Shared Information and Program Plagiarism Detection

Xin Chen, Brent Francia, Ming Li, *Member, IEEE*, Brian McKinnon, and Amit Seker

***Abstract*—A fundamental question in information theory and in computer science is how to measure similarity or the amount of shared information between two sequences. We have proposed a metric, based on Kolmogorov complexity, to answer this question and have proven it to be universal. We apply this metric in measuring the amount of shared information between two computer programs, to enable plagiarism detection. We have designed and implemented a practical system SID (Software Integrity Diagnosis system) that approximates this metric by a heuristic compression algorithm. Experimental results demonstrate that SID has clear advantages over other plagiarism detection systems. SID system server is online at http://software.bioinformatics.uwaterloo.ca/SID/.**

***Index Terms*—Kolmogorov complexity, program plagiarism detection, shared information.**

I. Introduction

A common thread between information theory and computer science is the study of the amount of information contained in an ensemble [18], [19] or a sequence [9]. A fundamental and very practical question has challenged us for the past 50 years: Given two sequences, how do we measure their similarity in the sense that the measure captures all of our intuitive concepts of "computable similarities"? Practical reincarnations of this question abound. In genomics, are two genomes similar? On the Internet, are two documents similar? Among a pile of student Java programming assignments, are some of them plagiarized?

This correspondence is a part of our continued effort to develop a general and yet practical theory to answer the challenge. We have proposed a general concept of sequence similarity in [3], [11] and further developed more suitable theories in [8] and then in [10]. The theory has been successfully applied to whole genome phylogeny [8], chain letter evolution [4], language phylogeny [2], [10], and, more recently, classification of music pieces in MIDI format [6] and extensions in database area [16]. In this correspondence, we report our project of the past three years aimed at applying this general theory to the domain of detecting programming plagiarisms.

A plagiarized program, following the spirit of Parker and Hamblen [15], is a program that has been produced from another program with trivial text edit operations and without detailed understanding of the program. It is a prevailing problem in university courses with programming assignments. Detecting plagiarism is a tedious and challenging task for university instructors. A good software tool would help the instructors to safeguard the quality of education. More generally, the methodology developed here has other applications such as detecting Internet plagiarism. Yet, the goal of this work goes beyond a particular application. Through these efforts, together with other work [3], [11], [8], [4], [10], [6], we hope to develop and justify a general and practical theory of shared information between two sequences.

Many program plagiarism detection systems have already been developed [1], [7], [20], [23]. Based on which characteristic properties they employ to compare two programs, these systems can be roughly grouped into two categories: attribute-counting systems and structure-metric systems. A simple attribute-counting system [14] only counts the number of distinct operators, distinct operands, total number of operators of all types, and total number of operands of all types, and then constructs a profile using these statistics for each program. A structure-metric system extracts and compares representations of the program structures; therefore, it gives an improved measure of similarity and is a more effective practical technique to detect program plagiarism [21]. Widely used systems, such as Plague [20], MOSS [1], JPlag [13], SIM [7], and YAP family [23] are all structure-metric systems. Such systems usually consist of two phases: the first phase involves a tokenization procedure to convert source codes into token sequences by a lexical analyzer; the second phase involves a method to compare those token sequences. Note that a basic problem underlying in the second phase of a structure-metric system is how to measure similarity of a pair of token sequences. An inappropriate metric lets some plagiarisms go unnoticed, and a well-defined but nonuniversal [9] metric can always be cheated.

Wise [22] presented three properties an algorithm measuring program similarity must have: a) each token in either string is counted at most once; b) transposed code segments should have a minimal effect on the resulting similarity score; c) this score must degrade gracefully in the presence of random insertions or deletions of tokens. It is disputable whether these three criteria are sufficient. For example, many other things should also have minimal effects: duplicated blocks, almost duplicated blocks, insertion of irrelevant large blocks, etc. In fact, there are simply too many cases to enumerate.

We will take a radically different approach. We will take one step back, away from specific applications, such as program plagiarism detection. We will look at an information-based metric that measures the amount of information shared between two sequences, any two sequences: DNA sequences, English documents, or, for the sake of this correspondence, programs. Our measure is based on Kolmogorov complexity [9] and it is universal. The universality guarantees that if there is any similarity between two sequences under any computable similarity metric, our measure will detect it. Although this measure is not computable, in this correspondence we design and implement an efficient system SID (Software Integrity Diagnosis system) to approximately calculate this metric score (thus, SID may be also be defined as Shared Information Distance). These are detailed in Sections III and IV-A. In Section II we first survey related work, and in Section IV we introduce our plagiarism detection system SID. Experimental results are given in Section V.

II. Related Work in Plagiarism Detection

This section surveys several plagiarism detection systems. Many such systems exist; here, we only review four typical and representative systems.

A token in this paper refers to a basic unit in a programming language such as keywords like "if, ""then, " "else," "while," standard arithmetic operators, parentheses, or variable types. A precise definition of tokens is available at http://software.bioinformatics.uwaterloo.ca/SID/TestDef.html. A parser parses a program into a sequence of tokens. We can naturally assume that all correct parsers return identical token sequences for the same program.

Manuscript received March 15, 2003; revised March 18, 2004. This work was initiated at Bioinformatics Laboratory, Computer Science Department, University of California, Santa Barbara, Santa Barbara, CA 93106, USA, and was supported in part by the National Science Foundation under ITR Grant 0085801 and REU Grant.

