

Міністерство освіти і науки України  
Департамент науки і освіти Харківської облдержадміністрації  
Харківське територіальне відділення МАН України

Відділення: комп'ютерних наук  
Секція: комп'ютерні системи та мережі

## МЕТОД ПОШУКУ ТА УСУНЕННЯ ПОВТОРЮВАНИХ ЧАСТИН У ПОЧАТКОВОМУ КОДІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Роботу виконав:  
Човпан Ігор Сергійович,  
учень 11 класу Харківського  
Навчально-виховного комплексу  
№45 «Академічна гімназія»  
Харківської міської ради  
Харківської області

Науковий керівник:  
Руккас Кирило Маркович,  
професор кафедри теоретичної та  
прикладної інформатики  
механіко-математичного  
факультету Харківського  
національного університету  
ім. В.Н. Каразіна, доктор  
технічних наук, доцент

## Тези

...

## ЗМІСТ

ВСТУП . . . . .	4
РОЗДІЛ 1. Характеристика існуючих методів знаходження повторюваних частин у кодї . . . . .	5
1.1. Основних види повторюваних частин . . . . .	5
1.2. Основні методи пошуку повторюваних частин . . . . .	6
1.2.1. Пошук збігу рядків початкового коду . . . . .	7
1.2.2. Використання токенів . . . . .	8
1.2.3. Метод порівняння функцій . . . . .	8
1.2.4. Застосування графа програмних залежностей . . . . .	8
1.2.5. Метод порівняння дерев . . . . .	9
РОЗДІЛ 2. Новий алгоритм пошуку повторюваних частин . . . . .	11
2.1. Парсинг коду . . . . .	11
2.2. Визначення частин коду, які підлягають порівнянню . . . . .	11
2.3. Знаходження повторюваних частин . . . . .	13
2.4. Перетворення на послідовність фрагментів у кодї . . . . .	16
2.5. Використання алгоритму для видалення клонів . . . . .	17
РОЗДІЛ 3. Порівняння з іншими методами . . . . .	19
РОЗДІЛ 4. Висновки . . . . .	22

**ВСТУП**

Ваш ВСТУП

## РОЗДІЛ 1.

### ХАРАКТЕРИСТИКА ІСНУЮЧИХ МЕТОДІВ ЗНАХОДЖЕННЯ ПОВТОРЮВАНИХ ЧАСТИН У КОДІ

Щоб проаналізувати методи знаходження повторюваних частин, треба визначити, що таке повторювана частина.

Вважатимемо дублікатом фрагмент, який є ідентичним з іншим фрагментом коду.

Тоді повторювана частина коду – фрагмент, в якого є дублікати.

Визначемо основні види повторюваних частин.

#### 1.1. Основних види повторюваних частин

Як зазначено у роботах [8], [5] та [2], виділяється 4 головних типи повторюваних частин.

- I тип – повна копія без модифікацій, окрім пробілів та коментарів;

<pre>double xx = Math.cos(angle); double yy = Math.sin(angle); xx*=2; yy*=2; if (xx&gt;PI)     xx = 2*PI-xx; if (yy&gt;PI)     yy = 2*PI-yy;</pre>	<pre>double xx = Math.cos(angle); // of course using math package!! double yy = Math.sin(angle); xx *= 2; yy *= 2; if ( xx &gt; PI) //That is VERY important statement!     xx = 2 * PI - xx; if (yy &gt; PI)     yy = 2 * PI - yy;</pre>
--	---

Рис. 1.1. Приклад копії I типу на мові Java

- II тип – синтаксично однакова копія, змінюються лише назви змінних, назв функцій, тощо;
- III тип – копія з подальшими змінами; доданими, зміненими або видаленими інструкціями;

```

void func(double angle) {
    double xx = Math.cos(angle);
    double yy = Math.sin(angle);
    xx*=2;
    yy*=2;
    if (xx>PI)
        xx = 2*PI-xx;
    if (yy>PI)
        yy = 2*PI-yy;
    write(xx);
}

void veryImportantFunc(double ang){
    double aa = Math.cos(ang);
    double bb = Math.sin(ang);
    aa*=2;
    bb*=2;
    if (xx>PI_CONST)
        aa = 2*PI_CONST-aa;
    if (yy>PI_CONST)
        bb = 2*PI_CONST-bb;
    writeToFile(aa);
}

