Wydział Podstawowych Problemów Techniki
Politechnika Wrocławska

Porównanie kodów źródłowych poprzez budowę ich drzew składniowych

Piotr Kołodziejczyk
Nr indeksu: 244750

Praca inżynierska napisana
pod kierunkiem
dra Macieja Gębali

Politechnika
Wrocławska
Wrocław 2021

Spis treści

Wstęp

Programy realizujące tę samą lub podobną funkcjonalność często mają podobne kody źródłowe. Wynika to z ograniczonych możliwości zapisania tego samego na różne sposoby czy konieczności użycia danego algorytmu. Czasem jednak podobieństwo kodów programów nie jest przypadkowe i wynika z próby użycia kodu już istniejącego. To zaś może wiązać się z próbą plagiatu.

O podobnych kodach mówi się, kiedy realizują tę samą funkcjonalność w zbliżony sposób. Jednoznaczne określenie miary podobieństwa jest jednak trudne. Z jednej strony, programy napisane w tym samym języku implementujące proste algorytmy siłą rzeczy nie mogą różnić się w znaczący sposób. Z drugiej zaś, nawet bez zaawansowanej wiedzy programistycznej można tak zmodyfikować kod, by wizualnie czy przy pobieżnej analizie nie wykazywał podobieństwa do innego.

Istnieje szereg rozwiązań realizujących wykrywanie plagiatów, które różnią się zastosowanymi algorytmami. Najprostsze bazują na porównywaniu ciągów znaków i szukaniu podobnych w obu programach lub na ilościowym porównaniu słów kluczowych czy operatorów występujących w kodach. Najpopularniejsze podejście wykorzystuje analizę leksykalną i algorytmy porównujące ciągi tokenów [2, 8]. Innym sposobem, który wykorzystano w tej pracy, jest porównanie drzew składniowych obu programów, co dodatkowo pozwala na wykorzystanie informacji o strukturze programu.

Zakres pracy obejmuje:

- nakreślenie problemu podobieństwa programów,
- przegląd algorytmów obliczania podobieństwa grafów o strukturze drzewa,
- przedstawienie kodów źródłowych w postaci struktury drzewa,
- implementacja aplikacji obliczającej stopień podobieństwa dwu kodów źródłowych napisanych w języku Python,
- przeprowadzenie testów skuteczności aplikacji.

Praca została podzielona na cztery rozdziały. Pierwszy rozdział poświęcony jest omówieniu analizy kodów źródłowych. Przedstawiono w nim możliwe modyfikacje programów z podziałem na leksykalne i strukturalne w kontekście plagiatów. Przybliżono tam również ideę analizy leksykalnej kodów źródłowych oraz powstawania drzew składniowych

Drugi rozdział poświęcony jest przeglądowi algorytmów pozwalających porównać struktury drzew. Przedstawiono w nim algorytmy znajdowania największego wspólnego lasu oraz wyznaczania odległości edycyjnej drzew z przykładami.

W trzecim rozdziale opisano zagadnienia techniczne: wybór języka implementacji algorytmów oraz badanych kodów. Przedstawiono tam również moduł `ast` biblioteki języka Python, wykorzystanej do konstrukcji struktury drzewa programu.

Czwarty rozdział opisuje sposób użycia zaimplementowanej aplikacji oraz przedstawia przeprowadzone eksperymenty i badania skuteczności algorytmów wybranych do porównania kodów programów.

Ostatni rozdział zawiera podsumowanie przeprowadzonych prac teoretycznych i implementacyjnych. Przedstawia jakość osiągniętych celów oraz dalsze możliwości rozwoju aplikacji.

Analiza problemu

2.1 Modyfikacje programów

Istnieje wiele zabiegów pozwalających dokonać zmian w kodzie źródłowym programu, które nie wpływają na jego działanie. Podobnie jak [4, 8] można podzielić je na dwie podstawowe kategorie: leksykalne oraz strukturalne.

Do pierwszej grupy można zaliczyć takie modyfikacje jak:

- zmiana formatowania kodu,
- dodanie lub usunięcie komentarzy,
- zmiana języka komentarzy,
- zmiana nazw zmiennych, funkcji, klas etc.

Stosowanie powyższych zabiegów nie wymaga znajomości języka programowania ani zrozumienia działania kodu. Nie wpływają one bezpośrednio na działanie kodu, zmieniają go głównie wizualnie, a wpływ części z nich jest niwelowany w trakcie procesu analizy leksykalnej.

Do modyfikacji strukturalnych zaliczają się takie jak:

- zmiana kolejności deklaracji zmiennych,
- zmiana kolejności bloków kodu,
- wydzielanie nowych funkcji,
- zmiana instrukcji sterujących,
- dodanie dodatkowych zmiennych czy fragmentów kodu niewpływających na jego działanie.

Druga grupa modyfikacji wymaga już podstawowej znajomości języka programowania oraz umiejętności analizy działania programu. Mogą one bowiem wpłynąć na jego przebieg i poprawność.

W przypadku plagiatów najpopularniejszymi zmianami w kodzie są te leksykalne - wymagają najmniej czasu oraz wysiłku. Ponadto, w przypadku dużej liczby zmian strukturalnych trudno określić, czy faktycznie jest to zmieniony kod, czy autor co najwyżej wzorował się na innym. Dlatego podstawową funkcjonalnością programu stwierdzającego możliwość zajścia plagiatu powinna być co najmniej niepodatność na zmiany leksykalne.

2.2 Proces analizy leksykalnej

Kod źródłowy programu jako plik tekstowy jest trudny do analizy, dlatego dokonuje się jego przetworzenia. Pierwszym krokiem jest zamiana ciągu znaków (tworzących leksemy) na ciąg symboli leksykalnych, posiadających określone znaczenie. Tekst jest analizowany na podstawie zbioru wyrażeń regularnych, określających m.in. sposoby zapisu komentarzy, słowa kluczowe czy przestrzeń nazw zmiennych. Program, który tego dokonuje nazywany jest lekserem. Tabela 2.1 przedstawia przykładowe mapowanie leksem - token.

