МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
Факультет електроніки та інформаційних технологій

Кафедра комп‘ютерних наук

Кваліфікаційна робота магістра

ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ ОЦІНКИ РІВНЯ ПОДІБНОСТІ ТЕКСТІВ ПРОГРАМ

Здобувач освіти гр. ІНмз – 11с	Олександр КНИШ
Науковий керівник, доцент, к.т.н.	Сергій ПЕТРОВ
В. о. завідувача кафедри доцент, к.т.н.	Ігор ШЕЛЕХОВ

Суми 2022

Сумський державний університет

(назва вузу)

Факультет *ЕлІТ* Кафедра *Комп'ютерних наук*

Спеціальність *«122 - Комп'ютерні науки»*

Затверджую:

В.о. зав.кафедрою _____

_____

"_____" _____ 20___р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ МАГІСТРА

Книшу Олександру Борисовичу

(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) *Інформаційна технологія визначення рівня схожості текстів програм студентів*

затверджую наказом по інституту від "____" _____ 20 ___ р. № _____

2. Термін здачі здобувачем вищої освіти закінченої роботи_____

3. Вхідні данні до роботи_____

_____

_____

_____

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)

1) Огляд видів дублювання кодів та аналіз технологій, котрі застосовуються для виявлення нечітких дублікатів;2) Постановка та формування завдання дослідження; 3) Розробка моделі технології визначення рівня схожості програмного забезпечення; 4) Створення додатків з лістингами програм; 5) Аналіз результатів.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) __

_____

6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання _____

Керівник _____
(підпис)

Завдання прийняв до виконання _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів дипломного проекту (роботи)	Термін виконання проекту (роботи)	Примітка
1.	Огляд видів дублювання кодів та аналіз технологій, котрі застосовуються для виявлення нечітких дублікатів.		
2.	Постановка та формування завдання дослідження.		
3.	Розробка моделі технології визначення рівня схожості програмного забезпечення.		
4.	Створення додатків.		
5.	Аналіз результатів та оформлення пояснювальної записки до дипломного проекту.		

Студент – дипломник _____
(підпис)

Керівник проекту _____
(підпис)

РЕФЕРАТ

Записка: 60 стор., 17 рис., 3 таблиці, 3 додатки, 24 літературних джерел.

Об’єкт дослідження — Інформаційна технологія визначення рівня схожості текстів програм студентів.

Мета роботи — розробка технології виявлення визначення рівня схожості, нечітких дублікатів текстів програмного забезпечення.

Результати — у ході роботи проведено огляд видів дублювання кодів та аналіз технологій, методів та інструментів, котрі застосовуються для виявлення нечітких дублікатів текстів програмного забезпечення у роботах студентів та у інших проектах, де використовуються різні підходи. У роботі проаналізовані підходи, засновані на метриках, аналізі абстрактного синтаксичного дерева (AST), узгодженні шаблонів послідовності токенів або графа залежностей програми (PDG). У дипломній роботі реалізовано та описано технологію визначення рівня схожості текстів програмного забезпечення, а саме виявлення нечітких дублікатів, засновану на абстрактному синтаксичному дереві. У процесі реалізації поставленої задачі обчислюються хеш-значення вузлів синтаксичного дерева та порівнюються між собою. Алгоритм Winnowing був застосований для оптимізації використаної пам’яті. Розроблене програмне забезпечення дозволяє, в автоматичному режимі, визначати рівень схожості текстів програм студентів, виявляючи дублікати коду на різних мовах програмування.

ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ ВИЗНАЧЕННЯ РІВНЯ СХОЖОСТІ ТЕКСТІВ ПРОГРАМ СТУДЕНТІВ, INFORMATION TECHNOLOGY FOR DETERMINING THE LEVEL OF SIMILARIRITY OF STUDENT PROGRAM TEXTS, ЦИФРОВИЙ ВІДБИТОК, АБСТРАКТНЕ СИНТАКСИЧНЕ ДЕРЕВО, АЛГОРИТМ WINNOWING

ЗМІСТ

ВСТУП 6

1. АНАЛІЗ ПРОБЛЕМИ 9

1.1 Існуючі види дублювання коду 9

1.2 Огляд методів виявлення нечітких дублікатів 12

1.2.1 Текстові підходи засновані на рядках (string-based, line-based) 14

1.2.2 Лексичні підходи (token-based) 15

1.2.3 Синтаксичні підходи (tree-based) 15

1.2.4 Семантичні підходи (semantics-based) 17

1.2.5 Гібридні підходи 18

1.3 Постановка задачі 19

2. АЛГОРИТМ ДОСЛІДЖЕНЬ 21

2.1 Лексичний аналіз вхідного коду 22

2.2 Цифровий відбиток піддерев 23

2.2.1 Ковзаюча хеш функція 24

2.2.2 Алгоритм Winnowing та приклад його роботи 26

2.3 Індексація цифрових відбитків 27

2.4 Вилучення кластерів зі збігами 28

3. ПРАКТИЧНА РЕАЛІЗАЦІЯ ПОСТАВЛЕНОЇ ЗАДАЧИ 29

3.1 Практична реалізація абстрактного синтаксичного дерева 30

3.2 Реалізація цифрових відбитків піддерев 32

3.3 Практична реалізація індексування цифрових відбитків 34

3.4 Результатів аналізу, їх представлення. 35

3.5 Тестування системи 36

ВИСНОВКИ 39

СПИСОК ЛІТЕРАТУРИ

Ошибка! Закладка не определена.

ДОДАТКИ 43

ВСТУП

Дублювання коду в комп'ютерному програмуванні є поширеною помилкою проектування програмного забезпечення, де послідовність інструкцій існує аналогічна (якщо не ідентична) у багатьох частинах вихідного коду у вигляді програмного забезпечення. Тобто дублювання коду – це повторювання послідовності вихідного коду в рамках однієї або декількох програм. Таку послідовності іноді називають клонами коду чи просто клонами. У більшості випадків дублювання виникає тоді, коли в проекті працює кілька людей, причому над різними його частинами. Вони працюють над подібними завданнями, але не знають, що колега вже написав подібний код, який можна використовувати замість написання власного. Зустрічається і непряме дублювання, коли конкретні ділянки коду відрізняються зовні, хоча і виконують одну і ту ж задачу. Таке дублювання буває досить складно виявити і виправити. В окремих випадках дублювання створюється навмисно. Найчастіше, в поспіху, коли підтискають терміни здачі проекту. Початківець програміст бачить у вже написаному коді фрагмент, що виглядає " майже так, як потрібно» і не може встояти перед спокусою просто скопіювати код кудись в інше місце. Автоматизований процес, який дає можливість знаходити дублікати у вихідному коді, має назву виявлення клонів. Дубльований код створює проблеми обслуговування, важливість яких зростає зі збільшенням кількості дубльованого коду. Чим більше дублюється коду, тим більше коду потрібно підтримувати. Копіювання коду є небажаним і може призвести до значного збільшення проблем таких, як [1,2]:

- ✓ зростання важкості підтримки;
- ✓ виникання помилки при програмуванні в клонованому коді приводить до її неоднократного повторення;
- ✓ зниження читабельності коду;

- ✓ якщо дубльований код повинен розвиватись, то потрібні зміни у всіх клонах;
- ✓ зростання розміру системи;
- ✓ зниження швидкодії та продуктивності системи.

Проблема обслуговування клонованого коду полягає ще і в тому, що при виявлені помилок у одному фрагменті коду, що копіювався, в всіх подібних фрагментах потрібно перевірити, а при необхідності виправити одну і ту саму помилку. Інші види технічного обслуговування, такі як розширення або адаптації, також треба застосовувати декілька разів. Якщо копії не були задокументовані, їх дуже важко виявити і виправити у великих системах. Тому для виправлення ситуації з клонованими програмними кодами, коли мова іде про великі проекти, використовуються автоматичні методи.

Різні дослідження показують, що значна частина, від 7% до 23% коду у великих системах програмного забезпечення є дублікатами [3,4]. Дублювання коду суперечить основному принципу теорії баз даних, а саме: уникайте надмірності. Невиконання цього принципу призводить до аномалій оновлення, які сильно збільшують витрати на обслуговування коду. В цьому випадку одне і те ж зміна потрібно ввести в усі дублікати. І в кращому випадку, час витрачений на внесення змін, і тестування коду збільшується пропорційно кількості дублікатів. А в гіршому-деякі місця в коді можуть бути пропущені, і виправлення всіх помилок може затягнуться на місяці або навіть роки. Багато великих компаній витрачають на утримання існуючих систем більше, ніж на розвиток нових.

Дублювання коду - це проблема, що також часто зустрічається і на академічних курсах, деякі студенти використовують чужі проекти, копіювати чужі коди, зі змінами або без них і отримувати некоректні оцінки, загрожуючи цілісності академічної системи. [5,6].

Отже в загальному випадку, дублікатами є будь-які семантичні одиниці програми, що реалізують один алгоритм однаковим способом, але на практиці дублікати визначаються саме за рахунок порівняння послідовностей певних

елементів, у вигляді яких представляється вихідний код програми. У даній роботі буде розглянута проблема дублювання коду в програмних продуктах студентів, які створюються в процесі отримання освіти чи під час проведення олімпіад, інших конкурсних проєктів з програмування. Автоматична перевірка дублікатів коду спрощує процес перевірки та оцінювання навчальних робіт студентів.

Актуальність проблеми полягає у знаходженні дублікатів та зводиться до пошуку максимальних послідовностей синтаксичних конструкцій у вихідному коді програми, для яких індекс подібності перевищує задане граничне значення, в автоматичному режимі. Реалізація відстеження клонів не повинна залежати від мови програмування, на якій створений продукт, котрий перевіряється, але повинна забезпечити достатній рівень точності.

1. АНАЛІЗ ПРОБЛЕМИ

1.1 Існуючі види дублювання коду

В IT співтоваристві немає чіткої сформульованої класифікації типів клонування, але до однієї з найбільш конструктивних, за способом їх походження, є запропонована нижче структура:

Типи клонів:

- ✓ Identical

Простий copy-paste. З точністю до пробілів, табуляції та коментарів.

- ✓ Renamed

Теж copy-past, але трохи складніше: з точністю до літерів, змінних констант, КЛАСІВ, типів, оформлення та коментарів. В основному це дублікати, які були перенесені зі зміною: назва функцій, змінних і можливим форматуванням.

- ✓ Similar

У скопійований код внесені більш серйозні модифікації. Це можуть бути додаткові рядки або невеликі зміни в уже наявних. У них змінюється логіка і вже всередині них виникають розриви, серед яких потрібно шукати схожість.

- ✓ Semantic

Functional similarity, semantic similarity. Якщо дві ділянки коду роблять одне і теж, то вони функціонально однакові. Хоча написані можуть бути дуже по-різному і навіть на різних мовах.

