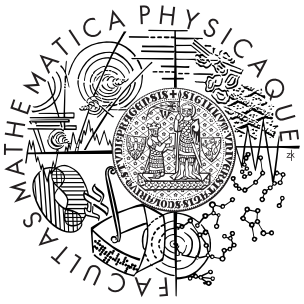
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



František Vodsloň

Vyhodnocování podobnosti zdrojových textů

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Tomáš Holan, Ph.D.
Studijní program: Informatika, obor programování

2007

Děkuji RNDr. Tomáši Holanovi, Ph.D. za poskytnutý konzultační čas a podnětné připomínky k mojí práci.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 30. května 2007 František Vodsloň

Obsah

Název práce: Vyhodnocování podobnosti zdrojových textů
Autor: František Vodsloň
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí bakalářské práce: RNDr. Tomáš Holan, Ph.D.
e-mail vedoucího: tomas.holan@mff.cuni.cz

Abstrakt: Cílem této práce je vytvořit aplikaci pro vyhodnocování podobnosti zdrojových textů. Účelem systému bude napomoci uživateli v hledání plagiátů v dané množině zdrojových textů. Součástí aplikace bude také databáze nahraných zdrojových textů. Systém bude umožňovat uživateli nahrávat zdrojové texty do databáze a porovnávat je mezi sebou podle různých kritérií. Aplikace by měla být využívaná zejména pedagogy, kterým přinese efektivní nástroj pro usnadnění hledání plagiátů mezi zadanými úkoly nebo zápočtovými programy.

Klíčová slova: porovnávání, zdrojový kód, plagiát, podobnost

Title: Source codes similarity evaluation
Author: František Vodsloň
Department: Department of Software and Computer Science Education
Supervisor: RNDr. Tomáš Holan, Ph.D.
Supervisor's e-mail address: tomas.holan@mff.cuni.cz

Abstract: Purpose of this work is to create an application for source codes similarity measuring. The main purpose of this software is to help user with searching for plagiarism in set of source files. The application will also contain database of uploaded source files. User will be able to upload source files into database and compare them with each other, using various criterions and settings. The application is intended mainly for teachers and lectors, who can use it as an useful tool for simplification of plagiarism detection in set of homeworks or semestral works.

Keywords: comparison, source code, plagiarism, similarity

Kapitola 1

Úvod

Cílem této práce je navrhnout a implementovat informační systém, který bude porovnávat zdrojové texty. Účelem systému bude rozpoznat, zda se v dané množině zdrojových textů vyskytují plagiáty (zdrojové texty odvozené jeden od druhého). Součástí aplikace bude také databáze zdrojových textů. Systém bude umožňovat porovnávat zdrojové texty mezi sebou podle různých kritérií, nahrávat zdrojové texty do databáze, spouštět hromadná porovnávání v rámci jedné kolekce[1]. Aplikace by měla být využívaná zejména pedagogy, kterým přinese efektivní nástroj pro usnadnění hledání plagiátů mezi zadanými úkoly nebo zápočtovými programy.

1.1 Koncepce systému

Systém je koncipován jako webová aplikace a skládá se ze dvou různých částí — porovnávacího programu a nadstavby, která tvoří zároveň uživatelské rozhraní programu. Tyto dvě části jsou na sobě téměř nezávislé, proto se jim budu ve své práci věnovat zvlášť. Porovnávací program je spustitelná aplikace, která porovnává vždy dva zdrojové texty. Ze standardního vstupu čte instrukce, které spouštějí dílčí porovnávací metody, a na konci běhu vydá jedno číslo — výsledek porovnávání. Naproti tomu nadstavba aplikace (uživatelské rozhraní) je tvořena sadou PHP skriptů určených pro interpretaci webovým serverem. Nadstavba obstarárá kromě uživatelského rozhraní kompletní komunikaci s databází (od načítání zdrojových textů po ukládání výsledků porovnávání do databáze) a k samotnému porovnávání dvojice souborů používá externí volání porovnávacího programu.

Tato nezávislost porovnávacího programu a uživatelského rozhraní má oproti integrovanému řešení několik výhod:

- Obě části mohou být vyvíjeny nezávisle na sobě (pouze je nutno dodržet rozhraní porovnávacího programu a nadstavby).

[1] kolekce je množina zdrojových textů, které spolu souvisí, například zápočtové programy z Pascalu, úkoly ze cvičení, atd.

- Porovnávací program bude jakožto zkompilovaný sputitelný program rychlejší, než kdyby byl napsán v jazyce PHP a interpretován webovým prohlížečem.
- PHP rozhraní umožní snadnou integraci aplikace do jiných informačních systémů založených na stejné platfomě, například do nedávno vzniklého systému na správu úkolů a zápočtových úloh *CodEx*.

Naopak mezi nevýhody tohoto řešení patří:

- Komunikace obou částí systému (předávání vstupních a výstupních dat, řešení vyjímek a chybových stavů) je náročnější než u integrovaného řešení.
- Instalace systému je složitější (lze vyřešit instalačním skriptem).

Kapitola 2

Porovnávání zdrojových textů

V této kapitole se budu zabývat obecným problémem, jak porovnat dva zdrojové texty za účelem zjištění, zda není jeden odvozen od druhého. Základním úskalím tohoto problému je samotná definice této „míry odvozenosti“ a dále určení hranice, kdy porovnávané texty označíme za plagiáty. V případě, že si plagiátor dá velkou práci, může být kopie tak zdařilá, že se v podstatě jedná o jiný kód, používající z originálu třeba jen několik málo (i když klíčových) prvků. Naopak nezřídka mohou dva autoři řešící stejný problém nezávisle na sobě vytvořit velmi podobný kód (zvláště u menších úloh). Zaměříme se proto pouze na způsob, jak co nejlépe určit míru podobnosti dvou zdrojových textů s ohledem na možné plagiátorství. Vzhledem k následnému využití programu pro hledání plagiátů v množině zdrojových textů bude tato míra podobnosti vodítkem pro uživatele systému — bude se zajímat o dvojice textů s nejvyšším koeficientem podobnosti.

2.1 Předzpracování zdrojových textů

Jedněmi z nejčastěji používaných technik plagiátorů jsou změny formátování zdrojového textu (odsazování po blocích, prázdné řádky) a změny v komentářích. Takové změny nemají žádný vliv na výsledný program, jsou pro překladač nezajímavé[1], ale dokáží poměrně úspěšně změnit vzhled zdrojového textu tak, že při vizuálním porovnávání skryjí případnou podobnost. Proto je vhodné před samotným porovnáváním odstranit z textů komentáře a rozdíly ve formátovnání. Zároveň bychom mohli využít specifických vlastností zdrojových textů, které je odlišují od běžných textových dokumentů. Programovací jazyky mají svoji pevně danou syntax a zdrojové texty jsou tvořeny lexikálními elementy jakyka. Pro následná porovnávání bude zajisté mnohem efektivnější nahlížet na zdrojové texty jako na řetězec lexikálních elementů jazyka než jako na pouhý text.

Z výše uvedených důvodů vyplývá, že by bylo vhodné porovnávané texty předzpracovat, provést jejich lexikální analýzu. Výsledkem této analýzy bude řetězec tokenů[2], oproštěný

[1] existují programovací jazyky, jejichž syntax závisí na formátování, těmi se však v práci nezabývám.

[2] tokeny jsou vnitřní reprezentace lexikálních elementů jazyka - každý lexikální element odpovídá jednomu tokenu. Příklady lexikálních elementů jsou identifikátor, celé číslo, středník, klíčové slovo „for“.

od komentářů a formátování. Dalším nesporným přínosem této analýzy bude sjednocení vstupu pro dílčí porovnávácí metody. Porovnávací metody pracující s řetězcem tokenů budou nezávislé na programovacím jazyce původních zdrojových textů a přidání podpory nového jazyka bude poměrně snadné — stačí nadefinovat množinu tokenů daného jazyka a sestrojit jeho lexikální parser.

V průběhu lexikální analýzy zároveň vytvářím symbolickou tabulku identifikátorů. Ke každému výskytu tokenu „identifikátor“ je přiřazen index do symbolické tabulky, kde je uložen název identifikátoru (různé výskyty jednoho identifikátoru mají přiřazen stejný index). Symbolická tabulka je prozatím využívána ke zjištění počtu různých identifikátorů vyskytujících se ve zdrojovém textu. Toto číslo tvoří zajímavou charakteristiku zdrojových textů, která se dá porovnávat (viz níže).

2.2 Porovnávací metody

V této kapitole přinesu obecný popis podporovaných porovnávacích metod a zmíním také metody, které byly v průběhu vývoje programu naimplementovány, avšak nakonec byly vyřazeny pro jejich nízkou vypovídací hodnotu. Výsledkem každé metody je číslo v intervalu $< 0, 1 >$, přičemž 0 znamená žádnou podobnost, naopak 1 znamená úplnou shodu. Výsledky jednotlivých dílčích porovnání jsou průběžně zaznamenávány a na konci běhu programu jsou využity ke spočítání výsledné míry podobnosti.

Porovnávací metody se dají v zásadě rozdělit na dva typy:

První typ metod nejprve odděleně spočítá určitou charakteristiku obou zdrojových textů a následně tyto dvě veličiny porovná. Výsledek metody je dán vzorcem

$$\frac{\min(a, b)}{\max(a, b)} \tag{2.1}$$

kde a, b jsou spočítané veličiny (charakteristiky) souborů. Do této skupiny metod patří porovnávání počtů výskytu tokenů, porovnávání směrodatných odchylek výskytu tokenů a porovnávání počtů různých identifikátorů.

