UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Marko Lugarič

PODOBNOST IZVORNE KODE

DIPLOMSKO DELO NA
UNIVERZITETNEM ŠTUDIJU

Mentor: doc. dr. Marko Robnik Šikonja

Ljubljana, 2008

ZAHVALA

Zahvaljujem se svojemu mentorju doc. dr. Marku Robniku Šikonji za pomoč in nasvete pri izdelavi diplomske naloge, svoji družini pa za vso podporo med časom študija.

KAZALO

POVZETEK

V diplomski nalogi je opisan pregled problematike, povezan z detekcijo plagiarizirane izvorne kode, ki se pojavlja pri seminarskih nalogah v obliki programov. Namen dela je opisati trenutno najbolj uspešne rešitve, jih oceniti, kako se obnesejo v praksi, nato pa izdelati novo metodo za detektiranje podobnosti izvorne kode. Metoda, ki sem jo izdelal, temelji na idejah in predstavitvah podatkov, ki se uporabljajo pri strojnem učenju tekstovnih podatkov. Vsak program se predstavi s številom posameznih ključnih besed nekega programskega jezika. Dodamo atribute, ki predstavljajo število in vrsto spremenljivk v programu. Na tako predstavljenih programih se izvede rojenje, ki opozori na potencialno plagiarizirane programe. Za testiranje in izboljševanje metode sem izdelal označeno bazo, s katero sem utežil posamezne atribute. Metodo sem testiral na praktičnih primerih seminarskih nalog. Tako pridobljene rezultate sem analiziral ter predlagal možne izboljšave metode.

KLJUČNE BESEDE: podobnost izvorne kode, plagiat, rojenje, konstrukcija atributov, strojno učenje

ABSTRACT

We propose a detection method for plagiarised source code in programs written by students. The purpose of this work is to present state of-the-art solutions, evaluate them and then construct a new method for detection of plagiarism in source code. Method constructed uses the ideas and data representation found in text mining. Every program is presented with keywords from a program. The number of variables and their types are included as attributes. The method extracts subsets of programs (the clusters of plagiarism) such that each program within a particular subset has been derived from the same original. A set of plagiarized programming assignments is created to test and improve our method. Finally the method is tested on real programming assignments. The results are analysed and improvements are suggested.

KEYWORDS: source code similarity, plagirism, clustering, attribute construction, data mining

1. UVOD

Znano je, da vsi študenti ne opravljajo svojih obveznosti na povsem pošten način [1]. Goljufanje pri pisanju seminarskih nalog v obliki programov je eden od nepoštenih načinov. Zato je pomembno tudi odkrivanje programov z zelo podobno izvorno kodo (plagiatov). Plagiat bi lahko opisali kot izvorno kodo programa, ki je skopirana, vendar modificirana z raznimi spremembami. Plagiator ponavadi ne pozna dobro delovanja programa. Ročno odkrivanje plagiatov je časovno zelo potratno, saj moramo za pregled izvirnosti n programov pregledati n * (n – 1) / 2 parov izvornih kod. Zato se je že sredi sedemdesetih let pojavila prva programska oprema, ki je avtomatizirala odkrivanje plagiatov. Do danes je bilo razvitih nekaj programov, od katerih sta dva precej razširjena v akademski sferi in ju bom v nadaljevanju opisal.

V grobem sisteme za odkrivanje plagiatov delimo na tiste, ki štejejo atribute ter strukturno metrične sisteme. Primer enostavnega številno atributnega sistema je štetje števila posameznih operatorjev, števila operandov, števila različnih operatorjev ter števila različnih operandov. Programi z zelo podobnimi temi štirimi števili so označeni kot sumljivi. Boljši atributni sistemi štejejo tudi število vrstic, število deklariranih spremenljivk, prireditvenih stavkov, pogojnih stavkov, število zank, število podprogramov, število klicev podprogramov, število ključnih besed, itd. Kot boljši so se v praksi izkazali strukturno metrični sistemi. Primeri teh sistemov so JPlag, MOSS, YAP, Plague. Ti sistemi imajo dve fazi: najprej iz izvorne kode z leksikalnim analizatorjem dobijo neko zaporedje znakovnih nizov, nato pa na nek način primerjajo ta zaporedja znakovnih nizov.

Naš pristop je, da na podlagi idej in predstavitev podatkov, ki se uporabljajo pri strojnem učenju tekstovnih podatkov [2], razvijem in testiram drugačno metodo za odkrivanje podobnosti izvorne kode. Najprej sem vsak program predstavil z atributi v obliki vreče besed ter dodal še različne atribute, ki vsebujejo informacije o lastnostih programa, npr. številu in vrsti spremenljivk, dolžini zank, strukturi ugnezdenih zank, grafu klicev, bločni strukturi. Te atribute sem nato preizkusil na zbirkah izvorne kode v programskem jeziku java, ki so jih v različnih semestrih oddajali študenti pri laboratorijskih vajah predmetov Algoritmi in podatkovne strukture in Osnove algoritmov in podatkovnih struktur. S tehniko rojenja (angl. clustering) sem sestavil roje, v katerih so posamezni študenti. Roji z več kot enim študentom predstavljajo sumljive primere podobnosti (prepisovanja) med študenti v roju.

Naj omenim še, da vsi programi za odkrivanje plagiatov vedno vrnejo nek seznam sumljivo podobnih primerov, dokončno odločitev, ali gre resnično za plagiat ali ne, pa še vedno naredi učitelj, ki ročno pregleda sumljive primere.

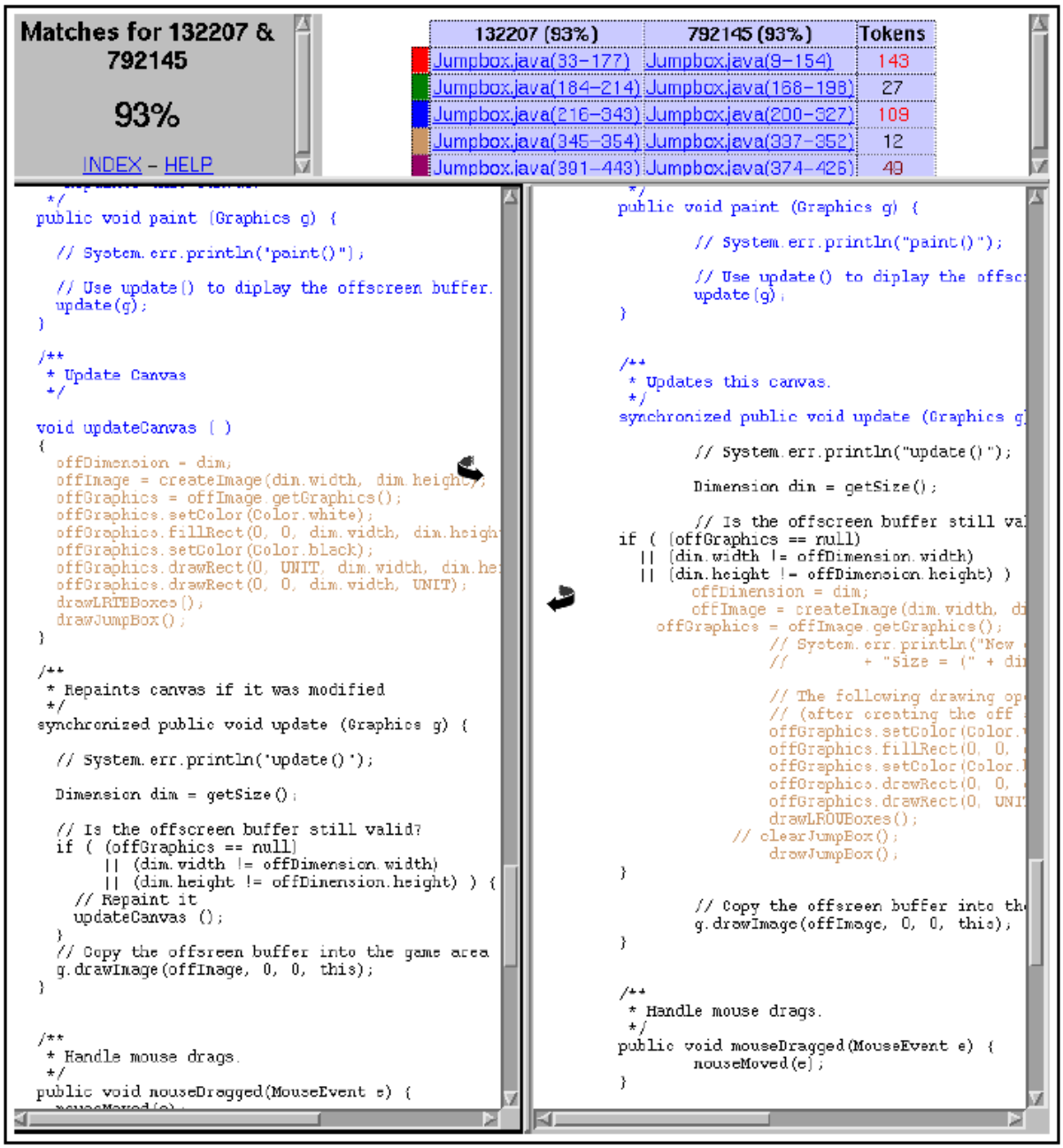
Diplomsko delo je razdeljeno na 5 poglavij. Uvodnemu opisu predmeta in namena dela sledi v drugem poglavju prikaz obstoječih rešitev za odkrivanje podobnosti izvorne kode. Največ pozornosti sem namenil tretjemu poglavju, v katerem je prikazana metoda za iskanje podobnosti izvorne kode, ki sem jo sam izdelal, in četrtemu poglavju, v katerem je prikazano testiranje metode na označeni bazi in na realnih primerih. V zaključnem poglavju je povzet opis opravljenega dela in ideje za izboljšanje.

