Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

ДИПЛОМНА РОБОТА

Удосконалення методів виявлення повторів та надлишковості програмного коду
Назва теми

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр ДРІПЗ.2001114.02.03.00 ПЗ

Виконав студент ІІ курсу група ІПЗм-20-2 _____ Н.І. Праворська
Підпис Ініціали, прізвище

Керівник д. ф.-мат. н., професор _____ Л.П. Бедратюк
Науковий ступінь, звання Підпис Ініціали, прізвище

Нормоконтролер к.т.н., доцент _____ О.М. Яшина
Підпис Ініціали, прізвище

До захисту допускаю:
Завідувач кафедри інженерії
програмного забезпечення _____ Л. П. Бедратюк
Підпис Ініціали, прізвище

9 грудня 2021 р.

Хмельницький 2021

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Програмування та комп'ютерних і телекомунікаційних систем

Кафедра Інженерії програмного забезпечення

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Л. П. Бедратюк
02 09 2021 р.

ЗАВДАННЯ

НА ДИПЛОМНУ РОБОТУ

Праворській Наталії Іванівні

Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Удосконалення методів виявлення повторів та надлишковості програмного коду

Керівник проекту (роботи) Бедратюк Леонід Петрович, доктор фіз-мат. наук, професор

Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 25.08.2021 р. № 102

2. Строк подання студентом проекту (роботи) на кафедру 04.12.2021 р.

3. Вихідні дані до проекту (роботи) Матеріали переддипломної практики

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити) Дослідження предметної області та постановка задачі, розробка мовно-незалежного інкрементного детектору повторів, дослідження локально-чутливого хешування в якості методу для розширення МНІДП, оцінювання продуктивності МНІДП у порівнянні з детектором комерційного рівня SIG по виявленню клонів, оцінювання ефективності ЛЧХ при порівнянні з МНІДП

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень) Презентаційні матеріали (слайди)

6. Консультанти розділів дипломного проекту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання « 01 » вересня 2021р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1 Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження; визначення структури дипломної роботи	01.092 – 07.09.2021	
2 Робота над розділом 1 дипломної роботи – вивчення літературних джерел, аналіз відомих методів, підходів та засобів за темою роботи; висновки до розділу й постановка задачі	08.09 – 25.09.2021	
3 Робота над розділом 2 дипломної роботи – розробка методів вирішення поставленої задачі; розробка МНІДП; висновки до розділу	26.09 – 10.10.2021	
4 Робота над науковими статтями	11.10 – 20.10.2021	
5 Робота над розділом 3 дипломної роботи розробка МНІДП, який буде використовувати метод ЛЧХ; висновки до розділу	11.10 – 26.10.2021	
6 Робота над розділом 4 дипломної роботи – оцінювання продуктивності МНІДП та ЛЧХ; висновки до розділу	27.10 – 15.11.2021	
7 Узгодження постановки задачі, отриманих результатів та висновків; написання вступу, загальних висновків, оформлення джерел посилання та додатків, оформлення пояснювальної записки та графічних матеріалів згідно вимог стандартів	06.11 – 30.11.2021	
8 Попередній захист ДР	Листопад (згідно графіка)	
9 Перевірка ДР на плагіат, нормконтроль, отримання відгуків та рецензій. Брошурування (зшиття) пояснювальної записки)	22.11.2021 – 04.12.2021	
10 Підготовка до захисту та захист ДР	06.12.2021-10.12.2021	

Студент _____ І.І. Праворська
Підпис Ініціали, прізвище

Керівник проекту (роботи) _____ Л.І. Бедратюк
Підпис Ініціали, прізвище

РЕФЕРАТ

Тема дипломної роботи: Удосконалення методів виявлення повторів та надлишковості програмного коду.

Автор роботи: Праворська Наталія Іванівна.

Керівник проекту: Бедратюк Леонід Петрович.

Пояснювальна записка: 103 с., 34 рис., 10 табл., 4 дод., 83 джерел.

МЕТОДИ ВИЯВЛЕННЯ ПОВТОРІВ, НАДЛИШКОВОСТІ, ПРОГРАМНИЙ КОД, ЛОКАЛЬНО-ЧУТЛИВЕ ХЕШУВАННЯ, КЛОНИ, ІНКРЕМЕНТНИЙ КРОК, ІНДЕКС КЛОНУ, МОВА ПРОГРАМУВАННЯ, C, JAVA, SOLIDITY-COVERAGE.

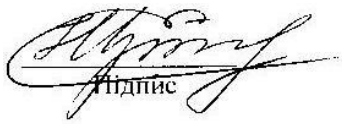
Метою роботи є розробка мовно-незалежного інкрементного детектору для виявлення повторів та надлишковості в програмному коді.

У дипломній роботі проаналізовано існуючі підходи та методи для виявлення повторів та надлишковості в програмному коді, які не залежать від мови програмування, розроблено мовно-незалежний інкрементний детектор. Досліджено придатність локально-чутливого хешування в якості методу, розширення та покращення первісно запропонованого підхід. Розроблено інкрементний детектор повторів, з використанням метод ЛЧХ. Проведена оцінка продуктивності МНІДП та порівняння її з підходом комерційного рівня SIG по виявленню клонів, для вивчення переваг, які може запропонувати інкрементний підхід. Виконано оцінювання ЛЧХ з точки зору ефективності, порівнюючи його з МНІДП та надаючи результати стосовно продуктивності цього розширення.

Для написання програмного коду мовно-незалежного інкрементного детектора було використано мову Java.

Практична значимість отриманих результатів полягає у розробці МНІДП, спроможного виявляти блоки клонів з точною відповідністю та визначено придатність підходу на основі ЛЧХ для розширення розробленого МНІДП.

22. 11. 2021р.
Дата


Підпис

ABSTRACT

Master's thesis: «Improving methods for detecting duplication and redundancy of program code»

Author: Natalia Ivanivna Pravorska.

Head of work: Leonid Petrovich Bedratyuk.

Master's thesis consist of: 103 p., 34 pic, 10 tab., 4 add., 83 srs.

METHODS OF DETECTION OF REPETITIONS, SURPLUSNESS, PROGRAM CODE, LOCALITY-SENSITIVE HASHING, CLONES, INCREMENT STEP, CLONE INDEX, PROGRMMING LANGUAGE, C, JAVA, SOLIDITY-COVERAGE

The aim of the work is to develop a language-independent incremental detector to detect repetitions and redundancies in the program code.

The thesis analyzes the existing approaches and methods for detecting repetitions and redundancies in the program code, which do not depend on the programming language, developed a language-independent incremental detector. The suitability of locally sensitive hashing as a method, extension and improvement of the originally proposed approach is investigated. An incremental repeat detector has been developed using the LSH method. The performance of MNIDP was evaluated and compared with the commercial SIG approach to clone detection, to study the benefits that the incremental approach can offer. The evaluation of LSH from the point of view of efficiency is performed, comparing it with MNIDP and providing results concerning the productivity of this expansion.

The Java language was used to write the code of the language-independent incremental detector.

The practical significance of the obtained results lies in the development of MNIDP, capable of identifying blocks of clones with exact correspondence and determined the suitability of the approach based on LSH for the expansion of the developed MNIDP.

22.11.2021 р.
Дата


Підпис

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ТБ	–	технічний борг
МНІДП	–	мовно-незалежний інкрементий детектор повторів
ЛЧХ	–	локально-чутливе хешування
BAP	–	Binary Analysis Platform
ГЗС	–	граф залежності системи
ГЗП	–	граф залежності програми
АСД	–	абстрактне синтаксичне дерево
УСД	–	узагальнене суфіксне дерево
LOC	–	Line Of Code (рядок коду)
LOCs	–	Lines Of Code (рядки коду)

ВСТУП

У 1992 році для пояснення довгострокових наслідків, спричинених швидкими та поспішними технічними компромісами задля здоров’я програмної системи, було введено термін технічного боргу (ТБ) [1] [2]. В цей же час з’явилася потреба у знаходженні компромісу між написанням короткострокового, швидкого та невпорядкованого коду за рахунок збільшення зусиль на технічне обслуговування у порівнянні з написанням довгострокового легшого у обслуговуванні, чистого коду підкріпленого цілеспрямованим мисленням [3].

Так зване дублювання коду, яке носить назву форми технічного боргу – є терміном, що описує дублювання вихідного коду в програмній системі [4]. У більшості випадків створення повторів та надлишковості (клонування) в коді відбувається доволі часто через простоту операції копіювання та вставки фрагментів коду розробниками. Також через те, що бізнес-логіка функції аналогічна або ідентична існуючому коду [5] [6]. Спираючись на факти минулих досліджень доведено, що значна кількість дублювання, виявлена в досліджуваних програмних системах становить від 7% до 29% у великих кодових базах [7] [8]. Результатом стає зниження ремонтопридатності основного програмного проекту.

Ряд різноманітних проблем в вихідному коді системи, виникає саме через наявність значної долі дублювання коду. Збільшення розміру кодової бази, і як наслідок, збільшення затрат на обслуговування, зокрема, є наслідком дублювання [9]. Крім цього, коли виникає помилка в одному з екземплярів блоку з повторами (клону) та надлишковістю, перевірці підлягає кожен інший блок на наявність тієї ж помилки та можливості потенційного виправлення [10]. Для вирішення останнього, треба не лише знати списки дубльованих блоків коду, а й володіти значною кількостю часу, потрібного для проходження по всім екземплярам. Нарешті, через дублювання виникають проблеми з точки зору розуміння програмного коду і ускладнень майбутнього рефакторінгу [11].

Важливо, що закладення основ для майбутнього ручного або автоматичного рефакторінгу, який призводить до більш чистого та зручного у супроводі коду, полягає в автоматичному виявленні клонів (блоків з повторами та надлишковістю) у сучасних програмних проектах. В цьому відношенні, різноманітні методи виявлення блоків з повторами, запропоновані на сьогодні, в основному працюють зі всією кодовою базою системи. Незалежно від величини внесених змін, подібні методи для кожної версії вихідного коду, використовують в якості вхідних даних всю систему. Такий підхід може добре працювати для стабільних застарілих систем, які зрідка оновлюються, але в сучасну еру, це не являється ідеальним варіантом, через гнучку розробку програмного забезпечення. В процесі виявлення блоків з повторами та надлишковістю для наступної перевірки, будуть виконуватися збиткові обчислення, які додаються до загального часу виконання.

Потреба в інкрементних підходах виникла через вказані вище недоліки, поряд з еволюцією практик розробки програмного забезпечення та появи таких концепцій, як безперервна інтеграція/розробка (CI/CD), гнучкість та швидкість. В цьому контексті основною ідеєю виступає повторне використання інформації, яка отримується в результаті аналізу одного перегляду, для наступного, виключаючи непотрібні неефективні за часом операцій.

При дослідження увага фокусувалася на SIG (група вдосконалення програмного забезпечення), діяльність якої стосується проблем, пов'язаних із якістю програмного забезпечення. Одна з основних сфер діяльності SIG відноситься до ремонтопридатності програмних продуктів. Більш конкретно, компанія розробила автоматизовані інструменти, за допомогою яких проводилася оцінка ремонтопридатності на основі ряду заздалегідь визначених критеріїв, таких як обсяг і складність коду, зв’язок модулів, тощо. Детектор повторів та надлишковості буде використано для виявлення такого критерію, як доля дубльованого коду в кодовій базі проекту. Враховуючи це, SIG дуже добре підходив для даного дослідження детектора повторів із реальною застосовністю слів.

В даному дослідженні представлено розробку незалежного від мови програмування інкрементного детектору повторів (клонів) та оцінка його продуктивності в контексті SIG. У відповідності з моделлю ремонтопридатності SIG проводиться вимірювання та оцінка якості коду [12], вимірюючи дублювання коду, як одну з властивостей власного коду. В роботі розроблена методика носить назву мовно-незалежний інкрементний детектор повторів (МНІДП).

У якості вхідних даних, в дослідженні використовуються програмні проекти різного розміру, під час оцінювання та порівняння запропонованого підходу з традиційним підходом виявлення клонів, який використовується SIG. Проведені експерименти показують покращення часу, необхідного для виявлення повторюваних фрагментів коду. Більш того, проводиться перевірка для подальшого розширення та покращення детектора МНІДП, який запропоновано спочатку, на предмет використання локально-чутливого хешування [13] – методу оцінки подібності найближчого сусіда. Виходячи з перших результатів, отриманих в дослідженні, виявлено, що підхід, заснований на локально-чутливому хешуванні (ЛЧХ), не відповідає характеристикам вихідного підходу, хоча присутня можливість для майбутніх досліджень з метою вивчення потенційних покращень.

Актуальність проблеми. Ремонтопридатність програмної системи знаходиться під сильним впливом дубльованого коду [12]. Виникнення надмірних обчислень відбувається через те, що традиційні методи використовують всю систему в якості вхідних даних, коли з плином часу аналіз доводиться часто повторювати. Кожен раз при цьому, доведеться користуватися комітом – операцією, в процесі якої відбувається фіксація набору змін, де вказано, який файл змінився і що саме в ньому змінилося, при чому ці зміни стають частиною основної ревізії сховища. Для кожної зміни коду, яка на практиці може бути пов’язана з комітом, при використанні традиційних підходів виникає необхідність повторного запуску процесу виявлення блоків з повторами (клонів). Обробка всієї програмної системи при цьому починається з нуля, незважаючи на величину змін, які відбулися, тобто було оновлено один файл або декілька рядків в декількох

файлах. Виникнення необхідності у більш детальних підходах на рівні комітів, пов’язано з тим, що пошук блоків з повторами та надлишковістю (клонів) у великій кодовій базі, може бути дуже дорогим як за часом, так і за вимогами пам’яті.

Для скорочення кількості непотрібних операцій та часу, необхідного для виявлення клонів, виникла ідея в застосуванні інкрементих методів, в яких повторно використовується інформація для різних версій. Хоча в минулому вже запропоновано ряд різних інкрементних підходів, переважна більшість з них мовно-залежні, тобто потребують синтаксичного аналізатора для обробки базового коду. З одного боку є можливість проведення більш глибокого аналізу та виявлення більш складних типів клонів (блоків з повторами та надлишковістю) більш високого рівня, але з іншого присутні обмеження, коли треба піддати обробці проекти, написані на непопулярних мовах програмування, для яких складною задачею стає створення парсеру або виконання пошуку. Більш того, якщо розгляд проводити в контексті SIG, мовно-незалежній підхід виявлення повторів (клонів) дозволяє знаходити дубльовані блоки в коді рівномірно на різних мовах програмування. На практиці, це дає змогу зробити мовно-незалежні висновки про ремонтопридатність. Наскільки відомо, то інкрементний повністю мовно-незалежний метод, ще не запропоновано. Дане дослідження дає змогу вияснити, як можна спроектувати та розробити такий мовно-незалежній інкрементний детектор повторів. Основою буде існуючий текстовий інкрементний підхід, первісно запропонований Хаммелом та ін. [14]. Хоча за замовчуванням підхід не є мовно-незалежним, проводиться його модифікація для досягнення поставленої задачі і отримання бажаного результату. Оцінювання детектору в контексті SIG та порівняння його з існуючим детектором компанії, дає змогу побачити переваги, які може запропонувати такий інктементний підхід. Крім цього буде досліджено ЛЧХ, як способу розширення такого підходу, намагаючись подолати деякі недоліки МНІДП. Далі відбудеться порівняння підходів один з одним, та дослідження потенціалу ЛЧХ в нашому контексті.

Сфера застосування. В роботі увага приділяється розробці мовно-незалежного інкрементного детектора повторів, дослідженню його ефективності та оцінюванню продуктивності. Для забезпечення можливості подальшого обслуговування програмної системи, а також для ідентифікації та видалення блоків дубльованого коду, необхідне виявлення клонів. Рефакторінг програмного коду (наприклад, шляхом об'єднання дублікатів до одного класу чи функції) є можливість проводити на основі вже існуючих клонів вручну або автоматично, і як результат усувати пов'язані з цим недоліки та роблячи кодову базу більш перспективною.

Мови та методи. Більш глибоке та детальне виявлення повторів та надлишковості в програмному коді можливе при зосередженні уваги на конкретній мові програмування, особливо на такій популярній, як Java [15]. Знаходження семантично схожих блоків коду з повторами є прикладом підходу, який дозволяє знаходити все більш складні клони. Для цього виконується дослідження різних методів виявлення, таких як: на основі токенів, на основі дерев, на основі графів і т.п. Однак, оскільки SIG не має обмеження стосовно підтримуваних мов програмування, то пошук чи створення аналізатора для менш популярних мов, виявляється доволі складною задачею. До того ж, в контексті дослідження фіксування відбувалося на мовно-незалежних детекторах, створення яких можливе тільки з використанням текстових методів.

Типи блоків коду з повторами та надлишковістю (клонів коду). Розрізняють чотири різні типи клонів. Їх зазвичай називають типом 1, типом 2 і так далі до типа 4 [5] [6] [11]. В дослідженні вся увага була зосереджена на блоках з повторами (клонами 1-го типу), які відносяться до фрагментів коду повністю ідентичних. Основою вибору для досягнення кінцевої мети, тобто проведення порівняння отриманих результатів дослідження, стало використання існуючого сучасного підходу, яким користується SIG. Тому мовно-незалежний інкрементний детектор повторів буде проектуватися та розроблятися з урахуванням клонів (блоків з повторами та надлишковістю) тільки 1-го типу.

Керування блоками з повторами та надлишковістю (клонами). Під час дослідження не проводилось обговорення та підтримка керування блоками коду з повторами та надлишковістю (клонами) [16] [17]. Це означає, що не відбувалося відслідковування того, як йшов розвиток клонів в різних версіях основного програмного проекту. Навпаки, увага приділялася тому, що відбувалося в контексті конкретного коміта, тобто тим клонам, які вводилися чи видалялися конкретною редакцією коду.

Оцінка високого рівня. Два запропонованих інкрементних детектори блоків з повторами та надлишковістю не порівнювалися з існуючими сучасними інкрементними методами. Метою дослідження не було виявлення на скільки ефективними виявляються підходи у порівнянні з іншими відповідними дослідженнями. Навпаки, для МНІДП треба було провести оцінювання продуктивності та вивчення її у порівнянні з сучасним підходом SIG для виявлення клонів. До того ж, з точки зору запропонованого в дослідженні підходу, заснованого на ЛЧХ, вивчалося, чи можливо його використовувати для покращення продуктивності. Обидва інструменти є загальнодоступними[1], таким чином є можливість проведення додаткових досліджень для вивчення підходів з різних точок зору.

Інкрементні виявлення. Основною метою даного дослідження виступає розробка та оцінка мовно-незалежної інкрементної техніки виявлення блоків коду з повторами та надлишковістю (клонів). Для цього на початку надається представлення про сучасний стан літератури в галузі інкрементного виявлення клонів та досліджуються сучасні текстові методи. Далі відбувається розробка детектора повторів, який буде відповідати поставленим в дослідженні вимогам. Тобто, детектор, який незалежно від мови програмування зможе працювати інкрементно і виявляти блоки з повторами та надлишковістю (а саме, клони 1-го типу).

1 Обидва інструменти доступні на Github: https://github.com/agamvrinos/LIICD

Розширення ЛЧХ. Для досягнення поставленої в дослідженні мети, виконується додаткове розширення запропонованого інкрементного детектора з використанням локально-чутливого хешування (ЛЧХ). Це метод, який використовується для ефективного пошуку найближчих сусідів. Практично, в нашому випадку, це означає ефективну оцінку подібності між вихідними файлами. Тобто, проводиться порівняння останнього з запропонованим детектором МНІДП, висвітлення отриманих результатів та надання ідей для подальших експериментів.

Оцінка. Маючи інкрементні підходи на основі МНІДП та ЛЧХ, проводиться їх оцінювання. Для запропонованого детектора МНІДП спочатку іде вимір його продуктивності, а потім оцінювання її у контексті SIG. Конкретніше, відбувається ізоляція процесу виявлення блоків з повторами (клонами) інструменту SAT SIG. Даний інструмент використовується SIG для проведення аналізу якості програмного проекту та вимірювання різних показників, таких як його ремонтопридатність – і порівняння з запропонованим в дослідженні підходом. Для детектора на базі ЛЧХ проводяться експерименти та його оцінювання, порівнюючи його продуктивність та МНІДП.

Об’єкт дослідження – вихідні програмні коди, в яких можуть бути повтори та надлишковості.

Предмет дослідження – методи виявлення повторів й надлишковості програмного коду та їх розширення.

Мета роботи – удосконалення методів виявлення повторів й надлишковості програмного коду

Основними завданнями роботи виступають:

– дослідження мовної незалежності в контексті інкрементного виявлення блоків програмного коду з повторами та надлишковістю (клонів) і розробка МНІДП. Основою даного інструменту виступає існуюча методика інкрементного текстового виявлення клонів, розроблена Хаммелом та ін. [14];

– дослідження придатності ЛЧХ в якості методу, який можна використати з метою розширення та покращення первісно запропонованого підходу;

– розробка інкрементного детектора повторів, який буде використовувати метод ЛЧХ;

– оцінювання продуктивності МНІДП та порівнянні її з підходом комерційного рівня SIG по виявленню клонів, для вивчення переваг, які може запропонувати інкрементний підхід;

– оцінювання ЛЧХ з точки зору ефективності, порівнюючи його з МНІДП та надаючи результати стосовно продуктивності цього розширення.

Наукова новизна – удосконалення методів виявлення повторів та надлишковості програмного коду, використовуючи інструмент аналізу незалежний від мови програмування за рахунок текстового інкрементального підходу.

Практична цінність:

– спроектовано та розроблено МНІДП, спроможного виявляти блоки клонів з точною відповідністю;

– визначено придатність підходу на основі ЛЧХ для розширення розробленого МНІДП.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

Існування значного числа повторів та надлишковості у вихідному коді тягне за собою ряд різноманітних проблем. Наслідком збільшення розміру кодової бази, який виникає через дублювання блоків коду, стає підвищення витрат на обслуговування [11]. При цьому, якщо помилка була виявлена в одному з продубльованих фрагментів коду, кожен інший екземпляр має пройти перевірку на наявність тієї ж помилки та бути виправлений [18]. Це буде вимагати не тільки знання усіх клонів у списку, а й достатньо великих часових витрат для проходження через усі екземпляри. Нарешті, повторення та надлишковість не сприяє розумінню програмного коду, який може в майбутньому піддаватися рефакторингу.

1.1 Підходи, використовувані при аналізі виконуваного коду

При розборі робіт, які є на сьогодні актуальними в питаннях аналізу виконуваного коду, спостерігається наступна тенденція, проводиться розбиття аналізу на: статичний та динамічний. Статичний аналіз – це аналіз, при якому виконується дослідження програмного коду продукту без його реального виконання, а також він в собі містить методи аналізу потоку керування, абстрактної інтерпретації, а також аналізу потоку даних. Сюди ж відносяться методи, які використовують символьне виконання. При цьому статичними аналізаторами використовуються наступні конструкції (проміжні представлення програми), а саме: графи виклику функцій, графи потоку керування, абстрактні синтаксичні дерева, графи залежностей програми і системи і т.д.

Динамічний аналіз – це аналіз, який дозволяє виконувати дослідження виконуваного коду підчас і після виконання програми. Динамічні аналізатори при такому підході потребують інформацію про вхідні дані, та ефективність таких

аналізаторів напряму буде залежати від кількості і якості отриманих даних. До динамічного аналізу прийнято відносити: динамічне символьне виконання, тестування програмного забезпечення (фаззінг), відладку та профілювання.

Кожний з вище згаданих підходів має як свої переваги, так і недоліки. При статичному аналізі треба використовувати всі можливі шляхи виконання та всі значення змінних. Таким чином, статичній аналіз має змогу виявити дефекти на відміну від динамічного аналізатора, навіть, якщо такий використовувався доволі довгий час. Тобто динамічний аналізатор може виявляти подібне лише в тому разі, якщо виконуваний шлях пройшов через точку дефекту, при деяких значеннях змінних. На відміну від динамічних аналізаторів статичним, зазвичай, не треба інформації при вхідні дані. Окрім цього, перші потребують використання емуляторів чи апаратури, якій притаманна архітектура виконуваного коду, що аналізується. Але при цьому, в разі роботи статичних аналізаторів, можливі хибні спрацювання, їх відсоток складає приблизно 20-70 одиниць [19] через неповне відновлення інформації про програму (для прикладу, виклик віртуальних функцій). А також неможливість не тільки використання всієї інформації, яка отримана під час проведення аналізу, а й обробки цієї інформації.

Однак незважаючи на ці недоліки в життєвому циклі розробки програмного забезпечення статичний аналіз виконуваного коду дуже широко використовується. За допомогою статичних аналізаторів спеціалісти та компанії, які займаються розробкою ПЗ, можуть не тільки знаходити дефекти в кодах, але й суттєво підвищити надійність розроблюваних програмних продуктів.

1.2 Огляд найбільш поширених методів статичного аналізу виконуваного коду програм

Спираючись на інформацію роботи Кіндера [20], який провів розробку і реалізацію для здійснення аналізу виконуваного коду архітектури x86, основними властивостями системи виступають:

– аналіз інтервалів значень виконується на основі моделі пам’яті (згідно роботи [21]). Символьною базовою адресою виступає ідентифікатор регіону, яким помічені значення, присвоєні регістрам та областям пам’яті. Проводиться ідентифікація таким чином виказувачів на область глобальної пам’яті, стек та кучу, та передбачується, що їх перекриття не відбувається. Збіжність алгоритму при цьому забезпечується обмеженням кількості значень для кожної змінної для кожного місцерозташування;

– використання проміжної мови. В послідовність інструкцій проміжної мови відбувається транслювання складних машинних інструкцій. Проводиться заміна переходом без умов під час трансляції як виклику функцій, так і повернення з функцій. Це дозволяє подолати деякі перетворення коду, які можуть приводити до заплутувань;

– дезасемблювання за вимогою. За один раз відбувається дезасемлювання тільки однієї інструкції, замість того, щоб намагатися провести їх дезасемлювання якомога більше. Під час абстрактної інтерпретації відбувається трансляція тільки інструкції, яка відповідає наступному етапу виконання. За допомогою такого методу є змога долати перекриваючі інструкції, оскільки не треба фіксованого представлення, при якому відбувається співставлення кожного байту в одній команді. В залежності від контексту виконання проводиться інтерпретація одних і тих самих байтів як різних інструкцій;

– проектування потоку керування. Інтегрований метод аналізу потоку керування та потоку даних на основі абстрактної інтерпретації запропоновано для оптимального розв’язку непрямого гілкування.

Інструмент Jakstab містить реалізовані методи. Відбувається аналіз платформою виконуваних файлів не тільки Windows, а й Linux архітектури x86.

BAP – є платформою бінарного аналізу з відкритим кодом, а також виступає фреймворком, для якого немає потреби змінювати код для розширення функціональності, а достатньо реалізації свого плагіну. За допомогою BAP [22] можливо проводити аналіз виконуваних файлів архітектури x86 та ARM. На першому етапі проводиться дизасемблювання виконуваного коду на основі

лінійного алгоритму. Другий етап характеризується отриманням проміжного представлення, при якому не має залежності від архітектури та на існує побічних ефектів. Статичне єдине присвоювання [23] виступає формою проміжного переставлення, завдяки якому виконується аналіз та оптимізація, відбувається побудова DEF-USE, USE-DEF ланцюги та знищення мертвого коду (підтримки з пам'яттю немає, враховуються тільки флаги й регістри).

В роботі [24] проводиться огляд BitBlaze (аналогічної BAP) – системи, призначеної для зберігання даних за рахунок високоефективної системи охолодження, до складу якої входить:

– Vine – компонент статичного аналізу;

– TEMU – компонент динамічного аналізу;

– Rudder – об'єднання 1-го та 2-го аналізу, а саме конкретного та символьного.

Система може дозволяти виконувати наступні дії: перше – асемблер можна отримати з дизасемблера IDA Pro (інтерактивний дизасемблер, маючий широке використання в сфері реверс-інжинірінгу); друге – проводиться транслювання асемблера в проміжне представлення VEX IL (високорівнева мова); третє – транслювання VEX IL на VINE IL (низькорівнева мова).

В праці було проведення адаптації аналізу значень з [21] для представлення у мові низького рівня (VINE IL). Базуючись на отриманих даних відбулися наступні види аналізу: аналіз потоку даних (аналіз активних змінних; розповсюдження констант; видалення мертвого коду); в графі потоку керування (ГПК) проводиться відновлення непрямих викликів та переходів; генерація С-коду з мови низького рівня (VINE IL); генерація графів залежностей програми.

В роботі Балакришнана та Репса [21] представлений метод аналізу інтервалів значень, потрібний для аналізу виконуваного коду з архітектурою x86, використаний в системі CodeSurfer/x86 (проведення аналізу виконуваного файлу без вихідного коду). По-перше виконуваний файл проходить дизасемблювання з використанням IDA Pro [25] з використанням інформації, яка відновилася за допомогою IDA Pro (статично відомі адреси пам'яті та зміщення, графи функцій

та потоку керування, а також виклики до функцій стандартних бібліотек (для ініціації використовується алгоритм Fast Library Identification and Recognition Technology – F.L.I.R.T [26]). Було створено пагін Connector для IDA Pro, за посередництвом якого будувалися структури даних для представлення інформації, яка надходила від IDAPro. В плагіні, ґрунтуючись на структурі даних, відбувалася розробка алгоритму для аналізу інтервалів значень, доповнюючи та виправляючи інформацію, відновлену IDAPro. Система CodeSurfer/x86 враховуючі непрямі переходи проводить поліпшення графів потоку керування та графу виклику функцій. Побудова власної колекції проміжних представлень, яка складається з графа залежностей системи (ГЗС), абстрактних синтаксичних дерев, графа виклику функцій, графа потоку керування, відбувається на основі отриманої інформації [27]. Всі графи залежності програми (ГЗП) кожної функції складають ГЗС. Інструкції в програмі виступають вершинами ГЗП, відповідно залежності по даним та по керуванню між інструкціями стають ребрами графу. Вершини між собою з’єднуються між процедурними ребрами, яким відповідають залежності даних між фактичними параметрами та формальними параметрами (або значеннями, які повертаються), а також залежності потоку керування між викликами функцій.

Інформація відновлюється спираючись на розроблену абстрактну модель і представляє собою дані про глобальні і локальні змінні, вказівники та структури, масиви та об’єкти з класів (та підоб’єкти з підкласів), непрямі переходи та непрямі виклики через виказувачі функцій. На основі абстрактної інтерпретації авторами проекту були розроблені декомпілятор та алгоритм аналізу аліасів (Alias Analysis – не є методом оптимізації, але дає змогу підвищити ефективність інших алгоритмів оптимізації і призначений для відслідковування звернень за вказівником до тієї чи іншої змінної) [28].

Статичний аналізатор GUEB, розглянутий в праці [29] виконує виявлення використання після вивільнення двійкових файлів, тобто відбувається знаходження дефектів застосування пам’яті після того, як відбувається вивільнення та подвійного вивільнення в виконуваному коді. Даний аналізатор на

початку відстежує дії по отриманню та видаленню пам’яті в кучі, а згодом передачу цих даних. Також іде врахування GUEB аліасів, реалізованих на основі аналізу значень. Згодом вся отримана інформація буде використана для виявлення дефектів. Останній етап складається з отримання підграфів ГПК. Вони дають змогу мати інформацію про те, в якій точці програми відбувалося вивільнення пам’яті та за посередництвом яких інструкцій проходила передача інформації.

Метод пошуку дефектів для виконуваного коду x86, представлений у роботі [30], оснований на аналізі позначених даних, з використанням методу статичного символьного виконання.

Проміжне представлення REIL [31] – яке являє собою проміжну мову оберненого проектування і використовується в якості платформи для пошуку критичних дефектів, розглянуто в [32]. За посередництвом IDA Pro відбувається дизасемблювання, згодом інформація поступає в BinNavi [33] для генерації REIL-представлення. На наступному кроці відбувається трансляція REIL у eREIL, яке є розширенням представлення REIL, але з додаванням нових інструкцій. В праці авторами відбулася адаптація аналізу інтервалів значень [21] для представлення в eREIL. Основою платформи виступає міжпроцедурний аналіз. Внутрішньо процедурний аналіз збирає результати всіх значень програми окрім локальних змінних з використанням аналізу інтервалів. Аналіз позначених даних і аналіз залежностей за даними втілені у платформі. Авторами реалізований пошук дефектів форматного рядка та деяких типів переповнення буферу спираючись на аналіз позначених даних.

Наступний метод аналізу базується на міжпроцедурному аналізі доступних виразів. В ньому для кожного блоку проходить визначення виразів, які вбиваються або генеруються. Тут же відбувається обробка впливу викликів функцій на доступний вираз. Попередження про дефект видається в тому разі, якщо вираз не доступний, але все одно використовується. Цей метод статичного аналізу для пошуку дефектів використання пам’яті після вивільнення в виконуваному коду програмного продукту з архітектурою x86 описаний в праці [34].

Пошук дефектів форматного рядка в виконуваних файлах x86, закладено в статичний метод [35]. За розробленим алгоритмом відбувається аналіз позначених даних, знаходження простих випадків дефектів форматного рядка всередині однієї функції.

1.3 Огляд методів пошуку клонованих частин виконуваного коду

Нерідко намагаючись пришвидшити написання програмного коду розробники вдаються до копіювання його частин, вставляючи блоки у потрібне місце. Це може не тільки призводити в подальшому до різноманітних помилок, але й до суттєвого збільшення як вихідного, так і виконуваного коду. Спираючись на дані досліджень [9] [36], виявилося, що біля 20 % коду виявляються подібними фрагментами (тобто є клонами). На сьогодні існує широке коло методів для пошуку подібних блоків вихідного коду (а саме клонів) [5], [6], [11], [37]-[40]. При цьому треба відмітити, що може відбуватися створення клонів виконуваного коду компілятором, шляхом копіювання деяких частин, яких немає навіть в початковому коді. Саме виявлення повторів (клонованих частин) виконуваного код лежить в основі пошуку не тільки шкідливих програм і семантичних помилок, а й порушення авторських прав й т.п.

Клони коду можна знайти в різних формах і не всі однакові. Проводиться наступний поділ клонів виконуваного коду:

– повністю співпадаючі фрагменти коду (точна копія), з урахуванням відмінностей лише в пробілах, пробілах і коментарях;

– синтаксично ідентичні фрагменти коду, які можуть відрізнятися за типами, значеннями даних, іменами регістрів, варіаціями ідентифікаторів, літералів, коментарів;

– синтаксично ідентичні фрагменти коду, які можуть бути відмінними за типами, значеннями даних, іменами регістрів, а також деякими інструкціями, які можуть бути присутніми або відсутніми у конкретному фрагменті;

– семантично подібні фрагменти коду, зазвичай ідентифікуються, як клони тупу 4. Для прикладу, реалізація за допомогою перемикача «switch» функціональності оператора «if».

Для прикладу, обирається одна з найбільш популярних мов програмування Java [15], завдяки якій є можливість більш глибокого та детального виявлення повторюваних блоків вихідного коду (клонів). До того ж існує підхід, який дозволяє знаходити більш складні повторювані блоки, як семантично подібні клони. В такому разі кращим було б дослідити різні методики виявлення, а саме: граф, токен, дерево тощо.

