Fakulta riadenia a informatiky

Michal Ďuračík, Ing.

Autoreferát dizertačnej práce

Algoritmy pre identifikáciu plagiátov zdrojových kódov

na získanie akademického titulu **„philosophiae doctor“ (v skratke PhD.)**
v študijnom programe doktorandského štúdia
aplikovaná informatika

v študijnom odbore:
9.2.9 aplikovaná informatika

Žilina, apríl 2019

Dizertačná práca bola vypracovaná v dennej forme doktorandského štúdia **na Katedre softvérových technológií Fakulte riadenia a informatiky Žilinskej univerzity v Žiline**

Predkladateľ: **Ing. Michal Ďuračík**
Katedra softvérových technológií
Fakulta riadenia a informatiky
Žilinská univerzita v Žiline

Školiteľ: **doc. Ing. Emil Kršák, PhD.**
Katedra softvérových technológií
Fakulta riadenia a informatiky
Žilinská univerzita v Žiline

Oponenti: **prof. Ing. Karol Matiaško, PhD.**
Katedra informatiky
Fakulta riadenia a informatiky
Žilinská univerzita v Žiline

Autoreferát bol rozoslaný dňa: .....

Obhajoba dizertačnej práce sa koná dňa ..... o ..... h. pred komisiou pre obhajobu dizertačnej práce schválenou odborovou komisiou v študijnom odbore **39.2.9 aplikovaná informatika**, **v študijnom programe aplikovaná informatika**, vymenovanou dekanom Fakulty riadenia a informatiky Žilinskej univerzity v Žiline dňa .....

prof. Ing. Karol Matiaško, PhD.
predseda odborovej komisie
študijného programu **aplikovaná informatika**
v študijnom odbore **9.2.9 aplikovaná informatika**
Fakulta riadenia a informatiky
Žilinská univerzita
Univerzitná 8215/1
010 26 Žilina

Úvod

V súčasnosti sa čoraz častejšie aj v bežnom živote objavuje (tento zvyčajne akademický pojem) plagiátorstvo. S rozmachom informačných technológií je čoraz jednoduchšie dostať sa k rôznym zdrojom a vytvoriť plagiát. Na druhú stranu tento rozmach umožňuje aj rozvoj nástrojov, ktoré takéto formy plagiátorstva dokážu odhaliť.

Najväčší dôraz sa v súčasnosti kladie na vývoj metód a nástrojov na odhaľovanie plagiátov textových dokumentoch. Tento prístup je pochopiteľný, nakoľko len na Slovensku sa ročne vyprodukujú desaťtisíce akademických prác. Vyhľadávanie plagiátov v zdrojovom kóde je zaujímavou oblasťou, ktorá prináša nové výzvy. Existuje niekoľko nástrojov, ktoré umožňujú vyhľadávať plagiáty v zdrojovom kóde, ale zatiaľ nikto nevyhľadáva plagiáty v takom rozsahu, ako sa to deje pri textových prácach.

Cieľom tejto práce je preskúmať dostupné metódy spracovania zdrojového kódu a navrhnúť nové, ktoré by dokázali efektívnejšie vyhľadávať plagiáty vo veľkej databáze zdrojových kódov.

V práci sme si vymedzili nasledujúce základné ciele:

- *Návrh vhodnej a efektívnej reprezentácie zdrojového kódu* – po spracovaní zdrojového kódu je nutné navrhnúť vhodnú formu reprezentácie, ktorá umožní efektívne vyhľadávanie plagiátov. V kapitole 4 sa venujeme porovnaniu jednotlivých metód reprezentácie zdrojového kódu.
- *Identifikácia a využitie metód klasterizácie pre potreby zdrojového kódu* – zdrojový kód musíme byť schopný analyzovať a vedieť v ňom vyhľadávať podobné časti. Vzhľadom na požiadavku spracovania veľkého množstva sa ukazuje použitie klasterizácie ako vhodnej metódy. Problémom klasterizácie a návrhu metódy, ktorá umožní inkrementálne pridávať nový zdrojový kód do systému sa venujeme v kapitole 5.
- *Definícia spôsobu interpretácie klastrov a identifikácia plagiátov* – tomuto cieľu sa bližšie venujeme v kapitole 6, kde navrhujeme algoritmy, ktoré na základe pripravených klastrov dokážu vyhľadávať zhodné časti zdrojových kódov. Tieto zhody je potrebné analyzovať, pospájať a vyhodnotiť, aby sme vo finále dostali len relevantné výsledky.
- *Nájsť metódy overenia spoľahlivosti algoritmov* - V súčasnosti neexistujú systémy, ktoré by pracovali na podobnom koncepte. Porovnaním nájdených plagiátov s výsledkami z iných systémov a následnou ručnou kontrolou jednotlivých výsledkov sme schopní overiť spoľahlivosť jednotlivých algoritmov.

Práca sa nebude detailne zaoberať právnymi aspektami plagiátorstva a hodnotením situácií, kedy je možné použiť kód prebratý z internetu (aj keď to jeho licencia dovoľuje). Práca sa bude venovať len algoritmom na vyhľadávanie podobnosti a filtrovaniu týchto podobností. Rozhodnúť, či sa jedná o plagiát musí, vždy používateľ tohto systému.

1 Motivácia

Problém plagiátorstva a iných spôsobov podvádzania je v poslednom období veľmi diskutovanou témou [1]. V spoločnosti sa rozoberajú prípady, kedy boli udeľované tituly za záverečné práce, ktoré obsahujú vysoký podiel zhôd s inými prácami či literatúrou. Tieto prípady pochádzajú prevažne z minulosti, keď ešte záverečné práce neboli evidované v elektronickej forme. Elektronická evidencia záverečných prác v centrálnom registri záverečných prác (CRZP) je vďaka novele zákona o vysokých školách z roku 2009 povinná od roku 2010. Od roku 2010 sa okrem toho zavádza povinnosť kontrolovať predmetné práce pomocou antiplagiátorského systému (APS). Antiplagiátorsky systém ma na starosti identifikáciu zhodných častí prác. Na Slovensku sa pre tieto účely využíva systém ANTIPLAG.

Pojem „plagiátorstvo“ nie je všeobecne definovaný zákonom. Cieľom tejto práce nebude tento pojem definovať, ale pre potreby pochopenia tejto práce uvedieme niekoľko bežne používaných definícií.

- **Plagiát** - napodobnenie alebo doslovný odpis cudzieho diela bez udania predlohy; dielo, ktoré takto vzniklo [2].
- **Plagiátorstvo** - preberanie práce alebo ideí niekoho iného a ich vydávanie za svoje vlastné [3].

Pojem plagiátorstvo sa často používa práve v akademickom prostredí, a chápe sa ako porušenie zásad akademickej etiky. Často sa ale stáva, že pri plagiátorstve dochádza k porušeniu autorských práv, ktoré už zákonom ošetrené je. K plagiátorstvu dochádza v prípade úmyselného, ale aj neúmyselného použitia cudzieho zdroja bez správnej citácie takého zdroja. Rôzne APS poskytujú reporty, ktoré samy o sebe nepotvrdzujú ale ani nevyvracajú to, že predmetná práca je plagiát. Tieto reporty slúžia ako podklad pre skúšobnú komisiu, ktorá následne rozhoduje o originalite predloženej práce.

Reporty z APS obsahujú informácie o dokumentoch, s ktorými má kontrolovaná práca nájdenú zhodu. Pod pojmom **zhoda** chápeme časť textu, alebo zdrojového kódu, ktorá vykazuje vysokú mieru podobnosti. V prípade textových prác sa ako zhodné časti označia časti textu, ktoré majú v dostatočnom rozsahu rovnaký, alebo veľmi podobný obsah. Pod pojmom „veľmi podobný obsah“ máme na mysli mierne zmenený slovosled alebo vypustenie, či pridanie určitých slov.

Problém plagiátorstva sa netýka len akademického prostredia. Narastajúce plagiátorstvo môžeme nájsť v rôznych oblastiach od akademického prostredia [4], cez publicistiku [5] až k patentom [6] v komerčnej sfére.

Na druhú stranu existujú štúdie [7,8], ktoré poukazujú na postupné znižovanie miery plagiátorstva v akademickom prostredí v poslednom období. Práve tieto štúdie dokazujú dôležitosť APS. Obe spomínané štúdie zaznamenali pokles miery plagiátorstva po integrácii APS do procesu obhajoby prác.

Hlavným dôvodom úspechu týchto systémov je ich globálne použitie. Každý APS overuje kontrolovanú prácu voči svojej databáze. Medzi najznámejšie systémy patria už spomínaný systém *ANTIPLAG*, používaný na Slovensku alebo celosvetovo známy systém *Turnitin,* ktorý využívajú mnohé univerzity vo svete, ale aj rôzni vydavatelia vedeckých publikácií.

1.1 Plagiátorstvo v zdrojovom kóde

Vo všeobecnosti sa vnímanie plagiátorstva v zdrojovom kóde nie veľmi líši od plagiátorstva v textových prácach. Dôvody na vznik plagiátov sú v oboch prípadoch totožné. V súčasnosti žijeme v dobe jednoduchého prístupu k rozličným informáciám pomocou internetu. Veľké množstvo rozličných prác, publikácií, kníh a zdrojového kódu je voľne dostupných na rôznych portáloch. Táto jednoduchosť nabáda študentov ku kopírovaniu týchto voľne dostupných zdrojov do svojich prác.