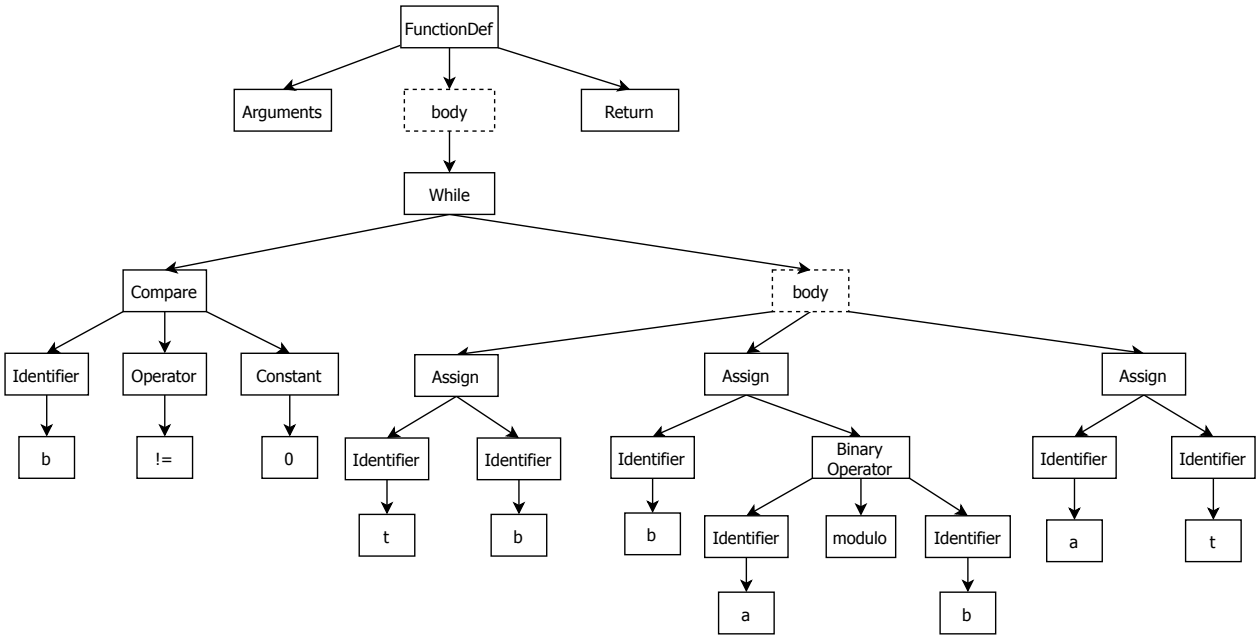
Przetworzony kod źródłowy w postaci listy tokenów może być dalej analizowany przez parser, do czego używa on gramatyki, czyli zbioru reguł określających zasady tworzenia wyrażeń i konstrukcji w języku. Wynikiem działania parsera może być drzewo wyprowadzenia lub drzewo składniowe.

leksem	token
`int`	identyfikator typu zmiennej
`x`	identyfikator zmiennej
`=`	operator przypisania
`y`	identyfikator zmiennej
`+`	identyfikator binarnego działania sumowania
`3`	literał całkowitoliczbowy

Tablica 2.1: Przykładowe porównanie leksemów i tokenów dla wyrażenia `int x = y + 3`

2.3 Drzewa składniowe

Drzewo składniowe (ang. abstract syntax tree, AST) to drzewiasta struktura przedstawiająca kod źródłowy. Poszczególne jego wierzchołki reprezentują wyrażenia, instrukcje, zmienne, etc. W przeciwieństwie do drzewa wyprowadzenia, w drzewie składniowym pomijane są takie elementy kodu jak nawiasy czy znaki interpunkcyjne. Rysunek 2.1 przedstawia przykładowe drzewo AST.



Rysunek 2.1: Przykładowe drzewo składniowe dla algorytmu Euklidesa

Przegląd algorytmów

Aby w sposób wymierny porównać dwie struktury potrzebne jest zdefiniowanie funkcji $f : (T, T) \rightarrow \mathbb{R}$ wyznaczającej odległość pomiędzy nimi. W pracy tej funkcja ta została zdefiniowana przy pomocy odległości edycyjnej drzew (tree edit distance, TED) oraz największego wspólnego lasu. Poniższy rozdział przedstawia zasadę działania oraz algorytm wyznaczania tych miar.

3.1 Drzewa

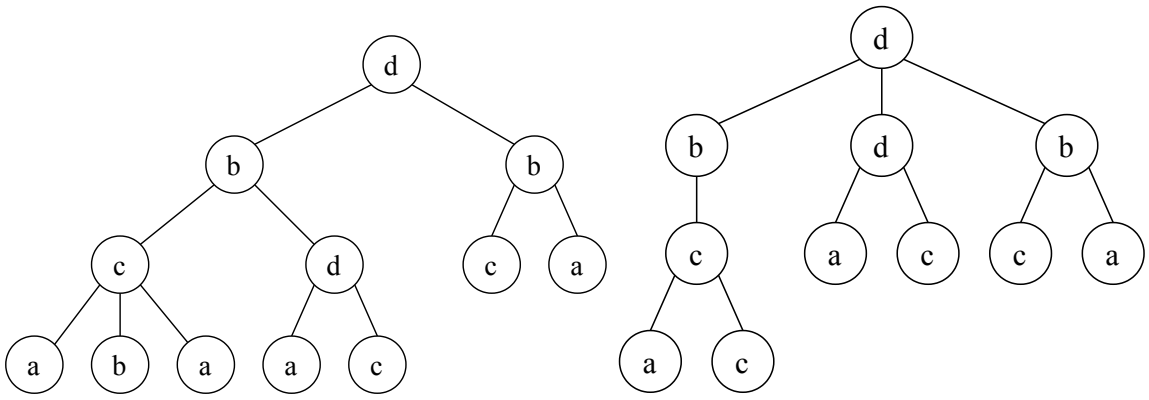
Omawiane w tym rozdziale funkcje działają na grafach o strukturze drzewa. Poniżej przedstawiono ich podstawowe cechy i własności.

Graf $T = (V, E)$, gdzie V to zbiór wierzchołków, a E - zbiór krawędzi, nazywa się drzewem, jeśli jest grafem spójnym i acyklicznym. Oznacza to, że pomiędzy dowolnymi dwoma wierzchołkami $w, v \in V$ istnieje dokładnie jedna ścieżka. Jeśli w drzewie wyróżniony jest jeden wierzchołek - korzeń, to takie drzewo nazywane jest ukorzenionym. Można wtedy dla każdego wierzchołka v określić jego głębokość $d \in \mathbb{N}$, która wyraża długość ścieżki z v do korzenia drzewa.

Dwa ukorzenione drzewa $T_1 = (V_1, E_1)$ i $T_2 = (V_2, E_2)$ są izomorficzne ($T_1 \cong T_2$), jeśli istnieje bijekcja $f : V_1 \rightarrow V_2$, taka że

$$(\forall x,y \in V_1)((x,y) \in E_1 \iff (f(x),f(y)) \in E_2).$$

Las $F = \{T_1, ..., T_n\}$ to zbiór rozłącznych drzew. Wspólny las F drzew T_1 i T_2, to taki las, że istnieją lasy $F_1 \subseteq T_1$ i $F_2 \subseteq T_2$, że $F \cong F_1 \cong F_2$. Innymi słowy, to zbiór poddrzew występujących jednocześnie w T_1 i T_2 z dokładnością do izomorfizmu. Las F drzew T_1 i T_2 to największy wspólny las, jeśli nie istnieje wspólny las G, taki że $|G| > |F|$.



