COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY

COCHIN – 682022

2010

Seminar Report

On

PLAGIARISM DETECTION TECHNIQUES

Submitted By

Sangeetha Jamal

In partial fulfillment of the requirement for the award of

Degree of Master of Technology (M.Tech)

In

Software Engineering

ABSTRACT

Plagiarism refers to “the act of copying materials without actually acknowledging the original source”. Plagiarism has seen a widespread activity in the recent times. The increase in the number of materials available now in the electronic form and the easy access to the internet has increased plagiarism. Manual detection of plagiarism is not very easy and is time consuming due to the vast amount of contents available. Techniques are available now which help us to detect plagiarism. As the amount of programming code being created is increasing, techniques are available now to detect plagiarism in source code also. Current research is in the field of development of algorithms that can compare and detect plagiarism. In this paper a few techniques used in the Plagiarism Detection is shown along with some tools which are being used.

Keywords : Plagiarism, Plagiarism Detection, Plagiarism Prevention, Source Code

Plagiarism, Large Repositories

CONTENTS

1. INTRODUCTION

Plagiarism has become a world-wide problem and is increasing day by day. This problem is getting worse mainly because of the increase in the volume of on-line publications. Relying only on exact-word or phrase matching for plagiarism detection is not sufficient now. People have started paraphrasing or rearranging words to give a new look to their sentences and thus declare themselves as authors of the material.

Using Plagiarism Detection Techniques we can compare a given material with any target material which is either a particular document or in a repository. Different techniques used in the Plagiarism Detection algorithms are discussed in detail here. Here I have given more emphasis on source code related plagiarism. A few case studies show that detection can be done within a large repository. The efficiency and time of the output depends on the algorithms used.

2. DEFINITION OF PLAGIARISM

Plagiarize according to the Merriam-Webster Online Dictionary is

- to steal and pass off (the ideas or words of another) as one's own
- to use (another's production) without crediting the source
- to commit literary theft
- to present as new and original an idea or product derived from an existing source.

The expression of original ideas is considered intellectual property and is protected by copyright laws, just like original inventions. Almost all forms of expression fall under copyright protection as long as they are recorded in some way (such as a book or a computer file). In other words, plagiarism is an act of fraud. It involves both stealing someone else's work and lying about it afterward.

The following are considered as plagiarism

- turning in someone else's work as your own
- copying words or ideas from someone else without giving credit
- failing to put a quotation in quotation marks
- giving incorrect information about the source of a quotation
- changing words but copying the sentence structure of a source without giving credit
- copying so many words or ideas from a source that it makes up the majority of your work, whether you give credit or not

Plagiarism can be deliberate or accidental. Figure 1 shows the range between Deliberate and Accidental Plagiarism. Deliberate plagiarism is done when a person’s self esteem is very low. The person, therefore, actually steals the property of somebody else and claims it to be his own. He might also hire somebody to do his work. Accidental plagiarism is done when somebody unknowingly cites a phrase or copies words without acknowledging the author of the material.

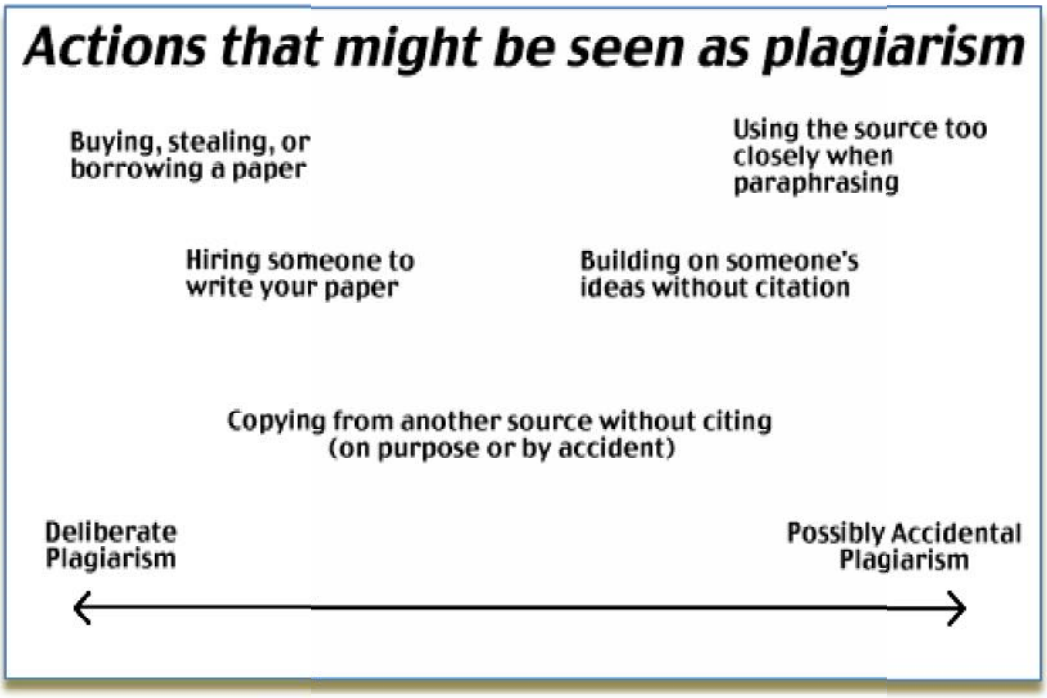

Figure 1. Deliberate Plagiarism vs Accidental Plagiarism

In the present age of computers and internet the attack of plagiarism has increased. Digitalized documents have only provided a means to increase plagiarism. This has led to an adverse effect on the learning criteria of youngsters today. They are depriving themselves of better learning opportunities and have become addicted to plagiarism in such a way that the true identity of the material is lost. This has also led to professional dishonesty in the job front.

