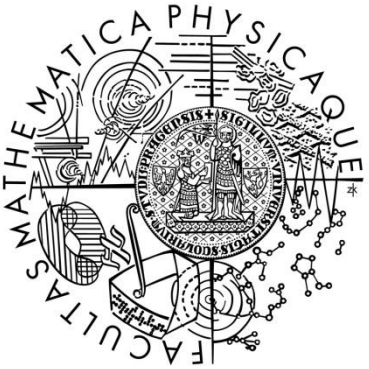
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Vladimír Sadloň

Detekce duplicit v rozsáhlých webových bázích dat

Katedra softwarového inženýrství

RNDr. Leo Galamboš Ph.D.

Studijní program: Informatika

Studijní obor: Softwarové systémy

Praha 2011

Týmto by som sa chcel poďakovať vedúcemu svojej práce RNDr. Leo Galambošovi Ph.D. za cenné rady a podporu, a svojej rodine a snúbenici za trpezlivosť, ktorú so mnou mali počas tvorby tejto práce.

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V …..... dne..... podpis

Název práce: Detekce duplicit v rozsáhlých webových bázích dat

Autor: Vladimír Sadloň

Katedra / Ústav: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Leo Galamboš Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Tato diplomová práce se zabývá metodami používanými k detekci duplicitních dokumentů, a možností jejich integrace do internetového vyhledávače. Nabízí přehled běžně používaných metod, z nichž vybírá metodu aproximace Jaccardovy míry podobnosti v kombinaci se šindelováním. Vybranou metodu přizpůsobuje k implementaci v prostředí internetového vyhledávače Egothor. Cílem práce je představit tuto implementaci, popsat její vlastnosti a nalézt nejvhodnější parametry tak, aby detekce probíhala pokud možno v reálném čase. Důležitou vlastností metody je také možnost vykonávat dynamické změny nad kolekcí indexovaných dokumentů.

Klíčová slova: plagiátorství, detekce duplicitních dokumentů, online vyhledávání.

Title: Detection of Duplicates in Huge Web Databases

Author: Vladimír Sadloň

Department: Department of Software Engineering

Supervisor: RNDr. Leo Galamboš Ph.D., Department of Distributed and Dependable Systems

Abstract: This master thesis analyses the methods used for duplicity document detection and possibilities of their integration with a web search engine. It offers an overview of commonly used methods, from which it chooses the method of approximation of the Jaccard similarity measure in combination with shingling. The chosen method is adapted for implementation in the Egothor web search engine environment. The aim of the thesis is to present this implementation, describe its features, and find the most suitable parameters for the detection to run in real time. An important feature of the described method is also the possibility to make dynamic changes over the collection of indexed documents.

Keywords: plagiarism, duplicity document detection, online search.

Obsah

1. Úvod

1.1. Motivácia

Slovo „plagiátorstvo“ pochádza z latinského „plagiarus“, v doslovnom preklade znamená toto slovo „únosca“. Prvýkrát ho použil rímsky básnik Martial, keď obvinil svojho rivala, Fidentinusa, z krádeže svojich veršov [1]. Plagiátorstvo, ako prisvojovanie si cudzích myšlienok, teda určite existuje už niekoľko tisícročí, ale až v osemnástom storočí začalo byť pomaly považované za problém, a začalo sa operovať s pojmami ako originalita a duševné vlastníctvo [1].

Rozmach plagiátorstva potom priniesla druhá polovica dvadsiateho storočia, kedy osobné počítače a možnosť prevedenia jednotlivých diel do elektronickej podoby plagiátorom výrazne uľahčili prácu. Definitívny zlom nastal s rozmachom internetu, kde jednoduchá dostupnosť obrovského množstva zdrojov umožňuje jednoducho vytvoriť plagiát pomocou metódy Copy&Paste. A pokiaľ plagiátor vkladá cudzí text medzi vlastný a prípadne mierne mení slovosled či používa synonymá, je pravdepodobnosť „ručného“ odhalenia takéhoto plagiátu pomerne malá.

Najmä, i keď nielen, v Slovenskej a Českej republike sa v poslednom období dostávajú na verejnosť závažné prípady plagiátorstva, pre Slovenskú republiku sú to napríklad prípady popísané v [2], [3], [4], pre Českú republiku napríklad kauzy popísané v [5], [6], [7]. Vo všetkých prípadoch sa jedná o známe osobnosti verejného či akademického života, a bez ohľadu na právne dôsledky pre spomenuté osoby je jasné, že v našej spoločnosti je plagiátorstvo stále významným fenoménom, proti ktorému je treba bojovať. Zdá sa totiž, že spoločnosť sa stále riadi citátom: „Čo je to originalita? Neodhalené plagiátorstvo.“ od Williama Ralpha Ingeho.

Jedným z problémov plagiátorstva je aj fakt, že duševné vlastníctvo, respektíve vlastníctvo nejakej myšlienky je pre mnohých ľudí ťažko uchopiteľný koncept, navyše v mnohých prípadoch neexistuje platná legislatíva, ktorá by dokázala jednoznačne odpovedať na mnohé otázky s plagiátorstvom spojené – napríklad ako nakladať s prepisom myšlienky vlastnými slovami, či možno

necitovanie myšlienok z vlastnej predchádzajúcej práce považovať za plagiátorstvo, ako postihovať tzv. „ghostwriting“ (vytvorenie práce, ktorú daná osoba vydáva za svoje dielo, iným človekom) a ďalšie. Tieto problémy sa reálne nedajú vyriešiť inak, ako zmenou postoja celej spoločnosti k plagiátorom a plagiátorstvu, čo je najlepšie možné neustálym zdôrazňovaním škodlivosti a nebezpečnosti plagiátorstva, spolu s vytváraním čo najväčších bariér pre plagiátorov. Jednou z takýchto bariér je určite aj maximalizovanie pravdepodobnosti, že plagiátorstvo bude zistené, na čo je potrebné mať efektívny systém ako plagiáty odhaľovať, a práve vytvorenie takéhoto systému je hlavným cieľom tejto práce.

Plagiátorstvo je v slovníkoch definované ako „neoprávnené privlastnenie, imitácia alebo odcudzenie myšlienok, nápadov, či celkového prejavu iného autora a vydávanie tohto za vlastné dielo“ [8]. Z tohto pohľadu môže byť plagiátom takmer čokoľvek, či už sa jedná o akékoľvek umelecké dielo, počítačový software, technické riešenie nejakého problému, a podobne. V tejto práci sa ale budeme zaoberať iba takým typom plagiátorstva, ktoré postihuje oblasti, kde majú výsledné diela textovú podobu.

1.2. Štruktúra práce

Kapitola 2 obsahuje vymedzenie pojmov a ciele tejto práce, ako aj kritériá, podľa ktorých budeme hodnotiť splnenie jednotlivých cieľov. Kapitola 3 obsahuje prehľad používaných metód a ich rozdelenie podľa niektorých relevantných kritérií. V kapitole 4 popíšeme použité prostredie pre implementáciu systému na detekciu plagiátov, v kapitole 5 sa budeme zaoberať výberom najvhodnejšieho algoritmu a jeho prispôsobením pre on-line detekciu plagiátov. Kapitola 6 pojednáva o implementácií systému v prostredí internetového vyhľadávača Egothor. Obsahom siedmej kapitoly bude experimentálne overenie vlastností použitého systému, v ôsmej kapitole potom použitý systém porovnáme s už existujúcimi implementáciami. Záverečná kapitola prináša zhrnutie výsledkov tejto práce.

2. Vymedzenie problémov a cieľov práce

2.1. Typy plagiátorstva

Ako bolo spomenuté v úvode, cieľom tejto práce je zaoberať sa iba plagiátmi diel, ktoré sú vo svojej výslednej podobe z väčšej časti tvorené textom. Je to jednak z toho dôvodu, že koncept plagiátorstva je napríklad pre umelecké diela, ako sú obrazy či sochy, ešte oveľa ťažšie uchopiteľný, jednak preto, že najzávažnejšie prípady sa väčšinou týkajú prác v textovej podobe, a to najmä na akademickej pôde, kde sa podvody objavujú pri semestrálnych, ale aj bakalárskych, záverečných, či dokonca rigoróznych prácach. Podľa štúdie [9], plagiátorstvo „vzrástlo medzi rokmi 1994 až 1997 o neuveriteľných 744%“. V tejto istej štúdií sa takisto objavuje fakt, že „54% študentov priznalo (do istej miery) že sa dopustili plagiátorstva z Internetu, a 74% študentov priznalo že sa aspoň raz počas štúdia dopustili „závažného podvodu“.

Na stránkach plagiarism.org je uvedené nasledovné rozdelenie plagiátorstva na jednotlivé typy [10]:

a. Plagiátorstvo bez uvedených zdrojov
    1) Kopírovanie celej práce od slova do slova.
    2) Kopírovanie značnej časti (častí) z jedného zdroja, bez akýchkoľvek zmien v texte.
    3) Kopírovanie z viacerých rôznych zdrojov, s prípadnými zmenami robenými tak, aby na seba jednotlivé časti nadväzovali.
    4) Jemné zmeny v texte, jedná sa najmä o pozmenenie kľúčových slov a fráz.
    5) Zásadné zmeny v použitých slovách a frázach. V tomto prípade si dal plagiátor značnú prácu s použitím synoným, zmenou slovosledu či poradia jednotlivých viet.
    6) Kopírovanie vlastnej predchádzajúcej práce bez uvedenia zdroja.
b. Plagiátorstvo s uvedenými zdrojmi

1) Uvedenie že text je citovaný, ale neuvedenie presného zdroja, kde sa pôvodná práca dá nájsť.
2) Nesprávne uvedenie zdroja, čo spôsobí dohľadanie pôvodnej práce nemožným.
3) Uvedenie zdroja, ale neumiestnenie presne citovaného textu do úvodzoviek. Plagiátor týmto síce priznáva originálnu myšlienku pôvodnej práci, ale prisvojuje si interpretáciu danej myšlienky.
4) Správne uvedenie všetkých zdrojov, ale práca neobsahuje takmer žiaden originálny text.
5) „Perfektný zločin“ – plagiátor vo väčšine prípadov správne cituje zdroje, ale v niekoľko málo prípadoch zdroje necituje a prisvojuje si tak pôvod necitovaných myšlienok.

Táto práca si kladie za cieľ odhaľovať najmä typy plagiátov z časti a., posúdenie správnosti citácie zdrojov necháva na používateľovi.

2.2. Definícia pojmu plagiát

V mnohých prípadoch je veľmi ťažké zadefinovať pojem „plagiát“. Pre automatickú detekciu plagiátov je však nutné takúto definíciu zaviesť, pričom vágna definícia z úvodu práce rozhodne nie je dostačujúca.

Majme dokumenty P a Q, oba dokumenty sú rozdelené na časti p_i a q_i, $i = 1..N, j = 1..M$, kde N, respektíve M, je počet častí dokumentu. Časťou dokumentu sa v tomto prípade myslí akákoľvek lingvistická jednotka - slovo, veta, odsek, respektíve určitý počet zreťazených lingvistických jednotiek – tak, aby platilo:

$$\sum_{i=1}^{n} p_i = P \tag{2.1}$$

Inými slovami, dokument môžeme rozdeliť na jednotlivé slová, alebo časti po 200 slov, respektíve na jednotlivé vety, alebo časti po 10 viet, a tak ďalej.

Teraz, pre jednotlivé časti dokumentov je potrebné spočítať ich vzájomnú podobnosť. Majme tzv. podobnostnú funkciu $sim(a, b)$, ktorá pre dve časti dokumentu a a b spočíta ich vzájomnú podobnosť. Označme túto podobnosť s. Pre časti dokumentov P a Q teda platí:

$$s_{i,j} = sim(p_i, q_j), i = 1..n, j = 1..m \quad (2.2)$$

Následne potrebujeme hodnotu, ktorá bude oddeľovať podobné časti dokumentov od rôznych, to znamená že potrebujeme hodnotu, označme ju δ, pre ktorú bude platiť, že časť p_i je plagiátom q_j, pokiaľ $sim(p_i, q_j) > \delta$. Z toho dostávame množinu podobných dvojíc S, pre ktorú platí:

$$S = \{(p_i, q_j) | s_{i,j} > \delta\} \quad (2.3)$$

Konečne, majme definovanú hodnotu t, ktorá bude udávať minimálny počet podobných jednotiek v množine S tak, aby mohol byť dokument P považovaný za plagiát dokumentu Q (respektíve vice versa). Inými slovami, P a Q sú si podobné práve vtedy, ak $|S| > t$.

V tejto práci sa nebudeme snažiť výrazne obmedzovať hodnoty δ a t, ale vydáme sa skôr cestou prezentovania všetkých výsledkov užívateľovi, zoradených podľa δ, respektíve t, aby mohol sám posúdiť, či je daný dokument plagiátom alebo nie. Tento prístup bol zvolený najmä kvôli skutočnosti, že je niekedy veľmi ťažké posúdiť, či je daná časť textu citovaná korektne, alebo sa jedná o plagiátorstvo, a teda je vždy nutná verifikácia živou osobou. Ďalším dôvodom je aj skutočnosť, že ani veľké množstvo rovnakého textu nemusí vždy znamenať, že sa jedná o plagiát – „v oblastiach ako je literatúra či právo práce často pozostávajú z hypotézy nasledovanej stovkami citácií z rôznych zdrojov na jej potvrdenie alebo vyvrátenie. V iných oblastiach, napríklad v matematike, je nutné poskytnúť čitateľovi odborný základ k danej téme, takže vlastné výsledky a dôkazy môžu zaberať dokonca menej ako jednu tretinu práce“ [11].

2.3. Pravdepodobnosť označenia plagiátu

Dôležitou vlastnosťou systému na odhaľovanie plagiátov je aj presnosť, s akou sa danému systému darí plagiáty označovať. To znamená, ako často systém neoznačí dokument, ktorý je plagiátom iného dokumentu, a naopak, ako často označí ako plagiát dokument, ktorý ním nie je.

Na tento účel boli zavedené pojmy „falošné negatívum“, čiže nesprávne neoznačenie plagiátu, a „falošné pozitívum“, čiže nesprávne označenie dokumentu, ktorý plagiátom nie je.

2.4. Dynamické zmeny nad množinou referenčných dokumentov

Množina referenčných dokumentov v systéme určenom na odhaľovanie duplicít môže mať značnú veľkosť – rádovo až desaťtisíce dokumentov v systéme obsahujúcom záverečné práce študentov univerzity, prípadne stotisíce dokumentov pre veľké on-line repozitáre. Množinou referenčných dokumentov sa v tomto prípade myslí kolekcia dokumentov, voči ktorej systém kontroluje, či zadaný dokument nie je plagiátom niektorého z dokumentov z referenčnej množiny podľa definície z kapitoly 2.2.

Predspracovanie takejto množiny pre systém na detekciu plagiátov si vyžiada určitý čas, a vzhľadom k veľkosti množiny je jasné, že tento čas bude signifikantný. Preto by bolo vhodné, aby v prípade prírastku, respektíve úbytku, dokumentov do referenčnej množiny nebolo nutné opätovne spracovávať všetky dokumenty, ale stačilo spracovať iba zmienený prírastok či úbytok.

2.5. Očakávaná doba odpovede

Efektivita systému na odhaľovanie plagiátov nezávisí iba od toho, s akou presnosťou sú duplicity nachádzané. Významným faktorom sa totiž stáva aj očakávaná doba odpovede systému, t.j. za akú dobu je systém schopný poskytnúť užívateľovi relevantné informácie. Tento faktor je obzvlášť dôležitý u webových aplikácií, kde má užívateľ tendenciu jednoducho zatvárať stránky, ktoré negenerujú žiadnu odpoveď do určitého času. Podľa štúdie [12] je tento čas maximálne trinásť sekúnd v prípade, že užívateľ nie je vôbec upovedomený o tom, že by sa stránka načítala a maximálne tridsaťosem sekúnd v prípade, že stránka zobrazovala „progress bar“ (tento „progress bar“ sa pohyboval obojsmerne – sprava doľava, zľava doprava, atď. – a nemal v sebe zahrnutú informáciu o zostávajúcom čase, respektíve o už vykonanej časti práce).

Názory jednotlivých štúdií na akceptovateľnú dobu odpovede sú pomerne rôzne. Jedna z prvých prác na túto tému, [20], uvádza časové odhady 0,1 sekundy pre interaktívnu prácu, 1 až 2 sekundy pre „zobrazenie ďalšej stránky z dátami“ a 15 sekúnd na vyriešenie komplexnej úlohy zadanej systému. Ďalšia práca z roku 1991

([13]), či novšia z roku 2010 ([14]) uvádzajú časy podobné predchádzajúcej štúdií, konkrétne sa v štúdií [14] uvádza:

- Čas 0,1 sekundy dáva užívateľovi pocit okamžitej odpovede – to značí, že užívateľ má pocit že výsledok akcie zapríčinil on a nie počítač.
- Čas do 1 sekundy udržuje myšlienkové pochody užívateľa neprerušené. Užívateľ môže pociťovať určité oneskorenie, a teda vie, že výsledok akcie vygeneroval počítač, ale stále cíti že má nad systémom kontrolu a nemusí na systém čakať.
- Čas do 10 sekúnd udržuje užívateľovu pozornosť – po desiatich sekundách sa užívateľ začne sústrediť na iné veci a má problém sa opäť preorientovať na pôvodný problém

Ďalšie práce uvádzajú rôzne iné limity, napríklad 12 sekúnd v štúdii [15] je uvedených ako doba, po ktorej výrazne rastie nespokojnosť užívateľa, niektoré staršie štúdie stanovujú tento čas oveľa vyššie, napríklad v [16] to je 41 sekúnd, po ktorých užívateľ o stránku výrazne stráca záujem.

V tejto práci sa budeme samozrejme snažiť o čo najrýchlejšiu dobu odpovede. Vzhľadom k tomu, že v predchádzajúcich štúdiách sa maximálna akceptovateľná doba odpovede pohybuje v intervale do 10 sekúnd, chceli by sme tiež dosiahnuť dobu odpovede maximálne 10 sekúnd.

Problematika očakávanej doby odpovede je samozrejme omnoho zložitejšia, mnohé štúdie sa venujú takzvanej „vnímanej“ dobe odpovede, ktorá sa môže od tej reálnej značne líšiť ([17], [18], [19]), ako aj skúmaniu ďalších faktorov, akými sú vek, typ internetového pripojenia, skúsenosti užívateľa, demografické a kultúrne hľadisko ([17], [21]) a podobne, toto však presahuje rámec tejto práce.

2.6. Systém z užívateľského hľadiska

Cieľom práce je vytvoriť systém, ktorý by bol dostupný čo najširšej vrstve užívateľov, či už študentom, ktorí si chcú skontrolovať svoje práce, pedagógom, ktorí potrebujú skontrolovať práce študentov, alebo komukoľvek inému. Z tohto hľadiska sa iste javí najvhodnejšou stratégiou implementovať systém ako webovú aplikáciu. Asi najväčšou nevýhodou takéhoto riešenia sú určité obmedzenia vo funkcionalite užívateľského rozhrania vzhľadom napríklad k riešeniu klient –

server, ale prihliadajúc ku skutočnosti, že užívateľ potrebuje k ovládaniu systému na detekciu plagiátov pomerne málo akcií, v zásade iba akciu na vloženie dokumentu a pár ďalších akcií na prehliadanie výsledkov, nemá toto znevýhodnenie vplyv na implementovaný systém. Navyše, vývoj frameworkov pre webové aplikácie stále napreduje a dnes existuje veľmi málo prípadov, kedy sa nejaké riešenie nedá v takýchto aplikáciách uskutočniť.

Predpokladáme, že nie všetci užívatelia systému budú skúsení v používaní počítača, preto by bolo vhodné, aby systém bol čo najintuitívnejší, a prezentoval užívateľovi výsledky v nejakej štandardnej forme, podobnej väčšine dnešných vyhľadávačov. Vhodné by tiež bolo, ak by mal užívateľ možnosť priamo zobraziť podobné časti porovnávaných dokumentov. Dôležitá je taktiež aj podpora aspoň základných formátov dokumentov, aby užívateľ nemusel zložito manipulovať s dokumentom iba kvôli tomu, aby ho do systému vôbec mohol vložiť.

2.7. Zhrnutie cieľov práce

Ciele tejto diplomovej práce sú nasledujúce:

1) Preštudovať a zhrnúť existujúce postupy používané na detekciu plagiátov či duplicitných dokumentov
2) Z týchto postupov vybrať najvhodnejší postup a upraviť ho tak, aby spĺňal nasledujúce požiadavky:
    a. Detekcia čo najväčšieho množstva plagiátov bez uvedených zdrojov
    b. Minimalizovať množstvo falošných negatív a falošných pozitív
3) Implementovať tento postup a navrhnúť systém na detekciu plagiátov tak, aby:
    a. Priemerná doba odpovede systému neprekročila 10 sekúnd
    b. Bolo možné vykonávať dynamické zmeny nad množinou referenčných dokumentov
    c. Systém bol užívateľsky prívetivý, čo značí predovšetkým poskytnúť užívateľovi výsledky v ľahko čitateľnej forme, a podporu základných formátov dokumentov
4) Otestovať systém na dátach čo najviac sa približujúcich reálnym dátam z akademického prostredia, medziiným:

   a. Otestovať ako sa systém správa pri zmenách najdôležitejších parametrov z hľadiska časovej náročnosti
   b. Zhodnotiť časové, priestorové a I/O nároky systému
5) Porovnať implementovaný systém s inými dostupnými systémami riešiacimi podobnú úlohu

3. Prehľad existujúcich metód na odhaľovanie duplicitných dokumentov

Aj keď je cieľom tejto práce najmä detekcia plagiátorstva, mnohé existujúce metódy a postupy boli pôvodne vytvorené na mierne odlišné úlohy, prípadne sa z takýchto metód vyvinuli. Preto niektoré tieto metódy a postupy považujeme za vhodné uviesť v nasledujúcom prehľade.

Jednou z prvých metód bola metóda na detekciu duplicít v zdrojových kódoch programov, založená na rozdelení jednotlivých zdrojových kódov do tried podľa štyroch parametrov – počet jedinečných operátorov, počet jedinečných operandov, počet všetkých výskytov operátorov a počet všetkých výskytov operandov [22]. Už v tejto práci je ale spomenutých niekoľko problémov, ktoré v detekcii duplicitných dokumentov pretrvávajú dodnes, ako napríklad problém s rozdielnou dĺžkou porovnávaných dokumentov, či problémy súvisiace so zmenou iba malej časti dokumentu.

