DOI: https://doi.org/10.36910/6775-2524-0560-2022-49-05
УДК 004.4'23
Єрмоленко Денис Вадимович, бакалавр,
https://orcid.org/0000-0002-5122-4069
Марченко Олександр Іванович, к.т.н., доцент,
https://orcid.org/0000-0002-4537-3420
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського», м. Київ, Україна

АЛГОРИТМИ СПОСОБУ ПОРІВНЯННЯ ПРОГРАМ, НАПИСАНИХ LISP-ПОДІБНИМИ МОВАМИ, НА ОСНОВІ АБСТРАКТНИХ СЕМАНТИЧНИХ ДЕРЕВ

Єрмоленко Д.В., Марченко О.І. **Алгоритми способу порівняння програм, написаних Lisp-подібними мовами, на основі абстрактних семантичних дерев.** У даній статті запропоновано два алгоритми виявлення переміщених фрагментів дерев. Ці алгоритми є частинами реалізації способу порівняння програм, написаних Lisp-подібними мовами, на основі абстрактних семантичних дерев. Перший з алгоритмів для виявлення переміщених фрагментів дерев використовує обхід лісу дерев, а другий алгоритм з цією метою використовує хеш-таблицю. Було проведено порівняння швидкодії запропонованих алгоритмів. Зазначено напрямок подальших досліджень щодо покращення цих алгоритмів.
Ключові слова: лісп, s-вираз, абстрактне семантичне дерево, хеш-таблиця, обхід дерева, порівняння дерев.

Yermolenko D., Marchenko O. Algorithms of the technique for comparing programs written in Lisp-like languages based on abstract semantic trees. This article proposes two algorithms for detecting moved tree fragments. These algorithms are part of implementation of the technique for comparing programs written in Lisp-like languages based on abstract semantic trees. The first of the algorithms for detecting moved tree fragments uses traversal of the forest of trees, and the second algorithm uses a hash table. Speed of the proposed algorithms was compared. A way for further research on improving these algorithms is offered as well.
Keywords: lisp, s-expression, abstract semantic tree, hash table, tree traversing, tree comparing.

Постановка проблеми. Кодова база сучасних програмних проєктів містить велику кількість рядків коду, а розробка проєктів виконується з використанням систем контролю версій, таких як git, subversion тощо, де зміни вносяться невеликими частинами коду. Стандартні програми для перегляду змін між конкретними версіями проєкту, такі як git diff, gitk, git GUI, або сайти такі, як gitlab, чи github виконують порівняння файлів різних версій проєкту на текстовому рівні без враховування синтаксису та семантики тих мов програмування, на яких написані конкретні файли проєкту. У свою чергу, порівняння саме на структурно-семантичному рівні, на відмінну від текстового порядкового порівняння, дозволяє відкидати чисто стилістичні зміни й сфокусуватись розробникам на аналізі суттєвих структурних змін у коді, наприклад, додавання, видалення, модифікацію чи переміщення відповідних фрагментів коду. Засоби порівняння на структурному рівні ґрунтуються на способах порівняння абстрактних семантичних дерев, що враховують специфіку саме тих мов програмування, для яких були створені. Такі засоби існують для таких популярних мов програмування як C, C++, Java, C# тощо. Але менш популярні мови програмування, чи мови, що тільки набирають свою популярність таких засобів не мають. Останніми роками популярність функціонального програмування та мов, що підтримують функціональну парадигму програмування швидко зростає, наприклад мови Clojure [1]. Тому дослідження способів порівняння програм, написаних мовами із Lisp-подібним синтаксисом є задачею актуальною. У тезах конференції [2] авторами був запропонований спосіб порівняння абстрактних семантичних дерев програм, написаних Lisp-подібними мовами. Дана стаття є поглибленим та більш детальним описом алгоритмів цього способу.

Термінологія.

АСД – абстрактне семантичне дерево.

Алгоритм пошуку TED (tree edit distance) – це алгоритм пошуку відстані редагування між впорядкованими деревами, що є послідовністю мінімальних за ціною операцій редагування, які перетворюють одне дерево в інше [3].

EBNF (extended Backus-Naur form) – розширена нотація Бекуса Наура, яка є способом запису правил контекстно-вільної граматики.

