Milan Freml, David Chalupa, Marek Mego,
Peter Mindek, Michal Noskovič a Matej Sabo

Podpora kontroly plagiarizmu

Tímový projekt I.

Študijné programy: Informačné systémy, Softvérové inžinierstvo
Vedúca projektu: Mgr. Daniela Chudá, PhD.
Ak. rok: 2009/10

Obsah

Kapitola 1

Úvod do problematiky

1.1 Čo je plagiarizmus?

Plagiarizmom sa nazýva využívanie cudzích prác, obrázkov alebo myšlienok bez toho, aby sa na ne autor patrične odvolával. Je to teda prisvojenie si časti cudzej práce. S plagiátorstvom sa dnes bežne stretávame hlavne v školstve, no neobchádza ani iné oblasti každodenného života. Postoj k nemu je v dnešnej spoločnosti čoraz menej tolerantný. Keďže sa s plagiátorstvom stretávame najčastejšie v školstve, jedná sa najmä o kopírovanie častí cudzích bakalárskych alebo diplomových prác bez odvolania sa na zdroj. S nástupom internetu sa možnosti získavania informácií rozšírili, čo paradoxne do veľkej miery napomohlo k rozšíreniu fenoménu plagiátorstva.

Autor, ktorý čerpá myšlienky z iných prác, má povinnosť uviesť k zdroju týchto myšlienok pôvodného autora a článok alebo prácu, z ktorej čerpal. To môže urobiť viacerými spôsobmi. Najčastejšie sa používajú úvodzovky na vymedzenie citovaného textu, poznámka pod čiarou a použitá literatúra, ktorá sa väčšinou uvádza na konci práce.

V spoločnosti existuje snaha autorov plagiátorov odhaliť. Táto snaha však naráža na tvorivosť a vynaliezavosť plagiátorov. Tí sa snažia cudzie myšlienky formulovať tak, aby prípadný čitateľ, respektíve osoba, ktorá prácu bude kontrolovať, nemala pochyby o tom, že práca je originálna a všetky myšlienky pochádzajú od autora práce.

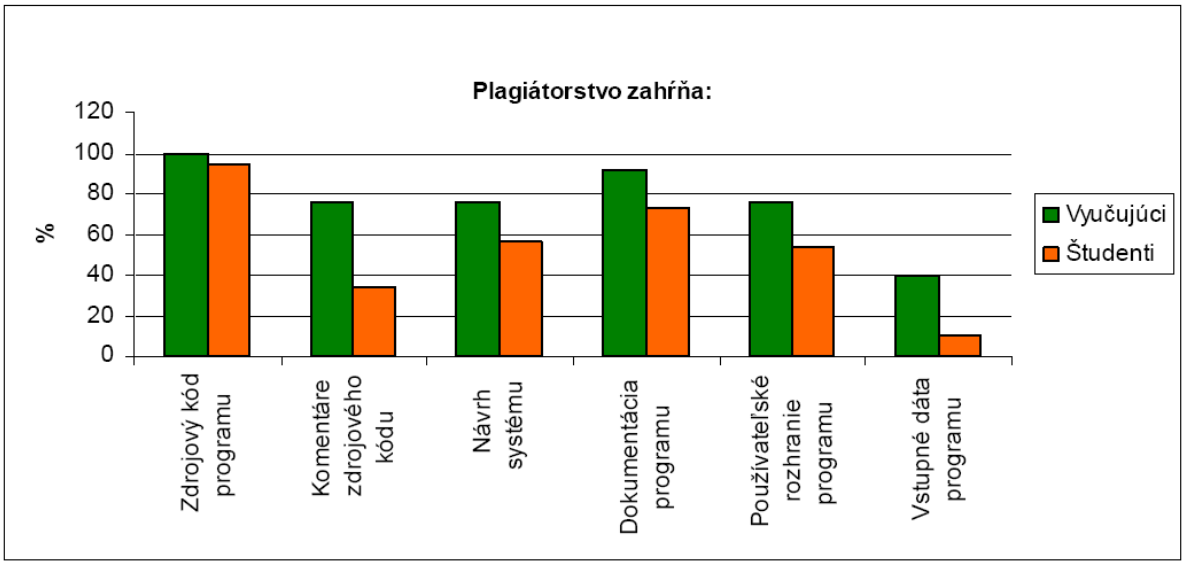
Za týmto účelom vzniklo veľa systémov a programov, ktoré majú za úlohu kontrolovať práce alebo zdrojové kódy a hľadať v nich podobnosť. Ak je podobnosť medzi dvoma dokumentmi nad určitou percentuálnou hranicou, sú tieto dokumenty označené ako podozrivé a je na učiteľovi, aby urobil následnú kontrolu. Niektoré programy mu môžu túto kontrolu značne uľahčiť, a to tým, že graficky vyznačia v dokumentoch časti, ktoré sa nachádzajú v oboch prácach.

1.2 Formy plagiarizmu a ich špecifiká

1.2.1 Zdrojové kódy

Používanie cudzích zdrojových kódov bez odkazovania sa na pôvodného autora sa zaraďuje taktiež medzi plagiátorstvo. Jedná sa totiž o použitie cudzích myšlienok, ktoré nie sú vyjadrené vo forme textu, ale vo forme zdrojového kódu. K softvérovým projektom sa často viažu aj textové dokumenty, ktoré obsahujú napríklad analýzu, dokumentáciu alebo návod na používanie výsledného programu. K týmto dielam, ktoré sú nedielnou súčasťou práce autora, sa taktiež vzťahuje jeho duševné vlastníctvo a nemôžu byť použité bez uvedenia zdroja.

Na obr. 1.1 môžeme vidieť výsledok ankety, ktorú vykonali Bianka Kováčová a Pavol Humay vo svojich diplomových prácach, v ktorých sa zaoberajú plagiátorstvom. Z nej je vidieť, čo študenti a profesori zahŕňajú pod pojem plagiátorstvo.



Obr. 1.1: Anketa na tému plagiátorstva[1]

Z prieskumu je zrejmé, že väčšina respondentov pokladá zdrojový kód za veľmi dôležitý a je potreba sa naň pri odvodených prácach odvolávať.

Zisťovanie podobnosti v zdrojových kódoch je veľmi náročná úloha, nakoľko existuje veľa metód, pomocou ktorých sa dá pravdepodobnosť na odhalenie podobnosti znížiť. Jednoduché plagiáty dokáže odhaliť aj osoba, ktorá porovnáva dva súbory a dokáže na 100% povedať, že sa jedná o plagiáty, cez zložitejšie až po tie najzložitejšie techniky, ktoré sa dajú odhaliť až za pomoci sofistikovaných nástrojov.

Všetky tieto techniky sa snažia pozmeniť pôvodný zdrojový kód tak, aby bol na pohľad

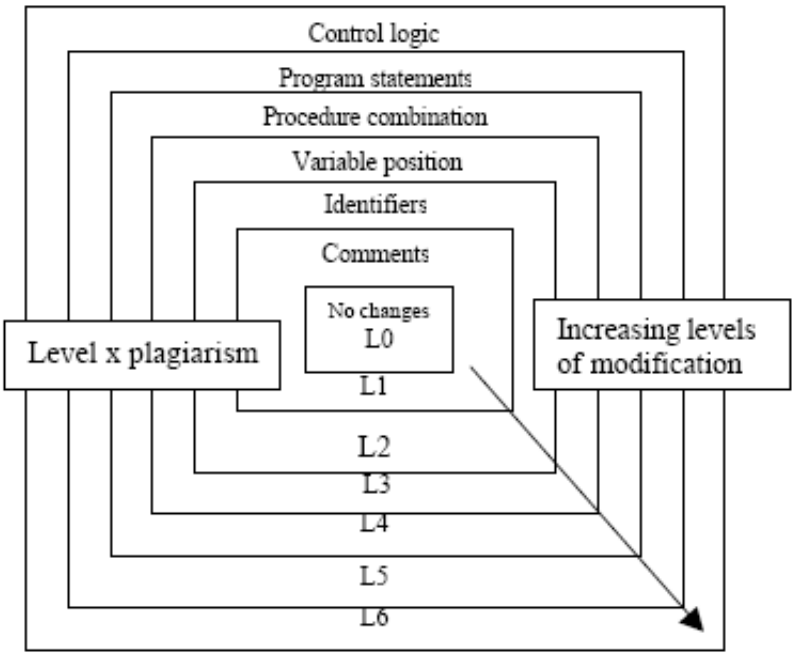
[1]Bianka Kováčová: Podobnosť a rozpoznanie opísaného programu. Diplomová práca, 2009.

rozdielny, ale výsledok programu bol nezmenený. To znamená, že pôvodný aj odvodený program na rovnaký vstup dajú rovnaký výstup.

Podľa Paula Clougha [1] môžeme zmeny v zdrojových kódoch rozdeliť podľa ich zložitosti na:

1. Žiadna zmena
2. Zmena komentárov
3. Zmena identifikátorov
4. Zmena pozície premenných
5. Kombinácia procedúr a funkcií
6. Správanie programu
7. Kontrola logiky programu

Predošlý zoznam sa dá graficky zobraziť pomocou prehľadného grafu, ktorý je zobrazený na obr. 1.2.



Obr. 1.2: Vykonané zmeny v zdrojových kódoch, prebrané z [1]

Z obrázka je zrejmé, že metód na maskovanie plagiátorstva je viac a líšia sa v zložitosti a v čase, ktorý je potrebný na ich zapracovanie.

Najmenej namáhavé sú postupy L1 a L2. Tu stačí v práci pozmeniť alebo vymazať len komentáre pôvodného autora. Taktiež je možné premenovať názvy premenných a funkcií, ktoré program používa. V prípade premenných plagiátori často menia ich typ. Napríklad prepíšu premennú, ktorá bola pôvodne deklarovaná ako celočíselný typ, na typ s desatinnou čiarkou. Medzi tieto jednoduché spôsoby môžeme taktiež zaradiť pridávanie medzier, tabulátorov, prázdnych riadkov do zdrojového kódu. Postupy, ktoré patria do tejto skupiny sú teda najjednoduchšie a preto je aj ich odhalenie najmenej zložité. Na ich odhalenie netreba často ani zložitý nástroj, ale stačí len manuálna kontrola.

Ďalšou skupinou zmien, sú zmeny, ktoré už vyžadujú zložitejšie prostriedky na odhalenie. Do tejto skupiny sa dajú zaradiť postupy L3 a L4. Jednou z možností je napríklad nahradenie podmienok za while cykly alebo naopak. Tento postup je znázornený na nasledujúcej ukážke 1.3.

```
3  if ($premenna){
4      doThis();
5  }
6
7  while ($premenna){
8      doThis();
9      break;
10 }
```

Obr. 1.3: Nahradenie podmienok za cykly

Jednou z možností, ako oklamať osobu, ktorá kontroluje prácu je nahradenie volaní funkcie samotnou implementáciou funkcie. Týmto sa stáva odvodená práca neprehľadnejšia a má samozrejme viacej riadkov. Do tejto skupiny môžeme taktiež zaradiť nahradenie matematických konštánt zápisom cez matematické operátory +,-,*,/. Konštantu 6 je možné zapísať napríklad ako 1*2+5-1.

Doteraz spomenuté postupy si nevyžadovali veľkú námahu a množstvo času na zrealizovanie. Preto aj ich odhalenie je viac menej nenáročné. Ostatné postupy z obrázku č. 1.2 si už vyžadujú určitú námahu a na vytvorenie plagiátu je potrebné väčšie množstvo času. Sem môžeme zaradiť napríklad premiešanie poradia vzájomne nezávislých príkazov, blokov vo vetvách switchu, alebo zámenou poradia konštrukcie if else. Pokiaľ chceme spraviť kód neprehľadný, je možné doň pridať zbytočný kód, ktorý sa buď nevykoná, alebo jeho vykonanie nebude mať vplyv na výsledok programu.

Takéto sofistikované metódy majú za následok, že program, ktorý má hľadať podobnosť medzi zdrojovými kódmi, túto podobnosť nenájde, alebo bude podobnosť veľmi malá, čo nebude zo strany učiteľa vnímané ako plagiát.

1.2.2 Slovenské texty

S problematikou plagiátorstva sa častejšie spájajú texty ako programové kódy a preto je plagiátorstvo textov aj viacej rozšírené. V dnešnej dobe má človek obrovské množstvo zdrojov, z ktorých môže čerpať informácie. Za týmto účelom vzniklo na internete veľa webových stránok, ktoré ponúkajú už vypracované referáty, alebo za určitý poplatok je majiteľ stránky ochotný vypracovať danú prácu. Tento systém je využívaný najmä na stredných školách, kde sa problém plagiarizmu rieši menej ako na vysokých školách. Samozrejme všetky práce sa líšia kvalitou a prevedením, ktorým chcel autor zatajiť, že väčšina jeho práce nie je pôvodná.

Takéto práce môžeme rozdeliť podľa zdrojov z ktorých vznikli na:

Kopírovaný plagiát
: Autor odvodenej práce v pôvodnej práci len zmenil meno pôvodného autora a názov samotnej práce. Tento spôsob môžeme zaradiť medzi najjednoduchšie formy plagiarizmu a preto sa takéto práce aj najľahšie odhaľujú či už za pomoci programu na to určeného, alebo učiteľa.

Modifikovaný plagiát
: V tomto diele je zmenených viacej častí. Okrem mena samotného autora a názvu práce sa v texte zmenia formulácie, poprípade sa pomení poradie kapitol. Do práce sa pridajú zdroje z internetu, najčastejšie z elektronických encyklopédií ako je napríklad Wikipédia. Do práce je možné pridať ďalšie obrázky, alebo niektoré existujúce obrázky vymazať.

Rešeršný plagiát
: V tomto diele sú použité myšlienky z viacerých zdrojov, ktoré sa týkajú problematiky diela. Samotné dielo je poskladané z viacerých originálnych prác iných autorov.

Odhaľovanie plagiátov v písanom texte

V nasledujúcom odseku sú popísané spôsoby, akými sa dá odhaliť plagiátorstvo v písaných textoch aj bez pomoci elektronických nástrojov. Každý jeden z nás má svoj špecifický slovník, ktorý používa pri písaní prác. Pomocou tejto vlastnosti sa dá jednoduchým spôsobom odhaliť práca, ktorá s veľkou pravdepodobnosťou nie je originálna, alebo obsahuje časti z iných prác. Medzi tieto vlastnosti patrí:

1. *Použitie slov*
   Každý jeden z nás používa svoj slovník a používa špecifické slová, ktoré nemusia používať ostatní. Analýzou týchto slov sa dá ľahko zistiť či je autorom textu samotný študent.

2. *Zmena slov*
   Tento bod má súvis s bodom 1. Ak študent vo svojej práci začne používať výrazy, ktoré predtým nepoužíval, je možné že daný text niekde prevzal, nakoľko človek často nemení svoj zaužívaný slovník a je pre každého viac menej jedinečný.

3. *Celistvost' textu*
   Tento bod hovorí o tom, či je text jeden celok, alebo niektoré jeho časti do textu nezapadajú.

4. *Interpunkcia a gramatické chyby*
   Pokiaľ majú dve rozdielne práce rovnaké gramatické chyby, alebo je v oboch v špecifických častiach alebo slovách vynechaná interpunkcia je veľmi pravdepodobné, že jedna z prác je skopírovaná.

5. *Stavba viet*
   Tak ako má každý jeden autor špecifickú slovnú zásobu, používa aj vlastné konštrukcie viet, v ktorých svoju slovnú zásobu používa. Autor môže používať veľmi dlhé vety, alebo naopak krátke vety, v ktorých zhrnie svoju myšlienku. Ak autor vo svojich prácach dlhodobo používa jeden štýl písanie viet, je málo pravdepodobné, že ho bezdôvodne v inej práci zmení. Indikátorom plagiátu môže byť aj výskyt jednotlivých slov, ktoré sa v práci nachádzajú. Či už sa jedná o počet jednotlivých slov, alebo ich špecifické umiestnenie.

Vyššie spomenuté body samozrejme neplatia vždy a pre všetky práce. Je to len pomôcka pre učiteľa a rýchly spôsob, akým môže nájsť podozrivé práce. Niektoré z týchto spôsobov napríklad nemôžu fungovať na prácach, pri ktorých má väčšia skupina za úlohu napísať prácu na rovnakú tému. Vo väčšine prác sa budú vyskytovať rovnaké špecifické pojmy, ktoré s prácou súvisia a bez nich nie je možné prácu napísať. Medzi tieto pojmy sa dajú zaradiť':

- Mená
- Dátumy
- Geografické lokácie
- Špecifické pojmy z problematiky, ktorou sa práca zaoberá

Naopak, existujú slová, ktorých výskyt v práci môže opravujúcemu indikovať, že autor práce ho prebral z inej práce. Tieto slová sa nazývajú hapax legomena [1] slová. Jedná sa o slová, ktoré sú v práci použité práve raz.

Parafrázovanie

"Parafrázovanie je lingvistická operácia, ktorá spočíva vo vyjadrení toho istého obsahu, ako má východiskový výraz, konštrukcia alebo výpoveď, inými výrazovými prostriedkami." [2] Medzi najčastejšie druhy parafrázovania zaraďujeme:

- Použitie synoným vo vete – Niektoré kľúčové slová sú nahradené ich synonymami.
  Pôvodná veta:
  Bol brutálne napadnutý a zavraždený.

Pozmenená veta:
Muž bol brutálne napadnutý a zabitý.

- Zmenenie typu vety
  Pri tomto spôsobe sa do pôvodnej vety vkladajú takzvané transition slová ako napríklad ale, takže, pretože, kvôli. Pridanie týchto slov často zmení typ vety.
  Pôvodná veta:
  Technológia môže spôsobit' katastrofu.
  Pozmenená veta:
  Katastrofa je spôsobená kvôli technológiám.

- Zmenenie slovosledu vo vete
  V originálnej vete sa pomení poradie slov, tak aby nová veta mala rovnaký význam.
  Pôvodná veta:
  Technológie môžu zlepšit' kvalitu života ak si dôkladne plánujeme budúcnost'.
  Pozmenená veta:
  Ak si dôkladne plánujeme budúcnost', technológie nám môžu zlepšit' kvalitu života [1]

1.3 Prehľad prístupov k detekcii plagiarizmu

1.3.1 Prehľad metód spracovania zdrojových kódov

Pre všetky metódy platí, že lepšie výsledky vykazujú po predspracovaní kódu, za čo môžeme považovat' jeho tokenizáciu. Tokenizácia môže byt' implementovaná rôznymi spôsobmi a výsledok predspracovania kódu má významný vplyv na určenie podobnosti.

Metódy spracovania zdrojového kódu môžeme rozdelit' podľa prístupu k zdrojovému kódu na:

- *Štrukturálne* – metódy hľadajúce v kóde rovnaké štrukturálne prvky
- *Neštrukturálne* – metódy hľadajúce jednoducho podobnosti textu zdrojových kódov

Medzi štrukturálne metódy môžeme zaradit' napríklad hľadanie najväčšieho spoločného podgrafu.
Neštrukturálne metódy d'alej môžeme rozdelit' na:

- Štandardné
- Metódy pre porovnávanie textov

Do štandardných metód na porovnávanie zdrojových kódov môžeme zaradit' Rabinov-Karpov algoritmus, prípadne Greedy String Tiling.

Na zist'ovanie plagiarizmu v zdrojových kódoch sa však po predspracovaní často používajú aj metódy na spracovanie textov. Ich prehľad sa nachádza v nasledujúcej kapitole.

1.3.2 Prehľad metód spracovania slovenských textov

Zdeněk Češka uvádza rozdelenie metód pre detekciu plagiarizmu v textových dokumentoch na základe dvoch hlavných kritérií [3]. Podľa zložitosti metódy rozdeľujeme nasledovne:

- *Povrchné* – Metódy nevyužívajú žiadne lingvistické pravidlá, pracujú iba s pôvodným textom.
- *Štrukturálne* – Metódy sa snažia čiastočne porozumieť dokumentu.

Druhé kritérium predstavuje počet spracovaných dokumentov. Na základe spomínaného kritéria môžeme metódy zaradiť do nasledujúcich štyroch kategórií:

- *Jednotlivé* – Metódy spracovávajú v rovnakom čase iba jeden dokument.
- *Párové* – Metódy vykonávajú spracovanie dvoch dokumentov naraz.
- *Multidimenzionálne* – Metódy spracovávajú aspoň tri dokumenty naraz.
- *Korpálne* – Metódy spracovávajú naraz celú skúmanú vzorku.

Pozrime sa detailnejšie na rozdelenie podľa prvého kritéria. Povrchné metódy spravidla pracujú priamo s dokumentom, ktorý skúmame. Štrukturálne metódy naopak pred samotným spracovaním textu vykonávajú predspracovanie. V slovenskom jazyku toto predspracovanie pozostáva z dvoch činností:

1. *Odstránenie stop-slov* – Existujú slová, ktoré tvoria veľkú časť dokumentov v prirodzenom jazyku, ale principiálne neovplyvňujú význam textu. Spomínané slová nazývame stop-slovami a môžeme ich jednoducho vylúčiť z textu.
2. *Lematizácia* – Pod pojmom lematizácie rozumieme prevod slova na základný tvar.

Druhé kritérium rozdeľuje metódy na základe počtu dokumentov, ktoré spracovávame naraz. Vzhľadom k tomu, že si kladieme za cieľ skúmať podobnosť, prirodzenými metódami sú predovšetkým metódy párové, z ktorých si predstavíme napr. metódu 3-gramy alebo LCS. Vydáme sa ale aj ku korpálnej metóde latentnej sémantickej analýzy.

Kapitola 2

Metódy a prístupy k detekcii plagiarizmu

2.1 Predspracovanie zdrojových kódov a dokumentácie

2.1.1 Stop-slová a lematizácia v slovenských textoch

Predspracovanie slovenských textov pre potreby bližšej analýzy textu prebieha v dvoch fázach.

Prvú fázu tvorí odstránenie tzv. stop slov, sémanticky nevýznamných slov. Sú to zväčša krátke slová, predložky, spojky, častice prípadne samostatné číslice.

V druhej časti, nazývanej lematizácia, sa každé slovo zmení na jeho slovníkový tvar prípadne na koreň pôvodného slova. Túto transformáciu možno dosiahnuť niekoľkými spôsobmi.

Veľmi jednoduchá ale účinná metóda je použitie databázy všetkých slov v slovníku a ich príslušných možných tvarov. Následne možno spárovať ľubovoľné slovo obsiahnuté v slovníku s jeho základným tvarom. Problém nastane v prípade slov, ktoré nie sú obsiahnuté v slovníku. Sú to prevažne nové slová, mená, priezviská, názvy firiem. Koreň slova z takýchto slov je možné získať iba algoritmickou detekciou, ktorá však nemusí byť vždy presná.

Ďalším problémom môže byť nutnosť rýchleho prístupu do databázy a vyhľadávanie v nej. Dôležitý je práve fakt, že samotný slovník slovenského jazyka obsahuje približne 150000 slov v základnom tvare, ktoré majú ďalších približne 5 miliónov tvarov, čo nie je málo. Možným úskalím sa javí aj samotná dostupnosť a prítomnosť spomínanej databázy v systémoch, kde je lematizácia potrebná. Aj zo spomenutých dôvodov je snaha vyvinúť algoritmické riešenie, ktoré nie je založené len na úplnosti a dostupnosti databázy všetkých slov.

Potrebné riešenie existuje v anglickom jazyku implementované ako Porterov algoritmus. Tento prístup je založený na odstraňovaní známych prípon zo slov. V slovenčine je však

tento spôsob takmer nemožný prípadne veľmi neefektívny, pretože by bolo treba odvodiť všetky pravidlá na ohýbanie slov. V dokumente [4] je však navrhnutý zaujímavý spôsob riešenia uvedeného problému pomocou "cross-over" algoritmu. Tento algoritmus predpokladá, že máme k dispozícii databázu všetkých slovenských slov v ich základnom tvare a ďalej reprezentatívnu vzorku slov a ich tvarov po skloňovaní rozložených na slabiky.

Algoritmus funguje na základe podobného skloňovania slov. K zadanému slovu sa vždy vyberie slovo z databázy s rovnakou koncovkou. Ďalej sa zistí základný tvar vybraného slova, pričom sa sleduje zmena koncovky slova pri tomto prechode a následne sa algoritmus snaží vykonať podobnú transformáciu aj na zadanom slove. Výsledné slovo sa nakoniec skontroluje v databáze všetkých slov v základnom tvare. Ak sa také slovo našlo, predpokladáme, transformácia prebehla úspešne.

Kvalita výsledku samozrejme závisí od správneho výberu slova s rovnakým skloňovaním, čo však možno zefektívniť ak dané slovo bude mať napríklad ten istý rod. Samozrejme výsledok nie je vždy korektný, známe sú napríklad problémy pri jednoslabičných slovách, pri zmene základu slova pri skloňovaní a iné. Dôležité však je, že testovaním [4] sa ukázalo, že správnosť výsledku pri použití spomenutého algoritmu je vyše 90%. Táto hodnota je veľmi dobrá aj vzhľadom nato, že nepotrebujeme databázu všetkých slov a ich skloňovaní.

Príklad predspracovania textu:

1. vyradenie stop slov z textu
   Mama mi povedala, aby som kúpil 10 rožkov a priniesol aspoň jednu limonádu.
   Mama povedala kúpil 10 rožkov priniesol aspoň jednu limonádu.

2. úprava slov na základný tvar
   Mama povedať kúpiť 10 rožok priniesť aspoň jedna limonáda.

2.1.2 Tokenizácia zdrojového kódu

Tokenizácia je proces, pri ktorom sa snažíme inštrukcie pôvodného kódu nahradiť inými symbolmi tak, aby sme ich význam zovšeobecnili. To znamená, že viacerým pôvodným symbolom môžeme priradiť jeden a ten istý symbol (token) . Týmto spôsobom zabezpečíme, že ak plagiátor zmení určitú časť kódu za inú funkčne ekvivalentnú časť, tak po predspracovaní oboch kódov tokenizáciou získame identické výsledky. Pri tokenizáciu časti kódu `while x>3;` môžeme podľa stupňa generalizácie získať napríklad takéto výsledky:

```
while VARIABLE > 3
while
CYCLE_begin
```

Je teda zrejmé, že samotný úspech tokenizácie závisí aj od zvoleného stupňa generalizácie.

Nespornou výhodou tokenizácie je skutočnosť, že takýmto spôsobom môžeme úspešne porovnávať zdrojové kódy, ktoré sú naprogramované inými programovacími jazykmi. V každom jazyku existuje istá forma podmienok, cyklov, metód a podobne. Tieto prvky sa pri tokenizácii prevedú na jeden, pre všetky jazyky rovnaký symbol.

Pre lepšie pochopenie možných tokenov je uvedená nasledujúca tabuľka, kde sa jednotlivým identifikátorom v zdrojovom kóde priradí symbol, token.

Tabuľka 2.1: Tabuľka tokenov

Identifikátor v zdrojovom kóde	**Token**
`Int, float, double, boolean, byte, long, short`	NUMBER
`String, StringBuffer, StringBuilder, char`	STRING
`while, do-while, for, repeat-until, if then, else`	BLOCK
`<, >, <=,>=, ==, !=`	COMPARATOR
ľubovoľná premenná	VARIABLE
`+, -, *, /,`	OPERATOR

2.2 Porovnávanie zdrojových kódov

2.2.1 Rabinov-Karpov algoritmus

Rabinov–Karpov algoritmus patrí medzi klasické prístupy k riešeniu problému vyhľadávania podreťazcov. V prvom kroku vypočítame hash reťazca i podreťazca, tým sa konvertuje úloha porovnávania reťazcov na úlohu porovnávania hashov s kontrolou zhody reťazcov. Následne iterujeme nad reťazcom a počítame hashe pre všetky podreťazce typu 1..m, 2..m+1, atď. Porovnávame hashe. V prípade zhody porovnávame samotné reťazce. Algoritmus môžeme vyjadriť v nasledujúcom pseudokóde.

Algoritmus 2.2.1 – Rabinov-Karpov algoritmus [5]

```
RabinKarp(string s[1..n], string sub[1..m])
  hsub = hash(sub[1..m]); hs = hash(s[1..m])
  for i=1..(n-m+1)
    if (hs == hsub)
      if (s[i..i+m-1] == sub)
        return i
      hs = hash(s[i+1..i+m])
  return not found
```

2.2.2 Metóda Greedy String Tiling

Greedy String Tiling predstavuje metódu na porovnávanie tokenizovaných zdrojových kódov, v ktorej implementácii sa spravidla využíva Rabinov-Karpov algoritmus. Cyklicky iterujeme nad tokenmi, využívame tzv. zaznačovanie tokenov, na začiatku každej iterácie sú všetky tokeny v reťazcoch A a B nezaznačené. Pre všetky neoznačené tokeny v A a B potom hľadáme čo najdlhšiu sekvenciu zhodných tokenov. Vytvoríme množinu takýchto sekvenciu, ktorej tokeny následne zaznačíme. Pokračujeme ďalšou iteráciou. Algoritmus špecifikujeme nasledujúcim pseudokódom.

Algoritmus 2.2.2 – Greedy String Tiling [6]

```
GreedyStringTiling(TokenVector A, TokenVector B)
  tiles = {};
  do
     maxMatch = MINML;
     matches = {};
     Forall unmarked tokens A[a] in A
       Forall unmarked tokens B[b] in B
         j = 0;
         while (A[a+j] == B[b+j] && unmarked(A[a+j]) && unmarked(B[b+j]))
           j++;
         if (j == maxMatch)
           matches = matches + match(a,b,j);
         else if (j > maxMatch)
           matches = {match(a,b,j)};
           maxMatch = j;
     Forall match(a,b,maxMatch) in matches
       For j = 0..(maxMatch-1)
         markToken(A[a+j]);
         markToken(B[b+j]);
       tiles = tiles + match(a,b,maxMatch);
   while (maxMatch > MINML);
   return tiles;
```

2.3 Porovnávanie slovenských textov

2.3.1 Metóda N-gramov

Jednu z najjednoduchších a paradoxne aj najúčinnejších metód porovnávania fráz v textoch predstavuje metóda N-gramov. Pod N-gramom budeme rozumieť N-ticu slov, ktoré v texte nasledujú bezprostredne za sebou.

Uvažujme napr. metódu 3-gramov. Pri porovnávaní dvoch súborov vtedy vyberieme

1., 2. a 3. slovo z obidvoch súborov a porovnáme. Celý proces opakujeme nad všetkými možnými dvojicami 3-gramov, ktoré v daných súboroch existujú (vrátane prekrývajúcich sa 3-gramov).

Porovnanie dvoch N-gramov možno najjednoduchšie vykonať prostredníctvom naivnej metódy, t.j. porovnávať frázy postupne po znakoch.