Ďalším smerom, ktorým sa niektoré práce vydali, je prevencia plagiátorstva namiesto jeho detekcie. Toto zahŕňa použitie najrôznejších metód na znemožnenie kopírovania textu, značkovanie dokumentov, či už viditeľné, alebo neviditeľné, a podobne. Týmto metódam sa venujú napríklad práce [24], [25]. Nevýhodou takýchto postupov je však zvýšenie zložitosti používania dokumentov užívateľmi, ktorí nechcú plagiarizovať, a navyše sťaženie či znemožnenie elektronického kopírovania neznamená, že sa text nedá jednoducho odpísať.

Nárast výkonu počítačov v deväťdesiatych rokoch minulého storočia umožnil detegovať duplicity nielen v zdrojových kódoch, ale aj v dokumentoch prirodzeného jazyka. Jednou z prvých prác na túto tému bola detekcia podobných súborov v súborovom systéme od Udiho Manbera [23], ktorá položila základy pre nasledujúci výskum.

3.1. Základné rozdelenie metód

Rozdelenie podľa účelu metódy

Účelom metód na detekciu duplicitných dokumentov nemusí nutne byť iba detekcia plagiátorstva. Jedným z problémov súčasného webu je existencia veľkého počtu duplicitných, alebo takmer duplicitných stránok, ktoré majú takmer totožný obsah Tieto stránky sa ale líšia v pre užívateľa nepodstatných detailoch, ako napríklad metadáta stránky, alebo reklamy. Je nežiaduce, aby internetový vyhľadávač v rámci odpovede na dotaz vracal viacero de facto rovnakých stránok, preto je nutné takéto duplicity eliminovať.

Toto rozdelenie nie je samozrejme striktné, väčšina vyvinutých algoritmov sa s úspechom používa v oboch prípadoch, skôr sa jedná o nastavenie a vyladenie parametrov systému tak, aby čo najlepšie vyhovoval vybranej možnosti.

Rozdelenie podľa spôsobu použitia

Metódy sa delia na metódy typu $1{:}n$ a metódy typu $n{:}n$. Metódy typu $1{:}n$ sa používajú v prípade, že je potrebné rozhodnúť, či je konkrétny dokument podobný niektorému dokumentu z danej kolekcie n dokumentov. Metódy typu $n{:}n$ sa používajú v prípade, že je potrebné v kolekcií n dokumentov nájsť všetky dvojice podobných dokumentov.

Mohlo by sa zdať, že každá metóda $1{:}n$ sa môže jednoduchou úpravou zmeniť na metódu $n{:}n$, to však nie je celkom pravda, pretože takto naivný prístup by častokrát znamenal zbytočne veľkú časovú zložitosť. Príklad je možné nájsť v štúdii [29].

Rozdelenie podľa jazykovej závislosti

Metódy delíme podľa jazykovej závislosti na jazykovo závislé, teda schopné detegovať iba duplicity v jednom jazyku – takáto je väčšina ďalej popisovaných metód. Jazykovo nezávislé metódy sú schopné nachádzať duplicity bez ohľadu na jazyk predloženého dokumentu – väčšinou pracujú iba so štatistickými vlastnosťami textu ako priemerný počet slov vo vete, resp. dokumente, priemerná dĺžka odsekov, a podobne. Medzi takéto metódy patrí napríklad metóda popísaná v [33]. Konečne

viacjazyčné metódy sú schopné detegovať duplicity aj medzi dokumentmi napísanými v rôznych jazykoch, príkladom takejto metódy je [34] alebo [52].

3.2. Metódy založené na šindľovaní

Majme dokument D, ktorý je postupnosťou lingvistických jednotiek – písmen, slov, viet, atď. Ďalej majme kladné prirodzené číslo k. Potom definujme šindeľ ako ľubovoľnú podpostupnosť dokumentu D dĺžky k. Šindľovanie (z anglického „shingling“), respektíve k - šindľovanie je definované ako množina všetkých unikátnych šindľov dokumentu D. Napríklad, majme dokument (táto, rekurzia, je, táto, rekurzia, je, táto, rekurzia). Potom 4-šindľovanie tohto dokumentu je množina {(táto, rekurzia, je, táto), (rekurzia, je, táto, rekurzia), (je, táto, rekurzia, je)}.

Najvýznamnejšími parametrami šindľovania sú výber lingvistickej jednotky, výber vhodného k a výber vhodnej podmnožiny k – šindľovania dokumentu, teda výber reprezentácie dokumentu. Výber vhodného k je dôležitý pre redukciu falošných pozitív, respektíve falošných negatív, kde s rastúcim číslom k rastie aj počet falošných negatív, teda plagiátov, ktoré sa nepodarilo odhaliť. Na druhú stranu pre malé k môže dôjsť k prudkému nárastu falošných pozitív, jednoducho preto, že niektoré ustálené slovné spojenia sa môžu nachádzať vo viacerých prácach a vôbec to nemusí znamenať že sa jedná o plagiát (Napr. „majme prirodzené číslo n“). Výber vhodnej podmnožiny zasa ovplyvňuje pomer presnosť systému verzus rýchlosť systému – čím väčšia podmnožina, tým bude systém presnejší, ale i pomalší.

Vybraná podmnožina sa samozrejme z pochopiteľných dôvodov neukladá v originálnej podobe, namiesto toho sa spočíta odtlačok (hash) každého šindľa pomocou niektorej z vhodných hashovacích metód, a pre účely jednotlivých algoritmov sa používa tento hash. Samozrejmá je správna voľba hashovacej funkcie tak, aby dochádzalo k čo najmenej kolíziám, a teda aj k čo najmenej falošným pozitívam.

Práca [26] používa ako lingvistické jednotky celé vety, skúma $1 \leq k \leq 5$ a rôzne stratégie výberu reprezentácie dokumentu. Experimenty však autori práce vykonávajú na veľmi obmedzenej kolekcií (92 dokumentov). V rámci práce vyvinuli systém nazývaný COPS – COpy Protection System. Heinze [27] používa ako lingvistické jednotky jednotlivé písmená, a to najmä preto, že podľa jeho výsledkov

je pri konverzií dokumentov v najrôznejších formátoch do textových súborov (OCR, Postscript, atď.) členenie na ostatné jednotky značne nespoľahlivé. Používa menšiu podmnožinu k – šindľovania pre indexované dokumenty, a väčšiu pre vyhľadávanie. Svoje výsledky získava na množine cca. 3000 dokumentov. Vyvinutý systém nazýva Koala. Štúdia [28] navrhuje premenlivú dĺžku šindľov s tým, že najkratšie a najdlhšie šindle nebudú využívané. V práci [31] je navrhnutý výber reprezentatívnej podmnožiny tak, aby vzdialenosť medzi dvomi šindľami nepresahovala (zvolenú) konštantu. Na veľmi rozsiahlych dátach (až 30 000 000 dokumentov) testoval šindľovací algoritmus Broder [32], ktorý definoval podobnosť dvoch dokumentov na základe Jaccardovej miery podobnosti vybraných reprezentácií daných dokumentov. Jeho algoritmus je však stavaný na $n{:}\,n$ spracovanie celej množiny dokumentov. Vo svojej ďalšej práci ([29]) navrhuje postup, ktorý nazýva super – šindľovanie, teda vytváranie šindľov zo šindľov. Práca [30] takisto definuje podobnosť dokumentov na základe Jaccardovej miery, a ponúka rozdelenie algoritmu na inicializačnú fázu a fázu vyhodnotenia dotazu, kde fáza vyhodnotenia dotazu je spracovanie $1{:}\,n$ so zložitosťou lineárnou k počtu dokumentov v korpuse. Táto práca detekciu duplicitných dokumentov realizuje pomocou integrácie s internetovým vyhľadávačom Egothor. Tento vyhľadávač bude využívať aj naša práca, jeho bližší popis sa nachádza v kapitole 4.

3.3. Metódy založené na vektorovom modeli dokumentu

Vektorový model dokumentu ([39], [40]) je pojem veľmi dobre známy z oblasti získavania informácií (information retrieval). V zásade sa jedná o model, v ktorom je každý dokument reprezentovaný množinou určitých vlastností, tieto vlastnosti následne tvoria vektor v euklidovskom priestore, kde každý rozmer zodpovedá jednej z vlastností. V klasickom vektorovom modeli zodpovedá jeden rozmer jednému slovu, respektíve termu, na spočítanie jednotlivých súradníc dokumentu sa používa najčastejšie schéma, $tf - idf$ kde

$$d_t = tf_{t,d}.idf_d \tag{3.1}$$

teda súradnica vektoru zodpovedajúca termu t je vypočítaná ako násobok frekvencie termu t v dokumente d, a inverznej frekvencie termu t v celom korpuse.

Samozrejme sú možné aj iné spôsoby výpočtu súradníc, napr. sublineárne tf škálovanie, kde

$$d_t = \begin{cases} 1 + \log tf_{t,d}\,, & tf_{t,d} > 0 \\ x, & tf_{t,d} \leq 0 \end{cases} \tag{3.2}$$

Toto odráža skutočnosť, že dvadsať výskytov termu t nemusí nutne znamenať dvadsaťnásobné zvýšenie dôležitosti oproti jednému výskytu [40]. Vektory pre jednotlivé dokumenty majú samozrejme rovnakú dĺžku.

Vektory je navzájom možné porovnávať za pomoci podobnostných mier, napríklad:

Kosínusová miera: $C(A,B) = \frac{\sum_i a_i b_i}{\sqrt{|a|}\sqrt{|b|}}$

Skalárny súčin: $S(A,B) = \sum_i a_i b_i$

či vzdialeností, ako sú euklidovská, Hammingova či editačná.

Štúdia [41] zmieňuje nevýhody kosínusovej miery a skalárneho súčinu, ako sú nezávislosť na počte výskytov daného termu a malú pravdepodobnosť odhalenia situácie, pokiaľ je jeden dokument obsiahnutý v inom, väčšom. Navrhuje vlastný model nazývaný Relative Frequency Model, kde sa snaží brať do úvahy iba slová používané v oboch porovnávaných dokumentoch s približne rovnakou frekvenciou, a následne pomocou asymetrickej metódy „obsahuje“ vybrať maximum z hodnôt A obsahuje B, B obsahuje A. Práca [42] navrhuje mnoho rôznych stratégií výpočtu d_t, najlepšie výsledky dosahuje podľa autorov výpočet

$$d_t = \frac{1}{1+\log(1+|f_d - f_q|)} \cdot \sum_{t \in q \cap d} \frac{\frac{N}{f_t}}{1+|f_{d,t} - f_{q,t}|} \tag{3.4}$$

kde N je počet dokumentov v kolekcií, f_t je počet dokumentov obsahujúcich term t, f_d je počet termov v dokumente d, $f_{t,d}$ je počet výskytov termu t v dokumente d, a q a d označujú dva dokumenty. Dosiahnuté závery porovnáva s metódami založenými na šindľovaní, a vychádzajú mu približne rovnaké výsledky, mierne favorizujúce vektorový model. Práca [43] sa zaoberá spôsobom, ako čo najefektívnejšie vyhľadávať podobné vektory. Vzhľadom k tomu, že rozmer vektoru je rovný počtu slov v korpuse, táto úloha môže byť veľmi časovo náročná. Autori práce využívajú na komprimáciu vektorov Charikarovu metódu „simhash“ [44],

založenú na princípe tzv. locality sensitive hashing. Táto metóda, na rozdiel od klasického hashovania, kde aj dva takmer totožné vektory môžu mať úplne odlišné hashe, pracuje tak, že podobným vektorom priraďuje podobné hashe. Autori v práci ďalej skúmajú čo najefektívnejší spôsob ako porovnávať výsledné hashe na základe tzv. Hammingovej vzdialenosti, teda ako pre dané k nájsť hashe, ktoré sa od zadaného líšia maximálne na k pozíciách. Postup autorov práce [45] je spočítať frekvencie pre každý term, odstrániť termy s najvyššou a najnižšou frekvenciou, zoradiť ich podľa určitého kritéria a z výsledku spočítať hash, ktorý sa bude používať na porovnávanie. Táto metóda je bohužiaľ veľmi citlivá na zmenu niektorých slov, čo je typická obrana plagiátorov pred odhalením, tento nedostatok sa pokúsila odstrániť nadväzujúca štúdia [46].

3.4. Metódy založené na použití internetových vyhľadávačov

Motivácia k použitiu takýchto metód je pomerne priamočiara. Takmer každý pedagóg už dostal výbornú esej od podpriemerného študenta a pokiaľ mal určité podozrenie na plagiátorstvo, skúsil zadať pár vybraných fráz do internetového vyhľadávača, takže sa sám stal akýmsi systémom na vyhľadávanie dokumentov. Metódy založené na použití internetových vyhľadávačov sú vlastne iba automatizáciou tohto postupu.

Tento prístup má mnoho výhod, asi najvýraznejšou je, že odpadá množstvo práce spojenej s prípravou a inicializáciou korpusu a taktiež s jeho následným udržiavaním. Ďalšou výhodou je presun záťaže (výpočtová kapacita, atď.) priamo na užívateľa. Nevýhodou je, že v prípade korpusu dokumentov, ktoré by nemali byť, alebo jednoducho nie sú, verejne dostupné, tento prístup zlyháva, pretože vyhľadávač samozrejme môže používať iba dokumenty, ktoré už má indexované.

Práca [33] používa pohyblivé okno danej dĺžky a pri rozhodovaní o jednotlivých dotazoch používa ako kritérium priemerný počet písmen v slove, na základe úvahy že v „učenejšom“ texte sa používajú dlhšie slová, a je oveľa pravdepodobnejšie že takýto text bude aj predmetom plagiátorstva. Práca zmieňuje aj ďalšie obmedzenie, a to 1000 dotazov na deň pri použití Google API. Toto už však neplatí a nové Google AJAX Search API je možné použiť bez obmedzenia.

Generovaním dotazov na základe prekrývajúcich sa n- gramov sa zaoberajú autori práce [35], k algoritmu pridávajú ešte jeden krok, ktorým je spočítanie podobnosti nájdených výsledkov so zadaným dokumentom, čo samozrejme postup značne spresňuje. Niektoré komerčné nástroje sa tiež vydávajú týmto smerom, napríklad [36].

Ďalšou možnosťou je nevyužívať internetové vyhľadávače spôsobom popísaným vyššie, ale integrovať detekciu duplicít priamo do konkrétneho vyhľadávača, ideálne v kombinácií s niektorou s vyššie spomenutých metód. Tým pádom získame výhody spojené s prípravou a inicializáciou korpusu, a odstránime nevýhodu nemožnosti indexovať neverejné dokumenty. O tento prístup sa snaží napríklad práca [30], či mnohé komerčné nástroje, napríklad [37] či [38].

3.5. Ostatné metódy

Práca [47] využíva na detekciu duplicitných dokumentov algoritmus na porovnávanie reťazcov založený na sufixových stromoch, vytváraných modifikovaným Ukkonenovým algoritmom. Svoje výsledky však prezentuje na veľmi malej kolekcií dokumentov. Štúdia [48] sa pokúša zistiť „ako ľudia píšu“ za pomoci štýlu (časté slová v dokumente, použitie rovnakých funkčných slov, atď.) a slovníku (tf-idf model, atď.) a na základe toho sa snaží určiť plagiarizované dokumenty. Ako testovaciu kolekciu dokumentov používa rôzne preklady svetovej literatúry do anglického jazyka. Typická zbraň plagiátorov proti odhaleniu je modifikácia časti alebo častí textu. Proti tomu bojujú autori práce [49], ktorí porovnávajú dokumenty na rôznych úrovniach (slová, vety, odstavce) za pomoci Levenshteinovej vzdialenosti, čím dokážu efektívne odhaľovať vsunutie, vymazanie, či zmenu lingvistickej jednotky na danej úrovni. Štúdia [53] sa snaží o podobný prístup, na úrovni slov rieši zámenu, vsunutie a vymazanie, prípadne preusporiadanie, na úrovni viet ich rozdelenie, či spájanie. Na základe počtu podobných viet potom stanovuje podobnosť celých dokumentov. Zámenu slov sa snaží riešiť na vyššej úrovni ako je jednoduchá Levenshteinova vzdialenosť. Predstavme si nasledujúce tri vety:

(1) Jedlo pripravujeme pomaly
(2) Jedlo varíme pomaly
(3) Jedlo jeme pomaly

Pri použití Levenshteinovej vzdialenosti by všetky páry viet mali rovnaké ohodnotenie, pričom je ale jasné, že vety (1) a (2) sú si navzájom podobnejšie ako vety (1) a (3). Preto sa zavádzajú tzv. korelačné faktory slov, ktoré sa spočítajú dopredu, a priraďujú rôznym druhom podobných slov (synonymá, hypernymá, hyponymá, atď.) adekvátne váhy. Ešte ďalej ide práca [50], ktorá sa snaží o istú formu strojového chápania prirodzeného jazyka. Popísaný algoritmus by sa dal rozdeliť do troch fáz, prvou je zámena slov na základe korelačných faktorov slov (napr. WordNet). Následne používa syntaktickú analýzu na pochopenie syntaktickej štruktúry, a sémantickú analýzu na pochopenie významu. Sami autori však priznávajú, že ich algoritmus je veľmi pomalý a mal by byť použitý iba v prípade silného podozrenia na plagiátorstvo.

Štúdia [51] porovnáva dokumenty na základe viet, a navrhuje rôzne modely na zisťovanie ich podobnosti (prekryv slov, tf-idf model, pravdepodobnostné modely, atď.). Na základe dosiahnutej podobnosti potom zostavuje celkové skóre pre podobnosť dokumentov. Práca [52] navrhuje detegovať plagiáty za pomoci metódy nazvanej „Singular Value Decomposition“ (SVD). Metóda extrahuje n - gramy zvolenej dĺžky z dokumentu a odstráni príliš časté, respektíve príliš vzácne hodnoty. Následne použije latentnú sémantickú analýzu (LSA) na získanie podobnosti medzi jednotlivými n – gramami. Podobnosť dokumentov je potom daná počtom spoločných n – gramov. Autori štúdie [53] vyvinuli hybridný algoritmus založený na kombinácií upraveného Levenshteinovho a Smith – Watermanovho algoritmu. Ako lingvistickú jednotku používajú slová, najskôr pomocou Levenshteinovho algoritmu rozdelia maticu (vzniknutú pri behu algoritmu) na menšie regióny, na ktoré následne použijú zjednodušený Smith – Watermanov algoritmus. Autori však sami uvádzajú, že metóda (zatiaľ) nie je vhodná pre veľké kolekcie dokumentov vzhľadom na časovú náročnosť Smith – Watermanovho algoritmu.

3.6. Postupy používané vo väčšine metód

Niektoré postupy používané pri detekcii plagiátov sa používajú vo väčšine vyššie spomenutých metód. Prevažne sa jedná o postupy použité v úvodnej fáze algoritmov, pri príprave textu na ďalšie spracovanie. Patrí sem:

- **Odstránenie príliš častých lingvistických jednotiek.** Ak uvažujeme napríklad slová, tak príliš časté slová, ako sú spojky, predložky, či v anglickom jazyku členy, nie sú vzhľadom k sémantickej informácií obsiahnutej v dokumente vôbec dôležité. Takéto slová sa nazývajú „stop words“, a samozrejme, množina týchto slov je jazykovo závislá. Odstránenie príliš častých lingvistických jednotiek sa používa najmä v súvislosti so slovami, ale ani vety či odseky nie sú výnimkou. Takýmto odstránením sa znížia priestorové nároky a zníži časová náročnosť algoritmu.
- **Odstránenie príliš zriedkavých lingvistických jednotiek.** Pokiaľ sa daná lingvistická jednotka nachádza v kolekcií príliš malý počet krát, pravdepodobne nie je pre korpus príliš podstatná (napr. preklepy, a pod.) a je teda možné ju vynechať. Odstránením sa opäť znížia priestorové nároky a časová náročnosť algoritmu.
- **Odstránenie termov so špeciálnymi znakmi.** Opäť sa jedná o skutočnosť, že nie je pravdepodobné, aby term, ktorý obsahuje nealfanumerické znaky, niesol nejakú sémantickú informáciu (ak neberieme do úvahy dokumenty matematického či technického rázu). Napríklad internetové stránky obsahujú mnoho tagov a entít, a je určite rozumné takéto termy neuvažovať.
- **Odstránenie príliš krátkych lingvistických jednotiek.** Uvažujme nasledujúce výňatky:

  „Vraví sa, že čas je nepriateľom lásky. Nie je to pravda!“

  „Traduje sa, že blond farba vlasov pochádza zo slova biely, bledý. Nie je to pravda!“

  Dané dva výňatky určite pojednávajú o rozdielnych témach, napriek tomu obsahujú rovnakú vetu. Takéto krátke vety, respektíve iné lingvistické jednotky, nenesúce žiadnu sémantickú informáciu je dobré neuvažovať, aby nedošlo k zbytočnému nárastu falošných pozitív.
- **Odstránenie príliš dlhých lingvistických jednotiek.** Štúdia [45] zmieňuje odstránenie slov dlhších ako 25 znakov, ktoré zredukovalo veľkosť slovníka na cca. 60% a dokonca zvýšilo presnosť pri vyhľadávaní duplicitných dokumentov.
- **Prevod celého textu na malé (resp. veľké) písmená.** Takýto prevod pomáha značne zmenšiť veľkosť slovníku, ako aj spresniť výsledky, keďže

jednou z častých techník používanou plagiátormi je rozdelenie, respektíve spájanie viet. Uvažujme nasledujúce fragmenty:

(1) ...a číslo je deliteľné n. Potom môžeme povedať, že ...

(2) ...a číslo je deliteľné n – potom môžeme povedať, že...

Ak by sme text slepo previedli na postupnosť termov, z ktorej ďalej vytvoríme hash, bez prevodu textu by dané dva hashe neboli rovnaké, pričom je jasné že dané dva fragmenty pojednávajú o tej istej veci používajúc rovnakú terminológiu.