Приклад для кожного з цих типів клонів можна побачити в лістінгах, представлених на рис. 1-5. Приклади показують, що чим вищий тип клону, тим важче виявити відповідні блоки з такими повтореннями (клонами).

```
1 int myFunc(int n) {
2     int sum = 0; // cm1
3     int a = 2;
4     for (int i = 0; i <=n; i++){
5         sum = sum + a;
6     }
7     return sum;
8 }
```

Рисунок 1 – Лістінг оригінального фрагменту коду

```
1 int myFunc(int n) {
2     int sum = 0; // cm1
3     int a = 2; // cm2
4
5     for (int i = 0; i <= n; i++){
6         sum = sum + a;
7     }
8     return sum;
9 }
```

Рисунок 2 – Лістінг клону типу 1

```
1 float myFunc(int n) {
2     float summary = 0; // cm1
3     int a = 2; // cm2
4
5     for (int i = 0; i <= n; i++){
6         summary = summary + a;
7     }
8     return summary;
9 }
```

Рисунок 3 – Лістінг клону типу 2

```
int myFunc(int n) {
    int sum = 0; // cm1
    // deleted line
    for (int i = 0; i <= n; i++)
        sum = sum + 2;
        i++; // new line
    }
    return sum;
}
```

Рисунок 4 – Лістінг клону типу 3

```
int myFunc(int n) {
    int sum = 0; // cm1
    int a = 2;
    int i = 0;
    while (i <= n) {
        sum = sum + a;
        i++;
    }
    return sum;
}
```

Рисунок 5 – Лістінг клону типу 4

Левова доля сьогоднішніх методів, використовують звичайний спосіб в підході виявлення повторюваного коду. Однак, в останні роки, все більше набувають популярності додаткові методи, які проводять аналіз вихідного та виконуваного коду на більш дрібному рівні, причому є намагання уникнути зайвих перерахунків, завдяки передачі інформації між версіями. Незважаючи на підходи, методи класифікують за представленням вихідного коду на такі основні категорії:

– текстові підходи. Проводиться розгляд фрагмента коду з позиції текстової рівності, складні перетворення в вихідному коді не застосовуються. Результатом є широке охоплення повторюваних блоків (клонів 1 типу), а також виявлення подібних дефектів незалежно від основної мови [6]. Негативним аспектом виступає те, що при таких підходах немає спроможності виявляти більш складні, вищі типи клонів. Інструменти, які використовують текстовий підхід розглядають блок виконуваного коду, як послідовність байтів та рядків коду асемблера, порівнюючи кожну пару фрагментів коду для здійснення пошуку ідентичних послідовностей;

– підходи, засновані на токенах. Подібні методи носять назву лексичних. Інструменти, в основі яких лежить цей підхід, проводять дизасемблювання виконуваного файлу. Асемблер, який отримано в результаті, розбивається на коди операцій та операндів. Послідовність, яка складається з узагальнених або

відфільтрованих типів кодів і операндів, перевіряється з метою виявлення повторюваних блоків коду.

У праці [41] Шульманом була запропонована система, спроможна відшукувати клони функцій у виконуваному файлі. Це була перша робота, яка давала змогу виявлення повторюваних блоків коду на рівні функцій. Реалізована система має під собою наступну ідею – створення хешу кожної з функцій в виконуваному файлі, який підлягає аналізу, а згодом збереження її у базі даних. Наявність клонів в коді підтверджується ідентичними хеш-значеннями. При обчислення хеш-значень основою яких виступатимуть opstring'и (аргументи функцій) необхідно врахувати наступне:

– в хеші використовують не код операнду, а тільки код операції інструкції;

– у відповідну мнемоніку проводиться перетворення кодів операцій. Приклад, команда push ecx – виступає кодом операції, тому push присвоюється opstring'гу замість числового значення;

– результат буде більш точним, якщо в opstring будуть додані мітки місце розташування;

– проводиться нормалізація викликів системних функцій (формат ASCII чи UNICODE не враховується).

В праці [42] пропонується застосування алгоритму, спроможного виявляти повторювані блоки виконуваного коду при групування шкідливих програм. Клонами при проведення огляду окремих частин блоків коду, прийнято вважати фрагменти, які мають подібні шматки.

Створення типів (моделей) шкідливого ПЗ, які базуються на особливостях їх виконуваного коду та виявлення таких типів в файлах, призначених для тестування, стало метою даного проекту. Принцип переупорядкування коду, а саме переупорядкування команд чи блоків, покладено в основу побудови подібних моделей. Алгоритми, базовані на цьому підході, можуть виявляти в програмному коді клони 1 і 2 типу.

Лексичні аналізатори використовують послідовність токенів отриманих з вихідного коду в процесі – лексичного аналізу. Послідовності токенів порівнюють

між собою, а не послідовності вихідного коду. При виявленні повторень з незначними відмінностями (клони 2-го типу) ці підходи виявляються дуже ефективними. Прикладом виступають перейменування [8].

Синтаксичні підходи поділені на:

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

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

Семантичні підходи. Проводиться розподіл на два підкласи: графічно-орієнтовані і гібридні методи. При застосуванні програмованого аналізатора відбувається не синтаксичне порівнювання двох блоків коду, а семантичне. Клон буде виявлено порівнянням підграфів, при графічно-орієнтованому методі, шляхом побудови ГЗП з вихідного коду. Комбінування вищезгаданих технік виступає основою гібридного методу.

Підхід, заснований на поведінці програми. Система Reanimator, яка представлена в роботі [43], дає змогу ідентифікувати у шкідливих програмах їх приховані функціональності. Система базується на факті, що при динамічному аналізі шкідливих програм, відбувається фіксація виконання шкідливого програмного забезпечення та отримується повідомлення про його поведінку. Задачею виступає запуск великої кількості шкідливих програм та визначення їх динамічно різних функціональних можливостей. При подальшій ідентифікації цих можливостей, відповідні функції зразу виявлятимуться в нових шкідливих програмах. Система Reanimator працює в три кроки. Створення функціональної моделі відповідної для різних моделей поводжень відбувається на перших двох етапах. Перевірка прихованих поводжень на основі побудованої моделі

проводиться на третьому етапі. Методи, які використовують вказаний алгоритм, дають змогу виявити клони 1-3 типів.

1.4 Оцінки метрик

В області виявлення повторів на надлишковості програмного коду детектори клонів зазвичай прийнято порівнювати, використовуючи три основних показники (оцінки): відгук, точність та масштабованість [8], [44].

Відгук. Вказує на долю копій коду в базі коду, яку спроможний виявити детектор клонів. Вимірювання досить складні, оскільки треба для порівнювання мати еталонний показник. Відгук обчислюється за наступною формулою (1) [10]:

$$\text{Відгук} = \frac{\text{Кількість правильно виявлених клонів}}{\text{Кількість виявлених клонів}} \quad (1)$$

Точність. Вказує на частку повторів, які знаходив детектор і які виявлялися справжніми клонами, а не помилковими. Вираз, який дає можливість розрахувати точність в контексті виявлення повторів за формулою (2) [10]:

$$\text{Точність} = \frac{\text{Кількість правильно виявлених клонів}}{\text{Кількість загальновиявлених клонів}} \quad (2)$$

Масштабованість. Вказує на вимоги до часу і простору базового детектора клонів, забезпечуючи йому можливість масштабування до більш великих баз, які місять коди.

1.5 Проблеми виявлення повторів та надлишковості програмного коду

При проектуванні та розробці детектора виявлення блоків з повторами та надлишковістю (а саме, клонів) зазвичай виникає рід питань, які треба

розв’язувати. Одним з найважливіших є те, що при розробці інструменту для виявлення клонованих блоків програмного коду, треба, щоб була змога виявляти кілька типів клонів. Має бути забезпечена водночас і висока точність та відгук при застосуванні на практиці [6]. Особливо складною задачею постає виявленні клонів 4 типу (семантичних клонів), яка в існуючий на сьогодні літературі представлена в якості нерозв’язної проблеми [11]. Коли справа доходить до проведення аналізу великих кодових баз, то велика кількість інструментів погано масштабуються [45]. Наступною проблемою, з якою стикаються розробники засобів виявлення дубльованих блоків коду, виступає портативність, а саме спроможність інструментом з легкістю виявляти повтори в програмних кодах, написаних на різних мовах програмування [6].

1.6 Технології виявлення повторів та надлишковості програмного коду

Даний розділ присвячений огляду різноманітних традиційних і додаткових методів, які рекомендуються на протязі багатьох років.

1.6.1 Традиційні техніки

Під традиційними методами виявленні повторюваних та надлишкових фрагментів програмного коду вважаються такі, основою яких виступає використання всієї кодової бази програмного продукту в якості вхідних даних для виявлення клонів. Розмір змін внесених до конкретної редакції коду в даному випадку, не враховується. Текстові методи, представлені нижче, вважаються найбільш підходящими:

– в праці [46] Duploc, де при побудові повторюваних блоків коду використовується «точкова матриця», після проводиться виявлення клонів, виконуючі аналіз їх при пошуку шаблонів;

– в роботі Джонсона та ін. [47] – аналіз виконується незалежно від мови програмування виявленні клонів шляхом співставлення підрядків;

– в праці Маркуса та ін. [48] – для визначення семантично подібних блоків програмного коду та проведення аналізу кодової бази програмної системи залучається статичний аналіз.

Найбільш обговорюваний підхід, який зустрічається в літературі, стосовно методу виявлення клонів заснованого на токенах вважається підхід CCFinder [49]. Авторами запропоновано використання залежних від мови програмування принципів перетворення для маркування (токенізації) необробленого вихідного коду. Для фактичного виявлення повторень та надлишковості запропоновано алгоритм співставлення дерева суфіксів.

Цікавими виявилися наступні методи:

– Dup [50] – для отримання можливості виявлення клонів 1 та 2 типу, використовується спеціальний тип структури даних, а саме параметризоване суфіксне дерево;

– CP-Miner [18] – для ефективної ідентифікації скопійованого вихідного коду, використовуються частий подальший аналіз даних та метод інтелектуального аналізу даних.

Новітнє дослідження, таке як CCLearner [51] вивчає токенізацію у поєднанні з глибоким навчанням, подібне зустрічається і у праці Вайта та ін. [52]

Найбільш поширеними в галузі деревоподібних та метричних методів виступають:

– CloneDr [53] – за допомогою генератора компілятора відбувається генерування анотованого дерева синтаксичного аналізу. Використовуючи метрики характеристик проходить процес виявлення піддерев та їх порівняння;

– Deckard [54] запропонована техніка для виявлення клонів. Для ефективної кластеризації характеристик подібних векторів, з використанням комбінації АСД та ЛЧХ, тобто хешування з урахуванням місцерозташування;

– в роботі Мейранда та ін. [55] – виступає метрико-орієнтованою технологією, при якій відбувається розрахунок різноманітних метрик за іменами, макетами, виразами і тому подібними функціями. Повтори в коді будуть знайдені шляхом виявлення функцій, які мають однакові значення для обчислених метрик;

– у праці Контоянеса та ін. [56] – виявлення повторень в коді пропонується двома методами. При першому методі проводиться чисельне порівняння значень, які отримані за допомогою відомих метрик, та виконується класифікація фрагментів коду по блоках від початку і до кінця. В другому методі залучається динамічне програмування для проведення обчислень та створення звітів про початкові та кінцеві блоки з використанням мінімальної відстані редагування.

Серед найбільш відомих прийомів, заснованих на графах виділяють:

– Duplix [57] – є здатність знаходити максимально подібні графи ГЗП з високою точністю та відгуком;

– запропонований у праці Комондора та ін [58], має таку саму здатність, як попередній детектор;

– GPLag [59] – використовує майнінг (буквально, діяльність, направлена на створення нових структур, спроможних забезпечувати функціонування платформ криптовалюти) ГЗП з метою виявлення плагіату. Має здатність виявляти код плагіату, який був навмисно замаскований, чого не можуть подібні інструменти.

Гібридними підходами виступають наступні:

– у праці Фанореза та ін. [60] представлено метод, який об’єднує АСД для ідентифікації клонів коду та текстовий підхід для усунення помилкових спрацювань;

– в роботі Егравела та ін. [61] для виявлення клонів типу 1-3, запропоновано поєднання підходів на основі токенів з підходами на основі тексту, використовуючи переваги кожного з них відповідно до типу.

Детектор Oreo, представлений в роботі Сайні та ін. [62], дає змогу з високою точністю та відгуком виявляти клони тупі 1 та клони тупу 4. Проводиться пошук програмної інформації та програмних метрик, а також машинне навчання.

1.6.2 Додаткові методи

На відміну від досліджень, які використовують інструменти повного системного аналізу, розробок, що пропонують додаткові методи є небагато. Переважно вони зосереджуються на методах, основою яких становлять токени, дерева або графіки. Це буде вимагати від синтаксичного аналізатора або парсера проводити побудову необхідних структур даних спираючись на відповідну мову (мови) програмування системи. Фактично з самого початку більшість з них не мають намір підтримувати незалежність від мови програмування. Таким чином вони розробляють детектори, які з одного боку спроможні виявляти більш складні типи клонованого коду (блоків коду з повторами та надлишковістю), але з іншого потребують налаштування для конкретної мови програмування.

Натомість, вперше була запропонована інкрементна методика виявлення клонів авторами роботи Гоуд та Кошке [63]. Для представлення вихідного коду було рекомендовано задіяти деревоподібну техніку, яка використовує глобальну структуру даних УСД. Подібна техніка на базі дерева ClemanX пропонувалася у праці Нувена та ін [64]. Тут кожний вихідний файл представляється як АСД. Для забезпечення співставлення подібності кожне піддерево такого АСД, додатково представлялося характеристичним вектором.

Згодом було запропоновано додаткове виявлення блоків з повторами і надлишковістю коду у праці Хіго та ін. [65] на основі ГЗП. В базі даних проводиться збереження ГЗП кожного методу кожного оновленого файлу, який побудовано на етапі аналізу. На наступному кроці виявлення, після ручного введення користувачем запускається вибірка відповідних графів ЗП з бази даних, які є основою для виявлення клонів. Інший алгоритм, що базується на основі індексу, запропонований в роботі Хаммела та ін. [14]. Основною структурою даних, яка використовується виступає індекс клону. Він представляє собою глобальну структуру даних, яка нагадує типовий інвертований індекс. Цей метод, хоча і використовує лексичний аналізатор для перетворення вихідного коду в токени, але серед всієї літератури з цього питання є найближчим до незалежного

від мови інкрементного підходу. Останньою на сьогодні є техніка, запропонована Рагкетветсагул та ін. [66] і яка спирається на токени. Метод додатково створює чотири різних представлення коду і це дає можливість виявлення чотирьох різних типів клонів.

1.7 Локально-чутливе хешування (ЛЧХ)

Алгоритм пошуку найближчих сусідів на сьогодні є найбільш популярним ймовірністним методом, призначеним для пониження розмірності багатовимірних даних. Тобто пошук, наприклад, подібних документів виявляється досить простим. Базуючись на метриці подібності проводиться порівняння кожного документу з будь-яким іншим документом. При тому, що такий грубий підхід добре працює при невеликих об’єднаннях документів, але виникають проблеми з поганим масштабуванням через вимоги до часу. По мірі того, як збільшується кількість документів, такі вимогу зростають квадратично [67]. Значне скорочення обчислювального часу, потрібного для процесу пошуку [13] відбувається з використанням наближених схем, що базуються на основі ЛЧХ. Даний розділ присвячено огляду основних концепцій ЛЧХ на високому рівні, а також проводиться розбір різноманітних його застосувань в області виявленні повторів на надлишковості в програмному коді.

Основна ідея локально-чутливого хешування полягає в використанні різних хеш-функцій для проведення хешування базових точок даних декілька раз. При цьому буде гарантуватися, що подібні елементи мають більший шанс зустрітися та опинитися в одному хеш-сегменті, на відміну від різнорідних елементів [68]. Тільки тоді перевірку на подібність проходять елементи, які і потрапили в хеш-сегмент, вони також відомі як пари кандидатів.

Внутрішня будова ЛЧХ

Існує велика кількість різноманітних схем локально-чутливого хешування, представлених для використання. Для вибору схеми порівняння документів на подібність, зазвичай користуються базовою мірою подібності. Серед таких показників можуть бути:

– відстань Хемінга [69];
– косінусна відстань або косинус подібності [70];
– метрика Жаккара або коефіцієнт Жаккара [71].

Так наприклад SimHashing [67] метод швидкої оцінки того, наскільки подібні два документа (набори) використовується при виборі схеми ЛЧХ у випадку косинуса подібності, а MinHashing [72] стає базою для досягнення того ж порівняння у випадку коефіцієнта Жаккара. В цьому дослідженні увага буде зосереджена на використанні метрики Жаккара та MinHashing, оскільки в ряді експериментів [73] було встановлено, що такі підходи переважають над SimHashing. Окрім того, з використанням SimHashing не можливо виявити дуже віддалену подібність (низькі відсотки). Всі базові компоненти процесу представлені на рис. 6.

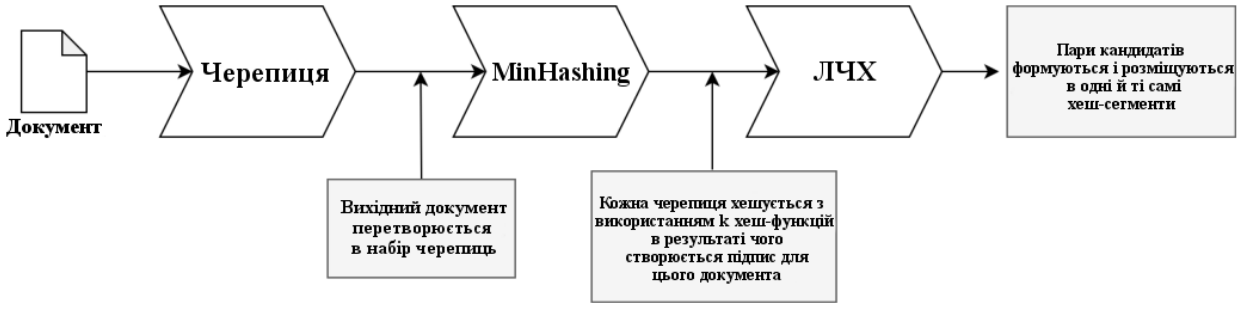


Рисунок 6 – Підопераційї локально-чутливого хешування

Конкретна схема ЛЧХ представлена на рис 6. Складається з трьох основних підопераційї:

– черепиця (Shingling). Відбувається перетворення документа в набір k-черепиць. Він може бути будь-яким: від простих підрядків довжини k до комбінації з k слів. На ймовірність виявлення збігів буде впливати вибір того, що приймається за черепицю. Якщо черепицею виступає 1 символ, то виявлення збігів в іншому документі буде значно вище, ніж знайти збіг з черепицею більшої довжини;

– MinHashing. Після того, як на попередньому кроці були отримані набори, вони можуть виявитися досить об'ємними, тому порівняння стане неефективним. Для часткового вирішення проблеми завдяки MinHashing відбувається перетворення черепиць в менші представлення фіксованої довжини – сигнатуру. Потім коефіцієнт Жаккара використовується не до набору черепиці, а до елементів згенерованого підпису. Збереження інформації набору в максимально можливому ступені, забезпечується згенерованими таким чином підписами. Однак при застосування подібного кроку розрахунок точної подібності між двома файлами стає неможливим, оскільки відбувається втрата інформації. Подібність в цьому разі буде розраховуватися базуючись на оцінці, яка може призвести до точних результатів. З урахуванням всіх обставин, для генерування підпису, який буде являти собою документ, проводиться хешування кожного набору за допомогою k хеш-функцій. Для кожної з цих хеш-функцій обирається мінімальне значення хеш-функції. Для прикладу, отримується підпис із півсотнею значень MinHash на основі 50 випадкових хеш-функцій. Звідси виходить, що більша ймовірність зіткнень і більший рівень помилок буде наслідком використання меншої кількості застосованих хеш-функцій;

– ЛЧХ. За допомогою MinHashing прибирається «бич розмірності», який супроводжує набори черепиці. Весь процес стає неефективний через необхідність порівняння кожного підпису з іншим будь-яким підписом. Ось тут знаходить своє застосування в останньому кроці конвеєра ЛЧХ. Метод, який використовується на цьому кроці носить назву бендінгу або окільцьовування (зв'язування). В матриці розміщується k хеш-значень для кожного документу, потім відбувається розбиття матриці на b смуг, які складаються з r рядків. Тобто при використання дванадцяти

хеш-функцій розбиття проводиться на чотири смуги по три рядки. Потім при появі нового документу $D_{нов}$ потребується визначити, чи відбувається утворення пари кандидатів з іншим документом. Відбувається перегляд кожної смуги та виявлення документів, маючих однакові значення MinHash в кожному рядку цієї смуги. При знаходженні смуги, в якій всі рядки збігаються, буде продовжене повне порівняння між цими документами. Вирішальним являється вибір кількості смуг та рядків, оскільки він становить поріг подібності, над яким вважається, що два документи однакові.

1.8 Висновки

Сфери застосування локально-чутливого хешування дуже різноманітні. В контексті ідентифікації сутності воно було використане Верріос та ін. [74] та Ебрахем та ін. [75]. При біометричному індексування відбитків пальців методи запропоновані Капеллі та ін. [76] та Джоу та ін. [77] спираються також на застосування ЛЧХ. В контексті спільної фільтрації та алгоритмів рекомендацій нерідко відбувається використовується ЛЧХ. Наприклад, для персоналізації широко відомих новин, пошукова система Google використовує ЛЧХ [78], а проблема масштабованості існуючих алгоритмів спільної фільтрації буде вирішена з застосуванням локально-чутливого хешування у Джанг та ін. [79]. В дослідженнях минулих років ЛЧХ застосовувалося, також в контексті виявлення клонів. Використання подібного підходу для створення детектору на основі дерева було запропоновано у Deckard [54]. ЛЧХ тут використовується для порівняння подібності характерних векторів, які були вилучені піддеревами згенерованого АСД. Деревоподібний підхід також лежить в основі схожої методології, представленої, як ClemanX [64]. Нарешті детектор для виявлення клонів 3 типу було запропоновано Хаммелом та ін. [14], де в якості майбутнього розширення виступає ЛЧХ.

2 ДОДАТКОВИЙ ДЕТЕКТОР ПОВТОРЕНЬ ТА НАДЛИШКОВОСТІ ПРОГРАМНОГО КОДУ

Проводячи огляд досліджень, викладених в роботі Хаммела та ін. [14] було вирішено запропонувати підхід удосконалення методів виявлення повторів та надлишковості програмного коду на основі мовно-незалежного інкрементного детектора повторів (МНІДП). Більшість додаткових досліджень, за основу мають деревоподібні та графічні методи, тобто вони суворо залежать від мови програмування. Рішенням в поході МНІДП – є взяття за основу тексту, тобто метод стає базою для досліджень незалежних від мови. Ця методика не являється сугубо мовно-незалежною, але через те, що буде включений етап токенізації, за допомогою незначних корегувань досягнуто потрібного результату. В розділі зазначені зміни, які забезпечили мовну незалежність і того, як вони відобразилися в запропонованому детекторі МНІДП. При цьому надається деталізація аналізу внутрішнього складу (а саме, елементів) детектора та пояснень роботи на різних етапах процесу виявлення[2].

2.1 Огляд мовно-незалежного інкрементного детектора повторів

Проводиться поділ інктементного методу даного дослідження на два основних робочих процеси. В свою чергу кожний буде включати в себе додатково ряд підпроцесів. Проміжне представлення пов’язується з першим процесом і зберігається для повторного використання в версіях проекту. Таке об’єднання (сектор, pool) буде прийматися за індекс повторення (Clone Index – індекс клонування). Весь проект програмного забезпечення використовується цим процесом в якості вхідних даних. Запуск його відбувається одноразово на початку всього конвеєра виявлення повторів (клонованих блоків коду). Другий процес, який посилається на логіку, проводить запуск фактичної процедури виявлення

2 Вихідний код доступний на Github: https://github.com/agamvrinos/LIICD

повторів та надлишковості коду та виводить виявлені клони. Другий процес буде запущено після оновлення основної бази з кодами. В реальному налаштуванні це відбувається після того, як новий запис (commit) буде розміщено в репозиторій керування версіями.

2.2 Процес створення індексу дублювання

У робочому процесі, коли за його допомогою детектор повторень створює індекс повторення (Індекс клону), вирізняють два кроки. Процес представлено на рис. 7.

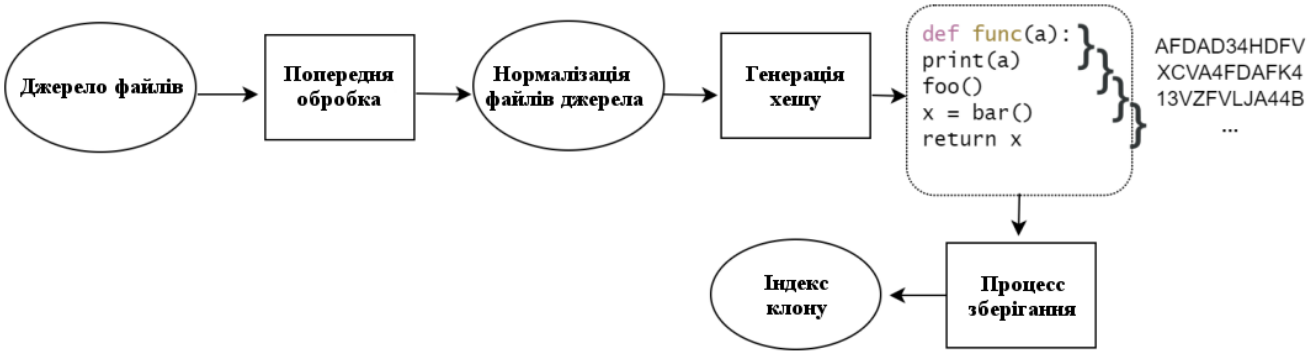


Рисунок 7 – Підкроки робочого процесу створення «Індексу клону»

Спершу на етапі попередньої обробки подається кожний з файлів програмного проекту, який піддався аналізуванню. Відбувається зміна коду (наприклад, прибираються зайві пусті рядки) та визначення ступеню деталізації порівняння. На наступному етапі проводиться групування операторів модифікованого вихідного коду в послідовності, основою яких виступає попередньо визначений параметр конфігурації. Він визначає розмір групування та потім відбувається хешування груп. В «Індексі клону» зберігаються отримані хеші разом з додатковими метаданими (ім’я файлу та індекс оператора в файлі).

Під час даного процесу не виконується жодної логіки виявлення повторів та надлишковості для визначення початкового стану бази кодів, що стосується

дублювання. Тобто, всіх існуючих блоків повторень у кодовій базі системи поки що немає. Однак, це те, чого можна досягти проводячи запит «Індексу клону» з кожним файлом початкової бази кодів. Цей крок не буде застосований, оскільки не вимагається досліджувати еволюцію клонів, а зацікавленість була тільки в дублікатах коду, які були знищені або створені в кожній новій версії.

2.3 Послідовність кроків робочого процесу

Ініціювання робочого процесу відбувається кожного разу, при відправлені наступного нового збереження до репозиторія основаного на контролі версій, де розміщена відповідна програмна система. В контексті даного дослідження, так як запропонований МНІДП не інтегрований з відповідною платформою керування версіями (наприклад, в якості плагіну), буде вручну змодельована процедура. Запуск такої процедури в іншому випадку проходив в реальних умовах автоматично. Виконання проводиться за допомогою файлу конфігурації JSON[3], який вказує оновлені файли, разом з типом відповідного оновлення для кожного збереження (commit), який буде аналізуватися. У типовому коміті міститься інформація, на основі якої відбувається генерація подібного файлу конфігурації. На рис. 8 представлено лістинг файлу конфігурації, призначеного для аналізу двох комітів.

В кожному коміті містяться файли, які були модіфіковані (M), додані (A), знищені (D) або перейменовані (R).

В більшості підетапи інкрементного робочого процесу нагадують такий же процес створення індексу. Фактично ідентичними являються етапи попередньої обробки і генерації хешу. В даному разі зв’язані операції будуть застосовані не до всієї кодової бази, а тільки до визначених файлів.

3 Файл config. json - це файл конфігурації плагіна, який містить основні дані, необхідні для реєстрації плагіна в редакторах.

```
{
  "commits": [
  {
    "id": "cb8f645e0f",
    "changes": [
      { "type": "M", "filename": "lib/plugins/loader.py" }
    ]
  },
  {
    "id": "564907d8ac",
    "changes": [
      { "type": "A", "filename": "fragments/test_refactor.yml" },
      { "type": "D", "filename": "fragments/arch_linux.json" },
      { "type": "R", "filename": [
        "test/facts/system/distribution/__init__.py",
        "test/facts/system/__init__.py" ]
      }
    ]
  }]
}
```

Рисунок 8 – Приклад файлу конфігурації JSON, який показує змінені файли за кожним комітом.

На наступному етапі проводиться порівняння згенерованих хешів з хешами, які зберігаються в постійному «Індексі клону» під час процесу, який на рис. 9 вказаний як «Виявлення клону». Дублювання буде виявлене, якщо два хеши збігаються.

Рисунок 9 – Підетапи інкрементного робочого процесу

Через те, що відбувалася генерація хеш-значень шляхом хешування фіксованого числа операторів коду, наприклад N, то всі виявлені повтори (клони) будуть мати довжину N. Звідси випливає, що потрібне застосування додаткової

логіки для виявлення і розширення блоків з повторами та надлишковістю до їх максимальної довжини.

Як видно з рис. 9 на останніх кроках необхідне оновлення «Індексу клону». Це дасть змогу провести на наступній ітерації порівняння хешів, які були створені файлами в майбутньому коміті з тими хешами, що збереглися в оновленому «Індексі клону». Ця операція буде виконуватися одночасно з виявленням в програмному коді блоків з повторами і надлишковістю.

2.4 Проведення попередньої обробки вихідного коду

У підході Хаммела та ін. [14] пропонується застосувати нормалізацію, яка на етапі попередньої обробки вихідного коду буде містити токенизацію. Виконується такий крок для виявлення клонів 2 типу, шляхом перетворення коду у послідовності лексем. На рис. 10 представлено ілюстрацію, яка введена в початкове дослідження.

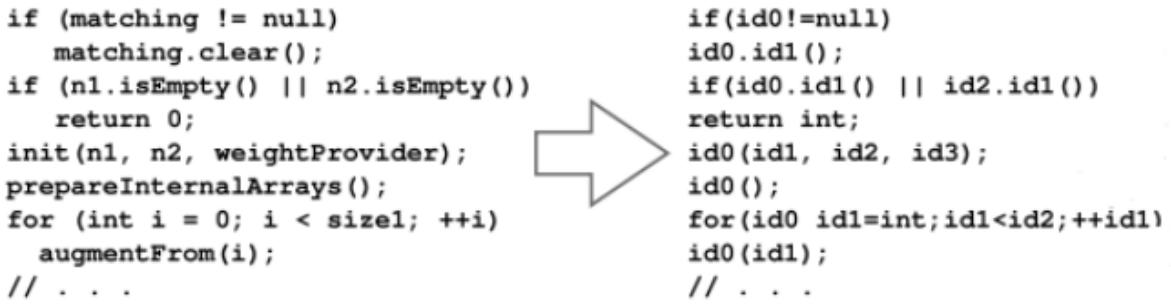


Рисунок 10 – Етап нормалізації, застосований в дослідженні Хаммела та ін. [14]

Однак для перетворення в токени елементів вихідного коду, таких як ідентифікатори, літерали, тощо, необхідний синтаксичний аналізатор (парсер). При цьому мова всього процесу тоді стає специфічною, через те, що для різних мов програмування потрібні різні синтаксичні аналізатори. Задача пошуку або створення синтаксичного аналізатора ускладнюється і для менш популярних мов

програмування. Однак підтримка набору парсерів та використання відповідного, заснованого на основній мові (чи мовах) проекту, не є метою нашого дослідження.

Спираючись на ці факти, в дослідженні не проводиться токенизація вихідного коду, а відбувається застосування тільки основних кроків попередньої обробки. Ці кроки не будуть впливати на функцію запропонованого МНІДП. Тобто можливість виявлення клонів 2 типу буде виключена. В дослідженні вся основна увага буде приділена точним клонам, тому подібна модифікація все ще задовольняє задачі, які поставлені в роботі. Тому, підетапами попередньої обробки, які будуть застосовані для аналізу вихідного коду являються:

– видалення початкових та кінцевих пробілів;
– видалення подвійних пробілів;
– видалення порожніх рядків.

Відмітимо, що видалятися коментарі не будуть, оскільки стилів коментарів у різних мовах програмування багато і їх видалення буде вимагати додаткової роботи. Синтаксичний аналізатор для видалення коментарів не обов’язковий, але все одно необхідно буде виконати сегментацію системи по компонентах різних мов для автоматизації процесу ідентифікації та видалення коментарів.

В роботі Паскарелла та ін. [80] проводиться вивчення коментування вихідного коду програм з відкритим кодом та промислових програм на базі мови Java. Спираючись на проведені дослідження виявлено, що в таких програмах існує низький відсоток співвідношення коду до коментаря. Так для систем з відкритим кодом показник знаходиться в межах 6,3-12,1%, в промислових програмах ці дані менші і становлять 0,1-2,5%. Великих розходжень у висновках з приводу видалення коментарів не очікується. Тому треба притримуватися простих кроків попередньої обробки, які згадувалися вище. Приклад їх використання в реальному вихідному коді представлено на рис. 11.

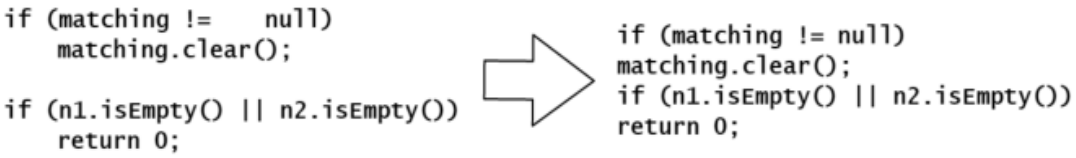


Рисунок 11 – Основні етапи попередньої розробки, застосовані в нашому дослідженні

2.5 Представлення вихідного коду

Для представленого детектора виявлення повторів та надлишковості програмного коду, основою виступає текст. Мається на увазі, що відсутні будь-які спеціалізовані перетворення, які застосовувалися до необробленого вихідного коду. Однак до складу проміжної інформації, яка зберігається і буде використана повторно підчас кожної редакції коду, прості попередньо оброблені оператори не входять. Фактичною інформацією, яка збережена в «Індексі клоні», являються хеш-значення разом з метаданими. Отримуються подібні хеш-значення шляхом хешування блоків, складених з фіксованої кількості операторів.

Основою виступає ковзаюче вікно, де відбувається хешування один за одним послідовних блоків, маючих розмір CHUNK_SIZE, починаючи з діапазону [0, CHUNK_SIZE - 1] до [LINES_COUNT-CHUNK_SIZE, LINES_COUNT - 1]. На рис. 12, представлено приклад даного процесу із заздалегідь визначеним CHUNK_SIZE встановленим в 2. Мінімальна довжина клону неминуче визначається вибором значення для CHUNK_SIZE.

