SOFTWARE PLAGIARISM DETECTION

USING ABSTRACT SYNTAX TREE

AND GRAPH-BASED

DATA MINING

By

HSI-YUE SEAN HSIAO

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma

2002

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the Degree of
MASTER OF SCIENCE
May, 2005

SOFTWARE PLAGIARISM DETECTION

USING ABSTRACT SYNTAX TREE

AND GRAPH-BASED

DATA MINING

Thesis Approved:

Dr. Istvan Jonyer
Thesis Adviser

Dr. Debao Chen

Dr. Johnson Thomas

Dr. A. Gordon Emslie
Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere gratitude to my advisor, Dr. István Jónyer, for his intelligent supervision, guidance, and advice. My sincere gratitude extends to my other committee members Dr. Johnson Thomas and Dr. Debao Chen, for their guidance, assistance and suggestions.

I would also like to give my profound appreciation to my beloved grand parents and parents for their encouragement. Thanks also go to my uncles and sister for their selfless support.

Finally, I would like to express my special appreciation to my wife, Jane, for all of her support and love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Since the computer was invented, software plagiarism has always been a serious problem. People can copy other's painstaking efforts, and pretend it is written by them or use it arbitrarily. Plagiarism is even easier to commit in the age of the Internet. At the same time, plagiarism is not easy to catch. Thus, we would like to automate the discovery of cases of plagiarism.

Currently, the techniques for plagiarism detection, such as Attribute Counting System [3, 9], Measure of Software Similarity (MOSS) [10], Yet Another Plague (YAP) [7], and JPlag [13] are focusing on text patterns. In this research, I will use graph-based data mining to analyze the syntactic structure of software, using their abstract syntax tree (AST). The reason we chose to compare the AST of programs rather than the source code itself is because the AST describes the structure of a computer program. We hypothesize that the syntactic structure of software is important in detecting plagiarism, because this structure holds repetitive patterns that only occur in ASTs of similar, potentially plagiarized software. That is, there will be no two similar syntactic structures unless the source codes are similar.

Based on the hypothesis, we created a tool to analyze the AST of computer programs. Based on our hypothesis, the system works by extracting repetitive patterns from ASTs of programs, then compares these patterns. High similarity between the patterns would

mean high likelihood of plagiarism. Sine the AST is a tree structure, we chose a graph-based data mining system called SubdueGL to for the extraction of patterns.

This work is organized as follows. In Chapter 2, I will explain how the text patterned-based techniques work, their advantages and disadvantages. In Chapter 3, I will discuss the fundamental concepts of our software plagiarism detector system, which includes a compiler, graph-based data mining, tree converter, and a graph matcher. In Chapter 4, I will illustrate the experiments we have performed. In Chapter 5, I will discuss the results that we have obtained from the experiments, and the advantages and disadvantages of our system. In Chapter 6, I will conclude this research and discuss the future work.

CHAPTER 2

RELATED WORK

In this chapter, I will discuss the current plagiarism detection concepts. Section 2.1 will introduce the earliest plagiarism detection method–Attribute Counting System. Section 2.2 will introduce the Structure-Metrics System, and Section 2.3 will introduce the Document Fingerprinting Using Graph Grammar Induction system.

2.1 Attribute Counting Systems

The Attribute Counting System (ACS) [3, 9, 12] is the earliest system which used Halstead's software science metrics to detect similarity between program pairs. The measurable and countable properties are:

n1 = number of unique or distinct operators

n2 = number of unique or distinct operands

N1 = total usage of all of the operators

N2 = total usage of all of the operands

ACS focuses on the reserved words in a programming language to judge similarity, and uses the above properties to calculate the similarity metrics. However, G. Whale has demonstrated that a system based on attribute counters is incapable of detecting similar programs [11].