- **Redukcia slov na koreňovú formu (stemming)**. Ďalšou typickou zbraňou plagiátorov je zmena tvaru slova, či už sa jedná o zmenu času u slovies („...a tým sme vypočítali, že...“ verzus „...a tým vypočítame, že...“), pridanie predpony či koncovky, zmena pádu a podobne. Napríklad v slovenčine sa všetky slová „ženba, oženiť, ženský, vyženiť“ redukujú na koreňovú formu -žen-. Vo väčšine vyššie uvedených prác sa používa algoritmus, ktorý vyvinul M. Porter [55]. Tento algoritmus však funguje správne iba pre texty v anglickom jazyku, nakoľko úloha redukcie slov na koreňovú formu je už zo svojej podstaty jazykovo závislá.
- **Lematizácia.** Vo svojej podstate sa jedná o proces podobný stemmingu, je však značne zložitejší. Kým stemming koreňovú formu slova väčšinou iba aproximuje, lematizácia sa snaží vždy získať skutočnú koreňovú (resp. normalizovanú) formu [56], snaží sa napríklad o pochopenie kontextu slova. Výsledky štúdie [56] naznačujú, že pre morfologicky jednoduché jazyky, ako napríklad angličtina, dosahuje lepšie výsledky stemming, ale pre morfologicky bohaté jazyky je vhodnejšie použiť lematizáciu.

4. Prostredie pre implementáciu

Ako prostredie pre implementáciu bol zvolený internetový vyhľadávač Egothor 2, a to najmä kvôli svojej variabilite, výkonu a možnosti transakčného spracovania. V neposlednom rade hrala úlohu aj skutočnosť, že zdrojový kód tohto vyhľadávača je voľne k dispozícii, čo ho činí veľmi vhodným na použitie v akademickom prostredí. Egothor je vytvorený v programovacom jazyku Java.

Ako takmer každý vyhľadávač, aj Egothor je rozdelený na dve viac – menej samostatné časti. Prvou je robot, určený na sťahovanie navštívených internetových stránok, a druhou je vyhľadávací index, čo je dátová štruktúra určená na rýchle vyhodnocovanie zadaných dotazov. Oddelenie týchto dvoch častí mimo iné znamená, že Egothor je možné využiť ako klasický internetový vyhľadávač, tak ako aj vyhľadávač pre uzavretú množinu dokumentov, v ktorej sú prípadné zmeny vykonávané ručne. V ďalšom texte nás bude zaujímať iba vyhľadávací index, keďže spôsob fungovania robota pre túto prácu nie je dôležitý. Podrobnejší popis Egothoru je k dispozícii v originálnej dokumentácii [57], popis vyššie spomenutého transakčného spracovania potom v [58].

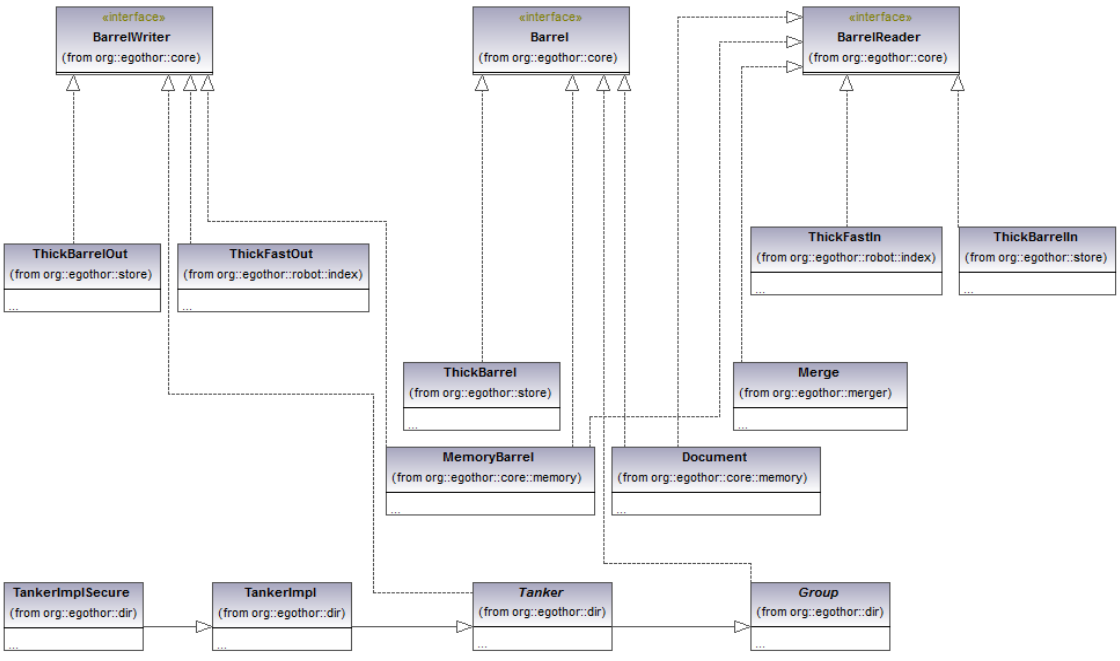
4.1. Tvorba vyhľadávacieho indexu

Základnou stavebnou jednotkou vyhľadávacieho indexu je rozhranie Barrel, respektíve triedy, ktoré toto rozhranie implementujú. Barrel je v zásade plnohodnotným vyhľadávacím indexom sám osebe, jeho obsahom je slovník termov nachádzajúcich sa v danom Barreli, invertovaný zoznam ku každému termu a metadáta jednotlivých dokumentov. Hlavnou funkciou rozhrania Barrel je vrátenie výsledkov na daný dotaz. Rozhranie Barrel implementujú napríklad triedy Document pre samostatný dokument, ThickBarrel pre barel uložený v externej pamäti, MemoryBarrel pre barel uložený v hlavnej pamäti a iné. Odteraz budeme termínom Barrel (s veľkým písmenom na začiaku slova) označovať rozhranie, termínom barel (s malým písmenom na začiatku slova a jedným r) označovať ľubovoľnú triedu implementujúcu toto rozhranie.

Samozrejme, jednotlivé barely je nutné nejakým spôsobom načítať, respektíve zapisovať, či už do hlavnej alebo externej pamäti. Na to slúžia rozhrania

BarrelReader, respektíve BarrelWriter a ich implementácie. Rozhranie BarrelReader implementujú medziiným triedy ThickFastIn a ThickBarrelIn, ktoré slúžia na načítanie barelu z externej pamäti, trieda Merge, ktorá slúži na spájanie jednotlivých barelov, a triedy Document a MemoryBarrel, ktoré sú zároveň i implementáciami rozhrania Barrel. Z implementácií rozhrania BarrelWriter sú dôležité najmä ThickBarrelOut a ThickFastOut, slúžiace na zápis do externej pamäte.

Jednotlivé barely sú síce samostatné vyhľadávacie indexy, je ale nutné reprezentovať ich navonok ako jeden celistvý vyhľadávací index. Na to slúži trieda Group, ktorá je jednoducho iba zoskupením samostatných barelov, tváriacich sa ako jeden barel, a jej potomkovia Tanker, TankerImpl a TankerImplSecure. Trieda Tanker má oproti triede Group navyše schopnosť pripojenia nového barelu, trieda TankerImpl pridáva jednotlivé rutiny potrebné pre prácu s vyhľadávacím indexom a konečne trieda TankerImplSecure pridáva bezpečné spracovanie pri použití viacerých vlákien. Odteraz budeme termínom Tanker (s veľkým písmenom na začiatku slova) označovať rozhranie, termínom tanker (s malým písmenom na začiatku slova) označovať ľubovoľnú triedu implementujúcu toto rozhranie. (Neúplný) diagram znázorňujúci vzťahy jednotlivých rozhraní a ich implementácií znázorňuje obrázok 4.1.



Obr. 4.1- Vzťahy rozhraní a ich implementácií

Algoritmus na tvorbu vyhľadávacieho indexu z jednotlivých dokumentov je v podstate veľmi jednoduchý. Najprv si ukážeme tvorbu barelu z dokumentov. Majme kolekciu dokumentov D, potom:

```
1  for (Document d : D){
2       pridaj d do MemoryBarrel m
3  }
4  Vytvor novú implementáciu triedy BarrelWriter bw
5  Zapíš m do bw
```

Algoritmus 4.1 – Tvorba barelu z dokumentov

Takýchto barelov je možné vytvoriť ľubovoľný počet. Pokiaľ už sú z kolekcie dokumentov vytvorené barely, je tvorba celého vyhľadávacieho indexu, teda Tankeru, veľmi priamočiara. Majme kolekciu barelov B, potom:

```
1  Vytvor tanker t
2  For (Barrel b : B){
3       Pridaj b do t pomocou volania metódy
   t.append()
4  }
5  Ulož zmeny vykonané na tankeri pomocou volania
   metódy t.commit()
```

Algoritmus 4.2- Tvorba indexu z barelov

Týmto spôsobom sme teda vytvorili plnohodnotný vyhľadávací index.

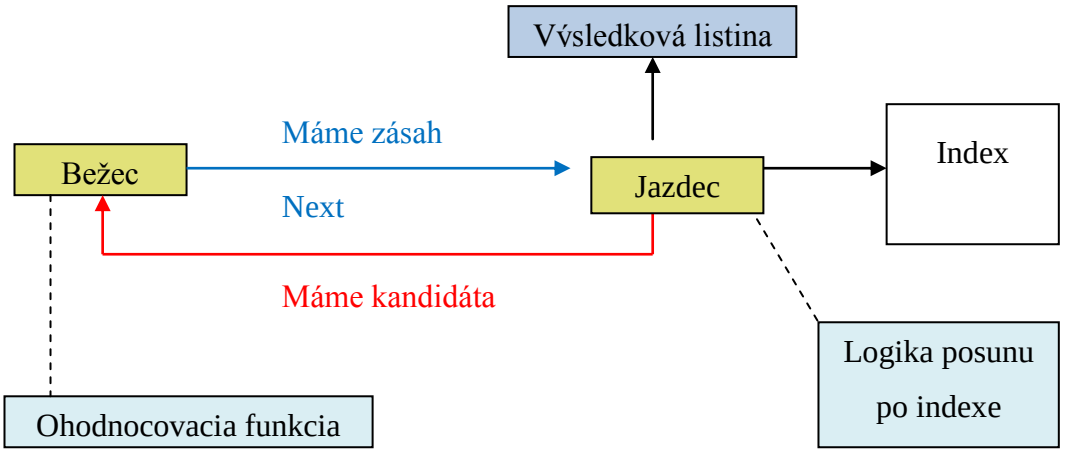
4.2. Filtre

Egothor ponúka podporu aj pre najrôznejšie filtre. Úloha filtra je vlastne veľmi jednoduchá, zo vstupného prúdu termov vytvorí nový prúd, nad ktorým boli vykonané požadované zmeny. Táto funkcionalita je výhodná napríklad pri predspracovaní textu, popísanom v kapitole 3.6. Keďže vstupom aj výstupom filtra je prúd termov, je možné filtre navzájom reťaziť, a tým dosiahnuť požadovanú úroveň spracovania textu pred zaradením dokumentu do vyhľadávacieho indexu. Medzi už implementované filtre patrí napríklad filter na prevod textu na malé

písmená, filter na odstránenie príliš častých (respektíve nevýznamových) slov, a ďalšie.

4.3. Dotazy nad vyhľadávacím indexom

Samozrejme, najdôležitejšou funkciou internetového vyhľadávača je riešenie zadaných dotazov. Pokiaľ vynecháme analýzu dotazu, t.j. zisťovanie, čo všetko sa má vyhľadávať a aké sú závislosti medzi jednotlivými termami v dotaze, sú pre vyhľadávanie najdôležitejšie objekty, nazývané v súlade s terminológiou použitou v [57], Jazdec a Bežec. Jazdec má v sebe zabudovanú logiku, ktorá mu umožňuje prechádzať údaje splňujúce podmienku dotazu, teda údaje ktoré sú potenciálnymi kandidátmi na zásah, Bežec je objektom ktorý je vlastne priamym spojením reprezentácie dotazu s Jazdcom. Bežec žiada Jazdca o kandidátov na zásah, a keď usúdi, že daný zásah je relevantný, pošle Jazdcovi túto informáciu, a ten následne zaradí daný zásah do výsledkovej listiny. Komunikácia medzi Jazdcom a Bežcom je znázornená na obrázku 4.2, prevzatého z [57].



Obr. 4.2- Komunikácia medzi jazdcom a bežcom prevzatá z [57]

Ponúka sa otázka, prečo vlastne rozdeľovať Jazdca a Bežca, keď by obe funkcionality mohol zastávať jediný objekt. Dôvodov pre to je mnoho, spomeňme napríklad situáciu, v ktorej sú vyhľadávací index a vyhodnocovacia funkcia na dvoch rôznych počítačoch, alebo situáciu, v ktorej je v systéme viac uzlov, ktoré ponúkajú Jazdca s rôznymi typmi rozšírenia, takže určite nie je možné reprezentovať Jazdca i Bežca ako jeden objekt. Viac modelových situácií aj s diskusiou sa nachádza v [57].

4.4. Programové rozhranie dotazu

Dotaz v programovom rozhraní tvorí stromová štruktúra tzv. Query objektov. Listom takéhoto stromu je typicky nejaká reprezentácia vyhľadávaného slova, vnútorným uzlom môžu byť napríklad logické operátory AND, OR, alebo NOT. Pre nás bude (ako uvidíme v ďalšom texte) najdôležitejším uzlom tzv. uzol skupiny (QGroup), ktorý môže mať ľubovoľný počet poduzlov. Uzlu skupiny sa dajú zapnúť rôzne príznaky, ktoré určujú spôsob vyhodnotenia všetkých poduzlov, to značí, či sú tieto vyhodnocované v móde OR, AND alebo NOT. Z historických dôvodov musí byť uzol skupiny taktiež koreňovým uzlom každej stromovej štruktúry reprezentujúcej dotaz v programovom rozhraní. Po pripravení celej stromovej štruktúry dotazu túto štruktúru predáme rozhraniu Jazdec/Bežec, ktoré sa postará o jej spracovanie. Pre bližšiu špecifikáciu programového rozhrania dotazu viď opäť [57].

5. Výber vhodnej metódy

Z kapitoly 3 zaoberajúcej sa prehľadom používaných metód je jasné, že možných spôsobov ako postupovať pri hľadaní duplicitných dokumentov je značné množstvo, do úvahy však pripadajú iba tie, ktoré spĺňajú, respektíve majú potenciál spĺňať ciele práce vymedzené v kapitole 2. Jedná sa najmä o minimalizáciu množstva falošných pozitív a falošných negatív, a o adekvátnu časovú náročnosť algoritmu, tak, aby očakávaná doba odpovede neprekročila zvolený čas.

5.1. Výber typu metódy

Metódy založené na použití internetových vyhľadávačov môžeme zavrhnúť hneď na začiatku, a to jednak preto, že je nelogické aby detekcia plagiátov integrovaná do konkrétneho vyhľadávača používala iné vyhľadávače, jednak preto, že pre využitie iných internetových vyhľadávačov je nutným predpokladom skutočnosť, že daný vyhľadávač má zaindexované dokumenty, pre ktoré platí, že zadaný dokument môže byť ich plagiátom. To však často nebýva, a ani nemôže byť pravda, keďže bohužiaľ ani pre akademické prostredie stále neplatí, že všetky práce sú voľne online dostupné.

Ďalej môžeme vylúčiť všetky metódy, ktorých časová náročnosť znemožňuje cieľ očakávanej doby odpovede 10 sekúnd, takže sa v zásade môžeme sústrediť na metódy založené na šindľovaní a metódy založené na vektorovom modeli. K záveru, že z hľadiska presnosti detekcie a časovej náročnosti sú v podstate najlepšie tieto dve metódy prišla napríklad aj štúdia [59] či práca [30]. Práca [30] ďalej hodnotí vhodnosť oboch metód, a konštatuje, že aj keď sa metódy založené na vektorovom modeli javia ako vhodnejšie k preskúmaniu, ich hlavným nedostatkom je nutnosť komprimácie modelu kvôli dĺžke dokumentu. Zo snáh o odstránenie tohto nedostatku bolo zatiaľ úspešné iba využitie Charikarovej metódy simhash, tá ale je už v literatúre dostatočne popísaná.

Metódy založené na šindľovaní na druhej strane oveľa viac vyhovujú definícií plagiátorstva tak, ako ho chápe táto práca a to tak neformálnej, ako aj formálnej, z podkapitoly 2.2. Zjednodušene povedané, metódy založené na vektorovom modeli ignorujú poradie slov, a teda považujú dva dokumenty za

podobné v prípade, že obsahujú veľké množstvo rovnakých slov, čo ale nemusí nutne znamenať, že sú to plagiáty, ale skôr skutočnosť, že pojednávajú o rovnakej téme. To znamená, že sa skôr hodia na detekciu duplicitných (internetových) stránok, prípadne na detekciu podobných stránok (viď funkcia „Podobné stránky“ vo vyhľadávači Google), ale sú už menej vhodné na detekciu plagiátov. Vedecké práce, ktoré sú písané na rovnakú tému, rozhodne nie sú zriedkavým javom, publikácie od jedného autora sú písané podobným štýlom a slovníkom, a podobne. Dokonca aj vo veľmi jednoduchom prípade, akým je modelová situácia, kedy vyučujúci zadá študentom prácu na rovnakú tému, tak vzhľadom k tomu, že téma je rovnaká, a študenti navštevujú rovnaký predmet, je pravdepodobné, že budú používať aj veľmi podobný slovník. Preto sa nám na detekciu plagiátov javia ako lepšie metódy založené na šindľovaní, ktoré reflektujú aj poradie slov. Používanie rovnakých fráz, viet, či dokonca väčších celkov totiž už vyvoláva vážne podozrenie na plagiátorstvo.

5.2. Jaccardova miera podobnosti

Veľmi zaujímavý postup je použitý v publikácií [60], kde autor vhodne kombinuje použitie Jaccardovej miery podobnosti so šindľovaním. Majme dokument D, a označme $T(D)$ množinu všetkých termov z dokumentu D. Potom Jaccardova miera podobnosti dokumentov D_1 a D_2 je definovaná ako veľkosť prieniku množín termov dokumentov, delená veľkosťou zjednotenia množiny termov dokumentov.

$$J(D_1, D_2) = \frac{|T(D_1) \cap T(D_2)|}{|T(D_1) \cup T(D_2)|} \tag{5.1}$$

Počítať Jaccardove koeficienty takýmto spôsobom, to jest naivným algoritmom, ktorý využíva priamo definíciu Jaccardovej miery by však bolo zbytočne časovo náročné a neefektívne, navyše, takýto naivný algoritmus by počítal Jaccardove koeficienty presne, čo nie je nutné.

Preto publikácia [60] popisuje tzv. pravdepodobnostný algoritmus, ktorého cieľom je zníženie časovej zložitosti algoritmu za cenu určitej chyby pri výpočte Jaccardovho koeficientu. Majme korpus dokumentov W, a nech všetky termy z W majú priradený jednoznačný číselný identifikátor $T = \{1,2,..,M\}$, kde M je počet termov v korpuse. Ďalej majme náhodne zvolenú permutáciu $\pi: T \to T$, a pre množiny X,Y, ktoré sú podmnožinou T označme $\pi(X) \subset T, \pi(Y) \subset T$ množinu

prvkov získaných použitím permutácie π na každý prvok množiny X, respektíve Y. Potom platí:

$$Pr_{\pi}(\min(\pi(X)) = \min(\pi(Y))) = \frac{|X \cap Y|}{|X \cup Y|} \tag{5.2}$$

Dôkaz tejto vety môžeme tiež nájsť v [60]. Použitie vety v algoritme na detekciu duplicitných dokumentov tkvie v tom, že ak zvolíme väčší počet takýchto permutácií, tak Jaccardov koeficient spočítame ako pomer počtu permutácií, v ktorých sa dané dva dokumenty zhodujú k počtu všetkých permutácií. Množinu týchto náhodne vybraných permutácií označme Π. [60] potom uvádza aj konkrétny algoritmus na výpočet podobnosti všetkých dvojíc dokumentov z korpusu W:

```
1  for (Permutácia π : Π){
2        for (Dokument D : W){
3              s = min(π(T(D)))
4              Zapíš dvojicu (s,D) do súboru f_π
5        }
6        Zotrieď f_π  podľa s – výsledkom sú súvislé
         bloky s pevným s obsahujúce všetky D
7        for (s : f_π){
8              for ((D_1,D_2) : všetky dvojice dokumentov
                    z jedného bloku){
9                    Zapíš (D_1,D_2) do g_π
10             }
11       }
12       Zotrieď g_π podľa (D_1,D_2)
13 }
14 Merge všetkých g_π do g v poradí (D_1,D_2), spočítaj
   počet   (D_1,D_2) záznamov, vydeľ tento počet |Π|
```

Algoritmus 5.1- Výpočet podobnosti všetkých dvojíc dokumentov z korpusu

Tento algoritmus je teraz potrebné prispôsobiť pre detekciu plagiátov. Doteraz sme uvažovali, že všetky termy z korpusu W budú mať priradený jednoznačný číselný identifikátor. Túto úvahu však môžeme rozšíriť, a priradiť

jednoznačný číselný identifikátor každému šindľu vzniknutému z korpusu W. Potom, pokiaľ budeme postupovať rovnako ako v predchádzajúcom prípade, dostaneme Jaccardovu mieru podobnosti dvoch dokumentov, ale už so zohľadnením poradia slov v dokumente. Takýto postup je takisto popísaný v [60], a ďalej rozvinutý v práci [30], kde autorka ešte rozdeľuje dokument na menšie lingvistické jednotky, konkrétne na odseky a vety, a počíta Jaccardove koeficienty medzi týmito jednotkami, čím detekciu značne spresňuje. (Určité spresnenie by bolo možné dosiahnuť aj zvyšovaním počtu vybraných permutácií, tam ale nemáme zaručenú rovnomernú distribúciu vybraných miním, a môže sa teoreticky stať že všetky minimá pripadnú na jeden konkrétny, respektíve malý počet konkrétnych šindľov). Navyše je takýto prístup plne v súlade s definíciou plagiátu z podkapitoly 2.2. Práca [30] prezentuje nasledujúci algoritmus:

```
1  for (Permutácia π : Π){
2        for  (Dokument D : W){
3             for (Lingvistická jednotka d_i : D){
4                  r = min(π(S(d_i, k)))
5                  Zapíš dvojicu (r,d_i) do súboru f_π
6             }
7        }
8        Zotrieď f_π  podľa r, d_i
9        Agreguj záznamy z f do (r,MJ_r), kde MJ_r  je
         množina   jednotiek s reprezentantom r
10       for (Minimum r : f_π){
11            for (Jednotka d_1, d_2 : MJ_r){
12                 Zapíš dvojicu (d_1, d_2) do súboru g_π
13            }
14       }
15       Zotrieď súbor g_π  podľa (d_1, d_2)
16 }
17 g = merge všetkých súborov g_π
18 Agreguj záznamy z g do (d_1, d_2, J)
```

```
19 return (d1, d2, J / |Π|)
```

Algoritmus 5.2- Výpočet podobností pre dvojice dokumentov rozdelené na jednotky

5.3. Prispôsobenie pre detekciu 1:n

Predchádzajúce dva algoritmy (5.1 a 5.2) sú typu $n:n$, to znamená že počítajú podobnosť všetkých dvojíc dokumentov v korpuse medzi sebou. Vzhľadom k tomu, že cieľom tejto práce je detekcia plagiátov typu $1:n$, je potrebné tieto algoritmy upraviť tak, aby toho boli schopné. To docielime tak, že algoritmus rozdelíme na dve fázy, takzvanú inicializačnú, kde sa pokúsime „predpočítať" čo najviac údajov, čiže vlastne vytvoriť si nejaký index, a vyhľadávaciu, kde tieto údaje budeme využívať. Cieľom je samozrejme minimalizácia časovej zložitosti vyhľadávacej fázy, respektíve fázy vyhodnotenia dotazu. Práca [30], ako bolo spomenuté v kapitole 3.2, taktiež využíva internetový vyhľadávač Egothor, a vzhľadom k tomu, že tento vyhľadávač stavia index za pomoci barelov, ako bolo uvedené v kapitole 4.1, uvádza algoritmus pre inicializačnú fázu pre jeden barel, ktorý budeme označovať B:

```
1  for (Permutácia π : Π){
2        for  (Dokument D : B){
3             for (Lingvistická jednotka di : D){
4                  r = min(π(S(di, k)))
5                  Zapíš dvojicu (r,di) do súboru pomfπ
6             }
7        }
8        Zotrieď pomfπ  podľa (r, di)
9        fπ = merge fπ a pomfπ
10 }
```

Algoritmus 5.3 – Inicializačná fáza pre jeden barrel

Nasledujúce odhady z práce [30] sú uvedené preto, aby bolo jasné, z čoho sme vychádzali a ako sme postupovali pri odhadovaní časovej zložitosti v kapitole 5.4. Sekundárny dôvod uvedenia algoritmov i odhadov je možnosť porovnania,

akým výrazným prínosom z hľadiska časovej zložitosti je použitie vhodných dátových štruktúr a skutočná integrácia do vyhľadávača (opäť viď. kapitola 5.4).