UML (unified modeling language) – уніфікована мова моделювання, яка є мовою графічного опису для візуалізації, специфікації, конструювання та документування програмних систем.

LCS (longest common subsequence) – найдовша спільна підпослідовність.

S-вираз – є варіантом представлення інформації у вигляді вкладених списків. Поділяються на 2 види: списки (у контексті даної статті lisp-списки) та атоми [4].

Атом – будь-що, що не є lisp-списком. Наприклад числа, символи, рядки тощо. Є термінальними елементами s-виразів.

Lisp-список – це список, чиї елементи, а саме інші ліспові списки чи атоми, записуються зазвичай всередині круглих дужок та розділені між собою пробільними символами.

Віртуальний lisp-список – це lisp-список, який семантично існує в певному контексті, але в текстовому вигляді ніяк не виділений. Класичними прикладами такого lisp-списку є тіло функції, опис аргументів функції чи явне вказання передачі певних ключових аргументів при виклику певної функції.

Def-s-вираз – це s-вираз, який є виразом верхнього рівня та який визначає певний структурний елемент, як: функція, метод, клас, змінна, константа, пакет, макрос тощо.

Емпіричний режим пошуку переміщених піддерев – це режим, коли при збігу додаткової інформації (хешів, кількості вузлів тощо) в кореневих вузлах піддерев, вони вважаються однаковими й не проводиться перевірка на точний їх збіг.

Аналіз останніх досліджень і публікацій.

В роботі [5] представлено дослідження, у якому висловлюються критичні зауваження для застосування класичного варіанту алгоритму пошуку TED у контексті порівняння саме АСД. Було вказано такі його недоліки [5]:

- виконується в кращому випадку за час порядку $O(n^2m^2)$, де n та m це відповідно кількість вузлів у дерев, що порівнюються. Це не дозволяє використовувати його для проєктів з дуже великою кількістю рядків коду, де будемо мати АСД десь з десятками тисяч вузлів на один файл;
- операція виявлення "relabeled" (тобто перемаркованого) вузла АСД не враховує семантичного чи синтаксичного сенсу, що може, наприклад, призвести до виявлення факту хибного перемаркування вузла з цілочисельною константою у вузол, що позначає умовний оператор.
- не виконує виявлення переміщення фрагментів дерева (найважливіший недолік).

У роботі [6] показано, що, в загальному випадку, порівняння дерев з операцією знаходження переміщених фрагментів є NP-повною задачею. Через це в роботах, де досліджується порівняння АСД з знаходженням переміщених фрагментів, пропонуються певні обмеження, спрощення чи емпіричні підходи для того, щоб ця задача перестала бути NP-повною. Наприклад в [5] при попередній підготовці двох АСД, які відповідають програмам, написаним на С-подібних мовах, перед етапом порівняння пропонуються виконувати такі операції, як:

- згортання певних груп вузлів дерева, які представляють певні синтаксичні категорії, в один вузол. Наприклад для випадку Java програм, можна згортати вузли для класів, інтерфейсів, визначень, методів, полів чи блоків коду тощо;
- сплющення піддерев;
- видалення з вхідних дерев їх спільних піддерев на основі збігів хешів згорнутих вузлів та сплющених піддерев.

Але досліджень та способів порівняння, які були б спеціально адаптованими для порівняння АСД програм, написаних мовами із Lisp-подібним синтаксисом, авторами знайдено не було.

Постановка завдання.

Як зазначалося вище, авторами був запропонований спосіб порівняння абстрактних семантичних дерев програм, написаних Lisp-подібними мовами [2]. Завданням даної статті є розробка детальних алгоритмів реалізації цього способу та оцінювання їх швидкодії.

Особливості синтаксису Lisp-подібних мов.

Відзначимо важливу особливість синтаксису Lisp-подібних мов, яка полягає у використанні s-виразів. S-вирази є однією із зручних форм запису деревоподібних даних у текстовому вигляді, і їх можна поділити на дві категорії: списки та атоми. Синтаксис запису s-виразів у формі EBNF показано на рисунку 1.

```
<S-expr> ::= <Atom> | <List>;
<List> ::= '(' <S-expr>* ')' ;
```

Рис. 1. Синксис s-виразів у формі EBNF

Важливо зазначити, що нетермінал <Atom> розкривається надалі тільки у термінальні символи s-виразів, і тому атоми будуть термінальними вузлами відповідного йому дерева, а корені піддерев, які у свою чергу представляють списки, будуть нетермінальними вузлами дерева. Треба також зауважити, що пустий список, тобто "()", сприймається як специфічний атом і також буде термінальним вузлом дерева.

