Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

FIIT-5220-56303

Bc. Ján Súkeník

ANALÝZA ZDROJOVÝCH KÓDOV S VYUŽITÍM ABSTRAKTNÝCH SYNTAKTICKÝCH STROMOV

Diplomová práca

Študijný program: Softvérové inžinierstvo
Študijný odbor: 9.2.5 Softvérové inžinierstvo
Miesto vypracovania: Ústav informatiky a softvérového inžinierstva, FIIT STU, Bratislava
Vedúci práce: Ing. Peter Lacko, PhD.

máj 2013

S T U
F I I T
SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
Fakulta informatiky a informačných technológií

Zadanie diplomovej práce

Meno študenta: **Bc. Súkeník Ján**

Študijný program: Softvérové inžinierstvo

Študijný odbor: Softvérové inžinierstvo

Názov práce: **Analýza zdrojových kódov s využitím abstraktných syntaktických stromov**

Samostatnou výskumnou a vývojovou činnosťou v rámci predmetov Diplomový projekt I, II, III vypracujte diplomovú prácu na tému, vyjadrenú vyššie uvedeným názvom tak, aby ste dosiahli tieto ciele:

Všeobecný cieľ:

Vypracovaním diplomovej práce preukážte, ako ste si osvojili metódy a postupy riešenia relatívne rozsiahlych projektov, schopnosť samostatne a tvorivo riešiť zložité úlohy aj výskumného charakteru v súlade so súčasnými metódami a postupmi študovaného odboru využívanými v príslušnej oblasti a schopnosť samostatne, tvorivo a kriticky pristupovať k analýze možných riešení a k tvorbe modelov.

Špecifický cieľ:

Vytvorte riešenie, zodpovedúce návrhu textu zadania, ktorý je prílohou tohto zadania. Návrh bližšie opisuje tému vyjadrenú názvom. Tento opis je záväzný, má však rámcový charakter, aby vznikol dostatočný priestor pre Vašu tvorivosť.

Riaďte sa pokynmi Vášho vedúceho.

Pokiaľ v priebehu riešenia, opierajúc sa o hlbšie poznanie súčasného stavu v príslušnej oblasti alebo o priebežné výsledky Vášho riešenia alebo o iné závažné skutočnosti, dospejete spoločne s Vaším vedúcim k presvedčeniu, že niečo v texte zadania a/alebo v názve by sa malo zmeniť, navrhnite zmenu. Zmena je spravidla možná len pri dosiahnutí kontrolného bodu.

Miesto vypracovania: Ústav informatiky a softvérového inžinierstva FIIT STU v Bratislave

Vedúci práce: **Ing. Peter Lacko, PhD.**

Termíny odovzdania:

podľa harmonogramu štúdia platného pre semester, v ktorom máte príslušný predmet (Diplomový projekt I, II, III) absolvovať podľa Vášho študijného plánu

Predmety odovzdania:

V každom predmete dokument podľa pokynov na www.fiit.stuba.sk v časti: home > Informácie o > štúdiu > organizácia štúdia > diplomový projekt

V Bratislave dňa 13. 2. 2012

prof. Ing. Pavol Návrat, PhD.
riaditeľ Ústavu informatiky a softvérového inžinierstva

S T U
F I I T
SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
Fakulta informatiky a informačných technológií

Návrh zadania diplomovej práce

Finálna verzia do diplomovej práce [1]

Študent:

Meno, priezvisko, tituly:	Ján Súkeník, Bc.
Študijný program:	Softvérové inžinierstvo
Kontakt:	xsukenik

Výskumník:

Meno, priezvisko, tituly:	Peter Lacko, Ing. PhD.

Projekt:

Názov:	Analýza zdrojových kódov s využitím abstraktných syntaktických stromov
Názov v angličtine:	Source Code Analysis Using Abstract Syntax Trees
Miesto vypracovania:	Ústav informatiky a softvérového inžinierstva, FIIT STU, Bratislava
Oblasť problematiky:	Metódy a nástroje pre podporu vývoja softvérových systémov

Text návrhu zadania[2]

V súčasnosti sa nemalé úsilie venuje snahe získavať rôzne informácie z repozitárov zdrojových kódov softvérových systémov. Jedným z možných využití týchto dát je identifikácia častí kódov, ktoré sú s najväčšou pravdepodobnosťou náchylné na chyby. Od konkrétneho zápisu programu sa dá abstrahovať pomocou syntaktických stromov, čo môže uľahčiť spomínané získavanie informácií.

Analyzuje možnosti použitia abstraktných syntaktických stromov pri porovnávaní zdrojových kódov. Zamerajte sa na vyhľadávanie zduplikovaných častí kódu a porovnávanie nasledujúcich verzií toho istého zdrojového kódu. Zistite, aké informácie je možné získať s využitím takejto analýzy a ako je možné získané informácie ďalej použiť. Navrhnite spôsob, akým sa dá porovnávanie zdrojových kódov pomocou abstraktných syntaktických stromov aplikovať pri získavaní informácií z repozitárov softvérových systémov. Príkladom takej informácie je práve určovanie miest zdrojových kódov, ktoré sú najviac náchylné na chyby, napr. na základe frekvencie a rozsahu zmien vykonávaných v danej časti kódu v minulosti.

Implementujte aplikáciu, ktorá bude slúžiť na analýzu repozitárov zdrojových kódov. Vytvorené riešenie overte na histórii zdrojového kódu dostatočne veľkého softvérového systému, napríklad niektorého z väčších open-source projektov.

---

[1] Vytlačiť obojstranne na jeden list papiera

[2] 150-200 slov (1200-1700 znakov), ktoré opisujú výskumný problém v kontexte súčasného stavu vrátane motivácie a smerov riešenia

Literatúra[3]

- Giger, E., Pinzger, M., Gall, H. C.: Comparing fine-grained source code changes and code churn for bug prediction. Waikiki, Honolulu: ACM, 2011. Proceeding of the 8th working conference on Mining software repositories. s. 83-92. 978-1-4503-0574-7.
- Kagdi, H., Collard, M. L., Maletic, J. I.: Towards a taxonomy of approaches for mining of source code repositories. St. Louis, Missouri: ACM, 2005. Proceedings of the 2005 international workshop on Mining software repositories. s. 1-5. 1-59593-123-6.
- Neamtiu, I., Foster, J. S., Hicks, M.: Understanding source code evolution using abstract syntax tree matching. St. Louis, Missouri: ACM, 2005. Proceedings of the 2005 international workshop on Mining software repositories. s. 1-5. 1-59593-123-6.

Vyššie je uvedený návrh diplomového projektu, ktorý vypracoval(a) Bc. Ján Súkeník, konzultoval(a) a osvojil(a) si ho Ing. Peter Lacko, PhD. a súhlasí, že bude takýto projekt viesť v prípade, že bude pridelený tomuto študentovi.

V Bratislave dňa 9.1.2012

Podpis študenta

Podpis výskumníka

Vyjadrenie garanta predmetov Diplomový projekt I, II, III

Návrh zadania schválený: áno / ~~nie~~[4]

Dňa: 7.2.2012

Podpis garanta predmetov

[3] 2-3 vedecké zdroje, každý na samostatnom riadku a s údajmi zodpovedajúcimi bibliografickým odkazom podľa normy STN ISO 690, ktoré sa viažu k téme zadania a preukazujú výskumnú povahu problému a jeho aktuálnosť (uveďte všetky potrebné údaje na identifikáciu zdroja, pričom uprednostnite vedecké príspevky v časopisoch a medzinárodných konferenciách)

[4] Nehodiace sa prečiarknite

Anotácia

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
Študijný odbor: Softvérové inžinierstvo

Autor: Bc. Ján Súkeník
Diplomová práca: Analýza zdrojových kódov s využitím abstraktných syntaktických stromov
Vedúci diplomovej práce: Ing. Peter Lacko, PhD.

máj 2013

V súčasnosti sa pri vývoji softvéru bežne využívajú informácie z rôznych podporných nástrojov, napríklad verziovacích systémov, systémov na monitorovanie chýb a pod. Najpresnejším zdrojom informácií o samotnom softvéri sú však stále jeho zdrojové kódy. Tie sa dajú analyzovať, napríklad s cieľom nájsť také časti, ktoré s vysokou pravdepodobnosťou obsahujú chyby. Od zápisu kódov sa dá abstrahovať pomocou abstraktných syntaktických stromov.

Častým zdrojom chýb, resp. zvýšených nákladov počas vývoja alebo údržby softvéru, je zduplikovaný kód. Jeho agresívne refaktorovanie môže byť podľa existujúceho výskumu nielen zbytočné, ale niekedy až nežiadúce. Preto v práci navrhujeme metódu, ktorá umožňuje aktívne monitorovať duplikáty v softvérových projektoch priebežne počas ich vývoja. Je špeciálne navrhnutá tak, aby umožnila výsledky rýchlo aktualizovať po každej (malej aj veľkej) zmene zdrojových kódov. Metódu v práci overujeme na reálnych softvérových projektoch. Na základe výsledkov si dovolíme tvrdiť, že pri nasadení v praxi má potenciál zvýšiť kvalitu softvéru a znížiť náklady na jeho vývoj.

Annotation

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES
Degree Course: Software Engineering

Author: Bc. Ján Súkeník
Master's Thesis: Source Code Analysis Using Abstract Syntax Trees
Supervisor: Ing. Peter Lacko, PhD.

2013, May

Nowadays we use information from many different tools to support the process of software development, for example source code repositories, bug tracking systems, etc. The most accurate source of such information is the code of the software itself. We can analyse the source code, for example in order to find parts that may contain bugs. During such analysis, abstract syntax trees can enable us to ignore all the details related to text representation of the code.

Duplicated code is very common source of bugs, or at least source of increased costs during software development and maintenance. According to the existing research, its aggressive refactoring can be redundant or sometimes even harmful. We therefore propose a method which enables developers to actively and continuously track duplicates during the development. It is designed so that the results can be immediately updated right after every change of the source code (small or large). We tested the method on real software projects. Based on the results, we are confident that our tool has the potential to increase quality of software projects and reduce development costs.

Obsah

1

Úvod

Pri vývoji softvéru bežne používame veľa rôznych podporných systémov. Sú nimi napríklad verziovacie systémy zdrojových kódov, systémy na správu chýb, atď. Tieto nástroje obsahujú množstvo cenných informácií, z ktorých sa však v praxi reálne využíva len zlomok.

Najpresnejšie a nespochybniteľné informácie o softvérových systémoch obsahujú samotné zdrojové kódy. Preto je vhodné začať práve pri ich analýze. Abstrakciu od konkrétneho zápisu kódu vieme vykonať pomocou abstraktných syntaktických stromov. Tie sa následne dajú jednoduchšie spracovať a využiť napríklad pri hľadaní takých častí kódu, ktoré s veľkou pravdepodobnosťou obsahujú chyby. Takéto informácie nám môžu pomôcť lepšie riadiť proces vývoja a takisto zvýšiť kvalitu vytvoreného softvéru.

Jedným z častých zdrojov chýb softvérových systémov je zduplikovaný kód. Fowler et al. [9] ho vo svojej knihe uvádzajú na prvom mieste spomedzi všetkých pachov zdrojových kódov. Pri vývoji a údržbe softvéru nepochybne spôsobuje zvýšené náklady. Preto sa jeho automatickému vyhľadávaniu venovalo už množstvo pozornosti a navrhlo celé spektrum rôznych metód, ktoré sa však v praxi používajú len zriedkavo. Až nedávno sa výskumníci pozreli na to, ako sa zduplikovaný kód v reálnych projektoch vyvíja a aké má reálne dopady na množstvo chýb [14]. Výsledky naznačujú, že agresívne vyhľadávanie a refaktorovanie duplikátov nemusí byť vždy tou správnou cestou. Ako vhodnejšiu alternatívu navrhujú zduplikovaný kód počas vývoja softvéru len aktívne sledovať [28].

Nástroje, ktoré by takéto sledovanie umožňovali, resp. zjednodušovali, však neexistujú. V práci preto jeden navrhujeme. Jedná sa o metódu automatického vyhľadávania zduplikovaného kódu špeciálne navrhnutú tak, aby umožnila odhalené duplikáty jednoducho a rýchlo aktualizovať pri každej malej zmene zdrojových kódov. Nástroj sa vďaka tomu dá použiť práve na aktívne sledovanie zduplikovaného kódu počas vývoja softvéru. Aktualizácie duplikátov sa vykonávajú takmer v reálnom čase a vývojár na ne môže promptne reagovať, ak to uzná za vhodné.

V prvej časti práce uvádzame prehľad výskumu, ktorý bol v oblasti automatickej detekcie zduplikovaného kódu a analýzy jeho evolúcie už vykonaný. Výsledkom je odhalenie už spomínaného otvoreného problému v oblasti údržby duplikátov. Následne navrhujeme metódu, ktorá adresuje tento problém tak, že veľmi efektívne indexuje zdrojové kódy a vďaka tomu umožňuje rýchle vyhľadávanie duplikátov. Následne opisujeme experimenty, vďaka ktorým sme potvrdili, že navrhnutá metóda funguje tak, ako sme očakávali. Myslíme si, že pri nasadení v praxi má potenciál zvýšiť kvalitu softvéru a zároveň, aspoň čiastočne, znížiť náklady na jeho vývoj.

2

Evolúcia softvéru a predpovedanie chýb

Odstraňovanie chýb je nepochybne jednou z najdôležitejších aktivít vykonávaných počas životného cyklu všetkých softvérových systémov. Začína počas vývoja, pokračuje pri testovaní a nasadzovaní a tvorí veľkú časť fázy údržby softvéru. Neskoro odhalené chyby a ich následné odstraňovanie spotrebujú veľa prostriedkov, najmä finančných. Preto sa hľadajú spôsoby, ktoré umožnia odhaliť čo najviac chýb čo najskôr, keď je ich odstránenie ešte lacné – ideálne priamo počas vývoja.

V praxi sa najčastejšie hľadajú chyby v softvérových systémoch pomocou rôznych druhov testovania. Okrem toho existujú metódy, ktoré sa snažia predpovedať kvalitu softvéru bez toho, aby museli vedieť čo a ako má konkrétny softvér robiť. Upozorňujú vývojárov na tie časti zdrojových kódov, ktoré by mohli obsahovať chyby. Príkladmi sú statická a dynamická analýza kódu, sledovanie rôznych metrík, atď.

2.1 Prehľad metód predpovedania kvality softvéru

V nasledujúcom texte uvádzame krátky prehľad niektorých metód, ktoré slúžia na predpovedanie chýb softvérových systémov.

Numerické a štatistické metódy. Prvú skupinu metód tvoria rôzne numerické a štatistické modely. Takéto predpovede chýb Fenton et al. [6] delia na: (1) predpovede na základe metrík veľkosti a zložitosti kódu, (2) predpovede na základe metrík testovania, (3) predpovede využívajúce informácie o procese vývoja a jeho kvalite, (4) kombinované prístupy.

Fenton et al. [6] opisujú aj problémy, ktoré s týmito metódami súvisia. Intuitívne je ich hlavnou nevýhodou to, že neposkytujú informácie o konkrétnych chybách. Sú to len akési odhady počtu možných chýb, odporúčania veľkostí modulov, ktoré mávajú najmenšie množstvo chýb. Ich používanie však môže mať zmysel minimálne z hľadiska celkového hodnotenia procesu vývoja.

Hľadanie návrhových vzorov a antivzorov. Hľadanie návrhových vzorov v softvérových systémoch je ďalšou skupinou metód, ktoré sa dajú využiť pri hľadaní a predpovedaní chýb. Myšlienka je jednoduchá. Ak vieme, ako by mal správne implementovaný vzor vyzerať a vieme, kde v kóde je použitý, je možné overiť, či implementácia obsahuje chyby. S tým súvisí aj vyhľadávanie antivzorov [23] – teda tých častí kódov, o ktorých dopredu vieme povedať, že sú nesprávne implementované a určite budú v budúcnosti zdrojom problémov. Známym zástupcom tejto skupiny je napríklad nástroj FindBugs[1].

Získavanie informácií zo systémov na podporu vývoja. V súčasnosti sú neodmysliteľnou súčasťou vývoja a údržby softvéru rôzne podporné systémy, napríklad repozitáre zdrojových kódov, systémy na monitorovanie chýb, atď. Všetky sú možným zdrojom veľkého množstva užitočných informácií. Jedným z možných spôsobov ich využitia je práve identifikovanie častí programov, resp. zdrojových kódov, ktoré s vysokou pravdepodobnosťou obsahujú chyby. Takýto druh informácií sa však často nezískava a ani ďalej nevyužíva. Problematikou sa však zaoberajú viaceré práce [13, 7, 24].

2.2 Evolúcia zdrojových kódov

Nezanedbateľné množstvo výskumu sa venuje aj evolúcii softvérových systémov. Pri ňom sa však len málokedy dá spoliehať napr. na príslušnú dokumentáciu, alebo popisy zmien uvedené vo verziovacích systémoch. Najlepším a najpresnejším zdrojom informácií o evolúcii softvérových projektov sú ich zdrojové kódy [10]. V literatúre bolo navrhnutých viacero rôznych techník, ktoré získavajú informácie z histórie zdrojových kódov a vďaka ktorým je možné presne sledovať vývoj príslušných softvérových systémov.

Purushothaman et al. [24] dôkladne analyzovali 15 ročnú históriu zdrojových kódov telefónneho prepínača (angl. *switch*), vyvíjaného spoločnosťou Lucent Technologies. Celá história pozostávala približne zo 100 miliónov riadkov kódu v C a C++ a ďalších 100 miliónov riadkov hlavičkových a tzv. *make* súborov. Pri analýze využili informácie zo systému na verziovanie, sledovanie zmien a chýb. Tie mohli autori považovať za dostatočne presné, pretože spoločnosť má striktne definované procesy pre používanie týchto systémov [24].

Autori sa zameriavajú na analýzu malých zmien. Spomínajú, že programátori sa obvykle nezaoberajú ich následkami, pretože ich intuitívne považujú za bezvýznamné. Často po nich nevykonávajú ani testovanie. Cieľom ich práce bolo teda preskúmať, či a ako sa malé zmeny líšia od veľkých, aké sú medzi nimi vzťahy, či bývajú malé zmeny zdrojom chýb a pod. [24].

[1] http://findbugs.sourceforge.net/

Pri analýze Purushothaman et al. [24] rozlišujú 4 druhy zmien, podľa cieľa, s akým boli vykonané:

- *korektívne* – opravujú chyby, 32,8% všetkých zmien,
- *vylepšujúce* – prinášajú nejaké vylepšenie, napr. z pohľadu výkonnosti, udržiavateľnosti a pod., 8% všetkých zmien,
- *adaptívne* – prinášajú novú funkcionalitu, 47,8% všetkých zmien,
- *po inšpekcii* – zmeny vykonané po inšpekcii kódu, 9% všetkých zmien.

Jednoriadkové zmeny mali približne rovnaké rozloženie, obsahovali viac korektívnych (39,6%) a menej adaptívnych zmien (37,6%).

Ďalej autori zmeny rozdeľujú podľa toho, aké operácie boli kvôli nim vykonané v zdrojovom kóde:

- *modifikácia* – úprava existujúcich riadkov kódu, 28,1% všetkých zmien,
- *pridanie* – vytvorenie nových riadkov kódu, 40,1% všetkých zmien,
- *odstránenie* – vymazanie riadkov kódu, 3,1% všetkých zmien,
- *kombinácie uvedených* – 28,7% všetkých zmien.

Pre jednoriadkové zmeny bolo rozloženie odlišné. Až 65% tvorili modifikácie, 28,5% pridania riadkov a v 6,5% prípadov bol riadok kódu odstránený.

Čo sa týka veľkosti jednotlivých zmien, autori zistili, že v 10% zmien vykonaných v jednom súbore sa zmena týkala iba jedného riadku. 50% všetkých zmien ovplyvnilo 10 alebo menej riadkov kódu a 95% všetkých zmien ovplyvnilo 50 a menej riadkov kódu [24].

Ďalšie experimenty boli zamerané na analýzu závislostí medzi zmenami. Až v 60% prípadov boli raz modifikované riadky kódu neskôr zmenené znova (autori ale neuvádzajú koľko času medzi takýmito zmenami uplynulo). Až 40% zmien vykonaných s cieľom opraviť chyby spôsobilo ďalšie chyby [24].

Autori ďalej zistili, že zmena jedného riadku spôsobila chybu iba v 4% prípadov (pridanie nového riadku v približne 2%, modifkácia v 5%). Pre väčšie počty riadkov hodnoty stúpali a boli približne rovnaké pre pridanie aj modifikáciu. Až pri 50 riadkoch sa medzi pridávaním a modifikáciou objavili väčšie rozdiely, nové riadky mali viac následných chýb. To je v súlade s intuitívnym predpokladom o tom, že programátori si pri modifikácii kódu dávajú väčší pozor, ako pri vytváraní nového.

Autori ďalej tvrdia, že sa im nepodarilo nájsť dôkaz o tom, že odstránenie menej ako 10 riadkov kódu by niekedy spôsobilo chybu. Tieto výsledky, spolu s celkovo nízkym množstvom vymazaného kódu, podporujú intuitívny

predpoklad o tom, že programátori mažú len taký kód, o ktorom sú úplne presvedčení, že nie je potrebný.

Purushothaman et al. [24] v článku ale upozorňujú, že získané informácie môžu byť do značnej miery ovplyvnené procesmi vývoja a vykonávanými inšpekciami kódu.

2.3 Predpovedanie chýb s využitím evolúcie kódu

Ďalšia skupina metód, ktorá môže pomôcť pri predpovedaní chýb a kvality softvéru, je založená na využívaní informácií o jeho evolúcii. Základom takýchto predpovedí môže byť skutočnosť, že často meniaci sa kód má väčšiu chybovosť. Zvýšenú pozornosť by bolo možno vhodné venovať aj novému kódu. Naopak, časti, ktoré dlhú dobu neboli zmenené, a teda pravdepodobne pracujú správne, majú nízku pravdepodobnosť chýb.

Evolúcia kódu môže byť dokonca použitá aj v kombinácii s metódami uvedenými vyššie. Lozano et al. [19] napríklad hovoria o tom, ako by sledovanie postupného vývoja jednotlivých antivzorov v kódoch mohlo pomôcť sformulovať pravidlá vývoja softvéru, ktoré by znížili jeho chybovosť. Systémy na podporu vývoja tiež obsahujú neoceniteľné informácie v súvislosti s evolúciou softvérových systémov a ich zdrojových kódov. Podobné experimenty ako vykonali napr. Purushothaman et al. [24] by mohli odhaliť, aké konkrétne zmeny alebo typy zmien zdrojových kódov vedú k následným chybám.

Kim et al. [16] predstavujú vo svojej práci teoretický model a praktickú implementáciu nástroja, ktorý predpovedá chyby softvérových systémov na základe ich evolúcie.

Autori predstavujú *BugCache* a *FixCache*, čo sú dva rôzne zoznamy chybných fragmentov kódu. *BugCache* je aktualizovaná hneď vtedy, keď je chyba v zdrojovom kóde vytvorená. V skutočnosti však vytvorenie chyby nevieme odhaliť okamžite. *BugCache* je preto len teoretický model. Dá sa použiť až v momente, keď je známa kompletná história projektu. *FixCache* je na druhej strane aktualizovaná až vtedy, keď je chyba opravená. Algoritmus nájde v histórii zmenu, ktorá chybu spôsobila [15, 30] a podľa toho pridá nové, podozrivé časti kódu. Je to teda reálna implementácia predpovedania chýb.

Pri pridávaní potencionálne chybných fragmentov algoritmus používa tri pravidlá [16]:

1. *priestorová lokálnosť* – ak bola v nejakej časti kódu chyba, budú sa v nej s vysokou pravdepodobnosťou nachádzať aj iné, ďalšie chyby,

2. *časová lokálnosť* – ak sa v kóde nachádza chyba, je pravdepodobné, že sa bude nachádzať aj v súvisiacich kódoch (ako súvisiaci kód autori označujú také fragmenty, ktoré sú často menené spolu),

3. *modifikácie a pridávanie nového kódu* – nedávno zmenené alebo vytvorené fragmenty kódu majú vyššiu pravdepodobnosť, že obsahujú chyby.

Fragmenty kódu sú zo zoznamu podozrivých odstránené vtedy, keď je zoznam plný (má iba obmedzenú veľkosť). Kandidátov na vymazanie autori vyberajú (1) na základe času, kedy v nich bola naposledy odhalená chyba, alebo (2) podľa toho, v ktorých fragmentoch bolo opravených najmenej chýb.

Pri použití najlepších nastavení bola presnosť predikcie 73 - 95%. Testy autori vykonávali na siedmich open-source projektoch [16].

Rahman et al. [25] vo svojej práci vyhodnocujú úspešnosť *FixCache* [16] a porovnávajú ju s inými, štandardnými technikami predpovedania chýb. Experimenty vykonávali na piatich open source projektoch.

Vykonané experimenty potvrdili, že súbory vo *FixCache* majú naozaj väčšiu hustotu chýb, ako ostatné súbory. Autori tiež zistili, že *FixCache* má tendenciu dávať do zoznamu podozrení také súbory, ktoré obsahujú viac kódu. Ak bolo v zozname 10% súborov, tak obsahovali spolu 20 - 30% všetkého kódu [25]. Ďalej sa ukázalo, že časová lokálnosť má pri pridávaní súborov väčší efekt, ako ostatné dve pravidlá pridávania. Rahman et al. [25] dokonca ukazujú, že ostatné pravidlá nemajú skoro žiaden reálny prínos a časová lokálnosť je zodpovedná za takmer celú prediktívnu silu *FixCache*. Autori tiež zistili, že najlepšie výsledky *FixCache* dosahuje, ak ruší svoje podozrenia na základe počtu opravených chýb. Rahman et al. [25] na záver *FixCache* porovnali s naivnou metódou, ktorá zoradí súbory podľa počtu predchádzajúcich chýb. Najprekvapivejším zistením bolo, že *obidva spôsoby mali rovnakú úspešnosť.*

Lewis et al. [18] sa inšpirovali zisteniami Rahman et al. [25] a implementovali predpovedanie chýb založené na počte už opravených chýb. Pri praktickom nasadení nástroja sa ukázalo, že takýto algoritmus sa len pomaly adaptuje na zmeny (t.j. ak bol nejaký kód opravený, aj tak bude ešte dlho označovaný za podozrivý). Preto navrhujú dávať kódu s nedávno opravenými chybami vyššiu prioritu, ako kódu, v ktorom boli chyby opravené dávnejšie.

3

Abstraktné syntaktické stromy

Stromové dátové štruktúry sa v oblasti spracovávania zdrojových kódov (napr. pri vývoji kompilátorov) stali veľmi obľúbenými. Predtým, ako kompilátor zdrojový kód preloží, musí vykonať viacero krokov. Jedným z nich je aj vytvorenie takej reprezentácie kódu, ktorá neobsahuje zátvorky, bodkočiarky, atď. – t.j. tie časti zdrojového kódu, ktoré nijako neovplyvňujú výsledný program. Táto reprezentácia ďalej napríklad zjednocuje viaceré možné zápisy rovnakých konštrukcií (odstraňuje tzv. *syntactic sugar*) a pod. Jednou z takýchto reprezentácií sú práve abstraktné syntaktické stromy.

Zdrojový kód môže byť do textového súboru zapísaný rôznymi spôsobmi. Abstraktné syntaktické stromy však nie sú ovplyvnené týmto zápisom. Práve takáto abstrakcia umožňuje jednoduchšie spracovanie a sofistikovanejšiu analýzu zdrojových kódov.

Knižníc, ktoré zo zdrojových kódov programov vytvárajú abstraktné syntaktické stromy, vo všeobecnosti existuje málo. Väčšinou sú súčasťou kompilátorov alebo vývojových prostredí a takmer vždy pracujú iba s jedným programovacím jazykom. Od výberu konkrétneho nástroja (a podporovaného jazyka) závisí celá implementačná časť práce.

Existujú aj univerzálne nástroje, ktoré umožňujú vytvoriť abstraktné syntaktické stromy z kódov viacerých (v ideálnom prípade všetkých) programovacích jazykov. Takýto nástroj by bol vhodný najmä preto, že by sa nebolo potrebné obmedzovať na jeden konkrétny cieľový jazyk.

Java Development Tools (Java). Vývojové prostredie Eclipse obsahuje sadu nástrojov s názvom Java Development Tools[1], ktoré obsahujú knižnice na vytváranie abstraktných syntaktických stromov zo zdrojových kódov programov napísaných v jazyku Java.

Na API, ktoré knižnica ponúka, sme sa pozreli detailnejšie. Jeho veľkou nevýhodou je, že vrcholy stromu neumožňujú iteráciu cez všetky deti. Každý typ vrcholu má presne definované detské vrcholy, ku ktorým sa dá dostať len cez príslušné *get* metódy. To znamená, že pri spracovaní takéhoto stromu by

[1] http://www.eclipse.org/jdt/

bolo treba mať výber detí pre každý z niekoľkých desiatok typov vrcholov ošetrený samostatne.

NRefactory (C#). Open-source vývojové prostredie iSharpDevelop obsahuje NRefactory[2]. Tento nástroj vytvára abstraktné syntaktické stromy z kódov programov v jazyku C#.

Roslyn (C#). Spoločnosť Microsoft pracuje na kompilátore jazyka C# s názvom Roslyn[3]. Ten taktiež umožňuje získať reprezentáciu zdrojových kódov C# programov vo forme abstraktných syntaktických stromov. Kompilátor je momentálne stále vo vývoji, dostupná je len jeho ukážka, aj keď verejné API by sa už v budúcnosti meniť nemalo.

RubyParser (Ruby). Zdrojové kódy napísané v jazyku Ruby sa dajú parsovať na abstraktné syntaktické stromy knižnicou RubyParser[4]. Knižnica vytvorí veľmi jednoduchý a prehľadný strom reprezentovaný pomocou S-výrazov (angl. *S-expressions*).

Ripper (Ruby). Ripper[5] je API, ktoré je súčasťou oficiálneho Ruby interpretera od verzie 1.9. Ním vytvorené abstraktné syntaktické stromy sú tiež reprezentované pomocou S-výrazov, v porovnaní s knižnicou RubyParser sú však omnoho väčšie a zložitejšie. K nástroju Ripper chýba dokumentácia, preto je jeho praktické použitie veľmi obtiažne.

Antlr. Antlr[6] na základe definície gramatiky ľubovoľného jazyka automaticky vytvorí pre tento jazyk parser, kompilátor alebo interpreter, atď. Následne umožňuje pristupovať aj k abstraktnému syntaktickému stromu, ktorý sa interne vytvorí počas spracovávania príslušného zdrojového kódu. Veľkým obmedzením však je, že aj gramatiky najpopulárnejších jazykov pre Antlr sú vytvárané iba jednotlivcami, často nie sú plne funkčné a ich údržba a zlepšovanie nie sú nijako koordinované, v mnohých prípadoch ani neprebiehajú.

The DMS Software Reengineering Toolkit je proprietárna sada nástrojov[7] pre automatickú analýzu, modifikáciu alebo preklad zdrojových kódov rôznych programovacích jazykov. Súčasťou je aj vytváranie abstraktných syntaktických stromov. Nástroje sú ale platené.

[2] https://github.com/icsharpcode/NRefactory/
[3] http://msdn.microsoft.com/en-us/roslyn
[4] https://github.com/seattlerb/ruby_parser
[5] http://www.ruby-doc.org/stdlib-1.9.3/libdoc/ripper/rdoc/Ripper.html
[6] http://www.antlr.org/
[7] http://www.semanticdesigns.com/Products/DMS/DMSToolkit.html

4

Detekcia zduplikovaného kódu

Kopírovanie existujúcich častí kódov na iné miesta je nepochybne najpopulárnejší spôsob znovupoužitia kódu, napriek tomu, že sa vo všeobecnosti považuje za nežiadúce. Doterajší výskum v oblasti údržby softvéru naznačuje, že softvérové systémy obsahujú 5-10% zduplikovaného kódu [1, 11], čo určite nie je zanedbateľné množstvo. M. Fowler et al. [9] dokonca uvádzajú zduplikovaný kód ako jeden z ich známych a odbornou verejnosťou akceptovaných pachov zdrojových kódov.

Hlavným dôvodom, kvôli ktorému sa zduplikovaný kód považuje za nežiadúci, je jeho náchylnosť na chyby. Každý programátor vie, že nekonzistentné zmeny jednotlivých duplikátov bývajú bežným zdrojom problémov.

Odhaľovaniu zduplikovaného kódu sa v súčastnosti venuje veľká pozornosť a existuje široká škála spôsobov, pomocou ktorých sa dá vykonávať. Krátke prehľady uvádzajú viaceré práce [26, 4, 11, 17].

Metódy využívajúce porovnávanie textu alebo metriky kódu. Najtriviálnejším spôsobom odhaľovania zduplikovaného kódu je priame porovnávanie textu zdrojových súborov, alebo porovnávanie kódu na základe jeho rôznych vlastností. Aj sofistikovanejšie varianty týchto metód sú ale citlivé na malé zmeny (napr. vo formátovaní) a v praxi sa nepoužívajú [4].

Metódy založené na tokenoch. Na vylepšenie metód, ktoré porovnávali len text zdrojových kódov, sa začala používať tokenizácia [4]. Tá abstrahuje od konkrétneho zápisu zdrojového kódu a detailov jazyka. Zduplikovaný kód sa následne dá odhaľovať porovnávaním postupností vytvorených tokenov.

Metódy využívajúce grafy závislostí. Ďalším zaujímavým spôsobom je modelovanie programov ako grafov volaní funkcií alebo toku dát. Izomorfné (t.j. rovnaké, podobné) podgrafy potom reprezentujú zduplikovaný kód [17]. Hľadanie izomorfizmu grafov je vo všeobecnosti NP úplný problém, v praxi sa takéto hľadanie duplikátov rôzne optimalizuje [4].