The rapid increase in the digital documents and the easy access to the material has led to increase in plagiarism. This has in turn decreased the chances of detecting plagiarism. Now it is more and more difficult to find plagiarized content. So currently we have started many methods and techniques to detect and to avoid plagiarism.

3. AVOIDING PLAGIARISM

Plagiarism in all kinds of work has made people to sit and think regarding the ways to avoid plagiarism. Mainly two methods exist to avoid plagiarism.

- Plagiarism prevention
- Plagiarism Detection

4. PLAGIARISM PREVENTION

- A collaborative effort should be made to recognize and to counter plagiarism at every level.
- We should educate students about the appropriate use and acknowledgement of all forms of intellectual material.
- Minimize the possibility of submission of plagiarized content while not reducing the quality and rigor of assessment.
- Installing highly visible procedures for monitoring and detecting cheating.

Plagiarism prevention is difficult to achieve and takes a long time to inculcate but the effects are long term.

5. PLAGIARISM DETECTION

Plagiarism can be detected manually or with the help of software manual detection takes more effort. Now the detection Techniques is software programming methods which are easier, simpler and faster to detect plagiarism.

Culwin and Lancaster define a four stage process for detecting plagiarism and is shown in figure 2.

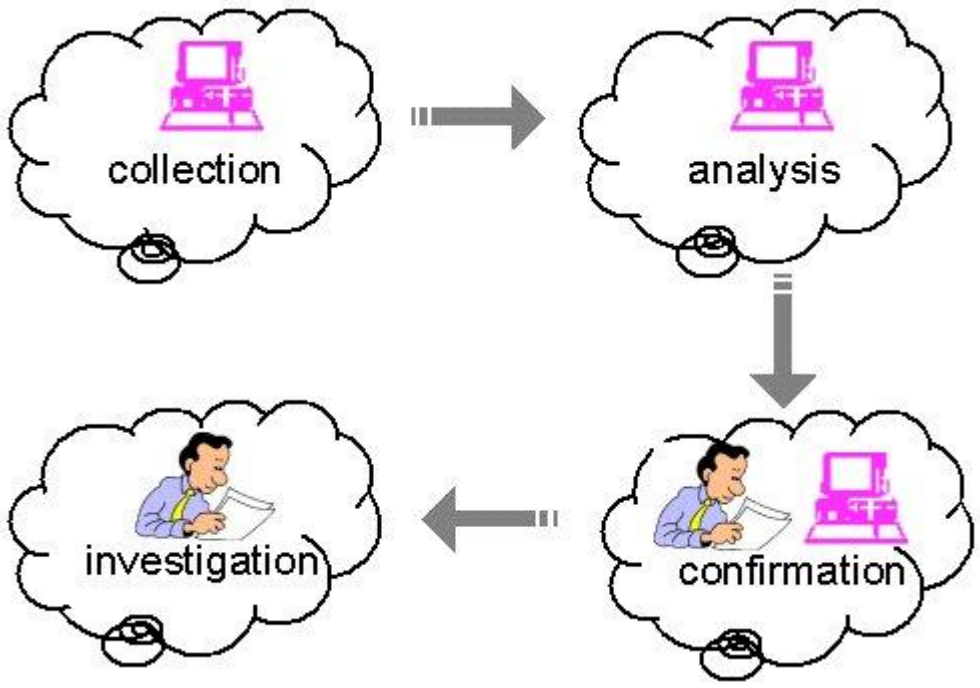

Figure 2: Culwin and Lancaster's four stages of detecting plagiarism

The **collection** stage may be defined as the process of electronically collecting and pre-processing student submissions into a suitable format.

Analysis is defined as "where the submissions are compared with each other and with documents obtained from the Web and the list of those submissions, or pairs, that require further investigation is produced."

Verification (confirmation) is required to ensure that those pairs reported as being suspicious are worth investigating with a view to possible disciplinary action (this is a task normally undertaken by humans, since value judgements may be involved).

The final stage**, investigation**, will determine the extent of the alleged misconduct and will "also involve the process of deciding culpability and possible penalties."

5.1. Plagiarism detection system

Most existing detectors are specially designed to process natural language text or program source code. Systems designed for finding similarities in natural language texts mainly searched the Internet for the possible matches. Text comparisons use simple comparison methods aiming mostly at processing speed and wide coverage. The program source code usually performs a pair wise comparison between single submissions only. Though sophisticated procedures are being developed which compares with multiple source code programs simultaneously.

5.1.1 Text Based Detection systems

- **Substring Matching**

Substring matching approaches try to identify maximal matches in pairs which then are used as plagiarism indicators. Typically, the substrings are represented in suffix trees, and graph-based measures are employed to capture the fraction of the plagiarized sections.

- **Keyword Similarity**

Topic identifying keywords are extracted and compared to keywords of other documents. If the similarity exceeds a threshold, the candidate documents are divided into smaller pieces, which are then compared recursively.