2. OBSTOJEČE REŠITVE ZA ODKRIVANJE PODOBNOSTI IZVORNE KODE

Poglejmo si najprej obstoječe rešitve za odkrivanje podobnosti izvorne kode in njihove lastnosti.

2.1 JPLAG

JPlag [11] je sistem za ugotavljanje podobnosti med izvornimi datotekami. Podpira programske jezike java, C#, C++, scheme in navadni tekst. Tipično se uporablja za ugotavljanje plagiatorstva pri seminarskih nalogah v obliki programov, lahko pa se uporablja tudi pri detekciji ukradene ali kopirane izvorne kode. Ima lep prikaz rezultatov v zapisu html. Napisan je v javi ter je na spletu dostopen brezplačno ob predhodni registraciji na spletnih straneh, ki sistem dajejo na razpolago zainteresiranim. S klientom, ki ga zaženemo na spletni strani JPlaga, se najprej prijavimo v sistem, nato pa izberemo izvorne kode programov, ki bi jih radi primerjali. Ti se nato pošljejo na JPlagov strežnik, kjer jih primerjalni algoritem primerja vsakega z vsakim. Kot izhod dobimo html strani, kjer JPlag po vrsti prikaže najbolj podobne pare izvorne kode. Vsak podoben par si lahko tudi ogledamo, saj nam JPlag posebej pokaže obe izvorni kodi ter barvno označi tiste dele kode, ki so zelo sumljivo podobni (slika 2.1).



Slika 2.1: Primer prikaza sumljivih delov kode v html formatu kot ga izdela JPlag. Z modro in rjavo sta pobarvana sumljiva dela kode.

2.1.1 JPLAGOV PRIMERJALNI ALGORITEM

JPlagov primerjalni algoritem [3] je sestavljen iz dveh korakov, in sicer:

- Vsa izvorna koda programov se najprej sintaktično preveri, nato pa preoblikuje v zaporedje vnaprej definiranih nizov.

- Znakovni nizi, ki smo jih dobili v prvem koraku, se v parih primerjajo med saboj.

Metoda se v imenuje »Greedy String Tilling«. Med vsako izmed primerjav poskuša JPlag pokriti enega izmed generiranih znakovnih nizov z generiranimi znakovnimi podnizi iz druge datoteke v največji možni meri. Procent znakovnih nizov, ki jih je mogoče pokriti na ta način, je vrednost podobnosti. Najdeni deli so nato prikazani in ustrezno barvno označeni v izhodni datoteki.

2.1.2 SPREMINJANJE IZVORNE KODE V ZNAKOVNE NIZE

Spreminjanje programov v znakovne nize je edini del, ki je odvisen od programskega jezika. Za programski jezik java in scheme je implementiran celoten sintaksni analizator, medtem ko je za C in C++ implementiran le leksikalni analizator. Znakovni nizi v katere se prevedejo programi, morajo biti izbrani tako, da izražajo strukturo programa, saj jih je na ta način težko spremeniti, npr. while(System.in.read() != -1) se spremeni v APPLY, BEGIN_WHILE znakovni niz. Vsi presledki, komentarji in imena identifikatorjev se ignorirajo, dodajo pa se semantične informacije, kjer je to možno, saj na ta način zmanjšamo število naključnih ujemanj. Zato se npr. v programskem jeziku java prvi znak "{" takoj za deklaracijo metode, zamenja z nizom BEGIN_METHOD. Na sliki 2.2 je prikazano spreminjanje izvorne kode v znakovne nize za kratek javanski razred.

	Java source code	Generated tokens
1	`public class Count {`	BEGIN_CLASS
2	`  public static void main(String[] args)`	VAR_DEF, BEGIN_METHOD
3	`            throws java.io.IOException {`	
4	`    int count = 0;`	VAR_DEF, ASSIGN
5		
6	`    while (System.in.read() != -1)`	APPLY, BEGIN_WHILE
7	`      count++;`	ASSIGN, END_WHILE
8	`    System.out.println(count+" chars.");`	APPLY
9	`  }`	END_METHOD
10	`}`	END_CLASS

Slika 2.2: Primer kode v programskem jeziku java in pripadajoči znakovni nizi.

2.1.3 PRIMERJANJE ZAPOREDJA NIZOV

Za primerjanje dveh zaporedij znakovnih nizov se uporablja algoritem »Greedy String Tiling« (slika 2.3). Pri primerjavi dveh zaporedij nizov A in B algoritem išče maksimalen podniz z naslednjimi lastnostmi: podniz se nahaja v A in B, podniz je dolg kolikor je mogoče, podniz se ne prekriva s katerim že predhodno najdenim podnizom. Da se izognemo zelo kratkim podnizom, mora biti vsak podniz dolg najmanj M nizov. Ta algoritem je hevrističen, saj bi iskanje optimalne maksimalne množice trajalo predolgo. Psevdokoda za algoritem se nahaja na sliki 2.3.

```
0   Greedy-String-Tiling(String A, String B) {
1       tiles = {};
2       do {
3           maxmatch = M;
4           matches = {};
5           Forall unmarked tokens A_a in A {
6               Forall unmarked tokens B_b in B {
7                   j = 0;
8                   while (A_{a+j} == B_{b+j} &&
9                           unmarked(A_{a+j}) && unmarked(B_{b+j}))
10                      j++;
11                  if (j == maxmatch)
12                      matches = matches ⊕ match(a, b, j);
13                  else if (j > maxmatch) {
14                      matches = {match(a, b, j)};
15                      maxmatch = j;
16                  }
17              }
18          }
19          Forall match(a, b, maxmatch) ∈ matches {
20              For j = 0 ... (maxmatch − 1) {
21                  mark(A_{a+j});
22                  mark(B_{b+j});
23              }
24              tiles = tiles ∪ match(a, b, maxmatch);
25          }
26      } while (maxmatch > M);
27      return tiles;
28  }
```

Slika 2.3: Psevdokoda za algoritem Greedy String Tiling. Operator v vrstici 12 doda podniz v množico podnizov, če se trenutni podniz ne prekriva s katerim podnizom v množici podnizov. Trojček (a, b, l) predstavlja podniz, ki se začenja v programu A na mestu a, v programu B na mestu b ter je dolg dolžine l.

Algoritem v do-while zanki ponavlja naslednja dva koraka:

- Prvi korak (vrstice 5-18): V obeh zaporedjih iščemo najdaljša ujemanja. To naredimo s tremi vgnezdenimi zankami. V prvi gremo skozi vse znakovne nize v A, v drugi primerjamo trenuten znakovni niz v A z vsemi znakovnimi nizi v B. Ko najdemo ujemanje, se najbolj notranja zanka sprehaja po nizih, ki sledijo trenutnemu nizu v A in B in najde konec ujemanja. Na ta način dobimo množico najdaljših skupnih podnizov v obeh programih A in B.

