Akademia Górniczo-Hutnicza im. Stanisława Staszica

w Krakowie


# System wykrywania podobieństw kodów źródłowych w projektach studenckich

Praca dyplomowa

Jarosław Szczęśniak

Promotor: dr inż. Darin Nikolow

7 czerwca 2013

# Spis treści

# 1 Wstęp

Plagiat jest to kradzież, przywłaszczenie cudzego pomysłu lub utworu i opublikowanie go pod własnym nazwiskiem. Skopiowaną pracą może być obraz, utwór muzyczny, wiersz, książka lub pomysł[1].

Oprócz wymienionych wyżej, najczęściej kopiowanych prac, również plagiatowanie kodów źródłowych, w dobie wszechobecnego internetu i nieograniczonej wymiany informacji, zaczyna być coraz częściej problemem.

Obecnie istenieje wiele skutecznych metod badania podobieństw w medycynie, grafice czy pracach naukowych. W przypadku kodów źródłowych również powstały już systemy, które pomagają w wykrywaniu plagiatów, jednak są to metody dużo bardziej złożone niż te, które bazują na tekście.

## 1.1 Cel i zakres pracy

Celem pracy jest zbudowanie systemu, który będzie wspomagał wykrywanie plagiatów. 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.

Wymagania funkcjonalne w stosunku do języka wewnętrznego:

- obsługa składni asemblera x86,
- obsługa tworzenia zmiennych,
- obsługa definiowania funkcji,
- obsługa pętli,
- obsługa wyrażeń matematycznych.

Wymagania funcjonalne w stosunku do mechanizmu wykrywania podobieństw:

- zwracanie realnego, czytelnego wyniku analizy dwóch kodów źródłowych,
- obsługa języka wewnętrznego,
- optymalny czas działania,
- odporność na zmianę nazewnictwa,
- odporność na zmianę kolejności wykonywanych instrukcji,

Praca składa się ze wstępu, pięciu rozdziałów oraz spisu literatury.

W rozdziale pierwszym opisuje sposoby definiowania języka naturalnego oraz sposoby jego analizy. Podstawowymi zagadnieniami są: gramatyka języka, analiza leksykalna oraz składniowa, będące częścią teorii kompilacji.

Rozdział drugi poświęcony jest językowi programowania Asembler, który jest głównym punktem stworzonej aplikacji. Znajomość języka, jego składni, typów zmiennych i instrukcji, to podstawa podczas jego analizy.

Rozdział trzeci to opis dostępnych metod porównywania tekstu oraz grafów, zbudowanych na podstawie kodu źródłowego.

W rozdziale czwartym znajduje się opis projektu przykładowej aplikacji CodeComp oraz wyniki jej działania. Przedstawiona jest jej architektura, sposób działania i dokładny opis procesu analizy i porównywania kodów źródłowych.

Podsumowanie pracy znajduje się w rozdziale piątym. Rozdział zawiera uwagi końcowe oraz słabe punkty, czyli możliwości udoskonalenia.

Praca zakończona jest spisem literatury zawierającym 29 pozycji.

## 1.2 Wprowadzenie do problematyki

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 zwykłe porównywanie tekstu nie jest wystarczającą metodą, na której należy polegać[2]. Najbardziej popularna klasyfikacja metod służących do wykrywania plagiatów kodów źródłowych uwzględnia sześć różnych podejść, skupiających się na[3]:

- tekście - metoda bardzo szybka, ale może być łatwo oszukana poprzez zmianę nazw identyfikatorów i funkcji,
- tokenach - większośc systemów opiera sie 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 Dependecy 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.

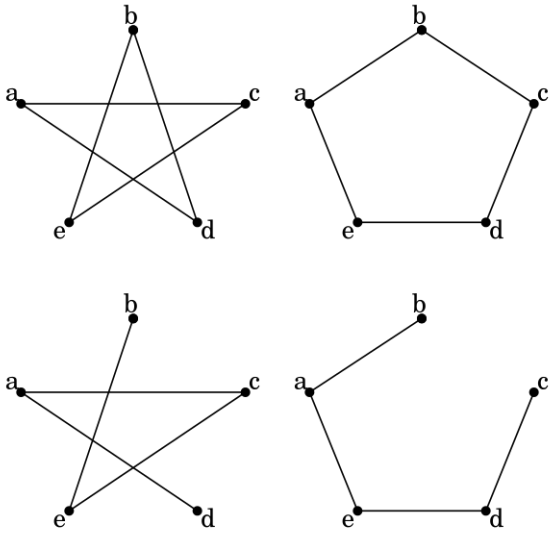
W swojej pracy zdecydowałem się na podejście do problemu pod różnymi kątami i wykorzystanie kilku dostępnych metod.

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[11]. Jest ona skutecznie wykorzystywana w rozpoznawaniu mowy, analizie DNA, korekcie pisowni i innych dziedzinach bazujących na rozpoznawaniu wzorców.

Z tego względu miara odmienności napisów wydaje się idealnym sposobem na zaimplementowanie metody wykrywającej plagiaty – jeden kod źródłowy jest wzorcem poszukiwanym w drugim źródle. Na tym jednak rola odległości Levenshtein'a się nie kończy.

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), dodatkowo wykorzystując odległość edycyjną.

Izomorfizm grafów (a także drzew) jest wykorzystywany do sprawdzania podobieństwa grafów. Grafy A i B nazywami 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

# 2 Teoria kompilacji

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

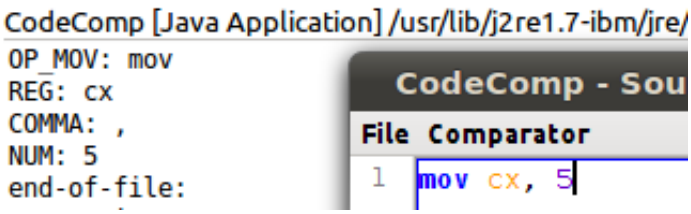
Etap analizy składa się z trzech etapów pośrednich - analizy leksykalnej, składniowej i semantycznej. Polega on na rozłożeniu kodu źródłowego na czynniki składowe oraz na budowie jego reprezentacji pośredniej.

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.

## 2.1 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

Najczęściej wykorzystywane symbole 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 $|tekst|$ wynosi 5.

Prefiksem napisu nazywamy pewną liczbę symboli rozpoczynających napis. Sufiksem nazywamy pewną liczbę symboli kończących napis.

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

## 2.2 Wyrażenia regularne

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

## 2.3 Gramatyka

Podstawowym elementem gramatyki każdego języka jest leksem, czyli wyraz będący abstrakcyjną jednostką każdego języka. To, co definiuje leksemy, to zestaw reguł określonych na podstawie specyfikacji języka programowania, czyli jego gramatyki.

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

W niektórych językach programowania kod dzieli się na bloki za pomocą par symboli (np. „{'' i „}"), które razem z innymi białymi znakami również opisane są

za pomocą leksemów, lecz nie są brane pod uwagę podczas dalszych etapów analizy.

## 2.4 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.

### 2.4.1 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 również pełnić rolę filtra, 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 opisuję wymagane symbole leksykalne.

Jednym z najbardziej popularnych generatorów analizatorów leksykalnych jest program Flex[5], który posiada swoją implementację w Javie o nazwie JFlex[6].

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.

## 2.5 Analiza składniowa

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

### 2.5.1 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.

### 2.5.2 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/J[9]. Jest to rozszerzenie generatora BYACC stworzonego na uniwersytecie Berkeley[8], który na podstawie gramatyki opisanej w notacji zbliżonej do BNF (Backus Normal Form)[13] generuje parser w języku Java.

# 3 Wprowadzenie do Asemblera

Historia asemblera sięga lat 50. XX wieku, kiedy powstał pierwszy asembler stworzony przez niemieckiego inzyniera 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.

## 3.1 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[14].

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

Rysunek 3: Główne różnice pomiędzy składniami Intel i AT&T[14]

## 3.2 Typy danych

Typy danych w języku asembler 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 wchodzą liczby całkowite i zmiennoprzecinkowe. Kody alfanumeryczne wykrzystywane są do przechowywania napisów.

## 3.3 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.

```
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.

```
mov eax, [0xFFFFFF]
```

**Adresowanie rejestrowe**

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

```
mov eax, ebx
```

**Adresowanie pośrednie rejestrowe**

W adresowaniu pośrednim rejestrowym (ang. indirect addressing mode) adres operandu zapisany jest w jednym z rejestrów. Oznacza to, że rejestr staje się wskaźnikiem do wartości operandu.

