Janko Cvitaš
0036390589
Mentor: Marin Golub, dr.sc.

seminarski rad

Detekcija plagijata

svibanj, 2005

Fakultet elektrotehnike i računarstva

Sadržaj:

1. Uvod – plagijati

"Književno, znanstveno ili drugo djelo nastalo prepisivanjem u cjelini, u bitnim dijelovima ili u prepoznatljivim dijelovima i prisvajanjem tuđeg rada uloženog u to djelo."-Vladimir Anić, Riječnik hrvatskog jezika, Novi Liber, Zagreb 1991

Plagijatorstvo nikad nije bilo lakše nego danas. Prije ekspanzije Inteneta prepisivanje nečijeg tuđeg rada je bilo vrlo zahtjevno, a rezultati tj. sličnost je bila očita. Potencijalni plagijatori imali su na raspolaganju ograničenu količinu materijala, najčešće iz knjiga. Plagijat se stvarao ručnim prepisivanjem i zahtjevao je puno truda i vremena, a budući su izvori plagijarizma bili vrlo stručno pisani, rizik od otkrivanja je bio velik. U takvim uvjetima, plagijatorstvo je bilo neisplativo, pa time naravno i manje rašireno.

Danas, Internet omogućava pronalaženje tisuća relevantnih radova u malom vremenu, te je uvelike olakšano stvaranje plagijata radova, članaka pa čak i knjiga. Gotovo je nemoguće iz tolike količine izvornih tekstova programa i tekstova pronaći izvore plagijata. Osim u samoj količini dostupnih izvora, problem leži i u tome što je teško primjetiti plagijat na temelju "prestručnog" izražavanja, ili prekompleksnog načina programiranja.

Elektronički oblik se vrlo lako reproducira, a zbog svoje "javnosti" se još uvijek (pogrešno) ne smatra intelektualnim vlasništvom, te se manje osvrće na činjenicu da je ilegalno kopirati tuđe sadržaje i prezentirati ih kao svoje. Uz razne programe (*peer to peer*) i ostale načine za distribuciju elektroničkih podataka, plagijatorstvo je postalo uobičajena praksa.

Kao reakcija na sve veći broj mogućnosti plagijatora za predstavljanje tuđeg rada kao svojeg, razvijen je velik broj algoritama i metoda, raspoređen u još većem broju programa za detekciju plagijata. Takvi alati se koriste u pravnim ustanovama, u slučaju optužbi plagijatorstva komercijalnog softvera, te u obrazovnim ustanovama za provjeru samostalnosti rada učenika i studenata. Osim toga, te iste metode i programi koriste se za praćenje različitih verzija programa (posebno za open-source programe), za optimiranje, i za razne oblike analize izvornih tekstova programa. Ovisno o algoritmu usporedbe, programi se mogu koristiti za usporedbu izvornih tekstova pisanih u raznim programskim jezicima i običnog teksta.

Cilj ovog seminara je prikazati razne programe za detekciju plagijata izvornih tekstova programa i općenite algoritme i metode korištene u tu svrhu. Neki od njih se mogu koristiti i za usporedbu običnih tekstova, naravno uz odgovarajuće prednosti i mane. Kao praktičan dio seminara, implementiran je Ratcliff/Obershelp algoritam unutar programa za detekciju plagijata studentskih programa.

2. Programka podrška za detekciju plagijata

2.1 Analiza teksta

U ovom seminaru bit će prikazani razni alati za detekciju plagijata izvornih tekstova programa, kao i običnog teksta. Postoji više raznih pristupa kako odrediti sličnost između dva izvorna teksta. Naravno, osim sličnosti mogu se tražiti i razlike, u programima za analiziranje promjena za dvije verzije izvornih tekstova. Velik broj riješenja za detekciju plagijata ne uzima u obzir strukturne i sintaksne informacije sadržane u izvornom tekstu programa. Ta činjenica staje na put kompleksnijoj analizi sličnosti između izvornih tekstova programa.

Razlog tim ograničenjima je način gledanja usmjeren prema znakovima unutar datoteka sa izvornim tekstovima programa. Očito, to nije programerski pogled na izvorni tekst programa, i se gubi ideja po kojoj je taj izvorni tekst napisan. Kod takvih alata, leksička struktura ostaje sačuvana i uspoređuje se, ali informacije o sintaksnoj strukturi se nigdje ne pohranjuju i ne koriste za analizu.

Drugi pristup ima kompilatorski usmjeren način gledanja na izvorni tekst programa. Kod ovog pristupa, sve sintaksne informacije su pohranjene u apstraktno sintaksno stablo i razlike (tj. sličnosti) se gledaju kroz operacije na grafu tog stabla. Pri tome se gubi leksička struktura izvornog teksta programa i postoji problem kod povezivanja sličnosti i mjesta u originalnom dokumentu.

Treći, moglo bi se reći kombinirani, pristup je umetanje informacija o sintaktičkoj strukturi u sam izvorni tekst programa. Pri tome sama leksička struktura izvornog teksta programa ostaje sačuvana, a može se detaljnije analizirati razlika između izvornih tekstova programa, bilo na temelju sintaksne strukture, ili programske metrike (broj varijabli, petlji i slično).

U daljnjem dijelu seminara opisani su razni programi koji predstavljaju razne karakteristične načine traženja plagijata. Prvo su opisane općenite strategije uspoređivanja, a nakon toga slijedi opis raznih algoritama koji se mogu koristiti u te svrhe. Popularnost upotrebe tih algoritama ovisi o njihovoj efikasnosti, te razini do koje su razvijeni.

2.2 Programi za usporedbu tekstova *Diff* i *Patch*

Diff je razvijen u ranim 1970-ima na operacijskom sustavu *Unix*. Jedna od verzija tog programa ugrađena je i u sam operacijski sustav. Radi se o jednom od jednostavnijih alata za usporedbu, baziranih na tekstu tj. znakovima. *Diff* se stoga može koristiti i za usporedbu običnog teksta, a ne samo izvornog teksta programa. Ovaj program uzima dvije tekstualne datoteke kao ulaze i uspoređuje ih na temelju linija. To znači da program uspoređuje znakovne nizove svake linije pomoću metode pronalaženja najdužeg zajedničkog podslijeda (*longest common subsequence*).

Kao izlaz, *Diff* javlja koje od linija su dodane, koje pobrisane, a koje izmijenjene. To je vrlo grub način usporedbe, i mala promjena u sintaksnoj strukturi

programa može imati ogroman utjecaj na linijski raspored programa. Tako jedna promjena u sintaknoj strukturi može predstavljati višestruku linijsku promjenu. Kao promjenu dobivamo samo broj linije, dok precizniji kontekst razlike moramo odrediti tako da sami provjeravamo okolni izvorni tekst programa i odredimo koji sintaksni elementi izvornog teksta programa su izmijenjeni (i jesu li izmijenjeni). Ovaj program daje malu sigurnost u određivanju sličnosti, i lako ga je natjerati da ocijeni sličnost sa vrlo niskom ocjenom u slučaju da se koristi za utvrđivanje plagijata. Uspješnija uporaba bi bila za određivanje razlike između dva programa ili teksta, kada se očekuje mali broj razlika. Postoji više načina na koji se može pokrenuti taj program o čemu ovisi izgled izlaza iz programa.

Primjer pokretanja:

```
bash$ diff -c a b
*** a   Wed May 11 21:47:34 2005
--- b   Wed May 11 21:59:36 2005
***************
*** 1 ****
--- 1,3 ----
  ovo je neki kod,
+ bla
+ to sad ide kao tekst.
```

Prva linija u ova dva teksta je zajednička, dok su u drugom tekstu dodane druga i treća linija. Ovisno o redoslijedu navođenja, ispisuje se koje su linije izbrisane, te se određuje koje su izmijenjene. Osim toga, izlaz ne omogućava detaljniju analizu, ali ga se može proslijediti u program *Patch*.

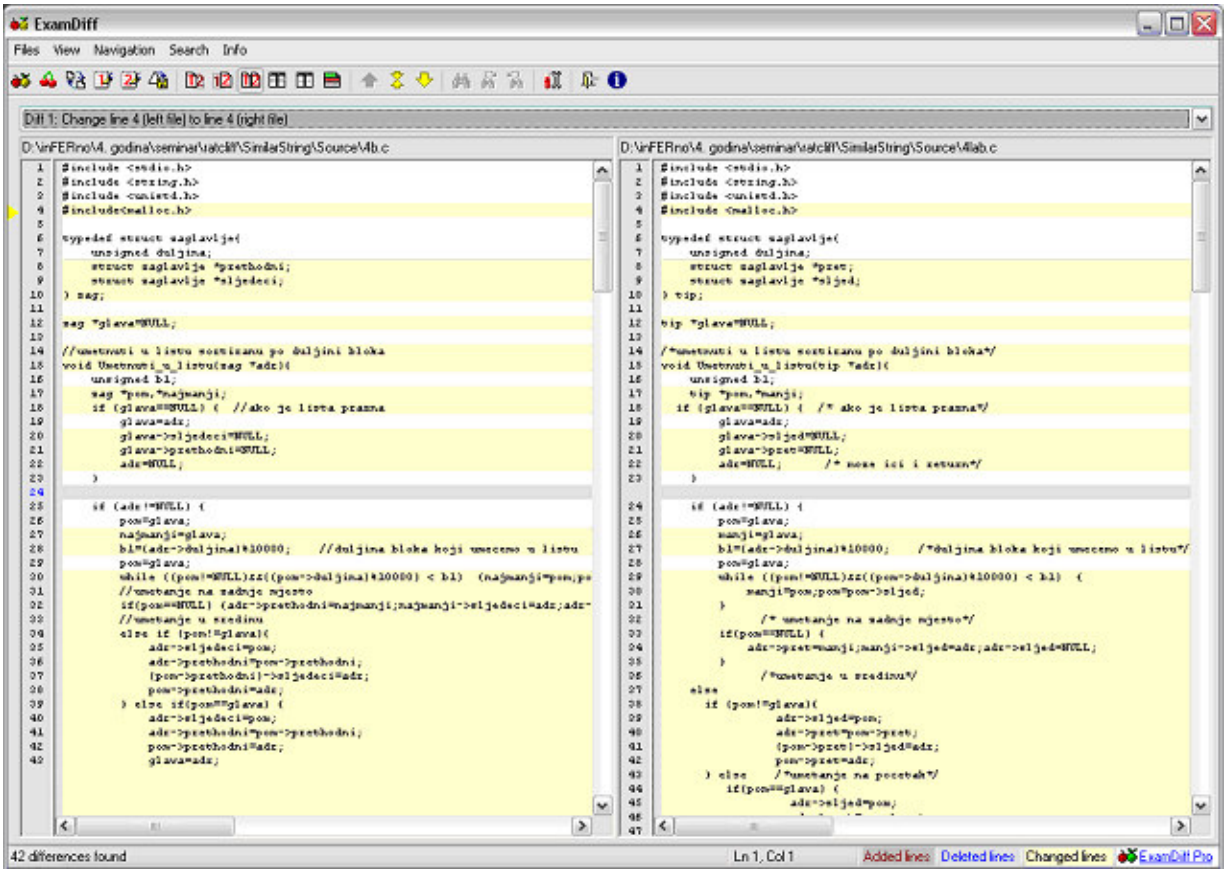
Patch uzima "zakrpu" iz datoteke koju je stvorio program *Diff* i pokušava primjeniti razlike zapisane u toj datoteci i stvoriti "zakrpane" tj. *patched* verzije. Taj program nema svrhu otkivanja razlike nego samo prilagođavanja izvornih tekstova programa. [13]

Ovaj program očito ne pruža veliku učinkovitost u detektiranju plagijata, jer provjerava odnose linija između dva izvorna teksta. Taj postupak se može popraviti uključivanjem nekih opcija poput zanemarivanja promjena u praznim mjestima (*white space*), ali svejedno nije baš prikladan za ovu svrhu. Plagijator vrlo lako može postići dojam da nema sličnosti između dva izvorna teksta programa, običnim umetanjem komentara ili promjenom rasporeda linija, dodavanjem praznih linija i sličnim postupcima koji uopće ne mijenjaju ponašanje programa nego samo njegov izgled prije prolaska kroz kompilator, što se ne može nazvati dovoljnom razlikom da oba izvorna teksta programa proglasimo originalnim.