This approach assumes that plagiarism usually happens in topically similar documents.

- **Exact Fingerprint Match**

The documents are partitioned into term sequences, called chunks, from which digital digests are computed that form the document's fingerprint. When the digests are inserted into a hash table, collisions indicate matching sequences. For the fingerprint computation a standard hashing approach such as MD5 hashing is employed, which suffers from two severe problems: (1) it is computationally expensive, (2) a small chunk size (3-10 words) must be chosen to identify matching passages, which additionally increases the effort for fingerprint computation, fingerprint comparison, and fingerprint storage.

- **Text parsing**

Any sentence of the text can be automatically represented in the form of the tree, which reflects the structure of the sentence. For example, the phrase *the monkey ate the banana* will be parsed by such software as shown in Figure 3.

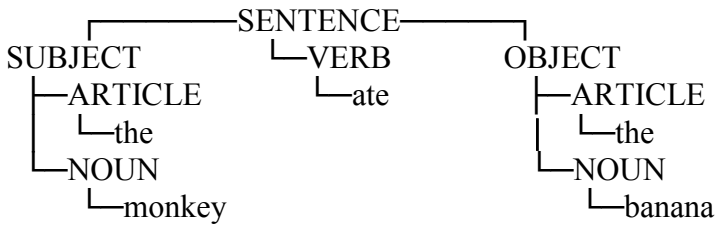

Figure 3. Parsing the Sentence

Once a parse tree is created, we can invoke a tree matching procedure. Initially the algorithm builds a flowchart-styled parse tree for each file to be analyzed. Then for each pair of files, the algorithm performs a rough "abstract comparison", when only types of the parse tree elements (like ASSIGNMENT, LOOP, BRANCHING) are taken into account. This is done recursively for the each level of tree nodes. If the similarity percentage becomes lower than some threshold at some step, the trees are immediately treated as not similar. If the abstract comparison indicates enough similarity, a special low-level "micro comparison" procedure is invoked. At this point

each node represents an individual statement. Thus, each tree node turns into a separate subtree that has to be compared with the corresponding subtree taken from another file.

The tree matching can help to reveal rewording. If we treat the children of every tree node as an unordered collection of nodes, e.g. the phrases *the monkey ate the banana* and *the banana was eaten by the monkey* will be very close after the tokenization.

5.1.1.1. Tools used for text based plagiarism

Detection tools are present that operate on free text and also that finds similarity in spreadsheets, diagrams, scientific experiments, music or any other non-textual corpora. Tools can be divided in three based on the type of corpus: tools that operate only intra-corpally (where the source and copy documents are both within a corpus), tools that operate only extra-corpally (where the copy is inside the corpus and the source outside) and tools that operate both – intra- and extra-corpally.

Most contemporary detection systems adopt a lexical-structural approach to identify these transformations: source programs are tokenized, and profiles are created and compared. While some academic institutions have developed their own in-house detections systems, such as Big Brother, there are also services available through a Web interface. The main players in this field are SIM, YAP, MOSS (Measure of Software Similarity) and JPLAG.

- **SIM (Software Similarity Tester)**

The SIM system tokenizes source programs and compares strings using pattern-matching algorithms based on work from the human genome project.

- **YAP (Yet Another Plague)**

The YAP approach also tokenizes source programs and retains only those tokens which are concerned with the structure of the program. This is based on a lexicon which is created specifically for each programming language. The output is a numeric profile which computes the closeness between two programs. This closeness between programs is partly a function of the programming language chosen and the type of

task undertaken (for instance, the COBOL programming language is by nature a highly structured and verbose language, and can lead to very similar programs, likewise with Visual Basic).

- **MOSS (Measure of Software Similarity)**

MOSS can be applied to a range of programming languages. Registered instructors can submit batches of programs to the MOSS server, and results are returned to a website. Little information is available on how the tool works (presumably because if this were known, it would be possible to evade detection), but it is based on the syntax or structure of a program, rather than the algorithms which drive the program The MOSS database stores an internal representation of programs, and then looks for similarities between them.

- **JPLAG**

JPLAG compares submitted programs in pairs, and is based on the assumption that plagiarists may vary the names of variables or classes, but they are least likely to change the control structure of a program. It has a very powerful graphical interface.

The performance of both MOSS and JPLAG were evaluated in the JISC (Joint Information Services Committee) report. The survey concluded that JPLAG was easier to use but supported fewer languages than MOSS and could not deal with programs which do not parse. As students often submit files which do not parse, such a limitation would mean that many files would not be under consideration. Neither JPLAG nor MOSS is easy to use: JPLAG requires that the user have Netscape 4 or use an applet, while MOSS requires configuration of perl files on a UNIX account.

- **Turnitin**

Corpus-based plagiarism detection software takes as input a suspect document and an archived corpus of authenticated documents, compares the suspect document to the corpus, and outputs passages that the suspect document shares with the corpus and a measure of the likelihood that the author plagiarized material. At a more operational level, a corpus-based protocol is implemented in several ways. TurnItIn.com, the most high profile company in the field, employs document source

analysis to generate digital fingerprints of documents, those submitted for authentication, those in the archive, and those available through ProQuest, and it complements the search of its in-house archive with searches of the World Wide Web.