```
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.

```
mov eax, [0xFFFFFF + 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.

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

## 3.4 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 $src_1$, $src_2$
  - [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 $src_1$, $src_2$

- 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 $dest$
  - j$Warunek$ $dest$
  - ret[fn] $number$
- Operacje różne:
  - bswap $reg$
  - int $src$
  - lea $dest, src$
  - nop

# 4 Metody porównywania

Wykrycie plagiatu polega na porównywaniu. W przypadku tekstu sprawa jest prosta – wystarczy porównywać kolejne znaki w dwóch tekstach. Istnieje wiele metod skupiających się na tekście. Sposób działania jest prosty, kluczową sprawą jest tutaj efektywność i optymalny czas działania.

W przypadku kodów źródłowych sprawa jest bardziej skomplikowana i wykorzystanie algorytmów porównujących tekst jest niewystarczające. Obecnie najbardziej popularnym i efektywnym sposobem jest porównywanie zbudowanych grafów na podstawie struktur kodów źródłowych.

## 4.1 Porównywanie tekstu

Każda z opisanych poniżej 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.

### 4.1.1 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 tekscie 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 tekscie 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 przepisywana 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 został przedstawiony na Listingu 1.

```
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 }
```

Listing 1: Przykład implementacji algorytmu brute-force poszukujący wzorca $y$ w tekście $x$.

### 4.1.2 Algorytm Boyer-Moore

Algorytm Boyer-Moore został stworzony w 1977 roku i do tej pory uważany jest za najbardziej optymalny algorytm poszukiwania wzorca w tekscie. 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 teksćie porównując kolejne znaki tekstu do wzorca, lecz w odróżnieniu od metody brute-force wykorzystuje informacje zebrane za pomocą fukcji preprocesujących, aby pominąć jak najwięcej znaków w tekście, które na pewno nie pasują do wzorca.

Procedura zaczyna się w miejscu $i = n$ (gdzie $n$ to długość tekstu $T$), w taki sposób, że koniec wzorca $P$ jest wyrównany do końca 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 algorytmu został przedstawiony na Listingu 2.

```
1  public List<Integer> match() {
2   List<Integer> matches = new LinkedList<Integer>();
3   computeLast();
4   computeMatch();
5
6   int n = T.length();
7   int m = P.length();
8   int i = n - 1;
9   int j = m - 1;
10   while (i >= 0 && i < n) {
11    try {
```

```
12      if (P.charAt(j) == T.charAt(i)) {
13          if (j == 0) {
14           matches.add(i);
15           j = m - 1;
16      }
17          j--; i--;
18     } else {
19      i -= Math.max(match[j], last[T.charAt(i)]
20          - n + 1 + i);
21      j = m - 1;
22     }
23    } catch (Exception ex) { (...)  }
24   }
25   return matches;
26 }
```

Listing 2: Przykładowa implementacja algorytmu Boyer-Moore wyszukująca wzorzec $P$ w tekście $T$.

### 4.1.3 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 ilośc operacji na pojedynczych znakach (wstawienia, usunięcia, podmiany) potrzebnej do zamiany jednego słowa w drugie.

**Definicja**

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

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

**Przykład**

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

1. **i**nżynier → **m**nżynier
2. m**n**żynier → m**a**żynier
3. ma**ż**ynier → ma**g**ynier
4. mag**y**nier → mag**i**nier
5. magi**n**ier → magisier
6. magis**i**er → magis**t**er

**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
- ogległość 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 zastossowanie 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 reprezentacje 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 [15].

## 4.2 Izomorfizm grafów

Problem izomorfizmu 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[17].

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

- drzewo[18],
- graf planarny[19],
- graf przedziałowy[21],
- graf permutacji[22],
- niepełne drzewo rzędu k[23],
- graf o ograniczonym stopniu[24],
- graf o ograniczonym genus (rodzaj topologiczny)[25].

Izomorfizm grafów bada dokładną zgodność pomiędzy nimi, natomiast podczas wykrywania plagiatów zachodzi również potrzeba sprawdzenia, czy w danym kodzie źródłowym istnieje tylko pewna jego część (np. jedna funkcja), która występuje w drugim źródle.

Problem izomorfizmu grafu może być uogólniony do problemu izomorfizmu podgrafu, który można zastosować do rozwiązania wyżej opisanego problemu.

### 4.2.1 Izomorfizm podgrafu

Problem izomorfizmu podgrafu polega na określeniu, czy dla podanych grafów $G$ i $F$, istnieje taki podgraf grafu $G$, który jest izomorficzny z grafem $F$. Należy on do klasy problemów NP-zupełnych.

Grafem G nazywamy niepusty zbiór $V$ elementów $p$, zwanych węzłami, oraz zbiór $E$ składający się z różnych nieuporządkowanych par węzłów należących do $V$. Pary węzłów należące do $E$ nazywamy krawędziami.

Podgrafem grafu $G$ nazywamy graf, którego wszystkie węzły i krawędzie należą do grafu $G$. Graf $G_\alpha$ jest izomorficzny z podgrafem grafu $G_\beta$ wtedy i tylko wtedy, gdy istnieje zgodność w stosunku 1:1 pomiędzy zbiorami węzłów tego podgrafu i grafu $G_\alpha$, zachowująca sąsiedztwa węzłów[27].

### 4.2.2 Algorytm Ullmann'a

Algorytm Ullmanna to algorytm przeszukiwania w głąb i polega na znalezieniu wszystkich izomorficznych grafów $G_\alpha = (V_\alpha, E_\alpha)$ i podgrafów grafu $G_\beta = (V_\beta, E_\beta)$. Został wymyślony przez Juliana Ullmann'a w 1976 roku[26]. Ze względu na swoją wysoką efektywność jest to wciąż jeden z najczęściej wykorzystywanych algorytmów do dokładnego porównywania grafów.

#### Opis algorytmu

Założenia algorytmu[27]:

- $p_\alpha \in V_\alpha$ - węzły grafu $G_\alpha$,
- $q_\alpha \in E_\alpha$ - krawędzie grafu $G_\alpha$,
- $p_\beta \in V_\beta$ - węzły grafu $G_\beta$,
- $q_\beta \in E_\beta$ - krawędzie grafu $G_\beta$,
- $A = [a_{ij}]$ - macierz sąsiedztwa grafu $G_\alpha$,
- $B = [b_{ij}]$ - macierz sąsiedztwa grafu $G_\beta$,

- $C = [c_ij] = M'(M'B')^T$, gdzie $T - transpozycja$ - spermutowana macierz $B$,
- $M' = p_\alpha X p_\beta$ - macierz, której elementy przyjmują wartość 0 lub 1; każdy wiersz zawiera dokładnie jeden element o wartości 1 i żadna kolumna nie zawiera wiecej niż jeden element o wartości 1; może być wykorzystana podczas permutacji wierszy i kolumn macierzy $B$ w celu wyprodukowania macierzy $C$
- $F = \{F_1, \ldots, F_i, \ldots, F_{p_\beta}\}$ - wektor przechowujący wykorzystane kolumny w trakcie działania algorytmu
- $H = \{H_1, \ldots, H_d, \ldots, H_{p_\alpha}\}$ - wektor przechowujący odległość od korzenia wybranej kolumny
- $H_d = k$, jeżeli $k - ta$ kolumna znajduje się w odległości $d$ od korzenia
- Jeżeli twierdzenie $(\forall i \forall j)(a_{ij} = 1) \Rightarrow (c_ij = 1)$ jest prawdziwe, to $M'$ określa izomorfizm pomiędzy $G_\alpha$ i podgrafem grafu $G_\beta$.

Pierwszym krokiem jest stworzenie $p_\alpha X p_\beta$ elementowej macierzy $M^0 = [{m^0}_{ij}]$, w której:

$${m^0}_{ij} \begin{cases} = 1, & \text{gdy stopień } j - tego \text{ węzła grafu } G_\alpha \geq \text{stopniowi } i - tego \text{ węzła grafu } G_\alpha \\ = 0, & \text{w przeciwnym wypadku.} \end{cases} \quad (0)$$

Następnie, wykonujemy główną część algorytmu:

1. $M = M^0,\ d = 1;\ H_1 = 0;$
   Dla każdego $i = 1, .., p_\alpha$, ustaw $F_i = 0;$
2. Jeśli nie istnieje takie $j$, dla którego $m_{dj} = 1$ i $F_j = 0$ przejdź do kroku nr 7;
   $M_d = M;$
   Jeśli $d = 1$ to $k = H_1$, w przeciwnym wypadku $k = 0$
3. $k = k + 1;$
   Jeśli $m_{dk} = 0$ lub $F_k = 1$ to przejdź do kroku nr 3;
   dla każdego $j \neq k$ ustaw $m_{dj} = 0;$

4. Jeśli $d < p_\alpha$ to przejdź do kroku nr 6, w przeciwnym wypadku sprawdź warunek(0) i wypisz, jeśli znaleziono izomorfizm;

5. Jeśli nie istnieje $j > k$, dla którego $m_{dj} = 1$ i $F_j = 0$ to przejdź do kroku nr 7;
   $M = M_d$;
   przejdź do kroku nr 3;

6. $H_d = k$, $F_k = 1$; $d = d + 1$;
   przejdź do kroku nr 2;

7. Jeśli $d = 1$ to zakończ algorytm,
   $F_k = 0$; $d = d - 1$, $M = M_d$, $k = H_d$;
   przejdź do kroku nr 5;

### 4.2.3 Algorytm VF2

Algorytm VF2 jest deterministycznym algorytmem do dokładnego poszukiwania grafów. W założeniu jest podobny do algorytmu Ullmann'a - bada izomorfizm grafów i podgrafów nie zmieniając ich topologii. Autorzy algorytmu - bazując i porównując go z algorytmem Ullmann'a - skupili się na tym, aby zminimalizować zużycie pamięci. Wykorzystują oni równanie stanu (ang. State Space Representation) w procesie porównywania oraz pięć metod przycinania drzewa przeszukiwań, co pozwala na równoczesne porównywanie syntaktyki i semantyki pary węzłów [16].

#### Opis algorytmu

Proces porównywania grafu $G_1 = (N_1, B_1)$ i grafu $G_2 = (N_2, B_2)$ tworzy odwzorowanie $M$, które składa się z par $(n, m)$, gdzie $n \in G_1$ i $m \in G_2$, z których każda reprezentuje odwzorowanie węzła $n$ z grafu $G_1$ na węzeł $m$ z grafu $G_2$.

Odwzorowanie $M \subset N_1 x N_2$ jest izomorfizmem, jeśli $M$ jest bijekcją, która zachowuje strukturę gałęzi dwóch grafów.

Każdy stan $s$ procesu porównywania może być przypisane do częściowego rozwiązania mapowania $M(s)$, które zawiera tylko pewien podzbiór $M$. $M(s)$ jednoznacznie identyfikuje dwa podgrafy $G_1(s)$ i $G_2(s)$ składające się tylko z węzłów istniejących w $M(s)$ i krawędzi, które je łączą.

Bazując na tych założeniach, przejście ze stanu $s$ to kolejnego stanu $s'$ polega na dodaniu pary $(n, m)$ identycznych węzłów do częściowych grafów skojarzonych ze stanem $s$.

Wśród wszystkich dostępnych stanów SSR, tylko niewielki ich podzbiór jest zgodny z porządanym typem morfizmu, czyli że nie istnieje żaden warunek, który wyklucza możliwość uzyskania kompletnego wyniku. Warunek zgodności mówi o tym, że częściowe grafy $G_1(s)$ i $G_2(s)$ należące do $M(s)$ są izomorficzne.

Algorytm wykorzystuje zestaw reguł sprawdzających warunek zgodności, które umożliwiają wygenerowanie wyłącznie stanów spójnych. Reguły te nazwane są regułami wykonalności.

Dana jest funkcja wykonalności $F(s, n, m) = F_{syn}(s, n, m) \wedge F_{sem}(s, n, m)$, gdzie $F_{syn}$ zależy tylko od struktury grafu, a $F_{sem}$ zależy od atrubutów. Funckja $F(s, n, m)$ jest prawdziwa, jeśli stan po dodaniu pary węzłów $(n, m)$ spełnia wszystkie reguły wykonalności.

