Final Report

Implementation of GPLAG[1] and comparison versus widely available tools for plagiarism detection in student code submissions

Nicolas Montanaro

1 Initial proposal

1.1 Summary of initial proposal

With the increasing popularity of the computer science field and related disciplines, we're observing that enrollment in STEM majors for universities across the country is increasing rapidly year after year. Many of these fields of study require some knowledge of computer programming. Foundational programming knowledge is not exclusive to computer scientists and software engineers - future electrical engineers, bioinformaticists, mathematicians, 3D artists amongst others all benefit greatly from basic programming knowledge. In order to meet this growing demand, the popularity of MOOCs (massive open online courses) has drastically increased. Websites like Coursera offer a plethora of courses - often developed at major universities - for free to people online. How do we enable these students to learn effectively and receive helpful feedback for a topic that is, for many, very difficult? By examining the related problem of code cloning, we may be able to find a solution. The code cloning problem deals with the detection of similar patterns of code. Imagine for a moment we copy a particular function from one piece of code across multiple others. Each time we copy it, it is modified slightly to fit this particular usage. However, we later discover that our original algorithm - the one we copied several times - has an error. How can we detect the code we copied, and is there a way we can make our changes to the original block of code but have it modify all of the other instances? Using a combination of existing techniques to detect code cloning in combination with relatively new techniques of subgraph alignment applied to analyzing student submissions for introductory computer programming assignments, can we effectively create a system that allows for graders to easily grade hundreds of submissions at once? By using techniques that compare Program Dependency Graphs representing multiple student submissions against one solution in combination with graph alignment techniques, we should be able to determine the effectiveness of detecting errors in student submissions and providing meaningful feedback to many students at once.

1.2 Modifications to initial proposal

The initial proposal focused largely on providing personalized feedback for students taking online programming MOOCs. This would have been done using an existing method of subgraph matching using Program Dependence Graphs and graph alignment.

It was eventually determined, as we'll see, that through some explorations as part of compiling the milestones that this goal wasn't particularly useful. A fully implemented personalized student feedback system would have been fantastic but was too broad and lofty a goal for the scope of this independent study. Instead, I decided to focus on the filter-less im-

plementation of the GPLAG[1] algorithm and how it compares to the more commonly used JPlag[2] and conQAT[3] tools for code clone detection, specifically for the use case of detecting plagiarized code in student submissions. The functionality of detecting "plagiarized" pieces of code can easily be extended to providing personalized feedback.

2 Major milestones

2.1 SourcererCC investigation

2.1.1 Introduction

For the first milestone of this independent study current code-cloning detection algorithms and software were investigated. The state-of-the-art for code cloning detection is SourcererCC [4]. Despite being the leading tool for code-cloning detection, the available documentation [5] for practical running of the project and analysis of the results is somewhat sparse. The purpose of this milestone was to create a simple to use repository containing all necessary components to run SourcererCC on our datasets. Initial runs of the program using the datasets we've compiled previously have shown mixed results.

2.1.2 Algorithm

SourcererCC's proposal paper provides evidence enough that it is the cutting-edge of code-cloning techniques. It was tested against four other publicly available tools: CCFinderX [6], Deckard [7], iClones [8], and NiCad [9]. CCFinderX is cited as being a popular and reasonably effective tool to detect identical fragments of code minus whitespaces, tabs, and comments, and difference in identifier names and literal values. The other three tools are used for comparison using code that is syntactically similar in addition to all of the previous constraints. Each one of these tools takes quite a different approach to detecting clones and is used for different types of clone detection. Figure 1, taken from [4], shows SourcererCC's detection process.

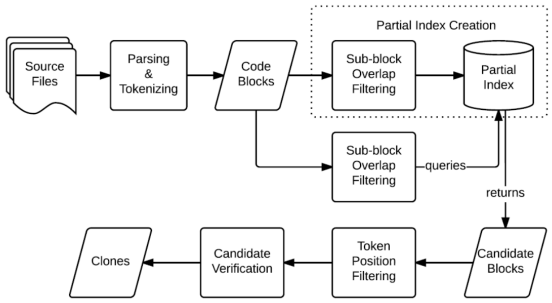


Figure 1: SourcererCC's clone detection process

SourcererCC's method of detecting code clones is actually functionally very similar to other widely-available tools, as we'll see shortly. The two primary steps necessary to perform detection in the tool itself are:

1. **Tokenization**, during which input files are parsed and converted into token strings. Tokenization for three languages are supported: Java, Scheme, and C++.

