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

19 maja 2013

# Spis treści

# 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 zwykłe 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]:

- tekscie - 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.

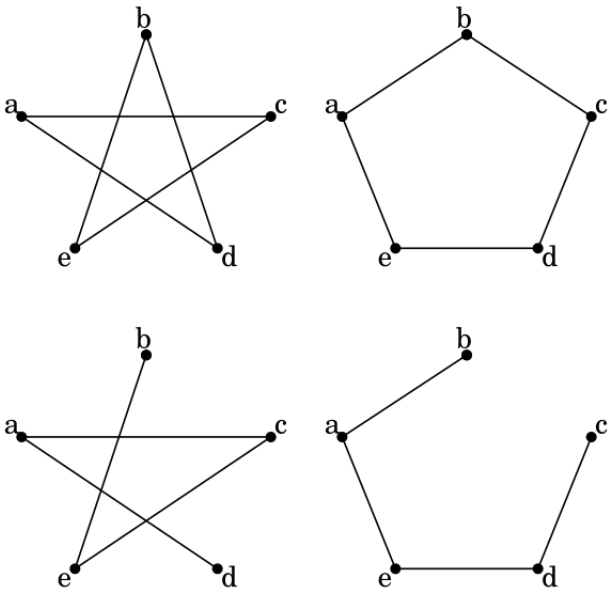
W swojej pracy zdecydowałem się na podejście do problemu pod różnymi kątami i wykorzystanie kilku dostępnych metod.

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[7]. 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 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

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.

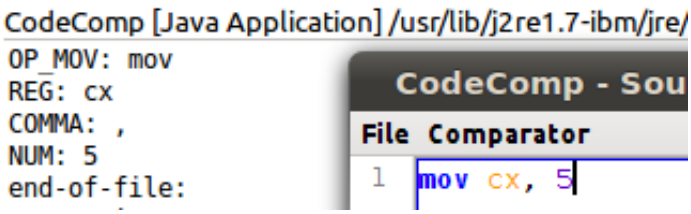
Etap analizy składa się z trzech etapów pośrednich - analizy leksykalnej, składniowej i semantycznej. Polega on na rozłożeniu kodu źródłowego na czynniki składowe oraz na budowie jego reprezentacji pośredniej.

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 |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.

## 3.2 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.

## 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 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.

## 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 opisuję 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[6], 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 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.

## 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[10].

| | 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[10]

## 4.2 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.

## 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, [0xFFFFFF]
```

**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, [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.

```
1 mov eax, [0xFFFFFF + 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 $src_1$, $src_2$
  - [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

# 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 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 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 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 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 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 [11].

## 5.2 Izomorfizm grafu

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[13].

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

- drzewo[14]
- graf planarny[15]
- graf przedziałowy[17]
- graf permutacji[18]
- niepełne drzewo rzędu k[19]
- graf o ograniczonym stopniu[20]
- graf o ograniczonym genus (rodzaj topologiczny)[21]

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.

## 5.3 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[22].

### 5.3.1 Algorytm brute-force

Algorytm brute-force to algorytm przeszukiwania w głąb i polega na znalezieniu wszystkich izomorficznych grafów $G_\alpha = (V_\alpha, E_\alpha)$ i podgrafów grafu $G_\beta = (V_\beta, E_\beta)$.

#### Opis algorytmu

Założenia[22]:

$p_\alpha \in V_\alpha \qquad q_\alpha \in E_\alpha \qquad p_\beta \in V_\beta \qquad q_\beta \in E_\beta$

$A = [a_{ij}]$ - macierz sąsiedztwa grafu $G_\alpha$

$B = [b_{ij}]$ - macierz sąsiedztwa grafu $G_\beta$ $C = [c_ij] = M'(M'B')^T, gdzieT-transpozycja$

$M' = p_\alpha X p_\beta$ - 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:

$(\forall i \forall j)(a_{ij} = 1) \Rightarrow (c_i j = 1)$

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[22]:

