Debreceni Egyetem

Informatikai Kar

Forráskód hasonlóság

Témavezető

Noszály Csaba

egyetemi tanársegéd

Készítette:

Ari Gábor

programtervező informatikus(BSc.)

Debrecen

2011

Tartalomjegyzék

I. Bevezetés

Szakdolgozatom témája a mindennapi életben folyamatosan használt forráskód hasonlóság ellenőrző algoritmusok bemutatása. A különböző algoritmusok működésének ismertetése, illetve a valós életbeli használhatóságának szemléltetése.

Napjainkban mivel egyre több hallgató van a felsőoktatásban, beleértve leginkább az informatikus hallgatókat, évről évre szükséges a hallgatók számonkérése. Az egyre növekvő hallgatói létszám pedig elengedhetetlenné teszi azt, hogy ki lehessen szűrni az egyes számonkérések esetében a csalásokat, másolásokat.

Informatikus hallgatók esetében beadandó programozási feladatok tekintetében elengedhetetlen a plágiumellenőrzés. Ennek hiányában a programok másolási aránya elképesztően magas mértéket öltene.

Ezen csalások kiszűrésére tervezték a plágium ellenőrző programokat, melyek nem csak egy-két, hanem több tucat vagy akár több száz program esetében használhatók. Segítségükkel optimális esetben kiderül, hogy mely programok hasonlítanak – másoltak. Így lehetőség nyílik a kérdéses munkák behatóbb vizsgálatára, és szükség szerint a munkák kizárására.

Viszont nem csak ilyen alkalmazási lehetősége van a forráskód hasonlóság vizsgáló programoknak, hanem üzleti viszonylatban is lehet szerepük. Például egy projekt tekintetében a fejlesztés során, mikor a különböző verziók között egyre kisebb az átláthatóság, felismerhetők a különböző, fejlesztés során kialakult programverziók.

Egy cégen belül is lehet jelentősége annak, hogy az egyes munkák mennyire hasonlók, tehát nem csak az oktatásban van jelentősége a plágiumellenőrzésnek. Olyan esetekben ahol például egy közös célkitűzést kell megvalósítani, szükségszerű az egyes munkák összehasonlítása.

Célom a szakdolgozatommal nem más, mint a különböző plágiumellenőrző algoritmusok bemutatása a teljesség igénye nélkül, illetve egy algoritmus, a SIM behatóbb ismertetése. Az egyes algoritmusoknál nem térek ki az alkalmazási lehetőségekre, leginkább csak az elméleti, működési alapokra.

A SIM algoritmus esetében szeretném bemutatni, hogy az algoritmus milyen hatékonysággal működik, leginkább olyan értelemben, hogy milyen szintű hasonlóságokat képes felfedezni az algoritmus. Tehát milyen átalakításokkal lehet egy programot átalakítani

úgy, hogy az ellenőrző programok ne fedezzék fel az átalakításokat. Az átalakítások révén megismerni azt, hogy milyen szintű erőforrás szükséges az ellenőrző programok kikerülése érdekében, beleértve az időt, mint meghatározó tényezőt, és az átalakításhoz szükséges tervezést – mentális tényezőt.

Célom így nagyobb rálátást kapni az egyes programok működésére, illetve a plágiumellenőrző algoritmusok működésére, hatékonyságára nagyobb rálátást adni.

II. Kezdet

Egy alapvető és nagyon praktikus kérdés vetődött fel az elmúlt 50 évben: adott a két sorozat, hogyan mérjük a hasonlóságot abban az értelemben, hogy a hasonlóság mértéke megfeleljen intuitív elképzeléseinknek.
Ezen kérdés alapján sok új kérdés merül fel a gyakorlatban. A genomikában, a két genom hasonlósága, az interneten, a két dokumentum hasonlósága, diákok között Java programozási feladatok megoldásainak hasonlóságának felderítése. Sok plágium ellenőrző rendszert, algoritmust dolgoztak ki. Az alapján mely jellemző tulajdonságok alapján hasonlítanak két programot, a rendszerek nagyjából két csoportba oszthatóak: attribútum-számláló rendszerek és struktúra metrikus rendszerek.

Az attribútum-számláló rendszerek csak a különböző operátorokat, különböző operandusokat, az összes operátorok számát és az összes operandusok számát számolják, majd felépítenek egy profilt ezen statisztikák alapján.
A struktúra-metrikus rendszerek megpróbálják figyelembe venni a programok struktúráját, ezért ezen rendszerek fejlettebb hasonlósági mértéket adnak és hatékonyabb technikát használnak a forráskód hasonlóság vizsgálatára.

Széles körben használt rendszerek a teljesség igénye nélkül: Plague, MOSS, JPlag, SIM és YAP struktúra-metrikus rendszer családok.

A rendszerek két működési fázisból állnak: az első fázis tartalmazza a tokenizálási folyamatot ami a forráskódot token sorozattá alakítja a lexikai elemző segítségével; a második fázis tartalmazza az elkészített token sorozatok összehasonlítását.
Meg kell jegyezni, hogy a legfontosabb probléma a struktúra-metrikus rendszerek második fázisában az, hogy hogyan kell mérni a tokensorozatok hasonlóságát. Nem megfelelő metrika alkalmazásával a hasonlóságok nem derülnek ki és a jól definiált de nem univerzális metrika mindig átverhető.

Három alapvető kritériumot kell teljesíteni a hasonlóság kereső programoknak:

(a) minden tokent meg kell számolni minden sztringben legfeljebb egyszer

(b) az áthelyezett kódszegmenseknek minimális hatást kell gyakorolnia a végleges hasonlósági számra

(c) a hasonlósági számnak csökkennie kell a tokenek véletlen beszúrásakor vagy törlésekor

Vitatható, hogy e három kritérium elegendő-e. Például sok más dolog is minimális hatást válthat ki, például: duplikált blokkok, közel duplikált blokkok, lényegtelen nagy blokkok beillesztése, stb…
Tény, hogy egyszerűen túl sok olyan eset létezik, hogy fel lehessen sorolni. (Whale cikkében bővebb elemzés is megtalálható erről a témáról:
http://www.docstoc.com/docs/36151305/Software-Metrics-and-Plagiarism-Detection)

Más megközelítést kell tenni. Visszalépünk, el a hasonlóság vizsgáló programoktól. Megvizsgáljuk az információ alapú metrikát, ami a két szekvencia közötti megosztott információt méri (bármely két szekvencia között) - DNS szekvenciák, dokumentumok, vagy programok között. A mérésünk a Kolmogorov komplexitáson alapul, ami univerzális. Az univerzalitás garantálja, hogy ha hasonlóság van két szekvencia között, a mérésünk megtalálja. Jóllehet hogy ez a mérték nem kiszámítható, bemutatom egy hatékony SID rendszeren, hogy hozzávetőlegesen hogy számoljuk ki a metrikus számot(ennél fogva az SID megosztott információ távolságát lehet értelmezni - Shared Information Distance). Később lesz látható a járulékos munka a különböző hasonlóság vizsgáló programok esetében, illetve az egyes programok teljesebb bemutatása.

III. Kapcsolódó munkák a hasonlóság ellenőrzésben

Bemutatásra kerül a teljesség igénye nélkül pár plágium ellenőrző rendszer. Mivel nagyon sok rendszer létezik, ezért csak 4 kerül elemzésre.

1. Attribútum számoló rendszerek

A kezdeti attribútum számoló metrikus rendszerek Maurice Halstead szoftvermetrikáit használták a hasonlósági mutató kiszámolására programpárok között, négy programbeli statisztika alapján:

- n_1 = a különböző operátorok száma
- n_2 = a különböző operandusok száma
- N_1 = az operátor előfordulások száma az összes különböző típust beleértve
- N_2 = az operandus előfordulások száma az összes különböző típust beleértve

A két metrikus érték a fenti értékek ismeretében következőképpen alakul:

- $V = (N_1 + N_2)\log_2(n_1+n_2)$
- $E = \frac{n_1 N_2 (N_1+N_2) log_2(n_1+n_2)}{2n_2}$

Ahol V-t a program információtartalmaként E-t pedig a megértéséhez, reprodukálásához, karbantartásához szükséges munka nagyságaként is szokás értelmezni. Bővebben lásd ebben a könyvben:
http://books.google.hu/books?id=3g8wtcpFHZcC&dq=halstead+metrics&source=gbs_navlinks_s.
Whale megmutatta hogy az attribútum számoló rendszerek nem képesek megfelelően észlelni a hasonló programokat.
Erről a témáról bővebben az alábbi publikációban: http://comjnl.oxfordjournals.org/content/33/2/140.full.pdf

2. Struktúrametrikus rendszerek

MOSS. A tokenizálási folyamatot gyors részsztring ellenőrzési folyamat követi. Az újonnan fejlesztett MOSS rendszer rostáló algoritmust használ a hasonlóságok megtalálására.