```

Рис. 1.2. Приклад копії II типу

```

void func(double angle) {
    double xx = Math.cos(angle);
    double yy = Math.sin(angle);
    double PI = Math.acos(-1);
    xx*=2;
    yy*=2;
    if (xx>PI)
        xx = 2*PI-xx;
    if (yy>PI)
        yy = 2*PI-yy;
    write(xx);
}

void doCalc(double ang){
    double bb = Math.sin(ang);
    double aa = Math.cos(ang);
    print("before"+aa);
    aa*=2;
    if (xx>PI_CONST)
        aa = 2*PI_CONST-aa;
    bb*=2;
    if (yy>PI_CONST)
        bb = 2*PI_CONST-bb;
    print(aa);
}

```

Рис. 1.3. Приклад копії III типу

- IV тип – частина, що робить ідентичні обчислювання, але синтаксично імплементована інакше.

## 1.2. Основні методи пошуку повторюваних частин

Існує багато прийомів, що використовуються для пошуку повторюваних частин у початковому коді програмного забезпечення.

Перелічим основні методи пошуку:

- пошук збігу рядків початкового коду;
- використання токенів;
- метод порівняння функцій;

```

int fibonacci(int n) {
    int sum1=0, sum2=1;
    for (int i=2; i<=n; i++){
        int sum3 = sum1+sum2;
        sum1 = sum2;
        sum2 = sum3;
    }
    return sum2;
}

int[] mem = new int[...];
int fib(int n) {
    if (mem[n]!=0)
        return mem[n];
    if (n<2)
        return 1;
    mem[n] = fib(n-1)+fib(n-2);
    return mem[n];
}

```

Рис. 1.4. Приклад копії IV типу

- застосування графа програмних залежностей;
- метод порівняння дерев.

Далі визначимо усі переваги і недоліки кожного з методів.

### 1.2.1 Пошук збігу рядків початкового коду

Обчислюється ступінь схожості для кожної пари рядків за допомогою відстані Левенштейна. Емпірично встановлюється мінімальна величина, за якої вважається, що 2 рядки є копіями одна одну.

Переваги цього методу:

- добре знаходить копії I типу;
- невеликий час виконання порівняно з іншими методами;
- підтримка будь-якої мови програмування.

Недоліки методу:

- велика кількість хибнонегативних результатів;
- нестійкість до різних ”шумів”: коментарів, змінених назв функцій або змінних, тобто неможливість знайти дублікати II та III типу.
- Не враховуються особливості мови програмування.

Прикладом використання є програма PMD.

### 1.2.2 Використання токенів

Початковий код розбивається на токени, при пошуці порівнюються послідовності токенів. Головною перевагою цього методу є стійкість до переформатування початкового коду, зміни назв змінних. Недоліком є те, що токенізатори враховують тільки базові особливості мови програмування, тому багато послідовностей, які вважаються копіями, насправді самі по собі не мають сенсу. [10] Прикладом використання такого методу є програма CCFnderX.

```
return x;          return a;          void,myFunc,(,int,b,){,b,++,;
void func(int y) { void myFunc(int b) {
    y++;           b++;
```

Рис. 1.5. Частини коду, що вважатимуться копіями; приклад розбиття коду на токени

### 1.2.3 Метод порівняння функцій

За допомогою парсера мови програмування знаходять усі функції у початковому коді. Далі усі ці функції порівнюються між собою або за допомогою спеціально обраної «поганої» геш-функції, або за допомогою обчислення коефіцієнту схожості (наприклад, коеф. Жаккара). Метод гарно знаходить збіги між різними функціями, розпізнаються копії I-III типу, проте він не може знайти повторювані частини всередині функції. Прикладом використання є [13].