Відзначимо також, що при побудові АСД для s-виразів можна взагалі не змінювати форму початкового дерева розбору, з якого будується АСД, окрім випадків виділення віртуальних списків, оскільки такі списки визначаються семантикою тих чи інших s-виразів і їх неможливо виділити на етапі синтаксичного розбору.

Розглянемо коротко суть запропонованого авторами в [2] способу порівняння АСД програм, написаних Lisp-подібними мовами.

Автори пропонують виділяти наступні 3 типи вузлів дерев при порівнянні АСД двох def-s-виразів, def-1 та def-2:

- вузли типу "same" – відповідний вузол є однаковим в def-1 та def-2, тобто не змінив свого положення в дереві;
- вузли типу "unique" – відповідний вузол є унікальним для def-v1 або def-v2, тобто зустрічається тільки в одному із порінвюваних def-s-виразів;
- вузли типу "moved" – відповідний вузол та дерево, коренем якого він є, було переміщено з АСД виразу def-1 до іншої локації АСД виразу def-2.

Запропонований спосіб ґрунтується на використанні цих категорій вузлів і складається із таких етапів:

- підготовка вхідних абстрактних семантичних дерев;
- формування пар def-s-виразів для порівняння;
- виявлення вузлів типу "same"у конкретній парі порівняння;
- виявлення вузлів типів "moved" та "unique".

Опис представлення інформації у вхідних АСД. Ієрархія класів вузлів вхідних АСД показана у вигляді UML діаграми на рисунку 2. У цій ієрархії маємо базовий клас AstNode, що зберігає інформацію, яку мають усі вузли АСД. Базовий клас AstNode має підкласи AtomAstNode та ListAstNode, які відповідно зберігають інформацію про атоми, які є термінальними вузлами, та lisp-списки, які є нетермінальними вузлами дерев.

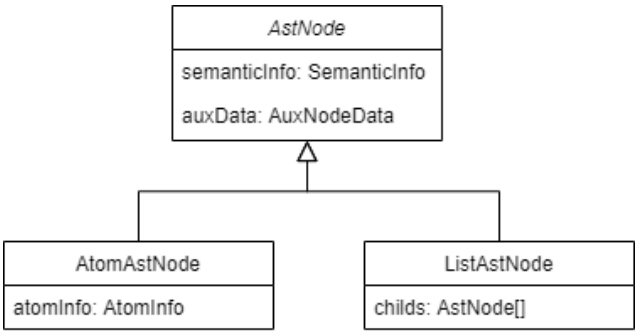

Рис. 2. UML діаграма ієрархії класів вузлів АСД

Поля підкласів AstNode повинні заповнюватися для кожного вузла на етапі підготовки вхідних АСД. На рисунку 3 показано UML діаграму класу AuxNodeData. який містить додаткову інформацію для роботи алгоритмів.

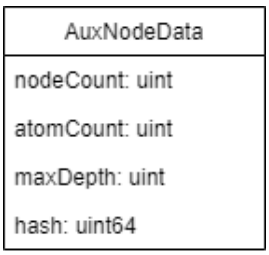

Рис. 3 UML діаграма класуAuxNodeData, який містить додаткову інформацію вузлів АСД

Поле maxDepth цього класу зберігає інформацію про максимальну глибину піддерева, коренем якого є даний вузол, а поле hash зберігає інформацію про хеш цього піддерева.

