

**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА**

Факультет комп'ютерних наук та кібернетики
Кафедра математичної інформатики

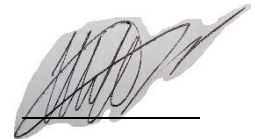
Курсова робота

за спеціальністю 122 Комп'ютерні науки

на тему:

**ПЕРЕВІРКА НА ПЛАГІАТ ПРОГРАМНОГО КОДУ
ЛАБОРАТОРНИХ РОБІТ СТУДЕНТІВ**

Виконав студент 3-го курсу
Шевченко Максим Олексійович

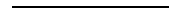


(підпис)

Науковий керівник:

Доцент

Панченко Тарас Володимирович



(підпис)

Засвідчую, що в цій курсовій роботі немає
запозичень з праць інших авторів без
відповідних посилань.

Студент



(підпис)

Реферат

Обсяг роботи 33 сторінки, 2 ілюстрації, 12 джерел посилань.

ПЕРЕВІРКА НА ПЛАГІАТ, ТОКЕНІЗАЦІЯ, НОРМАЛІЗАЦІЯ ПРОГРАМНОГО КОДУ, НАЙДОВШИЙ СПІЛЬНИЙ ПІДРЯДОК, МЕТОД ШИНГЛІВ.

Об'єктом роботи є організація перевірки лабораторних робіт в навчальному закладі і підходи до розробки програмного забезпечення для цих цілей. Предметом роботи є дослідження способів маскування плагіату та алгоритмів його виявлення.

Метою роботи є створення та програмна реалізація алгоритмів перевірки програмного коду на плагіат із його попередньою нормалізацією для подальшої інтеграції продукту із системою обліку лабораторних робіт.

Методи дослідження: дослідження існуючих на даний момент алгоритмів виявлення плагіату в програмному коді, їх модифікація. Інструменти розробки: безкоштовне та вільно поширюване інтегроване середовище Visual Studio 2019 Community Edition, мова програмування C#.

Результат роботи: досліджено основні способи приховування плагіату в програмному коді; реалізовано алгоритм перетворення програмного коду до певного універсального стандартного вигляду; досліджено та реалізовано методи виявлення плагіату як у програмному коді, так і в довільних текстових файлах; протестована ефективність та достовірність роботи отриманого продукту.

Зміст

ВСТУП	4
РОЗДІЛ 1. ДОСЛІДЖЕННЯ СПОСОБІВ ПРИХОВУВАННЯ ПЛАГІАТУ .	6
РОЗДІЛ 2. ДОСЛІДЖЕННЯ АЛГОРИТМІВ ПЕРЕВІРКИ НА ПЛАГІАТ .	11
2.1 Основні типи алгоритмів перевірки плагіату	11
2.2 Аналіз поставленого завдання та вибір методів	12
РОЗДІЛ 3. РЕАЛІЗАЦІЯ ПРОГРАМИ	14
3.1 Нормалізація та токенизація коду	14
3.2 Метод пошуку найдовшого спільного підрядка.....	16
3.3 Метод шинглів.....	18
3.4 Проблеми під час реалізації та отримані результати	19
ВИСНОВКИ	22
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ	23
ДОДАТКИ	25

ВСТУП

Актуальність роботи та підстави для її виконання. Проблема плагіату є однією з найкритичніших не тільки в навчальних закладах, а й у всьому науковому товаристві. Використання студентами чужої інтелектуальної праці у своїй роботі загалом не має на меті присвоєння собі чужих здобутків, а робиться для того, щоб полегшити собі життя й отримати хоч якісь бали за предмет. Проте таке «полегшення життя» призводить до зниження рівня знань та ставить інших студентів у не вигідне положення, адже їм доводиться самим вивчати матеріал і витрачати час на виконання завдання.

У викладачів, які приймають роботи, немає ніяких засобів перевірки плагіату, крім власної пам'яті. Зрозуміло, що з великим потоком різних лабораторних робіт студентів неможливо запам'ятати, хто яку роботу здавав. Та й система обліку лабораторних являє собою лише оцінку в журналі та, інколи, звіт. Навіть при наявності бази даних усіх лабораторних робіт викладач самотужки не зможе ефективно порівняти кожную роботу з усіма попередніми. Крім того, щороку з'являються нові потоки студентів, які отримують ті самі завдання. Тобто з кожним новим навчальним роком кількість зданих робіт буде лінійно збільшуватися, що ще ускладнить життя викладачів.

У сучасну добу можливе створення автоматичної системи перевірки лабораторних робіт на плагіат без значних ресурсних затрат. У такій системі зберігалися б усі виконані роботи всіх студентів усіх курсів за всі роки та кожна нова праця перевірялася з попередніми. Це не тільки полегшило б роботу викладачів, а й підвищило б загальний рівень знань у навчальних закладах.

Загальнодоступні засоби перевірки на плагіат саме програмного коду не поширені в такій мірі, як засоби перевірки звичайних текстових робіт. Тому я вважаю, що дослідження та реалізація алгоритмів перевірки програмного коду на плагіат є актуальною. Створений проект має на меті зробити перший крок до створення загальної бази обліку лабораторних робіт реалізацією алгоритмів порівняння програмних кодів.

Для досягнення цієї мети було поставлено такі **завдання**:

- Дослідити способи маскуванню плагіату в програмному коді.
- Дослідити та реалізувати алгоритми знаходження плагіату.
- Розробити програмне забезпечення, яке можна було б легко інтегрувати в систему обліку лабораторних робіт.
- Перевірити коректну роботу створеного продукту.

РОЗДІЛ 1. ДОСЛІДЖЕННЯ СПОСОБІВ ПРИХОВУВАННЯ ПЛАГІАТУ

Найбільш очевидним способом плагіату є використання чужої лабораторної роботи без будь-яких змін. Зрозуміло, що такий спосіб піддається викриттю навіть без додаткових засобів, бо лабораторні загалом здають в один період часу. Викладач може запам'ятати деякі з них і помітити, що вони повторюються. Проте студенти можуть використовувати лабораторні своїх знайомих, які навчалися декілька років тому, що ускладнює викриття недобросовісних осіб.