2.3.2 Najdlhší spoločný podreťazec

Problém najdlhšieho spoločného podreťazca dvoch reťazcov (angl. Longest Common Substring, ďalej len LCS) patrí medzi typické aplikácie dynamického programovania. Algoritmus 2.3.1 popisuje, ako je využívaná tabuľka – dvojrozmerné pole back na memorizáciu výsledkov pre i-tu pozíciu prvého a j-tu pozíciu druhého reťazca znakov. Rekurzívna formulácia algoritmu je nasledujúca [7]:

$$c[i,j] = \begin{cases} 0 & ak\ i = 0 \vee j = 0 \\ c[i-1, j-1] + 1 & ak\ i,j > 0 \wedge x_i = y_j\,. \\ \max\{c[i.j-1], c[i-1,j]\} & inak \end{cases} \tag{2.1}$$

Algoritmus 2.3.1 – Hľadanie LCS pomocou dynamického programovania „zdola nahor“ [7]

```
LCS(X,Y)
  m <- LENGTH[X]; n <- LENGTH[Y];
  for i <- 1 to m, do c[i,0] <- 0;
  for j <- 0 to n, do c[0,j] <- 0;
  back <- c;

  for i <- 1 to m, do
    for j <- 1 to n do
      if (x[i] = y[j])
        c[i,j] <- c[i-1, j-1]+1;
        back[i,j] <- "UP&LEFT";
      else if (c[i-1,j] >= c[i,j-1])
        c[i,j] <- c[i-1,j];
        back[i,j] <- "UP";
      else
        c[i,j] <- c[i,j-1];
        back[i,j] <- "LEFT";
  return c and back
```

Metóda najdlhšieho spoločného podreťazca vykazuje pre detekciu plagiarizmu relatívne nízke percentuálne ohodnotenia podobnosti, preto je potrebné dbať na to, aby bola nastavená dostatočne nízka hranica, po ktorej prekročení systém vyhlási súbory za podobné. Spomínaná hranica sa môže pohybovať približne okolo 3%.

Algoritmus 2.3.2 – LCS pomocou Generalised Suffix tree

Problém najdlhšieho spoločného podreťazca dvoch reťazcov (LCS) je možné implementovať aj pomocou generalizovaného stromu prípon (GST). Najprv sa vytvorí strom reťazcov, čo má zložitosť O(n) za predpokladu že máme konštantú veľkosť použitej abecedy. Následne sa nájde najhlbší vnútorný uzol, z ktorého vedú cesty do listov všetkých reťazcov v podstrome pod ním. Napríklad ak chceme nájsť najdlhší spoločný podreťazec pre slová ÄBBA", ÄBABä "BABA", rozšírime ich o unikátne znaky na konci stringov. Dostaneme napríklad ÄBBA0", ÄBAB1ä "BABA2". Zostavíme teraz GST, ktorého listami sú buď priamo čísla 0, 1, alebo 2 alebo reťazce ktoré obsahujú práve jedno z týchto čísel. Teraz hľadáme najhlbší uzol, tak aby sme sa z neho ešte vedeli dostať do aspoň jedného listu ktoré obsahuje číslo 0, aspoň jedného obsahujúce číslo 1 a podobne aj číslo 2. Zistíme že do takýchto uzlov sa vieme dostať zo štyroch rôznych uzlov. Prve dva uzly sú hneď pod koreňom a dávajú podreťazce Ää "B". Ďalšie dva sú ich nasledovníci a dávajú podreťazce ÄBä "BA". Tieto dva podreťazce sú teda riešením problému.

2.3.3 Frekvencie slov v dokumentoch

Ďalší prístup k detekcii podobnosti v slovenských textoch predstavuje počítanie frekvencií, s ktorými sa jednotlivé slová nachádzajú v skúmaných textoch. Jednoduchú metódu, založenú na porovnávaní týchto početností, nazývame metódou TF.

Modifikáciu metódy TF môžeme získať tak, že vylúčime slová, ktoré sa vyskytujú v danej vzorke veľmi často, uvažujeme následne iba frekvencie výskytov vzácnejších slov, čo pri rovnakej téme článkov vedie na rozdiel od LCS práve k vysokým podobnostiam. Jedná sa o metódu inverznej frekvencie slov v dokumentoch (angl. Inverse Document Frequency, skr. IDF).

2.3.4 Latentná sémantická analýza

V nasledujúcich riadkoch sa oboznámime s metódou, ktorá už využíva zložitejší prístup k detekcii plagiarizmu v dokumentoch. Latentná sématická analýza sa, ako alternatívna metóda, zakladá na reprezentácii vzorky dokumentov pomocou lineárneho modelu. Koncepciu metódy a jej využiteľnosť predstavil Zdeněk Češka [3].

Fáza predspracovania v metóde LSA

Predspracovanie prirodzeného jazyka prebieha v LSA rovnako, ako sme opísali v časti 2.1.1, zmažú sa stop-slová, vykoná sa lematizácia. Extrahujeme z dokumentov frázy ako N-gramy, pričom odporúčaná hodnota N sa pohybuje približne v intervale od 5 do 10. Následne sa zo vzorky odstránia frázy, ktoré sa nachádzajú iba v jednom dokumente. Naopak, časté frázy odstraňujeme vtedy, keď ich zastúpenie v dokumentoch presahuje určitú hranicu. Ostatné frázy sa použijú na tvorbu matice A.

Model dokumentov a jeho vyjadrenie pomocou matice

Budeme mať maticu A rozmeru n x m. Jej riadky popisujú model, ktorý určuje, ktoré frázy dokument reprezentovaný i-tym stĺpcom matice obsahuje. V poradí j-ty riadok matice reprezentuje frázu j. Potom ohodnotíme prvky matice hodnotami podľa nasledujúceho vzťahu [3]:

$$a[i,j] = \begin{cases} \frac{1}{2} + \frac{PF[i,j]\log\left(\frac{|n|}{DF[j]}\right)}{2\max_i\{PF[i,j]\}log(|n|)} & ak\ fraza\ j\ je\ v\ dokumente\ i \\ 0 & inak \end{cases}. \tag{2.2}$$

kde PF[i,j] znamená frekvenciu výskytu frázy j v dokumente i, DF[j] znamená počet dokumentov, v ktorých sa nachádza fráza j a |n| je počet skúmaných dokumentov. Zložitosť vzorca pramení z požiadavky, aby sa koeficienty matice pohybovali v intervale <0.5;1>, čím sa dosiahne výraznejší "odstup" od nuly, ktorá reprezentuje absenciu frázy v danom dokumente.

Algoritmus singulárnej dekompozície matíc

Definujme vlastné číslo štvorcovej matice M ako taký skalár λ, že platí:

$$Mx = \lambda x \tag{2.3}$$

pre nejaký vektor x, ktorý nazývame vlastným.

Nech A je teraz obdĺžniková matica rozmeru n x m (napr. naša matica modelu dokumentov). Uvažujme matice AA^T a A^TA, kde operátor T značí transpozíciu matice. Uvedené matice sú obidve evidentne štvorcové (rozmerov m x m, resp. n x n). Pod singulárnou dekompozíciou matice A rozumieme súčin:

$$A = U\Sigma VT. \tag{2.4}$$

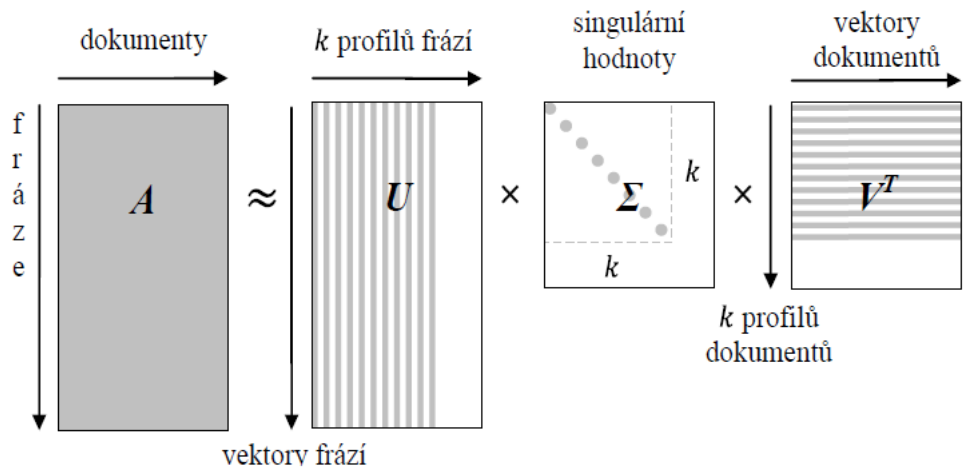
Postulujme, že matice U a V majú k spoločných vlastných čísel. Matica Σ rozmeru k x k obsahuje spomínané vlastné čísla AA^T a A^TA, nazývame ich singulárnymi číslami matice A. Matice U rozmeru n x k a V rozmeru m x k potom v stĺpcoch obsahujú vlastné vektory, prislúchajúce k maticiam AA^T , resp. A^TA. Spomínané vektory nazývame takisto singulárnymi vektormi matice A.

Interpretácia singulárnej dekompozície vo vzťahu k LSA je zobrazená na nasledujúcom obrázku. Matice U, resp. V obsahujú vo svojich stĺpcoch vektory - profily jednotlivých fráz, resp. dokumentov.

Metódu singulárnej dekompozície možno implementovať viacerými spôsobmi. Niekoľko z nich priamo ponúka vedecká knižnica pre jazyk C – GNU Scientific Library [8].

Výpočet korelačnej matice podobnosti dokumentov

Profily, ktoré získame singulárnou dekompozíciou matice dokumentov nám ešte o hľadanej podobnosti priamo nič nepovedia. Preto je potrebné pristúpiť k záverečnej fáze LSA.



Obr. 2.1: Singulárna dekompozícia v kontexte LSA, prebrané z [3]

V prvom kroku vynásobíme jednotlivé profily dokumentov zodpovedajúcimi singulárnymi číslami, čo môžeme zapísať pomocou maticového súčinu:

$$B = \Sigma V^T. \tag{2.5}$$

Stĺpce matice B teraz musíme normalizovať, t.j. každý stĺpcový vektor musíme vynásobiť skalárom tak, aby jeho veľkosť bola rovná 1. Vznikne nám tak matica ||B||. Položme:

$$S* = ||B||T||B||. \tag{2.6}$$

Uvedeným výpočtom už získame maticu, ktorej prvky už popisujú podobnosti medzi jednotlivými dokumentmi vo vzorke. Vzhľadom k využitiu filtra vo fáze predspracovania sú hodnoty ale skreslené, preto musíme vykonať nasledujúci výpočet, ktorým získame finálnu korelačnú maticu podobnosti dokumentov S:

$$S(X,Y) = S^*(X,Y)\sqrt{\frac{|ph_{orig}(X)| * |ph_{orig}(Y)|}{|ph_{red}(X)| * |ph_{red}(Y)|}}. \tag{2.7}$$

V uvedenej rovnici phorig predstavuje počet pôvodných fráz a phorig počet fráz, ktoré v dokumente zostávajú po redukcii.

2.4 Alternatívne prístupy k detekcii plagiarizmu

Kontrola zdrojových kódov – inštrukcie

Tokenizácia zdrojového kódu by mohla prebiehať na úrovni jednotlivých príkazov – každý príkaz by bol jeden token. Toto by mohlo byť realizované prevodom zdrojového kódu do jazyku symbolických inštrukcií, prípadne do postupnosti pseudoinštrukcií, sada ktorých by

bola navrhnutá špeciálne pre tento účel. Pri tomto prevode by boli cykly a podmienky zapísané pomocou skokov, podobne, ako je to pri kompilovaní programu do strojového kódu.

Vďaka tomu by sa zvýšila schopnosť programu odhaľovať techniky používané plagiátormi, akými sú napríklad prepísanie while cyklov na for cykly a podobne. Ďalšou možnosťou je nahradenie volaní lokálnych funkcií ich implementáciou, v prípade, že je to možné. Na takto predspracovanom zdrojovom kóde by sa mohlo vykonať porovnanie pomocou ľubovoľnej metódy na porovnávanie textov – jednotlivé pseudoinštrukcie by boli spracovávané rovnako ako slová v základnom tvare. Bol by však kladený dôraz predovšetkým na rovnaké postupnosti inštrukcií, pretože rovnaké skupiny inštrukcií s rôznym poradím pravdepodobne neznamenajú rovnaké zdrojové kódy.

Samotné inštrukcie by neobsahovali dáta, ktoré majú pri kontrole plagiarizmu veľkú výpovednú hodnotu (až na niekoľko špeciálnych prípadov).

Kontrola zdrojových kódov – graf

Ďalšou metódou kontroly by mohlo byť vytvorenie grafovej reprezentácie kontrolovaného zdrojového kódu. Vrcholy grafu by tvorili dátové štruktúry a bloky kódu, hrany by predstavovali vzťahy medzi týmito entitami. V takomto grafe by sa hľadal najväčší spoločný podgraf, pričom by sa brali do úvahy taktiež typy jednotlivých vrcholov. Typmi sa myslí napríklad funkcia, štruktúra, if-podmienka, for-cyklus, a podobne.

String-blurring algoritmus

V našom systéme chceme experimentovať s naším vlastným algoritmom, ktorý sme nazvali String-blurring. Jeho podstatou je vynechanie všetkých bielych znakov z textu (medzery, tabulátory, konce riadkov, a pod.) a uloženie jednotlivých znakov do jednorozmerného poľa. Každý znak je v tomto poli reprezentovaný svojou číselnou hodnotou. Pole je následne „rozmazané“ – hodnota každého prvku poľa je nahradená hodnotou váženého priemeru hodnôt okolitých prvkov, pričom váhy sú určené normálnym (gaussovským) rozdelením.

Z dvoch takto predspracovaných polí sa vyberú všetky možné prekrývajúce sa podpostupnosti, ktoré sa od seba odčítajú. Vo výsledkoch tvorených rozdielom hodnôt podpostupností sa následne vyhľadá najdlhšia podpostupnosť prvkov, ktorých absolútna hodnota je menšia ako zadaný prah. Dĺžka tejto podpostupnosti je výstupom algoritmu a tvorí index podobnosti daných súborov.

Okrem prahu podobnosti, na základe ktorého sa vyhľadávajú najdlhšie podpostupnosti, je možné meniť silu rozmazania – počet susedných prvkov, ktoré tvoria vážený priemer novej hodnoty prvku poľa.

Algoritmus je pomerne odolný voči malým a početným zmenám textu, ktoré by potenciálne dokázali oklamať algoritmus Longest common substring. Relevantnosť výsledkov však klesá so silou rozmazania, ktorá presiahne určitú hodnotu. Táto hodnota musí byť zistená experimentálne.

Kapitola 3

Analýza existujúcich riešení

Pre lepšie pochopenie vlastností, aké má budúca aplikácia poskytovať sa analyzovali niektoré v súčasnosti používané nástroje na detekciu plagiarizmu. Menovite SIDPlag, Jplag, SIM, MOSS, YAP, Plades a Sherlock. Niektoré sú primárne určené na zdrojové kódy, iné na porovnávanie textov. Všetky nástroje sme testovali na rovnakých vstupoch.

Slovenské texty sme získali na základe predchádzajúcich prác kolegov od Daniely Chudej. Pri zdrojových kódoch sa na chvíľu zastavíme. V kapitole 1.2 sú spomenuté základné princípy vytvárania plagiátov z originálnych zdrojových kódov. V nasledujúcej tabuľke je uvedená metodika pre tvorbu fiktívnych plagiátov za účelom testovania už existujúcich programov na detekciu plagiátorstva ako i vytváraného nástroja v tomto projekte. Týmto spôsobom sa zabezpečia objektívne a porovnateľné výsledky. V tabuľke sú zobrazené štyri úrovne plagiátorstva pričom každý nasledujúci zahŕňa aj zmeny predošlého plagiátu.

Tabuľka 3.1: Úrovne plagiátorstva

amatér	premenovanie premenných, prehodenie funkcií, metód, zmena typu premenných, zmena odsadenia, pridávanie prázdnych riadkov, medzier, zmena komentárov
študent	prepis podmienok na while cykly a naopak, nahradenie volanie funkcií ich implementáciou, nahradenie konštánt, define
pokročilý	prepis jednoduchých operácii na vlastné funkcie, zmena poradia vzájomne nezávislých inštrukcií, blokov vo `switch`, poradie `if {} else {}`, čiastočné pridanie a odobratie zbytočného kódu a úprava podmienok pomocou zbytočného kódu
guru	pridanie zbytočného kódu za každú operáciu, drobná zmena funkcionality

3.1 Podobnosť zdrojových kódov

Táto kapitola je zameraná na nástroje primárne určené na podobnosť zdrojových kódov, to však neznamená, že sa nedajú použiť na detekciu plagiarizmu v slovenských textoch. Túto skutočnosť naznačujú aj niektoré dosiahnuté výsledky.

3.1.1 SIDPlag

Implementácia: Je programovaný v jazyku Java.

Algoritmus: Ideou algoritmu podľa dokumentácie je rozloženie si problému na dve časti. V prvej časti nástroj tokenizuje kód, tak aby zamenil všetky jednoducho zameniteľné prvky kódu na jeden všeobecný symbol. V ďalšej časti je použitý voľne dostupný algoritmus na porovnávanie textov Greedy String Tiling. Z uvedeného vyplýva, že zaujímavou časťou riešenia je práve časť tokenizácie. Program používa základné symboly pre tokenizáciu ako sú: zámena premenných, cyklov či zrušenie nepodstatných častí. Takýto druh tokenizácie by bol dostačujúci, ak by boli používateľovi poskytnuté možnosti pre pridávanie vlastných symbolov, čo je však len ťažko realizovateľné.

Podmienky použitia: Pre použitie SIDPlag-u musí používateľ mať nainštalovanú JRE v bližšie nešpecifikovanej verzií. Jedná sa voľne dostupnú aplikáciu, nie je nutná registrácia.

Použitie: Program SIDplag , slúži na porovnávanie zdrojových kódov jazyku java a C. Ďalej program umožňuje aj porovnávanie bežného textu, čo vyplýva z implementačného algoritmu programu, avšak o tejto funkcii nikde nie je zmienka.

Vizualizácia: SIDPlag zobrazuje podobnosti dvojíc súborov v prehľadnej tabuľke. Je možné vybrať konkrétnu dvojicu a zobraziť ich obsah vedľa seba. Tiež farebne vyznačuje rovnaké časti.

Klady:

- pri relatívne jednoduchom algoritme sa dajú získať akceptovateľné a vierohodné výsledky
- prehľadné zobrazenie výsledkov

Zápory:

- používateľovi nie sú poskytnuté možnosti pre pridávanie vlastných symbolov pre tokenizáciu
- program sa dá teda ľahko obísť potencionálnym plagiátorom, pridaním zbytočného kódu
- problém, ak plagiátor vkladá do kódu nič nevykonajúce, zbytočné časti implementácie
- program až príliš zaujatý voči plagiátorom, keď udával mieru podobnosti vyššiu ako 100%, táto situácia nastala vtedy, keď program chybne vyhodnotil podobnosť stringov (531%)

- nutnosť zadávania zdrojového adresára pri každom novom porovnávaní
- nekorektné určovanie veľkosti okna, pri príliš dlhej ceste k súborom je potrebné ručne okno zväčšovať, inak sú nedostupné tlačidlá

Výsledky testovania: Výsledky sú dobré pri jednoduchých algoritmoch. Pri pridávaní zbytočných inštrukcií však strácajú na relevantnosti. Výsledky testov sú v tabuľke 3.2.

Tabuľka 3.2: Výsledky SIDPlag

C

Originál – plagiát	Podobnosť
program1 – program1 (plagiát amatérsky)	86.71%
program1 – program1 (plagiát študentský)	71.79%
program1 – program1 (plagiát pokročilý)	nedetekované
program1 – program1 (plagiát „guru“)	nedetekované
program2 – program2 (plagiát amatérsky)	chybové hlásenie
program2 – program2 (plagiát študentský)	chybové hlásenie
program2 – program2 (plagiát pokročilý)	chybové hlásenie
program2 – program2 (plagiát „guru“)	chybové hlásenie

Java

Originál – plagiát	Podobnosť
program1 – program1 (plagiát amatérsky)	59%
program1 – program1 (plagiát študentský)	7.38%
program1 – program1 (plagiát pokročilý)	0.17%
program2 – program2 (plagiát amatérsky)	1.26%
program2 – program2 (plagiát študentský)	1.34%
program2 – program2 (plagiát pokročilý)	0.63%
program2 – program2 (plagiát „guru“)	0.63%

Celkový dojem: Čo sa týka myšlienky riešenia problému, nedá sa povedať, že je zlá ale

ani najlepšia. Používateľské rozhranie je vporiadku, až na zmienený problém so zadávaním adresára. Rýchlosť systému je akceptovateľná vzhľadom na objem porovnávaní.

3.1.2 JPlag

Jplag sa špecializuje na odhaľovanie plagiátorstva v zdrojových kódoch. Podporuje jazyky Java, C#, C, C++ a Scheme, ale dá sa použiť aj pri hľadaní podobnosti v normálnom texte. Program slúži na odhaľovanie plagiátorstva medzi študentami, ale taktiež aj na nelegálnom kopírovaní softvéru. Výrobca píše, že JPlag bol už viackrát použitý v majetkových súdnych prípadoch, v ktorých bol expertmi použitý ako dôkazový materiál a svedectvo.

Implementácia: Program je naprogramovaný v jazyku Java, má klient-server architektúru.

Algoritmus: JPlag transformuje zdrojový kód programu do postupností tokenov. Každý token sa dá chápať ako jeden znak. Pre jazyk Java zverejnili tvorcovia tohto programu celkovo 39 rôznych tokenov. Po transformácii na tokeny ich následne porovnáva a hľadá zhody pomocou algoritmu GST. Teda hľadá podobné reťazce zložené z týchto tokenov.

GST porovnávanie prebieha po pároch. Pri každom porovnávaní sa snaží JPlag pokryť jeden prúd tokenov podreťazcom druhého čo najlepšie ako sa dá. Percento prúdu tokenov ktoré sa dá pokryť druhým je hodnota podobnosti.

Predpríprava, teda premena programu na reťazce tokenov je jediný proces JPlag-u, ktorý je závislý od programového jazyka. Implementácie pre jazyky Java a Scheme obsahujú úplný parser, zatiaľ čo modul pre jazyk C obsahuje iba skener.

Podmienky použitia: Pre použitie Jplagu musí byť používateľ zaregistrovaný. Registrácia je na základe formulára s overením identity na základe webového sídla školy, ktoré ukazuje email žiadateľa, študenti k Jplagu nemajú prístup pre prípadné zneužitie.

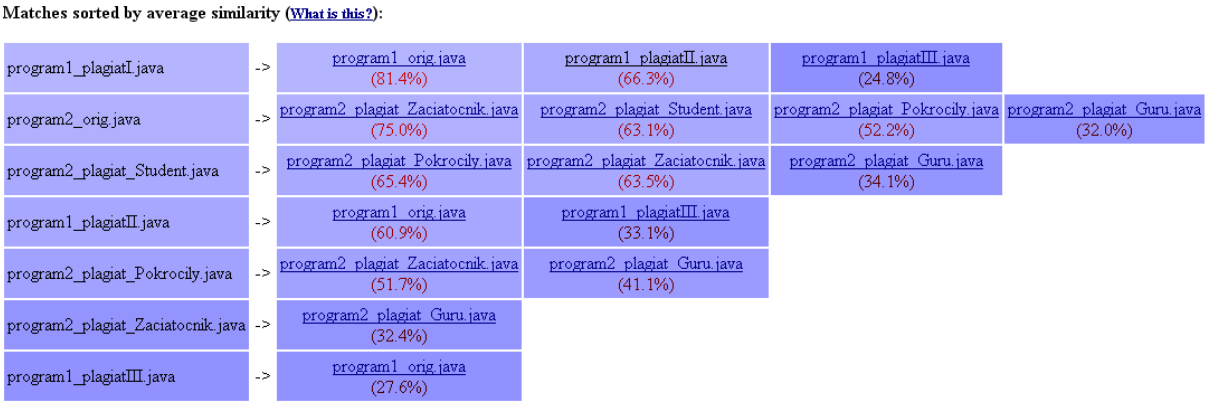
Použitie: Program má intuitívne grafické rozhranie a ponúka rôzne nastavenia. Funguje ako webová služba, preto je jednoducho dostupný pre užívateľa z ľubovolného miesta. Užívateľ si vyberie ktoré súbory sa majú kontrolovať a tie sa následne prenesú na server, kde sa okontrolujú. Testovanie je rýchle, rádovo niekoľko minút na 100 programov s niekoľko sto riadkovým kódom.

Program hľadá podobnosti iba vo zvolených programoch, nekontroluje programy na Internete. Výsledok hľadania sa užívateľom prehľadne zobrazí vo forme html kódu. Užívateľ si taktiež môže nastaviť zložitosť prehľadávania, miesto kam sa majú výsledky uložiť, alebo napríklad aj akú podobnosť má brať JPlag ako podozrivú.

Vizualizácia: Jplag vizualizuje podobné časti kódu ich jednoduchým farebným zvýraznením. Na obrázku 3.1 sú znázornené výsledky nášho testovania. Na jednotlivé súbory je možné kliknúť, čím sa zobrazia dvojice zdrojových kódov súborov vedľa seba so zvýraznenou podobnosťou. Takto sa dá jednoducho zistiť ktoré časti súborov sú si vzájomne podobné. JPlag taktiež zobrazuje počet zhodujúcich sa tokenov v jednotlivých segmentoch, podľa čoho vyrátava celkové percento zhody.

Klady:

- intuitívne rozhranie



Obr. 3.1: Výsledky testovania v programe Jplag na vzorke programov v jazyku Java

- škálovateľnosť nastavení
- klient-server architektúra
- veľmi dobé výsledky pre Java súbory
- prehľadná vizualizácia výsledkov
- archivácia predošlých testovaní
- pri zobrazení podľa priemernej podobnosti sa nevyskytli falošné plagiáty

Zápory:

- zložitejšie získavanie softvéru
- nutnosť byť učiteľom/zamestnancom v oblasti školstva
- slabšia úspešnosť pri súboroch v jazyku C
- podpora iba pre jazyky Java, C#, C, C++ a Scheme

Výsledky testovania:

JPlag dosahoval veľmi dobré výsledky pri odhaľovaní plagiátov v jazyku Java, no horšie výsledky v jazyku C. To môže byť spôsobené tým, že pre Javu je implementovaný spomínaný úplný parser, zatiaľ čo pre C iba skener.

Jednoduchšie plagiáty v jazyku Java odhalil na 50-81% podobnosť, ale aj pri prepracovanejších plagiátoch dosahovala podobnosť približne aspoň 30%, čo je dobrá úspešnosť. V jazyku C boli odhalené iba jednoduchšie plagiáty, ostatné ostali nedetekované. To je ale možné pripísať aj kratšiemu zdrojovému kódu 40-60 riadkov.

Dobrým výsledkom taktiež je, že pri zobrazení výsledkov podľa priemernej podobnosti (average similarity) neboli nájdené falošné plagiáty, teda vyššie percento podobnosti pri nesúvisiacich programoch. Táto podobnosť pri algoritme maximálnej podobnosti (maximum similarity) dosahovala v niektorých prípadoch skoro až 50%, čo je veľmi nežiadúce. Na druhej strane je možné, že táto podobnosť vznikla z dôvodu použitia rovnakej metodiky pri tvorbe plagiátov, alebo podobného charakteru programov, keďže boli písané rovnakým autorom.

Výsledky testovania sú v tabuľke 3.3.

Tabuľka 3.3: Výsledky JPlag

C

Originál – plagiát	Podobnosť
program1 – program1 (plagiát amatérsky)	77%/68%
program1 – program1 (plagiát študentský)	26%/18%
program1 – program1 (plagiát pokročilý)	nedetekované
program1 – program1 (plagiát „guru“)	nedetekované
program2 – program2 (plagiát amatérsky)	54%/46%
program2 – program2 (plagiát študentský)	nedetekované
program2 – program2 (plagiát pokročilý)	nedetekované
program2 – program2 (plagiát „guru“)	nedetekované

Java

Originál – plagiát	Podobnosť
program1 – program1 (plagiát amatérsky)	77%/86%
program1 – program1 (plagiát študentský)	64%/57%
program1 – program1 (plagiát pokročilý)	45%/25%
program1 – program1 (plagiát „guru“) 41%/20%	
program2 – program2 (plagiát amatérsky)	75%/74%
program2 – program2 (plagiát študentský)	62%/63%
program2 – program2 (plagiát pokročilý)	49%/55%
program2 – program2 (plagiát „guru“)	35%/29%

Celkový dojem: veľmi užitočný nástroj, ktorý dosahuje prekvapivé výsledky pre zdrojové kódy v jazyku Java, no horšie pre jazyk C. Nevýhodami sú obmedzenia iba na vybrané jazyky a dostupnosť iba pre pedagógov, ktorým je určení. Oproti týmto nevýhodám ponúka program viaceré výhody, vďaka čomu je veľmi užitočný pri odhaľovaní plagiátov a šetrení času.

3.1.3 SIM

Implementácia: SIM je open-source textová aplikácia implementovaná v jazyku C, k dispozícii pre MS-DOS.

Algoritmus: SIM využíva tokenizáciu zdrojového kódu, pričom v tokenizovaných kódoch hľadá najdlhší spoločný podreťazec [9].

Podmienky použitia: Jedná sa o voľne stiahnuteľnú aplikáciu pod licenciou GNU GPL, nie je nutná registrácia.

Použitie: Textové rozhranie poskytuje porovnanie množiny zdrojových kódov, percentuálna podobnosť je určovaná relatívne – súbor 1 obsahuje X% obsahu súboru 2, opačne vychádza iná hodnota. Pre porovnávanie sú k dispozícii rôzne EXE súbory, pre každý jazyk jeden.

Vizualizácia: Systém zobrazí percentuálnu podobnosť súborov, vizualizáciu zhodných častí podporuje, ale absentujú v nej prvky zvýraznenia.

Klady:

- poskytuje relevantnú funkcionalitu
- výhoda príkazového riadku - GUI "nezavadzia"
- výsledok je čistý text, výsledky sa dajú relatívne ľahko využiť ďalej
- podpora relatívne slušnej škály jazykov a dokonca aj paradigiem - C, Pascal, Lisp, Java, ...
- implementovaný v C, veľmi rýchly
- podporuje aj porovnávanie čistého textu

Zápory:

- chaoticky vyzerajúca prezentácia dát pre bežného používateľa
- slabá vizualizácia, súvisiaca s absenciou GUI
- pre každý jazyk má systém jeden EXE súbor, čo predstavuje otáznu koncepciu z hľadiska rozšíriteľnosti
- porovnávanie komentárov si vyžaduje spúšťať porovnanie súborov ako textu

Výsledky našich testov: Výsledky sú v tabuľke 3.4.

