ŽILINSKÁ UNIVERZITA V ŽILINE
FAKULTA RIADENIA A INFORMATIKY

Algoritmy pre identifikáciu plagiátov zdrojových kódov

DIZERTAČNÁ PRÁCA

2019 **Ing. Michal Ďuračík**

ŽILINSKÁ UNIVERZITA V ŽILINE
FAKULTA RIADENIA A INFORMATIKY

Algoritmy pre identifikáciu plagiátov zdrojových kódov

DIZERTAČNÁ PRÁCA

28360020193005

Študijný program:	Aplikovaná informatika
Študijný odbor:	9.2.9 Aplikovaná informatika
Pracovisko:	Katedra softvérových technológií
	Fakulta riadenia a informatiky
	Žilinská univerzita v Žiline
Školiteľ:	doc. Ing. Emil Kršák, PhD.
Školiteľ špecialista:	Ing. Patrik Hrkút, PhD.

Žilina, 2019 **Ing. Michal Ďuračík**

Anotácia

Typ práce:	Dizertačná práca
Akademický rok:	2018/2019
Názov práce:	Algoritmy pre identifikáciu plagiátov zdrojových kódov
Autor:	Ing. Michal Ďuračík
Školiteľ:	doc. Ing. Emil Kršák, PhD.
Školiteľ špecialista:	Ing. Patrik Hrkút, PhD.
Jazyk:	Slovenčina
Počet strán:	143
Počet obrázkov:	60
Počet tabuliek:	9
Počet referencií:	70
Kľúčové slová:	plagiátorstvo, zdrojový kód, detekcia plagiátorstva, charakteristické vektory

Poďakovanie

Touto cestou by som sa chcel poďakovať za pomoc, odborné vedenie, cenné rady, nápady a pripomienky doc. Ing. Emilovi Kršákovi, PhD a Ing. Patrikovi Hrkútovi, PhD., ktoré mi venovali pri vypracovaní mojej dizertačnej práce.

Zároveň by som sa chcel veľmi pekne poďakovať všetkým kolegom a rodine za konzultácie pri riešení rôznych problémov, podporu a spríjemňovanie pracovného prostredia.

V neposlednom rade je potrebné spomenúť aj všetkých plagiátorov, vďaka ktorým mohla vzniknúť táto práca.

Abstrakt

ĎURAČÍK, Michal: *Algoritmy pre identifikáciu plagiátov zdrojových kódov.* [Dizertačná práca] Žilinská univerzita v Žiline. Fakulta riadenia a informatiky. Katedra softvérových technológií. - Vedúci dizertačnej práce: doc. Ing. Emil Kršák, PhD. – Žilina: FRI ZU, 2019, 143 s.

Táto dizertačná práca sa zaoberá problematikou plagiátorstva v zdrojovom kóde. Tejto oblasti sa v súčastnosti nevenuje až taká pozornosť, ako plagiátorstvu v textových prácach. V práci vychádzame zo skúsenosti s bojom voči plagiátorstvu na Fakulte riadenia a informaiky, Žilinskej univerzite v Žiline. Cieľom práce je návrh metódy, ktorá umožní vyhľadávanie plagiátov vo veľkom rozsahu, podobne, ako je to bežné pri textových dokumentoch. Pri návrhu tejto metódy využívame znalosti z oblasti vyhľadávania plagiátov v textových dokumentoch. Práca popisuje možnosti spracovania a reprezentácie zdrojového kódu. Na reprezentáciu zdrojového kódu využívame charakteristické vektory, ktoré následne organizujeme pomocou navrhnutej metódy inkrementálneho klasteringu. Na základe klasteringu vytvárame databázu zdrojového kódu, voči ktorej následne overujeme jednotlivé zdrojové kódy. Výsledkom navrhnutej a implementovanej metódy je systém, ktorý umožnuje vyhľadávanie plagiátov v zdrojovom kóde vo väčšom meradle, ako bolo so súčasnými nástrojmi možné. Práca sa okrem návrhu metódy venuje aj overovaniu jednotlivých jej fáz a porovnaniu dosiahnutých výsledkov s bežne, v tejto oblasti, používanými nástrojmi.

Kľúčové slová: plagiátorstvo, zdrojový kód, detekcia plagiátorstva, charakteristické vektory

Abstract

ĎURAČÍK, Michal: *Algorithms for source code plagiarism identification.* [Dissertation thesis] - University of Žilina. Faculty of Management Science and Informatics. Department of Software Technology. - Supervisor: doc. Ing. Emil Kršák, PhD. - Žilina, FRI ZU, 2019, 143 p.

This dissertation thesis deals with the issue of plagiarism in the source code. At the moment, this area is not given as much attention as looking for a plagiarism in a text works. The work is based on the experience with the fight against plagiarism at the Faculty of Management science and Informatics, University of Žilina. The aim of the thesis is to design a method, that allows for large-scale plagiarism searches, similar to methods used in the text documents. When designing this method, we use the knowledge of plagiarism in text documents. The work describes the possibilities of processing and representation of source code. We use characteristic vectors to represent the source code, which we then organize using the proposed incremental clustering method. Based on clustering, we create a source code database, against which we then verify individual source code fragments. The proposed and implemented method results in a system, that allows plagiarism to be searched in the source code on a larger scale, than was possible with current tools. In addition to the method design, the work also deals with the verification of its individual phases and the comparison of the achieved results with commonly used tools in this area.

Key words: plagiarism, source code, plagiarism detection, characteristic vectors

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam skratiek

APS	Antiplagiátorsky systém
AST	Abstraktný syntaktický strom
CPU	Procesor
CRZP	Centrálny register záverečných prác
DTM	Document-term matrix
FRI	Fakulta riadenia a Informatiky
HLC	Vysoko – úrovňové klony
LDA	Latent Dirichlet allocation
MOSS	Measure Of Software Similarity
NLP	Natural language processing
UNIZA	Žilinská Univerzita v Žiline

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. V prípade zdrojového kódu tomu napomáha aj Open-source kód (otvorený zdrojový kód), ktorý umožňuje programátorom programovať, zdieľať a distribuovať svoj kód bez obmedzení. Táto jednoduchosť nabáda študentov ku kopírovaniu týchto voľne dostupných zdrojov do svojich prác.

Problémom odhaľovania plagiátov v zdrojovom kóde je z nášho pohľadu predovšetkým to, že na rozdiel od textových dokumentov v súčasnosti neexistuje žiadny voľne dostupný komplexný systém alebo služba, ktorá by umožňovala detekciu plagiátorstva v globálnom meradle. Vďaka tomu nie sú k dispozícii ani žiadne globálne štatistiky, ktoré by popisovali mieru plagiátorstva v tejto oblasti.

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 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. Pozrieme 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

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

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.

Pokiaľ budeme identifikovať plagiáty len na základe podobnosti algoritmov a štruktúr, môžeme naraziť na ďalší problém. Tým problémom je, že pri programovaní sa často používajú naučené a overené princípy, ktoré vedú ku štrukturálne rovnakému kódu, bez ohľadu na programátora. Ako príklad môžeme uviesť návrhové vzory, ktoré často predpisujú presnú štruktúru objektov alebo algoritmov, pomocou ktorých môžeme vyriešiť bežné programátorské úlohy. Pri vyhodnocovaní je nutné s týmto javom počítať, či už pri strojovom spracovaní alebo aj pri manuálnom vyhodnocovaní.

Posledným problémom, ktorý v tejto časti spomenieme, je využívanie zdrojového kódu, ktorého licencia to umožňuje. V rámci zdrojového kódu existujú rôzne licencie, ktoré umožňujú jeho použitie aj v cudzích programoch bez uvedenia pôvodného zdroja či licencie. Z hľadiska autorského zákona je takéto použitie úplne v poriadku. V rámci akademickej sféry, ale toto nemusí byť akceptovateľné. Využitie voľne dostupného zdrojového kódu je študentom zvyčajne dovolené pod podmienkou jeho označenia v zdrojovom kóde. Problémom je, že týmto zavádzame určité dodatočné pravidlá nad autorský zákon. V rámci akademickej praxe sa ukázalo, že toto označovanie nie je vždy nutné. To, či je potrebné označiť prevzatý kód, závisí od typu úlohy. Pokiaľ je úlohou študenta naprogramovať konkrétny algoritmus alebo skupinu algoritmov, je potrebné každý prevzatý kód označovať, aby bol jasný podiel študenta na celkovom riešení. Na

druhej strane máme predmety, ktorých cieľom je tvorba komplexnejších riešení, ktoré sa často skladajú z rôznych menších komponentov a cieľom študenta je správne pospájať jednotlivé komponenty. V tomto prípade nie je nutné zvlášť označovať každý komponent a jeho časti, pretože v tejto fáze sa už nehodnotí schopnosť študenta programovať, ale navrhovať a spájať malé časti do väčších celkov.

V nasledujúcej časti si ukáže zopár príkladov, ktoré demonštrujú vyššie popísané problémy. V prvom prípade na obrázku 1 môžeme vidieť ukážku dvoch častí zdrojového kódu, ktoré sa líšia len v názve identifikátorov, komentároch a v poradí niektorých príkazov. V tomto konkrétnom prípade niet pochýb o plagiátorstve.

```
var nastavec = 0;
var prepinac = 0;
if (TextKrp.Text != "")
{
    if (int.TryParse(TextKrp.Text, out var result))
        {
            prepinac = 1;
            nastavec = result;
        }
        else
        {
            MessageBox.Show("KRP ID must be a number", "Error",
                MessageBoxButton.OK, MessageBoxImage.Error);
            return;
        }
    }
else
{
    prepinac = -1;
    nastavec = 0;
}
```

Kód A

```
var adapt = 0;
var swtch = 0;
if (tbKRP.Text != "")
{
    //Skusim nacitat hodnotu
    if (int.TryParse(tbKRP.Text, out var res))
    {
        adapt = res;
        swtch = 1;
    }
    else
    {
        MessageBox.Show("KRP is not a number.", "Error!!!",
        MessageBoxButton.OK, MessageBoxImage.Error);
        return;
    }
}
else
{
    adapt = 0;
    swtch = -1;
}
```

Kód B

Obrázok 1: Porovnanie dvoch kódov so zmenenými identifikátormi

V druhom prípade na obrázku 2 vidíme ukážku kódu, ktorý spracováva parametre z príkazového riadku. Ako môžeme vidieť, rozdiel medzi kódom A a kódom B je len v tom, že jeden používa konštanty a má správne ošetrené nedefinované prípady, a druhý používa priamo literály v zdrojovom kóde. Po funkčnej stránke sú oba zdrojové kódy takmer identické. Takáto zhoda ale nemusí vždy znamenať, že sa jedná o plagiát.

```
if (lastSwitch != null
    && lastSwitch.Equals(Constants.Clubs))
{
    clubsAdress = element;
}
else
{
    if (lastSwitch != null
        && lastSwitch.Equals(Constants.Players))
    {
        playersAdress = element;
    }
    else
    {
        Console.WriteLine(@"Invalid argument");
    }
}
```

Kód A

```
if (lastSwitch.Equals("-clubs"))
{
    clubsAdress = element;
}
else
{
    if (lastSwitch.Equals("-players"))
    {
        playersAdress = element;
    }
    else
    {
        Console.WriteLine("Invalid argument");
    }
}
```

Kód B

Obrázok 2: Porovnanie dvoch kódov s malou modifikáciou

```
if (lastSwitch.Equals("-clubs"))
{
    clubsAdress = element;
}
else
{
    if (lastSwitch.Equals("-players"))
    {
        playersAdress = element;
    }
    else
    {
        Console.WriteLine("Invalid argument");
    }
}
```

Kód A

```
if (cargo.HasItem("car"))
{
    weight = carWeight;
}
else
{
    if (cargo.HasItem("boat"))
    {
        weight = boatWeight;
    }
    else
    {
        Container.SetDesc("Empty container");
    }
}
```

Kód B

Obrázok 3: Porovnanie dvoch kódov s rovnakou štruktúrou

Na obrázku 3 môžeme vidieť veľmi podobnú situáciu ako bola na obrázku 1. Oba zdrojové kódy majú rovnakú štruktúru a zmenené názvy identifikátorov. Pri analýze ale dospejeme k názoru, že `Kód A` reprezentuje iný algoritmus ako `Kód B` aj napriek tomu, že zo štrukturálneho pohľadu sú rovnaké.

```
public void PrintMaximum(int[] ints)
{
    var max = int.MinValue;
    foreach (var i in ints)
    {
        if (i > max) max = i;
    }
    Console.WriteLine(max);
}
```

Kód A

```
public void PrintMaximum(int[] ints)
{
    var max = ints
        .OrderByDescending(i => i)
        .FirstOrDefault();
    Console.WriteLine(max);
}
```

Kód B

Obrázok 4: Porovnanie kódov s rovnakou funkčnosťou a rozdielnou štruktúrou

```
public void PrintMaximum(int[] ints)
{
    var max = ints
        .OrderByDescending(i => i)
        .FirstOrDefault();
    Console.WriteLine(max);
}
```

Kód A

```
public void PrintMaximum(int[] ints)
{
    int max = Arrays.stream(ints)
      .max()
      .getAsInt();
    System.out.println(max);
}
```

Kód B

Obrázok 5: Porovnanie kódov v rozdielnom programovacom jazyku

Posledným príkladom sú algoritmy, ktoré na vonkajší pohľad robia to isté, ale každý je implementovaný iným spôsobom, t.j. má inú vnútornú štruktúru. Obrázky 4 a 5 ukazujú rovnaký algoritmus implementovaný v prvom prípade rozdielnym spôsobom a v druhom prípade dokonca v rozdielnom programovacom jazyku (Kód A je v C# a Kód B v Jave).

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 [10][11] existuje niekoľko príčin vzniku plagiátorstva. Medzi najčastejšie zaraďujeme [12]:

- 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 (viac v sekcii 1.1.3). 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.

Využitie APS na zdrojový kód má podľa nášho názoru viac výhod ako využívanie APS pri textových dokumentoch. Zavedenie týchto systémov má predovšetkým pozitívny vplyv na morálku. Na druhej strane, vždy budú existovať študenti, ktorí si budú chcieť prácu uľahčiť a nájsť slabiny APS. Súčasné nástroje dokážu celkom spoľahlivo odhaliť základné modifikácie, ktoré boli ukázané na obrázkoch 1 a 3. Na druhej strane komplikovanejšie modifikácie, ako je napríklad na obrázku 4, alebo prepis algoritmu do iného programovacieho jazyka (obrázok 5) vyžadujú od študenta istú dávku vedomostí a času. Je samozrejme na zváženie, či takéto sofistikovanejšie úpravy zdrojového kódu považovať za plagiát. Podľa nášho názoru toto závisí od prípadu k prípadu, ale zvyčajne ak

študent dokáže v takejto miere modifikovať zdrojový kód, nemali by sme takto modifikovanú prácu považovať za plagiát.

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. Študenti v rámci týchto predmetov musia samostatne vypracovať semestrálnu prácu, vďaka ktorej preukážu svoje schopnosti. Tému semestrálnej práce si študenti volia samostatne, takže sa očakáva veľká variácia jednotlivých riešení. Ďalším predmetom, na ktorom sme spustili vyhľadávanie plagiátov je predmet *Algoritmy a údajové štruktúry*. Tento predmet je taktiež ideálnym kandidátom, pretože študenti nemajú dovolené používať cudzí kód, a sú hodnotení za vlastný kód a algoritmy. Tento predmet sa odporúča v 4. semestri štúdia na bakalárskom stupni a je povinný pre všetkých študentov študijného programu informatika. Posledným predmetom, na ktorom sme analyzovali plagiáty, bol predmet *Pokročilé objektové technológie*. Tento predmet zaraďujeme do inžinierskeho stupňa štúdia a je nastavený ako povinne voliteľný. Rozdiel tohto predmetu oproti predošlým trom je v tom, že študenti už nie sú povinní programovať všetky algoritmy od základu, ale celý predmet sa zaoberá tvorbou aplikácií s využitím rôznych .net technológií.

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

1. Zozbierali sme všetky práce.
2. Spustili sme analýzu plagiátorstva.
3. 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.

Podľa nás je táto kontrola v predmetoch prvého ročníka veľmi dôležitá. Tým dávame študentom od začiatku najavo, že plagiátorstvo u nás nemá miesto. Tento krok je aj dôležitý pri motivácii študentov, ktorí by odovzdali plagiát z dôvodu, že to tak robia všetci alebo z lenivosti. Rozšírenie tejto kontroly stojí na ochote vyučujúcich, pretože analýza a vyhodnocovanie plagiátov stojí určitý čas, a je len na nich, či ho budú ochotní tomu venovať.

Ďalším pozitívnym prínosom, ktorý mala táto iniciatíva je, že všetci novo-nastupujúci študenti dostávajú v rámci prípravného kurzu informácie o plagiátorstve a ako sa mu vyvarovať. Okrem toho čoraz viac vyučujúcich informuje o tomto probléme svojich študentov priebežne počas celého štúdia. Pre zníženie prípadov pokusu o plagiát začiatkom

školského roka 2017/2018 prebehla na celofakultnej úrovni letáková kampaň, ktorá sa snažila upozorniť študentov na plagiátorstvo.

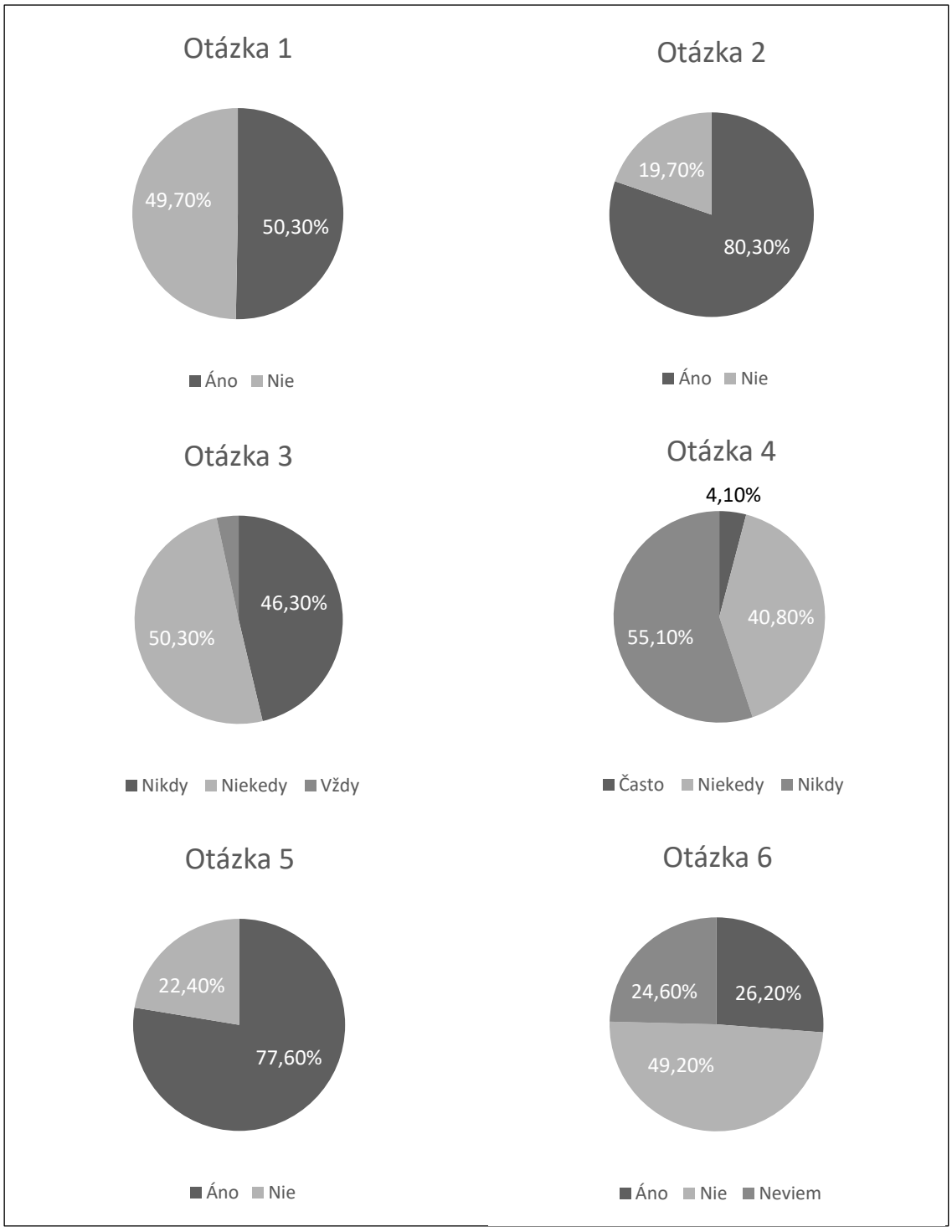
1.1.4 Kopírovanie zdrojového kódu študentami FRI

Jednou z príčin vzniku plagiátorstva je aj nevedomosť študentov o tomto probléme. Pri programovaní je bežné, že programátor hľadá na internete rôzne rady. Tieto rady sú často dostupné na stránkach typu *stackoverflow*[3] vo formáte úryvku kódu, ktorý rieši daný problém. Okrem nej je možné nájsť na internete veľké množstvo tutoriálov alebo rôznych ukážok, ktoré môžu, alebo nemusia podliehať voľnej licencii. Pokiaľ študent skopíruje z internetu určitú časť zdrojového kódu, môže sa dopustiť plagiátorstva. Často má takéto bezhlavé kopírovanie zdrojového kódu aj iné dôsledky. Existujú výskumy [13], ktoré poukazujú na to, že kopírovanie zdrojového kódu z internetu vedie napríklad k rôznym bezpečnostným problémom.

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. V prieskume sme sa respondentov pýtali na nasledujúce otázky:

1. Skopírovali ste niekedy časť zdrojového kódu (alebo celý) od spolužiaka, ktorý ste potom použili vo svojej domácej úlohe alebo semestrálnej práci?
2. Skopírovali ste niekedy do svojho programu kód, či už počas štúdia, alebo aj mimo, ktorý pochádzal z internetu (StackOverflow, Codecademy...)?
3. Pri kopírovaní zdrojového kódu z internetu označíte príslušnú časť a zdroj, z ktorého pochádza?
4. Kopírujete zdrojový kód, o ktorom si nie ste istý, čo robí?
5. Okrem "štandardnej" funkcionality (kód robí to, čo potrebujem) sa snažíte zistiť, čo presne všetko tento kód robí, ako to robí a aké má nedostatky?
6. Považujete kopírovanie cudzieho zdrojového kódu do vlastnej práce za plagiátorstvo?
7. Akým spôsobom váš postoj ovplyvnila minuloročná kampaň proti plagiátorstvu na FRI?

[3] https://stackoverflow.com



Obrázok 6: Výsledky prieskumu

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.

Odpoveď	Podiel
Kód síce kopírujem naďalej, ale snažím sa ho modifikovať aby to nebol plagiát	27,70%
Túto kampaň som nepostrehol	25,60%
Nikdy som neskopíroval kód, takže kampaň len potvrdila moje presvedčenie	22,60%
Kopírujem kód ako predtým	20,20%
Už kód nekopírujem pretože to považujem za plagiátorstvo	3,90%

Tabuľka 2: Podiel odpovedí na 7. otázku dotazníka

Posledná otázka v prieskume mala za cieľ získať spätnú väzbu študentov na kampaň proti plagiátorstvu, ktorá prebiehala počas akademického roka 2017/2018. Ako ukazujú výsledky, kampaň mala priamy pozitívny vplyv len na malú časť respondentov. Na druhej strane aj to, že študenti budú kopírovať ďalej, ale budú sa snažiť skopírovaný kód modifikovať, má svoj prínos. V predchádzajúcich kapitolách sme konštatovali že, na to, aby študent dokázal zmodifikovať kód tak, aby prešiel kontrolou v APS, musí mať dostatočné schopnosti a často takto zmodifikovaná práca už nemôže byť považovaná za plagiát.

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.

1.2 Štruktúra práce

V práci sa budeme zaoberať návrhom metódy a jej komponentov na odhaľovanie plagiátorstva v zdrojovom kóde. V prvej kapitole sme uviedli motiváciu, ktorá nás k riešeniu tejto problematiky viedla. Samotné riešenie rozoberáme od druhej kapitoly.

V kapitole 2 budeme analyzovať súčasný stav danej problematiky. Porovnáme dostupné metódy na vyhľadávanie plagiátov v zdrojovom kóde a v textových dokumentoch, ktoré považujeme za príbuznú oblasť.

Hlavné jadro práce – návrh metódy na vyhľadávanie plagiátov, je obsiahnutý v kapitole 3. Jednotlivé fázy tohto algoritmu sú popísané v kapitolách 4 až 6. V práci v kapitole 4 sa venujeme návrhu algoritmov na spracovanie zdrojového kódu. Skúmame dostupné metódy a vyhodnocujeme ich vhodnosť pre metódu vyhľadávania plagiátov v kapitole 4.1.3. Práca pokračuje metódami klasterizácie zdrojového kódu v kapitole 5. Kapitola popisuje využitie K-Means klasteringu na zdrojový kód, skúma rôzne charakteristiky, na základe ktorých navrhujeme metodiku nastavovania parametrov tohto klasteringu. V poslednej časti v kapitole 6 sa venujeme vyhodnocovaniu plagiátov a overeniu navrhnutých algoritmov.

V poslednej kapitole zhodnotíme dosiahnuté výsledky a popíšeme možnosti rozšírenia navrhnutých algoritmov. V prílohe sa nachádzajú rôzne ukážky plagiátov, ktoré tento algoritmus dokáže, respektíve nedokáže detegovať.

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.

2.1 Charakteristika

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.

2.2 Spracovanie a reprezentácia

Prvým krokom je spracovanie vstupných dát. Rozdiely medzi textovými dokumentmi a zdrojovým kódom existujú, a preto sa aj ich spracovanie pre potreby hľadania podobností mierne líši. Je zrejmé, že zdrojový kód sa spracováva v porovnaní s textovými dokumentmi jednoduchšie, pretože je priamo určený na spracovanie počítačom.

Spracovanie textových dokumentov môžeme rozdeliť na 5 základných fáz:

- prevod dokumentu na čistý text,
- tokenizácia,
- odstránenie stop slov,
- stemming and lematizácia,
- reprezentácia dokumentu.

Nástroje, ktoré spracovávajú textové dokumenty majú na vstupe často dokumenty v rôznych formátoch (pdf, docx...). Tieto dokumenty obsahujú okrem samotného textu množstvo dodatočných informácií, formátovanie ... Vo väčšine prípadov je prvým krokom **prevod do tzv. čistého textu**. Pri tomto procese sa z dokumentu extrahuje významný text. V rámci tejto fázy sa často vykonáva aj normalizácia – transformácia textu do základnej formy.

Po prevode dokumentu na text nasleduje **tokenizácia**. Tá ma za úlohu rozdeliť text (reprezentovaný ako prúd bajtov) na významové časti – tokeny. Token je arbitrálna jednotka textu, ktorá rozširuje lingvistický význam pojmu slovo [14]. Základným pravidlom je, že akýkoľvek reťazec znakov medzi dvoma medzerami sa považuje za token.

Po tokenizácii sa niekedy odstránia tzv. **stop slová**. Stop slová sú často používané slová, ktoré vďaka svojmu častému výskytu neprinášajú pri analýze relevantné informácie. Úplne najbežnejším príkladom je v prípade angličtiny slovo „The“, ktoré sa nachádza takmer v každej vete dokumentu.

Ďalším krokom je **stemming a lematizácia**. Obe majú za úlohu zjednotiť rozličné tvary slov. Napríklad „včela“, „včely“, „včele“, „včelám“ do základného tvaru - „včela“. Stemming je jednoduchší prístup, ktorý pomocou orezávania predpôn a prípon hľadá koreň slova. Lematizácia využíva morfologickú analýzu na určenie základného slova, z ktorého bolo analyzované slovo odvodené.

Posledným krokom je **reprezentácia dokumentu**. Pri reprezentácii sa využívajú rôzne modely (bag of words, vector model), alebo sa dokument reprezentuje pomocou n-gramov.

Spracovanie zdrojového kódu je veľmi podobné. Hlavný rozdiel medzi zdrojovým kódom a textovým dokumentom je v ich štruktúre. Štruktúra zdrojového kódu je naproti textovým dokumentom rozmanitejšia. Na rozdiel od textu, kde je význam ukrytý v

jednotlivých slovách a ich kombinácii, je význam zdrojového kódu skrytý v samotnej štruktúre.

Spracovanie zdrojového kódu môžeme rozdeliť na tieto kroky:

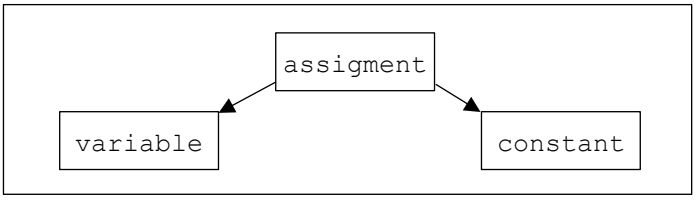
- tokenizácia
- čistenie
- reprezentácia zdrojového kódu

Keďže zdrojový kód sa píše ako čistý text bez formátovania, tak na rozdiel od textových dokumentov nie je potrebné prvotné spracovanie, ale rovno sa vykonáva tokenizácia. Na rozdiel od textu, zdrojový kód môže obsahovať aj časti, ktoré sú pre jeho funkčný význam nepodstatné. Tieto časti sa nazývajú komentáre a môžu obsahovať dodatočné informácie. Komentáre zvyčajne slúžia len pre programátora, nie pre program samotný. Keďže sú ale komentáre súčasťou gramatiky jazyka, ich odstránenie je pomerne jednoduché a zvyčajne sa vykonáva pri tokenizácii.

Tokenizácia zdrojového kódu je náročnejšia ako pri texte. Pri zdrojovom kóde nie sú za tokeny považované jednotlivé slová, ale sekvencie znakov, ktoré majú podľa gramatiky jazyka určitý význam. Navyše, výstup z procesu tokenizácie netvorí zoznam týchto sekvencií. Tieto sekvencie sú nahradené príslušnými identifikátormi na základe ich významu v rámci gramatiky príslušného jazyka. Napríklad výraz `a = 5` môže byť spracovaný na tokeny: *variable, assignment_operator, constant*. Pre porovnanie, ak by sme ho spravovali pomocou textového tokenizátora dostali by sme tri tokeny: *a, =, 5*

Po tokenizácii je potrebné zdrojový kód určitým spôsobom reprezentovať. Keďže sa jedná o tokeny, je možné použiť reprezentácie ako pri textových dokumentoch (bag of words, vector model...). Ako sme už ale uviedli, veľká časť informácie o zdrojovom kóde je ukrytá v jeho štruktúre. Pri už spomenutých reprezentáciách o túto informáciu prichádzame. Táto štruktúra nie je dôležitá len pre potreby analýzy, ale využívajú ju aj programovacie jazyky samotné. Niektoré programovacie jazyky si vystačili pri kompilácii s týmito tokenami, ktoré takmer 1:1prekladali do spustiteľného kódu. Postupom času, ako sa programovacie jazyky zdokonaľovali, ich gramatika sa stávala komplikovanejšou, prišli nové spôsoby reprezentácie zdrojového kódu. Základnou z nich je syntaktický strom.

Syntaktický strom usporadúva tokeny do stromu na základe ich štruktúry odvodenej od gramatiky jazyka. Niekedy pridáva ďalšie informácie o kontexte daného kódu. Náš príklad s výrazom `a = 5` môžeme vidieť v podobe syntaktického stromu na obrázku **7.**



Obrázok 7: Zjednodušená ukážka syntaktického stromu

Tieto stromy sa veľmi často využívajú na programovú modifikáciu zdrojového kódu napríklad vo vývojových prostrediach. Okrem syntaktických stromov sa používajú aj abstraktné syntaktické stromy (AST). Ako aj názov napovedá, sú abstraktné a popisujú význam daného kódu nezávisle od konkrétnej syntaxe. AST sa zvyčajne konštruujú zo syntaktických stromov.