Надалі для унаочнення розглянемо просту програму, на прикладі якої покажемо зміни програмного коду. Нехай такою програмою буде програма для додавання двох чисел. Звісно, студенти створюють набагато складніші додатки, проте для прикладу й така згодиться. Тож у нашому прикладі отримали наступне порівняння програм (деяка частина стандартного коду не представлена в порівнянні):

Оригінальний код	Плагіат
<pre>static void Main(string[] args) { int a = 1; int b = 2; int c = a + b; Console.WriteLine(c); }</pre>	<pre>static void Main(string[] args) { int a = 1; int b = 2; int c = a + b; Console.WriteLine(c); }</pre>

Найпростішим способом маскуванню плагіату є перейменування змінних. Це не потребує ніяких знань та робить код справді не схожим на оригінал. Якщо правильно все зробити, то подібність коду помітити буде досить складно, адже оператори займають не значну частину коду. До того ж, сучасні інтегровані середовища розробки мають функцію рефакторингу, завдяки якій перейменування змінних стає неймовірно простим. Після нескладних змін отримуємо наступний код:

Оригінальний код	Плагіат
<pre>static void Main(string[] args) { int a = 1; int b = 2; int c = a + b; Console.WriteLine(c); }</pre>	<pre>static void Main(string[] args) { int firstValue = 1; int secondValue = 2; int result = firstValue + secondValue; Console.WriteLine(result); }</pre>

Ще одним простим способом маскування є додавання, видалення або зміна коментарів коду чи тексту, який виводить програма під час роботи. Крім зміни вигляду самої програми, це призведе ще й до зміни вигляду її результату роботи, що значно ускладнить пошук однакових робіт. Нижче наведено можливий вигляд коду при використанні цього методу:

Оригінальний код	Плагіат
<pre>static void Main(string[] args) { int a = 1; int b = 2; int c = a + b; Console.WriteLine(c); }</pre>	<pre>static void Main(string[] args) {//програма для додавання двох чисел Console.WriteLine("This program adds two number"); int a = 1; int b = 2; int c = a + b; Console.WriteLine(\$" {a} + {b} = {c}"); }</pre>

Менш очевидною, проте досить ефективною і неочікуваною є заміна типів змінних і (або) операторів доступу до функцій. Результат такої заміни схожий на перейменування змінних і має той самий ефект на очне порівняння кодів:

Оригінальний код	Плагіат
<pre>static void Main(string[] args) { int a = 1; int b = 2; int c = a + b; Console.WriteLine(c); }</pre>	<pre>static void Main(string[] args) { double a = 1.0; double b = 2.0; double c = a + b; Console.WriteLine(c); }</pre>

Більш популярним та ефективним є спосіб переносу частини коду в окрему функцію або клас. Сам алгоритм залишається без змін, проте його частини будуть розкидані по всій програмі й відслідкувати кроки виконання буде складно. У зворотному випадку це теж працює: можна код із декількох функцій зібрати в одну. Узагалі, розбиття програми на декілька окремих файлів (по класам, наприклад) сильно ускладнить роботу викладачам, адже їм доведеться перемикати безліч вікон без можливості відразу побачити весь код реалізації. Нижче наведено приклад виокремлення коду в додаткову функцію:

Оригінальний код	Плагіат
<pre>static void Main(string[] args) { int a = 1; int b = 2; int c = a + b; Console.WriteLine(c); }</pre>	<pre>static int Sum(int a, int b) { return a + b; } static void Main(string[] args) { int a = 1; int b = 2; int c = Sum(a, b); Console.WriteLine(c); }</pre>

Якщо студент хоч трохи вміє програмувати та розуміє основні принципи роботи програми, він може дещо (або повністю) змінити структуру коду, не зашкодивши самому алгоритму. При вмілому використанні цього методу спільним між двома програмами буде лише результат роботи, а вся структура та процес виконання будуть відрізнятися. При цьому постає питання, чи можна таку роботу вважати плагіатом? З однієї сторони, студент просто переробив раніше зроблений алгоритм, але з іншої – створив власну його реалізацію. Це питання стає ще критичнішим з огляду на те, що завдання лабораторних робіт часто в усіх однакове. При цьому не буде дивним, якщо різні за структурою програми реалізують один алгоритм. Залишу це питання без відповіді. Приклад незначної зміни структури коду:

Оригінальний код	Плагіат
<pre>static void Main(string[] args) { int a = 1; int b = 2; int c = a + b; Console.WriteLine(c); }</pre>	<pre>static void Main(string[] args){ int b = 2; int a = 1; Console.WriteLine(b + a); }</pre>

Ще одним простим і таким, який не вимагає особливих знань, способом маскуванню плагіату є додавання непотрібних операцій або частин коду. Цей метод може гарно замаскувати запозичення коду, проте цим не треба зловживати, адже при збільшенні кількості непотрібних рядків зростає ймовірність їх виявлення. Якщо викладач буде уважним, плагіат коду буде викрито шляхом ігнорування непотрібного коду. Приклад вигляду коду:

Оригінальний код	Плагіат
<pre>static void Main(string[] args) { int a = 1; int b = 2; int c = a + b; Console.WriteLine(c); }</pre>	<pre>static void Main(string[] args) { int a = 1; int b = 2; int d = 45; int u = -32; u = a + d - b; int c = a + b; Console.WriteLine(c); }</pre>

Напевно, найменш очевидним способом плагіату є написання лабораторної іншою мовою програмування. У своїй більшості структура коду мов програмування дуже схожа, якщо навіть не однакова. Звісно, мови відрізняються типами змінних, доступними бібліотеками або реалізацією деяких функцій. Проте для загальних робіт, де не вимагається використання можливостей конкретної мови, студент може скопіювати код, написаний, наприклад, мовою C#, у проект мови C++ й виправити помилки, які виникнуть. Якщо ж студент не лінивий і має бажання, то він може власноруч переписати код на менш подібну мову (наприклад, із C# на Python). У такому випадку визначити запозичення буде складно. Для виразності наведено приклад плагіату коду мовою C++:

Оригінальний код	Плагіат
<pre>static void Main(string[] args) { int a = 1; int b = 2; int c = a + b; Console.WriteLine(c); }</pre>	<pre>#include <iostream> using namespace std; int main(){ int a = 1; int b = 2; int c = a + b; cout << c << endl; return 0; }</pre>

РОЗДІЛ 2. ДОСЛІДЖЕННЯ АЛГОРИТМІВ ПЕРЕВІРКИ НА ПЛАГІАТ

2.1 Основні типи алгоритмів перевірки плагіату

Алгоритмів перевірки програмного коду на плагіат досить багато. Їх можна класифікувати залежно від внутрішнього формату представлення програмного коду, який вони використовують для пошуку співпадінь [1].

