FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Vyhledávání opsaných programů
Student:	Michal Dvořák
Vedoucí:	Ing. Ladislav Vagner, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

1. Seznamte se s problematikou porovnávání podobnosti počítačových programů. Zaměřte se na detekci plagiátů počítačových programů, které byly odevzdané jako studentská řešení cvičných školních úloh.
2. Seznamte se se stávajícím řešením detekce plagiátů v systému Progtest.
3. Navrhněte vylepšení stávajícího řešení. Zaměřte se na oblast algoritmickou (různé strategie hledání podobných částí programů) i na oblast implementační (zrychlení hledání využitím paralelismu a/nebo GPU).
4. Implementujte navržená vylepšení.
5. Otestujte realizované řešení na sadě testovacích dat. Porovnejte a diskutujte dosažené výsledky.

Elektronicky schválil/a doc. Ing. Jan Janoušek, Ph.D. dne 11. února 2021 v Praze.

Bakalářská práce

VYHLEDÁVÁNÍ OPSANÝCH PROGRAMŮ

Michal Dvořák

Fakulta informačních technologií
Katedra teoretické informatiky
Vedoucí: Ing. Ladislav Vagner, Ph.D.
12. května 2022

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Seznam algoritmů

V první řadě bych rád poděkoval vedoucímu mé bakalářské práce, Ing. Ladislavu Vagnerovi, Ph.D. Konzultace s Vámi pro mě byly oporou, posouvaly mě vpřed a mnoho jsem se během nich naučil. Dále bych rád poděkoval mé rodině za podporu nejen při studiu. Děkuji mamince za jazykové korektury. Speciální díky patří mému tatínkovi, jehož „Už to máš hotové?“ mě nenechávalo ani na moment na pochybách, že termín odevzdání se neúprosně blíží. Velké díky patří mé přítelkyni, která mi pomáhala na každém kroku. Bez ní tato práce nevznikla. Rád bych také poděkovat Ing. Petru Májovi za čas, který mi věnoval na konzultaci, Ing. Tomášovi Nováčkovi, Aleně Brožové, Bc. Veronice Dvořákové a Vojtěchu Krejsovi za pomoc s jazykovou korekturou a děkuji také všem mým přátelům a kolegům za podporu a motivaci.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 12. května 2022 .....

Abstrakt

Plagiátorství zdrojových kódů je problematika, se kterou se musí potýkat nejeden vysokoškolský kurz programování. Tato bakalářská práce se zaměřuje na vylepšení detekce opsaných programů pro systém ProgTest, portál pro podporu výuky programování na Fakultě informačních technologií ČVUT v Praze. Je navržen a implementován nový porovnávač, který využívá algoritmus Running Karp-Rabin Greedy String Tiling, jenž lze nalézt také v existujících *state of the art* detektorech plagiátorství. Nový porovnávač je schopen odhalit rafinovanější způsoby opisování, jako např. prohození funkcí. Dále také dokáže odstranit z odevzdání sdílený kód, který je studentům distribuován jako šablona řešení. Testování ukazuje, že nový porovnávač je pro velká data 3krát až 4krát rychlejší než současné řešení.

Klíčová slova plagiátorství zdrojových kódů, detekce plagiátorství, podobnost počítačových programů, ProgTest, Running Karp-Rabin Greedy String Tiling

Abstract

Source code plagiarism is an issue that many programming courses face. The aim of this thesis is to improve source code plagiarism detection in ProgTest, a submission management system employed at the Faculty of Information Technology, CTU in Prague. A new tool has been implemented for this purpose. The tool makes use of an algorithm called Running Karp-Rabin Greedy String Tiling, which has been proven effective in many state of the art plagiarism detectors. The new tool is able to recognise a wider variety of techniques used to hide plagiarism, such as block transpositions. Furthermore the tool is capable of removing parts of shared code which have been distributed by the teachers as a submission template. Experiments have observed that the new tool is up to 3–4× faster compared to the current solution.

Keywords source code plagiarism, plagiarism detection, computer program similarity, ProgTest, Running Karp-Rabin Greedy String Tiling

Seznam zkratek

AST	Abstract Syntax Tree
GST	Greedy String Tiling
LCS	Longest Common Subsequence
MOSS	Measure of System Similarity
OOP	Object-Oriented Programming
RKRGST	Running Karp-Rabin Greedy String Tiling

Úvod

Plagiátorství zdrojových kódů je problematika, s níž se potýká nejeden vysokoškolský kurz programování. Cvičné úlohy, které studenti během semestru vypracovávají, mohou čítat až tisíce odevzdání a není v silách vyučujících všechna řešení ručně zkontrolovat. Je vhodné problém delegovat na automatizovaný systém, který by pomocí strojového zpracování vytipoval konkrétní podezřelé případy, a usnadnil tak instruktorům kurzu práci s finální revizí.

Tato práce se zaměřuje na systém ProgTest, odevzdávací a vyhodnocovací portál pro podporu výuky programování na Fakultě informačních technologií ČVUT v Praze. V předmětech, kde je nasazen, jsou přes ProgTest studentům zpřístupněna zadání domácích úloh, studenti odevzdávají na ProgTest svá řešení a ProgTest jejich práci automaticky hodnotí. Zpravidla na konci semestru ProgTest provede jednorázovou kontrolu, která určí, zdali u některých odevzdání existuje podezření z opisování. Tato kontrola je poměrně časově a výpočetně náročná. Navíc existují způsoby, jak opisování zamaskovat tak, aby jej současný porovnávač neodhalil.

Cílem této práce je zdokonalit detekci plagiátů na ProgTestu. Pozornost je věnována nejen volbě, návrhu a vylepšení algoritmu, který by umožnil odhalit rafinovanější metody opisování, ale také paralelizaci finálního řešení, díky níž lze urychlit běh programu pro velká data. Nový porovnávač je pak otestován na sadě studentských programů, které byly odevzdány jako řešení domácích úloh v předchozích letech, a porovnán se současným řešením, které je na ProgTestu nasazeno.

Kapitola 1

Plagiátorství zdrojových kódů

Tato kapitola uvádí čtenáře do dění a popisuje nejčastější problémy, s nimiž se musí detekce plagiátů zdrojových kódů potýkat. Dále klade obecné předpoklady na automatizovaný systém, který by měl úspěšně pomáhat vyučujícím při zpracovávání velkého množství odevzdání.

V akademické sféře chápeme plagiátorství zdrojových kódů jako proces, při kterém student úmyslně zkopíruje celý program nebo část programu jiného studenta, případně provede dodatečné změny za účelem zamaskování opsaných částí, a následně program odevzdá, vydávaje ho za vlastní řešení. Plagiátorství zdrojových kódů se v mnoha ohledech liší od plagiátorství textů v přirozeném jazyce. Nese s sebou specifika, kvůli kterým je obecně snazší se ho dopustit a zároveň obtížnější jej odhalit [1].

Programovací úlohy, se kterými se setkávají studenti prvních ročníků vysoké školy, jsou zpravidla zaměřeny na procvičení konkrétního algoritmu nebo konkrétní programátorské dovednosti [2]. Je rozumné, zejména u jednodušších úloh, očekávat podobnosti i mezi dvěma programy, které a priori plagiáty nejsou, přinejmenším v tom, že oba takové programy musí splnit zadání úlohy, implementovat předepsané rozhraní nebo dodržet formát výstupu. Tím se odhalení skutečných plagiátů o něco ztěžuje.

Díky rigidní formální struktuře programovacích jazyků mohou studenti ve svém textovém editoru systematicky provádět rozsáhlé úpravy a přetvořit tak původní program téměř k nepoznání, aniž by však změnili jeho chování nebo jeho výstup [2]. Mnohé programovací jazyky umožňují programátorům vyjádřit algoritmy několika různými zápisy. Moderní programovací jazyky se navíc pyšní tzv. *syntaktickými pozlátky*[1], které umožňují zapsat často používané programátorské idiomy expresivnějším nebo lépe čitelným způsobem (viz obrázek 1). Přestože se podobné vlastnosti těší u programátorů velké oblibě a jejich programy jsou pak bezesporu přehlednější, naneštěstí to zároveň napomáhá studentům, kteří by rádi své opisování zamaskovali.

Rozmach internetu v posledních dvou dekádách umožnil sdílet kód rychleji a snáz než kdy dřív. Iniciativy jako Free Software Foundation [3] aktivně podporují distribuci a sdílení kódů, řešení mnoha problémů tak lze snadno dohledat ve veřejných repozitářích. Ačkoliv není možné kontrolovat řešení studentů proti všem veřejně dostupným programům (objem dat je obrovský), může se stát, že vícero studentů čerpá inspiraci ze stejného zdroje, zvlášť pokud je to zdroj snadno dohledatelný.

Stanovit hranici mezi opsaným programem a originálem je tudíž velmi obtížné. Na rozdíl od „klasického“ plagiátorství, které je svázáno poměrně přesnou definicí podléhající citační etice, neexistuje obecný úzus o tom, co za plagiátorství zdrojových kódů považovat, a co už ne [4]. Faktory, které rozhodnutí ovlivňují, zahrnují volbu programovacího jazyka, obtížnost programovací úlohy, ale i počet studentů, kteří mají kurz zapsaný.

[1] Z angl. *syntactic sugar*.

Výpis kódu 1 Procházení kolekce prvků je jeden z nejčastějších příkladů, kde mají syntaktická pozlátka dobrý význam. V C++ je díky nim od standardu C++11 kód výrazně stravitelnější.

```
vector<int>  container { 1, 2, 3 };

// C++98
for ( vector<int>::const_iterator it = container . cbegin ();
      it != container . cend ();
      ++ it )
  cout << *it << endl;

// C++11
for ( const auto & x : container )
  cout << x << endl;
```

Pro potřeby této práce se proto přidržíme definice z článku *Plagiarism by Student Programmers* [1] a za opsaný program budeme považovat takový program, u kterého kvalifikovaný člověk rozhodne nade vší pochybnost, že se jedná o program opsaný. Roli kvalifikovaného člověka zpravidla plní vyučující předmětu.

Z této definice také plyne, že žádný automatizovaný systém nedokáže pro zadaný program s dokonalou přesností určit, zdali se jedná o program opsaný, či nikoliv. Místo toho je úkolem systému vyznačit v korpusu odevzdaných programů takovou podmnožinu řešení, která se jeví jako podezřelá, a poskytnout vyučujícímu co nejvíce informací o tom, čím jsou právě tato odevzdání podezřelá, aby pak vyučující mohl udělat co možná nejlépe informované rozhodnutí.

Přestože je hranice mezi opsaným programem a originálem přinejlepším mlhavá, literatura se shoduje na několika obecných vlastnostech, které by měl porovnávač splňovat, aby se jeho výsledky daly považovat za relevantní.

1.1 Kódování souborů

Kódování souborů označuje způsob, jakým jsou znaky textu převedeny na sekvence bajtů v paměti. Studenti programovacích kurzů zpravidla vypracovávají domácí úlohy na svých osobních počítačích. Jelikož se kódování souborů liší v závislosti na operačním systému, nastavení textového editoru a osobních preferencích, odevzdávací systém, do kterého studenti svá řešení nahrávají, jich musí zvládat celou řadu. Pokud navíc odevzdávací systém umožňuje studentům opět si stáhnout svá starší odevzdání, je jasné, že je musí takový systém na své straně ukládat v nezměněné podobě.

Během porovnání dvou zdrojových kódů hraje kódování zásadní roli. Porovnávač, který na vstupu předpokládá soubory v určitém kódování, může vykazovat značně zkreslené výsledky, pokud se ve skutečnosti kódování vstupních souborů liší. Pro demonstraci uvažujme jednoduchý zdrojový kód v jazyce C uložený v kódování ASCII a následně jeho identickou kopii, tentokrát však uloženou v kódování UTF-16. Na obrázku 1.1 můžeme vidět, že binární reprezentace těchto souborů na disku je značně odlišná.

Porovnávač by se měl umět s různými kódováními souborů vypořádat, aby zajistil správné a konzistentní výsledky. Prakticky zde existují dvě řešení:

1. Před spuštěním porovnávače se všechny soubory převedou do jednotného kódování. Implementace porovnávače bude jednodušší, protože program může očekávat na vstupu soubory ve stejném kódování. Soubory na disku nicméně musíme (alespoň dočasně) zduplikovat, kopie jsou nyní v námi zvoleném kódování. Dle počtu odevzdání a dispozic odevzdávacího systému,

```c
#include <stdio.h>

int main ( void )
{
  printf ( "Hello, world!\n" );
  return 0;
}
```

```
6923 636e 756c 6564 3c20 7473 6964 2e6f
3e68 0a0a 6e69 2074 616d 6e69 2820 7620
696f 2064 0a29 0a7b 2020 7270 6e69 6674
2820 2220 6548 6c6c 2c6f 7720 726f 646c
5c21 226e 2920 0a3b 2020 6572 7574 6e72
3020 0a3b 0a7d
```

```
fffe 2300 6900 6e00 6300 6c00 7500 6400
6500 2000 3c00 7300 7400 6400 6900 6f00
2e00 6800 3e00 0a00 0a00 6900 6e00 7400
2000 6d00 6100 6900 6e00 2000 2800 2000
7600 6f00 6900 6400 2000 2900 0a00 7b00
0a00 2000 2000 7000 7200 6900 6e00 7400
6600 2000 2800 2000 2200 4800 6500 6c00
6c00 6f00 2c00 2000 7700 6f00 7200 6c00
6400 2100 5c00 6e00 2200 2000 2900 3b00
0a00 2000 2000 7200 6500 7400 7500 7200
6e00 2000 3000 3b00 0a00 7d00 0a00
```

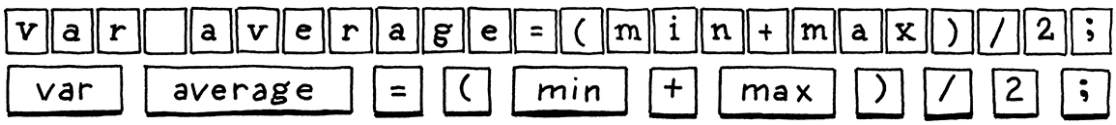
Obrázek 1.1 Nahoře vidíme referenční zdrojový kód, dole výstupy příkazu `hexdump`. Nalevo je soubor uložen v kódování `ASCII`, napravo je soubor uložen v kódování `UTF-16`. Přestože se jedná o ten samý zdrojový kód, binární reprezentace obou souborů je velmi odlišná.

zejména kapacity úložného prostoru, nemusí být vytváření kopií žádoucí. Pro pohodlné používání je také nezbytné zabalit volání porovnávače do pomocného skriptu, který konverzi zprostředkuje a který po doběhnutí kontroly vytvořené duplicity opět odstraní.