Metódy využívajúce abstraktné syntaktické stromy. Pre našu prácu je najzaujímavejšou práve táto skupina metód. Pri odhaľovaní zduplikovaného kódu vyhodnocuje podobnosť abstraktných syntaktických stromov. Podobne, ako pri metódach využívajúcich tokenizáciu, aj tu dochádza k abstrakcii od textového zápisu zdrojového kódu. Pri použití tejto skupiny metód sa však ťažšie odhaľujú čiastočne zmenené duplikáty.

4.1 Jednorazové odhaľovanie duplikátov

Väčšina z existujúcich metód a nástrojov na hľadanie zduplikovaného kódu vykonáva kontrolu len jednorázovo [11]. V nasledujúcom texte sa zameriame najmä na tie, ktoré využívajú abstraktné syntaktické stromy.

4.1.1 Prehľad existujúcich jednorazových metód

Baxter et al. [1]. Autori proces odhaľovania zduplikovaného kódu v abstraktných syntaktických stromoch delia na tri fázy: (1) hľadanie zduplikovaných podstromov, (2) hľadanie zduplikovaných sekvencií podstromov, (3) zovšeobecňovanie duplikátov – pre každý zduplikovaný podstrom sa kontroluje, či nie je zduplikovaný celý podstrom jeho rodiča. V súvislosti s odhaľovaním duplikátov autori opisujú tri hlavné problémy:

- *škálovateľnosť* – počet podstromov je vo všeobecnosti veľmi veľký na to, aby sa všetky dali navzájom porovnať,
- *odhaľovanie podobných podstromov* – v praxi nás nezaujímajú iba úplne odkopírované kódy, ale aj čiastočne modifikované,
- *problém podklonov* – je potrebné vždy odhaľovať iba najväčšie duplikáty.

Sager et al. [27]. Autori v článku opisujú experimenty s cieľom odhaľovať podobné triedy napísané v jazyku Java s využitím abstraktných syntaktických stromov pomocou: (1) najväčšieho rovnakého podstromu hľadaného zdola nahor, (2) najväčšieho rovnakého podstromu hľadaného zhora nadol a (3) pomocou rozdielov stromov (cena zoznamu zmien). Prezentované výsledky hovoria, že tretí spôsob dosahuje výrazne lepšie výsledky ako prvé dva.

4.1.2 Existujúce jednorazové nástroje

Nástrojov, ktoré jednorazovo odhaľujú zduplikovaný kód existuje veľký počet. Roy et al. [26] mnohé z nich vo svojej práci dôkladne porovnávajú a vyhodnocujú ich úspešnosť. Napriek najlepšej snahe autorov sú však aj ich výsledky a závery často nejednoznačné. Ďalšie takéto práce sme nenašli. Podľa nášho

názoru to implikuje, že porovnávanie rôznych detektorov duplikátov je mimoriadne ťažká úloha. Toto tvrdenie podporujú aj Chatterji et al. [3], ktorí uvádzajú, že autori jednotlivých metód a nástrojov často využívajú rozdielne predpoklady o zduplikovanom kóde a následne aj rôzne definície duplikátov. Na záver zhodnocujú, že v oblasti automatickej detekcie duplikátov chýba zhoda v tom, čo vlastne zduplikovaný kód je.

Na ukážku vyberáme niektoré nástroje, ktoré dokážu spracovať zdrojové kódy napísané v jazyku Ruby:

- *Simian*[1] je nástroj, ktorý odhaľuje duplikáty porovnávaním textu, čo mu umožňuje podporovať širokú škálu programovacích jazykov; na druhej strane neberie do úvahy špecifiká týchto jazykov,

- *Flay*[2] je detektor vytvorený pre jazyk Ruby; využíva hašovanie abstraktných syntaktických stromov so zachovaním informácií o ich štruktúre,

- *RubyMine*[3] je IDE pre vývoj v Ruby a obsahuje zabudovanú detekciu duplikátov; nepodarilo sa nám zistiť, na akom princípe je založená.

4.2 Inkrementálne odhaľovanie duplikátov

Pre softvér, ktorý sa neustále mení, či už vo vývoji alebo údržbe, znamenajú jednorázové metódy odhaľovania duplikátov jednu z nasledujúcich troch možností [20]:

1. kontrola sa musí vykonávať opakovane na všetkých zdrojových kódoch, čo môže byť výpočtovo a časovo náročné,

2. skontrolujú sa iba tie skupiny duplikátov, ktoré už boli odhalené v minulosti, čo môže spôsobiť, že nebudú odhalené nové duplikáty,

3. do kontroly sa zahrnú iba súbory, ktoré boli od poslednej kontroly zmenené, čo môže opäť spôsobiť, že niektoré duplikáty nebudú odhalené.

Hummel et al. [11] a Nguyen et al. [20] preto tvrdia, že praktická použiteľnosť nástrojov na odhaľovanie zduplikovaného kódu je závislá na tom, aby vedeli fungovať *inkrementálne*. Inkrementálna detekcia duplikátov však musí byť rovnako kompletná a presná ako opakovaná kontrola všetkých zdrojových kódov (t.j. ako v prípade možnosti č. 1).

[1] http://www.harukizaemon.com/simian/
[2] http://ruby.sadi.st/Flay.html
[3] http://www.jetbrains.com/ruby/

4.2.1 Prehľad existujúcich inkrementálnych metód

Hummel et al. [11]. Autori v článku prezentujú algoritmus, ktorý využíva tokenizáciu kódov a ich následné indexovanie.

Indexovanie vykonávajú na úrovni fragmentov kódu. Jeden fragment je tvorený n po sebe nasledujúcimi normalizovanými a tokenizovanými príkazmi. Zaindexované sú všetky možné n-tice po sebe nasledujúcich príkazov, pričom sa navzájom prekrývajú (t.j. n príkazov tvorí jeden fragment, $n + 1$ príkazov tvorí dva fragmenty, atď.). Index je zoznam týchto n-tíc. Umožňuje vyhľadávanie podľa zahašovaných hodnôt alebo podľa názvov súborov.

Veľkosť zduplikovaného kódu môže byť väčšia ako veľkosť n-tíc. Najväčšie duplikáty Hummel et al. [11] odhaľujú tak, že spájajú po sebe nasledujúce zduplikované fragmenty do jedného.

Čo sa týka udržiavania indexu aktuálneho, Hummel et al. [11] používajú len dve operácie – pridanie nového súboru a vymazanie existujúceho súboru. Tento prístup však môže byť zbytočne výpočtovo náročný pri zmene súborov. Purushothaman et al. [24] totiž tvrdia, že pri modifikácii súboru je až v 50% prípadov ovplyvnených len 10 alebo menej riadkov kódu.

Hummel et al. [11] uvádzajú aj experimenty s implementáciou prezentovaného algoritmu. Vytvorenie indexu pre približne 42 miliónov riadkov kódu (v 210 tisíc súboroch) trvalo 7 hodín a index zaberal trikrát viac miesta, ako samotný kód. Hľadanie jedného fragmentu podľa zahašovanej hodnoty a kompletné vymazanie a pridanie jedného súboru trvali v priemere necelú sekundu.

Chilowicz et al. [4]. V tomto článku je prezentované inkrementálne odhaľovanie duplikátov, ktoré využíva abstraktné syntaktické stromy. Pri ich indexovaní autori každý podstrom (smerom od listov ku koreňu) reprezentujú ako trojicu: (1) veľkosť podstromu, (2) zahašovanú hodnotu, ktorá zohľadňuje štruktúru podstromu, (3) odkaz na koreň daného podstromu a jeho rodiča. Ako duplikáty sú dva stromy označené len ak majú rovnakú veľkosť a zahašovanú hodnotu. Podstromy menšie ako definovaná hranica sa preskakujú.

Index abstraktných syntaktických stromov je zložený z dvoch rôznych B+-k-stromov [4]:

1. prvý umožňuje vyhľadávanie podľa zahašovaných hodnôt a je zoradený podľa veľkosti podstromov, čo umožňuje odhaľovať najväčšie duplikáty,

2. druhý je zoradený podľa rodičov zaindexovaných podstromov, umožňuje vyhľadávať všetky podstromy zadaného vrcholu.

Zahašované hodnoty použité v indexe zohľadňujú štruktúru podstromov. Ich výpočet je rozdielny pre jednotlivé vrcholy a pre celé podstromy.

Vrcholy majú priradené konštantné zahašované hodnoty podľa toho, aký je ich typ. Všetky vrcholy rovnakého typu majú teda rovnakú zahašovanú hodnotu. Podľa toho, aké duplikáty sa majú odhaľovať, autori odporúčajú, aby mali aj niektoré rôzne typy vrcholov rovnakú zahašovanú hodnotu (napr. *for* a *while*, alebo všetky číselné primitívne typy a pod.). Funkciu, ktorá počíta konštantné zahašované hodnoty pre ľubovoľný vrchol n budeme ďalej označovať $H_c(n)$.

Vytváranie zahašovaných hodnôt pre celé podstromy je o niečo komplikovanejšie. Autori experimentujú s nasledujúcimi spôsobmi:

Hašovanie pomocou Dyckových slov. Dyckove slovo $D(l)$ každého listu stromu l je určené dvoma celými číslami $H_1(l)$ a $H_2(l)$, ktoré sú vypočítané z hodnoty $H_c(l)$. Pre vútorný vrchol n s deťmi $c_1, ..., c_k$ autori Dyckove slovo $D(n)$ určujú ako postupnosť $H_1(n)D(c_1)...D(c_n)H_2(n)$. Dyckove slová sú nakoniec ešte hašované Karp-Rabinovou funkciou.

Hašovanie kryptografickými funkciami. Zahašovaná hodnota ktoréhokoľvek podstromu s koreňom n a deťmi $c_1, ..., c_k$, pričom C je kryptografická hašovacia funkcia (MD5 alebo SHA-1) sa vypočíta rekurzívne ako $H(n) = C(H_c(n)H(c_1)...H(c_k))$.

Chilowicz et al. [4] vykonali experimenty s uvedenými spôsobmi vytvárania zahašovaných hodnôt. Počet kolízií bol vo všetkých prípadoch takmer rovnaký. Výpočtový čas potrebný na hašovanie pomocou Dyckových slov sa zväčšuje s veľkosťou podstomov, prakticky bol však o niečo rýchlejší ako výpočet s kryptografickými funkciami. Autori ale argumentujú, že algoritmus bol obmedzovaný rýchlosťou vstupno-výstupných operácií pri práci s indexom, a preto rozdiely v čase výpočtu pri hašovaní nehrali dôležitú úlohu.

Vzhľadom na povahu hašovacích funkcií je zrejmé, že ako duplikáty môžu byť označené aj úplne rozdielne podstromy. Autori preto navrhujú kontrolovať to, či obidva podstromy obsahujú rovnaké deti. Táto kontrola sa dá ľahko vykonať vďaka druhej časti indexu.

Chilowicz et al. [4] sa zaoberajú aj hľadaním postupností viacerých zduplikovaných podstromov. Vstupom do tohto procesu sú skupiny už odhalených duplikátov. Samotné odhaľovanie sa vykonáva pomocou príponových polí a intervalových stromov.

Vzhľadom k tomu, že malé podstromy sú pri indexovaní preskakované, prezentovaná metóda neodhalí ani veľké zduplikované postupnosti malých podstromov (napr. nejakú odkopírovanú postupnosť príkazov vo funkcii). Autori tento problém podrobnejšie nespomínajú. Navrhujú iba spájať malé susedné podstromy už pri indexovaní a vložiť ich do indexu ako jeden záznam. Toto riešenie však problém neodstráni úplne. Ak napríklad vrchol stromu obsahuje 10 detí a do indexu vložia 2 intervaly – interval 1-5 a 6-10, zduplikovaná

postupnosť príkazov 3-8 nebude odhalená. Jednou z možností, ako problém odstrániť, je indexovať postupnosti podobne ako Hummel et al. [11].

Nguyen et al. [21]. Autori v článku prezentujú ďalšiu inkrementálnu metódu odhaľovania duplikátov, ktorá využíva abstraktné syntaktické stromy. Podobnosť podstromov počítajú ako vzdialenosť ich charakteristických vektorov, ktoré sú vytvárané s využitím *Exas* [20]. Výhodou tejto metódy je, že nehľadá len presné kópie kódu, ale umožňuje odhaliť aj zmenené duplikáty.

Exas je algoritmus na extrakciu vlastností orientovaných grafov (resp. stromov). Získané vlastnosti sú špecifické tým, že zachytávajú aj štruktúru analyzovaných grafov, vďaka čomu je Exas vhodný na ich vzájomné porovnávanie. V grafoch vyhľadáva nasledujúce zložky:

- (p, q)-vrcholy – všetky vrcholy grafu rovnakého typu, ktoré majú p vstupných hrán a q výstupných hrán,
- n-cesty – všetky postupnosti vrcholov dĺžky n, ktoré sú spojené hranou.

Charakteristický vektor grafu, ktorý Exas vytvorí, obsahuje počet výskytov jeho jednotlivých (p, q)-vrcholov a n-ciest. Táto reprezentácia bola inšpirovaná nástrojom *Deckard* [12], ktorý daný vektor vytvára len z počtu výskytov jednotlivých typov vrcholov. Nezachytáva tak žiadne informácie o štruktúre. Deckard môžeme zároveň považovať za špeciálny prípad algoritmu Exas, ak sa pri vytváraní vektora použijú iba 1-cesty (t.j. samostatné vrcholy) [20].

V [21], pri použití Exas so stromami, autori nepoužívajú (p, q)-cesty. Namiesto nich do charakteristického vektora vkladajú všetky možné postupnosti súrodeneckých vrcholov. Dôvody zmeny neuvádzajú.

Pri odhaľovaní duplikátov hľadajú Nguyen et al. [21] také podstromy, resp. fragmenty kódov, ktorých charakteristické vektory majú malú vzájomnú Manhattanskú vzdialenosť:

$$vzdialenosť = 1 - \frac{2 * \|v_1 - v_2\|_1}{\|v_1\|_1 + \|v_2\|_1} \tag{4.1}$$

Výsledná hodnota je relatívna vzhľadom na veľkosť stromu, t.j. väčšie duplikáty môžu byť viac odlišné a stále správne odhalené.

Porovnávanie charakteristických vektorov všetkých podstromov by bolo výpočtovo náročné. Autori preto najprv vektory rozdelia do skupín a následne počítajú vzdialenosti len v jednej skupine. Vytváranie skupín vykonávajú pomocou lokálne senzitívneho hašovania. Pravdepodobnosť kolízie sa pri ňom zvyšuje so zvyšujúcou sa podobnosťou hašovaných vektorov [21] (t.j. podobné vektory majú podobnú zahašovanú hodnotu). Kolízie môžu stále nastať a rozdielne kódy môžu byť vložené do rovnakej skupiny, alebo duplikáty do rôznych skupín. Na odstránenie problému autori charakteristický vektor podstromu

hašujú viackrát a vkladajú do viacerých skupín, čo znižuje pravdepodobnosť chýb. Lokálne senzitívne hašovanie je podrobnejšie opísané v prílohe A.1.

Vo vytvorených skupinách sa podozrenia na duplikáty potvrdzujú (vzorec 4.1). Potvrdené duplikáty autori následne po dvojiciach vkladajú do finálnych skupín duplikátov, resp. grafu duplikátov:

- ak ani jeden fragment ešte nie je zaradený v skupine, vytvorí sa nová,
- ak už jeden z fragmentov je v niektorej skupine duplikátov, druhý fragment sa vloží do tej istej skupiny,
- ak sa každý fragment nachádza v inej skupine, obe sa spoja do jednej.

Najväčšie možné duplikáty Nguyen et al. [21] odhaľujú tak, že hľadajú skupiny, ktorých všetky fragmenty kódov sú obsiahnuté vo fragmentoch inej skupiny. Detaily implementácie tohto záverečného spracovania autori neopisujú.

Nguyen et al. [21] uvádzajú aj výsledky praktických experimentov s implementáciou opísaného algoritmu. Jednorázová analýza kódov JDK 5 trvala 7 minút, pri hranici podobnosti 0,95 uvádzajú autori presnosť 96%. Jednorázová analýza jednej (poslednej) revízie kódov GEclipse trvala 39 sekúnd, postupná analýza revízií 1 000 – 2 000 trvala 500 sekúnd (0,5 sekundy na 1 revíziu).

4.3 Je zduplikovaný kód vždy nežiadúci?

Kapser a Godfrey [14] vo svojej práci uvádzajú rozsiahly zoznam dôvodov, resp. situácií pri ktorých dochádza k duplikovaniu zdrojových kódov. Zaujímavosťou je, že nie všetky z nich považujú autori za nežiadúce, napr. ak vznikli z dôvodu rozdielov platforiem alebo hardvéru, z dôvodu nedostatkov použitého programovacieho jazyka a pod.

Ďalší smer v rámci výskumu zduplikovaného kódu sa zaoberá sledovaním jeho evolúcie. Pate et al. [22] poskytujú vyčerpávajúci prehľad viacerých takýchto prác. Medzi niektoré zo zaujímavých zistení patria:

- veľký počet duplikátov v systémoch ostával dlhú dobu, čo implikuje, že duplikáty sa viac pridávali ako odoberali,
- konzistentné zmeny sa vykonávali priemerne v 40% skupín s duplikátmi,
- ak sa v zduplikovanom kóde vykonala nekonzistentná zmena, len málokedy sa neskôr opravila na konzistentnú; to naznačuje, že duplikáty sa ďalej (úmyselne) vyvíjali nezávisle,
- 2 štúdie tvrdia, že až približne 70-80% duplikátov sa počas sledovaného obdobia nezmenilo vôbec [28].

Zistenia uvedené vyššie hovoria o tom, že (1) konzistentné zmeny v duplikátoch sú vykonávané pomerne často a zároveň hovoria, že (2) nekonzistentné zmeny sú väčšinou vykonávané naschvál. Môže sa teda zdať, že zduplikovaný kód nespôsobuje toľko chýb, aby sa nimi bolo potrebné zaoberať. V uvedených prácach boli ale duplikáty sledované *v produkčných verziách* softvéru. Tie predtým určite prešli napríklad testovaním, vďaka ktorému mohli byť mnohé chyby spôsobené duplikátmi nájdené a odstránené.

Nenašli sme žiadnu prácu zameranú na sledovanie vývoja duplikátov, ktorá by bola vykonaná s menšou granularitou, napríklad až na úrovni jednotlivých tzv. *commit*-ov vo verziovacom systéme. Takisto sme v dostupnej literatúre nenašli žiadnu štúdiu, ktorá by vyhodnocovala, koľko času vývojári reálne trávia opravovaním chýb súvisiacich s duplikátmi. Pri vývoji systémov s veľkým množstvom zduplikovaného kódu tieto aktivity, na základe vlastných skúseností, môžu zaberať nezanedbateľné množstvo času.

Uvedené práce každopádne naznačujú, že agresívne refaktorovanie zduplikovaného kódu *nemusí byť vždy potrebné* alebo dokonca nemusí byť vôbec možné. Saha et al. [28] dokonca tvrdia, že vhodnejšie by bolo len sledovať evolúciu zdrojových kódov a duplikátov v nich. Vďaka tomu bude možné upriamiť pozornosť vývojárov na duplikáty, ktoré sú naozaj nežiadúce a mohli by spôsobiť chyby. Napríklad také, ktoré boli nekonzistentne zmenené.

5

Návrh riešenia

5.1 Definovanie problému

Softvérové projekty obsahujú nezanedbateľné množstvo zduplikovaného kódu – často až okolo 5-10% [1]. Práce prezentované v kapitole 4.3 na druhej strane naznačujú, že zduplikovaný kód v produkčných verziách softvéru nespôsobuje veľa chýb. Ďalej hovoria o tom, že skupiny duplikátov bývajú často upravované konzistentne, čo podľa nášho názoru znamená jednu z týchto dvoch možností:

1. vývojári si pamätajú, kde všade sa zduplikovaný kód nachádza a vďaka tomu v ňom vedia vykonávať konzistentné zmeny,

2. chyby spôsobené nekonzistentnými zmenami sa odhalia počas testovania.

Každý z uvedených dvoch prípadov nepochybne spôsobuje zbytočné zvýšenie nákladov na vývoj. Nenašli sme však žiadne štúdie, ktoré by vyhodnocovali, koľko času trávia vývojári údržbou zduplikovaného kódu.

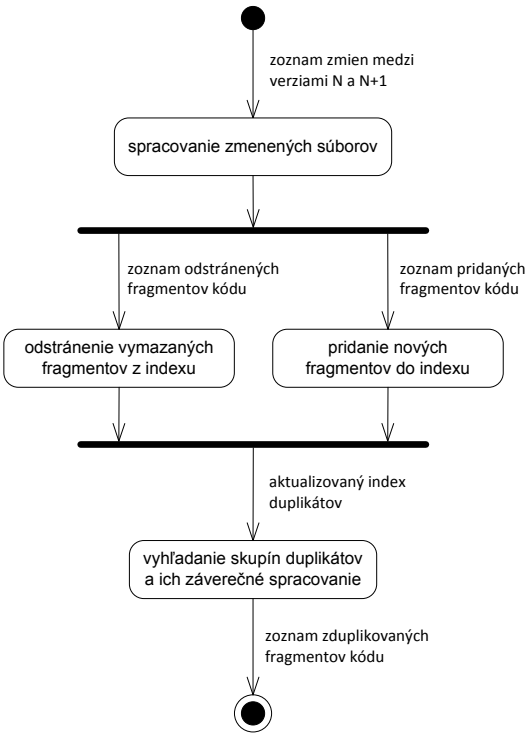
Súhlasíme teda s názorom Saha et al. [28], ktorí tvrdia, že nástroje na detekciu zduplikovaného kódu by mali umožňovať jednoduchšiu údržbu duplikátov počas toho ako sa zdrojový kód mení a systém vyvíja. Na základe toho sme identifikovali nasledujúce požiadavky na takýto nástroj:

- musí vedieť vyhľadávať zduplikovaný kód,

- umožní sledovať evolúciu skupín zduplikovaného kódu medzi viacerými verziami softvéru, pričom verzie môžu mať veľmi malú granularitu,

- podporuje spracovanie malých inkrementálnych zmien kódu, pretože takým spôsobom prebieha vývoj softvéru (t.j. aby pri malej zmene nebolo potrebné znova spracovávať aj nezmenené súbory),

- ak má nástroj fungovať inkrementálne, potrebuje si udržiavať existujúce kódy nejakým spôsobom indexované; tento index by mal byť dosť malý na to, aby ho vývojári mohli mať vo svojich počítačoch,



Obr. 5.1: Všeobecný návrh jednej iterácie inkrementálneho odhaľovania zduplikovaného kódu.

- musí pracovať v reálnom čase, aby vývojári vedeli o duplikátoch a ich zmenách priamo počas práce na danej časti kódu,
- integrovateľný s verziovacími systémami alebo vývojovými prostrediami.

V tejto kapitole opisujeme návrh metódy a nástroja, ktoré spĺňajú všetky uvedené požiadavky.

5.2 Vyhľadávanie zduplikovaného kódu

Detekciu duplikátov sme sa rozhodli vykonávať inkrementálne (kapitola 4.2.1). Na obrázku 5.1 je znázornený diagram, ktorý obsahuje návrh jednej iterácie takéhoto odhaľovania duplikátov. Iterácie budú vykonávané napríklad pri každom vytvorení novej verzie zdrojových kódov vo verziovacom systéme. Vstupom do procesu je zoznam súborov, ktoré boli medzi starou a novou verziou zdrojového kódu zmenené. Výstupom je zoznam skupín zduplikovaného kódu. Celý proces sa skladá zo štyroch menších problémov.

5.2.1 Spracovanie zmenených súborov

Najjednoduchší spôsob, ako zistiť, či zmenené súbory obsahujú alebo neobsahujú zduplikovaný kód, je porovnať ich so všetkými ostatnými súbormi v projekte. Takéto riešenie by určite bolo zbytočne pomalé, najmä pri malých zmenách vo veľkých projektoch. Navyše plánujeme duplikáty odhaľovať s využitím abstraktných syntaktických stromov. Vyhľadávanie spoločných podstromov nie je triviálna úloha a celý proces by ešte viac spomalila.

Navrhujeme preto zdrojové kódy, resp. ich príslušné abstraktné syntaktické stromy indexovať. Fragmenty zdrojových kódov z aktuálnej verzie projektu budeme udržiavať vo vhodnej reprezentácii a vhodnej dátovej štruktúre. Pri zmene zdrojových kódov bude stačiť tento index iba aktualizovať. Tým plánujeme (1) zjednodušiť porovnávanie stromov a (2) zrýchliť vyhľadávanie duplikátov aj vo veľkom množstve kódu.

Pri návrhu konkrétnej metódy indexovania zdrojových kódov sme identifikovali tri problémy:

1. abstraktné syntaktické stromy môžu obsahovať vrcholy, ktoré pri porovnávaní nie sú potrebné alebo ich chceme zámerne ignorovať (napríklad číselné konštanty, reťazce znakov a pod.),

2. nikdy nebude zduplikovaný celý strom, ale len jeho menšie podstromy,

3. potrebujeme vytvoriť takú reprezentáciu stromov, resp. podstromov, ktorá bude zachovávať informácie o ich štruktúre a zároveň ich umožní rýchlo medzi sebou porovnávať.

Pozrime sa teraz na to, ako uvedené problémy navrhujeme riešiť.

Zjednodušovanie stromov. Pri porovnávaní dvoch zdrojových kódov (resp. ich abstraktných syntaktických stromov) je vhodné ignorovať niektoré ich časti. Ide napríklad o názvy premenných alebo funkcií, číselné konštanty, reťazce znakov a podobne. Vo všeobecnosti môžeme povedať, že každý vrchol stromu má svoj typ (napr. definovanie premennej, volanie funkcie) a hodnotu (napr. samotný názov premennej). Pri porovnávaní stromov budeme brať do úvahy len typy vrcholov a nie ich hodnoty.

Abstraktné syntaktické stromy môžu ďalej obsahovať vrcholy, ktoré sa pri ich porovnávaní dajú úplne vynechať (bez straty presnosti porovnávania) alebo nahradiť za iné vrcholy (napríklad *while* a *for*, *switch* a *if*). Kvôli tomu by bolo vhodné stromy predspracovať a zjednodušiť.

Rôzne programovacie jazyky majú však odlišné reprezentácie abstraktných syntaktických stromov (napr. obsahujú odlišné typy vrcholov). To dokonca platí aj pre rôzne parsery jedného jazyka. Ako príklad môžu slúžiť ab-

straktné syntaktické stromy vytvorené knižnicami *RubyParser* a *Ripper* (kapitola 3) pre rovnaký kód napísaný v jazyku Ruby (viď. príloha B.1).

Zjednodušovanie stromov by bolo závislé nielen na programovacom jazyku, ale aj na knižnici použitej pri parsovaní. Očakávame, že by zlepšilo presnosť odhaľovania duplikátov, no zrejme len v malom množstve špecifických prípadov. V ďalšom texte navrhujeme takú metódu porovnávania fragmentov kódu, ktorá dovoľuje, aby mohli byť aj čiastočne odlišné kódy označené ako duplikáty. Práve tam by sa mohla presnosť, ktorú by sme získali zjednodušovaním stromov, stratiť.

Rozhodli sme sa teda pri porovnávaní stromov brať do úvahy len typy vrcholov a nie ich hodnoty. Všetky ostatné možné zjednodušovania stromov sme odložili a sú súčasťou ďalšej možnej práce na projekte.

Rozdeľovanie stromu na menšie časti. Ako už bolo spomínané vyššie, vo väčšine prípadov budú zduplikované len relatívne malé fragmenty zdrojových kódov. Preto abstraktné syntaktické stromy rozdelíme na menšie podstromy a každý z nich budeme ďalej spracovávať samostatne.

Na algoritmus, ktorým sa budú stromy rozdeľovať na menšie podstromy, sme stanovili nasledujúce požiadavky:

1. jednotlivé fragmenty by mali mať približne rovnakú veľkosť,
2. fragmenty by sa mali navzájom prekrývať,
3. fragmenty nemusia nevyhnutne obsahovať všetky úrovne abstraktného syntaktického stromu; zduplikované môžu byť len (1) príkazy v metóde (t.j. len spodné úrovne stromu, listy) alebo len (2) kombinácie príkazov na vyššej úrovni (napr. tri cykly a dve podmienky), pričom ich telá budú odlišné (t.j. len stredné alebo vrchné úrovne stromu, bez listov).

Na základe uvedených požiadaviek sme navrhli algoritmus, ktorý chceme na vytváranie podstromov používať. Vstupom pre algoritmus sú, okrem samotného stromu, ešte dva parametre: (1) h_{max} – maximálna hĺbka a (2) n_{min} – minimálny počet vrcholov vo vytvorených podstromoch. Jeho princíp je načrtnutý vo výpise 5.1.

Pre každý vrchol vstupného stromu vytvoríme práve jeden podstrom. Koreňom tohto podstromu bude práve spracovávaný vrchol, pričom podstrom nebude mať väčšiu hĺbku ako je h_{max}. Takto vytvorené podstromy sa budú prekrývať vertikálne. Ukážka sa nachádza na obrázku 5.2 – maximálna hĺbka podstromov je stanovená na 3 a minimálny počet vrcholov je 5.

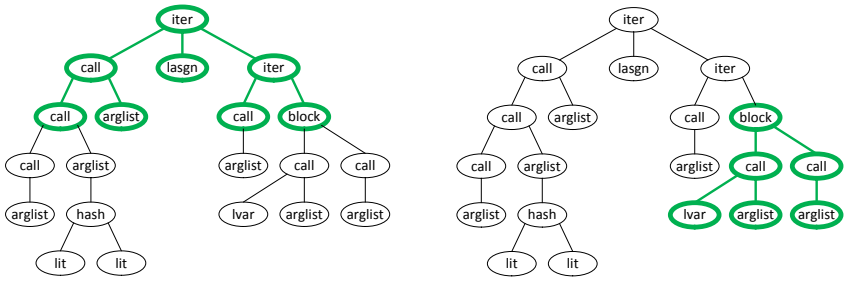
Takýmto algorimom budú, najmä v blízkosti listov, vznikať podstromy, ktoré budú príliš malé (nebudú mať dostatočný počet vrcholov). Bola by však

Algoritmus 5.1 Algoritmus na vytváranie podstromov

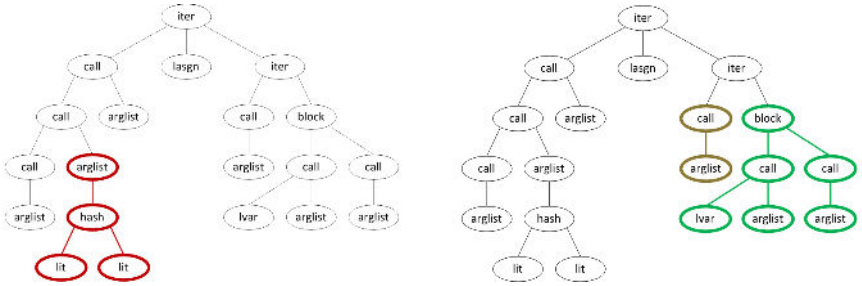
```
1: function VYTVOR PODSTROMY (s, h_max, n_min)
2:     podstromy ← []
3:     for all vrchol k v strome s do
4:         p ← podstrom s koreňom k a hĺbkou h_max
5:         v ← k
6:         while počet vrcholov p < n_min and v má súrodenca vpravo do
7:             v ← pravý súrodenec v
8:             q ← podstrom s koreňom v a hĺbkou h_max
9:             p ← spoj podstromy p a q
10:        end while
11:        if počet vrcholov p >= n_min then
12:            pridaj podstrom p do zoznamu podstromy
13:        end if
14:    end for
15:    return podstromy
16: end function
```

škoda takéto podstromy preskakovať. Najmä, ak sa ich niekde nachádza viacero vedľa seba. Príkladom môže byť ľubovoľná postupnosť jednoduchých výrazov vo funkcii. Ak teda narazíme na malý podstrom, skúsime k nemu pridať aj podstrom z jeho nasledujúceho (t.j. pravého) súrodenca. Toto opakujeme až kým (1) podstrom nebude dostatočne veľký alebo (2) už neexistuje ďalší pravý súrodenec, ktorého by bolo možné do podstromu pridať (a takýto podstrom nakoniec predsa len preskočíme). Obrázok 5.3 vľavo znázorňuje neakceptovaný podstrom a vpravo podstrom, ktorý vznikol spojením dvoch súrodencov.

Celá postupnosť spracovania stromu, ktorý bol znázornený na obrázkoch 5.2 a 5.3, je detailne rozpísaná v prílohe B.2.



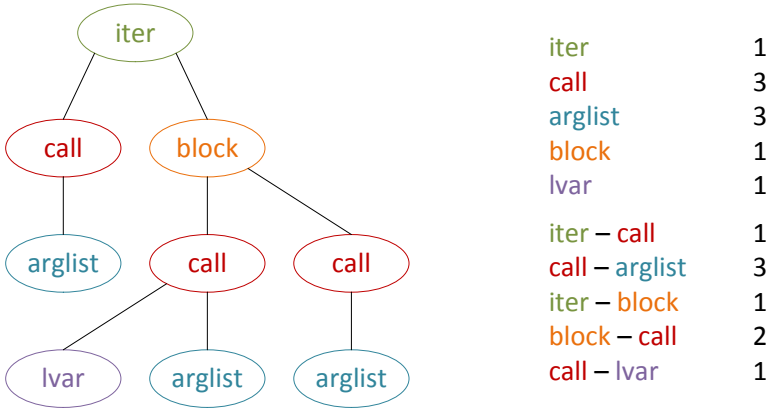
Obr. 5.2: Ukážka podstromov vytvorených algoritmom navrhnutým na rozdeľovanie abstraktných syntaktických stromov na menšie časti. Maximálna hĺbka podstromov je 3.



