

A Comparison of Similarity Techniques for Detecting Source Code Plagiarism

Bradley Beth
The University of Texas at Austin
bbeth@cs.utexas.edu

May 12, 2014

ABSTRACT

Academic dishonesty is a universal problem. Detecting duplicated text among natural language artifacts is a well-documented task. However, performing similar analysis on source code presents unique problems. In this paper, I present a comparison of the application of various techniques in textual similarity processing on source code. Beyond this, I investigate the application of textual similarity algorithms on the Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Intermediate Representation (IR) produced by the LLVM compiler toolchain. Aggregate similarity scores of a variety of methods compare favorably against the current state-of-the-art source code plagiarism tool.

1. INTRODUCTION

Academic dishonesty is a serious issue that may result in severe consequences. At the University of Texas at Austin, infractions may result in course failure or expulsion [17]. Notably, among a number of offenses, plagiarism is given specific attention. The university applies the term plagiarism to “...include[s], but is not limited to, the appropriation of, buying, receiving as a gift, or obtaining by any means material that is attributable in whole or in part to another source, including words, ideas, illustrations, structure, computer code, and other expression or media, and presenting that material as one’s own academic work being offered for credit or in conjunction with a program course requirement [16].”

This paper addresses techniques applicable to the phrase “...structure, [and] computer code...” in relation to computer science coursework. As numerous tools exist for automated plagiarism detection (PD) for natural language text, such as TurnItIn Originality Check¹, institutions often include plagiarism filters as part of the assignment submission process. The abstract nature of computer source code, however, limits the feasibility of applying such tools to computational artifacts. Although programming languages have notoriously rigid syntax specifications—as compared to natural language constructs—to negate possible ambiguities in program semantics, other features, such as arbitrary identifier names, variable whitespace, and nonlinear sequencing of code, present difficulties in textual similarity analysis unique to program source code.

In this paper, I outline a series of approaches to address these difficulties and give anecdotal test results on small

programs written in C. Each of these approaches relies on a number of stages in the LLVM compiler toolchain [7]. The compiler-generated translation and extracted program structure alleviate the lack of predictability in arbitrary naming and formatting requirements in high-level source. Using these supplementary analyses and leveraging existing text and data structure similarity algorithms, each of a number of known plagiarism-obscuring strategies can be detected with moderate to complete success.

2. RELATED WORK

The basis for much of the work in plagiarism detection stems from the Natural Language Processing (NLP) and Information Retrieval (IR) communities. Although automated plagiarism detection has a long history, the application of new techniques is still an open area of research. Many of the statistical techniques, such as n -grams and latent semantic analysis, used in other areas of text processing are applied to PD [1, 2, 12]. Commercialization of effective new techniques drives research in the area [11].

Application of these techniques to source code is also a current area of research [5, 18]. The literature base for source code PD, however, is much smaller than that of general PD. This is possibly due, in part, to the lack of commercialization in the area. The current state-of-the-art system, MOSS [9], is a free service provided by Stanford University. However, as MOSS is a client/server service with PD processing done completely server-side, specifics of its approach are not known. The limited documentation on MOSS suggests that it relies primarily on a document fingerprinting technique called *winnowing* [15].

However, commercialization has driven research in a synergistic field—code clone detection. Although the motivations are different, both clone detection and source code plagiarism detection ultimately rely on determining the similarity of arbitrary code segments. As such, much of the work done in clone detection is applicable to source code PD. In particular, clone detection has a rich history of using compiler tools to aid in similarity detection, such as Abstract Syntax Trees and Program Dependence Graphs [3, 14].

3. EXPERIMENTAL DESIGN

The goal of this project is to compare different approaches to similarity measurement found across the literature on textual plagiarism detection, source code plagiarism detection, and code clone detection on a battery of common plagiarism-obscuring methods used on school programming

¹http://turnitin.com/en_us/features/originalitycheck

assignments. Each of these methods will be discussed in turn, and are largely identified in the source code PD body of related work (see Section 2).