YAP. A hasonlósági számot használják a YAP rendszerekben, melynek értéke 0 – 100-ig terjed, százalékos egyezésnek nevezik, reprezentálva a távolságot a „nem-egyezéstől” a „teljes-egyezésig”. Az alábbi formula alapján kerül kiszámításra a hasonlóság:

$$\text{Egyezés} = \frac{(same - diff)}{minfile} - \frac{(maxfile - minfile)}{maxfile}$$

Százalékosegyezés = max(0, Egyezés) * 100

ahol a maxfile és a minfile a nagyobb és a kisebb fájl hossza, illetve, a same a megegyező tokenek száma a két fájlban, a diff pedig az egysoros különbségek száma az egyező tokenekből létrejövő blokkokon belül.
Célja, hogy megtalálja a maximális sort a közös összefüggő részsztringekben amíg csak lehet, amelyek egyike sem fedi át a már használt tokeneket a többi részsztringben.

SIM. A SIM plágiumellenőrző rendszer összehasonlítja a tokenszekvenciákat dinamikus sztring igazítási technológiát használva. Ez a technika először minden karakterpárhoz értéket – pontszámot – rendel, sorrendben. Például az egyezés 1 pont, eltérés -1, üres -2. A legmagasabb pontszámú blokk adja az irányítottsági pontot. A pont két szekvencia között definiálva van úgy, hogy a maximális pont minden igazítás között legyen, ami könnyedén kiszámítható dinamikus programozás segítségével. Ezzel a definícióval az egyezési mérték két szekvencia között a következőképp alakul:

$$s = \frac{2 * \text{pont}(s,t)}{(\text{score}(s,s) + \text{score}(t,t))}$$

Ez egy normalizált érték 0 és 1 között. Ez a metrika problémába ütközik a kódszegmensek áthelyezésekor a dinamikus programozási technika alkalmatlansága miatt.

SID

Definiálva van az információtávolság két szekvencia között, hogy minimális befektetéssel át lehessen alakítani egy szekvenciát egy másikká, és vissza is hasonlóképp. Míg az információtávolság több jó tulajdonsággal rendelkezik, problémás a hasonló szekvenciákra nézve. Például a genomikában. Az E. coli és a H. influenza testvér fajok. Habár nagy a genomikai hasonlóság, a két genom nagymértékben különbözik: az előbbi 5millió bázispárral rendelkezik, míg a másik 1.8 millióval.

Sokkal használhatóbb az információ alapú szekvencia távolság a hasonlóság mérésére szekvenciapárok között. Ennek a definíciója a Kolmogorov komplexitáson vagy algoritmus entrópián alapszik. Egy x sztring Kolmogorov komplexitása K(x), ami méri a szekvencia abszolút információ tartalmát, amit x tartalmaz. Ez a K(x) egy hossz, bitek számában kifejezve. Adott egy másik szekvencia, y, a feltételes Kolmogorov komplexitás: K(x|y), méri x információ mennyiségét y nélkül. A definíció alapján K(x|y) a legrövidebb program hossza az y bemeneten, ami x-et adja.

Az információ alapú szekvencia távolság a következőképpen alakul:

$$d(x,y)=1-\frac{(K(x)-K(x|y))}{K(xy)}$$

ahol a tört számlálója K(x)-K(x|y) y információ mennyisége x-el kapcsolatban. A Kolmogorov komplexitásban nagy mennyiségben igaz az hogy K(x) – K(x|y) ≈ K(y)- K(y|x), ezért d(x,y) szimmetrikus. Ezt az információt kölcsönös információnak is hívják x és y között. A nevező K(xy) az információ teljes mennyisége az összefűzött xy sztringben. Míg a közös információ nem metrikus – mivel nem elégíti ki a háromszög egyenlőtlenséget – , a d(x,y) metrikus. Kiderül, hogy a d(x,y)-t lehet tekinteni az információ távolság normalizált verziójának:

$$d(x,y) \approx \frac{(K(x|y)+K(y|x))}{K(xy)}$$

A d(x,y) metrikus. A megfelelő metrika garantálni tudja a mért távolság kisebb értékét, annál valószínűbben minél jobban hasonlít egymásra a vizsgált két program. Igazolva van az univerzális tulajdonság a távolsághoz, abban az értelemben, hogy minden távolsághoz

számítható távolság függvény (D) - kielégítve sok nagyon általános tulajdonságot - és a távolság normalizálható minden sztring esetében.

d(x,y)<= 2*D(x,y)+O(1)

A plágiumellenőrzést ezen kontextusban a következőképp lehet értelmezni: ha két program hasonlósága mérhető szintű, akkor a hasonlóságuk a d szám alatt van. Így legalábbis bizonyos értelemben, a szoftveres plágiumellenőrző rendszer olyan mértéken alapszik, ami nem kijátszható.

IV. SID

Az első formula bemutat egy ideális metrikát azáltal, hogy miként lehet mérni az információ mennyiséget melyen a két program osztozik. Szerencsétlenségre a jól ismert Kolmogorov komplexitás nem számítható. Az SID-t annak a reményében fejlesztették, hogy a d(x,y) durva közelítése még mindig használható eredményt hoz. Az SID-ben tömörítési algoritmushoz folyamodnak hogy heurisztikusan közelítsék a Kolmogorov komplexitást. A következőképp közelítjük a d(x,y)-t:

$$d(x,y) \approx 1 - \frac{(Comp(x) - Comp(x|y))}{Comp(xy)}$$

ahol a Comp(x) (Comp(x|y.)) reprezentálja a tömörítési algoritmus használatával keletkező tömörített sztring méretét. Az elmélet természetesen engedélyezi az SID-nek, hogy kezelje a feltételes Kolmogorov komplexitás alapján számított d(x,y|ProfessorCode) általános kódot. Ez a folyamat az SID-ben implementálva van.

1. Tokentömörítési algoritmus

SID a d(x;y) ésszerű közelítésén alapszik. Elvileg nem kiszámítható tömörítési algoritmus tudja közelíteni a d(x;y)-t. Gyakorlatban egyre jobban közelítjük d(x; y)-t. A tokentömörítési algoritmus az SID-hez lett tervezve, az alkalmazás egyedi szükségleteinek feltárására. A tokentömörítési algoritmus követelményei a tradicionális adattömörítéshez képest a következőképp alakulnak:

- A tradicionális tömörítési algoritmus gyakran igényel valós idejű(lineáris) futást. Az SID a lehető legjobb tömörítést igényli, míg az időkövetelménye sokkal lazább. A program szuperlineáris időben fut, de el kell érni a lehető legjobb tömörítési arányt a speciális típusú adatok miatt.
- Sajátos megfogalmazása d(x,y)-nak: megköveteli x feltételes tömörítésének számítását adott y, Comp(x|y) vagy Comp(xy) mellett. A plagizált programpár tipikusan hosszú és megközelítőleg duplikált blokkokat tartalmaz. Egy általános célú univerzális tömörítési algoritmus nem elegendő ahhoz, hogy kezelni tudja ezeket. Kezelni ezeket a pontos ismétlődéseket, Lempel-Ziv kutatásán alapuló algoritmusokkal, melyek határtalan bufferrel rendelkeznek - lehetséges. Az SID-nek veszteség nélküli tömörítési algoritmus kell, hogy hatékonyan kezelje a hozzávetőleges egyezéseket.

SID által használt elemző letöltésére a következő oldalon van lehetőség: http://www.cobase.cs.ucla.edu/pub/javacc. Minden token byte-onként van reprezentálva. A tokentömörítés követi Lempel-Ziv (LZ) adattömörítési sémát: először megkeresi a leghosszabb hozzátevőlegesen duplikált részsztringet, majd kódolja a pointer által – ami az előző előfordulásra mutat – plusz korrekciókkal. Az implementáció eltér a standard LZ tömörítési sémától azáltal hogy az alkalmazás egyedi jellemzőit és a felső követelményeket is figyelembe veszi. Először, a tokentömörítés figyelembe vesz minden előzőekben talált lehetséges részsztringet, hogy megkeresse az optimális egyezéshez való tömörítést. A tipikus LZ-type algoritmus végtelen buffer mérettel el tudja kerülni a hosszú ismételt sztringeket, mert csak egy része a kódolt stringeknek lenne hozzáadva a look-up szótárhoz ami a kódolási folyamat alatt épül fel. A kódolási idő csökkentését célozza meg memória hely csökkentéssel, annak árán hogy angol szövegeknél kicsit rosszabb a tömörítés. A token szekvenciák esetében hosszú és megközelítőleg azonos programrészek vizsgálatakor az ilyen algoritmusok

gyengébb tömörítést eredményeznek, pl: Compress, Gzip-9, Bzip-2, PPM-9. Meg kell jegyezni hogy a Lempel-Ziv típusú algoritmusokat végtelen bufferrel és pontos egyezésekkel már a múltban megcsinálták. A tipikus LZ algoritmusok nem kezelik a hozzátevőleges egyezéseket. A tömörítési program hozzátevőlegesen duplikált részsztringeket és kódolt eltéréseket keres. Tokentömörítés küszöb funkciót használ hogy meghatározza hogy melyik közelítő egyezés jobb a többi alternatívánál (például a kódolt részek külön-külön, vagy bizonyos részeit egyáltalán nem kódolva). Így több duplikált szakaszt lehet majd kombinálni hogy egy nagyobb szakaszt alkosson néhány eltéréssel.

