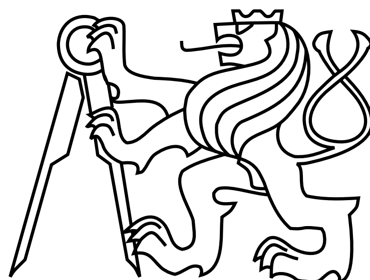
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačové grafiky a interakce



Diplomová práce

Zjišťování podobností mezi dokumenty ve formátu LaTeX

Martin Rendla

Vedoucí práce: Ing. Ondřej Guth

Studijní program: Otevřená informatika, strukturovaný, Magisterský

Obor: Softwarové inženýrství

11. května 2014

Poděkování

Rád bych poděkoval Ing. Ondřeji Guthovi za cenné podněty, věcné připomínky a vstřícnost při konzultacích. Dále děkuji své rodině za podporu.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze, 11. května 2014 ....

Abstract

The subject of this thesis is to develop an application for plagiarism detection in documents in LaTeX format. What's special about this detection is that the content is ignored and the files are only compared on a syntax basis. Its purpose is to detect plagiarism in a situation where the text is a part of the assignment to make sure everybody processes the text to the LaTeX format independently without copying others. In addition, the thesis provides an analysis of the subject matter including practical experiments.

Abstrakt

Předmětem práce je vytvoření aplikace na detekci plagiátů v dokumentech ve formátu LaTeX. Detekce je zvláštní v tom, že je ignorován obsah a soubory jsou poměřovány pouze na základě syntaxe. Účelem je pomoci odhalit plagáty v situaci, kdy je text součástí zadání a je třeba dohlédnout, aby tento text do formátu LaTeX zpracoval každý sám bez kopírování od ostatních. Kromě vývoje aplikace se práce zabývá i rozborem problematiky včetně praktických experimentů.

Obsah

1 Úvod **1**

2 Popis problému, specifikace cíle **3**

3 Existující řešení a postupy **5**

3.1 Tokenizace 5
3.2 Hashing 5
3.3 Vyhledávání řetězců 6
3.3.1 Algoritmy pro vyhledávání vzorku v textu 6
3.3.1.1 Rabinův-Karpův algoritmus 6
3.3.1.2 Knuttův-Morrisův-Prattův algoritmus 6
3.3.1.3 Boyerův-Mooreův algoritmus 7
3.3.2 Konečné automaty 7
3.3.3 Suffixové stromy 8
3.4 Základní dělení systémů na detekci plagiátů 8
3.4.1 Intrinsické nástroje 9
3.4.2 Extrakorpální nástroje 9
3.4.3 Intrakorpální nástroje 9
3.5 Detekce plagiátů ve volném textu 9
3.5.1 Lematizace 9
3.5.2 Bag of words 10
3.5.3 Fingerprinting 10
3.5.4 Stylometrická analýza 10
3.5.5 Existující nástroje 11
3.5.5.1 COPS 11
3.5.5.2 CHECK 11
3.6 Detekce plagiátů ve zdrojovém kódu 12
3.6.1 Softwarové metriky 12
3.6.2 Počítání atributů 12
3.6.3 Structure profile 13
3.6.4 Existující nástroje 13
3.6.4.1 Plague 13
3.6.4.2 YAP3 13

4 Návrh řešení 15
4.1 Volba reprezentace dokumentu . . . 15
4.1.1 Structure profile . . . 15
4.1.2 Syntaktický strom . . . 16
4.1.3 Document Object Model . . . 16
4.1.4 Fingerprinting . . . 16
4.2 Volba fingerprintu . . . 16
4.2.1 Bag of Words . . . 16
4.2.2 N-gramy . . . 17
4.2.3 Vektor výskytů podřetězců . . . 17
4.3 Návrh rozdělění aplikační logiky . . . 17
4.4 Volba algoritmu pro počítání výskytů podřetězců . . . 18
4.4.1 Konečný faktorový automat . . . 18
4.4.2 Suffix tree a suffix trie . . . 19
4.4.2.1 Suffix tree . . . 21
4.4.2.2 Suffix trie . . . 22
4.5 Volba filtrů podřetězců . . . 22
4.6 Volba metriky . . . 24
4.6.1 Kosinová podobnost . . . 24
4.6.2 Manhattanská vzdálenost . . . 24
4.6.3 Eukleidovská vzdálenost . . . 24
4.6.4 Minkowského vzdálenost . . . 24
4.6.5 Mahalanobisova vzdálenost . . . 25
4.6.6 Čebyševova vzdálenost . . . 25

5 Popis implementace řešení 27
5.1 Volba programovacího jazyka . . . 27
5.2 Implementace struktury tříd . . . 27
5.3 Lexikální analyzátor . . . 30
5.3.1 Regulární výrazy . . . 30
5.3.1.1 Matcher . . . 30
5.3.1.2 Capturing groups . . . 30
5.4 Vytvoření číselné reprezentace dokumentu . . . 31
5.5 Konstrukce konečného faktorového automatu . . . 32
5.5.1 Konstrukce nedeterministického konečného faktorového automatu . . . 32
5.5.1.1 Implementace počátečního stavu . . . 33
5.5.1.2 Implementace ostatních stavů . . . 33
5.5.2 Převod na deterministický konečný automat . . . 33
5.5.2.1 Množina existujcích stavů . . . 34
5.5.2.2 Konstrukce vektoru počtů výskytů . . . 34
5.5.3 Problémy s rychlostí algoritmu . . . 34
5.6 Konstrukce suffixového stromu . . . 34
5.6.1 Implementace datové struktury . . . 34
5.6.1.1 Implementace uzlu . . . 34
5.6.1.2 Implementace hrany . . . 35
5.6.1.3 Pomocná třída ActivePoint . . . 35

5.6.2 Implementace Ukkonenova algoritmu . . . 35
5.6.2.1 increaseActiveLength . . . 35
5.6.2.2 addNode . . . 37
5.6.2.3 addNodeToActivePoint . . . 37
5.6.2.4 append . . . 37
5.6.2.5 split . . . 37
5.6.2.6 canonizeActivePoint . . . 37
5.6.3 Úpravy oproti původnímu Ukkonenovu algoritmu . . . 38
5.6.4 Součet počtu výskytů . . . 38
5.7 Implementace filtrů . . . 38
5.8 Metriky porovnávání . . . 39

6 Experimenty **41**
6.1 Vzdálenost versus vážená vzdálenost . . . 41
6.2 Test vlivu procentuálních změn na metriku . . . 41
6.2.1 Tvorba testovací sady . . . 42
6.2.2 Výsledky testu . . . 42
6.2.2.1 Srovnání filtrů . . . 42
6.2.2.2 Srovnání metrik . . . 43
6.3 Test vlivu přítomnosti společného bloku textu na metriku . . . 44
6.3.1 Tvorba testovací sady . . . 44
6.3.2 Výsledky testu . . . 44
6.3.2.1 Srovnání filtrů . . . 44
6.3.2.2 Srovnání metrik . . . 46
6.4 Zhodnocení vhodné konfigurace . . . 47

7 Testování **51**
7.1 Volba testovacích prostředků . . . 51
7.1.1 Unit testy . . . 51
7.1.2 Systémové testy . . . 51
7.2 Implementace unit testů . . . 52
7.2.1 Třída LatexLexer . . . 52
7.2.1.1 Metoda tokenize . . . 52
7.2.2 Třída Alphabet . . . 52
7.2.2.1 Metoda getCharacter . . . 52
7.2.3 Třída Nfa . . . 53
7.2.3.1 Metoda construct . . . 53
7.2.4 Třída Dfa . . . 53
7.2.5 Třída SuffixTree . . . 53
7.2.5.1 Metoda construct . . . 53
7.2.6 Třída Corpus . . . 54
7.2.6.1 Metoda calculateEucleidianDistances . . . 54
7.2.6.2 Metoda calculateTaxicabDistances . . . 54
7.2.6.3 Metoda getDocumentComparisons . . . 54
7.3 Realizace systémových testů . . . 54
7.3.1 Tvorba testovacích dat . . . 55

7.3.2 Podpůrná aplikace na tvorbu plagiátů 55
7.3.2.1 Změny odsazení 55
7.3.2.2 Náhrada textu náhodnou kombinací slov 55
7.3.2.3 Odstranění komentářů 55
7.3.2.4 Prohození řádků 56
7.4 Testovací data 56
7.5 Výstupy testů 57
7.5.1 Test na množině úplně rozdílných souborů 58
7.5.2 Test na množině různých souborů se stejným obsahem 58
7.6 Zhodnocení 59

8 Závěr **61**

9 Seznam použitých zkratek **65**

A Uživatelská příručka **67**
A.1 Formát vstupu 67
A.1.1 Volba filtrů 67
A.1.2 Volba metriky 67
A.1.3 Volba formátu výstupu 68
A.1.4 Příklad vstupu 68
A.2 Formát výstupu 68

B Obsah přiloženého CD **69**

Seznam obrázků

4.1 Faktorový NFA s epsilon přechody pro dvojici řetězců *abca* a *cba* 20
4.2 Faktorový NFA bez epsilon přechodů pro dvojici řetězců *abca* a *cba* 20
4.3 Faktorový DFA pro dvojici řetězců *abca* a *cba* 20
4.4 Suffix trie pro řetězec *abdabc* 23
4.5 Suffix tree pro řetězec *abdabc* 23

5.1 Class diagram - PlagiTeX 28
5.2 Activity diagram - konstrukce suffixového stromu 36

6.1 Graf - Test vlivu procentuálních změn na metriku, Eukleidova vzdálenost . . 45
6.2 Graf - Test vlivu procentuálních změn na metriku, Manhattanská vzdálenost . 45
6.3 Graf - Test vlivu přítomnosti společného bloku textu na metriku, Eukleidovská vzdálenost (soubor číslo 5 má s originálem společný blok textu) 48
6.4 Graf - Test vlivu přítomnosti společného bloku textu na metriku, Manhattanská vzdálenost (soubor číslo 5 má s originálem společný blok textu) 49

7.1 PlagiTex Plagiator - screenshot 56
7.2 Graf výsledků - Test na množině úplně rozdílných souborů (plagiáty jsou barevně odlišeny) 60
7.3 Graf výsledků - Test na množině různých souborů se stejným obsahem (plagiáty jsou barevně odlišeny) 60

Seznam tabulek

3.1 Knuttův-Morrisův-Prattův algoritmus – Tabulka částečných shod pro vzorek *abcabbcabc* 7
3.2 Boyerův-Mooreův algoritmus – Tabulka pro pravidlo špatného znaku pro vzorek *abcabbcabc* v textu *ababcdeadebcededacba* 7

5.1 Regulární výrazy k rozpoznávání lexikálních elementů 31
5.2 Ukázka číslených reprezentací tokenů 32

7.1 Výsledky testu na množině úplně rozdílných souborů 58
7.2 Výsledky testu na množině různých souborů se stejným obsahem 59

Kapitola 1

Úvod

Nejjednodušší formou plagiátu je odevzdání cizí práce bez jakýchkoli změn. Často však bývají plagiátoři rafinovanější a kombinují buďto text více prací, nebo cizí text s vlastním. Kromě toho existuje celá řada postupů, jak plagiát zamaskovat. U běžného textu to může být třeba změna větné skladby nebo přidávání slov navíc. U zdrojového kódu existují metody refaktorování kódu, které kód navenek změní, ale zachovají jeho funkčnost. Hranice mezi plagiátem a originálem není pevně stanovena, je tedy v určitých případech možné považovat práci, která je částečně převzatá, ale je v ní velké množství změn a původního obsahu, za originál. Proto by měl finální rozhodnutí, je-li dílo plagiát, mít člověk, a systém na detekci plagiátu použít jen jako nástroj.

Kapitola 2

Popis problému, specifikace cíle

Cílem práce je vytvořit aplikaci sloužící k odhalování plagiátů v dokumentech ve formátu LaTeX. Nástrojů na odhalování plagiátů v tomto typu dokumentů již bylo několik vytvořeno, zásadní rozdíl je však v tom, na co se při detekci zaměřujeme a co považujeme za plagiát. Doposud vytvořené nástroje na detekci poměřovaly dokumenty na základě obsahu a syntaxi buďto ignorovaly, nebo používaly pouze jako pomocnou informaci. Aplikace, jejíž vývoj je předmětem této práce, rozpoznává plagiáty na základě syntaxe a obsah textu ignoruje. Aplikace totiž bude zpracovávat množinu dokumentů o stejném nebo podobném obsahu (obsah dokumentu je součástí zadání) a jejím úkolem je určit, které dokumenty byly zpracovány do LaTeXu samostatně a které byly okopírovány. Důležitou součástí práce je rozbor existujících řešení, návrh vlastních řešení, jejich implementace a experimentální pozorování. Bude realizováno několik různých variant řešení, jejichž srovnáním v testech určíme finální podobu aplikace.

Kapitola 3

Existující řešení a postupy

Tato kapitola se zabývá existujícími postupy a nástroji na detekci plagiátů. Začíná seznámením se základy problematiky, jako jsou například způsoby reprezentace dokumentu, zpracování a porovnávání dokumentů. Následuje část věnovaná základnímu rozdělení systémů na detekci plagiátů podle různých kritérií, jako je třeba typ porovnávaných dokumentů. Hlavním dvěma kategoriím, detekci ve volném textu a detekci ve zdrojovém kódu, jsou věnovány samostané podkapitoly. V těch se nejprve seznámíme s algoritmy a postupy, které jsou běžně využívány v dané kategorii detekčních nástrojů a na závěr je představeno několik konkrétních existujících systémů a řešení.

3.1 Tokenizace

Dokumenty podezřelé z plagiátorství nebývají zpravidla porovnávány v jejich hrubé, nezpracované formě. Než dojde na samotné vyhodnocování podobností, bývají soubory předzpracovány. Prvním krokem při zpracování dokumentu je u velké části nástrojů na detekci plagiátů tokenizace. Tokenizace je převod vstupní posloupnosti znaků na posloupnost tokenů [2]. Tokenizace probíhá nejčastěji sekvenčně, od začátku do konce vstupního textu. Token bývá definován typem a hodnotou, v obyčejném textu může být typ tokenu třeba slovo nebo interpunkce, u zdrojového kódu může být typem tokenu třeba identifikátor nebo číslo. Lexikální analyzátor (tokenizer) se zpravidla snaží v textu rozpoznat vzorek, který může být vyjádřen různými způsoby, například regulárním výrazem nebo konečným automatem.

3.2 Hashing

Hashing je výpočetní postup, který přiřadí datům číselnou reprezentaci o pevně dané délce. Narozdíl od kódování není možné z hashe zpětně vypočítat původní data a pro různá data může vycházet stejný hash. Typický příklad je číselná reprezentace slova či části textu. Hashing má využití při vyhledávání a porovnávání řetězců, používá se také u datových struktur, jako je například hashovací tabulka. Při porovnávání a vyhledávání řetězců se často používa takzvaný rolling hash. Principem rolling hashe je, že v textu se posouvá okénko o pevně dané délce od začátku ke konci a pro každý podřetězec se spočítá hash. Ten je možné přepočítat

na základě hashe předchozího okénka a posledního znaku nového okénka s konstantní složitostí. Nejznámějším vyzužitím rolling hashe je Robinův-Karpův algoritmus pro vyhledávání řetězců, který se běžně využívá v detekci plagiátů, například v systém YAP3.

3.3 Vyhledávání řetězců

Na vyhledávání řetězců existuje mnoho různých algoritmů a postupů. Ty se liší v rychlosti, paměťové náročnosti a také v množině řetězců, které chceme vyhledávat. Některé algoritmy jsou určeny pro vyhledávání jednoho řetězce, jiné pro konečnou množinu řetězců a jiné pro nekonečnou množinu řetězců (regulární výrazy). Největším problémem vyhledávání řetězců je velká výpočetní složitost, proto se využívá buďto u malých množin dokumentů, nebo v kombinaci s jinými metodami.

3.3.1 Algoritmy pro vyhledávání vzorku v textu

Existuje mnoho algoritmů, které slouží k vyhledávání podřetězců odpovídajících zadanému vzorku (patternu) v textu. Při vyhledávání jednoho konkrétního vzorku probíhá porovnávání pomocí posuvného okénka o délce vzorku. Nejjednodušší z nich je naivní přístup, kdy se okénko posouvá vždy s jednotkovým krokem a porovnán je vždy celý vzorek. Tento postup, ač je implementačně nejjednodušší, je velmi neefektivní. Proto byla vyvinuta řada algoritmů, které tento postup vylepšují. Jedním z možných vylepšení je využití hashe (například Rabinův-Karpův algoritmus), nebo efektivnější zvolení posunu okénka (například Knuttův-Morrisův-Prattův algoritmus).

3.3.1.1 Rabinův-Karpův algoritmus

Rabinův-Karpův algoritmus je jedním z nejpoužívanějších algoritmů k vyhledávání výskytů daného řetězce v textu. Oproti naivnímu algoritmu je jeho výpočetní čas značně snížen díky využití rolling hashe. Prakticky je možné vyhledávat text s lineární složitostí, v nejhorším případě (čím více výskytů je v textu, tím je pomalejší) je časová složitost $\Theta(mn)$, kde m je délka vyhledávaného řetězce a n je délka textu [8]. Úspora spočívá v tom, že porovnáván není text, ale číselná hodnota a až v případě shody hashe dojde na porovnávání řetězců. Rabinův-Karpův algoritmus je vhodný i pro vyhledávání více než jednoho vzorku, vzorky však musí mít všechny stejnou, pevně danou délku. Nelze ho tedy použít k vyhledání všech podřetězců.

3.3.1.2 Knuttův-Morrisův-Prattův algoritmus

Knuttův-Morrisův-Prattův je ukázkou algoritmu pro vyhledávání textu, kde úspora výpočetního času nespočívá v hashování, ale v tom, že si algoritmus pamatuje částečné shody a vyhledávací okénko se posune rovnou na místo příštího možného výskytu, na rozdíl od naivního přístupu, u kterého se okénko posouvá s jednotkovým krokem [10]. Přeskakování políček, kde je výskyt vyloučen, je realizován pomocí tabulky částečných shod [17]. Ta obsahuje pro každou pozici ve vzorku délku nejdelšího prefixu, který je zároveň suffixem podřetězce vzorku,

a	b	c	a	b	b	c	a	b	c
0	0	0	1	2	0	0	1	2	3

Tabulka 3.1: Knuttův-Morrisův-Prattův algoritmus – Tabulka částečných shod pro vzorek *abcabbcabc*

a	b	c	d	e
3	2	1	10	10

Tabulka 3.2: Boyerův-Mooreův algoritmus – Tabulka pro pravidlo špatného znaku pro vzorek *abcabbcabc* v textu *ababcdeadebcededacba*

který začíná na pozici 0 a končí na dané pozici (tedy prefixu vzorku končícím na dané pozici). V případě neshody ve vzorku můžeme tuto délku použít jako počet znaků, o kolik může okénko skočit. Časová složitost algoritmu je $O(m+n)$.

3.3.1.3 Boyerův-Mooreův algoritmus

Boyerův-Mooreův algoritmus je dalším příkladem algoritmu, který využívá informace o dosavadním prohledávání k efektivnějším skokům porovnávacího okénka. Na rozdíl od Knuttova-Morrisova-Prattova algoritmu je vzorek porovnáván s textem pozpátku. Porovnávání začíná na konci vzorku a na pozici k v textu, kde k je délka vzorku. Při neshodě vzorku s textem je velikost posunu okénka dána dvěma pravidly [9]:

- pravidlo špatného znaku – The Bad Character Rule
- pravidlo dobrého suffixu – The Good Suffix Rule

Pro každé z těchto pravidel je předzpracovaná tabulka. Tabulka pro pravidlo špatného znaku obsahuje jednu položku pro každý znak použitý v textu. K tomuto znaku je přiřazena číselná hodnota vyjadřující délku skoku. V případě neshody znaku mezi vzorkem a textem, je na základě znaku přečteného v textu v tabulce dohledána délka skoku. Pro znaky, které se ve vzorku nevyskytují, je délka skoku rovna délce vzorku, v opačné případě je délka skoku rovna $k-j$, kde k je délka vzorku a j je index posledního výskytu daného znaku ve vzorku.

Algoritmus může v určitých případech běžet i rychleji než s lineární složitostí, jindy se však může rychlostí blížit naivnímu řešení. Existuje zjednodušená varianta, známá jako Boyer–Moore–Horsepool algorithm. V té se počítá pouze s prvním pravidlem.

3.3.2 Konečné automaty

Konečný automat je matematický model využívaný zejména k vyhledávání a rozpoznávání textů. Je definován jako pětice $(Q, \sum, \delta, q_0, F)$, kterou tvoří:

- Q – konečná množina stavů

- $\sum$ – vstupní abeceda
- δ – přechodová funkce
- q_0 – počáteční stav
- F – množina koncových stavů

Jako vstup automatu slouží posloupnost vstupních symbolů. Automat začíná v počátečním stavu a na základě vstupních symbolů a přechodové funkce přechází do následujících stavů. Pokud se automat nachází v koncovém stavu, znamená to, že vstupní text je automatem přijat, pokud v určitém stavu není pro daný vstupní symbol definována přechodová funkce, vstupní text je automatem odmítnut. Podle přijetí či odmítnutí vstupního řetězce zjistíme, jestli je řetězec slovem jazyka rozhodovaného příslušným konečným automatem.