2. **Greedy String Tiling**, the algorithm for which is shown in Figure 4, during which each token string is attempted to be "covered" by another substring from a different file as well as possible. Percentage coverage is used as the similarity metric between the source files.

2.1.3 Results

The datasets we have previously compiled for use with studying graph alignment techniques were used for testing the capabilities of SourcererCC. The data is Java solutions for code problems posted on `CodeChef.com`. Figures 2 and 3 show the results of running the tool on two of our datasets with a threshold of 80% similarity.

378	163	0	0	553	0	0	0	index	11
0	47	0	368	2629	2407	0	0	search	
501	295	0	0	816	0	0	0	index	13
0	54	0	579	3551	3326	0	0	search	
450	198	0	0	669	0	0	0	index	8
0	52	0	368	2582	2362	0	0	search	

Figure 2: CSV output of running SCC with RGAME dataset

378	163	0	0	553	0	0	0	index	11
0	47	0	368	2629	2407	0	0	search	
501	295	0	0	816	0	0	0	index	13
0	54	0	579	3551	3326	0	0	search	

Figure 3: CSV output of running SCC with CIELRCPT dataset

The meaning of these outputs are not specified anywhere in the documentation for the tool, nor in the paper introducing the tool. In addition, while the two tables have a different number of rows, the first four rows are identical in both instances. This suggests the tool is not working properly with the datasets being passed to it. A number of different parameters and runs were performed, all with the same results. Even lowering the threshold to 0 did not change the results, and the results we are getting cannot be properly interpreted.

2.1.4 Post-mortem

This milestone was primarily helpful in figuring out what direction to avoid going in. The source code for SourcererCC is very complex and almost no documentation is provided. The paper focuses mostly on performance rather than usage. Even though the GitHub page[5] provides documentation for running the tool - which I was able to do successfully - there is no outline for how to interpret the results. Additionally, there is no information given while the tool is being run, or specification for how the input files need to be formatted. Based on the results of this milestone and conversations with Prof. Rivero I decided to head in a different direction.

2.2 Current solutions vs. GPLAG

2.2.1 Introduction

The goal of the second milestone of this indepdendent study was to specify the end goal of the project, familiarize myself with a number of other existing tools used for code-clone detection, learn about possible improvements to the current state-of-the-art, and begin implementation of an alternative. The examined existing tools are JPlag [2] and conQAT [3] which both use tokenization for plagiarism detection. Next, I learned about GPLAG [1] and how it related to graph-alignment and subgraph matching techniques to detect plagiarism, whether the code samples be semantically or syntactically similar.

2.2.2 JPlag & conQAT: Tokenization-based Approaches to Clone Detection

JPlag has existed since 2001 and uses a tokenization-based approach to detecting plagiarism between source code files. The two steps performed for this are:

1. **Tokenization**, during which input files are parsed and converted into token strings. Tokenization for three languages are supported: Java, Scheme, and C++.

2. **Greedy String Tiling**, the algorithm for which is shown in Figure 4, during which each token string is attempted to be "covered" by another substring from

a different file as well as possible. Percentage coverage is used as the similarity metric between the source files.

```
0   Greedy-String-Tiling(String A, String B) {
1       tiles = {};
2       do {
3           maxmatch = M;
4           matches = {};
5           Forall unmarked tokens A_a in A {
6               Forall unmarked tokens B_b in B {
7                   j = 0;
8                   while (A_{a+j} == B_{b+j} &&
9                           unmarked(A_{a+j}) && unmarked(B_{b+j}))
10                      j++;
11                  if (j == maxmatch)
12                      matches = matches ⊕ match(a, b, j);
13                  else if (j > maxmatch) {
14                      matches = {match(a, b, j)};
15                      maxmatch = j;
16                  }
17              }
18          }
19          Forall match(a, b, maxmatch) ∈ matches {
20              For j = 0 ... (maxmatch − 1) {
21                  mark(A_{a+j});
22                  mark(B_{b+j});
23              }
24              tiles = tiles ∪ match(a, b, maxmatch);
25          }
26      } while (maxmatch > M);
27      return tiles;
28  }
```

Figure 4: Greedy string tiling pseudocode

These two steps are actually almost identical to SourcererCC's process of code clone detection - first tokenization, followed by comparison of tokens between source files. The primary difference is that SourcererCC allows for custom tokenization files to be input making it language independent.

conQAT is comparatively much younger having been released in 2013. While similar to JPlag in that it tokenizes the source code it instead uses token comparison to a generated suffix-tree. This is actually the same method used by another clone detection tool, iClones [8]. We can see an example output provided on the conQAT website of a detected clone in Figure 5. This is similar to the visual output provided by JPlag as can be seen in Figure 6.