Спираючись на описані вище види клонів, можна структурувати їх у вигляді схеми (Рисунок 1.1):

Нижче розглянемо 4 типи клонування коду за текстовими і за функціональними ознаками. Ці типи були запропоновані авторами у працях [4,7]:

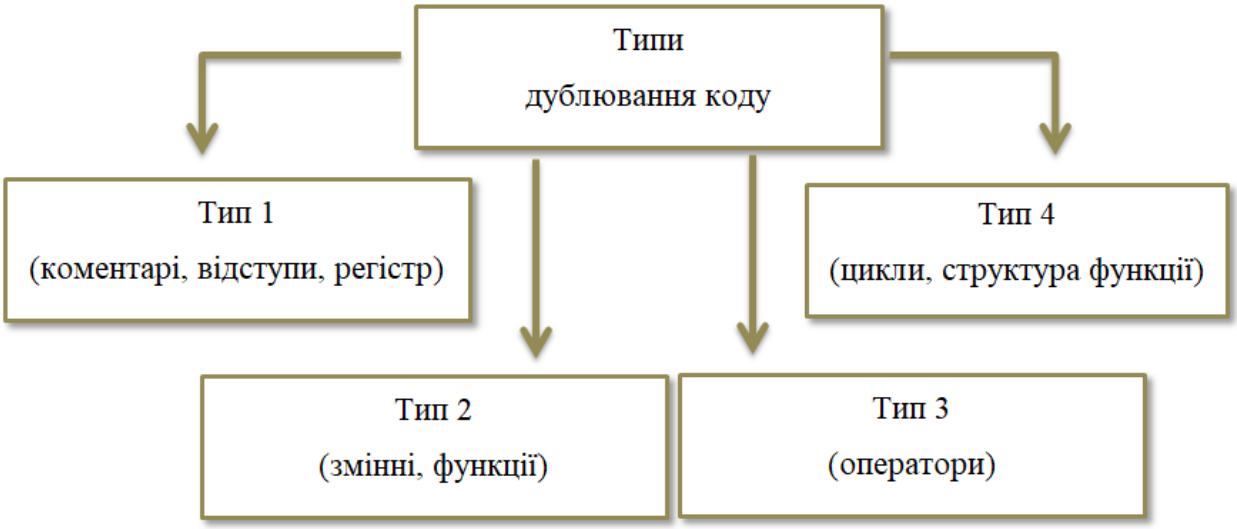

Рисунок 1.1 – чотири типи копіювання коду

- Тип 1 – код дублюється з мінімальними змінами (або і без них). Такі зміни стосуються коментарів, відступів чи регістрів (Рисунок 1.2)

```
1. int gcd(int a, int b) {
2.     if (b == 0) return a;
3.     else return gcd(b, a % b);
4. }
```

```
1. int gcd(int a, int b) {
2.     if (b == 0)
3.         return a;
4.     else
5.         return gcd(b, a % b);
6. }
```

Рисунок 1.2 - приклад клонування коду першого типу

- Тип 2 - клонування фрагменту коду, зі змінами в ідентифікаторах. У такому випадку можуть відбуватися зміни назв: змінної, класу, типу, функції і.п. (Рисунок 1.3)

```
1. int gcd(int a, int b) {
2.     if (b == 0) return a;
3.     else return gcd(b, a % b);
4. }
```

```
1. long gcd(long c, long d) {
2.     if (d == 0) return c;
3.     else return gcd(d, c % d);
4. }
```

Рисунок 1.3 - приклад клонування коду другого типу

- Тип 3 - фрагменти коду типу 1 чи 2, котрі у подальшому модифікуються шляхом зміни, додавання чи вилучення операторів. (Рисунок 1.4)

```
1. int gcd(int a, int b) {
2.     if (b == 0) return a;
3.     else return gcd(b, a % b);
4. }
```

```
1. int gcd(int a, int b) {
2.     if(b == 0)
3.         return a;
4.     if(a == 0)
5.         return b;
6.     return gcd(b, a % b);
7. }
```

Рисунок 1.4 - приклад клонування коду третього типу

- Тип 4 - декілька фрагментів коду, які виконують одні й ті ж самі функції, але імплементовані різними синтаксичними конструкціями, вони навіть можуть бути реалізовані різними мовами програмування. (Рисунок 1.5)

Клони 4 типу показують, лише те, що різні частини програми, які написані абсолютно по-різному, вирішують одну поставлену задачу, яка відомо заздалегідь. Даний тип не може бути врахований у вирішенні поточної задачі.

```
1. int gcd(int a, int b) {
2.     if (b == 0) return a;
3.     else return gcd(b, a % b);
4. }
```

```
1. int gcd(int a, int b) {
2.     while(b != 0) {
3.         swap(a, b);
4.         b = b % a;
5.     }
6.     return a;
7. }
```

Рисунок 1.5 - приклад клонування коду третього типу

Розробники програмного забезпечення та студенти, у ході навчання, можуть одночасно застосовувати декілька різних типів змін, а також можуть додатково змінювати певні частини програми, наприклад, додавання деяких невідповідних функцій. Ці фактори виявляти плагіат коду на практиці. Клони 1 та 2-го типів засвідчують про ймовірне копіювання, клони 3-го типу можливі у випадках, коли вирішуються типові алгоритмічні задачі. Роками розробники використовували в своїх проектах copy-past, а потім викладали в загальний доступ, де цикл повторювався. Легко використовувати готовий код, але як же складно його змінювати і доповнювати, коли він весь складається з дублікатів, скопійованих у інших розробників. Щоб не потрапити в таку ситуацію, розглянемо методи боротьби з клонами.

1.2 Огляд методів виявлення нечітких дублікатів

Очевидно, що надмірне дублювання коду тягне за собою збільшення його розмірів. Окрім збільшення часу компіляції, програміст ускладнює розуміння коду-короткочасна пам'ять людини оптимально працює з 7±2 об'єктами [21], а наявність клонів призводить до штучного зниження цього порогу через необхідність враховувати дублювання коду.

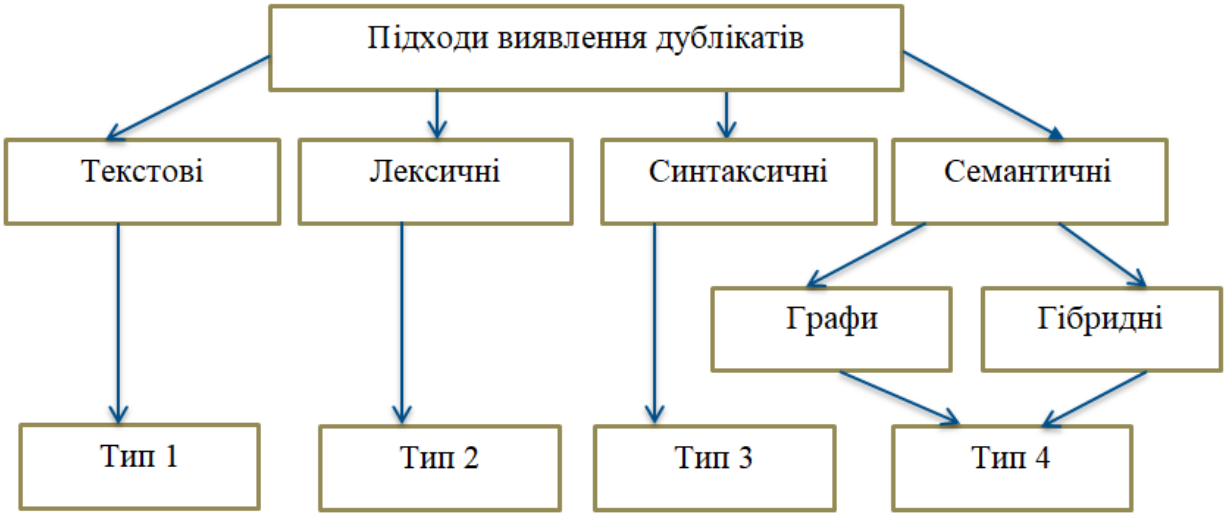

Рисунок 1.6 – Основні підходи практичного виявлення дублікатів

Більшість із запропонованих в літературі підходів для виявлення клонів фокусуються головним чином на синтаксичної схожості фрагментів коду, оскільки визначення семантичної схожості дуже складно.

Методи виявлення дублікатів пов'язані з представленнями коду на яких вони базуються. Клони типу 1, як правило виявляються за допомогою текстових підходів, вони розбивають програмний код на рядки (зазвичай розподіл здійснюється по рядках вихідного файлу), тип 2 аналізується та виявляється за допомогою лексичних підходів, іншими словами підходів на основі токенів, вихідний код розбивається на лексеми (вирази, оператори), утворюючи так звану послідовність токенів. Клони типу 3 - за допомогою синтаксичних підходів, дані методи представляють програмний код у вигляді абстрактного синтаксичного дерева (АСД). Точні або близькі збіги піддерев можуть бути знайдені шляхом їх порівняння з АСД. В якості альтернативного підходу може застосовуватися метод відбитків (fingerprints) для піддерев. Піддерева зі схожими відбитками є ознакою дублювання коду. Для виявлення клонів 4 типу використовують методи, що враховують семантичну складову програмного коду. Семантичні підходи – це підходи на основі графів і гібридні.

В методі використовуються граф залежності (program dependency graphs) і розщеплення (slicing) програми для пошуку ізоморфних підграфів (тобто мають ідентичну форму) з метою ідентифікації клонів. Також існує метод, що дозволяє групувати виявлені дублікати до тих пір, поки зберігається семантична складова початкового коду. Такий підхід може застосовуватися для автоматичної процедури вилучення функцій при рефакторингу коду. Однак його масштабування для використання у великих програмних продуктах досить важко. Підходи на основі графів створюють графи програмних залежностей, а гібридні підходи об'єднують кілька різних методів [8,9].

1.2.1 Текстові підходи засновані на рядках (string-based, line-based)

Текстові підходи ґрунтуються на текстовому порівнянні всіх рядків коду програми. Таким чином, кожен фрагмент коду представляється у вигляді послідовності рядків. Два фрагменти вважаються схожими, якщо складові їх рядки схожі між собою. Як приклад, заснований на цьому методі, програма «Dup», яка дозволяє знаходити дублікат або пов'язаний код у великих програмних продуктах. При цьому можна або шукати повністю ідентичні фрагменти, або проводити пошук секцій коду, які збігаються між собою за винятком деяких параметрів, таких як: імена змінних і константи. Крім того,» Dup " може побудувати графік структурної складності системи, що відображає незвично складні файли, а також фрагменти коду, які слід рефакторингувати. Такий підхід майже не використовує трансформацію та не проводить нормалізацію вихідного коду перед тим, як проводяться фактичні порівняння, крім видалення коментарів та зайвих символів пробілу. За допомогою алгоритмів хешування, або суфіксного дерева виконується пошук схожих підстрок, котрі збігаються по розміру, іншими словами, належної довжини. Даний алгоритми використовують автори у роботах [10,11]. Перевага текстового підходи, що він не залежать від мови програмування, але дуже великий недолік, що зазвичай такий підхід може виявити лише дублікати типу

1, тому він вкрай рідко застосовуються для практичного вирішення даної задачі.

1.2.2 Лексичні підходи (token-based)

Алгоритми в яких застосовується лексичний аналіз, засновані на токенах (token-based), трансформують вихідний код в послідовність токенів. Потім ця послідовність аналізується на входження дублюючих підпослідовностей, що може служити ознакою програмних клонів. У порівнянні з підходами, заснованими на рядках, методи, що базуються на токенах, більш стійкі до таких змін коду, як форматування, додавання пробілів і коментарів. Перетворення вихідного коду в послідовність лексичних токенів відбувається з урахуванням мови програмування.

Бренда Бейкер у своїй роботі [3] запропонувала знаходити клони на основі токенів. Вихідний код, в процесі реалізації алгоритму пошуку клонів, конвертується у послідовність токенів, за допомогою використання лексичного аналізатора. У процесі токени поділяються на два типи: параметричні та непараметричні, перші кодуються за допомогою індексу входження у строку, а другі рахуються, за допомогою функтора хешування. Далі всі префікси, що отримуються з послідовності символів, представляються у вигляді суфіксного дерева. Умова пошуку клону полягає у тому, якщо два суфікси мають один загальний префікс, то відповідно, префікс зустрічається більше одного разу, а це може вважатися знайденим дублікатом.

1.2.3 Синтаксичні підходи (tree-based)

В основі підходу лежить синтаксичний аналізатор, що використовується для перетворення вихідного коду програмного продукту у абстрактні синтаксичні дерева, вони потім можуть бути обстежені за допомогою підходів порівняння дерев або структурних метрик з метою пошуку клонів.

Синтаксичні підходи поділені на:

– деревоподібні. За допомогою аналізатора відбувається побудова абстрактного синтаксичного дерева (АСД), де порівнянню підлягають його піддерева. Виявлення більш складного типу повторюваних блоків у коді, якому сприяють методи цієї категорії, можливе завдяки абстракції дерев вихідного коду;

– метричні (метричний підхід). Базою виступають метрики. Проводиться обчислення ряду метрик, відповідних кожному з блоків вихідного коду. Для виявлення клонування відбувається аналіз та порівняння отриманих метричних векторів.

