CSCI 729 Final Report
Naive implementation of GPLAG[1]

Nicolas Montanaro Bernard Cecchini

1 Goal iterations

1.1 Original project description

The paper that we have chosen is categorized under Graph Alignment, and involves GRAAL (GRAph ALigner) Algorithms. After we initially go through and break down various elements within the paper, we will be able to better summarize and report on how the paper will correspond with our project research and implementations. Our project will consist of two main elements pertaining to Graph Alignment. One element will be using SourcererCC, and the other will be using a current RIT project that incorporates various graph alignment techniques to analyze code segments.

Our research and initial project will consist of understanding how each of these pieces of software operates, as well as targeting the specific alignment properties of each technology. We would also like to agree on a couple of different small programs, or segments of code that we will be able to successfully run through each software and obtain results. Once we have enough examples to generate results from each technology, we would like to analyze these results for similarities pertaining to alignment. By the end of the project, we would like to be able to draw and present strong conclusions based on enough sample data, to adequately represent similarities as well as differences in alignment properties pertaining to SourcererCC and the RIT GRAAL based project.

1.2 Shift of focus

Initially our focus was set on the implementation of GPLAG, followed by the comparison to SourcererCC and GRAAL. Before we dove into the actual implementation of GPLAG we noticed that our initial idea of comparison to SourcererCC was most likely not going to work out. This is due to a few reasons, one of the reasons being the poor documentation for SourcererCC. This poor documentation fails to specify how to interpret the results obtained from running SourcererCC, it also fails to specify the input file types and formats. Because of this ambiguity, we were unable to isolate a specific file format, as well as unable to obtain the results from SourcererCC necessary to be able to make a meaningful comparison between SourcererCC and GPLAG. After arriving at this conclusion, we began working our way through our GPLAG implementation.

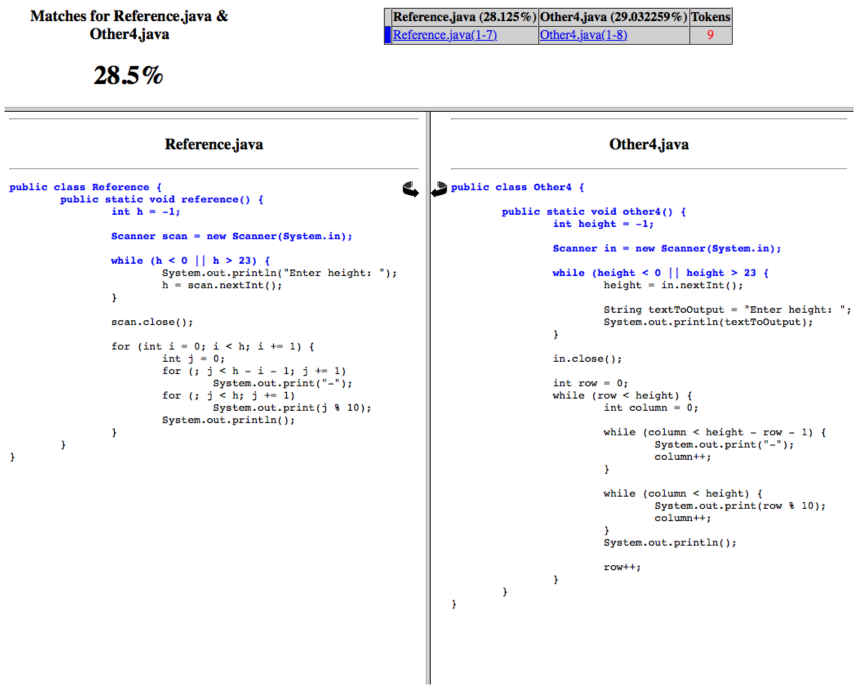
At this point, we encountered various issues that brought up concerns, which we then attempted to resolve. Ultimately some of the issues that we encountered slowed our overall implementation progress, and because of this, we ended up finalizing our implementation at a much late point in time than we had originally anticipated. We will examine more of the specifics of our implementation in the following sections.

