Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Комп'ютерно-інформаційних систем і програмної інженерії
(назва факультету)
Комп'ютерних наук
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр
(назва освітнього ступеня)

на тему: **Аналітичне опрацювання JavaScript кодів для веб-застосунків**

Виконав(ла): студент(ка) 6 курсу, групи САм-61
спеціальності
124 «Системний аналіз»
(шифр і назва спеціальності)

(підпис) Гац В. Р. (прізвище та ініціали)

Керівник (підпис) Гром'як Р. С. (прізвище та ініціали)

Нормоконтроль (підпис) Мацюк О. В. (прізвище та ініціали)

Завідувач кафедри (підпис) Боднарчук І.О. (прізвище та ініціали)

Рецензент (підпис) Карпінський М.П. (прізвище та ініціали)

Тернопіль
2020

АНОТАЦІЯ

Аналітичне опрацювання JavaScript-кодів для веб-застосунків// Кваліфікаційна робота «Магістр» // Гац Владислав Русланович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра комп'ютерних наук, група САм-61 // Тернопіль, 2020 // 85 с., рис. – 40 , табл. – 3 , бібліогр. – 59 , додат. –2.

Ключові слова: КЛОНУВАННЯ, КЛОНИ КОДУ, ДУБЛІКАТИ КЛОНУ, МЕТОДИ ПОШУКУ КЛОНІВ.

Для розробки високоякісного ПЗ необхідно знаходити клони коду і пов'язані з ними семантичні помилки в проектах, що містять десятки мільйонів рядків вихідного коду, при цьому забезпечивши високу точність(прийнятний рівень помилкових спрацьовувань) і масштабованість (прийнятний час роботи). Для адаптації скопійованого фрагмента коду розробник може відредагувати його. Іноді відредагований код може настільки відрізнятися від оригіналу, що визначити, звідки його скопіювали, практично неможливо.

В процесі виконання кваліфікаційної роботи:

В першому розділі проведено аналіз предметної області зокрема розглянуто типи клонів коду та методи пошуку, проведено теоретичний аналіз методики й сформовано задачі дослідження.

В другому розділі реалізовано розробку методів пошуку клонів коду на основі семантичного аналізу, пошук схожих під графів на основі ізоморфізму дерев та реалізовано апробацію методів пошуку клонів.

В третьому розділі висвітлено архітектуру та функціональність системи, запропоновано інтегровану систему тестування та розроблені методи та засоби пошуку клонів коду для JavaScript.

ANNOTATION

Analytical processing of JavaScript-codes for web applications // Qualification work "Master" // Gats Vladislav Ruslanovich // Ternopil National Technical University named after Ivan Pulyuy, Faculty of Computer Information Systems and Software Engineering, Department of Computer Science, Sam Group -61 // Ternopil, 2020 // 85 p., Fig. - 40, table. - 3, bibliogr. - 59, appendix. –2.

Key words: CLONING, CODE CLONES, DUPLICATE CLONE, METHODS OF CLONE SEARCH.

To develop high-quality software, it is necessary to find code clones and related semantic errors in projects containing tens of millions of lines of source code, while ensuring high accuracy (acceptable level of false positives) and scalability (acceptable runtime). To adapt the copied code snippet, the developer can edit it. Sometimes the edited code can be so different from the original that it is almost impossible to determine where it was copied from.

In the process of performing the qualification work:

In the first section the analysis of the subject area is carried out, in particular the types of code clones and search methods are considered, the theoretical analysis of the technique is carried out and the research tasks are formed.

In the second section, the development of methods for finding clones of code based on semantic analysis, the search for similar subgraphs based on the isomorphism of trees and the testing of methods for finding clones.

The third section highlights the architecture and functionality of the system, proposes an integrated testing system and developed methods and tools for finding code clones for JavaScript.

ЗМІСТ

ВСТУП

Актуальність теми. Широке поширення вільного програмного забезпечення призвело до частого використання готових фрагментів вихідного коду при розробці нового програмного забезпечення (ПЗ). Розробники можуть використовувати як інтернет-ресурси,так і код, написаний ними самими або їх колегами. Згідно з результатами досліджень, програми можуть містити до двадцяти відсотків клонів коду(копійованих фрагментів). Безконтрольне клонування може призвести до збільшення розміру початкового та бінарного коду програми, виникненню семантичних помилок, ускладнення підтримки ПЗ та ін. Відомо, що проекти FreeBSD і Linux містять сотні помилок, пов'язаних з клонуванням коду.

Для розробки високоякісного ПЗ необхідно знаходити клони коду і пов'язані з ними семантичні помилки в проектах, що містять десятки мільйонів рядків вихідного коду, при цьому забезпечивши високу точність(прийнятний рівень помилкових спрацьовувань) і масштабованість (прийнятний час роботи). Для адаптації скопійованого фрагмента коду розробник може відредагувати його. Іноді відредагований код може настільки відрізнятися від оригіналу, що визначити, звідки його скопіювали, практично неможливо.

У літературі розрізняються три основних типи клонів. КлонитимуT_1 виникають, коли при вставці клонованого фрагмента коду адаптація до контексту не проводиться. Клони типу T_1 виникають, коли адаптація зводиться_до заміни ідентифікаторів.

У більш складних випадках, коли адаптація вимагає не тільки заміни ідентифікаторів, а й інших змін, виникають клони типу T_3. З відомих п'яти підходів до пошуку клонів коду (текстового, лексичного,метричного, синтаксичного і семантичного) тільки останній дозволяє виявити клони типу T_3 з досить високою точністю. Але більша частина існуючих інструментів, що базуються на семантичному підході володіють великою обчислювальною складністю, що робить їх складними і не дозволяє використовувати в реальних проектах.

Частина знайдених клонованих фрагментів коду може бути неповністю адаптована до контексту, в який вони були вставлені, що призводить до виникнення семантичних помилок (фрагмент обчислює не те,що очікує розробник). Існуючі інструменти пошуку семантичних помилок можуть, в основному, виявляти помилки тільки в клонах типів T_1 і T_2.

Вони спочатку знаходять клони коду шляхом лексичного або синтаксичного аналізу, після чого проводиться додатковий аналіз для виявлення допущених помилок.

При цьому інструменти на основі лексичного аналізують високий рівень помилкових спрацьовувань (тобто низьку точність), оскільки не враховують контекст програми. Інструменти, що використовують синтаксичний аналіз, не можуть знайти всі семантичні помилки, оскільки після перейменування змінних може істотно змінитися абстракт несинтаксичне дерево.

Таким чином, на сьогоднішній день не багато інструментів пошуку клонів коду, які забезпечували б необхідний рівень точності і масштабування до десятків мільйонів рядків вихідного коду, ні досить точних інструментів пошуку семантичних помилок в клонах. Тому розробка адекватних математичних і програмних засобів, що дозволяють вирішувати вказані проблеми є актуальною задачею.

Зв'язок роботи з науковими програмами, планами, темами

Напрямок виконаних досліджень безпосередньо пов'язаний з науково-дослідним напрямком кафедри.

Мета і задачі дослідження

Метою роботи є розробка математичних і програмних засобів для пошуку клонів коду і семантичних помилок, що виникають при некоректній адаптації скопійованих фрагментів коду.

Для вирішення поставленої мети вирішуються наступні завдання:

1) проаналізувати недоліки існуючих підходів до пошуку клонів коду ісемантичних помилок, що виникають при неправильному використанніскопійованих фрагментів коду;

2) розробити і реалізувати методи пошуку клонів коду на основісемантичного аналізу програми, що володіють високою точністю і масштабуванням до десятків мільйонів рядків вихідного коду;

3) здійснити програмну реалізацію запропонованих математичних методів;

4) провести тестування та апробацію розроблених методів та програмних засобів.

Об'єкт дослідження – процеси пошуку клонів коду для JavaScript.

Предмет дослідження – методи та програмні засоби пошуку клонів коду для JavaScript.

Методи дослідження.

Дослідження проводилися на базі комплексного системного аналізу, семантичного аналізу, теорії графів.

Наукова новизна одержаних результатів

Запропоновано метод пошуку клонів коду на основі семантичного аналізу програм, який на відміну від відомих використовує перетворення графу залежностей програми в дерево з подальшим пошуком ізоморфних під дерев.

Практичне значення одержаних результатів

Для практичного використання отриманих теоретичних результатів розроблено програмне забезпечення пошуку клонів коду для JavaScript, що дозволяє здійснювати пошук клонів коду для всіх мов програмування, які підтримують трансляцію в проміжне представлення LLVM, зокрема C і C++.

Особистий внесок магістранта

Всі результати отримані автором самостійно.

1 АНАЗІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ

1.1. Типи клонів коду та методи їх пошуку

У роботі [1] розглядається три типи клонів коду і п’ять основних підходів до їх пошуку. Кожен підхід має свої переваги і обмеження. Залежно від розв’язуваної задачі і вимог, що пред’являються, вибирається конкретний підхід пошуку клонів коду, який дозволяє вирішити задачу оптимальним чином. Існують комбіновані методи пошуку клонів коду, що складаються у використанні різних підходів на різних фазах пошуку.

За визначенням фрагментом коду називається довільна послідовність рядків. Клони коду типів T_1, T_2, T_3 визначаються наступним чином (рисунок 1.1):

T_1. Фрагменти коду, які можуть відрізнятися тільки пробілами,коментарями і форматуванням коду;

T_2. Всі клони типу T_1, а також фрагменти коду, які можуть також відрізнятися: іменами змінних; типами змінних; значеннями змінних і констант;

T_3. Всі клони типу T_2, а також фрагменти коду, в яких можуть бути додані або видалені інструкції та змінні.

Ступінь схожості фрагментів коду F_1 і F_2 обернено пропорційна кількості операцій редагування фрагмента F_1 (перейменування змінних,видалення, додавання і зміна інструкцій), щоб вийшов фрагмент F_2.

Для пошуку клонів коду використовується п'ять основних підходів [2]:текстовий, лексичний, синтаксичний, семантичний і метричний. Розроблено також інструменти пошуку клонів, в яких застосовуються комбінації декількох підходів.

Оригінальний код	Клон типу T_1

`void sumF(int n, float *F){`
`float sum = 0.0;`
`for (int i = 0; i<n; i++){`
`sum = sum + F[i];`
`}`
`}`
	`void sumF(int n, float *F){`
	`float sum = 0.0; //Ком`
	`for (int i = 0; i<n; i++){`
	`___ sum = sum + F[i];`
	`}`
	`}`

Клон типу T_2	Клон типу T_3

```
void sumI(int n, int *F){
 int sum = 0; //Ком
 for (int i = 0; i<n; i++){
    ___ sum = sum + F[i];
 }
}
```

```
void prodI(int n, int *F){
 int prod = 1; //Ком
 for (int i = 0; i<n; i++){
    ___ prod = prod * F[i];
 }
}
```

Рисунок 1.1 - Приклади трьох типів клонів коду

1.2. Аналіз методів пошуку клонів

1.2.1.Текстовий підхід

Методи на основі текстового підходу розглядають вихідний код програми як послідовність рядків. Для пошуку схожих фрагментів коду використовуються три основних техніки: порядкове порівняння, порівняння під строк,складових так званого відбитку коду, і порівняння перетворених за допомогою мови TXL фрагментів коду.

Спочатку всі файли проекту об’єднуються в один файл. Потім виконується попарне порівняння всіх рядків об’єднаного файлу зі збереженням результатів матриці M (елемент матриці $M[i][j] = 1,$ якщо в об’єднаному файлі рядок i дорівнює рядку j, в іншому випадку $M[i][j] = 0$).

Інструмент Duploc, який представлено на рисунку 1.2вважає клоном послідовність співпадаючих рядків, знайдених на основі отриманої матриці M. Тому він може знайти тільки клони типу T_1.

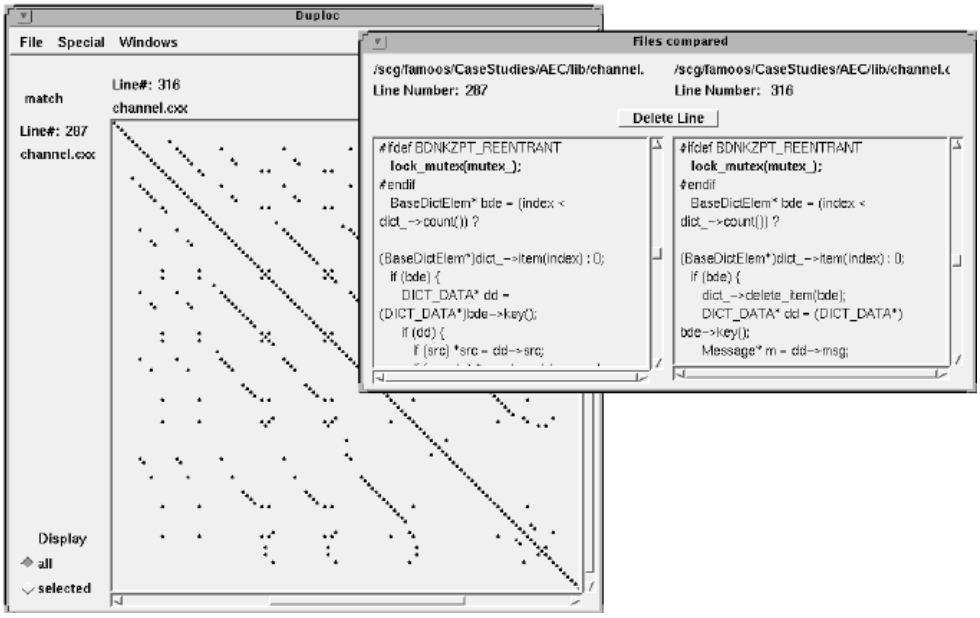

Рисунок 1.2 - Інструмент Duploc

Інструмент DuDe який представлено на рисунку 1.3 використовуючи матрицю M, розширює отримані клони. Якщо для пари клонів (P_1, P_2) існує інша пара клонів (T_1, T_2) така,що відстані між парами фрагментів P_1, T_1 і P_2, T_2 досить малі, то кожна з пар P_1, T_1 і P_2, T_2 об'єднується, причому у фрагмент, отриманий об'єднанням P_i, T_i додаються рядки коду, розташовані між P_i, $T_i i = 1,2$. Такий підхід дозволяє знаходити більше клонів, ніж Duploc. Відмітимо, що інструмент DuDe дозволяє знаходити навіть клони типу T_3 , правда, з невисокою точністю.

Johnson (рисунок 1.4) вводить поняття відбитка (fingerprint) фрагмента коду і замість порядкового порівняння всіх рядків фрагментів коду порівнює їх відбитки. Такий підхід дозволяє швидше проводити аналіз, так як порівнюються тільки підмножини рядків вихідного коду. Варто зазначити, що при такому підході збільшується рівень помилкових спрацьовувань, оскільки у фрагменті коду можуть перебувати рядки коду, що не були обрані для порівняння.

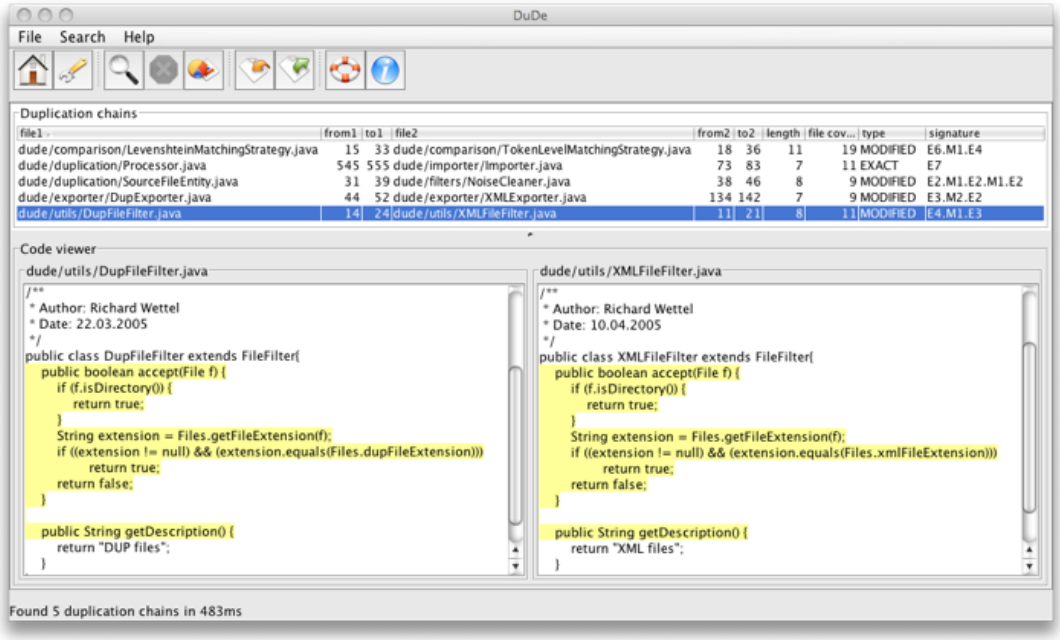

Рисунок 1.3- Інструмент DuDe

Інструмент sif знаходить файли, які мають спільні частини. Для цього вінпорівнює обрані з цих файлів підмножини рядків (опечатки файлів). Напрактиці інструмент показав, що може знайти як точно скопійовані файли,так і файли, у яких загальна частка становить 25%.

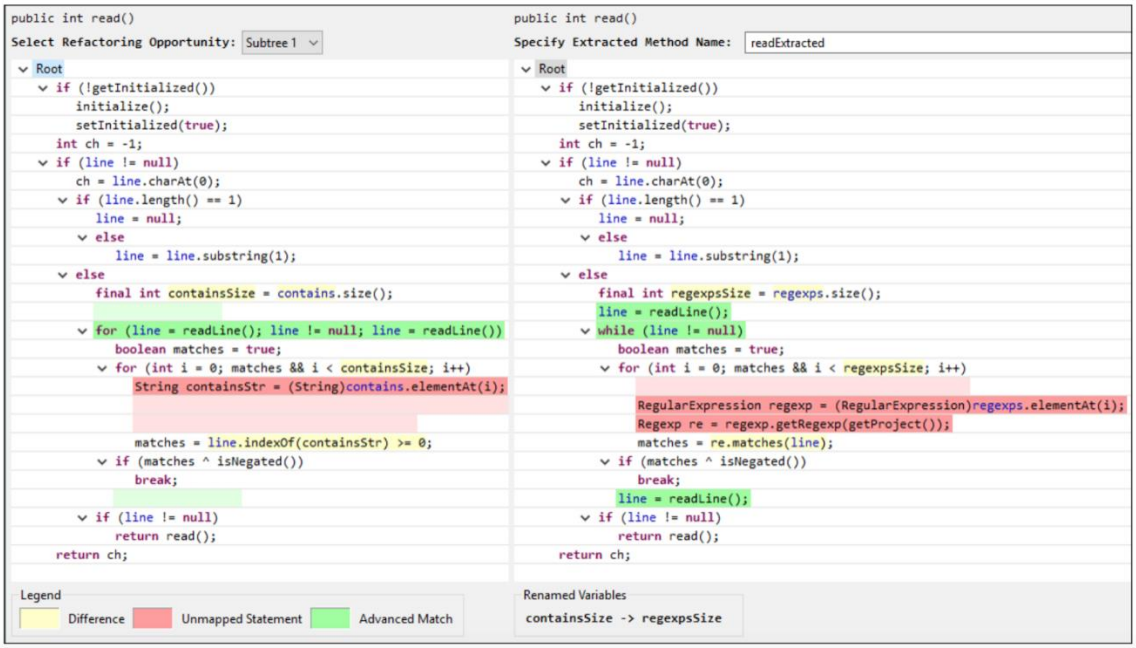

Рисунок 1.4-Інструмент Johnson

1.2.2.Використання мови TXL

Програма на TXL є опис граматичних правил в нормальній формі Бекуса-Наура (BNF) і опис правил переписування термів. В інструментах пошуку клонів TXL використовується для нормалізації фрагментів вихідного коду (нормалізація полягає у виключенні несуттєвих відмінностей у фрагментах коду).

Інструмент NiCad (рисунок 1.5)спочатку нормалізує вихідний код програми звикористанням TXL. Після цього форматує перетворений код таким чином, щоб потенційні зміни розробника після клонування розміщувалися в окремих рядках. Після цього код розбивається на фрагменти,кожен з яких розглядається як потенційний клон. Всі фрагменти попарно порівнюються для знаходження максимально великого загального числа співпадаючих рядків, яке і вважається клоном.

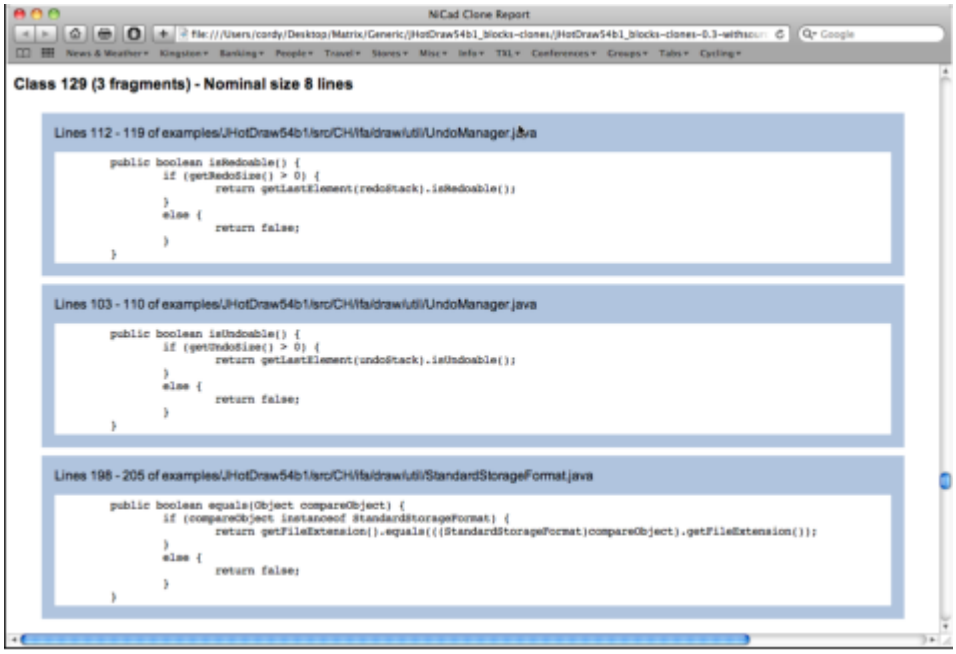

Рисунок 1.5- Інструмент NiCad

1.2.3. Лексичний підхід

За допомогою лексичного аналізу з вихідного коду можна отримати послідовність токенів. Алгоритми пошуку клонів коду шукають збіг

підпослідовності токенів. Вихідний код, яки відповідно збігається з послідовністю токенів, вважається клоном.

Інструмент Dup, який представлено на рисунку 1.6 з послідовності токенів будує суфікс не дерево, на основі якого знаходить співпадаючі максимальні підпослідовності токенів. Він може знаходити тільки клони типів T_1, T_2.

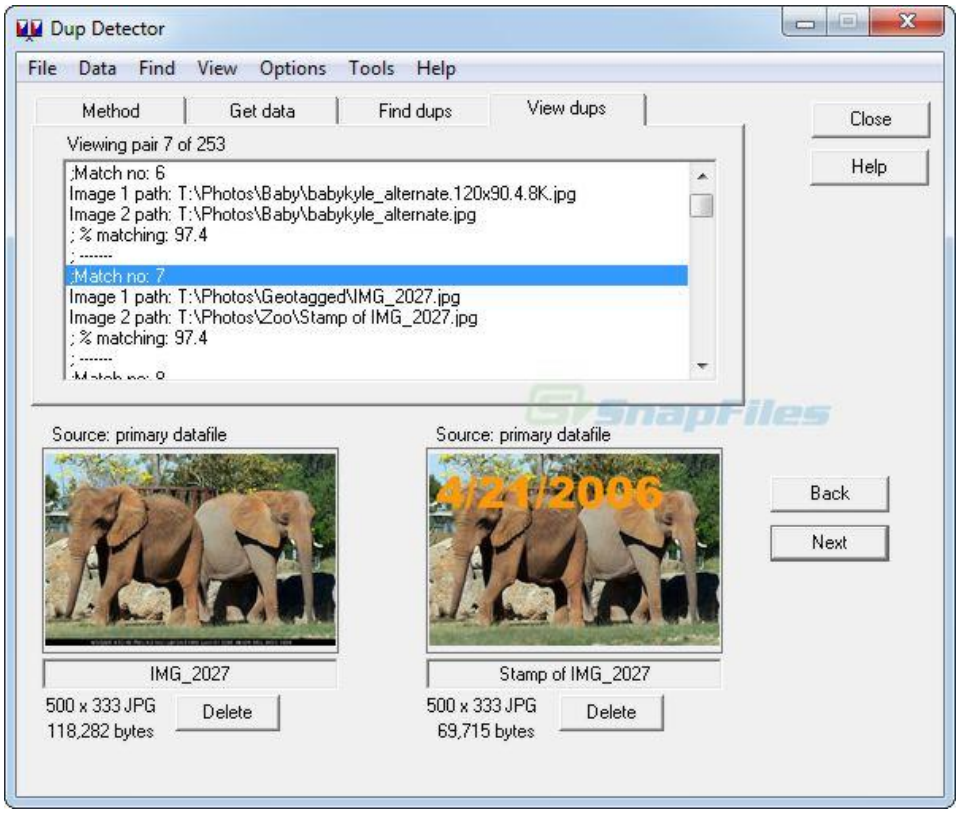

Рисунок 1.6-Інструмент Dup

Одним з найпопулярніших інструментів пошуку клонів коду єCCFinder (рисунок 1.7), який використовує суфікс не дерево для пошуку ідентичнихпідпослідовностей токенів. Якщо розмір аналізованого файлу настільки великий, що файл не поміщається в пам’ять, інструмент аналізує такий файл частинами. Завдяки метрикам оцінки схожості знайдених клонів, CCFinder дає можливість знаходження фрагментів коду, які копіювалися найчастіше.

