Detection of plagiarism in the programming code

SOTARO SUZUKI 317340

1. General description

In today's software development environment, programs encompass potentially millions of lines of code, distributed across hundreds of source files. The breadth of these projects can prove overwhelming, such that a single individual is unable to control or verify every aspect of the code. Consequently, emphasis is placed on the readability and clarity of the code to facilitate team collaboration and efficient understanding of the pre-existing segments. Tools that detect similarities in code fragments are invaluable in such a context, as they enable the team to assess the code quality, readability, and redundancies. Furthermore, they play an important role in identifying potential plagiarism within the project's code.

Even with their limitations, these tools substantially ease the burden of code verification by spotlighting similar parts rather than requiring a line-by-line examination of all files.

- Determining Plagiarism in Programming

Plagiarism is generally understood to be the act of copying someone else's work (or parts of it) and asserting authorship by obscuring the original source. This concept applies to a wide array of intellectual properties, including images, graphics, photos, songs, poems, academic papers, scientific publications, and even video games.

However, when applied to programming, the criteria for plagiarism becomes ambiguous. There isn't a defined threshold for the number of "similar lines of code" that would indicate whether a source file is a plagiarized version of another. In essence, even a single copied line of code could potentially indicate plagiarism.

Consider a scenario where a group of students are tasked with developing the same algorithm independently. If student A copies a single line of code from student B, it technically fits the definition of plagiarism. But what if the copied line is a common one like "return 0"? By that metric, nearly all works would be plagiarized. This illustrates that the threshold for programming plagiarism is not clearly defined, and relative measures are often utilized to quantify the extent of the plagiarism.

- Common Pitfalls

The process is prone to two types of errors:

- False positives: The algorithm incorrectly flags a condition as present when it is not.
- False negatives: The algorithm fails to detect a condition that is indeed present.

Additionally, a significant overlap between two codes (e.g., 80%) does not necessarily imply plagiarism. The context is crucial; for small programs or generic algorithms, a 60% similarity may not indicate plagiarism. Conversely, even a 10% similarity in complex programs, video games, or specialized algorithms can raise red flags.

Unlike natural languages, plagiarism detection in programming languages is relatively straightforward due to their rigid grammar and syntax. A simple phrase like "Al has a cat" could be expressed in countless ways in natural language, whereas code presents fewer possibilities.

1. Types of plagiarism algorithms

The algorithms used to detect plagiarism can be divided into several categories:

- strings
- tokens
- syntax trees
- dependency graphs
- metrics
- hybrid

1. Strings

Algorithms that check the exact match of character strings are very fast. They are very sensitive to changes in the compared texts, so you can easily trick them by renaming variables, changing the order of lines or changing one type of loop to another. On the other hand, this is the price you pay for the speed of execution of this algorithm

2. Tokens

These algorithms work very similarly to those working on strings of data. However, at an earlier stage, they convert selected strings into predefined structures. For example, all strings which consist of digits are marked as token - number. Thanks to this they are still very fast, however, they remain insensitive to changes in constant variables.

3. Syntax tree

Plagiarism algorithms that use syntax trees detect plagiarism at the syntactic level. Unlike algorithms that compare strings or tokens, they work on a lexicographic level. Syntactic trees detect similarities, e.g. between assignment instructions which, despite different variable names, mean the same thing. It's like comparing different versions of the phrase "Al has a cat" which in the end mean the same thing.Because operations on syntax trees are much more difficult than those on strings or tokens, these algorithms are much slower.

4. Metrics

Metrics are also called guild vectors. They are sets of numbers that characterize a given code, e.g. number of lines, bytes, loops, conditional instructions, variables etc. Metrics can be quickly calculated compared. Their disadvantage is the low precision of detecting similar code fragments

5. Hybrid

Hybrid algorithms combine the features of several of the algorithms described above.