2. Porovnávač zařídí převod do jednotné reprezentace při načítání souborů do paměti. Logika porovnávače bude složitější, při načítání vstupních souborů musí program správně rozpoznat kódování a data zpracovat do jednotné formy. Používání porovnávače je ale uživatelsky příjemnější. Nemusíme odevzdání duplikovat a navíc nehrozí, že bychom konverzi zapomněli provést a dostali tak při porovnání nesprávné výsledky.

1.2 Lexikální analýza

Lexikální analýza je proces, při kterém sekvenci znaků na vstupu převedeme na posloupnost tzv. tokenů. Jedná se o vůbec první fázi při překladu zdrojových kódů [5]. Během lexikální analýzy je vstupní soubor zbaven komentářů, bílých znaků a dalších symbolů, které nejsou pro sémantiku jazyka zajímavé. Proces ilustruje obrázek 1.2.



Obrázek 1.2 Proces lexikální analýzy, která přemění vstupní textový řetězec na sekvenci tokenů. Obrázky převzány z [6].

Porovnávač by měl v úvodní fázi provádět lexikální analýzu a v dalších fázích pracovat pouze se sekvencemi tokenů, které původní soubory reprezentují. Změna formátování, přidání nebo odebrání komentářů a přejmenování identifikátorů jsou jedny z nejjednodušších změn, které lze ve zdrojovém kódu provést, což je nejspíš také důvod, proč patří mezi nejčastější pokusy o zamaskování opisování [7]. Tokenizace zajistí porovnávači robustnost vůči těmto primitivním změnám.

1.3 Společný základ

U vybraných programovacích úloh je studentům spolu s textovým zadáním distribuována také šablona řešení. Jedná se zejména o takové úlohy, které netestují pouze funkčnost programu, ale také správnost návrhu. Tento typ úloh je typický pro kurzy objektově orientovaného programování (OOP). Šablona obsahuje základ implementace (např. předepsané rozhraní, které je nutné dodržet) a často také vzorek testů, kterými si studenti mohou program sami otestovat, než jej odevzdají k ohodnocení.

Lze očekávat, že u takových úloh bude šablona řešení obsažena ve valné většině odevzdání. To může saturovat výsledky porovnávače. Odevzdání, která šablonu obsahují, budou vykazovat vyšší míru podobnosti a hrozí, že se mezi podezřelé případy dostanou odevzdání tzv. *falešně pozitivní*, tj. odevzdání, která jsou porovnávačem nesprávně označena za opsaná. Uvažujme extrémní případ, ve kterém dva různí studenti omylem odevzdají šablonu řešení bez vlastních úprav[2]. Takové programy během porovnání vykazují 100% podobnost, přestože se o plagiáty nejedná.

Ne vždy se musí jednat o nevyřešitelný problém. Pokud je šablona zaručeně součástí všech odevzdání, všechna porovnání budou zasažena stejně. Řešením může být upravení citlivosti porovnávače – zvýšení hladiny, která rozděluje podezřelá odevzdání od těch, která se již za podezřelá nepovažují. Ovšem i v takovém případě to představuje nepříjemnost, která vyžaduje bdělost na straně instruktora kurzu při zadávání úlohy a následně manipulaci s konfigurací porovnávače.

Záruku, že je šablona obsažena ve všech odevzdáních, však často nemáme. Je proto považováno za důležité, aby porovnávač dokázal společný základ odfiltrovat a při porovnání zohlednil pouze kód, který studenti sami do odevzdání přidali.

1.4 Granularita detekce

Uvažujme dvě autentická odevzdání dvou různých studentů, která řeší stejnou úlohu. Přestože ani jeden není plagiátem toho druhého, lze očekávat, že si budou místy podobná. To je způsobeno jednak tím, že oba programy implementují to samé zadání (mají stejný cíl), ale také tím, že u programovacích jazyků, stejně jako u přirozeného jazyka, lze určité „fráze“ s vysokou pravděpodobností nalézt téměř v libovolném zdrojovém kódu. Tak jako bychom mezi dvěma libovolnými texty v anglickém jazyce našli mnoho „shodných“ *the*, najdeme mezi dvěma libovolnými zdrojovými kódy v programovacím jazyce C mnoho „shodných“ deklarací proměnných primitivních nebo knihovních typů (např. `int x;`).

Takové shody jsou ale zpravidla příliš krátké a nezajímavé. Detekovat je by mělo pro porovnání spíše kontraproduktivní důsledky. Porovnávač by tudíž měl umožnit upravovat granularitu detekce, tzv. *hladinu šumu*, a všechny shody kratší nežli tato hranice považovat za nedůležité.

[2] Ačkoliv je příklad označen za „extrémní“, zkušenost ukazuje, že téměř každá úloha na ProgTestu, u které studenti dostávají šablonu řešení, je několika takovými případy poznamenána.

Kapitola 2
Existující řešení

Tato kapitola popisuje existující metody pro detekci plagiátorství zdrojových kódů. Metody jsou nejdřív klasifikovány a následně podrobněji popsány. Detailní pozornost je věnována algoritmu Greedy String Tiling, který je důležitý pro další kapitoly této práce. V závěru kapitoly jsou zmíněny dva konkrétní nástroje, které se pro detekci opisování používají.

První automatizované nástroje pro detekci plagiátorství zdrojových kódů vznikaly už v 70. letech minulého století. Informační technologie se v té době stávaly přístupnější pro širokou veřejnost a nárůst v počtu studentů programovacích kurzů šel ruku v ruce se zvýšenou mírou opisování, která rychle podnítila výzkum v oblasti její strojové detekce [8].

2.1 Klasifikace používaných metod

Způsobů, jak nástroje pro detekci plagiátorství zdrojových kódů klasifikovat, je mnoho. Jejich vyčerpávající přehled lze nalézt mj. v [9, 10]; pro tuto práci se omezíme pouze na některé z nich.

Z pohledu historického vývoje lze nástroje a jejich metody rozdělit do tří skupin:

- metody založené na výpočtu atributů,
- metody založené na strukturální analýze,
- novější přístupy.

Metody založené na výpočtu atributů využívají hodnoty tzv. softwarových metrik, podobnost určují na základě počtu operátorů, identifikátorů a jiných měřitelných vlastností. Metody založené na strukturální analýze hledají podobnosti v textové reprezentaci zdrojových kódů nebo v sekvencích tokenů, na které jsou zdrojové kódy převedeny během lexikální analýzy. Novější přístupy zahrnují pokročilé metody, jako je abstraktní interpretace nebo hledání shod mezi syntaktickými stromy.

Zajímavý vhled do způsobů, jakými konkrétní metody pro určení podobnosti pracují, může poskytnout také klasifikace z [9], která v tomto ohledu rozlišuje dvě skupiny:

- singulární metriky,
- párové metriky.

Singulární metriky v první fázi extrahují vektor hodnot (tzv. *otisk*) zvlášť pro každé odevzdání. Otisk odevzdání reprezentuje. Teprve v druhé fázi jsou otisky zkombinovány do výsledku,

jenž určí míru podobnosti. Párové metriky naopak pracují s dvěma odevzdáními najednou a míru podobnosti určují přímo z nich.[1]

2.2 Metody založené na výpočtu atributů

Metody založené na výpočtu atributů jsou typické pro starší systémy, které vznikaly v 70. a 80. letech 20. století. Tyto metody se opírají o statistickou analýzu.

Obecná myšlenka je následující: Pro každé odevzdání spočítáme n atributů. Atribut může být libovolná měřitelná vlastnost zdrojového kódu, v praxi se však používají převážně softwarové metriky. Softwarovou metriku chápeme jako číselné vyjádření vybrané vlastnosti programu [11], např. počet operátorů nebo cyklomatická složitost (počet nezávislých průchodů programem). Hodnoty vypočtených metrik uložíme do n-tice, která odevzdání reprezentuje. Tyto n-tice interpretujeme jako body v n-dimenzionálním prostoru. Hledání podobných odevzdání je pak otázkou určení vzdáleností mezi těmito body. Dvojice, které jsou blízko u sebe, jsou podezřelé z opisování.

$$f\left(\begin{array}{l}\texttt{int main (void)}\\ \texttt{\{}\\ \quad\texttt{printf ("Hello, attributes!\textbackslash n");}\\ \quad\texttt{return 0;}\\ \texttt{\}}\end{array}\right) = (a_1, a_2, a_3, \ldots, a_n)$$

Obrázek 2.1 Znázornění výpočtu atributů. Funkce přijímá zdrojový kód a vrací n-tici hodnot.

Z popisu je zřejmé, že kvalita detekce těchto metod významně závisí na volbě atributů. Některé metriky, např. počet operátorů, jsou robustní vůči přejmenování identifikátorů a také vůči prohazování funkcí – žádná z těchto transformací počet operátorů nemění. Jiné metriky, např. počet řádků, jsou náchylné i k těm nejmenším změnám, jako je odstranění komentářů. Takové metriky je jednoduché ošálit triviálními úpravami.

Není proto překvapující, že porovnávače v této kategorii experimentovaly jak s různými volbami atributů, tak s jejich počtem. Rané implementace využívaly Halsteadovy softwarové metriky [12], kterými jsou:

- počet unikátních operátorů,
- počet unikátních operandů,
- počet všech operátorů,
- počet všech operandů.

Nástroje, které následovaly, rozšířily paletu o výpočty metrik, jako je počet proměnných, počet podmíněných příkazů nebo počet smyček [13]; pozdní implementace čítaly až dvacet metrik [14]. Jiné zase připojily k metrikám váhy, díky kterým bylo možné některé z nich prioritizovat před jinými [15].

Tyto pokusy se však setkávaly s průměrnou úspěšností. Slabou stránkou metod založených na výpočtu atributů je zejména to, že převodem zdrojových kódů na vektor čísel se v procesu ztratí příliš mnoho informací o struktuře porovnávaných programů. Podrobnější studie [16] nakonec ukázala, že v porovnání s novějšími metodami strukturální analýzy, které se v té době začaly objevovat, dokáží metody založené na výpočtu atributů dosáhnout pouze nízké kvality detekce a hodí se pro odhalení jen jednodušších forem opisování.

[1] V programátorském žargonu by se dalo říct, že singulární metriky fungují jako unární funkce, zatímco párové metriky fungují jako binární funkce.

2.3 Metody založené na strukturální analýze

Metody založené na strukturální analýze pracují přímo s textovou reprezentací zdrojových kódů. Při porovnání hledají podobnost mezi řetězci znaků, příp. mezi sekvencemi tokenů, na které jsou zdrojové kódy převedeny během lexikální analýzy[2]. Díky tomu mají k dispozici kompletní informaci o podobě porovnávaných programů, a dosahují tak obecně vyšší kvality detekce než metody založené na výpočtu atributů.

Nástroje, které využívají metody založené na strukturální analýze, se opírají o algoritmy stringologie[3]. Mezi hlavní zástupce těchto algoritmů patří např. hledání nejdelší společné podposloupnosti nebo výpočet editační vzdálenosti. Tyto algoritmy jsou si velice podobné. Nejdelší společná podposloupnost (v angl. *longest common subsequence*, zkráceně LCS) dvou řetězců X, Y se zabývá, jak název napovídá, nalezením nejdelší možné podposloupnosti, která je obsažena jak v X tak v Y. Podposloupnost řetězce X chápeme jako řetězec, který z X vznikne odebráním žádného nebo více znaků. Pro dva identické řetězce je tudíž délka nejdelší společné podposloupnosti rovna délce řetězců.

Editační vzdálenost aplikuje podobný přístup. Nehledá však u X, Y společné části, nýbrž označuje počet operací vložení, odebrání nebo změny znaku, které je potřeba provést, abychom řetězec X přetvořili na řetězec Y. Pro dva identické řetězce je tudíž editační vzdálenost rovna nule.

Aplikací libovolného z výše zmíněných algoritmů při detekci plagiátorství dokážeme spolehlivě rozpoznat společnou strukturu dvou porovnávaných programů. Interpretace editační vzdálenosti může být velmi intuitivní: „Kolik změn provedl student, aby zamaskoval opisování od jiného studenta?“ Ani jeden z algoritmů si však nedokáže poradit s transpozicí celých bloků. Jednoduchou úpravou zdrojových kódů, např. proházením funkcí, lze algoritmy snadno ošálit. Nepříjemný důsledek, který to při detekci opisování může mít, lze nahlédnout v sekci 3.5.

Jiné algoritmy [17, 18] se snaží tento nedostatek adresovat. Jedním z nich je také algoritmus Greedy String Tiling [19], který využívají existující *state of the art* detektory plagiátů [7]. Tento algoritmus je důležitý také pro další kapitoly této práce, proto si jej detailněji přiblížíme.

2.3.1 Greedy String Tiling

Greedy String Tiling (GST) [19, 20] je hladový algoritmus pro porovnání dvou řetězců, který byl navržen tak, aby dokázal odhalit transpozice bloků. Algoritmus je postaven na hledání *shod*[4] mezi dvěma vstupními řetězci. Shoda označuje posloupnost tokenů, která je obsažena v obou řetězcích. Shody se navzájem nepřekrývají. Každý token může být obsažen maximálně v jedné shodě. Algoritmus se snaží pokrýt řetězce co největším počtem shod. Podobnost řetězců je pak určena počtem pokrytých tokenů v obou řetězcích vůči celkovému počtu tokenů v obou řetězcích.

Greedy String Tiling je heuristický algoritmus. Nalezené pokrytí nemusí být maximální, protože algoritmus přednostně vybírá větší shody před těmi menšími. Maximální pokrytí by navíc zohledňovalo situace, kdy více menších shod dokáže vytvořit větší pokrytí než menší počet větších shod. Tento problém však nemá známé řešení s polynomiální operační složitostí.