- Drugi korak (vrstice 19-25): Označimo vsa zaporedja nizov, ki se ne prekrivajo. To pomeni, da se označijo vsi pripadajoči znakovni nizi teh podnizov, ki jih v nadaljnjem iskanju ujemanj več ne upoštevamo.

Ta dva koraka se ponavljata, dokler ne najdemo nobenega ujemanja več. Končni rezultat so zaporedja označenih nizov, ki jih uporabimo pri računanju podobnosti programov A in B ter prikazu istih ali modificiranih delov kode. V najslabšem primeru je časovna zahtevnost O((A + B)^3), kar lahko s pomočjo ideje algoritma Rabin-Karp [4] še izboljšamo na O(A + B).

2.1.4 JPLAG V PRAKSI

V praksi velja JPlag za enega najboljših detektorjev plagiatov, kar se je pokazalo tudi na testnih primerih, ter na bazi primerov, ki sem jo zgradil. Uspešno zavajanje JPlaga bi zahtevalo precej dela, saj bi bilo treba spremeniti celoten program na več mestih. Zelo uporabna je tudi možnost, da mu podamo izvorne datoteke, ki jih dobijo vsi študenti na razpolago za lažjo implementacijo. Tem datotekam JPlag pravi »base files« in jih ne uporablja pri primerjavi. Na ta način kot rezultat ne dobimo označene izvorne kode, ki je pri vseh študentih enaka.

2.2 MOSS

MOSS [12] je eden od uspešnejših sistemov za odkrivanje podobnosti izvorne kode narejen leta 1997. MOSS je kratica za »Measure of software similarity« in je primarno namenjen za odkrivanje plagiatorstva pri seminarskih nalogah v obliki programov. Zaradi svoje zasnove lahko primerja med sabo zelo veliko programskih jezikov. Trenutno podprti so: C, C++, java, C#, python, Visual Basic, javascript, fortran, ML, haskell, lisp, scheme, pascal, modula2, ada, perl, TCL, matlab, VHDL, verilog, spice, MIPS assembly, a8086 assembly, HCL2.

Deluje tako, da s skripto pošljemo na MOSS strežnik izvorne datoteke, ki bi jih radi primerjali. Jedro sistema je algoritem winnowing [4] na strežniku, s katerim iz vsake izvorne datoteke pridobimo tako imenovane prstne odtise. Pari izvornih kod programov, ki imajo največ skupnih prstnih odtisov, so posredovani uporabniku v obliki html strani, kjer so barvno označeni deli kode, ki so sumljivo podobni.

2.2.1 ALGORITEM WINNOWING

S tem algoritmom dobimo tako imenovane prstne odtise dokumenta, v našem primeru izvorne kode programa. Algoritem je sestavljen iz naslednjih korakov:

- Prvi korak: Najprej odstranimo nepomembne stvari za algoritem, kot so presledki, znaki za novo vrstico, znaki, ki niso črke. Vse črke pretvorimo v male črke. Npr. *public void run()* se pretvori v *publicvoidrun*.

- Drugi korak: Takšna koda se nato pretvori v k-grame, kjer je k konstanta, ki jo izbere uporabnik. Npr. pri k = 6 bi iz *publicvoidrun* dobili zaporedje k-gramov *public, ublicv, blicvo, licvoi, icvoid, cvoidr, voidru, oidrun.* K-gramov v programu je približno toliko, kot ima program znakov.

- Tretji korak: Za vsak k-gram izračunamo vrednost razpršilne funkcije. Tako dobimo za celoten izvoren program neko zaporedje števil, npr. iz k-gramov *public, ublicv, blicvo, licvoi, icvoid, cvoidr, voidru, oidrun* bi dobili *77 72 42 17 98 50 19 90*. Razpršilna funkcija je izbrana tako, da je možnost kolizije čim manjša. V tem primeru ista številka v dveh dokumentih zelo gotovo pomeni isti niz.

- Četrti korak: Iz takega zaporedja števil se izbere neko podmnožico števil, ki predstavljajo prstni odtis dokumenta. Enostaven način za to je, da izberemo števila, ki imajo ostanek 0 pri deljenju z nekim fiksnim številom p (0 mod p). Pri algoritmu winnoving podmnožico števil izberemo na naslednji način: najprej določimo neko število w, ki nam določa širino okna. V tem oknu imamo w števil. Celotno zaporedje števil iz tretjega koraka zapišemo v zaporedja oken. Iz zaporedja *77 72 42 17 98 50 19 90* bi dobili pri širini okna w = 3 zaporedje oken: *(77, 72 , 42), (72, 42, 17), (42, 17, 98), (17, 98, 50), (98, 50 ,19), (50, 19, 90)*.

- Peti korak: Iz vsakega okna izberemo najmanjše število. Števila ne izberemo, če je to najmanjše število enako kot najmanjše število v prejšnjem oknu. Izjema pri tem je le, če je novo število (najbolj desno) prav tako najmanjše število. Zaporedju tako izbranih števil rečemo prstni odtis dokumenta. Npr. iz zaporedja oken *(77, 72 , 42), (72, 42, 17), (42, 17, 98), (17, 98, 50), (98, 50 ,19), (50, 19, 90)* bi dobili prstni odtis *42, 17, 19*.

Algoritem na ta način najde vsa podzaporedja v izvorni kodi, ki so manjša ali enaka številu w + k – 1. Izbira konstant w in k vpliva na hitrost in natančnost prepoznavanja podobnosti. Konstanta k je meja za šum; zaporedij, krajših od k znakov ne prepoznamo. Konstanta w vpliva na prostorsko in posledično časovno zahtevnost; večji kot je w, manj prstnih odtisov bo imel dokument. Večji w pomeni slabše prepoznavanje podobnosti, tako da moramo izbrati ustrezen kompromis.

2.2.2 PRIMERJAVA DVEH IZVORNIH DATOTEK

MOSS si ob izračunu vsakega prstnega odtisa zapomni tudi njegovo lokacijo; v kateri izvorni datoteki se nahaja ter v kateri vrstici kode je. V prvem koraku zgradi indeks iz vseh izvornih datotek, ki preslika prstni odtis v lokacije. To pomeni, da dobimo prstne odtise iz vseh datotek. Na podoben način deluje invertiran indeks [14] pri spletnih iskalnikih. V drugem koraku dobi iz posamezne izvorne datoteke d prstne odtise in nato

pogleda v indeksu, če so v več kot eni datoteki. Če so, so dodani v listo ujemajočih se prstnih odtisov izvorne datoteke d. Vsak ujemajoč se prstni odtis v izvorni datoteki d je lahko v drugih izvornih datotekah d1, d2... Listo ujemajočih se prstnih odtisov vsake izvorne datoteke sortiramo. Iz tega ustvarimo pare izvornih datotek (d, d1), (d, d2), itd. s skupnimi prstnimi odtisi. Pari z največjim številom prstnih odtisov se vrnejo kot rezultat uporabniku. Za boljši prikaz se barvno označi sumljiva koda. Ena od dobrih strani MOSSa je, da mu lahko posredujemo dele kode, ki naj jih ne upošteva pri primerjavi kot npr. določeni razredi v javi, ki jih napiše inštruktor in jih uporabljajo vsi učenci.

2.2.3 MOSS V PRAKSI

MOSS se je pri mojih testiranjih izkazal kot najboljša rešitev, saj je odkril največ plagiatov. Dobro mu je uspelo prepoznati celo datoteke, ki so bile le delno kopirane, ostali del izvorne kode pa je uporabnik verjetno napisal sam.

2.3 OSTALE REŠITVE

Poleg že opisanih sistemov obstajajo tudi drugi sistemi za prepoznavanje podobnosti izvorne kode, vendar se njihova uspešnost v praksi ne more kosati s sistemoma JPLag in MOSS. Delujejo po zelo različnih principih. Kot primer navajam Pdetect [5], ki šteje ključne besede v posameznih programih, na podlagi katerih izračuna podobnosti med vsemi pari izvornih datotek, nato na podlagi tega generira graf, ki služi za rojenje. Nekateri programi poskušajo s pomočjo kompresije [6] izračunati podobnost, saj naj bi ista programa imela večjo stopnjo kompresije. Ideje so zanimive, vendar v praksi rezultati ne dajejo tako dobrih rezultatov kot JPlag in MOSS.

