukub järgmisesse valikusse edasi.

```
var a = 3;
switch (a) {
  case 1:
        alert("a=1");
        break;
  default:
      alert("a=3");
  case 2:
        alert("a=2");
        a = 4;
        break;
    case 4:
        alert("a=4");
        break;
};
```

Joonis 17: Case lause, mida ei saa väljendada *if* ja *else* abil

5.3 Avaldiste transleerimine

Enamik programmeerimiskeeli tunneb paljusid sissehitatud operaatoreid, millest arendaja saab oma avaldised konstrueerida, nendeks on näiteks aritmeetika-, loogika- ja sõneavaldised. Iga avaldise jaoks pole erineva abstraktse koodipuu tipu loomine mõistlik. Näiteks ainuüksi omistamisoperaatoreid on JavaScriptis 16, millest 15 muudavad samal ajal ka omistatavat väärtust. Seetõttu käsitletakse abstraktses süntaksipuus erinevaid avaldisi kui funktsioonide väljakutseid. Näiteks avaldis a += 1 tõlgitakse joonisel 18 kujutatud abstraktseks koodipuuks.



Joonis 18: Avaldisele a += 1 vastav abstraktne koodipuu

5.4 Abstraktse koodipuu tippude kontekst

Enamus lekseemi ja süntaksipõhised koodikloonide otsijaid ignoreerivad identifikaatoreid täielikult, et tuvastada teist tüüpi koodikloone. See suurendab aga valepositiivsete kloonide arvu, sest juhul kui nad kasutavad erinevaid muutujaid, või rakendavad erinevaid funktsioone on neil tõenäoliselt ka erinev tähendus. Tippude kontekst muutub aga veelgi olulisemaks, kui on tarvis lubada koodisegmentide ringiliikumist, sest see võimaldab kehtestada liikumisreeglid. Ilma nendeta piisaks vaid elementide loendamisest, mis oleks sarnasuse hindamisel liiga ebatäpne, sest sama elementide arvuga programmid võivad täita hoopis erinevaid ülesandeid. Käesolevas töös on abstraktse koodipuu tipu kontekst defineeritud järgmiselt:

Definitsioon. Abstraktse koodipuu tipu *kontekstiks* nimetatakse muutujaid, mida see tipp või tema järglased loevad või kirjutavad, samuti konstante ja funktsioone, mida selle tipu või tema järglaste poolt kasutatakse.

Seega jagunevad tipu kontekstis hoitavad elemendid kolmeks:

a) muutujad,

b) funktsioonid ja

c) konstandid.

Lisaks jaguneb muutujate kontekst omakorda lugemise ja kirjutamise kontekstideks. Funktsioonid muutujat x lugevate ja kirjutavate tipuhulkade leidmiseks defineerime vastavalt $read(x)$ ja $write(x)$.

Abstraktse koodipuu igale tipule leitakse kontekst läbides see puu lõppjärjestuses. Terminalidele kontekst määramine toimub algoritmi 1 järgi. Mitteterminalide kontekst saadakse rekursiivselt kõigi selle järglaste kontekstide ühendina, erandiks on ainult funktsiooni deklaratsioon, mis oma järglaste konteksti edasi ei kanna. Abstraktse koodipuu jaotamisest funktsioonidesse on lähemalt kirjutatud järgmises peatükis 5.5.

Algorithm 1: Terminali konteksti määramine

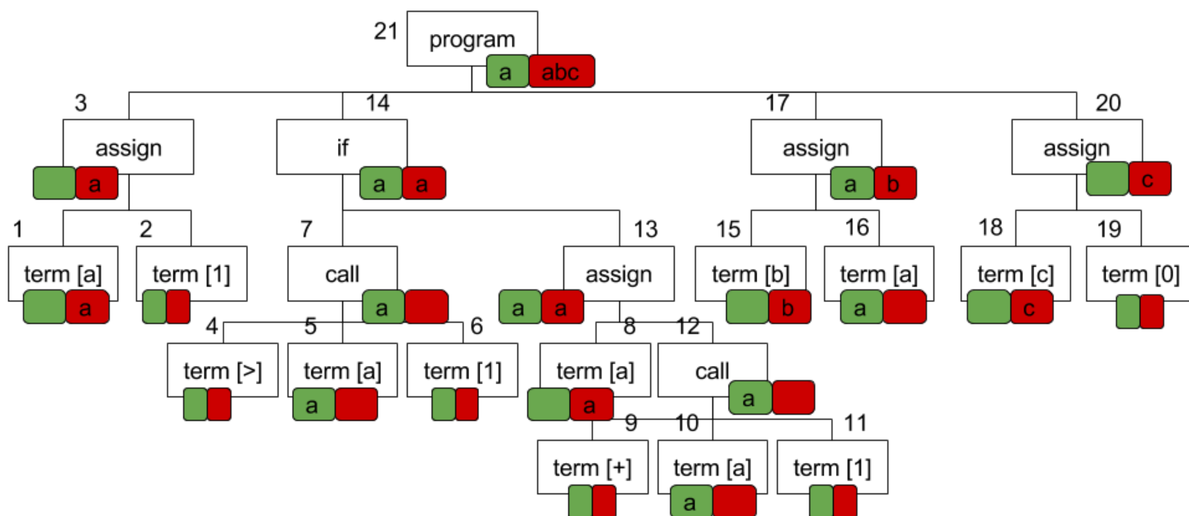
Input: Terminal
Result: Terminalile on määratud kontekst

```
1  if terminal on omistamise vasakus pooles then
2      terminal.muutujaKirjutamiskontekst ← terminal.tekst;
3  else
4      if sama terminal on mõne teise omistamise vasakus pooles then
5          terminal.muutujaLugemiskontekst ← terminal.tekst;
6      else
7          if terminal on funktsiooni vasakus pooles then
8              terminal.funktsioonikontekst ← terminal.tekst;
9          else
10             terminal.konstandikontekst ← terminal.tekst;
```

Joonisel 19 on toodud näiteprogramm kolme muutujaga ning joonisel 20 kujutatud on talle vastav abstrakne koodipuu, kus muutujate lugemiskontekst, on tähistatud rohelisega ja kirjutamiskontekst punasega.

```
a = 1;
if (a > 3)
  a++;
b = a;
c = 0;
```

Joonis 19: Konteksti määramise näitekood



Joonis 20: Joonisele 19 vastav abstraktne koodipuu muutujate kontekstiga

Joonisel 21 on näidatud abstraktset koodipuu kujutamist rakenduses. Selle mitteterminalidele on lisatud neile vastavad kontekstid kujul "[lugemise kontekst|kirjutamise kontekst]". Selle puu abil saab kasutaja hinnata, kui hästi sai rakendus hakkama tema sisestatud lähtekoodi transleerimisega, ning on seetõttu kasulik tööriist analüsaatori arendamisel.

5.5 Abstraktse koodipuu jagamine funktsioonideks

Analüüsile sisendiks antud programm võib omakorda sisaldada mitmeid alamprogramme, milleks on JavaScripti puhul funktsioonide deklaratsioonid. Kuna ANTLR parsib kogu lähtekoodi üheks süntaksipuuks, on mõistlik need ka abstraktsesse koodipuusse eraldi alampuudena lisada. Ometi on need tipud erinevad teistest tippudest, kuna nende sisemuses olev kontekst ei ole seotud ülejäänud programmi kontekstiga. Samuti on hea, kui funktsiooni väljakutsel saab seda seostada konkreetse funktsiooni definitsiooniga.

Abstraktse koodipuu eraldamine funktsioonideks on realiseeritud eraldi järeltöötlusfunktsioonina *ExtractFunctions*, mis läbib abstraktse koodipuu rekursiivselt ning määrab funktsioonide deklaratsioonid etteantud sõnastikku. Sõnastikku ei panda koodifaili põhifunktsiooni, sest selle poole ei saa teised funktsioonid pöörduda.

Sourcecode ACT Migration Actions

```
0 Program  [c_p1,c_p2|]
1   Function myFunction [c_p1,c_p2|]
2     Block  [c_p1,c_p2|]
3       p1
4       p2
5     Return  [c_p1,c_p2|]
6       Call * [c_p1,c_p2|]
7         p1
8         p2
```

© 2016 - Clone Analyser

Joonis 21: Abstraktse koodipuu kuva

6 Koodisegmentide sarnasuse hindamine

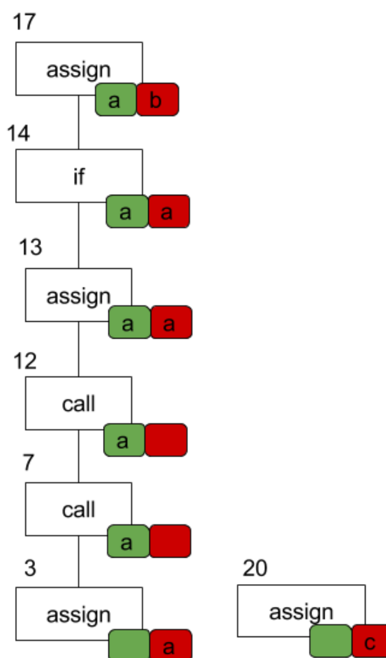
Relatsiooni R nimetatakse järjestuseks, kui see on refleksiivne, antisümmeetriline ja transitiivne ning seda tähistatakse sümbolitega $\preceq$ ja $\leq$. Kui järjestus R on defineeritud hulgal X, siis paari (X, R) nimetatakse järjestatud hulgaks. Relatsiooni R nimetatakse lineaarseks järjestuseks, kui see on järjestus ja iga $x, y \in X$ korral kehtib xRy või yRx. Järjestust, mis võib olla mitte lineaarne, nimetatakse ka osaliseks järjestuseks ning vastavat paari (X, R) osaliselt järjestatud hulgaks [MN08].