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;

### 5.3.2 Algorytm Ullmann'a

### 5.3.3 Algorytm VF2

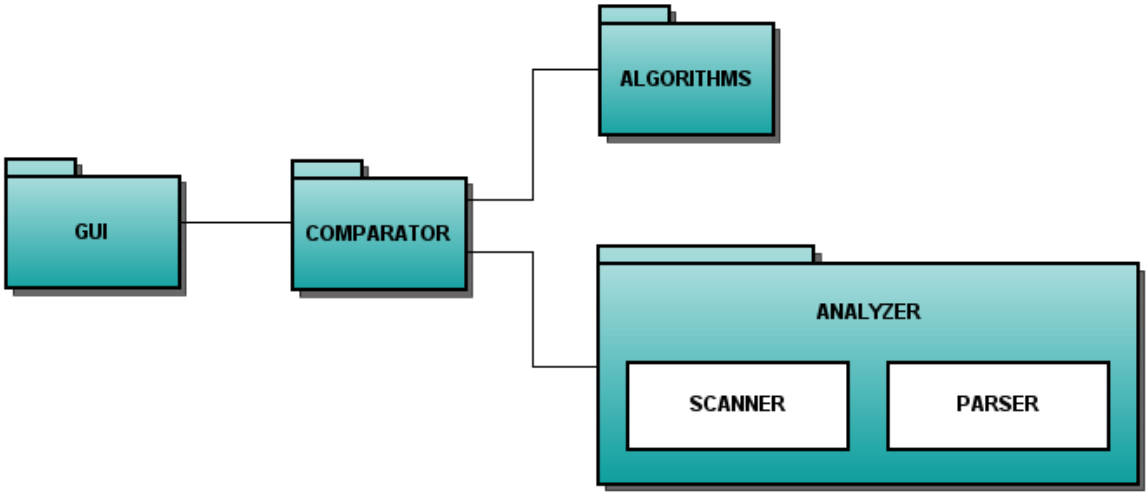
[12]

# 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 głownych pakietów: Comparator, Algorithm, Analyzer oraz GUI.




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.

