УДК 004.43

[1]Амонс О.А., к.т.н.
[1] Киричек О.О.,
[2]Киричек Г.Г., к.т.н., доц.

СПОСІБ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ВИЯВЛЕННЯ ПЛАГІАТУ ПРОГРАМНОГО КОДУ C# НА ОСНОВІ ШАБЛОНІВ

[1] Національний технічний університет України «КПІ»
[2] Запорізький національний технічний університет

Розглянуто питання розробки, дослідження та удосконалення існуючих алгоритмів пошуку плагіату в програмному коді, а також відсіювання шаблонних ділянок коду, які вносять суттєву помилку при визначенні подібності. Реалізація даного алгоритму дозволяє підвищити точність оцінки наявності плагіату в програмному коді за рахунок застосування до проектів алгоритму фільтрації шаблонних ділянок коду та формально перетворених ділянок

Вступ

Інтенсивний розвиток інформаційних технологій та розширення сфер їх застосування привів до того, над розробкою та супроводом таких систем зараз працюють мільйони програмістів, які пишуть величезну кількість програмного коду. У зв'язку з цим важною проблемою стає пошук плагіату у програмному коді.

Задачею систем пошуку плагіату є автоматичне виявлення (за заданими критеріями) того, чи була використана у програмі деяка чужа ідея, або фрагменти програмного коду. На практиці певним чином задаються деяка функція близькості і поріг, за якими можна визначити наскільки ймовірно, що деяка частина програмного коду була запозичена [1].

Розробка застосувань на платформі Framework.Net виконується переважно мовою *C#*, яка найбільш оптимізована під платформу. Одною з переваг розробки з використанням сучасних засобів розробки є автоматична генерація базового каркасу застосувань, форм та спеціалізованих класів. Це знімає з програміста роботу по виконанню рутинних задач з формуванню однотипного програмного коду, який на відрізняється від аналогічного коду у інших проектах. Але з точки зору аналізу програмного коду на плагіат це вносить суттєву помилку в загальну оцінку.

В статті пропонується спосіб підвищення ефективності виявлення плагіату програмного коду *C#* за рахунок виключення автогенерованого програмного коду. Даний підхід розвиває алгоритм виявлення плагіату [2], що інтегровано до розробленої системи виявлення плагіату [3].

Актуальність і доцільність даного дослідження полягає у необхідності покращення алгоритмів визначення повторного використання програмного коду у творчих проектах студентів, задля підвищення точності аналізу власного внеску студентів проектів, за рахунок застосування до розробленого проекту алгоритму фільтрації шаблонних ділянок коду та формально перетворених ділянок.

Плагіат у програмному коді

Пошук плагіату у програмних застосувань краще виконувати на основі програмного коду. Переваги в тому, що він містить більше специфічних властивостей та характеристик притаманних автору. Вміст *.exe, .dll* та інших бінарних кодів теж можна аналізувати, але вони містять вже набагато менше специфіки і при виконанні різних компіляторів бінарних кодів на основі вихідних кодів або різних налаштувань оптимізації на виході будуть різні за вмістом бінарні файли. Слід зазначити, що при експертних юридичних оцінках теж використовується програмний код.

Пошук плагіату у програмному коді може використовується з різною метою:

- Аналіз плагіату у юридичній експертизі [4-6];
- Аналіз плагіату у освіті з метою виявлення власної частини роботи студента у навчальній роботі (лабораторних та практичних роботах, курсових, дипломах, тощо.) [2,7,8].
- Аналіз плагіату при ревізіях програмних проектів з метою оцінки якості програмного коду, та прийняття рішення про необхідність оптимізації частин програмного коду [1,4,5].

Огляд існуючих алгоритмів

Алгоритми виявлення плагіату у програмних кодах прийнято поділяти на атрибутно-статистичний, структурний, комбінований [5, 8].

Атрибутні методи основані на чисельному виразі деяких при знаків програми та порівняння отриманих чисел. Програми з близькими чисельними характеристиками потенціально схожі [9]. Недоліком такого підходу є те, що різні програми можуть отримувати близькі характеристики [8].

Структурні методи враховують контекст програм. Такі підходи використовують токенізацію програмного коду для більш компактного його представлення [1,8].