Rysunek 3.1: Przykładowe drzewa T_1 i T_2

3.2 Tree edit distance

Jednym ze sposobów porównania dwu drzew T_1 i T_2 jest wyznaczenie takiego ciągu operacji na wierzchołkach, że ich zastosowanie przekształci T_1 w T_2. Niech operacje te będą zdefiniowane następująco (podobnie jak w [7]):

- (λ, v) - wstawienie wierzchołka v (jako liścia) do drzewa,
- (v, λ) - usunięcie wierzchołka v (liścia) z drzewa,
- (w, v) - zamiana wierzchołka w na v.

Dodatkowo, definiuje się funkcję kosztu $\gamma : (w, v) \rightarrow \mathbb{R}$ dla każdego typu operacji. Jeśli E to taki ciąg operacji, że przekształca T_1 w T_2, to koszt takiej transformacji wyraża się jako $\sum_{(w,v)\in E} \gamma(w,v)$. Odległością edycyjną drzewa (TED) jest wtedy $\delta(T_1, T_2) = min\{\gamma(E), \text{gdzie E jest transformacją z } T_1 \text{ na } T_2\}$ - koszt transformacji o najmniejszym możliwym koszcie.

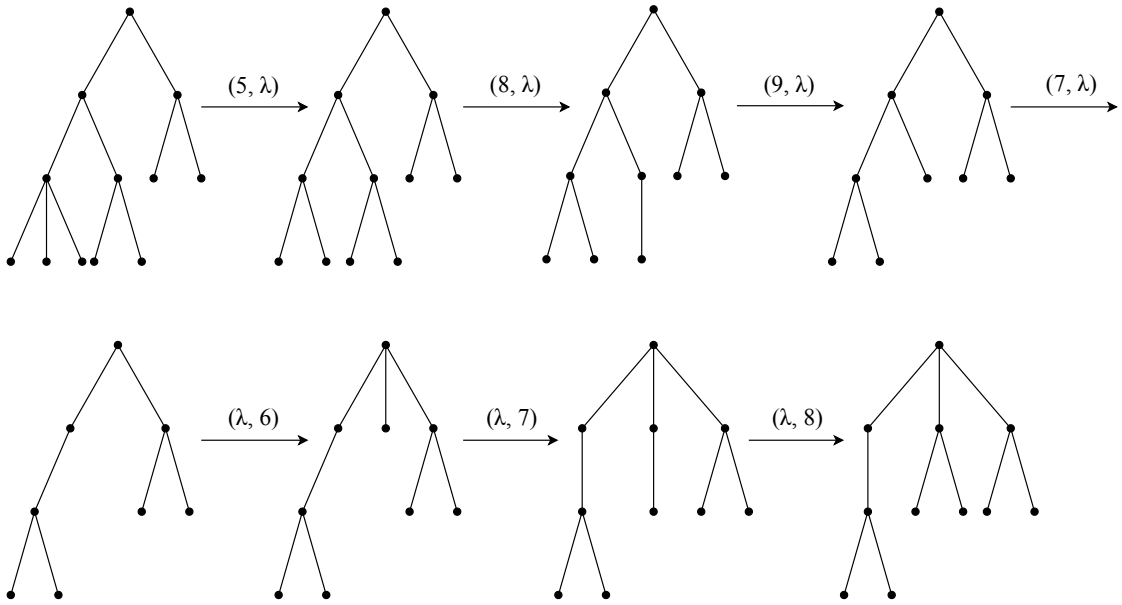
Podobnie jak w [5, 7], przyjęto, że $\gamma(w, v) = 1$, jeśli $w = \lambda$ albo $v = \lambda$ (czyli w przypadku operacji usunięcia lub wstawienia liścia), i $\gamma(w,v) = 0$ w przeciwnym przypadku - przemianowanie wierzchołka. Przyjmuje się, że etykieta wierzchołka ma mniejsze znaczenie niż struktura drzewa, dlatego koszt tej operacji jest pomijany.

Na tej podstawie, podobieństwo dwu drzew określa się wzorem

$$sim(T_1, T_2) = 1 - \frac{\delta(T_1, T_2)}{|V_1| + |V_2| - 2}.$$

Największy możliwy koszt δ dla skrajnie różnych, niepustych i ukorzenionych drzew wynosi $|V_1| + |V_2| - 2$ - kiedy należy usunąć wszystkie wierzchołki z jednego drzewa i wstawić z drugiego (korzeń występuje zawsze, dlatego suma wierzchołków została pomniejszona o 2). Wtedy podobieństwo drzew wynosiłoby 0. Z drugiej strony, dla drzew izomorficznych z dokładnością do etykiet, podobieństwo będzie wynosić 1.

Rysunek 3.2 przedstawia przykładową transformację drzewa T_1 w T_2 z użyciem podstawowych operacji. Dla uproszczenia pominięto kroki zmian etykiet wierzchołków.



Rysunek 3.2: Przykładowa transformacja T_1 w T_2 składająca się z czterech operacji usunięcia wierzchołka oraz trzech operacji wstawienia (zmiany etykiet pominięte). Numery wierzchołków przy etykietach kolejnych kroków biorą pod uwagę kolejność pre-order przejścia drzew.

3.2.1 Tree edit graph

Problem znalezienia najmniejszej odległości edycyjnej można przeformułować do problemu najkrótszej ścieżki w grafie (tree edit graph, TEG).