2.3 Program za usporedbu izvornih tekstova *ExamDiff Pro*

Ovaj se alat bazira na istim metodama kao i *Diff* program na operacijskom sustavu *Unix*. Program je napisan za operacijski sustav *Windows*. *ExamDiff* omogućava jednostavnije uspoređivanje datoteka koje sadržavaju tekst, izvorni tekst programa, a čak i ostale vrste datoteka. Prednost ovog alata nad običnim *Diff* programom je korištenje korisnički-ljubaznog grafičkog sučelja za isticanje sličnosti i razlika dvaju izvorna teksta programa. Dodatna proširenja su uspoređivanje direktorija, datoteka sa UNICODE znakovima i slično. Uporabom ovog programa vidi se razina uspješnosti korištenja ovakvog pristupa za detekciju plagijata. Naime *ExamDiff* uspoređuje linije i govori nam samo je li linija dodana, izbrisana ili izmjenjena. Ovisno o tome, linije se označavaju raznim bojama. Najveći nedostatak je što najobičnija izmjena imena varijabli i korisnički definiranih funkcija uzrokuje da se većina programa označi tako da pokazuje da su linije izmijenjene, iako zapravo nema razlike između ta dva izvorna teksta programa. Također, dodavanje linijskih komentara na isti način uzrkokuje označavanje linije kao izmjenjene. Na kraju, dva izvorna teksta programa koja imaju veliku sličnost, bivaju u velikom postotku označeni kao izmijenjeni, te se zahtijeva "ručna" provjera nadležne osobe da li se radi o stvarnoj promjeni ili samo leksičkoj promjeni bez utjecaja na izvođenje i sintaksnu i semantičku strukturu. Ako se radi o velikom broju izvornih tekstova programa koji zahtijevaju usporedbu, postaje neisplativo koristiti ovaj pristup. Ovakvo linijsko pregledavanje izvornog teksta programa bilo bi prikladnije za traženje razlika između dvije verzije izvornog teksta programa kako bi se otkrilo što je izmijenjeno, i kada se ne očekuje prevelik broj razlika. Tada bi bilo razumno "ručno" pregledati koje su te razlike, iako samim programom ne dobivamo nikakve informacije o njima. [3]

Program je isproban na dva izvorna teksta koji su usporedbom Ratcliff/Obershelp algoritmom (vidi poglavlje 4.) ocijenjeni sa sličnosti od oko 80%. Korištenjem *ExamDiff* (ili običnog *Diff*) je većina linija označena kao izmijenjena, čime je izgubljena razumljivost razlika tj. sličnosti tih izvornih tekstova programa.



Slika 2.1: *ExamDiff* (obojene linije su "izmijenjene")

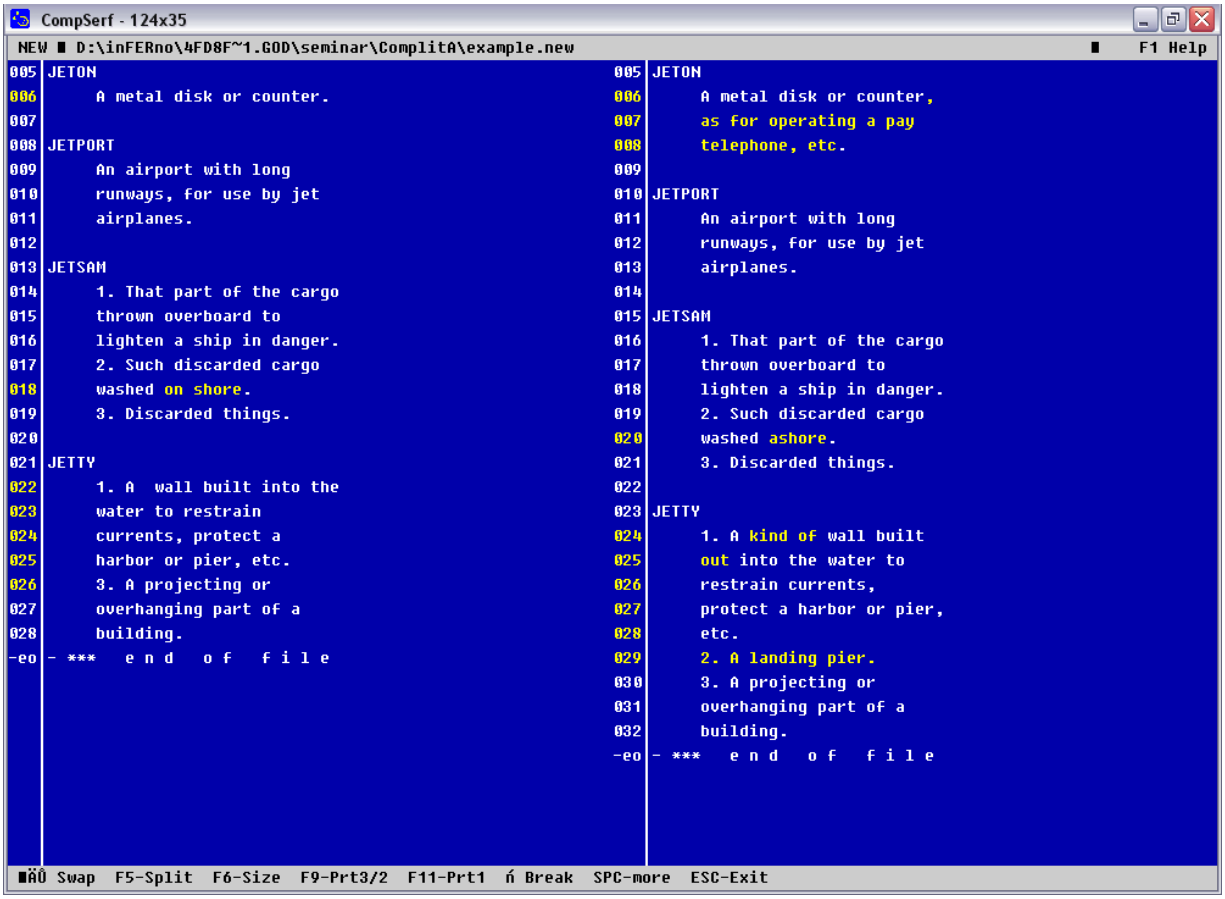
2.4 Program za usporedbu izvornih tekstova *CompSerf*

CompSerf je jedan od nekoliko sličnih programa napravljenih za različite operacijske sustave (*Windows*, *Os2*...), koji uspoređuju prvenstveno izvorne tekstove programa. Pristup koji ovaj program koristi je pregledavanje različitih verzija riječ po riječ. Te "riječi" mogu biti odvojene razmacima (blank), ali i standardnim programskim delimiterima, tj. znakovima korištenim u programskim jezicima za odvajanje blokova, izraza, i slično. Izlaz programa su oba izvorna teksta programa koje smo uspoređivali sa istaknutim razlikama između njih. Uspoređivanje razlika riječ po riječ se obavlja od dna prema vrhu, i za razliku od linijski definiranih usporedbi ovaj program nije osjetljiv na rastavljanje izraza kroz više redaka. Kada se otkrije razlika između riječi, može se odrediti i linija u kojoj je došlo do te izmjene, brojeći linije. Pomoću ove sitnije granulacije, tj. zbog usporedbe riječi, ovakva metoda je sposobnija uočiti sličnost između dvaju izvornih tekstova programa koji su jednaki po sadržaju, a različiti isključivo po rasporedu linija, što obično ne utječe na sintaksnu strukturu programa. Budući razmještaj ne utječe (najčešće) na izvođenje programa, ovakav pristup je značajan napredak u odnosu na navedeno linijsko uspoređivanje.

Osim što uspoređuje izvorne tekstove programa riječ po riječ, ovaj program ima još nekoliko opcija koje uvelike povećavaju preciznost usporedbe ovisno o vrsti izvornog teksta progama koji uspoređujemo. Te opcije se odnose na procesiranje ključnih riječi i fraza. Time je omogućeno zanemarivanje nekih promjena koje ne

smatramo bitnima. Dakle, možemo definirati način isključivanja datuma, oznaka vremena i brojeva stranica iz skupa razlika koje smatramo ključnima za našu usporedbu. Osim toga na raspolaganju nam je zanemarivanje komentara, zaglavlja, podnožja i slično. Moguće je i navesti više riječi koje će se u usporedbi smatrati ekvivalentnima te se neće prijaviti razlika između njih. Postoje dva načina rada, WORD i PGM. U WORD načinu se razmaci smatraju oznakom prestanka riječi pa bi stoga zapis datuma "01/02/05" bio smatran jednom riječi. U PGM načinu rada se koriste svi standardni programski delimiteri i ovaj izraz se razdvaja u pet riječi: "01", "/", "02", "/", "05". Moguće je zadati da se taj cijeli izraz smatra jednom riječi, u oba načina rada, zadajući "##/##/##" kao frazu koja se smatra cjelinom.

Postoje još neki srodni programi (od istog proizvođača) koji imaju neke slične mogućnosti. Npr. *TigSerf* koji ima vrlo slične rezultate kao *CompSerf*, ima mogućnost uspoređivanja većih nizova (*look-ahead*) i može naći više parova usporedbi, ali se izvršava sporije od svoje "slabije" inačice *CompSerf*. [4][1]



Slika 2.2: *CompSerf* (razlike su obojene)

2.5 Program za usporedbu izvornih tekstova programa *Comparator-Filterator*

Ovo je program za brzo pronalaženje zajedničkih dijelova u dva ili više programskih stabala. Nazvan je po zastarjelom astronomskom instrumentu "*blink comparator*" koji je korišten za pronalaženje objekata što se miču u fotografijama zvjezdanih polja. *Filterator* je postprocesor za izlaz iz *Comparatora* koji često zna sadržavati šum, pa *Filterator* ima mogućnost filtriranja po važnosti.

Comparator radi tako da najprije podijeli izvorni tekst programa u preklapajuće komade, koji su inicijalno postavljeni na tri linije izvornog teksta. Nakon toga se pomoću *hashing* funkcije računaju vrijednosti sažetaka tih dijelova. Rezultat je lista vrijednosti sažetaka koja se pregledava i iz nje se izbacuju svi unikatni elementi. *Comparator* stvara izvještaj sa svim nakupinama linija izvornog teksta programa (tri linije) sa spojenim dijelovima koji se preklapaju. Posljedica ovakvog pristupa je da Comparator pronalazi zajedničke dijelove izvornog teksta programa potpuno neovisno o tome gdje se nalaze. Dakle, postignuta je neosjetljivost na reorganiziranje tj. premještanje dijelova izvornog teksta programa. Osim toga postoje opcije čijim uključivanjem se zanemaruju razlike zbog praznih mjesta (*white space*), te je moguće postaviti program tako da je neosjetljiv na promjene u postavljanju zagrada (za izvorni tekst programa pisan u jeziku C). Još jedna od korisnih opcija je brisanje komentara iz izvornog teksta programa, tj. njegovog zanemarivanja u traženju sličnosti. Izlaz iz programa može biti i lista vrijednosti sažetaka, tako da ne moramo imati nužno sam izvorni tekst programa da bismo ga usporedili. Te liste se pohranjuju u .SCF datotekama što je kratica od *Source Comparison Format*. Za jedno ime datoteke kao ulazni parametar program stvara .scf datoteku za taj izvorni tekst programa i daje sadržaj .scf datoteke na standardni izlaz. Pomoću opcije –s postavlja se veličina na koju se izvorni tekst programa reže (inicijalno 3 linije). Smanjenjem broja linija koje program uzima kao najmanje jedinice, postiže se uočavanje manjih istih dijelova, ali se i povećava "šum" na izlazu. Povećavanjem "izrezaka" se smanjuje "šum", ali se i gubi uočavanje sitnijih elemenata sličnosti. Pomoću –m opcije određuje se minimalna veličina raspona koji će biti dan na izlaz. Inicijalno je ta veličina postavljena na nulu, a postavljanjem na veći iznos od veličine izrezaka izbacuje se velika količina bezvrijednog sadržaja iz izlaznog izvještaja. Program pokušava izbaciti šum iz izlaznog izvještaja, a segment izvornog teksta programa se smatra šumom ako datoteka ima ekstenziju .c ili .h i segment sadrži isključivo prazna mjesta (white space), interpunkcijske znakove, C ključne riječi i "C include" linije. Dakle, linija mora sadržavati korisničke varijable, imena funkcija ili korisnički definirane makro-funkcije. Drugi oblik šuma koji se izbacuje je u slučaju da datoteka ima .sh ekstenziju a linija se sastoji samo od shell ključnih riječi i interpunkcijskih znakova. Autor ovakvo izbacivanje naziva "significance filtering" i ta opcija se može isključiti.

Comparator dakle ne pokušava obaviti bilo koji oblik semantičke analize i nije pomoću njega moguće otkriti relativno trivijalne promjene poput izmjene imena varijabli i slično. Kako i sam autor naglašava, ovo nije alat za pronalaženje plagijata ideja (patenti) nego za pronalaženje plagijata u izrazu ideja (copyright). Prednost ovakvog načina traženja sličnosti je korištenje funkcija računanja sažetaka koje u vrlo velikoj mjeri onemogućuju izjednačavanje različitih dijelova izvornog teksta programa.