2.2 Structure-Metric Systems

2.2.1 Measure of Software Similarity

Measure of Software Similarity (MOSS) [10] is a system that detects document-based textual similarity. It works as follows. First, it collects the significant words from the desired document and cancels the noise data. The noise data, such as comments, white space, punctuations, and capitalizations, can be ignored by applying “whitespace insensitivivity” and “noise suppression”. After cleaning the data of noise, MOSS will combine the rest of the strings and divide them into small substrings by k-grams. K-grams are the number of characters of a substring. For example, 3-grams means the length of each substring is three. Then MOSS will assign index numbers which are created by a hash function to represent each substring. Then MOSS will compare the index numbers of two files to judge similarity. The MOSS system is available online, and allows users to send documents or source code files over the internet.

2.2.2 Yet Another Plague

Yet Another Plague (YAP) [7] is a system that also focuses on text patterns, but it works differently than MOSS. First, YAP reorganizes the source code by doing the following: Remove comments, translate upper-case to lower case, map of synonyms to a common form, reorder functions, and remove all tokens that are not reserved words. Basically, YAP wants to find a maximal set of common contiguous substrings. YAP3, the third version of YAP, is the newest version which was introduced in 1996.

2.2.3 JPlag

JPlag WWW system allows users to compare the documents over the internet. It is

another system which works with the text pattern. JPlag system operates in two phases. The first phase converts the source code into token strings. In the first phase, the “noise cancellation” is applied to ignore white spaces, comments, and names of identifiers. And the second phase compares each token string that from the first phase to the substring of some token strings from the first phase. The “Greedy String Tiling” method is used for the comparison and generating the similarity value. JPlag was publicly available online since 2001. It supports C, C++, and Java currently.

2.3 Document Fingerprinting Using Graph Grammar Induction

Document Fingerprinting Using Graph Grammar Induction (DFGGI) was introduced in 2004 [1]. It converts the source code to a graph based on the textual relationship, and use graph-based data mining technique to find the fingerprint of the source code. Then it compares the fingerprints to judge the similarity between two source codes. Essentially, this system is another branch of using graph-based data mining to detect plagiarism behaviors, except DFGGI focuses on the textual relationship and our system focuses on the structure of the source code.

CHAPTER 3

SOFTWARE PLAGIARISM DETECTOR

In this chapter, I will discuss all the concepts and algorithm that involved in this research. Section 3.1 is the overview of our software plagiarism detector. Section 3.2 will discuss the compiler and how the abstract syntax tree extracted. Section 3.3 will discuss the format converter which will be applied to convert AST format to Subdue format. Section 3.4 will discuss the graph-based data mining algorithm–Subdue. Section 3.5 will discuss the similarity computing method–Submatch.

3.1 Overview

Unlike the existing structure-metric plagiarism detecting methods, we chose to examine the structure of programs in the form of the abstract syntax tree (AST). In our system, we first send the source code to the compiler, which parses the code and generates the AST. Then, we extract repetitive patterns from the AST using a graph-based data mining system, called SubdueGL. Lastly, we compare the patterns reported by SubdueGL on pairs of ASTs, to arrive at the measure of similarity.

The compiler to be used in the system is dependent of the programming language used. We restricted our study to the C language, and used the GNU C compiler, gcc. The output from gcc must be converted to a graph format that is compatible with SubdueGL, which is accomplished using a simple format converter. After the graph files are analyzed by SubdueGL, the results will be compared by a graph matching program called

Submatch that computes the similarity between graphs. Figure 1 shows a diagram of the system, and how it is applied to compare two computer programs.

Our system consists of four major parts: AST extraction, graph conversion, SubdueGL, and the Submatch system. In the remaining portion of this chapter, I introduce each of these in detail.