2 GPLAG

2.1 Current solutions vs. GPLAG

2.1.1 Tokenization-based Approaches to Clone Detection

JPlag[2] has existed since 2001 and uses a tokenization-based approach to detecting plagiarism between source code files. During tokenization input files are parsed and converted into token strings. Tokenization for three languages are supported: Java, Scheme, and C++. Figure [?] shows the visual representation of identical tokens detected between two source files in JPlag.



Matches for Reference.java & Other4.java

28.5%

Reference.java (28.125%)	Other4.java (29.032259%)	Tokens
Reference.java(1-7)	Other4.java(1-8)	9

Reference.java

```
public class Reference {
        public static void reference() {
                int h = -1;

                Scanner scan = new Scanner(System.in);

                while (h < 0 || h > 23) {
                        System.out.println("Enter height: ");
                        h = scan.nextInt();
                }

                scan.close();

                for (int i = 0; i < h; i += 1) {
                        int j = 0;
                        for (; j < h - i - 1; j += 1)
                                System.out.print("-");
                        for (; j < h; j += 1)
                                System.out.print(j % 10);
                        System.out.println();
                }
        }
}
```

Other4.java

```
public class Other4 {

        public static void other4() {
                int height = -1;

                Scanner in = new Scanner(System.in);

                while (height < 0 || height > 23 {
                        height = in.nextInt();

                        String textToOutput = "Enter height: ";
                        System.out.println(textToOutput);
                }

                in.close();

                int row = 0;
                while (row < height) {
                        int column = 0;

                        while (column < height - row - 1) {
                                System.out.print("-");
                                column++;
                        }

                        while (column < height) {
                                System.out.print(row % 10);
                                column++;
                        }
                        System.out.println();

                        row++;
                }
        }
}
```

Figure 1: JPlag's visual clone detection output

2.1.2 Problems with Tokenization Approach

The tokenization-based approach to detecting code clones is currently the most popular. Several well-known and widely-used tools exist like JPlag, SourcererCC, and conQAT[3]. They all deal with the tokenized data in different ways but they share one thing in common: tokenization is performed as a pre-processing step, and tokens are the data structure used to detect clones in the code. The primary problem with this approach is that it does not deal with the *semantic* similarities of source files, just the *syntactic* similarities. Examining the common techniques used by plagiarists, as outlined in the GPLAG paper[1], can help us understand the types of differences present in code. The five categories of plagiarism disguises are:

1. **Format alteration**, in which items typically discarded by the compiler such as whitespace and comments are changed.

2. **Identifier renaming**, in which variables or function names are changed.

3. **Statement reordering**, in which snippets of code that are not dependent on previous code are moved around. Most commonly, location of function definitions in source files are changed.

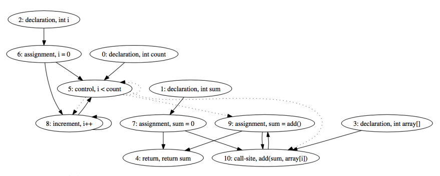
4. **Control replacement**, in which a control structure is replaced with a logically equivalent one. For example, a `for` loop being replace by an equivalent `while` loop, or `if` statements being replaced by logically equivalent checks that evaluate the same way.

5. **Code insertion**, in which additional code not present in the original is inserted into the clone. Can't alter the logic structure of the original.

Tokenization is able to detect renamed variables and functions well because they rely on the types being mapped to the tokens rather than relying purely on the content of the strings. However, tokenization isn't able to capture differences in control flow when modified - `for` loops replaced with `while` loops won't be detected because they will be tokenized differently. Similarly, code insertion and statement reordering will also break tokenization in many cases.

2.2 Program Dependency Graph Approach

An approach that forgoes all of the issues associated with tokenization for detecting code clones is using program dependence graphs. Figure 2 shows an example of converting source code to a PDG, taken from the GPLAG paper.