Programmi koodi võib vaadelda lineaarse järjestusena $\leq$ kõiki temale abstraktse koodipuu tippe sisaldaval hulgal Π. Elementide $\sigma \in \Pi$ ja $\tau \in \Pi$ vahel kehtib relatsioon $\sigma \leq \tau$ parajasti siis, kui σ läbitakse abstraktse koodipuu keskjärjestuses enne τ, või $\sigma = \tau$.

Koodisegmentide konteksti piires liikumise lubamiseks tuleb esmalt vaadelda, millised elementide vahetamise operatsioonid võivad mõjutada programmi semantikat. Seejuures võib jätta kõrvale tipud "programm", "plokk"ja "term", sest nende asukoht ja olemasolu sõltub alati teistest tippudest. Kõigi ülejäänud tippude hulga tähistame $\Delta \subseteq \Pi$. Programmi semantikat võivad mõjutada sellised elementide vahetamised φ, mille jaoks leiduvad muutuja väärtust muutev element $\alpha \in \Delta$ ja sama muutuja väärtust lugev või kirjutava element $\beta \in \Delta$, mille korral $\beta \preceq \alpha \xrightarrow{\varphi} \alpha \preceq \beta \wedge \alpha \neq \beta$. Seega et säilitada semantikat, ei tohi abstraktse koodipuu elementide järjekorra vahetamise tulemusena sattuda muutuja väärtust kirjutav element ettepoole teisest sama muutuja väärtust lugevast või kirjutavast elemendist. Seega semantikat säilitav järjestusrelatsiooniks nimetame relatsiooni $\preceq$ abstraktse koodipuu tippe sisaldaval hulgal Δ, milles iga kahe elemendi $\sigma, \tau \in \Delta$ korral kehtib $\sigma \preceq \tau$ parajasti siis, kui $\sigma \leq \tau$ ja leidub muutuja x mille korral $x \in read(\sigma)$ ja $x \in read(\tau) \vee write(\tau)$ ning vastupidi.

Kuna semantikat säilitav järjestusrelatsioon ei pruugi kehtida iga kahe abstraktse koodipuu tipu vahel, on nende tippude hulk osaliselt järjestatud hulk ja mistahes lineaarne täiend sellel hulgal on üheks võimaluseks seda programmi kirja panna. Joonisel 22 on näiteprogrammile 19 vastava semantikat säilitava relatsiooni Hasse diagramm. Terminale ja tippu "programm"pole relatsioonis tarvis kujutada, sest need ei saa oma asukohta muuta. Jooniselt on näha, et kõik ülejäänud tipud peale kahekümnenda on antud programmi puhul lineaarselt järjestatud ja nende omavahelist järjekorda seega muuta ei tohi.

Seevastu tipp 20 on täiesti eraldi ja pole relatsioonis ühegi teise tipuga, see tipp võib olla lähtekoodis ükskõik millisel kohal.



Joonis 22: Joonisele 19 vastav Hasse diagramm

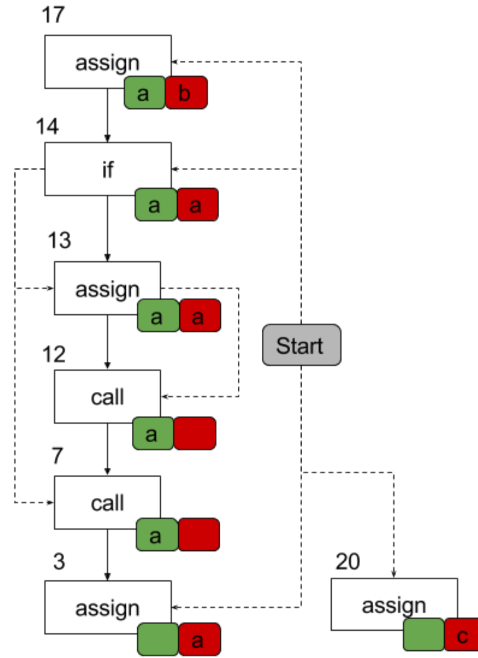
6.1 Abstraktse koodipuu teisendamine graafiks

Osaliselt järjestatud hulga kõige levinumaks kujutamise viisiks on suunatud graaf [Mic09]. Selle graafi $G = (V, E)$ tippude hulgaks V on kõigi programmi mitteterminalide hulk Δ ning kahe tipu $i, j \in V$ vahel on kaar $e = (i, j)$ kui nende vahel kehtib vahetu järgnevusrelatsioon $i \lhd j$, mis tähendab, et kehtib

- $i \preceq j$,
- $i \neq j$ ja
- ei leidu $t \in \Delta$ nii, et $i \preceq t \preceq j$.

Selle graafi hoidmiseks kasutatakse töös külgnevusjärjendeid (Adjacency list), sest sarnasuse hindamise algoritmil on tarvis kiiresti leida etteantud tipu kõiki naabreid. Kuna graaf on suunatud, siis on eraldi defineeritud järjendid sisse tulevate ja välja minevate kaarte jaoks. See võimaldab ka kiiresti leida mistahes tipu i sissetulevate kaarte arvu $id(i)$ ja välja minevate kaarte arvu $od(i)$.

Joonisel 23 on kujutatud näidisprogrammi 19 põhjal konstrueeritud graafi. Järgnevusrelatsioonile vastavad kaared on tähistatud pideva joonega ning puustruktuuri kaared on tähistatud punktiirjoonega. Kuna graafis ei hoita tippu "programm", on joonisele lisatud "start"tipp, mis seob kokku kõik vanemateta tipud.



Joonis 23: Joonisele 19 vastav suunatud graaf

6.2 Semantikat säilitav sarnasuse hindamine

Kui isomorfism on alati üheselt määratud, siis objektide sarnasuse meetrika defineerimiseks on mitmeid võimalusi. Sarnasuse meetrika S peab vastama järgmistele tingimustele:

a) piiratud vahemik, ehk $0 \leq S(X, Y) \leq 1$;

b) refleksiivus, ehk iga elemendi sarnasus iseendaga on alati 1, ehk $S(X, X) = 1$;

c) sümmeetrilisus, ehk $S(X, Y) = S(Y, X)$;

d) transitiivus, ehk $S(X, Z) = S(X, Y) + S(Y, Z)$.

Käesolevas töös kasutatatakse sarnasuse hindamiseks graafide sarnasusmaatriksit. Selle meetodi kohaselt on kaks graafi tippu $i \in V(A)$ ja $j \in V(B)$ teineteisega sarnased, kui nende naabrite vahel leidub sarnane kujutus. Algoritm põhineb universaalsel graafi naabrite võrdlemise algoritmil [Sas16] ning töötab mistahes suunatud graafide A ja B korral. Algoritmi üldkuju on järgmine:

1. Koostatakse $|V(A)| \times |V(B)|$ reaalarvuline maatriks.

2. Iga maatriksi lahter algväärtustatakse nende tipuastmete suhte põhjal järgnevalt:

$$x_{ij} = \frac{m_{in}/n_{in} + m_{out}/n_{out}}{2}.$$

Kus $m_{in} = max(id(i), id(j))$ ja $n_{in} = min(id(i), id(j))$ ning m_{out}, n_{out} on analoogsed.

3. Iteratiivselt parandatakse maatriksi väärtuseid:

$$x_{ij}^{k+1} \leftarrow \frac{s_{in}^{k+1}(i,j) + s_{out}^{k+1}(i,j)}{2}$$

Milles sissetulevate ja väljaminevate tippude sarnasuse saab leida järgnevalt:

$$s_{in}^{k+1} \leftarrow \frac{1}{m_{in}} \sum_{l=1}^{n_{in}} x^k$$

$$s_{out}^{k+1} \leftarrow \frac{1}{m_{out}} \sum_{l=1}^{n_{out}} x^k$$

Väärtus x^k on vastava naabertipu sarnasus eelmisel iteratsioonil. Tavalise suunatud graafi korral on selleks sarnasusmaatriksis olev väärtus. Ülejäänud väärtused m_{in}, n_{in}, m_{out} ja n_{out} on samad, mis eelmises punktis.

4. Kolmandat punkti korratakse seni, kuni maatriks on tasakaalustunud. Töö lõpetatakse, kui viimasel iteratsioonil väärtused enam oluliselt ei muutu.