Koristeći RXOR funkciju možemo očekivati prvu pogrešnu usporedbu nakon 6,074,000,999 izrezaka, što je gotovo nemoguće dostići čak i kod vrlo velikih izvornih tekstova programa poput Unix/Linux izvornih tekstova. Ograničenje koje izaziva korištenje RXOR funkcije za računanje sažetaka je mogućnost krivih usporedbi u slučaju da se koriste izresci veći od 512 znakova. Način pretraživanja liste vrijednosti sažetaka je *brute-force* tj. iscrpnim pretraživanjem, ali unatoč tome program se izvodi razumno brzo. [5]

2.6 Program za usporedbu izvornih tekstova programa *Jplag*

2.6.1 Opis programa

Sustav Jplag razvijen je za usporedbu plagijata izvornih tekstova programa pisanih u programskim jezicima Java, Scheme, C i C++. Program je dostupan kao Internet usluga (servis). Izlaz iz programa je skup HTML stranica koje omogućavaju detaljno proučavanje i razumijevanje sličnosti između programa.
Način na koji Jplag provodi usporedbu:

- svi izvorni tekstovi programa koji se uspoređuju se parsiraju i pretvaraju u nizove koji se nazivaju "token strings". (*token* = značenjska jedinica, *string* = povezani niz znakova, u ovom slučaju "tokena")
- Dobiveni nizovi tokena se uspoređuju u parovima te se traži sličnost svakog od tih parova. Ta usporedba se provodi tako da se jedan od nizova tokena pokušava u što boljem postotku pokriti podnizovima tokena iz drugog niza tokena (u ovom dijelu teksta podniz će se smatrati povezanim). Postotak koji se uspije "pokriti" je mjera sličnosti.

Java izvorni tekst:

```
1)public class Count {
2)    public static void main(String[] args)
3)          throws Java.io.IOException {
4)    int count = 0;
5)    while(System.in.read() != -1)
6)          count++;
7)    System.out.println(count);
8)    }
9)}
```

Niz tokena dobiven iz tog izvornog teksta progama:
1)BEGINCLASS
2)VARDEF,BEGINMETHOD
3)(nema tokena)
4)VARDEF,ASSIGN
5)APPLY,BEGINWHILE
6)ASSIGN,ENDWHILE
7)APPLY
8)ENDMETHOD
9)ENDCLASS

2.6.2 Pretvaranje izvornih tekstova programa u nizove "*tokena*"

Prvi dio postupka usporedbe tj. pretvaranja u tokene je jedini dio programa koji je vezan uz sam programski jezik koji se uspoređuje. Ti programski moduli koji se koriste za jezike Java i Scheme implementiraju potpune jezične parsere za te jezike. Za jezike C++ i C koristi se samo "*scanner*". Parser kao *front-end* dio ovog alata omogućava da se više informacija o semantičoj strukturi unese u token.

Tokeni se izabiru tako da ne odražavaju površinska obilježja, nego bit programa, budući je taj dio plagijatoru teško promijeniti. Tako se prazni znakovi (*white space*) i komentari ne koriste za stvaranje tokena s obzirom da da ni na koji način ne utječu na sam program i najjednostavnije se mijenjaju. Algoritmi koji se baziraju na provjeru linija, riječi ili znakova a koji nemaju mogućnost zanemarivanja ovih dijelova su vrlo ranjivi na takve promjene. Neki se tokeni također ne smatraju korisnima, tako recimo osnovni skup tokena za Javu potpuno ignorira "*expression*"-e osim za dodijeljivanje i pozive funkcija. Broj linije se pohranjuje u sam token čime je omogućeno vraćanje unazad i određivanje mjesta u izvornom tekstu programa otkud je potekla određena sličnost.

2.6.3 Usporedba nizova "tokena"

Za usporedbu se koristi "pohlepni" (greedy) algoritam. Kod usporedbe dva niza tokena, pokušava se naći skup podnizova tokena koji su jednaki uz poštivanje nekih pravila. Svaki token iz jednog izvornog teksta progama se može upariti sa najviše jednim tokenom iz drugog izvornog teksta. Zbog tog pravila je nemoguće upariti dijelove izvornog teksta programa koji su umnoženi u izvornom tekstu - plagijatu. Podnizovi se traže neovisno o položaju u znakovnom nizu, čime se eliminira utjecaj drugačijeg razmještaja u plagijariziranom izvornom tekstu programa. Duži upareni podnizovi se smatraju boljima od kratkih koji su sumnjiviji, te se može raditi o sasvim slučajnim sličnostima (slova, dijelovi riječi...).

Koristeći ova pravila provode se dvije faze uspoređivanja:

U prvoj fazi se traže najdulji jednaki neprekinuti podnizovi. To se obavlja pomoću tri ugniježđene petlje. Prva petlja prolazi kroz sve tokene iz prvog izvornog teksta programa, dok druga uspoređuje trenutni token iz prve petlje sa svim tokenima iz drugog izvornog teksta. Ukoliko su tokeni identični, najugnježđenija, treća petlja pokušava usporediti i upariti što više idućih tokena.

Druga faza služi za označavanje svih uparenih (jednakih) podnizova tokena maksimalne duljine. Takvi označeni tokeni se ne mogu koristiti za daljnje uspoređivanje. Time se osigurava pravilo da će svaki token biti uparen samo jednom.

Ove dvije faze se ponavljaju sve dok se i dalje pronalaze novi parovi jednakih tokena. U svakom koraku se duljina najvećeg uparenog dijela smanjuje bar za 1 pa je stoga sigurno da će algoritam završiti. Osim toga, moguće je postaviti najmanju duljinu podniza tokena koja se uzima u obzir kod rezultata.

Mjera sličnosti se može odrediti na dva načina. Ako želimo da se sličnost od 100% prijavi u slučaju da su oba niza tokena potpuno jednaka, onda moramo uključiti oba znakovna niza u izračuanavanje. Ako pak želimo da se u slučaju kopiranja i proširivanja jednog izvornog teksta programa također prijavi sličnost od 100%, onda ćemo uportijebiti samo kraći od ta dva izvorna teksta tj. niza tokena.
Za prvi način izračunavanja bi vrijedilo:

sličnost(A,B)=2*pokrivenost(zajednički tokeni) / (|A| + |B|)
pokrivenost= $\Sigma_{pododaraju(a,b,duljina)}$ duljina

2.6.4 Složenost algoritma

Prva faza algoritma je zahtjevnija od druge, s obzirom da postoje maksimalno (|A| - MML) * (|B| - MML) istih dijelova (MML=*MinimumMatchLength*) koji mogu biti pronađeni. U drugoj fazi su u najgorem slučaju svi tokeni određeni za označavanje što se obavlja u vremenu linearno ovisnom o duljini kraćeg niza tokena.

Najgori slučaj: Radi olakšavanja analize složensti, pretpostavimo da su oba niza tokena jednako duga. U najgorem slučaju tri ugnjiežđene petlje se maksimalno izvršavaju, što znači da se u svakoj iteraciji pronađe samo jedan zajednički dio maksimalne duljine između dva niza tokena, da je ta duljina za jedan manja od maksimalne duljine u prethodnoj iteraciji i da se taj zajednički podniz uvijek nalazi na kraju preostalog neoznačenog dijela niza tokena. U tom slučaju obavljanje *k* iteracija pokrije niz koji ima n=k(k+1)/2 tokena. Broj uparenih dijelova je otprilike drugi korijen od (2n).

Najbolji slučaj: Čak i ako se nizovi token potpuno razlikuju, potrebno je usporediti sve tokene iz prvog niza tokena sa svima iz drugog. To zahtijeva |A| * |B| usporedbi.
Složenost algoritma je: $O(n^3)$

2.6.5 Efikasnost algoritma i napadi

Promatra se najmanji segment na koji je potrebno obaviti neku od tehniku prikrivanja plagijata da bi algoritam ocijenio taj dio i original različitima. Ta lokalna razlika (local confusion) ovisi o skupu tokena koji se koriste i o minimalnoj duljini podniza koji se mora upariti u oba izvorna teksta programa da bi se obilježio kao jednak. Ako se u segment tokena koji je minimalne dujine umetne linija teksta tj. token, tada Jplag neće označiti taj segment jer će duljina rastavljenih dijelova biti manja od minimalne. Ako pak segment minimalne duljine podijelimo na dva podsegmenta i njih zamijenimo (tamo gdje je to moguće), Jplag neće to označiti jer će smatrati to kao dva segmenta (podniza) koji su manji od minimalne dozvoljene duljine (naravno ta dva dijela ne smiju bit identična). To su neke od tehnika prikrivanja.

Postizanje te lokalne razlike je osnova za uspješan napad na Jplag. Svejedno, samo za vrlo male programe, koji su najviše dva puta dulji od minimalne duljine podniza, je dovoljno stvoriti jednu lokalnu razliku da bi napad bio uspješan. Za dulje programe ovisi o postotaku koji se uzima kao granična sličnost potrebna da bi se

jedan od dva izvorna teksta programa proglasio plagijatom. Za savršeni napad koji bi bio uspješan za bilo koji kritični postotak, bilo bi potrebno stvoriti "local confusion" na svim segmentima koji su minimalne dozvoljene duljine. Takva metoda prikrivanja plagijata zahtijeva dosta posla i pitanje je da li se isplati plagijatoru uložiti takav trud. Osim toga plagijator mora pronaći dovoljno tehnika prikrivanja s obzirom da nisu sve primjenjive na svim dijelovima izvornog teksta programa.

Naravno postoji cijeli niz napada tj. tehnika prikrivanja koje nemaju učinak pri korištenju metode otkrivanja koju koristi Jplag. Mnoge od tih metoda bi imale dosta utjecaja na tekstualno orijentirane alate za detekciju plagijata. Neki od tih napada su:

- Ubacivanje, promjena ili brisanje komentara.
- Izmjena teksta u I/O operacijama
- Izmjena imena varijabli, funkcija i klasa
- Spajanje i razdvajanje deklaracija varijabli u istoj deklaraciji (ista linija...)
- Ubacivanje, izmjena ili brisanje modifikatora (private, protected...)
- Promjena konstantnih vrijednosti

Postoji cijeli niz načina na koje se može postići "local confusion" kod JPlag programa, a njihova uspješnost ovisi o skupu tokena koji se koriste tj. o razini detaljnosti na kojoj se pregledava program. Sa manjim skupom tokena, više se različitih dijelova izvornog teksta programa označava istim tokenom, pa je i teže obaviti takvu promjenu koja će biti primjećena JPlagom. Time se naravno povećava mogućnost da neki dijelovi koji objektivno nisu plagijati budu procijenjeni kao isti.

Neke od metoda napada koje su bile uspješne kod tipičnog skupa tokena:

- Modifikacija kontrolnih struktura: npr. zamijena *for* petlje sa *while* petljom, ili zamijena *switch* izraza sa većim brojem *if* izraza.
- Privremene varijable i podizrazi: npr. uvođenje ili brisanje varijabli koje se koriste za pohranjivanje međurezultata kod računanja nekih izraza, ili promjena naredbe za inicijalizaciju određenog polja u pojedinačnu inicijalizaciju svakog elementa. Uglavnom, radi se o razdvajanju i spajanju dijelova računanja nekog izraza.
- Ubacivanje malih metoda u izvorni tekst programa, umjesto njihovih poziva
- Promjena dosega djelovanja poput pomicanja *try-catch* strukture prema rubovima metode umjesto da se iznimke hvataju oko dijela koji ih uzrokuju.
- Uvođenje namjernih grešaka u program
- Reorganiziranje strukture programa, npr. prebacivanje nekih članskih funkcija u pomoćne klase te pozivanje tih funkcija kao dijelova druge klase će uzrokovati manju ocjenu sličnosti.

2.6.6 Ukupni dojam

Program Jplag je alat za usporedbu izvornih tekstova programa i detekciju plagijata s mnoštvom mogućnosti. Najveći nedostatak je ograničen broj ulaznih jezika koje program podržava. Za podržane programske jezike omogućena je uspješna analiza razlika i sličnosti izvornih tekstova programa i čini se da je ovakav pristup puno napredniji od tekstualno orijentiranih metoda. Pitanje koje se postavlja je svrha za koju se taj program koristi. Ako se radi o usporedbi dva izvorna teksta programa (npr. nekih komercijalnih programa), a za jedan od njih se sumnja u plagijatorstvo, onda je