```
1  PROCEDURE Match(s)
2   WEJSCIE: posredni stan s; dla poczatkowego stanu s_0  M(s_0) = ∅
3   WYJSCIE: odwzorowanie pomiedzy dwoma grafami
4
5   IF M(s) obejmuje wszystkie wezly grafu G_2 THEN
6    ZWROC M(s)
7   ELSE
8    oblicz zestaw par kandydatow P(s) do dolaczenia do M(s)
9    FOREACH p IN P(s)
10    IF spelniono wszystkie reguly wykonalnosci dla dolaczenia p do
          M(s) THEN
11     oblicz stan s' otrzymany poprzez dolaczenie p do M(s)
12     CALL Match(s')
13    END IF
14   END FOREACH
15   przywroc struktury danych
16  END IF
17 END PROCEDURE Match
```

Listing 3: Pseudokod algorytmu VF2.

### 4.2.4 Odległość edycyjna grafów

Opisane w poprzednich podrozdziałach algorytmy skupiają się na dokładnym porównywaniu grafów. Jednak w przypadku określenia stopnia podobieństwa pomiędzy dwoma kodami źródłowymi, należy skorzystać z metody, która dokonuje niedokładnego porównania grafów [29][30].

Algorytm obliczający odległość edycyjną grafów wykorzystuje metodę węgierską [28] do stworzenia macierzy kosztów. Metoda ta pozwala na rozwiązanie problemu przypisania (w tym wypadku będzie to koszt operacji zastąpienia, dodania lub usunięcia) w czasie wielomianowym i będzie wykorzystana do zwrócenia najmniejszej odległości pomiędzy węzłami grafów.

Metoda ta polega na stworzeniu macierzy $NxM$, gdzie $N$ to liczba węzłów w pierwszym grafie, a $M$ to liczba węzłów w drugim grafie. Macierz podzielona jest na cztery części:

- Lewa górna - zawiera koszt operacji zastąpienia,
- Prawa górna - zawiera koszt operacji usuwania,
- Lewa dolna - zawiera koszt operacji dodawania,
- Prawa dolna - jest wypełniona zerami.

Następnie wykonywane są główne kroki algorytmu:

1. Dla każdego wiersza znajdź najmniejszy element i odejmij go od pozostałych elementów w danym wierszu.
2. Dla każdej kolumny znajdź najmniejszy element i odejmij go od pozostałych elementów w tej kolumnie.
3. Zaznacz wszystkie zera znajdujące się macierzy jak najmniejszą ilością linii.
   Jeśli ilość linii jest mniejsza od $N$ to znajdź najmniejszy element wsród elementów nieoznaczonych i odejmujemy go od wszystkich pozostałych elementów nieoznaczonych, które leżą na przecięciu linii zakreśleń.
   W przeciwnym wypadku, jeśli ilość linii jest równa $N$, przejdź do kroku nr 4.