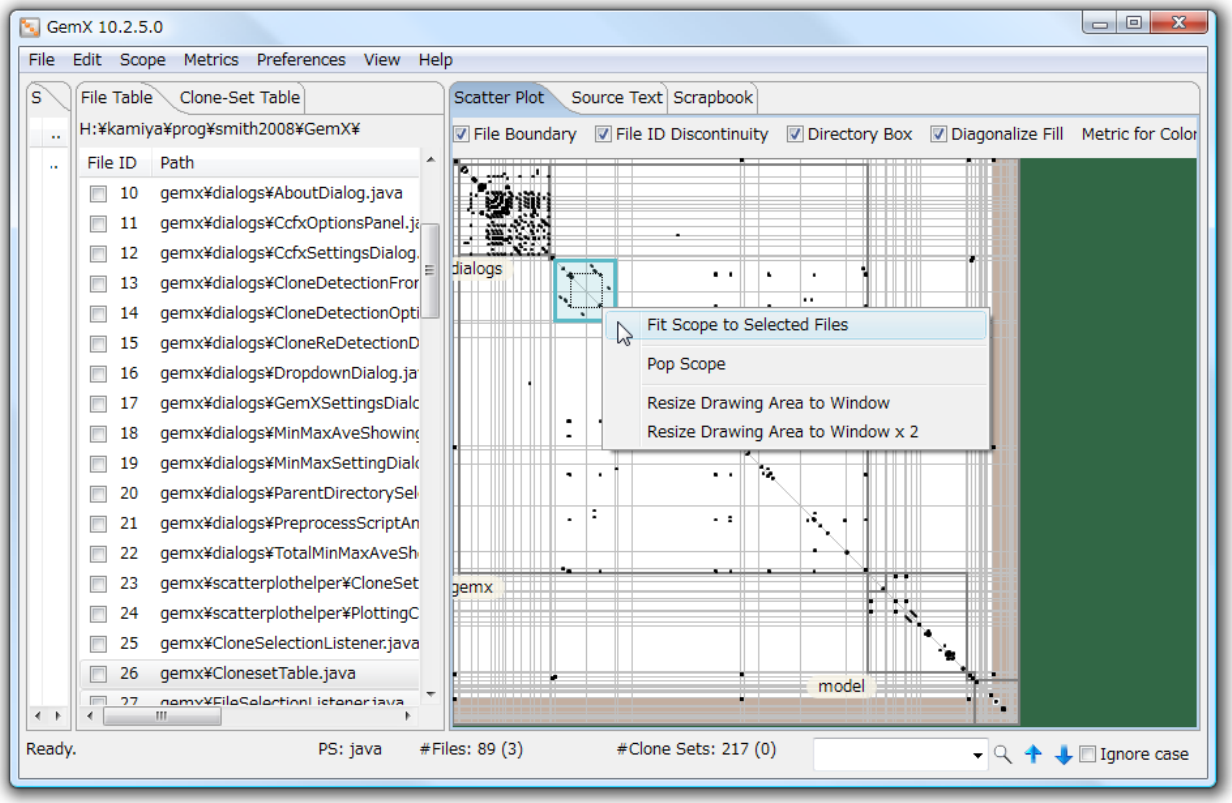

Рисунок 1.7-Інструмент CCFinder

Інструмент CloneDetective (рисунок 1.8) шукає клони коду використовуючи суфіксне дерево. Він дозволяє визначати скопійовані фрагменти коду в яких містяться помилки.

CP-Miner використовує для пошуку клонів методи видобутку даних (datamining). Він визначає повторювані підпослідовності токенів [3] як клони коду. Інструмент знаходить помилки, пов’язані з некоректною адаптацією клонованого фрагмента коду. CP-Miner знаходить тільки клони типів T_1, T_2.У порівнянні з CCFinder він працює повільніше.

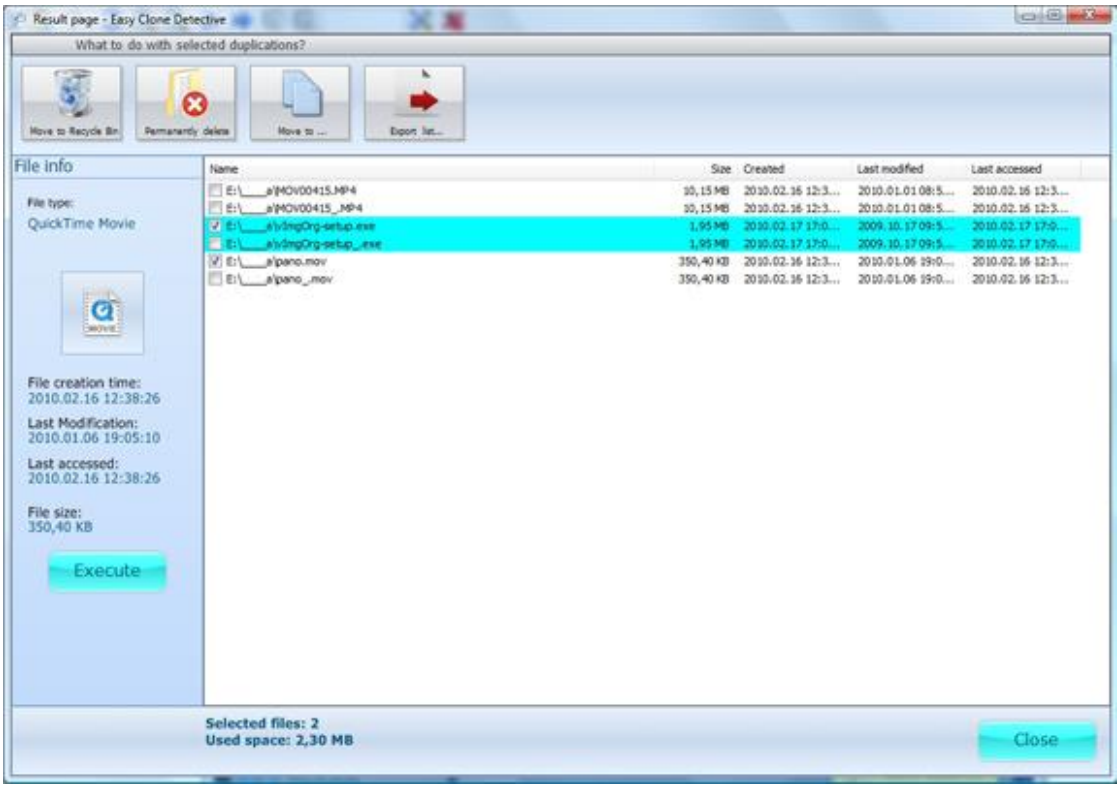

Рисунок 1.8-Інструмент CloneDetective

Інструменти на основі лексичного аналізу можуть знаходити всі клонити пів T_1, T_2 з високою точністю. З них CCFinder маштабується до десятків міліонів рядків вихідного коду. Таким чином, для завдань, в яких потрібно шукати тільки клони типів T_1, T_2, найкращим є лексичний підхід.

Клони типу T_3 виявляються інструментами, що базуються на цьому підході з настільки низькою точністю, що вони непридатні для пошуку таких клонів.

1.2.4.Синтаксичний підхід

Пошук клонів коду здійснюється на основі абстрактного синтаксичного дерева (АСД), іноді використовується АСД в місці дерева розбору (ДР). Ці структури будуються синтаксичним аналізатором, який доступний в багатьох компіляторах.

Інструмент Yang (рисунок 1.9) визначає синтаксичну відмінність двох версій коду. Для обох версій коду він будує АСД, після чого шляхом застосування методів динамічного програмування знаходить пари ізоморфних під дерев.

Вершини, що не входять в такі під дерева, вважаються синтаксичною відмінністю двох функцій. Перевага цього інструменту у порівнянні з linux diff полягає в тому, що він показує не унікальні рядки фрагментів коду, а конкретні інструкції.

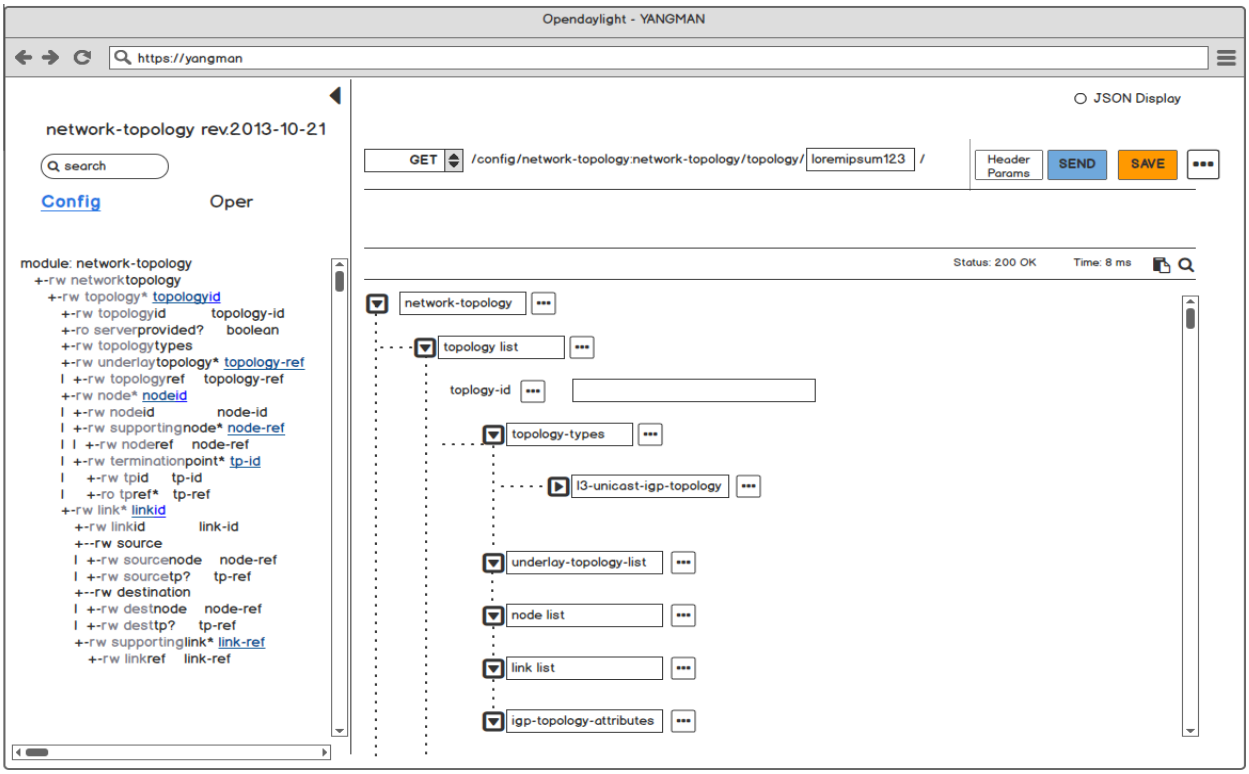

Рисунок 1.9-Інструмент Yang

Інструменти Falke et al і Tairas et al роблять пошук ізоморфних під дерев АСД з використанням суфіксного дерева. Інструмент Falke спочатку знаходить клони типів T_1, T_2, після чого знаходить клони типу T_3, об'єднуючи однакові під дерева АСД. Інструмент Tairaspeалізований як розширення компіляторної інфраструктури Microsoft Phoenix. Ці інструменти мають високу точність тільки при пошуку клонів коду типів T_1, T_2. Вони можуть пропускати клони типу T_2, якщо при адаптаціїскопійованого фрагмента не всі змінні були перейменовані коректно,так як в цьому випадку структура АСД може змінюватися.

1.2.5.Семантичний підхід

Пошук клонів коду здійснюється на основі графа залежностей програми (ГЗП). ГЗП - граф, який об'єднує інформацію про потоки даних і потоки

управління. Вершини ГЗП позначені кодами операцій, аребра - типами залежностей (за даними, або з управління). У ГЗП зберігається вся інформація про семантику і структуру функції, що дозволяє більш точно її аналізувати. Алгоритми, що працюють на основі семантичного підходу,здійснюють пошук схожих, або ізоморфних під графів в кожній парі ГЗП.

Krinke (рисунок 1.10) визначає клони коду як схожі під графи ГЗП. Для пошуку схожих під графів вибираються ідентичні вершини ГЗП, які представляють предикати програми. Ці вершини розглядаються як початкові схожі під графи. Початкові під графи розширюються шляхом додавання нових ідентичних суміжних вершин. Метод реалізований для програм, написаних наANSI-C.

Komondoor et al використовує для пошуку ізоморфних під графів ГЗПтехніку зворотного слайсинга. Після того як ізоморфні під графи знайдені,вони об’єднуються в групи.

Інструмент Higo et al хеше вершини ГЗП (на основі вихідного коду відповідного даній вершині), після чого вибирається пара вершин з однаковими мешами. Прямий і зворотний слайсинг запускаються для обраної пари вершин. В ході слайсинга (slicing) нові вершини додаються ізоморфні підмножини вершин, якщо вони мають однакові хеш-кодування.

Рисунок 1.10-Інструмент Krinke

Horwitz, Krinke, Komondoor et al і Higo et al [6] знаходять всі три типи клонів коду з високою точністю, але мають складність $O(N^3)$, де N -число вершин ГЗП, що не дозволяє використовувати ці інструменти для аналізу програм, що містять мільйони рядків коду.

Дослідження існуючих інструментів пошуку клонів коду на основі семантичного аналізу показали, що вони можуть знаходити всі три типів клонів коду з високою точністю, але, як правило, погано масштабуються.

1.2.6.Підхід на основі метрик

Алгоритми обчислюють ряд метрик для фрагментів коду і порівнюють не фрагменти коду, а вектори отриманих метрик. Зазвичай метрики обчислюються для АСД або ГЗП досліджуваного фрагмента.

Mayrand et al [7] будує за вихідним кодом АСД, для якого обчислює чотири метрики на основі: (1) імен змінних програми; (2) розміщення вихідного коду у файлі; (3) використаних інструкцій; (4) потоку управління програми.

Patenaude et al знаходить клони і фрагменти коду, які можуть мати помилки конструювання. Для порівняння фрагментів коду обчислюються чотири метрики: (1) обчислюється розмір класу, кількість і типи методів класу; (2)відображає зв'язки та ієрархію даного класу в системі; (3) обчислює складність методів класу, при цьому враховується розмір методів і складність McCabe (обчислюється на основі графа потоку управління); (4) відображає ієрархію успадкування. Інструмент працює для програм, написаних на мові Java.

Всі інструменти на основі метрик мають високу продуктивність і масштабуються до мільйонів рядків вихідного коду, але у них низька точність.

1.2.7 Комбінований підхід

Для пошуку клонів в вихідному коді Sutton [7] пропонує застосовувати еволюційний алгоритм (ЕА). Вихідний код поділяється на фрагменти, кожен з яких вважається окремою групою клонів. Завдання ЕКА полягає в мінімізації кількості груп клонів шляхом об'єднання подібних груп. В ході мутації змінюється розмір фрагментів коду з метою збільшення кількості клонів в кожній групі, одночасно скорочується кількість груп.

В роботі [7] викладено метод пошуку клонів коду для мов Java, C #і Ruby з використанням XML представлення програми під назвою PALEX, побудова якого відбувається на першому етапі аналізу. PALEX – представлення типів операцій, з генероване лексичним аналізатором.

Існує три основних типи операцій: зрушення, редукція і читання лексеми. На другому етапі будується суфіксне дерево на основі PALEX.

Клоникодувизначаютьсяякідентичніпослідовностіоперацій.Алгоритмпошук уклонівкоду, описаний в роботі [8],спрямований напошук схожих функцій. Для кожної функції, після лексичного аналізу, отримують послідовність лексем. Подібніділянкирізнихфункційвизначаютьсязадопомогоюсуфіксногодерева.
Наосновіотриманихспівпадаючихпослідовностейлексемкожнафункціяподіляється напідфункції (факторизується). Після факторизації будується граф викликів програми.
Задопомогоютрьохспеціальнихметриквизначаєтьсясхожістьпобудованихграфів.

Інструмент Clone Miner [9], використовуючи RTF [8], отримує групи послідовностей ідентичних лексем, а потім робить додатковий аналіз для побудови множини подібних методів і файлів.

1.2.8 Порівняння розглянутих підходів

З існуючих п'яти основних підходів до пошуку клонів коду текстовий, лексичний, синтаксичний і метричний не можуть досить точно знаходити клони

типу T_3. Текстовий та лексичний підходи не можуть знаходити клони типу T_3, оскільки такі клони розрізняються як вихідним кодом, так і послідовністю лексем. Синтаксичний підхід не знаходить клони типу T_3, тому що при видаленні або додаванні інструкцій може змінитися структура АСД. Метричнийпідхідмаєнизькуточністьприпошукувсіхтрьохтипівклонівкоду. Тільки методи на основі семантичного аналізу вміють знаходити всі клони з досить високою точністю. Але якправило, ці методи мають велику обчислювальну складність і не можуть бути застосовні для аналізу десятків мільйон рядків вихідного коду.

Отже,длястворенняявисокоточногоінструментупошукуклонівкодунеобхіднознизитиобчислювальнускладністьметодівнаосновісемантичногопідходу.

1.3. Постановка задачі дослідження

У роботі запропонований підхід заснований на семантичному аналізі програми. Пошук максимально подібних під графів проводиться в чотири етапи, що дозволяє ефективно і точно вирішувати задачу. Спочатку граф залежностей програми поділяють напідграфи (одиниці порівняння - ОП, що розглядаються як потенційні клони один іншого. Обробка всіх пар одиниць порівняння проводиться в дві фази. На першій фазі за лінійний час відсіваються завідомо несхожі пари одиниць порівняння. Якщо пара одиниць порівняння не була відсіяна, до неї на другий фазі застосовується наближений алгоритм пошуку максимально схожих під графів. Після того як максимальні схожі підграфи знайдені, проводиться фільтрація помилкових спрацьовувань шляхом перевірки рядків вихідного коду, відповідних схожим під графам. Якщо рядки фрагмента вихідного коду, відповідні знайденому підграфу,знаходяться на відстані, меншій p (параметр, заданий користувачем) і довжина фрагмента більша за розмір мінімального клону (задається користувачем), то вони вважаються клонами.

Метою роботи є розробка математичних і програмних засобів для пошуку клонів коду ісемантичних помилок, що виникають при некоректній адаптації скопійованих фрагментів коду.

Виходячи із поставленої мети, у магістерському дослідженні вирішуються такі завдання:

1) проаналізувати недоліки існуючих підходів до пошуку клонів коду ісемантичних помилок, що виникають при неправильному використанніскопійованих фрагментів коду;

2) розробити і реалізувати методи пошуку клонів коду на основісемантичного аналізу програми, що володіють високою точністю і масштабуванням до десятків мільйонів рядків вихідного коду;

3) здійснити програмну реалізацію розроблених математичних методів;

4) провести тестування та апробацію розроблених методів та програмних засобів.

Висновки до першого розділу

1. Проаналізовано переваги і недоліки відомих методів та програмних засобів для визначення клонів коду.

2. Розглянуто процес визначення клонів коду, виділено основні особливості. Показано, що виникає задача оцінки затрат часу та можливості масштабування.

3. Здійснено постановку задачі дослідження, виділено основне завдання магістерського дослідження, а саме розробка метод у пошуку клонів коду на основі семантичного аналізу програм.

2 МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ ПОШУКУ КЛОНІВ КОДУ

2.1. Розробка методів пошуку клонів коду на основі семантичного аналізу

Запропонований підхід заснований на семантичному аналізі програми. Пошук максимально схожих під графів проводиться в чотири етапи, що дозволяє ефективно і точно вирішувати задачу. Спочатку граф залежностей програми поділяються напідграфи (одиниці порівняння - ОП), що розглядаються як потенційні клони один іншого. Обробка всіх пар ОП проводиться в дві фази. На першій фазі за лінійний час відсіваються завідомо несхожі пари ОП. Якщо пара ОП не була відсіяна, до неї на другій фазі застосовується наближений алгоритм пошуку максимально схожих під графів. Після того як максимально схожі під графи знайдені, проводиться фільтрація помилкових спрацьовувань шляхом перевірки рядків вихідного коду, відповідних схожим під графам. Якщо рядки фрагмента вихідного коду, відповідні знайденому підграфу,знаходяться на відстані, меншій (параметр, заданий користувачем) і довжина фрагмента більша за розмір мінімального клону (задається користувачем), то вони вважаються клонами.

Існує кілька методів поділу ГЗП на ОП. Один з методів розділяє граф на слабко зв’язані компоненти [18], які розглядаються як ОП. Недолік цього методу в тому, що розміри фрагментів можуть сильно варіюватися. Фрагмент може бути настільки великим, що він буде містити фрагменти коду, які можуть виявитися клонами. Інший підхід розділяє граф на під графи, будь-які два з яких мають не більше ніж загальних ребер [15].

Цей метод дозволив знайти середньому в два рази більше клонів. Розглянемо алгоритм поділу ГЗП на ОП. Для оцінки якості пропонованого алгоритму були реалізовані методи поділу, які описані в [15] (IST - Interesting Semantic Threads).

Результати тестування на наборі проектів з відкритим вихідним кодом (Linuxkernel, OpenSSL, LLVM) показали, що число знайдених клонівпри використанні запропонованого алгоритму зростає в кілька разів. Ребра ГЗП розглядаються як інтервали. Поділ графу відбувається відповідно тих вершин, які мають мінімальну кількість пересічних ребер. Ефективність даного алгоритму обумовлена такими особливостями поділу ГЗП на під графи:

- вершини отриманих під графів відповідають послідовні рядки вихідного коду;

- кількість рядків коду всіх ОП приблизно однакова;

- кількість ребер між ОП зводиться до мінімуму, що забезпечує максимальну семантичну незалежність отриманих під графів.

Алгоритм отримує на вхід ГЗП, розмір (кількість рядків) вихідного коду,що відповідає одному підграфу, і відсоток, який показує можливу похибку розміру вихідного коду одного підграфу. Алгоритм повертає множину під графів, де кожному під графа відповідає фрагмент коду з послідовними рядками коду.

1. Вхідні параметри: - ГЗП, - розмір вихідного коду, відповідний підграфу, - відсоток можливої варіації.

2. Кожній вершині ГЗП зіставити число, яке показує кількість пересічних ребер в цій точці. Відсортувати вершини графа за відповідними номерами рядків вихідного коду (після сортування вершини знаходяться у векторі). Для кожної вершини розглянути всі вершини ,для яких існує ребро (. У всіх вершин, які знаходяться інтервалі (або , якщо в відсортованому векторі вершина менше ніж , збільшити лічильник перетинів на одиницю.

3. Розглянути всі елементи , що лежать в інтервалі

$$\left[N, N + \frac{N * P}{100}\right]$$

елемент має мінімальне значення лічильника перетинів, вибрати точкою перетину.

4. Елементи вектора до точки перетину додати в окрему множину і видалити з . Зберегти , після чого повторювати крок 3, доки не стане порожнім, або там виявиться менше, ніж елементів. Якщо порожній, перейти

до кроку 5, якщо в виявилося менше, ніж елементів, додати у множину , зберегти , видалити елементи з і перейти до кроку 5.

5. Для всіх збережених множин створити граф . Вершини графа – це елементи множину . Для двох елементів і з в графі буде ребро, якщо існує відповідне ребро в графі .

На рисунку 2.1 ілюструється робота алгоритму після того, як вершини були відсортовані за відповідними рядками вихідного коду.