Tabuľka 3.4: Výsledky SIM

C

Originál – plagiát	Podobnost'
program1 – program1 (plagiát amatérsky)	95%/91%
program1 – program1 (plagiát študentský)	nedetekované
program1 – program1 (plagiát pokročilý)	nedetekované
program1 – program1 (plagiát „guru“)	nedetekované
program2 – program2 (plagiát amatérsky)	61%/55%
program2 – program2 (plagiát študentský)	nedetekované
program2 – program2 (plagiát pokročilý)	nedetekované
program2 – program2 (plagiát „guru“)	nedetekované

Java

Originál – plagiát	Podobnost'
program1 – program1 (plagiát amatérsky)	88%/76%
program1 – program1 (plagiát študentský)	5%/5%
program1 – program1 (plagiát pokročilý)	nedetekované
program1 – program1 (plagiát „guru“)	nedetekované
program2 – program2 (plagiát amatérsky)	80%/80%
program2 – program2 (plagiát študentský)	nedetekované
program2 – program2 (plagiát pokročilý)	nedetekované
program2 – program2 (plagiát „guru“)	nedetekované

Delphi

Originál – plagiát	Podobnost'
program1 – program1 (plagiát amatérsky)	99%/98%
program1 – program1 (plagiát študentský)	nedetekované
program1 – program1 (plagiát pokročilý)	nedetekované

program1 – program1 (plagiát „guru“) nedetekované

Celkový dojem: Solídny nástroj, ktorý spĺňa základné požiadavky, pričom spracovanie vzorky prebieha veľmi rýchlo. Jeho nedostatkom je predovšetkým nedostatočná schopnosť zachytiť zložitejšie plagiátorské prístupy a otázna je taktiež jeho využiteľnosť pre ďalšie jazyky (vhodnejšie by bolo využitie externých definícií štruktúr jazyka). Výhodou je podpora viacerých paradigiem programovania, čo ide ruka v ruke so širším dosahom aplikácie (netestujú sa len jazyky s inou syntaxou, ale aj inou koncepciou).

3.1.4 MOSS

Implementácia: MOSS je klient-server aplikacia, klient je implementovaný ako Perl skript.

Algoritmus: Keďže porovnávanie prebieha na serveri a autori algoritmus nezverejnili kvôli potencionálnemu zneužitiu študentmi, nevieme určiť ako samotné porovnávanie prebieha. Autori sa však odkazujú na článok, k dispozícii na [10], kde sú spomenuté niektoré algoritmy na detekciu plagiarizmu.

Podmienky použitia: Pre použitie aplikácie je nutné sa registrovať, registrovať sa môže ktokoľvek. Po poslaní mailu na základe inštrukcií príde automatická odpoveď, v ktorej sa nachádza samotný klient.

Použitie: Pre používateľa OS typu Unix/Linux, ktorý má rád jednoduché textové aplikácie veľmi príjemné, keďže perl je štandardnou súčasťou týchto OS. Je potrebné len spustiť dodaný skript s parametrom pre súbory, ktoré sa majú otestovať.

Po uploadovaní súborov na server sa na serveri vykoná kontrola a následne klient dostane http link s prezentáciou výsledkov. Výsledky sú na serveri uložené ďalších 14 dní, potom sú automaticky zmazané. Zhody sú zobrazené prehľadne podľa súborov, usporiadané zostupne podľa počtu zhodných riadkov.

Dá sa použiť na detekciu plagiarizmu v širokej palete programovacích jazykov. Pre slovenské texty je však trochu obmedzený, berie do úvahy totiž iba ASCII kódovanie.

Vizualizácia: Okrem percentuálnej zhody sa dajú zobraziť súbory vedľa seba, kde je farbami jasne označené, ktoré časti sa zhodujú a ktoré sú odlišné (ak je prehodené poradie segmentov, farby zhodných segmentov sú rovnaké).

Klady:

- použitie je vcelku jednoduché a prehľadné, skript zvládne použiť naozaj každý, kto má k dispozícii stroj s vyššie spomenutým OS
- možnosť porovnávať celé adresáre ako programy, teda jeden program na jeden adresár (oceníme v prípade rozsiahlejších projektov)
- možnosť na vloženie tzv. basefile = obsahuje kód, o ktorom vyučujúci predpokladá, že ho budú zdrojové súbory obsahovať (riešenie špecifickej úlohy)

- pri použití vhodných parametrov je možné určiť počet prípadov, kedy výskyt zhodného segmentu kódu ešte považujeme za plagiarizmus a kedy za štandardnú súčasť všetkých kódov, čím sa dá vyhnúť použitiu basefile a súčasne zlepšiť výpovednú hodnotu výsledkov
- dobrá prezentácia výsledkov

Zápory:

- "nepodporaïného ako ASCII kódovania, pre použitie na slovenské texty je to značne obmedzujúce. nevedomosť o použitých algoritmoch na detekciu
- slabá detekcia prípadov, kedy bol kód zmenený trochu viac ako len premenovanie premenných, čo dokazujú aj výsledky testov
- pre bežného používateľa môže byť problém v použití klienta, keďže väčšina používateľov dokáže použiť len programy s GUI

Výsledky testovania: Pri programových kódoch vie odhaliť čisté kopírovanie, prípadne jednoduché zmeny, pri sofistikovanejšom plagiarizme však výsledky nie sú dobré. Napríklad pri doplňovaní podmienok o zbytočné klauzule, prípadne prepísaní typov cyklov tieto nevie detekovať. Toto je možné vidieť aj v prezentovaných testovacích výsledkoch. Pri použití na textové súbory odhalí aj tie, kde je prehodené poradie slov vo vete a niektoré vety sú navyše, ale ich počet nesmie prevyšovať 1-2 v danom segmente. Celkové výsledky sú v tabuľke 3.5.

Časová náročnosť:
programy JAVA, C, Delphi:
časová náročnosť: 2.525s, z toho 2 sekundy upload súborov na server
slovenské texty:
časová náročnosť: 8.788s, z toho 6 sekúnd upload súborov na server

Tabuľka 3.5: Výsledky MOSS

C

Originál – plagiát	Podobnosť
program1 – program1 (plagiát amatérsky)	82%/78%
program1 – program1 (plagiát študentský)	nedetekované
program1 – program1 (plagiát pokročilý)	nedetekované
program1 – program1 (plagiát „guru“)	nedetekované
program2 – program2 (plagiát amatérsky)	57%/53%

program2 – program2 (plagiát študentský)	nedetekované
program2 – program2 (plagiát pokročilý)	nedetekované
program2 – program2 (plagiát „guru“)	nedetekované

Java

Originál – plagiát	Podobnosť
program1 – program1 (plagiát amatérsky)	75%/87%
program1 – program1 (plagiát študentský)	57%/59%
program1 – program1 (plagiát pokročilý)	9%/5%
program2 – program2 (plagiát amatérsky)	80%/79%
program2 – program2 (plagiát študentský)	33%/31%
program2 – program2 (plagiát pokročilý)	14%/14%
program2 – program2 (plagiát „guru“)	4%/2%

Delphi

Originál – plagiát	Podobnosť
program1 – program1 (plagiát amatérsky)	93%/92%
program1 – program1 (plagiát študentský)	56%/53%
program1 – program1 (plagiát pokročilý)	32%/28%
program1 – program1 (plagiát „guru“)	27%/22%

Celkový dojem: MOSS je veľmi dobrý nástroj pre testovanie plagiátov zdrojových kódov. Najlepšie sa uplatní v prípade jednoduchého skopírovania kódu, prípadne ľahkých modifikácií. Má výbornú podporu pre vysoký počet jazykov. Nevýhodou je nižšia rozlišovacia schopnosť sofistikovanejších plagiátov a taktiež pre Windows používateľa aj implementačné prostredie klienta.

3.1.5 YAP

YAP (Yet Another Plague) je systém pre detekciu plagiarizmu v programových kódoch (stránka autora naznačuje, že by do istej miery mal fungovať aj s anglickými textami). Dostupné sú verzie YAP1, YAP2 a YAP3.

Implementácia: Implementácia systému YAP je rozdelená do dvoch častí – tokenizácia a samotné vyhľadávanie plagiátov. Časť tokenizácie je spoločná pre všetky 3 verzie a je implementovaná ako shell skript.

YAP1 je implementovaný ako Bourne-shell skript využívajúci rôzne štandardné aplikácie, primárne sdiff (slúži na porovnávanie textových súborov). Zvyšné dve verzie sú vytvorené v jazyku Perl. Pre dosiahnutie vyššej rýchlosti porovnávania sú algoritmy verzií YAP2 a YAP3 implementované v jazyku C.

Algoritmus: YAP2 využíva Heckelov algoritmus, pri YAP3 je to Running Karp-Rabin, Greedy String Tiling algoritmus (Michael J. Wise: String Similarity via Greedy String Tiling and Running Kapr-Rabin Matching). Verzia YAP3 vďaka použitiu RKR-GST dokáže zdetegovať plagiáty aj so zmeneným poradím podretazcov.

Podmienky použitia: YAP 1,2,3 je voľne stiahnuteľný zo stránky autora. Komerčné použitie je podmienené získaním súhlasu autora.

Použitie: Detekcia plagiarizmu prebieha v dvoch krokoch. Prvým je tokenizácia textov. Táto je relevantná iba pre programové kódy, pretože odstraňuje všetky symboly, ktoré nie sú súčasťou slovníka pre daný programovací jazyk. Tokenizácia sa dá opísať nasledovne [11]:

- komentáre a textové konštanty sú odstránené
- všetky písmená sú dekapitalizované
- rôzne synonymá sú nahradené ich bežnou formou
- ak je to možné, funkcie/procedúry sú rozvité v poradí volania
- všetky symboly, ktoré nie sú súčasťou slovníka pre daný programovací jazyk,sú odstránené

Tokenizácia má teda za cieľ previesť zdrojové kódy na čo najvšeobecnejšiu formu (v ideálnom prípade takú, že neautorský program bude zhodný s jeho originálom, aj napriek drobným zmenám v neautorskej verzií). Je zaujímavé všimnúť si bod o nahrádzaní funkcií ich implementáciou. Nahrádzanie je vykonané v poradí, v akom sa funkcie navzájom volajú, čo umožňuje pracovať aj volaniami funkcií z iných funkcií.

Dôležitá je poznámka, že nahradenie prebehne, iba ak je to možné (príkladom, kde to možné nie je, sú rekurzívne funkcie – naivný prístup nahrádzania by viedol v zacykleniu programu, ktorý by sa snažil nahradiť volanie funkcie implementáciou, ktorá by túto funkciu volala). Vďaka odstráneniu všetkých symbolov, ktoré nie sú súčasťou jazyka, systém efektívne oddeľuje dáta a samotný kód. Ignorovanie dát je logické. Dá sa totiž očakávať, že rovnaké zadanie bude viesť k riešeniam s obsahujúcim rovnaké dáta, aj keď nepôjde o plagiarizmus.

Fáza porovnávania vychádza zo súborov so symbolmi, ktoré boli vytvorené v predchádzajúcej fáze. Porovnávané sú potom dvojice týchto súborov. Systémy YAP1, YAP2 a YAP3 sa líšia iba v tejto fáze – tokenizácia je spoločná pre všetky tri verzie programu.

Vizualizácia: YAP je implementovaný ako konzolová aplikácia a jeho jediným výstupom je textový súbor s výsledkami.

Klady:

- jednoduchosť použitia
- tokenizácia
- nahradenie funkcií ich implementáciou
- použité algoritmy

Zápory:

- obmedzenie na Linux/Unix
- malá prenositeľnosť spôsobená (na dnešnú dobu) exotickou implementáciou
- pre bežného používateľa môže byť problém v použití klienta, keďže väčšina používateľov dokáže použiť len programy s GUI

Výsledky testovania: Systém YAP sa nám napriek vynaloženému úsiliu nepodarilo správne nakonfigurovať a spustiť, a preto sme ho nemohli otestovať.

Celkový dojem: Strohé, nekompromisné a rýchle riešenie hľadania plagiátov v programoch, jednoducho rozšíriteľné o schopnosť rozlišovať ďalšie programovacie jazyky vďaka oddelenej tokenizácii vstupných súborov.

3.2 Podobnosť slovenských textov

3.2.1 PlaDeS

Vznikol v rámci tímového projektu študentov Tomáša Hlatkého a Michala Kompana pod vedením Daniely Chudej.

Implementácia: Jedná sa o aplikáciu určenú pre operačný systém Windows. Program je nutné inštalovať a pre beh je potrebný .NET 3.5 framework.

Algoritmus: Aplikácia v sebe implementuje 4 druhy analýzy textu.

1. 3-gramy
2. najdlhšia spoločná podpostupnosť [7]
3. inverzná frekvencia slov v dokumente [12]
4. frekvencia výskytu slov [13]

Podmienky použitia: Aplikácia je voľne prístupná na webe.

Použitie: Programu je možné nastaviť hraničnú podobnosť ako aj možnosť vypnúť diakritiku pri porovnávaní. Podľa slov autorov to nemá výrazný vplyv na výsledky porovnávania. Pri porovnávaní slovenských textov je možné z textov odstrániť stop slová a previesť lematizáciu, čo urýchľuje samotnú analýzu.

Program podporuje súbory vo formátoch pdf a doc, respektíve zip archívy. Je možné prehľadávať adresáre, v ktorých sa budú porovnávať všetky súbory, ako aj vyberať požadované súbory.

Program má jednoduché používateľské prostredie, ktoré podľa mňa bohato postačuje na prácu s programom. Ovládacie prvky sú logicky oddelené a nie je teda problém program ovládať. Program vizuálne informuje používateľa o vykonávanej činnosti čo osobne považujeme za veľmi kladný krok. Pri spustení programu sa na obrazovke objaví splash screen, počas ktorého sa zrejme nahrávajú z disku konfiguračné súbory.

Pri skúške programu nastala chyba pri vyberaní súborov na analýzu. Program spadol ak nebol spustený s administrátorskými právami. Pri ďalšom spustení program fungoval už správne.

Vizualizácia:

Klady:

- Príjemné používateľské prostredie
- Možnosť nastavenia parametrov a metódy detekcie
- Možnosť vybrať na otestovanie konkrétne súbory alebo celý adresár
- Rýchlosť, spracovanie a analýza 60 doc súborov trvala 12 minút

Zápory:

- Nepodporuje obyčajné textové súbory

Výsledky testovania: Výsledky sú v tabuľke 3.6.

Tabuľka 3.6: Výsledky Plades

Metóda : 3gramy

Originál – plagiát	Podobnosť
v122_MSI2008_michalek_original - v123_MSI2008_michalek_plagiatcopy	96%
v122_MSI2008_michalek_original - v124_MSI2008_michalek_plagiatmodify	75%
v122_MSI2008_michalek_original - v125_MSI2008_michalek_plagiatresers	77%
v095_MSI2008_kozisek_original - v096_MSI2008_kozisek_plagiatcopy	98 %

v095_MSI2008_kozisek_original - v097_MSI2008_kozisek_plagiatmodify	65%
v095_MSI2008_kozisek_original - v098_MSI2008_kozisek_plagiatresers	83%

Metóda : LCS

Originál – plagiát	Podobnost'
v122_MSI2008_michalek_original - v123_MSI2008_michalek_plagiatcopy	10%
v122_MSI2008_michalek_original - v124_MSI2008_michalek_plagiatmodify	8%
v122_MSI2008_michalek_original - v125_MSI2008_michalek_plagiatresers	9%
v095_MSI2008_kozisek_original - v096_MSI2008_kozisek_plagiatcopy	11%
v095_MSI2008_kozisek_original - v097_MSI2008_kozisek_plagiatmodify	8%
v095_MSI2008_kozisek_original - v098_MSI2008_kozisek_plagiatresers	11%

Metóda : IDF

Originál – plagiát	Podobnost'
v122_MSI2008_michalek_original - v123_MSI2008_michalek_plagiatcopy	100%
v122_MSI2008_michalek_original - v124_MSI2008_michalek_plagiatmodify	0%
v122_MSI2008_michalek_original - v125_MSI2008_michalek_plagiatresers	0%
v095_MSI2008_kozisek_original - v096_MSI2008_kozisek_plagiatcopy	0%
v095_MSI2008_kozisek_original - v097_MSI2008_kozisek_plagiatmodify	0%
v095_MSI2008_kozisek_original - v098_MSI2008_kozisek_plagiatresers	0%

Metóda : TF

Originál – plagiát	Podobnost'
v122_MSI2008_michalek_original - v123_MSI2008_michalek_plagiatcopy	100%
v122_MSI2008_michalek_original - v124_MSI2008_michalek_plagiatmodify	98%
v122_MSI2008_michalek_original - v125_MSI2008_michalek_plagiatresers	94%
v095_MSI2008_kozisek_original - v096_MSI2008_kozisek_plagiatcopy	100%
v095_MSI2008_kozisek_original - v097_MSI2008_kozisek_plagiatmodify	97%

Celkový dojem: Program PlaDeS sa radí medzi jednoduchšie programy, no aj napriek tomu ponúka veľmi dobré výsledky pri hľadaní podobností v súboroch. Má v sebe implementované pokročilé algoritmy na hľadanie zhôd, pričom každý jeden z nich má výhody ako aj nevýhody. Program sa jednoducho obsluhuje a práca s ním je príjemná.

3.2.2 Sherlock

Sherlock je program, ktorý hľadá podobnosti v textových dokumentoch.

Implementácia: Open source, standalone, textová aplikácia, k dispozícii aj sherlock prepísaný do Javy [14].

Algoritmus: Na kontrolu využíva hashovaciu funkciu. Tá priradí bloku slov číslený odtlačok.

Podmienky použitia: Sherlock je voľne prístupný na stiahnutie

Použitie:

- ovládanie pomocou prepínačov
- hraničná podobnosť
- počet slov na vytvorenie signatúry
- skladá sa z jedného súboru
- výstup do konzoly alebo súboru
- je aj súčasťou BOSSu

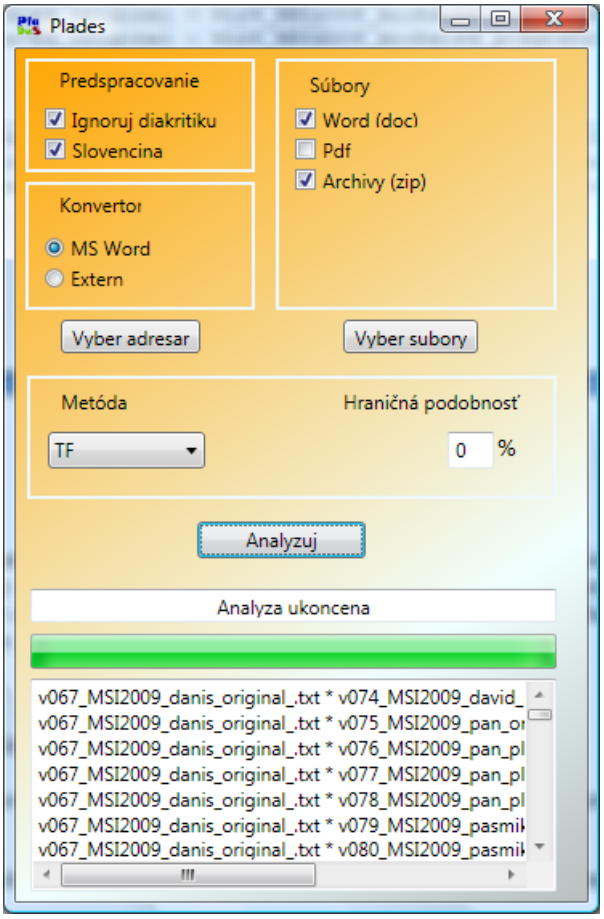
Klady:

- veľmi rýchly, 69 textových súborov, každý $\tilde{2}$0kB za zlomok sekundy

Zápory:

- nefunkčné podaktoré prepínače, ak je potreba otestovať 124 súborov, nutnosť všetky vypísať
- vypisuje len percentuálnu zhodu medzi súbormi, žiadne bližšie informácie

Výsledky testovania: Výsledky sú v tabuľke 3.7.



Obr. 3.2: Rozhranie programu Plades

Tabuľka 3.7: Výsledky Sherlock

C

Originál – plagiát	Podobnosť
program1 – program1 (plagiát amatérsky)	22%
program1 – program1 (plagiát študentský)	nedetekované
program1 – program1 (plagiát pokročilý)	nedetekované
program1 – program1 (plagiát „guru“)	nedetekované
program2 – program2 (plagiát amatérsky)	8%
program2 – program2 (plagiát študentský)	nedetekované
program2 – program2 (plagiát pokročilý)	nedetekované
program2 – program2 (plagiát „guru“)	nedetekované

Java

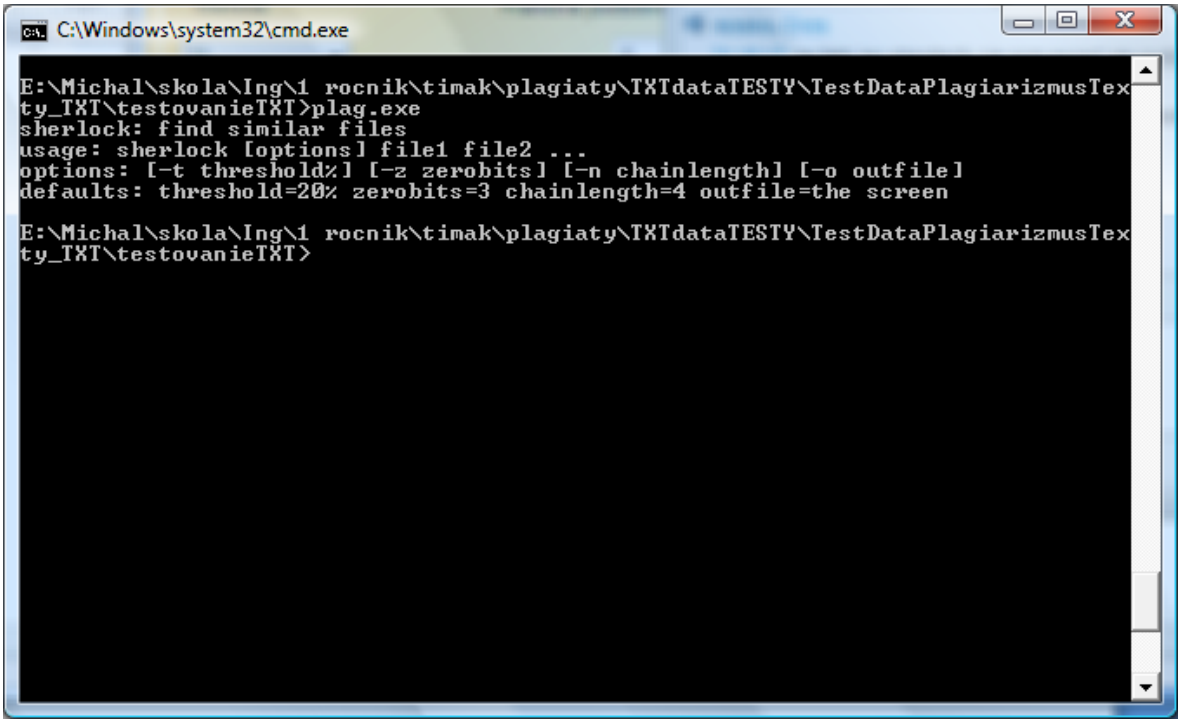
Originál – plagiát	Podobnosť
program1 – program1 (plagiát amatérsky)	55%
program1 – program1 (plagiát študentský)	8%
program1 – program1 (plagiát pokročilý)	5%
program2 – program2 (plagiát amatérsky)	8%
program2 – program2 (plagiát študentský)	4%
program2 – program2 (plagiát pokročilý)	4%
program2 – program2 (plagiát „guru“)	nedetekované

Delphi

Originál – plagiát	Podobnosť
program1 – program1 (plagiát amatérsky)	16%
program1 – program1 (plagiát študentský)	12%
program1 – program1 (plagiát pokročilý)	8%
program1 – program1 (plagiát „guru“)	8%

Texty

Originál – plagiát	Podobnosť
v122_MSI2008_michalek_original - v123_MSI2008_michalek_plagiatcopy	100%
v122_MSI2008_michalek_original - v124_MSI2008_michalek_plagiatmodify	85%
v122_MSI2008_michalek_original - v125_MSI2008_michalek_plagiatresers	67%
v095_MSI2008_kozisek_original - v096_MSI2008_kozisek_plagiatcopy	99%
v095_MSI2008_kozisek_original - v097_MSI2008_kozisek_plagiatmodify	78%
v095_MSI2008_kozisek_original - v098_MSI2008_kozisek_plagiatresers	78%



Obr. 3.3: Rozhranie programu Sherlock

Celkový dojem: Program Sherlock je jednoduchá konzolová aplikácia, ktorá napriek svojej jednoduchosti podáva veľmi dobré výsledky pri porovnávaní textových súborov. Pôvodná verzia sa distribuuje prostredníctvom zdrojového kódu, ktorý je potrebné skompilovať, čo môže odradiť niektorých používateľov. Nakoľko je program určený na porovnávanie textov, nie je vhodné ho používať na porovnávanie zdrojových kódov, nakoľko dáva nepresné výsledky.

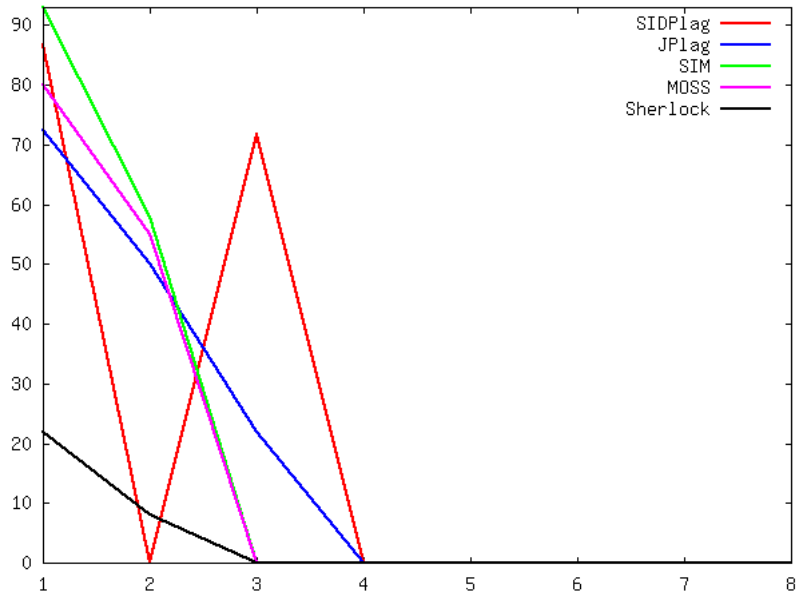
3.3 Zhrnutie analýzy existujúcich nástrojov

3.3.1 Nástroje na porovnávanie zdrojových kódov

Pri porovnávaní jednotlivých nástrojov sme sa sústredili predovšetkým na jazyky C a Java, pričom pri niektorých nástrojoch sme analyzovali aj ich úspešnosť pre Delphi (nakoľko nie všetky analyzované nástroje podporujú jazyk Pascal a jeho „deriváty").

Metodológia pre tvorbu testovacej vzorky bola rovnaká pre všetky jazyky, avšak programy, ktoré sme využívali ako testovacie dáta pre jazyk C boli o čosi kratšie ako použité programy v jazyku Java. Ukazuje sa, že aj tento rozdiel má pomerne výrazný vplyv na úspešnosť detekcie u všetkých nástrojov.

Pracovali sme s množinou zdrojových kódov upravených podľa našej metodológie, ktoré sme porovnávali s ich originálnymi verziami. V grafe na obrázku 3.4 predstavujú na osi x body 1-2 náročnosť začiatočníka, 3-4 náročnosť študenta, 5-6 náročnosť pokročilého a 7-8 náročnosť „guru" pre zdrojové kódy v jazyku C. S rastúcou hodnotou na osi x rastie teda aj náročnosť testovacieho vstupu, prirodzené je preto očakávanie, že funkcie podobnosti budú mať u jednotlivých nástrojov nerastúce tendencie.

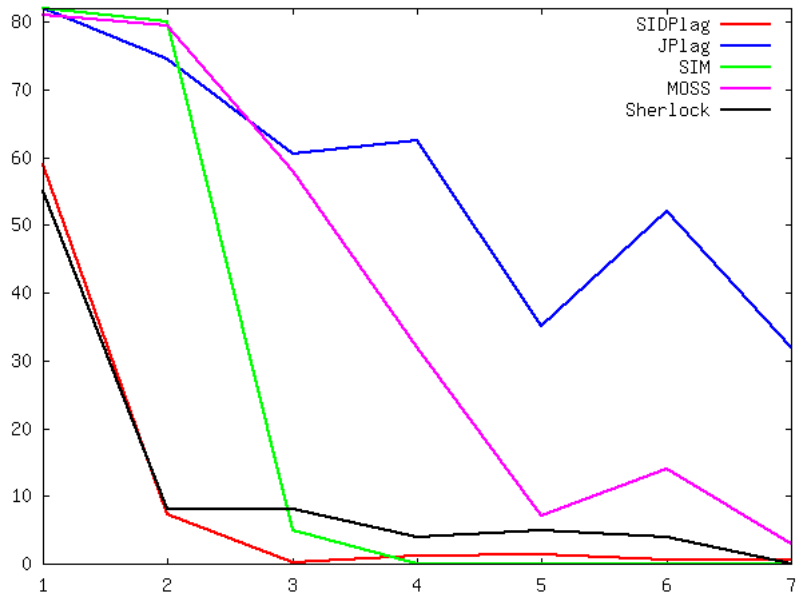


Obr. 3.4: Porovnanie úspešnosti nástrojov na zdrojové kódy v jazyku C

Graf ukazuje, že úspešnosť jednotlivých nástrojov prudko klesá medzi úrovňou začiatočníka a študenta. Je však nutné poznamenať, že tento jav môže súvisieť práve so spomínanou dĺžkou zdrojových kódov, ktorá bola pre testované súbory v jazyku C nižšia. Ako najúspešnejší nástroj možno na základe tejto analýzy vnímať JPlag, relatívne dobré výsledky dosahuje ale aj MOSS. SIDPlag vykazuje v analýze veľmi otázny výsledok, najmä

úspešnosť pre tretí testovací vstup sa zdá byť málo dôveryhodná.

V jazyku Java sme pracovali s mierne dlhšími súbormi a to sa odrazilo aj na samotnom grafe, ktorý sa nachádza na obrázku 3.5. Rozdiel medzi nástrojom JPlag a ostatnými aplikáciami je už výraznejší, avšak celkom solídnu úspešnosť vykazuje stále aj MOSS. Graf však ilustruje zaujímavý jav, zdá sa totiž, že pri dlhších zdrojových kódoch sa robustnosť techník „maskovania" plagiarizmu klesá.