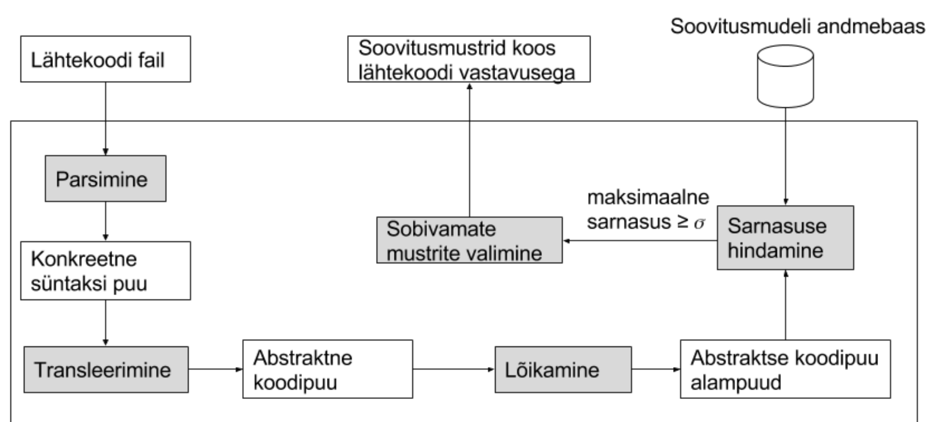
5. Tulemuseks saadud maatriksist valitakse parim tippude vastavus ning võetakse selle aritmeetiline keskmine kogu graafi sarnasuseks.

Kuna abstraktses koodipuus on kõigil tippudel erinev tähendus, siis ei saa omavahel sobitada erinevaid tüüpi tippe. Seetõttu kontrollitakse x^k leidmisel esmalt nende tippude tüüpe, kui need ei kattu, siis $x^k = 0$, alles seejärel võetakse väärtus sarnasusmaatriksist. Kui võrreldavatel tippudel puuduvad kõik naabrid, siis on nende tippude sarnasus 1 [Nik12].

7 Analüüsi töövoog

Kasutaja sisestatud koodi kontrollimiseks soovitusmudeli vastu on kasutusel sarnane protsess soovitusmudeli koostamisele. Lisatud on kasutaja jaoks mugav veebiliides oma lähtekoodi esitamiseks ja täiendamiseks. Veebipõhise lähtekoodi sisestamiseks on kasutatud teksti toimetajat CodeMirror [Cod16], mis võimaldab teksti värvida ja abistab kasutajat selle sisestamisel. Sarnaselt migreerimisele saab ka analüüsi alustades määrata kasutaja klastrite sarnasuse künnise ja kas ta soovib vastandada ainult täpseid funktsioonikattuvusi.

Sisestatud lähtekood läbib parsimise, transleerimise ja lõikamise etapid, kuid seekord on oluline, et igasse alampuusse salvestatakse sellele vastavad algus- ja lõpp-punkt lähtekoodis. Nende abil sobitatakse soovitusmustrid hiljem esialgses lähtekoodis õigetesse kohtadesse. Seda töövoogu on kujutatud joonisel 24.

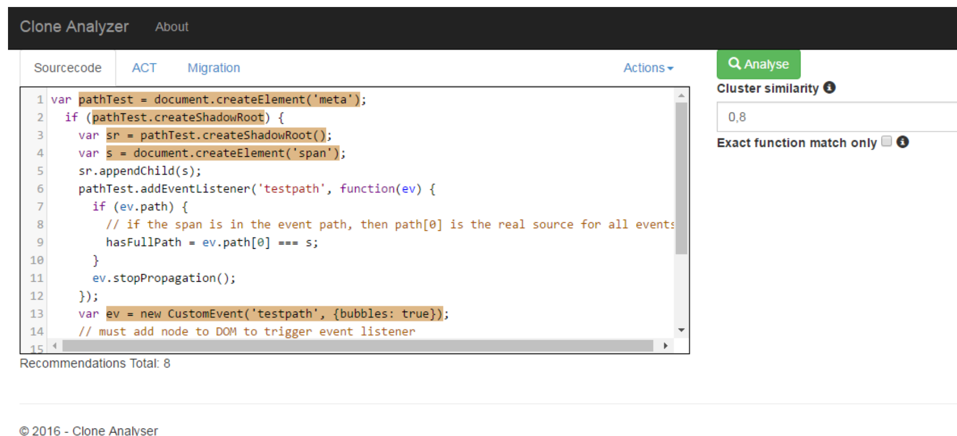


Joonis 24: Analüüsi töövoog

Sobivate koodiklastrite leidmine toimub sama moodi nagu mudeli koostamisel, ainult leitud klastrid saadetakse tagasi põhiprogrammile. Kui sobivaid klastreid leiti mitu, siis valitakse neist ainult kõige rohkem kloone sisaldav klaster ning saadetakse see. Selliselt jõuab põhiprogrammi mitte rohkem kui üks soovitusmuster ühe alampuu kohta. Need soovitusmustrid saadetakse koos lähtekoodiga kliendile. Kliendi poolel märgendatakse soovitustega koodilõigud värvidega 25. Kui klient vajutab soovitusele, avaneb modaalaken, mis kuvab kasutajale soovituse ning võimaldab tal see kas kinnitada või tagasi lükata 26.

7.1 Konteksti asendamine lähtekoodis

Kuna sarnasust kontrollitakse vaid programmi struktuuri põhjal, siis võib lähtekoodi üks-ühele soovitamine luua imelikke olukordi, kus programmi struktuur on õige, aga soovitatud muutujate, konstantide ja funktsioonide nimed on hoopis erinevad. Selle parandamiseks on katsetatud töös konteksti asendamise tehnikat, mis kirjutab lähtekoodis kõik muutujate nimed üle talle sarnase klooni abstraktse koodipuu kontekstis olevate nimedega. Näiteks kui kasutaja on oma koodis kasutanud kindla nimega muutujat, siis on sama nimi ka soovitatud koodis. Nimede asendamiseks lähekoodi tekstis ei saa kasutada standardteegi sõneasendusmeetodit, sest see võib asendada tähti ka sõna keskel. Selle asemel



Joonis 25: Märgendatud koodisegment peale analüüsi

on loodud sarnane meetod, mis lubab vahetada ainult tervet sõna korraga.

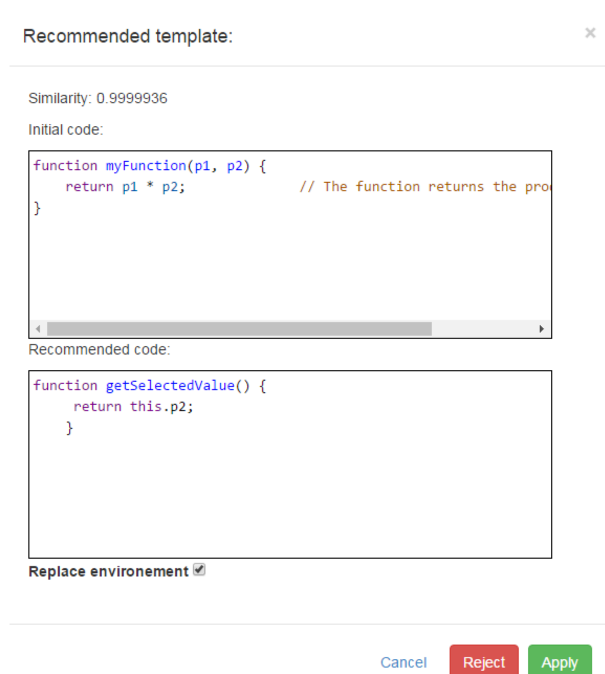
Töö raames loodud rakenduses tehakse konteksti vahetamist elementide järjekorra põhjal selles kontekstis. See töötab kenasti lihtsamate programmide puhul, kuid võib suuremate kloonide korral anda valepositiivseid tulemusi. Rakenduse edasiarendusena on mõistlik hoopis sarnasusmaatriksi põhjal leida kõige paremad muutujate vastavused ja teha vahetused selle järgi.

7.2 Rakenduse arhitektuur

Rakendus põhineb klient-server arhitektuuril, milles JavaScriptis kirjutatud veebipõhine klient suhtleb ASP.NET-is loodud kloonianalüsaatori teenusega. Lisaks on serveripoolne lähtekood jagatud kahe projekti vahel:

- CloneAnalyser - veebiteenus, mis suhtleb kliendi ja koodirepositooriumeid sisaldava andmebaasiga ning viib läbi analüüsi projekti.
- CloneParser - sisaldab funktsionaalsust lähtekoodi parsimiseks, transleerimiseks ja koodisegmentide võrdlemiseks. Projekt kompileerub DLL failiks, mida kasutab CloneAnalyser.

Veebiteenuse suhtlust kliendi ja koodirepositooriumite andmebaasiga on kujutatud joonisel 27 oleval jadaskeemil (*sequence diagram*). Jooniselt on näha, et kui analüüsi töövoog toimub kõik ühe päringuga, siis migreerimise puhul saadab server esmalt kliendile sobivate lähtekoodifailide loetelu ning seejärel käivitatakse iga faili parsimiseks eraldi päring. Selline lahendus võimaldab kliendil analüüsiprotsessi paremini juhtida, sest iga faili eduka migreerimise kohta saab klient tagasisidet. Näiteks kuvatakse kliendile progressi riba, millelt ta näeb, kuidas analüüs edeneb ja saab visuaalse hinnangu, kaua töö lõppemiseks veel aega võiks kuluda, tundide pikkuse migreerimisprotsessi puhul on see üsna oluline. Teiseks võimaldab see käivitada mitu päringut korraga ja kuna iga päring jookseb serveris eraldi lõimes, siis muudab see ka kogu protsessi kiiremaks.