Zložitosť inicializačnej fázy teda odhaduje práca [30] nasledovne: „Odhadnime zložitosť tejto fázy najprv pre priechod jednou pevnou permutáciou. Zložitosť na riadkoch 2-7 je lineárna vzhľadom k počtu lingvistických jednotiek vo všetkých b dokumentoch barelu B, čo môžeme zapísať ako $O(bJ_d)$, kde J_D je priemerný počet jednotiek v dokumente. Na riadku 8 triedime postupnosť dĺžky bJ_D, čo dáva zložitosť $O(bJ_d \log(bJ_D))$. Na riadku 9 zatrieďujeme postupnosť dĺžky bJ_D do postupnosti dĺžky nJ_D, kde n je aktuálny počet dokumentov v korpuse, čo dáva zložitosť $O((n+b)J_D)$. Celková zložitosť je teda $O(bJ_d \log(bJ_D) + nJ_D)$. Pre $p = |\Pi|$ permutácií je teda zložitosť $O(pbJ_d \log(bJ_D) + pnJ_D)$. Zložitosť celej inicializačnej fázy pre vytvorenie indexu zo všetkých dokumentov korpusu W veľkosti N odvodíme nasledujúcim postupom. Je potrebné vykonať $\frac{N}{b}$ priechodov zložitosti $O(pbJ_d \log(bJ_D) + pnJ_D)$, kde $n = 0,1,..,N-b$. Prvý člen zložitosti nezávisí na n, a prispieva teda k celkovej zložitosti hodnotou $O(pNJ_d \log(bJ_D))$. Druhý člen na n závisí, a dá sa sčítať podľa vzorca pre aritmetickú postupnosť, takže dostávame $O(\frac{pJ_DN^2}{b})$. Celová zložitosť je teda $O(pNJ_d \log(bJ_D) + \frac{pJ_DN^2}{b}) = O(\frac{pJ_DN^2}{b})$, pretože $p \ll N, b \ll N, J_D \ll N$. A pretože p, b a J_D môžeme považovať za konštanty, je celková zložitosť kvadratická voči počtu dokumentov v korpuse.“ [30].

Fáza vyhodnotenia dotazu prebieha nezávisle od inicializačnej fázy, práca [30] popisuje algoritmus nasledovne: „Najskôr sú obvyklým spôsobom vypočítaní reprezentanti pre lingvistické jednotky dokumentu z dotazu, ďalej sa v súbore f_π vyhľadajú záznamy s reprezentantmi zhodnými s niektorými reprezentantmi dotazovaného dokumentu. Potom sa do súboru g_π zapíšu všetky dvojice jednotiek so zhodnými reprezentantmi tak, že prvá jednotka z dvojice je z korpusu a druhá je z dotazovaného dokumentu. Na základe zlievania všetkých súborov sú potom spočítané odhady Jaccardovych koeficientov.“. Označme dokument z dotazu Q, pseudokód algoritmu vyzerá potom takto:

```
1  for (Permutácia π : Π){
2       for (Lingvistická jednotka d_i : Q){
3            r = min(π(S(d_i, k)))
4            Zapíš dvojicu (r,d_i) do súboru pomf_π
5       }
6       Zotrieď súbor pomf_π  podľa (r,d_i)
7       Zo súboru f_π opíš do súboru h_π dvojice (r,d_i)
        kde r∈pomf_π
8       Agreguj záznamy z h_π do HMJ_r, kde HMJ_r je
        množina jednotiek s reprezentantom r
9       Agreguj záznamy z pomf_π do (r,MJ_r), kde MJ_r je
        množina jednotiek s reprezentantom r
10      For (Minimum r : pomf_π) {
11           For (Jednotka d_1 : HMJ_r){
12                For (Jednotka d_2 : MJ_r){
13                     Zapíš dvojicu (d_1, d_2) do súboru
                       g_π
14                }
15           }
16      }
17      Usporiadaj súbor g_π podľa (d_1, d_2)
18 }
19 g = merge všetkých súborov g_π
20 Agreguj záznamy z g do (d_1, d_2, J)
21 return (d_1, d_2, J / |Π|)
```

Algoritmus 5.4 – Algoritmus vyhodnotenia dotazu

Práca [30] odhaduje časovú zložitosť tohto algoritmu nasledovne:

„Odhadnime časovú zložitosť najprv pre pevnú permutáciu. Zložitosť na riadkoch 2-5 je lineárna vzhľadom k počtu lingvistických jednotiek dotazovaného dokumentu, čo sa dá napísať ako $O(J)$, kde J je počet lingvistických jednotiek v dokumente. Na riadku 6 je potrebné zotriediť postupnosť dĺžky J, čo dáva

zložitosť $O(J \log(J))$. Na riadku 7 je potrebné prejsť celý súbor f_π veľkosti $O(NJ_D)$, kde N je počet dokumentov v korpuse, a J_D je priemerný počet jednotiek v dokumente. Súbor h_π i agregovanie súborov h_π a pomf_π, sú v algoritme iba pre názornosť, a v skutočnej implementácií sa nenachádzajú, môžeme teda riadky 7-9 z úvahy vynechať.

Odhad zložitosti riadkov 10-16 je problematický, zložitosť tejto časti je lineárna voči počtu dvojíc vypísaných do súboru g_π , a tento počet je závislý na tom, koľkým jednotkám korpusu sú jednotky z dotazovaného dokumentu podobné. Triviálny, ale značne nadsadený odhad by bol počet možností pre prvú zložku krát počet možností pre druhú zložku, teda $O(NJ_DJ)$. Pre druhú zložku tento odhad ponecháme (pre úplnú kópiu bude táto hranica dosiahnuteľná), ale pre prvú ju upravíme pomocou odhadu R maximálneho počtu duplikátov ľubovoľnej lingvistickej jednotky v korpuse (tento odhad ale bohužiaľ nepoznáme). Získame teda odhad $O(RJ)$. Na riadku 17 musíme túto postupnosť zotriediť, čo dáva zložitosť $O(RJ \log(RJ))$.

Dohromady teda pre pevnú permutáciu získavame zložitosť $O(J + \log(J) + NJ_D + RJ + RJ \log(RJ)) = O(NJ_D + RJ \log(RJ))$. Pre $p = |\Pi|$ permutácií je potom celková zložitosť riadkov 1-18 $O(pNJ_D + pRJ \log(RJ))$. Riadok 19 má zložitosť jednopriechodového zlievania p súborov priemernej dĺžky $O(RJ)$, čo je $O(pRJ)$. Riadky 20 a 21 si vystačia s lineárnym časom voči dĺžke súboru g, čo je opäť $O(pRJ)$. Celková zložitosť tejto fázy je teda $O(pNJ_D + pRJ \log(RJ))$. Keďže p, R a J_D môžeme považovať za konštanty, a polynóm v J môžeme zanedbať, je zložitosť lineárna voči počtu dokumentov v korpuse.“ [30]

5.4. Integrácia do internetového vyhľadávača

Algoritmus z predchádzajúcej práce má podľa nás dve zásadné nevýhody, ktoré znemožňujú jeho nasadenie v reálnej aplikácii. Aj keď sa práca snaží o integráciu do internetového vyhľadávača, v skutočnosti je úroveň tejto integrácie veľmi nízka. Navrhované algoritmy vôbec nevyužívajú žiadnu zo štruktúr internetového vyhľadávača, a tým pádom ani žiadnu z jeho výhod – v zásade sa celý proces integrácie obmedzil na vytvorenie akéhosi ďalšieho „indexu“ (súbor f_π z algoritmu 5.3), v ktorom sa vyhľadávanie deje pomerne primitívnym spôsobom, priechodom

celého súboru bez ohľadu na skutočnosť, že veľké časti tohto súboru nás absolútne nezaujímajú. Konkrétne sa jedná o riadok 7 algoritmu 5.4, kde sa pre každú permutáciu prechádzajú všetky záznamy zo súboru, pričom nás ale zaujímajú iba záznamy, ktoré majú reprezentanta patriaceho do množiny reprezentantov vytvorených pri spracovaní dotazovaného dokumentu. Druhá nevýhoda potom úzko súvisí s prvou, respektíve je jej priamym dôsledkom, a je to fakt, že časová zložitosť algoritmu je jednoducho príliš veľká na reálne použitie, a takisto spracovávaním nepotrebných dát zbytočne rastú I/O nároky systému.

Oba tieto problémy sa pokúsime odstrániť skutočnou integráciou zvolenej metódy do internetového vyhľadávača. Je potrebné si totiž uvedomiť, že jednotlivé šindle môžeme chápať ako klasické termy – šindeľ totiž reprezentujeme ako hash zvolenej časti textu, čo je celé číslo, čo je postupnosť znakov, a teda s ním môžeme nakladať ako s klasickým termom. Nuž a podpora pre prácu s termami je už v (takmer každom) internetovom vyhľadávači zabudovaná v podobe invertovaného indexu, čo je štruktúra, kde sú zaindexované jednotlivé termy, a každý term má asociovaný zoznam identifikátorov dokumentov, v ktorých sa nachádza. Viac o invertovaných indexoch pojednáva napríklad [40] alebo [57].

Z povahy algoritmu však pre nás, ak chceme využiť invertovaný index, vyplýva ešte jeden problém, ktorý je nutné vyriešiť. Porovnávania totiž môžeme robiť iba v rámci reprezentantov jednej permutácie, inými slovami, ak sa aj dve dvojice (r, d_i) rovnajú, do výsledku ich môžeme zahrnúť iba v prípade, že sú z jednej permutácie. Práca [30] to rieši rozdelením „indexu“ na p častí (kde p je počet súborov) a pre každú permutáciu prechádza iba index patriaci k danej permutácií. Tento spôsob sa nám nezdá vhodný, nakoľko so zvyšovaním počtu permutácií by sa vždy zvýšil aj počet potrebných indexov. Ďalším faktorom je, že je žiaduce, aby integrácia čo najmenej narušila pôvodné schémy vyhľadávača. Preto je podľa nás lepším riešením pripojiť identifikátor permutácie k identifikátoru dokumentu v invertovanom zozname, takže v prípade, že algoritmus narazí na term ktorý do danej permutácie nepatrí, nebude ho brať do úvahy.

Poslednou väčšou úpravou algoritmu je vnorenie cyklu cez všetky permutácie dovnútra ostatných cyklov. V predchádzajúcich algoritmoch to musel byť vonkajší cyklus kvôli tomu, že index bol rozdelený na viacero častí za účelom

dosiahnutia porovnávania vybraných reprezentantov iba v rámci jednej permutácie, toto rozdelenie sme však odstránili, a v takom prípade je programátorsky príjemnejšie pracovať s cyklami v obrátenom poradí. Algoritmus pre inicializačnú časť vyzerá teda nasledovne:

```
1  for (Dokument D : B){
2        for (Lingvistická jednotka di : D){
3              for (Permutácia p : Π){
4                    r = min(π(S(di, k)))
5                    Ulož trojicu (r, di, p) do pamäte
6              }
7        }
8        Postupnosť q = Zotrieď (r, di, p) postupne
         podľa r, di, p
9        Zatrieď q do invertovaného indexu
10 }
```

Algoritmus 5.5 – Upravený algoritmus pre inicializačnú časť

Odhadnime teraz zložitosť inicializačnej fázy. Zložitosť na riadkoch 2-7 sa dá zapísať ako $O(pJ_D)$, kde p je počet permutácií a J_D je priemerný počet lingvistických jednotiek v dokumente. Na riadku 8 triedime postupnosť dĺžky pJ_D, čo dáva zložitosť $O(pJ_D \log(pJ_D))$. Odhad zložitosti riadku 9 je veľmi problematický, keďže veľmi záleží aj na konkrétnej implementácií invertovaného indexu, a to najmä na implementácií slovníku termov. V prípade, že je však invertovaný index implementovaný pomocou niektorého hashovacieho algoritmu, odohráva sa nájdenie príslušného termu v konštantnom čase, a v tom prípade sa jedná o zatriedenie každého z pJ_D záznamov do už zotriedeného zoznamu. Pokiaľ označíme R maximálny počet duplicitných jednotiek v dokumente, potom tento zoznam môže mať dĺžku maximálne pR – to preto, že teoreticky sa nám môže stať že jeden term bude vybraný ako reprezentant pre každú jednu permutáciu. Zložitosť riadku 9 potom bude $O(p^2RJ_D)$. Zložitosť celého algoritmu teda môžeme napísať ako $O(bpJ_D + bp^2RJ_D)$.

Zložitosť celej inicializačnej fázy pre vytvorenie indexu zo všetkých dokumentov korpusu W veľkosti N odvodíme podobne ako pre algoritmus 5.3. Je potrebné vykonať $\frac{N}{b}$ priechodov zložitosti $O(bpJ_D + bp^2RJ_D)$, a vzhľadom k tomu, že ani jeden člen zložitosti nezávisí na n, výsledná zložitosť bude mať tvar $O(NpJ_D + Np^2RJ_D)$. No a keďže p, R a J_D môžeme považovať za konštanty (viď odhad zložitosti predchádzajúcich algoritmov), je celková zložitosť lineárna vzhľadom k počtu dokumentov v korpuse.

Algoritmus pre vyhľadávaciu časť bude vyzerať nasledovne:

```
1  for (Lingvistická jednotka di : Q){
2        for (Permutácia p : Π){
3             r = min(π(S(di, k)))
4             Ulož trojicu (r, di, p) do pamäte
5        }
6  }
7  Postupnosť q = Zotrieď (r, di, p) postupne podľa r,
   di, p
8  for (Reprezentant (r, di, p) : q) {
9        Nájdi term r v invertovanom indexe
10       Postupnosť z = Prejdi celý zoznam
   prislúchajúci  danému termu a vyber relevantné
   záznamy
11 }
12 Zotrieď z podľa (di, p)
```

Algoritmus 5.6 – Upravený algoritmus vyhodnotenia dotazu

Odhadnime zložitosť vyhľadávacej časti. Zložitosť na riadkoch 1-6 je lineárna vzhľadom k počtu lingvistických jednotiek v dokumente a počtu permutácií, takže ju môžeme odhadnúť ako $O(pJ_D)$. Na riadku 7 triedime postupnosť dĺžky pJ_D, takže dostávame zložitosť $O(pJ_D \log(pJ_D))$. (Tento krok nie je nutný, ale v prípade že sa stane, že sa na rovnakého reprezentanta namapuje viac lingvistických jednotiek, je výhodnejšie mať túto postupnosť zotriedenú. Takisto je to výhodnejšie v prípade, že slovník termov nemáme implementovaný pomocou

niektorého hashovacieho algoritmu, ale napríklad pomocou index-sekvenčného algoritmu.) Riadok 9 má v prípade, že je slovník termov implementovaný pomocou hashovacieho algoritmu konštantnú zložitosť, a riadok 10 má zložitosť lineárnu vzhľadom k možnému počtu záznamov v zozname prislúchajúcemu jednému termu, ktorý sme v odhade zložitosti pre inicializačnú fázu odhadli ako pR. Celková zložitosť riadkov 8-11 je teda $O(p^2RJ_D)$. Odhad zložitosti riadku 12 je komplikovanejší, ale teoreticky môžeme predpokladať, že postupnosť z bude mať maximálnu dĺžku pRJ_D, čiže maximálny počet duplicitných jednotiek v korpuse krát priemerný počet lingvistických jednotiek v dokumente (t.j. každá lingvistická jednotka z dokumentu Q bude mať v korpuse maximálne R podobných jednotiek) krát počet permutácií (pre každú permutáciu jeden záznam) V tom prípade bude mať riadok 12 zložitosť $O(pRJ_D \log(pRJ_D))$. Celková zložitosť algoritmu je $O(pJ_D + pJ_D \log(pJ_D) + p^2RJ_D + pRJ_D \log(pRJ_D))$. Vidíme, že zložitosť vyhľadávacej časti algoritmu vôbec nezávisí na počte dokumentov v korpuse, teda je konštantná vzhľadom k počtu dokumentov v korpuse.

Toto je možné demonštrovať na jednoduchom príklade. Majme kolekciu N dokumentov, ktoré nebudú mať spoločný ani jediný term. Z toho plynie, že nebudú mať ani jeden spoločný šindeľ, teda každý invertovaný zoznam bude mať dĺžku práve jedna. Ďalej, majme vstupný dokument D, ktorý je rozdelený na jednotky d_j, $|d_j| = j$, a $|\Pi| = p$. To znamená, že dokument nám vygeneroval pj šindľov, pre jednoduchosť budeme predpokladať, že sú všetky rôzne. To znamená, že je treba prehľadať maximálne pj invertovaných zoznamov. Keďže prístup k invertovanému zoznamu má konštantnú zložitosť, a každý invertovaný zoznam má dĺžku práve jedna, bude zložitosť nezávislá na N.

Teraz, ak ku kolekcii pridáme ďalších N dokumentov (tzn. bude mať veľkosť $2N$), ktoré nebudú mať žiadny spoločný šindeľ medzi sebou, ani s pôvodnými dokumentmi, nič sa nezmení, a opäť bude potrebné prehľadať maximálne pj invertovaných zoznamov dĺžky jedna, takže zložitosť je aj v tomto prípade nezávislá na N.

Samozrejme, takto jednoduchý príklad v reálnom nasadení sotva niekedy nastane, a budeme musieť prechádzať celé invertované zoznamy dĺžky väčšej ako jedna, ale keďže predpokladáme, že $R \ll N$, náš odhad zostáva v platnosti.

5.5. Parametre metódy

Najdôležitejšími parametrami metódy z hľadiska výkonu sú použité lingvistické jednotky, dĺžka šindľu (resp. n-gramu) a počet permutácií.

Ako lingvistické jednotky budeme skúmať vety, alebo časti dokumentu rozdelené po k slovách, kde $k > 0$, každá časť okrem poslednej obsahuje minimálne k slov a každá časť končí celou vetou, t.j. pokiaľ už má daná časť k slov, počkáme na koniec vety a časť ukončíme. Týmto spôsobom modelujeme odseky, a to z dôvodu, že programátorsky deliť na odseky wordovské dokumenty, alebo dokumenty vo formáte pdf bohužiaľ nie je dosť dobre možné, a to najmä kvôli zložitému rozpoznávaniu konca odseku. Dokumenty ako lingvistické jednotky nevhodné k detekcií plagiátov vylúčila už práca [30].

Štúdia [61] sa venuje tejto téme, a podľa jej výsledkov je najlepšie použiť šindle dĺžky dve alebo tri slová, s tým, že šindle dĺžky tri slová sú výhodnejšie v zmysle množstva falošných pozitív a negatív, práca [30] odporúča použiť šindle práve s touto dĺžkou. Ako najvhodnejšie sa teda javí, aby jeden šindeľ tvorili tri slová.

Čo sa týka počtu použitých permutácií, autorka štúdie [30] uvádza, že najvhodnejší počet permutácií pri použití viet ako lingvistických jednotiek je 5 alebo 10. Otázkou je, či v takomto prípade nebude časová náročnosť algoritmu príliš veľká, a či by nebolo vhodné uvažovať v prípade viet menšiu hodnotu. Pre odseky dokonca práca [30] vhodný odhad počtu permutácií neponúka, preto sa v ďalšom texte budeme snažiť o nájdenie najvhodnejšieho počtu permutácií pre odseky i vety tak, aby spĺňali požiadavky uvedené v kapitole 2.2.

Ďalšími dôležitými parametrami sú hodnota Jaccardovho koeficientu, pri ktorej budeme dve lingvistické jednotky považovať za podobné, a počet lingvistických jednotiek, pre ktorý budeme dva dokumenty už považovať za plagiát jeden druhého. Ako je však uvedené v kapitole 2.2, tieto dva parametre sa nebudeme snažiť odhadnúť, ale skôr užívateľovi ponúkneme zoznam výsledkov zoradený podľa týchto dvoch parametrov.

5.6. Pridávanie a mazanie dokumentov z indexu

Použitie invertovaného indexu v našom algoritme má ešte jednu výhodu, a tou je, že pridávanie a mazanie dokumentov z indexu je už veľmi dobre preskúmané, a teda sú známe vhodné postupy, ako túto funkcionalitu vhodne implementovať. Egothor implementuje metódu známu ako dynamické indexovanie (dynamic indexing) [40], ktorá je založená na myšlienke pomocného indexu, respektíve viacerých pomocných indexov. Nové dokumenty sú zaraďované do tohto pomocného indexu a výsledková listina je potom spojením výsledku dotazu nad hlavným indexom a výsledkom dotazu nad pomocným indexom. Zmazané dokumenty sú označované pomocou bitového vektoru, tak, aby sa dali pred vrátením výsledku vyfiltrovať. Aktualizácia dokumentu je potom realizovaná ako zmazanie a pridanie dokumentu do indexu.

