VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ROZPOZNÁNÍ PLAGIÁTŮ ZDROJOVÉHO KÓDU V JAZYCE PHP

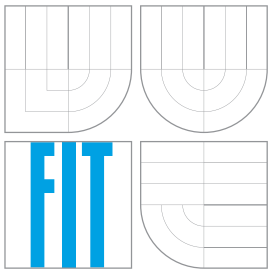
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ONDŘEJ KRPEC

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ROZPOZNÁNÍ PLAGIÁTŮ ZDROJOVÉHO KÓDU V JAZYCE PHP

PLAGIARISM RECOGNIZER IN PHP SOURCE CODE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — ONDŘEJ KRPEC
AUTHOR

VEDOUCÍ PRÁCE — Ing. ZBYNĚK KŘIVKA, Ph.D.
SUPERVISOR

BRNO 2015

Abstrakt

Cílem práce je vytvořit aplikaci, která rozpozná plagiáty projektů napsaných v jazyce PHP. Za plagiátorství lze považovat úmyslné kopírování cizího kódu, případně jeho transformací a jeho vydávání za vlastní.

Abstract

Main goal of this thesis is to create application, which can detect plagiarism in source code written in PHP language. Plagiarism is viewed as a form of code obfuscation where plagiarists deliberately perform semantics preserving transformations of original version to pass it off as their own.

Klíčová slova

PHP, plagiát, odhalování plagiátů, Halsteadova metrika

Keywords

PHP, plagiarism, plagiarism detection, Halstead metric

Citace

Ondřej Krpec: Rozpoznání plagiátů zdrojového kódu v jazyce PHP, bakalářská práce, Brno, FIT VUT v Brně, 2015

Rozpoznání plagiátů zdrojového kódu v jazyce PHP

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zbyňka Křivky, Ph.D.

.....
Ondřej Krpec
6. května 2015

Poděkování

Velmi rád bych poděkoval vedoucímu mé bakalářské práce Ing. Zbyňku Křivkovi, Ph.D. za jeho připomínky, odborné rady, čas a ochotu při konzultacích.

Obsah

Kapitola 1

Úvod

Plagiátorství je závažným problémem nejen ve vzdělávacích a vědeckých institucích. Jako takový, má velmi dlouhou a bohatou historii. Mezinárodní norma ČSN ISO 5127-2003 jej popisuje jako představení duševního díla jiného autora, půjčeného nebo napodobeného vcelku nebo zčásti, jako svého vlastního. Nejvíce případů plagiátorství se stále nachází v akademických institucích, kde studenti kopírují své práce mezi sebou, v přesvědčení, že zahladili všechny stopy vedoucí k odhalení plagiátu.

Pokud se zaměříme čistě na plagiáty ve zdrojových kódech, můžeme je definovat jako program, který byl vytvořený z jiného programu tak, aby na první pohled nebylo možné rozeznat originál od kopie. Mezi nejčastější transformace zdrojového kódu, tedy ty, na které se tato práce zaměřuje můžeme označit změnu komentářů, identifikátorů, řídících struktur, restrukturalizaci zdrojového kódu nebo změnu toku programu (výměna podmínek ve vyhodnocení `if-else`).

V rámci této bakalářské práce se pokusím ukázat způsoby, jak odhalit právě plagiáty studentských prací se zaměřením na jazyk PHP [13] a poukázat na problémy, ke kterým může při automatické detekci plagiátorství docházet.

Kapitola 2

Analýza problému

2.1 Plagiátorství

Jak již bylo uvedeno výše, plagiátorství je na akademické půdě závažným problémem a tudíž by mělo být i příslušně potrestáno. Nicméně plagiátorství zdrojových kódů sebou přináší několik problémů, které značně stěžují schopnost odlišit plagiát od originálu. Tyto problémy byly rozděleny do šesti kategorií [15], z nichž pouze jednu můžeme označit jako plagiát.

1. **Zdrojový kód třetích stran**, kterým jsou myšleny různé open-source kódy, případně různé knihovny.
2. **Nástroje na automatické generování kódu**, kde jako příklad můžeme uvést vývojové prostředí Eclipse, které je schopné automaticky vytvářet některé metody.
3. **Obvykle používané identifikátory** jako například proměnné *result* nebo *i*.
4. **Obvykle používané algoritmy** budou ve většině případech implementované stejným způsobem. Jako příklad zde lze uvést téměř libovolný řadicí algoritmus.
5. **Společný autor** jednoho nebo více programů může vytvořit více různých verzí programů, které se mohou jevit jako plagiáty, protože autor má tendenci psát kód svým naučeným způsobem.
6. **Opsaný kód**, který jako jediný může být označen jako plagiát, protože zde došlo ke kopírování nebo transformaci cizího kódu bez patřičného uvedení jeho autora.

Detekce plagiátorství na akademické úrovni sebou bohužel přináší ještě další problémy. Zadávané úkoly, hlavně ty v začátečnických kurzech programování bývají standardizované a velmi striktně zadávané, což může vyústit v podobně napsané programy, přestože studenti vypracovávali zadaný úkol samostatně.

2.2 Typy plagiátorství

Jako plagiátorství označujeme nejen jednoduché zkopírování zdrojového kódu, ale také jeho transformace. Tyto mohou být velice jednoduché, jako například pouhá změna, odebrání nebo přidávání komentářů, případně přejmenování proměnných, ale mohou být také komplikovanější jako třeba změna struktury kódu, tj. různé vnořování funkcí nebo přepsání `for` cyklů na `do-while` cykly. Případně je možné, že se autor může pokusit k zahlazení stop

využít modularity. Modularita nebo také rozčlenění programu do několika spolupracujících modulů (souborů) je při práci na rozsáhlejších projektech samozřejmostí. Tento prostředek nicméně umožňuje plagiátorovi vhodně schovat opsaný kód tak, že prohodí funkce mezi jednotlivými moduly. Na základě těchto předpokladů bylo definováno šest úrovní plagiátorství zdrojových kódů [4] od nejjednodušších technik až po ty nejkomplikovanější.

1. Úroveň – změna komentářů ve zdrojovém kódu
2. Úroveň – změna názvů identifikátorů
3. Úroveň – změna pozice proměnných ve zdrojovém kódu
4. Úroveň – změna konstant a funkcí
5. Úroveň – změna cyklů
6. Úroveň – změna struktur určených pro kontrolu toku programu

Tato práce je zaměřena na odhalení plagiátorství ve všech šesti úrovních, přičemž hlavní pozornost je věnována prvním čtyřem úrovním, které by měly být odhaleny vždy již při povrchním porovnávání zdrojových kódů.

2.3 Jazyk PHP

Jazyk PHP se ve své první formě objevil již v roce 1994, kdy se R. Lerdorf rozhodl, že vytvoří jednoduchý systém, který bude započítávat přístup na webové stránky. První verze byla napsána v jazyce PERL, nicméně vzhledem ke značnému zatížení serveru bylo poté PHP přepsáno do jazyka C. V průběhu let následovalo vydání několika dalších verzí, až nakonec v roce 2003 byla oficiálně vydána beta verze PHP5, která přinesla největší změnu v podobě přidání objektového modelu.

Jazyk získal velké uplatnění zejména z důvodu, že je nezávislý na platformě a rozdíly v různých operačních systémech se omezují pouze na několik systémově závislých funkcí. Nicméně stejně jako ostatní jazyky, má i PHP nevýhody. Největší nevýhodou je fakt, že se jedná o jazyk interpretovaný, což znamená, že při jakémkoliv spuštění i toho nejmenšího skriptu, je potřeba soubor s tímto skriptem znovu kompilovat.