2.3 Analýza

Doteraz sme popisovali len spôsob spracovania textových dokumentov a zdrojového kódu. Toto spracovanie je dôležité najmä pre to, aby sme vôbec tieto súbory vedeli programovo analyzovať. Pri hľadaní podobností medzi dokumentmi sa ako základ využíva niektorá z už spomenutých reprezentácií. Na základe použitej reprezentácie môžeme metódy rozdeliť na dve hlavné skupiny.

V prvej nájdeme metódy, ktoré používajú niektorú zo štatistických reprezentácií dokumentu (bag of words, vector model...). Výhodou týchto metód je to, že pracujú s reprezentáciou dokumentu, ktorá je zvyčajne výrazne menšia ako pôvodný dokument. Nevýhodou je to, že výsledkom porovnania je len „číslo“, ktoré určuje vzdialenosť medzi dokumentami (Euklidovská vzdialenosť, Jaccardovský koeficient...) [15].

V druhej skupine nájdeme metódy, ktoré pracujú s plnou reprezentáciou dokumentu (reprezentácie založené napríklad na n-gramoch). Ich hlavná výhoda je, že dokážu nájsť konkrétne časti, ktoré sa zhodujú. Naopak nevýhodou je pamäťová a výpočtová náročnosť, ktorá býva vyššia.

Často sa obe tieto skupiny kombinujú. Najskôr sa využíva niektorá metóda z prvej skupiny na klastrovanie dokumentov. Toto klastrovanie zvyčajne zmenší množstvo

dokumentov na detailné porovnávanie. Následne sa použije niektorá metóda z druhej skupiny za účelom nájdenia konkrétnych podobností medzi dokumentmi.

Existuje množstvo prác, ktoré sa venujú skúmaniu podobností v textových dokumentoch alebo v zdrojovom kóde. Zoznam metód používaných na určovanie podobností medzi textovými dokumentmi môžeme nájsť popísaný v práci *A Survey of Text Similarity Approaches* [16]. Autori sa okrem štandardných metód pracujúcich s tokenami venujú aj iným metódam. Tie ale v kontexte zdrojového kódu veľký zmysel nemajú, pretože používajú rôzne korpusy určené pre výskum jazyka. V rámci zdrojového kódu je používaných tiež niekoľko metód. Prehľad týchto metód môžeme nájsť v práci *Source Code Plagiarism Detection 'SCPDet': A Review* [17]. Autori v nej okrem jednotlivých metód popisujú aj hotové nástroje na odhaľovanie plagiátorstva v zdrojových kódoch.

2.3.1 Klastrovanie dokumentov

Klastrovacie algoritmy používané na textové dokumenty sa na zdrojový kód priamo aplikovať nedajú, pretože pri tokenizácii textových dokumentov dostávame rozmanitý zoznam tokenov (termov). Pri zdrojovom kóde je tento zoznam značne oklieštený, pretože programovacie jazyky mávajú obmedzené množstvo tokenov. Každý väčší kúsok zdrojového kódu bude obsahovať skoro všetky dostupné tokeny v programovacom jazyku. Okrem štandardnej tokenizácie zdrojového kódu je možné zo zdrojového kódu extrahovať identifikátory a príslušnú analýzu vykonať na nich. Správna extrakcia identifikátorov a ich následné spracovanie opäť nie je také jednoduché, ale príslušné metódy už na to existujú [18].

Základné metódy klastrovania dokumentov na základe vzdialeností medzi nimi sa používajú aj v prípade textových dokumentov [19] aj v zdrojových kódoch [20]. Spomenuté práce ukazujú, že využitie euklidovskej vzdialenosti je možné pre text, ale aj pre zdrojový kód. Ďalším príkladom môžu byť metódy založené na fuzzy logike. Tie ukazujú svoj potenciál v prípade textových dokumentov [21], ale aj zdrojového kódu [22]. V súčasnosti sú množstvá porovnávaných dokumentov veľké a väčšina metód naráža na pamäťové limity. Preto boli na klastrovanie textu použité metódy založené na metóde *Latent dirichlet allocation* (LDA) [23]. Rovnaká metóda sa ukázala použiteľná aj pre zdrojový kód [24] a okrem klasifikácie sa uplatnila aj v iných oblastiach [25].

2.3.2 Vyhľadávanie zhodných častí

Pri hľadaní zhodných častí sa v prípade textových dokumentov pracuje s podobnosťami na úrovni znakov alebo tokenov. Pri zdrojovom kóde sú to tokeny alebo niektoré zo spomenutých stromových reprezentácií. Najznámejšou metódou používanou aj pre textové dokumenty [26] aj zdrojový kód [27] je metóda *Running Karp-Rabin Matching and Greedy String Tiling* (RKR-GST). Táto metóda je založená na hľadaní najdlhšieho spoločného podreťazca v oboch dokumentoch. V prípade textových dokumentov využíva reprezentáciu dokumentu pomocou jednotlivých znakov. Pri zdrojovom kóde je kód reprezentovaný reťazcom tokenov.

Druhou často používanou metódou, či už v prípade textových dokumentov [28], ale aj zdrojových kódov [29] je *Winnowing*. Táto metóda využíva tzv. lokálne značkovanie. Základom jej algoritmu je reprezentácia dokumentu pomocou n-gramov. Tieto n-gramy sa následne používajú na výpočet odtlačku dokumentu. Nakoniec sa hľadajú podobnosti medzi týmito odtlačkami.

Mnoho algoritmov má spoločné základy. Líšia sa predovšetkým v spôsobe, akým spracovávajú vstupný súbor. Následná analýza môže byť dokonca rovnaká. To ale neznamená, že použitím algoritmu vytvoreného pre text, dosiahneme rovnako dobré výsledky aj v prípade zdrojového kódu.

2.4 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:

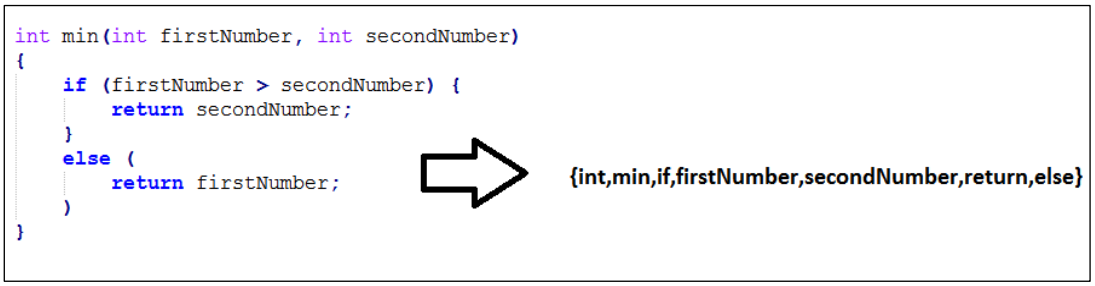
- Analýza zdrojového kódu ako textu v prirodzenom jazyku.
- Analýza prúdu tokenov.
- Analýza modelu zdrojového kódu.

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.

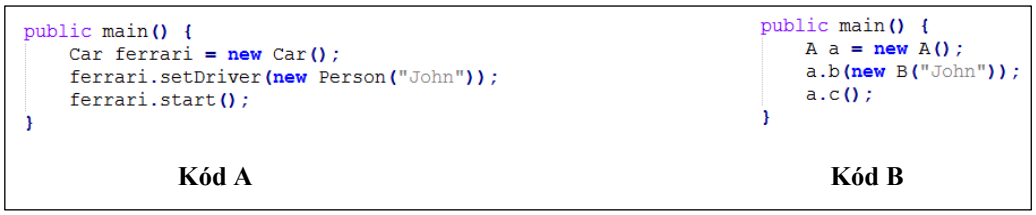
2.4.1 Analýza zdrojového kódu ako textu

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.



Obrázok 8: Extrakcia výrazov zo zdrojového kódu

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 [18]. Problém môže nastať pri vlastnom kóde programátora. Každý programátor si môže pomenovávať svoje identifikátory podľa vlastnej predstavy a vo vlastnom jazyku. Na obrázku 9 sa nachádzajú dva útržky zdrojového kódu, ktorý robí to isté. Kód A obsahuje logicky pomenované identifikátory a komentáre a kód B nie. Pre NLP je ťažké spracovať kód B pretože preňho nemá žiaden význam.



Obrázok 9: Porovnanie dobrého a zlého kódu

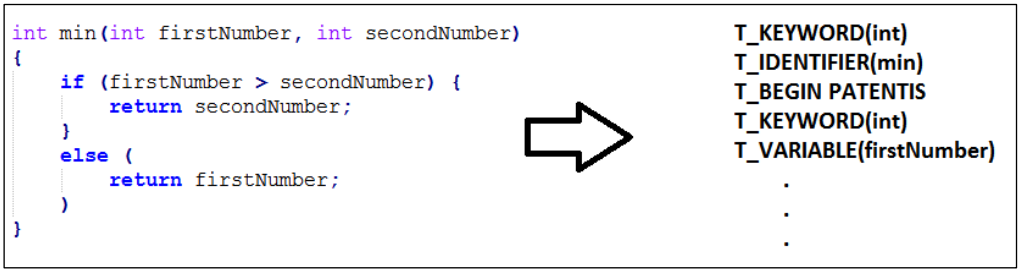
Spracovaniu textových dokumentov pomocou NLP sa venuje množstvo prác, ale využitie NLP na spracovanie zdrojového kódu nie je zatiaľ úplne bežné. Existujú práce, ktoré ho používajú na rôzne analýzy zdrojového kódu [30][31].

Ďalší smer, ktorým sa zaoberá výskum v tejto oblasti je spôsob reprezentácie zdrojových kódov. Doteraz spomenuté práce využívali predovšetkým reprezentáciu zdrojového kódu ako súboru termov respektíve n-gramy z týchto termov. V prípadoch, keď sa pracuje s viacerými zdrojovými kódmi, je štandardnou formou reprezentácie *Document-term matrix* (DTM). DTM je matica, ktorej riadky zodpovedajú jednotlivým zdrojovým kódom a stĺpce predstavujú termy. Matica popisuje výskyt termov v jednotlivých zdrojových kódoch. Každý riadok matice je vektorom daného zdrojového kódu.

Hlavným problémom DTM je jej veľkosť. Pamäťovú náročnosť sa často rieši implementáciou pomocou riedkych matíc. Výpočtovú zložitosť to naopak zhoršuje. Okrem toho efektívnosť nie je dobrá, pretože ak by sme chceli vyhľadávať napríklad synonymá, tak by výpočtová náročnosť opäť narástla. Tento problém sa rieši pomocou LDA [24].

2.4.2 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.



Obrázok 10: Ukážka tokenizácie zdrojového kódu

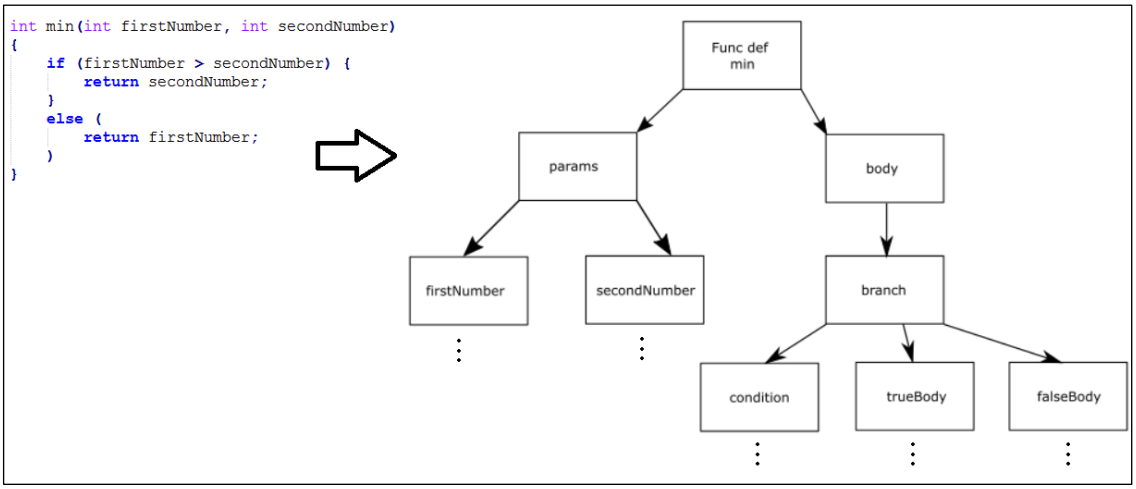
Táto úroveň spracovania zdrojového kódu je najstaršia spomedzi ostatných spomenutých. Používa sa už od začiatkov programovacích jazykov. Postupom času sa vylepšuje o nové poznatky z iných oblastí, alebo na nej stavajú nové a modernejšie metódy. Využíva sa od nástrojov na podporu programovania, ktoré 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*.

2.4.3 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. Medzi algoritmy, ktoré úspešne používajú AST na odhaľovanie plagiátorstva v zdrojových kódoch patrí *CodeCompare* [33] a *AST-CC* [34]. Okrem neho existujú práce, ktoré popisujú rôzne využitie AST na odhaľovanie podobností v zdrojových kódoch [35].



Obrázok 11: Transformácia zdrojového kódu na AST

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. Táto myšlienka je podrobne popísaná napríklad v práci *Improved Plagiarism Detection Algorithm Based on Abstract Syntax Tree* [33]. Autori v nej popisujú aj možnosti ďalšej optimalizácie algoritmov hľadania podobností s využitím AST.

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 [36][37] v zdrojových kódoch, čo je veľmi podobná oblasť.

2.5 Hľadanie klonov / plagiátov v zdrojovom kóde

Ako sme už uviedli, vyhľadávanie klonov („zhodných“ častí v rámci jedného / viacerých zdrojových kódov) a identifikácia plagiátov sú dve veľmi podobné oblasti. V literatúre [38] sa uvádzajú rôzne definície a kategórie klonov. Klony sa rozdeľujú do dvoch základných skupín. Jednoduché klony, ktoré vznikli kopírovaním častí zdrojového kódu z jedného miesta na druhé a vysoko-úrovňové klony - HLC (z anglického *Higher Level Clones*). HLC sa rozdeľujú do 4 kategórií:

- **Behaviorálne klony** – dva fragmenty zdrojového kódu majú rovnakú funkcionálnu reprezentáciu, a pri rovnakých vstupoch dávajú rovnaké výstupy.
- **Konceptuálne klony** – vznikajú napríklad pri aplikovaní návrhových vzorov pri vývoji softvéru.
- **Štrukturálne klony –** rovnaká štruktúra zdrojového kódu zabezpečujúca operácie nad rôznymi druhmi dát.
- **Modelové klony –** klony ktoré vznikajú vďaka podobnostiam vo fáze návrhu za pomoci UML.

Rôzne algoritmy na vyhľadávanie klonov dokážu odhaliť rôzne klony na základe spomenutých kategórií. Jednoduché klony nie je veľký problém odhaliť, ale v prípade HLC sa väčšinou výskumné práce zameriavajú na odhaľovanie štrukturálnych klonov.

V prípade vyhľadávania plagiátov je situácia jednoduchšia. Je zrejmé, že štrukturálne klony je potrebné brať v súvislosti s plagiátorstvom do úvahy. Identifikácia konceptuálnych a modelových klonov nemá zmysel v súvislosti s plagiátorstvom. Pri behaviorálnych klonoch si treba položiť otázku, či má zmysel považovať za plagiát dva rôzne zdrojové kódy, ktoré robia z funkčného hľadiska to isté.

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 [39] 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.

Všetky spomenuté nedostatky podľa nášho názoru vyplývajú z pôvodnej myšlienky, na základe ktorej boli tieto metódy navrhnuté. Tieto systémy boli primárne určené na vyhľadávanie plagiátov v rámci určitej skupiny študentov. Ako sme rozoberali v kapitole 1.1.1, na základe našich skúseností, tento spôsob už v súčasnosti nestačí.

S problémom **zastaranosti** sa stretávame hlavne v prípade, keď potrebujeme spracovať zdrojový kód v niektorom z modernejších programovacích jazykoch. Ako príklad si môžeme uviesť systém JPlag, ktorý sme používali pri vyhľadávaní plagiátov na predmetoch *Informatika 1* a *2* a predmete *Algoritmy a údajové štruktúry*. Systém síce deklaruje podporu pre programovací jazyk *Java 7*, ale v súčasnosti (začiatok roku 2019) je aktuálna verzia *Java 11* a najstaršou oficiálne podporovanou verziou Javy je v súčasnosti *Java 8*. Študenti majú pri výučbe prístup k verzii *8* a tým pádom sa stávalo, že niektoré súbory nedokázal JPlag spracovávať. Obdobný problém nastával aj keď sme sa pokúšali spracovávať zdrojový kód v jazyku C++ na predmete *Algoritmy a údajové štruktúry* respektíve *Operačné systémy* kde JPlag mal opäť problém so spracovaním súborov využívajúcich prvky novšej syntaxe. Pokiaľ sa systému JPlag nepodarí spracovať súbor, je tento vylúčený z následného spracovania, čo môže spôsobiť prehliadnutie určitých zhôd.

Druhým spomenutým problémom je **uzavretosť**. V prípade systému JPlag toto tvrdenie neplatí, pretože jeho zdrojové kódy sú voľne dostupné, a tým pádom si každý môže doprogramovať spracovanie ľubovoľného programovacieho jazyka. Druhý spomenutý systém (MOSS), ktorý sme používali na vyhľadávanie plagiátov v predmete *Pokročilé objektové technológie*, je postavený ako služba, do ktorej je možné nahrať zdrojové súbory a ona na základe nich vygeneruje report.

Obzvlášť veľkými problémami, ktoré odrádzajú od používania týchto systémov vo väčšom meradle, sú **zložitosť a pracnosť,** s ktorou sa dokážeme dopracovať k výsledku. Štandardný postup, keď chceme vykonať kontrolu plagiátov, pozostáva z niekoľkých fáz.

Prvou fázou je príprava zdrojových súborov, ktoré chceme skontrolovať. JPlag požaduje ako vstupný parameter cestu k zložke, kde sa nachádzajú vypracované zadania. Zdrojové kódy je potrebné stiahnuť a umiestniť do jedného priečinka. Tento priečinok následne musí obsahovať jednotlivé zadania vo vlastných priečinkoch. Pri zobrazovaní reportu sa názov týchto priečinkov zoberie a prezentuje sa ako názov zadania. Pokiaľ je zdrojom týchto súborov LMS Moodle, tak ten automaticky pomenúva zložky menom študenta, ktorý prácu odovzdal. Ďalšou komplikáciou v prípade použitia Moodle je to, že

Moodle síce umožňuje hromadné stiahnutie vypracovaných úloh, ale tieto úlohy stiahne vo forme, v akej ich tam študenti nahrali - rôzne formáty archívov *(.zip, .rar* ...). Všetky takto zabalené riešenia je nutné najskôr rozbaliť. V prípade spracovávania zdrojových súborov z viacerých akademických rokov (chceme skontrolovať, či študent neodpísal prácu od staršieho spolužiaka), je potrebné do rovnakej zložky umiestniť aj staršie práce. V tomto prípade sa nám osvedčilo pridať pred meno študenta aj akademický rok, v ktorom daná práca vznikla.

Po príprave súborov nasleduje fáza vyhľadávania zhôd. Oba spomínané systémy (JPlag a MOSS) po spustení kontroly vykonajú porovnanie každej práce s každou a vygenerujú report. Pred samotným spustením algoritmu je potrebné nastaviť niekoľko parametrov. Medzi základné patria: programovací jazyk, citlivosť a ignorované súbory. Ako sa ukázalo hlavne pri analýze úloh z predmetu *Informatika 1*, je nevyhnutné vyčleniť určité súbory z porovnávania. Študenti často pri riešení semestrálnych prác používajú určité komponenty (zdrojový kód), ktorý dostali počas semestra. Oba systémy umožňujú nahrať tzv. „základné“ súbory, voči ktorým potom nehľadajú zhodu. JPlag okrem toho umožňuje špecifikovať názvy súborov, ktoré má ignorovať. Toto je výhodné hlavne z toho dôvodu, že nie je nutné do systému nahrávať všetky súbory a ich rozličné verzie, ktoré študenti dostali počas semestra.

JPlag **Search Results**

Language:	Java1.7 Parser
Submissions:	136 **(2 have not been parsed successfully)**
Matches displayed:	9 (Treshold: 50.1%) (average similarity) 160 (Treshold: 50.0%) (maximum similarity)
Date:	2018-05-21
Minimum Match Length (sensitivity):	9
Suffixes:	.java, .jav, .JAVA, .JAV

Distribution:

90% - 100%	0	
80% - 90%	0	
70% - 80%	0	
60% - 70%	1	#
50% - 60%	8	#
40% - 50%	73	#
30% - 40%	226	###
20% - 30%	643	##########
10% - 20%	3678	##
0% - 10%	4551	##

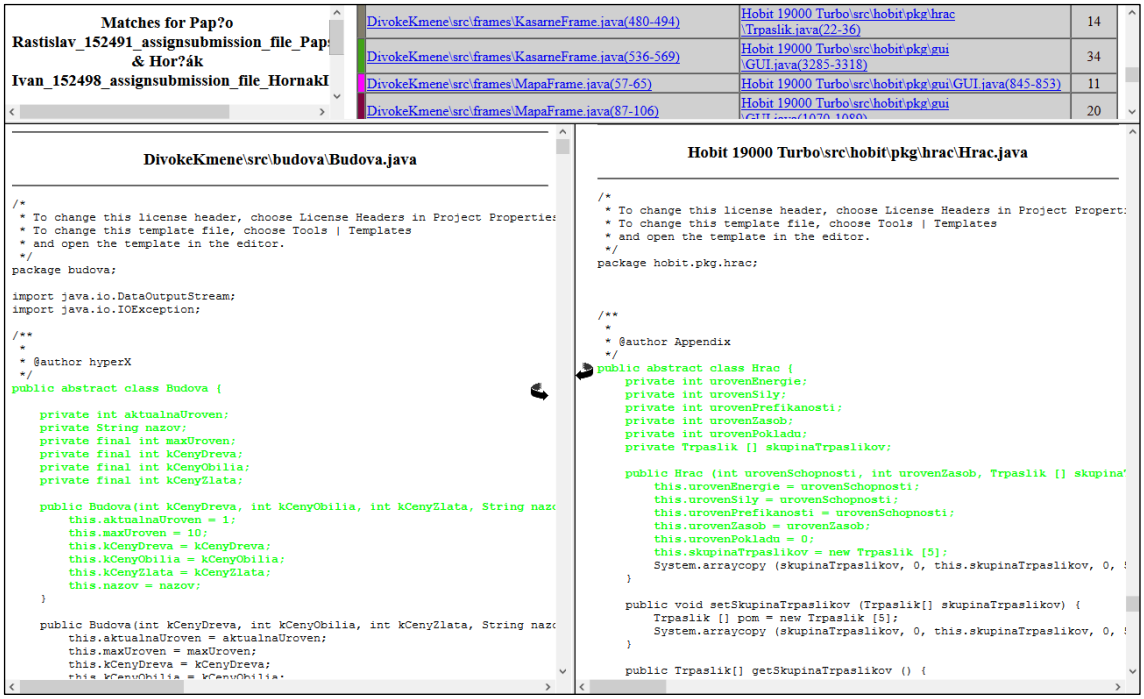
Matches sorted by average similarity (What is this?):

Vrábe? Milan_152446_assignsubmission_file_VrabelMilanINF2	->	Révészová Natália_152463_assignsubmission_file_NatáliaRévészováInf2 (68.7%)	
Pia?ková Veronika_152639_assignsubmission_file_VeronikaPiackovaInf2	->	Pavli?ková Kristína_152325_assignsubmission_file_Semestralna Práca Pavli?ková Kristína 5zv013 (59.9%)	Duda Dávid_152428_assignsubmission_file_David_Duda_semestralna_praca (52.1%)

Obrázok 12: Ukážka reportu zo systému JPlag

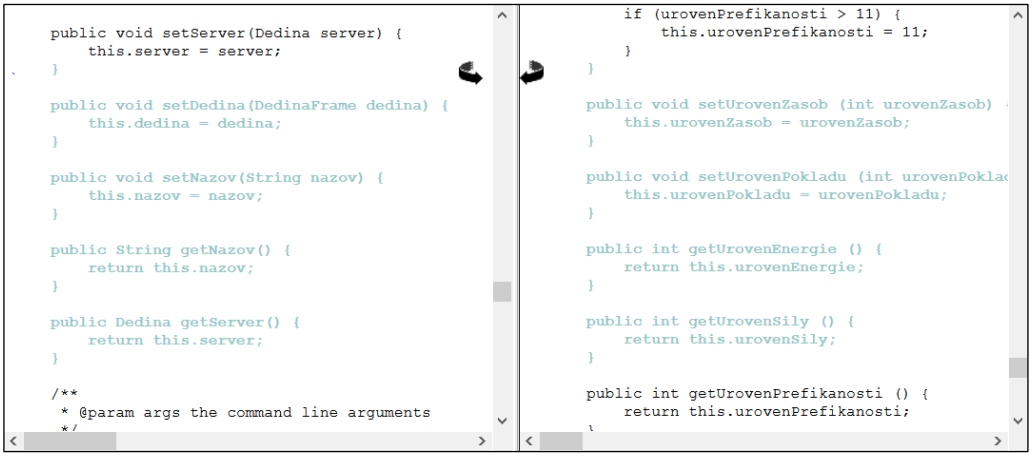
Po vygenerovaní reportu musí tento report vyhodnotiť človek a určiť, ktoré práce obsahujú plagiát, a ktoré nie. Konkrétny príklad týchto reportov môžeme vidieť

na obrázkoch 12 a 13. Na prvom z nich je zobrazený prehľad, v ktorom vidíme štatistiku a jednotlivé skupiny zhodných prác. JPlag sa snaží zhlukovať jednotlivé podobné práce do väčších skupín, čo následne zjednodušuje kontrolu. Na druhom obrázku vidíme už detail podobností medzi dvoma prácami a ich časťami. V hornej časti je zoznam nájdených zhôd a v dolnej časti vyfarbené príslušné časti zdrojového kódu, v ktorom bola nájdená zhoda.



Obrázok 13: Ukážka reportu nájdenej zhody systému JPlag

Ďalším problémom, ktorý zhoršuje čitateľnosť reportov, je časté nachádzanie nevýznamných zhôd. Takéto zhody sú síce po formálnej stránke správne, ale pri vyhodnocovaní plagiátorstva nám veľa nepovedia. Totižto často ide o bežné časti kódu, ktoré môžeme nájsť takmer v každom programe. Na obrázku 14 môžeme vidieť príklad takejto zhody. Z formálneho hľadiska vidíme, že označený kód je totožný, líši sa len v názve identifikátorov. Z logického hľadiska nemá zmysel takúto zhodu brať do úvahy, pretože kód zobrazený na obrázku 14 sa nachádza v každom programe naprogramovanom v programovacom jazyku *Java*. Moderné vývojové prostredia nám dokážu takýto kód generovať úplne automatizovane. Podobných príkladov ako tento môžeme nájsť viacero. Bližšie sa tomuto problému budeme venovať v kapitole 6.1.2.

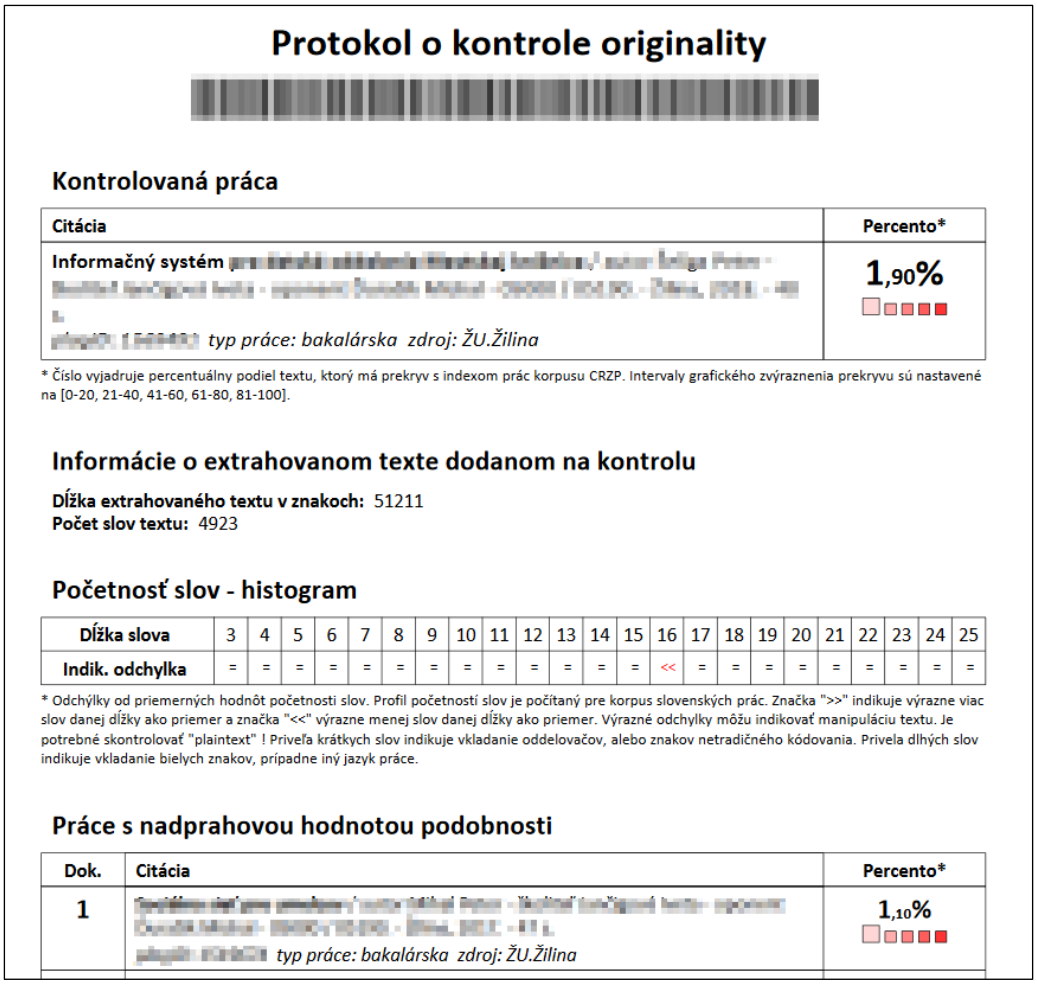


Obrázok 14: Ukážka nevýznamnej zhody

Posledným problémom, na ktorý sme narazili pri vyhľadávaní plagiátov pomocou súčasne dostupných nástrojov, bola ich **ťažkopádnosť až neschopnosť spracovať väčšie množstvo zadaní**. Systém MOSS po určitom objeme prác odmietol prijímať ďalšie a spustil vyhľadávanie zhôd len medzi časťou požadovanej skupiny. Pri využití systému JPlag sme na podobné problémy síce nenarazili, ale vzhľadom na to, že máme dostupné dáta len z posledných dvoch akademických rokov, tak sme možno iba na jeho limit ešte nenarazili. Problém so systémom JPlag spočíval hlavne v neprehľadnosti výsledného reportu počas hľadania plagiátov medzi viacerými akademickými rokmi. Pri kontrole reportu nás zaujímajú len zhody aktuálnych prác a zhody medzi starými prácami by sme chceli ignorovať. JPlag nám niečo takéto neumožňuje a aj napriek tomu, že sme vymysleli efektívny spôsob pomenovania zadaní, čo nám vo vygenerovanom reporte pomohlo odlíšiť zadania z aktuálneho akademického roku od tých starších, časová náročnosť na vyhodnotenie takýchto reportov rastie exponenciálne s narastajúcim množstvom historických dát. Okrem samotného interpretovania výsledkov, ktoré nás trápi v prvom rade, je tu aj otázka efektívnosti samotného algoritmu. Pokiaľ chcem skontrolovať na plagiáty práce z roku 2019, nepotrebujeme, aby systém nanovo porovnával jednotlivé práce zo starších rokov medzi sebou.