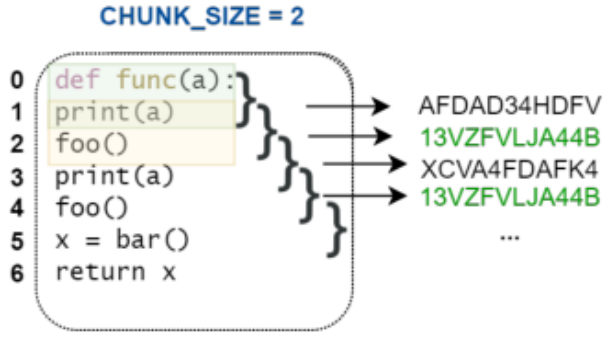


Рисунок 12 – Ковзаюче вікно хешування на основі CHUNK_SIZE

Потім в «Індексі клону» зберігається хеш-значення для кожного хешованого блоку. Там же міститься додаткова метаінформація, яка буде використана під час фази фактичного виявлення блоків з повторами і надлишковістю. До її складу входить:

- ім’я файлу, тобто файлу, де розміщено хешований блок;
- оператор індексу;
- рядок початку блоку;
- рядок кінця блоку.

Зазначимо, що «Індекс клону» можна зберігати, як в пам’яті, так і в реальній базі даних. В нашому дослідженні збереження інформації проводиться в пам’яті, хоча в реальних умовах потрібна буде фактична база даних.

2.6 Виявлення повторів та надлишковості у програмному коді

При виникненні кожної нової зміни коду або коміті, буде ініціюватися процес виявлення повторів та надлишковості. По-перше детектор має провести обробку кожного файлу, який включено до даного коміту, і на основі попередньо визначеного CHUNK_SIZE продовжує хешування послідовних блоків коду. По-друге відбувається порівняння згенерованих хешів з тими хешами, які збережені в «Індексі клону». На виявлення повторень (клонованих частин коду) буде вказувати будь-яка пара співпадаючих значень хешу, який складається рівно з кількості рядків CHUNK_SIZE. Але цього буває недостатньо, оскільки іноді блок з повторюваним кодом може складатися з більшої кількості рядків, чим визначено CHUNK_SIZE. Відтак, детектору треба скористатися додатковою логікою, при запуску якої відбудеться дослідження, чи є можливість додатково розширити мінімальний ідентифікований клон. Саме в дослідженні Хаммела та ін. [14] аналітично представлені і пояснюються деталі цього процесу. Однак, на високому рівні буде працювати визначення того, чи будуть перекриватися окремі екземпляри клонів. Наприклад, якщо блок з повтореннями коду складається з п’яти рядків, починаючи з індексу 0 до 4, інший блок в цьому ж файлі

знаходиться між рядками з індексами 1 і 5, то автоматично це буде означати, що подібні фрагменти перекриваються і окремі клони можуть об’єднуватися в один, який буде займати рядки від 0 до 5. Важливо, що даний процес об’єднання може відбуватися тільки через природу подібного детектора, спроможного спеціально фокусуватися на однакових блоках коду. Невірним виявляється припущення у разі виявлення нечітких клонів на базі оцінок.

Під час процесу виявлення повторів (клонів) та надлишковості, у фрагментах коду відбувається оновлення «Індексу клону» та підготовка його до наступного разу, коли буде запущений інкрементний покроковий процес, при настінні нового коміту. Зокрема, в залежності від типу змін, проводиться обробка пакетами змін файлів, які включені в коміт. При цьому відбуваються наступні дії:

– проходить обробка пакету, відповідного видаленню. Створюється запит «Індексу клону» з записами індексів файлів, які було видалено. Це дозволить визначити, які з клонів були видалені, та прибрати виявлені з індексу. Подібна обробка видалень дуже важлива в першу чергу через те, що в іншому разі (наприклад, спочатку проходила обробка пакету оновлених фалів) відбулося б порівняння відповідних записів індексу з застарілим «Індексом клону»;

– іде обробка оновлених файлів, які відповідають перейменуванню. Це аналогічно обробці видалення. Проходить виявлення повторів в коді і знищуються записи, що відповідають старим назвам файлів. Потім знову їх додають, доповнюючи новими записами з відповідними їм оновленими іменами файлів;

– наступний крок – це оновлені файли, тобто, це розглядається як знищення з подальшим створенням. Як у випадку видалення, спочатку іде запит індексу з неоновленими версіями файлів, для виявлення клонів, які були знищені. Потім проводиться видалення застарілих записів з індексу. Згодом генерується індекс запису для оновлених версій файлів та знову створюється запит індексу, для знаходження повторів коду (клонів), які були додані. В фіналі відбувається оновлення індексу новими записами;

– у простому випадку новостворених файлів іде генерація відповідних записів індексу, виявляються повтори в коді і в кінці проводиться оновлення індексу з додаванням записів до «Індекс клону».

2.7 Висновки

Вихідними даними для детектора являються необроблені текстові логи (текстові файли, в яких зберігається інформація про відвідування, параметрах відвідувань якогось сайту і помилки, які виникали на цьому), які містять вказівки на виявлені повтори (клони) та надлишковості коду. Більш докладно, у логах включено: (1) ім'я файлу для кожного екземпляра клону, (2) початковий індекс кожного клону у попередньо обробленому файлі, (3) початок та кінець рядків, (4) кількість блоків коду довжиною CHUNK_SIZE, які сприяли остаточному максимальному дублюванню. На рис. 13 представлено цей лістинг, рис. 14-15 демонструє лістінги відповідних файлів кінцевого результату.

```
1 (.../project/Test.java|0|0-6) -
2 (.../project/Test2.java|1|1-7) - 2
```

Рисунок 13 – Приклад лістінгу вихідних даних детектору

```
0 import java.lang.*;
1 import java.io.*;
2 class Test {
3     public static void main(
      String []args) {
4         System.out.println("Hello
       world");
5     }
6 }
```

Рисунок 14 – Лістинг Test.java

```
0 import java.util.*;
1 import java.lang.*;
2 import java.io.*;
3 class Test {
4     public static void main(
      String []args) {
5         System.out.println("Hello
       world");
6     }
7 }
```

Рисунок 15 – Лістинг Test2.java

Детектором в даному прикладі було виявлено повтори коду між файлами Test.java та Test2.java. В 1-му екземпляр клонованого блоку знаходиться між

рядками 0-6, тоді як у 2-му можна побачити його у рядках 1-7. Важливо звернути увагу на те, що спостерігається відповідність індексів рядкам, після проведення попередньої обробки файлу. Це значить, що наприклад, якщо в файлі Test.java порожній рядок містився під індексом 0 – блок з повторами був переміщений у рядки від 1 до 7 початкового файлу – екземпляр коду з повторами (клон) все одно буде виявлено в рядках від 0 до 6, оскільки під час фази попередньої обробки відбулося видалення порожнього рядка. Детектором буде виведена кількість блоків коду, які посприяли отриманню повтору. В даному прикладі вибраний CHUNK_SIZE був призначений рівним 6. В результаті цифра 2 підтверджує те, що відбулося використання двох блоків коду довжиною 6, які перекривають один одного. Це призвело до виявлення повторів та надлишковості (клону) довжиною 7.

Всі вище викладені корегування оригінального методу Хамелла [14], зможуть забезпечити мовну незалежність при використанні етапу токенизації, чим і буде досягнення умов виконання поставлених в роботі завдань розробки МНІДП.

3 ДОДАТКОВЕ ВИЯВЛЕННЯ ПОВТОРІВ ТА НАДЛИШКОВОСТІ ПРОГРАМНОГО КОДУ З ВИКОРИСТАННЯМ ЛЧХ

В розділі проводиться обговорення способів розширення алгоритму додаткового виявлення повторів і надлишковості в коді, який був запропонований Хаммелом та ін. [14] і як можна провести вдосконалення для досягнення незалежності від мови програмування. Приділяється увага емпіричним результатам, представленим у оригінальному дослідження, оскільки їх ефективність викликає сумніви. Даний розділ представляє розширення інктементаного детектора повторів, представленого в главі 2. Для цього буде використовуватися метод, представлений в розділі 1.7, відомий як «Локально-чутливе хешування» (ЛЧХ) і проводитися дослідження його придатності в контексті детектору (МНІДП). Розглядається мотивація щодо розширення детектору, а потім заглиблення у внутрішній склад, та роз'яснення відмінності від початкового підходу та які з частин не будуть підлягати змінам[4].

3.1 Мотивація

Спираючись на результати експериментів, які проводилися авторами роботи Хаммел та ін. [14] виникла мотивація розробки розширеного підходу. Під час одного з експериментів відбувається аналіз Eclipse SDK (v3.3) (платформи для програмування та компіляції додатків на Java). Сам проект програмного забезпечення нараховує понад 42 млн. рядків коду та більш ніж 200 000 файлів вихідного коду, написаних на мові програмування Java. В дослідженні вимірюється час потрібний для створення початкового індексу клонування і час, який треба затратити для його запиту і оновлення. Результати експерименту представлено в таблиці 1.

4 Вихідний код доступний на Github: https://github.com/agamvrinos/LIICD

Таблиця 1 – Аналітичні вимірювання Eclipse SDK (v3.3) [77]

Створення індексу (повне)	7 год. 4 хв.
Індексний запит (на файл)	0.21 сек. медіана 0.91 сек. середнє
Оновлення індексу (на файл)	0.85 сек. середнє

На основі цих вимірювань легко побачити, що час, потрібний на створення індексу повторів (клонів) досить великий (складає 7 годин 4 хвилини). Не зважаючи на те, що експеримент проводився на досить застарілому обладнанні, як на сьогоднішній час, часові затрати виявилися досить суттєвими. Мета нашого дослідження вияснити, чи можна підвищити продуктивність цього кроку використовуючи ЛЧХ, і в кінцевому результаті скоротити час необхідний для створення проміжної інформації.

3.2 Огляд

Ідеєю даного підходу виступає те, що згідно з оригінальним дослідженням [14] операція обчислення всього індексу клонування (індексу блоку з повторами та надлишковістю) з нуля виявляється дуже затратною за часом. Тому пропонується для отримання ефективної оцінки подібності файлів програмного проекту скористатися ЛЧХ. Згодом, для тих файлів, в яких знайдені повторення, виконується обчислення елементів індексу на льоту та, як описано в главі 2, виконується операція виявлення. При використанні такого підходу вимагається досягти компромісу. Хоча здається, слідуючи інтуїції, може відбутися покращення продуктивності детектору під час створення індексу, але з’являється негативний вплив на наступні аспекти:

– відгук (Recall). Якщо файли виявилися схожими, то тільки для них при подібному підході відбувається процес виявлення повторів в програмному коді. Звідси випливає, що будуть існувати пропущені блоки з повторами коду. Наприклад, якщо файли дуже великі, то здебільшого вони відрізняються, але в них присутні деякі ідентичні фрагменти коду. Для зменшення вірогідності

пропуску повторів та надлишковості в коді (хоча повністю така можливість не буде виключена) необхідно вибрати низькій поріг подібності файлів при їх порівнянні;

– продуктивність запитів (Query Performance). Час необхідний для запиту та виявлення блоків коду з повторами при кожному новому коміті буде збільшуватися, через додаткові накладні витрати на обчислення. Такі витрати виникають при обчисленні записів індексу на льоту для пар подібних файлів. Однак, якщо час створення індексу буде суттєво зменшений, то до такої поведінки можна ставитися терпимо. Відбувається виділення двох основних робочих процесів, які згадувалися в главі 2 подібного детектору клонів. Для повторного використання в версіях, так само у першому процесі створюється початкова проміжна інформації. Однак в цьому разі інформація буде різнитися з індексом клонування детектора МНІДП. З іншого боку відбувається повторний запуск другого процесу для кожної нової версії програмної системи. При використанні ЛІХ файли, які були зачеплені, переміщаються в сегменти разом з існуючими файлами, схожими до кожного з них та вище визначеного раніш порогу подібності.

3.3 Робочий процес створення індексу подібності на базі ЛЧХ

Даний процес складається з ряду підкроків, так само як і процес створення «Індексу клону», приведеному в главі 2. Підкроки робочого процесу в результаті призведуть до створення початкового стану проміжної інформації. Так само, як і в детекторі МНІДП, початкова точка така сама, тобто будуть використані такі самі кроки для попередньої обробки. Однак, частина процесу, яка залишилася, буде мати суттєві відмінності. В цьому випадку хешування блоків коду та зберігання їх в так званому «Індексі клону», з метою групування подібних файлів в одному місці і буде використовуватися ЛЧХ. В розділі 1.7.1 описувалися підкроки ЛЧХ, які будуть використані. Це черепиця (shingling) та генерація підписів (minhash) для заданих файлів, а також групування схожих. Нижче надано детальний опис

кожного з них. Високорівневе представлення робочого процесу показано на рис. 16.

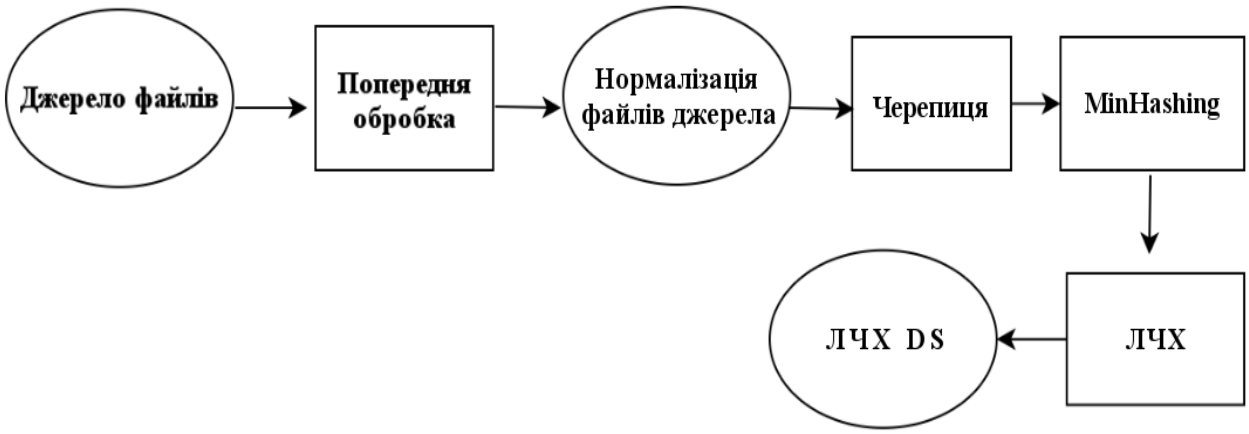


Рисунок 16 – Підкроки робочого процесу створення індексу на основі ЛЧХН

Так само, як в випадку з детектором МНІДП набір постійної проміжної інформації буде названо індексом, який на рис. 10 представлено як ЛЧХ DS[5]

3.4 Інкрементний робочий процес на основі ЛЧХ

Ініціація кожного разу інкрементного кроку, подібна до відповідного процесу підходу МНІДП, відбувається при оновленні кодової бази базового програмного проекту до нової версії. Незмінними залишаються всі звичайні операції стосовно попередньої обробки для даного процесу. Відмінність від підходу МНІДП становить інша частина конвеєра. Більш конкретно, то в випадку детектора, який базуватиметься на основі ЛЧХ, спочатку проводиться ідентифікація на подібність файлів в існуючій кодовій базі, з файлом на який впливатиме коміт. Для цього буде використано ЛЧХ, та відбудеться запит індексу на основі ЛЧХ. Після цього кожен файл потрапляє в хеш-сегмент з подібними файлами. Так само, використовується той самий підхід, як і в детекторі МНІДП, для схожих файлів. Генерується хеш-значення та виконується той самий процес для виявлення повторів та надлишковості в програмному коді. На рис. 17

[5] DS (Data Structures and Algorithms) – структури даних та алгоритми

представлено огляд високого рівня, який відображується підкроками даного процесу. Коміти в цьому випадку мають вигляд файлу конфігурації JSON.

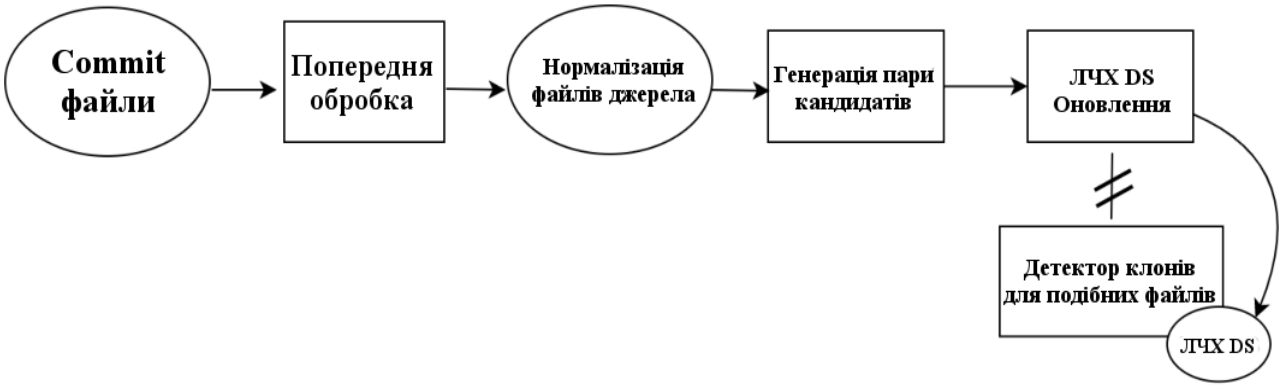


Рисунок 17 – Підкроки робочого процесу, основаного на ЛЧХ

3.5 Декомпозиція підходу

Даний розділ містить рішення стосовно кожного окремого підкроку реалізації та розбиття ЛЧХ в контексті заданого розширення. Для цього розширення пропонується використовувати набір даних (datasketch)[6] – який представлений сторонньою бібліотекою, в який присутня вбудована реалізація MinHash ЛЧХ.

3.5.1 Черепиця

Даний підхід, висвітлено в розділі 1.7.1. Черепиця або shingling, являється процесом, в ході якого іде перетворення вихідного документа в набір черепиць. В даному випадку документами виступають вихідні файли проекту, який підлягає аналізу. Після кроку, в ході якого відбувається попередня обробка, отримується файл, під дією конвертації перетворений в набір черепиць. Тобто, кожний рядок файлу – окрема черепиця. На рис. 18 представлено результат цього процесу, де

[6]Бібліотека доступна на Github: https://github.com/ekzhu/datasketch

показано, як проводиться перетворення попередньо обробленого файлу в набір черепиць, яким відповідають відповідні рядки цього файлу.

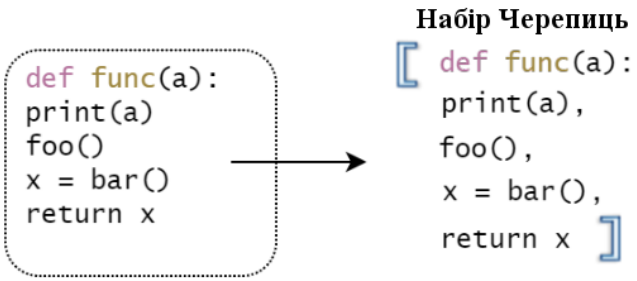


Рисунок 18 – Перетворення попередньо обробленого файлу в набір черепиці

3.5.2 MinHashing

MinHashing – є наступним кроком у процесі загального алгоритму ЛЧХ. Тобто, метою цього кроку являється усунення явища великих наборів черепиць, через які відбувається збільшення часу, необхідного для розрахунків метрики подібності або коефіцієнта Жаккара. При цьому виникає необхідність застосування *k-хэш-функций.* Під їх впливом проходить хешування для кожного окремого файлу, кожної черепиці в наборі черепиць. Далі за алгоритмом іде вибір нижнього значення хеш-функції для кожної з k-хэш-функций та завдяки цьому генерується підпис. На рис. 19 зображено другий підкрок процесу.

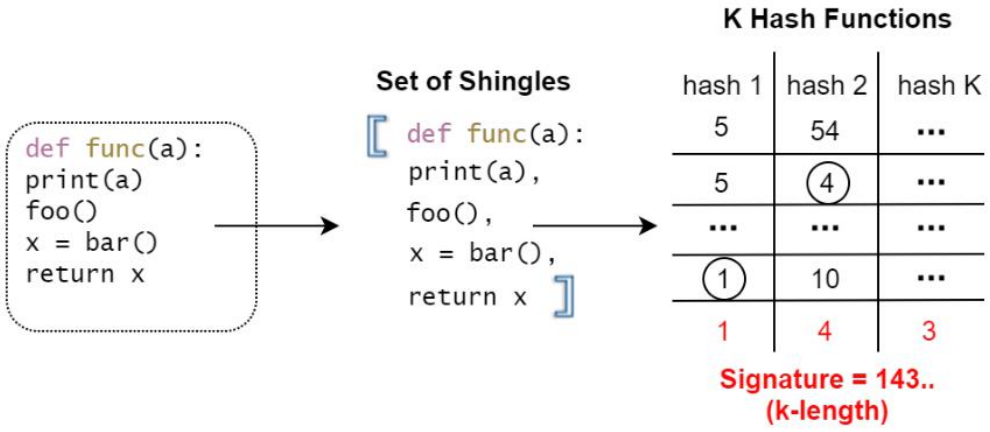


Рисунок 19 – Формування підпису за допомогою MinHashing

Дане перетворення поєднано з втратою інформації, через перетворення наборів на підписи з фіксованою довжиною. Саме тому, не має можливості використати їх для обчислення точної подібності. В роботі [68], однак було продемонстровано, що можна навіть в цьому випадку одержати точні оцінки. Звідси випливає, що точність оцінки залежить від довжина підписів, тобто має залежність від кількості використаних хеш-функцій. Оцінка тим краще, чим функція більше, але при цьому процес її обчислення стає більш трудомістким. Можна фактично задати коефіцієнт похибки оцінки наступною формулою (3):

$$error = \frac{1}{\sqrt{k}}, \tag{3}$$

де k – є кількість хеш-функцій, тобто наприклад буде отримано 6,25% ймовірностей на помилковонегативних або помилковопозитивних спрацювань для 256 хеш-функцій. За допомогою параметра конфігурації datasketch можна підтримати зміну кількості хеш-функцій.

3.6 Висновки

Збільшення часу обчислень, який необхідний для порівняння кожного підпису з будь-яким іншим підписом, являється останнім кроком підходу ЛЧХ. Як вже згадувалося в 1 главі, використовується така техніка, як бендінг (тобто, banding – окільцьовування, зв'язування, об'єднання). Техніка полягає в тому, всі хеш-значення для кожної черепиці з набору, що складають собою матрицю, розбиваються на b смуг, кожна з яких в свою чергу складається з r рядків. Далі слідує розгляд документу D_1 для формування пари кандидатів з іншим документом D_2. При цьому, необхідно, щоб кожний рядок визначеного діапазону D_1 співпадав з кожним рядком відповідного діапазону в D_2. Якщо таке відбулося, то документи будуть вважатися парами кандидатів і потрапляють до одного хеш-сегменту. Однак доведеться обирати самостійно значення для b і r, стосовно

налагоджуваної реалізації ЛЧХ. При цьому потрібно завжди звертати увагу, як ці значення впливають на поріг подібності. Навпаки іде робота datasketch, дозволяючи визначити потрібний поріг, завдяки автоматичному обчисленню правильних значень цих параметрів.

Всі вище розглянуті розширення алгоритму оригінального підходу Хамелла [14], пояснюють причини використання ЛЧХ, в якості розширення інктементаного детектора повторів та сприяють досягненню мовної незалежності розробленого МНІДП. Також проводиться освітлення відмінності від початкового підходу й вказуються частини, які змінювати не буде потреби.

4 ОФОРМЛЕННЯ РОЗРОБКИ МНІДП

В розділі представлено опис кількісних експериментів, які були проведені та пояснення їх відповідності завданням даного дослідження. Для кожного окремого типу експерименту згадувалися різні метрики оцінки, які були використані. Проводилось дослідження їх придатності цьому експерименту. Надано опис процесу збору інформаційного фонду та обговорення імітації серії вхідних комітів для забезпечення цілей аналізу дослідження.

1.1 Види експериментів

В планах дослідження постає необхідність проведення експериментів, які зможуть відповісти на завдання дослідження представлені у вступі. Отже було розроблено для початку дві реалізації детектора виявлення повторів і надлишковості в програмному коді з потрібними функціями. Далі вимагається проведення кількісних експериментів для конкретних випадків.

1.1.1 Оцінка МНІДП

На питання, як діє підхід МНІДП, для початку треба визначити, принцип роботи запропонованого детектора, який було отримано в наслідок корегування підходу представленого Хаммел та ін. [14]. Щоб получити представлення про продуктивність МНІДП, його запускають для п’яти програмних систем, проводячи вимірювання вимог до часу та пам’яті. Тобто, для кожної системи відбувається аналіз серії з п’ятдесяти комітів та виміри часу і пам’яті, які будуть потрібні для процесу створення індексу. Також проходить вимірювання середнього часу, потрібного для обробки п’ятдесяти комітів під час кроків інкрементного процесу. Для спостереження за поведінкою МНІДП по відношенню до виявлених повторів та надлишковості у програмному коді, постає

потреба у проведенні додаткових експериментів. Зокрема, відбувається аналіз десяти самих останніх комітів для кожної з систем, існуючих в наборі даних і виявлення тих повторів (клонів), які додавалися та знищувалися.

1.1.2 Оцінювання МНІДП проти SIG

В ході експериментів відбувається порівняння ефективності запропонованого у дослідженні підходу МНІДП та сучасного традиційного підходу SIG, для виявлення повторів та надлишковості в програмному коді. Остаточна мета – це перевірка покращення, яке має забезпечувати інкрементний підхід, в тому разі, коли процес виявлення повторюється регулярно для кожного нового перегляду програмного проекту.

Порівнювання виявляється непростою справою, оскільки детектор клонів SIG відрізняється від МНІДП в багатьох відношеннях. Конкретніше, процес виявлення повторів в програмному коді SIG відпрацьовує в межах SAT (інструмент, для запуску окрім виявлення клонів, ще великої кількості додаткових процесів). Тому з'являється потреба в ізолюванні процесу виявлення клонів та виміру його окремо від інших процесів. Крім цього, відбувається виведення всіх клонів в версії програмного забезпечення через те, що детектор SIG не є інкрементним. Напроти, МНІДП дає змогу виведення тільки нещодавно доданих або видалених блоків повторів (клонів) в визначеній версії системи. З цього випливає, що неможливо співставити напряму результати двох підходів, саме з точки зору виявлення повторів.

В даному дослідженні проводиться заміри і порівняння двох вищезгаданих підходів на основі часу, потрібного для процесу виявлення повторів та надлишковості в програмному коді. Зокрема, відбувається застосування детектору SIG для серії з півсотні комітів, заміри середнього часу, потрібного для обробки та порівняння результату з результатами індивідуальної оцінки МНІДП. Щодо виявлених кожним інструментом повторів (клонів коду) та надлишковості, та врахування описаної відмінності в вихідних даних кожного з підходів, на

практиці відбувається перевірка на співпадіння клонів, виявлених цими детекторами. Однак, через те, що запропонований в дослідженні метод, та детектор SIG не являються ймовірнісними, заснованими на тексті методами (тобто необроблений вихідний код, не підлягає ніяким перетворенням, які можуть призвести до втрати інформації, що тягне за собою зниження точності та повноти), то можна очікувати співпадіння виявлених блоків повторень, виявлених обома підходами. Щодо пам’яті, то знову підходи виявляються неспівставними, через складність ізоляції процесу виявлення блоків повторів SIG та необхідності проведення вимірювання пам’яті тільки для цієї частини процесу.

1.1.3 Оцінка розширень, заснованих на ЛЧХ

Для відповіді на питання розширення та вдосконалення початкового підходу, шляхом використання ЛЧХ, виникає необхідність в вимірюванні продуктивності запропонованого розширення на базі ЛЧХ та порівнянні його з продуктивністю МНІДП. Так само, як і при оцінюванні мовно-незалежного інкрементного детектору повторів, досягається це виміром часу виконання та потреб в пам’яті запропонованого в дослідженні розширення на основі ЛЧХ. Порівнянні цих показників з відповідними результатами для підходу з використанням детектору МНІДП. Є можливість проведення індивідуального порівняння, оскільки обидва підходи являються інкрементними, та в їх основі полягає одна і та сама основна ідея. Відбувається порівняння метрик для кожного з двох процесів, процесу створення індексу та процесу інкрементного кроку.

Знову буде застосоване вище згадане експериментальне налаштування, де використані п’ять програмних систем і півсотні комітів для кожної з них. Важливо звернути увагу на те, що дослідження направлене лише на вивчення запропонованого розширення на базі ЛЧХ з точки зору ефективності. Тобто, запропонований метод може призвести до підвищення продуктивності в будь-якому з двох робочих процесів: створення індексу та інктементного кроку.

Логічно було б провести вимір відгуку реалізації заснованої на ЛЧХ, оскільки цей метод за визначенням включає ймовірність пропуску блоків коду з повторами (клонів) та надлишковістю, але такі дослідження проводитися не будуть. Після того, як буде визначено спочатку, чи достатньо ефективною виявилася реалізація на основі ЛЧХ для покращення підходу МНІДП, тільки потім має сенс отримати висновки, які містили б корисну інформацію, про втрату клонів. Однак результатом нашого дослідження стає вивчення ефективності підходу, а відповідні емпіричні експерименти по втраті блоків коду з повторами та надлишковістю залишаються в якості майбутньої роботи.

1.2 Вхідна перевірка

Для перевірки результатів, які будуть отримані після застосування розроблених детекторів, виникає потреба в еталонному тесті для обраних програмних систем. Тобто, потрібно отримати оцінку того, чи являються отримані виявлені блоки з повторами та надлишковістю дійсними та повними, а саме, чи були виявлені всі доступні клони. Під час розробки запропонованого дослідження подібний набір перевірочних даних не був доступним. В зв’язку з цим перевірка результатів запропонованого підходу на основі МНІДП та ЛЧХ в контексті даного невеличкого проекту відбувалася вручну. Було протестовано і перевірено декілька варіантів використання, які стосувалися знищення, оновлення, створення чи перейменування файлів. Вручну створено коміти для імітації оновлення реальної програмної системи. Звичайно, перевірка на повноту для детектора на основі ЛЧХ не проводилась, через характер даного методу. Зверніть увагу, хоча результат тестувався для невеликого проекту, очікується, що запропонований підхід буде відпрацьовувати так само і для великих кодових баз, якщо охоплюватимуться всі сценарії варіантів використання. Тести, які були проведені, відносяться до наступних варіантів використання:

– додавання новостворених файлів. Виконувалося тестування сценаріїв, в яких ці файли представляли один або декілька блоків з повторами (клонами) та

надлишковістю, а інші сценарії, в яких дублювання не додавалося. У першому типі сценаріїв проводилася перевірка на правильність виявлення блоків з повторами та надлишковістю коду, тоді як в другому перевірялося, що клони не виводилися;

– перейменування файлів. Відбувалася перевірка, що блоки коду з повторами в перейменованих файлах були виявлені та зареєстровані з оновленим ім’ям файлу;

– оновлення файлів. Проводилося тестування декількох сценаріїв, в яких знищувалися та додавалися блоки коду в існуючі файли, перевіряючи виявлення знищених або доданих фрагментів програмного коду з повторами та надлишковістю;

– знищення файлів. Тестування стосувалося видалених файлів, в яких містилися блоки коду з повторами (клонами) та надлишковістю і підтвердження в цьому разі, що відбулося вірне виявлення видалених клонів.

1.3 Конфігурація інструментів.

Параметри конфігурації, які були використані в експериментах для реалізації детекторів клонів, заснованих на МНІДП та ЛЧХ, представлені в таблиці 2.

Таблиця 2 – Конфігурація реалізацій МНІДП та ЛЧХ

МНІДП	МНІДП з розширенням ЛЧХ		
CHUNK_SIZE	CHUNK_SIZE	PERMUTATIONS	THRESHOLD
6	6	64	0.2

Параметр CHUNK_SIZE – це кількість рядків у кожному блоці коду, який підлягає хешуванню. Значення його було обрано рівним 6, що означає мінімальний розбір блоку програмного коду з повторами та надлишковістю, та вказує на ідентичну мінімальну довжину клону, яку SIG визначає за замовчуванням на даний час.

Параметр PERMUTATIONS – це кількість функцій, в запропонованій реалізації на основі ЛЧХ, які використовуються для процесу MinHashing. З точки зору порівняння подібності, значення цього параметра рівне 64, призводить до рівня помилок, розрахованих за формулою (4)

$$\text{error} = 1/\sqrt{64} = 12{,}5\% \quad (4)$$

Тобто, коли відбувається ідентифікація двох фалів як подібних/не подібних, ймовірність того, що ця ідентифікація помилкова складає 12,5%.

Параметр THRESHOLD – поріг, який прирівнюється до 0,2, та перетворюється в 20%, вказуючи на самий низький поріг подібності, для якого визначається подібність двох файлів.

1.4 Інфраструктура ЛЧХ та інформаційний фонд

Для досягнення мети експериментів, при їх проведенні запускається дві категорії тестів на одній машині з налаштуванням апаратного забезпечення представленого у таблиці 3.

Таблиця 3 – Технічні характеристики забезпечення

	Специфікація
Пам’ять	32 GB
Процесор	Intel Xeon E5-2650 v2 @2.6GHz

Інформаційний фонд

В ході проведення досліджень для різних реалізацій детекторів, було використано в якості вихідних даних п’ять проектів з відкритим вихідним кодом. Вибір відбувався таким чином, щоб була змога охопити відносно широкий спектр проектів, які мають різний розмір та написаних на різних мовах програмування.

Конкретніше, відбувалося вимірювання розміру кожного проекту з точки зору LOCs (Lines Of Code), з застосуванням інструменту CLOC[7]. Даний інструмент спроможний проводити підрахунок пустих рядків, рядків коментарів та фізичних рядків вихідного коду на багатьох мовах програмування. Вимірювання точної частки дублювання у вибраних системах неможливе, через характер запропонованих в дослідженні детекторів, спроможних виводити блоки коду з повторами (клони), які були додані або видалені в певній версії, замість того, щоб виводити всі клони і цій версії. Однак, спираючись на статистичні дані, які отримані з аналізу, проведеного в рамках SIG 192 програмних систем, виявилося, що середнє дублювання коду складає приблизно 13%, при стандартному ± 12%. В даному розділі роботи представлено процес вимірювання LOCs для кожної системи, а також процес фільтрації вихідних частин.

1.4.1 Початкова колекція інформаційного фонду

Завдяки вимірюванню великої кількості проектів з відкритим кодом, для створення колекції початкового інформаційного фонду, було обрано п’ять з них. При цьому їх LOCs коливаються в діапазоні приблизно від 300 000 до 23 000 000 рядків коду. На рис. 20 представлено лістінг, де вказані параметри конфігурації CLOC, які було використано на цьому етапі.

```
$ cloc --skip-uniqueness {target_project}
```

Рисунок 20 – Початкова конфігурація CLOC

Обрані проекти зведені в таблицю 4, де також вказана мова програмування, загальна кількість файлів, та кількість рядків в програмному коді LOCs.