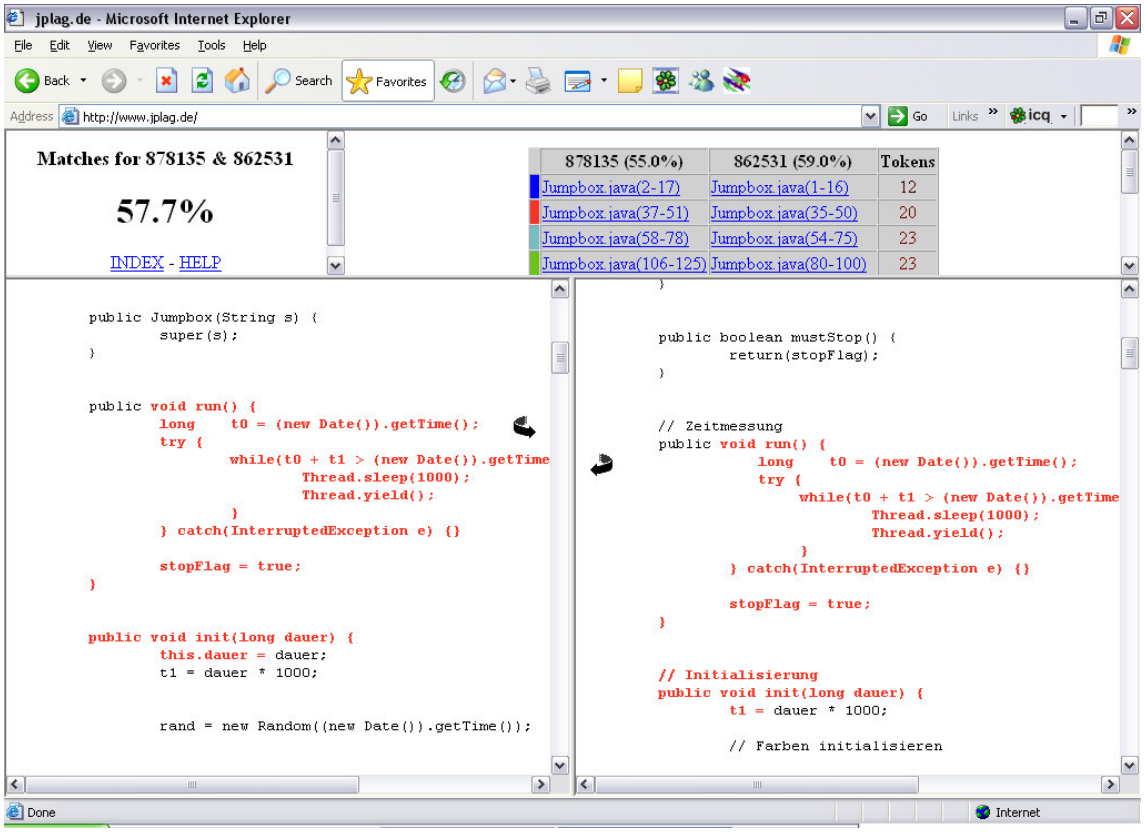
vrlo prikladan jer lako određuje koji dijelovi su kopirani, unatoč nekim trivijalnim promjenama koje plagijator napravi. U drugom slučaju, ako se koristi za usporedbu velikog broja izvornih tekstova programa koji imaju cilj obavljati isti posao(npr. studentski radovi) onda je pitanje koliko se mogu zamenariti one razlike koje nemaju utjecaja na stvaranje tokena. [9]

90% - 100%	2	#
80% - 90%	0	.
70% - 80%	1	#
60% - 70%	0	.
50% - 60%	1	#
40% - 50%	1	#
30% - 40%	3	#
20% - 30%	20	#
10% - 20%	430	##########################
0% - 10%	1195	##

Slika 2.3: Prikaz distribucije sličnosti izračunate uporabom Jplag programa

826764	->	826366 (100.0%)	
861641	->	861005 (98.4%)	861061 (25.0%)
943151	->	942261 (70.4%)	
878135	->	862531 (57.7%)	
861301	->	861196 (40.1%)	862246 (35.9%)
862564	->	862326 (35.9%)	
829683	->	826833 (30.6%)	
861061	->	827654 (25.1%)	

Slika 2.4: Prikaz sličnih programa (iznad sličnosti od 10%)



Slika 2.5: Istovremeni prikaz sličnih programa

2.7 Program za usporedbu izvornih tekstova programa *SIM*

Ovaj program nije izvorno nastao kao alat za detekciju plagijata, nego kao alat za pronalaženje istih dijelova izvornog teksta programa unutar velikih programa poput operacijskih sustava, kompilatora i slično, radi smanjivanja veličine izvornog teksta programa i optimiranja. Algoritam se bazira na pretvaranju izvornog teksta programa u "osnovne tokene", ali na drugačiji način od prethodno opisanog Jplag programa. Funkcije za pretvaranje u tokene se nalaze u zasebnom programskom modulu. One reduciraju sve identifikatore u jedan osnovni token <*idf*>, sve znakovne nizove u token <*string*> i jednostruke znakove (*character*) u <*char*>. Ostali elementi izvornog teksta programa poput zareza, točka-zareza, operatora itd. se prihvaćaju nepromijenjeni kao tokeni. Iz izvornog teksta se izbacuju svi elementi koji nemaju utjecaj na program, poput komenatra. Razmještaj (*layout*) se ne uzima u obzir, dakle, izbacuju se praznine, znakovi prijelaza u novi red i slično. Ovakav pristup zahtijeva da modul koji pretvara izvorni tekst programa u tokene u sebi sadrži leksički analizator za programski jezik u kojem su pisani uspoređivani izvorni tekstovi programa. Time je ovaj program vezan za samo jedan programski jezik, a za ostale bi trebalo prilagoditi taj front-end modul. Izvorni tekst programa veličine oko 2000 znakova će biti reduciran u otprilike 100 tokena. Nakon pretvorbe počinje usporedba zajedničkih dijelova.

Međutim, za svaki token vrlo je vjerojatno da će biti pronađen u oba izvorna teksta programa, budući se radi o ograničenom skupu ključnih riječi, znakova,

identifikatora i konstanti. Zato je potrebno odrediti minimalnu duljinu podniza tokena koji će se prihvatiti kao plagijarizirani dijelovi. Ako tu veličinu postavimo na premali iznos, dobit ćemo šum tj. uparivanje nekih elemenata koji su premali ili prejednostavni da bi bili smatrani kopijama. U suprotnom slučaju, ako taj minimum postavimo na prevelik iznos, neke sličnosti među programima neće biti prijavljene. Uz dobro defninirano stvaranje tokena i pravilno određenu duljinu minimalnog podniza, ova metoda prijavljuje većinu sličnosti koje imaju smisla. Ta duljina se kreće oko 24 tokena. Duljina manja od 18 tokena daje dosta sumnjivih prijava sličnosti dok duljina iznad 30 tokena zanemaruje neke bitne sličnosti poput kratkih *for* petlji i slično. Nizovi tokena se uspoređuju metodom najdužeg zajedničkog podniza (*longest common substring*), koja je neosjetljiva na razmještaj za razliku od metode najdužeg zajedničkog podslijeda (*longest common subsequence*).

Autor programa pretpostavlja da ukoliko i postoje uspješne metode napada, one su dovoljno komplicirane da je količina rada trošena u prikrivanje plagijata otprilike jednaka radu koji bi bio potreban za pisanje originalnog programa. Svi primjeri plagijata sa sličnosti preko 80% su pregledani "ručno" i utvrđeno je da su svi ti programi zaista plagijati. Niži postotak kritične sličnosti (60%) daje krive rezultate, pogotovo za kraće programe (jedna ili jedna i pol stranica). Problem je u tome da je teško ocijeniti da li je taj kratki program kopija ili se samo za 40% razlikuje od pretpostavljenog originala.

Ovakav pristup se čini vrlo uspješnim za detekciju plagijata. Jedina očita mana mu je vezanost za konkretni programski jezik i nemogućnost usporedbe ne-programskog teksta. Očito proširenje ovog programa bila bi prilagodba za razne programske jezike što bi zahtijevalo stvaranje leksičkih modula za svaki od njih. [7]

```
File 4b.c: 1493 tokens, 1 non-ASCII character
File lab4.2.c: 1238 tokens
Total: 2731 tokens

4b.c: line 214-221                      |4b.c: line 225-232                [56]
     if (pom->duljina<10000) {          |     if (pom->duljina>10000) {
    broj=(pom->duljina%10000)*sizeof(z  |      broj=(pom->duljina%10000)*sizeof
      printf("Blok %p: ",pom+1);        |      printf("Blok %p: ",pom+1);
      printf(" Velicina u blokovima:%l  |      printf(" Velicina u blokovima:%l
     }                                  |     }
     pom=pom->sljedeci;                 |     pom=pom->sljedeci;
    }                                   |    }
    while(pom!=NULL);                   |    while(pom!=NULL);

4b.c: line 190-199                      |lab4.2.c: line 193-204            [55]
 printf("2) Oslobodi \n");              |   printf("1) Dodjela bloka \n");
 printf("3) Ispisi \n");                |   printf("2) Oslobodi blok \n");
 printf("4) Kraj \n");                  |   printf("3) Kraj \n");
 while(1) {                             | while(1){
 printf("\nUpisi:");                    |
  izbor=getchar();                      | printf("\nUnos : ");
  switch(izbor){                        |  izbor=getchar();
  case '1': printf("\n Velicina memori  |
    scanf("%d",&i);                     |  switch(izbor){
   p=(zag *) dodjeli(i);                |  case '1': printf("\nKoliko memorije
                                        |    scanf("%d",&i);
                                        |    p=(tip *) dodjeli(i);

lab4.2.c: line 63-67                    |lab4.2.c: line 92-96              [41]
  // zauzima se memorija za novi blok   |   // zauzima se memorija za novi blok
  novi = (void *) sbrk(blok * sizeof(s  |   novi = (void *) sbrk(blok * sizeof(
                                        |
  // postavljanje pokazivaca novog blo  |   // postavljanje pokazivaca novog bl
  ((struct zaglavlje *)novi)->sljed =   |   ((struct zaglavlje *)novi)->sljed =

4b.c: line 200-207                      |lab4.2.c: line 206-213            [36]
    if (p==NULL){                       |    if (p==NULL){
     printf("\nNe moze se zauzeti memo  |    printf("\nNe moze se zauzeti memor
    }                                   |    }
    printf("\n");                       |    printf("\n");
    break;                              |    break;
  case '2': printf("Oslobodi blok memo  |  case '2': printf("Oslobodi blok memo
    scanf("%p",&k);                     |    scanf("%p",&k);
    oslobodi(k);                        |    oslobodi(k);

lab4.2.c: line 14-16                    |lab4.2.c: line 20-22              [30]
// funkcija postavlja prvi bit podatka  |// funkcija postavlja prvi bit podatka
void zauzmi (struct zaglavlje *pok) {   |void obrisi (struct zaglavlje *pok) {
 pok->duzina = pok->duzina | 0x8000000  | pok->duzina = pok->duzina & 0x7ffffff

lab4.2.c: line 151-152                  |lab4.2.c: line 161-162            [28]
 // obavlja spajanje sa prethodnim pra  | // obavlja spajanje sa sljedecim praz
if(tekuci->pret != NULL && !zauzeto(te  |if (tekuci->sljed != NULL && !zauzeto(
```

Slika 2.6: Izvještaj usporedbe programa *SIM*

2.8 Program za usporedbu izvornih tekstova programa *MOSS (Measure Of Software Similarity)*

MOSS (Measure Of Software Similarity) je program vrlo široko korišten za detekciju plagijata koji je dostupan kao internetska usluga od 1997. Metoda na kojoj se bazira ovaj program je računanje digitalnog "otiska prsta" za svaki izvorni tekst programa ili tekst. Program nije ovisan o vrsti programskog jezika, ali mu se mogu zadati neke jezično uvjetovane upute koje onda omogućavaju zanemarivanje klasičnih dijelova izvornog teksta programa koji nemaju utjecaja na smisao izvornog teksta programa kao što su razmaci i komentari. Najmanja jedinica izvornog teksta programa koja se proučava ovim programom se naziva k-gram i označava podniz od *k* slijednih znakova iz izvornog teksta programa. Iz izvornog teksta programa se dakle uzimaju svi podnizovi od *k* znakova. Primjerice za znakovni niz "danas je lijep dan", prvo se izbacuju praznine, po potrebi bi se sva slova pretvorila u mala, da dobijemo niz "danasjelijepdan". Ako pretpostavimo da je *k* jednak 4, skup svih *k*-grama bi bio: {dana,anas,nasj,asje,sjel,jeli,lije,ijep,jepd,epda,pdan}. O duljini k-grama ovisi naravno kakva će bit preciznost pronalaženja sličnost i količina šuma. Prag na koji treba postaviti vrijednost *k* mora biti takva da su sličnosti manje duljine od *k* nezanimljive za usporedbu. Tako je recimo nezanimljivo uočavanje da dva teksta na engleskom koriste riječ "the".

Ti k-grami se koriste dalje tako da im se izračuna vrijednost sažetka (*hash*). Tako za ovaj navedeni tekst dobijemo npr. sljedeći niz vrijednosti sažetka: {77, 72, 42, 17, 98, 50, 15, 95....} (vrijednosti su izmišljene). Algoritam pretpostavlja da je pretraživanje i uspoređivanje svih preopširno za izvođenje, pa se od svih vrijednosti sažetaka izaberu neke koje predstavljaju digitalni "otisak prsta" tog dokumenta koji se onda uspoređuje sa otiscima drugih dokumenata tj. izvornih tekstova programa. Razni pristupi koriste uspoređivanje vrijednosti sažetka, no postoje razlike u pristupu kako odabrati neke od vrijednosti sažetaka koje će stvoriti otisak specifičan za taj izvorni tekst programa. MOSS koristi "prozore" određene veličine *w*. Dakle, na isti način na koji su stvarani k-grami uzima se *w* uzastopnih vrijednosti sažetaka. Tako bi prozori izgledali(za *w*=4): (77, 72, 42, 17), (72, 42, 17, 98), (42, 17, 98, 50)...