2.3.1 Specifika jazyka PHP

Jak již bylo uvedeno výše, tak objektový model byl do jazyka přidán až později, což nyní programátorům umožňuje vybrat si, jestli budou své programy psát využívajíc imperativního nebo objektově orientovaného paradigmatu. Tento výběr ovšem také napomáhá šíření plagiátorství, protože lze přepsáním originálního zdrojového kódu do jiného programovacího paradigmatu vytvořit na první pohled odlišný kód.

Další důležitou vlastností tohoto jazyka je také to, že se jedná o dynamicky typovaný jazyk tzn. že datový typ je vázán na hodnotu, nikoliv na proměnnou. Nezapomeňme také na fakt, že PHP obsahuje pouze asociativní pole, tedy ve skutečnosti se jedná o hašovací tabulky [10], které ukládají páry klíč – hodnota. Klíčem následně může být pouze celé číslo nebo řetězec. Jedno pole může dokonce obsahovat jak klíče celočíselné, tak řetězcové.

2.4 Přehled stávajících nástrojů pro detekci plagiátů

Ve všech případech vzniku plagiátů se jedná o stejný přístup k tvorbě programu. Struktura originálního programu je pozměněna tak, aby sémantika kódu byla zachována. K detekci plagiátorství tedy potřebujeme mít k dispozici nástroje, které jsou schopné buď porovnávat texty nebo rovnou analyzovat zdrojové kódy. Tato kapitola se zabývá přesně takovými nástroji, které již jsou aktuálně dostupné.

Nástrojů na efektivní odhalení plagiátů je dnes nepřeberné množství, nicméně pouze málo z nich je schopné analyzovat zdrojové kódy a ještě méně je jich schopné analyzovat pro jazyk PHP.

2.4.1 The Sherlock Plagiarism Detector

Tento nástroj [14] patří k těm rychlejším detektorům, které se dají najít. K detekci plagiátů využívá digitálních podpisů. Digitální podpis je číslo, které je vytvořeno pozměněním několika slov na vstupu do řady bitů a spojení těchto bitů do jednoho výsledného čísla. Nástroj funguje jak na textových souborech, tak zdrojových kódech a ostatních digitálních formátech. Samotný program je napsán v jazyce C a hojně využívá nízkoúrovňové operace a efektivní algoritmy. To mu zajišťuje rychlé získávání výsledků, nicméně jeho velkou nevýhodou je fakt, že je schopen dokumenty porovnávat pouze bez jakékoliv předchozí analýzy. Z tohoto důvodu je sice nástroj vhodný pro porovnání textů, nikoliv však zdrojových kódu, u kterých je vhodné porovnávat kódy na určité úrovni abstrakce.

2.4.2 MOSS

MOSS [12] je zkratka anglického „Measure of Software Similarity“, což v překladu znamená „Odhad Podobnosti Programů“. Tento systém vyvinul v roce 1994 A. Aiken pro univerzitu v Berkeley. Jeho velkou výhodou je jeho efektivnost, která byla ověřena léty používání na mnoha univerzitách po celém světě. K odhalení plagiátů využívá tento systém otisky dokumentů, kterým je věnována kapitola 2.5.5. Jeho nevýhodou je fakt, že program není open-source. Navíc přesto, že podporuje velké množství programovacích jazyků, ke dnešnímu dni mu stále schází rozšíření, které by umožnilo odhalovat plagiáty v jazyce PHP.

2.4.3 CodeMatch

Poslední zmíněný program je schopen porovnání zdrojových kódů v nepřeberném množství jazyků v relativně krátkém čase. Navíc na rozdíl od výše zmíněných nástrojů je jako výsledek schopen vyprodukovat databázi s projekty a jejich podobností a následně podezřelé páry projektů vyexportovat jako HTML. Každý takový pár následně obsahuje i detailní zprávu, ve které lze nalézt důvody, proč byly tyto páry vyhodnoceny jako možné plagiáty.

CodeMatch [11] na určení podobnosti mezi dvěma programy využívá hned několik algoritmů např. porovnání komentářů, sekvencí instrukcí nebo identifikátorů. Z každé fáze jsou poté získány mezivýsledky a ty poté interpretovány na stupnici 0 – 100, kdy větší skóre znamená, že je více pravděpodobné, že jedna ze zkoumaných prací bude plagiát. Nicméně velkou nevýhodou je, že tento nástroj není distribuován pod žádnou volně dostupnou licencí, nýbrž je placený.

2.5 Přehled stávajících metod na detekci plagiátorství

Jak již bylo naznačeno v úvodu, plagiát zdrojového kódu můžeme definovat jako program, který byl vytvořen z jiného programu s určitým počtem transformací tak, aby na první pohled nebylo možné poznat plagiát od originálu. Z tohoto předpokladu vyplývá velmi důležitý požadavek na vlastnost výsledného programu a metod, které využívá. V této oblasti již bylo provedeno mnoho výzkumů [2], které zpracovávají přehled použitelných metod pro detekci plagiátů pro přirozené i programovací jazyky [1]. V této kapitole nás ovšem budou zajímat pouze metody, které se zaměřují na nalezení plagiátů ve zdrojových kódech.

2.5.1 Porovnání textových řetězců

Jednou z nejjednodušších metod na detekci plagiátorství v přirozených jazycích i zdrojových kódech je porovnávání textových řetězců. Tato technika nebere v potaz sémantiku dokumentů, ale pouze pořadí slov a písmen. Toto prosté porovnání implementuje například UNIXový nástroj *diff*, který dokáže porovnat dva vstupní dokumenty, zobrazit jejich společné části a vyhodnotit podobnost těchto dokumentů na základě nalezení nejdelší podposloupnosti [5] společné všem posloupnostem v určené množině posloupností. Na obrázcích 2.1 a 2.2 lze vidět ukázku dvou podobných zdrojových kódů, které slouží k výpočtu determinantu kvadratické rovnice. Následně na obrázku 2.3 je zobrazeno, jak může vypadat výstup nástroje *diff* pro dva podobné zdrojové texty.

```
1  function determinant($a, $b, $c) {
2      return $b*$b - 4*$a*c;
3  }
```

Příklad 2.1: Původní funkce na výpočet determinantu kvadratické rovnice.

```
1  function plagirism($x, $y, $z) {
2      $tmp = $y*y;
3    return $tmp - 4*$x*$z;
4  }
```

Příklad 2.2: Pozměněná funkce na výpočet determinantu kvadratické rovnice.

Z výsledku programu *diff* 2.3 vyplývá, že tato metoda zkoumání podobnosti zdrojových textů není úplně vhodná, hlavně z důvodu, že žádným způsobem neprovádí analýzu zdrojového textu a výsledný plagiát tak může být lehce zakrýt pomocí pouhého přejmenování proměnných nebo změnou struktury zdrojového kódu.

```
1    1c1
2    < function determinant($a, $b, $c)
3    ---
4    > function plagirism($x, $y, $z)
5    3c3,4
6    <     return $b*$b - 4*$a*c;
7    ---
8    >     $tmp = $y*y;
9    >     return $tmp - 4*$x*$z;
```

Příklad 2.3: Ukázka výstupu UNIXové utility *diff*.