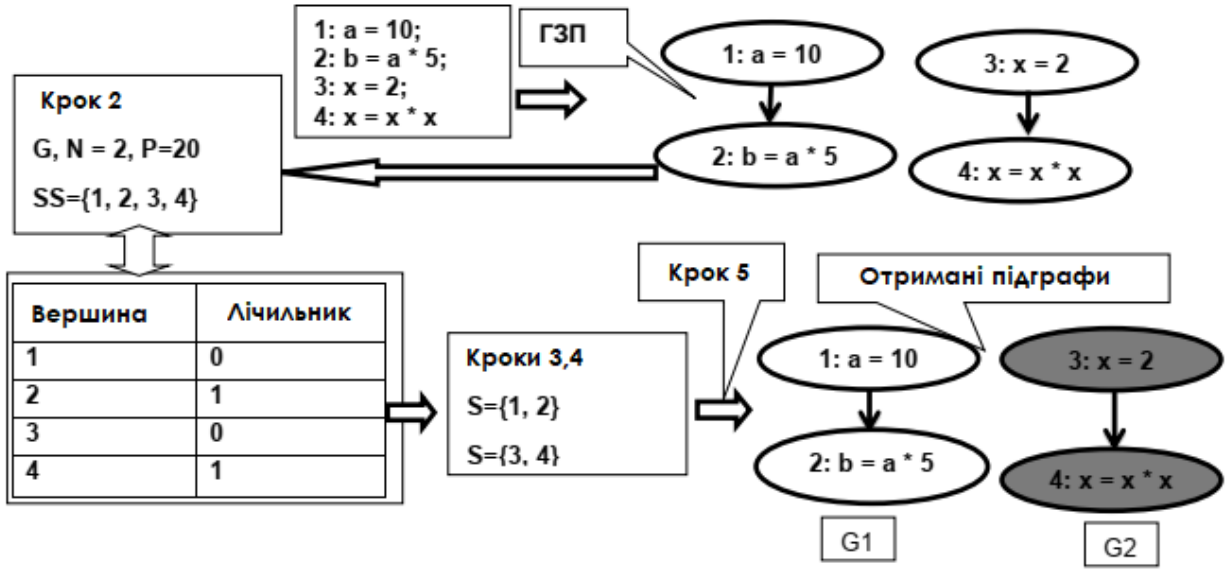

Рисунок 2.1-Приклад роботи алгоритму розподілу графу залежності програми на одиниці порівняння

Складність запропонованого алгоритму $O(n$, де - кількість вершин графа, - середній степінь вершин.

Згідно з дослідженнями [18], клони коду складають не більше 20відсотків коду програмних проектів (добре спроектоване ПЗ має менший відсоток клонів). Перед перевіркою пар ОП на схожість доцільно спочатку відсіяти пари ОП, які свідомо не є клонами. Такий відсів може бути проведено за лінійний час. Кожна вершина графа має мітку (код відповідної операції), у схожих вершин мітки повинні збігатися.

Алгоритми відсіву розглядають наявність у пари ОП достатньої кількості вершин з однаковими мітками. Вихідний код, відповідний таким вершинам, повинен бути розташований в тексті програми таким чином, щоб розмір отриманого фрагмента не був менший за розмір мінімального клону.

Перший алгоритм зберігає мітки вершин ГЗП , в хеш-таблиці відповідно. Якщо

$$|T_1 \cap T_2| \leq k * L,$$

тоді , не можуть мати схожі підграфи бажаного розміру, де розмір мінімального клону, середня кількість вершин ГЗП відповідна одному рядку вихідного коду.

ГЗП має обмежену кількість різних типів вершин (мітки вершин)характеристичний вектор ГЗП - це спеціальний вектор довжини , де - кількість різних типів вершин ГЗП. Кожен елемент вектора – це кількість вершин в ГЗП, що мають спеціальний тип (наприклад, кількість вершин, відповідних операцій додавання).

Другий алгоритм обчислює характеристичний вектор для кожного ГЗП. Якщо евклідова відстань між двома векторами більше, ніж задане число (користувач задає степінь схожості шуканих клонів $sim \in$, на основі цього визначається $d = 1 -$, тоді відповідні ГЗП не містять бажані схожі під графи.

Складність цих алгоритмів лінійна від кількості вершин,відповідної пари ОП.

Розглянемо алгоритм пошуку схожих підграфів максимального розміру в парі ГЗП / ОП, заснований на слайсингу. Цей алгоритм застосовується для пари ОП, якщо за допомогою алгоритму перевірки не доведено, що дана пара не може бути клоном.

Алгоритм пошуку схожих під графів: запропонований алгоритм є наближеним бо завдання пошуку максимальних схожих/ізоморфних під графів складна. Прямий / зворотний слайсинг і покрокова фільтрація незбіжних вершин працюють на основі даного алгоритму. Спочатку для кожної вершини першого графа вибирається максимально схожа з нею вершина з другого графа. Після цього всі пари вершин вважаються схожими під графами і розширюються шляхом додавання відповідних інцидентних вершин. Алгоритм повертає найбільшу пару під графів,отриманий при розширенні як максимальні схожі. Треба відзначити, що запропонований алгоритм є наближенням і не гарантує максимальність знайдених схожих під графів.

1. Вхідні параметри: графи , . Вихідні параметри: множини вершин з , , де і множина вершин схожих під графів максимального розміру.

2. Побудувати множину пар вершин . Пара ⟨, де з , з , увійдеш тільки в тому випадку, якщо для вершини вершина така, що ⟨має найбільшу кількість ідентичних сусідніх вершин. Якщо пара ⟨ увійшла в , то і не братимуть участі в створенні нових пар.

3. Для даної пари ⟨ з створити порожні множини і і додати в , в .

4. Для всіх пар вершин ⟨v, де з , з , що не були розглянуті в 4 і 5 кроці, щоб переглянути, чи є вони ідентичними(чи збігаються мітки). Якщо так, перейти до кроку 5. Якщо немає, пропустити дану пару. Якщо в ході даного кроку не була знайдена пара нерозглянутих ідентичних вершин, перейти до кроку 6.

5. Позначити як розглянуті. Знайти множини всіх нерозглянутих інцидентних вершин . Побудувати перетин множин цих вершин(критерієм рівності елементів вважається збіг відміток).

Вершини, що потрапили в перетин, додати в і відповідно. Перейти до кроку 4.

6. Запам’ятати пару (,). Повторити крок 3 для нерозглянутих пар з . Всіх збережених пар (,) повернути пару максимального розміру. Розміром пари (,) вважається кількість елементів найменшої множини.

Даний алгоритм має перевагу в порівнянні з відомим слайсингом: одночасно використовуються прямий і зворотний слайсинг, що дозволяє знайти деякі ідентичні графи, які неможливо знайти,використовуючи тільки прямий або тільки зворотний слайсинг.

На рисунку 2.2 показана робота алгоритму для однієї пари вершин (кроки 3, 4, 5, 6).

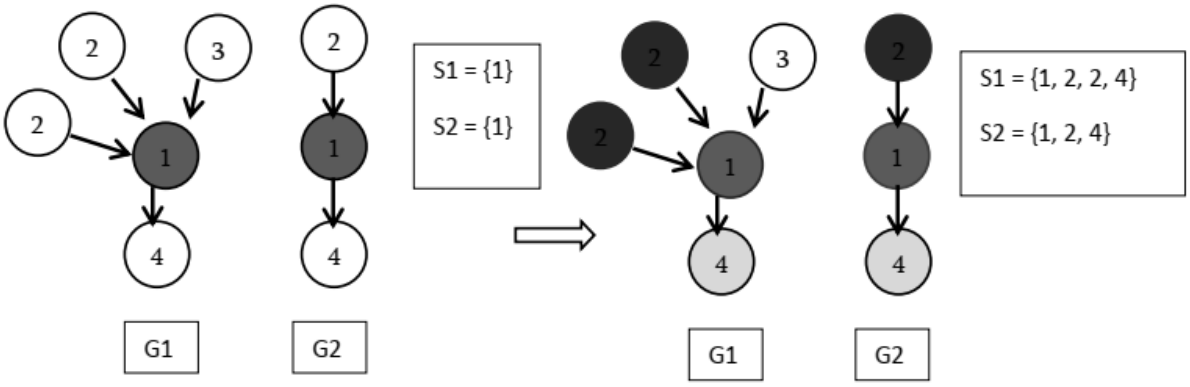

Рисунок 2.2-Приклад роботи алгоритму: вершини з міткою «2» знайдено при зворотному слайсингу вершин з міткою «1»

Складність цього алгоритму $O(nm(nlog(n)+mlog(m))$, де n і m– кількість вершин в першому і в другому графі відповідно.

2.2. Пошук схожих під графів на основі ізоморфізму дерев

Пошуку клонів коду на основі ізоморфізму дерев складається з двох основних етапів. Найпершому етапі ГЗП перетворюється в дерево (рисунок 2.3), при перетворенні додаються нові ребра і вершини, щодаєможливістьзберегтимаксимальнукількістьінформаціїпропервиннийграф. Надругомуетапіпроводитьсяпошукмаксимальнихізоморфнихпіддерев,якірозгляда ютьсяякклоникоду.

Такийпідхіддозволяєзастосуватиточніалгоритмипошукумаксимальнихізоморфних піддерев, що дозволяє швидше шукати клони типівT_1 і T_2 в порівнянні з слайсингом (у цього алгоритмі нижча обчислювальна складність, ніж у слайсингу). Для завдань, де треба знайти тільки клони типів T_1 і T_2 найкраще підходить цей алгоритм.

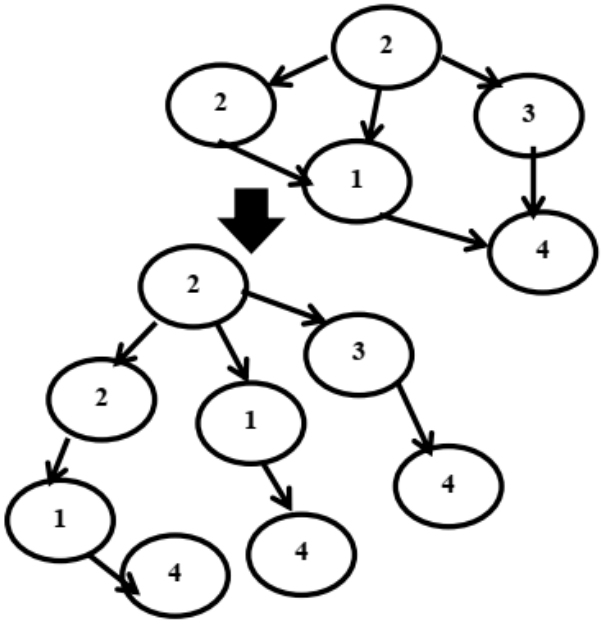

Рисунок 2.3-Перетворення ГЗП в дерево

Перетворення ГЗП в дерево в своючергуділитьсянадвіфази.Спочаткувиконуєтьсявидаленняназадспрямованихребер і топологічне сортуванняГЗП. Сортування починається з початкової вершини (вершина, у якої немає вхідних ребер, кожен ГЗП має таку вершину). Потім в зворотному порядку розглядаються рівні вершин, отримані в результаті сортування. Вершини, у яких більше одного вхідного ребра, перетворюються наступним чином.

Припустимо, щоV - це вершина, у якій є вхідні ребра від вершин $W_1 \dots W_n$. Ребра(W_i, V) відсортовані відповідно мітках (тип операції відповідної інструкції) $(W_i$ ($(W_n$ - максимальний).

Алгоритм нумерує вершини пар отриманих дерев. В ході нумерації ізоморфні вершини отримують однакові номери.

Алгоритм:

1. На вхід отримуємо пару дерев T_1 і T_2, повертаємо множину вершині з аморфних під дерев.

2. Вершини першого графа нумеруються значеннями 0, а вершини другого - 1, і дерева об’єднуються. Коріння дерев додаються в чергу.

3. Береться перший елемент з черги, і для нього генерується номер. Якщо обраний елемент не є листом, його наступники додаються в чергу. Процес повторюється, поки черга не стане порожньою.

4. Всі вершини дерева T_1, для яких є вершина з однаковим номером зT_2 додаються в множинуF. Аналогічним чином всі вершини дерева T_2, для яких є вершина з однаковим номером з T_1 додаються в множинуS.

У множинах F і S містяться вершини ізоморфних під дерев T_1 і T_2.

Генератор номерів:

1. На вхід отримує вершину U. Алгоритм використовує глобальний лічильник C для нумерації.

2. Для U створюється вектор Vec, що містить мітку U і номер попередникаU (якщо є попередник). Якщо такий вектор вже існує, в таблиці векторів $tabVec$ переходимо до кроку 3, якщо ні - до кроку 4.

3. Номеру U присвоюється відповідне значення лічильника Vec з $tabVec$. Uдодається в вектор $Isomorphic\,[U]$ ($Isomorphic$ є вектором векторів вершин дерева).

4. У $tabVec$ додається вектор Vec і поточне значення лічильника C. номером Uприсвоюється значення C. U додається в вектор $Isomorphic\,[U]$. значення Cзбільшується на одиницю.

Складність цього алгоритму $O(n+m)^2 + log(n+m))$, де n і m– кількість вершин в першому і в другому дереві відповідно.

На рисунку 2.4 показана робота алгоритму після кроку 2. Вхідні дерева T_1 і T_2 розмічені і об’єднані в одне, після чого кореневі вершини додані в чергу вершин.

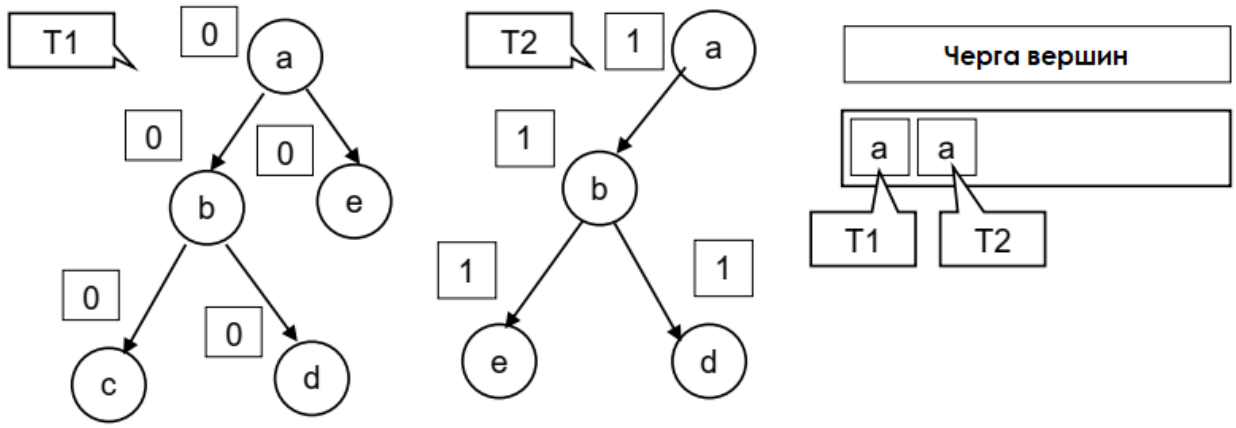

Рисунок 2.4- Об'єднане розмічене дерево

На рисунку 2.5 показана нумерація вершини «а» дерева T_1.

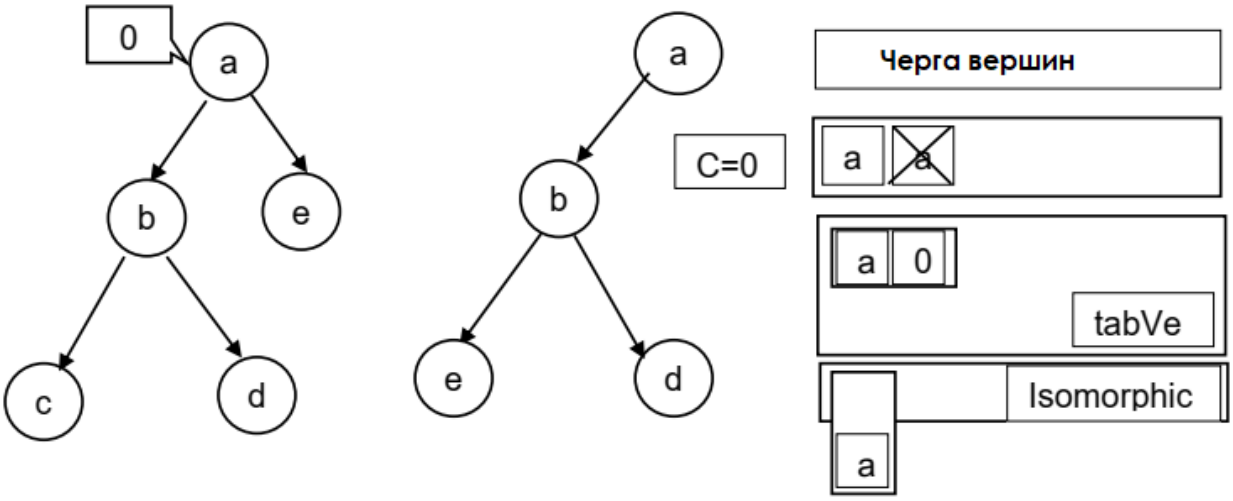

Рисунок 2.5-Нумерація вершин

Рисунок 2.6 показує дерево після повної нумерації. Як видно з рисунка,ізоморфні вершини отримують однакові номери.

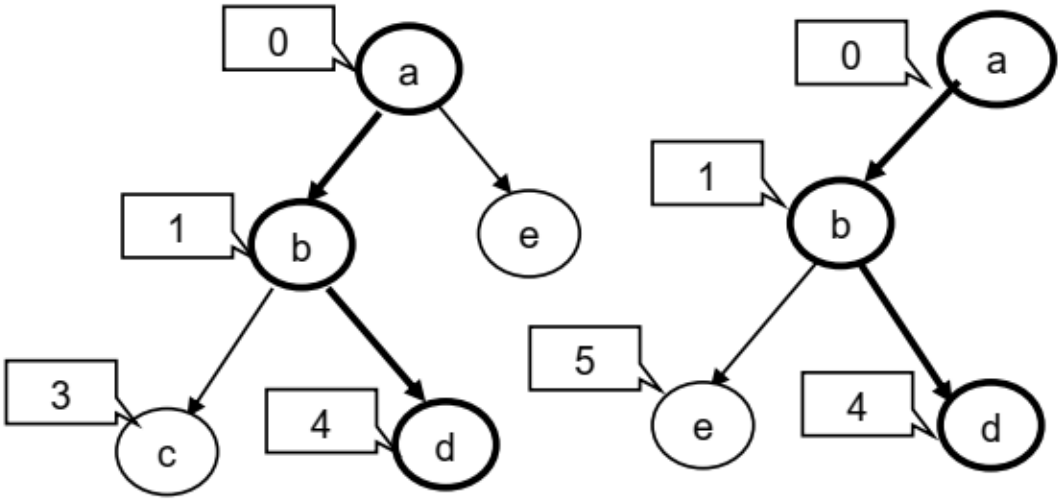

Рисунок 2.6-Ізоморфні дерева з однаковими номерами

Найбільшою точністю володіє алгоритм пошуку максимально схожих під графів на основі слайсингу. Він дозволяє знаходити фрагменти коду, в яких були зроблені суттєві зміни. Висока точність досягається завдяки тому, що алгоритм повністю враховує семантику клонованого і модифікованого фрагментів коду. Схожі під графи,які були знайдені в ході роботи алгоритму, часто виходять ізоморфними або мають великі ізоморфні під графи.

2.3. Апробація методів пошуку клонів

Комп’ютер, на якому проводилося тестування - Intel Core i3 CPU 540 з8ГБ оперативної пам'яті. Для тестування було обрано такі проекти з відкритими початковими кодами: Linux, Firefox Mozilla, LLVM/Clang iOpenSSL. Linux є операційною системою сімейства Unix. Firefox Mozillaє браузерним движком, розробленим компанією Mozilla. LLVM/Clang -компіляторна інфраструктура. В даний момент вона широко використовується як для оптимізації, так і для різних аналізів програм. OpenSSL є криптографічного бібліотекою, яка підтримує безпеку на рівні транспортування і сокетів (Transport Layer Security - TLS і Secure Sockets Layer- SSL). Вона розроблена на основі бібліотеки SSLeay.

Рисунок 2.7 показує кількість рядків вихідного коду проаналізованих проектів. Ядро Linux містить приблизно 13.9мільйона рядків вихідного коду. Firefox Mozilla, LLVM / Clang і OpenSSL містять відповідно 5.1, 1.4 і 0.27 мільйона рядків коду, написаних на C/C ++.

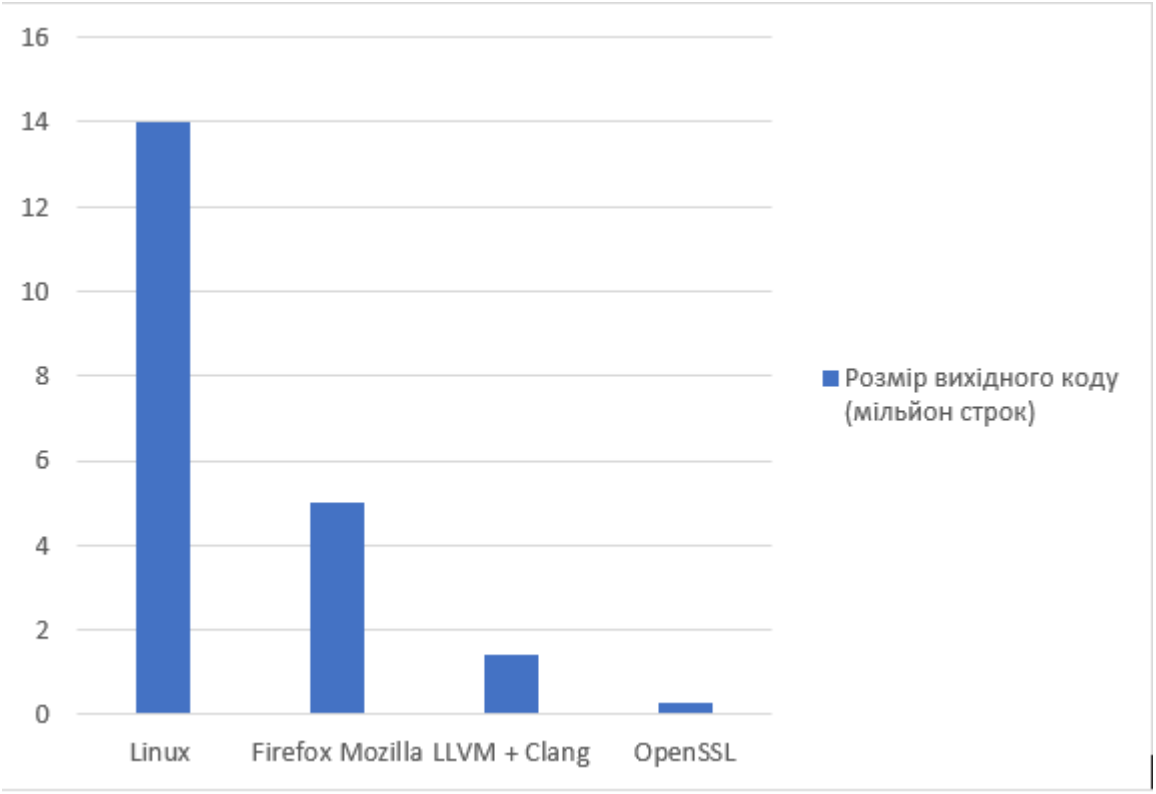

Рисунок 2.7-Кількість рядків вихідного коду

На рисунку 2.8 наводиться порівняння часу компіляції з урахуванням генерації ГЗП. Час компіляції ядра Linux збільшується з 1.3 до 2 години,що становить близько 50%. Для інших проектів погіршення складають менше 50%.

Рисунок 2.9 показує розмір отриманих ГЗП. Розмір ГЗП для ядра Linux складає 439 мегабайт. Для Firefox Mozilla, LLVM/Clang і OpenSSL розмір становить 329, 200 і 12 мегабайт відповідно.