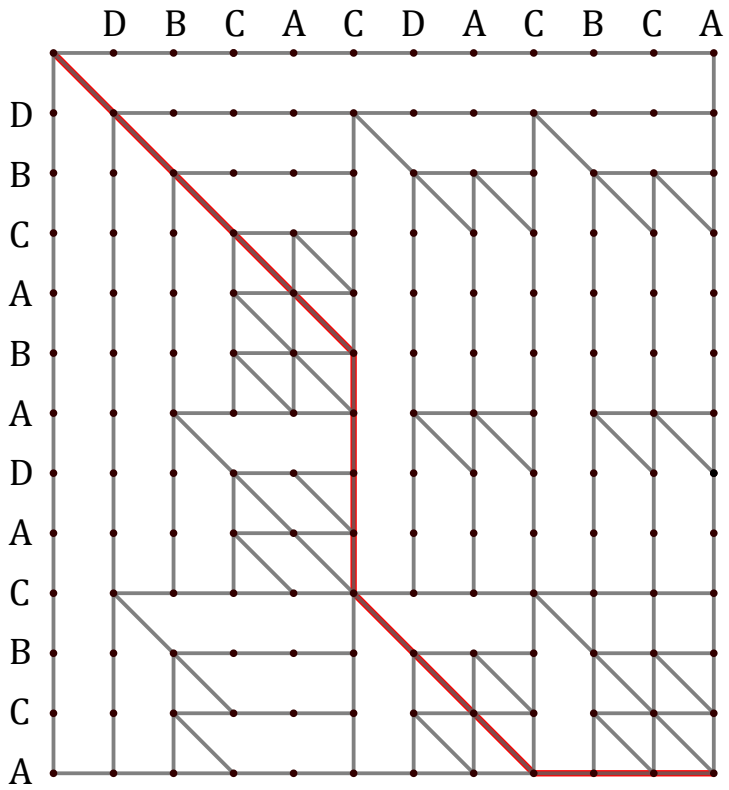
Niech $T_1 = (V_1, E_1)$ i $T_2 = (V_2, E_2)$. TEG to graf złożony z $(|V_1|+1)\cdot(|V_2|+1)$ wierzchołków tworzących prostokątną kratę - każdemu wierzchołkowi odpowiada para wierzchołków z obu drzew (poza sztucznymi v_0 i w_0). Uporządkowane i numerowane są one w kolejności pre-order przechodzenia drzew. Ponadto, krawędzie tworzone są wg zasad:

- $(v_i w_j, v_{i+1} w_j) \in E \iff v_{i+1}.d \geqslant w_{j+1}.d$ - krawędź pionowa,
- $(v_i w_j, v_{i+1} w_{j+1}) \in E \iff v_{i+1}.d = w_{j+1}.d$ - krawędź ukośna,
- $(v_i w_j, v_i w_{j+1}) \in E \iff v_{i+1}.d \leqslant w_{j+1}.d$ - krawędź pozioma,

gdzie atrybut d to głębokość wierzchołka w drzewie.

Krawędź pionowa $(v_i w_j, v_{i+1} w_j)$ odzwierciedla usunięcie wierzchołka v_i z T_1, a warunek głębokości gwarantuje, że w_{j+1} nie należy do poddrzewa T_2 z korzeniem w w_j, czyli, że w_{j+1} nie musi być wstawiony do T_2 zanim v_{i+1} będzie usunięty z T_1. Pozioma krawędź $(v_i w_j, v_i w_{j+1})$ obrazuje wstawienie wierzchołka w_{j+1} do T_2, a zadany warunek zapewnia, że v_{i+1} nie musi być usunięty przed wstawieniem w_{j+1}. Krawędź ukośna reprezentuję zamianę wierzchołków, czyli przeetykietowanie wierzchołków na tej samej głębokości.

Dowolna ścieżka z lewego górnego do prawego dolnego wierzchołka tego grafu jest poprawną transformacją jednego drzewa w drugie (formalny dowód w [7]). Znalezienie najkrótszej takiej ścieżki jest zatem równoznaczne ze znalezieniem TED. Zbudowanie grafu oraz znalezienie tej ścieżki jest możliwe w czasie $O(|V_1| \cdot |V_2|)$ z użyciem algorytmu dynamicznego.



Rysunek 3.3: Tree edit graph dla drzew z rysunku 3.1 z zaznaczoną najkrótszą ścieżką

3.2.2 Algorytm znajdowania najkrótszej ścieżki

Do przedstawienia problemu najkrótszej ścieżki w grafie edycyjnym tworzona jest macierz D rozmiaru $(|V_1| + 1) \cdot (|V_2 + 1|)$, taka że $D[i,j]$ jest wartością najkrótszej ścieżki z rogu grafu do wierzchołka (i,j). Wypełnianie macierzy następuje według poniższych zasad:

$$D[i,j] = \min(m_1, m_2, m_3), \text{ gdzie}$$
$$m_1 = D[i-1, j-1] + c_u(T_1[i], T_2[j]), \text{ jeśli}((i-1, j-1), (i,j)) \in G, \infty \text{ w p.p.},$$
$$m_2 = D[i-1, j] + c_d(T_1[i]), \text{ jeśli}((i-1, j), (i,j)) \in G, \infty \text{ w p.p.},$$
$$m_3 = D[i, j-1] + c_i(T_2[j]), \text{ jeśli}((i, j-1), (i,j)) \in G, \infty \text{ w p.p.}$$

Trwając w przyjętej wcześniej konwencji, dla dowolnych v, w, koszt zmiany etykiety $c_u(v, w)$ równy jest 0, natomiast koszty wstawienia $c_i(v)$ i usunięcia wierzchołka $c_d(v)$ wynoszą 1.

3.3 Największy wspólny las

Inną możliwością zdefiniowania odległości pomiędzy drzewami jest wykorzystanie pojęcia największego wspólnego lasu. Niech dane są drzewa T_1 i T_2 i ich największy wspólny las F. Wtedy

$$\delta(T_1, T_2) = 1 - \frac{|F|}{max(|T_1|, |T_2|)}$$

to odległość pomiędzy drzewami. Odległość ta jest znormalizowaną metryką, więc spełnia warunki:

- $0 \leqslant \delta(T_1, T_2) \leqslant 1, \delta(T_1, T_2) = 0$ wtedy i tylko wtedy, gdy $T_1 \cong T_2$,
- $\delta(T_1, T_2) = \delta(T_2, T_1)$,
- $\delta(T_1, T_3) \leqslant \delta(T_1, T_2) + \delta(T_2, T_3)$,

co udowodniono w [6].

Miarę podobieństwa wyraża się wtedy jako:

$$sim(T_1, T_2) = \frac{|F|}{max(|T_1|, |T_2|)}.$$

Jeśli drzewa są izomorficzne, to $|F| = |T_1| = |T_2|$, więc $sim(T_1, T_2) = 1$, a dla całkiem różnych drzew $(|F| = 0)$, podobieństwo to równa się 0.