Nevýhodou takéhoto prístupu je značná časová zložitosť, ktorá je $O(T^2/n)$, kde T je celkový počet záznamov v indexe a n je počet záznamov v pomocnom indexe. Preto [40] uvádza postup nazývaný logaritmické zlučovanie (logarithmic merging), ktorý je založený na myšlienke že máme $\log_2(T/n)$ indexov veľkosti $2^0.n, 2^1.n, 2^2.n, \ldots$, kde má celá konštrukcia indexu zložitosť $O(T\log(T/n))$. Samozrejme, v takomto prípade potrebujeme zlučovať výsledky dotazu nad každým indexom.

Ďalšou nevýhodou je fakt, že v prípade veľkého počtu zmazaných záznamov takáto štruktúra prestáva fungovať efektívne, a čo je ešte väčšia nevýhoda, takúto situáciu je pomerne zložité detegovať [62]. Preto [62] prichádza s riešením, ktoré vždy zaručuje minimálny počet „živých“, teda nezmazaných, dokumentov v jednom indexe (bareli) v závislosti na celkovom počte dokumentov v tomto bareli. Pokiaľ počet nezmazaných dokumentov klesne pod toto minimum, spustí sa reorganizácia indexu, a po tejto operácií bude predchádzajúca podmienka opäť v platnosti. Toto riešenie je taktiež implementované vo vyhľadávači Egothor.

Vzhľadom k tomu, že neočakávame, že v prípade systému postaveného na detekciu plagiátov bude dochádzať k veľkému počtu operácií mazania, používame riešenie využívajúce jednoduchší spôsob logaritmického zlučovania. Použitie

riešenia s podmienkou minimálneho počtu nezmazaných dokumentov sa ale dá ľahko nastaviť pri vytváraní indexu zmenou inicializačného parametra.

6. Popis implementácie

V tejto kapitole sa zameriame na základný popis implementácie zvolenej metódy v prostredí internetového vyhľadávača Egothor. Ako bolo spomenuté v kapitole 4, Egothor je napísaný v programovacom jazyku Java, a teda aj naša implementácia je vytvorená práve v tomto jazyku. Vzhľadom k tomu, že stupeň integrácie s vyhľadávačom Egothor je pomerne vysoký, nepodarilo sa nám oddeliť implementáciu detekcie plagiátov od vyhľadávača tak, aby sa mohla nachádzať v samostatných Javovských balíčkoch. Budeme sa teda snažiť popisovať implementáciu po akýchsi logických celkoch. Podrobný popis jednotlivých tried a metód sa potom nachádza v programátorskej dokumentácii k tejto práci.

Vzhľadom k tomu, že práca [30] bola takisto implementovaná v prostredí tohto internetového vyhľadávača, využívame v našej implementácii niektoré jej funkcionality. Konkrétne sa jedná o veci, ktoré nesúvisia priamo s použitým algoritmom, ale skôr s prípravou dát pre tento algoritmus. Sú to:

- Priraďovanie číselných identifikátorov jednotlivým termom, respektíve postupnosti termov za použitia hashovacej funkcie. V zásade sa jedná o vytváranie konkrétnych šindľov.
- Generovanie náhodných permutácií využívaných v algoritme
- Filtre (viď kapitola 4.2) pre vytváranie prekrývajúcich sa postupností k slov

Bližší popis týchto funkcionalít je popísaný práve v [30].

6.1. Vytváranie invertovaného indexu

Kvôli vytvoreniu invertovaného indexu schopného detekcie plagiátov bolo potrebné rozšíriť štruktúru, ktorá je prvkom invertovaného zoznamu patriaceho k jednému termu. Klasickému invertovanému zoznamu by teoreticky stačilo držať si vo svojich prvkoch identifikátor dokumentu. Ale vzhľadom k tomu, že potrebujeme rozlišovať menšie lingvistické jednotky ako celý dokument, pridali sme k identifikátoru dokumentu aj identifikátor odseku (resp. časti, viď kapitola 5.5) v dokumente a identifikátor vety v odseku. Pokiaľ ako lingvistickú jednotku používame odsek, identifikátor vety je samozrejme neprítomný. Ďalej, ako bolo spomenuté v kapitole

5.4, potrebujeme vyriešiť situáciu s porovnávaním jednotlivých reprezentantov iba v rámci jednej permutácie, čo tiež zvládneme pomocou pridania identifikátoru permutácie do invertovaného zoznamu. Implementácia týchto invertovaných zoznamov je realizovaná za pomoci tried IDuplListMetadataRead a IDuplListMetadataWrite nachádzajúcich sa v balíčku org.egothor.store. Tieto triedy sú oproti pôvodným triedam IListMetadataRead a IListMetadataWrite obohatené o možnosť čítať, resp. zapisovať do invertovaného zoznamu aj identifikátory permutácií, odsekov a viet.

Majme dva dokumenty s obsahom:

1) *Ema melie mak. Mama melie mak.*
2) *Tato melie mak.*
   Ema melie mak.

Povedzme, že tieto dokumenty budú mať identifikačné číslo 1 a 2. Klasická implementácia invertovaného zoznamu by z tohto dokumentu vytvorila nasledujúcu štruktúru (vynechávame váhy termov):

Term	Invertovaný zoznam
<WORD>ema	1,2
<WORD>mak	1,2
<WORD>mama	1
<WORD>melie	1,2
<WORD>tato	2

Tabuľka 6.1 – Invertované zoznamy pre slová

Ak ale chceme dokument prispôsobiť na detekciu duplicít, do slovníku termov sa pridajú aj šindle pre jednotlivé lingvistické jednotky, a k nim IListMetadataWrite zapíše do invertovaného zoznamu okrem identifikátoru dokumentu aj identifikátor permutácie, identifikátor odstavca a identifikátor vety (v tomto poradí). Samozrejme, zapisujú sa iba relevantné informácie, takže v prípade že je zvolenou jednotkou odsek, identifikátor vety sa nezapisuje.

Zvoľme ako lingvistickú jednotku vetu, a počet použitých permutácií rovný jednej, potom budú invertované zoznamy pre vytvorené šindle vyzerať nasledovne (odseky i vety sa číslujú od 0):

Term	Invertovaný zoznam
<DUP>112598745444	(1,1,0,0), (2,1,1,0)
<DUP>854712345546	(1,1,0,1)
<DUP>991266354113	(2,1,0,0)

Tabuľka 6.2 – Invertované zoznamy pre šindle

6.2. Tvorba dotazu pre vyhľadávanie

Pri tvorbe dotazu z dokumentu si musíme uvedomiť, že množstvo šindľov vygenerovaných pre jeden dokument môže byť značne veľké (v ráde tisícov) a je potrebné prehľadať index pre všetky šindle. Ako je popísané v kapitole 4.4, Egothor pre tvorbu dotazu používa stromovú štruktúru a v rámci nej objekty typu Query, nás bude zaujímať najmä objekt, ktorý tvorí tzv. uzol skupiny. Musíme si totiž uvedomiť, ako prebieha samotné vyhľadávanie – rozhranie Jazdec/Bežec totiž pri vyhodnocovaní dotazu nečíta vždy práve jeden invertovaný zoznam, ale kvôli efektivite číta vždy invertované zoznamy všetkých termov dotazu. Ak však dotaz tvorí príliš veľa termov, stane sa takýto postup naopak značne neefektívnym, pretože sa z disku číta naraz z príliš mnoho miest, čím dochádza k zbytočne veľa diskovým operáciám. Preto sme sa rozhodli termy zoskupiť do uzlov skupín po pevne stanovenom počte termov, a všetky tieto skupiny združiť do koreňového uzla skupiny. Rozhraniu Jazdec/Bežec je potom predávaná vždy práve jedna takáto podskupina, čo zaručuje, že toto rozhranie bude naraz spracovávať najviac stanovený počet šindľov. Počet invertovaných zoznamov, ktoré je z časového hľadiska ideálne spracovávať naraz, sa pokúsime určiť v ďalšom texte. Rozhranie Jazdec/Bežec je realizované pomocou triedy DuplicityRider nachádzajúcej sa v balíčku org.egothor.query.rider a tried DuplicityTermRunner a DuplicityVectorRunner nachádzajúcich sa v balíčku org.egothor.query.runner.

6.3. Spracovanie dotazu

Samotné spracovanie dotazu na duplicitu dokumentu sa od štandardného spracovania líši v dvoch veciach. Prvou je vyhľadávanie ďalšieho kandidáta na zásah Jazdcom, kde sa Jazdec musí prispôsobiť skutočnosti, že invertovaný index neobsahuje iba identifikátory dokumentu, ale aj jeho častí a identifikátory permutácií. Druhou je samotné ukladanie zásahov do výsledkovej listiny. Pokiaľ

máme zásah v klasickom modeli vyhľadávania, nie je potrebné s ním už nič ďalej vykonávať a môžeme ho priamo zaradiť do výsledkovej listiny. V algoritme 5.6 je síce popísaný podobný postup, kde sa zásah zaraďuje do výsledkovej listiny priamo, a tá sa triedi až nakoniec, v skutočnosti je však výhodnejšie a prehľadnejšie udržiavať si už zotriedenú postupnosť, do ktorej nájdené zásahy zatrieďujeme. Výsledková listina je realizovaná za pomoci triedy DuplicityResultList, výsledková listina používaná v užívateľskom rozhraní potom pomocou triedy DuplicityResultListBean. Obe triedy sa nachádzajú v balíčku org.egothor.query.result.

6.4. Podpora rôznych formátov

Egothor mal v sebe pôvodne zabudovanú podporu iba na parsovanie dokumentov vo formáte html a podporu na parsovanie streamov (prúdov znakov), no a pretože jedným z cieľov tejto práce bolo umožniť užívateľovi pohodlnú prácu s rôznymi formátmi, bol za týmto účelom vytvorený balíček tried, ktoré takúto prácu umožňujú. Toto sa deje pomocou konverzie podporovaných formátov na html dokumenty, s ktorými už Egothor vie pracovať. Podporované formáty sú pdf, doc (t.j. formát Microsoft Office 97-2003) a rtf (a samozrejme html). Podpora rôznych formátov sa nachádza v balíčku org.egothor.parser.transformer.

6.5. Implementácia slovníku termov

Vo vyhľadávači Egothor je slovník termov implementovaný iba pomocou index-sekvenčného algoritmu. Ten funguje tak, že slovník je abecedne zotriedený, a rozdelený na časti, nazývané buckety. Každý bucket má na svojom začiatku uvedený posledný term, ktorý sa v ňom nachádza, takže v prípade že je vyhľadávaný term abecedne až za posledným termom v buckete, nie je nutné tento bucket prechádzať. Ďalej, implementácia si vždy pamätá posledný bucket ktorý prechádzala, takže v prípade že ďalší dotaz je na term, ktorý sa abecedne nachádza až za predchádzajúcim termom, nemusí prechádzať celý slovník od začiatku, ale môže jednoducho pokračovať.

Keďže v našom algoritme ale predpokladáme prístup do slovníku termov s konštantnou zložitosťou, bolo nutné pridať implementáciu pomocou hashovacieho algoritmu. K tomuto účelu bola zvolená štandardná implementácia Faginovho

rozšíriteľného hashovania. Nachádza sa spoločne s pôvodnou implementáciou v balíčku org.egothor.store a je realizovaná pomocou tried TermsHashImpl a TermsHashWriter.

6.6. Užívateľské rozhranie

Užívateľské rozhranie vyhľadávača duplicitných dokumentov je realizované ako webová aplikácia vytvorená za pomoci technológie Java Server Faces. V rámci tejto technológie využívame tiež služby knižnice Tomahawk ([71]) od spoločnosti Apache. Základom aplikácie je IndexBean, čiže trieda spravujúca celý index, ktorú využívajú všetky ostatné triedy (beany).

Úvodná obrazovka je podobná obrazovke štandardných internetových vyhľadávačov (napríklad Google), užívateľ má možnosť do vyhľadávača nahrať súbor z pevného disku, alebo zadať úryvok textu priamo do okna na to určeného. Takisto na obrazovke vidí preňho dôležité informácie o aktuálnom nastavení vyhľadávača, a to používanú lingvistickú jednotku a použitý počet permutácií (čo pre užívateľa znamená maximálne skóre ktoré môže daná jednotka dosiahnuť).

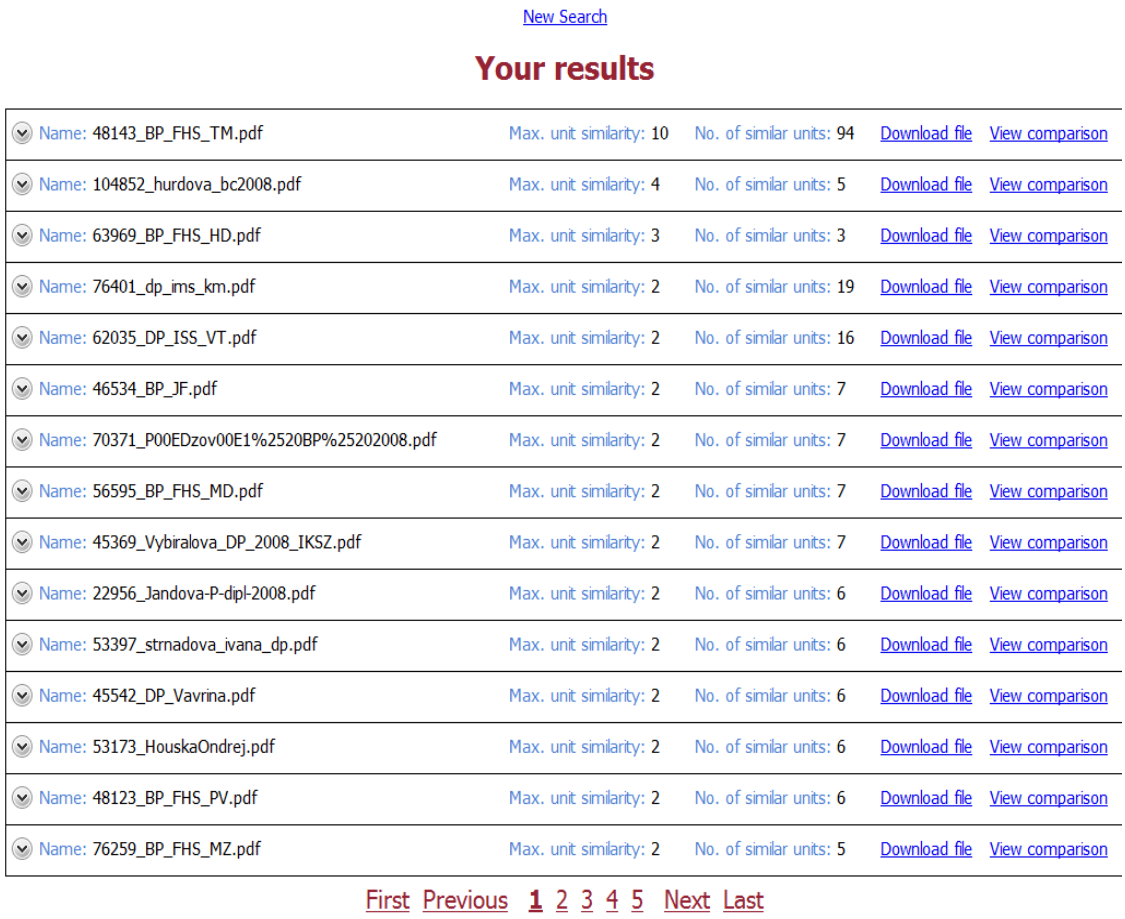
Na tejto obrazovke si môže nastaviť taktiež minimálnu hodnotu podobnosti, od ktorej bude lingvistická jednotka zaradená do výsledkovej listiny. Poslednou akciou je potom autentifikácia užívateľa, ktorá je dôležitá v prípade že požaduje prístup k celým dokumentom. O spracovanie (naparsovanie) dokumentu sa potom postará trieda DataInputBean, ktorá taktiež vykoná dotaz nad indexom a výsledok dotazu posunie triede DataModelBean, ktorá sa stará o zobrazovanie výsledkov užívateľovi. Vstupnú obrazovku môžeme vidieť na obrázku 6.1.

Obr. 6.1 – Úvodná obrazovka aplikácie Egothor duplicity finder

Trieda DataModelBean zobrazuje výsledky po stránkach, s tým, že metadáta každého dokumentu (názov dokumentu, a pod.) sa načítajú iba pre tie dokumenty, ktoré sú práve zobrazené, čo tiež prispieva k zrýchleniu dotazu z pohľadu užívateľa. Na každej stránke sa môže nachádzať maximálne S dokumentov, takže v prípade že algoritmus vráti $N \gg S$ podobných dokumentov, nemusia sa metadáta načítať pre všetkých S dokumentov, ale načítajú sa iba pre top N dokumentov (pri prvom zobrazení) a potom vždy pre ďalšie aktuálne dokumenty, podľa toho ako užívateľ prechádza medzi stránkami.

Výsledky sú organizované v prehľadnej tabuľke, kde riadky tvoria jednotlivé podobné dokumenty D_p. Každý riadok obsahuje názov dokumentu (tak ako bol

vložený do systému), počet podobných jednotiek a maximálnu hodnotu podobnosti δ, ktorá bola nájdená medzi jednotkami dotazovaného dokumentu a dokumentu D_p. Ďalej každý riadok obsahuje tri akcie – stiahnutie dokumentu, podrobný náhľad na podobné jednotky dokumentu, a fulltextové porovnanie dokumentu D_p s dotazovaným dokumentom. Pohľad na tabuľku výsledkov môžeme vidieť na obrázku 6.2.



Obr. 6.2 – Výsledková listina aplikácie

Stiahnutie dokumentu je chránené heslom kvôli skutočnosti, že aj keď by záverečné práce študentov mali byť voľne dostupné, univerzita chce mať často prehľad o tom kto mal k jednotlivým prácam prístup a navyše systém nemusí byť nutne nasadený iba v akademickom prostredí.

Podrobný náhľad jednotiek dokumentu sa objaví po rozkliknutí ikony s obrázkom šípky, a nachádzajú sa v ňom všetky podobné jednotky spoločne s s hodnotou podobnosti. Pre lepšiu prehľadnosť sú jednotky organizované do skupín, každú skupinu tvoria jednotky nasledujúce ihneď po sebe s rovnakou hodnotou

podobnosti. . Zvoľme odsek ako lingvistickú jednotku, a nech má nájdený dokument desať odsekov, z čoho je päť podobných (Tabuľka 6.3).

Jednotky	Hodnota podobnosti
Jednotka 1	10
Jednotka 2	10
Jednotka 4	8
Jednotka 5	9
Jednotka 6	9

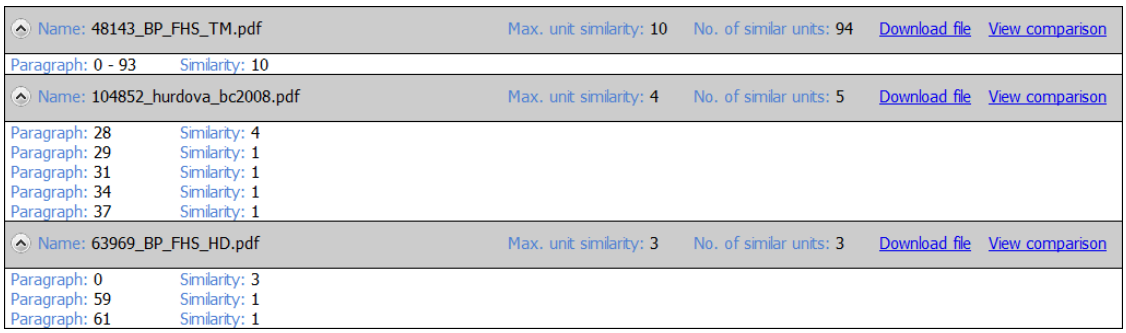
Tabuľka 6.3 – Tabuľka jednotiek pred úpravou

Výsledná tabuľka bude potom vyzerať nasledovne (Tabuľka 6.4).