Iz tih prozora se uzima najmanja vrijednost sažetaka, ukoliko se ne radi o istoj vrijednosti kao i u prošlom prozoru. Naime, u dva susjedna prozora često se desi da je ista vrijednost najmanja. Tada se ta jednaka vrijednost ne dodaje u skup otisaka. Ukoliko se u prozoru nađu dvije iste najmajne vrijednosti, u listu se dodaje vrijednost koja je najviše desno. Osim same vrijednosti sažetka, zapisuje se i mjesto iz koje potječe ta vrijednost. To je korisno radi određivanja o kojem dijelu izvornog teksta programa se radi. Time je završeno stvaranje otisaka za taj izvorni tekst. Taj postupak je nazvan "prosijavanje" (*winnowing*).

Dalje se uspoređuju otisci raznih dokumenata i korisniku se prijavljuju parovi dokumenata sa najviše istih otisaka. Do ovog trena nema provjere izvornog teksta programa, nego samo otisaka izvornih tekstova. [10]

3. Algoritmi i metode korišteni za detekciju plagijata

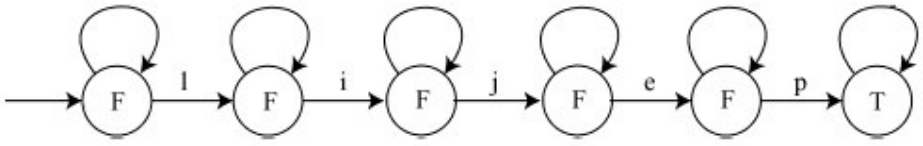
3.1 Uvod

U prethodnom poglavlju opisani su razni programi za detekciju plagijata te su navedeni općeniti principi na kojem oni djeluju. Nadalje će detaljnije biti opisani algoritmi koji su razvijeni u istu svrhu. Neki od njih se koriste u navedenim programima, a neki su još u razvoju. Oni se razlikuju po uspješnosti i po vremenu izvođenja. Što se tiče razlike u općem pristupu, imamo algoritme za manipulaciju i analizu bazirani na tekstu, algoritmi za odabir vrijednosti sažetka (*hash*) segmenata programa u svrhu stvaranja prepoznatljivih otisaka, te razne metode umetanja dodatnih informacija u izvorni tekst programa. Svaki od tih algoritama ima određene prednosti i mane, pa njihovu "dobrotu" treba promatrati u kontekstu za koji ih želimo koristiti.

3.2 Najduži zajednički podslijed *(Longest Common Subsequence)*

Najduži zajednički podslijed (Longest Common Subsequence) je jedan od poznatih algoritama za uspoređivanje znakovnih nizova i koristi se u nekim alatima za detekciju plagijata kao što je diff. Diff koristi ovaj algoritam za uspoređivanje dvije linije izvornog teksta programa. Treba razlikovati ovaj algoritam od algoritma za nalaženje najdužeg zajedničkog podniza.

Prije riješavanja ovog problema treba početi sa lakšim primjerom, a to je traženje uzorka (pattern) u duljem znakovnom nizu (text). Taj problem se riješava izgradnjom jednostavnog konačnog automata. Recimo da je uzorak kojeg tražimo u duljem nizu znakova "lijep", a da je dulji niz "danas je lijep dan". Tada naš automat za svako slovo iz uzorka prelazi u iduće stanje dok za sve ostale znakove ostaje u istom stanju. Ideja je pronaći uzorak u duljem tekstu, makar dijelovi uzorka nisu jedan do drugoga u tom duljem tekstu. Izlaz automata iz stanja u koje se dođe zadnjim slovom je pozitivan i označava da je pronađen uzorak u duljem tekstu.



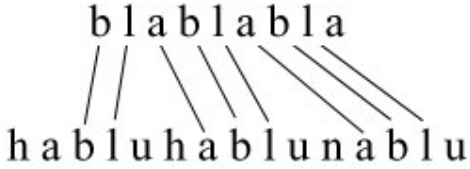
Slika3.1: DKA za detektiranje uzorka u znakovnom nizu.

```
podniz(char * P, char * T)
 {
        while (*T != '\0')
            if (*P == *T++ && *++P == '\0')
                return TRUE;
        return FALSE;
 }
```

Slika 3.2: Pseudokod za jednostavno detektiranje uzorka

Prvi idući problem na koji nailazimo u analizi teksta se pojavljuje ako dulji znakovni niz ne sadrži manji uzorak u cijelosti. U tom slučaju, tražit ćemo najdulje podslijedove koji se pojavljuju u oba niza. Taj problem se naziva "najduži zajednički podslijed" (longest common subsequence). Sada oba znakovna niza postaju ekvivalentna u značenju i možemo ih jednostavno nazivati niz A i niz B. Neka je *m* duljina niza A i *n* duljina niza B. Konačni automat iz prošlog primjera ne daje rješenje ovog problema, nego samo najveći zajedniči "prefiks", iako taj prefiks može biti nepovezan.

Riješenje problema leži u rekurzivnom algoritmu. Kao primjer ćemo uzeti dva znakovna niza: "blablabla" i "habluhablunablu". Ako napišemo ta dva niza jedan iznad drugoga, možemo linijama povezati slova koja su zajednička. Ako te linije vučemo tako da povezujemo redom slova iz prvog niza sa ekvivalentnim slovima koja također odabiremo po redu iz drugog niza, postići ćemo da se nijedna linija ne siječe, i ta uparena slova nazivamo podslijed (subsequence).



Slika3.3: Podslijed

Ako dva znakovna niza započinju istim slovom, najsigurnije je spojiti ta dva slova kao dio podslijeda. Ako se to prvo slovo podslijeda može naći na nekom desnijem mjestu od prvog, linija kojom se spajaju ta dva slova u oba niza se može preusmjeriti u lijevo bez da se uzrokuje presjecanje linija. To je sigurno jer se radi o prvom slovu. Ako se prva slova razlikuju, nemoguće je da će oba biti dio podslijeda, nego će biti potrebno bar jedno od njih (ili oba) ukloniti. Kada se algoritam obavi za prva dva slova (jednaka su ili se bar jedno uklanja) ostatak znakovnih nizova predstavljaju isti problem, samo se radi o kraćim nizovima. Rekurzivno riješenje je očito.

```
int nzp_rekurzivni (char * A, char * B)
    {
      if (*A == '\0' || *B == '\0') return 0;
      else if (*A == *B) return 1 + nzp_rekurzivni (A+1, B+1);
      else return max(nzp_rekurzivni (A+1,B), nzp_rekurzivni (A,B+1));
}
```

Slika3.4 Pseudokod rekurzivnog pronalaženja najduljeg zajedničkog podslijeda

Ovakvo rješenje zahtijeva dosta vremena, ako primjerice dva znakovna niza nemaju zajedničkih rješenja, zadnja linija se uvijek izvršava.

Greška u pristupu ovakvog riješenja je što se isti podproblemi pozivaju više puta (podproblem je izvršavanje algoritma nad istim podnizovima). Argumenti u podproblemima su sufiksi nizova A i B, dakle postoji maksimalno (m+1)(n+1) podproblem. Pokretanjem ove funkcije desi se 2^n poziva (za približno jednake m i n). Očito je da se funkcija poziva više puta za određene podprobleme. Ideja boljeg riješenja je samo jednom izračunati podproblem i zapisati "vrijednost", a kasnije samo pogledati vrijednost iz zapisa umjesto ponovnog izračunavanja. U rekurzivno riješenje se dodaje dio izvornog teksta programa koji to obavlja.

```
int nzp_rekurzivni (char * AA, char * BB)
    {
      A = AA; B = BB;
      for (i = 0; i <= m; i++)
          for (j = 0; j <= m; j++)
            L[i,j] = -1;

      return podproblem(0, 0);
    }
int podproblem(int i, int j)
    {
      if (L[i,j] < 0) {
          if (A[i] == '\0' || B[j] == '\0') L[i,j] = 0;
          else if (A[i] == B[j]) L[i,j] = 1 + podproblem(i+1, j+1);
          else L[i,j] = max(podproblem(i+1, j), podproblem(i, j+1));
      }
      return L[i,j];
}
```

Slika 3.5 Pseudokod poboljšane verzije rekurzivnog pronalaženja najduljeg zajedničkog podslijeda

Polje L služi za zapis riješenja određenog problema, te ukoliko je vrijednost negativna znači da još nije izračunata vrijednost za taj podproblem. Podproblem se sad poziva jednom iz glavne funkcije i najviše dva puta svaki put kada se popuni

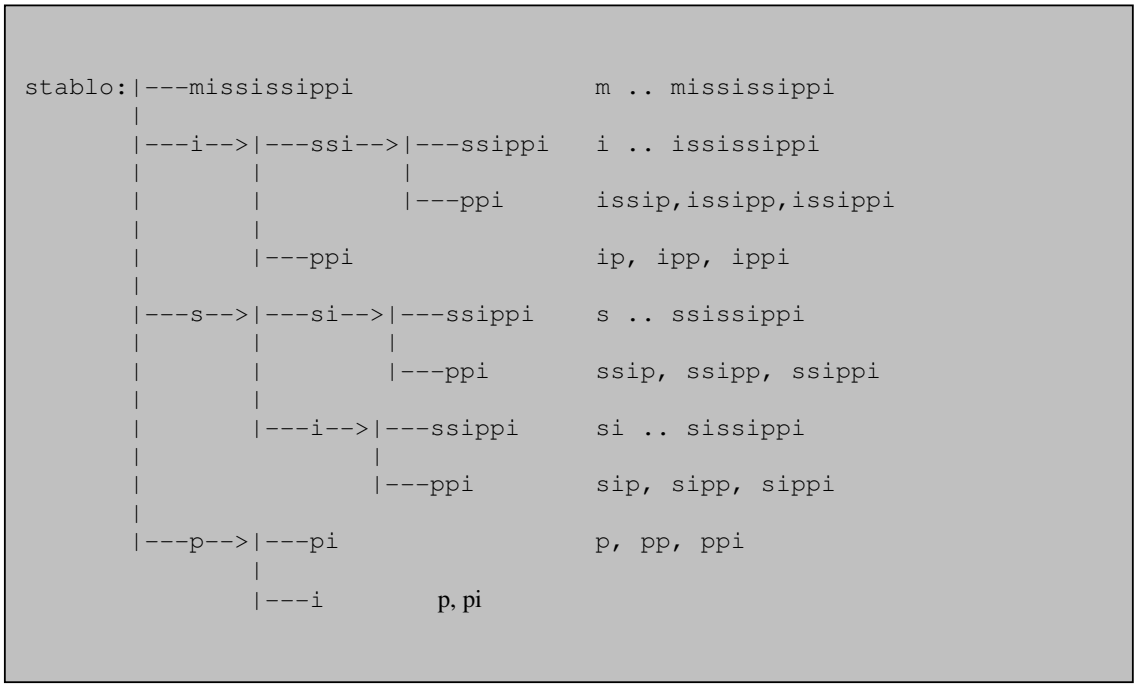
vrijednost niza L. Znači za (m+1)(n+1) podproblema, sada imamo 2(m+1)(n+1)+1 poziva funkcije za potproblem i složenost pada na O(mn). [7]

3.3 Najduži zajednički podniz (*Longest Common Substring*)

Ova metoda se razlikuje od prethodne po tome što se podniz smatra skup znakova iz originalnog znakovnog niza koji je povezan, dok se podslijedom smatra bilo koji podskup znakova iz originalnog niza koji se može dobiti brisanjem nula ili više znakova originalnog niza. Algoritam se temelji na korištenju stabala sufiksa. Stablo sufiksa je struktura podataka koja omogućava riješavanje raznih problema vezanih uz znakovne nizve u linearnom vremenu. Ako znakovni niz označimo sa str = $t_1t_2t_3...t_n$ onda je Ti = $t_i...t_n$ sufiks od str koji počinje na poziciji *i*.
Primjer:
T_1 = mississippi = str
T_2 = issisippi
T_3 = ssisippi
T_4 = sissippi
T_5 = issippi
T_6 = ssippi
T_7 = sippi
T_8 = ippi
T_9 = ppi
T_{10} = pi
T_{11} = i
T_{12} = (prazno)

Ako sortiramo te sufikse po slovima, možemo uočiti da neki od njih imaju zajedničke prefikse:
T_{11} = i
T_8 = ippi
T_5 = issippi
T_2 = ississippi
T_1 = mississippi
T_{10} = pi
T_9 = ppi
T_7 = sippi
T_4 = sissippi
T_6 = ssippi
T_3 = ssisippi

Stablo sufiksa se dobije tako da sufiksi sa zajedničkim prefiksom imaju zajednički korijen u stablu.



