Software Similarity Detection

Matthew Heon Dolan Murvihill

September 10, 2015

Abstract

In response to growing academic dishonesty in undergraduate computer science and electrical and computer engineering courses, we present *Checksims*, a similarity detector designed to highlight suspicious assignments for instructor review. We report the design rationale for the software, and describe our detection of dozens of previously undetected cases of academic dishonesty in such classes.

Contents

1 Introduction

Some members of the Computer Science and Electrical and Computer Engineering departments at Worcester Polytechnic Institute have perceived a recent increase in academic dishonesty. Most have concerned the unauthorized and unacknowledged copying of program source code. The first reports came from Nicholas DeMarinis, a Teaching Assistant in an ECE department embedded systems programming course, who noticed several instances of source code that he considered to be suspiciously similar in an embedded systems programming course. He identified several cases of academic dishonesty, including one where students had extensively obfuscated the copied code.

Meanwhile, Professor Hugh C. Lauer of the Computer Science department encountered several instances of academic dishonesty in his own courses. A member of the course staff of his second-year programming class noticed a pair of students who submitted near-identical assignments, which he brought to Professor Lauer's attention. Several assignments later, another member of the course staff identified another set of students with very similar assignment — this time, by noticing that both students had submitted assignments that had identical, incorrect output.

In both cases, the unauthorized copying was caught by coincidence. Both courses had more than one teaching assistant; if the copied assignments had been graded by different teaching assistants, the copying would almost certainly not have been detected. It is statistically unlikely that copied assignments happen to be graded by the same course staff members every time, which leads us to believe that many cases of unauthorized copying have gone undetected.

The copying does not slip through because the course staff cannot recognize dishonesty when they see it. Instead, dishonest students escape because there are so many assignments that the staff does not have time to properly check them all for copying. in short, the problem is caused by data overload, which computers are well equipped to manage. So, Professor Lauer commissioned us to construct a system for automatically detecting suspicious assignments, with the intent of integrating it into his freshman and sophomore-level courses to help identify assignments that require manual review.

1.1 Defining Academic Dishonesty

Different professors have different definitions of what constitutes illegal copying. In one class, two different students submitting similar algorithms might be considered to have engaged in unauthorized copying, while in another class their submissions might be considered acceptable. We used the definition below when building our tool. A wider discussion of varying definitions of academic dishonesty is contained in Section 3.1.

In Professor Lauer's courses, students are encouraged to collaborate on a whiteboard and design pseudocode solutions, but they must type in and debug their own programs from there. The direct copying of source code is prohibited. This way, the code must pass "through the brain". The hope is that students will gain more understanding by typing independently than by copying code directly. In Professor Lauer's view, direct copying does constitute academic dishonesty. Such rules are described in the course syllabus, and are clearly explained during the first lecture as well.

Using source code from the web follows the same rules as collaborating with other students. The use of pre-existing pseudocode as inspiration is considered acceptable, so long as the student types his or her own solution, and does not directly copy his or her entire solution from the Internet.

1.2 Project Goals

The goal of this project is to produce a software tool to provide easy flagging of unusually similar source code submissions for followup by course staff. The tool must be an easy-to-use desktop application. Future work would focus on streamlining the grading workflow by integrating the similarity detection tool with existing software tools (for example, for project submission). To facilitate future modifications and expansion, the tool must be open-source and modular.

2 Definitions

Several fundamental similarity detection terms are not in common use. For clarity, we provide their definitions here.

- The terms *academic dishonesty* and *unauthorized copying* are used throughout this paper to refer to the use of another's work without attribution and without the permission of the instructor. Such behavior is often referred to as "cheating" or "plagiarism," but we deliberately avoid those terms where possible, as they can be perceived to be controversial or judgmental.

- A *corpus* (plural *corpora*) is a body of work. A corpus may contain many documents written by many people. An example of a corpus in the world of academic dishonesty detection might be all the student submissions for a specific assignment.

- A *Token* is a piece of a larger input. Tokens are usually generated by an algorithm called a *Tokenizer*, which breaks an input string up in a consistent manner (for example, at each newline). *Tokens* are formed from the broken-up chunks of the input.

- A *False Positive* is a result that is reported, but should not have been. In the context of a similarity detection system, it would be two or more submissions that are flagged as unusually similar, but are not considered to be illegally copied.

- A *False Negative* is a result that is not reported by the tool, but should have been. In the context of a similarity detection system, it would be two results that are not flagged as unusually similar in spite of having been created by unauthorized copying.

3 Literature Review

This section summarizes our review of existing literature in the area of similarity detection. It is focused on three areas: what is academic dishonesty (Section 3.1), what algorithms exist for detecting academic dishonesty (Section 3.3), and what preexisting solutions might also solve the specific problems that inspired our solution (Section 3.4).

3.1 Academic Dishonesty

There have been a number of scholarly attempts to provide a definition for the term "academic dishonesty." Thomas Lancaster, in a 2005 survey of similarity detection systems, defines academic dishonesty as "The process by which students submit work for academic credit which contains other people's unacknowledged words or ideas" [23]. Lancaster provides a solid foundation, but there is still a question of what specific acts constitute academic dishonesty.

The definition of academic dishonesty has been complicated by the emergence of code-hosting websites such as *GitHub* and *Bitbucket*, and Q&A sites such as *StackOverflow*. On *StackOverflow*, students may ask questions on how to complete an assignment. The answers they receive might include example source code, and most would agree that copying this example code without attribution constitutes academic dishonesty. However, some professors feel that the use of any information, even hints as to algorithms or pseudocode versions of a solution, obtained from such a source constitutes academic dishonesty.

Students may also host code they wrote for a class on *GitHub*, where classmates or future students may copy the code without permission or attribution. In such cases, almost anyone familiar with academia would agree that the student who copied the given code was guilty of unauthorized copying. However, many go further, arguing that the student who originally hosted the code is guilty of academic dishonesty for enabling copying to take place — even if that student is unaware that the copying occurred [13].

In all cases, different instructors have different definitions of which offenses constitute unauthorized copying, complicating the creation of tools to assist in detecting the practice. There is no consensus among academics as to what degree of copying constitutes academic dishonesty, but most professors insist that they "know it when they see it." Mike Joy and Michael Luck provide some example behaviors that they would consider dishonest [19]:

- "A weak student produces work in close collaboration with a colleage*[sic.]* in the belief that it is acceptable."

- "A weak student copies, then edits, a colleage's program, with or without the colleage's permission, hoping that this will go unnoticed."

- "A poorly motivated (but not necessarily weak) student copies, and then edits, a colleague's program, with the intention of minimizing the work needed."

A 2006 survey of UK professors produced a broad spectrum of results for what professors perceive is (and is not) academic dishonesty [13]. The sharing of source code, comments, overall design, documentation, and user interface were all near-universally perceived to be unauthorized copying. Specifically, use of code from other sources without acknowledgment (even if the source code was adapted to the student's specific application, or rewritten from another language) was considered to be academic dishonesty.

However, many respondents noted that the degree of adaptation was an important factor, indicating that while 100% similarity was almost certainly indicative of unauthorized copying, sufficient changes to the code would render it "original"; respondents disagreed as to what, exactly, constitutes "sufficient changes."

A majority of respondents indicated that almost every offense involving the unauthorized duplication of source code could indeed be considered academic dishonesty given the right circumstances, except in cases where the copied code was written by the submitting student. In such cases (for example, the submission of an assignment written previously for a different class, with slight modifications), most respondents answered that it was a violation of course policy, but not a matter of academic dishonesty.

3.1.1 Detection of Academic Dishonesty

There has been a great deal of research focused on the detection of academic dishonesty. The detection of unauthorized copying draws heavily on the fields of Natural Language Processing and Information Retrieval to process student submissions and efficiently store and retrieve information about similarities [5]. Thomas Lancaster presents a thorough overview of existing work in his 2005 paper, including a comparison of all existing academic dishonesty detection solutions available at time of publication [23].

Academic dishonesty detection is often broken into two broad fields: detection of unauthorized copying in source code, and detection of unauthorized copying in natural language [23, 12]. The source code detection problem is generally considered the easier of the two, because programming languages follow fixed grammars; the explicit, unambiguous syntax of programming languages eliminates the need for advanced natural language processing. We focus on source code similarity detection.

3.1.1.1 Obfuscation When engaging in academic dishonesty, violators have an interest in preventing the detection of similarities in their programs. Consequently, they often take steps to obfuscate code in an attempt to hide or remove similarities. Programs and algorithms that attempt to detect academic dishonesty must be resistant to common obfuscation techniques to be successful.

Geoffrey Whale listed 12 methods of defeating similarity detection in a 1990 paper [31]. Whale's widely cited list is reproduced below. It is typically presented in order of sophistication, least to greatest.

1. Changing comments or formatting (for example, adding whitespace)
2. Changing identifiers
3. Changing the order of operands in expressions (for example, $1 + 2$ into $2 + 1$)
4. Changing data types (substituting floats for integers, or exploding structures into separate variables)

5. Replacing expressions with semantically identical equivalents (for example, `!x` with `x == false`)
6. Adding redundant statements or variables
7. Changing the order of independent statements
8. Changing the structure of iteration statements
9. Changing the structure of conditional statements
10. Replacing procedure calls with procedure bodies
11. Introducing non-structured statements such as `GOTO`s
12. Combining original and copied program fragments

Similarities are not only useful as indicators of academic dishonesty; the next section describes similarity detection techniques that are designed for other applications.

3.2 Other Applications of Similarity Detection

There are a number of fields with an interest in detecting similarities between input documents, most for purposes completely different than ours. These inputs are not necessarily source code, but the fields in question may have developed general-purpose algorithms or techniques that could be useful in the construction of our application.

3.2.1 Code Clone Detection

Most professional software engineers agree that source code should not be duplicated in a large programming project, yet it often is. Naturally, a number of tools have been developed to search for *code clones*, or instances of duplicated source code. An early code clone detector, *dup*, was developed at AT&T. *Dup* searches C source files line-by-line and can be used to detect parameterized matches — files that match when eliminating certain differences such as variable names. In 1993, *dup* found parameterized matches for 19% of the complete source code of a version of the X Window System, and 23% of a large (1.1 million line of code) proprietary AT&T system [3].

3.2.2 Bioinformatics

The comparison of similar sequences is common practice in Bioinformatics. Strands of DNA, RNA, or proteins are often sequenced and compared for a variety of reasons. A well-known example is the use of DNA testing in the criminal justice system, where DNA comparisons are used to identify the perpetrators of crimes from trace evidence. A number of commonly-used similarity detection algorithms were originally developed for bioinformatics use — for example, the Smith-Waterman algorithm described in Section 3.3.3.1 [30].

3.2.3 Copyright Infringement Detection

Some people have tried to use similarity detection techniques to detect copyright infringement. The earliest copyright infringement detection tool we found was *COPS* (COpyright Protection System), by Sergey Brin, James Davis, and Hector Garcia-Molina, in 1995 [7]. Copyright infringement detection is a very similar problem to academic dishonesty detection, but there is much more research explicitly focused on academic dishonesty than on copyright infringement.

3.3 Algorithms

Previous research into similarity detection largely falls into two categories: *feature comparisons* and *structural comparisons*. Feature comparison algorithms build a profile of various attributes of the input documents (for example, number of distinct tokens, overall word count, average number of characters per line) and compares the profiles of submissions to determine if they are unusually similar. Structural comparisons compare the content of submissions — for example, comparing the specific tokens that form the inputs [2]. Within structural comparisons, there are two broad subcategories: *vector-distance* algorithms and *fingerprinting* algorithms.

3.3.1 Syntax Awareness in Comparison

Similarity detection on source code offers a number of advantages over working with natural language. Every programming language can be tokenized and parsed according to a grammar defined by the language. Through this grammar, we can identify the purpose of all input tokens — variable name, language keyword, function name, etc. This permits the use of powerful normalization techniques that are not available when dealing with natural languages.

The abstract syntax tree representation of a program highlights similarities that may be harder to spot in plaintext. By parsing input submissions into such syntax trees, similarity detection can be performed in a less ambiguous manner. As compilers and interpreters normally perform this task when preparing to compile or execute code, performing this parsing is not an undue burden when tokenizing submissions. By parsing input submissions into an abstract syntax tree (as a compiler or interpreter normally would do to compile or execute the code), similarity detection can be performed on a representation of the program where similarities that might be ambiguous in plaintext become clear. For example, the operations $1 + 2$ and $2 + 1$ are identical in purpose, but plaintext comparison would typically not be able to identify their similarity beyond the shared + character. However, parsing into a syntax tree would create identical + nodes with children 1 and 2 for both, identifying that they are identical (as addition is commutative). Furthermore, it becomes possible to apply a consistent normalization scheme to things like identifiers. By renaming identifiers in a consistent manner, it becomes possible to remove the effectiveness of some approaches to obfuscating similarities (namely, renaming functions and variables) [2].

The use of parsing and related normalizations can be a powerful tool to detect obfuscated similarities. All 13 of the obfuscation techniques discussed in Section 3.1.1.1 can be identified and defeated by using normalized abstract syntax trees produced by a parser [2]. The obvious disadvantage of this approach is that the parsing phase is language-sensitive. This limits a similarity detection system to a few languages and imposes a significant burden to support additional languages. In addition, language-specific parsing entails information loss; some features that may be used indicate similarity, such as comments or identifier misspellings, are often lost during parsing and by similar normalizations. Despite its drawbacks, almost every similarity detection system targeted at source code uses a syntax-aware parsing phase to catch similarities that might otherwise be undetectable [31] [2] [23].

3.3.2 Greedy String Tiling

Greedy String Tiling is a popular algorithm that is most notably used by *JPlag*, described in Section 3.4.2 [27]. First, the algorithm locates the longest string that is shared between the two documents and designates that string as a "tile." It replaces the tile with the empty string, then tiles the next longest substring that is not already part of a tile. It continues tiling the next largest shared string until the largest match falls below some threshold. The algorithm is guaranteed to terminate because the length of the maximum match decreases with each step. Greedy string tiling runs in $O(n^3)$ worst case time, but in $O(n)$ best case [27].

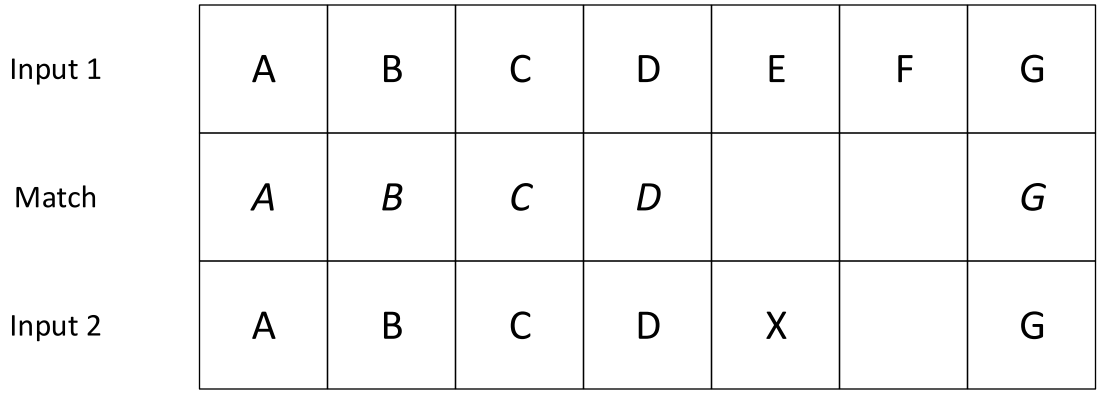
JPlag introduced several improvements on Greedy String Tiling. First, it searches for matching strings using Karp-Rabin string matching, as described in Section 3.3.4.1. Second, when searching for the longest common substring between documents, it skips a number of comparisons equal to the longest common substring found so far. If the next character after that jump is a match, then, and only then, does the algorithm step back to try to verify the string. Finally, *JPlag* enforces that the shorter document is always treated as the query document. *Jplag*'s optimizations reduce the average-case running time of GST to $O(n)$ [27].

3.3.3 Vector-Distance Algorithms

Vector-distance algorithms compare two or more inputs and identify the number and sequence of edits that must be made to transform one of the inputs into the other(s). The complement of this sequence of edits is the set of things that did not change — the similarities between the two documents. Vector-Distance algorithms are typically the slowest of all similarity detection algorithms, but fingerprinting and feature comparison may not identify the best possible match.

3.3.3.1 Smith-Waterman Algorithm The *Smith-Waterman* algorithm was developed by Temple Smith and Michael Waterman in 1981 for the comparison of DNA sequences [30]. It is a dynamic programming algorithm that seeks to find the optimal alignment of two strings. Unlike algorithms that solve the traditional longest common substring problem, the Smith-Waterman algorithm is tolerant of skipped or unmatched characters. Figure 1 shows a sample alignment of two strings, "ABCDEFG" and "ABCDXG." The longest common substring of the two would be "ABCD," but a local alignment also captures the matched "G" character.

Smith-Waterman, like most vector-distance approaches, compares pairs of inputs. Each input is placed on an axis of a 2-dimensional matrix, one token to a row or column, as is



Input 1	A	B	C	D	E	F	G
Match	*A*	*B*	*C*	*D*			*G*
Input 2	A	B	C	D	X		G

Figure 1: A local string alignment of the type generated by the *Smith-Waterman Algorithm*

shown in Figure 2. The array is initialized to 0, and is then filled top to bottom, left to right. During filling, each cell is initially filled with the largest value of its predecessors (directly-adjacent cells to the left, above, and to the upper left of the cell). If the characters on the X and Y axis match, the cell is incremented by a fixed value; if the characters do not match, the cell is decremented (unless it is 0, in which case no action is taken). The largest value in the array represents the best alignment of the two inputs [30]. Due to the need to hold and then fill this array, Smith-Waterman is an $O(n * m)$ algorithm in time and memory, where n and are the length of the two inputs.