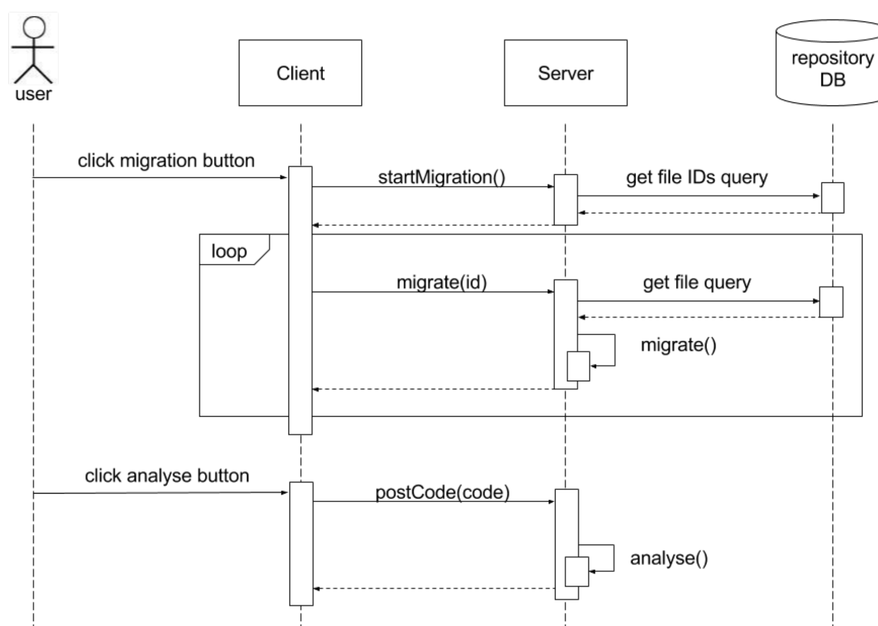


Joonis 26: Koodisegmendile vajutades avanev hüpikaken soovitusega

8 Tulemused

Töös kirjeldatud sarnasuse hindamise algoritm vastab küll sellele püstitatud nõuetele, kuid praktilises kasutamiseks on tal mõned puudused, mis avalduvad jõudluses ja täpsuses. Naabrite vastavuse põhine sarnasuse hindamise algoritm on ruutkeerukusega abstraktse koodipuu tippude arvu suhtes. Lisaks ei pruugi sarnasusmaatriks erandjuhtudel stabiliseeruda, seega tuleb algoritmis võimalike iteratsioonide arv piirata konstandiga [ZV08]. Üldjoontes ei ole sarnasuse hindamise algoritmi jõudlus antud rakenduse puhul suureks probleemiks, sest enamus võrreldavad alampuud on väikesed ja arvutus ise tehakse mälus.

Teiseks probleemiks on see, et algoritm töötab hästi isomorfsete graafide korral määrates nende sarnasuseks alati ühe, kuid leidub palju olukordi, kus see tagastab valepositiivseid tulemusi. Üheks selle probleemi põhjuseks on see, et kuigi algoritm kontrollib tippude vastavust, ei võrdle ta funktsioonide nimesid. Sisseehitatud funktsioonide korral lubab see soovitada veidraid asju, näiteks $a + b$ asemel $a.b$. Mõlemad on küll funktsiooni rakendamised, kuid täisarvude korral pole teine variant võimalik. Selle probleemi lahendamiseks on algoritmi lisatud võimalus ainult sama nimega sisseehitatud funktsioone vastandada. See aga vähendab võimalikke soovitusi, sest mõnda tehet saab mitut moodi kirja panna, näiteks kahe muutuja võrdlemiseks võib kasutada operaatorit $==$ või $===$. Lisaks selgus testimise käigus, et kasutatud sisendandmestiku põhjal koostatud soovitusmudel ei sisaldanud piisavalt palju erinevaid vasteid samale programmile, et neist oleks õnnestunud koguda iga operaatori kombinatsiooni jaoks vaste. Seetõttu on analüüsi käigus mõistlik sissehitatud funktsioonide täpne võrdlemine välja jätta, mis lubab analüsaatoril pakkuda huvitavamaid lahendusi.



Joonis 27: Veebiteenuse jadaskeem migreerimise ja analüüsi töövoogude jaoks

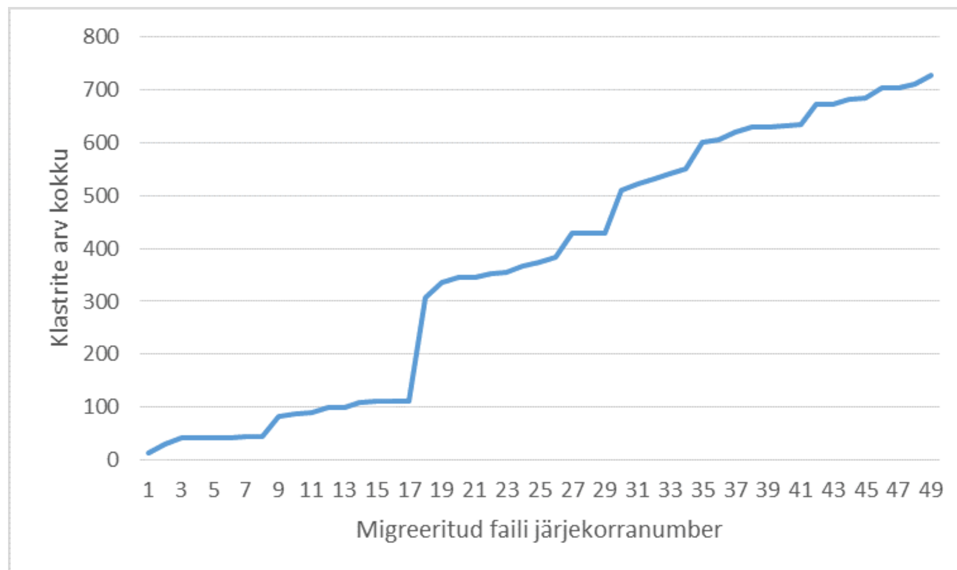
8.1 Algoritmi jõudluse hindamine

Töös kirjeldatud algoritmi jõudluse hindamisel tahame teada eelkõige seda, kuidas süsteem skaleerub, kui lasta sellel koguda soovitusmustreid pikema aja jooksul. Seetõttu on jõudluse hindamisel eelkõige uuritud migratsiooni töövoo jõudlust, kasutades reaalseid andmeid repositooriumite andmebaasist. Jõudlustestide tegemiseks kasutatud arvutis on nelja tuumaline i7-2600K protsessor ja 16 GB mälu, samuti asuvad nii klient kui ka server samas lokaalses võrgus. Kõigi katsete puhul on sarnasuse teguriks valitud 0,8.

Joonisel 28 on kujutatud klastrite arvu kasv migreerimisprotsessi jooksul, kus x-teljel on migreeritud failide arv ja y-teljel on andmebaasis olevate klastrite arv. Nagu jooniselt näha, siis kasvab see küllaltki lineaarselt, ainult 18. faili töötlemisel on tekkinud märgatav tõus. Lähemalt uurides selgub, et 18. fail on ka abstraktse koodipuu tippude poolest suurim.

Jõudluse hindamiseks on võrreldud keskmiselt ühe tipu migreerimiseks kuluvat aega, käivitades esmalt ainult üks paralleelne protsess korraga. Joonisel 29 on kajastatud töömahu kasv ajas, kus x-teljel on migreeritud failide arv ja y-teljel on faili migreerimiseks kulunud aeg jagatud sellele failile vastava abstraktse koodipuu tippude arvuga. See näitab iga faili kohta, kui kaua kulus keskmiselt ühe tipu migreerimiseks. Lisaks on tähistatud sinise joonega optimeerimata protsess, mis võrdleb alampuid kõikide klastritega ning punasega on tähistatud optimeeritud protsess, mis jätab võrdlemata väga erineva tippude arvuga klastrid, nagu kirjeldatud migreerimise töövoos 4.

Asjaolu, et paljude puude võrdlemata jätmine ei muuda analüüsi oluliselt kiiremaks illustreerib hästi seda, et kogu protsessi pudelikaelaks pole mitte võrdlemisalgoritmi kiirus, vaid suur soovitusmudelite hulk andmebaasis. Sest kuigi optimeeritud protsessis jääb suur hulk võrdluseid ära, tuleb nende kohta siiski pärida andmebaasist vastavas klastris olevate tippude arv. Samuti jäetakse välja just sellised võrdlused, kus ühes alampuus on märgatavalt rohkem tippe kui teises, need võrdlused on aga enamasti lihtsamad.



Joonis 28: Klastrite arvu kasvamine

Selleks, et hinnata kuidas muutub analüüsi kiirus paralleelsete migreerimisprotsesside arvust, on võrreldud aegu, mis kulub esimese 50 faili migreerimisele. Tabelis 5 on näidatud migreerimisele kuluva aja muutus sõltuvalt paralleelsete protsesside arvust. Nagu sealt näha, siis kahe paralleelse protsessi lisamisega väheneb migreerimisele kuluv aeg poole võrra. Täiendavate protsesside lisamine olulist ajavõitu ei andnud.