X. Chen is with the Department of Computer Science, University of California, Riverside, Riverside, CA 92502 USA (e-mail: xinchen@cs.ucr.edu).

B. Francia and M. Li are with the School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada (e-mail: mli@uwaterloo.ca).

B. McKinnon and A. Seker are with the Computer Science Department, University of California, Santa Barbara, Santa Barbara, CA 93106 USA.

Communicated by E.-h. Yang, Guest Editor.

Digital Object Identifier 10.1109/TIT.2004.830793

A. Attribute Counting Systems

The earliest attribute-counting-metric system [14] used Halsted's software science metrics to measure the level of similarity between program pairs, using four simple program statistics:

- $\eta_1 =$ number of distinct operators,
- $\eta_2 =$ number of distinct operands,
- $N_1 =$ total number of operator occurrences, over all distinct types,
- $N_2 =$ total number of operand occurrences, over all distinct types.

Two measure metrics calculated from the above are

- $V = (N_1 + N_2)\log_2(\eta_1 + \eta_2)$,
- $E = [\eta_1 N_2 (N_1 + N_2)\log_2(\eta_1 + \eta_2)]/(2\eta_2)$.

Whale [21] has demonstrated that a system based on attribute counters is incapable of detecting sufficiently similar programs.

B. Structure-Metric Systems

MOSS. Details of the algorithm used by the MOSS system [1] are not given to prevent circumvention. But it is believed to be a tokenizing procedure followed by a fast substring-matching procedure. Experimental results show that pairs of files are sorted by the size of their matched token blocks. The newly improved MOSS system uses a winnowing algorithm to select fingerprints [17].

YAP. A similarity score used in YAP system [23] is a value from 0 to 100, called percent-match, representing the range from "no-match" to "complete-match." It is obtained by the formulas

$$\text{Match} = (\text{same} - \text{diff})/\text{minfile} - (\text{maxfile} - \text{minfile})/\text{maxfile}$$
$$\text{PercentMatch} = \max(0, \text{Match}) * 100$$

where maxfile and minfile are the lengths of the larger and smaller of the two files, respectively, variable "same" is the number of tokens common to two files, and variable "diff" is the number of single-line differences within blocks of matching tokens.

Its aim is to find a maximal set of common contiguous substrings as long as possible, each of which does not cover a token already used in some other substrings.

SIM. The SIM plagiarism detection system [7] compares token sequences using a dynamic programming string alignment technique. This technique first assigns each pair of characters in an alignment a score. For example, a match scores 1, a mismatch scores -1, and a gap scores -2. The highest score of a block gives the score of an alignment. The score between two sequences is then defined to be the maximum score among all alignments, which is easily calculated via dynamic programming technique. With this definition a similarity measure between two sequences is defined as follows:

$$s = 2 * \text{score}(s,t)/(\text{score}(s,s) + \text{score}(t,t))$$

which is a normalized value between 0 and 1. It is known that this metric gets into trouble due to the incapability of dynamic programming technique to deal with transposed code segments.

III. Shared Information

In [11] and [3], we have defined the information distance between two sequences to be roughly the minimum amount of energy needed to convert one sequence to another sequence and *vice versa*. While information distance has nice properties, it is problematic for evolutionarily related sequences. For instance, in genomics, *E. coli* and *H. influnza* are sister species. However, despite their high genome similarity, the two genomes differ greatly in sizes: the former has about 5 million bases and the latter 1.8 million bases. Information distance defined in [3], [11] would more likely classify *H. influnza* with another unrelated but short species, say with 2 million bases, than *E. coli*.

A more useful information-based sequence distance to measure similarity between sequence pairs was proposed in [8]. It has been successfully applied to the construction of whole genome phylogenies [8], [5], chain letter evolutionary history [4], and language classification [2], [10].[1] Its definition is based on Kolmogorov complexity or algorithmic entropy [9]. The Kolmogorov complexity of string x, $K(x)$, measures the amount of absolute information the sequence x contains. That is, $K(x)$ is the length, in number of bits, of the shortest program that on empty input prints x. Given another sequence y, the conditional Kolmogorov complexity $K(x|y)$ measures the amount of information of x given y for free. By definition, $K(x|y)$ is the length of the shortest program that on input y prints x. We refer the reader to [9] for formal definitions and other applications of Kolmogorov complexity. An information-based sequence distance was then defined as

$$d(x,y) = 1 - \frac{K(x) - K(x|y)}{K(xy)} \tag{1}$$

where the numerator $K(x) - K(x|y)$ is the amount of information y has about x. A deep theorem in Kolmogorov complexity states that $K(x) - K(x|y) \approx K(y) - K(y|x)$, hence $d(x,y)$ is symmetric [9]. This information is also called mutual information between x and y. The denominator $K(xy)$ is the total amount of information in the concatenated string xy. While the mutual information is not a metric because it does not satisfy the triangle inequality, the distance $d(x,y)$ is a metric, up to a vanishing error term. It turns out that $d(x,y)$ so defined can be viewed as a normalization of a version of information distance defined in [11]

$$d(x,y) \approx \frac{K(x|y) + K(y|x)}{K(xy)}.$$

The normalization scales the distance to within the range $[0, 1]$. In [8], we have shown that $d(x,y)$ is a metric. A proper metric can guarantee that the smaller the value of the measured distance, the more likely the two compared programs are plagiarized.