In 2004, Robert Irving adapted the Smith-Waterman algorithm for program similarity detection [16]. By repeatedly applying the algorithm to a pair of inputs, removing the detected overlay afterwards, Irving was able to identify all the local alignments of the two inputs over a given threshold. He presents a set of optimizations to the original Smith-Waterman algorithm that improve its performance when applied repeatedly to the same inputs by removing the need for recomputation of the unchanged parts of the array.

3.3.4 Fingerprinting Algorithms

Fingerprinting algorithms (also known as *feature extraction algorithms*) extract a number of *fingerprints* (or *features*) that can be used to identify a document. These fingerprints are then added to a database (with a list of references back to the inputs that contained them). To identify similarities, an input's fingerprints are computed, added to the database, and then looked up to see whether any submissions other than the current one have the same fingerprints [28].

Fingerprints are typically the hashes of one or more tokens of the input. Typically, before being inserted into the database, a small subset of the tokens is selected, and the rest are discarded; this is done to reduce the number of entries required in the database. A typical solution might be to discard all hashes except those that are congruent to 0 modulo a certain number p, but better fingerprint selectors exist [28].

$$H = \left(\begin{array}{cccccccccc} & - & A & C & A & C & A & C & T & A \\ - & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & 2 & 1 & 2 & 1 & 2 & 1 & 0 & 2 \\ G & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ C & 0 & 0 & 3 & 2 & 3 & 2 & 3 & 2 & 1 \\ A & 0 & 2 & 2 & 5 & 4 & 5 & 4 & 3 & 4 \\ C & 0 & 1 & 4 & 4 & 7 & 6 & 7 & 6 & 5 \\ A & 0 & 2 & 3 & 6 & 6 & 9 & 8 & 7 & 8 \\ C & 0 & 1 & 4 & 5 & 8 & 8 & 11 & 10 & 9 \\ A & 0 & 2 & 3 & 6 & 7 & 10 & 10 & 10 & 12 \end{array}\right)$$

Figure 2: The array built by the *Smith-Waterman Algorithm*

3.3.4.1 Karp-Rabin String Matching The archetypal fingerprinting algorithm is *Karp-Rabin* string matching. *Karp-Rabin* generates "rolling hashes" of the two input submissions (the hash of characters N to M of the input, then the hash of characters $N+1$ to $M+1$, and so on). The sets of hashes for the inputs are checked against each other, and the matches are used to identify common substrings. [20]. *Karp-Rabin* string matching is a fundamental technique used and built upon by many later string-matching papers [28, 9, 12, 2, 8, 17].

3.3.4.2 n-gram Fingerprinting By far the most common document fingerprinting approach is called *n-gram fingerprinting.* Figure 3 presents an overview of *n-gram* fingerprinting. The document is tokenized and considered as sequences of n tokens, or *n-grams.* The *n-grams* overlap, so that a document with L tokens is considered as $L-n+1$ *n-grams.* The hashes of these *n-grams* are stored in a database and checked against those from other documents. The hashes are usually stored with pointers to their occurrences to aid in detection.

Most corpora are far too large to store the the hashes of every *n-gram* from every document in the database. Usually, a small subset of all hashes are selected through an algorithm; these serve as the document's *fingerprints.* This does entail some information loss; even word-for-word identical documents only match if they both contain an *n-gram* that was selected as a fingerprint. The fingerprint selection algorithm is consequently very important. A very simple approach is to select *n-grams* whose hashes are divisible by some number p [28], but this technique can leave gaps of arbitrary length between fingerprints, so that a large chunk of similar text might slip by.

Schleimer, Wilkerson, and Aiken [28] presented an alternative fingerprint selection algorithm in 2003 called *winnowing.* After dividing the document's tokens into *n-grams*, winnowing further divides the *n-grams* into *windows* of size t. Winnowing always selects exactly one hash from each window, guaranteeing that a match of t or more consecutive tokens is

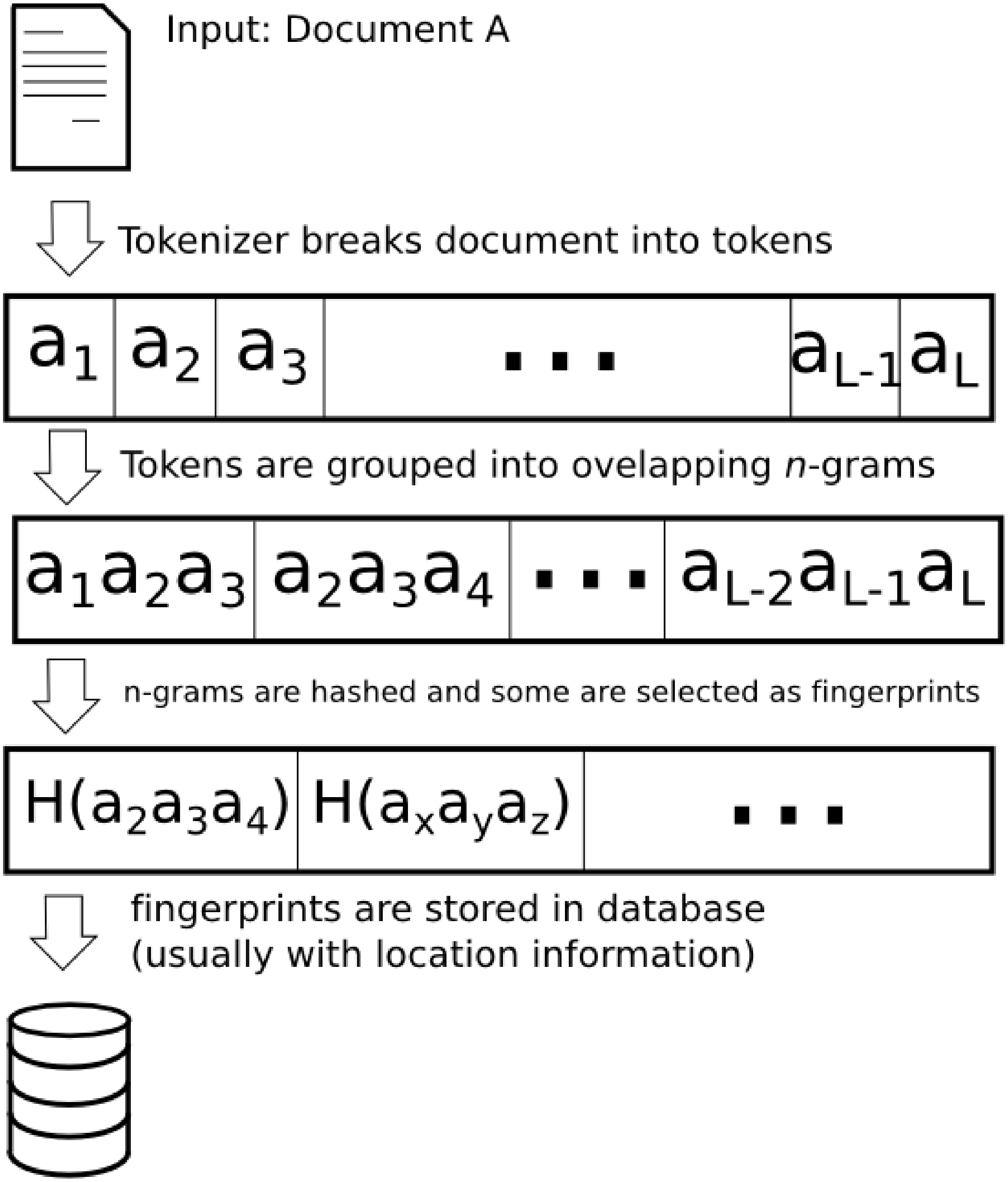

Input: Document A
Tokenizer breaks document into tokens
a_1 a_2 a_3 ... a_{L-1} a_L
Tokens are grouped into ovelapping *n*-grams
$a_1a_2a_3$ $a_2a_3a_4$ ... $a_{L-2}a_{L-1}a_L$
n-grams are hashed and some are selected as fingerprints
$H(a_2a_3a_4)$ $H(a_xa_ya_z)$ ...
fingerprints are stored in database
(usually with location information)

Figure 3: An overview of *n-gram* fingerprinting

identified.

Winnowing has proven to be extremely powerful; it is used by the current industry standard source code similarity detector, *Measure of Software Similarity*, or MOSS, described in Section 3.4.1.

3.3.5 Feature Comparison

Feature Comparison, also known as *Attribute Counting*, attempts to compare features of inputs not related to their structure (comparing the actual tokens that form the inputs). These comparisons are typically based on *profiles* of the input, representing a composite of a number of defining features — for example, line count, word count, character count, average word per line [2].

Feature comparison algorithms were common in the early days of similarity detection in the 1980s and early 1990s [25, 12]. However, it has become less popular of late because of the growing effectiveness of structural comparison algorithms such as *Smith-Waterman* and fingerprinting. Feature comparison algorithms can be just as accurate as structural comparison algorithms, but require a great deal of tuning as the differences in the profiles of documents containing unauthorized copying and those that do not are typically very small [2]. We found one reference to the creation of a similarity detection system using this algorithm after 2000, but no others [18]. The paper describing this system provided no evidence that the aforementioned criticisms were not valid, and did not provide any results to substantiate its effectiveness. Given the lack of popularity of this approach and its noted disadvantages, we did not pursue this line of inquiry.

3.3.6 Other Approaches

Burrows, Tahaghoghi, and Zobel [8] have developed a highly scalable approach to similarity detection using advanced information retrieval techniques, which can handle checking for similarity among tens of thousands of source code documents. The Burrows approach requires users to select a function of two documents that outputs a "similarity score" for them. Because the effectiveness of the Burrows approach requires using an intelligent similarity scoring function, a number of such functions have been developed, including the Okapi BM25 function and a family of functions developed by Ciesielski, Nelson, and Tahaghogi using genetic programming [11].

Belkhouche, Nix, and Hassell [4] have contributed an elaborate academic dishonesty detection approach that compares C programs by converting them to structures representing control flow, data tables, and other high level concepts. Their implementation is highly language-specific.

So many other detectors of unauthorized copying have been developed that describing all of them here would be impractical. Lancaster and Culwin have constructed a detailed and very helpful taxonomy of commonly known copy detection tools, and have described those tools in terms of that taxonomy [23].

3.4 Existing Solutions

A number of programs exist for finding similarities between source code. Some of these, like MOSS, are intended for use in detecting academic dishonesty; but very few are publicly available. Many solutions mentioned in literature were never released, or have not been maintained for many years. The two noteworthy products, MOSS and *JPlag*, are described below.

3.4.1 MOSS

MOSS, an acronym for *Measure of Software Similarity*, is a solution developed by Professor Alex Aiken of Stanford in 1994 [6]. Today, two decades later, it is considered the gold standard in software similarity detection [5]. MOSS is free for non-commercial use (though it was previously restricted only for use in academia). It is an online service, not a software tool that can be deployed. Its primary interface is a Perl script that provides a command-line frontend to submit code for analysis. Results are provided via email, and can take several hours to arrive [6]. MOSS is based on the n-gram Fingerprinting algorithm with winnowing described in Section 3.3.4.2. Most of the details about the implementation are public, but the tuning parameters of the algorithm are kept private.

Despite having used the same algorithm for over a decade, MOSS remains highly competitive. In a battery of tests run in 2014 using several similarity detection algorithms, MOSS posts detection results comparable every other algorithm tested [5]. In fact, when new similarity detection systems are published, they often compare their results with those of MOSS.

All academic dishonesty detectors suffer from the diversity of definitions of unauthorized copying, and MOSS is no exception. MOSS often flags behavior that our advisor, our primary customer, does not consider to be unauthorized copying. The secrecy of MOSS's tuning parameters and database also prevent researchers from independently evaluating its performance or reproducing any of its results.

3.4.2 JPlag

JPlag is a web service developed by Guido Malpohl in 1996. *JPlag* compares closely with the performance of MOSS, and some academic dishonesty detection tool developers choose to forgo testing their own tools against MOSS in favor of *JPlag* [27, 11].

JPlag allows users to submit an archive of programming files to its web service; the files will then be compared pairwise against each other. *Jplag* first runs the programs through a parser or scanner for the appropriate programming language, then uses the outputs from the parser or scanner as the input strings to its backend algorithm; only the front-end parsing step is language-dependent [27]. *JPlag* is advertised to support C, C++, Java, Scheme, C#, and even natural language.

JPlag's core algorithm is Greedy String Tiling, with a number of optimizations, as described in Section 3.3.2.

4 Requirements

Our advisor, Professor Lauer, commissioned the *Checksims* similarity detection tool with the following requirements:

- The program should be usable by course staff with very little or no training, and should produce output in a form that can be easily interpreted.
- The output itself should not be a definitive accusation of academic wrongdoing; instead, it should simply flag suspicious submissions for further review by course staff.
- The detector should to be complete and usable within seven weeks; this requirement placed a severe time limit on implementation and encouraged the implementation of a relatively small set of features.
- The detector should not attempt to perform language-specific analysis of the source code, but instead only interpret submissions as plaintext. The language-agnosticism requirement came from the time constraint and the potential that the detector would be used for a number of classes using various languages.
- The algorithm should be run locally and preferably be easy to invoke once student submissions are closed.
- Finally, the detector should be made to match Professor Lauer's specific definition of similarity and academic dishonesty, which is very permissive of relatively similar code so long as it was typed separately.

Source code similarity detection tools almost always parse input submissions into syntax trees, as it is very easy to disguise (intentionally or unintentionally) similar code through a number of small tweaks (swapping argument or operand order, for example). It was apparent to us that performing any kind of syntax tree analysis was incompatible with the requirement that the system produced should operate on plaintext only. The plaintext requirement was the overriding concern — syntax-based parsing limits the languages that can be used with a detector, and it would also be difficult to implement given our time constraints.

The requirements seem to lead to a tool that is small in scope, implementing only a subset of the functionality that might eventually be desirable. The solution should be modular and easily extensible, so that new features can be added to the very basic initial feature set; the intention is to pass the tool on to future project teams, or to the open source community. The requirements emphasize an extensible architecture, proper documentation and unit testing, and a useful, extensible test suite.

The requirements lead naturally to a client-based solution (as opposed to a hosted solution such as MOSS). The short implementation time led to very simple frontend and user-interface code, ensured that as much implementation time as possible was spent working on the actual similarity detection code. A client based solution does have the notable disadvantage of complicating access to larger corpora of assignments, which have to be downloaded and run locally. While a larger data set is certainly desirable, implementation simplicity was more important. Furthermore, an open-source, client-based solution offers the ability for technically capable users to easily modify our solution to meet their own needs. Finally, a client-based solution is transparent in its required operation — there is no magic occurring "behind the curtain" on the server. The transparency of the design also makes it very easy to independently verify results.

Instructors and teaching assistants at WPI use Windows, Mac OS X, and Linux. The tool should be platform-independent and as easy to use as possible on all three major operating systems. A graphical interface would improve usability, but due to implementation time constraints it was not added as a requirement.

5 Approach

Our similarity detection tool, *Checksims*, was built to fulfill all requirements outlined above. *Checksims* is a client-based Java application implementing a pair of similarity detection algorithms and several output formats. It performs similarity detection on an arbitrary number of student submissions within a single assignment, and it produces output that can easily identify submissions that need to be checked by hand for unauthorized copying.

Checksims uses a simple, modular architecture designed for easy extensibility. It is designed to be trivial to add new similarity detection algorithms or output strategies without significant changes to the program, thus enabling future projects to expand on our work and offer further features without requiring significant changes to the core program, increasing productivity and decreasing the likelihood of introducing bugs.

5.1 Architecture

Checksims has a roughly linear architecture, composed of a number of discrete components, most with one input and one output. The overall service accepts a set of student submissions as input, and returns usable output. The overall architecture is shown in Figure 4 and is described below.

1. Student submissions are first processed by the *tokenizer*. The tokenizer identifies all files within the submission, applies a tokenizing algorithm on them, and yields the resulting tokenized submissions.

2. The tokenized submissions are then passed into a *common code remover*, which removes code designated as "common" from all tokenized submissions to ensure it is not matched.

3. One or more *preprocessors* are applied to the tokenized submissions, transforming the tokens to improve accuracy.

4. Submissions are then grouped into pairs. A user-selectable *similarity detector*, which implements a *similarity detection algorithm*, is then run on all possible pairs of submissions, and the results are recorded in a *similarity matrix.*

5. The similarity matrix is then passed to a user-selected *output strategy*, which produces a human-readable form of the output for parsing.