Підходи на основі порівняння дерев: методи на основі дерев, як вже зазначалось раніше, визначають дублікати шляхом знаходження подібних піддерев. При цьому, імена змінних, класів, літерали та інші токени в програмному забезпеченні можуть бути абстраговані та представлені в деревовидному поданні, що дає змогу більш точно виявляти клони. Такий підхід було представлено авторами у праці [12] та реалізовано у інструменті DECKARD. Основна ідея даного підходу полягає в обчисленні певних характеристик векторів для апроксимації та структурування інформації в АСД в евклідовому просторі. На наступному кроці використовується хешування і ураховується розташування (метод Locality-sensitive hashing - LSH) для того, щоб кластеризувати схожі вектори з використанням метрики евклідової відстані. Дублікатами вважаються результуючі кластери векторів.

Підходи на основі метрик: у всіх методах на основі метрик проводиться порівняння векторів показників, які збираються із фрагментів коду, без застосування безпосереднього порівняння коду. Для однієї чи декількох синтаксичних одиниць, до них відносяться: клас, функція, метод, оператор, проводиться обчислення набору програмних метрик, які мають назву функцій відбитків пальців, іншими словами цифрових відбитків, а потім значення отриманих метрик порівнюються для пошуку клонів кодів відповідних синтаксичних одиниць. У переважній більшості випадків вихідний код програмного продукту спочатку аналізується у вигляді абстрактного

синтетичного дерева, чи графу потоку управління, а потім на їх основі обчислюються метрики [13,14].

Метод запропонований авторами [15] використовує представлення яке має назву Intermediate Representation Language (IRL) для характеристики кожної функції у вихідному коді. Клон коду визначається лише парою цілих тіл функцій, які мають однакові метричні значення.

1.2.4 Семантичні підходи (semantics-based)

Існують підходи, які орієнтовані на семантику, та використовують статичний аналіз програм, з метою надання більш точної інформації, аніж просто синтаксична подібність. У даних методах програмний код представлений у вигляді графа залежностей програми (PDG). Граф залежностей програми – це об'єднаний граф потоку даних і графа потоку управління. Вершини PDG - Це інструкції програми, а ребра – залежності між ними. Є дві основних типів ребер: ребра виражають залежності за даними і ребра виражають залежності по управлінню. PDG найбільш загальне уявлення програми, в ньому зберігається вся інформацію про його семантиці і структурі, що дозволяє більш точно аналізувати програму. Воно використовується в задачах оптимізації і пошуку семантично схожих ділянок коду (приклад PDF графа наведено на Рисунку 1.7, пунктирні Стрілки це залежності по управлінню). Алгоритми, що працюють на PDG, намагаються знайти ізоморфні підграфи в парі PDG-графіків.

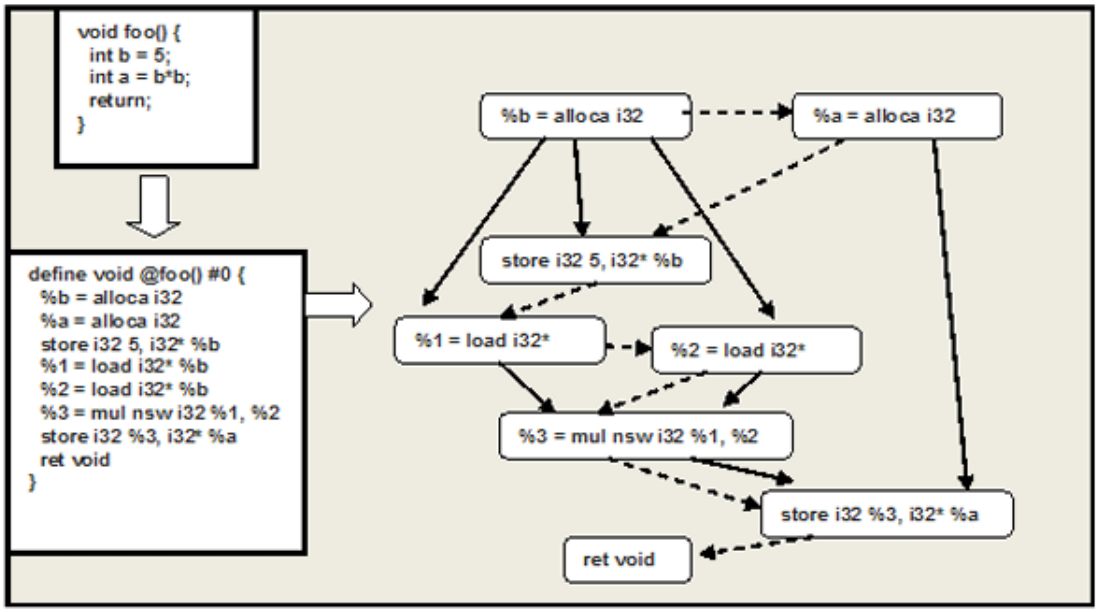

Рисунок 1.7 – Приклад PDG

Один з відомих інструментів виявлення дублікатів кодів на основі графів залежностей програми запропонували Komondoor і Horwitz [16], він знаходить ізоморфні підграфи PDG використовуючи зворотній програмний зріз. Krinke [17] у своїй роботі запропонував ітераційний підхід для того, щоб виявити подібних підграфів у PDG.

1.2.5 Гібридні підходи

Крім перерахованих вище підходів існують також методи виявлення клонів, які комбінують синтаксичні і семантичні характеристики. Як приклад, Leitao [18] пропонує підхід, який гібридно поєднує синтаксичні методи, котрі засновані на метриках абстрактних синтаксичних дерев, і семантичні методи з використанням графів викликів у поєднанні зі спеціалізованими функціями порівняння.

Подібними гібридними підходами є також наступні:

– у праці Фанореза та ін. [22] представлено метод, який об’єднує АСД для ідентифікації клонів коду та текстовий підхід для усунення помилкових спрацювань; – в роботі Еграувела та ін. [23] для виявлення клонів типу 1-3, запропоновано поєднання підходів на основі токенів з підходами на основі

тексту, використовуючи переваги кожного з них відповідно до типу. Детектор Oreo, представлений в роботі Сайні та ін. [24], дає змогу з високою точністю та відгуком виявляти клони тупі 1 та клони тупу 4. Проводиться пошук програмної інформації та програмних метрик, а також машинне навчання.

1.3 Постановка задачі

Метою даного дипломного проєкту є розробка інформаційної технології, котра надасть можливість виявляти дублікати програмного коду у студентських роботах. За допомогою розробленої системи користувачі зможуть виявляти дублювання кодів 1-3 типів, тим самим зменшувати різноманітні проблеми в вихідному коді системи, які виникають саме через наявність значної долі дублювання коду. Окрім, як вже говорилось, зменшення розміру кодової бази, зменшує затрати на її обслуговування, а виявлення порушень студентами правил написання програмних продуктів і своєчасне виправлення, підвищить рівень якості освіти.

Для досягнення поставленої мети сформульовані наступні задачі:

1) створення системи виявлення клонів 1-3 типів;
2) роботоспроможність методики незалежно від різних мов програмування;
3) проаналізувати результати та представити у вигляді звіту.

Для пошуку клонів 1-3 типів, із вище розглянутих методик, на мою думку, найкращим є підхід з використанням абстрактного синтаксичного дерева. Алгоритм аналізу де задіяний даний метод не потребує витрати значної кількості часу для реалізації перевірки, що є важливим при застосуванні для аналізу великої кількості робіт. Для можливості використання, не залежно від різних мов програмування, даний алгоритм потребує різноманітні словники токенів. Для того, щоб спростити реалізації цього завдання, можна використати готовий генератор токенів з назвою «tree-sitter».

Для виконання поставленої задачі з виявлення дублікатів кодів у програмних проектах є дуже важливим врахувати наступні аспекти при реалізації проекту:

- роботи можуть бути реалізовані на різних мовах програмування;
- при розробці програмного забезпечення може застосовуватись кілька типів змін;
- висока точність виявлення клонів коду для програм, де не великий об’єм оригінального коду.

2. АЛГОРИТМ ДОСЛІДЖЕНЬ

У даній роботі реалізований алгоритм виявлення клонів, який складається з наступних послідовних кроків (Рисунок 2.1):

- ✓ аналіз вхідного коду і подальше представлення його у вигляді абстрактного синтаксичного дерева;
- ✓ наступний крок - представлення кожного піддерева у вигляді цифрового відбитку;
- ✓ процес індексація цифрових відбитків;
- ✓ отримання статистичних результатів.

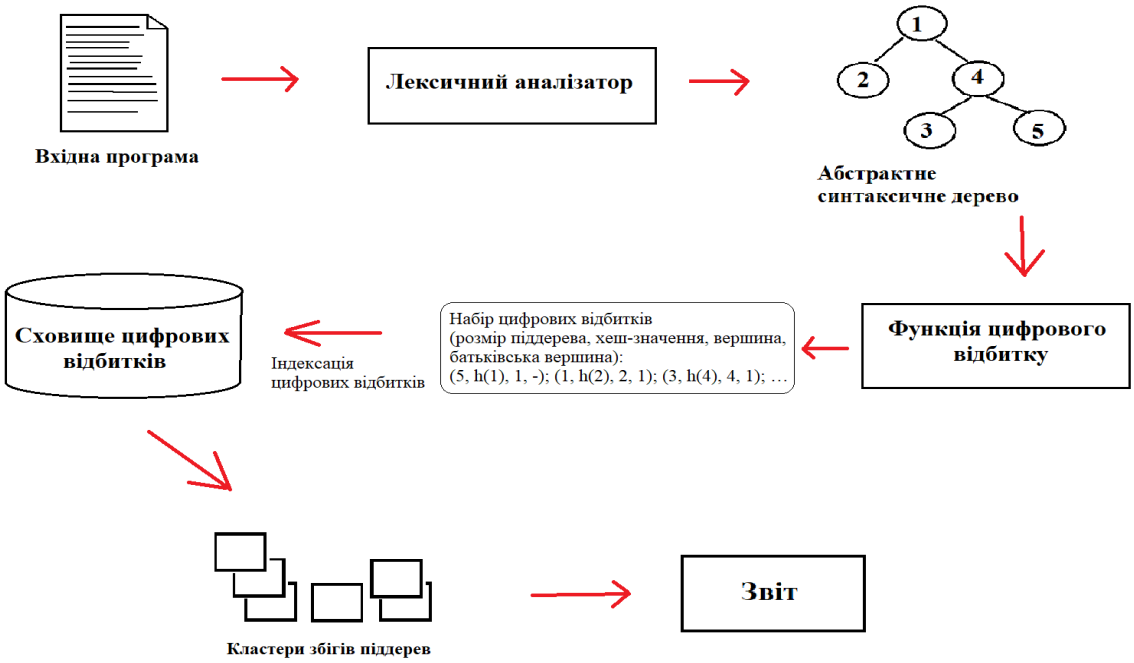

Рисунок 2.1 – Алгоритм виявлення клонів

Отже застосований статичний аналіз – це аналіз, при якому виконується дослідження програмного коду продукту без його реального виконання, а також він в собі містить методи аналізу потоку керування, абстрактної інтерпретації, а також аналізу потоку даних. Сюди ж відносяться методи, які використовують символьне виконання. При цьому статичними аналізаторами використовуються

наступні конструкції (проміжні представлення програми), у даній роботі, абстрактне синтаксичне дерево.

2.1 Лексичний аналіз вхідного коду

Передумовою побудови абстрактного синтаксичного дерева є лексичний аналіз вхідної програми.

Лексичний аналіз виконується шляхом перетворення вхідного рядку із послідовності символів в послідовність токенів, а токени в свою чергу поділяються на класи (Рисунок 2.2).