### 1.2.4 Застосування графа програмних залежностей

Згідно з [7], граф програмних залежностей (далі просто граф) – представлення програми як графа, у якому кожна вершина - інструкція у програмі, а також зв'язані з цією інструкцією оператори та операнди; ребрами у такому графі є дані, від яких залежить виконання цієї інструкції та умови, за яких ця інструкція виконається. Дві частини програми вважаються ідентичними, якщо їх графи ізоморфні. Головною перевагою є те, що цей граф не залежить від переставлення інструкцій, зміни назв функцій, тощо; не залежить від аспектів реалізації. Недоліки методу:

- Дуже довгий час роботи, оскільки завдання пошуку ізоморфних підграфів є NP-повною, і може бути вирішена за поліноміальний час тільки для пла-



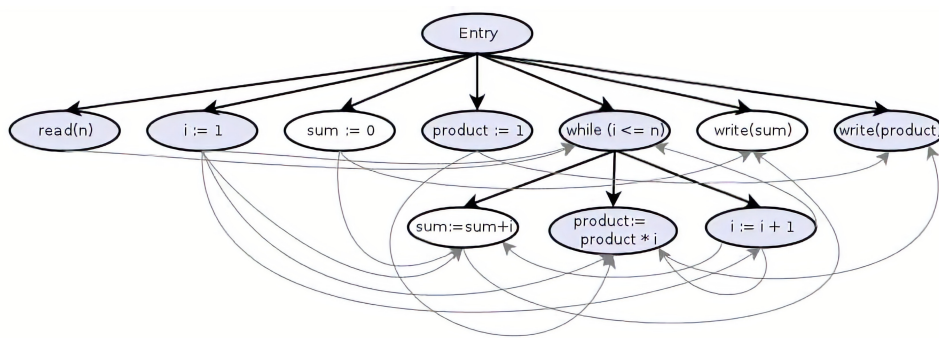


Рис. 1.6. Приклад графа програмних залежностей [15]

нарних графів, що не обов'язково виконається для графа програмних залежностей;

- Такий метод не зможе знайти дублікати у коді, який не виконується у загальному випадку, оскільки у граф додаються лише виконані інструкції.

Приклад використання: [9].

### 1.2.5 Метод порівняння дерев

У цьому методі використовуються абстрактні синтаксичні дерева (АСД). Згідно з [18], абстрактне синтаксичне дерево – позначене і орієнтоване дерево, в якому внутрішні вершини співставлені з відповідними операторами мови програмування, а листя з відповідними операндами.

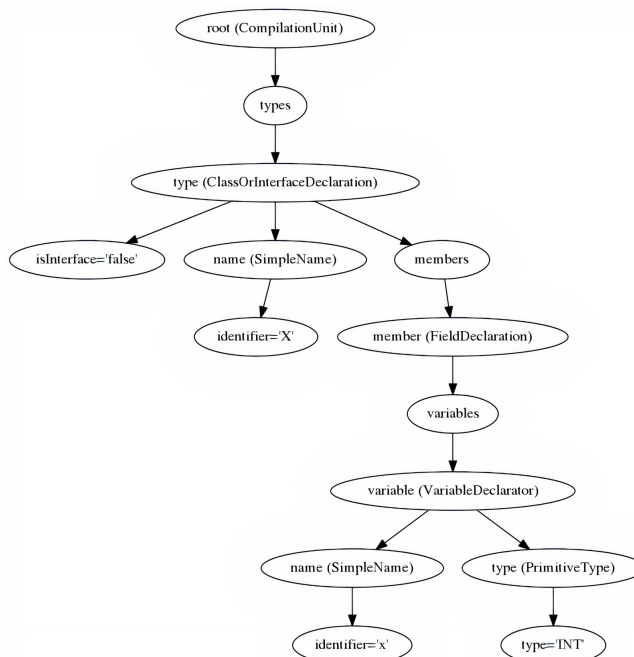


Рис. 1.7. Приклад абстрактного синтаксичного дерева [17]

Щоб визначити, чи є частина коду копією іншої частини, знаходять відповідні їм піддерева, а далі ці піддерева порівнюються між собою.

Підходів до порівняння піддерев досить багато. Так, наприклад, у роботі [6] усі піддерева, що відповідають класам у початковому коді, порівнюються кожен з кожним за допомогою обчислення відстані між деревами. Автор відмічає, що цей метод є найточнішим порівняно з іншими, але має дуже довгий час роботи. Наприклад, код плагінів `org.eclipse.compare-plugin`, що складався зі 114 класів перевірявся на копії більше ніж годину.