- **Glatt**

Glatt Plagiarism Services exemplifies an interrogative approach to plagiarism detection. Work suspected of being plagiarized is given to the Glatt Plagiarism Screening Program, which is free standing, non-Web-based software. The program replaces every fifth word of the suspected paper with a standard size blank, and the student is then prompted to supply the missing words. The number of correct responses, the amount of time intervening between responses, and various other factors are considered in calculating a plagiarism probability index.

5.1.2. Source Code based plagiarism

The ease with which one student can copy computer codes from another increase the likelihood that some plagiarism will take place on a given work. This, combined with the fact that many programs are graded on what results they produce sometimes with a cursory glance at the source code, if any provides a relatively low risk environment for a student to conduct plagiarism. Finally, even if instructors wish to check to make sure no two assignments are very similar, the numbers are once again in the students favor. For a typical entry level class, there may be over 100 submissions per assignment. Checking for plagiarism can easily become an extremely time consuming task. Software which could identify pairs of assignments which contain a high degree of similarity would be extremely beneficial to instructors by allowing them to focus their time spent looking for plagiarism to likely plagiarized assignments. Furthermore, such software would provide peace of mind that each student submitted unique work.

The accepted model of classifying plagiarism of computer code by Faidhi and Robinson is shown in figure 4.

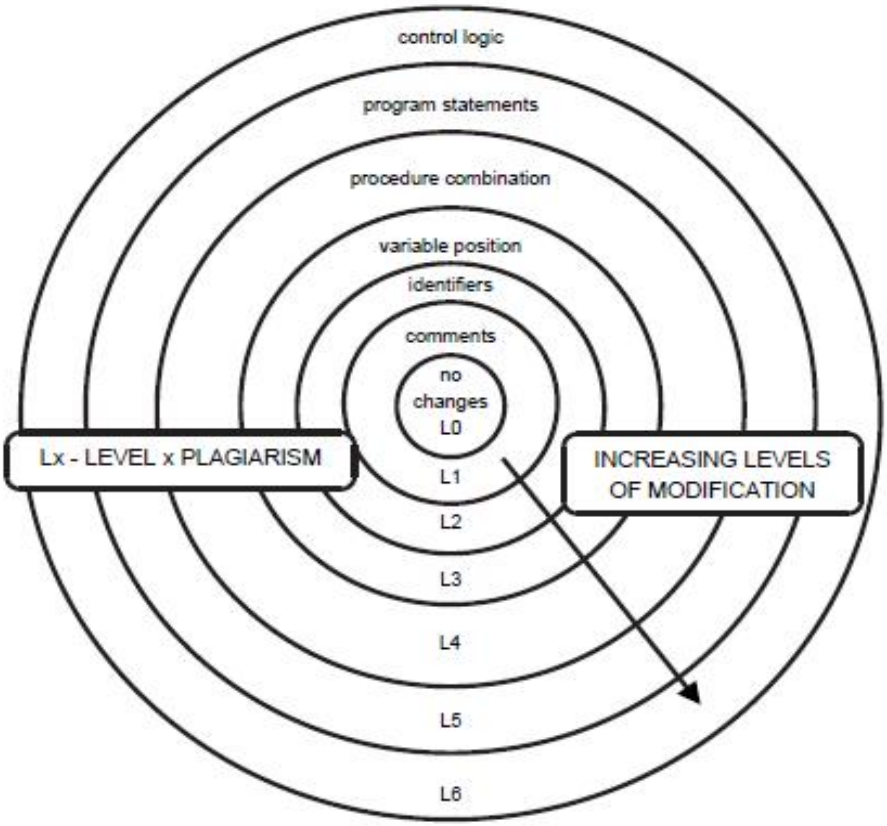

Figure 4: Program Modification Hierarchy

This hierarchy system describes the plagiarism content in student code. As we increase in the plagiarism level, the amount of understanding required to make necessary changes/edits also increases. The first three levels from L0- L2 represent the majority of edits that takes place when plagiarizing code. Based on these observations the focus on detecting plagiarism lies in categorizing the first three levels.

5.1.2.1 Source Code Detection Algorithms

According to Roy and Cordy, source-code similarity detection algorithms can be classified as based on either

- **Strings** - look for exact textual matches of segments, for instance five-word runs. Fast, but can be confused by renaming identifiers.

- **Tokens** - as with strings, but using a lexer to convert the program into tokens first. This discards whitespace, comments, and identifier names, making the system more robust to simple text replacements. Most academic plagiarism detection systems work at this level, using different algorithms to measure the similarity between token sequences.
- **Parse Trees** - build and compare parse trees. This allows higher-level similarities to be detected. For instance, tree comparison can normalize conditional statements, and detect equivalent constructs as similar to each other.
- **Program Dependency Graphs (PDGs)** - a PDG captures the actual flow of control in a program, and allows much higher-level equivalences to be located, at a greater expense in complexity and calculation time.
- **Metrics** - metrics capture 'scores' of code segments according to certain criteria; for instance, "the number of loops and conditionals", or "the number of different variables used". Metrics are simple to calculate and can be compared quickly, but can also lead to false positives: two fragments with the same scores on a set of metrics may do entirely different things.
- **Hybrid approaches** - for instance, parse trees + suffix trees can combine the detection capability of parse trees with the speed afforded by suffix trees, a type of string-matching data structure.