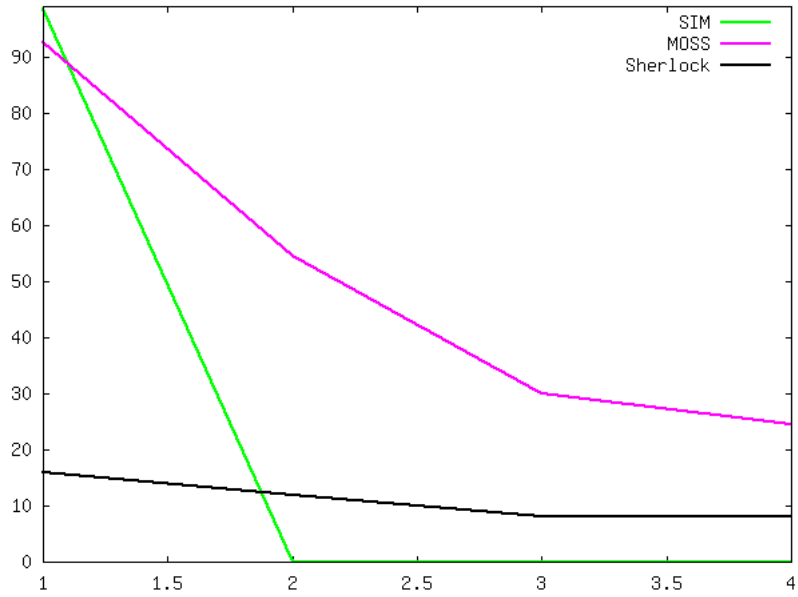
Obr. 3.5: Porovnanie úspešnosti nástrojov na zdrojové kódy v jazyku Java

Vzhľadom k tomu, že JPlag nepodporuje Deplhi, pre tento jazyk sa do popredia dostáva práve nástroj MOSS, ako je možné vidieť na obrázku 3.6.

Táto analýza má však aj ďalší rozmer. Medzi problémy, ktorým je potrebné pri diskutovanej problematike čeliť, je totiž aj tvorba relevantných testovacích dát, ktorá vôbec nie je triviálnou záležitosťou. S kolegami sme sa zhodli v názore, že manuálne upravovanie zdrojových kódov je časovo veľmi náročnou činnosťou (najmä pokiaľ sa vyžaduje dodržiavanie stanovenej metodológie). Je preto potrebné uvažovať aj o reálnych dátach, ktoré síce nepodliehajú navrhnutej metodológii, ale spĺňajú reálne parametre skúmaného problému a umožňujú sledovať úspešnosť jednotlivých prístupov.

3.3.2 Porovnanie nástrojov na slovenské texty

Pre porovnanie nástrojov na slovenské texty bola použitá vzorka desiatich nezávislých súborov, z ktorých boli vyrobené testovacie vzorky reprezentujúce plagiáty. Používame tri úrovne plagiátov slovenských textov ktoré sú popísané v nasledujúcej tabuľke.



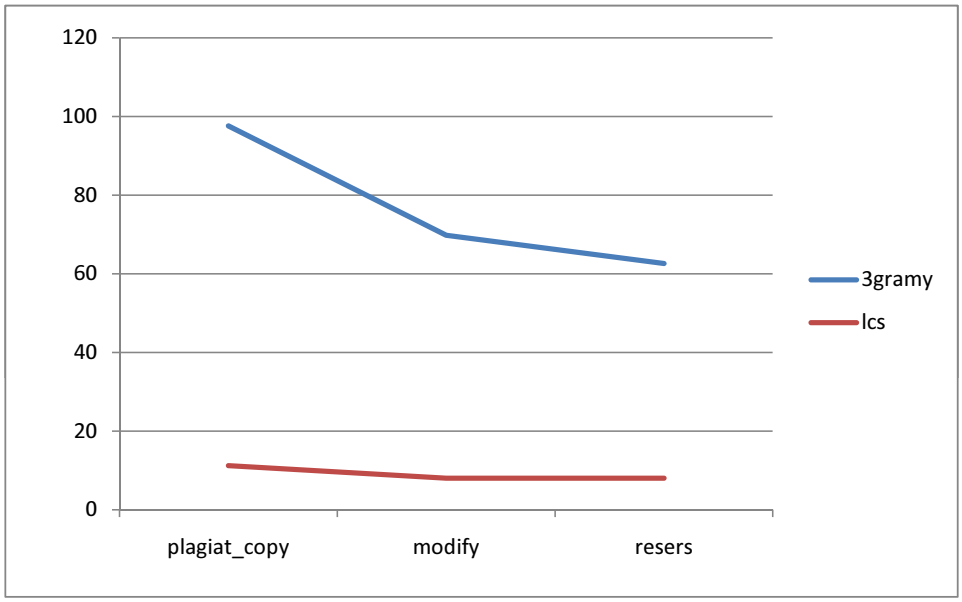
Obr. 3.6: Porovnanie úspešnosti nástrojov na zdrojové kódy v jazyku Delphi

Tabuľka 3.8: Typy plagiátov

plagiát	**popis**
plagiat_copy	kópia originálu, zmena názvu práce
modify	úprava formulácii, výmena kapitol, zmena formátovania, pridanie textu, obrázkov, vymazávanie častí
resers	plagiát poskladaný z viacerých zdrojov

Pomocou programu Plades a Sherlock boli porovnané originálne súbory s ich plagiátmi. Výsledná hodnota úspešnosti odhalenia plagiátov pre každú metódu je aritmetickým priemerom všetkých desiatich vzoriek.

Na nasledujúcich obrázkoch sú výsledky porovnávania súborov pomocou spomenutých programov. V programe Plades boli testované len metódy LCS a 3gramy pretože sa zistilo, že metódy frekvenčnej analýzy v tomto programe neboli korektne implementované.

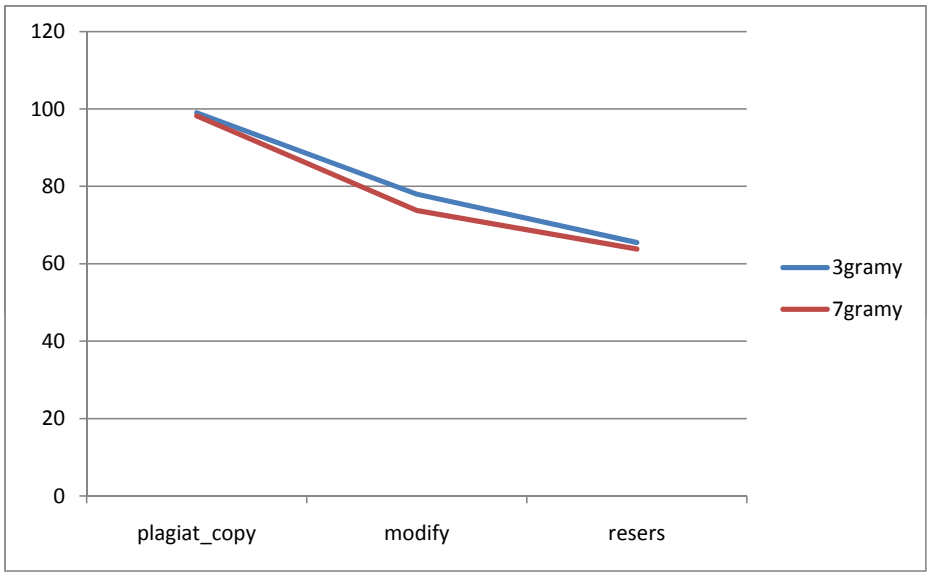


Obr. 3.7: Výsledok porovnania textov nástrojom Plades

Výsledky ukázali, že metóda N-gramov dáva rovnaké výsledky pri oboch programoch. Zmenou hodnoty n, v rozpätí od 3 do 7 sa v programe Sherlock výsledky zásadne nemenili. Pri zvyšovaní hodnoty n, krivka podobnosti klesá, avšak rozdiely medzi jednotlivými úrovňami plagiátov zostávajú zachované. Výsledky vytvorené pomocou metódy N-gramov sú akceptovateľné a rozdiel medzi jednotlivými úrovňami plagiátov je zrejmý. Táto metóda sa ukázala ako účinná v oboch programoch.

Metóda LCS v programe Plades nepriniesla požadované výsledky pri odhaľovaní plagiátov. Jej výsledky sú prevažne konštantné alebo len vo veľmi malom rozsahu. Preto nemožno spoľahlivo určiť hranicu kedy možno dokument považovať za plagiát.

Oba programy sú teda použiteľné k detekcii plagiarizmu pri použití metódy n-gramov a vykazujú veľmi podobné výsledky.



Obr. 3.8: Výsledok porovnania textov nástrojom Sherlock

Kapitola 4

Vizualizácia

4.1 Vizualizácia výsledkov

Jedným z problémov pri odhaľovaní plagiátorstva je grafické zobrazenie výsledkov, ktoré vrátia programy slúžiace na odhaľovanie plagiátorstva. Samotná vizualizácia týchto výsledkov je potrebná vtedy, keď je nutné vizuálne skontrolovať podobnosť dvoch alebo viacerých súborov, ktoré boli označené ako plagiáty. Vizualizácia umožní presné označenie podobných alebo rovnakých častí.

4.2 Kategorizácia metód

Metódy, ktoré sa používajú na vizualizáciu výsledkov porovnávania sa dajú rozdeliť podľa úrovne, na ktorej vizualizujú výsledky. Vo všeobecnosti tieto metódy zobrazujú podobnosť medzi:

1. **Súbormi**
   Tieto metódy dokážu povedať, ktoré konkrétne súbory sú si podobné, no nedokážu povedať, ktoré časti v týchto súboroch sú si podobné.
   Do tejto skupiny patria metódy:
   (a) Graf
   (b) Kruhový graf
   (c) Histogram
2. **Textami**
   Na rozdiel od metód, ktoré ukazujú zhodu len medzi súbormi, tieto metódy vizualizujú aj zhody v dokumentoch. Dokážu teda povedať, ktoré časti sú si podobné a graficky ich zobraziť.
   Do tejto skupiny patria metódy:
   (a) Arc diagram

(b) Farebné zvýraznenie

(c) Tag clouds

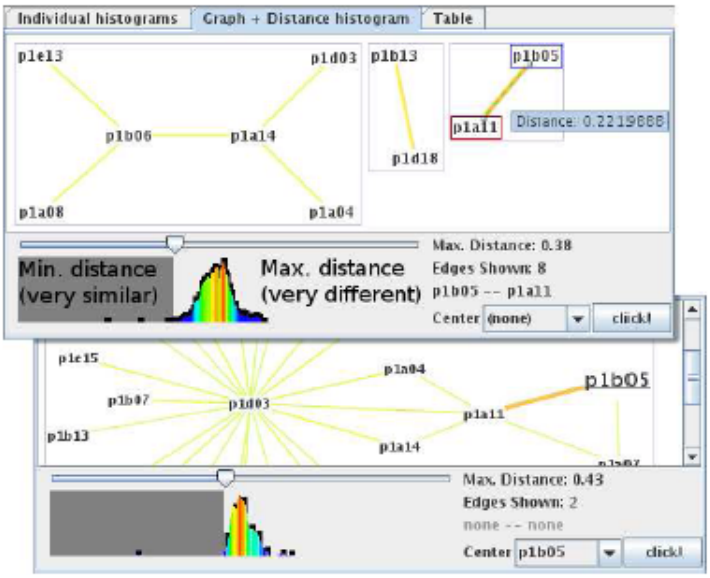
(d) Word spectrum diagram

4.2.1 Graf

Tento druh vytvára zo zadaných súborov graf. V tomto grafe sú umiestnené porovnávané súbory. Súbory, ktoré boli označené ako podobné, sú spojené hranou. Pomocou tejto hrany sa dá popísať podobnosť, ktoré 2 súbory majú. Konkrétne pomocou jej

- Farby
- Šírky
- Dĺžky

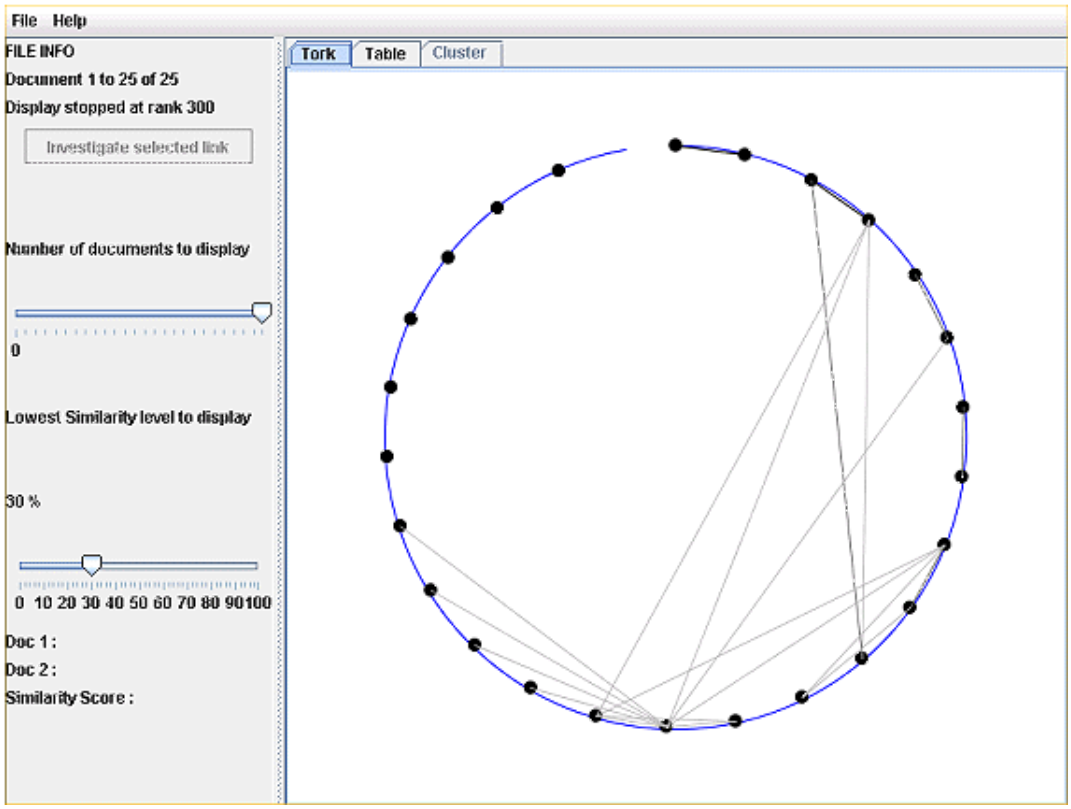
Napríklad, čím je hrana dlhšia, tým menej sú si 2 súbory podobné. Pomocou tejto metódy je možné súbory usporiadať aj do skupín. Je možné tento druh vizualizácie použiť na jeden konkrétny súbor, okolo ktorého sa následne vytvorí graf z ostatných súborov, poprípade vytvoriť podmnožiny zo súborov.



Obr. 4.1: Graf [1]

[1] Zdroj: Freire,M.2008.VisualizingprogramsimilarityintheAcplagiarismdetectionsystem. InProceedingsoftheWorkingConferenceonAdvancedVisualinterface(Napoli,Italy,May28-30, 2008).AVI'08.ACM,NewYork,NY,404-407.

Alternatívou k usporiadaniu súborov do grafu je ich usporiadanie do kruhu, ktoré má za následok lepší prehľad o podobnosti medzi súbormi. Tento spôsob je zobrazený na obrázku číslo 2.

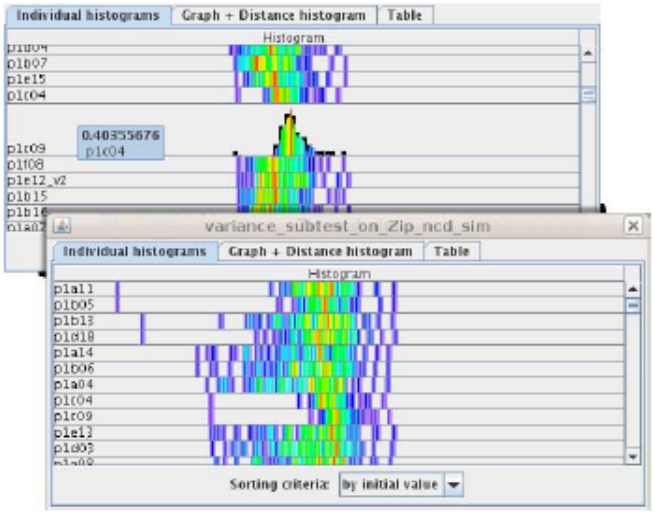


Obr. 4.2: Kruhový graf[2]

4.2.2 Histogram

Táto metóda vychádza z predchádzajúcej metódy, ktorá zobrazuje výsledok pomocou grafu. Na rozdiel od nej však nepoužíva priamo graf, ale ho zobrazuje pomocou histogramu. Jednotlivé čiary z grafu sú reprezentované stĺpcami histogramu. Pomocou ich dĺžky a farby je možné vizualizovať podobnosť medzi súbormi. Čím je stĺpec histogramu väčší, tým podobnejšie sú si súbory. Ak je podobnosť veľká, má stĺpec farbu ladenú do červena, naopak, pri nízkej zhode je farba zelená.

[2]Zdroj: `http://www.ascilite.org.au/conferences/perth04/procs/lancaster.html`



Obr. 4.3: Histogram[3]

4.2.3 Arc diagram

Táto metóda slúži na označovanie rovnakých podreťazcov v textoch. Text, ktorý sa analyzuje je rozdelený na podreťazce dĺžky n a zobrazený na jednom riadku. Podreťazce, ktoré sa v texte nachádzajú viackrát, sú spojené čiarou, ktorá má tvar poloblúka. Čím je táto čiara hrubšia, tým dlhšie podreťazce sú spojené. Čím je poloblúk vyšší, tým ďalej sa nájdené podreťazce nachádzajú. Výsledok tejto metódy sa podobá na dúhu, nakoľko sú jednotlivé podreťazce spojené práve poloblúkom.

Obr. 4.4: Arc diagram[4]

Tento spôsob je však nevhodný pre podreťazce, ktoré majú príliš krátku dĺžku. Pokiaľ by sa porovnávali podreťazce dĺžky 1 alebo 2, výsledný graf by bol neprehľadný z toho

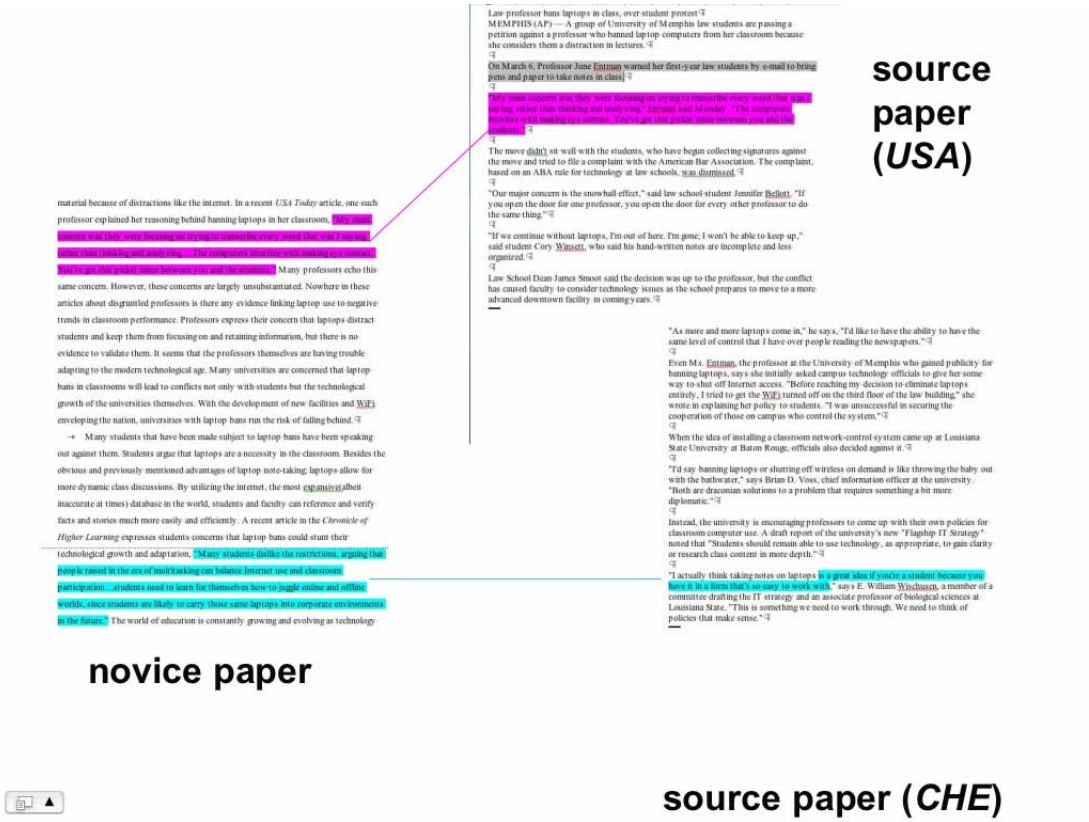
[3]Zdroj: Freire,M.2008.VisualizingprogramsimilarityintheAcplagiarismdetectionsystem. InProceedingsoftheWorkingConferenceonAdvancedVisualinterfaces(Napoli,Italy,May28-30, 2008).AVI'08.ACM,NewYork,NY,404-407.

[4]Zdroj: Dr.ThorstenBÃijring,Text&Documents,VisualizingandSearchingDocuments, VorlesungWintersemester2007/08

dôvodu, že táto metóda by našla v texte príliš veľa zhôd a vytvorila príliš „hustú dúhu“.

4.2.4 Farebné zvýraznenie

Pomocou tejto metódy sa farebne označujú textové bloky, ktoré sú si podobné. Osoba, ktorá kontroluje 2 súbory priamo vidí, vedľa seba zobrazené 2 súbory a priamo vidí, ktoré časti sú podobné. Táto metóda sa používa vo väčšine nástrojov na detekciu plagiarizmu. Patrí medzi ne napríklad Moss. Pomocou farby bloku sa dá napríklad určiť percentuálna zhoda medzi jednotlivými blokmi.



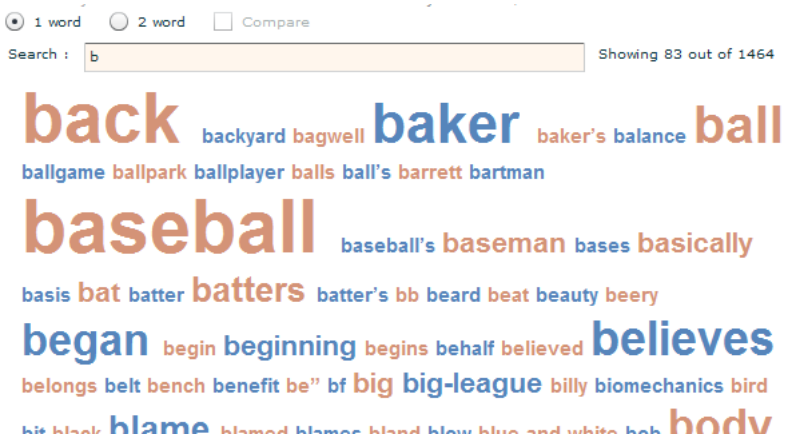
Obr. 4.5: Farebné zvýraznenie[5]

4.2.5 Tag clouds

Táto metóda vytvára z textu takzvaný „tag clouds“, čo je vlastne zoznam slov, ktoré sa v texte nachádzajú. Každému slovu priradí veľkosť jeho zobrazenia na základe toho, koľko krát sa v texte nachádza. Čím väčší je jeho výskyt, tým väčšie sa toto slovo zobrazí medzi

[5]Zdroj: `http://english236-w2008.pbworks.com/f/1206144080/example3.jpg`

ostatnými slovami. Pokiaľ majú 2 súbory podobný „tag clouds“, dá sa predpokladať, že ich obsahy sú si podobné.



Obr. 4.6: Tag clouds[6]

4.2.6 Word spectrum diagram

Metóda, ktorá pracuje na podobnom princípe ako predchádzajúca metóda Tag clouds. Táto metóda porovnáva výskyty jednotlivých slov v 2 súboroch a zobrazuje slová na základe ich výskytu v oboch súboroch. Jej výsledok je vidieť na grafe, ktorý je zobrazený na obrázku číslo 7.

Obr. 4.7: Word spectrum diagram[7]

Pokiaľ sa konkrétne slovo nachádza slovo v oboch súboroch, je umiestnené do stredu grafu, v opačnom prípade je umiestnené na kraji grafu. Bočné strany grafu reprezentujú

[6]Zdroj: `http://www.timshowers.com/wp-content/uploads/2008/08/tagcloud.gif`

[7]Zdroj: `http://www.timshowers.com/wp-content/uploads/2008/08/american-chinese-dist5.jpg`

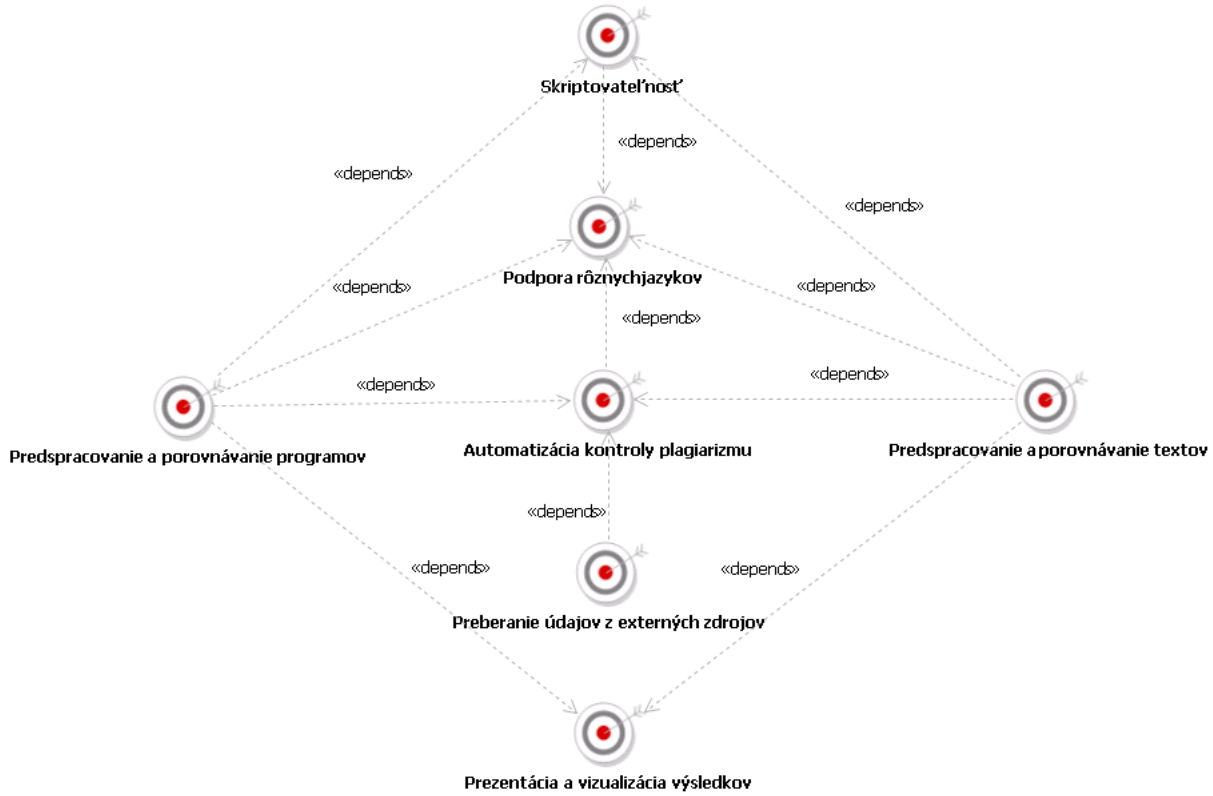
súbory, z ktorých pochádzajú slová. Veľkosť slov je tak ako v prípade Tag clouds daná ich výskytom v textoch.

Kapitola 5

Požiadavky na systém

5.1 Ciele projektu

Hlavné ciele nášho produktu sú zobrazené na obrázku 5.1. Tieto ciele spolu úzko súvisia, čo na obrázku predstavujú ich spojenia. Vo väčšine prípadov je úspešnosť jedného ciela závislá na úspešnosti iného, no v niektorých prípadoch ide o vzájomnú závislosť.



Obr. 5.1: Diagram cieľov projektu

- *Predspracovanie a porovnávanie programov*
  Produkt bude porovnávať zdrojové kódy rôznych programov a hľadať podobnosti. Zdrojový kód pred samotným porovnávaním predspracuje podľa charakteristiky daného jazyka a jednotlivé inštrukcie zamení za tokeny. Samotné porovnávanie už môže teoreticky ďalej prebiehať nezávisle na type pôvodného jazyka kódu.

- *Predspracovanie a porovnávanie textov*
  Ďalším cieľom je vyhľadávanie plagiátov medzi textami v bežnom jazyku. Produkt sa bude zameriavať najmä na slovenský jazyk, ale neskôr bude možné rozšíriť funkcionalitu aj o porovnávanie textov v anglických a ďalších jazykoch. Pred samotným porovnávaním sa text taktiež predpripraví, čím sa odstránia prázdne znaky a stop slová. Tieto dva prvé ciele patria medzi najdôležitejšie a od nich sa odvíjajú ostatné.

- *Podpora rôznych jazykov*
  Tento cieľ nadväzuje na predchádzajúce ciele. Produkt bude vedieť pracovať so zdrojovými kódmi viacerých jazykov, aby sa dosiahla čo najvyššia využiteľnosť. Taktiež bude možné spomínané rozšírenie o podporu textov v cudzích jazykoch.

- *Skriptovateľnosť*
  Skriptovateľnosť priamo súvisí s podporou viacerých jazykov. Podľa skriptov sa bude ovládať prevažne predspracovanie kódu a textu. Skriptovaním sa zabezpečí jednoduché rozšírenie programu, prípadne opravenie existujúcich chýb v algoritmoch. Vďaka tomu nebude nutné pri pridávaní a čiastočnom menení funkcionality zasahovanie do kódu programu, a tak budú môcť užívatelia, prípadne iní študenti vylepšovať tento produkt.

- *Preberanie údajov z externých systémov*
  Údaje z Internetu, informačných systémov a sietí môžu výrazne zlepšiť úspešnosť odhaľovania plagiátov.

- *Prezentácia a vizualizácia výsledkov*
  Keďže výsledok testovania plagiátov nie je konečný, ale vo väčšine prípadov je nutné ručné kontrolovanie programov a textov používateľom produktu, je veľmi dôležité aby výsledky testovania boli zrozumiteľné pre používateľa. Dobrá vizualizácia výsledkov môže výrazne ušetriť jeho čas.

- *Automatizácia kontroly plagiarizmu*
  Tento cieľ úzko súvisí s ostatnými cieľmi a funkcionalitou produktu. Produkt bude automaticky zisťovať jazyk kódu, alebo textu, predpriprávi ho, získa údaje z externých zdrojov a zobrazí výsledky, ktoré bude aj vizualizovať.

5.2 Analýza požiadaviek

Program by mal umožňovať kontrolovať zdrojové kódy (minimálne jazyky Java a C) a slovenské texty a vyhľadávať v nich plagiáty. Predspracovanie vstupných súborov (predov-

šetkým programových zdrojových kódov) by mala byť skriptovateľná, aby bolo možné jednoducho pridávať podporu pre ďalšie programovacie jazyky. Skripty by mali zabezpečovať celkové predspracovanie a svoj výstup poskytnúť systému na kontrolu podobnosti.