Furthermore, in [8] (and [10]) we have proved a "universality" property for this distance in the sense that for any other computable distance function D, satisfying some very general conditions and normalized to range $[0, 1]$, for all strings x, y

$$d(x,y) \le 2D(x,y) + O(1).$$

In the context of program plagiarism detection, this can be interpreted as follows: if two programs are "similar" under any computable measure, then they are similar under our measure d. Thus, at least in theory, a software plagiarism system built based on such measure is not cheatable. For example, all three items stated by Wise [22, Sec. I], such as the transposed code segments, and those not stated by Wise, such as segment duplication, are all special cases of our new metric.

IV. SID—A Software Integrity Diagnosis System

Formula (1) presents an ideal metric by which we can measure how much information two programs share. Unfortunately, it is well known that Kolmogorov complexity is not computable [9]. SID was built based on the hope that some crude approximation of $d(x,y)$ can still yield a

[1]The application to language classification was first investigated by authors of [2] using a similar but asymmetric measure.

TABLE I
COMPRESSION RESULTS (BYTES) OF FIVE TOKEN SEQUENCES. s^2 REPRESENTS ss. THE THIRD LINE OF EACH CELL IS CALCULATED FROM $\mathrm{Comp}(ss) - \mathrm{Comp}(s)$, APPROXIMATING $K(s|s)$ WHICH SHOULD BE ZERO IN THEORY. PPM ALGORITHM PERFORMS THE BEST FOR ORIGINAL TOKEN SEQUENCES, BUT IT IS UNIFORMLY WORSE THAN TOKENCOMPRESS TO COMPRESS DUPLICATED SEQUENCES. THE SEQUENCES USED HERE ARE AT http://software.bioinformatics.uwaterloo.ca/sid/testfiles.html

Filename	file size	Compress	Gzip-9	Bzip-2	PPM-9	TokenCompress
JavaParserMain	565	272	253	241	**189**	200
JavaParserMain^2	1130	458	264	278	236	**202**
		186	11	37	47	**2**
MultipartRequest	1676	769	606	586	**529**	552
MultipartRequest^2	3352	1322	631	698	678	**559**
		553	25	112	149	**7**
AnOperon	2109	805	643	620	**552**	594
AnOperon^2	4218	1414	674	741	735	**601**
		609	31	121	183	**7**
EcoliInc	2866	995	644	664	599	**593**
EcoliInc^2	5732	1729	678	774	786	**602**
		734	34	110	187	**9**
Parser	4850	1888	1241	1260	**1158**	1202
Parser^2	9700	3302	1310	1496	1540	**1218**
		1414	69	236	382	**16**

useful system. In SID, we resort to a compression algorithm to heuristically approximate Kolmogorov complexity. That is, we approximate $d(x,y)$ by

$$d(x,y) \approx 1 - \frac{\mathrm{Comp}(x) - \mathrm{Comp}(x|y)}{\mathrm{Comp}(xy)}$$

where $\mathrm{Comp}(\cdot)$ $(\mathrm{Comp}(\cdot|\cdot))$ represents the size of the (conditional) compressed string by a compression program. It is possible to extend the theory to allow SID to handle the common code given by the professor by defining a conditional shared information distance $d(x,y|\mathrm{ProfessorCode})$ using the conditional Kolmogorov complexity. This feature is implemented in SID.

Note that the Greedy-String-Tiling method used in YAP (YAP3) can now be considered as a very special case of $d(x,y)$. Meanwhile, three problems [22] that plague most of the other metrics may all be viewed as special cases of $d(x,y)$.

A. The TokenCompress Algorithm

SID depends on a reasonable approximation to $d(x,y)$. In principle, no computable compression algorithm can approximate $d(x,y)$ [9]. In order to approximate $d(x,y)$ better in practice, a TokenCompress algorithm is specially designed for SID, exploring unique requirements of our application. The requirements of TokenCompress differ from the traditional data compression requirements as follows.

- Traditional compression algorithms are often required to run in real time or linear time. The SID system needs the best compression possible, while the time requirement is looser. Our program may run in superlinear time, but it should achieve as good a compression ratio as possible for our particular type of data.
- Our formulation of $d(x,y)$ requires us to compute conditional compression of x given y, $\mathrm{Comp}(x|y)$ or $\mathrm{Comp}(xy)$. Plagiarized program pairs typically contain long and approximately duplicated blocks, as the cheaters copy and do simple edits to change the program [15]. A general-purpose universal compression algorithm is not sufficient to handle these, as shown in the second row in each cell in Table I. To handle exact repeats, research on Lempel–Ziv (LZ) type algorithms with unbounded buffers can be found, for example, in [24]. Approximate matching has been extensively studied for lossy compression, see for example [12]. SID needs a nonlossy compression program to effectively deal with approximate repeats.

The parser used in SID is downloaded from http://www.cobase.cs.ucla.edu/pub/javacc/. Each token is represented by a byte. TokenCompress follows the LZ data compression scheme [25]: it first finds the longest approximately duplicated substring; and then encodes it by a pointer to its previous occurrence plus edit corrections. The implementation differs from the standard LZ compression scheme by taking the unique features of our application and the above requirements into consideration.

First, TokenCompession considers all possible substrings located previously to search for an optimal match to compress. A typical LZ-type algorithm with bounded buffer size would miss long duplicated substrings because only a part of encoded substrings would be added into a lookup dictionary (or a sliding window of a few kilobytes long) built during the encoding process. It aims to use less encoding time and less memory space, at the cost of a little worse compression efficiency on common English texts. For token sequences with long and approximately identical program sections, such algorithms give inferior compression results, as clearly shown in the second row in each cell in Table I, for a host of well-known programs: Compress, Gzip-9, Bzip-2, PPM-9. We note that LZ-type algorithms with unbounded buffers and exact matches have been done in the past, see [24].