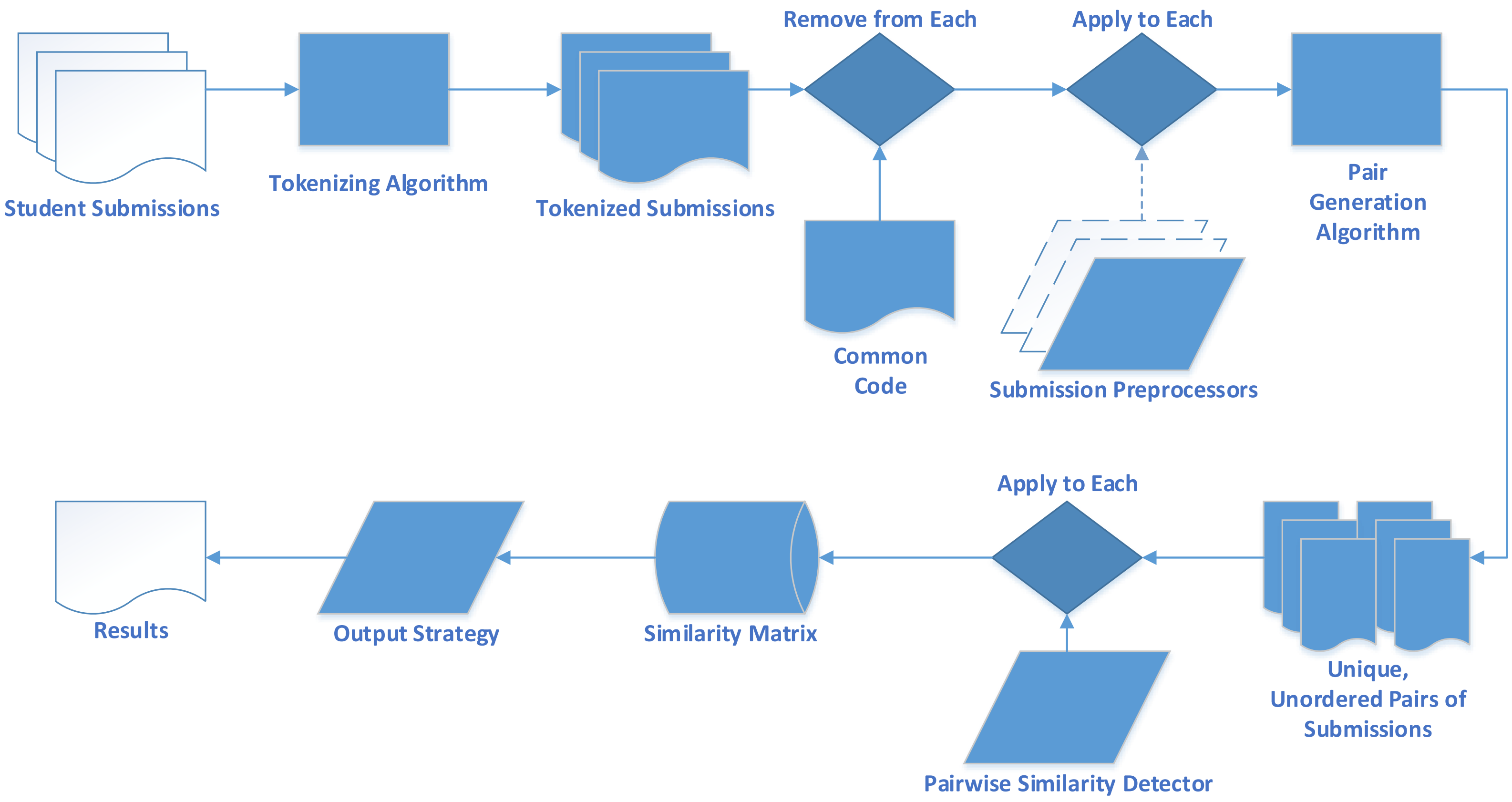


Figure 4: The Architecture of *Checksims*

5.2 Submissions and Tokenization

Checksims accepts an input directory containing a number of student submissions and a pattern to match files to be composed into a submission. It is assumed that each submission is contained within a single subdirectory of the input directory — that is, all student code is located in subdirectories of the input directory, and each subdirectory contains the code of exactly one student. All files within a single subdirectory are considered to belong to a single student, and are compared with a *match pattern* to determine whether they are checked. This allows *Checksims* to only include source files; for example, a match pattern of `*.{c,h}` would only match C source files, and would ignore README files.

Submissions are then split into tokens for comparison by a similarity detection algorithm. Several tokenizers are included to perform this task, all resulting in a linked list of tokens representing the original input. The same tokenization algorithm is used for all submissions to ensure internal consistency. None of the provided tokenizing algorithms affect the order of the input submission, though some may alter the original submission by removing whitespace. The provided tokenizing algorithms operate on the plaintext of the submission only. No attempt is made to parse the input into a syntax tree (or even use the grammar of the input language at all), pursuant to our stated goal of performing plaintext only comparisons. Parsing using a language-specific grammar could be added to *Checksims*, but as no provision was made for it initially, would probably be more difficult than simply plugging in a new similarity detection algorithm.

5.3 Preprocessing and Common Code Removal

Once submissions have been accepted and tokenized, *Checksims* performs common code removal. Common code removal accepts a submission that contains code that is expected to be present in all submissions — for instance, templates, copyright notices or helper functions provided by the instructor. This code is tokenized using the same method used for all other submissions, and similarity detection is performed between it and each submission. Any code matching the common code is removed before the main similarity detection algorithm is run. Common code removal can also improve the performance of more expensive similarity detection algorithms by making submissions smaller.

Checksims offers the option of modifying submissions with one or more preprocessors after performing common code removal but before performing similarity detection. Preprocessors manipulate the token representations of submissions to normalize them prior to running detection algorithms. The only preprocessor in the current release converts all letters to lower case. Preprocessors are implemented modularly, and more are planned in the future.

5.4 Similarity Detection

After submissions have been tokenized and normalized, *Checksims* applies a pairwise similarity detection algorithm to all unique unordered pairs of submissions, obtaining the similarity of every submission in the input with every other input submission — a complete picture of the similarities within the group. At present, two pairwise detection algorithms are included with *Checksims*: *Line Comparison*, and *Smith-Waterman.*

Comparisons are carried out pairwise in order to simplify the construction of *Checksims* and enable easy multithreading. An alternative would be to create a database of features (token sequences, for example) for all submissions encountered, and compare new submissions against this database to check for matches (while adding their own unique features to it, so future checks will include them). Pairwise detection is easy to run in parallel because of its lack of shared state, and easy to represent in a similarity matrix.

Many similarity detectors, including MOSS, check submissions against a database instead of running a pairwise comparison. Using a central database reduces the space complexity of the framework from $O(n^2)$ to $O(n)$ when running detection algorithms that can take advantage of it. *Checksims* does not at present support this architecture, because it would complicate implementation of multithreading by introducing a shared resource (the database). It would probably be a good idea to add support for this architecture in the future.

The following subsections describe the two similarity detectors provided with the current release of *Checksims*.

5.4.1 Line Comparison

The *Line Comparison* algorithm is a special case of *n-gram fingerprinting*, as described in Section 3.3.4.2. *Line Comparison* is meant to work with line-sized tokens. It hashes each input token and creates a map of each hash to each occurrence (the position and submission where the token was found). Hash collisions are identified as hashes that map to more than one occurrence, and collisions involving both submissions are tokens shared between the two. The percentage of tokens involved in such collisions is tallied and reported as the final result.

It is worth noting that line comparison is actually a *feature extraction* algorithm, and could be run with a central database of submissions if desired. However, since *Checksims* only implements pairwise comparisons, the "database" used is a hash table that must be rebuilt every time a submission is compared. The current architecture is not particularly efficient, but because of the high speed of modern hashing algorithms, the loss of performance does not have a noticeable impact on execution speed.

Line Comparison is a simple algorithm that was implemented as a proof of concept. It runs extremely fast in linear time with the size of both submissions, but it misses a number of similarities due to the nature of hash collisions. Even a trivial change (a single letter added or removed, for example) results in a different hash, causing the changed token to not be matched. Indeed, all of the obfuscation techniques in Section 3.1.1.1 can defeat the *Line Comparison* algorithm, unless preprocessors — which cannot easily thwart sophisticated obfuscations such as arithmetic operand reordering — are applied to combat them. Furthermore, almost all hashes enforce the property that any change in the input results in major changes to the output. Therefore, it is impossible to tell the degree of similarity between two tokens simply by comparing their hashes. Therefore, it is impossible to identify similar lines just from their hashes, preventing *Line Comparison* from being used to identify very similar hashes. Line Comparison remains in *Checksims* both as a proof of concept and example of a simple algorithm, and to quickly identify extremely similar or identical submissions (an initial check that can be run prior to a slower but more accurate algorithm such as *Smith-Waterman*).

Line Comparison is designed to work with tokens that represent lines; its usefulness with significantly coarser or finer tokenizers is questionable. In the character tokenization case, for example, almost every submission is presumably written using the same subset of characters (capital and lowercase letters, numbers, and punctuation used as language-specific syntax). These characters are shared between almost every submission; it is unusual to see a similarity result of less than 99% when using *Line Comparison* with character tokenization.

5.4.2 Smith-Waterman Algorithm

The *Smith-Waterman* algorithm is the primary similarity detection algorithm included in *Checksims*. It is described in Section 3.3.3.1. *Smith-Waterman*'s first published use in academic dishonesty detection was by Robert Irving [16]. When run against our test corpus, Smith-Waterman defeated multiple methods of obfuscation described in Section 3.1.1.1, including changed identifier names. We believe *Smith-Waterman* caught most of the instances of academic dishonesty in our test corpus. Smith-Waterman is described in Section 3.3.3.1. Section 8. discusses its performance.

Smith-Waterman scales poorly. Its running time and memory usage both scale as $O(m * n)$ where m and n are the size of the two submissions being compared (after tokenizing). If both submissions are the same size, scaling is $O(n^2)$. Initially, the algorithm ran too slowly to be useful, even on classes of 40-50 students. We made several optimization passes to improve performance, including changing the default tokenization algorithm for *Smith-Waterman* from character tokens to whitespace-separated tokens, reducing the number of tokens by an estimated factor of four. After these optimizations, *Smith-Waterman* is fast enough for typical use cases at WPI (classes of 60-70 students and submissions of 500 to 1000 tokens), though substantially larger class sizes or an increase in average size of the submissions themselves greatly increases the time required for the algorithm to complete. Larger class sizes are more manageable than larger submission sizes. Based on the results described in Section 8.1.2, we estimate that a class of 100 to 110 students might finish in around two hours while a large increase in token count could cause individual comparisons to take days to run and require dozens, if not hundreds, of gigabytes of memory.

5.5 User-Friendly Output

After all possible similarities have been computed, *Checksims* formats the results into a “similarity matrix” as described in Section 5.5.1, and then uses an output strategy to format and print the resulting matrix. A variety of output strategies are available. The specific strategy used is user-specified. All output strategies focus on presenting information in a usable fashion, with an emphasis on identifying unusually large similarities easily.

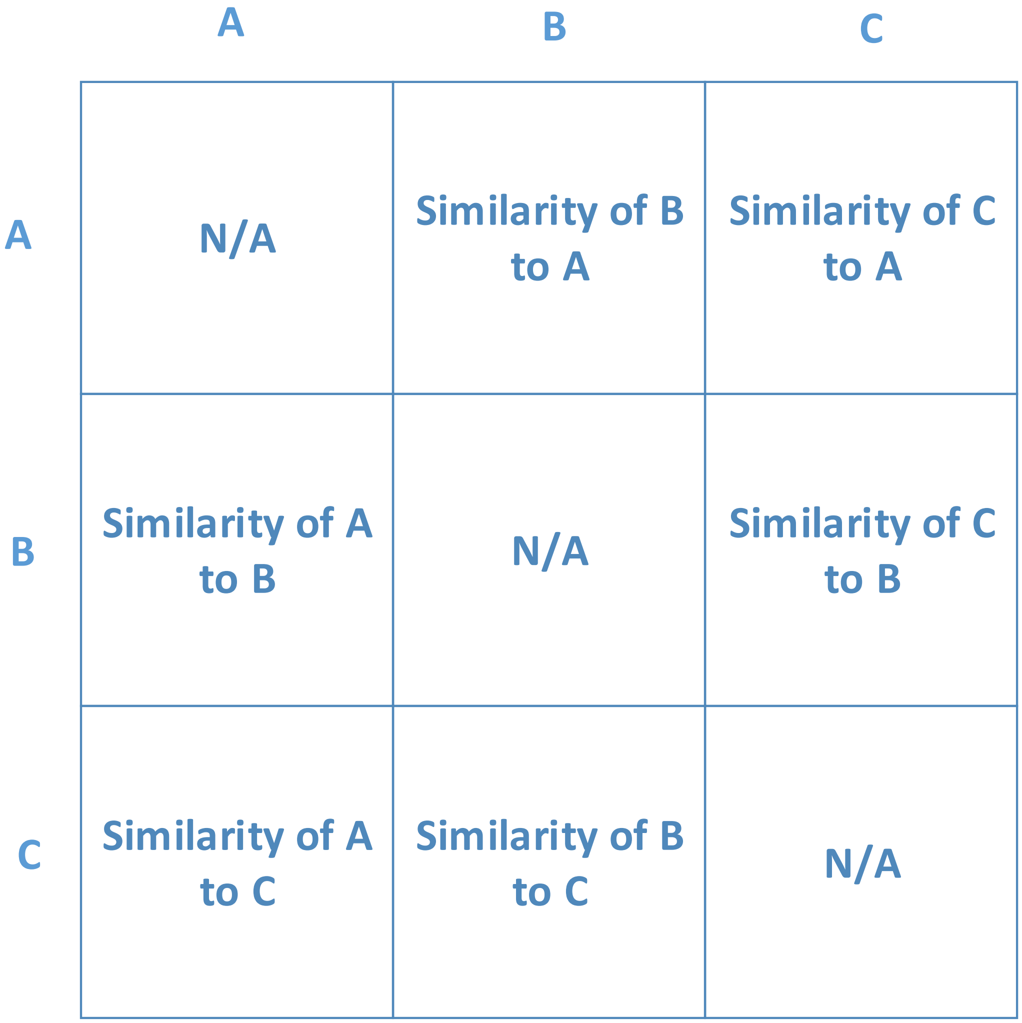
Like preprocessors and algorithms, output strategies are pluggable modules, allowing new output strategies to be written and inserted with ease, with the restriction that they can only display information contained in the similarity matrix. Some information that might be desirable to display (for example, the specific matching tokens) is not present in the similarity matrix, placing limits on what output formats are possible. It may be desirable to make additional information available to output strategies in the future.

5.5.1 Similarity Matrix

Output strategies work from a “similarity matrix”. These matrices are built from the complete results of the similarity detection algorithm, and contain the similarity of every submission to every other. As the name would imply, a similarity matrix is a NxN matrix (with N being the number of submissions), with each cell representing the similarity of one submission with another.

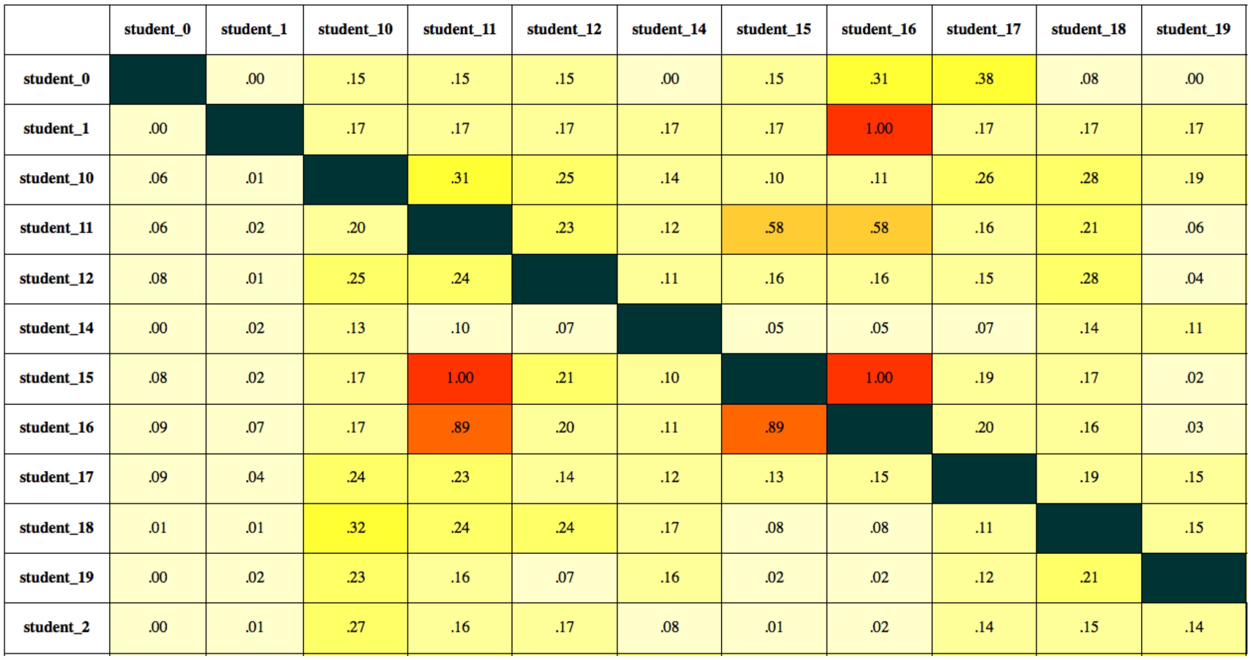
In a similarity matrix, the submissions used in similarity detection are counted, and a square matrix of that dimension is created. Submissions are each assigned a row and column. Every cell is initialized as the similarity of the submissions that define its intersection (specifically, column submission’s similarity to row submission). If the row and column submissions are the same, the cell is ignored (declared as empty). An example is shown in Figure 5. Each cell shows the similarity of the submission on the X axis with the submission on the Y axis. In Figure 5, the bottom-right corner cell shows the percentage similarity of submission A to submission C — that is, the percent of submission A’s tokens that are shared with submission C.

Some output strategies print the similarity matrix itself, possibly with visual aids to “call out” unusually large similarities. For example, the `HTML` output strategy produces a web page containing a similarity matrix with cells color-coded to allow the eye to pick out the most similar submissions by hand. A screenshot of sample output from this output strategy is shown in Figure 6.



	A	B	C
A	N/A	Similarity of B to A	Similarity of C to A
B	Similarity of A to B	N/A	Similarity of C to B
C	Similarity of A to C	Similarity of B to C	N/A

Figure 5: A sample similarity matrix



	student_0	student_1	student_10	student_11	student_12	student_14	student_15	student_16	student_17	student_18	student_19
student_0		.00	.15	.15	.15	.00	.15	.31	.38	.08	.00
student_1	.00		.17	.17	.17	.17	.17	1.00	.17	.17	.17
student_10	.06	.01		.31	.25	.14	.10	.11	.26	.28	.19
student_11	.06	.02	.20		.23	.12	.58	.58	.16	.21	.06
student_12	.08	.01	.25	.24		.11	.16	.16	.15	.28	.04
student_14	.00	.02	.13	.10	.07		.05	.05	.07	.14	.11
student_15	.08	.02	.17	1.00	.21	.10		1.00	.19	.17	.02
student_16	.09	.07	.17	.89	.20	.11	.89		.20	.16	.03
student_17	.09	.04	.24	.23	.14	.12	.13	.15		.19	.15
student_18	.01	.01	.32	.24	.24	.17	.08	.08	.11		.15
student_19	.00	.02	.23	.16	.07	.16	.02	.02	.12	.21	
student_2	.00	.01	.27	.16	.17	.08	.01	.02	.14	.15	.14

Figure 6: Sample output from the HTML output strategy

6 Need for Evaluation

Once *Checksims* had been written, the next logical step was to test it to ensure functionality. The program could not be considered complete unless it met the requirements identified in Section 4 - for example, a low false negative rate. Given this, we sought to obtain suitable test data for use in verifying our implementation.