Konečné automaty se dělí na deterministické (DFA - Deterministic finite automaton) a nedeterministické (NFA - Non-deterministic finite automaton). Rozdíl mezi těmito dvěma typy je v přechodové funkci. Zatímco u deterministického automatu je v každém stavu definován pro každý vstupní znak nejvýše jeden následující stav, u nedeteministického může být následujících stavů více. Pro každý nedeterministický automat existuje ekvivalentní deterministický automat, na převod mezi těmito dvěma typy se používá algoritmus založený na sjednocování množin stavů.

3.3.3 Suffixové stromy

Suffixový strom je datová struktura sloužící k vyhledávání v textu. Velkou předností suffixových stromů je, že konstrukce může probíhat v lineárním čase a s lineární paměťovou složitostí. Hrany suffixového stromu tvoří buďto podřetězce textu, pro který je konstruován, nebo jednotlivé znaky. První varianta je paměťově úspornější, je však implementačně složitější. Listy stromu tvoří suffixy vstupního řetězce, uzly na cestě z kořenu k listům představují prefixy suffixů, tedy jednodušeji řečeno podřetězce vstupního řetězce. Suffixový strom může sloužit kromě vyhledávání také například k počítání výskytů podřetězce nebo nalezení nejdelšího opakujícího se podřetězce [13].

3.4 Základní dělení systémů na detekci plagiátů

Nejdůležitější kritérium dělení systémů na detekci plagiátů je přítomnost množiny dokumentů k porovnání, zvané korpus. Systémy a nástroje, které při detekci plagiátů pracují pouze v rámci daného souboru, nazýváme intrinsické a ty ostaní nazýváme extrinsické. Extrinsické nástroje dále dělíme do dvou podskupin: intrakorpální a extrakorpální. Intrakorpální systémy mají vlastní množinu dokumentů, zatímco extrakorpální systémy využívají nějaký externí zdroj, jako je třeba internet.

3.4.1 Intrinsické nástroje

Intrinsické nástroje na detekci plagiátů se vyznačují tím, že nemají k dispozici žádné dokumenty, které by s potenciálním plagiátem porovnaly. Jejich úkolem je tedy rozhodnout, jedná-li se o plagiát, jen na základě samotného dokumentu. Intrinsické nástroje se snaží hledat nekonzistence ve stylu psaní, jinými slovy detekovat, jestli je dokument psán jedním člověkem, nebo jestli mají různé pasáže textu různé autory. Typický intrinsický nástroj na detekci plagiátů je tedy založen na stylometrické analýze dokumentu.

3.4.2 Extrakorpální nástroje

Extrakorpální nástroje porovnávají odevzdaný dokument s nějakým externím zdrojem. Jednou z možností extrakorpální detekce je vyhledávání dat na internetu ve snaze najít zdroj, ze kterého byl text dokumentu okopírován. Extrakorpálním zdrojem ale může být také třeba databáze.

3.4.3 Intrakorpální nástroje

Intrakorpální detekce plagiátů se omezuje na porovnávání dokumentů v rámci korpusu. Porovnávání může probíhat v párech - porovnávání dvojic dokumentů každý s každým, nebo lze využít statistické operace nad celým korpusem. V případě velkého korpusu může být porovnávání dokumentů zdlouhavé, proto můžeme využít metadat k úspoře výpočetního času. Metadata jsou doplňující předzpracovaná data, která nenesou žádnou novou informaci, slouží však k zefektivnění detekce. Jedním z nejjednodušších příkladů metadat je rozdělení souborů na staré a nové - tím pádem nebude třeba vždy porovnávat všechny dokumenty, pouze nové se starými, případně nové navzájem. Příkladem intrakorpálního nástroje na detekci plagiátů je i aplikace, jejíž vývoj je předmětem této práce.

3.5 Detekce plagiátů ve volném textu

Znaků, podle kterých můžeme dokument označit za plagiát, je celá řada. Mezi nejjednoznačnější znaky plagiátu patří shodné pasáže. Může to být ale také podobný styl psaní, typická větná skladba nebo slovní zásoba. Dalším ukazatelem může být nekonzistence textu. Pokud se v dokumentu mění styl psaní, může to vypovídat o tom, že dokument se skládá z prací více než jednoho autora. Kromě samotného obsahu dokumentu je také možné detekovat okopírované pasáže podle formátování textu.

3.5.1 Lematizace

Lematizace je převedení slov do základního tvaru, tedy například 1. pádu jednotného čísla u podstatných jmen, nebo infinitiv u sloves. Lematizace se používá pouze u volného textu, při zpracování zdrojového kódu nemá uplatnění.

3.5.2 Bag of words

Tato metoda spočívá ve vytvoření seznamu jedinečných slov nacházejících se v textu a přiřazování počtu výskytů ke každému slovu. Text je tedy reprezentován vektorem přirozených čísel, k detekci plagiátů poslouží reprezentace dokumentů ve vektorovém prostoru. Ve vektoru mohou být obsaženy buďto výskyty všech slov v textu, nebo může být pomocí speciální strategie vybrán určitý počet slov, která jsou pro text nejvíce charakteristická. Typicky jsou odstraněna slova jako předložky a spojky, která se vyskytují prakticky v každém textu a nenesou důležitou sémantickou informaci. Při porovnávání mohou mít různé prvky vektoru přiřazenu různou váhu.

3.5.3 Fingerprinting

Fingerprinting je metoda, která využívá pro porovnávání dokumentů jejich otisk zvaný fingerprint. Tento otisk je tvořen kolekcí čísel, která vzniká zpracováním dokumentu tak, aby fingerprint co nejvíce charakterizoval obsah dokumentu a zároveň umožňoval efektivní a přesné porovnávání. Otisky dokumentů bývají uchovávány v metadatech, což je úspora výpočetního času, neboť každý dokument stačí zpracovat jen jednou a pak už jen porovnávat existující fingerprint.

Nejtypičtější příklad fingerprintu je množina podřetězců zvaných N-gramy. N-gram je souvislá posloupnost n tokenů (slov). Text je reprezentován jako n-tice slov jdoucích za sebou, typické jsou například trojice slov - trigramy. Číslo n se nazývá granularita a na jeho správné volbě závisí přesnost detekce. Pokud je granularita příliš malá, bude docházet k falešným poplachům, tedy označení za plagiát u dokumentu, jehož podobnost se vzorem je pouze náhodná. Pokud zvolíme granularitu příliš velkou, kvalita detekce bude příliš citlivá na drobné úpravy k zamaskování plagiátu a pravděpodobnost odhalení se sníží. Podobnost dvou n-gramů je následně možné porovnávat podle celé řady metrik. Jednou z nejjednodušších patří poměr shodných slov k celkovému počtu různých slov v obou n-gramech.

$$r(A,B) = \left| \frac{S(A) \cap S(B)}{S(A) \cup S(B)} \right| \tag{3.1}$$

Další často používanou metrikou je kosinová podobnost. Tu můžeme určit tak, že vytvoříme pro každý dokument vektor počtu výskytů n-gramů v textu a podobnost mezi dokumentu vyjádřit pomocí kosinu úhlem mezi dvěma vektory. Kosinová podobnost se používá například u systému CHECK.

3.5.4 Stylometrická analýza

Stylometrická analýza má za úkol určit autora na základě specifického stylu psaní. Ten může být rozpoznán například na základě typické větné skladby nebo použitého slovníku. Rozpoznání stylu psaní je tím přesnější, čím větší je objem textu, pomocí kterého styl autora definujeme. Často se používá celý korpus dokumentů od jednoho autora, který slouží jako trénovací množina dokumentů. Pro identifikaci autora se používá fingerprint, tedy zjednodušená číslená reprezentace. Fingerprinting již byl v této práci zmíněn v souvislosti s porovnáváním

dokumentů, v tomto případě se však otisk nepoužívá k identifikaci dokumentu, ale autora. Takovýto otisk se nazývá writeprint [1].

3.5.5 Existující nástroje

Zbytek této podkapitoly je věnován konkrétním příkladům detekčních systémů. Nejedná se o vyčerpávající studii existujících řešení, cílem je ukázat využití postupů popsaných v této kapitole v praxi. Popsány jsou dva příklady existujících nástrojů, COPS a CHECK, oba zpracovávající dokumenty ve formátu TeX (LaTeX v případu systému CHECK). První je uveden jako příklad využití hashování pro porovnání obsahu dokumentů, druhý je ukázka využití syntaxe LaTeXu pro efektivnější detekci, ačkoli struktura dokumentu má jen pomocnou funkci - hlavní část detekce je založená na porovnávání obsahu. V obou případech se nejedná o komerční nástroje, ale experimentální systémy vyvinuté v rámci univerzitního výzukumu.

3.5.5.1 COPS

Systém COPS (COpy Protection System) zpracovává dokumenty ve formátu TeX, DVI a troff. Prvním krokem je převod dokumentu do kanonické formy, což je prostý text [5]. COPS nevyužívá při detekci plagiátu strukturu dokumentu, informace o struktuře jsou z dokumentu odstraněny a dále je zpracováván jako řetězec ASCII znaků. Text je rozdělen na menší útvary zvané chunks. Chunk má proměnlivou délku a je definován dvojicí obsah-místo výskytu. Tyto chunky jsou uchovány v hashovací tabulce, ke každému chunku je tedy vypočítán hash. V této tabulce jsou následně dohledány shody mezi dokumenty a pokud procento shod překročí stanovenou hranici, je dokument označen jako plagiát.

3.5.5.2 CHECK

Systém na detekci plagiátů CHECK se specializuje na dokumenty psané v LaTeXu [6]. Jeho zvláštností je, že k porovnání dvou dokumentů nevyužívá pouze obsah, ale také formát dokumentu, zejména členění do kapitol a podkapitol, velká část syntaxe LaTeXu je však zanedbána. Systém vytvoří pro každý soubor strukturu nazvanou Structural characteristic (SC), což je strom, jehož uzly představují kapitoly, podkapitoly a odstavce, která slouží k zefektivnění detekce. Samotné porovnání je podobné metodě Bag of Words. Pro každý uzel je vybrána množina klíčových slov, která mají přiřazenou váhu a jsou lematizována. Tato množina se vybírá tak, aby vybraná slova byla pro text co nejvíce charakteristická. Slova označená v syntaxi LaTeXu tučně či kurzívou mají přiřazenou vyšší váhu než ostatní slova. Pro kořen grafu je množina slov vybrána z celého dokumentu, pro ostatní uzly pouze z příslušného textového celku. Při porovnání je z dvou množin slov vybrán jejich průnik a výskyty v porovnávaných textech reprezentují body v prostoru vektorů. Pomocí jejich kosinové podobnosti se určí metrika podobnost v intervalu 0 až 1, kde 1 je největší podobnost. Kosinová podobnost používá jako metriku podobnosti kosinus úhlu, který dva vektory svírají. Tato metrika je aplikována na uzly rekurzivně od kořenu směrem k listům grafu. Pokud je podobnost uzlů v obou dokumentech větší než předem daná hranice, postupuje se k porovnání potomků uzlu, v opačném případě jsou texty označeny za rozdílné a další zkoumání je tedy zbytečné. To je značná úspora výpočetního času. Pokud metrika podobnosti přesáhne hranici

i pro listy grafu, jsou dané kusy textu podrobeny porovnání řetězců, které je časově náročné, ale díky filtraci pomocí rekurzivního porovnávání SC je jeho časová složitost přijatelná.

3.6 Detekce plagiátů ve zdrojovém kódu

Detekce plagiátů v zdrojovém kódu často využívá metody, které byly původně určeny k refaktorování a analýze kódu. Jedním z typických rysů zdrojového kódu je přesně vymezená syntaxe. To může být výhodou i nevýhodou. Nevýhodou je například to, že je zpravidla potřeba použít nástroj na detekci plagiátů speciálně navržený pro daný jazyk. Znalost syntaxe kódu nám však také poskytuje spoustu možností, jak efektivněji detekovat plagiáty. Detekce plagiátů ve zdrojovém kódu je značně ztížena tím, že plagiátoři mají velký výběr metod, jak plagiát zamaskovat. Zatímco u volného textu jsou možnosti maskování omezené na jednoduché úpravy, jako je vkládání nebo mazání slov a přeskupování textu, plagiátor zdrojového kódu má k dispozici značné množství postupů, kterými může refaktorovat kód tak, aby se zdál na první pohled zcela odlišný. Mezi tyto postupy patří:

- přejmenování identifikátorů
- změny datových typů
- přidávání redundantního kódu
- změny struktury kódu - třeba přemísťování bloků bez změny funkce
- změny komentářů, případně jejich přidávání a mazání

Plagiátor může také ekvivalentně nahrazovat řídící struktury, například u jazyku C a jemu podobných nahradit switch-case syntaxi ekvivalentním zápisem pomocí if bloků. Kód také nemusí být okopírován kompletně, mohou být použité pouze určité pasáže zkombinované s původním kódem. Při takovýchto úpravách je už však sporné, jedná-li se stále o plagiát.

3.6.1 Softwarové metriky

Softwarová metrika je jakákoli měřitelná hodnota vyjadřující vlastnost software. Typická softwarová metrika je například počet řádek kódu. Metriky se používají při analýze kódu, měření kvality, plánování a k jiným účelům a můžou sloužit také k odhalování podobností v kódu.

3.6.2 Počítání atributů

V začátcích detekce podobností v programovém kódu byl jako reprezentace dokumentu určená k porovnání využit vektor o pevně dané velikosti, kde každý prvek představoval jednu softwarovou metriku. Jako historicky první měřítko porovnání dvou kusů kódu byl využit Halsteadův profil, což je čtveřice čísel: celkový počet operandů, celkový počet operátorů, počet unikátních operandů a počet unikátních operátorů.

V následujících letech probíhaly pokusy o využití vhodnějších metrik, jako počet řádek v kódu, počet klíčových slov či počet použitých proměnných. Problémem těchto metrik však zůstává, že ve velkém počtu případů je podobnost dvou profilů čistě náhodná.

3.6.3 Structure profile

Pokročilejší metody detekce softwarových plagiátů využívají strukturu programu k vytvoření výstižnější reprezentace programového kódu. Tato reprezentace se nazývá structure profile a vychází ze structure listu, tedy seznamu řídících struktur. Struckture list je podroben různým transformacím tak, aby dvě na první pohled jiné verze stejného kódu měly stejný nebo velmi podobný profil.

3.6.4 Existující nástroje

3.6.4.1 Plague

Structure profiles jsou využity u systému na detekci plagiátů Plague. Porovnávání samotných profilů však není dostačující metoda na spolehlivé určení plagiátů, slouží pouze jako předstupeň k samotnému porovnávání. Úkolem porovnání structure profiles v systému Plague je vyfiltrovat úzký okruh párů dokumentů, u kterých je podezření, že by se mohlo jednat o plagiáty. Konečné srovnání dokumentů je založeno na porovnávání řetězců. Tato metoda se vyznačuje značnou časovou složitostí, proto je použita v kombinaci s porovnáváním structure profiles, aby bylo dosaženo jak přijatelné časové složitosti, tak přesného porovnání dokumentů.

3.6.4.2 YAP3

Systém na detekci YAP (Yet Another Plague) vychází ze systému Plague. Existuje ve třech verzích, ta nejnovější se jmenuje YAP3. Detekce probíhá ve fázích, nejprve je vstupní text tokenizován. Při tokenizaci prochází text následujícími úpravami:

- odstranění komentářů a textových konstant
- převedení všech velkých písmen (upper case) na malá (lower case)
- převede synonyma do běžné formy
- provede expanzi funkcí v pořadí volání
- odstraní tokeny, které nejsou v lexikonu jazyka

Takto zpracovaný text následně porovnává Rabinovým-Karpovým algoritmem na vyhledávání řetězců.

Kapitola 4

Návrh řešení

Tato kapitola se soustředí na volbu technologií a algoritmů řešení. Součástí kapitoly je zhodnocení přínosu existujících řešení z předchozí kapitoly pro účel naší aplikace a zdůvodnění nutnosti vytvořit novou aplikaci místo využití některého z existujících nástrojů. Volba technologií zahrnuje technické detaily, jako je například volba programovacího jazyka. Dále se kapitola zabývá volbou reprezentace dokumentu. vhodné metriky porovnání, algoritmů a datových struktur tak, aby byl výsledek co nejpřesnější a časová i paměťová složitost přijatelná.

4.1 Volba reprezentace dokumentu

Při detekci plagiátů nebývají zpravidla porovnávány nezpracované soubory v jejich hrubé podobě, ale jejich vhodné reprezentace. Jedním z hlavních cílů analýzy řešení je zvolit vhodnou formu reprezentace souboru tak, aby důležité informace zůstaly zachovány a porovnávání těchto reprezentací bylo co nejjednodušší, nejpřesnější a nejefektivnější. Pro volbu vhodné reprezentace dokumentu si je tedy nejprve potřeba ujasnit, které informace jsou důležité a které ne, Vzhledem k tomu, že aplikace bude detekovat plagiáty výhradně v LaTeXové syntaxi a obsah ignorovat, bude výhodné při zpracování dokumentu syntaxi a obsah oddělit. Vhodný postup je tokenizace, tedy rozdělení na základní lexikální elementy, zvané tokeny. Výstupem je posloupnost těchto tokenů, přičemž token je charakterizován typem a klíčovým slovem. Klíčových slov je v LaTeXu omezené množství, můžeme tedy dosáhnout úspory tím, že každému klíčovému slovu přiřadíme číselnou reprezentaci a dále pracujeme s dokumentem zpracovaným do podoby číselné posloupnosti. Poté, co jsou vstupní dokumenty předzpracovány do číselné podoby, je potřeba zvolit vhodnou metodu porovnání. Zbytek podkapitoly bude věnován rozboru uplatnitelnosti různých způsobů porovnávání dokumentů běžně využívaných v detekci plagiátů.

4.1.1 Structure profile

Structure profile se využívá při detekci plagiátů ve zdrojovém kódu. Správně zvolený structure profile by měl rozeznat kód i po povrchních změnách, jako jsou například přesuny bloků kódu nebo přejmenování identifikátorů. Structure profile pomocí zjednodušené číselné reprezentace vyjadřuje strukturu větvení kódu, zejména pomocí podmíněných příkazů a cyklů. Tyto řídící struktury se v LaTeXu nevyskytují, structure profile proto v našem případě nemá využití.

4.1.2 Syntaktický strom

Zatímco structure profile nemá pro porovnávání dokumentů ve formátu LaTeX uplatnění, porovnávání syntaktického stromu je o něco vhodnější. Narážíme zde však na několik problémů. Jedním z nich je obtížná porovnatelnost stromů. Další problém, kvůli kterému bylo nakonec od využití syntaktického stromu pro porovnání dokumentů upuštěno, je poměrně malá vypovídací hodnota syntaxe LaTeXu. Pokud totiž nemáme k dispozici interpret LaTeXu, nedokážeme v mnoha případech jednoznačně určit, jak se bude sytaxe větvit. Například pokud neznáme definice všech příkazů, nedokážeme určit, co je parametr příkazu a co ne. Počet parametrů se ze samotného kódu nedá určit, tím pádem bychom mohli kód jednoznačně větvit jen podle begin-end bloků, což je nedostačující.

4.1.3 Document Object Model

Další možná reprezentace dokumentu je Document Object Model, známý pod zkratkou DOM. Známé je využití DOM u webových stránek, lze s ním však pracovat i u LaTeXových dokumentů, příkladem je framework PlasTeX. Jde o hierarchicky organizovanou strukturu objektů k vykreslení. Problémem využití DOM pro porovnávání dokumentů je, že můžeme mít stejný DOM pro více dokumentů s různou syntaxí. Protože v případě naší aplikace je důležitější syntaxe než obsah, je tato reprezentace dokumentu nevhodná.

4.1.4 Fingerprinting

Dokument lze reprezentovat jako vektory o pevně dané délce n. S takovými vektory můžeme nakládat jako s body v n-rozměrném prostoru. Metoda, kdy je dokumentu přidělen otisk v podobě číselné reprezentace o pevně dané délce, se nazývá fingerprinting a je popsána v předchozí kapitole. Existuje celá řada typů vzdáleností mezi vektory. Kromě vzdálenosti mezi dvěma body můžeme také využít statistické výpočty nad množinou bodů. To je výhodné, neboť cílem aplikace není jen najít podobnosti mezi dokumenty, ale najít části kódu, které se jsou společné pro malý počet dokumentů a tím nejvíce podezřelé. Pro přesnost detekce je naprosto zásadní vhodná volba otisku (fingerprintu).

4.2 Volba fingerprintu

Fingerprinting je metoda, jež má využití především při detekci plagiátů ve volném textu. Při vytváření otisku souboru dochází k značné redukci objemu informací, je tedy třeba dát si velký pozor, abychom nepřicházeli o důležitá data. Dalším problematickým aspektem vytváření otisku dokumentu ve formátu LaTeX je, že mezi dokumenty může existovat velká podobnost, aniž by se jednalo o plagiáty.

4.2.1 Bag of Words

Vyjádření reprezentace dokumentu pomocí metody bag of words se využívá při detekci plagiátů ve volném textu, při detekci ve zdrojovém kódu není příliš vhodná. U volného textu má

totiž autor k dispozici velkou slovní zásobu, zatímco syntaxe zdrojových kódů je pevně vymezena. U Bag of Words také není zohledněno pořadí lexikálních elementů. Pokud by byla použita metoda Bag of Words pro detekci plagiátů v LaTeXové syntaxi, velmi pravděpodobně by docházelo k velkému množství falešných shod, protože počty výskytů lexikálních elementů by se s velkou pravděpodobností poměrně dosti shodovaly i u dokumentů, které nejsou plagiáty.