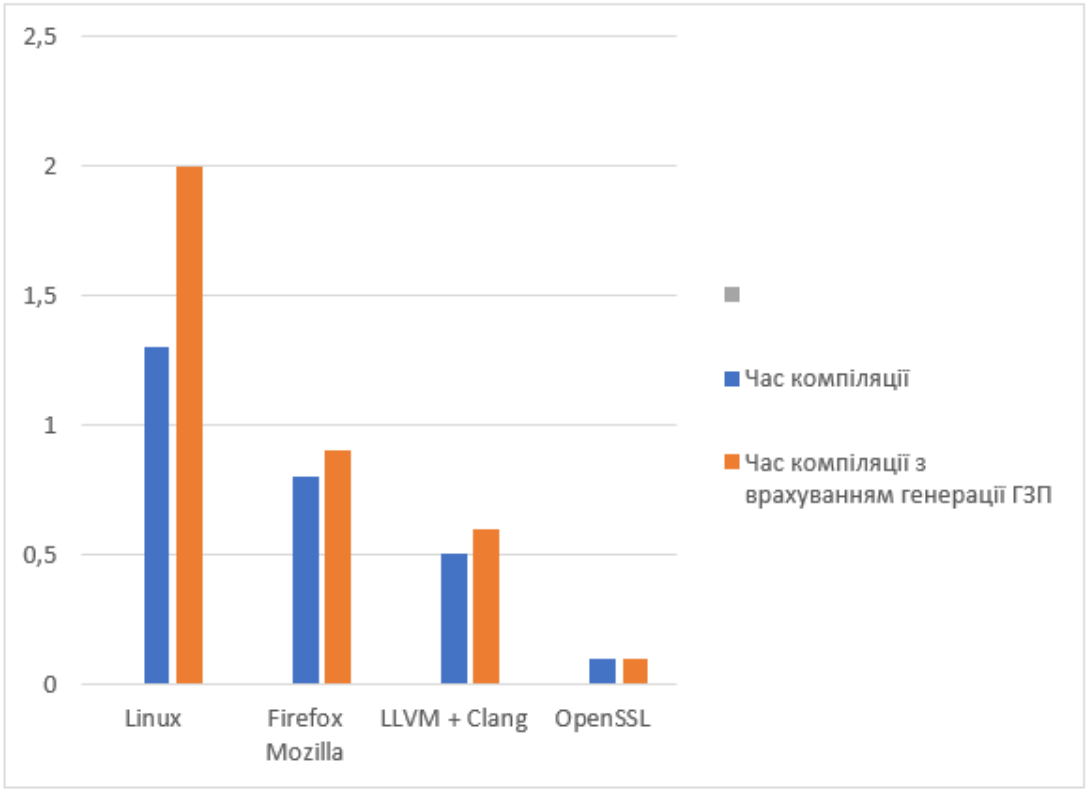

Рисунок 2.8- Порівняння часу компіляції

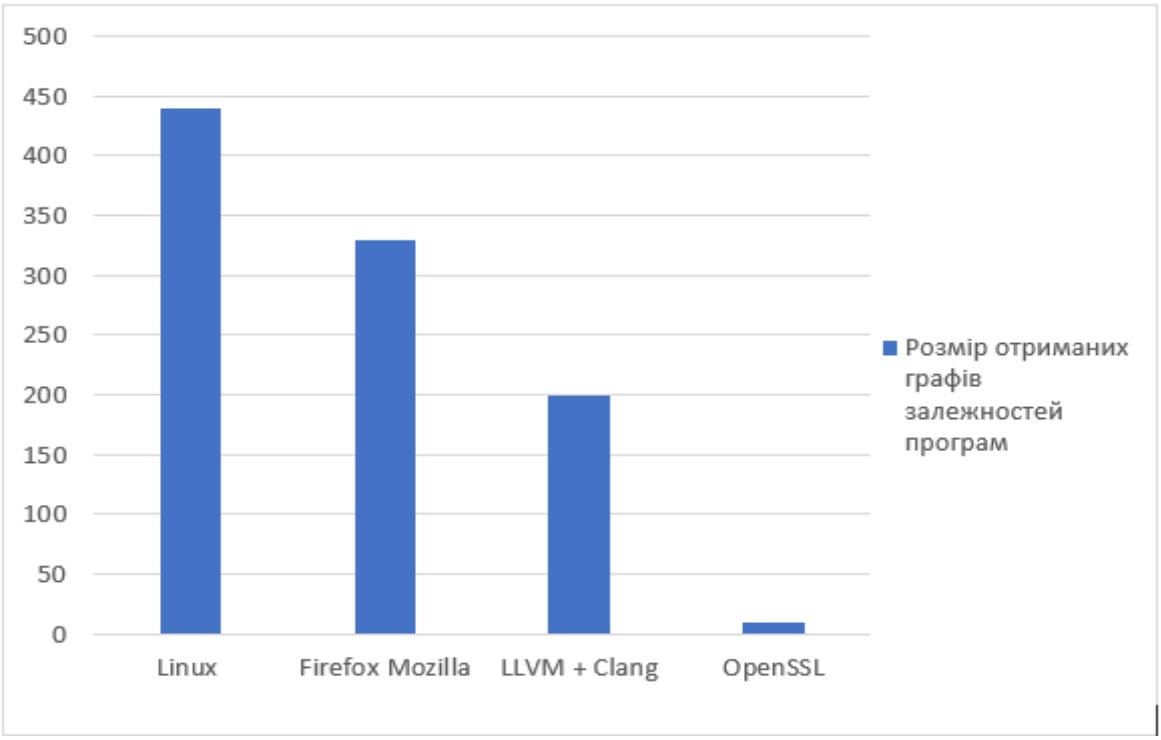

Рисунок 2.9-Розмір отриманих графів залежностей програм

Розмір ОП налаштовується двома параметрами: мінімальним розміром ОП (25 рядків вихідного коду) і верхньою межею варіації ОП (100%), тобто розмір ОП варіюється в інтервалі [25,50] (рисунок 2.10). Зазначені параметри обрані для тестування і пошуку найбільш цікавих клонів. Рисунок2.10 показує кількість отриманих ОП після поділу ГЗП. Для Linux, Firefox Mozilla,LLVM / Clang і OpenSSL кількість ГЗП виростає з 33369, 39325, 7884 і 1054відповідно до 40257, 48921, 11776 і 1588. У середньому, кількість ГЗП збільшується на 25%.

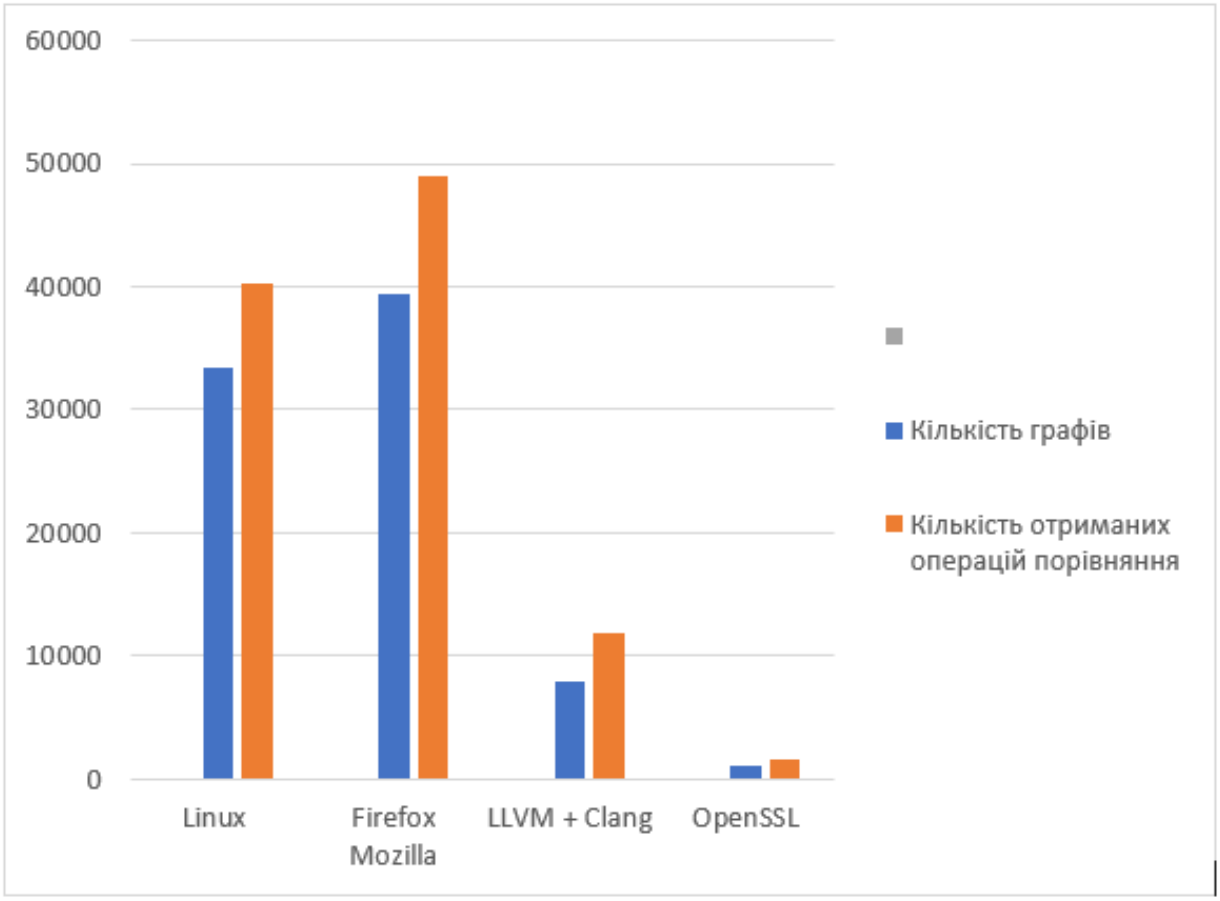

Рисунок 2.10-Кількість отриманих операцій порівняння

На рисунку 2.11 показано час роботи алгоритму, що розділяє ГЗП на ОП. Поділ ГЗП на ОП для Linux займає лише 72 секунди. Для FirefoxMozilla, LLVM/Clang і OpenSSL 46, 26, 2.3 секунди відповідно.

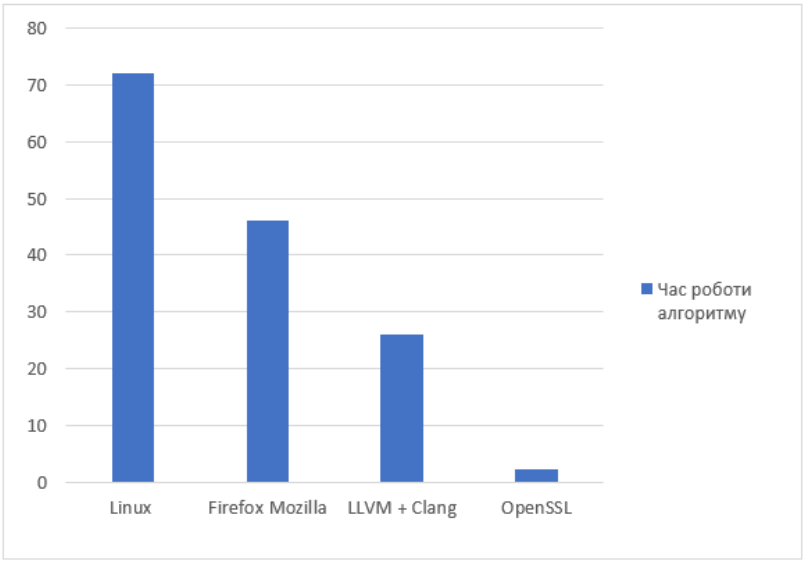

Рисунок 2.11-Час роботи алгоритму, що розділяє ГЗП на ОП

На рисунку 2.12 наводиться порівняння знайдених клонів коду в залежності від алгоритму. Завдяки застосуванню розробленого алгоритму, кількість знайдених клонів коду для ядра Linux зросла з 913(алгоритм IST) до 1965. Для проектів Firefox Mozilla, LLVM / Clang і OpenSSL кількість знайдених клонів зросла відповідно з 485, 62, 19 до 708, 134,50. При застосуванні алгоритму WCC кількість знайдених клонів коду для Linux, Firefox Mozilla, LLVM / Clang і OpenSSL становить628, 351, 31 і 17 відповідно.

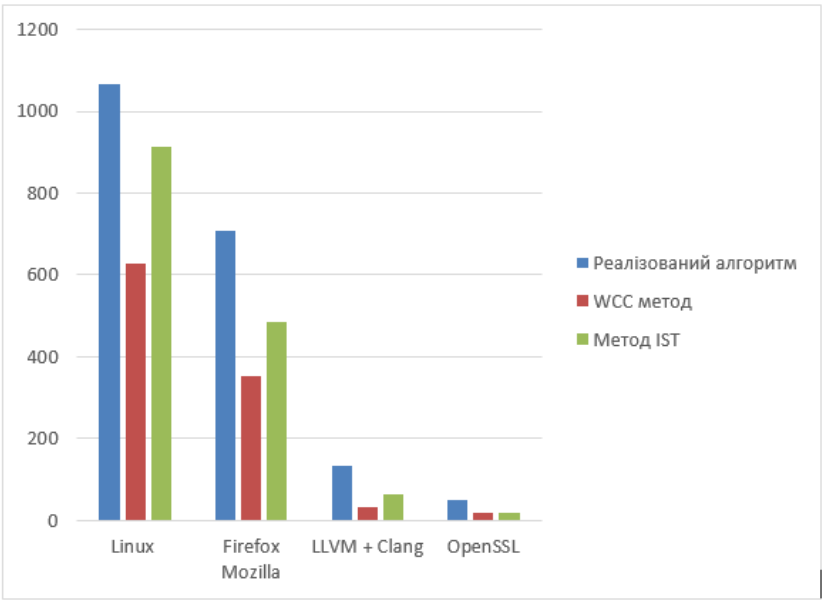

Рисунок 2.12-Порівняння алгоритмів

На рисунку 2.13 показано час роботи алгоритму. Розмір мінімального клону - 25 рядків, схожість клонів - більше 90 відсотків. Ядро Linux аналізується за 34.43 години.

Час аналізу Firefox Mozilla, LLVM / Clang iOpenSSL становить 15.81, 3.52 та 0.023 години відповідно.

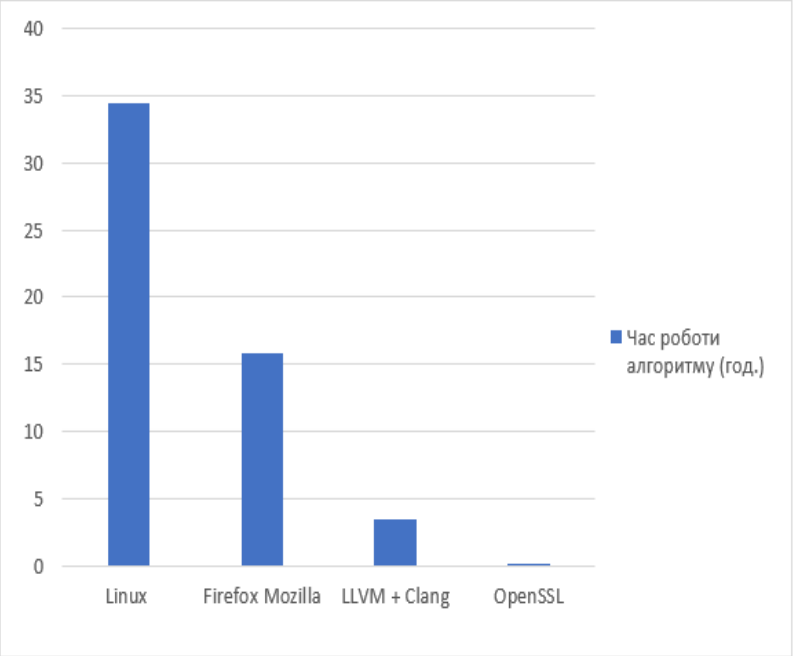

Рисунок 2.13-Час роботи алгоритмів пошуку клонів коду

Для ядра Linux з 1965 знайдених клонів тільки 73 виявилися помилковими. Для проектів Firefox Mozilla, LLVM/Clang і OpenSSL з знайдених708, 134 і 50 клонів помилковими виявилися відповідно 41, 6 і 3 (рисунок 2.14).

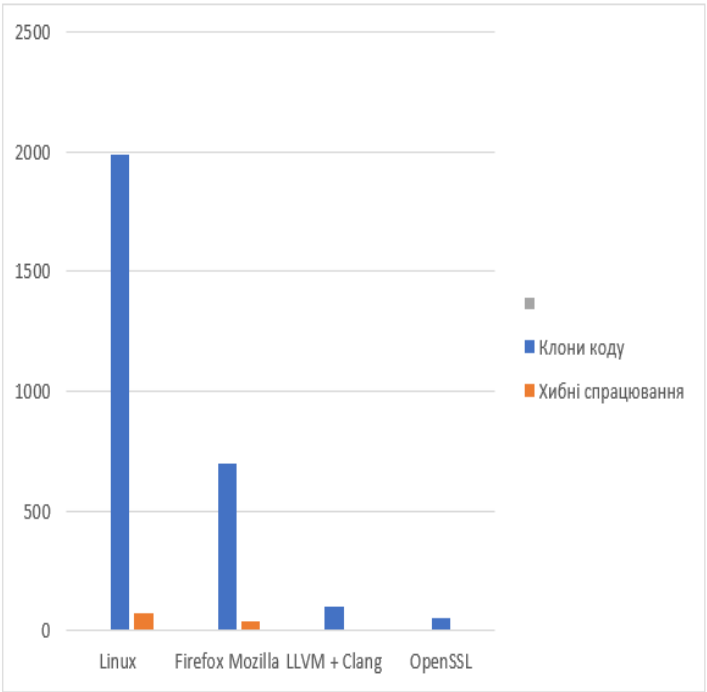

Рис. 2.14.Кількість знайдених клонів і рівень помилкових спрацьовувань

У таблиці 2.1 наведено приклад знайденого клону для проекту Firefox Mozilla.

Таблиця 2.1

Приклад знайденого клону

	Клон 1	**Клон 2**
firefox-29.0.1	**mozilla-release/intl/icu/source/common/ucnv_u16.c** **59: length=(int32_t)(pArgs->sourceLimit-source);** **60: if(length<=0) {** **61: /* no input, nothing to do */** **62: return;** **63: }** **64:** **65: cnv=pArgs->converter;** **66:** **67: /* write the BOM if necessary */** **68: if(cnv->fromUnicodeStatus ==** **UCNV_NEED_TO_WRITE_BOM) {** **69: static const char bom[]={ (char)0xfe, (char)0xff };** **70: ucnv_fromUWriteBytes(cnv,** **71: bom, 2,** **72: &pArgs->target, pArgs->targetLimit,** **73: &pArgs->offsets, -1,** **74: pErrorCode);** **75: cnv->fromUnicodeStatus=0;** **76: }** **77:** **78: target=pArgs->target;** **79: if(target >= pArgs->targetLimit) {** **80:** ***pErrorCode=U_BUFFER_OVERFLOW_ERROR;** **81: return;**	**mozilla-release/intl/icu/source/common/ucnv_u16.c** **658: length=(int32_t)(pArgs->sourceLimit-source);** **659: if(length<=0) {** **660: /* no input, nothing to do */** **661: return;** **662: }** **663:** **664: cnv=pArgs->converter;** **665:** **666: /* write the BOM if necessary */** **667: if(cnv->fromUnicodeStatus ==** **UCNV_NEED_TO_WRITE_BOM) {** **668: static const char bom[]={ (char)0xff, (char)0xfe };** **669: ucnv_fromUWriteBytes(cnv,** **670: bom, 2,** **671: &pArgs->target, pArgs->targetLimit,** **672: &pArgs->offsets, -1,** **673: pErrorCode);** **674: cnv->fromUnicodeStatus=0;** **675: }** **676:** **677: target=pArgs->target;** **678: if(target >= pArgs->targetLimit) {** **679:** ***pErrorCode=U_BUFFER_OVERFLOW_ERROR;** **680: return;**

Висновки до друго розділу

1. Розроблено метод пошуку клонів коду на основі семантичного аналізу, який складається з чотирьох основних етапів, а саме поділ графу залежностей програм на одиниці порівняння; фільтрування несхожих пар одиниць порівняння; пошук максимальних схожих під графів; фільтрація помилкових спрацьовувань.

2. Здійснено процедуру тестування та апробації запропонованого методу пошуку клонів, яка підтвердила її ефективність.

3 ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ ПОШУКУ КЛОНІВ КОДУ

3.1. Архітектура та функціональність системи

Для побудови інструменту пошуку клонів коду програм в якості бази обраний LLVM [9]. LLVM є компіляторною інфраструктурою з відкритим вихідним кодом на мові C++. В рамках цього проекту представлений інструментарій для розробки ПЗ. LLVM (рисунок 3.1) містить статичний компілятор, компонувальник, віртуальну машину і JIT-компілятор, які працюють над єдиним внутрішнім представлення програми (біт код). Біткод може бути представлений трьома різними формами: у текстовому вигляді; у вигляді структур даних в оперативній пам’яті; в двійковому вигляді. Інфраструктура LLVM дає можливість зберегти біт код в проміжних об’єктних файлах для подальшої оптимізації, в тому числі динамічної. Усі надані LLV Мможливості по обробці внутрішнього представлення (в тому числі різні аналізи, перетворення і т.д.) можуть бути застосовні до біткоду.

Тому компіляторна інфраструктура LLVM є зручною платформою для семантичного аналізу програми. Використовуючи аналізи LLVM з біткода, можна отримати інформацію про потік даних і про потік управління. З біткода також можна отримати інформацію про номери рядків вихідного коду, з якого він був побудований, якщо є налагоджувальна інформація (при компіляції дана опція «-ggdb»). Основними особливостями LLVM є: реалізація на Сі ++;модульна і розширювана архітектура; статичний компілятор з можливістю динамічної компіляції біткода.

Для LLVM є кілька компіляторів переднього плану: C, C ++,Objective-C (Clang, GCC / dragonegg).

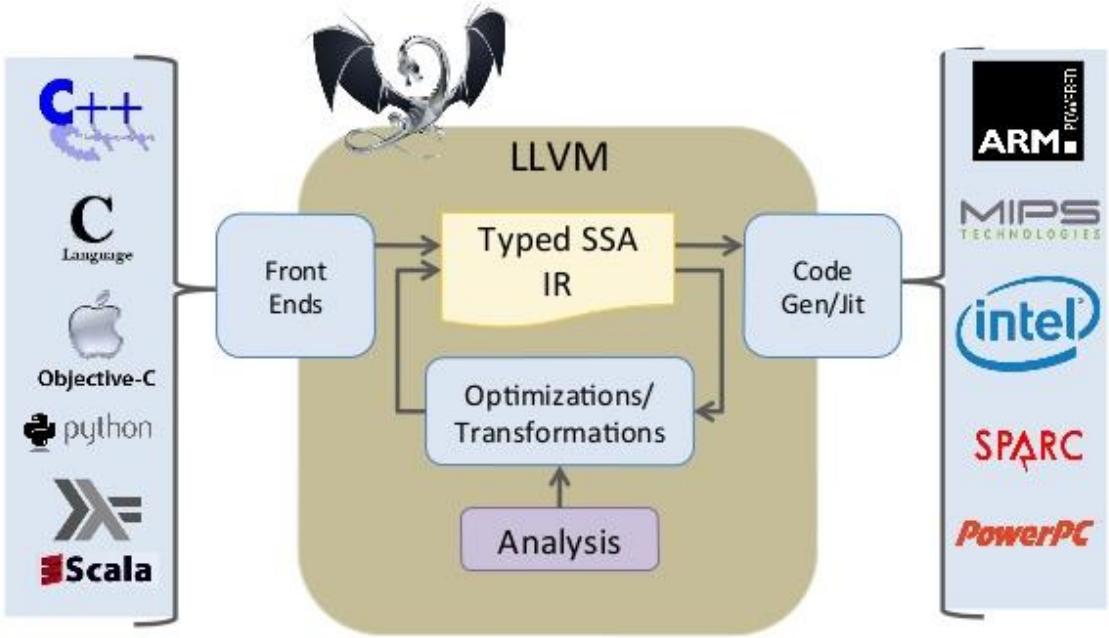

Рисунок 3.1- Структура LLVM (Low Level Virtual Machine)

Генерація ГЗП проекту проводиться на основі біткода під час компіляції проекту. Пошук клонів коду проводиться окремо. Для цього реалізований окремий інструмент, який на вхід отримує директорію з ГЗП графами і здійснює пошук клонів. Він також містить систему автоматичної генерації клонів коду, для аналізу і поліпшення розроблених алгоритмів. Реалізований інструмент підтримує паралельну обробку для багатоядерних систем. Одночасно можуть бути оброблені кілька параграфів для визначення клонів коду.

Наша модель інструменту пошуку клонів коду заснована на семантичному підході, так як мета інструменту - знайти всі клони з найбільшою точністю. Інструмент також повинен бути масштабованим для аналізу проектів з вихідним кодом в кілька мільйонів рядків. На рисунку 3.2 представлено діаграму класів розробленого інструменту. Він складається з двох основних частин. Перша частина створює ГЗП-граф з внутрішнього представлення LLVM під час компіляції проекту (рисунок 3.3). Цей етап розроблений як прохід компіляторної інфраструктури LLVM. Друга частина інструменту відповідає за аналіз ГЗП-графів з метою знаходження клонів.

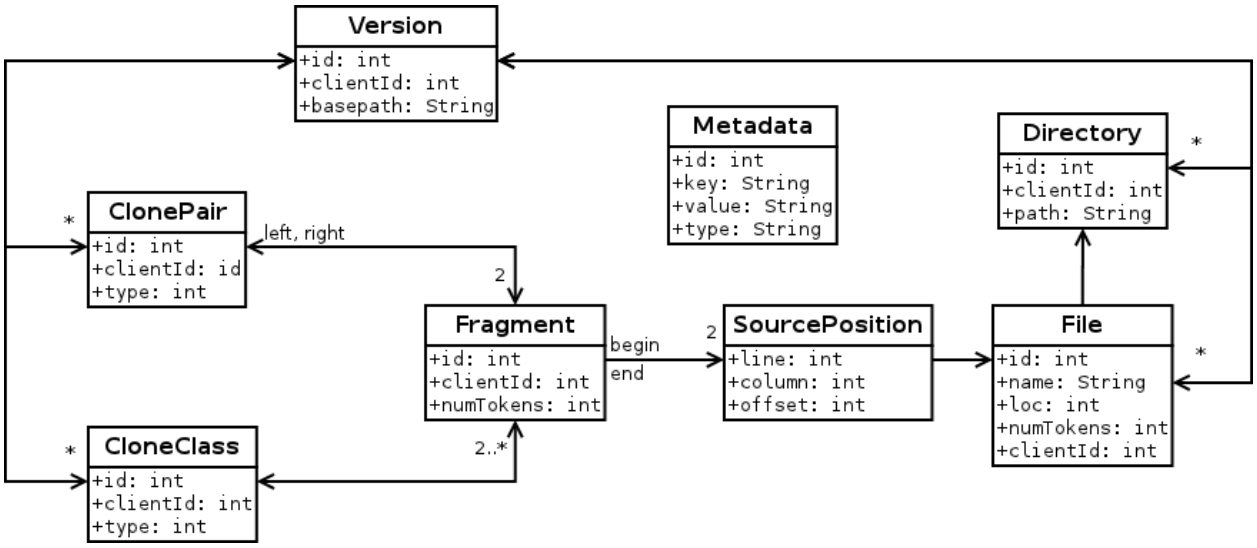

Рисунок 3.2-Діаграма класів інструменту пошуку клонів коду

Пошук клонів коду проводиться в чотири етапи: поділ ГЗП на одиниці порівняння; фільтрування несхожих пар одиниць порівняння; пошук максимальних схожих під графів; фільтрація помилкових спрацьовувань. Друга частина розроблена як інструмент пакета LLVM.