Posledným problémom, ktorý taktiež súvisí s možnosťami porovnávania prác voči veľkej databáze je to, že súčasné systémy neumožňujú získať výsledky len pre jednu prácu. Dokážeme síce spustiť porovnanie všetkých prác a následne získať výsledky pre požadovanú prácu, ale ako bolo spomenuté, takéto porovnávanie je neefektívne. Mohlo by sa zdať, že takýto report pre jednu prácu nie je dôležitý, keďže vždy potrebujeme skontrolovať práce v rámci celého ročníka. Zo skúseností z predmetov *Informatika 1* a *Informatika 2,* kde študenti musia svoje semestrálne práce osobne obhájiť, môžeme

povedať, že report z APS dostupný pri obhajobe, by výrazne zvýšil šance vyučujúceho na odhalenie plagiátu. Na uplatnenie takéhoto prístupu aj na iných predmetoch je nutné odstrániť všetky prekážky, ktoré by hodnotiaceho odrádzali od využívania týchto systémov. Hlavným argumentom v tomto prípade bude zrejme náročnosť na vyhodnocovanie takého reportu. Keď si porovnáme výstup so systému JPlag na obrázkoch 12 a 13 a napríklad report zo systému ANTIPLAG (obrázok 15) môžeme jednoznačne skonštatovať, že reporty poskytované systémom JPlag sú oveľa náročnejšie na vyhodnotenie. Naopak report zo systému ANTIPLAG poskytuje skúšajúcemu presné informácie o konkrétnej úlohe, ktorú aktuálne vyhodnocuje.



Protokol o kontrole originality

Kontrolovaná práca

Citácia	Percento*
Informačný systém [redacted] *typ práce: bakalárska zdroj: ŽU.Žilina*	1,90%

* Číslo vyjadruje percentuálny podiel textu, ktorý má prekryv s indexom prác korpusu CRZP. Intervaly grafického zvýraznenia prekryvu sú nastavené na [0-20, 21-40, 41-60, 61-80, 81-100].

Informácie o extrahovanom texte dodanom na kontrolu

Dĺžka extrahovaného textu v znakoch: 51211
Počet slov textu: 4923

Početnosť slov - histogram

Dĺžka slova	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Indik. odchylka	=	=	=	=	=	=	=	=	=	=	=	=	=	<<	=	=	=	=	=	=	=	=	=

* Odchýlky od priemerných hodnôt početnosti slov. Profil početností slov je počítaný pre korpus slovenských prác. Značka ">>" indikuje výrazne viac slov danej dĺžky ako priemer a značka "<<" výrazne menej slov danej dĺžky ako priemer. Výrazné odchylky môžu indikovať manipuláciu textu. Je potrebné skontrolovať "plaintext" ! Priveľa krátkych slov indikuje vkladanie oddelovačov, alebo znakov netradičného kódovania. Privela dlhých slov indikuje vkladanie bielych znakov, prípadne iný jazyk práce.

Práce s nadprahovou hodnotou podobnosti

Dok.	Citácia	Percento*
1	[redacted] *typ práce: bakalárska zdroj: ŽU.Žilina*	1,10%

Obrázok 15: Ukážka protokolu o kontrole originality zo systému ANTIPLAG

Tento problém by sa dal vyriešiť dodatočným automatickým spracovaním výsledkov zo systémov JPlag a MOSS. Problémom ostáva, že okrem reportu vo forme HTML dokumentu neposkytujú tieto systémy žiadne ďalšie formáty, ktoré by umožňovali automatizované spracovávanie.

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 popísaných v kapitole 3.1, 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. V kapitole 2.4 sme sa venovali špeciálnym metódam spracovania zdrojového kódu. Ako sa ukázalo, tieto metódy dosahujú lepšie výsledky, resp. poskytujú lepšie možnosti pri vyhľadávaní plagiátov. Preto sa 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.

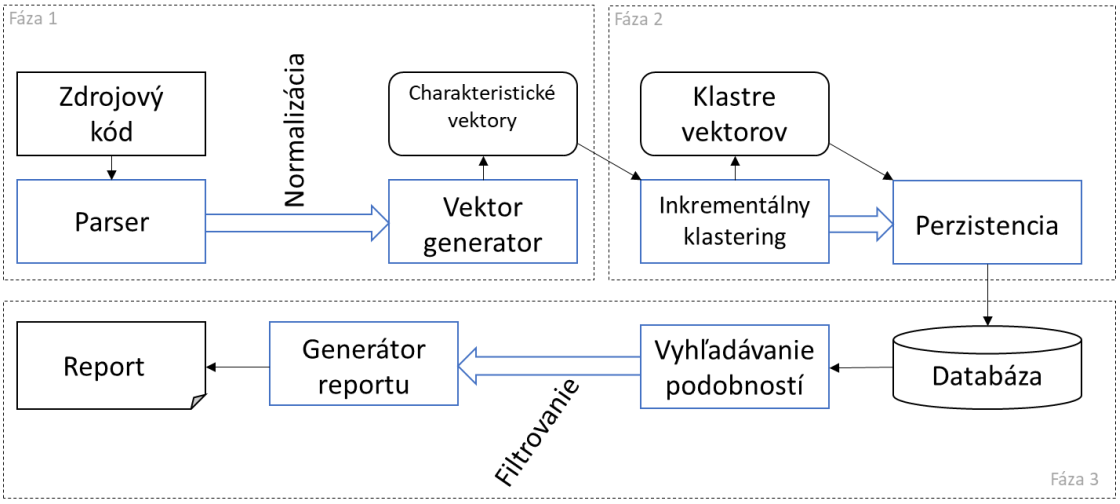
Perzistencia dát je dôležitá hlavne z pohľadu vyhľadávania plagiátov voči historickým dátam, resp. iným zdrojom. Tieto zdroje musia byť určitým spôsobom uchovávané, aby sa o ich manažment nemusel starať používateľ a jednotlivé dáta boli prístupné neustále. Tvorba reportov pre konkrétnu prácu bude pozostávať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

V tejto kapitole popíšeme metódu umožňujúcu vyhľadávať plagiáty vo veľkom množstve zdrojových kódov a popíšeme jej jednotlivé komponenty. Pri návrhu sme vychádzali hlavne z požiadaviek špecifikovaných v predchádzajúcej kapitole. Navrhnutá metóda je zobrazená na obrázku 16.



Obrázok 16: 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. Jednotlivé fázy reprezentujú:

1. Spracovanie a reprezentácia zdrojového kódu.
2. Zhlukovanie a perzistencia dát.
3. Vyhľadávanie podobností a tvorba reportu.

V prvej fáze sa nachádzajú algoritmy umožňujúce spracovať zdrojový kód. Tieto algoritmy rozdeľujeme do dvoch skupín. V prvej z nich sú algoritmy, ktoré spracujú zdrojový kód a vytvoria jeho model vo forme syntaktického stromu. Tieto algoritmy budú vždy špecifické pre každý programovací jazyk, ktorý budeme chcieť spracovávať. Druhú skupinu tvoria algoritmy, ktoré transformujú syntaktický strom do formy vhodnej pre ďalšie spracovanie. Ako vhodnú formu pre reprezentáciu zdrojového kódu sme na základe analýzy zvolili charakteristické vektory. Medzi spracovanie a vektorizáciu môžeme zaradiť

algoritmy, ktoré budú syntaktický strom normalizovať, vďaka čomu je možné odhaliť plagiáty aj v prípade bežne používaných trikov na oklamanie APS [33][34].

V druhej fáze nastáva zhlukovanie podobných vektorov. Pre zhlukovanie podobných vektorov využijeme algoritmus K-Means, ktorý mierne modifikujeme pre účely tejto metódy. Výsledkom tejto fázy budú dáta, ktoré sú pripravené na uloženie do databázy vo forme, ktorá umožní ich jednoduché vyhľadávanie. Cieľom klasteringu je okrem predprípravy dát na vyhľadávanie aj určité logické rozdelenie dát na súvisiace celky. Takéto rozdelenie bude základom pre škálovateľnosť celej metódy.

Posledná fáza sa zaoberá získavaním jednotlivých zhôd z databázy a ich vyhodnocovaním. V tejto fáze sa ešte pred samotným generovaním reportu môže zapojiť filter nevýznamných zhôd, ktorý slúži na sprehľadnenie reportov.

Jednotlivé fázy sú medzi sebou takmer nezávislé. Jediný prvok, od ktorých závisia, je formát prenášaných dát medzi fázami. Medzi prvou a druhou fázou sa prenáša spracovaný model zdrojového kódu, ktorý je reprezentovaný pomocou charakteristických vektorov. A most medzi fázou 2 a 3 tvorí databáza, v ktorej sú vo vhodnej forme uložené spomínané vektory. Takéto rozdelenie nám prináša ďalšie možnosti pri škálovaní celej metódy.

V nasledujúcich kapitolách si detailnejšie predstavíme jednotlivé fázy navrhovanej metódy. V kapitole 4 sa zameriame na rôzne možnosti spracovania zdrojového kódu a výber vhodných reprezentácií. V kapitole 5 preskúmame možnosti klasifikácie charakteristických vektorov, navrhneme a overíme metódu inkrementálneho klasteringu a navrhneme spôsob perzistencie týchto dát. Metódam vyhľadávania plagiátov, tvorbe reportov a filtrovaniu nevýznamných zhôd sa budeme venovať v kapitole 6.

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. Ich prehľad sme uviedli v kapitole 2.4. Medzi dva základne prístupy môžeme zaradiť reprezentáciu zdrojového kódu pomocou prúdu tokenov a AST. V našej ďalšej práci sa reprezentáciám pomocou tokenov zaoberať nebudeme, nakoľko sme sa rozhodli využiť reprezentácie využívajúce syntaktické stromy. 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 [33].

4.1.1 Abstraktný syntaktický strom

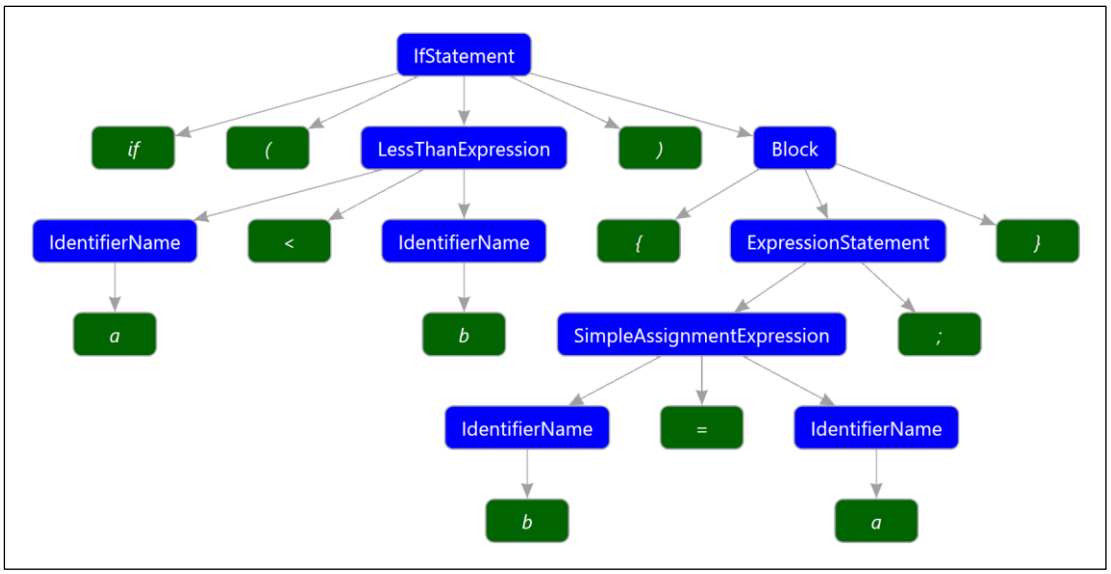
AST je jednou z najčastejších metód reprezentácie zdrojového kódu vo forme stromovej štruktúry. Overenie jednotlivých metód reprezentácie zdrojového kódu sme sa rozhodli aplikovať na programovací jazyk C#. Je to moderný programovací jazyk a spolu s .NET knižnicou poskytuje pohodlné prostredie pre vývoj. 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*)[4]. 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.

Majme jednoduchý kód ktorý vyzerá nasledovne:

```
if (a > b)
{
    a = b;
}
```

Jeho syntaktický strom môžeme vidieť na obrázku 17. Na tomto strome môžeme vidieť dva základné druhy uzlov. Prvým z nich sú tzv. *SyntaxNode*, na obrázku zobrazené modrou farbou. Tie vyjadrujú abstraktný pohľad na štruktúru zdrojového kódu. Druhou skupinou sú tzv. *SyntaxToken* uzly (na obrázku zobrazené zelenou farbou). Tie reprezentujú jednotlivé tokeny.



Obrázok 17: Ukážka syntaktického stromu

Treťou skupinou uzlov, ktoré na obrázku zobrazené nie sú, sú uzly *SyntaxTrivia*. Táto skupina obsahuje časti, ktoré nie sú pre význam zdrojového kódu dôležité. Nájdeme tu napríklad medzery, prázdne riadky a komentáre.

Pre každý uzol dokážeme určiť rozsah, ktorý v zdrojovom kóde zahŕňa. Dôležitou vlastnosťou je, že každý *SyntaxNode* uzol má svoj typ. Na obrázku môžeme vidieť zobrazené práve tieto typy uzlov (*ForStatement, VariableDeclaration*...). Ďalšou

[4] https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/

vlastnosťou tohto stromu je, že každý *SyntaxNode* musí mať aspoň jedného potomka. *SyntaxToken* a *SyntaxTrivia* sú vždy listami tohto stromu. Jednotlivé *SyntaxNode* uzly vždy zastrešujú určitú súvislú časť zdrojového kódu – blok. Post–order prehliadkou tohto stromu po *SyntaxToken* uzloch môžeme získať štandardný prúd tokenov.

4.1.2 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.

4.1.3 Porovnanie hašovania a vektorizácie AST

V literatúre môžeme nájsť dve základné metódy na linearizáciu syntaktického stromu. Výhodnosť rôznych hašovacích metód už bola porovnávaná v článku *Syntax tree fingerprinting: a foundation for source code similarity detection* [32], ale aj napriek tomu sme sa rozhodli vykonať experiment, v ktorom porovnáme základnú metódu hašovania s vektorovými charakteristikami uzla. Cieľom experimentu je určiť, ktorý z týchto prístupov sa viac hodí pri vyhľadávaní plagiátov v akademickom prostredí.

Na rovnakej množine dát vypočítame podobnosti jednotlivých študentských prác s využitím oboch prístupov a tieto výsledky porovnáme s referenčnými hodnotami. Za referenčné hodnoty budeme považovať hodnoty vypočítané pomocou systému MOSS (keďže ten jediný dokáže odhaľovať plagiáty v kóde napísanom v C#). Pri určovaní podobností jednotlivých študentských prác nebudeme hľadať podobné sekvencie v daných zadaniach, ale podobnosť určíme len na základe podobností celých tried a metód. Pri triedach, respektíve metódach, budeme určovať podobnosť na základe ich reprezentácie (haš, vektor). Tento prístup nám síce neumožní odhaliť plagiáty so 100% vierohodnosťou, ale umožní nám rádovo porovnať jednotlivé metódy reprezentácie. Okrem toho je vhodné poznamenať, že v tomto experimente budeme porovnávať časti zdrojového kódu len na základe určitých charakteristík, a výsledné nájdené podobnosti budú takmer určite obsahovať množstvo nerelevantných zhôd. Napriek tomu nebudeme tieto nájdené podobnosti filtrovať na základe porovnania podobností ich stromových štruktúr (ako sa to bežne robí v druhom kroku algoritmov na odhaľovanie plagiátorstva s využitím syntaktických stromov [33][34]), pretože ich filtráciou by sme odstránili rozdiely, ktoré obe metódy dosahujú.

V našich algoritmoch budeme používať dva parametre. Parameter α z rozsahu 0 - 1 sa používa pri meraní podobností tried a parameter β z rozsahu 0 - 1 určujúci prah pri určovaní podobností na úrovni metód.

Základnou metrikou je **podobnosť riešení**. Podobnosť riešenia *a* a *b* je definovaná ako:

$$\boldsymbol{simA(a,b)} = \frac{|\boldsymbol{a} \cap \boldsymbol{b}|}{|\boldsymbol{a} \cup \boldsymbol{b}|} \quad (1)$$

kde čitateľ reprezentuje počet **podobných tried** a menovateľ reprezentuje početnosť tried v riešeniach. Triedy C_a a C_b označíme ako podobné ak $simC(C_a,C_b) >= \alpha$ a C_a a C_b majú aspoň jednu dvojicu **podobných metód** alebo obe neobsahujú žiadne metódy. Metódy m_1 a m_2 definujeme ako podobné ak $simM(m_1, m_2) >= \beta$. Podobnosť tried (*simC*) a podobnosť metód (*simM*) je definovaná pre každú reprezentáciu zvlášť a konkrétny vzorec uvedieme neskôr.

Algoritmus porovnávania je dvojkrokový. V úvodnom kroku sa pre dané riešenia určí podobnosť tried a na základe nej sa pre vhodné dvojice tried preskúmajú a identifikujú podobné metódy. V druhom kroku sa na základe nich identifikuje množina podobných tried pre riešenia *a* a *b*.

Dôležité je, že pri tomto experimente nehľadáme presné zhody medzi dvoma riešeniami, pretože by to zahŕňalo porovnávanie celých stromových štruktúr, ale na základe popisovaných reprezentácií sa snažíme určiť, ktorá je sama o sebe vhodnejšia na určenie plagiátov.

4.1.3.1 Definícia množiny dát

Ako testovaciu množinu dát sme zvolili riešenia semestrálnych prác z predmetu *Pokročilé objektové technológie*. Zadaním bolo vytvorenie textovej hry (konzolová aplikácia). Konkrétny obsah hry si každý študent zvolil sám.

Testovacia množina obsahovala 59 riešení od študentov (v príkladoch označené ako *pr_1* až *pr_59*) a okrem nich sme vyrobili 2 plagiáty. Prvý plagiát (*cp_01*) je 100% kópia riešenia *pr_8*. Druhý plagiát (*cp_02*) vznikol z riešenia *pr_48* manuálnymi úpravami (odstránenie, úprava a pridanie nového kódu).

Celkovo množina obsahuje 61 riešení, ktoré obsahujú spolu 1050 tried a 2468 metód. Tieto triedy sú rozdelené do 958 súborov. Počet riadkov kódu v C# je 49653.

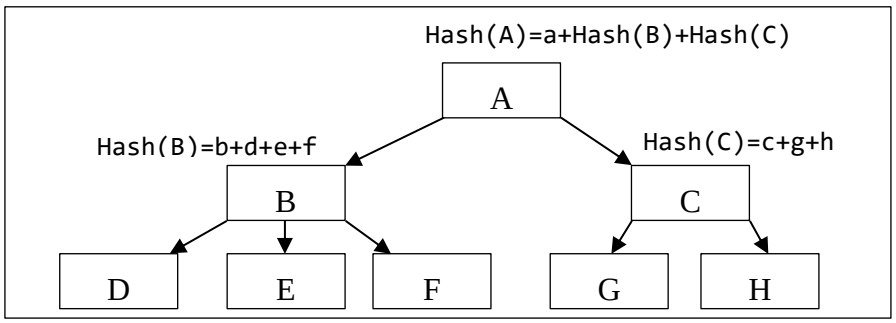
4.1.3.2 Generovanie podobností pomocou hašovania

Prvým prístupom ako určovať *simC* a *simM* je porovnávanie hašov príslušných tried a metód. Konkrétny vzorec uvedieme na konci tejto kapitoly. Myšlienka hašovania je inšpirovaná článkom *Improved Plagiarism Detection Algorithm Based on Abstract Syntax Tree [33]*. Pri tomto algoritme postupne prechádzame smerom od listov ku koreňu po syntaktickom strome a počítame haš pre jednotlivé uzly. V našom prípade sme sa rozhodli využívať len uzly typu *SyntaxNode,* nakoľko dobre popisujú význam zdrojového kódu a zbytočne nezavadzajú závislosti vyplývajúce z konkrétnej syntaxe.

Majme podstrom s koreňom v uzle X. Haš pre tento podstrom vyjadríme ako $Hash(X)$, ktorý vypočítame:

$$\boldsymbol{Hash(X) = x + \sum_{c \in Childrens(X)} Hash(c)} \quad (2)$$

kde x je vlastný haš koeficient konkrétneho uzla, ktorý závisí od jeho typu a $Childrens(X)$ sú potomkovia uzlu X. Výpočet hašu je pre každý uzol rovnaký. Takto počítaný hash nám napríklad jednoducho umožní odlíšiť časti kódu s rôznou veľkosťou (keďže pri sčítaní daný haš s narastajúcim počtom uzlov rastie). Znázornenie tohto výpočtu je zobrazené na obrázku 18.



Obrázok 18: Výpočet hašu v AST

Koeficienty x, pre jednotlivé typy uzlov, nie sú v žiadnom z dostupných článkov, ktoré používajú tento prístup, presne definované, preto sme zvolili niekoľko postupov, ako tieto koeficienty určiť.

Prvým prístupom bolo náhodné priradenie týchto hodnôt jednotlivým typom uzlov. Pri priraďovaní sme postupovali tak, že postupne sme priraďovali nepárne čísla začínajúce od 1 jednotlivým typom ulov. Dosiahli sme tým unikátnosť hodnôt pre všetky dostupné typy uzlov. Výber nepárnych čísel bol zvolený preto, lebo pri súčtových typoch hašov vykazujú lepšie vlastnosti.

Pri druhom prístupe sme využili prvočísla. Prvočísla sme podobne ako v prvom prípade priraďovali tak, aby každý typ uzla dostal unikátny koeficient. Pri priraďovaní prvočísel sme postupovali tak, že sme si jednotlivé typy uzlov usporiadali podľa ich relatívnej pozície v AST. Úplne navrchu zvyčajne býva uzol reprezentujúci menné priestory, potom nasledujú uzly reprezentujúce triedy, metódy.... Úplne na druhom konci sú to uzly reprezentujúce identifikátory či literály.

Prvočísla sme najskôr začali priraďovať od vrchu pomyselného AST stromu (uzla reprezentujúceho definíciu menného priestoru). V tomto prípade mali najväčší vplyv na výslednú hodnotu hašu metódy a triedy konkrétne výrazy (lebo mali vysoké hodnoty koeficientov).

Potom sme toto priradenie otočili, a najmenšie hodnoty koeficientov dostali identifikátory, výrazy... V tomto prípade konkrétne identifikátory a výrazy nemali až taký veľký vplyv na konkrétnu hodnotu hašu jednotlivých metód a tried.

Po výpočte hašov pre jednotlivé podstromy môžeme pristúpiť k porovnávaniu riešení. Pri porovnávaní postupujeme podľa popísaného algoritmu a *simC* a *simM* definujeme nasledovne:

$$\boldsymbol{simC(C_a, C_b) = 1 - \frac{|Hash(C_a) - Hash(C_b)|}{\min\{Hash(C_a), Hash(C_b)\}}} \quad (3)$$

$$\boldsymbol{simM(m_1, m_2) = 1 - \frac{|Hash(m_1) - Hash(m_2)|}{\min\{Hash(m_1), Hash(m_2)\}}} \quad (4)$$

kde pod C_a, C_b, m_1, m_2 rozumieme korene syntaktického stromu príslušných tried, respektíve metód.

4.1.3.3 Generovanie podobností s použitím číselných charakteristík

Druhým spôsobom ako určovať *simC* a *simM* sú číselné charakteristiky tried a metód reprezentované pomocou charakteristických vektorov. Pri triedach a metódach môžeme identifikovať rôzne charakteristiky. Nech v_C je charakteristický vektor pre triedu C. Zložky tohto vektora sme určili nasledovne:

- počet atribútov,
- počet metód,
- počet vlastností.

Takto definovaný vektor pre triedy nám umožní na rozdiel od hašovania jednoduchší výpočet, pretože nemusíme pri porovnávaní tried prechádzať celý podstrom danej triedy. Obsahuje početnosti základných členov každej triedy.

Nech v_m je charakteristický vektor pre metódu m. Na rozdiel od tried máme pri metódach viac možností ako definovať ich charakteristický vektor. Ako základné charakteristiky metódy sme určili:

- počet parametrov,
- počet výrazov v tele metódy,
- maximálna výška podstromu reprezentujúceho danú metódu.

Počet parametrov je jednou zo základných charakteristík, ktoré môžeme na každej metóde nájsť. Počet výrazov a maximálna výška podstromu, reprezentujúceho danú metódu, slúžia na vyjadrenie komplexnosti danej metódy.

Okrem týchto základných charakteristík môžeme do charakteristického vektora metódy zaradiť aj početnosť jednotlivých typov uzlov, ktoré sa nachádzajú v podstrome danej metódy. V našom experimente sme určili 7, podľa nás významných, typov uzlov, ktoré dodatočne popisujú štruktúru danej metódy. Sú to uzly pre podmienené vetvenie, cykly, deklaráciu premennej a volanie metódy.

Pri experimentoch sme uvažovali aj o váhovaní jednotlivých zložiek týchto vektorov. Vďaka váhovaniu môžeme docieliť, aby napríklad počet metód v triede mal väčší význam ako počet atribútov. Podobne môžeme určiť váhu jednotlivých zložiek aj pri metódach, vďaka čomu môžeme skúmať vplyv zložiek na celkovú efektivitu porovnávania.

Po definícii základných princípov môžeme definovať *simC* a *simM* pri vyhľadávaní podobností s využitím charakteristických vektorov.

$$\boldsymbol{simC}(\boldsymbol{C_a}, \boldsymbol{C_b}) = \frac{\sum_{i=0}^{n} \mathbf{1} - \frac{\left|v_{C_a}(i) - v_{C_b}(i)\right|}{\max\left\{v_{C_a}(i), v_{C_b}(i)\right\}} \boldsymbol{h(i)}}{\sum_{i=0}^{n} \boldsymbol{h(i)}} \tag{5}$$

kde n je dĺžka charakteristického vektora v_C a h je vektor váh pre charakteristický vektor triedy.

$$\boldsymbol{simM}(\boldsymbol{m_1}, \boldsymbol{m_2}) = \frac{\sum_{i=0}^{n} \mathbf{1} - \frac{\left|v_{m_1}(i) - v_{m_2}(i)\right|}{\max\left\{v_{m_1}(i), v_{m_2}(i)\right\}} \boldsymbol{g(i)}}{\sum_{i=0}^{n} \boldsymbol{g(i)}} \tag{6}$$

kde n je dĺžka charakteristického vektora v_m a g je vektor váh pre charakteristický vektor metódy.

4.1.3.4 Výsledky porovnania

V tejto časti uvedieme 7 reprezentatívnych konfigurácií experimentu, na základe ktorých následne vyhodnotíme experiment. Parametre α a β pre jednotlivé experimenty sú uvedené v tabuľke 3. Okrem toho tabuľka obsahuje aj početnosti nájdených zhôd medzi riešeniami, triedami a metódami. Stĺpec „Podobnosti riešení“ vyjadruje počet nájdených kombinácii dvojíc riešení, ktoré majú aspoň jednu zhodu na úrovni tried. Stĺpec „Podobnosti tried“ vyjadruje celkový počet nájdených dvojíc tried, ktoré algoritmus označil za podobné a stĺpec „Podobností metód“ vyjadruje celkový počet dvojíc metód, ktoré boli algoritmom označené za podobné.

Názov konfigurácie	α	β	Podobnosti riešení	Podobnosti tried	Podobnosti metód
Vektor1	0,8	0,98	989	21924	1650
Vektor2	0,8	0,8	1263	24243	8113
Hash1	0,8	0,98	1356	13596	6906
Hash2	0,8	0,8	1582	19544	40624
Hash3	0,8	0,98	1534	26225	14201
Vektor3	0,8	0,98	1023	22204	1754
Vektor4	0,8	0,98	960	21770	1030

Tabuľka 3: Testované konfigurácie

V uvedenej tabuľke sú celkovo 4 zo 7 konfigurácii počítané pomocou charakteristických vektorov. Prvé dve konfigurácie (Vektor1 a Vektor2) sa líšia len v parametri β. Charakteristické vektory metód v týchto dvoch konfiguráciách obsahovali základné charakteristiky metódy a početnosti 7 významných typov uzlov. Váhové koeficienty boli nastavené pre každú zložku tohto vektora na 1. Experimentovali sme aj s rôznymi váhovými koeficientami pre časť vektora, ktorá obsahovala základné charakteristiky metódy, no výrazné rozdiely vo výsledkoch sme nezaznamenali.

Z týchto výsledkov sme potvrdili, že zníženie hodnoty parametra β z 0,98 na 0,80 spôsobí nárast v počte nájdených zhôd, tak ako sme očakávali. Pri ďalších experimentoch s použitím vektorov sme sa rozhodli používať fixnú hodnotu β a menili sme spôsob tvorby vektorov.

V konfigurácii „Vektor3“ sme charakteristickým vektorom metódy reprezentovali početnosť každého typu uzlu, ktorý sa v našom syntaktickom strome druhu *SyntaxNode* môže nachádzať. Konfigurácia „Vektor4“ bola podobná predchádzajúcej, líšili sa len v tom, že jej charakteristický vektor metódy bol rozšírený o základné charakteristiky

metódy. Váhové koeficienty boli v týchto konfiguráciách pre všetky zložky vektora rovné 1.

Zvyšné konfigurácie boli založené na počítaní podobností pomocou hašovania. Prvé dve z nich („Hash1“ a „Hash2“) sa líšia len v hodnote parametra β. Tieto konfigurácie obsahovali ako vlastné hašovacie koeficienty uzlov prvočísla zoradené zostupne (tj. hierarchicky najvyššie typy uzlov získali najvyššie koeficienty). Opačné zoradenie koeficientov dosahovalo výrazne horšie výsledky, takže sme sa ním veľmi nezaoberali.

Pre parameter β platí, podobne ako v prípade vektorov, že zníženie hodnoty spôsobí nárast v počte nájdených zhôd.

Posledná konfigurácia („Hash3“) používa náhodne zvolené vlastné hašovacie koeficienty uzlov.

Hodnotu parametru α sme v priebehu experimentovania nemenili práve preto, že má hlavný vplyv na počet nájdených podobností. Hodnota 0.8 bola zvolená na základe toho, že poskytuje dobrý pomer medzi množstvom nájdených podobností a relevantnosťou týchto podobností.

Na vyhodnotenie jednotlivých konfigurácií sme používali porovnanie nami nameraných podobností s plagiátmi, ktoré určil algoritmus MOSS. Keďže MOSS počíta podobnosť na základe dĺžky zhodných sekvencií kódu, a náš algoritmus podľa počtu zhodných tried, porovnávať priamo tieto podobnosti nemá zmysel. Namiesto toho sme porovnávali poradie jednotlivých plagiátov podľa zvolených konfigurácií, s poradím dosiahnutým v systéme MOSS. V tabuľke 4 môžeme vidieť porovnanie týchto poradí.

Táto tabuľka nám ukazuje, že vektory sú vo všeobecnosti lepšie ako haše. V prípade vektorov sa ako najlepšia konfigurácia ukázala konfigurácia „Vektor3“. Zaujímavé je aj to, že konfigurácia „Vektor4“, ktorej charakteristický vektor metódy obsahoval najviac zložiek, sa ukázala horšia ako „výherná“ konfigurácia.

V prípade konfigurácií používajúcich hašovanie je zrejmý rozdiel medzi konfiguráciou „Hash1“, ktorá obsahuje určitým spôsobom usporiadané vlastné hašovacie koeficienty, oproti konfigurácii „Hash3“, kde boli tieto koeficienty rozmiestnene náhodne.