2. Assumption

We set up following restrictions to the project:

- Only code written in C# 6 or earlier version is analysed
- Programs only use types provided in the base class library namespaces:

```
"System",
"System.IO",
"System.Net",
"System.Linq",
"System.Text",
"System.Text.RegularExpressions",
"System.Collections.Generic"
```

- Analysed code must compile either to executable or dynamically linked library

Restrictions given clearly lead us to finding plagiarism in programs written in C#.

3. Idea

We dedicated several hours to analyzing the entire lifecycle of a program from coding to execution. We especially examined the conversion of C# source code to Intermediate Language instructions. The structure of Intermediate Language code, intended for machine rather than human interpretation, enables easier analysis compared to human-written C#.

To illustrate this, consider the following code snippet:

```
using System;
public class Program
{
    public static int PowerRanger(int power, int min, int max)
    {
            int db=0;

            int i=1;

            while (Math.Pow((double) i, power)<=max) {
                if (Math.Pow((double) i, power)>=min) db++;
```

```
                i++;
            }
            return db;
    }
}
```

By performing transformation to IL we get:

```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
// .z\V.4..
  .ver 4:0:0:0
}
.assembly PowerRanger1.dll
{
  .custom instance void [mscorlib]System.Runtime.CompilerServices.Compi
lationRelaxationsAttribute::.ctor(int32) = ( 01 00 08 00 00 00 00 00 )
  .custom instance void [mscorlib]System.Runtime.CompilerServices.Runti
meCompatibilityAttribute::.ctor() = ( 01 00 01 00 54 02 16 57 72 61 70
4E 6F 6E 45 78   // ....T..WrapNonEx

                                    63 65 70 74 69 6F 6E 54 68 72 6F
77 73 01 )       // ceptionThrows.
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module PowerRanger1.dll.dll
// MVID: {68CF488B-4252-4A27-B7AA-8ED50643E58D}
.imagebase 0x10000000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
.class public auto ansi beforefieldinit Program
       extends [mscorlib]System.Object
{
  .method public hidebysig static int32  PowerRanger(int32 power,
                                                     int32 min,
                                                     int32 max) cil man
aged
  {
    // Code size       42 (0x2a)
    .maxstack  2
    .locals init (int32 V_0,
             int32 V_1)
    IL_0000:  ldc.i4.0
    IL_0001:  stloc.0
```

```
    IL_0002:  ldc.i4.1
    IL_0003:  stloc.1
    IL_0004:  br.s       IL_001b

    IL_0006:  ldloc.1
    IL_0007:  conv.r8
    IL_0008:  ldarg.0
    IL_0009:  conv.r8
    IL_000a:  call       float64 [mscorlib]System.Math::Pow(float64,
                                                            float64)
    IL_000f:  ldarg.1
    IL_0010:  conv.r8
    IL_0011:  blt.un.s   IL_0017

    IL_0013:  ldloc.0
    IL_0014:  ldc.i4.1
    IL_0015:  add.ovf
    IL_0016:  stloc.0
    IL_0017:  ldloc.1
    IL_0018:  ldc.i4.1
    IL_0019:  add.ovf
    IL_001a:  stloc.1
    IL_001b:  ldloc.1
    IL_001c:  conv.r8
    IL_001d:  ldarg.0
    IL_001e:  conv.r8
    IL_001f:  call       float64 [mscorlib]System.Math::Pow(float64,
                                                            float64)
    IL_0024:  ldarg.2
    IL_0025:  conv.r8
    IL_0026:  ble.s      IL_0006

    IL_0028:  ldloc.0
    IL_0029:  ret
  }
  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  } // end of method Program::.ctor

}
```

We have found out that what's important is the flow in the program i.e. logic behind data processing. Therefore we have stripped everything unrelated to actual algothitm behaviour. We have ended up with list of instructions (below only part of it is shown to indicate how whole data looks like):

```
ldc.i4.0
stloc.0
ldc.i4.1
stloc.1
br.s
ldloc.1
conv.r8
ldarg.0…
```

Afterwards, we applied different kinds of string similarity measures (some of which were adjusted to better work with transformed IL code). The result is a program that is able to tell whether the two source codes are similar, even if they were altered in a way to deceive human evaluator.