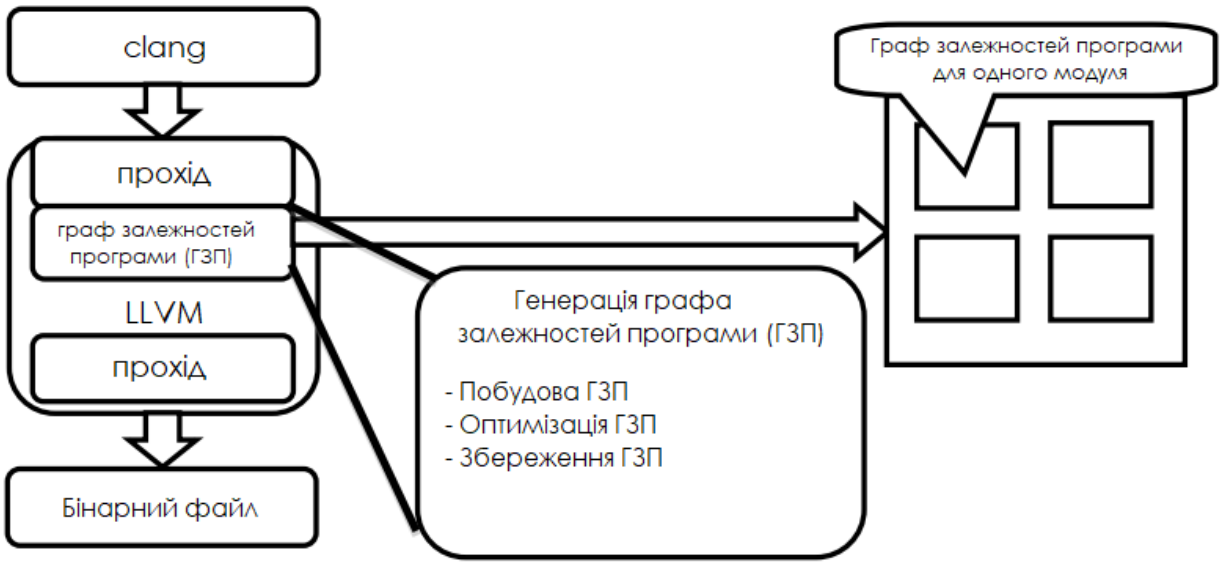

Рисунок 3.3-Аргітектура додатку (генерація графу залежностей програми)

Генерація ГЗП забезпечується компіляторним проходом LLVM на основі проміжного представлення LLVM (біт код). Вершинами графа є інструкції біткода, ребра виходять шляхом аналізу потоку даних і потоку управління (рисунок 3.4).

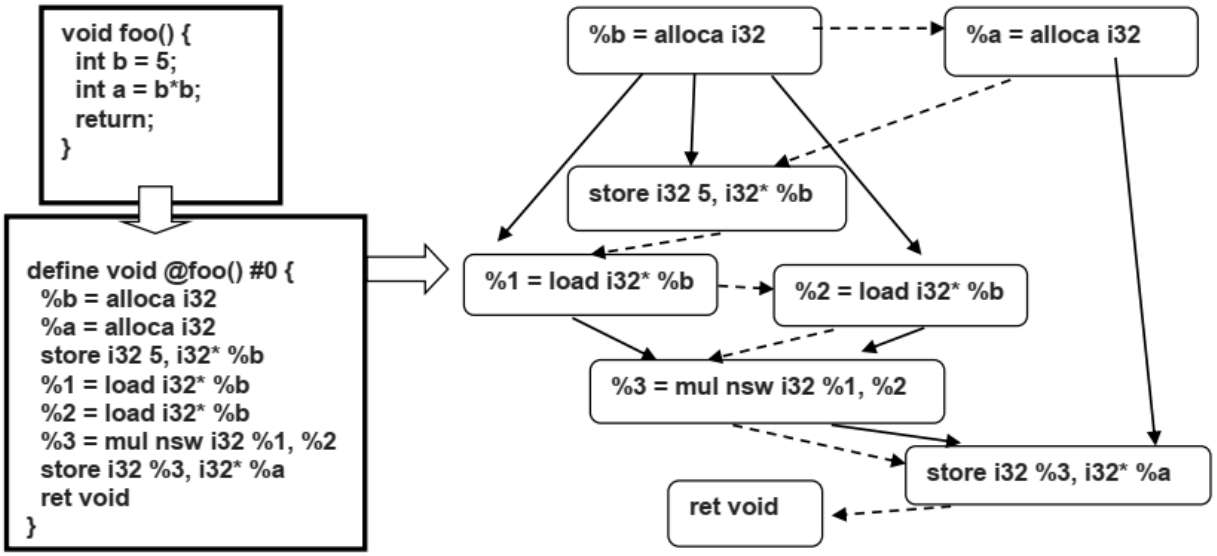

Рисунок 3.4-Приклад графу залежностей програми

Інструмент забезпечує генерацію ГЗП з трьома різними рівнями деталізації, що дозволяє ефективно шукати клони конкретного типу. У графі першого рівня знаходиться тільки ребра, отримані LLVM use-def аналізом.

Граф другого рівня містить також ребра, отримані з використанням аналізу аліасів. Максимальна кількість інформації є в графі третього рівня деталізації: в ньому є всі ребра першого і другого рівнів, а також ребра, що відображають залежності з управління. У завданнях, де потрібно швидко знайти тільки клони типів T_1 і T_2, використовується граф першого або другого рівня. Для ефективного пошуку клонів типу T_3 необхідно використовувати граф третього рівня, що знижує швидкість роботи.

За замовчуванням генерується ГЗП першого рівня (тільки на основі LLVMuse-def аналізу). Для генерації графів другого і третього рівнів деталізації передбачені окремі опції користувача.

Інструмент дозволяє групувати вершини ГЗП за операціями і типами змінних. Існує три типи групування вершин ГЗП:

1. Вершини ГЗП, яким відповідають логічні операції, отримують однакові мітки (код операції відповідної інструкції).

2. Вершини ГЗП, яким відповідають арифметичні операції, отримують однакові мітки.

3. Вершини ГЗП, яким відповідають змінні програми, отримують однакові мітки, незалежно від типів цих змінних.

При логічному групуванню вершини ГЗП, що відповідають різним логічним операціям, розглядаються як ідентичні. Якщо в клонованій ділянці коду одна логічна операція була змінена на іншу, інструмент вважатиме, що ці фрагменти повністю збігаються. У разі групування за типом, зміна типів змінних не впливатиме на ступінь схожості фрагментів коду. Для кожного типу групування вершин ГЗП передбачені окремі опції користувача.

Після того як ГЗП будуть побудовані, вони оптимізуються і зберігаються в файлах. Під оптимізація ми ГЗП маються на увазі такі операції:

1. Видалення вершин, які не мають ніяких ребер.

2. Видалення вершин, яким не відповідає вихідний код. Такі вершини можуть виникнути через те, що біт код LLVM представляється у вигляді SSA форми.

Можливий пошук клонів коду в рамках декількох проектів, для цього необхідно тільки помістити ГЗП цих проектів в одну директорію і запустити інструмент для них.

Після завантаження ГЗП в пам’ять, вони поділяються на одиниці порівняння (рисунок 3.5). Всі пари одиниць порівняння розглядаються як потенційні клони один одного. Поділ графа на одиниці порівняння має проводитися акуратно, щоб реальні клони виявилися в окремих одиницях порівняння. В іншому випадку в одиницю порівняння потрапить тільки частина реального клону, внаслідок чого клон знайдеться частково або взагалі не буде знайдений. Завдання полягає в тому, щоб в результаті кожна одиниця порівняння повністю містить потенційний клон.

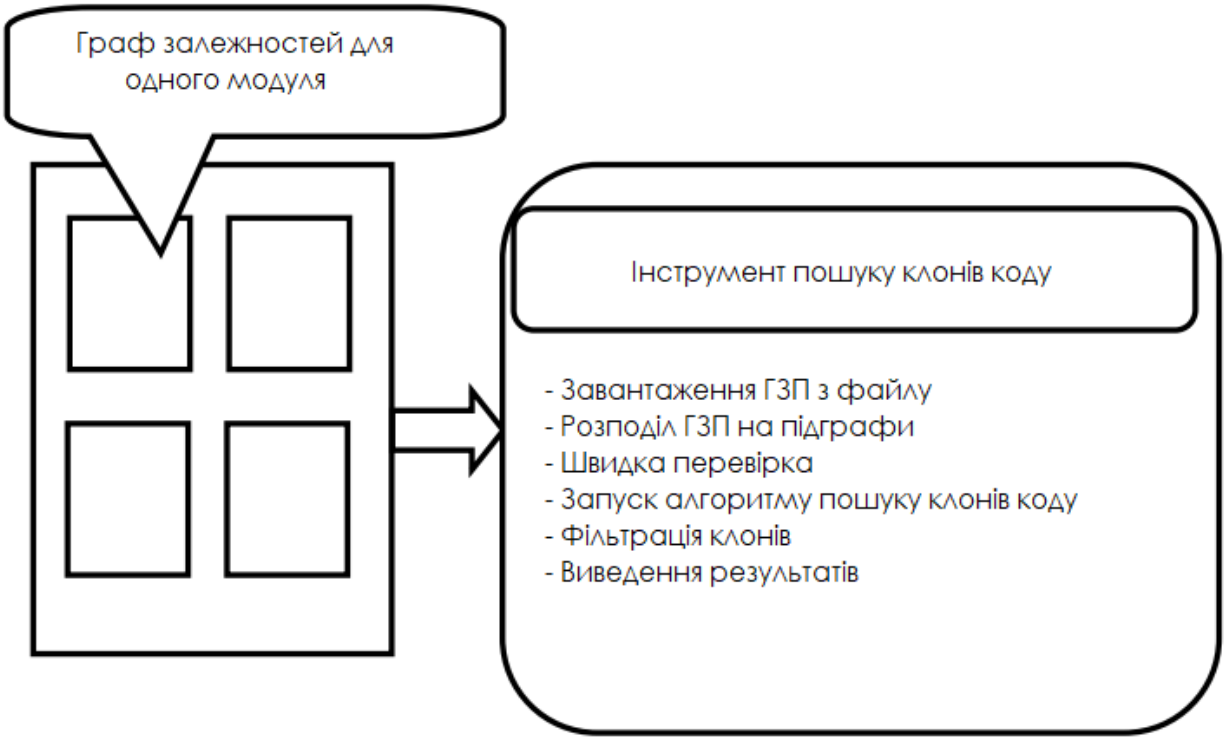

Рисунок 3.5-Приклад графу залежностей програми

Після поділу ГЗП на ОП буде розпочато процес їх порівняння з метою знаходження клонів. Інструмент містить два типи алгоритмів порівняння. Перший тип алгоритмів перевіряє пару одиниць порівняння на те, що вони не є клонами.

Складність таких алгоритмів - лінійна, залежить від кількості вершин в парі ОП. Другий тип - це наближені алгоритми пошуку схожих підграфів. Обчислювальна складність цих алгоритмів досить велика, вона може досягати кубічного ступеня від вершин графа. Так як більшість пар одиниць порівняння не є клонами, для них недоцільно буде застосовувати алгоритми другого типу. Таким чином, спочатку будуть запущені алгоритми першого типу, які за лінійний час доведуть, що більшість пар не є клонами.

Останній крок у процесі пошуку клонів коду - фільтрація помилкових спрацьовувань. Знайдені пари ізоморфних під графів додатково перевіряються алгоритмами фільтрації. Необхідність застосування фільтра виникає через те, що ми визначаємо поняття клону для вихідного коду програми, а шукаємо клони як ізоморфні під графи. Виходить, що клон повинен бути якоюсь послідовністю рядків у файлі (не обов’язково наступних один за одним, але обов’язково розташованих на незначній відстані). Мета фільтрації полягає в тому, щоб перевірити, що рядки фрагмента коду, відповідають схожим під графам,

знаходяться досить близько один одному. Цю перевірку потрібно зробити після того, як схожі під графи знайдені; в іншому випадку, алгоритм пошуку клонів може пропустити деякі клони коду.

3.2. Інтегрована система тестування

В проекті реалізована окрема опція, яка дозволяє запустити інструмент в режимі від лагодження. У цьому випадку автоматично генерується база клонів для даного проекту, і для генерованих клонів запускаються реалізовані алгоритми пошуку клонів коду. Точність розробленого алгоритму визначається кількістю знайдених клонів з бази. Запропоновано два підходи до автоматичної генерації клонів коду.

Перший підхід (рисунок 3.6) використовує існуючі стандартні проходи LLVM, а також спеціальні проходи. Інструмент в режимі тестування для кожної функції отримує два графи:

- перший, оригінальний, виходить на основі вихідного проміжного представлення LLVM; другий граф - на основі перетвореного проміжного представлення. Алгоритм пошуку клонів коду запускається для оригінального і перетвореного графа. Так як перетворення LLVM не змінюють семантику програми, отримані графи повинні бути знайдені як клони.

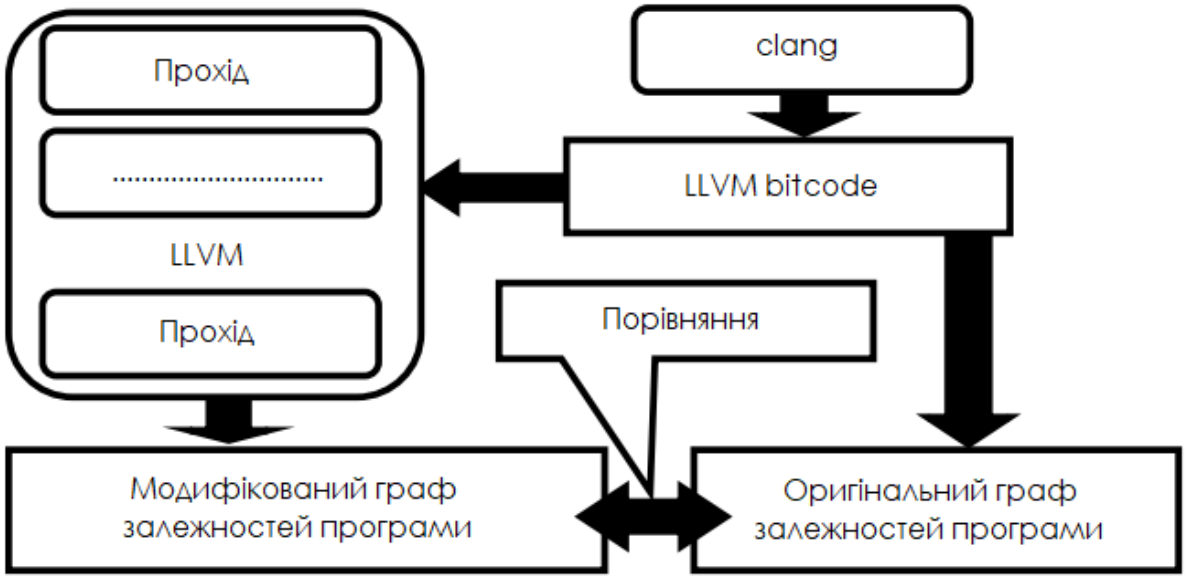

Рисунок 3.6-Схема генерації клонів коду

Розглянемо вплив перетворення викликів функцій на ГЗП. Мета даного перетворення створити функцію-перехідник для виклику функції. Виклик потрібної функції в переходнику визначається на основі важко обчислювального предиката. Виклик початкової функції в коді замінюється на виклик перехідника. Таке перетворення майже не зачіпає потік управління і даних початкової функції. Тому відповідні ГЗП оригінальної і перетвореної функції мають незначні відмінності, що дозволяє запропонованим алгоритмам з легкістю знайти з генеровані клони коду.

Базовим блокам функції присвоюються номери. На початку функції вставляється блок «диспетчер». Послідовності виконання базових блоків визначається диспетчером на основі інформації переданої їй попереднім виконаним блоком. Це перетворення зачіпає в основному потік управління і додає нові інструкції. ГЗП перетвореної функції містить додаткові вершини і ребра з управління. Нові ребра за даними існують тільки між новими інструкціями. Таким чином, ГЗП оригінальної функції міститься в графі перетвореної функції як ізомерний під граф (можливо, за винятком деяких ребер управління). Такі клони коду знаходяться при використанні алгоритмів на основі слайсингу та ізоморфізму дерев.

Під час компіляції все константні рядки, крім тих, що містяться в агрегатних типах (масиви, контейнери зі стандартної бібліотеки), шифрують ся. Створюються спеціальні функції: шифратор і дешифратор. Перед кожним

використанням рядка викликається дешифратор, а після шифрувальна функція. Це перетворення в основному зачіпає потік даних. ГЗП перетворюваної функції містить додаткові вершини,відповідні викликам шифрувальника і дешифратора. Також містить нові ребра за даними між парами вершин, відповідні інструкції константного рядка і виклику шифрувальника / дешифратора. В цьому випадку ГЗП оригінальної функції міститься в ГЗП як ізомерний під граф.

У графі потоку управління утворюються несвідомі ділянки шляхом додавання «фіктивних» ребер з заголовків циклів в їх тіла. Для додавання таких ребер генеруються непрозорі предикати, на основі яких додаються фіктивні ребра. Таке перетворення в більшій частині зачіпає потік управління. ГЗП перетворюваної функції містить нові вершини, відповідні доданим непрозорим предикатам і ребра управління. Нові ребра за даними можуть бути додані тільки між вершинами, відповідним непрозорим предикатам. Запропонований метод знаходять перетворювану функцію як клон оригіналу.

Для всіх циклів функції додаються «фіктивні» ребра з одного в інший. Фіктивні ребра виходять на основі непрозорих предикатів. Таке перетворення в основному впливає на потік з управління. Але також може зачіпати потік даних в залежності від використаних непрозорих предикатів. ГЗП перетворюваної функції відрізняється від оригінального як ребрами потоку даних і управління, так і новими вершинами. Тільки алгоритм на основі слайсингу визначає, що перетворювана функція є клоном оригіналу.

Переплетення функцій складається з чотирьох основних етапів. На першому кроці об’єднуються сигнатури двох функцій і генерується додатковий параметр, на основі якого буде працювати код потрібної функції. На другому кроці об’єднуються базові блоки обох функцій в переплетену, створюється предикат на основі з генерованого додаткового параметра, який вирішує, код якої функції повинен працювати. На третьому етапі генеруються спеціальні базові блоки, які будуть працювати при виконанні коду обох функцій. На останньому етапі виклик переплетених функцій замінюється на виклик створеної функції. Таке перетворення значно змінює ребра потоку даних і управління. Додає множину нових вершин. ГЗП оригінальної і перетворюваної функції мають сильні відмінності. Тільки алгоритм на основі слайсингу здатний виявити такі клони.

Розглянемо вплив ускладнення аналізу потоку даних на ГЗП. Локальні змінні переносяться в глобальну область видимості. Проводиться спеціальний аналіз графа викликів функцій, який дозволяє визначити, зміну глобальної змінної в якій функція не буде впливати на коректну роботу програми. Після цього в кожній функції здійснюються деякі обчислення з тими глобальними змінними, яків даному випадку можна «зіпсувати». Таке перетворення впливає на потоки даних і управління. Основна структура ГЗП не змінюється. Запропонований метод [10] знаходить такі клони. Алгоритми на основі метрик та ізоморфізму дерев мають порівняно низьку точність і можуть пропускати деякі клони даного виду.

Позитивні константи в коді програми розбиваються на складові довільним чином. Замість використання оригінальної константи використовується сума розбитих чисел. Дане перетворення в основному стосується потоку даних, також додаються нові інструкції. Зазвичай ГЗП перетворюваної функцій не сильно відрізняється від оригінальної ГЗП. Запропоновані в роботі алгоритми знаходить такі клони з досить високою точністю.

Для кожного виклику функції генерується копія тіла функції, після чого кожен виклик оригінальної функції замінюється на виклик відповідної копії. Це перетворення не стосується потоку даних і управління, тому ГЗП копіює функції ізоморфні один одному.

В ході перетворення вибираються послідовність базових блоків, де перший блок має одне вхідне ребро, а останній - одне надзвичайне. Для створення помилкового циклу додається «фіктивне» ребро з останнього блоку в перший базовий блок.

Це перетворення в основному впливає на потік управління і додає нові інструкції. Воно може впливати на потік даних,якщо непрозорий предикат буде використовувати змінні програми. Зазвичай ГЗП оригінальної і перетворюваної функцій не сильно відрізняються, тому алгоритми на основі слайсинга та ізоморфізму дерев знаходять такі клони.

В роботі здійснено тестування точності реалізованих алгоритмів пошуку клонів коду на основі слайсингу,ізоморфізму дерев і метрик. Для проектів Linux,Firefox Mozilla, LLVM/Clang і OpenSSL автоматично генерується множина клонів коду. Після цього алгоритми пошуку клонів коду запускаються для

оригінальною і генерованої пари ГЗП. На рисунку 3.7 наведено усереднений відсоток точності реалізованих алгоритмів за всіма проектами.

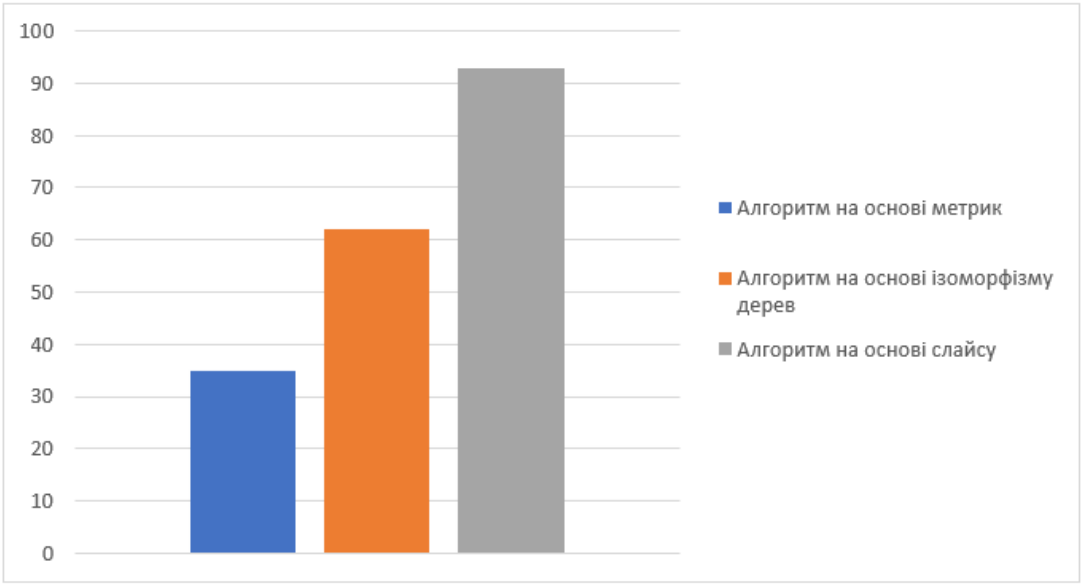

Рисунок 3.7-Порівняння точності реалізованих алгоритмів

В інструменті містяться два спеціальних скрипти для аналізу знайдених клонів. Перший скрипт дозволяє переглянути вихідний код клонів та відповідні їм графи (рисунок 3.8). Скрипт підтримує збереження знайдених клонів в HTML і EXCEL форматах.

Рисунок 3.8-Інструмент перегляду результатів

Другий скрипт дає можливість простежити історію копіювання конкретного фрагмента коду. Також дозволяє інтерактивно просуватися подереву файлів, який містить клони даного фрагмента(рисунок 3.9).