Алгоритм підготовки вхідних АСД. Для етапу підготовки вхідних АСД розроблено алгоритм prepareAst, псевдокод якого показано на рисунку 4, а алгоритм хешування вузлів дерева був взятий з [7]. Зазначимо, що для даного випадку рекомендується використовувати швидкі функції хешування, а не криптографічні.

```
function prepareAst(treeRoot: AstNode) -> AuxNodeData
   var auxData: AuxNodeData := new AuxNodeData();
   if (treeRoot instanceof AtomAstNode)
      auxData.atomCount := 1;
      auxData.nodeCount := 1;
      auxData.maxDepth := 1;
      auxData.hash := hash(treeRoot.atomInfo) +
                      hash(treeRoot.semanticInfo);
   else
      var subTreesAuxData: AuxNodeData[treeRoot.childs.size];
      for ([subtreeRoot: AuxNodeData, i: Integer] in treeRoot.childs)
          subTreesAddData[i] := prepareAst(subtreeRoot);
      auxData += hash(treeRoot.semanticInfo);
      for ([stAddData, i] in subTreesAddData)
          auxData.atomCount += stAddData.atomCount;
          auxData.nodeCount += stAddData.nodeCount;
          auxData.hash += pow(31,i + 1) * auxData.hash;
      auxData.maxDepth := findMaxDepth(subTreesAddData) + 1;
   setAuxData(treeRoot, auxData);
   return auxData;
```

Рис. 4. Псевдокод алгоритму підготовки вхідних АСД

Алгоритм формування пар def-s-виразів для порівняння. Для реалізації наступного етапу запропонованого способу був розроблений алгоритм формування пар def-s-виразів однакових типів для подальшого їх порівняння за їх іменами. Вхідними даними цього алгоритму є списки інформації про def-s-вирази кдасу Def-s-expr, UML-діаграма якого показана на рисунку 5.

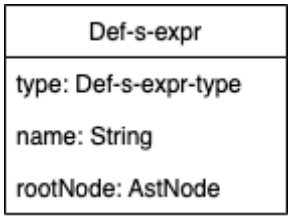

Рис. 5. UML діаграма класу Def-s-expr, який містить інформацію про def-s-вираз

Псевдокод цього алгоритму показано на рисунку 6.

```
type-alias Def-s-expr-pairs = List<Pair<Def-s-expr,Def-s-expr>>

function get-def-s-expr-pairs(def-s-exprs-1 : List<Def-s-expr>,
                              def-s-exprs-2 : List<Def-s-expr>) -> Def-s-expr-pairs
    var def-s-expr-pairs = new Def-s-expr-pairs();
    for (def-s-expr in def-s-exprs-1)
        var found-def-s-expr: Def-s-expr = findByTypeAndName(def-s-expr, def-s-exprs-2);
        if (found-def-s-expr is not null)
            addTo(def-s-expr-pairs, makePair(def-s-expr, found-def-s-expr));
    return def-s-expr-pairs;
```

Рис. 6. Псевдокод алгоритму формування пар def-s-виразів за іменем

Приклад результату роботи формування пар порівняння для файлів v1 та v2, які містять програми, записані мовою Common Lisp, показано на рисунку 7.

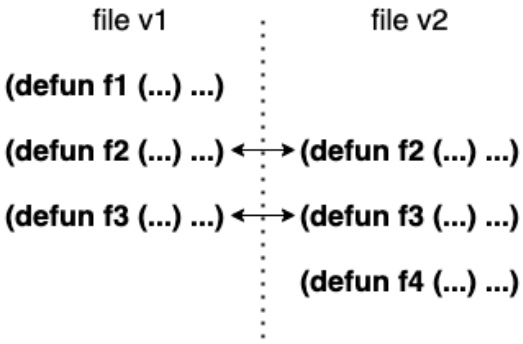

Рис. 7. Приклад утворення пар порівняння за іменем та типом def-s-вираза

Алгоритм етапу виявлення вузлів типу "same". Для виявлення «same» вузлів використовується модифікований алгоритм пошуку LCS, який був запропонований в [8].

Цей алгоритм формує співпадаючі пари вузлів в абстрактних семантичних деревах виразів def-1 та def-2, позначені як «same» вузли, а також відповідні списки піддерев def-1-unsure-subtrees та def-2-unsure-subtrees, які містять корені піддерев, статус вузлів яких допоки невідомий.

Алгоритми виявлення вузлів типів "moved" та **"unique"**. Виявлення вузлів типу «moved» – це фактично операція виявлення переміщених піддерев. Пропонується два алгоритми для виконання цього етапу:

- алгоритм з використанням обходу вузлів лісу піддерев def-2-unsure-subtrees;
- алгоритм з використанням хеш-таблиці, куди заносяться всі піддерева з def-2-unsure-subtrees.