Dáta na kontrolu by mali byť získavané z viacerých zdrojov, konkrétne z pevného disku, databázy, a informačného systému (AIS, Moodle). Podporovanými vstupnými formátmi okrem textových súborov by mal byť minimálne formát Microsoft Word, prípadne aj Portable Document Format (PDF).

Aplikácia by mala ponúkať prehľadné používateľské rozhranie, ktoré by umožňovalo nastavovať parametre jednotlivých porovnávacích algoritmov, určovať súbory na kontrolu, a pod. Malo by byť možné určiť, čo s čím porovnávať – jeden dokument s množinou dokumentov, každý dokument s každým a pod.

Ďalšou funkciou by mala byť prehľadná vizualizácia výsledkov. Minimálne by to malo byť farebné zvýraznenie podozrivých častí vstupných súborov, aby tieto mohli byť následne podrobené manuálnej kontrole. Systém by mal taktiež poskytovať grafické zobrazenie štatistiky porovnávania.

Keďže porovnávanie textov (predovšetkým pre veľký počet vstupných súborov) je časovo náročná operácia, porovnávacie algoritmy by mali byť optimalizované tak, aby čo najefektívnejšie využívali hardvér počítača. Mala by byť implementovaná podpora multiprocesorových systémov (spracovávanie textov vo viacerých samostatných vláknach).

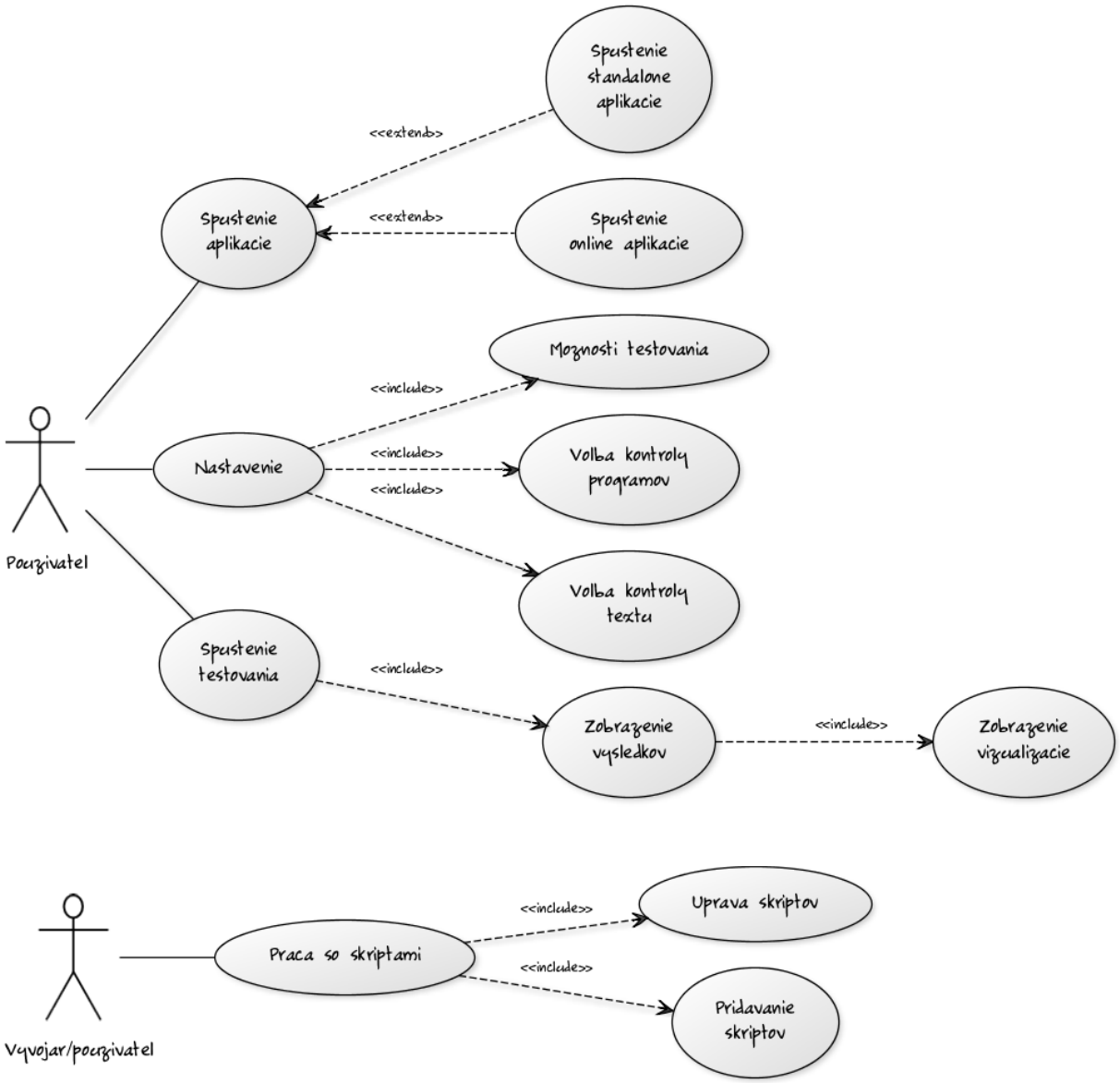
5.3 Model prípadov použitia

Na obrázku 5.2 sú zobrazené prípady použitia produktu a hráči ktorý vystupujú. So systémom bude pracovať v prvom rade používateľ. Keďže sa nejedná o systém so zložitou sieťovou architektúrou, ale skôr o samostatnú aplikáciu, nebude nutná jeho dodatočná údržba. Preto v prípadoch použitia nevystupuje administrátor. Uvažujeme iba používateľa a prípadného vývojára, ktorý by chcel produkt rozšíriť pomocou skriptov.

- *Spustenie aplikácie*
  Užívateľ pred prácou s produktom, musí najprv samotnú aplikáciu spustiť. Na tento prípad použitia nadväzujú nasledujúce dva, podľa toho o aký druh aplikácie sa jedná.

- *Spustenie standalone aplikácie*
  Užívateľ spustí aplikáciu na svojom počítači, kde ju má uloženú. Jedná sa o bežnú aplikáciu a po jej spustení sa zobrazí okno s jednotlivými položkami. Jednotlivými tlačítkami ovláda funkcionalitu aplikácie.

- *Spustenie online aplikácie*
  Pokiaľ užívateľ nemá aplikáciu vo svojom počítači, môže ju spustiť aj online pomocou webového rozhrania. Aplikácia sa mu zobrazí v okne prehliadača a jej funkcionalita bude rovnaká ako v predošlom prípade. Po práci s aplikáciou jednoducho zavrie okno, alebo záložku v prehliadači.



Obr. 5.2: Diagram modelu prípadov použitia

- *Nastavenie*
  Užívateľ si bude môcť prispôsobiť program. Vybrať si aký algoritmus ma program používať, ako chce mať zobrazené výsledky, kam ich ukladať a hlavne musí vybrať súbory, ktoré sa majú testovať.

- *Možnosti testovania*
  Užívateľovi sa po voľbe druhu testovania, teda či chce testovať programy alebo texty, zobrazí nastavenie tohto testovania. Pre oba druhy testovania budú niektoré spoločné nastavenia programu.

- *Voľba kontroly programov*
  Užívateľ vyberie možnosť kontrolovania zdrojových kódov programov. Po tejto voľbe sa mu zobrazí výber algoritmu, nastavenie hranice podobnosti pre podozrivé programy, formulár pre výber vstupných a výstupných zložiek a ďalšie.

- *Voľba kontroly textu*
  Pri voľbe testovania dokumentov bude taktiež môcť si voliť z dostupných algoritmov pre testovanie a predprípravu, alebo vybrať jazyk v akom je dokument napísaný, pokiaľ je dostupný.

- *Spustenie testovania*
  Po nastavení aplikácie a testovania zvolí užívateľ začatie testovanie. Počas testovania sa bude zaznamenávať a zobrazovať trvanie testu, odhadovaný čas pre dokončenie a stavový riadok s percentuálnym dokončením testu.

- *Zobrazenie výsledkov*
  Užívateľovi sa zobrazia výsledky testovania, ktoré sa taktiež uložia do zvolenej zložky na počítači, pre neskoršiu prácu s nimi. Budú to prevažne dvojice porovnávaných súborov s percentuálnymi, alebo inými zhodami podľa algoritmu a zvýraznenie jednotlivých podobných fragmentov.

- *Zobrazenie vizualizácie*
  Pre lepšiu prehľadnosť si užívateľ vyberie možnosť vizualizácie výsledkov jednou z dostupných metód. Táto vizualizácia sa zobrazí a taktiež uloží vo forme obrázku po dokončení testovania.

- *Práca so skriptami*
  Program bude ponúkať možnosť práce so skriptami. V hlavnom menu programu užívateľ zvolí kliknutím na príslušné tlačítko túto voľbu. Pri práci s online aplikáciou to bude bezpečnostne ošetrené z dôvodu neželaných zásahov do existujúcej funkcionality.

- *Úprava skriptov*
  Pri úprave skriptov sa mu zobrazí ich zoznam a užívateľ si vyberie daný skript a zvolí si či ho chce zmeniť, prípadne zmazať. Pri zmene sa mu zobrazí okno s textom, kde

ho môže upravovať a ukladať zmeny. Po dokončení práce zavrie formulár voľbou návratu do menu.

- *Pridávanie skriptov*
  Pridávanie skriptov bude podobné ako ich úprava. Pri tejto voľbe sa užívateľovi zobrazí prázdne okno, kde môže napísať daný skript. Nad ním si vyberie názov skriptu a počas práce bude môcť ukladať zmeny. Po napísaní skriptu sa podobne ako pri úprave skriptov vráti späť do menu.

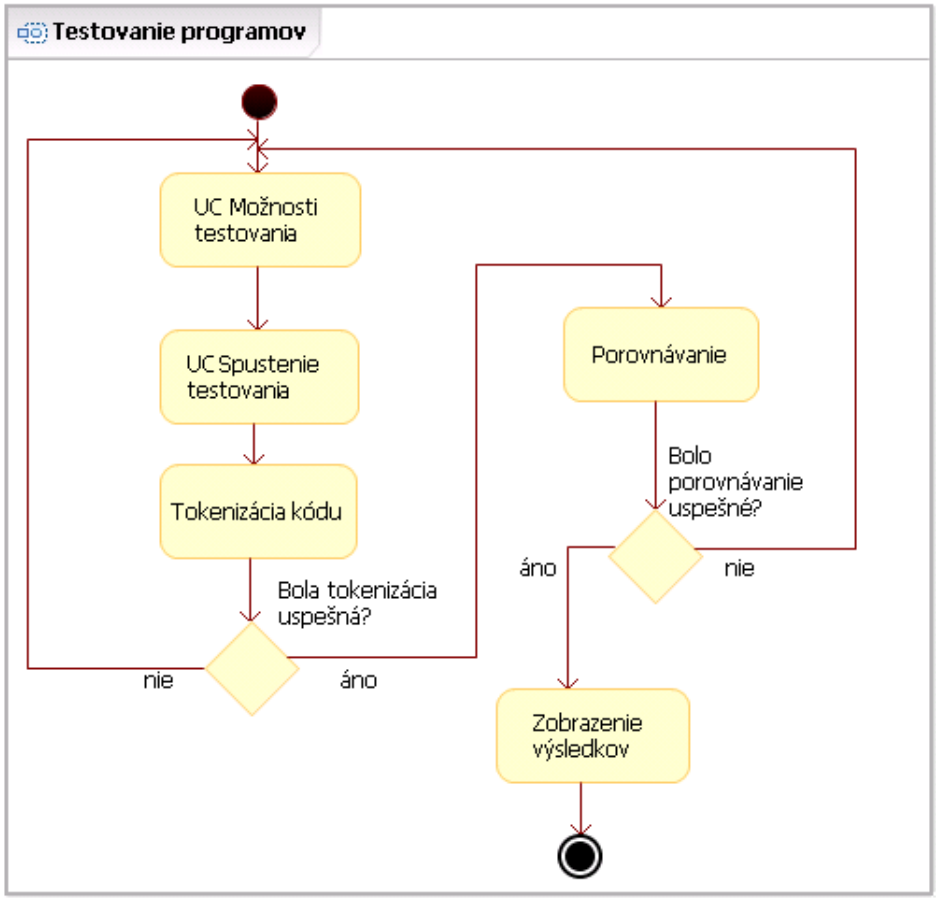
5.4 Procesný model

Testovanie programov

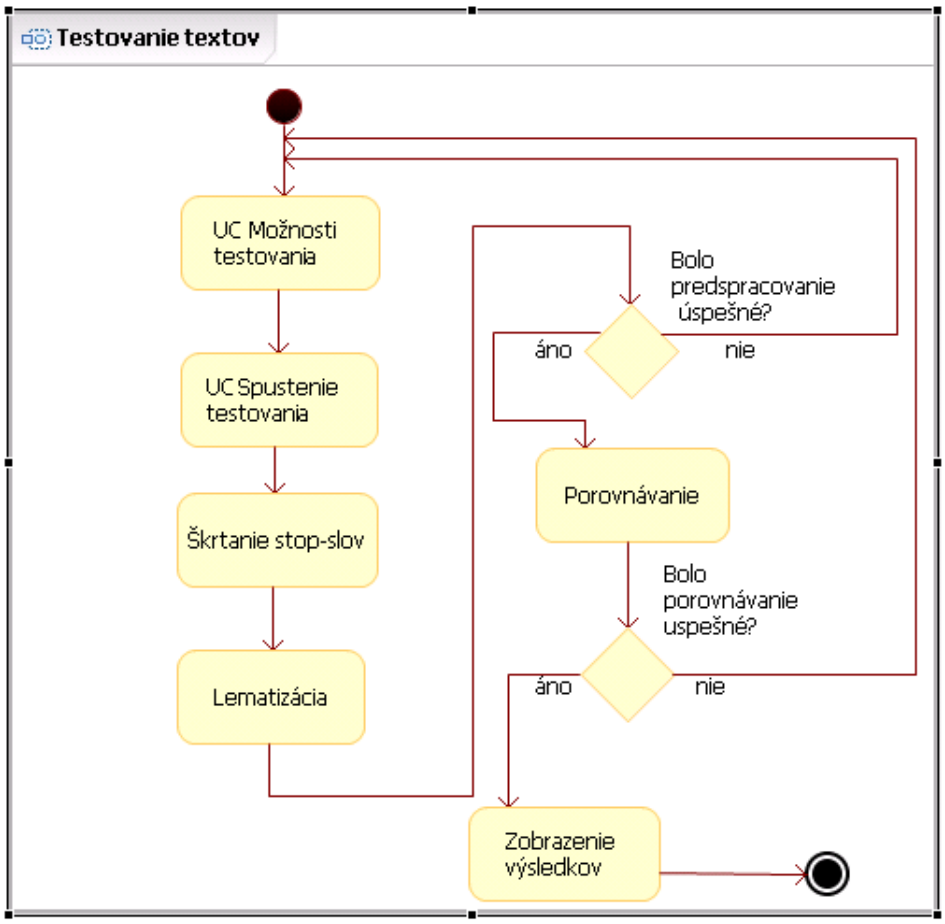
Na obrázku 5.3 je zobrazený model procesu testovania súborov so zdrojovým kódom. Na začiatku si užívateľ prispôsobí nastavenie testovania a následne ho spustí. Program potom najprv tokenizuje kód súborov a potom ich porovnáva. Ak pri jednom z týchto procesov došlo k chybe, program ju zobrazí a vráti sa naspäť. Na konci zobrazí výsledky testovania.

Testovanie textov

Na obrázku 5.4 je zobrazený model procesu testovania súborov obsahujúcim text. Podobne ako pri testovaní programov si užívateľ najprv nastaví spôsob testovania a potom spustí samotné testovanie. Program potom najprv škrtá stop-slová a ostatné slová upravuje na základný tvar. Ak všetko prebehlo úspešne pokračuje porovnávaním a na konci zobrazí výsledok testovania. Ak došlo k chybe vráti sa na začiatok.



Obr. 5.3: Proces testovania programov



Obr. 5.4: Proces testovania textov

Kapitola 6

Návrh riešenia

V tejto kapitole sa budeme venovať návrhu nášho riešenia. Najprv predstavíme architektúru, ktorá odráža vlastnosti danej aplikačnej domény, ktoré sme mali o možnosť získať a zúročiť ich pri jej návrhu. Nasleduje podrobný návrh niektorých častí systému, s dôrazom najmä na tie komponenty, ktorých implementácia.

6.1 Architektúra systému

Architektúra systému je navrhnutá tak, aby bola možné systém jednoducho rozširovať o podporu nových programovacích jazykov (v prípade kontroly zdrojových kódov), prirodzených jazykov (kontrola textu), prípadne formátov vstupných súborov. Znázornená je na obr. 6.1.

Jadro (CORE)

Úlohou jadra je sprostredkovávanie komunikácie medzi ostatnými modulmi. Jadro by malo byť čo najjednoduchšie a jeho funkcionalita by mala byť čo najmenšia.

Graphical User Interface

Modul Graphical User Interface (GUI) má za úlohu získavať vstupy od používateľa a zobrazovať výstupy systému. Jeho funkcionalita by mala byť obmedzená na minimum, a všetky dáta by mu mali byť poskytnuté inými modulmi.

Command Line Interface

Command Line Interface (CLI) slúži na ovládanie aplikácie z príkazového riadku. Správanie programu je v takomto prípade ovplyvňované zadanými parametrami.

Combiner

Úlohou modulu Combiner je vytvárať dvojice vstupných súborov, ktoré budú prostredníctvom jadra odoslané modulu Parser manager na kontrolu podobnosti. Dvojice sa budú vyberať z množiny načítaných vstupných súborov na základe pravidiel, ktoré modulu Combiner poskytne modul Compare manager.

Compare manager

Compare manager definuje pravidlá, na základe ktorých sa vytvárajú dvojice súborov určených na kontrolu vzájomnej podobnosti. Vytvárané dvojice súborov sú postupne posielané modulu Parse manager. Základnou funkcionalitou modulu je vytvorenie všetkých usporiadaných dvojíc danej množiny vstupných súborov. Táto funkcionalita môže byť rozšírená o rôzne iné pravidlá vytvárania dvojíc, ktoré môžu byť užitočné pri špeciálnych prípadoch použitia systému.

Parse manager

Tento modul spravuje skripty, ktoré definujú pravidlá predspracovania vstupných súborov. Vstupom pre parse manager je dvojica vstupných súborov, na ktoré je aplikovaný daný parser skript. Úlohov parser skriptov je predspracovať vstupné súbory pred aplikovaním porovnávacieho algoritmu. Porovnávacie algoritmy sú súčasťou modulu Parser manager.

Loader

Úlohou tohto modulu je načítanie textu z rôznych formátov súborov. Súbory sú získavané z pevného disku, databáz a rôznych externých systémov. Výstupom modulu je množina textových súborov, ktoré budú spracovávané systémom.

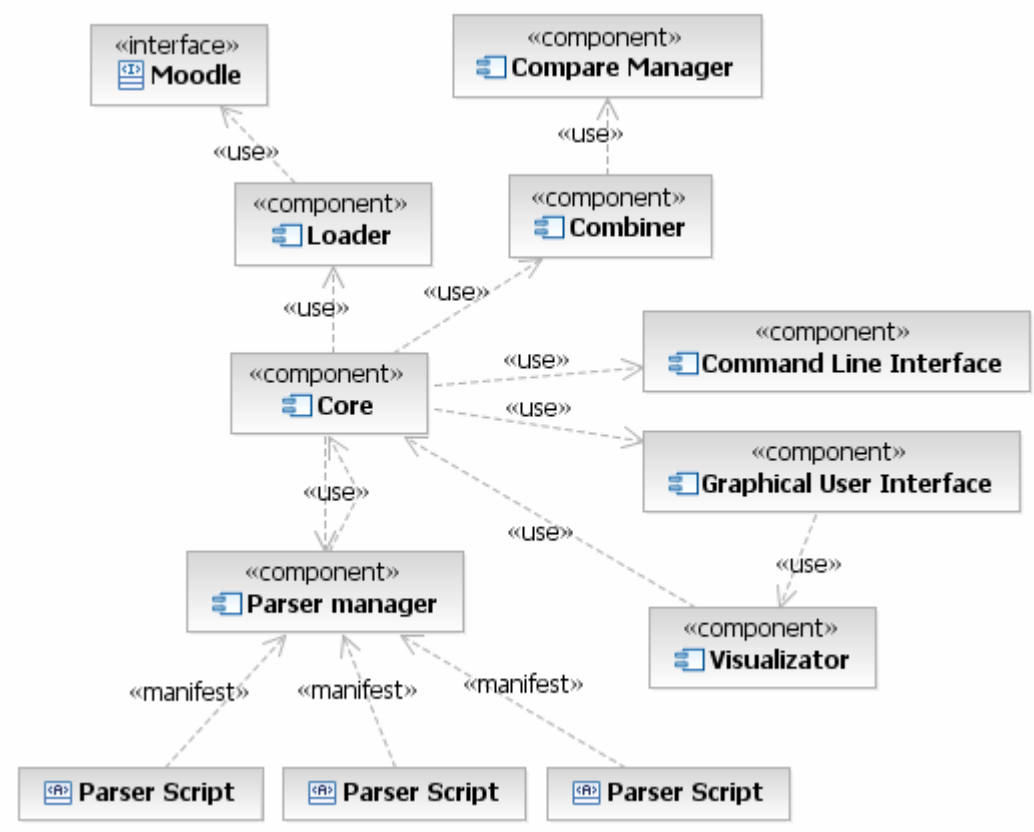
Vizualizátor

Vizualizačný modul má za úlohu zobraziť prostredníctvom GUI modulu porovnanie vstupných súborov, ktoré boli vyhodnotené ako podobné. Môže taktiež obsahovať vytváranie štatistík porovnávania. Dáta na vizualizáciu mu poskytne modul Core.

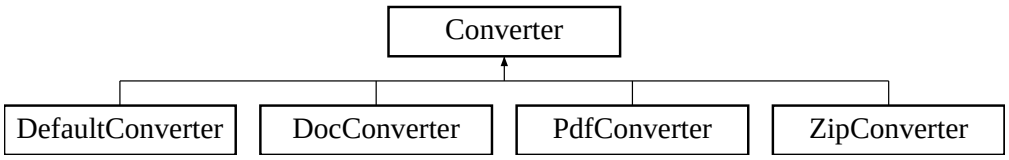
6.2 Načítanie dát (Loader)

Konverzia formátov

Pre úspešné porovnávanie obsahov súborov je nutné mať všetko v rovnakom formáte. Pre porovnávanie súborov sa budú využívať textové súbory v kódovaní utf8. Súbory sa skonvertujú a prekopírujú do pracovného adresára aplikácie. Jednotlivé odvodené moduly Converter budú vykonávať konverziu z rôznych iných formátov do textovej formy.



Obr. 6.1: Architektúra systému



Obr. 6.2: Dekompozícia modulu Converter

DefaultConverter - tento Converter nebude robiť žiadnu konverziu, jednoducho načíta zdrojový súbor a prekopíruje ho do pracovného adresára. Bude použitý ak aplikácia nebude schopná identifikovať formát zdrojového súboru.

DocConverter - ako už sám názov napovedá, bude sa jednať o konverziu z .doc formátu bežného v kancelárskej aplikácii MS Office. Na konverziu bude slúžiť externá aplikácia, ktorú tento modul zavolá. Do úvahy prichádza AbiWord a OpenOffice, keďže to sú Open Source aplikácie a nebude problém s licencovaním.

PdfConverter - skonvertuje z formátu pdf, čo je bežne používaný formát pre dokumenty určené na tlač a publikovanie na webe. Na konverziu bude opäť slúžiť externá aplikácia pdftotext, bežne dodávaná s Open Source balíkom Poppler.

ZipConverter - rozbalenie zip archívu do pracovného adresára. Zip archív sa rozbalí do podadresára pracovného adresára, pričom názov podadresára bude zhodný s názvom zip archívu. Takto bude možné jednoducho identifikovať, z ktorého archívu súbory pochádzajú, ak ich bude na vstupe viac. Súčasne ZipConverter bude vedieť iniciovať spojenie viacerých súborov do jedného, čo je nutnosť ak máme 1 program rozdelený do viac súborov zdrojového kódu.

Dôležitou súčasťou konverzie formátu je samotná detekcia a konverzia kódovania zdrojového súboru. Študenti totiž môžu používať rôzne kódovania (od CP1250 bežného na OS Windows, po utf-8 či ISO-8859-2, ktoré sú bežné na OS typu Unix). Na detekciu a konverziu kódovania sa použije externá aplikácia.

Integrácia s LMS Moodle

Do systému moodle je možné pristupovať priamo pomocou prístupu na súborový systém alebo pomocou existujúcich php metód implementovaných v moodle. Obe metódy budú implementované alebo aspoň z časti až kým nebude definitívne jasné, ktorá je výhodnejšia.

Metóda prístupu na disk

Fyzická cesta k súborom odovzdania moodle je *datafolder/idKurz/moddata/assigment/idMiestaOdovzdania/idPouzivatela/subor* pričom premenné sú idKurz, idMiestaOdovzdania, idPouzivatela.

Na server, kde beží systém moodle sa pridajú skripty potrebné k zabezpečeniu nasledujúcich úkonov:

- Získanie popisu jednotlivých kurzov a ich id.
- Získanie popisu miest odovzdaní a ich priradenie ku kurzom
- Získanie priradenia mien používateľov a ich id.

Tieto úkony budú vykonané pomocou dotazov do databázy PostgreSQL implementované v novo vytvorených skriptoch ktoré budú mať prístup do databázy systému moodle. Žiadne pôvodné súbory moodle nebudú menené.

Po tom ako získame cesty k súborom, treba tieto súbory spojiť do jedeného súboru napríklad použitím zip programu a následne ponúknuť na stiahnutie. Zabezpečenie jednotlivých skriptov bude riešenie pomocou http access metódy.

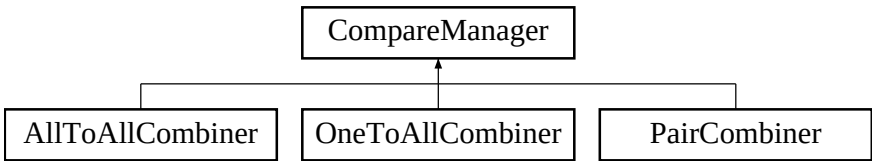
Metóda využitia funkcií systému moodle

Pri tejto metóde je potrebné identifikovať dôležité metódy v systéme, ktoré vyskladajú http link na konkrétne odovzdanie a následne ho ponúknu na stiahnutie. Najjednoduchšie je nájsť priamo skript, ktorý ponúka okrem iného aj všetky odovzdania v danom mieste odovzdania na stiahnutie. Následne treba odfiltrovať zo skriptu potrebné http linky a to buď na strane servera alebo klienta.

6.3 Manažment porovnávania (Compare Manager)

Keďže niektoré metódy porovnávania nedosahujú rovnakých výsledkov nezávisle na tom, či sa porovnáva prvý súbor s druhým alebo opačne, nie je klasické párovanie podľa všetkých usporiadaných dvojíc postačujúce. V tomto prípade je rozdiel na poradí súborov vo dvojici a je potrebná obojsmerná kontrola. Preto bude môcť Compare manager vytvoriť aj takéto dvojice. Bude ich ale dvakrát viac, a preto aj takáto kontrola bude trvať dvakrát dlhšie. Naopak, v niektorých situáciách je potrebné kontrolovať iba jeden súbor s ostatnými, ktoré sú v databáze odovzdaných súborov. Vtedy je zbytočné a zdĺhavé, aby sa kontrolovali všetky súbory vzájomne. Compare manager bude preto môcť vytvoriť aj také párovanie pri ktorom sú vytvorené dvojice jedného kontrolovaného súboru so všetkými ostatnými. Preto bude aplikácia obsahovať celkovo tri rôzne možnosti párovania súborov:

- OneToAllCombiner - každy s každým bez opakovania
- AllToAllCombiner - každý s každým s opakovaním (obojsmerne)
- FirstToAllCombiner - prvý s každým bez opakovania



Obr. 6.3: Dekompozícia modulu CompareManager

6.4 Predspracovanie

Odstraňovanie stop-slov a lematizácia

Pred samotným porovnávaním textov sa texty najskôr predspracujú. Predspracovanie textu obnáša:

- **Odstránenie stop slov**
  Stop slová sa načítajú z textového súboru, ktorý obsahuje stop slová oddelené znakom nového riadku a uložia sa do kolekcie. Tá sa neskôr predá ako argument funkcii, ktorá tieto stop slová bude odstraňovať. Na tento účel sa použijú regulárne výrazy.

- **Lematizácia**
  Na lematizáciu použijeme CDB databázu, čo je implementácia hash mapy, umožňujúca rýchle vyhľadávanie. Tak ako pri odstraňovaní stop slov použijeme regulárne výrazy. Pomocou nich rozložíme spracovávaný text na jednotlivé slová. Ak sa bude konkrétne slovo nachádzať v databáze, bude nahradené jeho základným tvarom.

Takto upravené texty sú pripravené na porovnávanie.

Tokenizácia

Porovnávaniu zdrojových kódov predchádza predspracovanie vstupných súborov – tokenizácia. Pri nej bude vytvorená z každého zdrojového kódu postupnosť symbolov (tokenov), ktorá predstavuje logický obsah tokenizovaného súboru. Jednotlivé tokeny sú reprezentované celočíselnými hodnotami. Tokenizáciu zdrojových kódov je potreba vykonávať pre rôzne programovacie jazyky. Spočiatku bude aplikácia zameraná na C a Javu, ostatné jazyky je možné doplniť neskôr, počíta sa so samostatným modulom pre každý ďalší jazyk. Rozdielnosť tokenizácie pritom bude spočívať v rozdielnom spracovaní syntaxe ale aj paradigmy programovacieho jazyka.

Predspracovanie zdrojových kódov bude prebiehať vo viacerých krokoch. Samotná tokenizácia sa bude vykonávať nad reťazcami, ktoré sú do istej miery modifikovanými verziami pôvodných vstupných súborov. Toto predspracovanie pred tokenizáciou odstráni nadbytočné zlomy riadkov, biele znaky, komentáre, textové reťazce a rôzne jazykové definície (v jazyku C známy `#define`).

Upravený zdrojový kód bude následne pretvorený na strom, ktorého vrcholy reprezentujú logické bloky v zdrojovom kóde. Bloky budú určované nie len na základe zložených zátvoriek, ale aj kľúčových slov, ktoré deklarujú nový blok (v jazyku C sú to napríklad kľúčové slová if, while, for, atď.). Hrany stromu budú vytvorené vložením špeciálnej riadiacej konštrukcie priamo do tokenizovaného zdrojového kódu. Táto konštrukcia bude odkazovať na blok, ktorý sa na tomto mieste v pôvodnom zdrojovom kóde nachádzal.