6.1 Data Sources

Obtaining source code with known similarities is not an easy task, however. Given the intended use of our program, the most relevant source code would be from student assignments in computer science courses at WPI, and in particular from intro-level courses. Using this as test data does, however, prove problematic for several reasons, detailed below.

Existing similarity detectors for textual works often use sets of procedurally generated works, with known degrees of similarity, to assess the functioning of their programs. This presents another potential source of data, though again not without issues.

6.1.1 Student Code

Source code from students in Computer Science classes is an attractive test data option at first glance. This most closely matches the intended use case for *Checksims* — the identification of similar code submissions for programming projects. A set of test data built from submissions from previous offerings of the same classes that *Checksims* may be deployed in will most closely mirror its use in the real world.

Using student code does, however, raise a number of important concerns. The first of these is a substantial privacy concern. Student submissions are typically only available to the course staff (a professor and typically several teaching assistants). The students, when submitting, were not notified that their programs might be used in an academic study. Furthermore, it might be possible to obtain an approximation of a particular student's grade based on that student's submission, compromising his or her academic privacy.

In addition to privacy concerns, student code presents another obstacle. Prior to this project, there was no similarity detection system in common use in the WPI Computer Science department. Consequently, aside from occasional submissions identified by course staff as overly similar, there is no concrete record of which submissions within a group are similar. Any results on such code will have no baseline to compare against.

Despite these obstacles, student code remained the most desirable source of test data. The head of WPI's Computer Science department was contacted for an opinion on the use of anonymized student code, and stated that it could be used if all personally identifiable information was removed prior to it being given to the authors for use in testing. Given this, an anonymization script was written to remove such information and generate usable test code.

6.1.1.1 Anonymization Script Student code has the potential to compromise the privacy of the submitting student, but such concerns can be alleviated if all personally identifiable information can be removed. The head of WPI's Computer Science department gave permission for student code to be used if this could be done, which prompted the authors to construct a script to strip such information.

An examination was made of student code that one of the authors (an undergraduate teaching assistant for WPI's CS department) had access to. From this, several common forms of personally identifiable information were located. The first, and easiest to remove, were the filenames of the students' submission directories, which contained the usernames of the submitters. Simply renaming the directories was enough to remove this as a concern. Comments were the next concern, containing the vast majority of remaining personally identifiable information. Identifying information like names and usernames almost never occurred outside of comments. After consulting with the Computer Science department head, it was decided that stripping comments and submission names was sufficient to satisfy the requirement that personally identifying information was removed, with the caveat that a professor manually review anonymized code to ensure that no obvious identifying information remained. Professor Lauer graciously offered to perform this final vetting.

A script was constructed in Bash to accomplish this goal in a largely-automated fashion to make it feasible to obtain large bodies of test code. The script's use is limited to submissions written in languages that use comments delineated with the "`//`" and "`/* */`" symbols. The anonymization script uses a verbatim copy of the *remcomms* Sed script by Brian Hiles for removing C comments [1]. The full text of this script is included as Appendix C

Using this script, a large volume of test code was obtained from several past offerings of CS2301 (Systems Programming for Non-Majors, taught in C) and CS2303 (Systems Programming Concepts, taught in C and C++). All programming assignments for 9 previous

[1] available at http://sed.sourceforge.net/grabbag/scripts/remccoms3.sed

offerings of these courses were obtained, totaling to over 357 thousand lines of code [2]. Six of the nine courses were taught by Professor Lauer, the rest by other instructors. Between the 9 courses, there were a total of 43 student assignments, with an average of 73 students per course and 147 lines of code per student submission.

6.1.1.1.1 Accuracy Without Comments Given that anonymized student code tests were performed on submissions stripped of comments, the question is raised as to whether this is a reasonable set of data to test our program. Actual source code will, in all likelihood, have a sizable number of comments. The absence of comments has the potential to substantially alter results — for example, two submissions that differed only by comments would appear as 100% similar when, with comments, they might only be 70% to 80%. Indeed, there is the potential that removing comments might increase the accuracy of a similarity detection program (though it could also decrease accuracy under other circumstances). Because it is trivial to strip comments from files before running *Checksims*, we believe these results represent a lower bound on the tool's effectiveness. The question of whether *Checksims* might be even more effective on source code with comments is beyond the scope of this analysis.

6.1.1.2 Baseline Output A baseline for comparison was necessary to truly use student code as test data for *Checksims*. While the output of the program can be investigated to verify that any reported similarities do exist and to eliminate false positives, it is impossible to identify false negatives (submissions that contain unusual similarities but are not flagged by our software) without baseline output identifying all similar submissions. We consider consider false negatives a very undesirable characteristic, so we would prefer to verify that they are not present.

A manual investigation of an assignment would certainly prove the most precise manner of identifying similarities. However, since our obtained test data exceeds 357,000 lines of code, a manual audit of all test data is impossible to complete in a reasonable timeframe. Even auditing a single nontrivial assignment would prove extremely time consuming, given class sizes of 40 to 60 students for most of the test data. It would be possible to audit only a subset, but this greatly reduces the utility of having such a large volume of test data.

An alternative to manual auditing is to use an existing piece of similarity detection software to provide baseline results. However, no existing piece of software truly matches the definition of academic dishonesty and unauthorized copying used when building *Checksims*, as was described in Section 1.1. This could lead to a great number of false positives (for a less permissive definition) or negatives (for a more permissive definition).

Given that it is possible to manually review results to identify false positives (and doing so is far less time intensive than a manual audit), we chose to use a similarity detection program with a less stringent definition of similarity to obtain baseline results, then manually remove false positives. In doing so, we would also obtain a figure for the number of false positives such a system would produce if used in place of *Checksims*, providing insight as to how necessary the construction of a new system was.

[2] as computed by David A. Wheeler's "SLOCCount" program, a free program for computing the number of lines in a body of source code

We chose to use MOSS to generate our baseline results. MOSS, described in Section 3.4.1, is a freely available similarity detection service intended for identifying academic dishonesty. Based on previous use, our advisor believes it uses a more sensitive definition of academic dishonesty than that used to construct *Checksims*. Furthermore, MOSS is considered *the* benchmark in similarity detection by most researchers. many papers on new algorithms in similarity detection compare their results against MOSS [4, 5]. By doing the same, we can obtain a measure of our output quality compared against the gold standard in academic dishonesty detection for source code [5].

It is noteworthy that using MOSS for baseline results do not completely eliminate false negatives. If both MOSS and *Checksims* fail to detect an unusual similarity, our experiment will fail to record a false negative. We believe this event will be uncommon because of the quality and maturity of the algorithms used by MOSS (described in Section 3.4.1). It is unlikely that MOSS will miss similarities indicative of academic dishonesty [28]. Similarities that manage to escape both MOSS and *Checksims* will continue to be a challenge, but we are not aiming to produce a perfect solution, only one that is "good enough" — we do not expect *Checksims* to catch cases that have eluded the industry benchmark solution.

6.1.2 Simulated Similarity

Most natural language similarity detectors are tested and tuned with procedurally generated sets of work containing deliberately-inserted similarities. A number of such corpora are freely available online. They offer the advantage of easy verifiability — unlike student submissions, there is a well-defined set of similarities in the corpus, so output can easily be verified. There are no privacy concerns, because the test data was "written" procedurally by a program.

These simulated corpora only exist for natural language, however, where our search is for similarity in source code. We have no reason to be confident that results obtained by testing *Checksims* against natural language will correspond to our tool's performance against source code. We did use them in the project as functional tests for individual algorithms, providing a reproducible set of results to test for, but we do not infer anything about the tool's quality from their results.

7 Experimental Verification

After obtaining a large volume of test data, we executed several experiments to provide full-scale functional testing of *Checksims*. Comparisons between the two shipped algorithms, *Line Comparison* and *Smith-Waterman*, illustrate the benefits and disadvantages of both algorithms. Comparisons against MOSS demonstrate a comparable false-negative rate and a benchmark against a proven similarity detection program. Together, the experiments provided data to allow us to draw conclusions on whether *Checksims* was ready for its intended job — deployment for use by WPI course staff.

7.1 Algorithm Comparisons

Checksims presently provides two algorithms for similarity detection, *Line Comparison* and *Smith-Waterman* (detailed in Section 5.4.1 and Section 5.4.2). The first set of tests on *Check-*

sims focused on comparing these two algorithms to measure their sensitivity and runtime.

From our research, we could form hypotheses on the behavior of both algorithms. *Smith-Waterman*, guaranteed to find the most efficient local alignment of strings, should be more accurate by far than *Line Comparison*, which will fail to identify lines that differ by even one character. However, *Smith-Waterman*'s quadratic runtime and large memory requirements should mean that it is slower by far than *Line Comparison*.

Smith-Waterman is hypothesized to trade speed for accuracy, and *Line Comparison* accuracy for speed; if both hypotheses were correct, then each algorithm would have a use case and should be included in the final release. If, however, one of the algorithms does not provide its claimed advantage, there is no reason for its inclusion, and it can be removed from the final release.

To perform this experiment, *Checksims* was run twice on every assignment in our test data, once using each algorithm. The default tokenization strategy was used for both algorithms. Output was saved to a unique file for each assignment, and program runtime saved to another file. From these, results were computed. Results with similarities under 70% were immediately discarded, to limit the number of manual checks that would have to be done. Results over 70% were reported for both algorithms over all assignments. Subsequently, we identified four assignments with a significant number of similarities, and compared the results for the *Smith-Waterman* and *Line Comparison* algorithms — how many results were shared between them, and how many were caught by only one algorithm. No false positive testing was performed.

7.2 MOSS Comparison

We aimed to learn several things by comparing the output of *Checksims* and MOSS on several assignments. By comparing the number of false positives produced by both programs, we aimed to determine whether MOSS, with its more sensitive definition of what constitutes academic dishonesty, produced an inordinate number of false positives compared to *Checksims*. Furthermore, we aimed to estimate the false negative rate of *Checksims*.

To perform this experiment, we first identified groups of assignments from our test data that had a few, some, and many similarities according to data from the previous experiment. Two of each were chosen and checked with MOSS.

The MOSS results for the chosen assignments were examined by hand to identify false positives — similarities identified by MOSS that did not match our definition of academic dishonesty. The number of false positives (and the original number of results) were recorded, and the false positives thrown out. The *Checksims* results from the previous experiment for both algorithms were then examined by the same process, with false positives being identified and thrown out, and the number of overall matches and false positives being recorded. The false positive rates of MOSS and *Checksims* (both Line Comparison and *Smith-Waterman*) can be compared, both in absolute terms and as a percentage of results recorded. Finally, the results were overlaid to see which matches are missing from either program — what MOSS found but *Checksims* did not, or *Checksims* found but MOSS did not.

8 Results

The experiments described in Section 7 were performed on the anonymized student data described in Section 6.1.1. This section summarizes their results, and attempts to draw conclusions about the state of *Checksims* and its algorithms from them.

8.1 Algorithm Comparison

Our algorithm comparison tests, described in Section 7.1, were split into three overall tests. The first compared the overall number of results detected by each algorithm across all our test data. The second compared the running times of both algorithms across all test data. The third and final test compared the results returned by the two algorithms for a subset of all the assignments, to determine what matches were identified by one algorithm but not another.

8.1.1 Overall Results Detected

Figure 7 shows all results from applying both algorithms to every assignment in the test data. Results under 70% are not shown, as they are less interesting (a far lower proportion will be cases of unauthorized copying) and they are orders of magnitude more than the significant results. It is clear that *Smith-Waterman* is a far more sensitive algorithm than *Line Comparison*, detecting a great many more instances in every range save 100% similarity (which both algorithms should be able to detect easily). This sensitivity is not necessarily an indicator of accuracy, however; it is possible that all of the results reported by *Smith-Waterman* are simply false positives.

Because of the obvious limitations of the *Line Comparison* algorithm detailed in Section 5.4.1), we do not believe this to be the case. A large number of the additional results found by *Smith-Waterman* are almost certainly real instances of unauthorized copying (or, at the least, cases that should have been brought to the attention of course staff for manual review). This matches our hypothesis that *Smith-Waterman* is the more accurate of the two algorithms (though, again, we cannot prove this with only these results).

All known occurrences of common code were removed from our test data. In several cases, we were unable to obtain the common code from the instructors who gave the assignment. In each of these cases, we identified one student who had 10% similarity to all others. On further examination, such students typically had submitted no code save the common starter code. Consequently, we were able to use their submissions as common code for these assignments.

8.1.1.1 Previous Undiscovered Academic Dishonesty As Figure 7 shows, the *Smith-Waterman* algorithm identified 17 cases of 100% similarity, and a further 35 over 90%. As was mentioned in Section 6.1.1.1, all our test data is taken from actual courses — previous offerings of sophomore-level Computer Science classes. Given this, we can assume that the results in Figure 7 represent a large number of cases of academic dishonesty, though we cannot say how many of the reported results are actually cases of academic dishonesty. We can, however, draw reasonable inferences.

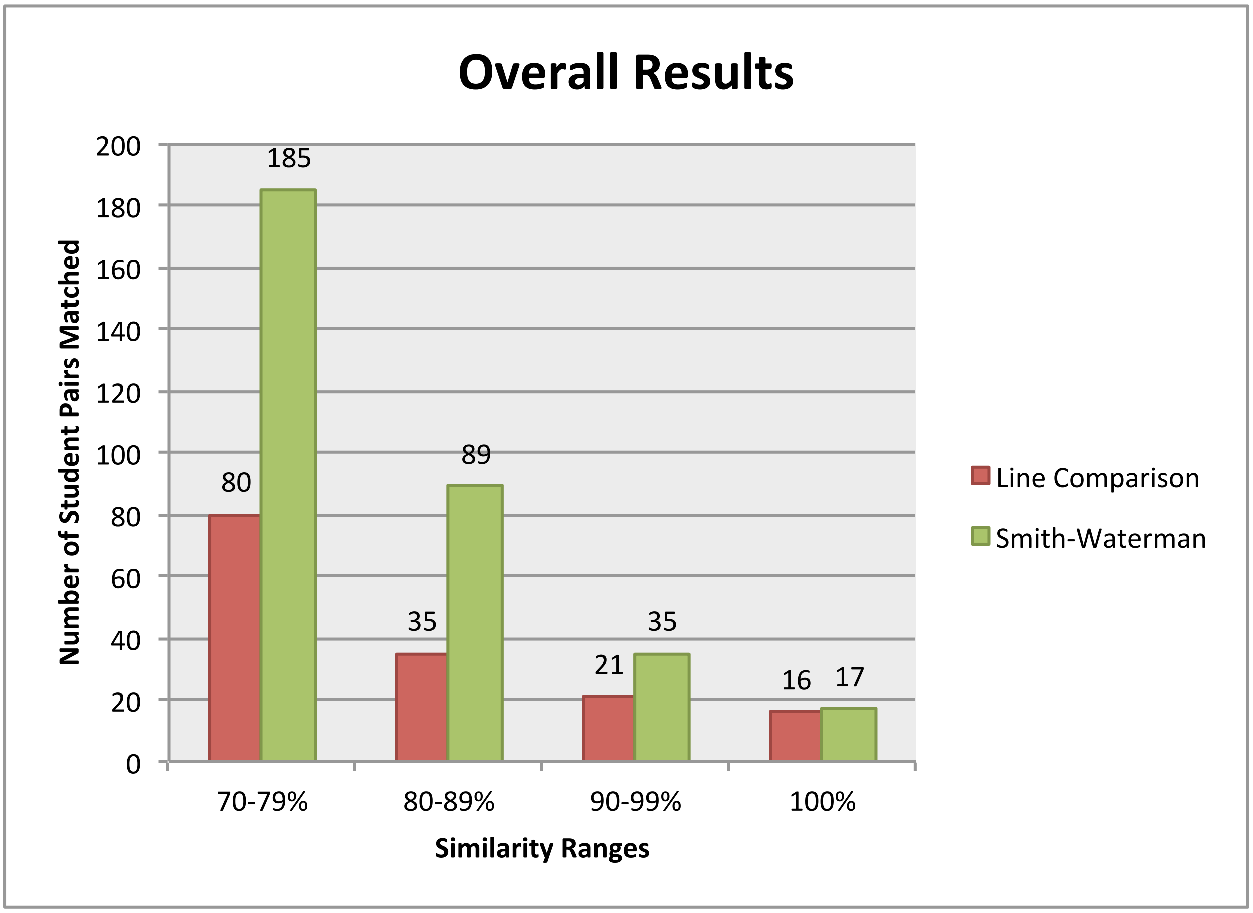


Figure 7: Results detected by *Line Comparison* and *Smith Waterman* across all assignments

The 100% results are almost certainly cases of academic dishonesty. The notion that two students could have typed in 100% identical code independently is completely implausible. We investigated several of these pairs, and all we found were what we consider to be academic dishonesty. Results from the 90% to 99% range are also nearly-certain cases of academic dishonesty. We reviewed 6 of these pairs of assignments, and found that all of them match our definition of academic dishonesty.