A	B	MOSS	Vektor1	Vektor2	Hash1	Hash2	Hash3	Vektor3	Vektor4
pr_8	cp_01	**1**	0 = 0	1 = 0	1 = 0	1 = 0	1 = 0	1 = 0	1 = 0
pr_48	cp_02	**2**	3 ▼ -1	4 ▼ -2	7 ▼ -5	3 ▼ -1	6 ▼ -4	3 ▼ -1	3 ▼ -1
pr_12	pr_29	**3**	4 ▼ -1	5 ▼ -2	2 ▲ 1	2 ▲ 1	4 ▼ -1	4 ▼ -1	4 ▼ -1
pr_26	pr_50	**4**	14 ▼ -10	17 ▼ -13	16 ▼ -12	27 ▼ -23	19 ▼ -15	14 ▼ -10	14 ▼ -10
pr_14	pr_12	**5**	6 ▼ -1	6 ▼ -1	5 = 0	8 ▼ -3	8 ▼ -3	6 ▼ -1	6 ▼ -1
pr_14	pr_39	**6**	12 ▼ -6	13 ▼ -7	12 ▼ -6	14 ▼ -8	13 ▼ -7	12 ▼ -6	12 ▼ -6
pr_50	pr_12	**7**	7 = 0	7 = 0	3 ▲ 4	5 ▲ 2	2 ▲ 5	7 = 0	7 = 0
pr_14	pr_50	**8**	7 ▲ 1	7 ▲ 1	8 = 0	10 ▼ -2	9 ▼ -1	7 ▲ 1	7 ▲ 1
pr_39	pr_50	**9**	11 ▼ -2	11 ▼ -2	10 ▼ -1	11 ▼ -2	10 ▼ -1	11 ▼ -2	11 ▼ -2
pr_39	pr_26	**10**	17 ▼ -7	18 ▼ -8	17 ▼ -7	19 ▼ -9	16 ▼ -6	15 ▼ -5	17 ▼ -7
pr_14	pr_29	**11**	4 ▲ 7	2 ▲ 9	5 ▲ 6	8 ▲ 3	4 ▲ 7	4 ▲ 7	4 ▲ 7
pr_39	pr_12	**12**	9 ▲ 3	9 ▲ 3	11 ▲ 1	12 = 0	11 ▲ 1	9 ▲ 3	9 ▲ 3
pr_50	pr_29	**13**	2 ▲ 11	3 ▲ 10	4 ▲ 9	5 ▲ 8	3 ▲ 10	2 ▲ 11	2 ▲ 11
pr_39	pr_29	**14**	10 ▲ 4	10 ▲ 4	9 ▲ 5	7 ▲ 7	7 ▲ 7	10 ▲ 4	10 ▲ 4
pr_26	pr_29	**15**	13 ▲ 2	15 = 0	14 ▲ 1	26 ▼ -11	14 ▲ 1	13 ▲ 2	13 ▲ 2
pr_26	pr_12	**16**	15 ▲ 1	14 ▲ 2	14 ▲ 2	20 ▼ -4	17 ▼ -1	16 = 0	15 ▲ 1
pr_14	pr_26	**17**	15 ▲ 2	15 ▲ 2	13 ▲ 4	16 ▲ 1	14 ▲ 3	16 ▲ 1	15 ▲ 2
			59	66	64	85	73	55	59

Tabuľka 4: Porovnanie poradí podobností podľa jednotlivých algoritmov

V tabuľke môžeme vidieť porovnanie poradí jednotlivých dvojíc zadaní. Pri každom poradí máme navyše rozdiel voči poradiu, ktoré zadanie dosiahlo v systéme MOSS. V týchto výsledkoch môžeme vidieť niekoľko anomálií. Prvou z nich je podobnosť medzi *pr_26* a *pr_50* ktorá sa pri použití algoritmu MOSS umiestnila na 4. pozícii no v prípade nášho algoritmu skončila v priemere o 13 pozícií nižšie. Táto anomália bola spôsobená tým, že tieto dve riešenia obsahovali množstvo automaticky generovaného kódu, ktorý bol zhodný, no nachádzal sa len v jednej triede.

Druhou anomáliou bola podobnosť medzi *pr_50* a *pr_29*, kde MOSS umiestnil túto dvojicu na 13. miesto a v našom algoritme sa umiestnila vo väčšine prípadov za 100% klonom. Táto anomália bola opäť spôsobená automaticky generovaným kódom. Rozdiel oproti prvej anomálii spočíval v tom, že *pr_29* obsahoval veľmi malé množstvo vlastných tried, vďaka čomu sa výrazne prejavil vplyv automaticky generovaného kódu.

4.1.3.5 Zhodnotenie

Pomocou tohto jednoduchého experimentu sme si overili reálne možnosti jednotlivých prístupov. Na rozdiel od citovaných článkov sme sa nesnažili optimalizovať AST stromy, ale preskúmať implementačnú náročnosť a efektivitu spomínaných prístupov. V prípade hašovania bol najväčší problém vo voľbe vlastných hašovacích koeficientov a je škoda, že citované články spôsob voľby koeficientov nepopisujú. Hašovanie sa ukázalo ako jednoduchý, no nie príliš efektívny prístup. V prípade podobností metód vidíme oproti

vektorom obrovský nárast v počte nájdených zhôd, vďaka čomu by sa zvýšila náročnosť pri detailnom spracovávaní týchto zhôd.

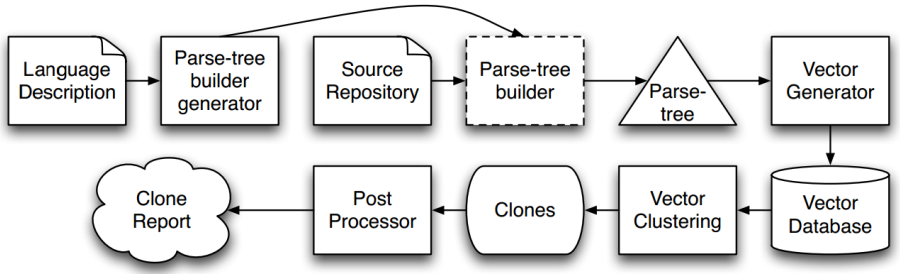
Použitie vektora, ako charakteristiky či už metódy alebo triedy, viedlo celkovo k lepším výsledkom. Implementácia tohto prístupu bola náročnejšia, pretože pre rôzne celky zdrojového kódu (trieda, metóda) sme výpočet tohto vektoru realizovali odlišne. Na základe vykonaných experimentov sme usúdili, že 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 číselných charakteristík. Tento rozdiel nie je taký výrazný hlavne preto, ako sú jednotlivé algoritmy implementované. Aj pre hašovanie aj pre číselné charakteristiky, 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

V našej práci sme sa rozhodli, na základe experimentov popísaných v predchádzajúcej kapitole, reprezentovať zdrojový kód pomocou charakteristických vektorov. Doteraz sme sa venovali len reprezentácii tried a metód pomocou vektorov. V tejto kapitole si tento prístup zovšeobecníme, pretože pri hľadaní plagiátov musíme vyhľadávať podobnosti aj na menších častiach zdrojového kódu akými sú metódy.

Ako inšpiráciu sme si zvolili prácu *DECKARD: Scalable and Accurate Tree-based Detection of Code Clones* [37], ktorú sme už spomínali v súvislosti s vyhľadávaním klonov v zdrojovom kóde. 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.



Obrázok 19: Architektúra systému DECKARD [37]

Na obrázku 19 je znázornená systémová architektúra nástroja DECKARD. 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.

Autori zabezpečili škálovateľnosť tohto nástroja vďaka využitiu *Locality Sensitive Hashing* (LSH) pri procese klasterizácie. V súčasnosti však existujú aj efektívnejšie spôsoby klasterizácie, akým je využitie LSH [41].

4.2.1 Definícia prostredia a pojmov

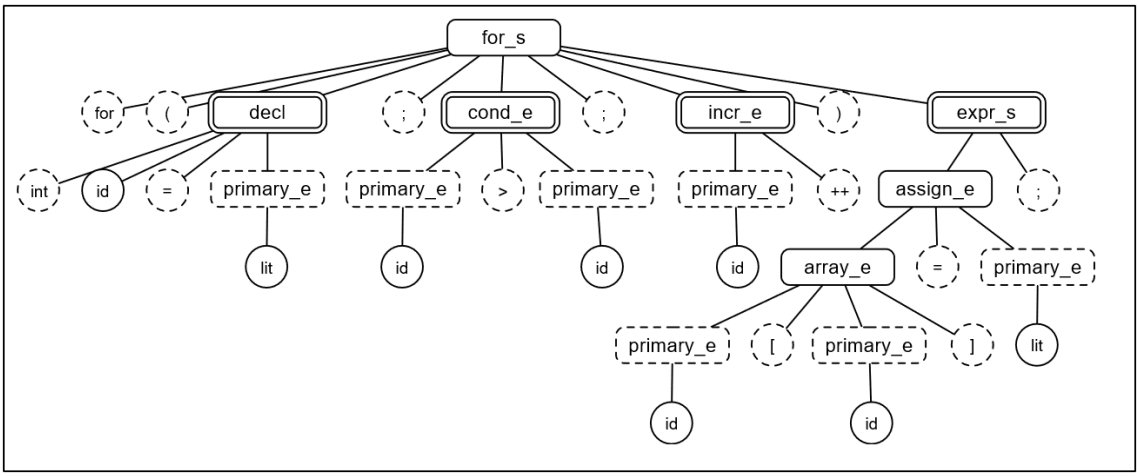
Pokiaľ chceme algoritmus popisovaný v spomenutej práci implementovať v našom prostredí, musíme si najskôr uvedomiť rozdiel v spracovaní zdrojového kódu. DECKARD používa parser na tvorbu tzv. parse-tree (strom, ktorý pozostáva z vhodne usporiadaných tokenov) a my používame *syntax analyzer* na tvorbu syntaktických stromov. Rozdiel týchto stromov si demonštrujeme na príklade, ktorý obsahuje kód validný aj v C++, ale aj v C#. Tento kód obsahuje cyklus, ktorý vynuluje prvky poľa.

```
for (int i = 0; i < n; i++)
{
    x[i] = 0;
}
```

Obrázok 20: Ukážka algoritmu - cyklus for

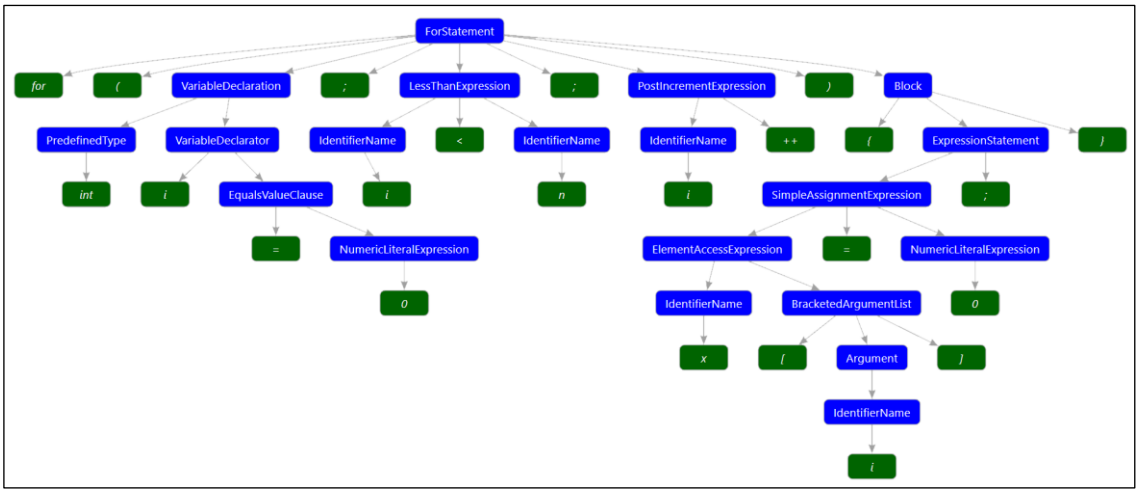
Ako prvý si popíšeme parse tree, ktorý používa DECKARD. Na obrázku 21 je zobrazený parse tree daného výrazu. V tomto strome nájdeme niekoľko druhov uzlov z pohľadu DECKARD algoritmu. Uzly zobrazené v krúžku sa nazývajú terminálne. Tieto terminálne uzly obsahujú priamo tokeny pochádzajúce zo zdrojového kódu a už nemajú potomkov. Ostatné uzly sú neterminálne. Ďalšie rozdelenie uzlov je na relevantné (orámované plnou) čiarou a irelevantné (označené čiarkovanou čiarou). Niektoré uzly sa pri generovaní vektorov môžu spájať (v obrázku orámované dvojitou čiarou).

Takéto rozdelenie autori zaviedli hlavne preto, lebo nie všetky uzly sú potrebné na zachytenie štrukturálnych informácií v kontexte, v akom sa používajú, alebo boli zavedené, aby zjednodušili gramatický popis daného jazyka. Takýmito uzlami môžu byť napríklad tokeny reprezentujúce zátvorky „(“ a „)“.



Obrázok 21: Ukážka parse tree používaného v nástroji DECKARD [28]

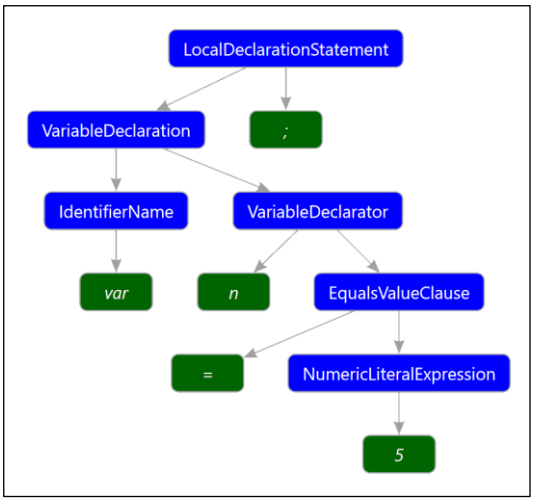
V našom prostredí používame syntaktický strom. Ukážka takéhoto stromu pre rovnaký kód ako v prechádzajúcom príklade, je zobrazená na obrázku 22.



Obrázok 22: Ukážka nami používaného syntaktického stromu

Oba stromy majú veľmi podobnú štruktúru. Ako vidíme, náš syntaktický strom obsahuje detailnejší popis jednotlivých blokov zdrojového kódu. Delenie na relevantné a irelevantné uzly sme už popisovali a využívali v predchádzajúcich kapitolách, a preto by sme ho naďalej zachovali. *SyntaxNode* uzly budeme považovať za relevantné a ostatné za irelevantné.

Na druhej strane, štruktúra syntaktického stromu je bohatá a v niektorých prípadoch sme sa rozhodli označiť aj niektoré zo *SyntaxNode* uzlov za irelevantné. Ako príklad môžeme ukázať syntaktický strom deklarácie lokálnej premennej. Na obrázku 23 môžeme vidieť takýto strom. V tomto prípade uzly *VariableDeclaration* a *VariableDeclarator* sú pre nás irelevantné, nakoľko priamo závisia na svojom rodičovi a nepridávajú novú informáciu o štruktúre kódu.



Obrázok 23: Syntaktický strom výrazu - deklarácia premennej

4.2.2 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. Autori vo svojej práci aj matematicky dokázali pomocou *Yang et al., Thm. 3.3* [42], že takto definované vektory dostatočne popisujú stromové štruktúry.

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. Parametrami tohto algoritmu sú *T* (syntaktický strom) a *M* (minimálna veľkosť

vektora). Minimálna veľkosť vektora slúži na obmedzenie množstva vygenerovaných vektorov. Jej cieľom je zamedziť generovanie vektorov z malých blokov kódu, ktoré sú pre analýzu podobností nevýznamné. Pod pojomom **veľkosť vektora** rozumieme súčet hodnôt jeho jednotlivých zložiek. Táto hodnota vyjadruje počet uzlov, ktoré daný vektor pokrýva.

```
1: function V_GEN(T : tree, M : int): vectors
2:     V ← ∅
3:     for all node N in (post-order T) do
4:         V_N ← ∑_{n ϵ childrens(N)} V_n
5:         if IsRelevant(N) then
6:             id ← IndexOf(N)
7:             V_N[id] ← V_N[id] + 1
8:         end if
9:         if IsSignificant(N) ∧ ContainsEnoughTokens(V_N, M) then
10:            V ← V ∪ {V_N}
11:        end if
12:    end for
13:    return V
14: end function
```

Algoritmus 1: Generovanie charakteristických vektorov

Algoritmus začína post-order prehliadkou syntaktického stromu. Pre každý uzol (N) začneme výpočet jeho charakteristického vektora (V_N) sčítaním vektorov jeho potomkov (riadok 4). Následne, pokiaľ sa jedná o relevantný uzol (implementované v metóde `IsRelevant`), zistíme pozíciu, ktorá je priradená uzlu N v charakteristickom vektore (implementované v metóde `IndexOf`, ktorá určuje príslušný index na základe typu uzla). Vo vektore V_N zvýšime hodnotu na príslušnej pozícii. V ďalšom kroku overíme, či sa jedná o významný uzol (pomocou metódy `IsSignificant`), a či vektor má dostatočnú veľkosť (metóda `ContainsEnoughTokens` kontroluje veľkosť vektoru voči parametru M). Ak vektor spĺňa obe požiadavky, pridáme ho do množiny vygenerovaných vektorov (V). Úplne nakoniec algoritmus vráti zoznam vygenerovaných vektorov (V).

V algoritme sa uvádza nový pojem – významný uzol. Vďaka tomuto konceptu môžeme obmedziť, na ktorých úrovniach syntaktického stromu sa budú generovať vektory. V našej implementácii, ale aj v implementácii DECKARD algoritmu, sú množiny významných a relevantných uzlov totožné.

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.

4.2.3 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. Uvažujme jednoduchý príklad:

```
1 class A {
2     void Method() {
3         int a = 5;
4         if (a > 0) {
5             a++;
6         }
7     }
8 }
```

Obrázok 24: Ukážka zdrojového kódu jednoduchej triedy

Z kódu na obrázku 24 budú v základnom algoritme (pri vhodnej voľbe parametra *M*) vygenerované vektory pre podmienku (riadky 4 – 6), metódu (riadky 2 – 7) a triedu (riadky 1 – 8). Pri spájaní vektorov budeme prechádzať po zdrojovom kóde postupne a môžeme získať (pri vhodnej voľbe parametrov) vektory napríklad pre riadky 1 – 3 alebo 3 – 6. Niektoré zo spájaných vektorov nemusia mať príliš veľký význam (riadky 1 – 3) a iné (riadky 3 – 6) môžu byť významné.

Algoritmus pracuje s vektormi uzlov, vygenerovanými v predchádzajúcom základnom algoritme. Algoritmus používaný nástrojom DECKARD sa skladá z niekoľkých krokov. V prvom kroku serializuje parse tree v post-order poradí. Následne sa po takto serializovanom strome posúva okno, v rámci ktorého pospája vektory uzlov, ktoré sa v tomto okne nachádzajú. Parametrami tohto algoritmu je veľkosť okna (definovaná minimálnou dĺžkou spájaného vektora) a posunom, o ktorý sa okno posunie po každom vygenerovanom vektore. Väčšie veľkosti okna zabezpečia, že väčšie časti kódu budú pospájané, a vďaka posunom môžeme určiť, ako sa budú tieto časti prekrývať.

V našom prípade nebolo možné tento algoritmus aplikovať priamo, nakoľko sme pri popisovanom algoritme narazili na problém viacnásobného sčítania vektorov. Pokiaľ pri post-order prehliadke stromu spracovávame len terminálne uzly (listy), zlúčené vektory

správne reprezentujú spracovanú časť stromu. Pokiaľ ale pri prehliadke traverzujeme smerom ku koreňu, tak sčítaním vektorov potomkov s vektorom otca získavame 2x započítané vektory potomkov, nakoľko tie už sú započítané vo vektore otca. Tento problém sme vyriešili tak, že okrem východiskového uzla sme pri spájaní uzlov použili ich vlastný vektor.

```
1: function V_MERGE(T : tree, M : int, S : int): vectors
2:     ST ← Serialize(T); V ← ∅
3:     step ← 0; front ← ST.first
4:     repeat
5:         if RightStep(step, M) then
6:             V_merged ← V_front
7:             n ← NextNode(front)
8:             while n != ST.last ∧ ¬ContainsEnoughTokens(V_merged, M) do
9:                 n ← NextNode(n)
10:                id ← IndexOf(n)
11:                V_merged[id] ← V_merged[id] + 1
12:            end while
13:        end if
14:        front←NextNode(front)
15:        step ← step + 1
16:    until front = ST.last
17:    return V
28: end function
```

Algoritmus 2: Spájanie vektorov

Parametrami algoritmu sú *T* (syntaktický strom), *M* (minimálna veľkosť vektora) a *S* (veľkosť kroku). Algoritmus postupne prechádza po serializovanom strome. Na začiatku každého kroku vyhodnotí, či sa jedná o správny krok (metóda *RightStep* podľa parametra *M*). Následne vlastný algoritmus spájania pozostáva z niekoľkúch krokov. Prvým z nich je priradenie priradenie východzieho vektora. Ďalej algoritmus pokračuje, a kým nemá výsledne spojený vektor dostatočnú veľkosť, alebo sme nenarazili na koniec, pripočítavame do výsledného vektora vlastné vektory nasledujúcich uzlov. Po dosiahnutí požadovanej veľkosti pridáme spojený vektor do množiny *V* a posunieme sa na ďalší krok. Úplne na konci algoritmus vráti zoznam vygenerovaných vektorov.

Takýto algoritmus spájania 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. Na príklade si ukážeme, kedy má takýto prípad význam.

```
1  class A {
2      void Method() {
3          int a = 5;
4          int b = 6;
5          int c = a * b;
6          int d;
7          if (c > 5) d = a;
8          else d = b;
9      }
10 }
```

Obrázok 25: Ukážka komplexnejšieho zdrojového kódu

V prípade tohto kódu budú základným algoritmom (pri vhodnej voľbe parametrov) vygenerované len 2 vektory (vektor pre triedu – 1 až 10 a vektor pre metódu – 2 až 9). Z výrazov obsiahnutých v metóde vektory generované nebudú, keďže ich vektory nemajú dostatočnú veľkosť. V prípade použitia nami navrhovaného algoritmu dokážeme generovať vektory z riadkov 3 – 5, 4 – 6, 5 – 7, 6 – 7, 7 – 8, ktoré nám pomôžu pri hľadaní čiastkových plagiátov.

Podobne ako v základnom algoritme spájania vektorov má aj náš algoritmus parametre T (syntaktický strom), M (minimálna veľkosť vektora) a S (veľkosť kroku). Algoritmus postupne prechádza jednotlivé uzly syntaktického stromu. Pokiaľ nájde neterminálny uzol, ktorý má dostatočne dlhý vektor (lebo nemá zmysel spájať potomkov, pokiaľ ani ich otec nemá dostatočnú dĺžku vektora), riadok 4, začne postupne spájať potomkov podobne ako v predchádzajúcom prípade. V tomto algoritme nespájame vlastné vektory, ako sme to robili v prechádzajúcom algoritme, ale robíme vektorový súčet vektorov daných uzlov (riadok 13). Na konci algoritmu vrátime zoznam nájdených vektorov.

```
1: function V_SIB_MERGE(T : tree, M : int, S : int): vectors
2:     V ← ∅
3:     for all node N in T do
4:         if IsNonTerminal(N) ∧ ContainsEnoughTokens(V_N) then
5:             CH ← N.childrens
6:             step ← 0; front ← CH.first
7:             repeat
8:                 if RightStep(step, M) then
9:                     V_merged ← V_front
10:                    n ← NextNode(front)
11:                    while n != CH.last ∧ ¬ContainsEnoughTokens(V_merged, M) do
12:                        n ← NextNode(n)
13:                        V_merged ← V_merged + V_n
14:                    end while
15:                end if
16:                front←NextNode(front)
17:                step ← step + 1
18:            until front = CH.last
19:        end if
20:    end for
21:    return V
22: end function
```

Algoritmus 3: Spájanie súrodeneckých uzlov do vektorov

4.2.4 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. Ukážme si príklad:

```
1  class A {
2      void Method() {
3          int a = 5;
4          int b = 6;
5          int c = a * b;
6          int d;
7          if (c > 5) d = a;
8          else d = b;
9      }
10 }
```

```
1 class B {
2     void MethodB() {
3         int x = 8;
4         int y = 4;
5         int z = a * b;
6     }
7 }
```

Obrázok 26: Príklad zdrojového kódu dvoch podobných metód

S použitím pôvodného algoritmu sme boli schopní detegovať klon medzi týmito dvoma triedami ako zhodu na riadkoch 1 až 5. S využitím nášho modifikovaného algoritmus sme našli zhodu len na riadkoch 3 až 5. 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. Menšie množstvo vygenerovaných vektorov zrýchli proces hľadania zhodných častí kódu, a taktiež aj pamäťové nároky potrebné pri tomto procese.

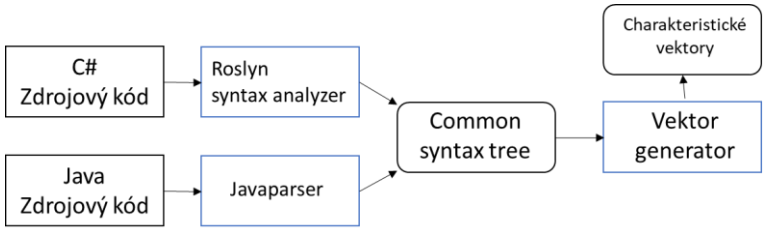
4.3 Adaptácia algoritmu na iný programovací jazyk

Všetky doteraz popisované spôsoby spracovania, vektorizácie a overovania boli realizované nad zdrojovým kódom naprogramovaným v programovacom jazyku C#. Pri spracovaní sme využívali nástroj *syntax analyzer*, ktorý dokázal transformovať zdrojový kód a reprezentovať ho pomocou syntaktického stromu. Ako sme v prechádzajúcej kapitole popisovali, náš algoritmus má výstup kompatibilný so systémom DECKARD.

Tým pádom sme schopní využiť možnosti systému DECKARD ako náhrady nášho algoritmu.

Napriek tomu, že systém DECKARD obsahuje podporu pre programovací jazyk Java, rozhodli sme sa implementovať do nášho algoritmu aj podporu pre tento jazyk, čím sme si overili univerzálnosť tohto systému. Vyber Javy nebol náhodný, *Java* je jeden z najpoužívanejších jazykov pri výučbe na našej fakulte, a ako si ukážeme v kapitole 6.3.2, aj DECKARD trpí zastaranosťou, a nedokáže dobre spracovať niektoré moderne napísané zdrojové kódy.

Pre spracovanie zdrojového kódu v *Jave* sme využili nástroj *javaparser*[5], ktorý nám umožňuje spracovávať zdrojový kód pomocou syntaktického stromu. Pre začlenenie do nášho algoritmu sme vytvorili modul, ktorý pomocou spomenutého nástroja spracuje zdrojový kód, a následne exportuje model zdrojového kódu vo vhodnej forme. Tento model je následne načítaný, a naším algoritmom vektorizácie je úplne rovnakým spôsobom spracovaný. Detail tejto metódy je zobrazený na obrázku 21.



Obrázok 27: Ukážka multijazyčného modulu na parsovanie zdrojového kódu

Pri zavedení podpory viacerých jazykov sme pridali ďalšiu vrstvu, ktorá bude spoločná pre všetky nástroje spracovávajúce zdrojový kód. **Spoločný stromový model zdrojového kódu** je len zjednodušením syntaktického stromu. Každý uzol tohto stromu obsahuje informáciu o type, rozsahu kódu, ktorý tento uzol pokrýva a zoznam potomkov. Pridanie podpory pre ďalší jazyk nebude zahŕňať nič viac, ako pripravenie modulu, ktorý transformuje zdrojový kód do formátu spoločného stromového modelu.

[5] https://javaparser.org/

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. V ďalšej časti si **popíšeme rôzne optimalizácie**, ktoré sme pri využívaní štandardného K-Means algoritmu museli vykonať, aby sme dokázali spracovávať **požadované množstvá zdrojového kódu** a spôsob, akým tieto optimalizácie prispejú ku škálovateľnosti algoritmu. Nakoniec navrhneme **metódu perzistencie** dát na základe klasteringu s využitím relačnej databázy.

5.1 Metódy zhlukovania

V literatúre môžeme nájsť rôzne prístupy, ktoré využívajú metódy zhlukovania zdrojového kódu. Existujú metódy, ktoré sa pokúšajú využitím klasteringu identifikovať určitým spôsobom význam zdrojového kódu [43][44] a odhaľovať podobnosti medzi rôznymi softérovými systémami [45]. Jeho využitie často nájdeme pri nástrojoch, ktoré pomáhajú vývojárom pri údržbe softvéru [46], vyhľadávaní chýb [47] a lepšom pochopení softvéru [48][49].

Klastering sa používa aj v oblasti vyhľadávania plagiátov v zdrojovom kóde [50][51][52], kde slúži ako nástroj, ktorého úlohou je na vyššej úrovni pospájať zdrojové kódy, a na základe vzniknutých skupín odhaľovať možné plagiáty.

Všetky spomenuté metódy pri spracovaní zdrojového kódu pracujú s kódom ako celkom. Našou úlohou bude klastrovať vektory, takže pri ďalších úvahách môžeme abstrahovať od pôvodu týchto vektorov, a pracovať s nimi ako s obyčajnými vektormi.

Tento postoj nám uľahčí prácu, pretože algoritmy, ktoré slúžia na zhlukovanie vektorov, existujú v dostatočnom počte a bežne sa využívajú.

5.2 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. Na vyčíslenie podobnosti môžeme použiť bežné metódy na výpočet vzdialenosti v n-dimenzionálnom vektorovom priestore [15] ako napríklad Euklidovskú či Mannhatanskú vzdialenosť. V našom algoritme sme sa rozhodli využiť Euklidovskú vzdialenosť na určenie podobnosti dvoch vektorov. Táto metrika je definovaná podľa vzorca

$$\boldsymbol{d(p,q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}} \tag{7}$$

kde p, q sú dva vektory a n je rozmer vektora.

Pri návrhu konkrétneho algoritmu na zhlukovanie charakteristických vektorov budeme vychádzať zo známych algoritmov. Existujú rôzne skupiny algoritmov [53][54], ktoré by sa dali využiť. Ako príklad môžeme uviesť algoritmy založené na centrách (napríklad K-Means), založené na rozložení dát či hustote (napríklad DBSCAN [55]). Jednotlivé

prístupy sme analyzovali, a ako najvhodnejší sa ukázal K-Means. Tento algoritmus sa zameriava na rozdelenie jednotlivých vektorov do *k* skupín, kde každý vektor následne priradí do práve jednej takejto skupiny (klastra). Toto priradenie sa realizuje na základe vzdialeností k centrám jednotlivých klastrov, kde sa daný vektor priradí do klastra, ku ktorého centru je najbližšie. Centrum následne slúži ako reprezentácia celého klastra.

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.

Efektívnosť K-Means algoritmu vo všeobecnosti závisí od počiatočného rozdelenia klastrov. Medzi bežne používané metódy inicializácie patria tzv. Lloydov-Forgyho algoritmus a náhodné rozloženie vstupných dát [56]. Prvý spomenutý algoritmus je založený na náhodnom výbere centier klastrov, a následnom priradení vektorov k týmto náhodne vybraným klastrom. Druhý algoritmus vektory náhodne rozdelí do *k* skupín a na základe týchto skupín následne spočíta centrá príslušných klastrov. Každá z metód má svoje výhody a nevýhody, ale prvá spomenutá metóda sa zvykne preferovať [56].

Druhá fáza začína so základným rozložením klastrov a snaží sa toto rozloženie optimalizovať. Táto fáza má dva kroky:

- priradenie vektorov ku klastrom,
- aktualizácia centier.

V prvom kroku sa každý vektor priradí na základe vzdialenosti do príslušného klastra. Môže sa stať, že vektor bude mať rovnakú vzdialenosť k viacerým centrám, no aj v takomto prípade bude priradený len k jednému z nich. Po priradení všetkých vektorov nasleduje krok aktualizácie centier. V tomto kroku sa vypočíta nové centrum ako priemer zo všetkých priradených vektorov. Tieto dva kroky sa opakujú dovtedy, kým sa poloha centier mení. K-Means algoritmus je príklad NP ťažkej úlohy. Popísaný algoritmus negarantuje, že takto nájdené riešenie bude optimálne, ale suboptimálne.

```
1: function K_Means(V : vector list, k : int, C : list of clusters): list of clusters
2:     repeat
3:         change ← 0
4:         UnassingAll(V)
5:         for all vector v in V do
6:             c ← Min(Distance(v, C_j) j∈{1..k})
7:             Assign(v,c)
8:         end for
9:         for all cluster c in C do
10:            change ← change + Recalculate(c)
11:        end for
12:    while change > 0
13:    return C
14: end function
```

Algoritmus 4: Optimalizačná časť K-Means algoritmu

Algoritmus 4 popisuje druhú fázu K-Means algoritmu. Tento algoritmus pozostáva z troch častí.