[2] Z hlediska strojového zpracování se však de facto jedná o stejnou věc. Jak znaky, tak tokeny jsou z pohledu počítače pouze čísla. Zatímco znaky mají číselnou hodnotu implicitně přiřazenou podle tabulky, jako je ASCII, tokeny mají číselnou hodnotu přiřazenou programátorem, který lexikální analyzátor implementuje. Proto budeme zejména v této sekci používat pojmy znak a token záměnně.

[3] Věda o zpracování řetězců a posloupností.

[4] Původní znění algoritmu v [19] rozlišuje na tomto místě dva pojmy – *tile* a *match*. První zmíněný označuje shody, které jsou finálně zvoleny do výstupní sady řešení, zatímco druhý zmíněný označuje dočasné shody, které byly nalezeny vnitřní logikou algoritmu a které je třeba protřídit a vybrat z nich ty, které se přesunou mezi *tiles*. Vzhledem k tomu, že oba pojmy odkazují na stejný „datový typ“, vystačíme si tady pouze s jedním z nich.

Algoritmus 1 Pseudokód pro algoritmus Greedy String Tiling.

```
function GST ( X[0..m], Y[0..n], initialMatchLen )
  // inicializace
  res      = new Array of matches
  xMarked  = new Array ( 0..m, false )
  yMarked  = new Array ( 0..n, false )

  do
    // inicializace jedné iterace
    tmp  = new Set of matches
    best = initialMatchLen

    for i in 0..m
      if not xMarked[i]
        for j in 0..n
          if not yMarked[j]
            // najdi co nejdelší shodu
            k = 0
            while i + k < m and j + k < n
              if X[i+k] != Y[j+k]
                 or xMarked[i+k]
                 or yMarked[j+k]
                break
              else
                k ++

            // ulož kandidáta
            if k > best
              tmp = { ( i, j, k ) }
              best = k
            else if k == best
              tmp += ( i, j, k )
            else
              // zahoď

    for (i, j, l) in tmp
      for k in 0..l
        xMarked[i+k] = true
        yMarked[j+k] = true
      res += (i, j, l)

  while best > initialMatchLen

  // výsledek
  return res

```

Pseudokód algoritmu je vidět v ukázce 1. Koncepčně probíhá hledání shod ve dvou smyčkách (řádky 14 a 16), které procházejí oba řetězce. Pro každý token z řetězce X se podíváme na každý token z řetězce Y. Pokud se rovnají, pokusíme se v porovnání pokračovat a „natáhnout“ shodu na nejdelší možnou (řádky 19–26). Jakmile již pokračovat nelze, další tokeny se nerovnají nebo jsou obsaženy v jiné shodě, porovnáme nalezenou shodu s dosavadní nejlepší shodou z předchozích iterací (řádky 28–35). Pokud je nová shoda stejně dlouhá jako dosavadní maximum, přidáme ji k němu do množiny. Pokud je však nová shoda lepší než dosavadní maximum, založíme novou množinu, do které ji přidáme, a délku nové shody si poznamenáme. Pokud je nová shoda kratší, zahodíme ji. Jakmile obě smyčky doběhnou, držíme v množině *tmp* všechny nejdelší shody, které bylo při tomto průchodu možné nalézt. Protože shody ukládáme do množiny, máme zajištěno, že se navzájem nepřekrývají. Nalezené shody překlopíme do seznamu s výsledky (řádky 37–41). Zároveň všechny tokeny, které shody pokrývají, označíme, aby již nebyly při porovnání dále uvažovány.

Tento proces se opakuje (smyčka na řádce 9), dokud je šance, že se v řetězcích stále nacházejí „zajímavé“ shody. Zajímavé shody jsou pro nás takové, které jsou dostatečně velké, aby jejich detekce nezanášela do výsledků šum (viz sekce 1.4). Hranici šumu kontroluje vstupní parametr $initialMatchLen$. Před spuštěním dvou vnitřních smyček je délka dosavadní nejlepší shody nastavena na tuto hranici (řádek 12). Pokud po doběhnutí smyček zůstala hodnota nezměněna, víme, že v následujících iteracích již nelze nalézt žádné další zajímavé shody. Algoritmus proto končí (řádek 43) a výstupem je seznam všech uložených shod.

Podobnost ze seznamu nalezených shod vypočítáme podle vzorce:

$$\text{similarity} = 100 \cdot \frac{2 \cdot \text{markedTiles}}{m + n}$$

2.3.1.1 Analýza algoritmu

Greedy String Tiling je konečný algoritmus. Tento fakt vychází z pozorování, že v každé iteraci vnější smyčky (řádek 9) označí shody, které jsou kratší než shody v iteraci předchozí. Po konečném počtu kroků musí algoritmus skončit.

Greedy String Tiling má operační složitost $O((m+n)^3)$ [21]. V nejhorším případě jsou pokryty všechny tokeny v obou řetězcích, ale každá použitá shoda má různou délku, např. $X = abbccc$ a $Y = cccbba$.

Greedy String Tiling má paměťovou složitost lineární vzhledem k délce vstupních řetězců. Kromě samotných řetězců a pomocných polí, které udržují informace o tom, jaké tokeny jsou označené, potřebujeme ukládat nalezené shody. Shoda v algoritmu figuruje jako uspořádaná trojice čísel, její paměťový otisk lze považovat za konstantní. Protože neukládáme shody, které se navzájem překrývají, v nejhorším případě uložíme za každý token právě jednu shodu.

2.3.1.2 Optimalizace

Pseudokód algoritmu Greedy String Tiling z ukázky 1 popisuje postup při naivní implementaci. Je vhodný pro demonstraci základních myšlenek, na kterých je algoritmus postaven. Jeho operační složitost je však velmi vysoká. V realných implementacích se proto aplikuje řada optimalizací. Přestože tyto optimalizace nezlepšují asymptotickou složitost algoritmu, pozorovatelná složitost se blíží $O(m + n)$. Tři nejdůležitější optimalizace si proto představíme.

Optimalizace č. 1 se soustředí na vnitřní dvě smyčky. Smyčky prochází všechny dvojice tokenů z $X \times Y$. Pokud nám ovšem do konce procházení schází méně tokenů, než kolik je délka nejdelší dosavadní shody, nemá cenu v procházení pokračovat. Takové iterace nemají šanci vylepšit lokální maximum a je možné smyčku opustit dřív. Implementaci optimalizace ukazuje obrázek 2. Jako příklad uvažujme situaci, ve které jsou vstupní řetězce identické. V původní verzi algoritmu projdou vnitřní smyčky všechny dvojice tokenů v X a Y. První iterace objeví shodu délky m, resp. n, kterou si poznamená. Iterace, které následují, postupně objeví shody délky $(m - 1)$,

$(m-2)$, atd. Každá taková shoda je však na konci zahozena, protože je kratší než shoda, která byla nalezena v první iteraci. Algoritmus provede kvadratický počet operací. Nyní uvažujme stejnou situaci, ale algoritmus vylepšený o zmíněnou optimalizaci. Vnitřní smyčky procházejí tokeny v X a Y. První iterace objeví shodu délky m, resp. n, kterou si poznamená. Druhá iterace se již neprovede, protože žádná další iterace nemůže skóre vylepšit. Algoritmus provede lineární počet operací.

Algoritmus 2 Pseudokód optimalizace, která krátí počet iterací vnitřních smyček.

```
13
14    for i in 0..m
15      if not xMarked[i]
16        for j in 0..n
17          if not yMarked[j]
18            // najdi co nejdelší shodu
```

```
13
14    for i in 0..(m - best + 1)
15      if not xMarked[i]
16        for j in 0..(n - best + 1)
17          if not yMarked[j]
18            // najdi co nejdelší shodu
```

Optimalizace č. 2 se soustředí na nalezení nejdelší shody (řádky 19–26). Podobně jako první optimalizace se stará o to, abychom zbytečně neprocházeli iterace, které se na konci ukážou být zbytečné. Abychom nalezenou shodu zařadili mezi výsledky, musí být alespoň tak dlouhá, jako je dosavadní maximum. Prvních *best* tokenů se tudíž musí rovnat. Optimalizace spočívá v tom, že porovnání jednotlivých tokenů začíná od konce, nikoliv od začátku. Takový krok se může zdát na první pohled marnivý. Pokud se tokeny rovnají, je jedno, v jakém pořadí je porovnáme. Abychom ukázali, kde se optimalizace významně uplatní, uvažujme následující příklad: Právě doběhla i-tá iterace smyčky, která prochází tokeny z X. Během této iterace byla nalezena shoda $s = (i, j, l)$ a jedná se o nejdelší dosavadní shodu. Na této shodě se tedy podílely tokeny $X_i, X_{i+1}, \ldots, X_{i+l-1}$ a $Y_j, Y_{j+1}, \ldots, Y_{j+l-1}$. Předpokládejme nyní, že tato shoda nemohla být delší, protože se tokeny X_{i+l} a Y_{j+l} nerovnají, nebo byl jeden z nich označen z předchozí iterace. Potom všech $l-1$ následujících iterací objeví podřetězce této shody, tj. shody $(i+1, j+1, l-1)$, $(i+2, j+2, l-2)$, atd. Všechny tyto shody jsou ale na konci zahozeny, protože jsou kratší než shoda s. Algoritmus přesto provede kvadratický počet porovnání, protože neshodu objeví až u posledního tokenu. To výrazně prodlužuje dobu běhu. Pokud logiku obrátíme a jako první budeme porovnávat tokeny X_{i+l}, této situaci se vyhneme. Implementace optimalizace č. 2 lze vidět na obrázku 3.

Algoritmus 3 Pseudokód optimalizace, která krátí počet iterací potřebných pro nalezení shody.

```
18    // najdi co nejdelší shodu
19    k = 0
20    while i + k < m and j + k < n
21      if X[i+k] != Y[j+k]
22         or xMarked[i+k]
23         or yMarked[j+k]
24        break
25      else
26        k ++
27
```

```
18    // najdi co nejdelší shodu
19    k = best - 1
20    while k >= 0
21      if X[i+k] != Y[j+k]
22         or xMarked[i+k]
23         or yMarked[j+k]
24        break
25      else
26        k --
27    if k >= 0
28      break
29
30    k = best
31    while i + k < m and j + k < n
32      if X[i+k] != Y[j+k]
33         or xMarked[i+k]
34         or yMarked[j+k]
35        break
36      else
37        k ++
38
```

Optimalizace č. 3 se soustředí na rychlé vyhledání zajímavých shod. Naivní algoritmus toto řeší pomocí dvou vnitřních smyček. Pro každý token z X se podíváme na všechny tokeny z Y, vždy se provede kvadratický počet porovnání. Na problém však můžeme nahlédnout z jiného úhlu pohledu. Zajímavé shody musí mít délku alespoň $initialMatchLen$. Hledání zajímavých shod proto můžeme přeformulovat jako hledání všech podřetězců z X délky $initialMatchLen$ v Y (hledání jehly v kupce sena). Zde můžeme využít myšlenku z algoritmu Karp-Rabin [22], která nám dovolí rychle eliminovat pozice v Y, na kterých se podřetězce z X nenacházejí.

Algoritmus Karp-Rabin je založen na hešování vstupních řetězců. Jeho generalizace, kterou použijeme pro vylepšení Greedy String Tiling, je následující. V X, Y zahešujeme všechny podřetězce délky $initialMatchLen$. Hešování je obecně poměrně výpočetně náročná operace, proto Karp-Rabin využívá speciální typ hešovací funkce, tzv. *posuvnou* hešovací funkci[5]. Posuvné hešovací funkce dokáží velmi rychle vypočítat heš pro $(i + 1)$-tý podřetězec, známe-li heš pro i-tý podřetězec. V důsledku toho dokážou vypočítat heše pro všechny podřetězce v lineárním čase tím, že se „posouvají“ po vstupním řetězci. Heše z řetězce Y si uložíme do hešovací tabulky spolu s informací o tom, kde se v řetězci Y nachází podřetězec, který heš reprezentuje. Při vyhledávání zajímavých shod nyní procházíme heše z X, nikoliv tokeny. Každý heš vyhledáme v hešovací tabulce, a najdeme tak podřetězce z Y, které se hešují na stejnou hodnotu. Tak lze rychle nalézt všechny výskyty zajímavých shod. Je nutné podotknout, že shodu je potřeba ověřit. Může se totiž stát, že se dva nestejné podřetězce z X, resp. Y zahešují na stejnou hodnotu. Při vhodném výběru hešovací funkce však tato situace nastává minimálně.

Pseudokód vylepšené verze algoritmu nazvané Running Karp-Rabin Greedy String Tiling (RKRGST), která implementuje všechny tři zmíněné optimalizace, je vidět v ukázce 4.

2.3.1.3 Analýza algoritmu s optimalizacemi

Konečnost algoritmu je stále zajištěna, optimalizace podmínku konečnosti nijak neupravují. Paměťová složitost algoritmu je stále lineární. Nově vytváříme pouze hešovací tabulku a ukládáme heše, jejich počet je však řádově stejný jako počet tokenů vstupních řetězců.

Asymptotická složitost je stále $O((m + n)^3)$ [19]. Vypadá to, že jsme si nepomohli. Díky optimalizacím jsme však výrazně snížili *pozorovatelnou* složitost, tj. složitost, kterou algoritmus v praxi vykazuje. Pozorovatelná složitost se blíží $O(m + n)$ [19].

[5] Z angl. *rolling hash function*.

Algoritmus 4 Pseudokód pro algoritmus Running Karp-Rabin Greedy String Tiling

```
function RKRGST ( X[0..m], Y[0..n], initialMatchLen )
  // inicializace
  res        = new Array of matches
  xMarked    = new Array ( 0..m, false )
  yMarked    = new Array ( 0..n, false )
  xHash      = new HashList from X
  yHash      = new HashList from Y
  yHashTable = new HashTable
  // heše z Y vlož do hešovací tabulky
  for (idx, hash) in yHash
    yHashTable[hash] . add ( idx )

  do
    // inicializace jedné iterace
    tmp  = new Set of matches
    best = initialMatchLen

    for (i, hash1) in xHash
      // optimalizace #1
      if i >= m - best + 1
        break
      if not xMarked[i]
        for j in yHashTable[hash1]
          // optimalizace #1
          if j >= n - best + 1
            break
          if not yMarked[j]
            // najdi co nejdelší shodu
            k = best - 1
            /* kód z optimalizace #2, zkráceno
             */

            // ulož kandidáta
            if k > best
              tmp = { ( i, j, k ) }
              best = k
            else if k == best
              tmp += ( i, j, k )
            else
              // zahoď

    for (i, j, l) in tmp
      for k in 0..l
        xMarked[i+k] = true
        yMarked[j+k] = true
      res += (i, j, l)

  while best > initialMatchLen

  // výsledek
  return res

```

2.4 Novější přístupy

Pod pojmem *novější přístupy* shrneme metody, které vznikly v posledních několika letech. Jedná se jak o nové přístupy k detekci plagiátorství zdrojových kódů, tak o metody rozšiřující nebo doplňující přístupy již existující. Novější metody se vyznačují zejména tím, že kromě textové reprezentace zdrojových kódů navíc během porovnání uvažují dodatečné informace, např. informace získané sémantickou analýzou [10].