3. ISKANJE PODOBNOSTI V IZVORNI KODI

Moja metoda za iskanje podobnosti izvorne kode, ki jo bom podrobneje predstavil v nadaljevanju, je potekala v naslednjih dveh korakih:

- Najprej sem vsak program oziroma vse njegove izvorne datoteke predstavil z vrečo besed (angl. bag-of-words) ter ji dodal še atribute, ki so predstavljali informacijo o spremenljivkah v programu. Na ta način sem vsakega študenta, oziroma njegov program predstavil kot en primer, v kateri ima vsak atribut neko številsko vrednost. Končni rezultat tega koraka je datoteka z vsemi študenti.

- Datoteko, pridobljeno v prvem koraku, sem uvozil v programski paket Weka [10], namenjen podatkovnemu rudarjenju (angl. data mining). Z metodo rojenja sem nato pridobil različne roje. Če so roji vsebovali več kot enega študenta, je to sumljiv primer.

3.1 PRIDOBIVANJE ATRIBUTOV

Za predstavitev neke izvorne datoteke z vrečo besed sem prebral izvorno datoteko ter preštel javanske ključne besede. To sem storil za vse izvorne datoteke programa. Kot dodatne atribute sem preštel še spremenljivke in njihove tipe v vseh izvornih datotekah programa. Za ključne besede sem se odločil, ker se njihovo število v izvorni datoteki ohranja kljub spremembi imena spremenljivk, zamenjavi vrstnega reda stavkov, spremembam vizualne podobe z dodajanjem presledkov, tabulatorjev, spremembam komentarjev, menjavam imen metod, razredov in spremembi konstantnih vrednosti. Obstajajo seveda tudi spremembe, ki spremenijo število ključnih besed, kot npr. sprememba modifikatorja dostopa, menjava for zanke z while ali obratno, menjava switch pogojnega stavka z zaporedjem if stavkov, brisanje kratkih metod (angl. inlining), sprememba dela kode v novo metodo, združevanje pri deklaracijah istih spremenljivk, itd. Število in vrsta spremenljivke kot posamezen atribut pa se mi je zdel naslednji primeren atribut, ki bi lahko izboljšal detekcijo.

3.1.1 ŠTETJE KLJUČNIH BESED

Pri štetju sem štel vse javanske ključne besede: *abstract, assert, boolean, break, byte, case, catch, char, class, continue, default, do, double, else, enum, extends, final, finally, float, for, if, implements, import, instanceof, int, interface, long, native, new, package, private, protected, public, return, short, static, strictfp, super, switch, synchronized, this, throw, throws, transient, try, void, volatile, while.*

Najprej sem se lotil štetja z java Pattern in Matcher razredoma, s katerima sem primerjal izvorno kodo z nekim regularnim izrazom, v katerem je bila ključna beseda. Ta princip

se ni preveč dobro obnesel, saj je štel tudi ključne besede v komentarjih ter znakovnih nizih, kar seveda niso ključne besede. Zato sem se lotil izdelave leksikalnega analizatorja za programski jezik java, s katerim bi natančno preštel število ključnih besed. Ta bi poleg tega, da razbije izvorno kodo javanskega programa na osnovne simbole (angl. token), štel tudi ključne besede. Ker izdelava leksikalnega analizatorja za nek programski jezik vzame kar nekaj časa, so se že v preteklosti pojavili generatorji izvorne kode za leksikalni analizator. Kot vhod ti generatorji sprejmejo regularne izraze, ki predstavljajo osnovne simbole. Osnovni simboli so: ključne besede, operatorji, komentarji, številske konstante, znakovne konstante itd. Na osnovi teh regularnih izrazov nato generatorji zgradijo leksikalni analizator.

Uporabil sem orodje JFlex [8], ki izdela izvorno kodo v programskem jeziku java. JFlex vzame kot vhod datoteko v .lex formatu, v kateri so regularni izrazi in morebitna uporabnikova koda. Primer regularnega izraza za celo število v javi: *[1-9][0-9]*[lL]? | 0[0-7]*[lL]? | 0[xX][0-9a-fA-F]*[lL]?* . Večino regularnih izrazov sem našel na internetu, saj so nekateri precej zapleteni. Kot uporabnikovo kodo sem dodal spremenljivke za ključna števila, ki so se ob vsaki ključni besedi povečala za ena.

3.1.2 ŠTEVILO IN VRSTA SPREMENLJIVK

Za pridobivanje števila in vrste spremenljivk je potrebno narediti sintaksno analizo (angl. syntactic analysis). To je druga faza pri prevajanju izvorne kode programa v izvršilni program. Programu, ki to naredi, pravimo sintaksni analizator (angl. parser). Kot vhod dobi osnovne simbole iz leksikalnega analizatorja. V primeru, da je program sintaktično pravilen, sintaksni analizator kot izhod vrne drevo izpeljav, ki se nato uporabi v naslednjih fazah prevajanja. V primeru, da program ni pravilen, sintaksni analizator vrne sporočilo o napaki.

Ker je sintaksni analizator precej zapleten, obstajajo programi, ki generirajo izvorno kodo zanj. Kot vhod ti programi dobijo gramatiko programskega jezika (slika 3.1): končne simbole, nekončne simbole, produkcije ter začetni simbol. Na tej osnovi generirajo izvorno kodo za sintaksni analizator. Uporabil sem program JavaCUP [9], ki generira sintaksni analizator v programskem jeziku java. Celotno gramatiko za verzijo jave 1.4 sem dobil na internetu [13] ter jo vpisal v datoteko s .cup končnico. Nato sem dodal del uporabniške kode, tako da sem povezal leksikalni analizator, ki sem ga izdelal pred tem. Na ta način sem lahko preveril sintaksno pravilnost vhodnih programov.

```
// 19.4) Types, Values, and Variables
type    ::= primitive_type
    |   reference_type
    ;
primitive_type ::=
        numeric_type
    |   BOOLEAN     {: varType = "booleanVar"; :}
    ;
numeric_type::= integral_type
    |   floating_point_type
    ;
integral_type ::=
        BYTE    {: varType = "byteVar"; :}
    |   SHORT   {: varType = "shortVar"; :}
    |   INT     {: varType = "intVar"; :}
    |   LONG    {: varType = "longVar"; :}
    |   CHAR    {: varType = "charVar"; :}
    ;
floating_point_type ::=
        FLOAT   {: varType = "floatVar"; :}
    |   DOUBLE  {: varType = "doubleVar"; :}
    ;
```

Slika 3.1: Prikaz petih produkcij v datoteki .cup za javo 1.4. Z malimi črkami so označeni nekončni simboli, z velikimi pa končni simboli. Med oklepaji je uporabniška koda. Iz slike je razvidno, da se v javi tip lahko razvije v primitivni tip (boolean, byte, short, int, long, char, float, double) ali pa v referenčni tip.

Ker sem želel dobiti število ter vrsto spremenljivk, sem moral pri določenih produkcijah, povezanih z deklaracijami, dodati kodo za štetje. Gre za produkcije lokalnih spremenljivk, spremenljivk razreda ter parametrov v metodi ali konstruktorju. Pri vsaki od teh produkcij je bilo treba pogledati, kakšnega tipa je spremenljivka. Se pravi, ali gre za nek osnovni tip ali razred. Poleg tega pa sem preveril, ali ni spremenljivka deklarirana kot tabela. Na ta način sem dobil za vsako izvorno datoteko neko množico parov (tip, število). To sem ponovil za vse izvorne datoteke nekega študenta, s čimer sem dobil množico parov za vse datoteke. Nato sem izdelal atribute iz vseh množic vseh študentov. Za imena atributov sem uporabil imena tipov. V kolikor študent ni imel deklariranega tipa, je imel pri tem atributu vrednost 0.

Po nekaj testiranjih sem moral zgornji postopek dopolniti. Ker algoritem šteje vse spremenljivke, šteje tudi spremenljivke tistega tipa, ki jih definiramo sami, npr. deklariramo nek razred *Node,* v drugem delu pa uporabimo deklaracijo *Node a*. Ker so imena teh razredov verjetno ena prvih stvari, ki jih bo nek goljuf spremenil, je smiselno

odstraniti atribute, ki predstavljajo razrede, ki jih je deklariral uporabnik. Smiselno je ohraniti le tiste, ki jih študenti dobijo kot pomoč pri izdelavi programa. Na ta način se zmanjša število atributov, saj vsak uporabnik ponavadi deklarira kar nekaj svojih razredov.