2.5.2 Tokenizace

Tokenizace je proces, který probíhá v rámci lexikální analýzy, jenž je součástí každého překladače.[1] V průběhu tokenizace jsou načítány znaky ze vstupního zdrojového souboru. Tyto znaky reprezentují zdrojový kód a v průběhu lexikální analýzy jsou z těchto znaků vytvořeny symboly programu zvané lexémy. Tyto lexémy jsou následně reprezentovány ve formě tokenů. Token je řetězec složený z jednoho nebo více znaků, které jsou v daném jazyce důležité jako skupina. Jako příklad se lze podívat na tabulku 2.1, kde lze vidět výstup procesu tokenizace pro vstupní kód *result = 5 * 5*. Každý výstupní token má kromě původní hodnoty také určený význam v rámci zdrojového kódu.

Typ	**Hodnota**
T_VARIABLE	result
T_ASSIGNMENT	=
T_LNUMBER	5
T_MULTIPLY	*
T_LNUMBER	5
T_SEMICOLON	;

Tabulka 2.1: Tokeny vygenerované pro výraz *result = 5 * 5*

Z tabulky 2.1 také vyplývá, že díky zajištění určité míry abstrakce nad zdrojovým kódem, lze dosáhnout za pomocí tokenizace lepších výsledků při porovnávání dvou podobných projektů než při využití pouhého porovnání textových řetězců. Pokud budeme porovnávat výsledky procesu tokenizace mezi sebou, zjistíme, že pouhé přejmenování proměnných nebo změna komentářů nemá na výsledek porovnání žádný vliv. Této metody je využíváno prakticky ve všech nástrojích na odhalování plagiátů zdrojových kódů jako příklad si můžeme uvést třeba již zmíněný systém MOSS 2.4.2 nebo CodeMatch 2.4.3.

Samozřejmě proces tokenizace není žádná technika k přímému odhalení plagiátorství, nicméně je velmi vhodný pro přípravu zdrojových textů, ke zpracování dalšími, níže uvedenými metodami.

2.5.3 Halsteadova metrika

Celým názvem Halsteadova metrika velikosti programu je softwarová metrika, kterou v roce 1977 představil M. H. Halstead [7]. Je založena na předpokladu, že všechny zdrojové texty se skládají z konečného počtu programových jednotek, tzv. tokenů 2.5.2, které jsou rozeznatelné překladačem. Jak již bylo zmíněno, tak počítačový program poté může být brán jako posloupnost tokenů, které mohou být klasifikovány jako operátory nebo operandy. Cílem této metriky je identifikovat takové vlastnosti programů, které by byly snadno vyčíslitelné, a které by mezi sebou měly určité souvislosti. Proto Halstead definoval čtyři základní proměnné, ze kterých lze poté vypočítat konkrétní metriky programu. Tyto proměnné jsou následující:

- η_1 = počet unikátních operátorů
- η_2 = počet unikátních operandů

[1] Více o fázích překladače na http://www.abclinuxu.cz/clanky/programovani/jazyky-a-prekladace-1-uvod

- N_1 = celkový počet operátorů
- N_2 = celkový počet operandů

Pokud ze zdrojového textu jsme schopní vypočítat výše zmíněné proměnné, jsme poté následně schopni vypočítat i Halsteadovy metriky zdrojového kódu pomocí pomocí následujících vzorců.

$$\eta = \eta_1 + \eta_2 \tag{2.1}$$
$$N = N_1 + N_2 \tag{2.2}$$
$$\hat{N} = \eta_1 log\eta_1 + \eta_2 log\eta_2 \tag{2.3}$$
$$V = N * log_2\eta \tag{2.4}$$
$$D = \frac{\eta_1}{2} * \frac{N_2}{\eta_2} \tag{2.5}$$
$$E = D * V \tag{2.6}$$

Kde jednotlivé výsledky znamenají:

- η – Velikost slovníku
- N – Délka program
- $\hat{N}$ – Odhadnutá délka programu
- V – Objem
- D – Programová náročnost
- E – Programátorské úsilí

Tyto proměnné lze vypočítat ze zdrojového kódu velice rychle, ať už pro celý kód nebo jeho části. Získané hodnoty poté stačí porovnat s hodnotami získanými z jiných zdrojových kódu. Nicméně nevýhodou této metriky je bohužel fakt, že je možné získat dva stejné výsledky pro rozdílně pracující části kódu. I přes tento nedostatek se tyto metriky dají úspěšně využít při odhalovaní plagiátorství zdrojových textů.

```
1  function getAverage($array) {
2    $value = 0;
3    foreach ($array as $arrayItem) {
4      $value += $arrayItem;
5    }
6    return $value / count($array);
7  }
```

Příklad 2.4: Ukázka kódu pro výpočet Halsteadových metrik.

Jako příklad lze využít kód definovaný v 2.4, pro který jsou výsledné metriky následující:

$$\hat{N} = \eta_1 log\eta_1 + \eta_2 log\eta_2 = 17.47 \tag{2.7}$$
$$V = N * log_2\eta = 20.02 \tag{2.8}$$
$$D = \frac{\eta_1}{2} * \frac{N_2}{\eta_2} = 3.00 \tag{2.9}$$
$$E = D * V = 60.06 \tag{2.10}$$

2.5.4 Levenshteinův algoritmus

Levenshteinův algoritmus je jedním z nejpoužívanějších algoritmů pro detekci plagiátů ať již ve zdrojových kódech nebo v přirozeném jazyce. Vymyslel jej v roce 1965 V. Levenshtein [6] a pracuje s předpokladem, že vzdálenost dvou řetězců je definována jako minimální počet operací vkládání, mazání a substituce takových, aby po jejich provedení byly zadané řetězce totožné. Tuto techniku lze poté uplatnit na dva zdrojové kódy, ať už bez jakékoliv předchozí analýzy nebo po provedení procesu tokenizace a z výsledků algoritmu rozhodnout, jak jsou si vstupní texty podobné. Z matematického hlediska, Levenstheinova vzdálenost mezi dvěma řetězci a, b je dána $lev_{a,b}(|a|, |b|)$ kde

$$lev_{a,b}(i,j) = \begin{cases} max(i,j) & \text{pokud } min(i,j) = 0, \\ min \begin{cases} lev_{a,b}(i-1,j) + 1 \\ lev_{a,b}(i,j-1) + 1 \\ lev_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{jinak.} \end{cases} \tag{2.11}$$

kde $1_{(a_i \neq b_j)}$ je charakteristická funkce[2] rovna 0, když $a_i = b_j$ a 1 jinak.

Jako příklad Levenshteinovy vzdálenosti lze uvést třeba jména *Pavel* a *Pavla*. Jejich vzdálenost je dva, protože minimální počet operací, které je potřeba provést k transformaci jednoho jména ve druhé je právě dvě. Tedy pokud bychom chtěli transformovat jméno *Pavel* na jméno *Pavla*, stačí pouze odebrat ze slova *Pavel* písmeno *e* a ke vzniklému slovu *Pavl* přidat na konec písmeno *a*. Tudíž po dvou operacích nám vznikne výsledné jméno *Pavla*.

Samotný algoritmus má bohužel nevýhodu, kdy v případě, že porovnávané řetězce jsou příliš dlouhé, tak se výrazně zvyšuje doba výpočtu algoritmu. Ta totiž roste zhruba úměrně s velikostí porovnávaných řetězců.

2.5.5 Otisky dokumentů

Využívání otisků dokumentů je jedním z nejpopulárnějších způsobů na automatizované odhalování plagiátorství. Používá jej třeba již zmíněný systém MOSS 2.4.2. Metoda funguje na principu identifikace některých specifických řetězců v dokumentu, ze kterých následně vytvoří unikátní otisk dokumentu. Teoreticky by tak rozdílné dokumenty měly mít vždy otisky rozdílné a stejně tak, podobné dokumenty by měly mít otisky podobné.

