

Akademia Górniczo-Hutnicza im. Stanisława Staszica
w Krakowie



AGH

**System wykrywania podobieństw kodów
źródłowych w projektach studenckich**

Praca dyplomowa

Jarosław Szczęśniak

Promotor: dr inż. Darin Nikolow

19 maja 2013

Spis treści

1	Wstęp	4
2	Cel i zakres prac	4
3	Teoria kompilacji	7
3.1	Symbole	7
3.2	Wyrażenia regularne	8
3.3	Gramatyka	8
3.4	Analiza leksykalna	9
3.4.1	Skaner	9
3.5	Analiza składniowa	10
3.5.1	Parser	10
3.5.2	Typy parserów	11
4	Wprowadzenie do Asemblera	12
4.1	Instrukcje	12
4.2	Typy danych	13
4.3	Adresowanie	14
4.4	Lista instrukcji	15
5	Metody porównywania	18
5.1	Porównywanie tekstu	18
5.1.1	Algorytm brute-force	18
5.1.2	Algorytm Boyer-Moore	20
5.1.3	Odległość Levenshtein'a	22
5.2	Izomorfizm grafów	24
5.2.1	Algorytm VF2	24
6	Projekt systemu CodeComp	25
6.1	Architektura systemu	25
6.2	Uruchamianie aplikacji	29
6.3	Korzystanie z aplikacji	29

6.4	Opis procesu	30
6.4.1	Preprocessing	30
6.4.2	Analiza	30
6.4.3	Porównywanie	30
6.5	Wyniki	31
7	Wnioski	32
8	Bibliografia	33

1 Wstęp

2 Cel i zakres prac

Celem pracy jest zbudowanie systemu, który będzie wspomagał wykrywanie podobieństw w danych kodach źródłowych. Docelowym zagadnieniem z jakim ma zmierzyć się system jest porównywanie kodów źródłowych napisanych w języku Asembler.

Osiągnięcie celu głównego wymaga zrealizowania szeregu celów cząstkowych:

- zapoznanie się z dostępnymi metodami porównywania tekstu
- zdefiniowanie języka wewnętrznego:
 - symboli leksykalnych
 - gramatyki języka
- implementacja co najmniej jednej metody porównywania
- dostarczenie aplikacji analizującej kod źródłowy
- dostarczenie środowiska do porównywania kodów źródłowych
- dostarczenie przykładowych programów w języku wewnętrznym

Istnieje kilka skutecznych metod porównywania tekstu, poczynając od najprostszej i zarazem najmniej efektywnej metody brute-force, poprzez szybsze funkcje wykorzystujące zróżnicowane metody (np. funkcje hashujące), do algorytmów wykorzystujących deterministyczne automaty skończone.

Jednak w przypadku porównywania kodów źródłowych programów komputerowych problem staje się dużo bardziej złożony i zwykle porównywanie tekstu nie jest najbardziej efektywną metodą, na której należy polegać [1]. Najbardziej popularna klasyfikacja metod służących do wykrywania plagiatów kodów źródłowych uwzględnia sześć różnych podejść, skupiających się na [2]:

- tekście - metoda bardzo szybka, ale może być łatwo oszukana poprzez zmianę nazw identyfikatorów i funkcji,

- tokenach - większość systemów opiera się na tej metodzie, sposób działania taki sam jak podczas porównywania tekstu, jednak uniezależniony od identyfikatorów, komentarzy i białych znaków,
- drzewach składniowych - pozwala na porównywanie kodów na wyższym poziomie, umożliwiając porównanie całych sekwencji kodu, np. wyrażeń warunkowych, czy funkcji,
- Program Dependency Graphs (PDGs) - inaczej zwane również multigrafami - grafy skierowane, które dokładnie odwzorowują logikę i przepływ programu, bardzo złożone i czasochłonne,
- metrykach - wykorzystanie funkcji oceny na podstawie częstości występowania pewnego fragmentu kodu (np. pętli, wywołań funkcji, ilość wykorzystanych zmiennych), niezbyt skuteczna metoda, ponieważ dwa kody wykonujące całkowicie różne rzeczy mogą mieć tą samą ocenę,
- hybrydzie - połączenie dowolnych dwóch powyższych metod.

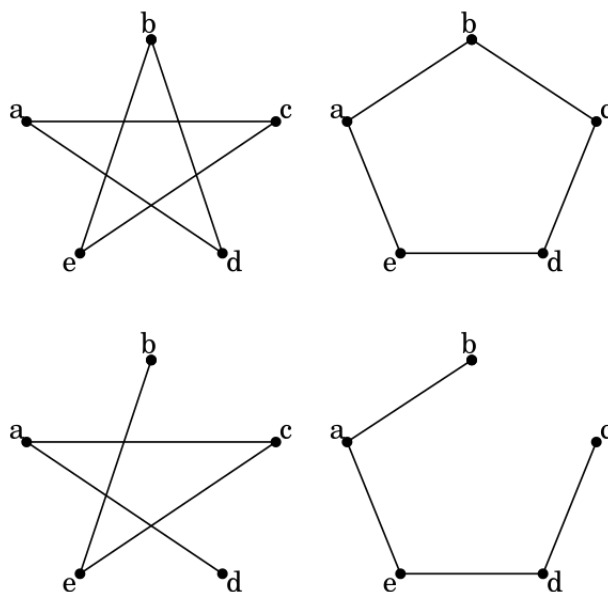
W swojej pracy zdecydowałem się na podejście do problemu pod różnymi kątami i wykorzystanie kilku dostępnych metod.

Oprócz sposobów wymienionych powyżej istnieje jeszcze miara odmienności napisów (także odległość edycyjna, ang. „edit distance” lub „Levenshtein distance”), zaproponowana przez Vladimira Levenshtein’a w 1965 roku[6]. Jest ona skutecznie wykorzystywana w rozpoznawaniu mowy, analizie DNA, korekcie pisowni, oraz w rozpoznawaniu plagiatów.

Z tego względu miara odmienności napisów wydaje się idealnym sposobem na zaimplementowanie metody wykrywającej plagiaty na podstawie porównywania tekstu i tokenów.