Недоліком структурних методів є їх складність та обчислювальна трудоємність, а також реалізація, як правило, опирається на синтаксис конкретного язика програмування [4, 9].

Прикладами методів основаних на токенізації є метод відбитків [11] та алгоритм Хескела [12].

Комбінований підхід використовується для пошуку плагіату у великій базі програм [8]. Для цього на першому етапі за допомогою атрибутного метода відбираються схожі програми, а на другому етапі виконується порівняння відібраних програм структурним методом.

В статі розглядається додаткова обробка отриманого токенізованного представлення програмного коду для фільтрації авто генерованого коду. Після фільтрації токенізоване представлення можна використовувати для роботи у структурних методах Хескела та методі відбитків [2-3].

Спосіб підвищення ефективності виявлення плагіату програмного коду

Запропонований спосіб підвищення ефективності виявлення плагіату програмного коду виконується у 2 проходи.

- Відсікання програмного коду *C#*, що автоматично генерується середовищем розробки *.Net*, наприклад, *MS Visual Studio.*
- Приведення фрагментів алгоритмів програмного коду *C#* до уніфікованого вигляду.

Обидва підходи основуються на трансформації токенізованого представлення на основі шаблонів. Для підвищення якості відсікання та приведення коду на відміну від базового алгоритму токенізації нам необхідно тимчасово зберегти іменування змінних. Після завершення трансформації іменування змінних з токенізованого представлення видаляється.

Токенізація програмного коду

Задача токенізованого представлення коду – збереження суттєвих та ігнорування легко модифікованих деталей.

Токенізація виконується наступним чином:

- Кожному оператору мови приписується код. Коди токенів сформовані для мови *C#* представляються у таблиці 1.
- З отриманих кодів будується рядок, зберігаючи послідовність токенів у відповідності до програмного коду. Один токен – один оператор.
- Тимчасово при токенізації змінним призначаємо токени з діапазону *X*00-*X*99

Таблиця 1. Зарезервовані слова, які використовуються в алгоритмах

№	Токен	Слово
1	001	abstract
2	002	as
3	003	base
4	004	bool

5	005	break
6	007	case
7	008	catch
8	009	char
9	010	checked
10	011	class
11	012	const
12	013	continue
13	014	decimal|float|double
14	015	default
15	016	delegate
16	017	do
17	019	else
18	020	enum
19	021	event
20	022	explicit
21	023	extern
22	024	false
23	025	finally
24	026	fixed
25	028	for|while|foreach
26	030	goto
27	031	if
28	032	implict
29	033	in
30	034	int|long|short|sbyte|byte|int16|int32
31	035	interface
32	036	internal
33	037	is
34	038	lock
35	041	new
36	042	null
37	043	object
38	044	operator
39	045	out
40	046	override
41	047	params
42	048	private
43	049	protected
44	050	public
45	051	readonly
46	052	ref
47	053	return
48	055	sealed
49	057	sizeof
50	058	stackalloc
51	059	static
52	060	string
53	061	struct
54	062	switch
55	063	this
56	064	throw
57	065	true
58	066	try
59	067	typeof
60	068	uint|ulong|ushort
61	070	unchecked
62	071	unsafe
63	074	virtual
64	075	volatile
65	076	void
66	078	dictionary
67	079	list
68	080	{
69	081	}
70	082	<
71	083	>
72	084	collection
73	085	[]
74	086	()
75	087	=
76	088	;
77	089	++
78	090	+=
79	091	in
80	092	=>

Відсіювання автосгенерованих ділянок коду

Видалення з файлів автосгенерованих ділянок коду, досягається, виключивши токенізоване подання фрагментів коду, що відповідають шаблонам з використанням яких створюються проекти [2,3].

Розглянемо приклад проведення відсіювання шаблонних ділянок коду на прикладі файлу .Designer.cs.