Jednotky	Hodnota podobnosti
1-2	10
4	8
5-6	9

Tabuľka 6.4 – Tabuľka jednotiek po úprave

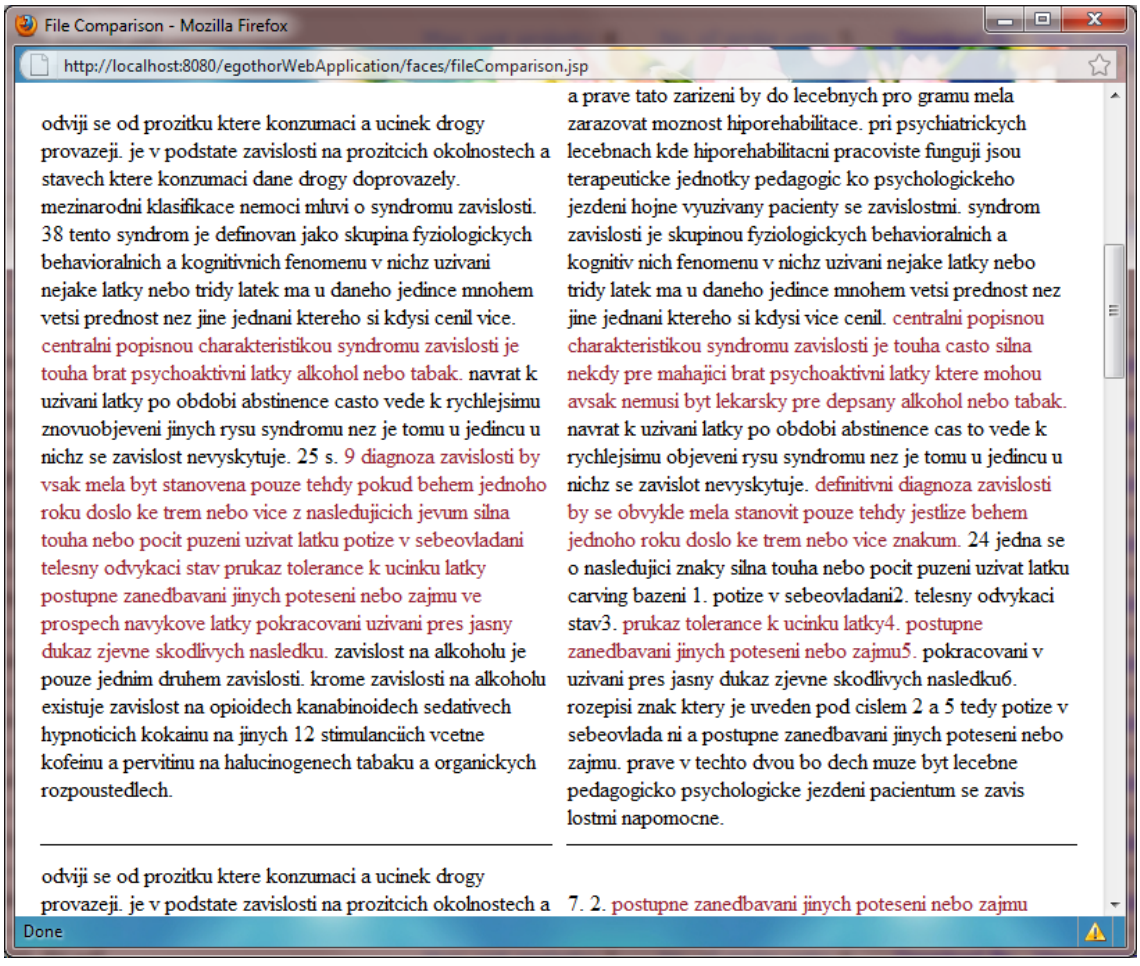
Podrobný náhľad na jednotky dokumentu môžeme vidieť na obrázku 6.3



Obr. 6.3 – Náhľad na jednotky dokumentu

Ak má výsledný dokument so zadaným dokumentom viacero podobných jednotiek, v tomto náhľade sa zobrazí iba jednotka, ktorá dosahuje najväčšiu podobnosť. Je to preto, že ak je jedna jednotka z výsledného dokumentu podobná viacerým jednotkám so zadaného dokumentu, je vysoko pravdepodobné, že ostatné podobnosti, okrem tej z najvyššou hodnotou, budú pre užívateľa nerelevantné, napr. založené na podobnosti nevýznamových slov, ako „...a je to preto, že...“, „...stalo sa to preto, že...“ a pod. Ostatne, nie je pravdepodobné, že napríklad v prípade plagiátorstva by plagiátor kopíroval do svojej práce dvakrát rovnakú časť textu. Všetky podobné jednotky sa ale samozrejme zobrazia v okne textového porovnania.

Poslednou akciou je textové porovnanie (Direct Source Comparison – DSC) dotazovaného dokumentu s dokumentom D_p. O toto porovnanie sa stará trieda FileComparisonBean, ktorá otvorí nové okno prehliadača, a v tomto okne organizuje priamo pohľad na text podobných dokumentov. Ten je do istej miery nezávislý na lingvistických jednotkách, a to v tom zmysle, že sa snažíme označovať podobnosť pre konkrétne nájdené šindle. To znamená, že pre každý podobný šindeľ je zobrazený celý odsek obsahujúci daný šindeľ, a v ňom sú potom zvýraznené konkrétne vety, do ktorých šindeľ zasahuje. To znamená, že ak by bol lingvistickou jednotkou nasledujúci odsek: „Toto je odsek. Je veľmi krátky. Práve skončil.“, a šindeľ by bol tvorený trojicou slov „odsek je veľmi“, budú zvýraznené vety „Toto je odsek. Je veľmi krátky“. Samozrejmosťou je potom organizovanie jednotiek tak, aby podobné jednotky dotazovaného dokumentu a dokumentu D_p boli hneď vedľa seba pre lepšiu prehľadnosť, a boli zoradené podľa dosiahnutej miery podobnosti. Pre lepšiu predstavu fulltextového porovnávania slúži obrázok 6.4.



Obr. 6.4 – Náhľad textového porovnania

6.7. Nastavenie minimálnej hodnoty Jaccardovho koeficientu

Ako sme naznačili už v kapitole 2.2, nebudeme sa snažiť obmedzovať hodnotu δ pre ktorú platí, že ak máme dve jednotky p a q, a definovanú podobnostnú funkciu $sim(p, q)$, tak v prípade, že p je plagiátom q (alebo vice versa), tak $sim(p, q) > \delta$. Niekedy sa ale môže stať, že užívateľ dostane príliš veľké množstvo výsledkov, ktoré budú mať so vstupom príliš malú podobnosť na to, aby to preňho bolo relevantné, prípadne bude chcieť zobraziť iba veľmi podobné jednotky. V tom prípade by bolo vhodné, aby si bol schopný sám nastaviť minimálnu hodnotu, od ktorej by sa mali jednotky zaraďovať do výsledkovej listiny.

V našom prípade je takouto podobnostnou funkciou aproximovaná hodnota Jaccardovho koeficientu, ktorá sa spočíta (viď. kapitola 5) ako počet šindľov zhodných pre obe jednotky delený počtom použitých permutácií. V tom prípade teda nemusíme užívateľovi v nastavení ponúkať preňho ťažko pochopiteľné koeficienty, ale namiesto toho mu ponúkneme nastavenie minimálneho zhodného počtu šindľov, čo v zásade splní rovnakú funkciu a bude to preňho oveľa zrozumiteľnejšie.

7. Experimentálne overenie vlastností systému

V tejto kapitole otestujeme, ako sa systém správa na reálnych dátach z akademického prostredia. Prioritou samozrejme bude nastaviť parametre systému tak, aby pri spracovaní dokumentu dosahoval maximálne časy uvedené v kapitole 2. Na druhej strane je taktiež dôležité, aby systém nestratil príliš mnoho, čo do pravdepodobnosti označenia plagiátu, na optimálne nastavenie parametrov uvedených v práci [30].

Nebudeme teda už skúmať vhodnosť použitého algoritmu k detekcií plagiátov, pretože to už v dostatočnej miere popisuje práca [30]. Budeme sa snažiť postupovať skôr tak, že vezmeme optimálne nastavenie parametrov, ktoré uvádza práca [30], t.j. lingvistická jednotka bude veta a počet permutácií bude 10, a toto nastavenie budeme považovať za referenčné. Cieľom teda bude nájsť také nastavenie parametrov, aby čas celého spracovania dotazu nepresiahol 10 sekúnd, a pomer počtu nájdených jednotiek, resp. pomer skóre ktoré dosiahne top N jednotiek oproti referenčnému nastaveniu bol čo najbližší 1.

Časy ktoré budeme skúmať rozdelíme na tri časti, a síce na čas potrebný k spracovaniu dokumentu, teda čas potrebný k vytvoreniu kolekcie šindľov, čas samotného behu dotazu a čas potrebný k stavbe výstupu pre užívateľa. Pri čase stavby výstupu je kritické získavanie metadát dokumentu (t.j. jeho názov, umiestnenie atď.) z indexu, preto budeme merať čas potrebný na získanie metadát prvých dvadsiatich dokumentov – ako bolo popísané v predchádzajúcej kapitole, metadáta dokumentu sa načítajú v momente, keď sú výsledky pre dané dokumenty po prvýkrát zobrazené, a na jednej stránke je naraz zobrazených dvadsať dokumentov.

Naše kolekcia dát obsahuje viac ako 7500 kvalifikačných prác študentov Karlovej Univerzity, jedná sa o kvalifikačné práce zo všetkých úrovní (bakalárske, diplomové, rigorózne, atď.), v dobe získavania týchto dát toto číslo tvorilo približne polovicu všetkých prác v elektronickom systéme na zadávanie prác Karlovej Univerzity. Dá sa teda povedať, že naša kolekcia spĺňa požiadavku na „dáta čo

najviac sa približujúce reálnym dátam z akademického prostredia“ tak do obsahu dokumentov v kolekcií ako aj veľkosti kolekcie. Čo sa týka digitálnych kolekcií ostatných univerzít, tak sú čo do veľkosti veľmi podobné digitálnej kolekcií Karlovej Univerzity, napr. digitálna kolekcia University of Newcastle [63] obsahuje cca 9000 prác, digitálna kolekcia Colorado State University [64] obsahuje cca. 20 000 prác, digitálna kolekcia Texas A&M University [65] obsahuje cca. 1000 záverečných prác, atď.

Samozrejme, problémom merania časovej náročnosti je aj závislosť na hardwarovej konfigurácií stroja, na ktorom sú experimenty vykonávané. Preto uvádzame našu konfiguráciu, ktorá je:

- Procesor: Intel i3 Dual-Core, 3,06 GHz, 4MB L3 Cache

- Pamäť: 4GB DDR3 1333 MHz

- Externá pamäť: Western Digital HDD, 64 MB Cache, 5400 – 7200 RPM

Ďalším problémom je skutočnosť, že nie sme schopní úplne kontrolovať všetky procesy operačného systému, ako napríklad aké dáta si drží v operačnej pamäti, akým spôsobom rozdeľuje úlohy medzi jadrá procesoru, ako optimalizuje diskové operácie a podobne. Časť týchto problémov by sa možno dala odstrániť reštartovaním systému po behu každej jednotlivej úlohy (napríklad dáta držané v operačnej pamäti, prípadne v cache procesora), ale jednak by sme ani zďaleka neodstránili všetky problémy, a jednak by tento spôsob nezodpovedal reálnemu nasadeniu systému, kde sa v operačnej pamäti a v cache procesora vždy nachádzajú nejaké dáta.

7.1. Počet naraz spracovávaných invertovaných zoznamov

V priebehu spracovania algoritmu sa z externej pamäte číta viacero invertovaných zoznamov z indexu, ktorý je uložený na externej pamäti, naraz. Takýto index môže mať značnú veľkosť, pre 7500 dokumentov je to cca. 1 – 3 GB v závislosti na nastavení parametrov. Označme počet naraz spracovávaných invertovaných zoznamov N, a povedzme, že pri spracovaní dokumentu získame 1000 šindľov. Potom týchto 1000 šindľov rozdelíme na časti po N šindľoch, a každú takúto časť spracovávame naraz, teda čítame naraz invertované zoznamy pre N rôznych

šindľov, a teda N invertovaných zoznamov. Našim cieľom je zistiť, či, a do akej miery je takéto N podstatné vzhľadom k času behu dotazu, a v prípade že sa ukáže že podstatné je, chceme vybrať najvhodnejšie takéto N.

Už len z laického pohľadu by počet naraz čítaných invertovaných zoznamov mal byť dôležitý. Pri spracovávaní jedného, prípadne niekoľko málo zoznamov bude celý proces neefektívny, naopak, pri spracovávaní príliš veľa zoznamov naraz zasa môže dochádzať k veľkému počtu seek operácií, čo tiež povedie k zníženiu časovej efektivity.

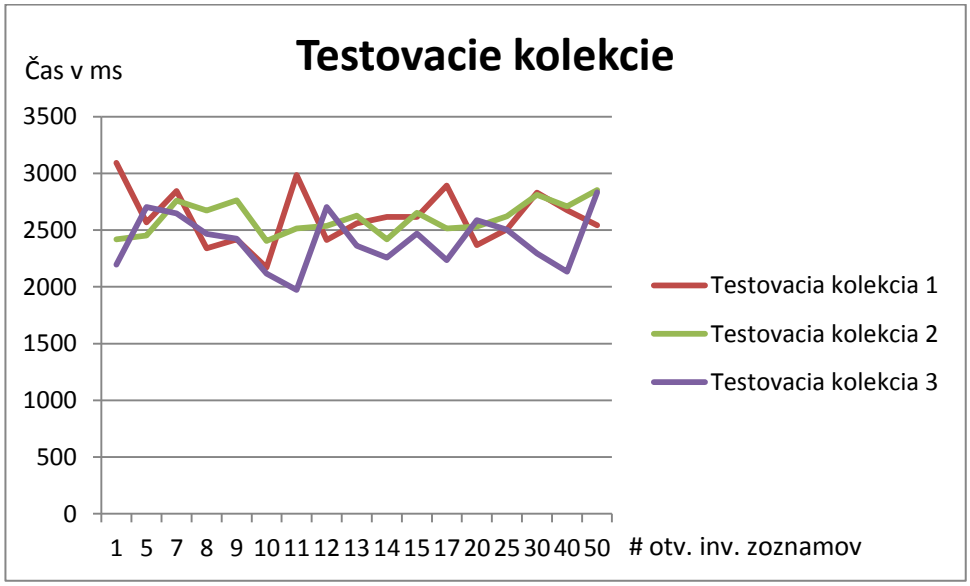
Experiment bol koncipovaný nasledovne. Náhodne sme s dokumentov obsiahnutých v indexe vybrali trikrát pätnásť dokumentov, a takýchto pätnásť dokumentov tvorilo jednu testovaciu kolekciu. Aby sme čo najviac eliminovali vplyv držania dát v operačnej pamäti, prípadne v cache procesora, používali sme tri identické indexy, a jednotlivé testovacie kolekcie sme spracovávali vždy na náhodne zvolenom indexe z tejto trojice. Pre každú testovaciu kolekciu sme spustili šesť behov pre každé N. V tomto experimente sme sledovali iba samotný čas behu dotazu, keďže počet naraz spracovávaných invertovaných zoznamov nemá vplyv na čas spracovania dokumentu, ani na čas stavby výstupu. Výsledky experimentu uvádza tabuľka 7.1, v každom stĺpci je uvedený priemerný čas behu dotazu pre jednu kolekciu, v záverečnom stĺpci je potom uvedený výsledný priemerný čas behu dotazu. Časy sú uvádzané v milisekundách. Graf 7.1 uvádza priemery testovacích kolekcií, graf 7.2 slúži pre znázornenie celkového priemeru. Experiment bol vykonávaný s nastavením parametrov:

- Lingvistická jednotka: Úsek dlhý 100 slov
- Počet permutácií: 10

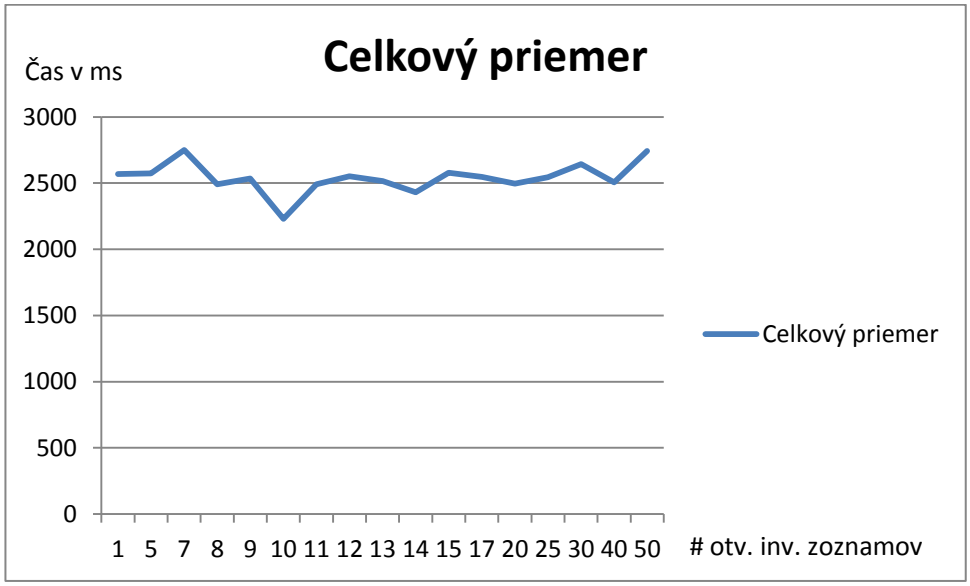
Počet naraz spracovávaných invertovaných zoznamov	Testovacia kolekcia 1	Testovacia kolekcia 2	Testovacia kolekcia 3	Celkový priemer
1	3093.666667	2418.833333	2194.333333	2568.944444
3	2567	2453.5	2702.666667	2574.388889
7	2845	2758.5	2647.333333	2750.277778
8	2338.333333	2671.166667	2467	2492.166667
9	2418.833333	2762	2424.333333	2535.055556
10	2170.833333	2404.166667	2116.666667	2230.555556
11	2985.333333	2513.666667	1971.833333	2490.277778

12	2413.833333	2537.5	2704.166667	2551.833333
13	2559.166667	2626.5	2362.166667	2515.944444
14	2615.833333	2419.166667	2257.333333	2430.777778
15	2616	2653.666667	2468.833333	2579.5
17	2893.166667	2513.333333	2233.833333	2546.777778
20	2368.166667	2531	2587.166667	2495.444444
25	2506.333333	2621	2504	2543.777778
30	2829.166667	2809.833333	2294	2644.333333
40	2676.333333	2709.5	2134.5	2506.777778
50	2541.833333	2853.333333	2832.166667	2742.444444

Tabuľka 7.1 – Počet naraz spracovávaných zoznamov vzhľadom k času dotazu



Graf 7.1 – Počet zoznamov vzhľadom k času dotazu pre jednotlivé kolekcie



Graf 7.2 – Počet zoznamov vzhľadom k času dotazu- priemer

Na výslednom grafe vidíme, že navzdory očakávaniu sa časy behu dotazu vzhľadom na počet naraz spracovávaných invertovaných zoznamov nijak významne nelíšia, v podstate to vyzerá že efektivita začína klesať až pri spracovaní viac ako 40 invertovaných zoznamov, medzi hodnotou 10 a 40 sú časy viac – menej konštantné. Môžeme si však všimnúť minimálnej hodnoty, ktorá je už celkom výrazná, a táto hodnota je dosiahnutá pri spracovávaní desiatich invertovaných zoznamov naraz. V ďalších experimentoch je teda používaná hodnota $N = 10$.

7.2. Implementácia slovníku termov

Parametre vymenované v nadpise tejto podkapitoly sú na sebe do značnej miery závislé. Počet permutácií, a teda počet šindľov, pre jednu lingvistickú jednotku je závislý na použitej jednotke – je samozrejmé, že pre vetu je zbytočný príliš vysoký počet permutácií, na druhej strane, pre odsek či celý dokument je nutné použiť vyšší počet permutácií k dosiahnutiu požadovanej presnosti. Na to následne nadväzuje implementácia slovníku termov (pre bližší popis viď kapitola 6.5), kde veľkosť bucketu v hashovanej i index-sekvenčnej implementácií je závislá na celkovom počte termov – čím väčší počet termov, tým by mala byť väčšia aj veľkosť jedného bucketu. Preto nie je možné skúmať hodnoty týchto parametrov nezávisle, ale musíme skúmať aj ich vzájomný vplyv.

V tejto podkapitole sa budeme zaoberať iba účinkom vyššie uvedených parametrov na časovú náročnosť algoritmu, ich vplyvom na funkčnosť algoritmu (tzn. počet nájdených duplicitných jednotiek, počet nájdených duplicitných dokumentov a pod.) sa budeme zaoberať neskôr.

Skúsme najskôr experimentálne zistiť najvhodnejšiu implementáciu slovníka termov pre špecifické hodnoty lingvistickej jednotky a počtu permutácií. Ako sme uviedli v predchádzajúcej kapitole, nebudeme uvažovať dokument ako lingvistickú jednotku. Taktiež nebudeme uvažovať klasický odsek, ale skôr delenie dokumentu na S slov, kde S bude nadobúdať hodnoty $S = \{100,200,300,400,500\}$. Pre odseky budeme uvažovať počty permutácií 10, 20, 30 a 40. Pokiaľ bude lingvistická jednotka veta, budeme uvažovať počty permutácií 5, a 3. Tak ako v predchádzajúcom experimente, opäť máme pripravené tri testovacie kolekcie dokumentov, na ktorých budeme experiment vykonávať, tzn. pre každú konfiguráciu

určenú štvoricou (implementácia slovníku termov, počet slov v lingvistickej jednotke, počet permutácií, veľkosť bucketu v slovníku termov) spustíme výpočet nad každou testovacou kolekciou. Pre každú testovaciu kolekciu si následne budeme všímať priemerný čas trvania jedného dotazu, a z týchto priemerných časov nakoniec dostaneme výsledný priemerný čas pre danú veľkosť bucketu.

Ostáva nám iba zvoliť si veľkosti bucketov, pre ktoré budeme nasledujúci pokus realizovať. Veľkosť sa udáva v počte termov, ktoré jeden bucket obsahuje. Označme túto veľkosť B. Je potrebné si uvedomiť, že minimálne pri implementácií slovníku termov pomocou rozšíriteľného hashovania (viď. kapitola 6.5) budú malé hodnoty B značne neefektívne, pretože celá štruktúra bude mať príliš mnoho vrstiev, takže sa zvýši počet prístupov na externú pamäť. Ukážme si teda výsledky pre veľkosti bucketov $B = 2^n, n = 10..12$. Výsledky experimentu uvádza tabuľka 7.2.

			Priemerný čas behu dotazu (ms) Index – sekvenčná implementácia			**Priemerný čas behu dotazu (ms) Hashovaná implementácia**		
Jedn.	**Počet slov**	**Počet perm.**	**Veľkosť 1024**	**Veľkosť 2048**	**Veľkosť 4096**	**Veľkosť 1024**	**Veľkosť 2048**	**Veľkosť 4096**
Odsek	100	10	9227	11420	18766	22446	9616	17544
Odsek	100	20	15638	23095	36368	48087	21248	35850
Odsek	100	30	23503	34250	55634	97353	64868	62412
Odsek	100	40	31700	46350	74189	185349	119586	101963
Odsek	200	10	4904	7197	11822	15355	7219	11458
Odsek	200	20	8560	12282	19553	22837	11585	18224
Odsek	200	30	12509	17903	28700	40518	18669	29040
Odsek	200	40	16682	23896	38876	50712	25862	37828
Odsek	300	10	3189	4466	6972	7974	4486	6471
Odsek	300	20	5973	8483	13569	16020	8298	12691
Odsek	300	30	8752	12393	20175	22871	12703	18480
Odsek	300	40	11459	16371	26667	37001	18287	26680
Odsek	400	10	2520	3470	5345	6548	3779	5121
Odsek	400	20	4652	6758	10603	14000	6840	10454
Odsek	400	30	6758	9689	15640	17830	9234	14386
Odsek	400	40	9152	12546	20387	23325	13120	18753
Odsek	500	10	2142	2892	4333	6475	3298	4567
Odsek	500	20	3995	5356	8440	10109	5334	7968
Odsek	500	30	5530	7787	12551	16263	7770	12200
Odsek	500	40	7146	10378	16578	18883	9850	15280
Veta	-	3	12648	18397	29023	41379	18093	29600

Veta	-	5	21258	30727	48614	76898	48944	52080

Tabuľka 7.2 – Implementácia slovníku termov vzhľadom k času behu dotazu

Z tabuľky je vidieť nasledujúce skutočnosti:

- Implementácia za pomoci rozšíriteľného hashovania je výrazne závislá na celkovom počte termov v slovníku. Počet termov v slovníku je priamo úmerný zvolenému počtu permutácií (čím viac permutácií tým, viac reprezentantov pre danú lingvistickú jednotku) a nepriamo úmerný počtu slov v lingvistickej jednotke (so zmenšujúcim sa počtom slov je dokument rozdelený na viacero jednotiek). Toto platí pre každú veľkosť bucketu.