The previous classification was developed for code refactoring, and not for academic plagiarism detection (an important goal of refactoring is to avoid duplicate code, referred to as code clones in the literature). The above approaches are effective against different levels of similarity; low-level similarity refers to identical text, while high-level similarity can be due to similar specifications. In an academic setting, when all students are expected to code to the same specifications, functionally equivalent code (with high-level similarity) is entirely expected, and only low-level similarity is considered as proof of cheating.

A few major detecting techniques are discussed here.

- **Detection via Lexical Similarities**

The process of lexical analysis takes the human readable source code for a program and converts it into a stream of lexical tokens which a parser or compiler may use to extract meaning from the source. During the lexical analysis phase, the source code undergoes a series of transformation. Some of these transformations, such as the identification of reserved words, identifiers, and numbers are beneficial for plagiarism detection.

Consider the following two snippets of Java Code:

```
int[] A = {1,2,3,4};
for(int i = 0; i < A.length; i++) {
      A[i] = A[i] + 1;
}
```

```
int[] B = {1, 2, 3, 4};
for(int j = 0; j < B.length; j++) {
        B[j] = B[j] + 1;
}
```

Figure 5: Two similar Java code snippets

Clearly, there is no semantic difference between the two code snippets. However, by changing the array A to B and the looping variable from i to j we have introduced differences on all but the last lines.

The lexical stream of the 2 snippets of code is the following:

```
LITERAL_int LBRACK RBRACK IDENT ASSIGN LCURLY NUM_INT COMMA NUM_INT
COMMA NUM_INT COMMA NUM_INT RCURLY SEMI
LITERAL_for LPAREN LITERAL_int IDENT ASSIGN NUM_INT SEMI IDENT LT
IDENT DOT IDENT SEMI IDENT INC RPAREN LCURLY
NUM_INT SEMI
RCURLY
```

In the above lexical stream, all references to the Array A or the loop counter i are represented as IDENT tokens in the lexical stream. Therefore, both the java snippets will have the exact lexical stream. Changes in the whitespace or comments do not

affect the lexical stream. Using this realization it is easier to develop an algorithm for detecting L0-L2 plagiarism. This method of converting each program to lexical stream and then to check the similarity for expected plagiarism is used by YAP detection system.

- **Detection via Parse Tree Similarities**

The parse tree or derivation tree built from the lexical for a program also exhibits structure for a given program. A compiler, during the compilation process builds a parse tree which represents the program. The parse tree for figure 5 is shown in figure 7.

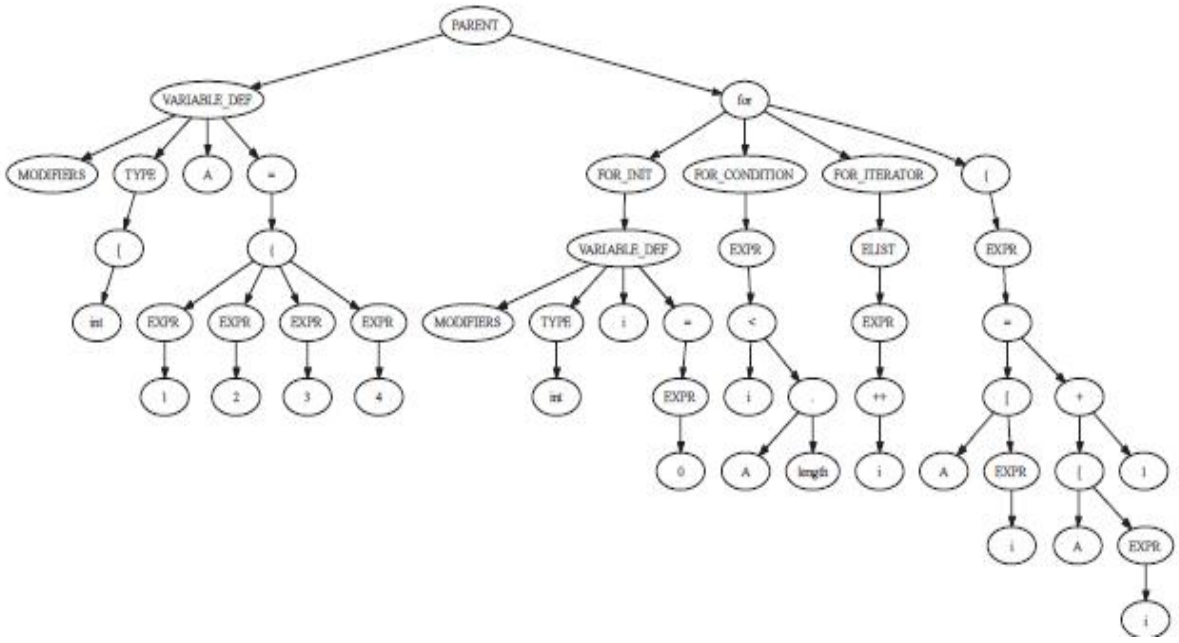

Figure 7: Parse tree for snippet of Java code

The parse tree will have the same structure for both the snippet of code as the lexical streams are same. An algorithm for detecting plagiarism using this method would first, parse each program. Next, for each pair of parse trees, it attempts to find as many common subtrees as possible. This can be done by using the isomorphic rule. Use this number as a measure of similarity between the two programs.

- **Detection via Program Dependence Graphs (PDG)**

A program dependence graph (PDG) is a graph representation of the source code. Basic statements like variable declarations, assignments, and procedure calls are represented by program vertices in PDGs. Each vertex has one and only one type. The types are shown in Table 1.