1. Zrušíme aktuálne priradenie vektorov ku klastrom.
2. Pre každý vektor spočítame vzdialenosť ku všetkým centrám klastrov a priradíme ho ku klastru s najmenšou vzdialenosťou.
3. Prepočítame centrá jednotlivých klastrov pomocou funkcie `Recalculate`. Táto funkcia okrem vypočítania nových centier spočíta aj rozdiel medzi starou a novou pozíciou centra.

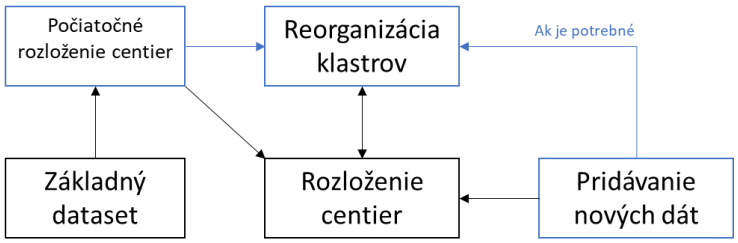
Tento proces sa opakuje dovtedy, kým sa nezmení poloha centier.

5.3 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.



Obrázok 28: Metóda inkrementálneho klasteringu

Na obrázku 28 môžeme vidieť schematický nákres popisovanej metódy inkrementálneho klasteringu. Metoda si počas celého času udržiava aktuálny stav klastrov, ktorý sa postupným pridávaním nových vektorov dostáva do neoptimálneho stavu, a v prípade potreby sa rozloženie klastrov optimalizuje kompletným prepočítaním centier.

5.4 Validácia inkrementálneho K-Means algoritmus

V predchádzajúcej kapitole sme navrhli inkrementálny algoritmus na klastrovanie charakteristických vektorov. V tejto kapitole sa budeme venovať validácii hypotéz, ktoré sme vyslovili pri návrhu algoritmu. Validácia bude prebiehať na datasetoch v programoch v programovacom jazyku C# s využitím nášho algoritmu na spracovanie zdrojového kódu popísanom v kapitole 4.2. Okrem validácie sa budeme venovať aj otázke, ako správne stanoviť potrebný počet klastrov.

5.4.1 Testované datasety

Testovanie navrhnutej metódy prebiehalo na niekoľkých datasetoch. V tejto sekcii popíšeme dva testované datasety, ktoré boli vybrané ako reprezentatívna vzorka. Tieto datasety budeme aj ďalej v práci označovať ako **Dataset1** a **Dataset2**. Dataset 1 obsahuje

zbierku študentských prác zozbieraných na predmete *Pokročilé objektové technológie*. Dataset 2 obsahuje niekoľko voľne dostupných zdrojových kódov z internetu. Pre každý dataset si uvedieme určité základné charakteristiky.

Pri generovaní charakteristických vektorov z týchto datasetov boli použité dve konfigurácie algoritmu na generovanie vektorov:

1. Minimálny počet tokenov = 30, posuvné okno = 0.
2. Minimálny počet tokenov = 50, posuvné okno = 2.

Dataset 1

Tento dataset pozostáva zo študentských prác naprogramovaných v programovacom jazyku C#. Celkovo obsahuje **227** úloh z **5** odlišných skupín od jednoduchých konzolových aplikácií po systémy na správu databázy s GUI rozhraním. Obsahuje **3989** súborov s **291862** riadkami kódu a **72107** riadkami komentárov. Celkovo bolo vygenerovaných **169013** charakteristických vektorov pre tento dataset. **79590** z nich bolo unikátnych.

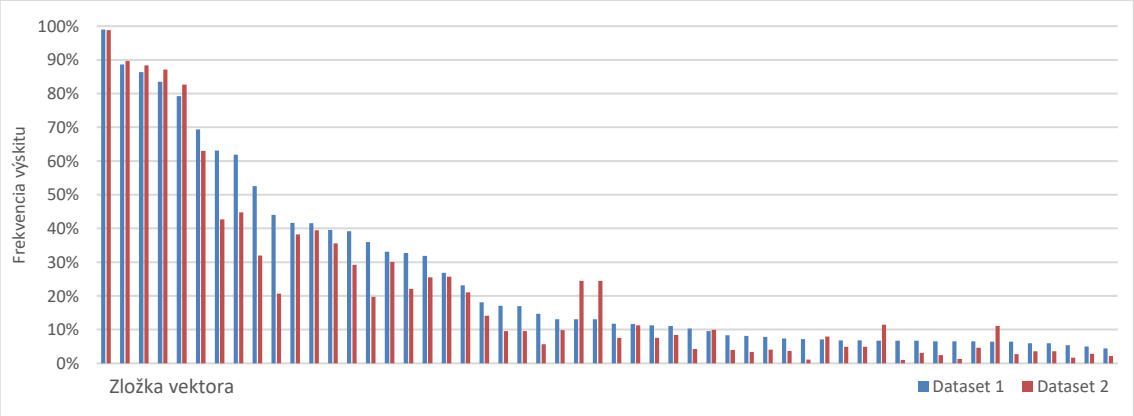
Dataset 2

Študentské práce sú často špecifickým príkladom zdrojového kódu, a niekedy sa ani nepodobajú na kód skutočného softvéru. Rozhodli sme sa do nášho porovnávania pridať aj zdrojový kód reálneho softvéru, ktorý vzniká mimo akademického prostredia. Preto, aby sme vedeli jednotlivé datasety porovnávať, sme sa sústredili na zdrojový kód v jazyku C#. Ako reprezentatívnu vzorku sme zobrali top 20 projektov z open-source repozitáru github.com označených ako C# a zoradených podľa počtu hviezd. Z týchto 20 projektov sme odstránili projekt, ktorý obsahoval zdrojové kódy systému Mono[6], pretože sám tento projekt má viac kódu ako zvyšných 19 projektov. Tým, že sme vyberali 20 projektov, sme chceli dosiahnuť určitú diverzitu, a výberom Mona, by sme o ňu podľa nášho názoru prišli. Týchto 19 projektov obsahovalo **19.629** súborov **3.334.620** riadkov kódu a **797.259** riadkov komentárov. Náš algoritmus vygeneroval **2.845.862** vektorov, z ktorých **848.981** bolo unikátnych.

[6] https://www.mono-project.com

5.4.2 Charakteristiky testovaných datasetov

Pre jednotlivé datasety sme okrem analýzy zdrojových súborov vykonali aj analýzu charakteristických vektorov, ktoré z týchto datasetov vznikli. Postupne analyzujeme zastúpenie jednotlivých zložiek vektora v datasete a priemernú „dĺžku vektorov“.



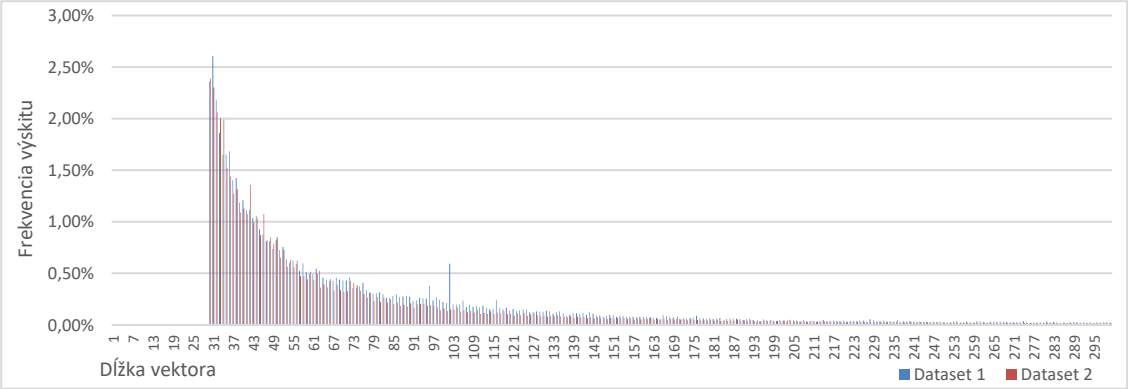
Obrázok 29: Porovnanie frekvencie výskytu jednotlivých zložiek vektora

Na obrázku 29 môžeme vidieť frekvenciu najpočetnejších 50 zložiek vektora v rámci datasetov. Prvky sú zoradené podľa frekvencie v datasete 1. Ako môžeme z priloženého obrázka vidieť, rozdelenie frekvencií jednotlivých zložiek vektora je v oboch testovaných datasetoch veľmi podobné aj napriek odlišnosti pôvodných zdrojových kódov týchto datasetov. Zaujímavý je aj zoznam najčastejších zložiek vektora, pretože tieto zložky priamo reprezentujú výskyt určitých konštrukcií v zdrojovom kóde. Medzi najčastejšie patria uzly vyjadrujúce:

1. `IdentifierName`,
2. `MemberAccessExpression`,
3. `ArgumentList`,
4. `Argument`,
5. `InvocationExpression`,
6. `LiteralExpression`,
7. `ExpressionStatement`,
8. `Block`,
9. `BinaryExpression`,
10. `AssignmentExpression`.

Najčastejším prvkom sú identifikátory (názvy tried, metód, premenných...). Tento výsledok bol očakávaný, pretože identifikátory sú základným prvkom v každom programovacom jazyku. Existuje množstvo prác [17][18], ktoré pri analýze využívajú práve tieto identifikátory na pochopenie zdrojového kódu. Medzi ďalšie významné prvky patrí prístup k atribútom a metódam, zoznam parametrov, volanie metód, literály, výrazy, bloky...

Druhou charakteristikou, na ktorú sa pozrieme je početnosť „dĺžok“ jednotlivých vektorov. Pod pojmom „dĺžka vektora“ budeme myslieť veľkosť podstromu syntaktického stromu, z ktorého daný vektor vznikol. Keďže jednotlivé zložky vektora reprezentujú početnosť daného typu uzlu v podstrome, z ktorého vektor vznikol, bude táto dĺžka vektora rovná súčtu všetkých hodnôt zložiek vektora.



Obrázok 30: Porovnanie dĺžok vektorov v datasete 1 a datasete 2

Rozloženie dĺžok vektorov od 1 po 300 môžeme vidieť na obrázku 30. Náš algoritmus na generovanie vektorov neobmedzuje maximálnu dĺžku vektora, preto sa v datasete nachádzajú aj vektory dlhšie ako 300, ale pre lepšiu názornosť sme zobrazili graf len po dĺžku 300, čo znázorňuje okolo 94% vektorov z prvého datasetu a 95% vektorov z datasetu druhého. Na základe tohto porovnania vieme povedať, že vektory v oboch datasetoch majú približne rovnaké rozdelenie dĺžok.

Na základe spomenutých charakteristík sa dá usúdiť, že aj napriek tomu, že datasety obsahovali kvalitatívne odlišný zdrojový kód, vygenerované vektory majú rovnaké charakteristiky naprieč obom datasetom.

5.4.3 Experimentálne určenie počtu klastrov

Základným problémom pri využití K-Means algoritmu je správne zvoliť počet klastrov – parameter *k*. V literatúre nenájdeme presnú metódu, pomocou ktorej by sme mali určovať počet klastrov. Pre určenie správneho počtu vykonáme niekoľko pokusov a budeme sledovať určité charakteristiky. Medzi sledované charakteristiky zaradíme:

- polomer klastrov,
- vzdialenosť medzi klastrami,
- entropia klastrov,
- podobnosť vektorov v klastroch.

Vzhľadom na to, že K-Means je algoritmus, ktorý zoskupuje vektory okolo určitého centra, tak veľkosť klastrov bude jedným z parametrov, ktoré sa oplatí sledovať. **Veľkosť klastrov** môžeme definovať viacerými spôsobmi, my sme sa rozhodli využiť polomer klastra ako mieru pre veľkosť. Pod pojmom **polomer klastra** chápeme najväčšiu vzdialenosť zo všetkých vektorov prislúchajúcich klastru od centra. Táto vzdialenosť je počítaná ako $r = \max\big(d(v,c)\big); v \in V$ kde V je množina vektorov priradená ku klastru, a c je vektor, ktorý reprezentuje centrum klastra. V tomto prípade funkcia d predstavuje Euklidovskú vzdialenosť. Naším cieľom bude **minimalizovať** priemernú hodnotu tejto veličiny, pretože ak sa nám podarí spraviť klastre dostatočne malé, potom každý klaster bude obsahovať len veľmi podobné vektory. Takéto rozdelenie klastrov by napomohlo k efektívnej identifikácii podobností v zdrojových kódoch. Je zrejmé, že vzhľadom na rozsiahlosť a množstvo tvarov, ktoré môže zdrojový kód nadobúdať, nebude ľahké nájsť takto malé klastre.

Ďalším dôležitým parametrom je **vzdialenosť medzi klastrami**. Pre jednoduchosť, pod pojmom vzdialenosť klastrov budeme myslieť Euklidovskú vzdialenosť ich centier. **Maximalizácia minimálnej vzdialenosti klastrov** by pomohla nášmu algoritmu. Na jednej strane požadujeme čo najmenšie klastre (čo spôsobí, že počet klastrov sa bude blížiť k počtu unikátnych vektorov), na druhej strane chceme dosiahnuť čo najväčšiu separáciu medzi klastrami, takže v konečnom dôsledku by mali vzniknúť malé, ale dobre rozložené klastre.

Tieto dva parametre by štandardne stačili na určenie správneho počtu klastrov, ale chystáme sa použiť klastering ako nástroj na predtriedenie dát, takže sme pridali ďalšie dva parametre – entropiu klastrov a podobnosť vektorov v klastroch.

Entropia klastrov je počítaná ako súčet entropií jednotlivých prvkov vektorov v každom klastri. Na výpočet entropie používame Shannonov vzorec [57] definovaný ako

$$\boldsymbol{H(X) = -\sum_{i=1}^{n} P(x_i)\, log_2 P(x_i)} \tag{8}$$

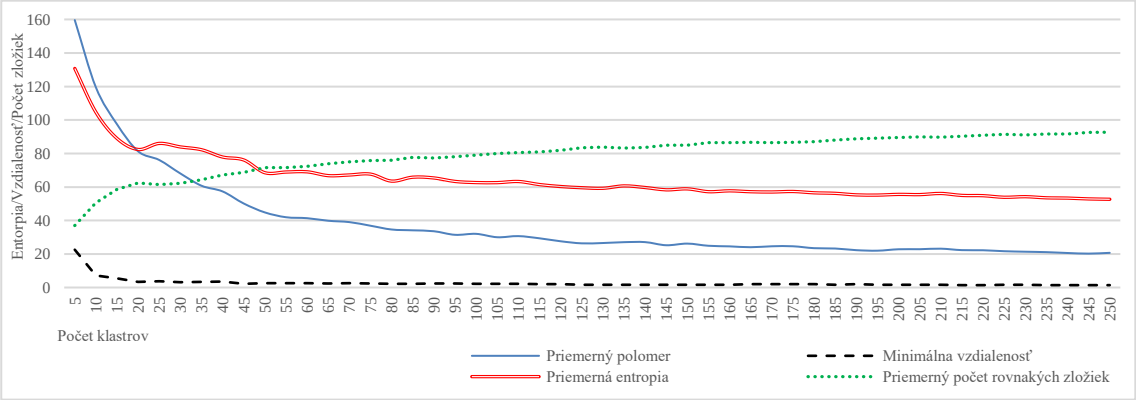
kde X je diskrétna náhodná premenná s možnými hodnotami $\{x_1, \dots, x_n\}$ a $P(X)$ je jej hustota pravdepodobnosti. Pri výpočte entropie klastra nemôžeme túto formulu použiť priamo. Najskôr musíme vypočítať entropiu každej zložky vektora zvlášť, a následne ich súčtom dostaneme entropiu klastra. Celkovú entropiu klastra môžeme zapísať ako $H(C) = \sum_{i=1}^{k} H(C_i)$, kde k je dimenzia vektora a C_i je rozdelenie pravdepodobnosti i-teho prvku vektora. Naším cieľom bude **minimalizovať priemernú entropiu** klastrov, pretože

menšia entropia znamená, že klaster obsahuje menej diverzifikované vektory, čo je lepšie pre účely vyhľadávania.

Posledná meraná charakteristika taktiež úzko súvisí s entropiou. Charakteristika „**podobnosť vektorov v klastri**“ reprezentuje počet zložiek vektora, ktoré majú v rámci klastra nulovú entropiu. Touto charakteristikou sledujeme počet zložiek vektora, ktoré sú rovnaké v rámci celého klastra. Tento počet nám dá priamočiary odhad, o koľko nám klastering zjednoduší vyhľadávanie podobných vektorov. **Maximalizácia** tejto hodnoty je veľmi dôležitá pre odhad počtu klastrov.

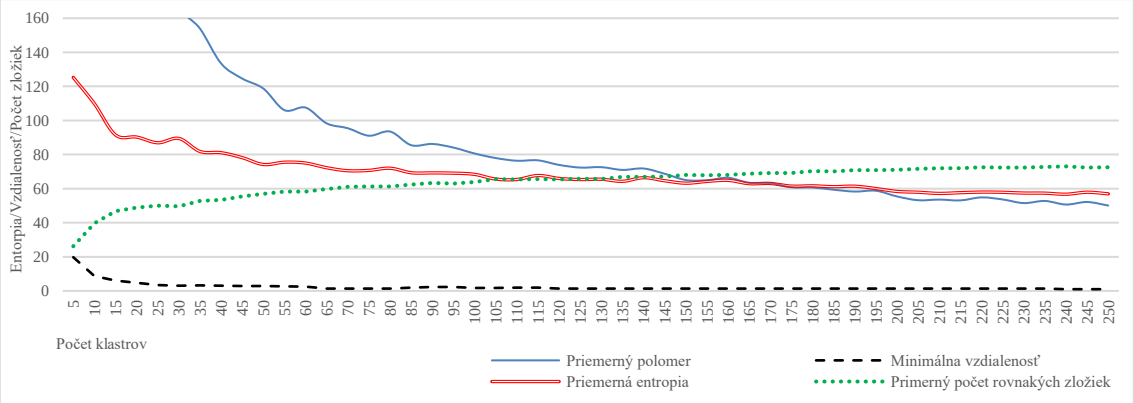
5.4.4 Vyhodnotenie experimentálneho určenia počtu klastrov

Po definovaní sledovaných charakteristík sme spustili algoritmus K-Means niekoľkokrát s rozdielnou hodnotou parametra *k*. Začali sme s počtom 5 a postupne sme tento počet zvyšovali s krokom 5 až po hodnotu 250. Výsledky pre dataset 1 môžeme vidieť na obrázku 31 a výsledky pre dataset 2 na obrázku 32.



Obrázok 31: Charakteristiky klastrov pre dataset 1

Keď porovnáme výsledky pre jednotlivé datasety zistíme, že sa až tak nelíšia. Jediným rozdielom medzi nimi je, že v druhom prípade máme pomalší nábeh a ustaľovanie jednotlivých veličín. Na základe ďalšej analýzy sme zistili, že tieto rozdiely sú spôsobené rozdielnou veľkosťou datasetov. Dataset 2 obsahuje približne 10x viac unikátnych vektorov. Pri zmenšení veľkosti druhého datasetu náhodným výberom vektorov z neho sme dostali charakteristiky, ktoré sa správali približne rovnako ako v prípade prvého datasetu.



Obrázok 32: Charakteristiky klastrov pre dataset 2

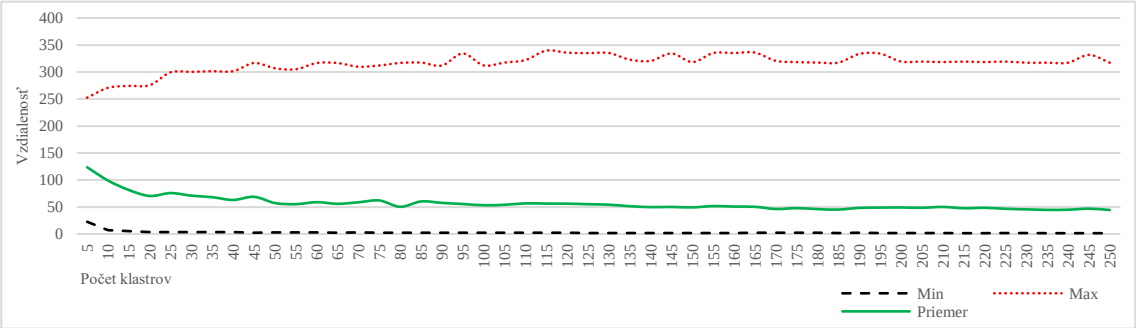
Pri bližšom pohľade na jednotlivé charakteristiky môžeme vidieť niekoľko zaujímavých vlastností. Priemerná hodnota **veľkosti klastrov** s narastajúcim počtom klastrov klesá až do hodnoty 50 v prvom prípade, a 70 v prípade druhom. Následne so zvyšujúcim sa počtom klastrov je jej pokles čoraz viac menší. Ako sme spomínali predtým, naším cieľom je minimalizovať túto hodnotu, no ako sa ukazuje, zvyšovanie počtu klastrov nad určitú úroveň nedáva zmysel.

Rovnakú situáciu môžeme pozorovať aj v prípade **priemernej entropie** a počtu **rovnakých zložiek vektora** v rámci klastra. Na rozdiel od charakteristiky veľkosť klastrov, charakteristika počet klastrov, pri ktorých táto hodnota prestane výrazne klesať, je menšia. Priemerná entropia s narastajúcim počtom klastrov podľa očakávania klesá. Možno to z obrázkov 31 a 32 nie je jasné, ale aj v tomto prípade platí zákon zachovania entropie – entropia sa nemôže stratiť. S narastajúcim počtom klastrov je entropia znížená o informáciu, ktorú získame pridaním ďalších klastrov. Ako z grafov vidíme, čím viac klastrov pridáme, tým menej informácie za každý ďalší klaster získame.

Poslednou meranou charakteristikou bola **minimálna vzdialenosť** medzi klastrami. Táto vzdialenosť sa s narastajúcim počtom klastrov zmenšuje, tak ako sme predpokladali. Túto zmenu môžeme ale pozorovať len v prvých pár krokoch, a potom aj s narastajúcim počtom ostáva konštantná. Táto vlastnosť vznikla pre to, lebo vektory nie sú v priestore rozmiestnené náhodne, ale vznikajú časti, kde je vektorov veľa a kde ich je naopak málo. V závislosti od počiatočného rozloženia klastrov [58] sa môže stať, že viacero centier bude umiestnených do určitého lokálneho zhluku vektorov. Pokiaľ je tento zhluk dostatočne izolovaný, centrá sa nedokážu presunúť inam a vznikne sub-optimálne riešenie, ktoré sa pridávaním počtu klastrov nedokáže zlepšiť. Štandardný K-Means algoritmus nedokáže

tento problém riešiť. Dá sa mu však predísť sofistikovanejšími metódami výberu počiatočných centier.

Na vyriešenie tohto problému sme sa rozhodli zmeniť túto poslednú charakteristiku, a namiesto minimálnej vzdialenosti z pomedzi všetkých klastrov budeme maximalizovať priemernú minimálnu vzdialenosť všetkých klastrov. Tento spôsob nám umožní lepšie monitorovať rozloženie jednotlivých klastrov. Pre úplnosť môžeme sledovať aj maximálnu minimálnu vzdialenosť medzi klastrami.

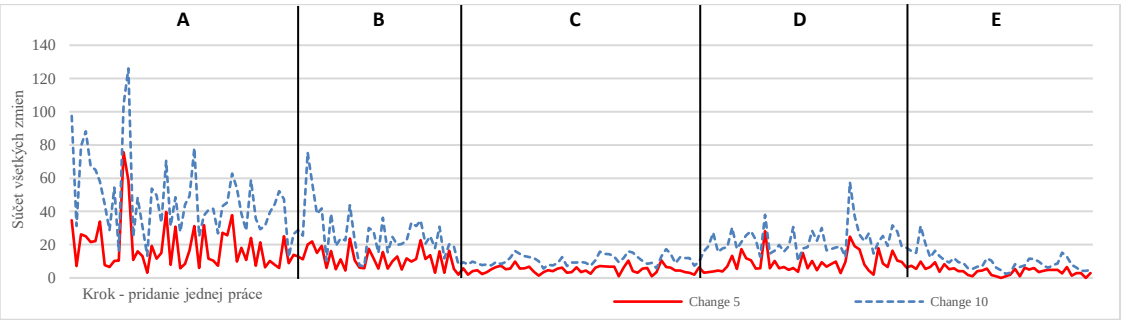


Obrázok 33: Minimálna vzdialenosť medzi klastrami - dataset 1

Ako môžeme na obrázku 33 vidieť, priemerná hodnota klesá podobne ako minimálna. Maximálna hodnota v tomto prípade je taktiež zaujímavá. Postupne stúpa, až dosiahne maximum, a potom už len osciluje. To môže znamenať, že klastre sú už dobre rozložené a pridávanie ďalších klastrov len rozdelí niektoré väčšie na viacero menších.

5.4.5 Efektivita inkrementálneho K-Means algoritmu

Základným predpokladom pre funkčnosť metódy inkrementálneho klasteringu je to, že s postupným pridávaním ďalších a ďalších vektorov sa pozícia klastrov ustáli a pridávanie nových vektorov nebude spôsobovať výrazne posuny v rozložení klastrov. Na overenie tejto hypotézy sme vykonali experiment. Využili sme dataset 1 (pretože obsahuje študentské práce, tj. reprezentuje reálne dáta, ktoré má takýto systém spracovávať). Na začiatok sme pripravili základnú množinu prác (vybrali sme prvých 10) a vykonali klasický K-Means algoritmus s počtom klastrov rovným 100. Potom sme pokračovali, a postupne sme pridávali do systému ďalšie práce po jednej. Po pridaní každej práce sme vykonali reklastering a merali sme zmenu pozícii centier jednotlivých klastrov. Zmena pozície je definovaná ako súčet euklidovských vzdialeností všetkých centier voči ich predošlému stavu. Vzhľadom na predpoklad oscilácie centier okolo určitého priemeru sme počítali aj zmenu voči priemeru centier z posledných 5 pozícií, a posledných 10 pozícií.



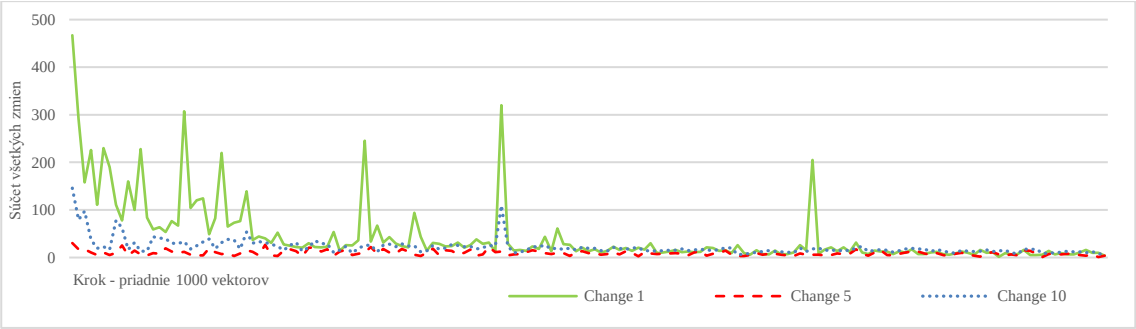
Obrázok 34: Zmena polohy centier klastrov pri postupnom pridávaní prác

Na obrázku 34 je možné vidieť súčet zmien polôh centier klastrov pri postupnom pridávaní nových prác. Graf zobrazuje len zmeny voči priemeru predchádzajúcich piatich, resp. desiatich pozícií, nakoľko sa potvrdila oscilácia v prípade okamžitých zmien.

Graf sme rozdelili na 5 častí (od A po E). Každá z týchto častí reprezentuje jeden z piatich typov úloh, ktoré sa v našom datasete nachádzali. Pokiaľ budeme skúmať každý sektor individuálne, môžeme pozorovať určité trendy. Vo väčšine prípadov môžeme pozorovať postupný pokles vo veľkosti zmien. V častiach A a B môžeme vidieť pokles vo veľkostí zmien, ktorý je spôsobený určitým ustaľovaním systému vďaka narastajúcemu počtu vektorov. V časti C sú veľkosti zmien veľmi malé, čo bolo spôsobené rozsahom zadania úloh, ktoré boli pridávané do systému v časti C. V sektore D opäť začínajú zmeny rásť, tentoraz spôsobené väčšou voľnosťou zadania prác, ktoré sa spracovávali v tomto sektore. V poslednej časti môžeme vidieť podobné správanie ako v časti C.

Zaujímavým zistením bolo, že zmena voči priemeru posledných desiatich pozícii je zvyčajne väčšia ako zmena voči priemeru posledných piatich pozícií. Na základe tohto pozorovania vieme konštatovať, že aj napriek tomu, že sa klastre postupne stabilizujú, tak pri tejto stabilizácii neoscilujú okolo určitej centrálnej pozície, ale pomaly sa presúvajú.

Hlavným problémom tohto experimentu bola diverzita dát – každá úloha bola trochu iná. Práve táto diverzita spôsobuje, že klastre sa pridávaním ďalších a ďalších prác neustále posúvajú. Na potvrdenie našej prvotnej hypotézy, že s narastajúcim množstvom dát sa klastre budú ustaľovať, sme sa pokúsili túto diverzitu odstrániť tým, že sme síce použili rovnaké dáta, ale zobrali sme všetky vektory zo všetkých úloh, náhodne ich premiešali, a postupne sme takto premiešané vektory pridávali do systému v skupinkách po 1000. Po pridaní 1000 vektorov sme vykonali reklastering a ako v predchádzajúcom prípade merali zmenu v rozložení klastrov. Výsledky môžeme vidieť na obrázku 35.



Obrázok 35: Zmena polohy centier klastrov pri pridávaní premiešaných dát

Na grafe môžeme vidieť zaujímavé správanie. Pokiaľ sa pozeráme na zmeny oproti predchádzajúcim pozíciám, vidíme postupné ustaľovanie centier s niekoľkými výkyvmi. Tieto výkyvy boli pravdepodobne spôsobené vložením vektoru, ktorý popisuje určitú unikátnu programátorskú techniku, ktorá sa doposiaľ v našom datasete neobjavila. Táto zmena napriek tomu nespôsobuje výkyv v dlhodobom meradle, ako môžeme vidieť na zvyšných dvoch krivkách. Opäť ako v prechádzajúcom prípade môžeme zo začiatku vidieť že zmena voči priemeru posledných 10 pozícií je väčšia ako zmena voči priemeru posledných 5 pozícií. Na rozdiel od predchádzajúceho prípadu dochádza k rýchlemu ustáleniu aj týchto hodnôt, až sa v dlhodobom rozsahu jednotlivé zmeny už takmer vôbec neprejavujú.

5.4.6 Zhodnotenie

V tejto kapitole sme sa venovali validácii metódy inkrementálneho klasteringu. Na začiatku 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.

5.5 Vyhľadávanie vektorov s využitím klasteringu

Jednou z hlavných operácií, ktorú od našej metódy požadujeme, je vyhľadávanie vektorov. Cieľom je pre daný vektor vyhľadať v systéme vektor, alebo vektory, ktoré sú mu podobné. V súčasnom návrhu metódy sa uspokojíme s vyhľadávaním totožných vektorov. Ako základ na vyhľadávanie nám poslúžia vytvorené klastre.

Algoritmus vyhľadávania vektorov môžeme rozdeliť na dva kroky:

1. Pre daný vektor nájdi prislúchajúci klaster.
2. Prechádzaj vektory vo vybranom klastri a vyber zhodné.

Prvý krok algoritmu je jednoduchý; spočítaj Euklidovskú vzdialenosť vyhľadávaného vektora a všetkých centier. Vyber klaster na základe najmenšej vzdialenosti. V prípade, že je viacero klastrov s najmenšou vzdialenosťou alebo požadujeme aj vyhľadávanie podobných, a nie len zhodných vektorov, preskúmaj aj ostatné blízke klastre.

Ako vidíme aj v tomto prípade, počet klastrov je dôležitý, pretože pri veľkom množstve klastrov by bolo náročné nájsť klaster s najmenšou vzdialenosťou k danému vektoru.