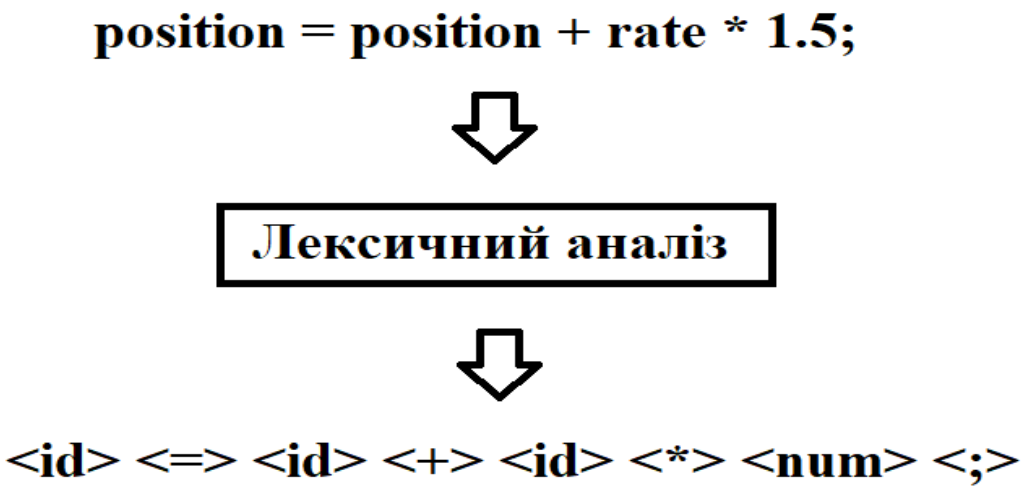

Рисунок 2.2 – Лексичний аналіз вихідного коду

У наведеному прикладі (Рисунок 2.2) після проведення лексичного аналізу було отримано шість класів токенів: <id> - ідентифікатор, <num> - число, <=> - оператор присвоєння, <+> - оператор додавання, <*> - оператор множення, <;> - «розділювач». Далі отримані класи будуть представлені у вигляді абстрактного синтаксичного дерева (Рисунок 2.3).

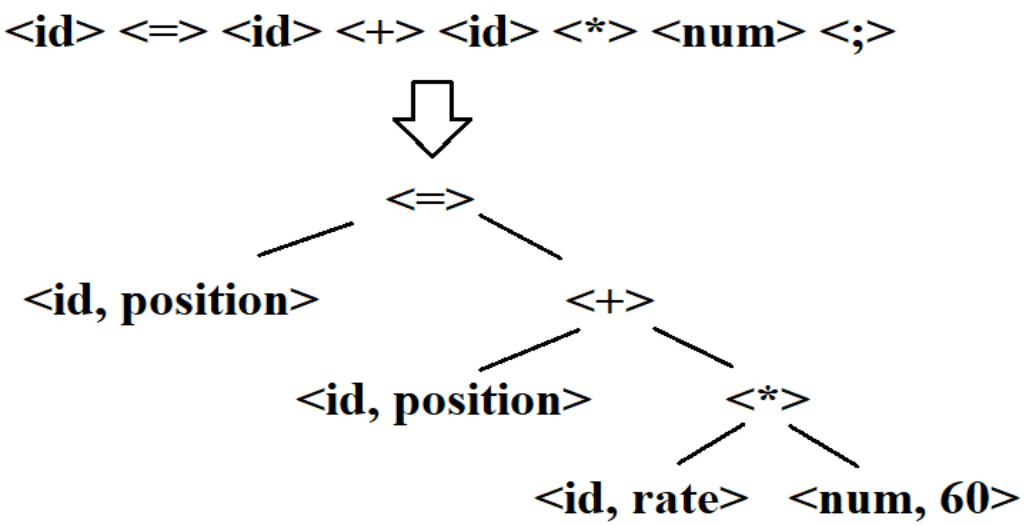

Рисунок 2.3 – Абстрактне синтаксичне дерево

Абстрактне синтаксичне дерево, утворене після лексичного аналізу, є структурним представленням вхідного програмного продукту. Воно очищене від елементів конкретного синтаксису шляхом порівняння лінійної послідовності створених токенів з формальною мовою. Вершинами синтаксичного дерева є оператори, до яких приєднуються їх аргументи, які у свою чергу можуть бути складними вершинами. У прикладі (Рисунок 2.3) абстрактне синтаксичне дерево побудоване з токенів, вершини містять додаткові атрибути – імена ідентифікаторів та значення чисел. У дерево не потрапив «розділювач», тому що він не має відношення безпосередньо до семантики даного фрагмента програми, а лише до конкретного синтаксису язика.

2.2 Цифровий відбиток піддерев

Кожне піддерево, після того як воно було отримано, абстрактного синтаксичного дерева обов'язково потрібно порівняти. У більшості синтаксичних дерев структура є досить складною і громіздкою, тому пряме

порівняння кожного піддерева, з якого складається синтаксичне дерево є досить складним, вимагає затрат часу та неефективним. Отже, для зменшення використовування обсягів пам’яті та підняття рівня ефективності порівняння, усі піддерева будуть мати вигляд хеша, що можно реалізувати за допомогою поліноміального кільцевого хеша. Ковзаюча хеш функція використовується для оптимізації підрахунку хешів. Далі отриманий набір хешів фільтрується за допомогою алгоритму Winnowing [19].

2.2.1 Ковзаюча хеш функція

Із застосуванням хеш-функції пов'язані два нюанси.

По-перше, алгоритм настільки хороший, наскільки хороша його хеш-функція. Якщо при використанні хеш-функції мають місце численні помилкові спрацьовування, то порівняння символів буде виконуватися занадто часто. У цьому випадку дуже важко вважати цей метод більш ефективним, ніж наївний алгоритм.

По-друге, кожен раз, коли підрядок проходить по тексту, обчислюється новий хеш, що вкрай неефективно, адже в цьому випадку продуктивність така ж.

Ковзаюча хеш функція – це функція, в якій вхідні дані хешуються тільки у рамках деякого вікна.

Основою виступає ковзаюче вікно, де відбувається хешування один за одним послідовних блоків, маючих розмір CHUNK_SIZE, починаючи з діапазону [0, CHUNK_SIZE - 1] до [LINES_COUNT-CHUNK_SIZE, LINES_COUNT - 1]. На Рисунку 2.4, представлено приклад даного процесу із заздалегідь визначеним CHUNK_SIZE встановленим в 2. Мінімальна довжина клону неминуче визначається вибором значення для CHUNK_SIZE.

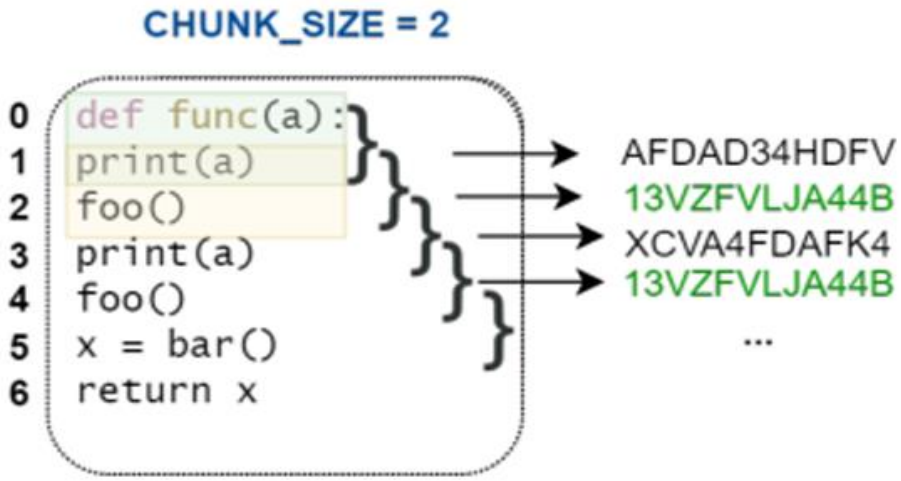

Рисунок 2.4 – Ковзаюче вікно хешування на основі CHUNK_SIZE

Щоб перерахувати значення хеша необхідно знати попереднє значення хешу, тобто значення вхідних даних, що залишилися за межами вікна, та значення даних, що потрапили у вікно. Тобто, якщо $x = h(a_1 a_2 \dots a_n)$ представляє собою хеш послідовності $a_1 a_2 \dots a_n$, то хеш $h(a_2 a_3 \dots a_n a_{n+1})$ для наступної послідовності $a_2 a_3 \dots a_n a_{n+1}$ може бути розрахований за допомогою функції $f(x, a_1, a_{n+1})$ [20].

Формула поліноміального кільцевого хеша:

$$h(a_1 a_2 \dots a_n) = a_1 x^{n-1} + a_2 x^{n-2} + \dots + a_n x^0, \quad (2.1)$$

де n – довжина послідовності, x – константа.

Формула ковзаючої хеш функції:

$$h = (h_p - a_p x^{n-1}) \cdot x + a_n \,, \quad (2.2)$$

де h_p – хеш попередньої послідовності, a_p – попередній символ, a_n – новий символ. Складність ковзаючої хеш функції лінійна - O(n), де n – довжина послідовності.

2.2.2 Алгоритм Winnowing та приклад його роботи

Winnowing використовує ковзаюче вікно його розмір w, яке проходить по списку хешів фільтруючи їх. Алгоритм на кожному кроці знаходить мінімальне значення хеша у вікні, якщо таких декілька, використовує крайнє праве та записує його до списку результатів, якщо даний хеш ще не був використаний. Коли вікно досягає кінця послідовності, записаний набір хешів приймається як цифрові відбитки документа.

Нижче наведено приклад роботи алгоритму Winnowing на короткому прикладі тексту (Рисунок 2.5).

Алгоритм Winnowing призводить до отримання цифрових відбитків з кількома корисними властивостями. Використовуючи k-грами та розмір вікна w, зберігатимуться такі властивості [19]:

- ✓ Нечутливість до шуму: збіги коротші за k не виявляються. Алгоритм хешує підрядки довжини k, що означає, що відповідні сегменти повинні мати принаймні довжину k, щоб бути хешовані.
- ✓ Рівномірний розподіл хешів: Winnowing гарантовано вибере принаймні один хеш для кожної довжини вікна. До того моменту, коли вікно зміститься на одну одиницю, попередній найнижчий хеш повинен вийти за рамки вікна, і, таким чином, алгоритм змушений вибрати новий найнижчий хеш з поточного вікна. Ця властивість корисна, оскільки гарантує, що жодна частина документа не буде пропущена з набору цифрових відбитків.
- ✓ Гарантовано виявлення довгих збігів: дві вищенаведені властивості гарантують, що збіги довжиною принаймні w + k - 1 завжди виявляються.

1) Вхідний текст
A do run run run, a do run run

2) Послідовність 5-грам отриманих з тексту
adoru dorun orunr runru unrun nrunr runru
unrun nruna runad unado nador adoru dorun
orunr runru unrun

3) Гіпотетична послідовність хешей 5-грам
77 74 42 17 98 50 17 98 8 88 67 39 77 74 42 17 98

4) Вікна хешей, при w = 4

(77, 74, 42, **17**)	(74, 42, 17, 98)
(42, 17, 98, 50)	(17, 98, 50, **17**)
(98, 50, 17, 98)	(50, 17, 98, **8**)
(17, 98, 8, 88)	(98, 8, 88, 67)
(8, 88, 67, 39)	(88, 67, **39**, 77)
(67, 39, 77, 74)	(39, 77, 74, 42)
(77, 74, 42, **17**)	(74, 42, 17, 98)

5) Цифрові відбитки обрані Winnowing
17 17 8 39 17

Рисунок 2.5 – Приклад використання алгоритму Winnowing

2.3 Індексація цифрових відбитків

У результаті дій пункту 2.2 кожне піддерево, що входить у склад абстрактного синтаксичного дерева, має вигляд цифрового відбитка. Цифровий відбиток піддерева x — є кортеж, до якого входить його розмір w(t), хеш-значення h(x), кореневий вузол і батьківський вузол, окрім того файл і відповідний номер рядка. У так званому сховищі цифрових відбитків піддерев існує подвійна індексація: кортежі сортуються у порядку від більшої до меншої ваги, а потім відбувається індексація за значенням хеша та батьківським вузлом. Цей процес реалізується за допомогою структури даних, що має

деревовидну форму B+. Використання цієї структури дає можливість ітеруватися по сховищу, отримуючи найбільш вагомі клони, окрім того, можна отримати всі відбитки дочірніх піддерев, що відносяться до даного вузла. Складність даної структури обчислюється для кожної операції наступним чином $O(\log\frac{t}{2n})$, де t — порядок дерева, або коефіцієнт розгалуження; n - кількість елементів дерева.