Na tym jednak rola odległości Levenshtein’a się nie kończy. Ponieważ porównanie samego tekstu nie jest najbardziej skutecznym sposobem na wykrywanie plagiatów w kodach źródłowych, większy sens i sposób na uzyskanie największej dokładności w porównywaniu mają metody skupiające się wokół grafów i drzew (które są szczególnymi przypadkami grafu).

Izomorfizm grafów (a także drzew) jest wykorzystywana do sprawdzania podobieństwa grafów. Grafy A i B nazywamy izomorficznymi wtedy, gdy istnieje bijekcja zbioru wierzchołków grafu A na zbiór wierzchołków grafu B. Innymi słowy oznacza to, że grafy A i B są takie same, jednak poddane pewnej permutacji wierzchołków. Rozstrzygnięcie izomorficzności dwóch grafów jest problemem klasy NP.



Rysunek 1: Grafy położone obok siebie są izomorficzne

Okazuje się, że odległość edycyjna jest z powodzeniem wykorzystywana w badaniu izomorficzności grafów i to wykorzystam ją jako kolejną metodę do wykrywania podobieństw pokrywającą porównywanie drzew składniowych (czyli grafów).

3 Teoria kompilacji

Kompilator to program, który tłumaczy kod źródłowy na kod wynikowy. Składa się on z dwóch głównych etapów - analizy i syntezy.

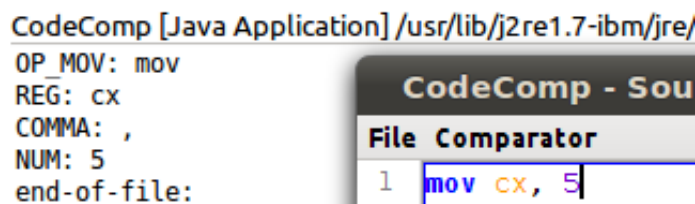
Etap analizy składa się z trzech etapów pośrednich - analizy leksykalnej, składniowej i semantycznej. Polega on na rozłożeniu kodu źródłowego na czynniki składowe oraz na budowie jego reprezentacji pośredniej.

Ten właśnie etap wydaje się być odpowiednim procesem, dzięki któremu można będzie wykonać porównanie dwóch kodów źródłowych na odpowiednio wysokim, abstrakcyjnym poziomie, aby uniezależnić się od wykorzystywanego nazewnictwa lub organizacji i kolejności zdefiniowanych instrukcji w kodach źródłowych.

Język formalny to podzbiór zbioru wszystkich wyrazów nad skończonym alfabetem. Język formalny jest podstawowym pojęciem w informatyce, logice matematycznej i językoznawstwie. Aby zdefiniować język formalny musimy najpierw zdefiniować jego alfabet, składający się z symboli. Ciągi symboli nazywamy napisami, a dowolny zbiór tych ciągów to język formalny.

3.1 Symbole

Symbole leksykalne to abstrakcyjne ciągi znaków, które definiowane są podstawie określonych przez analizator reguł i przekazywane do dalszych etapów analizy. Reguły, według których są one definiowane, budowane są najczęściej z wyrażeń regularnych. Z symbolami leksykalnymi skojarzona jest też wartość leksykalna.



Rysunek 2: Przykład przedstawiający pewne wyrażenie oraz wynik działania parsera w postaci listy tokenów i ich wartości

Symbole, których używany najczęściej to litery lub cyfry. Alfabet jest to niepusty zbiór, składający się ze skończonej liczby symboli. Przykładem alfabetu, może być alfabet języka polskiego lub alfabet Morse'a.

Napis jest to skończony ciąg symboli, które należą do podanego alfabetu. Długość napisu oznaczamy jako $|n|$ i jest to ilość symboli w napisie „n”, przykładowo długość $|\text{tekst}|$ wynosi 5.

Prefiksem napisu nazywamy pewną liczbę symboli rozpoczynających napis. Su-
fiksem nazywamy pewną liczbę symboli kończących napis.

Język jest dowolnym zbiorem napisów ustalonym nad pewnym określonym alfa-
betem. Językiem może być zbiór pusty, zbiór zawierający pusty napis lub pewien
podzbiór ze zbioru wszystkich łańcuchów nad określonym alfabetem.

3.2 Wyrażenia regularne

Wyrażenia regularne to specyficzne wzorce, które umożliwiają zwięzłe i elastyczne
sposoby wyszukiwania dopasowań ciągów tekstu, poszczególnych znaków lub wzor-
ców znaków.

W tej pracy wyrażenia regularne wykorzystane zostaną tylko w jednym miejscu
- w analizatorze leksykalnym. Dzięki nim zdefiniowane zostaną odpowiednie wzorce,
które wychwycą w podanym kodzie źródłowym białe znaki, zwykłe znaki oraz liczby,
z których zbudowane są zmienne, funkcje i identyfikatory.

3.3 Gramatyka

To, co definiuje leksemy, to zestaw reguł określonych na podstawie specyfikacji języka
programowania, czyli gramatyki.

Obecnie najczęściej stosuje się wyrażenia regularne do zdefiniowania reguł. Określają
one zestaw możliwych sekwencji, które są użyte do zbudowania konkretnych lek-
semów.