3.1 Project Caveats

There were a number of minor setbacks during the project that are worth mentioning:

1. Although I attempted to confine work to the stock VirtualBox .ova shell supplied in Assignment #0, the use of external tools necessitates installing additional software packages. Namely, (1) visualization and processing of the .dot graph files supplied by clang and llvm passes requires tools in the graphviz package, (2) use of the RTED tree edit distance algorithm implementation supplied by the authors is predicated on a working java installation, and (3) a robust regular expressions library is required for text preprocessing and manipulation. After failed attempts to install and use the C++ Boost libraries for this purpose in the VirtualBox environment, I opted instead to use the existing perl installation for these tasks.
2. Many of the similarity measures investigated in this project should be largely applicable across languages that are supported by llvm. However, given my working environment consists only of the clang compiler for C, my testing reflects only analysis of C programs.
3. Given the lack of a suitable testing corpus of plagiarized C programs, I created a small set of test programs to artificially simulate code plagiarism. Clearly, a more thorough analysis requires a larger, more authentic test set.
4. MOSS, the state-of-the-art tool for source code PD, includes an itemized listing of likely problematic code segments. However, given the emphasis of testing a breadth of similarity measures rather than focusing on one, this feature falls outside of the scope of this project. This is a good direction for future work.
5. Each of the approaches assumes that testing data should share no common codebase. A logical extension to the project would include specifying a given code template to ignore or remove during preprocessing.

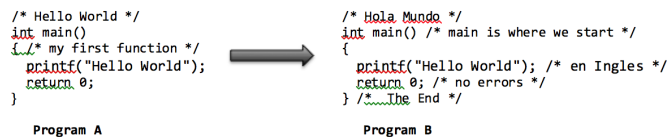
3.2 Plagiarism-obscuring “Attacks”

Verbatim source code plagiarism is easily detected. For example, simple usage of the tool, `diff -s "file1" "file2"`, indicates whether or not two files are identical. Furthermore, the textual differences and similarities between two files which are not identical are trivially obtained through common command line tools such as `diff` and `comm`, respectively.

However, plagiarized work is rarely submitted unaltered. There exist common strategies to obscure plagiarized content from the trivial analyses noted above. These malicious strategies—which I name *plagiarism “attacks”*—are documented in the source code PD literature. Those which this project focuses upon are outlined below:

1. **Comment alteration.** Programming languages typically offer the ability to annotate program source code.

These annotations are not semantically relevant to program behavior and are discarded by the compiler when transforming source to object code. As such, they are fairly commonly abused to obscure source code. The source of Program A is easily transformed to the drastically different text of Program B simply through the addition or deletion of program comments without altering how the program functions (see Figure 1). Although the object code for each is identical to the other, the source code differs on every line.



```

/* Hello World */
int main()
{
    /* my first function */
    printf("Hello World");
    return 0;
}
Program A

```

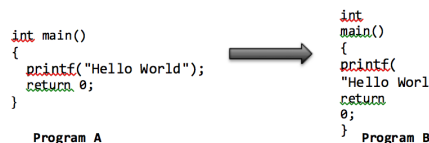
```

/* Hola Mundo */
int main() /* main is where we start */
{
    printf("Hello World"); /* en Ingles */
    return 0; /* no errors */
} /* The End */
Program B

```

Figure 1: An example of comment alteration.

2. **Whitespace padding.** In programming languages that use delimiter notation to separate statements and code blocks (in contrast to Python, which uses whitespace indentation to denote blocks), constraints on the use of whitespace are minimal. Though whitespace is used in parsing C code, the amount of whitespace may vary wildly. This is possible because whitespace in program source is only used to delimit tokens by the compiler toolchain. As long as tokens are separable, the degree of separation is irrelevant. An attacker may use this lack of syntactic significance to restructure code to appear structurally different (see Figure 2).



```

int main()
{
    printf("Hello World");
    return 0;
}
Program A

```

```

int
main()
{
printf(
    "Hello World");
return
0;
}
Program B

```

Figure 2: An example of whitespace padding.