(a) Program Dependence Graph of the Procedure `sum`

```
int sum(int array[], int count)
{
  int i, sum;
  sum = 0;
  for(i = 0; i < count; i++){
    sum = add(sum, array[i]);
  }
  return sum;
}

int add(int a, int b)
{
  return a + b;
}
```

(b) Summation over an Array

Figure 2: Example source code and generated PDG

When we convert our source files to a program dependence graph the problem of code detection becomes synonymous with the problem of subgraph isomorphism detection. Although this is a known NP-complete problem the GPLAG algorithm uses two types of filtering to reduce the search space and allow us to find isomorphs in a reasonable amount of time. They are as follows:

1. **Lossless filtering**: we ignore subgraph matches of size smaller than K. Additionally, all PDG pairs that are not $\gamma-$isomorphic are discarded. The definition of $\gamma-$isomorphism is provided in [1].

2. **Lossy filtering**: sufficiently dissimilar PDG pairs are discarded. The details of lossless filtering are a bit complex but the details be found in [1].

The pseudocode for GPLAG's algorithm with my additional annotation can be seen in Figure 3. The algorithm requires quite a bit of pre-processing in that the PDGs need to be generated in advance. The authors of the paper use CodeSurfer [1] for PDG generation which we do not have access to. However, the Graph Alignment implementation we have access to includes basic generation of PDGs.

Algorithm 1 GPLAG($\mathcal{P}$, $\mathcal{P}'$, K, γ, α)

```
Input: $\mathcal{P}$: The original program
       $\mathcal{P}'$: A plagiarism suspect
       $K$: Minimum size of nontrivial PDGs, default 10
       $\gamma$: Mature rate in isomorphism testing, default 0.9
       $\alpha$: Significance level in lossy filter, default 0.05
Output: $\mathcal{F}$: PDG pairs regarded to involve plagiarism

1: $\mathcal{G}$ = The set of PDGs from $\mathcal{P}$            Pre-generated PDGs
2: $\mathcal{G}'$ = The set of PDGs from $\mathcal{P}'$          Paper used CodeSurfer
3: $\mathcal{G}_K = \{g | g \in \mathcal{G}$ and $|g| > K\}$          Discard PDGs below certain size
4: $\mathcal{G}'_K = \{g' | g' \in \mathcal{G}$ and $|g'| > K\}$       Easy search space reduction
5: for each $g \in \mathcal{G}_K$
6:     let $\mathcal{G}'_{K,g} = \{g' | g' \in \mathcal{G}'_K, |g'| \geq \gamma |g|, (g, g')$ passes filter$\}$
7:     for each $g' \in \mathcal{G}'_{K,g}$
8:         if $g$ is $\gamma$-isomorphic to $g'$          Implemented in C++
9:             $\mathcal{F} = \mathcal{F} \cup (g, g')$          Used VFLib
10: return $\mathcal{F}$;
```

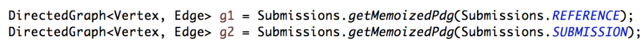
Figure 3: Example source code and generated PDG

[1]http://grammatech.com

2.3 Initial hurdles

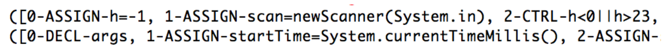
We encountered many issues while attempting to implement GPLAG as described in the paper. The foundation of any filtering or algorithmic-based results is dependent upon the ability to analyze as well as perform computations on a PDG.

The creation of a proper PDG is critical, and not a quick process to implement from scratch. As stated previously the authors of the GPLAG paper used a commercially available tool that we did not have access to. It likely produces PDGs in a very specific way that may allow for greater optimization of the algorithm. Instead, the following figures show PDG generation via the GRAAL implementation of PDG generation.



```
DirectedGraph<Vertex, Edge> g1 = Submissions.getMemoizedPdg(Submissions.REFERENCE);
DirectedGraph<Vertex, Edge> g2 = Submissions.getMemoizedPdg(Submissions.SUBMISSION);
```