Slika 3.6 Stablo sufiksa

Najduži zajednički podniz se traži pomoću generaliziranog stabla sufiksa. Generalizirano stablo sufiksa sadrži sve sufikse više znakovnih nizova (izvornih tekstova programa) koje uspoređujemo. Čvorovi takvog stabla moraju biti označeni ovisno da li pripadaju prvom, drugom ili oba osnovna niza. Najveći zajednički podniz će biti najdublji čvor koji je označen kao dio oba osnovna niza. Drugi način je spajanje oba znakovna niza str1 i str2 u jedan "str1$str2#" gdje je "$" posebna oznaka kraja za str1 a "#" posebna oznaka kraja za str2. Najveći zajednički podniz u tom slučaju će biti čvor koji nije list, a ima "...$..." i "...#..."(bez $) ispod sebe. [9]

3.4 Metode odabiranja "otiska" iz skupa vrijednosti sažetaka

3.4.1 Odabiranje vrijednosti sažetka

Neki od programa za detekciju plagijata se oslanja na stvaranje relativno malih i prepoznatljivih otisaka za svaki izvorni tekst programa koji se uspoređuje. U opisu MOSS programa prikazan je jedan od načina na koji se odabire samo određeni broj k-grama iz ukupnog dobivenog skupa. Razlog tome je činjenica da bi skup svih vrijednosti sažetaka bio preglomazan, a usporedba dva takva skupa vremenski prezahtjevna, zbog čega bi se izgubila bilo kakva prednost ovakvog uspoređivanja. MOSS program, kao što je navedeno, odabire samo najmanje vrijednosti sažetaka iz svakog "prozora" i skup tako dobivenih vrijednosti smatra otiskom reprezentativnim za taj izvorni tekst programa. Evo još nekih strategija uzimanja otisaka:

3.4.2 Karp-Rabin uspoređivanje znakovnih nizova

Karp-Rabin algoritam za brzo uspoređivanje znakovnih nizova je jedna od prvih metoda za stvaranje "otisaka prsta" bazirana na k-gramima. Poticaj za razvoj ovog algoritma je bilo traženje uzoraka unutar nizova u genetičkim istraživanjima. Ideja je usporedba svih vrijednosti sažetaka k-grama, što je očito vrlo vremenski zahtjevno. Ovaj se algoritam na to ne obazire, nego pokušava smanjiti vrijeme potrebno da se izračunaju sve te vrijednosti za duge znakovne nizove unutar kojih se traži neki podniz. Predlaže se funkcija računanja sažetka koja omogućava da se i+1 vrijednost sažetka k-grama brzo izračunava iz vrijednosti vrijednosti sažetka i-tog k-grama. Neka je k-gram k-znamenkasti broj $c_{1...}c_k$ u nekoj bazi *b*. Kao funkcija računanja sažetka se uzima:

$$H(c_1...c_k) = c_1 * b^{k-1} + c_2 * b^{k-2} + c_3 * b^{k-3} + ... + c_k$$

Da bi se izračunala vrijednost sažetka idućeg k-grama moramo obaviti vrlo jednostavan račun:

$$H(c_2...c_{k+1}) = (H(c_1...c_k) - c_1 * b^{k-1}) * b + c_{k+1}$$

S obzirom da je b^{k-1} konstanta svaka iduća vrijednost se računa pomoću dvije operacije zbrajanja, i dvije operacije množenja. Te operacije se uzimaju "modulo" neku vrijednost, najčešće najveća vrijednost za cijeli broj (kompjuterski ograničena) tako da je ova metoda vrlo primjenjiva u standardnoj strojnoj aritmetici.

Slabost ovakve funkcije za računanje sažetka je što dodavanje c_i (obično se radi o maloj vrijednosti) utječe samo na niže bitove dobivene vrijednosti. Poboljšana verzija funkcije trebala bi biti sposobna potencijalno utjecati na sve bitove. Moguća funkcija koja bi imala to poboljšanje:

$$H'(c_2...c_{k+1}) = (H(c_1...c_k) - c_1 * b^{k}) + c_{k+1}) * b$$

3.4.3 Uspoređivanje "svi protiv svih"

Prvu strategiju za uspoređivanje većih količina dokumenata osmislio je Manber, koji je neovisno otkrio Kerp-Rabin algoritam, i upotrijebio ga za detekciju sličnosti datoteka. Za razliku od prošlog primjera želja je bila usporediti sve parove k-grama u većem skupu datoteka tj. tekstova. "Svi protiv svih" strategija je glavna teškoća u stvaranju otisaka. Ako gledamo datoteku teksta, postoji k-gram za svaki byte i potrebni su vrijednosti sažetka od 4 byta. Uzimanje svih tih vrijednosti bi stvorilo indeks veći od samih originalnih dokumenata. Zbog toga je potrebno na prikladan način odabrati manji podskup.

Vrlo jednostavna, ali i vrlo nepostojana metoda bila bi uzimanje svakog i-tog k-grama. Ovaj pristup je potpuno neotporan na premještanje, umetanje ili brisanje. Dodavanje samo jednog znaka na početak datoteke-plagijata pomiče sve k-grame za jedno mjesto, i i-ti k-grami se više neće poklapati.

Pristup koji je Manber odabrao je uzimanje onih vrijednosti koje su jednake *0 mod p*. Na taj način odabrane vrijednosti sažetaka su neovisne o položaju, i osigurava se neosjetljivost metode na premještanje dijelova izvornog teksta programa.

Heintze predlaže uzimanje fiksno određenog broja najmanjih vrijednosti kao otisak. Glavni nedostatak ovog pristupa je što pronalazi samo plagijate vrlo slične jedne drugima, a prilično je neefikasan za otkrivanje progama koji proširuju kopirane dijelove izvornog teksta programa. Ukoliko je izvorni tekst programa - plagijat puno veći od onog iz kojeg je uzet dio izvornog teksta programa, skup najmanjih vrijednosti će se vjerojatno dosta razlikovati. [10]

3.4.4 Ostale tehnike

Neki pristupi se ne baziraju na *k-gramima* nego na fiksnim segmentima određenima npr. prozorom koji uzima fiksni broj linija izvornog teksta programa. Takav pristup recimo ima Ctcompare program koji je već opisan.

3.5 Uporaba XML-a u usporedbi izvornih tekstova programa

3.5.1 XML u analizi izvornog teksta programa

Novi pristupi se orijentiraju na umetanje informacija o sintaksnoj strukturi izvornog teksta programa unutar samog izvornog teksta. Time se zadržava leksička struktura, a stvara se podloga za detaljnu analizu izvornog teksta programa. Teško je pronaći gotove alate za detekciju plagijata koji se koriste ovom metodom, ali postoji više izvora koji se bave tom tematikom.

3.5.2 Meta-razlikovanje (*meta-differencing*)

Ovakav naziv je odabran zbog dodatnih informacija o razlikama između dva izvorna teksta programa koje se mogu doznati jednostavnim upitima. Cijeli pristup je izgrađen nad XML prikazom izvornog teksta programa, nazvan *srcML*. Navedeni prikaz sadrži ugrađene sintakse informacije na taj način da se ne upliće u tekstualni kontekst izvornog teksta programa. Time je zadržan leksički "izraz" programera. Postoje gotovi alati poput *XQuery* i *XPath* koji omogućavaju dohvaćanje informacija iz izvornog teksta programa obogaćenih XML-om. Pomoću ovakvih programa, lako je utvrditi koje promjene su uvedene u izvornom tekstu programa.

Nakon ove pretvorbe u XML postoje razni načini uporabe informacija sadržanih u takvom izvornom tekstu programa. Meta-razlikovanje se u najnižem sloju oslanja na izvorne tekstove uspoređivanih programa i razlike dobivene jednostavnim programom za uspoređivanje *Diff*. Razlike su sadržane u tekstualnim datotekama. Sljedeći sloj obrade pretvara izvorni tekst programa i sadržaj datoteka sa razlikama u *srcML*. Najviši sloj aplikacije se oslanja na alate za manipulaciju XML-om da bi se dobio željeni izvještaj o sličnostima ili razlikama u izvornim tekstovima programa.

Za razliku od ostalih metoda, ovaj pristup ne zahtjeva izbacivanje razmaka, komentara, pretprocesorskih naredbi i sličnih elemenata iz izvornog teksta programa, jer oni ne utječu na način na koji se informacije crpe iz izvornog teksta. S obzirom da cijela struktura ostaje sačuvana, omogućena je obostrana transformacija između *srcML*-a i originalnog dokumenta s izvornim tekstom programa. Obavlja se jedino pretvorba nekih znakova koji su sastavni dijelovi XML-a poput "<". Oni se kodiraju, ali to ne utječe u nekom bitnom smislu na cijeli proces.

Prevoditelj za prevođenje jezika C++ u *srcML* dopušta unošenje XML oznaka oko dijelova izvornog teksta programa koji nas zanimaju, kao što su početak i kraj bloka, ili imena identifikatora. Dijelovi koji su nezanimljivi poput operatora u izrazima se ne označavaju XML oznakama. Ovakvo parsiranje omogućava umetanje XML-a u dobro-formirane izvorne tekstove programa, npr. sa dobro raspoređenim vitičastim zagradama (C++) koji su nepotpuni i koje se ne može kompilirati. Parsiranje se izvodi metodom rekurzivnog spusta. Postoje više dostupnih programa, koji su obično nadogradnje kompilatora za razne programske jezike, i čiji izlazi su XML prikazi izvornog teksta programa. Naravno, postoje razlike između takvih prikaza, sa razlikama u svojim oznakama i dijelovima izvornog teksta programa koje označvaju (npr. *GCC-XML*, *CPP2XML*...).

Meta-diferenciranje se koristi *srcDiff* formatom (XML verzija izlaza iz *Diff* programa). Taj format se dobiva kao kombinacija *srcML* verzija izvornih tekstova programa koje uspoređujemo upotrebljavajući izlaz iz *Diff* programa za kontroliranje razlika. U *srcDiff* formatu su označeni dijelovi koji su isti, izmijenjeni i dodani (obrisani). Sadržani su dijelovi iz oba izvorna teksta programa koje uspoređujemo. Ti dijelovi se ne smiju preklapati, svaki XML element mora početi i završiti prije nego drugi počne (a ne završi unutar nad-elementa). Dozvoljeno je ugnježđivanje. Dozvoljeno je da ista XML oznaka zatvara dvije jednake početne oznake.

Dobiveni format se može dalje koristiti za analizu tog izvornog teksta programa. Najveći nedostatak je korištenje linijski granuliranog programa *Diff*, zbog čega će se u *srcDiff* formatu naći više razlika nego što je potrebno (zbog neefikasnosti *Diff*-a). [14]

4. Programsko ostvarenje algoritma Ratcliff/Obershelp

4.1 Uvod

Ovaj algoritam se bazira na takozvanom "*Gestalt pristupu*". Taj pojam ima izvor u psihologiji, ali je primjenjiv u teoriji raspoznavanja uzoraka. Gestalt opisuje kako ljudski mozak prepoznaje uzorak unutar skupa elemenata i shvaća ga kao funkcionalnu cjelinu koja se ne može predstaviti kao suma svojih dijelova. Primjer koji se spominje za prikazivanje *Gestalt* pristupa je mogućnost čovjeka da prepozna sliku u vizualnoj zagonetci gdje je potrebno spojiti sve točke da bismo dobili sliku. Ljudski mozak popunjava dijelove koji mu nedostaju u slici i dobiva informaciju o čemu se radi. Na isti se način *Ratcliff/Obershelp* algoritam odnosi prema prepoznavanju uzoraka u tekstu.

Ratcliff/Obershelp se bavi usporedbom jednodimenzionalnih nizova znakova te vraća vrijednost usporedbe koja se uzima kao mjera ili postotak sličnosti. Mjera sličnosti se dobiva kao dvostruki broj zajedničkih znakova unutar znakovnih nizova podijeljena sa brojem ukupnih znakova tih nizova.

Algoritam su razvili W. Ratcliff i John A. Obershelp 1983. godine. Povod i potreba za razradu takvog algoritma nalazio se u razvoju edukacijskog softvera koji se djelomično bazirao na sustavu pitanja i odgovora koji su bili upisivani. Jasno je da je više takvih odgovora moglo biti točno, pa je bilo potrebno razviti algoritam koji prepoznaje "dovoljno" točne odgovore. *Ratcliff/Obershelp* algoritam uspoređuje dan odgovor sa potpuno točnim odgovorom, i može zanemariti sitne greške u pravopisu i slično. Osim ovakve primjene, algoritam je našao praktičnu primjenu u raznim komercijalnim softverima koji se oslanjaju na baratanju znakovnim nizovima poput softvera za provjeru pravopisa, programi za pretraživanje baza podataka (riječnici, tražilice itd.), jezičnim prevoditeljima i slično...

Veliki pomak korištenjem ovog algoritma je postignut baš u jezičnim prevoditeljima (kompilatorima). Primjerice u jezčnom prevoditelj za programski jezik C, *Ratcliff/Obershelp* algoritam je omogućio prepoznavanje pogrešno napisanih imena varijabli, funkcija i ključnih riječi. Krivo napisanu riječ prevoditelj bi ovim algoritmom usporedio sa postojećim deklariranim varijablama i funkcijama i time dobio mjere (postotke) sličnosti sa svim tim riječima. Najsličniju od njih bi predložio programeru kao moguću izmjenu, a u slučaju da nije moguće naći riječ sa mjerom sličnosti većim od nekog graničnog (npr. 60%) prevoditelj bi prijavio grešku.

Primjena se može naći i u softverima koji se bave analizom teksta. Riječi postižu razne jezične promjene kada se koriste u običnom (neprogramskom) tekstu. Korištenjem ovog algoritma, i usporedbom sa određenom bazom podataka (riječnikom) može se sa određenim faktorom sličnosti odrediti o kojoj se riječi radi. (npr. prepoznavanje "automobil", automobilA", "automobilOM" ...)