The 70% to 89% range, however, contains a number of similarities that may not be caused by academic dishonesty. In some cases, students implemented the same algorithm in very similar ways. Given that there are not many ways to write a simple algorithm (for example, Bubblesort), especially if typical loop counter conventions (`i` for outermost loop, `j` for next innermost, etc) are followed. Six of the nine courses we obtained code from were offerings of CS2301, an intro-level course targeted at non-Computer Science majors. Most assignments for CS2301 are very simple, requiring only trivial algorithms (without much variation in algorithm choice or implementation). Furthermore, Professor Lauer permits the copying of algorithms out of the textbook, so students may end up with completely identical versions of simple functions (like binary tree insertion). Given this, many similarities in the 70% to 89% range may have produced their code independently, but using very similar algorithms and pseudocode — acceptable behavior, according to Professor Lauer.

The potential difference between the two assignments that form our test data is emphasized by the graph in Figure 8. Almost all similarities identified are from CS2301, despite it only containing approximately 70% of our test code. Given this, the smaller similarities seen may well be due to the reasons identified in the previous paragraph.

Even discounting a large number of the similarities present as potentially not cases of academic dishonesty, our results do paint a concerning picture of undetected past incidences of academic dishonesty within CS2301. Professor Lauer has confermed that to his knowledge, almost none of the near certain cases we identified (90% and higher) were caught. We hope that the deployment of *Checksims* may be able to reduce these numbers as word of its use spreads.

8.1.2 Runtime Comparison

Figure 9 shows the runtime of the *Smith-Waterman* algorithm for every assignment in our dataset. No assignment took over 2400 seconds (40 minutes) to complete, and the vast majority finished in under 360 seconds (6 minutes). We originally intended to produce a comparative graph also showing results from *Line Comparison*, but the results from that algorithm were sufficiently similar that we did not deem it necessary to graph them. No assignment took longer than 1 second to complete using *Line Comparison*; the graph, compared to the results from *Smith-Waterman*, would be a flat line slightly above 0. This supports our hypothesis that *Line Comparison* is the faster of the two algorithms, and confirms its usefulness as a fast initial pass to identify highly similar submissions.

It is noteworthy that, while no assignment shown completed in over 2400 seconds (40 minutes), we were forced to manually intervene on one occasion. The algorithm hung for 4 hours performing similarity detection on one pair of students on one assignment in our sample data (and likely would have run much longer, had we not ended it prematurely). We investigated the assignment more closely, and found that two students had submitted an

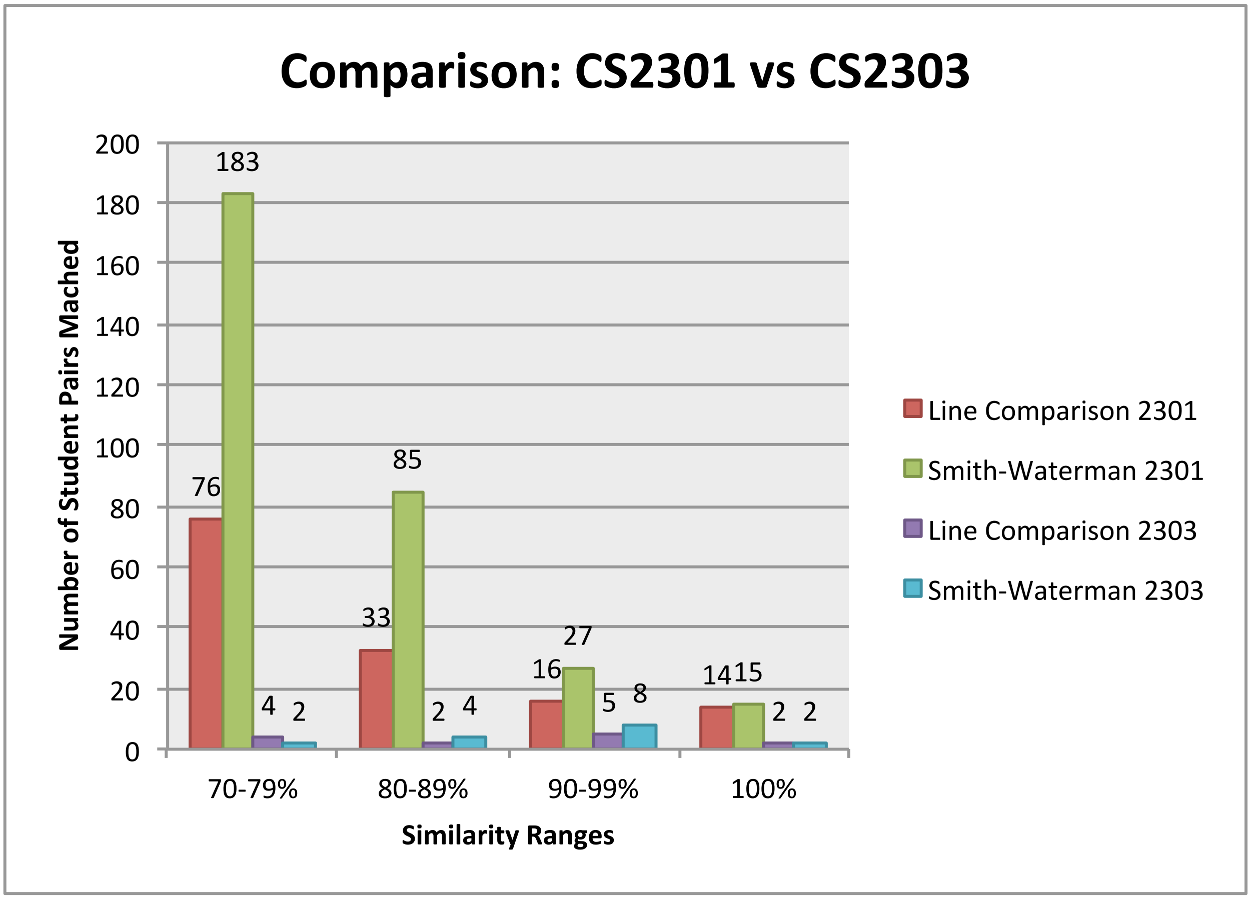


Figure 8: Results in Figure 7 split by course

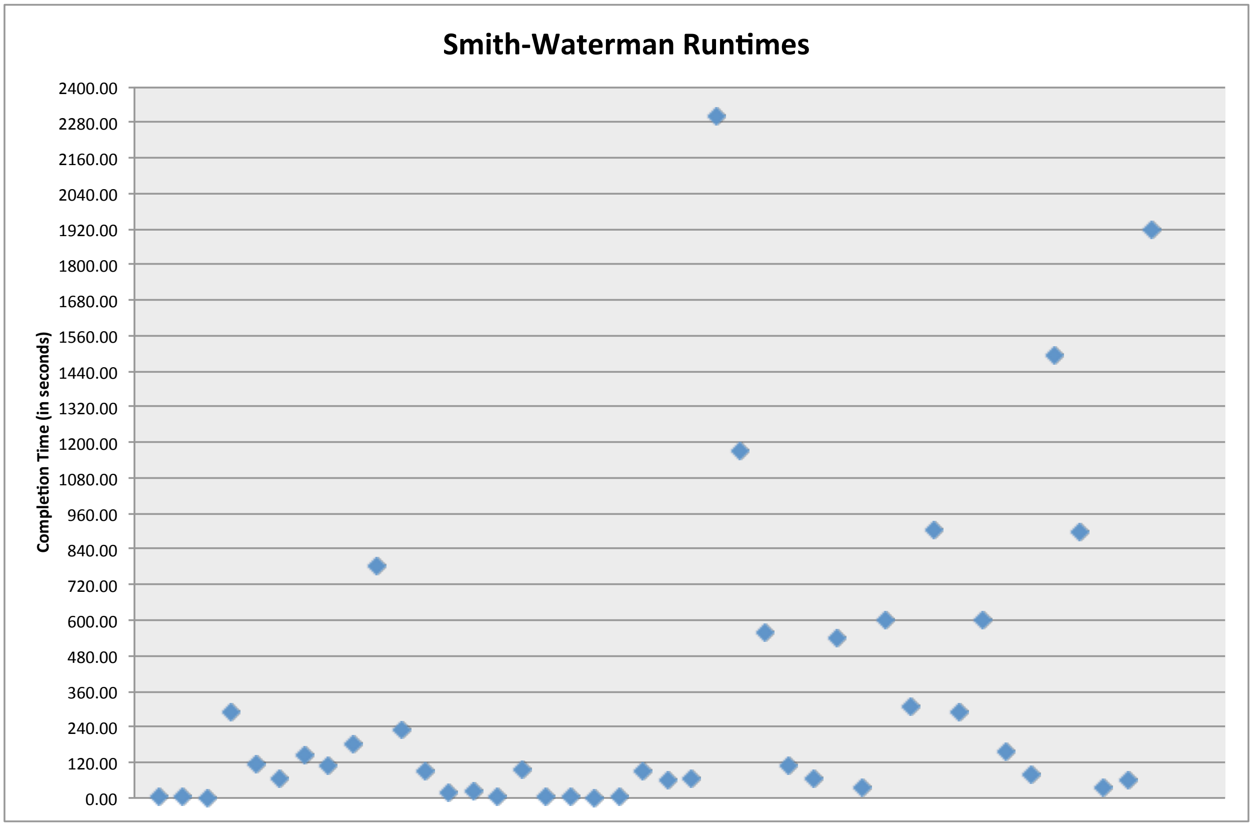


Figure 9: Runtime of the Smith-Waterman algorithm on our sample data. Each data point is one anonymized assignment.

unorthodox solution using large sets of lookup tables. The two assignments were sufficiently different to remove unauthorized copying as a factor; we believe they both came upon the solution independently. The pair of assignments using these lookup tables were approximately 3000 and 3500 lines each, and were approximately 13000 and 15000 tokens after being run through the default tokenizer of the *Smith-Waterman* algorithm. In comparison, typical student submissions for this assignment were perhaps 100 lines of code, and 300 to 400 tokens in size. By removing these two assignments from the comparison, we were able to reduce the runtime of Smith-Waterman to around one minute.

8.1.3 Algorithm Comparison

Figure 10 shows the results from 6 randomly-selected assignments from the overall dataset. The results are grouped by assignment, and show how many results were detected by each algorithm for each of the assignments. It is immediately clear that, for two of the six assignments, the results for *Line Comparison* are strict subsets of those from *Smith-Waterman*. However, the trend is somewhat reversed for assignments three and five, where a majority of results were detected by *Line Comparison* and not by *Smith-Waterman*. Manual review of these results indicates that most of these results are cases of one student submitting very little to no code (typically in the dozens of lines, with many being trivially simple — for example, } or `return;`). Many of these lines are also contained in larger assignments, so the

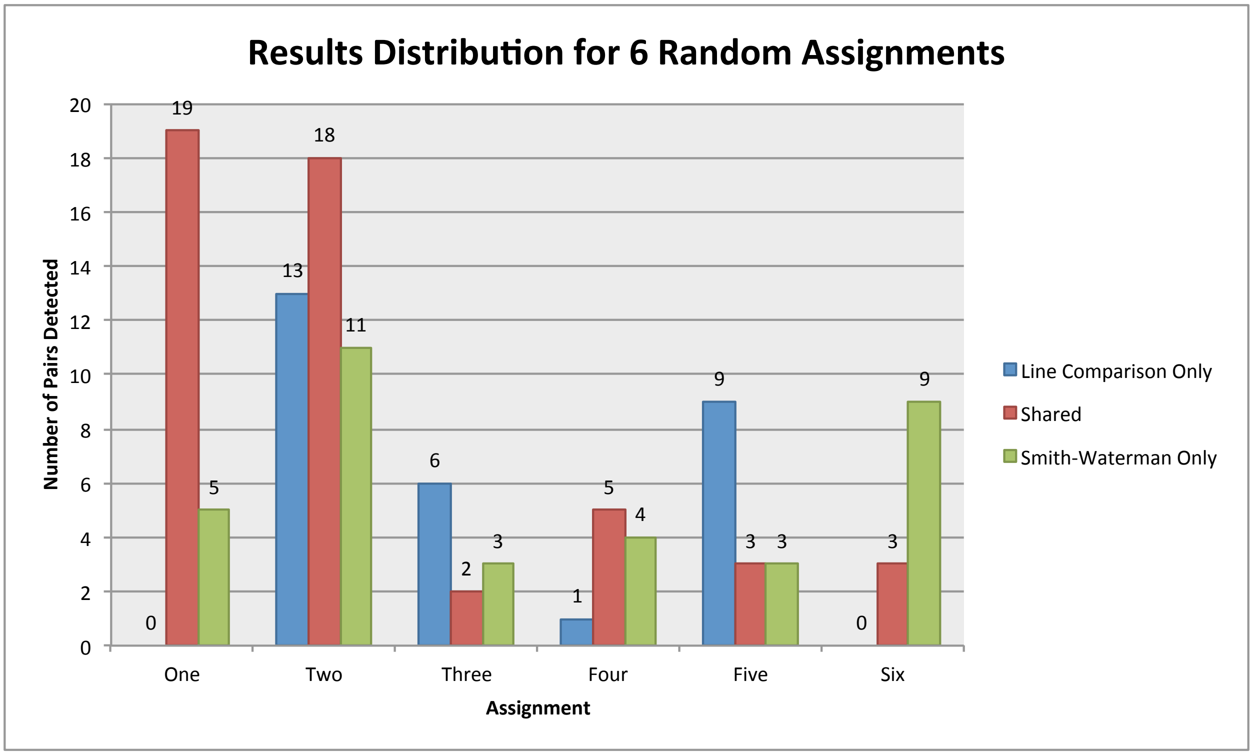


Figure 10: Detection Algorithm Results for 6 Selected Assignments

trivial assignment appears to be very similar to larger assignments to *Line Comparison* (the larger assignments typically display inverse similarities that are very small — 5 to 20% being common). *Smith-Waterman* has awareness of the ordering of tokens within a document, and consequently will ignore trivial sequences like `{ return; }` if they are not present in the submission being compared to in an almost-identical form. Through this, we can conclude that almost all results detected by *Line Comparison* but not *Smith-Waterman* are false positives, providing further evidence for the accuracy of the *Smith-Waterman* algorithm.

8.2 MOSS Comparison

We identified six assignments, containing over 120 individual student submissions, from our overall dataset to run through MOSS using the methodology explained in Section 7.2. These submissions were run through MOSS, and the results were compared to those obtained using *Checksims* using the Smith-Waterman algorithm.

For the purposes of this comparison, we must clarify our definition of "significant results." In *Checksims* and throughout this paper, we have generally defined this as any result with a similarity percentage of over 70%. MOSS, however, does not have such a definition, and instead presents results ordered by number of similar tokens, with nothing to signify which results are worthy of being inspected and which are not. For each comparison, we were able to identify a point in the results where results after had many fewer similar tokens than results before; we used these points to define significant results in MOSS, with any results coming before being considered significant. We acknowledge that this is not an optimal method of selecting significant results, but could not find a better solution. Given the ease of selecting relevant results in *Checksims*, we suggest that the overall usability of our results are better than MOSS, are no further research was performed to validate our suspicion.

Overall, we found that every significant similarity identified by MOSS was also identified by *Checksims*. The reverse, however, was not true. Several small submissions (of 75 lines and under) were identified as being very similar to other submissions by *Checksims*, but were not found similar by MOSS. We found eight such submissions, and visually examined each to make a final determination of which piece of similarity detection software was correct. We sided with *Checksims* in all cases, as the assignments were very similar on visual examination. This may indicate a weakness of MOSS with very small submissions, but we have insufficient data to draw conclusions.

It is worth noting that the definition of similar matches differs between *Checksims* and MOSS. *Checksims* ranks similarities based on the percent of tokens in an assignment that match another assignment. MOSS, on the other hand, ranks similarities based on the overall number of tokens matching another assignment. Because of this, some matches considered significant by *Checksims* are not considered significant to MOSS, which we observed several times in the results. Small submissions containing a high percentage of matched tokens were ranked far higher by *Checksims* than MOSS, though MOSS did identify the similar tokens (and high percentage similarity). We consider such submissions to be identified by both *Checksims* and MOSS, even though they were ranked highly by one and not the other. We did not observe any matches ranked highly by MOSS but not *Checksims*, though this could theoretically occur in very large submissions with a large number of matching tokens, but a low percentage of matching tokens.

Our comparison also does not test one of the potential strengths of MOSS over *Checksims*. MOSS performs syntax-aware comparisons, which lets it perform much more powerful normalizations than *Checksims* is capable of at present. These normalizations are described in Section 3.3.1, and should allow MOSS to defeat deliberate attempts to obfuscate similar code that would otherwise go unnoticed. Comparing data including such submissions might highlight the advantages of MOSS over *Checksims*, but we could not find any submissions in our dataset that contained any significant attempt to obfuscate similarities. Because of this, our comparisons are more favorable to *Checksims* than the real world may be. It is possible that the prevalence of deliberately obfuscated similarities may rise if knowledge that a similarity detection system is in use becomes widespread, which would make the performance of *Checksims* in such cases more important, and we would like to perform further experiments in this direction.

9 Real-World Usage

During this project, *Checksims* was used in real-world situations on two occasions. Course staff (including one of the authors) made use of development versions of the program to attempt to identify academic dishonesty in student submissions in ongoing courses. Though results from these real-world uses cannot be published for privacy reasons, they provide valuable insight into how *Checksims* would be used in typical class.

9.1 Embedded Computing in Engineering Design

The first usage of *Checksims* was in an Electrical and Computer Engineering department course, *Embedded Computing in Engineering Design*. A teaching assistant in that course, Nicholas DeMarinis, became concerned that some of his students may have been collaborating on their microcontroller code beyond was was allowed by course rules. He was provided an early version of *Checksims* to verify his suspicions. Though he was not permitted to share his results, he provided valuable feedback on improving the usability of the program. He requested the ability to remove common code from a submission and an output format suitable for import into a spreadsheet program for performing statistics calculations, both of which are present in *Checksims* as of the time of this writing.