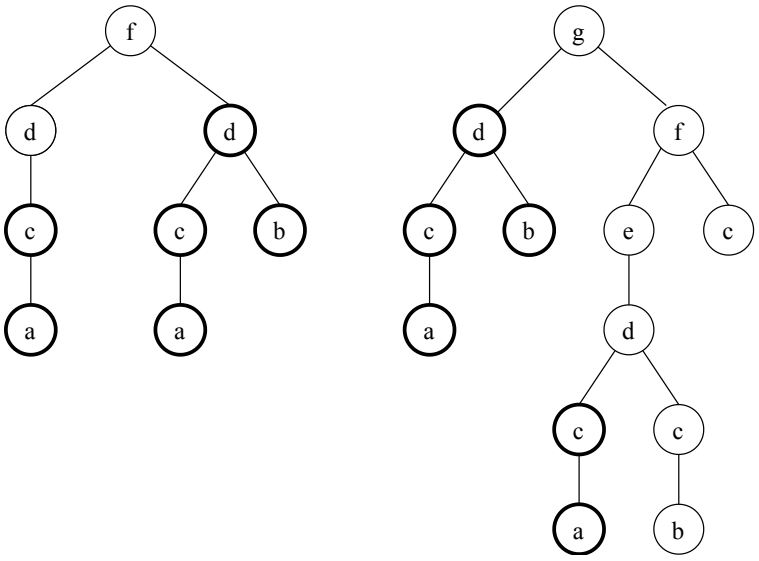
Rysunek 3.4 przedstawia przykładowy wspólny las dla dwu drzew.

3.3.1 Obliczanie największego wspólnego lasu

Niech T będzie drzewem o n wierzchołkach, v wierzchołkiem w T. Certyfikat dla v to taka liczba naturalna ϕ_v, $1 \leqslant \phi_v \leqslant n$, że dla wierzchołka s drzewa T, będącego korzeniem poddrzewa T_s, jeśli $\phi_s = \phi_v$, to $T_s \cong T_v$. Tym samym, jeśli dwa wierzchołki mają taki sam certyfikat, to poddrzewa ukorzenione w tych wierzchołkach są izomorficzne.

Algorytm znalezienia największego wspólnego drzewa wymaga obliczenia certyfikatów dla wszystkich wierzchołków. Można tego dokonać w czasie liniowym, przechodząc wszystkie wierzchołki w górę od liści do korzenia. Dla każdego wierzchołka v, jeśli para (etykieta, certyfikaty dzieci) już wystąpiła, to znaczy, że istnieje drzewo izomorficzne do drzewa o korzeniu w v i ma już nadany certyfikat, który przypisywany jest wierzchołkowi v. W przeciwnym wypadku jako certyfikat wierzchołka v ustala się najmniejszą liczbę naturalną niebędącą jeszcze certyfikatem. Całość przedstawia algorytm 3.1.

Korzystając z obliczonych certyfikatów można znaleźć las wspólnych poddrzew obu drzew, co przedstawia algorytm 3.2. Na początku tworzone są dwie listy L_1 i L_2 nieoznaczonych wierzchołków drzew odpowiednio T_1 i T_2, posortowanych nierosnąco po wartościach certyfikatów. By zachować liniową złożoność algorytmu



Rysunek 3.4: Dwa drzewa z zaznaczonym największym wspólnym lasem (pogrubione wierzchołki)

Pseudokod 3.1: Obliczanie certyfikatów dla wierzchołków drzew T_1 i T_2

```
   Input: Drzewa T1 i T2
1  count = 0
2  certificates = {}
3  foreach node ∈ T1 ∪ T2 w porządku post-order do
4  |  subtree_id = (node.name, (child.certficate foreach child in node.children))
5  |  if subtree_id ∈ certificates then
6  |  |  node.certificate = certificates[subtree_id]
7  |  else
8  |  |  certificates[subtree_id] = count
9  |  |  node.certificate = count
10 |  |  count = count + 1
```

należy użyć sortowania kubełkowego. Dalej następuje równoległe przejście po obu listach, w czasie którego oznaczane są wierzchołki należące do największego wspólnego lasu - jeżeli istnieją wierzchołki w obu drzewach z tym samym certyfikatem, to poddrzewa zakorzenione w tych wierzchołkach są izomorficzne, a więc należą do wspólnego lasu. Wszyscy potomkowie są oznaczani, a rozmiar lasu zwiększany o rozmiar poddrzewa. Posortowanie według certyfikatów gwarantuje, że badane aktualnie poddrzewo jest możliwie największe, tj. nie jest częścią większego wspólnego poddrzewa. Ponadto, istnieje możliwość dodania parametru `min_size`, określającego minimalną wielkość poddrzewa, która może być brana pod uwagę (np. by pojedyncze wierzchołki nie były wliczane).

Cała procedura działa w czasie $O(|T_1|+|T_2|)$ - w obu algorytmach każdy wierzchołek drzewa odwiedzany jest stałą liczbę razy.

3.3.2 Sortowanie kubełkowe

Na początku algorytmu znajdowania największego wspólnego lasu lista wierzchołków obu drzew musi być zostać posortowana po wartościach certyfikatów. Aby złożoność całego algorytmu pozostała liniowa, należy wykorzystać sortowanie kubełkowe, które dla liczb o rozkładzie jednostajnym (ogólniej, jeśli suma kwadratów rozmiarów kubełków jest liniowa względem łącznej liczby elementów) działa w czasie $O(n)$ [3].

Procedura sortowania polega na podziale przedziału [0,1) na n podprzedziałów jednakowych rozmiarów, a następnie wstawieniu liczb do odpowiednich "kubełków". Wewnątrz nich, liczby (z poprzednich założeń jest

Pseudokod 3.2: Największy wspólny las drzew T_1 i T_2

```
Input: Drzewa T1 i T2 z obliczonymi wartościami certyfikatów, wartość min_size
L1 - lista wierzchołków z T1
L2 - lista wierzchołków z T2
common = 0
posortuj kubełkowo L1 i L2 po wartościach certyfikatów nierosnąco
while L1 i L2 niepuste do
    if v.marked or v.certificate < w.certificate then
        v = L1.pop()
    else if w.marked or v.certificate > w.certificate then
        w = L2.pop()
    else
        if v.size > min_size then
            common += v.size
        v.marked = True
        dla każdego potomka v ustaw marked = True
        v = L1.pop()
        w.marked = True
        dla każdego potomka w ustaw marked = True
        w = L2.pop()
    return common
```

ich niewiele), są sortowane przez wstawianie. Na koniec podprzedziały są scalane, tworząc posortowany ciąg. Całość przedstawia pseudokod 3.3.

Pseudokod 3.3: Algorytm sortowania kubełkowego

```
Input: Tablica liczb A, taka że ∀i 0 ⩽ A[i] ⩽ 1
n = A.length
B[0..n-1] - nowa tablica
for i=1 to n do
    utwórz pustą listę B[i]
for i=1 to n do
    wstaw A[i] na listę B[⌊nA[i]⌋]
for i=1 to n-1 do
    posortuj listę B[i] przez wstawianie
połącz listy B[0], ..., B[n − 1] z zachowaniem kolejności
```

3.4 Zliczanie wierzchołków

W celu analizy skuteczności powyższych algorytmów, zaimplementowano również mniej złożony sposób porównania dwu drzew, polegający na zliczaniu wystąpień takich samych tokenów w obu strukturach.