Typical LZ-type algorithms also do not handle approximate matches. By approximate match, in this correspondence, we mean two sequences can be converted to each other with relatively few operations of the edit distance: insertion, deletion, replacement. Our compression program searches for approximately matched substrings and encodes mismatches in term of insertion, deletion, and replacement. TokenCompress uses a threshold function to determine if an encoding of an approximate match is better than other alternatives (such as encoding parts separately or not encoding some parts at all). Some more implementation details of the threshold function and the encoding method can be found in [5]. Thus, several duplicated sections may be then combined to form a larger section with a few mismatches.

The TokenCompress algorithm is briefly described below. Conditional compression is implemented similarly. Similarly matched code segment pairs (i.e., repeat pairs) are recorded in a file and later presented in a result-viewing webpage. Experiments show that for the plagiarism detection purposes, as shown in the second row in each

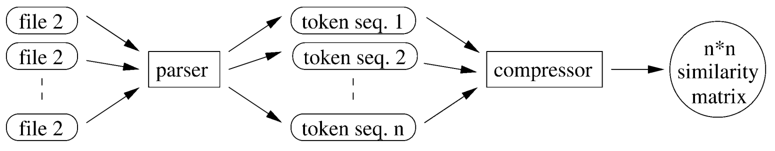


Fig. 1. Block diagram of SID's design.

cell in Table I, TokenCompress significantly outperforms the widely used compression programs Compress, Gzip-9, Bzip-2, PPM-9, for five random token sequences.

```
Algorithm TokenCompress
Input: A token sequence $s$
Output: $Comp(s)$ and matched repeat pairs
$i = 0$;
An empty buffer $B$;
while $(i < |s|)$
  $p = \mathrm{FindRepeatPair}(i)$;
  if $(p.compressProfit > 0)$
    $\mathrm{EncodeRepeatPair}(p, compFile)$;
    $i = i + p.length$;
    $\mathrm{OutputRepeatPair}(p, repFile)$;
  else
    $\mathrm{AppendCharToBuffer}(s_i, B)$;
    $i++$;
$\mathrm{EnodeLiteralZone}(B, compFile)$;
return $Comp(s) = compFile.file\_size$;
    and repeat pairs stored in $repFile$.
```

B. System Integration

SID works in two phases. In the first phase, source programs are parsed to generate token sequences by a standard lexical analyzer. In the second phase, TokenCompress algorithm is used to compute, heuristically, the shared information metric $d(x, y)$ between each program pair within the submitted corpus. Finally, all the program pairs are ranked by their similarity distances. Instructors can compare the program source codes side by side using the SID graphical user interface. Similar parts are highlighted. A block diagram of the design of SID is shown in Fig. 1.

The SID system has been implemented in Java. Instructors can submit all the programs they wish to compare by creating a zip file that contains each program in its own directory, then submitting it to the SID server through a Web browser. Although the current version of SID only accepts Java and C++ programs as inputs, it can compare programs in any other language if a parser for that language is incorporated into the SID system.

V. Experiments: Sensitivity and Specificity Analysis

Through a series of experiments, large scale and specially designed, we test the sensitivity and specificity of SID system.

A. Large-Scale Experiments

SID has been extensively tested and used. Users have used SID positively to catch plagiarism cases. This section contains several of our own routine experiments.

A UCSB Experiment. A collection of 58 undergraduate homework assignments at University of California, Santa Barbara (UCSB) were submitted to SID. Top three of the most similar program pairs are shown in Table II with their similarity scores $R(x, y) = 1 - d(x, y)$ and p, which is the percentage of program codes are exactly copied. Further examination of the source files confirmed such findings. The first pair was determined to be identical copies to each other, only with several trivial modifications. The second highest scored program pair did not look alike. Initially, the instructor came to one of the authors (ML) and said: "Look, these two programs do not look alike. One is 6 pages and the other is 9." We asked him to find out. The instructor wrote to the two students asking for truth while giving them amnesty of any wrong doings. The two students, turned out to be a couple, wrote back admitting that they have always "worked together" but they did not really "copy."