Medzi najpoužívanejšie systémy na detekciu plagiátorstva patrí systém nazývaný *Measure of Software Similarity* (MOSS) od Stanfordskej univerzity a systém *JPlag* z Nemecka. Oba tieto systémy boli vytvorené pred viac ako desiatimi rokmi a ich vývoj pokračuje až do súčasnosti. Okrem nich sa môžeme stretnúť ešte s ďalšími systémami ako napríklad *Plague, YAP* a iné.

Tieto systémy dokážu vyhľadávať plagiáty v rámci určitej relatívne malej vzorky dát. Pokiaľ chceme takéto systémy použiť v procese výučby narazíme na problém. Proces výučby je totižto charakteristický tým, že každoročne v ňom vzniká množstvo nových dát, ktoré treba kontrolovať aj medziročne, a preto je potrebné zaviesť určitú inkrementálnu analýzu týchto dát, tak ako to robia úspešné textové APS.

Aktuálnosť tohto problému dokazuje aj to, že postupne začínajú vznikať komerčné systémy špecializované na vyhľadávanie plagiátov v zdrojovom kóde. Platforma *codio*[1] (online platforma na výučbu programovania, management študentov, zadaní...) integrovala algoritmy na kontrolu plagiátorstva priamo do procesu hodnotenia. Táto kontrola plagiátorstva podobne ako ostatné spomenuté systémy dokáže nájsť plagiáty medzi odovzdanými zadaniami v rámci daného kurzu. Zaujímavejšou službou z tohto pohľadu je služba *Codequiry*[2]. Tá vznikla koncom roka 2018 a poskytuje funkcionalitu APS pre zdrojový kód. Využíva dobre známe technológie z doteraz používaných nástrojov a kombinuje ich s využitím umelej inteligencie.

1.1.1 Problémy pri určovaní plagiátov v zdrojovom kóde

Hlavnou úlohou APS, ako už bolo spomenuté vyššie, je nájsť časti zdrojového kódu, ktoré by mohli byť plagiátom. Problémom ale ostáva, že na rozdiel od textových prác, kde je celkom dobre definované čo je plagiát [9], je v prípade zdrojového kódu rozoznanie plagiátu komplikovanejšie. Samozrejme, existujú prípady, kde je plagiátorstvo zjavné. Jedným z takýchto prípadov je 100% zhoda, keď dvaja študenti odovzdajú to isté. Vzhľadom na to, že

[1] https://codio.com
[2] https://codequiry.com

pri programovaní nezáleží na názve identifikátorov (z funkčného hľadiska) sa často stretávame s prácami, ktoré sa líšia len v názvoch identifikátorov. Takéto práce môžeme opäť prehlásiť za plagiáty. Týmto prípadom sa detailne venovať nebudeme, pretože tie môžeme vždy automaticky prehlásiť za plagiát. Pozrime sa ale na prípady, kde sa študent snaží „zamaskovať“ plagiát.

Kreatívnosť študentov v tejto oblasti je pomerne veľká. My sa budeme hlavne zaoberať tými základnými metódami. V prípade textu sa jedná zväčša o výmenu, pridanie alebo vynechanie slov, viet so zachovaním pôvodnej myšlienky. Často môžeme nájsť aj prípady, v ktorých sa jedná o doslovný preklad z iného jazyka. Na vytvorenie takejto práce stačí, znalosť jazyka, v ktorom je práca písaná. V prípade zdrojového kódu nie sú tieto operácie takéto jednoduché, pretože nositeľom významu v zdrojovom kóde je hlavne štruktúra a poradie jeho jednotlivých prvkov. Jednoduché nahradenie jednotlivých prvkov nie je často možné a pokus o to by spôsobil nefunkčnosť výsledného riešenia. Určité modifikácie možné sú, ale na ich aplikovanie je často potrebná hlbšia znalosť daného programovacieho jazyka, ale aj problému, ktorý práca rieši. To pre nás znamená, že musíme byť schopní detegovať aj určité modifikácie. Zaujímavá z tohto pohľadu je otázka, aké veľké modifikácie musíme byť schopní ešte detegovať. Pretože čím ďalej pôjdeme, tak tým sa situácia viac komplikuje. Definícia plagiátorstva hovorí o „napodobnení alebo doslovnom odpise“, čo by v konečnom dôsledku mohlo znamenať, že všetky práce na jednu tému (zadanie) budú plagiát, lebo poskytujú rovnakú funkcionalitu. Pri hľadaní plagiátov sa preto musíme zamerať na hľadanie podobností medzi jednotlivými algoritmami a celkovou štruktúrou zdrojového kódu, pretože tak ako pri texte, kde každý študent pri písaní práce na konkrétnu tému napíše túto prácu určitým svojským štýlom, tak každý programátor navrhne iné algoritmy a inú štruktúru programu, ktoré mu pomôžu dosiahnuť cieľ.

Ako sa ukazuje, detekcia plagiátov, obzvlášť v zdrojovom kóde, nie je jednoduchá úloha. Pri detekcii plagiátov v zdrojovom kóde narážame na problémy ako návrhové vzory, voľne dostupný kód a nejednotná definícia toho, čo za plagiát považujeme, a čo už nie.

1.1.2 Príčiny vzniku plagiátov

V tejto kapitole sa budeme krátko venovať príčinám vzniku plagiátorstva. Podľa výskumov existuje niekoľko príčin vzniku plagiátorstva. Medzi najčastejšie zaraďujeme [10]:

nedostatok času,

nedostatok vedomostí,

pretože to robia všetci,

pretože skopírovaná práca bude lepšia ako vlastná,

lenivosť,

nevedomosť.

Spomenuté výskumy sa síce venovali všetkým druhom prác, nie len zdrojovému kódu, ale ponúkajú dobrý prehľad, ktorý sa podľa našich skúseností vyskytuje aj pri plagiátoch v zdrojovom kóde. Každý zo spomenutých dôvodov predstavuje z pohľadu študenta vyústenie

iného problému. Pokiaľ chceme odstrániť problém plagiátorstva, mali by sme sa zamerať aj na riešenie toho, čo študentov vedie k plagiátorstvu. Niektoré z dôvodov (ako napríklad nedostatok času alebo nedostatok vedomostí) môžu súvisieť so zle nastavenými pravidlami alebo so zlým spôsobom výučby. Zavedenie APS v tomto prípade zlepšiť situáciu nedokáže. Naopak, v prípadoch, keď sa jedná o plagiát z lenivosti alebo jednoducho z presvedčenia, že to tak robia všetci, je APS výborným nástrojom ako donútiť študentov, aby pracovali samostatne.

1.1.3 Plagiátorstvo na FRI

Otázkou plagiátorstva v zdrojovom kóde sa zaoberáme aj na Fakulte riadenia a informatiky (FRI) na Žilinskej univerzite v Žiline (UNIZA). Pred rokom 2017 sa na FRI takmer nepoužívali APS. V roku 2017 sme začali monitorovať plagiátorstvo pomocou dostupných nástrojov na pár vybraných povinných predmetoch, ktoré s kolegami vyučujeme. Medzi tieto predmety patria predmety *Informatika 1* a *Informatika 2,* ktoré sú základnými predmetmi prvého ročníka študijných programov informatika a počítačové inžinierstvo. Ďalším predmetom, na ktorom sme spustili vyhľadávanie plagiátov je predmet *Algoritmy a údajové štruktúry*. Posledným predmetom, na ktorom sme analyzovali plagiáty, bol predmet *Pokročilé objektové technológie*.

Na detekciu plagiátov sme využívali voľne dostupné nástroje *JPlag* a *MOSS*. Postup pri odhaľovaní plagiátov prebiehal nasledovne:

- Zozbierali sme všetky práce.
- Spustili sme analýzu plagiátorstva.
- Manuálne sme vyhodnotili výsledky a zostavili skupiny študentov, ktorí odovzdali podobné práce.

Pri nájdených zhodách, ktoré boli medzi prácami z rovnakého akademického roku sme sa nepokúšali určiť, kto je autorom originálu, a kto vytvoril plagiát. V ďalšej časti tejto kapitoly budeme všetkých dotknutých študentov považovať za podozrivých z podvodu, pretože z pohľadu APS, a často aj učiteľa, ktorý nemá kompetencie, aby robil policajta, nie je dôležité, kto od koho odpísal, alebo kto komu dal odpísať. V prípade, keď plagiát vznikol odkopírovaním práce, ktorá bola odovzdaná v predchádzajúcich rokoch, sa za plagiátora samozrejme považuje len študent z aktuálne kontrolovaného akademického roku. V tabuľke 1 je zobrazený počet študentov, ktorí boli týmto spôsobom odhalení v priebehu posledných troch akademických rokov. Údaje, ktoré neboli merané, sú v tabuľke označené znakom „-“. Pri niektorých predmetoch sú celkové počty rozpísané. Číslo bez indexu označuje počet plagiátov v rámci jedného akademického roka. Číslo s indexom 1 označuje práce, ktoré vznikli odkopírovaním práce z predchádzajúceho roka. V prípade *Informatiky 1* sa v akademickom roku 2018/2019 vyskytla situácia, kde určitý počet študentov zneužil príliš špecifické inštrukcie od cvičiaceho a neprimerane si zjednodušili prácu. Tento prípad bol taktiež vyhodnotený ako plagiát a v tabuľke je označený indexom 2.

