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

РОЗРОБКА МЕТОДА 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 – 2020

Тези

...

ЗМІСТ

ВСТУП

Ваш ВСТУП

РОЗДІЛ 1.
ХАРАКТЕРИСТИКА ІСНУЮЧИХ МЕТОДІВ ЗНАХОДЖЕННЯ ПОВТОРЮВАНИХ ЧАСТИН У КОДІ

Для того 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 [7], [4] та [2], 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;
```

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

```
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);
}
```

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);
}
```

Приклад копії III типу

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;
- застосування графа програмних залежностей;
- метод порівняння дерев.

Далі визначимо усі переваги і недоліки кожного з методів.

```
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];
}
```

Приклад копії IV типу

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 метода:

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

Недоліки метода:

- 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. [9] 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++;
```

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 коду. Прикладом використання є [12].

1.2.4 Застосування графа програмних залежностей

Згідно з [6], 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

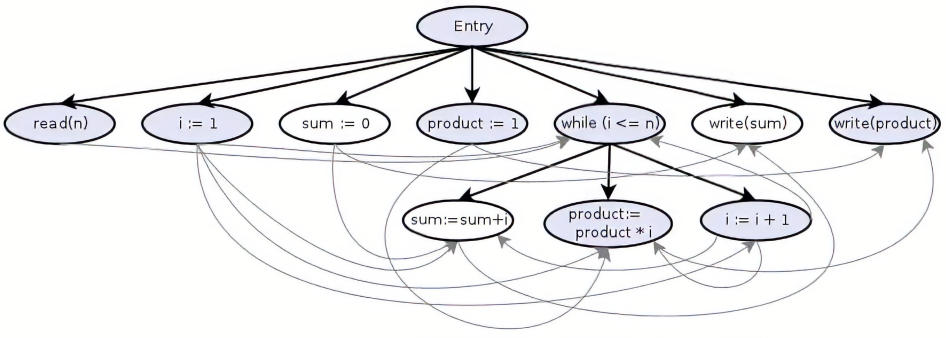

Рис. 1. Приклад графа програмних залежностей [13]

ізоморфні. Головною перевагою є те, що цей граф не залежить від переставлення інструкції програми; зміни назв функцій, змінних, 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-

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: [8].

1.2.5 Метод порівняння дерев

У цьому методі використовуються абстрактні синтаксичні дерева (АСД). Згідно з [15], абстрактне синтаксичне дерево – позначене і орієнтоване дерево, в якому внутрішні вершини співставлені з відповідними операторами мови програмування, а листя з відповідними операндами.

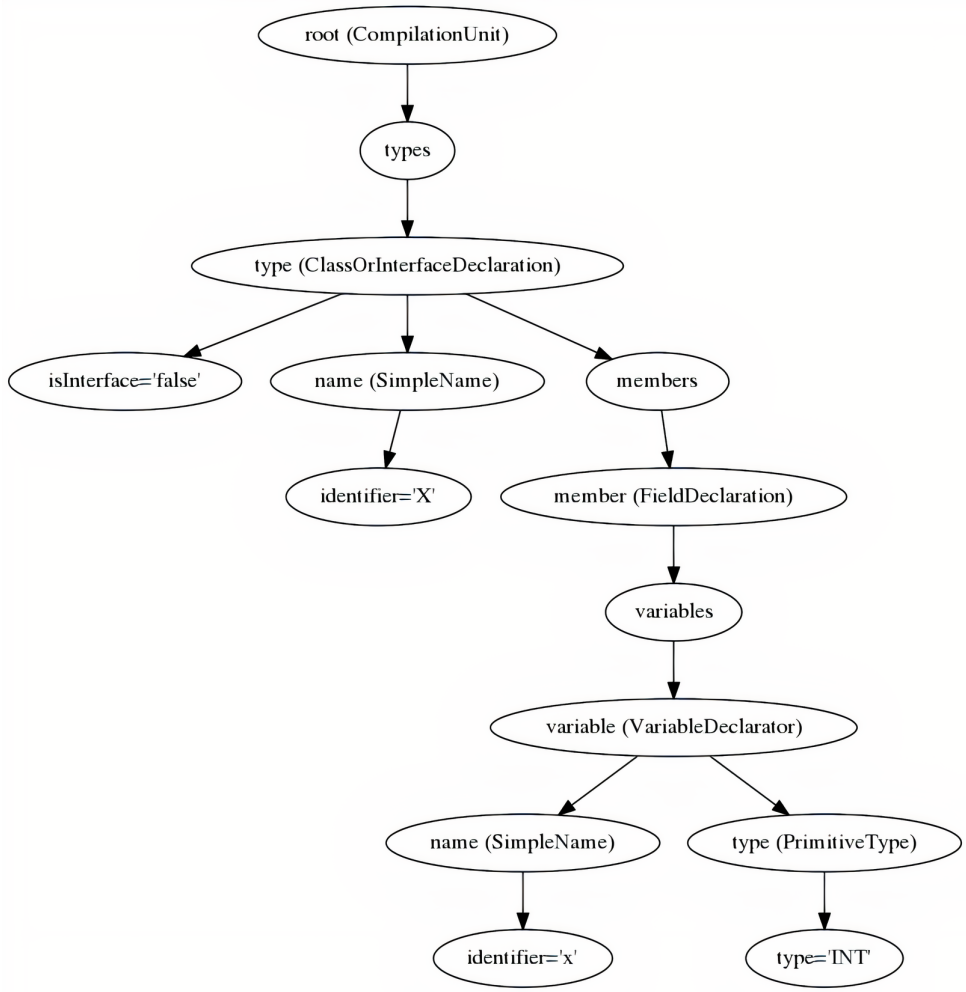

Рис. 2. Приклад абстрактного синтаксичного дерева [14]

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

Підходів порівняння піддерев досить багато. Так, наприклад, у роботі [5] 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 [3] теж стверджується, що алгоритм знахождення між деревами має занадто велику складність обчислення, тому автором було запропоновано 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. [4]

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

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++;
    print(a);
  }
}
```