Druhý typ metod tvoří řetězcové metriky. Řetězcové metriky jsou metody, které různými způsoby počítají vzdálenost dvou řetězců (v našem případě se jedná o řetězce tokenů). Spočítaná vzdálenost je znormalizována vzhledem k délce řetězců, čímž dostaneme požadovanou míru podobnosti v intervalu $< 0, 1 >$. Do této skupiny metod patří levenshteinova vzdálenost a nejdelší společný podřetězec.

V následujících úvahách o složitosti metod budou proměnné m, n představovat délky řetězců tokenů porovnávaných zdrojových textů.

2.2.1 Levenshteinova vzdálenost

Tato metrika udává nejmenší počet znakových operací potřebných ke konverzi jednoho řetězce na druhý. Znakové operace jsou vložení, vymazání a substituce tokenu. Míra podobnosti počítaná pomocí této metriky má dobrou vypovídací hodnotu. Zjednodušeně se

dá říci, že tato metoda spočítá počet kroků, které plagiátor vykonal při úpravách kódu — toto „odvozování kódu“ se typicky skládá z dílčích kroků vkládání, mazání a substituce příkazů (resp. tokenů), což je přesně to, na co se tato metrika zaměřuje.

Nevýhodou této metody je její časová složitost, dá se implementovat nejlépe v čase $O(m \times n)$ (metodou dynamického programování, viz oddíl 4.1.2 na straně 17). Z toho důvodu je u této metody zavedena možnost počítat pouze přibližnou editační vzdálenost. Nepovinný parametr u udává vzdálenost „výhledu“ metody, algoritmus poté nebere v potaz vkládání nebo mazání velikého počtu (více než u) tokenů na jednom místě řetězce. Pro většinu řetězců je výsledek tohoto ořezávání velmi podobný přesně spočítané vzdálenosti a časová složitost tohoto algoritmu s ořezáváním je $O(u \times \max(m, n))$, kde u je výše uvedená konstanta.

2.2.2 Nejdelší společný podřetězec

Tato metoda počítá délku nejdelšího společného podřetězce dvou vstupních řetězců tokenů. Čím je nalezený podřetězec delší (vzhledem k délce vstupních řetězců), tím jsou si vstupní soubory podobnější. Vypovídací hodnota této metody je podobně jako u předchozí metody velmi dobrá. Časová složitost metody je $O(m \times n)$.

2.2.3 Porovnávání počtu výskytů tokenů

Tato metoda spočítá výskyty daného tokenu v obou zdrojových textech a následně tato dvě čísla porovná (dle vzorce 2.1). Povinný parametr metody určuje token, jehož výskyt se porovnává. Lze porovnávat také předdefinované skupiny tokenů — konkrétně počty klíčových slov, cyklů, řídících konstrukcí jazyka nebo všech tokenů. Jelikož se prochází oba řetězce[3], časová složitost této metody je $O(\max(m, n))$.

Toto porovnání má také poměrně dobré výsledky, zvláště při uvážlivé volbě zkoumaného tokenu. V odvozených zdrojových textech se většinou příliš neliší výskyt podmínek, cyklů či identifikátorů. Nevýhodou je že tato metoda může vykazovat velikou podobnost i u textů, které spolu vůbec nesouvisí, pouze je v nich náhodou stejný počet určitého tokenu. Zároveň je spolehlivost této metody přímo úměrná velikosti porovnávaných textů — u dvou velmi malých programů obsahujících jeden či dva cykly nemá smysl tuto veličinu porovnávat. Přesto je tato metoda dobře použitelná u specifických případů porovnávání (viz. oddíl 2.3).

2.2.4 Porovnávání počtu různých identifikátorů

Tato metoda porovnává počet použitých identifikátorů ve zdrojových textech. Rozdíl mezi touto metodou a předešlou metodou s parametrem „identifikátor“ je v tom, že předešlá metoda porovnává počty *výskytů* tokenu „identifikátor“, zatímco tato metoda porovnává

[3]vyjímku tvoří porovnání počtů všech tokenů — tento počet je k dispozici jakožto součást výstupu prvotní analýzy souborů, proto porovnání má složitost $O(1)$

počty *různých* identifikátorů. Díky tomu, že počet různých identifikátorů je spočítán již při prvotní analýze zdrojového textu (jedná se o počet prvků symbolické tabulky identifikátorů), pracuje tato metoda v konstantním čase. Spolehlivost této metody je podobná jako u porovnávání počtu výskytů tokenů.

2.2.5 Směrodatná odchylka výskytu tokenů

Buď X veličina nabývající hodnot udávajících vzdálenosti[4] dvou sousedících *výskytů* daného tokenu (resp. skupiny tokenů) ve zdrojovém textu. Tato metoda porovnává směrodatnou odchylku veličiny X se směrodatnou odchylkou stejné veličiny u druhého zdrojového textu. Volně přeloženo porovnává „pravidelnost“ výskytu daného tokenu (resp. skupiny tokenů). Pokud je token v obou zdrojových textech stejně „(ne)pravidelně rozmístěn“, metoda vrací velikou míru podobnosti, naopak pokud rozmístění je u jednoho souboru pravidelné a u druhého ne, vrací metoda malou podobnost.

Tato metoda dobře odhaluje plagiáty, ale podobně jako u porovnávání počtu výskytů tokenů občas vydá falešný výsledek — vyšší míru podobnosti než ve skutečnosti je. Pro krátké zdrojové texty (mohutnost veličiny X v řádu jednotek) je tato metoda nepoužitelná. Časová složitost tohoto porovnání je $O(\max(m, n))$.

2.2.6 Vyřazené metody

Hammingova vzdálenost
Hammingova vzdálenost patří mezi řetězcové metriky a udává počet shodných tokenů na odpovídajících si pozicích v řetězci. Jedná se vlastně o modifikaci editační vzdálenosti, kde jedinou povolenou znakovou operací je substituce tokenu. Časová složitost této metody je lineární, ale vypovídací hodnota pro naše účely je mizivá. Stačí vložit jeden token na začátek jednoho z řetězců, a metoda vykáže téměř nulovou podobnost (tokeny se posunou a přestanou se tedy na odpovídajících pozicích shodovat).

Porovnávání počtu bílých znaků
Tato metoda jako jediná nepracuje s řetězci tokenů, nýbrž s původními vstupními soubory. Metoda počítá a poté porovnává počty bílých znaků (mezera, tabulátor, znak nového řádku) obsažených v souborech. Je zřejmé, že výsledek této metody jde velmi jednoduše ovlivnit přidáním nebo ubráním bílých znaků a proto byla tato metoda z programu vyřazena.

2.3 Využití porovnávacích metod v praxi

V tabulce 2.1 je uveden přehled zmíněných porovnávacích metod (proměnné m, n představují délky řetězců tokenů porovnávaných zdrojových textů, proměnné r, s představují velikosti zdrojových textů v bytech). Každá metoda má svá specifika a zdaleka ne všechny

[4] vzdáleností je myšlen počet tokenů mezi dvěmi výskyty

Metoda	Časová složitost	Spolehlivost metody
Levenshteinova vzdálenost	$O(m \times n)$	velmi dobrá
Levenshteinova vzdálenost s ořezáváním	$O(\max(m, n))$	velmi dobrá
Nejdelší společný podřetězec	$O(m \times n)$	velmi dobrá
Porovnávání počtu tokenů	$O(\max(m, n))$	dobrá
Porovnávání počtu různých identifikátorů	$O(1)$	dobrá
Směrodatná odchylka výskytu tokenů	$O(\max(m, n))$	průměrná
Hammingova vzdálenost	$O(\max(m, n))$	špatná
Porovnávání počtu bílých znaků	$O(\max(r, s))$	špatná

Tabulka 2.1: Přehled porovnávacích metod

metody se hodí pro všechny typy porovnávání. V této kapitole se budu zabývat tím, které metody se hodí pro jaká porovnávání a jakými způsoby lze z dílčích výsledků metod složit výslednou míru podobnosti.

2.3.1 Typy porovnávání

S ohledem na budoucí využití programu pro hromadná porovnávání můžeme počítat s dvěma hlavními typy užití aplikace (a s tím souvisejícími typy porovnávání).

První užití spočívá v hledání plagiátů mezi jednorázově zadanými úkoly. V takovém případě se předpokládá, že daná množina zdrojových textů (kolekce) je k dispozici najednou, že se využije k jednomu porovnání (které vydá nejpodobnější dvojice textů) a další zdrojové texty nebudou následně do kolekce přidávány. Porovnávat se bude každý soubor s každým,[5] provede se tedy obecně $O(k^2)$ porovnání dvojic textů, kde k je počet souborů v kolekci. V tuto chvíli je pro nás nejpodstatnější fakt, že zdrojové texty v tomto typu kolekcí *řeší stejnou úlohu*, tedy že jsou si vzájemně mnohem podobnější, než texty řešící obecně různé úlohy. Z toho důvodu se pro tato porovnávání hodí i metody, které nejsou při porovnávání obecně odlišných textů příliš spolehlivé. Patří sem zejména porovnávání počtu tokenů a směrodatná odchylka výskytu tokenů, obě tyto metody se zaměřují pouze úzce na jednu charakteristiku textů, proto mohou vydat falešný výsledek (vyšší míru podobnosti) i u zdrojových textů řešících odlišné úlohy. Nejlepších výsledků dosahují řetězcové metriky, jejichž spolehlivost je velmi dobrá. Jsou však časově náročnější než výše uvedené metody, což se při relativně vysokém počtu porovnávaných dvojic souborů negativně projevuje na délce porovnávání.