У роботі [3] теж стверджується, що алгоритм знаходження відстані між деревами має занадто велику складність обчислення, тому автором був запропонований інший підхід. Усі піддерева гешуються за допомогою вибраної «поганої» геш-функції та розподіляються у  $B$  бакетів, де  $B \approx N/10$ . Далі кожне піддерево порівнюється лише з піддеревами з цього ж бакету за формулою:

$$\text{Схожість} = \frac{2 * S}{2 * S + L + R}.$$

$S$  – кількість однакових вершин у обох піддеревах,  $L$  – кількість вершин, що присутні лише у першому піддереві,  $R$  – кількість вершин, що є тільки у другому піддереві.

Отже, головними перешкодами до використання цього методу є:

- Великі час роботи та алгоритмічна складність; [1]
- Досить низький відсоток знайдених копій через використання додаткових евристик. [5]

Далі буде запропоновано новий підхід, що значно зменшить час роботи, необхідний для знаходження копій, та, у той же час, збільшить ефективність їх знаходження.

## РОЗДІЛ 2.

### НОВИЙ АЛГОРИТМ ПОШУКУ ПОВТОРЮВАНИХ ЧАСТИН

Алгоритм складається із 4 кроків:

1. парсинг коду та перетворення його у абстрактне синтаксичне дерево;
2. визначення частин коду, які має сенс порівнювати;
3. знаходження повторюваних частин;
4. перетворення знайденого результату до конкретних елементів у коді.

Далі опишемо детальніше кожен крок.

#### 2.1. Парсинг коду

Компілятор практично будь-якої мови програмування на якомусь кроку перетворює код у абстрактне синтаксичне дерево. Для простоти, будемо працювати з кодом, написаним на мові програмування Java. Щоб перетворити код у абстрактне синтаксичне дерево, використаємо `JavaParser`. Алгоритм не складно змінити для підтримки будь-якої іншої мови програмування.

#### 2.2. Визначення частин коду, які підлягають порівнянню

У загальному випадку, структура коду у об'єктно-орієнтованих мовах програмування виглядає таким чином:

```
import com.google.tools;
class X {
    int a=0;
    X(int a, int b) {
        this.a = a;
    }
    void incrementAndPrint() {
        a++;
    }
}
```

Рис. 2.1. Приклад коду у об'єктно-орієнтованих мовах програмування

Можна визначити головні елементи практично кожної програми, а саме:

- підключення інших пакетів, бібліотек;
- декларування класу та його елементи (поля);
- декларування функцій та обчислення якогось результату.

Порівнянню і подальшому опрацюванню підлягають лише ті частини коду, які можна винести до іншої функції. Цими елементами є тільки ствердження у функціях.

Пояснемо на прикладі:

```
import com.google.tools;
class X {
    int a=0;
    X(int a) {
        this.a = a;
    }
    void incrementAndPrint() {
        a++;
        print(a);
    }
}
```

Рис. 2.2. Приклад коду

Курсивом виділені фрагменти коду, що будуть далі опрацьовані.

Визначимо вираз («expression») як найменшу неподільну операцію та параметр до цієї операції.

Будь яке велике обчислення можна розбити на вирази.

У операціях розгалуження вважатимемо виразами лише додаткові умови у них, але не самі операції.

Блок – непорожня послідовність виразів, укладених між фігурними дужками та впорядкованих за порядком обходу алгоритма DFS у абстрактному синтаксичному дереві.

Алгоритм розбиває код на блоки таким чином, що вміст одного блоку не зустрічається у іншому.

В кожного виразу є свої координати: перша координата( $x$ ) – номер блоку, у якому є цей вираз, друга координата( $y$ ) – знаходження виразу у блоці.

```

void func(){
    int x = 10;
    x = x+1;
    while (x>3){
        System.out.println(x*2);
        x--;
    }
}

```

[int x = 10;; x = x + 1;; x > 3,  
System.out.println(x \* 2);, x--;]

Рис. 2.3. Приклад розбиття коду на вирази у блоці

Порівнянню з іншою послідовністю підлягає будь-яка послідовність виразів, що йдуть поспіль, та знаходяться у одному блоці.

### 2.3. Знаходження повторюваних частин

У загальному випадку, у клонах є доволі багато ідентичних виразів. Для простоти будемо вважати, що клони починаються з ідентичного виразу. У майбутньому планується підтримка випадку, коли клони починаються не з ідентичного виразу.