A tokentömörítési algoritmus rövid ismertetése

A feltételes tömörítést használó algoritmusok hasonlóképp vannak implementálva. Hasonlóképp egyező kódszegmens-párok (azaz ismétlődési párok) egy fájlban találhatók és később lesznek bemutatva egy weblapon mely az eredményeket tartalmazza. Kísérletek azt mutatják, hogy a plagizáláskor a tokentömörítés jelentősen felülmúlja a széles körben használt tömörítő programok tömörítését, pl gzip-9, bzip-2, PPM-9, öt véletlenszerűen jelképes szekvencián.

Algorithm TokenCompress
Input: A token sequence s
Output: $Comp(s)$ and matched repeat pairs
$i = 0$;
An empty buffer B;
while $(i < |s|)$
  p = FindRepeatPair(i);
  if $(p.compressProfit > 0)$
    EncodeRepeatPair(p, $compFile$);
    $i = i + p.length$;
    OutputRepeatPair(p, $repFile$);
  else
    AppendCharToBuffer(s_i, B);
    $i++$;
EnodeLiteralZone(B, $compFile$);
return $Comp(s) = compFile.file\_size$;
  and repeat pairs stored in $repFile$.

2. Rendszer integráció

Az SID két fázisban működik. Az elsőben, a forrás programok elemzésre kerülnek, mégpedig úgy hogy tokeneket generálnak a lexikai fordító alapján.
A másodikban, a tokentömörítési algoritmust használják, heurisztikusan, a programpárok között az információs metrika d(x; y) meghatározásához.
Végül, minden programpár rangsorolásra kerül a hasonlósági távolság alapján. Az oktatók össze tudják hasonlítani a program kódokat egymás mellett az SID grafikus interfészt használva. Az SID blokk diagramja a következőképp néz ki:

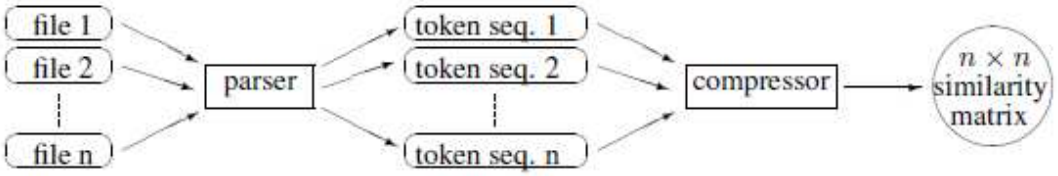


Figure 1: Block diagram of SID's design.

Az SID Java-ban van implementálva. Az oktatók beadhatják az összes programot amelyet össze kívánnak hasonlítani mégpedig egy zip fájl létrehozásával, amely tartalmazza az összes programot a saját könyvtárában, majd feltöltik egy SID szerverre egy böngészőn keresztül. Bár a jelenlegi változatában az SID csak a Java és C++ programokat fogad el bemenetként, össze tud hasonlítani programokat bármely más nyelven, ha az elemző azon adott nyelven be van építve az SID rendszerbe.

Implementálták az SID-t és a webes szolgáltatást, hogy online lehessen elvégezni plágiumellenőrzést. A rendszer alapja a távolság metrika, ami a normalizált mértékét adja meg a megosztott információnak a két program között. A mérték általános, így elméletileg nem átjátszható. A gyakorlatban, a mérőszám nem kiszámítható, az SID rendszer speciális tömörítő programot használ, hogy heurisztikusan közelítse a Kolmogorov komplexitást. Ahogy implementálva van, az SID nem csak a három szükséges tulajdonságot, hanem finomabb hasonlóságokat is észrevesz. Kis és nagy léptékű kísérletek azt mutatják, hogy a

SID nagyobb érzékenységgel és specifikussággal rendelkezik, mint a népszerű plágium észlelő rendszerek, köszönhetően a megfelelő, általános használt metrikának. Számos vizsgálat bizonyítja, hogy egy rossz metrika hamis pozitív, illetve téves negatív eredményeket tud szolgáltatni.

V. YAP

A YAP program kifejlesztését egy régebbi plágiumellenőrző szoftver, a Plague, ihlette, a minél nagyobb hatékonyságra törekvés volt. Először az ötletet adó Plague ismertetése következik
A Plague három fázisra osztható:

Az első szakaszban minden egyes fájlból tokensorozatokat állít elő, valamint egy listát szerkezet mutatókkal ellátva, átdolgozva, mint egy struktúrált profilt, amely összefoglalja a programban használt struktúrákat. A komponens szerkezeti metrika képviseli az iterációs és kiválasztási kimutatásokat és programrészeket.

A második szakaszban a struktúra profilok összehasonlításra kerülnek, és a legközelebbi szomszédpárok meghatározott kombinációját használja a nyelv specifikus távolságot mérő

funkcióihoz. Várható, hogy végül sok beadvány marad párosítatlanul és minden párosított beadvány előrelép a következő szakaszba.

Ebben a fázisban a token sorozatok összehasonlításra kerülnek, egy a leghosszabb közös részsorozat variánsát használó algoritmus alapján.

Míg a Plague valóban nagyon hatékony, mégis felvet néhány problémát a lehetséges felhasználók számára:

Plauge jelenleg alkalmazható Pascal, Prolog, Bourne Shell és a Llama-ban írt programokhoz (Pascal fordító generátor), minden egyes új nyelvhez való verzió megírása jelentős erőfeszítéseket igényel, kezdve egy elemző megszerkesztésével a célnyelvhez és a távolság mérőszámok használatra való kiválasztásával a második szakaszban ismertetettek alapján.

A használat során az eredmények két lista által, űrlap formájában érkezik vissza, indexek szerint rendezve, H és a HT, amelyeket értelmezni kell. A Plague kézikönyv útmutatást ad, az értelmezést illetően.

A Plague részei jelenleg Pascalban íródtak, és bár jó minőségű C-implementációk gyakoribbak, a Pascal implementációk ritkábbak és implementáció függő jellemzőkkel rendelkeznek. Plague is használja a GIVE rendszer segédprogramjait.

Míg a harmadik ezen problémák közül áthidalható némi erőfeszítéssel, és a második csak egy kis gyakorlatot igényel, az első probléma viszont szignifikáns. Ezt szem előtt tartva, a cél a jelenlegi projekt volt, hogy egy plágium ellenőrző rendszer jöjjön létre, amely egyszerű és hordozható, bár valószínűleg szükséges áldozni a Plague pontosságából és sebességéből. A rendszer neve YAP.1

2. A YAP komponensei

Plague három fázisa helyett a YAP kettőt tartalmaz: egy generáló szakaszt, amelyben egy token fájl jön létre minden egyes beadványhoz, azt kiértékeli, majd következik az

összehasonlítási szakasz, ahol minden token hasonlításra kerül. Token fájlok általában kisebb, 100-150 tokenből állnak egy kis feladathoz, és 400-700 token egy nagy feladathoz. A tokenek reprezentálják a "szignifikáns" elemeket a kiválasztott nyelven, általában vagy nyelvi struktúrákat vagy beépített funkciókat; azonosítókat (kivéve a funkciók vagy eljárások neve), és minden konstanst figyelmen kívül hagynak.

Az egyszerűség érdekében úgy döntöttek az elején, hogy a tokenizálási szakaszt valamennyi nyelven a YAP programok külön osztályába kell tenni. Azonban a tokenizálási osztály egy közös struktúrát igényel. Ez a következő

1.
Kapcsolódó munkálatok a forráskódokkal kapcsolatban:
Eltávolítja az észrevételeket és nyomtatási karakterláncokat kisbetűssé alakítja. Eltávolítja a nem érvényes azonosítókat a primitív tokenek listájából. Ezt használják a UNIX segédprogramok, *tr* és a *sed*. A C_YAP, 2 a C előfeldolgozó, cpp, is használja.

2.
Megváltoztatja a szinonimákat általános formára, például LISP_YAP-ban a *second* és *cadr* átalakításra kerül *car* és *cdr*-re, mialatt a C_YAP ban a *strncmp strcmp*-é alakul.

3.
Meghatározza a függvény/eljárás blokkokat. A LISP_YAP-ban ez triviális, és semmi különös tennivalóra nincs szükség; a C_YAP-ban a UNIX ctag-eket együtt használjuk awk-val.
4.
Miután a funkcióblokk azonosítva lettek, nyomtatásra kerülnek a funkció blokkok a hívási sorrendben. Egyfajta "makro-bővítés", kivéve, hogy egy blokk csak egyszer bővülhet(hogy megakadályozzák a tokenek számának megugrását). Egy alszekvencia meghívja a már kibővített funkciót, ami helyére egy számozott token kerül hogy reprezentálja a funkciót. Azonban, a programnak kicsinek(208 sor) kell lenni hogy könnyen testre szabható legyen a különböző célnyelvekhez.

