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

Харків – 2021

Тези

Метод пошуку та усунення повторюваних частин у початковому коді програмного забезпечення;

Автор роботи: Човпан Ігор Сергійович;
Харківське територіальне відділення МАН України;
Харківський навчально-виховний комплекс №45 «Академічна гімназія» Харківської міської ради Харківської області; 11 клас; м. Харків;
Науковий керівник: Руккас Кирило Маркович, професор кафедри теоретичної та прикладної інформатики механікоматематичного факультету Харківського національного університету ім. В.Н. Каразіна, доктор технічних наук, доцент.

1). Вступ. Серед розробників програмного забезпечення практика копіювання та вставки коду є досить поширеною. Вона може призводити до катастрофічних наслідків при обслуговуванні програмного забезпечення. Копії стають джерелом помилок та вразливостей, для їх підтримки потрібно забагато ресурсів. Через досить велику відносну кількість копій у початковому коді, пошук та їх усунення є важливою задачею.

Існує досить багато методів пошуку копій, але в кожного є свої недоліки. Наприклад, у методах, використовуючих абстрактне синтаксичне дерево, головною перешкодою є завелика алгоритмічна складність. Евристики, що застосовуються при усуненні цієї проблеми, є досить неефективними.

Метою роботи є аналіз існуючих методів знаходження повторюваних частин, створення нового алгоритму знаходження й усунення копій, що має задовільні ефективність та складність.

2). Результати. Був створений новий алгоритм, який визначає потенційні місця знаходження копій, перетворює код у список виразів, обчислює максимум функції кількості потенційно видалених рядків коду від відрізка виразів, відмічає та замінює відповідні виразам частини коду на виклик новоствореної функції.

Новий метод знаходить та видаляє клони у великих проектах (Apache OpenOffice, Tomcat, тощо) за досить малий час, має задовільні показники точності та відклику.

За результатами експерименту продемонстровано, що алгоритм є оптимальним.

ЗМІСТ

ВСТУП

Серед розробників програмного забезпечення практика копіювання та вставки коду є досить поширеною. Не дивлячись на те, що це може бути корисним та зручним навиком у короткостроковій перспективі, подібна практика призводить до катастрофічних наслідків при обслуговуванні програмного забезпечення.
Так, наприклад, будь яке удосконалення чи усунення помилки потрібно буде робити для кожної копії, що робиться далеко не завжди. Через це істотно збільшується кількість потенційних вразливостей та помилок у коді, на підтримку програмного забезпечення великих розмірів потрібно значно більше ресурсів.
Досліди показують, що доволі велику частину від початкового коду проекту займають копії (у середньому 5-10% [**Baxter98**]).
Через це, пошук та усунення повторюваних частин у початковому коді є важливою задачею.
Існує досить багато методів пошуку копій, але в кожного є свої недоліки. Наприклад, у методах, використовуючих абстрактне синтаксичне дерево, головною перешкодою є завелика алгоритмічна складність. Евристики, що застосовуються при усуненні цієї проблеми, є досить неефективними. Наприклад, у роботі [**Baxter98**] асимптотика створеного алгоритма залежить квадратично від кількості рядків у початковому коді. Це призводить до того, що для пошуку клонів у проектах відносно невеликого розміру потрібно занадто багато часу. [**Sager06**]
Метою роботи є аналіз існуючих методів знаходження повторюваних частин у початковому коді програмного забезпечення, та створення нового алгоритму знаходження і усунення копій, що має задовільні ефективність та час, потрібний для обчислення на проектах великого розміру.

РОЗДІЛ 1.
ХАРАКТЕРИСТИКА ІСНУЮЧИХ МЕТОДІВ ЗНАХОДЖЕННЯ ПОВТОРЮВАНИХ ЧАСТИН У КОДІ

Щоб проаналізувати методи знахождення 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 [**Gautam16**], [**Dang15**] та [**Thummalapenta10**], виділяється 4 головних типи повторюваних частин.

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

```
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;
```

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

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 = sum3+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 программування.

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, зміні 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. [**Koschke06**] 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 повторювани частини всередині функції. Прикладом використання є [**Yang18**].