Обидва ці алгоритми використовують одну спільну структуру даних, а саме список maybe-moved-from-v1, який формується з піддерев списку def-1-unsure-subtrees. Основний інваріант списку maybe-moved-from-v1 полягає у тому, що елементи списку (тобто піддерева) мають знаходитися в порядку спадання кількості вузлів у піддеревах.

Обидва алгоритми використовують однаковий базовий алгоритм, псевдокод якого показано на рисунку 8.

```
function find-moved-and-unique (maybe-moved-from-v1: List<AstNode>,
                                maybe-moved-in-v2: T)
  while (maybe-moved-from-v1 is not empty)
    val pattern-tree: AstNode = pop(maybe-moved-from-v1);
    val equal-tree-in-v2: AstNode = find-in(maybe-moved-in-v2,
                                            pattern-tree);
    if (equal-tree-in-v2 is not null)
      mark-all-node(pattern-tree, MOVED);
      mark-all-node(equal-tree-in-v2, MOVED);
    else
      mark-root(pattern-tree, UNIQUE);
      val list-of-SubTs: List<AstNode> = decay(pattern-tree);
      add-correctly-to(maybe-moved-from-v1,list-of-SubTs);
```

Рис. 8. Псевдокод базового алгоритму

Різниця між цими двома алгоритмами полягає в типі аргументу maybe-moved-in-v2 та реалізації функції find-in (на рисунку 8 вони виділені жовтим кольором).

Опишемо деталі реалізації алгоритму з використанням обходу вузлів лісу піддерев.

Функція цього алгоритму має параметр maybe-moved-in-v2 типу List<AstNode>, до якого при виклику цієї функції передається аргумент def-2-unsure-subtrees, що є списком вузлів АСД.

Функція find-in реалізує обхід по лісу піддерев, що знаходиться в списку maybe-moved-in-v2 й намагається знайти в ньому піддерево, у якого статус не є «moved» та всі додаткові параметри збігаються з додатковими параметрами дерева шаблону pattern-tree. Якщо при цьому використовується емпіричний режим пошуку переміщених піддерев, то вважається, що виявлене піддерево збігається з шаблонним деревом pattern-tree емпірично, в іншому випадку, проводиться точна перевірка на збіг. Особливе значення для оптимізації обходу дерева при роботі цього алгоритму має інформаційне поле вузла max-depth: якщо значення поля max-depth шаблонного дерева pattern-tree вже збігається зі значенням поля max-depth поточного дерева current-tree, що перевіряється, але співпадіння ще не було виявлене, то подальшого заглиблення у дерево current-tree не відбувається.

Опишемо деталі реалізації алгоритму з використанням хеш-таблиці.

Функція цього алгоритму має параметр maybe-moved-in-v2 типу HashTable<uint64, List<AstNode>>, до якого при виклику цієї функції передається аргумент maybe-moved-in-v2-ht, що є хеш-таблицею з ключами типу uint64, значеннями якої є список вузлів АСД.

Хеш-таблиця maybe-moved-in-v2-ht формується проходом по списку maybe-moved-in-v2 та занесенням піддерев із цього списку до комірок хеш-таблиці, в яких знаходяться списки піддерев, що мають однаковий з ними хеш. Приклад такої ініціалізації показано на рисунку 9. При створенні хеш-таблиці потрібно враховувати, що її розмір не буде перевищувати сумарної кількості всіх піддерев, що знаходяться у списку def-2-unsure-subtrees. Тому, якщо відома сумарна кількість всіх піддерев, то доцільно передавати цю кількість до функції створення хеш-таблиці, щоб задати її початковий розмір, оскільки це прискорить ініціалізацію хеш-таблиці.