  Dôvodom pre to je skutočnosť zmieňovaná vyššie, a síce, že so zvyšujúcim sa počtov termov v slovníku rastie počet použitých bucketov, čím rastie počet vrstiev rozšíriteľného hashovania, a teda aj počet prístupov na externú pamäť.
- Najlepšie výsledky dosahuje implementácia za pomoci rozšíriteľného hashovania pre veľkosť bucketu 2048. To je dané tým, že pre menšie veľkosti bucketu nastáva prípad rastu počtu vrstiev, pre väčšie je zasa neefektívne prechádzať veľké buckety.
- Priemerný čas behu dotazu pre index – sekvenčnú implementáciu klesá s veľkosťou bucketu, najlepšie výsledky teda dosahuje pre veľkosť bucketu 1024.
- Najlepšie výsledky dosahované index – sekvančnou implementáciou sú lepšie ako najlepšie výsledky dosahované implementáciou za pomoci rozšíriteľného hashovania, navyše, ako si ukážeme v nasledujúcej časti, trend klesania priemerného času behu dotazu s klesajúcou veľkosťou bucketu bude ešte pokračovať.

Z uvedených skutočností vyplýva, že napriek tomu, že teoreticky má prístup do slovníku termov za pomoci rozšíriteľného hashovania konštantnú časovú zložitosť, a index – sekvenčný prístup lineárnu časovú zložitosť, prakticky dosahuje index – sekvenčná implementácia lepšie výsledky.

7.3. Veľkosť bucketu pre index – sekvenčnú implementáciu

Skúsme teraz nájsť najvhodnejšiu veľkosť bucketu pre index – sekvenčnú implementáciu slovníka termov. Z minulej podkapitoly vyplynulo, že pokiaľ $B = 2^n, n = 10..12$, tak čas potrebný k behu dotazu klesá spoločne s klesajúcim n. Preto si aj ukážme výsledky pre $n = 6..9$. Experiment budeme koncipovať podobne ako v predchádzajúcom prípade, pridáme ešte ale delenie dokumentu na časti po 50 slov. Experiment sme opakovali trikrát, výsledné hodnoty sú priemerom jednotlivých behov. Výsledky experimentu sú uvedené v tabuľke 7.3.

			Priemerný čas trvania dotazu (ms)				
Jedn.	**Počet slov**	**Počet perm.**	**1024**	**512**	**256**	**128**	**64**
Odsek	50	10	14281	9928	8235	8614	8770
Odsek	50	20	29640	21167	18765	23090	19492
Odsek	50	30	45440	36333	33489	38133	39501
Odsek	50	40	63568	52582	45806	52528	52638
Odsek	100	10	9227	5717	4705	4899	4879
Odsek	100	20	15638	11428	9467	9900	10038
Odsek	100	30	23503	17512	13551	16143	17429
Odsek	100	40	31700	24249	21895	22977	21605
Odsek	200	10	4904	3566	2901	2915	2919
Odsek	200	20	8560	6087	4988	5317	5472
Odsek	200	30	12509	8808	7324	7469	7894
Odsek	200	40	16682	11787	9868	9840	9951
Odsek	300	10	3189	2386	2121	2172	2228
Odsek	300	20	5973	4304	3736	3704	3892
Odsek	300	30	8752	6177	5142	5322	5596
Odsek	300	40	11459	8079	6795	6835	6973
Odsek	400	10	2520	1950	1569	1673	1684
Odsek	400	20	4652	3457	2925	3002	3015
Odsek	400	30	6758	5243	4040	4082	4029
Odsek	400	40	9152	6557	5240	5296	5333
Odsek	500	10	2142	1649	1399	1519	1666
Odsek	500	20	3995	2884	2414	2904	3247
Odsek	500	30	5530	4128	3394	3772	3544
Odsek	500	40	7146	5245	4304	4337	4385
Veta	-	3	12648	9621	7693	8308	7567
Veta	-	5	21258	16900	13142	13560	14517

Tabuľka 7.3 – Najvhodnejšia veľkosť bucketu vzhľadom k času behu dotazu

Z tabuľky je vidieť, že namerané hodnoty klesajú spoločne s klesajúcou veľkosťou bucketu až do hodnoty $B = 2^8$, a to pre ľubovoľnú trojicu (lingvistická jednotka, počet slov v jednotke, počet použitých permutácií). Od tejto hodnoty majú namerané časy opäť miernu tendenciu stúpať, i keď nie zvlášť signifikantne. Z experimentu teda vyšlo, že najvýhodnejšie je používať veľkosť bucketu rovnú 2^8, teda 256.

7.4. Výber vhodnej jednotky, počtu slov a počtu permutácií

V predchádzajúcich kapitolách sme zistili, aká implementácia je najvhodnejšia z časového hľadiska. Teraz musíme zistiť, aká implementácia je najvhodnejšia z hľadiska počtu a presnosti nájdených dokumentov, samozrejme s tým, aby sme splnili cieľ odpovede systému do 10 sekúnd z kapitoly 2.

Experiment je opäť koncipovaný podobne ako experimenty z predchádzajúcich podkapitol, avšak pridali sme ďalšie parametre, ktoré sledujeme. Sú to jednak časové parametre, konkrétne:

- Čas prípravy reprezentácie dokumentu
- Čas získania metadát prvých 20 dokumentov, ktoré sa zobrazia užívateľovi.

Tieto časy nebudeme uvádzať priamo, ale iba súhrnne ako celkový čas dotazu, spoločne s priemerným časom trvania dotazu. Ďalšie parametre, ktoré sledujeme, súvisia s presnosťou vyhľadávania, konkrétne sa jedná o:

- Počet nájdených podobných lingvistických jednotiek
- Počet nájdených podobných dokumentov
- Celkové skóre top 20 zásahov
- Celkové skóre top 10 zásahov

Na tento účel sme k sledovaným lingvistickým jednotkám pridali vetu s počtom použitých permutácií 10, ktorá nám bude slúžiť ako referenčná jednotka. Je to z toho dôvodu, že práca [30] dospela k záveru, že práve táto kombinácia použitej lingvistickej jednotky a počtu permutácií je na detekciu plagiátov najvhodnejšia. Výsledky experimentu uvádza tabuľka 7.4.

				Počet podobností		Celkové skóre	
Jedn.	**Počet slov**	**Počet perm.**	**Celkový čas (ms)**	**Jednotky**	**Dokumenty**	**Top 20**	**Top 10**
Odsek	50	10	9841	115429	5082	100	55
Odsek	50	20	20426	240723	5398	182	99
Odsek	50	30	35251	269272	5239	248	138
Odsek	50	40	47600	417495	5419	346	190
Odsek	100	10	6298	44804	4080	78	42
Odsek	100	20	11103	115539	5066	141	77
Odsek	100	30	15238	135529	4951	181	101
Odsek	100	40	23537	223668	5232	241	134
Odsek	200	10	4495	18310	2922	61	34
Odsek	200	20	6551	51437	4505	106	59
Odsek	200	30	8931	68030	4468	137	78
Odsek	200	40	11504	115340	4867	176	101
Odsek	300	10	3669	10846	2135	54	30
Odsek	300	20	5300	33712	4038	90	49
Odsek	300	30	6737	43837	4002	117	66
Odsek	300	40	8410	75834	4566	150	86
Odsek	400	10	3112	7551	1688	49	28
Odsek	400	20	4452	25784	3716	79	45
Odsek	400	30	5587	31835	3680	108	61
Odsek	400	40	6885	55969	4250	133	77
Odsek	500	10	2975	5889	1408	44	24
Odsek	500	20	3997	19879	3322	71	40
Odsek	500	30	4925	25313	3374	99	56
Odsek	500	40	5907	42967	3999	121	70
Veta	-	3	9297	122568	4748	57	29
Veta	-	5	14810	209535	5097	92	47
Veta	-	10	36385	312509	5212	180	92

Tabuľka 7.4 – Počet podobností pre jednotlivé implementácie

Z výsledkov vyplývajú nasledujúce skutočnosti:

- Počet nájdených podobných jednotiek i dokumentov klesá so vzrastajúcim počtom slov v jednotke, respektíve s klesajúcim počtom použitých permutácií. To je očakávaný výsledok, pretože, ako už bolo spomínané v predchádzajúcej kapitole, oba tieto faktory znižujú veľkosť reprezentácie dokumentu (resp. jednotky), a teda aj pravdepodobnosť správneho označenia podobnosti.

- Zmenšovanie lingvistických jednotiek má menší význam vzhľadom k počtu nájdených podobných jednotiek i dokumentov ako zvyšovanie počtu permutácií. Skúsme porovnať jednotky, ktoré majú približne rovnako veľkú reprezentáciu jednotlivých dokumentov. Takéto jednotky definujeme tak, že ak označíme prvú jednotku A a druhú B, tak potom pre ne platí, že:

$$\frac{Počet\ slov_A}{Počet\ permutácií_A} = \frac{Počet\ slov_B}{Počet\ permutácií_B} \tag{7.1}$$

  Takisto pre ne logicky musí platiť, že celkový čas celého dotazu by mal byť približne rovnaký. Teraz, ak sa pozrieme napríklad na odsek s počtom slov 100 a počtom permutácií 10, a porovnáme ho s odsekom s počtom slov 200 a počtom permutácií 20, druhý menovaný dosahuje lepšie výsledky. Podobne je to i pri ostatných dvojiciach ako (100, 20) a (200, 40) a podobne. To je dané tým, že s vyšším počtom reprezentantov rastie pravdepodobnosť toho, že sa aspoň nejaké dva šindle budú zhodovať, a to najmä pri krátkych šindľoch, aké sme si zvolili, ale celkové dosiahnuté skóre podobných jednotiek bude veľmi nízke.
- Pomer celkového dosiahnutého skóre top 20, respektíve top 10 podobných jednotiek je vyšší u jednotiek z malým počtom permutácií. To je v zásade dané skutočnosťou, ako duplicitné dokumenty vznikajú, či už úmyselne alebo neúmyselne. Ide o to, že autor vezme malú časť textu z nejakého zdroja, dopíše kus vlastného textu, opäť použije časť textu z nejakého zdroja, atď. Preto, ak dokument rozdelíme na príliš veľké úseky, a zvolíme väčšie množstvo permutácií, bude sa nutne stávať, že mnoho šindľov bude vybratých práve z autorského textu, čo bude samozrejme celkové skóre znižovať.

Z uvedených skutočností vyplýva, že je vhodnejšie vydať sa cestou čo najmenších jednotiek, a zvoliť vyšší počet permutácií iba v prípade, že to dovoľujú časové nároky kladené na systém. Tie sme v kapitole stanovili tak, že priemerná doba odpovede systému by nemala presiahnuť 10 sekúnd. Medzi najvhodnejších kandidátov teda patria nasledujúce jednotky:

				Počet podobností		Celkové skóre	
Jedn.	**Počet slov**	**Počet perm.**	**Celkový čas (ms)**	**Jednotky**	**Dokumenty**	**Top 20**	**Top 10**
Odsek	50	10	9841	115429	5082	100	55
Odsek	100	10	6298	44804	4080	78	42
Veta	-	3	9297	122568	4748	57	92

Tabuľka 7.5 – Najvhodnejšie parametre implementácie

Všetky tri spĺňajú kritérium priemernej odpovede do 10 sekúnd, i keď odsek s počtom slov 50 a počtom permutácií 10, i veta s počtom permutácií tri sa tejto hranici blížia dosť výrazne. Vzhľadom k tomu, že reálne nasadenie systému bude fungovať ako webová aplikácia, a dnešné hardwarové riešenia serverov sú rozhodne výkonnejšie ako hardware použitý na spracovávanie týchto výsledkov, nemyslíme si, že by to malo znamenať zásadný problém.

Otázkou je, či zvoliť ako jednotku odsek s počtom slov 50, alebo vetu. Domnievame sa, že v našom prípade je o niečo výhodnejšie zvoliť odsek, a to z dôvodu, že veta je predsa len príliš malá jednotka, a často sa stáva, že v prácach na podobnú tému sa vyskytuje značné množstvo podobných formulácií, čo výsledky do istej miery skresľuje. Takisto prezentácia výsledkov vo webovom rozhraní je oveľa prehľadnejšia pre odsek – napríklad v prehľade podobných jednotiek pre vety býva často až príliš mnoho záznamov, čo na užívateľa bude pôsobiť mätúco.

7.5. Veľkosť indexu a časová náročnosť pridávania ďalších dokumentov do indexu

Veľkosť celého indexu (t.j. všetkých dát potrebných k detekcií duplicitných dokumentov) je pri nami zvolenej implementácií 1,48 GB. Index sa skladá z 11 barelov, ktoré majú každý inú veľkosť (dôsledok implementácie indexu, viď kapitola 5.6), štruktúr samotného indexu, a textovej reprezentácie dokumentov, ktorá má veľkosť cca. 374 MB.

Čo sa týka časovej náročnosti pridávania ďalších dokumentov do indexu, k 7500 dokumentom sme skúsili dvakrát pridať ďalších 200 dokumentov, čo podľa nás odráža napríklad počet záverečných prác jednej fakulty vždy v termíne obhajob. Dokumenty sme rozdelili na 4 barely po 50 dokumentov, v jednotlivých bareloch sa nachádzajú nasledujúce štruktúry:

Štruktúra	Popis	Priemerná veľkosť (kB)
doc.aux	Štruktúra s metadátami dokumentov	2921
ils.dta	Štruktúra invertovaných zoznamov	2429
prx.aux	Štruktúra obsahujúca informáciu o výskyte termov na konkrétnych dokumentoch	1788
trm.aux	Štruktúra slovníku termov	4034
Celkom		**11172**

Tabuľka 7.6 – Štruktúra jedného barelu

Časová náročnosť pridania všetkých barelov do indexu je nasledujúca:

- Priemerná doba vytvorenia barelov z dokumentov: 248 008 milisekúnd, čo je cca. 248 sekúnd, čo je cca. 4 minúty a 8 sekúnd.
- Priemerná doba začlenenia barelu do indexu je 22 411 milisekúnd, čiže cca. 22 sekúnd.

Pridávanie nových dokumentov je teda dostatočne rýchle na to, aby mohlo prebiehať v reálnej aplikácií prakticky kedykoľvek s minimálnym dopadom na užívateľov.

7.6. I/O nároky systému

Hodnotiť I/O nároky systému je vždy problematické, pretože, ako sme už zmieňovali v úvode tejto kapitoly, nikdy nie je úplne jasné, ako sa bude správať operačný systém, aké dáta si ponechá v hlavnej pamäti, ako rýchlo je schopný narábať s cache procesora, koľko má táto úrovní, a podobne. Ďalším problémom je fakt, že pri súčasnej veľkosti sa index pohodlne vojde do hlavnej pamäte celý, a teda teoreticky nebude potrebné pristupovať na externú pamäť nikdy.

Experiment bol koncipovaný podobne ako predchádzajúce, máme 3 testovacie kolekcie po 15 dokumentov, tento krát sledujeme počet I/O operácií (čítanie, zápis) a veľkosť prečítaných dát. Veľkosť zapísaných dát neuvádzame, pretože počet operácií zápisu je v našej implementácií vždy nulový. Na získanie výsledkov experimentu bol použitý voľne dostupný nástroj Process Monitor [70]. Výsledky experimentu uvádza tabuľka 7.7, výsledky sú uvádzané ako priemer pre jeden dokument.

	# read operácií	# write operácií	Veľkosť prečítaných dát (MB)
Kolekcia 1	83030	0	345
Kolekcia 2	84608	0	349
Kolekcia 3	70830	0	293
Priemer	79489	0	329

Tabuľka 7.7 – I/O Nároky systému

Čo sa týka vplyvu hlavnej pamäte, tak pokiaľ je index používame často, priemerný počet I/O operácií pre dokument s častým používaním klesá. Konkrétne, pri piatom spustení experimentu bol celkový počet potrebných I/O operácií nižší cca. o štvrtinu.

7.7. Počiatočná hodnota minimálneho počtu zhodných šindľov

Ako sme spomenuli v kapitole 6.7, užívateľovi kvôli zrozumiteľnosti ponúkneme radšej nastavenie minimálneho počtu zhodných šindľov, ako možnosť presne nastaviť Jaccardov koeficient, ktorý bude jednotky rozdeľovať na originály a duplikáty.

Práca [30] stanovuje optimálnu hodnotu aproximovaného Jaccardovho koeficientu na 0,21, táto hodnota podľa nej najlepšie rozdeľuje jednotky na podobné a rozdielne. Ak sa ale pozrieme na to, ako dospela k tejto hodnote, tak sa pri celkovom počte permutácií 10 snažila zistiť, aký minimálny počet zhodných šindľov by najlepšie rozdeľoval jednotky na originály a duplikáty, a nájdená hodnota 0,21 zodpovedá počtu zhodných šindľov 3. Vzhľadom k tomu, že sme dospeli k záveru, že budeme tiež používať počet permutácií 10, ponúkneme užívateľovi tento počet ako počiatočnú hodnotu aplikácie, samozrejme s tým, že si túto hodnotu bude môcť ľubovoľne zmeniť.

8. Porovnanie s dostupnými nástrojmi a zhodnotenie cieľov práce

8.1. Prehľad dostupných nástrojov

Porovnanie s inými dostupnými nástrojmi je pomerne zložitá záležitosť, pretože bohužiaľ značná časť nástrojov nie je voľne dostupná, ale distribučné podmienky týchto nástrojov sú nastavené tak, že sú k dispozícii iba platiacim zákazníkom, prípadne členom určitých skupín, ako napr. zamestnanci vzdelávacích inštitúcií a podobne. Medzi takéto nástroje patrí napríklad iThenticate [54], TurnItIn [36], systém Masarykovej Univerzity, Theses [66], či Eve2 [67].

Tým, ktorý vyvíja projekt Theses.cz ale založil portál, odevzdej.cz [68], ktorý taktiež slúži na kontrolu podobnosti zadaného dokumentu, problémom je, že nie je jasné, voči čomu sa zadané dokumenty porovnávajú, či aká je referenčná kolekcia veľká. Jediná informácia, ktorú užívateľ dostane, je „Overte si svoj textový dokument, či nie je podobný ďalším skúmaným textom a vybraným zdrojom z Internetu.“ [68]. Systém sa takisto veľmi ťažko hodnotiť ako „online“, pretože po nahraní súboru sa doba odpovede (ktorú užívateľ dostane prostredníctvom e-mailu) pohybuje v rozmedzí niekoľkých hodín.

Ďalším problémom je, že mnoho systémov nie je schopných pracovať nad určitou kolekciou dokumentov, ale ako referenčnú množinu dokumentov používajú celý internet (viď. kapitola 3.4 – Metódy založené na použití internetových vyhľadávačov). Medzi takéto (aspoň čiastočne – tzn. trial verzia, atp.) voľne dostupné nástroje patrí napríklad Plagiarism Finder [69] alebo Plagiarism Detector [36]. Porovnanie metódy použitej v tejto práci a systému Plagiarism Detector sa zaoberá práca [30], jej výsledky hovoria jednoznačne v prospech nami zvolenej metódy.

8.2. Porovnanie času behu dotazu

Možnosť, ktorá sa ale sama ponúka je porovnať výsledky dosiahnuté v tejto práci s prácou [30], a to preto, že použitý algoritmus je z logického hľadiska

rovnaký. Bohužiaľ sa to ukázalo ako pomerne zložitá záležitosť, pretože pamäťové nároky aplikácie vyvinutej ako časť práce [30] sú tak vysoké, že pri nastavení parametrov vyhľadávania tak, ako v predchádzajúcej kapitole, nie je možné index spracovať. Tá istá situácia nastáva pre index, ktorý má zhruba polovičnú veľkosť, i pre index, ktorý má veľkosť štvrtinovú. Index, ktorý má ale zhruba osminovú veľkosť (konkrétne 1000 dokumentov) už ale práca [30] spracovať dokáže, takže skúsme porovnať obe riešenia na takto veľkom indexe.

Experiment bol koncipovaný podobne ako predchádzajúce experimenty. Opäť sme zvolili tri testovacie kolekcie dokumentov, v každej testovacej kolekcii sa nachádza pätnásť dokumentov. Každú testovaciu kolekciu sme porovnávali s vytvoreným indexom, a toto porovnávanie sme zopakovali pre každú kolekciu 5 krát. Z týchto porovnaní sme následne vypočítali priemerné hodnoty potrebné na spracovanie jedného dokumentu. Tieto hodnoty uvádza tabuľka 8.1.

	Aplikácia vyvinutá v rámci tejto práce	Aplikácia vyvinutá v rámci práce [30]
Kolekcia 1	6115 ms	33158 ms
Kolekcia 2	5459 ms	30177 ms
Kolekcia 3	4499 ms	21253 ms
Priemer	5358 ms	28169 ms

Tabuľka 8.1 – Porovnanie času behu dotazu s prácou [30]

Z tabuľky je vidno, že aplikácia vyvinutá ako súčasť tejto práce je oproti aplikácií z práce [30] zhruba päťnásobne rýchlejšia, čo je očakávaný výsledok, keďže zložitosť algoritmu je o rád nižšia (viď. kapitola 6). Bohužiaľ, vzhľadom k pamäťovým nárokom aplikácie z práce [30], nebolo možné vykonať tento experiment na rozsiahlejších dátach, kde by sa pravdepodobne ukázal ešte markantnejší rozdiel.

8.3. Porovnanie I/O nárokov

Porovnanie sme opäť vykonávali na indexe, ktorý má zhruba osminovú veľkosť oproti indexu na ktorom boli vykonávané experimenty popisované v kapitole 7. Experiment je podobný experimentu z podkapitoly 7.6, máme tri testovacie kolekcie po 15 dokumentov, a všímame si počet I/O operácií (read, write) a celkovú veľkosť načítaných, resp. zapísaných dát. Výsledky uvádzame ako priemer pre jeden

dokument, na ich získanie bola opäť použitá aplikácia Performance Monitor [70]. Výsledky dosiahnuté aplikáciou vyvinutou v rámci tejto práce uvádza uvádza tabuľka 8.2, výsledky dosiahnuté aplikáciou vyvinutou v rámci práce [30] uvádza tabuľka 8.3.