2.4 Вилучення кластерів зі збігами

Ітерація по сховищу, де знаходяться цифрові відбитки дозволяє отримати кластери, які є точними збігами піддерева. Кластери будуть вважатися однаковими, якщо мають однакову вагу та однакове значення хеша. Але, є мала ймовірність отримати хибнопозитивні результати. Щоб зменьшити кількості хибнопозитивних результатів необхідно збільшити довжину хеш-значень, при цьому значно зросте розмір сховища.

З метою вирішенням даної проблеми, яка ускладнює процес реалізації алгоритму, вводиться додаткова ітерація по фіксованій кількості дочірніх елементів. Розглянемо два піддерева, з вершинами x і y, їх відповідні дочірні елементи $x_1, \ldots, x_i$ і $y_1, \ldots, y_i$. Допустим x і y мають однакові значення і ваги, і хеша, тоді додатково порівнюються значення ваги та хеша саме дочірніх пар $(x_1, y_1), \ldots, (x_i, y_i)$. Даний підхід реалізовується за допомогою вказівника на батьківський вузол цифрового відбитка, котрий дозволяє, конкретно заданому вузлу, отримати цифрові відбитки саме його дочірніх елементів. Цей підхід не вимагає десеріалізації синтаксичних дерев.

3. ПРАКТИЧНА РЕАЛІЗАЦІЯ ПОСТАВЛЕНОЇ ЗАДАЧИ

Для того, щоб виявити нечіткі дублікати коду, запропонований алгоритм на основі абстрактного синтаксичного дерева. В ході розробки технології, яка дає змогу виявляти клони з використанням даного підходу, враховуються наступні важливі питання: аналіз вхідного програмного коду, побудова абстрактного синтаксичного дерева, наступний крок хешування піддерев, індексація цифрових відбитків та написання звіту про отриманні результати. Архітектура відповідної системи на Рисунках 3.1-3.2.

Рисунок 3.1 – Алгоритм виявлення клонів

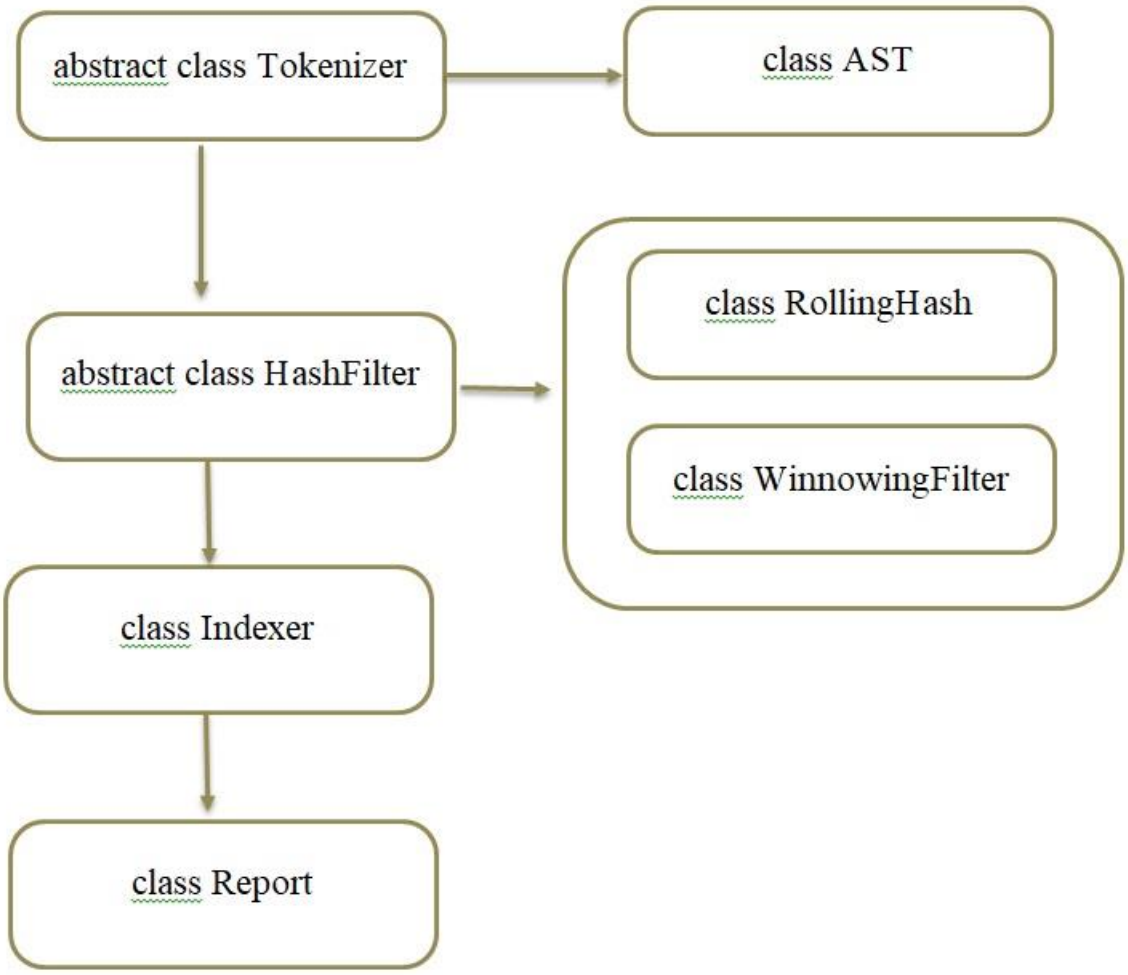

Рисунок 3.2 – Архітектура системи виявлення клонів

Завдяки модульній архітектурі розробленої системи, вона ,при необхідності, може бути доповнена новим функціоналом без великих затрат часу, включаючи інтеграцію підтримки різноманітних мов програмування. Дана побудована система була практично реалізована за допомогою мови програмування TypeScript.

3.1 Практична реалізація абстрактного синтаксичного дерева

Щоб побудувати абстрактне синтаксичне дерево була використана бібліотека «Tree-sitter». Ця бібліотека може створити конкретне синтаксичне дерево для вихідного програмного коду, вона підтримує більш ніж сорок мов програмування.

Реалізація абстрактного синтаксичного дерева з використанням бібліотеки «Tree-sitter»:

```
import { default as Parser, SyntaxNode } from "tree-sitter";

export class AST {
  public static supportedLanguages = ["c", "c-sharp", "java",
"javascript", "python"];

  public static IsSupportedLanguage(language: string): boolean {
    return this.supportedLanguages.includes(language);
  }

  public static RegisterLanguage(language: string): void {
     try {
       require("tree-sitter-" + language);
     }
     catch (error) {
        throw new Error("tree-sitter-${language} language not found.");
     }
     this.supportedLanguages.push(language);
   }

  public AST(text: string): string {
    const tree = this.parser.parse(text);
    return tree.rootNode.toString();
  }
}
```

Приклад абстрактного синтаксичного дерева на Рисунку 3.3.

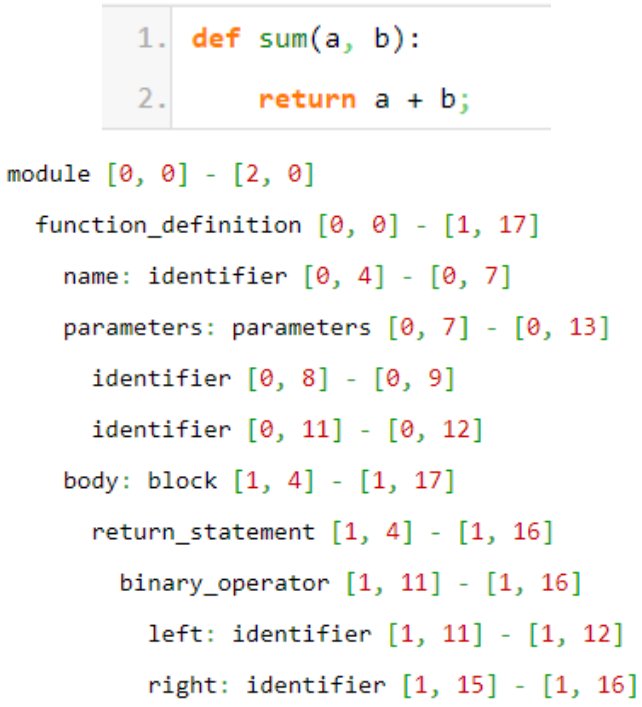

Рисунок 3.3 – Приклад побудови абстрактного синтаксичного дерева

Додавання до списку нових мов програмування можливо використовуючи команди yarn global add tree-sitter-[мова програмування].

3.2 Реалізація цифрових відбитків піддерев

Кожне піддерево абстрактного синтаксичного дерево може бути представленим у вигляді хеш значення використовуючи функції поліноміального кільцевого хеша. Щоб оптимізувати підрахунку хешів, була використана ковзаюча хеш функція. Після цього, отриманий набір хешів фільтрується. Щоб відфільтрувати хеші використовуємо алгоритм Winnowing.

Сам підрахунок хеша виконується з використанням модульної арифметики. У підрахунку застосовується операція множення, а модуль помножений сам на себе, не повинен бути більше допустимого числового значення мови TypeScript. Максимальне числове значення, що може бути використане в TypeScript є $2^{53} - 1$. Тоді $mod = 33554393$ – максимальне 26-бітне просте число.

При хешуванні використовується константа. Враховуючи що символи вхідного токену в більшості своїй мають значення менше за 128, доцільно обрати значення константи, таке що $127 \cdot base < mod$. Отже, для підрахунку поліноміального кільцевого хешу $base = 747287$.

Реалізація підрахунку поліноміального кільцевого хеша:

```
public hashToken(token: string): number {
  let hash = 0;
  for (let i = 0; i < token.length; i++) {
    hash = ((hash + token.charCodeAt(i)) * this.base) % this.mod;
  }
  return hash;
}
```

При підрахунку різних хешів використовують різні значення констант. Для ковзаючої хеш функції $base = 9999991$.

Реалізацію ковзаючої хеш функції:

```
  public NextHash(token: number): number {
    this.hash = (this.hash * this.base + this.hashes[index] *
this.maxBase + token)  % this.mod;
    this.hashes[index] = token;
    this.index = (this.index + 1) % this.k;
    return this.hash;
  }
```

Реалізацію алгоритму Winnowing:

```
 public async *fingerprints(tokens: string[]):
 AsyncIterableIterator<Fingerprint> {
     const hash = new RollingHash(this.k);
     let window: string[] = [];
     let filePos: number = -this.k;
     let bufferPos = 0;
     let    minPos = 0;
     const buffer: number[] = new
 Array(this.windowSize).fill(Number.MAX_SAFE_INTEGER);

     for await (const [hashedToken, token] of this.hashTokens(tokens)) {
       filePos++;
       window = window.slice(-this.k + 1);
```

```
        window.push(token);
        if (filePos < 0) {
          hash.nextHash(hashedToken);
          continue;
        }
        // minPos define far right hashing.
        bufferPos = (bufferPos + 1) % this.windowSize;
        buffer[bufferPos] = hash.nextHash(hashedToken);
        if (minPos === bufferPos) {
          // Scan buffer starting from bufferPos for the far right
minimal hashing.
          let i = (bufferPos + 1) % this.windowSize;
          for (; i !== bufferPos; i = (i + 1) % this.windowSize) {
            if (buffer[i] <= buffer[minPos]) {
              minPos = i;
            }
          }

          const offset = (minPos - bufferPos - this.windowSize) %
this.windowSize;
          const start = filePos + offset;
        }
        else {
          if (buffer[bufferPos] <= buffer[minPos]) {
            minPos = bufferPos;
            const start = filePos + ((minPos - bufferPos -
this.windowSize) % this.windowSize);
          }
         }
        yield {
           hash: buffer[minPos],
           start,
           stop: start + this.k - 1,
        };
      }
```

3.3 Практична реалізація індексування цифрових відбитків

Функція *cloneDetector()* фактично виконує порівняння файлів. Файли представляються у вигляді токенів за допомогою абстрактних синтаксичних дерев. А далі кожен з токен набуває вигляд цифрового відбитку. Для кожного цифрового відбутку проводиться перевірка на наявність еденьтчного хешу в дереві. Якщо, під час перевірки, хешу не було виявлено, то такий цифровий відбиток додається до дерева, у іншому випадку, цифровий відбиток додається до нашого звіту. В кінці функції, після повної перевірки, виконується формування звіту. Реалізацію функції *cloneDetector()*:

```
public async cloneDetector(files: File[]): Promise<Report> {
    const report = new Report();
    const tokenizedFiles = files.map(f =>
this.tokenizer.tokenizeFile(f));
    for (const file of tokenizedFiles) {
      let kgram = 0;
      for await (const { hash, start, stop  } of
hashFilter.fingerprints(file.Ast)) {
        // add kgram to file
        file.kgrams.push(new Range(start, stop));

        const part: Occurrence = {
          file,
          side: { index: kgram, start, stop, data, Region.merge(
          file.mapping[start],
          file.mapping[stop]
        ) }
        };

        // look if the index already contains the given hashing
        const matches = this.index.get(hash);

        if (matches) {
          report.addOccurrences(hash, part, ...matches);
          matches.push(part);
        } else {
          this.index.set(hash, [part]);
        }
        kgram += 1;
      }
    }
    report.create();
    return report;
  }
```

3.4 Результатів аналізу, їх представлення.