Найбільш поширеними загалом, а не тільки в перевірці програмного коду, є алгоритми порівняння рядків тексту. Вони базуються на пошуку спільних частин тексту. Це добре підходить для пошуку прямих запозичень як у звичайних текстах, так і в програмному коді. Якщо ж якимось модифікувати код (навіть звичайним перейменуванням змінних), то цей тип алгоритмів виявить себе зовсім не ефективним у перевірці робіт студентів.

Цього недоліку позбавлені алгоритми з використанням токенизації. У цих методах код перед перевіркою перетворюється в набір tokenів – ідентифікаторів мови, таких як типи змінних, оператори доступу, числа, функції та ін. Після такого перетворення виконується пошук плагіату одним із методів порівняння рядків. Основним недоліком цього методу є необхідність створення лексичного аналізатора для виконання токенизації та нехтування структурою програми при перевірці.

Досить цікавими є алгоритми оцінки різних метрик програми, як от кількість циклів, коментарів, викликів функцій, їх аргументи та ін. Так як алгоритм не враховує структуру програми та те, що робить код, точність цього методу досить низька. До того ж зміна метрик програми є одним із найпростіших способів приховування плагіату.

Для порівняння структури коду найкраще використовувати методи з побудовою синтаксичних дерев або графу залежності програми. Власне, синтаксичне дерево й являє собою структуру коду. Побудувати його досить просто, дотримуючись загальних правил вигляду структури коду конкретної мови програмування. Проте цей метод буде не достовірним, навіть якщо студент

просто переставить рядки місцями. Граф залежності програми не вказує на структуру коду напряду, проте він дозволяє відстежити процес виконання програми. Цей метод є найкращим для знаходження плагіату, про він складний у реалізації та й сама складність алгоритму висока, особливо для великих проєктів.

2.2 Аналіз поставленого завдання та вибір методів

Жоден метод перевірки програмного коду на плагіат не є ідеальним. Кінцеве рішення щодо унікальності роботи має приймати викладач власноруч. Ми можемо лише обрати оптимальний алгоритм, який буде себе найкраще показувати саме для конкретної поставленої задачі.

З огляду на можливі способи приховування плагіату, найкраще було б реалізувати метод побудови графу залежності програми. Але є одна дуже значна проблема конкретно для порівняння лабораторних робіт – спільна умова. Так як цей метод досліджує виконання програми, то постійно будуть виникати хибно-позитивні результати щодо реалізацій одного алгоритму різними способами. Цю проблему можна вирішити лише надаючи кожному студенту унікальний варіант завдання, виконання якого призведе до створення роботи, повністю відмінної від інших робіт. Проте цього немає зараз та не буде найближчим часом. Переважна відмінність у варіантах завдань полягає в різних вхідних даних або необхідності виконувати обчислення різних функцій. Логіка роботи програми при цьому не змінюється. Крім того, студенти можуть зберігати свої роботи й передавати нащадкам, що призведе до необхідності постійного створення нових варіантів завдань.

На мою думку, використання токенізованих алгоритмів є оптимальним для поставлених цілей. По-перше, виконання попередньої нормалізації та токенізації коду дозволить виявити перейменування в коді. По-друге, пошук спільних підрядків дозволить виявити переміщення частин коду. По-третє, реалізація самих алгоритмів порівняння рядків досить проста, основна ж складність полягає

в реалізації алгоритму токенизації. По-четверте, ці алгоритми дозволять перевіряти не лише програмний код, а й звичайні текстові реферати чи доповіді студентів. При значній зміні структури коду такий алгоритм буде не ефективним. Проте виконання складних маніпуляцій над чужою роботою для маскування плагіату або потребує попереднього освоєння принципів роботи програми, або призводить до їхнього мимовільного вивчення.

Використання інших методів вважаю не практичним, так як виконання студентом найпростіших маніпуляцій з кодом призведе до значних втрат відсотку знайденого плагіату.

Отже, для виконання роботи я обрав токенизований підхід, а саме попереднє перетворення програмного коду в рядок стандартного вигляду (виконання токенизації та нормалізації коду) з наступним застосуванням рядкових методів перевірки плагіату, конкретно методів шинглів та найдовшого спільного підрядка, як яскравих представників цього класу.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ПРОГРАМИ

3.1 Нормалізація та токенизація коду

Завдання нормалізації полягає в перетворенні програмного коду в деякий універсальний стандартний текстовий вигляд [5], який можна дослідити на плагіат стандартними рядковими методами. Так як одним із способів приховування плагіату є використання подібних ідентифікаторів мови програмування (наприклад, `int` і `long`, `private` й `protected`), то було б зручно представляти ці ключові слова в нормалізованому тексті однаковим способом. Для цього потрібно попередньо виконати лексичний аналіз мови, знайти всі ключові слова й визначити їх тип. Тобто провести токенизацію коду [6].

Для початку необхідно визначити всі ключові слова мови та згрупувати подібні. Після цього виконати заміну в коді цих слів однаковими ідентифікаторами. Так як після нормалізації виконується порівняння рядків (тип `string`), то для збереження пам'яті, яку використовує програма під час роботи, найраціональнішим є заміна всіх ключових слів одним символом (різним для кожного токenu).

Між усіма ключовими словами в мові стоять певні роздільники. Завдяки цьому їх можна легко знайти, попередньо замінивши роздільники на універсальний знак пунктуації. Якщо виконати ці дії, то ми зможемо виконати пошук ключових слів, орієнтуючись по одному конкретному роздільнику.