Predmet / Ak. Rok	2016/2017	2017/2018	2018/2019
Algoritmy a údajové štruktúry	30	0	-
Informatika 1	-	4	$2 + 2^1 + 20^2$
Informatika 2	6	$2 + 7^1$	-
Pokročilé objektové technológie	-	4	0

Tabuľka 1: Počet nájdených plagiátov

Na základe štatistík po zaradení APS do procesu výučby na prezentovaných predmetoch môžeme konštatovať, že došlo k zníženiu miery plagiátorstva na predmetoch, ktoré sú určené pre starších študentov. V prípade predmetov určených pre prvákov nepozorujeme výrazný pokles. Štatistiky nám ale ukazujú, že aj tu študenti postupne upúšťajú od kopírovania prác medzi sebou, ale kopírovanie prác od starších spolužiakov je stále prítomné.

1.1.4 Kopírovanie zdrojového kódu študentami FRI

V tejto súvislosti sme spravili prieskum na FRI v období 16.09.2018 do 22.9.2018. Prieskum sa konal pred začiatkom semestra, takže sa ho zúčastnili študenti druhého a vyšších ročníkov akademického roka 2018/2019. Študent zapísaný v druhom ročníku bude v prieskume označený ako študent, ktorý ukončil prvý ročník, tretiak bude študent s ukončeným druhým ročníkom atď.

Celkovo sa tohto anonymného prieskumu zúčastnilo 183 študentov a absolventov. Z toho bolo 90% mužov a 10% žien. Najviac, až 28%, bolo absolventov, nasledovali s podielom 21% študenti, ktorí ukončili 2. a 3. ročník. Za nimi bolo 17,5% ukončených prvákov a zvyšok tvorili študenti, ktorí skončili 4. ročník.

Z výsledkov vyplýva, že len 80% študentov niekedy skopírovalo kód, ktorý našli na internete. Tento výsledok nás prekvapil, pretože kopírovanie zdrojového kódu z internetu (pri dodržaní istých zásad) považujeme za bežnú prácu zo zdrojmi. To znamená, že takmer 20% opýtaných študentov nedokáže pracovať zo zdrojmi. Väčšina týchto študentov sú skončení prváci a druháci. Študentov z vyšších ročníkov, ktorí nikdy neskopírovali kód z internetu, je minimum.

Medzi ďalšie zaujímavé poznatky, ktoré z prieskumu vyplynuli, patrí fakt, že okolo 50% respondentov kopíruje do svojich semestrálnych prác a domácich úloh kód od svojich spolužiakov. 46% študentov, ktorí niekedy skopírovali zdrojový kód z internetu, nikdy neoznačilo príslušnú skopírovanú časť.

Podľa nás závažnejším problémom, ktorý tento prieskum ukázal je to, že až 40% študentov kopíruje aj zdrojový kód, o ktorom nevedia, čo robí. V otázke plagiátorstva sa 49% študentov vyjadrilo, že nepovažujú za plagiátorstvo, keď do svojho zdrojového kódu skopírujú cudzí zdrojový kód. Tento stav poukazuje aj na morálne hodnoty a informovanosť študentov v súvislosti s plagiátorstvom.

Na základe výsledkov toho prieskumu sa jasne preukázala potreba APS pre zdrojový kód. Okrem potreby potláčať plagiátorstvo je možné využiť APS aj ako vzdelávací nástroj pre študentov v oblasti etického používania cudzieho zdrojového kódu.

2 Súčasné metódy vyhľadávania podobností

V tejto kapitole sa zameriame na popis metód na spracovanie a vyhľadávanie podobností v zdrojovom kóde a texte. Popíšeme spoločné a rozdielne princípy analýzy zdrojového kódu a textových dokumentov ako východisko pre našu ďalšiu prácu. V súčasnosti sa rýchlo rozvíjajú algoritmy, ktoré spracovávajú a analyzujú veľké množstvá textových dokumentov. Zdrojový kód je taktiež reprezentovaný pomocou textu, a preto je možné princípy využívané v textových dokumentoch s menšími úpravami uplatniť aj na zdrojový kód.

Napriek tomu, že textové dokumenty a zdrojový kód sú si v mnohých oblastiach podobné, existujú medzi nimi určité rozdiely.

Textové dokumenty obsahujú text napísaný v určitom jazyku (alebo viacerých jazykoch). Tento text môžeme na základe štruktúry a významu rozdeliť na *odstavce*, *vety, slová, písmená.* Pri spracovaní textových dokumentoch sa zvyčajne ako s najmenšou jednotkou pracuje so **slovom**. Textové dokumenty okrem iného obsahujú aj formátovanie (tučné písmo, veľkosť a farba písma...), ktoré môže dodávať ďalšie informácie o význame daného textu.

Zdrojové kódy sú určené na spracovanie počítačom. Platia pre ne prísnejšie pravidlá ako pre obyčajný text. Obsahujú určitú štruktúru, ktorá zodpovedá gramatike jazyka, v ktorom je zdrojový kód napísaný. V závislosti od jazyka môžu obsahovať *bloky (triedy, metódy, funkcie...), výrazy...* a pod. Základnou jednotkou je *token*, ktorý je najmenšou časťou zdrojového kódu. Zdrojový kód neobsahuje formátovanie, aké poznáme z textových dokumentov.

2.1 Špecifiká analýzy zdrojového kódu

Zdrojový kód je na rozdiel od textu priamo určený na spracovávanie počítačom. Pri analýze zdrojového kódu je ale potrebné počítať s tým, že zdrojový kód píšu ľudia a zvoliť správnu metódu na konkrétnu analýzu. Podobnosť zdrojového kódu môžeme určovať na základe významu alebo štruktúry. Tieto metódy preto môžeme rozdeliť na tri skupiny:

- Analýza zdrojového kódu ako textu v prirodzenom jazyku.
- Analýza prúdu tokenov.
- Analýza modelu zdrojového kódu.

Tieto metódy sa líšia v úrovni prístupu k zdrojovému kódu. Prvou z nich je analýza zdrojového kódu ako čistého textu. V nej sa predpokladá, že programátor dodržuje konvencie a zdrojový kód obsahuje dodatočné informácie popisujúce jeho význam. Algoritmy, ktoré sa používajú, majú za cieľ extrahovať práve tieto dodatočné informácie a na základe nich určovať tematicky podobné zdrojové kódy. Ďalšia úroveň je podobná ako tá predchádzajúca, ibaže neskúma význam textu z pohľadu programátora, ale z pohľadu počítača, ktorý zdrojový kód „vidí“ ako zoznam príkazov. Jednotlivé sekvencie príkazov majú v kontexte gramatiky jazyka svoj význam a prikazujú počítaču, čo sa má vykonať. Poslednou úrovňou je analýza modelu zdrojového kódu. Na rozdiel od predchádzajúcej úrovne model pridáva dodatočné

informácie o kontexte, v ktorom sú príkazy použité a umožňuje komplexnejšie analýzy zdrojového kódu.

2.1.1 Analýza zdrojového kódu ako textu

Prvou možnosťou je analýza zdrojového kódu, ako by to bol obyčajný text inak nazývaná aj *Natural language processing* (NLP). Pod pojmom *natural language* rozumieme jazyk, ktorý používajú ľudia pri bežnej komunikácii (slovenčina, angličtina...). Aplikácia metód z NLP priamo na zdrojový kód prináša niekoľko problémov.

Programovacie jazyky poskytujú štandardnú knižnicu so základnými funkciami, triedami a metódami. Tie bývajú pomenované logicky a podľa určitých konvencií, takže ich spracovanie nepredstavuje väčší problém. Problém môže nastať pri vlastnom kóde programátora. Každý programátor si môže pomenovávať svoje identifikátory podľa vlastnej predstavy a vo vlastnom jazyku, čo v určitom zmysle značne komplikuje analýzu.

Spracovaniu textových dokumentov pomocou NLP sa venuje množstvo prác, ale využitie NLP na spracovanie zdrojového kódu nie je zatiaľ úplne bežné. Existujú práce, ktoré ho používajú na rôzne analýzy zdrojového kódu [11] .

2.1.2 Analýza tokenov

Pri analýze zdrojového kódu pomocou tokenov sa využívajú podobné metódy ako pri spracovávaní pomocou NLP. Rozdiel spočíva v chápaní významu konkrétnej sekvencie znakov. V prípade NLP sa používa význam, ktorý do daného tokenu dal programátor. Naopak, pri analýze tokenov je tento význam nedôležitý a používa sa len význam na základe gramatiky jazyka.

Táto úroveň spracovania zdrojového kódu je najstaršia spomedzi ostatných spomenutých. Používa sa už od začiatkov programovacích jazykov. Postupom času sa vylepšuje o nové poznatky z iných oblastí, alebo na nej stavajú nové a modernejšie metódy. Využíva sa od nástrojov na podporu programovania, ktoré pomocou nej napríklad kontrolujú správnosť syntaxe, až po veľké systémy na kontrolu kvality a bezpečnosti zdrojového kódu. Existujú rôzne systémy a prístupy k riešeniu tohto problému, ale za hlavné sa považujú *JPlag* a *MOSS*.

2.1.3 Analýza modelu zdrojového kódu

Prúd tokenov je efektívna forma reprezentácie zdrojového kódu. Avšak pomocou tohto prístupu sa ťažko analyzujú komplexnejšie problémy. Pokiaľ chceme vykonávať zložitejšie analýzy, tak tento prúd musíme transformovať do vhodnej formy. Skoro každý nástroj, ktorý pracuje priamo s prúdom tokenov, využíva svoju vlastnú formu, v ktorej tieto tokeny ukladá. Ako univerzálna reprezentácia sa často využíva AST.

AST svoje využitie prirodzene nachádza aj v algoritmoch na odhaľovanie plagiátorstva. Jeho hlavnou výhodou oproti algoritmom, ktoré pracujú s textom alebo tokenmi je to, že priamo popisuje štruktúru programu bez konkrétnych implementačných detailov.