У результаті після проведення індексування всі цифрові відбитки, які зустрічаються в кількох файлах, вони збираються, а потім об’єднуються в один результуючий звіт.

Даній звіт складається з усіх пар файлів, які мають шо найменше один спільний цифровий відбиток, а також наступні показники:

- схожість - представляє ту частку спільних відбитків, які є між двома файлами;

- найдовший фрагмент – це найбільша довжина послідовних цифрових відбитків, що збігаються між двома файлами;
- повне перекриття - це ідентичність значень спільних цифрових відбитків.

Приклад звіту представлено в таблиці 3.1.

Таблиця 3.1 – Приклад звіту

Файл 1	**Файл 2**	**Схожість**	**Найдовший фрагмент**	**Повне перекриття**
Example1.c	Example2.c	0.94	63	218
Example2.c	Example3.c	0.89	56	208
Example1.c	Example3.c	0.88	56	204

Текс файлів Example1.c, Example2.c, Example3.c представлено у додатку Б.

3.5 Тестування системи

Результати реалізації алгоритму проведення перевірки системи на знаходження дублікатів різних типів представлені у таблиці 3.2. В таблиці зазначений відсоток розпізнавання різних типів дублікатів. Перевірка проводилвась з залученням 40 файлів вихідного коду з системи ejudge.

Таблиця 3.2 – Статистика виявлення дублікатів різних типів

Тип дублікату	**Відсоток розпізнавання**
Повна копія	100%
Зміни у коментарях, відступах та ідентифікаторах	100%
Зміна порядку рядків коду	92%
Зміна, додавання чи вилучення операторів	91%

Помітно, що даний алгоритм успішно виявляє клони 1-3 типів.

Реалізована система підтримує наступні мови програмування: C, C#, Java, JavaScript та Python. Приклад роботи системи для перерахованих мов програмування представлені у таблиці 3.3. Текст файлів представлені у додатку Б.

Таблиця 3.3 – Результати роботи системи для різних мов програмування

Файл 1	**Файл 2**	**Схожість**	**Найдовший фрагмент**	**Повне перекриття**
java_sample.java	java_sample-copy.java	0.98	9	38
c_sample.c	c_sample-copy.c	0.78	12	28
c-sharp_sample.cs	c-sharp_sample-copy.cs	0.89	29	90
python_sample.py	python_sample-copy.py	0.99	67	134
js_sample.js	js_sample-copy.js	1	36	72

Систему застосовувалась до 310 файлів вихідного програмного коду, що представляють собою вісім різноманітних завдань з програмування на мові C. Файли для аналізу були взяті з системи ejudge з курсу «Програмування». Два рішення завдання, що мають схожість менше 70%, вважаються незалежними роботами.

На Рисунку 3.4 показано розподіл подібності коду студентських посилань.

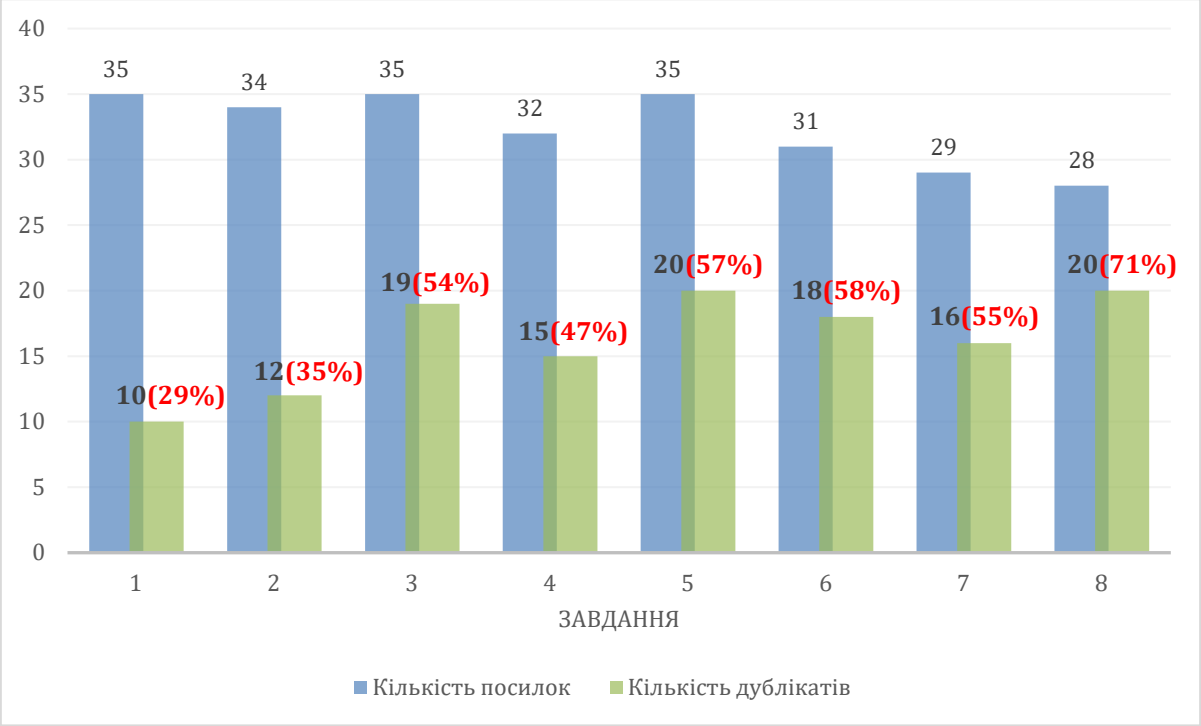

Рисунок 3.4 – Статистика дублікатів коду у роботах студентів

Перевірка показала, що майже половтна робіт є дублікатами. Серед них 382 пари мають подібність 98%.

Нижче на Рисунку 3.5 наведено приклад двох робіт, що мають коефіцієнт схожості 0,85. Дані роботи вважаються такими що містять дублікати.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/times.h>
#include <strings.h>
#include <ctype.h>

int ()
{
  int t1, t2, t;
  int timeinsec, nofattempts;
  char url[100], url1[80];
  strcpy(url, "url1");
  strcpy(url1, " url2");
  char word[15], *chk;
  chk = "word";
  FILE *fp;
  int syst = 1;
  fp = fopen("words", "r");
  t1 = time();
  while(chk != NULL)
  {
    chk = fgets(word, 15, fp);
    if (chk == NULL) exit(1);
    word [ strlen(word) - 1 ] = '\0';
    strcat(url, word);
    strcat(url, url1);
    nofattempts = nofattempts + 1;
```

```
#include <sys/times.h>
#include <sys/time.h>
#include <strings.h>
#include <ctype.h>

int ()
{
  int time1, time2, time_var;
  int timeinsec, nofattempts;
  char url[100], url1[80];
  strcpy(url, "u1");
  strcpy(url1, " u2");
  char word[15], *chk;
  chk = "word";
  FILE *fp;
  int syst = 1;
  fp = fopen("words", "r");
  time1 = time();
  while(chk != NULL)
  {
    chk = fgets(word, 15, fp);
    if (chk == NULL) exit(1);
    word [ strlen(word) - 1 ] = '\0';
    strcat(url, word);
    strcat(url, url1);
```

Рисунок 3.5 – Дублікат програмного коду

ВИСНОВКИ

У ході проведення роботи було створено, описано та реалізовано технологію виявлення нечітких, неявних, дублікатів вихідного програмного коду на основі абстрактного синтаксичного дерева. Один з найбільш часто зустрічаються варіантів їх застосування — статичний аналіз коду. Статичні аналізатори не виконують переданий їм код. Однак, незважаючи на це, їм потрібно розуміти структуру програм. В роботі розроблений інструмент, який знаходить структури в коді, котрі часто зустрічаються. Звіти такого інструменту допоможуть в рефакторингу, дозволять зменшити дублювання коду. Алгоритм може ефективно виявляти нечіткі дублікати коду використовуючи генерування хеш-значень, а потім порівнюючи їх . За допомогою даної системи можна аналізувати програми, в результаті виявляються спільні цифрові відбитки між файлами. Тестування системи показало що клони перших трьох типів розпізнаються з вірогідністю 91-100 %, враховуючі таку точність розпізнавання дублікатів клонів такий розроблений алгоритм може мати своє практичне застосування у визначення рівня схожості текстів програм студентів.

Результати дослідження вказують, що доволі часто в виникають програмні коди з клонами, які іноді складають більше 90% подібності. Оскільки при тестуванні 310 студентських робіт виявилось, що майже половини є дублікатами. Дана технологія має можливість підтримати кілька мов програмування, в ці роботі їх було використано 5, та вимоги щодо швидкості проведення аналізу, з метою використання її під час перевірки програмних кодів у рамках проведення навчального процесу.

СПИСОК ЛІТЕРАТУРИ

1. Аллен Е. шаблони помилок у Java.
2. Вашішт А. та ін. детальне дослідження клонування програмного коду. 2018. Том 9. С. 20-32.
3. Бейкер Б. С.про пошук дублювання і близького до дублювання у великих програмних системах // зворотна англ. - Робота. Конф. Пуття. IEEE, 1995. С. 86-95.
4. Рой К. К., Корді Дж.Р. Емпіричне дослідження функціональних клонів у програмному забезпеченні з відкритим кодом / / Proc. - Робота. Конф. Зворотний англ. WCRE. 2018. С. 81-90.
5. Маріані л., Мікуччі Д. Аудентес / / ACM Trans. Обчисливши. Освіта. ACM PUB27 Нью-Йорк, Нью-Йорк, США, 2012. Том 12, № 1.
6. Пірс Дж., Зіллес К. дослідження моделей плагіату учнів та кореляції з оцінками // Proc. Конф. Інтеграл. Технол. в комп'ютер. Наука. Освіта. ИТиССЕ. Асоціація обчислювальної техніки, 2017. С. 471-476.
7. Беллон С. та ін. порівняння та оцінка інструментів виявлення клонів / / IEEE Trans. Софтв. Англ. 2007. Том 33, № 9. С. 577-591.
8. Віславський Т. та ін. LICCA: інструмент виявлення міжмовних клонів // 25th IEEE Int. Конф. Софтв. Анал. Эвол. Реінжиніринг, SANER 2018-Proc. Інститут інженерів електротехніки та електроніки Inc., 2018. Тому 2018-Березень. Стр. 512-516.
9. Мансо А. та ін. виявлення плагіату в алгоритмах-тематичне дослідження з використанням Algorithmi. 2020.
10. Дюкасс с., Рігер м., Демейер с. незалежний від мови підхід до виявлення дубльованого коду // Конф. Софтв. Майнт. 1999. С. 109-118.
11. для J. J.-P. конференції С. 1993 року, 1993 рік не визначено. Визначення надмірності у вихідному коді за допомогою відбитків пальців // dl.acm.org.
12. Цзян л. та ін. ДЕКАРД: масштабоване та точне виявлення клонів коду на основі дерева / / Proc. - Інт. Конф. Софтв. Англ. 2007. С. 96-105.

13. Новак м, Джой м, Кермек Д.виявлення подібності вихідного коду та інструменти виявлення, що використовуються в академічних колах // ACM Trans. Обчисливши. Освіта. ACM PUB27 Нью-Йорк, Нью-Йорк, США, 2019. Т. 19, № 3.
14. Салман Хан м., Салман м. UNF Digital Commons запропонований підхід до моделювання теми для виявлення клонів коду. 2019.
15. Майран Дж., Леблан К., Мерло Е.М. експеримент з автоматичного виявлення клонів функцій у програмній системі з використанням метрик // Конф. Софтв. Обслуговування. IEEE, 1996. С. 244-253.
16. Комондур р., Хорвіц с. використання нарізки для виявлення дублювання у вихідному коді / / Лект. Примітки обчислюються. Науки (включаючи підсер. Лекція. Примітки Штучні. Інтелект. Лекція. Нотатки з біоінформатики). Спрінгер, Берлін, Гейдельберг, 2019. Т. 2126 LNCS. Стр. 40-56.
17. Кринке Дж. Ідентифікація аналогічного коду за допомогою графіків залежностей програм / / зворотна англ. - Робота. Конф. Вип. 2001. С. 301-309.
18. Лейтан А. М. виявлення надлишкового коду за допомогою R 2 D 2 // Програмне забезпечення. Квал. Дж. 2004 124. Спрінгер, 2004. Том 12, № 4. С. 361-382.
19. Шлеймер с., Вілкерсон Д. С., Айкен а. відсіювання: локальні алгоритми відбитків пальців. 2003.
20. Ся у. та ін. Ddelta: підхід до швидкого дельта-стиснення, заснований на дедуплікації // Perform. Оцінка. Північна Голландія, 2018. Том 79. С. 258-272.
21. Міллер г.а. магічне число сім, плюс-мінус Два // психологічний огляд. Тому 63. 1956. С. 81-97.
22. Фунаро Марко, Брага Даніеле, Кампі Алессандро, Гецці Карло. Гібридний підхід (синтаксичний та текстовий) до виявлення клонів. У матеріалах 4-го Міжнародного семінару з клонування програмного забезпечення, с.79-

80. ACM, 2010.

23. Агравал Акшат, Ядав Суміт Кумар. Гібридний підхід, заснований на токенах і тексті, для пошуку схожих сегментів коду. У 2013 році четверта Міжнародна конференція з обчислювальної техніки, комунікацій та мережевих технологій (ICCCNT), сторінки 1-4. IEEE, 2013.

24. Саїні Вайбхав, Фармахініфарахані Фаріма, Лу Ядонг, Бальді П'єр, Влопес Крістіна. Орео: виявлення клонів у зоні сутінків. У матеріалах 26-го спільного засідання ACM 2018 року з Європейської конференції з розробки програмного забезпечення

ДОДАТКИ

Додаток А. Програмний код

AST.ts