TABLE II
Similarity Measure Scores of a Program Pair

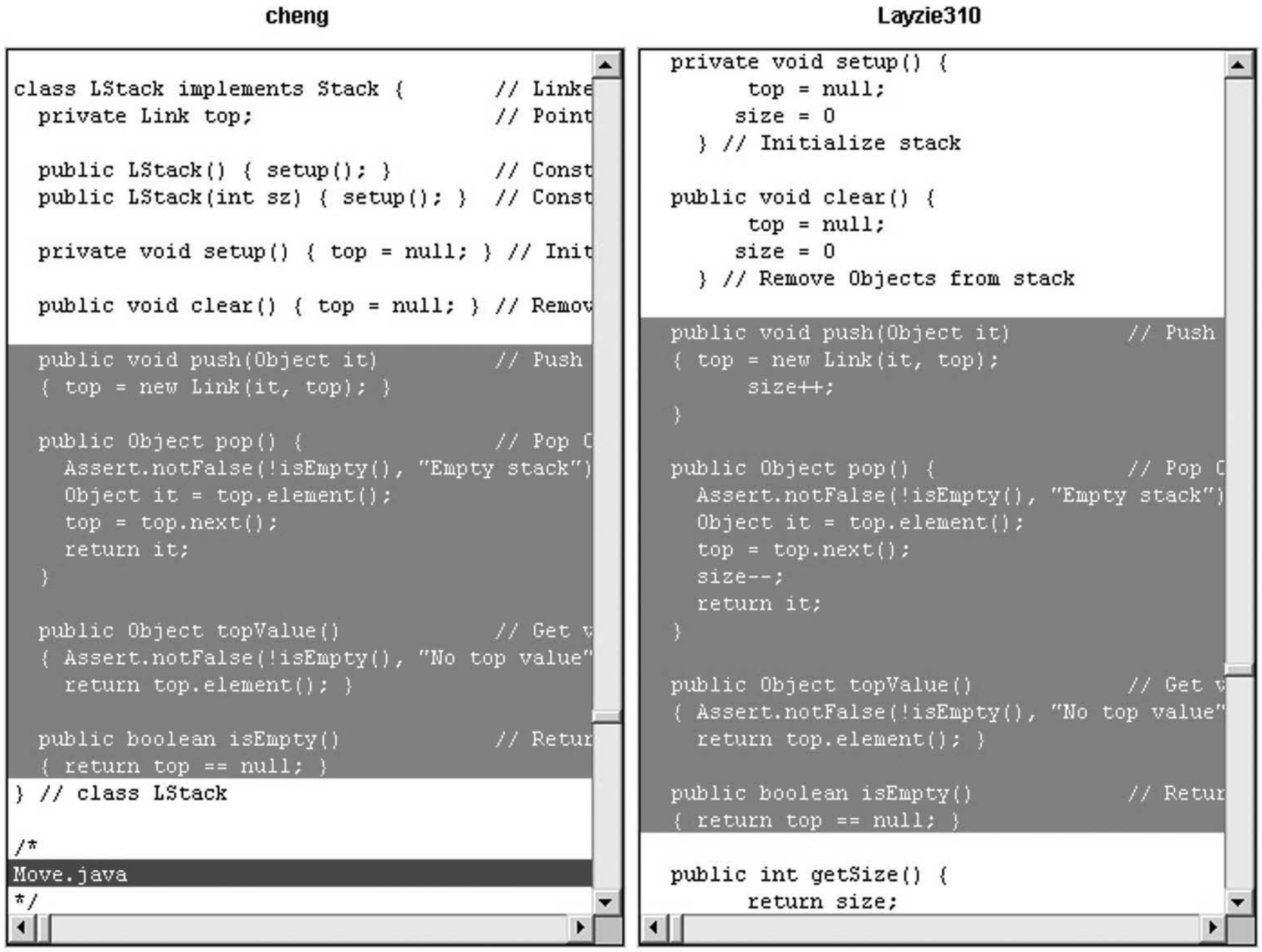
student 1	student 2	$R(s,t)$	$R(t,s)$	p
u1	u2	0.8830	0.8737	93.79%
u3	u4	0.1826	0.1687	30.88%
u5	u6	0.1142	0.1336	20.50%

UW CS 341 Assignment 3. A collection of 171 undergraduate algorithm programming assignments were submitted to SID for comparison with the known knowledge that three pairs of students had been confirmed as cheaters. The cheaters were caught by a TA who previously used MOSS to identify any suspicious pairs of programs. The top three results returned by SID were in fact the same three pairs of confirmed cheaters.

UW CS 341 Assignment 4. Another collection of 171 undergraduate algorithm programming assignments were submitted to both SID and MOSS by a TA for comparison. The TA then looked at the top matches from both MOSS and SID. The top flagged pair by MOSS and SID were the same, after a careful investigation by the TA it was confirmed that the pair of programs in question had in fact been plagiarized. There was only one confirmed case of cheating, the scores for both MOSS and SID indicated that only one pair of programs shared a lot of information.

UW CS 133 (Java Programming) Assignment 4. This submission consisted of 340 assignments. The TAs for the course used MOSS to help finding any suspicious program pairs, and as a result of a lengthy investigation, there were six pairs of confirmed cases of cheating. SID was run on the same assignments. SID flagged five of the pairs confirmed as cheating at the top of its list, the sixth was ranked lower but comparable to MOSS' ranking. A visual inspection of the pairs that SID had flagged higher than the last cheating pair suggested that they likely have been plagiarized, too. No further investigation was done as it was too late in the term to follow up on.

UW CS 133 Assignments 5, 6, and 7. These three submissions consisted of around 340 programming projects each. The TAs for the course did not run any type of plagiarism detection on the programs, so there were no confirmed cases of cheating. The projects were submitted to both SID and JPlag. For each assignment, the top matches of JPlag and SID were then investigated visually. The top pairs from SID and JPlag were the same with minor scoring variations—they were apparent plagiarism cases. It was too late in the term to follow up with the students.



```
class LStack implements Stack {        // Linke
  private Link top;                    // Point

  public LStack() { setup(); }         // Const
  public LStack(int sz) { setup(); }   // Const

  private void setup() { top = null; } // Init

  public void clear() { top = null; } // Remov

  public void push(Object it)          // Push
  { top = new Link(it, top); }

  public Object pop() {                // Pop C
    Assert.notFalse(!isEmpty(), "Empty stack")
    Object it = top.element();
    top = top.next();
    return it;
  }

  public Object topValue()             // Get v
  { Assert.notFalse(!isEmpty(), "No top value"
    return top.element(); }

  public boolean isEmpty()             // Retur
  { return top == null; }
} // class LStack

/*
Move.java
*/
```

```
private void setup() {
      top = null;
     size = 0
  } // Initialize stack

private void clear() {
      top = null;
     size = 0
  } // Remove Objects from stack

public void push(Object it)          // Push
{ top = new Link(it, top);
      size++;
}

public Object pop() {                // Pop C
  Assert.notFalse(!isEmpty(), "Empty stack")
  Object it = top.element();
  top = top.next();
  size--;
  return it;
}

public Object topValue()             // Get v
{ Assert.notFalse(!isEmpty(), "No top value"
  return top.element(); }

public boolean isEmpty()             // Retur
{ return top == null; }

public int getSize() {
      return size;
```