Detail of Methodology

Our program is designed to take two C# source code files as input and generate a similarity score based on multiple metrics. The process consists of four main steps: Compilation, Decompilation to IL, Cleansing, and Comparison.

- Compilation and Decompilation

Initially, the input source code files are compiled into binaries. The binaries are then decompiled to their IL representation. This step neutralizes any stylistic differences in the code and leaves only the structural and logical similarities intact.

- Cleansing

The IL code is then 'cleansed' to filter out unnecessary information and highlight the key components of the code structure. Specifically, only the lines containing 'IL_' are retained, and extraneous information is removed, leaving only the IL instruction set.

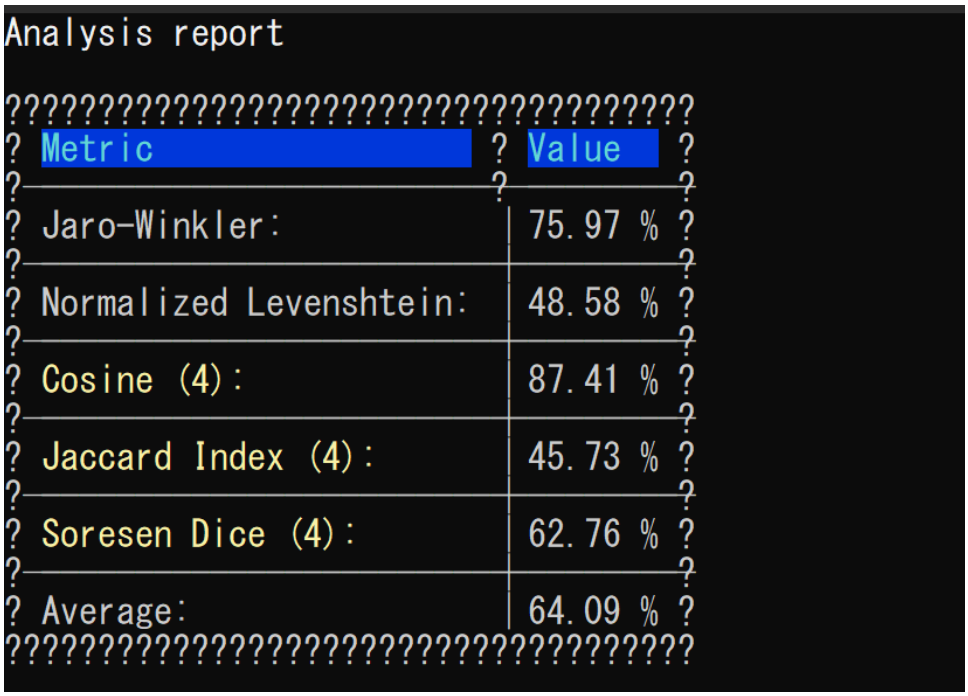
- Comparison

Finally, the cleansed IL code of both files is compared using several string similarity metrics, including Jaro-Winkler, Normalized Levenshtein, Cosine similarity, Jaccard index, and Sorensen Dice. Each metric offers a different perspective on the similarity of the two code files. An average of these metrics is then computed to provide a comprehensive similarity score.

- Results and Discussion

Our program generates a console report, providing the calculated similarity percentages for each metric and the average percentage. By using IL code and

multiple similarity metrics, Our program provides an effective and fair comparison that could detect sophisticated plagiarism attempts.



Conclusion

Our program offers a effective method to detect programming plagiarism by analyzing the IL representation of C# code and utilizing various string similarity metrics. Future work may focus on extending this approach to other programming languages and improving efficiency.