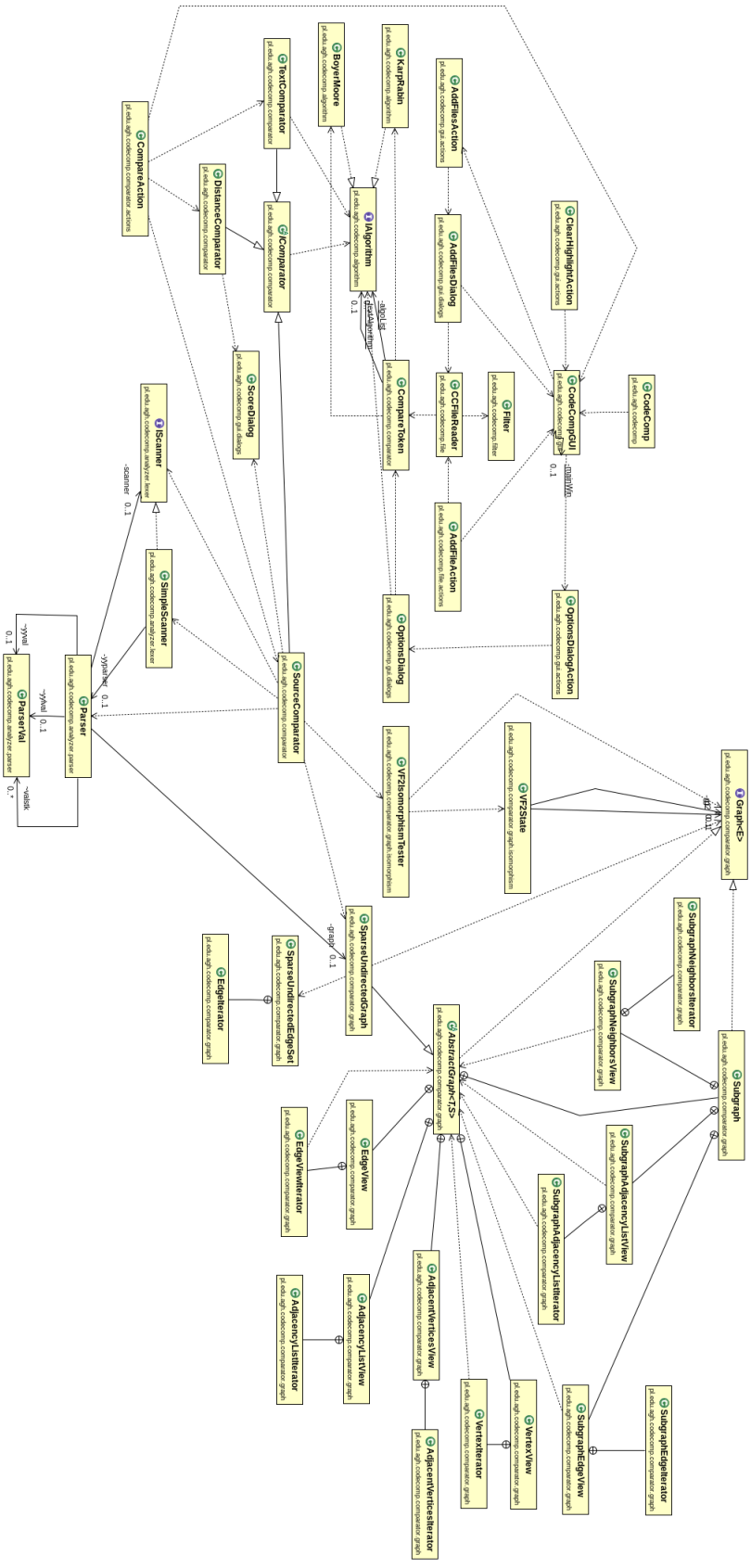
Algorithm to pakiet, w którego skład wchodzi zestaw algorytmów porównywania

tekstu oraz badania izomorficzności grafów.

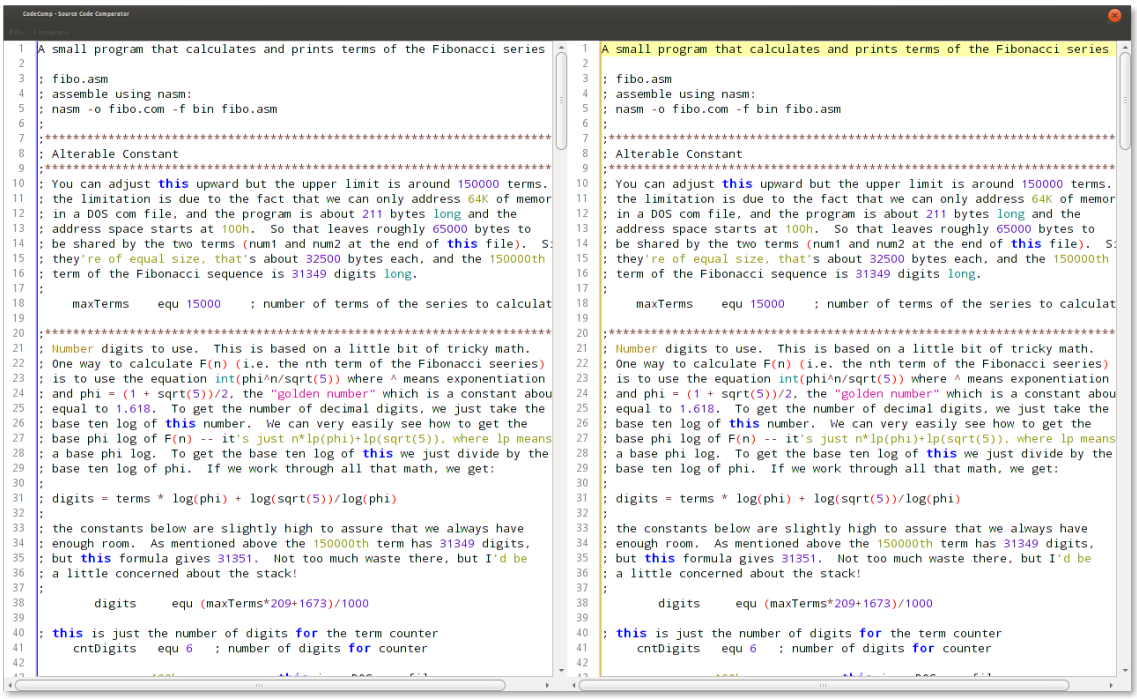
Algorytmy porównywania tekstu wykorzystane zostały do graficznego zaznaczania fragmentów identycznych w porównywanych kodach źródłowych. Na wejście przyjmują teksty kodów źródłowego w całości, następnie dzielą je na pojedyncze wyrazy i sprawdzają, czy dany wyraz z jednego kodu źródłowego znajduje się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