Застосуємо алгоритм відсіювання для фрагмента програми представленого у лістингу 1:

Лістинг 1.

```
// -----------------------------------------------
// <copyright file="MainWindow.xaml.cs" company="NDIIP" />
// -----------------------------------------------
using System.Windows;
using BookReader.ViewModel;
using BookReader.Model.CommonLogic;
namespace BookReader.View.Views
{
   public partial class MainWindow : Window
   {
      public MainWindow()
      {
         InitializeComponent();
         MainViewModel vm = new MainViewModel();
         Serializer ser = new Serializer();
         vm.CommonBookFormatList =
ser.DeSerializeBooks("BooksFromLastSession.txt");
         foreach (CommonBookFormat item in vm.CommonBookFormatList)
         {
            vm.OpenCommonFormat(item);
         }
         DataContext = vm;
      }
      private void Tombstoning()
      {
         Serializer ser = new Serializer();
         ser.SerializeBooks("BooksFromLastSession.txt", (DataContext as
MainViewModel).CommonBookFormatList);
         (DataContext as MainViewModel).SerializeRecentBooks();
      }
      /// <summary>
      /// Handling event of closing application by menu option.
      /// </summary>
      private void Window_Closing(object sender, RoutedEventArgs e)
      {
         Tombstoning();
         App.Current.Shutdown();
      }
   }
}
```

В результаті застосування алгоритму отримано ділянку коду, який містить лише внесені в порівнянні з шаблоном зміни. Результуюче відображення коду зберігається у вигляді послідовності токенів. Наведений у лістингу 2 фрагмент програми відповідає отриманій послідовності:

Лістинг 2.

```
MainViewModel vm = new MainViewModel();
Serializer ser = new Serializer();
vm.CommonBookFormatList = ser.DeSerializeBooks("BooksFromLastSession.txt");
foreach (CommonBookFormat item in vm.CommonBookFormatList)
{
    vm.OpenCommonFormat(item);
}
DataContext = vm;
private void Tombstoning()
{
    Serializer ser = new Serializer();
    ser.SerializeBooks("BooksFromLastSession.txt", (DataContext as
MainViewModel).CommonBookFormatList);
    (DataContext as MainViewModel) .SerializeRecentBooks();
}
private void Window_Closing(object sender, RoutedEventArgs e)
{
Tombstoning();
App.Current.Shutdown();
}
```

Пошук шаблонів на приведення фрагментів коду до універсального вигляду

Задля аналізу на наявність схожих алгоритмів в програмному коді необхідно брати до уваги змінні, а отже для них необхідно виділити діапазон дозволених значень.

Тобто для кожної зі змінних виділяємо токен з діапазону *X*00-*X*99.

Наприклад маємо наступну ділянку коду:

```
public int SumFor(int[] array)
{
   for (var index = 0; index <
array.Length; index++)
   {
      _sum += array[index];
   }
   return _sum;
}
```

Згідно таблиці токенізації (табл.1) даний код буде виглядати наступним чином:

public|int|Variable1|int|[]|variable2|{|for|variable3|=|;|variable3|<|variable2|;|variable3|++|{|variable4|+=|variable2|variable3|;|}|return|variable4|;|}

050|034|X01|034|085|X02|080|028|X03|087|088|X03|082|X02|088|X03|089|080|X04|090|X02|X03|088|081|053|X04|088|081

Де *X*01, *X*02, *X*03, *X*04 змінні, які можуть бути будь якими. Послідовність токенів, що відповідає наведеній, буде означати, що алгоритм той самий.

Усі вони виконують одну функцію. При необхідності визначення більш глибокого аналізу з урахуванням алгоритмічної подібності, необхідно щоб токенізоване представлення кожного з приведених ділянок коду було одне для всіх. Тобто в процесі токенізації ми можемо виділити послідовність операторів і змінних, які відповідають відомому алгоритму. Це дає нам можливість змінити існуючу послідовність і представити ділянки однакового алгоритму згідно механізму вкладених токенів наведених у роботі [13].