4.2.2 N-gramy

Dokument může být vyjádřen jako množina n-gramů, tato reprezentace se často využívá při detekci plagiátů ve volném textu. Fingerprint bývá často tvořen kolekcí hashů n-gramů vybraných pomocí určité strategie. Problémem u tohoto přístupu je velká ztráta dat. Pokud chceme výrazně redukovat velikost fingerprintu, snižujeme tím zároveň detekční schopnosti aplikace. Jinak řečeno, může se stát, že aplikace podezřelou pasáž přehlédne. Další nevýhodou je pevně daná délka n-gramu. Vzhledem k tomu, že chceme nalézt mezi dokumenty co nejunikátnější společné pasáže, může volba n-gramů jako fingerprintu snižovat rozeznávací schopnosti. Pokud je granularita příliš malá, zaniknou unikátní podřetězce v šumu, pokud je příliš velká, může dojít k tomu, že je aplikace přehlédne. Tato metoda je tím pádem vhodná pro srovnání procentuální shodnosti obsahu dvou dokumentů, pro detekci podezřelých podřetězců v LaTeXovém kódu však není dost citlivá.

4.2.3 Vektor výskytů podřetězců

Další možností je vektor počtu výskytů podřetězců v dokumentu. Fingerprint je zde tvořen vektorem, kde každý prvek vektoru vyjadřuje počet výskytů určitého podřetězce. Zde je problém, že podřetězců je ohromné množství a může růst velmi prudce v závislosti na velikosti korpusu. Je tedy třeba vytvořit vhodnou strategii výběru co nejmenšího počtu podřetězců, které mají pro detekci plagiátů největší význam. Tuto strategii nazveme filtr a v textu se budeme dále zabývat vhodnou volbou filtru tak, aby byl vektor co nejmenší a zároveň docházelo k co nejmenší ztrátě důležitých dat. Tato reprezentace dokumentu je sice paměťově náročná, na rozdíl od ostatních však nezpůsobuje tak drastickou ztrátu dat, je proto pro použití na detekci plagiátů nejvhodnější.

4.3 Návrh rozdělění aplikační logiky

V předchozí kapitole jsme si představili různé přístupy k zpracování dokumentu do vhodné formy. V aplikaci je toto jen první krok na cestě od nezpracovaných dokumentů až k výsledné metrice. Zpracování dokumentu můžeme rozdělit na několik fází. Nejprve je třeba zpracovat dokument z původního textu do formy, se kterou jde dále snáz pracovat, první krok tedy bude lexikální analýza. Lexikální analýza rozpozná syntaxi dokumentu a zpracuje dokument na posloupnost lexikálních elementů. Závěrem minulé podkapitoly bylo, že nejvhodnější reprezentací dokumentu je vektor počtů výskytů podřetězců, další fází tedy bude algoritmus, který tento vektor sestaví. Také bylo zmíněno, že tento vektor bude mít značnou velikost a bude jej třeba zredukovat. To se provede v další fázi, filtru podřetězců. Nyní zbývá nad

množinou těchto vektorů vypočítat metriku podobnosti (respektive rozdílnosti) pro každou dvojici vektorů. Pokud to tedy shrneme, aplikace má 4 na sebe navazjící fáze:

- Lexikální analyzátor
- Algoritmus počítající počty výskytů podřetězců
- Filtr
- Výsledná metrika

U každé fáze je třeba dbát na to, aby neměla příliš velkou paměťovou či výpočetní složitost. Tak jako řetěz je tak silný, jako jeho nejslabší článek, je i u naší aplikace jedna fáze zpracování příliš výpočetně náročná aplikace poběží neúnosně dlouho, i kdyby ostatní fáze byly rychlé a efektivní. Nejproblematičtější je v tomto ohledu pravděpodobně algoritmus na počítání výskytů podřetězců, je však také třeba nepodcenit metriku porovnávání.

4.4 Volba algoritmu pro počítání výskytů podřetězců

Existuje mnoho algoritmů, pomocí kterých je možné počítat výskyty podřetězců v textu. Jejich společným problémem je výpočetní složitost. Značná část těchto algoritmů však pracuje na principu vyhledávání vzorku v textu (Rabinův-Karpův algoritmus, Knuttův-Morrisův-Prattův algoritmus a další). Úkolem algoritmu je však vyhledávat všechny možné podřetězce, počet možností se tím tedy značně zužuje. V úvahu připadají dvě varianty:

1. konstrukce konečného faktorového automatu
2. konstrukce suffixového stromu

První varianta je implementačně výrazně jednodušší, je však zároveň pomalejší. Konstrukce suffixového stromu pomocí Ukkonenova algoritmu vyniká lineární paměťovou i výpočetní složitostí, implementace je však velmi pracná. Původní plán byl realizovat počítání výskytů podřetězců pomocí konstrukce konečného faktorový automatu, nakonec však došlo na implementaci obou variant.

4.4.1 Konečný faktorový automat

Všechny podřetězce několika daných řetězců je možné vyjádřit jako konečný faktorový automat. Nejjednodušší na sestavení je nedeterministický faktorový automat s ε-přechody, zkráceně označovaný ε-NFA. ε-přechody jsou přechody, pomocí kterých se můžeme dostat do následujícího stavu bez vstupního znaku. Množina stavů, do kterých se dostaneme pouze pomocí ε-přechodů, se nazývá ε-closure. Sestavení faktorového ε-NFA pro množinu řetězců provedeme následovně [7]: Nejprve vytvoříme počáteční stav. Dále čteme každý vstupní řetězec od začátku ke konci a pro každý znak vytvoříme stav označený číslem souboru a pozicí znaku. Přidáme přechod označený přečteným znakem, který směřuje z naposled přidaného stavu do nového stavu. Pokud je přečten první znak řetězce, přechod vychází z počátečního stavu. Přidáme ε-přechody z počátečního stavu do všech ostatních stavů s výjimkou stavů představujících poslední znak řetězce. Na závěr označíme všechny stavy kromě počátečního jako konečné.

Pro implementaci je lepší, pokud je automat bez ε-přechodů. Na to existuje jednoduchý algoritmus [14]. Ten funguje ve třech krocích:

1. Stav je označen jako konečný, pokud se v jeho ε-closure nachází nějaký konečný stav
2. Pokud existuje ze stavu p do stavu q přechod označený určitým znakem, je přidán přechod označený tímto znakem také ze stavu p do všech stavů v ε-closure stavu q
3. Všechny ε-přechody jsou smazány

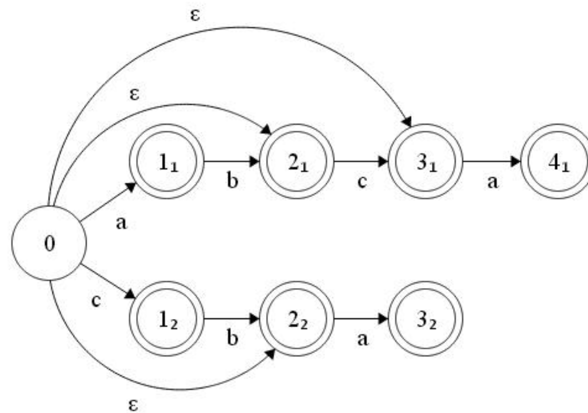
V případě naší aplikace není potřeba konstruovat ε-NFA, protože už samotná konstrukce NFA bez ε-přechodů je poměrně jednoduchá a lze ji provést v lineárním čase následujícím postupem, velmi podobným postupu při vytváření ε-NFA:

Nejprve vytvoříme počáteční stav. Dále čteme každý vstupní řetězec od začátku ke konci a pro každý znak vytvoříme stav označený číslem souboru a pozicí znaku. Přidáme přechod označený přečteným znakem, který směřuje z posledně přidaného stavu do nového stavu. Pokud je přečten první znak řetězce, přechod vychází z počátečního stavu. Přidáme přechod označený přečtenám znakem, které směřuje z počátečního do nového stavu (tento krok je jediné, v čem se algoritmus liší od konstrukce ε-NFA). Na závěr označíme všechny stavy kromě počátečního jako konečné.

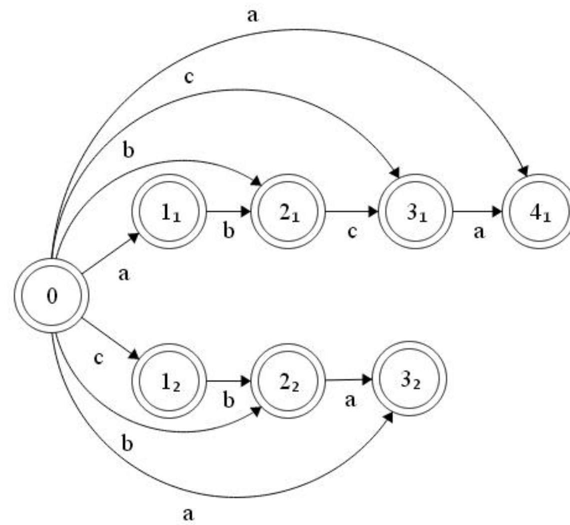
Poslední fáze je algoritmus na převod nedeterministického konečného automatu na deterministický. Tento algoritmus je založený na spojování množin stavů NFA [16]. Každý stav DFA vznikne sloučením množiny stavů NFA. Začneme v počátečním stavu, pro každý vstupní znak máme množinu následujících stavů. Tyto stavy spojíme do stavů DFA a přidáme do grafu DFA. Tyto stavy zároveň přidáme do fronty otevřených stavů, ze kterých postupně odebíráme. Do této fronty je stav přidán pouze v případě, že se už v grafu nevyskytuje. Z fronty otevřených stavů postupně odebíráme a provádíme stejný postup jako u počátečního stavu - pro každou podmnožinu stavů NFA získáme množinu následujícch stavů pro každý vstupní symbol a tu spojíme do stavu DFA. V případě, že stav DFA pro danou podmnožinu již v grafu DFA existuje, přidáme pouze přechod pro příslušný vstupní znak.

4.4.2 Suffix tree a suffix trie

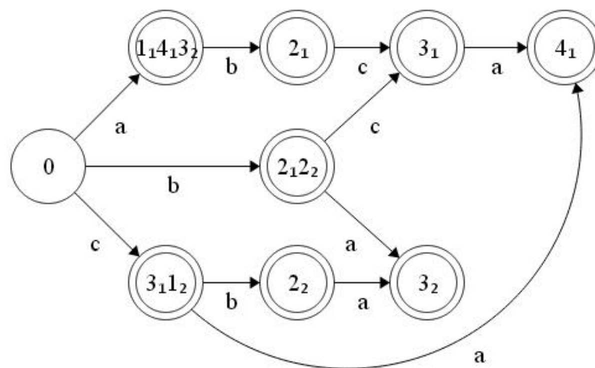
Suffixový strom vychází z myšlenky, že každý podřetězec je příponou (suffixem) předpony (prefixu) původního textu [12]. Jeho hlavní výhodou je rychlost. Suffixový strom je možné zkonstruovat v lineárním čase a je pomocí něj možné v lineárním čase zjistit, nachází-li se daný řetězec v textu, pro který byl suffixový strom zkonstruován, je-li daný řetězec jeho suffixem a také kolikrát se v textu vyskytuje. Je také možné použít jednodušší, ale méně efektivní variantu suffixového stromu zvanou Suffix trie.



Obrázek 4.1: Faktorový NFA s epsilon přechody pro dvojici řetězců *abca* a *cba*



Obrázek 4.2: Faktorový NFA bez epsilon přechodů pro dvojici řetězců *abca* a *cba*



Obrázek 4.3: Faktorový DFA pro dvojici řetězců *abca* a *cba*

4.4.2.1 Suffix tree

Konstrukce suffixového stromu je implementačně náročná, ale velmi efektivní metoda vyhledávání řetězců v textu. Suffixový strom je datová struktura tvořená uzly a hranami, představující množinu suffixů vstupního textu. Uzly představují jednotlivé podřetězce, přičemž listy stromu představují suffixy. Počet listů tím pádem odpovídá délce vstupního testu. Vnitřní uzly mají vždy alespoň 2 potomky[13], počet uzlů je maximálně $2n$, kde n je délka textu. Pro vnitřní uzel vždy platí, že počet výskytů podřetězce, jenž uzel představuje, se rovná počtu listů podstromu, jehož je daný uzel kořenem, což umožňuje jednoduše ze suffixového stromu vyčíst počet výskytů všech podřetězců daného textu. Hrany stromu představují podřetězce textu a jsou definovány pomocí dvou hodnot: pozice počátečního znaku a pozice konečného znaku. Díky tomu, že jedna hrana zabírá konstantně velké místo v paměti a počet hran je o 1 menší než počet uzlů, je paměťová složitost suffixového stromu lineární. Pro každý uzel platí, že z něj nemůže vycházet více hran začínajících stejným znakem.

Ke konstrukci suffixového stromu se používá Ukkonenův algoritmus[11]. Ukkonenův algoritmus má lineární časovou i paměťovou složitost, čímž se řadí mezi nejlepší algoritmy podobného typu. Suffixový strom je vytvářen inkrementálně při procházení textu po jednotlivých znacích od začátku ke konci. Při procházení jsou do stromu přidávány suffixy počínající přečteným znakem a končící koncem textu. Konec textu se často označuje jako ukončovací znak - $. V případě, že ve stromě neexistuje hrana vedoucí z kořene začínající přečteným znakem, je tento suffix pouze přidán jako nový list, do kterého vede nově vytvořená hrana z kořene. Pokud však už ve stromu hrana z kořene se stejným počátečním znakem existuje (což znamená, že už ve stromu existuje suffix, který má s novým suffixem společný prefix), nový suffix se nepřidá a začíná procházení stromu. Algoritmus si pamatuje poslední navštívený uzel (aktivní uzel), hranu, na které se nacházíme (aktivní hranu) a pozici na té hraně (aktivní délka). Tato pozice se posouvá, pokud je na aktivní hraně nalezen přečtený znak, pokud dojdeme na konec hrany, jako aktivní uzel se nastaví uzel na konci této hrany. Pokud se nacházíme v uzlu, jako aktivní hrana se nastaví hrana začínající přečteným znakem. Prohledávání se zastaví, když následující znak na aktivní hraně neodpovídá přečtenému znaku, nebo pokud se nacházíme v uzlu, ve kterém chybí hrana začínající přečteným znakem. Nyní je potřeba aktualizovat strom. Pokud se nacházíme v uzlu, je třeba přidat novou hranu s novým listem. Pokud se nacházíme uprostřed hrany, je třeba hranu rozdělit a novou hranu s novým listem přiřadit novému uzlu, který rozděluje aktivní hranu. Tuto úpravu je potřeba provést i na všechny suffixy řetězce daného aktivním uzlem, aktivní hranou a aktivní délkou. K tomu slouží odkazy mezi uzly zvané suffix links. Ty propojují vnitřní uzly tak, že suffix link uzlu odkazuje na uzel, který představuje jeho o 1 kratší suffix. Pomocí těchto odkazů se postupně dostaneme až ke kořenu, přičemž každý uzel v konstantním čase aktualizujeme. Počet navštívených uzlů odpovídá délce suffixu, lineární složitost je tedy zachována. Po aktualizaci stromu se opět nacházíme v kořenu, tedy jako aktivní uzel je nastaven kořen a aktivní délka je vynulována a procházení textu pokračuje.

Úpravy oproti původnímu Ukkonenovu algoritmu Vzhledem k tomu, že suffix tree je vytvářen pro množinu řetězců namísto jednoho řetězce, je třeba počítat s tím, že mohou nastat situace, které při konstrukci stromu pomocí obyčejného Ukkonenova algiritmu nenastanou. Jako příklad si můžeme představit situaci, kdy máme v množině dva stejné řetězce -

struktura stromu bude stejná, jako by byl v množně jen jeden, ale počty výskytů podřetězců se budou lišit. Neplatí už tím pádem ve všech případech pravidlo, že počet výskytů se rovná počtu listů podgrafu. Další zásadní rozdíl je v tom, že pokud je v původním suffixovém stromu přidán list, je jistota, že tento uzel zůstane listem po celou dobu běhu algoritmu. To bohužel v naší situaci neplatí. Představme si, že máme dva řetězce, řetězec A a řetězec B, kde A je prefix řetězce B. Jako první je zpracován řetězec A. Při procházení řetězce B nastane situace, kdy je třeba k listům stromu přidat nového potomka, list tedy přestane být listem. Pokud bychom postupovali podle obyčejného Ukkonenova algoritmu, narazíme na problém, kdy aktualizujeme list a je potřeba akualizovat i všechny jeho suffixy. Protože v původním Ukkonenovu algoritmu se vytvářejí suffix linky jen u vnitřních uzlů (u listů nejsou potřeba) je třeba doplnit algoritmus o vytváření suffix linků u všech uzlů, včetně listů. Potom už není problém aktualizovat list, protože můžeme postupovat pomocí suffix linků až do kořene, stejně jako u vnitřních uzlů. Další problém, který je třeba zohlednit, je situace, kdy doběhne algoritmus v okamžiku, kdy se nacházíme uprostřed hrany. V tom případě je třeba hranu rozdělit (a stejně tak i u všech kratších suffixů), protože podřetězec na této pozici bude mít u procházeného souboru o 1 větší počet výskytů než podřetězec představovaný koncovým uzlem aktivní hrany.

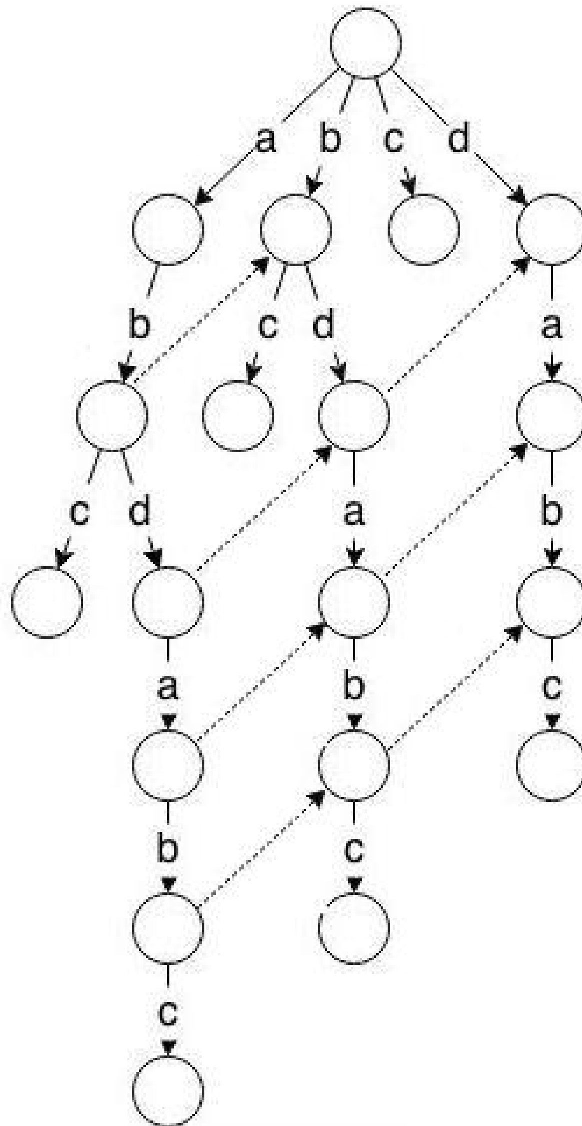
Součet počtu výskytů Součet vychází z pravidla, že počet výskytů řetězce se rovná počtu listů v podstromu, jehož je uzel kořenem. Toto pravidlo v našem případě nemusí vždy platit, algoritmus je tedy třeba trochu upravit. Stále platí, že k výskytům uzlů se přičítají výskyty potomků, list však může mít více než jeden výskyt a vnitřní uzel může mít větší počet výskytů než součet výskytů u jeho potomků. Při konstrukci stromu se přiřadí počty výskytů uzlům - hlavně listům, ale také vnitřním uzlům – v případě, že v tomto uzlu skončíme po zpracování tokenizovaného souboru. Ve druhé fázi proběhne součet počtů výskytů – k počtům výskytů vnitřních uzlů se přičtou počty výskytů potomků. Takto postupujeme směrem od listů ke kořenu. Sčítání musí probíhat ve správném pořadí, k tomu poslouží fronta zpracovaných uzlů.

4.4.2.2 Suffix trie

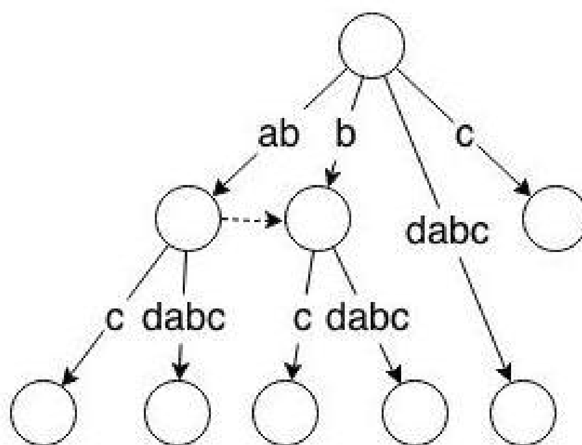
Suffixový strom se dá sestavit i jednodušší metodou, kdy na hranách nejsou podřetězce, ale znaky. Tento algoritmus dokáže vyhledat vzorek v čase lineárně závislém na délce vzorku, jeho konstrukce však pracuje s kvadratickou složitostí[12]. Použití suffixového stromu pomocí Ukkonenova algoritmu je efektivnější, je však složitější na pochopení a implementaci.