3.1.3 UTEŽEVANJE ATRIBUTOV

Preden sem se na tako pripravljenih podatkih lotil rojenja, sem atribute utežil. To sem storil tako, da sem vrednosti posameznega atributa dodal meta informacijo o uteži. Pri izbiri atributov, ki sem jih utežil, sem želel utežiti atribute, ki se kljub raznim spremembam v izvorni kodi spreminjajo zelo malo ali pa nič. Primer takega atributa je npr. število zank v programu (seštevek for in while ključnih besed).

3.2 ROJENJE V PROGRAMSKEM PAKETU WEKA

Po pridobitvi vseh ključnih besed ter vrste in števila spremenljivk za vse študente sem le-te zapisal v format .csv (slika 3.2), ki ga sprejme Weka. S to datoteko sem v Weki začel postopek rojenja. Weka izračunane roje pripne kot atribut na koncu vsakega primera (slika 3.3). Na izbiro imamo več tipov rojenja, najbolj znana sta SimpleKMeans ter EM clustering. Določimo lahko tudi atribute, ki se ne upoštevajo. V našem primeru je to ime študenta oziroma njegova vpisna številka, saj ta atribut ne vpliva na uspešnost detekcije. Ker pri večini metod za rojenje kot parameter podamo število rojev, je vprašanje, kakšen K izberemo. Sam sem najprej uporabil K = število primerov – 1 ter nato testiral z vedno manjšim K, dokler ni bil približno K = 0.8 primerov. Ob takšnem zmanjševanju K se roji zelo malo spreminjajo, zato ni potrebno vedno pogledati vseh rojev. Ob vsakem rojenju s K roji je potrebno ročno preveriti, ali gre pri rojih z več kot enim študentom dejansko za goljufanje.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	imeStudenta	abstract	assert	boolean	break	byte	case	catch	char	class	continue	default	do	double
2	C1_original	13	0	4	3	0	0	1	0	4	0	0	0	
3	C1_plagiat1	13	0	4	3	0	0	1	0	4	0	0	0	
4	C1_plagiat5	13	0	4	3	0	0	1	0	4	0	0	0	
5	C2_original	13	0	7	2	0	0	2	0	6	0	0	0	
6	C2_plagiat1	13	0	7	2	0	0	2	0	6	0	0	0	
7	C3_original	13	0	8	0	0	0	1	0	8	0	0	0	
8	C4_original	0	0	6	1	0	0	4	0	6	0	0	3	
9	C4_plagiat1	0	0	5	0	0	0	4	0	6	0	0	3	
10	C5_original	14	0	9	0	0	0	1	1	8	0	0	0	
11	C6_original	13	0	6	0	0	0	1	0	6	0	0	0	
12	C7_original	0	0	2	1	0	0	2	0	7	0	0	0	
13	C8_original	26	0	8	1	0	0	0	2	7	0	0	1	
14	C8_plagiat1	26	0	8	1	0	0	1	2	7	0	0	0	
15	C9_original	13	0	6	8	0	0	2	0	5	0	0	0	
16	C10_original	13	0	9	0	0	0	6	8	9	0	0	0	
17	C11_original	13	0	7	0	0	0	1	0	6	0	0	0	
18	C12_original	13	0	7	2	0	0	2	1	7	0	0	0	
19	C13_original	13	0	4	2	0	0	0	1	3	2	0	0	
20	C13_plagiat1	13	0	4	2	0	0	0	1	3	2	0	0	
21	C14_original	13	0	17	2	0	0	7	3	13	1	0	0	
22	C15_original	12	0	7	3	0	0	1	0	7	0	0	0	
23	C16_original	0	0	2	0	0	0	0	0	2	0	0	0	
24	C17_original	13	0	6	1	0	0	1	1	5	0	0	0	

Slika 3.2: Prikaz .csv datoteke označene baze. V prvi vrstici vidimo atribute, v vseh naslednjih pa vrednosti posameznih atributov. Vsaka vrstica predstavlja en program.

No.	imeStudenta Nominal	**cluster** Nominal	abstract Numeric	assert Numeric	boolean Numeric	break Numeric	byte Numeric	case Numeric	catch Numeric	char Numeric	class Numeric	continue Numeric	default Numeric
1	C1_original	cluster10	13.0	0.0	4.0	3.0	0.0	0.0	1.0	0.0	20.0	0.0	0.0
2	C1_plagiat1	cluster10	13.0	0.0	4.0	3.0	0.0	0.0	1.0	0.0	20.0	0.0	0.0
3	C1_plagiat5	cluster10	13.0	0.0	4.0	3.0	0.0	0.0	0.0	0.0	20.0	0.0	0.0
4	C2_original	cluster32	13.0	0.0	7.0	2.0	0.0	0.0	2.0	0.0	30.0	0.0	0.0
5	C2_plagiat1	cluster32	13.0	0.0	7.0	2.0	0.0	0.0	1.0	0.0	30.0	0.0	0.0
6	C3_original	cluster16	13.0	0.0	8.0	0.0	0.0	0.0	1.0	0.0	40.0	0.0	0.0
7	C4_original	cluster2	0.0	0.0	6.0	1.0	0.0	0.0	4.0	0.0	30.0	0.0	0.0
8	C4_plagiat1	cluster2	0.0	0.0	4.0	0.0	0.0	0.0	2.0	0.0	30.0	0.0	0.0
9	C5_original	cluster28	14.0	0.0	9.0	0.0	0.0	0.0	1.0	1.0	40.0	0.0	0.0
10	C6_original	cluster22	13.0	0.0	6.0	0.0	0.0	0.0	1.0	0.0	30.0	0.0	0.0
11	C7_original	cluster29	0.0	0.0	2.0	1.0	0.0	0.0	2.0	0.0	35.0	0.0	0.0
12	C8_original	cluster3	26.0	0.0	8.0	1.0	0.0	0.0	0.0	2.0	35.0	0.0	0.0
13	C8_plagiat1	cluster4	26.0	0.0	8.0	1.0	0.0	0.0	1.0	2.0	35.0	0.0	0.0
14	C9_original	cluster11	13.0	0.0	6.0	8.0	0.0	0.0	2.0	0.0	25.0	0.0	0.0
15	C10_original	cluster1	13.0	0.0	9.0	0.0	0.0	0.0	6.0	8.0	45.0	0.0	0.0
16	C11_original	cluster13	13.0	0.0	7.0	0.0	0.0	0.0	1.0	0.0	30.0	0.0	0.0
17	C12_original	cluster8	13.0	0.0	7.0	2.0	0.0	0.0	2.0	1.0	35.0	0.0	0.0
18	C13_original	cluster31	13.0	0.0	4.0	2.0	0.0	0.0	0.0	1.0	15.0	2.0	0.0
19	C13_plagiat1	cluster31	13.0	0.0	4.0	2.0	0.0	0.0	0.0	1.0	15.0	2.0	0.0
20	C14_original	cluster20	13.0	0.0	17.0	2.0	0.0	0.0	7.0	3.0	65.0	1.0	0.0
21	C15_original	cluster26	12.0	0.0	7.0	3.0	0.0	0.0	1.0	0.0	35.0	0.0	0.0
22	C16_original	cluster19	0.0	0.0	2.0	0.0	0.0	0.0	0.0	0.0	10.0	0.0	0.0
23	C17_original	cluster5	13.0	0.0	6.0	1.0	0.0	0.0	1.0	1.0	25.0	0.0	0.0
24	C18_original	cluster30	0.0	0.0	1.0	0.0	0.0	0.0	1.0	1.0	25.0	0.0	0.0

Slika 3.3: Prikaz rezultata v Weki. Kot rezultat se pripne nov atribut cluster, ki pove, h kateremu roju spada posamezen program.

4. TESTIRANJE

Za testiranje in izboljševanje svoje metode sem naredil množico programov, za katere vem, ali so plagiarizirani ali ne. Nato sem testiral še na seminarskih nalogah iz predmetov Algoritmi in podatkovne strukture in Osnove algoritmov in podatkovnih struktur.

Pri izdelavi označene baze sem uporabil 32 programov iz leta 2006 pri predmetu Algoritmi in podatkovne strukture. Od tega sem jih 8 spremenil. Na ta način sem dobil bazo 40 primerov, nad katerimi sem izvajal svojo metodo. Za nalogo so učenci morali izdelati besedno statistiko nad poljubnim slovenskim besedilom z vsaj 10.000 besedami. Izdelati so morali slovar, v katerem so vse različne besede iz tega besedila, ter seznam, v katerem so elementi: število prebranih besed, število različnih besed, zadnja nova beseda. Seznam so morali implementirati s kazalci, algoritme pa uporabiti na rekurziven način, kjer je bilo to primerno. Kot rezultat analize je moral program izpisati seznam po desetinah. Študenti so imeli na voljo določene razrede, s katerimi so lažje realizirali nalogo.