Type	Description
call-site	Call to procedures.
control	If, switch, while, do-while, or for.
declaration	Declaration for a variable or formal parameter.
assignment	Assignment expression.
increment	++ or −− expression
return	Function return expression.
expression	General expression except the above three, like one with ? operator
jump	Goto, break, or continue
label	Program labels
switch-case	Case or Default

Table 1: Program vertex type

A program dependence graph is a directed, labeled graph which represents the data and the control dependencies within one procedure. It depicts how the data flows between statements and how statements are controlled by other statements. The data and control dependencies between statements are represented by edges between program vertices in PDGs. An example depicting the PDG of the procedure **sum** is shown in Figure 9. Figure 8 explains the program.

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

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

Figure 8: Summation of an array

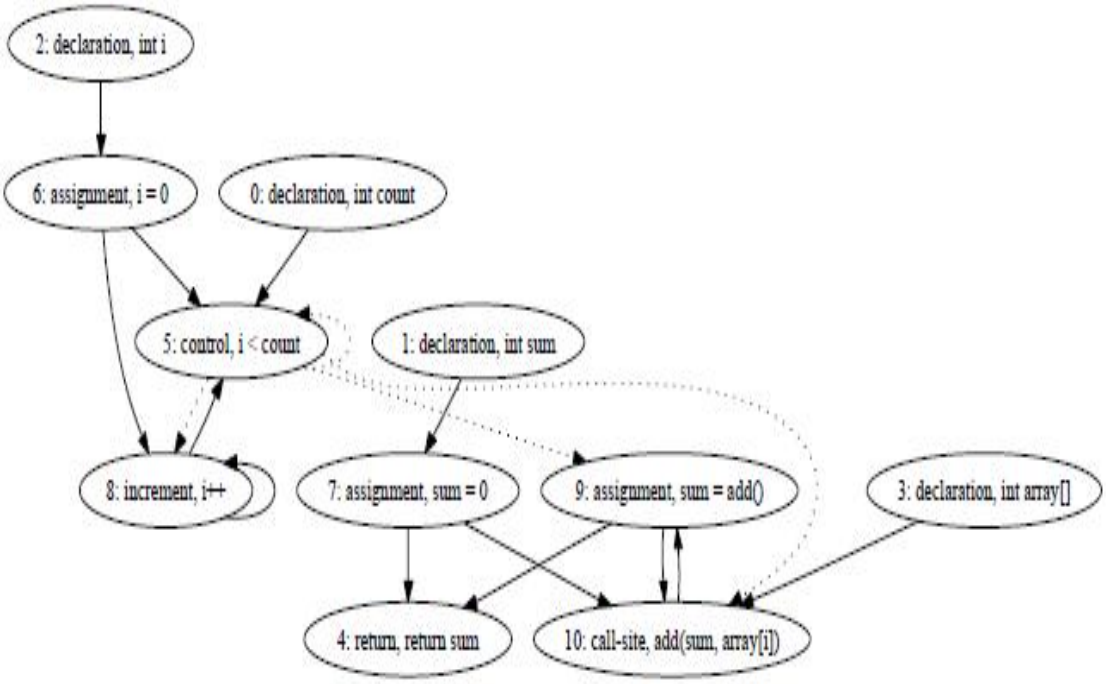

Figure 9: Program Dependence Graph of summation of array

Data and control dependencies are plotted in solid and dashed lines respectively. Specifically, the text inside each vertex gives its vertex id, vertex type and corresponding source code

- **Detection via Metrics**

Similarity analysis using software metrics requires that each fragment be characterized by a set of features measured by metrics such as number of passed parameters, the number of statements etc. Metrics computation requires the parsing of source code to identifying interesting fragments and extract software metrics. In particular, each fragment was modeled by the following seven software metrics:

- Number of function calls (CALLS)
- Number of used or defined local variables (LOCALS)
- Number of used or defined non-local variables (NONLOCALS)
- Number of parameters (PARNUM)

- Number of statements (STMNT)
- Number of branches (NBRANCHES)
- Number of loops (NLOOPS)

Metrics extraction can be performed in a time linear respect to the number of fragments.

5.1.2.2 Tools used for code based plagiarism

- **JPlag**

JPlag finds similarities among multiple sets of source code files. It compares bytes of text as explained in the text based plagiarism tool. JPlag is also aware of the programming language syntax and program structure. It is more robust. It currently supports Java, c#, C, C++ and natural language text. It has a very powerful graphical interface for presenting the results. It does not search the web rather it searches for plagiarism between two given content.

JPlag is made available with a free account but the accounts must be valid with account holders from institutions or organizations.

- **MOSS**

MOSS (Measure Of Software Similarity) checks for plagiarism in programming classes. It has been effectively used as the algorithm used is significantly better compared to other cheating algorithms. It can analyze code written in various languages like C, C++, Python, Visual Basic, Javascript, FORTRAN, Lisp, Ada etc.

MOSS is being provided as an internet service. A list of files is to be given for comparing and this tool checks for any plagiarism. Currently MOSS submission script is for Linux. It highlights the passages that are similar.

- **CodeMatch**

CodeMatch compares thousands of source code files in multiple directories and subdirectories to determine those files which are closely correlated. The technique used here considerably increases the speed of detecting the source code plagiarism. It is also useful for finding open source code within proprietary code, determining common authorship of two different programs, and discovering common, standard algorithms within different programs.