Obr. 5.3: Ukážka podstromov, ktoré algoritmus na rozdeľovanie abstraktných syntaktických stromov na menšie časti neakceptuje (vľavo) a ktoré spojí s podstromom súrodencov (vpravo). Minimálny počet vrcholov je 5.

Reprezentácia stromu. Ako už bolo spomínané vyššie, potrebujeme zvoliť takú reprezentáciu abstraktných syntaktických stromov, ktorá zachová informáciu o ich štruktúre a umožní ich rýchle porovnávanie. Plánujeme používať metódu Exas, ktorá je opísaná v kapitole 4.2.1 a navrhli ju Nguyen et al. [21, 20]. Jedným z dôvodov jej použitia je aj fakt, že umožní odhaľovať nielen presné duplikáty, ale dokonca aj čiastočne modifikované.

Exas v podobe, ktorá je prezentovaná v [20], zostaví pre vstupný strom jeho charakteristický vektor z počtu tzv. (p, q)-vrcholov a n-ciest. Pre hlboké stromy však bude počet možných n-ciest veľmi veľký (exponenciálne rastie so zvyšujúcim sa n). Preto navrhujeme zaviesť horné ohraničenie pre dĺžku n-ciest, ktoré sa pri zostavovaní charakteristického vektora použijú (t.j. horné



Obr. 5.4: Ukážka charakteristického vektora, ktorý bol vytvorený s použitím zjednodušenej metódy Exas (t.j. vynechané (p, q)-vrcholy a odmedzená dĺžka n-ciest na 2).

ohraničenie hodnoty n). Ďalej navrhujeme pri zostavovaní charakteristického vektora úplne vynechať (p, q)-vrcholy. Ich počet bude v porovnaní s počtom n-ciest veľmi nízky a domnievame sa, že pri následnom porovnávaní charakteristických vektorov by nemali veľkú váhu. Ich vynechanie však výrazne zjednoduší implementáciu. Žiadne experimenty na potvrdenie správnosti tohto rozhodnutia sme však nevykonali. Príklad abstraktného syntaktického stromu a zodpovedajúceho charakteristického vektora sa nachádza na obrázku 5.4.

Takto získané charakteristické vektory môžeme relatívne jednoducho porovnávať. Stále budú obsahovať až tisícky prvkov. Aj v prípade, ak použijeme iba 1 a 2-cesty. Väčšina z nich bude nulová, alebo nebudú mať pri porovnávaní stromov veľký význam. S cieľom ešte viac urýchliť porovnávanie podstromov preto navrhujeme redukovať dimenziu charakteristických vektorov.

Reduckia dimenzie. Prehľad viacerých existujúcich metód redukcie dimenzie vektorov uvádza napr. Fodor [8]. Ak hovoríme o redukcii vektorov s tisíckami dimenzií na desiatky alebo stovky dimenzií, aj najjednoduchšie metódy (napr. PCA) sú výpočtovo príliš náročné. Na riešenie tohto problému sú omnoho vhodnejšie náhodné projekcie, ktoré sú nielen rýchlejšie, ale dosahujú aj takmer rovnako kvalitné výsledky [8]. V [21] autori na vytváranie skupín zduplikovaných fragmentov kódu používajú lokálne senzitívne hašovanie [5]. To je algoritmus, ktorý sa obvykle používa na vyhľadávanie k najbližších susedov v dátach s veľkým počtom dimenzií, pričom využíva práve redukciu dimenzie pomocou náhodných projekcií. Jeho princíp opisujeme v prílohe A.1.

Nguyen et al. [21] používajú lokálne senzitívne hašovanie spôsobom uvedeným v nasledujúcom texte. Výsledkom samotnej operácie hašovania je jedno reálne číslo pre každý vstupný charakteristický vektor. Po zahašovaní všetkých vektorov vzniknú body, ktoré sú rozložené na priamke (t.j. v 1-rozmernom priestore). Priamku autori následne rozdelia na intervaly konštantnej dĺžky a tieto intervaly tvoria skupiny zduplikovaného kódu. Takýto prístup však podľa nášho názoru nie je vhodný, pretože sa môže veľmi ľahko stať, že:

- hranica intervalu bude uprostred niektorej skupiny zduplikovaných kódov a táto skupina bude následne nesprávne rozdelená na dve menšie,
- metóda označí fragmenty ako zduplikované aj vtedy, ak interval bude obsahovať jeden fragment na svojom začiatku a druhý na konci, pričom jeho vnútro bude úplne prázdne.

Uvedené nedostatky navrhujeme odstrániť nasledujúcim spôsobom. Ako už bolo spomínané, po zahašovaní charakteristického vektora dostaneme reálne číslo, čím sme vlastne zredukovali dimenziu tohto vektora na 1. V prípade, že lokálne senzitívne hašovanie zopakujeme k-krát, ako je to opísané v prílohe A.1, dostaneme pre jeden charakteristický vektor k reálnych čísel. Inými

slovami, vznikne nový vektor s k dimenziami. Ak je počet dimenzií tohto vektora menší ako počet dimenzií pôvodného charakteristického vektora, tak sme vykonali redukciu dimenzie.

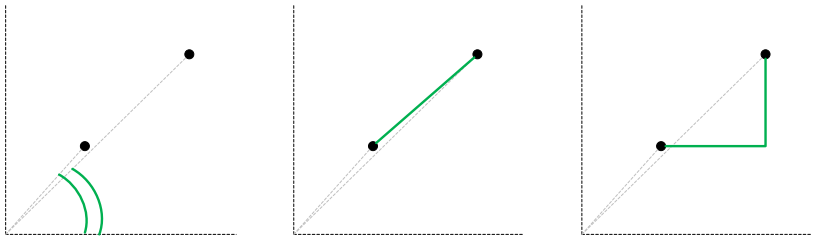
Takáto redukcia má nasledujúce charakteristiky, ktoré vyplývajú z princípu, na ktorom je lokálne senzitívne hašovanie založené (viď. príloha A.1):

- charakteristické vektory, ktoré sú si podobné, budú mať určite vždy a bez výnimky podobné aj príslušné zredukované vektory,
- pre rozdielne charakteristické vektory sa môžu niekedy vytvoriť podobné zredukované vektory (viď. obrázok A.1 vpravo), avšak pravdepodobnosť takejto situácie sa znižuje so zvyšujúcim sa k, t.j. so zvyšujúcim sa počtom vykonaných hašovaní (viď. obrázok A.2).

5.2.2 Vyhodnocovanie podobnosti kódov

Ak už máme pre fragmenty zdrojových kódov vytvorené charakteristické vektory, treba odmerať ich podobnosť. Existuje viacero metód akými sa dajú vektory vo všeobecnosti porovnávať. My sme zvažovali (1) kosínusovú podobnosť, (2) euklidovská vzdialenosť a (3) manhattanská vzdialenosť. Jednotlivé metódy sú ilustrované na obrázku 5.5.

Kosínusová podobnosť. Kosínusová podobnosť porovnáva uhly, pod akými vektory vychádzajú zo stredu súradnicovej sústavy. V našom prípade však túto metódu nie je vhodné použiť. Charakteristické vektory zostavujeme tak, že spočítame koľkokrát sa v abstraktnom syntaktickom strome kódu vyskytujú jeho jednotlivé vrcholy. Môže sa však stať, že jeden kód obsahuje z každého typu vrcholu približne n-násobne (napr. dvojnásobne) viac kusov ako iný. Kosínusová podobnosť takýchto kódov bude takmer rovnaká, napriek tomu, že už aj intuitívne sú porovnávané kódy odlišné (jeden je väčší ako druhý).



Obr. 5.5: Metódy porovnávania vektorov, zľava doprava sú to: (1) kosínusová podobnosť, (2) euklidovská vzdialenosť a (3) manhattanská vzdialenosť.

Euklidovská a manhattanská vzdialenosť. Tieto metódy považujeme za vhodné na určovanie podobnosti zdrojových kódov. Vyhovujú nám aj z hľadiska kombinácie s lokálne senzitívnym hašovaním (viď. príloha A.1). Rozhodli sme sa používať manhattanskú vzdialenosť (podobne ako Nguyen et al. [21]), (1) na základe niekoľkých jednoduchých experimentov a (2) preto, lebo metóda je jednoduchšia a jej výpočet je rýchlejší.

5.2.3 Index zduplikovaných kódov

Fragmenty zdrojových kódov už dokážeme zjednodušiť na vektor a príslušné vektory vieme navzájom porovnať. Zostáva už len odhaliť skupiny zduplikovaných kódov. Porovnávanie každého vektora so všetkými ostatnými by stále bolo výpočtovo veľmi náročné a zbytočne pomalé. Potrebujeme teda vybrať lepší spôsob, ktorý umožní nájsť zduplikované skupiny kódov.

Predtým, ako sa pustíme do vyberania konkrétnych algoritmov alebo dátových štruktúr, zosumarizujme si vlastnosti dát, s ktorými treba pracovať:

1. väčšina zdrojových kódov nebude zduplikovaná, ich vektory budú v priestore rovnomerne rozložené,
2. dáta budú obsahovať len malý počet skupín s duplikátmi,
3. každá skupina bude obsahovať len niekoľko zduplikovaných fragmentov,
4. nevieme dopredu povedať koľko skupín sa bude v dátach nachádzať, nemusia tam byť žiadne a zároveň ich tam môže byť veľmi veľa,
5. vieme definovať maximálnu možnú veľkosť jednej skupiny s duplikátmi.

Problém rozdeľovania dát do skupín na základe ich podobnosti sa dá riešiť klastrovaním. Niektoré požiadavky na klastrovací algoritmus vyplývajú z vyššie uvedených vlastností dát. Medzi ďalšie požiadavky patria:

1. možnosť rýchlo zistiť, do ktorého klastra patrí nejaký fragment kódu, aby sme vedeli obratom povedať, či je alebo nie je kód zduplikovaný,
2. možnosť priebežne pridávať alebo odstraňovať body z klastrov, čo je dôležité, ak chceme zduplikovaný kód odhaľovať inkrementálne a nechceme zakaždým vykonávať celé klastrovanie od začiatku.

Komplexný prehľad rôznych algoritmov a klastrovacích metód uvádza napr. Berkhin [2]. Existuje celá skupina hierarchických klastrovaní, ktoré vytvárajú strom klastrov a umožňovali by tak vyhľadávanie v logaritmickom čase. Takmer žiadna z nich však nespĺňa našu podmienku inkrementálnosti. Takisto dopredu nevieme ani len približne odhadnúť, koľko má byť v dátach klastrov.

Naše požiadavky spĺňa algoritmus *Birch*, ktorý je detailne opísaný v prílohe A.2. Medzi hlavné dôvody, kvôli ktorým sme zvolili práve túto metódu klastrovania, patria:

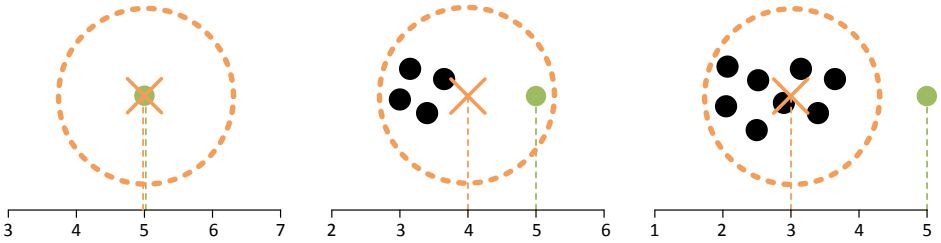
- patrí do skupiny hierarchických klastrovaní, čo znamená, že klastre interne udržiava v stromovej štruktúre, ktorá umožní vyhľadávanie, vkladanie a mazanie v logaritmickom čase,
- metóda bola navrhnutá so zámerom byť inkrementálna, čo znamená, že iniciálne klastrovanie prebieha úplne rovnako ako následné aktualizácie,
- stačí len jeden prechod dátami, čo znamená, že každý fragment kódu stačí spracovať presne raz,
- nemusí si pamätať vložené dáta, udržiava len informácie o klastroch, čo znamená, že pamäťové nároky budú veľmi malé,
- klastrovanie môže vykonávať na základe euklidovskej alebo manhattanskej vzdialenosti, čo je v našom prípade dôležité (viď. kapitola 5.2.2).

Pre úplnosť uvedieme, že za index zduplikovaných kódov budeme považovať strom, ktorý si algoritmus *Birch* vytvára a udržiava.

Aktualizácia indexu. Pri aktualizácii indexu môžu nastať nasledujúce situácie (obrázok 5.1): (1) starý kód bol v novej verzii vymazaný a príslušné podstromy treba z indexu odstrániť, (2) v novej verzii bol pridaný nový kód, ktorý sa musí do indexu vložiť. Zmenený kód plánujeme spracovávať tak, že najprv vymažeme z indexu jeho starú verziu a následne vložíme novú. Tým však stratíme samotnú informáciu o tom, že kód bol zmenený (a nie je napr. čerstvo vytvorený). Táto stratená informácia by mohla pomôcť presnejšie sledovať evolúciu duplikátov (napr. ak sa v jednej inštancii duplikátu opraví chyba, v ostatných ju treba opraviť tiež). Jej využitie by mohlo byť súčasťou ďalšej práce na projekte.

Posúvanie centroidov klastrov. Dôsledkom inkrementálneho klastrovania je, že jednotlivé klastre sa neustále v čase menia. Pridávajú sa a odoberajú sa z nich body. To znamená, že centroidy klastrov sa postupne pohybujú. Výhodou takéhoto pohybu je, že sa vedia dynamicky prispôsobiť klastrovaným dátam. Pohyb klastrov má však aj dva súvisiace nežiadúce dôsledky.

Prvý z dôsledkov sa prejavuje v listoch *Birch* stromu. Listy majú obmedzenú veľkosť (ako je uvedené v prílohe A.2). Tá v našom prípade definuje, ako veľmi môžu byť dva fragmenty kódu odlišné, aby sme ich stále považovali za duplikáty. V dôsledku posúvania klastrov však môže nastať situácia, ktorá

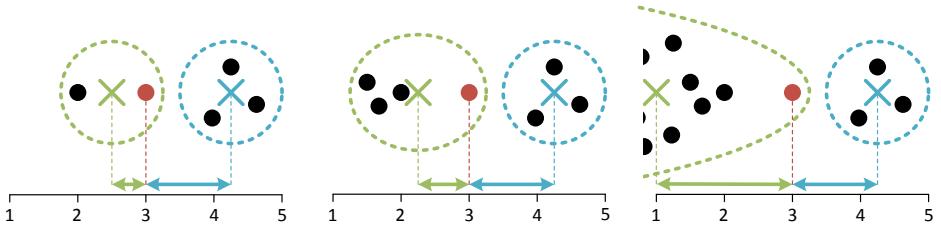


Obr. 5.6: Klaster, resp. list *Birch* stromu sa postupne posúva doľava, pretože body sú do neho vkladané v nevhodnom poradí. Zelený bod by po vložení vštkých bodov do tohto klastra nemal patriť.

je znázornená na obrázku 5.6. Body sú do listov vkladané v špecifickom, nevhodnom poradí. To spôsobí, že klaster (resp. jeho centroid) sa stále posúva doľava. Nakoniec sa stane to, že bod, ktorý sa v liste síce stále fyzicky nachádza, má už od stredu klastra väčšiu vzdialenosť, ako je maximálne povolená. Pri inom vstupnom poradí dát by bol vložený do úplne odlišného klastra, čo by bolo správne riešenie.

Druhý dôsledok je spôsobený rovnakým problémom, no týka sa nielen listov, ale aj vnútorných vrcholov. Tie nemajú definovanú maximálnu veľkosť, čiže body z nich nemôžu „vypadnúť". Centroidy klastrov, ktorým zodpovedajú vnútorné vrcholy, sa však stále posúvajú a môžu sa k sebe približovať alebo sa od seba vzdaľovať. Na obrázku 5.7 je takáto situácia ilustrovaná.

Obidva opísané problémy vznikajú len pri naozaj nevhodnom vstupnom poradí bodov. Domnievame sa, že reálne k nim bude dochádzať len zriedkavo (viď. kapitola 6.3.7). Aktualizácie indexu navyše navrhujeme (z dôvodu jednoduchosti implementácie) vykonávať tak, že starú verziu zmeneného súboru z indexu najprv odstránime a novú vložíme. To spôsobí, že skôr či neskôr sa



Obr. 5.7: Stred vnútorného vrcholu *Birch* stromu sa postupne posúva doľava, pretože body sú do neho vkladané v nevhodnom poradí. Červený bod po takejto postupnosti operácií nevieme nájsť, pretože je bližšie ku modrému vrcholu (resp. podstromu), no reálne sa nachádza v zelenom vrchole.

tieto zle zaklastrované body z indexu vyberú a následne opäť vložia, tentokrát už pravdepodobne na správne miesto. Vplyv posúvania centroidov na výsledky detekcie duplikátov bude zrejme len minimálny a môžeme ho zanedbať.

Problém však nastáva práve pri mazaní existujúceho kódu z indexu. Keďže vnútorné vrcholy aj listy sa posúvajú, môžu sa posunúť až tak, že nejaký fragment kódu budeme hľadať v úplne inom klastri, alebo úplne inej časti stromu, ako je tá, do ktorej bol pôvodne vložený. Príklad sa nachádza na obrázku 5.7. Červený bod bol vložený do ľavého klastra, alebo podstromu. Jeho centroid sa následne posunul. Pri vyhľadávaní bodu (napr. práve s cieľom vymazať ho) je však už bližšie centroid pravého klastra a hľadaný fragment kódu teda nenájdeme. Na riešenie týchto problémov navrhujeme:

1. listy (ktoré majú obmedzenú veľkosť) po vložení nového fragmentu kódu skontrolujú, či do nich stále patria aj všetky už predtým vložené prvky; ak sa nájde taký, ktorý do listu nepatrí, list ho zo seba vylúči; vylúčené fragmenty sa dostanú k rodičovi listu (t.j. prvému vnútornému vrcholu) a ten ich vloží do vhodnejšieho listu (alebo vytvorí nový list),

2. na vyhľadávanie a mazanie kódu z indexu si vytvoríme hašovaciu tabuľku, ktorá bude mapovať fragmenty kódu (názov súboru a čísla riadkov) na listy *Birch* stromu, do ktorých boli fragmenty vložené; vďaka tomu nájdeme ľubovoľný konkrétny fragment kódu v konštantnom čase a jeho vymazanie zo stromu sa bude stále vykonávať v logaritmickom čase; potrebujeme si však navyše pamätať celú hašovaciu tabuľku.

Perzistencia indexu. Vytvorený index je potrebné nejakým spôsobom medzi verziami uchovávať. Predpokladáme, že bude mať aj pri veľkom množstve zaindexovaných kódov dostatočne malú veľkosť na to, aby sa ho oplatilo celý načítať do pamäte. Mal by byť čo najmenší aj uložený na disku (aby ho každý vývojár mohol mať vo vlastnom počítači). Navrhli sme preto vlastný, kompaktný binárny formát, ktorého popis sa nachádza v prílohe B.3. Reálne veľkosti indexu, v závislosti od množstva kódu, sú uvedené v kapitole 6.4.1.

Hašovaciu tabuľku, ktorú používame na vyhľadávanie konkrétnych fragmentov kódu v indexe, serializovať do súboru netreba. Pri načítavaní serializovaného indexu sa dá vytvoriť dynamicky veľmi jednoducho a rýchlo.

5.2.4 Detekcia zduplikovaného kódu.

Detekcia duplikátov je automaticky vykonávaná už počas aktualizácie indexu, pretože každý fragment je vložený do klastra, v ktorom sa nachádzajú podobné kódy. Takto vytvorené skupiny duplikátov ešte nie sú finálne, pretože:

1. môžu obsahovať fragmenty, ktoré nie sú v skutočnosti zduplikované (angl. *false positives*),

2. fragmenty sa môžu navzájom prekrývať, preto ich treba vo finálnych výsledkoch rozumne spojiť dokopy.

V nasledujúcom texte uvádzame návrh riešení uvedených problémov.

Potvrdzovanie duplikátov. Indexovanie sme sa snažili navrhnúť tak, aby informácia o štruktúre zdrojových kódov zostala zachovaná. Mnohé iné informácie o fragmentoch kódu však strácame – počas vytvárania charakteristických vektorov, alebo neskôr, pri redukcii dimenzie. Následkom toho je, že naša metóda môže v niektorých prípadoch označiť kódy ako duplikáty, aj keď v skutočnosti podobné nie sú.

Vďaka experimentom opísaným v kapitole 6 však vieme, že k takýmto situáciám nedochádza veľmi často. Väčšina výsledkov sú skutočné duplikáty. Predpokladáme, že ďalšie kontrolovanie výsledkov by neprinieslo výrazné zlepšenie, a preto sme sa ním ďalej nezaoberali. Je však súčasťou ďalšej možnej práce na projekte.

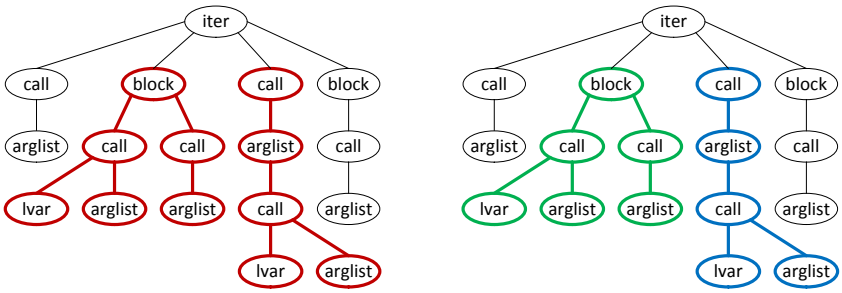
Skupiny duplikátov vo všeobecnosti obsahujú malý počet fragmentov zdrojových kódov. To umožňuje všetky tieto fragmenty navzájom dôkladne porovnať, každý s každým. Napríklad aj nejakou pomalšou metódou. Rovnako problém riešia aj Nguyen et al. [21].

Výsledky však môžu obsahovať aj také kódy, ktoré sú síce technicky zduplikované, ale nie je ich vhodné alebo možné refaktorovať (napr. z dôvodu limitácií programovacieho jazyka). Aj takéto prípady by sa dali z finálnych výsledkov vyfiltrovať. Bolo by však potrebné analyzovať význam jednotlivých fragmentov kódu. K tomuto problému sme sa v práci nedostali a je tiež súčasťou ďalšej možnej práce na projekte.

Spájanie prekrývajúcich sa výsledkov. Do indexu zdrojových kódov sa pri spracovaní vkladajú fragmenty, ktoré sa navzájom prekrývajú, alebo sú dokonca úplnou súčasťou jeden druhého. To znamená, že vo výsledkoch sa potom môže rovnaký duplikát nachádzať aj viackrát. Konkrétne môžu nastať 3 nasledujúce situácie:

- v jednom klastri sa nachádza viacero prekrývajúcich sa fragmentov rovnakého kódu a žiaden iný kód – takéto fragmenty nemôžeme považovať za duplikáty (je to v skutočnosti len 1 fragment) a z výsledkov treba takéto klastre vynechať,

- v jednom klastri sa nachádza viacero prekrývajúcich sa fragmentov rovnakého kódu a zároveň ďalšie, úplne iné (ale podobné) fragmenty – klas-



Obr. 5.8: Problém s horizontálnym prekrývaním podstromov. Vľavo je znázornená časť stromu, ktorá je zduplikovaná. V pravej časti sú zvýraznené dva podstromy, ktoré budú našou metódou označené ako dva rôzne duplikáty.

ter obsahuje zduplikovaný kód z viacerých miest projektu, čiže ho do finálnych výsledkov určite treba zaradiť; prekrývajúce sa fragmenty kódu navrhujeme a plánujeme v tomto prípade spojiť do jedného,

- rôzne fragmenty z toho istého kódu sa nachádzajú vo viacerých rôznych klastroch; inými slovami povedané, rovnaké riadky kódu sa nachádzajú vo viacerých odlišných skupinách duplikátov – navrhujeme takýto kód vypísať vo výsledkoch viackrát a príslušné skupiny a duplikáty nespájať dokopy, pretože:
  - bez ďalšieho spracovania výsledkov nevieme povedať, či by sme náhodou nespojili také skupiny zduplikovaného kódu, ktoré sú odlišné a mali by byť uvedené samostatne (ako je ilustrované napríklad na obrázku 6.3),
  - ak chceme označovať duplikáty priamo vo vývojovom prostredí, nezáleží na tom, v koľkých rôznych skupinách sa kód nachádzal; zaujíma nás len, ktoré iné kódy sú mu podobné.

Spájanie susediacich duplikátov. Náš algoritmus vo svojej aktuálnej podobe nepodporuje spájanie susediacich duplikátov. Takáto situácia je znázornená na obrázku 5.8. Na ľavej strane je červenou farbou vyznačená časť stromu, ktorá je zduplikovaná. Vzhľadom na spôsob, ktorým aktuálne vytvárame podstromy, naša metóda v takejto situácii odhalí dva menšie duplikáty namiesto jedného veľkého (znázornené na pravej strane obrázka 5.8, každý z duplikátov je vyznačený inou farbou).

Opäť by sa dal problém vyriešiť napríklad záverečným spracovaním. Takéto spájanie by však nebolo úplne triviálne. Čo ak sú modrý aj zelený fragment z obrázka 5.8 na jednom mieste síce zduplikované spolu, ale na inom

mieste je odkopírovaný len zelený? Myslíme si, že v takom prípade je lepšie susediace duplikáty nespájať. Navyše, napríklad pre potreby vyznačovania zduplikovaných riadkov vo vývojovom prostredí nás nezaujíma, či sa nachádzali v jednej alebo viacerých skupinách. Problémom sme sa ďalej nezaoberali.

5.3 Vlastná definícia zduplikovaného kódu

Chatterji et al. [3] vo svojej práci dospeli k názoru, že zatiaľ neexistuje všeobecne akceptovaná definícia zduplikovaného kódu a jeho typov. Autori jednotlivých metód a existujúcich nástrojov používajú vlastné definície, aj keď ich často v prácach explicitne nespomínajú. Všeobecne akceptovaná definícia je určite potrebná minimálne z dôvodu objektívneho porovnávania jednotlivých metód odhaľovania duplikátov. Pre návrh a implementáciu týchto metód však nie je nevyhnutná a prax ukazuje, že kvalitné detektory duplikátov sa dajú vytvoriť aj bez nej.

Aby bolo jasné, aké typy zduplikovaného kódu odhalí nami navrhnutá metóda, uvádzame v nasledujúcom texte vlastnú definíciu zduplikovaného kódu. Odvodíme ju z návrhu opísaného v kapitole 5.2.

Aký veľký bude duplikát? V prvom rade si treba pripomenúť, že navrhnutá metóda využíva abstraktné syntaktické stromy. Tie, okrem iného, abstrahujú od textového zápisu zdrojového kódu. To znamená, že veľkosti zduplikovaných fragmentov nevieme definovať v riadkoch zdrojového kódu. Vo všeobecnosti platí, že:

- jednoduché výrazy, napríklad samostatné volanie metódy, sú veľmi malé na to, aby sa dali považovať za duplikáty; to aj v prípade, že sú napísané vo viacerých riadkoch, napríklad ak je každý parameter metódy uvedený na samostatnom riadku,
- na druhej strane, komplikované výrazy (napríklad podmienky, cykly, atď.) bývajú dostatočne veľké na to, aby mohli byť označené ako duplikáty; to aj v prípade, ak sa nachádzajú len na 1 riadku kódu.

Samotné veľkosti možných duplikátov sú ovplyvnené tým, ako sa abstraktné syntaktické stromy delia na menšie podstromy (kapitola 5.2.1). Tento algoritmus nám hovorí, že podstromy (a teda fragmenty kódu) majú definovanú:

- minimálnu veľkosť – minimálny počet vrcholov podstromu,
- čiastočne aj maximálnu veľkosť – podstromy môžu byť ľubovoľne široké (vo všeobecnosti takto môžu siahať aj cez desiatky riadkov kódu) ale majú obmedzenú hĺbku.

Podstromy majú obmedzenú hĺbku, čoho dôsledkom je, že ako zduplikovaný kód môžu byť označené také fragmenty, ktoré majú podobné vyššie úrovne (napr. špecifická postupnosť cyklov a podmienok) a odlišné telá.

Kedy sú 2 fragmenty kódu považované za zduplikované? Aby sme zodpovedali túto otázku, musíme sa pozrieť na spôsob, akým sa jednotlivé fragmenty porovnávajú.

Podobnosť fragmentov sa počíta ako (manhattanská) vzdialenosť ich charakteristických vektorov. Fragmenty sú označené ako duplikáty ak je ich podobnosť menšia, ako pevne stanovená hodnota. To znamená, že aj čiastočne rozdielne fragmenty kódu môžu byť označené ako duplikáty. Vôbec pri tom nezáleží na tom, v čom konkrétne sa dané fragmenty líšia.

Na vzdialenosť charakteristických vektorov má najväčší vplyv spôsob, akým sa tieto vektory vytvárajú. Tu je dôležité poznamenať, že charakteristické vektory sa vytvárajú len z typov vrcholov. Všetky názvy identifikátorov alebo hodnoty konštánt sa ignorujú. Dva fragmenty kódu, ktoré sa líšia len v konštantách alebo názvoch, budú označené ako úplne totožné.

5.4 Sledovanie evolúcie duplikátov v čase

Hlavnou výhodou metódy odhaľovania duplikátov, ktorú sme navrhli, je inkrementálnosť. Doteraz sme sa nestretli s iným nástrojom, ktorý by duplikáty v zdrojových kódoch dokázal odhaľovať inkrementálne, aj napriek tomu, že niekoľko takýchto metód už bolo navrhnutých [11, 21, 4]. Práve túto inkrementálnosť by sme chceli pri používaní nástroja využiť.

Okrem samotnej informácie o zduplikovaných častiach kódov, nám inkrementálnosť *implicitne* poskytuje aj informácie o tom, ako sa duplikáty menia v čase. Z pohľadu vývojára sú dôležité najmä nasledujúce udalosti:

- *vytvorenie nového duplikátu* – nastane vtedy, ak sa do klastra, ktorý doteraz obsahoval len jeden fragment kódu (alebo viacero prekrývajúcich sa fragmentov) pridá ďalší, nový fragment (ktorý sa neprekrýva s už existujúcimi); vývojár by mal byť upozornený, že vytvoril nový duplikát,

- *odstránenie alebo modifikácia jednej inštancie duplikátu* – nastane, ak sa z klastra, ktorý obsahuje viacero rôznych fragmentov, nejaký fragment odstráni alebo zmení (v súčasnej implementácii je zmena fragmentu len postupné odstránenie starého a následné pridanie nového fragmentu); obidve operácie implikujú, že vývojár zmenil zduplikovaný kód na jednom mieste a pravdepodobne by mal zmeniť aj jeho ostatné výskyty,

- *odstránenie duplikátu* – ak v klastri ostane iba jeden fragment kódu (alebo viacero takých, ktoré sa prekrývajú), znamená to, že vývojár danú skupinu duplikátov refaktoroval a tým odstránil.

Jednorazové metódy odhaľovania duplikátov nemajú ako implicitne poskytnúť takéto informácie. Jedinou možnosťou by bolo ukladať si odhalené duplikáty z každej verzie (alebo aspoň z posledných 2 verzií) a použiť nejaké dodatočné spracovanie, ktoré by výstupy porovnalo a tak zistilo, či duplikáty pribudli alebo odbudli. Takéto spracovanie by určite zabralo nenulový čas a aj samotné jednorazové vyhľadávanie duplikátov zaberá viac času. Hrozí teda, že informácie o duplikátoch a ich evolúcii sa dostanú k vývojárovi s oneskorením, napríklad až keď sa venuje inému problému.

Ako sme už uviedli, naša metóda dokáže informácie o evolúcii duplikátov poskytovať implicitne. Nie je na to potrebné žiadne ďalšie spracovanie. Vývojár teda môže dostať informácie o zmenách v duplikátoch v reálnom čase – t.j. vtedy, keď ich najviac potrebuje a môže ich naozaj využiť.

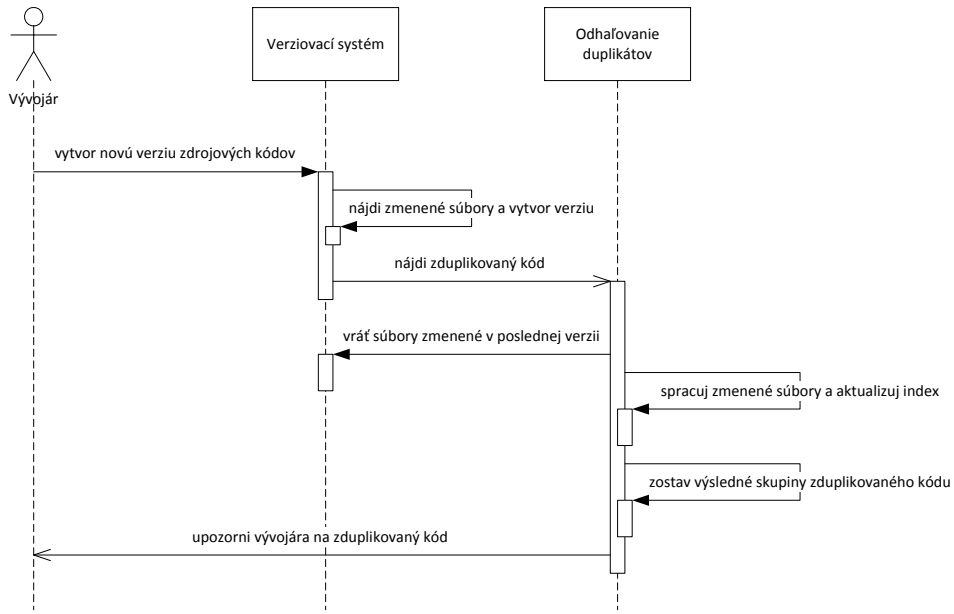
Podporu pre získavanie takýchto informácií sme zatiaľ neimplementovali. Najmä z dôvodu, že pri experimentoch (kapitola 6) sme ju nepotrebovali. Implementácia je však triviálna a vrámci budúcej práce na projekte s ňou určite počítame.