9.2 Machine Organization and Assembly Language

A nearly-complete version of *Checksims* was applied to the first assignment in a course for which one of the authors, Matthew Heon, was a teaching assistant. This assignment was an excercise in optimization and bitwise operations and was composed of a relatively small C project with a great deal of common code. In a class of 68, *Checksims* detected three students with extremely similar submissions. The grading system for that assignment was highly automated and involved very little human interaction. Without *Checksims*, It is unlikely that the similarity would have been detected.

10 Future Work

Checksims was built to be extensible and easily support new features, as described in Section 5. While this is generally considered to be a good design philosophy, it was especially important because we did not have time to add so many features that we felt might be valuable. There is much work that can be done in the future to add these missing features.

We feel that one of the most important features that can be added to *Checksims* is the ability to use fingerprinting algorithms with a persistent database of student submissions; such an improvement would allow fast checking not only within a single assignment (as *Checksims* does today), but also against every assignment previously run through the detector — detecting students who, for example, used another student's assignment from a previous term. Furthermore, advanced fingerprinting algorithms offer most of the accuracy of Vector Distance approaches (like *Smith-Waterman*) while being substantially faster; for details, see Section 3.4.1). The addition of a fast comparison algorithm based on MOSS would alleviate the speed disadvantages of *Smith-Waterman* on large classes and assignments.

To add some of the benefits of fingerprinting algorithms to the *Smith-Waterman* algorithm, it would be useful to add the ability to specify an "archive" directory, which would contain a number of student submissions from previous years. The archived submissions would be compared against all student submissions from the current year, but not against each other, greatly reducing the number of comparisons that must be made and removing extraneous results about previous years. Adding such a feature would be a relatively small set of changes that would make the *Smith-Waterman* algorithm much more usable when comparing with past assignments.

Additional output strategies would be greatly beneficial to the use of *Checksims*. While we feel that our HTML and Threshold output strategies are adequate for everyday use, they fall short of being ideal for usability. For example, an output strategy that would offer the option to view the similarities detected between two assignments would potentially be very valuable to course staff, as it could speed their investigation of suspected cases of similarity substantially.

Usability improvements for *Checksims* could also come from tools to make it easier to apply to assignments. Professor Lauer has suggested integration with *TurnIn*, an in-house project submission platform used at WPI, which could run a *Checksims* scan automatically after submissions for each assignment are closed. Another option would be a GUI wrapper for *Checksims* to automate common tasks (for example, placing student submissions into separate folders, or decompressing submissions given as .zip or .tar files).

It was suggested to the authors that the comparison of programs at runtime could provide useful metrics for similarity detection — for example, Valgrind profiling of memory use and execution time. We did not find significant investigation into such techniques in our literature review, and they could certainly be investigated using *Checksims*.

11 Conclusion

Checksims provides instructors with a simple, cross-platform interface that allows programming instructors to rapidly check large numbers of assignments for academic dishonesty. It has been very successful in tests on old assignments, and shows accuracy equivelant to the industry-standard solution in initial testing. Moreover, its limited real-world deployment has already revealed a dramatic amount of unauthorized copying — 37 cases of near-certain academic dishonesty were found in 43 total assignments from 9 previous offerings of courses. Additional work and expanded deployment will likely increase its success.

References

[1] Aleksi Ahtiainen, Sami Surakka, and Mikko Rahikainen. “Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises.” In: *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*. ACM. 2006, pp. 141–142.

> GNU GPL licensed Java 1.5 source code plagiarism detection software dating to 2006. No evidence of active development after this point.

[2] Christian Arwin and Seyed M.M. Tahaghoghi. “Plagiarism detection across programming languages.” In: *Proceedings of the 29th Australasian Computer Science Conference-Volume 48*. Australian Computer Society, Inc. 2006, pp. 277–286.

> Introduces *XPlag*, a tool designed to detect instances of code copied from one programming language into another. *XPlag* functions by comparing the intermediate representation of two programs written in different languages but built by the same compiler suite.

[3] Brenda S. Baker. “On finding duplication and near-duplication in large software systems.” In: *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*. IEEE. 1995, pp. 86–95.

> Presents *Dup*, a tool for locating instances of duplication or near-duplication in a large software system.

[4] Boumediene Belkhouche, Anastasia Nix, and Johnette Hassell. “Plagiarism detection in software designs.” In: *Proceedings of the 42nd annual Southeast regional conference*. ACM. 2004, pp. 207–211.

> Focuses on very high-level comparison, analyzing design (code structure, data structures) of two programs for similarity. Performs progressively higher-level analysis on C programs at five levels of abstraction. Very language-specific approach.

[5] Bradley Beth. “A Comparison of Similarity Techniques for Detecting Source Code Plagiarism.” In: (2014).

> Beth measures the performance of four approaches to plagiarism detection against a simulated corpus of plagiarized C programs using five distinct obfuscation techniques: comment alteration, whitespace padding, identifier renaming, code reordering, and refactoring algebraic expressions. The project measures the effectiveness of Levenshtein edit distance (in source and in the LLVM intermediate representation bitcode), tree edit distance in the abstract syntax tree, graph edit distance in the control flow graphs, and w-shingling, both in the source and in the IR bitcode. The algorithms are also checked against an unrelated piece of source code to check for false positives, and their performance was also compared with the performance of MOSS.

The author limited the corpus to C programs, but believes the results will be similar for any compiler that uses the LLVM toolchain. The corpus is, however, very small, and perhaps not very realistic. The results are certainly not conclusive, but they provide a decent starting point.

It seems that most approaches detect some attacks well, but not others. The author had comments and whitespace removed before running the checks, so both of those approaches did poorly. It seems that changes to order and nomenclature are best detected when checking compiler-generated structures rather than the source itself. w-shingling against the LLVM Intermediate Result performed "the best."

"w-shingling measures the proportion of n-gram sequences two documents have in common to the total number of n-gram sequences that occur in either document." Beth speculates that MOSS uses a similar n-gram fingerprinting retrieval system called winnowing.

[6] Kevin W. Bowyer and Lawrence O. Hall. "Experience using 'MOSS' to detect cheating on programming assignments." In: *Frontiers in Education Conference, 1999. FIE'99. 29th Annual.* Vol. 3. IEEE. 1999, 13B3–18.

Old paper (1999) but very relevant as it directly discusses plagiarism in a programming class setting. Describes MOSS, a web-based plagiarism detection service still made available by Stanford (upwards of a decade after initial inception). MOSS does not provide details of algorithms used internally, does not provide source code, but does provide a useful benchmark for usability given its popularity.

[7] Sergey Brin, James Davis, and Hector Garcia-Molina. "Copy detection mechanisms for digital documents." In: *ACM SIGMOD Record.* Vol. 24. 2. ACM. 1995, pp. 398–409.

Presents COPS, a fingerprinting copy detection approach based on chunking. Brin, et al., note that chunking approaches cannot make use of the underlying structure of the document. A chunk is simply a group of units that do have some structural meaning (units are remarkably similar to "tokens" as we describe them in other parts of the literature).

Brin et al. present four chunking strategies:

1. One unit equals one chunk
2. Non-overlapping chunks of multiple units
3. Multiple-unit chunks with maximum overrlap
4. Using the smallest chunk whose hash is divisible by some number p.

One unit equals one chunk was rejected for its high space requirement. Non-overlapping chunks were rejected because of their phase dependence. All-overlapping chunks were rejected for having the same space requirement as one-unit chunks, and because an attacker could defeat it by changing one unit in each chunk. Expanding chunks until the chunk hash is divisible by some number p; note the similarity to document fingerprinting approaches. The authors choose the last strategy.

[8] Steven Burrows, Seyed M.M. Tahaghoghi, and Justin Zobel. “Efficient plagiarism detection for large code repositories.” In: *Software: Practice and Experience* 37.2 (2007), pp. 151–175.

Focuses on efficiency over large data sets. Details indexing algorithms adapted from genomic information retrieval for maintaining a database of source code that new submissions can be compared against, in a very efficient manner (so as to scale to many thousands or tens of thousands of submissions).

[9] E. Buss et al. “Investigating reverse engineering technologies for the CAS program understanding project.” In: *IBM Systems Journal* 33.3 (1994), pp. 477–500.

Presents various reverse engineering assistance techniques designed to assist with program understanding. Used IBM SQL/DS, a large tool written in the proprietary PL/AS language, as a reference system. The program undertook seven top-level goals:

1. Detect uninitialized data, pointer errors, and memory leaks
2. Detect data type mismatches
3. Find incomplete uses of record fields
4. Find similar code fragments
5. Localize algorithmic plans
6. Recognize inefficient or high-complexity code
7. Predict the impact of change.

The heading “Pattern Matching” in the background was fairly useful. They mention a few methods:

- Text analysis: The huge advantage of text analysis, of course, is that it is language-independent. Interesting quote: “For some understanding purposes, less analysis is better; syntactic and semantic analysis can actually information content in the code, such as formatting, identifier choices, white space, and commentary. Evidence to identify instances of cut-and-paste is lost as a result of syntactic analysis.” The text analysis research occurred at IBM and was done by Johnson, who coauthored this paper as well as others in this bibliography.
- Syntactic analysis: This research occurred at the University of Michigan and did not seem to be interested in detecting cloned code. However, the results might be at least somewhat applicable. The authors complain about existing (mostly text-based and graph-based) search tools, and then present SCRUPLE, a pattern-based query system with a powerful pattern language that allows the programmer to search, for instance, for three nested loops in order to find a matrix multiplication.
- Semantic analysis: This research occurred at McGill University, and clone detection was just one of their goals. The McGill research focuses on representing code as a vector of complexity metrics; close vectors are more likely to be similar (this is a classic vector distance algorithm). The McGill research uses five metrics:
  1. Number of functions called
  2. Ratio of input-output variables to fanout
  3. McCabe’s cyclomatic complexity
  4. Albrecht’s function-point quality metric

5. Henry-Kafura's information flow quality metric

The researchers measure distance in two ways: by Euclidean distance, and by clustering thresholds on each axis. Needless to say, this approach is language-dependent.

Text analysis found 727 copied lines out of a 51 655 line sample of source code from SQL/DS. The processing took two hours. They say that subgroup's research is now focused on finding approximate matches, implying to me that their findings include only exact matches. Syntactic analysis was not used to detect program similarity, and no quantitative results were presented for semantic analysis. The authors present a few qualitative takeaways:

1. Domain-specific knowledge is critical in easing the interpretation of large software systems
2. Program representations for efficient queries are essential
3. Many kinds of approaches are needed in a comprehensive reverse engineering approach
4. An extensible approach is needed to consolidate these diverse approaches into a unified framework

They closed with a discussion of how they integrated their tool into SQL/DS, which is not relevant to this project.

[10] Xin Chen et al. "Shared information and program plagiarism detection." In: *Information Theory, IEEE Transactions on* 50.7 (2004), pp. 1545–1551.

Attempts to "take a step back" and develop a universal measure for the amount of information shared between two sequences, be they DNA, text, or source code, which can then be used to make a determination on plagiarism. However, to make use of this algorithm, the program must be parsed into tokens to remove whitespace issues (amongst other reasons). Solution is named SID — Software Integrity Diagnosis.

[11] Vic Ciesielski, Nelson Wu, and Seyed Tahaghoghi. "Evolving similarity functions for code plagiarism detection." In: *Proceedings of the 10th annual conference on Genetic and evolutionary computation.* ACM. 2008, pp. 1453–1460.

Discusses the use of genetic algorithms to tune an existing algorithm for similarity evaluation (Okapi) for optimum accuracy, and furthermore uses particle swarm genetic optimization to devise novel formulas for plagiarism detection.

[12] Paul Clough et al. "Old and new challenges in automatic plagiarism detection." In: *National Plagiarism Advisory Service, 2003; http://ir.shef.ac.uk/cloughie/index.html*. Citeseer. 2003.

TODO.

[13] Georgina Cosma and M.S. Joy. "Source-code plagiarism: A UK academic perspective." In: (2006).

A survey of UK academics focused not on how to detect plagiarism, but what it is in the context of source code and programming classes.

[14] Maxime Crochemore et al. "A fast and practical bit-vector algorithm for the longest common subsequence problem." In: *Information Processing Letters* 80.6 (2001), pp. 279–285.

Efficient solution to Longest Common Subsequence problem, which has important implications for plagiarism detection (though it cannot cope with comments, whitespace, etc on its own).

[15] Mark Gabel, Lingxiao Jiang, and Zhendong Su. "Scalable detection of semantic clones." In: *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on.* IEEE. 2008, pp. 321–330.

This presents a scalable approach to identifying "semantic codes"— semantically equivalent source code blocks (here presented in the context of the detection of dead/redundant code, but plagiarism applications are obvious). Reports that fingerprinting is not as good as some other methods — the conclusion mentions the existing "identity algorithm" is more accurate in their testing.

[16] Robert W. Irving. "Plagiarism and Collusion Detection using the Smith-Waterman Algorithm." In: *University of Glasgow* (2004).

A similarity detection algorithm for plaintexts intended for plagiarism detection. According to conclusion, very accurate, but slow — perhaps too slow for anything but very small batches of files.

[17] J. Howard Johnson. "Substring Matching for Clone Detection and Change Tracking." In: *Software Maintenance, 1994. Proceedings, International Conference on.* ieee. 1994, pp. 120–126.

Presents a tool for locating similarities in text, including source code. The tool works in six steps:

1. Perform text-to-text transformations on each file
2. Break the text into potentially overlapping substrings
3. Generate a database of "raw matches" by finding the substrings that match
4. Iterate to describe the matches more concisely
5. Perform task-specific data reduction
6. Summarize high-level matches

Steps 2, 3, and 4 work only on exact matches, so any partial matching must be done via normalization in step 1. Common transformations include white space removal, comment removal, and identifier renaming. Steps 2-4 have the advantage of being language-agnostic. Note that this approach is reminiscent of the document fingerprinting approaches to plagiarism detection. Step 3 is done by karp-rabin string matching [20]. In step 4, they perform tasks such as merging consecutive matches and other "lossless" compression strategies. Step 5 is where one might perform "lossy" compression, such as eliminating certain text (like, for instance, a copyright notice) that is expected to be cloned.

The authors tested their prototype on gcc, versions 2.5.8 and 2.3.3; both releases together total 1440 files with a combined size of 40 megabytes. They found a total of 988 "clusters," or matched substrings. 315 of these clusters were of type "abx" (contained in one file from each release) or type "ab=" (contained in one file from each release with the same name). These represent code that was not changed between releases, or that was moved to a different location. There were a very large number of abx clusters as a result of a large naming convention change done by the gcc team between the two releases. They identified some software cloning, and areas of massive changes. The authors reported very few nonsense matches.

[18] Edward L. Jones. "Metrics Based Plagarism Monitoring." In: *Journal of Computing Sciences in colleges*. Vol. 16. 4. consortium for computing sciences in colleges. 2001, pp. 253–261.

A developed example of "feature comparison" — creates and examines "profiles" of program features (line count, number of unique tokens, average line length, number of spaces, that sort of thing). No evidence is presented that it is actually effective, and indeed they do not test on real-world data (only note that they intend to use it in their own courses).

[19] Mike Joy and Michael Luck. "Plagiarism in Programming Assignments." In: *Education, IEEE transactions on* 42.2 (1999), pp. 129–133.

Joy and Luck provide a definition of plagiarism, "unacknowledged copying of documents or programs," and name several potential causes of plagiarism.

Joy and Luck describe two obfuscation techniques: lexical changes (comment changes, formatting, changing identifier names, etc.) and structural changes (loop replacement, ifs to cases, statement ordering, refactoring, etc.)

They describe two pair comparison techniques: comparing attribute counts, and comparing structure.

They present an algorithm called sherlock, with the following requirements:

- must be reliable
- must be simple to change for a new language
- must have an "efficient interface"
- output must be clear to somene unfamiliar with the programs

incremental comparison compares five times: in original form, with whitespace removed, with comments removed, with both removed, and tokenized. looks for "runs" with a maximum allowed size and density of "anomalies." looks for and reports maximum length runs.

Presents a very interesting visualization with a point for each submission and similarities connected by lines; shorter lines correspond to closer matches.

[20] Richard M. Karp and Michael O. Rabin. "Efficient Randomized Pattern-Matching Algorithms." In: *IBM journal of research and development* 31.2 (1987), pp. 249–260.

This is a seminal paper that is cited by almost every article in this bibliography. It seems that almost all program similarity detection tools use karp-rabin string matching to search for and verify matches.

Presents a generalized string-matching algorithm that works in "real time" (each bit of input can be processed as soon as it comes in and requires constant time) and in a constant number of registers. It requires keeping a substring in memory of the same length as the one you are searching for. Authors claim that it seems to be competitive on classical strings only for larger substring sizes, but it has a huge advantage in being able to search two-dimensional arrays, higher dimensional arrays, and even irregular shapes with the same mathematical background.

"The idea of using fingerprinting techniques for string-matching problems is not new. Many such techniques based on check sums and hash functions can be found in the literature. What is new is the particular way of choosing the fingerprinting functions at run time. This randomization technique permits us to establish very strong properties of our algorithms, even if the input data are chosen by an intelligent adversary who knows the nature of the algorithm."

First presents a generalization of all string matching problems, and explains how the simple pattern-matching problem fits that framework.

Karp and Rabin present three algorithms for deciding whether there is a match given an input string X of length n, an output string Y, a finite set of fingerprinting functions S, and a set of valid indices R. Brief descriptions follow:

- Computes k fingerprinting functions on each substring of length n in Y. If they all report a match, halts immediately. This algorithm can report a false match, but only if all k fingerprint functions collide.
- Computes only one fingerprint function on each substring of length n in Y, but goes back and verifies the string after a match; false matches are caught and discarded.
- Same as above, but changes to a different fingerprinting function after a false match.

The rest of the paper introduces and thoroughly explores the properties of several fingerprinting functions. The details are outside the scope of this project.