W niektórych językach programowania kod dzieli się na bloki za pomocą par
symboli (np. „{” i „}”), które razem z wszelkimi białymi znakami również opisane
są za pomocą leksemów, lecz nie są brane pod uwagę podczas dalszych etapów


```

32 LineTerminator      = \r|\n|\r\n
33 WhiteSpace          = {LineTerminator} | [ \t\f]
34
35 Label               = ("."+[:letter:][:digit:]*)([:letter:][:digit:]* ":+)
36 Register            = e?(a|ax|cx|dx|bx|sp|bp|ip|si|di|ss)
37 /* Identifier       = [:letter:]*[\_]*[:digit:]* */
38 Identifier          = [a-zA-Z0-9\_]*
39
40 Number              = {DecLiteral} | {HexLiteral}
41 DecLiteral           = "'"? [:digit:]* "'"?
42 HexLiteral          = (0x)?[0-9a-fA-F:]+(H|h)?
43
44 SingleArithmetic    = aaa|daa|inc|dec
45 Arithmetic          = cmp|add|sub|sbb|div|idiv|mul|imul|sal|sar|rcr|rcr|rol|ror|adc
46 SingleTransfer      = clc|cmc|cld|cli|push|pushf|pusha|pop|popf|popa|stc|std
47 Transfer            = mov|in|out|xchg|sti|cbw|cwd|cwde
48 Misc                = nop|lea|int|rep|repne|repe
49 SingleLogic         = not
50 Logic               = and|or|xor|neg
51 Jump                = call|jmp|je|jz|jcxz|jp|jpe|ret|jne|jnz|jecxz|jnp|jpo
52 Store               = stow|stosb|stosd
53 Compare             = scas|scasb|scasw|scasd

```

Rysunek 3: Lista tokenów oraz wykorzystanych wyrażeń regularnych, które je definiują.

analizy.

3.4 Analiza leksykalna

Analiza leksykalna służy do dzielenia strumienia znaków wejściowych na grupy (symbole), do których dopasowywane są pewne sekwencje (leksemy) na podstawie reguł (wzorców), w celu przekazania ich do dalszego etapu analizy. Leksem to abstrakcyjna jednostka leksykalna, która może obejmować słowa kluczowe, identyfikatory, etykiety, operatory, separatory, komentarze lub inne symbole specjalne.

3.4.1 Skaner

Skaner, czyli podstawowa część analizatora leksykalnego, zajmuje się głównym etapem analizy. Posiada on informacje na temat sekwencji znaków, które mogą być wykryte i zapisane jako symbole.

W wielu przypadkach pierwszy znak, który nie jest znakiem białym może posłużyć do dedukcji, z jakiego rodzaju symbolem mamy do czynienia (longest-match rule). W przypadku bardziej skomplikowanych języków potrzebna jest możliwość cofania

się do wcześniej wczytanych znaków.

Skaner wczytuje kolejne znaki ze strumienia wejściowego i klasyfikuje je według reguł w symbole leksykalne, które następnie przekazywane są do analizy składniowej. Jako jednostka rozpoczynająca analizę i wczytująca tekst źródłowy może służyć również jako filtr, który pomija pewne elementy, takie jak np. komentarze, białe znaki lub znaki nowego wiersza.

Możliwe jest i czasem konieczne napisanie własnego analizatora, aczkolwiek jest to zwykle zbyt skomplikowane, czasochłonne i nieopłacalne. Z tego względu analizatory, zwane lekserami, zwykle generowane są za pomocą specjalnych narzędzi, które przyjmują na wejściu wyrażenia regularne, które opisują wymagane symbole leksykalne.

Jednym z najbardziej popularnych generatorów analizatorów leksykalnych jest program Flex[?], który posiada swoją implementację w Javie o nazwie JFlex[4].

To właśnie JFlex został wykorzystany do wygenerowania skanera na podstawie gramatyki, która składa się z trzech części: kodu użytkownika, zdefiniowanych za pomocą wyrażeń regularnych tokenów oraz definicji reguł leksykalnych.

3.5 Analiza składniowa

Analiza składniowa to proces analizy tekstu, złożonego z symboli, którego celem jest wyklarowanie jego struktury gramatycznej na podstawie zdefiniowanej gramatyki.

3.5.1 Parser

Jest to komponent, który wykonując analizę składniową, sprawdza jej poprawność i tworzy pewną strukturę danych (często tzw. drzewo składniowe), składającą się z symboli wejściowych. Korzysta z analizatora leksykalnego, który zaopatruje go w zestaw symboli leksykalnych zbudowanych ze strumienia wejściowego. Innymi słowy parser analizuje kod źródłowy i tworzy jego wewnętrzną reprezentację.

3.5.2 Typy parserów

Zadaniem parsera jest określenie czy i w jaki sposób symbole leksykalne mogą być przekształcone na symbole gramatyczne. Istnieją dwa sposoby, dzięki którym osiągniemy rządany efekt:

- parsowanie zstępujące (top-down) – polega na znalezieniu symboli wysuniętych najbardziej na lewo przeszukując drzewo składniowe z góry na dół.
- parsowanie wstępujące (bottom-up) – polega na sprawdzeniu począwszy od słowa wejściowego i próbie redukcji do symbolu startowego, analizę zaczynamy od liści drzewa posuwając się w kierunku korzenia.

Do wygenerowania parsera opierającego się na metodzie zstępującej wykorzystany został program BYACC/Jbyaccj. Jest to rozszerzenie generatora BYACC stworzonego na uniwersytecie Berkeley[?], który na podstawie gramatyki opisanej w notacji zbliżonej do BNF (Backus Normal Form)[?] generuje parser w języku Java.

4 Wprowadzenie do Asemblera

Historia asemblera sięga lat 50. tych XX wieku, kiedy powstał pierwszy asembler stworzony przez niemieckiego inżyniera Kondrada Zuse. Stworzył on pierwszy układ elektromechaniczny, który na podstawie wprowadzanych przez użytkownika rozkazów i adresów tworzył taśmę perforowaną zrozumiałą dla wyprodukowanego przez niego komputera Z4.

Asembler jest to program, który tłumaczy kod źródłowy, napisany w języku niskopoziomowym, na kod maszynowy. Proces ten nosi nazwę asemblacji.

Język asembler (ang. **assembly language**) jest to język programowania komputerowego, należący do grupy języków niskopoziomowych. Oznacza to, że jedna instrukcja języka asembler odpowiada dokładnie jednej instrukcji kodu maszynowego, przeznaczonej do bezpośredniego wykonania przez procesor.

Istnieje kilka różnych implementacji języka asembler. Składnia każdego z nich jest zależna od architektury konkretnego procesora. Najbardziej popularnym obecnie asemblerem jest Asembler x86.

4.1 Instrukcje

Każda instrukcja języka asembler jest reprezentowana przez łatwy do zapamiętania tzw. mnemonik, który w połączeniu z jednym lub wieloma argumentami tworzy kod operacji (ang. **opcode**).