5.4.1 Integrácia so systémom na správu verzií

Na správu zdrojových kódov softvéru sa dnes už bežne používajú verziovacie systémy. Náš nástroj na odhaľovanie duplikátov je možné prepojiť s niektorým takýto systémom. Návrh integrácie si predstavujeme nasledovne (obrázok 5.9):

1. vývojár vloží novú verziu kódov do verziovacieho systému,
2. verziovací systém fyzicky vytvorí novú verziu, po skončení automaticky spustí nástroj na detekciu duplikátov,
3. nástroj na detekciu duplikátov z verziovacieho systému zistí, ktoré súbory boli v poslednej verzii pridané, odstránené alebo zmenené a podľa toho aktualizuje index duplikátov,
4. nástroj na detekciu duplikátov následne vytvorí pre používateľa vyhodnotenie, v ktorom budú vypísané všetky duplikáty odhalené v najnovšej verzii zdrojových kódov, resp. popis zmien, ktoré medzi verziami v duplikátoch nastali.

Ani implementácii tejto časti sme sa doteraz nevenovali. Dôvodom bolo to, že pre potreby overenia navrhnutej metódy sme ju nepotrebovali.



Obr. 5.9: Sekvenčný diagram znázorňujúci integráciu medzi systémom na verziovanie zdrojových kódov a našim nástrojom na odhaľovanie duplikátov.

5.4.2 Integrácia s vývojovým prostredím

Ďalšou možnosťou, ako navrhnutý nástroj reálne využiť, je pridať ho priamo do niektorého z vývojových prostredí. Vývojové prostredie by zduplikované časti kódov mohlo zvýrazňovať priamo vo svojom editore a tým neustále a v reálnom čase upozorňovať vývojárov na zduplikovaný kód. Takisto by vývojárov mohlo v reálnom čase upozorňovať na udalosti súvisiace s evolúciou duplikátov v projekte. Podľa nášho názoru by práve takáto integrácia bola pre vývojárov najväčším prínosom. Jej implementácii sme sa však v práci tiež nevenovali.

6

Overenie riešenia

V práci navrhujeme a implementujeme nástroj, ktorý vykonáva inkrementálnu detekciu duplikátov. Odborníci ani odborná verejnosť sa však nevedia zhodnúť na tom, čo by malo byť považované za zduplikovaný kód a čo nie [31, 3]. Overovanie a vzájomné porovnávanie nástrojov na detekciu duplikátov je kvôli tomu veľmi ťažké a subjektívne. My sme ho vykonávali v troch etapách:

- v prvej sme sa snažili dokázať, že navrhnutá metóda naozaj dokáže nájsť zduplikovaný kód a že bude schopná konkurovať existujúcim nástrojom,
- v druhej etape experimentov sme zisťovali, ako rôzne hodnoty vstupných parametrov ovplyvňujú dosahované výsledky a aká kombinácia parametrov je najviac vhodná pre praktické použitie,
- na záver sme sa snažili dokázať, že pri postupnom (a dlhodobom) inkrementálnom spracovávaní (1) presnosť odhaľovania duplikátov neklesá a (2) že sa odhalia také isté duplikáty, ako pri jednorazovom spracovaní.

Zvyšok kapitoly obsahuje podrobný opis vykonaných experimentov, analýzu ich výsledkov a diskusiu o ich dôsledkoch.

6.1 Návrh overenia

Identifikovali sme niekoľko otázok, ktoré by sme mali byť po vykonaní overenia schopní zodpovedať:

1. Je nami navrhnutá metóda vhodná pre odhaľovanie duplikátov?
2. Aké vstupné parametre dosahujú najlepšie výsledky?
3. Ako dobre funguje nami navrhnutá metóda ako celok, napríklad na vzorových dátach, alebo v porovnaní s inými nástrojmi?
4. Odhalíme pri inkrementálnej detekcii presne tie isté skupiny duplikátov ako pri nezávislom spracovaní každej verzie? Nezníži sa pri použití inkrementálnej detekcie presnosť identifikácie duplikátov?

6.2 Porovnanie s existujúcimi nástrojmi

Walenstein et al. [31] sa zaoberali problémami, ktoré sú spojené s vytváraním vzoriek zduplikovaného kódu. Upozorňujú, že aj ľudia mávajú často odlišné názory na to, či je nejaký kód duplikát alebo nie. V jednom konkrétnom prípade nechali autori troch odborníkov nezávisle na sebe klasifikovať 317 fragmentov kódu a len 5 z nich všetci traja klasifikovali rovnako.

Walenstein et al. [31] ďalej tvrdia, že to, či je kód duplikát alebo nie, býva závislé aj (1) od použitých technológií alebo (2) od cieľa, kvôli ktorému sa duplikáty vyhľadávajú (napr. refaktoring a jednoduchšia údržba kódu, alebo zmenšovanie veľkosti kódu, atď.). Zdôrazňujú, že referenčné vzorky dát by tieto ciele mali zohľadňovať. Na záver vyjadrujú pochybnosti o tom, či je vôbec možné vytvoriť niektoré takéto referenčné vzorky duplikátov.

Nám sa žiadne vzorky zduplikovaného kódu nájsť neporadilo. To znamená, že nevieme úplne objektívne porovnať našu metódu voči ostatným existujúcim. Aby sme však mohli vykonať aspoň čiastočné overenie, pre potreby experimentov sme pripravili vlastnú vzorku duplikátov.

Pozostávala z 9 súborov, ktoré spolu obsahovali približne 730 riadkov zdrojového kódu (vrátane prázdnych riadkov a komentárov). Všetky súbory obsahovali tzv. *unit testy*. Tie sme zvolili preto, lebo práve testovací kód často obsahuje zduplikované časti, napríklad úplne rovnaké alebo podobné tzv. *test cases*. Niektoré z duplikátov (1) sa líšia len v použitých konštantách, iné (2) majú rozdielne podmienky vykonávania alebo (3) kontrolujú odlišné časti výsledku, atď. Inými slovami, snažili sme sa vybrať také súbory, ktoré obsahujú zduplikovaný kód rôznych typov a veľkostí.

Použitá vzorka súborov sa nachádza na DVD v prílohe E v adresári `datasets/manual_experiments/wanted_tests`. Nástroje, s ktorými sme počas experimentov našu metódu porovnávali, boli *Simian*, *Flay* a *RubyMine*. Všetky boli predstavené v kapitole 4.1.

6.2.1 Manuálne vyhodnotenie

Priebeh experimentov. Manuálne experimenty spočívali v spustení každého z nástrojov nad testovacou vzorkou zdrojových kódov. Pre spustenie *Simian*-u sme použili príkaz:

```
> simian-2.3.33.exe -defaultLanguage=ruby -balanceCurlyBraces
  -balanceParentheses -balanceSquareBrackets -ignoreCharacterCase
  -ignoreCharacters -ignoreCurlyBraces -ignoreIdentifierCase
  -ignoreIdentifiers -ignoreLiterals -ignoreModifiers
  -ignoreNumbers -ignoreStringCase -ignoreStrings
  -ignoreSubtypeNames -ignoreVariableNames -threshold=2
  samples\wanted_tests\*.rb
```

Flay sme spustili takýmto príkazom:

```
> flay -m 1 samples\wanted_tests\*.rb
```

RubyMine sme používali priamo v IDE s nasledujúcimi parametrami: anonymizácia lokálnych premenných, atribútov, metód a konštánt: áno, neukazovať duplikáty jednoduchšie ako: 10 (menej sa nedalo nastaviť), anonymizuj neobvyklé podvýrazy jednoduchšie ako: 5. Náš nástroj sme na záver spustili s parametrami, ktoré by podľa experimentov v kapitole 6.3 mali dosahovať najlepšie výsledky. Snažili sme sa zakaždým vybrať také nastavenia, resp. vstupné parametre, ktoré umožnili odhaliť čo najviac duplikátov.

Výstupy každého z nástrojov sme priložili na DVD (príloha E), do `results/manual_experiments/raw_output`. RubyMine výstup zobrazuje len v používateľskom rozhraní a preto sme ho museli prepísať do textového súboru, ktorý sa nachádza v adresári `results/manual_experiments/pretty_output`. V tom istom adresári sa nachádzajú aj výstupy ostatných nástrojov v rovnakom formáte pripravené na spracovanie algoritmom.

Vyhodnotenie experimentov. Výsledky sme sa snažili vyhodnotiť ručne. Veľmi rýchlo sme sa však aj my presvedčili, že ide o komplikovanú a často nejednoznačnú úlohu [31]. Informácie a závery uvedené v nasledujúcom texte preto treba považovať za *subjektívne*.

Pri spracovávaní výsledkov sme vynechali nástroj *Simian*. Napriek tomu, že jeho výstup bol technicky správny, odhalené duplikáty sa nedali skoro vôbec porovnávať s výsledkami ostatných nástrojov. *Simian* je založený na porovnávaní textu (viď. kapitola 4.1) a neberie do úvahy syntax programovacieho jazyka. Následkom bolo napríklad to, že mnohé duplikáty začínali v jednej metóde a končili v inej, ako je ilustrované na obrázku 6.1.

Výsledky ostatných nástrojov sme ručne spojili, resp. zjednotili. Vzniklo 57 skupín zduplikovaného kódu, pričom niektoré z nich sa navzájom prekrývali. To znamená, že 1 fragment kódu, alebo jeho časť, sa mohli nachádzať aj vo viacerých skupinách s duplikátmi.

Až 30 skupín obsahovalo iba 1 riadok kódu. 23 z týchto 30 jednoriadkových skupín by sa však nedalo refaktorovať alebo to neboli v skutočnosti

```
1.    ...
2.    assert_true(results, "Should be correct")
3.  end
4.
5.  def test_not_cited_results
6.    chunk = chunks(:first)
7.    ...
```

```
1.    ...
2.    assert_false(results, "Should not be correct")
3.  end
4.
5.  def test_correctly_cited_results
6.    chunk = chunks(:second)
7.    ...
```

Obr. 6.1: Zduplikované fragmenty odhalené nástrojom *Simian*, ktoré začínajú v jednej metóde a končia v inej.

Tabuľka 6.1: Počty skupín s duplikátmi, v ktorých jednotlivé nástroje našli všetky alebo aspoň niektoré duplikáty.

	Nájdené všetky duplikáty zo skupiny			
	Celkom	Správne	Nerefaktorovateľné	Nesprávne
Flay	35	15	13	7
RubyMine	11	10	1	0
Náš nástroj	16	12	4	0
	Nájdené aspoň niektoré duplikáty zo skupiny			
	Celkom	Správne	Nerefaktorovateľné	Nesprávne
Flay	41	20	14	7
RubyMine	17	16	1	0
Náš nástroj	22	17	5	0

duplikáty. Iba 7 z nich sme považovali za naozajstné duplikáty. Veľkú väčšinu z týchto 30 jednoriadkových duplikátov našiel iba Flay a ostatné nástroje nie. Zo všetkých 57 skupín sme len 27 považovali za naozajstné duplikáty. 22 by sa podľa nášho názoru nedalo (alebo nemalo) refaktorovať a 8 skupín neobsahovalo zduplikovaný kód (tzv. false positives).

Zaujímavé je, že každý z nástrojov v približne jednej štvrtine prípadov neodhalil všetky výskyty niektorého duplikátu, ktoré sa v kóde reálne nachádzali a ktoré odhaliť mal. Detailné výsledky sa nachádzajú v tabuľke 6.1.

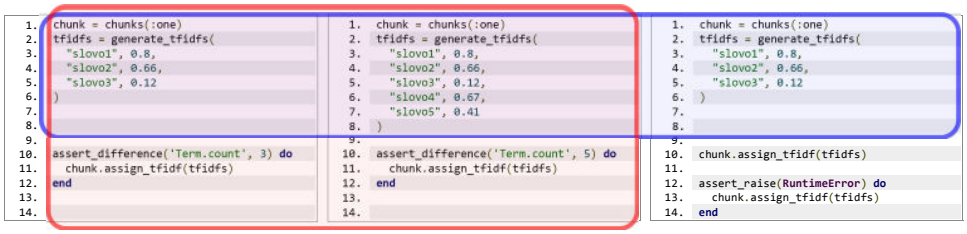
Podrobné informácie o jednotlivých duplikátoch, o tom do akých skupín sme ich spojili, odkazy na príslušné časti zduplikovaných kódov, typy duplikátov a mnohé ďalšie informácie sa nachádzajú na DVD v prílohe E, v súbore s názvom `results/manual_experiments/final_merged.xlsx`.

Problémy pri manuálnom vyhodnocovaní. Na záver tejto časti by sme ešte chceli na dvoch príkladoch ukázať, prečo je manuálne porovnávanie rôznych nástrojov na hľadanie zduplikovaného kódu náročné a nejednoznačné. Prvý príklad je znázornený na obrázku 6.2. Fragmenty sa líšia v názve identifikátora na riadku 4. V Ruby by bolo možné takýto kód refaktorovať, ale to určite neplatí pre všetky programovacie jazyky. *RubyMine* fragmenty zaradil do rozdielnych skupín (t.j. neoznačil ich ako duplikáty), *Flay* a náš nástroj

```
1.  a = analyses(:os_two)
2.  a.calculate_plagiarism_level()
3.
4.  assert_not_nil a.plagiarism_level
5.  ...
```

```
1.  a = analyses(:os_two)
2.  a.calculate_plagiarism_level()
3.
4.  assert_not_nil a.plagiarism_words
5.  ...
```

Obr. 6.2: Zduplikované fragmenty, ktoré sa líšia v názve identifikátora na riadku 4. Dali by sa refaktorovať?



Obr. 6.3: Tri fragmenty, ktoré majú zduplikované rôzne časti. Je lepšie ako duplikáty označiť len prvé dva, alebo radšej len vrchné časti všetkých troch?

ich vložili do rovnakej skupiny (t.j. označili ako duplikáty). Netrúfame si však povedať, ktorá z možností je lepšia.

Na obrázku 6.3 je znázornený ďalší príklad. Pri vyhodnocovaní výsledkov sme sa stretli s rôznymi jeho variantami. Majme 3 kúsky kódu, pričom všetky obsahujú rovnakú vrchnú časť. Prvé 2 navyše obsahujú aj rovnakú spodnú časť. Predstavme si, že jeden detektor označí ako duplikáty len všetky 3 vrchné časti, spodné vynechá. Druhý nástroj označí ako zduplikované celé prvé 2 fragmenty a tretí fragment vynechá. Ani jeden z nich evidentne neponúkol úplne správne riešenie, ako ich teda navzájom porovnať?

6.2.2 Automatizované vyhodnotenie

Počas vyhodnocovania prechádzajúcich experimentov sme si všimli, že čím menšie boli fragmenty kódu, tým presnejšie sme vedeli povedať, či sú alebo nie sú zduplikované. Pri jednoriadkových fragmentoch sme často dokonca nemali žiadne pochybnosti, či boli alebo neboli odkopírované. Vďaka tomuto zisteniu sme mohli navrhnúť metódu, ktorá nám umožnila automaticky vyhodnotiť presnosť s akou jednotlivé nástroje odhalia duplikáty v našej dátovej vzorke.

Prvý krok uvedenej metódy bol však stále manuálny a subjektívny. Každý riadok kódu v testovacej vzorke (9 súborov) sme označili ako:

- nezduplikovaný — riadky, ktoré by nemali byť ako duplikáty označené *za žiadnych podmienok*; vrátane komentárov a prázdnych riadkov, okolo ktorých sa žiaden zduplikovaný kód nenachádza,
- zduplikovaný — *nepochybne zduplikovaný* riadok kódu spolu so zoznamom všetkých jeho ostatných výskytov,
- ignorovaný — ak nezáleží na tom, či bude označený ako duplikát alebo nie; napríklad prázdne riadky a komentáre okolo alebo vo vnútri iného zduplikovaného kódu.

Takto označené riadky kódov z testovacej vzorky sa nachádzajú aj na DVD v prílohe E v adresári `datasets/manual_experiments/wanted_annotations`.

Tabuľka 6.2: Presnosť a pokrytie dosiahnuté pri automatickom spracovaní testovacej vzorky jednotlivými nástrojmi.

	Presnosť	**Pokrytie**	TP	FP	FN
Simian	10,21%	46,24%	468	4115	544
Flay	46,78%	17,98%	128	207	830
RubyMine	80,22%	14,03%	142	35	870
Náš nástroj	72,07%	65,00%	743	288	400

Vytvorili sme teda akési očakávané výsledky a tie sme následne mohli porovnať s výsledkami každého z testovaných nástrojov. To nám umožnilo vypočítať presnosť a pokrytie (angl. *precision* a *recall*) pre každý z nástrojov. Výsledky uvádzame v tabuľke 6.2.

Na základe týchto výsledkov však nemôžeme testované nástroje medzi sebou objektívne porovnávať. Takisto nemôžeme vyvodzovať žiadne závery o ich kvalite. Napríklad preto, lebo (1) pri označovaní riadkov ako zduplikovaných sme použili vlastnú definíciu duplikátov (viď. [3]), alebo preto, lebo (2) vzorka bola malá a nemusela odzrkadľovať to, aké duplikáty sa nachádzajú v iných, väčších projektoch. Čo však môžeme na základe výsledkov povedať je, že (1) naša metóda vie určite nájsť zduplikovaný kód a že (2) ostatným nástrojom je schopná konkurovať.

6.3 Hodnoty vstupných parametrov

Automatizované vyhodnotenie opísané v predchádzajúcej kapitole je ideálne aj keď chceme zistiť, ako vstupné parametre vplývajú na presnosť odhaľovania duplikátov. To umožní nájsť takú kombináciu parametrov, pri ktorých bude naša metóda poskytovať najpresnejšie výsledky.

Poďme si v krátkosti zopakovať, aké rôzne vstupné parametre vlastne používame:

- *maximálna hĺbka podstromov* – určuje, aké hlboké môžu byť jednotlivé fragmenty kódov; nepriamo ovplyvňuje ich maximálnu veľkosť,
- *minimálny počet vrcholov podstromov* – určuje minimálnu veľkosť vytváraných fragmentov kódu,
- *maximálna dĺžka použitých n-ciest* – z akých dlhých postupností vrcholov sa vytvoria charakteristické vektory; tým ovplyvňuje ich dĺžku,
- *dĺžka zredukovaného vektora* – charakteristický vektor sa viacerými lokálne senzitívnymi hašovaniami zredukuje na túto dĺžku,

- *maximálna vzdialenosť zduplikovaných fragmentov* – hovorí o tom, ako ďaleko od seba môžu byť dva fragmenty, aby boli považované za duplikáty; určuje maximálnu veľkosť klastrov v *Birch* strome,

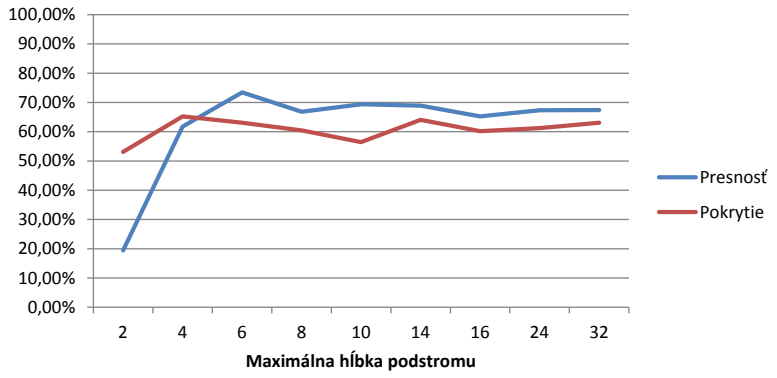
- *vetvenie Birch stromu* – maximálny počet detí, ktoré môžu obsahovať vnútorné vrcholy stromu; ovplyvňuje šírku a hĺbku stromu.

Ako testovaciu vzorku sme opäť použili 9 súborov z predchádzajúcich experimentov. Samotné experimenty prebiehali veľmi jednoducho. Opakovane sme spustili vyhľadávanie duplikátov s rôznymi hodnotami parametrov a sledovali, ako sa mení presnosť a pokrytie. Hodnoty jednotlivých parametrov pochádzali z vopred definovaných intervalov.

6.3.1 Maximálna hĺbka podstromov

Pri prvých experimentoch sme chceli zistiť, ako na dosiahnuté výsledky vplýva veľkosť fragmentov kódu. Maximálna veľkosť fragmentov je čiastočne obmedzená maximálnou hĺbkou podstromov. Čiže ak bude mať hĺbka nízke hodnoty, aj vytvorené fragmenty budú malé. Predpokladáme, že výsledky by v taktom prípade mali byť menej presné (budú obsahovať viac tzv. *false positives*), pretože príliš malé časti kódu môžu byť podobné s inými, aj keď neboli v skutočnosti odkopírované.

Graf na obrázku 6.4 náš predpoklad potvrdzuje. Zobrazuje závislosť presnosti a pokrytia od maximálnej hĺbky podstromov. Hodnoty maximálnej hĺbky nie sú na vodorovnej osi rozložené lineárne. Ostatné parametre boli konštantné a zvolené podľa výsledkov experimentov v nasledujúcich kapito-



Obr. 6.4: Závislosť presnosti a pokrytia od maximálnej hĺbky vytváraných podstromov; pre minimálny počet vrcholov 8, zredukovaný charakteristický vektor dĺžky 8, maximálnu vzdialenosť zduplikovaných fragmentov 20, n-cesty dĺžky 2 a vetvenie Birch stromu 6.

lách. Na DVD v prílohe E sa nachádzajú výsledky rovnakého experimentu aj pre viaceré rôzne hodnoty ostatných parametrov. Všetky však skončili s podobným výsledkom.

Z výsledkov jednoznačne vyplýva, že príliš malá maximálna hĺbka má za následok menšiu presnosť pri hľadaní duplikátov. Zaujímavejšie je, že po dosiahnutí určitej zlomovej hranice (v tomto prípade približne 4 až 6) už výsledky ostávali konštantné.

Pri väčšej maximálnej hĺbke budú aj vytvorené podstromy určite väčšie. Najmä tie z vrchných úrovní abstraktných syntaktických stromov. Veľké podstromy znamenajú väčšie fragmenty kódu a to znamená, že by nemali byť nájdené malé duplikáty. Výsledky by teda mali byť horšie. Na spodných úrovniach stromov, pri listoch, však budú stále vznikať aj nízke podstromy. Nameraná konštantná presnosť teda naznačuje, že:

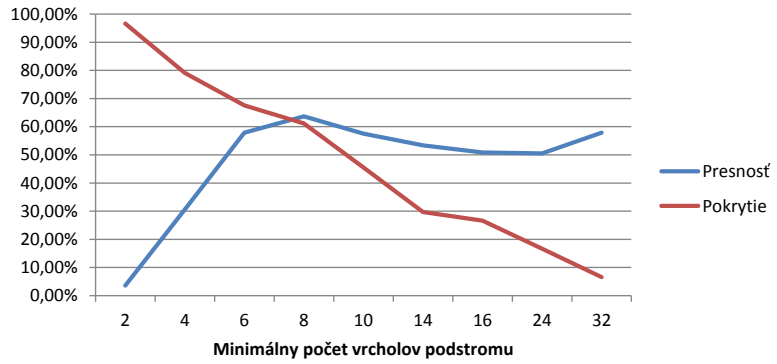
- buď vzorka neobsahovala malé duplikáty, čo podľa nášho názoru nie je pravda, pretože sme sa v nej s takými stretli,
- alebo sa väčšina duplikátov (aspoň tých malých) nachádzala na spodných úrovniach abstraktných syntaktických stromov, a preto boli aj pri veľkej maximálnej hĺbke správne nájdené.

6.3.2 Minimálny počet vrcholov podstromov

Veľkosť fragmentov kódu je ovplyvnená aj minimálnym počtom vrcholov, ktoré príslušné podstromy obsahujú. Predpokladáme, že ak bude táto hodnota nízka, budú vznikať malé fragmenty kódu a tie môžu byť podobné iným aj vtedy, ak neboli v skutočnosti odkopírované. Naopak, ak bude príliš vysoká, nenájdu sa malé duplikáty a presnosť by sa mala zhoršiť.

Na grafe na obrázku 6.5 je znázornená závislosť dosiahnutých výsledkov od rôznych hodnôt minimálneho počtu vrcholov. Upozorňujeme, že opäť nie sú na vodorovnej osi rozložené lineárne. Dĺžka zredukovaného vektora, maximálna vzdialenosť zduplikovaných fragmentov, dĺžka n-ciest a vetvenie *Birch* stromu mali konštantné hodnoty. Experiment bol vykonaný pre rôzne maximálne hĺbky (viď. kapitola 6.3.1) a výsledné hodnoty sú ich priemerom. Na DVD v prílohe E sa dá vidieť, že graf je takmer rovnaký aj pre každú z rôznych maximálnych hĺbok samostatne.

Z výsledkov je jasné, že príliš malé fragmenty naozaj nie sú na odhaľovanie duplikátov vhodné, lebo takmer všetok kód je potom označený ako zduplikovaný (nízka presnosť a vysoké pokrytie). Okolo počtu vrcholov 7 alebo 8 dosahuje presnosť maximum a pre vyššie počty vrcholov ostáva približne rovnaká. Pokrytie však so stúpajúcou minimálnou veľkosťou fragmentov klesá, čo súhlasí s intuitívnym predpokladom – čím väčšie sú fragmenty, tým viac malých a skutočných duplikátov nebude nájdených.

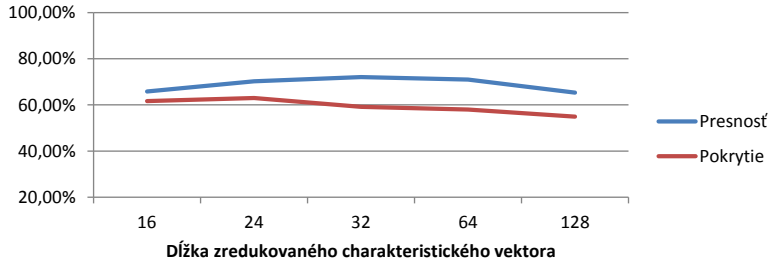


Obr. 6.5: Závislosť presnosti a pokrytia od minimálneho počtu vrcholov vytváraných podstromov; pre zredukovaný charakteristický vektor dĺžky 8, maximálnu vzdialenosť zduplikovaných fragmentov 20, n-cesty dĺžky 2 a vetvenie Birch stromu 6.

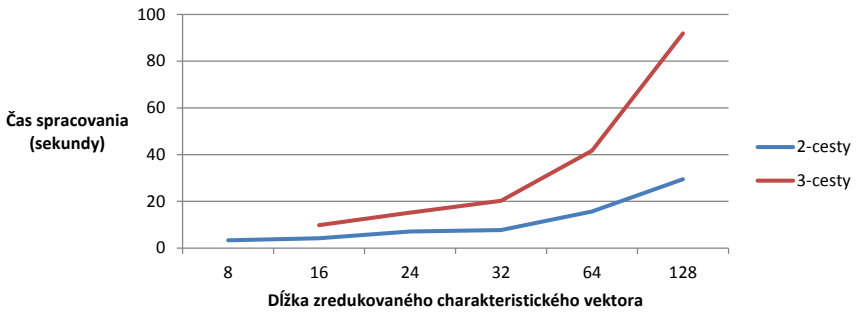
6.3.3 Maximálna dĺžka použitých n-ciest

Ďalej nás pri vyhodnocovaní zaujímalo, či zvýšenie dĺžky n-ciest, z ktorých sa generuje charakteristický vektor pre každý fragment zdrojového kódu, pomôže zvýšiť presnosť odhaľovania duplikátov. Predpokladali sme, že čím dlhšie sú n-cesty, tým presnejšiu informáciu o štruktúre daného fragmentu máme a tým presnejšie vieme fragmenty porovnávať.

Graf na obrázku 6.6 znázorňuje najlepšie namerané hodnoty presnosti a pokrytia pre rôzne dĺžky zredukovaného charakteristického vektora ak použijeme n-cesty dĺžky 3. Hodnoty presnosti sa pohybujú okolo 70% a hodnoty pokrytia okolo 60%. Pri použití n-ciest dĺžky 2 vo všetkých ostatných experimentoch sme dosiahli rovnaké najlepšie výsledky. Z toho vyplýva, že 2-cesty umožňujú zduplikovaný kód odhaliť rovnako presne.



Obr. 6.6: Najlepšie dosiahnuté hodnoty presnosti a pokrytia pre rôzne dĺžky zredukovaného charakteristického vektora, ak použijeme n-cesty s maximálnou dĺžkou 3. Nie sú nijako lepšie ako pri použití dĺžky 2.



Obr. 6.7: Priemerný čas vyhľadávania duplikátov v závislosti od dĺžky zredukovaného charakteristického vektora pre n-cesty dĺžky 2 a 3. Minimálny počet vrcholov, maximálna hĺbka aj maximálna vzdialenosť fragmentov mali rôzne hodnoty (neovplyvňujú čas spracovania), vetvenie Birch stromu bolo 6.

Na obrázku 6.7 sa nachádza graf, ktorý zobrazuje dôležitý rozdiel medzi použitím n-ciest dĺžky 2 a 3. Je ním čas, ktorý je potrebný na vyhľadanie duplikátov. V prípade 3-ciest je viac ako dvakrát väčší. Vzhľadom na to, že 3-cesty neposkytujú väčšiu presnosť a spôsobujú násobne dlhšie vyhľadávanie duplikátov, nemá zmysel ich používať.

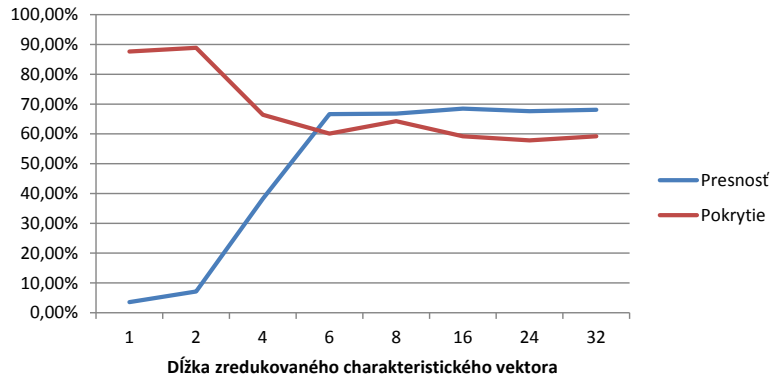
6.3.4 Dĺžka zredukovaného charakteristického vektora

Ďalším parametrom, ktorého vplyv na výsledky vyhľadávania duplikátov sme chceli preskúmať, bola dĺžka na akú sa zredukuje charakteristický vektor fragmentov pred ich porovnávaním. Predpokladali sme, že príliš krátke vektory zhoršia presnosť odhaľovania duplikátov. Zbytočne dlhé vektory by ale presnosť odhaľovania duplikátov už nemali nijako ovplyvniť.

Graf na obrázku 6.8 zobrazuje výsledok tohto experimentu. Použili sme konštantný minimálny počet vrcholov, maximálnu hĺbku podstromu aj vetvenie *Birch* stromu. Dĺžka n-ciest bola 2 a maximálne vzdialenosti duplikátov sa pohybovali medzi hodnotami 10 a 20. Zobrazené presnosti a pokrytia sú priemerom nameraných hodnôt.

Výsledky experimentu jednoznačne potvrdzujú náš predpoklad – príliš krátke vektory znižujú presnosť odhaľovania duplikátov. Naopak vektory dlhšie ako 6 už presnosť odhaľovania duplikátov nijako nezlepšia. Rovnaké výsledky možno vidieť aj na obrázku 6.6, kde boli použité n-cesty dĺžky 3.

Graf na obrázku 6.7 ilustruje, že so zväčšujúcou sa dĺžkou zredukovaného vektora stúpa aj čas potrebný pre vyhľadávanie duplikátov. Keď chceme aby bol čo najnižší, treba použiť aj krátke dĺžky charakteristických vektorov.

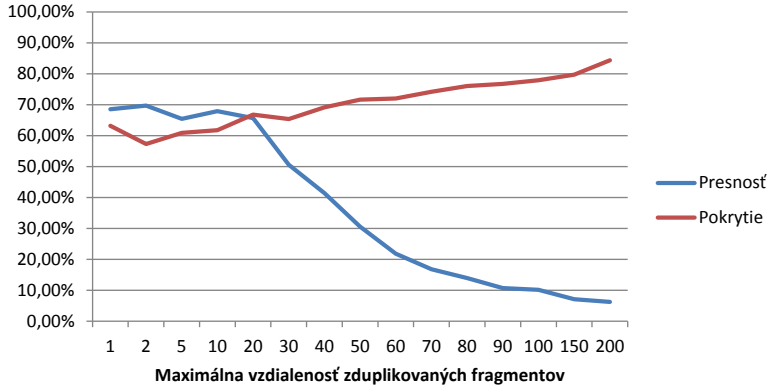


Obr. 6.8: Presnosť a pokrytie v závislosti od dĺžky zredukovaného charakteristického vektora. Minimálny počet vrcholov 8, maximálna hĺbka 6, 2-cesty, maximálna vzdialenosť duplikátov medzi 10 a 20, vetvenie Birch stromu 6.