Jeśli T_1 i T_2 zostaną potraktowane jako multizbiory tokenów obu drzew, to wtedy ich podobieństwo wyraża stosunek liczby elementów wspólnych do rozmiaru większego z nich:

$$sim(T_1, T_2) = \frac{|T_1 \cap T_2|}{max(|T_1|, |T_2|)}.$$

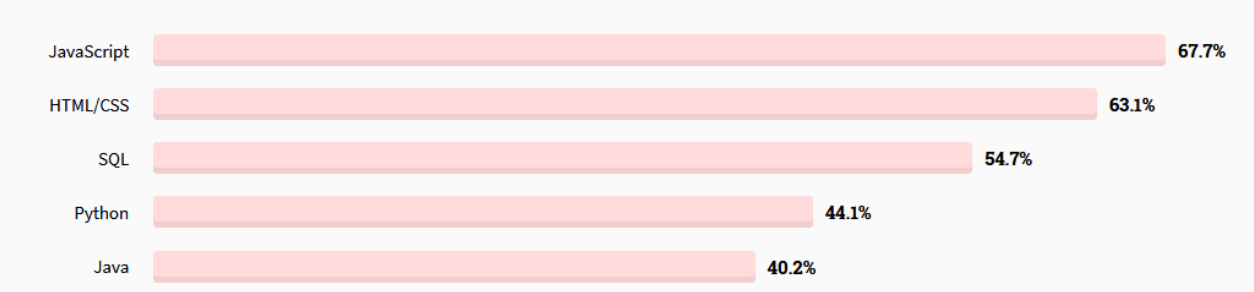
Implementacja systemu

Aplikacja realizująca opisane algorytmy została stworzona w języku Python 3. Również obiektem badania podobieństwa są kody napisane w tym języku. Podstawowym narzędziem jest moduł `ast`, umożliwiający automatyczne parsowanie kodu programu do drzewa składniowego.

4.1 Język Python

Python to wysokopoziomowy język programowania powstały w 1991. To język wieloparadygmatowy - wspiera programowanie obiektowe, imperatywne oraz funkcyjne. Jest dynamicznie typowany, automatycznie zarządza pamięcią. Posiada wiele implementacji, standardowa (CPython) napisana została w języku C.

Język Python jest obecnie jednym z najpopularniejszych języków programowania. Według badania Stack Overflow Developer Survey, na podstawie odpowiedzi ponad 57 tysięcy użytkowników, jest czwartym najczęściej używanym językiem programowania [1]. Ponadto, według tego samego badania, jest na trzecim miejscu pośród języków, których programiści wyrażają zainteresowanie w dalszym jego używaniu, oraz na pierwszym miejscu wśród języków, których programiści chcą zacząć używać. Duża popularność uzasadnia słuszność badania podobieństwa kodów programów właśnie w Pythonie.



Rysunek 4.1: Ranking pięciu najczęściej używanych języków programowania wg Stack Overflow Developer Survey

4.2 Moduł ast

Moduł ast dostępny w standardowej bibliotece języka Python umożliwia tworzenie i przetwarzanie drzew składniowych zadanych kodów źródłowych. Bazą dla tworzonego drzewa jest klasa `ast.AST`, po której dziedziczą klasy reprezentujące wszystkie rodzaje wierzchołków drzewa składniowego.

Kod źródłowy 4.1: Struktura obiektu zwróconego przez `ast.parse()` dla `{x: x**2 for x in numbers}`

```
Expression(
body=DictComp(
    key=Name(id='x', ctx=Load()),
    value=BinOp(
        left=Name(id='x', ctx=Load()),
```

```
        op=Pow(),
        right=Constant(value=2)),
    generators=[
        comprehension(
            target=Name(id='x', ctx=Store()),
            iter=Name(id='numbers', ctx=Load()),
            ifs=[],
            is_async=0)]))
```

Struktura obiektu zwracanego przez metodę `ast.parse()` jest nieregularna, dlatego w celu ułatwienia późniejszego porównywania, na jej podstawie tworzona jest klasyczna struktura drzewa 4.2. Metoda `build_tree()` (4.3) dla każdego obiektu w `ast.AST` iteruje po jego dzieciach (korzystając z wbudowanej metody `ast.iter_child_nodes()` dodając je do tworzonej struktury.

Kod źródłowy 4.2: Implementacja klas drzewa oraz pojedynczego wierzchołka

```
class Node():

    def __init__(self, name: str) -> None:
        self.parent = None
        self.children = []
        self.name = name


class Tree():

    def __init__(self, root: Node) -> None:
        self.root = root
```

Kod źródłowy 4.3: Metoda `build_tree` tworząca obiekt `tree` na podstawie struktury `ast.AST`

```
def build_tree(ast_object):

    # add all ast_node children to tree_node children list
    def walk_children(tree_node, ast_node):
        for child in ast.iter_child_nodes(ast_node):
            new_node = Node(type(child).__name__)
            tree_node.children.append(new_node)
            child.parent = tree_node
            walk_children(new_node, child)

    root_node = Node(type(ast_object).__name__)
    tree = Tree(root_node)
    walk_children(root_node, ast_object)
    return tree
```

Działanie i testy

W poniższym rozdziale przedstawiony został sposób użycia programu realizującego opisane wcześniej algortymy oraz omówione zostały testy ich skuteczności. Zalecane jest używanie najnowszej wersji języka Python (w czasie powstawania tej pracy to wersja 3.9.0), aby wszystkie konstrukcje gramatyczne użyte w porównywanych kodach zostały rozpoznane przez moduł `ast`.

5.1 Użycie

Powstały program (kod dołączony na płycie CD w katalogu `aplikacja`) uruchamia się z poziomu linii komend. Danymi wejściowymi są dwa pliki zawierające kody programów w języku Python. Kody muszą być poprawne gramatycznie, tak by moduł `ast` zbudował drzewo składniowe. Opcjonalne (wykluczające się) parametry `-lcf`, `-c`, `-ted` pozwalają wybrać, który z zaimplementowanych algorytmów ma być wykorzystany do porównania kodów. W przypadku niepodania żadnego z nich, przedstawione jest porównanie działania wszystkich trzech. Ponadto, przy wyborze `-lcf`, wymagany jest dodatkowy argument `-n`, definiujący minimalny rozmiar poddrzew, które mają być wliczane do wspólnego lasu. Interfejs pomocy zaimplementowany przy użyciu wbudowanego modułu `argparse` przedstawia kod 5.1, natomiast przykładowy wynik działania programu - 5.2.