4. Począwszy od pierwszego wiersza i poruszając się w dół przypisz operację do węzłów, która posiada najmniejszy koszt.

Wynikiem działania algorytmu, jest dwuelementowa tablica wynikowa, która reprezentuje minimalny koszt operacji przekształcenia jeden węzeł w drugi.

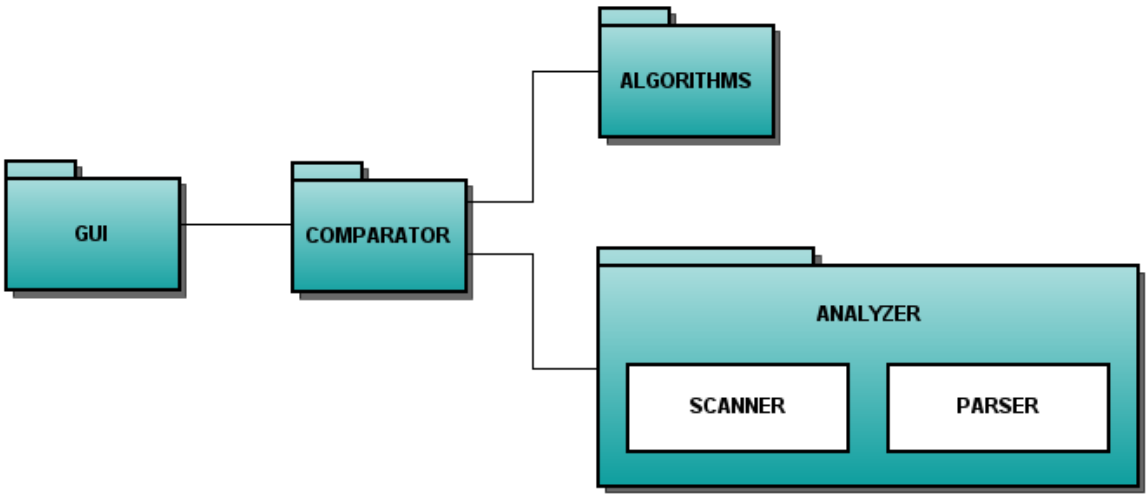
Wartości z całej tablicy są dodawane i otrzymana suma stanowi odległość edycyjną pomiędzy dwoma porównywanymi grafami.

# 5 Projekt systemu CodeComp

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

## 5.1 Architektura systemu

Program składa się z kilku głownych pakietów: Comparator, Algorithm, Analyzer oraz GUI (patrz Rys. 4).




Rysunek 4: Diagram przedstawiający strukturę aplikacji

Comparator jest podstawowym, statycznym obiektem, który na wejściu pobiera kody źrodłowe 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.

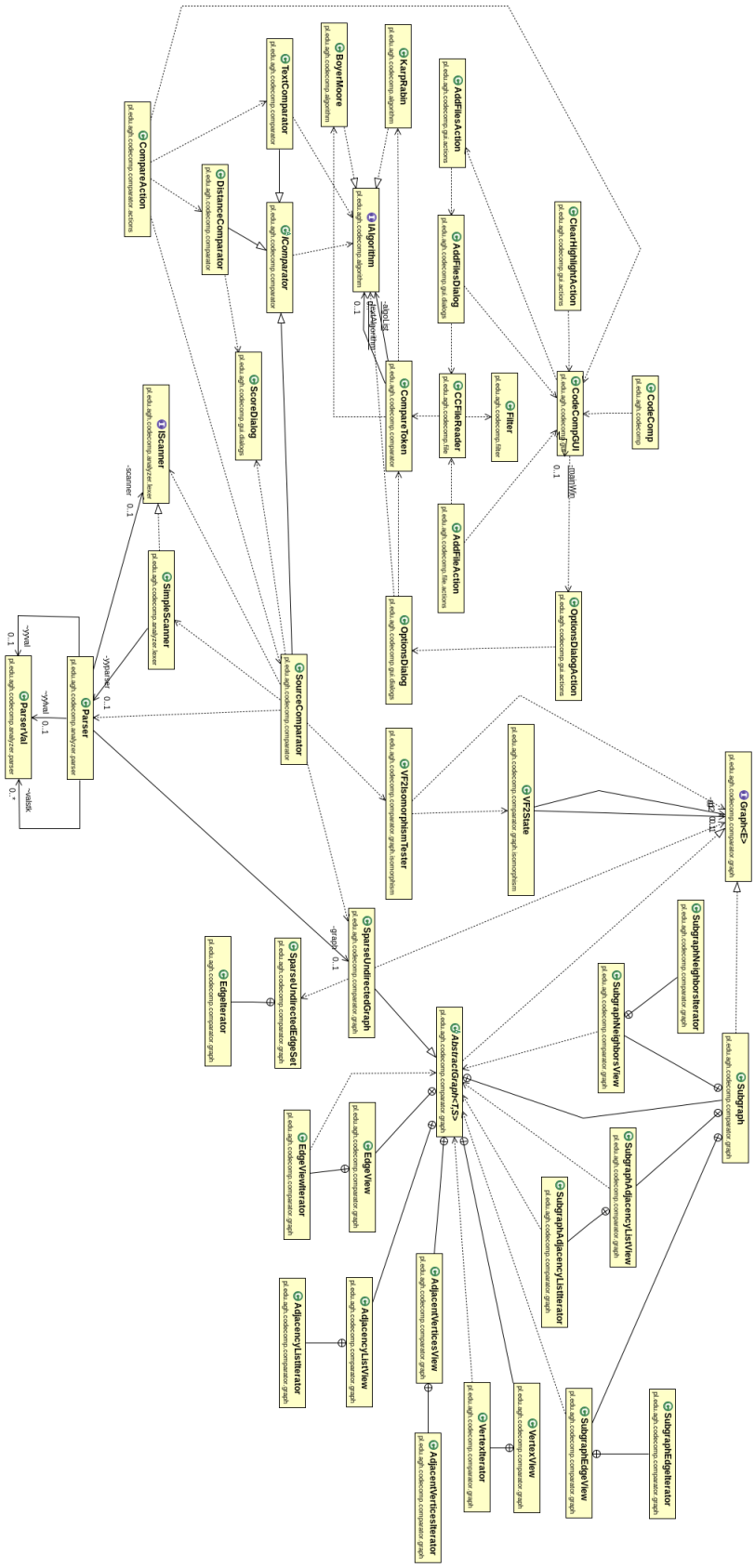
Wynikiem działania komparatora jest procentowa wartość podobieństwa pomiędzy dwoma kodami źródłowymi oraz graficzna prezentacja miejsc identycznych w porównywanych źródłach.

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.

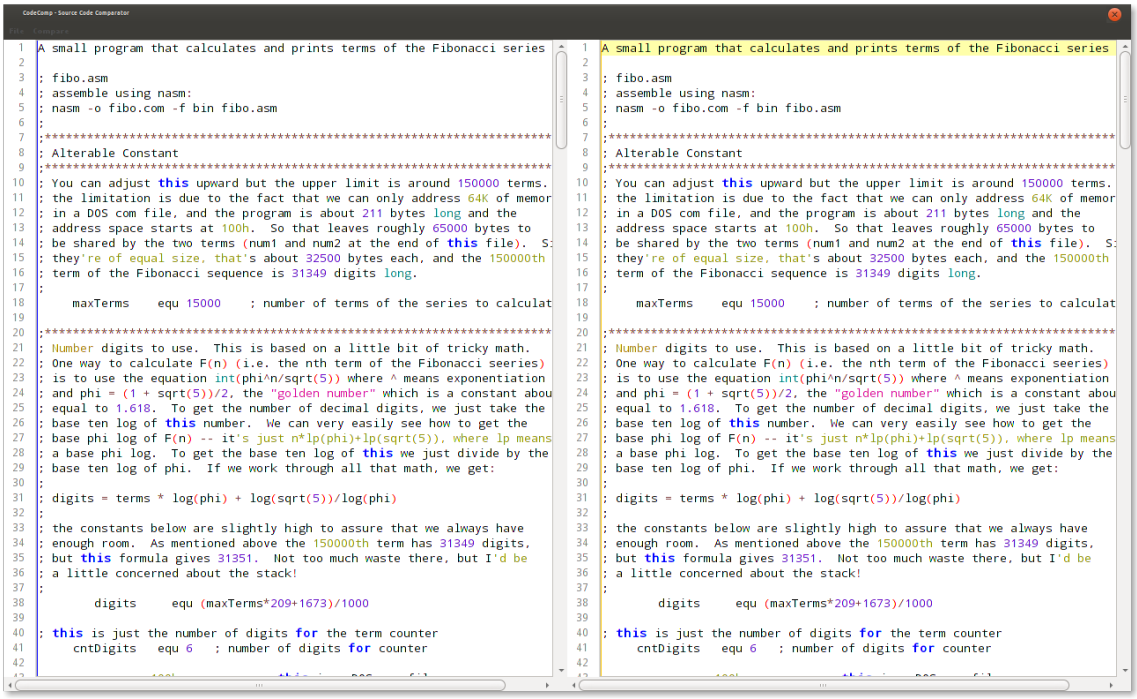
Algorytm obliczania podobieństwa grafów przyjmuje na wejście dwa grafy nieskierowane i oblicza dystans pomiędzy nimi. Pakiet zawiera również algorytm VF2 do sprawdzenia, czy grafy są izomorficzne. Wynikiem jego działania jest wartość liczbowa oraz procentowa obliczonego porównania.

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.




Rysunek 5: Diagram klas aplikacji CodeComp.

Program posiada graficzny interfejs użytkownika stworzony za pomocą bilbioteki 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 w górnej części okna (patrz Rys. 6).




Rysunek 6: Główne okno programu CodeComp.

## 5.2 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:

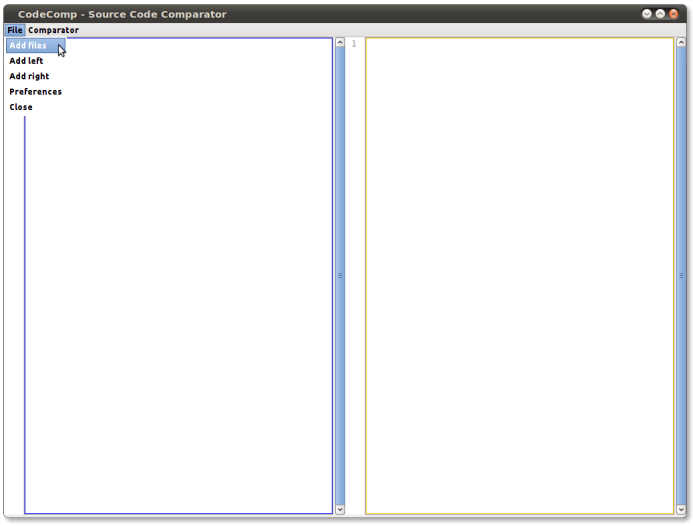
- RSyntaxArea - wykorzystywana do prezentowania kodów źródłowych,
- JFlex - odpowiada za wygenerowanie analizatora leksykalnego (skaner),
- BYacc - odpowiada za wygenerowanie analizatora składniowego (parser).

Aplikacja dołączona została w postaci kodu źrodłowego 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
```