4.5 Volba filtrů podřetězců

Filtr podřetězců má za úkol zredukovat délku vektoru počtů výskytů. Zde je třeba najít vhodný kompromis mezi úsporou paměti a ztrátou dat. Mezi nejzákladnějšími kriterii filtrování podřetězců může být počet souborů, ve kterých se vyskytuje - pokud se vyskytuje pouze v jednom souboru, informace o počtu jeho výskytů je při hledání plagiátů nepodstatná. Pokud se naopak podřetězec vyskytuje ve velkém počtu souborů, je pravděpodobné, že shodnosti ve výskytech jsou náhodné, protože ze jedná o podřetězec, který se běžně vyskytuje v LaTeXových dokumentech, aniž by to musely být plagiáty. Dalším kritériem může být minimální nebo maximální délka podřetězce. Pokud stanovíme minimální délku, nastanou



Obrázek 4.4: Suffix trie pro řetězec *abdabc*



Obrázek 4.5: Suffix tree pro řetězec *abdabc*

dva problémy - pokud nastavíme minimální délku malou (například 2 a více), redukce délky vektoru bude minimální, pokud stanovíme tuto minimální délku vyšší, můžeme narazit na problémy při porovnávání malých souborů. Pro srovnání je naimplementováno více filtrů a na testovacích datech je zhodnocena jejich uplatnitelnost.

4.6 Volba metriky

4.6.1 Kosinová podobnost

Kosinová podobnost je jednoduše kosinus mezi dvěma vektory. Hodnota této vzdálenosti je tedy v rozmezí 0 až 1, kde 0 je nejmenší a 1 je největší podobnost. Používá u některých existujících nástrojů na detekci plagiátů jako metrika podobnosti, příkladem je systém CHECK, zmíněný v předchozí kapitole.

4.6.2 Manhattanská vzdálenost

Tato vzdálenost, nazývaná též Taxikářská vzdálenost, patří mezi nejjednodušší metriky porovnávání vektorů. Její název vychází z pravoúhlého uspořádání ulic na Manhattanu, tvořících pravidelnou mřížku. Taxikářská se nazývá proto, že vyjadřuje vzdálenost, kterou ujede taxikář v této síti ulic z jednoho bodu do druhého. Pro výpočet stačí sečíst rozdíly jednotlivých složek vektoru viz vzorec:

$$d(\vec{v}, \vec{w}) = \sum_{i=1}^{n} |v_i - w_i| \tag{4.1}$$

4.6.3 Eukleidovská vzdálenost

Eukleidovská vzdálenost je ze všech metrik rozdílnosti vektorů nejpoužívanější. Často, pokud se uvádí pojem vzdálenost, je tím automaticky myšlena Eukleidovská vzdálenost. V dvojrozměrném prostoru se vypočítá známou Pythagorovou větou, v n-rozměrném prostoru je zobecněna vzorcem:

$$d(\vec{v}, \vec{w}) = \sqrt{\sum_{i=1}^{n} (v_i - w_i)^2} \tag{4.2}$$

Kromě geometrie má využití například v oblasti data miningu. Jako příklad můžeme uvést klasifikační metodu k-means, která rozdělí množinu vektorů na k tříd podle Eukleidovské vzdálenosti od jejich těžišť.

4.6.4 Minkowského vzdálenost

Pro předchozí dvě metriky existuje zobecnění v podobě Minkowského vzdálenosti.

$$d(\vec{v}, \vec{w}) = \sqrt[\lambda]{\sum_{i=1}^{n} |v_i - w_i|^\lambda} \tag{4.3}$$

Manhattanská vzdálenost je speciálním případem Minkowského vzdálenosti s parametrem $\lambda = 1$, pro Eukleidovskou vzdálenost platí $\lambda = 2$.

4.6.5 Mahalanobisova vzdálenost

Mahalanobisova vzdálenost se podobá Eukleidovské vzdálenosti. Na rozdíl od všech dosud zmíněných vzdáleností je bezrozměrná. K výpočtu je třeba znát kovarianční matici (ve vzorci označenou S). Pokud tvoří S jednotková matice, je Mahalanobisova vzdálenost stejná jako Eukleidovská. Kovarianční matice slouží k tomu, aby se měřítko a sklon os přizpůsobily rozložení bodů v prostoru.

$$d(\vec{v}, \vec{w}) = \sqrt{(\vec{v} - \vec{w})S^{-1}(\vec{v} - \vec{w})} \tag{4.4}$$

Kovarianční matice, někdy také označovaná jako varianční-kovarianční matice, je čtvercová matice $n \times n$, kde n je délka vektorů, pro které je kovariance počítána. Na diagonále jsou hodnoty variance, na všech ostatních místech kovariance. Kovariance se spočítá následujícím vzorcem:

$$Cov(x, y) = (x_i - \overline{x})(y_i - \overline{y})/N \tag{4.5}$$

V tomto vzorci je N celkový počet vektorů, $\overline{x}$ a $\overline{y}$ jsou průměrné hodnoty proměnných, pro které kovarianci počítáme. Ze vzorce je patrné, že kovarianční matice je souměrná podle diagonály.

Jedna ze základních myšlenek při návrhu aplikace byla, že detekce plagiátů nebude jen na základě počtu výskytů, ale že zvýšená pozornost bude věnována podřetězcům, které jsou společné pro malý počet souborů a v ostatních se nevyskytují. Ty jsou totiž jako potenciální plagiáty nejvíce podezřelé. Z tohoto důvodu se zdá Mahalanobisova vzdálenost jako nejvhodnější ze zmíněných metrik. Její velikou slabinou je však veliká výpočetní náročnost. Pro její spočítání je třeba maticových výpočtů, které jsou, vzhledem k velkému rozměru vektoru, velice výpočetně náročné. Pro výpočet je třeba vytvoření inverzní matice a dvě maticová násobení, byla proto dána přednost výpočetně méně náročným metodám.

Mahalanobisova vzdálenost je využívána mimo jiné při klasifikaci. Příkladem využití Mahalanobisovy vzdálenosti u klasifikačních metod je shluková analýza. Objekty jsou pomocí shlukové analýzy rozdělovány do tříd na základě vzájemné podobnosti a rozdílnosti od objektů z jiných tříd.

4.6.6 Čebyševova vzdálenost

Mezi další vzdálenosti, které stojí alespoň za zmínku počítáme Čebyševovu vzdálenost, která vyjadřuje maximum ze všech rozdílů jednotlivých složek vektoru. Na šachovnici představuje počet tahů, které potřebuje král na přesun z jednoho hracího pole na druhé, proto se pro ni také používá pojmenování Šachová vzdálenost.

Kapitola 5

Popis implementace řešení

V této kapitole je popsána realizace řešení. Detekce plagiátů se skládá z několika na sebe navazujících fází, implementaci každé z nich je věnována podkapitola. Tyto podkapitoly jdou za sebou chronologicky tak, jak za sebou jdou jednotlivé fáze.

5.1 Volba programovacího jazyka

Jako programovací jazyk pro implementaci byla zvolena Java (JDK 1.6). Hlavním důvodem je platformní nezávislost. Dalším důvodem je, že aplikace sestává z mnoha funkčních jednotek, bude tedy třeba využít unit testů – k tomu výborně poslouží Javovský framework JUnit.

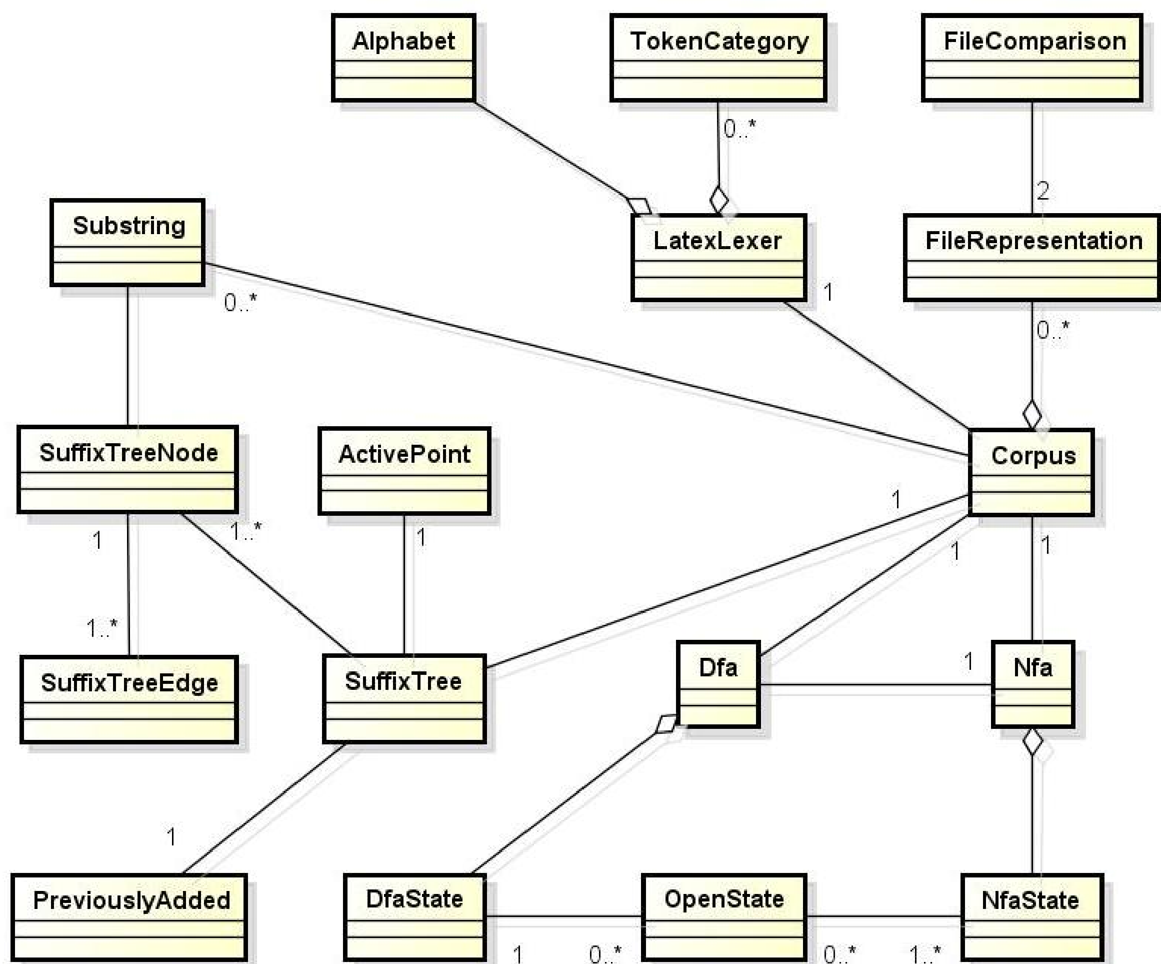
5.2 Implementace struktury tříd

Třída LatexLexer V této třídě se nachází logika pro tokenizaci. Třída obsahuje kolekci objektů třídy TokenCategory, které obsahují informace o typech tokenů potřebné k běhu lexikálního analyzátoru.

Třída TokenCategory Tato třída slouží jen pro uložení pomocných dat pro lexikální analyzátor. Obsahuje informace, které jsou vyžity při identifikaci různých typů tokenů. Tato třída obsahuje regulární výraz, pomocí kterého typ tokenu identifikujeme a případně extrahujeme text.

Třída Corpus Třída zastřešující celou množinu dokumentů. Obsahuje datové struktury pro uložení vstupních souborů. Kromě číselných sekvencí představujících tokenizované soubory obsahuje i množinu vektorů výskytů podřetězců v těchto souborech a metody pro jejich porovnání. Těchto metod je více a jejich srovnání je věnován prostor v kapitole Experimenty.

Třída Alphabet Třída obsahuje informace o abecedě, do které jsou vstupní soubory převedeny při tokenizaci. Znaky abecedy jsou čísla 0 až n, kde $n + 1$ je počet různých typů tokenů. Obyčejný text je 0, komentář je 1 a další znaky se do abecedy přidávají za běhu. V rámci třídy abeceda jsou datové struktury, ve kterých jsou uloženy dvojice token - číselná reprezentace a metoda pro převod tokenu na číslo.



Obrázek 5.1: Class diagram - PlagiTeX

Třída Nfa Třída pro uložení nedeterministického faktorového automatu obsahující metodu pro jeho konstrukci. Z důvodu úspory paměti a výpočetního času nejde o obecnou implementace NFA a datové struktury jsou speciálně zjednodušeny pro konkrétní potřeby aplikace, Jsou rozlišeny dva druhy stavů - počáteční, který je jako jediný nedeterministický, a všechny ostatní stavy, které jsou objekty třídy NfaState.

Třída NfaState Tato třída obdsahuje identifikaci souboru a pozice znaku v něm, odkaz na následující stav a znak na přechodu do příštího stavu. Jak už bylo zmíněno, jde o stav NFA, přesto má pouze jeden přechod do jednoho konkrétního stavu, je tedy deterministický.

Třída Dfa Třída obsahující algoritmus pro sestavení deterministického konečného faktorového automatu, neobsahující však jeho strukturu. Algoritmus pro sestavení automatu ze vstupního DFA je použit jen pro sestavení vektoru podřetězců, a tak je součástí třídy jen datová struktura pamatující si existující stavy, přechody však nejsou implementovány a automat jako takový vlastně není sestaven.

Třída DfaState Třída představující stav deterministického konečného automatu. Obsahuje kolekci následujících stavů.

Třída OpenState OpenState je otevřený stav při procházení grafu. Nese jen dočasně využitou informaci o procházených uzlech.

Třída SuffixTree Třída představující suffixový strom a logiku pro jeho sestavení. Má jedinou veřejnou metodu contruct, která sestaví suffixový strom, a několik neveřejných metod, které se při sestavování využívají. Metoda construct rovnou spočítá počty výskytů podřetězců a sestaví nezpracované (nevyfiltrované) vektory počtů výskytů. Z členských proměnných a datových struktur je veřejná pouze root - kořen stromu. Ke zbylým uzlům můžeme přistupovat přecházením z uzlu do uzlu pomocí hran. Dále obsahuje neveřejný seznam listů, ten se používá při sčítání počtů výskytů.

Třída SuffixTreeNode Tato třída představuje uzel suffixového stromu včetně odkazů na hrany, suffix linku a pole počtu výskytů v jednotlivých souborech.

Třída SuffixTreeEdge Tato třída představuje hranu suffixového stromu - definovanou pomocí indexu počátku podřetězce, konce podřetězce a indexu souboru, ve kterém se podřetězec představovaný hranou nachází.

Třída ActivePoint Pomocná třída představující místo v suffixovém stromě. Toto místo je definováno pomocí aktivního uzlu (posledního navštíveného uzlu), aktivní hrany (hrany na které se nacházíme) a aktivní délky (pozice na hraně, kde se nacházíme). Navíc obsahuje celkovou aktivní délku, což je vzdálenost z kořene do bodu, ve kterém se aktuálně nacházíme v grafu. Tato hodnota slouží při přiřazování hodnot délek podřetězců uzlům.

Třída PreviouslyAdded Pomocná třída obsahující odkaz na poslední přidaný vnitřní uzel a list. Slouží při vytváření suffix linků.

Třída FileRepresentation Třída obsahující důležité informace o souboru jako název, surová data a tokenizovanou číselnou reprezentaci . Obsahuje metodu na načtení ze souboru.

Třída FileComparison Jedna z nejjednodušší tříd, obsahuje pouze dvojici souborů a číslo, označující jejich rozdílnost. Posloupnost objektů této třídy, seřazená vzestupně podle rozdílnosti, je konečným výstupem algoritmu.

5.3 Lexikální analyzátor

Lexikální analyzátor slouží k převodu vstupního souboru na posloupnost čísel, které je dále zpracovávána.

5.3.1 Regulární výrazy

K rozpoznávání lexikálních elementů slouží engine Matcher, pomocí kterého můžeme sekvenčně vyhledávat v textu. Nevyhledáváme přitom pouze podle jednoho vzorku (patternu), ale máme jeden vzorek pro každý typ tokenu, s výjimkou obyčejného textu, ten je extrahován z volných míst v textu mezi rozpoznanými tokeny. Všechny vzorky se nakonec spojí do jednoho velkého regulárního výrazu, kde jsou jednotlivé vzorky rozlišené jako takzvané Capturing groups.

5.3.1.1 Matcher

Matcher umožňuje provádět sekvenční operace na základě vzorku (třída Pattern) [18]. Třída Pattern představuje zkompilovanou podobu regulárního výrazu. Pomocí matcheru můžeme vyhledávat výskyty vzorku v textu, nahrazovat je zadaným řetězcem, nebo rozhodovat, jestli text odpovídá vzorku. Podřetězec textu, se kterým matcher pracuje, se nazývá region. Implicitně je region celý text, můžeme ho však také ručně přenastavit. V lexikálním analyzátoru je použita metoda find, která se pokusí vyhledat první výskyt vzorku v textu buďto od začátku regionu, nebo od prvního znaku za posledním výskytem. Matcher si pamatuje poslední výskyt a metoda find funguje sekvenčně, stačí ji tedy opakovaně volat, aniž by při každém volání vracela stejné výsledky. Matcher je možné resetovat, přičemž příjdeme o informace o dosavadních operacích a jako region se nastaví celý text.

5.3.1.2 Capturing groups

Capturing groups v regulárních výrazech se značí kulatými závorkami. Slouží k rozdělení regulárního výrazu na menší celky a v lexikálním analyzátoru jsou využity k dvěma účelům: rozpoznání typu tokenu a extrakce vnitřního textu. Pokud regulární výraz rozpozná hledaný vzorek v textu, můžeme extrahovat pouze určitou část rozpoznaného textu. Například u příkazu `\begin{<text>}` můžeme extrahovat text uvnitř závorky. Capturing groups může být

<table>
<tr><th>Typ tokenu</th><th>Kód</th><th>Capturing group</th></tr>
<tr><td>Obyčejný text</td><td></td><td></td></tr>
<tr><td>Začátek prostředí</td><td><code>\\begin\{([\w*]+)\}</code></td><td>1</td></tr>
<tr><td>Konec prostředí</td><td><code>\\end\{([\w*]+)\}</code></td><td>3</td></tr>
<tr><td>Příkaz</td><td><code>\\(\w+)</code></td><td>5</td></tr>
<tr><td>Otevírající závorka</td><td><code>\[{|</code></td><td>7</td></tr>
<tr><td>Zavírající závorka</td><td><code>\]}|</code></td><td>8</td></tr>
<tr><td>Komentář</td><td><code>\%(.*)\\n</code></td><td>9</td></tr>
</table>

Tabulka 5.1: Regulární výrazy k rozpoznávání lexikálních elementů

v jednom vyhledávaném vzorku více a jsou identifikovány indexem. Indexy se přiřazují zleva doprava od 1 výš. Index 0 představuje celý regulární výraz. Regulární výrazy a příslušné capturing groups jsou vypsány v tabulce:

Z těchto capturing groups je složen dlouhý vzorek, jenž má následující podobu:

$$(R_1)|(R_3)|(R_5)|(R_7)|(R_8)|(R_9) \tag{5.1}$$

V tomto vzorku jsou obsaženy všechny regulární výrazy, označené R_i, kde i je číslo capturing group. Důvod, proč netvoří tato čísla souvislou postupku je ten, že některé regulární výrazy v sobě mají obsaženou „zahnízděnou"' capturing group. Například regulární výraz pro začátek prostředí obsahuje kulaté závorky a tím pádem capturing group uvnitř capturing group. Číslo této zahnízděné capturing group je 2, neboť číslování probíhá vždy zleva doprava.

Části vstupního textu, které nejsou rozpoznány jako součát žádného lexému z tabulky výše, jsou zpracovány jako obyčejný text, pokud neobsahují pouze prázdné znaky – v tom případě nejsou zpracovány vůbec.

5.4 Vytvoření číselné reprezentace dokumentu

Logika pro převod rozpoznaných tokenů na jejich číselnou reprezentaci je obsažena ve třídě Alphabet. Pro každý rozpoznaný token se lexikální analyzátor dotáže metodou getCharacter na číselný kód tokenu. Pokud již číselná reprezentace pro daný token existuje, je pouze navrácen číselný kód, pokud ještě neexisuje, je potřeba ji vytvořit. Aplikace si pamatuje věechny dosud převedené tokeny a jejich číselné kódy. K uložení těchto informací slouží kolekce zvaná HashMap, která obsahuje páry klíč-hodnota. Pro každý typ tokenu je zvláštní HashMap, kde jako klíč slouží klíčové slovo (například „section") a jako hodnota jeho číselná reprezentace. Číselná reprezentace je 0 pro obyčejný text, 1 pro komentář a vyšší hodnoty, rostoucí s jednotkovým krokem, pro ostatní tokeny. Číselné hodnoty jednotlivých tokenů tedy závisí na pořadí jejich prvních výskytů.