Із прикладу 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.

Будемо вважати, що 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);
  }
}
```

Приклад коду

Курсивом виділені фрагменти коду, що будуть далі опрацьовані.
Визначимо вираз («expression») як найменша неподільна операція 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) – номер у вираза послідовності цього блока.

```
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--;
  }
}
```

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 між собою 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-функція 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 (`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.

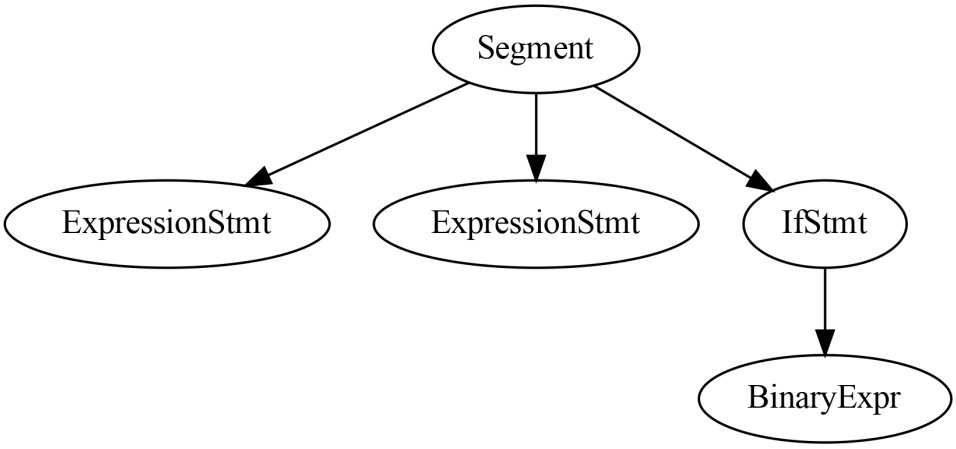

Рис. 3. Приклад створюваного графу

$goodGraphs$ – 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);

[1]Відстань зміни графа обчислюється за допомогою алгоритму APTED. [10][11]

– 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

$$D(F) = [3, maxPotentialLen]$$

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

де:

– 3 – константа, обрана емпірично, щоб відрізки меншої довжини не вважалися клонами, бо вони не мають сенсу для програміста.
– $maxPotentialLen$ – максимально можлива довжина відрізка;
– $exprCount$ – кількість виразів у блоці;
– $y_{nearest}$ – у-координата найближчого виразу-клона;
– y_{start} – у-координата початкового виразу;

$F(len)$ – 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 фрагменти у коді

РОЗДІЛ 3.
ПОРІВНЯННЯ З ІНШИМИ МЕТОДАМИ

...

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. *Dang S.* Performance Evaluation of Clone Detection Tools // International Journal of Science and Research (IJSR). — 2015. — Квіт. — Т. 4. — С. 1903—1906.
5. Detecting similar Java classes using tree algorithms / T. Sager [та ін.] //. — 01.2006. — С. 65—71. — DOI: 10.1145/1137983.1138000.
6. *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.
7. *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.
8. 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.
9. *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.

10. *Pawlik M.*, *Augsten N.* Efficient Computation of the Tree Edit Distance // ACM Trans. Database Syst. — New York, NY, USA, 2015. — Бер. — Т. 40, № 1. — ISSN 0362-5915. — DOI: 10.1145/2699485. — URL: https://doi.org/10.1145/2699485.

11. *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.

12. 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.

13. *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.

14. *Viswanadha S.*, *Gesser J.* Inspecting an AST. — 2018. — URL: https://javaparser.org/inspecting-an-ast/ ; Онлайн.

15. *Вікіпедія*. 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 ; [Онлайн; цитовано 5-листопад-2020].