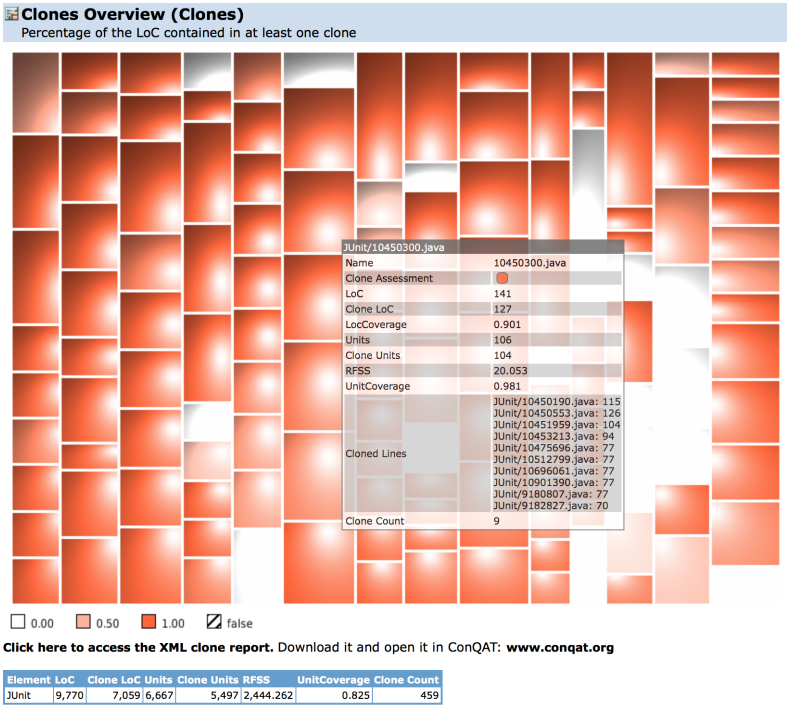


Figure 5: conQAT's visual clone detection output

Figure 6: JPlag's visual clone detection output

2.2.3 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. 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. This, in turn, will allow for more semantic matching to be done using graph-alignment techniques to provide student feedback for MOOC programming classes. 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.

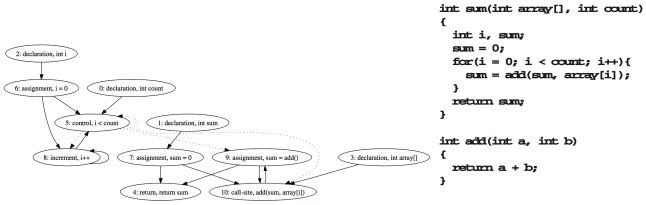
4. **Control replacement**, in which a control structure is replaced with a logically equivalent one. For example, a `for` loop being replace by an equivalent `while` loop, or `if` statements being replaced by logically equivalent checks that evaluate the same way.

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.

2.2.4 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 7 shows an example of converting source code to a PDG, taken from the GPLAG paper.



(a) Program Dependence Graph of the Procedure `sum` (b) Summation over an Array

Figure 7: 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 8. 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
γ: Mature rate in isomorphism testing, default 0.9
α: 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: $\mathcal{G}_K = \{g | g \in \mathcal{G} \text{ and } |g| > K\}$ — Discard PDGs below certain size
4: $\mathcal{G}'_K = \{g' | g' \in \mathcal{G} \text{ and } |g'| > K\}$ — Easy search space reduction
5: **for each** $g \in \mathcal{G}_K$
6: **let** $\mathcal{G}'_{K,g} = \{g' | g' \in \mathcal{G}'_K, |g'| \geq \gamma|g|, (g, g') \text{ passes filter}\}$
7: **for each** $g' \in \mathcal{G}'_{K,g}$
8: **if** g is γ-isomorphic to g' — Implemented in C++
9: $\mathcal{F} = \mathcal{F} \cup (g, g')$ — Used VFLib
10: **return** $\mathcal{F}$;