5.

Felismeri és kinyomtatja a tokeneteket reprezentáló részeket a nyelvben a megadott szótárból.

A második összehasonlító fázis közös az összes implementációban. Ebben a UNIX *find* segédprogramot használják összegyűjteni egy listát a korábban elkészített token fájlokról. Figyelmen kívül hagyva ismétlődő vizsgálatokat, az egyes token fájlokat ezután hasonlítják az összes többihez az *sdiff* segítségével, amelynek kimenetét egy egyszerű *awk* szkript elemez az alábbiak szerint. Jelenleg, a részei ennek a második fázisban vannak összekötve egy másik shell scriptel. Ez meglehetősen lassú, a jövőben felülvizsgálatra kerül; újrakódolása a nemrég megjelent Perl nyelvben.

3. Eredmény

Az eredménye minden elemzésnek egy érték 0-tól 100-ig terjedő tartományból "nincs egyezés"-től a "teljes egyezés"-ig. A használt képlet:
Match = (azonos diff) / minfile - (maxfile-minfile) / maxfile;
PercentMatch = max (0, Match) × 100;

ahol a maxfile és a minfile a nagyobb és a kisebb fájlok hossza, illetve, a same a megegyező tokenek száma a két fájlban, a diff pedig az egysoros különbségek száma az egyező tokenekből létrejövő blokkokon belül.
A PercentMatch érték: ha egy adott hasonlóság meghaladja a felhasználó által parancssorban megadott küszöbértéket, akkor az összehasonlítás részei, közösen a PercentMatch-el tárolásra kerülnek a második fájlban, yap.summary. Végül, amikor az összes hasonlítás elkészítette a yap.summary-t, rendezve lesz csökkenő sorrendben a PercentMatch tekintetében, majd a középérték és átlagos szórásérték kiszámításra kerül, majd hozzá lesznek fűzve a fájlhoz.

Kiszámítása különösen az átlagos szórásnak nem igazán helytálló, mivel az összehasonlítás viszonylagos. Először is, az összes beadvány ugyanazt a problémát célozza meg, ugyanazon a nyelven. Továbbá lépéseket tesznek annak érdekében, hogy az egyes nyelvek használatát általános formára hozzák.

Ez kiderül, hogy nem lehet probléma, mivel alkalmazható valamennyi az összehasonlításra, ez csak azt jelenti, hogy nem nulla feltétel elkerülhetetlen. Nagyobb probléma Whale "redundáns" illeszkedése. Azaz, ha A beadvány hasonlít B beadványra és B hasonlít C-re, egy hamis találat lesz A és C között(Ugyanez nem mondható el egyértelműen a nem-egyezésekről.)

Amint javasolt, a helyes megoldás egy olyan fa építése lenne, amely kizárólag az "alapvető" egyezéseket tartalmazza. A fának tartalmaznia kellene minden egyezést, és így a fa építése során minden összehasonlítás bekerül. Ez szükségtelen, ha valaki hajlandó elfogadni, hogy az átlagot és a szórást is valószínűleg túlbecsüli a valós átlag és a szórás.

Mivel a kikötés szükséges, hogy bármilyen megjelenő 3 vagy több eltérést a standard átlagtól érdemes közelebbről megvizsgálni.

VI. SIM

Bemutatom kialakításának célját, tervezését és végrehajtását a SIM nevű programnak, ami két C program közötti hasonlóságot mér (eredetileg C programok elemzéséhez tervezték, azonban mára a főbb nyelvek nagy részére kiterjeszthető). Használható sok program plagizálásra is. Ez a szoftver része egy projektnek, ami az informatika tanítását segíti.

1. Bevezetés

Számítógépes ismeretek oktatása azzal kezdődött, hogy létrehozták az első számítógép tudományi részleget több mint harminc évvel ezelőtt, és még a szakemberek sem tudtak lépést tartani a kollégák más számítógép tudományi alrészek fejlődésével. Ma már van egy sor lenyűgöző szoftverrendszer, ami rendelkezésre áll szoftvertervezőknek, áramkör tervezőknek, a hálózati rendszergazdáknak, numerikus elemzőknek, és a digitális művészeknek. Ezzel szemben a számítógép-tudomány oktatói még mindig a hagyományos eszközökre és technikákra támaszkodnak: legfőbb eszköz az értékelés során még mindig egy emberi mérce. Valószínűsíthető, hogy ez a szűkössége a szoftvereszközöknek nem a pedagógusok szorgalmának hiánya miatt van, hanem amiatt hogy nincs kiforrott elmélet ezen a területen. Kutatás a számítógépes tanítás elméletében, ami kivizsgálja a korlátait és költségeit különböző stratégiáknak a tanulás terén, csak a 80as évek közepén kezdődött. Hasonlóképpen, a kutatás a program ellenőrzésben - ami a programok korrektségét méri - kezdetét vette a 1980-as évek végén. Mindkét fejlődő terület generál mély és meglepő eredményeket, amelyek minden bizonnyal javulást eredményeznek a gyakorlatban a számítógépes ismeretek oktatásával kapcsolatban a közeljövőben.

De ezen a ponton nem minden szoftver a számítógép tudomány oktatásában igényel további megalapozó munkát. Például, értékelési feladatok alacsonyabb szintű programozási tanfolyamok vizsgáin. Ezek a tanfolyamok általában számos programozási feladatot tartalmaznak egyszerű megoldásokkal, amelyeknek értékelni kell nem csak a helyességét, hanem a stílusát és az egyediségét.

Erre a feladatra kidolgozott szoftvernek kellene vizsgálni a programok felépítését, és hasznos lenne a már jól kidolgozott sztring algoritmusok elméletének előnyeit használni. Bemutatom a SIM tervezését és működését, ami a strukturális hasonlóságot mér két számítógépes program között. Ez a program közvetlen alkalmazása azoknak a sztring összehangolás technikáknak, amelyeket nemrég fejlesztettek ki a DNS sztringek közötti hasonlóságok kimutatására. Adott két program, a SIM először csökkenti őket fává, egy standard lexikális elemzőt használva. Megtekinti az elemző fákat sztringként, majd a SIM igazítja őket üres részek beszúrásával, hogy megtalálja a tokenek maximális közös részsorozatát.

A pontszám 0.0 és 1.0 között lesz, a két program hasonlósága alapján.

A jelenlegi formájában, a SIM futási ideje 0 (s^2), ahol s a legnagyobb méret a parse fáknak. A nagy része a SIM algoritmusnak C++ ban van implementálva, a grafikus felhasználói felület pedig Tcl/Tk-ben.

Kísérletek a valós adatokon azt mutatták, hogy a SIM ellenáll a széles körű névváltoztatásnak, programrészek és függvények áthelyezésének, kommentek és üres helyek hozzáadásának és eltávolításának. A SIM meglehetősen gyors a kisméretű programok esetében: összehasonlítása minden 56 programpárnak, amelynek átlagos hossza 3415 bájt mintegy három és fél perc.

A Unix *diff* parancs is használható a két program közötti hasonlóság mérésére, de az alap szöveges egység egy sor helyett a parse fa token, mint a SIMben, de a diff nem ismeri fel a névváltoztatásokat vagy szöveges modulok átrendezéseit.

Baker Dump programja jelez minden maximális egyezési küszöböt túllépő egyezést két program között. Képes észlelni szisztematikusan (paraméterezhető) a nagy modulok névváltoztatását és áthelyezését, de ez hatástalan, ha hamis kódrészletek kerülnek be, vagy ha kis blokk átrendezése történt.

Aiken MOSS programja UC Berkeley plágiumainak felderítésére került kifejlesztésre, továbbá ez az algoritmus már vizsgálja a program szerkezetét, de a mögöttes algoritmust nem hozták nyilvánosságra.

Egyéb eszközök kifejezetten a plágium kimutatására, mint a [I/O, 11 J heurisztikán alapszik, ami becsli a szavak és a szimbólumok frekvenciájának statisztikáját, és így lehet, hogy magas a hamis pozitív eredmények száma.

2. String Igazítás

Ez a rész demonstrálja, hogy a sztring igazítási algoritmus - amit a SIM használ - hogy hasonlít össze két programot. A két igazítandó karakterláncot - s és t (az esetlegesen eltérő hosszúságú) - úgy kapjuk meg, hogy üres tereket beillesztünk a karakterláncokba, hogy azok hossza azonos legyen. Ne feledjük, hogy sok lehetséges igazítás létezik. Például két igazítása az alábbi sztringeknek, a "masters" és a "stars"-nak: masters masters sta rs stars.

A karakterpárok a következőképpen adottak az igazítás szerint: az egyezés m, az eltérés d, és a gap büntetés g, ahol m, d és g választott értékek. A blokkpárok pontszámai az összegzése az egyedi pontszámoknak, és a legtöbb pontszámot kapott blokkpárok adják az igazítás pontszámát. A fenti példára, ha m=1, d=-1 és g= -2 és „r&s” és „sters/stars” a legnagyobb pontszámú blokkok az első és második igazításban.

