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

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

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, 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 ІТ, 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 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 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 і його вдосконалення, 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;

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 с	4.6 с
Kooboo	22 хв	397 с	16.31 с
Tensorflow	8 год. 30 хв	177.1 с	91.29 с
Openjdk-jdk14u	N/A	N/A	56.34 с
Linux Kernel	N/A	N/A	>321.21 с

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

№ з/п	# 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.

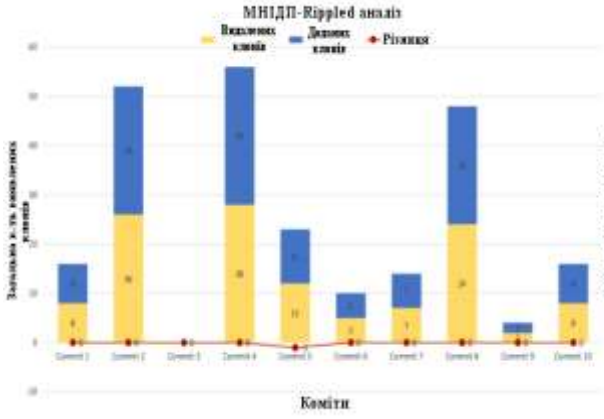

Рис. 1. Виявлення клону для Rippled

Рис. 2. Виявлення клонів для Kooboo

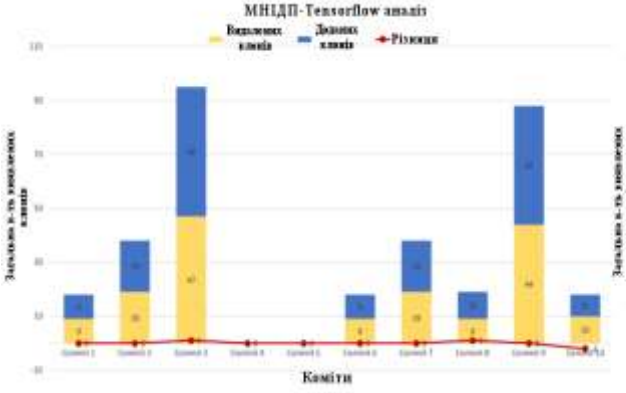

Рис. 3. Виявлення клонів для Tensorflow

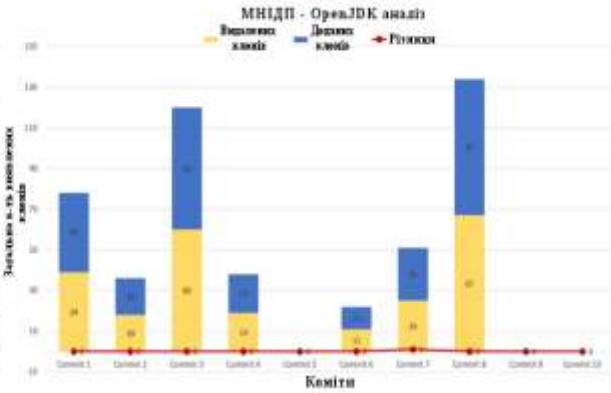

Рис. 4. Виявлення клону для OpenJDK

Доцільним став розгляд подальшого розширення та покращення детектора МНІДП, застосовуючи метод оцінки подібності найближчого сусіда – локально-чутливе хешування (ЛЧХ) [4]. Алгоритм пошуку найближчих сусідів на сьогодні є найбільш популярним ймовірнісним 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.

Рис. 5. Підоперації локально-чутливого хешування

Складається 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 генерованого підпису. Збереження інформації набору в максимально можливому ступені, забезпечується генерованими 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 смуги по три рядки. Потім при появі нового документу $D_{нов}$ потребується визначити, чи відбувається утворення пари кандидатів з іншим документом. Тоді іде 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.

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 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 компоненту, 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 інкрементних 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. Підґрунтям 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].

References

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.

8. 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.