Metody zmíněné níže lze považovat do určité míry za experimentální. Literatury, která se jimi zabývá, není mnoho. Publikace, které metody prezentují, často nezahrnují srovnání s existujícími řešeními, a je tudíž obtížné zhodnotit jejich kvalitu [10]. Vybrané novější přístupy proto shrneme pouze krátkým výčtem a stručným popiskem:

Vodoznaky

Tyto metody využívají tzv. vodoznaky jako způsob, kterým lze identifikovat autora odevzdání. Vodoznak je speciální kousek informace, který je tajně vložen do zdrojového kódu po jeho odevzdání. Vodoznak není vidět v klasickém textovém editoru, ale lze jej zobrazit při inspekci binární reprezentace souboru v programu jako např. `hexdump`. Vodoznak funguje jako identifikátor studenta. Pokud by se jiný student rozhodl zkopírovat cizí kód nebo jeho část, zkopíruje s ním také vodoznak původního autora. Při inspekci odevzdaných řešení je tak možné odhalit opsané programy.

Je zřejmé, že tato metoda detekce plagiátů vyžaduje speciální prostředí, aby se dala uplatnit. V programovacím kurzu, kde byly vodoznaky nasazeny, studenti nahrávají svá řešení pomocí aplikace, která po odevzdání vodoznaky propíše do zdrojového kódu [23]. Metoda má tu nevýhodu, že jakmile studenti zjistí, jak vodoznaky fungují, je velice snadné je obejít. Na druhou stranu při úspěšné aplikaci dokáže metoda vodoznaků poskytnout o podezřelých případech další informace, např. rozlišit to, kdo od koho opisoval.

Zkompilovaný kód

Tyto metody namísto textové podoby zdrojového kódu operují nad zkompilovaným kódem. Mohou být proto aplikovány při detekci plagiátorství napříč různými programovacími jazyky. To se hodí u programovacích úloh, které nejsou vázány na jeden konkrétní programovací jazyk. Nevýhodou těchto metod je to, že během kompilace ztrácíme velké množství informací o původní podobě programu, zejména pokud se procesu účastní také fáze optimalizace.

Shlukování

Tyto metody pomáhají při interpretaci výsledků. Shlukování označuje proces, při kterém identifikujeme celé skupiny odevzdání, které jsou si podobné. Shlukování není samostatná metoda, kterou by šlo použít pro určení podobnosti, jedná se o nadstavbu existujících řešení. Shlukování dokáže odhalit, zdali od sebe neopisovalo více lidí navzájem, a může být vhodnou formou vizualizace výsledků.

Abstraktní reprezentace

Tyto metody nejdříve převádějí zdrojový kód do abstraktní reprezentace (např. do formy AST) a teprve v ní jsou odevzdání porovnána. Abstraktní reprezentace zachycují více informací o struktuře a sémantice programu. To lze využít pro kvalitnější výsledky porovnání. Převod zdrojového kódu do abstraktní reprezentace je však složitější, je nutné napsat vlastní syntaktický analyzátor nebo využít interní reprezentace existujících překladačů. Interní reprezentace překladačů však nejsou určeny pro detekci plagiátorství a mohou kód zoptimalizovat do formy, která není pro detekci plagiátorství vhodná.

2.5 *State of the art* nástroje

Označení *state of the art* používáme pro nástroje, které mezi detektory plagiátů považujeme za jedny z nejlepších. V případě, že implementujeme vlastní porovnávač, je můžeme v jistém ohledu považovat za referenční řešení. Hledáme-li nástroj, který použít, začínáme právě u nich jako mezi těmi prvními. Určit, které nástroje by měly patřit mezi *state of the art*, bohužel není vůbec jednoduché. Mnoho vědeckých článků, které s novými nástroji přichází, neposkytuje dostatečně objektivní porovnání s jinými implementacemi. Většina nástrojů také není veřejně přístupná, porovnání s nimi tudíž ani provést nelze. Z těch, které dostupné jsou, je jen hrstka open-source[6], aby bylo možné jejich implementaci kvalitativně posoudit. Nepomáhá ani fakt, že obecně neexistuje standardizovaný způsob, jak systém pro detekci plagiátorství zdrojových kódů testovat. Drtivá většina nástrojů vzniká na akademické půdě a je testována na soukromých datasetech studentských odevzdání.

Přes to však existuje několik nástrojů, které jsou v literatuře vícekrát zastoupeny a které se v testování opakovaně umístily na horních příčkách. Dva takové nástroje byly vybrány jako zástupci. Kromě algoritmů, které pro detekci používají, věnujeme pozornost také jazykové podpoře, uživatelskému rozhraní a způsobu vizualizace výsledků.

2.5.1 MOSS

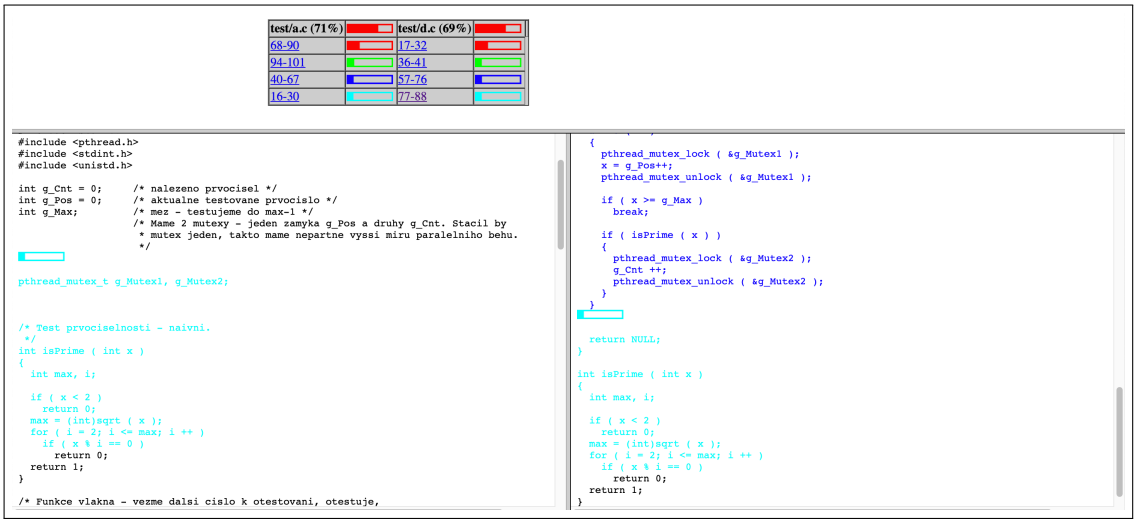
MOSS [24] (Measure of System Similarity) je nástroj pro detekci plagiátorství zdrojových kódů vyvinutý na Stanfordově univerzitě. MOSS není open-source a je dostupný pouze jako webová služba. To může být problém, pokud správa studentských odevzdání podléhá právním předpisům jako GDPR. Pro používání systému MOSS je potřeba se registrovat. Registrací uživatel obdrží unikátní identifikátor a odevzdávací skript, kterým je možné nahrávat soubory k porovnání.

Odevzdání probíhá spuštěním skriptu na příkazové řádce, argumenty jsou zdrojové kódy k porovnání, volitelně také zdrojový kód reprezentující základ implementace a programovací jazyk, ve kterém jsou zdrojové kódy napsány. MOSS vyžaduje soubory v kódování UTF-8, na stránkách je proto dostupný skript pro konverzi souborů do UTF-8. MOSS podporuje celou řadu programovacích jazyků včetně C/C++ a Java, které jsou nejčastější u univerzitních programovacích kurzů. Po úspěšném odevzdání je uživateli přiřazena unikátní URL adresa, kde najde výsledky porovnání ve formě HTML stránek. Výsledky obsahují výpočet podobností pro jednotlivé dvojice odevzdaných zdrojových kódů, u každé dvojice je také možné zobrazit detailnější náhled, který zahrnuje výpis shodných bloků ve zdrojovém kódu a jejich vizualizaci (viz obrázek 2.2). Výsledky jsou dostupné po dobu dvou týdnů.

MOSS pro porovnání používá algoritmus založený na porovnávání otisků [25], který jsme si zatím nezmiňovali; shrňme ho tedy pár větami: Zdrojové kódy jsou nejdřív převedeny na posloupnost tokenů a ty jsou poté po skupinkách stejné délky hešovány. Hešování probíhá za pomoci stejné hešovací funkce, která je zmíněna v sekci Greedy String Tiling. Na řetězec hešů je vzápětí aplikován algoritmus zvaný *Winnowing*[7], který z řetězce vybere vhodnou podmnožinu hešů, které mají odevzdání reprezentovat. Heše v této podmnožině označujeme za *otisky*. Všechny otisky jsou spolu s informací o tom, ve kterém odevzdání se nacházejí, uloženy do jednoho velkého indexu, kde jsou seřazeny podle hodnoty. Pokud se v jedné přihrádce sejdou stejné heše, které však pocházejí z různých zdrojových kódů, máme shodu. V poslední fázi se tudíž shody seskupují a skládají do větších souvislých shod. Výsledky jsou pak reportovány uživateli skrz webové rozhraní.

[6] Jejich zdrojový kód je veřejný.

[7] Pojem *winnowing* v angličtině označuje proces prosévání obilí. Vzhledem k tomu, jak algoritmus funguje, se jedná o velmi příhodný název.



Obrázek 2.2 Snímek obrazovky z webového rozhraní MOSS. Obrázek zachycuje výstup porovnání dvou zdrojových kódů, ve kterých byly nalezeny podezřelé shody. Korespondující podobné části zdrojových kódů jsou zvýrazněny stejnými barvami.

2.5.2 JPlag

JPlag [3] je nástroj pro detekci plagiátorství, který vznikl na univerzitách Karlsruhe a Erlangen-Nürnberg. JPlag byl dříve dostupný jako webová služba, v roce 2015 byl však jeho zdrojový kód zveřejněn pod licencí GPL3 a od té doby je dostupný pouze jako offline nástroj. Je to jeden z mála open-source nástrojů pro detekci plagiátorství. To se může hodit zejména, pokud je potřeba porovnání zpracovávat lokálně a není možné sdílet zdrojové kódy přes síť, např. protože se jedná o studentská odevzdání, která podléhají GDPR.

JPlag lze použít jako program na příkazové řádce nebo jej zapojit do vlastního projektu skrz programové rozhraní, které nabízí. JPlag podporuje jazyky jako C/C++, Java nebo Scheme, podporu lze rozšířit o další jazyky dodáním příslušného lexikálního analyzátoru. Zajímavostí je, že přestože JPlag pracuje s řetězci tokenů, pro jazyky Java a Scheme implementuje kompletní parser. Ten pak využívá k tomu, aby generoval tokeny, které lépe vystihují sémantiku programu. Pro jazyk Java takové tokeny mohou zahrnovat např. speciální symboly pro začátek a konec funkce, kterými je nahrazena klasická sekvence tokenů `public static void myFunc ()`. Pro jazyky C/C++ implementuje JPlag pouze lexikální analyzátor.

Pro porovnání zdrojových kódů JPlag používá algoritmus Running Karp-Rabin Greedy String Tiling. Výstupem porovnání je sada HTML stránek, které obsahují výčet porovnaných dvojic a jejich míru podobnosti. JPlag také nabízí detailní náhledy, vizualizace je podobná jako u MOSS.

Kapitola 3
ProgTest

Tato kapitola představuje současné řešení, které ProgTest využívá pro detekci plagiátorství mezi studentskými programy. Kapitola začíná stručným popisem programovacích úloh, které studenti řeší. Následně se věnuje rozboru současného porovnávače a diskutuje jeho silné a slabé stránky. Poznatky z této kapitoly vycházejí převážně z konzultací s vedoucím bakalářské práce Ing. Ladislavem Vagnerem, Ph.D.

ProgTest je odevzdávací, testovací a vyhodnocovací systém pro podporu výuky programování na Fakultě informačních technologií ČVUT v Praze.

Programovací úlohy, které studenti na ProgTestu řeší, mají zpravidla dvě podoby:

1. student vypracuje samostatný program,
2. student vypracuje modul, který se zapojí do testovacího prostředí.

V prvním případě student vypracuje celý program. Testovací prostředí program testuje sadou vstupů a odpověď programu porovnává oproti očekávaným výstupům. Tento typ úloh se hodí zejména pro jednodušší programy.

V druhém případě student implementuje pouze část programu, např. funkci nebo třídu, kterou ProgTest zapojí do testovacího prostředí. Tento typ úloh testuje nejen správnou funkčnost programu, ale také správnost návrhu rozhraní a je typický pro předměty, které vyučují modulární programování a OOP. Součástí zadání může být základ implementace (viz sekce 1.3), který nastíní, jak má rozhraní vypadat, aby splnilo kritéria úlohy, případně jím může být sada testů pro základní prozkoušení požadované funkcionality.

ProgTest využívají zejména předměty prvních ročníků, které čítají až stovky nastoupivších studentů, a automatizované hodnocení odevzdání je tudíž nezbytností. Stejně tak je vhodné automatizovat detekci plagiátorství mezi odevzdanými programy. ProgTest pro tyto účely implementuje svůj vlastní porovnávač, který pro sadu odevzdání rozhodne, zdali nějaká z nich nevykazují neobvyklou míru podobnosti. Kontrola opisování probíhá zpravidla na konci semestru, případně ve vybraných okamžicích během semestru (např. doběhnutí jedné domácí úlohy, ukončení zkouškového termínu). Řešení domácích úloh jsou jednorázově zpracována a podezřelé případy postoupeny manuální inspekci vyučujících.