Většina technik implementující tuto techniku je postavena na využívání *k-gramů*. *K-gram* je sousedící podřetězec o délce *k*. Zdrojový text je poté rozdělen do těchto *k-gramů*, kde *k* je parametr zvolený uživatelem. Je třeba vzít v úvahu, že existuje skoro stejný počet *k-gramů* jako je znaků v dokumentu, protože každý nový znak je začátkem nového *k-gramu* (výjimkou jsou znaky na $k - 1$ pozicích). Na příkladu 2.5 lze vidět, jak vytvoření *k-gramů funguje*. Nejprve jsou odstraněny bílé znaky a následně je vytvořena sekvence *k-gramů*. Nad těmito *k-gramy* je poté spuštěna hašovací funkce a z jejího výsledku poté vybrán vzorek, který slouží jako otisk dokumentu. Velmi často pro výběr vzorků volí pouze haše, které odpovídají vzorci $0 \bmod p$[3], pro nějaké fixně zvolené p. Tento přístup je používaný především z důvodu, že je snadno implementovaný a zachovává pouze $1/p$ hašů jako otisků. Nevýhodou tohoto přístupu je to, že nám nedává žádné záruky, že podobné budou podobné části mezi zdrojovými kódy detekovány, protože *k-gram* sdílený mezi dvěma kódy je nalezen pouze v případě, že se jedná o haš o velikosti $0 \bmod p$.

[2] Více o charakteristické funkci na http://www.math.muni.cz/~forbel/M3121/M3121_S4.pdf
[3] Více o výběru vzorků na http://igm.univ-mlv.fr/~mac/ENS/DOC/sigmod03-1.pdf

```
1  public static String s = "Hello";
2  publicstaticStrings="Hello";
3  publi ublic blics licst icsta cstat stati tatic aticS ticSt icStr cStri
     Strin tring rings ings= ngs=" gs="H s="He ="Hel "Hell Hello ello" llo";
4  10701 11107 9382 10239 10003 9496 10975 11013 9314 11035 10000 9401 8022
     11062 10851 10851 10034 10476 9860 10805 5742 3364 6960 9660 10306
5  10701 10003 9314 8022 10476 5742 3364
```

Příklad 2.5: Ukázka principu vytvoření otisku dokumentu ze zdrojového textu.

K překonání tohoto problému se využívají okna. Definujeme okno o velikosti w tak, aby obsahovalo w po sobě jdoucích hašů *k-gramů*. Vybráním alespoň jednoho otisku z každého takového okna omezujeme maximální mezeru mezi vybranými otisky. Tímto docílíme toho, že algoritmus je schopný detekovat minimálně jeden *k-gram* v jakémkoliv sdíleném podřetězci, který má délku alespoň $w + k - 1$.

Winnowing

Jedním z nejvíce používaných způsobů pro vybrání vzorku hašů *k-gramů* je metoda zvaná Winnowing [9]. Metoda je založena na dvou principech.

1. Pokud existuje podřetězec, který je alespoň tak dlouhý, jako délka garantovaného prahu t[4], tak musí být vždy nalezen.

2. Nechceme detekovat podobné řetězce, které jsou kratší než práh šumu[5] k.

Konstanty t a $k \leq t$ jsou voleny uživatelem. Konstantu t volíme proto, že velmi krátké úseky kódu, jako třeba klíčová slova, které jsou si podobné jsou pro detekci plagiátů nezajímavé. Co se týče konstanty k, tak čím větší bude její hodnota, tím více si můžeme být jisti, že nalezené podobnosti nejsou náhodné. Na druhou stranu vyšší hodnota k omezuje možnost zjistit změny, které byly provedeny reorganizací zdrojového textu. Z tohoto důvodu je nutné zvolit konstantu k optimálně tak, aby byly odhaleny restrukturalizace zdrojového textu a zároveň nedocházelo k detekcím příliš krátkých kusů kódu.

```
1  public static String s = "Hello";
2  publicstaticStrings="Hello";
3  publi ublic blics licst icsta cstat stati tatic aticS ticSt icStr cStri
     Strin tring rings ings= ngs=" gs="H s="He ="Hel "Hell Hello ello" llo";
4  10701 11107 9382 10239 10003 9496 10975 11013 9314 11035 10000 9401 8022
     11062 10851 10851 10034 10476 9860 10805 5742 3364 6960 9660 10306
5  (10701 11107 9382 10239) (11107 9382 10239 10003) (9382 10239 10003 9496)
     (10239 10003 9496 10975) (10003 9496 10975 11013) (9496 10975 11013 9314)
     (10975 11013 9314 11035) (11013 9314 11035 10000) (9314 11035 10000 9401)
     (11035 10000 9401 8022) (10000 9401 8022 11062) (9401 8022 11062 10851)
     (8022 11062 10851 10851) (11062 10851 10851 10034) (10851 10034 10476
     9860) (10034 10476 9860 10805) (10476 9860 10805 5742) (9860 10805 5742
     3364) (10805 5742 3364 6960) (5742 3364 6960 9660) (3364 6960 9660 10306)
6  9382 9496 9314 8022 10034 9860 5742 3364
```

Příklad 2.6: Tvorba otisku textu z příkladu 2.5 za využití metody Winnowing pro okno o velikosti 4

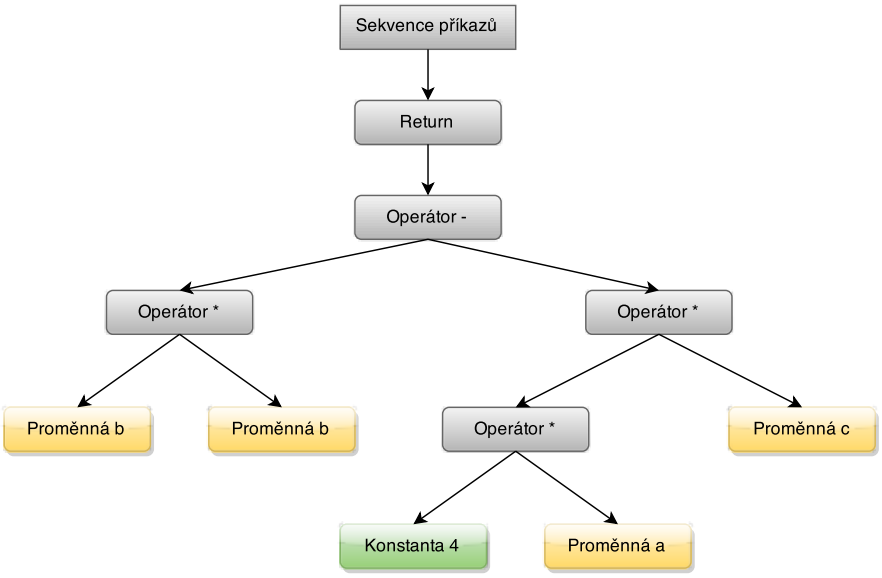
[4] Minimální délka řetězce, který se bude vyhledávat.
[5] Maximální délka řetězce, který se bude ignorovat.

Výběr hašů je zde založen na principu, že z každého okna vybereme minimální hodnotu haše. V případě, že v jednom okně existuje více než jeden haš s minimální hodnotou, je vybrán ten s nejpravějším výskytem. Takto vybrané haše jsou poté uloženy jako otisky dokumentu.