Причиною обрання програмних систем з LOCs нижче описаного діапазону являється те, що справжні переваги інктементного підходу стають видимими

[7] Інструмент доступний на Github: https://github.com/AlDanial/cloc

лише для проектів з відносно великою кількістю рядків в програмному коді, коли повторне виявлення з використанням традиційних підходів стає доволі повільним.

Таблиця 4 – Початковий інформаційного фонду проектів з відкритим кодом

	Мова програмування	Кількість файлів	Кількість LOCs
Linux Kernel	C	57.205	23.229.768
OpenJDK-14	Java	60.444	12.045.316
Tensorflow	C++, Python	12.387	3.194.893
Kooboo	C#, JS, HTML, CSS	4.109	670.265
Ripple	C / C++	1.399	312.011

Обране об’єднання програмних проектів, як видно з таблиці, охоплює широкий спектр мов програмування та кількості (LOCs) рядків. Надані значення у стовпці «Кількість стовпців LOCs» – є результатом додавання, що відповідає коментарям, і тих, які посилаються на фактичне джерело коду. До уваги не приймаються пусті рядки, оскільки їх буде знищено на етапі попередньої обробки кожного детектору.

1.4.2 Фільтрація виконуваного коду

Для вимірювання різних показників ремонтопридатності, включаючи виявлення повторів та надлишковості у програмному коді SIG використовує такий інструмент, як SAT. За допомогою нього також є можливість виконання відповідного аналізу тільки тих частин кодової бази, що позначені як виконуваний код.

Ігнорування відбувається і коду, поміченого, як тестовий код, разом із іншими незначними файлами, такими як README.md або файли журналів. Після обробки SAT тільки частини всієї кодової бази, в результаті, що логічно, іде зменшення кількості LOCs, які піддаються обробці. Для об’єктивного порівняння детектора SAT та запропонованої в дослідженні реалізації, треба враховувати і

таке знищення. Тому знову доведеться використовувати CLOC для оцінки LOCs для кожного проекту, включаючи цього разу неревалентні каталоги. Точна конфігурація представлена в лістінгу на рис. 21. Нові вимірювання для вищезгаданих проектів подані в таблиці 5.

```
$ cloc --skip-uniqueness --exclude-dir=test,tests,doc,examples,licences
  ,lib --not-match-f='^.*test.*$ {target_project}
```

Рисунок 21 – Конфігурація фільтрації CLOC

Таблиця 5 – Відфільтрований інформаційний фонд проектів з відкритим кодом

	Мова програмування	Кількість файлів	Кількість LOCs
Linux Kernel	C	56.270	22.963.637
OpenJDK-14	Java	23.905	7.498.482
Tensorflow	C++, Python	8.610	2.120.650
Kooboo	C#, JS, HTML, CSS	4.064	668.055
Ripple	C / C++	1.073	207.166

1.4.3 Фільтрація невірних файлів

Програмні проекти, в тому числі ті, які обрані в нашому інформаційному фонді, зазвичай складаються з ряду файлів у двійковому форматі. Обробка подібних файлів для виявлення, в потрібному контексті, блоків з повторами та надлишковістю позбавлена сенсу, оскільки не можна отримати цінної інформації. Отже при проведенні обробки кожного відповідного проекту, додатково будуть виключені двійкові байти та файли з неюнікодовими символами в обох реалізаціях МНІДП та ЛЧХ. В додатку Б представлено список з вказуванням усіх розширень файлів, які піддавалися фільтрації. Увага звертається на те, що, оскільки подібні файли утворює лише невелика частина всієї кодової бази, то

видалення таких файлів, як і очікується, лише незначно вплине на кількість рядків (LOC) у коді, які згадуються у попередніх розділах.

1.5 Моделювання комітів

Постає необхідність моделювання процесу відправки комітів в репозиторій управляння версіями, де буде розміщено проект, який підлягає аналізу. Це забезпечуватиме спостереження та вимірювання того, як працюють детектори під час покрокового робочого процесу. Таке моделювання, як було представлено в главі 2, відбувалося з використанням файлів конфігурації JSON. Подібні файли складаються за списків комітів, які піддаються аналізу, а також змін (створення, оновлення, знищення, перейменування), які кожний з цих комітів вносить.

В контексті експериментів, при проведенні досліджень в ході роботи, для кожного з програмних проектів відбувається аналіз п’ятдесяти останніх комітів, виключаючи злиття, та вимірюються показники, які відповідають кожній з описаних категорій експериментів. Самий останній коміт для кожного програмного проекту, показує снепшот (знімок файлової системи – копія файлів та каталогів файлової системи на даний момент часу), використаний під час аналізу та представлений в Додатку А. Оскільки детектори за визначенням виключать деякі з цих півсотні комітів, які можуть складатися із змінених файлів, то існує ймовірність обробки менше ніж п’ятдесяти. Наприклад, коміт буде пропускатися, якщо він включатиме лише модифікації тестових файлів (які пропускаються детекторами), то зрештою ніякі файли не оброблятимуться.

1.6 Результати експерименту

Проводиться демонстрація результатів експериментів, опис яких було представлено в попередніх пунктах розділу 4. Більш конкретно, для початку будуть розглянуті результати експериментів з запропонованим детектором МНІДП, які висвітлені в розділі 2. Далі будуть представлені результати

експериментальних спроб вимірювання ефективності традиційного підходу SIG, щодо виявлення блоків з повторами та надлишковістю в програмному коді. Порівняння підходу із запропонованим додатковим детектором повторів та надлишковості. І в кінці надаються результати проведення експериментів, завдяки яким вивчено ефективність розширення, основою якого виступає ЛЧХ. Відбувається вивчення отриманих результатів порівняно з результатами експериментів з МНІДП.

1.6.1 Вимірювання показників МНІДП

Таблиця 6 містить точні виміри, які використані для експериментів з оцінювання запропонованого МНІДП.

Таблиця 6 – Виміри МНІДП

Проект	Кількість рядків (LOCs) прочитаних в комітах	Оброблено	Час створення індексу (сек)	Середній час інктемент-ного кроку (сек)	Стандартне відхиленні інкремент-ного кроку (сек)
Rippled	208.100	42	3.75	0.85	1.31
Kooboo	681.143	50	16.28	0.03	0.04
Tensorflow	3,814.652	45	65.89	4.29	3.32
OpenJDK-14	3,377.211	46	48.53	4.72	5.07
Linux Kernel	23.603.823	45	321.21	N/A	N/A

З таблиці випливає, що порівняно з початковими оцінками, отриманими за допомогою CLOC, LOC для кожного проекту, буде коливатися в межах похибки приблизно 2,7%. Виключення становить Tensorflow та OpenJDK, для яких значну частину від загальної кодової бази, вочевидь, складають виключені каталоги та розширення файлів. Більш того, майже всі коміти з п’ятдесяти, які піддавалися аналізу для кожної кодової бази, були оброблені. Тут нагадуванням є те, що всі пропущені коміти відносяться до комітів, які зачіпають тільки файли, проігноровані реалізацією запропонованого в дослідженні підходу в першу чергу

(наприклад, текстові файли). Наступні два стовпця дають представлення про середній час, який пройшов з того моменту, коли був створений індекс, та інкрементні кроки (кроки з прирощуванням) робочих процесів для реалізації МНІДП. Останній стовпець висвітлює стандартне відхилення для вимірювання часу інкрементного кроку.

1.6.2 Створення індексу для МНІДП

Вимірювання часу

Показники отримані при вимірюванні часу для процесу створення індексу запропонованим в дослідженні детектором МНІДП представлені на рис.22.

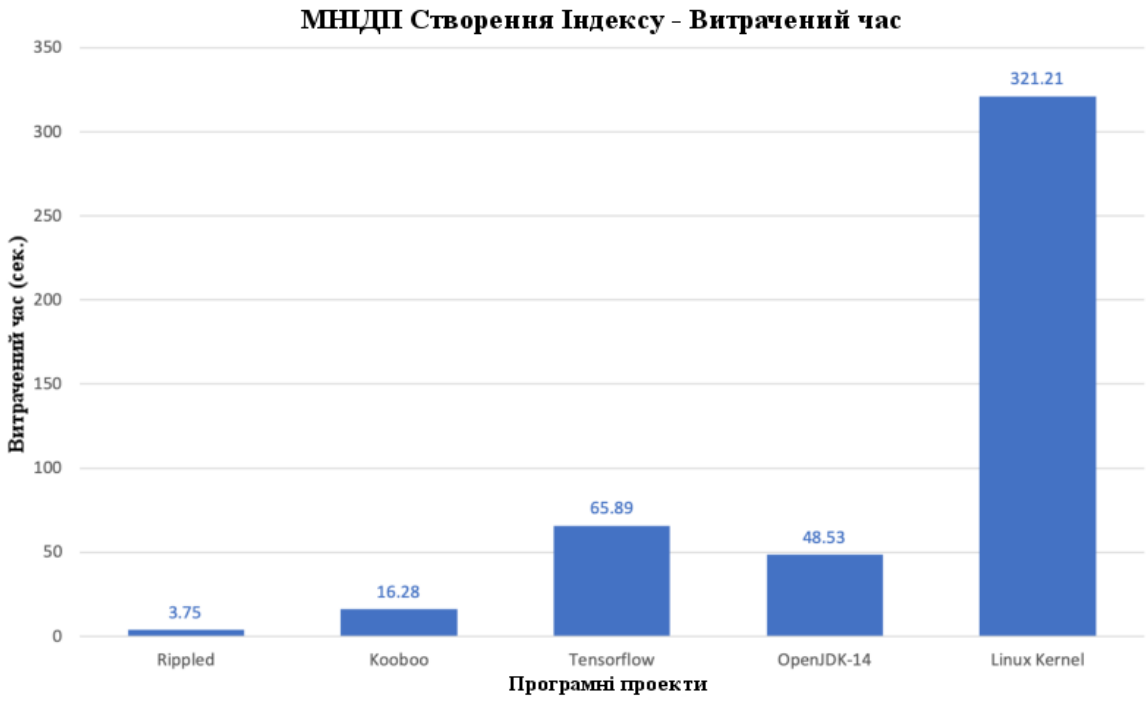


Рисунок 22 – МНІДП – Час виконання процесу створення індексу

Загальний час потрібний для створення індексу, який необхідний для детектора МНІДП, як можна побачити, коливатиметься в межах від декількох секунд для невеликих систем (наприклад, Rippled) до приблизно п’яти хвилин для великих систем (наприклад, ядро Linux), маючи на увазі, що час потрібний на

виконання цього кроку залежить від розміру програмної системи. Це очікувано, оскільки, час для розрахунку вимагатиметься більшим і він залежить від індексів повторів (клонів вихідного коду), які у системах з великою кількістю вихідних файлів та LOC досить великі. Тим не менше, загальний час створення індексу коливатиметься на досить низькому рівні. Враховується те, що подібний час зайняв саме більше, приблизно п’ять хвилин для такої великої системи, як ядро Linux, яка складається більш ніж з 20 мільйонів LOC.

Вимірювання пам’яті

Для аналізу п’яти програмних систем, які використані в даному дослідженні, використовується об’єм пам’яті (див. таблицю 7), необхідний для запропонованого детектора МНІДП.

Таблиця 7 – МНІДП – Показники пам’яті

Проект	Пам’ять (MB)
Rippled	129
Kooboo	348
Tensorflow	1791
OpenJDK-14	1712
Linux Kernel	12500

Представлені цифри все ж таки, в основному, відповідають вимогам пам’яті перших систем, незважаючи на те, що вимірювання проводилися на протязі всього аналізу (сюди ж включено створення індексу та інкрементні кроки) запропонованих систем. Короткочасні сплески використання пам’яті, як буде висвітлено в розділі 5.1.2., здебільше спричинялися виконанням додаткових кроків за декілька секунд. Звідси випливає, що при розгляді загального використання пам’яті, яке необхідне для конкретної реалізації детектора, їх поява немає значення.

На рисунку 23 представлено використану пам’ять, необхідну для підходу МНІДП на протязі кожного аналізу. Як видно, навіть враховуючі сучасні

стандарти пам’яті, показники для чотирьох перших систем досить незначні. Однак, використання пам’яті зростає до більш високих рівнів, коли справа стосується більш крупної системи ядра Linux.

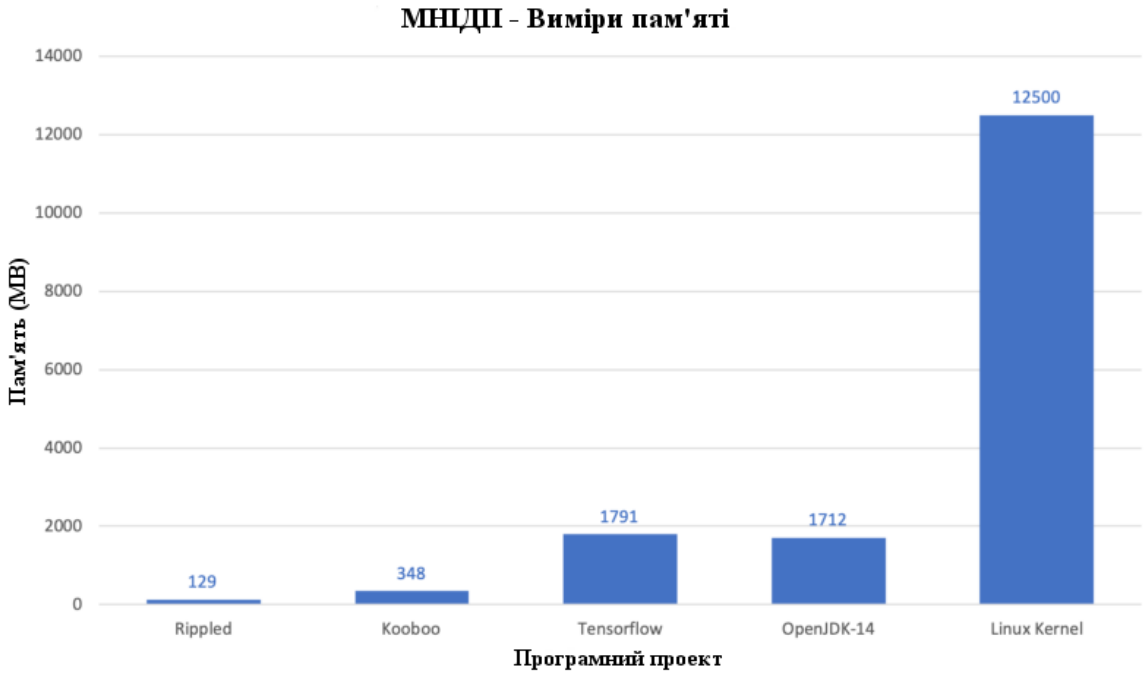


Рисунок 23 – Сукупні вимоги до пам'яті

1.6.3 Інкрементний крок для МНІДП

Зміни витраченого часу інкрементного кроку реалізації МНІДП представлено на рис. 24. Даний процес запускається кожного разу, коли відбуваються зміни коду в формі коміту. Результати, отримані для систем Rippled, Tensorflow і OpenJDK, дають представлення про те, що такий процес не потребує в середньому більше ніж кілька секунд. Стосовно незвичайного відхилення під час експериментів для Kooboo, то такі низькі показники виправдовують себе у випадках, коли процес виявлення блоків з повторами та надлишковістю взагалі не запускається. Відбувається подібне в разі, якщо файли, на які впливає коміт, не виводять та не знищують клони. Подібне призводить до того, що найбільш трудомісткі частини процесу виявлення не виконуватимуться. Нарешті, через величину ядра системи Linux запропонована установка не змогла відпрацювати із

навантаженням пам’яті цього процесу. Зауважимо, що відбулося це, не через вимоги до пам’яті для інкрементного кроку як такого, а через підпроцес перевірки контролю версій, працюючого в розробленому в дослідженні додатку. Оскільки для великих кодових баз такий процес буде вимагати значного об’єму пам’яті, більшого від того, який на той час був доступним в машині експериментального налаштування.

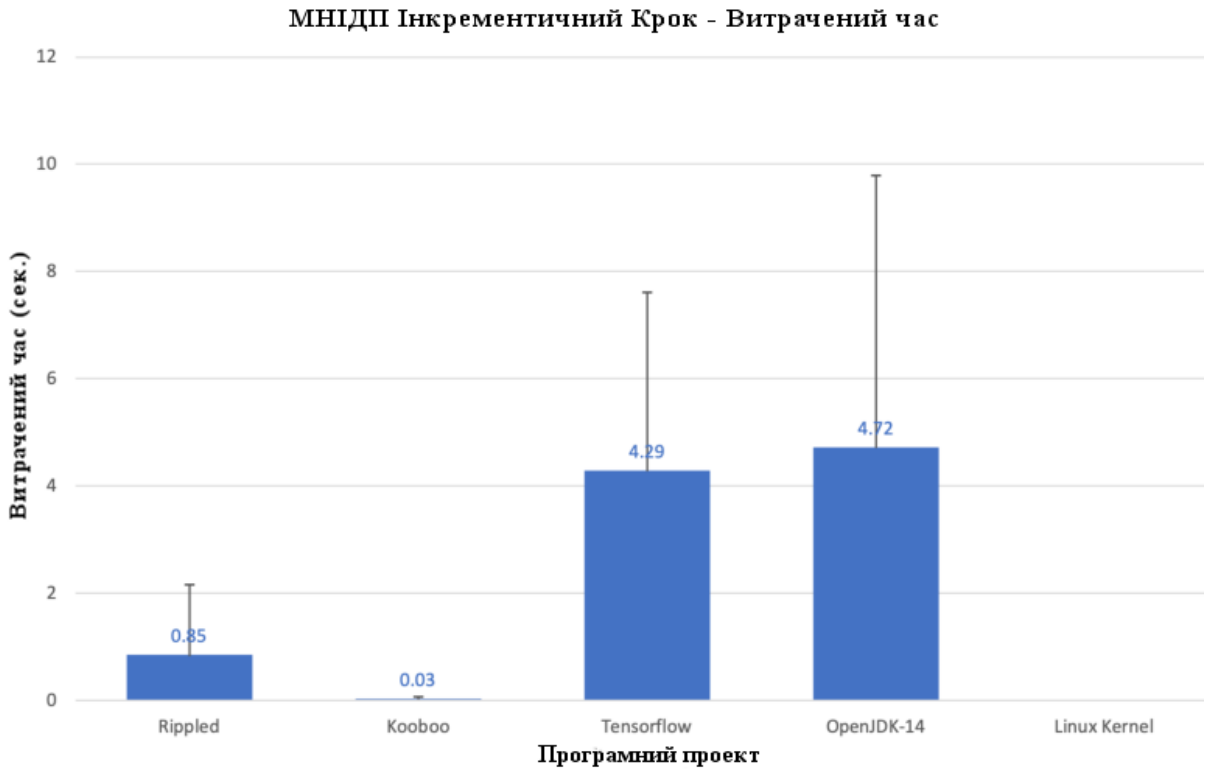


Рисунок 24 – МНІДП – середній час виконання для інкрементного кроку

В більш загальному, велика кількість різноманітних факторів впливає на час виконання, необхідний для інкрементного кроку. Під елементами, які можуть впливати на результати розуміють: кількість файлів, які включені в коміт; тип змін (створення, оновлення, видалення, перейменування); довжина аналізованих файлів; поріг подібності в випадку ЛЧХ. Були проведенні додаткові експерименти, щоб отримати повне представлення про взаємодію всіх цих факторів. Важливо відмітити, що загальна продуктивність цього кроку більш ефективна в порівнянні з процесом створення індексу. Це добре видно на

рисунках 22 і 24 проводячи їх порівняння. Це суттєво, враховуючи, що процес виконується кожного разу, коли в код вносяться нові зміни.

1.6.4 Виявлення повторів та надлишковості в програмному коді

Стосовно виявлення в програмному коді блоків з повторами та надлишковістю (інакше, клонів), проводиться запуск МНІДП для чотирьох програмних систем, для яких наша машина спроможна витримати навантаження інкрементного кроку. Більш конкретно, для спрощення було вирішено піддавати аналізу десять самих останніх комітів для кожної системи. Потім підрахувати кількість доданих та знищених клонів, які будуть виявлені запропонованим інструментом. Загальний огляд кількості файлів, на які впливає кожний коміт, представлені в таблиці 8.

Таблиця 8 – Файли, які зазнали впливу на програмну систему для кожного аналізованого коміту

№п/п коміту	# Оновлені файли в коміті			
	Rippled	Kooboo	Tensorflow	OpenJDK-14
1	1	6	1	3
2	2	3	1	1
3	0 (пропущений)	1	2	4
4	3	1	0 (пропущений)	1
5	1	1	0 (пропущений)	0 (пропущений)
6	2	1	1	1
7	1	2	1	2
8	3	3	1	5
9	2	1	2	0 (пропущений)
10	1	3	1	0 (пропущений)

Хоча типи змін, які вносяться в проаналізовані файли (додавання, оновлення, перейменування, видалення) не були вказані, більшість з них відповідають модифікаціям існуючих файлів. Записи в таблиці, відмічені як «0 (пропущений)», відповідають комітам, які не піддавалися обробці, з причини

включення в них тільки файлів, які за замовчуванням ігноруються МНІДП (наприклад, текстові файли).

На рисунках 25-28 представлені блоки з повторами та надлишковістю, які були додані та видалені для кожного коміту кожної програмної системи. Нагадування, МНІДП проводить розгляд модифікованих файлів, як видалення, за яким іде створення.

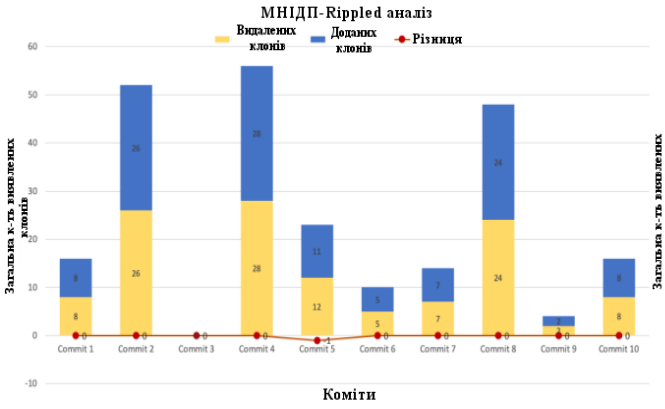


Рисунок 25 – Виявлення клонів для Rippled

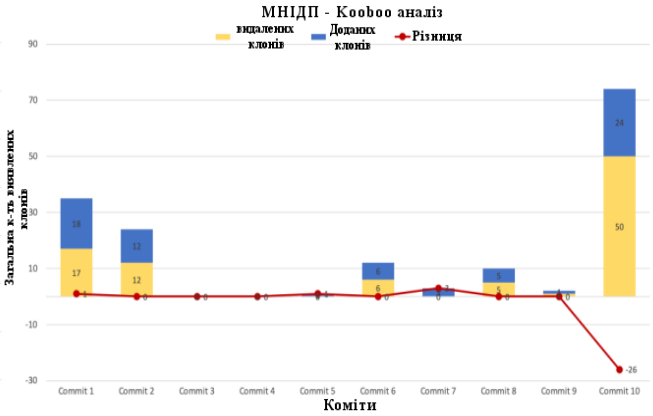


Рисунок 26 – Виявлення клонів для Kooboo

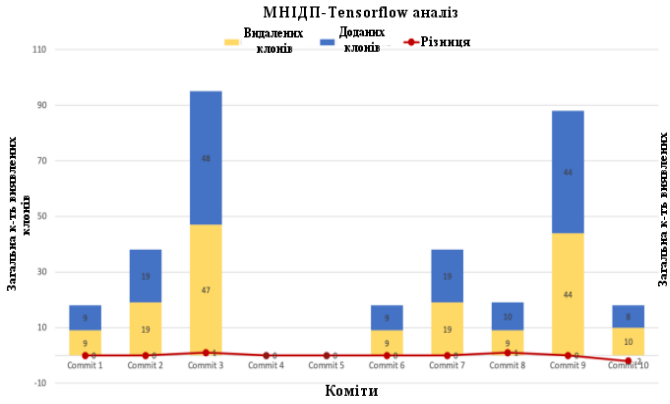


Рисунок 27 – Виявлення клонів для Tensorflow

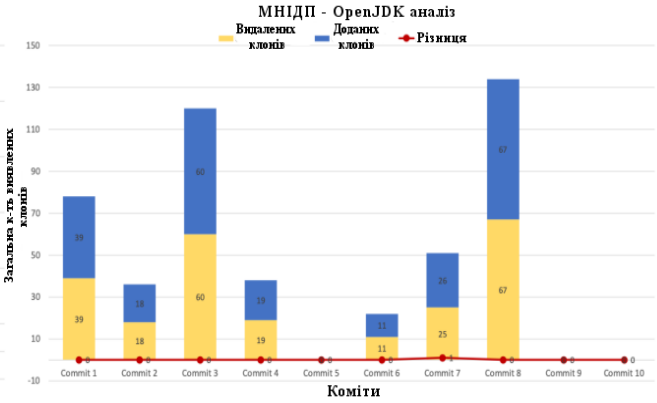


Рисунок 28 – Виявлення клонів для OpenJDK-14

Це призводить до того, що блоки з повторами та надлишковістю, відповідні зміненим файлам, будуть виявлені, як знищені під час кроку видалення та як заново додані під час підкроку додавання. Це зображене на рис. 25-28, де в

багатьох випадках кількість видалених клонів відповідає кількості доданих. Інформація, яка дозволяє зробити висновок, чи були будь-які клони додані або видалені, представлена різницею між двома цифрами.

Іншим спостереженням є те, що доля блоків з повторами в кодовій базі відповідної системи не залежить від більшості комітів. Фактично, лише під впливом одного коміту змінилася загальна кількість клонів в системах Rippled і OpenJDK, тоді як у випадку Tensorflow та Kooboo, два та чотири коміти видалили/додали клони, відповідно. Додатково були досліджені вихідні дані МНІДП для комітів, стосовно кількості виявлених клонів, для яких виявлена кількість блоків з повторами та надлишковістю здається більшою (наприклад, коміт 8 для OpenJDK). У подібних випадках, більшість зареєстрованих клонів, які зазвичай включені до багатьох файлів системної кодової бази, відповідають блокам коду/коментарям. Таким прикладом може бути запис в перших рядках деяких вихідних файлів фіксованої інформації, яка стосується ліцензій.

1.7 Оцінювання МНІДП проти SIG

Результати нашого дослідження, в яких аналізується продуктивність загального аналізу SAT разом з підпроцесом, вбудованого в той же інструмент і призначеного для виявлення блоків з повторами та надлишковістю в програмному коді, представлені в таблиці 9. Конкретніше, було використано інструмент SAT SIG при проведенні аналізу набору даних для запропонованого дослідження, який складається з п’яти проектів з відкритим кодом. Зважаючи на те, що SAT є складним інструментом, до якого входить множина базових операцій, що не мають відношення до виявлення дубльованого коду, було прийнято рішення ізолювати відповідні частини та провести заміри частки загального часу, який пройшов. Ця частка напряму пов’язана з виявленням блоків з повторами та надлишковістю в програмному коді.

Таблиця 9 – Параметри SAT та загальний час виявлення МНІДП

Проект	Загальний час аналізу SAT	Час виявлення клонів	Створення індексу МНІДП та час інкрементного кроку
Rippled	4 хв.	5.63 сек.	4.6 сек.
Kooboo	22 хв.	397 сек.	16.31 сек.
Tensorflow	8 год. 30 хв.	177.1 сек.	91.29 сек.
OpenJDK-14	N/A	N/A	56.34 сек.
Linux Kernel	N/A	N/A	>321.21 сек.

Експериментальне програмне забезпечення, запропоноване в дослідженні з успіхом змогло б завершити аналіз трьох програмних систем, але для систем OpenJDK и Linux, це зробити не вдалося б через їх складність та розмір. Тим не менш, можна спостерігати, що навіть якщо час, потрібний для виявлення блоків коду з повторами та надлишковістю, складає лише невеликий фрагмент від загального часу аналізу, для SAT буде потрібна значно більша кількість часу для проведення аналізу. Особливо видно, що для Tensorflow загальний час аналізу становив більше ніж 8 годин. Це число вказує на обмеження кількості перевірок, які можна було б провести, до п’яти. Стосовно показників часу для виявлення клонів, то вони коливаються від декількох секунд до пари хвилин для великих та більш ускладнених систем. Треба, звернути увагу на те, що час виявлення блоків з повторами та надлишковістю, залежить не тільки від розміру пакету, в контексті SAT. Додатковими факторами, які впливають на ці вимірювання, виступають: складність самого коду, мова програмування, на якій написана система. Відповідно, і на зменшення часу виявлення блоків з клонами коду для Tensorflow у порівнянні з більш меншою системою Kooboo.

При неодноразовому повторі запуску процесу виявлення клонів, відбувається швидке накопичення окремих вимірів, які на перший погляд можуть видатися доволі малими. Вплив виміряного часу, коли процес виявлення повторів та надлишковості в програмному коді повторюється часто, можна добре зрозуміти з представлених діаграм на рис. 29. Зокрема було використано в якості параметра

систему Tensorflow, для демонстрації того, як час, необхідний для виявлення клонів зростатиме по мірі росту кількості комітів.

Як було вказано, хоча різниця для одного коміту і не виглядає великою, але для великої кількості комітів результати виглядають по-різному, наприклад 100 або 500. В останньому випадку, на проведення всього аналізу, наприклад, використання традиційного інструменту SIG вимагатиме приблизно 1387 хвилин – або коло 23 годин. З іншого боку, для проведення тієї ж процедури при використання поетапного підходу необхідно лише близь 37 хвилин.

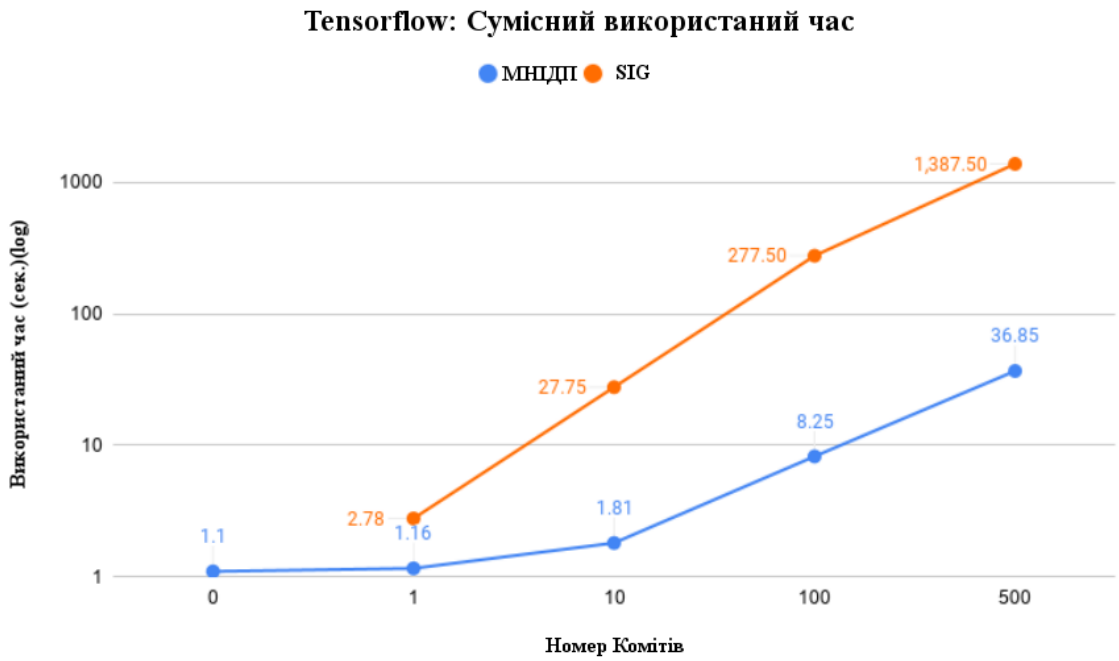


Рисунок 29 – Сукупне порівняння детектора SIG з детектором МНІДП

1.8 Вимірювання розширення на основі ЛЧХ

Для проведення оцінювання запропонованої в дослідженні реалізації на основі ЛЧХ, потрібно провести порівняння її з результатами експериментів з детектором МНІДП. Для виконання індивідуального порівняння між ними проводяться індивідуальні вимірювання для двох робочих процесів для кожного з двох підходів, що представлено в таблиці 10. Треба звернути увагу, що для цього типу досліджень були використані такі самі експериментальні процедури, які

представлені в розділі 5.1. Це означає, що для кожної з систем відбулась обробка однокової кількості LOC, а також однакової кількості комітів.

Таблиця 10 – Вимірювання розширення на основі МНІДП та ЛЧХ

Проект	Час створення індексу (сек)		Середній час інкрементного кроку (сек)		Стандартне відхилення інкрементного кроку (сек)		Пам’ять (MB)	
	МНІДП	ЛЧХ	МНІДП	ЛЧХ	МНІДП	ЛЧХ	МНІДП	ЛЧХ
Rippled	3.75	10.42	0.85	0.82	1.31	1.35	129	60
Kooboo	16.28	37.82	0.03	0.25	0.04	0.04	348	122
Tensorflow	87	192.8	4.29	1.57	3.32	2.13	1791	524
OpenJDK-14	51.62	139.87	4.72	4.26	5.07	6.34	1712	429
Linux Kernel	321.21	954.56	N/A	4.38	N/A	10.23	12500	2600

1.8.1 Створення індексу для розширення ЛЧХ

Вимірювання часу

Велика різниця в часі між двома реалізаціями, отримана в результаті, стосується часу, потрібного для створення індексу. Можна побачити з рис. 30, що для всіх програмних систем в нашому інформаційному фонді, підхід МНІДП в порівнянні з розширення на основі ЛЧХ, практично в три рази швидший.

Нагадування – для процесу MinHashing при реалізації на основі ЛЧХ було використано 64 хеш-функції. Кількість хеш-фунцій була обрана такою низькою, саме з метою створення порогу в відношенні того, скільки часу потрібно для цього розширення для створення індексу, оскільки дане число призводить до коефіцієнту помилок подібності в 12,5%. Додаткові затрати часу, які в подальшому призводять до збільшення часу створення індексу реалізації на основі ЛЧХ, виникають саме через збільшення кількості хеш-функцій, і таким чином, зниження кількості помилок.