Tieto algoritmy sú založené na jednoduchej myšlienke porovnávania stromových štruktúr. Keďže porovnávanie stromových štruktúr je náročná operácia, tak používané algoritmy využívajú transformáciu týchto stromov do lineárnych foriem. V súčasnosti sa na to používajú dva prístupy:

- hašovanie,
- charakteristické vektory.

Myšlienka hašovania je založená na tom, že pre každý uzol stromu dokážeme vypočítať jeho haš. Následne sa pomocou špeciálneho algoritmu porovnávajú tieto haše a určujú sa podobné časti zdrojového kódu.

Ďalšou často používanou možnosťou je reprezentácia zdrojového kódu pomocou vektorov. V súvislosti s plagiátorstvom sa nám nepodarilo nájsť práce, ktoré by pri hľadaní plagiátov používali vektory. Vektory sa využívajú v oblasti identifikácie klonov [12] v zdrojových kódoch, čo je veľmi podobná oblasť.

3. Metóda vyhľadávania plagiátov v zdrojovom kóde

V predchádzajúcej kapitole sme sa venovali analýze súčasných metód na vyhľadávanie plagiátov, respektíve zhôd, ktoré sú základom APS systémov. Z analýzy vyplýva, že vyhľadávanie plagiátov nie je jednoduchá a triviálna operácia, ale predstavuje komplexný proces. V prípade textových dokumentov existujú rôzne metódy a v súčasnosti prevládajú veľké centralizované riešenia (ANTIPLAG, Turnitin, ...). Hlavnou výhodou týchto systémov je množstvo dát, ktoré dokážu spracovávať. Čím väčšie množstvo prác má takýto systém v databáze, tým je jeho spoľahlivosť vyššia.

Počas analýzy sme nedokázali nájsť takýto systém pre oblasť zdrojového kódu. V súčasnosti je už možné pozorovať prvé lastovičky aj v tejto oblasti. Existujú práce, ktoré síce popisujú adaptáciu textovo orientovaných systémov na zdrojový kód, ale aj naša analýza ukázala, že zdrojový kód, aj keď je to len text, sa od textových dokumentov odlišuje. Postupom času vznikali rôzne metódy spracovania zdrojového kódu. V poslednom období sa ukazujú výhody práce so zdrojovým kódom vo forme stromu.

V ďalších podkapitolách sa budeme venovať dôvodom, pre ktoré nie sú súčasné riešenia dostatočné, popíšeme požiadavky, ktoré by mala ideálna metóda spĺňať a popíšeme nami navrhnutú metódu.

3.1 Nedostatky súčasných metód

Bežne používané metódy pre odhaľovanie plagiátorstva v zdrojovom kóde vznikali spoločne s metódami pre textové dokumenty už od minulého storočia. V prípade textových dokumentov vývoj napreduje ďalej, vznikajú pokročilejšie metódy, ktoré pri odhaľovaní plagiátov spoliehajú na strojové učenie a veľké systémy, ktoré tieto metódy integrujú. Pre zdrojový kód môžeme nájsť tiež veľké množstvo inovatívnych prístupov, ale málo z nich je aj reálne využívaných pri detekcii plagiátorstva.

Za hlavné nedostatky, na ktoré sme narazili počas využívania súčasne dostupných systémov považujeme:

zastaranosť,

- uzavretosť,
- zložitý proces vyhodnocovania plagiátov,
- nemožnosť využitia systémov vo veľkom meradle.

3.2 Požiadavky na novú metódu

Pred návrhom vlastnej metódy na vyhľadávanie plagiátov v zdrojom kóde sme na základe analýzy problému plagiátorstva popísaného v kapitole 1.1 a praktických problémov, na ktoré sme narazili počas používania voľne dostupných nástrojov, sformulovali požiadavky a vlastnosti navrhovanej metódy. Medzi základné požiadavky patria:

- Využitie efektívnych spôsobov spracovania a reprezentácie zdrojového kódu.
- Efektívna organizácia zdrojového kódu.
- Škálovateľný spôsob vyhľadávania zhôd.
- Perzistencia dát.
- Tvorba reportov pre konkrétnu testovanú úlohu.
- Možnosť filtrovať nájdené zhody.
- Univerzálnosť.

Všetky v súčasnosti používané systémy na odhaľovanie plagiátov v zdrojovom kóde používajú rovnaké metódy, ako sa používajú v textových dokumentoch. Na rozdiel od ostatných prístupov pokúsime využiť syntaktické stromy ako základ pre navrhovanú metódu.

Pokiaľ chceme vytvoriť veľkú databázu, voči ktorej sa budú porovnávať testované práce, musíme pri návrhu metódy myslieť na škálovateľnosť. Pod pojmom škálovateľnosť máme na mysli možnosti rozšírenia systému s ohľadom na narastajúce množstvo dát.

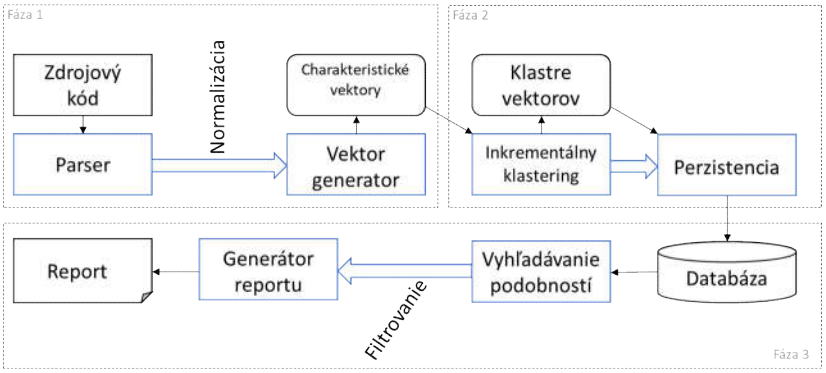
Perzistencia dát je dôležitá hlavne z pohľadu vyhľadávania plagiátov voči historickým dátam, resp. iným zdrojom. Tieto zdroje musia byť určitým spôsobom uchovávané, aby sa o ich manažment nemusel starať používateľ a jednotlivé dáta boli prístupné neustále. Tvorba reportov pre konkrétnu prácu bude pozostávať práve z porovnania tejto práce voči perzistentne uloženým dátam.

Navrhovaná metóda musí umožniť automatické alebo poloautomatické odstraňovanie nevýznamných zhôd tak, aby používateľ dostával v reportoch len relevantné zhody.

Poslednou požiadavkou bola univerzálnosť. Pod pojmom univerzálnosť sa rozumie jednoduchá možnosť napríklad adaptovať podporu pre nový programovací jazyk alebo zmeniť ľubovoľný proces používaný v metóde vyhľadávania plagiátov.

3.3 Návrh novej metódy

Pri návrhu sme vychádzali hlavne z požiadaviek špecifikovaných v predchádzajúcej kapitole. Navrhnutá metóda je zobrazená na obrázku nižšie.



Obrázok 1: Diagram komponentov metódy na vyhľadávania plagiátorstva

Hlavnou prednosťou metódy je jej modulárnosť, pretože je rozdelená do troch fáz a každá sa skladá z dvoch algoritmov. Detailnému popisu jednotlivých fáz sa budeme venovať v nasledujúcich kapitolách. Jednotlivé fázy reprezentujú:

- Spracovanie a reprezentácia zdrojového kódu.
- Zhlukovanie a perzistencia dát.
- Vyhľadávanie podobností a tvorba reportu.

4 Spracovanie a reprezentácia zdrojového kódu

V tejto kapitole sa budeme venovať spracovaniu a reprezentácii zdrojového kódu. Celý postup budeme demonštrovať na spracovaní zdrojového kódu v jazyku C#, ale rovnaký spôsob sa dá použiť aj pre iné programovacie jazyky. Na záver kapitoly uvedieme príklad pre spracovanie aj iného zdrojového kódu ako kódu napísanom v programovacom jazyku C#.

Priamo parsovaniu zdrojového kódu sa venovať nebudeme, nakoľko existuje veľké množstvo nástrojov, ktoré nám zdrojový kód dokážu spracovať. Detailnejšie sa zameriame na reprezentáciu zdrojového kódu prostredníctvom syntaktického stromu a popíšeme transformácie syntaktických stromov do štruktúr, ktoré sú jednoduchšie pre spracovávanie algoritmami na vyhľadávanie plagiátov.

4.1 Spôsoby reprezentácie zdrojového kódu

V súčasnosti sa pre účely analýzy zdrojového kódu používajú rôzne metódy reprezentácie. Medzi dva základne prístupy môžeme zaradiť reprezentáciu zdrojového kódu pomocou prúdu tokenov a AST. Súčasné vedecké výskumy ukazujú, že pre účely vyhľadávania ako plagiátov tak aj klonov v zdrojových kódoch, sú metódy založené na stromových reprezentáciách zdrojového kódu výhodnejšie [12].

AST je jednou z najčastejších metód reprezentácie zdrojového kódu vo forme stromovej štruktúry. Overenie jednotlivých metód reprezentácie zdrojového kódu sme sa rozhodli aplikovať na programovací jazyk C#. Je to moderný programovací jazyk a spolu s .NET

knižnicou poskytuje pohodlné prostredie pre vývoj. Jednou z výhod analýzy programového kódu napísaného v jazyku C# je aj jednoduchá možnosť spracovania tohto kódu. Na rozdiel od iných jazykov je pre C# dostupný oficiálny parser zdrojového kódu vyvíjaný priamo Microsoftom, vďaka čomu je zaručená aj budúca kompatibilita. Zameranie nášho riešenia na C# ale neznamená, že nie je všeobecne použiteľné na ľubovoľný programovací jazyk. Jazyk C# nám slúži len na demonštráciu princípov, ktoré je s istými úpravami aplikovateľné na ľubovoľný jazyk.