5.5.1 Vyhľadávanie vektora v rámci jedného klastra

Po nájdení príslušného klastra potrebujeme zadefinovať aj postup, pomocou ktorého budeme hľadať vektory v rámci jedného klastra. Základnou metódou na urýchlenie vyhľadávania je indexácia dát. Skúsme sa pozrieť na možnosti indexácie vektorov v rámci klastra. Indexácia nám sama o sebe zrejme stačiť nebude. Pri vyhľadávaní použijeme index len na zúženie množstva vektorov, ktoré budeme musieť prejsť a kompletne porovnať.

5.5.2 Indexovanie vektorov v rámci klastra

V tejto kapitole si rozoberieme možnosti indexácie vektorov v klastroch. Samozrejme, mohli by sme indexovať celý vektor. Metódy, ktoré toto dokážu existujú [59], ale index, ktorý by pokrýval celý vektor, by bol enormne veľký a jeho údržba by bola náročná

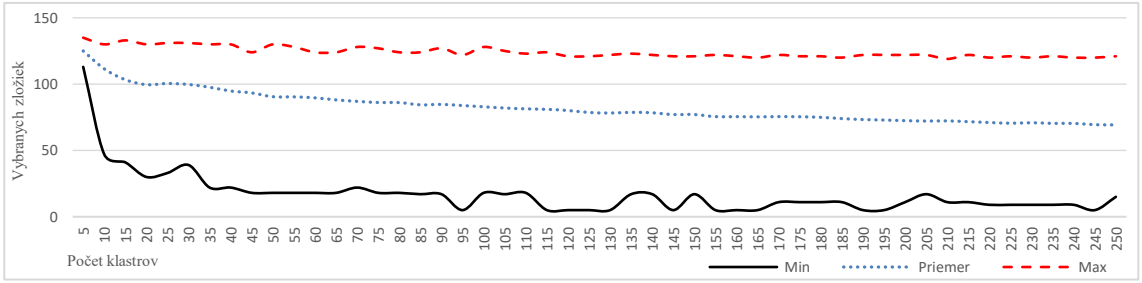
(hlavne vo fáze pridávania nových vektorov). Naším cieľom je vyhľadať presnú zhodu, takže nepotrebujeme pri konštrukcii uvažovať o multidimenzionálnom vyhľadávaní aj keď máme multidimenzionálne dáta. Takéto metódy vyhľadávania sú často veľmi komplexné. My potrebujeme kompozitný index. Štandardné lineárne indexy (založené na B-stromoch, používané v databázových systémoch [60]) sú na implementáciu kompozitných indexov dostatočné. Ak chceme ale takýto index využiť, musíme vybrať zložky vektora, pomocou ktorých budeme celý vektor indexovať. Na to, aby bol takýto index efektívny, musíme vybrať na jednej strane čo najmenší počet zložiek, no na druhej strane tieto zložky musia byť dostatočné na efektívne nájdenie príslušného vektora. Na výber významných zložiek vektora sme vyskúšali niekoľko metód:

- Výber zložiek, ktoré sú odlišné v rámci klastra.
- Výber zložiek s veľkou entropiou.
- Postupný výber zložiek na základe podmienenej entropie.

Jednotlivé metódy si popíšeme v nasledujúcich kapitolách.

Výber zložiek, ktoré sú odlišné v rámci klastra

Prvou a najjednoduchšou metódou, o ktorej sme už aj uvažovali pri analýze počtu klastrov, je výber zložiek na základe toho, či sú v danom klastri vo všetkých vektoroch rovnaké alebo nie. Pôvodne sme predpokladali, že vektory po klasteringu budú mať v rámci klastrov už len malú diverzitu. Tento predpoklad sa ale nepotvrdil.



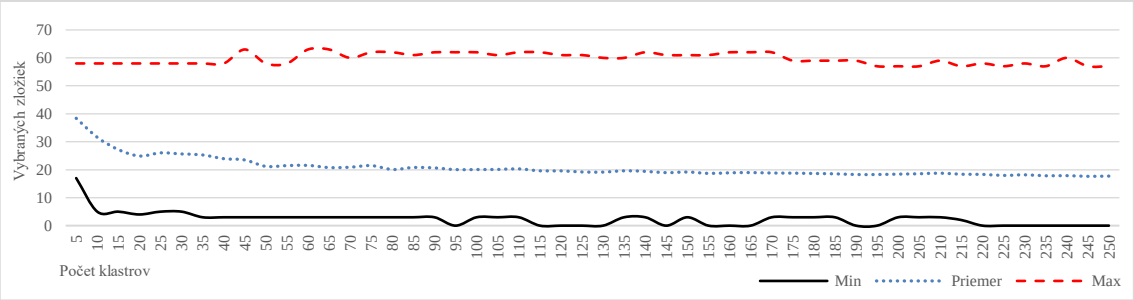
Obrázok 36: Počet zložiek vektora, ktoré nie sú rovnaké v rámci celého klastra

Na obrázku 36 môžeme vidieť minimálny, maximálny a priemerný počet zložiek vektora, ktoré nie sú rovnaké v rámci klastra v závislosti od počtu klastrov. Ako príklad si zoberme počet klastrov rovný 100. Pri tomto počte by sme s touto metódou museli indexovať v priemere 82 zložiek, čo je takmer polovica celej dimenzie vektora. Problémom ale je, že v najhoršom možnom prípade by bolo nutné indexovať až 128 z celkových 162 zložiek vektora. Je zjavné, že tento prístup asi nebude vhodný, pretože na

jednej strane, indexovaním všetkých navrhovaných zložiek by sme pri vyhľadávaní vektorov získali odpoveď už na základe prehľadávania indexu, a dodatočné filtrovanie by už potrebné nebolo, no indexovať čo i len 80 zložiek nie je jednoduché.

Výber zložiek na základe entropie

Druhým navrhnutým spôsobom na výber zložiek na indexovanie je výber zložiek na základe entropie. Tento prístup je na rozdiel od predchádzajúceho založený na fakte, že nepotrebujeme indexovať zložky tak, aby sme dostali presnú odpoveď už pri prehľadávaní indexu, ale vytvorený index bude slúžiť len na ďalšie prefiltrovanie. Pri výbere zložiek budeme vyberať zložky s najvyššou entropiou, pretože už z definície entropie je zrejmé, že filtrovanie na základe takýchto zložiek nám umožní dobre pred-filtrovať výsledky. Entropiu zložiek vektora sme už definovali pri výpočte entropie klastrov v kapitole 5.4.2. Pri výbere prvkov budeme vyberať len tie, ktorých entropia je väčšia ako 1. Minimálny, maximálny a priemerný počet zložiek s entropiou väčšou ako 1 v závislosti od počtu klastrov je zobrazený na obrázku 37.



Obrázok 37: Počet zložiek vektora s entropiou > 1

Na rozdiel od prvej metódy je priemerný počet zvolených zložiek touto metódou oveľa nižší. Pri počte klastrov 100 potrebujeme v priemere len 20 indexovaných zložiek oproti 82 v predchádzajúcom prípade. Ďalším zaujímavým faktom je to, že maximálny počet indexovaných zložiek je celkom stabilný a udržuje sa na hodnote 60, čo je menej ako priemer 82 v predchádzajúcom prípade. Aj napriek týmto zlepšeniam je 20 v priemere a 60 v najhoršom možnom prípade priveľa na indexovanie.

Výber zložiek na základe podmienenej entropie

Posledným spôsobom na výber zložiek vektora, ktoré budeme indexovať, je výber zložiek na základe podmienenej entropie. Tento prístup využíva fakt, že jednotlivé zložky vektora nie sú medzi sebou nezávislé. Pri generovaní vektorov sa snažíme odstrániť nevýznamné uzly, ale ostáva množstvo takých, ktoré sú na sebe závislé. Ako príklad

uvažujme vektor, ktorý vznikol z uzla, definujúci metódu. Ak vieme že zložka, ktorá označuje „počet definícií metódy“ má hodnotu „1“, potom vektor bude celkom logicky obsahovať „1“ aj v početnosti zložiek, ktoré reprezentujú napríklad „zoznam parametrov metódy“ alebo „telo metódy“, takže vidíme jasnú závislosť.

Táto závislosť poukazuje na to, že výber zložiek čisto na základe ich vlastnej entropie nebude najefektívnejším spôsobom. Pri výbere zložiek musíme určitým spôsobom zobrať do úvahy aj tieto závislosti. Metódy na výber najdôležitejších komponentov existujú [61], no využívajú sa často ešte pred samotným klasteringom.

Metóda výberu zložiek na základe podmienenej entropie sa využíva hlavne pri konštrukcii rozhodovacích stromov [62], ktoré majú rovnaké požiadavky ako náš algoritmus. Podmienená entropia je definovaná ako

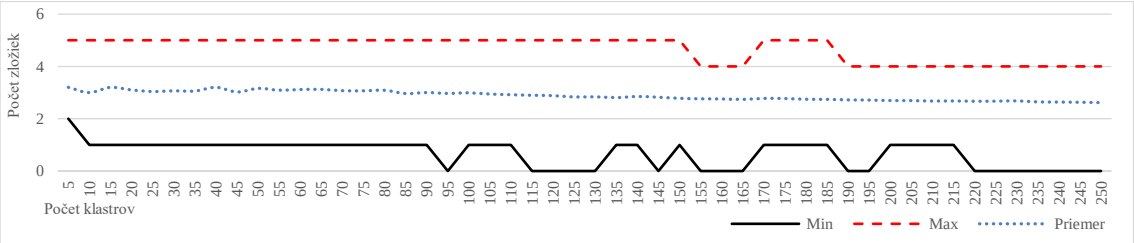
$$\boldsymbol{H(Y|X) = H(X,Y) - H(X)} \tag{9}$$

kde X a Y sú dve náhodné premenné popisujúce rozdelenie konkrétnych zložiek vektora a $H(X,Y)$ je ich združená entropia. Naším cieľom je nájsť takú sekvenciu zložiek $Y_1, \dots Y_n$, ktorá má najvyššiu podmienenú entropiu $H(Y_n|Y_1, \dots Y_{n-1})$. Algoritmus na vyhľadanie takejto postupnosti je nasledovný:

```
1: Y1 ← indexOf(max(H(Xi)) for all Xi in X
2: n ← 1
3: while exists(i in X; H(Xi| Y1..Yn) > 1) do
4:     Yn ← indexOf(max(H(Xi| Y1..Yn)) for all Xi in X
5:     n ← n + 1
6: end while
```

Algoritmus 5: Výber elementov na základe podmienenej entropie

Výsledkom tohto algoritmu je pre každý klaster postupnosť zložiek vektora, ktorá je vhodná na indexovanie. V algoritme vyberáme zložky pokiaľ existuje zložka s podmienenou entropiou > 1. To znamená, že pri vyhľadávaní vektorov nám nebude ako v predchádzajúcom prípade stačiť prehľadávať len index, ale bude potrebné aj následné dofiltrovanie výsledkov. Pokiaľ by sme chceli vyhľadávať vektory len pomocou indexu, musíme upraviť podmienku na výber sekvencie zložiek a vyberať zložky, až kým vyberieme všetky nezávisle, tj. podmienená entropia ostatných zložiek bude nulová.



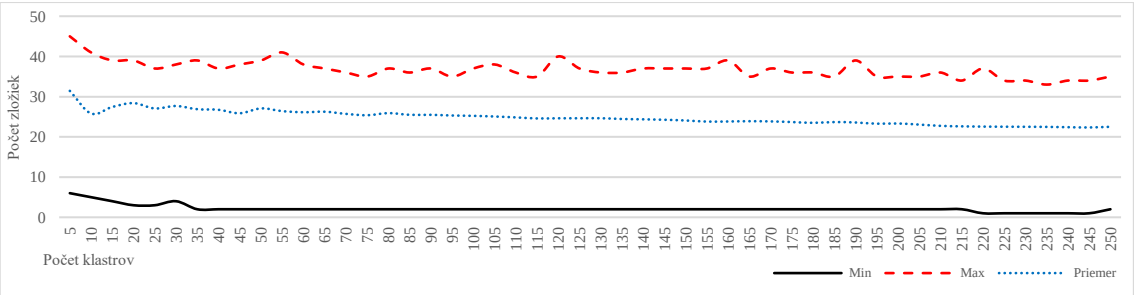
Obrázok 38: Počet zvolených zložiek na základe podmienenej entropie

Na obrázku 38 môžeme vidieť minimálny, maximálny a priemerný počet zložiek vektora, ktoré treba zvoliť na to, aby zvýšené zložky mali podmienenú entropiu menšiu ako 1. Tieto výsledky ukazujú, že ak budeme voliť zložky správnym spôsobom, tak v priemere nám stačí zvoliť 3 a maximálne 5. Jednotlivé zvolené zložky sa líšia klaster od klastra, takže to znamená že aj klastering prináša určité benefity. Experimenty ukazujú, že počet potrebných zložiek, ktorý by sme museli zvoliť ak by sme nepoužívali klastering a priamo indexovali celú množinu, by bol pre náš dataset rovný 6.

Zaujímavé bolo zistiť, ktorých 6 zložiek vektora je podľa tohto prístupu najvýznamnejších.

- *LiteralExpression* – literály (konštanty).
- *MemberAccessExpression* – prístup k metóde, atribútu objektu.
- *Argument* – parameter metódy a funkcie.
- *IdentifierName* – názvy tried, metód, premenných…
- *InvocationExpression* – volanie metódy, funkcie.
- *AssignmentExpression* – priradenie.

Pre ilustráciu ešte uvedieme počet zložiek potrebných k úplnej indexácii klastrov. Získané hodnoty môžeme vidieť na obrázku 39. Ako môžeme vidieť, v tomto prípade je počet potrebných prvkov rádovo väčší.



Obrázok 39: Počet potrebný zložiek na úplnú indexáciu klastrov

5.5.3 Zhodnotenie

V predchádzajúcich kapitolách sme sa venovali metódam vyhľadávania vektorov. Toto 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. V prípade nášho datasetu (dataset 1) nám stačí priemerne vyhľadávať podľa 3 zložiek, a už sme schopní požadovaný vektor dohľadať dostatočne rýchlo. V prípade druhého datasetu sme síce výsledky neprezentovali, ale priemerný počet zložiek sa v tomto prípade pohyboval okolo 4.

Indexovať 5 zložiek vektora nie je žiaden problém, preto aj v ďalšej práci budeme fixne používať index zložený práve z piatich prvkov.

5.6 Efektivita výpočtovej zložitosti K-Means algoritmu

Základný algoritmus K-Means vo svojej špecifikácii popisuje len princíp, ale nepopisuje možnosti efektívnej implementácie. Pokiaľ chceme využívať tento algoritmus v našej metóde, potrebujeme navrhnúť a implementovať rôzne vylepšenia. Pri vylepšovaní algoritmu musíme zvážiť, ktoré časti má pre nás zmysel optimalizovať, a ktoré nie. Vo väčšine prípadov sa snažíme vylepšovať prvotnú fázu algoritmu – nájdenie počiatočných centier [63][64]. Toto rozloženie je dôležité z dvoch dôvodov – lepšie štartovacie rozloženie zrýchľuje algoritmus a dokáže odstrániť problém suboptimálnych riešení, kde algoritmus skĺzne do lokálneho minima [65].

Napriek dôležitosti tohto počiatočného rozloženia sme sa rozhodli v tejto kapitole nevenovať optimalizácii počiatočného rozloženia klastrov. Pre náš algoritmus až tak zmysel nemá, nakoľko sa vykoná len na začiatku, a počas celého behu sa už len prepočítavajú centrá. Pri tomto prepočte sa vychádza zo stavu, ktorý bol predtým, a nie je nutné inicializovať klastre nanovo. Algoritmus K-Means (algoritmus 4) je relatívne jednoduchý, takže má len pár aspektov, ktoré dokáže zefektívniť. Naša analýza ukázala, že najviac problematickou časťou je priraďovanie vektorov do klastrov, pretože táto operácia ma štandardne zložitosť $O(n * k)$ kde n je počet vektorov a k je počet klastrov. Pre každú kombináciu vektor – klaster je nutné vypočítať vzdialenosť, a výpočet Euklidovskej vzdialenosti pre vektory dĺžky 162 taktiež trvá určitú dobu.

V tejto sekcii sa zameriame na optimalizáciu druhej časti K-Means algoritmu, pretože tú budeme vykonávať často. Vylepšovať budeme algoritmus v niekoľkých rovinách:

- pridanie paralelizácie,
- vylepšenie logickej štruktúry algoritmu,
- vylepšenie implementačných techník,
- predspracovanie vstupných dát.

V nasledujúcich kapitolách sa budeme venovať jednotlivým technikám. Na záver zhodnotíme spomínané techniky a porovnáme rýchlosť algoritmu pred a po aplikácii týchto techník.

5.6.1 Paralelizácia algoritmu

Jednou zo základných techník zrýchlenia algoritmov je ich paralelizácia. V literatúre môžeme nájsť viacero metód, ktoré sa snažia paralelizovať K-Means algoritmus [66][67]. V našej práci sme sa rozhodli preskúmať a implementovať niektoré základné metódy.

Paralelizácia algoritmov vo všeobecnosti nie je triviálnou úlohou [68]. Cieľom je čo najviac využiť dostupnú výpočtovú kapacitu CPU. Pri základnej implementácii algoritmu sme pozorovali počas behu využitie CPU na úrovni 25% (príklad 1 na obrázku 40), pretože test prebiehal na procesore, ktorý disponoval 2 jadrami s hyperthreadingom.

Vzhľadom na veľkú dimenziu vstupných dát a početnosť operácie výpočet vzdialenosti, sme sa ako prvé rozhodli paralelizovať výpočet Euklidovskej vzdialenosti. Pri výpočte sme rozdelili vstupné vektory na niekoľko častí a paralelne spočítali vzdialenosti týchto častí. Nakoniec sme spojili čiastkové výsledky. Tento princíp je vo všeobecnosti známy pod pojmom *MapReduce*. Rovnaký prístup sme použili aj pri výpočte nových centier klastrov, kde sme paralelne počítali jednotlivé zložky výsledného centra. Ako môžeme vidieť na príklade 2 na obrázku 40, tento spôsob už dokázal využiť takmer celé dostupné výpočtové prostriedky CPU. Na druhej strane, tento prístup (ako bude uvedené vo výsledkoch) prináša značné režijné náklady. Správa vlákien (pomocou ktorých bola implementovaná paralelizácia), synchronizácia a ďalšie aspekty tohto prístupu neumožňujú využiť možnosti paralelizácie naplno a množstvo výpočtového výkonu je spotrebovaného na réžiu operačného systému. Existujú prístupy, ktoré túto réžiu dokážu čiastočne zredukovať využitím tzv. thread workers pool pri ktorej odpadá určitá réžia so správou prostriedkov. Tento prístup síce využil dostupné výpočtové prostriedky na maximum, ale

benefit, ktorý sme vďaka nemu získali, bol menší ako pridaná réžia na strane operačného systému.

Obrázok 40: Využitie CPU pri jednotlivých metódach paralelizácie

Na základe týchto poznatkov sme náš finálny návrh paralelizácie postavili na vyššej úrovni algoritmu. Prvým krokom, ktorý budeme paralelizovať, je operácia priraďovania vektorov ku klastrom. Jednotlivé priradenia sú medzi sebou nezávislé, takže túto operáciu môžeme vykonávať paralelne. Na začiatku si rozdelíme množinu vektorov, ktoré ideme priraďovať na *n* skupín (*n* rovné počtu dostupných CPU jadier). A následne paralelne vykonávame priraďovanie vektorov. Pri implementácii si treba dať pozor na to, ako je operácia priradenia implementovaná, pretože môže vyžadovať určitú mieru synchronizácie. V druhom kroku si rozdelíme klastre na *n* skupín a paralelne budeme počítať centrá pre jednotlivé klastre. Ako v predchádzajúcom prípade, jednotlivé klastre sú medzi sebou nezávislé, takže takýto prístup by nemal spôsobovať žiadny problém.

Ako môžeme vidieť na príklade 3 z obrázku 40, tento prístup nedosahuje 100% využitie dostupných výpočtových prostriedkov. Tento problém je spôsobený nerovnomernou veľkosťou klastrov, takže niektoré skupiny sú dopočítané rýchlejšie ako iné. Riešením by mohlo byť inteligentnejšie delenie klastrov na skupiny alebo využitie prístupu, kde by výpočet každého klastra predstavoval samostatnú úlohu, a tieto úlohy by boli plánované na základe dostupnosti výpočtových prostriedkov postupne. Na druhej strane, naše výsledky ukazujú, že zlepšenie výkonu vďaka tomuto prístupu je minimálne, no implementačná komplexnosť značne narastie.

5.6.2 Heuristika pre zaraďovanie vektorov do klastrov

V tejto kapitole si ukážeme spôsob, pomocou ktorého môžeme zrýchliť fázu priraďovania vektorov do klastrov. Na základe analýzy behu výpočtu sme zistili, že až 68% času spotrebovaného pri zaraďovaní vektorov do klastrov trvá výpočet vzdialeností. Ďalším zaujímavým pozorovaním bolo, že až na zanedbateľný počet prvotných krokov, v 99% prípadov vektory ostávajú v klastroch, v ktorých boli v predchádzajúcom kroku. Napriek tejto skutočnosti, algoritmus musí spočítať vzdialenosť ku všetkým ostatným centrám, čo stojí veľa výpočtového času.

Na vyriešenie tohto problému sme navrhli modifikáciu, ktorá odstráni zbytočné operácie výpočtu vzdialeností.

```
1: function K_Means(V : vector list, k : int, C : list of clusters): list of clusters
2:     repeat
3:         change ← 0
4:         m ← {}
5:         for all cluster c in C do
6:             m(c) ← Sort(Distance(c, C_j)), ∀ j∈{1..k} ∧ C_j != c)
7:         end for
8:         for all vector v in V do
9:              c ← LastCluster(v)
10:             dToLast ← Distance(v, c)
11:             c ← Min(Distance(v, b), dToLast), ∀ b∈m(c)∧Distance(v, m(c))<2*dToLast
12:             Assign(v,c)
13:         end for
14:         for all cluster c in C do
15:            change ← change + Recalculate(c)
16:         end for
17:    while change > 0
18:    return C
19: end function
```

Algoritmus 6: Modifikácia K-Means algoritmu redukujúca počet výpočtov vzdialeností

Na začiatku každého kroku modifikovaný algoritmus spočíta maticu vzdialeností medzi jednotlivými klastrami, a každému klastru priradí zoznam dvojíc – vzdialenosť, druhý klaster, zoradený podľa vzdialenosti vzostupne. Vďaka tomuto vieme pre každý klaster povedať, ako vzdialené sú od neho ostatné klastre. Tento zoznam využijeme pri priraďovaní vektorov. Pre každý vektor najskôr spočítame vzdialenosť ku pôvodnému klastru (v algoritme označená ako `dToLast`), a následne postupne prechádzame zoznamom najbližších klastrov klastra, v ktorom bol vektor pôvodne, dovtedy, kým vzdialenosť medzi klastrami nie je väčšia ako `2*dToLast`. Počas tejto iterácie kontrolujeme vzdialenosť vektora ku skúmanému klastru, a v prípade potreby aktualizujeme najmenšiu nájdenú vzdialenosť.

Matematicky môžeme ukázať, koľko operácií výpočtu vzdialeností nám tento spôsob ušetrí. Označme počet klastrov ako k a počet vektorov ako n. V základnej implementácii algoritmu je potrebných presne $n * k$ výpočtov vzdialeností medzi vektormi a klastrom. V nami navrhovanej implementácii je potrebných $\frac{k*k-1}{2}$ výpočtov na spočítanie matice vzdialeností medzi klastrami. Ďalších n výpočtov je potrebných na výpočet vzdialeností k centrám pôvodných klastrov (možno znížiť zapamätaním si pôvodnej vzdialenosti a jej využitím v prípade, že sa centrum klastra nezmenilo). Poslednou časťou, ktorú je potrebné započítať, je priemerný počet klastrov (označíme x), ktoré je nutné preskúmať v okolí pôvodného klastra. Na základe meraní v prípade datasetu 1 bola táto hodnota rovná 3,2.

Celkový počet operácií výpočtu vzdialeností je tým pádom $\frac{k*k-1}{2} + (x+1) * n$. Je zrejmé, že tento počet bude nižší než pôvodných $n * k$.

5.6.3 Efektívnosť implementácie

Okrem optimalizácii na vyššej úrovni sme preskúmali aj optimalizácie konkrétnych implementačných detailov daného algoritmu v závislosti od programovacieho jazyka a fyzického hardvéru. Náš algoritmus bol implementovaný v .net frameworku v programovacom jazyku C#. Tento jazyk je vysokoúrovňový programovací jazyk a neumožňuje priamo ovplyvňovať operácie, ktoré bude fyzicky hardvér vykonávať. Taktiež nemôžeme priamo povedať kompilátoru, aby využíval napríklad vektorové inštrukcie.

Prvotná implementácia algoritmu v značnej miere využívala hlavne rôzne *LINQ*[7] rozšírenia pre pohodlnosť. Ako príklad môžeme uviesť metódu `zip`[8], pomocou ktorej sme počítali Euklidovskú vzdialenosť. Túto metódu sme nahradili obyčajným cyklom, čo mierne zrýchlilo celý algoritmus. Podobný prístup sme používali aj na výpočet centier, a aj v tomto prípade náhrada výrazu za obyčajný cyklus mierne zrýchlila algoritmus. Tieto zmeny priniesli zrýchlenie rádovo len v jednotkách percent.

Pri analýze sa ukázala ďalšia možnosť na zrýchlenie algoritmu na výpočet centier. Pôvodný algoritmus na výpočet centier (algoritmus 7) počítal každú zložku vektora samostatne. Možno to na prvý pohľad nepredstavuje problém, ale keď si uvedomíme, ako sú jednotlivé vektory uložene v pamäti, takýto prístup znamená, že algoritmus musí pri výpočte každej zložky nahrávať z operačnej pamäte do procesora veľkú časť dát.

```
1: mean ← [sizeof(vector)]
2: for i in 1..sizeof(vector) do
3:     mean(i) ← average(v(i))∀ v ∈ V
4: end for
```

Algoritmus 7: Algoritmus výpočtu nových centier klastra

Okrem toho, súčasné procesory masívne využívajú cache a rôzne techniky, pomocou ktorých sa snažia dopredu si pripravovať dáta, ktoré by mohli v budúcnosti potrebovať (CPU prefeatch). Vzhľadom na to, že vektory sú uložené na náhodných miestach v pamäti, CPU nedokáže predpovedať, ktoré dáta bude potrebovať, a tým pádom spracovanie každého ďalšieho vektora znamená čakanie na načítanie hodnoty z operačnej pamäte.

[7] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/
[8] https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.zip

Pre zlepšenie týchto vlastností sme modifikovali algoritmus tak, aby počítal priemer všetkých zložiek vektora naraz. Tento prístup má síce rovnaký počet operácií ako predchádzajúci, ale výsledky ukazujú, že je rádovo rýchlejší.

```
1: mean ← [sizeof(vector)]
2: for all vector v in V do
3:     for i in 1..sizeof(vector) do
4:         mean(i) ← mean(i) + v(i)
5:     end for
6: end for
7: mean(i) ← mean(i) / count(V) ∀i ∈ {1..sizeof(vector)]}
```

Algoritmus 8: Efektívnejší algoritmus na výpočet centier klastrov

Pokiaľ budeme analyzovať, prečo je táto verzia algoritmu efektívnejšia, tak zistíme, že táto verzia efektívnejšie využíva CPU cache a spomínaný prefetch. Prístup ku cache CPU je rádovo rýchlejší ako prístup k dátam, ktoré sa nachádzajú v operačnej pamäti. Ďalším dôvodom je aj to, že CPU načítava dáta z pamäte do svojej cache pamäte v blokoch, takže keď sa snažíme pristupovať k nejakej zložke vektora, existuje dosť veľká šanca, že táto zložka už v cache bude. Navyše takýto sekvenčný prístup (postupne prechádzame všetkými zložkami vektora) uľahčuje aj predikciu CPU prefetch, ktorý dopredu pripravuje dáta, ktoré budú potrebné v budúcnosti.

Ako sme videli, správna implementácia algoritmov a využitie CPU cache môžu často pomôcť pri zrýchlení algoritmov bez nutnosti ich modifikácie.

5.6.4 Redukcia rozmeru dát

Doteraz sme popisovali metódy optimalizácií, ktoré sa sústredili na zlepšenie algoritmov alebo ich implementácie. Žiadna z týchto metód nemala žiaden vplyv na výstup algoritmu. V tejto kapitole si popíšeme trochu iný prístup k zefektívneniu.

V podkapitole 5.5.2 sme sa zaoberali výberom prvkov vhodných na indexáciu. Ukázalo sa, že naše vektory obsahujú množstvo závislých zložiek, ktorých odstránením by sme dosiahli zefektívnenie algoritmu. Keďže táto úloha je komplexnejšia, nebudeme sa ňou priamo zaoberať v tejto časti. Namiesto toho ukážeme, že ak aj zredukujeme množstvo dát, ktoré nám budú vstupovať do klasteringu, tak hlavnému dôvodu, pre ktorý klastering využívame, to neuškodí.

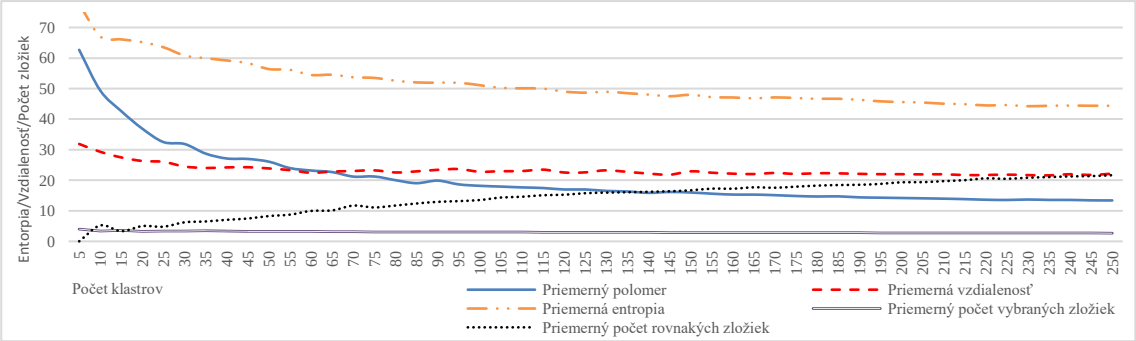
Prvou možnosťou, ktorej sa budeme venovať, je odstránenie pridlhých vektorov. Dĺžku vektora sme definovali v kapitole 5.4.3 a vzhľadom na to, že náš algoritmus nelimituje maximálnu dĺžku vygenerovaného vektora, tak sa často stáva, že vznikajú vektory, ktoré pokrývajú celý zdrojový kód. Takéto vektory nemá zmysel porovnávať,

pretože vždy budú príliš špecifické, a efektívnejšie bude porovnávať menšie časti, a následne tieto menšie časti spájať do väčších zhôd. Je dôležité dodať, že tým, že odstránime niektoré veľké vektory, nestratíme takmer žiadnu informáciu o zdrojovom kóde, nakoľko takéto vektory vždy vznikli skladaním menších, a tieto menšie vektory nám v dostatočnej miere pokryjú zdrojový kód. Hlavnou otázkou zostáva, ako určiť maximálnu dĺžku vektora. Empiricky sme na základe experimentov túto hodnotu stanovili na 250. Pri tomto počte stále dokážu menšie vektory dostatočne dobre pokryť zdrojový kód.

Druhou možnosťou, ako zefektívniť klastering, je zníženie dimenzie vektorov. Pri redukcii počtu zložiek sme zvažovali dva prístupy. V prvom prístupe sme sa rozhodli odstrániť zložky, ktoré majú nízku entropiu, nakoľko nám pri vyhľadávaní vektorov aj tak nepomôžu. A v druhom prístupe sme vyberali zložky na základe ich frekvencie, t.j. zložky, ktoré sú vo väčšine vektorov nulové.

Z navrhnutých prístupov sme na základe experimentov zvolili druhý popisovaný. Na základe datasetu 1 sme vybrali 80 zložiek, ktoré sa vyskytovali v menej ako 0.1% vektorov.

V tomto navrhovanom prístupe je ale dôležité si uvedomiť, že odstránenie spomenutých 80 zložiek sa aplikuje len na fázu klastrovania. V ďalších algoritmoch (hlavne pri vyhľadávaní zhôd) budeme pracovať s pôvodnými vektormi.