2.5.6 Abstraktní syntaktické stromy

Abstraktní syntaktický strom (AST) je vlastně stromovou reprezentací abstraktní syntaktické struktury zdrojového kódu napsaného v programovacím jazyce. Jeho vnitřními uzly jsou operátory a listy operandy. Ačkoliv se těchto stromů využívá v překladačích především pro optimalizaci kódu, pomocí algoritmů na porovnání stromových struktur [3] lze tyto stromy využít i k jejich porovnání.

Cílem při porovnávání zdrojových kódu pomocí AST je nalezení maximálního možného počtu společných podstromů a spočítání jejich vzdálenosti. Tato technika je velice podobná výpočtu Levenshteinovy vzdálenosti 2.5.4, kdy vzdálenost mezi jednotlivými stromy je rovna počtu operací přidání, odebrání nebo substituce nutné k transformaci jednoho stromu do druhého.



Obrázek 2.1: Ukázka AST pro funkci na výpočet determinantu definovanou dříve 2.1

Syntaxe je u AST abstraktní v tom smyslu, že nereprezentuje každý detail, který se v reálné syntaxi vyskytuje. Například na obrázku 2.1 vidíme, že se zde nevyskytuje středník oznamující konec příkazu. Případně takové seskupující závorky jsou ve stromové struktuře implicitní a syntaktické konstrukce jako *if* – *podmínka* – *then* mohou být vyznačeny pouze jediným uzlem se dvěma větvemi. To činí abstraktní syntaktické stromy odlišné od stromů konkrétních a umožňuje je efektivně využívat pro detekci plagiátorství, neboť je v nich automaticky obsažena jistá úroveň abstrakce zdrojového kódu.

Kapitola 3

Návrh řešení

Po analýze problémů, které mohou nastat při detekci plagiátů se budeme věnovat návrhu aplikace, která bude schopná efektivně odhalit plagiátorství ve studentských prací za pomoci některých metod zmíněných v kapitole 2.5.

Jedním z požadavků na výslednou aplikaci bylo její rozčlenění do samostatně fungujících částí, které bude možno využít jak jednotlivě, tak v celku. Z tohoto důvodu bylo navrženo, aby celý systém fungoval v několika fázích viz. obrázek 3.1, které si detailněji popíšeme v následujících kapitolách.

Aplikace byla navrhována tak, aby byla snadno udržovatelná a rozšiřitelná, třeba v navazující diplomové práci. Z tohoto důvodu, byly všechny fáze kromě první navrženy tak, aby byly jazykově nezávislé a tak v případě potřeby kontroly plagiátů i v jiném jazyce než PHP je nutné přidat pouze syntaktický analyzátor zdrojového jazyka. Ve finálním návrhu aplikace se využívá čtyř fází, které jsou sice schopné pracovat i samostatně, ale navazují na sebe a tvoří tak jeden fungující celek.

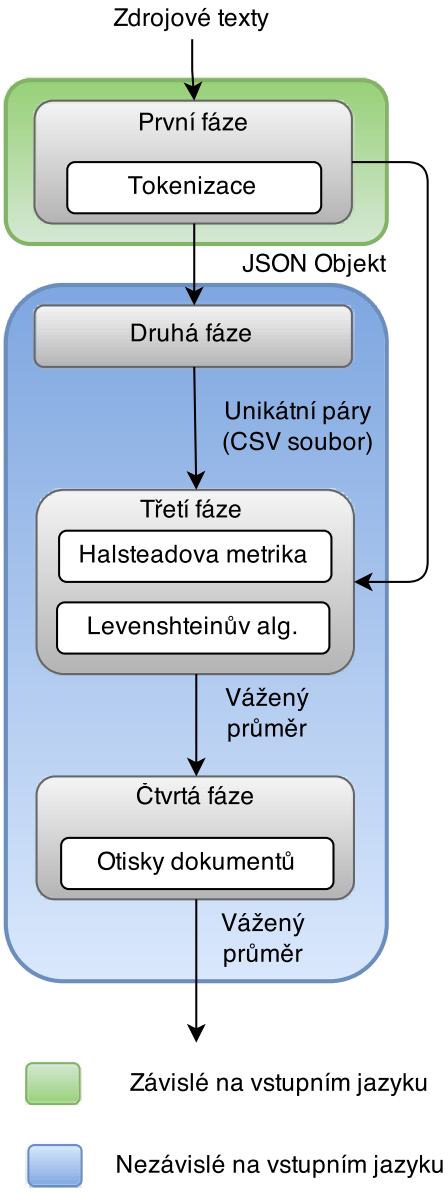
3.1 Zpracování vstupních dat

Pro první fázi programu bylo potřeba navrhnout systém, který by připravil vstupní zdrojové texty pro další fáze tak, aby mohly být snadno a efektivně porovnávány. Jelikož je v případě naší aplikace nutné, aby prováděla analýzu zdrojového textu, bylo nutné vymyslet strukturu, do které by se data o zdrojových textech ukládala. Jako vhodné řešení se jevilo využít některého ze serializačních formátů. Vzhledem k objemu dat, které se budou při analýze kódů ukládat, byl nejlepší možností serializační formát JSON[1]. Důvodů pro zvolení právě tohoto formátu pro uchování dat bylo hned několik. Za prvé je tento formát velice kompaktní a k samotným datům nepřidává mnoho řídících a kontrolních dat navíc a za druhé je velmi snadno čitelný i pro člověka.

Serializační formát bylo nutné zvolit právě z důvodu rozčlenění naší aplikace do jednotlivých fází. Tedy v první fázi je možné zpracovat a analyzovat vstupní data, uložit je do struktury a tuto strukturu následně uložit do souboru a uchovat ji tak pro pozdější použití. Tento přístup umožní uživateli vytvořit si šablony zdrojových kódů, které pak lze použít pro srovnání s dalšími, novějšími zdrojovými kódy bez toho, aniž by bylo potřeba analyzovat i zdrojové kódy textů, které jsou uložené ve vytvořené šabloně.

Výše zmíněný přístup umožňuje při využití šablon ušetřit podstatnou část strojového času potřebnou na načtení všech zdrojových dat, případně pokud je aplikaci již dodána

[1] Více o formátu JSON na http://tools.ietf.org/html/rfc7159



Obrázek 3.1: Návrh struktury nástroje na odhalení plagiátorství v jazyce PHP

šablona se zpracovanými zdrojovými texty, lze tuto fázi programu úplně přeskočit a spustit až druhou fázi naší aplikace.

3.2 Stránkování

Odhalování plagiátorství spotřebuje spoustu strojového času, při velkém množství času i několik hodin. Z tohoto důvodu by bylo velice nepříjemné zjistit, že se aplikace neplánovaně ukončila, což se klidně může stát, pokud je aplikace spuštěna na serveru s omezeným

procesorovým časem.

Jako způsob překonání tohoto problému bylo navrženo využít metody stránkování. Jedná se o systém, kdy je porovnán pouze uživatelem specifikovaný počet párů. Počet těchto párů tvoří z celkového počtu párů jednu stránku. Pokud by bylo potřeba vyhodnotit následující páry projektů stačilo by tak pouze pomocí vstupních parametrů výsledného programu zadat počet párů určených ke zpracování a nadcházející číslo stránky.

Avšak uživatel nemusí být schopen předpokládat, kolik dvojic stihne výsledná aplikace vyhodnotit před tím, než vyprší jí přidělený čas. Proto jsem se rozhodl, po vytvoření aplikace provést sadu experimentů 4.4, které následně definují ideální výchozí hodnotu pro velikost jedné stránky.