Dalším typickým užitím aplikace je situace, kdy pedagog dostane zápočtový program a chtěl by zjistit, zda nebyl odvozen od jiného zápočtového programu. K takovému účelu slouží kolekce, kde se sdružují všechny zápočtové úlohy napsané v daném programovacím jazyce. Zápočtový program je porovnán (nejvýše jednou) s každým textem v kolekci, provádíme tedy $O(k)$ porovnání, kde k je počet souborů v kolekci[6]. Soubory v této kolekci řeší

[5] pouze v obecném případě, aplikace umožňuje množinu dvojic souborů, které se budou porovnávat, účině omezit dle zadaných podmínek — viz. oddíl 3.4.1 na straně 15

[6] předpokládá se, že tento počet se bude postupně zvyšovat — každý nově porovnávaný zápočtový

obecně *různé úlohy* a vhodnými porovnávacími metodami jsou řetězcové metriky nebo porovnávání počtu použitých identifikátorů (u dalších metod si musíme dát pozor na falešně pozitivní výsledky). Kvůli větší rozmanitosti zdrojových textů se dají v tomto případě úspěšně aplikovat podmínky, které omezí množinu souborů k porovnání, a tím i výrazně zrychlí celý proces porovnávání — více se možnostem nastavení těchto podmínek věnuji v kapitole 3.4.1 na straně 15.

2.3.2 Skládání dílčích měr

Při vhodně zvolených porovnávacích metodách je dostačujícím měřítkem podobnosti zdrojových textů *aritmetický průměr* výsledků použitých metod. V případě, že jedna či více metod vykazuje vyšší (a nepravidelnou) chybovost, může být výsledná míra počítaná pomocí aritmetického průměru negativně zkreslena. V takovém případě je vhodnější použít namísto průměru *medián* z dílčích výsledků. Nevýhodou tohoto přístupu je nutnost mít k dispozici větší množství dílčích výsledků — počítat medián z dvou či tří výsledků metod výše popsaný problém nevyřeší. Posledním způsobem, který využívám u většiny instrukčních souborů, je *vážený průměr* výsledků metod, kde spolehlivějším metodám přiřadím vyšší váhu a naopak.

program bude nahrán do kolekce, aby posloužil uživatelům při dalších porovnáváních

Kapitola 3

Nadstavba aplikace

Přístup do aplikace mají pouze zaregistrovaní uživatelé, kteří se přihlašují so systému zadáním přihlašovacího jména a hesla. Autorizace je vyžadována kvůli zamezení přístupu nepovolaným osobám (například studentům) a kvůli identifikaci uživatele systému. Aplikace rozlišuje dva typy uživatelů – běžného uživatele a administrátora systému. Běžný uživatel má oproti administrátorovi omezená práva, konkrétní rozdíly budu uvádět dále v textu.

3.1 Administrace kolekcí

Aplikace umožňuje uživateli vytvářet vlastní kolekce. Kolekce může být vytvořena jako *soukromá* nebo *sdílená*. Do soukromé kolekce má přístup pouze její vlastník (uživatel, který jí založil). Tento druh kolekcí se hodí například pro jednorázové úkoly zadané jedním pedagogem, kde není zapotřebí, aby do kolekce zasahovali jiní uživatelé. Naproti tomu do sdílených kolekcí mohou všichni uživatelé nahrávat zdrojové texty a spouštět porovnávání. Příklad sdílené kolekce je kolekce všech zápočtových úkolů z jazyka Pascal. Administrátor má přístup do všech kolekcí bez omezení.

3.2 Nahrávání zdrojových textů do databáze

Nahrávání zdrojových textů do databáze lze provádět dvěma způsoby — jednotlivě a po celých kolekcích. Při nahrávání jednoho textu uživatel zadá základní informace o textu (zejména jméno studenta, datum vypracování a do jaké kolekce zdrojový text patří) a dále zvolí, zda se má po nahrání text porovnat se všemi zdrojovými texty v kolekci. Druhý způsob nahrávání umožňuje uživateli nahrát celý balík zdrojových textů najednou. Uživatel nahraje do systému jeden archiv typu *.zip*, aplikace nejprve vytvoří kolekci pro tyto zdrojové texty, následně archiv rozbalí a nahraje postupně všechny zdrojové texty z archivu do vytvořené kolekce. Tento způsob nahrávání velice zjednoduší práci zejména pokud chce uživatel spustit porovávání na předem dané množině zdrojových textů — například na již několikrát zmiňovaných úkolech ze cvičení.

Zdrojový text uložený v databázi se v drtivé většině případů porovnává opakovaně. To nás vede k úvaze, zda bychom nemohli některé informace předpočítat a zamezit tak opakovanému provádění stejných výpočtů. Nezbytnou částí porovnávání dvou souborů, která pro daný zdrojový text má stále stejný výsledek, je lexikální analýza. Proto se lexikální analýza provádí vždy již v momentě, kdy je soubor nahráván do databáze. Výsledky analýzy[1] se uloží do souboru a při každém porovnávání, v němž daný zdrojový text figuruje, se místo opětovného provádění analýzy tyto informace ze souboru načtou. Kvůli této vlastnosti přibyla nová funkcionalita do porovnávacího programu. Jednak je možno porovnávací program spustit ve zvláštním režimu, kdy pouze provede lexikální analýzu zadaného zdrojového textu, výsledky uloží do souboru a skončí, druhak porovnávací program umí výsledky analýzy načíst ze souborů místo toho aby analýzu prováděl.

3.3 Porovnávání na pozadí

Časovou náročnost porovnávání ovlivňují v zásadě tři faktory: počet porovnávaných dvojic zdrojových textů, velikost zdrojových textů a náročnost použitých porovnávacích metod. Při nevhodné kombinaci těchto tří faktorů může být časová náročnost porovnávání velmi vysoká. Proto aplikace umožňuje spustit porovnávání na pozadí. Uživatel pak nemusí čekat na kompletní výsledky celého porovnávání, lze si dosažené výsledky prohlížet průběžně, i když nejsou ještě zpracovány všechny dvojice souborů, nebo může s aplikací přestat pracovat a výsledky si prohlédnout kdykoliv později. Z toho důvodu jsou všechny výsledky porovnávání průběžně ukládány do databáze, kde také po dokončení porovnávání zůstanou. Uživatel se k nim může kdykoliv vrátit a dosažené výsledky si prohlížet v přehledné tabulce, kterou lze řadit podle různých kritérií.

3.4 Nastavení porovnávání

Průběh porovnávání se dá ovlivňovat a nastavovat v několika rovinách.

3.4.1 *Co* se bude porovnávat

Porovnávání se dají spouštět ve dvou režimech:

1. Porovnávat jeden soubor se všemi ostatními v kolekci. Tato možnost je k dispozici po nahrání nového zdrojového textu.

2. Porovnávat všechny soubory v kolekci mezi sebou, tedy každý s každým.

Mimo toho aplikace umožňuje při spouštění porovnávání zadat podmínky, které mohou výrazně omezit množinu dvojic souborů určených k porovnání. Tyto podmínky se týkají

[1] jedná se o řetězec tokenů a další charakteristiky souboru — počet různých identifikátorů a celkový počet tokenů

počtu tokenů a velikosti zdrojových textů. Zadaná čísla udávají, o kolik procent se smí maximálně lišit počty tokenů (resp. velikosti souborů), aby byla dvojice souborů zahrnuta do porovnávání. Soubory, které se liší co do počtu tokenů (resp. velikosti souborů) více než je zadaná mez, jsou ignorovány. Ukazuje se, že soubory, které jsou odvozené jeden od druhého, se v drtivé většině neliší co do počtu tokenů o více než 50%. U podmínky na velikosti souborů je nutno tuto mez posunout nahoru, protože velikost souborů se dá mnohem lépe ovlivnit přidáním komentářů, bílých znaků nebo přejmenováním identifikátorů. Aktivací těchto podmínek tedy docílíme výrazného zrychlení porovnávání, přičemž neriskujeme, že neodhalíme některé plagiáty.

Informace potřebné k vyhodnocení výše uvedených podmínek (počet tokenů a velikost souboru) jsou spočteny při nahrávání zdrojových textů a jsou uloženy v databázi společně s dalšími informacemi o souboru. Vyhodnocování těchto podmínek proto probíhá v rámci výběrového dotazu do databáze a nestojí nás téměř žádný čas navíc.

3.4.2 *Jak* se bude porovnávat

Způsob, jakým bude program porovnávat dvojice zdrojových textů, je dán použitým instrukčním souborem. V aplikaci je několik předdefinovaných instrukčních souborů, které se liší svou časovou náročností a způsobem využití. Mimo to může uživatel vytvářet vlastní soubory instrukcí, díky nimž může spouštět libovolné podporované porovnávací metody a libovolně je kombinovat do výsledné míry podobnosti. Je pouze nutno dodržet syntax instrukčního souboru, aby byl pro porovnávací program srozumitelný.