Algorytm sprawdzania izomoriczności grafów przyjmuje na wejście dwa grafy skierowane i wykorzystuje algorytm VF2 do obliczenia dystansu pomiędzy nimi. Wynikiem jego działania jest wartość liczbowa oraz procentowa 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.


pl.edu.agh.codecomp.comparator.actions
ComparerAction
TextComparator
BoyerMoore
KarpRabin
pl.edu.agh.codecomp.algorithm
AddFilesAction
pl.edu.agh.codecomp.gui.actions
DistanceComparator
pl.edu.agh.codecomp.comparator
IComparator
IAlgorithm
AddFilesDialog
pl.edu.agh.codecomp.gui.dialogs
ClearHighlightAction
ScoreDialog
CompareToken
CCFileReader
pl.edu.agh.codecomp.file
Filter
CodeCompGUI
CodeComp
pl.edu.agh.codecomp
IScanner
pl.edu.agh.codecomp.analyzer.lexer
AddFileAction
pl.edu.agh.codecomp.file.actions
OptionsDialog
OptionsDialogAction
SimpleScanner
Parser
ParserVal
pl.edu.agh.codecomp.analyzer.parser
SourceComparator
VF2IsomorphismTester
pl.edu.agh.codecomp.comparator.graph.isomorphism
VF2State
Graph<E>
pl.edu.agh.codecomp.comparator.graph
SparseUndirectedGraph
SparseUndirectedEdgeSet
EdgeIterator
SubgraphNeighborsIterator
SubgraphNeighborsView
AbstractGraph<TS>
Subgraph
EdgeViewIterator
EdgeView
SubgraphAdjacencyListView
SubgraphAdjacencyListIterator
AdjacencyListIterator
AdjacencyListView
AdjacentVerticesView
VertexIterator
VertexView
SubgraphEdgeView
SubgraphEdgeIterator
AdjacentVerticesIterator

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.



## 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:

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

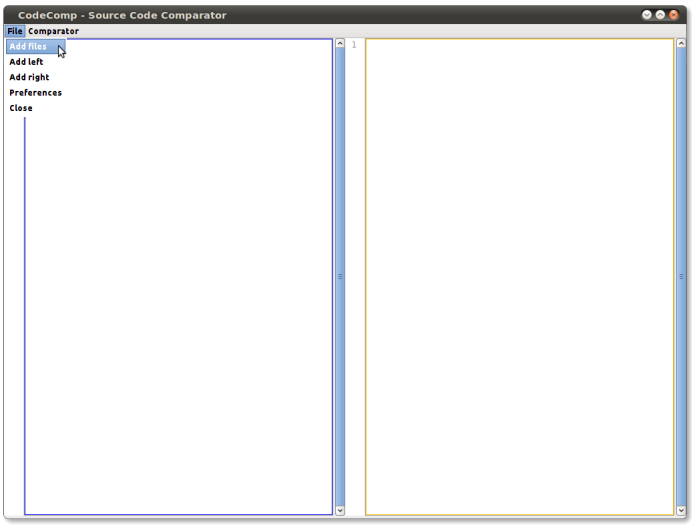
Aplikacja dołączona została w postaci kodu źrodłowego oraz skompilowanego archiwum JAR, który można uruchomić na dowolnym środowisku z zainstalowanym pakietem uruchomieniowym JRE w wersji 1.7 za pomocą polecenia:

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

## 6.3 Korzystanie z aplikacji

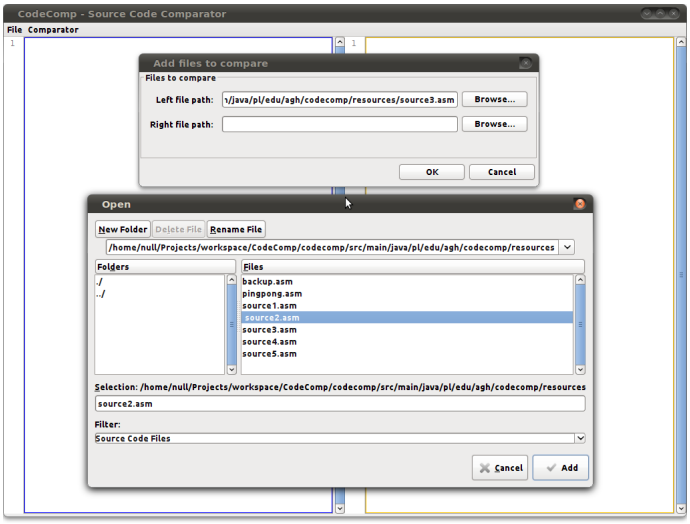
Po uruchomieniu aplikacji ukazuje nam się główne okno programu. Aby dodać kody źródłowe do porównania należy z głównego menu wybrać:

**Files → Add Files**




Rysunek 5: Okno, które ukazuje się po uruchomieniu aplikacji CodeComp.

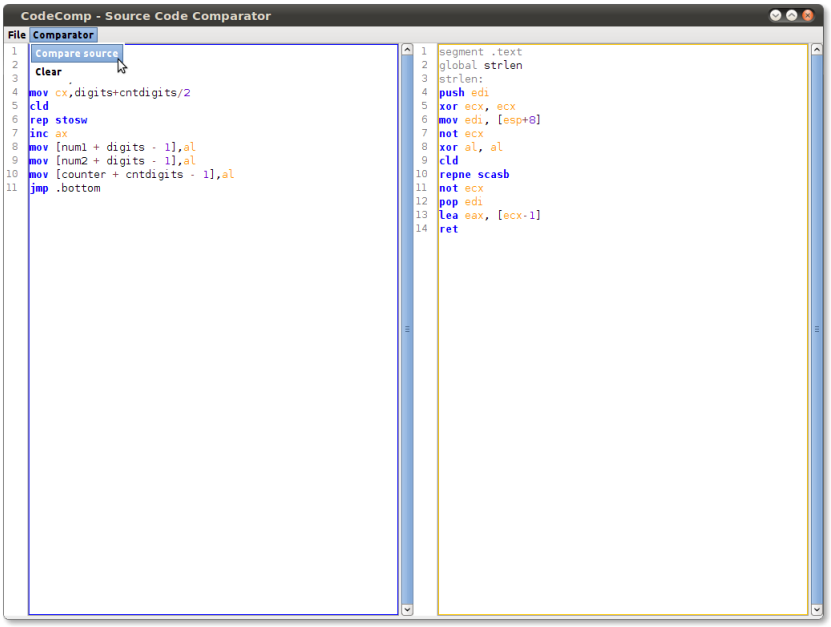
W otwartym oknie wybieramy kody źródłowe, które chcemy porównać i klikamy **Ok**.