Розглянемо фрагменти коду що описують методи знаходження суми та їх токенізовані формули:

1. Сума елементів масиву з використанням циклу *For*.

```
public int SumFor(int[] array)
{
   for (var index = 0; index < array.Length;
index++)
   {
      _sum += array[index];
   }
   return _sum;
}
```

Токенізоване представлення:

050|034|X01|034|085|X02|080|028|X03|087|088|X03|082|X02|088|X03|089|080|X04|090|X02|X03|088|081|053|X04|088|081

2. Сума елементів масиву з використанням *Foreach*.

```
public int SumForeach(int[] array)
{
   foreach (var i in array) _sum += i;
   return _sum;
}
```

Токенізоване представлення:

050|034|X01|034|085|X02|080|028|X03|091|X02|X04|090|X03|088|053|X04|088|X04|088|081

3. Сума елементів масиву з використанням *lambda* виразів.

```
public int SumLambda(int[] array)
{
   array.ToList().ForEach(item => { _sum
+= item; });
   return _sum;
}
```

Токенізоване представлення:

050|034|X01|034|085|X02|080|X02|X03|092|080|X04|090|X03|088|081|088|053|X04|088|081

Розглянемо порівняння формальних представлень для першого і третього алгоритмів:

1. A | (override|virtual) | B & { & B & [&] & X01 &) & { & for & (& X02 & = & ; & X02 & (<,>,<=,>=) & ; & X02 & (++,--) & { & X03 & (+=,-=) & X01 & [& X02 &] & ; & } & return & X03 & ; & }

2. A | (override|virtual) | B & { & B & [&] & X01 &) & { & X01 & . & ToList() & . & ForEach & (& X03 & => & { & X04 & += & X03 & ; & } &) & ; & return & X04 & ; &}

A = (public|private|internal|protected)

B = (int|double|decimal|float)

Таким чином порівнюючи два алгоритми в вигляді початкового програмного коду ми можемо стверджувати, що результат їх виконання буде ідентичний. Отже, формальне представлення кожного з двох алгоритмів теж ідентичне.

Для поліпшення пошуку схожих токенів можна використовувати базу предтокенізованих уявлень схожих алгоритмів. Таке рішення дозволить скоротити час на пошук міток і оцінки плагіату.

Використання відкритих суспільних систем для наповнення подібної бази дозволить за короткий проміжок часу створити еталонні токенізовані подання відомих алгоритмів рішення тривіальних завдань.

Використовуючи настільки глибокий аналіз коду викладачеві стане простіше оцінити творчу складову рішення студентом завдання, або оцінити тривіальність запропонованого рішення.

Експериментальна частина

В експерименті використовуємо 18 проектів на мові *C#* платформи *.NET* Framework компанії Microsoft. Усі проекти однотипні, створені з використанням патерну *MVVM*.

Об’єми проектів залежать від виду роботи та курсу, на якому вона виконується. Приблизна оцінка об’ємів на основі виконаних робіт з програмування, які виконуються на кафедрі АУТС: 100 – 800 рядків коду – об’єм лабораторної роботи в залежності від курсу; близько 1600 – розрахункова робота; 3000 – об’єм курсової роботи; 4000 – 5000 робота освітньо-кваліфікаційного рівня бакалавр. Визначимо залежність кількості шаблонного коду від об’єму програми. Для отримання

результатів проаналізуємо по три проекти на кожний об'єм програми.

Результати дослідження залежності кількості шаблонного та автогенерованого коду від об'єму програми наведено на рисунку 1.