Na spracovanie zdrojového kódu používame .NET Compiler Platform (projekt Roslyn) . Tento nástroj slúži ako platforma pre kompiláciu C# kódu, naprogramovaná v programovacom jazyku C#. V našej práci z neho využívame syntax analyzer, pomocou ktorého generujeme syntaktický strom, ktorý následne spracovávame.

4.1 Transformácia AST na lineárne štruktúry

Keďže porovnávanie stromových štruktúr je výpočtovo náročná operácia, rozhodli sme sa, pre hľadanie plagiátov, transformovať tento strom do lineárnej štruktúry, resp. kolekcie štruktúr. V analýze sme popisovali dve základné možnosti – hašovanie a vektorizácia.

Pomocou tohto jednoduchého experimentu sme si overili reálne možnosti jednotlivých prístupov. Hašovanie sa ukázalo ako jednoduchý, no nie príliš efektívny prístup. Použitie vektora, ako charakteristiky viedlo celkovo k lepším výsledkom. Implementácia tohto prístupu bola náročnejšia, pretože pre rôzne celky zdrojového kódu (trieda, metóda) sme výpočet tohto vektoru realizovali odlišne. Na základe vykonaných experimentov sme usúdili, že vektor pozostávajúci z početností jednotlivých typov uzlov daného podstromu sa javil ako najlepšia možnosť.

Z časového hľadiska sa ukazuje, že využitie hašovania je o 30% rýchlejšie ako využitie vektora. Tento rozdiel nie je taký výrazný hlavne preto, ako sú jednotlivé algoritmy implementované. Aj pre hašovanie aj pre vektory, je nutné vypočítať tieto charakteristiky pomocou traverzovania celého podstromu, čo je výrazne časovo náročnejšie ako samotné porovnávanie hašov alebo vektorov.

4.2 Vektorizácia zdrojového kódu

Pre vektorizáciu zdrojového kódu sme si ako inšpiráciu zvolili prácu *DECKARD: Scalable and Accurate Tree-based Detection of Code Clones* [11]. Autori v tejto práci zavádzajú algoritmus na tvorbu charakteristických vektorov. Okrem toho popisujú škálovateľný spôsob klasterizácie týchto vektorov pre účely vyhľadávania klonov. Zdrojový kód tohto algoritmu je verejne dostupný, a využívaný pri rôznych výskumných prácach aj v súčasnosti.

Algoritmus má niekoľko fáz. Na začiatku sa na základe gramatiky vygeneruje parser. Na tieto účely sú používané tzv. parser-generátory ako *yacc*, *bison* a *ANTLR*. V ďalšom kroku sú pomocou týchto generátorov transformované zdrojové kódy do príslušných parsovacích stromov (parse trees). Následne sú na základe týchto stromov generované vektory. Tieto vektory sú klastrované na základe ich Euklidovskej vzdialenosti. Nakoniec sa vykoná post-processing, pomocou ktorého sa vygenerujú reporty o nájdených klonoch.

Algoritmus vektorizácie zdrojového kódu je dostupný pre programovacie jazyky C, C++, Java a PHP takže ho nemôžeme jednoducho použiť v našom prostredí jazyka C#. Preto sme si tento algoritmus prispôsobili pre naše potreby.

4.2.1 Generovanie vektorov

Pri zavádzaní charakteristických vektorov do našej práce prevezmeme pojmy definované autormi DECKARD algoritmu. Charakteristické vektory slúžia na zachytenie štrukturálnej informácie stromov alebo lesu stromov. Charakteristický vektor daného podstromu je definovaný ako bod $< c_1, \ldots, c_n >$ v Euklidovskom priestore, kde každé c_i reprezentuje početnosť výskytu určitého vzoru v podstrome. Základný vzor, ktorý môžeme v tomto prípade sledovať, je početnosť výskytu jednotlivých druhov uzlov. Samozrejme nebudeme vytvárať zložku v tomto vektore pre každý typ uzla. Pri analýze sme si určili relevantné a irelevantné uzly, a práve toto rozdelenie využijeme, a pri zostavovaní vektora sa zameriame len na relevantné uzly.

Vektory generujeme pre všetky relevantné uzly v syntaktickom strome. Pri generovaní postupujeme post-order prehliadkou stromu tak, že pre každý uzol dostaneme jeho charakteristický vektor sčítaním vektorov potomkov s vlastným vektorom otca.

Algoritmus používaný v nástroji DECKARD sme priamo adaptovali na naše prostredie. Výstupom tohto algoritmu je množina charakteristických vektorov popisujúca daný zdrojový kód. Tieto vektory sú generované od najmenších dostatočne veľkých častí až po celý kód (triedu, program). Vektory sa popísaným spôsobom generujú pre každý zdrojový súbor, a následne sa vložia do spoločnej množiny, ktorá popisuje celý program.

4.2.2 Spájanie vektorov

Základný algoritmus generovania vektorov popisuje spôsob tvorby charakteristických vektorov pre stromy a podstromy. Druhou fázou je tvorba vektorov pre priľahlé podstromy a lesy. Myšlienka spájania vektorov je založená na tom, že na rozdiel od základného algoritmu, ktorý generuje vektory na základe štruktúry kódu, spájanie generuje vektory pre susedné časti zdrojového kódu.

Algoritmus spájania vektorov ako je popisovaný v systéme DECKARD môže často vygenerovať, podľa nás, nevýznamné vektory (napríklad vektor, ktorý pokrýva kód, ktorý začína v polovici jednej metódy a končí v polovici druhej metódy). Preto sme navrhli jeho modifikáciu. Táto modifikácia vychádza z predpokladu, že plagiáty vznikajú predovšetkým kopírovaním celých blokov (napríklad celá metóda, cyklus...) alebo sa kopíruje viacero príkazov (telá metód). Prvý spôsob je dobre obsiahnutý v procese základného generovania vektorov. Druhý spôsob je okrem iného obsiahnutý v algoritme spájania vektorov. Náš modifikovaný algoritmus má za cieľ spájať súrodenecké uzly. Mohlo by sa zdať, že spájanie súrodeneckých uzlov nemá veľký význam, pretože vektory týchto uzlov zvyčajne bývajú obsiahnuté v uzle ich rodiča. V prípade použitia nami navrhovaného algoritmu dokážeme generovať vektory, ktoré nám pomôžu pri hľadaní čiastkových plagiátov.

4.2.3 Overenie algoritmov vektorizácie zdrojového kódu

Pri overovaní správnosti navrhnutých a implementovaných algoritmov sme využili architektúru systému DECKARD. Tá nám jednoducho umožňuje použiť vektory, ktoré sme vygenerovali naším algoritmom v našom programovom prostredí, a klastrovať ich pomocou algoritmu DECKARD.

Ďalej sme využili skutočnosť, že **Java** (jazyk, pre ktorý dokáže generovať vektory DECKARD) a **C#** (jazyk, pre ktorý sme generátor vektorov implementovali my), sú celkom podobné. Začali sme s kódom napísaným v Jave a pomocou DECKARD algoritmu sme preň vytvorili klon report (zoznam duplikátnych častí zdrojového kódu). Následne sme tento kód, s minimálne potrebnými zmenami, pretransformovali do C#. Z tohto C# kódu sme vygenerovali pomocou našich algoritmov vektory, z ktorých sme následne pomocou DECKARD algoritmu vytvorili klon report.

Ukázalo sa, že nájdené klony boli v oboch jazykoch takmer totožné. Rozdiely boli v početnosti nájdených zhôd. V prípade javy bolo nájdených, čo sa týka početnosti, menej klonov ako v prípade C#, ale po analýze prekrytia jednotlivých zhôd sa ukázalo, že pokrývajú rovnakú časť zdrojového kódu. Spôsobené to je tým, že pri porovnávaní sme použili rovnaké hodnoty parametrov jednotlivých algoritmov. Náš algoritmus generuje viac vektorov, pretože strom, nad ktorým pracuje, je bohatší. Tento problém sa samozrejme dá odstrániť zmenou parametrov.

V druhej fáze sme porovnávali pôvodný algoritmus s naším modifikovaným algoritmom. Prvým zásadným rozdielom je množstvo vygenerovaných vektorov, ktoré môže byť aj o 50% menšie (táto hodnota závisí od štruktúry daného kódu). Na druhej strane, aj počet nájdených zhôd je nižší. Porovnaním týchto nájdených zhôd sme zistili, že kompletne chýbajú zhody časti kódov na rôznych úrovniach. Na záver môžeme povedať, že pôvodný algoritmus je presnejší ako náš modifikovaný algoritmus. Musíme si preto položiť otázku, či potrebujeme hľadať takéto typy podobností aj pri identifikácii plagiátorstva. Vzhľadom na to, že momentálne ešte nemáme algoritmus na identifikáciu plagiátov, nemáme jasnú odpoveď na túto otázku. Na druhej strane, menšie množstvo vygenerovaných vektorov zrýchli proces hľadania zhodných častí kódu, a taktiež aj pamäťové nároky potrebné pri tomto procese.