Specyfikacja i format operacji zależy architektury procesora (ang **ISA - Instruction Set Architecture**), która definiuje listę dostępnych rozkazów procesora, typów danych, rejestrów, obsługę wyjątków i przerwań.

Argumentami operacji mogą być (w zależności od architektury): rejestry, wartości stosu, adresy w pamięci, porty I/O, itp.

Lista typów operacji składa się z operacji:

- arytmetycznych
- logicznych
- transferowych

- skokowych
- różnych

Asembler x86 posiada dwie podstawowe składnie: Intel i AT&T. Składnia Intel jest dominującą w systemach DOS i Windows, natomiast AT&T w systemach Unixowych[9].

	AT&T	Intel
Porządek argumentów	Parametr źródłowy przed docelowym eax := 5 zapisujemy jako mov \$5, %eax	Parametr docelowy przed źródłowym eax := 5 zapisujemy jako mov eax, 5
Rozmiar argumentów	Mnemoniki na końcu posiadają przyrostek, oznaczający rozmiar argumentu (np. 'q' oznacza qword, 'l' oznacza long) addl \$4, %esp	Wynioskowane z nazwy użytego rejestru (np. rax, eax, ax, al oznaczają odpowiednio qword, long, word, byte) add esp, 4
Typy zmiennych	Argumenty muszą być poprzedzane znakiem '\$', a rejestry znakiem '%'	Asembler automatycznie wykrywa, czy dany argument jest stałą, zmienną, rejestrem, liczbą, itp..
Adresowanie	Ogólna składnia: DISP(BASE,INDEX,SCALE) movl mem_location(%ebx,%ecx,4), %eax	Wykorzystuje zmienne i nawiasy kwadratowe, dodatkowo muszą być użyte słowa kluczowe oznaczające rozmiar argumentu mov eas, dword [ebx + ecx*4 + mem_location]

Rysunek 4: Główne różnice pomiędzy składniami Intel i AT&T[9]

4.2 Typy danych

Typy danych w języku assembler są ściśle związane z pojęciem dyrektyw.

Dyrektywy to instrukcje, które definiują typ danych, ich rozmiar, zasięg i alokują je w pamięci.

Dyrektywa	Typ danych
DB	Byte (8b)
DW	Word (16b)
DD	Doubleword (32b)
DQ	Quadword (64b)
DT	Ten bytes (80b)

Typy danych dzielimy na dwie kategorie: numeryczne i alfanumeryczne. W skład danych numerycznych wchodzi liczby całkowite i zmiennoprzecinkowe. Kody alfanumeryczne wykorzystywane są do przechowywania napisów.

4.3 Adresowanie

Adresowanie określa sposób, w jaki odnosimy się do adresu w pamięci, który określa położenie argumentów rozkazu.

Adresowanie natychmiastowe

Adresowanie natychmiastowe (ang. **immediate addressing**) to najprostsza metoda adresowania, w którym zamiast podawania adresu operandu jego wartość podana jest jawnie w instrukcji. Ten sposób adresowania jest możliwy tylko dla stałych.

```
1 || mov eax, 5
```

Adresowanie bezpośrednie

Adresowanie bezpośrednie (ang. **direct addressing**) polega na podaniu w instrukcji adresu pamięci, w którym znajduje się operand. W ten wygodny sposób można wykorzystać używając zmiennych globalnych, których adres nie zmienia się w trakcie działania programu.

```
1 || mov eax, [0xFFFFFFFF]
```

Adresowanie rejestrowe

Adresowanie rejestrowe (ang. **register addressing**) wykorzystuje rejestr, w którym operand jest przechowywany.

```
1 || mov eax, ebx
```

Adresowanie pośrednie rejestrowe

W adresowanie pośrednie rejestrowe (ang. **indirect addressing mode**) adres operandu zapisany jest w jednym z rejestrów. Oznacza to, że rejestr staje się wskaźnikiem do wartości operandu.

```
1 || mov eax, [ebx]
```

Adresowanie indeksowe

Adresowanie indeksowe (ang. **indexed addressing**) polega na dodaniu do wartości rejestru, w którym znajduje się adres operandu, tzw. offsetu, czyli przesunięcia względem początku bloku danych.

```
1 || mov eax, [0xFFFFFFFF + ebx * 1]
```

Adresowanie indeksowe z wartością bazową

Adresowanie indeksowe z wartością bazową (ang. **based indexed addressing**) występuje tylko w niektórych rodzajach procesorów. Polega na obliczeniu adresu operandu poprzez zsumowanie wartości dwóch rejestrów i dodaniu opcjonalnego przesunięcia. Pierwszy rejestr zawiera adres podstawowy, drugi indeks. Indeks mnożony jest przez rozmiar słowa, a do całości dodawane jest przesunięcie.

```
1 || mov eax, [0xFFFFFFFF + ebx + ecx * 1]
```

4.4 Lista instrukcji

List instrukcji to najważniejszy element wykorzystywany w parserze. Dokładne i odpowiednie zdefiniowanie listy jest kluczowe podczas rozpoznawania tokenów na podstawie kodu źródłowego i stworzenia poprawnego grafu.