Figure 8: Example source code and generated PDG

[1] http://grammatech.com

2.2.5 Post-mortem

This milestone was extremely helpful. It gave me a much clearer direction on where to go for the remainder of the semester. SourcererCC was not particularly promising and didn't seem to offer much beyond what existing tools offer, granted, no 1:1 benchmark was ever performed. Implementing GPLAG was a feasible goal given the amount of time remaining in the semester and it boasted very good performance while really only being a modification of *GraphQL*, a topic covered and implemented as a part of CSCI 729.

3 GPLAG

3.1 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.

3.1.1 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 I 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. This is the primary reason why, as we'll see in section 4.2, GPLAG sometimes detects completely different source files are plagiarized.

4 Comparisons

4.1 Dataset

The dataset these comparisons were performed was unfortunately quite small. The PDG generation tool used in the GPLAG implementation was ported from previous work performed by another student. It is highly dependent on the source files being structured in a particular way. Because of this I wasn't able to test GPLAG, conQAT and JPlag on a large, real-world dataset.

Instead, 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.

4.2 GPLAG benchmarks

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.

4.3 JPlag benchmarks

JPlag results were mixed. Since JPlag uses a tokenization-based approach to detecting plagiarized code it is susceptible to falling short when a number of simple obfuscation techniques are used.

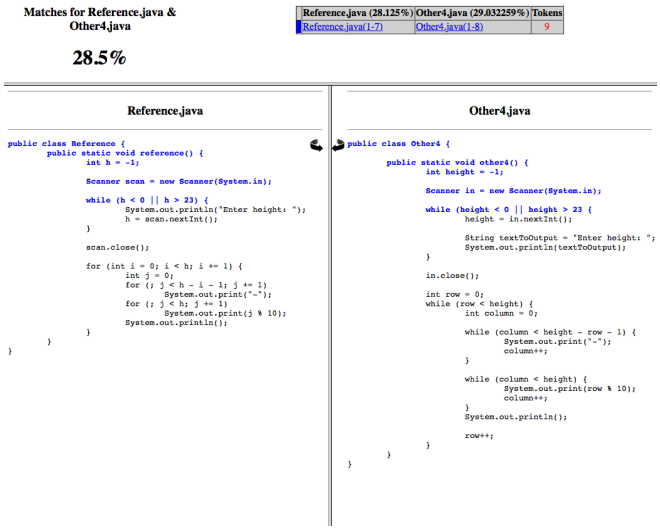


Figure 9: Comparison between `Reference.java` and `Other4.java`. Other4 should be considered plagiarized because of the semantic similarities it shares with Reference, but instead JPlag only detects a minimal amount of similarity.

As we can see in Figure 9, `Other4.java` is only considered 28.5% similar to `Reference.java`. In reality, the semantic similarity between these two files is identical. Indeed, they will print the exact same results. JPlag is thus fails due to **control replacement** and **statement reordering**, two plagiarism techniques discussed in the original paper for GPLAG and cited as major reasons why tokenization-based detection tools are not robust enough.

In comparison, when the GPLAG implementation is run over these two files, it detects `Other4.java` as plagiarized 100% of the time. This is an ideal result even without implementation of lossy and lossless filtering.

4.4 conQAT benchmarks

conQAT results were poor compared to GPLAG and JPlag. Additionally, conQAT is released as a modified complete distribution of Eclipse. The Eclipse version being used is somewhat old and had trouble running on newer macOS and Windows distributions. This further brings its usefulness into question.

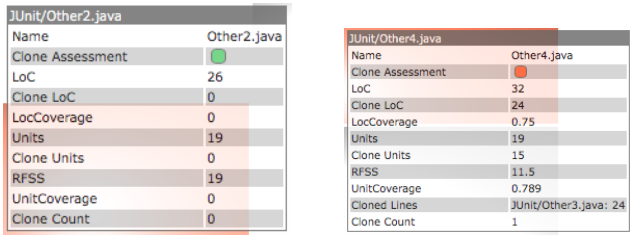


Figure 10: conQAT results from running over all source files. `Other2.java` is identical to `Reference.java` except for 1 line but is not detected as a clone. `Other4.java` is correctly detected as a clone, but only with `Other3.java` and only one line is detected as cloned.