1.2.4 Застосування графа програмних залежностей

Згідно з [**Ferrante87**], 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

Рис. 1.6. Приклад графа програмних залежностей [**pdg-example**]

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 планарних 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: [**Liu06**].

1.2.5 Метод порівняння дерев

У цьому методі використовуються абстрактні синтаксичні дерева (АСД). Згідно з [**wiki:ast**], 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 [**ast-example**]

Щоб визначити, чи є частина коду копією іншої частини, знаходять відповідні їм піддерева, а далі ці піддерева порівнюються між собою.
Підходів до порівяння піддерев досить багато. Так, наприклад, у роботі [**Sager06**] 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-plug-in`, що складався зі 114 класів перевірявся на копії більше ніж годину.
У роботі [**Baxter98**] теж стверджується, що алгоритм знахождення 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 дру-

гому піддереві.

Отже, головними перешкодами до використання цього методу є:

- Великі час роботи та алгоритмічна складність; [**Ain19**]
- Досить низький відсоток знайдених копій через використання додаткових евристик. [**Dang15**]

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){                    [int x = 10;, x = x + 1;, x > 3,
    System.out.println(x*2);         System.out.println(x * 2);, x--;]
    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 списка 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 кожної вершини 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».
Слід зазначити, що

$$D(F) = [3, maxPotentialLen] \backslash \forall len :$$

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

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

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

[1] Відстань зміни графа обчислюється за допомогою алгоритму APTED. [**Pawlik15**][**Pawlik16**]

$$maxPotentialLen = min(exprCount, y_{nearest}) - start_y,$$

де:

– 3 – константа, обрана емпірично, щоб відрізки меншої довжини не вважалися клонами, бо вони не мають сенсу для програміста.
– $maxPotentialLen$ – 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 відрізку 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.

```
[xx == 466.0f,xx == 143.0f,xx ==
   466.0f,xx++;,xx == 143.0f,xx++;]
```

```
if (xx==466.0f || xx==143.0f) {
  if (xx==466.0f)
    xx++;
  else if (xx==143.0f)
    xx++;
}
```

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 списка 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 війшла до найбільшої спільної підпослідовності, обернемо у оператор `if`. Умовою цього оператору 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 void veryImportantFunction()
{
  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);
}
```

```
static final double PI = 3.1415;
static void superComputing()
{
  double aa =
     Math.cos(PI/2)-Math.sin(PI/2);
  double bb =
     Math.sin(PI/2)+Math.cos(PI/2);
  if (aa==345.0f || aa==0f){
    if (aa==345.0f)
      aa++;
    else if (aa==0.0f)
      aa++;
  }
  aa*=2;
  bb*=2;
  System.out.println(aa+" "+bb);
}
```

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);
}
```

(а) Змінений код початкових функцій

```
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);
}
```

(б) Нова функція, що була створена алгоритмом

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

РОЗДІЛ 3.
АНАЛІЗ ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНОГО МЕТОДУ

Порівняємо запропонований алгоритм з іншими методами на наступному обладнанні:

- процесор – `Intel Core i5-8250U`, 1.60Ггц
- оперативна пам'ять – 8 Гб;
- операційна система – `Windows 10`.

Зазвичай порівняння виконується за 3 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.

Використання інших характеристик так чи інакше потребує ручної перевірки, що не є об'єктивним через потенційну упередженість.

3.1. Порівняння з іншими методами за точністю та відкликом

Для підрахунку перших двох параметрів необхідно вияснити, які пари клонів потрібно вважати правильними.
Це можна зробити двома способами: перевіряючи на реальних проектах знайдені інструментами пари клонів ([**Bellon07**]), або порівнюючи результат роботи інструменту зі згенерованими парами.
Як показано у роботі [**Roy18**], використання першого варіанту проблематично, оскільки у такому разі ручна перевірка неминуча. Відмічається, що присутня значна непослідовність у визначенні типів клонів. У роботі [**Charpentier15**]

також зазначається на розбіжності висновків незалежних суддів у визначенні правильності знайдених пар клонів.
Приведені вище недоліки практично ліквідовані у бенчмарку «BigCloneBench» [**Svajlenko14**], який використовує автоматично сгенеровану базу даних клонів «IJaDataset», для якої були узяті реалізації визначених алгоритмів з подальшою їх зміною декількома способами: видалення рядка, коментування рядка, тощо. На його основі був створений фреймворк «BigCloneEval» для спрощення перевірки інструментів знаходження клонів, який і будемо використовувати для підрахунку параметрів точності та відклику.