3. **Identifier renaming.** Unlike natural language, in which each sequence of meaningful symbols has a specific use(s) determined through convention, programming languages do not. Rather than define the meaning of all meaningful symbols through convention, programming languages specify a minimal set of “reserved words” that have predetermined meanings. As such, any programmer-defined identifiers may be renamed without affecting program semantics. Note that the sentences in Figure 3 have different meanings, but the source code fragments do not.
4. **Code reordering.** Another distinction between source code and natural language narrative is the importance of sequence. This paper, for example, has constraints on the ordering of its sections. The Introduction must come first, the Conclusion must be last, and the Results should follow the Design description. However, the equivalent construct in programming languages—the procedure—is not so confined. The beginning point

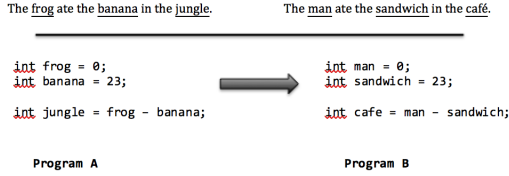


Figure 3: An example of identifier renaming.

of execution of a program (e.g., `main` in C), may be located at any point in the source file in relation to other procedures. Though it is executed first, it may be defined last. Note that this does not affect execution order, so that a would-be attacker could rearrange procedure definitions while maintaining program semantics (see Figure 4).



Figure 4: An example of code reordering.

5. **Algebraic expressions.** An attack that does have a natural language equivalent is the reordering of algebraic expressions. Just as conjunctions may sometimes have an arbitrary order of referents without changing the meaning of a sentence (e.g., “Alice and Bob met in Paris.” *vs.* “Bob and Alice met in Paris.”), some algebraic expressions may be manipulated without changing the semantics of an instruction (see Figure 5). There exist many variations of attacks of this sort, leveraging properties such as commutativity, associativity, and distribution.



Figure 5: An example of algebraic expressions.

3.3 Approaches to Similarity Measurement

There exist many well-defined metrics to measure similarity. Each of these is typically specified in terms of an inverse characteristic—*edit distance*. Edit distance is defined as the minimum number of applications from a defined set of operations necessary to transform one instance of a structure to another. Variations in edit distance correspond to differences in structures, the set of defined operations, and relative weights among their applications.

In this project, I calculate edit distances across three different data structures—strings, trees, and graphs:

- **Levenshtein String Edit Distance.** Defining measures of the edit distance between strings has a long

history. The most commonly used today is the Levenshtein string edit distance, originally developed for coding theory applications [8]. However, today its use is ubiquitous—with applications varying from automated spell checking to DNA sequencing.

Levenshtein string edit distance is defined in Equation 1, where each of the character insertion, in-place edit, and deletion operations are given equal unit weighting.

$$SED_{s,t}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0 \\ \min \begin{cases} SED_{s,t}(i-1,j) + 1 \\ SED_{s,t}(i,j-1) + 1 \\ SED_{s,t}(i-1,j-1) + \begin{cases} 0 & \text{if } s_i = t_j \\ 1 & \text{if } s_i \neq t_j \end{cases} \end{cases} & \text{otherwise} \end{cases} \quad (1)$$

The Levenshtein string edit distance is implemented as a memoized recursive process in this project, and it is used to measure similarity of both source and LLVM intermediate representation (IR) bitcode.

- **RTED—Robust Algorithm for Tree Edit Distance.**

Much like string edit distance, tree edit distance has numerous uses and a long history. As trees are often used for common computational tasks, such as sorting, searching, and maintaining a hierarchy, establishing a metric for comparison among trees is necessary. As with the Levenshtein string edit distance metric, tree edit distance is typically defined as the minimum cost to transform one tree to another using node insertions, deletions, and relabeling.