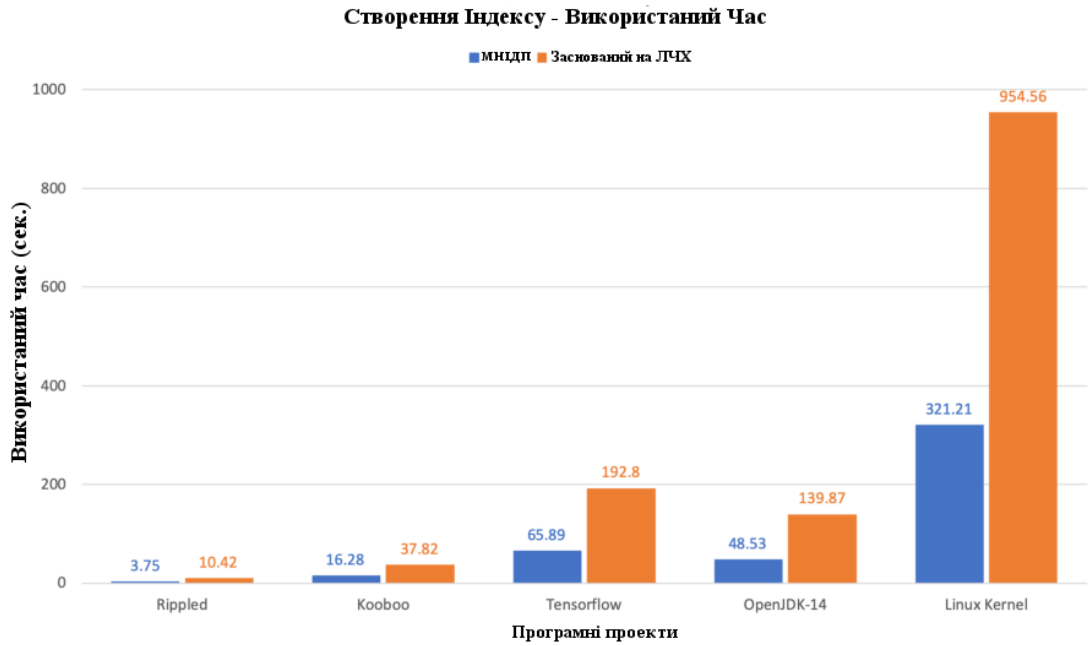


Рисунок 30 – Час виконання процесу створення індексу для двох реалізацій

Вимірювання пам'яті

Потреби в пам’яті розширення на основі ЛЧХ, які побудовані разом з проведеними в дослідженні попередніми вимірами для підходу МНІДП представлені на рис. 31.

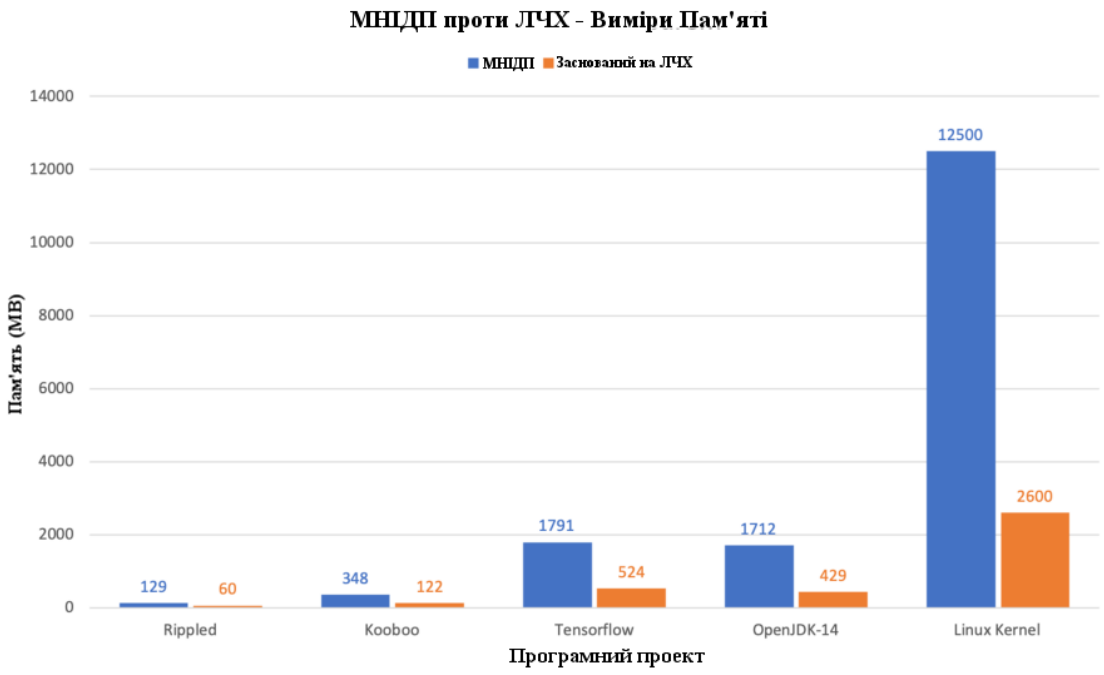


Рисунок 31 – Вимоги до пам'яті для двох реалізацій

В процесі дослідження встановлено, що рівні використання пам’яті підходу МНІДП в два-три рази вище, ніж для підходу, заснованого на ЛЧХ. Особливо, це стосується випадку з ядром Linux, існування суттєвої різниці між двома підходами: для завершення виявлення МНІДП вимагатиметься приблизно в п’ять разів більше пам’яті.

1.8.2 Інкрементний крок для ЛЧХ

Відповідні вимірювання стосовно виконання інкрементного кроку для двох реалізацій представлено на рис. 32. З першого погляду відповідні отримані дані виглядають доволі заплутаними. Все через те, що очікувалася гірша продуктивність для усіх проаналізованих проектів, у порівнянні з підходом МНІДП ніж запропонованого забезпечення на основі ЛЧХ, з урахуванням додаткового етапу обчислення записів індексу повторів (клонів) і надлишковості на льоту.

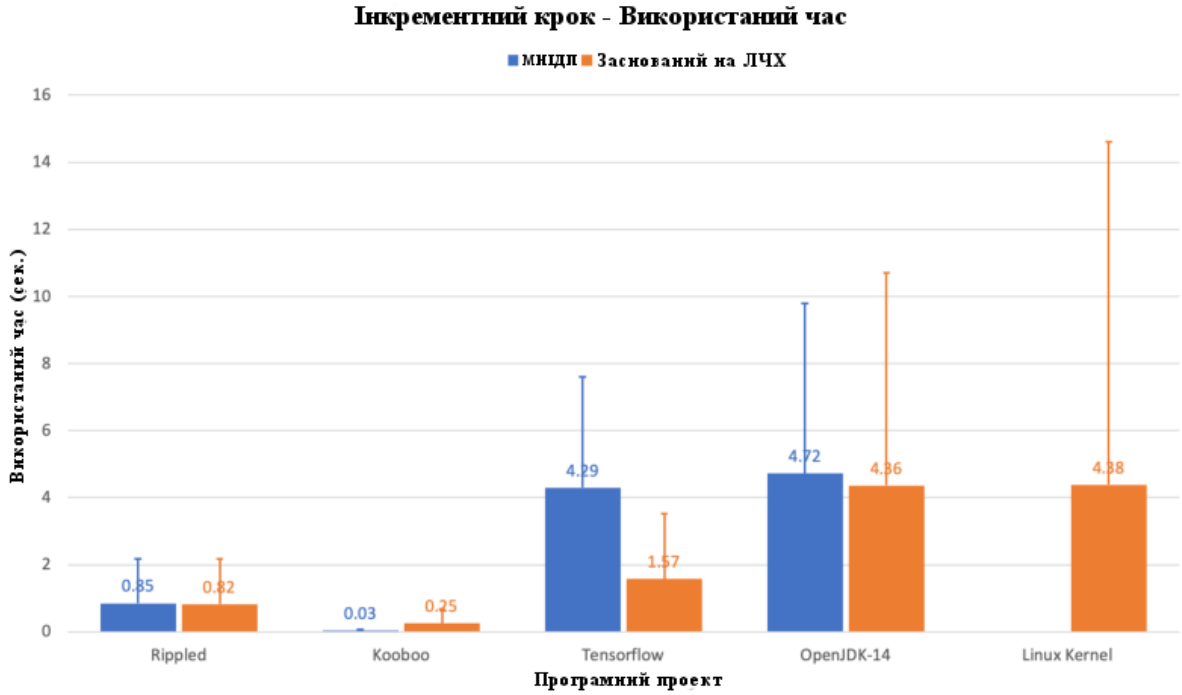


Рисунок 32 – Час виконання процесу інкрементних кроків для двох реалізацій

Однак після проведення додаткового дослідження, стало зрозуміло, що в багатьох випадках обраний поріг подібності в 20% був недостатньо низьким, для ідентифікації великої кількості файлів на подібність і виявилося причиною того, що це не відобразилося в нашому візуалі. І як результат, багато виявлень були пропущені, що і призвело до зменшення часу, показано на діаграмі рис. 32.

При подальшому проведенні досліджень причин, що спричинили таку велику різницю між забезпеченнями, виявилося, що частина MinHash в загальній схемі ЛЧХ є доволі важкою з точки зору обчислень. Фактично, приблизно 37% часу (як видно з рис. 33) при створенні індексу було витрачено на підкрок MinHashing, тобто це випливає з вихідних даних, отриманих ізольованими вимірюваннями часу кроку створення індексу для реалізації ЛЧХ з використанням системи Tensorflow. Решта, що лишилася (а саме 63%) розділилися між черепицею (Shingling) та процесом бендінгу (process of banding) (відображається, як ЛЧХ), причому перша вимагає більш за все часу.

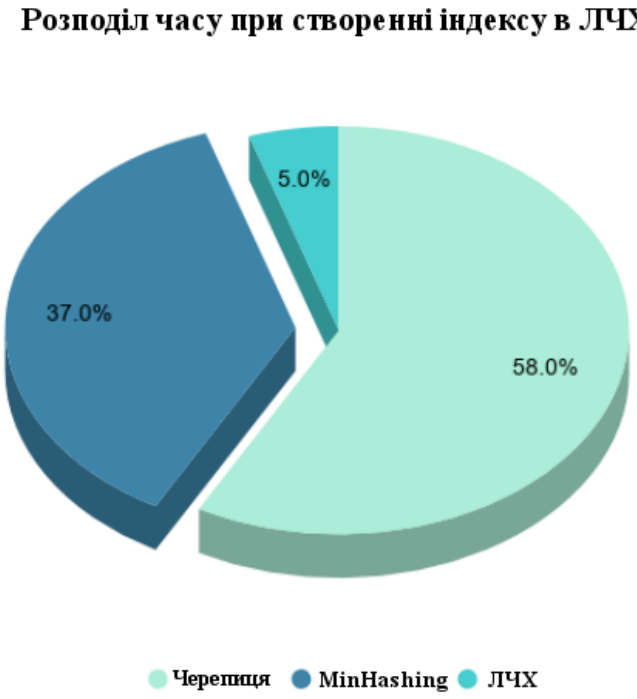


Рисунок 33 – Підвиміри створення індексу для реалізації на основі ЛЧХ

1.8.3 Змінні хеш-функцій

В розділі 5.1.3 згадувалося, що для виконання задач нашого дослідження з реалізацією на основі ЛЧХ використовувалися 64 хеш-фкнції для процесу MinHashing. Також, оскільки MinHashing – одна з самих трудомістких частин реалізації ЛЧХ, було б цікаво дослідити, як впливатиме зміна в кількості хеш-функцій на інкрементний крок та загальний час створення індексу. Наслідки збільшення кількості хеш-функцій, відповідно зменшення кількості помилок, часу необхідного на створення відповідного індексу та часу обробки інкрементного кроку представлені на рис. 34. Конкретніше, було використано програмний проект Kooboo та проведені заміри часу створення індексу для 64, 128, 256 і 512 хеш-функцій, що в свою чергу, відповідатиме частоті помилок 12,5%, 8,8%, 6,25% і 4,4% відповідно.

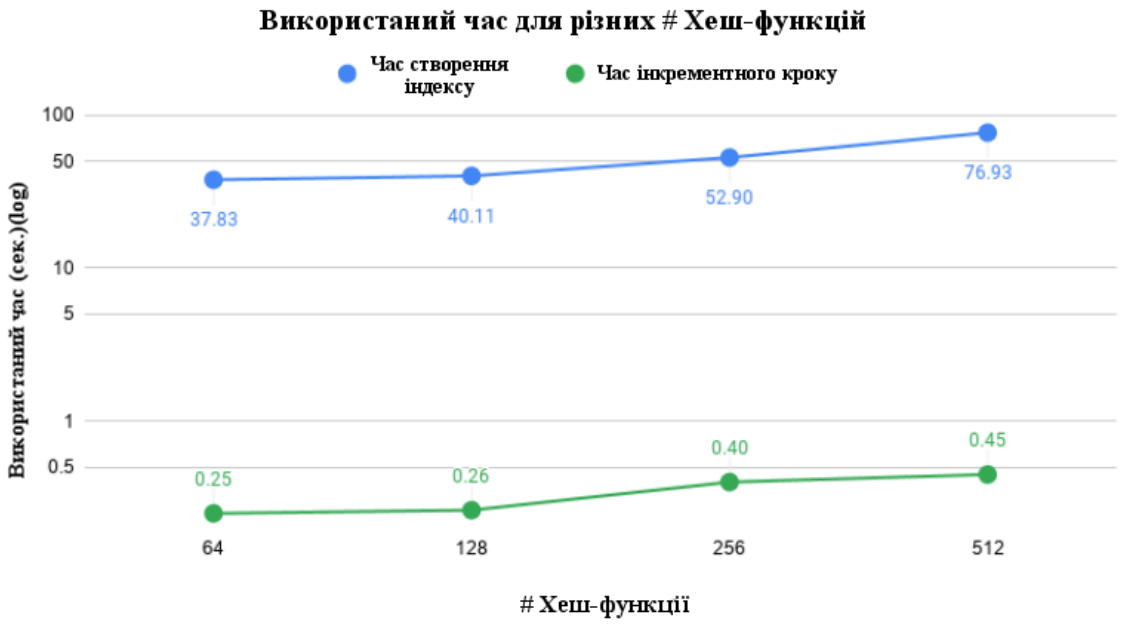


Рисунок 34 – Значення часу виконання для різної кількості хеш-функцій в процесі MinHashing

З діаграми виходить, що важливу роль, при обробці двох робочих процесів, потрібний час, залежить від кількості хеш-функцій. Зокрема, час створення

індексу практично подвоївся, якщо порівнювати дані вимірів самого нижнього порогу в 64 та самого високого з 514 хеш-функцій. Ця поведінка узгоджується з вимірюваннями інкрементного кроку процесу, де спостерігалося збільшення аналогічної величини.

1.9 Результати досліджень

Даний підрозділ дає інтерпретацію результатів проведення дослідження та обговорюються його наслідки. Крім цього вказуються загрози достовірності даного дослідження та надаються поради для використання у практиці запропонованих детекторів у SIG.

Нижче надаються висновки, які можна зробити вивчивши результати проведених досліджень, тобто аналіз результатів проведених експериментів і їх відповідність поставленим в дослідженні задачам.

1.9.1 Підхід МНІДП

Після проведеної роботи можна зазначити, що у порівнянні з вимірами, які повідомляються в роботі Хаммела та ін. [14], детектор МНІДП працює швидше, ніж очікувалося. Враховуючі те, що створення індексу для системи з 20 мільйонів рядків (LOCs) зайняло не більше п’яти хвилин, з урахуванням вимірювань часу створення індексу МНІДП, це вказує на достатньо хороше масштабування підходу.

Згідно з проведеними в дослідженні вимірами інкрементного кроку процесу виявлено, що виконання конкретної операції займає декілька секунд. Саме останнє має вирішальне значення, з урахуванням того, що при новій зміні коду у формі коміту, кожного разу відбувається ініціація процесу. Хоча необхідно проводити додаткові дослідження, для визначення, як саме фактори, які впливають на продуктивність цього процесу, насправді діють на нього. Але після

отримання результатів дослідження можна зробити висновок, що цей крок виконується ефективно.

Стосовно пам’яті, то дослідження довело, що використання пам’яті підходом МНІДП, як правило, занизьке для сьогоднішніх стандартів, але може швидко збільшуватися при проведенні аналізу великих систем. Виключенням стає випадок з ядром Linux Kernel, де максимальне використання пам’яті склало 12,5 ГБ. Цей показник є суттєвим, незважаючи на те, що такі системи скоріш за все вважаються виключення, ніж правилом, з точки зору розміру кодової бази. Що стосується відмови МНІДП запустити інкрементний крок для того ж проекту через брак пам’яті, занепокоєння не викликає. Все через те, що це було викликано перевіркою виправлень системи контролю версій, яка виконується в запропонованому додатку. При реальному сценарії, коли моделювання комітів не відбувається, ця операція не відпрацьовується, що й буде усувати будь-які стрибки використання пам’яті.

1.9.2 Порівняння МНІДП та SIG

Доволі цінна інформація отримана в результаті проведеного аналізу з використанням внутрішнього інструменту SAT SIG. По-перше було виявлено, що коли в якості початкових даних використовуються більш крупні системи, то SAT показує всі недоліки. Хоча всі експерименти дослідження були проведені на високотехнологічній машині, на виконання аналізу Tensorflow знадобилося для завершення більше ніж 8 годин. В той же час SAT не зміг провести аналіз більших і складніших кодів бази, таких як OpenJDK та Linux Kernel. Це виступило обмеженням інструменту SIG, незважаючи на те, що кількість подібних систем невелика.

З’ясовано, що процес виявлення блоків коду з повторами та надлишковістю в рамках SAT займає дуже велику частку від загального часу аналізу, після того, як проведені додаткові вимірювання часу. Однак деякі цікаві висновки отримані після порівняння цих вимірів з результатами експериментів МНІДП за допомогою

серії аналізів комітів. Зокрема, загальний витрачений час швидко зростає, оскільки необхідно піддавати аналізу все більше і більше змін, хоча самі вимірювання виглядають невеликими, коли виконується аналіз однієї зміни системи. Стає очевидним, що традиційний детектор, такий, яким користується SIG, виявляється непрактичним при інкрементному налаштуванні. Оскільки проблема б не виникала, в разі низької частоти вхідних комітів, але не в епоху сьогодення, коли кожного дня іде здійснюються десятків або навіть сотень комітів.

1.9.3 Порівняння МНІДП та розширення на основі ЛЧХ

Проведені в дослідженні експерименти також надали деяку корисну інформацію на основі оцінки ефективності запропонованого розширення на основі ЛЧХ. Конкретніше, в деяких випадках у порівнянні з реалізацією МНІДП, етап створення індексу в підході на основі ЛЧХ виявився в два, а в деяких випадках в три рази повільнішим. Можливою причиною, яка призводить до цього може бути складність операції MinHash, яка складає значну частину загальної схеми ЛЧХ. Це стає очевидним, якщо враховувати, що хешування кожної черепиці для кожного набору черепиць під час MinHashing має проводитися обумовленими k хеш-функціями. Навпаки, з точки зору використання пам’яті, що призвело до скорочення у п’ять разів для випадку Linux, ЛЧХ виявився набагато ефективнішим. Однак, тільки це не являється виправданням для використання даного підходу замість МНІДП.

Вимірювання інкрементного кроку, крім цього, не відповідало нашим первинним очікуванням. Передбачалося, що процес інкрементного кроку реалізації на основі у порівнянні з МНІДП буде значно повільнішим, через обчислення записів індексу на льоту. Однак в ході проведеного дослідження було отримано протилежні результати. Практично, це виправдовувалося тим, що використаний поріг подібності не став причиною виявлення великої кількості

збігів між вихідними файлами. Для отримання кращого розуміння його поведінки виникає необхідність проведення подальших досліджень зв’язку часу виконання, потрібного для потоку інкрементного кроку реалізації на основі ЛЧХ та порогу подібності.

Другим висновком виявилося те, що для прогнозування часу, необхідного для інкрементного кроку процесу, не буде виступати хорошим показником розмір програмної системи. Вочевидь, на час впливатимуть наступні фактори: кількість файлів в коміті; тип змін та довжина (в LOC) цих файлів. При проведенні майбутньої розробки треба буде додатково вивчити поведінку цього робочого процесу у відношенні цих змінних.

На основі отриманих результатів дослідження виявлено, що підхід МНІДП працює краще, що довели виміри інкрементного кроку двох реалізацій. Більш докладні висновки стосовно інкрементного кроку можна буде зробити на основі проведення в майбутньому додаткових експериментів для реалізації на основі ЛЧХ. Однак будь-яка спроба буде марною без проведення попереднього дослідження того, що час виконання етапу створення індексу реалізації на основі ЛЧХ може бути зменшений до кращого або аналогічного рівня, як у підході МНІДП.

1.10 Висновки

Результати експериментів проведеного дослідження для детектора МНІДП та тих, що представлені в роботі Хаммела та ін. [14] не мають повного узгодження. Спостерігалася значна різниця в вимірах часу для створення індексу та інкрементного кроку робочого процесу. Звісно, були очікування, що під впливом трьох основних факторів при проведенні вимірювань, будуть присутні незначні відхилення. Цими факторами виступили:

– експериментальний пристрій. На відміну від апаратного забезпечення, яке було задіяло в оригінальному дослідженні Хаммела та ін. [14], забезпечення, яке використовувалося в наших експериментах, працювало набагато швидше;

– нормалізація. Виключення в дослідженні запропонованої реалізації етапу токенизації оригінального детектора, для врахування незалежності від мови програмування, призводить до видалення додаткових накладних витрат, необхідних для цього процесу;

– збереження у пам’яті. В роботі було використано постійність пам’ять на відміну від оригінального дослідження, де вимірювання стосувалися реалізації, яка зберігала проміжну інформацію в базі даних.

З урахуванням вище згаданих факторів очікувалося, що час, необхідний для створення індексу буде зменшений. Хоча очікування підтверджується отриманими висновками, але різниця набагато більша, ніж може бути виправдана цими факторами. Зокрема, час потрібний для створення індексу для проекту, який складається приблизно з 40 мільйонів рядків коду (LOC), згідно початковому дослідженню становив 7 годин і 4 хвилини. Однак, при проведенні нашого експерименту такий показник був вже трохи більше 5 хвилин для МНІДП та близько 15 хвилин для підходу на основі ЛЧХ, стосовно ядра Linux Kernel з розміром вдвічі меншим. Ці виміри вказують на різницю часу обробки приблизно в 42 та 14 факторів, відповідно, за припущенням, що підхід Хаммела та ін. масштабується лінійно. На основі цього спостереження, можна сказати, будь які спроби в подальшому покращень створення індексу будуть доволі складними, оскільки це проходить набагато швидше, ніж очкувалося.

Стосовно додаткового кроку, отримані з проведеного в роботі експерименту результати дають нове розуміння у порівнянні з оригінальним дослідженням. Більш конкретно, отримані виміри, дозволяють зробити висновок про те, що розмір системи впливає на загальний час інкрементного кроку. Спочатку це не логічно, оскільки запит і оновлення хеш-індексу повинні призвести до подібних вимірювань постійного часу. Однак, можливе виникнення колізій, які призводять до додаткових накладних витрат у загальному обчисленні у великих системах, де

індекс заповнений дуже великою кількістю записів. Крім цього, вплив на окремі вимірювання здійснюється декількома факторами, такими як: кількість файлів в коміті, довжина цих файлів, а також тип змін, які вони вносять. Нарешті, у початковому дослідженні відбувається імітація комітів шляхом випадкового видалення і повторного додавання вихідних файлів. Оскільки в нашому дослідженні використовувалися фактичні коміти, то результати дозволяють краще зрозуміти поведінку робочого процесу інкрементного кроку.

1.10.1 Відповідність поставленим задачам

Внутрішня відповідність

SAT вимірювання. Для порівняння продуктивності запропонованого мовно-незалежного інкрементного детектора повторів та надлишковості з характеристиками традиційного детектора, який має вбудований SAT, було використано інструмент SAT SIG. При цьому SAT є складним інструментом, який утворюється множиною різноманітних компонентів. Одні з них використовуються для розрахунку показників ремонтопридатності, інші пов’язані з виявленням блоків в програмному коді з повторами (клонів). Так як SAT доволі складний і має велику кількість підкроків, то ізоляція операцій, які пов’язані з виявленням клонів коду виступає доволі непростою задачею. Хоча проводився ретельний відбір підмножини операцій для який відбувалися вимірювання витраченого часу, можуть бути додаткові невраховані підетапи. При цьому не очікується, що якщо такі кроки і існують, то внесуть великий вклад в виміряний час, що минув і зробить отримані висновки недійсними.

Охоплення клонами

Проведені додаткові експерименти з використанням тестів виявилися б ідеальними для ретельної перевірки, стосовно охоплення клонами для детектора МНІДП. Тобто те, чи вірно запропонований інструмент визначає всі сценарії, при

яких може виникнути блок коду з повторами та надлишковістю. В дослідженні вручну відбувалося створення комітів, які імітують декілька варіантів використання: зміну, додавання, перейменування чи видалення файлу, в контексті коміту. Тим не менш, може бути виникнення крайніх випадків, які не розглядалися і тому не піддавалися тестуванню на МНІДП.

Зовнішня відповідність

В досліджені було використано набір даних програмних систем, який нараховує тільки п’ять одиниць. Хоча відбір проводився таким чином, щоб системи навмисно відрізнялися за розміром кодової бази та мовам програмування, на яких були написані, нові результати можуть бути отримані шляхом експериментів з більш крупним інформаційним фондом систем.

1.10.2 Застосування в рамках SIG та потенційне використання

Проведене дослідження дає можливість розглянути придатність запропонованих детекторів в контексті SIG. Спочатку проведено заглиблення у потенційне використання цих детекторів, а далі розглянуто деякі важливі моменти, які треба було б врахувати, якщо будь-який з цих детекторів були частиною колекції SIG.

Згідно з [81] аналітичні моделі SIG не включають можливість збереженості для всієї кодової бази, яка використовує дельта-модель ремонтопридатності (DMM – Delta Maintainability Model), а лише тільки для вимірів змін в коді. Замість того, що вимірювати вплив змін коду на ремонтопридатність всієї системи, дана модель дозволяє поступово вимірювати зміни коду та оцінювати якість їх коду. Однак, що стосується виявлення клонів (блоків коду з повторами та надлишковістю), то модель DMM може виявити блоки з повторами та надлишковістю, які були знайдені в уражених файлах конкретного коміту. Це не є оптимальним, оскільки клон може бути пропущений, через те, що додавання у

файл A, може призвести до появи клону з файлом B, а останній не буде являтися частиною файлів, які зачеплені комітом. Потенційно обидві реалізації, які представлені в даному дослідженні, є можливість використовувати в якості зовнішнього інструменту, який застосовує DMM, або навіть вбудований в реалізацію DMM чи для подолання вище описаної проблеми. Для вирішення пов'язаного з DMM обмеження, обидві реалізації фактично, підтримують виявлення блоків коду з повторами (клонами) та надлишковістю поза контекстом коміту. Крім цього, представлені детектори володіють двома суттєвими вимогами необхідними для потреб SIG: спроможність виявлення клонів 1-го типу та мовну незалежність. Також при додаванні кроку нормалізації з'являється потенціал для виявлення клонів 2-го типу. Але в цьому разі буде потреба у використання синтаксичного аналізатора для конкретної мови програмування, що надасть можливість усунути мовно-незалежну характеристику для цих детекторів, в рамках даного дослідження.

1.10.3 Огляд характеристик

Снепшот клонів

Однією з характеристик запропонованих інктементних детекторів являється те, що вони не видають повний снепшот усіх блоків з повторами та надлишковістю (клонів) в кодовій базі, а дають лише клони, які при кожній перевірці коду додавалися або видалялися. Для вирішення проблеми існують наступні варіанти:

– введення логіки керування клонами. Була б можливість агрегації клонів від початку кодової бази (або перевірки, коли всі блоки з повторами були відомі) до потрібної перевірки, переглядаючи ті, які були додані або видалені;

– введення параметру, який використовуючи кожний окремий файл в кодовій базі, дозволив би проводити запит індексу. З урахування того, що всі

файли системи мають пройти через процесс виявлення, який згадувався в даному дослідженні, то на подібне рішення буде витрачено дуже багато часу.

Сталість даних

При виявленні блоків коду з повторами (клонів) та надлишковістю, кожен з запропонованих детекторів користується проміжною інформацією, яка підходить для всіх версій. Ця інформація зберігається в пам’яті існуючими реалізаціями. Однак в реальних умовах кращим вибором буде база даних. В поточному налаштуванні, щоб обчислений індекс не був втрачений, додатки повинні завжди знаходитися в режимі очікування. Навпроти, це не було би вимогою до постійності жорсткого диску, з якого записи індексу є можливість отримати в будь-який момент часу.

Формат виведення

Під час розробки запропонованих в дослідженні детекторів, не проводився розгляд будь-якої потенційної інтеграції з інструментами SIG, які є зараз або будуть існувати в майбутньому. Інакше треба було б проводити дослідження, на предмет можливості зв’язку таких інструментів з нашими реалізаціями. Отже треба буде проводити додаткові дослідження, для врахування того, що формат виводу не був спроектований таким чином, щоб без перешкод користуватися ними.

Видалення коментарів

Видалення коментарів, нарешті, може бути функцією, яку потенційно бажано мати. Для можливості відповідним чином проводити обробку коментарів, написаних різними стилями, вимагатиметься додаткова попередня обробка кодової бази системи, і можливо, її сегментація за технологіями (мовами програмування). У контексті SIG процеси, які видаляють коментарі до коду, вже доступні і є можливість їх використання для цього. Тобто, в контексті

дослідження видалення коментарів є тільки питанням інтеграції даних процесів з запропонованими детекторами.

ВИСНОВКИ

Методи інкрементного виявлення блоків з повторами та надлишковістю в програмному коді, з’явилися через необхідність вирішувати питання ремонтопридатності, пов’язані з дублюванням програмного коду, у поєднанні з бажанням проводити багаторазовий запуск процесу виявлення фрагментів з клонами. Для більшості з існуючих подібних методів виникає потреба в мовному синтаксичному аналізаторі. А саме компоненті, який автоматично виключає можливість мовної незалежності для відповідних запропонованих детекторів клонів. Отже, мови програмування, не можуть бути проаналізовані в контексті виявлення блоків з повторами та надлишковістю, оскільки для них пошук або створення синтаксичного аналізатора є складною задачею.

Метою даного дослідження було визначення способу створення мовно-незалежного детектора повторів на надлишковості програмного коду та дослідження, яку інформацію має бути збереженою, для того, щоб детектор працював інкрементно (покроково). Для цього був модифікований оригінальний алгоритм Хаммела та ін. та проведені досліди використання ЛЧХ у спробі розширення та покращення оригінального підходу. В процесі експерементів виявлено, що для досягнення мовної незалежності, при цьому задовольняючи вимоги, встановлені на початку цього дослідження, можна використовувати проміжне представлення модифікованого «Індексу клону». В ході експериментів також встановлено, що підхід МНІДП виявився набагато швидший, ніж очікувалося, при цьому залишаючи обмежені можливості для подальшого покращення. Останнє пройшло перевірку в ході спостереження за продуктивністю реалізації, заснованої на ЛЧХ, яка працювала гірше в обумовленій формі. Враховуючі всі аспекти, в контексті даного дослідження, було побудовано мовно-незалежний інкрементний детектор повторів, заснований на підході Хаммела та ін. [14] та розширення його за допомогою ЛЧХ. Але в контексті оригінального

алгоритму потрібні додаткові дослідження для вивчення потенціалу підходу на основі ЛЧХ.

Крім цього, іншою метою даного дослідження було з'ясування адекватності роботи такого інкрементного підходу у порівнянні з традиційним детектором комерційного рівня, наприклад, вбудованим в інструмент SAT SIG.

Для цього було обрано набір даних з п'яти систем і проведено запуск SAT для кожної з них, з наступним вимірюванням часу, необхідного для виявлення блоків коду з повторами та надлишковістю. Отримати результати аналізу SAT для самих великих та складних систем нашого інформаційного фонду не вдалося, через те, що інструмент не зміг їх обробити. Однак, для менших систем все ж таки дані аналізу були отримані. У порівнянні з вимірюваннями інкрементного детектора, коли процес виявлення повторюється для серії комітів, результати вказують на значне покращення накопиченого часу виявлення клонів. Тобто, якщо б компанія SIG використала запропонований підхід, то це вплинуло на економію часу та ресурсів. Підґрунтям для можливої інтеграції запропонованих підходів в рамках моделі ремонтопридатності (DMM) виступає те, що вони мають здатність виявляти блоки з повторами та надлишковістю поза контекстом коміту [81].

Майбутні дослідження

Одним з направлень майбутніх досліджень – є подальше вивчення способів покращення придатності локально чутливого хешування для розширення мовно-незалежного інкрементного детектора повторів, запропонованого на початку даного дослідження.

Згідно з обговорюваними вимірами, доволі затратною по часу операцію конкретної схеми ЛЧХ, яка використана в даному дослідженні, виступає MinHashing. Досить складним процесом є те, що на цьому етапі кожна черепиця з кожного набору має бути хешована за допомогою k хеш-функцій. Подальші дослідження в області ЛЧХ призвели до спроб усунути дане вузьке місце. За

допомогою однієї хеш-функції можна досягти однієї поведінки, що показано в супутніх дослідженнях, таких як SuperMinHash [82] та Лі та ін. [83]. Безумовно, варто провести спроби, оскільки подібна оптимізація може призвести до значного покращення часу, який потрібен для створення відповідного індексу підходу, заснованого на ЛЧХ.

Реалізація на основі ЛЧХ, за своєю природою може призвести до зменшення часу відгуку. В даній реалізації, тільки для файлів, які були признані подібними на основі визначеного порогу подібності, відбувається запуск процесу виявлення блоків з повторами та надлишковістю. Тому експериментуючи з різними порогами подібності було б цікаво дослідити частку пропущених клонів. Якщо час необхідний для етапу створення індексу може бути зменшеним до рівнів, порівнянних з підходом МНІДП, то в такому випадку будуть корисними додаткові вимірювання, які відносяться до процесу інкрементного кроку в підході на основі ЛЧХ.

Кількість вихідних елементів, на яких застосовано метод, являються одним з факторів, який впливатиме на ефективність будь-якої схеми ЛЧХ. В даному дослідженні це відповідає кількості вихідних файлів в кодовій базі програмної системи.

Для кращого розуміння впливу цього фактору на продуктивність методу, виявляється вимірювання та порівняння продуктивності проектів однакового розміру (з точки зору кількості рядків – LOC), але з різною кількістю вихідних файлів.

Перелік джерел посиланнь:

1. Cunningham Ward. The wycash portfolio management system. ACMSIGPLANOOPS Messenger, 4(2):29–30, 1992.

2. Li Zengyang, Avgeriou Paris, Liang Peng. A systematic mapping study on technical debt and its management. Journal of Systems and Software, 101:193–220, 2015.

3. Buschmann Frank. To pay or not to pay technical debt. IEEE software, 28(6):29–31, 2011.

4. Fowler Martin. Refactoring: improving the design of existing code. Addison-Wesley Professional, 2018.

5. Rattan Dhavleesh, Bhatia Rajesh, Singh Maninder. Software clone detection: A systematic review. Information and Software Technology, 55(7):1165–1199, 2013

6. Abdullah Sheneamer, Jugal Kalita. A survey of software clone detection techniques. International Journal of Computer Applications, 137(10):1–21, 2016.

7. Miryung Kim, Vibha Sazawal, David Notkin, Gail Murphy. An empirical study of code clone genealogies. In Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, pages 187–196, 2005.

8. Roy Chanchal K, Cordy James R., Koschke Rainer. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of computer programming, 74(7):470–495, 2009.

9. Roy C. K., Cordy J. R. «An empirical study of function clones in open source software systems» в Proceedings of the 15th Working Conference on Reverse Engineering, 2008.

10. Chanchal K Roy, James R Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In 2009 International Conference on Software Testing, Verification, and Validation Workshops, pages 157–166. IEEE, 2009.

11. Bellon Stefan, Koschke Rainer, Antoniol Giulio, Krinke Jens, Merlo Ettore. Comparison and evaluation of clone detection tools. IEEE Transactions on software engineering, 33(9):577–591, 2007.