3.3 Vytvoření dvojic

Za předpokladu, že již máme zpracována vstupní data je potřeba vytvořit dvojice zdrojových kódu, které se budou určené k porovnání v dalších fázích. Proto, jak lze názorně vidět i na obrázku 3.1, vstupem do druhé fáze projektu bude JSON objekt uchovávající analyzovaná data a informace o projektech.

Cílem této fáze je vytvořit unikátní dvojice mezi všemi projekty, které jsou určeny ke kontrole plagiátů. Stejně jako v případě první fáze bylo třeba navrhnout i tuto fázi tak, aby ji bylo možné spustit samostatně při dodání validního vstupního souboru. Jako výstup druhé fáze byl poté zvolen formát CSV[2], jelikož v něm lze snadno uchovat názvy dvojic, které jsou určené k porovnání.

```
1  xlogin00,xlogin01
2  xlogin00,xlogin99-template
3  xlogin01,xlogin99-template
```

Příklad 3.1: Ukázka části CSV souboru s vytvořenými páry určenými k porovnání pro zdrojové kódy souborů xlogin00 xlgoin01 a šablony xlogin99.

Jak již bylo zmíněno v sekci 3.1, aplikace byla navrhnuta tak, aby bylo možné vytvořit šablony zdrojových kódů, které poté bude možné porovnávat s ostatními projekty. Ovšem při vytváření unikátních párů určených k porovnání by nedávalo smysl porovnávat zdrojové texty v šablonách mezi sebou. Z tohoto důvodu jsou veškeré zdrojové texty ze vstupní šablony označeny příponou *-template* viz. obrázek 3.1, což nám poté v implementaci umožní nevytvářet dvojice právě mezi těmito zdrojovými texty.

Velmi důležitou části návrhu je fakt, že takto realizovaná druhá fáze a samozřejmě všechny následující jsou zcela nezávislé na vstupním jazyku a v případě dodání vstupního JSON objektu lze porovnat zdrojové texty v jakémkoliv jazyce.

3.4 Povrchové vyhledávání plagiátů

Tato sekce popisuje třetí fázi projektu, která slouží k redukci počtu dvojic, které se porovnávají. Redukce je zde navrhnuta tak, že dvojice zdrojových textů načtené z CSV souboru budou nejprve nalezeny ve vstupním JSON objektu a poté následně porovnány za využití Halsteadových metrik 2.5.3 a Levenshteinova algoritmu 2.5.4. Obě metody jsou uzpůsobeny k tomu, aby jejich výsledek bylo možné vyjádřit jako procentuální podobnost zdrojových

[2]Více o formátu CSV na http://tools.ietf.org/html/rfc4180

textů, které jsou aktuálně porovnávány. K seskupení jednotlivých dílčích výsledků z jednotlivých metod a získání jedné výsledné hodnoty v řádech procent je v aplikaci využíváno váženého aritmetického průměru[3].

Jelikož se jedná o samostatnou fázi této aplikace, je nutné výsledky této fáze zase ukládat do připraveného souboru. Jako ideální se jeví využít vstupního CSV souboru a ke každé dvojici přidat míru její podobnosti, jak lze vidět například na obrázku 3.2.

```
1  xlogin00,xlogin01,54%,1
2  xlogin00,xlogin99-template,61%,8
3  xlogin01,xlogin99-template, 83%,6
```

Příklad 3.2: Ukázka vyhodnocení povrchového vyhledání plagiátů pro zdrojové texty z příkladu 3.1.

Dvojice zdrojových textů, které překračují určitou experimentálně zjištěnou mez podobnosti viz. 4.4 by poté měly být určeny k důkladnějšímu prohledání ve čtvrté fázi aplikace. Nicméně tato hranice není jediným měřítkem, zda by bylo vhodné pár, který je aktuálně porovnáván prozkoumat důkladněji. Z důvodu přesnější detekce, je do výstupního souboru z této fáze přidán i počet Levenshteinových bloků, které si byly mezi zdrojovými texty velmi podobné[4].

3.5 Hloubkové vyhledávání plagiátů

Třetí fáze byla navržena tak, aby dokázala vyfiltrovat dvojice zdrojových kódů, které neobsahují žádné známky o tom, že by se mohlo jednat o plagiát. Takovéto řešení umožní více se věnovat dvojicím, které vykazují určité podezření. Pro důkladnější kontrolu takovýchto dvojic byl vybrán algoritmus Winnowing [9] implementující techniku tvorby otisků dokumentu.

Čtvrtá fáze tedy pracuje s ohodnoceným CSV souborem, který byl v předchozí fázi vytvořen, případně přímo dodán na vstup této fáze. S přihlédnutím k faktu, že algoritmus používáný v této fázi aplikace je využíván MOSS 2.4.2, jedním z nejpopulárnějších nástrojů pro odhalování plagiátorství v akademickém prostředí, rozhodl jsem se toho využít a pro nastavení algoritmu použít hodnoty, které byly zjištěny jako nejvhodnější v průběhu používání právě tohoto nástroje.

Samotné porovnání již probíhá obdobně jako ve třetí fázi, kdy jsou zdrojové texty porovnávány pomocí Levenshteinovy vzdálenosti. Výsledky této metody je taktéž vhodné za využití váženého aritmetického průměru vyhodnotit jako finální hodnotu podobnosti páru zdrojových textů. Tyto výsledky je poté vhodné podobně jako na konci třetí fáze uložit do souboru ve formátu CSV, aby bylo dohledatelné, které dvojice zdrojových kódů vykazují vysokou míru podobnosti a bylo možné je tak manuálně prohlédnout a rozhodnout zda je podezření oprávněné nebo nikoliv.

[3] Více o váženém aritmetickém průměru na http://www.naseskola.net/aktual/08-09/vazeny_prumer.pdf
[4] Hodnota určující přílišnou podobnost mezi dvěma bloky byla určena experimentálně viz. kapitola 4.4.

Kapitola 4

Implementace a testování

Tato kapitola popisuje implementaci nástroje na odhalení plagiátorství navrženého výše 3, jeho testování a provedené experimenty. Celý nástroj je implementován v jazyce PHP verze pět. Tento jazyk byl pro implementaci vybrán z důvodu velmi efektivně implementovaných vestavěných funkcí na práci se soubory, tvorbě JSON objektů a funkcí určených pro syntaktickou analýzu zdrojových textů v jazyce PHP, na který se tento nástroj zaměřuje.

Nástroj byl implementován s využitím objektově orientovaného paradigmatu a ze samotného nástroje jde následně vidět, že při jeho tvorbě byla využita jistá inspirace návrhovým vzorem *model–view–controller* [8], kdy veškerá kontrolní logika je implementována v balíčku *controllers* a veškeré objekty uchovávající data v balíčku *entity*.

4.1 Zpracování vstupních dat

Pro samotné zpracování dat je nejprve nutné načíst všechny zdrojové texty, o což se velmi rychle postará vestavěná funkce *file_get_contents*. Tyto zdrojové texty jsou následně převedeny na posloupnost tokenů pomocí další vestavěné funkce *token_get_all*. Využívání těchto funkcí vede k velice rychlému zpracování stovek zdrojových textů v řádech desítek sekund.

V momentě, kdy jsou všechny zdrojové kódy načteny do paměti a převedeny na posloupnost tokenů, jsou z nich odstraněny veškeré bílé znaky a pokud byl při startu aplikace zadán i parametr *–c*[1], tak jsou ze zdrojových textů odstraněny i veškeré komentáře.