[21] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. "A Formal Investigation of *diff3*." In: *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*. Springer, 2007, pp. 485–496.

A discussion of diff3, a three-way version of the conventional diff algorithm. This could be used for plagiarism detection (detect similarities between two files that are not shared by a third, given reference code shared by all students).

[22] Jens Krinke et al. "Distinguishing Copies from Originals in Software Clones." In: *Proceedings of the 4th international workshop on software clones*. ACM. 2010, pp. 41–48.

Another paper that doesn't really solve the plagiarism problem, and instead attempts to find duplicate/dead code. This one is interesting because of its categorization metrics,

though — it attempts to classify code as either a straight duplicate, close copy, or unclassifiable (some duplicated code, but not enough to conclusively classify).

[23] Thomas Lancaster and Fintan Culwin. "Classifications of Plagiarism Detection Engines." In: *Innovation in Teaching and Learning in Information and Computer Sciences* 4.2 (2005).

Lancaster provides a detailed taxonomy of plagiarism detection tools, and describes all (then) commonly-known examples in terms of that taxonomy.

[24] Mummoorthy Murugesan et al. "Efficient Privacy-Preserving Similar Document Detection." In: *The VLDB Journal; The International Journal on Very Large Data Bases* 19.4 (2010), pp. 457–475.

Attempts to detect similar documents when the text of the document is not available, for instance, when checking for plagiarism between conferences with confidential systems.

[25] Alan Parker et al. "Computer Algorithms for Plagiarism Detection." In: (1989).

Defines a five-level plagiarism "spectrum" to categorize the different types of obfuscations that are commonly used in duplicated student assignments, then provides an overview of some then-advanced similarity detection algorithms. All seven algorithms examined are feature-extraction algorithms that analyze similarities in software metrics.

[26] Martin Potthast et al. "An Evaluation Framework for Plagiarism Detection." In: *Proceedings of the 23rd International Conference on Computational Linguistics: Posters*. COLING '10. Beijing, China: Association for Computational Linguistics, 2010, pp. 997–1005. URL: `http://dl.acm.org/citation.cfm?id=1944566.1944681`.

Potthast et al. formalize a plagiarism as a 4-tuple consisting of the plagiarizing document, the copied document, and the plagiarized and original passages within each. They then explain that it is impossible to find an adequate source of "true" plagiarized material for a number of valid reasons, and describe three ways of generating a corpus: pay humans to plagiarize, use sources of legitimately copied material such as wire stories, or use an algorithm to mutate the document.

They present PAN-PC-10, a plagiarism corpus created with Mechanical Turk and an algorithmic approach. They compare the corpus with existing corpora Clough09 and METER, but stop short of claiming that any one database is the best.

[27] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. *JPlag: Finding plagiarism among a set of programs*. Tech. rep. Technical Report 2000-1, Fakultat fur Informatik, Universitat Karlsruhe, D-76128 Karlsruhe, Germany, 2000.

Presents JPlag, a similarity detection tool based on greedy string tiling, but with optimizations to decrease the average runtime from $O(n^2)$ to $O(n)$. Jplag achieves results comparable to MOSS, but provides a web interface rather than an email-based one.

[28] 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*. ACM. 2003, pp. 76–85.

Schleimer et al. present a document fingerprinting algorithm called winnowing, and describe its use in Stanford's MOSS service. They describe the concept of "k-gram filtering," where a document of n tokens is described as a sequence of $(n - k + 1)$ overlapping k-grams, with a k-gram being a sequence of k tokens. In k-gram filtering, each k-gram is hashed, and stored, with its document ID and location, in a lookup table. The k-grams are reduced to a smaller list using a filtering algorithm, and future documents can be checked against this corpus of document "fingerprints."

Our current LineCompare code, described in 5.4.1 is an implementation of k-gram filtering; in LineCompare, each line is a token, a document is represented as a sequence of 1-grams, and the 1-grams are filtered using the identity function.

According to the paper, current (at publication) k-gram filters suffer from a number of disadvantages. The biggest one is that the filter used is typically a mod-p filter; a mod-p filter accepts a k-gram x if $H(x)$ is congruent to zero mod p. Mod-p filters are weak because the fingerprints selected from the document are uneven — there could be huge runs of n-grams that do not hash to zero mod p. In principle, the maximum "gap width" in a document is unbounded, and in practice it is often longer than most web pages. Mod-p especially chokes on low-entropy data — a long string of zeroes, for instance, will either go completely unfingerprinted, or fingerprinted every single time.

The paper's contribution is winnowing, a k-gram filter that guarantees an upper bound on the distance between fingerprints in a document. That means that a copy that is longer than the maximum gap width is guaranteed to be detected. Winnowing has achieved widespread adoption, including by MOSS, and its merit has caused this paper to accumulate 711 citations on Google Scholar.

Schleimer, et al. introduce the concept of a "local algorithm," an algorithm that selects a document fingerprint from a "window" of consecutive k-grams with length w. An algorithm is local if it meets two conditions:

1. For each possible window, the algorithm selects at least one fingerprint from within that window, and
2. The choice depends only on the contents of that window, not on any other.

The authors demonstrate that if two documents are compared with a k-gram filter using a local algorithm with window size w, the comparison will detect at least one k-gram from each shared substring of length $w + k - 1$. The minimum density (asymptotic proportion of fingerprinted k-grams to total k-grams) of a local fingerprint selection algorithm is $1.5(w + 1)$. Winnowing has an asymptotic density of $2/(w + 1)$, leading the authors to claim it is "within 33% of optimal."

The related work describes the Karp-Rabin algorithm, which finds occurrences of a substring in a larger string [20]. SCAM [29] uses vector distance between documents to find copies. Baker [3] presents a concept called "parameterized matches," which can rename parameters to be equal and more easily detect copies that way.

They ran winnowing ($w = 100$) and mod-50, an implementation of mod-p with $p = 50$, on random data and found that both selected approximately the same number of fingerprints per unit of data. Against a corpus of a half million web pages, they found that both came close to their expected fingerprint density, but that mod-50 was highly non-

uniform: mod-50 scanned a run of 29 900 non-whitespace, non-tag characters without selecting a fingerprint. Any duplication within those characters would be completely undetected by mod-50.

Winnowing ended up fingerprinting extremely densely in low-entropy data, so the authors presented a very minor adjustment called "robust winnowing" to correct.

[29] Narayanan Shivakumar and Hector Garcia-Molina. "SCAM: A copy detection mechanism for digital documents." In: (1995).

Presents SCAM, a vector distance approach to copy detection. Like previous work, SCAM breaks documents into chunks and then compares the chunks for overlap; but instead of chunking into sentences or paragraphs like previous work, SCAM chunks by words. Most previous work in this area simply compared the size of the overlap against the size of the document, but that doesn't work if you are chunking by words; so the authors propose a new similarity approach based on vector distances between word counts.

[30] Temple F. Smith and Michael S. Waterman. "Identification of common molecular subsequences." In: *Journal of molecular biology* 147.1 (1981), pp. 195–197.

Describes the Smith-Waterman algorithm for comparing genetic sequences. This algorithm produces an optimal global overlay between two strings drawn from any alphabet. This original paper is very focused on genetics research but the algorithm itself is a crucial part of Irving's paper on similarity detection.

[31] Geoff Whale. "Identification of program similarity in large populations." In: *The Computer Journal* 33.2 (1990), pp. 140–146.

Whale provides an overview of the state of similarity detection in the year 1990. The document contains a discussion of the prevalence and motivation for students to copy others' assignments, common techniques for obscuring unauthorized copying, the various metrics to evaluate similarity detectors and the poor state of evaluation methodologies, and a broad overview of the most common approaches at the time. The paper closes with a comparison of the effectiveness of the most popular approaches.

Appendices

A User Guide

Checksims is a tool for detecting source code similarities in an arbitrary number of user-provided programming projects. Its primary purpose is to flag potential cases of academic dishonesty in programming assignments. *Checksims* is not intended to detect academic dishonesty on its own, but rather to act as a tool to identify suspicious assignments for review by course staff.

Checksims accepts a number of submissions (programming assignments) as input, applies a tokenizer to transform each submission into a series of tokens, and then applies a pairwise similarity detection algorithm to all possible pairs of submissions. The results of the algorithm are then printed via an output strategy.

A.1 Installing Checksims

Checksims is distributed as an executable Java package (`.jar` file). As a Java application, *Checksims* is cross-platform and should run on any system capable of running a Java virtual machine (JVM). The provided Jar file is completely self-contained and requires no installation, and should be named as follows:

```
checksims-1.1.1-jar-with-dependencies.jar
```

Note that 1.1.1 represents the current version of *Checksims* at the time of this writing, and may be different for the version you receive.

Note that *Checksims* requires a Java 8 virtual machine. The latest version of the Oracle JVM is recommended, and can be found at the following URL:

```
https://www.java.com/en/download/index.jsp
```

A 64-bit processor and JVM are strongly recommended. Some *Checksims* detectors can consume a substantial amount of memory, potentially more than the 4GB maximum available to a 32-bit JVM. A 64-bit JVM can prevent a number of memory-related program crashes.

A.2 Running Checksims

Checksims is a command-line application, and is typically invoked from the operating system's shell or command prompt. The `.jar` file given can be run using Java as follows:

```
java -jar PATH/TO/CHECKSIMS_JAR.jar <ARGUMENTS>
```

It may be desirable to rename the provided `.jar` file or write a wrapping shell script to reduce the amount of typing required for this basic invocation.

A.2.1 Arguments

Checksims has two mandatory arguments: a single *glob* match pattern, and at least one directory to scan for submissions.

The *glob* match pattern is a shell-style match pattern used to identify files to include in submissions. Wildcard characters accepted by a shell are permitted; for example, providing a `*` does match every file in a submission, while `*.c` includes all C files. To ensure that these are not parsed by your shell, it is recommended to escape this pattern (typically using double quotes — `"*.c"` for example).

After the glob match pattern, one or more directories to search for submissions must be provided. *Checksims* assumes each subdirectory of these search directories is a submission. It identifies any files matching the given glob pattern within a submission directory, append all matching files together, and tokenize the collection. By default, it is not recursive and will only identify files in the submission directory, but not in any subdirectories. An argument is provided to enable recursion through subdirectories to generate submissions (see section A.2.1.1 for details). File names are discarded during this process, but the contents of all matching files will be present. Each submission is named for its *root* directory (that is, the subdirectory of a submissions directory); if a directory containing two subdirectories named “A” and “B” is provided as a search directory for submissions, two submissions named “A” and “B” will be created.

After creation, any empty submissions (no files found matching given pattern, or only empty files found) are removed prior to running the detection algorithm.

At present, there is no way of differentiating submissions beyond placing them within separate directories.

A.2.1.1 Optional Arguments Before the glob matcher, you may place a number of arguments to control the operation of *Checksims*. These are detailed below:

- `-a`, `-algorithm`: Specify algorithm to use for similarity detection. Available options are `linecompare` and `smithwaterman` at present, and can be listed with the `-h` option. If no algorithm is given, the default is used.

- `-c`, `-common`: Perform common code removal. Specify a directory containing common code (files within this directory will be identified using the same glob matcher as normal submissions).

- `-f`, `-file`: Output to a file. Must provide filename of output file as argument. The name of the output strategy used will be appended to the given filename as an extension. If more than one output strategy is given, more than one output file will be produced, each with the given filename but with differing extensions.

- `-h`, `-help`: Print usage information and available algorithms, preprocessors, and output strategies.

- `-j`, `-jobs`: Specify number of threads to use. Defaults to number of CPUs available on your system.

- `-o`, `-output`: Specify output format(s) to use. More than one can be provided; if so, separate them with commas. Available options are `html`, `csv`, and `threshold` at present, and can be listed with the `-h` option. If no output format is given, the default is used.

- `-p`, `-preprocess`: Specify preprocessors to apply. More than one can be provided; if so, separate them with commas. At present, the only available option is `lowercase`. Available options can be listed with the `-h` option. If this argument is not provided, no preprocessors are applied.

- `-r`, `-recursive`: Recursively traverse subdirectories when generating submissions.

- `-t`, `-token`: Specify tokenization to use. Available options are `line`, `whitespace`, and `character` at present, and can be listed with the `-h` option. If the `-t` option is not given, the default tokenization for the algorithm is used.

- `-v`, `-verbose`: Verbose debugging output.

- `-vv`, `-veryverbose`: Very verbose debugging output. Overrides `-v` if both are specified.

- `-version`: Print current version of *Checksims*.

Checksims contains built-in usage information and descriptions of its arguments, which can be printed by supplying the `-h` or `-help` flag. The output mirrors the information provided above, though it may be more up-to-date.

A.2.1.2 JVM Arguments A number of arguments can also be passed to the Java virtual machine. These are usually placed in the command line as follows:

```
java <JVM_ARGS> -jar PATH/TO/CHECKSIMS_JAR.jar <ARGS>
```

These arguments are well-documented and can be used on all Java virtual machines. Several commonly-used flags are detailed below.

- `-d64`, `-d32`: Specify a 64 or 32 bit JVM, respectively. Some JVMs will only support 32 or 64 bit, but not both. Using a 64-bit JVM where available is preferred to enable the VM to use more than 4GB of memory.

- `-Xmx`: Specify a maximum amount of memory for the JVM to use. The number can be formatted as [Amount][Unit] where [Unit] is M for megabyte or G for gigabyte. Note that the number is specified immediately after the flag, with no = character. For example, to set the JVM to use at most 4GB of ram, specify `-Xmx4G` at the command line.

A.2.1.3 Sample Command Line

A typical command line invocation of *Checksims* is shown below.

```
java -d64 -jar PATH/TO/CHECKSIMS_JAR.jar -a smithwaterman -o html,csv -v -r
          -f ./out "*.c"  SUBMISSION_DIR_ONE   SUBMISSION_DIR_TWO
```

This instructs *Checksims* to do the following:

- Use a 64-bit JVM to prevent memory issues (`-d64`).
- Use an algorithm named `smithwaterman` to perform similarity detection (`-a`).
- Generate output using two strategies, `html` and `csv` (`-o`), and save this output in two files names `out.html` and `out.csv` (`-f`).
- Perform similarity detection on all files with extension `".c"` in directories `SUBMISSION_DIR_ONE` and `SUBMISSION_DIR_TWO`.

A.2.2 Common Errors and Solutions

This section contains a number of common errors that can occur while using *Checksims*, and suggests potential fixes.

A.2.2.1 Out of Memory Errors

A Java Out of Memory exception occurs when *Checksims* uses all the memory available to the Java virtual machine. This is usually caused by running a complex comparison algorithm (for example, *Smith-Waterman*) on large submissions.

The first potential fix is to increase the amount of memory available to the JVM. This can be done by installing 64-bit Java and passing the `-d64` flag to use a 64-bit JVM (enabling the use of more than 4GB of memory). If a 64-bit JVM is already installed, a larger amount of memory can be provided using the `-Xmx` flag.

If more memory cannot be allocated to the JVM, it is also possible to reduce the amount of memory used by *Checksims*. This can be done in a number of ways. Firstly, reducing the number of threads used with the `-j` flag will cause a substantial decrease in the amount of memory used. Each thread uses roughly the same amount of memory, so a reduction from 4 to 2 threads should cause *Checksims* to use half as much memory. Furthermore, changing the tokenization used can impact the number of tokens stored, which has substantial implications for algorithm memory use. Changing from Character to Whitespace tokenization for *Smith-Waterman*, for example, will usually result in a 4-fold reduction in memory use.

A.2.2.2 No Submissions Detected

In the case that *Checksims* cannot build any student submissions to compare, the first step is usually to check the glob match pattern used. Ensure that any characters that might be interpreted by your shell (for example, `*`) are properly escaped (single or double quoted on Linux or OS X, double quoted on Windows). Furthermore, check that the glob match pattern is syntactically valid for your platform.

Verify that you are passing *Checksims* a directory containing a number of student submissions, each of which is contained in a single subdirectory of the directory passed to *Checksims*. Even if each student submission is a single file, it must be contained in a subdirectory. Student submission directories may contain subdirectories themselves without issue.

A.3 Description of Tokenizations

Checksims breaks submissions into a sequence of tokens as they are read in. Several options are provided, each providing a tradeoff of speed for performance. Each similarity detection algorithm provides a default tokenization that has been chosen to optimize its performance for typical usage, but this default can be overridden at runtime if desired. This may be desirable, as tokenization has strong implications for algorithm accuracy and performance.

Only one tokenization is supported at any given time; it is impossible to request that *Checksims* tokenize one submission using character tokenization, and another using whitespace tokenizations. This is done to ensure a uniform basis for token comparison.

Three tokenization options are provided by default: `Character`, `Whitespace`, and `Line`. Their advantages and disadvantages are listed below.

A.3.1 Character Tokenization

The simplest tokenization method, `Character` tokenization, breaks a submission into the characters that compose it and builds a token for each character. Whitespace characters (spaces and newlines) are treated as tokens. No deduplication of whitespace is done — if a submission contains three consecutive spaces, all will be treated as independent tokens.

`Character` tokenization has the slowest performance of all the tokenization schemes as it generates far more tokens for the algorithm to process. However, for most algorithms, `Character` tokenization will be the most conducive to accuracy, as it can identify largely similar words and lines that would otherwise be ignored. `Character` tokenization also uses slightly more memory to store compared to the other tokenization schemes — usually not enough more to cause problems. However, `Character` tokenization may have a more serious impact on the amount of memory used by certain algorithms (such as Smith-Waterman).

A.3.2 Whitespace Tokenization

`Whitespace` tokenization breaks a submission apart at whitespace characters (spaces, tabs, newlines) to create tokens. Whitespace characters are removed as part of the splitting process, and are not included as tokens.

`Whitespace` tokenization represents a balance between performance and accuracy. With preprocessing (lowercasing to remove case ambiguity, etc), it can retain much of the accuracy of `Character` tokenization while substantially improving performance (assuming `Whitespace` tokens are on average four characters, a fourfold reduction in token count can be expected, even ignoring the deletion of whitespace).