Kapitola 4

Implementace

4.1 Porovnávací program

Program je určen pro operační systém linux a je napsán v jazyce C/C++. Z počátku program provede lexikální analýzu vstupních zdrojových textů. Posléze čte ze standardního vstupu po řadě instrukce, které spouštějí jednotlivé porovnávací metody. Program končí po načtení speciální instrukce *return*, jež skládá výsledky předchozích dílčích porovnání do výsledné míry podobnosti.

4.1.1 Lexikální analýza

Lexikální parser je implementován pomocí konečného automatu, který přijímá jazyk tvořený lexikálními elementy daného programovacího jazyka. Kód lexikálního parseru je vygenerován programovacím nástrojem FLEX (fast lexical analyzer generator), jehož vstupem je soubor regulárních výrazů, popisujících lexikální elementy jazyka. V současné době program podporuje jazyky Pascal a C. Symbolická tabulka názvů identifikárotů, stejně jako řetězce tokenů jsou implementovány pomocí kontejnerů knihovny STL.

4.1.2 Porovnávací metody

Z porovnávacích metod stojí za zmínku o implementaci levenshteinova vzdálenost a nejdelší společný podřetězec. Obě tyto řetězcové metriky jsou počítány metodou dynamického programování.

Levenshteinova vzdálenost
Algoritmus, který budu popisovat, vychází ze známého algoritmu uvedeného např. v [1]. Označme si $r = [1 \dots m]$ a $s = [1 \dots n]$ porovnávané řetězce. V průběhu algoritmu vytváříme postupně matici X velikosti $(m + 1) \times (n + 1)$, ve které označme řádky $0, \dots, m$ a sloupce $0, \dots, n$. V této matici platí invariant, že na pozici X_{ij} je hodnota, která udává levenshteinovu vzdálenost (pod)řetězců $r[1 \dots i]$ a $s[1 \dots j]$. Je tedy zřejmé, že námi hledaná vzdálenost řetězců r a s se nachází v matici na pozici X_{mn}.

Inicializace: Naplníme první řádek a první sloupec matice tak, že
`X[0,i]:=i`, $i = 1, \ldots, m$
`X[j,0]:=j`, $j = 1, \ldots, n$
Krok algoritmu: Pro každé i, j spočítáme X_{ij} následovně:

```
if r[i] = s[j] then cost := 0
               else cost := 1
X[i, j] := minimum(
                    X[i-1, j] + 1,
                    X[i, j-1] + 1,
                    X[i-1, j-1] + cost
                  )
```

Důkaz správnosti algoritmu je uveden v [1].

Levenshteinova vzdálenost s ořezáváním
Výpočet probíhá podobně jako u metody bez ořezávání s tím rozdílem, že v tomto případě se nevyplňuje celá matice X, nýbrž pouze konstantně široký pruh kolem hlavní diagonály. Šířka tohoto pruhu je daná parametrem metody u. Předpokládejme, že $u \ll \max(m, n)$ (v opačném případě by ořezávání nemělo smysl) a že $m \geq n$ (v opačném případě řetězce prohodíme).

Algoritmus pracuje ve třech fázích. V první fázi naplní v matici X prvky $X_{0,0}$ až $X_{u,2u}$ stejným způsobem, jako u verze bez ořezávání. V další fázi naplňuje postupně části řádků $u + 1$ až $n - u$, konkrétně prvky $X_{j,j-u} \ldots X_{j,j+u}$, kde $j = u + 1, \ldots, n - u$ (jedná se výše uvedený pruh kolem diagonály). Tyto prvky jsou až na první a poslední prvek vypočítávány také obvyklým způsobem. První prvek části řádku ($X_{j,j-u}$) je vypočítán jako

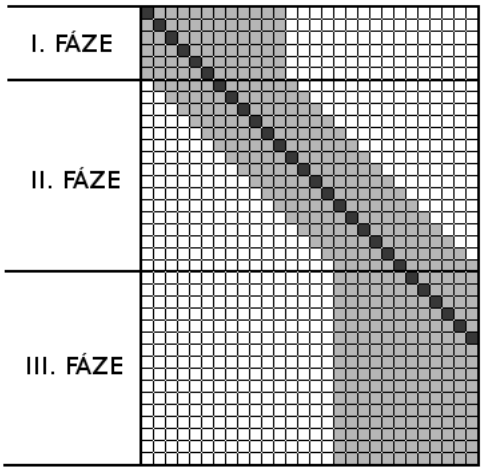
```
X[i, j] := minimum(
                    X[i, j-1] + 1,
                    X[i-1, j-1] + cost
                  )
```

a poslední prvek ($X_{j,j+u}$) je vypočítán jako

```
X[i, j] := minimum(
                    X[i-1, j] + 1,
                    X[i-1, j-1] + cost
                  )
```

v obou případech z toho důvodu, že prvek vlevo, respektive nahoře od počítaného prvku není spočítán (byl ořezán). Poslední fáze doplní zbytek řádků matice — jsou spočítány prvky $X_{n-u+1,n-2u} \ldots X_{m,n}$. Schématicky je toto ořezávání zobrazeno na obrázku 4.1.

Nejdelší společný podřetězec
Algoritmus hledání délky nejdelšího společného podřetězce je podobný algoritmu pro výpočet levenshteinovy vzdálenosti. Vycházím z algoritmu uvedeného v [2].



Obrázek 4.1: Schéma výpočtu levenshteinovy vzdálenosti s ořezáváním

Stejně jako u levenshteinovy vzdálenosti budeme postupně konstruovat matici X velikosti $(m+1) \times (n+1)$. V této matici platí invariant, že na pozici X_{ij} je hodnota, která udává délku nejdelšího společného podřetězce (pod)řetězců $r[1 \ldots i]$ a $s[1 \ldots j]$. Obdobně jako v předchozím případě najdeme délku nejdelšího společného podřetězce r a s na pozici X_{mn}.
Inicializace: První řádek a první sloupec matice naplníme nulami:
`X[0,i]:=0`, $i = 1, \ldots, m$
`X[j,0]:=0`, $j = 1, \ldots, n$
Krok algoritmu: Pro každé i, j spočítáme X_{ij} následovně:

```
if r[i] = s[j] then
        X[i,j] := X[i-1,j-1] + 1
else
        X[i, j] := maximum(
                     X[i-1, j],
                     X[i, j-1],
                   )
```

Důkaz správnosti algoritmu je uveden v [2].

4.1.3 Vyhodnocování instrukce *return*

Instrukce *return* má za argument výraz, který obsahuje odkazy na výsledky předešlých dílčí porovnání (je možno se odkazovat i na průměr či medián z výsledků metod) a hodnota tohoto výrazu tvoří výslednou míru podobnosti souborů. Při zpracovávání této instrukce se

nejprve nahradí odkazy na výsledky porovnávacích metod (resp. průměr či medián metod) za konkrétní číselné hodnoty. V další fázi je tento výraz vyhodnocen, na vyhodnocování využívám knihovnu „Simple Expression Evaluator“ (autor: Parsifal Software).

4.2 Nadstavba aplikace

Nadstavba aplikace je napsána v jazyce PHP (verze 5.2.0) a použitým databázovým serverem je MySQL (verze 4.0). Důvodem pro volbu této platformy je její veliká rozšířenost a snadná dostupnost. Uvedená verze jazyka PHP je důležitá z toho důvodu, že ve skriptech využívám nový objektový model jazyka, který je k dispozici od verze 5, a dále rozšíření pro práci s archivy *Zip File*, které je k dispozici od verze 5.2.0. Na prezentační část aplikace, tj. generování html kódu, používám šablonovací systém *Smarty* [5].

4.2.1 Struktura databáze

V databázi jsou uloženy informace o uživatelích systému (tabulka *user*), kolekcích a nahraných zdrojových textech (tabulky *collection* a *srcFile*), výsledcích předešlých porovnávání (tabulky *comparison* a *comparisonResult*) a uložených instrukčních souborech (tabulka *instr*). Na obrázku 4.2 je uvedena struktura databáze na úrovni tabulek a jejich relací.