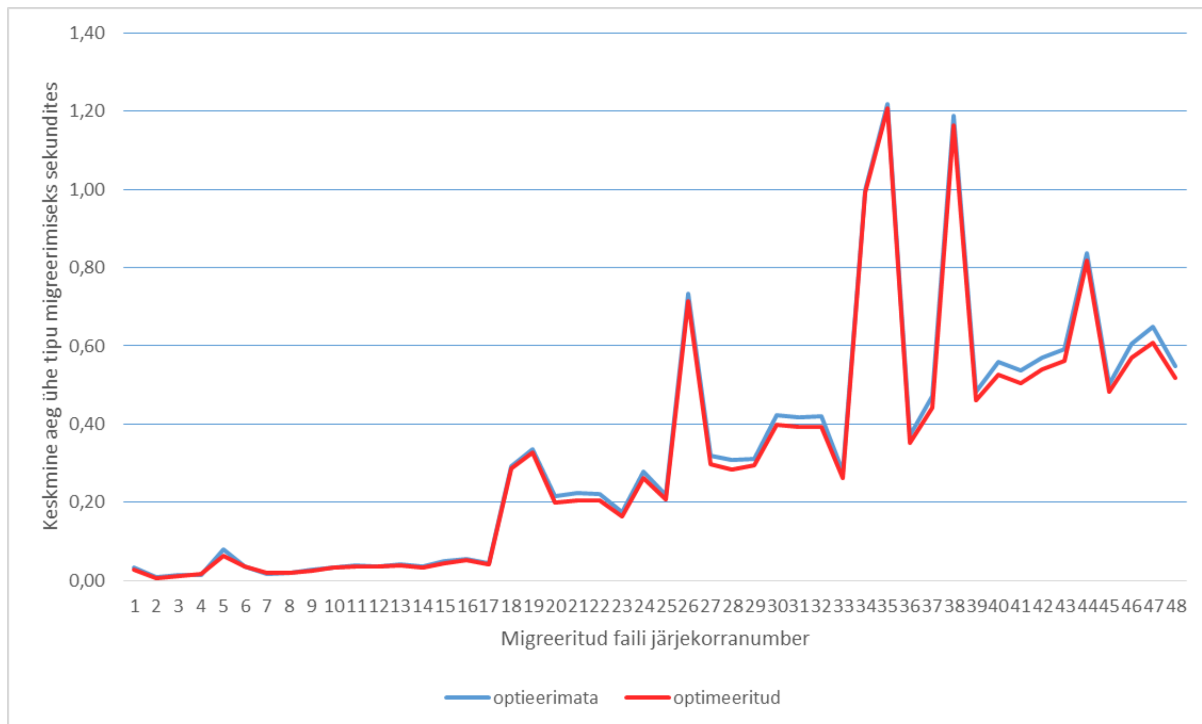
Tabel 5: Migratsiooni aja sõltuvus paralleelsete protsesside arvust

Protsesside arv	**Migratsiooni aeg**
1	60m
2	37m
3	29m
5	28m
10	27m
20	27m

Kogu koodirepositooriumite andmebaasi migreerimiseks, mis sisaldab 222 JavaScripti faili, kulus kokku 6 tundi ja 17 minutit. Võrreldes teiste abstraktsel süntaksipuul või sõltuvusgraafil põhinevate kloonituvastusmeetoditega jääb see hinnanguliselt samasse suurusjärku. Sellegi poolest on antud analüüsimeetodi puhul oht, et suurema andmestiku korral kasvab töö maht liiga palju.

8.2 Migreeritud koodisegmentide jaotus

Koodisegmentide jaotuse analüüsimiseks koostati esimese 50 faili põhjal soovitusmudel nii sisseehitatud funktsioonide nimesid arvestades kui ka arvestamata. Esimesel juhul saadi soovitusmudel 658 klastrist, mis sisaldasid kokku üle 4000 klooni ja need on jaotunud joonisel 30 näidatud jaotuse kohaselt. Nagu joonisel näha, siis enamus klastreid sisaldab ainult ühte koodisegmenti, ehk neile ei ole leitud ühtegi sarnast vastet. Rohkem kui kümmet klooni sisaldavaid klastreid leiti ainult 31. Kõige suuremas klastris on



Joonis 29: Töömahu kasvu võrdlus optimeerimata ja optimeeritud algoritmi korral

aga 908 klooni ning see vastab funktsiooni väljakutsele, mis ei ole kuigi huvitav. Suuruselt järgmises klastris on 316 klooni ning see vastab kahele järjestikusele funktsiooni väljakutsele.

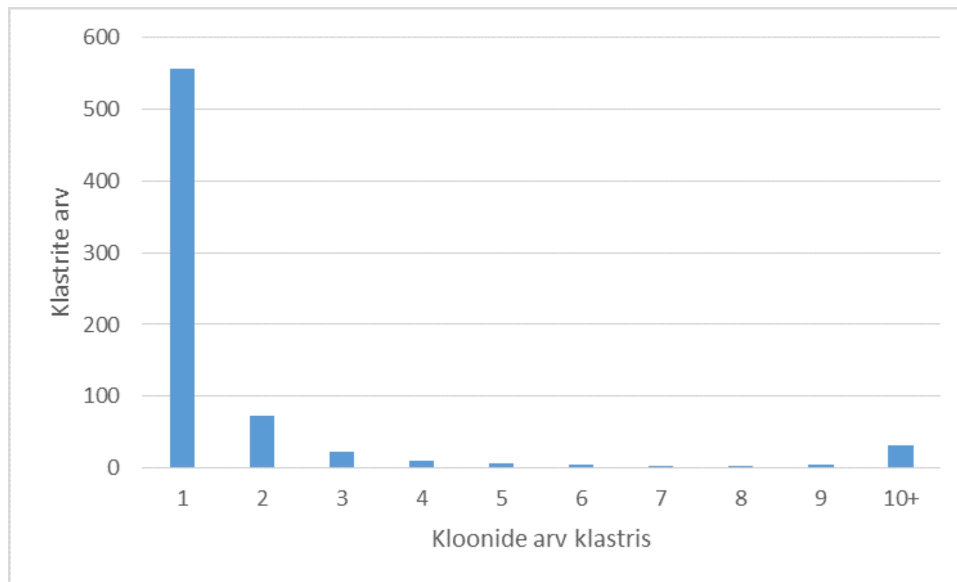
Järgmisena koostati soovitusmudel sisseehitatud funktsioonide nimesid arvesse võttes, mis tähendab, et näiteks liitmise ja korrutamise funktsioonide väljakutseid teineteisega ei sobitatud. Klastri suuruste võrdlust nende kahe meetodi korral on kujutatud joonisel 31, mille x-teljel on klastrid järjestatud suuremast väiksemani ning väärtus y-teljel näitab mitu klooni on vastas klastris. Joonisel ei ole näidatud klastreid, milles on vähem kui kolm klooni. Nagu jooniselt näha, siis funktsiooni nimedega arvestades on suurimad klastrid tunduvalt väiksemad ning keskmise suurusega klastreid on rohkem. Kõige suurem klaster vastab jätkuvalt funktsiooni väljakutsele, kuid funktsiooni nime arvestades on seal ainult parameetri võtmised objektist. Ülejäänud operatsioonid jagunevad teiste klastrite vahel.

Kokkuvõtteks saab öelda, et kui funktsioonide nimesid mitte võrrelda, tekib ainult paar väga suurt klastrit ning nende põhjal soovitab analüsaator rohkem valepositiivseid tulemusi.

8.3 Tulemuste kvaliteedi hindamine

Käesoleva töö puhul ei leidu ühte õiget soovitusmustrit, vaid pakutud mustri sobivus ja kasulikkus sõltuvad suuresti kasutaja eesmärgist ja eelistustest. Seetõttu on analüsaatori kvaliteedi hindamiseks läbiviidud ekspertuuring, milles osales kuus Tartu tarkvaraarendajat, kes puutuvad oma töös igapäevaselt kokku JavaScripti koodiga. Hindamismeetodi loomisel on tuginetud Melniku tööle [MGR02], milles on keskendutud järgmistele hinnangutele:

1. Tulemuse täpsuse hindamine, mille eesmärgiks on välja selgitada, kui palju tööd



Joonis 30: Kloonide jaotus klastrites

peab kasutaja tegema, et pakutud lahendus muuta soovitud tulemuseks.

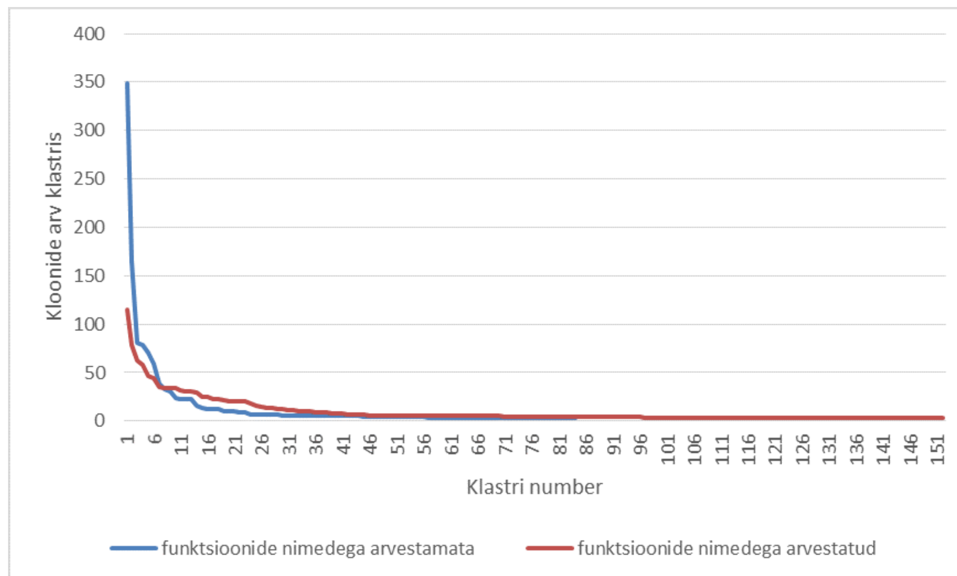
2. Oodatud tulemuste hindamine, mille eesmärgiks on uurida, kas saadud tulemused sisaldasid kasutaja jaoks kõige oodatumat lahendust.