Převod na číselnou reprezentaci si demonstrujeme na jednoduchém příkladu:

Token	Typ tokenu	Kód
Obyčejný text	Obyčejný text	0
Komentář	Komentář	1
\section	Příkaz	2
{	Otevírající závorka	3
}	Zavírající závorka	4
\begin{itemize}	Začátek prostředí	5
\item	Příkaz	6
\end{itemize}	Konec prostředí	7

Tabulka 5.2: Ukázka číslených reprezentací tokenů

```
% komentář
\section{Sekce}
V~následujícím seznamu
se nachází několik položek
\begin{itemize}
\item Jedna
\item Dva
\begin{itemize}
\item Tři
\item Čtyři
\end{itemize}
\item Pět
\end{itemize}
```

Pro komentář a obyčejný text jsou číselné kódy dány předem (0 pro text a 1 pro komentář), pro ostatní tokeny jsou přiděleny v pořadí výskytu. Číselné kódy pro tento příklad jsou vypsány v tabulce. Pokud pomocí abecedy v této tabulce přeložíme vstupní soubor na posloupnost čísel, vyjde nám pro tento dokument číselná reprezentace:

$$1023040566566767$$

5.5 Konstrukce konečného faktorového automatu

5.5.1 Konstrukce nedeterministického konečného faktorového automatu

Implementace nedeterministického konečného faktorového automatu je z důvodu úspory paměti a výpočetního času zjednodušená. Automat je zkonstruován pouze k jednomu konkrétnímu účelu, proto nebylo třeba implementovat datové struktury a třídy tak, aby pomocí nich šlo zkonstruovat jakýkoli NFA. Konstrukce vychází z faktů, že jediný nedeterministický stav

(stav, kde existuje pro jeden vstupní znak přechod do více následujících stavů) je počáteční stav. Ostatní stavy mají vždy jeden vstupní a jeden vystupující přechod (s výjimkou tavů představujícch poslední pozici v souboru, ten má pouze vstupní přechod) a všechny jsou konečné.

5.5.1.1 Implementace počátečního stavu

Počáteční stav je implementován velice minimalisticky. Na rozdíl od ostatních stavů nepředstavuje konkrétní výskyt řetězce ve vstupní posloupnosti čísel (tokenizovaném vstupním souboru), jediné informace, které je potřeba do jeho implementace obsáhnout je množina přechodů vycházejících z tohoto stavu. Stav je tedy implementován pomocí již dříve zmíněné kolekce HashMap, kde jako klíč je vstupní znak a jako hodnota je množina následujících stavů, realizovaná pomocí kolekce ArrayList. Tato kolekce obsahuje stavy představující jednotlivé výskyty daného znaku v číselných reprezentacích souborů.

5.5.1.2 Implementace ostatních stavů

Každý z ostatních stavů představuje jeden podřetězec v některé ze vstupních číselných posloupností. Jsou implementovány pomocí třídy NfaState, která obsahuje následující informace o stavu:

- Výstupní symbol - symbol na výstupním přechodu
- Následující stav (objekt třídy NfaState)
- Index souboru - identifikace vstupního souboru
- Index posledního znaku podřetězce
- Čislo stavu . slouží při konstrukci DFA

5.5.2 Převod na deterministický konečný automat

Algoritmus pro převod NFA na DFA je využit jako prostředek k vypočítání vektorů výskytů podřetězců v číselných reprezentacích vstupních souborů. Celý postup je založen na myšlence, že každý stav NFA představuje určitý podřetězec v jednom konkrétním souboru. Při konstrukci DFA dochází ke slučování stavů, během kterého jsou do jednoho stavu DFA sloučeny všechny stavy NFA představující výskyt stejného podřetězce na různých místech různých číselných posloupností. Počet stavů NFA sloučených do jednoho stavu DFA tedy představuje počet výskytů určitého podřetězce v korpusu. Vzhledem k tomu, že jsou stavy NFA opatřeny informací o tom, ke kterému souboru náleží, můžeme tedy spočítat, kolikrát se který podřetězec vyskytuje v daném souboru (respetive jeho číselné reprezentaci). Z těchto počtů výskytů je už za běhu programu sestavován vektor pro každý soubor. Pokud bychom nepoužili opatření na redukci déky vektoru (popsaná později), byl by rozměr těchto vektorů roven počtu stavů DFA bez počátečního stavu.

5.5.2.1 Množina existujcích stavů

Existující stavy DFA vznikly sloučením množiny stavů NFA, k identifikaci existujícího stavu tedy stačí znát čísla stavů, ze kterých byl vytvořen. Vzhledem k tomu, že se nesnažíme o konstrukci funkčního automatu a algoritmus pro převod NFA na DFA slouží jen jako prostředek k spočítání výskytů podřetězců „jediné co potřebujeme vědět je, jestli stav existuje nebo ne. Toho docílíme pomocí kolekce HashSet, která obsahuje klíče, ke kterým není přiřazena žádná hodnota, důležitá je pouze informace zda je klíč přítomen v množině nebo není. Jako datový typ tohoto klíče je zvolen BigInteger, tedy celé číslo s neomezenou maximální velikostí. Klíč se počítá tak, že každý bit BigIntegeru představuje jeden konkrétní stav NFA. Pokud je stav přítomen v množině stavů, ze kterých je stav DFA složen, je bit nastaven jako 1. Pozice onoho bitu je dána dříve zmíněným číslem NFA, které má rozsah 0 až $(n-1)$, kde n je počet stavů NFA, pokud nepočítáme počáteční stav.

5.5.2.2 Konstrukce vektoru počtů výskytů

Vektory výskytů maji před spuštěním algoritmu dimenzi 0, při vytvoření nového stavu je přidána dimenze k vektoru počtů výskytů podřetězců.

5.5.3 Problémy s rychlostí algoritmu

V praxi byl algoritmus shledán časově i paměťově nehospodárný. Doba běhu nad množinou větších souborů dostahuje až desítek vteřin, u většího objemu dat můžeme narazit i na nedostatek paměti. Při spuštění nad korpusem tvořeným 33 soubory o velikosti 1–92kB trvá běh aplikace přibližně 20s, zatímco řešení se suffixovým stromem doběhne během vteřiny. Proto je konstrukce vektoru počtů výskytů pomocí podmnožinové konstrukce DFA ponechána mezi zdrojovými soubory, není však využita jako součást aplikace.

5.6 Konstrukce suffixového stromu

Datová struktura použitá v aplikaci není tradičním suffixvým stromem. Původní Ukkonenův algoritmus bylo třeba modifikovat, neboť se nejedná o strom obsahující prefixy jednoho řetězce, ale o strom se všemi suffixy konečné množiny řetězců.

5.6.1 Implementace datové struktury

5.6.1.1 Implementace uzlu

Třída SuffixTreeNode představuje uzel suffixového stromu a uzel je reprezentací podřetězce z množiny vstupních řetězců. Obsahuje i některé proměnné, které neslouží přímo pro konstrukci suffixového stromu, většinou jsou použity při sčítání počtů výskytů. Třída obsahuje následující informace a datové struktury:

- Množinu hran vycházejících z uzlu - implementovanou jako HashMap, kde jako klíč slouží první znak hrany a jako hodnota objekt třídy SuffixTreeEdge

- Suffix link - odkaz na uzel představující suffix kratší o 1
- Odkaz na rodiče - uzel, jehož je tento uzel potomkem, nemá význam při konstrukci, slouží při počítání výskytů řetězců v hotovém stomě
- Počet započítaných potomků - slouží také při sčítání počtu výskytů, pokud je roven počtu vycházejících hran, znamená to, že podstrom, jehož kořenem je tento uzel, je už zpracován
- Pole počtu výskytů - pro každý soubor obsahuje počet výskytu podřetězce v daném souboru

5.6.1.2 Implementace hrany

Třída SuffixTreeNode představuje hranu v suffixovém stromu. Třída obsahuje následující informace a datové struktury:

- Pozice začátku podřetězce ve vstupním dokumentu
- Pozice konce podřetězce ve vstupním dokumentu
- Index dokumentu - specifikuje, ve kterém dokumentu se nachází podřetězec vymezený předchozími dvěma proměnnými (navíc oproti původnímu suffixovému stromu)
- Koncový uzel - objekt třídy SuffixTreeNode

5.6.1.3 Pomocná třída ActivePoint

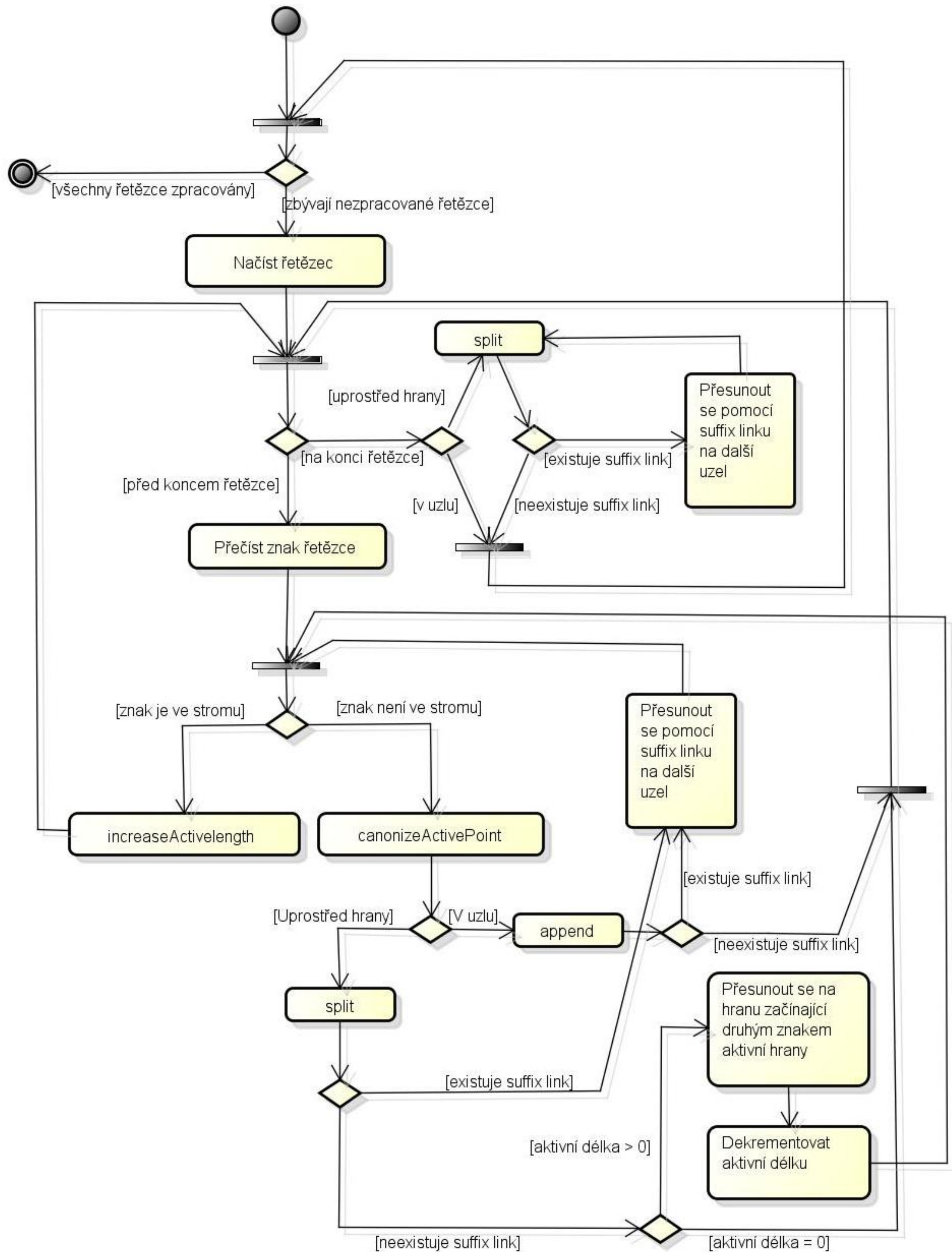
Obsahuje informace o aktuální pozici v suffixovém stromě při procházení. Obsahuje odkaz na poslední navštívený uzel (aktivní uzel), aktivní hranu - hranu uprostřed níž se nacházíme a aktivní délku - pozici na této hraně, na které se nacházíme.

5.6.2 Implementace Ukkonenova algoritmu

Implementace vychází z Ukkonenova algoritmu, nedrží se ho však do detailů. Pravděpodobně by bylo možné ještě efektivitu algoritmu vylepšit, například pomocí efektivnější realizace množiny hran (ta je tvořená pomocí datové struktury HashMap), stávající řešení však pracuje rychle i s velkou množinou dat. Implementace suffixového stromu je rozdělena na několik metod, z nichž každá je v tomto textu zvlášť popsána, a její volání je zachyceno v activity diagramu pro větší názornost.

5.6.2.1 increaseActiveLength

Tato metoda je volána, je-li ve stromu nalezen přečtený znak. Způsobí pouze pozunutí pozice ve stromu. V případě, že se nacházíme uprostřed hrany, je pouze inkrementována pozice na hraně (aktivní délka). Pokud se takto dostaneme do uzlu, je aktivní délka vynulována. Pokud se před voláním nacházíme v uzlu, je jako aktivní hrana nastavena hrana začínající přečteným znakem a aktivní délka se nastaví na hodnotu 1.



Obrázek 5.2: Activity diagram - konstrukce suffixového stromu

5.6.2.2 addNode

Tato metoda je volána, pokud není přečtený znak nalezen v suffixovém stromu. Aktualizuje strom v aktuálním uzlu pomocí volání addNodeToActivePoint (viz níže) a posouvá se při tom po suffix links až do kořene. Pokud se nacházíme v kořenu a aktivní délka je větší než 0 (nacházíme-li se uprostřed hrany), nepřecházíme na další suffix pomocí suffix linku, ale pomocí přechodu na hranu představujcí suffix aktivní hrany. To provedeme tak, že vezmeme znak na druhé pozici na aktivní hraně. jako novou aktivní hranu zvolíme hranu začínající tímto znakem a dekrementujeme aktivní délku. Takto postupujeme, dokud je aktivní délka větší než 0. Jakmile se nacházíme v kořeni a aktivní délka je 0, aktualizace stromu je hotova.

5.6.2.3 addNodeToActivePoint

Nejprve volá funkci CanonizeActivePoint, tím aktualizuje pozici ve stromu tak, aby se nacházela správně v grafu (při posunu pomocí suffix linku se může stát, že je aktivní délka větší než délka aktivní hrany, potom je potřeba posun do dalšího uzlu, abychom správně ukazovali na existující pozici v grafu). Aktualizuje strom v daném místě, aktualizuje pouze jeden suffix (addNode aktualizuje suffix a všechny jeho suffixy). Podle pozice, ve které se nacházíme, volá jednu ze 3 metod:

- append - pokud se nacházíme v uzlu
- split - pokud se nacházíme uprostřed hrany
- increaseActivelength - pokud už ve stromu znak existuje

5.6.2.4 append

Přidá k danému uzlu novou hranu a nový list. Nová hrana směřuje z původního uzlu do nového uzlu. Listu je přiřazen 1 výskyt u zrovna procházeného souboru a 0 výskytů u ostatních souborů.

5.6.2.5 split

Rozdělí danou hranu nově přidaným vnitřním uzlem a přidá k novému uzlu hranu vedoucí do nového listu. Listu je přiřazen 1 výskyt u zrovna procházeného souboru a 0 výskytů u ostatních souborů.

5.6.2.6 canonizeActivePoint

Při posunu pomocí suffix linků může dojít k situaci, kdy je aktivní délka (pozice na hraně) větší než délka aktivní hrany. Tato situace nastává, když je hrana vycházející z uzlu, na který ukazuje suffix link, na nějakém místě rozdělena, a aktivní hrana na tomto místě rozdělena není. V tom případě je aktivní délka zkrácena o zbývající délku hrany a přejde se do dalšího uzlu. Tento postup se opakuje, dokud není aktivní délka menší než délka aktivní hrany.

5.6.3 Úpravy oproti původnímu Ukkonenovu algoritmu

Suffix linky mezi listy Algoritmus si pamatuje poslední přidaný vnitřní uzel a poslední přidaný list pomocí třídy PreviouslyAdded. Poslední přidaný vnitřní uzel slouží k vytváření suffix linků vnitřních uzlů a poslední přidaný list slouží k vytváření suffix linků listů.

Půlení hran a úpravy počtu výskytů Pokaždé, když je dokončeno zpracovávání řetězce. pokud se nenacházíme v kořenu, nebo aktivní délka je větším než 0, je potřeba upravit počty výskytů. V případě, že se po zpracování tokenizovaného souboru nacházíme v uzlu, inkrementujeme pro daný soubor počet výskytů u příslušného řetězce a všech jeho suffixů. V případě, že se nacházíme uprostřed hrany, je potřeba hranu rozpůlit, protože koncový bod hrany se bude lišit počtem výskytů od prostředního (nově přidaného) uzlu. Nově přidaný uzel má přiřazen 1 výskyt u daného souboru, 0 u ostatních. Rozpůlení hran se aplikuje i na hrany představující suffixy této hrany.

5.6.4 Součet počtu výskytů

Při konstrukci suffixového stromu jsou listům přiřazeny při jejich vytvoření hodnota 1 u souboru, který je zrovna procházen, a 0 u ostatních souborů. Uzel, ve kterém skončíme po procházení souboru, má počet výskytů zvýšen o 1 u daného souboru. Po konstrukci suffixového stromu dojde na sčítání hodnot - to probíhá směrem od listů ke kořenu. Pro každý uzel se spočítá suma vlastních výskytů a výskytů jeho potomků. To se realizuje pomocí fronty uzlů ke zpracování. Nejprve jsou do fronty přidány všechny listy. Výskyty odebraného uzlu přičteme k výskytům jeho rodiče a počet zpracovaných potomků rodiče je inkrementován. V případě, že se počet zpracovaných potomků rodiče rovná počtu z něj vycházejících hran (tedy celkovému počtu potomků), je uzel zpracován a je přidán do fronty. Z fronty jsou uzly odebírány, dokud není prázdná, potom je součet výskytů dokončen.

5.7 Implementace filtrů

Po dokončení konstrukce vektoru počtu výskytů je naplněna kolekce ArrayList objektů třídy Substring. Úkolem filtru je smazat z kolekce podřetězce, které nesplňují podmínku pro průchod filtrem. Podmínky se mohou skládat z více kritérií a jsou založeny na vlastnostech podřetězců, jako je třeba délka či počet reprezentací dokumentů, ve kterých se vyskytuje. Délka výstupního vektoru může být buďto libovolná, nebo limitovaná na určitý počet prvků (to v případě, že by výpočet metriky byl časově nebo paměťově náročný a pro větší rozměr vektoru by trval neúnosně dlouho). K srovnání kvality filtrů slouží speciální testovací sada dat. Experimentálně byly naimplementovány i některé filtry, o kterých se dá předem předpokládat, že budou vykazovat špatné výsledky. Implementovány byly následující filtry:

- Filtr na základě počtu souborů, ve kterých se vyskytuje - podřetězec se musí vyskytovat alespoň v počtu souborů daném vstupním parametrem

- Filtr na základě počtu souborů, ve kterých se vyskytuje - podřetězec se musí vyskytovat nanejvýš v počtu souborů daném vstupním parametrem

- Filtr na základě minimální délky podřetězce
- Filtr, který vybere ze všech podřetězců n nejkratších
- Filtr, který vybere ze všech podřetězců n nejdelších
- Filtr, který vybere ze všech podřetězců n s výskyty v nejmenším počtu souborů

5.8 Metriky porovnávání

Součástí práce je srovnání různých metrik, proto bylo vybráno několik metrik, které byly shledány nejvhodnější, a ty byly naiplementovány. Při spuštění programu z příkazové řádky je možné zvolit, jakou metrikou se budou vektory výskytů porovnávat. Mahalanobisova vzdálenost, o které se původně uvažovalo jako o nejvhodnější metrice, nakonec nebyla implementována vzhledem k velikým nárokům na výpočetní prostředky. Volba metrik se tedy zůžila na Eukleidovskou vzdálenost a Manhattanskou vzálenost a jejich modifikace. Pro obě vzdálenosti byly naimplementovány vážené variaty, což znamená, že výsledná vzdálenost byla vydělena koeficientem - vahou. Váha je tvořena počtem podřetězců, které se v reprezentaci daného dokumentu vyskytují. Při porovnávání dvou dokumentů je jako váha použit počet podřetězců v delším (obsahujícím více podřetězců) ze dvou porovnávaných tokenizovaných dokumentů. Tato úprava má za účel eliminovat zkreslení způsobené rozdílnými délkami souborů. Kratší soubory totiž mají méně podřetězců a tím pádem i méně rozdílných podřetězců a jsou tím pádem pravděpodobněji vyhodnoceny jako plagiáty.

Naimplementovány jsou následující metriky:

- Eukleidovská vzdálenost
- Vážená Eukleidovská vzdálenost
- Manhattanská vzdálenost
- Vážená Manhattanská vzdálenost

Pokud není zvolena metrika, je implicitně použita vážená Eukleidovská vzdálenost.

Původně bylo v plánu implementovat Mahalanobisovu vzdálenost jako hlavní metriku. Na rozdíl od ostatních metrik neporovnává pouze dva body ve vektorovém prostoru, ale počítá s rozložením všech bodů. K tomu využívá inverzní kovarianční matici. Už v počátku se však ukázalo, že vzdálenost nemůže být použita v prostoru o tolika rozměrech (1 podřetězec představuje jeden rozměr) z důvodu veliké paměťové a výpočetní složitosti. První krok implementace bylo vytvoření metody na výpočet vzdálenosti s jednotkovou maticí místo inverzní kovarianční matice, druhým bylo implementovat metodu na tvorbu inverzní kovarianční matice. Už první krok však ukázal, že výpočet vzdálenosti je příliš paměťově náročný.