6.3.5 Maximálna vzdialenosť fragmentov

Ďalej sme sa zamerali na zisťovanie, akým spôsobom vplýva hodnota maximálnej vzdialenosti jednotlivých fragmentov kódu na kvalitu detekcie duplikátov. Určuje, ako veľmi odlišné môžu byť dva kódy, aby boli stále považované za duplikát. Čím väčšia bude maximálna vzdialenosť, tým tolerantnejšia by naša metóda mala byť voči rozdielom v porovnávaných fragmentoch kódu.

Hodnota maximálnej vzdialenosti závisí od dĺžky zredukovaného vektora. Čím viac dimenzií vektor má, tým väčšia by mala byť aj vzdialenosť fragmentov pri zachovaní rovnakej presnosti.



Obr. 6.9: Závislosť presnosti a pokrytia od maximálnej vzdialenosti zduplikovaných fragmentov; pre 2-cesty, zredukovaný vektor dĺžky 8 a vetvenie 6. Minimálny počet vrcholov a maximálna hĺbka podstromov boli rôzne.

Obidva tieto intuitívne predpoklady sme počas experimentov potvrdili. Na obrázku 6.9 sa nachádza graf, ktorý zobrazuje závislosť presnosti a pokrytia od hodnoty maximálnej možnej vzdialenosti fragmentov. Dĺžka zredukovaného vektora bola v tomto prípade 8. Hodnoty vzdialeností na vodorovnej osi nie sú rozložené lineárne.

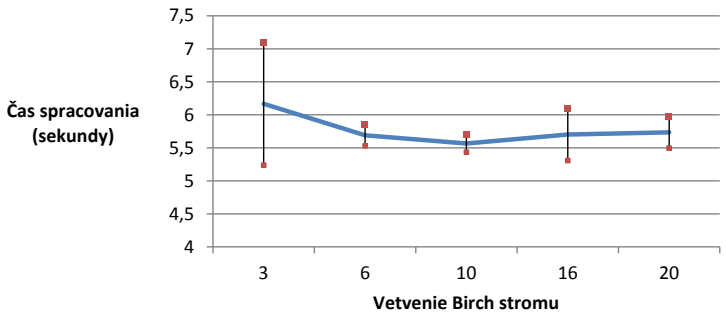
Na DVD v prílohe E sa nachádzajú výsledky rovnakého experimentu s použitím iných dĺžok zredukovaného vektora. Vo všetkých prípadoch bola najlepšia dosiahnutá hodnota presnosti približne 70% a najlepšia hodnota pokrytia približne 60%. Zaujímavé je, že presnosť je pri malých hodnotách vzdialeností približne konštantná. Po dosiahnutí hraničnej hodnoty (v tomto prípade približne 20) však začne postupne klesať.

6.3.6 Vetvenie Birch stromu

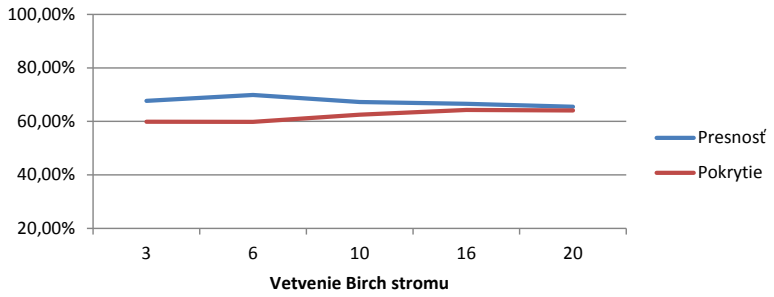
V neposlednom rade nás zaujímalo aj to, akú hodnotu zvoliť pre vetvenie Birch stromu. Tá hovorí o tom, aký maximálny počet detí môžu mať jeho vnútorné vrcholy. Ovplyvňuje teda výslednú šírku a hĺbku stromu. Zaujímalo nás, či vetvenie môže mať vplyv aj na presnosť alebo pokrytie, ale nečakali sme medzi nimi žiadnu súvislosť. Hĺbka stromu a maximálny počet detí vnútorných vrcholov by mohli ovplyvniť čas potrebný pre vyhľadávanie duplikátov.

Čím je strom hlbší, tým dlhšie bude trvať vkladanie, vyhľadávanie a mazanie fragmentov kódu. V tom prípade by bolo vhodné preferovať menej hlboké a širšie stromy. Na druhej strane, čím viac detí je vo vnútorných uzloch, tým viac výpočtov treba vykonať pre vybratie najbližšieho z nich. Potom by bolo rozumnejšie preferovať hlbšie a menej široké stromy.

Z grafov na obrázkoch 6.10 a 6.11 vyplýva, že hodnota vetvenia nemá vplyv ani na presnosť a pokrytie výsledkov, ani na čas vyhľadávania.



Obr. 6.10: Čas spracovania (so štandardnou odchýlkou) nezávisí od vetvenia Birch stromu. Maximálna hĺbka podstromov bola 6, minimálny počet vrcholov 8, n-cesty mali dĺžku 2 a zredukovaný vektor obsahoval 8 prvkov. Maximálne vzdialenosti fragmentov boli z intervalu 5-20 (nemali vplyv na čas).

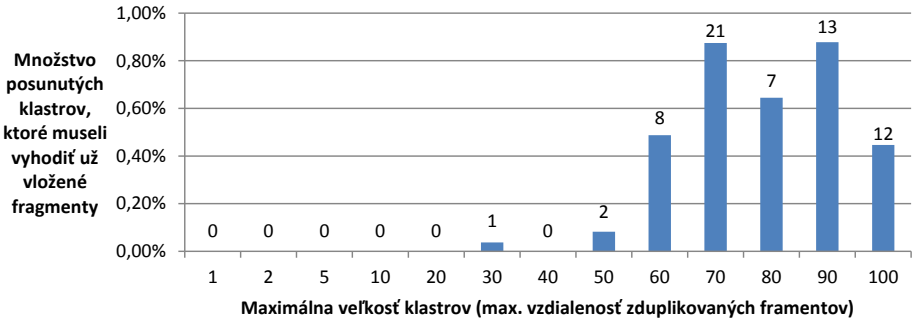


Obr. 6.11: Presnosť a pokrytie v závislosti od vetvenia Birch stromu. Minimálny počet vrcholov v podstrome bol 8, maximálna hĺbka 6 a dĺžka n-ciest 2, zredukovaný vektor obsahoval 8 prvkov a maximálna vzdialenosť fragmentov bola z intervalu 5-20.

6.3.7 Posúvanie centroidov klastrov

Okrem výberu vhodných hodnôt pre jednotlivé vstupné parametre sme sa pri experimentoch pozreli aj na posúvanie klastrov. Konkrétne sme chceli vedieť, ako často dochádza k situácii, že klastre obsahujú bod, ktorý by do nich nemal patriť. To sa môže stať vtedy, keď sa fragmenty kódu do indexu vložia v nevhodnom poradí. Problém bol bližšie opísaný v kapitole 5.2.3. Domnievali sme sa, že k takýmto situáciám nedochádza často a že ich môžeme zanedbať.

Obrázok 6.12 znázorňuje, v akom množstve klastrov došlo pri experimentoch k neželanému posunutiu, v závislosti od maximálnej veľkosti klastrov. Pri experimente boli použité 2-cesty a dĺžka zredukovaného vektora 8. Vidíme, že zlé klastrovanie nastalo v necelom 1% prípadov. Aj to len vtedy, ak boli klas-



Obr. 6.12: Množstvo klastrov, z ktorých bol kvôli posunu vylúčený nejaký fragment kódu. Čísla nad stĺpcami udávajú konkrétny počet vylúčených fragmentov. Použité dáta pochádzajú zo 150 experimentov, pri ktorých boli použité 2-cesty a dĺžka zredukovaného vektora 8.

tre príliš veľké (v kapitole 6.3.5 sme zistili, že pri vzdialenosti zduplikovaných fragmentov väčšej ako 20 dochádzalo k výraznému poklesu presnosti). Chyby odhaľovania duplikátov v dôsledku posúvania klastrov môžeme teda zanedbať.

6.3.8 Zhrnutie

Z experimentov, ktoré sme opísali v tejto kapitole, vyplynuli najvhodnejšie hodnoty pre jednotlivé parametre:

- *maximálna hĺbka podstromov*: 6; podľa výsledkov by stačila nižšia, nechceli sme použiť hraničnú hodnotu, ak by sa pri iných dátach posunula,
- *minimálny počet vrcholov podstromov*: 8,
- *maximálna dĺžka použitých n-ciest*: 2,
- *dĺžka zredukovaného vektora*: 8; podľa experimentov by síce stačilo aj 6, ale vyššiu hodnotu sme zvolili kvôli tomu, ak by sa pri neskorších experimentoch objavili nové kombinácie 2-ciest, ktoré by predĺžili charakteristický vektor (čím dlhší je originálny charakteristický vektor, tým dlhší by mal byť aj zredukovaný),
- *maximálna vzdialenosť zduplikovaných fragmentov*: 10; podľa experimentov by sme mohli použiť až hodnotu 20, ale (1) medzi 10 a 20 sa neprejavil žiaden rozdiel v presnosti a (2) hodnotu sme nechceli nechať na hranici, ak by sa táto hranica pri spracovaní iných dát posunula,
- *vetvenie Birch stromu*: 10; vetvenie nemalo vplyv na čas spracovania ani kvalitu výsledkov, hodnotu sme vybrali intuitívne.

Výsledky a dáta všetkých experimentov, ktoré boli opísané v tejto kapitole, sa nachádzajú aj na DVD v prílohe E. Samotné spracované vyhodnotenie sa dá nájsť v súbore `results/automatic_experiments/params_results.xlsx`. Nespracované výstupy experimentov (vrátane nájdených duplikátov) sa nachádzajú v adresári `results/automatic_experiments/params_data`.

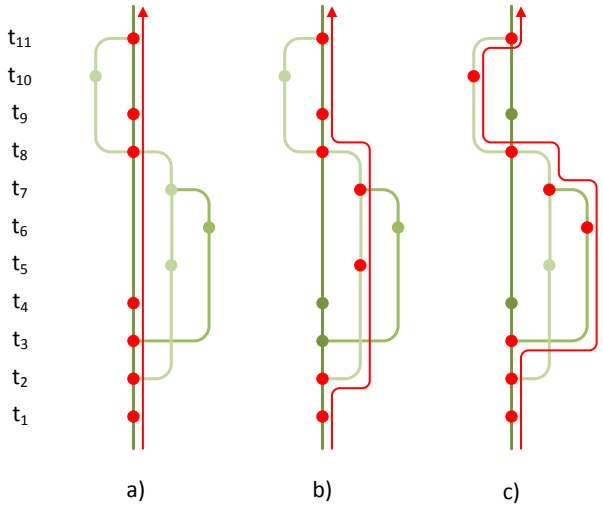
Ako bolo uvedené v úvode kapitoly, naša testovacia vzorka dát pozostávala iba z 9 súborov a približne 730 riadkov kódu. Napriek tomu, že vzorka pochádza z reálneho softvérového projektu, (1) nemusí byť dostatočne veľká alebo (2) dostatočne reprezentatívna na to, aby odrážala zduplikovaný kód vo všeobecnosti. V takom prípade by získané hodnoty vstupných parametrov nemuseli byť optimálne globálne. Napriek tomu ich však môžeme považovať aspoň za blízke týmto globálne optimálnym hodnotám.

6.4 Experimenty s históriou veľkých projektov

V tejto sekcii sú prezentované výsledky experimentov, ktoré sme vykonávali na histórii väčších softvérových projektov. Použili sme webovú (Ruby on Rails) aplikáciu *Wanted*, ktorá bola vyvíjaná v rámci predmetov Tímový projekt 1 a 2 počas akademického roku 2011/2012 a open-source Ruby knižnicu *Grit*:

- aplikácia *Wanted* bola aktívne vyvíjaná počas doby 1 roka štyrmi vývojármi; repozitár obsahuje približne 770 revízií; v poslednej pozostáva z približne 11000 riadkov Ruby kódu,
- open-source knižnica *Grit*[1] je vyvíjaná od roku 2007, posledná revízia je z februára 2013; celkovo repozitár obsahuje približne 510 revízií a v poslednej verzii obsahuje asi 7500 riadkov Ruby kódu; knižnica slúži na prácu s *Git* repozitármi a je používaná napríklad aj službou *GitHub*.

Git a vetvenie. Predtým, ako začneme spracovávať históriu vývoja spomínaných projektov, sa musíme pozrieť ako funguje vetvenie vo verziovacom systéme Git. Vždy, keď viacerí vývojári paralelne pracujú na zdrojovom kóde, implicitne používajú rôzne vetvy. Situácia je ilustrovaná na obrázku 6.13, kde sa nachádza 11 usporiadaných revízií vykonaných v časoch t_1 až t_{11}.



Obr. 6.13: 11 chronologicky zoradených revízií s viacerými paralelnými vetvami. Pri spracovaní histórie projektu treba postupovať vždy len pozdĺž jednej z vetiev, pričom nezáleží na tom, ktorá to bude. Obrázok ilustruje 3 možné cesty cez týchto 11 revízií.

[1] https://github.com/mojombo/grit

Dve paralelné vetvy vznikli napríklad v čase t_2. Predstavme si teraz, že jeden vývojár sa rozhodne úplne odstrániť nejaký súbor s kódom – v revíziách t_3 alebo t_4. Celý ho vymaže, pretože napríklad už nie je potrebný. Druhý vývojár má však stále iba revíziu t_2. O odstránenom súbore nič nevie a začne ho napríklad refaktorovať. Keď svoju prácu dokončí, vytvorí revíziu t_5. Tá vznikne chronologicky neskôr ako revízie t_3 a t_4, avšak robí zmeny v kóde, ktorý v dvoch starších revíziách už neexistuje. Pri spojení vetiev sa samozrejme všetko vyrieši a správne zosynchronizuje.

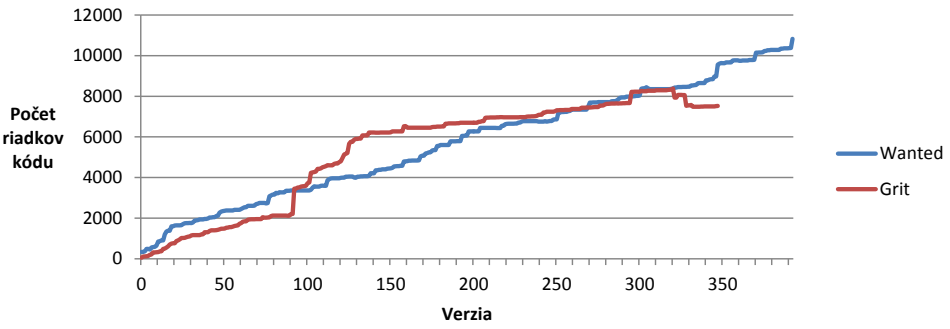
Aký problém to však spôsobuje? Ak by sme všetky revízie chceli spracovávať chronologicky, tak sa v tomto prípade stane, že pri spracovávaní t_5 budeme hľadať zdrojový kód, ktorý sme pri spracovávaní predchádzajúcich revízií už vymazali. Z viacerých paralelných vetiev si teda pri spracovaní histórie projektu treba vybrať vždy len jednu. Nezáleží na tom ktorú, pretože kód sa pri spájaní vetiev zosynchronizuje.

Obrázok 6.13 znázorňuje 3 rôzne cesty, ktorými sa dá prejsť cez daných 11 revízií. Všimnime si, že každá cesta obsahuje iné revízie (vyznačené na červeno) – niektoré má navyše, iné ignoruje. Pri našich experimentoch sme v prípade obidvoch projektov náhodne vybrali 1 takúto cestu. Pri projekte *Wanted* obsahovala 393 revízií a cesta z projektu *Grit* obsahovala 348 revízií.

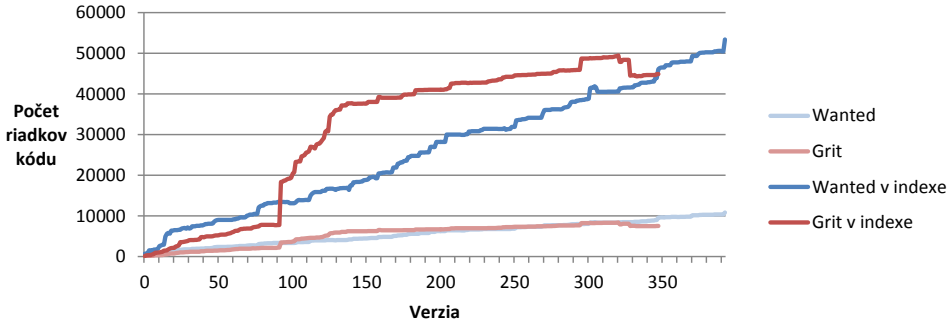
6.4.1 Veľkosť projektov

Pozrime sa na to, koľko kódu obsahovali obidva projekty počas svojej histórie, v jednotlivých spracovávaných verziách. Graf sa nachádza na obrázku 6.14. Vidíme, že v nových verziách väčšinou kód do obidvoch projektov pribúdal.

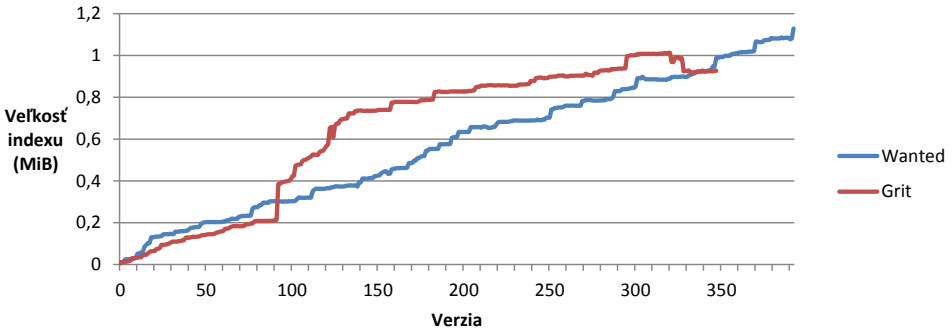
Pri indexovaní kódu sa vytvárajú fragmenty, ktoré sa prekrývajú. To znamená, že index bude obsahovať niektoré riadky zdrojového kódu viackrát. Keby sme teda zobrali všetky fragmenty kódu z indexu a položili ich vedľa seba



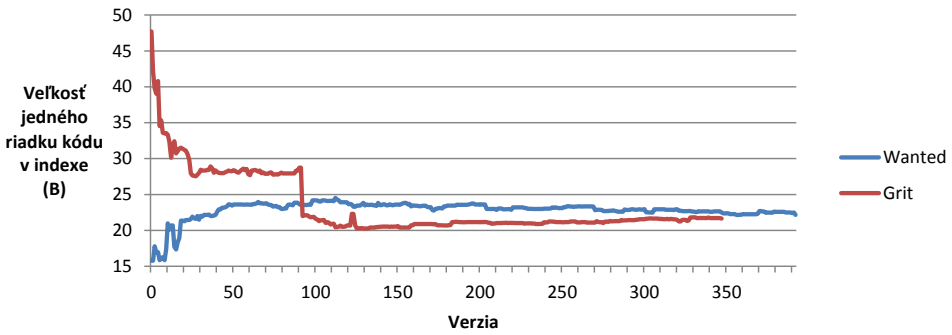
Obr. 6.14: Súčet počtov riadkov vo všetkých súboroch s príponou `.rb` pre každú zo spracovávaných verzií projektov *Grit* aj *Wanted*.



Obr. 6.15: Súčet počtov riadkov všetkých fragmentov kódu, ktoré boli v danej verzii v indexe, pre obidva projekty. V posledných verziách index obsahuje 5 až 6-násobne viac kódu.



Obr. 6.16: Veľkosť serializovaného indexu pre jednotlivé verzie.



Obr. 6.17: Veľkosť indexu v prepočte na 1 riadok zaindexovaného kódu. Ustáli sa na hodnote približne 23 bajtov.

(bez ohľadu na to, že sa prekrývajú), dokopy by mali omnoho viac riadkov ako reálne v projekte existuje. Tento počet je znázornený na obrázku 6.15.

A aká bola veľkosť samotného indexu? Tú zobrazuje graf na obrázku 6.16. Jeho tvar je rovnaký ako v prípade počtu riadkov kódu v projekte a v indexe. To znamená, že najväčší vplyv na veľkosť indexu má práve množstvo kódu, ktoré je v ňom vložené. Veľkosť indexu a množstvo kódu v indexe mali v obidvoch prípadoch viac ako 99,8% koreláciu.

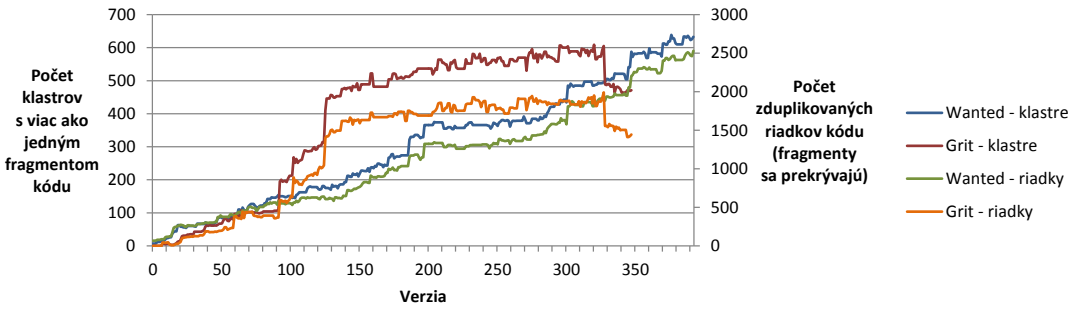
Na obrázku 6.16 tiež vidno, že index s 50000 riadkami kódu zaberá serializovaný len asi 1,1 MiB miesta. To znamená, že 1 riadok kódu zaberá približne len 23 B (viď. obrázok 6.17).

6.4.2 Zduplikovaný kód v použitých projektoch

Koľko zduplikovaného kódu sme teda v obidvoch projektoch našli? *Wanted* v poslednej verzii obsahoval 633 takých skupín s duplikátmi, ktoré v sebe mali 2 a viac fragmentov kódu. Spolu všetky pozostávali z 2531 unikátnych riadkov. *Grit* obsahoval v poslednej verzii takýchto skupín 471 a bolo v nich 1444 unikátnych riadkov kódu. Množstvo odhalených duplikátov v starších verziách obidvoch projektov sa nachádza na obrázku 6.18.

Prečo hovoríme o tak vysokom počte skupín so zduplikovaným kódom (500 – 600)? Je to preto, lebo ľubovoľný fragment, alebo jeho časť, sa mohli nachádzať vo výsledkoch aj viackrát, vo viacerých skupinách. Takáto situácia bola častá, napríklad ak 1. skupina obsahuje celú zduplikovanú metódu, aj s jej definíciou. 2. skupina potom obsahuje len telo tej istej metódy (v strome bolo o úroveň nižšie, malo vytvorený samostatný podstrom) a 3. skupina napríklad obsahuje len *for* cyklus z tela danej metódy.

Prečo sme takéto prekrývajúce sa duplikáty a skupiny nespojili dokopy? Je to kvôli situáciám, aká je napríklad znázornená na obrázku 6.3. Ak by sme uvedené duplikáty spojili do 1 finálnej skupiny, niekto by mohol namietať,



Obr. 6.18: Počet skupín zduplikovaného kódu a počet zduplikovaných riadkov pre každú revíziu obidvoch projektov.

že 3. fragment nie je duplikátom prvých dvoch. Vypísať uvedené duplikáty v 2 samostatných skupinách (aj za cenu toho, že sa prekrývajú) sa nám preto zdalo byť vhodnejšie riešenie.

Wanted v poslednej verzii obsahoval 10826 riadkov kódu a *Grit* 7522 riadkov. V priemere bolo teda v každej verzii *Wanted* zduplikovaných 18,9% riadkov kódu (štandardná odchýlka 2,78%) a v každej verzii *Grit* 21,01% riadkov (s odchýlkou 6,13%).

To sú dosť veľké množstvá zduplikovaného kódu. V tomto bode sme zapochybovali o úspešnosti navrhnutej metódy. Potrebovali sme preto skontrolovať, či výsledky neobsahujú až príliš veľa tzv. *false positives*? Pri manuálnej kontrole sme však väčšinu výsledkov považovali za naozaj správne.

Rozhodli sme sa teda výsledky aspoň približne porovnať s niektorým z existujúcich nástrojov na odhaľovanie zduplikovaného kódu. *Simian* sa nám pri experimentoch opísaných v kapitole 6.2.1 neosvedčil, *Flay* síce oznámi, na ktorom riadku duplikát začína, ale kde končí sa od neho nedozvieme (to znamená, že nevieme ľahko spočítať celkový počet zduplikovaných riadkov). Ostal nám teda len *RubyMine*. Ten v projekte *Wanted* našiel približne 1400 zduplikovaných riadkov kódu, v projekte *Grit* 750 riadkov (a vieme o ďalších približne 120 riadkoch, ktoré vynechal). *RubyMine* teda tvrdí, že obidva projekty obsahovali približne 12% zduplikovaného kódu. Ak k tomu pridáme fakt, že *RubyMine* duplikáty odhaľuje len s veľmi nízkym pokrytím (tabuľka 6.2), tak 20% zduplikovaného kódu začne znieť omnoho reálnejšie.

6.4.3 Jednorazové vs. inkrementálne spracovanie

Pri ďalších experimentoch sme chceli porovnať jednorazové a inkrementálne odhaľovanie duplikátov. Najzaujímavejšie by určite bolo porovnať našu inkrementálnu metódu s existujúcimi jednorazovými nástrojmi. Tie by sme však museli vyhodnocovať manuálne a v kapitole 6.2.1 sme už opísali, že takéto vyhodnotenie (1) je veľmi obtiažne spraviť a (2) jeho výstupy sú subjektívne a preto len orientačné. Ak by sme chceli vykonať experimenty ako v kapitole 6.2.2, potrebovali by sme manuálne označiť duplikáty v mnohých verziách obidvoch projektov, ktoré majú tisíce riadkov zdrojových kódov. To tiež nevyzerá ako realizovateľná možnosť.

Predchádzajúce experimenty ukázali, že navrhnutá metóda vie odhaliť zduplikovaný kód. Nás momentálne zaujíma len porovnanie jednorazového a inkrementálneho spracovania. Chceme vedieť, či dlhodobé aktualizácie indexu nebudú mať za následok napríklad zníženie presnosti odhaľovania duplikátov. Na to nám však netreba metódu porovnávať s konkurenčnými nástrojmi. Stačí, ak ju porovnáme samu so sebou.

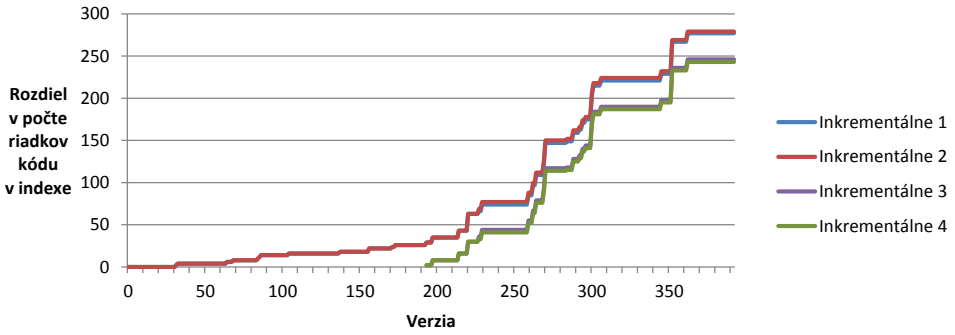
Každú revíziu z celej histórie testovacieho projektu najprv spracujeme samostatne, jednorazovo. Potom celú históriu projektu spracujeme postupne,

inkrementálne. V ideálnom prípade by pre každú z verzií malo jednorazové spracovanie odhaliť presne také isté duplikáty, ako inkrementálne. Realita však bude určite iná, pretože *Birch* je pri klastrovaní citlivý na poradie vstupných dát [32]. To bude pri jednorazovom spracovaní určite iné, ako pri inkrementálnom. Chceli by sme teda zistiť, či sa výsledky medzi jednorazovým a inkrementálnym spôsobom budú líšiť len tým, že fragmenty mali iné vstupné poradie, alebo či na ne vplýva aj niečo iné – napríklad práve nejaká forma degradácie indexu.

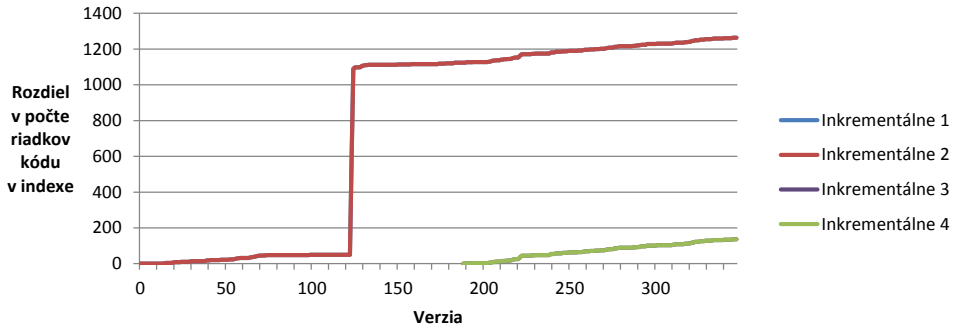
Vykonané experimenty. Na obidvoch projektoch sme vykonali viaceré experimenty, resp. viaceré vyhľadávania duplikátov. V nasledujúcom texte sa na ne budeme odkazovať nasledujúcimi názvami:

- *jednorazové* spustenie – každá verzia bola spracovaná úplne nezávisle; čas spracovania sme nemerali, ale pohyboval sa rádovo v hodinách,
- *inkrementálne 1* a *inkrementálne 2* – každá verzia bola spracovaná tak, že sa len aktualizoval index prechádzajúcej verzie projektu; dĺžka trvania bola rádovo desiatky minút,
- *inkrementálne 3* a *inkrementálne 4* – náhodne zvolená verzia, približne zo stredu histórie projektu, bola zaindexovaná jednorazovo a všetky novšie verzie od nej boli následne spracované inkrementálne.

Rozdiel v počte zaindexovaných riadkov kódu. Na získaných výsledkoch sme si všimli fakt, že pri inkrementálnom spracovaní obsahoval index vždy viac riadkov zdrojového kódu, ako pri jednorazovom spracovaní. Obrázok 6.19 zobrazuje tieto rozdiely v projekte *Wanted*.



Obr. 6.19: Rozdiel v počte zaindexovaných riadkov medzi jednorazovým a každým inkrementálnym spracovaním - projekt *Wanted*.



Obr. 6.20: Rozdiel v počte zaindexovaných riadkov medzi jednorazovým a každým inkrementálnym spracovaním - projekt *Grit*.

Na obrázku 6.19 je vidieť, že v inkrementálnom indexe sa postupne a nárazovo objavovalo čím ďalej, tým viac zaindexovaného kódu. To implikuje, že sa tam raz za čas dostane taký kód, ktorý sa z indexu už nikdy nevymaže. Zároveň môžeme na obrázku 6.19 vidieť, že pri polovičných inkrementálnych spusteniach 3 a 4 rozdiel stúpal rovnako, ako pri 1 a 2. To znamená, že rozdiel v počte zaindexovaných riadkov nie je závislý od veku indexu, resp. od počtu verzií, počas ktorých index existoval. Naopak, rozdiel je pravdepodobne závislý od samotného zdrojového kódu. Spôsobuje ho zrejme len implementačná chyba, ktorú sa nám však nepodarilo odhaliť.

V prípade projektu *Wanted*, kde bolo v poslednej verzii v indexe vyše 53000 riadkov kódu, je tento rozdiel necelých 300 riadkov zanedbateľný. Tvorí iba 0,5% zaindexovaného kódu. Trochu horšia bola situácia pri projekte *Grit*. Na obrázku 6.20 vidno, že počas spracovávania dvoch po sebe idúcich revízií do indexu pribudlo až 1000 riadkov, ktoré potom ostali zabudnuté. Zo 45000 zaindexovaných riadkov je to približne 2,67%.

Dôvod, kvôli ktorému sa v inkrementálnom indexe objavujú zabudnuté riadky kódu, sme sa rozhodli ďalej nehľadať. Najmä preto, lebo (1) množstvo zabudnutého kódu je relatívne nízke a (2) jeho dôsledok sa dá jednoducho minimalizovať znovuvytvorením indexu (napr. ako v prípade inkrementálnych spracovaní 3 a 4 na obrázku 6.20). Experimenty v kapitole 6.5 dokazujú, že zabudnutý kód sa na finálnych výsledkoch nijako neprejaví.

Rozdiely v odhalených duplikátoch. Ďalším dôležitým faktorom pri porovnávaní jednorazového a inkrementálneho spracovania je to, či odhalia rovnaký zduplikovaný kód. Práve na to sme sa chceli pozrieť v nasledujúcich experimentoch. Porovnávať odhalené duplikáty manuálne, v stovkách verzií, by bolo nerealizovateľné. Úlohu sme preto opäť automatizovali, a to nasledujúcim spôsobom. Pre každú jednu verziu obidvoch projektov sme zobrali zoznam

Tabuľka 6.3: Koľko zduplikovaných riadkov navyše našlo jednorazové a inkrementálne spracovanie vzhľadom na počet riadkov, ktoré označili obidva spôsoby rovnako.