12. Heitlager Ilja, Kuipers Tobias, Visser Joost. A practical model for measuring maintainability. In 6th international conference on the quality of information and communications technology (QUATIC 2007), pages 30–39. IEEE, 2007.

13. Indyk Piotr, Motwani Rajeev. Approximate nearest neighbors: towards removing the curse of dimensionality. In Proceedings of the thirtieth annual ACM symposium on Theory of computing, pages 604–613, 1998.

14. Hummel Benjamin, Juergens Elmar, Heinemann Lars, Conradt Michael. Indexbased code clone detection: incremental, distributed, scalable. In 2010 IEEE International Conference on Software Maintenance, pages 1–9. IEEE, 2010.

15. Meyerovich Leo A, Rabkin Ariel S. Empirical analysis of programming language adoption. In Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications, pages 1–18, 2013.

16. Wit Michiel De, Zaidman Andy, Deursen Arie Van. Managing code clones using dynamic change tracking and resolution. In 2009 IEEE International Conference on Software Maintenance, pages 169–178. IEEE, 2009.

17. Roy Chanchal K, Zibran Minhaz F, Koschke Rainer. The vision of software clone management: Past, present, and future (keynote paper). In 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), pages 18–33. IEEE, 2014.

18. Li Zhenmin, Lu Shan, Myagmar Suvda, Zhou Yuanyuan. Cp-miner: Finding copy-paste and related bugs in large-scale software code. IEEE Transactions on software Engineering, 32(3):176–192, 2006.

19. Ivannikov V. P., Belevantsev A. A., Borodin A. E., Ignatiev V. N., Zhurikhin D. M., Avetisyan A. I. «Static analyzer Svace for finding defects in a source program code,» Programming and Computer Software, т. 40, № 5, pp. 265-275, 2014.

20. Kinder J. Static analysis of x86 executables. Ph.D. thesis, Technische Universitat Darmstadt, 2010

21. Balakrishnan G., Reps T. «WYSINWYX: What You See Is Not What You eXecute,» ACM Transactions on Programming Languages and Systems , т. 32, № 6, pp. 1-84, 2010.

22. Brumley D., Jager I., Avgerinos T., Schwartz E. J. «A Binary Analysis Platform,» Lecture Notes in Computer Science, т. 6806, 2011.

23. Rosen B., Wegman M. N., Zadeck. K. F. «Global value numbers and redundant computations,» в Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1988.

24. Song D., Brumley D., Yin H., Caballero J., Jager I., Kang M. G., Liang Z., Newsome J., Poosankam P., Saxena P. «BitBlaze: A New Approach to Computer Security via,» в In Proceedings of the 4th International Conference on Information Systems Security, 2008.

25. «IDA Pro» Hex-Rays [Електронний ресурс]. Режим доступу: https://www.hex-rays.com/products/ida.

26. «IDA F.L.I.R.T. Technology,» Hex-Rays [Електронний ресурс]. Режим доступу: https://www.hex-rays.com/products/ida/tech/flirt/index.shtml

27. Ferrante J., Ottenstein K., Warren J. «The program dependence graph and its use in,» Trans. on Prog. Lang. and Syst. (TOPLAS), pp. 319-349, 1987.

28. Cousot P., Cousot R. «Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points,» в Principles of Programming Languages (POPL), 1977.

29. Feist J., Mounier L., Potet ML. «Statically detecting use after free on binary code,» Journal of Computer Virology and Hacking Techniques, т. 10, № 3, pp. 211-217, 2014.

30. Cova M., Felmetsger V., Banks G., Vigna G. «Static Detection of Vulnerabilities in x86 Executables,» в 22nd Annual Computer Security Applications Conference, 2006.

31. «REIL - The Reverse Engineering Intermediate Language» Zynamics BinNavi [Електронний ресурс]. Режим доступу: https://www.zynamics.com/binnavi/manual/html/reil_language.htm.

32. Cheng S., Yang J., Wang J., Wang J., Jiang F. «LoongChecker: Practical Summary-Based Semi-simulation to Detect Vulnerability in Binary Code,» в 10th International Conference on Trust, Security and Privacy in Computing and Communications, Changsha, 2011.

33. «Zynamics BinNavi is the leading open source binary code reverse engineering tool based on graph visualization» Zynamics BinNavi [Електронний ресурс]. Режим доступу: https://www.zynamics.com/binnavi.html

34. Dewey D., Reaves B., Traynor P.. «Uncovering Use-After-Free Conditions in Compiled Code,» в 0th International Conference on Availability, Reliability and Security, Toulouse, 2015.

35. Ganapathy V., Seshia S. A., Jha S., Reps T.W., Bryant R. E. «Automatic discovery of API-level exploits,» в 27th International Conference on Software Engineering, Saint Louis, MO, USA, 2005.

36. Baker B. «On finding duplication and near-duplication in large software systems» в Proceedings of the 2nd Working Conference on Reverse Engineering, 1995.

37. Tairas R., Gray J. «Phoenix-based clone detection using suffix trees» в Proceedings of the 44th Annual Southeast Regional Conference, 2006.

38. Kamiya T., Kusumoto S., Inoue K. «CCFinder: A multilinguistic tokenbased code clone detection system for large scale source code,» в IEEE Transactions on Software Engineering, 2002.

39. Jiang L., Misherghi G., Su Z., Glondu S. «DECKARD : Scalable and accurate tree-based detection of code clones,» в Proceedings of the 29th International Conference on Software Engineering, 2007.

40. Sargsyan S., Kurmangaleev S., Belevantsev A., Aslanyan H., Baloian A. «Scalable tool for code clone detection based on semantic analysis of program,» Trudy. ISP RAS, т. 1, pp. 39-50, 2015.

41. Schulman A. «Finding binary clones with opstrings function digests: Part III» Dr. Dobb's Journal, 2005.

42. Karim M.E., Walenstein A., Lakhotia A., Parida L. «Malware phylogeny generation using permutations of code,» Computer Virology, pp. 13-23, 2005.

43. Comparetti P., Salvaneschi G., Kirda E., Kolbitsch C., Kruegel C., Zanero S. «Identifying dormant functionality in malware programs,» в Securityand Privacy (SP), 2010 IEEE Symposium on, 2010.

44. Svajlenko Jeffrey, Roy Chanchal K. Evaluating clone detection tools with bigclonebench. In 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 131–140. IEEE, 2015.

45. Sajnani Hitesh, Saini Vaibhav, Svajlenko Jeffrey, Roy Chanchal K, VLopes Cristina. Sourcerercc: Scaling code clone detection to big-code. In Proceedings of the 38th International Conference on Software Engineering, pages 1157–1168, 2016.

46. Ducasse Stéphane, Rieger Matthias, Demeyer Serge. A language independent approach for detecting duplicated code. In Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360), pages 109–118. IEEE, 1999.

47. Johnson J Howard. Substring matching for clone detection and change tracking. In ICSM, volume 94, pages 120–126, 1994.

48. Marcus Andrian, Maletic Jonathan I. Identification of high-level concept clones in source code. In Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001), pages 107–114. IEEE, 2001.

49. Kamiya Toshihiro, Kusumoto Shinji, Inoue Katsuro. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering, 28(7):654–670, 2002.

50. Baker Brenda S. On finding duplication and near-duplication in large software systems. In Proceedings of 2nd Working Conference on Reverse Engineering, pages 86–95. IEEE, 1995.

51. Li Liuqing, Feng He, Zhuang Wenjie, Meng Na, Ryder Barbara. Cclearner: A deep learning-based clone detection approach. In2017IEEEInternationalConference on Software Maintenance and Evolution (ICSME), pages 249–260. IEEE, 2017.

52. White Martin, Tufano Michele, Vendome Christopher, Poshyvanyk Denys. Deep learning code fragments for code clone detection. In 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 87–98. IEEE, 2016.

53. Baxter Ira D, Yahin Andrew, Moura Leonardo, Sant'Anna Marcelo, Bier Lorraine. Clone detection using abstract syntax trees. In Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272), pages 368–377. IEEE, 1998.

54. Jiang Lingxiao, Misherghi Ghassan, Su Zhendong, Glondu Stephane. Deckard: Scalable and accurate tree-based detection of code clones. In Proceedings of the 29th international conference on Software Engineering, pages 96–105. IEEE Computer Society, 2007.

55. Mayrand Jean, Leblanc Claude, Merlo Ettore. Experiment on the automatic detection of function clones in a software system using metrics. In icsm, volume 96, page 244, 1996.

56. Kontogiannis Kostas A, DeMori Renator, Merlo Ettore, Galler Michael, Bernstein Morris. Pattern matching for clone and concept detection. Automated Software Engineering, 3(1-2):77–108, 1996.

57. Krinke Jens. Identifying similar code with program dependence graphs. In Proceedings Eighth Working Conferenceon Reverse Engineering, pages301–309.IEEE,2001.

58. Komondoor Raghavan, Horwitz Susan. Using slicing to identify duplication in source code. In International static analysis symposium, pages 40–56. Springer, 2001.

59. Liu Chao, Chen Chen, Han Jiawei, Yu Philip S. Gplag: detection of software plagiarism by program dependence graph analysis. In Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 872–881. ACM, 2006.

60. Funaro Marco, Braga Daniele, Campi Alessandro, Ghezzi Carlo. A hybrid approach (syntactic and textual) to clone detection. In Proceedings of the 4th International Workshop on Software Clones, pages 79–80. ACM, 2010.

61. Agrawal Akshat, Yadav Sumit Kumar. A hybrid-token and textual based approach to find similar code segments. In 2013 fourth international conference on computing, communications and networking technologies (ICCCNT), pages 1–4. IEEE, 2013.

62. Saini Vaibhav, Farmahinifarahani Farima, Lu Yadong, Baldi Pierre, VLopes Cristina. Oreo: Detection of clones in the twilight zone. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 354–365, 2018.

63. Göde Nils, Koschke Rainer. Incremental clone detection. In 2009 13th European Conference on Software Maintenance and Reengineering, pages 219–228. IEEE, 2009.

64. Nguyen Tung Thanh, Nguyen Hoan Anh, Al-Kofahi Jafar M, Pham Nam H, Nguyen Tien N. Scalable and incremental clone detection for evolving software. In 2009IEEEInternationalConferenceonSoftwareMaintenance, pages491–494.IEEE, 2009.

65. Higo Yoshiki, Yasushi Ueda, Nishino Minoru, Kusumoto Shinji. Incremental code clone detection: A pdg-based approach. In 2011 18th Working Conference on Reverse Engineering, pages 3–12. IEEE, 2011.

66. Ragkhitwetsagul Chaiyong, Krinke Jens. Siamese: scalable and incremental code clone search via multiple code representations. Empirical Software Engineering, pages 1–49, 2019.

67. Beyer Kevin, Goldstein Jonathan, Ramakrishnan Raghu, Shaft Uri. Whenis“neaest neighbor” meaningful? In International conference on database theory, pages 217–235. Springer, 1999.

68. Leskovec Jure, Rajaraman Anand, Ullman Jeffrey David. Mining of Massive Datasets. Cambridge University Press, USA, 2nd edition, 2014. ISBN 1107077230.

69. Gionis Aristides, Indyk Piotr, Motwani Rajeev, et al. Similarity search in high dimensions via hashing. In Vldb, volume 99, pages 518–529, 1999.

70. Sood Sadhan, Loguinov Dmitri. Probabilistic near-duplicate detection using simhash. In Proceedings of the 20th ACM international conference on Information and knowledge management, pages 1117–1126, 2011.

71. Broder Andrei Z. On the resemblance and containment of documents. In Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171), pages 21–29. IEEE, 1997.

72. Charikar Moses S. Similarity estimation techniques from rounding algorithms. In Proceedings of the thiry-fourth annual ACM symposium on Theory of computing, pages 380–388, 2002.

73. Cohen Edith. Size-estimation framework with applications to transitive closure and reachability. Journal of Computer and System Sciences, 55(3):441–453, 1997.

74. Verroios Vasilis, Garcia-Molina Hector. Top-k entity resolution with adaptive locality-sensitive hashing. In 2019 IEEE 35th International Conference on Data Engineering (ICDE), pages 1718–1721. IEEE, 2019.

75. Ebraheem Muhammad, Thirumuruganathan Saravanan, Joty Shafiq, Ouzzani Mourad, Tang Nan. Distributed representations of tuples for entity resolution. Proceedings of the VLDB Endowment, 11(11):1454–1467, 2018.

76. Cappelli Raffaele, Ferrara Matteo, Maltoni Davide. Fingerprint indexing based on minutia cylinder-code. IEEE transactions on pattern analysis and machine intelligence, 33(5):1051–1057, 2010.

77. Zhou Wei, Hu Jiankun, Wang Song. Enhanced locality-sensitive hashing for fingerprint forensics over large multi-sensor databases. IEEE Transactions on Big Data, 2017.

78. Das Abhinandan S, Datar Mayur, Garg Ashutosh, Rajaram Shyam. Google news personalization: scalable online collaborative filtering. In Proceedings of the 16th international conference on World Wide Web, pages 271–280, 2007.

79. Zhang Kunpeng, Fan Shaokun, Wang Harry Jiannan. An efficient recommender system using locality sensitive hashing. In Proceedings of the 51st Hawaii International Conference on System Sciences, 2018.

80. Pascarella Luca, Bruntink Magiel, Bacchelli Alberto. Classifying code comments in java software systems. Empirical Software Engineering, 24(3):1499–1537, 2019.

81. di Biase Marco, Rastogi Ayushi, Bruntink Magiel, van Deursen Arie. The delta maintainability model: measuring maintainability of fine-grained code changes. In 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), pages 113–122. IEEE, 2019.

82. Ertl Otmar. Superminhash-a new minwise hashing algorithm for jaccard similarity estimation. arXiv preprint arXiv:1706.05698, 2017.

83. Li Ping, Owen Art, Zhang Cun-Hui. One permutation hashing for efficient search and learning. arXiv preprint arXiv:1208.1259, 2012.

ДОДАТОК А
КОЛЕКЦІЯ ІНФОРМАЦІЙНОГО ФОНДУ

Додаток містить інформацію про стан кожної з програмних систем у запропонованому наборі даних, які використовувалися під час аналізу в дослідженні

В таблиці А.1 відображений стан сховища на основі git кожного проекту з відкритим кодом, який було застосовано під час проведення аналізу в дослідженні. Конкретніше, іде вказування на основну гілку, яка використовувалася разом з інформацією про найостанній (найновіший) коміт для конкретної гілки. Для отримання версії програмної системи, проаналізовані в дослідженні необхідно починаючи з зафіксованих комітів для кожної з системи і повертаючись назад до 50 комітів в кожному випадку – виключаючи злиття.

Таблиця А.1 – Снепшоти проекту на момент аналізу

Проект	Гілка	HEAD commit id	HEAD commit meta-info
Rippled	develop	cd78ce3	Автор: Carl Hua Дата: Tue Apr 14 09:50:15 2020 -0400 Додавання автоматизації PR для проектних плат
Kooboo	master	39406b7	Автор: CN name Дата: Mon Jun 15 09:58:57 2020 +0800 fixed monaco bug
Tensorflow	master	f926d8c	Автор: A. Unique TensorFlower Дата: Mon Jun 15 03:38:44 2020 -0700 Представляє новий експериментальний пакет, який: - визначає схему налаштування делегатів - визначає механізм плагіна C ++ за допомо-гою схеми ...
Openjdk-jdk14u	master	e23aaed	Автор: Prasadrao Koppula Дата: Thu Jun 11 21:54:51 2020 +0530 8246031: SSLSocket.getSession() doesn’t close...
Linux	master	9cb1fd0	Автор: Linus Torvalds Дата: Sun May 24 15:32:54 2020 -0700 Linux 5.7-rc7

HEAD commit id – посилання на ідентифікатор коміту

HEAD commit meta-info – посилання на ідентифікатор метаінформації коміту

ДОДАТОК Б

ВИКЛЮЧЕНІ КАТАЛОГИ ТА ФАЙЛИ

В таблиці Б.1 представлені каталоги та розширення, які ігнорувалися обома запропонованими реалізаціями. На двох різних етапах проходить подібне виключення: під час початкового розбору кодової бази та під час аналізу включених до коміту змін.

ВИКЛЮЧЕНІ КАТАЛОГИ	вузлові модулі, активи, збірка, класи, gradle, ліцензії, icu, dcn21, пристрої, документи, тест, тести, приклади, журнали змін
ВИКЛЮЧЕНІ ФАЙЛИ	.3dm, .3ds, .3g2, .3gp, .7z, .a, .aac, .adp, .ai, .aif, .aiff, .alz, .ape, .apk, .ar, .arj, .asf, .au, .avi, .bak, .baml, .bh, .bin, .bk, .bmp, .btif, .bz2, .bzip2, .cab, .caf, .cgm, .class, .cmx, .cpio, .cr2, .cur, .dat, .dcm, .deb, .dex, .djvu, .dll, .dmg, .dng, .doc, .docm, .docx, .dot, .dotm, .dra, .DS Store, .dsk, .dts, .dtshd, .dvb, .dwg, .dxf, .ecelp4800, .ecelp7470, .ecelp9600, .egg, .eol, .eot, .exe, .f4v, .fbs, .fh, .fla, .flac, .fli, .flv, .fpx, .fst, .fvt, .g3, .gh, .gif, .epub, .graffle, .gz, .gzip, .h261, .h263, .h264, .icns, .ico, .ief, .img, .ipa, .iso, .jar, .jpeg, .jpg, .jpgv, .jpm, .jxr, .key, .ktx, .lha, .lib, .lvp, .lz, .lzh, .lzma, .mng, .lzo, .m3u, .m4a, .m4v, .mar, .mdi, .mht, .mid, .midi, .mj2, .mka, .mkv, .mmr, .mobi, .mov, .movie, .mp3, .mp4, .mp4a, .mpeg, .mpg, .mpga, .mxu, .nef, .npx, .numbers, .nupkg, .o, .oga, .ogg, .ogv, .otf, .pages, .pbm, .pcx, .pdb, .pdf, .pea, .pgm, .pic, .png, .pnm, .pot, .potm, .potx, .ppa, .ppam, .ppm, .pps, .ppsm, .ppsx, .ppt, .pptm, .pptx, .psd, .pya, .pyc, .pyo, .pyv, .qt, .rar, .ras, .raw, .resources, .rgb, .rip, .rlc, .rmf, .rmvb, .rtf, .rz, .s3m, .s7z, .scpt, .sgi, .shar, .sil, .whl, .sketch, .slk, .smv, .snk, .so, .stl, .suo, .sub, .swf, .tar, .tbz, .tbz2, .tga, .xlam, .tgz, .thmx, .tif, .tiff, .tlz, .ttc, .ttf, .txz, .udf, .uvh, .uvi, .uvm, .uvp, .uvs, .uvu, .viv, .vob, .war, .wav, .wax, .wbmp, .wdp, .weba, .webm, .webp, .wim, .wm, .wma, .wmv, .wmx, .woff, .woff2, .wrm, .wvx, .xbm, .xif, .xla, .xls, .xlsb, .xlsm, .xlsx, .xlt, .xltm, .xltx, .xm, .xmind, .xpi, .xpm, .xwd, .xz, .z, .zip, zipx, .txt, .md, .bat, .jks, .sh, .prpt, .ini, .db, .plist, .ver, .pb, .data-00000-of-00001, .index, .golden, .pbtxt.gz, .mdb, .meta, .bytes, .lite, .h5, .data-00000-of-00002, .data-00001-of-00002, .map, .elf, .skb, .skp, .dtbo, .mat, .dll, .Rascal, .exr, .blend, .pfb, .xcf, .odg, .out, .sgml, .pfx, .fig, .mo, .install

ДОДАТОК В
(Обов’язковий)

КОПІЇ НАУКОВИХ ПУБЛІКАЦІЙ

ISSN 2307-5732
DOI 10.31891/2307-5732

НАУКОВИЙ ЖУРНАЛ **3.2021**

ВІСНИК

Хмельницького національного університету

Технічні науки

Technical sciences

SCIENTIFIC JOURNAL
HERALD OF KHMELNYTSKYI NATIONAL UNIVERSITY

2021, Issue 3, Volume 297

Хмельницький

ВІСНИК
ХМЕЛЬНИЦЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ
серія: Технічні науки

Затверджений як фахове видання категорії «Б»,
РІШЕННЯ АТЕСТАЦІЙНОЇ КОЛЕГІЇ № 1643 ВІД 28.12.2019 та №409 від 17.03.2020

Засновано в липні 1997 р. ***Виходить 6 разів на рік***

Хмельницький, 2021, № 3(297)

Засновник і видавець: Хмельницький національний університет
(до 2005 р. – Технологічний університет Поділля, м. Хмельницький)

Включено до науково-метричних баз:

Google Scholar	http://scholar.google.com.ua/citations?hl=uk&user=alUP9OYAAAAJ
Index Copernicus	http://jml2012.indexcopernicus.com/passport.php?id=4538&id_lang=3
Polish Scholarly Bibliography	https://pbn.nauka.gov.pl/journals/46221
CrossRef	http://doi.org/10.31891/2307-5732

Технічний редактор Горященко К. Л., к.т.н.
Редактор-коректор Броженко В. О.

Рекомендовано до друку рішенням вченої ради Хмельницького національного університету, протокол № 17 від 24.06.2021 р.

Адреса редакції: редакція журналу "Вісник Хмельницького національного університету"
Хмельницький національний університет
вул. Інститутська, 11, м. Хмельницький, Україна, 29016

☎ (038-2) 67-51-08 **web:** http://journals.khnu.km.ua/vestnik
http://lib.khnu.km.ua/visnyk_tup.htm

e-mail: visnyk.khnu@khmnu.edu.ua
visnyk.khnu@gmail.com

Зареєстровано Міністерством України у справах преси та інформації.
Свідоцтво про державну реєстрацію друкованого засобу масової інформації
Серія КВ № 9722 від 29 березня 2005 року

ЗМІСТ

ЕКОЛОГІЯ

КОМП'ЮТЕРНІ НАУКИ, ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ, СИСТЕМНИЙ АНАЛІЗ ТА КІБЕРБЕЗПЕКА

МАШИНОБУДУВАННЯ, МЕХАНІКА ТА МАТЕРІАЛОЗНАВСТВО

ЕЛЕКТРОМЕХАНІКА, ЕЛЕКТРОТЕХНІКА ТА ЕНЕРГЕТИКА

АВТОМАТИЗАЦІЯ, ТЕЛЕКОМУНІКАЦІЇ ТА РАДІОТЕХНІКА

ТЕХНОЛОГІЇ ХІМІЧНОЇ, ХАРЧОВОЇ ТА ЛЕГКОЇ ПРОМИСЛОВОСТІ

МОДЕЛЮВАННЯ ТА ПРОЕКТУВАННЯ ТЕХНІЧНИХ СИСТЕМ

DOI 10.31891/2307-5732-2021-297-3-39-45
УДК 004.056.5:621

Н. І. ПРАВОРСЬКА, О. В. БАРМАК, Д. М. МЕДЗАТИЙ
Хмельницький національний університет
Т. В. ШЕСТАКЕВИЧ
Львівська політехніка

ПРОЦЕС ВИЯВЛЕННЯ БЛОКІВ З ПОВТОРАМИ І НАДЛИШКОВІСТЮ ПРИ ВИКОРИСТАННІ МОВНО-НЕЗАЛЕЖНОГО ІНКРЕМЕНТНОГО ДЕТЕКТОРА

Для уникнення порушень нормального функціонування розробленого програмного забезпечення, спричиненого помилками, навіть коли розробкою займаються професіонали, використовується ряд автоматизованих інструментів, які дають змогу проводити оцінювання програмного коду. Для виявлення помилок, які з'являються через дублювання блоків виконуваного коду, зазвичай застосовують різноманітні детектори. Важливість при розробці подібних детекторів полягає в тому, щоб продукт не був залежним від мови програмування та мав нескладний алгоритм знаходження клонованих блоків коду. В основі підходу мовно-незалежного детектора повторів покладено метод, який базується на використанні індексу клону. Він представляє собою глобальну структуру даних, яка нагадує типовий інвертований індекс. За основу такого підходу береться текст, тобто метод стає базою для досліджень незалежних від мови.

Ключові слова: програмний код, мовно-незалежний детектор, інкрементний підхід, робочий процес, індекс повторення, індекс клону, хеш-функція, хеш-значення, коміт, репозиторій.

N. I. PRAVORSKA, O. B. BARMAC, D. M. MEDZATIY
Khmelnytsky National University
T. V. SHESTAKEVYCH
Lviv Polytechnic

THE PROCESS OF DETECTING BLOCKS WITH REPETITIONS AND EXCESS BUILDING USING A LANGUAGE-INDEPENDENT INCREASE DETECTOR

To avoid malfunctions of the developed software caused by errors, even when developed by professionals, a number of automated tools are used, which allow to evaluate the software code. A variety of detectors are commonly used to detect errors that occur due to duplicate blocks of executable code. The importance of developing such detectors is that the product is not dependent on the programming language and has a simple algorithm for finding cloned blocks of code. The approach of the language-independent repetition detector is based on a method based on the use of the clone index. It is a global data structure that resembles a typical inverted index. This approach is based on the text, ie the method becomes the basis for research independent of language. In recent years, additional methods have become increasingly popular, which analyze the source and executable code at a smaller level, and there are attempts to avoid unnecessary recalculations, by transferring information between versions.

Reviewing the research presented in the works of scientists dealing with this problem, it was decided to propose an approach to improve methods for detecting repetitions and redundancy of program code based on language-independent incremental repetition detector (MNIDP). Most additional research is based on tree-like and graphical methods, ie they are strictly dependent on the programming language. The solution in the MNIDP campaign is to take the text as a basis, ie the method becomes the basis for research independent of language. This technique is not strictly language-independent, but due to the fact that the tokenization stage will be included, with the help of minor adjustments the desired result has been achieved. This provides a detailed analysis of the internal composition (namely, elements) of the detector and explanations of the work at different stages of the detection process.

Keywords: program code, language-independent detector, incremental approach, workflow, iteration index, clone index, hash function, hash value, commit, repository.

Постановка проблеми

Порушення нормального режиму функціонування розробленого програмного забезпечення іноді можуть виникнути при допущенні помилок навіть, коли цим займаються фахівці професіонали. Незважаючи на це, правки можна внести на любому з етапів життєвого циклу програмного продукту. Однак, якщо цей процес буде тривати до останніх етапів розробки ПЗ, то витрати як самої розробки, так і супроводу (а саме фінансові і часові) можуть стати дуже великими. При цьому треба враховувати, що при експлуатації помилки у програмному забезпеченні, яке використовується в деяких важливих людських сферах (наприклад, медицині, транспорті та тому подібне) можуть становити небезпеку життю та здоров'ю людини. Тому розробники, зазвичай, при розробці програмних продуктів активно використовують інструменти, спроможні проаналізувати код та виявити дефекти на ранніх стадіях життєвого циклу.

При розборі робіт, які є на сьогодні актуальними в питаннях аналізу виконуваного коду, спостерігається наступна тенденція, проводиться розбиття аналізу на статичний та динамічний. Кожний з вище згаданих підходів має як свої переваги, так і недоліки. При статичному аналізі треба використовувати всі можливі шляхи виконання та всі значення змінних. Таким чином, статичній аналіз має змогу виявити дефекти на відміну від динамічного аналізатора, навіть, якщо такий використовувався доволі довгий час. Тобто динамічний аналізатор може виявляти подібне лише в тому разі, якщо виконуваний шлях пройшов через точку дефекту, при деяких значеннях змінних. На відміну від динамічних аналізаторів статичним, зазвичай, не треба інформації про вхідні дані. Окрім цього, перші потребують використання емуляторів чи апаратури, якій притаманна архітектура виконуваного коду, який підпадає під аналіз.

Левова частка сучасних методів аналізу виконуваного коду, використовують звичайний спосіб в підході виявлення блоків з повторами та надлишковостями. Однак, в останні роки, все більше набувають

популярності додаткові методи, які проводять аналіз вихідного та виконуваного коду на більш дрібному рівні, причому є намагання уникнути зайвих перерахунків, завдяки передачі інформації між версіями. Незважаючи на підходи, методи класифікують за представленням вихідного коду на такі основні категорії:

– текстові підходи;
– підходи, основані на токенах;
– синтаксичні підходи;
– семантичні підходи;
– підходи, засновані на поведінці програми.

На відміну від досліджень, які використовують інструменти повного системного аналізу, розробок, що пропонують додаткові методи є небагато. Переважно вони зосереджуються на підходах, основою яких становлять токени, дерева або графіки. Це буде вимагати від синтаксичного аналізатора або парсера проводити побудову необхідних структур даних спираючись на відповідну мову (мови) програмування системи. Фактично з самого початку більшість з них не мають намір підтримувати незалежність від мови програмування. Таким чином вони розробляють детектори, які з одного боку спроможні виявляти більш складні типи клонованого коду (блоків коду з повторами та надлишковістю), але з іншого потребують налаштування для конкретної мови програмування.

Аналіз останніх джерел

Натомість, вперше була запропонована інкрементна методика виявлення клонів авторами роботи Гоуд та Кошке [1]. Для представлення вихідного коду було рекомендовано задіяти деревоподібну техніку, яка використовує глобальну структуру даних узагальненого суфіксного дерева (УСД). Подібна техніка на базі дерева ClemanX пропонувалася у праці Нгуена та ін. [2]. Тут кожний вихідний файл представляється як АСД (абстрактне синтаксичне дерево). Для забезпечення співставлення подібності кожне піддерево такого АСД, додатково представлялося характеристичним вектором.

Згодом було запропоновано додаткове виявлення блоків з повторами і надлишковістю коду у праці Хіго та ін. [3] на основі ГЗП (графів залежності програми). В базі даних проводиться збереження ГЗП кожного методу кожного оновленого файлу, який побудовано на етапі аналізу. На наступному кроці виявлення, після ручного введення користувачем запускається вибірка відповідних графів ЗП з бази даних, які є основою для виявлення клонів. Інший алгоритм, який базується на основі індексу, запропонований в роботі Хаммела та ін. [4]. Основною структурою даних, яка використовується, виступає індекс клону. Він представляє собою глобальну структуру даних, яка нагадує типовий інвертований індекс. Цей метод, хоча і використовує лексичний аналізатор для перетворення вихідного коду в токени, але серед всієї літератури з цього питання є найближчим до незалежного від мови інкрементного підходу.

Проводячи огляд досліджень, викладених в роботі Хаммела та ін. [4] було вирішено запропонувати підхід удосконалення методів виявлення повторів та надлишковості програмного коду на основі мовно-незалежного інкрементного детектора повторів (МНІДП). Більшість додаткових досліджень, за основу мають деревоподібні та графічні методи, тобто вони суворо залежать від мови програмування. Рішенням в поході МНІДП – є взяття за основу тексту, тобто метод стає базою для досліджень незалежних від мови. Ця методика не являється сугубо мовно-незалежною, але через те, що буде включений етап токенізації, за допомогою незначних корегувань досягнуто потрібного результату. Вирішено провести деякі зміни, які забезпечили мовну незалежність і того, як вони відобразилися в запропонованому детекторі МНІДП. При цьому надається деталізація аналізу внутрішнього складу (а саме, елементів) детектора та пояснень роботи на різних етапах процесу виявлення.

Огляд мовно-незалежного інкрементного детектора повторів. Проводиться поділ інкрементного методу даного дослідження на два основних робочих процеси. В свою чергу кожний буде включати в себе додатково ряд підпроцесів. Проміжне представлення пов'язується з першим процесом і зберігається для повторного використання в версіях проекту. Таке об'єднання (сектор, pool) буде прийматися за індекс повторення (Clone Index – індекс клону). Весь проект програмного забезпечення використовується цим процесом в якості вхідних даних. Запуск його відбувається одноразово на початку всього конвеєра виявлення повторів (клонованих блоків коду). Другий процес, який посилається на логіку, проводить запуск фактичної процедури виявлення повторів та надлишковості коду та виводить виявлені клони. Другий процес буде запущено після оновлення основної бази з кодами. В реальному налаштуванні це відбувається після того, як новий запис (коміт, commit) буде розміщено в репозиторій керування версіями.

Процес створення індексу дублювання. У робочому процесі, коли за його допомогою детектор повторень створює індекс повторення (Індекс клону), вирізняють два кроки. Процес представлено на рис. 1.

Рис. 1. Підкроки робочого процесу створення «Індексу клону»

Спершу на етапі попередньої обробки подається кожний з файлів програмного проекту, який піддався аналізуванню. Відбувається зміна коду (наприклад, прибираються зайві пусті рядки) та визначення ступеню деталізації порівняння. На наступному етапі проводиться групування операторів модифікованого вихідного коду в послідовності, основою яких виступає попередньо визначений параметр конфігурації. Він визначає розмір групування та потім відбувається хешування груп. В «Індексі клону» зберігаються отримані хеші разом з додатковими метаданими (ім'я файлу та індекс оператора в файлі).

Під час даного процесу не виконується жодної логіки виявлення повторів та надлишковості для визначення початкового стану бази кодів, що стосується дублювання. Тобто, всіх існуючих блоків повторень у кодовій базі системи поки що немає. Однак, це те, чого можна досягти проводячи запит «Індексу клону» з кожним файлом початкової бази кодів. Цей крок не буде застосований, оскільки не вимагається досліджувати еволюцію клонів, а зацікавленість була тільки в дублікатах коду, які були знищені або створені в кожній новій версії.

Послідовність кроків робочого процесу. Ініціювання робочого процесу відбувається кожного разу, при відправлені наступного нового збереження до репозиторія основаного на контролі версій, де розміщена відповідна програмна система. В контексті даного дослідження, так як запропонований МНДП не інтегрований з відповідною платформою керування версіями (наприклад, в якості плагіну), буде вручну змодельована процедура. Запуск такої процедури в іншому випадку проходив в реальних умовах автоматично. Виконання проводиться за допомогою файлу конфігурації JSON, який вказує оновлені файли, разом з типом відповідного оновлення для кожного збереження (commit), який буде аналізуватися. У типовому коміті міститься інформація, на основі якої відбувається генерація подібного файлу конфігурації. На рис. 2 представлено лістинг файлу конфігурації, призначеного для аналізу двох комітів.

```
{
  "commits": [
  {
    "id": "cb8f645e0f",
    "changes": [
      { "type": "M", "filename": "lib/plugins/loader.py" }
    ]
  },
  {
    "id": "564907d8ac",
    "changes": [
      { "type": "A", "filename": "fragments/test_refactor.yml" },
      { "type": "D", "filename": "fragments/arch_linux.json" },
      { "type": "R", "filename": [
        "test/facts/system/distribution/__init__.py",
        "test/facts/system/__init__.py" ]
      }
    ]
  }]
}
```

Рис. 2. Приклад файлу конфігурації JSON, який показує змінені файли за кожним комітом

В кожному коміті містяться файли, які були модифіковані (M), додані (A), знищені (D) або перейменовані (R). В більшості підетапи інкрементного робочого процесу нагадують такий же процес створення індексу. Фактично ідентичними являються етапи попередньої обробки і генерації хешу. В даному разі зв'язані операції будуть застосовані не до всієї кодової бази, а тільки до визначених файлів.

На наступному етапі проводиться порівняння генерованих хешів з хешами, які зберігаються в постійному «Індексі клону» під час процесу, який на рис. 3 вказаний як «Виявлення клону». Дублювання буде виявлене, якщо два хеши збігаються.