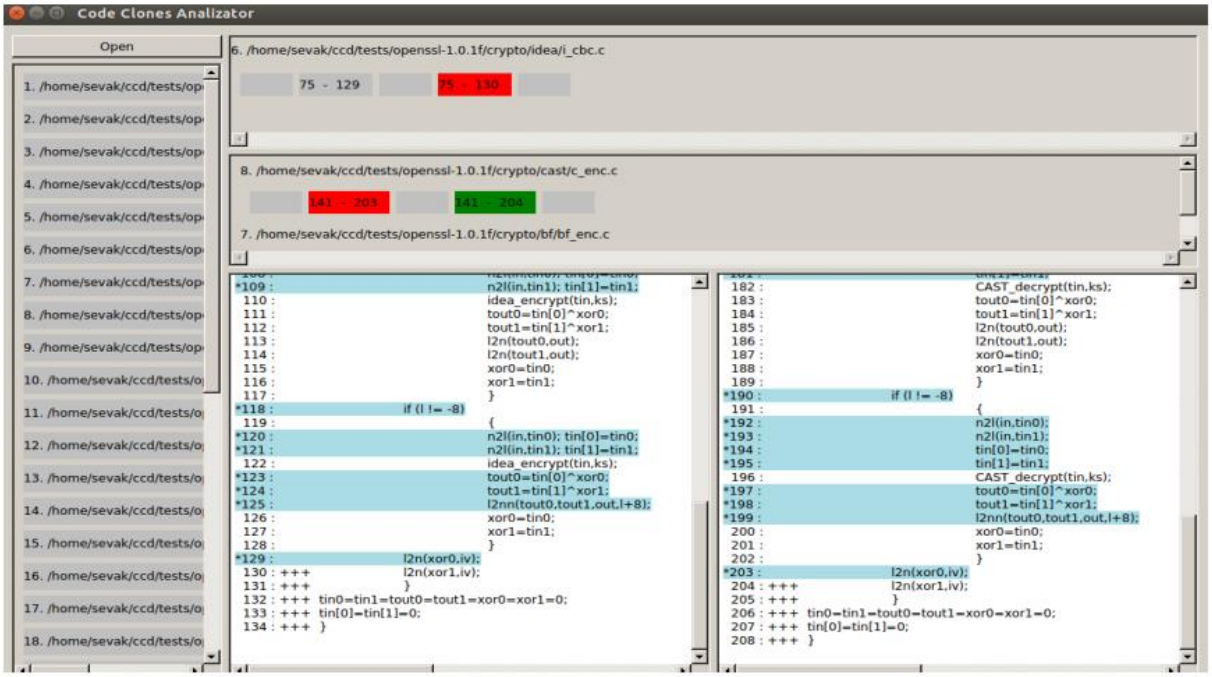

Рисунок 3.9-Інструмент перегляду результатів

В подальших дослідженнях планується реалізація засобів пошуку клонів коду з використанням багатоядерних системи.

3.3.Дослідження розроблених методів та засобів пошуку клонів коду для JavaScript

В останні роки широке поширення одержали програми наскриптових мовах. На сьогоднішній день однією з популярних скриптових мов є JavaScript. Завдяки зростанню продуктивності персональних комп’ютерів і вбудованих систем JavaScript стало можливо використовувати для програмування веб-додатків. Вже існують операційні системи для мобільних телефонів і планшетів, в яких мова JavaScript є однією з основних мов для розробки додатків (Tizen [9] і FirefoxOS [9]). У таких системах користувацький додаток являє собою набір веб-сторінок, написаних на мові JavaScript. У зв’язку з цим все більше зростають вимоги до продуктивності програм-скриптів. Швидке зростання кількості JavaScript додатків ставить питання про ефективне виконання таких додатків. Питання вирішується застосуванням динамічної(Just-In-Time) компіляції. Деякі великі компанії, такі як Google,Mozilla і Apple, пропонують власні динамічні компілятори для мовиJavaScript. Динамічний компілятор працює під час виконання програми,і час,

витрачений ним на компіляцію, додається до загального часу виконання програми. Завдяки зібраній інформації про профілі працюючої функції динамічні компілятори можуть оптимізувати ділянки коду під час виконання скриптів.

Швидке зростання кількості вихідного коду додатків, написаних наJavaScript, також призводить до виникнення все більшої кількості клонів коду. До останнього часу JavaScript вважався інтерпретованою мовою, і для нього не здійснювалися складні аналізи. Оптимізучі динамічні компілятори, що використовують проміжне представлення мовиJavaScript з'явилися відносно недавно. Тому на сьогоднішній день не існує інструменту пошуку клонів коду для мови JavaScript, який був б заснований на семантичному аналізі програми. Існуючі інструменти засновані на текстовому, лексичному або синтаксичному аналізі, що не дозволяє досягти найбільшої точності. У цьому розділі протестований метод пошуку клонів коду для мови JavaScript на основі семантичного аналізу програми. За базу інструмент використовує V8 JIT [16] компілятор компанії Google. На основі проміжного представлення (Hydrogen) V8 отримується набір графів залежностей програм для скриптів. Пошук клонів коду проводиться шляхом застосування розробленого інструменту пошуку клонів коду.

Динамічний компілятор мови JavaScript V8 [16] складається з двох окремих компіляторів (рисунок 3.10).

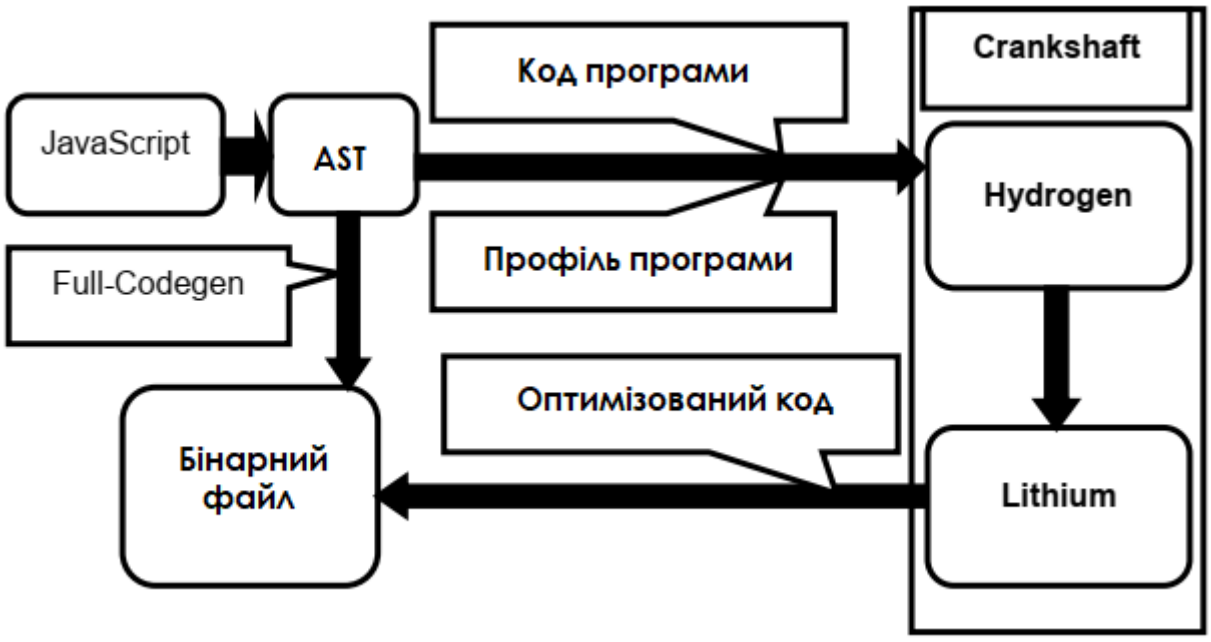

Рисунок 3.10-Архітектура динамічного компілятора V8

Подання Hydrogen за своєю структурою схоже на проміжне представлення LLVM. Воно являє собою граф потоку керування,що складається з базових блоків. Базові блоки містять послідовності інструкцій в SSA формі. Кожна інструкція крім операндів містить список інструкцій, які використовують її результат. Таким чином, представлення Hydrogen містить всі залежності між настановами щодо наступних кроків за даними, так і за управління.

Граф залежностей програми (ГЗП) містить залежності за даними із управлінням. ГЗП - спрямований граф, в якому вершинами є інструкції програми, а ребра відповідають залежностям між інструкціями. Розрізняються два типи ребер: ребра першого типу представляють залежності з управління, ребра другого типу - залежності за даними. Як відомо, є три типи залежностей за даними:

1. Справжня залежність. Значення, записане в пам’ять першою інструкцією,читається в другій.

2. Антизалежність. Значення, записане в пам’ять другої інструкції,читається в першій.

3. Залежність по виходу. Обидві інструкції записують значення по одній і тій ж адресі.

Розглянемо побудову графа залежностей програми за представленням Hydrogen. Для кожної інструкції Hydrogen будується нова вершина ГЗП. Тип (мітка) нової вершини визначається кодом операції відповідної інструкції Hydrogen. У вершині ГЗП зберігається посилання на вихідний код інструкції. Між двома вершинами додається залежність за даними, якщо між відповідними до інструкцій представлення Hydrogen є залежність за даними. Тип залежності відзначається на доданому ребрі. Між двома вершинами ГЗП додається залежність по управлінню, якщо інструкція, відповідна першій вершині, виконується перед інструкцією, відповідної другій вершині (рисунок 3.11).

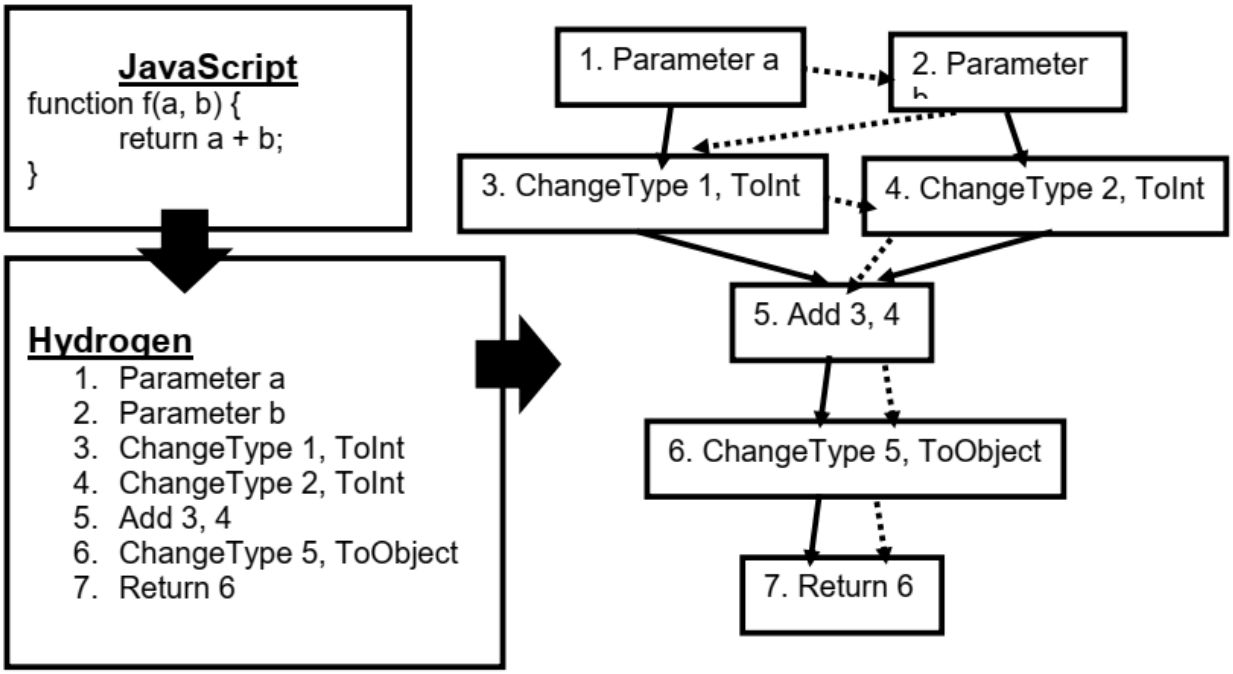

Рисунок 3.11-Приклад перетворення представлення Hydrogen в ГЗП (граф залежностей програми)

На рисункц3.12 показані зміни, які були зроблені в динамічному компіляторі V8, щоб згенерувати ГЗП для додатків, написаних намові JavaScript.

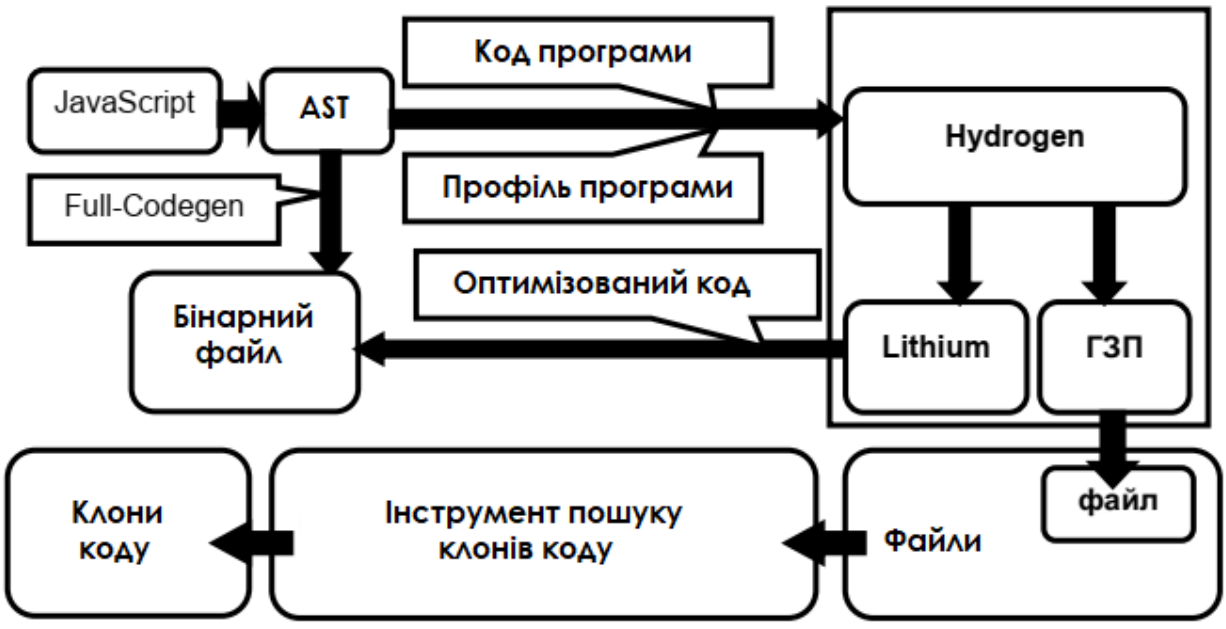

Рисунок 3.12-Пошук клонів коду для мови JavaScript на основі динамічного компілятора V8.

Отриманий ГЗП зберігається в файлі. Інструмент пошуку клонів коду запускається для з генерованих ГЗП файлів, яктількикомпіляторV8 закінчує свою роботу. Слід зазначити, що формат ГЗП точно такий же, що і формат ГЗП для мов C/C ++, що генерується компілятором LLVM.

Цедаєможливістьвикористовуватирозробленіінструментипошукуклонівкоду і для мови JavaScript.

Тестування інструменту для JavaScript проводилося на Intel Core i3 CPU540 з 8 ГБ оперативної пам’яті. Розмір мінімального клону становив 10 рядків вихідного коду. Були проаналізовані тестові набори SunSpider, Kraken iOctane.ТестовийнабірSunSpiderствориливрозробникибраузераWebKit(компанія Apple). У ньому містяться різні обчислення з реальних проектів:

1. графічні обчислення;

2. доступ і операції з різними полями об’єктів;

3. по бутові операції;

4. рекурсивні виклики;

5. криптографічні обчислення;

6. операції з об’єктами, що представляють час (DateTime objects);

7. обчислення, пов’язані з регулярними виразами;

8. рядкові операції.

Тестовий набір Kraken, розроблений компанією Mozilla, містить наступні тести:

1. алгоритм пошуку A*;

2. обробка аудіо (звуку);

3. обробка зображень;

4. розбір даних в форматі JSON;

5. обробка криптографічних даних.

Тестовий набір Octane розроблений компанією Google. У ньому містяться тести для перевірки пропускної здатності інтерактивних веб-додатків і збірки сміття. Нижче наведено список тестів:

1. Симуляція ядра операційної системи. Містить функціонали, що відповідають за завантаження об’єктів, виклик функції, оптимізацію коду і видалення повторного коду.

2. Тести орієнтовані на поліморфізм і ООП.

3. Операції з регулярними виразами, отримані на основі аналізу більше 50 популярних веб-сторінок.

4. Рішення рівнянь 2D. Містить читання і запис масивів, а також операції з числами з плаваючою комою.

5. Криптографічні обчислення, що містять бітові операції.

6. Автоматичне управління пам’яттю, яке дозволяє швидко створювати і видаляти об’єкти.

7. Тести, що вимірюють затримку збирача сміття.

8. 3D симуляція.

9. Завантаження коду: вимірює час розбору і компіляції JavaScript.

Нарисунку3.13показанокількістьрядківпроаналізованихпроектів.Octane містить більше 350.000 вихідного коду, написаного на JavaScript.

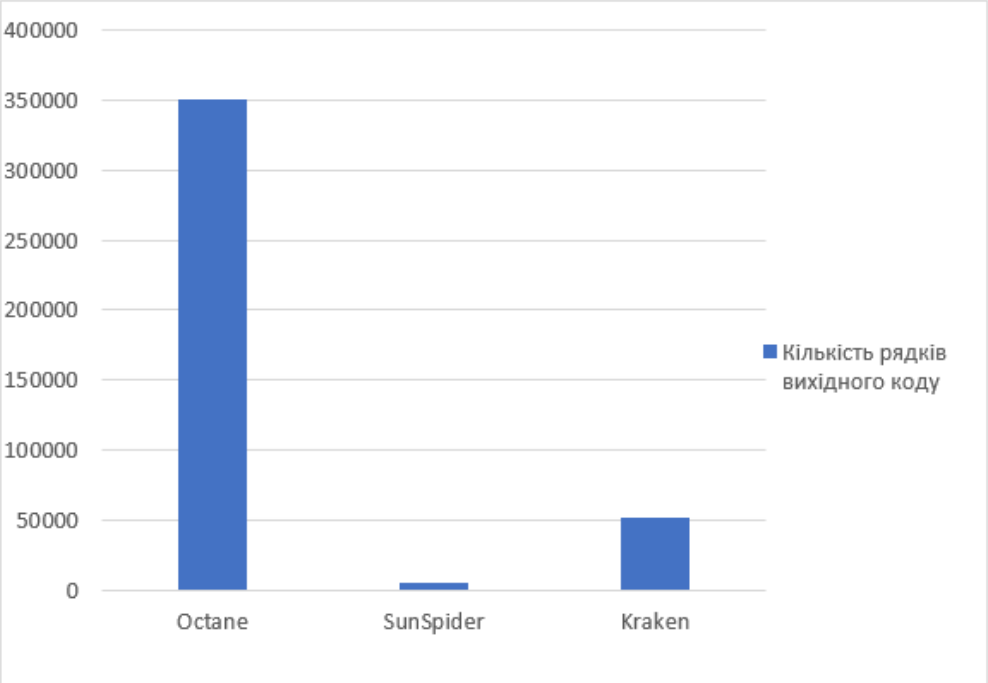

Рисунок 3.13-Кількість рядків вихідного коду проаналізованих проектів

На рисунку3.14 показаний розмір ГЗП проаналізованих проектів. Для Octane він становить 26.2 мегабайта, для SunSpider і Kraken 0.8 і 2.5відповідно.

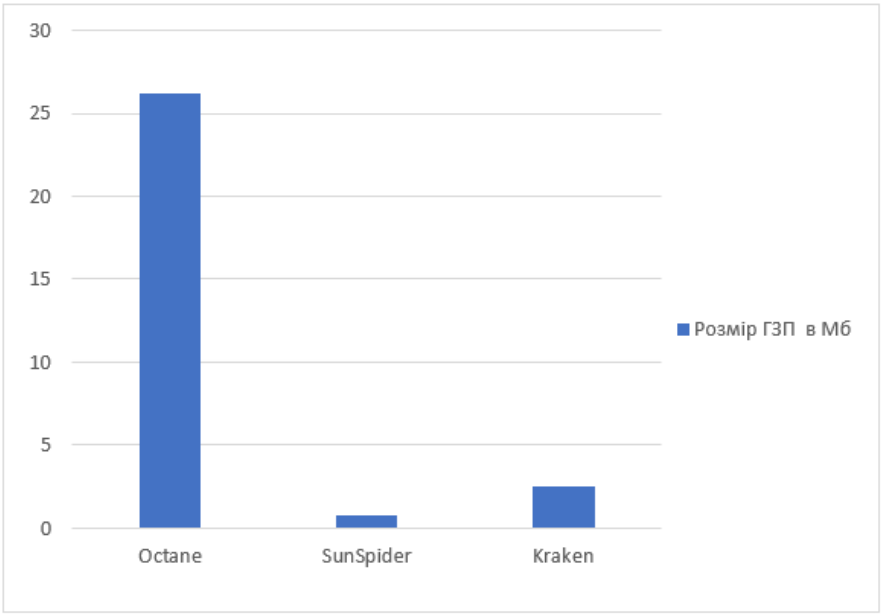

Рисунок 3.14- Розмір ГЗП проаналізованих проектів

На рисунку3.15 показано час пошуку клонів коду. Для Octane він становить 428 секунд, для SunSpider і Kraken 19.4 і 14.2 секунди відповідно.

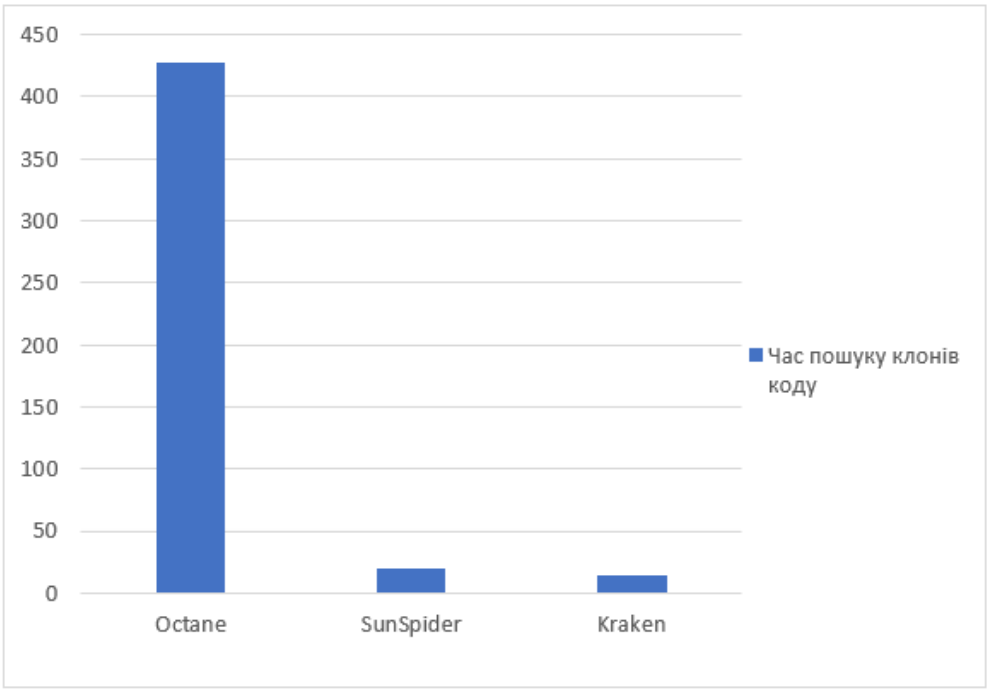

Рисунок 3.15- Час пошуку клонів коду проаналізованих проектів

На рисунку 3.16 показано кількість знайдених клонів і кількість помилкових спрацьовувань. Інструмент знайшов 342 і 10 клонів для Octane і SunSpider відповідно. SunSpider має одне помилкове спрацьовування. Kraken не містить жодного клону.

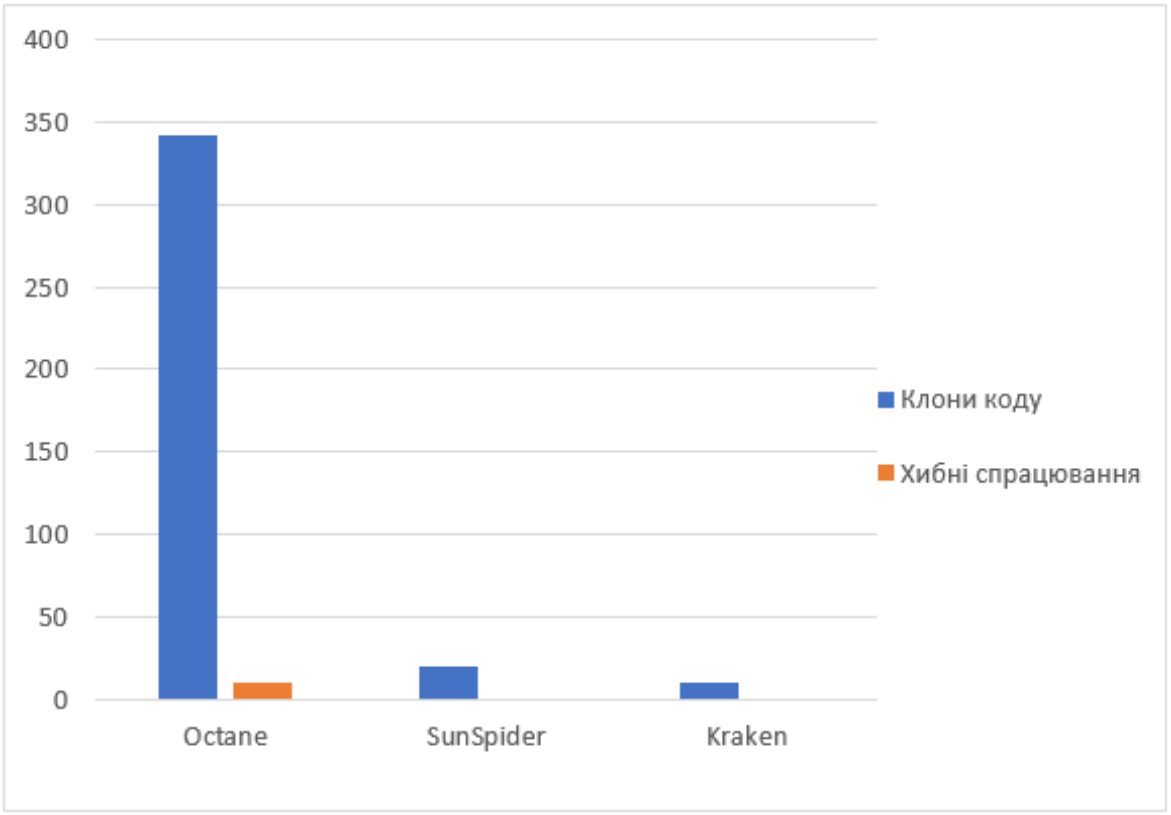

Рисунок 3.16- Кількість знайдених клонів і помилкових спрацьовувань

Розроблений інструмент пошуку клонів коду для мови JavaScript був порівняний з інструментом CloneDR [22]. Тестування проводилося на наборі тестів Octane. Для обох інструментів розмір мінімального клону становив 10рядків вихідного коду, а схожість - більше 90 відсотків. CloneDR знайшов 35клонів, а розроблений інструмент 342. Обидва інструменти мали 21 спільний клон.