Az igazítási pontszámok 2 és 3. A sztring igazítás kiválóan alkalmas pontatlan egyezések kiszűréséhez, széles körben használják a számítógépes biológiában DNS-szálak közötti kapcsolatok kimutatására. Az optimális igazítási pontszám két karakterlánc között az összes igazítás maximális pontszáma. Ezt az értéket ki lehet számítani a dinamikus programozás segítségével.

Formálisan, adott két karakterlánc, s és t, definiáljuk a D(i,j)-t hogy a legoptimálisabb igazítási pontszámként szerepeljen a két részsztring, s[l...i] és t[l..j] között.

$\max_{1<=i<=|s|,1<=j<=|t|} D(i,j)$ az érték amit keresünk

Definiáljuk a pontszámot:

$$\text{score}(s[i], t[j]) = \begin{cases} m, \text{ha } s[i] = t[j] \\ d, \text{egyébként} \end{cases}$$

A következő rekurzióval számolható a megoldás:

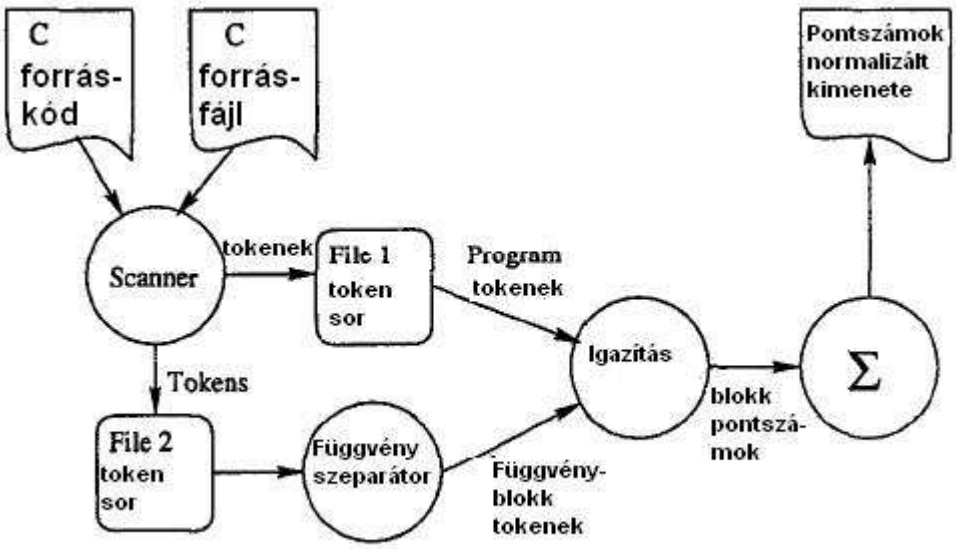
$$D(i,j) = \max \begin{cases} D(i-1, j-1) + \text{score}(s[i], t[j]) \\ D(i-1, j) + g \\ D(i, j-1) + g \\ 0 \end{cases}$$

A peremfeltételek adottak D(l, i) = i*g és D(j, 1) = j*g által. A D mátrix elemei számíthatók az első sor és oszlop a peremfeltételekkel való inicializálásával és az elemek balról jobbra és fentről lefelé való kiértékelésével.

D(i,j) lehetséges értéke csak a D(i – 1, j-1), D(i - 1,j), és D(i, j-1) értékektől függ.

3. Dizájn és Implementáció

A SIM implementálása 1980 programsorban történt C++ban és 393 programsorban Tcl/Tk-ban. A SIM blokk diagramja a következőképp néz ki:



A lexikális elemző automatikusan generálja a Unix *flex* parancs által adott megfelelő részhalmazát a C nyelvtannak, így a SIM egyszerűen módosítható hogy más nyelvekben írt programokkal dolgozzon. Minden token képvisel vagy egy aritmetikai, vagy egy logikai műveletet, egy írásjelet, egy C makrót, egy kulcsszót, egy numerikus vagy sztring konstanst, egy megjegyzést vagy egy azonosítót.

Például:
for (i = 0; i < max; i++)
helyettesíthető a következő token sorozattal:

TKN-FOR TKN-LPAREN TKN-ID-I TKN-EQUALS TKN-ZERO . . .

A token számok kulcsszavakhoz és speciális szimbólumokhoz előre definiáltak. Minden megjegyzés - nem számít milyen hosszú - helyébe fix token kerül és az üres helyek ignorálva lesznek. E tokenizáló folyamat csökkenti a program a parse fáját, ami általában sokkal rövidebb, és eltörli a lényegtelen információkat, mint például az üres helyeket és hozzászólásokat, mielőtt az összehasonlítás végrehajtódik. Miután a két forrás program tokenizálása megtörtént, a második program token sorozata részekre oszlik, mindegyik képviseli egy modulját az eredeti programnak. Minden ilyen modul így igazodik az első program token sorozatához. Ez a technika lehetővé teszi a SIM hasonlóság felderítését még abban az esetben is, amikor modulok pozíciói a programban permutáltak.

Az aktuális igazítás a következőképp van pontozva:

- az egyezés magában foglalva a két azonosító tokent, 2 pont, más egyezések 1
- a gap -2
- az eltérés magában foglalva a két azonosító tokent 0; más eltérés -2.

Ennek magyarázata világos: azonosítók egy egyezése tekinthető rendkívülinek és így a pontszám legmagasabb, míg az alaki eltérés a két azonosítónál tekinthető kevésbé jelentősnek, mint a strukturális eltérés egy azonosítónál és operátornál.

A teljes igazítási pontszám az egyes blokkok egyéni pontszámából kerül kiszámításra, majd 0.0 és 1.0 közötti normalizálás után osztjuk az első és a második program igazítási pontjainak az összegével.

$$s=\frac{2*score(p1,p2)}{(score(p1,p1)+score(p2,p2))}$$

A Tk/Tcl esetében grafikus felhasználói felület áll rendelkezésre, hogy lehetővé tegye az összehasonlítást a referencia fájl és fájlkollekció között, és megjelenítse és nyomtassa az eredményeket oszlopdiagram formájában. Különböző színekkel vannak jelölve a

hasonlóságok (piros, sárga és zöld) a hasonlóság mértékétől és a referencia program alapján meghatározott értékhatároktól függően. A különálló felület megjelenítéséhez az eredmények felhasználásához *gnuplot* is rendelkezésre áll.

4. Üzembe helyezés és eredmény

Egyszerű kísérletként egy kis C program esetében és megváltoztatásra került minden változó és függvény neve, kommentek törlésre kerültek, fordított sorrendbe kerültek a szomszédos programrészek - amennyiben lehetséges – és a funkciók sorrendje permutált. A megváltoztatott program és a referenciaprogram közötti hasonlóság 0.4 lett, ami már vizsgálatot indokol.
Értelmezni a pontértéket, sokkal megbízhatóbb, mint a grafikonok vizsgálata az összes SIM pontszámnál sok program esetében, illetve hogy egy sor rögzített küszöbértéket használnánk.
Ennek illusztrálására a SIM egy csoport, 36 ténylegesen beadott házi feladat programon került tesztelésre. Minden programot a referenciaként használt program összehasonlítja a többivel, és a pontszámok táblázatos formában kerültek be egy oszlopdiagramba. Az eredmény egy sor összehasonlítás volt.
A grafikonban, a bal szélső sáv jelöli a referencia program pontszámát saját maga ellen, ami 1,0.

A statisztika az egész csoportra vonatkozóan, mint fájlméretek, token tömb mérete, és futási ideje a 630 összehasonlításnak megjelenítésre kerültek. Programméretek bájtban, az idő másodpercben.

5. Következtetések

Arra tervezték és implementálták a szoftvert, hogy mérje a hasonlóságot két C program között, amelyeket fel lehet használni plágium kimutatására programozási házi feladatoknál az alacsonyabb szintű számítástechnikai képzéseknél. Megállapítható, hogy a program hatékony az általános módosítások ellen, mint például a név változtatás, programrészek és funkciók áthelyezésénél, valamint a hozzáadása vagy eltávolítása megjegyzéseknek és üres helyeknek.

Mint már említetésre került, a SIMet ki lehet terjeszteni, hogy értékelje nem csak az egyediséget, hanem a stílust és helyességet számítógépes programoknál.
Ezt az eszközt elképzelhető hogy használni lehetne az interaktív vizsgáztatásra az első szintű programozási tanfolyamokon.
Vannak ismert heurisztikák, hogy csökkentsék a négyzetes futási idejét az alapul szolgáló karaktersorozat összehangoló algoritmusnak, amely tervek szerint a következő SIM változat tartalmazni fog. Ráadásul nem kell futtatni a SIMet minden lehetséges programpáron a plágium felderítése érdekében.
Mivel a csalások száma általában kicsi, ez gyakran elég hogy megtalálja a legmagasabb pontszámú párt egy programcsoportban. Ha nincs csalási bizonyíték ezekből a párokból, az oktatónak nem szükséges vizsgálnia a többit. Jelenleg dolgoznak az al-négyzetes algoritmusokon a legmagasabb pontszámú pár megkeresésének érdekében.