6. DISADVANTAGES OF THE PLAGIARISM DETECTION TECHNOLOGY

- Plagiarism Detection systems are built based on a few languages. To check for plagiarism with the same software can be difficult.
- Most of the detection software checking is done with some repository situated in an organization. Other people are unable to access it and verify for plagiarism.
- As the number of digital copies are going up the repository size should be large and the plagiarism Detection software should be able to handle it.
- There is some plagiarism detection software available which ask us to load a file to their link. Once done the file is copied to their database and then checked for plagiarism. This also comes with an inherent chance of our data being leaked or hacked for other purposes.

7. CASE STUDY

7.1 Efficient and effective plagiarism detection for large repository

A study by Steven Burrows et al, RMIT, Australia saw that many of the plagiarism detection software was very slow and laborious for large number of data. Here a technique is described which uses search engines and local alignment to identify likely matches. Both the techniques are described below.

Search Engines

Web search engine allows users to submit a query as a list of keywords and to receive a list of possibly relevant web pages that contain most or all the keywords. Here the search engine indexes huge volumes of data in a compressed fashion for efficient retrieval. Then an inverted index is used by the search engines to facilitate an efficient retrieval.

An inverted index consists of 2 main components –a lexicon and a series of inverted lists. The lexicon stores all distinct words in the collection. An example of the lexicon is shown in figure 10. The name of fruits is the lexicon in our case. The term banana has a term frequency of, which means that the term banana occurs in one document. The inverted list shows that it appears in the document 26 and the number of occurrences in the document is 2.

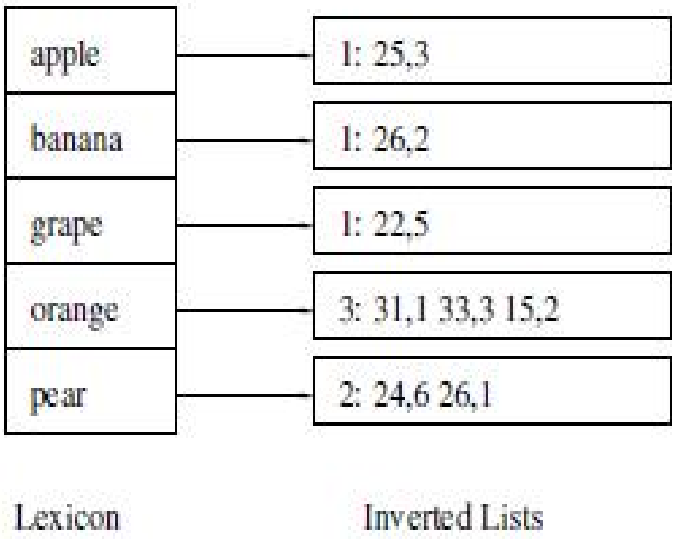

Figure 10: Inverted index list

This index indicates that a search engine is highly scalable for large collections of data. Each term is stored once in the lexicon and the inverted lists comprise of integers that are highly compressible.

The query engine is the component that the user most closely interacts with. The user requests information by providing a query string that the query engine parses. The

query engine retrieves matching inverted lists for the terms the user provided. The ranking engine provides matching results to the user ranked by decreasing estimated relevance. Relevance is estimated using a similarity measure.

Local Alignment

Local alignment is a string matching technique. Local alignment is used for plagiarism detection. Let us take an example and explain the local alignment method.

Consider the local alignment of the two very short sequences "ACG" and "ACT" having a string length of 3. Local alignment can be calculated on two sequences s and t, of lengths l_1 and l_2 using a matrix of size $(l_1 + 1) \times (l_2 + 1)$. The matrix size for our column is initialized with zeros. The rest of the cells will score depending on whether there is a match, mismatch or insertion / deletion between each pair of characters between the two sequences. For every match a score of 1 is given. Subsequent matches will get a score based on the previous score. In all other cases a score of -1 is given.

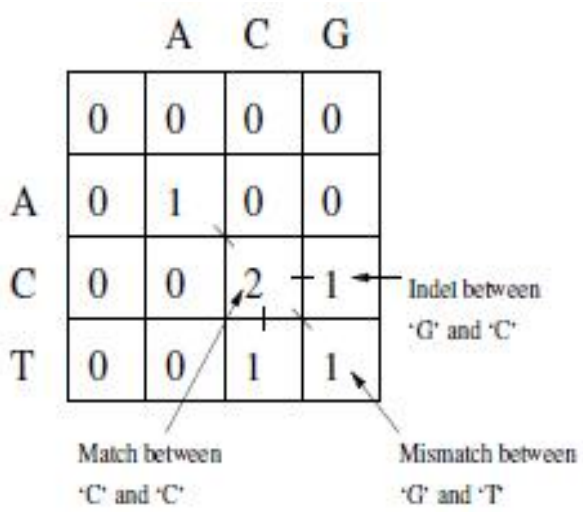

Figure 11: Local Alignment Matrix

In the string "ACG" and "ACT", we see that there is a match in A and C, shown in figure 11. So a merit score is added. Then G and T is encountered and a mismatch occurs so the score in the cell is reduced by 1. Insertion or deletions to optimize the alignment are called as indels. Any indels will again score a -1 with regard to the score

to the previous cell to the left or upwards. In the example taken, any score below zero is taken to be the minimum value of zero. In actual checking there can be a lot of mismatches and indels and so the number of negative values gets increased. So it is better to take a minimum value of zero.