(a) PDG Creation Java

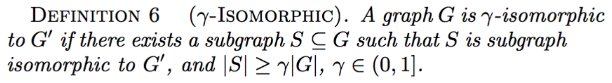


```
([0-ASSIGN-h=-1, 1-ASSIGN-scan=newScanner(System.in), 2-CTRL-h<0||h>23,
([0-DECL-args, 1-ASSIGN-startTime=System.currentTimeMillis(), 2-ASSIGN-
```

(b) PDG toString

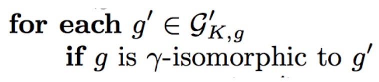
Figure 4: PDG creation in Java and its partial toString representation.

The second issue that we encountered, was that there was a disconnect with definitions in the GPLAG paper. This disconnect specifically had to do with the definition of $\gamma-$isomorphism. It appears that in the paper, $\gamma-$isomorphism is defined correctly, however it is notated incorrectly in the algorithm. In the definition of $\gamma-$isomorphism, it refers to how a graph G can be $\gamma-$isomorphic to another graph G'. This is what we implemented in our version of GPLAG, but in the algorithm, they notate that one needs to check if a node in G is $\gamma-$isomorphic to a node in G'. This is simply not possible by the definition given in the beginning of the paper. This was a conceptual hurdle for us to overcome before we could move on to our implementation. Figure 5 shows represents this disconnect.



DEFINITION 6 (γ-ISOMORPHIC). *A graph G is γ-isomorphic to G' if there exists a subgraph $S \subseteq G$ such that S is subgraph isomorphic to G', and $|S| \geq \gamma|G|$, $\gamma \in (0, 1]$.*

(a) $\gamma-$isomorphic Definition



```
for each $g' \in \mathcal{G}'_{K,g}$
    if $g$ is $\gamma$-isomorphic to $g'$
```

(b) $\gamma-$isomorphism Check

Figure 5: Disconnect between the $\gamma-$isomorphic definition and check.

For time reasons, we wanted to implement the most basic version of GPLAG possible, so we eliminated all of the filtering from the algorithm. The reason for this is because the filtering only optimizes performance by cutting the search space. We wanted to see some result from GPLAG even if it was not obtained in the most optimal way. It is also important to note that the authors from the paper, can contribute much of their success to heavy preprocessing, primarily filtering the search space prior to running the algorithm.

2.4 Implementation

Algorithm 1 GPLAG(D, Q, γ, t)

```
Input  : D: The Data PDG
         Q: The Query PDG
         γ: The gamma threshold
         t: The percentage plagiarized threshold
Output: true: plagiarism detected
        false: plagiarism not detected

Q' = Q \{v};
if |Q'| ≥ γ |D| then
    if SubgraphMatching(D, Q', t) then
        return true
    else
        GPLAG(D, Q', γ, t)
    end
else
    return false
end
```

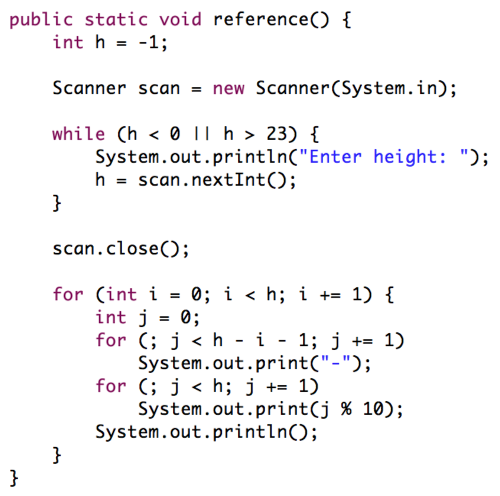
The GPLAG algorithm is a naive version of the algorithm described in [1]. The inputs are a data and query PDG which are represented using the JGraphT library. γ is the threshold proposed in [1] that effectively reduces our search space by only checking for subgraph matches when $|Q'| \geq \gamma|D|$. t is the threshold to perform $SubgraphMatching$ with. If a subgraph is found but $|solution| < t$, the match is ignored. This prevents us from considering trivial subgraph matches as candidates for plagiarism.