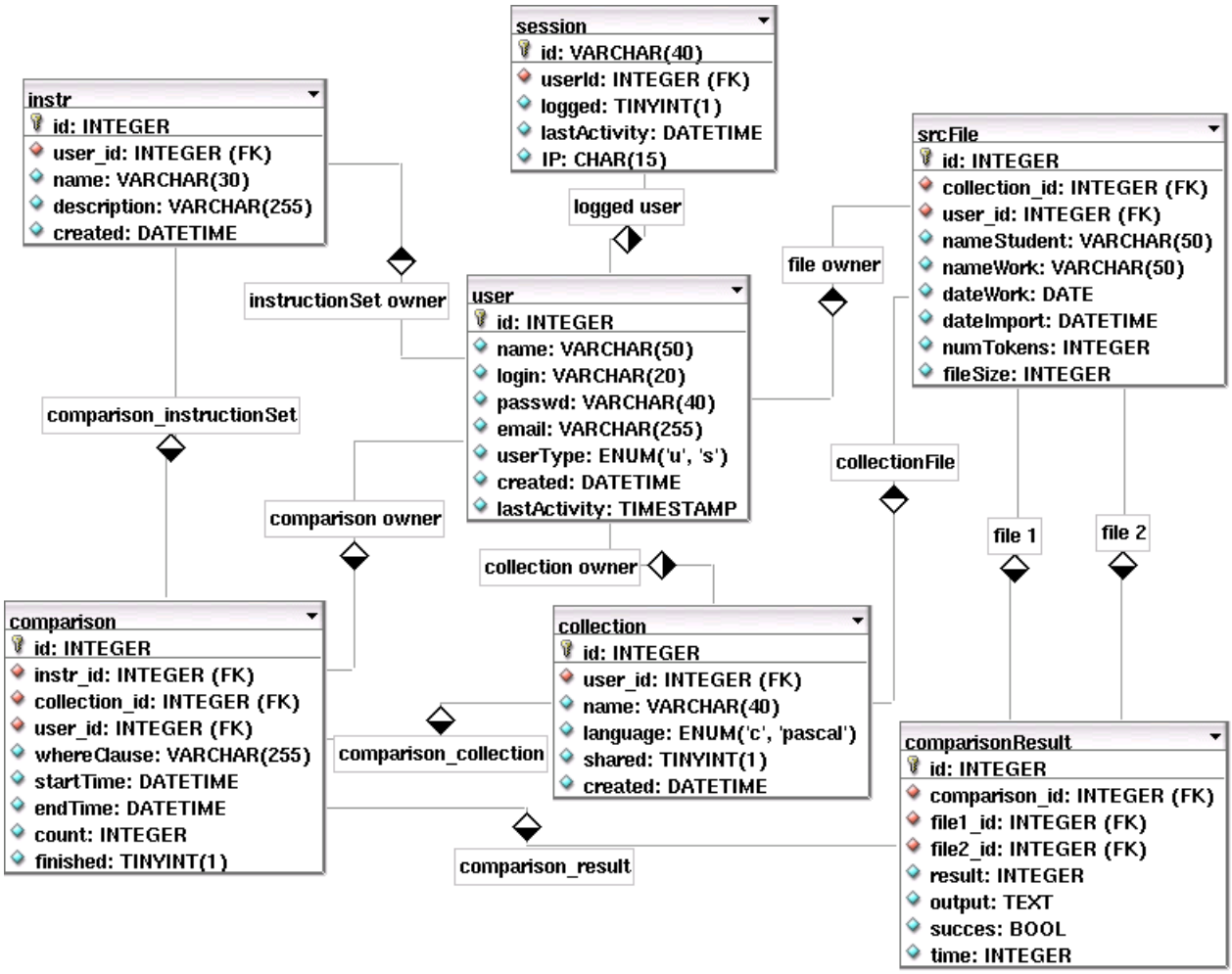
Velikou nevýhodou MySQL verze 4.0 je chabá podpora cizích klíčů, veškerou referenční integritu proto řeší sama aplikace. MySQL používám pouze jako úložiště dat.

4.2.2 Koncepce tříd

Z důvodu přehlednosti a rozšiřitelnosti kódu zapouzdřuji veškerou komunikaci s databází do tříd. Díky novému objektovému modelu PHP 5 to lze dělat velmi efektivně. Každá databázová entita má svůj obraz v PHP třídě. Všechny tyto třídy mají společného předka — třídu *NormalObject*, která sjednocuje načítání informací o objektu z databáze a dále základní operace jako je vkládání, úprava a mazání objektu, resp. záznamu v databázi. Odvozené třídy pouze nadefinují jméno tabulky a jaké vlastnosti třída má (resp. jaké sloupce má příslušná tabulka).
Příklad: třída *User* je definována následovně:

```
class User extends NormalObject {
    protected $_table = 'user';
    protected $_properties = array ('name', 'login', 'passwd', 'userType',
                                    'created', 'lastActivity');

    function __construct($id='') {
        parent::__construct($id);
    }
}
```



Obrázek 4.2: Struktura databáze — tabulky a relace

Voláním `$user = new User($id);` se vytvoří nová instance třídy *User*, která se inicializuje na hodnoty, které má záznam v tabulce *user* s primárním klíčem `$id`. Díky možnosti přetěžovat metody `__get($variable)` a `__set($variable, $value)`, které se volají při pokusu o čtení nebo zápis neexistujícího atributu metody, lze k položkám třídy přistupovat přímo, například `$user->name`. Zároveň je možno záznam jednoduše vkládat, upravovat či mazat, např. `$user->delete();`. Všechnu tuto funkcionalitu obstarávají metody nadřazené třídy *NormalObject*.

Výběrové dotazy do databáze jsou také zapouzdřeny do tříd. Tyto třídy reprezentují jakési kontejnery objektů. Dle zadaných kritérií načtou do pole primární klíče dané tabulky, jež těmto kritériím vyhovují. Například po příkazu

```
$admin_list = new Users("userType='s'");
```

bude pole `$admin_list->ids` obsahovat primární klíče všech aministrátorů systému[1]. Také tyto třídy mají jednoho předka *NormalObjects*, který sestavuje a vykonává SQL dotazy.

4.2.3 Ostatní

V této kapitole zmíním několik dalších problémů, které jsem při implementaci nadstavby aplikace řešil.

Porovnávací program je spouštěn externím voláním pomocí PHP funkce `exec`. Tato funkce zároveň vrátí výstup a návratovou hodnotu programu. Z návratové hodnoty zjistíme, zda porovnávání proběhlo v pořádku, a z výstupu programu zjistíme míru shodnosti programů.

Z důvodu, že aplikace má umožňovat porovnávání na pozadí, využívám k hromadným porovnáváním řádkový interpret PHP — *php-cli* (command-line interpreter). Rozdíl v porovnávání na popředí a na pozadí je poté pouze v tom, zda je interpret *php-cli* spuštěn na pozadí či na popředí. Následující příklady demonstrují tento rozdíl.

```
  /* příklad spuštění porovnávání na popředí: */
exec ("/usr/bin/php -f /www/plagfind/compare_exec.php 50 > /dev/null");

  /* příklad spuštění porovnávání na pozadí: */
exec ("/usr/bin/php -f /www/plagfind/compare_exec.php 50 > /dev/null &");
```

Parametr skriptu `compare_exec.php` (ve výše uvedených příkladech číslo 50) udává primární klíč záznamu v databázi, kde jsou uloženy informace o tom, co se má porovnávat. Skript vybere dvojice souborů z databáze, spouští jednotlivá porovnání (viz. předchozí odstavec) a výsledky průběžně zapisuje do databáze.

Poslední zmínka o implementaci se týká autorizace uživatele. Informace o relaci uživatele jsou uloženy v databázi v tabulce *session*, jedná se zajména o identifikátor zalogovaného uživatele a čas poslední aktivity. Při přihlášení do aplikace získá uživatel pro svou relaci jednoznačný náhodně vygenerovaný identifikátor, který zároveň tvoří klíč záznamu v tabulce *session*. Tento identifikátor relace je předáván v odkazech a formulářích v rámci aplikace jako parametr typu *get*. Na začátku každého skriptu se nalézá část týkající se autorizace. Nejprve aplikace hledá v poli `$_GET` identifikátor relace a s ním souvosející záznam v databázi. Pokud je záznam nalezen, ještě se ověří, zda se uživatel neodhásil či zda nebyl příliš dlouho neaktivní. Pokud autorizace proběhla v pořádku, skript pokračuje v činnosti, v opačném případě je prohlížeč přesměrován na přihlašovací stránku, kde se mu zobrazí chybová hláška.

[1] písmeno 's' v tomto případě znamená *supervizor*, což je jiné označení administrátora systému

Kapitola 5

Uživatelská příručka

Pro zájemce o vyzkoušení aplikace bez nutnosti složité instalace je určena verze na serveru *urtax* Matematicko-fyzikální fakulty Univerzity Karlovy. Adresa aplikace je