Dodatkowo, w celu automatyzacji testów stworzono skrypt `test_script.py`, porównujący ze sobą wszystkie pliki w danym katalogu. Jego uruchomienie wygląda następująco: `python test_script.py <katalog>`.

Kod źródłowy 5.1: Instrukcja dla wywołania `python comparator.py --help`

```
usage: comparator.py [-h] [-c | -ted | -lcf] [-n N] file1 file2

Python codes similarity measurement.
When no optional parameter, similarity calculated using all three methods

positional arguments:
  file1              First python file to be analysed
  file2              Second python file to be analysed

optional arguments:
  -h, --help         show this help message and exit
  -c, --counting     Check similarity by counting the same nodes
  -ted, --tree_edit_distance
                     Check similarity by calculating the tree edit distance
  -lcf, --largest_common_forest
                     Check similarity by finding the largest common forest
  -n N               Minimal size of a subtree, required when -lcf
```

Kod źródłowy 5.2: Wynik działania programu dla dwu podobnych kodów

```
Similarity:
0.9744 - calculated with largest common forest algorithm
0.9901 - calculated by counting common nodes
0.9483 - calculated with tree edit distance algorithm
```

5.2 Testy skuteczności

Badanie skuteczności aplikacji zostały podzielone na dwie zasadnicze części. Pierwsza z nich polega na porównywaniu kodów realizujących tę samą funkcjonalność, poddanych przerobieniu. Druga część to badanie podobieństwa kodów realizujących tę samą funkcjonalność, ale napisanych niezależnie. Oczekiwania względem testów są takie, by w przypadku pierwszej grupy uzyskać współczynniki podobieństwa jak najbliższe jedności, a dla drugiej - jak najmniejsze.

Kody programów do przeprowadzenia poniższych testów zostały napisane przez ochotników - studentów informatyki Politechniki Wrocławskiej.

5.2.1 Porównanie plagiatów

W celu zbadania skuteczności algorytmów w znajdowaniu plagiatów przeprowadzono test polegający na stworzenia kodu testowego, a następnie 'splagiatowaniu' przez różne osoby. Ich zadaniem było przerobienie go tak, jak zrobiliby to w przypadku takiej konieczności. W środowisku akademickim chodziłoby o takie przerobienie kodu, by oceniający nie zauważył podobieństwa do kodu innej osoby. Co istotne, osoby przerabiające kody nie znały sposobu działania algorytmów.

Testowane kody to:

- implementacja struktury drzewa czerwono-czarnego z podstawowymi operacjami: wyszukiwania wartości, wstawiania i usuwania wierzchołków,
- kodowanie pliku `*.tga` z użyciem filtra dolno- i górnoprzepustowego i kwantyzatora.

Analizie poddane zostały zmodyfikowane kody z kodem wyjściowym oraz, dla porównania, kody powstałe niezależnie. Średnia liczba wierzchołków porównywanych drzew wynosi 1600 w przypadku drzew czerwono-czarnych i 3100 dla problemu kodowania. Tabele 5.1 oraz 5.3 przedstawiają wyniki porównania odpowiednich kodów. Ponadto w 5.2, dla porównania, pokazano wyniki analizy dla pisanych niezależnie implementacji drzew RBT.

	Zliczanie wierzchołków	Tree edit distance	Największy wspólny las			
			1	2	3	4
Podobieństwo	0,917	0,795	0,867	0,850	0,832	0,819
σ	0,070	0,118	0,088	0,099	0,098	0,097

Tablica 5.1: Test dla drzew czerwono-czarnych: porównanie kodów przerabianych. Wyniki trzeciego algorytmu podzielone według wymaganego minimalnego rozmiaru poddrzewa przy obliczaniu wspólnego lasu (w kolejnych tabelach tak samo).

	Zliczanie wierzchołków	Tree edit distance	Największy wspólny las			
			1	2	3	4
Podobieństwo	0,735	0,477	0,676	0,649	0,618	0,602
σ	0,084	0,134	0,089	0,099	0,106	0,108

Tablica 5.2: Test dla drzew czerwono-czarnych: porównanie kodów pisanych niezależnie.

	Zliczanie wierzchołków	Tree edit distance	Największy wspólny las			
			1	2	3	4
Podobieństwo	0,933	0,909	0,912	0,906	0,876	0,872
σ	0,037	0,062	0,049	0,053	0,069	0,071

Tablica 5.3: Test dla kodowania: porównanie kodów przerabianych.

5.2.2 Porównanie niezależnych kodów

Drugą kategorią testów było badanie podobieństwa kodów pisanych niezależnie. Pozwala to określić jakie są szanse na popełnienie tzw. błędu pierwszego rodzaju (*false positive*). Programy testowe dotyczyły:

- znalezienia sposobu na wydanie reszty jak najmniejszą liczbą monet,
- znalezienia n-tej liczby Catalana,
- stworzenia struktury drzewa binarnego i wypisanie wierzchołków w kolejności post-order,
- znalezienia n najmniejszych i n największych elementów listy.

W tej kategorii porównano w sumie 61 par kodów, a odpowiednie wyniki przedstawia tabela 5.4.

		Zliczanie wierzchołków	Tree edit distance	Największy wspólny las			
				1	2	3	4
Test 1	Podobieństwo	0,621	0,471	0,499	0,441	0,152	0,133
	σ	0,121	0,079	0,102	0,096	0,107	0,108
Test 2	Podobieństwo	0,538	0,463	0,403	0,326	0,125	0,122
	σ	0,135	0,116	0,101	0,082	0,074	0,074
Test 3	Podobieństwo	0,501	0,355	0,397	0,362	0,272	0,258
	σ	0,204	0,069	0,160	0,139	0,108	0,085
Test 4	Podobieństwo	0,597	0,660	0,482	0,433	0,319	0,291
	σ	0,104	0,122	0,093	0,081	0,102	0,102

Tablica 5.4: Porównanie wyników dla kodów pisanych niezależnie.