This project uses the RTED algorithm for calculating tree edit distance [10]. The current state-of-the-art algorithm for calculating tree edit distance, RTED is $O(n^3)$ in the number of nodes, n , contained in the larger of compared trees. As the metric is used comparatively across trees, I use an unweighted cost function for each of the edit operations, just as is done with the Levenshtein distance algorithm.

In short, RTED measures the distance between two trees given a path decomposition in one tree. Through successive recursive pruning steps, nodes in left and right subtrees in each of the candidate trees are mapped to one another. The minimum distance between two trees corresponds to the minimum distance obtained among all single-path decompositions in the tree.

The RTED tree edit distance metric is used solely in comparing Abstract Syntax Trees (ASTs) in this project, a technique borrowed from the literature on clone detection.

- **Graph Edit Distance.** Significantly more complex than tree edit distance, general graph edit distance suffers from the need for *seriation* [13]. As cycles and partial order relations may be present in graphs, determining which nodes should precede others is a difficult task. As the graphs used in this project are Control Flow Graphs (CFGs), seriation is loosely maintained by enforcing the partial orders present in the dominator tree for a given code segment. In this way, patterns among nodes between two different CFGs should be roughly maintained. Partial orders over dominance in one graph will also be present in the other, given they contain similar semantics.

Additionally, the search space for all possible edits in a graph is *NP-complete*. As such, a number of heuristic approaches exist to produce a locally optimal solution. As I take the approach of considering only the *structure* of the graphs—ignoring any possible attributes tied to each node—the problem of graph edit distance can be reduced to that of determining a comparable string edit distance [6].

Reducing the structure of a graph to a string proceeds as follows:

1. Generate an $n \times n$ *adjacency matrix* for the graph, where each of the n rows and columns correspond to the nodes in the original graph. Each of the elements in the matrix corresponds to the presence (1) or the absence (0) of an edge in the original graph from the row node to the column node. As a CFG is a directed graph, the resulting adjacency matrix need not be symmetrical about the diagonal.
2. Seriate the entries according to the desired ordering function. In this project, the goal is to maintain dominance relationships, so a node that dominates another will occur earlier in the matrix.
3. Flatten the adjacency matrix to a *1-Dimensional* vector of size $n^2 \times 1$ by appending rows to one another in increasing order. Consider this vector as a string of symbols, where each of the entries 0 and 1 are mapped to the characters ‘0’ and ‘1’.

A string edit distance metric (such as Levenshtein) may be applied to two such strings. However, note that two adjacency matrices of differing sizes will produce strings differing in length in the square of the difference in the number of nodes. As a final step, this difference in size is accounted for by first increasing the size of the smaller of two adjacency matrices to match that of the larger—filling the corresponding entries with zeroes—before flattening the matrix to a vector. This resizing is mirrored in the calculation of string edit distance, but counting each such additional node as a single insertion operation.

- **Document Fingerprinting.** This similarity measure is unlike the others in that it is not tied to calculating an edit distance. Instead, the text is partitioned and hashed with the idea that the reduction in dimensionality resulting from using a hash will be more feasibly compared to the hashes of other documents while preserving the same similarity relationship. In other words, if one hashed value is equal to another hashed value, it is likely that the corresponding texts are identical as well.

Winnowing, the method utilized by MOSS [9], is such a technique. An n -gram model of the source text is generated for some given n .² Each of these n -grams

²Briefly, an n -gram is a sequence of n distinct tokens. Typically, they are used to stochastically model the joint probability distribution of the occurrences of given sequences of tokens. As they are modeled as Markov processes with associated independence assumptions, they are computationally efficient.

is hashed to a numeric value, and a subset of these are chosen as an identifying fingerprint of the original text.