Obrázok 41: Vybrané charakteristiky klastrov pri zjednodušených vektoroch

Ako môžeme na obrázku 41 vidieť, jednotlivé charakteristiky majú podobný priebeh ako pri klasteringu, kde boli použité všetky dáta v plnom tvare (obrázok 31). Jediný rozdiel je v absolútnych hodnotách, kde priemerná entropia je nižšia o 25-30%. Priemerná minimálna vzdialenosť je zhruba o 50% nižšia, čomu napomohlo práve odstránenie „vytŕčajúcich“ zložiek vektora. Ostatné charakteristiky taktiež poklesli, ale ich pokles nie je až taký výrazný.

Na základe týchto pozorovaní môžeme povedať, že vytvorené klastre sú menšie, kompaktnejšie. Vplyv tejto redukcie na efektívnosť vyhľadávania plagiátov sme neposudzovali, pretože tá nie je od spôsobu klasterizácie kriticky závislá. Na druhej strane môžeme povedať, že zredukovanie množstva dát zrýchli proces klasteringu.

5.6.5 Zhodnotenie

V tejto kapitole zhodnotíme vplyv jednotlivých optimalizácií na výslednú rýchlosť K-Means algoritmu. Všetky uvedené časy budú vychádzať z priemeru 5 meraní s využitím dát z datasetu 1. Inicializácia klastrov bola pre porovnanie v každom z meraných prípadov rovnaká. Inicializáciu a prvotné priradenie vektorov do klastrov sme nemerali. Pri meraní sme merali **priemernú dĺžku jedného cyklu** algoritmu, t.j. priradenie vektorov do klastrov a prepočet centier. Namerané časy boli merané s počtom klastrov rovným 100. Každé ďalšie meranie zahŕňa aj použitie techník z predchádzajúcej fázy. Konfigurácia systému, na ktorom boli merania uskutočnené, bola nasledovná: Intel Core I5 6200U CPU, 16GB RAM a SSD disk.

	Vkladanie vektorov	Výpočet centier	Prepočty vzdialenoti	Spolu
Pôvodná verzia	153,6	1,7	-	155,3
Paralelizácia	49,9	1,1	-	51
Heuristika na priraďovanie vektorov	16,3	1,1	0,04	17,4
Optimalizacia implementácie	5,1	0,1	0,02	5,2

Tabuľka 5: Porovnanie výpočtovej rýchlosti K-Means algoritmu (v sekundách)

5.7 Škálovateľnosť

Na škálovateľnosť sa myslelo už pri návrhu metódy. Dobrá škálovateľnosť bude dôležitá hlavne do budúcnosti, pretože v súčasnosti nemáme k dispozícii také množstvo dát, ktoré by sme nevedeli spracovať na jednom stroji. Cieľom tejto časti bude navrhnúť spôsob rozdelenia dát a algoritmov na viacero výpočtových uzlov. Pri škálovateľnosti sa zameriame hlavne na škálovateľnosť smerom nahor, pretože dáta budú do systému vždy len pribúdať. Konkrétnu implementáciu v tejto kapitole popisovať nebudeme, namiesto toho sa zameriame na teoretické poznatky, nutné k takejto implementácii.

Celá škálovateľnosť metódy bude založená na klasteringu. Jednotlivé klastre budú slúžiť ako jednotky, na základe ktorých budeme riešenie škálovať. Overeniu základnej myšlienky škálovateľnosti sme sa v podstate venovali v kapitole o paralelizácii algoritmu. Paralelizácia je taktiež istá forma škálovania, len s tým, že dáta sú jednoducho zdieľatelné

medzi všetkými výpočtovými uzlami. Tento koncept sa dá rozviesť ďalej a začať deliť celú metódu na základe klastrov. Môžeme začať s jedným uzlom a v prípade potreby pridávať ďalšie uzly a premiestňovať klastre medzi uzlami. V kapitole o paralelizácii sme ukázali, že aj dáta sú v značnej miere lokalizované (pri priraďovaní vektorov do klastrov v 99% prípadov vektor ostane v pôvodnom klastri, a výpočet centier závisí len od dát príslušného klastra). Jedinými zdieľanými dátami naprieč uzlami budú polohy centier klastrov.

Je zrejmé, že maximálny počet uzlov, na ktorý dokážeme takéto riešenie rozdeliť, je rovný počtu klastrov. Na druhej strane sme tiež ukázali, že optimálny počet klastrov s narastajúcim množstvom dát pomaly rastie, a pridávanie ďalších klastrov neznižuje efektivitu algoritmu. Na základe týchto princípov dokážeme riešenie škálovať v prípade potreby.

5.8 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.

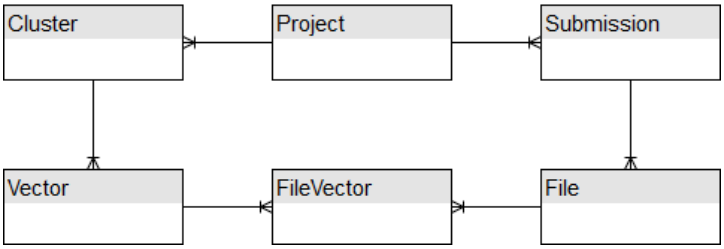
Požiadavky na štruktúru pre uloženie dát sú nasledovné:

- 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.

Existujú rôzne možnosti ukladania takého typu multidimenzionálnych dát [69][70]. 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. Zjednodušene náš navrhnutý dátový model vyzerá nasledovne:



Obrázok 42: Zjednodušený dátový model pre ukladanie vektorov

Základnou entitou v našom systéme je entita `Vector`. Táto entita reprezentuje jeden unikátny vektor, ktorý je súčasťou projektu. `Vector` obsahuje informácie o klastri, do ktorého patrí, hodnoty jednotlivých zložiek vektora a hodnoty vybraných 5 významných zložiek. V databáze má táto entita vytvorený zložený index nad atribútom odkazujúcim klaster a atribútmi, ktoré reprezentujú zvolených 5 významných prvkov vektora. Tento index slúži na vyhľadávanie vektorov v databáze tak, ako sme to navrhli v kapitole 5.5. Entita `Cluster` slúži na uchovanie informácií o konkrétnom klastri: centrum a zoznam piatich významných zložiek vektora v rámci daného klastra. Poslednou, pre účely klasteringu dôležitou entitou, je entita `FileVector`. Táto entita reprezentuje konkrétny vektor, ktorý pochádza z určitého súboru. Entita nemá hodnoty zložiek vektora uložené priamo, ale obsahuje odkaz na entitu `Vector`, ktorá tieto hodnoty obsahuje. Entity `File` a `Submission` obsahujú dodatočné informácie k vektorom, potrebné pre algoritmus vyhľadávania plagiátov.

Keďže navrhnutá štruktúra má byť univerzálna, pridali sme entitu `Project`, ktorá rozdeľuje jednotlivé vektory. Pod pojmom projekt v tejto fáze budeme rozumieť vektory, ktoré vznikli rovnakým algoritmom s rovnakými parametrami pri spracovaní zdrojového kódu. To je veľmi dôležité, pretože metóda, tak ako je navrhnutá, neumožňuje priame porovnávanie zdrojového kódu naprogramovaného v rôznych programovacích jazykoch.

Takto navrhnutá štruktúra 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.

5.9 Zhodnotenie

V kapitole 5 sme sa venovali možnostiam klasterizácie a perzistencie zdrojového kódu. Navrhli sme metódu, pomocou ktorej môžeme klastrovať zdrojový kód reprezentovaný charakteristickými vektormi. Táto metóda má dva základné komponenty. Prvým z nich je inkrementálny klastering, ktorý sa snaží udržiavať vektory optimálne rozdelené na niekoľko skupín, a pomáha určiť spôsob vyhľadávania vektorov. Druhý je databáza, ktorá uchováva vektory v organizácii, ktorú určil klastering.

Pri implementácii systému bude najdôležitejším komponentom, ktorý bude musieť neustále fungovať, databáza. Na úvod síce bude potrebný určitý výpočtový výkon na výpočet klastrov z nejakých počiatočných dát, ale následne môže byť tento uvoľnený. Počas samotného behu systému bude hlavným komponentom databáza, do ktorej budeme môcť vkladať vektory pri danej konfigurácii klastrov a vyhľadávať zhody.

Takýto návrh systému spolu s faktom, že pri narastajúcom množstve dát bude klesať potreba reklasterizácie, umožní systému efektívne využívať výpočtové prostriedky. Ako už bolo povedané, pri bežnom fungovaní systému bude nutný len beh databázy, a v prípade, keď bude nutná reklasterizácia sa dočasne pridá výpočtový výkon.

Navrhnutý systém považujeme za dostatočne robustný. V prípade narastajúceho počtu je možné oba komponenty škálovať (databázové systémy to umožňujú bežne a v kapitole 5.7 sme si popísali aj možnosti škálovania K-Means algoritmu).

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. Navrhneme niekoľko riešení, ktoré podľa nášho názoru dokážu tieto problémy z časti potlačiť. Na záver overíme navrhnuté algoritmy a porovnáme ich s doteraz používanými systémami.

6.1 Algoritmus

Základom nášho algoritmu na vyhľadávanie plagiátov, je dátová štruktúra navrhnutá v kapitole 5.8. 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. Jednotlivé časti budú rozobraté do detailov v nasledujúcich kapitolách.

6.1.1 Vyhľadávanie zhodných vektorov

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.

V prvom kroku algoritmus vyhľadá pre každý vektor z kontrolovanej úlohy rovnaké vektory v iných úlohách, ktoré sa nachádzajú v databáze. Týmto spôsobom vznikne zoznam dvojíc $< vf_1, vf_2 >$ kde vf_1 je vektor, ktorý pochádza z kontrolovanej úlohy a vf_2 je k nemu nájdený zhodný vektor, pochádzajúci z inej úlohy. Tieto dva vektory majú rovnaké zložky (lebo také sme hľadali), ale líšia sa v súbore, z ktorého pochádzajú a pozície v ňom. Algoritmus, ktorý vygeneruje tento zoznam dvojíc, je na základe navrhnutej štruktúry databázy jednoduchý.

Po získaní zoznamu dvojíc nasleduje jeho transformácia, pri ktorej rozdelíme tento zoznam na základe úlohy, z ktorej vektor vf_2 pochádzal. Týmto spôsobom dostaneme zoznam prác, ktoré môžu byť s kontrolovanou prácou podobné, spolu zo zoznamom jednotlivých podobností.

6.1.2 Spájanie nájdených zhôd

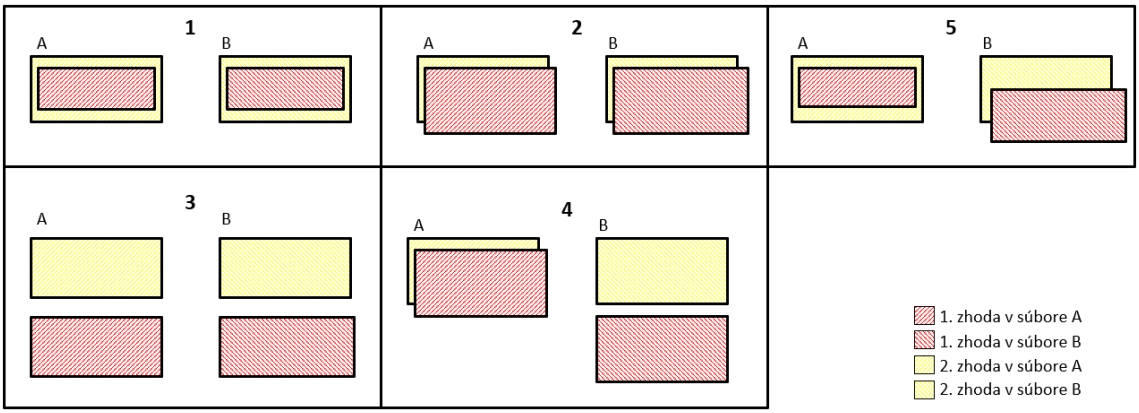
Pre lepšie spracovanie a následnú vizualizáciu je nutné nájdené zhody medzi dvoma úlohami dodatočne spracovať. Na to, aby sme mohli určiť, nakoľko sa jednotlivé zadania podobajú, potrebujeme získať disjunktné časti zdrojového kódu, v ktorých bola nájdená zhoda. Keďže vektory boli generované tak, aby 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. Algoritmus spájania vektorov je nasledovný:

```
1: function MERGE_MATCHES(M : listOfMatches): matchesInFiles
2:     matchesByFile ← DivideByFiles(M);
3:     for all fileMatches F in matchesByFile do
4:         while ¬ContainsOnlyDistintMatches(F) do
5:             for all match X in F do
6:                 if CanMerge(X, F) then
7:                     MergeMatchInto(X, F)
8:                 end if
9:             end for
10:        end while
11:    end
12:    return matchesByFile
13: end function
```

Algoritmus 9: Spájanie nájdených zhôd

Pokiaľ sa pokúšame spájať jednotlivé zhody, môžeme naraziť na niekoľko situácií. Niektoré z nich sa dajú jednoducho spojiť, iné môžu spôsobiť problémy. Celkovo sa nám

podarilo identifikovať 5 základných situácií, ktoré môžu pri spájaní vektorov nastať. Jednotlivé situácie sú zobrazené na obrázku 43.



Obrázok 43: Ukážka situácií ktoré môžu nastať pri spájaní vektorov

Rozoberme si jednotlivé prípady zobrazené na obrázku 43 podrobnejšie.

1. V oboch súboroch jedna zhoda kompletne prekrýva druhú – v tomto prípade vektory spojíme (jednoducho odstránime menšiu z nich).
2. Zhody sa prekrývajú v jednotlivých súboroch podobne* - v tomto prípade opäť zhody spojíme.
3. Zhody v jednotlivých súboroch nasledujú jedna za druhou** - zhody spájame.
4. Zhody sa prekrývajú len v jednom súbore – spájanie nevykonáme.
5. Zhody sa prekrývajú v súboroch v rôznych množstvách*** - spájanie nevykonáme.

* Pod pojmom podobne myslíme, že veľkosť prekryvu týchto zhôd je rovnaká v oboch súboroch.

** Pri zhodách, ktoré nasledujú jedna za druhou, musíme vziať do úvahy skutočné nasledovanie. Pretože jednotlivé vektory vznikajú z konkrétnych logických častí a blokov, môže sa stať, že tieto bloky sú oddelené len nevýznamnými znakmi (medzery, prázdne riadky, komentáre). Pri spájaní zhôd budeme spájať zhody, ktoré nie sú tesne vedľa seba, pokiaľ sa medzi nimi nachádzajú len tieto nevýznamné znaky.

*** Pokiaľ v zdrojovom kóde narazíme na takýto prípad, tak to vo väčšine prípadov poukazuje na duplicitu v kóde. Na základe pozorovania sme sa rozhodli takéto zhody nespájať, pretože postupnou iteráciu algoritmu na spájanie zhôd sa zvyčajne ukáže, že tieto

zhody sú súčasťou väčších zhodných celkov. Pri postupnom spájaní týchto väčších zhodných celkov sa z takýchto dvojíc stane prípad, ako bol popísaný v situácii 1.

6.1.3 Výpočet zhody pre dvojicu prác

Poslednou časťou celého algoritmu je výpočet miery podobností pre konkrétne dvojice prác. Algoritmus spájania zhôd dokáže pospájať zhody do väčších celkov vhodné na reprezentáciu. Pri výpočte miery podobností to ale nestačí (pretože vznikajú zhody 4. a 5. druhu).

Podobnosť úloh A a B je vyjadrená pomocou vzorca:

$$\boldsymbol{Sim(A,B)} = \frac{\sum_{\boldsymbol{m} \in \boldsymbol{matches(A,B)}} \boldsymbol{coverage(m_A)}}{\boldsymbol{coverage(A)}} \quad (10)$$

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.

Pri výpočte tohto pokrytia musíme pracovať so zhodami, ktoré sa neprekrývajú. Jednoduchým algoritmom môžeme nájsť všetky disjunktné množiny zdrojového kódu, ktoré pokrývajú nájdené zhody. V tomto prípade už nie je nutné komplikované spájanie zhôd, ale jednoducho pospájame zhody, ktoré sa prekrývajú v rámci jedného súboru.

6.2 Problémy pri vyhodnocovaní plagiátorstva

V tejto kapitole sa budeme bližšie venovať možnostiam odstránenia niektorých problémov, ktoré vznikajú pri vyhľadávaní plagiátov. V kapitole 1.1.1 sme popísali problémy, na ktoré sme narazili pri používaní bežne dostupných nástrojov. Niektoré z problémov týchto nástrojov sa pokúsime odstrániť.

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.

6.2.1 Odstraňovanie nevýznamných častí zdrojového kódu

V tejto kapitole si popíšeme metódu na odstránenie nevýznamných častí zdrojového kódu, ktorú sme sa rozhodli začleniť do nášho algoritmu. 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.

6.2.1.1 Manuálna anotácia

Pre potreby manuálnej anotácie sme do dátového modelu pridali ku každému vektoru skóre. Toto skóre určuje významnosť daného vektora v súvislosti s vyhodnocovaním plagiátorstva. V súčasnosti navrhnutý algoritmus berie do úvahy len dve hodnoty tohto skóre: 1 – vektor je zaradený do porovnávania a 0 – vektor do porovnávania zaradený nie je. Overenie tohto prístupu sme vykonali na datasete pozostávajúcom zo semestrálnych prác z predmetov *Informatika 1* a *2*. Tento dataset bude bližšie popísaný v časti 6.3.2.

Pri manuálnej anotácii sme vybrali niekoľko semestrálnych prác a ručne označovali časti kódu, ktoré považujeme za nevýznamné. Časti, ktoré boli označené, sú:

- import príkazy
- automaticky generovaný kód generovaný Netbeans GUI dizajnérom
- automaticky generované „`main`“ metódy

Počas tohto procesu bolo nutné si uvedomiť, že neanotujeme časti zdrojového kódu, ale vektory, ktoré tento zdrojový kód pokrývajú. Každá časť zdrojového kódu môže byť pokrytá 0 až *n* rôznymi vektormi. Pri anotovaní bolo potrebné si uvedomiť, či daný vektor pokrýva len nevýznamnú časť zdrojového kódu alebo nie.

```
1   /*
2    * To change this license header, choose License Headers in Project Properties.
3    * To change this template file, choose Tools | Templates
4    * and open the template in the editor.
5    */
6   package semestralnaPraca.hra.subor;
7
8   import java.io.FileInputStream;
9   import java.io.FileNotFoundException;
10  import java.io.FileOutputStream;
11  import java.io.IOException;
12  import java.io.ObjectInputStream;
13  import java.io.ObjectOutputStream;
14  import semestralnaPraca.hra.kasuba.LogikaHry;
15  import semestralnaPraca.hra.tvary.Bomber;
16  import semestralnaPraca.hra.tvary.Nepriatel;
17  import semestralnaPraca.hra.tvary.Tehla;
18
19  /**
20   *
21   * @author kasub
22   */
23  public class PracaSoSuborom {
24
25      private static LogikaHry object = LogikaHry.getINSTANCIA();
26
```

Obrázok 44: Označovanie nevýznamných import výrazov

Na obrázku 44 vidíme príklad označenej časti zdrojového kódu. Šedou zvýraznená časť obsahuje iba vektory, ktoré sú nevýznamné, a červená časť označuje časti kódu, ktoré sú pokryté vektormi. Môžeme si všimnúť, že komentár na riadkoch 1 až 5 pokrytý vektormi nie je.

```
79          pridajButton.setText("Pridaj");
80          pridajButton.addActionListener(new java.awt.event.ActionListener() {
81              public void actionPerformed(java.awt.event.ActionEvent evt) {
82                  pridajButtonActionPerformed(evt);
83              }
84          });
85
86          opravButton.setText("Oprav");
87          opravButton.setEnabled(false);
88          opravButton.addActionListener(new java.awt.event.ActionListener() {
89              public void actionPerformed(java.awt.event.ActionEvent evt) {
90                  opravButtonActionPerformed(evt);
91              }
92          });
93
94          vymazButton.setText("Vymaž");
95          vymazButton.setEnabled(false);
96          vymazButton.addActionListener(new java.awt.event.ActionListener() {
97              public void actionPerformed(java.awt.event.ActionEvent evt) {
98                  vymazButtonActionPerformed(evt);
99              }
100         });
```

Obrázok 45: Príklad označenia automaticky generovaného kódu

Ďalším celkom pekným príkladom je označenie automaticky generovaného kódu. Ako môžeme vidieť na obrázku 45, podarilo sa nám označiť časti kódu, ktoré generuje program Netbeans pri návrhu GUI aplikácií. Takýto kód môžeme nájsť takmer v každej GUI aplikácii, ktorá bola navrhnutá v programe Netbeans.

Tieto ukážky mali slúžiť ako príklad. Takýmto spôsobom môžeme označovať rôzne časti zdrojového kódu. Celkovo sme týmto spôsobom označili 263 unikátnych vektorov, ktoré pokrývali zhruba 8% celého zdrojového kódu.

6.2.1.2 Poloautomatické odstraňovanie s využitím klasteringu

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. Jednotlivé podiely v našom konkrétnom prípade môžeme vidieť v tabuľke 6, ktorá zobrazuje klastre, v ktorých podiel označených vektorov presiahol 1%. Jednotlivé klastre sme manuálne skontrolovali a overili, či skutočne obsahujú len nevýznamné vektory.

Klaster	Počet vektorov	Unikátnych vektorov	Označených		Správnosť označenia
56	2303	319	33	10%	Áno
51	9053	333	16	5%	Áno
26	22978	3024	56	2%	Áno
58	3866	828	15	2%	Áno
92	4412	2192	26	1%	Nie
27	3032	915	10	1%	Áno
73	4417	2050	19	1%	Nie

Tabuľka 6: Vyhodnotenie nevýznamného kódu v klastroch

Ukázalo sa, že väčšina označených klastrov skutočne obsahovala len označené nevýznamné vektory. Odstránením týchto piatich klastrov sme odstránili okolo 15% zdrojového kódu, ktorý bol pokrytý týmito klastrami.

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.

6.2.2 Filtrovanie automaticky generovaného kódu

Na základe skúseností so spracovaním zdrojového kódu v programovacích jazykoch C# a Java sme dospeli k záveru, že pokiaľ chceme odstrániť automaticky generovaný kód v programoch napísaných v programovacom jazyku Java, musíme použiť techniky popísané v prechádzajúcej kapitole. V prípade C# je situácia jednoduchšia. Spomínané importy na začiatku súborov sa tu vyskytujú síce tiež, ale MS Visual Studio[9] pri vývoji ukladá automaticky generovaný kód mimo zdrojový kód programátora. Vďaka tomuto

[9] Vývojové prostredie používané na vývoj aplikácií v jazyku C#

faktu je možné zaviesť určité pravidlá pre filtrovanie zdrojových súborov na základe názvu.

6.2.3 Normalizácia vstupných dát

Myšlienka normalizácie vstupných dát vychádza z potreby odhaľovať bežné pokusy o maskovanie plagiátov. Výskumy ukazujú, že použitie techník normalizácie na úrovni syntaktických stromov dokáže zlepšiť detekčné schopnosti algoritmov [33][34].

Základné techniky a pokusy o zamaskovanie skopírovanej úlohy (medzi ktoré patria: zmena názvu identifikátorov, zmena odsadenia, medzier, pridávanie riadkov alebo komentárov) dokážeme odhaliť bez nutnosti nejakým spôsobom normalizovať zdrojový kód. Všetky spomenuté techniky sa snažia zamaskovať vlastnosti zdrojového kódu, ktoré náš algoritmus neberie do úvahy.

Medzi zložitejšie techniky (ktoré ale už vyžadujú aj určitú znalosť študenta) môžeme zaradiť:

- premiestňovanie kódu medzi jednotlivými metódami,
- zmena štruktúry kódu,
- zmena parametrov metód,
- úprava postupnosti príkazov,
- ...

Väčšina z týchto techník určitým spôsobom modifikuje štruktúru zdrojového kódu, a pri modifikácii štruktúry náš algoritmus nedokáže vždy dostatočne dobre vyhľadávať plagiáty. Vzhľadom na spôsob, akým sa generujú vektory, náš algoritmus dokáže nájsť aj podobnosť pri istej zmene štruktúry. Táto zmena musí byť lokálna (napr. prehodenie dvoch výrazov), kde síce na úrovni jednotlivých riadkov nastala zmena, ale napríklad metóda má z vonkajšieho pohľadu rovnakú štruktúru.

V súvislosti s týmto problémom sme analyzovali prístup, popísaný v práci [33]. Autori vyžívajú rôzne techniky na normalizáciu podmienok. Popisujú situácie, kedy sú zamenené poriadia jednotlivých vetiev. Venujú sa možnostiam obracania výrazov v samotnej podmienke (napríklad výraz `a<=5` má rovnakú logickú hodnotu ako výraz `6>a`). Náš algoritmus väčšinu situácií, ktoré popisuje spomínaný článok, dokáže bez nutnosti ďalšej modifikácie detegovať. Rozhodli sme sa preto neimplementovať dodatočnú normalizáciu

s týmto cieľom. Bližšiu ukážku testovaných modifikácií, ktoré algoritmus dokázal detegovať, je možné nájsť v prílohe.

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 budeme posudzovať správnosť výsledkov manuálnou kontrolou, ale aj porovnaním s existujúcimi systémami. Okrem prác študentov sme algoritmus overovali aj pomocou pripravených fragmentov kódu s cieľom overiť jeho správanie v rôznych situáciách. Zoznam a ukážky týchto situácií je možné nájsť v prílohe.

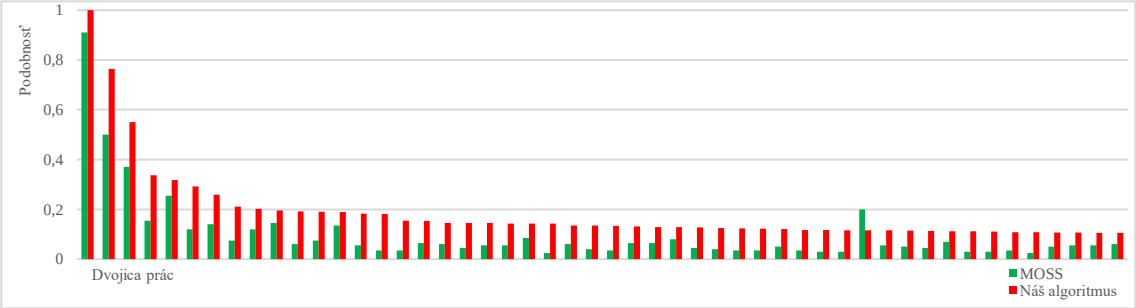
6.3.1 Porovnanie so systémom MOSS

Pre účely vyhodnotenia nášho algoritmu sme si pripravili dataset prác v naprogramovaných v programovacom jazyku C#. Vychádzali sme pri tom z už používaného datasetu 1, kde sme pridali jednu prácu 2x, aby tak vznikla 100% zhoda, ktorá nám poslúži ako referenčný ukazovateľ. Okrem toho dataset obsahoval niekoľko plagiátov, ktoré boli vytvorené priamo študentami.

Prvým krokom pri analýze výsledkov nášho algoritmu bola manuálna kontrola nájdených zhôd. Vzhľadom na relatívne veľkú početnosť nájdených zhôd sme nevyhodnocovali všetky zhody, ale zamerali sme len na niektoré zaujímavé. Najskôr sme sa pozreli na náš vyrobený plagiát. Ten algoritmus detegoval ako 100% zhodu. Ostatné plagiáty, o ktorých sme vedeli dosiahli taktiež pomerne vysoké skóre zhody (55% - 75%). Na základe manuálnej kontroly týchto zadaní môžeme potvrdiť, že pridelené percento zodpovedá množstvu zhodného kódu v týchto prácach.

Percento zhody medzi ostatnými zadaniami bolo relatívne nízke (do 30%). Zo skúseností z iných nástrojov musíme povedať, že takéto percento zhody určite nepoukazuje na plagiát.

Naše dosiahnuté výsledky sme porovnali s výsledkami zo systému MOSS. Problémom pri využití systému MOSS, na ktorý sme narazili, bolo obmedzenie v maximálnom množstve prác, ktoré systém dokáže spracovať. Jednotlivé zadania sme museli porovnávať po skupinách, a následne jednotlivé výsledky zlúčiť. Ďalším problémom bolo, že výstup, ktorý generuje systém MOSS, nám neumožňuje porovnávať konkrétne zhody. Pri porovnávaní sme sa museli zamerať len na porovnanie miery podobnosti, ktoré daným zadaniam príslušné systémy určili.



Obrázok 46: Porovnanie podobností pre vybraných 50 úloh

Na obrázku 46 môžeme vidieť porovnanie nájdených zhôd pre 50 najviac podobných dvojíc úloh. Dvojice boli vyberané na základe podobnosti, ktorú im určil náš systém. Na grafe môžeme vidieť, že vo všeobecnosti náš algoritmus priznáva vyššiu mieru podobností ako MOSS. Systém MOSS dokonca aj našej 100% kópii priradil podobnosť len 91%. Tento rozdiel je spôsobený v rozdielnom počítaní výslednej zhody. Systém MOSS na rozdiel od nás počíta mieru podobnosti na základe počtu podobných riadkov. Presný dôvod nenájdenia 100% podoby nepoznáme. Z reportu, ktorý vygeneroval systém MOSS, bolo zrejmé, že niektoré prázdne riadky neboli označené ako zhodné, a tým pádom dvojica prác získala nižšie skóre. Na základe tohto porovnania môžeme povedať, že náš algoritmus lepšie spracováva časti súborov, ktoré neobsahujú priamo kód (voľné riadky, medzery...)

6.3.2 Porovnanie so systémom JPlag

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*. V rámci porovnávania zo systémom JPlag sme sa rozhodli využiť aj metódy na odstraňovanie nevýznamného kódu popísané v kapitole 6.2.1. Pri

spracovaní zdrojového kódu sme vygenerovali dve sady charakteristických vektorov. Keďže aj pri návrhu štruktúry vektorov sme sa inšpirovali algoritmom DECKARD, prvá sada vektorov bola vygenerovaná týmto nástrojom. Týmto spôsobom sme chceli overiť kompatibilitu nášho riešenia s generátorom, ktorý je súčasťou systému DECKARD. Druhá sada bola vygenerovaná vlastným algoritmom popísaným v kapitole 4.

Dataset 3

Tento dataset pozostáva zo semestrálnych prác z predmetov *Informatika 1* a *Informatika 2* z rokov 2016 – 2018. Celkovo dataset obsahuje **555** prác. Práce pozostávajú zo 6,889 súborov. Pomocou DECKARD algoritmu bolo vygenerovaných 425,563 vektorov z ktorých 164,356 bolo unikátnych. Pomocou nášho algoritmu sme vygenerovali 456,672 vektorov a 227,825 z nich bolo unikátnych. Parametre použité pri generovaní boli rovnaké ako v prípade datasetov 1 a 2.

Tieto zadania obsahujú niekoľko plagiátov vytvorených priamo študentami. Niektoré z nich sú takmer 100% kópia, iné sú do určitej miery modifikované. Dataset obsahoval aj práce niekoľkých študentov, ktorý sa dopustili tzv. sebaplagiátorstva, keď pri opakovaní predmetu odovzdali dva roky po sebe rovnakú semestrálnu prácu.

Pri vyhodnocovaní sme postupovali podobne, ako v predchádzajúcom prípade. Ani systém JPlag neumožňuje programové spracovanie nájdených zhôd. Opäť sme boli odkázaní len na porovnanie vyčíslených percentuálnych zhôd.

Pre porovnanie algoritmov sme vybrali 17 prác, ku ktorým našiel JPlag aspoň jednu zhodu s podobnosťou vyššou ako 50%. Pomocou nášho algoritmu sme ku zvoleným prácam vyhľadali všetky plagiáty (opäť sme si určili, že práce, ktoré budú mať podobnosť väčšiu ako 50%, označíme za možné plagiáty). Všetky nájdené zhody sme manuálne vyhodnotili, a určili, ktoré z nich považujeme skutočne za plagiáty.