- Operacje arytmetyczne bez lub z 1 argumentem:

- `aaa`
- `daa`
- `inc dest`
- `dec dest`

- Operacje arytmetyczne z dwoma argumentami:

- `adc src, dest`
- `add src, dest`
- `cmp src1, src2`
- `[i]div src, dest`
- `[i]mul src, dest`

- *sbb src, dest*
- *sub src, dest*
- *sa[lr] number, dest*
- *rc[lr] number, dest*
- *ro[lr] number, dest*
- Operacje logiczne bez lub z jednym argumentem:
 - *nop*
 - *not dest*
 - *neg dest*
- Operacje logiczne z dwoma argumentami:
 - *and src, dest*
 - *or src, dest*
 - *xor src, dest*
 - *sh[rl] dest, src*
 - *test src₁, src₂*
- Operacje transferu bez lub z jednym argumentem:
 - *clc*
 - *cld*
 - *cli*
 - *cmc*
 - *pop[ad]*
 - *pop[f] dest*
 - *push[ad]*
 - *push[f] src*
 - *st[cdi]*

- Operacje transferu z dwoma argumentami:

- `mov src, dest`
- `movzx src, [dest]`
- `in src, dest`
- `out src, dest`
- `xchg dest, dest`

- Operacje skoku:

- `call dest`
- `jmp dest`
- `jWarunek dest`
- `ret[fn] number`

- Operacje różne:

- `bswap reg`
- `int src`
- `lea dest, src`
- `nop`

5 Metody porównywania

Każda z opisanych metod posiada szereg cech ją charakteryzujących. Są to:

- faza preprocesingu - przygotowania danych wejściowych do analizy
- ilość potrzebnej pamięci dodatkowej
- złożoność czasowa - ilość wykonywanych operacji względem ilości danych wejściowych

5.1 Porównywanie tekstu

5.1.1 Algorytm brute-force

Metoda brute – force, w teorii, jest najskuteczniejszą metodą, ponieważ sukcesywnie sprawdza wszystkie możliwe kombinacje w poszukiwaniu rozwiązania problemu. W praktyce jednak, algorytmy oparte na tej metodzie są niezwykle nieoptymalne, ze względu na czas wykonywania, przez co są rzadko stosowane.

Algorytm polega na sprawdzeniu wszystkich pozycji w tekście pomiędzy znakiem 0 i $n-m$, gdzie m to długość poszukiwanego wzorca, a n to długość tekstu. Algorytm weryfikuje, czy wzorzec zaczyna się na danej pozycji. Jeśli tak to pozycja elementu w tekście zapisywana jest do bufora. Następnie przesuwa wskaźnik w prawo po każdej próbie. Jeśli na kolejnych pozycjach znajdują się wszystkie kolejne elementy z wzorca to do tablicy wynikowej przepisywany jest zawartość bufora. Algorytm może być wykonywany w dowolnym kierunku, od przodu lub od tyłu.

Metoda nie wymaga żadnej fazy przed rozpoczęciem wykonywania algorytmu. Ze względu na charakter metody wymaga dodatkowego miejsca w pamięci. Złożoność czasowa wynosi $O(m*n)$, a ilość operacji potrzebna do wykonania algorytmu wynosi $2n$.

Przykładowy kod źródłowy algorytmu brute – force, wyszukującego określony wzorzec w podanym tekście:

```
1 void BF(char *x, int m, char *y, int n) {  
2     int i, j;  
3     for (j = 0; j <= n - m; ++j) {  
4         for (i = 0; i < m && x[i] == y[i + j]; ++i);  
5         if (i >= m)  
6             OUTPUT(j);  
7     }  
8 }
```

5.1.2 Algorytm Boyer-Moore

Algorytm Boyer-Moore został stworzony w 1977 roku i do tej pory uważany jest za najbardziej optymalny algorytm poszukiwania wzorca w tekście. Dowodem na to jest fakt, że algorytm ten jest zaimplementowany w większości najbardziej popularnych aplikacji z opcją „Znajdź” lub „Zamień”.

Algorytm szuka danego wzorca w tekście porównując kolejne znaki tekstu do wzorca, lecz w odróżnieniu od metody brute-force wykorzystuje informacje zebrane za pomocą funkcji preprocesujących, aby pominąć jak najwięcej znaków w tekście, które na pewno nie pasują do wzorca.

Procedura zaczyna się w miejscu $k = n$, w taki sposób, że początek wzorca P jest wyrównany do początku tekstu T . Następnie poszczególne znaki z P i T są porównywane począwszy od indeksu n w P i indeksu k w T , poruszając się od prawej do lewej strony wzorca P . Porównywanie trwa dopóki znaki w P i T nie są różne lub do momentu dotarcia do początku P (co oznacza, że znaleziono wystąpienie w tekście), a następnie indeks porównywania przesuwany jest w prawo o maksymalną wartość wskazywaną przez zdefiniowane reguły. Czynności są powtarzane do momentu sprawdzenia całego tekstu T .

Reguły przesuwania indeksu definiowane są za pomocą tabel tworzonych za pomocą funkcji preprocesujących.

Przykład implementacji:

```
1 public List<Integer> match() {
2     List<Integer> matches = new LinkedList<Integer>();
3     computeLast();
4     computeMatch();
5
6     int i = text.length() - 1;
7     int j = pattern.length() - 1;
8     while (i >= 0 && i < text.length()) {
9         try {
10             if (pattern.charAt(j) == text.charAt(i)) {
11                 if (j == 0) {
12                     matches.add(i);
13                     j = pattern.length() - 1;
14                 }
15                 j--; i--;
16             } else {
17                 i -= Math.max(match[j], last[text.charAt(i)]
18                     - text.length() + 1 + i);
19                 j = pattern.length() - 1;
20             }
21         } catch (Exception ex) { (...) }
22     }
23     return matches;
24 }
```

5.1.3 Odległość Levenshtein’a

Odległość Levenshtein’a - inaczej odległość edycyjna (ang. **edit distance**) - to metryka napisu, która określa różnicę pomiędzy dwiema sekwencjami znaków. Została wymyślona i zdefiniowana przez rosyjskiego naukowca Vladimira Levenshtein’a w 1965 roku. Jej wynikiem jest minimalna ilość operacji na pojedynczych znakach (wstawienia, usunięcia, podmiany) potrzebnej do zamiany jednego słowa w drugie.