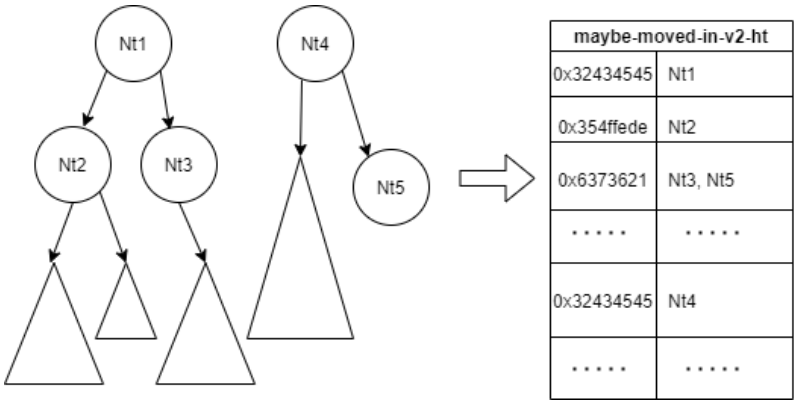

Рис. 9. Приклад ініціалізації хеш-таблиці maybe-moved-in-v2-ht
піддеревами зі списку def-2-unsure-subtrees

Функція find-in бере у шаблонного дерева pattern-tree його хеш та шукає відповідний йому список піддерев в хеш-таблиці maybe-moved-in-v2-ht. Приклад роботи функції find-in у алгоритмі з використанням хеш-таблиці показано на рисунку 10. Якщо відповідний список було знайдено, то у ньому виконується пошук піддерева, тип якого є відмінним від "moved" та яке емпірично чи точно збігається з шаблонним деревом pattern-tree. Якщо таке піддерево було виявлено, то воно зі списку не видаляється, оскільки для подальшої роботи алгоритму це не потрібно, але така операція тільки збільшить час його роботи. Тобто під час роботи структура хеш-таблиці не змінюється, а змінюються тільки статуси піддерев, що знаходяться у ньому.

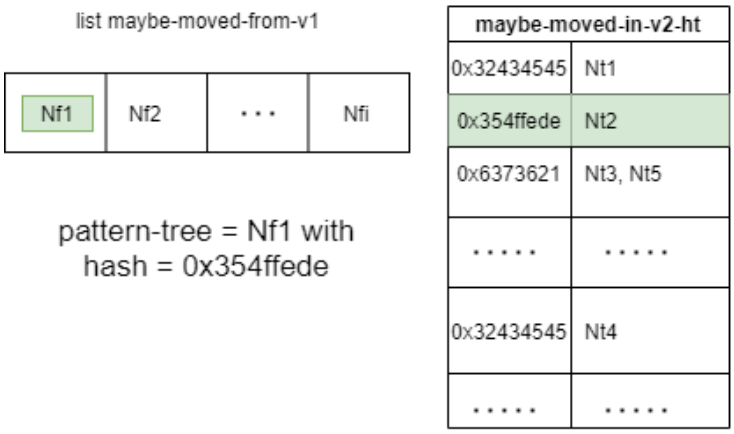

Рис. 10. Приклад роботи функції find-in у алгоритмі з використанням хеш-таблиці

Порівняння швидкодії алгоритмів виявлення переміщених фрагментів. Для вимірювання швидкодії алгоритмів було створено програму, яка генерує дві версії Lisp-файлу, що мають певну задану структуру, але є різними. Ці файли містять одне визначення функції ім'я якої однакове у двох версіях вхідних файлів. Ця функція містить три виклики довільних функцій, передаючи їм три аргументи, які можуть бути або атомами у вигляді символів, або іменами інших довільних функцій з трьома параметрами. Одним із головних параметрів при генерації Lisp-файлів є значення максимальної глибини викликів max-depth.

Перший випадок вимірювання швидкодії: тіло певної функції в обох версіях є зовсім різним, але має однакову структуру. Цей випадок вимірювання визначає швидкість виявлення повної незбіжності тіл функцій і цікавий тим, що всі можливі для виділення піддерева не є переміщеними й, таким чином, відбувається «максимальне» навантаження на представлені алгоритми. Порівняння швидкості пошуку обома алгоритмами для цього випадку показано на рисунку 11. Як видно з цього рисунку, алгоритм з обходом лісу має перевагу до значення max-depth=2, після чого алгоритм з використанням хеш-таблиці стає швидшим.