5 Zhlukovanie, vyhľadávanie a perzistencia vektorov

Po spracovaní zdrojového kódu sme získali zoznam vektorov, ktoré nám charakterizujú zdrojový kód. Týchto vektorov je relatívne veľa, a je ich potrebné spracovať. Klastering sa ukazuje ako ideálna metóda, pomocou ktorej môžeme tieto vektory roztriediť na skupiny, a jednotlivé skupiny spracovávať samostatne. Okrem toho nám umoží rozdeliť vektory aj na skupiny vektorov, ktoré sú si podobné. Taktéto rozdelenie je možné využiť pri vyhľadávaní, vďaka čomu nemusíme vyhľafávať vo veľkej skupine vektorov, ale stačí nám pri vyhľadávaní podobností prehľadávať len jeden klaster.

V tejto kapitole sa budeme venovať metódam klasifikácie zdrojového kódu, popíšeme si aplikáciu **K-Means** metódy na charakteristické vektory, a navrhneme jej **rozšírenie pre potreby kontinuálneho spracovania zdrojového kódu**. Po klasifikácii vektorov sa budeme venovať overeniu inkrementálneho využitia K-Means a vyhľadávaniu zhôd pomocou K-

Means. Nakoniec navrhneme **metódu perzistencie** dát na základe klasteringu s využitím relačnej databázy.

5.1 Zhlukovanie vektorov pomocou K-Means algoritmu

Klastering sa zvyčajne používa na zhlukovanie rozličných vzoriek dát alebo pri datamingu. V našom konkrétnom prípade chceme využiť klastering ako nástroj na pred-triedenie vektorov, ktoré nám umožní následne zjednodušiť vyhľadávanie podobných vektorov. Štandardne sa pri použití metód klasteringu v oblasti dataminingu rozdelí vstupná množina dát na dve skupiny – trénovaciu a kontrolnú. Trénovacia množina sa využije na tvorbu klastrov. S využitím dát z kontrolnej množiny sa následne overí presnosť tejto klasifikácie. V našom prípade nemáme množinu, ktorá je konečná, pretože systém musí umožniť postupné zaraďovanie ďalších a ďalších prác. Naším cieľom bolo navrhnúť metódu klasteringu, ktorá by umožňovala postupné pridávanie nových dát, a postarala by sa o udržiavanie klastrov v optimálnej konfigurácii.

Pre potreby klasteringu je nutné zadefinovať niekoľko pojmov. Na začiatok musíme uviesť, čo pre nás znamená, že vektory sú si podobné. Charakteristický vektor pozostáva zo zložiek, ktoré reprezentujú početnosť jednotlivých typov uzlov syntaktického stromu, z ktorého pochádza. Jednotlivé zložky tým pádom popisujú obsah zdrojového kódu (početnosť konkrétnych prvkov programovacieho jazyka), ale aj štruktúru (početnosť štrukturálnych prvkov syntaktického stromu). Pokiaľ máme kúsky zdrojového kódu, ktoré sa líšia len v detailoch, ich charakteristické vektory sa budú taktiež len mierne líšiť v hodnotách niektorých zložiek. Pokiaľ sú tieto dva kusy kódu rozdielne, ich vektory budú taktiež rozdielne. V našom algoritme sme sa rozhodli využiť Euklidovskú vzdialenosť na určenie podobnosti dvoch vektorov.

Jednotlivé metódy klastrovania sme analyzovali, a ako najvhodnejší sa ukázal K-Means. Základný K-Means algoritmus má dve fázy. V prvej fáze sa vyberú centrá klastrov a v druhej fáze sa snažíme optimalizovať rozloženie týchto centier premiestňovaním vektorov medzi klastrami a následným prepočtom centier novo vytvorených klastrov.

5.2 Inkrementálne použitie K-Means algoritmu

Pre potreby vyhľadávania zdrojového kódu vo veľkom meradle potrebujeme algoritmus K-Means modifikovať. Zo zrejmých dôvodov (výpočtová náročnosť) nie je možné vykonávať takýto klastering vždy, keď budeme chcieť vygenerovať report pre nové zadanie. Aj napriek tomuto sme sa rozhodli využiť základnú verziu K-Means algoritmu. V našej metóde budeme algoritmus K-Means používať opakovane. Najskôr z určitého základného datasetu vygenerujeme pomocou K-Means východzie rozloženie klastrov. Potom budeme postupne pridávať nové vektory do takto postaveného systému a budeme sa snažiť udržať klastre v „optimálnom“ rozložení. Základnou možnosťou, ktorou môžeme túto požiadavku docieliť, je vykonanie druhej fázy K-Means algoritmu po každom novom pridanom vektore. Vďaka tomu, že celý algoritmus bude štartovať z už takmer optimálneho stavu, bude počet krokov na jeho dokončenie minimálny. Na druhej strane aj takýto prístup by bol neefektívny. Dá sa totižto predpokladať, že pridanie jedného vektora do tohto systému nespôsobí zásadnú zmenu

v rozložení klastrov. Na základe našich meraní, pri dostatočne veľkom úvodnom datasete, spôsobí pridanie jedného vektora zmenu rozloženia klastrov vo veľmi malom množstve prípadov. Pri dostatočne veľkom množstve vektorov sa tieto malé posuny naakumulujú. Našim navrhnutým riešením je vykonávať tento reklastering nie po každom vektore, ale vždy až po určitom množstve pridaných vektorov.

Problémom tohto prístupu naďalej ostáva to, že musíme správnym spôsobom určiť, kedy je reklastering potrebný. Pre naše potreby nevadí, že klastre nebudú vždy optimálne rozložené. Určitá miera neoptimality neovplyvní využitie klasteringu ako spomínaného nástroja na pred-triedenie vektorov. Ďalej predpokladáme, že s narastajúcim počtom vektorov v tomto systéme bude vplyv nových vektorov na rozloženie klesať, a teoreticky by sa rozloženie časom mohlo aj ustáliť.

5.3 Validácia inkrementálneho K-Means algoritmus

Pri validácii sme spracovali pomocou K-Means klasteringu dva rozdielne datasety. Ukázalo sa, že aj napriek ich rozdielnosti, sú charakteristické vlastnosti vektorov (početnosť jednotlivých zložiek a dĺžka vektorov) takmer rovnaké. Dôležitou otázkou, ktorou sme sa zaoberali v tejto časti, bolo určenie potrebného počtu klastrov. Z výsledkov vyplýva, že čím viac klastrov zaradíme, tým lepšie výsledky dostaneme. Na druhej strane si treba uvedomiť, že so zvyšujúcim sa počtom klastrov narastajú aj výpočtové nároky na samotný klastering. Našťastie naše výsledky ukazujú, že pridávať klastre sa zvyčajne oplatí len po určitú hranicu, a pridávať klastre nad túto hranicu nemá zmysel. Ďalším zaujímavým poznatkom bolo, že táto hranica postupne s narastajúcim počtom vektorov rastie. Vďaka tomuto nie je možné jednoducho stanoviť presný počet klastrov.

Druhým problémom, ktorým sme sa zaoberali, bolo overenie možností inkrementálneho klasteringu. Výsledky ukazujú, že naša hypotéza, že s narastajúcim množstvom vektorov v systéme klesá vplyv nových vektorov na polohy centier, bola pravdivá. Jedinou otázkou ale zostáva, kedy treba robiť reklastering. Na túto otázku nemáme jednoznačnú odpoveď, no na základe pozorovaní máme odporúčanie vykonávať reklastering minimálne pri zmene typu úlohy, ktorú do systému pridávame. Ako sa ale ukazuje, postupom času bude reklastering menej potrebný a pravidlá na jeho vykonávanie sa môžu upraviť.

5.4 Vyhľadávanie vektorov s využitím klasteringu

Toho vyhľadávanie je dôležité pri hľadaní zhodných častí. Ako sme na začiatku uviedli, vyhľadávať budeme len úplne zhodné vektory a potrebujeme preto rýchlu metódu, ako nájsť požadovaný vektor. Navrhnutá metóda pozostáva z dvoch častí. Najskôr nájdeme klaster, do ktorého vyhľadávaný vektor patrí, a následne vyhľadáme vektor v príslušnom klastri.

Ukázalo sa, že jednotlivé zložky vektora sú medzi sebou závislé, takže výber zložiek na základne entropie nebude poskytovať dobré výsledky. Lepším spôsobom výberu sa ukázal postupný výber zložiek na základe podmienenej entropie. Indexovať 5 zložiek vektora nie je žiaden problém, preto aj v ďalšej práci budeme fixne používať index zložený práve z piatich prvkov.

5.5 Perzistencia dát

Poslednou časťou, ktorej sa budeme v tejto kapitole venovať je perzistencia dát. Tá je veľmi dôležitá, pretože väčšinu času budú vektory uložené v určitej perzistentnej štruktúre. Formu tejto štruktúre dá klastering, ale samotné vyhľadávanie podobností musí zabezpečiť práve táto štruktúra.

Požiadavky na štruktúru pre uloženie dát sú nasledovné:

- Efektívne vkladanie nových dát, na základe aktuálneho rozloženia klastrov.
- Možnosti vyhľadávania podobných vektorov.
- Možnosť získať všetky dáta, ktoré sú potrebné pri reklasteringu.

Vzhľadom na to, že chceme na základe tejto dátovej štruktúry vyhľadávať plagiáty, a generovať reporty, je nutné, aby v sebe zahŕňala okrem samotných vektorov aj dodatočné informácie o daných vektoroch (odkaz na zdrojový súbor, pozíciu v súbore, informáciu o úlohe, z ktorej pochádza...). Špeciálne požiadavky na vyhľadávanie nemáme, štruktúra musí iba dokázať indexovať vybrané prvky vektorov.