V této práci se zabýváme novou verzí porovnávače, který by měl současný porovnávač nahradit nebo doplnit. Abychom mohli vylepšení konkretizovat, je potřeba analyzovat současný porovnávač a identifikovat místa, kde se nabízí prostor k zlepšení.

3.1 Základní charakteristiky

Současný porovnávač je program na příkazové řádce a bez grafického rozhraní. Podporuje kontrolu plagiátů pro zdrojové soubory v jazyce C/C++. Výstupem porovnání dvou zdrojových souborů je míra podobnosti – procentuální shoda mezi dvěma soubory. Porovnávač je naimplementován v jazyce C++.

3.2 Režimy běhu

Porovnávač disponuje třemi režimy běhu:

Jeden proti jednomu Porovnávač obdrží na vstupu dvě odevzdání, která mezi sebou porovnává, a míru podobnosti vypíše na standardní výstup. Tento režim běhu nalézá uplatnění při bližší inspekci dvou zdrojových kódů (např. během manuální inspekce vyučujícím) a poskytuje flexibilitu připojit k výstupu dodatečné informace o průběhu porovnání. Ukázka běhu porovnávače v režimu *jeden proti jednomu* je vidět na obrázku 3.1.

Jeden proti všem Porovnávač obdrží na vstupu výčet odevzdání a jedno konkrétní odevzdání, které s nimi má porovnat. Tento režim běhu se používá vzácně, může však být užitečný, pokud je potřeba experimentovat s dalšími parametry porovnávače a sledovat, jak se míra podobnosti pro konkrétní zdrojový kód mění.

Všichni proti všem Porovnávač obdrží na vstupu výčet všech odevzdání, která je potřeba zkontrolovat. Porovnávač vytvoří dvojice tak, aby se každé odevzdání porovnalo se všemi ostatními odevzdáními ve vstupní sadě, a pro každou takovou dvojici spustí režim *jeden proti jednomu*. Výsledky porovnání zapisuje do výstupního souboru, ze kterého pak vyučující vychází při manuální inspekci. Režim běhu *všichni proti všem* lze spustit také vícevláknově.

```
$ dup_pair src-001235/000601485 src-001235/000608820
File src-001235/000601485 tokenized
    tokens: 375
File src-001235/000608820 tokenized
    tokens: 408
Match: 45.69 %
```

Obrázek 3.1 Ukázka běhu porovnávače v režimu *jeden proti jednomu*. Výpis obsahuje dodatečné informace o velikosti porovnávaných souborů. Symbol `$` označuje indikátor příkazové řádky.

3.3 Formát vstupu a výstupu

Režimy běhu *jeden proti všem* a *všichni proti všem* na vstupu očekávají výčet odevzdání, která jsou kontrole na opisování podrobena. Výčet odevzdání je uložen v textovém souboru, který je porovnávači přiložen na příkazové řádce. Každý řádek souboru reprezentuje jedno odevzdání a má následující formát:

```
<uidSubmit> <uidPerson> <isNew> <fileName>
```

kde

`uidSubmit` je číslo, které označuje unikátní identifikátor odevzdání,

`uidPerson` je číslo, které označuje unikátní identifikátor studenta, v souboru nicméně může být více odevzdání od stejného studenta,

`isNew` je číslo, které označuje příznak, zdali se jedná o odevzdání nové (potom je hodnota `1`), nebo staré (potom je hodnota `0`),

`fileName` je řetězec, který udává cestu k souboru se zdrojovým kódem.

Příznak `isNew` si zaslouží detailnější vysvětlení. Odevzdání z konkrétního běhu dané úlohy nejsou totiž porovnávána pouze mezi sebou, nýbrž kontrola na opisování se provádí také zpětně, oproti odevzdáním té samé úlohy z minulých let (pokud se úloha opakuje). Starší odevzdání však již není potřeba porovnávat mezi sebou, předpokládáme, že tato kontrola proběhla v době, kdy byla odevzdání „nová“. Příznak `isNew` zajistí, že taková odevzdání dokážeme odlišit.

Výstupem režimu *všichni proti všem* je textový soubor, ve kterém jsou zaznamenány výsledky pro jednotlivé dvojice. Každý řádek souboru reprezentuje porovnání jedné dvojice a má následující formát:

```
<uidSubmit1> <uidSubmit2> <ratio>
```

kde

`uidSubmit1` je číslo, které označuje identifikátor prvního odevzdání,

`uidSubmit2` je číslo, které označuje identifikátor druhého odevzdání,

`ratio` je číslo, které označuje procentuální shodu mezi dvěma porovnávanými odevzdáními.

3.4 Porovnání zdrojových kódů

Porovnání dvou zdrojových kódů pak probíhá v třech fázích:

1. zdrojové soubory jsou načteny z disku,
2. zdrojové soubory jsou převedeny na sekvence tokenů,
3. sekvence tokenů jsou porovnány na podobnost.

V první fázi jsou soubory načteny do paměti a převedeny do jednotného kódování – UTF-32. Práce s UTF-32 je pohodlná, v C/C++ lze každý znak reprezentovat jedním objektem typu `char32_t`. Jedná se o častý postup při práci s vícero kódováními. Pro tyto účely porovnávač implementuje sadu tříd, které detekují kódování vstupního souboru a následně zajišťují převod do kódování UTF-32.

V druhé fázi nastupuje lexikální analýza (viz sekce 1.2), během které jsou zdrojové kódy zbaveny komentářů a bílých znaků a převedeny na sekvence tokenů. Token je dvojice `(type, attr)`, kde `type` označuje typ tokenu (např. „identifikátor“) a `attr` označuje hodnotu atributu, který je tokenu přiřazen (např. jméno identifikátoru – „totalSum“). Proces ilustruje obrázek 3.2. Porovnávač pro účely tokenizace implementuje vlastní lexikální analyzátor pro jazyk C/C++ s podporou do verze C++20 včetně.

Poté, co jsou soubory přeměněny na sekvence tokenů, přichází na řadu samotné porovnání. Algoritmus, který porovnání implementuje, je založen na myšlence hledání nejdelší společné podposloupnosti a je velmi blízký algoritmu vážené editační vzdálenosti.

Algoritmus přiřadí dvojici odevzdání skóre na základě toho, jak *odlišná* jsou. Čím nižší skóre, tím více se odevzdání jeví jako podezřelá; dvojice s nulovým skóre tak značí úplnou shodu. Skóre je odvozeno od délky nejdelší společné podposloupnosti, na rozdíl od klasických řešení LCS však porovnávač mezi dvěma tokeny rozlišuje několik úrovní podobnosti v závislosti na tom, zdali se tokeny nerovnají vůbec, zdali se rovnají pouze jejich typy nebo zdali se rovnají jejich typy a zároveň jejich atributy.

```
int x = 10;
```

```
{ (  LEX_KW_INT,     NULL  ),
  (  LEX_IDENT,      "x"   ),
  (  LEX_OP_AS,      NULL  ),
  (  LEX_INTEGER,    10    ),
  (  LEX_OP_SEMCOL,  NULL  ) }
```

Obrázek 3.2 Proces tokenizace zdrojového kódu, každý token je reprezentován dvojicí `(type, attr)`. Typ tokenu je zastoupen symbolickou konstantou. Pokud atribut chybí (token žádný atribut nepotřebuje), je tento fakt vyjádřen symbolem `NULL`.

Výpočet skóre se tak opírá celkem o čtyři typy vah:

`tokenSkip` ohodnocení stavu, ve kterém je token přeskočen,

`tokenMismatch` ohodnocení stavu, ve kterém se tokeny neshodují,

`attrMismatch` ohodnocení stavu, ve kterém se typy tokenů shodují, ale jejich atributy ano,

`tokenMatch` ohodnocení stavu, ve kterém se shodují jak typy tokenů, tak jejich atributy.

Ukázka 5 popisuje pseudokód algoritmu. Algoritmus na vstupu očekává dvě sekvence tokenů (řetězce), X o délce m a Y o délce n, a hodnoty vah, které se výpočtů účastní. Jádro algoritmu se pak po vzoru LCS soustředí na vyplňování pomocné tabulky A. Tabulka A má $m+1$ sloupců a $n+1$ řádků[1]. V buňce $A_{i,j}$ nalezneme *nejnižší* skóre, kterým lze ohodnotit porovnání podřetězce X', který z X obsahuje pouze prvních i tokenů, a podřetězce Y', který z Y obsahuje pouze prvních j tokenů. Po doběhnutí algoritmu tak v buňce $A_{m,n}$ nalezneme skóre pro řetězce X, Y. Inicializace tabulky A probíhá na řádcích 10–14.

Vyplňování tabulky mají na starosti dvě vnořené smyčky (řádky 17–25) a logika vyplňování se řídí předpisem

$$A_{i,j} = \min \begin{cases} A_{i-1,j} + \text{tokenSkip} \\ A_{i,j-1} + \text{tokenSkip} \\ A_{i-1,j-1} + f(X_i, Y_j) \end{cases}$$

kde

$$f(a,b) = \begin{cases} \text{tokenMismatch} & \text{pokud} \quad a.type \neq b.type \\ \text{attrMismatch} & \text{pokud} \quad a.type = b.type \wedge a.attr \neq b.attr \\ \text{tokenMatch} & \text{pokud} \quad a.type = b.type \wedge a.attr = b.attr \end{cases}$$

Algoritmus využívá dynamické programování pro dosažení optimální doby běhu. Operační složitost algoritmu je $\Theta(m \cdot n)$, protože plní tabulku `A` o velikosti $(m+1) \cdot (n+1)$, každou buňku navštíví právě jednou a v každé buňce stráví konstantní množství času. Vlastní implementace dále využívá paměťově efektivní verzi algoritmu (pseudokód tento fakt však nezachycuje), stejnou optimalizaci, jakou lze aplikovat pro problém LCS. Její paměťová složitost je tudíž lineární vzhledem k velikosti vstupních řetězců [26].

Výstupem porovnávače nicméně není přímo vypočtené skóre, takový údaj není pro člověka příliš vypovídající. Místo toho porovnání určí procentuální shodu mezi dvěma porovnávanými zdrojovými kódy. Vypočtené skóre je proto potřeba převést. Nejdřív je vypočteno maximální skóre, na které může dvojice dosáhnout

$$\text{scoreMax} = \min(m,n) \cdot \text{tokenMismatch} + \text{abs}(m-n) \cdot \text{tokenSkip}$$

a následně je rozdíl mezi maximálním skóre a vypočteným skóre vyjádřen procentuálně

$$\text{result} = 100 \cdot \frac{\text{scoreMax} - \text{score}}{\text{scoreMax}}$$

čímž je dosaženo požadovaného výsledku.

[1] Rozměr tabulky je oproti velikosti vstupních řetězců v každém směru větší. Jedná se o tzv. zarovnání (angl. *padding*), které následně zjednodušuje logiku algoritmu. Díky němu není potřeba zvlášť ošetřovat okrajové případy jako `A[i-1,j-1]` kontrolou, zdali nevystupujeme mimo hranice pole.

Algoritmus 5 Pseudokód algoritmu pro porovnání dvou sekvencí tokenů.

```
function        compareSeq          ( X[1..m],
                                      Y[1..n],
                                      tokenSkip,
                                      tokenMismatch,
                                      attrMismatch,
                                      tokenMatch )
  A = new Array ( 0..m, 0..n )

  // inicializace
  A[0,0] = 0
  for i in 1..m
    A[i,0] = A[i-1,0] + tokenSkip
  for j in 1..n
    A[0,j] = A[0,j-1] + tokenSkip

  // výpočet skóre
  for i in 1..m
    for j in 1..n
      skip = min ( A[i-1,j], A[i,j-1] ) + tokenSkip
      if X[i] . type != Y[j] . type
        A[i,j] = min ( skip, A[i-1,j-1] + tokenMismatch )
      else if X[i] . attr != Y[j] . attr
        A[i,j] = min ( skip, A[i-1,j-1] + attrMismatch )
      else
        A[i,j] = min ( skip, A[i-1,j-1] + tokenMatch )

  // výsledek
  return A[m,n]
```

3.5 Vyhodnocení

Porovnávač se dokáže vypořádat s různými kódováními souborů. Díky tokenizaci zdrojových kódů je také odolný vůči nejčastějším pokusům o zamaskování opisování, jako je změna formátování nebo úprava komentářů. Algoritmus, který porovnávač používá pro určení míry podobnosti, dokáže dobře odhalit situace, ve kterých jsou do opsaného zdrojového souboru přidány další kusy kódu. Je také citlivý ke změně atributů (např. přejmenování proměnných).

Porovnávač má však některé nedostatky. Nejpalčivějším problémem je to, že při porovnání zachovává pořadí tokenů, což v důsledku znamená, že zpravidla nedokáže odhalit transformace bloků. Jsou-li tyto bloky na sobě sémanticky nezávislé (jak tomu u funkcí a tříd často bývá), je jejich prohození triviální záležitostí. Problém demonstruje výpis kódu 2, kde opsaný zdrojový kód oproti originálu zamění pořadí funkcí[2] – porovnávač transformaci nedokáže odhalit a reportovaná podobnost klesne dramaticky pod 50 %.

Algoritmus navíc nedokáže odfiltrovat společný základ. U jednodušších úloh, které základ implementace nepotřebují, to není problém. Takových úloh ale není mnoho. U úloh, které základ implementace poskytují, je potřeba vhodně nastavit práh detekce tak, aby porovnávač nevyka-

[2] Ve skutečnosti se nejedná pouze o prohození funkcí, protože vykreslené funkce na sobě závisejí, jedna volá druhou. V jazyce `C/C++` je potřeba přidat dopřednou deklaraci, soubor `gcd2.c` má proto o pár tokenů víc než soubor `gcd1.c`, výsledky porovnání však tato úprava ovlivní minimálně a u objemnějších zdrojových kódů je pak prakticky zanedbatelná.