Алгоритм нормалізації повинен бути стійким щодо перейменування змінних та функцій. Тому необхідно кожній змінній надавати власний ідентифікатор. Якщо ці ідентифікатори будуть унікальними для кожної змінної, то звичайна перестановка рядків призведе до незадовільної роботи алгоритму порівняння рядків, адже зміниться порядок символів у порівнюваних рядках. Якщо ж усі змінні будуть токенизуватися однаковим ідентифікатором, то це призведе до постійного повтору однакових символів у результуючому рядку, що збільшить можливість хибно-позитивного результату перевірки рядків на плагіат. Виходячи з цього я вирішив виконувати токенизацію змінних та функцій

тим токеном, який відповідає типу змінної або типу результату, який повертає функція. Це можливе завдяки тому, що тип змінної в більшості випадків вказується перед її оголошенням (`int i;`) або ж після (`var i = 5; //i має тип числа`). Поширені сьогодні лексичні аналізатори виконують токенізацію лише по виду змінної [9], чого в моєму розумінні не достатньо для виконання якісного аналізу з вищеописаних причин. Використання такого алгоритму робить нормалізований код більш стійким до змін структури коду (особливо до перестановки рядків), адже в мовах програмування операції проводяться над однаковими типами даних.

Подібні ключові слова зручно зберігати у файлі, щоб можна було легко додавати та змінювати їх у залежності від потреб. Також у цьому файлі можна зберігати всі роздільники мови. Крім цього, зручною є можливість конкретного вказання символів, на які будуть замінятися частини мови в тексті. Такий файл являтиме собою щось подібне до граматики мови.

У своїй реалізації я створив універсальний файл для мов C, C++, C# та Java, у якому зберігаються всі роздільники та ключові слова, які можна використати при написанні коду (частина цього файлу вказана в додатку А). Файл має наступну структуру:

- символ, який відокремлюватиме логічні частини у файлі;
- перелік роздільників, які використовуються в мові програмування;
- перелік ключових слів мови, подібні з яких розміщені в одному рядку;
- після кожного набору ключових слів указаний символ, яким заміниться це слово в нормалізованому коді.

Використання зовнішнього файлу дозволяє змінювати вигляд нормалізованого коду модифікацією цього файлу без зміни самих алгоритмів нормалізації.

Однією з проблем, яку необхідно вирішити, це додавання в код непотрібних коментарів чи текстів виводу інформації. Так як можлива не лише

зміна існуючого тексту, а й додавання нового, то було вирішено видаляти з коду всі коментарі та не програмний текст. Це не призведе до зміни логіки програми, проте дасть змогу визначати ще один тип плагіату.

Із всього вищесказаного отримали наступні кроки алгоритму нормалізації програмного коду:

- 1) зчитати файл із граматиною мови (роздільники, ключові слова та символи, на які їх замінювати);
- 2) видалити з коду коментарі та не програмний текст;
- 3) знайти та замінити всі ключові слова на відповідні їм токени;
- 4) знайти та замінити числа на токен, який відповідає числовому типу даних;
- 5) решту використаних слів, які не є ключовими для даної мови, замінити на токен, який знаходиться зліва чи справа від цього слова.

Конкретна програмна реалізація цього алгоритму знаходиться в додатку Б.

3.2 Метод пошуку найдовшого спільного підрядка

Найдовший спільний підрядок (англ. longest common substring) – підрядок двох і більше рядків, котрий має найбільшу довжину [2].

Формально, найдовшим спільним підрядком рядків s_1, s_2, \dots, s_n називається рядок w^* , для якого виконується умова $\|w^*\| = \max(\{\|w\| \mid w \subset s_i, i = 1, \dots, n\})$, операція $w \subset s_i$ позначає те, що рядок w є підрядком рядка s_i .

Для знаходження найдовшого спільного підрядка двох рядків s_1 і s_2 , довжина яких m і n відповідно, необхідно заповнити матрицю A_{ij} розміром $(m+1) \times (n+1)$, дотримуючись наступного алгоритму:

$$\begin{cases} A_{0j} = A_{i0} = 0, & j = 0 \dots n, \quad i = 0 \dots m, \\ A_{ij} = 0, \quad s_1[i] \neq s_2[j], & i \neq 0, \quad j \neq 0, \\ A_{ij} = A_{i-1, j-1}, \quad s_1[i] = s_2[j], & i \neq 0, \quad j \neq 0. \end{cases}$$

За даного алгоритму вважається, що символи в рядках нумеруються з одиниці.

Найбільше число A_{uv} в матриці являє собою довжину найдовшого спільного підрядка. Сам підрядок знаходиться за наступним правилом: $s_1[u - A_{uv} + 1] \dots s_1[u]$ та $s_2[v - A_{uv} + 1] \dots s_2[v]$.

Рівень плагіату визначається відношенням отриманої довжини спільного підрядка до довжини рядка, який досліджуємо на плагіат:

$$r(s_1, s_2) = 1 - \frac{|LCS(s_1, s_2)|}{|s_1|}$$

Головним недоліком цього методу є використання великої кількості пам'яті для збереження матриці A . Проте попередня нормалізація коду значно зменшує кількість порівнюваних символів, тому витрати пам'яті не будуть критичними.

Ще одним недоліком цього методу є нестійкість перед доданими у код непотрібними символами. Для прикладу розглянемо умовний варіант вигляду нормалізованого коду програми. Якщо оригінальна програма має вигляд “abcdef” та студент використав код без змін, то алгоритм знайде найдовший спільний підрядок “abcdef” і рівень плагіату становитиме 100%. Проте, додавши в код непотрібну частину, можна сильно зменшити відсоток знайденого плагіату. Нехай після деяких махінацій отримали вигляд коду “abckdef”. Виконавши пошук спільного підрядка, отримали результатом рядок “abc” (або “def”) і рівень плагіату $\approx 50\%$. Тобто ми удвічі знизили ефективність методу елементарною зміною коду.

Щоб зробити алгоритм стійким у даному випадку, повторюватимемо пошук спільного підрядка й видалятимемо його з коду до тих пір, поки не залишиться лише оригінальна частина.

Варто зазначити, що набір символів, із яких може складатися нормалізований рядок, обмежений. А отже перевірка плагіату буде завжди знаходити хоч якийсь його відсоток.

Щоб цього уникнути, необхідно визначити мінімальну довжину спільного підрядка, яка б свідчила про плагіат. Це можна зробити, наприклад, обравши довжину певної структури, яка найчастіше зустрічається в коді програм. Такою структурою я обрав цикл `for`. Оголошення цього циклу загалом має стандартний вигляд: `for (int i = 0; i < 20; i++){ /*код*/ }`. Після токенизації моїм алгоритмом отримуємо наступний вигляд: `f(nn=n;n<n;n++){/*решта токенів*/}`. Довжина такого рядка 16 символів, а отже мінімальною довжиною підрядка, яка б свідчила про плагіат, я обрав 17 символів.