Fig. 2. A similar code segment detected by SID.

B. Case Studies—To Be and Not to Be Caught

Avoiding being caught is the goal of the plagiarizers. In this section, we explore ways to beat the plagiarism detection systems and test which system is more resilient to these attacks.

Simple attacks. Simple adversary tests, using well-known tricks, were systematically performed against JPlag, MOSS, and SID. In all of our tests, JPlag, MOSS, and SID were all resilient to variable and function renaming, refactoring the program by introducing new methods, reordering of functions and methods, reordering and swapping variable names, and insertion of a few irrelevant blocks. Details of these test results are provided at http://software.bioinformatics.uwaterloo.ca/SID/tests.html.

A standard test case. Gitchell and Tran [7] gave a simple and well-designed sample that can be used to test the reliability of similarity metrics. In this sample, the copier changed all variable and function names, removed comments, inverted the order of adjacent statements whenever possible, and permuted the order of the functions from one program to another. We submitted this program pair to SID. The result is presented in Table III, where each entry is

$$R(x,y) = 1 - d(x,y) = \frac{K(x) - K(x|y)}{K(xy)}.$$

As expected from theory, $R(x,y)$ is approximately symmetric, $R(x,y) \approx R(y,x) \approx 0.4$.

TABLE III
SIMILARITY SCORES OF A PROGRAM PAIR

Similarity score	Original_copy	Modified_copy
Original_copy	0	0.4031
Modified_copy	0.3969	0

In order to further interpret such a similarity score intuitively, consider a program pair x and y of the same size. Assume that p percentage of program codes are exactly copied between x and y and other parts are not compressible. Then

$$R(x,y) = p/(2-p).$$

Thus, a value $R(x,y) = 0.4$ in the above example implies $p = 57.14\%$ of program codes are identical when x and y are of same size. This warrants this program pair to be further examined by the instructor. Notice that due to compression overhead, the $R(x,y)$ value such as 0.4 should be considered rather high. When such scoring method was applied to biological sequences, genomes from sister species often have values around 0.4 [8].

Random insertions—How to cheat JPlag and MOSS. Plagiarizers do not understand the original programs (otherwise our instructors would have already achieved their educational goals), hence they cannot delete things from it. Thus, a common trick is to insert irrelevant statements such as $\text{int } x = 0;$ into the code in hopes that such insertions will confuse the detection mechanism. Such

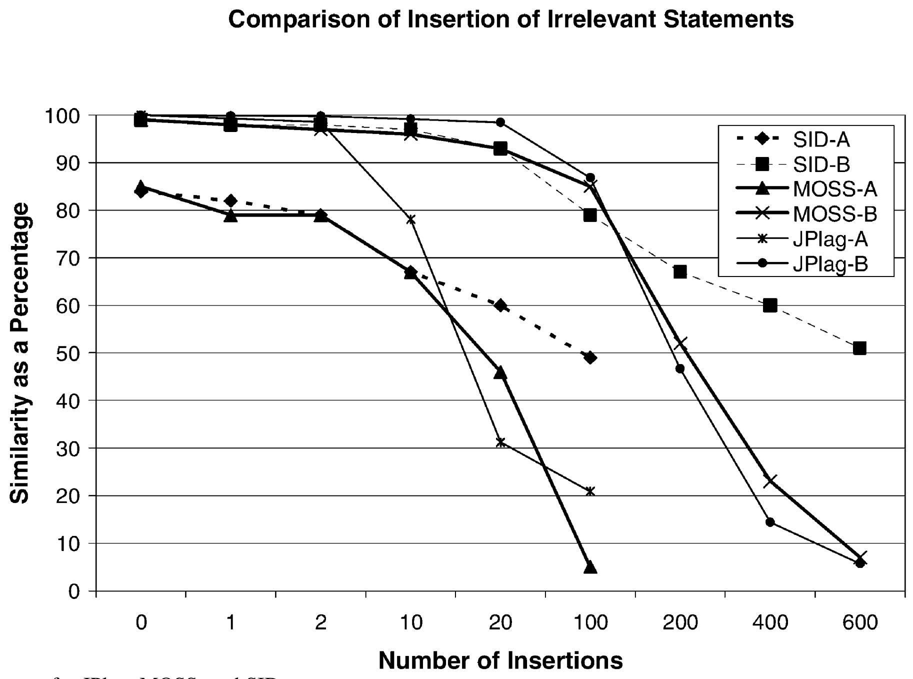


Fig. 3. Random insertion tests for JPlag, MOSS, and SID.

random insertions can be camouflaged under, for example, debugging statements like System.out.println("debug").