Pre naše účely sme sa rozhodli zvoliť relačnú databázu, pretože tá nám umožní vytvoriť požadované typy indexov a efektívne vyhľadávať pomocou nich zhody a k samotným vektorom uložiť aj požadované doplňujúce informácie. Ďalšou užitočnou vlastnosťou relačných databáz sú aj možnosti ich škálovateľnosti. Využitie pesistenie pomocou relačnej databázy nám okrem toho umožňuje veľmi efektívnu implementáciu algoritmu na vyhľadávanie podobností, nakoľko väčšina práce pre vyhľadávanie vektorov je vykonaná už pri vkladaní dát do databázy, a pri generovaní plagiátov dokážeme tieto dáta veľmi efektívne z databázy získať.

6. Vyhľadávanie zhodných častí zdrojových kódov

Poslednou časťou našej navrhovanej metódy na vyhľadávanie plagiátov v zdrojovom kóde je vyhľadávanie podobných častí zdrojového kódu. V tejto kapitole sa venujeme algoritmu, pomocou ktorého dokážeme z databázy, ktorú sme si pripravili pomocou klasteringu, vyhľadať zhodné vektory, prefiltrovať ich, pospájať a vygenerovať report. Okrem samotného algoritmu sa venujeme aj problémom s vyhľadávaním plagiátov, ktoré sme popisovali v predchádzajúcich kapitolách.

6.1 Algoritmus

Základom nášho algoritmu na vyhľadávanie plagiátov, je relačná databáza. Nevýhodou takéhoto návrhu je to, že sme schopní vygenerovať report len pre práce, ktoré vopred pridáme do databázy. Tento prístup ale z pohľadu navrhovanej metódy je úplne v poriadku. Naším cieľom je, aby každý kód, ktorý raz do databázy pridáme, mohol byť ďalej používaný pri vyhľadávaní plagiátov. Na rozdiel od bežne používaných algoritmov, navrhovaný algoritmus nebude vyhľadávať plagiáty v celej množine dát, ale umožní získať zoznam zhôd pre jednu konkrétnu prácu.

Dva fragmenty zdrojového kódu budú označené ako zhodné pokiaľ ich charakteristické vektory sú zhodné. To znamená, že len kód, ktorý má rovnakú štruktúru, bude považovaný na základe nášho algoritmu za plagiát.

Algoritmus vyhľadávania plagiátov pozostáva z niekoľkých častí. Prvou z nich je získanie podobností z databázy, druhou je spájanie a filtrovanie týchto podobností, a v tretej časti sa pre odhalené dvojice prác spočíta miera podobnosti.

Prvou fázou algoritmu na vyhľadávanie zhodných častí zdrojového kódu je algoritmus na vyhľadanie podobných vektorov. Vstupom do tohto algoritmu je identifikačné číslo úlohy, pre ktorú chceme nájsť zhodné vektory. Algoritmus okrem toho potrebuje prístup k dátam z klasteringu.

Pre lepšie spracovanie a následnú vizualizáciu je nutné nájdené zhody medzi dvoma úlohami dodatočne spracovať. Na to, aby sme mohli určiť, nakoľko sa jednotlivé zadania podobajú, potrebujeme získať disjunktné časti zdrojového kódu, v ktorých bola nájdená zhoda. Keďže vektory boli generované tak, aby sa prekrývali, tak v zozname zhôd nájdeme veľké množstvo vektorov, ktoré sa nejakým spôsobom prekrývajú. Cieľom druhej časti algoritmu bude pospájať tieto prekrývajúce sa časti do väčších disjunktných celkov. Každý vektor obsahuje informáciu o súbore, z ktorého pochádza a rozsahu, ktorý v danom súbore pokrýva.

Poslednou časťou celého algoritmu je výpočet miery podobností pre konkrétne dvojice prác.Podobnosť úloh A a B je vyjadrená pomocou vzorca:

$$Sim(A,B) = \frac{\sum_{m \in matches(A,B)} coverage(m_A)}{coverage(A)} \quad (1)$$

Je dôležité poznamenať, že takto definovaná podobnosť nie je symetrická t.j. $Sim(A,B) != Sim(B,A)$. Funkcia $coverage(A)$ vo vzorci reprezentuje množstvo zdrojového kódu, ktoré je pokryté vektormi (pod pojmom množstvo máme namysli počet znakov). Pri výpočte pokrytia si treba dať pozor na to, že nie každá časť zdrojového kódu musí byť pokrytá nejakým vektorom. Pre každú úlohu ale ostáva táto hodnota konštantná, takže pre efektívnosť algoritmu môžeme túto hodnotu prepočítať v procese pridávania práce do databázy. Čitateľ reprezentuje množstvo kódu, ktoré je obsiahnuté v nájdených zhodách.

6.2 Problémy pri vyhodnocovaní plagiátorstva

Pri hľadaní plagiátorstva nejde len o nejaké vyhľadanie podobných prác. Cieľom APS by malo byť, čo najviac uľahčiť prácu používateľovi. My sme sa rozhodli zamerať na dve oblasti – odstraňovanie nevýznamných zhôd, a predspracovanie alebo normalizáciu zdrojového kódu.

V každom zdrojovom kóde nájdeme nejaké časti, ktoré nie sú z pohľadu detekcie plagiátov významné. Ako príklad môžeme uviesť rôzne importy na začiatku zdrojového kódu, či automaticky generovaný kód pomocou vývojového prostredia. Odstránenie importov by sa dalo realizovať už pri spracovaní zdrojového kódu, ale naším cieľom bolo navrhnúť univerzálnejšiu metódu.

V súčasnosti nepoznáme automatizovaný spôsob, ktorý by umožnil určiť, ktorá časť zdrojového kódu je potrebná, a ktorá nie. Navrhnutý spôsob teda pozostáva z manuálnej anotácie niekoľkých vektorov, a potom na základe týchto označených vektorov vyberieme klastre, ktoré môžeme vylúčiť z procesu vyhľadávania plagiátov.

Manuálne označené časti zdrojového kódu sme využili na ďalšiu analýzu. Ešte pri návrhu metódy klastrovania sme predpokladali, že podobné vektory skončia v rovnakých klastroch. A jedným zo spôsobov využitia klasteringu malo byť aj roztriedenie dát na významné a nevýznamné časti.

Analyzovali sme podiel označených vektorov v klastroch. Označili sme klastre v ktorých podiel označených vektorov presiahol 1%. Jednotlivé klastre sme manuálne skontrolovali a overili, či skutočne obsahujú len nevýznamné vektory. Ukázalo sa, že väčšina označených klastrov skutočne obsahovala len označené nevýznamné vektory.

Vďaka tomuto prístupu sa nám podarilo zefektívniť vyhľadávanie plagiátov (lebo je nutné porovnávať menej vektorov) a zjednodušiť prácu používateľovi, pretože mu nebudú servírované pre neho nevýznamné zhody.

6.3 Overenie algoritmu

V tejto kapitole si popíšme, ako sme navrhnutý algoritmus, a v podstate aj celú metódu, overovali. Pri overovaní jednotlivých algoritmov sme používali študentské práce, ktoré vznikli na FRI v rokoch 2014 až 2018. Pri validácii metód, ktoré slúžia na vyhľadávanie sú dôležité dve kritériá:

- úplnosť,
- relevancia.

Pod pojmom **úplnosť** sa rozumie schopnosť algoritmu nájsť všetky zhody obsiahnuté v datasete. **Relevancia** zas hovorí to tom, aký je pomer medzi správne identifikovanými, v tomto prípade plagiátmi, a tými, ktoré algoritmus za plagiát označil, ale v skutočnosti o plagiát nejde. Pri systémoch na detekciu plagiátov sú tieto kritériá často subjektívne.

V práci sme porovnávali náš naverhnutý algoritmus so systémami MOSS a JPlag. Pri porovnaní so systémom MOSS sme dosiahli porovnateľné výsledky. Po detailom porovnaní môžeme povedať, že náš algoritmus lepšie spracováva časti súborov, ktoré neobsahujú priamo kód (voľné riadky, medzery...).

Druhým systémom, s ktorým sme náš algoritmus porovnávali, bol systém JPlag. Pre tieto účely sme si pripravili dataset, ktorý pozostával zo semestrálnych prác z predmetov *Informatika 1* a *Informatika 2*. Pri porovnávaní zo systémom JPlag sme generovali vektory pomocou systému DECKARD ale aj pomocou nami implementovaného algoritmu na tvorbu vektorov. Pri analýze sa v prípade systému DECKARD ukázalo, že nastal problém pri vektorizácii zdrojového kódu. Systém DECKARD nebol schopný spracovať všetky súbory. 12% súborov kompletne chýbalo a pre 10% súborov nebol vygenerovaný dostatočný počet vektorov. Tieto komplikácie znížili celkovú efektivitu algoritmu pri využití systému DECKARD.

Pri využití nami implementovaného algoritmu na vektorizáciu zdrojového kódu boli výsledky lepšie, ale ani ten nebol schopný detegovať úplne všetky plagiáty. V tomto prípade bolo problémom až príliš striktné reprezentovanie zdrojového kódu pomocou vektorov - v jednej z prác boli zo všetkých metód odstránené kľúčové slová `this`, čo spôsobilo, že algoritmus nebol schopný nájsť podobnosti.

7 Zhodnotenie výsledkov

Pri overovaní algoritmu sme využívali viacero prístupov. Overovali sme reálne študentské práce, ale aj pre tieto účely špeciálne vytvorené fragmenty zdrojového kódu.