Once the score is put in all the cells the local alignment matrix is traversed to find the highest score in any cell. Then we traceback the path, until we reach a zero. This traceback path denotes the optimal path of high similarity. These are also called as multiple local alignment. Identifying multiple local alignments is much more difficult.

Based on these two methods student assignments were checked for plagiarism. As the student code cannot be directly given for checking the program was tokenized into 2 different formats. As most of the program contains characters, it need not be considered for indexing. The program code is unsuitable for local alignment so we need to tokenize the submissions into another format. Each programmatically significant piece of code is taken as a single piece of code. This also considerably reduces the file size while preserving the program structure. To prepare the code for local alignment, each programming construct is substituted with an alphanumeric character to form a contiguous token stream. Once an index is created then querying can be done. Indexing here has a convenience compared to JPlag, that it builds the index only once and then reuses it. Querying will return the ten most similar programs in a hierarchy. JPlag computes the similarity of all assignment submissions exhaustively every time.

Once the querying is done then local alignment method can be used. This method will evaluate the similarity of program pairs that were highly ranked in the previous step. The input to the local alignment process is the output of our query from the previous step above a threshold similarity score. This information tells whether further program pairs are to be processed or not. The results are presented to the user in order of most similar to least similar. Manual verification is to be done to remove the false positives.

Evaluation

Burrows collected 296 student programs written in the C programming language. He used a repository containing 61,540 programs. This collection was from previously written assignment programs of students. To inspect each program manually is difficult in such large data collection. An automatic first –pass filter was used. The accuracy of plagiarism detected was higher by using this method compared to even JPlag which is a well known plagiarism detection tool.

7.2 CodeMatch

CodeMatch Version 3 is a tool created by Zeidman Consulting, California, U.S. a leading research and development firm. It compares multiple source code files in order to detect plagiarism, copyright infringement, intellectual property theft, and patent infringement. CodeMatch was developed for use by lawyers, technical experts, and expert witnesses to help pinpoint computer source code that may have been copied from another program.

CodeMatch runs on a personal computer using the Windows operating system. CodeMatch uses sophisticated algorithms to compare source code files and rank them according to similarity. CodeMatch compares thousands of source code files in multiple directories and subdirectories to determine which files are the most highly correlated. This can be used to significantly speed up the work of finding source code plagiarism, because it can direct the examiner to look closely at a small amount of code in a handful of files rather than thousands of combinations. CodeMatch is also useful for finding open source code within proprietary code, determining common authorship of two different programs, and discovering common, standard algorithms within different programs.

Previously most of the plagiarism Detection software used an algorithm known as Rabin-Karp algorithm. The Rabin-Karp algorithm used hash functions to improve the speed and accuracy of the detection. CodeMatch improved upon this by using the five algorithms which were already known to people.

The five algorithms are:

- Source Line Matching
- Comment Line Matching
- Word Matching
- Partial Word Matching
- Semantic Sequence Matching

CodeMatch compares every file in one directory with every file in another directory, including all subdirectories if requested. CodeMatch produces a database that can then be exported to an HTML basic report that lists the most highly correlated pairs of files. You can click on any particular pair listed in the HTML basic report see an HTML detailed report that shows the specific items in the files (statements, comments, identifiers, or instruction sequences) that caused the high correlation.

The main disadvantage of using this method is that as it does checking line-by-line and then word-by word and then partial word-by-word the time taken to compare is very slow. The time taken can be in hours or days.

CodeMatch supports the following programming languages: Basic, C, C++, Delphi, Java, MASM, Pascal, Perl, SQL, Verilog, and VHDL. New languages can be added easily upon request.

8. CONCLUSION

Plagiarism is rampant now. With most of the data available to us in digital format the venues for plagiarism is opening up. To avoid this kind of cheating and to acknowledge the originality of the author new detection techniques are to be created. Not only systems with speed but also new systems should which can be able to collect information about plagiarism in the web or large repositories.

As there are a large number of detection tools available for text based plagiarism the number of copying incidents have reduced considerably in this field. Currently we

use a lot of computer based applications. To protect the intellectual property in the source code new techniques are to be developed and implemented.

9. REFERENCES

[1] Steven Burrows 1, Seyed M. M. Tahaghoghi 1 & Justin Zobel, Proceedings of the Second Australian Undergraduate Students' Computing Conference, 2004 http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.69.4500&rep=rep1&type=pdf

[2] Georgina Cosma, Mike Joy, Daniel White and Jane Yau, 9th August 2007 ,ICS, University of Ulster http://www.ics.heacademy.ac.uk/resources/assessment/plagiarism/

[3] Plagiarism detection software, How effective it is?, CSHE, http://www.cshe.unimelb.edu.au/assessinglearning/docs/PlagSoftware.pdf

[4] Maeve Paris, School of Computing & Intelligent Systems, University of Ulster

http://www.ics.heacademy.ac.uk/Events/conf2003/maeve.htm

[5] Dr. Dobb's, http://www.drdobbs.com/184405734;jsessionid=ASPANV2RKXIO3QE1GHOSKH4ATMY32JVN

[6] S.A.F.E, E http://www.safe-corp.biz/products_codesuite.htm