Pri pisanju plagiatov sem uporabljal različne tehnike goljufanja. Večino idej sem našel v člankih, ki primerjajo različne metode odkrivanja plagiatov. Eden izmed ciljev goljufa je, da videz programa že na prvi pogled ni preveč podoben. V nadaljevanju so opisane tehnike goljufij za vsak primer posebej.

4.1 IZDELAVA OZNAČENE BAZE

Prvi primer:

- sprememba imen vseh spremenljivk, razredov ter metod.

Gre za zelo enostaven primer, saj sprememba imena spremenljivk ne vpliva na število ključnih besed. Prav tako nima nobenega vpliva na število spremenljivk.

Drugi primer:

- sprememba imen vseh spremenljivk, razredov ter metod;
- sprememba vizualne podobe z dodajanjem presledkov, tabulatorjev, novih vrstic;
- menjava vrstnega reda deklaracij spremenljivk, prireditvenih stavkov;
- sprememba izpisov na zaslon;
- sprememba konstant.

Tudi ta primer je lahek, saj nobena od omenjenih sprememb ne vpliva na število ključnih besed ali število spremenljivk.

Tretji primer:

- sprememba imen spremenljivk, dodajanje presledkov, tabulatorjev, sprememba komentarjev itd.;
- menjava vrstnega reda deklaracij spremenljivk in njihovih morebitnih inicializacij;
- sprememba operatorja ++ v navadno seštevanje;
- brisanje izjeme;
- brisanje komentarjev, izpisa;
- združevanje deklaracij istih tipov. Npr. *int a; int b; int c*; postane *int a, b, c;*.

Ta primer je prvi, ki spremeni število ključnih besed z združevanjem deklaracij ter brisanjem izjeme.

Četrti primer:

- sprememba imen spremenljivk, dodajanje presledkov, tabulatorjev itd.;
- menjava vrstnega reda deklaracij spremenljivk;
- združevanje deklaracij;
- sprememba for zank v while zanke;
- sprememba else if v zaporedje gnezdenih if stavkov;
- namesto da uvozimo celoten paket, uvozimo vsak razred posebej;
- brisanje try catch bloka, dodajanje throws v metodo.

Razen prvih dveh metod vse ostale spremenijo število ključnih besed.

Peti primer:

- sprememba imen spremenljivk, dodajanje presledkov, tabulatorjev itd.;
- menjava vrstnega reda deklaracij spremenljivk;
- združevanje deklaracij;
- brisanje dela kode, ki izvaja preverjanje, ali je uporabnik dal ime datoteke pri zagonu. Originalni pisec programa je precej zakompliciral to preverjanje, zato sem celoten del zamenjal z IOExceptionom ter izpisom.

Brisanje kode je lahko precej nevarno, saj hitro zmanjša število ključnih besed ter število spremenljivk.

Šesti primer (slika 4.1):

- sprememba imen spremenljivk, dodajanje presledkov, tabulatorjev, brisanje komentarjev, itd.;
- menjava vrstnega reda deklaracij spremenljivk;
- združevanje deklaracij;
- brisanje krajše metode ter njena postavitev na mesto, kjer se jo je pred tem klicalo;
- ustvarjanje novih spremenljivk pri vmesnem izračunu;
- brisanje modifikatorjev dostopa (public, private, static).

Zadnje štiri metode spreminjajo število ključnih besed, ustvarjanje novih spremenljivk pa spreminja tudi število spremenljivk.

Sedmi primer:

- sprememba imen spremenljivk, dodajanje presledkov, tabulatorjev, brisanje komentarjev, itd.;
- menjava vrstnega reda deklaracij, spremenljivk;
- združevanje deklaracij;
- dodajanje this pri dostopu spremenljivk instance objekta.

Zadnji dve metodi vplivata na število ključnih besed.

Osmi primer:

- sprememba imen spremenljivk, dodajanje presledkov, tabulatorjev, brisanje komentarjev itd.;
- menjava vrstnega reda deklaracij, spremenljivk;
- spremembe izpisov, konstant ;
- združevanje deklaracij;
- brisanje krajše metode ter njena postavitev na mesto, kjer se jo je pred tem klicalo;
- brisanje nepotrebne kode.

Zadnje tri metode spreminjajo število ključnih besed, zadnji dve pa tudi število spremenljivk.

Obstajajo tudi druge metode goljufij, ki pa jih zaradi specifičnosti kode nisem mogel uporabiti. Gre za:

- sprememba switch stavka z zaporedjem if stavkov;
- dodajanje break stavka v vsak primer switch stavka;
- sprememba ? v if stavek;
- premik return stavka, ki sledi if stavku, v nov else stavek;
- sprememba prireditve majhne tabele s prirejanjem posameznih elementov tabele;
- spremembe podatkovnih struktur (npr. String zamenjamo s tabelo znakov);
- menjava več spremenljivk istega tipa s tabelo in obratno, itd..

```
public static void main(String args[]) {


   BufferedWriter out, out2;
   String text = "";
   InputStreamReader in;
   int data;
   char c, ch;
   Slovar slovar;
   List seznamL;


   try {
      in = new InputStreamReader(new FileInputStream("input.txt"));
      out = new BufferedWriter(new FileWriter("cleared.txt", false));

      slovar = new CellCursor();
      seznamL = new ListLinked();
      while ((data = in.read()) != -1) {
         c = (char) data;
         ch = Character.toUpperCase(c);
         if (Character.isLetterOrDigit(ch)) {
            text = text + ch;
         }
         else {
            if(text != "") {
               out.write(text+"\r\n");
            }
            text = "";
         }
      }

      in.close();
      out.close();
      insertIntoLists(slovar, seznamL);
      out2 = new BufferedWriter(new FileWriter("Output.txt", false));
      seznamL.printToFile(out2, 1, seznamL.retrieveNrEl(seznamL.last()), 1);
      out2.close();
   }
   catch (IOException iox) {
      System.out.println("IOException: " + iox.getMessage());
   }
}
```

```
public static void main(String [] args) throws IOException {

   System.out.println("Seminarska I\n");

   clearFile();
   D = new CellCursor();
   L = new ListLinked();

   fillStructures();

   File base = new File("./", "Output.txt");
   BufferedWriter writer = new BufferedWriter(new FileWriter("Output.txt", false));

   L.printToFile(writer, 1, L.retrieveNrEl(L.last()), 1);
   writer.close();
}//main
                  |
//Funkcija gre cez besedilo in zapisuje elemente (besede)
//brez locil v novo "ocisceno" datoteko
public static void clearFile() throws IOException {
   File base = new File("./", cleared_file);
   InputStreamReader reader = new InputStreamReader(new FileInputStream(input_file));
   BufferedWriter writer = new BufferedWriter(new FileWriter(cleared_file, false));
   int data = reader.read();
   String buff = "";

   do {
      char ch = (char) data;
      ch = Character.toUpperCase(ch);
      if(Character.isLetterOrDigit(ch)) {
         buff += ch;
      }
      else {
         if(buff != "") {
            writer.write(buff+"\r\n");
         }
         buff = "";
      }
      data = reader.read();
   } while(data != -1);
   reader.close();
   writer.close();
}
```

Slika 4.1: Prikaz primera plagiatorstva. Z barvami so označeni plagiarizirani deli kode.

4.2 PREDPROCESIRANJE

Nekatere programe študentov je bilo potrebno tudi malce spremeniti. Pojavil se je namreč problem, ker sem napisal sintaksni analizator za verzijo jave 1.4, nekateri študenti pa so uporabljali javo 1.5 (statični import ali generiki). Takšno kodo sem ustrezno popravil. Kljub tem popravkom se rezultati ne bi razlikovali, če bi napisal sintaksni analizator za verzijo jave 1.5, saj generiki nimajo vpliva na moje štetje, statičnih import stavkov pa je bilo zanemarljivo malo. Izpustil sem programe napisane v drugih programskih jezikih.