Знаходження повторюваних частин працює наступним чином:

- для кожного виразу обчислити геш-функцію для кожного виразу;  
У геш-функції враховуються усі типи кожного з вершин АСД, що лежать нижче, ніж відповідна вершина до цього виразу; типи буквальних виразів (наприклад, 123.0f це тип float, "123" – тип String).
- створити асоціативний масив, де ключем є геш, а значенням є список координат усіх виразів;
- перетворити асоціативний масив на масив зі списків до кожного ключа;
- відсортувати масив за зростанням наступної функції:

$$f(list) = \max_{\forall expr \in list} expr_y$$

Це потрібно для того, щоб потенційний відрізок не обмежувався іншим, вже відміченим як копію, відрізком.

- для кожного списку координат з цього масива:

### 1. Знайти максимум функції:

$$F(len) = (len - 2 * T - 1) * goodGraphs - len - 2$$

Де  $len$  – довжина відрізка виразів. При обчисленні функції створюється дерево структури відрізка виразів, де кожний вираз зустрічається один раз.

Один вираз є батьком («parent») іншого, якщо перший вираз є частиною операції розгалуження, і другий вираз знаходиться у тілі цієї операції. В кожній вершині є свій надпис («label»). Вважатимемо надписом кожної вершини тип відповідного виразу.

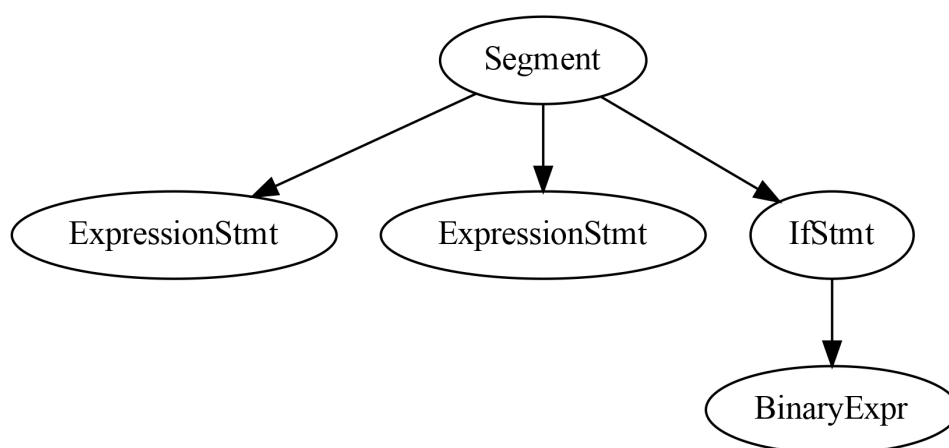


Рис. 2.4. Приклад створюваного графу

$goodGraphs$  – кількість графів (окрім першого), для яких відстань зміни графа до першого є меншою за константу  $T^1$ .

Відстань зміни графа вимірюється за 3 параметрами:

- вартість видалення вершини (дорівнює 0.5);
- вартість зміни «надпису» вершини (дорівнює 1);
- вартість вставки вершини (дорівнює 0.5).

Для кращого знаходження дублікатів III типу, вартості видалення і вставки зменшені, тоді вартість переставлення виразу буде такою ж, як і вартість зміни «надпису».

<sup>1</sup>Відстань зміни графа обчислюється за допомогою алгоритму APTEd. [11][12]

Слід зазначити, що

$$D(F) = [3, \max PotentialLen] \setminus \forall len :$$

$$\exists piece, piece_y \geq start_y + len, piece \in varusages,$$

$$\exists decl, start_y \leq decl_y < start_y + len,$$

$$decl \in declarations, decl_{var} = piece_{var};$$

$$\max PotentialLen = \min(exprCount, y_{nearest}) - start_y,$$

де:

- 3 – константа, обрана емпірично, щоб відрізки меншої довжини не вважалися клонами, бо вони не мають сенсу для програміста.
- $\max PotentialLen$  – максимально можлива довжина відрізка;
- $start$  – почтаковий вираз відрізка;
- $varusages$  – множина усіх використань змінних у цьому відрізку;
- $declarations$  – множина усіх декларацій змінних у цьому відрізку;
- $decl_{var}, piece_{var}$  – імена відповідних змінних;
- $exprCount$  – кількість виразів у блоці;
- $y_{nearest}$  – у-координата найближчого виразу-клона.