`http://urtax.ms.mff.cuni.cz/~fvodslon/plagfind/`

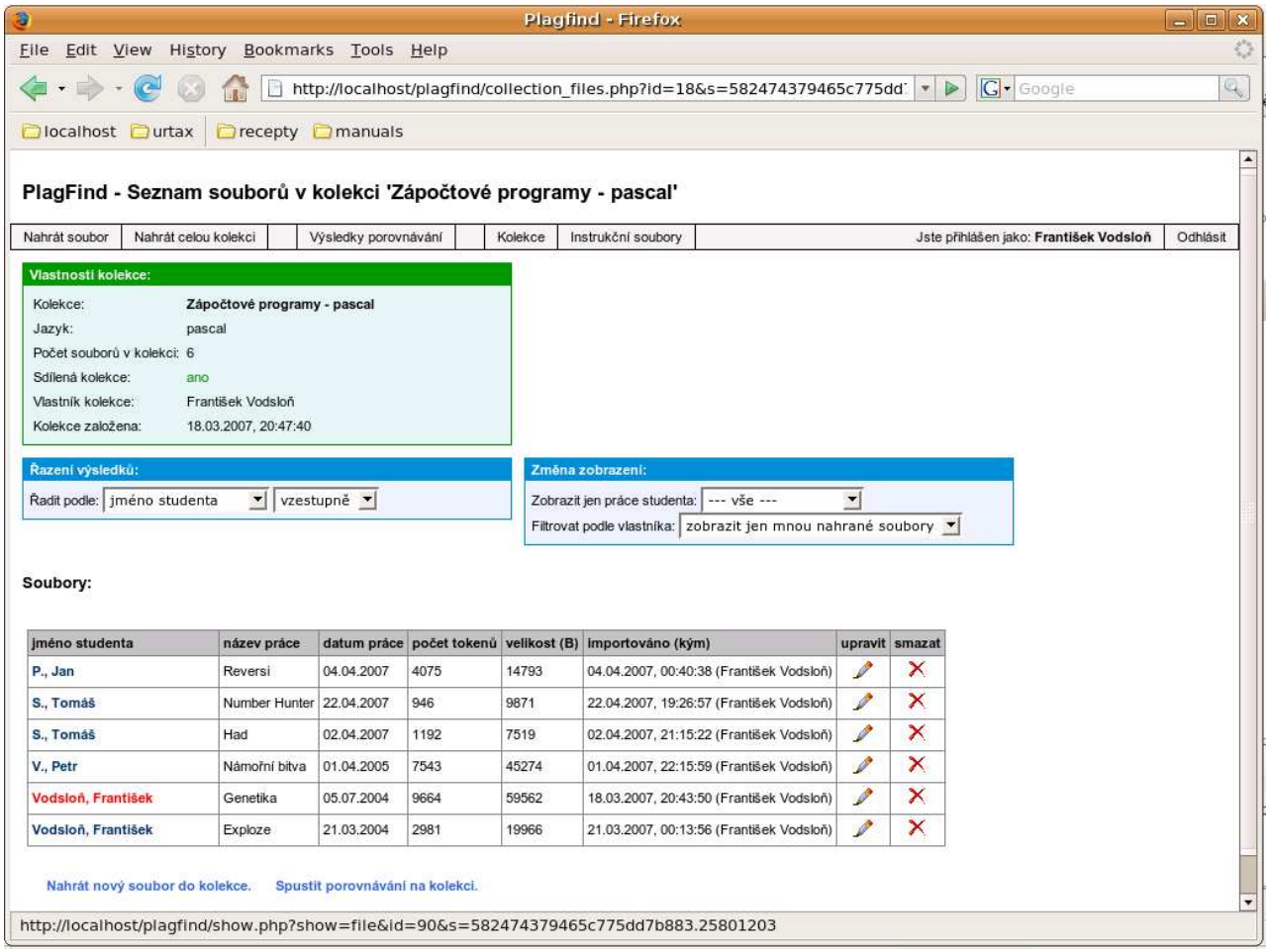
Přihlašovací údaje jsou následující:
Jméno: **test**
Heslo: **test**

5.1 Ovládání aplikace

Ovládání aplikace je velmi intuitivní. Všechny základní úkony jsou dostupné po přihlášení přes příslušné záložky v horní části stránky aplikace. Kliknutím na záložku *Kolekce* se zobrazí seznam všech uživateli přístupných kolekcí. Po kliknutí na název konkrétní kolekce se v tabulce vypíší zdrojové texty obsažené v kolekci (viz. obrázek 5.1). Na této stránce je také odkaz na spuštění porovnávání na kolekci. Po dokončení porovnávání se zobrazí stránka s dosaženými výsledky (viz. obrázek 5.2). Na stejnou stránku se můžete dostat kdykoliv později přes záložku *Výsledky porovnávání*. Dosažené výsledky jsou zobrazeny v tabulce, které je primárně řazena podle spočítané míry shodnosti. Zobrazení se dá řadit podle různých kritérií přes ovládací prvky v okně *Řazení výsledků*. Po kliknutí na jméno autora textu se uživateli zobrazí v novém okně zdrojový kód jeho práce, může tedy zdrojové texty takto vizuálně porovnat (viz. obrázek 5.3). Nahrávání zdrojových textů do databáze lze provádět přes záložky *Nahrát soubor* nebo *Nahrát celou kolekci*.

5.2 Formát instrukčních souborů

Tento oddíl je určen pro uživatele, kteří chtějí využít možnosti definovat vlastní instrukční soubory. Tato možnost je k dipozici po kliknutí na záložku *Instrukční soubory* (viz. obrázek 5.4). Každá instrukce musí být na samostatném řádku, instrukce se skládá z identifiká-

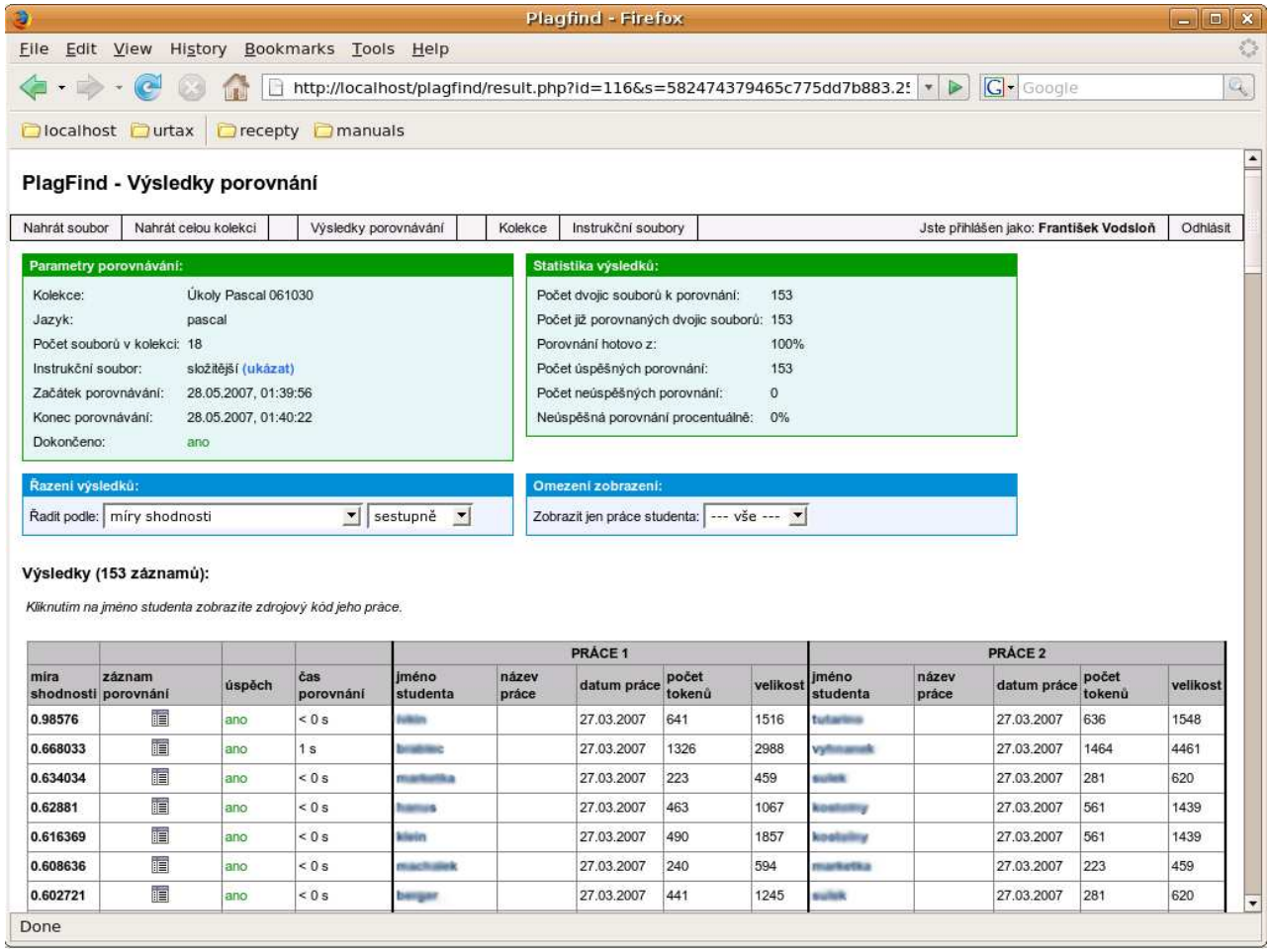


Obrázek 5.1: Přehled zdrojových textů v kolekci

toru porovnávací metody, následovaného případným parametrem. Přehled podporovaných instrukcí je uveden v tabulce 5.1.

Číslený parametr u levenshteinovy vzdálenosti udává míru ořezávání metody (viz oddíl 2.2.1). Parametr typu *token* u porovnávání počtu tokenů a směrodatných odchylek výsktytu tokenů udává token či skupinu tokenů, které se mají porovnávat. Hodnoty, kterých může nabývat tento parametr, jsou vypsány v příloze A.

Poslední instrukcí musí být speciální instrukce `return`, která vrací výslednou míru podobnosti. Parametr instrukce je matematický výraz obsahující konstanty, odkazy na předešlá porovnávání, závorky a matematické operátory `+`, `-`, `*`, `/`. Na výsledky předešlých porovnávání se odkazuje přes sekvence znaků `$1`, `$2`, `$3`, ... kde `$1` se nahradí výsledkem první použité metody, `$2` se nahradí výsledkem druhé použité metody a tak dále. Mimo to lze využít dalších dvou speciálních sekvencí: `$avg` znamená aritmetický průměr použitých metod a `$med` značí medián použitých metod.