Рис. 3. Підетапи інкрементного робочого процесу

Через те, що відбувалася генерація хеш-значень шляхом хешування фіксованого числа операторів коду, наприклад N, то всі виявлені повтори (клони) будуть мати довжину N. Звідси випливає, що потрібне застосування додаткової логіки для виявлення і розширення блоків з повторами та надлишковістю до їх максимальної довжини.

Як видно з рис. 3 на останніх кроках необхідне оновлення «Індексу клону». Це дасть змогу провести на наступній ітерації порівняння хешів, які були створені файлами в майбутньому коміті з тими хешами, що збереглися в оновленому «Індексі клону». Ця операція буде виконуватися одночасно з виявленням в програмному коді блоків з повторами і надлишковістю.

Проведення попередньої обробки вихідного коду. У підході Хаммела та ін. [4] пропонується застосувати нормалізацію, яка на етапі попередньої обробки вихідного коду буде містити токенизацію. Виконується такий крок для виявлення клонів 2 типу (синтаксично ідентичних фрагментів коду, які можуть відрізнятися за типами, значеннями даних, іменами регістрів, варіаціями ідентифікаторів, літералів, коментарів), шляхом перетворення коду у послідовності лексем. На рис. 4 представлено ілюстрацію, яка введена в початкове дослідження.

```
if (matching != null)                    if(id0!=null)
   matching.clear();                     id0.id1();
if (n1.isEmpty() || n2.isEmpty())        if(id0.id1() || id2.id1())
   return 0;                             return int;
init(n1, n2, weightProvider);            id0(id1, id2, id3);
prepareInternalArrays();                 id0();
for (int i = 0; i < size1; ++i)          for(id0 id1=int;id1<id2;++id1)
  augmentFrom(i);                        id0(id1);
// . . .                                 // . . .
```

Рис. 4. Етап нормалізації, застосований в дослідженні Хаммела та ін. [4]

Однак для перетворення в токени елементів вихідного коду, таких як ідентифікатори, літерали, тощо, необхідний синтаксичний аналізатор (парсер). При цьому мова всього процесу тоді стає специфічною, через те, що для різних мов програмування потрібні різні синтаксичні аналізатори. Задача пошуку або створення синтаксичного аналізатора ускладнюється і для менш популярних мов програмування. Однак підтримка набору парсерів та використання відповідного, заснованого на основній мові (чи мовах) проекту, не є метою проведеного дослідження.

Спираючись на ці факти, в дослідженні не проводиться токенізація вихідного коду, а відбувається застосування тільки основних кроків попередньої обробки. Ці кроки не будуть впливати на функцію запропонованого МНДП. Тобто можливість виявлення клонів 2 типу буде виключена. В дослідженні вся основна увага буде приділена точним клонам. Тому, підетапами попередньої обробки, які будуть застосовані для аналізу вихідного коду є видалення:

– початкових та кінцевих пробілів;

– подвійних пробілів;

– порожніх рядків.

Відмітимо, що видалятися коментарі не будуть, оскільки стилів коментарів у різних мовах програмування багато і їх видалення буде вимагати додаткової роботи. Синтаксичний аналізатор для видалення коментарів не обов'язковий, але все одно необхідно буде виконати сегментацію системи по компонентах різних мов для автоматизації процесу ідентифікації та видалення коментарів.

В роботі Паскарелла та ін. [5] проводиться вивчення коментування вихідного коду програм з відкритим кодом та промислових програм на базі мови Java. Спираючись на проведені дослідження виявлено, що в таких програмах існує низький відсоток співвідношення коду до коментаря. Так для систем з відкритим кодом показник знаходиться в межах 6,3–12,1 %, в промислових програмах ці дані менші і становлять 0,1–2,5 %. Великих розходжень у висновках з приводу видалення коментарів не очікується. Тому треба притримуватися простих кроків попередньої обробки, які згадувалися вище. Приклад їх використання в реальному вихідному коді представлено на рис. 5.

```
if (matching !=    null)
    matching.clear();

if (n1.isEmpty() || n2.isEmpty())
    return 0;
```

```
if (matching != null)
matching.clear();
if (n1.isEmpty() || n2.isEmpty())
return 0;
```

Рис. 5. Основні етапи попередньої розробки, застосовані в нашому дослідженні

Представлення вихідного коду. Для представленого детектора виявлення повторів та надлишковості програмного коду, основою виступає текст. Мається на увазі, що відсутні будь-які спеціалізовані перетворення, які застосовувалися до необробленого вихідного коду. Однак до складу проміжної інформації, яка зберігається і буде використана повторно підчас кожної редакції коду, прості попередньо

оброблені оператори не входять. Фактичною інформацією, яка збережена в «Індексі клоні», являються хеш-значення разом з метаданими. Отримуються подібні хеш-значення шляхом хешування блоків, складених з фіксованої кількості операторів.

Основою виступає ковзне вікно, де відбувається хешування один за одним послідовних блоків, маючих розмір CHUNK_SIZE, починаючи з діапазону [0, CHUNK_SIZE - 1] до [LINES_COUNT-CHUNK_SIZE, LINES_COUNT - 1]. На рис. 6, представлено приклад даного процесу із заздалегідь визначеним CHUNK_SIZE встановленим в 2. Мінімальна довжина клону неминуче визначається вибором значення для CHUNK_SIZE.

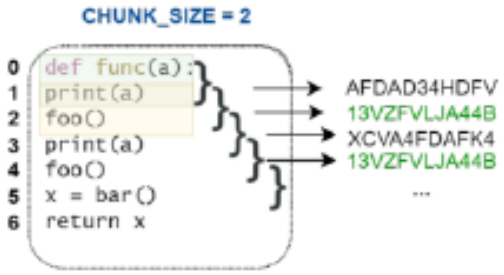


Рис. 6. Ковзне вікно хешування на основі CHUNK_SIZE

Потім в «Індексі клону» зберігається хеш-значення для кожного хешованого блоку. Там же міститься додаткова метаінформація, яка буде використана під час фази фактичного виявлення блоків з повторами і надлишковістю. До її складу входить:

Ім'я файлу, тобто файлу, де розміщено хешований блок.

Оператор індексу.

Рядок початку блоку.

Рядок кінця блоку.

Зазначимо, що «Індекс клону» можна зберігати, як в пам'яті, так і в реальній базі даних. В дослідженні збереження інформації проводиться в пам'яті, хоча в реальних умовах потрібна буде фактична база даних.

Виявлення повторів та надлишковості у програмному коді. При виникненні кожної нової зміни коду або коміті, буде ініціюватися процес виявлення повторів та надлишковості. По-перше детектор має провести обробку кожного файлу, який включено до даного коміту, і на основі попередньо визначеного CHUNK_SIZE продовжує хешування послідовних блоків коду. По-друге відбувається порівняння згенерованих хешів з тими хешами, які збережені в «Індексі клону». На виявлення повторень (клонованих частин коду) буде вказувати будь-яка пара співпадаючих значень хешу, який складається рівно з кількості рядків CHUNK_SIZE. Але цього буває недостатньо, оскільки іноді блок з повторюваним кодом може складатися з більшої кількості рядків, чим визначено CHUNK_SIZE. Відтак, детектору треба скористатися додатковою логікою, при запуску якої відбудеться дослідження, чи є можливість додатково розширити мінімальний ідентифікований клон. Саме в дослідженні Хаммела та ін. [4] аналітично представлені і пояснюються деталі цього процесу. Однак, на високому рівні буде працювати визначення того, чи будуть перекриватися окремі екземпляри клонів. Наприклад, якщо блок з повтореннями коду складається з п'яти рядків, починаючи з індексу 0 до 4, інший блок в цьому ж файлі знаходиться між рядками з індексами 1 і 5, то автоматично це буде означати, що подібні фрагменти перекриваються і окремі клони можуть об'єднуватися в один, який буде займати рядки від 0 до 5. Важливо, що даний процес об'єднання може відбуватися тільки через природу подібного детектора, спроможного спеціально фокусуватися на однакових блоках коду. Невірним виявляється припущення у разі виявлення нечітких клонів на базі оцінок.

Під час процесу виявлення повторів (клонів) та надлишковості, у фрагментах коду відбувається оновлення «Індексу клону» та підготовка його до наступного разу, коли буде запущений інкрементний покроковий процес, при настінні нового коміту. Зокрема, в залежності від типу змін, проводиться обробка пакетами змін файлів, які включені в коміт. При цьому відбуваються наступні дії:

1) проходить обробка пакету, відповідного видаленню. Створюється запит «Індексу клону» з записами індексів файлів, які було видалено. Це дозволить визначити, які з клонів були видалені, та прибрати виявлені з індексу. Подібна обробка видалень дуже важлива в першу чергу через те, що в іншому разі (наприклад, спочатку проходила обробка пакету оновлених фалів) відбулося б порівняння відповідних записів індексу з застарілим «Індексом клону»;

2) іде обробка оновлених файлів, які відповідають перейменуванню. Це аналогічно обробці видалення. Проходить виявлення повторів в коді і знищуються записи, що відповідають старим назвам файлів. Потім знову їх додають, доповнюючи новими записами з відповідними їм оновленими іменами файлів;

3) наступний крок – це оновлені файли, тобто, це розглядається як знищення з подальшим створенням. Як у випадку видалення, спочатку іде запит індексу з неоновленими версіями файлів, для виявлення клонів, які були знищені. Потім проводиться видалення застарілих записів з індексу. Згодом генерується індекс запису для оновлених версій файлів та знову створюється запит індексу, для знаходження повторів коду (клонів), які були додані. В фіналі відбувається оновлення індексу новими записами;

4) у простому випадку новостворених файлів іде генерація відповідних записів індексу, виявляються повтори в коді і в кінці проводиться оновлення індексу з додаванням записів до «Індекс клону».

Вихідні дані. Такими даними для детектора являються необроблені текстові логи (текстові файли, в яких зберігається інформація про відвідування, параметрах відвідувань якогось сайту і помилки, які виникали на цьому), які містять вказівки на виявлені повтори (клони) та надлишковості коду. Більш докладно, у логах включено: (1) ім'я файлу для кожного екземпляра клону, (2) початковий індекс кожного клону у попередньо обробленому файлі, (3) початок та кінець рядків, (4) кількість блоків коду довжиною CHUNK_SIZE, які сприяли остаточному максимальному дублюванню. На рис. 7 представлено цей лістинг, рис. 8–9 демонструє лістинги відповідних файлів кінцевого результату.

```
1 (.../project/Test.java|0|0-6) -
2 (.../project/Test2.java|1|1-7) - 2
```

Рис. 7. Приклад лістингу вихідних даних детектора

```
0 import java.lang.*;
1 import java.io.*;
2 class Test {
3     public static void main(
      String []args) {
4         System.out.println("Hello
       world");
5     }
6 }
```

Рис. 8. Лістинг Test.java

```
0 import java.util.*;
1 import java.lang.*;
2 import java.io.*;
3 class Test {
4     public static void main(
      String []args) {
5         System.out.println("Hello
       world");
6     }
7 }
```

Рис. 9. Лістинг Test2.java

Детектором в даному прикладі було виявлено повтори коду між файлами Test.java та Test2.java. В 1-му екземпляр клонованого блоку знаходиться між рядками 0–6, тоді як у 2-му можна побачити його у рядках 1–7. Важливо звернути увагу на те, що спостерігається відповідність індексів рядкам, після проведення попередньої обробки файлу. Це значить, що наприклад, якщо в файлі Test.java порожній рядок містився під індексом 0 – блок з повторами був переміщений у рядки від 1 до 7 початкового файлу – екземпляр коду з повторами (клон) все одно буде виявлено в рядках від 0 до 6, оскільки під час фази попередньої обробки відбулося видалення порожнього рядка. Детектором буде виведена кількість блоків коду, які посприяли отриманню повтору. В даному прикладі вибраний CHUNK_SIZE був призначений рівним 6. В результаті цифра 2 підтверджує те, що відбулося використання двох блоків коду довжиною 6, які перекривають один одного. Це призвело до виявлення повторів та надлишковості (клону) довжиною 7.

Висновки

Для отримання представлення про продуктивність МНІДП, його запускають для п'яти програмних систем, проводячи вимірювання вимог до часу та пам'яті. Тобто, для кожної системи відбувається аналіз серії з п'ятдесяти комітів та виміри часу і пам'яті, які будуть потрібні для процесу створення індексу. Також проходить вимірювання середнього часу, потрібного для обробки п'ятдесяти комітів під час кроків інкрементного процесу. Для спостереження за поведінкою МНІДП по відношенню до виявлених повторів та надлишковості у програмному коді, постає потреба у проведенні додаткових експериментів. Зокрема, відбувається аналіз десяти останніх комітів для кожної з систем, існуючих в наборі даних і виявлення тих повторів (клонів), які додавалися та знищувалися.

В ході експериментів відбувається порівняння ефективності запропонованого у дослідженні підходу МНІДП та сучасного традиційного підходу SIG для виявлення повторів та надлишковості в програмному коді. Остаточна мета – це перевірка покращення, яке має забезпечувати інкрементний підхід, в тому разі, коли процес виявлення повторюється регулярно для кожного нового перегляду програмного проекту.

Література

1. Nils Göde and Rainer Koschke. Incremental clone detection. In 2009 13th European Conference on Software Maintenance and Reengineering, pages 219–228. IEEE, 2009.

2. Tung Thanh Nguyen, Hoan Anh Nguyen, Jafar M Al-Kofahi, Nam H Pham, and Tien N Nguyen. Scalable and incremental clone detection for evolving software. In 2009 IEEE International Conference on Software Maintenance, pages 491–494. IEEE, 2009.

3. Yoshiki Higo, Ueda Yasushi, Minoru Nishino, and Shinji Kusumoto. Incremental code clone detection: A pdg-based approach. In 2011 18th Working Conference on Reverse Engineering, pages 3–12. IEEE, 2011.

4. Benjamin Hummel, Elmar Juergens, Lars Heinemann, and Michael Conradt. Indexbased code clone detection: incremental, distributed, scalable. In 2010 IEEE International Conference on Software Maintenance, pages 1–9. IEEE, 2010.

5. Luca Pascarella, Magiel Bruntink, and Alberto Bacchelli. Classifying code comments in java software systems. Empirical Software Engineering, 24(3):1499–1537, 2019.

Рецензія/Peer review : 13.05.2021 р. Надрукована/Printed :30.06.2021 р.

ISSN 2219-9365
DOI: 10.31891/2219-9365

Міжнародний науково-технічний журнал

ВИМІРЮВАЛЬНА ТА ОБЧИСЛЮВАЛЬНА ТЕХНІКА В ТЕХНОЛОГІЧНИХ ПРОЦЕСАХ

2021, № 1

International scientific-technical journal

MEASURING AND COMPUTING DEVICES IN TECHNOLOGICAL PROCESSES

2021, Issue 1

Хмельницький 2021
Khmelnytskyi 2021

МІЖНАРОДНИЙ НАУКОВО-ТЕХНІЧНИЙ ЖУРНАЛ
ВИМІРЮВАЛЬНА ТА ОБЧИСЛЮВАЛЬНА ТЕХНІКА В ТЕХНОЛОГІЧНИХ ПРОЦЕСАХ

Затверджений як фахове видання (перереєстрація), група «Б»
Наказ МОН 28.12.2019 №1643

Засновано в травні 1997 р. *Виходить 2 рази на рік*

Хмельницький, 2021, № 1 (67)

Засновник і видавець: Хмельницький національний університет
(до 2005 р. — Технологічний університет Поділля, м. Хмельницький)

Наукова бібліотека України ім. В.І. Вернадського http://nbuv.gov.ua/j-tit/votte

Журнал включено до наукометричних баз:

Index Copernicus	http://jml2012.indexcopernicus.com/p24781565,3.html h-індекс 49,97
Google Schoolar	http://scholar.google.com.ua/citations?user=nwN_nusAAAAJ&hl=uk - індекс 9
Національна бібліотека України ім. В.І. Вернадського	http://nbuv.gov.ua/j-tit/votte

Рекомендовано до друку рішенням Вченої ради Хмельницького національного університету, протокол № 17 від 27.05.2021

Адреса редакції: **Україна, 29016,**
м. Хмельницький, вул. Інститутська, 11,
Хмельницький національний університет
редакція журналу "Вимірювальна та обчислювальна техніка в технологічних процесах"
☎ **067-347-74-57**
e-mail: **mscientificjournal@gmail.com**
web: **http://journals.khnu.km.ua/index.php/MeasComp**

Зареєстровано Міністерством України у справах преси та інформації.
Свідоцтво про державну реєстрацію друкованого засобу масової інформації
Серія КВ № 23279-13119ПР від 24 травня 2018 року (перереєстрація)

Міжнародний науково-технічний журнал
«ВИМІРЮВАЛЬНА ТА ОБЧИСЛЮВАЛЬНА ТЕХНІКА В ТЕХНОЛОГІЧНИХ ПРОЦЕСАХ»
ISSN 2219-9365

ЗМІСТ

Міжнародний науково-технічний журнал
«ВИМІРЮВАЛЬНА ТА ОБЧИСЛЮВАЛЬНА ТЕХНІКА В ТЕХНОЛОПЧНИХ ПРОЦЕСАХ»
ISSN 2219-9365

Міжнародний науково-технічний журнал
«ВИМІРЮВАЛЬНА ТА ОБЧИСЛЮВАЛЬНА ТЕХНІКА В ТЕХНОЛОПЧНИХ ПРОЦЕСАХ»
ISSN 2219-9365

УДК 004.056.5:621
DOI: 10.31891/2219-9365-2021-67-1-8

ПРАВОРСЬКА Н.І., БЕДРАТЮК Л.П.,
ФОРКУН Ю.В., ЯШИНА О.М.
Хмельницький національний університет

МОВНОНЕЗАЛЕЖНИЙ ДЕТЕКТОР ДЛЯ ВИЯВЛЕННЯ І УСУНЕННЯ ПОВТОРІВ ТА НАДЛИШКОВОСТЕЙ ПРОГРАМНОГО КОДУ

Під час розробки програмного забезпечення існує ймовірність того, що в програмному коді можуть траплятися помилки, які допускають навіть фахівці-розробники, припускаючись дублюванню частин коду. Для усунення майбутніх збоїв в режимі функціонування програмного продукту, існує ряд автоматизованих інструментів, спроможних проводити оцінювання ремонтопридатності на основі ряду заздалегідь визначених критеріїв, таких як обсяг і складність коду, зв'язок модулів, тощо. Автоматичне виявленні блоків з повторами та надлишковостями в програмному коді сучасних проектів стає основою для майбутнього ручного або автоматичного рефакторінгу, який призводить до більш чистого та зручного у супроводі коду. Одним з таких інструментів виступає запропонований мовнонезалежний детектор, який використовує інкрементний підхід та його покращення з використанням локально-чутливого хешування.

Ключові слова: програмний код, мовно незалежний детектор, інкрементний підхід, локально-чутливе хешування

PRAVORSKA N., BEDRATYUK L.,
FORKUN YU., YASHYNA O.
Khmelnitsky national university, Ukraine

LANGUAGE-INDEPENDENT DETECTOR FOR DETECTING AND ELIMINATING REPETITIONS AND EXCESSES OF SOFTWARE CODE

When developing software (software) there is a possibility that mistakes made even by developers, in the future will lead to violations of the normal operation of the software product. Corrections can usually be made at any stage of the software life cycle. However, it should be borne in mind that the detection and correction of errors in the program code in the final stages of development can have a very significant impact on the costs (both financial and time) of software development and maintenance. In addition, some errors are dangerous to human life and health if they appear during operation. Therefore, in the development of software products in their life cycle, a variety of tools have become widely used, which analyze the software code and help identify defects. A number of different problems in the source code of the system arise precisely because of the presence of a significant share of code duplication. The increase in the size of the code base, and accordingly, the increase in maintenance costs, in particular, is a consequence of duplication. In addition, when an error occurs in one of the instances of the block with duplicates (clones) and redundancies, every other block is subject to verification for the same error and the possibility of potential correction. To solve the last problem, you need not only to know the lists of duplicate blocks of code, but also to have a significant amount of time required to go through all the instances. Finally, duplication causes problems in terms of understanding the program code and complications of future refactoring. It is important that the foundation for future manual or automatic refactoring, which leads to a cleaner and easier to maintain code, is the automatic detection of clones (blocks with repetitions and redundancies) in modern software projects. In this regard, the various methods of detecting blocks with repetitions proposed today, mainly work with the entire code base of the system. Regardless of the magnitude of the changes, similar methods for each version of the source code use the entire system as input. This approach may work well for stable outdated systems that are occasionally updated. However, at the current stage of IT development, this is not an ideal option due to flexible software development. In the process of detecting blocks with repetitions and redundancies for the next check, unprofitable calculations will be performed, which are added to the total execution time.

During the software development, there is a probability that in the program code there may be errors that allow even developers specialists, assuming duplicate parts of the code. In order to eliminate future failures in the functioning of the software, there are a number of automated tools that are capable of evaluating repairability based on a number of predefined criteria, such as the scope and complexity of the code, communication of modules, etc. Automatic detection of repetitions and excesses in the software code of modern projects becomes the basis for future manual or automatic refactoring, which leads to a cleaner and convenient code accompaniment. One of these instruments is the proposed linguistic detector that uses an incremental approach and improving it using locally-sensitive hashing.

Keywords: Program code, Language independent detector, incremental approach, locally-sensitive hashing

Вступ. При розробці програмного забезпечення (ПЗ) існує ймовірність того, що допущені помилки навіть фахівцями-розробниками, в майбутньому призведуть до порушень нормального режиму функціонування програмного продукту. Виправлення, звичайно можна провести на будь-якому з етапів життєвого циклу ПЗ. Однак, треба враховувати, що виявлення та виправлення помилок в програмному коді на останніх етапах розробки можуть дуже суттєво вплинути на витрати (як фінансові, так і часові) розробки та супроводу ПЗ. До того ж деякі помилки становлять небезпеку життю і здоров'ю людей, якщо вони стануть проявлятися на етапі експлуатації. Тому при розробці програмних продуктів в їх життєвому циклі,

Міжнародний науково-технічний журнал
«ВИМІРЮВАЛЬНА ТА ОБЧИСЛЮВАЛЬНА ТЕХНІКА В ТЕХНОЛОГІЧНИХ ПРОЦЕСАХ»
ISSN 2219-9365

широкого використання набули різноманітні інструменти, які проводять аналіз коду програмного забезпечення і сприяють виявленню дефектів. Ряд різноманітних проблем в вихідному коді системи, виникає саме через наявність значної долі дублювання коду. Збільшення розміру кодової бази, і відповідно, збільшення затрат на обслуговування, зокрема, є наслідком дублювання [1]. Крім цього, коли виникає помилка в одному з екземплярів блоку з повторами (клонами) та надлишковостями, перевірці підлягає кожен інший блок на наявність тієї ж помилки та можливості потенційного виправлення [2]. Для вирішення останньої задачі, треба не лише знати списки дубльованих блоків коду, а й мати значну кількість часу, потрібного для проходження по всім екземплярам. Нарешті, через дублювання виникають проблеми з точки зору розуміння програмного коду і ускладнень майбутнього рефакторінгу [1]. Важливо, що закладення основ для майбутнього ручного або автоматичного рефакторінгу, який призводить до більш чистого та зручного у супроводі коду, полягає в автоматичному виявленні клонів (блоків з повторами та надлишковостями) у сучасних програмних проектах. В цьому відношенні, різноманітні методи виявлення блоків з повторами, запропоновані на сьогодні, в основному працюють зі всією кодовою базою системи. Незалежно від величини внесених змін, подібні методи для кожної версії вихідного коду, використовують в якості вхідних даних всю систему. Такий підхід може добре працювати для стабільних застарілих систем, які зрідка оновлюються. Однак, на сучасному етапі розвитку IT, це не являється ідеальним варіантом, через гнучку розробку програмного забезпечення. В процесі виявлення блоків з повторами та надлишковостями для наступної перевірки, будуть виконуватися збиткові обчислення, які додаються до загального часу виконання.

Основи розробки мовно-незалежного інкрементного детектору повторів (МНІДП).Потреба в інкрементних підходах виникла через вказані вище недоліки, поряд з еволюцією практик розробки програмного забезпечення та появи таких концепцій, як безперервна інтеграція/розробка (CI/CD – continuous integration/development), гнучкість та швидкість. В цьому контексті основною ідеєю виступає повторне використання інформації, яка отримується в результаті аналізу одного перегляду, для наступного, виключаючи непотрібні неефективні за часом операцій. Деякими компаніями (наприклад, SIG – група вдосконалення програмного забезпечення) розробляються автоматизовані інструменти, спроможні проводити оцінювання ремонтопридатності на основі ряду заздалегідь визначених критеріїв, таких як обсяг і складність коду, зв'язок модулів, тощо. Для виявлення подібних критеріїв використовуються детектори повторів та надлишковостей (клонованих частин коду). На основі отриманих результатів визначається доля дубльованого коду в кодовій базі проекту. Саме компанія SIG була розробником подібного детектору повторів із реальною застосовністю слів. Якщо брати за основу контекст SIG, то пропонується розробка незалежного від мови програмування інкрементного детектору повторів (клонів) та оцінка його продуктивності. В роботі розроблена методика носить назву мовно-незалежний інкрементний детектор повторів (МНІДП). При цьому підході використовувався алгоритм, який базується на основі індексу, запропонований Бенджаміном Хаммелом та ін [3]. Основною структурою даних, яка використовується виступає індекс клону. Він представляє собою глобальну структуру даних, яка нагадує типовий інвертований індекс. Цей метод, хоча і використовує лексичний аналізатор для перетворення вихідного коду в токени, але серед всієї літератури з цього питання є найближчим до незалежного від мови інкрементного підходу. По суті, проводиться поділ інктементного методу даного дослідження на два основних робочих процеси. В свою чергу кожний буде включати в себе додатково ряд підпроцесів. Проміжне представлення пов'язується з першим процесом і збережене для повторного використання в версіях проекту. Таке об'єднання (сектор) буде вважатися за індекс повторення (клонування). Весь проект програмного забезпечення використовується цим процесом в якості вхідних даних. Запуск його відбувається одноразово на початку всього конвеєра виявлення повторів (клонованих блоків коду). Другий процес, який посилається на логіку, проводить запуск фактичної процедури виявлення повторів та надлишковості коду та виводить виявлені клони. Процес буде запущено після оновлення основної бази з кодами. В реальному налаштуванні це відбувається після того, як новий запис (commit) буде розміщено в репозиторій керування версіями.

Результати досліджень з використанням МНІДП та його вдосконаленн, шляхом застосування локально-чутливого хешування (ЛЧХ). В якості вихідних даних для проведення досліджень було використано програмні проекти різного розміру, під час оцінювання та порівняння запропонованого підходу з традиційним підходом виявлення клонів, застосовуваний SIG. Такими проектами виступили:

Rippled – система додавання автоматизації PR для проектних плат, написана на мовах C / C++;

Kooboo – безкоштовна система керування контентом, написана на мовах C#, JS, HTML, CSS;

Tensorflow – відкрита програмна бібліотека для машинного навчання, розроблена компанією Google для вирішення завдань побудови і тренування нейронної мережі з метою автоматичного знаходження та класифікації образів, досягаючи якості людського сприйняття, написана на мовах C++, Python;

Openjdk-jdk14u – це реалізація платформи Oracle Java Standard Edition із відкритим кодом. OpenJDK корисний для розробки програм Java і забезпечує повне середовище виконання для запуску програм Java, написана на мові Java;

Міжнародний науково-технічний журнал
«ВИМІРЮВАЛЬНА ТА ОБЧИСЛЮВАЛЬНА ТЕХНІКА В ТЕХНОЛОПЧНИХ ПРОЦЕСАХ»
ISSN 2219-9365

Linux Kernel – ядро операційної системи, що відповідає стандарту POSIX. Складає основу операційних систем сімейства Linux, написана на мові C.

Конкретніше, було використано інструмент SAT SIG при проведенні аналізу набору даних для запропонованого дослідження, який складається з п'яти проектів з відкритим кодом. Зважаючи на те, що SAT є складним інструментом, до якого входить множина базових операцій, що не мають відношення до виявлення дубльованого коду, було прийнято рішення ізолювати відповідні частини та провести заміри частки загального часу, який пройшов. Ця частка напряму пов'язана з виявленням блоків з повторами та надлишковостями в програмному коді.

Результатом експериментів з використанням МНІДП стало покращення часу, необхідного для виявлення повторюваних фрагментів коду, згідно параметрів SAT, що можна побачити з таблиці 1.

Таблиця 1.

Параметри SAT та загальний час виявлення МНІДП

Проект	Загальний час аналізу SAT	Час виявлення клонів	Створення індексу МНІДП та час інкрементного кроку
Rippled	4 хв.	5.63 сек.	4.6 сек.
Kooboo	22 хв.	397 сек.	16.31 сек.
Tensorflow	8 год. 3о хв.	177.1 сек.	91.29 сек.
Openjdk-jdk14u	N/A	N/A	56.34 сек.
Linux Kernel	N/A	N/A	>321.21 сек.

Стосовно виявлення в програмному коді блоків з повторами та надлишковостями (інакше, клонів), проводиться запуск МНІДП для чотирьох програмних систем, для яких наша машина спроможна витримати навантаження інкрементного кроку (через розміри ядра системи Linux, її не опрацьовували). Більш конкретно, для спрощення було вирішено піддавати аналізу десять самих останніх комітів (тобто операцій, які відправляють останні зміни вихідного коду в репозиторії, що робить ці зміни частиною основної ревізії репозиторіїв) для кожної системи. Потім відбувався підрахунок кількість доданих та знищених клонів, які будуть виявлені запропонованим інструментом. Загальний огляд кількості файлів, на які впливає кожний коміт представлені в таблиці 2.

Таблиця 2.

Файли, які зазнали впливу на програмну систему для кожного аналізованого коміту

№ п/п коміту	# Оновлені файли в коміті			
	Rippled	Kooboo	Tensorflow	OpenJDK-14
1	1	6	1	3
2	2	3	1	1
3	0 (пропущений)	1	2	4
4	3	1	0 (пропущений)	1
5	1	1	0 (пропущений)	0 (пропущений)
6	2	1	1	1
7	1	2	1	2
8	3	3	1	5
9	2	1	2	0 (пропущений)
10	1	3	1	0 (пропущений)

Хоча типи змін, які вносяться в проаналізовані файли (додавання, оновлення, перейменування, видалення) не були вказані, більшість з них відповідають модифікаціям існуючих файлів. Записи в таблиці, відмічені як «0 (пропущений)», відповідають комітам, які не піддавалися обробці, з причини включення в них тільки файлів, які за замовчуванням ігноруються МНІДП (наприклад, текстові файли). Це призводить до того, що блоки з повторами та надлишковостями, відповідні зміненим файлам, будуть виявлені як знищені під час кроку видалення та як заново додані під час підкроку додавання.

На рис. 1-4 представлені блоки з повторами та надлишковостями, які були додані та видалені для кожного коміту кожної програмної системи. До речі, МНІДП проводить розгляд модифікованих файлів, як видалення (інакше знищення), за яким іде створення.

Зауважимо, що через величину ядра системи Linux, тут так само, запропонована установка не змогла відпрацювати із навантаженням пам'яті цього процесу і візуальних даних не представлено.

В багатьох випадках кількість видалених клонів відповідає кількості доданих, що зображено на рис. 1-4. Інформація, яка дозволяє зробити висновок чи були будь-які клони додані або видалені, представлена різницею між двома цифрами. Іншим спостереженням є те, що доля блоків з повторами в кодовій базі відповідної системи не залежить від більшості комітів. Фактично, лише під впливом одного коміту змінилася загальна кількість клонів в системах Rippled і OpenJDK, тоді як у випадку Tensorflow та Kooboo, два та чотири коміти видалили/додали клони, відповідно. Додатково були досліджені вихідні дані МНІДП для комітів, стосовно кількості виявлених клонів, для яких виявлена кількість блоків з повторами та

надлишковостями здається більшою (наприклад, коміт 8 для OpenJDK). У подібних випадках, більшість зареєстрованих клонів, які зазвичай включені до багатьох файлів системної кодової бази, відповідають блокам коду/коментарям. Таким прикладом може бути запис в перших рядках деяких вихідних файлів фіксованої інформації, яка стосується ліцензій.

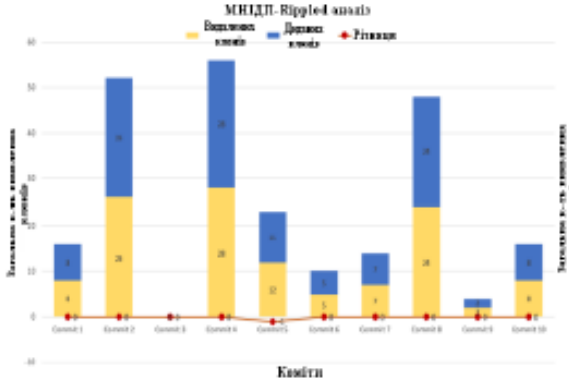


Рис. 1. Виявлення клону для Rippled

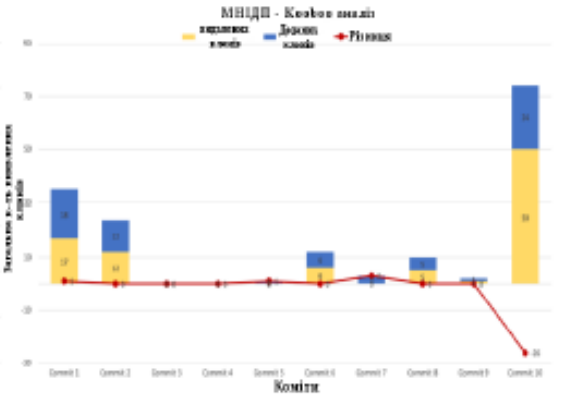


Рис. 2. Виявлення клонів для Kooboo

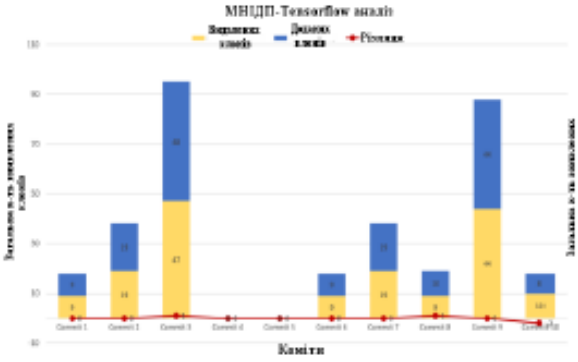


Рис. 3. Виявлення клонів для Tensorflow

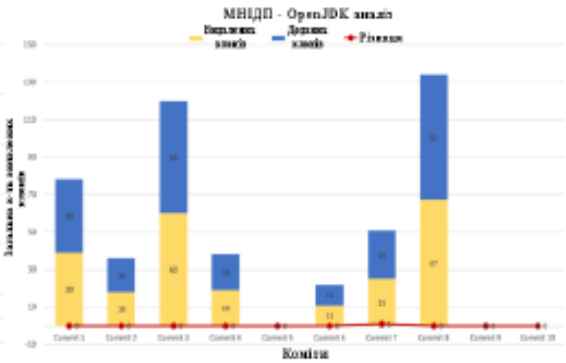


Рис. 4. Виявлення клону для OpenJDK

Доцільним став розгляд подальшого розширення та покращення детектора МНДП, застосовуючи метод оцінки подібності найближчого сусіда – локально-чутливе хешування (ЛЧХ) [4]. Алгоритм пошуку найближчих сусідів на сьогодні є найбільш популярним ймовірністним методом, призначеним для пониження розмірності багатовимірних даних. Тобто пошук, наприклад, подібних документів виявляється досить простим. Базуючись на метриці подібності проводиться порівняння кожного документу з будь-яким іншим документом. При тому, що такий грубий підхід добре працює при невеликих об'єднаннях документів, але виникають проблеми з поганим масштабуванням через вимоги до часу. По мірі того, як збільшується кількість документів такі вимогу зростають квадратично [5]. Значне скорочення обчислювального часу, потрібного для процесу пошуку відбувається з використанням наближених схем, що базуються на основі локально-чутливого хешування. Основна ідея ЛЧХ полягає в використанні різних хеш-функцій для проведення хешування декілька раз базових точок даних. При цьому буде гарантуватися, що подібні елементи мають більший шанс зустрітися та опинитися в одному хеш-сегменті, на відміну від різнорідних елементів [4]. Тільки тоді перевірку на подібність проходять елементи, які і потрапили в хеш-сегмент, вони також відомі як пари кандидатів.