Need hinnangud on kasulikud seetõttu, et nad iseloomustavad subjektiivselt süsteemi täpsust ja tundlikkust. Lisaks tulemuste hindamisele on töös hinnatud ka analüüsiprotsessi mugavust ja rakenduse kasutajasõbralikkust ning arvestatud ekspertide tagasisidega rakenduse paremaks muutmisel. Testimise ettevalmistusena pidid osalejad välja valima ühe JavaScripti koodifaili oma projektist, sellega tutvuma ning otsustama, mida nad ise seal muudatksid. Testsessioon toimus iga eksperdiga individuaalselt ning kestis orienteeruvalt 15 kuni 30 minutit. Testimise käigus täitsid osalejad küsitlusvormi, mille vastused on toodud töö lisas. Küsimustik sisaldas järgmiseid vabatekstilisi küsimusi:

1. Mitu koodirida on testimiseks kasutatud koodifailis?
2. Mitu soovitust analüsaator välja pakkus?
3. Mitu soovitust olid kasulikud?
4. Mitu soovitust olid mõistlikud, aga neist polnud antud kontekstis kasu?
5. Mitu soovitust olid täiesti ebavajalikud?
6. Millistest soovitustest oleks rohkem kasu olnud?

Seejärel küsiti ekspertide hinnangut analüsaatori kasutajakogemuse kohta, mille igale küsimusele sai valida hinnagu viie palli skaalal, kus üks tähistas kõige halvemat ning viis kõige paremat varianti. Samuti oli vastajal iga hinnangu juures võimalik põhjendada oma vastust. Küsimused olid järgmised:

1. Kuidas hindad rakenduse kasutajaliidest?



Joonis 31: Klastrite suurused funktsioonide nimesid arvestades ja arvestamata

2. Kuidas hindad analüüsiprotsessi kiirust igapäevatöös kasutamiseks?
3. Kui mugavaks hindad analüüsitulemuste läbivaatamist ja soovituste rakendamist?
4. Kuidas hindad sellise tööriista konseptsiooni üldiselt?

Viimaseks küsiti ekspertidelt nende soovitusi analüsaatori paremaks muutmiseks, millele nad said vabas vormis vastata.

Testimise tulemusena selgus, et reaaleluliste rakenduste analüüsimisel ei ole loodud staatilisest analüsaatorist palju kasu. Kokku vaatasid osalejad läbi 90 soovitust, millest ainult ühte oleks saanud modifitseeritud kujul rakendada. Seevastu 91,2% kõikidest soovitustest hinnati arendajate poolt ebavajalikeks, ehk valepositiivseteks tulemusteks. Seitsmel juhul tundus soovitus mõislik, kuid antud kontekstis polnud sellest kasu.

Kuna kasulikke soovitusi, mida saaks kohe programmis rakendada, leiti liiga vähe, siis ei õnnestunud soovituste kohandamist testida. Soovitused, mida eksperdid ise ootasid, olid enamasti üsna spetsiifilised, näiteks sooviti, et analüsaator pakuks välja parema algoritmi või alternatiivse teegi, mida kasutada. Samuti oodati, et rakendus tunneks paremini Angulari [Ang16] süntaksit või soovitaks stiilimuudatusi, näiteks kontrolliks nimekonventsioonidele vastavust ja eemaldaks lähtekoodi sisaldavaid kommentaare.

Rakenduse kasutajaliidest hindasid eksperdid keskmise hindega neli. Ühe suure puudusena toodi välja, et rakenduses analüsaator kaotab peale ühe soovituse aksepteerimist kõik ülejäänud ära. See käitumine on aga hetkel taotluslik, sest koodilõigu asendamine teistsuguse pikkusega koodilõigu vastu võib teiste soovituste asukohad sassi ajada. Samuti toodi välja, et soovituste teksti kastid võiksid olla suuremad ning koodi värvimisel võiks kasutada rohkem värve, et soovitusi üksteisest paremini eristada.

Analüüsi kiirusega olid eksperdid enamasti rahul, peamiselt arvati, et koodi järeltöötluseks on see piisavalt kiire, vaid reaalajas kontrollimiseks ei sobi.

Analüüsitulemuste läbi vaatamise ja rakendamise protsess oli ekspertide arvates küllaltki mugav, seega on sarnast protsessi mõistlik ka tulevikus soovituste hindamiseks kasutada.

Enamikule ekspertidest rakenduse üldine kontseptsioon meeldis. Neist mitu leidsid, et kui seda täiendada ja analüüsi tulemuste kvaliteeti parandada, saab igati kasuliku tööriista. Kõige levinum soovitus analüsaatori täiendamiseks oli erinevate teekide süntaksi toe lisamine, millest täpsemalt toodi välja standardteek, Angular ja Three.js [Thr16].

8.4 Autori hinnang töö tulemustele

Süsteem õppis hästi peamiselt lihtsaid programmeerimise mustreid, nagu aritmeetiliste avaldiste rakendamine ja pisemad funktsioonid, mida ta oskab ka välja pakkuda. Nendest mustritest pole arendajale aga kuigi palju kasu, sest need segmendid on kirjutatud täpselt nii nagu arendaja plaanis ja mõne teise funktsiooni kasutamine oleks seal vale. Samuti soovitab analüsaator mitmel juhul täpselt sama koodi, mida arendaja on juba kirjutanud.

Sarnasuse hindamiseks sobib töös kirjeldatud lahendus hästi, kuid analüsaator jääb hätta keerukamate programmide transleerimisega. See tekitab olukordi, kus abstraktses koodipuus on suure koodilõigu kohta vähe tippe, sest need on transleerimise käigus puudu jäänud. Selle tulemusena soovitatakse vahel kasutajale suuri koodilõike, mis ei seostu kuidagi analüüsitud programmiga.

Koodirepositooriumite andmebaas, mille põhjal käesolev soovitusmudel loodi, oli küllaltki mahukas, kuid siiski ei õnnestundu sealt piisavalt palju suuremaid koodisegmente koguda. Seetõttu ei leidunud analüsaator ka piisavalt täpseid vasteid kasutaja sisestatud koodile. Kuna nii andmete migreerimine kui ka analüüsi töövoog põhinevad samal algoritmil, siis võib tulevikus täiendada süsteemi selliselt, et kasutaja poolt analüüsitud failid migreeritakse automaatselt soovitusmudelile juurde. Sellisel viisil saab luua tööriista, mis muutub üha targemaks, mida rohkem seda kasutatakse. Lisaks saab sel viisil hinnata juba andmebaasis olemasolevate koodisegmentide sobivust ja sellest lähtuvalt muuta soovitusmustrite kaalusid.

Tulevikuarendusena on mõistlik koguda ka infot programmis kasutatavate funktsioonide kohta ning luua tabel enamlevinud teekide funktsioonidest. Selline meetod võimaldaks õppida ka muid teeke peale standardteegi ning oleks oluline täiendus analüsaatori kasulikuks muutmiseks. Lisaks tuleks rohkem tähelepanu pöörata sellele, kuidas eraldada stiililisi mustreid lihtsalt sarnastest koodisegmentidest, sest praeguse programmi struktuuril põhineva sarnasuse hindamine ei arvesta semantikast tulenevate stiilierinevustega.

Kuigi loodud prototüüplahendus pole veel tarkvaratööstuses kasutatav, leidub mitmeid võimalusi selle lahenduse edasi arendamiseks. Kuna üheks peamiseks probleemiks on valepositiivsete soovituste hulk, siis tuleks kaaluda ka teisi soovitusmudeli koostamise lahendusi, mis võimaldavad teha targemaid otsuseid. Näiteks kasutada koodirepositooriumites olevat muudatuste ajalugu ja lisatud kommentaare, mille põhjal koodisegmentide kasulikkust hinnata.

9 Kokkuvõte

Käesolevas töös anti ülevaade mustritel põhineva soovitusmudeli kasulikkusest staatilises analüüsis ning kirjeldati töövoogu ja algoritme koodikloonidel põhineva soovitusmudeli loomiseks. Samuti sai koostatud tööriist, mis neid tehnikaid rakendab ning aitab selle meetodi sobivust hinnata.

Loodud süsteem klasterdab treeningandmed sarnasuse põhjal ning valib igale klastrile ühe isendi soovitusmalliks. Kuna sarnasust on vaja hinnata erinevate arendajate kirjutatud koodilõikude vahel, tuleb seda teha kolmandat tüüpi kloonide põhjal, mis lubavad koodisegmentide vahetamist ja koodilõikude lisamist ning eemaldamist.

Töös anti ülevaade erinevate koodikloonide tuvastamise tehnikatest ning tutvustati nende ülesehitust ja tööprintsiipe. Lisaks võrreldi neid tehnikaid omavahel, mille tulemusena loodi vastavate kloonituvastustööriistade võrdlustabel.