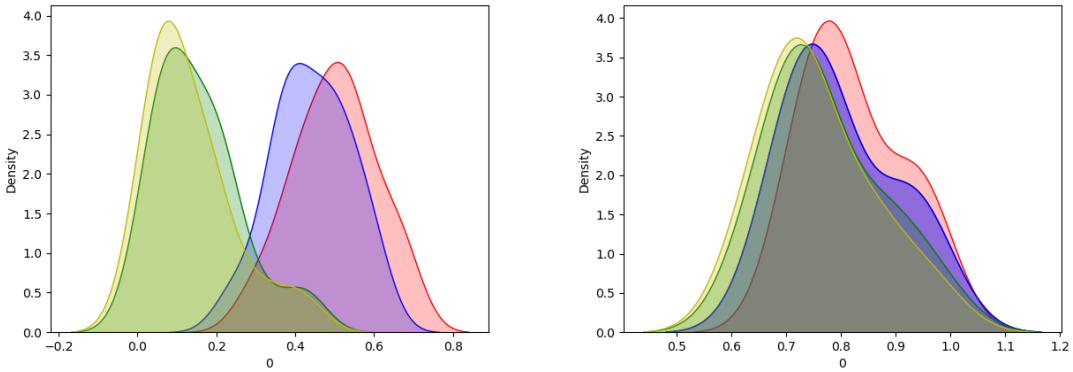
5.3 Omówienie wyników

Podobieństwo większości porównywanych kodów wynosi nie mniej jak 30-40%. Jest to zrozumiałe, jako że testowane pary kodów realizują tę samą funkcjonalność. Widać jednak zdecydowaną różnicę w wynikach testów w obu badanych przypadkach - kody splagiatowane wykazują najczęściej podobieństwo na poziomie powyżej 75%, kody niezależne - poniżej 60%.

W obu kategoriach wyniki najbliższe jedności daje najprostszy algorytm - polegający na zliczaniu jednakowych wierzchołków. Ponieważ nie wykorzystuje on żadnych informacji o strukturze programu, zbyt upodabnia kody. Da on więc najwyższe wyniki dla kodów splagiatowanych, ale też nadmiernie wysokie dla pisanych niezależnie lub niepowiązanych ze sobą.

Algorytmy porównujące drzewa składniowe wykazują większą skuteczność - chociaż dają mniejszy wskaźnik dla kodów splagiatowanych, to dla różnych kodów dają wartości bliższe zeru. W większości przypadków znajdowanie największego wspólnego lasu i odległości edycyjnej daje zbliżone rezultaty, chociaż ten pierwszy algorytm wydaje się być bardziej stabilny.

Warto zwrócić uwagę na rozbudowę algorytmu wspólnego lasu o parametr minimalnego rozmiaru wliczanego poddrzewa. Fakt coraz mniejszego podobieństwa wraz ze wzrostem wartości parametru nie powinien dziwić, ale widoczna jest różnica w spadku w zależności od rodzaju testu. Dla kodów plagiatowanych spadek ten jest zwykle rzędu kilku setnych, podczas gdy dla kodów pisanych niezależnie jest on znacznie większy. Przedstawia to wykres 5.1.



Rysunek 5.1: Porównanie rozkładów podobieństwa w zależności od parametru rozmiaru minimalnego poddrzewa: 1) kody pisane niezależnie, 2) kody plagiatowane.

Podsumowanie

Przedmiotem pracy było stworzenie aplikacji porównującej kody źródłowe programów. Zrealizowane w jej ramach założenia obejmują między innymi:

- przedstawienie programów w postaci struktury drzewa,
- implementacja algorytmów porównujących drzewa składniowe: wyznaczanie największego wspólnego lasu oraz odległości edycyjnej drzew,
- przeprowadzenie badań skuteczności programu na podstawie zbioru kodów modyfikowanych i pisanych niezależnie.

Porównując zrealizowane zadania z celem pracy, można stwierdzić, że cel ten został osiągnięty.

Powstała aplikacja (której kod dołączony jest do pracy) pozwala porównać dwa kody źródłowe napisane w języku Python. W zależności od parametrów uruchomienia użytkownik pozna wynik działania danego algorytmu lub porównanie działania każdego z nich. Wynikami są liczby rzeczywiste z przedziału [0,1]. Im wartość bliższa jedności, tym wg danego algorytmu kody są do siebie bardziej podobne.

Integralną częścią pracy było przeprowadzenie badań skuteczności aplikacji. Do testów użyte zostały kody napisane przez studentów-ochotników, których celem było albo przerobienie danego kodu na kształt plagiatu, albo napisanie niezależnie programu realizującego tę samą funkcjonalność. Wyniki pokazały, że algorytmy można uznać za skuteczne. Dla kodów powstałych w wyniku testowego plagiatu podobieństwo wynosi najczęściej co najmniej 75% (często powyżej 85%), natomiast dla tych pisanych niezależnie - w większości przypadków poniżej 60%. Dla kodów realizujących różne cele wartości te rzadko przekraczają 30%. Otrzymane wskaźniki tworzą przybliżone przedziały, na podstawie których można wnioskować o stopniu pokrewieństwa kodów. W kontekście badania plagiatów wynik bliski jedności może stanowić sugestię ręcznego porównania dwu kodów i analizy ich podobieństwa.

6.1 Możliwości rozwoju

Pomimo że wszystkie cele pracy zostały zrealizowane, możliwości rozwoju aplikacji nie zostały wyczerpane. Należą do nich między innymi:

- rozszerzenie o porównywanie rozbudowanych programów składających się z wielu plików,
- dodanie możliwości porównywania zbioru programów i wskazywanie par najbardziej podobnych,
- przerobienie na aplikację www.

Realizacja powyższych dodatkowych funkcjonalności mogłaby zapewnić większą atrakcyjność dla użytkownika.

Bibliografia

[1] Stack overflow developer survey 2020.

[2] A. Bejarano, L. García, E. Zurek. Detection of source code similitude in academic environments. *Computer Applications in Engineering Education*, 01 2015.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Wprowadzenie do algorytmów*. The MIT Press, wydanie 3rd, 2009.

[4] M. Joy, M. Luck. Plagiarism in programming assignments. *IEEE Transactions on Education*, 42(2):129–133, 1999.

[5] T. Sager, A. Bernstein, M. Pinzger, C. Kiefer. Detecting similar java classes using tree algorithms. strona 65–71, 2006.

[6] G. Valiente. Simple and efficient tree comparison. (R01-1), Jan 2001.

[7] G. Valiente. *Algorithms on Trees and Graphs*. 01 2002.

[8] Z. Đurić, D. Gasevic. A source code similarity system for plagiarism detection. *The Computer Journal*, 56:70–86, 01 2013.

Zawartość płyty CD

Na dołączonej płycie CD znajdują się kody źródłowe aplikacji oraz programów testowych, uporządkowane według katalogów:

- praca - wersja elektroniczna powyższej pracy,
- aplikacja - kody źródłowe stworzonej aplikacji,
- testy - kody źródłowe programów testowych oraz wygenerowane wyniki.