Obrázek 5.2: Výsledky porovnávání

Příklady použití instrukce return:

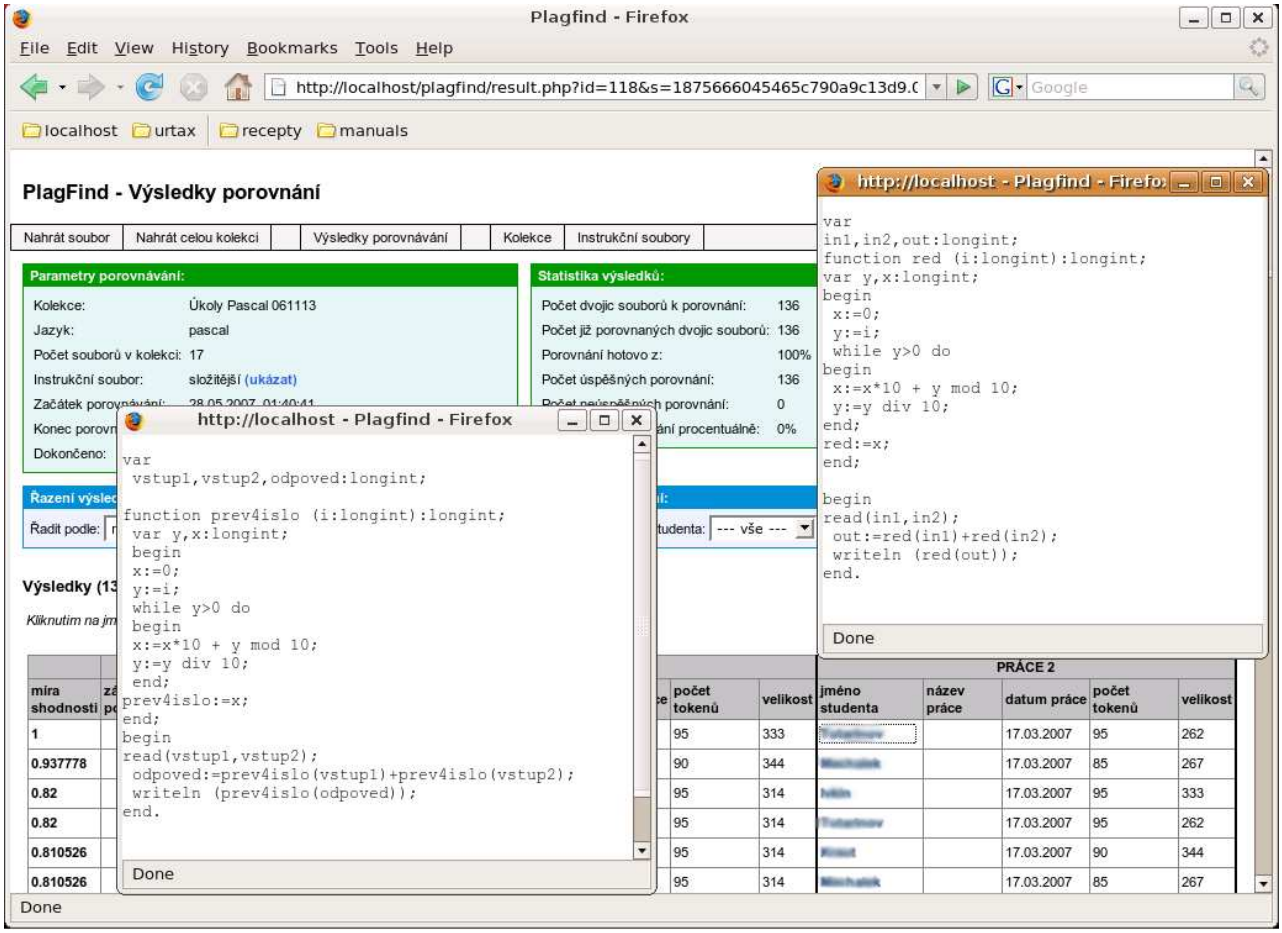
`return $med` — vrátí medián použitých metod

`return ($1 + 2* $2 + 2* $3 ) / 5` — vrátí vážený průměr metod

5.3 Systémové požadavky

Aplikace je určena pro operační systém Linux. K provozu aplikace je nutné následující programové vybavení:

- Webový server se zabudovanou podporou jazyka PHP verze 5.2.0 a vyšší.
- Databázový server MySQL verze 4.0 nebo vyšší.
- PHP-cli verze 5.2.0 nebo vyšší — řádkový interpret jazyka PHP



Obrázek 5.3: Vizuální porovnávání zdrojových textů

- Překladač jazyka C++ (konkrétně je na kompilaci použit g++)
- Flex — fast lexical analyzer generator

Na hardware žádné požadavky kladeny nejsou.

5.4 Instalace

Instalace aplikace se skládá ze dvou částí — sestavení porovnávacího programu a instalace webového rozhraní.

Zdrojové kódy porovnávacího programu jsou obsaženy v adresáři `plagfind-application`. Nahrajte celý tento adresář na svůj počítač, přejděte do podadresáře `install` a sestavte program příkazem:

```
make plagfind
```

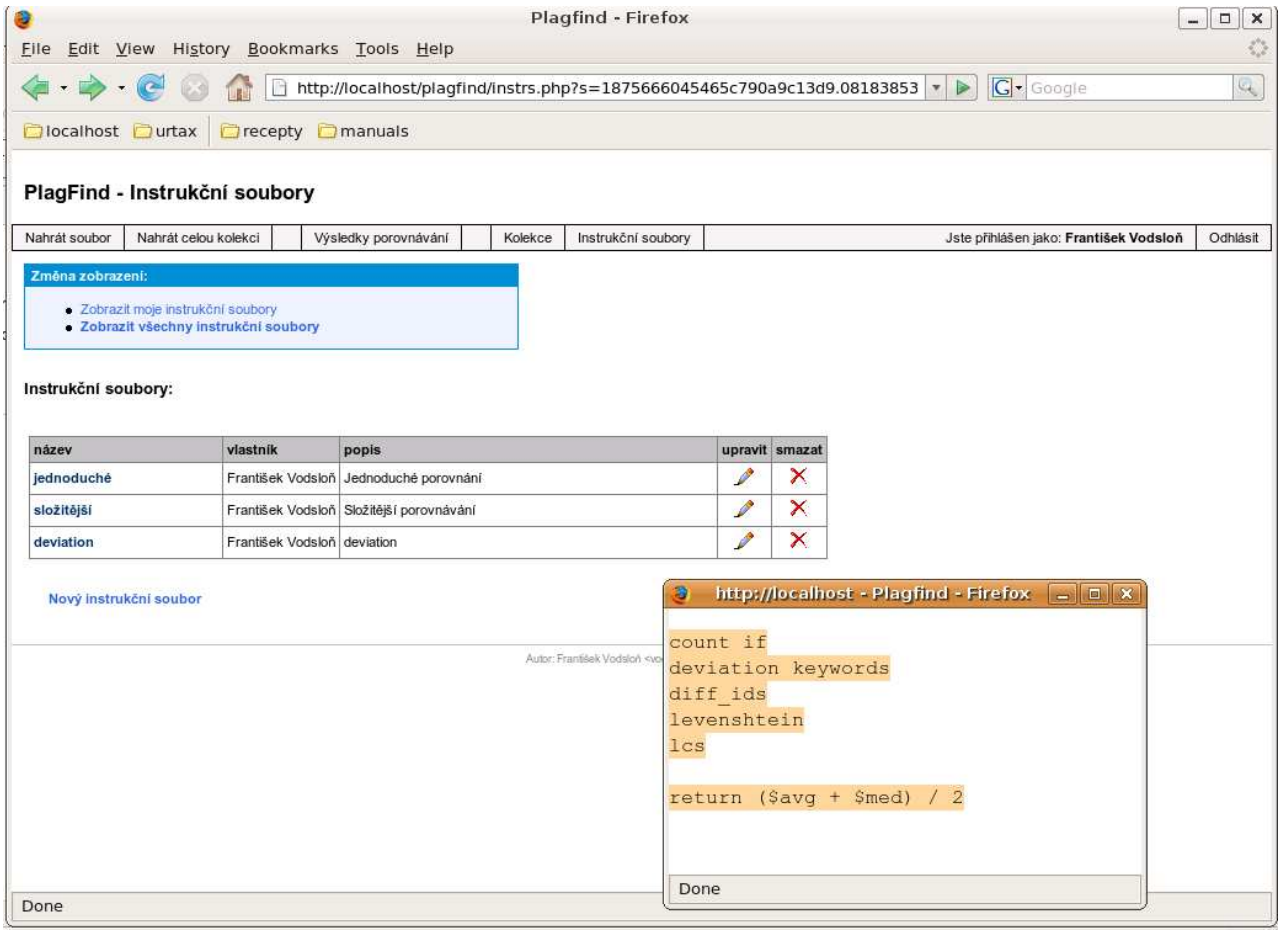
Identifikátor metody	Parametr	Metoda
`levenshtein`	číslo (nepovinný)	levenshteinova vzdálenost
`lcs`		nejdelší společný podřetězec
`count`	token (povinný)	porovnávání počtu tokenů
`deviation`	token (povinný)	směrodatná odchylka výskytu tokenů
`diff_ids`		porovnávání počtu různých identifikátorů

Tabulka 5.1: Podporované instrukce

Tím je instalace porovnávacího programu dokončena.