$SubgraphMatching$ performs subgraph matching, taking into account the threshold parameter to determine a specified level of plagiarism. This is represented by the following check which is run from within the function.

Algorithm 2 PlagiarismCheck

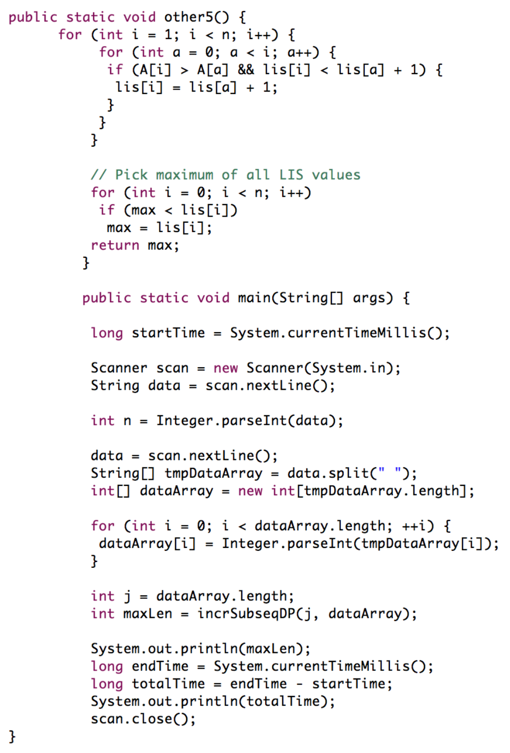
```
Input  : s: The solution space
         q: The Query PDG
         t: The percentage plagiarized threshold

if (|s| / |q|) ≥ t then
    return true
end
```

Once we were conceptually able to construct pseudo code, and successfully able to generate our PDGs, we were able to implement the entire procedure in Java. Figure 6 below shows two sample input programs for the Reference and Submission programs respectively. These programs are not plagiarized, we would expect to see our GPLAG implementation return this observation. The results of our GPLAG implementation with the two sample programs from Figure 6, are represented in Figure 7.



```
public static void reference() {
    int h = -1;

    Scanner scan = new Scanner(System.in);

    while (h < 0 || h > 23) {
        System.out.println("Enter height: ");
        h = scan.nextInt();
    }

    scan.close();

    for (int i = 0; i < h; i += 1) {
        int j = 0;
        for (; j < h - i - 1; j += 1)
            System.out.print("-");
        for (; j < h; j += 1)
            System.out.print(j % 10);
        System.out.println();
    }
}
```

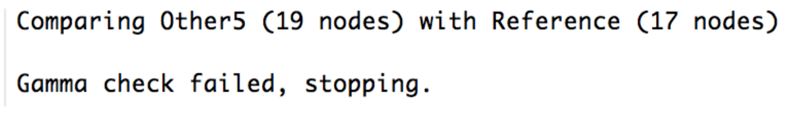
(a) The Reference source file



```
public static void other5() {
      for (int i = 1; i < n; i++) {
           for (int a = 0; a < i; a++) {
            if (A[i] > A[a] && lis[i] < lis[a] + 1) {
             lis[i] = lis[a] + 1;
            }
           }
          }

          // Pick maximum of all LIS values
          for (int i = 0; i < n; i++)
           if (max < lis[i])
            max = lis[i];
          return max;
         }

         public static void main(String[] args) {

          long startTime = System.currentTimeMillis();

          Scanner scan = new Scanner(System.in);
          String data = scan.nextLine();

          int n = Integer.parseInt(data);

          data = scan.nextLine();
          String[] tmpDataArray = data.split(" ");
          int[] dataArray = new int[tmpDataArray.length];

          for (int i = 0; i < dataArray.length; ++i) {
           dataArray[i] = Integer.parseInt(tmpDataArray[i]);
          }

          int j = dataArray.length;
          int maxLen = incrSubseqDP(j, dataArray);

          System.out.println(maxLen);
          long endTime = System.currentTimeMillis();
          long totalTime = endTime - startTime;
          System.out.println(totalTime);
          scan.close();
}
```