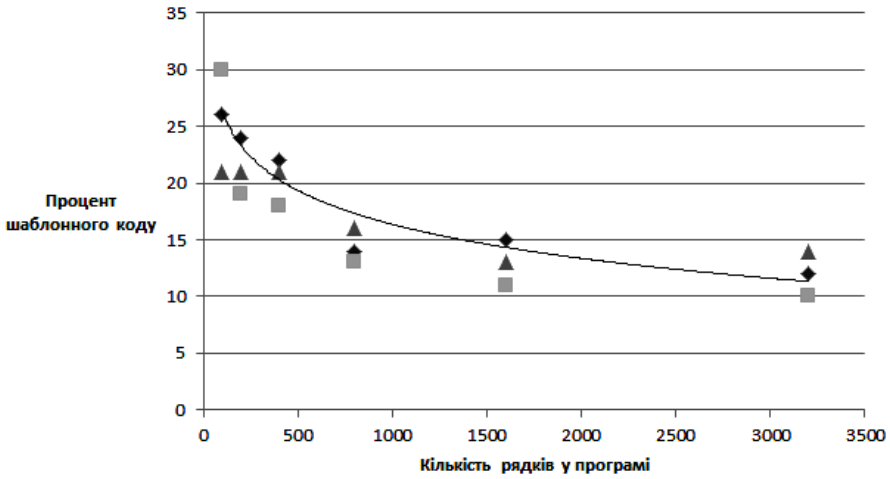

Рис. 1. Залежність проценту шаблонного коду від об'єму програми

Внесемо зміни в програмний код для отримання унікальності ~80% для шести різних по розміру програм інші 20% будуть авто генерованими та шаблонними ділянками коду. У таблиці 2 представлено результат порівняння проектів різними алгоритмами пошуку плагіату.

Результати застосування алгоритму без відсіювання наведено на рисунку 2 та з відсіюванням шаблонного коду наведено на рисунку 3.

Таблиця 2. Ефективність відсіювання шаблонних ділянок коду.

	100	200	400	800	1600	3200
Без використання відсіювання	25,5	22,55	20,8	20,31	20,1	20,05
З відсіюванням	12	8,35	6,3	5,61	5,15	4,95
Різниця між двома алгоритмами	13,5	14,2	14,5	14,7	14,95	15,1

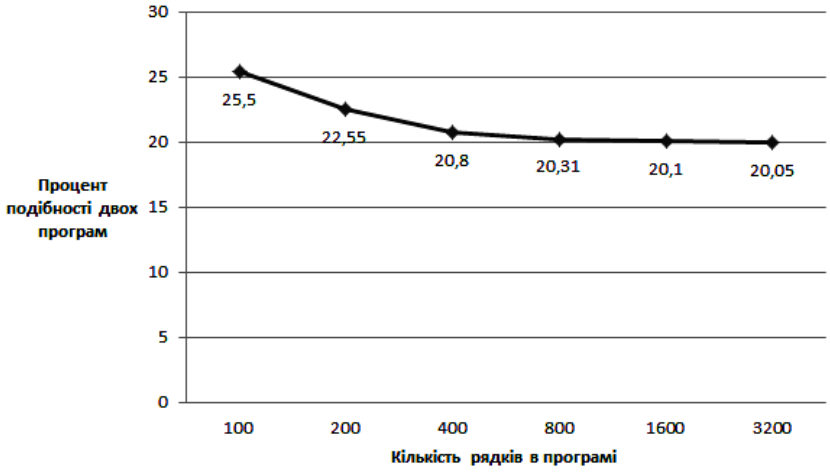

Рис. 2. Процент подібності без використання відсіювання

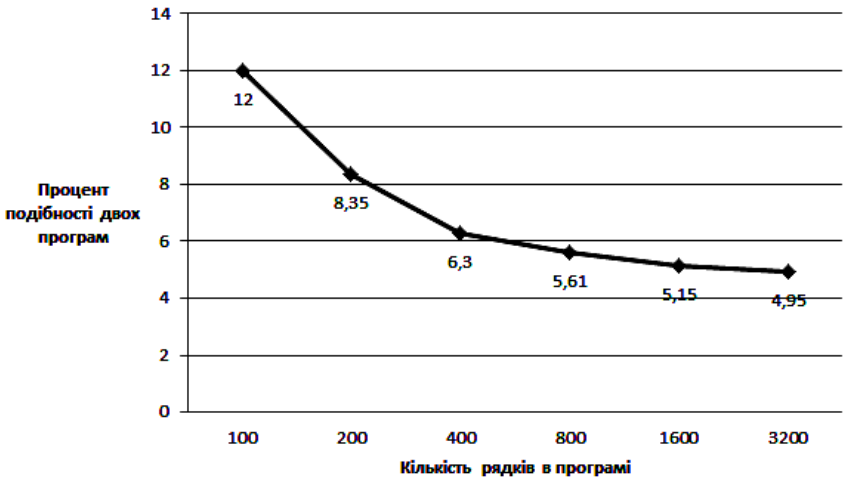

Рис. 3. Процент подібності з використанням відсіювання

Різниця між значеннями проценту подібності програмного коду для двох алгоритмів (рис.4) залежить від похибки виконаня алгоритму, і приблизно дорівнює одному проценту.