Kapitola 6

Experimenty

Kromě teoretických úvah o tom, jaké postupy je vhodné využít při detekci plagiátů ve formátu LaTeX, je také vhodné ověřit jejich účinnost testováním. Implementace zahrnuje různé filtry a metriky, lze je tedy použít v mnoha kombinacích. Abychom zhodnotili, která kombinace filtru a metriky je pro praktické využití vhodná a která ne, byly sestaveny dvě množiny experimentálních dat. Nad těmito daty je spouštěna aplikace s různými konfiguracemi pomocí dávkového souboru. Výsledky jsou zobrazeny v grafech a slovně zhodnoceny. Závěrem této kapitoly je volba nejvhodnější konfigurace aplikace.

6.1 Vzdálenost versus vážená vzdálenost

Vážená vzdálenost je vzdálenost dvou vektorů vydělená vahou. Jako váha je použita maximální hodnota počtu podřetězců v porovnávaných souborech (tedy počet podřetězců v delším ze dvou porovnávaných souborů). Experimenty, které jsou popsány v této kapitole, pracují s různými verzemi téhož souboru, testovací soubory jsou tedy téměř stejně velké. Proto mohou být při srovnání použity nevážené vzdálenosti, vzhledem k tomu, že poměry metrik jsou téměř stejné jako u vážených vzdáleností. V praxi však jsou využity vážené vzdálenosti, aby eliminovaly nepoměr v metrikách při porovnávání krátkých a dlouhých souborů.

6.2 Test vlivu procentuálních změn na metriku

První ze srovnávacích testů je založen na předpokladu, že metrika by měla růst s rostoucí rozdílností souborů, tedy soubory, které jsou si více podobné by měly mít metriku menší než soubory, ve kterých je mnoho rozdílů. Díky podpůrné aplikaci PlagiTeX Plagiator je možné vytvořit z původního souboru pozměněnou kopii procentuálně daným podílem změn. To umožňuje zachytit závislost procentuálních změn na metrice porovnávání do grafu a díky tomu graficky porovnat různé konfigurace aplikace. Za žádoucí výsledek se dá považovat, pokud graf roste stabilně a nekolísá. U grafů je třeba počítat s tím, že soubory byly náhodně generovány a kolísání grafu může být častečně způsobeno náhodou při vytváření testovacích dat.

6.2.1 Tvorba testovací sady

Testovací sada je tvořena různými verzemi téhož dokumentu (jako testovaný dokument je použita bakalářská práce psaná v LaTeXu). Z tohoto souboru je vytvořeno dvacet pozměněných souborů s různým rozsahem změn. Půlka těchto dokumentů je vytvořena z původního změnami obsahu textu, odsazení a komentářů v rozsahu 10%–100%, s krokem 10% (10%, 20%, 30% atd.). Druhá půlka souborů je vytvořena podobně, ale kromě těchto změn navíc dochází k změnám syntaxe prohazováním řádků, opět v rozsahu 10%–100%. V grafu je zachycena pouze druhá půlka, protože pro první půlku vycházejí nulové vzdálenosti – soubory jsou označeny za stejné.

6.2.2 Výsledky testu

6.2.2.1 Srovnání filtrů

Bez filtru První možnost je spočítat výslednou metriku nad množinou vektorů počtů výskytů nezpracovanou žádným filtrem. Objem dat ke zpracování je největší, zároveň však nedochází ke ztrátě dat. Křivka většinou roste, mírně však kolísá, což je způsobeno tím, že soubory jsou náhodně generované. Jedním z cílů experimentu je najít filtr, který dokáže toto kolísání eliminovat.

Minimální délka 5 Tento filtr má podobné vlastnosti jako ten předchozí - minimální ztráta dat, ale zároveň minimální redukce objemu dat. Tento filtr totiž odfiltruje všechny podřetězce délky 4 (délka není udávaná ve znacích, ale tokenech - lexikálních elementech) a méně a počet kombinací znaků o této délce je zanedbatelný oproti počtu všech podřetězců v korpusu.

Minimální délka 20 Při stanovení vyšší minimální délky je již redukce délky vektoru počtu výskytů znatelná. Křivka také stoupá rovnoměrně, bez výrazného kolísání, tento filtr se při experimentech ukázal jako jeden z nejlepších.

100 nejdelších Tento filtr se ukázal jako špatná volba - vzhledem k tomu, že byla délka vektoru značně zredukována, došlo k velké ztrátě dat, hlavní problém však je, že kratší podřetězce, pomocí kterých můžeme určit procentuální podobnost, jsou odfiltrovány, a tak na hodnoty grafu netvoří rostoucí křivku, jak by bylo žádoucí, ale hodnoty leží přibližně na vodorovné přímce.

100 nejkratších Zde jsou výsledky o něco lepší, stále však nedostačující. Dochází k velké redukci objemu dat a tím pádem i ztrátě informací. Křivka sice převážně roste, značně však kolísá. Vzhledem k tomu, že se do metriky počítají pouze krátké podřetězce, ztrácíme schopnost označit soubor jako podezřelý z plagiátorství na základě společného unikátního podřetězce.

Výskyt právě ve 2 souborech Tento filtr je dobrý v tom, že eliminuje podřetězce, které se vyskytují v mnoha souborech, a vyšší váhu tak mají podřetězce, které jsou unikátní, a pokud jsou ve dvou souborech a v ostatních se nevyskytují, může to znamenat, že daná pasáž byla okopírována. Pokud však máme porovnat dva soubory na základě procentuálních změn, výsledky jsou dost špatné - procentuální shodnost je z velké části dána společnými výskyty kratších řetězců běžně se vyskytujících v mnoha souborech.

Výskyt právě ve 2 souborech, minimální délka 20 Pokud je předchozí filtr zkombinován s filtrem na základě minimální délky, průběh křivky je velmi podobný, počet podřetězců je však zredukován.

Výskyt alespoň ve 2 souborech Tento filtr má za úkol odfiltrovat podřetězce, které se vyskytují pouze v jednom souboru, a tím pádem jsou pro porovnávání nepodstatné. Úkolem tohoto filtru je zredukovat délku vektoru počtu výskytů. Tento filtr sice způsobuje minimální ztrátu dat, zároveň je však redukce délky vektoru minimální - nejmenší ze všech vyzkoušených filtrů.

Výskyt alespoň ve 2 souborech, minimální délka 20 Tento filtr se ve srovnání s ostatními ukázal jako nejlepší. Průběh křivky je podobný jako u předchozího filtru, navíc dochází k větší redukci dat, protože jsou vynechány podřetězce nacházející se jen v 1 souboru, které jsou pro porovnání souborů nepodstatné.

Výskyt maximálně v 5 souborech Tento filtr dokáže poměrně dost zredukovat objem dat a výsledky nejsou úplně špatné - křivka má rostoucí tendenci, ale vykazuje, i když v mnohem menší míře, podobné nedostatky, jako filtr, kterým projdou podřetězce nacházejcící se v právě 2 souborech - tedy že je vyfiltrováno mnoho běžných podřetězců, jejichž počet výskytů je závislý na procentuální podobnosti dokumentů, křivka tedy neroste dostatečně rovnoměrně.

100 s výskytem v nejméně souborech Tento filtr má opět problém zmíněný u předchozího filtru a filtru, kterým projdou podřetězce nacházející se v právě 2 souborech. K tomu navíc dochází k značné redukci objemu dat a tím i informačního obsahu. Výsledky jsou tím pádem mizerné.

100 s výskytem v nejméně souborech, minimálně však ve 2 Tento filtr dopadl velmi podobně jako ten předchozí, tedy velmi špatně.

6.2.2.2 Srovnání metrik

Eukleidovská vzdálenost Oproti Manhattanské vzdálenosti hodnoty metriky pro jednotliuvé filtry nedosahují tak výrazných rozdílů. U většiny filtrů křivky stoupají celkem pravidelně a Eukleidovská vzdálenost tedy dosahuje dobrých výsledků.

Manhattanská vzdálenost Z grafu je znát, že u Manhattanské vzdálenosti roste hodnota metriky prudce s počtem rozměrů vektorového prostoru (tedy s počtem podřetězců). Tato vlastnost by mohla způsobit problémy u korpusu tvořeného soubory různých délek (délka vektoru počtů výskytů je sice pro všechny soubory stejná, pro kratší soubory je však vektor řidší). Testovací data použitá pro tuto kapitolu jsou tvořena téměř stejně velkými soubory, ve srovnávacích testů se to tedy neprojeví. Zkreslení metriky způsobené můžeme redukovat použitím vážené Manhattanské vzdálenosti, to je však třeba ověřit dalším testováním.

6.3 Test vlivu přítomnosti společného bloku textu na metriku

Druhý srovnávací test je založen na myšlence, že v plagiátu se může nacházet blok textu, který je společný s originálem, ale v ostatních souborech se nevyskytuje. Tento blok je dobré vodítko pro označení plagiátu a jeho přítomnost v dokumentu by se měla v metrice porovnávání projevit. Takový dokument by měl být označen jako podobnější originálu než ostatní. Testovací data jsou opět zpracována aplikací s různými konfiguracemi s cílem určit optimální konfiguraci.

6.3.1 Tvorba testovací sady

Testovací sada je opět tvořena různými verzemi téhož dokumentu (a stejného dokumentu jako v předchozím testu), nyní je však rozsah změn vždy 100%, všechny pozměněné soubory by tedy měly mít přibližně stejnou metriku při porovnání s původním souborem, s odchylkami danými náhodou při generování. V jednom z těchto souborů je ponechán krátký blok textu, který je stejný jako v původním dokumentu. Od tohoto souboru se při vhodné volbě konfigurace očekává, že metrika bude nižší, tedy že bude podobnější původnímu dokumentu než ostatní pozměněné soubory.

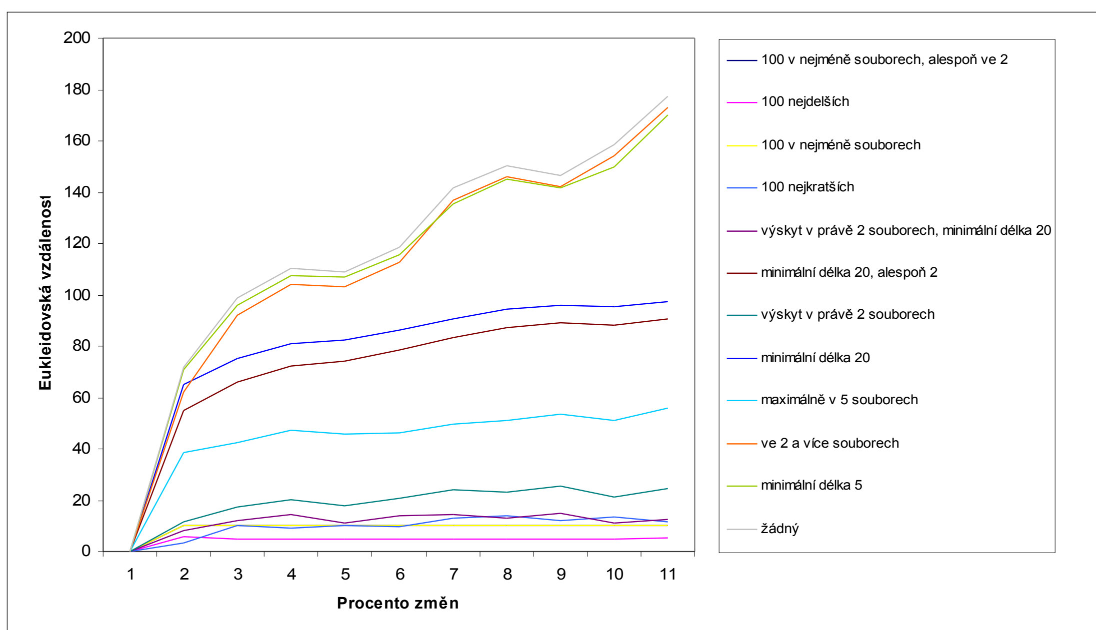
6.3.2 Výsledky testu

6.3.2.1 Srovnání filtrů

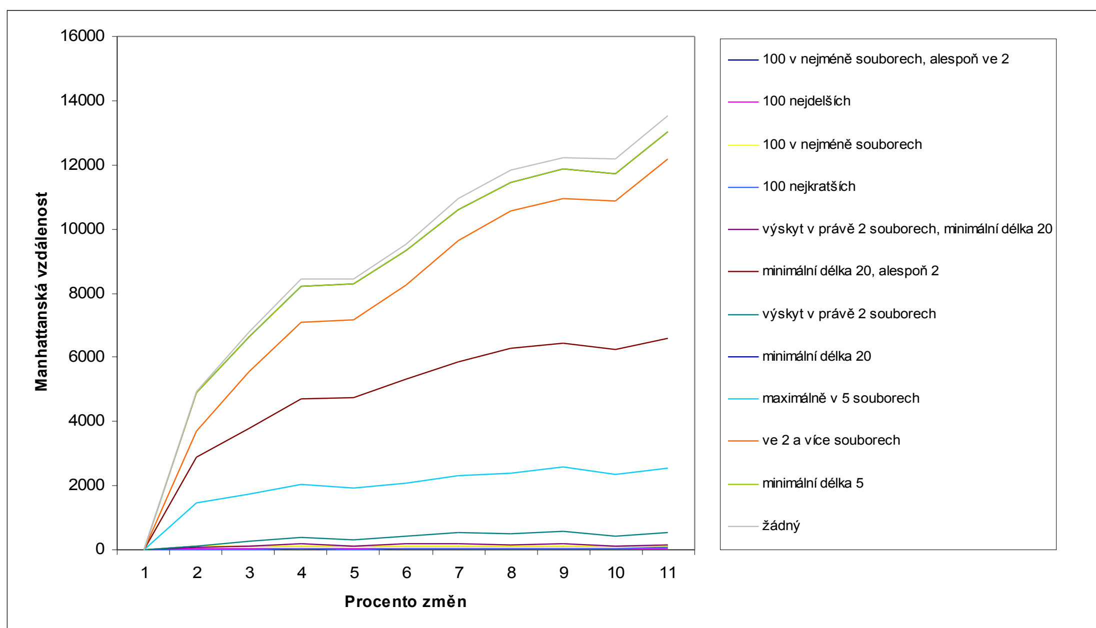
U tohoto srovnání nebudou uvedeny výsledky u všech filtrů, protože u většiny filtrů nebyla přítomnost krátkého společného bloku kódu z metriky znatelná. Zhodnocení je tedy napsáno jen u těch filtrů, s kterými bylo dosaženo dobrých výsledků, a ostatní jsou pouze uvedeny v seznamu nevhodných filtrů.

Výskyt právě ve 2 souborech Nejlepších výsledků dosáhl filtr, kterým projdou podřetězce vyskytující se právě ve dvou souborech. Důvodem je, že tímto filtrem neprojdou běžné podřetězce, které mají mnoho výskytů, ale pouze unikátní podřetězce, které jsou specifické pro dva soubory, a tím zvýší šanci odhalit plagiát na základě neobvyklého bloku kódu.

Výskyt právě ve 2 souborech, minimální délka 20 Tento filtr má přiblžně stejné výsledky jako předchozí, je však ještě lepší v tom, že více zredukuje objem dat. Rozdíl v metrice je také trochu znatelnější než u předchozího filtru, ačkoli je těžké stanovit přesně,



Obrázek 6.1: Graf - Test vlivu procentuálních změn na metriku, Eukleidova vzdálenost



Obrázek 6.2: Graf - Test vlivu procentuálních změn na metriku, Manhattanská vzdálenost

který filtr je lepší, protože náhoda při generování testovacích souborů hraje v porovnání nezanedbatelnou roli.

Minimální délka 20 Filtr, který odfiltruje všechny podřetězce kratší než 20, dosáhl poměrně dobrých výsledků. To je způsobeno tím, že je odfiltrováno velké množství nejběžnějších podřetězců, které by zastínily svým počtem výskytů unikátní podřetězec společný s originálem, který nám napovídá, že se může jednat o plagiát. Tento filtr byl vyzkoušen i v mírnější variantě - minimální délka 5, v té však dosáhl výrazně horších výsledků, protože redukce počet podřetězců jen minimálně.

Výskyt alespoň ve 2 souborech, minimální délka 20 Kombinace předchozího filtru s výskytem v minimálně dvou souborech vyšel ze srovnání jako třetí nejlepší. Dva filtry, které měly v tomto srovnání lepší výsledky, však nedopadly dobře v předchozím srovnání, takže tento filtr dopadl celkově nejlépe.

Výskyt maximálně v 5 souborech Tento filtr vykazuje poměrně dobré výsledky ze stejného důvodu jako filtr, kterým projdou podřetězce s výskytem v právě 2 souborech, rozdíl v metrice však není tolik znatelný. Zároveň ale dosahuje lepších výsledků v prvním experimentu.

Ostatní filtry U ostatních filtrů zanikne rozdíl v metrice díky velkému množství ostatních podřetězců a rozdíl metriky v grafu není znatelný. Mezi filtry, které se při srovnání neosvědčily patří:

- Bez filtru
- 100 nejktratších
- 100 nejdelších
- 100 s výskytem v nejméně souborech
- 100 s výskytem v nejméně souborech, minimálně však ve 2
- Minimální délka 5
- Výskyt alespoň ve 2 souborech

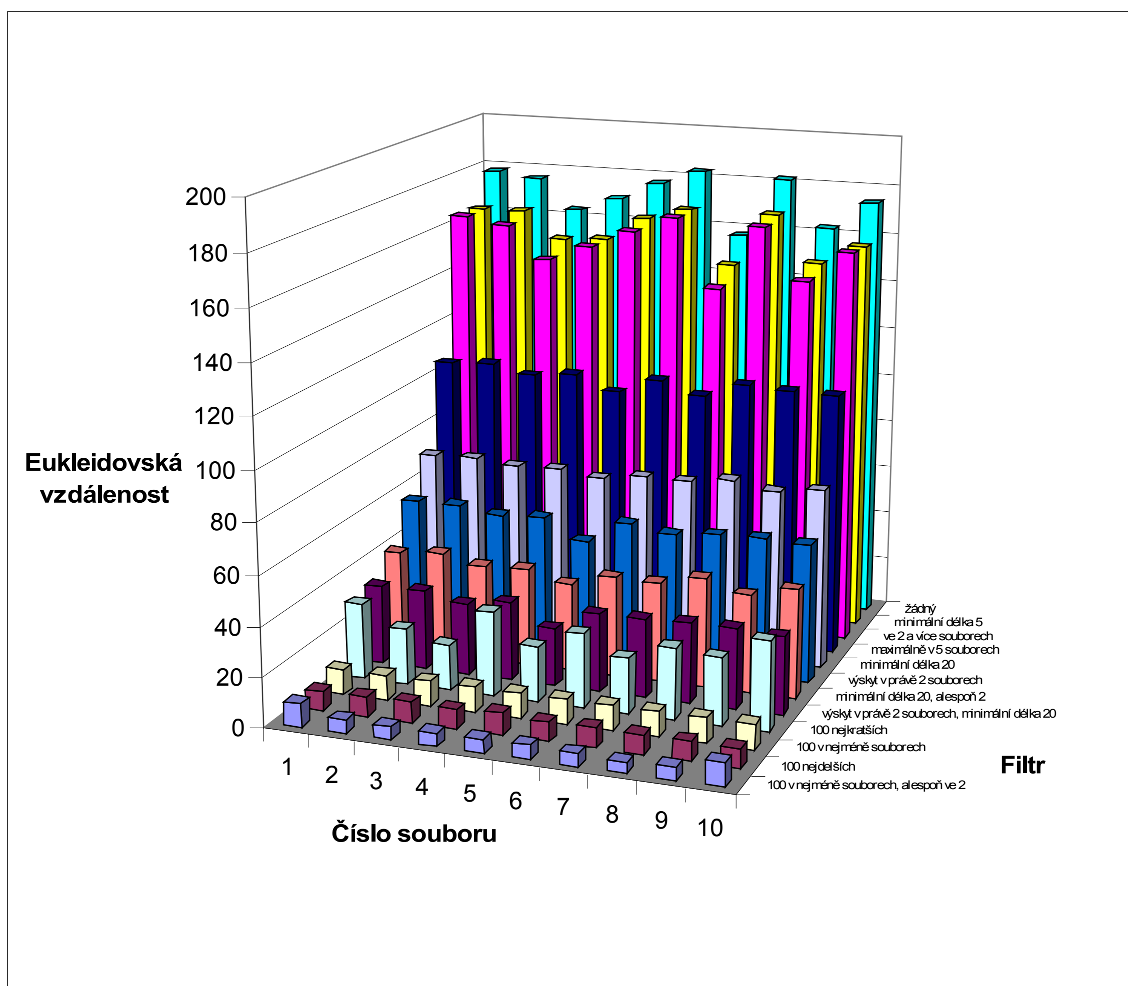
6.3.2.2 Srovnání metrik

Eukleidovská vzdálenost V grafu je u některých filtrů znát rozdíl v metrice u souboru obsahující text bloku společný s originálem oproti ostatním souborů, není však nijak závratný.