	# read operácií	# write operácií	Celkový počet operácií	Veľkosť načítaných dát (MB)	Veľkosť zapísaných dát (MB)
Kolekcia 1	69353	0	69353	272	0
Kolekcia 2	78665	0	78665	309	0
Kolekcia 3	65281	0	65281	257	0
Priemer	71100	0	71100	279	0

Tabuľka 8.2 – Porovnanie I/O nárokov – aplikácia vyvinutá v rámci tejto práce

	# read operácií	# write operácií	Celkový počet operácií	Veľkosť načítaných dát (MB)	Veľkosť zapísaných dát (MB)
Kolekcia 1	110870	72040	182910	420	308
Kolekcia 2	131211	76652	207863	511	322
Kolekcia 3	102489	69114	171603	391	287
Priemer	114857	72602	187459	441	306

Tabuľka 8.3 – Porovnanie I/O nárokov – aplikácia vyvinutá v rámci práce [30]

Z tabuľky vyplýva, že aplikácia vyvinutá v rámci tejto práce má zhruba 2,5 krát menšie nároky čo do celkového počtu I/O operácií, a takisto zhruba 2,5 krát menšie nároky čo do počtu načítaných a zapísaných dát. Toto je očakávaný výsledok, keďže v kapitole 5 sme ukázali, že použitím invertovaných zoznamov budeme spracovávať iba dáta relevantné pre daný dotaz, a nie úplne všetky dáta tak ako práca [30].

8.4. Zhodnotenie cieľov práce

Prehľad cieľov práce uvádza kapitola 2, ich zhrnutie sa nachádza v podkapitole 2.7. Táto podkapitola uvádza ich zhodnotenie.

1) **Preštudovať a zhrnúť existujúce postupy používané na detekciu plagiátov či duplicitných dokumentov**
   Prehľad existujúcich postupov sa nachádza v kapitole 3
2) **Z týchto postupov vybrať najvhodnejší postup a upraviť ho tak, aby spĺňal nasledujúce požiadavky:**

   a. **Detekcia čo najväčšieho množstva plagiátov bez uvedených zdrojov**
   b. **Minimalizovať množstvo falošných negatív a falošných pozitív**

   Popis výberu vhodnej metódy je uvedený v kapitole 5. Vhodnosť metódy na detekciu plagiátov overila už práca [30], my sme metódu upravili tak aby lepšie vyhovovala integrácií s internetovým vyhľadávačom a mala menšie časové nároky.

3) **Implementovať tento postup a navrhnúť systém na detekciu plagiátov tak, aby:**
   a. **Priemerná doba odpovede systému neprekročila 10 sekúnd**

      Po vyladení parametrov systému sa systém správa tak, že jeho priemerná doba odpovede neprekračuje 10 sekúnd, a pritom stále poskytuje relevantnú úroveň detekcie.
   b. **Bolo možné vykonávať dynamické zmeny nad množinou referenčných dokumentov**

      Systém bol navrhnutý tak, aby bolo možné vykonávať dynamické zmeny, popis časových nárokov pri prevedení týchto zmien uvádza kapitola 7.5.
   c. **Systém bol užívateľsky prívetivý, čo značí predovšetkým poskytnúť užívateľovi výsledky v ľahko čitateľnej forme, a podporu základných formátov dokumentov.**

      Systém sme sa snažili navrhnúť tak, aby bol čo najintuitívnejší, podpora základných formátov je takisto implementovaná. Krátky popis užívateľského rozhrania uvádza kapitola 6.6, kompletnú užívateľskú príručku je možné nájsť v prílohe tejto práce.
4) **Otestovať systém na dátach čo najviac sa približujúcim reálnym dátam z akademického prostredia, medziiným:**
   a. **Otestovať ako sa systém správa pri zmenách najdôležitejších parametrov z hľadiska časovej náročnosti**
   b. **Zhodnotiť časové a priestorové nároky systému**

Časové nároky systému pri zmenách parametrov skúma kapitola 7, konečné parametre sme zvolili tak aby bol splnený bod 2a. Priestorové nároky systému sú zhodnotené v kapitole 7.5.

5) **Porovnať implementovaný systém s inými dostupnými systémami riešiacimi podobnú úlohu**

Toto porovnanie sme vykonali v rámci tejto kapitoly, najvhodnejší systém na porovnávanie sa ukázala byť aplikácia vyvinutá v rámci práce [30], oproti nej naše riešenie dosahovalo niekoľkonásobne lepšie výsledky.

9. Záver

Táto práca sa zaoberá problémom efektívnej detekcie duplicitných dokumentov. Ponúka prehľad existujúcich metód a z týchto vyberá vhodnú metódu vzhľadom k zadefinovaným cieľom práce. Túto metódu ďalej upravuje tak, aby zlepšila jej časovú zložitosť, ktorú oproti pôvodnej metóde zlepšila o rád. Práca obsahuje detailný popis algoritmu zvolenej metódy, a jedným z jej výsledkov je plne funkčná (webová) aplikácia spĺňajúca definované ciele. Táto aplikácia je v rámci súčasne používaných metód unikátna tým, že funguje „online“, teda ponúka užívateľovi výsledky v reálnom čase. Implementácia je nasadená do reálnej prevádzky a je dostupná na adrese http://egothor.cythres.cz/egothorWebApplication.

Prínosom je aj možnosť vykonávať dynamické zmeny nad množinou indexovaných dokumentov (pridávanie, mazanie), čo odstraňuje nutnosť vytvárať vždy celý index odznova.

Implementácia bola podrobená testom na reálnych dátach (cca. 7500 záverečných prác z repozitára Univerzity Karlovy v Prahe) tak, aby boli nájdené jej optimálne parametre vzhľadom k času i presnosti spracovania. Ukázalo sa, že najlepšie výsledky sú dosiahnuté, ak je veľkosť bucketu v implementácií slovníku termov rovná cca. 2^8, a ako lingvistická jednotka veta a počet permutácií 3, alebo odsek obsahujúci cca. 50 slov a počet permutácií 10. Takisto sa ukázalo, že index – sekvenčná implementácia slovníku termov je vhodnejšia ako hashovaná implementácia.

Na záver sme zmienili niektoré komerčne i nekomerčne používané nástroje a porovnali našu implementáciu s implementáciou riešiacou podobnú úlohu, z čoho naša implementácia vyšla zhruba 2,5 krát lepšie, čo sa týka I/O nárokov a zhruba päťnásobne lepšie z hľadiska časovej náročnosti.

Možných smerov pre rozvíjanie použitej metódy je niekoľko, od lepšieho predspracovania textu (parsovanie dokumentov, možnosť porovnávať dokumenty písané v inom jazyku) cez detekciu správne citovaného textu až po pokusy s inými implementáciami slovníku termov. Najväčší potenciál ale vidíme v predspracovaní

textu, keďže parsovanie rôznych formátov a správne rozpoznávanie diakritiky, lingvistických jednotiek a podobne je samo osebe pomerne zložitý problém.

Zoznam použitých zdrojov

[1] LYNCH, J. The Perfectly Acceptable Practice of Literary Theft: Plagiarism, Copyright, and the Eighteenth Century. In *Colonial Williamsburg: The Journal of the Colonial Williamsburg Foundation*, Winter 2002-2003, vol. 24, no. 4, pp. 51–54.

[2] ŠIPOŠ, G. *Sme opúšťa novinárka pre plagiátorstvo.* 7.6.2007 [cit. 2011-03-10]. <http://spw.blog.sme.sk/c/98797/SME-opusta-novinarka-pre-plagiatorstvo.html>

[3] ŠIPOŠ, G. *Druhý najcitovanejší politológ mal problémy s plagiátorstvom*. 21.12.2010 [cit. 2011-03-10]. <http://spw.blog.sme.sk/c/251113/Druhy-najcitovanejsi-politolog-STV-mal-problemy-s-plagiatorstvom.html>

[4] ŠIMIČKOVÁ, J. *Plagiátori.* 3.12.2010 [cit. 2011-03-10]. <http://plus7dni.pluska.sk/plus7dni/zaujalo-nas/2010/12/plagiatori.html>

[5] *Plagiáty stály profesora funkci. Zatím dočasně*. 11.11.2010 [cit. 2011-03-10]. <http://www.lidovky.cz/plagiaty-staly-profesora-funkci-zatim-docasne-fqm-/ln_noviny.asp?c=A100513_000107_ln_noviny_sko&klic=236924&mes=100513_0>

[6] *Proděkan právnické fakulty v Plzni opsal dizertační práci*. 19.9.2009 [cit. 2011-03-10]. <http://www.lidovky.cz/prodekan-pravnicke-fakulty-v-plzni-opsal-dizertacni-praci-pnh-/ln_domov.asp?c=A090919_140948_ln_domov_mev>

[7] *Studentka objevila další Pernesův plagiát, opsal dějiny Lidových novin*. 6.5.2010 [cit. 2011-03-10]. <http://www.lidovky.cz/studentka-objevila-dalsi-pernesuv-plagiat-opsal-dejiny-lidovych-novin-130-/ln_domov.asp?c=A100506_195342_ln_domov_nev>

[8] *Plagiarism.* [cit. 2011-11-23]. <http://en.wikipedia.org/wiki/Plagiarism>

[9] KULATHURAMAIYER, N. - MAURER, H. Fighting plagiarism and IPR violation: why is it so important?. In *Information Services & Use*, 2007, vol.27, no. 4, pp. 185-191

[10] *Types of plagiarism*. [cit. 2011-03-12]. <http://www.plagiarism.org/plag_article_types_of_plagiarism.html>

[11] MAURER, H. - KAPPE, F. - ZAKA, B. Plagiarism - A Survey. In *Journal of Universal Computer Sciences*, 2006, vol. 12, no. 8, pp. 1050-1084.

[12] NAH, F. A study on tolerable waiting time: how long are Web users willing to wait? In *Behaviour & Information Technology*. 2004, vol. 23, no. 3, pp. 153-163.

[13] CARD, S. K. - ROBERTSON, G. G. - MACKINLAY, J. D. The information visualizer: An information workspace. In *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems.* 1991, pp. 181-186.

[14] NIELSEN, J. *Website Response Times.* c2007 [cit. 2011-03-17]. <http://www.useit.com/papers/responsetime.html>

[15] HOXMEIER, J.A. - DICESARE, C. System response time and user satisfaction: an experimental study of browser-based applications. In *Proceedings of the Americas Conference on Information Systems.* Long Beach, California: Association for Information Systems, 2000, pp. 140-145.

[16] RAMSAY, J. - BARBESI, A. - PREECE, J. A psychological investigation of long retrieval times on the world wide web. In *Interacting with Computers.* 1998, vol. 10, no. 1, pp. 77-86.

[17] ROSE, G. - EVARISTO, R. – STRAUB, D. Culture and Consumer Responses to Web Download Time: A Four-Continent Study of Mono and Polychronism. In *IEEE Transactions on Engineering Management.* 2003, vol. 50, no. 1, pp. 31-44.

[18] WU, G. The mediating role of perceived interactivity in the effect of actual interactivity on attitude toward the website. In *Journal of Interactive Advertising.* 2005, vol. 5, no. 2.

[19] ANTONIDES, G. - VERHOEF, P.C. - VAN AALST, M. Consumer perception and evaluation of waiting time: a field experiment. In *Journal of Consumer Psychology*. 2002, vol. 12, no. 3, pp. 193-202.

[20] MILLER, R. B. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I.* New York: ACM, 1968, pp. 267-277.

[21] SELVIDGE, P. Examining tolerance for online delays. In *Usability News*. 2003, vol. 5, no 1.

[22] OTTENSTEIN, K. J. An Algorithmic Approach to the Detection and Prevention of Plagiarism. In *ACM SIGCSE Bulletin*. 1976, vol. 8, no. 4, pp. 30-41.

[23] MANBER, U. Finding similar files in a large file system. In *Proceedings of the 1994 USENIX Conference on USENIX Winter 1994 Technical Conference, San Francisco, California*. Berkeley: USENIX Association. 1994, pp. 1-10.

[24] BERGHEL, H. - O'GORMAN, L. Protecting Ownership Rights Through Digital Watermarking. In *IEEE Computer*. 1996, vol. 29, no. 7, pp. 101-103.

[25] BRASSIL, J.T. - LOW, S. - MAXEMCHUK, N.F. Copyright Protection for the Electronic Distribution of Text Documents. In *Proceedings of the IEEE*. 1999, vol. 87, no. 7, pp. 1181-1196.

[26] BRIN, S. – DAVIS, J. - GARCIA-MOLINA, H. Copy Detection Mechanisms for Digital Documents. In *Proceedings of the ACM SIGMOD Annual Conference*. New York: ACM, 1995, pp. 398-409.

[27] HEINTZE, N. Scalable Document Fingerprinting. In *Proceedings of the Second USENIX Workshop on Electronic Commerce, Oakland, California*. 1996, pp. 191-200.

[28] FINKEL, R. A. – ZASLAVSKY, A. – MONOSTORI, K. – SCHMIDT, H. Signature Extraction for Overlap Detection in Documents. In *Proceedings of the twenty-fifth Australasian conference on Computer science*. 2002, vol. 4, pp. 59-64.

[29] BRODER, A. Z. – GLASSMAN, S. C. – MANASSE, M. S. – ZWEIG, G. Syntactic Clustering of the Web. In *Computer Networks and ISDN Systems*. 1997, vol. 29, no. 8-13, pp. 1157-1166.

[30] DUFKOVÁ, K. *Dynamická detekce plagiátů*. Praha, 2008. Diplomová práce na MFF UK, katedra Softwarového inženýrství. Vedoucí diplomové práce RNDr. Leo Galamboš, Ph.D.

[31] SCHLEIMER, S. – WILKERSON, D. S. – AIKEN, A. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the 2003 ACM*

SIGMOD International Conference on Management of Data. New York: ACM, 2003, pp. 76-85.

[32] BRODER, A. Z. Identifying and filtering near-duplicate documents. In *COM '00: Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*. Heidelberg: Springer, 2000, pp. 1–10.

[33] NIEZGODA, S. – WAY, T. P. *SNITCH:* A Software Tool for Detecting Cut and Paste Plagiarism. In *ACM SIGCSE Bulletin.* 2006, vol. 38, no 1, pp. 51-55.

[34] POTTHAST, M. - BARRÓN-CEDEÑO, A. – STEIN, B. - ROSSO, P. Cross – language plagiarism detection, In *Language Resources and Evaluation (30 January 2010).* 2010, vol. 45, no. 1, pp. 45-62.

[35] TASHIRO, T. – UEDA, T. – HORI, T. – HIRATE, Y. – YAMANA, H. *EPCI:* Extracting Potentially Copyright Infringement Texts from the Web, In *Proceedings of the World Wide Web (WWW)*, New York: ACM. 2007, pp. 1151–1152.

[36] *Plagiarism Detector.* [cit. 2011-06-05]. <http://www.plagiarism-detector.com>

[37] *SafeAssign.* c2007 [cit. 2011-06-05]. <http://www.mydropbox.com>.

[38] *TurnItIn.* c1998 [cit. 2011-06-05]. <http://www.turnitin.com>.

[39] BAEZA-YATES, R. - RIBEIRO-NETO, B. *Modern Information Retrieval*. First edition. Harlow, England: Addison Wesley, 1999, ISBN 0-201-39829-X.

[40] MANNING, C. – RAGHAVAN, P. – SCHÜTZE, H. *Introduction to Information Retrieval.* First edition. New York: Cambridge University Press, 1998. ISBN 978-0-521-86571-5.

[41] SHIVAKUMAR, N. - GARCIA-MOLINA, H. SCAM : A copy detection mechanism for digital documents. In *Proceedings of the Second Annual Conference on the Theory and Practice of Digital Libraries*. 1995, vol. 24, no. 2, pp. 398-409.

[42] HOAD, T. C. – ZOBEL, J. Methods for Identifying Versioned and Plagiarised Documents. In *Journal of the American Society for Information Science and Technology*. 2003, vol. 54, no. 3, pp. 203-215.

[43] MANKU, G. S. – JAIN, A. – SARMA, A. D. Detecting Near-Duplicates for Web Crawling. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*. New York: ACM, 2007, pp. 141-150.

[44] CHARIKAR, M. Similarity Estimation Techniques from Rounding Algorithms. In *Proceedings of 34th Annual Symposium on Theory of Computing*. New York: ACM, 2002, pp. 380-388.

[45] CHOWDHURY, A. – FRIEDER, O. – GROSSMAN, D. – MCCABE, M. C. Collection Statistics for Fast Duplicate Document Detection. In *ACM Transactions on Information Systems (TOIS)*. 2002, vol. 20, no. 2, pp. 171-191.

[46] KOLCZ, A. – CHOWDHURY, A. – ALSPECTOR, J. Improved Robustness of Signature-based Near-replica Detection via Lexicon Randomization. *In Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*. New York: ACM, 2004, pp. 605-610.

[47] MONOSTORI, K. – ZASLAVSKY, A. – SCHMIDT, H. Document Overlap Detection System for Distributed Digital Libraries. In *Proceedings of the fifth ACM conference on Digital libraries*, New York: ACM, 2000, pp. 226-227.

[48] UZUNER, Ö. – KATZ, B. – NAHNSEN, T. Using Syntactic Information to Identify Plagiarism. *Proceedings of the 2nd Workshop on Building Educational Applications Using NLP*. Stroudsburg: Association for Computational Linguistics, 2005, pp. 37-44.

[49] ZINI, M. – FABBRI, M. – MONEGLIA, M. – PANUNZI, A. Plagiarism Detection through Multilevel Text Comparison. In *Proceedings of the International Conference on Automated Solutions for Cross Media Content and Multi-Channel Distribution (AXMEDIS)*. Leeds: IEEE Computer Society, 2006, pp. 181–185.

[50] LEUNG, C. – CHAN, Y. A Natural Language Processing Approach to Automatic Plagiarism Detection. In *Proceedings of the 8th ACM SIGITE conference on Information technology education*. New York: ACM, 2007, pp. 213-218.

[51] METZLER, D. – BERNSTEIN, Y. – CROFT, W. – MOFFAT, A. – ZOBEL, J. Similarity Measures for Tracking Information Flow, In *Proceedings of ACM*

International Conference on Information and Knowledge Management (CIKM). New York: ACM, 2005, pp. 517–524.

[52] CESKA, Z. – TOMAN, M. – JEZEK, K. Multilingual Plagiarism Detection. In *Proceedings of the International Conference on Artificial intelligence: Methodology, Systems, and Applications.* Heidelberg: Springer, 2008, pp. 83–92.

[53] PERA, M. S. - NG Y. SimPaD: A word-similarity sentence-based plagiarism detection tool on Web documents. In *Web Intelligence and Agent Systems.* 2011, vol. 9, no. 1, pp. 27-41

[54] *iThenticate*. c1998 [cit. 2011-10-14]. <http://www.ithenticate.com>

[55] PORTER, M. F. 1980. An algorithm for suffix stripping. In *Readings in Information Retrieval*. San Francisco: Morgan Kaufmann Publishers Inc., 1997, pp. 313-316.

[56] TOMAN, M. - TESAR, R. - JEZEK, K. Influence of Word Normalization on Text Classification. In: *Proceedings of the 1st International Conference on Multidisciplinary Information Sciences & Technologies, Merida, Spain.* 2006, vol. 2, pp. 354–358.

[57] GALAMBOŠ, L. *Dokumentace Egothoru*, c2006 [cit. 2011-06-06]. <http://www.egothor.org/docs/e2.pdf>

[58] PODHORNÝ, J. *Transakce ve fulltextovém vyhledávacím stroji*. Praha, 2007. Diplomová práce na MFF UK, katedra Softwarového inženýrství. Vedoucí diplomové práce RNDr. Leo Galamboš, Ph.D.

[59] HENZINGER, M. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR '06)*. New York: ACM, 2006, pp. 284-291.

[60] CHAKRABARTI, S. *Mining the Web - Discovering Knowledge from Hypertext Data*. First edition. San Francisco: Morgan Kaufmann Press Inc., 2003. ISBN 1-55860-754-4.

[61] BARRÓN-CEDEÑO, A. - ROSSO, P. On Automatic Plagiarism Detection Based on n-Grams Comparison, In *Proceedings of the European Conference on IR Research on Advances in Information Retrieval (ECIR)*. Heidelberg: Springler, 2009, pp. 696–700.

[62] GALAMBOŠ, L. Dynamic inverted index maintenance. In *International Journal Of Computer Science*. 2006, vol. 1, no. 1, pp. 157-162.

[63] *Nova. The University of Newcastle Digital Repository*. c2007 [cit. 2011-07-27]. <http://nova.newcastle.edu.au/vital/access/manager/Index>.

[64] *CSU Libraries: Digital Repository*. c2009 [cit. 2011-07-27]. <http://digitool.library.colostate.edu/R?RN=664196436>.

[65] *Texas A&M University Libraries*. [cit. 2011-07-27]. <http://repository.tamu.edu/community-list>.

[66] *Vysokoškolské kvalifikační práce*. [cit. 2011-10-12]. < http://theses.cz>.

[67] *EVE2 Plagiarism Detection for Teachers*. c2007 [cit. 2011-10-12]. <http://www.canexus.com>.

[68] *Seminární a školní práce*. [cit. 2011-10-12]. < http://www.odevzdej.cz/>.

[69] *Plagiarism Finder*. [cit. 2011-10-12]. < http://www.plagiarismfinder.de>.

[70] RUSSINOVICH, M. – COGGSWELL, B. *Process Monitor*. 16.8.2011 [cit. 2011-11-29]. < http://technet.microsoft.com/en-us/sysinternals/bb896645>.

[71] *MyFaces Tomahawk*. [cit. 2011-09-08]. <http://myfaces.apache.org/tomahawk/index.html>.

Zoznam tabuliek

Obsah DVD

Na priloženom DVD sa nachádzajú nasledujúce položky:

- Text diplomovej práce vo formáte PDF
- Užívateľská dokumentácia k vyvinutej aplikácii - Egothor online duplicity finder tool
- Projekt egothor2 obsahujúci zdrojový kód jadra aplikácie
- Projekt egothorWebApplication obsahujúci zdrojový kód užívateľskej aplikácie
- Programátorská dokumentácia k obom týmto projektom
- Inštalačná príručka