U ovom seminarskom radu, Ratcliff/Obershelp algoritam je iskorišten za usporedbu programskih izvornih tekstova programa. Iako postoje određene slabosti algoritma, uspješnost usporedbe se čini dobra za ovu svrhu.

4.2 Prikaz rada algoritma

4.2.1 Opis rada

Ratcliff/Obershelp algoritam prima dva znakovna niza (stringa) A i B kao parametre. Unutar njih se traži najdulji zajednički podniz. Treba naglasiti da se ne radi o najdužem zajedničkom "podslijedu", jer se podrazumijeva da je taj najdulji znakovni niz neprekinut. Taj dio koji je zajednički u oba niza se koristi kao povezište. Povezište se više ne koristi za usporedbu i broj znakova koje ono sadrži se pamti, tj. prenosi u daljnji rad algoritma. Povezište dakle dijeli osnovne znakovne nizove na dva dijela, lijevi i desni. Ti dijelovi mogu biti i duljine nula. Time su stvorena zapravo dva podproblema prema kojima se upotrebljava algoritam na jednak način kao i prema osnovnim nizovima A i B. Parovi lijevih i desnih strana se stavljaju na stog i procedura se ponavlja dok ne nestane znakova za usporedbu.

4.2.2 Primjer izvođenja

Kao primjer uzmimo dvije duge riječi koje imaju određene zajedničke dijelove kako bi prikazali rad algoritma. Neka originalna riječ bude "**ortorinolaringologija**", a druga (krivo napisana ili plagijat) "**puertoricolaringalokija**". Algoritam ide tražiti najdulji podniz jednak u oba početna znakovna niza. To je u ovom slučaju podniz "**olaring**". To je povezište, duljine sedam znakova i dalje se ne uspoređuje. To povezište je podijelilo početne nizove na lijeve i desne strane. Lijeve strane su "**ortorin**" i "**puertoric**", a desne "**ologija**" i "**alokija**". Te dvije grupe se sad nalaze na stogu i nad njima se obavlja ista procedura.

Pretpostavimo da su lijeve strane na vrhu stoga. Njihov najveći zajednički podniz je "**rtori**". To postaje povezište, broj znakova se dodaje broju zajedničih znakova i taj dio se ne uspoređuje dalje. Sada su lijeve strane podijeljene na dva dijela "**o**" i "**pue**" te "**n**" i "**c**". Ovi skupovi se stavljaju na stog, ali u idućem koraku više nema zajedničkih znakova, i ova grana algoritma se završava.

Na stogu je sada ostala desna strana dobivena u prvom koraku algoritma, "**ologija**" i "**alokija**". Traži se najdulji zajednički podniz koji je u ovom slučaju "**ija**". Treba uočiti da se povezište sad nalazi sasvim desno i na stog će se staviti samo lijeve strane "**olog**" i "**alok**". Zajednički dio je "**lo**". U idućij iteraciji nema više zajedničkih znakova niti na jednoj strani i algoritam završava.

Mjera usporedbe koja se dobiva kao rezultat usporedbe je dvostruki zbroj duljina svih povezišta koja su određena u koracima algoritma, podijeljen zbrojem duljina oba početna niza. Zbroj duljina oba početna niza je 44, a zbroj duljina svih povezišta je 17. Na navedeni način, dobiva se sličnost od 77%.

Korak	Trenutni	Povezište	stog	Zajednički
0.	ortorinolaringologija, puertoricolaringalokija	-	ortorinolaringologija, puertoricolaringalokija	0
1.	ortorinolaringologija, puertoricolaringalokija	olaring	ortorin,puertoric ologija,alokija	7
2.	ortorin, puertoric	rtori	o,pue **-X** n,c **-X** ologija,alokija	12
3.	ologija, alokija	ija	olog, alok	15
4.	olog, alok	lo	o,a **-X** g,k -**X**	**17**

Tablica 4.1 Izvršavanje algoritma Ratcliff/Obershelp

4.3 Opis programskog rješenja algoritma

Dva znakovna niza koji se uspoređuju mogu se predstaviti kao "problem" koji se rješava. U svakom koraku algoritma dobivamo dva nova podproblema nad kojima se obavljaju iste postupke kao i nad početnim ulaznim problemom. Algoritam se sastoji od dvije funkcije. Pomoćna funkcija obavlja pripremne radnje, poput određivanja duljine niza. Ideja je naći sve podnizove zadana dva znakovna niza kako bi se svaki sa svakim mogao usporediti. Da bi se izbjeglo kopiranje dijelova niza, ulazni znakovni nizovi se globalno čuvaju izvan funkcija, a u glavnu funkciju se prenose samo indexi početka i kraja podniza koji se koristi.

```
pomoćna_funkcija()
   učitaj_nizove A, B;
   ako je A = B rezultat = 100;
   inace
         ako A=prazan ili B=prazan onda rezultat = 0;
             pronadi_sve_podnizove_od_A();
             pronadi_sve_podnizove_od_B();
             pozovi_glavnu_funkciju();
             izračunaj_rezulatat();
izadi iz funkcije

glavna_funkcija()
     ako smo došli do kraja izadi_iz_funkcije();
     za c1 = 1 do duljina_niza_A;
       za c2 = 1 do duljina_niza_B;
             i = 0;
             dok podniz(c1 + i) == podniz(c2 + 1)
                   i = i + 1;
                   ako je i > max tada
                         ns1 = c1;
                         ns2 = c2;
                         max = i;
                   ako (c1 + 1) > duljina(A) ili (c2 + 1)>duljina(B);
                   izadi iz petlje;
             nastavi petlju;
     max = max + glavna_funkcija(ns1 + max, duljina_niza_A, ns2 + max,
duljina_niza_B);
     max = max + glavna_funkcija(st1, ns1 - 1, st2, ns2 - 1);
     vrati max;
izadi iz funkcije
```

Slika 4.1: Pseudokod algoritma

4.4 Programska implementacija

U okviru ovog seminara, izrađena je implementacija Ratcliff/Obershelp algoritma koja uspoređuje velik broj zadanih ulaznih izvornih tekstova programa. Program je ostvaren u programskom jeziku C++. Pomoćna funkcija sadržava preinaku da umjesto znakovnih nizova za usporedbu prima imena datoteka koje treba uspoređivati. Datotekama se određuje duljina, i čitaju se izvorni tekstovi programa iz njih. Nakon toga se poziva funkcija Ratcliff(...) koja obavlja rekurzivni postupak koji je opisan u prethodnom tekstu.

```
double fileExtractor::Compare(char *first,char *second)
{
      int i,j;
      double retval;
      int l1,l2;

      if(!checked){
      ifstream prva(first, ios::in|ios::binary|ios::ate);
      size1 = prva.tellg();
      prva.seekg (0, ios::beg);
      delete [] str1;
      str1 = new char [size1+1];
      prva.read (str1, size1);
      prva.close();
      str1[size1]='\0';
      }
      ifstream druga(second, ios::in|ios::binary|ios::ate);

      size2 = druga.tellg();
      druga.seekg (0, ios::beg);
      delete [] str2;
      str2 = new char [size2+1];
      druga.read (str2, size2);
      druga.close();
      str2[size2]='\0';

      if(size1==0 || size2==0) retval=0;
      else
      {
            retval=100*((double)Ratcliff(0,size1,0,size2))/((size1) +
(size2))*2;
      }
      return retval;
}
```

Slika 4.2: Pomoćna funkcija

```
long fileExtractor::Ratcliff(long st1, long end1, long st2, long end2)
{
      long i,j,cnt,max,ns1,ns2;
      max=0;
      ns1=0;
      ns2=0;
      if((st1 >= end1) || (st2 >=end2) || (st1 < 0) || (st2 < 0))
            {
                  max=0;
            }
      else
      {
            for(i=st1;i<end1;i++)
                  {
                        for(j=st2;j<end2;j++)
                        {
                              cnt=0;

                              while(str1[i+cnt]==str2[j+cnt])
                              {
                                    cnt++;

                                    if(cnt > max)
                                    {
                                          ns1=i;
                                          ns2=j;
                                          max=cnt;

                                    }
                                    if(((i+cnt)>=end1) || ((j+cnt)>=end2))
                                    {
                                          break;
                                    }
                              }
                        }
                  }
                  if(max>0)
                  {
                  max=max + Ratcliff(ns1+max,end1,ns2+max,end2);
                  max=max + Ratcliff(st1,ns1,st2,ns2);
                  }
      }

      return max;
}
```

Slika 4.3: Glavna funkcija

Time je prikazana implementacija samog algoritma, a nadalje će biti objašnjen kontekst programa u kojem se ta implementacija koristi. Program je napravljen kao konzolna aplikacija u programu C++. Ideja je bila prenosivost programa na razne operacijske sustave, pa je uporabljen jezik koji se vrlo često koristi i prevodiv na većinu platformi. U početku je isprogramirana i verzija programa u programskom jeziku Java, vrlo zanimljiva zbog prenosivosti izvodivog koda na razne sustave, ali je ta ideja ipak odbačena zbog sporosti izvođenja.

Program kao parametar prima samo ime konfiguracijske datoteke u kojoj su zapisane sve postavke koje određuju način rada.

```
POPIS STARIH ULAZNIH PATHOVA:
in1ipo.txt
POPIS NOVIH ULAZNIH PATHOVA:
in.txt
DATOTEKA SA STARIM REZULTATIMA:
done.txt
PATH NOVIH REZULTATA:
donenew20.txt
KRITICNA SLICNOST:
80
SAMO USPOREDBA NOVIH(0 za da, 1 za usporedbu novih sa starima):
0
PATH HTML-a SA TABLICOM:
new20.html
NAPRAVI TOP LISTE?(0 za da, ostalo za ne):
0
PATH HTML-a SA LINKOVIMA NA TOP LISTE:
popis20.html
DIREKTORIJ SA TOP LISTAMA:
html20
BROJ PRVIH NA TOP LISTAMA:
10
DATOTEKA SA SPOJENIM STARIM I NOVIM ULAZIMA:
inspojeni20.txt
DATOTEKA ZA UPIS VREMENA IZVRSAVANJA:
vrijeme20.txt
```

Slika 4.4: Konfiguracijska datoteka

Program ima dva načina rada. Koji će se koristiti ovisi da li se program izvodi prvi put za ulazni skup datoteka ili postoje stari rezultati. Taj način rada se određuje promjenom vrijednosti u liniji:

SAMO USPOREDBA NOVIH (0 za da, 1 za usporedbu novih sa starima):0

Ostale linije uglavnom služe za određivanje lokacija raznih ulaznih i izlaznih datoteka. Ulazne datoteke su: "popis novih datoteka" za usporedbu, "popis starih datoteka" za usporedbu i "datoteka starih rezultata". Zadnje se dvije navedene koriste samo ako je vrijednost u konfiguracijskoj datoteci (navedena iznad) postavljena na 1. Ako i nije, ta linija mora postojati. U datoteci starih rezultata se nalaze imena datoteka koje su već uspoređene, indeksi koje te datoteke imaju u popisu starih datoteka i vrijednost sličnosti za svaki par tih datoteka.

Ako se radi u načinu rada gdje se starim rezultatima žele dodati novi, kombiniraju se "popis starih datoteka" i "datoteka starih rezultata". Ulazi zapisani u "popisu novih datoteka" se provode kroz Ratcliff/Obershelp algoritam. Svi se rezultati spajaju u formiranju ukupnog izvještaja.

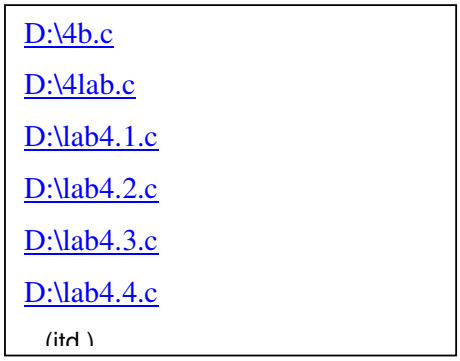
Drugi način rada čita samo "popis novih datoteka" i provodi te datoteke kroz Ratcliff/Obershelp algoritam, te formira izlazni izvještaj. Provedba algoritma bi se idealno trebala izvoditi svaki-protiv-svakog ulaznog izvornog teksta, ali to bi zahtijevalo previše vremena. Dakle za dva izvorna teksta programa A i B uspoređuju se samo A i B, a ne još i B i A. Iako zamjena poretka najčešće nema previše utjecaja, ipak ovo predstavlja određenu slabost programa.

Kako program ne sadržava grafičko sučelje za prikazivanje rezultata programa, jedan od izlaza sadrži HTML tablicu u kojoj se mogu vidjeti rezultati usporedbi svih ulaznih datoteka s izvornim tekstovima programa. Usporedbe koje imaju rezultat veći od vrijednosti koja je zadana kao kritična, označena je crvenom bojom.