	Grit	**Wanted**
Inkrementálne 1	15,38% (± 5,60%)	6,70% (± 2,71%)
Inkrementálne 2	15,28% (± 5,82%)	6,80% (± 2,84%)
Inkrementálne 3	14,39% (± 4,71%)	6,98% (± 8,69%)
Inkrementálne 4	15,00% (± 5,03%)	6,64% (± 8,29%)
Priemerne	15,02% (± 5,29%)	6,78% (± 5,63%)
Jednorazové 1	8,31% (± 3,53%)	6,47% (± 2,58%)
Jednorazové 2	11,06% (± 6,26%)	8,93% (± 4,40%)
Jednorazové 3	12,39% (± 3,18%)	8,29% (± 2,95%)
Jednorazové 4	9,81% (± 1,84%)	7,59% (± 2,64%)
Priemerne	10,40% (± 3,71%)	7,82% (± 3,14%)

duplikátov, ktoré boli odhalené inkrementálne a jednorazovo. Porovnali sme riadky kódu, ktoré sa v duplikátoch nachádzali a rozdelili ich do 3 skupín:

1. riadky, ktoré boli ako duplikáty označené aj pri jednorazovom a aj pri inkrementálnom spracovaní,
2. riadky, ktoré ako duplikáty označilo len jednorazové spracovanie,
3. riadky, ktoré ako duplikáty označilo len inkrementálne spracovanie.

Výsledky týchto experimentov sumarizuje tabuľka 6.3. Obsahuje pre obidva projekty informácie o tom, koľko zduplikovaných riadkov navyše odhalilo každé inkrementálne spracovanie. Takisto zobrazuje, koľko riadkov naviac našlo jednorazové spracovanie oproti každému zo štyroch inkrementálnych.

Inými slovami povedané, ak inkrementálne a jednorazové spracovanie označili ako duplikáty 100 rovnakých riadkov kódu, tak v prípade projektu *Grit* a 1. experimentu malo inkrementálne spracovanie vo výsledkoch ďalších 15 odlišných riadkov (dokopy 115) kódu a jednorazové ďalších 8 (dokopy 108).

Pri projekte *Grit* sme si všimli, že inkrementálne spracovanie malo tendenciu označiť viac zduplikovaného kódu (v priemere 15% vs. 10%). *Wanted* však rovnakú tendenciu nepotvrdil.

Zhoršovanie kvality výsledkov. Asi najdôležitejšia časť jednorazových a inkrementálnych experimentov je overenie, či pri inkrementálnom spracovaní nezačne postupne dochádzať k zhoršovaniu kvality dosahovaných výsledkov.

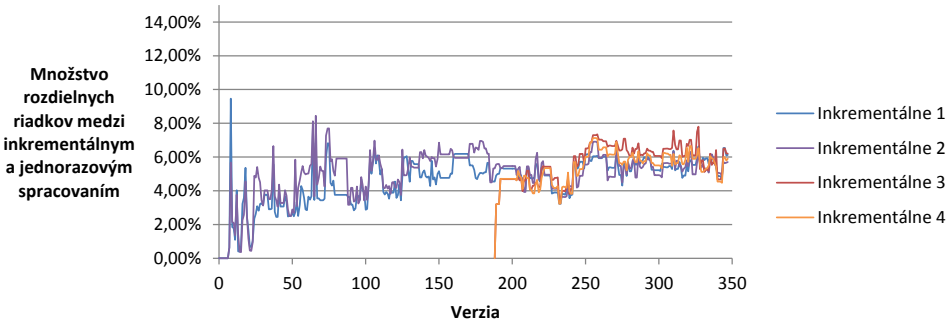
Prvou možnosťou, ako sa dá kvalita výsledkov vyhodnotiť, je sledovať, či sa postupne nezvyšuje rozdiel medzi duplikátmi nájdenými jednorazovým a inkrementálnym spracovaním. Nemôžeme sa však pozerať na absolútnu hodnotu tohto rozdielu. Na obrázku 6.14 bolo znázornené, že množstvo kódu v obidvoch projektoch časom pribúda. Z toho vyplýva, že kód bude časom obsahovať aj viac zduplikovaných riadkov. Čím viac duplikátov, tým väčší bude aj absolútny rozdiel medzi jednorazovým a inkrementálnym spracovaním.

Môžeme sa však pozrieť na relatívne hodnoty tohto rozdielu. Počty rozdielne označených riadkov predelíme celkovým počtom riadkov v projekte v danej verzii. Takto vypočítané rozdiely sú znázornené na obrázku 6.21 pre projekt *Grit* a na obrázku 6.22 pre projekt *Wanted*.

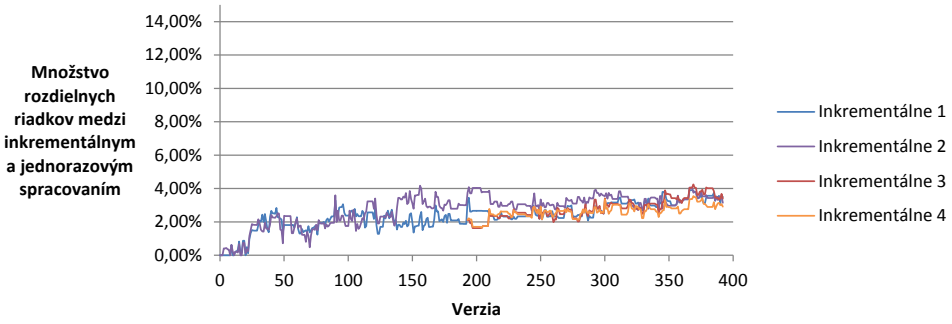
Ak vezmeme do úvahy iba inkrementálne spracovania 1 a 2, tak nevieme z pohľadu na graf jednoznačne povedať, či rozdiel stúpa alebo nie. Podľa nášho názoru tam dokonca nejaký minimálny rast vidieť. Tu však prichádzajú ku slovu inkrementálne spracovania 3 a 4. Tie boli inicializované približne v polovici histórie a inkrementálne spracovali len novšie verzie.

Predpokladajme teraz, že kvalita indexu sa časom zhoršuje. To znamená, že rozdiel v duplikátoch odhalených jednorazovým a inkrementálnym spracovaním sa pomaly zvyšuje. Po spracovaní prvej polovice histórie projektu by tento rozdiel už mohol byť viditeľný. Vytvorme teraz, v polovici histórie, nový index duplikátov. Počas spracovania druhej polovice histórie budeme aktualizovať obidva indexy. Ak naozaj dochádza k akejsi strate kvality, duplikáty získané zo staršieho indexu budú mať medzi inkrementálnym a jednorazovým spracovaním väčšie rozdiely, ako tie z novšieho indexu.

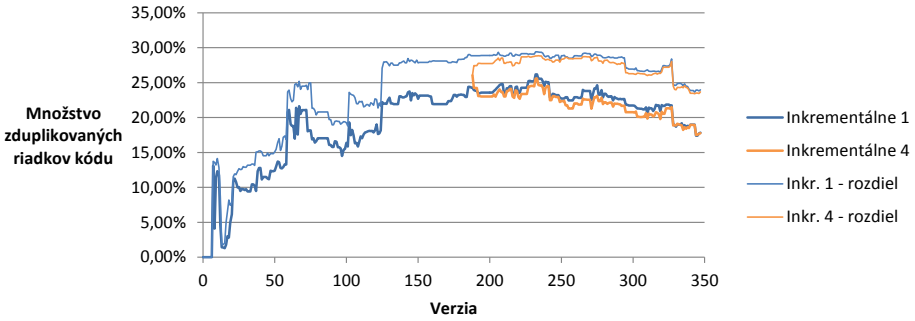
Na obrázku 6.21 aj 6.22 môžeme vidieť, že inkrementálne spracovania 3 a 4 majú okamžite po ich vytvorení skoro rovnaké hodnoty rozdielov, ako inkrementálne spracovania 1 a 2. Obrázky 6.23 a 6.24 znázorňujú, koľko percent kódu bolo ako duplikát rovnako označených jednorazovým aj inkrementálnym



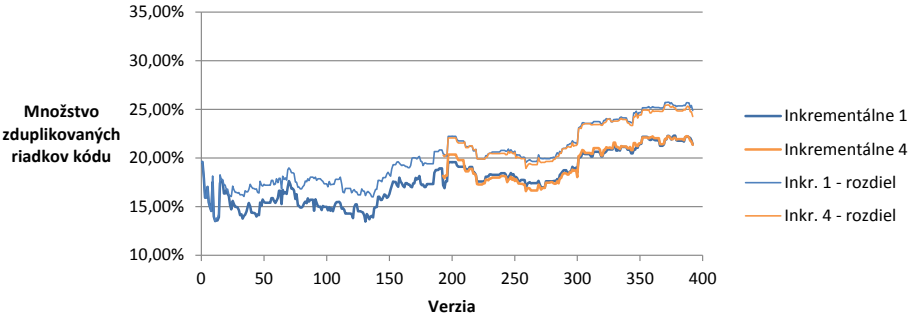
Obr. 6.21: Množstvo rozdielnych riadkov zduplikovaného kódu medzi každým inkrementálnym a jednorazovým spracovaním - projekt *Grit*.



Obr. 6.22: Množstvo rozdielnych riadkov zduplikovaného kódu medzi každým inkrementálnym a jednorazovým spracovaním - projekt *Wanted*.



Obr. 6.23: Množstvo rovnako a rozdielne označených riadkov zduplikovaného kódu medzi inkrementálnym a jednorazovým spracovaním - projekt *Grit*.



Obr. 6.24: Množstvo rovnako a rozdielne označených riadkov zduplikovaného kódu medzi inkrementálnym a jednorazovým spracovaním - projekt *Wanted*.

spracovaním (hrubá čiara). Tenká čiara znázorňuje, koľko kódu bolo navyše označených rozdielne. Aj tu vidíme, že duplikáty z polovičného spracovania takmer dokonale kopírujú duplikáty z kompletného spracovania.

Prezentované výsledky naznačujú, že ku strate kvality indexu nedochádza. Aspoň nie pri spracovaní „len" 400 verzií.

6.5 Vplyv veku indexu na presnosť a pokrytie

Experimenty opísané v predchádzajúcom texte naznačujú, že rozdiely medzi duplikátmi odhalenými jednorazovým a inkrementálnym spôsobom sa spolu s vekom indexu zrejme nezhoršujú. Tento výsledok sme však stále nepovažovali za jednoznačne potvrdený. Zároveň sme chceli vidieť aj to, ako sa s postupujúcim časom vyvíjajú hodnoty presnosti a pokrytia.

Aby sme vedeli vypočítať presnosť a pokrytie na projektoch *Grit* a *Wanted*, potrebovali by sme v každej verzii nájsť a manuálne označiť tisícky riadkov duplikátov. Zduplikovaný kód sa navyše medzi verziami stále mení. Dve rôzne verzie s rovnakým množstvom kódu môžu mať výrazne odlišné množstvo duplikátov. Inými slovami povedané, pri experimente, ktorý sme chceli vykonať, sme potrebovali stabilnejšie, resp. viac kontrolovateľné podmienky.

Rozhodli sme sa použiť tie isté dáta, ktoré sme používali pri experimentoch v kapitole 6.3. Bolo to 9 súborov, v ktorých sme manuálne označili duplikáty. Experiment sme navrhli nasledovne:

1. vložiť do indexu kód zo všetkých 9 súborov,
2. vyhodnotiť presnosť a pokrytie výsledkov,
3. k-krát vykonať nasledujúce (reálne sme vykonali 688 takýchto iterácií):
   a) vytvoriť náhodnú permutáciu z daných 9 súborov (preto, aby bola každá iterácia odlišná),
   b) každý súbor z indexu úplne celý vybrať a následne vložiť naspäť (tak prebieha aj inkrementálne spracovanie, stará verzia súboru sa z indexu celá odstráni a vzápätí sa do neho vloží nová verzia),
   c) vyhodnotiť presnosť a pokrytie výsledkov.

Počet riadkov kódu v indexe. Opäť sme sa najprv pozreli na to, koľko riadkov kódu sa v indexe po jednotlivých iteráciách nachádzalo. Aj tu sa prejavila rovnaká chyba ako v kapitole 6.4.3, ktorá spôsobovala, že niektoré riadky kódu ostali v indexe zabudnuté. Keďže sa v každej iterácii pridával a mazal rovnaký kód, vždy sa v indexe zabudlo aj rovnaké množstvo kódu – 12 riadkov. Po 688 iteráciách bolo dokopy v indexe 8200 riadkov zabudnutého kódu.

Zvyšný zaindexovaný kód mal spolu približne 4300 riadkov. Zabudnutého kódu bolo až dvojnásobne viac.

To je veľmi vysoké číslo. Ku zabúdaniu dochádzalo aj v takýchto kontrolovaných podmienkach – pridával a mazal sa stále rovnaký kód, dokonca stále rovnaké podstromy. To znamená, že s veľkou pravdepodobnosťou ide naozaj len o implementačnú chybu, ktorá sa zrejme bude dať jednoducho odstrániť.

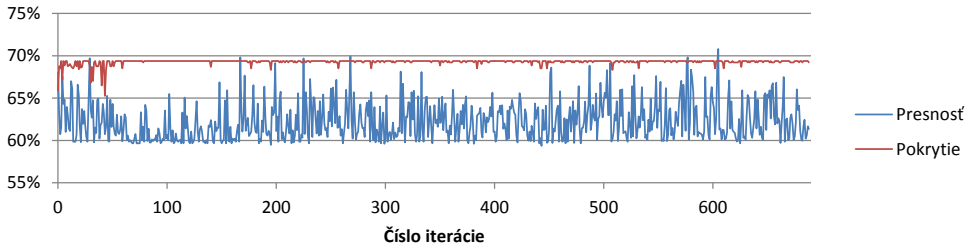
Presnosť a pokrytie. Omnoho zaujímavejšie však je, že napriek tomu, že v indexe boli po poslednej iterácii až 2/3 kódu navyše, na presnosti a pokrytí sa to vôbec neprejavilo – graf na obrázku 6.25.

Prečo zabudnuté riadky vôbec neovplyvnili výsledky? Boli totiž v každej iterácii rovnaké. To znamená, že pri klastrovaní boli vždy vložené do rovnakého listu v *Birch* strome. Konkrétne to boli dva listy, po poslednej iterácii obsahovali 5519 a 2764 fragmentov kódu (z čoho vyplýva, že v každej verzii sa zabudlo na 12 jednoriadkových fragmentov). Pri čítaní duplikátov z indexu však vykonávame krok, ktorý ich zo záverečných výsledkov odstránil. Kontrolujeme, či jednotlivé klastre v sebe neobsahujú rovnaké alebo prekrývajúce sa fragmenty. Ak áno, tak ich spojíme dokopy. Vďaka tomu sa každý z fragmentov vo výsledkoch objavil iba raz.

Graf na obrázku 6.25 tiež úplne jasne znázorňuje, že ani po skoro 700 iteráciách *vôbec nedochádza k žiadnemu poklesu presnosti alebo pokrytia*. Môžeme teda s istotou povedať, že na indexe duplikátov sa žiadne starnutie neprejavuje a *výsledky sa časom nezhoršujú*.

Bližší pohľad na získané výsledky. Z grafu je zrejmé, že pokrytie má celý čas takmer konštantnú hodnotu. V každej iterácii naša metóda odhalila skoro 70% očakávaných zduplikovaných riadkov – dovolíme si tvrdiť, že vždy to bolo rovnakých 70% riadkov. Presnosť sa však až náhodne mení.

Ak je pokrytie stále rovnaké a presnosť rôzna, znamená to, že rozdiely vo výsledkoch medzi jednotlivými iteráciami boli spôsobené tzv. *false positives*.



Obr. 6.25: Presnosť a pokrytie odhalených riadkov zduplikovaného kódu v manuálne označenej testovacej vzorke pre jednotlivé vykonané iterácie.

lib/grit/git-ruby.rb

```
121.         files.each do |ref|
122.           next if !File.file?(ref)
123.           id = File.read(ref).chomp
124.           name = ref.sub("#{prefix}/", '')
125.           if !already[name]
126.             refs << "#{name} #{id}"
127.             already[name] = true
```

lib/grit/git-ruby.rb

```
155.         files.each do |ref|
156.           next if !File.file?(ref)
157.
158.           id = File.read(ref).chomp
159.           name = ref.sub("#{prefix}/", '')
160.
161.           if !already[name]
162.             refs << "#{name} #{id}"
163.             already[name] = true
```

Obr. 6.26: Identický zduplikovaný kód nájdený v projekte *Grit*.

Bez ohľadu na vstupné poradie a bez ohľadu na číslo iterácie sú *skutočné duplikáty odhalené vždy rovnako a správne*. Rozdiely vo výsledkoch sú spôsobené len takým kódom, ktorý v skutočnosti nie je zduplikovaný.

6.6 Krátky pohľad na duplikáty v projekte Grit

Poďme sa pozrieť na ukážky duplikátov, ktoré náš nástroj odhalil v poslednej verzii projektu *Grit*. Viac príkladov sa nachádza v prílohe D a kompletný výstup je dostupný na DVD v prílohe E.

Väčšinu výsledkov sme po manuálnom prezretí považovali za bezchybné. Jeden zo správne odhalených duplikátov uvádzame na obrázku 6.26. Fragmenty sa nachádzajú v rovnakom súbore, je medzi nimi necelých 30 riadkov.

Podarilo sa nám takisto nájsť duplikát, ktorého fragmenty sa líšia len hodnotou číselnej konštanty. Nachádza sa na obrázku 6.27. Podľa nášho názoru ide s veľkou pravdepodobnosťou o typický príklad chyby spôsobenej zduplikovaným kódom. Konštanta by na obidvoch riadkoch zrejme mala byť rovnaká. Kód však nepoznáme a preto je možné, že čísla sú odlišné zámerne.

lib/grit/git-ruby/internal/loose.rb

```
32.           path = @directory + '/' + sha1[0...2] + '/' + sha1[2..39]
```

lib/grit/git-ruby/internal/loose.rb

```
97.           path = @directory+'/'+sha1[0...2]+'/'+sha1[2..40]
```

Obr. 6.27: Zduplikovaný kód, ktorý možno obsahuje chybu, kvôli rozdielnej hodnote číselnej konštanty.

lib/grit/git-ruby/internal/pack.rb

```
65.     raise PackFormatError, "pack #@name has unknown pack file version #{ver}"
```

lib/grit/git-ruby/internal/pack.rb

```
135.    raise PackFormatError, "pack #@name has discontinuous index #{i}"
```

Obr. 6.28: Zduplikovaný kód, ktorý nemá zmysel refaktorovať.

lib/grit/git-ruby/repository.rb

```
573.        get_object_by_sha1(tree_sha).entry.each do |e|
574.          looking_for << File.join('.', e.name)
```

lib/grit/git.rb

```
356.        raise CommandFailed.new(argv.join(' '), status.exitstatus, process.err)
```

Obr. 6.29: Výsledok, ktorý neobsahuje zduplikovaný kód.

Niektoré odhalené duplikáty sa nedajú refaktorovať. Príklad sa nachádza na obrázku 6.28. Nakoniec sa vo výsledkoch objavili aj také fragmenty, ktoré vôbec neobsahujú zduplikovaný kód – obrázok 6.29.

6.7 Zhrnutie

Na záver si zosumarizujme, čo sme vďaka vykonaným experimentom zistili:

- navrhnutá metóda je určite vhodná na odhaľovanie zduplikovaného kódu,
- vybrali sme takú kombináciu vstupných parametrov, ktorá dosahuje najlepšie výsledky a umožňuje rýchle spracovanie zdrojových kódov,
- na vlastnej dátovej vzorke naša metóda fungovala lepšie, ako existujúce nástroje; pre vykonanie viac reprezentatívnych experimentov by sme potrebovali väčšiu (a zrejme nie nami vytvorenú) dátovú vzorku,
- pri inkrementálnom spracovaní dochádza k postupnému zväčšovaniu indexu duplikátov, ostávajú v ňom zabudnuté fragmenty kódu; to je pravdepodobne spôsobené implementačnou chybou,
- okrem zabudnutého kódu pri inkrementálnom spracovaní nedochádza ku žiadnej inej strate kvality indexu alebo ku zhoršovaniu výsledkov,
- rozdiely v duplikátoch odhalených jednorazovým a inkrementálnym spracovaním sme síce odhalili, mali však približne konštantnú veľkosť (vzhľadom na celkové množstvo kódu); predpokladáme, že skutočné duplikáty boli odhalené rovnako a rozdiely spôsobili len tzv. *false positives*.

7

Záver

Cieľom tejto práce bolo navrhnúť spôsob, akým sa zo zdrojových kódov softvérových systémov dajú získať informácie o tom, ktoré ich časti s veľkou pravdepodobnosťou obsahujú chyby. Zamerali sme sa na zduplikovaný kód. Najprv sme podrobne analyzovali aktuálny stav v oblasti jeho automatického vyhľadávania. Zistili sme, že nedávny výskum označuje agresívne refaktorovanie zduplikovaných častí kódov nielen za zbytočné, ale niekedy aj nežiadúce. Vhodnejším riešením vyzerá byť aktívne sledovanie duplikátov. Nástroje, ktoré by takéto sledovanie umožňovali, však neexistujú.

V práci sme navrhli metódu, ktorá slúži na sledovanie vývoja zduplikovaného kódu. Metóda mala využívať abstraktné syntaktické stromy. Okrem toho sme identifikovali 4 ďalšie hlavné požiadavky: (1) musí odhaľovať aj čiastočne modifikovaný zduplikovaný kód, (2) musí byť rýchla, aby sa výsledky mohli vývojárom prezentovať počas toho, ako na kóde pracujú, (3) musí vedieť duplikáty rýchlo aktualizovať po vykonaní (malých) zmien v kóde a (4) musí mať nízke pamäťové nároky, aby sa vývojárom oplatilo nástroj používať aj vo svojich počítačoch.

Problém sme rozdelili na viacero malých častí. Tak malých a dostatočne všeobecných, že sme na ich riešenie boli následne schopní nájsť už existujúce algoritmy alebo dátové štruktúry, prípadne navrhnúť vlastné. Počas realizácie riešenia sme navyše navrhli niekoľko vylepšení:

- algoritmus, ktorý z abstraktných syntaktických stromov vytvorí množstvo malých, horizontálne aj vertikálne sa prekrývajúcich podstromov,
- pri vytváraní charakteristických vektorov pre podstromy sme použili len 1 a 2-cesty, ktoré ich štruktúru odrážajú dostatočne dobre,
- počet dimenzií charakteristických vektorov výrazne redukujeme s využitím metódy náhodných projekcií, resp. lokálne senzitívneho hašovania,
- podobnosť vektorov sme navrhli merať manhattanskou vzdialenosťou,
- vytváranie skupín duplikátov sme navrhli vykonávať klastrovaním,

- klastrovací algoritmus *Birch* sme modifikovali tak, aby si pre všetky vložené body, resp. vektory, mohol pamätať aj ďalšie ľubovoľné dáta,
- ďalej sme *Birch* pozmenili tak, aby aj pri nevhodnom poradí vstupných dát klastre nikdy neobsahovali také body, ktoré by do nich nemali patriť,
- navrhli sme spôsob, akým je možné automaticky vyhodnotiť kvalitu vyhľadávania duplikátov (medzi viacerými rôznymi nástrojmi, ale aj medzi rôznymi spusteniami rovnakého nástroja),
- dokázali sme, že navrhnuté inkrementálne spracovávanie zdrojových kódov odhalí aj v reálnych softvérových projektoch rovnaké duplikáty, ako nezávislé jednorazové spracovania,
- na reálnych projektoch sme ukázali, že ani po spracovaní mnohých zmien, resp. revízií, presnosť a pokrytie odhaľovania duplikátov neklesajú.

Ďalšia práca. Napriek tomu, že sme na projekte vykonali množstvo práce, stále ešte ostáva vyriešiť niekoľko otvorených, resp. odložených problémov. Všetky boli opísané v príslušných častiach návrhu. Najdôležitejšie z nich sú:

- sofistikovanejšie predspracovanie abstraktných syntaktických stromov, s cieľom zvýšiť presnosť odhaľovania duplikátov,
- analýza priestoru, v ktorom sa nachádzajú charakteristické vektory s cieľom vykonávať presnejšiu redukciu dimenzie,
- aktualizácie existujúceho kódu v indexe vykonávať ako atomické operácie, aby sme mali informáciu o tom, že fragment bol zmenený,
- záverečná kontrola skupín s duplikátmi pomalým a presnejším algoritmom, opäť s cieľom zvýšiť presnosť výsledkov,
- sofistikovanejšie záverečné spracovanie výsledkov, ktoré *správne zlúči* prekrývajúce sa skupiny duplikátov, alebo skryje také výsledky, ktoré sa napríklad nedajú refaktorovať,
- integrácia s verziovacími systémami alebo vývojovými prostrediami.

A

Použité algoritmy a dátové štruktúry

A.1 Lokálne senzitívne hašovanie

Bežne sa stretávame s dátami, ktorých jednotlivé prvky sú reprezentované ako vektory. Podobnosť takýchto prvkov je následne definovaná ako vzdialenosť ich vektorov. Veľmi častou úlohou, ktorú je potrebné s takýmito dátami riešiť, je vyhľadávanie podobných položiek. Datar et al. [5] navrhli jedno možné riešenie tohto problému a nazvali ho lokálne senzitívne hašovanie. Len pre zopakovanie, klasické hašovacie algoritmy fungujú nasledovne:

- pre rovnaké dáta určite vypočítajú rovnakú zahašovanú hodnotu,
- pre čo i len trochu odlišné dáta vrátia úplne rozdielne hodnoty.

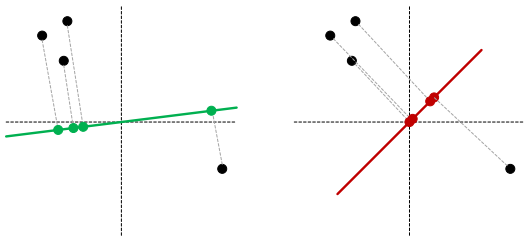
Lokálne senzitívne hašovanie sa od klasického hašovania odlišuje v druhom spomínanom bode. Pre dáta, ktoré sú podobné, zaručene vypočíta aj podobné zahašované hodnoty.

Lokálne senzitívne hašovanie patrí medzi algoritmy, ktorých výstup závisí na náhodných vstupných hodnotách. Takéto algoritmy síce nezaručujú najlepší možný výsledok v každom prípade, ale s vysokou pravdepodobnosťou vrátia taký, ktorý je veľmi blízky najlepšiemu. Túto pravdepodobnosť je možné dokonca zvýšiť za cenu vyššej výpočtovej náročnosti [29].

Jadrom metódy je skalárny súčin dvoch vektorov:

$$h(\vec{v}) = \vec{v} \cdot \vec{x}$$

kde $\vec{v}$ je bod v priestore, ktorý sa má zahašovať a $\vec{x}$ je vektor, ktorého položky sú vybraté náhodne. Vstupný bod takouto operáciou zredukujeme na jedno reálne číslo. Pre viacero bodov takýmto spôsobom vytvoríme viacero reálnych čísel. Zahašované hodnoty si preto môžeme predstaviť ako body umiestnené na priamke, ktorá je definovaná vstupným náhodným vektorom $\vec{x}$. Na obrázku A.1 vľavo je takáto operácia ilustrovaná pre 2-rozmerný priestor. Je ľahké si všimnúť, že ak boli body blízko pri sebe v pôvodnom priestore, sú blízko pri sebe aj po premietnutí na priamku.



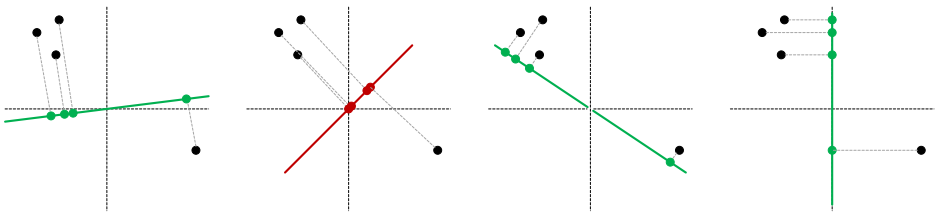
Obr. A.1: Ukážka princípu lokálne senzitívneho hašovania bodov v 2-rozmernom priestore. Vľavo je znázornený vhodne zvolený náhodný vektor, vpravo je znázornené hašovanie s použitím nevhodného náhodného vektora.

Môže však nastať aj situácia ilustrovaná na obrázku A.1 vpravo. Ak náhodný vektor $\vec{x}$ zvolíme nesprávne, vytvorené haše môžu byť podobné napriek tomu, že sa v pôvodnom priestore nachádzali ďaleko od seba. Následky, ktoré vzniknú použitím takéhoto nesprávne zvoleného náhodného vektora, sa dajú jednoduchým spôsobom zmierniť tak, že pre každý bod vykonáme viacero rôznych hašovaní (obrázok A.2). Pri každom treba použiť iný náhodný vektor $\vec{x}$.

Lokálne senzitívne hašovanie sa používa aj v nasledujúcom tvare [5, 29]:

$$h(\vec{v}) = \left\lfloor \frac{\vec{v} \cdot \vec{x} + b}{w} \right\rfloor$$

čo znamená, že priamku, na ktorej sú zahašované body umiestnené, rozdelíme na intervaly so šírkou w. Všetky body, ktoré patria do jedného intervalu, budú mať rovnakú zahašovanú hodnotu a mali by si byť navzájom podobné. Jeden interval obsahuje len veľmi malý počet prvkov, v porovnaní so všetkými dátami. Vyhľadávanie najviac podobného prvku v intervale sa vďaka tomu dá vykonať veľmi rýchlo. To aj v prípade, ak je nevyhnutné použiť pomalé, alebo komplikované porovnávanie prvkov.



Obr. A.2: Následky, ktoré môžu vzniknúť pri použití nesprávne zvoleného náhodného vektora je možné minimalizovať takým spôsobom, že vykonáme viacero rôznych lokálne senzitívnych hašovaní.

Dôležité je aj vytváranie náhodného vektora $\vec{x}$. Jeho prvky musia byť zvolené z niektorého z tzv. stabilných rozdelení: (1) Cauchy-ho ak sa zaujímame o manhattanskú vzdialenosť medzi hašovanými dátami alebo (2) Gauss-ovho pre euklidovskú vzdialenosť. Vysvetlenie tejto požiadavky a teóriu skrývajúcu sa za ním opisujú Datar et al. [5].

A.2 Klastrovací algoritmus Birch

Birch je metóda klastrovania, ktorú navrhli Zhang et al. [32]. Hlavným cieľom pri jej návrhu bolo vytvoriť klastrovací algoritmus pre veľké objemy dát. Birch vyžaduje len jeden prechod cez klastrované dáta – jeho časová zložitosť je z pohľadu potrebných vstupno-výstupných operácií lineárna.

Autori uvádzajú štyri výhody algoritmu Birch v porovnaní s inými klastrovacími metódami:

1. každé rozhodnutie pri klastrovaní dokáže Birch vykonať bez toho, aby musel prezrieť všetky dáta alebo klastre,

2. husté oblasti rozdelí do malého počtu klastrov, pričom v každom bude veľa bodov, riedke oblasti do veľkého počtu klastrov s malým množstvom bodov v každom z nich,

3. pri klastrovaní využíva vyvážený strom, v ktorom si nemusí pamätať body, ktoré do neho boli vložené a preto nespotrebuje veľa pamäte,

4. ak sa vynechajú jeho voliteľné časti, Birch bude kompletne inkrementálny a nikdy nebude potrebovať všetky svoje dáta v pamäti naraz.

Jadrom algoritmu Birch sú tzv. charakteristiky klastrov (angl. *clustering features*) a z nich vytvorený klastrovací strom (angl. *CF tree*).

Charakteristika klastra. Charakteristika klastra je definovaná ako trojica $(N, \vec{LS}, SS)$, ktorá sumarizuje všetky potrebné informácie o danom klastri. Každá trojica pozostáva z: N – počet prvkov v klastri, $\vec{LS}$ – lineárny súčet prvkov v klastri, t.j. $\sum_{i=1}^{N} \vec{X}_i$ a SS – kvadratický súčet prvkov, t.j. $\sum_{i=1}^{N} \vec{X}_i^{\,2}$. Takisto platí, že charakteristika klastra, ktorý vznikne spojením dvoch existujúcich klastrov, sa vypočíta ako $(N_1 + N_2, \vec{LS}_1 + \vec{LS}_2, SS_1 + SS_2)$ [32].

Takéto charakteristiky úplne stačia na to, aby bolo kedykoľvek možné vypočítať až 5 rôznych druhov vzdialeností medzi jednotlivými klastrami. Pre nás je najzaujímavejšou manhattanská (a euklidovská) vzdialenosť centroidov.

Klastrovací strom. Algoritmus Birch si interne vytvára vyvážený strom, ktorý má dva parametre: b – faktor vetvenia a t – maximálnu veľkosť jedného klastra. Vrcholy v tomto strome sa delia na vnútorné vrcholy a listy [32].

Každý vnútorný vrchol obsahuje svoju charakteristiku klastra a nachádza sa pod ním maximálne b detských vrcholov. Vnútorný vrchol reprezentuje jeden veľký klaster vytvorený spojením všetkých svojich podklastrov.

Každý list klastrovacieho stromu taktiež obsahuje svoju charakteristiku klastra. Rozdiel oproti vnútorným vrcholom spočíva v tom, že listy reprezentujú najmenšie (t.j. výsledné) klastre. To znamená, že môžu v sebe obsahovať neobmedzený počet bodov, pokiaľ budú spĺňať podmienku, že priemer klastra bude menší alebo rovný maximálnemu možnému priemeru klastrov t.

Jednou z výhod takéhoto klastrovacieho stromu je, že si nemusí pamätať všetky body, ktoré do neho boli vložené. Namiesto toho si pamätá len charakteristiky pre jednotlivé klastre.