The fingerprinting method used in this project is similar, but accounts for every n -gram sequence in the text. *w-shingling* is a fingerprinting technique used to determine similarity in the information retrieval (IR) community [4]. Similarity between two documents is measured as the proportion of n -gram sequences the documents have in common to the number of n -gram sequences that occur in either document (see Equation 2). Note that these are set operations; this method discounts the multiplicity of observed token sequences.

$$sim_{A,B} = \frac{|S(A) \cap S(B)|}{|S(A) \cup S(B)|} \quad (2)$$

In this project, *w-shingling* is used to measure similarity in both source and compiler-generated IR code.

3.4 Evaluation Methodology

Each of the techniques in the previous section is measured pairwise against variations of short C programs. These programs are meant to reflect exercise-level programming tasks. The original versions of the test programs are exemplar samples retrieved from Rosetta Code. The first is a primitive ray-tracing routine, named “Death Star”³, and the second, unrelated example is a root solver for 3rd order polynomial equations, “Roots of a function”⁴. The two programs are comparable in length and difficulty, but share no common codebase.

3.5 “Attack” Test-Cases

To simulate plagiarism scenarios, variations of the programs are created according to each of the techniques described in Section 3.2. To control for and measure each of the attack methods independently, each of the variations contains only modifications that are ascribed to the corresponding attack. In this way, calculating the similarity between the original program and the variation with all identifiers renamed will measure the effectiveness of that attack in particular. Additionally, each of the variations is semantically equivalent to the original; the output generated by each variation is identical.

The variations are as follows:

- *Comment alteration.* The variant source file contains none of the pre-existing comments in the original and introduces a significant number of new comments interspersed throughout the text.
- *Whitespace padding.* Variation in the whitespace delimiters is included throughout, including instances of additional whitespace as well as removal of redundant whitespace in the original source.
- *Identifier renaming.* A variant source file is created that is identical to the original, except that all variable and procedure identifiers are altered.
- *Code reordering.* This variant source file swaps ordering among procedures as well as between neighboring statements that are not dependent on one another.

³http://rosettacode.org/wiki/Death_Star#C

⁴http://rosettacode.org/wiki/Roots_of_a_function#C

- *Algebraic expressions.* In this final variant, reordering at both the instruction and operand level is introduced as long as such reordering is allowable under the algebraic properties of commutativity, associativity, and distribution.

3.6 Common Preprocessing Tasks

Both the source and IR are preprocessed to remove irrelevant sequences of text for ease of analysis, such as repeated whitespace and comments. Though this trivializes the effects of string edit distance and *w-shingling* on the corresponding attacks, I view this as insignificant to the project goals. Each of these preprocessing tasks is an application of a single regular expression, consequently rendering both of these attacks irrelevant under most text-based analyses.

The ASTs generated by the `clang` compiler are preprocessed to facilitate interface with the RTED tree edit distance tool. The original adjacency list representation of the tree is converted to a bracketed tree notation in which a node’s depth is mirrored in the number of nested brackets that contain it. Thus, a perfect binary tree of depth 3 would read as follows:

$$\{a\{b\{d\}\{e\}\}\{c\{f\}\{g\}\}\}$$

The CFGs produced by `llvm` are transformed into adjacency matrices to facilitate calculating graph edit distance (see Section 3.3). Additionally, the smaller of two CFGs of differing sizes is padded with isolated nodes until the adjacency matrices are equal in size. Thus, given two matrices such as the following,

$$A = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

$$B = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix},$$

A is transformed to A' such that

$$A' = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

After resizing, A' and B are measured for similarity.

Finally, the IR produced by `llvm` is preprocessed by running the original bitcode through stock optimizing transform passes. Specifically, the following passes are performed to aid in similarity measurement:

- **-mem2reg** As memory access by default in LLVM is not in single static assignment (SSA) form, the IR formatting for variables stored in memory and calculated temporaries differs. This transform pass promotes most memory accesses to virtual registers, allowing for easier comparison and manipulation of variables.
- **-strip** Identifier names are arbitrary and thus irrelevant to program semantics. This, in fact, forms a basis

for one method of attack by potential plagiarizing students (see Section 3.2). This transform pass removes the names and symbols of registers and function definitions.