Існує велика кількість різноманітних схем локально-чутливого хешування, придатних для використання. Конкретна схема ЛЧХ представлена на рис 5.

Міжнародний науково-технічний журнал
«ВИМІРЮВАЛЬНА ТА ОБЧИСЛЮВАЛЬНА ТЕХНІКА В ТЕХНОЛОПЧНИХ ПРОЦЕСАХ»
ISSN 2219-9365

Рис. 5. Підоперації локально-чутливого хешування

Складається дана схема ЛЧХ з трьох основних підоперацій:

Черепиця (Shingling). Відбувається перетворення документа в набір k-черепиць. Він може бути будь-яким від простих підрядків довжини k до комбінації з k слів. На ймовірність виявлення збігів буде впливати вибір того, що приймається за черепицю. Якщо черепицею виступає 1 символ, то виявлення збігів в іншому документі буде значно вище, ніж знайти збіг з черепицею більшої довжини.

MinHashing. Після того, як на попередньому кроці були отримані набори, вони можуть виявитися досить об'ємними, тому порівняння стане неефективним. Для часткового вирішення проблеми завдяки MinHashing відбувається перетворення черепиць в менші представлення фіксованої довжини – сигнатуру. Потім коефіцієнт Жаккара [6] використовується не до набору черепиці, а до елементів згенерованого підпису. Збереження інформації набору в максимально можливому ступені, забезпечується згенерованими таким чином підписами. Однак при застосування подібного кроку розрахунок точної подібності між двома файлами стає неможливим, оскільки відбувається втрата інформації. Подібність в цьому разі буде розраховуватися базуючись на оцінці, яка може призвести до точних результатів. З урахуванням всіх обставин, для генерування підпису, який буде являти собою документ, проводиться хешування кожного набору за допомогою k хеш-функцій. Для кожної з цих хеш-функцій обирається мінімальне значення хеш-функції. Для прикладу, отримується підпис із півсотнею значень MinHash на основі 50 випадкових хеш-функцій. Звідси виходить, що більша ймовірність зіткнень і більший рівень помилок буде наслідком використання меншої кількості застосованих хеш-функцій.

ЛЧХ. За допомогою MinHashing прибирається «бич розмірності», який супроводжує набори черепиці. Весь процес стає неефективний через необхідність порівняння кожного підпису з іншим будь-яким підписом. Ось тут знаходить своє застосування в останньому кроці конвеєра ЛЧХ. Метод, який використовується на цьому кроці носить назву бандінгу або окільцьовування. В матриця розміщується k хеш-значень для кожного документу, потім відбувається розбиття матриці на b смуг, які складаються з r рядків. Тобто при використання дванадцяти хеш-функцій розбиття проводиться на чотири смути по три рядки. Потім при появі нового документу $D_{нов}$ потребується визначити, чи відбувається утворення пари кандидатів з іншим документом. Тоді іде перегляд кожної смути та виявлення документів, маючих однакові значення MinHash в кожному рядку цієї смуги. При знаходженні смуги, в якій всі рядки збігаються, буде продовжене повне порівняння між цими документами. Вирішальним являється вибір кількості смут та рядків, оскільки він становить поріг подібності, над яким вважається, що два документи однакові.

Однак в ході дослідження виявилося, що підхід, який спирався на ЛЧХ не відповідає характеристикам вихідного підходу, хоча присутня можливість для майбутніх досліджень для вивчення можливих покращень. Увага приділяється не тільки розробці мовно-незалежного інкрементного детектора повторів і оцінці його ефективності та продуктивності, а також для забезпечення можливості подальшого обслуговування програмної системи, ідентифікації та видалення блоків з кодом, який дублюється, для сприяння виявленню клонів. Усування недоліків, які виникають через існування блоків з повторами та надлишковостями стає можливим завдяки рефакторінгу програмного коду. Проводиться подібний рефакторінг вручну або автоматично, після чого кодова база набуває більшої перспективності.

При зосередженні уваги на конкретній мові програмування (особливо на такій популярній як Java [7]) виникає змога для проведення глибокого та детального виявлення повторів та надлишковості в програмному коді. Більш складні клоновані частини коду є можливість знайти на прикладі семантично схожих блоків коду з повторами. При цьому проводиться застосування різних методів, спроможних виявити клони блоків програмного коду, таких як: на основі токенів, на основі дерев, на основі графів і т.п. Пошук чи створення аналізатора для менш популярних мов стає дуже складною задачею, так як SIG не має обмеження стосовно мов програмування, які підтримуються. Тому дослідження було сфокусовано на мовно-незалежних детекторах, створення яких можливе тільки з використанням текстових методів.

Методи інкрементного виявлення блоків з повторами та надлишковостями в програмному коді, з'явилися через необхідність вирішувати питання ремонтопридатності, пов'язані з дублюванням

Міжнародний науково-технічний журнал
«ВИМІРЮВАЛЬНА ТА ОБЧИСЛЮВАЛЬНА ТЕХНІКА В ТЕХНОЛОГІЧНИХ ПРОЦЕСАХ»
ISSN 2219-9365

програмного коду, у поєднанні з бажанням проводити багаторазовий запуск процесу виявлення фрагментів з клонами. Для більшості з існуючих подібних методів виникає потреба в мовному синтаксичному аналізаторі. А саме компоненті, який автоматично виключає можливість мовної незалежності для відповідних запропонованих детекторів клонів. Отже, мови програмування, не можуть бути проаналізовані в контексті виявлення блоків з повторами та надлишковостями, оскільки для них пошук або створення синтаксичного аналізатора є складною задачею.

Однією з характеристик запропонованих інктементних детекторів являється те, що вони не видають повний снепшот усіх блоків з повторами та надлишковостями (клонів) в кодовій базі, а дають лише клони, які при кожній перевірці коду додавалися або видалялися. Для вирішення проблеми існують наступні варіанти:

1. Введення логіки керування клонами. Була б можливість агрегації клонів від початку кодової бази (або перевірки, коли всі блоки з повторами були відомі) до потрібної перевірки, переглядаючи ті, які були додані або видалені.

2. Введення параметру, який використовуючи кожний окремий файл в кодовій базі, дозволив би проводити запит індексу. З урахування того, що всі файли системи мають пройти через конвеєр виявлення, який згадувався в даному дослідженні, то на подібне рішення буде витрачено дуже багато часу.

Два запропонованих інкрементних детектори повторів та надлишковостей не порівнювалися з існуючими сучасними інкрементними методами. При використанні запропонованого МНІДП проводилося оцінювання продуктивності та вивчення її у порівнянні з сучасним підходом SIG для виявлення клонів. Також для покращення продуктивності вивчалася можливість підходу заснованого на ЛЧХ.

Метою даного дослідження було визначення способу створення мовно-незалежного детектора повторів на надлишковостей програмного коду та дослідження, яку інформацію має бути збережено, для того, щоб детектор працював інкрементно (покроково). Для цього був модифікований вихідний алгоритм Хаммела та дослідили використання ЛЧХ у спробі розширити та покращити вихідного підходу. В процесі роботи виявлено, що для досягнення мовної незалежності, при цьому задовольняючи вимоги, встановлені на початку цього дослідження, можна використовувати проміжне представлення модифікованого «Індексу клону».

Висновки. В ході експериментів також встановлено, що підхід МНІДП виявився набагато швидший, ніж очікувалося, при цьому залишаючи обмежені можливості для подальшого покращення. Останнє пройшло перевірку в ході спостереження за продуктивністю реалізації, заснованої на ЛЧХ, яка працювала гірше в обумовленій формі. Враховуючі всі аспекти, в контексті даного дослідження, було побудовано мовно-незалежний інкрементний детектор повторів, заснований на підході Хаммела та ін. [3] та розширивши за допомогою ЛЧХ. Але в контексті вихідного алгоритму потрібні додаткові дослідження для вивчення потенціалу підходу на основі ЛЧХ.

Крім цього, іншою метою даного дослідження було з'ясування адекватності роботи такого поетапного підходу у порівнянні з традиційним детектором комерційного рівня, наприклад, вбудованим в інструмент SAT SIG. Для цього було обрано набір даних з п'яти систем і проведено запуск SAT для кожної з них, з наступним вимірюванням часу, необхідного для виявлення блоків з повторами та надлишковостями. Отримати результати аналізу SAT для самих великих та складних систем нашого інформаційного фонду не вдалося, через те, що інструмент не зміг їх обробити. Однак, для менших систем все ж таки дані аналізу були отримані. У порівнянні з вимірюваннями інкрементного детектора, коли процес виявлення повторюється для серії комітів, результати вказують на значне покращення накопиченого часу виявлення клонів. Тобто, якщо б компанія SIG використала запропонований підхід, то це вплинуло на економію часу та ресурсів. Підгрунтям для можливої інтеграції запропонованих підходів в рамках моделі ремонтопридатності (DMM – Delta Maintainability Model) виступає те, що вони мають здатність виявляти блоки з повторами та надлишковостями поза контекстом коміту [8].

Література

1. Bellon S., Koschke R., Antoniol G., Krinke J., Merlo E., Comparison and evaluation of clone detection tools. IEEE Transactions on software engineering, 33(9):577–591, 2007.

2. Li Z., S., S Myagmar S., Zhou Y., Cp-miner: Finding copy-paste and related bugs in large-scale software code. IEEE Transactions on software Engineering, 32(3):176–192, 2006.

3. Hummel B., Juergens E., Heinemann L., Conradt M., Indexbased code clone detection: incremental, distributed, scalable. In 2010 IEEE International Conference on Software Maintenance, pages 1–9. IEEE, 2010.

4. Indyk P., Motwani R., Approximate nearest neighbors: towards removing the curse of dimensionality. In Proceedings of the thirtieth annual ACM symposium on Theory of computing, pages 604–613, 1998.

5. Beyer K., Goldstein J., Ramakrishnan R., Shaft U., When is "nearest neighbor" meaningful? In International conference on database theory, pages 217–235. Springer, 1999.

6. Broder A., On the resemblance and containment of documents. In Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171), pages 21–29. IEEE, 1997.

7. Meyerovich L., Rabkin A., Empirical analysis of programming language adoption. In Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications, pages 1–18, 2013.

di Biase M., Rastogi A., Bruntink M., van Deursen A., The delta maintainability model: measuring maintainability of fine-grained code changes. In 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), pages 113–122. IEEE, 2019.

ДОДАТОК Г
(Обов'язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРИАЛИ

Кафедра інженерії програмного забезпечення

Удосконалення методів виявлення повторів та надлишковості програмного коду

Виконала: *студент гр. ІПЗм-20-2* *Праворська Н. І.*

Керівник:, *д. ф.-мат. н., професор* *Бедратюк Л. П.*

Об'єкт, предмет, мета дослідження

Об'єкт дослідження. Вихідні програмні коди, в яких можуть бути повтори та надлишковості.

Предмет дослідження. Методи виявлення повторів й надлишковості програмного коду та їх розширення.

Мета роботи. Удосконалення методів виявлення повторів й надлишковості програмного коду

Задача роботи. Розробка мовно-незалежного інкрементного детектору повторів (МНІДП) для виявлення в програмному коді блоків з повторами та надлишковістю.

Актуальність проблеми

Актуальність теми роботи полягає в необхідності розробки інкрементного методу незалежного від мови програмування з виявлення повторів та надлишковості програмного коду , який сприятиме скороченню кількості непотрібних операцій та часу, необхідних для цього.

В минулому вже запропоновано ряд різних інкрементних підходів, переважна більшість з них мовно-залежні, тобто потребують синтаксичного аналізатора для обробки базового коду.

Завдання дослідження

Основними завданнями роботи виступають:

1. Дослідження мовної незалежності в контексті інкрементного виявлення блоків програмного коду з повторами та надлишковістю і розробка МНІДП. Основою даного інструменту виступає існуюча методика інкрементного текстового виявлення клонів, розроблена Хаммелом та ін.
2. Дослідження придатності ЛЧХ в якості методу, який можна використати з метою розширення та покращення первісно запропонованого підходу.
3. Розробка інкрементного детектора повторів, який буде використовувати метод ЛЧХ.
4. Оцінювання продуктивності МНІДП та порівнянні її з підходом комерційного рівня SIG по виявленню клонів, для вивчення переваг, які може запропонувати інкрементний підхід.
5. Оцінювання ЛЧХ з точки зору ефективності, порівнюючи його з МНІДП та надаючи результати стосовно продуктивності цього розширення.

Наукова новизна

Удосконалення методів виявлення повторів та надлишковості програмного коду, використовуючи інструмент аналізу незалежний від мови програмування за рахунок текстового інкрементального підходу.

Практичне значення

Практична цінність отриманих результатів полягає в успішній розробці підходів, методів та алгоритмів пошуку та виявлення блоків з повторами та надлишковостями в програмному коді незалежно від мови програмування

Завдяки поліпшення характеристик методу у порівнянні з існуючими рішеннями, спроектовано та розроблено МНІДП, спроможного виявляти блоки клонів з точною відповідністю. Визначено придатність підходу на основі ЛЧХ для розширення розробленого МНІДП.

Аналіз стану проблеми і інших рішень

Основні проблеми у галузі:

- втрата часу через використання традиційними методами всієї системи в якості вхідних даних, коли з плином часу аналіз доводиться часто повторювати;
- виникнення надмірних обчислень при пошуку повторів та надлишковостей у програмному коді
- обмеження мовою програмування

Мовно-незалежний інкрементний детектор

В основі роботи МНІДП лежить алгоритм, який базується на основі індексу, запропонований в роботі Хаммела - основною структурою даних, яка використовується виступає індекс клону.

Він представляє собою глобальну структуру даних, яка нагадує типовий інвертований індекс.

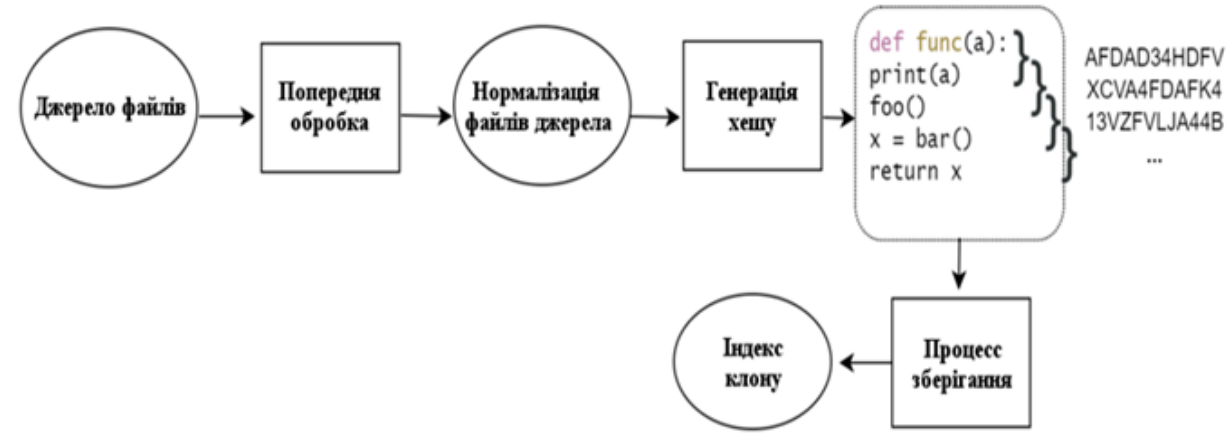
Цей метод, хоча і використовує лексичний аналізатор для перетворення вихідного коду в токени, але серед всієї літератури з цього питання є найближчим до незалежного від мови інкрементного підходу.

Основні робочі процеси методу

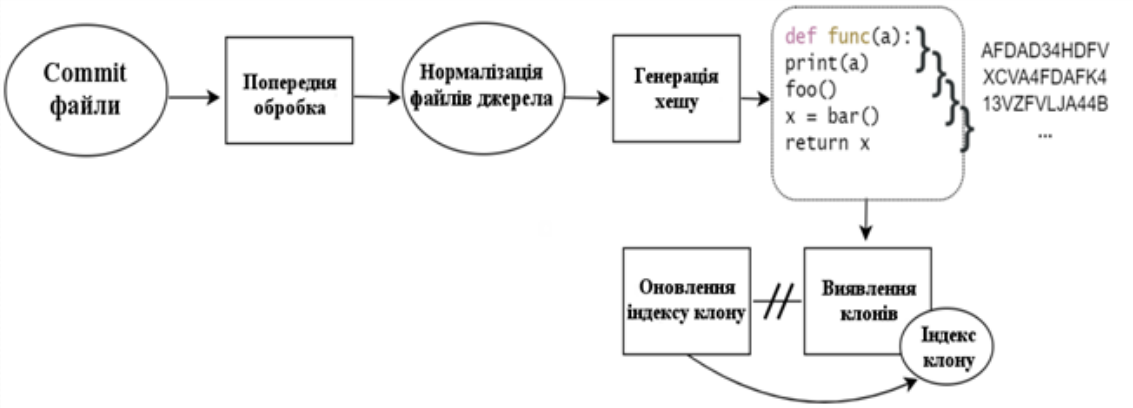
Проводиться поділ інктементного методу даного дослідження на два основних робочих процеси. В свою чергу кожний буде включати в себе додатково ряд підпроцесів.

Проміжне представлення пов'язується з першим процесом і зберігається для повторного використання в версіях проекту

Підкроки робочого процесу створення «Індексу клону»



Підетапи інкрементного робочого процесу



Дублювання буде виявлене, якщо два хеши збігаються

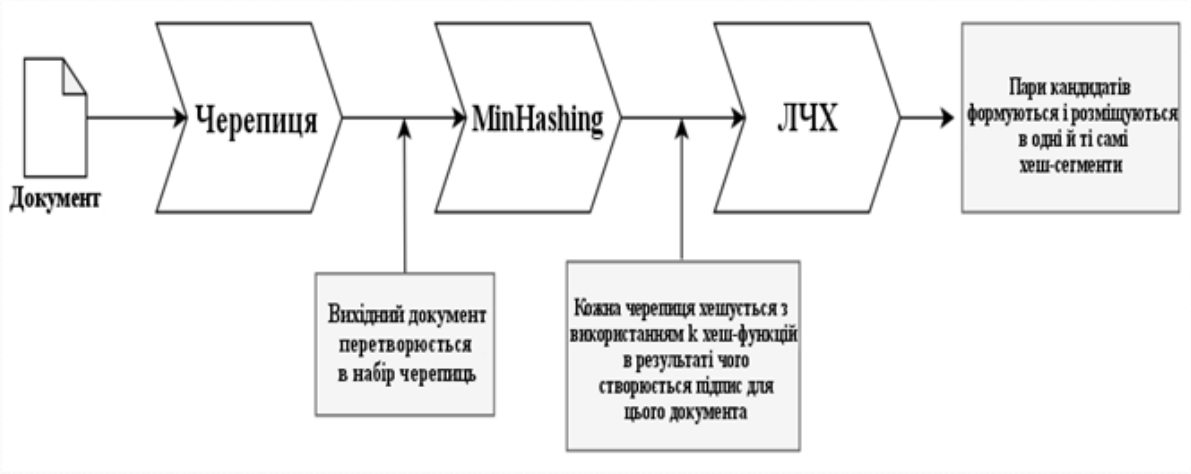
Локально-чутливе хешування

Алгоритм пошуку найближчих сусідів на сьогодні є найбільш популярним ймовірністним методом, призначеним для пониження розмірності багатовимірних даних.

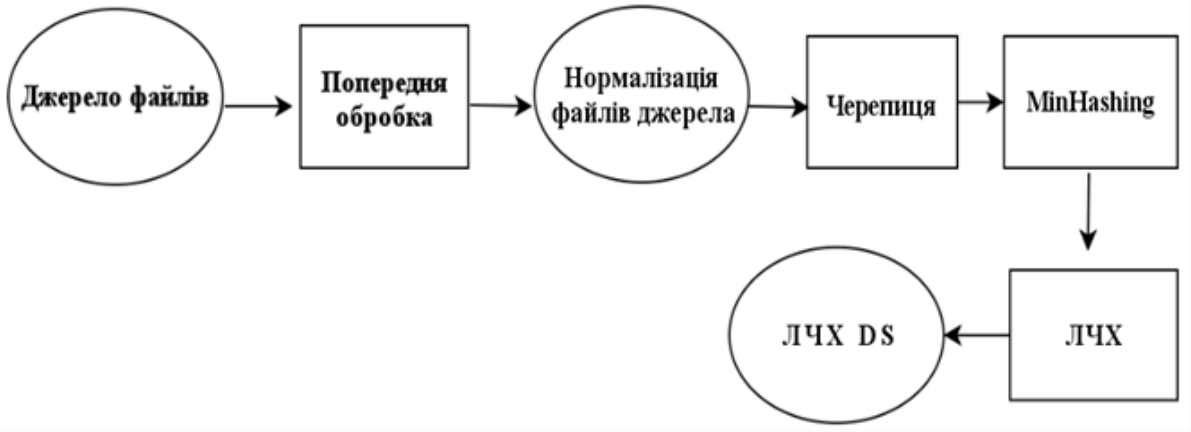
Значне скорочення обчислювального часу, потрібного для процесу пошуку відбувається з використанням наближених схем, що базуються на основі ЛЧХ.

Основна ідея ЛЧХ полягає в використанні різних хеш-функцій для проведення хешування базових точок даних декілька раз. При цьому буде гарантуватися, що подібні елементи мають більший шанс зустрітися та опинитися в одному хеш-сегменті, на відміну від різнорідних елементів

Підоперації локально-чутливого хешування



Підкроки робочого процесу створення індексу в МНІДП на основі ЛЧХ



Таблиця 1. Початковий інформаційного фонду проектів з відкритим кодом

	Мова програмування	Кількість файлів	Кількість LOCs
Linux Kernel	C	57.205	23.229.768
OpenJDK-14	Java	60.444	12.045.316
Tensorflow	C++, Python	12.387	3.194.893
Kooboo	C#, JS, HTML, CSS	4.109	670.265
Ripple	C / C++	1.399	312.011

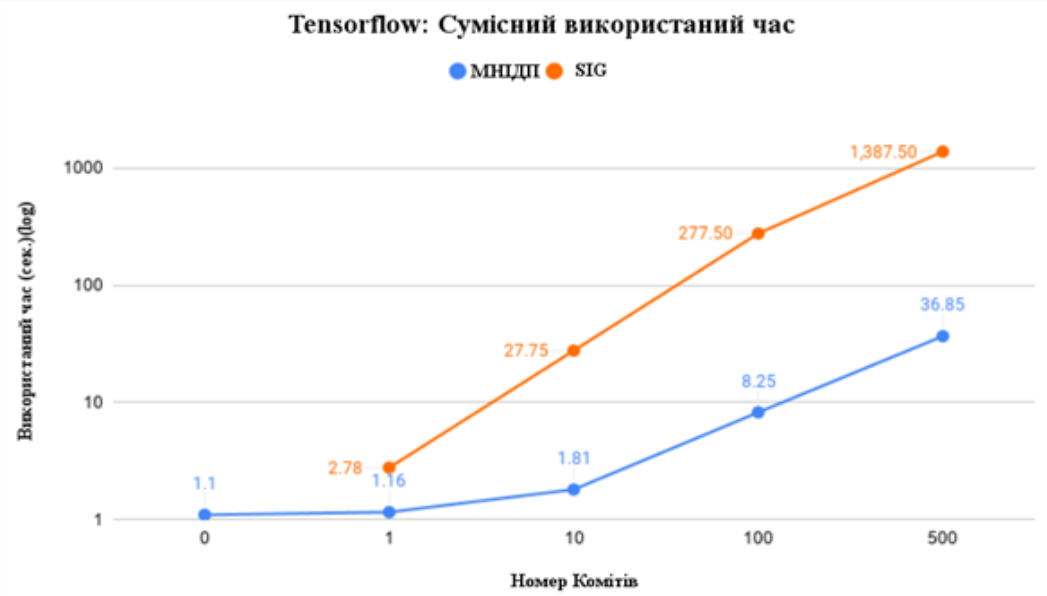
Таблиця 2. Виміри МНІДП, які використані для експериментів

Проект	Кількість рядків (LOCs) прочитаних в комітах	Оброблено	Час створення індексу (сек)	Середній час інктемент-ного кроку (сек)	Стандартне відхиленні інкремент-ного кроку (сек)
Rippled	208.100	42	3.75	0.85	1.31
Kooboo	681.143	50	16.28	0.03	0.04
Tensorflow	3,814.652	45	65.89	4.29	3.32
OpenJDK-14	3,377.211	46	48.53	4.72	5.07
Linux Kernel	23.603.823	45	321.21	N/A	N/A

Таблиця 3. Параметри SAT SIG та загальний час виявлення МНІДП

Проект	Загальний час аналізу SAT	Час виявлення клонів	Створення індексу МНІДП та час інкрементного кроку
Rippled	4 хв.	5.63 сек.	4.6 сек.
Kooboo	22 хв.	397 сек.	16.31 сек.
Tensorflow	8 год. 30 хв.	177.1 сек.	91.29 сек.
OpenJDK-14	N/A	N/A	56.34 сек.
Linux Kernel	N/A	N/A	>321.21 сек.

Сукупне порівняння детектора SAT SIG з детектором МНІДП



Таблиця 4. Вимірювання розширення на основі МНІДП та ЛЧХ

Проект	Час створення індексу (сек)		Середній час інкрементного кроку (сек)		Стандартне відхилення інкрементного кроку (сек)		Пам'ять (MB)	
	МНІДП	ЛЧХ	МНІДП	ЛЧХ	МНІДП	ЛЧХ	МНІДП	ЛЧХ
Rippled	3.75	10.42	0.85	0.82	1.31	1.35	129	60
Kooboo	16.28	37.82	0.03	0.25	0.04	0.04	348	122
Tensorflow	87	192.8	4.29	1.57	3.32	2.13	1791	524
OpenJDK-14	51.62	139.87	4.72	4.26	5.07	6.34	1712	429
Linux Kernel	321.21	954.56	N/A	4.38	N/A	10.23	12500	2600

Рисунок 6. Час виконання процесу створення індексу для двох реалізацій – МНІДП та ЛЧХ

Наукові публікації

1. Праворська Н.І., Бедратюк Л.П, Форкун Ю.В. Яшина О.М. Мовно-незалежний детектор для виявлення і усунення повторів та надлишковостей програмного коду. Вимірювальна та обчислювальна техніка в технологічних процесах. – Хмельницький, 2021. №1, с. 56-61

2. Праворська Н.І, Бармак О.В., Медзатий Д.М., Шестакевич Т.В. Процес виявлення блоків з повторами і надлишковістю при використанні мовно-незалежного інкрементного детектору. Вісник ХНУ, серія Технічні науки, №3, 2021, с.39-45

Висновки та рекомендації

В результаті виконання дипломної роботи було проведено системний аналіз в області виявлення повторів та надлишковостей програмного коду, визначено недоліки існуючих рішень.

Запропоновано метод та рішення, які дозволять покращити знаходження блоків з повторами, та доведено їх ефективність. На основі отриманих даних спроектовано та втілено у конкретний прикладний результат (детектор) концепцію мовно-незалежного інкрементного підходу, що вирішує проблеми майбутнього рефакторингу програмних проектів.

В ході експериментів також встановлено, що підхід МНІДП виявився набагато швидший, ніж очікувалося, при цьому залишаючи обмежені можливості для подальшого покращення.

Підґрунтям для можливої інтеграції запропонованого підходу в рамках моделі ремонтопридатності виступає те, що він має здатність виявляти блоки з повторами та надлишковістю поза контекстом коміту.

Mon Nov 22 13:27:09 EET 2021, Хіврич Володимир Русланович, Хмельницький національний університет, ХНУ

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 35.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 12%**

ID: 96947 Назва: Удосконалення методів виявлення повторів та надлишковості програмного коду Додано в БД: 2021-11-22 Автора: Н.І. Праворська Керівники: Л.П. Бедратюк Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	144265	1190	52927 (37%)	474 (40%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми
95880	Назва: Звіт з переддипломної практики Додано в БД: 2021-09-28 Автора: Праворська Н.І. Керівники: Бедратюк Л.П. Консультанти: Опоненти:	50831 (35.0%)	446 (37.0%)

UNICHECK

Ім'я користувача:
Кафедра ІПЗ

ID перевірки:
1009361791

Дата перевірки:
26.11.2021 09:34:47 EET

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
26.11.2021 09:57:10 EET

ID користувача:
100005589

Назва документа: **ДР_Праворська**

Кількість сторінок: **102** Кількість слів: **20942** Кількість символів: **156961** Розмір файлу: **1.52 MB** ID файлу: **1009344152**

18.1%
Схожість

Найбільша схожість: **12.2%** з Інтернет-джерелом (http://elar.khnu.km.ua/jspui/bitstream/123456789/10624/1/9-2.pdf)

17.1% Джерела з Інтернету	346	Сторінка 104
1.65% Джерела з Бібліотеки	94	Сторінка 108

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи	22

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: «Удосконалення методів виявлення повторів та надлишковості програмного коду»

Автор: Праворська Наталія Іванівна
Спеціальність: 121 – Інженерія програмного забезпечення
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Науковий керівник: Бедратюк Леонід Петрович, д. ф.-мат. н., професор
Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	**відповідає**
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Сумарний обсяг всіх запозичень, визначений системами виявлення збігів ідентичності/ схожості, а саме:

- системою Anti-Plagiarism, складає 35% і адресується до 1 джерела, а саме 2-го розділу звіту з переддипломної практики студента, який є складовою частиною роботи;
- системою Unicheck, складає 18,1% і адресується до одного джерела, а саме виданої студентом статті у фаховому журналі, які є складовою частиною дипломної роботи.

Таким чином, всі виявлені запозичення, з урахуванням наведених обґрунтувань, не є плагіатом. Робота приймається до захисту.

Керівник _____ Л. П. Бедратюк

Гарант ОП _____ О. М. Яшина

Завідувач кафедри _____ Л. П. Бедратюк

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА ДИПЛОМНУ РОБОТУ

Дипломник Праворська Наталія Іванівна

Тема Удосконалення методів виявлення повторів та надлишковості програмного коду

Спеціальність 121 – Інженерія програмного забезпечення

Обсяг дипломної роботи:

Кількість листів креслень _____; кількість сторінок записки 103

1.Короткий зміст ДР та прийнятих рішень У магістерській роботі проведено аналіз існуючих підходів та методів виявлення повторів та надлишковості програмного коду. Удосконалено існуючий текстовий інкрементний підхід. Спроектовано мовно-незалежний інкрементний детектор повторів. Проведено оцінювання в контексті SIG та порівняння його з існуючим детектором компанії. Досліджено локально-чутливе хешування, як способу розширення такого підходу, для подолання деяких недоліків. Проводилося порівняння підходів один з одним, та дослідження потенціалу ЛЧХ в контексті даного дослідження.

2. Висновок про відповідність ДР поставленому завданню Дипломна робота освітнього ступеня «магістр» у повній мірі відповідає поставленому завданню, як у теоретичній, так і практичній її частині

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У вступі обґрунтовується актуальність теми роботи, формулюються цілі і завдання дослідження, описується наукова новизна та практична значимість отриманих результатів. У першому розділі охарактеризовано структуру предметної області, існуючі підходи та методи виявлення повторів та надлишковостей в програмному коді, виконана розгорнута постановка задачі. У другому розділі досліджено методи та підходи вирішення поставлених задач. Взятий за основу текстовий інкрементний метод був модифікований з метою досягнення поставленої задачі і отримання бажаного результату та представлено розширення інктементаного детектора повторів. Спроектовано та розроблено МНІДП. У третьому розділі були розглянуті способи розширення алгоритму додаткового виявлення повторів і надлишковості в коді та проведено вдосконалення для досягнення незалежності від мови програмування. У четвертому розділі розглянуто питання, що стосується кількісних експериментів, які були проведені та пояснення їх відповідності завданням даного дослідження. Також було проведено емпіричне дослідження, спрямоване на доведення значного покращення накопиченого часу виявлення повторів в програмному коді. Обґрунтована ефективність запропонованого методу та МНІДП, представлено рекомендації з подальшого застосування.

4. Позитивні сторони роботи Дипломна робота містить низку інноваційних рішень, зокрема, було доведено ефективність модифікованого оригінального алгоритму Хаммела та проведені досліди використання ЛЧХ у спробі розширення та покращення оригінального підходу, запровадження проміжного представлення модифікованого «Індексу клону» в процесі виявлення дубльованих блоків.

5. Негативні сторони роботи В роботі була спроба покращення модифікованого оригінального алгоритму Хаммела локально-чутливим хешуванням для подолання деяких недоліків МНІДІІ, хоча в процесі експериментів виявилося, що таке розширення працювало гірше в обумовленій формі. Тому в контексті оригінального алгоритму потрібні додаткові дослідження для вивчення потенціалу підходу заснованого на ЛЧХ.

6. Оцінка графічного оформлення та пояснювальної записки роботи Графічне оформлення виконано відповідно до теми дипломної роботи з дотриманням вимог стандартів. У загальному графічне оформлення виконано на достатньому рівні. Пояснювальна записка відповідає вимогам стандартів до її оформлення.

7. Відгук про роботу в цілому В цілому дипломна робота заслуговує позитивної оцінки. Весь матеріал дипломної роботи структурований, чіткий та послідовний. Усі розділи роботи є послідовними та логічними, що дозволяє чітко зрозуміти викладений матеріал у рамках тематики дипломної роботи. Графічний матеріал дозволяє наочно побачити доцільність та ефективність рішень, які були прийняті за основу для вирішення поставленої задачі.

8. Інші зауваження

9. Оцінка дипломної роботи Розглянувши позитивні і негативні сторони представленої дипломної роботи, можна зробити висновок, що вона заслуговує оцінки «відмінно»

РЕЦЕНЗЕНТ (прізвище, ім'я, по-батькові, посада, місце роботи)
Мартинюк В. В., д. т. н., проф., завідувач кафедри автоматизації та комп'ютерно-інтегрованих технологій"

"21" 12 2021 р. (підпис)