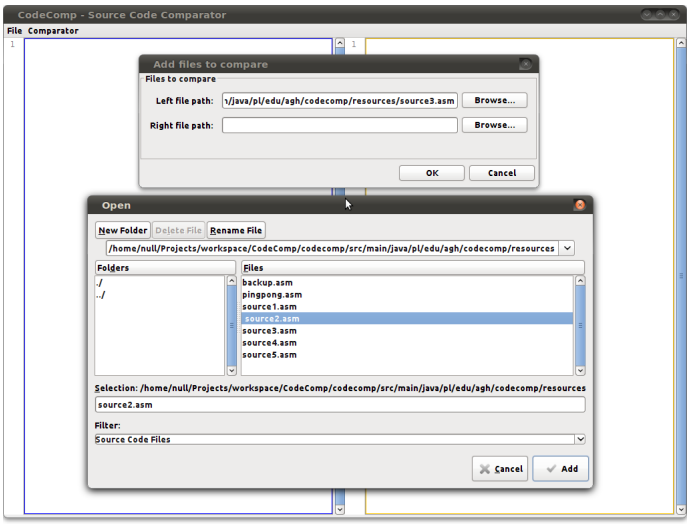
## 5.3 Korzystanie z aplikacji

Po uruchomieniu aplikacji ukazuje nam się główne okno programu. Aby dodać kody źródłowe do porównania należy z głównego menu wybrać *Files* → *Add Files* (patrz Rys. 7).



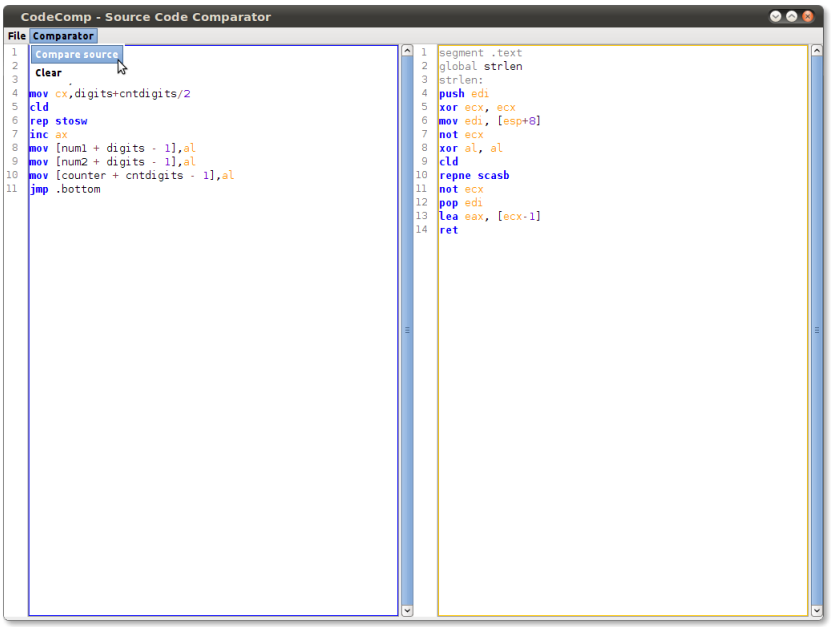

Rysunek 7: Okno startowe aplikacji CodeComp.

W otwartym oknie wybieramy kody źródłowe, które chcemy porównać i klikamy *Ok* (patrz Rys. 8).



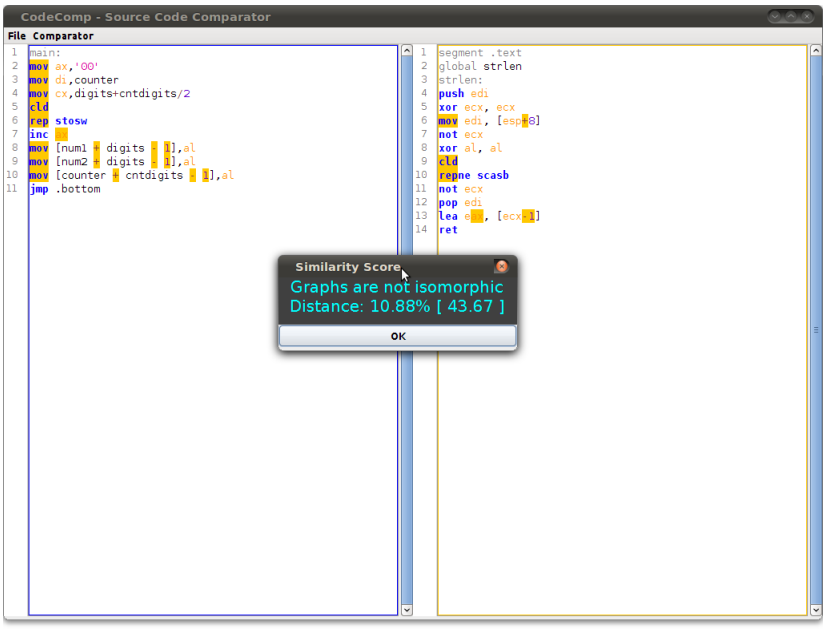

Rysunek 8: Wybór plików w programie CodeComp.

Następnie z menu głównego wybieramy *Compare* → *Compare source* (patrz Rys. 9).




Rysunek 9: Uruchomienie analizy w programie CodeComp.

Wynik powinien pojawić się po chwili w osobnym oknie (patrz Rys. 10).




Rysunek 10: Wynik działania aplikacji CodeComp.

## 5.4 Opis procesu

### 5.4.1 Preprocesing

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

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

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

### 5.4.2 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.

JFlex[6], który został wykorzystany do stworzenia analizatora leksykalnego, generuje pliki źródłowe w języku Java na podstawie pliku specyfikacji, który składa się z trzech części:

- kod użytkownika,
- opcje i deklaracje,
- reguły leksykalne.

W sekcji **Kod użytkownika** implementuje się kod, który w całości zostanie skopiowany i umieszczony w klasie wygenerowanego skanera. Zazwyczaj umieszcza się tu tylko i wyłącznie deklaracje pakietu i wymagane klasy do zaimportowania.

Sekcja **Opcje i deklaracje** jest dużo bardziej złożona. Definiuje się w niej opcje generatora JFlex oraz deklaracje makr (zmiennych). Makro składa się z identyfikatora, operatora przypisania ('=') oraz wyrażenia regularnego. Lista makr w aplikacji CodeComp została przedstawiona na listingu 4.

```
1  LineTerminator = \r|\n|\r\n
2  WhiteSpace = {LineTerminator} | [ \t\f]
3  Label = ("."+[:letter:]+[:digit:]*)|([:letter:]+[:digit:]*":"+)
4  Register = e?(al|ax|cx|dx|bx|sp|bp|ip|si|di|ss)
5  Identifier = [a-zA-Z0-9\_]*
6  Number = {DecLiteral} | {HexLiteral}
7  DecLiteral = "'"? [:digit:]* "'"?
8  HexLiteral = (0x)?[0-9a-fA-F:]+(H|h)?
9  SingleArithmetic = aaa|daa|inc|dec
10 Arithmetic = cmp|add|sub|sbb|div|idiv|mul|imul|sal|sar|rcl|rcr|
       rol|ror|adc
11 SingleTransfer = clc|cmc|cld|cli|push|pushf|pusha|pop|popf|popa|
       stc|std
12 Transfer = mov|in|out|xchg|sti|cbw|cwd|cwde
13 Misc = nop|lea|int|rep|repne|repe
14 SingleLogic = not|neg
15 Logic = and|or|xor
16 Jump = call|jmp|je|jz|jcxz|jp|jpe|ret|jne|jnz|jecxz|jnp|jpo
17 Store = stosw|stosb|stosd
18 Compare = scas|scasb|scasw|scasd
```

Listing 4: Makra zdefiniowane w programie CodeComp.

Ostatnią częścią tej sekcji jest deklaracja stanów leksykalnych, które będą mogły być wykorzystane w następnej sekcji.

**Reguły leksykalne** to sekcja, która składa sie z wyrażeń regularnych i akcji wykonywanych przez skaner po dopasowaniu wyrażenia regularnego. Podczas wczytywania danych wejściowych skaner śledzi wszystkie wyrażenia regularne i wykonuje akcję wyrażenia, które posiada najdłuższe dopasowanie (patrz Listing 5). Jeśli dwa wyrażenia regularne posiadają najdłuższe dopasowanie do aktualnie przetwarzanej danej wejściowej to wykonywana jest akcja przypisana do tego, który jest zdefin-

iowany jako pierwszy w specyfikacji.