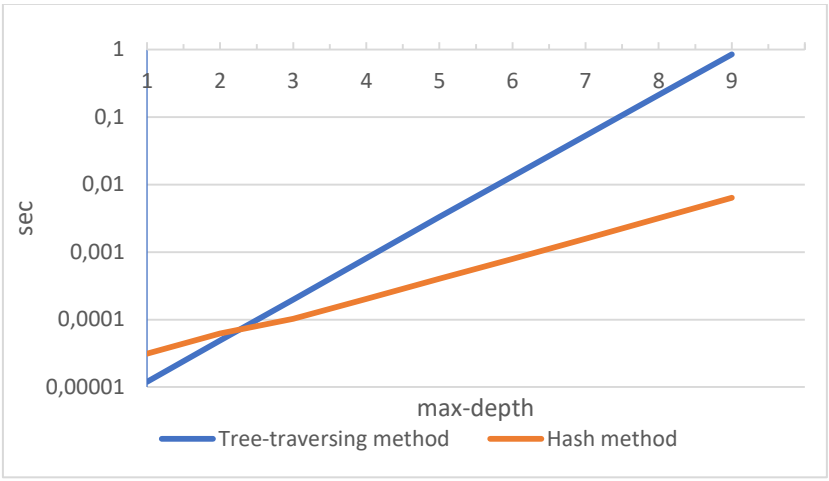

Рис. 11. Графік тестування 1-го випадку вимірювання швидкодії

Другий випадок вимірювання швидкодії: дерева вхідних АСД в обох версіях є різними до ярусу, де вже знаходяться переміщені фрагменти (назвемо цей параметр як depth-of-moved-fragments). Значення max-depth не змінюється й встановлюється рівним 14. Порівняння швидкості пошуку обома алгоритмами для цього випадку (рис. 12) показує цікавий результат: алгоритм з обходом лісу дерев має менший час роботи, тобто виграє по швидкодії до значення depth-of-moved-fragments=2, а далі починає лінійно уповільнюватися. З іншої сторони, алгоритм з використанням хеш-таблиці не тільки почав показувати кращу швидкість роботи, але його швидкодія навіть трохи зростала до значення depth-of-moved-fragments=4. Це пов'язано з тим, що при збільшенні значення depth-of-moved-fragments зменшується кількість вузлів, які треба перевіряти на збіг. Але цей факт не допомагає першому алгоритму з обходом лісу дерев, оскільки для цього алгоритму зі зменшенням кількості вузлів, які треба перевіряти, одночасно збільшується кількість вузлів, які треба обійти, щоб дійти до кореня відповідного переміщеного фрагмента.

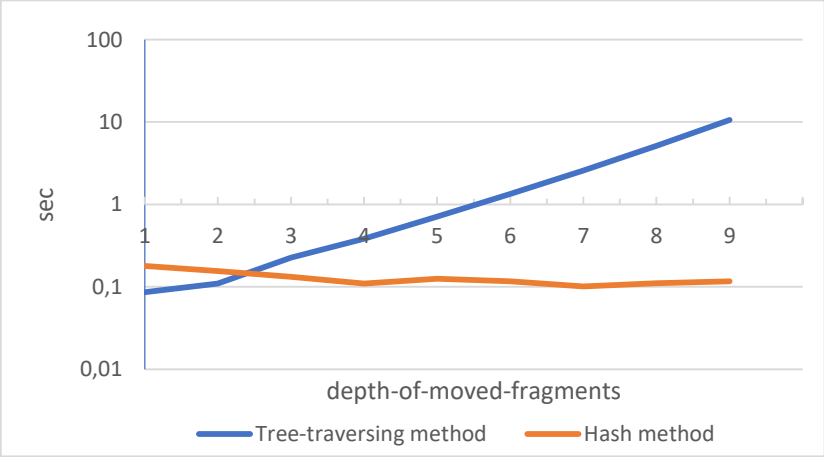

Рис.12. Графік тестування 2-го випадку вимірювання швидкодії

Висновки. У даній статті запропоновано два алгоритми виявлення переміщених фрагментів дерев, які є частинами реалізації раніше запропонованого авторами способу порівняння програм, написаних Lisp-подібними мовами, на основі абстрактних семантичних дерев. Перший з алгоритмів з метою виявлення переміщених фрагментів дерев використовує обхід лісу дерев, а другий алгоритм використовує хеш-таблицю.