4.3 TESTIRANJE NA OZNAČENI BAZI

Testiranje je potekalo po naslednjem postopku. Najprej sem iz označene baze s svojim programom pridobil atribute za vsak program. Uvozil sem jih v program Weka ter izvajal različne metode rojenja. Pri atributih sem izvedel testiranje z naslednjimi atributi:

- v prvem primeru so bili atributi samo javine ključne besede;
- v drugem primeru so bili atributi javine ključne besede ter število in vrsta spremenljivk;
- v tretjem primeru so bili atributi javine ključne besede ter število in vrsta spremenljivk v programu. Nekatere atribute sem utežil. Uteži sem dodal tako, da sem atributom, ki so se mi zdeli pomembni, dodal meta informacijo o uteži. Večja kot je utež, bolj pomemben je atribut. Dodal sem tudi nekatere spremembe, kot so štetje true, false, null, ter [] operator. Ključni besedi for in while sem štel kot seštevek in ne kot posamezni ključni besedi. Na ta način sem rešil problem spreminjanja for zanke v while zanko ali obratno.

Ko sem imel izbrane atribute, sem izbral metodo rojenja. Uporabil sem naslednje metode:

- SimpleKMeans: Metoda KMeans generira K naključnih točk (centrov rojev) v prostoru atributov. Vsak primer je dodeljen najbližjemu centru roja. Z vsemi primeri v rojih se izračuna nov center roja. Vsi primeri se ponovno dodelijo najbližjim centrom. Postopek se ponavlja, dokler primeri ne menjajo rojev ali pa se centri rojev nehajo spreminjati;
- FarthestFirst: Ta metoda je implementacija algoritma »Farthest First Traversal Algorithm« [7];
- Expectation-maximization (EM): Ta statistična metoda izračuna verjetnosti pripadanju rojem glede na neko verjetnostno porazdelitev. Cilj algoritma je maksimirati verjetnosti podatkov glede na končne roje.

Na označeni bazi sem uporabil tudi programa JPlag in MOSS. V tabeli 4.1 so rezultati, ki sem jih dobil. V prvem stolpcu je opisana metoda, ki sem jo uporabil za odkrivanje podobnosti izvorne kode, v drugem pa število najdenih plagiatov med vsemi plagiati. V označeni bazi je bilo 40 primerkov, od tega 8 plagiatov.

Tabela 4.1: Rezultati na označeni bazi

	NAJDENIH / VSEH
Ključne besede : SimpleKMeans	6 / 8
Ključne besede : FarthestFisrt	5 / 8
Ključne besede : EM	6 / 8
Ključne besede + štetje spremenljivk : SimpleKMeans	7 / 8
Ključne besede + štetje spremenljivk : FarthestFisrt	7 / 8
Ključne besede + štetje spremenljivk : EM	5 / 8
Ključne besede + štetje spremenljivk z utežmi : SimpleKMeans	7/ 8
Ključne besede + štetje spremenljivk z utežmi : FarthestFisrt	8 / 8
Ključne besede + štetje spremenljivk z utežmi : EM	4 / 8
JPLAG	8 / 8
MOSS	8 / 8

4.3.1 KOMENTAR REZULTATOV NA OZNAČENI BAZI

Rezultati na označeni bazi so zelo dobri, vendar je potrebno vedeti, da naloga ni bila preveč težka in posledično dokaj kratka z vidika števila vrstic kode. Vse spremembe, ki so spremenile število ključnih besed, so spremenile njihovo število v omejenem obsegu. Kot najtežja za detekcijo sta se izkazala primera 6 in 8. Tu samo ključne besede niso zadostovale za odkritje plagiatov, saj je bilo potrebno uporabiti tudi število spremenljivk ter dodajanje uteži posameznim atributom. V precej problematične metode goljufanja bi uvrstil brisanje modifikatorjev dostopa, brisanje nepotrebne kode, zamenjava switch z if stavki, ki zahtevajo zelo malo časa za spremembo. Zelo težavno je tudi zmanjševanje števila metod in posledično več kode na mestu, kjer je bil prej klic metode. Verjetno bi z uporabo vseh možnih metod uspeli prelisičiti program, vendar tako spreminjanje zahteva tudi precej časa. Najbolj pogoste metode, na katere pomislijo goljufi, kot so menjava imen spremenljivk, metod in razredov, menjava vrstnega reda, sprememba izpisov, sprememba komentarjev ali dodajanje presledkov, so obsojene na neuspeh. Med algoritmi rojenja se je presenetljivo najbolje odrezal FarthestFirst.

Pri dodajanju uteži sem stremel k temu, da utežim atribute, ki se kljub goljufanju spreminjajo čim manj. Uporabil sem naslednje vrednosti:

- class je dobil utež 5,
- new je dobil 4,
- return je dobil 2,
- void je dobil 2,
- atribut, ki predstavlja število zank (while + for) je dobil 10,
- vse spremenljivke in njihovo število so dobili 10.

Slabost pristopa s ključnimi besedami, ki pa se na označeni bazi ni pokazala, je dodajanje ali brisanje večjega dela kode, ki precej spremeni število ključnih besed ter število spremenljivk. Primer takih goljufij so npr. razredi ali metode, ki se nikoli ne kličejo (taki primeri se pojavijo npr. ko nekdo v svojo mapo z izvorno kodo doda cel kup datotek, ki jih dejansko program nikoli ne rabi). Drugi primer je, ko ima pisec originalnega programa kar nekaj kode, ki jo je uporabljal za testiranje, vendar se v končni različici programa ta koda ne kliče. Goljuf lahko z brisanjem te kode precej zmanjša število ključnih besed in število spremenljivk. Tretji primer goljufanja je brisanje določene funkcionalnosti, npr. originalni pisec naredi recimo grafični vmesnik, medtem ko goljuf zbriše vso kodo za grafični vmesnik ter dela vse operacije preko tekstovnega vmesnika.

4.4 TESTIRANJE NA REALNIH PRIMERIH

Testiranje je potekalo na treh bazah programov iz predmetov Algoritmi in podatkovne strukture (APS) ter Osnove algoritmov in podatkovnih struktur (OAPS). Kot rezultat sem upošteval vse najdene plagiate, ki sem jih našel s programoma JPlag ter MOSS, saj bi ročno pregledovanje tako velikega števila programov vzelo preveč časa. Pri nekaterih programih je šlo zagotovo tudi za delna kopiranja, vendar teh programov po definiciji ne štejemo med plagiate. V spodnjih tabelah (tabela 4.2, tabela 4.3, tabela 4.4) so rezultati na posameznih bazah programov.

Tabela 4.2: Rezultati na bazi APS 06/07 – prva seminarska naloga. V tej bazi je bilo 115 programov.

	NAJDENIH / VSEH
Ključne besede : FarthestFisrt	2 / 3
Ključne besede : EM	0 / 3
Ključne besede + štetje spremenljivk z utežmi : FarthestFisrt	1 / 3
Ključne besede + štetje spremenljivk z utežmi : EM	1 / 3
JPLAG	2 / 3
MOSS	3 / 3

Tabela 4.3: Rezultati na bazi OAPS 06/07 – prva seminarska naloga. V tej bazi je bilo 60 programov.

	NAJDENIH / VSEH
Ključne besede : FarthestFisrt	0 / 7
Ključne besede : EM	0 / 7
Ključne besede + štetje spremenljivk z utežmi : FarthestFisrt	2 / 7
Ključne besede + štetje spremenljivk z utežmi : EM	1 / 7
JPLAG	7 / 7
MOSS	7 / 7

Tabela 4.4: Rezultati na bazi OAPS 06/07 – druga seminarska naloga. V tej bazi je bilo 48 programov.