3.3 Метод шинглів

Алгоритм шинглів (англ. w-shingling) – алгоритм, розроблений для пошуку копій та дублікатів тексту у веб-документі. Засіб для виявлення плагіату [3].

Шингли (англ. shingles – луски) – виділені з тексту підпоследовності слів [4].

Алгоритм складається з наступних кроків [4]:

- 1) нормалізація тексту;
- 2) розбиття тексту на шингли;
- 3) обрахування хеш-значень отриманих шинглів;
- 4) випадкова вибірка певної кількості значень контрольних сум;
- 5) порівняння отриманих вибірок.

Рівень плагіату знаходиться за формулою $r(s_1, s_2) = \frac{|S(s_1) \cap S(s_2)|}{|S(s_1) \cup S(s_2)|}$, де $S(s)$ є множиною обраних хеш-значень шинглів рядка.

Розмір шинглів w я обрав із тих самих міркувань, що й мінімальну довжину спільного підрядка. Нормалізація тексту проводиться попередньо й в алгоритм (у попередній теж) передається вже нормалізований рядок. Обрахунок хеш-значення виконую стандартним методом мови C# `string.GetHashCode()`.

Випадкову вибірку я вирішив не обирати, щоб зробити результат виконання алгоритму сталим для однакових вхідних даних. Завдяки хешуванню кожен шингл матиме значення типу `int`. Завдяки цьому алгоритм використовує навіть менше пам'яті, ніж попередній алгоритм пошуку найдовшого підрядка.

Формулу обрахунку коефіцієнту плагіату я теж трохи змінив з огляду на те, що студент може використати у своїй роботі не весь код оригінальної програми, а лише частину (наприклад, виконати не всі вимоги, поставлені завданням лабораторної). Формула після зміни виглядає наступним чином:

$$r(s_1, s_2) = \frac{|S(s_1) \cap S(s_2)|}{|S(s_1)|}.$$

Такий її вигляд дозволяє оцінювати

оригінальність роботи без огляду на розмір роботи, яку було використано за основу.

3.4 Проблеми під час реалізації та отримані результати

Головною складністю в розробці було створення власного алгоритму нормалізації та токенизації коду. У початкових його версіях я намагався всі використані слова в програмі токенизувати окремими символами. Але це було складно реалізувати так, щоб кожен токен мав свій унікальний символ. Доводилося деякі слова замінювати двома чи більше символами в нормалізованому коді. Це призвело до збільшення довжини результуючих рядків та погіршенню роботи алгоритмів, адже вони розраховані на те, що кожен символ являє собою окремий токен. Після цього було прийнято рішення використовувати лише ті символи, які використовуються для токенизації ключових слів мови, які знаходяться у файлі з граматикою.

Щодо цього файлу, то в ньому я теж намагався спочатку вказувати не всі символи для трансляції. Проте динамічна генерація цих символів робить процес виконання програми нормалізації не очевидним і збільшує час його роботи. До того ж, отриманий нормалізований код буде неможливо інтерпретувати (якщо така потреба виникне), бо користувачу не відомий сам алгоритм створення

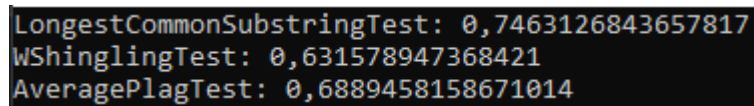
символів. Якщо ж кожне перетворення буде попередньо описане, то завжди можна дізнатися, яке ключове слово приховане під символом.

В алгоритмі пошуку найдовшого спільного підрядка в перших версіях програми я видаляв всі входження знайденого підрядка як із коду, який перевіряємо на плагіат, так і з оригінального коду. Проте на деяких тестових даних це показало себе не ефективно, бо після видалення знайденого підрядку з тексту він міг з'явитися знову в неоригінальній програмі. Для прикладу, нехай оригінальна робота матиме вигляд “abcdkabcdk”, а її копія – “kabcsabcsddk” (тобто була виконана перестановка рядків). Після першого проходження циклу знайдено буде підрядок “abcd”. Після видалення всіх його входжень програми матимуть вигляд “kk” та “kabcsdk” відповідно. Наступним буде знайдено лише підрядок з одного символу “k”, що не є свідченням плагіату. Як бачимо, рівень плагіату за такого підходу буде незначним. Одним із рішень цієї проблеми є видалення підрядку лише із коду, який перевіряємо на плагіат. Проте це значно збільшить час виконання алгоритму й використані ним ресурси. Як компроміс було вирішено видаляти з оригінальної програми лише перше входження підрядка, тоді як із роботи, підозрілої на плагіат, видаляти всі входження.

Основною проблемою методу шинглів була нестійкість перед частковим плагіатом коду (використання лише частини оригінальної програми). Цю проблему було вирішено зміною формули розрахунку коефіцієнта плагіату. Кінцеві версії алгоритмів подано в додатку В.

Якщо порівняти результати роботи алгоритмів, помітимо, що метод шинглів показує менший коефіцієнт плагіату, ніж метод найбільшого спільного рядка. Це можна пояснити наступним прикладом: оригінал $O = \text{“abcd”}$, плагіат $P = \text{“abkcd”}$, $w = 2$. Найдовшими спільними підрядками будуть “ab” та “cd”, після їхнього видалення отримали $P_1 = \text{“k”}$, $r_1 = 1 - 0,2 = 0,8$. $S(O) = \{\text{“ab”}, \text{“bc”}, \text{“cd”}\}$, $S(P) = \{\text{“ab”}, \text{“bk”}, \text{“kc”}, \text{“cd”}\}$, $S(P) \cap S(O) = \{\text{“ab”}, \text{“cd”}\}$, $r_2 = 0,5$. При порівнянні реальних кодів програм такі ситуації виникають нечасто, проте все одно метод шинглів поводить себе менш стійко щодо використання в плагіаті непотрібних

частин коду. Але варто зазначити деякий недолік методу найдовшого підрядка: після видалення частини коду можуть утворюватися нові підпоследовності, які алгоритм визначить плагіатом. Виходячи з вищесказаного, один алгоритм буде вказувати на більший відсоток плагіату, інший – на менший (рис. 1). Істина ж буде десь посередині.



```
LongestCommonSubstringTest: 0,7463126843657817
WShinglingTest: 0,631578947368421
AveragePlagTest: 0,6889458158671014
```

Рисунок 1. Результат виконання програми.