Tímto jsou všechny zdrojové kódy připraveny ke zpracování. Byla implementována struktura JSON objektu, jejíž ukázku lze vidět na obrázku 4.1. Všechny zdrojové kódy, které byly převedeny na posloupnost tokenů jsou zpracovány a uloženy do této struktury. Jak lze vidět z ukázky níže, každý zdrojový kód obsahuje informace o tokenech, ze kterých se skládá, vypočtené hodnoty Halsteadovy metriky pro jednotlivé funkce a metody, které byly ve zdrojovém kódu nalezeny a také sadu Levenshteinových bloků o maximální délce 255 znaků. Tato délka byla vybrána proto, že při porovnávání delších řetězců pomocí Levenshteinova algoritmu se velmi výrazně zvyšuje doba potřebná k porovnání dvou bloků.

Jelikož je JSON objekt v aplikaci reprezentován jako třída odvozená od třídy *std::Object* je nutné ji při ukládání do souboru transformovat do tradičního JSON objektu pomocí vestavěné funkce *json_encode*. Ukládání se provádí ihned po zpracování zdrojových textů do zmíněného objektu a umožňuje tak využít zpracované zdrojové texty i později. V případě potřeby tak lze zpracovaná data zase ze souboru načíst a převést pomocí vestavěné funkce

[1] Více o nastavitelných parametrech programu lze najít v příloze.

json_decode. Vytvořením a uložením JSON objektu končí první fáze a program pomalu postupuje do dalších.

```
{
  "Název projektu": {
    "path" : "Cesta k souboru",
    "dir" : "Název projektu",
    "files" : [
      {
        "filename" : "Název souboru",
        "content" : {
          "tokens" : [
            [Typ tokenu, hodnota tokenu, pozice v kódu]
          ],
          "halsteadBlocks" : [
            {
              "operators" : Počet operátorů ve funkci,
              "operands" : Počet operandů ve funkci,
              "uniqueOperators" : [
                Výčet unikátních operátorů ve funkci
              ],
              "uniqueOperands" : [
                Výčet unikátních operandů ve funkci
              ],
              "programLength" : Odhadnutá délka funkce,
              "volume" : Objem funkce,
              "difficulty" : Programová náročnost funkce
            }
          ],
          "levenshteinBlocks" : [
            [TYP_TOKENUTYP_TOKENUTYP_TOKENU]
          ]
        }
      }
    ]
  }
}
```

Příklad 4.1: JSON objekt obsahující zpracované informace o zdrojových textech.

4.2 Generování unikátních dvojic

Ve druhé fázi se využije JSON objekt předaný parametrem nebo vytvořený v první fázi. Pomocí metody *getUniquePairs($assignments, $templates = NULL)*, která implementuje vytvoření unikátních dvojic zdrojových souborů jako kombinace vstupní soubory – vstupní soubory, případně i vstupní soubory – šablony. V případě, že se v průběhu generování dvojic narazí na soubory se stejným názvem jsou tyto v případě, že oba pocházejí ze vstupního JSON objektu přeskočeny.

Následně jsou vygenerované dvojice uloženy pomocí metod na práci se soubory implementovaných ve třídě *FileUtils.php*. Ukázku vygenerovaného souboru lze vidět na příkladu 3.1. Tímto fáze dvě končí a v případě, že vstupními parametry nebylo specifikováno jinak, pokračuje aplikace k již samotnému porovnávání zdrojových kódů.

4.3 Implementace vyhledávání plagiátu

V návrhu je vyhledávání plagiátů rozděleno do dvou fází, což se odráží i na implementaci samotného programu, nicméně implementace těchto fází se liší pouze v přísnosti kontroly vyhledávání a použitých algoritmech, proto je v ní věnována souhrnně tato kapitola.

Nejprve je ovšem nutné vždy před samotným porovnáním vyhledat dvojici načtenou z CSV souboru. K tomu slouží v nástroji *ArrayUtils.php* implementovaná metoda *getAssignmentByName($firstAssignment, $secondAssignment, $enviroment)*, která je schopná v prostředí *$enviroment*, jenž obsahuje objekt se zdrojovými kódy vyhledávat právě zdrojové složky určené ke kontrole. Při vyhledávání se také využije přípona *-template*, která je přidána ke všem kódům, které pocházejí ze šablony. Ta nám zajistí, že se soubory s touto příponou budou vyhledávat pouze v šablonách a naopak soubory bez této přípony se budou vyhledávat pouze ve vstupním JSON objektu.

4.3.1 Hodnocení kódů pomocí metrik a Levenshteinova algoritmu

Pokud byly dané soubory nalezeny, je možné je začít porovnávat. Nejprve jsou zdrojové kódy hodnoceny podle Halsteadových metrik, kdy je pro každou metriku jednoho zdrojového textu spočítána procentuální podobnost se stejnou metrikou druhého zdrojového textu. Výsledky metrik z jednotlivých funkcí zdrojového kódu jsou sčítány a průměrovány tak, abychom ve výsledku dostali jednu hodnotu v procentech, značící do jaké míry jsou si soubory podle dané metriky podobné.

Výpočet podobnosti pro Levenshteinův algoritmus funguje na podobném principu. Levenshteinova vzdálenost je nejprve spočítána vždy pro dvojici bloků kódu, které lze vidět v příkladu 4.1. Jak již bylo zmíněno dříve, limit pro délku jednoho bloku je nastaven na 255 znaků. To znamená, že maximální vzdálenost mezi dvěma řetězci je 255 a minimální je 0. Pokud známe rozsah, lze z něj vypočítat procentuální podobnost mezi dvěma bloky. Výsledky z porovnání dvou bloků jsou poté sčítány a průměrovány podobně jako v případě Halsteadových metrik, což umožňuje získat jediný výsledek hodnotící podobnost dvou zdrojových kódu podle Levenshteinova algoritmu. Kromě získání podobnosti jsou taktéž uchovávány statistiky o výsledku porovnání jednotlivých bloků v případě, že překročí 90% mez podobnosti. Pokud jsou mezi dvěma zdrojovými soubory nalezeny více než tři takto podobné bloky jsou tyto automaticky označeny jako potenciální plagiáty.

Samotnou implementaci algoritmu pro výpočet Levenshteinovy vzdálenosti jsem se rozhodl přenechat na vestavěné funkci *levenshtein($stringA, $stringB)*. Je to za prvé z důvodu, že vestavěná funkce již nabízí veškeré optimalizace, které může jazyk PHP nabídnout a za druhé již sama omezuje maximální délku řetězce, který je možné porovnávat. Navíc tato funkce může být přetížena a je zde tak možné nastavit váhy jednotlivých operací přidání znaku, odebrání znaku nebo substituce.

4.3.2 Filtrování výsledků třetí fáze

Po porovnání všech párů jsou výsledky zpracovány a vyfiltrovány. Během zpracování výsledků je třeba brát v potaz, že většina zdrojových kódů se nejčastěji skládá z tokenů jako jsou například proměnné nebo textové řetězce. Navíc v případě využití tohoto nástroje na školních projektech se podobnost mezi zdrojovými kódy ještě zvyšuje například kvůli zpracovávání stejných vstupních parametrů programu nebo používání stejných regulárních výrazů, které odpovídají zadaní studentské práce. Z tohoto důvodu byl práh, která rozhoduje o tom, zda bude daný soubor dále prozkoumáván nastaven na 70% v případě výsledků

Levenshteinova algoritmu a na 80% alespoň ve dvou metrikách Halsteadova algoritmu. Toto se samozřejmě netýká prací, které obsahují tří a více bloků kódu, u kterých byla pomoci Levenshteinova algoritmu nalezena více než 90% podobnost. Tyto kombinace zdrojových kódu postupují do čtvrté fáze programu automaticky.