Klastrovanie jedného bodu, resp. vkladanie nového bodu do stromu sa skladá z nasledujúcich krokov [32]:

1. *výber listu* – od koreňa stromu rekurzívne zostupujeme nižšie, vždy do takého vrcholu, ku ktorému je vkladaný bod najbližšie,

2. *aktualizácia listu* – keď sa dostaneme ku listom, vyberie sa taký, ktorý je ku vkladanému bodu najbližšie; následne otestujeme, či list môže vkladaný bod absorbovať (listy, resp. klastre majú maximálnu veľkosť t):

   - ak áno, vložíme bod do listu, resp. klastra tak, že aktualizujeme charakteristiku príslušného klastra,
   - ak nie, vytvoríme pre vkladaný bod úplne nový list,

3. *aktualizácia cesty k listu* – každému vnútornému vrcholu, cez ktorý sme prešli po ceste k listu, treba:

   - aktualizovať jeho charakteristiku klastra,
   - vnútorné vrcholy obsahujú maximálne b potomkov a táto podmienka mohla byť vložením nového listu porušená – v takom prípade treba vnútorný vrchol rozdeliť na dva nové (vybrať dvoch najvzdialenejších potomkov a ostatných rozdeliť podľa toho, ku ktorému z nich sú bližšie); delenie môže byť potrebné vykonať na všetkých úrovniach stromu, pri delení koreňa dôjde k zvýšeniu hĺbky.

Ďalšie implementačné detaily, obmedzenia, ale aj výsledky testovacích experimentov autori uvádzajú v [32].

B

Technická dokumentácia

B.1 Ukážky abstraktných syntaktických stromov

V prílohe uvádzame ukážky abstraktných syntaktických stromov, ktoré vytvoria nástroje *RubyParser* a *Ripper* z identického zdrojového kódu.

Použitý zdrojový kód. Obidva uvedené stromy boli vytvorené z textového zápisu kódu, ktorý je uvedený vo výpise B.1.

Algoritmus B.1 Zdrojový kód, z ktorého boli stromy vytvorené

```
def assign_tfidf(tfidfs)

  raise RuntimeError.new('Chunk already ... ') unless terms.empty?

  tfidfs.first(10).each do |tfidf|
    terms.create(:word => tfidf[:word], :tfidf => tfidf[:tfidf])
  end

end
```

RubyParser. Ako sme už v práci uviedli, knižnica RubyParser vytvára veľmi kompaktné a prehľadné abstraktné syntaktické stromy.

```
[:defn,
 :assign_tfidf,
 [:args, :tfidfs],
 [:scope,
  [:block,
   [:if,
    [:call, [:call, nil, :terms, [:arglist]], :empty?, [:arglist]],
    nil,
    [:call,
     nil,
     :raise,
     [:arglist,
      [:call,
```

```
      [:const, :RuntimeError],
      :new,
      [:arglist, [:str, "Chunk already does have terms assigned"]]]]]],
  [:iter,
   [:call,
    [:call, [:lvar, :tfidfs], :first, [:arglist, [:lit, 10]]],
    :each,
    [:arglist]],
   [:lasgn, :tfidf],
   [:call,
    [:call, nil, :terms, [:arglist]],
    :create,
    [:arglist,
     [:hash,
      [:lit, :word],
      [:call, [:lvar, :tfidf], :[], [:arglist, [:lit, :word]]],
      [:lit, :tfidf],
      [:call,
       [:lvar, :tfidf],
       :[],
       [:arglist, [:lit, :tfidf]]]]]]]]]
```

Ripper. Abstraktný syntaktický strom vytvorený knižnicou Ripper je výrazne väčší, obsahuje viac vrcholov a je ťažšie čitateľný.

```
[:def,
 [:@ident, "assign_tfidf", [203, 6]],
 [:paren,
  [:params, [[:@ident, "tfidfs", [203, 19]]], nil, nil, nil, nil]],
 [:bodystmt,
  [:stmts_add,
   [:stmts_add,
    [:stmts_new],
    [:unless_mod,
     [:call,
      [:vcall, [:@ident, "terms", [205, 76]]],
      :".",
      [:@ident, "empty?", [205, 82]]],
     [:command,
      [:@ident, "raise", [205, 4]],
      [:args_add_block,
       [:args_add,
        [:args_new],
        [:method_add_arg,
         [:call,
          [:var_ref, [:@const, "RuntimeError", [205, 10]]],
          :".",
          [:@ident, "new", [205, 23]]],
         [:arg_paren,
          [:args_add_block,
           [:args_add,
            [:args_new],
            [:string_literal,
             [:string_add,
              [:string_content],
              [:@tstring_content,
               "Chunk already does have terms assigned",
               [205, 28]]]]],
```

```
          false]]]],
       false]]]],
   [:method_add_block,
    [:call,
     [:method_add_arg,
      [:call,
       [:var_ref, [:@ident, "tfidfs", [207, 4]]],
       :".",
       [:@ident, "first", [207, 11]]],
      [:arg_paren,
       [:args_add_block,
        [:args_add, [:args_new], [:@int, "10", [207, 17]]],
        false]]],
     :".",
     [:@ident, "each", [207, 21]]],
    [:do_block,
     [:block_var,
      [:params, [[:@ident, "tfidf", [207, 30]]], nil, nil, nil, nil],
      nil],
     [:stmts_add,
      [:stmts_new],
      [:method_add_arg,
       [:call,
        [:vcall, [:@ident, "terms", [208, 6]]],
        :".",
        [:@ident, "create", [208, 12]]],
       [:arg_paren,
        [:args_add_block,
         [:args_add,
          [:args_new],
          [:bare_assoc_hash,
           [[:assoc_new,
             [:symbol_literal,
              [:symbol, [:@ident, "word", [208, 20]]]],
             [:aref,
              [:var_ref, [:@ident, "tfidf", [208, 28]]],
              [:args_add_block,
               [:args_add,
                [:args_new],
                [:symbol_literal,
                 [:symbol, [:@ident, "word", [208, 35]]]]],
               false]]],
            [:assoc_new,
             [:symbol_literal,
              [:symbol, [:@ident, "tfidf", [208, 43]]]],
             [:aref,
              [:var_ref, [:@ident, "tfidf", [208, 52]]],
              [:args_add_block,
               [:args_add,
                [:args_new],
                [:symbol_literal,
                 [:symbol, [:@ident, "tfidf", [208, 59]]]]],
               false]]]]]],
        false]]]]]]],
 nil,
 nil,
 nil]]]
```

B.2 Rozdeľovanie stromu na podstromy

Na rozdeľovanie abstraktných syntaktických stromov na menšie podstromy sme navrhli algoritmus, ktorého opis sa nachádza v kapitole 5.2.1. V tejto prílohe ilustrujeme fungovanie tohto algoritmu na vzorovom strome. Každý krok spracovania je znázornený na obrázkoch B.1 a B.2, k jednotlivým obrázkom pridávame doplňujúci komentár.

V príklade používame maximálnu hĺbku $h_{max} = 3$ a minimálny počet vrcholov $n_{min} = 5$. Celý strom bude spracovaný v 16 krokoch:

1. začíname od koreňa stromu, vyznačený podstrom má hĺbku 3 a má viac ako 5 vrcholov, preto ho akceptujeme,

2. algoritmus pokračuje vrcholom, ktorý je prvým dieťaťom koreňa, aj tento podstrom bude akceptovaný,

3. vytvoríme podstrom z ďalšieho vrcholu, tiež spĺňa podmienky a akceptujeme ho; môžeme si všimnúť, že vrcholy sú spracovávané v rovnakom poradí, ako keby sme strom prehľadávali do hĺbky,

4. vo štvrtom kroku prvýkrát narazíme na vrchol, z ktorého nie sme schopní vytvoriť podstrom, ktorý by obsahoval aspoň 5 vrcholov,

5. ku príliš malému podstromu zo 4. kroku skúsime pridať podstrom z jeho pravého súrodenca; dokopy majú 5 vrcholov a podstrom akceptujeme,

6. v šiestom kroku spracujeme pravého súrodenca samostatne; podstrom má len 4 vrcholy, preto ho preskočíme; keby sme nepoužívali spájanie súrodencov, podstromy z krokov 4 a 6 by vôbec neboli zaindexované,

7. nasledujúci vrchol nemá žiadne deti, nemôžeme z neho teda vytvoriť žiaden podstrom; nemá ani žiadneho ďalšieho pravého súrodenca,

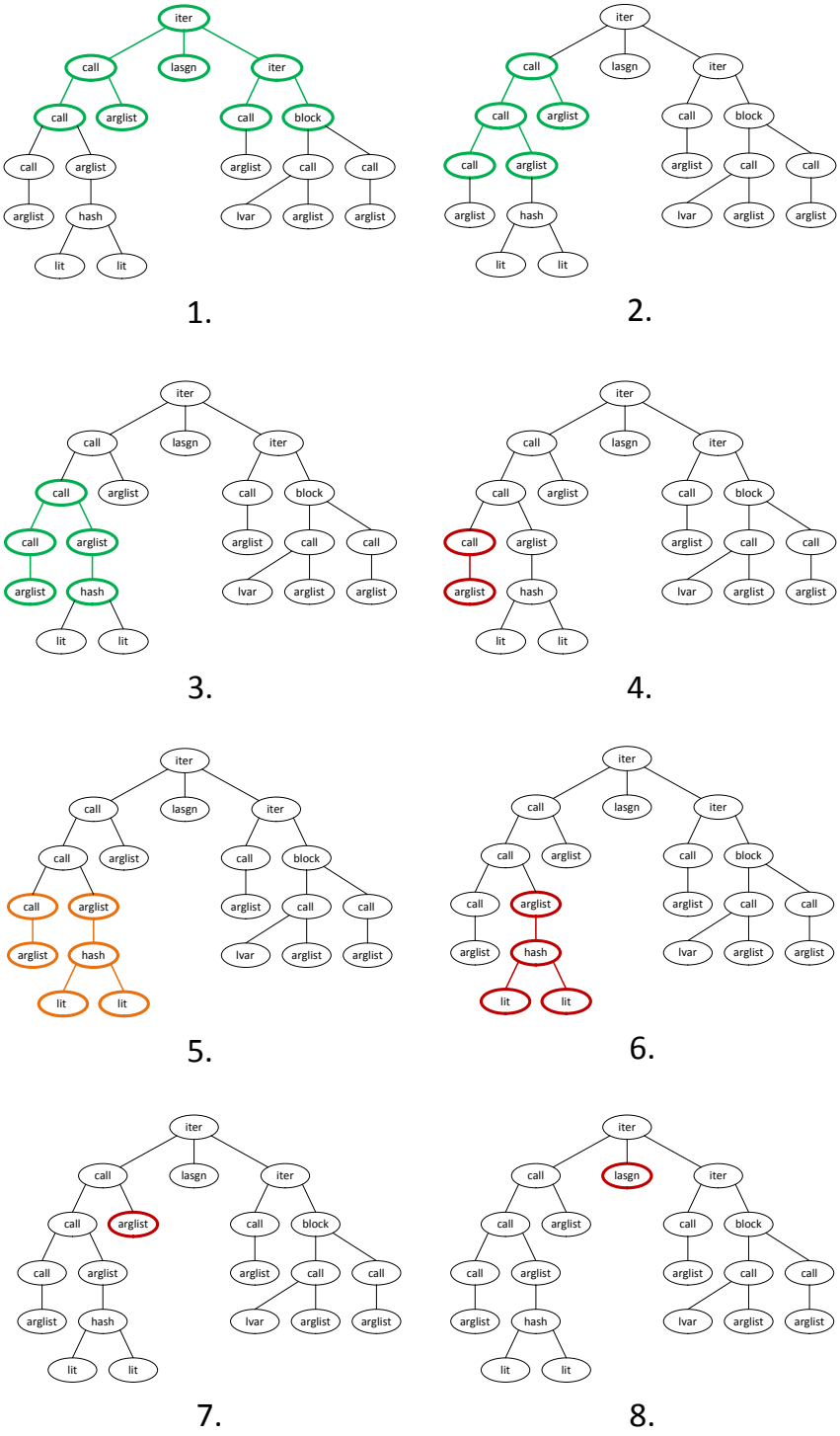
8. rovnako ako v predchádzajúcom kroku, samostatný vrchol preskočíme,

9. tentokrát však samostatný vrchol z predchádzajúceho kroku má pravého súrodenca, ich podstromy spojíme do jedného,

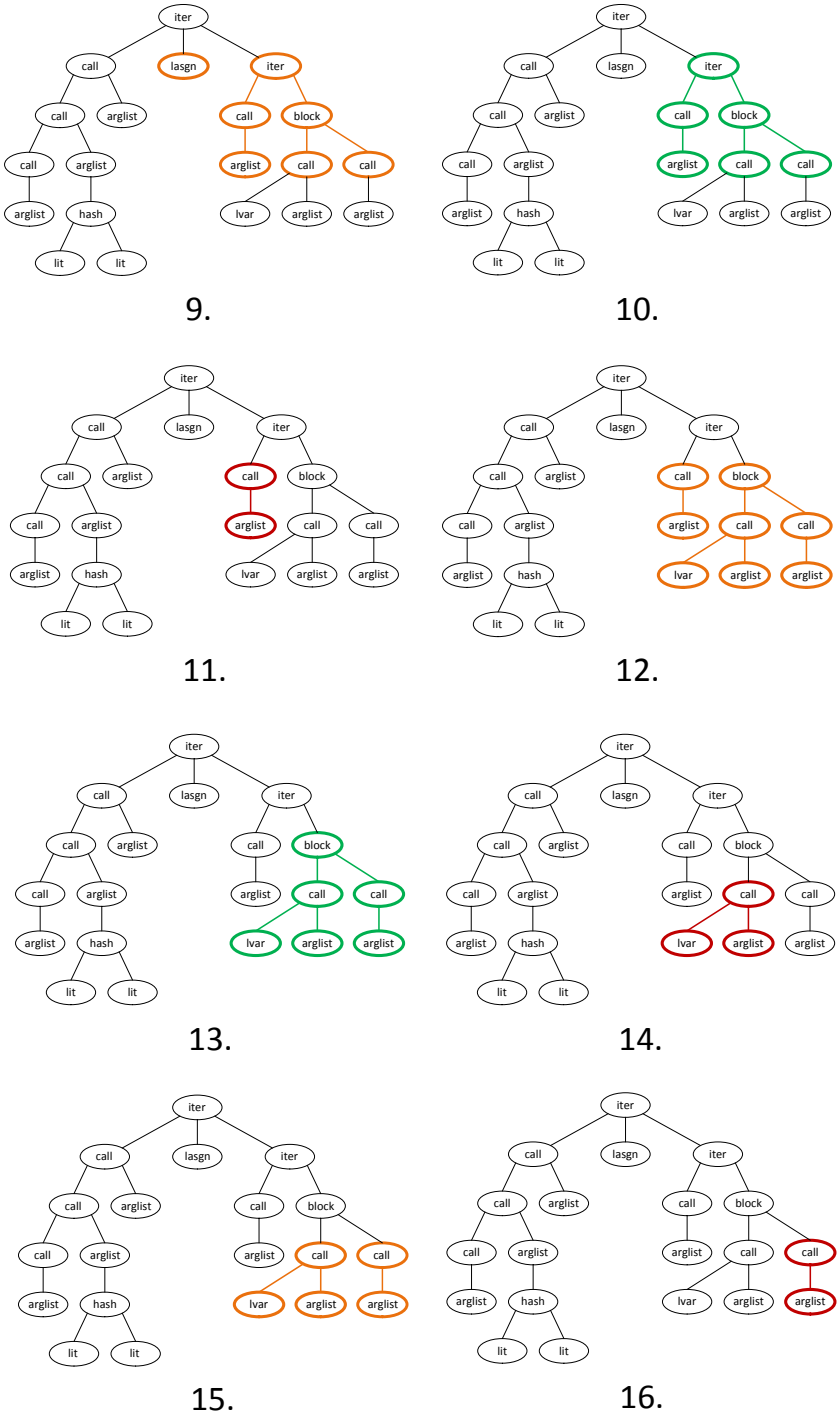
10. pravého súrodenca z prechádzajúceho kroku spracujeme aj samostatne, pretože podstrom pod ním je dostatočne veľký,

11. podstrom s 2 vrcholmi nie je dostatočne veľký,

12. k predchádzajúcim 2 vrcholom skúsime pridať podstrom pravého súrodenca, spolu majú dostatočnú veľkosť,



Obr. B.1: Prvých 8 krokov algoritmu na rozdeľovanie stromov.



Obr. B.2: Zvyšných 8 krokov algoritmu na rozdeľovanie stromov.

13. pravý súrodenec z prechdázajúceho kroku má opäť aj sám dostatočnú veľkosť, spravíme z neho úplne samostatný fragment,

14. už sme skoro na dne stromu, z 3 vrcholov sa nám nepodarí vytvoriť dostatočne veľký podstrom,

15. po pridaní pravého súrodenca ale 5 vrcholov nazbierame a preto aj z týchto listov ešte môžeme vytvoriť samostatný podstrom,

16. posledné 2 vrcholy nakoniec preskočíme; nebyť spájania podstromov, ani fragmenty z bodov 14 a 16 by neboli zaindexované.

Celkovo sme teda relatívne malý vzorový strom rozdelili až na 9 vertikálne aj horizontálne sa prekrývajúcich podstromov.

B.3 Formát použitý na serializáciu Birch stromu

V tejto prílohe opisujeme formát, v ktorom sa index duplikátov (resp. *Birch* strom) ukladá na disk. Aby bol vytvorený súbor čo najmenší, formát je binárny. Bajty sú ukladané v poradí *little endian*.

Jeho štruktúra je rekurzívna. To znamená, že časť súboru, ktorá zodpovedá niektorému vrcholu, v sebe obsahuje informácie o všetkých ostatných vrcholoch, ktoré sa nachádzajú pod ním.

Konfigurácia. Na začiatku súboru sa nachádza konfigurácia *Birch* stromu. Jej štruktúra je znázornená na obrázku B.3. Skladá sa z:

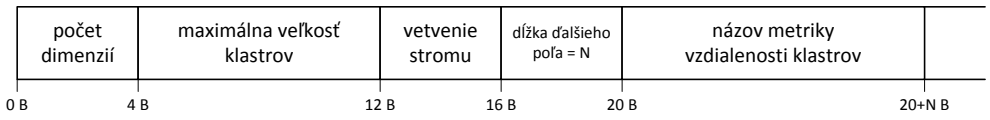
1. počet dimenzií D (4B, kladné celé číslo, tzv. *unsigned integer*) – do *Birch* stromu sa vkladajú body, ktoré pochádzajú z D-rozmerného priestoru, každý bod má teda D dimenzií,

2. maximálna veľkosť klastrov (8B, reálne číslo, tzv. *double precision*) – definuje, aké veľké môžu byť listy, resp. klastre,

3. vetvenie stromu (4B, kladné celé číslo, tzv. *unsigned integer*) – hodnota vetvenia *Birch* stromu,

4. dĺžka N poľa s názvom metriky vzdialenosti klastrov (4B, kladné celé číslo, tzv. *unsigned integer*) – definuje, koľko B zaberá reťazec znakov, ktorý nasleduje v ďalšom poli,

5. názov metriky vzdialenosti klastrov (N bajtov, reťazec znakov v UTF-8) – vždy má hodnotu `manhattan`, pretože inú metriku vzdialenosti zatiaľ naša implementácia nepodporuje.

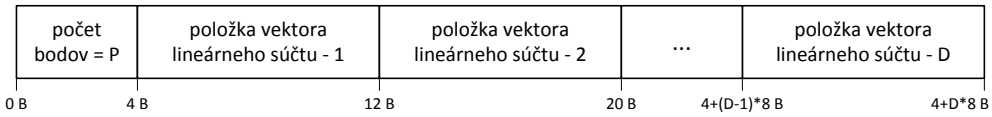


Obr. B.3: Štruktúra serializovanej konfigurácie; na začiatku súboru.

Vrcholy. Hneď za konfiguráciou nasleduje serializovaný koreň stromu. Koreň je vždy vnútorný vrchol (nikdy to nemôže byť list). Každý vrchol stromu má však v serializovanej podobe rovnaký začiatok (obrázok B.4) – bez ohľadu na to, či je to vnútorný vrchol alebo list. Štruktúra je nasledujúca:

1. počet bodov P (4B, kladné celé číslo, tzv. *unsigned integer*) – určuje koľko bodov, resp. fragmentov kódu sa nachádza v celom podstrome pod aktuálnym vrcholom,

2. serializovaný linerány súčet – každý vrchol si udržiava charakteristiku klastra (príloha A.2), ktorej súčasťou je lineárny súčet všetkých bodov z celého podstromu pod daným vrcholom; body majú D dimenzií a vektor lineárneho súčtu preto pozostáva z D reálnych čísel (každé zaberá 8B, tzv. *double precision*); kvadratický súčet pri používaní manhattanskej vzdialenosti nepotrebujeme a nemá zmysel ho serializovať.



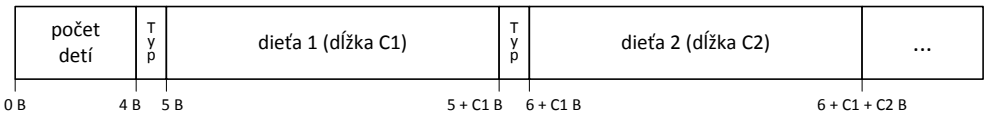
Obr. B.4: Spoločný začiatok vnútorných vrcholov aj listov.

Vnútorné vrcholy. Pri serializovaní vnútorných vrcholov treba ešte uložiť zoznam ich detí. Používame na to nasledujúcu štruktúru (obrázok B.5):

1. počet detí (4B, kladné celé číslo, tzv. *unsigned integer*) – koľko detí obsahuje tento vnútorný vrchol, resp. koľko detí sa má ďalej načítať,

2. typ dieťaťa (1B, znak) – má len 2 možné hodnoty, znak `L` (nasleduje serializovaný list) alebo `N` (nasleduje ďalší vnútorný vrchol),

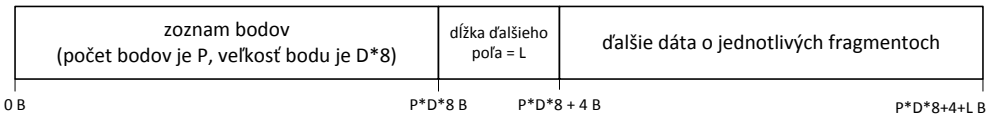
3. serializované dieťa – na tomto mieste sa nachádza ďalší rekurzívne serializovaný vrchol – buď vnútorný, alebo list; dĺžku si vďaka rekurzívnemu spracovaniu netreba explicitne ukladať.



Obr. B.5: Pokračovanie serializovaného vnútorného vrcholu.

Listy. V každom liste si navyše pre každý fragment kódu pamätáme jeho bod, názov súboru a riadky, na ktorých sa v súbore daný fragment nachádza. Serializovaná štruktúra je znázornená na obrázku B.6 a pozostáva z:

1. zoznam bodov, ktoré list obsahuje; počet bodov je P (viď. spoločný začiatok vrcholov), každý bod má D dimenzií a tvorí ho teda D serializovaných reálnych čísel (8B, tzv. *double precision*),
2. dĺžka L serializovaných dát o fragmentoch (4B, kladné celé číslo, tzv. *unsigned integer*) – určuje dĺžku nasledujúceho poľa,
3. dáta o fragmentoch (L bajtov, dáta serializované knižnicou *Marshal*) – zoznam dátových štruktúr v takom istom poradí, v akom je uvedený aj zoznam bodov (t.j. i-temu bodu zodpovedajú dáta na i-tom mieste v tomto zozname).



Obr. B.6: Pokračovanie serializovaného listu.

Dáta o jednom fragmente kódu sú tvorené jedným Ruby objektom typu *Hash* a vyzerajú nasledovne:

```
{ :filename => "lib/birch.rb", :lines => [105, 111] }
```

Uvedená štruktúra hovorí o tom, že fragment pochádza zo súboru s názvom `lib/birch.rb` a nachádza sa na riadkoch 105-111. V poslednej položke serializovaného listu *Birch* stromu sa nachádza celý zoznam, resp. pole takýchto dátových štruktúr. Pre jednoduchosť implementácie sme na serializáciu tohto zoznamu použili knižnicu *Marshal*[1], ktorá je štandardnou súčasťou Ruby.

[1] http://ruby-doc.org/core-1.9.3/Marshal.html

C

Používateľská príručka

V tejto prílohe uvádzame inštalačnú a používateľskú príručku. V rámci práce na projekte sme doteraz implementovali:

- všetky časti navrhnutého procesu detekcie duplikátov,
- pomocné nástroje na vykonanie experimentov,
- pomocné nástroje na vyhodnocovanie výsledkov.

V súčasnom stave však nástroju ešte chýba finalizácia, ktorá by ho umožnila jednoduchým spôsobom nasadiť a reálne používať.

Požiadavky. Práca bola implementovaná v jazyku Ruby (verzia 1.9). Na výbere konkrétneho interpretera by nemalo záležať. My sme používali 32-bitový *tcs-ruby 1.9.3p28* v operačnom systéme Windows 7.

Niektoré Ruby knižnice sú implementované v jazyku C a môže sa stať, že nemajú predkompilované verzie pre operačný systém Windows. V takom prípade je potrebné mať nainštalovaný kompilátor *MinGW*. Pre Windows existuje jeho balík nazvaný *RubyInstaller Development Kit*, ktorý sa po nainštalovaní automaticky postará o kompiláciu natívnych Ruby knižníc.

Pri analýze veľkých projektov sme čítali informácie o ich histórii z Git repozitárov. Ďalšou požiadavkou je teda mať nainštalovaný tento verziovací systém. My sme používali verziu 1.8.0.

Ostatné knižnice, na ktorých naša aplikácia závisí, sú:

- **bundler** (1.1.3) – správa závislostí v Ruby projektoch,
- **ruby_parser** (3.1.1) – parsovanie zdrojových kódov v jazyku Ruby na abstraktné syntaktické stromy,
- **grit** (2.5.0) – čítanie a práca s Git repozitármi,
- **yajl-ruby** (1.1.0) – načítavanie a serializácia dát do formátu JSON,
- **rspec** (2.11.0) – testovacia knižnica, implementovali sme s jej využitím tzv. *unit test*-y.

Inštalácia. Pre použitie vytvorenej aplikácie je potrebné nainštalovať všetky vyššie uvedené požiadavky. Nami používané verzie Ruby interpretera, *RubyInstaller Development Kit* balíka a verziovacieho systému Git pre Windows sú priložené na DVD v prílohe E. Po ich inštalácii je pre pohodlné používanie vhodné pridať cestu k Ruby a Git inštaláciám do premennej *PATH*.

Pre správu závislostí a použitých knižníc Ruby projekty často využívajú knižnicu s názvom *Bundler*. Použili sme ju aj my. Nainštalovať sa dá príkazom `gem install bundler`.

Bundler sa následne dokáže automaticky postarať o automatickú inštaláciu ostatných závislostí. V koreňovom adresári so zdrojovými kódmi aplikácie (nachádzajú sa tam súbory nazvané `Gemfile` a `Gemfile.lock`) treba spustiť príkaz `bundle install`.

Inicializácia. Pred použitím aplikácie na zdrojových kódoch nejakého projektu ju treba inicializovať. Podmienky sú, aby (1) projekt na správu verzií používal systém Git a (2) obsahoval zdrojový kód v jazyku Ruby (s príponou `.rb`). Počas inicializácie sa vykonajú nasledujúce operácie:

1. v koreňovom adresári projektu sa vytvorí podadresár slúžiaci na ukladanie všetkých dát a súborov, ktoré sú potrebné pri detekcii duplikátov; adresár má vždy rovnaký názov: `.duplicates`,
2. vygenerujú a uložia sa do súboru (formát JSON) náhodné vektory používané pri redukcii dimenzie; ak chceme odhaľovať zduplikovaný kód medzi rôznymi verziami projektu, treba vždy používať tie isté vektory,
3. vytvorí sa nový a zatiaľ prázdny *Birch* strom a serializuje sa do súboru.

Inicializácia potrebuje len 1 argument a ním je cesta k adresáru s projektom, v ktorom sa zduplikovaný kód bude vyhľadávať. Spustenie bude teda vyzerať napríklad takto:

```
> ruby bin/init.rb projects/grit
```

Nezávislé, jednorazové spracovanie verzií. Po inicializácii by malo nasledovať indexovanie aktuálnej (resp. poslednej) verzie zdrojových kódov. Takýto skript však vytvorený nemáme. Pri experimentoch sme používali podobný skript, ktorý jednorazovo indexoval každú jednu verziu z celej histórie projektu, nezávisle od ostatných. Upraviť ho, aby indexoval len poslednú verziu, by teda bolo triviálne.

Skript navyše neaktualizuje pôvodný súbor s indexom zdrojových kódov, ale vytvorí úplne nový. Pri overovaní riešenia sme totiž potrebovali mať pre každú verziu vytvorený samostatný index. Skript vygenerované súbory automaticky čísluje a dáva im voliteľnú príponu. Akceptuje 3 parametre:

1. cesta k adresáru s projektom; povinná,

2. počet posledných verzií, ktoré sa majú jednorazovo spracovať; nepovinný, predvolená hodnota je 1; pre spracovanie všetkých verzií z histórie sa dá použiť veľmi vysoké číslo, napr. 999999,

3. prípona vytvorených súborov s indexom; nepovinná, predvolená hodnota je `onetime`; vhodné pri vykonávaní viacerých nezávislých spracovaní, ich výstupy budú mať rôzne prípony.

Spustenie môže vyzerať napríklad takto:

```
> ruby bin/onetime.rb projects/grit 10 onetime_2
```

Inkrementálne spracovanie verzií. Poslednou časťou je inkrementálne spracovanie verzií. Pri experimentoch sme používali skript, ktorý spracováva zadaný počet posledných revízií. My sme ho väčšinou používali na spracovanie celej histórie projektu.

Skript na začiatku načíta *Birch* strom zo súboru `.duplicates/birch`. Tento súbor po inicializácii obsahuje prázdny index zdrojových kódov. V prípade spracovávania celej histórie projektu, od začiatku, nevzniká žiaden problém. Ak chceme spracovať len zadaný počet posledných revízií, potrebujeme súbor `.duplicates/birch` ručne nahradiť takým, ktorý obsahuje zaindexované kódy správnej verzie projektu (napr. získanej jednorazovým indexovaním). Aj tento skript by teda bolo triviálne upraviť tak, aby automaticky vedel zistiť, koľko posledných revízií má spracovať.

Pri svojej činnosti skript vygeneruje nový súbor s indexom po každej spracovanej revízii, pre potreby experimentov. Akceptuje 3 rovnaké parametre ako skript na jednorazové spracovanie:

1. cesta k adresáru s projektom; povinná,

2. počet posledných revízií, ktoré sa majú spracovať; nepovinný, ak nie je zadaný, spracuje sa celá história projektu,

3. prípona vytvorených súborov s indexom; nepovinná, predvolená hodnota je `incremental`; vhodné pri vykonávaní viacerých nezávislých spracovaní, ich výstupy budú mať rôzne prípony.

Spustenie môže vyzerať napríklad takto:

```
> ruby bin/incremental.rb projects/grit 10 incremental_3
```

Prezeranie štruktúry stromu v indexe. Jeden z pomocných skriptov slúži na vypísanie štruktúry *Birch* stromu, ktorý je serializovaný v súbore. Potrebuje 2 parametre:

1. cesta k adresáru s projektom; povinná,
2. prípona súboru so stromom, ktorý sa má vypísať; prechádzajúce skripty vytvárajú očíslované súbory s rôznymi príponami, tento skript potrebuje zadať požadované číslo spolu s príponou; parameter je nepovinný, ak nie je zadaný, použije sa prázdna (resp. žiadna) prípona.

Spustenie môže vyzerať napríklad takto:

```
> ruby bin/dump_structure.rb projects/grit 13.onetime
```

Výstup má nasledujúcu rekurzívnu štruktúru:

- vnútorný vrchol obsahuje 3 položky – počet fragmentov kódu, ktoré sa nachádzajú v celom jeho podstrome, počet jeho vlastných detí a zoznam týchto detí: `[fragmenty, deti, [dieťa 1, dieťa 2, ...]]`
- list obsahuje jediný údaj a tým je, koľko fragmentov kódu sa v ňom nachádza: `[fragmenty]`.

Prezeranie nájdených duplikátov. Vytvorili sme aj skript, ktorý umožňuje jednoduchým spôsobom vizualizovať nájdené duplikáty. Vytvorí z nich Javascript súbor, ktorý obsahuje aj samotné fragmenty kódu. Tento súbor potom stačí vložiť do pripravenej HTML šablóny. Následne sa dajú duplikáty prezerať v ľubovoľnom prehliadači. Skript potrebuje 3 parametre:

1. cesta k adresáru s projektom; povinná,
2. SHA identifikátor revízie, z ktorej sa majú fragmenty kódu načítať (pri experimentoch nás nezaujímali vždy len najnovšie verzie projektov),
3. prípona súboru s indexom, ktorého duplikáty sa majú vypísať; aj tento skript potrebuje zadať číslo verzie spolu s príponou; parameter je nepovinný, ak nie je zadaný, použije sa prázdna (resp. žiadna) prípona.

Spustenie môže vyzerať napríklad takto, výsledok sa vypíše na štandardný výstup:

```
> ruby bin/to_js.rb projects/grit 48907bafe588f7d01577acecc49a...
    15.incremental > duplicates.js
```

Do HTML šablóny sa vygenerovaný súbor dá vložiť takýmto riadkom:

```
<script src="duplicates.js"></script>
```

D

Duplikáty v projekte Grit

V tejto prílohe uvádzame ukážku duplikátov, ktoré sme získali s využitím nášho nástroja z poslednej verzie projektu *Grit*. Kompletný výstup analýzy sa nachádza na DVD v prílohe E.