A SIM-et kifejlesztő egyetemi honlapról szabadon letölthető a program. Több platformra fejlesztették a programot, tehát elérhető egyaránt Windows-ra és Linux-ra. A Windows-on futtatható verzió csak egy darab futtatható fájl, a Linux-os pedig egy tömörített fájl, mely további 73 fájlt tartalmaz. Ezen fájlok között szerepelnek a SIM leírásai, használati utasításai, illetve maga a SIM magja, maga az algoritmust tartalmazó .c kiterjesztésű állományok. Az algoritmus összetettsége miatt, illetve a hely szűke miatt csak egy kódrészlet következik:

```
#include <stdio.h>
#include <stdlib.h>

#include "settings.par"
#include "sim.h"
#include "options.h"
#include "newargs.h"
#include "language.h"
#include "error.h"
#include "hash.h"
#include "compare.h"
#include "pass1.h"
#include "pass2.h"
#include "pass3.h"
#include "stream.h"
#include "lex.h"

unsigned int MinRunSize = DFLT_MIN_RUN_SIZE;
int PageWidth = DFLT_PAGE_WIDTH;
FILE *OutputFile;
```

```
FILE *DebugFile;

struct text *Text;                      /* to be filled in by Malloc() */
int NumberOfTexts;                      /* number of text records */
int NumberOfNewTexts;                   /* number of new text records */
int ThresholdPerc = 0;                  /* threshold percentage to show */

const char *progname;                   /* for error reporting */

static const char *output_name;         /* for reporting */
static const char *min_run_string;
static const char *page_width_string;
static const char *threshold_string;

static const struct option optlist[] = {
        {'r', "minimum run size", 'N', &min_run_string},
        {'w', "page width", 'N', &page_width_string},
        {'f', "function-like forms only", ' ', 0},
        {'d', "use diff format for output", ' ', 0},
        {'T', "terse output", ' ', 0},
        {'n', "display headings only", ' ', 0},
        {'p', "use percentage format for output", ' ', 0},
        {'t', "threshold level of percentage to show", 'N', &threshold_string},
        {'e', "compare each file to each file separately", ' ', 0},
        {'s', "do not compare a file to itself", ' ', 0},
        {'S', "compare new files to old files only", ' ', 0},
        {'F', "keep function identifiers in tact", ' ', 0},
        {'i', "read arguments (file names) from standard input", ' ', 0},
        {'o', "write output to file F", 'F', &output_name},
        {'-', "lexical scan output only", ' ', 0},
        {0, 0, 0, 0}
};

static void print_stream(const char *fname);

int
main(int argc, const char *argv[]) {
        progname = argv[0];             /* save program name */
        argv++, argc--;                 /* and skip it */

        /* Set the default output and debug streams */
        OutputFile = stdout;
        DebugFile = stdout;

        /* Get command line options */
        {       int nop = do_options(progname, optlist, argc, argv);
                argc -= nop, argv += nop; /* skip them */
        }

        /* Treat the value options */
        if (min_run_string) {
                MinRunSize = strtoul(min_run_string, NULL, 10);
                if (MinRunSize == 0)
                        fatal("bad or zero run size; form is: -r N");
        }
        if (page_width_string) {
                PageWidth = atoi(page_width_string);
                if (PageWidth == 0)
```

```
                        fatal("bad or zero page width; form is: -w N");
        }
        if (threshold_string) {
                ThresholdPerc = atoi(threshold_string);
                if ((ThresholdPerc > 100) || (ThresholdPerc < 0))
                        fatal("threshold must be between 0 and 100");
        }
        if (output_name) {
                OutputFile = fopen(output_name, "w");
                if (OutputFile == 0) {
                        char msg[500];

                        sprintf(msg, "cannot open output file %s",
                                output_name);
                        fatal(msg);
                        /*NOTREACHED*/
                }
        }

        if (option_set('i')) {
                if (argc != 0)
                        fatal("-i option conflicts with file arguments");
                get_new_args(&argc, &argv);
        }

        if (option_set('-')) {
                /* it is the lexical scan only */
                while (argv[0]) {
                        print_stream(argv[0]);
                        argv++;
                }
                return 0;
        }

        /* Start processing */
        InitLanguage();

        /* Read the input files */
        Pass1(argc, argv);

        /* Set up the forward reference table */
        MakeForwardReferences();

        /* Compare the input files to find runs */
        Compare();

        /* Delete forward reference table */
        FreeForwardReferences();

        /* Find positions of the runs found */
        Pass2();

        /* Print the similarities */
        Pass3();

        return 0;
}
```

```
static void
print_stream(const char *fname) {
	fprintf(OutputFile, "File %s:", fname);
	if (!OpenStream(fname)) {
		fprintf(OutputFile, " cannot open\n");
		return;
	}

	if (!option_set('T')) {
		fprintf(OutputFile,
			" showing token stream:\nnl_cnt, tk_cnt: tokens");

		lex_token = EOL;
		do {
			if (TOKEN_EQ(lex_token, EOL)) {
				fprintf(OutputFile, "\n%u,%u:",
					lex_nl_cnt, lex_tk_cnt
				);
			}
			else {
				print_token (OutputFile, lex_token);
			}
		} while (NextStreamTokenObtained());

		fprintf(OutputFile, "\n");
	}

	CloseStream();
}
```

6. A SIM használata

A linuxos verziótól valamelyest egyszerűbb Windowson futtatható SIM használata, hiszen csak egy futtatható fájlt kell letölteni, és már használható is a program.

A program futásához szükséges paraméterként megadni azon programok elérhetőségét, illetve neveit, melyeket szeretnénk összehasonlítani. A megadott programok ezután összehasonlításra kerülnek, és az eredmény alapértelmezésként a képernyőre kerül kiíratásra. Célszerű ezért a kimenetet átirányítani - például egy txt fájlba - a későbbi elemzések megkönnyítésére.

A összehasonlítandó programok elérhetőségén kívül meg lehet adni opcionálisan paramétereket:

- r : minimális futási méret
- d : más formátumú kimenet

- p : százalékos egyezési szám adása
- o : kimenet megadott fájlba írása
- - : kizárólag lexikális elemzés adása
- s : a fájlt önmagával való összehasonlításának elkerülése
- i : paraméterek beolvasása az alapértelmezett bemenetről
- e: minden fájl minden fájllal való összehasonlítása

Az összehasonlítás alapjául szolgáló két egyszerű program forráskódja a következőképp alakul:

```
/*Original copy*/

#include <stdio.h>
#include <time.h>

#define SIZE 10

/*select a 'sample_size' random sample form 'size' elements*/

void sample(int a[], int size, int sample_size)
{
	int i, j;

	if(sample_size> size || sample_size<0)
		return;

	for(i = 0 ; i< size; ++i)
	{
	j=random()% (size – i );
	if( j< sample_size)
	{
	a[i]=1;
	--sample_size;
}
else
	a[i]=0;
	}
}

/*  return a random permutation of [1..size] */

void shuffle( int a[], int size)
{
	int i, j;
	int temp;

	for( i = 0; i< size; ++i)
		a[i] = i+1;
```

```
        for( i = 0; i< size - 1 ; ++i)
        {
        j= ( random() % (size – i)) + i;
                temp= a[i];
                a[i]=a[j];
                a[j]=temp;
}
}
/*print array* /

void print_array(int a[], int size)
{
        int i;
        for( i = 0; i< size; ++i)
                printf( ''%d '', a[i]);
        printf(''\n'');
}

void main(void)
{
        int a[SIZE];
        int sample_size;

        arandom(time(NULL));

        printf('' A random shuffle of %d elements: \n", SIZE);
shuffle(a,SIZE);
print_array(a,SIZE);

        sample_size= random() % SIZE + 1;
        printf('' A random sample of %d out of %d elements: \n'', sample_size, SIZE);
        sample(a, SIZE, sample_size);
        print_array(a, SIZE);
}
```

```
/*Modified copy*/

#define N 10
#include <time.h>
#include <stdio.h>

void main(void)
{
        int f;
int a[N];

        arandom(time(NULL));

shuff(array,N);
        printf('' A random shuffling: \n");
print(array,N);
```

```
	f= random() % N + 1;
	samp(array, N, f);
	printf(’’ A random sampling:\n’’);
	print(array, N);
}


void samp(int array[], int n, int m)
{
	int i, j;

	if(m> n || m<0)
		return;

	for(i = 0 ; i< n; ++i)
	{
	j=random()% (n – i );
	if( j< m)
	{
	array[i]=1;
	--m;
}
else
	array[i]=0;
	}
}




void print (int array[], int t)
{
	int i;
	for( i = 0; i< t; ++i)
		printf( ’’%d ’’, array[i]);
	printf(’’\n’’);
}


void shuff( int a[], int s)
{

int p;
int n, m;

	for(mi = 0; m< s; ++m)
		array[m] =m+1;


	for( m = 0; m< s - 1 ; ++m)
	{
	n= ( random() % (s – m)) + m;
		p= array[m];
		array[m]=array[n];
		array[n]=p;
}
}
```

VII. A különböző plágiumellenőrző programok tesztelése.

1. Esettanulmányok – elkapni és nem lebukni

Elkerülni a lebukást a másolók célja. Ebben a részben, utat keresek a plágium ellenőrző becsapására és teszteltem melyik rendszer a legellenállóbb a támadások ellen.