Висновки до третього розділу

На основі запропонованих методів розроблено та реалізовано інструмент пошуку клонів коду. Розроблені інструменти включені в компіляторну інфраструктуру LLVM. Проведено тестування розроблених методів та програмних засобів. Підтверджено ефективність запропонованих методів.

4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1. Вимоги щодо охорони праці при роботі з комп'ютерами. Інструкція для програміста.

Сучасний розвиток технічного та технологічного стану виробництва передбачає постійну автоматизацію та оптимізацію виробничих процесів. Сьогодні, напевно, важко уявити компанію, господарська діяльність в якій здійснювалась би без використання комп'ютерної техніки. Через масовий характер робіт, що виконуються працівниками за допомогою комп'ютера, законодавством України чітко врегульовано норми та вимоги до використання комп'ютерної техніки на підприємстві, безпосередньо й охорона праці при роботі з комп'ютером.

Вимоги до приміщення

Приміщення, в яких планується установка та подальша робота з комп'ютером, повинні відповідати проектній документації будинку, погодженій з уповноваженими державними органами. Крім того, роботодавець повинен враховувати санітарні нормативи освітлення, вимоги до параметрів мікроклімату (температура, відносна вологість), ступеня і сили вібрації, звукового шуму і вогнестійкості приміщення, а також характеристики електромагнітного, ультрафіолетового та інфрачервоного полів. Конкретні показники зазначених санітарних норм див. в Державних санітарих правилах і нормах роботи з візуальними дисплейними терміналами електронно-обчислювальних машин ДСанПІН 3.3.2.007-98, затверджених Постановою Головного державного санітарного лікаря України №7 від 10 грудня 1998 року.

Правила поширюються на умови й організацію праці при роботі з візуальними дисплейними терміналами (ВДТ) усіх типів вітчизняного та зарубіжного виробництва на основі електронно-променевих трубок (ЕПТ), що використовуються в електронно-обчислювальних машинах (ЕОМ) колективного використання та персональних ЕОМ (ПЕОМ). Так, наприклад, роботодавцю заборонено установлювати комп'ютери в приміщеннях, розташованих у підвалах будинків.

Для уникнення можливих аварій та замикань, поряд з приміщеннями, де вестиметься робота з комп’ютером (над чи під ними), також не дозволяється проведення робіт, що потребують здійснення надмірно вологих технологічних процесів. Відповідне приміщення повинно бути укомплектоване системами центрального або індивідуального опалення, кондиціонування чи вентиляції повітря. Але при установці зазначених систем, необхідно переконатись, що батареї опалення, водопровідні труби, вентиляційні кабелі тощо, надійно сховані під захисними щитками, які перешкоджатимуть можливому потраплянню робітника під напругу.

У кожній кімнаті, де обладнуватимуться робочі місця співробітників, що працюватимуть на комп’ютері, повинні бути наявні елементи природного та штучного освітлення. При цьому, на вікнах слід встановити легко регульовані жалюзі чи штори, які дозволять працівникам коригувати рівень освітлення в приміщенні. Бажано розмістити комп’ютери в кімнаті таким чином, щоб світло потрапляло на екрани моніторів з півдня чи північного сходу. З метою досягнення максимального рівня безпечності і охорони праці при роботі з комп’ютером, виробничі приміщення необхідно обладнати аптечками першої медичної допомоги, системами автоматичної пожежної сигналізації і вогнегасниками. В приміщенні, в якому разом працюють 5 або більше комп’ютерів, на видимому місці установлюється службовий вимикач, який у разі потреби дозволить повністю відключити електричне живлення кімнати.

Вимоги до особистого робочого місця програміста

Роботодавець, який використовує найману працю робітників, повинен забезпечити відповідність їхніх робочих місць комфортним та безпечним умовам. Розмір одного робочого місця має становити не менше 6 квадратних метрів. При необхідності, суміжні робочі місця співробітників, що працюють з комп’ютером, слід розділити перегородками висотою до 2 метрів. При визначенні достатнього розміру приміщення і робочого місця на одну особу необхідно додатково враховувати шафи, сейфи, тумби або інші предмети меблів чи обладнання, які знаходяться в кімнаті. На столі працівника можливо розмістити допоміжні для роботи пристрої (принтери, колонки, сканери), а також місця для зберігання документів, за умови, що це не обмежуватиме видимість екрану і не заважатиме

працівнику. У разі надмірного шуму чи вібрації технічного обладнання, роботодавець повинен забезпечити працівників антивібраційними килимками. Робочий стілець співробітника має бути підйомно-поворотним, легко регульованим за висотою та забезпечувати належну підтримку та зручне положення спини і хребта особи. Щодня необхідно проводити вологе прибирання приміщення, та очищати робоче місце та безпосередньо монітор комп'ютера від запиленості.

На підприємстві забороняється: проводити ремонт та технічне обслуговування комп'ютера за робочим місцем працівника; самочинно ремонтувати або намагатись здійснити технічне налагодження комп'ютера без залучення компетентних спеціалістів; складувати на робочому місці зайві документи, деталі та предмети, що не потрібні для роботи; використовувати монітори з нечітким зображенням та монітори, у яких наявні поламки екрану; працювати з матричним принтером без антивібраційного покриття та зі знятою кришкою. Допускати до роботи осіб, які не пройшли затверджений на підприємстві курс охорони праці для роботи з комп'ютером, не дозволяється.

Законодавство:

– Наказ Державного комітету України з промислової безпеки, охорони праці та гірничого нагляду «Про затвердження Правил охорони праці під час експлуатації електронно-обчислювальних машин» від 26.03.2010 № 65;

– Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин ДСанПіН 3.3.2.007-98, затверджені постановою Головного державного санітарного лікаря України від 10.12.1998 № 7;

– Примірну інструкцію з охорони праці під час експлуатації електронно-обчислювальних машин, затверджену наказом Міністерства доходів і зборів України від 05.09.2013 № 443.

4.2. Забезпечення електробезпеки користувачів ПК.

Приміщення із робочими місцями користувачів комп'ютерів для забезпечення електробезпеки обладнання, а також для захисту від ураження електричним струмом самих користувачів ПК повинні мати достатні технічні засоби захисту відповідно до ГОСТ 12.1.009-76, НПАОП 40.1-1.07-01 "Правила експлуатації електрозахисних засобів", НПАОП 40.1-1.21-98 "Правила безпечної експлуатації електроустановок споживачів", НПАОП 40.1-1.32-01 "Правила будови електроустановок. Електрообладнання спеціальних установок".

З метою запобігання ушкодженням, що можуть статися через ураження електричним струмом, загоряння, коротке замикання тощо, розроблено загальний стандарт безпеки IEC 950. Загальним стандартом електробезпечності для країн Європейської співдружності є Cemark.

Під час проектування систем електропостачання, монтажу силового електрообладнання та електричного освітлення будівель та приміщень для ПЕОМ необхідно дотримуватись вимог вищеназваних нормативно-правових актів, а також СН 357-77, Правил пожежної безпеки в Україні, ДСанПіН 3.3.2.007-98, розділів СНиП, що стосуються штучного освітлення і електротехнічних пристроїв, та вимог нормативно-технічної і експлуатаційної документації заводу-виробника ПЕОМ.

ЕОМ, периферійні пристрої ЕОМ та устаткування для обслуговування, ремонту та налагодження ЕОМ, інше устаткування (апарати управління, контрольно-вимірювальні прилади, світильники тощо), електропроводи та кабелі за виконанням та ступенем захисту мають відповідати класу зони за ПУЕ, мати апаратуру захисту від струму короткого замикання та інших аварійних режимів.

Під час монтажу та експлуатації ліній електромережі необхідно повністю унеможливити виникнення електричного джерела загоряння внаслідок короткого замикання та перевантаження проводів, обмежувати застосування проводів з легкозаймистою ізоляцією і, за можливості, перейти на негорючу ізоляцію.

Лінія електромережі для живлення ЕОМ, периферійних пристроїв ЕОМ та устаткування для обслуговування, ремонту та налагодження ЕОМ виконується як окрема групова трипровідна мережа, шляхом прокладання фазового, нульового робочого та нульового захисного провідників. Нульовий захисний провідник використовується для заземлення (занулення) електроприймачів.

Використання нульового робочого провідника як нульового захисного провідника забороняється. Нульовий захисний провід прокладається від стійки групового розподільчого щита, розподільчого пункту до розеток живлення. Не допускається підключення на щиті до одного контактного затискача нульового робочого та нульового захисного провідників. Площа перерізу нульового робочого та нульового захисного провідника в груповій трипровідній мережі повинна бути не менше площі перерізу фазового провідника.

Усі провідники повинні відповідати номінальним параметрам мережі та навантаження, умовам навколишнього середовища, умовам розподілу провідників, температурному режиму та типам апаратури захисту, вимогам ПУЕ.

У приміщенні, де одночасно експлуатується або обслуговується більше п'яти персональних ЕОМ, на помітному та доступному місці встановлюється аварійний резервний вимикач, який може повністю вимкнути електричне живлення приміщення, крім освітлення.

ПЕОМ, периферійні пристрої ПЕОМ та устаткування для обслуговування, ремонту та налагодження ЕОМ повинні підключатися до електромережі тільки з допомогою справних штепсельних з'єднань і електророзеток заводського виготовлення. Штепсельні з'єднання та електророзетки крім контактів фазового та нульового робочого провідників повинні мати спеціальні контакти для підключення нульового захисного провідника. Конструкція їх має бути такою, щоб приєднання нульового захисного провідника відбувалося раніше ніж приєднання фазового та нульового робочого провідників. Порядок роз'єднання при відключенні має бути зворотним. Необхідно унеможливити з'єднання контактів фазових провідників з контактами нульового захисного провідника.

Неприпустимим є підключення ПЕОМ та периферійних пристроїв ПЕОМ до звичайної двопровідної електромережі, в тому числі – з використанням перехідних пристроїв.

Електромережі штепсельних з'єднань та електророзеток для живлення ПЕОМ, периферійних пристроїв слід виконувати за магістральною схемою, по 3…6 з’єднань або електророзеток в одному колі. Штепсельні з'єднання та електророзетки для напруги 12 В та 36 В за своєю конструкцією повинні відрізнятися від штепсельних з'єднань для напруги 127 В та 220 В і мають бути

пофарбовані в колір, який візуально значно відрізняється від кольору штепсельних з'єднань, розрахованих на напругу 127 В та 220 В.

Індивідуальні та групові штепсельні з'єднання та електророзетки необхідно монтувати на негорючих або важкогорючих пластинах з урахуванням вимог ПУЕ та Правил пожежної безпеки в Україні.

Електромережу штепсельних розеток для живлення ПЕОМ, периферійних пристроїв ПЕОМ при розташуванні їх уздовж стін приміщення прокладають по підлозі поряд зі стінами приміщення, як правило, в металевих трубах і гнучких металевих рукавах з відводами відповідно до затвердженого плану розміщення обладнання та технічних характеристик обладнання.

При розташуванні в приміщенні за його периметром до 5 ПЕОМ, використанні трипровідникового захищеного проводу або кабелю в оболонці з негорючого або важкогорючого матеріалу дозволяється прокладання їх без металевих труб та гнучких металевих рукавів.

ВИСНОВКИ

В результаті виконання кваліфікаційної магістерської роботи отримано наступні теоретичні та практичні результати.

1. Проведений аналіз існуючих методів пошуку клонів коду і пошуку семантичних помилок, що виникають при неправильному копіюванні вихідного коду.

2. Запропоновано метод побудови набору графу залежностей коду проекту на основі проміжного представлення компіляторної інфраструктури LLVM, що дозволяє отримувати набір ГЗП під час компіляції проекту без додаткових накладних витрат.

3. Запропоновано метод побудови ГЗП проекту на основі проміжного представлення Hydrogen JIT компілятор V8.

4. Запропоновано метод поділу графа залежностей програм на під графи, що дозволяє зменшити кількості незнайдених справжніх клонів.

5. Запропоновано метод пошуку клонів коду на основі семантичного аналізу: поділ ГЗП напідграфи, відсіювання пар ГЗП, що не містять клонів, алгоритмами лінійної складності, виявлення максимально схожих під графів за допомогою слайсингу, фільтрація помилкових спрацьовувань. Метод масштабується до десятків мільйонів рядків коду.

6. Запропоновано схему автоматичної генерації клонів коду, яка дозволяє здійснювати оцінку точності реалізованих алгоритмів.

На основі запропонованих методів розроблено та реалізовано інструмент пошуку клонів коду. Розроблені інструменти включені в компіляторну інфраструктуру LLVM.

Серед напрямків подальшої роботи з даної тематики можна виділити найбільш важливі:

1. Пошук вразливостей на основі існуючих шаблонів. Ідея полягає в побудові бази ГЗП для програм, що містять вразливість. Для пошуку вразливосте й в проекті проводиться пошук схожих під графів з бази.

2. Застосування інструменту в задачах оптимізації розміру коду. Ідея полягає в тому, щоб інструмент автоматично створював функцію для групи клонів різних типів і заміняв клони викликами цієї функції.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. A.V. Burdakov, U.A. Grigorev, A.D. Ploutenko. Comparison of table joinexecution time for parallel DBMS and MapReduce, Software Engineering / 811:Parallel and Distributed Computing and Networks / 816: Artificial Intelligence andApplications Proceedings (March 18 – 18, 2014, Innsbruck, Austria), ACTA Press,2014.

2. Aleksey Burdakov, Uriy Grigorev, Andrey Ploutenko, Eugene Ttsviashchenko"Estimation Models for NoSQL Database Consistency Characteristics", 24thEuromicro International Conference on Parallel, Distributed, andNetwork-BasedProcessing (PDP), 2016, pp. 35-42, doi: 10.1109/PDP.2016.23.

3. Аткинсон Л., Сураски З. PHP5: Библиотека профессионала, 3–е издание: Пер. з англ. – М.: Вильямс, 2006. – 944 с.

4. Bettina Kemme. Gustavo Alonso. Database Replication: a Tale of Research acrossCommunities. Proceedings of the VLDB Endowment, Vol. 3, No. 1. P. 5-12.

5. Buasilovsky, P. Adaptive and intelligent Web-based educational systems / P. Buasilovsky, CPeylo// International Journal of Artificial Intelligence in Education. Special Issue on Adaptive and Intelligent Web-based Educational Systems -2003. -№13 (2-4). -P. 159-172.

6. Codd, Edgar F.: A Relational Model of Data for Large Shared Data Banks. In:Communications of the ACM 13 (1970), June, No. 6, p. 377–387.

7. Converse T., Park J., Morgan C. PHP5 and MySQL Bible. – Indianapolis, Canada: Wiley Publishing Inc., 2004. – 1083 p.

8. Ситник В.Ф. Системи підтримки прийняття рішень. – К.: Техніка, 2005. –164с.

9. Ситник В.Ф. Системи підтримки прийняття рішень. – К.: Техніка, 2005. –164с.

10. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T.Chandra, A. Fikes, and R. E. Gruber, Bigtable: a distributed storage system forstructured data, Proceedings of the 7th Conference on USENIX Symposium onOperating Systems Design and Implementation - Volume 7 (Seattle, WA,November 06 - 08, 2006), USENIX Association, Berkeley, CA, 15-15, 2006.

11. FreshKnowledge CMS - онтологічно-орієнтована система керування контентом, розроблена здобувачем [Електронний ресурс]. - Режим доступу : http://www.freshknowledge.net.

12. FreshKnowledge CMS - онтологічно-орієнтована система керування контентом, розроблена здобувачем [Електронний ресурс]. - Режим доступу : http://www.freshknowledge.net.

13. International Telecommunication Union Measuring the Information Society Report Volume 1 URL: https://www.itu.int/en/ITU-D/Statistics/Documents/publications/ misr2018/MISR-2018-Vol-1-E.pdf Çдата звернення 15.12.2019).

14. Let's Encrypt - Free SSL/TLS Certificates URL: https://letsencrypt.org/ Çдата звернення 15.12.2019).

15. Lewisand D. Lewisand D. Acomparison of two learning algorithms for text categorization [Електронний ресурс] / D. Lewisand, M. Ringuette // In Third Annual Symposium on Document Analysis and Information Retrieval. – 1994. – Режим доступу до ресурсу: http://www.research.att.com/~lewis/papers/lewis94b.ps.

16. Marshall, B. Convergence of Knowledge Deputatagement and E-Learning: the GetSmart Experience [Електроннийресурс] / Marshall, B., et al. // JCDL. - Houston, 2003. - Режимдоступу: http://ai.bpa.arizona.edu/go/intranet/ Publication/JCDL-2003-Marshall.pdf.

17. Marshall, B. Convergence of Knowledge Management and E-Learning: the GetSmart Experience [Електронний ресурс] / Marshall, B., et al. // JCDL. -Houston, 2003. - Режим доступу: http://ai.bpa.arizona.edu/go/intranet/ Publication/JCDL-2003-Marshall.pdf.

18. Murray, T. AuthoringIntelligentTutoringSystems: AnAnalysisoftheStateoftheArt / T. Murray. // InternationalJournalofArtificialIntelligenceinEducation. -1999.-№10-P. 98-129

19. TENCompetence- European research project for lifelong competence development [Електроннийресурс]. - Режимдоступу: **Ошибка! Недопустимый объект гиперссылки.**www.tencompetence.org/.

20. Y. Sheffer, R. Holz, P. Saint-Andre Summarizing Known Attacks on Transport Layer Security ÇTLS) and Datagram TLS ÇDTLS).

21. Андон, Ф. И. Логические модели интеллектуальных информационных систем / Ф. И. Андон, А. Е. Яшунин, В. А. Резниченко. - К: Наукова думка.-1999.-397 с.

22. Андон, Ф. И. Логические модели интеллектуальных информационных систем / Ф. И. Андон, А. Е. Яшунин, В. А. Резниченко. - К: Наукова думка.-1999.-397 с.

23. Антонченко, М. О. Експертні системи як засіб формування якісних знань учнів 7-8 класів з предметів природничого циклу: Автореф. дис. ... канд. пед. наук: 13.00.09 / Антонченко Марія Олексіївна ; Харк. держ. пед. ун-т ім. Г. С. Сковороди. -Х., 2001. - 16 с.

24. Валгина, Н. С. Теория текста: Учебное пособие / Н. С. Валгина. - Москва. Изд-во МГУП «Миркниги» - 1998. - 210 с.Андон, Ф. И. Логические модели интеллектуальных информационных систем / Ф. И. Андон, А. Е. Яшунин, В. А. Резниченко. - К: Наукова думка.-1999.-397 с.

25. Дейт К. Дж. Введение в системы баз данных: Пер. с англ. – 6–е изд. – К.: Диалектика, 2007. – 784 с.

26. Елизаренко, Г. Н. Проектирование компьютерных курсов обучення: концепция, язык, структура / Г. Н. Елизаренко. - К.: НТУУ «КПИ», 2001.

27. Елизаренко, Г. Н. Проектирование компьютерных курсов обучення: концепция, язык, структура / Г. Н. Елизаренко. - К.: НТУУ «КПИ», 2001.

28. Іщук В. І.Сетифікація програмного забезпечення на основі моделі якості/ В. І. Іщук, І. О. Боднарчук // Збірник тез доповідей VI Міжнародної науково-технічної конференції молодих учених та студентів „Актуальні задачі сучасних технологій“, 16-17 листопада 2017 року. — Т. : ТНТУ, 2017. — Том 2. — С. 73–74. — (Комп’ютерно-інформаційні технології та системи зв’язку).

29. Комп’ютерна система аутентифікації осіб/ В. А. Марків, Г. М. Осухівська, Ю. З. Лещишин, А. М. Луцків // Матеріали XX наукової конференції ТНТУ ім. І. Пулюя, 17-18 травня 2017 року. — Т. : ТНТУ, 2017. — С. 90–91. — (Інформаційні технології).

30. Краковецкий А. Кластеризация: алгоритмы k-means и c-means [Електронний ресурс] / Александр Краковецкий // Habrahabr. – 2009. – Режим доступу до ресурсу: http://habrahabr.ua/post/67078/.

31. Лапінський В. В., Габрусєв В. Ю. Основи операційних систем: Посібник для студентів. – К.: Вища школа, 2007. – 96 с.

32. Лаврищева, Е. М. Методы программирования: теория, инженерия, практика / Е. М. Лаврищева. - К.: Наукова думка. - 2006. - 451 с.

33. Лаврищева, Е. М. Методы программирования: теория, инженерия, практика / Е. М. Лаврищева. - К.: Наукова думка. - 2006. - 451 с.

34. Луцків А. М.Архітектури комп'ютерних систем опрацювання великих даних/ А. Луцків, В. Діденко // Матеріали VI науково-технічної конференції „Інформаційні моделі, системи та технології“, 12-13 грудня 2018 року. — Т. : ТНТУ, 2018. — С. 75. — (Комп’ютерні системи та мережі).

35. Люгер, Джордж, Ф. Искусственный интеллект: стратегии и методы решення сложных проблем / Люгер, Джордж, Ф. - 4-е издание.: Пер. с англ.- М.: Издательский дом «Вильяме», 2005. - 864 с.

36. Маєвський О. В. Будова та експлуатація ПК : Конспект лекцій / Маєвський О.В., Мацюк О.В., Смакула І.З. — Тернопіль : ПМП ”РОМС-К” , 2010 — 368 с. — ISBN 9665670786.

37. Маклаков С.В. BPwin и ERwin: CASE-средства для разработки информационных систем. – М.: Диалог-Мифи, 1999. - 295 с.

38. Марценко С. В.Математичне моделювання та статистичні методи обробки даних вимірювань в задачах моніторингу електронавантаження/ Марценко С.В. — Тернопіль , 2011 — 20 с.

39. Митчелл М., Оулдем Д., Самьюэл А. Программирование для Linux. Профессиональный поход. – М.: Вильямс, 2002. – 288 с.

40. Назаревич О.Комп'ютерні технології І CAD-програми в навчальному процесі: проблеми і методика/ Назаревич О., Назаревич Б. // Вісник Тернопільського державного технічного університету. — том 14. — с.176-178

41. Олецький О. В. Застосування формальних моделей онтологій для формалізації інформаційних потоків у системах управління контентом / О. В. Олецький //Теоретичні та прикладні аспекти побудови програмних систем. Матеріали міжнародної конференції TAAPSD’2005. Київ, 7-9 грудня 2005 р. - С. 26-29.

42. Основи програмування. Курс лекцій для студентів першого рівня вищої освіти за спеціальністю No 121 Інженерія програмного забезпечення/ Уклад.: М.Р. Петрик, О.Ю.Петрик - Тернопіль: ТНТУ 2018- 64 с.

43. Петровский, А. Б. Извлечение знаний для оценки кредитоспособности: подход теории мультимножеств / А. Б. Петровский // Труды Девятой национальной конференции по искусственному интеллекту с международным участием (КИИ-2004). - М.: Физматлит, 2004, том 2. -С. 853-860.

44. Самойлов, В. Д. Модельное конструирование компьютерных приложений / В. Д. Самойлов. - К.: Наукова думка. - 2007. - 198 с.

45. Самойлов, В. Д. Модельное конструирование компьютерных приложений / В. Д. Самойлов. - К.: Наукова думка. - 2007. - 198 с.

46. Семикин, В. А. Семантическая модель контента образовательных электронных зданий: Автореф. дис. … канд. тех. наук: 05.13.18 / Семикин Виктор Алексеевич ; Тюменск. гос. ун-т. - Тюмень, 2004. - 21 с.

47. Семикин, В. А. Семантическая модель контента образовательных электронных зданий: Автореф. дис. ... канд. тех. наук: 05.13.18 / Семикин Виктор Алексеевич ; Тюменск. гос. ун-т. - Тюмень, 2004. - 21 с.