Naše výsledky, ukazujú že sme v prípade tzv. „ctrl+c, ctrl+v“ plagiátov schopní nájsť plagiáty so 100% úspešnosťou ako v prípade študentských prác, tak aj rôznych testovaných fragmentoch. Pri teste týchto fragmentov sme používali fragmenty rôznej dĺžky. Testy ukázali, že schopnosť detegovať plagiáty závisí len od parametra minimálnej dĺžky vektora používaného pri vektorizácii zdrojového kódu. Nami testovaná hodnota „30“ dokázala pokryť všetky relevantné fragmenty. Takéto plagiáty sa ale bežne v študentských prácach nevyskytujú.

Častejšie sa stretávame s plagiátmi, ktoré prešli určitou modifikáciou. Základné techniky modifikácie – úprava komentárov, pridávanie / odoberanie bielych znakov, zmena názvu identifikátorov, náš algoritmus dokáže podobne ako v prechádzajúcom prípade odhaliť v 100% prípadov. Pre algoritmus totižto žiadna zo spomenutých modifikácií nepredstavuje zdroj informácií.

V prípade, že modifikácia mení štruktúru zdrojového kódu môžu nastať problémy. Ukázalo sa, že niektoré z nich ani nie sú potrebné, nakoľko samotná povaha navrhnutého algoritmu dokáže vyriešiť situácie, ktoré tieto metódy riešia. Na druhej strane, výsledky ukázali že aj niektoré zmeny v štruktúre – hlavne poprehadzovanie častí kódu dokáže náš algoritmus odhaliť. Pokiaľ študent prehadzuje väčšie bloky – napríklad metódy (bloky, ktoré sú pokryté samostatným vektorom), algoritmus dokázal tieto poprehadzované metódy správne identifikovať. Pokiaľ prehadzuje veľmi malé bloky – typicky jednotlivé výrazy (všetky bloky sú pokryté v jednom vektore), algoritmus taktiež dokáže spoľahlivo tieto prípady identifikovať. Problém nastáva niekedyvtedy, keď bloky, ktoré prehadzujeme, sú dostatočne malé, aby z nich nemohli vzniknúť samostatné vektory, ale dostatočne veľké na to, aby boli všetky pokryté jedným spoločným vektorom. Táto situácia sa dá riešiť rozličným nastavením príslušných parametrov.

Poslednou modifikáciou, o ktorej sme pri vyhodnocovaní uvažovali je pridávanie / odoberanie častí kódu. Tento typ modifikácie je najkomplikovanejší aj na vytvorenie pre študenta, ale aj na odhalenie z pohľadu algoritmu. Pri generovaní vektorov sa tieto generujú hierarchicky, t.j. aj malá zmena na najnižšej úrovni syntaktického stromu sa prejavuje na všetkých vektoroch. Hlavným komponentom, vďaka ktorému je možné odhaľovať aj tieto modifikácie, je fáza spájania vektorov pri generovaní charakteristických vektorov. Naše výsledky ale ukazujú, že ani tá nie je dostatočným riešením tohto problému. V prípade, že je zdrojový kód značne zanorený, nedokážeme efektívne potlačiť tieto typy modifikácií.

Pri hodnotení algoritmu neuvádzame výpočtovú náročnosť. V súčasnosti pokiaľ hľadáme všetky plagiáty v rámci veľkej skupiny prác je algoritmus pomalší v porovnaní so systémom JPlag. Na druhej strane, pokiaľ potrebujeme vyhodnotiť jednu prácu voči ostatným, ktoré už máme v databáze, je náš prístup rýchlejší. Pri návrhu algoritmu nebola našim cieľom jeho čo najefektívnejšia implementácia (optimalizovali sme len časti nevyhnutné pre potreby vyhodnocovania). Zameriavali sme sa hlavne na škálovateľnosť a spoľahlivosť detekcie. Okrem toho, náš algoritmus uchováva dáta v relačnej databáze, vďaka čomu bude na rozdiel od algoritmov ako JPlag, ktoré pracujú s dátami len v operačnej pamäti, pomalší. Hlavnými úzkymi hrdlami algoritmu sú vkladanie dát do databázy a reprezentácia už nájdených zhôd.

Pri možnostiach zlepšenia algoritmu vidíme dva základné smery. V prvom prípade je potrebné zlepšiť detekciu pri rôznych modifikáciách zdrojového kódu v snahe zabrániť odhaleniu plagiátu. Druhým smerom, ktorým je možné zlepšovať navrhnutý algoritmus, sú ďalšie možnosti detekcie nevýznamného kódu. V práci popisujeme nejaké základné postupy, ktoré sme použili. Okrem nich je možné nájsť v literatúre rôzne metódy založené na strojovom učení a umelej inteligencii, ktoré by bolo možné implementovať.

Po implementačnej stránke celého riešenia vidíme potrebu v efektívnejšej implementácii niektorých častí. Pre nasadenie tohto riešenia v akademickom prostredí bude potrebné implementovať aj vhodné grafické rozhranie. Medzi ďalšie možnosti rozširovania patria samozrejme pridanie ďalších programovacích jazykov. Táto požiadavka v princípe nepredstavuje problém, nakoľko na to bolo myslené pri návrhu.

Za hlavný prínos práce považujeme navrhnuté metódy, ale aj postupy nutné k vyhodnocovaniu plagiátorstva v zdrojovom kóde.

Použitá literatúra

[1] SKALKA, Ján. "Prevencia a odhaľovanie plagiátorstva." *Zber prác za účelom obmedzenia porušovania autorských práv v kvalifikačných prácach na vysokých školách* (2009).

[2] "Slovník slovenského jazyka" Dostupný online: http://slovnik.juls.savba.sk; dátum prístupu 21.01.2019

[3] "Oxford dictionary" Dostupný online: https://en.oxforddictionaries.com/definition/plagiarism; dátum prístupu 21.01.2019

[4] Hammond, Michael. "Cyber-plagiarism: are FE students getting away with words?." (2002).

[5] Veselý, Ondřej . "RESULTS OF SIMILARITY ANALYSIS OF ONLINE NEWS" (2015).

[6] Holbrook, Timothy R., and Lucas Osborn. "Digital patent infringement in an era of 3D printing." (2014).

[7] Curtis, Guy J., and Lucia Vardanega. "Is plagiarism changing over time? A 10-year time-lag study with three points of measurement." *Higher Education Research & Development* 35.6 (2016): 1167-1179.

[8] Kravjar J. "SK ANTIPLAG IS BEARING FRUIT." (2015)

[9] Benko, Juraj, and Elena Gogoláková. "PLAGIÁTORSTVO VO VEDE, ETICKÉ ŠTANDARDY A AUTORSKÝ ZÁKON." *Historický časopis (Historical Journal)* 4.64 (2016): 645-667.

[10] LETTERMAN, D. "Top Ten Reasons Students Plagiarize & What You Can Do." *Writing News* (2006).

[11] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. "On the naturalness of software. In Proceedings" *International Conference on Software Engineering, volume 20, pages 837-847, 2012.*

[12] Jiang, Lingxiao, et al. "Deckard: Scalable and accurate tree-based detection of code clones." *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007.

Zoznam vlastných publikácií

[1] ***Using concepts of text based plagiarism detection in source code plagiarism analysis*** / Ďuračík Michal (34%) - Kršák Emil (33%) - Hrkút Patrik (33%). In: Plagiarism across Europe and beyond 2017 : conference proceedings : May 24-26, 2017 Brno, Czech Republic. - Brno: [Mendel University], 2017. - ISBN 978-80-7509-493-3. - S. 177-186.

[2] ***Current trends in source code analysis, plagiarism detection and issues of analysis big datasets*** / Ďuračík Michal (34%) - Kršák Emil (33%) - Hrkút Patrik (33%). In: Procedia Engineering [elektronický zdroj]. - ISSN 1877-7058. - Vol. 192 (2017), online, s. 136-141.

[3] ***Source code representations for plagiarism detection [print]*** / Ďuračík Michal (34%) - Kršák Emil (33%) - Hrkút Patrik (33%). In: Learning technology for education challenges [print, electronic] : proceedings. - 1. vyd. - Cham: Springer International Publishing AG, 2018. - ISBN 978-3-319-95521-6. - s. 61-69 [print, online].

[4] ***Issues with the detection of plagiarism in programming courses on a larger scale [electronic]*** / Ďuračík Michal (34%) - Kršák Emil (33%) - Hrkút Patrik (33%). In: ICETA 2018 : Proceedings : 16th IEEE International Conference on Emerging eLearning Technologies and Applications. - New Jersey: Institute of Electrical and Electronics Engineers. - ISBN 978-1-5386-7912-8. - s. 141-147 [print].

[5] ***Scalable source code plagiarism detection using source code vectors clustering [print]*** / Ďuračík Michal (34%) - Kršák Emil (33%) - Hrkút Patrik (33%). In: Proceedings of 2018 IEEE 9th International Conference on Software Engineering and Service Science [print, electronic]. - ISSN 2327-0586. - 1. vyd. - Danvers: Institute of Electrical and Electronics Engineers, 2018. - ISBN 978-1-5386-6564-0. - s. 499-502 [print, online, CD-ROM].

[6] ***Semi-automatic identification of non-significant source code parts using clustering [print]*** / Ďuračík Michal (100%). In: Mathematics in science and technologies : proceedings of the MIST conference 2019. - [S.l.]: [s.n.]. - ISBN 9781794002180. - s. 17-21 [print]