W przypadku aplikacji **CodeComp** wykonywane akcje przekazują do parsera zdefiniowane symbole, które będą wykorzystywane jako tokeny podczas analizy składniowej.

```
1  /* operators */
2  "=" { return Parser.EQ; }
3  "," { return Parser.COMMA; }
4  "'" { return Parser.APOSTROPHE; }
5
6  /* operations */
7  {Arithmetic} { return Parser.OP_AR; }
8  {SingleArithmetic} { return Parser.OP_SAR; }
9  {Transfer} { return Parser.OP_MOV; }
10 {SingleTransfer} { return Parser.OP_SMOV; }
11 {Misc} { return Parser.OP_MISC; }
12 {Logic} { return Parser.OP_LOG; }
13 {SingleLogic} { return Parser.OP_SLOG; }
14 {Jump} { return Parser.OP_JMP; }
15 {Store} { return Parser.OP_STO; }
16 {Compare} { return Parser.OP_COMP; }
17
18 /* literals */
19 {Number} { return Parser.NUM; }
20
21 /* identifiers */
22 {Register} { return Parser.REG; }
23 {Label} { return Parser.LAB; }
24 {Identifier} { return Parser.ID; }
25
26 \" { string.setLength(0); yybegin(STRING); }
27
28 /* whitespace */
29 {WhiteSpace} { /* ignore */ }
```

Listing 5: Lista reguł leksykalnych w programie CodeComp.

Dodatkowo w tej sekcji mogą być wykorzystane stany leksykalne. Działają one jak etykiety i oznaczają, które wyrażenia regularne mogą być dopasowywane w trakcie, gdy analizator znajduje się w określonym stanie leksykalnym.

Kolejnym krokiem analizy 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.

Analizator składniowy został wygenerowany za pomocą narzędzia BYacc/J[9] na podstawie pliku specyfikacji, który składa się z trzech sekcji:

- deklaracje,
- reguły,
- kod użytkownika.

W sekcji **Deklaracje**, podobnie jak w przypadku analizatora składniowego, określamy pakiet projektu, w którym znajdzie się parser i importy wymaganych klas. Jednak w przypadku analizatora składniowego sekcja ta ma jeszcze inne, ważniejsze role. Oprócz wcześniej wspomnianych deklaracji określamy tutaj punkt startowy parsera oraz listę tokenów, czyli symboli obsługiwanych przez skaner (patrz Listing 6).

```
1  %{
2  package pl.edu.agh.codecomp.parser;
3  import java.io.IOException;
4  import java.util.HashMap;
5  import no.roek.nlpged.graph.Edge;
6  import org.fife.ui.rsyntaxtextarea.RSyntaxTextArea;
7  import pl.edu.agh.codecomp.graph.SimpleEdge;
8  import pl.edu.agh.codecomp.graph.SparseUndirectedGraph;
9  import pl.edu.agh.codecomp.lexer.IScanner;
10 import pl.edu.agh.codecomp.tree.Node;
11 %}
12 %start input
```

```
13 %token TEXT, NUM, OP_MOV, OP_SMOV, OP_AR, OP_SAR, OP_LOG,
       OP_SLOG, OP_MISC, OP_JMP, OP_STO, OP_COMP, REG, LAB, ID, EQ,
       COMMA, APOSTROPHE
14 %left '-' '+'
15 %left '*' '/'
16 %left NEG
17 %right '^'
```

Listing 6: Sekcja deklaracji analizatora składniowego w programie CodeComp.

Sekcja **Reguły** zawiera reguły gramatyczne, które składają się z identyfikatora, dowolnej liczby tokenów i literałów oraz przypisanej do identyfikatora akcji.

Parser na wejściu otrzymuje symbol leksykalny przekazany ze skanera i weryfikuje, czy jest on zgodny z gramatyką języka zdefiniowaną w tej sekcji. Po dopasowaniu odpowiedniej reguły parser wykonuje przypisaną do niej akcję.

Ostatnia sekcja - **Kod użytkownika** - jest przeznaczona na funkcje opisujące sposób parsowania oraz czynności, jakie użytkownik musi w trakcie działania parsera wykonać.

W aplikacji CodeComp zaimplementowana została funkcja, która wywołuje główną funkcję skanera zwracającą token (patrz Listing 7). Następnie token ten jest przekazywany do funkcji parsujących, które sprawdzają, czy token jest zgodny z gramatyką oraz funkcji pomocniczych (patrz Listing 8), które są wywoływane jako akcje reguł zdefiniowanych w poprzedniej sekcji. Funkcje pomocnicze tworzą odpowiednie węzły na podstawie wykrytych tokenów i dodają je do grafu wynikowego.

```
1  private int yylex() {
2   int tok = -1;
3   try {
4    tok = scanner.yylex();
5    Node node = new Node(yyname[tok], scanner.yytext());
6    allNodes.put(node, count);
7    yylval = new ParserVal(node);
8    count++;
9   } catch (IOException e) {
10    System.err.println(e.getMessage());
```

```
11   }
12   return tok;
13 }
```

Listing 7: Główna funkcja parsera.

```
1 private ParserVal mergedCompute(Node p, Node... child) {
2   for(Node c : child) {
3     p.addChild(c);
4     graph.add(new SimpleEdge(allNodes.get(p),allNodes.get(c)));
5   }
6   return new ParserVal(p);
7 }
```

Listing 8: Funkcja pomocnicza parsera.

W aplikacji CodeComp akcje przypisane do reguł tworzą węzły i krawędzie wynikowego grafu nieskierowanego.

Wynikiem wszystkich etapów analizy jest graf nieskierowany, którego węzły w sposób hierarchiczny będą przedstawiały strukturę porównywanych kodów źródłowych oraz będą przechowywać informacje na temat typów tokenów i ich wartości.

### 5.4.3 Porównywanie

Grafy zbudowane na podstawie porównywanych kodów źródłowych w etapie Analizy zostają odebrane przez moduł Comparator i przekazane jako parametr do algorytmu VF2 badającego izomorfizm oraz do algorytmu nieścisłego porównywania obliczającego odległość edycyjną pomiędzy nimi.

Wykorzystywany graf to graf nieskierowany składający się z węzłów i krawędzi. Węzły w grafie posiadają identyfikator oraz tekst, który będzie porównywany, składający się z tokenu (typu operacji) i jego wartości (konkretnej instrukcji). Krawędzie reprezentują połączenia pomiędzy węzłami. Posiadają identyfikator oraz relację wezłów, które łączą (rodzic → dziecko).

Wynik porównywania zaprezentowany jest użytkownikowi w postaci dwóch kodów źródłowych z zaznaczonymi identycznymi instrukcjami oraz okno dialogowe, które informuje, czy dane grafy są dokładnie takie same (izomorficzne) oraz przedstawia wartość liczbową odległości pomiędzy grafami (im mniejsza wartość liczbowa, a większy procent, tym kody źródłowe są bardziej podobne do siebie).

## 5.5 Wyniki

Do sprawdzenia poprawności działania aplikacji stworzonej na potrzeby projektu wykorzystano pliki źródłowe napisane w języku asembler przez studentów jako zadania na ćwiczenia laboratoryjne.

Aplikacja CodeComp podczas porównywania w założeniu powinna:

- określić realny procent podobieństwa pomiędzy dwoma kodami źródłowymi,
- być odporna na zmiany nazw zmiennych,
- być odporna na zmianę kolejności wykonywanych instrukcji.

**TEST 1**

Pierwszym testem jest porównanie dwóch dokładnie tych samych plików. Plik źródłowych w postaci przedstawionej na Listingu 9. został przeprocesowany do postaci przedstawionej na Listingu 10.

```
1  main:
2  ; initializes the two numbers and the counter.  Note that this
       assumes
3
4   mov     ax,'00'          ; initialize to all ASCII zeroes
5   mov     di,counter               ; including the counter
6   mov     cx,digits+cntDigits/2    ; two bytes at a time
7   cld                      ; initialize from low to high memory
8   rep     stosw            ; write the data
9   inc     ax               ; make sure ASCII zero is in al
10  mov     [num1 + digits - 1],al ; last digit is one
11  mov     [num2 + digits - 1],al ;
12  mov     [counter + cntDigits - 1],al
13
14 jmp      .bottom ; done with initialization, so begin
```