Our metric and its approximation TokenCompress algorithm naturally allow SID to guard against simple editing modifications, such as random insertions of irrelevant statements. A practical example is given in Fig. 2, where the highlighted similar code segments were detected by SID, but by neither JPlag [13] nor MOSS [1] in year 2002.

Since 2003, JPlag and MOSS have been improved [17]. However, the simple insertion tactic still defeats them. Fig. 3 shows the effects of inserting more and more irrelevant statements, on JPlag, MOSS ,and SID. respectively. The tests were conducted on both a small program (A) of roughly 100 lines of code and a large program (B) of roughly 1100 lines of code. Each program was inserted with a certain number of System.out.println("debug"); statements, as such statements often occur in code for debugging purposes in practice. It can be clearly seen in Fig. 3 that as the number of irrelevant statements increases JPlag and MOSS deteriorate to the point in which they think the programs are dissimilar, while SID, on the other hand, demonstrates a resilience to such cheating attempts as it never goes below a 50% similarity.

False positives for improper metrics. We now provide an admittedly artificial test. The programs in this test are of the form: P, $Q, Q_1, \ldots, Q_k$, where P is slightly larger in size than Q, Q is a trivial program from a textbook (a standard sorting program) such that all students would implement it similarly, Q_i's are simple variations of Q, say sorting reals, shorts, longs, and so on. Students A and A$'$ copied P, but wrote different Q. Students B and B$'$ implemented P independently, but used standard textbook Q. JPlag and MOSS give very high scores between the false positive pair B and B$'$, while missing the real culprits A and A$'$ when many B-pairs are present. However, SID reports on A and A$'$ copying case since it compresses $Q, Q_1, \ldots, Q_k$ to size $|Q|$ and hence discovers the nontrivially copied P part.

VI. Discussion

We have implemented SID and a Web service to detect program plagiarism online. The system is based on our new distance metric that measures the normalized amount of shared information between two programs. The measure is universal, hence it is theoretically not cheatable. In practice, the metric is not computable, SID system uses a special compression program to heuristically approximate Kolmogorov complexity. As implemented, SID not only holds the three properties required in [22], it also detects subtler similarities. Small- and large-scale experiments show that SID has higher sensitivity and specificity than the popular plagiarism detection systems we have tested, due to the proper metric we have used. Several tests vividly demonstrate how a wrong metric can give false positives as well as false negatives. In fact, we have even provided a simple method, in Fig. 3, of how to plagiarize against other programs except SID. In all of our tests, SID has not given a single false positive. Anything SID finds similar was later determined by a human to be similar beyond coincidence.

It is important to remember that the work presented in this correspondence is a part of our larger goal to establish and justify a general theory of shared information between two sequences. Successful applications already include genome–genome comparison [8], English document comparison [4], language classification [10], [2], as well as music classification [6].

Acknowledgment

The authors would like to thank Byron Weber Becker, Brona Brejova, Carrie Howells, Eliot Jacobson, and Kevin Lanctot for their support, and the referees and the special issue Guest Editors for their criticism and suggestions.

References

[1] A. Aiken. Measure of Software Similarity. [Online]. Available: http://www.cs.berkeley.edu/~aiken/moss.html
[2] D. Benedetto, E. Caglioti, and V. Loreto, "Language trees and zipping," *Phys. Rev. Lett.*, vol. 88, no. 4, 2002.
[3] C. H. Bennett, P. Gács, M. Li, P. Vitányi, and W. Zurek, "Information distance," *IEEE Trans. Inform. Theory*, vol. 44, pp. 1407–1423, July 1998.
[4] C. Bennett, M. Li, and B. Ma, "Chain letters and evolutionary histories," *Scientific Amer.*, pp. 71–76, June 2003.
[5] X. Chen, S. Kwong, and M. Li, "A compression algorithm for DNA sequences and its applications in genome comparison," in *Proc. 10th Workshop on Genome Informatics*, Tokyo, Japan, Dec. 1999, pp. 52–61.
[6] R. Cilibrasi, R. de Wolf, and P. Vitányi. (2003) Algorithmic Clustering of Music. [Online]. Available: http://arxiv.org/archive/cs/0303025