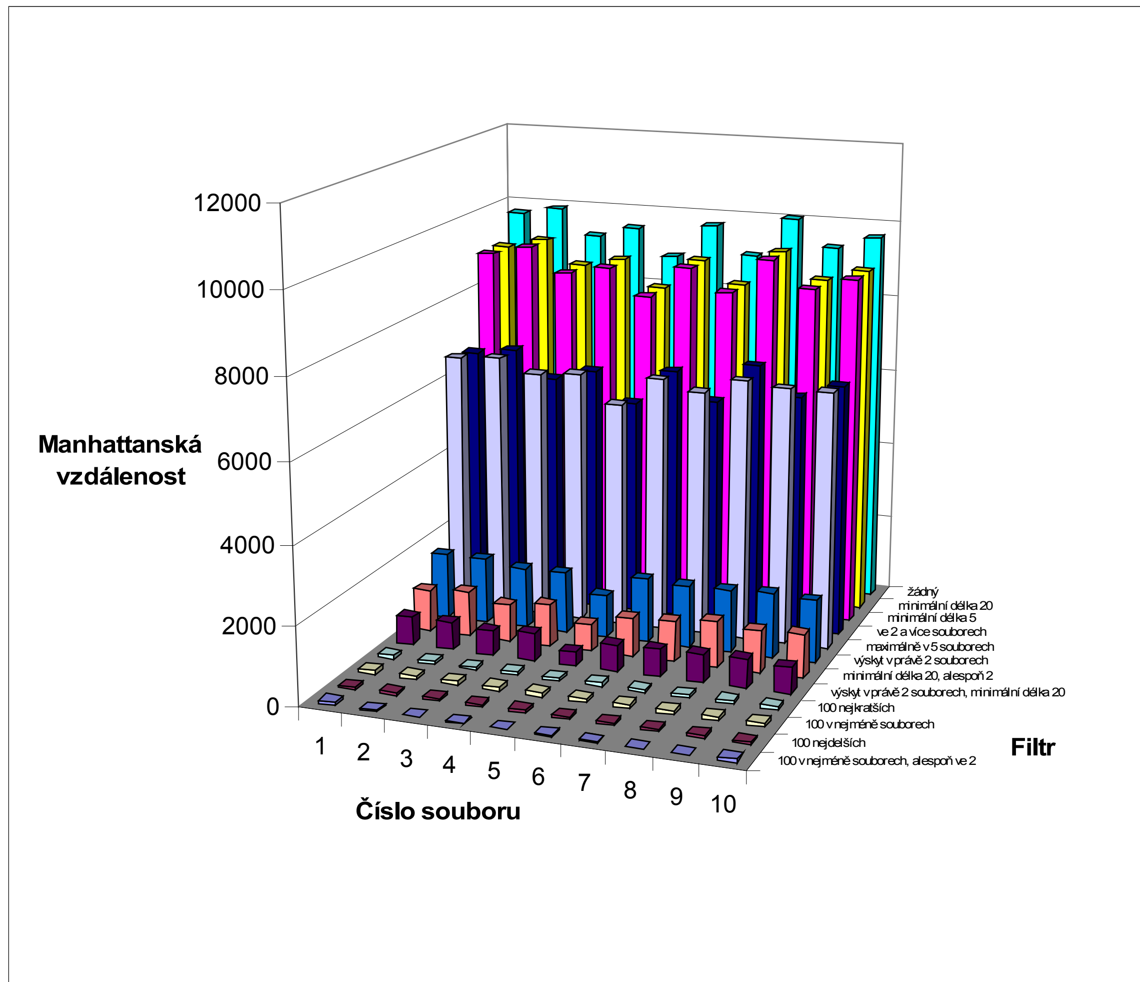
Manhattanská vzdálenost U Manhattanské vzdálenosti dosahují hodnoty metrik opět daleko extrémnějších hodnot oproti Eukleidovské vzdálenosti, i rozdíl mezi souborem obsahujícím text bloku společný s originálem a ostatními soubory je znatelnější. Z tohoto srovnání vyšla Manhattanská vzdálenost o něco lépe. Přesto nejsou celkové výsledky srovnání vzdáleností zcela jednoznačné a budou podrobeny dalšímu testování.

6.4 Zhodnocení vhodné konfigurace

Z výsledků předchozích dvou testů vyplývá, že nejvhodnější je použití filtru, kterým projdou podřetězce vyskytující se minimálně ve 2 souborech, a které mají určitou minimální délku. Tato kombinace filtrů dopadla nejlépe v prvním testu a dobře ve druhém testu. V tom měla trochu horší výsledky než filtry, kterými projdou podřetězce vyskytující se právě ve 2 souborech, tyto filtry však dosáhly poměrně špatných výsledků v prvním testu, je tedy zvolen nejvhodnější kompromis. Protože srovnání vzdáleností nedopadlo zcela jednoznačně a u exprimentálních srovnávacích testů nebyla zohledněna možnost rozdílné velikosti souborů, budou podrobeny závěrečnému testování vážené varianty obou testovaných vzdáleností.



Obrázek 6.3: Graf - Test vlivu přítomnosti společného bloku textu na metriku, Eukleidovská vzdálenost (soubor číslo 5 má s originálem společný blok textu)



Obrázek 6.4: Graf - Test vlivu přítomnosti společného bloku textu na metriku, Manhattanská vzdálenost (soubor číslo 5 má s originálem společný blok textu)

Kapitola 7

Testování

V této kapitole jsou popsány možné metody testování a zhodnocena jejich použitelnost pro ověření správnosti aplikace. Následuje zdůvodnění volby testovacích metod a popis implementace testů. Dále je zdokumentována tvorba testovací sady, průběh testování a jeho výstupy. Na závěr jsou zhodnoceny výsledky.

7.1 Volba testovacích prostředků

Vzhledem k tomu, že samotná definice plagiátu není přesně vymezená, je obtížné změřit úspěšnost aplikace jako celku. Aplikace však sestává z několika funkčních částí, jako je například konstrukce vektoru s počty výskytů, u kterých se správnost dá jednoznačně určit. Z těchto důvodů je třeba vhodně nakombinovat více testů různého typu, aby výsledky měly dostatečnou vypovídací hodnotu.

7.1.1 Unit testy

Unit testy se soustředí na testování jednotlivých jednotek kódu, které jsou testovány odděleně. Takovouto jednotkou bývá v objektově orientovaných jazycích zpravidla třída. Často jsou unit testy psány už při vývoji aplikace a jsou pravidelně spouštěny k ověření správnosti kódu. Tento přístup se nazývá Test-driven development. Zpracování dokumentu probíhá v několika fázích, každá je implementována jako třída. Pro každou z těchto tříd jsou napsány unit testy k prokázání její správné funkčnosti.

7.1.2 Systémové testy

Systémové testy se zaměřují na systém jako celek a patří do skupiny takzvaných Black-box testů. Black-box testy nepředpokládají znalost vnitřního fungování systému a testování spočívá v zadávání vstupů do systému a analýze výstupů. Tento druh testů je využit k zhodnocení celková úspěšnosti aplikace.

7.2 Implementace unit testů

Unit testy byly napsány pro každou třídu a metodu obsahující komplexnější logiku. Nebyly napsány například pro metody pro práci se soubory, konkrétně metodu vracející seznam souborů s příponou .tex a metodu, která načte text ze souboru. Při testech je kladen důraz na jednoduchost - proto jsou jako vstupy i výstupy pro porovnávání ručně zadána malá testovací data a výsledek porovnán s, rovněž jednoduchým, vzorem správného výstupu. Pokud by byly testy příliš komplikované, bylo by v případě selhání testu obtížné určit, jestli se chyba vyskytla v testované metodě, nebo v samotném unit testu. Pro každou třídu je vytvořeno 1 - 4 testů. K implementaci unit testů je využito frameworku JUnit.

7.2.1 Třída LatexLexer

Lexikální analyzátor provádí tokenizaci, správná funkce lexikálního analyzátoru použitého v aplikaci tedy je extrahovat z textu lexikální elementy mající syntaktický význam a obyčejný text ignorovat, respektive zpracovat jako generický textový token a ignorovat jeho obsah. Vstupem je vždy nezpracovaný text ve formátu LaTeX a výstupem sekvence tokenů. Za správnou lze funkci lexeru považovat, když ve výstupní posloupnosti jsou všechny tokeny jako ve vstupním souboru, ve správném pořadí a žádný nepřebývá.

7.2.1.1 Metoda tokenize

Metoda tokenize slouží k převodu vstupního souboru ve formátu LaTeX na posloupnost čísel. Číslená reprezentace se přiděluje za běhu v pořadí prvních výskytů tokenů, je tedy možné poměrně jednoduše tento převod provést ručně. Pro ověření správné funkčnosti tokenizace bylo vytvořeno pět krátkých vzorových textů a pro každý ručně vytvořena číselná posloupnost, výstup metody tokenize je porovnáván s touto posloupností. Kromě krátkých úryvků kódu je jako testovací vstup použit také obyčejný text bez LaTeXové syntaxe a prázdný řetězec.

7.2.2 Třída Alphabet

Třída alphabet slouží k převodu tokenů na číselný kód. Obsahuje jedinou metodu, getCharacter, která přiřadí tokenu na základě jeho typu a hodnoty celé číslo. Toto číslo roste s jednotkovým krokem a je přiřazováno tokenům v pořadí prvního výskytu, můžeme proto celkem jednoduše předpovědět, jaký kód bude tokenu přiřazen. Součásí třídy Alphabet jsou datové struktury, které slouží k zapamatování existujících kódů tokenů. To znamená, že dvě různá volání metody getCharacter se stejnými parametry mohou mít různé výsledky. Proto není testováno samostatné volání metody getCharacter, ale několik po sobě následujících volání.

7.2.2.1 Metoda getCharacter

Každý test tvoří posloupnost volání metody getCharacter, kde jako parametry slouží typ a hodnota tokenu. Správnost výstupu je kontrolována po každém volání.

7.2.3 Třída Nfa

U konstrukce nedeterministického konečného automatu testujeme jeho jedinou metodu, construct. Ta má za úkol na základě vstupních řetězců sestavit nedeterministický konečný faktorový automat. Vytvoření testovacích vstupů je velmi jednoduché, stačí jakákoli množina číselných posloupností, problematičtější je ověřování správnosti výsledku, neboť jde o mnohem složitější datovou strukturu než byly výstupy předchozích dvou zmíněných unit testů.

7.2.3.1 Metoda construct

Před voláním metody construct si vytvoříme testovací korpus. Ten tvoří objekt třídy corpus, do kterého jsou přidány objekty třídy FileRepresentation, kterým jsou přiřazeny ručně zadané číselné posloupnosti. Poté je zavolána metoda construct a výsledek ověřen. Ověřování je poměrně problematické, proto je záměrně zvolen menší počet kratších posloupností, jinak by byl kód unit testu moc nepřehledný a hrozilo by, že unit test bude hlásit selhání z důvodu chybného unit testu namísto samotné metody construct. Ověřování probíhá tak, že procházíme graf od počátčního uzlu a kontrolujeme znaky na přechodech a délky jednotlivých částí grafu.

7.2.4 Třída Dfa

Pro třídu Dfa byly vytvořeny unit testy, které však nakonec v projektu chybí. Důvodem jsou změny struktury aplikace, které by vyžadovaly přepsat testy a vzhledem k tomu, že algoritmus pro převod NFA na DFA nakonec není v aplikaci využit, by to byla ztráta času.

7.2.5 Třída SuffixTree

Účelem třídy SuffixTree je sestavení suffixového stromu pro množinu řetězců, kde ke každému uzlu je přiřazen počet výskytů podřetězce, který uzel představuje. Předmětem testování tedy je ověření, že se řetězce z této množiny ve stromu nacházejí a že počty výskytů souhlasí s předpokládanými hodnotami.

7.2.5.1 Metoda construct

Vstupem je objekt třídy Corpus, tedy množina vstupních dokumentů - pro každý test je tedy uměle vytvořen malý korpus - množina dokumentů v podobě posloupnosti tokenů. Výstupem metody construct je suffixový strom, ve kterém dohledáváme výskyty určitých podřetězců a srovnáváme s předpokládanými hodnotami. Součástí testu je i ověřování, jestli se řetězec v suffixovém stromu vůbec nachází. Podle způsobu ověřování správných počtů rozdělujeme testy do dvou kategorií - testy ověřující počet výskytů jednoho vzorku a testy ověřující počet výskytů všech podřetězců v korpusu. Ověření probíhá tak, že jsou pro podřetězec nejprve hrubou silou (neoptimalizovaným algoritmem) vypočítány počty výskytů a poté jsou tyto hodnoty porovnány z hodnotami vyhledanými v suffixovém stromu.

7.2.6 Třída Corpus

Třída corpus, kromě toho, že zastřešuje množinu souborů a jejich reprezentací, také obsahuje metody na konečné vyhodnocení pdobnosti plagiátů. Předmětem testování jsou metody, které vypočtou matici vzdáleností a metoda, která na základě této matice vrátí seřazenou posloupnost objektů třídy FileComparison, tedy dvojic souborů s číselnou hodnotou jejich rozdílnosti (vzdálenost 0 znamená shodné dokumenty). Dále jsou napsány testy pro všchny filtry podřetězců. Filtrů je poměrně dost, nebudou zde tedy popsány implemntace jednotlivých testů.

7.2.6.1 Metoda calculateEucleidianDistances

Testy pro metodu calculateEucleidianDistances vytvoří malý testovací korpus, které obsahuje vektory počtu výskytů představující porovnávané soubory. Výstupem metody calculateEucleidianDistances je pro každou dvojici dokumentů číslo vyjadřující vzdálenost těchto dvou dvou vektorů. Díky tomu je ověřování správnosti výsledku jednoduché, navrácená hodnota je porovnána s ručně spočítanou vzdáleností. Stený test je napsán i pro váženou variantu této metody.

7.2.6.2 Metoda calculateTaxicabDistances

Úplně stejným postupem je otestována i metoda calculateTaxicabDistances. Jediný rozdíl v postupu psaní testu je volání jiné metody pro spočítání metriky. Stený test je napsán i pro váženou variantu této metody.

7.2.6.3 Metoda getDocumentComparisons

Výstupem metod pro výpočet metrik je matice vzdáleností mezi soubory pro kombinace každého souboru s každým (s výjimkou porovnání souboru se sebou samým). Tato matice je vstupem metody getDocumentComparisons. Pro otestování si jako vstup vytvoříme testovací korpus a pro něj uměle vytvoříme v kódu testu matici vzdáleností. Předpoklad úspěšnosti testu je, že výstup, který je tvořen posloupností objektů třídy FileComparison, která obsahuje informace o dvojici souborů a jejich vzdálenost, je správně seřazen podle vzdáleností zadaných v matici.

7.3 Realizace systémových testů

Pro systémové testy je potřeba vhodná sada testovacích dat, tedy sada dokumentů simulující co nejlépe množinu odevzdaných dokumentů, včetně plagiátů, částečných plagiátů a samozřejmě originálů. Dokumenty jsou rozděleny do tříd podle úrovně původnosti od originálu až po kompletně okopírovanou práci. Systém můžeme pokládat za úspěšný, pokud dokáže dokumenty podle těchto tříd správně seřadit a nedojde k jejich pomíchání. Vstupem je název adresáře s testovacími daty, a další parametry výstupem je seznam dvojic souborů opatřených číselným vyjádřením rozdílnosti, seřazený od nejpodezřelejších k nejméně podezřelým.

7.3.1 Tvorba testovacích dat

Pro účinné zhodnocení úspěšnosti aplikace a nalezení případných nedostatků je vhodné vytvořit množinu dat, u které je možné dosáhnout měřitelných výsledků. Testovací soubory musí obsahovat různé druhy úprav textu běžně využívaných k zamaskování plagiátu. Množinu původních souborů, které jsou originaly, je potřeba doplnit o uměle vytvořené plagiáty. Toho je možné dosáhnout ručně dělanými úpravami nebo pomocí podpůrného nástroje na tvorbu plagiátů. První metoda má výhodu v tom, že úroveň plagiátu bude vyšší, při automatizovaném zpracování může dojít k vytvoření syntaktických chyb. Je však velmi pracná a u takto vytvořené množiny dat je obtížné objektivně zhodnotit výsledky. Druhá metoda má výhodu v tom, že stejné postupy jsou aplikovány na více souborů, máme tedy lepší možnost srovnání. Pracnost vytvoření testovací množiny dat také není tolik závislá na její velikosti, je tedy možné vytvořit velký korpus. Pro otestování aplikace byla zvolena kombinace obou přístupů - byla vytvořena aplikace PlagiTeX Plagiator, která umožňuje náhodné úpravy bloků textu a zároveň i ruční editaci. Předpokladem úspěchu aplikace je co nejmenší náchylnost k těmto úpravám, tedy co nejmenší metrika rozdílnosti mezi origiálem a kopií.

7.3.2 Podpůrná aplikace na tvorbu plagiátů

Účelem podpůrné aplikace je automatizovaně pozměnit zdrojový soubor tak, aby byl co nejobtížněji odhalitelný jako plagiát a zároveň byl co nejméně pozměněn jeho obsah. Aby byly výsledky co nejlépe měřitelné, je možnost zvolit si rozsah změn - měřeno v procentech dokumentu. Úpravy probíhají po blocích řádků a je možno si zvolit, kolik procent řádků bude pozměněno, a minimalní a maximální velikost bloku. Pokud jsou tedy úpavy aplikovány například na 50% dokumentu, znamená to, že upraven je v průměru každý druhý řádek.

7.3.2.1 Změny odsazení

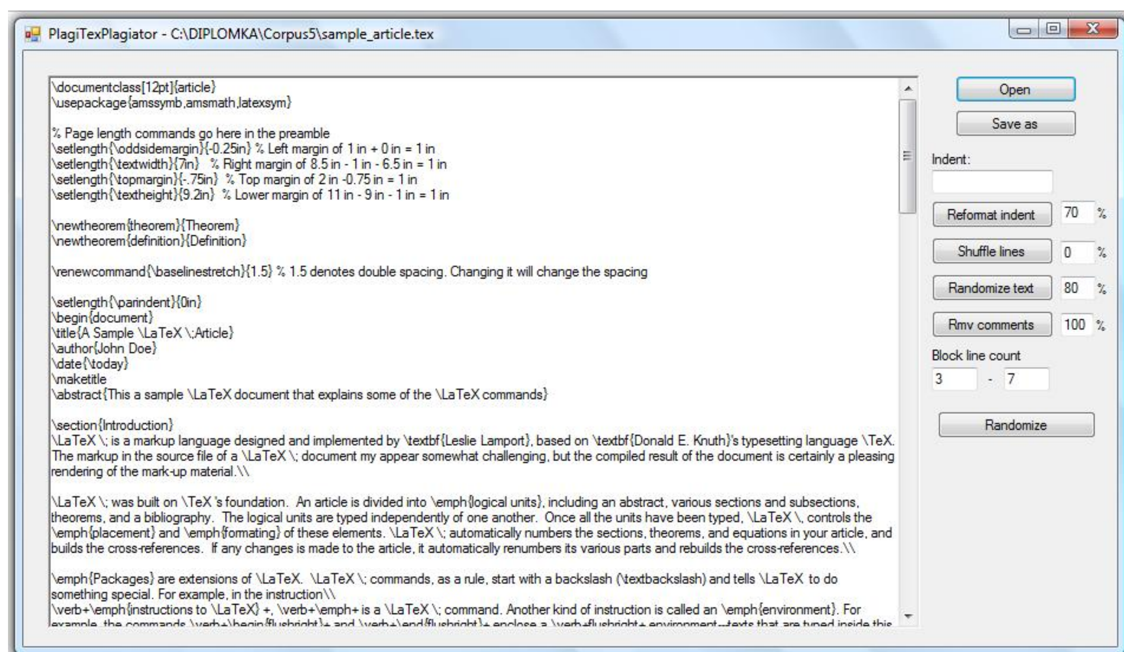
Náhodně vybraným blokům textu je přiřazeno odsazení tvořené náhodným počtem mezer.

7.3.2.2 Náhrada textu náhodnou kombinací slov

Volný text je nahrazen náhodnou kombinací slov. Vzhledem k zadání diplomové práce - hledání plagiátů v syntaxi v LaTeXu by neměl obsah textu hrát roli - důležité jsou značky. Součástí funkce je jednoduchý lexikální analyzátor, který má za úkol oddělit syntaxi od obsahu. Obsah je poté nahrazen náhodnou kombinací slov pocházející z textu Lorem ipsum - běžně využívaného modelového textu[19]. Počet slov náhodného výběru slov závisí na počtu slov původního textu.

7.3.2.3 Odstranění komentářů

Z textu jsou odstraněny komentáře, tedy konce řádků oddělené znakem %, pokud před ním není zpětné lomítko.



Obrázek 7.1: PlagiTex Plagiator - screenshot

7.3.2.4 Prohození řádků

Tato úprava způsobuje, na rozdíl od předchozích úprav, vážnější zásahy do syntaxe. V praxi je možné prohodit řádky či příkazy tak, aby se zachovala správnost a podoba dokumentu, jde to však spíše ve výjimečných příkladech. Tato automatizovaná úprava nezachovává správnost dokumentu a slouží spíše jako ukazatel náchylnosti detekčního algoritmu k drobným úpravám v syntaxi. Řádky jsou prohazovány v rámci krátkých bloků, nedochází tedy k situaci, kdy se řádek ze začátku dokumentu dostane na konec či naopak. Předpokládá se, že prohození řádků bude mít vliv na metriku podobnosti, aniž by to nutně musela být slabina algoritmu.