Ke snížení spotřeby strojového času ve čtvrté fázi tohoto nástroje se už nebudou porovnávat zdrojové kódy každý s každým, ale pouze páry, které čelí podezření z plagiátorství již z výsledné třetí fáze programu.

4.3.3 Detailní vyhledávání pomocí algoritmu Winnowing

Čtvrtá fáze programu je implementována velmi obdobně jako předchozí. Nejprve se vyhledají páry zdrojových kódů k porovnání a následně je z celé posloupnosti tokenů vytvořen otisk zdrojového kódu programu pomocí algoritmu Winnowing, jehož implementaci lze vidět na příkladu 4.2.

```
1  public function Winnowing($w) {
2    $window = array_fill(0, $w, NULL);
3    for ($i = 0; $i < $w; ++$i) $window[$i] = self::INT_MAX;
4    $windowRightEnd = 0;
5    $minHashIndex = 0;
6    // At the end of each iteration, min holds the position
7    // of the rightmost minimal has in the current window.
8    while(true) {
9      // shift the window by one
10     $windowRightEnd = ($windowRightEnd + 1) % $w;
11     $window[$windowRightEnd] = getNextHash();
12     if ($window[$windowRightEnd] == -1)
13       break;
14     if ($minHashIndex == $windowRightEnd) {
15       // The previous minimum is no longer in this window.
16       // Scan window leftward starting from window right end for
17       // the rightmost minimal hash.
18       for ($i = ($windowRightEnd - 1) % $w; $i != $windowRightEnd;
19            $i = ($i - 1 + $w) % $w)
20         if ($window[$i] < $window[$minHashIndex]
21           $minHashIndex = $i;
22       record($window[$minHashIndex],
23              global_pos($minHashIndex, $windowRightEnd, $w);
24     } else {
25       // Otherwise, the previous minimum is still in this window.
26       // Compare againts the new value and update min if necessary.
27       if ($window[$windowRightEnd] <= $window[$minHashIndex] {
28         $minHashIndex = $windowRightEnd;
29         record($window($minHashIndex],
30                global_pos($minHashIndex, $windowRightEnd, $w);
31       }
32     }
33   }
34 }
```

Příklad 4.2: Implementace algoritmu Winnowin v jazyce PHP.

Tato implementace algoritmu vytvoří unikátní otisky obou zdrojových kódů z páru a umožní je následně mezi sebou porovnat. Jako hašovací funkce, která byla potřebná k vytvoření otisků byla použita vestavěná funkce *md5*, která jak je vidět podle jejího názvu

využívá k hašování algoritmus MD5[2].

Na rozdíl od implementace porovnávání ve třetí fázi se zde výsledky nijak netransformují do výsledných procent značících úroveň podobnosti mezi dvěma zdrojovými texty. Pouze se vždy v páru vyhledávají stejné hodnoty hašů a počet stejných hašů se ukládá do paměti. V případě, že v kódech byly zaznamenány alespoň tři haše sdílené mezi oběma zdrojovými kódy jsou tyto označeny jako možný plagiát.

Kromě toho, že vyhledávání plagiátů pomocí otisků je více důkladnější než stejné vyhledávání pomocí metod použitých ve třetí fázi, je nesmírnou výhodou tohoto přístupu fakt, že v rámci algoritmu Winnowing lze velice snadno u otisků uchovávat i jejich přibližnou polohu ve zdrojovém kódu a v případě podezření z plagiátorství uživateli označit řádky, které se zdají být podobné.

Po prozkoumání veškerých zdrojových textů jsou výsledky z této fáze uloženy do souboru opět ve formátu CSV, tentokrát s rozšířenými informacemi ohledně podezřelých dvojic zdrojových kódů. Na příkladu níže 4.3 lze vidět ukázku několika řádků ve výsledném CSV souboru. Jeden řádek ve výstupním CSV souboru odpovídá páru, u kterého vzniklo podezření na plagiátorství. Tento řádek je poté uložen ve formátu, kdy první dvě hodnoty odpovídají názvům kontrolovaných souborů, následuje jejich procentuální podobnost a poté případně hrubá pozice kódu v souboru, který je společný pro oba zdrojové soubory.

```
1 xlogin00,xlogin01,81%,[301][503],[312][514]
2 xlogin02,xlogin07,68%,[12][703],[17][708],[23][714],[30][721]
3 xlogin03,xlogin05,83%,[231][107],[320][541]
4 xlogin31,xlogin17,54%,[456][123],[426][135],[432][325],[12][758]
```

Příklad 4.3: Ukázka výstupního souboru s podezřelými dvojicemi zdrojových kódů.

Výsledky byly ukládány tak, aby byly snadno čitelné i pro člověka a zároveň je bylo možné snadno exportovat do dalších formátů případně do dalších externích programů.

4.4 Testování a experimenty

4.4.1 Rozšíření do budoucnosti

[2]Více o algoritmu na http://tools.ietf.org/html/rfc1321

Kapitola 5

Závěr

Literatura

[1] B. Zeidman: *The Software IP Detective's Handbook: Measurement, Comparison, and Infringement Detection*. Prentice Hall, 2011, iSBN 978-0-137-03533-5.

[2] Clough, P.: Plagiarism in natural and programming languages: an overview of current tools and technologies. `http://ir.shef.ac.uk/cloughie/papers/plagiarism2000.pdf`, 2000-06-01 [cit. 2015-05.01].

[3] G. Valiente: *Algorithms on Trees and Graphs*. Springer, 2002, iSBN 3-5404-3550-6.

[4] J. A. W. Faidhi, S. K. R.: An empirical approach for detecting program similarity and plagiarism within a university programming environment. Computers and Education, 11(1), 11-19, 1986-05-16 [cit. 2015-04-30].

[5] L. Bergroth, H. Hakonen, T. Raita: *A Survey of Longest Common Subsequence Algorithms*. SPIRE, 2000, iSBN 0-7695-0746-8.

[6] Levenshtein, V. I.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady, 10(8), 707-710, 1966 [cit. 2015-05-02].

[7] M. H. Halstead: *Elements of Software Science (Operating and programming systems series)*. Elsevier Science, 1977, iSBN 0-4440-0205-7.

[8] R. Pecinovský: *Návrhové vzory*. Computer Press, 2007, iSBN 8-0251-1582-4.

[9] Schleimer, D. S. W. S.; Aiken, A.: Winnowing: local algorithms for document fingerprinting. Proceedings of the 2003 ACM SIGMOD international conference on Management of Data, 76-85, 2003 [cit. 2015-05-03].

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein.: *Introduction to Algorithms*. MIT Press, 2009, iSBN 978-0-262-03384-8.

[11] WWW stránky: CodeMatch. `http://www.safe-corp.biz/products codematch.htm`.

[12] WWW stránky: Moss : A System for Detecting Software Plagiarism. `http://theory.stanford.edu/ aiken/moss/`.

[13] WWW stránky: PHP: Hypertext Preprocessor. `http://www.php.net`.

[14] WWW stránky: The Sherlock Plagiarism Detector. `http://sydney.edu.au/engineering/it/ scilect/sherlock/`.

[15] Zeidman, B.: What, Exactly, Is Software Plagiarism? http://www.iptoday.com/pdf/2007/2/Zeidman-Feb2007.pdf, 2007-02-01 [cit. 2015-04-30].