Výpis kódu 2 Porovnávač nedokáže detekovat, že oproti originálu (nalevo) jsou v opsaném zdrojovém kódu (napravo) prohozené funkce, reportovaná podobnost je výrazně nižší.

```
// file: gcd1.c
#include <stdio.h>

int gcd ( int a, int b )
{
  if ( b == 0 )
    return a;
  return gcd ( b, a % b );
}

int main ( void )
{
  printf ( "%d\n", gcd ( 10, 20 ) );
  return 0;
}
```

```
// file: gcd2.c
#include <stdio.h>

int gcd ( int a, int b );

int main ( void )
{
  printf ( "%d\n", gcd ( 10, 20 ) );
  return 0;
}

int gcd ( int a, int b )
{
  if ( b == 0 )
    return a;
  return gcd ( b, a % b );
}
```

```
$ dup_pair gcd1.c gcd2.c
File gcd1.c tokenized
        tokens: 56
File gcd2.c tokenized
        tokens: 66
Match: 44.24 %
```

zoval příliš mnoho falešně pozitivních výsledků.

Algoritmus má také vyšší časovou složitost, kvadratickou. Pro porovnání dvou zdrojových kódů je taková časová složitost dostačující, porovnávač i pro delší programy (tisíce, desetitisíce tokenů) doběhne za jednotky sekund. V rámci jedné úlohy však bývá odevzdání mnoho, dvojic k porovnání jsou jednotky až desítky milionů. Zpracovat úlohu celou může zabrat desítky hodin. Problém částečně řeší zapojení více vláken, doba běhu se pak několikanásobně zkrátí, nicméně zkontrolovat všechny úlohy v jednom semestru stále trvá i několik dní.

Kapitola 4
Nový porovnávač

Tato kapitola popisuje návrh a implementaci nového porovnávače.

Návrh a implementace nového porovnávače vychází z požadavků kapitoly Plagiátorství zdrojových kódů, reaguje na nedostatky současného porovnávače popsané v kapitole ProgTest a inspiruje se existujícími *state of the art* nástroji zmíněnými v kapitole Existující řešení.

4.1 Volba algoritmu

Zásadním nedostatkem v současném porovnávači je to, že si nedokáže poradit s transpozicí bloků. Studenti mohou pomocí nenáročných změn přeskupit nezávislé části kódu a ošálit tak detekci opisování. V jazycích C/C++ mezi nezávislé části kódu patří zejména definice tříd a funkcí.

Prvním řešením by mohlo být oddělit od sebe jednotlivé funkce. Uvažujeme-li dvě odevzdání X a Y, pak by se každá funkce z X porovnala všemi funkcemi z Y. Na základě podobnosti by se určilo maximální párování. To by dokázalo transponované funkce odhalit. Problém ovšem spočívá v rozpoznání funkcí. Abychom byli schopni funkce rozpoznat, je potřeba provést syntaktickou analýzu kódu. Jednou z možností je napsat vlastní parser pro jazyky C/C++. Tyto jazyky jsou však nechvalně proslulé komplexní gramatikou, která je na mnoha místech kontextově závislá [27]. Napsat parser je proto poměrně obtížný úkol. Navíc by bylo nutné implementaci udržovat s příchodem nových jazykových standardů.

Alternativou je využít frontend existujícího překladače. V úvahu přichází Clang, který nabízí programové rozhraní pro manipulaci vnitřní reprezentace programu v podobě abstraktního syntaktického stromu (AST). Průchodem AST je možné funkce identifikovat. Clang jakožto profesionální nástroj navíc dokáže zpracovat velké množství C++ programů. Porovnávání zdrojových kódů přímo v Clang AST však není vhodné. Interní reprezentace překladače není navržena pro detekci plagiátorství. Obsahuje v sobě mnoho informací, které překladač potřebuje pro generování optimalizovaného kódu. Lepší volbou je využít AST pouze pro nalezení míst ve zdrojovém kódu, kde jednotlivé funkce začínají a končí. Program, který je v AST zachycen, nicméně nereprezentuje původní zdrojový kód přesně. V době, kdy se AST vytváří, již proběhly úvodní fáze překladu, mj. byl spuštěn preprocesor. Preprocesor odstraňuje bloky podmíněného překladu, expanduje makra a nahrazuje preprocesorové symboly. Funkce, které se v AST objeví, tudíž nemusí ve zdrojovém kódu vůbec existovat. Využitím Clang se navíc omezujeme na konkrétní implementaci jazyka. Programy, které Clang nedokáže zpracovat, se nemůžou porovnání účastnit. ProgTest při vyhodnocení domácích úloh používá překladač GCC. Existuje řada jazykových konstruktů, které GCC dokáže přeložit, a Clang ne [28]. Pokud by student takový kus kódu umístil do opsaného zdrojového kódu, zajistil by, že stále získá body za „vypracování“ úlohy (GCC program přeloží),

ale příhodně své odevzdání vyřadí z detekce plagiátů (Clang program nepřeloží).

Ani jeden z výše uvedených způsobů také není dobře rozšiřitelný. Volba řešení, které silně závisí na konkrétním programovacím jazyku, znamená, že se musí práce duplikovat pro každý jazyk, pro který by měla být v budoucnu podpora přidána.

Lepší variantou je použít algoritmus, který není závislý na konkrétním jazyce, ale který pořád dokáže odhalit prohození funkcí. Greedy String Tiling taková kritéria splňuje. Algoritmus pracuje pouze se sekvencemi tokenů. Tokeny je stále potřeba vytvořit pomocí jazykově závislého lexikálního analyzátoru, implementovat lexikální analyzátor je však výrazně jednodušší než implementovat syntaktický analyzátor. Lexikální analyzátory je navíc možné generovat pomocí programů jako je např. `flex` [29]. Greedy String Tiling však dokáže pracovat i s obyčejným textem. V takovém případě se každý znak považuje za samostatný token. Aplikace pro programovací jazyky bude mít horší výsledky, algoritmus bude hledat shody i mezi komentáři a bílými znaky. Není však už potřeba implementovat ani lexikální analyzátor. Pro implementaci nového porovnávače byl proto zvolen Greedy String Tiling.

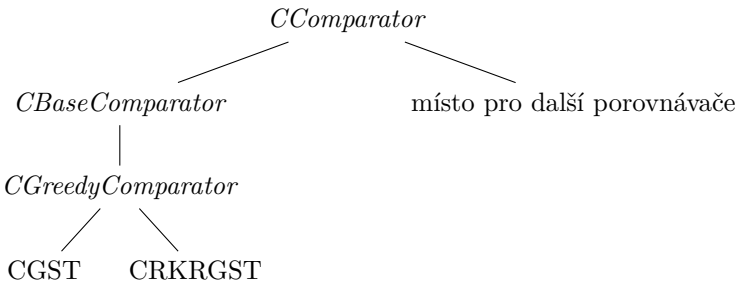
4.2 Návrh

Nový porovnávač pracuje v třech fázích, podobně jako současný porovnávač:

1. zdrojové soubory jsou načteny do paměti a převedeny do jednotného kódování,
2. zdrojové soubory jsou přeloženy na řetězce tokenů,
3. zdrojové soubory jsou porovnány pomocí algoritmu Greedy String Tiling.

Pro implementaci prvních dvou fází byly využity části existujícího porovnávače. Ten již implementuje převod do jednotného kódování UTF-32. Implementuje také lexikální analyzátor pro jazyk C/C++ včetně nejnovějšího standardu C++20. V převzatých částech byly provedeny jen menší modifikace.

Implementace třetí fáze je v novém porovnávači kompletně přepracována. Struktura programu se orientuje kolem sady tříd, které realizují porovnání zdrojových kódů. Třídy jsou součástí hierarchie:



Obrázek 4.1 Návrh tříd v novém porovnávači. Abstraktní třídy jsou vysázeny kurzívou.

Logika porovnání je zapouzdřena do objektů, které sdílí společné rozhraní v abstraktní třídě `CComparator`. Díky tomuto rozhraní lze snadno zaměnit algoritmus, který se pro porovnání použije, nebo přidat algoritmus nový. Rozhraní `CComparator` předepisuje jedinou metodu `compare(a,b)`, jejímž úkolem je porovnat dva řetězce tokenů. Třídy, které z `CComparator` dědí, musí tuto metodu implementovat.

```
double                  compare                 ( const CTokenList & a,
                                                  const CTokenList & b ) = 0;
```

Abstraktní třída `CBaseComparator` v hierarchii reprezentuje vlastnost porovnávače odfiltrovat sdílený kód. Algoritmy, které dokážou části kódu vyřadit z porovnání, dědí z této třídy. Ne všechny algoritmy umí společný základ odfiltrovat, proto je třída `CBaseComparator` potomkem obecnější třídy `CComparator`, která toto chování neočekává.

Algoritmus Greedy String Tiling je implementován ve dvou variantách – naivní implementace a optimalizovaná implementace, která využívá rolující hešovací funkci pro rychlé vyhledávání shod. Realizace obou variant je zastoupena třídami `CGST`, resp. `CRKRGST`. Implementace naivní varianty algoritmu je k dispozici pouze pro srovnání, není doporučeno ji využívat pro produkční nasazení. Hranice šumu byla po krátkém experimentování nastavena ve výchozím stavu na devět tokenů. Shody, které mají osm tokenů nebo méně, nejsou při porovnání uvažovány. Hranici šumu lze nicméně změnit skrz uživatelské rozhraní.

4.3 Odfiltrování společného základu

Porovnávač je nově schopen odfiltrovat sdílený kód. Uvažujme dvě odevzdání X, Y, která mezi sebou chceme porovnat, a jejich společný základ Z, jehož části do porovnání nemáme zahrnovat. Proces probíhá následovně:

1. spuštěním $GST(X, Z)$ nalezneme všechna místa, kde je sdílený kód obsažen v odevzdání X,
2. spuštěním $GST(Y, Z)$ nalezneme všechna místa, kde je sdílený kód obsažen v odevzdání Y,
3. nalezené shody jsou odstraněny z X, resp. z Y,
4. odevzdání X, Y teď již bez sdíleného kódu jsou porovnána spuštěním $GST(X, Y)$.

Modifikovat odevzdání nemusí být vždy žádoucí. Praktická implementace proto pro odfiltrování sdíleného kódu používá pomocná pole, která signalizují, které tokeny z odevzdání jsou po doběhnutí algoritmu označeny. V porovnáních $GST(X, Z)$ a $GST(Y, Z)$ jsou označeny ty tokeny, které odevzdání sdílí se společným základem. Pomocná pole jsou pak předána porovnání $GST(X, Y)$. Algoritmus tudíž začíná s některými tokeny již označenými, samotné řetězce tokenů však zůstanou nezměněny.

4.4 Paralelní zpracování

V současné době jsou téměř všechny osobní počítače vícejádrové. Programy, které dokážou výpočetní výkon více jader využít, mohou násobně urychlit svoji dobu běhu.

Při zapojení paralelizace je nezbytné identifikovat datově nezávislé části výpočtu. Tyto části lze rozdělit mezi vlákna a tím minimalizovat množství synchronizace. Některé algoritmy je vhodné paralelizovat interně. Algoritmus Greedy String Tiling se skládá z mnoha částí, které jsou na sobě závislé. Paralelizovat jej interně je netriviální. Lze však vlákna zapojit při zpracování dvojic odevzdání. Porovnání dvojic jsou na sobě nezávislá, a to i v případě, že se v obou dvojicích vyskytuje stejné odevzdání, data zdrojových kódů se totiž nemodifikují. K paralelizaci lze přistoupit dvěma způsoby:

1. paralelizace probíhá na úrovni jedné dvojice,
2. paralelizace probíhá na úrovni jednoho odevzdání.

V prvním případě je granularita paralelizace vyšší. Odevzdání jsou spárována do dvojic a dvojice jsou vloženy do fronty úloh, kde si je vybírají jednotlivá vlákna. Přístup k frontě úloh je potřeba synchronizovat, docílíme však lepšího dělení práce. Dvojic ke zpracování však může být mnoho. Uvažujme domácí úlohu, která čítá 7000 odevzdání. V kapitole Testování uvidíme, že takový počet odevzdání není ani nikterak extrémní případ. Úloha ve frontě je reprezentována

dvojicí čísel – identifikátory odevzdání, která se mají porovnat. Předpokládáme-li neznaménková čísla s velkým rozsahem, každá dvojice zabírá v paměti 16 B. Dvojic k porovnání může být až 24 496 500, fronta tedy v paměti zabere přinejmenším 373 MB[1]. To je poměrně mnoho vzhledem k tomu, že úlohy jsou do fronty vloženy jen proto, aby se z ní postupně odstranily. Řešením může být zpracovávat dvojice odevzdání paralelně, ale pro každé odevzdání zvlášť. Vlákna pracují v iteracích. Na začátku každé iterace se do fronty vloží všechny dvojice, jejichž levá strana je stejná – jedno konkrétní odevzdání. Počet takových dvojic nemůže být vyšší než počet odevzdání na vstupu. Po naplnění fronty se vlákna pustí do práce. Jakmile je jedno odevzdání odbaveno, vlákna se zastaví a fronta se připraví pro další odevzdání. Takové řešení má výrazně nižší paměťový otisk. Na druhou stranu se fronta musí opakovaně vyprazdňovat a připravovat pro další iterace, vlákna během této sekvenční části čekají. Toto řešení implementuje současný porovnávač.

V druhém případě je granularita paralelizace nižší. Úlohou pro jedno vlákno je zpracovat všechny dvojice pro jedno konkrétní odevzdání. Dělení práce není tak spravedlivé jako v prvním případě. Vlákna, na která připadnou delší odevzdání, musí zpracovat větší objem práce. Je možné, že jiná vlákna mezitím doběhnou. To se však může stát maximálně v $thr - 1$ případech, kde thr označuje počet vláken. Vzhledem k tomu, že spuštěných vláken jsou jednotky, maximálně nižší desítky, zatímco počet odevzdání počítáme v řádech tisíců, lze takové situace považovat za marginální. Implementace tohoto režimu paralelizace nevyžaduje zavedení explicitní fronty úloh, vláknům jsou přidělovány pouze identifikátory odevzdání, která mají zpracovat. Přidělení identifikátoru lze navíc zařídit prakticky bez synchronizace, moderní implementace si vystačí s atomickými instrukcemi. Toto řešení je implementováno v novém porovnávači.

[1] V dnešní době počítáme kapacitu paměti v desítkách GB, taková čísla nás tudíž příliš nevystraší. Problémem je spíš kvadratická závislost na počtu vstupních odevzdání. Nepředpokládáme sice, že by studenti v jednom běhu programovacího kurzu vyprodukovali násobně víc odevzdání než studenti v předchozím běhu, nesmíme ale zapomínat, že do porovnání jsou zahrnuta také odevzdání z minulých let. Čím častěji se úloha opakuje, tím více odevzdání je potřeba zpracovat.