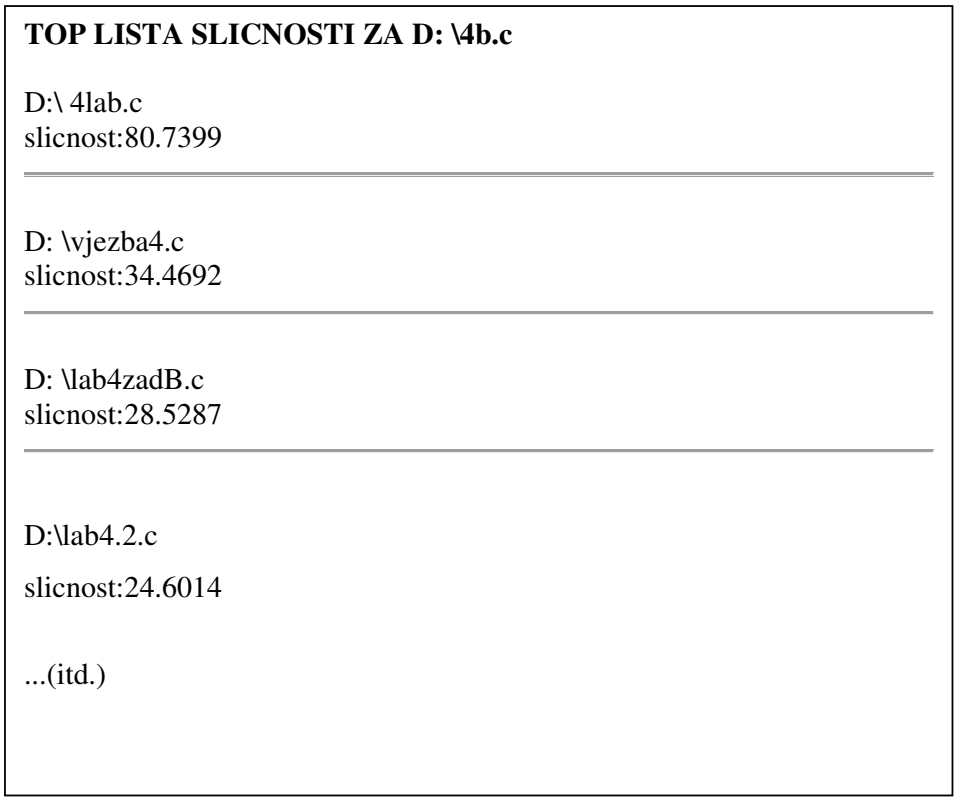
*	**1.)**	**2.)**	**3.)**	**4.)**	5.)	6.)	7.)	8.)	9.)	10.)	11.)	**12.)**	13.)	14.)	15.)	16.)	17.)	18.)	**19.)**
1.)4b.c	0	**80**	16	24	13	20	11	13	20	16	28	20	15	15	13	14	12	34	21
2.)4lab.c	**80**	0	9	17	9	17	6	13	17	14	25	20	14	15	12	16	5	26	19
3.)lab4.1.c	16	9	0	**84**	74	12	17	12	6	19	18	18	14	17	61	13	16	19	18
4.)lab4.2.c	24	17	**84**	0	75	11	18	11	8	17	15	19	19	18	63	12	15	21	19
5.)lab4.3.c	13	9	74	75	0	14	18	10	10	16	16	17	19	19	62	12	15	20	16
6.)lab4.4.c	20	17	12	11	14	0	9	23	16	16	14	21	20	17	15	8	14	23	20
7.)lab4.5.c	11	6	17	18	18	9	0	5	8	13	12	16	5	14	13	13	11	8	11
8.)lab4.6.c	13	13	12	11	10	23	5	0	16	12	18	24	26	15	14	13	21	23	23
9.)lab4.c	20	17	6	8	10	16	8	16	0	14	18	17	22	22	9	16	18	19	18
10.)lab4c.c	16	14	19	17	16	16	13	12	14	0	12	21	13	21	8	16	10	18	15
11.)lab4B.c	28	25	18	15	16	14	12	18	18	12	0	9	24	17	5	19	16	24	19
12.)lab4.c	20	20	18	19	17	21	16	24	17	21	9	0	15	17	8	11	16	17	**88**
13.)labos4.c	15	14	14	19	19	20	5	26	22	13	24	15	0	17	10	14	18	24	23
14.)malloc.c	15	15	17	18	19	17	14	15	22	21	17	17	17	0	20	14	13	16	19
15.)prvi.c	13	12	61	63	62	15	13	14	9	8	5	8	10	20	0	9	12	9	10
16.)vj4.c	14	16	13	12	12	8	13	13	16	16	19	11	14	14	9	0	13	12	9
17.)vj4_3.c	12	5	16	15	15	14	11	21	18	10	16	16	18	13	12	13	0	9	19
18.)vjezba4.c	34	26	19	21	20	23	8	23	19	18	24	17	24	16	9	12	9	0	14
19.)zad1.c	21	19	18	19	16	20	11	23	18	15	19	**88**	23	19	10	9	19	14	0

Tablica 4.2 Rezultati usporedbe pomoću Ratcliff/Obershelp algoritma

Za veliki broj ulaznih datoteka, ova tablica postaje velika i nepregledna. Za taj je slučaj bolje koristiti samo najlijeviji stupac koji označava kritične datoteke, i njih provjeravati u HTML popisu datoteka. Taj popis za svaku ulaznu datoteku sadrži linkove na top listu prvih *n* najsličnijih.



Slika 4.5: Datoteka s popisom



Slika 4.6: Datoteka s listom najsličnijih izvornih tekstova programa

4.5 Slabosti Ratcliff/Obershelp algoritma

Budući je ovaj algotiram tekstualno orijentiran, za detekciju plagijata potrebno je prvo provesti neke predradnje nad ulaznim izvornim tekstovima programa. Najbolje bi bilo izbaciti sve višestruke razmake, komentare i pretvoriti sva slova u mala (ili velika). No, čak i nakon takve obrade, algoritam i dalje ostaje ranjiv na razmještanje teksta. Najbolje je problem prikazati primjerom:
Neka su ulazni nizovi:

"AAAAAxcccxcccxcccxccc"
"cccycccycccycccyAAAAA"

Vidi se da su ovi nizovi jednaki u svim "ccc" dijelovima, i u dijelu "AAAAA". Zapravo, jedine razlike su "x" i "y", koje smatramo pokušajem plagijatora da zamaskira izvorni tekst programa. No, algoritam radi tako da prvo traži najveći podniz koji je zajednički, i na tom mjestu "reže" ulazne znakovne nizove. Dalje uspoređuje lijeve i desne strane, što u ovom slučaju uzrokuje problem. Najveći zajednički podniz je "AAAAA". Rezanjem znakovnih nizova na tom mjestu dobivamo:

lijevo:"", desno:"xcccxcccxcccxccc"
lijevo:"cccycccycccycccy", desno:""

Daljnje rekurzivno pozivanje funkcije će pokušati usporediti prazne nizove sa dijelovima koji sadrže puno "ccc" dijelova. Tu očito neće biti pronađena nikakva sličnost, i algoritam će stati. Time neće biti primjećena sličnost velikog dijela izvornog teksta programa.

Premještanje dijelova izvornog teksta programa, je očito učinkovit napad protiv Ratcliff/Obershelp algoritma. Osim toga, sve promjene teksta koje se ne uklanjaju pripremnim radnjama utječu na mjeru sličnosti koju vraća ovaj algoritam. Promjena imena varijabli, imena funkcija i znakovnih nizova koji se ispisuju na ekran smanjuju prijavljenu sličnost. To bi se moglo zaobići uporabom leksičkog analizatora koji bi zamijenio varijable i imena funkcija sa određenim znakom identifikatora, a sve znakovne nizove istom oznakom za "string", ali to bi zahtijevalo kompeksniji program.

Zbog navedene slabosti, postoje razlike u rezulatima kad se zamijene znakovni nizovi koji se uspoređuju. Ako postoje dva zajednička dijela maksimalne duljine, algoritam uzima prvog na koji je naišao.

Primjerice:

"AAAAblablablaBBBB"
"BBBBcccAAAAblaccc"

Ako se uspoređuje u ovom poretku, prvi najdulji podniz će biti "AAAA", zbog ugnježđenih petlji koje pregledavaju sve moguće podnizove. Nakon toga će se usporedbom desnih strana pronaći još i zajednički dio "bla". Sve ukupno, sedam zajedničkih znakova.

Ako zamijenimo poredak ulaznih znakovnih nizova:

"BBBBcccAAAAblaccc"
"AAAAblablablaBBBB"

Sada će se najprije kao najdulji podniz prepoznati "BBBB" jer unutarnja petlja pomiče kazaljku po znakovima drugog znakovnog niza. Nakon određivanja tog povezišta, lijeva strana prvog i desna strana drugog niza će biti prazni nizovi, i algoritam staje. Ukupan broj prijavljenih zajedničkih znakova je četiri.

Ovaj problem na testnim primjerima nije stvarao velike razlike u usporedbama, uglavnom na razini par slova, čime se stvarao samo manji šum.

5. Zaključak

U ovom seminarskom radu prikazani su razni alati za detekciju plagijata, te temeljni principi na kojima ti alati rade. Postoje razni pristupi, neki od njih se oslanjaju na sam tekst, neki se oslanjaju na strukturu programa dobivenu korištenjem dijelova jezičnih prevoditelja, a neki pokušavaju umetati dodatne informacije u izvorni tekst programa kako bi se poslije mogli analizirati. Svaki od tih pristupa može birati razne metode koje će koristiti. Tako postoje različite metode za usporedbu znakova u tekstu (najveći zajednički podslijed, najveći zajednički podniz, Ratcliff/Obershelp algoritam...), razni načini odabira vrijednosti dobivenih korištenjem funkcija za računanje sažetaka nad dijelovima teksta itd.

Postoje zahtjevi koji se odnose na neosjetljivost programa za detekciju plagijata na nebitne dijelove izvornog teksta programa. Neki od pristupa su sami po sebi neosjetljivi na to, jer iz cijele stukture izabiru dijelove koji su im bitni za usporedbu, a ostali se oslanjaju na normalizaciju izvornog teksta programa. Normalizacija je postupak brisanja komentara, viška praznina i oznaka novog reda. U velikom broju programskih jezika su komentari označeni na isti način, u suprotnom potrebno je prilagođavati postupak normalizacije specifičnom jeziku.

Opisani programi i algoritmi koriste se i za druge svrhe, i postoje razne prednosti svakog od njih. Neki uspješni za detekciju plagijata izvornog teksta programa su potpuno neupotrebljivi za detekciju plagijata običnog teksta. Obrnuto, programi dobri u otkrivanju plagijata običnog teksta su vrlo ranjivi na preinake izvornog teksta programa koje iz programerskog pogleda ne predstavljaju nikakvu promjenu. Teško je odrediti "najbolji" pristup, potrebno je najprije proučiti o kakvoj detekciji plagijata se radi i na temelju toga se odlučiti za najprimjereniji.

6.Literatura

[1] Complite File Comparison Family, grupa alata za usporedbu izvornih tekstova. Dostupno na web stranici http://world.std.com/~jdveale/

[2] CtCompare – C Token Comparator. Dostupno na web stranici http://minnie.tuhs.org/Programs/Ctcompare/

[3] ExamDiff Pro, opis alata za linijsku usporedbu izvornih tekstova programa. Dostupno na web stranici http://www.prestosoft.com/ps.asp?page=edp_examdiffpro

[4] Compserf, opis jednog od alata za usporedbu izvornih tekstova programa iz skupine Complite File Comparison Family. Dostupno na web stranici http://world.std.com/~jdveale/#compserf

[5] Comparator, opis alata za usporedbu izvornih tekstova na temelju vrijednosti sažetaka. Dostupno na web stranici http://www.catb.org/~esr/comparator/comparator.html

[6] Lutz Prechelt, Guido Malpohl, Michael Phlippsen "JPlag: Finding plagiarisms among a set of programs" Fakultaet fur Informatik, Universitat Karlsruhe http://page.mi.fu-berlin.de/~prechelt/Biblio/jplagTR.pdf

[7] Dick Grune, Matty Huntjens Vakgroep, "Detecting copied submissions in computer science workshops", Informatica Faculteit Wiskunde & Informatica,Vrije Universiteit, 1989 http://www.cs.vu.nl/~dick/sim.html

[8] Design and Analysis of Algorithms, opis algoritama za pronalaženje najduljeg zajedničkog podslijeda. Dostupno na web stranici http://www.ics.uci.edu/~eppstein/161/960229.html

[9] Suffix trees, opis pronalaženja najduljeg zajedničkog podniza pomoću stabala sufiksa. Dostupno na web stranici http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Tree/Suffix/

[10] Saul Schleimer, Daniel S. Wilkerson, Alex Aiken, "Winnowing: Local Algorithms for Document Fingerprinting" http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf

[11] Plagiarism, web stranice sa linkovima i informacijama vezanim uz plagijate i njihovu detekciju. Dostupno na http://www.web-miner.com/plagiarism

[12] Igor Mihaljko, "Neizrazita Logika: Algoritam Ratcliff/Obershelp

[13] Unix "man" stranice

[14] Jonathan I. Maletic, Michael L. Collard, "Supporting Source Code Difference Analysis"