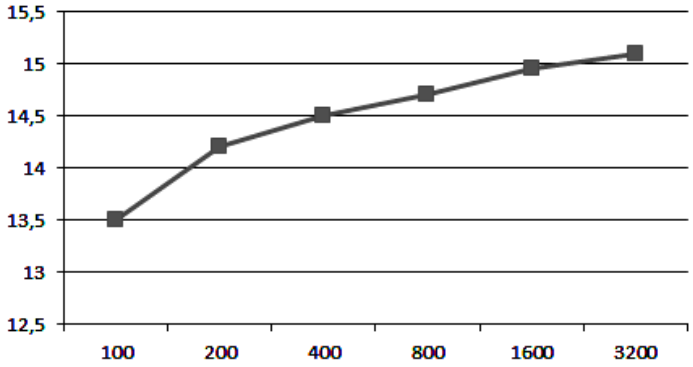

Рис. 4. Різниця між значенням подібності для двох алгоритмів

Проведемо експеримент швидкості виконання аналізу для тих самих програм. Один з проектів залишаємо без змін, в другому моделюємо деякі спроби приховати плагіат:

– заміна назв змінних, властивостей, методів, класів;

– зміна послідовності чергування методів (функцій), оголошення змінних та властивостей;

– зміна типів оголошення змінних.

Швидкість виконання алгоритмів наведено в таблиці 3.

Таблиця 3. Швидкість роботи алгоритмів, мс.

	100	200	400	800	1600	3200
Без використання відсіювання	40	71	120	228	452	948
З відсіюванням	86	149	267	429	710	1434
Час на фільтрацію	35	71	152	298	446	737

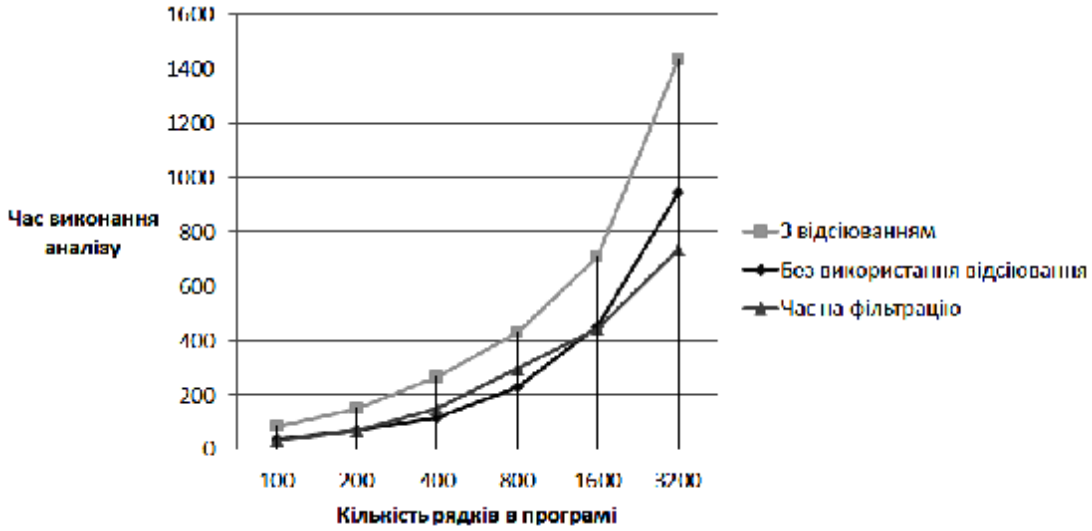

Рис. 5. Час виконання аналізу

Аналізуючи результати дослідження можна сказати, що запропонований підхід проводить відсіювання шаблонних ділянок коду і цим самим суттєво підвищує точність визначення подібності программи.

Натомість майже в два рази збільшився час аналізу програми. Час збільшився за рахунок виконання попередньої обробки токенізованого представлення для відсіювання автозгенерованого коду та приведення фрагментів коду до універсального вигляду. Слід зазначити, що час на порівняння проектів зменшився за рахунок суттєвого скорочення токенізованих представлень. Проте наведене порівняння є аналізом проектів один до одного. Очікується, що при виконанні порівняння проектів один до багатьох швидкість обробки збільшиться, бо зменшиться хешоване представлення програми до якої застосовується аналіз. Час на фільтрацію затрачується одноразово.