Додаткова умова додана, щоб не було таких випадків, коли декларація використованої змінної вже відсутня.  $F(len)$  – приблизна кількість рядків у коді, що будуть зекономлені.

$$F(len) = len * goodGraphs - goodGraphs - 2 * T * goodGraphs - len - 2,$$

де:

- $len * goodGraphs$  – приблизна початкова кількість рядків коду;
- $goodGraphs$  – кількість потрібних викликів нової функції, кожен виклик зазвичай займає 1 рядок;
- $2 * T * goodGraphs$  – приблизна кількість рядків коду, потрібного для врахування усіх відмінностей між послідовностями виразів;
- $len$  – приблизна кількість рядків коду, що є повністю ідентичним між усіма послідовностями;

– 2 – кількість рядків, необхідна щоб записати функцію у коді.

$$F(len) = len * goodGraphs - goodGraphs - 2 * T * goodGraphs - len - 2$$

$$F(len) = (len - 1) * goodGraphs - 2 * T * goodGraphs - len - 2$$

$$F(len) = (len - 2 * T - 1) * goodGraphs - len - 2$$

2. Якщо кількість відрізків більша за 1 і  $F(len) > 0$ , то усі вирази у відрізьку відмітити як вирази-клони.  $F(len)$  повинно бути більше ніж 0, бо у інакшому випадку у сгенерованому коді буде більше рядків, а це не має сенсу.

## 2.4. Перетворення на послідовність фрагментів у коді

Попереднім кроком були знайдені усі вирази-клони, але їх ще потрібно перетворити у послідовність фрагментів, бо самі по собі вирази показують лише уривки з початкового коду.

<pre>[xx == 466.0f, xx == 143.0f, xx ==  466.0f, xx++;, xx == 143.0f, xx++;]</pre>	<pre>if (xx==466.0f    xx==143.0f) {   if (xx==466.0f)     xx++;   else if (xx==143.0f)     xx++; }</pre>
--	---

Рис. 2.5. Приклад послідовності виразів-клонів та відповідна їм частина коду

Зробимо перетворення наступним чином: оберемо усі вершини у абстрактному синтаксичному дереві (АСД), для яких

$$\exists piece, LCA(son, piece_{ast}) = son,$$

де:

- $son$  – син закріпленої за блоком вершини у АСД,
- $piece$  – вираз,
- $piece_{ast}$  – відповідна виразу вершина у АСД.



Таким чином, для послідовності виразів-клонів закріплені вершини у абстрактному синтаксичному дереві. У подальшому називатимемо ці вершини інструкціями.

## 2.5. Використання алгоритму для видалення клонів

Для видалення списку повторюваних частин коду об'єднаємо їх у функцію. Зробимо це наступним алгоритмом:

1. Кожна повторювана частина коду являє собою список інструкцій. Тілом функції буде найбільша спільна підпослідовність цих списків.
2. Виконання кожної іншої інструкції, що не вийшла до найбільшої спільної підпослідовності, обернемо у оператор `if`. Умовою цього оператора стане параметр функції типу `boolean`.
3. Додати до усіх використаних змінних, полів, викликів функцій з іншої частини коду назву класу.
4. Усі буквальні вирази, що є у найбільшій спільній підпослідовності, та не є однаковими для усіх відповідних інструкцій, замінимо на параметр функції. Тип параметру може бути визначений за допомогою бібліотеки `JavaParser`.
5. Для кожної повторюваної частини замінити першу інструкцію у списку на виклик нової функції, інші інструкції видалити.
6. Нову функцію записати у файл `Copied.java`.

```

static final double PI = 3.1415;    static final double PI = 3.1415;
static void veryImportantFunction() static void superComputing()
{
    double xx =
        Math.cos(PI/2)-Math.sin(PI/2);
    double yy =
        Math.sin(PI/2)+Math.cos(PI/2);
    if (xx==456.0f || xx==123.0f){
        if (xx==456.0f)
            xx++;
        else if (xx==123.0f)
            xx++;
    }
    xx*=2;
    yy*=2;
    System.out.println(xx+" "+yy);
}