Definicja

Odległość Levenshtein’a pomiędzy napisami a i b określona jest wzorem $lev_{a,b}(|a|, |b|)$, gdzie:

$$lev_{a,b}(i, j) = \begin{cases} \max(i, j) & \text{if } \min(i, j) = 0, \\ \min \begin{cases} lev_{a,b}(i-1, j) + 1 \\ lev_{a,b}(i, j-1) + 1 \\ lev_{a,b}(i-1, j-1) + [a_i \neq b_j] \end{cases} & \text{else} \end{cases}$$

Przykład

Odległość edycyjna pomiędzy słowami *inżynier* i *magister* wynosi 6:

1. inżynier → mnżynier
2. mnżynier → mażynier
3. mażynier → magynier
4. magynier → maginier
5. maginier → magisier
6. magisier → magister

Cechy

Odległość edycyjna posiada kilka cech, które mają zastosowanie w każdym przypadku i warto je wykorzystać w trakcie implementowania algorytmu:

- odległość jest zawsze conajmniej równa różnicy długości dwóch napisów
- maksymalna wartość odległości jest równa długości dłuższego napisu
- odległość jest równa zero wtedy i tylko wtedy, gdy napisy są identyczne
- odległość pomiędzy dwoma napisami jest nie większa niż suma ich odległości pomiędzy trzecim napisem

Odległość Levenshtein'a jest wykorzystana w technikach dopasowania przybliżonego, które polega na znalezieniu takich napisów, które w przybliżeniu pasują do wzorca. Techniki te znajdują zastosowanie w korektorach pisowni, systemach rozpoznawania znaków (OCR - Optical Character Recognition) oraz w systemach do wykrywania plagiatów.

Głównym elementem w systemach wykrywania plagiatów jest zawsze graf. Może on posiadać postać drzewa, grafu skierowanego i nieskierowanego lub multigrafu. Stanowi on reprezentację kodów źródłowych, które porównujemy. Aby ocenić, w jakim stopniu dwa kodu źródłowe są do siebie podobne, musimy sprawdzić w jakim stopniu podobne są do siebie dwa grafy.

Temat podobieństwa grafów jest istotnym elementem podczas badań w przestrzeni rozpoznawania wzorców. Możliwość określenia dokładnego podobieństwa pomiędzy grafami jest kluczowa, a odległość edycyjna stała się podstawą całego procesu [10].

5.2 Izomorfizm grafów

Problem izomorficzności grafów polega na określeniu, czy dwa grafy są izomorficzne, czyli identyczne. Poza praktycznym zastosowaniem w wielu różnych dziedzinach, zagadnienie to jest ciekawe i, z punktu widzenia teorii złożoności obliczeniowej, uznawane za jedno z niewielu problemów decyzyjnych należących do klasy NP, których rozwiązania nie można zweryfikować w czasie wielomianowym oraz jednym z dwóch problemów, których złożoność pozostaje nierozstrzygnięta[12].

Jednocześnie problem izomorficzności dla szczególnych rodzajów grafów może być rozwiązany w czasie wielomianowym i w optymalny sposób. Te szczególne klasy grafów to:

- drzewo[13]
- graf planarny[14]
- graf przedziałowy[16]
- graf permutacji[17]
- niepełne drzewo rzędu k [18]
- graf o ograniczonym stopniu[19]
- graf o ograniczonym genus (rodzaj topologiczny)[20]

5.2.1 Algorytm VF2

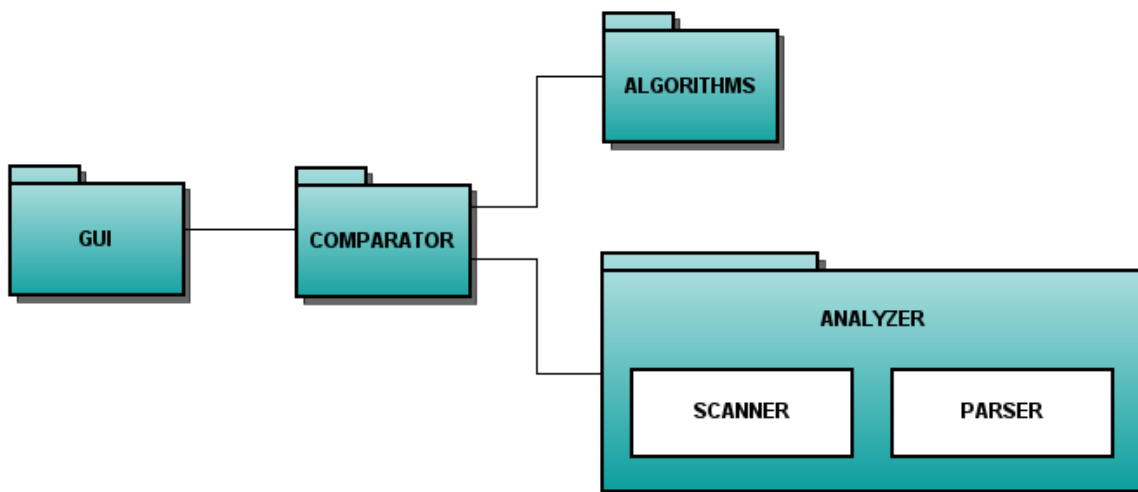
[11]

6 Projekt systemu CodeComp

W tym rozdziale przedstawiono opis aplikacji CodeComp, jej budowę wewnętrzną oraz sposoby instalacji i użycia.

6.1 Architektura systemu

Program składa się z kilku głównych pakietów: Comparator, Algorithm, Analyzer oraz GUI.



Rysunek 5: Diagram przedstawiający strukturę aplikacji

Comparator jest podstawowym, statycznym obiektem, który na wejściu pobiera kody źródłowe programów porównywanych i przekazuje je do analizatora. Analizator zwraca nieskierowane grafy porównywanych kodów źródłowych, które w dalszym etapie przekazane są jako parametry algorytmu wybranej metody weryfikowania podobieństwa.

Wynikiem działania komparatora jest procentowa wartość podobieństwa pomiędzy dwoma kodami źródłowymi oraz graficzna prezentacja miejsc identycznych w porównywanych źródłach.