Egyszerű támadások. a jól ismert trükkök, szisztematikusan végrehajtott támadások a JPlag, MOSS, és SID ellen. Minden tesztben a JPlag, MOSS, és SID mindegyik ellenáll a változó és függvény átnevezésekkel, változók átnevezésével és cserélésével, irreleváns blokkok beszúrásokkal szemben. Ezek a tesztek megtalálhatók a http://genome.math.uwaterloo.ca/SID/tests.html oldalon. *A standard teszt eset.* Gitchell és Tran egy egyszerű és szemléletes tesztet mutatott be ami megfelelő hasonlósági metrikák megbízhatósági tesztelésére. Ebben a tesztesetben a másoló kicserélt minden változó és függvény nevet, kitörölte a kommenteket, megcserélte szomszédos programrészeket ahol lehetséges és permutálta a függvények sorrendjét az egyik programból a másikba. Alávetettük ezt a programpárt az SID-nek. Az eredmény a táblázatban található, ahol R(x; y) = 1 - d(x; y) = $\frac{(K(x)-K(x|y))}{K(xy)}$ A várt eredmény, R(x; y) hasonlóan szimmetrikus, R(x; y)≈R(y; x)≈0:4.

Annak érdekében, hogy további értelmezésben ilyen hasonlósági pontszámot intuitív módon meg lehessen kapni, úgy a program pár x és y azonos méretűnek kell lennie. Tegyük fel hogy p százaléka a program kódnak pontosan másolt x és y között és egyéb részek nem tömöríthetők.

Majd, R(x; y) = p/(2 - p):

Így, az érték R(x; y) = 0:4 A fenti példa azt mutatja hogy p = 57:14% ban a programkód azonos, ha x és y hasonló méretűek.

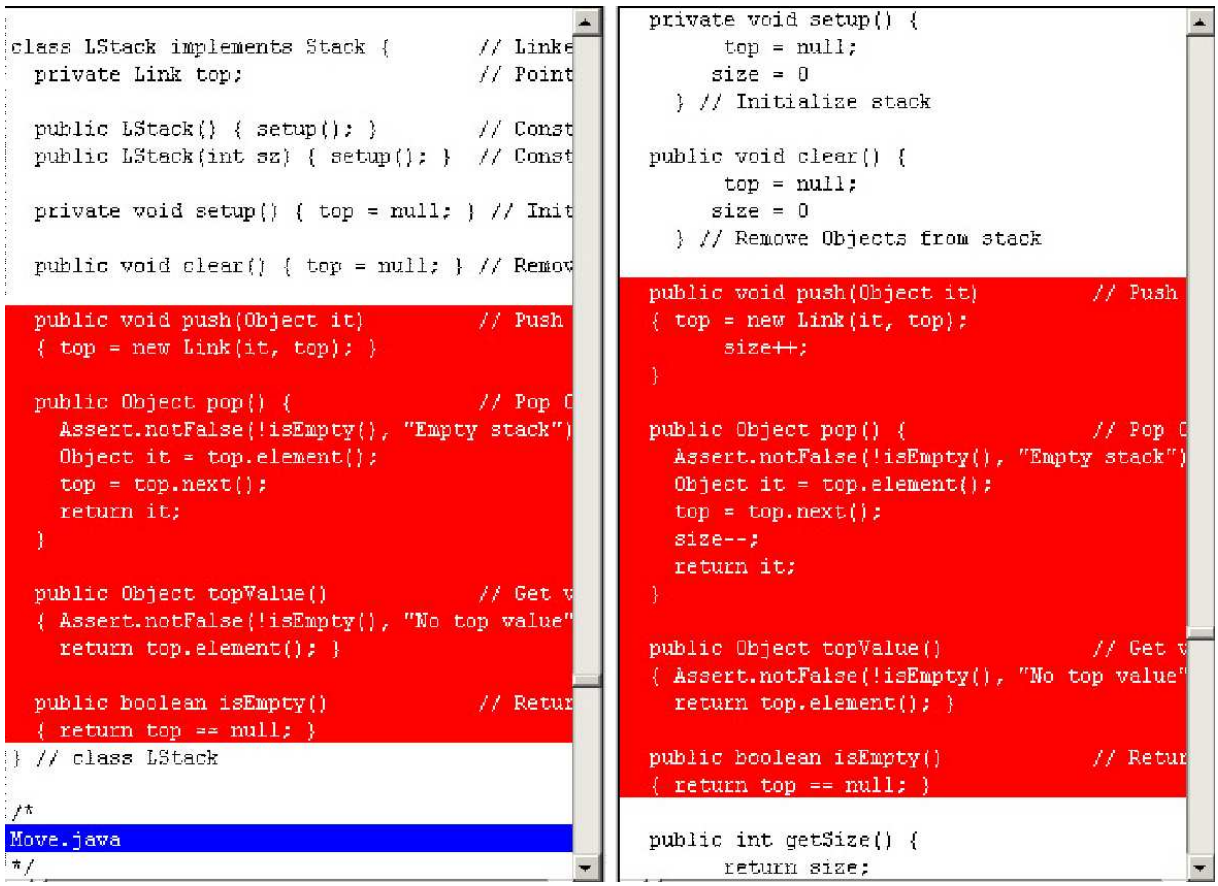
Ez garantálja, hogy a programpáron további vizsgálatokat kell végeznie az oktatónak.

Megfigyelhető, hogy a tömörítés miatt általánosan, az R(x; y) értéke, mint a 0,4 meglehetősen magas. Ha az ilyen pontozási módszert alkalmazták a biológiai szekvenciákon, genomokon testvér fajok esetében, gyakran 0,4 körüli érték jött ki.

Véletlen beszúrások . Hogy verjük át a JPlag-t és MOSS-t.

A másolók nem értik az eredeti programot (különben oktatók már elérték volna az oktatási céljaikat), így nem tudnak törölni dolgokat belőle. Így egy általános trükk az, hogy irreleváns kódokat helyeznek el, mint például „int x = 0”; remélve, hogy az ilyen beszúrások összezavarják a kereső mechanizmust. Az ilyen véletlen beszúrások rejtve tudnak maradni, például a hibakereső részekben, mint például System.out.println(.debug.). A metrikánk és annak tokentömörítési algoritmussal való közelítése természetesen lehetővé teszi a védelmet az SID elleni egyszerű szerkesztési módosításokkal szemben, az ilyen irreleváns részek beszúrásával szemben.

Egy gyakorlati példa az ábrán mutatja ahol a kiemelt hasonló kódszegmensek felfedésre kerültek az SID által, de egyikre sem derült fény sem a JPlag sem a MOSS által (2002-ben)



2003 óta a JPlag-t és a MOSS-t fejlesztették. Azonban az egyszerű beillesztés mindig legyőzi őket. Vizsgálatot hajtottak végre két kis programon (A) mintegy 100 sornyi kóddal, és egy

nagy programon (B), mintegy 1100 sornyi kóddal. Minden programba be lett szúrva egy bizonyos számú System.out.println (debug), rész, melyek gyakran előfordulnak hibakeresés során a gyakorlatban.

Az irreleváns kódok száma növekszik JPlag és MOSS esetében arra a pontra, ahol szerintük a programok hasonlóak, míg az SID másrészt bemutatja a rugalmasságot az ilyen csaló kísérleteken, mert soha nem megy az alábbiakban 50%os hasonlóság fölé.

Hamis pozitív eredmények nem megfelelő metrikákhoz

Egy kétségkívül mesterséges teszt. A programok az alábbi alakban vannak: *P, Q,* Q1, . . . , Qk, ahol P kicsit nagyobb méretű, mint a Q, Q egy triviális program egy tankönyvből (alap rendező program) hogy mindenki megértse és le tudja implementálni, Q_i Q Q variációi, mondjuk gyors, buborék rendezés. A és A' másolta P-t, de másképp írták meg Q-t. B és B' diákok P egymástól függetlenül impelementálták, de használták az alap tankönyvet. JPlag és MOSS nagyon magas értékeket adott a hamis pozitív B és B' párok között, mialatt az eredi bűnös hiányzott A és A' közül amikor sok B pár van jelen. Így SID jelenti A és A_0-on a másolást mivel tömöríti Q;Q_1; : : : ;Q_k-kat |Q| méretre ezáltal felfedezi a nem triviális másolt P részt.