(b) The Submission source file

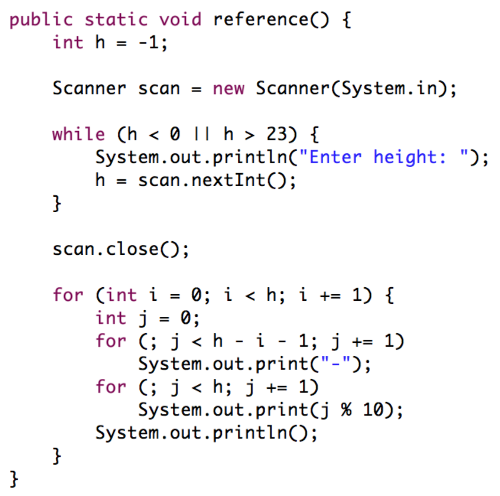
Figure 6: Both the Reference and Submission source code for showing no plagiarism.



```
Comparing Other5 (19 nodes) with Reference (17 nodes)

Gamma check failed, stopping.
```

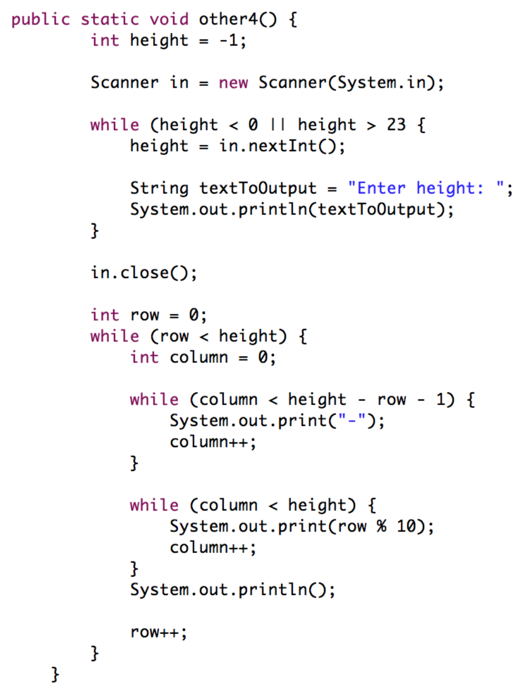
Figure 7: Output of GPLAG execution with Figure 6 sample programs as input.

If the gamma check fails, we know that plagiarism was not detected up until this point, and the search was cut because plagiarism will not be detected from this point forward. Therefore, if the gamma check fails, the Submission source code was not plagiarized. The follow figures show two sample input programs that are plagiarized, and the results from running our GPLAG implementation given these programs as input.



```java
public static void reference() {
    int h = -1;

    Scanner scan = new Scanner(System.in);

    while (h < 0 || h > 23) {
        System.out.println("Enter height: ");
        h = scan.nextInt();
    }

    scan.close();

    for (int i = 0; i < h; i += 1) {
        int j = 0;
        for (; j < h - i - 1; j += 1)
            System.out.print("-");
        for (; j < h; j += 1)
            System.out.print(j % 10);
        System.out.println();
    }
}
```

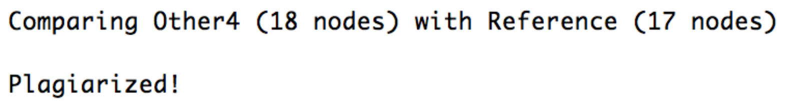
(a) The Reference source file



```java
public static void other4() {
        int height = -1;

        Scanner in = new Scanner(System.in);

        while (height < 0 || height > 23 {
            height = in.nextInt();

            String textToOutput = "Enter height: ";
            System.out.println(textToOutput);
        }

        in.close();

        int row = 0;
        while (row < height) {
            int column = 0;

            while (column < height - row - 1) {
                System.out.print("-");
                column++;
            }

            while (column < height) {
                System.out.print(row % 10);
                column++;
            }
            System.out.println();

            row++;
        }
    }
```

(b) The Submission source file

Figure 8: Both the Reference and Submission source code for detecting plagiarism.



```
Comparing Other4 (18 nodes) with Reference (17 nodes)

Plagiarized!
```

Figure 9: Output of GPLAG execution with Figure 12 sample programs as input.