7.4 Testovací data

K vytvoření sady testovacích dat je použita množina souborů, které nejsou vzájemné plagiáty. U některých z těchto souborů jsou následně uměle vytvořeny plagiáty zkopírováním obsahu s různým rozsahem maskovacích úprav, Podle typu a rozsahu těchto úprav jsou testovací soubory rozděleny do tříd, které označují úroveň plagiátu. Dokumenty se dělí do následujících tříd (v pořadí od nejvíce k nejméně okopírovaných), označených písmeny A-F:

- Kopie beze změn (A)
- Kopie se změnami obsahu, ale nezměněnou strukturou (B)
- Kopie se změnami obsahu, odsazení a komentářů (C)

- Kopie s drobnými změnami strunktury - prohazováním příkazů, vkládáním redundantní příkazů a podobnými (D)
- Částečná kopie - kombinace vlastního a okopírovaného textu (E)
- Originál (F)

Je důležité vyzkoušet i citlivost algoritmu na velikost souboru - množina dokumentů obsahuje soubory o různé velikosti, krátké kvůli vyvrácení náchylnosti algoritmu k označování krátkych souborů falešně jako plagiáty vzhledem k tomu, že algoritmus je založen na počítání počtu podřetězců a kratší soubor má podřetězců méně. Zároveň je třeba zahrnout do testovacích dat i velké soubory - jedním ze základních požadavků na aplikaci je doběhnutí v krátkém čase i pro velkou množinu dat.

K otestování byly vytvořeny dva různé korpusy - jeden obsahuje naprosto rozdílné soubory (od malých po rozsáhlé), druhý test obsahuje různé verze stejného dokumentu - soubory, které nejsou plagiáty, ale mají stejný obsah a liší se jen v syntaxi. Tyto originály jsou u obou korpusů doplněny o plagiáty vytvořené pomocí aplikace PlagiTeX Plagiator. Soubory jsou pojmenovány číslem a v případě plagiátu i písmenem označujícím třídu plagiátu (například soubor B1.tex vznikl ze souboru 1.tex změnami obsahu textu). Od úspěšného textu se očekává, že setřídí dvojice porovnávaných souborů tak, aby nejdříve byly originály s plagiáty co nejnižší třídy. V případě, že od jednoho dokumentu existuje více plagiátů, mohou být za vzájemné plagiáty označeny i soubory, které vznikly jako plagiáty jednoho společného souboru, aniž by byl jeden vytvořen editací druhého.

V kapitole Experimenty jsem došel k závěru, že nejlepší je kombinace filtru, kterým projdou podřetězce vyskytující se alespoň ve dvou souborech, a filtru kterým projdou podřetězce od délky 20. Minimální délka však byla v testech snížena s přihlédnutím k velikostem testovacích souborů (Při experimentech s filtry se pracovalo se soubory většího rozsahu - bakalářská práce, zatímco v testovacích datech pracujeme i s velmi malými soubory). Pro test na množině úplně rozdílných souborů byla stanovena minimální délka 3, pro test na množině různých souborů se stejným obsahem je minimální délka 5. Jako metrika byla použita vážená Eukleidovská vzdálenost a vážená Manhattanská vzdálenst (kvůli nejednoznačnostem ve srovnání v kapitole Experimenty), vzhledem k tomu, že první dopadla v testech výrazně lépe, jsou zde uvedeny pouze výsledky pro váženou Eukleidovkou vzdálenost.

7.5 Výstupy testů

Výstupem jsou všechny dvojice souborů seřazené podle metriky vzdálenosti, což je velké množství dat. Aby bylo možné názorně data zobrazit, byly použity dva různé způsoby zobrazení výsledků. Prvním je tabulka dvojic originál - plagiát, kde je u každé dvojice souborů uvedena vzdálenost a pořadí v seřazené posloupnosti dvojic. Dalším výstupem je graf znázorňující metriky rozdílnosti jednoho vybraného souboru v porovnání se všemi ostatními. K tomuto souboru je za tímto účelem vytvořen jeden plagiát od každé třídy. U tabulky si je třeba u uvedených pořadí uvědomit, že na pořadí dvojic souborů se projeví skutečnost,

Kategorie plagiátu	Originál	Plagiát	Pořadí	Vzdálenost
Kopie beze změn	1.tex	A1.tex	1 / 528	0
	3.tex	A3.tex	3 / 528	0
	7.text	A7.tex	4 / 528	0
	11.tex	A11.tex	2/ 528	0
Změněný pouze obsah	1.tex	B1.tex	6 / 528	0
	5.tex	B5.tex	9 / 528	0
	7.tex	B7.tex	10 / 528	0
	14.tex	B14.tex	8 / 528	0
Změněný obsah, odsazení a komentáře	1.tex	C1.tex	13 / 528	0
	10.tex	C10.tex	15 / 528	0
	11.tex	C11.tex	16 / 528	0
	15.tex	C15.tex	18 / 528	0
Změněná syntaxe	1.tex	D1.tex	26 / 528	0,0555
	4.tex	D4.tex	21 / 528	0,036
	6.tex	D6.tex	19 / 528	0,0195
	11.tex	D11.tex	22 / 528	0,0489
Změněná syntaxe	1.tex	E1.tex	37 / 528	0,0735
	8.tex	E8.tex	29 / 528	0,0623
	14.tex	E14.tex	71 / 528	0,0997

Tabulka 7.1: Výsledky testu na množině úplně rozdílných souborů

že pokud existuje k jednomu originálu více plagiátů, tyto plagiáty se umístí vysoko, jako by byly vzájemnými plagiáty. Proto jsou v tabulce mezi dvojicemi souborů s nulovými vzdálenostmi chybějící místa - nejde o chybu, tato místa jsou obsazena dvojicemi plagiátů majícími společný originál.

7.5.1 Test na množině úplně rozdílných souborů

Ve srovnání metrik vzdáleností vybraného souboru s jeho plagiáty dopadl první test dobře. Pořadí souborů seřazených podle vzdálenosti od vybraného souboru odpovídá pořadí tříd plagiátu, mezi plagiáty různých tříd jsou patrné rozdíly ve vzdálenosti od originálu. Soubory, které nejsou plagiátem vybraného souboru, se umístily v pořadí za plagiáty, a to s výraznou rezervou u většiny souborů. Ve srovnání všech dvojic souborů dopadlo perfektně srovnání originálů s plagiáty třídy A-C. Srovnání originálů s plagiáty třídy D dopadlo dobře, jsou tu však již znát rozdíly - to je v pořádku, neboť syntaxe byla pozměněna. Hůře dopadly částečné plagiáty - původní soubory doplněné blokem textu z jiného souboru. Většina se v pořadí souborů umístila v prvních 10%, jen jeden částečný plagiát skončil v porovnání s originálem na poměrně špatném umístění - 71. z 528.

7.5.2 Test na množině různých souborů se stejným obsahem

Ve srovnání metrik vzdáleností vybraného souboru s jeho plagiáty a ostatními soubory dopadl tento test o něco hůře než ten předchozí, stále však byly soubory seřazeny správně

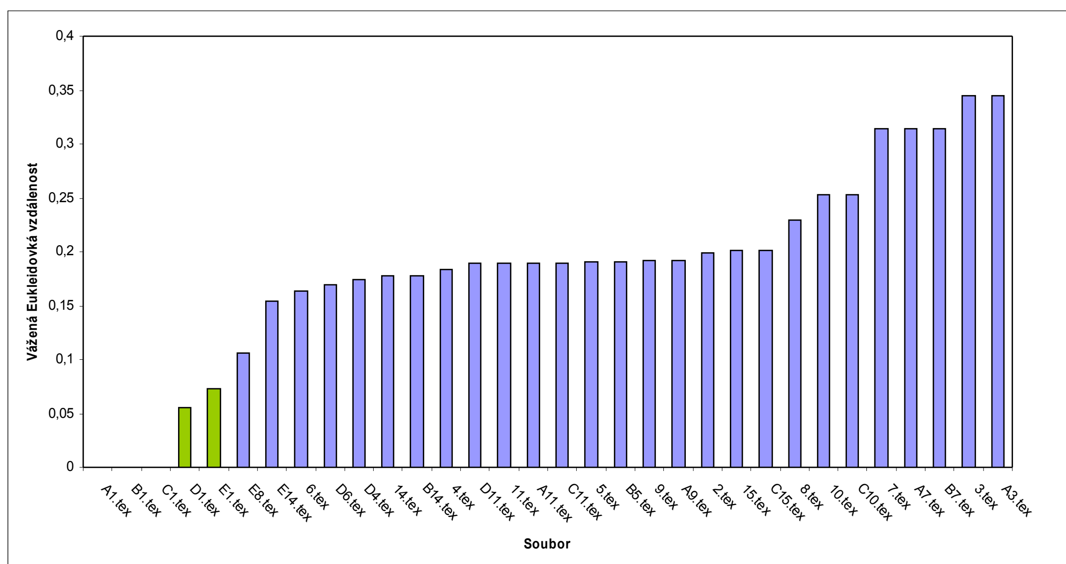
Kategorie plagiátu	Originál	Plagiát	Pořadí	Vzdálenost
Kopie beze změn	1.tex	A1.tex	1 / 780	0
	10.tex	A10.tex	2 / 780	0
	15.text	A15.tex	3 / 780	0
	23.tex	A23.tex	4 / 780	0
Změněný pouze obsah	1.tex	B1.tex	5 / 780	0
	2.tex	B2.tex	8 / 780	0
	8.tex	B8.tex	9 / 780	0
	19.tex	B19.tex	7 / 780	0
Změněný obsah, odsazení a komentáře	1.tex	C1.tex	10 / 780	0
	5.tex	C5.tex	14 / 780	0
	7.tex	C7.tex	15 / 780	0
	16.tex	C16.tex	13 / 780	0
Změněná syntaxe	1.tex	D1.tex	89 / 780	0,0275
	3.tex	D3.tex	33 / 780	0,0236
	6.tex	D6.tex	28 / 780	0,0213
	16.tex	D16.tex	21 / 780	0,0179
Změněná syntaxe	1.tex	E1.tex	110 / 780	0,0286
	12.tex	E12.tex	61 / 780	0,0264
	17.tex	E17.tex	30 / 780	0,0227

Tabulka 7.2: Výsledky testu na množině různých souborů se stejným obsahem

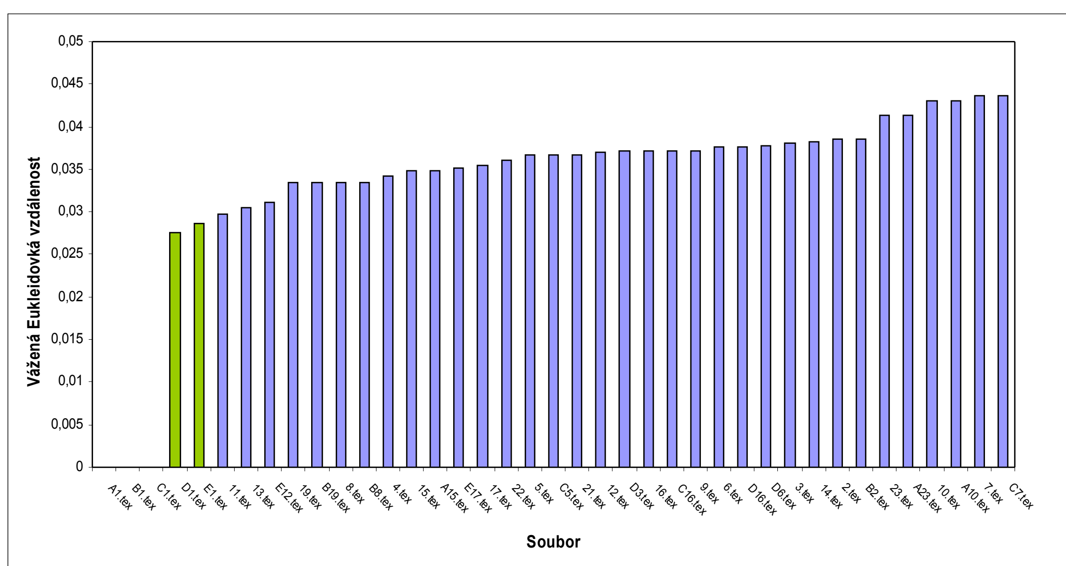
podle podobnosti tak, že plagiáty nižší třídy byly dříve než plagiáty vyšší třídy a nedošlo k výraznějšímu promíchání se soubory, které nejsou plagiátem vybraného souboru. Ve srovnání všech dvojic souborů dopadlo perfektně srovnání originálů s plagiáty třídy A-C. Od třídy D výš došlo místy k promíchání plagiátů s originály. To je způsobeno tím, že soubory v testovacím korpusu jsou si v mnoha případech velmi podobné, aniž by se jednalo o plagiáty. Největší slabinou jsou opět částečné plagiáty - původní soubory doplněné blokem textu z jiného souboru. Ty se všechny nacházejí v pořadí souborů v prvních 10%, jsou však od prvních příček pořadí dvojic souborů o něco vzdálenější než plagiáty nižších tříd.

7.6 Zhodnocení

Testy dopadly výborně pro plagiáty, u kterých nebyla pozměněna syntaxe, pouze obsah, odsazení a komentáře. U těchto souborů byla metrika vzdálenosti nulová, soubory byly tedy označeny jako totožné. Dobrých výsledků jsme dosáhli u plagiátů, kde byla pozměněna syntaxe. Největší slabinou byly částečné plagiáty, tedy soubory, které jsou z většiny originály, ale obsahují společnou část textu (nebo více společných částí) pocházející z jiného souboru. Podle očekávání dopadly lépe testy nad množinou zcela rozdílných souborů než testy nad množinou různých souborů se stejným obsahem. Unit testy proběhly se stoprocentní úspěšností.



Obrázek 7.2: Graf výsledků - Test na množině úplně rozdílných souborů (plagiáty jsou barevně odlišeny)



Obrázek 7.3: Graf výsledků - Test na množině různých souborů se stejným obsahem (plagiáty jsou barevně odlišeny)

Kapitola 8

Závěr

Výsledkem práce je kromě samotné aplikace PlagiTeX také rozbor existujících řešení a bylo realizováno mnoho postupů, které byly podrobeny experimentům a testování. Některé postupy byly již ve fázi rozboru existujících řešení označeny za nevhodné k použití, jiné byly implementovány, ale od jejich využití bylo upuštěno na základě srovnávacích testů. Stalo se tak buďto kvůli nepříznivým výstupům testů (některé filtry), nebo přílišné výpočetní náročnosti (konstrukce DFA). Kromě funkční aplikace na detekci plagiátů v souborech ve formátu LaTeX poskytuje tato práce také přehled o problematice detekce plagiátů v LaTeXovém kódu.

Literatura

[1] Michael Brennan, Sadia Afroz, Rachel Greenstadt, Adversarial Stylometry: Circumventing Authorship Recognition to Preserve Privacy and Anonymity
https://www.cs.drexel.edu/~sa499/papers/adversarial_stylometry.pdf

[2] Lexical Analysis (Tokenizing)
http://cg.scs.carleton.ca/~morin/teaching/3002/notes/lex.pdf

[3] Samarjit Chakraborty, Formal Languages and Automata Theory - Regular Expressions and Finite Automata
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.9977&rep=rep1&type=pdf

[4] Winnowing: Local Algorithms for Document Fingerprinting
http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf

[5] Sergey Brin, James Davis, Hector Garcia-Molina, Copy Detection Mechanisms for Digital Documents
http://diglib.stanford.edu:8091/diglib/pub/reports/copy-detect.pdf

[6] Antonio Si, Hong Va Leong, Rynson W. H. Lan, CHECK: A Document Plagiarism Detection System
http://www.cs.cityu.edu.hk/~rynson/papers/sac97.pdf

[7] Bořrivoj Melichar, Jan Holub, Tomáš Polcar, Text searching algorithms
http://www.stringology.org/athens/TextSearchingAlgorithms/tsa-lectures-1.pdf

[8] Greg Plaxton, String Matching: Rabin-Karp Algorithm
http://www.cs.utexas.edu/~plaxton/c/337/05f/slides/StringMatching-1.pdf

[9] G. Manacher, S. I. Graham, A Fast String Searching Alorithm
http://www.cs.utexas.edu/users/moore/publications/fstrpos.pdf

[10] Introduction to String Matching
http://www.cs.ubc.ca/~hoos/cpsc445/Handouts/kmp.pdf

[11] Esko Ukkonen, On–line construction of suffix trees
https://www.cs.helsinki.fi/u/ukkonen/SuffixT1withFigs.pdf

[12] Suffix Trees
http://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/suffixtrees.pdf

[13] Suffix Trees and its Construction
http://www.cs.duke.edu/courses/fall12/compsci260/resources/suffix.trees.in.detail.pdf

[14] NFA with epsilon transitions
http://web.cecs.pdx.edu/~sheard/course/CS581/notes/NfaEpsilonDefined.pdf

[15] Variance-Covariance Matrix
http://stattrek.com/matrix-algebra/covariance-matrix.aspxc

[16] NFA → DFA Subset Construction
http://www.idt.mdh.se/kurser/cd5560/10_01/examination/examination/NFA-DFA.pdf

[17] Kranthi Kumar Mandumula, Knuth-Morris-Pratt Algorithm
http://cs.indstate.edu/~kmandumula/presentation.pdf

[18] Class Matcher
http://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html

[19] Lorem Ipsum
http://www.lipsum.com/

Kapitola 9

Seznam použitých zkratek

DOM Document Object Model

NFA Non-deterministic Finite Automaton

DFA Deterministic Finite Automaton

Příloha A

Uživatelská příručka

A.1 Formát vstupu

Vstup je v podobě parametrů příkazové řádky. Jedinný povinný parametr je cesta k adresáři se vstupními soubory. Z toho se načtou všechny soubory s příponou .tex. Další parametry jsou nepovinné. Pomocí nich je možné zvolit konfiguraci algoritmu a formát výstupu.

A.1.1 Volba filtrů

Filtr lze zvolit parametrem příkazové řádky označujícím kód filtru následovaným dalším parametrem s číselnou hodnotou, která je ve filtru použita (Například pokud chceme filtr 100 nejkratších podřetězců, tak parametry budou „-s 100“). Je možné kombinovat více filtrů, filtry se provádí ve stejném pořadí, v jakém jdou po sobě parametry při volání aplikace. Kódy filtrů jsou následující:

- Filtr na základě počtu souborů, ve kterých se vyskytuje - podřetězec se musí vyskytovat alespoň v počtu souborů daném vstupním parametrem: -mnfc
- Filtr na základě počtu souborů, ve kterých se vyskytuje - podřetězec se musí vyskytovat nanejvýš v počtu souborů daném vstupním parametrem: -mxfc
- Filtr na základě minimální délky podřetězce: -ml
- Filtr, který vybere ze všech podřetězců n nejkratších: -s
- Filtr, který vybere ze všech podřetězců n nejdelších: -l
- Filtr, který vybere ze všech podřetězců n s výskyty v nejméně souborech: -snf

A.1.2 Volba metriky

Metriku lze na rozdíl od filtru zvolit jen jednu. Volba se opět provádí parametrem příkazové řádky. Pokud je více parametrů s volbou metriky, platí ten poslední. Kódy metrik jsou následující:

- Eukleidovská vzdálenost: -e
- Taxikářská (Manhattanská) vzdálenost: -t
- Vážená Eukleidovská vzdálenost: -we
- Vážená Taxikářská (Manhattanská) vzdálenost: -wt

A.1.3 Volba formátu výstupu

Je možné zvolit výstupní soubor pomocí parametru -o následovaného názvem výstupního souboru. Pokud není zvolen výstupní soubor, výstup je vypsán na standardní výstup. Ze všech možných dvojic souborů je možné omezit výpis pouze na porovnání s jedním vybraným souborem. Tento soubor můžeme zvolit parametrem -b následovaným názvem souboru.

A.1.4 Příklad vstupu

Následující příklad vstupu nastaví cestu ke korpusu na

```
C:\Corpus
```

, filtr tak, že jím projdou podřetězce vyskytující se alespoň ve 2 souborech a s délkou alespoň 3 a spočítá váženou Eukleidovskou vzdálenost jako metriku. Vypíše všechny porovnání se souborem 1.tex do souboru output.csv.

```
C:\Corpus\ -mnfc 2 -ml 3 -we -b 1.tex -o output.csv
```

A.2 Formát výstupu

Výstup je ve formátu CSV, tedy textový soubor obsahuící jednoduchou tabulku, kde jsou sloupce odděleny středníky a řádky odděleny znakem pro nový řádek. Formát CSV se běžně používá pro export dat a lze ho otevřít například v aplikaci Microsoft Excel. Příklad takového výstupního souboru:

```
Distance;File 1;File 2;
0,0000;C7.tex;7.tex;
0,0147;9.tex;21.tex;
0,0189;7.tex;15.tex;
0,0189;A15.tex;7.tex;
0,0189;C7.tex;15.tex;
0,0189;C7.tex;A15.tex;
```

Pokud není zadán název výstupního souboru, je vstup vypsán na standardní vstup.

Příloha B

Obsah přiloženého CD

CD obsahuje následující adresáře:

- PlagiTex – aplikace na detekci plagiátů
- Test – testovací data, včetně aplikace PlagiTex Plagiator, dávkových souborů na spouštění testů a souborů s výstupy
- Doc – text dokumetace ve formátu PDF, zdrojové soubory dokumentace a obrázky použité v dokumentaci