Po vytvorení stromu budú všetky vrcholy tokenizované – pre každý vrchol bude vytvorený zoznam tokenov, ktoré reprezentujú jednotlivé výrazy v tomto bloku (výrazom v jazyku C rozumieme časti kódu oddelené bodkočiarkou, v ostatných programovacích jazykoch obdobne). Špeciálnym tokenom je práve riadiaca konštrukcia pre definíciu bloku.

Zoznam ostatných navrhovaných tokenov je nasledovný (ale nie je problém kedykoľvek ho rozšíriť):

1. ASSIGN – priradenie
2. FUNC CALL – volanie známej funkcie (štandardné knižničné funkcie jazyka C a pod. Tu je vhodné poznať všetky štandardné knižničné funkcie daného jazyka a vhodnou dátovou štruktúrou je hash mapa.)
3. LOCAL FUNC CALL – volanie neznámej funkcie
4. OPERATER – aritmetická operácia
5. LOG OPERATOR – logická operácia
6. CONST – číselná konštanta
7. TEST – zodpovedá strojovým inštrukciám TEST, prípadne CMP (porovnanie hodnoty)
8. JUMP – zodpovedá strojovým inštrukciám skokov
9. LABEL – návestie

Tokeny TEST, JUMP a LABEL budú dopĺňané do blokov na základe toho, o aký blok sa jedná. Jednotlivé riadiace štruktúry programovacieho jazyka sa totiž vyznačujú istými charakteristikami, ktoré je možné opísať práve pomocou týchto tokenov. Tokenizácia preto vytvorí viac-menej zhodné postupnosti tokenov pre zdrojové kódy s rovnakou logickou štruktúrou, avšak zapísanou pomocou rôznych riadiacich štruktúr. Toto robí navrhovanú metódu tokenizácie unikátnou a súčasne robustnou voči zásahom plagiátorov do riadiacich štruktúr zdrojového kódu.

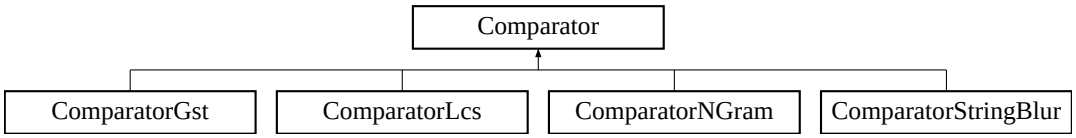
Posledným krokom tokenizácie bude vytvorenie postupnosti tokenov zo stromu reprezentujúceho zdrojový kód. Výstupom tejto akcie bude reťazec číselných označení jednotlivých tokenov, teda tokenizovaný zdrojový kód.

6.5 Komparátory

Na nasledujúcom obrázku sa nachádza vizualizácia dekompozície triedy Comparator do komparátorov pre jednotlivé metódy.

N-gramy

Pre správny beh n-gramov je potrebné mať k dispozícii texty s už odstránenými stop slovami. Jednotlivé stop slová musia byť oddelené medzerou. Následne porovnávame text A k textu B. V texte A vytvárame postupne N-tice slov. Prvú n-ticu bude tvoriť prvých



Obr. 6.4: Dekompozícia modulu Comparator

n slov. Druhú nticu tvorí druhé až n+1 slovo. Postupne vytvoríme všetky možné n-tice slov, ktoré nasledujú za sebou až do konca textu. Po každom vytvorení takejto n-tice kontrolujeme výskyt rovnakého reťazca v texte B. Podobnosť dvoch textov bude číselne reprezentovaná ako pomer úspešných nájdení zhody reťazca v texte B k počtu porovnaní.

Latentná sémantická analýza

Vzhľadom k tomu, že metóda LSA neoperuje nad dvojicami súborov, ale spracováva celú vzorku naraz, jej zakomponovanie do architektúry riešenia si vyžaduje separátne volanie komparátora priamo z jadra aplikácie. Komparátor pre LSA je preto jediným komparátorom, ktorý nededí od všeobecného komparátora.

Navrhovaná LSA pre našu aplikáciu sa skladá z dvoch častí. V prvej časti prebehne extrakcia fráz z dokumentov a ich uloženie do hashovacej tabuľky, spolu s príslušnými metadátami. V druhej časti sa podľa už uvedeného vzťahu vytvorí matica dokumentov a tá sa spracuje algoritmom SVD. Výsledkom je matica podobnosti.

LSA je metóda časovo náročná, preto okrem štandardnej implementácie riešenie bude obsahovať aj implementáciu stochastickú, ktorá bude danú frázu extrahovať s pravdepodobnosťou p. Pre aplikáciu navrhujeme 2 základné stochastické varianty LSA:

- stochastická LSA s pravdepodobnosťou extrakcie 20%
- stochastická LSA s pravdepodobnosťou extrakcie 5%

String-blurring

Tento komparátor je implementáciou nami navrhnutého algoritmu String-blurring určeného na detekciu podobnosti textových reťazcov. Rovnako, ako väčšina ostatných, jeho vstup tvoria dva textové reťazce (ktoré môžu byť predspracované). Samotný komparátor String-blurring vykonáva ďalšiu úroveň predspracovania, ktorou je odstránenie bielych znakov a prevedenie textov na postupnosti číselných hodnôt na základe číselných kódov Unicode. Algoritmus ďalej pracuje výhradne s týmito postupnosťami (signálmi diskrétnych hodnôt).

Prvým krokom algoritmu po predspracovaní je filtrácia vstupných signálov dolnopriepustným filtrom. Využíva sa pri tom gaussovský konvolučný filter – výsledné signály sú takpovediac „vyhľadené“ verzie pôvodných signálov, z čoho dostal algoritmus svoje meno („rozmazávanie reťazcov“). Odstránením vysokých frekvencií v tomto kroku je zabezpe-

čené, že algoritmus bude schopný odhaliť podobnosť i v textoch, ktoré boli mierne modifikované oproti originálu.

Ďalším krokom je vzájomné odčítanie filtrovaných signálov pri všetkých možných vzájomných posunutiach. Výsledkom je množina nových signálov, ktoré sú tvorené rozdielom prekrývajúcich sa častí vstupných signálov pri jednotlivých posunutiach. V týchto signáloch je vyhľadaná taká najdlhšia súvislá podpostupnosť, ktorá je tvorená hodnotami menšími ako zadaná prahová hodnota. Táto prahová hodnota, spolu s polomerom „rozmazávania" reťazcov, tvoria vstupné parametre algoritmu a ich hodnoty sú určené experimentálne testovaním na vzorke slovenských textov.

Výhodou nášho algoritmu je, že pracuje nad jednotlivými znakmi, nie nad celými slovami. Nevyžaduje preto predspracovanie a je možné využiť ho pre hľadanie podobnosti v textoch písaných ľubovoľným prirodzeným jazykom. Okrem toho, algoritmus je odolný voči malým zmenám v texte oproti originálu, ktoré môžu byť relatívne časté.

6.6 Grafické používateľské rozhranie (GUI)

Grafické používateľské rozhranie tvorí nadstavbu nad konzolovou aplikáciou, ktorá je zodpovedná za všetky výpočty súvisiace s porovnávaním súborov a detekciou plagiarizmu. Je tvorené samostatnou aplikáciou, ktorá umožňuje určiť potrebné vstupné parametre konzolovej aplikácie pomocou grafických ovládacích prvkov, a taktiež zobraziť výsledky prehľadným spôsobom.

Používateľské rozhranie je koncipované tak, aby bolo čo najotvorenejšie v zmysle rozšíriteľnosti, prípadne modifikovateľnosti a preto bola zvolená modulárna architektúra. GUI je tvorené šiestimi hlavnými modulmi – hlavné okno, správa nastavení, modul integrácie s konzolovou aplikáciou, modul vizualizácie výsledkov, modul exportu výsledkov, a modul integrácie so systémom Moodle.

Rovnako, ako konzolová časť aplikácie, používateľské rozhranie je nezávislé od platformy. Čo najväčšie množstvo ovládacích prvkov je súčasťou hlavného okna, vďaka čomu je aplikácia intuitívna a jednoducho použiteľná. Vďaka využitiu dokovacích okien je však toto možné zmeniť a niektoré ovládacie prvky umiestniť do samostatných okien oddelením dokovacieho okna od hlavného okna aplikácie.

6.6.1 Hlavné okno

Tento modul tvorí jadro grafického používateľského rozhrania. Sprostredkováva komunikáciu medzi ostatnými modulmi, a má taktiež za úlohu vykreslenie a správu všetkých grafických prvkov a samotného okna aplikácie. Zabezpečuje taktiež interakciu s používateľom prostredníctvom hlavného menu (a panela nástrojov, ktorý poskytuje rýchly prístup k najdôležitejším funkciám prístupným z hlavného menu).

6.6.2 Správa nastavení

Úlohou tohto modulu je poskytnúť používateľovi sadu grafických ovládacích prvkov, ktoré sú mapované na dostupné parametre konzolovej časti aplikácie. Ovládacie prvky sú rozdelené na dve logické skupiny – parametre kontroly a špecifikácia vstupných súborov. Každá z týchto skupín ovládacích prvkov je sústredená do dokovacieho okna, ktoré je súčasťou hlavného okna aplikácie, avšak je možné ho oddeliť, presunúť, prípadne skryť. Toto rozloženie podporuje ergonómiu práce s používateľským rozhraním.

6.6.3 Modul integrácie

Tento modul tvorí prostriedok komunikácie medzi používateľským rozhraním a konzolovou časťou aplikácie. Odovzdáva parametre zvolené prostredníctvom modulu správa nastavení konzolovej aplikácií, a spracováva jej výstup. Modul integrácie je taktiež zodpovedný za spustenie konzolovej aplikácie na pozadí a detekciu chybových stavov, ktoré môžu nastať. V neposlednom rade poskytuje hlavnému oknu informáciu o časovom priebehu kontroly, vďaka čomu je možné zobraziť tento priebeh graficky.

Všetky závislosti od konzolovej časti aplikácie sú sústredené v tomto module. Zároveň je rozhranie tohto modulu čo najjednoduchšie. Vytvorením alternatívnej verzie tohto modulu (prípadne jeho modifikáciou) je teda GUI teoreticky využiteľné aj pre iné konzolové aplikácie podobného zamerania.

6.6.4 Modul vizualizácie výsledkov

Po skončení kontroly konzolovou aplikáciou odovzdá modul integrácie výsledky modulu vizualizácie výsledkov. Tento modul spravuje jednotlivé vizualizátory – podmoduly, ktoré majú za úlohu zobraziť výsledky nejakým konkrétnym spôsobom. Modul vizualizácie výsledkov zobrazuje vizualizátory na záložkách a poskytuje im notifikácie o zmene výsledkov a o zmene prahu podobnosti – hodnoty, ktorá určuje, koľko percentná podobnosť sa považuje za podozrenie na plagiarizmus. Túto hodnotu určuje používateľ prostredníctvom modulu správy nastavení a vizualizátory ju patrične reprezentujú. Vo všeobecnosti sú dvojice súborov, ktorých podobnosť dosahuje minimálne hodnotu prahovej podobnosti, zobrazené výraznejším spôsobom (napr. červenou farbou).

Všetky vizualizátory sú rozšírením jednej spoločnej triedy, ktorá poskytuje potrebné rozhranie pre komunikáciu s modulom vizualizácie výsledkov. Samotný vizualizátor má teda jedinú úlohu – vykresliť výsledky. Integrácia nového vizualizátora do aplikácie potom spočíva vo vytvorení inštancie jeho triedy a umiestnenie tejto inštancie do zoznamu vizualizátorov, ktorý je súčasťou vizualizačného modulu. Aplikácia je teda veľmi jednoducho rozšíriteľná o nové spôsoby vizualizácie výsledkov.

6.6.5 Modul exportu výsledkov

Tento modul je veľmi podobný vizualizačnému modulu. Namiesto vizualizátorov však spravuje takzvané exportéry – podmoduly, ktoré sú zodpovedné za export výsledkov dp súboru v nejakom konkrétnom formáte. Modul exportu výsledkov pri tom poskytuje funkcionalitu, akou je zobrazenie dialógového okna, prípadne zobrazenie jednotlivých exportérov v hlavnom menu.

Modul exportu výsledkov je rovnako ako vizualizačný modul veľmi jednoducho rozšíriteľný o nové exportéry bez znalosti vnútornej logiky fungovania aplikácie.

6.6.6 Modul integrácie so systémom Moodle

Tento modul poskytuje možnosť stiahnuť odovzdané zadania zo systému Moodle. Použitá je pri tom http autentifikácia a šifrovaný prenos (https). Používateľ má možnosť zvoliť si konkrétne miesto, z ktorého sú odovzdané zadania stiahnuté vo formáte zip. Aplikácia stiahnuté zadania extrahuje z prijatého archívu a automaticky pripojí na koniec zoznamu kontrolovaných súborov. Odovzdané zadanie pritom môže obsahovať viacero súborov, a preto je každé zadanie uložené v samostatnom adresári (využíva sa schopnosť konzolovej aplikácie spájať súbory zo spoločného adresára do jedného, čím je zabezpečené, že všetky súbory jedného zadania sú kontrolované spoločne ako jedno zadanie).

6.7 Vizualizátor side-by-side

Vizualizácia bude implementovaná pomocou 2 zobrazených vedľa seba umiestnených textových polí. V nich budú zobrazované obsahy porovnávaných súborov. Zobrazený obsah sa bude dať jednoducho meniť pomocou select boxov, ktoré budú umiestnené nad textovými poliami. To umožní zobraziť ľubovolnú dvojicu súborov a následne ju vizualizovať. Pri vizualizácii budú farebne označené bloky textu, ktoré sa nachádzajú v oboch vybraných súboroch. Samotné informácie o podobnosti v dvojiciach súborov budú poskytovať triedy, ktoré sú zodpovedné za porovnávanie súborov. Tie súbory porovnajú a informácie potrebné na vizualizáciu zapíšu do súboru, ktorý sa uloží na disk. V ňom sa budú nachádzať informácie o rovnakých blokoch, ktoré sa našli pri porovnávaní. Vizualizátor si tento súbor načíta a farebne zvýrazní nájdené bloky. Pre lepšiu orientáciu bude každý blok textu označený inou farbou.

Kapitola 7

Implementácia

7.1 Jadro aplikácie

Jadro tvorí kostru celej aplikácie. Využíva funkcionalitu ostatných modulov na vykonávanie kontroly plagiarizmu. Jeho úlohou je využiť modul nahrávania na načítanie kontrolovaných súborov do pamäte, vytvoriť a spravovať vlákna reprezentované inštanciami triedy `ParserManager` (trieda modulu manažmentu analýzy), a po skončení kontroly zobraziť výsledky.

Modul jadra nie je tvorený samostatnou triedou. Implementovaný je ako funkcia `main()` a teda tvorí vstupný bod aplikácie. Okrem toho vyžíva pomocnú triedu `CoreUtilities`, ktorá okrem iného obsahuje zoznam mien kontrolovaných súborov.

Kontrola plagiarizmu je vykonávaná vo vláknach, pričom každá dvojica súborov je porovnávaná v samostatnom vlákne. Toto vlákno je tvorené triedou `ParserManager`, ktorá taktiež zabezpečuje vykonanie predspracovania súboru a samotného porovnávania súborov. Dvojice sú vytvárané triedou odvodenou z triedy `CompareManager` na základe počtu súborov načítaných modulom nahrávania. Jadro obmedzuje maximálny počet súčasne bežiacich vlákien a zobrazuje priebežný stav kontroly formou grafického ukazovateľa priebehu.

Jadro obsahuje aj podporu pre jednoducho parsovateľný výpis výsledkov. Je z neho možné jednoduchým parsovaním určiť názvy kontrolovaných súborov, priebeh predspracovania, priebeh kontroly (výsledky porovnávania jednotlivých dvojíc súborov sú zobrazené hneď, ako je kontrola tejto dvojice ukončená). Tento spôsob výpisu je využívaný používateľským rozhraním.

7.1.1 CoreUtilities

Statická trieda `CoreUtilities` má za úlohu konvertovať názvy vstupných súborov, ako ich jadru poskytne modul rozhrania pre príkazový riadok, do zoznamu inštancií triedy QString. Táto trieda taktiež spravuje nastavenia, ktoré boli zadané z príkazového riadka. Okrem toho sa stará o farebný výstup textu pre jednotlivé platformy.

7.2 Modul rozhrania pre príkazový riadok

Modul rozhrania pre príkazový riadok (angl. Command-Line Interface, d'alej len CLI) umožňuje používateľovi špecifikovať vstupné parametre pre našu aplikáciu pomocou argumentov a prepínačov v príkazovom riadku. Pre jednoduché nastavenia aplikácie je postačujúce, a preto sme ho zvolili za základnú formu interakcie nášho systému s používateľom. Výstupné dáta poskytuje v jednoduchej podobe, čím zjednodušuje napr. aj možnosť ich ďalšieho spracovania. Toto je obzvlášť dôležité pre prácu s grafickým rozhraním, ktoré tvorí samostatnú aplikáciu.

CLI ako ostatné moduly využíva prostriedky Qt. Je implementované v samostatnej triede Cli. Skúsený programátori môžu mať otázku, prečo sme nevyužili priamočiaru a jednoduchú C funkciu `getopt`. V praxi sa však vyskytli problémy s kódovaním vstupných prepínačov (názvy súborov pre porovnanie sú jedným zo vstupných parametrov, pričom je nutné predpokladať názvy s diakritikou).

Trieda Cli má metódu `help()`, ktorej názov je samovysvetľujúci, zobrazuje pomocníka – použitie textovej aplikácie a všetky prepínače s jednoduchým popisom. Popis všetkých prepínačov, ktorý vypíše help je tu:

```
Usage: ccopypaste [-h] [-cfgprt] [-m <level>] method file1 file2 [file3 [...]]
      -c: files are considered to be source codes and are tokenized
      -f: compare only the first file to the others
          (do not consider all possible unordered pairs)
      -g: spartan easy-to-parse output
      -h: help
      -m <level>: merge files from directories starting at <level> depth
          where level is positive integer
          For example if we have directory 'project1' and directory 'project2'
          and we want to compare this projects, the option would be '-m 0'.
          If we have directory 'projects', which contains directories 'project<1,100
          and we want to compare projects 1-100, the option would be '-m 1'.
      -p: compare all possible ordered pairs
          (-f and -p cannot be used together at the same time)
      -r: recursive
      -t: files are considered to be Slovak texts and are preprocessed
          (-c and -t cannot be used together at the same time)
      -v: export side-by-side visualization data
          (only with -c and GST method)
      method: method to compare the files
              supported methods:
                  3G - 3-grams
                  GST - Greedy String Tiling
                  SB - String-blurring
                  LCS - Longest Common Substring
```

```
                LSA - Latent Semantic Analysis
    file1, file2, ...: files to be compared
```

Hlavnou metódou Cli je `CommandLineInterface(int argc, QStringList argv)`. Táto sa stará o spracovanie všetkých prepínačov, ktoré sa nachádzajú v QStringListe argv. Výhodou tohto prístupu je už spomínaná možnosť použitia akéhokoľvek kódovania pre prepínače.

Po spracovaní prepínačov ostáva v argv už iba zoznam súborov, resp. adresárov, ktoré je potrebné spracovať. Všetky údaje o požiadavkách používateľa, vrátane smerníka na pole súborov, resp. adresárov na spracovanie, funkcia umiestni do triedy CoreUtilities, využívajúc jej metódy pre nastavenie zapuzdrených premenných. Z triedy CoreUtilities sú potom prístupné pre všetky relevantné moduly aplikácie. Úsek pre CLI dôležitého kódu z deklarácie triedy CoreUtilities:

```
class CoreUtilities
{
private:
static QStringList files;
/** True if file is going to be tokenized as source code*/
static bool tokenize;
/** True if file is going to be preprocessed as Slovak text*/
static bool preprocessLanguage;
/** Recursive search*/
static bool recursive;
/** Strict output*/
static bool strict;
/** Export visualization data*/
static bool exportVisualizationData;
/** Holds ID of compare method (3-grams, GST, etc.)*/
static short compareMethod;
/** Holds ID of compare management model (ALLCOUPLES or FIRSTTOOTHERS)*/
static short compareManagementModel;
/** > -1 if files are going to be merged */
static int level;
static bool mergeFiles;
public:
static void setRecursive(bool recursive);
static bool getRecursive();
static void setTokenize(bool tokenize);
static bool getTokenize();
static void setCompareMethod(int compareMethod);
static int getCompareMethod();
static void setCompareManagementModel(short cm);
```

```
static short getCompareManagementModel();
static void setPreprocessLanguage(bool preprocessLanguage);
static bool getPreprocessLanguage();
static void setStrict(bool strict);
static bool getStrict();
static void setExportVisualizationData(bool exportVisualizationData);
static bool getExportVisualizationData();
static void setLevel(int level);
static int getLevel();
static bool getMerge();
```

QStringList files určuje súbory alebo adresáre na vstupe, ich počet je prístupný z dátovej štruktúry QStringList. Príznaky tokenize, resp. preprocessLanguage udávajú, či sa majú vstupné údaje chápať ako zdrojové kódy alebo slovenské texty. Príznak recursive určuje, či sa majú adresáre na vstupe rekurzívne prehľadať. Príznak strict určuje jednoduchý výstup aplikácie pre ďalšie parsovanie. Príznak exportVisualizationData určuje, či sa majú do dočasného adresára zapisovať dáta pre vizualizáciu podobnosti súborov. Hodnoty compareMethod a compareManagementModel udávajú číslo porovnávacej metódy, resp. modelu tvorby dvojíc súborov, ktoré má aplikácia použiť.

7.3 Modul nahrávania

Modul nahrávania súborov má za účel nahrať súbory z disku do pamäte. V prvom kroku vytvorí kópie všetkých kontrolovaných súborov v dočasnom priečinku (v prípade textových súborov), respektíve súbory skonvertované do formátu txt. Každý z týchto súborov je neskôr načítaný do pamäte ako QString.Obvyklým formátom pre odovzdávanie dokumentácií a rôznych iných dokumentov sú súbory typu doc a pdf. Tieto treba najprv skonvertovať do formátu txt, aby mohli byť načítané do QStringu a súčasne bol zabezpečený jednotný prístup ku všektým typom súborov.

7.3.1 Konverzia pdf do txt

Kvôli požiadavke multiplatformovosti riešenia bola od počiatku preferovaná cesta použitia externej open-source aplikácie na konverziu pdf do txt súborov. Po istom hľadaní bol zvolený program *pdftotext*, ktorý je štandardnou súčasťou knižnice Poppler [15]. Výhodou je, že má predkompilovanú verziu na operačný systém Windows. Na systémoch typu Unix býva poppler obvyklou súčasťou predinštalovaného softvéru.

Implementácia konverzie z pdf do txt teda využíva spomenutý program pdftotext. Ten akceptuje 2 vstupné argumenty - vstupný súbor a výstupný súbor. Volanie externej aplikácie pdftotext je realizované v triede `PdfConverter` pomocou nasledovného kódu:

```
int PdfConverter::doConvert(QString input, QString output)
{
```

```
QStringList arguments;

// create arg to call external app
arguments << input << output;

// actualy call external app pdftotext
return QProcess::execute("pdftotext", arguments);
}
```

Input je vstupný súbor, Output je výstupný súbor, pričom tieto premenné dostane metóda ako svoje vstupné parametre. Pre spustenie externej aplikácie sa používa metóda `QProcess::execute`, ktorej parametrami je cesta k externej aplikácii a argumenty v objekte QStringList. Metóda vráti exit kód spusteného procesu.

7.3.2 Konverzia doc do txt

Konverzia doc do txt zatiaľ bol implementovaná pre rôzne platformy odlišne. Na to sme využili v Qt dostupné možnosti `#ifdef Q_OS_WIN32` a `#ifdef Q_OS_UNIX`.

V OS Windows sme pre konverziu zvolili aplikáciu Abiword, ktorá pomocou prepínača `--to=txt` skonvertuje zadaný doc súbor do textového súboru s rovnakým menom a v rovnakom adresári ako pôvodný doc. Tento textový súbor je potom ešte prenesený do pracovného adresára aplikácie.

V OS typu Unix sa o konverziu stará Openoffice [17] v spolupráci so skriptom `JODConverter` [16]. Openoffice beží na pozadí a na základe skriptu, ktorého vstupnými parametrami sú vstupný a výstupný súbor získame textový súbor z dokumentu typu doc.

7.3.3 Manipulácia so zip archívmi

Na extrakciu zip archívov bola zvolená multiplatformová aplikácia `unzip`. Unzip sa spúšťa s argumentami `unzip <zip súbor> -d <výstupný adresár>`. Výstupný adresár je štandardne cesta 'temp/zip súbor', všetky súbory zo zip archívu zadania.zip sa rozbalia do adresára 'temp/zadania.zip/'. V tomto adresári sa ešte vykoná usporiadanie všetkých textových a zdrojových súborov podľa veľkosti a ich spojenie do jedného výsledného súboru.

7.3.4 Integrácia s LMS Moodle

Integráciu so systémom moodle pri nutnosti sťahovania odovzdaných zadaní zo LMS moodle je možné vykonať nasledujúcimi spôsobmi.

- Úpravou php skriptov systému moodle
- Vytvorením vlastného skriptu s priamym prístupom na disk

Úprava kódu systému moodle

Pri tejto úprave je potrebné si naštudovať jednotlivé funkcie implementované v skriptoch . Prezeraním jednotlivých možností systému som našiel stránku submissions.php, ktorá vykonávala všetky nami požadované funkcie. Boli zobrazené jednotlivé súbory v danom mieste odovzdania. Po malej úprave v php skripte sa skutočne zobrazovali všetky odovzdané súbory a to aj už neaktívnych používateľov. Okrem iného sa zobrazovalo aj množstvo iného kódu. Zbytočný kód nebol odstránený úplne predovšetkým kvôli značnej previazanosti jednotlivých modulov v moodle a nutnosti zásahov do veľkého počtu týchto modulov. Stále však považujem toto riešenie za funkčné avšak nie ideálne keďže pri spúšťaní samotného skriptu by boli vykonané a zobrazené aj nepotrebné dáta respektíve úlohy. Tiež toto riešenie prináša nutnosť veľkých zmien v systéme.

Pri tejto úprave nie je potrebné riešiť prístupové práva, keďže prístupnosť jednotlivých dokumentov rieši samotný moodle podľa prihláseného používateľa.

Priamy prístup do file systém-u

Pri tomto prístupe nie je potrebné oboznámenie sa so samotným kódom systému moodle. Je však potrebné pochopiť vzťahy v databáze a význam niektorých tabuliek. Samotná cesta k súborovom systéme je pomerne zložitá, tvoria ju pod adresáre ktorých názov je identifikátor kurzu, miesta odovzdania či samotného študenta. Tieto identifikátory je potrebné nájsť v tabuľke. Vzťahy v tabuľkách sú nasledovné.

- Assignment - zoznam miest odovzdaní v jednotlivých kurzoch s ich
- AssignmentSubmissions - jednotlivé odovzdania študentov v konkrétnych miestach odovzdaní
- Users - zoznam študentov

Cesta k samotným súborom v súborovom systéme je takáto:
datafolder/idKurz/moddata/assigment/idMiestaOdovzdania/idPouzivatela/subor
Je teda zrejmé, že pomocou databázy a prístupu k filesystému je možné získať všetky odovzdané súbory a to nasledovne. Pre každú položku v tabuľke assignmentSubmissions vytvoríme jednu adresu v súborovom systéme podľa predošlej schémy. Id používateľa a miesta odovzdania získame z tejto tabuľky. Z tabuľky assignment získame id kurzu na základe id miesta odovzdania. Z tabuľky users získame meno a priezvisko pre spätnú identifikáciu používateľa na základe jeho ID.

Uvedené operácie som rozdelil do troch skriptov.

- Download.php s get parametrom miesta odovzdania - stiahne všetky súbory v danom mieste odovzdania a ponúkne ich používateľovi na stiahnutie v jednom zip súbore bez vytvárania dočasných súborov na systéme kde beží moodle.
- List.php - vráti textový súbor so zoznamom jednotlivých miesto odovzdaní a ich Id v konkrétnych kurzoch
- Users.php s get parametrom miesta odovzdania - vráti id spolu s menami všetkých používateľov, ktorý niečo vložili do daného miesta odovzdania

Pri tomto prístupe treba zabezpečit' obmedzený prístup k súborom pomocou hesla alebo pomocou blokovania IP adries kvôli možnému zneužitiu poskytnutých údajov. Keďže pristupujeme priamo do systému a nepoužívame overenie systému moodle sú tieto skripty bez dodatočnej ochrany verejne prístupné. Zabezpečenie bude implementované vo forme http access.

Výber prístupu
Predovšetkým kvôli značne menším nárokom na potrebné zásahy do systému moodle , kvôli jednoduchšej údržbe vytvoreného modulu a tiež vďaka celkovej jednoduchosti a priamočiarosti riešenia bol zvolený priamy prístup do file systém-u za pomoci databázy. Riešenie sa ukázalo ako funkčné a relatívne rýchle.

7.3.5 Spájanie súborov

Spájanie súborov zabezpečuje trieda `MergeFiles`. Jej úlohou je pospájat' v danom priečinku všetky súbory s určitou príponou do jedného súboru. Tieto súbory sú spájané podľa veľkosti. Spájanie súborov je zabezpečené pomocou 2 metód.

```
QList<QFileInfo> getFileInfoList(QString path, QList<QString> extensions);
```

Táto metóda rekurzívne prechádza zvolený adreasár a ukladá si informácie o súboroch, ktorých prípona sa nachádza v zozname prípon. Tento zoznam je argumentom funkcie. Funkcia nakoniec vráti informácie o všetkých súboroch, ktoré sa jej podarilo v adresári nájst'.

```
void mergeFiles(QList<QFileInfo> list, QString fileOutputPath);
```

Metóda `mergeFiles` je zodpovedná sa samotné spájanie súborov. Jej prvým argumentom je zoznam súborov, ktorý vracia metóda `getFileInfoList` a cesta k súboru, do ktorého budú následne spojené všetky súbory. Ešte pred samotným spojením, sa tieto súbory usporiadajú podľa veľkosti.

7.4 Modul manažmentu porovnávania

Implementácia bola vykonaná implementovaním triedy Compare manager, ktorá slúži na vytváranie kombinácii súborov. On sám nepracuje so súbormi, iba s ret'azcami, ktoré predstavujú cesty k týmto súborom. Podľa druhu kombinátora (combinera) následne vytvára kombinácia. Ponúka nasledujúce základné funkcie:

- Nastavenie a vrátenie zoznamu ciest k súborom
- Vrátenie nasledujúcej kombinácie
- Reset počítadla na nulu
- Vrátenie celkového počtu kombinácii

7.4.1 Trieda Pair Combiner

Vytvára kombinácie každého s každým bez opakovania, podľa poradia ako ich dostal. Teda ak dostane na vstup súbory s názvami 1, 2 a 3. Vytvorí kombinácie 1-2, 1-3 a 2-3. Všetkých kombinácii je teda n*(n-1)/2. Toto je výhodne použiť pri symetrických metódach, ktorých výsledky nezávisia od toho či sa porovnáva prvý súbor s druhým, alebo opačne.