- **-reassociate** This transform pass is instrumental in accounting for *algebraic expression* attacks. Types of values, such as constants, variables, and function arguments, are ranked and reassociated in reverse post traversal order of the containing function. This serves to standardize the order of operands across similar segments of code, nullifying trivial commutative operand reordering.

4. RESULTS

The comparative results from each similarity algorithm tested over each type of identified attack are listed in Table 1. For comparison against the state-of-the-art tool, the final column represents the aggregate similarity score given for each attack by MOSS. In addition, the final row represents a comparison of two completely different C source files. Any score greater than zero indicates the probability of a false positive match. No entries in the table correspond to the trivial comparison of a file with an identical copy. Each of the similarity algorithms identify the reflexive similarity case with a perfect 1.0 score.

Analysis of the results reveals that each algorithm is more strongly suited against some forms of attack than others. As the preprocessing stages strip comments from the source in all cases, the comment alteration attack performs poorly in all cases. Similarly, the obligatory removal of redundant whitespace makes this type of attack ineffective. It is interesting to note that the *w-shingling* algorithm over the program source performs poorly against the others when analyzing a whitespace padding attack. This is likely due to the way in which the *n-grams* are constructed as much of the additional whitespace added in the attack alters the makeup of tokens on each individual line.

It is also apparent that changes to program order (e.g., code reorder and algebraic expressions) as well as changes to nomenclature (e.g., identifier renaming) are more easily detected by methods that rely on compiler-generated structures for their analyses. This is to be expected, as the compiler is both able to reduce abstraction at the token level—through disregard of arbitrary identifier names—and at the structural level. For example, the CFG represents the order of execution a program may take rather than the order in which the source was written. As this logical order must be maintained to guarantee program semantics remain unchanged, its usage is useful in disambiguating *what* and *how* a program computes from its outward appearance.

Finally, the most impressive performance is attributable to *w-shingling* over optimized LLVM IR. It comparably or out-performs MOSS in each of the five attack methods tested. Without knowing the details of the MOSS algorithm, I can only hypothesize as to why this is so. Given the performance gains in detecting code reordering and algebraic expressions modes of attack, I believe that the distinction lies in the use of the LLVM compiler transform passes to standardize the code. In particular, MOSS performs the most poorly in detecting algebraic expressions attacks. As such, it is likely that MOSS considers instances of commutative operations such as $x+4$ and $4+x$ as essentially different,

	AST	CFG	lev_{src}	lev_{IR}	$w\text{-shingling}_{src}$	$w\text{-shingling}_{IR}$	MOSS
comment alteration	1.0	1.0	1.0	1.0	1.0	1.0	0.99
whitespace padding	1.0	1.0	0.942	1.0	0.646	1.0	0.99
identifier renaming	1.0	1.0	0.654	0.9994	0.0655	0.979	0.99
code reordering	0.9299	1.0	0.3425	0.4616	0.679	0.971	0.82
algebraic expressions	0.611	0.6	0.8923	0.5229	0.546	0.746	0.68
<code>roots.c</code>	<i>0.1832</i>	<i>0.0573</i>	<i>0.2</i>	<i>0.224</i>	<i>0.0</i>	<i>0.1754</i>	<i>0.0</i>

Table 1: Results of similarity measure algorithms run on the C program “Death Star” and each of its variants. The final row is a measure of similarity to a completely different program, “Roots of a function”. Each entry is a normalized similarity measure ranging from [0.0, 1.0]. This may be read as a measure indicating the probability that the two files are the same. Note that the missing entries comparing the original “Death Star” to itself are trivially 1.0 in all cases.

whereas the method of preprocessing used for *w-shingling* over IR reassociates operands of commutative operations, making them identical at later stages of analysis.