Kapitola 5
Testování

Tato kapitola popisuje metody, kterými byl nový porovnávač testován.

Nový porovnávač slibuje lepší kvalitu detekce a kratší dobu běhu. Je přirozené se ptát, jak si skutečně vede ve srovnání se současným řešením. Kromě jednotkových testů, které pokrývají malé instance a základní situace byla proto provedeno rozsáhlé testování.

5.1 Testovací data

Testování probíhalo na třech sadách studentských programů, které byly odevzdány jako řešení domácích úloh v uplynulých letech. Testovací sady byly zvoleny tak, aby pokrývaly různé typy úloh, které studenti řeší. Statistiky pro jednotlivé datasety lze nahlédnout v tabulce 5.1.

Vlastnost	Úloha 1	Úloha 2	Úloha 3
Počet odevzdání	7 320	2 669	2 803
Počet nových odevzdání	6 382	2 357	2 529
Počet starých odevzdání	938	312	274
Počet dvojic k porovnání	26 312 243	3 504 398	3 877 254
Průměrný počet řádků	76	172	435
Medián počtu řádků	70	142	398
Průměrný počet tokenů	420	1 034	2 828
Medián počtu tokenů	387	805	2 602

Tabulka 5.1 Statistiky pro testovací sady studentských programů.

Úloha 1 reprezentuje jednodušší úlohu úvodního programovacího kurzu. Studenti se učí používat základní jazykové konstrukty, jako jsou podmíněné příkazy a aritmetické výrazy. Řešení úlohy lze napsat na pár desítek řádek. Úloha 2 reprezentuje pokročilou úlohu úvodního programovacího kurzu. Studenti v úloze procvičují práci se staticky alokovaným polem. Implementace je trochu delší, pohybuje se kolem dvou set řádek. Úloha 3 reprezentuje průměrně náročnou úlohu pokročilého programovacího kurzu. Studenti v ní implementují třídu, která musí efektivně ukládat data. Řešení může mít i několik set řádek. Součástí zadání této úlohy je také základ řešení.

5.2 Test porovnání se současným řešením

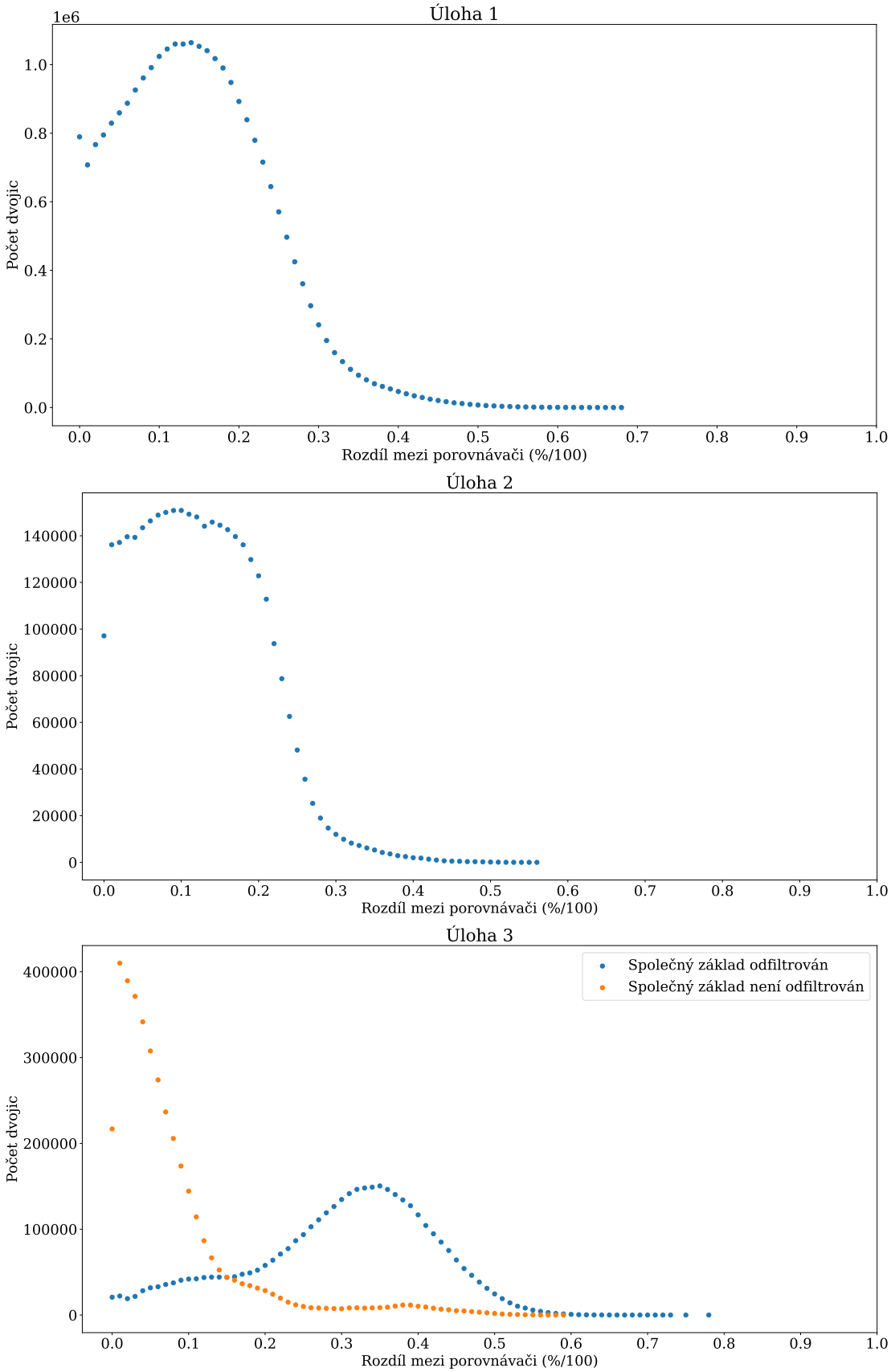
Tento test srovnává výsledky obou porovnávačů. Testování probíhalo následovně: Pro každou dvojici byla určena míra podobnosti oběma porovnávači. Míra podobnosti je vyjádřena jako číslo mezi 0 a 1. Porovnávaných dvojic je obecně mnoho, proto je srovnání realizováno jako četnost rozdílů podobností, které porovnávače jednotlivým dvojicím přiřadily. Výsledky měření pro každou úlohu jsou vidět na obrázcích 5.1.

Pro úlohy 1 a 2 jsou výstupy obou porovnávačů podobné. Ve většině případů není rozdíl větší než 0.3. Existují však dvojice, u kterých se podobnosti liší výrazněji. Tyto dvojice byly dále zkoumány. Ukázalo se, že současný porovnávač v těchto případech přiřazuje dvojicím podobnost 0. Důvodem je podmínka v algoritmu porovnání. Dvojice, které se délkou liší o více než 30 %, jsou automaticky označené za nepodobné. Měření zaznamenané v tabulce 5.2 ukazuje, že touto podmínkou je ovlivněna drtivá většina odevzdání, jejichž rozdíl v podobnosti je vyšší než 0.5. V ostatních případech to byl naopak nový porovnávač, který dvojici přiřadil podobnost 0. Důvodem byla nízká hranice šumu. V odevzdáních nebyly nalezeny žádné shody délky 9 tokenů nebo více.

V úloze 3 jsou rozdíly v podobnostech vyšší. Úloha 3 obsahuje společný základ. Nový porovnávač dokáže sdílený kód odfiltrovat. Podobnosti, které vykazuje, jsou tudíž o desítky procent nižší. Podezření potvrzuje dodatečné měření, ve kterém nový porovnávač pracuje v režimu bez odfiltrování společného základu (bez OSZ). Výsledky tohoto měření jsou v grafu 5.1 pro úlohu 3 vyznačeny odlišnou barvou. Rozdíly v podobnostech bez OSN jsou velmi blízké těm, které reportuje současný porovnávač.

Vlastnost	Úloha 1	Úloha 2	Úloha 3	Úloha 3 bez OSZ
Počet dvojic (diff $\geq$ 50%)	27 076	201	81 252	3 996
Z toho porovnávač přiřadil 0	27 063	201	31	3 996
Z toho kvůli podmínce	27 063	201	31	3 996

Tabulka 5.2 Počet dvojic ovlivněných podmínkou v současném porovnávači.



Obrázek 5.1 Srovnání obou porovnávačů.

5.3 Test skrytého opisování

Test skrytého opisování srovnává odolnost obou porovnávačů vůči specifickým úpravám v kódu. Pro účely testování byl vytvořen malý dataset opsaných programů. Programy v datasetu vznikly postupnou úpravou zdrojového kódu, který byl vybrán jako zástupce odevzdání z úlohy 3. Během úprav byl kladen důraz na zachování sémantické ekvivalence. Všechny opsané programy jsou validní a mají stejný výstup. Opsané programy jsou následně porovnány s originálem.

Výsledky testování jsou zaznamenány v tabulce 5.3. Prováděné změny byly zřetězeny. V momentě, kdy byly měněny názvy identifikátorů byly již upraveny komentáře a pozměněno formátování. V první řadě je originál porovnán se sebou samým. Tento scénář pokrývá situaci, kdy student odevzdá opsané řešení, aniž by v něm provedl vlastní úpravy. Následující úpravy postupují od triviálních k pokročilejším. Poslední test, záměna řídících struktur, zahrnuje převracení podmínek, prohazování logických větví podmíněných příkazů a výměnu příkazů pro smyčky.

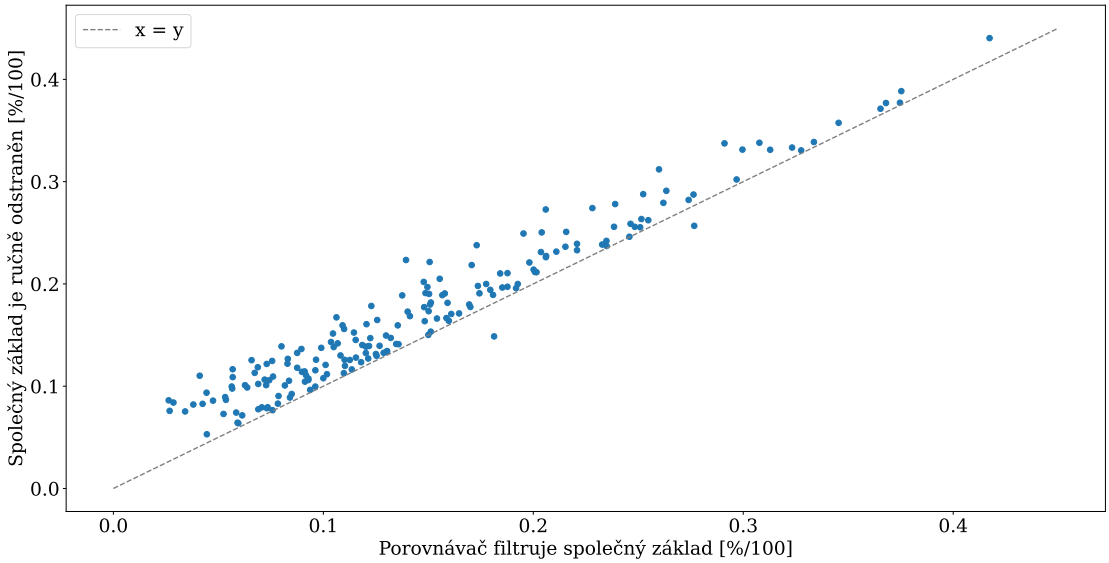
	ProgTest	Nový porovnávač
Transformace	Podobnost [%]	
Originál	100.0	100.0
Navíc upravené formátování	100.0	100.0
Navíc upravené komentáře	100.0	100.0
Navíc přejmenované identifikátory	98.1	100.0
Navíc proházené bloky	37.6	95.1
Navíc záměna řídících struktur	36.4	90.6

Tabulka 5.3 Výsledky srovnání porovnávačů na datasetu úmyslně opsaných programů.

5.4 Test eliminace společného základu

V sekci Test porovnání se současným řešením jsme si ukázali, že odfiltrování společného základu může výrazně ovlivnit výsledky porovnání. Je však potřeba otestovat, že odfiltrování společného základu funguje správně. Pro tyto účely byl vybrán vzorek dvaceti odevzdání z úlohy 3. Odevzdání byla nejdřív porovnána každé s každým v režimu s odfiltrováním společného základu. Následně byl ze všech dvaceti odevzdání ručně odstraněn sdílený kód. Takto modifikovaná odevzdání byla následně porovnána každé s každým v režimu bez odfiltrování společného základu.

Výsledky porovnání obou běhů lze vidět na obrázku 5.2. Na vodorovné ose najdeme výsledky porovnání v režimu s odfiltrováním společného základu. Na svislé ose najdeme výsledky porovnání v režimu bez odfiltrování společného základu poté, co byla odevzdání ručně modifikována. Můžeme si všimnout, že rozdíly mezi podobnostmi jsou velmi malé. Výsledky porovnání v režimu bez odfiltrování společného základu jsou obecně trochu vyšší. Nicméně takové výsledky jsou očekávané, ruční odebrání zdrojového kódu totiž není perfektní. Zdrojové kódy jsou zbaveny především dlouhých souvislých bloků, které sdílí se šablonou implementace. Mezi takové bloky patří zejména jednotkové testy a direktivy vkládající hlavičkové soubory. Ve zdrojových kódech nicméně mohou zůstat části předepsaného rozhraní tříd, proto je podobnost mírně vyšší. Rozdíly se však pohybují v jednotkách procent, z výsledků testování lze soudit, že odebrání společného základu funguje správně.



Obrázek 5.2 Výsledky testu eliminace společného základu.

5.5 Test rychlosti

Jedním z nedostatků současného řešení je delší doba běhu. Ta pramení z kvadratické složitosti algoritmu, který ProgTest při porovnání používá. Nový porovnávač aplikuje algoritmus Greedy String Tiling, který má lepší pozorovatelnou časovou složitost. Cílem testu rychlosti je porovnat dobu běhu obou nástrojů.