Rysunek 6: Wybrów plików w programie CodeComp.

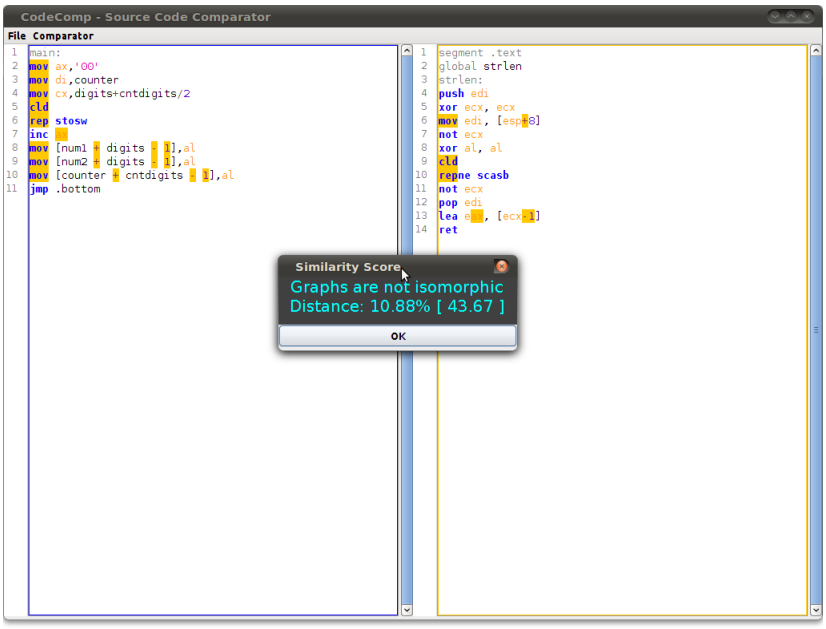
Następnie z menu głównego wybieramy:

**Compare → Compare source**




Rysunek 7: Uruchomienie analizy w programie CodeComp.

Wynik powinien pojawić się po chwili w osobnym oknie.




Rysunek 8: Wynik działania aplikacji CodeComp.

## 6.4 Opis procesu

### 6.4.1 Preprocesing

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: 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 (**'?'**).

### 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.

JFlex[4], 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.

```
32 LineTerminator          = \r|\n|\r\n
33 WhiteSpace              = {LineTerminator} | [ \t\f]
34
35 Label                   = ("."+ [:letter:]+[:digit:]*)|([:letter:]+[:digit:]* ":"+)
36 Register                = e?(al|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|rcl|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                   = stosw|stosb|stosd
53 Compare                 = scas|scasb|scasw|scasd
```

Rysunek 9: Lista makr zdefiniowanych 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. Jeśli dwa wyrażenia regularne posiadają najdłuższe dopasowanie do aktualnie przetwarzanej danej wejściowej to wykonywana jest akcja przypisana do tego, który jest zdefiniowany jako pierwszy w specyfikacji.

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

```
63 <YYINITIAL> {
64
65     /* operators */
66     "="                 { return Parser.EQ; }
67     ","                 { return Parser.COMMA; }
68     "'"                 { return Parser.APOSTROPHE; }
69
70     /* operations */
71     {Arithmetic}        { return Parser.OP_AR; }
72     {SingleArithmetic}  { return Parser.OP_SAR; }
73     {Transfer}          { return Parser.OP_MOV; }
74     {SingleTransfer}    { return Parser.OP_SMOV; }
75     {Misc}              { return Parser.OP_MISC; }
76     {Logic}             { return Parser.OP_LOG; }
77     {SingleLogic}       { return Parser.OP_SLOG; }
78     {Jump}              { return Parser.OP_JMP; }
79     {Store}             { return Parser.OP_STO; }
80     {Compare}           { return Parser.OP_COMP; }
81
82     /* literals */
83     {Number}            { return Parser.NUM; }
84
85     /* identifiers */
86     {Register}          { return Parser.REG; }
87     {Label}             { return Parser.LAB; }
88     {Identifier}        { return Parser.ID; }
89
90     \"                  { string.setLength(0); yybegin(STRING); }
91
92     /* whitespace */
93     {WhiteSpace}        { /* ignore */ }
94 }
```