48. Таненбаум Э., Вудхалл А. Операционные системы: разработка и реализация. Классика CS. – СПб.: Питер, 2006. – 576 с.

49. Титенко, С. В. FreshKnowledge - система управління навчальним Веб-контентом на семантичному рівні / С. В. Титенко, О. О. Гагарін // VII международная конференция «Интеллектуальный анализ информации ИАИ-2007», Киев, 15-18 мая 2007г. : Сб. тр. / Ред. кол. : С. В. Сирота (гл.ред.) и др. - К.: Просвіта, 2007. - С. 342-352.

~~50.~~ Титенко, С. В. FreshKnowledge - система управління навчальним Веб-контентом на семантичному рівні / С. В. Титенко, О. О. Гагарін // VII международная конференция «Интеллектуальный анализ информации ИАИ-

51. Федорова Д.Э., Семенов Ю.Д., Чижик К.Н. CASE-технологии. - М.: Горячая линия Телеком, Радио и связь, 2005. – 160 с.

52. Хоумер А., Улмен К. Dynamic HTML: справочник. – СПб.: Питер, 2000. – 465 с.

53. Харченко О. Г. Розробка та керування вимогами до програмного забезпечення на основі моделі якості/ Харченко О.Г., Яцишин В.В. // Вісник Тернопільського державного технічного університету. — том 14. — с.201-207

54. Холод Д. М. Проблеми захисту комп'ютерних систем / Д. М. Холод, Г. В. Шимчук // Збірник тез доповідей VI Міжнародної науково-технічної конференції молодих учених та студентів „Актуальні задачі сучасних технологій", 16-17 листопада 2017 року. — Т. : ТНТУ, 2017. — Том 2. — С. 179–180. — (Комп'ютерно-інформаційні технології та системи зв'язку).

55. Цвященко Є.В. Аналіз адекватності моделі узгодження реплік в кінцевомурахунку в базах даних NosQL // Інформаційні технології. - 2015. - Т.21.№ 11 - С. 840-848.

56. Чмир, І.О. Моделювання систем у середовищі UML (Unified Modeling Language) : навч. посібник / І. О. Чмир, М. Ф. Ус ; Черкаськ. акад. менеджменту. - Черкаси : ЧАМ, 2004. - 100 с.

57. Чмир, І.О. Моделювання систем у середовищі UML (UnifiedModelingLanguage) : навч. посібник / І. О. Чмир, М. Ф. Ус ; Черкаськ. акад. менеджменту. - Черкаси : ЧАМ, 2004. - 100 с.

58. Шатовська Т. Б. Комбінований ієрархічний підхід кластеризації документів [Електронний ресурс] / Т. Б. Шатовська, І. В. Каменєва // Харківський національний університет радіоелектроніки. – 2009. – Режим доступу до ресурсу:http://visnyk.vntu.edu.ua/index.php/visnyk/article/view/

59. Яцишин В. В. Технологія оцінювання якості WEB-застосувань / Яцишин В.В. // Вісник Тернопільського державного технічного університету. — том 14. — с.132-140

ДОДАТКИ

Додаток А

Лістинг програмного коду системи

```
package jp.naist.se.simplecc;

import java.io.File;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class CodeToken {

	private String text;
	private File file;
	private int line;
	private int charPositionInLine;
	private long hash;

	private static MessageDigest digest;

	static {
		try {
			digest = MessageDigest.getInstance("SHA-1");
		} catch (NoSuchAlgorithmException e) {
			throw new RuntimeException(e);
		}
	}

	public CodeToken(String text, String normalized, File f, int line, int charPositionInLine) {
		this.text = text;
		this.file = f;
		this.line = line;
		this.charPositionInLine = charPositionInLine;
		if (normalized != null) {
			this.hash = getHash(normalized);
		}
	}

	public static CodeTokengetTerminalToken() {
		return new CodeToken(null, null, null, 0, 0);
	}

	private long getHash(String s) {
		digest.update(s.getBytes());
		byte[] b = digest.digest();
		long hash = 0;
```

```
                for (int i=0; i<8; i++) {
                        hash = (hash << 8) + (long)b[i];
                }
                return hash;
        }

        public booleanisSameToken(CodeToken another) {
                return this.text != null &&another.text != null &&this.hash == another.hash;
        }

        public String getText() {
                return text;
        }

        public File getFile() {
                return file;
        }

        public int getLine() {
                return line;
        }

        public int getCharPositionInLine() {
                return charPositionInLine;
        }

        public int getEndCharPositionInLine() {
                return charPositionInLine + text.length();
        }

        @Override
        public String toString() {
                return file.getAbsolutePath() + "," + line + "," + charPositionInLine + "," + text + "," + hash;
        }
}
package jp.naist.se.simplecc;

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;

import org.antlr.v4.runtime.CharStreams;
import org.antlr.v4.runtime.Lexer;
import org.antlr.v4.runtime.Token;
```

```
/**
 * A simple type-2 clone detector.
 * @author Schur V.V.
 */
public class CloneDetectionMain {

	private static int MIN_TOKENS = 20;

	public static void main(String[] args) {
		ArrayList<CodeToken> tokens = readFiles(args);
		detectClones(tokens, MIN_TOKENS);
	}

	private static ArrayList<CodeToken>readFiles(String[] args) {
		ArrayList<CodeToken> tokens = new ArrayList<>();
		for (String arg: args) {
			DirectoryScan.scan(new File(arg), new DirectoryScan.Action() {
				@Override
				public void process(File f) {
					if (f.getName().toLowerCase().endsWith(".java")) {
						try {
							Java8Lexer lexer = new
Java8Lexer(CharStreams.fromPath(f.toPath()));
							for (Token t = lexer.nextToken(); t.getType() != Lexer.EOF; t
= lexer.nextToken()) {
								CodeToken token = new CodeToken(t.getText(),
getNormalizedText(t), f, t.getLine(), t.getCharPositionInLine());
								tokens.add(token);
							}
							tokens.add(CodeToken.getTerminalToken());
						} catch (IOException e) {
							e.printStackTrace();
						}
					}
				}
			});
		}
		return tokens;
	}

	private static String getNormalizedText(Token t) {
		switch (t.getType()) {
		case Java8Lexer.Identifier:
		case Java8Lexer.INT:
		case Java8Lexer.LONG:
```

```
                case Java8Lexer.FLOAT:
                case Java8Lexer.DOUBLE:
                case Java8Lexer.SHORT:
                case Java8Lexer.BOOLEAN:
                case Java8Lexer.BYTE:
                case Java8Lexer.VOID:
                        return "$p";
                default:
                        return t.getText();
                }
        }

        private static void detectClones(ArrayList<CodeToken> tokens, int threshold) {
                boolean[][] checked = new boolean[tokens.size()][tokens.size()];
                for (int i=0; i<tokens.size(); i++) {
                        for (int j=i+1; j<tokens.size(); j++) {
                                if (checked[i][j]) continue;

                                if (tokens.get(i).isSameToken(tokens.get(j))) {
                                        int match = 0;
                                        while (tokens.get(i + match).isSameToken(tokens.get(j + match))) {
                                                match++;
                                                checked[i+match][j+match] = true;
                                        }
                                        if (match >= threshold) {
                                                reportClone(tokens, i, j, match);
                                        }
                                }
                        }
                }
        }

        private static void reportClone(ArrayList<CodeToken> tokens, int start1, int start2, int length) {
                System.out.println("<pair>");
                printCode(tokens, start1, length);
                printCode(tokens, start2, length);
                System.out.println("</pair>");
        }

        private static void printCode(ArrayList<CodeToken> tokens, int start, int length) {
                CodeTokenstartToken = tokens.get(start);
                CodeTokenendToken = tokens.get(start + length - 1);
                assert startToken.getFile() == endToken.getFile();
                System.out.print(startToken.getFile().getAbsolutePath() + "," + startToken.getLine() + "," +
startToken.getCharPositionInLine() + "," + endToken.getLine() + "," + endToken.getEndCharPositionInLine() + ",");
```

```
                for (int i=0; i<length; i++) {
                        System.out.print(tokens.get(start + i).getText() + "\t");
                }
                System.out.println();
        }

}
package jp.naist.se.simplecc;
import org.antlr.v4.runtime.Lexer;
import org.antlr.v4.runtime.CharStream;
import org.antlr.v4.runtime.Token;
import org.antlr.v4.runtime.TokenStream;
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.atn.*;
import org.antlr.v4.runtime.dfa.DFA;
import org.antlr.v4.runtime.misc.*;

@SuppressWarnings({"all", "warnings", "unchecked", "unused", "cast"})
public class Java8Lexer extends Lexer {
        static { RuntimeMetaData.checkVersion("4.7", RuntimeMetaData.VERSION); }

        protected static final DFA[] _decisionToDFA;
        protected static final PredictionContextCache _sharedContextCache =
                new PredictionContextCache();
        public static final int
                ABSTRACT=1, ASSERT=2, BOOLEAN=3, BREAK=4, BYTE=5, CASE=6, CATCH=7, CHAR=8,
                CLASS=9, CONST=10, CONTINUE=11, DEFAULT=12, DO=13, DOUBLE=14, ELSE=15,
                ENUM=16, EXTENDS=17, FINAL=18, FINALLY=19, FLOAT=20, FOR=21, IF=22, GOTO=23,
                IMPLEMENTS=24, IMPORT=25, INSTANCEOF=26, INT=27, INTERFACE=28, LONG=29,
                NATIVE=30, NEW=31, PACKAGE=32, PRIVATE=33, PROTECTED=34, PUBLIC=35, RETURN=36,
                SHORT=37, STATIC=38, STRICTFP=39, SUPER=40, SWITCH=41, SYNCHRONIZED=42,
                THIS=43, THROW=44, THROWS=45, TRANSIENT=46, TRY=47, VOID=48, VOLATILE=49,
                WHILE=50, IntegerLiteral=51, FloatingPointLiteral=52, BooleanLiteral=53,
                CharacterLiteral=54, StringLiteral=55, NullLiteral=56, LPAREN=57, RPAREN=58,
                LBRACE=59, RBRACE=60, LBRACK=61, RBRACK=62, SEMI=63, COMMA=64, DOT=65,
                ASSIGN=66, GT=67, LT=68, BANG=69, TILDE=70, QUESTION=71, COLON=72, EQUAL=73,
                LE=74, GE=75, NOTEQUAL=76, AND=77, OR=78, INC=79, DEC=80, ADD=81, SUB=82,
                MUL=83, DIV=84, BITAND=85, BITOR=86, CARET=87, MOD=88, ARROW=89,
COLONCOLON=90,
                ADD_ASSIGN=91, SUB_ASSIGN=92, MUL_ASSIGN=93, DIV_ASSIGN=94, AND_ASSIGN=95,
                OR_ASSIGN=96, XOR_ASSIGN=97, MOD_ASSIGN=98, LSHIFT_ASSIGN=99,
RSHIFT_ASSIGN=100,
                URSHIFT_ASSIGN=101, Identifier=102, AT=103, ELLIPSIS=104, WS=105, COMMENT=106,
                LINE_COMMENT=107, ErrorCharacter=108;
        public static String[] channelNames = {
```

```
        "DEFAULT_TOKEN_CHANNEL", "HIDDEN"
};

public static String[] modeNames = {
        "DEFAULT_MODE"
};

public static final String[] ruleNames = {
        "ABSTRACT", "ASSERT", "BOOLEAN", "BREAK", "BYTE", "CASE", "CATCH", "CHAR",
        "CLASS", "CONST", "CONTINUE", "DEFAULT", "DO", "DOUBLE", "ELSE", "ENUM",
        "EXTENDS", "FINAL", "FINALLY", "FLOAT", "FOR", "IF", "GOTO", "IMPLEMENTS",
        "IMPORT", "INSTANCEOF", "INT", "INTERFACE", "LONG", "NATIVE", "NEW", "PACKAGE",
        "PRIVATE", "PROTECTED", "PUBLIC", "RETURN", "SHORT", "STATIC", "STRICTFP",
        "SUPER", "SWITCH", "SYNCHRONIZED", "THIS", "THROW", "THROWS", "TRANSIENT",
        "TRY", "VOID", "VOLATILE", "WHILE", "IntegerLiteral", "DecimalIntegerLiteral",
        "HexIntegerLiteral", "OctalIntegerLiteral", "BinaryIntegerLiteral", "IntegerTypeSuffix",
        "DecimalNumeral", "Digits", "Digit", "NonZeroDigit", "DigitsAndUnderscores",
        "DigitOrUnderscore", "Underscores", "HexNumeral", "HexDigits", "HexDigit",
        "HexDigitsAndUnderscores", "HexDigitOrUnderscore", "OctalNumeral", "OctalDigits",
        "OctalDigit", "OctalDigitsAndUnderscores", "OctalDigitOrUnderscore", "BinaryNumeral",
        "BinaryDigits", "BinaryDigit", "BinaryDigitsAndUnderscores", "BinaryDigitOrUnderscore",
        "FloatingPointLiteral", "DecimalFloatingPointLiteral", "ExponentPart",
        "ExponentIndicator", "SignedInteger", "Sign", "FloatTypeSuffix", "HexadecimalFloatingPointLiteral",
        "HexSignificand", "BinaryExponent", "BinaryExponentIndicator", "BooleanLiteral",
        "CharacterLiteral", "SingleCharacter", "StringLiteral", "StringCharacters",
        "StringCharacter", "EscapeSequence", "OctalEscape", "ZeroToThree", "UnicodeEscape",
        "NullLiteral", "LPAREN", "RPAREN", "LBRACE", "RBRACE", "LBRACK", "RBRACK",
        "SEMI", "COMMA", "DOT", "ASSIGN", "GT", "LT", "BANG", "TILDE", "QUESTION",
        "COLON", "EQUAL", "LE", "GE", "NOTEQUAL", "AND", "OR", "INC", "DEC", "ADD",
        "SUB", "MUL", "DIV", "BITAND", "BITOR", "CARET", "MOD", "ARROW", "COLONCOLON",
        "ADD_ASSIGN", "SUB_ASSIGN", "MUL_ASSIGN", "DIV_ASSIGN", "AND_ASSIGN",
        "OR_ASSIGN", "XOR_ASSIGN", "MOD_ASSIGN", "LSHIFT_ASSIGN", "RSHIFT_ASSIGN",
        "URSHIFT_ASSIGN", "Identifier", "JavaLetter", "JavaLetterOrDigit", "AT",
        "ELLIPSIS", "WS", "COMMENT", "LINE_COMMENT", "ErrorCharacter"
};

private static final String[] _LITERAL_NAMES = {
        null, "'abstract'", "'assert'", "'boolean'", "'break'", "'byte'", "'case'",
        "'catch'", "'char'", "'class'", "'const'", "'continue'", "'default'",
        "'do'", "'double'", "'else'", "'enum'", "'extends'", "'final'", "'finally'",
        "'float'", "'for'", "'if'", "'goto'", "'implements'", "'import'", "'instanceof'",
        "'int'", "'interface'", "'long'", "'native'", "'new'", "'package'", "'private'",
        "'protected'", "'public'", "'return'", "'short'", "'static'", "'strictfp'",
        "'super'", "'switch'", "'synchronized'", "'this'", "'throw'", "'throws'",
        "'transient'", "'try'", "'void'", "'volatile'", "'while'", null, null,
```

```
            null, null, null, "'null'", "'('", "')'", "'{'", "'}'", "'['", "']'",
            "';'", "','", "'.'", "'='", "'>'", "'<'", "'!'", "'~'", "'?'", "':'",
            "'=='", "'<='", "'>='", "'!='", "'&&'", "'||'", "'++'", "'--'", "'+'",
            "'-'", "'*'", "'/'", "'&'", "'|'", "'^'", "'%'", "'->'", "'::'", "'+='",
            "'-='", "'*='", "'/='", "'&='", "'|='", "'^='", "'%='", "'<<='", "'>>='",
            "'>>>='", null, "'@'", "'...'"
        };
        private static final String[] _SYMBOLIC_NAMES = {
            null, "ABSTRACT", "ASSERT", "BOOLEAN", "BREAK", "BYTE", "CASE", "CATCH",
            "CHAR", "CLASS", "CONST", "CONTINUE", "DEFAULT", "DO", "DOUBLE", "ELSE",
            "ENUM", "EXTENDS", "FINAL", "FINALLY", "FLOAT", "FOR", "IF", "GOTO", "IMPLEMENTS",
            "IMPORT", "INSTANCEOF", "INT", "INTERFACE", "LONG", "NATIVE", "NEW", "PACKAGE",
            "PRIVATE", "PROTECTED", "PUBLIC", "RETURN", "SHORT", "STATIC", "STRICTFP",
            "SUPER", "SWITCH", "SYNCHRONIZED", "THIS", "THROW", "THROWS", "TRANSIENT",
            "TRY", "VOID", "VOLATILE", "WHILE", "IntegerLiteral", "FloatingPointLiteral",
            "BooleanLiteral", "CharacterLiteral", "StringLiteral", "NullLiteral",
            "LPAREN", "RPAREN", "LBRACE", "RBRACE", "LBRACK", "RBRACK", "SEMI", "COMMA",
            "DOT", "ASSIGN", "GT", "LT", "BANG", "TILDE", "QUESTION", "COLON", "EQUAL",
            "LE", "GE", "NOTEQUAL", "AND", "OR", "INC", "DEC", "ADD", "SUB", "MUL",
            "DIV", "BITAND", "BITOR", "CARET", "MOD", "ARROW", "COLONCOLON", "ADD_ASSIGN",
            "SUB_ASSIGN", "MUL_ASSIGN", "DIV_ASSIGN", "AND_ASSIGN", "OR_ASSIGN",
"XOR_ASSIGN",
            "MOD_ASSIGN", "LSHIFT_ASSIGN", "RSHIFT_ASSIGN", "URSHIFT_ASSIGN", "Identifier",
            "AT", "ELLIPSIS", "WS", "COMMENT", "LINE_COMMENT", "ErrorCharacter"
        };
        public static final Vocabulary VOCABULARY = new VocabularyImpl(_LITERAL_NAMES,
_SYMBOLIC_NAMES);

        /**
         * @deprecated Use {@link #VOCABULARY} instead.
         */
        @Deprecated
        public static final String[] tokenNames;
        static {
            tokenNames = new String[_SYMBOLIC_NAMES.length];
            for (int i = 0; i<tokenNames.length; i++) {
                tokenNames[i] = VOCABULARY.getLiteralName(i);
                if (tokenNames[i] == null) {
                    tokenNames[i] = VOCABULARY.getSymbolicName(i);
                }

                if (tokenNames[i] == null) {
                    tokenNames[i] = "<INVALID>";
                }
            }
```

```
}

@Override
@Deprecated
public String[] getTokenNames() {
	return tokenNames;
}

@Override

public Vocabulary getVocabulary() {
	return VOCABULARY;
}



public Java8Lexer(CharStream input) {
	super(input);
	_interp = new LexerATNSimulator(this,_ATN,_decisionToDFA,_sharedContextCache);
}

@Override
public String getGrammarFileName() { return "Java8.g4"; }

@Override
public String[] getRuleNames() { return ruleNames; }

@Override
public String getSerializedATN() { return _serializedATN; }

@Override
public String[] getChannelNames() { return channelNames; }

@Override
public String[] getModeNames() { return modeNames; }

@Override
public ATN getATN() { return _ATN; }

@Override
public booleansempred(RuleContext _localctx, int ruleIndex, int predIndex) {
	switch (ruleIndex) {
	case 146:
		return JavaLetter_sempred((RuleContext)_localctx, predIndex);
	case 147:
		return JavaLetterOrDigit_sempred((RuleContext)_localctx, predIndex);
```

```
		}
		return true;
	}
	private booleanJavaLetter_sempred(RuleContext _localctx, int predIndex) {
		switch (predIndex) {
		case 0:
			return Character.isJavaIdentifierStart(_input.LA(-1));
		case 1:
			return Character.isJavaIdentifierStart(Character.toCodePoint((char)_input.LA(-2),
(char)_input.LA(-1)));
		}
		return true;
	}
	private booleanJavaLetterOrDigit_sempred(RuleContext _localctx, int predIndex) {
		switch (predIndex) {
		case 2:
			return Character.isJavaIdentifierPart(_input.LA(-1));
		case 3:
			return Character.isJavaIdentifierPart(Character.toCodePoint((char)_input.LA(-2),
(char)_input.LA(-1)));
		}
		return true;
	}

	public static final String _serializedATN =
		"\3\u608b\ua72a\u8133\ub9ed\u417c\u3be7\u7786\u5964\2n\u044c\b\1\4\2\t"+
		"\2\4\3\t\3\4\4\t\4\4\5\t\5\4\6\t\6\4\7\t\7\4\b\t\b\4\t\t\t\4\n\t\n\4\13"+
		"\t\13\4\f\t\f\4\r\t\r\4\16\t\16\4\17\t\17\4\20\t\20\4\21\t\21\4\22\t\22"+
		"\4\23\t\23\4\24\t\24\4\25\t\25\4\26\t\26\4\27\t\27\4\30\t\30\4\31\t\31"+
		"\4\32\t\32\4\33\t\33\4\34\t\34\4\35\t\35\4\36\t\36\4\37\t\37\4 \t \4!"+
		"\t!\4\"\t\"\4#\t#\4$\t$\4%\t%\4&\t&\4\'\t\'\4(\t(\4)\t)\4*\t*\4+\t+\4"+
		",\t,\4-\t-\4.\t.\4/\t/\4\60\t\60\4\61\t\61\4\62\t\62\4\63\t\63\4\64\t"+
		"\64\4\65\t\65\4\66\t\66\4\67\t\67\48\t8\49\t9\4:\t:\4;\t;\4<\t<\4=\t="+
		"\4>\t>\4?\t?\4@\t@\4A\tA\4B\tB\4C\tC\4D\tD\4E\tE\4F\tF\4G\tG\4H\tH\4I"+
		"\tI\4J\tJ\4K\tK\4L\tL\4M\tM\4N\tN\4O\tO\4P\tP\4Q\tQ\4R\tR\4S\tS\4T\tT"+
		"\4U\tU\4V\tV\4W\tW\4X\tX\4Y\tY\4Z\tZ\4[\t[\4\\\t\\\4]\t]\4^\t^\4_\t_\4"+
		"`\t`\4a\ta\4b\tb\4c\tc\4d\td\4e\te\4f\tf\4g\tg\4h\th\4i\ti\4j\tj\4k\t"+
		"\3\2\2\2\u0434\u0436\13\2\2\2\u0435\u0434\3\2\2\2\u0436\u0439\3\2\2\2"+
		"\u0437\u0438\3\2\2\2\u0437\u0435\3\2\2\2\u0438\u043a\3\2\2\2\u0439\u0437"+
		"\3\2\2\2\u043a\u043b\7,\2\2\u043b\u043c\7\61\2\2\u043c\u043d\3\2\2\2\u043d"+
		"\u043e\b\u0099\2\2\u043e\u0132\3\2\2\2\u043f\u0440\7\61\2\2\u0440\u0441"+
		"\7\61\2\2\u0441\u0445\3\2\2\2\u0442\u0444\n\27\2\2\u0443\u0442\3\2\2\2"+
		"\u0444\u0447\3\2\2\2\u0445\u0443\3\2\2\2\u0445\u0446\3\2\2\2\u0446\u0448"+
		"\3\2\2\2\u0447\u0445\3\2\2\2\u0448\u0449\b\u009a\2\2\u0449\u0134\3\2\2"+
		"\2\u044a\u044b\13\2\2\2\u044b\u0136\3\2\2\28\2\u028e\u0292\u0296\u029a"+
```

```
		"\u029e\u02a5\u02aa\u02ac\u02b0\u02b3\u02b7\u02be\u02c2\u02c7\u02cf\u02d2"+
		"\u02d9\u02dd\u02e1\u02e7\u02ea\u02f1\u02f5\u02fd\u0300\u0307\u030b\u030f"+
		"\u0314\u0317\u031a\u031f\u0322\u0327\u032c\u0334\u033f\u0343\u0348\u034c"+
		"\u035c\u0366\u036c\u0373\u0377\u037d\u038a\u0411\u041a\u0422\u042d\u0437"+
		"\u0445\3\b\2\2";
	public static final ATN _ATN =
		new ATNDeserializer().deserialize(_serializedATN.toCharArray());
	static {
		_decisionToDFA = new DFA[_ATN.getNumberOfDecisions()];
		for (int i = 0; i< _ATN.getNumberOfDecisions(); i++) {
			_decisionToDFA[i] = new DFA(_ATN.getDecisionState(i), i);
		}
	}
}

package jp.naist.se.simplecc;

import java.io.File;

import java.util.LinkedList;


public class DirectoryScan {

	public interface Action {
		public void process(File f);
	}


	public static void scan(File dirOrFile, Action action) {
		LinkedList<File> files = new LinkedList<File>();
		files.add(dirOrFile);
		while (!files.isEmpty()) {
			File f = files.removeFirst();
			if (f.isDirectory() &&f.canRead()) {
				File[] children = f.listFiles();
				for (File c: children) {
					if ((c.isDirectory() &&
						!c.getName().equals(".") &&
					!c.getName().equals("..")) || c.isFile()) {
						files.addFirst(c);
					}
				}
			} else if (f.isFile() &&f.canRead()) {
				action.process(f);
```

```
                    }
                }
            }

}
```