Висновки

В результаті проведеного дослідження запропоновано спосіб підвищення ефективності виявлення плагіату програмного коду *C#* на основі шаблонів. Розв'язана задача виявлення файлів шаблону і фрагментів формально перетвореного програмного коду, в розроблених програмних проектах, до застосування основних алгоритмів оцінки подібності. Отриманий алгоритм забезпечує виключення з проекту незмінних у різних проектах фрагментів коду, які є автозгенерованими, в результаті чого збільшена точність визначення наявності плагіату в програмному коді.

Подальшим розвитком є формалізація приведень структур циклів, що містять анонімні делегати, а також пошук інших шаблонів програмного коду, який може бути представлений у різному вигляді. Зараз виконуються роботи з пошуку методів збільшення швидкості порівняння проекту з базою проектів.

Список літератури

1. Baker B.S. On Finding Duplication and Near-Duplication in Large Software Systems // In Proceedings of the second IEEE Working Conference on Reverse Engineering (WCRE), July 1995. – P. 86–95.

2. А.А. Киричек, А.А. Амонс, Г.Г. Киричек. Алгоритм фильтрации для системы определения плагиата в программном коде // Вестник Национального технического университета «ХПИ». Серия: Новые решения в современных технологиях. – Харьков: ХНТУ «ХПИ», 2013. – С.78-82.

3. Амонс О.А., Зайцев С.Ю., Киричек О.О. Виявлення плагіату в програм-

ному коді C# // Вісник НТУУ «КПІ» 2011 р. – С.170-179.

4. L. Prechelt, G. Malpohl, and M. Philippsen. JPlag: Finding plagiarisms among a set of programs // Technical Report No. 1/00, University of Karlsruhe, Department of Informatics, March 2000.

5. Irving,R.W. Plagiarism and collusion detection using the Smith-Waterman algorithm // DCS Technical Report, Dept of Computing Science, University of Glasgow 2004. – pp 1-24

6. X. Chen, B. Francia, M. Li, B. McKinnon, A. Seker. Shared Information and Program Plagiarism Detection. // IEEE Trans. Information Theory, July 2004, 1545-1550

7. Neal R. Wagner. Plagiarism by Student Programmers [Электронный ресурс].- Режим доступа: http://www.cs.utsa.edu/~wagner/pubs/plagiarism0.html

8. Stepanova E.B. Krivtsov V.E. Process modeling in Educational IT-Projects // CSIT'2008: Proceedings of the 10-rd International Workshop on Computer Science and Information Technologies (Antalya, Turkey, September 15-17, 2008). – Ufa: Ufa State Aviation Technical University, 2008. – v. 1. – pp. 227-230.

9. Обзор автоматических детекторов плагиата в программах [Электронный ресурс].- Режим доступа: http://logic.pdmi.ras.ru/~yura/detector/survey.pdf

10. Dr Richard Li-Hua. From technology transfer to knowledge transfer – a study of international joint venture projects in China [Заглавие с экрана] - Режим доступа: http://knowledgemanagement.uk.net/resources/Richard Lihua paper.pdf

11. A. Aiken, S. Schleimer, D. Wikerson. Winnowing: local algorithms for document fingerprinting // In Pro ceedings of ACMSIGMOD Int. Conference on Management of Data, San Diego, CA, June 9–12, pp. 76–85. ACM Press, New York, USA, 2003.

12. Heckel, Paul. A Technique for Isolating Di erences Between Files // Communications of the ACM 21(4), pp. 264–268 (April 1978).

13. А.А. Киричек, А.А. Амонс, Г.Г. Киричек. Фильтрация шаблонов программного кода в студенческих проектах // III Всеукраинская научно-техническая конференция студентов, аспирантов и молодых ученых «Информационные управляющие системы и компьютерный мониторинг – 2013». 23-25 квіт. 2013 р.: тези доп. – Донецк: ДонНТУ, 2013. – С.36-42.

Статтю подано до редакції 04.12.2013