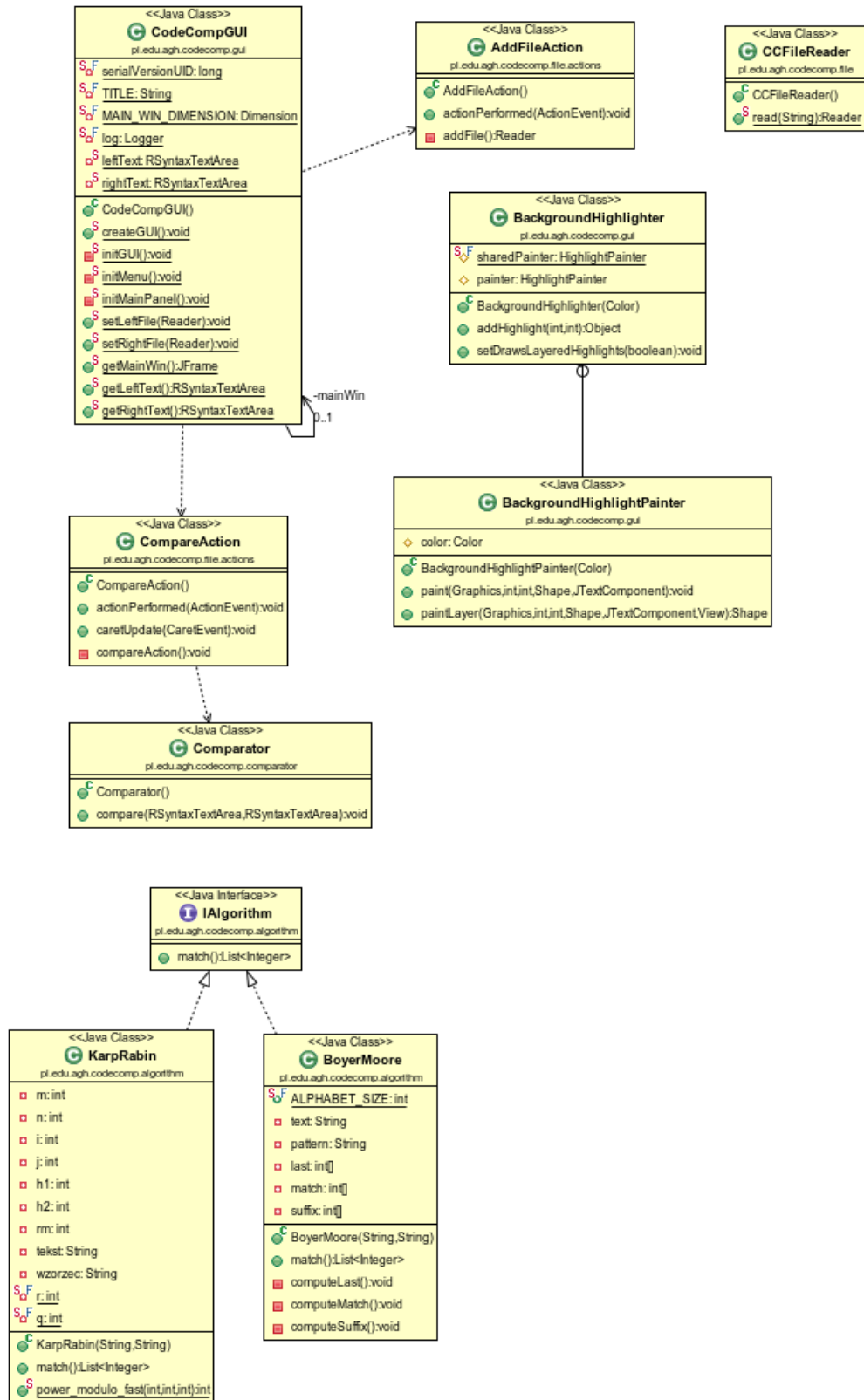
Algorithm to pakiet, w którego skład wchodzi zestaw algorytmów porównywania

tekstu oraz badania izomorficzności grafów.

Algorytmy porównywania tekstu wykorzystane zostały do graficznego zaznaczania fragmentów identycznych w porównywanych kodach źródłowych. Na wejście przyjmują teksty kodów źródłowego w całości, następnie dzielą je na pojedyncze wyrazy i sprawdzają, czy dany wyraz z jednego kodu źródłowego znajduje się w dowolnym miejscu w drugim źródle. Wynikiem działania algorytmów porównywania tekstu jest lista identycznych wyrazów oraz ich położenie w obu źródłach.

Algorytm sprawdzania izomorficzności grafów przyjmuje na wejście dwa grafy skierowane i wykorzystuje algorytm VF2 do obliczenia dystansu pomiędzy nimi. Wynikiem jego działania jest wartość liczbową oraz procentową obliczonej odległości.

Analyzer to komponent, który zajmuje się etapem analizy z teorii kompilacji. Składa się ze skanera - czyli analizatora leksykalnego - oraz z parsera - czyli analizatora składniowego. Wynikiem działania analizatora jest graf nieskierowany, w którym węzły to tokeny a krawędzie obrazują hierarchię w kodzie źródłowy. Graf wynikowy przekazywany jest do komparatora.



Program posiada graficzny interfejs użytkownika stworzony za pomocą biblioteki Swing. Składa się on z dwóch elementów tekstowych wyświetlających wczytane kody źródłowe oraz podświetlający elementy, które są identyczne/podobne. Wczytywanie oraz wszelkie inne operacje dostępne są z poziomu głównego menu znajdującego się na górze okna.

```

1 A small program that calculates and prints terms of the Fibonacci series
2
3 ; fibo.asm
4 ; assemble using nasm:
5 ; nasm -o fibo.com -f bin fibo.asm
6
7 *****
8 ; Alterable Constant
9 *****
10 You can adjust this upward but the upper limit is around 150000 terms.
11 the limitation is due to the fact that we can only address 64k of memor
12 in a DOS com file, and the program is about 211 bytes long and the
13 address space starts at 100h. So that leaves roughly 65000 bytes to
14 be shared by the two terms (num1 and num2 at the end of this file). S
15 they're of equal size, that's about 32500 bytes each, and the 150000th
16 term of the Fibonacci sequence is 31349 digits long.
17
18 maxTerms equ 15000 ; number of terms of the series to calculat
19
20 *****
21 Number digits to use. This is based on a little bit of tricky math.
22 One way to calculate F(n) (i.e. the nth term of the Fibonacci seeries)
23 is to use the equation  $\text{int}(\phi^n/\sqrt{5})$  where ^ means exponentiation
24 and  $\phi = (1 + \sqrt{5})/2$ , the "golden number" which is a constant abou
25 equal to 1.618. To get the number of decimal digits, we just take the
26 base ten log of this number. We can very easily see how to get the
27 base phi log of F(n) -- it's just  $n \cdot \text{lp}(\phi) + \text{lp}(\sqrt{5})$ , where lp means
28 a base phi log. To get the base ten log of this we just divide by the
29 base ten log of phi. If we work through all that math, we get:
30
31 digits = terms * log(phi) + log(sqrt(5))/log(phi)
32
33 the constants below are slightly high to assure that we always have
34 enough room. As mentioned above the 150000th term has 31349 digits,
35 but this formula gives 31351. Not too much waste there, but I'd be
36 a little concerned about the stack!
37
38 digits equ (maxTerms*209+1673)/1000
39
40 this is just the number of digits for the term counter
41 cntDigits equ 6 ; number of digits for counter
42
43