		Náš algoritmus			
		DECKARD		Náš generátor	
Práca	JPlag	Základný	Annotovaný	Základný	Annotovaný
130	3/1	1/1	1/1	1/1	1/1
489	1/1	1/1	1/1	1/1	1/1
382	1/0	0/0	0/0	0/0	0/0
242	1/1	0/0	0/0	0/0	0/0
435	1/0	0/0	0/0	0/0	0/0
272	2/0	0/0	0/0	0/0	0/0
356	1/1	1/1	1/1	1/1	1/1
186	1/1	1/1	1/1	1/1	1/1
446	5/1	1/1	1/1	1/1	1/1
271	1/1	1/1	1/1	1/1	1/1
306	1/1	0/0	0/0	0/0	1/1
546	3/1	1/1	1/1	1/1	1/1
198	2/0	0/0	0/0	0/0	0/0
9	1/0	0/0	0/0	0/0	0/0
179	1/0	0/0	0/0	0/0	0/0
297	1/1	1/1	1/1	1/1	1/1

Tabuľka 7: Porovnanie zhôd pre vybraných 17 úloh

V tabuľke 7 vidíme zoznam vybraných 17-tich prác. V prvom stĺpci je identifikátor práce, v druhom stĺpci je počet možných plagiátov odhalený systémom JPlag a za lomkou je počet tých, ktoré z nich boli reálnymi plagiátmi. V ďalších štyroch stĺpcoch vidíme počet plagiátov, ktoré odhalil náš algoritmus. V prvých dvoch z nich sú výsledky na základe vektorov, vygenerovaných algoritmom DECKARD. V posledných dvoch sú výsledky na základe nášho algoritmu na vektorizáciu zdrojového kódu. V oboch prípadoch uvádzame výsledky pred, a po manuálnom označení nevýznamných časti zdrojového kódu.

V tabuľke môžeme vidieť 8 prípadov (označených na zeleno), kde náš algoritmus dokázal na rozdiel od JPlagu nájsť len relevantné zhody. V siedmich prípadoch (bez označenia) algoritmus efektívne eliminoval nesprávne systémom JPlag reportované zhody. V jednom prípade (práca 306) algoritmus nedokázal nájsť skutočné plagiáty, ktoré sa systému JPlag odhaliť podarilo V tomto prípade jedine sada, ktorá bola vygenerovaná naším algoritmom s anotovaním nevýznamných častí zdrojového kódu, odhalila plagiát. V tabuľke 8 môžeme vidieť porovnanie počtu prác, ktoré boli nepravdivo označené za plagiáty.

JPlag		26
DECKARD	Základný	9
	Annotovaný	5
Náš generátor	Základný	14
	Annotovaný	4

Tabuľka 8: Počet falošne pozitívnych prípadov

Okrem porovnania počtu nájdených zhôd porovnáme jednotlivé podobnosti vybraných dvojíc úloh. Toto porovnanie je zaznamenané v tabuľke 9. Vo väčšine prípadov náš algoritmus zaznamenal nižšiu mieru podobností. Referenčná dvojica „356-415“ (2x odovzdaná tá istá práca tým istým študentom) získala 100% pri využití oboch algoritmov. Táto tabuľka nám ukazuje, prečo náš algoritmus neodhalil plagiát (pri maximálnom dosiahnutom skóre 11,5% ho len ťažko budeme považovať za plagiát). Okrem toho môžeme vidieť, že dvojice, ktoré JPlag falošne detegoval, dosiahli v našom algoritme relatívne nízke skóre.

Práca			Náš algoritmus				
			DECKARD		Náš generátor		
A	B	JPlag	Základný	Annotovaný	Základný	Annotovaný	Je plagiát?
356	415	100	100	99,07	100	99,3	Áno
186	163	97,8	99,2	99,8	95,4	95,5	Áno
271	138	79,6	70,4	70,7	68,2	69,3	Áno
446	450	78,6	68,3	49,8	71	69,8	Áno
130	272	77,4	57,2	57,2	52,3	52,3	Áno
271	224	68,7	53,8	47,2	8,2	1,4	Nie
306	106	68,4	16,4	19,2	46,6	52,8	Áno
297	182	67,7	51,2	30,2	53,8	39,7	Áno
546	520	66	59,4	51,2	70,7	75,9	Áno
489	441	61,8	53,8	53,8	51,2	50,8	Áno
130	259	60,8	28,4	28,4	12,1	10,9	Nie
198	188	59,9	40,2	16,1	46,8	43,6	Nie
546	337	58,3	44,1	12,6	43,8	44,8	Nie
382	111	56,8	3,4	3,4	5,8	5,8	Nie
242	426	56,5	7,1	7,1	11,5	11	Áno
435	347	56,5	39,1	39,1	20,3	19	Nie
446	296	54,9	28,9	12,4	45,3	42,6	Nie
446	133	53,6	33,8	9,1	46,3	44,8	Nie
272	109	53,1	23,7	23,7	23,9	23,7	Nie
130	109	52,3	34,2	34,2	18,2	15,1	Nie
272	259	50,4	12,4	12,4	16,9	16,9	Nie

Tabuľka 9: Porovnanie podobností vybranej dvojice zadaní

Pri analýze, prečo algoritmus nedokázal detegovať spomínané plagiáty, 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 a možnosti zlepšenia riešenia

V tejto kapitole si zhodnotíme výsledky, ktoré sa nám pomocou navrhnutej metódy podarilo dosiahnuť, a popíšeme možnosti ďalšieho rozširovania tejto metódy. Pri overovaní algoritmu sme využívali viacero prístupov. Overovali sme reálne študentské práce, ale aj pre tieto účely špeciálne vytvorené fragmenty zdrojového kódu.

Pri skúmaní študentských prác sme analyzovali rôzne spôsoby podvádzania. Tieto spôsoby sme okrajovo analyzovali v kapitolách 1.1.1 a 6.2. Na základe nich sme aj vytvorili testované fragmenty.

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. Týmto problémom sme sa venovali v samostatnej kapitole 6.2. Metódy pred-prípravy alebo normalizácie zdrojového kódu, ktoré môžeme nájsť v literatúre, je možné aplikovať aj na náš algoritmus. Ich overovaním sme sa ale v rámci tejto práce nezaoberali. 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 vtedy, 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.

Naša práca je postavená na algoritmoch hľadajúcich podobnosti v štruktúre zdrojového kódu. Tento prístup je vhodný, keď vyhľadávame napríklad redundantný kód. Pri identifikácii plagiátorstva treba brať do úvahy, že pri programovaní sa často používajú naučené techniky, návrhové vzory, vývojové prostredia generujú množstvo kódu za programátora, ... Z tohto dôvodu bežne používané nástroje identifikujú veľké množstvo falošne pozitívnych zhôd. Náš algoritmus nebol výnimkou, a tiež označil niekoľko prác ako plagiát aj napriek tomu, že práca plagiátom nebola. Navrhnutý spôsob anotácie nevýznamného zdrojového kódu sa ukázal ako prínosný hlavne pri eliminácii falošne pozitívnych zhôd, ktoré vznikli automatickým generovaním kódu pomocou vývojového prostredia.

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. Počas práce sme sa snažili zefektívniť tie časti, ktoré jeho rýchlosť spomaľovali najviac. V kapitole 5.6 sme ukázali, ako len pomocou zmeny implementácie je možné niekoľkonásobne zrýchliť K-Means algoritmus. 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. Konkrétnym návrhom sme sa venovali v kapitole 6.2.3, kde popisujeme možnosti normalizácie vstupných dát. Tento prístup ale nedokáže pomôcť pri pridávaní alebo odoberaní kúskov kódu. Pre vyriešenie tohto problému bude potrebné upraviť spôsob generovania vektorov – napríklad negenerovať vektor z úplného podstromu, ale len do určitej hĺbky. Ďalším spôsobom, ako riešiť tento problém, by bolo zavedenie vyhľadávania čiastočnej podobnosti vo vektoroch. Tento prístup by ale podľa našich predbežných experimentov značne zvýšil početnosť falošne pozitívnych zhôd. Pri dôkladnej analýze to ale podľa nášho názoru je vhodná cesta.

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.

V súčasnosti riešenie poskytuje podporu len pre jeden programovací jazyk súčasne. Úlohy je samozrejme možné rozdeliť na niekoľko častí a vyhľadávať plagiáty v týchto častiach osobitne. Do budúcnosti je potrebné navrhnúť spôsob spájania výsledkov z jednotlivých častí alebo navrhnúť univerzálnejší spôsob generovania vektorov, aby mohli byť generované vektory s rovnakými prvkami z rôznych programovacích jazykov.

Záver

V tejto práci sme sa zaoberali problematikou odhaľovania plagiátov v zdrojovom kóde. Naším cieľom bolo navrhnúť a overiť metódu na odhaľovanie plagiátov v zdrojovom meradle. Za hlavný prínos práce považujeme navrhnuté metódy, ale aj postupy nutné k vyhodnocovaniu plagiátorstva v zdrojovom kóde.

V prvej kapitole sa venujeme problému plagiátorstva všeobecne. Skúmame príčiny jeho vzniku a formy plagiátorstva, ktoré sa bežne objavujú. Prvá kapitola obsahuje zhodnotenie postoja študentov Fakulty riadenia a Informatiky k téme plagiátorstva.

V druhej kapitole sme analyzovali súčasné metódy na odhaľovanie plagiátorstva. Vzhľadom na podobnosť zdrojového kódu s textovými dokumentmi sme porovnali metódy vyhľadávania podobností v texte a v zdrojovom kóde. Zistili sme, že tieto prístupy sú veľmi podobné. Táto analýza nám slúžila ako základ pre návrh novej metódy na vyhľadávanie plagiátov v zdrojovom kóde popísanej v kapitole 3. Ako základ tejto metódy sme sa rozhodli využiť reprezentáciu zdrojového kódu s využitím syntaktických stromov.

Spracovaniu zdrojového kódu a výberu vhodnej reprezentácie sme sa venovali v kapitole 4. V návrhu sme síce špecifikovali, že zdrojový kód budeme reprezentovať pomocou syntaktického stromu, ale tento, ako sa ukázalo, nebol vhodný na účely porovnávania zdrojového kódu. Navrhovali a overovali sme rôzne možnosti transformácie syntaktického stromu, a ako najvhodnejšiu sme určili transformáciu na skupinu charakteristických vektorov, pomocou ktorých vo výslednom štádiu reprezentujeme zdrojový kód.

Cieľom našej práce bolo spracovávanie veľkého množstva zdrojového kódu, preto sme sa v kapitole 5 venovali možnostiam rozdeľovania, perzistencie a vyhľadávania vektorov. Na predtriedenie vektorov využívame K-Means algoritmus, ktorý sme modifikovali tak, aby ho bolo možné používať inkrementálne. Toto inkrementálne použitie je dôležité najmä preto, lebo práce do APS pribúdajú každoročne, a systém musí dokázať efektívne vyhodnocovať tieto nové práce. Vektory ukladáme v relačnej databáze, z ktorej sme pre konkrétnu prácu schopní efektívne získať zoznam možných plagiátov.

V kapitole 6 sme sa venovali návrhu algoritmu na detekciu plagiátov. Tento algoritmus sa skladá z niekoľkých krokov, počas ktorých transformujeme nájdené zhody medzi prácami na reporty, ktoré popisujú mieru plagiátorstva pre konkrétnu prácu. Okrem

toho sa venujeme aj problémom, ktoré vznikajú pri vyhodnocovaní plagiátov. Predstavujeme navrhnuté riešenia, a nakoniec algoritmus overujeme na reálnych študentských prácach. V poslednej kapitole sme zhodnotili výsledky, poukázali na nedostatky navrhnutého algoritmu, a navrhli ďalšie možnosti, ktoré by mohli celý algoritmus zefektívniť.

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] Comas-Forgas, Rubén, and Jaume Sureda-Negre. "Academic plagiarism: Explanatory factors from students' perspective." *Journal of Academic Ethics* 8.3 (2010): 217-232.

[11] Bamford, Jan, and Katerina Sergiou. "International students and plagiarism: An analysis of the reasons for plagiarism among international foundation students." *Investigations in university teaching and learning* 2.2 (2005): 17-22.

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

[13] Fischer, Felix, et al. "Stack overflow considered harmful? The impact of copy&paste on android application security." *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017.

[14] Garabík, Radovan, et al. "Tokenizácia, lematizácia a morfologická anotácia Slovenského národného korpusu." Dostupný z WWW: http://korpus. juls. savba. sk/publications (2006).

[15] Huang, Anna. "Similarity measures for text document clustering." *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand*. 2008.

[16] Gomaa, Wael H., and Aly A. Fahmy. "A survey of text similarity approaches." *International Journal of Computer Applications* 68.13 (2013).

[17] Gondaliya, Tapan P., Hiren D. Joshi, and Hardik Joshi. "Source Code Plagiarism Detection'SCPDet': A Review." *International Journal of Computer Applications* 105.17 (2014).

[18] Carvalho, Nuno Ramos, et al. "From source code identifiers to natural language terms." *Journal of Systems and Software* 100 (2015): 117-128.

[19] Huang, Anna. "Similarity measures for text document clustering." *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand*. 2008.

[20] de Hoon, Michiel JL, et al. "Open source clustering software." *Bioinformatics* 20.9 (2004): 1453-1454.

[21] Trappey, Amy JC, et al. "A fuzzy ontological knowledge document clustering methodology." *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 39.3 (2009): 806-814.

[22] Cosma, Georgina, and Giovanni Acampora. "A Fuzzy-based Approach to Programming Language Independent Source-Code Plagiarism Detection."

[23] Blei, David M., Andrew Y. Ng, and Michael I. Jordan. "Latent dirichlet allocation." *Journal of machine Learning research* 3.Jan (2003): 993-1022.

[24] Binkley, David, et al. "Understanding LDA in source code analysis." *Proceedings of the 22Nd International Conference on Program Comprehension*. ACM, 2014.

[25] Thomas, Stephen W., et al. "Studying software evolution using topic models." *Science of Computer Programming* 80 (2014): 457-479.

[26] Noynaert, J. Evan. "Plagiarism detection software." *Midwest Instruction and Computing Symposium*. 2006.

[27] Prechelt, Lutz, Guido Malpohl, and Michael Philippsen. "Finding plagiarisms among a set of programs with JPlag." *J. UCS* 8.11 (2002): 1016.

[28] Schleimer, Saul, Daniel S. Wilkerson, and Alex Aiken. "Winnowing: local algorithms for document fingerprinting." *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003.

[29] Bahmani, Zeinab, and Najmeh Taleb. "Fingerprinting Jar Files Using Winnowing and K-grams."

[30] J. K. van Dam. "Identifying source code programming languages through natural language processing. " (2016).

[31] 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*.

[32] Chilowicz, Michel, Etienne Duris, and Gilles Roussel. "Syntax tree fingerprinting for source code similarity detection." *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*. IEEE, 2009.

[33] Tao, Guo, et al. "Improved plagiarism detection algorithm based on abstract syntax tree." *Emerging Intelligent Data and Web Technologies (EIDWT), 2013 Fourth International Conference on*. IEEE, 2013.

[34] Zhao, Jingling, et al. "An AST-based code plagiarism detection algorithm." *Broadband and Wireless Computing, Communication and Applications (BWCCA), 2015 10th International Conference on*. IEEE, 2015.

[35] Lazar, Flavius-Mihai, and Ovidiu Banias. "Clone detection algorithm based on the Abstract Syntax Tree approach." *Applied Computational Intelligence and Informatics (SACI), 2014 IEEE 9th International Symposium on*. IEEE, 2014.

[36] Muddu, Basavaraju, Allahbaksh Asadullah, and Vasudev Bhat. "CPDP: A robust technique for plagiarism detection in source code." *Software Clones (IWSC), 2013 7th International Workshop on*. IEEE, 2013.

[37] 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.

[38] Kaur, Harpreet, and Raman Maini. "Identification of Recurring Patterns of Code to Detect Structural Clones." *Advanced Computing (IACC), 2016 IEEE 6th International Conference on*. IEEE, 2016.

[39] Rakian, Shima, ESFAHANI FARAMARZ SAFI, and Hamid Rastegari. "A Persian fuzzy plagiarism detection approach." (2015): 182-190.

[40] Xu, Rui, and Donald Wunsch. "Survey of clustering algorithms." *IEEE Transactions on neural networks* 16.3 (2005): 645-678.

[41] Lee, Mu-Woong, et al. "Instant code clone search." *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010.

[42] Yang, Rui, Panos Kalnis, and Anthony KH Tung. "Similarity evaluation on tree-structured data." *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 2005.

[43] Kuhn, Adrian, Stéphane Ducasse, and Tudor Gírba. "Semantic clustering: Identifying topics in source code." *Information and Software Technology* 49.3 (2007): 230-243.

[44] Maletic, Jonathan I., and Naveen Valluri. "Automatic software clustering via latent semantic analysis." *Automated Software Engineering, 1999. 14th IEEE International Conference On.*. IEEE, 1999.

[45] Doval, Diego, Spiros Mancoridis, and Brian S. Mitchell. "Automatic clustering of software systems using a genetic algorithm." *Software Technology and Engineering Practice, 1999. STEP'99. Proceedings*. IEEE, 1999.

[46] Rousidis, Dimitris, and Christos Tjortjis. "Clustering data retrieved from Java source code to support software maintenance: A case study." *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*. IEEE, 2005.

[47] Lukins, Stacy K., Nicholas A. Kraft, and Letha H. Etzkorn. "Source code retrieval for bug localization using latent dirichlet allocation." *2008 15th Working Conference on Reverse Engineering*. IEEE, 2008.

[48] Mancoridis, Spiros, et al. "Bunch: A clustering tool for the recovery and maintenance of software system structures." *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE, 1999.

[49] Mancoridis, Spiros, et al. "Using automatic clustering to produce high-level system organizations of source code." *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*. IEEE, 1998.

[50] Moussiades, Lefteris, and Athena Vakali. "PDetect: A clustering approach for detecting plagiarism in source code datasets." *The computer journal* 48.6 (2005): 651-661.

[51] Jadalla, Ameera, and Ashraf Elnagar. "PDE4Java: Plagiarism Detection Engine for Java source code: a clustering approach." *International Journal of Business Intelligence and Data Mining* 3.2 (2008): 121-135.

[52] Cosma, Georgina, and Mike Joy. "An approach to source-code plagiarism detection and investigation using latent semantic analysis." *IEEE transactions on computers* 61.3 (2012): 379-394.

[53] Wilks, Daniel S. "Cluster analysis." *International geophysics*. Vol. 100. Academic press, 2011. 603-616.

[54] Sculley, David. "Web-scale k-means clustering." *Proceedings of the 19th international conference on World wide web*. ACM, 2010.

[55] Tran, Thanh N., Klaudia Drab, and Michal Daszykowski. "Revised DBSCAN algorithm to cluster data with dense adjacent clusters." *Chemometrics and Intelligent Laboratory Systems* 120 (2013): 92-96.

[56] Hamerly, Greg, and Charles Elkan. "Alternatives to the k-means algorithm that find better clusterings." *Proceedings of the eleventh international conference on Information and knowledge management*. ACM, 2002.

[57] Coifman, Ronald R., and M. Victor Wickerhauser. "Entropy-based algorithms for best basis selection." *IEEE Transactions on information theory* 38.2 (1992): 713-718.

[58] Khan, Shehroz S., and Amir Ahmad. "Cluster center initialization algorithm for K-means clustering." *Pattern recognition letters* 25.11 (2004): 1293-1302.

[59] Berchtold, S., D. A. Keim, and H. P. Kriegel. "An index structure for high-dimensional data." *Readings in multimedia computing and networking* 451 (2001).

[60] Lin, Hung-Yi. "A compact index structure with high data retrieval efficiency." *Service Systems and Service Management, 2008 International Conference on*. IEEE, 2008.

[61] Feldman, Dan, Melanie Schmidt, and Christian Sohler. "Turning big data into tiny data: Constant-size coresets for k-means, pca and projective clustering." *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2013.

[62] Safavian, S. Rasoul, and David Landgrebe. "A survey of decision tree classifier methodology." *IEEE transactions on systems, man, and cybernetics* 21.3 (1991): 660-674.

[63] Celebi, M. Emre, Hassan A. Kingravi, and Patricio A. Vela. "A comparative study of efficient initialization methods for the k-means clustering algorithm." *Expert systems with applications* 40.1 (2013): 200-210.

[64] Pena, José M., Jose Antonio Lozano, and Pedro Larranaga. "An empirical comparison of four initialization methods for the k-means algorithm." *Pattern recognition letters* 20.10 (1999): 1027-1040.

[65] Selim, Shokri Z., and Mohamed A. Ismail. "K-means-type algorithms: A generalized convergence theorem and characterization of local optimality." *IEEE Transactions on pattern analysis and machine intelligence* 1 (1984): 81-87.

[66] Zhao, Weizhong, Huifang Ma, and Qing He. "Parallel k-means clustering based on mapreduce." *IEEE International Conference on Cloud Computing*. Springer, Berlin, Heidelberg, 2009.

[67] Stoffel, Kilian, and Abdelkader Belkoniene. "Parallel k/h-means clustering for large data sets." *European Conference on Parallel Processing*. Springer, Berlin, Heidelberg, 1999.

[68] Wolf, Michael E., and Monica S. Lam. "A loop transformation theory and an algorithm to maximize parallelism." *IEEE transactions on parallel and distributed systems* 2.4 (1991): 452-471.

[69] Pedersen, Torben Bach, and Christian S. Jensen. "Multidimensional database technology." Computer 34.12 (2001): 40-46.

[70] Chaudhuri, Surajit, Umeshwar Dayal, and Venkatesh Ganti. "Database technology for decision support systems." Computer 34.12 (2001): 48-55.

Zoznam 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]

Práce v tlači

[1] ***Searching source code fragments using incremental clustering*** / Ďuračík Michal - Kršák Emil - Hrkút Patrik, zaslaný do *Concurrency and Computation: Practice and Experience*.

Príloha A: Testované fragmenty zdrojového kódu

V tejto prílohe si pre lepšiu predstavu uvedieme reálne ukážky zdrojového kódu, ktorý sme testovali a reprezentuje základné možnosti modifikácie zdrojového kódu študentami.

V prvej skupine ukážok budeme pracovať so zdrojovým kódom metódy na výpočet faktoriálu. Ukážka kódu je zobrazená na obrázku 47. Základná verzia zdrojového kódu obsahuje naformátovaný kód a dokumentačné komentáre.

```
1  /**
2  * Vypočíta faktorial zadaného čísla
3  */
4  public int factorial(int n) {
5      //Čísla menšie ako 1 majú faktoriál rovný 1
6      if (n < 1) {
7          return 1;
8      }
9      else {
10         //Rekurzia
11         return n*factorial(n-1);
12     }
13 }
```

Obrázok 47: Kód metódy na výpočet faktoriálu

Prvá z modifikácií ktoré na náš algoritmus nemajú vplyv sú komentáre. Algoritmus ich pri porovnávaní neberie v úvahu. Kód na obrázku 48 algoritmus označí ako 100% zhodu voči základnej verzii.

```
1  public int factorial(int n) {
2      if (n < 1) {
3          return 1;
4      }
5      else {
6          return n*factorial(n-1);
7      }
8  }
```

Obrázok 48: Kód metódy na výpočet faktoriálu bez komentárov

Rôzne pokusy o modifikáciu jednotlivých identifikátorov (obrázok 49) alebo manipulácia s bielymi (obrázok 51) znakmi taktiež nespôsobujú algoritmu žiaden problém.

```
1  public int f(int a) {
2      if (a < 1) {
3          return 1;
4      }
5      else {
6          return a*f(a-1);
7      }
8  }
```

Obrázok 49: Kód metódy na výpočet faktoriálu s pozmenenými identifikátormi

Odstránenie blokov (kučeravé zátvorky za príkazom `if`) algoritmus na rozdiel od niektorých metód používajúcich tokeny neovplyvní a aj v tomto prípade dokáže nájsť zhodu. V prípade že zmeníme štruktúru algoritmu - ako napríklad na obrázku 50, kde bola odstránená vetva `else`, algoritmus už v takto malom kúsku zdrojového kódu zhodu nedokáže nájsť.

```
1  public int f(int a) {
2  if (a<1) return 1;
3  else return a*f(a-1);
4  }
```

Obrázok 51: Kód s minimom formátovania

```
1  public int f(int a) {
2  if (a<1) return 1;
3  return a*f(a-1);
4  }
```

Obrázok 50: Kód s minimom formátovania bez vetvy else

Na druhej strane, určitá modifikácia podmienok problém nepredstavuje. Ak zmeníme skalárne hodnoty, prípade obrátime podmienku, (obrázok 52) algoritmus dokáže takúto zmenu zachytiť a príslušný kód označiť za plagiát.

```
1  public int factorial(int n) {
2      if (0 >= n) {
3          return 1;
4      }
5      else {
6          return n*factorial(n-1);
7      }
8  }
```

Obrázok 52: Kód metódy na výpočet faktoriálu so zmenenou podmienkou

V určitých prípadoch je dokonca možné jednotlivé vetvy podmienky povymieňať. Schopnosť algoritmu detegovať takúto zmenu závisí predovšetkým od veľkosti takto modifikovaného zdrojového kódu a parametrov použitých pri generovaní vektorov. Na obrázku 53 môžeme vidieť prehodenie vetiev `if` a `else` ktoré algoritmus rozpoznať dokázal.

```
1  public int factorial(int n) {
2      if (n > 1) {
3          return n*factorial(n-1);
4      }
5      else {
6          return 1
7      }
8  }
```

Obrázok 53: Kód metódy na výpočet faktoriálu s vymenenými vetvami

V závislosti od veľkosti určitého bloku algoritmus nemá problém s ľubovoľným prehadzovaním jednotlivých príkazov. V prípade že je blok malý (jeho vektor je menší ako parameter maximálna dĺžka vektora) sú všetky zmeny automaticky pohltené v tomto vektore. V opačnom prípade sme závislý na porovnávaní čiastkových vektorov, ktoré

vznikli počas fázy spájania vektorov. Na obrázku 54 môžeme vidieť dve metódy, ktoré sa líšia len poradím, v akom sú jednotlivé príkazy v tele týchto metód. Vzhľadom na to, že metódy sú dostatočne veľké na to, aby z nich mohol vzniknúť vektor, takéto modifikácie nepredstavujú pre náš algoritmus problém.

```
1  public void insert(Node node) {
2      count++;
3      node.next = first;
4      node.prew = tail;
5      tail.next = node;
6      first.prew = node;
7  }
```

```
1  public void insert(Node node) {
2      node.next = first;
3      first.prew = node;
4      node.prew = tail;
5      tail.next = node;
6      count++;
7  }
```

Obrázok 54: Porovnanie dvoch metód s poprehadzovanými príkazmi

Vo väčšine prípadov je práve metóda tým najväčším blokom kódu, ktorý sa porovnáva, takže nezáleží na ich poradí. V prípade kratších metód (typicky „gettery“ a „settery“) pre ktoré sa negenerujú vektory, je tiež možné vyhľadávať plagiáty vďaka spájaniu vektorov. V tomto prípade je opäť možné detegovať plagiáty aj pri poprehadzovaných metódach v prípade, že prehodené metódy sú v jednom posuvnom okne. Na obrázku 55 vidíme ukážku zmeny poradia dvoch krátkych metód. Vo väčšine prípadov takto zmenené poradie metód nepredstavuje zásadný problém pre detekčne schopnosti nášho algoritmu.

```
1  public void setName(String name) {
2      this.name = name;
3  }
4
5  public String getName() {
6      return name;
7  }
```

```
1  public String getName() {
2      return name;
3  }
4
5  public void setName(String name) {
6      this.name = name;
7  }
```

Obrázok 55: Zmena poradia metód

Okrem pridávania/odoberania bielych znakov, medzier a prehadzovania menších výrazov či blokov sa pri detekcii plagiátorstva často stretávame s pridávaním falošného kódu. Takýto kód študenti najčastejšie pridávajú na koniec metód, alebo priebežne.

```
1  public int maxIndex(List numbers) {
2      int maxIndex = -1;
3      int max = MIN_INTEGER;
4      int counter = 0;
5      for (int item: numbers) {
6          if (item > max) {
7              max = item;
8              maxIndex = counter;
9          }
10         counter++;
11     }
12
13     return maxIndex;
14 }
```

Obrázok 56: Kód ktorý nájde index najväčšieho prvku

Na obrázku 56 môžeme vidieť ukážku jednoduchého kódu, ktorý nájde index najväčšieho prvku v zozname čísel. V prípade modifikácie, kde na koniec tohto algoritmu pridáme nevýznamný výraz (obrázok 57) už algoritmus, na rozdiel od predchádzajúcich príkladov, neoznačí celú metódu ako zhodnú, ale len jej časť. Na obrázku vidíme červeným označenú zhodnú časť metódy. To, či algoritmus nájde zhodu aj v tomto prípade, začne závisí od zvolených parametrov pri vektorizácii zdrojového kódu.

```
1  public int maxIndex(List numbers) {
2      int maxIndex = -1;
3      int max = MIN_INTEGER;
4      int counter = 0;
5      for (int item: numbers) {
6          if (item > max) {
7              max = item;
8              maxIndex = counter;
9          }
10         counter++;
11     }
12
13     max = 0;
14     return maxIndex;
15 }
```

Obrázok 57: Pridanie nevýznamného kódu na koniec metódy

V prípade že je pôvodný zdrojový kód je vloženými časťami rozdelený až na príliš malé časti (také z ktorých neboli vygenerované vektory) algoritmus nedokáže nájsť zhodu. Pre kód na obrázku 58 algoritmus nedokázal nájsť zhodu. Na druhej strane APS vždy slúži ako podporný systém pre manuálnu kontrolu. Pri hodnotení práce človekom sa takého prípady dajú pomerne ľahko identifikovať.

```
1  public int maxIndex(List numbers) {
2      int maxIndex = -1;
3      int max = MIN_INTEGER;
4      int counter = 0;
5      maxIndex = maxIndex;
6      for (int item: numbers) {
7          if (item > max) {
8              max = item;
9              maxIndex = counter;
10         }
11         maxIndex = maxIndex;
12         counter++;
13     }
14
15     max = 0;
16     return maxIndex;
17 }
```

Obrázok 58: Pridanie veľkého množstva nevýznamného kódu

V prípadoch zámerného odoberania zdrojového kódu (napríklad zjednodušenie funkcionality pôvodnej práce) sa systém správa podobne ako v predchádzajúcom prípade. V prípade malých blokov kódu takúto zmenu nenájde a v prípade väčších sa dokáže algoritmus vysporiadať aj s týmto typom modifikácie.

V kapitole 6.2 sme sa venovali problémom pri detekcii plagiátorstva. Najväčší nedostatok nášho algoritmu vyplýva zo štruktúry s ktorou pracuje. Na obrázku 59 môžeme vidieť ukážku komplexnejšej metódy s viacúrovňovým zanorením.

```
1  public (int, int, int) stats(Matrix matrix) {
2      Set uniqueValues = new Set();
3      int max = MIN_INTEGER;
4      int min = MAX_INTEGER;
5      for (Row row: matrix) {
6          for (int value: row) {
7              if (value > 0) {
8                  uniqueValues.add(value);
9                  max = Math.max(value, max);
10                 min = Math.min(value, min);
11             }
12         }
13     }
14     return (min, max, uniqueValues.size());
15 }
```

Obrázok 59: Ukážka komplexnejšej metódy

V prípade modifikácie kódu v podmienke (na obrázku 60 sme pridali jeden výraz na riadku 11) algoritmus nedokáže nájsť podobnosť, napriek tomu že je kód z prevažnej časti zhodný. Je to spôsobené tým že nemodifikované časti na rovnakej úrovni (deklarácia premenných, a výrazy vo vnútri podmienky) sú príliš malé na to, aby z nich vznikol vektor. Riešenie tohto problému sme už načrtli v kapitole 6.2.3 no v súčasnej verzii implementované nieje.

```
1  public (int, int, int) stats(Matrix matrix) {
2      Set uniqueValues = new Set();
3      int max = MIN_INTEGER;
4      int min = MAX_INTEGER;
5      for (Row row: matrix) {
6          for (int value: row) {
7              if (value > 0) {
8                  uniqueValues.add(value);
9                  max = Math.max(value, max);
10                 min = Math.min(value, min);
11                 value++;
12             }
13         }
14     }
15     return (min, max, uniqueValues.size());
16 }
```

Obrázok 60: Modifikovaná ukážka komplexnejšej metódy