Rysunek 10: 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[6] na podstawie pliku specyfikacji, który składa się z trzech sekcji:

- Deklaracje
- Reguły
- Kod użytkownika

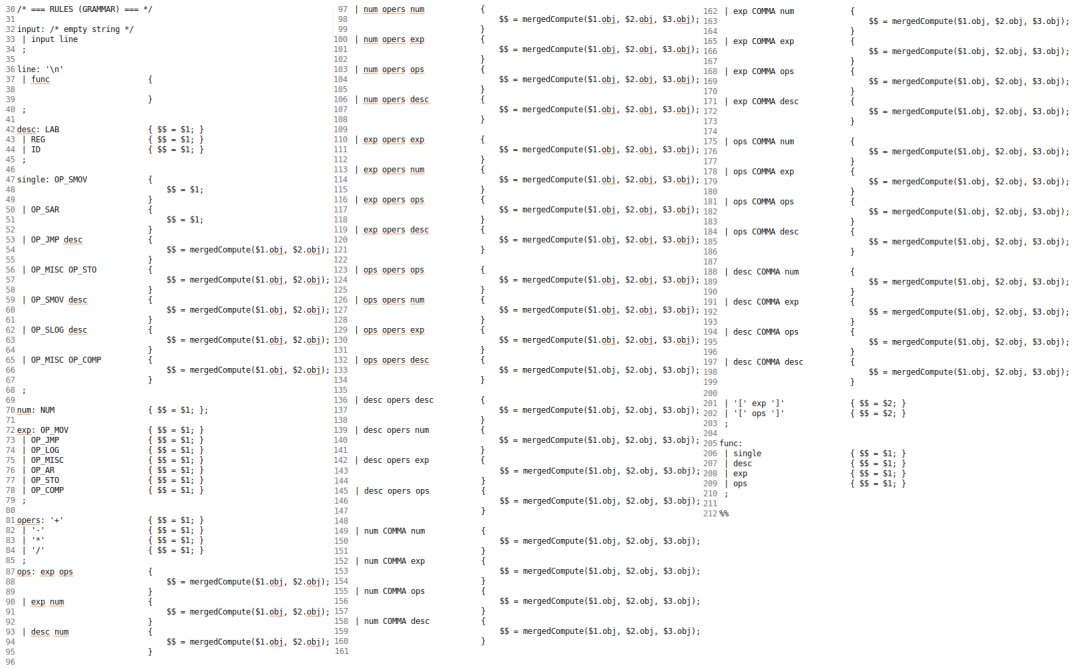
W sekcji **Deklaracje**, podobnie jak w przypadku analizatora składniowego, określamy pakiet projektu, w którym znajdzie się parser i importy wymaganych klas. Jednak w przypadku analizatora składniowego sekcja ta ma jeszcze inne, ważniejsze role. Oprócz wcześniej wspomnianych deklaracji określamy tutaj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

```
/* === DECLARATIONS === */

%{

package pl.edu.agh.codecomp.parser;

import java.io.IOException;
import java.util.HashMap;

import no.roek.nlpged.graph.Edge;

import org.fife.ui.rsyntaxtextarea.RSyntaxTextArea;

import pl.edu.agh.codecomp.graph.SimpleEdge;
import pl.edu.agh.codecomp.graph.SparseUndirectedGraph;
import pl.edu.agh.codecomp.lexer.IScanner;
import pl.edu.agh.codecomp.tree.Node;

%}

%start input
%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
%left '-' '+'
%left '*' '/'
%left NEG
%right '^'

%%
```