Töö raames koostatud sarnasuse hindamise mudel teisendab sisendiks oleva programmi abstraktse koodipuu kujule ning leiab selle igale tipule vastavad muutujate kontekstid. Muutujate järgnevuse põhjal selles struktuuris konstrueeritakse osalise järjestuse relatsioon, mis seob nende tippude järgnevused, mille muutmisel võib programmi semantika muutuda. Vastava relatsiooni ja abstraktse koodipuu struktuuri põhjal koostatakse suunatud graaf, mille servad näitavad koodisegmentide omavahelisi järgnevusi. Sellest tulenevalt taandub programmide sarnasuse hindamine graafi sarnasuse hindamiseks ning selleks kasutatakse töös sarnasuse maatriksil põhinevat algoritmi. See lähenemine lubab küll leida kolmandat tüüpi kloone, kuid võib anda valepositiivseid tulemusi.

Lisaks soovitusmudeli koostamisele on töös loodud staatilise analüsaatori prototüüp, mis oskab kasutaja sisestatud koodile soovitusi pakkuda. Analüsaator leiab soovitusmudelist sarnased programmeerimismustrid ning soovitab neist kõige enam kasutatud mustreid täiendusettepanekutena. Nende ettepanekute sobivust saab kasutaja ise hinnata ning sobivad muudatused oma koodi sisse viia.

Kuigi valminud prototüüp ei ole veel tarkvaratööstuse jaoks tootekõlblik lahendus, võimaldas selle peal läbiviidud testimine avastada mitmeid probleeme, mis sellisel töövool põhinevat analüüsi raskendavad. Kasutajatele paremate soovituste tegemiseks on vaja hankida koodisegmentide kohta lisainfot, näiteks milliseid standardteeke ta kasutab, või mida on arendaja selle kohta kommenteerinud. Ainuüksi koodi kokkulangevuse põhjal soovituse tegemine on liiga ebatäpne, et olla mõistlik. Tulevikuarendusena saab soovitusmudeli koostamisel lähtuda kasutajate kommentaaridest ja ajaloost koodirepositooriumis ning eelistada soovitusmudelisse mitte enim kasutatud, vaid parimaks hinnatud soovitusi. Samuti saab koguda andmeid rakenduse kasutamise käigus ning selle põhjal soovitusmudelit jooksvalt täiendada.

Viited

[Ang16] Angular. `https://angularjs.org/`, (vaadatud 18.05.2016).

[ANT16a] Antlr *ANother Tool for Language Recognition.* `http://www.antlr.org/`, (vaadatud 18.05.2016).

[ANT16b] Antlr grammar repository. `https://github.com/antlr/grammars-v4`, (vaadatud 18.05.2016).

[Bel02] Stefan Bellon. Vergleich von techniken zur erkennung duplizierten quellcodes. Master's thesis, University of Stuttgart, Germany, 2002.

[BKA+07] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Software Eng.*, 33(9):577–591, 2007.

[BS03] Stefan Bellon and Daniel Simon. Vergleich von klonerkennungstechniken (comparison of clone detection techniques). *Softwaretechnik-Trends*, 23(2):10–12, 2003.

[BYdM+98] Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM '98 Proceedings of the International Conference on Software Maintenance*, pages 368–377, 1998.

[Cod16] Codemirror. `https://codemirror.net/`, (vaadatud 18.05.2016).

[CR11] James R. Cordy and Chanchal K. Roy. The NiCad clone detector. pages 219–220, 2011.

[ecm16a] Ecma-262 standard. `http://www.ecma-international.org/publications/standards/Ecma-262.htm`, (vaadatud 18.05.2016).

[Ecm16b] Antlr4 grammar for ecmascript. `https://github.com/bkiers/ecmascript-parser`, (vaadatud 18.05.2016).

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees and Sequences.* Cambridge University Press, 1997.

[HRB90] Susan Horwitz, Thomas W. Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[JSL16] Jslint. `http://www.jslint.com/`, (vaadatud 18.05.2016).

[KFF06] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *2006 13th Working Conference on Reverse Engineering*, pages 253–262, 2006.

[Kil14] Karl Kilgi. Code clone detection using wavelets. Master's thesis, University of Tartu, Estonia, 2014.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Eng.*, 28(7):654–670, 2002.

[Kri01] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01*, pages 301–309, 2001.

[LPX+15] Yun Lin, Xin Peng, Zhenchang Xing, Diwen Zheng, and Wenyun Zhao. Clone-based and interactive recommendation for modifying pasted code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 520–531, 2015.

[MGR02] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Proceedings of the 18th International Conference on Data Engineering*, pages 117–128, 2002.

[Mic09] Christine Michel. Poset representation and similarity comparisons os systems in IR. *CoRR*, abs/0906.3085, 2009.

[MN08] Jiri Matousek and Jaroslav Nesetril. *An Invitation to Discrete Mathematics.* Oxford University Press, 2008.

[Nik12] Mladen Nikolic. Graph similarity scoring and matching. *Intelligent Data Analysis*, 16(6):865–878, 2012.

[NN07] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer.* Springer, 2007.

[RCK09] Chanchal Kumar Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, 2009.

[Sas16] Dulanga Sashika. Measuring graph similarity using neighbor matching. `https://wadsashika.wordpress.com/2014/09/19/measuring-graph-similarity-using-neighbor-matching/`, (vaadatud 18.05.2016).

[Sea16] Searchcode. `https://searchcode.com/`, (vaadatud 18.05.2016).

[SF03] Radomir S. Stankovic and Bogdan J. Falkowski. The haar wavelet transform: its status and achievements. *Computers & Electrical Engineering*, 29(1):25–44, 2003.

[Sim16] Simian. `http://www.harukizaemon.com/simian/`, (vaadatud 18.05.2016).

[Thr16] Three.js. `http://threejs.org/`, (vaadatud 18.05.2016).

[Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[ZV08] Laura A. Zager and George C. Verghese. Graph similarity scoring and matching. *Appl. Math. Lett.*, 21(1):86–94, 2008.

Lisad

Lisa 1 - rakenduse paigaldamine ja käivitamine

Töö raames loodud rakenduse testversioon on üles laetud Tartu Ülikooli Betamax'i serverisse, millele saab ligi Tartu Ülikooli sisevõrgu kaudu, aadressilt http://betamax/Clone-analyser/Home/Index.

Rakenduse saab paigaldada oma arvutisse, laadides lähtekoodi GitHub'i aadressilt https://github.com/jjaggo/CloneAnalyser.git. Selle käivitamiseks peab olema arvutisse paigaldatud eelnevalt SQL Server 2008 ja Visual Studio 2013. Web.config failis tuleb üle vaadata *connectionString*, sest see peab sobima lokaalsesse andmebaasi ühendamiseks. Samuti tuleb tagada rakendusele ligipääs Betamax'i serverisse, kus asub repositooriumite andmebaas. Selleks peab kasutaja olema Tartu Ülikooli internetivõrgus, näiteks VPN'i abil. Kui see on tehtud, saab käivitada-CloneAnalyser projekti, mis avab rakenduse veebilehe.

Lisa 2 - ekspertuuringu tagasiside

Järgnevalt on toodud ekspertide vastused rakenduse kvaliteedi hindamise küsimustikule. Tabelis 6 on kujutatud testimiseks kasutatud lähtekoodis olevate koodiridade arvu ning sellele välja pakutud soovituste arvu. Tabelis 7 on läbi vaadatud soovitused jagatud kolme rühma: kasulikud, mõistlikud, kuid mitte antud kontekstis kasulikud ja ebavajalikud.

Tabel 6: Analüsaatori pakutud soovituste võrdlus faili suurusega

Koodiridade arv testitud failis	**Väljapakutud soovituste arv**
394	133
47	10
196	29
45	16
94	13
230	4

Tabel 7: Analüsaatori soovituste kasulikkuse võrdlus

Läbivaadatud soovitusi kokku	**Kasulikke**	**Mõistlikke**	**Ebavajalikke**
18	1	5	12
10	0	1	9
29	0	0	29
16	0	1	15
13	0	0	13
4	0	0	4

Küsimusele, millistest soovitustest oleks rohkem kasu olnud, vastati järgnevalt:

1. Ei oska midagi otseselt soovitada, kui oleks lihtsam süntaks olnud, siis oleks ilmselt ka rohkem tulemusi andnud.
2. Rakendus võiks hoopis algoritme koodist leida ja pakkuda paremat algoritmi.
3. Oleks võinud $(newDate()).getTime()$ asemel soovitada Three.js *Clock* kasutamist.
4. Konventsioonisoovitused, näiteks: nimekonvensioonid, tühikute konvensioonid jms. JavaScript on raske keel, milles konvensioonidest kinni pidada, sest ühe asja saab palju rohkematel viisidel programmeerida võrreldes teiste keeltega.
5. Sellistest, mis oleks ka koodiga kokk läinud. Hetkel ei olnud ühestki pakutud soovitusest kasu ja iga järgmise funktsiooni juures pakkus samu variante. Arvan, et AngularJS ajas ta segadusse, sest analüüsitud kood oli puhas Angular.
6. Kasutute kommentaaride, mis sisaldavad lähtekoodi, eemaldamist.