2.4.1 Caveats & improvements

The original paper has an additional step of further reducing the search space of the algorithm by way of what they call **lossless** and **lossy** filtering. The implementation we used did not use either of these. Lossless filtering would be easy to implement going forward since it is simply ignoring subgraph matches of size smaller than some value K. Lossy filtering is much more complex and does not seem like it would offer significant benefit except on very large query and data graphs.

Perhaps the largest improvement that could be made is a more intelligent method of vertex removal in the creation of Q'. Because this implementation is randomly removing a vertex v to generate Q', there is the possibility of removing a vertex that makes the graph disconnected, or removing vertices that would otherwise be in the solution space.

2.5 Dataset

The dataset these comparisons were performed was unfortunately quite small. The PDG generation tool used in the GPLAG implementation was ported from the GRAAL implementation. It is highly dependent on the source files being structured in a particular way.

Since we were not able to use a real-world dataset, 8 source files were used with one being considered the "Reference" file. One file is completely different both semantically and syntactically from the reference file and should not be detected as plagiarized. The other files are either partially or largely plagiarized and should be detected as having plagiarized code present to the point where they are not considered original.

2.6 Test results

To test the GPLAG implementation, source files were compared to a reference file and a positive or negative output was counted. Because the implementation randomly removes nodes at every recursive step false positives are occasionally detected even when the source file is completely different than the reference. This is an expected behavior and can only be solved by determining a way to remove leaf nodes only, and avoid the removal of nodes that are likely in the solution space.

Testing with unplagiarized source			
Trial	**FP**	**N**	**% incorrect**
1	58	942	5.8
2	60	940	6
3	74	926	7.4
4	44	956	4.4
5	46	954	4.6
6	56	946	5.6
7	51	949	5.1
8	55	945	5.5
9	53	947	5.3
10	55	945	5.5

Table 1: Results of running GPLAG with completely different source and reference files. **FP** = false positives, **N** = negatives. % incorrect calculated by dividing false positives by 1000.

Table 1 shows the results of running GPLAG with a completely different source and reference file. Ten trials were performed. Each trial consisted of running GPLAG on the source file 1000 times with γ set to 0.8 and the threshold set to 0.9. Since the source file is in no way similar to the reference file, ideally the number of false positives for each trial would be zero. Because of the random removal of nodes we instead see the percentage of false positives to average $\approx 5.5\%$.

When the source file was semantically very similar to the reference file and therefore clearly plagiarized the false positive rate drops to 0%. The code was correctly detected as plagiarized all 1000 times for every trial.

Since this implementation of GPLAG randomly removes nodes the only way to accurately detect if a particular file is plagiarized when compared to the source would be to run the algorithm several times with the same input files and mark the source as plagiarized if above a certain threshold. While not ideal, this has proven to be effective, and since the percentage incorrect is relatively low it doesn't pose too much of a problem.

The most glaring issue with this implementation is slow running time. Since there is always a chance of detecting a false positive the only way to accurately categorize a particular file as plagiarized or not is to run the algorithm multiple times over the same files. This greatly increases the time it takes for the algorithm to complete running and generate output that can be trusted. Again, it should be stressed that this is a problem with the implementation performed for this study, not with the original algorithm presented in the paper.

References

[1] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. 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*, KDD '06, pages 872–881, New York, NY, USA, 2006. ACM.

[2] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with jplag. 8(11):1016–1038, nov 2002.

[3] CQSE GmbH. conQAT Overview. *CQSE GmbH Website*, 2017. https://www.cqse.eu/en/products/conqat/overview/.