Rysunek 11: 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.



```
/* === RULES (GRAMMAR) === */

input: /* empty string */
  | input line
  ;

line: '\n'
  | func                    {

                            }
  ;

desc: LAB                   { $$ = $1; }
  | REG                     { $$ = $1; }
  | ID                      { $$ = $1; }
  ;

single: OP_SMOV             {
                              $$ = $1;
                            }
  | OP_SAR                  {
                              $$ = $1;
                            }
  | OP_JMP desc             {
                              $$ = mergedCompute($1.obj, $2.obj);
                            }
  | OP_MISC OP_STO          {
                              $$ = mergedCompute($1.obj, $2.obj);
                            }
  | OP_SMOV desc            {
                              $$ = mergedCompute($1.obj, $2.obj);
                            }
  | OP_SLOG desc            {
                              $$ = mergedCompute($1.obj, $2.obj);
                            }
  | OP_MISC OP_COMP         {
                              $$ = mergedCompute($1.obj, $2.obj);
                            }
  ;

num: NUM                    { $$ = $1; };

exp: OP_MOV                 { $$ = $1; }
  | OP_JMP                  { $$ = $1; }
  | OP_LOG                  { $$ = $1; }
  | OP_MISC                 { $$ = $1; }
  | OP_AR                   { $$ = $1; }
  | OP_STO                  { $$ = $1; }
  | OP_COMP                 { $$ = $1; }
  ;

opers: '+'                  { $$ = $1; }
  | '-'                     { $$ = $1; }
  | '*'                     { $$ = $1; }
  | '/'                     { $$ = $1; }
  ;

ops: exp ops                {
                              $$ = mergedCompute($1.obj, $2.obj);
                            }
  | exp num                 {
                              $$ = mergedCompute($1.obj, $2.obj);
                            }
  | desc num                {
                              $$ = mergedCompute($1.obj, $2.obj);
                            }

| num opers num             {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| num opers exp             {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| num opers ops             {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| num opers desc            {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }

| exp opers exp             {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| exp opers num             {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| exp opers ops             {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| exp opers desc            {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }

| ops opers ops             {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| ops opers num             {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| ops opers exp             {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| ops opers desc            {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }

| desc opers desc           {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| desc opers num            {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| desc opers exp            {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| desc opers ops            {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }

| num COMMA num             {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| num COMMA exp             {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| num COMMA ops             {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| num COMMA desc            {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }

| exp COMMA num             {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| exp COMMA exp             {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| exp COMMA ops             {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| exp COMMA desc            {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }

| ops COMMA num             {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| ops COMMA exp             {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| ops COMMA ops             {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| ops COMMA desc            {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }

| desc COMMA num            {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| desc COMMA exp            {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| desc COMMA ops            {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }
| desc COMMA desc           {
                              $$ = mergedCompute($1.obj, $2.obj, $3.obj);
                            }

| '[' exp ']'               { $$ = $2; }
| '[' ops ']'               { $$ = $2; }
;

func:
| single                    { $$ = $1; }
| desc                      { $$ = $1; }
| exp                       { $$ = $1; }
| ops                       { $$ = $1; }
;
%%
```

Rysunek 12: Sekcja reguł analizatora składniowego w programie CodeComp.

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. Następnie token ten jest przekazywany do funkcji parsujących, które sprawdzają, czy token jest zgodny z gramatyką oraz funkcji pomocniczych, 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.

Główna funkcja parsera:

```
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++;
```

```
 } catch (IOException e) {
   System.err.println(e.getMessage());
 }
 return tok;
}
```

Funkcja pomocnicza parsera:

```
private ParserVal mergedCompute(Node p, Node... child) {
 for(Node c : child) {
  p.addChild(c);
  graph.add(new SimpleEdge(allNodes.get(p),allNodes.get(c)));
 }
 return new ParserVal(p);
}
```

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.

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

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

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

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

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

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

[12] 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

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

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

[15] 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,

[16] 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.

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

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

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

[20] 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

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

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