Приклад використання створених класів надано в додатку Г. Результат роботи програми при перевірці на плагіат остаточної та попередньої версії класу нормалізатора наведено в додатку Д. Як можна помітити, при перевірці на плагіат однакових робіт коефіцієнт плагіату рівний одиниці.

ВИСНОВКИ

У роботі було досліджено способи приховування плагіату в роботах студентів та основні типи методів його виявлення. Було обрано, досліджено, модифіковано під конкретне поставлене завдання та реалізовано два методи перевірки тексту на плагіат. Попередньо програмний код зводиться до певного стандартного вигляду за оригінальними алгоритмами токенизації та нормалізації.

Реалізовані методи дозволяють перевіряти на плагіат не тільки код програм, а й текстові роботи, що розширює сферу використання програми.

Завдяки модульній структурі програми (окремі файл із граматикою, класи нормалізації та перевірки коду на плагіат) можлива зміна та покращення її окремих частин без шкоди іншим компонентам.

Під час тестування роботи програми були виявлені та усунені її недоліки. Отриманий продукт виконує покладені на нього завдання та може бути використаний у системах обліку лабораторних робіт або інших системах, де необхідно виконувати перевірку робіт на плагіат.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Chanchal Kumar Roy and James R. Cordy, A Survey on Software Clone Detection Research // Technical Report No. 2007-541, September 2007.
2. Longest common substring problem [Електронний ресурс] – Режим доступу до ресурсу:
https://en.wikipedia.org/wiki/Longest_common_substring_problem
3. w-shingling [Електронний ресурс] – Режим доступу до ресурсу:
<https://en.wikipedia.org/wiki/W-shingling>
4. Алгоритм шинглов [Електронний ресурс] – Режим доступу до ресурсу:
https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%88%D0%B8%D0%BD%D0%B3%D0%BB%D0%BE%D0%B2
5. Совершенный код: нормализация данных [Електронний ресурс] – Режим доступу до ресурсу: <https://ru.hexlet.io/blog/posts/sovershennyy-kod-normalizatsiya-dannyh>
6. Lexical analysis [Електронний ресурс] – Режим доступу до ресурсу:
https://en.wikipedia.org/wiki/Lexical_analysis
7. Обзор автоматических детекторов плагиата в программах [Електронний ресурс] – Режим доступу до ресурсу:
<https://logic.pdmi.ras.ru/~yura/detector/survey.pdf>
8. Ступенчатый метод проверки исходного кода программы на плагиат [Електронний ресурс] – Режим доступу до ресурсу:
<https://nauchkor.ru/pubs/stupenchatyy-metod-proverki-ishodnogo-koda-programmy-na-plagiat-587d36555f1be77c40d58cf6>
9. Flex (lexical analyser generator) [Електронний ресурс] – Режим доступу до ресурсу: [https://en.wikipedia.org/wiki/Flex_\(lexical_analyser_generator\)](https://en.wikipedia.org/wiki/Flex_(lexical_analyser_generator))
10. Наибольшая общая подстрока [Електронний ресурс] – Режим доступу до ресурсу: http://wp.wiki-wiki.ru/wp/index.php/%D0%9D%D0%B0%D0%B8%D0%B1%D0%BE%D0%BB%D1%8C%D1%88%D0%B0%D1%8F_%D0%BE%D0%B1%D1%89%

[D0%B0%D1%8F %D0%BF%D0%BE%D0%B4%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0](#)

11. Шингл как метод для сравнения уникальности статей [Электронный ресурс] – Режим доступа до ресурсу: <https://textxpert.ru/shingl-tainstvennyiy-i-neponyatnyiy/>
12. Выявление плагиата в исходном коде программ студентов [Электронный ресурс] – Режим доступа до ресурсу: <https://docplayer.ru/43636911-Vyyavlenie-plagiata-v-ishodnom-kode-programm-studentov.html>

ДОДАТКИ

ДОДАТОК А

Grammar.txt: Блокнот

Файл Редагування Формат Вигляд Довідка

```

^=
.*
<<
<<=
=>
>>
>>=
-->*
>>>
>>>=
Э
byte sbyte int float decimal double uint ulong ushort long shortЭn
synchronized abstract override readonly static virtual private protected publicЭd
bool boolean true falseЭb
forЭя
foreachЭё
doЭr
whileЭс
usingЭs
char stringЭt
voidЭq
asЭw
baseЭе
breakЭr

```

ДОДАТОК Б

```

using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
using System.ComponentModel;
using System.Linq;

namespace kurs
{
    class Normalizator
    {
        string separator = "";
        static List<string> punctuator = new List<string>();//роздільники, записані у файлі
граматики
        static List<string> programKeywords = new List<string>();//усі ключові слова, які є в
програми
        static List<string> numbers = new List<string>();
    }
}

```

```

static Dictionary<string, string> keywordsDictionary = new Dictionary<string, string>();

public Normalizator(string grammar_file_path)//розташування файлу з граматиною
(роздільники + ключові слова)
{
    ReadGrammar(grammar_file_path);//зчитуємо граматику
}

public string GetNormalizedCode(string code)
{
    FindProgramKeywords(code);//шукаємо всі ключові слова, які є в програмі
    FillDictionary();//заповнюємо словник ключове слово - токен
    return NormalizeCode(code);//повертаємо нормалізований код
}

private string NormalizeCode(string code)//виконує нормалізацію коду
{
    code = RemoveComments(code);//видалення коментарів
    code = RemoveText(code);//видалення не програмного тексту

    //кожне ключове слово замінюється на відповідний йому токен
    programKeywords.Sort(CompareStringLength);
    foreach (string s in programKeywords)
    {
        if (s.Length > 1)
        {
            code = code.Replace(s, separator + keywordsDictionary[s]);
        }
        else
        {
            if (keywordsDictionary.ContainsValue(s))
            {
                code = code.Replace(separator + s, separator + separator);
                code = code.Replace(s, separator + keywordsDictionary[s]);
                code = code.Replace(separator + separator, separator + s);
            }
            else
            {
                code = code.Replace(s, separator + keywordsDictionary[s]);
            }
        }
    }
}

//числа замінюються на токен, який відповідає типу int
numbers.Sort(CompareStringLength);
foreach (string s in numbers)
    code = code.Replace(s, "#number#");
code = code.Replace("#number#. #number#", "#number#");
code = code.Replace("#number#, #number#", "#number#");