5. CONCLUSION

Given the increases in performance obtained over MOSS in testing different methods of plagiarism detection on this artificial dataset, I believe that there is room for improvement in implementing document fingerprinting techniques over source code. MOSS clearly has robust similarity detection methods, but fails in certain circumstances, such as semantics-preserving manipulation of algebraic expressions. However, work on detecting these types of equivalences is already plentiful. Using the tools constructed for source compilation and optimization, and leveraging the techniques used in synergistic areas of research, such as clone detection, can improve the robustness of plagiarism detection tools against these types of attacks.

Given the project caveats listed in Section 3.1 and the limited nature of my test dataset, I hesitate to call these results conclusive. However, they clearly motivate the idea that there is room for growth, and compiler tools may play a significant role in realizing this growth. Ideally, as future work, I would like to compare these results to those obtained through analyzing live student work, across a number of LLVM-supported languages, and with itemized results, obtained through matching source line numbers with their IR instruction counterparts.

References

- [1] Salha M Alzahrani, Naomie Salim, and Ajith Abraham. Understanding plagiarism linguistic patterns, textual features, and detection methods. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 42(2):133–149, 2012.
- [2] Alberto Barrón-Cedeño and Paolo Rosso. On automatic plagiarism detection based on n-grams comparison. In *Advances in Information Retrieval*, pages 696–700. Springer, 2009.
- [3] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant’Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377. IEEE, 1998.
- [4] Andrei Z Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29. IEEE, 1997.
- [5] Georgina Cosma and Mike Joy. An approach to source-code plagiarism detection and investigation using latent semantic analysis. *Computers, IEEE Transactions on*, 61(3):379–394, 2012.
- [6] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. A survey of graph edit distance. *Pattern Analysis and applications*, 13(1):113–129, 2010.
- [7] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master’s thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. <http://llvm.cs.uiuc.edu/>.
- [8] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady*, volume 10, page 707, 1966.
- [9] Aiken A Moss. A system for detecting software plagiarism. <http://www.cs.berkeley.edu/~aiken/moss.html>.
- [10] Mateusz Pawlik and Nikolaus Augsten. RTED: a robust algorithm for the tree edit distance. *Proceedings of the VLDB Endowment*, 5(4):334–345, 2011.
- [11] Martin Potthast, Tim Gollub, Matthias Hagen, Johannes Kiesel, Maximilian Michel, Arnd Oberländer, Martin Tippmann, Alberto Barrón-Cedeño, Parth Gupta, Paolo Rosso, et al. Overview of the 4th international competition on plagiarism detection. In *CLEF (Online Working Notes/Labs/Workshop)*, 2012.
- [12] Martin Potthast, Benno Stein, Alberto Barrón-Cedeño, and Paolo Rosso. An evaluation framework for plagiarism detection. In *Proceedings of the 23rd international conference on computational linguistics: Posters*, pages 997–1005. Association for Computational Linguistics, 2010.
- [13] Antonio Robles-Kelly and Edwin R Hancock. Graph edit distance from spectral seriation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 27(3):365–378, 2005.
- [14] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

- [15] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003.
- [16] The University of Texas at Austin. General catalog 2012–2013, §11–402. *academic dishonesty*. <http://catalog.utexas.edu/general-information/appendices/appendix-c/student-discipline-and-conduct/>, Date accessed: May 9, 2014.
- [17] The University of Texas at Austin, Division of Student Affairs. Consequences of academic dishonesty can be severe!, 2010. http://deanofstudents.utexas.edu/sjs/acadint_conseq.php, Date accessed: May 9, 2014.
- [18] Zoran Đurić and Dragan Gašević. A source code similarity system for plagiarism detection. *The Computer Journal*, 56(1):70–86, 2013.