[7] D. Gitchell and N. Tran, "A utility for detecting similarity in computer programs," in *Proc. 30th ACM Special Interest Group on Computer Science Education Tech. Symp.*, New Orleans, LA, 1998, pp. 266–270.
[8] M. Li, J. Badger, X. Chen, S. Kwong, P. Kearney, and H. Zhang, "An information-based sequence distance and its application to whole mitochondrial genome phylogeny," *Bioinformatics*, vol. 17, no. 2, pp. 149–154, 2001.
[9] M. Li and P. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications*, 2nd ed. New York: Springer-Verlag, 1997.
[10] M. Li, X. Chen, X. Li, B. Ma, and P. Vitanyi, "The similarity metric," in *Proc. 14th Annu. ACM-SIAM Symp. Discrete Algorithms*, Baltimore, MD, 2003, pp. 863–872.
[11] M. Li and P. Vitanyi, "Reversibility and adiabatic computation: Trading time and space for energy," *Proc. Roy. Soc. London*, ser. A, vol. 452, pp. 769–789, 1996.
[12] T. Łuczak and W. Szpankowski, "A suboptimal lossy data compression based on approximate pattern matching," *IEEE Trans. Inform. Theory*, vol. 43, pp. 1439–1451, Sept. 1997.
[13] G. Malpohl. JPlag: Detecting Software Plagiarism. [Online]. Available: http://www.ipd.uka.de:2222/index.html
[14] K. Ottenstein, "An algorithmic approach to the detection and prevention of plagiarism," *SIGCSE Bull.*, vol. 8, no. 4, pp. 30–41, 1977.
[15] A. Parker and J. Hamblen, "Computer algorithms for plagiarism detection," *IEEE Trans. Education*, vol. 32, pp. 94–99, May 1989.
[16] S. C. Sahinalp, M. Tasan, J. Macker, and Z. M. Ozsoyoglu, "Distance based indexing for string proximity search," in *Proc. Int. Conf. Data EngineeringICDE'2003*, Bangalore, India, Mar. 2003, pp. 125–138.
[17] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," in *Proc. ACM SIGMOD Conf.*, San Diego, CA, June 9–12, 2003, pp. 76–85.
[18] C. E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, vol. 27, pp. 379–423, July and Oct. 1948.
[19] W. Weaver and C. E. Shannon, *The Mathematical Theory of Communication*. Chicago, IL: Univ. Illinois Press, 1949.
[20] G. Whale, "Plague: Plagiarism Detection Using Program Structure," Dept. Comput. Sci., Univ. New South Wales, Kensington, Australia, Tech. Rep. 8805, 1988.
[21] ——, "Identification of program similarity in large populations," *Computer J.*, vol. 33, no. 2, pp. 140–146, 1990.
[22] M. Wise, "Running Karp–Rabin Matching and Greedy String Tiling," Dept. Comput. Sci., Sydney Univ., Sydney, Australia, Tech. Rep., 1994.
[23] ——, "YAP3: Improved detection of similarities in computer program and other texts," in *Proc. 27th SCGCSE Tech. Symp.*, Philadelphia, PA, 1996, pp. 130–134.
[24] E.-h. Yang and J. C. Kieffer, "On the performance of data compression algorithms based upon string matching," *IEEE Trans. Inform. Theory*, vol. 44, pp. 47–65, Jan. 1998.
[25] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inform. Theory*, vol. IT-23, pp. 337–343, May 1977.

Universal Entropy Estimation Via Block Sorting

Haixiao Cai, *Student Member, IEEE*,
Sanjeev R. Kulkarni, *Fellow, IEEE*, and Sergio Verdú, *Fellow, IEEE*

***Abstract*—In this correspondence, we present a new universal entropy estimator for stationary ergodic sources, prove almost sure convergence, and establish an upper bound on the convergence rate for finite-alphabet finite memory sources. The algorithm is motivated by data compression using the Burrows–Wheeler block sorting transform (BWT). By exploiting the property that the BWT output sequence is close to a piecewise stationary memoryless source, we can segment the output sequence and estimate probabilities in each segment. Experimental results show that our algorithm outperforms Lempel–Ziv (LZ) string-matching-based algorithms.**

***Index Terms*—Block sorting, Burrows-Wheeler transform (BWT), entropy estimation, piecewise stationary memoryless source, tree source.**

I. Introduction

Ever since Shannon's initial work on the entropy of English text [26], there has been significant interest in the problem of how to estimate the entropy of sources whose statistical characterization is unknown.

In the area of entropy estimation for independent and identically distributed (i.i.d.) sources, the extension to countably infinite alphabet has been considered in [1], [36]. It is shown in [36] that, for the class of discrete memoryless sources with a countable alphabet and finite entropy variance, i.e., $E[(-\log_2 P(Z))^2] < \infty$, there can be no universal estimator that converges at a rate faster than $O(1/(\log_2 n)^{1+\epsilon})$ for all sources and any $\epsilon > 0$. Furthermore, both the Lempel–Ziv (LZ) string matching estimator and the plug-in estimator achieve the convergence rate $O(1/\log_2 n)$. In [21], it is pointed out that in the undersampled regime, i.e., the alphabet size is comparable to n, the plug-in estimator can be contaminated by the bias, although the variance converges to zero fast. Hence, an entropy estimator aimed at simultaneously minimizing bias and variance is proposed and its application in neural science shown in [21]. In the setting of high-dimensional i.i.d. sources, the minimal spanning tree approach is proposed in [11] as an alternative entropy estimator.

Entropy estimators for sources with memory are often related to universal data compression algorithms. The entropy estimator introduced in [35] was motivated by the LZ data compression algorithm. The basic idea is to find the longest string pattern that has occurred previously in the sequence of length n. The length of this longest string is denoted by L_n. The corresponding estimator is $H_n = \log_2 n/L_n$. This universal estimator converges to the entropy in probability for all stationary ergodic processes [35].[1] Faster convergence can be achieved by an averaging procedure [15], [22], [27], which basically averages longest-

Manuscript received May 5, 2003; revised March 31, 2004. This work was supported in part by ARL MURI under Grant DAAD19-00-1-0466, Draper Laboratory under IR&D 6002 Grant DL-H-546263, and the National Science Foundation under Grant CCR-0312413. The material in this correspondence was presented in part at the IEEE International Symposium on Information Theory, Lausanne, Switzerland, June/July 2002.

The authors are with the Electrical Engineering Department, Princeton University, Princeton, NJ 08544 USA.

Communicated by E.-h. Yang, Guest Editor.
Digital Object Identifier 10.1109/TIT.2004.830771

[1]Almost-sure convergence may fail depending on the details of how $\boldsymbol{L}_n$ is defined (see [30]), although in the setting of double-sided sequences and static model, almost-sure convergence was proved in [20].