Postup instalace webové části aplikace:

1. Webová část aplikace je umístěna v adresáři `plagfind-www`. Nahrajte obsah tohoto adresáře na svůj počítač do umístění, kde míváte jiné webové aplikace.
2. Změňte uživatelská práva systému tak, aby mohl webový server zapisovat do adresářů `data/smarty/templates`; `files/instr`; `files/src`; `files/pre` a `include`.
3. V MySQL databázi založte nového uživatele a prázdnou databázi pro aplikaci.
4. Otevřete si webový prohlížeč a zadejte URL, kde je aplikace umístěna. Přejděte skrze odkaz na instalační skript, zde správně vyplňte všechny údaje a klikněte na tlačítko *Uložit*.
5. Kvůli zamezení nechtěných změn v konfiguraci vymažte instalační adresář `install`.



Obrázek 5.4: Administrace instrukčních souborů

Kapitola 6

Závěr

6.1 Zhodnocení práce

Ve vytvořené aplikaci nejsou možnosti rozšiřování funkcionality či zvyšování efektivity zdaleka vyčerpány (viz. následující kapitola). Přesto stávající verze tvoří fungující informační systém, který je dobře využitelný v praxi. Na řadě testovacích dat, které jsem měl k dispozici, jsem si ověřil očekávané fungování systému. U jedné kolekce domácích úkolů dokonce systém nalezl dva úkoly (od různých autorů) se 100% shodou — autoři pouze přejmenovali identifikátory a lehce změnili formátování souborů. Ukazuje se, že míra shodnosti nad 85% značí velikou pravděpodobnost, že zdrojové texty jsou odvozené jeden od druhého. Konkrétní výsledky porovnávání zde nebudu uvádět z důvodu zachování anonymity autorů testovacích dat.

Zkušenosti s provozem aplikace ukazují, že hromadné porovnání kolekce jednodušších úkolů čítající kolem 20 zdrojových textů (to jest řádově stovky porovnání dvojic souborů) se dá realizovat v řádech jednotek, výjimečně desítek sekund (závisí na mnoha faktorech — viz. kapitola 3.3 na straně 15). Horší situace nastává v případě rozsáhlých prací (např. zápočtových programů). Při použití některé řetězcové metriky s kvadratickou složitostí se můžeme dostat až na jednotky sekund trvající porovnání dvojice souborů. Celé hromadné porovnání poté může trvat tak dlouho, že systém ztrácí svou interaktivitu. Řešení spočívá ve spuštění porovnávání na pozadí a prohlédnutí si výsledků později.

Mírně zklamán jsem byl načítáním výsledků analýzy ze souborů namísto opětovného provádění lexikální analýzy. Očekával jsem mnohem větší úsporu času, než se se praxí ukázalo. Porovnávací metody jsou zřejmě časově mnohem náročnější než lexikální analýza, proto je úspora času minimální.

6.2 Možnosti dalšího vývoje

Porovnávací program
V následujících bodech uvedu několik možností, kudy by se mohl ubírat další vývoj porovnávacího programu:

- Přidání dalších porovnávacích metod.
- Zlepšení modularity programu v tom smyslu, aby podpora nových programovacích jazyků či porovnávacích metod šla přidávat co nejjednodušeji — například prostřednictvím dynamicky linkovaných knihoven.
- Rozšíření prvotní analýzy zdrojových textů o rozpoznávání typů identifikátorů, případně zjišťování dalších „charekteristik“ zdrojových textů.
- Instrukční soubor, který řídí běh programu, by mohl být obohacen například o podmínky či další rysy vyšších programovacích jazyků. Zatímco ve stávající verzi spouští program jednu porovnávací metodu za druhou a na konci vydá výsledek, mnohem efektivnější by bylo mít například možnost spustit jednu (rychlou) porovnávací metodu na začátek a podle výsledku této metody se rozhodnout zda pokračovat v porovávání (a využít složitějších metod pro co nejpřesnější výsledek) nebo zda rovnou skončit, protože další porovnávání by neměla smysl. Instrukční soubory by byly v podstatě tvořeny jednoduchým skriptovacím jazykem, což by přineslo zrychlení hromadných porovnávání a rozšíření možností programu.

Nadstavba aplikace
Nadstavba aplikace by se mohla vyvíjet například těmito směry:

- Zjednodušení instalace.
- Rozšíření možnosti výběru souborů, které se budou porovnávat — přidání dalších volitelných podmínek pro výběr, případně možnost přímo určit konkrétní soubory (množiny souborů) k porovnání.
- Podpora dalších databázových serverů vedle MySQL.
- Přidání administrace uživatelů systému.
- Větší uživatelská přívětivost při definování vlastních instrukčních souborů (kontrola správnosti syntaxe souborů).
- Další možnosti prezentace výsledků porovnávání — například vypsání *diffu* obou souborů.

Literatura

[1] *Levenshtein distance*
http://en.wikipedia.org/wiki/Levenshtein_distance

[2] *Longest common subsquence problem*
http://en.wikipedia.org/wiki/Longest_common_subsequence_problem

[3] *String Similarity Metrics for Information Integration*
http://www.dcs.shef.ac.uk/~sam/stringmetrics.html

[4] *PHP manual*
http://www.php.net/

[5] *Smarty template engine*
http://www.smarty.net/manual/en/

Příloha A

Seznam tokenů

Identifikátor	Popis	Identifikátor	Popis
`program`	klíčové slovo PROGRAM	`not`	klíčové slovo NOT
`label`	klíčové slovo LABEL	`nil`	klíčové slovo NIL
`const`	klíčové slovo CONST	`identifier`	identifikátor
`type`	klíčové slovo TYPE	`uint`	celé číslo
`var`	klíčové slovo VAR	`real`	reálné číslo
`begin`	klíčové slovo BEGIN	`string`	řetězec
`end`	klíčové slovo END	`;`	středník
`procedure`	klíčové slovo PROCEDURE	`.`	tečka
`function`	klíčové slovo FUNCTION	`,`	čárka
`packed`	klíčové slovo PACKED	`=`	rovnítko
`array`	klíčové slovo ARRAY	`:`	dvojtečka
`of`	klíčové slovo OF	`(`	levá (
`goto`	klíčové slovo GOTO	`)`	pravá)
`if`	klíčové slovo IF	`^`	stříška
`then`	klíčové slovo THEN	`..`	dvě tečky
`else`	klíčové slovo ELSE	`[`	levá [
`case`	klíčové slovo CASE	`]`	pravá]
`while`	klíčové slovo WHILE	`:=`	přiřazení
`do`	klíčové slovo DO	`reloper`	relační operátory
`repeat`	klíčové slovo REPEAT	`signaddoper`	aditivní operátory
`until`	klíčové slovo UNTIL	`muloper`	multiplikativní operátory
`for`	klíčové slovo FOR	`fordirect`	směr forcyklu (`to` / `downto`)
`or`	klíčové slovo OR		

Tabulka A.1: Tokeny jazyka Pascal

Identifikátor	Popis	Identifikátor	Popis
auto	klíčové slovo AUTO	uint	celé číslo
bool	klíčové slovo BOOL	real	reálné číslo
break	klíčové slovo BREAK	...	tři tečky
case	klíčové slovo CASE	>>=	operátor >>=
char	klíčové slovo CHAR	<<=	operátor <<=
complex	klíčové slovo COMPLEX	+=	operátor +=
const	klíčové slovo CONST	-=	operátor -=
continue	klíčové slovo CONTINUE	*=	operátor *=
default	klíčové slovo DEFAULT	/=	operátor /=
do	klíčové slovo DO	%=	operátor %=
double	klíčové slovo DOUBLE	&=	operátor &=
else	klíčové slovo ELSE	^=	operátor ^=
enum	klíčové slovo ENUM	|=	operátor |=
extern	klíčové slovo EXTERN	>>	operátor >>
float	klíčové slovo FLOAT	<<	operátor <<
for	klíčové slovo FOR	++	inkrementace
got	klíčové slovo GOTO	--	dekrementace
if	klíčové slovo IF	->	operátor ->
immaginary	klíčové slovo IMAGINARY	&&	logický součin
inline	klíčové slovo INLINE	||	logický součet
int	klíčové slovo INT	;	středník
long	klíčové slovo LONG	{	levá {
register	klíčové slovo REGISTER	}	pravá }
restrict	klíčové slovo RESTRICT	,	čárka
return	klíčové slovo RETURN	:	tvojtečka
short	klíčové slovo SHORT	=	rovnítko
signed	klíčové slovo SIGNED	(	levá (
sizeof	klíčové slovo SIZEOF	)	pravá)
static	klíčové slovo STATIC	[	levá [
struct	klíčové slovo STRUCT	]	pravá]
switch	klíčové slovo SWITCH	.	tečka
typedef	klíčové slovo TYPEDEF	&	bitový součin
union	klíčové slovo UNION	!	negace
unsigned	klíčové slovo UNSIGNED	~	vlnovka
void	klíčové slovo VOID	^	stříška
volatile	klíčové slovo VOLATILE	|	bitový součet
while	klíčové slovo WHILE	?	otazník
identifier	identifikátor	reloper	relační operátory
string	řetězec	signaddoper	aditivní operátory
character	znak	muloper	multiplikativní operátory

Tabulka A.2: Tokeny jazyka C

Identifikátor	Popis
`cycles`	cykly
`keywords`	klíčová slova
`control`	řídící konstrukce (podmínky, skoky, cykly)
`all`	všechny tokeny

Tabulka A.3: Skupiny tokenů — společné pro oba jazyky