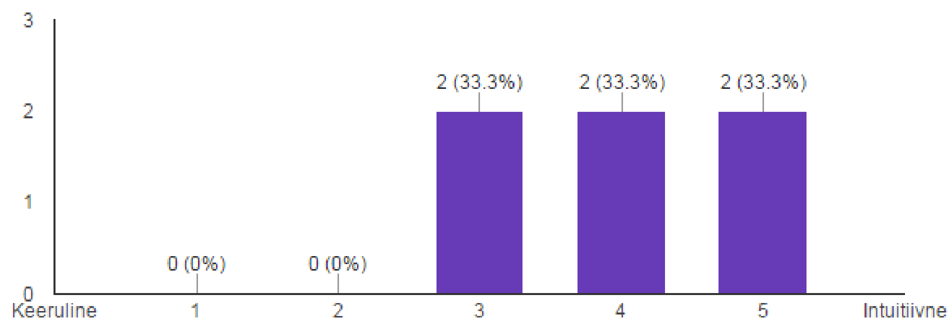
Kasutajaliidese intuitiivsust hindasid eksperdid joonisel 32 kujutatud hinnetega. Oma hinnangut põhjendati järgnevalt:

1. Tundus igati loogiline lahendus.

2. Analüüsi nupp ei olnud progressiribaga ning seetõttu ei teadnud kaua pean ootama. Soovituse rakendamise nupp kaotas ära kõik soovitused. Konteksti vahetamise nupu kohta pidin küsima, mida see teeb. Samas soovituste värvimine ja hiire peale viimisel värvi värvi vahetamine olid head. Üldiselt olid nupud mõistlike kohtade peal.

3. Kui ignoreerida mõningaid pisikesi veebidisaini põhitõdede rikkumisi, on kasutajaliides intuitiivne.

4. Polnud esimesel kasutamisel intuitiivne, analüüs märkis peaaegu kogu koodi ning raske oli eristada, millised koodi alad erinevate soovituste alla kuuluvad. Muudatuste vaatamise aknas võiksid tekstialad suuremad olla.



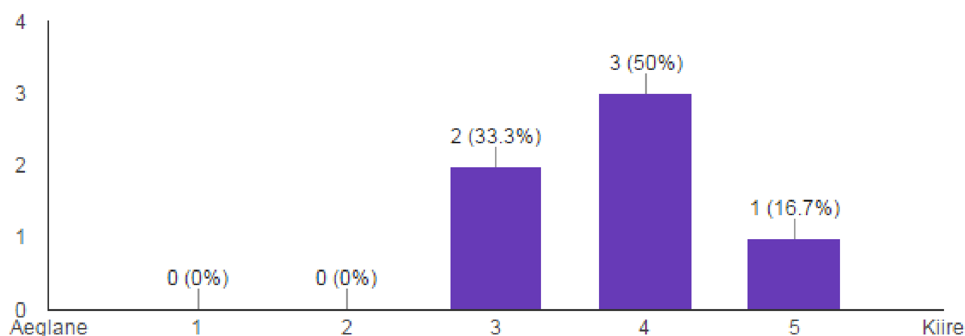
Joonis 32: Hinnangud rakenduse kasutajaliidesele

Kasutajaliidese kiirust igapäevatöös kasutamiseks hindasid eksperdid joonisel 33 kujutatud hinnetega. Oma hinnangut põhjendati järgnevalt:

1. Koodi järjeltöötluse jaoks on kiirus hea, aga reaalajas kasutada ei saa.

2. Arvestades, et sain rakendust kasutada demoplatvormil, kujutan ma ette, et seda saaks kiirendada võimsama riistvaralise lahendusega.

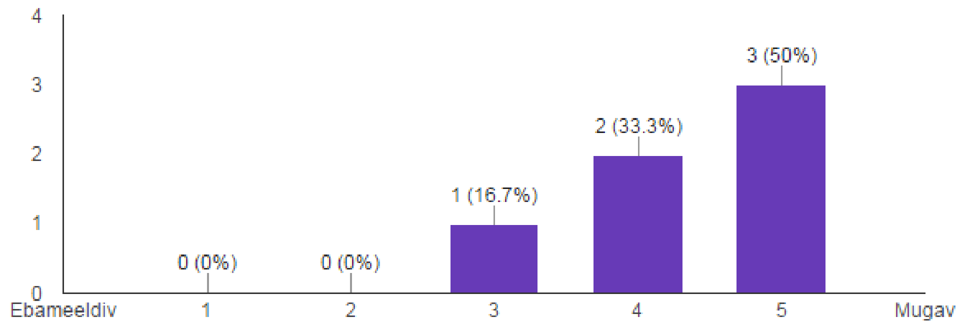
3. Väikese koodi kontrolli tegi üsna kiirelt, umbes 20 sekundiga.



Joonis 33: Hinnangud analüüsiprotsessi kiirusele igapäevatöös kasutamiseks

Analüüsi tulemuste läbivaatamise ja soovituste rakendamise protsessi hinnati joonisel 34 kujutatud hinnetega. Oma valikut põhjendati järgnevalt:

1. Ma ise ei oskaks seda paremini lahendada.
2. Mulle väga meeldis, et ta värvis teksti ning avanes eraldi aken, kus on kõik ühes kohas ja lihtne.
3. Soovitused ja tegelik kood võiks üksteise kõrval olla, siis on pikemate soovituste korral parem lugeda.
4. Mind häiris, et ühe muudatuse rakendamisel teised soovitused kaovad.



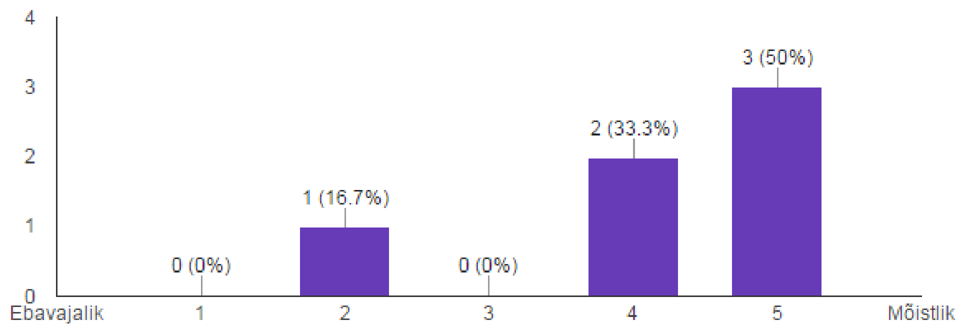
Joonis 34: Hinnangud analüüsitulemuste läbivaatamisele ja soovituste rakendamise mugavusele

Rakenduse üldist kontseptsiooni hinnati joonisel 35 kujutatud hinnetega. Oma valikut põhjendati järgnevalt;

1. Huvitav tööriist, päris lahe idee.
2. Kui see töötaks perfektselt, siis oleks väga kasulik. Eriti hea oleks, kui saaks kogu projekti koodi ühe nupuvajutusega läbi analüüsida.
3. Praegu jäi mulje, et süsteem otsib süntaktilist sarnasust, mitte semantilist. Näiteks nimeruumiga konstruktori asemel soovitas parameetri võtmist objektist. Ma arvan, et parem lähenemine oleks otsida just süntaktiliselt erinevast koodist asju, mida võiks semantiliselt identsete, kuid süntaktiliselt erinevate programmeerimismustritega lahendada. Võib-olla on praegustest süntaktilistest sarnasustest kasu olukorras, kus koodis on näiteks topeltsulud ja soovitus oleks need ära võtta.
4. Tundub natuke ambitsioonikas, aga huvitav oleks seda temaatikat edasi arendada, et välja töötada tootekõlblik produkt.
5. Kui rakendus saaks Angularist aru, siis oleks see väga kasulik.
6. Tööriistana on see igati kasulik.

Analüsaatori paremaks muutmiseks pakuti välja järgmised soovitused:

1. Võiks arvestada standardteekidega.



Joonis 35: Hinnangud rakenduse kontseptsioonile

2. Funktsioonide ja klasside nimed võiksid soovituses jääda samaks mis esialgses koodis, hetkel jäid samaks ainult muutujate nimed. Olukorras, kus on kasutatud kolme muutujat, aga sama asja saab teha ka kahega, võiks analüsaator selle ära parandada, näiteks luues kolmest ühe funktsiooni.

3. Analüsaator võiks aru saada, mida ma koodiga teha üritan. Näiteks kui ma kasutan Three.js teeki, siis soovitada mulle *Clock* klassi. Samuti võiks soovitada erinevate objektide poole pöördumise asemel võtta nende väärtus muutujasse, kui see muudab koodi kiiremaks.

4. Esiteks soovitan viia arendus paremale platvormile. Teiseks soovitan leida rohkem ressurssi prototüübi arendamiseks, potentsiaali on, hetkesisuga natuke toores.

5. Rakendus võiks pakkuda Angulari süntaksi tuge.

6. Muudatuste rakendamisel võiksid koodis ülejäänud soovitused säilida, et saaksin ka need ühekaupa läbi vaadata ja soovi korral rakendada.

Lisa 3 - Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, **Jaanus Jaggo**,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose
   Sarnaste koodisegmentide põhjal soovitusmudeli loomine,
   mille juhendaja on Siim Karus,

1.1 reprodutseerimiseks säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

1.2 üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace'i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.

3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, **19.05.2016**