	NAJDENIH / VSEH
Ključne besede : FarthestFisrt	1 / 2
Ključne besede : EM	1 / 2
Ključne besede + štetje spremenljivk z utežmi : FarthestFisrt	1 / 2
Ključne besede + štetje spremenljivk z utežmi : EM	1 / 2
JPLAG	2 / 2
MOSS	2 / 2

4.4.1 KOMENTAR REZULTATOV TESTIRANJA NA REALNIH PRIMERIH

Rezultati testiranja na realnih primerih so povprečni. Uspešno sem zaznal goljufe, ki so spreminjali imena spremenljivk, metod in razredov, menjali vrstni red stavkov, spreminjali izpise, dodajali presledke, tabulatorje, se pravi metode, ki ne spreminjajo števila ključnih besed in število spremenljivk. Edino presenečenje je, da sem našel dva primera, kjer sta goljufa zamenjala switch stavek z 11 if stavki. Sklepam, da jima ni uspelo s spremembami zmanjšati števila drugih ključnih besed, prav tako pa if, case in switch nimajo večje uteži. V nekaterih primerih se pozna, da določena koda manjka, ali pa je dodana datoteka, ki se nikoli ne kliče. V teh primerih začnejo nekateri atributi precej odstopati po številu in primer ni zaznan kot plagiat. Po analizi atributov v vseh treh bazah programov se je pokazalo, da so tipi spremenljivk in njihovo število bolj stabilni kot ključne besede. Težava je le, da število spremenljivk ni veliko. Z izjemo integer spremenljivk se spremenljivke ostalih tipov pojavljajo veliko redkeje. Z boljšo nastavitvijo uteži bi verjetno lahko izboljšal rezultate, vendar sem uteži nastavil tako, da sem maksimiral detekcijo na označeni bazi ter nato s temi utežmi testiral realne primere.

Verjetno je bil problem tudi prelahek oziroma prekratek program v označeni bazi, ki ni dopuščal velikega števila modifikacij.

4.4.2 METODE PLAGIATORSTVA V REALNIH PRIMERIH

Seznam metod, ki so jih uporabljali študenti v 12 plagiatih:

- sprememba imena spremenljivk, metod, razredov [9-krat],
- spremembe izpisov [10-krat],
- dodajanje presledkov, tabulatorjev, novih vrstic [5-krat],
- spreminjanje komentarjev [4-krat],
- spreminjanje switch stavka v zaporedje if stavkov [3-krat],
- brisanje ne nujno potrebne funkcionalnosti (brisanje GUI, brisanje grafa) [3-krat],
- brisanje switch stavka [2-krat],
- spreminjanje konstant [2-krat],
- uvedba nepotrebnih spremenljivk [2-krat],
- sprememba for v while ali obratno [2-krat],
- menjava vrstnega reda if stavka [2-krat],
- združevanje deklaracij [1-krat],
- prestavljanje kode v novo manjšo metodo [1-krat],
- odstranjevanje nepotrebne kode [1-krat],
- odstranjevanje this pri dostopu spremenljivke [1-krat].

5. ZAKLJUČEK

Na področju iskanja podrobnosti v izvorni kodi sta v literaturi kot najboljša omenjena sistema JPlag in MOSS. Izdelal sem novo metodo za iskanje podobnosti v izvorni kodi. Pri tem sem uporabil nekatere ideje iz literature in svoje ideje. Na koncu sem primerjal rezultate razvite metode z rezultati sistemov JPlag in MOSS.

Rezultat moje metode za iskanje podobnosti izvorne kode, ki temelji na atributnem sistemu, bi lahko bil boljši, vendar pa je potrebno upoštevati, da je problem, ki sem ga reševal, dokaj težak.

Izdelava programa za pridobitev atributov je zahtevala orodja in znanje iz področja prevajalnikov. Eden od problemov je bil, katere atribute izbrati, preden izvedemo rojenje. Za bolj zamudno se je izkazalo testiranje, saj sem testiral različne metode rojenja pri različnih nastavitvah.

Postopek z uporabo ključnih besed ter številom in vrsto spremenljivk je bil uspešen le pri lažjih primerih goljufanja. Verjetno bi z določenimi izboljšavami lahko dosegli nekoliko boljši rezultat na plagiatih, kjer so se izvršile samo modifikacije kode. Pri plagiatih, pri katerih je zbrisan ali dodan večji del kode, oziroma določene funkcionalnosti, pa mislim, da ni možna detekcija na tak način, saj goljuf v preveliki meri spremeni vrednosti atributov.

Izboljšali bi lahko tudi označeno bazo, tako da bi vanjo vključili daljše in zahtevnejše programe (s tem bi omogočili več vrst goljufij ter pogostejše goljufanje). Verjetno bi morali narediti celo več označenih baz z različnimi programi, da se metoda ne bi preveč prilagodila eni vrsti programov. Nadgradenja postopka bi bila zagotovo tudi ukinitev case in else ključne besede kot atributov. Namesto njiju bi vsak case šteli kot if, prav tako else if in else. Tako bi lahko bolj zanesljivo predstavili število vejitev v programu.

Naslednja možnost izboljšave je, da bi s sintaksnim analizatorjem analizirali npr. vgnezdene zanke in ustvarili atribute, ki bi predstavljali, kaj se dogaja v zanki (npr. ali so to vejitve, klici konstruktorjev, metod, aritmetični izrazi...), podobno kot to naredi JPlag pri spreminjanju izvorne kode v znakovne nize.

Ko imamo nek nabor atributov, bi lahko metodo še izboljšali z dobrim uteževanjem. Zanimivi za uteževanje so tisti atributi, ki pri modifikacijah kode zelo malo spreminjajo svojo vrednost. Upoštevati moramo tudi pogostost atributa ter težavnost izvedbe goljufije, ki bi lahko spremenila vrednost atributa. V primeru, da je goljufija zahtevnejša (zahteva več dela s strani goljufa), lahko atribut bolj utežimo.

Prikazano metodo za iskanje podobnosti izvorne kode bi izboljšali tudi tako, da bi pri rojenju omejili število primerov v roju na 4, saj goljufanje v večjih skupinah ni pogosto.

V povezavi s prikazom goljufij bi lahko dodali pozicijske informacije pri atributih, kjer je to smiselno. S tem bi lahko barvno označili dele izvorne kode, ki so sumljivo podobni.

Pri svojem delu sem ugotovil, da atributni sistemi zaenkrat ne morejo konkurirati s strukturno metričnimi, kot sta JPlag in MOSS. Poleg tega večina atributnih sistemov težko prikaže sumljive dele kode, prikaz sumljivih delov kode pa je zelo dober pripomoček pri končni odločitvi, ali gre za plagiat. Med MOSS in JPlagom bi dal majhno prednost MOSS-u, saj ocenjujem, da bolje zazna delne goljufe ter v večini primerov vrne manjše število bolj pomembnih sumljivih delov kode.

SLIKE

TABELE

LITERATURA

[1] Culwin. McLeod & Lancaster, Source Code Plagiarism in UK HE computing Schools, Issues, Attitudes and Tools. JISC Tehnical report 2001. Dostopno na: http://www.jiscpas.ac.uk/images/bin/southbank.pdf

[2] D. Mladenic, "Text-Learning and Related Intelligent Agents: A Survey“, *IEEE Intelligent Systems,* št. 14, zv. 4, str. 44-54, Julij/Avgust, 1999

[3] L. Prechelt, G. Malpohl & M. Phillepsen, "Finding plagiarism among a set of programs with JPlag“, *Journal of Universal Computer Science*, št. 8, zv. 11, str. 1016-1038, 2002.

[4] S. Schleimer, D.S. Wilkerson, A. Aiken, Winnowing: Local Algorithms for Document Fingerprinting. Dostopno na: http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf/

[5] L. Moussiades, A.Vakali, Pdetect, "A Clustering Approach for Detecting Plagiarism in Source Code Datasets", *The Computer Journal, š*t 46, zv, 6, str. 651-661, November 2005

[6] P. Campbell, F. Culwin, *Using compression to identify plagiarised programs*, Galway: Proc. 4th ICT LTSN Conference, Avgust 2003

[7] S. Hochbaum, D.B. Shmoys, "A best possible heuristic for the k-center problem", *Mathematics of Operations Research*, št 10, zv 2, str 180–184, 1985.

[8] JFlex: http://jflex.de/

[9] JavaCUP: http://www2.cs.tum.edu/projects/cup/

[10] Weka: http://www.cs.waikato.ac.nz/ml/weka/

[11] JPlag: https://www.ipd.uni-karlsruhe.de/jplag/

[12] MOSS: http://theory.stanford.edu/~aiken/moss/

[13] Java 1.4 gramatika: http://www.cs.princeton.edu/~appel/modern/java/CUP/

[14] Invertiran indeks: http://en.wikipedia.org/wiki/Inverted_index, 23.9.2008

IZJAVA

Študent Marko Lugarič izjavljam, da sem avtor tega diplomskega dela, ki sem ga napisal pod mentorstvom doc. dr. Marka Robnika Šikonje, in dovolim objavo diplomskega dela na fakultetnih spletnih straneh.

V Ljubljani, dne 13. 10. 2008 Podpis:

_____