```

```

code = code.Replace("#number#", separator + keywordsDictionary["int"]);
code = code.Replace(separator, "");

//видалення відступів
code = code.Replace(" ", "");
code = code.Replace("\n", "");
code = code.Replace("\r", "");

return code;
}

private int CompareStringLength(string a, string b)//порівнює довжину двох рядків
{
    if (a.Length == b.Length)
        return 0;
    if (a.Length > b.Length)
        return -1;
    return 1;
}

private void FillDictionary()//заповнює словник токенів
{
    bool allKeywordsAdded = true;
    do
    {
        for (int i = 1; i < programKeywords.Count; ++i)
        {
            if (!keywordsDictionary.ContainsKey(programKeywords[i]))
            {
                if (keywordsDictionary.ContainsKey(programKeywords[i - 1]))
                {
                    keywordsDictionary.Add(programKeywords[i],
keywordsDictionary[programKeywords[i - 1]]);
                }
                else
                {
                    if (keywordsDictionary.ContainsKey(programKeywords[i + 1]))
                    {
                        keywordsDictionary.Add(programKeywords[i],
keywordsDictionary[programKeywords[i + 1]]);
                    }
                    else
                    {
                        allKeywordsAdded = false;
                    }
                }
            }
        }
    }
}

```

```

    while (!allKeywordsAdded);
}

private void ReadGrammar(string path)//зчитує граматику з файлу
{
    StreamReader sr = new StreamReader(path);

    separator = sr.ReadLine();//роздільник у файлі з граматиною

    //зчитує роздільники в мові програмування
    string line = sr.ReadLine();
    while (!line.Equals(separator))
    {
        punctuator.Add(line);
        line = sr.ReadLine();
    }

    //зчитує всі стандартні ключові слова
    line = sr.ReadLine();
    while (!line.Equals(separator))
    {
        string[] spl = line.Split(separator);
        string[] buf = spl[0].Split(' ');
        if (spl.Length > 1)
        {
            foreach (string s in buf)
            {
                if (!keywordsDictionary.ContainsKey(s))
                    keywordsDictionary.Add(s, spl[1]);
            }
        }
        line = sr.ReadLine();
    }

    sr.Close();
}

private void FindProgramKeywords(string code)//пошук усіх використаних у програмі
ключових слів
{
    code = RemoveComments(code);//видалення коментарів
    code = RemoveText(code);//видалення не програмного тексту
    code = code.Replace("\n", " ");//видалення абзаців
    code = code.Replace("\r", " ");
    foreach (string s in punctuator)//видалення роздільників мови програмування
        code = code.Replace(s, " ");

    string[] splitCode = code.Split(' ');//тут знаходяться всі використані ключові слова
    foreach (string s in splitCode)//знаходить всі унікальні ключові слова

```

```

{
    if (s.Length > 0 && !programKeywords.Contains(s))
    {
        if (!IsNumber(s))
        {
            programKeywords.Add(s);
        }
        else
        {
            if(!numbers.Contains(s))
                numbers.Add(s);
        }
    }
}

private bool IsNumber(string n)//перевірка, чи є рядок числом
{
    if (!Char.IsNumber(n[0]))
        return false;
    for (int i = 1; i < n.Length; ++i)
        if (!Char.IsNumber(n[i]))
            return false;
    return true;
}

private string RemoveComments(string code)//видаляє коментарі
{
    //видалення коментарів виду //текст
    while (code.Contains("//"))
    {
        int i = code.IndexOf("//");
        string buf1 = code.Substring(0, i);
        string buf2 = code.Remove(0, i + 2);
        i = buf2.IndexOf("\n");
        if (i != -1) buf2 = buf2.Remove(0, i);
        code = buf1 + buf2;
    }

    //видалення коментарів виду /*текст*/
    while (code.Contains("/*"))
    {
        int i = code.IndexOf("/*");
        string buf1 = code.Substring(0, i);
        string buf2 = code.Remove(0, i + 2);
        i = buf2.IndexOf("*/");
        if (i != -1) buf2 = buf2.Remove(0, i + 2);
        code = buf1 + "\n" + buf2;
    }
}

```

```

        return code;
    }

    private string RemoveText(string code)//видаляє не програмний текст
    {
        //видалення тексту виду "текст"
        code = code.Replace("\\\\", " ");
        while (code.Contains("\\"))
        {
            int i = code.IndexOf("\");
            string buf1 = code.Substring(0, i);
            string buf2 = code.Remove(0, i + 1);
            i = buf2.IndexOf("\");
            if (i != -1) buf2 = buf2.Remove(0, i + 1);
            code = buf1 + buf2;
        }

        //видалення символів виду 'с'
        code = code.Replace("\\'", " ");
        while (code.Contains("\'"))
        {
            int i = code.IndexOf("\'");
            string buf1 = code.Substring(0, i);
            string buf2 = code.Remove(0, i + 1);
            i = buf2.IndexOf("\'");
            if (i != -1) buf2 = buf2.Remove(0, i + 1);
            code = buf1 + buf2;
        }

        return code;
    }
}

```

ДОДАТОК В

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace kurs
{
    class Plagiator
    {
        int MIN_LCS_LENGTH = 17; //мінімальна довжина однакової ділянки коду, яка свідчить
        про плагіат
    }
}

```