```

Рис. 2.6. Приклад коду до виконання алгоритмів

```

static final double PI =
    3.1415;
static void
    veryImportantFunction() {
Copied.
        veryImportantFunction(456.0f,
            123.0f, 456.0f, 123.0f);
    }
static void superComputing() {
Copied.
        veryImportantFunction(345.0f,
            0f, 345.0f, 0.0f);
    }
}

static final double PI = 3.1415;
static void veryImportantFunction(Double
    literalXx, Double literalXx2, Double
    literalXx3, Double literalXx4) {
    double xx = Math.cos(Main.PI / 2) -
        Math.sin(Main.PI / 2);
    double yy = Math.sin(Main.PI / 2) +
        Math.cos(Main.PI / 2);
    if (xx == literalXx || xx == literalXx2)
    {
        if (xx == literalXx3)
            xx++;
        else if (xx == literalXx4)
            xx++;
    }
    xx *= 2;
    yy *= 2;
    System.out.println(xx+" "+yy);
}

```

Рис. 2.7. Приклад роботи обох алгоритмів: знаходження та видалення клонів

### РОЗДІЛ 3.

#### ПОРІВНЯННЯ З ІНШИМИ МЕТОДАМИ

Порівняємо запропонований алгоритм з іншими методами.

Існують різні способи порівняння методів пошуку дублікатів у коді, доволі часто використованим способом є бенчмарк [4].

Проте, у роботі [16] зазначається, що цей бенчмарк може бути необ'єктивним, оскільки перевірка того, чи дійсно 2 частини коду є клонами виконується людиною, тому у використаних при розробці бенчмарка інструментів з'являється додаткова перевага. Також відмічається, що у новому бенчмарку «BigCloneEval» [14] ця проблема нівельована, тому будемо використовувати саме його.

Зазвичай порівняння виконується за 2 основними параметрами:

- Точність («precision»). Інструмент повинен бути детектувати якомога менше хибно-позитивних клонів.  
Точність обчислюється як відношення правильно знайдених пар клонів до усіх знайдених інструментом пар клонів.
- Відклик («recall»). Інструмент має коректно знаходити більшість пар клонів серед можливих (тобто відмічених людиною).  
Відклик обчислюється як відношення кількості коректно знайдених пар клонів до кількості відмічених людиною пар клонів.
- Час роботи інструменту.

Для порівняння з новим алгоритмом обрані наступні програми:

- CCFinderX;
- PMD;
- CloneDR.

Ці програми є реалізаціями різних методів, описаних у розділі 1. Виконаємо порівняння на наступному обладнанні:

- процесор – Intel Core i5-8250U, 1.60ГГц
- оперативна пам'ять – 8 Гб;

- операційна система – Windows 10.

Отримані наступні результати:

Назва інструменту	Точність	Відклик
CCFinderX	0.57	0.5
PMD	0.48	0.58
CloneDR	0.82	0.47
Новий алгоритм	0.76	0.53

Таблиця 3.1. Порівняння інструментів знаходження клонів

Як бачимо, новий алгоритм є значно точнішим, ніж інструменти CCFinderX, PMD менш точним ніж CloneDR, проте в цього інструменту відклик є меншим. Відклик нового алгоритму є більшим, ніж у всіх програм, окрім PMD, в якого точність є значно меншою.

Порівняємо час роботи алгоритму із іншими програмами на початковому коді наступних проектів:

- DnsJava (~40000 рядків коду);
- JFreeChart (~300000 рядків);
- Tomcat (~600000 рядків);
- OpenOffice (~800000 рядків).

Отримані результати наведені у таблиці.

Час обробки проекту (с), Назва інструменту	DnsJava	JFreeChart	Tomcat	OpenOffice
CCFinderX	12	64	195	289
PMD	1	3	5	15
CloneDR	18	270	1378	3249
Новий алгоритм	5	19	59	97

Таблиця 3.2. Потрібний інструментам час для обробки різних проектів

Можна побачити, що новий алгоритм є значно швидшим, ніж CCFinderX та

CloneDR, але повільнішим, ніж PMD.

Він є оптимальним варіантом за обраними показниками.