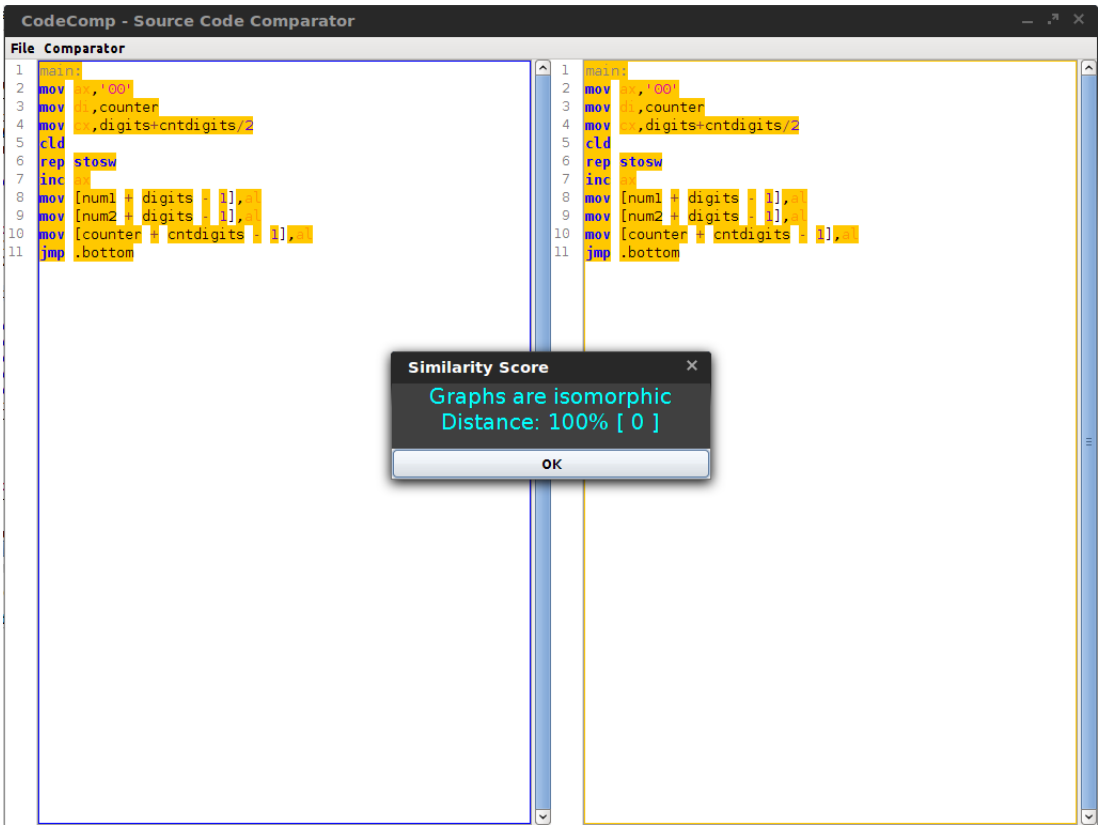
Listing 9: Przykładowy oryginalny kod źródłowy.

```
1  main:
2  mov ax,'00'
```

```
3  mov di,counter
4  mov cx,digits+cntdigits/2
5  cld
6  rep stosw
7  inc ax
8  mov [num1 + digits - 1],al
9  mov [num2 + digits - 1],al
10 mov [counter + cntdigits - 1],al
11 jmp .bottom
```

Listing 10: Przykładowy przeprocesowany kod źródłowy.

Następnie przeprocesowany kod źródłowy oraz jego dokładna kopia zostały porównane ze sobą. Wynik porównania znajduje się na Rysunku 11. Program stwierdził,

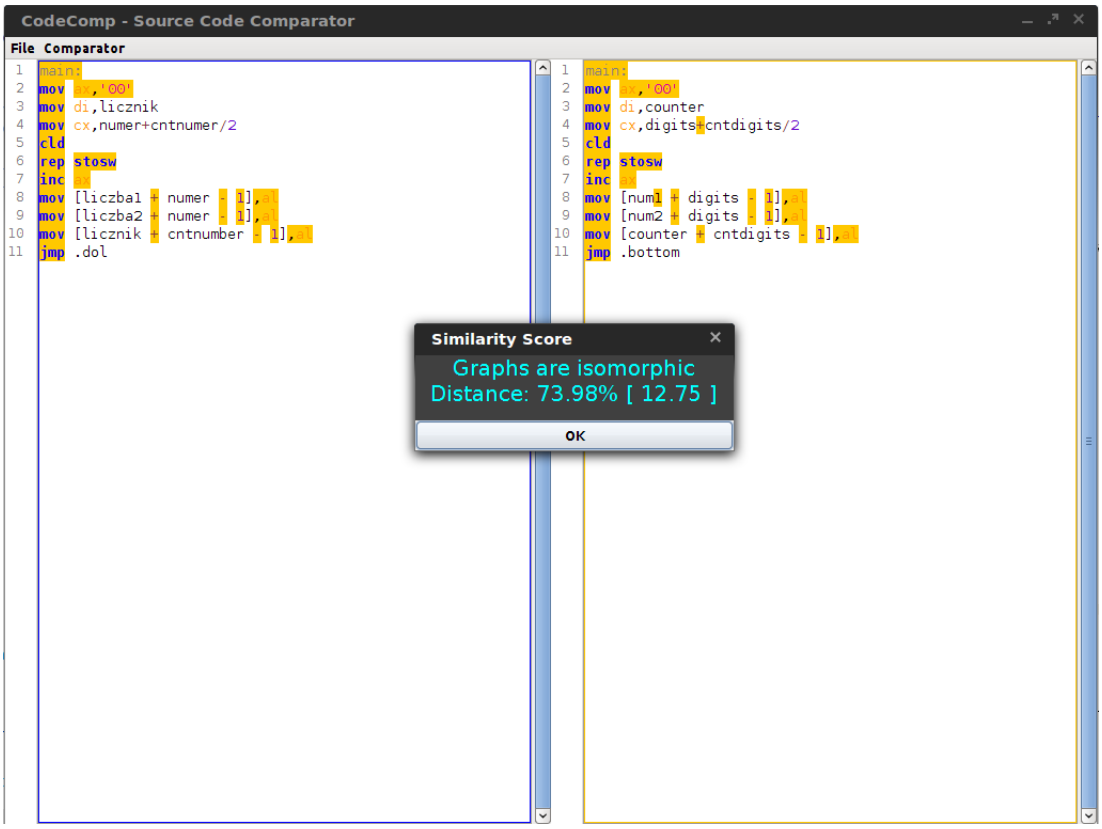



Rysunek 11: Wynik testu nr 1 - dwa identyczne kody źródłowe.

że grafy są izomorficzne oraz obliczył wartość odległości edycyjnej pomiędzy nimi równą 0, czyli wynik zgodny z prawdą.

**TEST 2**

W drugim teście wykorzystano dokładnie ten sam plik źródłowy, jednak zmieniono nazwy zmiennych:



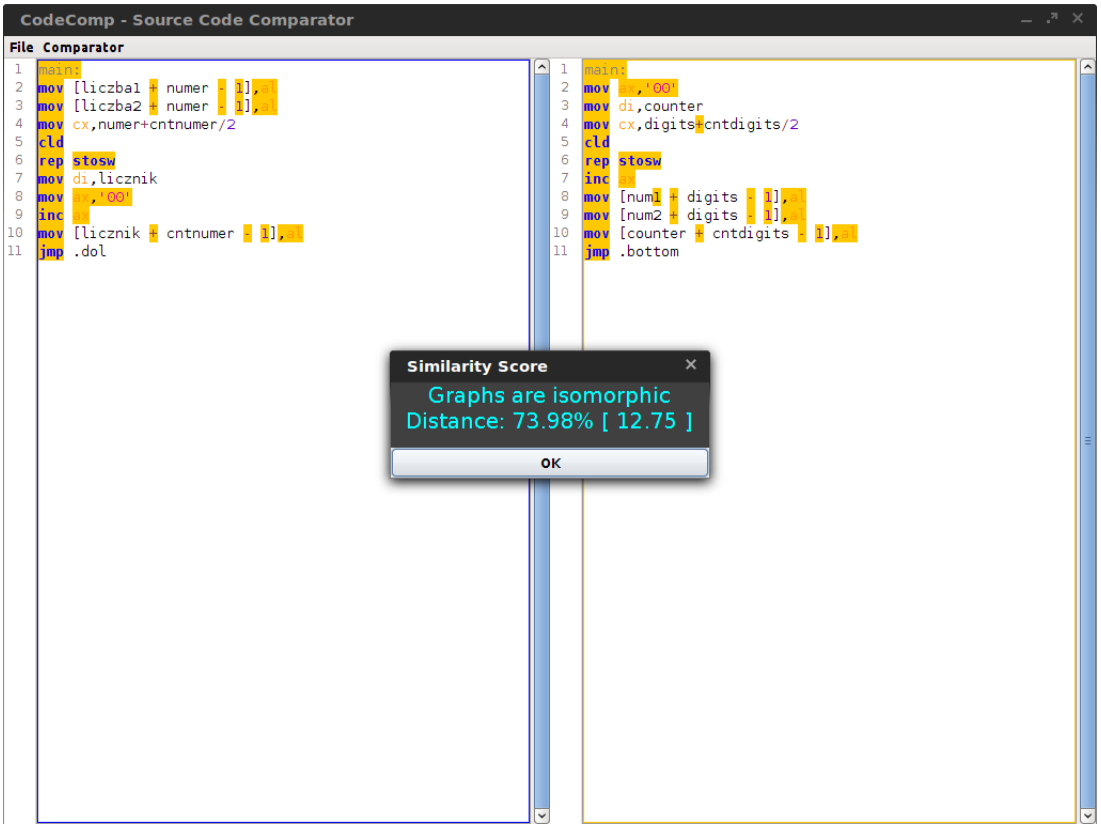

Rysunek 12: Wynik testu nr 2 - ten sam kod źródłowy ze zmienionymi nazwami zmiennych.