Obě řešení byla porovnána v sekvenčním režimu. Výsledky tudíž nejsou saturovány rozdíly v implementaci vícevláknového režimu. Měření probíhalo pomocí utility `GNU time` na operačním systému macOS s architekturou x86-64. Programy běžely v neizolovaném prostředí – na systému byly spuštěny další procesy. Oba porovnávače byly sestaveny překladačem Clang s vysokou úrovní optimalizace `-O3`.

Porovnávač	Úloha 1	Úloha 2	Úloha 3
ProgTest	2 610.56 s	1 584.06 s	18 559.47 s
Nový porovnávač	2 500.25 s	888.85 s	5 601.76 s
Nový porovnávač bez OSZ	N/A	N/A	4 100.42 s

Tabulka 5.4 Výsledky porovnání rychlostí obou porovnávačů.

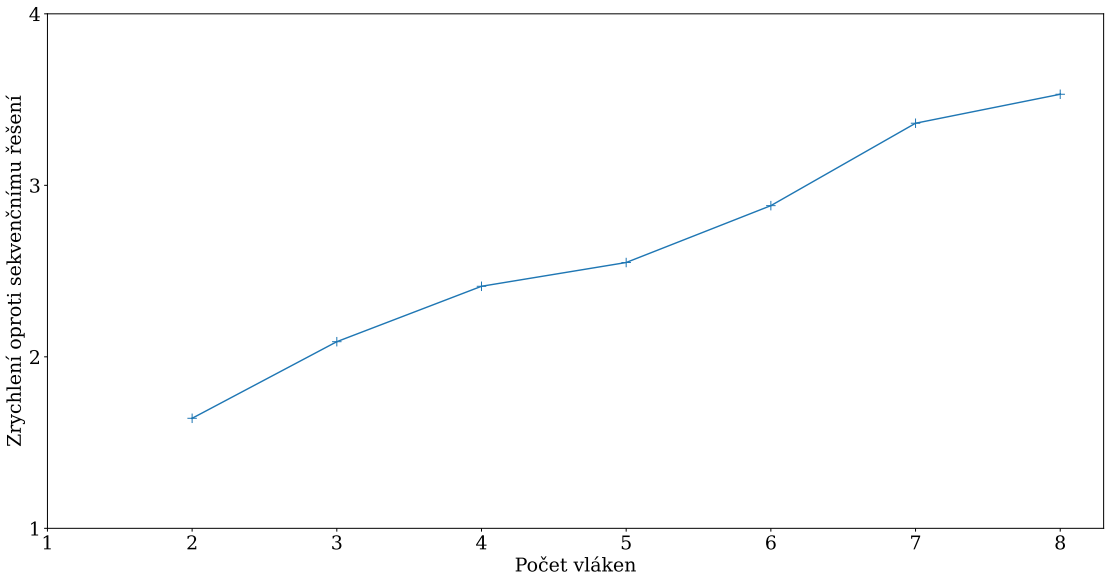
Výsledky měření jsou zaznamenány v tabulce 5.4. Časy pro zpracování úlohy 1 jsou kompetitivní. Programy v této úloze jsou velmi krátké, čítají pouze pár set tokenů. Rozdíly mezi kvadratickou složitostí porovnávače ProgTest a očekávanou lineární složitostí nového porovnávače nejsou na takto krátkých programech patrné. Začnou se však projevovat u úloh 2 a 3. Počet odevzdání v těchto sadách je nižší, programy jsou však delší a mohou čítat tisíce až desetitisíce tokenů. Naměřená doba běhu se významně liší, u poslední úlohy je nový porovnávač téměř 3krát až 4krát rychlejší.

Za zmínku stojí také rozdíly mezi novým porovnávačem v režimu s odfiltrovným společným základem a bez odfiltrovného společného základu. V podkapitole Odfiltrování společného základu jsme si ukázali, že eliminace sdíleného kódu je dosaženo opakovaným spuštěním algoritmu Greedy String Tiling. Dalo by se očekávat, že v režimu s OSZ bude nový porovnávač 2krát až 3krát pomalejší. Naměřené časy jsou však překvapivě podobné. Pravděpodobný důvod je ten, že soubor

se společným základem bývá zpravidla krátký a jeho části jsou ve studentských odevzdáních obsaženy jako celé bloky. Greedy String Tiling preferuje delší shody před kratšími, tyto velké bloky jsou tudíž rychle nalezeny a algoritmus po pár iteracích skončí. Navíc se tyto velké bloky neúčastní při porovnání dvou odevzdání mezi sebou. Algoritmus tudíž musí zpracovat méně dat.

5.6 Test škálovatelnosti

Test škálovatelnosti sleduje zrychlení, kterého je možné dosáhnout zapojením více vláken. Škálování je vyjádřeno jako zrychlení oproti sekvenčnímu řešení. Testování probíhalo na 4jádrovém procesoru, maximálně lze vytvořit 8 vláken. Výsledky měření lze vidět na obrázku 5.3. Škálovatelnost sleduje semi-lineární trend, není však perfektní. Důvodem může být horší rozdělení práce. Zrychlení je nicméně stále několikanásobné, doba běhu při zapojení 8 vláken je téměř 4krát kratší.



Obrázek 5.3 Škálovatelnost nového porovnávače.

Závěr

Cílem této bakalářské práce bylo vylepšit vyhledávání opsaných programů pro systém ProgTest. Byl naimplementován nový porovnávač, který využívá algoritmus Running Karp-Rabin Greedy String Tiling. Nový porovnávač lze také spustit v režimu více vláken. Porovnávač byl následně otestován na třech sadách studentských programů, které byly odevzdány jako řešení domácích cvičných úloh v uplynulých letech.

Nový porovnávač dokáže odhalit rafinovanější způsoby opisování, např. prohození funkcí a tříd. Dále umí odfiltrovat sdílený kód, který je studentům distribuován spolu se zadáním domácích úloh, a zpřesnit tím kvalitu detekce. Testování ukázalo, že nový porovnávač může být v sekvenčním režimu 3krát až 4krát rychlejší než současné řešení. Zapojením více vláken lze dále několikanásobně zkrátit dobu běhu pro velká data.

Všechny cíle byly splněny, práci však lze v budoucnosti dále rozšířit. Vhodným doplňkem by byla např. lepší vizualizace výsledků. Nový porovnávač určuje pouze míru podobnosti. Algoritmus, který nový porovnávač používá, lze však aplikovat pro nalezení a zvýraznění konkrétních částí kódů, které se zdají být opsané. Taková vizualizace by usnadnila vyučujícím práci při finální revizi.

Bibliografie

1. WAGNER, Neal R. Plagiarism by student programmers. *The University of Texas at San Antonio Division Computer Science San Antonio, TX*. 2000, roč. 78249. Dostupné také z: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1082.6313&rep=rep1&type=pdf.
2. MÁJ, Petr. *Originality Checking System*. 2009. Dipl. pr. Faculty of Electrical Engineering CTU in Prague.
3. *JPlag: Detecting Software Plagiarism and Collusion since 1996*. [B.r.]. Dostupné také z: https://github.com/jplag/JPlag.
4. COSMA, Georgina; JOY, Mike. Towards a Definition of Source-Code Plagiarism. *IEEE Trans. Educ.* 2008, roč. 51, č. 2, s. 195–200. ISSN 1557-9638. Dostupné z DOI: 10.1109/TE.2007.906776.
5. AHO, Alfred V; LAM, Monica S; SETHI, Ravi; ULLMAN, Jeffrey D. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007. Dostupné také z: https://web-app.usc.edu/soc/syllabus/20111/30086.pdf.
6. NYSTROM, Robert. *Crafting Interpreters*. Genever Benning, 2021. ISBN 9780990582946.
7. PRECHELT, L; MALPOHL, G; PHILIPPSEN, M. Finding plagiarisms among a set of programs with JPlag. *J. Univers. Comput. Sci.* 2002. ISSN 0948-695X. Dostupné také z: http://jucs.org/jucs_8_11/finding_plagiarisms_among_a/Prechelt_L.pdf.
8. SHAW, Mary; JONES, Anita; KNUEVEN, Paul; MCDERMOTT, John; MILLER, Philip; NOTKIN, David. Cheating Policy in a Computer Science Department. *SIGCSE Bull.* 1980, roč. 12, č. 2, s. 72–76. ISSN 0097-8418. Dostupné z DOI: 10.1145/989253.1165253.
9. LANCASTER, Thomas; CULWIN, Fintan. A Comparison of Source Code Plagiarism Detection Engines. *Computer Science Education*. 2004, roč. 14, č. 2, s. 101–112. ISSN 0899-3408. Dostupné z DOI: 10.1080/08993400412331363843.
10. NOVAK, Matija; JOY, Mike; KERMEK, Dragutin. Source-code Similarity Detection and Detection Tools Used in Academia: A Systematic Review. *ACM Trans. Comput. Educ.* 2019, roč. 19, č. 3, s. 1–37. Dostupné z DOI: 10.1145/3313290.
11. GREGOR, Filip. *Výpočet softwarových metrik*. České vysoké učení technické v Praze. Výpočetní a informační centrum., 2021. Dostupné také z: https://dspace.cvut.cz/handle/10467/95154. Dis. pr.
12. OTTENSTEIN, K J. An algorithmic approach to the detection and prevention of plagiarism. *SIGCSE Bull.* 1976, roč. 8, č. 4, s. 30–41. ISSN 0097-8418. Dostupné z DOI: 10.1145/382222.382462.

13. GRIER, Sam. A tool that detects plagiarism in Pascal programs. *SIGCSE Bull.* 1981, roč. 13, č. 1, s. 15–20. ISSN 0097-8418. Dostupné z DOI: 10.1145/953049.800954.

14. FAIDHI, J A W; ROBINSON, S K. An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Comput. Educ.* 1987, roč. 11, č. 1, s. 11–19. ISSN 0360-1315. Dostupné z DOI: 10.1016/0360-1315(87)90042-X.

15. DONALDSON, John L; LANCASTER, Ann-Marie; SPOSATO, Paula H. A plagiarism detection system. In: *Proceedings of the twelfth SIGCSE technical symposium on Computer science education*. St. Louis, Missouri, USA: Association for Computing Machinery, 1981, s. 21–25. SIGCSE '81. ISBN 9780897910361. Dostupné z DOI: 10.1145/800037.800955.

16. VERCO, Kristina L; WISE, Michael J. Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. In: *ACM International Conference Proceeding Series*. 1996, sv. 1, s. 81–88. Dostupné také z: https://www.researchgate.net/profile/Michael_Wise/publication/2783307_Software_for_Detecting_Suspected_Plagiarism_Comparing_Structure_and_Attribute-Counting_Systems/%5C-links/00b7d5297044377f8e000000.pdf.

17. HECKEL, Paul. A technique for isolating differences between files. *Commun. ACM.* 1978, roč. 21, č. 4, s. 264–268. ISSN 0001-0782. Dostupné z DOI: 10.1145/359460.359467.

18. TICHY, Walter F. The string-to-string correction problem with block moves. *ACM Trans. Comput. Syst.* 1984, roč. 2, č. 4, s. 309–321. ISSN 0734-2071. Dostupné z DOI: 10.1145/357401.357404.

19. WISE, M J. String similarity via greedy string tiling and running Karp-Rabin matching. *Online Preprint, Dec.* 1993, roč. 119, č. 1, s. 1–17. Dostupné také z: https://www.researchgate.net/profile/Michael_Wise/publication/262763983_String_Similarity_via_Greedy_String_Tiling_and_Running_Karp-Rabin_Matching/%5C-links/59f03226aca272a2500141f4/String-Similarity-via-Greedy-String-Tiling-and-Running-Karp-Rabin-Matching.pdf.

20. WISE, Michael J. YAP3: improved detection of similarities in computer program and other texts. In: *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, s. 130–134. SIGCSE '96. ISBN 9780897917575. Dostupné z DOI: 10.1145/236452.236525.

21. PRECHELT, Lutz; MALPOHL, Guido; PHLIPPSEN, Michael. *JPlag: Finding plagiarisms among a set of programs* [http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.34.3508&rep=rep1&type=pdf]. 2000. Dostupné také z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.34.3508&rep=rep1&type=pdf. Accessed: 2022-5-9.

22. KARP, Richard M; RABIN, Michael O. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.* 1987, roč. 31, č. 2, s. 249–260. ISSN 0018-8646. Dostupné z DOI: 10.1147/rd.312.0249.

23. DALY, Charlie; HORGAN, Jane. Patterns of plagiarism. *SIGCSE Bull.* 2005, roč. 37, č. 1, s. 383–387. ISSN 0097-8418. Dostupné z DOI: 10.1145/1047124.1047473.

24. *Plagiarism Detection* [https://theory.stanford.edu/~aiken/moss/]. [B.r.]. Dostupné také z: https://theory.stanford.edu/~aiken/moss/. Accessed: 2022-5-9.

25. SCHLEIMER, Saul; WILKERSON, Daniel S; AIKEN, Alex. Winnowing: local algorithms for document fingerprinting. In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. San Diego, California: Association for Computing Machinery, 2003, s. 76–85. SIGMOD '03. ISBN 9781581136340. Dostupné z DOI: 10.1145/872757.872770.

26. HIRSCHBERG, D S. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*. 1975, roč. 18, č. 6, s. 341–343. ISSN 0001-0782. Dostupné z DOI: 10.1145/360825.360861.

27. *Stack Overflow*. Is C++ context-free or context-sensitive? [https://stackoverflow.com/questions/14589346/is-c-context-free-or-context-sensitive]. [B.r.]. Dostupné také z: https://stackoverflow.com/questions/14589346/is-c-context-free-or-context-sensitive. Accessed: 2022-5-9.

28. *Stack Overflow*. Clang doesn't compile code but gcc and msvc compiled it [https://stackoverflow.com/questions/60303240/clang-doesnt-compile-code-but-gcc-and-msvc-compiled-it]. [B.r.]. Dostupné také z: https://stackoverflow.com/questions/60303240/clang-doesnt-compile-code-but-gcc-and-msvc-compiled-it. Accessed: 2022-5-9.

29. ESTES, Will. *flex: The Fast Lexical Analyzer - scanner generator for lexing in C and C++*. [B.r.]. Dostupné také z: https://github.com/westes/flex.

Obsah přiloženého média

```
└── src ..... Složka obsahující zdrojové kódy porovnávače
    └── thesis ..... Složka obsahující zdrojové kódy textu práce
```