## **РОЗДІЛ 4.**

### **ВИСНОВКИ**

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. A Systematic Review on Code Clone Detection / Q. U. Ain [та ін.] // IEEE Access. — 2019. — Т. 7. — С. 86121—86144. — DOI: 10.1109/ACCESS.2019.2918202.
2. An Empirical Study on the Maintenance of Source Code Clones / S. Thummalapenta [та ін.] // Empirical Software Engineering. — 2010. — Лют. — Т. 15. — С. 1—34. — DOI: 10.1007/s10664-009-9108-x.
3. Clone detection using abstract syntax trees / I. D. Baxter [та ін.] // Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272). — 1998. — С. 368—377. — DOI: 10.1109/ICSM.1998.738528.
4. Comparison and Evaluation of Clone Detection Tools / S. Bellon [та ін.] // IEEE Transactions on Software Engineering. — 2007. — Лип. — Т. 33. — С. 577—591. — DOI: 10.1109/TSE.2007.70725.
5. *Dang S.* Performance Evaluation of Clone Detection Tools // International Journal of Science and Research (IJSR). — 2015. — Квіт. — Т. 4. — С. 1903—1906.
6. Detecting similar Java classes using tree algorithms / T. Sager [та ін.] //. — 01.2006. — С. 65—71. — DOI: 10.1145/1137983.1138000.
7. *Ferrante J., Ottenstein K., Warren J.* The Program Dependence Graph and Its Use in Optimization. // ACM Transactions on Programming Languages and Systems. — 1987. — Лип. — Т. 9. — С. 319—349. — DOI: 10.1145/24039.24041.
8. *Gautam P., Saini H.* Various Code Clone Detection Techniques and Tools: A Comprehensive Survey // Smart Trends in Information Technology and Computer Communications / за ред. A. Unal [та ін.]. — Singapore : Springer Singapore, 2016. — С. 655—667. — ISBN 978-981-10-3433-6.
9. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis / C. Liu [та ін.] // Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. — Philadelphia, PA, USA : Association for Computing Machinery, 2006. — С. 872—881. — (KDD '06). — ISBN 1595933395. — DOI: 10.1145/1150402.1150522. — URL: <https://doi.org/10.1145/1150402.1150522>.

10. *Koschke R., Falke R., Frenzel P.* Clone Detection Using Abstract Syntax Suffix Trees // 2006 13th Working Conference on Reverse Engineering. — 2006. — С. 253—262. — DOI: 10.1109/WCRE.2006.18.
11. *Pawlik M., Augsten N.* Efficient Computation of the Tree Edit Distance // ACM Trans. Database Syst. — New York, NY, USA, 2015. — Бep. — Т. 40, № 1. — ISSN 0362-5915. — DOI: 10.1145/2699485. — URL: <https://doi.org/10.1145/2699485>.
12. *Pawlik M., Augsten N.* Tree edit distance: Robust and memory-efficient // Information Systems. — 2016. — Т. 56. — С. 157—173. — ISSN 0306-4379. — DOI: <https://doi.org/10.1016/j.is.2015.08.004>. — URL: <http://www.sciencedirect.com/science/article/pii/S0306437915001611>.
13. Structural Function Based Code Clone Detection Using a New Hybrid Technique / Y. Yang [та ін.] // 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC). Т. 01. — 2018. — С. 286—291. — DOI: 10.1109/COMPSAC.2018.00045.
14. *Svajlenko J., Roy C.* Evaluating Modern Clone Detection Tools // 2014 IEEE International Conference on Software Maintenance and Evolution. — 2014. — С. 321—330.
15. *Tengeri D., Havasi F.* Database Slicing on Relational Databases // Acta Cybernetica. — 2014. — Січ. — Т. 21. — С. 629—653. — DOI: 10.14232/actacyb.21.4.2014.6.
16. *Vislavski T., Rakic G.* Code Clone Benchmarks Overview // SQAMIA. — 2018.
17. *Viswanadha S., Gesser J.* Inspecting an AST. — 2018. — URL: <https://javaparser.org/inspecting-an-ast/>; Онлайн.
18. *Bikinedia.* Абстрактне синтаксичне дерево — Вікіпедія, — 2020. — URL: [https://uk.wikipedia.org/w/index.php?title=%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D0%B5\\_%D1%81%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%BD%D0%B5\\_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE&oldid=27659104](https://uk.wikipedia.org/w/index.php?title=%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D0%B5_%D1%81%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%BD%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE&oldid=27659104); [Онлайн; цитовано 5-листопад-2020].