```
import { default as Parser, SyntaxNode } from "tree-sitter";

export class AST {
  public static supportedLanguages = ["cpp", "c-sharp", "java",
"javascript", "python"];

  public static IsSupportedLanguage(language: string): boolean {
    return this.supportedLanguages.includes(language);
  }

  public static RegisterLanguage(language: string): void {
     try {
       require("tree-sitter-" + language);
     }
     catch (error) {
        throw new Error("tree-sitter-${language} language not found.");
     }
    this.supportedLanguages.push(language);
   }

  public AST(text: string): string {
    const tree = this.parser.parse(text);
    return tree.rootNode.toString();
  }
}
```

RollingHash.ts

```
export class RollingHash {

  private readonly hashes: number[];
  private readonly maxBase: number;
  private index = 0;
  private hash = 0;

  readonly k: number = 23;
  readonly mod: number = 33554393;
  readonly base: number = 9999991;
```

```
  constructor() {
    this.k = k;
    this.maxBase = this.mod - this.modPow(this.base, this.k, this.mod);
    this.hashes = new Array(this.k).fill(0);
  }

  private ModularPow(base: number, exp: number, mod: number): number {
    let result = 1;
    base = base % mod;
    while (exp > 1) {
      if (exp % 2 == 1) {
        result = (result * base) % mod;
      }
      exp = exp >> 1;
      base = (base * base) % mod;
    }
    return result;
  }

  public NextHash(token: number): number {
    this.hash = (this.hash * this.base + this.hashes[index] *
this.maxBase + token)  % this.mod;
    this.hashes[index] = token;
    this.index = (this.index + 1) % this.k;
    return this.hash;
  }
}
```

Winnowing.ts

```
import { Fingerprint, HashFilter } from "./hashFilter";
import { RollingHash } from "./rollingHash";

export class WinnowingFilter extends HashFilter {
  private readonly k: number;
  private readonly windowSize: number;

  constructor(k: number /*default is 23*/, windowSize: number) {
    this.k = k;
    this.windowSize = windowSize;
  }

  public async *fingerprints(tokens: string[]):
AsyncIterableIterator<Fingerprint> {
    const hash = new RollingHash(this.k);
```

```
    let window: string[] = [];
    let filePos: number = -this.k;
    let bufferPos = 0;
    let minPos = 0;
    const buffer: number[] = new
Array(this.windowSize).fill(Number.MAX_SAFE_INTEGER);

    for await (const [hashedToken, token] of this.hashTokens(tokens)) {
      filePos++;
      window = window.slice(-this.k + 1);
      window.push(token);
      if (filePos < 0) {
        hash.nextHash(hashedToken);
        continue;
      }
      // minPos define far right hashing.
      bufferPos = (bufferPos + 1) % this.windowSize;
      buffer[bufferPos] = hash.nextHash(hashedToken);
      if (minPos === bufferPos) {
        // Scan buffer starting from bufferPos for the far right
minimal hashing.
        let i = (bufferPos + 1) % this.windowSize;
        for (; i !== bufferPos; i = (i + 1) % this.windowSize) {
          if (buffer[i] <= buffer[minPos]) {
            minPos = i;
          }
        }

        const offset = (minPos - bufferPos - this.windowSize) %
this.windowSize;
        const start = filePos + offset;
      }
      else {
        if (buffer[bufferPos] <= buffer[minPos]) {
          minPos = bufferPos;
          const start = filePos + ((minPos - bufferPos -
this.windowSize) % this.windowSize);
        }
       }
      yield {
         hash: buffer[minPos],
         start,
         stop: start + this.k - 1,
      };
    }
```

```
  }
}

Utils.ts
ranging(): Report {
    let isFound = false;
    let direction;
    if (this.enabled) {
      let acceptable: SnappingType[] | undefined;
      const activeProviders =
this.providersContainer.getActiveProviders();
      for (const provider of activeProviders) {
        if (acceptable && !acceptable.includes(provider.type)) {
          continue;
        }
        const result = provider(finalPoint, direction);

        provider.drawTooltip(result);
        if (result.found) {
          result.snappingType = provider.type;
         direction = direction || result.direction;

          if (!result.canBeModifiedBy) {
            return result;
          } else {
            finalPoint = result.point;
            acceptable = result.canBeModifiedBy;
            isFound = true;
            snappingType = provider.type;
          }
        }
      }
    }
    return { point: finalPoint, found: isFound, snappingType:
snappingType };
  }

async restoreSnapshot() {
    await this._viewerService.ready();
    const sortedCommandsList = await
this._snapshotProvider.getCommandsForScenario();
    this.originalCommandSequence = sortedCommandsList;
    this.savedCommandSequence =
sortedCommandsList.filter(command => command.isSaved);
    for (const commandSnapshot of sortedCommandsList) {
```

```
      this.executeSnapshotCommand(commandSnapshot)
    }

    const finalVirtualEntities =
this._drawingManager.virtualModel.entities;
    for (const entity of finalVirtualEntities) {
      const command: Command<any> | undefined =
FactoriesScope.drawingCommandFactory.build(
        entity.virtualCurve.getCreateCommandType(),
        entity.virtualCurve.generateToolConfig(),
        CommandScope.LOCAL
      );

      if (command) {
        command.commandLinkId = entity.commandLinkId;
        command.setupDependencies(this._factory);
        command.redo();
      }
    }
    this._onSnapshotRestored.next(true);
  }
```

Index.ts

```
import { Tokenizer } from "../tokenizer";
import { Report, Occurrence } from "./report";
import { WinnowwingFilter } from "../WinnowwingFilter";
import { Options } from "../options";
import { HashFilter } from "../hashFilter";

export interface DolosOptions {
  k: number;
  windowSize: number;
  language: string;
}
export class DefaultOptions implements Options{
  public static defaultLanguage = "c";
  public static defaultKgramLength = 23;
  public static defaultKgramsInWindow = 17;
}

export class Indexer {
  private readonly tokenizer: Tokenizer;
  private readonly hashFilter: HashFilter;
  private readonly index: Map<Hash, Array<Occurrence>> = new Map();
```

```
  constructor(options: Options = new Options()) {
    this.hashFilter = new WinnowFilter(options.k, options.windowSize);
  }

private createQuadTrees() {
    this.logger.info(`Creating new Quadtrees`);
    const quadTreeBounds = this.getViewerExtentsBox();
    if (!quadTreeBounds) {
      return;
    }

    this._backgroundQuadTree = new Quadtree(quadTreeBounds, 10, 6);
    this._foregroundQuadTree = new Quadtree(quadTreeBounds, 10, 6);
    this._ready.next();
  }

public async cloneDetector(files: File[]){
    const tokenizedFiles = files.map(f => this.tokenizeFile(f));
    const report = new Report(this.options);
    for (const file of tokenizedFiles) {
      let kgram = 0;
      for (const { hash, start, stop} of
hashFilter.fingerprints(file.Ast)) {
        file.kgrams.push(new Range(start, stop));
        const part: Occurrence = {
          fileside: { index: kgram, start, stop, data,
        Region.merge(file.mpp [start],file.mpp [stop])}
        };
        // look if the index already contains the given hashing
        const matches = this.index.get(hash);
        if (matches) {
          report.addOccurrences(hash, part, ...matches);
          matches.push(part);
        } else {
          this.index.set(hash, [part]);
        }
        kgram += 1;
      }
    }
    report.create();
    return report;
  }
}
```

Додаток Б. Лістинг текстів файлів

Example1.c

```c
#include <stdio.h>
#include <malloc.h>
#include <conio.h>

int binarysearch(int a, int mass[], int n);
void InsertionSort(int n, int mass[]);

int main()
{
    int N, a;
    printf("Input N: ");
    scanf_s("%d", &N);
    int* mass;
    mass = (int *)malloc(N * sizeof(int));
    printf("Input the array elements:\n");
    for (int i = 0; i < N; i++)
        scanf_s("%d", &mass[i]);
    InsertionSort(N, mass);
    printf("Sorted array:\n");
    for (int i = 0; i < N; i++)
        printf("%d ", mass[i]);
    printf("\n");
    printf("Input variable 'a' for search: ");
    scanf_s("%d", &a);
    int k;
    k = binarysearch(a, mass, N);
    if (k != -1)
    {
        printf("The index of the element is %d\n", k);
    }
    else
        printf("The element isn't found!\n");
    free(mass);
    _getch();
    return 0;
}

int binarysearch(int a, int mass[], int n)
{
    int low, high, middle;
    low = 0;
    high = n - 1;
    while (low <= high)
    {
        middle = (low + high) / 2;
        if (a < mass[middle])
            high = middle - 1;
        else if (a > mass[middle])
```

```
            low = middle + 1;
        else
            return middle;
    }
    return -1;
}

void InsertionSort(int n, int mass[])
{
    int newElement, location;

    for (int i = 1; i < n; i++)
    {
        newElement = mass[i];
        location = i - 1;
        while (location >= 0 && mass[location] > newElement)
        {
            mass[location + 1] = mass[location];
            location = location - 1;
        }
        mass[location + 1] = newElement;
    }
}
```

Example2.c