Strangely, `Other2.java` is not detected as a clone of any other file despite `Reference.java` being identical except for one line. `Other4.java` is correctly detected as being a clone unlike in JPlag, but this is only because it detects one line as being cloned from `Other3.java`, not `Reference.java`.

conQAT's results aren't particularly robust and are somewhat confusing. The Eclipse frontend is somewhat more helpful in that it is similar to JPlag's side-by-side view - however, this doesn't matter much since the clone detection is clearly not as thorough as JPlag and completely pales in comparison to the GPLAG implementation.

5 Conclusions

5.1 GPLAG effectiveness

The results have definitively shown that even a naive implementation of the GPLAG algorithm is extremely effective at detecting both syntactic and semantic similarities between Java source code files. Although the naive implementation does return false positives the margin of error is calculable and can be circumvented by implementing a simple post-execution check.

JPlag is currently the most widely-used tool for detecting plagiarism in student code submissions. However, as we've seen, it fails to detect code that is clearly semantically identical to other submissions. This presents a huge gap in its effectiveness - plagiarists can simply change control flow and reorder statements to avoid detection entirely. It only performs consistently well for detection of copy and pasted code but simply is not robust enough to detect skilled plagiarists. Even worse would be its performance in the setting of detecting small portions of potentially stolen code that could exist in a corporate environment.

conQAT is not as widely used and the results I received from running it over the dataset were poor. However, it does have a rather comprehensive library of report generation scripts and I also imagine it would run more effectively over a larger dataset. It likely serves a somewhat different purpose as a tool, though, and this is apparent in the manner it is distributed. Being bundled together as a modified version of Eclipse is not a very modular way to distribute the software and probably wouldn't be practical in the case of running clone detection over huge codebases.

5.2 Future work

The naive implementation of GPLAG created as a part of this independent study can be improved in two major ways: adding lossy and lossless filtering, and intelligent removal of vertices from the query graph at the beginning of each recursive iteration.

GPLAG is a relatively simple algorithm but lacks any sort of intuitive report generation. I dealt with counters and console outputs to determine what it was doing. Both conQAT and JPlag have helpful frontends and generate very thorough, easy to comprehend HTML reports containing all detected clones and side-by-side comparisons of pieces of code. A more comprehensive reporting system for GPLAG would allow it to be used by a much wider audience.

The comparisons and benchmarks performed during this study were not on a large dataset. While the results I ended up with were interesting a more accurate representation of the performance of the three tools can only be obtained by running them with a larger real-world dataset.

A different method of PDG generation is also worth investigating. This is a key component of GPLAG's effectiveness. There are many ways to generate PDGs. The implementation used for this study is quite simple. The GPLAG authors themselves use CodeSurfer to generate the PDGs. The biggest problem with the PDG generator used for this study was how picky it is about source file formatting. A very large dataset of student code submissions will naturally have an extreme amount of variability in syntax and it would not be realistic to go through each source file and make it conform to the standards required by the PDG generator. Aside from being tedious this may actually change the results to not be representative of the actual effectiveness of GPLAG.

There is clearly a gap in the effectiveness of plagiarism detection and code-cloning tools despite the problems being very closely related. While the results obtained as a result of this study were preliminary they are very promising and confirm the effectiveness of GPLAG as claimed by the authors of the original paper. Development of a publicly available tool that uses a full implementation of of the algorithm would no doubt beat what is currently being used.

A Time breakdown

Time breakdown	
Task	**Time (hours)**
Understanding current code cloning techniques	≈ 10
Report summarizing current techniques	≈ 20
Research of alternatives	≈ 20
GPLAG implementation	≈ 40
Testing, benchmarking of GPLAG and other tools	≈ 20
Drawing conclusions & next steps, creating final report	≈ 20

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/.

[4] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. SourcererCC: Scaling Code Clone Detection to Big Code. *ArXiv e-prints*, December 2015.

[5] Mondego. Sourcerer CC. *GitHub repository*, 2016.

[6] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, Jul 2002.

[7] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105, May 2007.

[8] Nils Göde and Rainer Koschke. Incremental clone detection. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, CSMR '09, pages 219–228, Washington, DC, USA, 2009. IEEE Computer Society.

[9] J. R. Cordy and C. K. Roy. The nicad clone detector. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 219–220, June 2011.