Було також проведено порівняння швидкодії обох запропонованих алгоритмів та отримано результати, що демонструють більшу швидкодію алгоритму, який використовує хеш-таблиці під час знаходження переміщених фрагментів, перед алгоритмом, який використовує обхід лісу дерев, майже в усіх випадках.

Напрямком подальших досліджень для виявлення переміщених фрагментів програм може бути пошук варіантів враховування факту перейменування змінних, які не змінюють семантику цих переміщених фрагментів.

Список бібліографічного опису

1. State of Clojure 2022 [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://clojure.org/news/2022/06/02/state-of-clojure-2022.
2. Єрмоленко Д. В., Марченко О.І. Спосіб порівняння абстрактних семантичних дерев програм написаних Lisp-подібними мовами. Прикладна математика та комп'ютинг. ПМК, 2022 : п'ятнадцята наук. конф. магістрантів та аспірантів, 16–18 листопада 2022 р.: зб.тез доп./[редкол.: Дичка І.А. та ін.]. – К. : Просвіта, 2022. – с. 261-265.
3. Tree edit distance [Електронний ресурс] – Режим доступу до ресурсу: http://tree-edit-distance.dbresearch.uni-salzburg.at/.
4. S-expression [Електронний ресурс] – Режим доступу до ресурсу: https://www.computerhope.com/jargon/s/s-expression.htm.
5. Hashimoto, Masatomo, Mori. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. Proceedings [Електронний ресурс] / Hashimoto, Masatomo, Mori, Akira // Working Conference on Reverse Engineering, WCRE. 279 - 288. – 2008. – Режим доступу до ресурсу: https://doi.org/10.1109/WCRE.2008.44
6. Magniez F., .Rougemont M. Property testing of regular tree languages [Електронний ресурс] / F. Magniez and M. de Rougemont // Algorithmica, 49(2):127–146. – 2007. – Режим доступу до ресурсу: https://doi.org/10.1007/s00453-007-9028-3
7. Hashing tree structure [Електронний ресурс] – Режим доступу до ресурсу: https://www.baeldung.com/cs/hashing-tree.
8. *Єрмоленко, Д. В.* Засіб порівняння версій програм на мові LISP з використанням абстрактного семантичного дерева : дипломний проєкт бакалавра : 123 Комп'ютерна інженерія / Єрмоленко Денис Вадимович. – Київ, 2021. – 75 с.

References

1. "State of Clojure 2022", Robert Randolph , Clojure, https://clojure.org/news/2022/06/02/state-of-clojure-2022. Accessed 20 Nov. 2022.
2. Yermolenko D., Marchenko O. Algorithms of the technique for comparing programs written in Lisp-like languages based on abstract semantic trees. Applied Mathematics and Computing. AMC, 2022 : fifteen scientific conference of master students and postgraduates, 16–18 листопада 2022 р.: зб.тез доп./[редкол.: Дичка І.А. та ін.]. – К. : Просвіта, 2022. – с. 261-265.
3. "Tree edit distance", *Mateusz Pawlik*, http://tree-edit-distance.dbresearch.uni-salzburg.at/. Accessed 21 Nov. 2022
4. "S-expression", *Computer Hope*, https://www.computerhope.com/jargon/s/s-expression.htm. Accessed 21 Nov. 2022
5. *Hashimoto, Masatomo & Mori, Akira*. (2008). Diff/TS: A Tool for Fine-Grained Structural Change Analysis. Proceedings - Working Conference on Reverse Engineering, WCRE. 279 - 288. 10.1109/WCRE.2008.44. https://doi.org/10.1109/WCRE.2008.44. Accessed 21 Nov. 2022
6. *F. Magniez and M. de Rougemont*. Property testing of regular tree languages. Algorithmica, 49(2):127–146, 2007. https://doi.org/10.1007/s00453-007-9028-3. Accessed 21 Nov. 2022
7. "Hashing tree structure", *Said Sryheni*, Baeldung, https://www.baeldung.com/cs/hashing-tree. Accessed 22 Nov. 2022
8. Yermolenko D. V. (2021). Tool for comparing versions of programs in the LISP language using an abstract semantic tree [Diploma project, NTUU KPI]