2. Programkódok átalakítása

A következőkben kerül bemutatásra, hogy az adott programkód milyen szintű változtatása milyen mértékű egyezési értéket von maga után.

Először lássuk a Jplag eredményét a már látott (módosítatlan) példákon.

JPlag Search Results

Title:	Untitled2
Directory:	D:\szakdoga\vizsgalni
Programs:	copy.c - original.c
Language:	C/C++ Scanner [basic markup]
Submissions:	2
Matches displayed:	1 (Treshold: 67.1%) (average similarity) 1 (Treshold: 72.0%) (maximum similarity)
Date:	2011-03-05
Minimum Match Length (sensitivity):	12
Suffixes:	.cpp, .CPP, .c++, .C++, .c, .C, .h, .H, .hpp, .HPP

Distribution:

90% - 100%	0	
80% - 90%	0	
70% - 80%	0	
60% - 70%	1	##########
50% - 60%	0	
40% - 50%	0	
30% - 40%	0	
20% - 30%	0	
10% - 20%	0	
0% - 10%	0	

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

original.c -> copy.c (67.1%)

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

original.c -> copy.c (72.0%)

JPlag2.11.6

A kapott egyezési érték átlagos hasonlóság esetében 67.1%, míg a maximális hasonlóság esetében 72%.

Mint tudjuk, a két program teljes egészében megegyezik, csak a változónevek, függvénynevek vannak kicserélve, illetve a függvények sorrendje.

A SIM algoritmus által kapott a következőképp alakul:

```
File copy.c: 315 tokens
File original.c: 285 tokens
Total: 600 tokens

copy.c: line 35-52                    |original.c: line 17-32            [50]
 {                                    | {
 j = random();                        |  j = random() ;
 j%=(n-i);                            | j%=(size-i);
 if( j< m)                            |  if( j < sample_size )
  {                                   |   {
  array[i]=1;                         |   a[i]=1;
  --m;                                |   --sample_size;
  }                                   |   }
 else                                 |  else
  array[i]=0;                         |   a[i]=0;
 }                                    | }
}                                     |}
                                      |
                                      |/*  return a random permutation of [1.
                                      |
                                      |void shuffle( int a[], int size)
                                      |
void print (int array[], int t)       |

copy.c: line 27-34                    |original.c: line 9-16             [37]
void samp(int array[], int n, int m)  |void sample(int a[], int size, int sam
{                                     |{
 int i, j;                            | int i, j;
                                      |
 if(m> n || m<0)                      | if(sample_size > size || sample_size
  return;                             |  return;
                                      |
 for(i = 0 ; i< n; ++i)               | for(i = 0 ; i < size; i++)

copy.c: line 74-80                    |original.c: line 44-50            [32]
 n = n% ( s-m);                       | j%=(size-i);
 n += m;                              | j+= i;
  p= a[m];                            |  temp= a[i];
  a[m]=a[n];                          |  a[i]= a[j];
  a[n]=p;                             |  a[j]= temp;
}                                     | }
}                                     |}

copy.c: line 53-58                    |original.c: line 54-59            [29]
{                                     |{
 int i;                               | int i;
 for( i = 0; i< t; ++i)               | for( i = 0; i < size; ++i)
  printf("%d ", array[i]);            |  printf("%d ", a[i]);
 printf("\n");                        | printf("\n");
}                                     |}

copy.c: line 65-71                    |original.c: line 35-41            [28]
int n, m;                             | int temp;
                                      |
 for(m = 0; m< s; ++m)                | for( i = 0; i< size; ++i)
  a[m] =m+1;                          |  a[i] = i+1;
                                      |
                                      |
 for( m = 0; m < (s - 1) ; m++)       | for( i = 0; i < size - 1 ; ++i)
```

A SIM által kapott eredmény első ránézésre kevésbé átlátható, mint a Jplag esetében, mégis elegendő információt kapunk. Az eredeti program 77 sorból áll, míg a másolt 81ből. Mint látható sorok szerint megkapjuk a két program esetében a másolt részeket. Az eredeti program a másoltra 59%ban hasonlít, míg fordítva 65%

Kezdetben csak egy függvényt lesz meghagyva az eredeti programból, a többi kód helyére teljesen más programokból kerül kód beillesztésre.

A pusztán változónévbeli, tagolási különbségeket mindkét algoritmus egyaránt felfedezi.

A kezdeti program Print függvényét átültetve egy merőben különböző programba, hasonló eredmény tapasztalható a sim esetében, a két egyező függvényt megtalálja, míg a JPlag nem talál hasonlóságot.

Szerkezeti átalakítások tekintetében a SIM algoritmus hatékonyabban működik azon esetekben amikor a ciklusok átalakításra kerültek. A SIM felfedezi a for ciklusból képzett while ciklust és fordítva, míg a JPlag nem. Ugyanakkor a for ciklust átírva Do-While ciklusra nem talál egyezést – a JPlag sem.

Az úgynevezett üres kódok beszúrás – például egy ciklus amelyik csak visszaszámol – nem változtat a hasonlóságok megkeresésében. Így a lebukás elkerülését nem segíti ilyen sorok beillesztése a programkódba. Ezen sorok beszúrása ugyanúgy hatástalan függvényekben, eljárásokban való beszúrásakor, mint ciklusokon belül.

A program szerkezeti átalakítása eredményesnek bizonyul a plágium elkerülésére. A funkcionális kiemelés – függvény képzés esetében azonos működés mellett a SIM és JPlag egyaránt nem talál egyezést.

Logikai programozás tekintetében, logikai formulák felírásakor, két ekvivalens formula között különbséget tesz mind a SIM mind JPlag, így azonos formulák között a nem talál egyezést egyik algoritmus sem.

Elágaztató utasítások esetében a switch–case többszörös if-ekké való átalakítását nem fedezik fel az eddig vizsgált plágium ellenőrző algoritmusok.
Általánosságban tehát elmondható, hogy a plágiumészlelést ki lehet játszani olyan eszközök segítségével, melyek segítségével az eredeti program működése megváltozik. Leginkább úgy tekinthető, hogy azon esetekben, amikor az eredeti program működéséhez képest, a program vezérlése átadódik, nem derül fény a másolásra.

Viszont az ilyen átalakítások időigényesek(főleg hosszabb kód esetén) vagy pedig komplikáltak. Ezen dolgokat figyelembe véve, ki lehet jelenteni hogy aki biztosra megy annak érdekében hogy ne derüljön fény a csalásra, önerőből el tudná készíteni a programot.

VIII. Összegzés

A szakdolgozatom írása alatt behatóan megismerkedtem forráskód hasonlóság elemző programmal, illetve ezen programok működésével. Sikerült megismerkednem a különböző programok felépítésével, a kapcsolódó munkákkal, használatukkal, használati követelményeikkel.

Egyetemi tanulmányaim során, programozási beadandó feladatok esetében, több hallgatótársamtól is hallottam: „majd átírom és jó lesz”. Sok esetben viszont ez az átírás nem hozott sikereket, amit most már tudom, hogy miért nem. Csupán a változónevek kicserélése, ciklusok átalakítása nem hozza meg a kívánt eredményt. Viszont nagy valószínűséggel kijátszható az ellenőrző program megfelelő programozás technikai ismeretek elsajátításával. Ezen ismeretek tekintetében is több időt vehet igénybe a program átalakítása, mint a program önerőből való elkészítése.

A dolgozatom fő célja az volt, hogy bemutassak, illetve ismertessek olyan forráskód hasonlóság vizsgáló programokat, melyek mindenki számára elérhetők, interneten keresztül online módon használhatóak. A példaprogramok ekvivalens átalakításakor megtapasztaltam, hogy milyen szintű változtatásokat ismernek fel az ellenőrző programok, így már megértem, hogy például a beadandó programok esetében miért szükséges a lehetőség szerint minél több erőforrást fordítani a plágiumellenőrzésre.

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani mindenkinek, akinek tanítása, munkája, segítsége hozzájárult ahhoz, hogy ez a dolgozat elkészülhessen.

Köszönettel tartozom:

Noszály Csaba témavezetőként nyújtott segítségéért.

Irodalomjegyzék

- D Carrington, K Robinson and G Whale: Give: a System for Collecting and Testing Student Assignments
- G Whale: Identification of Program Similarity in Large Populations
- G Whale: Software Metrics and Plagiarism Detection
- Ming Li and Paul Vitanyi: An Introduction to Kolmogorov Complexity and Its Applications
- M. Li, J. Badger, X. Chen, S. Kwong, P. Kearney, H. Zhang: An information based sequence distance and its application to whole (mitochondria) genome distance
- X. Chen, B. Francia, M. Li, B. McKinnon, A. Seker: Shared Information and Program Plagiarism Detection
- http://dickgrune.com/Programs/similarity_tester/
- http://pam1.bcs.uwa.edu.au/~michaelw/YAP.html
- http://genome.math.uwaterloo.ca/SID/
- http://en.wikipedia.org/wiki/Kolmogorov_complexity
- http://en.wikipedia.org/wiki/LZW