A.3.3 Line Tokenization

`Line` tokenization splits the submission at line boundaries, creating a token from each line in the original. Non-newline whitespace characters (spaces and tabs) are retained.

`Line` tokenization represents the fastest but least precise tokenization option. It is capable of identifying exact duplication, but even trivial attempts to obfuscate similarities will prevent detection.

A.4 Description of Preprocessors

After a submission is converted into tokens, these tokens can then be manipulated to improve detection accuracy. This is accomplished by the use of predefined preprocessors. Two preprocessors are presently available. The `lowercase` preprocessor converts all letters to lowercase. The `deduplicate` preprocessors remove duplicated whitespace (spaces, tabs, and newlines).

A.5 Description of Algorithms

Checksims provides two detection algorithms at present. The first is *Smith-Waterman*. It offers accurate detection but slow performance. The second is *Line Comparison*. It is very fast, but not very accurate and easily fooled by obfuscation.

A.5.1 Smith-Waterman

The *Smith-Waterman* algorithm for string overlaying was originally developed to find optimal local alignments between DNA sequences for bioinformatics problems. Adapted to handle arbitrary alphabets, it proves a valuable tool for identifying similar token sequences. As a local alignment algorithm, it is capable of detecting sequences even when they are not completely identical. A small number of missing or unmatched tokens are tolerated, identifying more similarities than simply finding the longest common sequence. Furthermore, *Smith-Waterman* is guaranteed to identify the optimal local alignment — if common sequences exist, they will be found.

However, *Smith-Waterman*'s accuracy comes at a substantial performance cost. The algorithm itself is $O(n{*}m)$ where n and m are the lengths of the two sequences being compared; assuming equal and even growth of both sequences, the algorithm scales as roughly $O(n^2)$ (both for runtime and memory). For smaller submissions, Smith-Waterman can complete an entire class in a few minutes; for larger submissions, however, hours (or even days) may be required.

Because of the performance penalty of *Smith-Waterman*, it is recommended to use it with the `Whitespace` tokenization scheme, which it defaults to. This reduces the number of tokens present, greatly improving performance.

A.5.2 Line Comparison

The *Line Comparison* algorithm identifies identical tokens in both submissions. It is a trivial algorithm unique to Checksims, and notable for its speed. *Line comparison* hashes each input token, and identifies hash collisions (identical tokens). Similarity is reported on the number of collisions detected between the two submissions.

Line comparison makes one pass through each submission, and thus is $O(m + n)$. It is thus far faster than *Smith-Waterman*.

As the name of the algorithm indicates, it is only intended to be used with (and defaults to) the `Line` tokenization scheme. `Whitespace` tokenization results in a percent of shared words contained in submissions that is almost always very high and does not mean much about the actual similarity of two submissions. `Character` tokenization tends to result in greater than 99% similarity for all submissions, given that most all will be using the same basic alphabet (capital and lowercase letters, numbers, and language-appropriate syntax such as { or [).

Given the restriction to the use of `Line` tokenization, even small changes (for example, a single missing character) can result in otherwise extremely similar lines not being recorded as similar. It is possible that preprocessors could remove some trivial differences (for example, changes to whitespace or addition of comments). However, other alterations, like reordering statements or changes to identifier names, are very difficult to catch with preprocessors. *Line Comparison* is thus very limited in terms of accuracy.

A.6 Description of Output Strategies

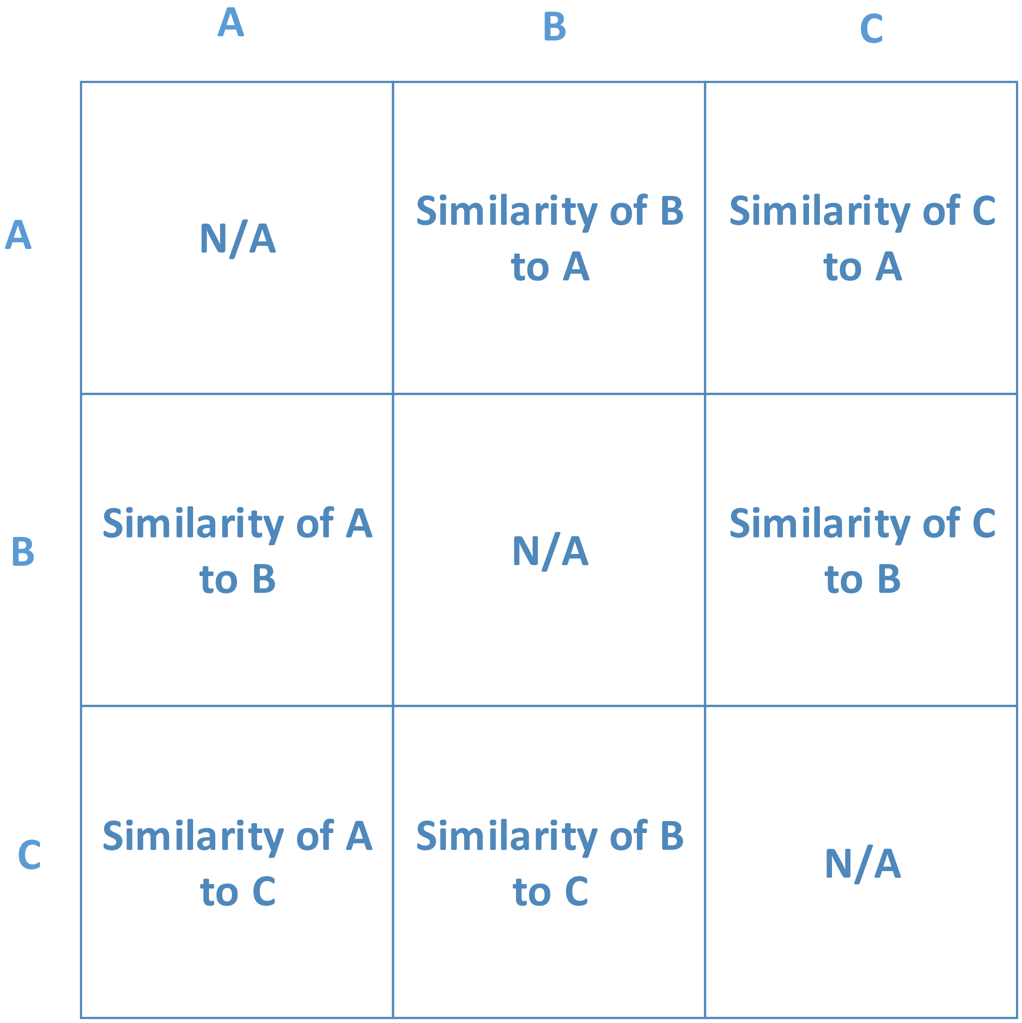
Once an algorithm has been applied to the submissions, the results must be printed in a usable format so they may be used and interpreted. Output strategies determine how this is done.

Results will often be presented as a square matrix, henceforth referred to as a *Similarity Matrix*. These matrices are built from the complete results of the similarity detection algorithm, and contain the similarity of every submission to every other. As the name would imply, a similarity matrix is a $N \times N$ matrix (with N being the number of submissions), with each cell containing a number representing the degree of similarity of one submission to another.

In a similarity matrix, the submissions used in similarity detection are counted, and a square matrix of that dimension is created. Submissions are each assigned a row and column. Every cell is initialized as the degree of similarity of the submissions that define its intersection (specifically, column submission's similarity to row submission). If the row and column submissions are the same, the cell is ignored (declared as empty). An example is shown in Figure A.1. Each cell shows the similarity of the submission on the X axis with the submission on the Y axis. In Figure A.1, the bottom-left corner cell shows the percentage similarity of submission A to submission C — that is, the proportion of submission A's tokens that are shared with submission C.

A.6.1 CSV

The CSV output strategy records output as a similarity matrix in comma-separated value format. This output format is computer-readable, not human-readable. It can be imported into Microsoft Excel or a number of other software statistics packages to generate statistics about detected similarities



	A	B	C
A	N/A	Similarity of B to A	Similarity of C to A
B	Similarity of A to B	N/A	Similarity of C to B
C	Similarity of A to C	Similarity of B to C	N/A

Figure A.1: A sample similarity matrix

A.6.2 HTML

The HTML output strategy produces a web page that can be opened in a typical web browser, presenting a colorized version of a similarity matrix. A color range (yellow to red) shows how similar each cell is, allowing easy visual identification of similar students and clusters of similarities.

A.6.3 Threshold

The threshold output strategy produces an ordered list of submission pairs that are sufficiently similar (by default, 60% or greater). The list is ordered from most to least similar, and omits all information about similarities below the threshold. This output strategy produces quickly actionable information about the most-similar submissions.

B Developer Guide

This guide is intended to serve as an introduction to *Checksims* to introduce new developers to the codebase. It is hoped that this will make contributing to the codebase easier and more accessible.

B.1 Obtaining the Source

The source code for *Checksims* is available on *Github*, at the following URL:

`https://github.com/mheon/checksims/`.

The source can be checked out using Git via the instructions located on the Github webpage, which are duplicated here for convenience:

```
git clone https://github.com/mheon/checksims.git
```

B.2 Building Checksims

Building *Checksims* has two requirements: a Java 8 JDK, and version 3 of the *Apache Maven* build system. The following assumes familiarity with Maven and its capabilities. The commands below must be run in the root of the cloned repository. All paths are relative to this directory.

`https://maven.apache.org/download.cgi`

Maven itself is written in Java, and the provided Jar files should be cross-platform.

Note that JDK 8 or later is a buildtime and runtime requirement. Earlier versions of the JDK will not be able to build and run Checksims.

Most testing was performed using the Oracle JDK across Linux and OS X, but other configurations (Linux/OS X + OpenJDK 8, Windows) should be supported without issue.

Checksims uses the typical Maven lifecycle commands for building. Again, a full review of Maven's capabilities is outside the scope of this document, but critical commands for building and testing will be briefly reviewed. Please note that all commands shown below are assumed to be run in the root of the cloned repository, and all paths given are relative to this directory.

To run unit tests:

```
mvn clean test
```

To build an executable `.jar` file:

```
mvn clean compile package
```

Build artifacts (executable `.jar` files) will be placed in the `target/` directory. By default, two `.jar` files will be produced: one with, and one without, library dependencies. They are easily differentiated — the `.jar` with dependencies included will be named as follows:

```
checksims-VERSION-with-dependencies.jar
```

Where VERSION is the current version number (1.1.1 at time of this writing).

B.3 Contributing: Modifying the Source

Checksims is fairly well documented and ships with unit tests, to enable ease of modifications. Furthermore, measures have been taken to make the addition of new algorithms, preprocessors, and output strategies especially easy. This section details the project structure to ease understanding of the source code and provide the location of critical project components.

B.3.1 Directory Structure

Checksims is structured as a typical Maven project; again, full description of this format is beyond the scope of the document, but important locations will be summarized.

All source code (production and test) is contained in the `src/` directory. Test-only code is found in `src/test/java/`, while main code is located in `src/main/java`. Runtime resources (non-Java source files, for example the template for generating HTML output) are located in the `src/main/resources/` directory.

Source files are contained in package-appropriate subdirectories of these main directories. In the case of a source file named `ChecksimRunner.java` in package `edu.wpi.checksims`, the location of this file would be:

```
src/main/java/edu/wpi/checksims/ChecksimRunner.java
```

Most Java IDEs should have the ability to import a Maven project; this should pick up all source directories automatically.

B.3.2 Source Structure

This section contains a description of the package and source structure of the project.

B.3.2.1 Package Structure The root package for *Checksims* is `edu.wpi.checksims`, with a number of subpackages, described below.

The `algorithm` subpackage contains core similarity detection functionality. All similarity detection algorithms (and the root interface `SimilarityDetector`, which all detection algorithms implement) are contained within `algorithm`. Algorithms at time of writing (*Line Similarity* and *Smith-Waterman*) are contained within appropriately-named subpackages (for example, `algorithm/smithwaterman` for the *Smith-Waterman* algorithm). It is intended that the same scheme will be used for future algorithms. The overall `algorithm` package also contains the `output` subpackage, containing all valid output strategies and the `SimilarityMatrixPrinter` interface, which all output strategies implement. Finally, the `algorithm` package also contains the `preprocessor` subpackage, containing all supporting preprocessor algorithms for token lists.

The `submission` subpackage contains code relating to the creation of submissions from directories and files on disk. All concrete classes within are `final`; this part of the interface is considered stable, and should not require much modification or extension.

The `token` subpackage contains code relating to tokenization of files and tokens generated. Several data structures for tokens, including `TokenList` (implementing `List<Token>`) and trees of arbitrary arity (contained in the `tree` subpackage), are included here. Concrete implementations of tokens themselves are `final`; again, this is considered a stable part of the interface.

The `util` subpackage contains general utility code.

B.3.2.2 Code Structure The entry point of *Checksims* is `ChecksimRunner` in the root package. This class contains both argument parsing and the `runChecksims` method, which controls the core functionality of the program.

`runChecksims` begins by initializing a list of submissions from the given directory, generating one submission per subdirectory of the given assignment directory(s). After the list of all submissions has been built, common code detection and removal is performed (if requested by the user). Preprocessors are then applied, sequentially, on each submission. Finally, `SimilarityMatrix.generate()` is called, creating a results matrix.

`SimilarityMatrix.generate()` generates a list of all unique unordered pairs of student submissions, then applies a given pairwise similarity detection algorithm to each submission. The results are then collated to produce the *similarity matrix*: a square array of floats, representing the similarity of each assignment when compared with each other assignment.

The generated similarity matrix is then passed to an output strategy to generate human-readable results. These are then printed (to `STDOUT` or, if requested, to a file), and the program exits.

Checksims includes a number of library dependencies. The most important of these are the *Apache Commons* and *Google Guava* libraries. These do overlap to an extent, but utilities from both are used throughout. In cases where both libraries provide overlapping functionality, the version provided by Apache Commons was preferred, though this is not a hard rule — if the API offered by Guava was felt to be superior, it was used instead. Finally, the *Apache Velocity* templating library is including to make the creation of complex output possible (for example, the included HTML output strategy). As in all Maven projects, the build process and dependencies are controlled by the `pom.xml` file in the root of the repository.

B.3.3 Adding an Algorithm

This section details the process of adding a similarity detection algorithm, though the process is very similar for a preprocessor or output strategy.

All similarity detection algorithms must be contained in package `algorithm` or a subpackage thereof. The core interface is `SimilarityDetector`, requiring three methods:

- `getName()`
- `getDefaultTokenType()`
- `detectSimilarity`

The `getName()` method returns the name of the algorithm, as the user will type it in on the command line. This must be a unique string (no two algorithms may have the same name), and it is highly recommended that it not contain spaces.

The `getDefaultTokenType()` method returns the default tokenization used by the algorithm. There are three supported tokenizations in *Checksims* at the time of this writing: `Line`, `Whitespace`, and `Character`. Each breaks submitted files up differently (by newline, by whitespace character, and into individual characters) to produce tokens. The returned token type from this method is the default for this algorithm (this can be overridden by the user at runtime, however).

The `detectSimilarity()` method accepts a pair of student submissions, performs similarity detection, and returns an `AlgorithmResults` object representing the results of the detection. Exceptions are permitted to be thrown on internal algorithm errors.

Algorithms must also implement a static `getInstance()` method taking no arguments and returning an instance (preferably a singleton) of the algorithm. All detection algorithms are automatically loaded at runtime using reflection using this method. As a result of this, no work is required to make an algorithm available outside of writing it and placing it in the `algorithm` package; all implementors of `SimilarityDetector` will be scanned via reflection and loaded automatically.

Similarity detection is multithreaded by default, and is performed using a single instance of the similarity detector. Consequently, it is not advised to have class-level mutable state or `synchronized` methods within a similarity detector. Instead, mutable state should be confined to the `detectSimilarity()` method and any helpers. The overall implementation of a similarity detector must be thread-safe.

C Anonymization Script

Listing 1: Anonymization Script Source

```bash
#!/bin/bash
# This script has ABSOLUTELY NO ERROR CHECKING
# But it shouldn't overwrite anything in typical case

student=0
srcdir="$1"
dstdir="$2"

mkdir -p "$dstdir"

find "$srcdir/students" "$srcdir/groups" -mindepth 1 -maxdepth 1 -type d -not -path "$srcdir/students" -not -path "$srcdir/groups" -print0 | while read -d $'\0' dir
do
  studentName=`basename "$dir"`
  echo "Anonymizing␣student␣$studentName"
  curDstDir="$dstdir/student_$student"
  # Make output directory
  mkdir -p "$curDstDir"
  # Increment student number
  student=$((student+1))

  curDir=`pwd`

  # Unzip anything we find. Ignore errors that might occur because there are no zip files.
  find "$dir" -type f -name '*.zip' -print0 | while read -d $'\0' zip
  do
    dirName=`dirname "$zip"`
    echo "Unzipping␣$zip␣to␣$dirName"
    unzip -o -d "$dirName" "$zip"
  done

  # Untar any tars
  find "$dir" -type f -name '*.tar' -print0 | while read -d $'\0' tar
  do
    dirName=`dirname "$tar"`
    cd "$dirName"
    echo "Untarring $tar to $dirName"
    tar -xf "$tar"
  done

  cd "$curDir"

  # Loop through all .c and .h files in that directory and anonymize them
  find "$dir" \( \( -type f -name '*.c' \) -or \( -type f -name '*.h' \) -or \( -type f -name '*.cpp' \) -or \( -type f -name '*.hpp' \) \)  -print0 | while read -d $'\0' file
  do
    # Run strip_comments script from GNU folks. Pipe output into output file.
    fileBasename=`basename "$file"`
    echo "Stripping comments from $file, outputting to $fileBasename"
    ./strip_comments.sed "$file" > "$curDstDir/$fileBasename"
  done
done

echo "Done!"
```