Для порівняння з новим алгоритмом обрані наступні програми:

- CCFinderX;
- PMD;
- CloneDR.

Ці програми є реалізаціями різних методів, описаних у розділі **??**. Отримані наступні результати:

Таблиця 3.1. Порівняння інструментів знаходження клонів за точністю та відкликом

Назва інструменту	Точність	Відклик
CCFinderX	0.57	0.5
PMD	0.48	0.58
CloneDR	0.82	0.47
Новий алгоритм	0.76	0.55

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, а відклик – лише трохи більшим.
Таким чином, ефективність нового алгоритму за показниками точності та виклику є задовільною.

3.2. Порівняння часу роботи

На відміну від відклику та точності, час роботи є параметром, який можна оцінювати на коді реальних проектів.
Порівняємо час роботи алгоритму із іншими програмами на початковому коді 16 проектів:

- Eclipse (∼20000 рядків `Java`-коду).
- Bval (∼30000 рядків);
- Attic-onami (∼40000 рядків);
- DnsJava (∼40000 рядків коду);
- Any23 (∼50000 рядків);
- Gorra (∼70000 рядків);
- JSPWiki (∼90000 рядків);
- OpenNLP (∼100000 рядків);
- JHotDraw (∼120000 рядків);
- Maven (∼120000 рядків);
- Oodt (∼160000 рядків);
- Juddi (∼180000 рядків);
- ArgoUML (∼300000 рядків);
- JFreeChart (∼300000 рядків);
- Tomcat (∼600000 рядків);
- OpenOffice (∼800000 рядків).

Отримані результати наведені на графіках.

Рис. 3.1. Залежність часу виконання від кількості рядків коду

Рис. 3.2. Залежність часу виконання інструментів (окрім CloneDR) від кількості рядків коду

За наведеними вище даними, можна побачити, що новий алгоритм є значно швидшим, ніж `CCFinderX` та `CloneDR` (у 3 та 30 разів відповідно), але трохи повільнишим, ніж `PMD`.

Зазначимо, що системи дуже великого розміру оброблюються швидко: проект із $\sim$800000 рядками коду оброблюється менш, ніж за 2 хвилини.

Час роботи, потрібний алгоритму для обчислення результату, є задовільним.

ВИСНОВКИ

Виділені 4 головних типи повторюваних частин:

- I тип – повна копія практично з відсутніми модифікаціями;
- II тип – змінюються лише назви змінних, функцій, класів;
- III тип – копія із доданими, зміненими, або видаленими інструкціями, зміненими назвами об’єктів;
- IV тип – копія, що значно відрізняється синтаксично, але робить ідентичні обчислювання.

Були проаналізовані наступні методи знаходження повторюваних частин у початковому коді програмного забезпечення:

- пошук збігу рядків початкового коду;
- використання токенів;
- метод порівняння функцій;
- застосування графа програмних залежностей;
- метод порівняння дерев.

Виявлені основні переваги і недоліки кожного з методів.
Створений новий алгоритм, який усуває недоліки методу порівняння дерев, а саме:

- Надто великі час роботи та алгоритмічна складність;
- Досить низький коефіцієнт відклику через використання додаткових евристик.

Зроблено порівняння між новим алгоритмом та існуючими методами за 3 основними показниками: точністю, відкликом та часом роботи.
Продемонстровано, що новий метод є оптимальним: має задовільні показники точності та відклику (0.76 та 0.55 відповідно); може використовуватитсь на системах дуже великого розміру: проект з $\sim$800000 рядками коду оброблюється менш, ніж за 2 хвилини.