Kombinácie sa nevytvoria na začiatku všetky z dôvodu spomaľovania chodu programu a zbytočnému hlteniu pamäte pri väčšom množstve súborov. Kombinovanie prebieha ako keby v dvoch vnorených cykloch, pričom vnútorný začína vždy na hodnote vonkajšieho a nie na 1. Trieda si pamätá hodnoty iterácii a podľa toho vie vždy vytvoriť a vrátiť nasledujúcu iteráciu podľa potreby (simuluje vnorene for cykly). Na konci sa vráti na začiatok. Nevýhodou je, že nevie v ľubovolnom čase vrátiť i-tu iteráciu, čo ale v programe nevyužívame.

7.4.2 Trieda All to All Combiner

Tento kombinátor vytvára už kombinácie každého s každým s opakovaním. Vytvára všetky kombinácie predošlého kombinátora a ich zrkadlové obrazy. Implementácia taktiež prebieha ako keby v dvoch vnorených cykloch s tým, že vnútorný začína na hodnote 1. Kombinácií je dvakrát viac, a teda n*(n-1). Túto metódu využívame pri nesymetrických metódach ako napríklad n-gramy.

7.4.3 Trieda One to All Combiner

Vytvára kombinácie prvého reťazca cesty súboru so všetkými ostatnými. Teda celkových kombinácii je n-1. Túto metódu kombinovania budeme využívať ak je potrebné otestovať iba jeden súbor oproti ostatným a netreba ostatné kontrolovať vzájomne.

Implementácia tohto kombinátora bola triviálna. Trieda si pamätá jedno číslo, ktoré je na začiatku 2 (kombinácia 1-2). Toto číslo zvyšuje až po n (kombinácia 1-n) a potom sa vráti späť na 2.

7.5 Modul manažmentu analýzy

7.5.1 Jadro modulu manažmentu analýzy

Modul manažmentu analýzy tvorí trieda `ParserManager`, ktorá rozširuje triedu `QThread`. Inštancie tejto triedy sa spúšťajú v samostatných vláknach. Vstupom pre triedu `ParserManager` je dvojica textových reťazcov, ktoré boli načítané z kontrolovaných súborov. Na základe zvolenej porovnávacej metódy trieda najskôr tieto reťazce predspracuje s využitím funkcionality príslušnej triedy, a následne vytvorí inštanciu príslušného komparátora. Ten porovná predspracované reťazce a výsledok vráti triede `ParserManager`. Trieda `ParserManager` obsahuje dve statické premenné - `count` a `threadCount`.

Po ukončení porovnávania, a teda tesne pred ukončením príslušného vlákna, inštancia triedy obe tieto premenné dekrementuje. Premenné určujú počet zatiaľ neukončených vlákien (premenná `count`), a počet momentálne spustených vlákien (premenná `threadCount`). Pomocou týchto premenných riadi jadro aplikácie maximálny počet súčasne spustených vlákien a určuje, či sú už pripravené všetky výsledky.

7.5.2 Predspracovanie textu

Implementácia vyhadzovania stop-slov

Pred samotným porovnaním súborov je potrebné tieto súbory upraviť. Táto úprava prinesie časovú aj pamäťovú úsporu. Stop slová v slovenskom jazyku sú slová, ktoré nenesú samé o sebe žiaden význam. Jedná sa najmä o predložky ,zámená a spojky. Nie sú teda pre text kľúčové a ich odstránením text nestratí význam. Odstránením týchto slov sa stáva porovnávanie značne rýchlejšie, nakoľko spracovaný text obsahuje menej slov.

Práve toto je úlohou algoritmu, ktorý tieto slová odstraňuje. Metóda, ktorá ich odstraňuje, potrebuje text, ktorý má spracovať a zoznam stop slov, ktoré budú počas spracovania odstránené. Tieto slová sú uložené v externom súbore, ktorý sa načíta pomocou Loadera. Ten súbor prečíta, a všetky slová predá metóde vo forme listu.

```
QList<QString> stopWords
```

Metóda následne prejde všetky prvky tohoto listu, a ak sa daný prvok nachádza v texte, je zmazaný. Takto upravený text napokon funkcia vráti.

Implementácia lematizácie

Nutnou podmienkou pre vykonanie lematizácie slovenských textov (opísaná v časti 2.1.1) je databáza všetkých tvarov slov a ich tvar v základnom tvare. Táto databáza bola prebratá zo Slovenskej akadémie vied vo formáte CDB.

Ďalšou podmienkou je nájsť aplikačné rozhranie(API), ktoré nám umožní túto databázu čítať. V tejto časti implementácie nastal závažný problém. Rozhranie API pre jazyk C++ sa mi v rozumnom čase nepodarilo nájsť. Našiel som však rozhranie API pre CDB++[1] databázy. Táto implementácia databázy obsahuje všetky kľúčové výhody CDB databázy súvisiace s rýchlosťou prehľadávania. Nevýhodou však je, že nepodporuje spätnú kompatibilitu s CDB databázou a tiež nevie riešiť kolízie. (tabuľka musí byť jednoznačná, jednej hodnote X prislúcha len jedna hodnota Y). Napriek uvedenému je pre naše účely CDB++ databáza vhodná a preto som pristúpil k tomuto alternatívnemu riešeniu.

Je potrebné konvertovať databázu vo formáte CDB do CDB++. Použil som aplikačné rozhranie CDB databázy v Jave a následne som vygeneroval všetky dáta do textových súborov. Pomocou vytvorených textových súborov som vytvoril novú CDB++ databázu, ktorú následne môžeme použiť v našej C++ implementácii.
Pre použitie databázy v CDB++ sú potrebné dva kroky:

[1] http://www.chokkan.org/software/cdbpp/

1. **Otvorenie databázy** `cdbpp::open`

2. **Dopyt na databázu** `cdbpp::get`

Vytvorená databáza je formátovaná v kódovaní UTF8. Preto treba zabezpečiť aby aj dopyt bol v kódovaní UTF8. Inak nastanú problémy pri hľadaní slov s diakritikou, pre ktoré ich základný tvar nebude nájdený.

V tejto chvíli bolo možné získať základný tvar ľubovoľného slova. Vstupom metódy na lematizáciu je však QString. Preto som vytvoril jednoduchý cyklus, ktorý zistí základný tvar slova a pridá ho do nového QString-u. V prípade že slovo nebolo nájdené, použije sa pôvodné slovo v dopyte.
Spomenutý cyklus pozostáva z týchto krokov

1. Získaj prvé slovo z QString-u a vymaž ho z pôvodného QString-u

2. Získaj základný tvar slova

3. Pridaj nové slovo do nového QString-u

7.5.3 Tokenizér zdrojového kódu

Porovnávaniu zdrojových kódov predchádza predspracovanie vstupných súborov – tokenizácia. Pri nej je vytvorená z každého zdrojového kódu postupnosť symbolov (tokenov), ktorá predstavuje logický obsah tokenizovaného súboru. Jednotlivé tokeny sú reprezentované celočíselnými hodnotami.

Predspracovanie zdrojových kódov prebieha vo viacerých krokoch. Samotná tokenizácia sa vykonáva nad reťazcami, ktoré sú do istej miery modifikovanými verziami pôvodných vstupných súborov. Toto predspracovanie pred tokenizáciou prebieha nasledovne:

1. odstránenie escapovaných zlomov riadku (znak nasledovaný zlomom riadku je nahradený medzerou, pričom zlom riadku je odstránený)

2. odstránenie komentárov a textových reťazcov – konštánt

3. nahradenie definícií `(#define)` v zdrojovom kóde ich významom

Z upraveného zdrojového kódu je následne vytvorený strom, ktorého vrcholy sú tvorené logickými blokmi v zdrojovom kóde. Hrany stromu sú vytvorené vložením špeciálnej riadiacej konštrukcie (BLOCK) priamo do tokenizovaného zdrojového kódu. Táto konštrukcia odkazuje na blok (jeho identifikačným číslom), ktorý sa na tomto mieste v pôvodnom zdrojovom kóde nachádzal.

Po vytvorení stromu sú všetky vrcholy (ktoré už neobsahujú žiadne riadiace štruktúry, iba odkazy na iné bloky) tokenizované – pre každý vrchol je vytvorený zoznam tokenov, ktoré reprezentujú jednotlivé výrazy v tomto bloku (výrazy sú obvykle oddelené bodkočiarkou). Špeciálnym tokenom je token __block ktorý hovorí, že ten-ktorý výraz je v skutočnosti odkazom na iný blok. Zoznam ostatných používaných tokenov sa oproti návrhu nezmenil.

Posledným krokom tokenizácie je vytvorenie postupnosti tokenov zo stromu reprezentujúceho zdrojový kód. Začína sa pritom posledným vrcholom stromu, ktorý reprezentuje koreňový prvok zdrojového kódu – celý dokument. Ostatné bloky sú v ňom vnorené (vnorenia je možné nájsť vďaka tokenom __block), a preto sa výsledná postupnosť tokenov vytvára rekurzívnym prechádzaním jednotlivých blokov začínajúc práve tým posledným. Keďže všetky bloky sú už tokenizované, tokeny jednotlivých blokov sú usporiadávané do jednej súvislej postupnosti. Výsledkom je tokenizovaný zdrojový kód, číselná postupnosť zapísaná do dátového typu `QString`.

Tokenizácia zdrojového kódu v jazyku Java

Jazyk Java má iný charakter zdrojového kódu, preto tokenizácia týchto zdrojových súborov je čiastočne odlišná. Pri predspracovaní kódu pred samotnou tokenizáciou sa nevykonáva odstraňovanie znaku a nahradzovanie výrazov `#define`. Miesto toho sú zo zdrojového kódu odstránené výrazy import aj s názvami jednotlivých tried. Samotná tokenizácia prebieha podobným algoritmom a používa rovnakú sadu tokenov, z dôvodu aby boli nebolo jednoduché vytvárať plagiáty prepisom z jazyka C do Java a opačne.

Tokenizácia sa líši hlavne z dôvodu nasledujúcich špecifických charakteristík jazyka Java:

1. používa objekty, ktoré sa vytvárajú príkazom new
2. pri vytvorení takýchto objektov sa niekedy taktiež používa priamo deklarácia niektorých ich funkcií
3. obsahuje štruktúru zoznam (list), ktorý sa deklaruje pomocou symbolov < a >
4. nepoužíva znaky * a & ako v jazyku C
5. používa špeciálne slová private, public, protected, volatile, synchronized a final
6. má odlišnou sadu základných druhov premenných a systémových, alebo výpočtových funkcií
7. používa bloky deklarované pomocou špeciálnych slov try, catch a finally pre odchytávanie výnimiek

Pri vytváraní objektov sa konštruktor berie ako volanie funkcie, pričom sa vyhodnocuje aj jeho vstup. Ak za konštruktorom priamo nasleduje blok s kódom (definovanie funkcií objektu a podobne), je tento kód taktiež tokenizovaný. Taktiež je ošetrený prípad ak vstupom funkcie sú takéto objekty. Tokenizuje sa celý vstup funkcie. Zoznamy a premenné private, public atď sú tokenizované ako bežné premenné. Taktiež bloky try a finally sa vyhodnocujú ako normálne bloky, pričom bloky catch sú zo zdrojového kódu odstránené pri predspracovaní.

7.5.4 Komparátor pre metódu N-gramy

Metódu N-gramov opísanú v časti 2.3.1 som implementoval v jazyku C++. Vytvorená funkcia má na vstupe dva QString-y a jej výstup predstavuje podobnosť vstupov v intervale <0,1>. Tvorí ju cyklus, ktorý získava všetky za sebou nasledujúce N-tice z jedného QString-u. Následne sa skontroluje výskyt každej vybranej N-tice v druhom QString-u. Výsledná hodnota, ktorú funkcia vracia, je pomer úspešných výskytov N-tice v druhom QString-u s počtom všetkých porovnaní.

7.5.5 Komparátor pre metódu LCS

Metódu LCS sme implementovali v triede ComparatorLCS. Táto trieda ma hlavnú funkciu compare(). Na vstup dostane dva reťazce predstavujúce text súboru a výstupom je dĺžka najdlhšieho reťazca v pomere ku vstupu. Keďže treba brať do úvahy dĺžky oboch reťazcov výsledok je 2*dlźka reťazca vydelená súčtom dĺžok vstupných reťazcov.

Funkcia je implementovaná metódou dynamického programovania. Pôvodne funkcia používala pomocné dvojrozmerné pole (tabuľku), ktorého jeden rozmer bol dĺžka prvého reťazca a druhý rozmer dĺžka druhého. Keďže súbory obsahujú väčšinou niekoľko tisíc znakov nebola táto implementácia vhodná. Tabuľka potrebovala príliš veľa pamäte a nastávala situácia s jej zahltením. Z toho dôvodu sme poopravili algoritmus.

Pôvodný algoritmus obsahoval dva vnorené for cykly v ktorých sa iteruje podľa dĺžok vstupných textov. Začína od prvého znaku v prvom texte až po posledný. Pre každý tento znak kontroluje znak v druhom texte. Vždy keď nájde zhodu medzi znakmi reťazca nastaví hodnotu v tomto políčku o jedno väčšie ako najväčšia predošlá hodnota (zistená porovnaním v predošlých iteráciách). Teda ak sa i-ty znak reťazca A zhoduje s j-tym znakom reťazca B treba túto zhodu treba zapísať do tabuľky. Funkcia zistí a nastaví hodnotu políčka tabuľky [i][j] o jedno väčšiu ako hodnotu na políčku [i-1][j]. Pritom kontroluje či táto hodnota nie je väčšia ako zatiaľ nájdená dĺžka spoločného podreťazca, ktorú na konci vráti v pomere k dĺžke textov.

Keďže algoritmus v každej i-tej iterácií pracuje iba so stĺpcami tabuľky s číslami i a i-1, pre riešenie je postačujúca tabuľka s iba dvoma stĺpcami. Tým sa výrazne znížili pamäťové nároky. V prvom stĺpci ma uložené hodnoty z predošlej iterácie a v druhom stĺpci hodnoty aktuálnej iterácie. Na druhej strane sa trocha zvýšila časová zložitosť algoritmu, pretože v každej iterácií je nutné si prekopírovať hodnoty z jedného stĺpca do druhého a potom pôvodné hodnoty vynulovať.

7.5.6 Komparátor pre metódu Greedy String Tiling

Úvahy o efektívnosti procesu porovnávania nás priviedli k názoru, že postupnosť tokenov bude vhodnejšie nereprezentovať ako rozsiahly reťazec znakov, ale ako vektor prirodzených čísel. Porovnanie dvoch čísel je podstatne menej náročnou operáciou ako porovnanie dvoch reťazcov znakov, čím sa znižuje výpočtová náročnosť procesu porovnávania celých súborov. Pre Greedy String Tiling sme preto zaviedli vo všeobecnom komparátore separátnu

virtuálnu metódu. Samotná porovnávacia metóda má nasledujúcu deklaráciu.

```
double ComparatorGst::compare
(QList<int> tokensA, QList<int> tokensB)
```

Algoritmus teda dostáva na vstupe súbory už vo forme postupností prirodzených čísel, pričom každé číslo predstavuje jeden token. Greedy String Tiling, ako sme už predtým spomenuli, cyklicky iteruje nad postupnosťou tokenov a označuje zhodné oblasti v súboroch pomocou trojice (a,b,j), kde a, resp. b sú začiatočné pozície zhôd v skúmaných postupnostiach a hodnota j vyjadruje dĺžku podozrivej oblasti. V každej iterácii je potrebné uchovávať množinu nájdených zhôd, z ktorých sa do množiny tzv. dlaždíc presúva vždy len najdlhšia zhoda pre danú iteráciu (resp. viacero zhôd, pokiaľ sa ich našiel väčší počet). Ak sa žiadna dostatočne dlhá zhoda nenájde, algoritmus končí.

Greedy String Tiling si teda vyžaduje implementáciu množiny trojíc (a,b,j), pri ktorej je nutné uvažovať o 3 dôležitých operáciách.

1. vyprázdenie množiny
2. vloženie konkrétnej trojice do množiny (pri nájdení zhody, resp. jej presunutí medzi dlaždice)
3. vyhľadanie najdlhšej nájdenej oblasti (t.j. rýchly prístup do množiny, ak je vopred známa hodnota j)

Pre implementáciu sme teda, najmä kvôli poslednej požiadavke, použili triedu QMap, ktorá umožňuje využívať hodnotu j ako kľúč. Každý element QMap je potom QVector dvojíc (a,b), ktoré sa v množine nachádzajú „v kombinácii" s danou hodnotou j.

```
typedef struct MATCH
{
int a;
int b;
} match;

QMap<int, QVector<match> > matches;
QMap<int, QVector<match> > tiles;
```

Okrem diskutovaných množín si Greedy String Tiling vyžaduje aj možnosť značkovania jednotlivých tokenov. Úplne postačujúcim riešením je v tomto prípade bitový vektor, ktorý pre každý token uchováva informáciu, či je označený, preto sme použili triedu QVector.

```
QVector<bool> markedTokensA;
QVector<bool> markedTokensB;
```

Samotný výpočet prebieha tak, ako sme už uviedli v pseudokóde (pozri kapitolu „metóda Greedy String Tiling"). Výstupom metódy je pritom reálne číslo, ktoré vyjadruje podiel medzi počtom tokenov patriacich do podozrivých oblastí a počtom všetkých tokenov v obidvoch skúmaných súboroch.

```
(double) tileLengthsSum / (double) tokenizedCodeLength
```

7.5.7 Komparátor pre algoritmus String-blurring

String-blurring je metóda hľadania podobnosti dvoch textových reťazcov pracujúca nad jednotlivými znakmi. Porovnávanie reťazcov metódou string-blurring prebieha v troch fázach. V prvej fáze sú zo vstupných textových reťazcov vytvorené polia celých čísiel, ktoré sú tvorené unicode kódmi alfanumerických znakov daných reťazcov. Všetky biele znaky, interpunkcia, ako aj iné nealfanumerické znaky sú pri hľadaní podobnosti ignorované. Okrem toho, písmená vstupných reťazcov sú konvertované na malé.

V druhej fáze sú tieto dve polia celých čísiel rozmazané - sú vytvorené nové polia čísiel také, že každý ich prvok je rovný váženému priemeru `n` okolitých prvkov pôvodného poľa (hodnota `n` je jedným z parametrov porovnávania). Váhy sa pritom počítajú podľa gaussovej funkcie. Výsledkom sú rozmazané reťazce (polia) čísel.

Samotné určovanie podobnosti reťazcov sa uskutočňuje v poslednej, tretej fáze. V rozmazaných reťazcoch (poliach celých čísel) sa hľadajú dva podreťazce rovnakej dĺžky (v každom vstupnom reťazci jeden) také, že rozdiel ich prvkov s rovnakým indexom je menší ako určitá zadaná hodnota. Podobnosť pôvodných reťazcov závisí od najväčšej možnej dĺžky takýchto podreťazcov. Výsledkom porovnávania je podiel dĺžky najdlhšieho nájdeného podreťazca k dĺžke kratšieho z oboch rozmazaných reťazcov.

7.6 Komparátor pre metódu LSA

Ako sme už spomenuli v analýze problému, latentná sémantická analýza je multidimenzionálna metóda, ktorá sa od ostatných odlišuje tým, že modeluje korpus ako maticu dokumentov, ktorú následne upravuje prostredníctvom aparátu, ktorý nám poskytuje lineárna algebra.

LSA sme implementovali pomocou komparátora, ktorý sa líši od ostatných. Vzhľadom k tomu, že LSA nepoužíva rovnaké mechanizmy, ako ostatné metódy, v toku riadenia programu má v podstate samostatnú vetvu, ktorej základ predstavuje využívanie služieb triedy `ComparatorLSA`.

LSA je zložená z dvoch fáz - extrakcie fráz a spracovania matice dokumentov. Fráza je v podstate N-gram, ktorý sa nachádza aspoň v jednom zo skúmaných dokumentov. O fráze si potrebujeme pamätať aj metadáta, preto údaje, ktorými ju reprezentujeme, sú nasledovné:

```
QString phrase;
int documentCount;
```

```
QVector<int> occursInDocuments;
QVector<int> numberOfOccurences;
```

Poznáme teda samotnú frázu, počet dokumentov, v ktorých sa vyskytuje, ich identifikátory a počty jednotlivých výskytov. Tieto dáta máme uložené v špecificky upravenej hashovacej tabuľke, kde kľúč predstavuje hash frázy. Operátory tabuľky sú `isin`, ktorá vráti true ak sa v tabuľke fráza nachádza, `update`, ktorá vykoná aktualizáciu metadát o fráze a operácie `insert`, resp. `clear`, ktoré vložia frázu do tabuľky, resp. tabuľku vyprázdnia.

Extrakcia fráz prebieha cez metódu `extractPhrases`, ktorej vstupom je predmetný súbor vo formáte QString. Metódu je potrebné zavolať na každý skúmaný súbor. Počet extrahovaných fráz aj spracovaných dokumentov si trieda `ComparatorLSA` pamätá sama.

Nad tabuľkou fráz vykonáme filtráciu - odstránime frázy, ktoré sa nachádzajú iba v jednom dokumente a frázy, ktoré sa nachádzajú vo veľmi veľa dokumentoch. Potom skonštruujeme maticu dokumentov, podľa vzťahu, ktorý sme si už uviedli v kapitole o algoritmoch detekcie plagiarizmu.

Pre dekompozíciu matice dokumentov používame knižnicu GNU GSL. Maticu definujeme a alokujeme takto:

```
gsl_matrix *documentMatrix = gsl_matrix_alloc(amountOfPhrases,corpusSize);
```

Dekompozíciu SVD vykonáme vďaka GSL len jediným príkazom:

```
gsl_linalg_SV_decomp(documentMatrix,matrixV,vectorS,tempVector);
```

Ostatné operácie sú implementované triviálne. Metóda vráti referenciu na maticu podobnosti dokumentov. Priestor si alokuje sama.

Kapitola 8

Testovanie

Testovanie vytvoreného nástroja bude prebiehať v nasledujúcich fázach

- Analýza požiadaviek
- Príprava testovacích stratégii
- Vykonávanie testov
- Reportovania chýb

8.1 Analýza požiadaviek

V tejto časti sa zaoberáme zozbieraním informácii o špecifikácii softvéru. Je potrebné zistiť ako sa očakáva, že softvér sa bude správať v jednotlivých situáciach. Taktiež je vhodné zistiť už známe chyby prípadne špeciálne vlastnosti softvéru.

8.2 Príprava testov

V procese prípravy testov definujeme prístup k jednotlivým testom. Pri testovaní budeme používať tieto metódy v poradí v akom sú spomenuté:

- prehliadka kódu
- testovanie základných jednotiek(unit)
- integračné testovanie
- systémové testovanie
- akceptačné testovanies

8.2.1 Prehliadka kódu (Walktrough)

Walktrough je neformálna prehliadka vytvoreného diela. V tomto projekte prehliadku používame predovšetkým na kontrolu zdrojových kódov a vytvorenej dokumentácie. Pri kontrole autor oboznamuje ostatných účastníkov tímu o svojom výsledku. Tí majú za úlohu oboznámiť sa s detailami diela a upozorniť na možné problémy a chyby.

8.2.2 Testovanie základných jednotiek(unit)

Týmto testovaním chceme zabezpečiť, aby individuálne časti kódu zodpovedali používateľským požiadavkám a želanej funkcionalite. Jednotkami testovania sú najmenšie testovateľné časti kódu aplikácie.

V prvom rade sa bude testovať aplikácia logicky podľa toho ako je objektovo vytvorená, teda podľa tried. Niektoré triedy budú ale výrazne dlhšie a budú obsahovať zložitejšie funkcie. Z toho dôvodu sa budú niektoré triedy testovať po menších častiach - po metódach (okrem tých triviálnych).

Testovanie po malých častiach zaručuje rýchlejšie odhalenie a lokalizáciu chýb v programe. Keďže triedy, resp. niektoré metódy sa budú kontrolovať samostatne, nebude ich chod ovplyvnený iným kódom. Pokiaľ test odhalí chybu, vieme hneď v ktorej triede, resp. metóde sa nachádza.

Testovanie budeme vykonávať zdola-nahor. Keďže niektoré metódy a triedy príliš závisia na iných triedach bolo by zbytočne zložité, alebo dokonca nemožné vykonávať ich testovanie samostatne. Preto sa najprv otestujú tie triedy a metódy, ktoré je možné otestovať samostatne. Potom sa otestujú tie, ktoré na nich závisia až kým sa neotestuje celá aplikácia.

Takéto testovanie je veľmi efektívne v odhaľovaní chýb a kontrole funkcionality. V praxi však môže byť zbytočne zdĺhavé. Je samozrejmé, že raz okontrolovaný kód sa nebude ďalej testovať, pokiaľ na ňom neboli vykonané zmeny. Výnimkou môže byt špeciálny prípad ak tento kód nevyhovuje inému, dodatočne implementovanému kódu, z objektového hľadiska. Taktiež ak napríklad vznikne chyba pri testovaní programov, je zbytočné kontrolovať triedy a metódy pre testovanie slovenských textov. Týmto sa docieli maximálna efektivita a úspešnosť testovania.

8.2.3 Integračné testovanie

Toto testovanie nasleduje zásadne po úspešnom otestovaní jednotlivých jednotiek zúčastnených na integračnom testovaní. Kontroluje sa predovšetkým korektná výmena dát medzi dvomi jednotkami. Ak je vstup jednej triedy závislý na výstupe inej triedy je dôležité aby vzájomné vstupy a výstupy boli vždy v očakávanom rozsahu hodnôt a nie mimo neho.

8.2.4 Systémové testovanie

V tomto type testujeme systém ako celok. Je nutné skontrolovať, či sa systém správa korektne vo všetkých prípadoch použitia nástroja podľa špecifikácie. Nie je nutné testovať všetky možné situácie, ktoré môžu nastať.

8.2.5 Akceptačné testovanie

Akceptačné testy sú záverečnou fázou testovania. Testujeme predovšetkým či nástroj zodpovedá špecifikácii. Ďalej je vhodné použiť aj iné ako očakávané vstupy a týmto spôsobom skontrolovať ošetrovanie nečakaných situácii.

8.3 Vykonávanie testov

Vykonávanie prebieha podľa plánu testovania. Na testovaní sa zúčastňujú výlučne osoby, ktoré nevytvárali samotný predmet testovania. Je totiž neprípustné aby testovanie mohlo byť ovplyvnené tvorcom, pretože by mohlo byť veľmi ľahko zmarené. Naopak je potrebné zachovať nezávislé a objektívne testovanie.

8.4 Reportovanie chýb

Pri výskyte a odhalení chyby testovaním nasleduje posledná fáza - reportovanie chýb. V tejto fáze je potrebné mať čo najpresnejšie informácie o povahe chyby. Sem patrí napríklad v akom kóde sa nachádza, kedy vzniká a aké sú vstupy a výstupy testovaných metód. Tieto údaje sa predajú implementátorovi tejto triedy (metódy), ktorý túto chybu následne opraví. Čím bolo testovanie dôkladnejšie, a teda čím reportovanie chýb ponúka presnejšie informácie implementátorovi o chybe, tým rýchlejšia a efektívnejšia bude jej oprava. Tým sa ušetrí čas implementátorovi, ktorý môže pracovať na inej práci, a taktiež ostatným implementátorom, ktorí čakajú na implementáciu tejto triedy (metódy). Našou snahou je tiež presným reportingom maximalizovať úspešnosť opravy chýb, čim sa minimalizuje výskyt ďalších chýb a potrieb ďalšieho testovania.

Samozrejmosťou je, že ak je chyba triviálneho charakteru je zbytočne zdĺhavé jej reportovanie, a preto ju opraví priamo tester. O výsledku potom informuje implementátora.

Kapitola 9

Experimentálne výsledky

V nasledujúcich riadkoch sa budeme venovať výsledkom, ktoré výsledok nášho úsilia dosahuje v predmetných oblastiach - detekcii plagiarizmu zdrojových kódov a slovenských textov. Preto sme rozdelili kapitolu na 2 časti.

V prvej časti sumarizujeme výsledky, ktoré nástroj Copypaste dosahuje na zdrojových kódoch vytvorených podľa našej metodiky a porovnávame ich s výsledkami riešení, ktoré sme predstavili v kapitole 3. Pre tento typ detekcie sa používa spravidla tokenizácia v kombinácii s komparačnou metódou Greedy String Tiling.

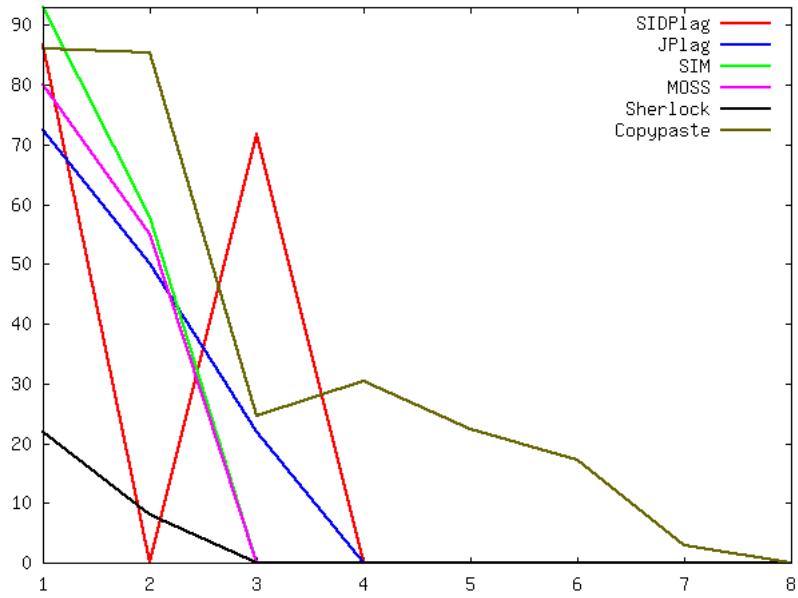
Druhá časť pokrýva oblasť slovenských textov, v ktorej využívame širšiu škálu komparačných metód, vrátane N-gramov, metódy najdlhšej spoločnej podpostupnosti (LCS), algoritmu String-blurring a v neposlednom rade aj multidimenzionálnej latentnej sémantickej analýzy. V tejto časti sa preto venujeme nielen porovnaniu s inými riešeniami, ale aj porovnaniu nami využitých komparačných metód. Spomenieme aj vplyv predspracovania textu na výsledky detekcie.

9.1 Analýza výsledkov v oblasti zdrojových kódov

Pri analýze našich experimentálnych výsledkov začneme s jazykom C. Na nasledujúcom obrázku sa nachádza graf, ktorý sme si predstavili už v rámci komparatívnej analýzy existujúcich riešení v kapitole 3, doplnený o výsledky detekcie naším nástrojom.

Priebeh krivky jednoznačne nasvedčuje, že naše riešenie predstavuje značné zlepšenie oproti doteraz vytvoreným riešeniam v oblasti detekcie plagiarizmu zdrojových kódov. Navyše, vo výsledkoch sa neprejavujú výraznejšie výkyvy (ako napr. u nástroja SIDPlag), ktoré by kvalitu riešenia spochybňovali. Naopak, relatívne stabilný, konkávny pokles úspešnosti detekcie s narastajúcou sofistikovanosťou plagiátorských techník (prerušený len jemným nárastom uprostred spektra) svedčí o relevancii stanovenej metodiky, ako aj vykonaných meraní.