W wyniku porównywania (patrz Rys. 12) aplikacja stwierdziła, że grafy obu kodów źródłowych są identyczne, czyli wykonują dokładnie te same instrukcje - można uznać to za pewnik, że jeden z tych kodów źródłowych to plagiat. Posiadają one także bardzo małą wartość odległości edycyjnej (wysoki procent prawdopodobieństwa plagiatu), co oznacza, że wystarczy wykonać niewielką ilość zmian, aby oba kody źródłowe doprowadzić do tej samej postaci.

Test ten udowadnia, że aplikacja jest odporna na zmianę nazw zmiennych.

**TEST 3**

W trzecim teście ponownie wykorzystano ten sam plik źródłowy, jednak tym razem oprócz zmiany nazw zmiennych zmieniono również kolejność wywoływania instrukcji:




Rysunek 13: Wynik testu nr 3 - ten sam kod źródłowy ze zmienionymi nazwami zmiennych i kolejnością wykonywanych instrukcji.

Wynik testu jest dokładnie taki sam, jak w teście numer 2 (patrz Rys. 13). Oznacza to więc, że aplikacja jest odporna na zmiany kolejności wykonywania instrukcji.

### TEST 4

W kolejnym teście z jednego kodu źródłowego skopiowano cztery z dziewięciu funkcji i dodano je w losowej kolejności do zupełnie innego kodu źródłowego. Skopiowane funkcje zajmowały mniej więcej 50% objętości tego kodu.

Procent prawdopodobieństwa plagiatu wyszedł zdecydowanie za niski (patrz Rys. 14), biorąc pod uwagę fakt, że skopiowano około 50% funkcji z jednego kodu do drugiego.


Rysunek 14: Wynik testu nr 4 - porównanie kodu źródłowego z jego 50%-ową kopią.

Stało się tak dlatego, ponieważ parser nie skończył pomyślnie analizowania całego kodu źródłowego. W kodzie źródłowym zostały użyte funkcje i składnia, która nie została zdefiniowana w regułach składniowych parsera.

# 6 Wnioski

Wyszukiwanie wzorców jest zadaniem nietrywialnym. Badania związane z tym tematem sięgają lat 60. ubiegłego wieku i wciąż trwają, dążąc do uzyskania jak najlepszych wyników w najbardziej optymalnym czasie dla coraz większej ilości danych.

Obecnie istnieje już kilka skutecznych metod z powodzeniem wykorzystywanych w systemach wykrywania wzorców w wielu różnych dziedzinach życia - również w informatyce jako systemy wykrywania podobieństw w kodach źródłowych.

Człowiek, znający składnię języka, analizujący dwa kody źródłowe, może w bardzo szybkim czasie określić z dużą pewnością, czy są one do siebie na tyle podobne, że jeden z nich może być plagiatem drugiego.

Dla systemu komputerowego nie jest już to takie proste zadanie. Aby mógł on wykonywać taką funkcję, musimy go najpierw nauczyć czytać tekst, następnie rozumieć kod źródłowy i stworzyć mechanizm, który pozwoli na analityczne porównywanie tego, co czyta.

Głównym celem tej pracy było stworzenie aplikacji, która wczytując dwa kody źródłowe sprawdzi, czy można jeden z nich uznać za plagiat drugiego.

Implementacja programu spełnia wszystkie postawione podstawowe cele i wymagania funkcjonalne. Zdefiniowana gramatyka języka wewnętrznego, chociaż uproszczona, pozwala na czytanie plików źródłowych, wykorzystujących podstawowe instrukcje języka asembler.

Analiza kodu źródłowego to pierwsze miejsce, które zostawia duże pole do udoskonalenia. Opisanie kompletnej składni danego języka nie jest rzeczą łatwą, szczególnie w przypadku języka asembler, który posiada wiele różnych notacji oraz różnic w składni zależnych od architektury procesora. Aplikację można natomiast łatwo rozszerzyć o dodatkowe reguły składniowe dla dowolnego języka, zdefiniując je w pliku konfiguracyjnym i generując na jego podstawie nowy parser. Reszta programu jest niezależna od wykorzystanego języka analizowanego.

Sam proces analizy i porównywania działa w sposób zadowalający, jednak tu również istnieje duże pole do zmian i ulepszeń. W tej pracy zaimplementowano jedynie jeden z wielu dostępnych sposobów i podejść na porównywanie kodów źródło-

wych. W testach udowodniono, że aplikacja jest odporna na najbardziej popularne metody oszukiwania (zmiana nazw zmiennych, kolejności wykonywanych funkcji). Brakuje tutaj analizy pełnego przepływu porównywanych programów oraz dokładniejszego oznaczania miejsc podobnych w obu kodach.

# 7 Bibliografia

## Literatura

[1] Encyklopedia PWN - Plagiat - Wydawnictwo PWN, Warszawa 1997

[2] Plagiarism Detection across Programming Languages - Christian Arwin, M.M. Tahaghoghi - Australia 2006

[3] Chanchal Kumar Roy and James R. Cordy - A Survey on Software Clone Detection Research - Canada 2007

[4] Lex - A Lexical Analyzer Generator - M. E. Lesk and E. Schmidt - http://dinosaur.compilertools.net/

[5] Flex - The Fast Lexical Analyzer - http://flex.sourceforge.net/

[6] The Fast Lexical Analyser Generator - Gervin Klein - http://jflex.de

[7] Lex  Yacc - John R. Levine, Tony Mason, Doug Brown - O'Reilly  Associates 1992

[8] BYACC - Berkeley Yacc – http://invisible-island.net/byacc/byacc.html

[9] BYACC/J - Tomas Hurka - http://byaccj.sourceforge.net/

[10] Three models for the description of language - Chomsky, Noam (1956) - IRE Transactions on Information Theory 2 (2): 113–24

[11] Binary codes capable of correcting deletions, insertions, and reversals - V. Levenshtein - Soviet Physics Doklady 1965

[12] Practical Graph Isomorphism - B.D. McKay - Congressus Numerantium, vol. 30, pp. 45-87, 1981

[13] Compiler Basics. Extended Backus Naur Form. - Farrell, James A. - 1995

[14] Linux assemblers:  A comparison of GAS and NASM - Ram Narayan - IBM Developerworks paper - 2007

[15] A survey of graph edit distance - Xinbo Gao, Bing Xiao, Dacheng Tao, Xuelong Li - London 2009

[16] 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

[17] Computers and Intractability: A Guide to the Theory of NP-Completeness - Michael Garey, David S. Johnson - USA 1979

[18] P.J. Kelly, Ą congruence theorem for trees"Pacific J. Math., 7 (1957) pp. 961–968; Aho, Hopcroft Ullman 1974.

[19] Hopcroft, J. E., John; Wong, J. (1974), Łinear time algorithm for isomorphism of planar graphs", Proceedings of the Sixth Annual ACM Symposium on Theory of Computing,

[20] 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.

[21] Booth, Kellogg S.; Lueker, George S. (1979), Ą linear time algorithm for deciding interval graph isomorphism", Journal of the ACM 26 (2)

[22] Colbourn, C. J. (1981), Ón testing isomorphism of permutation graphs", Networks 11: 13–21

[23] Bodlaender, Hans (1990), "Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees", Journal of Algorithms 11 (4): 631–643

[24] 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

[25] Miller, Gary (1980), "Isomorphism testing for graphs of bounded genus", Proceedings of the 12th Annual ACM

[26] An algorithm for subgraph isomorphism - Ullmann, Julian R. (1976), Journal of the ACM 23 (1): 31–42

[27] An Algorithm for Subgraph Isomorphism - J.R. Ullmann - National Physical Laboratory, Tedd, ngton, M, ddlcsex, England 1975

[28] The Assignment Problem and the Hungarian Method - Bruff, Derek - Harvard University 2005

[29] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento - Subgraph Transformations for the Inexact Matching of Attributed Relational Graphs - Computing, vol. 12, pp. 43-52, 1998

[30] W.H. Tsai and K.S. Fu - Subgraph Error-Correcting Isomorphisms for Syntactic Pattern Recognition - IEEE Trans. Systems, Man, and Cybernetics, vol. 13, pp. 48-62, 1983.

# Spis rysunków

# Spis listingów