```
#include <stdio.h>
#include <malloc.h>
#include <conio.h>

int binarysearch(int a, int mass[], int n);
void InsertionSort(int n, int mass[]);

int main()
{
    int N, a;
    printf("Input N: ");
    scanf_s("%d", &N);
    int* mass;
    mass = (int *)malloc(N * sizeof(int));
    printf("Input the array elements:\n");
    for (int i = 0; i < N; i++)
        scanf_s("%d", &mass[i]);
    InsertionSort(N, mass);
    printf("Sorted array:\n");
    for (int i = 0; i < N; i++)
        printf("%d ", mass[i]);
    printf("\n");
    printf("Input variable 'a' for search: ");
    scanf_s("%d", &a);
    int k;
    k = binarysearch(a, mass, N);
    if (k != -1)
```

```
    {
        printf("The index of the element is %d\n", k);
    }
    else
        printf("The element isn't found!\n");
    free(mass);
    _getch();
    return 0;
}

//add some minor changes
int binarysearch(int a, int mass[], int n)
{
    int low = 0, high = n - 1, middle;
    while (low <= high)
    {
        middle = (low + high) * 0.5;
        if (a < mass[middle])
            high = middle - 1;
        else if (a > mass[middle])
            low = middle + 1;
        else
            return middle;
    }
    return -1;
}

void InsertionSort(int n, int mass[])
{
    int newElement, location;

    for (int i = 1; i < n; i++)
    {
        newElement = mass[i];
        location = i - 1;
        while (location >= 0 && mass[location] > newElement)
        {
            mass[location + 1] = mass[location];
            location = location - 1;
        }
        mass[location + 1] = newElement;
    }
}
```

Example3.c

```
#include <stdio.h>
#include <malloc.h>
#include <conio.h>

int binarysearch(int a, int mass[], int n);
```

```c
void InsertionSort(int n, int mass[]);

int main()
{
    int N, a;
    printf("Input N: ");
    scanf_s("%d", &N);
    int* mass;
    mass = (int *)malloc(N * sizeof(int));
    printf("Input the array elements:\n");

    for (int i = 0; i < N; i++)
        scanf_s("%d", &mass[i]);
    InsertionSort(N, mass);

    printf("Sorted array:\n");
    for (int i = 0; i < N; i++)
        printf("%d ", mass[i]);
    printf("\n");
    printf("Input variable 'a' for search: ");
    scanf_s("%d", &a);
    int k;
    k = binarysearch(a, mass, N);

    if (k != -1)
        printf("The index of the element is %d\n", k);

    else
        printf("The element isn't found!\n");

    free(mass);
    _getch();
    return 0;
}

int binarysearch(int a, int mass[], int n)
{
    int low = 0, high = n - 1, middle;
    while (low <= high)
    {
        middle = (low + high) / 2;
        if (a < mass[middle])
            high = middle - 1;
        else if (a > mass[middle])
            low = middle + 1;
        else
            return middle;
    }
    return -1;
}
```

```
void InsertionSort(int n, int mass[])
{
    int newElement, location;
       int i = 1;
    while(i < n)
    {
        newElement = mass[i];
        location = i - 1;
        while (location >= 0 && mass[location] > newElement)
        {
            mass[location + 1] = mass[location];
            location = location - 1;
        }
        mass[location + 1] = newElement;
             i++;
    }
}
```

java_sample.java

```
import java.util.HashMap;

public class Main {
  public static void main(String[] args) {
    // Create a HashMap object called capitalCities
    HashMap<String,   String>   capitalCities   =   new   HashMap<String,
String>();

    // Add keys and values (Country, City)
    capitalCities.put("England", "London");
    capitalCities.put("Germany", "Berlin");
    capitalCities.put("Norway", "Oslo");
    capitalCities.put("USA", "Washington DC");
    System.out.println(capitalCities);
  }
}
```

java_sample-copy.java

```
import java.util.HashMap;

public class Main {
  public static void main(String[] args) {
    // Create a HashMap object called capitalCities
    HashMap<String,   String>   capitalCities   =   new   HashMap<String,
String>();

    // Add keys and values (Country, City)
    capitalCities.put("England", "London");
    capitalCities.put("Germany", "Berlin");
    capitalCities.put("Norway", "Oslo");
    capitalCities.put("USA", "Washington DC");
```

```
    System.out.println(capitalCities);
  }
}
```

c_sample.c

```
#include<stdio.h>
#include<conio.h>

int fact(int f) {
  if (f==0 || f==1) {
    printf("Calculated Factorial");
    return 1;
  }
  return f * fact(f - 1);
}

int main(void) {
  int f = 12;
  printf("The factorial of %d is %d \n", f, fact(f));
  getch();
  return 0;
}
```

c_sample-copy.c

```
#include<stdio.h>
#include<conio.h>

long factorial(long f) {
  if (f==0 || f==1) {
    printf("Calculated Factorial");
    return 1;
  }
  return f * fact(f - 1);
}

int main(void) {
  long f = 12;
  printf("The factorial of %d is %d \n", f, fact(f));
  getch();
  return 0;
}
```

c-sharp_sample.cs

```
using System;
class bubblesort
{
      static void Main(string[] args)
      {
```

```
            int[] a = { 30, 20, 50, 40, 10 };
            int t;
            Console.WriteLine("The Array is : ");
            for (int i = 0; i < a.Length; i++)
            {
                  Console.WriteLine(a[i]);
            }
            for (int j = 0; j <= a.Length - 2; j++)
            {
                  for (int i = 0; i <= a.Length - 2; i++)
                  {
                        if (a[i] > a[i + 1])
                        {
                              t = a[i + 1];
                              a[i + 1] = a[i];
                              a[i] = t;
                        }
                  }
            }
            Console.WriteLine("The Sorted Array :");
            foreach (int aray in a)
                  Console.Write(aray + " ");
            Console.ReadLine();
      }
}
```

c-sharp_sample-copy.cs

```
using System;
class bubblesort
{
      static void Main(string[] args)
      {
            int[] a = { 30, 20, 50, 40, 10 };
            int t;
            for (int i = 0; i < a.Length; i++)
            {
                  Console.WriteLine(a[i]);
            }
            for (int j = 0; j <= a.Length - 2; j++)
            {
                  for (int i = 0; i <= a.Length - 2; i++)
                  {
                        if (a[i] > a[i + 1])
                        {
                              t = a[i + 1];
                              a[i + 1] = a[i];
                              a[i] = t;
                        }
                  }
            }
```

```
            foreach (int aray in a)
                  Console.Write(aray + " ");
            Console.ReadLine();
      }
}
```

python_sample.py

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mylist = [
  { "_id": 1, "name": "John", "address": "Highway 37"},
  { "_id": 2, "name": "Peter", "address": "Lowstreet 27"},
  { "_id": 3, "name": "Amy", "address": "Apple st 652"},
  { "_id": 4, "name": "Hannah", "address": "Mountain 21"},
  { "_id": 5, "name": "Michael", "address": "Valley 345"},
  { "_id": 6, "name": "Sandy", "address": "Ocean blvd 2"},
  { "_id": 7, "name": "Betty", "address": "Green Grass 1"},
  { "_id": 8, "name": "Richard", "address": "Sky st 331"},
  { "_id": 9, "name": "Susan", "address": "One way 98"},
  { "_id": 10, "name": "Vicky", "address": "Yellow Garden 2"},
  { "_id": 11, "name": "Ben", "address": "Park Lane 38"},
  { "_id": 12, "name": "William", "address": "Central st 954"},
  { "_id": 13, "name": "Chuck", "address": "Main Road 989"},
  { "_id": 14, "name": "Viola", "address": "Sideway 1633"}
]

x = mycol.insert_many(mylist)

#print a list of the _id values of the inserted documents:
print(x.inserted_ids)
```

python_sample-copy.py

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mylist = [
  { "_id": 1, "name": "John", "address": "Highway 37"},
  { "_id": 2, "name": "Peter", "address": "Lowstreet 27"},
  { "_id": 3, "name": "Amy", "address": "Apple st 652"},
  { "_id": 4, "name": "Hannah", "address": "Mountain 21"},
  { "_id": 5, "name": "Sandy", "address": "Ocean blvd 2"},
  { "_id": 6, "name": "Michael", "address": "Valley 345"},
```

```
  { "_id": 7, "name": "Betty", "address": "Green Grass 1"},
  { "_id": 8, "name": "Richard", "address": "Sky st 331"},
  { "_id": 9, "name": "Susan", "address": "One way 98"},
  { "_id": 10, "name": "Vicky", "address": "Yellow Garden 2"},
  { "_id": 11, "name": "Ben", "address": "Park Lane 38"},
  { "_id": 12, "name": "William", "address": "Central st 954"},
  { "_id": 13, "name": "Chuck", "address": "Main Road 989"},
  { "_id": 14, "name": "Viola", "address": "Sideway 1633"}
]

x = mycol.insert_many(mylist)
print(x.inserted_ids)
```

js_sample.js

```
function compareName(a, b) {

    // converting to uppercase to have case-insensitive comparison
    const name1 = a.name.toUpperCase();
    const name2 = b.name.toUpperCase();

    let comparison = 0;

    if (name1 > name2) {
        comparison = 1;
    } else if (name1 < name2) {
        comparison = -1;
    }
    return comparison;
}

const students = [{name: 'Sara', age:24},{name: 'John', age:24}, {name:
'Jack', age:25}];

console.log(students.sort(compareName));
```

js_sample-copy.js

```
function compare(a, b) {

    // converting to uppercase to have case-insensitive comparison
    const name1 = a.name.toUpperCase();
    const name2 = b.name.toUpperCase();

    let comparison = 0;

    if (name1 > name2) {
        comparison = 1;
    } else if (name1 < name2) {
        comparison = -1;
```

```
    }
    return comparison;
}

const students = [{name: 'Sara', age:24},{name: 'John', age:24}, {name:
'Jack', age:25}];

console.log(students.sort(compareName));
```

Додаток В. Приклад програм з дублікатами.

Програма на рисунку 3.4 зліва:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/times.h>
#include <strings.h>
#include <ctype.h>

int ()
{
  int t1, t2, t;
  int timeinsec, nofattempts;
  char url[100], url1[80];
  strcpy(url, "url1");
  strcpy(url1, "url2");
  char word[15], *chk;
  chk = "word";
  FILE *fp;
  int syst = 1;
  fp = fopen("words", "r");
  t1 = time();
  while(chk != NULL)
  {
    chk = fgets(word, 15, fp);
    if (chk == NULL) exit(1);
    word [ strlen(word) - 1 ] = '\0';
    strcat(url, word);
    strcat(url, url1);
    nofattempts = nofattempts + 1;
    printf("\n %s %d\n",word,nofattempts);
    if (strlen(word) == 3)
    syst = system(url);
    if (syst == 0)
    {
       t2 = time();
       t = t2 - t1;
       timeinsec = t/1000000000;
       printf("\n !!! here's the passowrd:- %s",word);
       printf("\n Total .of atempts: %d\n",nofattempts);
       printf("\n The total time_var taken: %d seconds", timeinsec);
       exit(1);
    }
    strcpy(url, "");
    strcpy(url, "wget --http-user= --http-passwd=");
  }
}
```

Програма на рисунку 3.4 праворуч:

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <sys/times.h>
#include <sys/time.h>
#include <strings.h>
#include <ctype.h>

int ()
{
  int time1, time2, time_var;
  int timeinsec, nofattempts;
  char url[100], url1[80];
  strcpy(url, "u1");
  strcpy(url1, "u2");
  char word[15], *chk;
  chk = "word";
  FILE *fp;
  int syst = 1;
  fp = fopen("words", "r");
  time1 = time();
  while(chk != NULL)
  {
    chk = fgets(word, 15, fp);
    if (chk == NULL) exit(1);
    word [ strlen(word) - 1 ] = '\0';
    strcat(url, word);
    strcat(url, url1);
    if (strlen(word) == 3)
    {
      syst = system(url);
      nofattempts = nofattempts + 1;
      printf("\n %s %d\n",word,nofattempts);
    }
    if (syst == 0)
    {
       time2 = time();
       time_var = time2 - time1;
       timeinsec = time_var/1000000000;
       printf("\n The Password is: %s",word);
       printf("\n  of Attempts: %d\n",nofattempts);
       printf("\n Time Taken: %d seconds\n", timeinsec);
       exit(1);
    }
    strcpy(url, "");
    strcpy(url, "wget --http-user= --http-passwd=");
  }
}
```