Ďalší obrázok obsahuje výsledky, ktoré sme dosiahli v jazyku Java a ich porovnanie s ostatnými riešeniami. Krivka, ktorá charakterizuje tieto výsledky, vykazuje veľmi podobný priebeh, ako u nástroja JPlag, ktorý sme už v analýze označili za najlepší spomedzi analy-



Obr. 9.1: Porovnanie našich výsledkov s inými nástrojmi pre zdrojové kódy v jazyku C

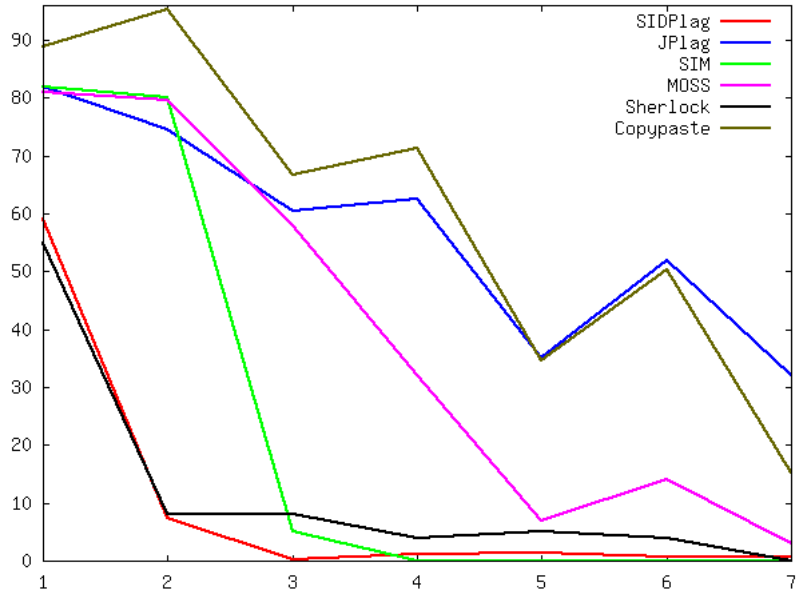
zovaných riešení. Uvedené meranie nasvedčuje tomu, že naše riešenie dosahuje minimálne také dobré výsledky, ako najlepšie nástroje, ktoré pre disktuovaný účel existujú.

9.2 Analýza výsledkov v oblasti slovenských textov

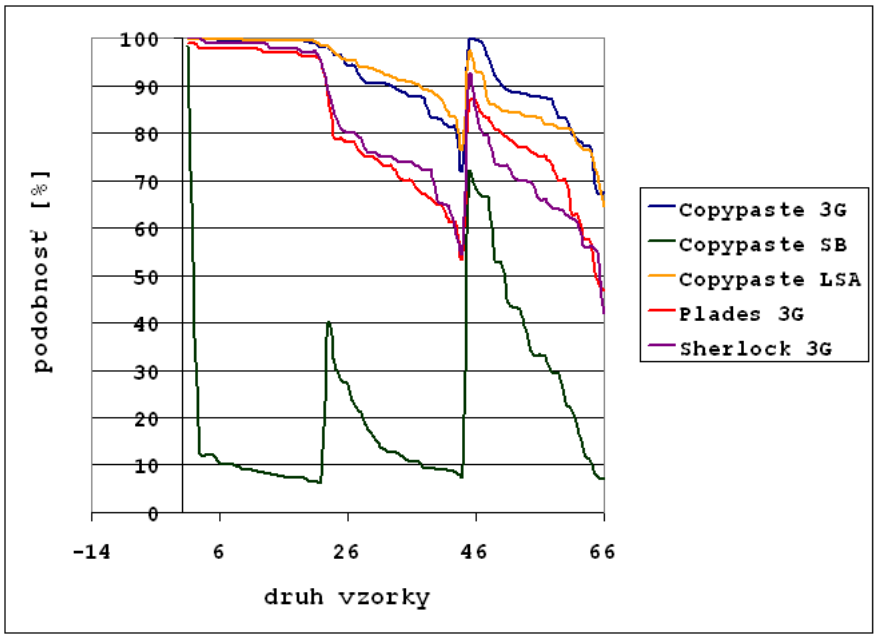
V rámci analýzy výsledkov v oblasti slovenských textov sme sa sústredili nielen na porovnanie s inými nástrojmi, ale aj na porovnanie našich metód. Pracovali sme s množinou 22 originálnych dokumentov, ku ktorým sme mali k dispozícii kopírované, modifikované, ako aj rešeršné plagiáty. Celkovo sme teda pracovali s korpusom 88 súborov, v ktorých sme sledovali detekciu plagiarizmu využívaním rôznych našich metód, ako aj metód, ktoré poskytujú nástroje PlaDes a Sherlock.

Na ďalšom obrázku sa nachádzajú výsledky tejto našej komparatívnej analýzy. Na x-ovej osi sa nachádzajú postupne hodnoty podobnosti originálov s kopírovanými plagiátmi (1-22), modifikovanými plagiátmi (23-44) a rešeršnými plagiátmi (45-66). V rámci každej kategórie sme výsledky kvôli prehľadnosti grafov utriedili od najväčšieho po najmenší.

Graf naznačuje, že naše metódy sú v prevahe oproti metódam konkurenčných nástrojov - vykazujú vyššie hodnoty detegovaného plagiarizmu. Najzaujímavejšie výsledky dosahujú metódy 3-gramy a LSA. Naša implementácia 3-gramov počíta vyššie hodnoty, ako implementácie konkurenčných nástrojov, pričom podobné čísla získavame aj metódou LSA. Zaujímavý jav predstavuje fakt, že výstup LSA je vyšší pri modifikovaných plagiátoch a výstup 3-gramov naopak pri rešeršných plagiátoch. Navhovaný algoritmus String-blurring



Obr. 9.2: Porovnanie našich výsledkov s inými nástrojmi pre zdrojové kódy v jazyku Java



Obr. 9.3: Porovnanie našich výsledkov s inými nástrojmi pre slovenské texty

Tabuľka 9.1: Porovnanie rýchlosti rôznych metód pre texty (časy v ms)

veľkost' vzorky	10	88	125
3-gramy	3,985	117,938	228,625
String-blurring	78,484	4885,5	-
LCS	455,609	-	-
LSA	-	164,125	-
Stochastická LSA (20%)	-	101,968	306,078
Stochastická LSA (5%)	-	48,703	121,734

v našej analýze veľmi neobstál.

V nasledujúcej tabuľke sa nachádza analýza časovej náročnosti jednotlivých metód v našom riešení. Z hľadiska efektívnosti sú najzaujímavejšími 3-gramy. LSA, ktorá poskytuje porovnateľne dobrú detekciu plagiarizmu, si však vyžaduje viac výpočtového času. Stochastické verzie LSA si však na testovaných dátach časovo počínali veľmi dobre. String-blurring vykazuje vyššiu výpočtovú náročnost', LCS je použiteľná iba na veľmi malé vzorky. Niektoré čísla nie sú v tabuľke uvedené z výpočtových dôvodov.

Kapitola 10

Zhodnotenie

V tejto dokumentácii sme sa zaoberali problematikou plagiarizmu a jeho kontroly prostredníctvom softvérových prostriedkov. Opísali sme zásadné aspekty rozoberanej oblasti a analyzovali sme možnosti, ktoré možno zohľadňovať pri kontrole plagiarizmu. Analyzovali sme existujúce metódy a algoritmy. Sústredili sme sa predovšetkým na bidimenzionálne metódy, ale spomenuli sme aj multidimenzionálnu metódu. Zamýšľali sme sa nad možnými alternatívnymi prístupmi. Značnú pozornosť sme venovali aj existujúcim nástrojom na kontrolu plagiarizmu, ktoré sme testovali na rovnakej množine údajov. Spomínaná analýza vyústila do komparácie jednotlivých nástrojov. Hľadali sme možné prístupy k vizualizácii výsledkov meraní v danej problematike nielen na úrovni súborov, ale aj na úrovni obsahov týchto súborov.

Určili sme ciele projektu, analyzovali a špecifikovali sme požiadavky na náš systém. Využili sme techniky prípadov použitia a procesných modelov. Navrhli sme architektúru, ktorá je modulárna a umožňuje dobre dekomponovať jednotlivé procesy. Zvolili sme technologický prístup, ktorý vedie k multiplatformovosti riešenia.

Opísali sme implementáciu jednotlivých modulov vytvoreného riešenia, s dôrazom na kľúčové aspekty procesov, ktoré jednotlivé moduly poskytujú. Venovali sme sa predovšetkým jadru aplikácie, rozhraniu pre príkazový riadok, nahrávaniu a konverzii súborov, integrácii s LMS Moodle, ako aj modelom porovnávania vzoriek. Ďalšími aspektmi boli predspracovanie slovenských textov pomocou odstraňovania stop-slov a lematizácia využívajúca získanú databázu, ako aj tokenizácia zdrojových kódov do sekvencií znakov. Z komparačných metód sme opísali N-gramy, Greedy String Tiling, String-blurring, LCS a latentnú sémantickú analýzu. Venovali sme sa aj vytvorenému grafickému pouívateľskému rozhraniu a vizualizátorom výsledkov.

Spomenuli sme aj niektoré úskalia, na ktoré sme počas implementácie narazili. Vyjadrili sme sa aj k otázke testovania produktu, ktorú sme ale podrobnejšie rozobrali v ďalšom dokumente - dokumentácii k riadeniu nášho projektu.

Vytvorené riešenie sme overili v experimentoch, ktorých výsledky sme zosumarizovali a zanalyzovali. Porovnanie s inými riešeniami svedčí o tom, že náš nástroj predstavuje pokrokové riešenie nielen z hľadiska architektonického, ale aj z hľadiska kvality procesu detekcie plagiarizmu.

Literatúra

[1] Clough, P.: Plagiarism in natural and programming languages: an overview of current tools and technologies. *Department of Computer Science, University of Sheffield* (2000), p. 1-31.

[2] Piaček, J., Kravčík, M.: Otvorená filozofická encyklopédia. Dostupné na internete: <http://ii.fmph.uniba.sk/~filit/fvp/parafraza.html>. [cit. 22-10-2009]

[3] Češka, Z.: Využití moderních přístupů pro detekci plagiátů. In: *Informačné technológie - aplikácie a teória*. 2008.

[4] Krajčí, S., Laclavík, M., Novotný, R., Turlíková L: The tool Morphonary: Using of word lemmatization in processing of documents in Slovak. *Institute of Computer Science, Univerzita Pavla Jozefa Šafárika*.

[5] Karp, R., Rabin, M.: Efficient Randomized Pattern-Matching Algorithms, *IBM Journal of Research and Development* 31(2), pp. 249–260 (1987).

[6] Wise, M:String Similarity via Greedy String Tiling and Running Karp-Rabin Matching. Dostupné na internete: <http://www.pam1.bcs.uwa.edu.au/~michaelw/ftp/doc/RKR_GST.ps>. [cit. 22-10-2009]

[7] Thanh Dao: The Longest Common Substring with Maximal Consecutive. September 2005. Dostupné na internete: <http://www.codeproject.com/KB/recipes/lcs.aspx>. [cit. 22-10-2009]

[8] GSL - GNU Scientific Library - GNU Project - Free Software Foundation (FSF). Dostupné na internete: <http://www.gnu.org/software/gsl/>. [cit. 22-10-2009]

[9] Grune, D., Huntjens, M.: Detecting Copied Submissions in Computer Science Workshops. *Vakgroep Informatica, Faculeit Wiskunde & Informatica*. 1989.

[10] Schleimer, S., Wilkerson, D. S., Aiken, A.: Winnowing: Local Algorithms for Document Fingerprinting. Marec 2003 [cit. 30-10-2009]. Dostupné na internete: <http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf>.

[11] Wise, Michael J.: Plagiarism Detection – YAP. Dostupné na internete: <http://www.pam1.bcs.uwa.edu.au/~michaelw/YAP.html>. [cit. 22-10-2009]

[12] Church, K. W., & Gale, W. A. Inverse document frequency (IDF): A measure of deviation from Poisson. In Proceedings of the third workshop on very large corpora (pp. 121-130). (1995) Dostupné na internete: <http://acl.ldc.upenn.edu/W/W95/W95-0110.pdf>. [cit. 22-10-2009]

[13] Sunehag, P. Using two-stage conditional word frequency models to model word burstiness and motivating tf-idf. In Meila, M. & Shen, X. (eds.) Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics (AISTATS 2007) (2007). <http://jmlr.csail.mit.edu/proceedings/papers/v2/sunehag07a/sunehag07a.pdf>. [cit. 22-10-2009]

[14] Demonstration of Sherlock - Plagiarism. Dostupné na internete: <http://www.ics.heacademy.ac.uk/resources/assessment/plagiarism/demo_sherlock.html>. [cit. 22-10-2009]

[15] Poppler, PDF rendering library based on the xpdf-3.0 code base. Dostupné na internete: <http://poppler.freedesktop.org/>. [cit. 08-12-2009]

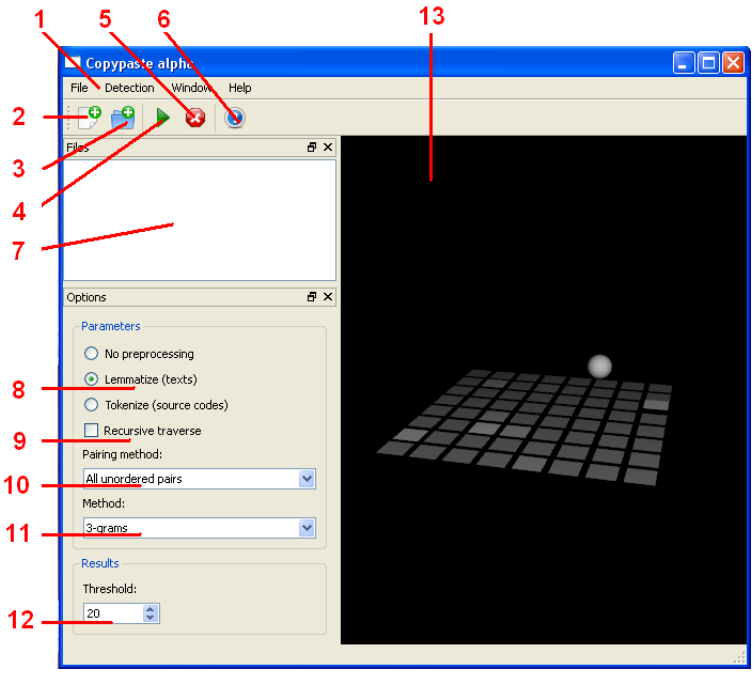
[16] JODConverter | Art of Solving. Dostupné na internete: http://www.artofsolving.com/opensource/jodconverter. [cit. 10-04-2009]

[17] Open Office, plnohodnotný kancelársky balík. Dostupné na internete: http://sk.openoffice.org/. [cit. 10-04-2009]

Príloha A - Používateľská príručka

Okno aplikácie

Na obrázku 10.1 je celé úvodné okno aplikácie, ktoré sa užívateľovi zobrazí po jej spustení. Aplikácia obsahuje tieto základné ovládacie prvky:



Obr. 10.1: Okno aplikácie

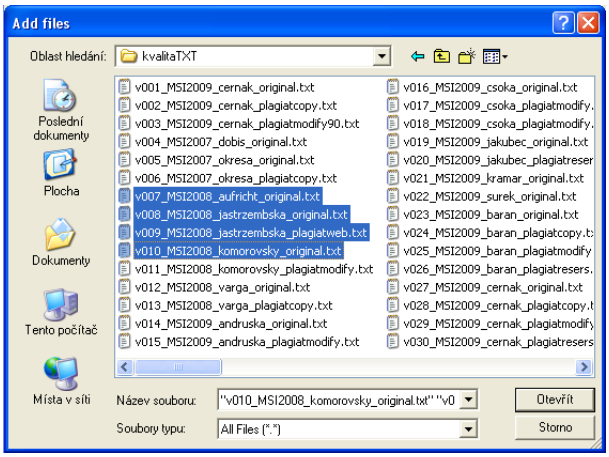
1. Hlavné menu, ktoré obsahuje položky Súbor, Kontrola, Okno a Pomoc. Užívateľ môže ovládať aplikáciu buď pomocou ikoniek, alebo položiek z tohto menu.
2. Tlačidlo pre pridanie súboru do zoznamu. (CTRL+A)
3. Tlačidlo pre pridanie zložky do zoznamu. (CTRL+D)
4. Tlačidlo pre spustenie kontroly. (CTRL+R)

5. Tlačidlo pre zastavenie kontroly. (Esc)
6. Tlačidlo pre zobrazenie manuálu. (F1)
7. Zoznam pridaných súborov a zložiek, ktoré sa budú kontrolovať.
8. Nastavenie predspracovania výberom metódy.
9. Nastavenie pre rekurzívne prehľadávanie zložiek pridaných do zoznamu. Otvoria sa všetky súbory v zložke a vo všetkých jej pod zložkách.
10. Nastavenie metódy párovania súborov.
11. Nastavenie metódy porovnávania.
12. Prah podobnosti. Páry súborov s rovnakou, alebo väčšou podobnosťou budú zvýraznené.
13. Okno aplikácie v ktorom sa zobrazia výsledky porovnania. Na začiatku zobrazuje jednoduchú animáciu.

Pridanie súborov a adresárov

1. Pre pridanie súborov do zoznamu kliknite na ikonu pre pridanie súboru (obr. 10.1 - prvok 2), alebo stlačte CTRL+A.
2. Zobrazí sa vám dialóg pre otvorenie súborov 10.2.
3. Označte súbor, ktorý chcete pridať.
4. Použitím kláves CTRL, alebo SHIFT môžete označiť viacero súborov súčasne.
5. Potvrďte voľbu stlačením tlačidla Otvoriť (popis tlačidla závisí od jazyka operačného systému).
6. Do zoznamu súborov sa pridajú zvolené súbory.

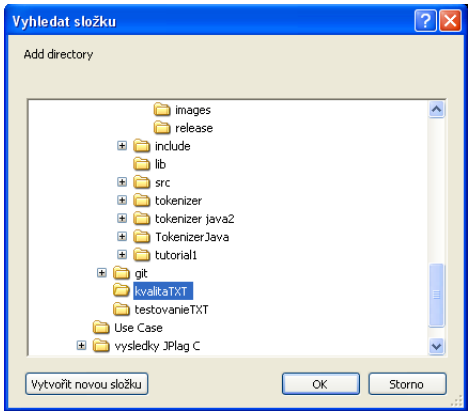
1. Pre pridanie zložiek do zoznamu kliknite na ikonu pre pridanie zložky (obr. 10.1 - prvok 3), alebo stlačte CTRL+D.
2. Zobrazí sa vám dialóg pre voľbu zložky 10.3.
3. Vyberte zložku ktorú chcete pridať.
4. Potvrďťe voľbu stlačením tlačidlo OK (popis tlačidla závisí od jazyka operačného systému).



Obr. 10.2: Pridanie súboru do zoznamu

5. Pokiaľ chcete aby boli pri kontrole otvorené súbory aj vo všetkých podzložkách (nie iba priamo v zložke), zaškrtnite voľbu Recursive traverse (obr. 10.1 - prvok 9).

6. Do zoznamu súborov sa pridá zvolená zložka, ktorej súbory budú otvorené pri spustení kontroly.



Obr. 10.3: Pridanie zložky do zoznamu

Kontrola textových súborov

1. Vyberte spôsob predspracovania (obr. 10.1 - prvok 8). Pre textové súbory sa odporúča možnosť lematizácia - Lemmatize (texts).

2. Vyberte spôsob párovania súborov (obr. 10.1 - prvok 10):

- All unordered pairs - budú vytvorené všetky kombinácie párov bez opakovania (súbory A s B).
- All ordered pairs - budú vytvorené všetky kombinácie párov s opakovaním (súbory A s B a aj B s A).
- First to all - budú vytvorené všetky kombinácie párov prvého súboru v zozname s ostatnými súbormi (iba A s B).

3. Vyberte metódu párovania súborov (obr. 10.1 - prvok 11):
   - 3-grams pre voľbu metódy 3-gramov
   - GST pre metódu Greedy String Tiling.
   - String-blurring.
   - LCS pre metódu najdlhšieho spoločného pod reťazca (longest common substring).
   - LSA pre metódu latentno-sémantickej analýzy.
4. Rôzne metódy porovnávania môžu mať rôzne výsledky.
5. Zvoľte hranicu podobnosti podozrivých párov súborov (obr. 10.1 - prvok 12).

Kontrola súborov zdrojovým kódom

1. Vyberte spôsob predspracovania (obr. 10.1 - prvok 8). Pre súbory so zdrojovým kódom sa odporúča možnosť tokenizácia - Tokenize (source codes).
2. Vyberte spôsob párovania súborov (obr. 10.1 - prvok 10):
   - All unordered pairs - budú vytvorené všetky kombinácie párov bez opakovania (súbory A s B).
   - All ordered pairs - budú vytvorené všetky kombinácie párov s opakovaním (súbory A s B a aj B s A).
   - First to all - budú vytvorené všetky kombinácie párov prvého súboru v zozname s ostatnými súbormi (iba A s B).
3. Vyberte metódu párovania súborov (obr. 10.1 - prvok 11):
   - 3-grams pre voľbu metódy 3-gramov
   - GST pre metódu Greedy String Tiling.
   - String-blurring.
   - LCS pre metódu najdlhšieho spoločného pod reťazca (longest common substring).

- LSA pre metódu latentno-sémantickej analýzy.

4. Pre zdrojové kódy sa odporúča metóda GST z dôvodu jej rýchlosti.
5. Rôzne metódy porovnávania môžu mať rôzne výsledky.
6. Zvoľte hranicu podobnosti podozrivých párov súborov (obr. 10.1 - prvok 12).

Spustenie kontroly

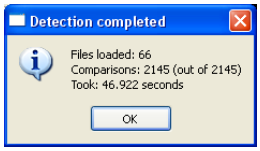
1. Po vybratí potrebných súborov a zložiek pre kontrolu a nastavenie kontroly kliknite na ikonu zelenej šípky (obr. 10.1 - prvok 4), alebo stlačte CTRL+R.
2. Spustí sa kontrola v troch fázach. Načítanie súborov, predspracovanie a porovnávanie.
3. Počas jednotlivých fáz kontroly je v spodnej časti okna aplikácie zobrazený stavový riadok priebehu kontroly.
4. Kontrola môže trvať niekoľko minút v závislosti od počtu a veľkosti súborov.
5. Pokiaľ by kontrola trvala príliš dlho, môžete ju prerušiť stlačením klávesy Esc, alebo tlačidla s krížikom (obr. 10.1 - prvok 5).

Zobrazenie výsledkov

1. Po ukončení kontroly sa zobrazí dĺžka trvania kontroly a počet súborov a porovnaní (obr. 10.4).
2. Automaticky bude okno s animáciou (obr. 10.1 - prvok 13) nahradené oknom s výsledkami kontroly (obr. 10.5).
3. Aplikácia podporuje rôzne druhy vizualizácie výsledkov, pre lepšiu orientáciu v nich. Pre výber vizualizácie zvoľte jednu zo záložiek vo vrchnej časti okna (obr. 10.5):
   - Side-by-side zobrazenie, ktoré zobrazí oba súbory v páre vedľa seba so zvýraznením podobných súčastí.
   - Základné zobrazenie obsahuje zoznam (strom) súborov a pod každým je súbor s ktorým bol porovnaný.
   - Table zobrazí výsledky kontroly v tabuľke. Riadky a stĺpce predstavujú jednotlivé súbory a v políčkach je podobnosť páru súborov.
   - Graph je vizualizácia v ktorej jednotlivé súbory sú zobrazené ako uzly. Podľa podobnosti je hrana medzi súbormi výraznejšia a kratšia (podobné súbory sú bližšie a opačne).

Graf sa snaží zaujať optimálnu polohu jednotlivých uzlov podľa podobnosti. Odznačením možnosti weighted (obr. 10.5) môžete zrušiť toto automatické usporiadanie.
Myšou je možné presúvať jednotlivé uzly, alebo celý graf podľa potreby.
Taktiež je možné si graf priblížiť, vycentrovať alebo prekresliť na začiatočný stav.

- Histogram zobrazí výsledky podobnosti všetkých párov v histograme.



Obr. 10.4: Dialóg ukončenia kontroly

Uloženie výsledkov do súboru

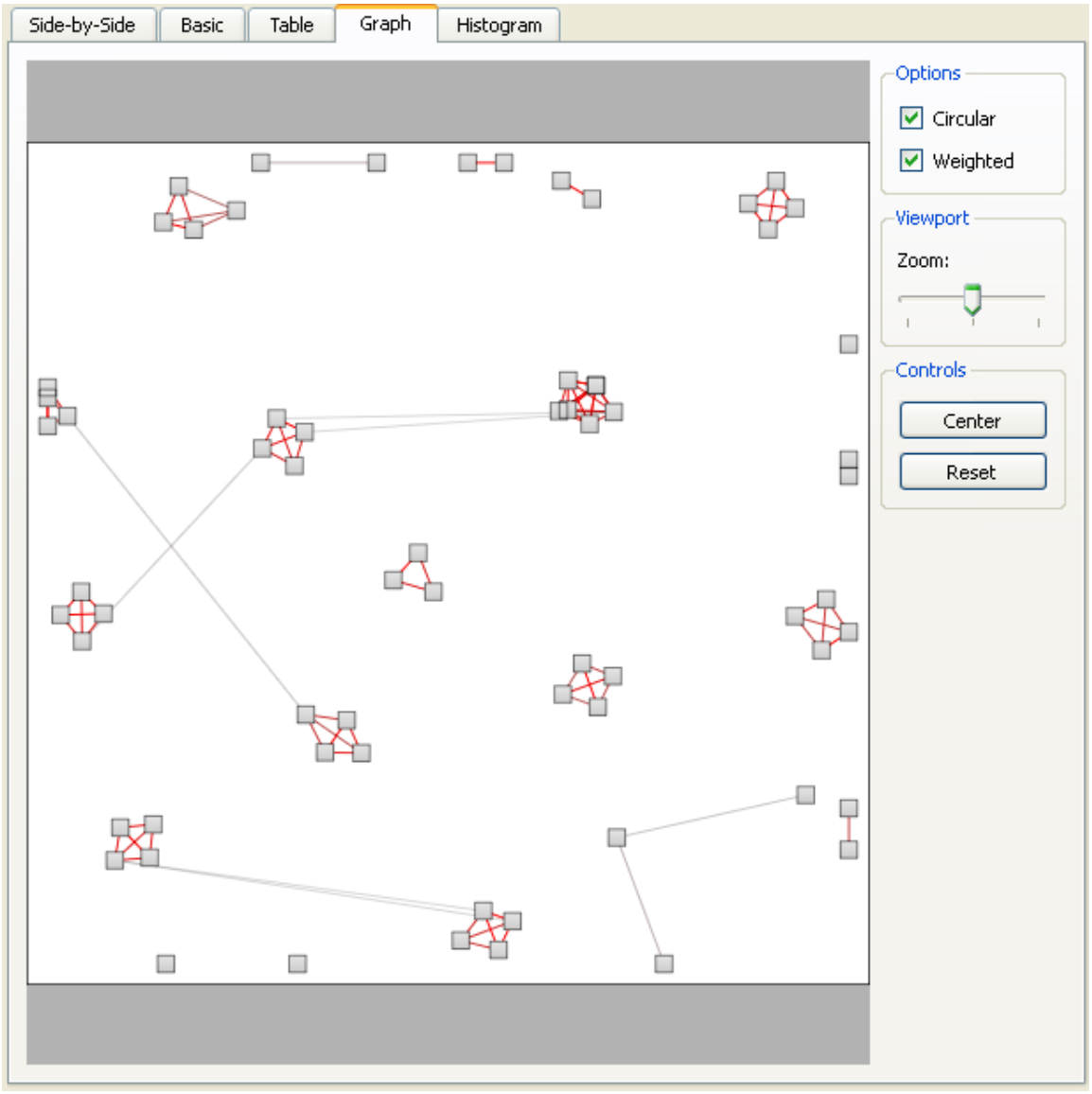
Jednotlivé výsledky kontroly je možné uložiť do súboru

1. Vyberte položku File z hlavného menu (obr. 10.1 - prvok 1).
2. Vyberte možnosť Export results.
3. Zvoľte typ súboru.
4. Otvorí sa vám dialóg pre uloženie súboru (obr. 10.6).
5. Zvoľte meno a umiestnenie súboru a stlačte Uložiť (popis tlačidla závisí od jazyka operačného systému).

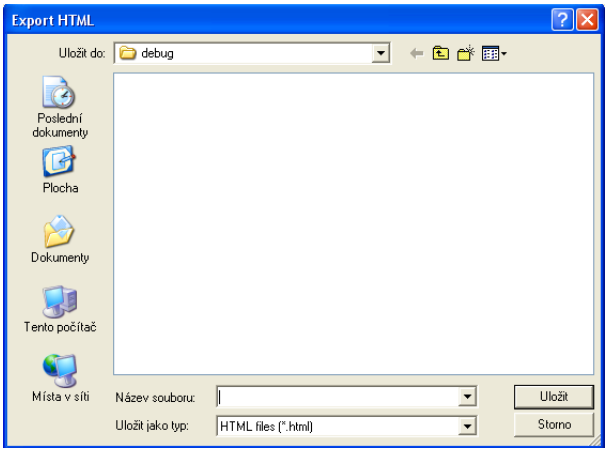
Zobrazenie anglického manuálu

Aplikácia obsahuje tento manuál v anglickom jazyku.

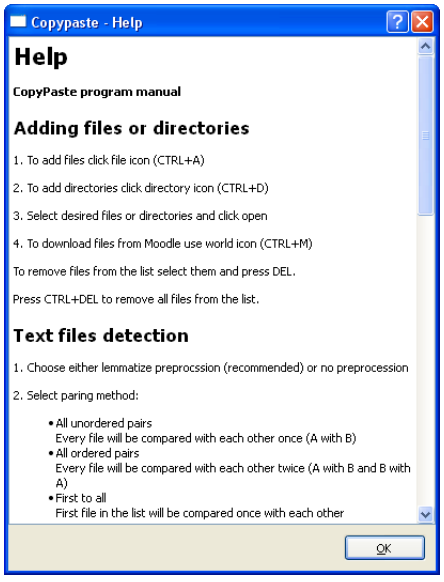
1. Stlačte klávesu F1, alebo tlačidlo s otáznikom v modrom krúžku (obr. 10.1 - prvok 6).
2. Zobrazí sa vám manuál aplikácie v anglickom jazyku (obr. 10.7).



Obr. 10.5: Vizualizácia výsledkov v grafe



Obr. 10.6: Uloženie výsledkov kontroly



Obr. 10.7: Anglický manuál aplikácie