Identické duplikáty. Pozrime sa najprv na ukážky niektorých úplne identických duplikátov. Prvý sa nachádza na obrázku D.1 a obsahuje duplikáty, ktoré pochádzajú z jedného súboru (sú od seba vzdialené 20 riadkov). Identické duplikáty na obrázku D.2 pochádzajú z dvoch rôznych súborov.

Rozdielne identifikátory a konštanty. Ako bolo spomínané v texte práce mnohokrát, pri porovnávaní kódov sa neberú do úvahy názvy identifikátorov. Príklad takého duplikátu sa nachádza na obrázku D.3. Takisto sa ignorujú aj hodnoty konštánt – napríklad čísel a reťazcov. Príklad je na obrázku D.4.

Nezávislé od riadkov kódu. Odhalené duplikáty nie sú závislé od množstva riadkov kódu. V predchádzajúcich príkladoch (napríklad obrázok D.1)

lib/grit/git-ruby/repository.rb

```
236.             if append
237.               mod_tree = []
238.               tree.each do |ent|
239.                 (info, fpath) = ent.split("\t")
240.                 mod_tree << [info, File.join(append, fpath)].join("\t")
241.               end
242.               mod_tree
```

lib/grit/git-ruby/repository.rb

```
263.           if append
264.             mod_tree = []
265.             tree.each do |ent|
266.               (info, fpath) = ent.split("\t")
267.               mod_tree << [info, File.join(append, fpath)].join("\t")
268.             end
269.             mod_tree
```

Obr. D.1: Identické duplikáty nachádzajúce sa v rovnakom súbore.

lib/grit/tree.rb

```
53.     def create_initialize(repo, atts)
54.       @repo = repo
55.
56.       atts.each do |k, v|
57.         instance_variable_set("@#{k}", v)
58.       end
59.       self
```

lib/grit/commit.rb

```
84.     def create_initialize(repo, atts)
85.       @repo = repo
86.       atts.each do |k, v|
87.         instance_variable_set("@#{k}", v)
88.       end
89.       self
```

Obr. D.2: Identické duplikáty nachádzajúce v rozdielnych súboroch.

lib/grit/git-ruby/repository.rb

```
126.       def in_packs?(sha_hex)
127.         # try packs
128.         packs.each do |pack|
129.           return true if pack[sha_hex]
130.         end
131.         false
```

lib/grit/git-ruby/repository.rb

```
135.       def in_loose?(sha_hex)
136.         loose.each do |lsobj|
137.           return true if lsobj[sha_hex]
138.         end
139.         false
```

Obr. D.3: Duplikáty, ktoré sa líšia v názvoch identifikátorov.

test/test_diff.rb

```
26.     assert_equal 'LICENSE',     diffs[0].a_path
27.     assert_equal 'MIT-LICENSE', diffs[0].b_path
```

test/test_diff.rb

```
33.     assert_equal 'README.md', diffs[1].a_path
34.     assert_equal 'README.md', diffs[1].b_path
```

test/test_diff.rb

```
40.     assert_equal 'Rakefile', diffs[2].a_path
41.     assert_equal 'Rakefile', diffs[2].b_path
```

test/test_diff.rb

```
45.     assert_equal 'PURE_TODO', diffs[3].a_path
46.     assert_equal 'TODO',      diffs[3].b_path
```

Obr. D.4: Duplikáty, ktoré sa líšia v hodnotách konštánt.

lib/grit/git.rb

```
436.        ext_args = args.reject { |a| a.empty? }.map { |a|
                (a == '--' || a[0].chr == '|' || Grit.no_quote) ? a : "\"#{e(a)}\"" }
```

lib/grit/git.rb

```
440.        ext_args = args.reject { |a| a.empty? }.map { |a|
                (a == '--' || a[0].chr == '|' || Grit.no_quote) ? a : "'#{e(a)}'" }
```

Obr. D.5: Duplikát napísaný na 1 riadku obsahuje komplikovaný kód.

sme videli príklady, ktoré siahajú cez viacero riadkov kódu. Najväčší odhalený duplikát, ktorý sme pri manuálnom prehliadaní výsledkov našli, mal 45 riadkov. Na obrázku D.5 sa nachádza duplikát, ktorý je síce napísaný len na 1 riadok kódu, ale obsahuje komplikovanú aplikačnú logiku. Jeden z najmenších duplikátov na aké sme narazili sa nachádza na obrázku D.6. Veľká časť výsledkov bola tvorená práve menšími duplikátmi.

Iné zaujímavé duplikáty. Na obrázku D.7 sa nachádza netriviálna podmienka. Na inom mieste kódu bola táto podmienka napísaná znova, ale v negovanom tvare. Práve takýto kód je ideálny kandidát na nečakané chyby a teda ideálny kandidát na refaktorovanie. Niektoré riadky kódu bývajú zduplikované aj viac ako len 2 až 3-krát. Napríklad 16-krát. Časť z tejto skupiny sa nachádza na obrázku D.8.

lib/grit/status.rb

```
17.       @files.select { |k, f| f.type == 'M' }
```

lib/grit/status.rb

```
21.       @files.select { |k, f| f.type == 'A' }
```

lib/grit/status.rb

```
25.       @files.select { |k, f| f.type == 'D' }
```

Obr. D.6: Jeden z najmenších duplikátov, aké sme vo výsledkoch našli.

lib/grit/git-ruby/repository.rb

```
360.      if (!opts[:max_count] || ((array.size + total_size) < opts[:max_count]))
```

lib/grit/git-ruby/repository.rb

```
367.       if (opts[:max_count] && (array.size + total_size) >= opts[:max_count])
```

Obr. D.7: Duplikát, ktorý má v 2. výskyte znegovanú podmienku.

test/test_merge.rb

```
7.    @r = Repo.new(File.join(File.dirname(__FILE__), *%w[dot_git]), :is_bare => true)
```

test/test_raw.rb

```
7.    @r = Grit::Repo.new(File.join(File.dirname(__FILE__), *%w[dot_git]), :is_bare => true)
```

test/test_rubygit_alt.rb

```
7.    @git1 = Grit::Repo.new(File.join(File.dirname(__FILE__), *%w[dot_git]), :is_bare => true)
```

test/test_rubygit_alt.rb

```
8.    @git2 = Grit::Repo.new(File.join(File.dirname(__FILE__), *%w[dot_git_clone]), :is_bare => true)
```

test/test_rubygit_alt.rb

```
9.    @git3 = Grit::Repo.new(File.join(File.dirname(__FILE__), *%w[dot_git_clone2]), :is_bare => true)
```

test/test_rubygit_iv2.rb

```
7.    @git = Grit::Repo.new(File.join(File.dirname(__FILE__), *%w[dot_git_iv2]), :is_bare => true)
```

test/test_rubygit.rb

```
7.    @git = Git.new(File.join(File.dirname(__FILE__), *%w[dot_git]))
```

test/test_head.rb

```
5.    @r = Repo.new(File.join(File.dirname(__FILE__), *%w[dot_git]), :is_bare => true)
```

test/test_rubygit_index.rb

```
7.    @base_repo = create_temp_repo(File.join(File.dirname(__FILE__), *%w[dot_git_iv2]))
```

test/test_tag.rb

```
5.    @r = Repo.new(File.join(File.dirname(__FILE__), *%w[dot_git]), :is_bare => true)
```

test/test_repo.rb

```
17.   gpath = create_temp_repo(File.join(File.dirname(__FILE__), *%w[dot_git]))
```

Obr. D.8: Až 16-krát zduplikovaný riadok kódu (zobrazených len 11 výskytov).

Duplikáty, ktoré sa nedajú refaktorovať. Niektoré správne odhalené duplikáty sa nemusia dať refaktorovať. Veľmi vhodný príklad je na obrázku D.9. Obidva fragmenty kódu sa nachádzajú v súbore rovno pod sebou. Ide o inicializáciu objektu, ktorý ako 2. parameter prijíma hašovaciu tabuľku s rôznymi atribútmi (v Ruby sa dajú takto zapisovať). Duplikát je technicky správny, pretože prvé dva atribúty sa od druhých dvoch líšia len v konštantách. Refaktorovať by ho bolo podľa nášho názoru zbytočné.

Na obrázku D.10 sa nachádza príklad duplikátu, ktorý by sa v niektorých jazykoch nedal refaktorovať. Ide o vetvy *switch* príkazu, podľa hodnoty premennej sa vytvárajú inštancie rôznych objektov. V Ruby by sa dal takýto kód refaktorovať jednoducho, autor však evidentne uprednostnil čitateľnosť.

lib/grit/blob.rb

```
93.  c = Commit.create(repo, :id => info[:id],
94.                         :author => Actor.from_string(info[:author]),
```

lib/grit/blob.rb

```
95.                         :authored_date => info[:author_date],
96.                         :committer => Actor.from_string(info[:committer]),
```

Obr. D.9: Správne odhalený duplikát, ktorý je ale zbytočné refaktorovať.

lib/grit/tree.rb

```
70.         when "tree"
71.           Tree.create(repo, :id => id, :mode => mode, :name => name)
```

lib/grit/tree.rb

```
72.         when "blob"
73.           Blob.create(repo, :id => id, :mode => mode, :name => name)
```

lib/grit/tree.rb

```
74.         when "link"
75.           Blob.create(repo, :id => id, :mode => mode, :name => name)
```

lib/grit/tree.rb

```
76.         when "commit"
77.           Submodule.create(repo, :id => id, :mode => mode, :name => name)
```

Obr. D.10: Duplikát, ktorý sa v niektorých jazykoch nemusí dať refaktorovať.

lib/grit/git-ruby.rb

```
82.         (sha1, sha2) = string.split('..')
83.         return [rev_parse({}, sha1), rev_parse({}, sha2)]
```

lib/grit/repo.rb

```
353.        shatype, ref = line.split("\t")
354.        sha, type = shatype.split(' ')
355.        [ref, sha, type]
```

lib/grit/repo.rb

```
513.        stuff, path = a.split("\t")
514.        mode, type, sha, size = stuff.split(" ")
```

Obr. D.11: Nesprávne odhalený duplikát, kódy sú jednoznačne odlišné.

lib/grit/git.rb

```
227.     # Gets a patch for a given SHA using `git diff`.
```

lib/grit/git.rb

```
232.     #
```

lib/grit/git.rb

```
235.       options, applies_sha = {}, options if !options.is_a?(Hash)
```

Obr. D.12: Jeden z úplne evidentne nesprávnych výsledkov.

Chybne odhalené duplikáty. Vo výsledkoch sa niekedy objavujú aj chybne odhalené duplikáty. Príklad je uvedený na obrázku D.11.

Niekedy sme narazili na evidentné nezmysly – obrázok D.12. Podrobne sme nezisťovali, prečo sa takýto „duplikát" vo výsledkoch objavil. Pravdepodobne ho má na svedomí nejaká implementačná chyba. Tá môže byť v ľubovoľnej časti spracovnia – od parsovania (autori knižnice *RubyParser* uvádzajú, že čísla riadkov môžu byť v niektorých prípadoch nesprávne[1]), až po samotné zobrazovanie. Podobných fragmentov sme síce našli viacero, považujeme to však stále za zanedbateľné množstvo.

Posledný príklad chybne odhaleného duplikátu je znázornený na obrázku D.13. Tento príklad je špecifický tým, že po technickej stránke ide o správne odhalený duplikát. V Ruby je `require` a `gem` volanie metód (aj bez zátvoriek), ktoré akceptujú reťazce znakov ako parametre. V druhom fragmente kódu sa tiež volajú metódy s reťazcami znakov ako parametrami. Významovo ich však za zduplikovaný kód považovať nemôžeme – prvý fragment vykonáva import potrebných knižníc, druhý obsahuje aplikačnú logiku.

test/helper.rb

```
3. require 'rubygems'
4. require 'test/unit'
5. gem "mocha", ">=0"
6. require 'mocha'
```

lib/grit/git-ruby/repository.rb

```
650.   FileUtils.mkdir_p('branches')
651.   add_file('description', 'Unnamed repository; edit this file to name it.')
652.   add_file('HEAD', "ref: refs/heads/master\n")
653.   FileUtils.mkdir_p('hooks')
```

Obr. D.13: Technicky správne odhalený duplikát, významovo však nemá zmysel a nedá sa refaktorovať.

[1] https://github.com/seattlerb/ruby_parser

Prekrývajúce sa fragmenty. V práci sme niekoľkokrát spomínali, že fragmenty kódu sa navzájom prekrývajú. Cieľom je, aby sme vedeli odhaliť duplikáty všetkých možných častí kódov, malých či veľkých. Vo výsledkoch sme sa rozhodli tieto duplikáty, resp. celé skupiny nijako nespájať. Niektoré skupiny s duplikátmi preto môžu byť súčasťou iných skupín, napríklad tak, ako znázorňujú obrázky D.14 a D.15.

lib/grit/git-ruby/internal/pack.rb

```
145.             if @version == 2
146.               data = read_data_v2(idx)
147.               data.each do |sha1, crc, offset|
148.                 yield sha1, offset
149.               end
```

lib/grit/git-ruby/internal/pack.rb

```
185.             if @version == 2
186.               data = read_data_v2(idx)
187.               data.each do |sha1, crc, offset|
188.                 yield sha1
189.               end
```

Obr. D.14: Duplikát, ktorý je súčasťou väčšieho duplikátu na obrázku D.15.

lib/grit/git-ruby/internal/pack.rb

```
143.         def each_entry
144.           with_idx do |idx|
145.             if @version == 2
146.               data = read_data_v2(idx)
147.               data.each do |sha1, crc, offset|
148.                 yield sha1, offset
149.               end
150.             else
151.               pos = OffsetStart
152.               @size.times do
153.                 offset = idx[pos,OffsetSize].unpack('N')[0]
154.                 sha1 = idx[pos+OffsetSize,SHA1Size]
155.                 pos += EntrySize
156.                 yield sha1, offset
157.               end
```

lib/grit/git-ruby/internal/pack.rb

```
183.         def each_sha1
184.           with_idx do |idx|
185.             if @version == 2
186.               data = read_data_v2(idx)
187.               data.each do |sha1, crc, offset|
188.                 yield sha1
189.               end
190.             else
191.               pos = SHA1Start
192.               @size.times do
193.                 sha1 = idx[pos,SHA1Size]
194.                 pos += EntrySize
195.                 yield sha1
196.               end
```

Obr. D.15: Duplikát, ktorého súčasťou je malý duplikát na obrázku D.14.

Dôvod, kvôli ktorému prekrývajúce sa duplikáty vo výsledkoch nespájame dokopy, môžeme ilustrovať napríklad výsledkami na obrázkoch D.16 a D.17. Prvý z nich zobrazuje 3 veľmi podobné podmienky (v Ruby je `%w()` tzv. *syntactic sugar* (skrátený zápis) pre pole s reťazami znakov). Na druhom obrázku sa nachádzajú väčšie duplikáty, ktoré sa prekrývajú s dvoma duplikátmi na prvom obrázku. Takto obidve skupiny duplikátov dávajú zmysel, po spojení by mohli byť považované za čiastočne chybný výsledok.

lib/grit/git-ruby/git_object.rb

```
312.           if !["blob", "tree", "commit", "tag"].include?(value)
```

lib/grit/git-ruby/internal/loose.rb

```
 51.             if !%w(blob tree commit tag).include?(type) || size !~ /^\d+$/
```

lib/grit/git-ruby/internal/loose.rb

```
119.           if !%w(blob tree commit tag).include?(type) || size !~ /^\d+$/
```

Obr. D.16: Zduplikovaný kód v podmienke, 2. a 3. fragment sú aj súčasťou duplikátov na obrázku D.17.

lib/grit/git-ruby/internal/loose.rb

```
 51.             if !%w(blob tree commit tag).include?(type) || size !~ /^\d+$/
 52.               raise LooseObjectError, "invalid object header"
 53.             end
```

lib/grit/git-ruby/internal/loose.rb

```
119.           if !%w(blob tree commit tag).include?(type) || size !~ /^\d+$/
120.             raise LooseObjectError, "invalid object header"
121.           end
```

Obr. D.17: Zduplikované fragmenty, ktorých časť (podmienka) sa nachádza aj v skupine na obrázku D.16.

E

Obsah priloženého DVD

Súčasťou práce sú aj elektronické materiály dostupné na priloženom DVD. Obsah disku je nasledujúci:

datasets/ dáta použité pri experimentoch

- **manual_experiments/**
  - **wanted_tests** dátová vzorka použitá pri prvom testovaní, manuálnych experimentoch a hľadaní najlepších hodnôt parametrov
  - **wanted_annotations/** manuálne označené duplikáty vo vzorke
- **real_projects/**
  - **grit/** Git repozitár použitej verzie projektu *Grit*

doc/

- **bibliography/** všetka použitá literatúra
- **sources/** zdrojové kódy tohto dokumentu
- **sukenik_dp1.pdf, sukenik_dp2.pdf** dokumenty odovzdané vrámci predmetov Diplomový projekt 1 a 2 v elektronickej podobe
- **sukenik_dp.pdf** tento dokument v elektronickej podobe
- **sukenik_iitsrc2013.pdf** článok prezentovaný na IIT.SRC 2013
- **sukenik_lacko_wikt2012.pdf** článok prezentovaný na WIKT 2012
- **sukenik_navrh-zadania.pdf** návrh zadania tejto práce
- **sukenik_vyskumny-zamer.pdf** výskumný zámer tejto práce

install/ inštalačné súbory Ruby, DevKit a Git

results/ výsledky všetkých vykonaných experimentov

- **automatic_experiments/** experimenty zamerané na hľadanie optimálnych hodnôt vstupných parametrov

params_data/ nespracované výstupy experimentov

correlations.csv, params_results.xlsx spracované výsledky experimentov, vrátane napr. grafov použitých v práci

clustering/ staršie experimenty vykonané pri výbere vhodného klastrovacieho algoritmu

manual_experiments/ ručne vyhodnocované experimenty

pretty_output/ formátované výstupy testovaných nástrojov

raw_output/ nespracované výstupy jednotlivých nástrojov

final_merged.xlsx spracované výsledky experimentov

real_projects/ experimenty na projektoch *Grit* a *Wanted*

grit/ nespracované výstupy jednorazového a 4 inkrementálnych spustení na projekte *Grit*

wanted/ nespracované výstupy jednorazového a 4 inkrementálnych spustení na projekte *Wanted*

results/ spracované výsledky experimentov, vrátane napr. grafov, z ktorých nie všetky boli prezentované v práci

wanted_iterations/ experiment, pri ktorom sme do indexu v iteráciách pridávali stále rovnaké kódy našej dátovej vzorky

output/ nespracované výstupy experimentu

iterations.xlsx spracované výsledky experimentu

visualisation/ vygenerované HTML súbory, ktoré umožňujú prezerať nájdené duplikáty

- duplikáty poslednej verzie projektu *Grit* spracovanej jednorazovo aj inkrementálne
- duplikáty poslednej verzie projektu *Wanted* spracovanej jednorazovo aj inkrementálne
- duplikáty našej dátovej vzorky po 688 odobratiach a pridaniach do indexu
- **template.html** – šablóna vizualizácie, do ktorej je možné vložiť ľubovoľný výstup skriptu `bin/to_js.rb`

src/

bin/, lib/ zdrojové kódy implementovaného nástroja v jazyku Ruby

spec/ unit testy

samples/, tools/ ostatné pomocné nástroja a dočasné súbory

F

Článok – WIKT 2012

V tejto prílohe sa nachádza článok, ktorý bol prijatý a prezentovaný na konferencii WIKT 2012.

Vyhľadávanie zduplikovaného kódu

Ján Súkeník, Peter Lacko

Fakulta informatiky a informačných technológií,
Slovenská technická univerzita v Bratislava,
Ilkovičova 3, 842 16 Bratislava
sukenik08@student.fiit.stuba.sk, lacko@fiit.stuba.sk

Abstrakt. Kopírovanie častí zdrojových kódov na viacero miest, ako jeden zo spôsobov znovupoužitia kódu, je v programátorskej praxi bežne zaužívaný. To aj napriek tomu, že každý programátor nie len vie, prečo by to nemal robiť, ale určite už aj doplatil na následky zduplikovaného kódu. Tie zahŕňajú napríklad viac úsilia, ktoré je potrebné vynaložiť na údržbu kódu, ako aj zvýšený počet chýb v softvéri, ktoré súvisia s tým, že zduplikovaný kód bol na niektorých miestach upravený a inde nie. Existuje však len hŕstka nástrojov, ktoré vyhľadávanie zduplikovaného kódu umožňujú a ani tie sa v praxi nepoužívajú. Domnievame sa, že dôvodmi môžu byť ich vysoké výpočtové alebo pamäťové nároky, komplikovaná obsluha, resp. prevádzka alebo nedostačujúce výsledky. Navrhujeme preto spôsob vyhľadávania zduplikovaného kódu, ktorý využíva abstraktné syntaktické stromy a kombináciu viacerých existujúcich algoritmov resp. metód z dostupnej literatúry. Našim cieľom je vytvoriť prakticky použiteľný nástroj na inkrementálnu detekciu zduplikovaného kódu, ktorý bude pri svojej činnosti využívať systémy na verziovanie zdrojových kódov.

Kľúčové slová: porovnávanie zdrojových kódov, zduplikovaný kód, abstraktné syntaktické stromy

1 Úvod

Bežnou súčasťou vývoja softvérových systémov je kopírovanie jednej časti kódu na viacero miest. Programátori duplikujú existujúci kód aj napriek tomu, že sa to vo všeobecnosti považuje za nesprávne. Doterajší výskum v oblasti údržby softvéru naznačuje, že programy obsahujú 5-10% zduplikovaného kódu [1, 3], čo určite nie je zanedbateľné množstvo. Zduplikovaný kód môže mať nepríjemné následky. Nekonzistentnosť zmien medzi jednotlivými duplikátmi môže spôsobiť chyby, čím sa v budúcnosti zvýši cena vývoja, resp. údržby softvéru. Preto je dôležité mať k dispozícii nástroje, ktoré budú programátorov na duplikáty upozorňovať.

Doteraz už bolo navrhnutých mnoho metód, pomocou ktorých je možné zduplikovaný kód odhaľovať. Krátke prehľady uvádzajú viaceré práce [2, 3, 4]. Medzi najčastejšie používané techniky na určovanie podobnosti zdrojových kódov patria:

- *porovnávanie textu* – citlivé aj na malé zmeny, napr. formátovanie,
- *rôzne metriky kódu* – tiež citlivé na malé zmeny, málo používané [2],

adfa, p. 1, 2011.

G

Článok – IIT.SRC 2013

V tejto prílohe sa nachádza článok, ktorý bol prijatý a prezentovaný na študentskej konferencii IIT.SRC 2013. Článok vyhral ocenenie *Best Paper Award*.

Detection of Code Clones: Necessity or a Myth?

Ján SÚKENÍK*

Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies
Ilkovičova 3, 842 16 Bratislava, Slovakia
sukenik08@student.fiit.stuba.sk

Abstract. Copy-pasting of source codes is the most popular and bug prone way of code reuse ever performed in an industry. All programmers are told to avoid cloning code and all ignore this advice whenever possible. Is the reason laziness? Would better tools help? Recent research suggests that aggressive refactoring of clones may not be necessary. We present an overview of this research and experiments with clone detectors for Ruby language, including our own tool. We show where they work well and where they fail and discuss the need for the tools that will simplify tracking of code clones evolution.

1 Introduction

Code cloning is undoubtedly the most popular way of code reuse. Clones are however perceived as a negative design characteristic and are usually attributed to the limitations of the developers [6]. M. Fowler et al. [4] even include duplicated code in their widely accepted list of code smells.

The main reason why clones are considered bad is their bug-proneness. All programmers know that inconsistent changes to duplicated fragments of code are common source of bugs. Recent research however suggests that code clones may not be as harmful as expected [10, 9].

In this paper we provide an overview of the research in the field of clone detection. Based on this research we conclude that the industry could benefit from the tools which would enable easier tracking and maintenance of code clones during software development. We furthermore propose one approach of such incremental clone detection and we compare it to the three already existing clone detectors while discussing the reasons why such comparisons are difficult to evaluate.

2 Related work

The majority of the research studying code clones so far has been focused on automating clone detection in order to discover clones and force developers to refactor them.

2.1 Existing clone detection tools

Roy et al. [10] provide a comprehensive overview and comparison of existing clone detection tools while dividing them into five groups: (1) text-based tools usually compare raw source codes; they may use some pre-processing or normalization, (2) token-based approaches transform source

* Master degree study programme in field: Software Engineering
Supervisor: Dr. Peter Lacko, Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies STU in Bratislava

H

Návrh článku do vedeckého časopisu

V prílohe sa nachádza návrh článku do vedeckého časopisu.

Source code analysis using the abstract syntax trees

Ján Súkeník, Peter Lacko

***Abstract*—Copy-pasting of source code is the most popular and bug prone way of code reuse ever performed in an industry. All programmers are told to avoid cloning code and all ignore this advice whenever possible. Is the reason laziness? Would better tools help? Recent research suggests that aggressive refactoring of clones can be redundant or sometimes even harmful. We present an overview of this research and discuss the need for the tools that will simplify tracking of duplicates during the evolution of the source code. Consequently we propose new method of code clone detection. The method enables to actively and continuously track duplicated code during the development. Its main advantage is that it can quickly update the results right after any change of the source code (small or large). We tested the method on a few software projects and based on the results we are confident that our tool has the potential to increase quality of software and reduce the development costs.**

***Index Terms*—Code clones, clone detection, abstract syntax trees, bug prediction, clones evolution.**

I. Introduction

CODE CLONING is undoubtedly the most popular way of code reuse. Clones are however perceived as a negative design characteristic and are usually attributed to the limitations of the developers [1]. M. Fowler et al. [2] even include duplicated code in their widely accepted list of code smells.

The main reason why clones are considered bad is their bug-proneness. All programmers know that inconsistent changes to duplicated fragments of code are common source of bugs. Recent research however suggests that code clones may not be as harmful as expected [3], [4].

In this paper we provide an overview of the research in the field of clone detection. Based on this research we conclude that the industry could benefit from the tools which would enable easier tracking and maintenance of code clones during software development. We furthermore propose one approach of such incremental clone detection and we compare it to the three already existing clone detectors while discussing the reasons why such comparisons are difficult to evaluate.

II. Related work

The majority of the research studying code clones so far has been focused on automating clone detection in order to discover clones and force developers to refactor them.

A. Existing clone detection tools

Roy et al. [3] provide a comprehensive overview and comparison of existing clone detection tools while dividing them into five groups:

1) text-based tools usually compare raw source codes; they may use some pre-processing or normalization,
2) token-based approaches transform source code into a sequence of tokens (using compiler-style lexical analysis), which is afterwards scanned for duplicated subsequences,
3) tree-based tools compare abstract syntax trees of the source code and try to find similar sub-trees using tree matching [5], structure-aware hashing [6] or by building characteristic vectors [7] etc.,
4) metrics-based approaches gather various metrics of source codes and detect duplicated fragments by comparing results of these metrics,
5) graph-based approaches turn programs into dependency graphs and consequently try to find isomorphic subgraphs.

We would like to highlight some tools which detect clones in Ruby code. Simian[1] is text-based tool which finds duplicates in multiple languages. Flay[2] is a clone detector specifically designed for Ruby. It uses structure-aware hashing of abstract syntax tree and detects cloned code by comparing these hashes. RubyMine[3] is an IDE for Ruby development which also offers a built-in detection of duplicated code.

B. Is cloned code always harmful?

Kapser and Godfrey [1] investigated multiple different motivations why code is being cloned. They identified number of different cloning patterns, some of which they did not consider harmful, for example in case of platform or hardware variation or templating due to a language limitations.

Another branch of research focuses on evolution of code clones. Pate et al. [4] provide an exhausting review. They show examples of studies, which conclude following:

- consistent changes are on average performed in approximately 40% of clone groups,
- when clones are changed inconsistently, they are rarely made consistent later, which implies that the inconsistent changes are made on purpose,
- two studies show that a big fraction of clones (60 – 80%) are never changed.

All of this implies that aggressive refactoring of code clones is not only unnecessary, but sometimes even not recommended or not possible at all. Saha et al. [9] argue that we should instead focus on tracking and managing clones during the evolution of software systems.

To be able to effectively track clones across evolving versions of software we need incremental code detection approaches. Even this kind of work was already done, for

J. Skenk is with the
P. Lacko is with the

[1] http://www.harukizaemon.com/simian/
[2] http://ruby.sadi.st/Flay.html
[3] http://www.jetbrains.com/ruby/

Literatúra

[1] Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)* (1998), IEEE Comput. Soc, pp. 368–377.

[2] Berkhin, P. A survey of clustering data mining techniques. *Grouping multidimensional data* (2006), 1–56.

[3] Chatterji, D., Carver, J. C., and Kraft, N. a. Claims and beliefs about code clones: Do we agree as a community? A survey. *2012 6th International Workshop on Software Clones (IWSC)* (June 2012), 15–21.

[4] Chilowicz, M., Duris, E., and Roussel, G. Syntax tree fingerprinting for source code similarity detection. In *2009 IEEE 17th International Conference on Program Comprehension* (May 2009), IEEE, pp. 243–247.

[5] Datar, M., Immorlica, N., Indyk, P., and Mirrokni, V. S. Locality-sensitive hashing scheme based on p-stable distributions. *Proceedings of the twentieth annual symposium on Computational geometry - SCG '04* (2004), 253.

[6] Fenton, N., and Neil, M. A critique of software defect prediction models. *IEEE Transactions on Software Engineering 25*, 5 (1999), 675–689.

[7] Fischer, M., Pinzger, M., and Gall, H. Populating a Release History Database from version control and bug tracking systems. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* (2003), IEEE Comput. Soc, pp. 23–32.

[8] Fodor, I. A survey of dimension reduction techniques, 2002.

[9] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. *Refactoring: Improving the Design of Existing Code*. 1990.

[10] Godfrey, M. An integrated approach for studying architectural evolution. *Proceedings 10th International Workshop on Program Comprehension*, 127–136.

[11] Hummel, B., Juergens, E., Heinemann, L., and Conradt, M. Index-based code clone detection: incremental, distributed, scalable. *2010 IEEE International Conference on Software Maintenance* (Sept. 2010), 1–9.

[12] Jiang, L., Misherghi, G., Su, Z., and Glondu, S. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *29th International Conference on Software Engineering (ICSE'07)* (May 2007), no. 0520320, IEEE, pp. 96–105.

[13] Kagdi, H., Collard, M. L., and Maletic, J. I. Towards a taxonomy of approaches for mining of source code repositories. *Proceedings of the 2005 international workshop on Mining software repositories - MSR '05* (2005), 1–5.

[14] Kapser, C. J., and Godfrey, M. W. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering 13*, 6 (July 2008), 645–692.

[15] Kim, S., Zimmermann, T., Pan, K., and Jr. Whitehead, E. Automatic Identification of Bug-Introducing Changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)* (2006), IEEE, pp. 81–90.

[16] Kim, S., Zimmermann, T., Whitehead Jr., E. J., and Zeller, A. Predicting Faults from Cached History. *29th International Conference on Software Engineering (ICSE'07)* (May 2007), 489–498.

[17] Koschke, R., Falke, R., and Frenzel, P. Clone Detection Using Abstract Syntax Suffix Trees. *2006 13th Working Conference on Reverse Engineering* (2006), 253–262.

[18] Lewis, C., and Ou, R. Bug Prediction at Google, 2011.

[19] Lozano, A., Wermelinger, M., and Nuseibeh, B. Assessing the impact of bad smells using historical information. *Ninth international workshop on Principles of software evolution in conjunction with the 6th ESEC/FSE joint meeting - IWPSE '07* (2007), 31.

[20] Nguyen, H., Nguyen, T., Pham, N., and Al-Kofahi, J. Accurate and efficient structural characteristic feature extraction for clone detection. *Approaches to Software*, 1 (2009), 440–455.

[21] Nguyen, T. T., Nguyen, H. A., Al-Kofahi, J. M., Pham, N. H., and Nguyen, T. N. Scalable and incremental clone detection for evolving software. *2009 IEEE International Conference on Software Maintenance* (Sept. 2009), 491–494.

[22] Pate, J. R., Tairas, R., and Kraft, N. A. Clone evolution: a systematic review. *Journal of Software: Evolution and Process 25*, 3 (Mar. 2013), 261–283.

[23] Pérez, J., and Crespo, Y. Perspectives on automated correction of bad smells. *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops - IWPSE-Evol '09* (2009), 99.

[24] Purushothaman, R., and Perry, D. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering 31*, 6 (June 2005), 511–526.

[25] Rahman, F., Posnett, D., Hindle, A., Barr, E., and Devanbu, P. BugCache for inspections. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11* (New York, New York, USA, 2011), ACM Press, p. 322.

[26] Roy, C. K., Cordy, J. R., and Koschke, R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming 74*, 7 (May 2009), 470–495.

[27] Sager, T., Bernstein, A., Pinzger, M., and Kiefer, C. Detecting similar Java classes using tree algorithms. *Proceedings of the 2006 international workshop on Mining software repositories - MSR '06* (2006), 65.

[28] Saha, R. K., Asaduzzaman, M., Zibran, M. F., Roy, C. K., and Schneider, K. A. Evaluating Code Clone Genealogies at Release Level: An Empirical Study. In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation* (Sept. 2010), IEEE, pp. 87–96.

[29] Slaney, M., and Casey, M. Locality-sensitive hashing for finding nearest neighbors. *Signal Processing Magazine, IEEE*, March (2008), 128–131.

[30] Śliwerski, J., Zimmermann, T., and Zeller, A. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes 30*, 4 (July 2005), 1.

[31] Walenstein, A., Jyoti, N., and Lakhotia, A. Problems creating task-relevant clone detection reference data. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.* (2003), no. c, IEEE, pp. 285–294.

[32] Zhang, T., Ramakrishnan, R., and Livny, M. BIRCH. *ACM SIGMOD Record 25*, 2 (June 1996), 103–114.