```

6.2 Uruchamianie aplikacji

Aplikacja dołączona do niniejszej pracy dyplomowej w całości napisana została w języku JAVA. Wykorzystuje ona podstawowe pakiety ze środowiska JRE 1.7 oraz kilka dodatkowych, niestandardowych bibliotek (rsyntaxtextarea).

Aplikacja dołączona została w postaci kodu źródłowego oraz skompilowanego archiwum JAR, który można uruchomić na dowolnym środowisku z zainstalowanym pakietem uruchomieniowym JRE w wersji 1.7 za pomocą polecenia:

```
java -jar codecomp.jar
```

6.3 Korzystanie z aplikacji

6.4 Opis procesu

6.4.1 Preprocessing

Pierwszym etapem całego mechanizmu powinna być faza preprocessingu, czyli odpowiedniego przefiltrowania plików źródłowych, celem usunięcia zbędnych białych znaków oraz komentarzy.

Za etap ten odpowiada Filtr, który posiada dwie funkcje: usunąć białe znaki oraz usunąć komentarze.

Pierwsza funkcja usuwa wszystkie białe znaki oraz usuwa puste linie. Druga funkcja usuwa wszystkie komentarze zaczynające się od znaku średnika (;) i znaku zapytania (?).

6.4.2 Analiza

Etap analizy dzieli się na 2 elementarne części:

- analizę leksykalną – wczytywanie znaków i grupowanie ich w większe symbole
- analizę składniową – sprawdzenie poprawności z gramatyką, stworzenie drzewa składniowego

Pierwszym krokiem jest proces analizy leksykalnej. Jej zadaniem jest wyłonienie odpowiednich symboli leksykalnych, które zostaną przekazane do analizatora składniowego.

Kolejnym krokiem jest parsowanie, które odbywa się za pomocą analizatora składniowego. Sprawdza on, czy przekazane symbole wejściowe są zgodne z gramatyką języka oraz buduje z nich graf nieskierowany, który będzie wykorzystywany podczas porównywania kodów źródłowych.

W wyniku całej analizy zbudowany zostanie graf nieskierowany, którego węzły w sposób hierarchiczny będą przedstawiały strukturę porównywanych kodów źródłowych oraz będą przechowywać informacje na temat typów tokenów i ich wartości.

6.4.3 Porównywanie

6.5 Wyniki

7 Wnioski

8 Bibliografia

Literatura

- [1] Plagiarism Detection across Programming Languages - Christian Arwin, M.M. Tahaghoghi - Australia 2006
- [2] Chanchal Kumar Roy and James R. Cordy - A Survey on Software Clone Detection Research - Canada 2007
- [3] Lex - A Lexical Analyzer Generator - M. E. Lesk and E. Schmidt - <http://dinosaur.compilertools.net/>
- [4] The Fast Lexical Analyser Generator - Gervin Klein - <http://jflex.de>
- [5] Lex Yacc, John R. Levine, Tony Mason, Doug Brown - O'Reilly Associates 1992
- [6] Binary codes capable of correcting deletions, insertions, and reversals - V. Levenshtein - Soviet Physics Doklady 1965
- [7] Practical Graph Isomorphism - B.D. McKay - Congressus Numerantium, vol. 30, pp. 45-87, 1981
- [8] Compiler Basics. Extended Backus Naur Form. - Farrell, James A. - 1995
- [9] Linux assemblers: A comparison of GAS and NASM - Ram Narayan - IBM Developerworks paper - 2007
- [10] A survey of graph edit distance - Xinbo Gao, Bing Xiao, Dacheng Tao, Xuelong Li - London 2009
- [11] A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs - Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, Mario Vento - IEEE Transactions on pattern analysis and machine intelligence, vol. 26, no. 10, October 2004
- [12] Computers and Intractability: A Guide to the Theory of NP-Completeness - Michael Garey, David S. Johnson - USA 1979

- [13] P.J. Kelly, A congruence theorem for trees" *Pacific J. Math.*, 7 (1957) pp. 961–968; Aho, Hopcroft Ullman 1974.
- [14] Hopcroft, J. E., John; Wong, J. (1974), Linear time algorithm for isomorphism of planar graphs", *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*,
- [15] Datta, S., Limaye, N., Nimbhorkar, P., Thierauf, T., Wagner, F. (2009). "Planar Graph Isomorphism is in Log-Space". 2009 24th Annual IEEE Conference on Computational Complexity. p. 203.
- [16] Booth, Kellogg S.; Lueker, George S. (1979), A linear time algorithm for deciding interval graph isomorphism", *Journal of the ACM* 26 (2)
- [17] Colbourn, C. J. (1981), On testing isomorphism of permutation graphs", *Networks* 11: 13–21
- [18] Bodlaender, Hans (1990), "Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees", *Journal of Algorithms* 11 (4): 631–643
- [19] Luks, Eugene M. (1982), "Isomorphism of graphs of bounded valence can be tested in polynomial time", *Journal of Computer and System Sciences* 25: 42–65
- [20] Miller, Gary (1980), "Isomorphism testing for graphs of bounded genus", *Proceedings of the 12th Annual ACM*