```

    public double LongestCommonSubstringTest(string test, string other)//модифікований
метод найдовшого спільного рядка
    //test - рядок, який перевіряємо на плагіат
    //other - рядок, із яким перевіряємо
    {
        int originalLength = test.Length;//початкова довжина рядка
        int lcsLength;//довжина найдовшого спільного рядка (HCP)
        do//повторюємо дії, поки HCP не буде закоротким
        {
            int n = test.Length;
            int m = other.Length;
            int[,] matr = new int[n, m];
            lcsLength = 0;
            int maxI = 0;
            for (int i = 0; i < n; i++)//заповнює матрицю пошуку HCP
            {
                for (int j = 0; j < m; j++)
                {
                    if (test[i] == other[j])
                    {
                        matr[i, j] = (i == 0 || j == 0) ? 1 : matr[i - 1, j - 1] + 1;
                        if (matr[i, j] > lcsLength)
                        {
                            lcsLength = matr[i, j];
                            maxI = i;
                        }
                    }
                }
            }

            if (lcsLength > 0)//якщо HCP знайдено
            {
                string lcs = test.Substring(maxI + 1 - lcsLength, lcsLength);//знаходимо цей HCP
                test = test.Replace(lcs, "");//видаляємо з рядка, який перевіряємо, всі входження
HCP
                other = other.Remove(other.IndexOf(lcs), lcsLength);//із рядка, з яким перевіряємо,
видаляємо лише перше входження HCP
            }
        }
        while (lcsLength >= MIN_LCS_LENGTH);//повторюємо дії, поки HCP не буде закоротким

        //коефіцієнт плагіату знаходиться як відношення довжини унікальної частини тексту до
всієї довжини рядка
        return 1.0 - (double)test.Length / originalLength;
    }

    public double WShinglingTest(string test, string other)//метод шинглів
    {
        //шукаємо шингли рядка test та відразу рахуємо їхній хеш

```

```

int testCountShingles = test.Length - MIN_LCS_LENGTH + 1;
List<int> testShingles = new List<int>();
for (int i = 0; i < testCountShingles; ++i)
{
    testShingles.Add(test.Substring(i, MIN_LCS_LENGTH).GetHashCode());
}

//шукаємо шингли рядка other та відразу рахуємо їхній хеш
int otherCountShingles = other.Length - MIN_LCS_LENGTH + 1;
List<int> otherShingles = new List<int>();
for (int i = 0; i < otherCountShingles; ++i)
{
    otherShingles.Add(other.Substring(i, MIN_LCS_LENGTH).GetHashCode());
}

//коефіцієнт плагіату знаходиться як відношення кількості однакових хешів до кількості
всіх хешів рядка test
return (double)testShingles.Intersect(otherShingles).Count() /
testShingles.Distinct().Count();
}

public double AveragePlagTest(string test, string other)//середнє арифметичне обох
методів
{
    return (LongestCommonSubstringTest(test, other) + WShinglingTest(test, other)) / 2;
}
}
}

```

ДОДАТОК Г

```

using System;
using System.IO;

namespace kurs
{
    class Program
    {
        static void Main(string[] args)
        {
            StreamReader sr = new StreamReader("input1.cs");
            string s1 = sr.ReadToEnd();
            sr.Close();
            sr = new StreamReader("input2.txt");
            string s2 = sr.ReadToEnd();
            sr.Close();
            Normalizator norm = new Normalizator("Grammar.txt");
            s1 = norm.GetNormalizedCode(s1);
            s2 = norm.GetNormalizedCode(s2);
        }
    }
}

```



```

StreamWriter sw = new StreamWriter("output.txt");
sw.WriteLine(s2);
sw.Close();

Plagiator plag = new Plagiator();
Console.WriteLine("LongestCommonSubstringTest: " +
plag.LongestCommonSubstringTest(s1, s2));
Console.WriteLine("WShinglingTest: " + plag.WShinglingTest(s1, s2));
Console.WriteLine("AveragePlagTest: " + plag.AveragePlagTest(s1, s2));

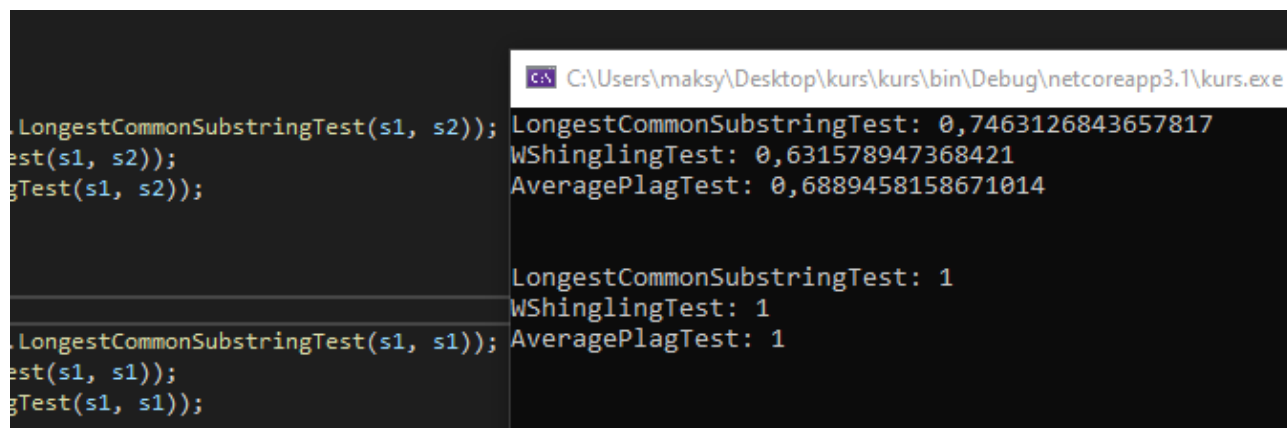
Console.WriteLine();
Console.WriteLine();

Console.WriteLine("LongestCommonSubstringTest: " +
plag.LongestCommonSubstringTest(s1, s1));
Console.WriteLine("WShinglingTest: " + plag.WShinglingTest(s1, s1));
Console.WriteLine("AveragePlagTest: " + plag.AveragePlagTest(s1, s1));

Console.ReadKey();
}
}
}

```

ДОДАТОК Д



```

C:\Users\maksy\Desktop\kurs\kurs\bin\Debug\netcoreapp3.1\kurs.exe
LongestCommonSubstringTest(s1, s2)); LongestCommonSubstringTest: 0,7463126843657817
est(s1, s2)); WShinglingTest: 0,631578947368421
gTest(s1, s2)); AveragePlagTest: 0,6889458158671014

LongestCommonSubstringTest: 1
WShinglingTest: 1
AveragePlagTest: 1
LongestCommonSubstringTest(s1, s1));
est(s1, s1));
gTest(s1, s1));

```