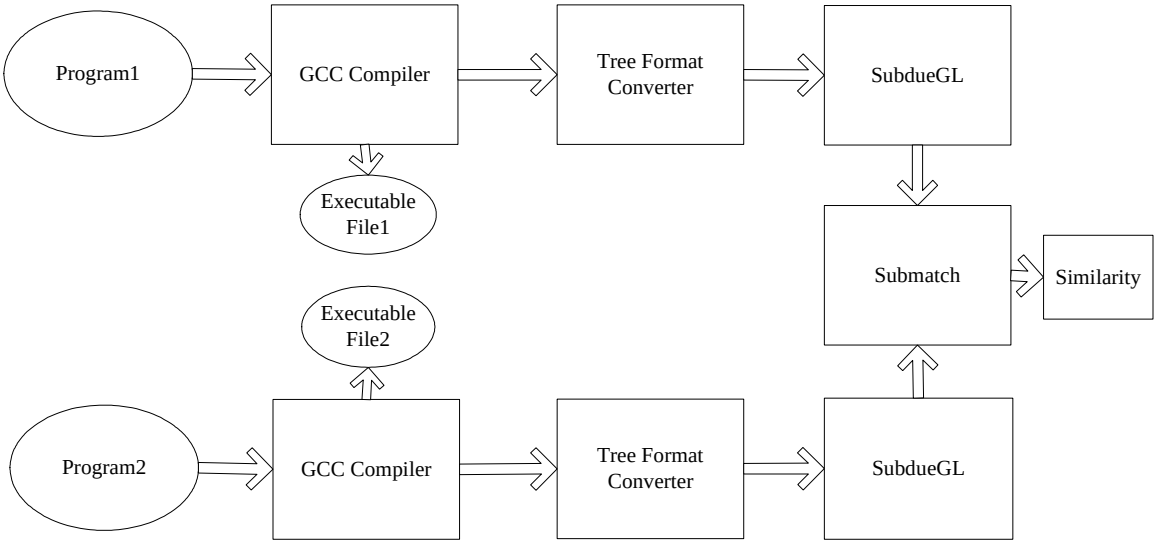

Figure 1. Diagram of Software Plagiarism Detector

3.2 Extracting the Abstract Syntax Tree

In our system we use a compiler to parse the source code into the abstract syntax tree. In the following section, I briefly describe how a compiler works. Section 3.2.1 will introduce the general idea of compiler; Section 3.2.2 will introduce how the abstract syntax tree is generated.

3.2.1 Overview

Compilers act as translators. They transform human-oriented programming languages into computer-oriented machine languages. A modern compiler is organized

into several phases. Figure 2 shows the general phases of the compiler.

In this research, we only use the first three phases, which are lexical analysis, parsing and semantic actions. This is because the semantic actions stage of the compiler generates the full abstract syntax tree, and the rest of the stages are not needed.

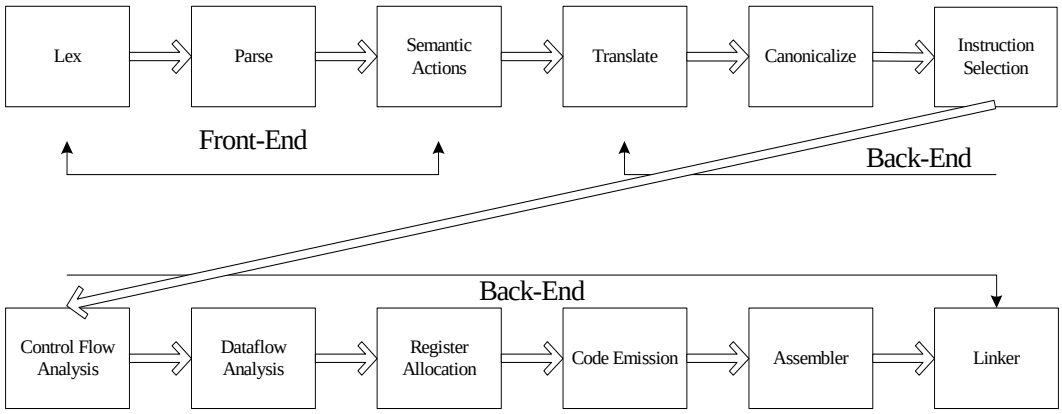

Figure 2. Phases of compiler

3.2.2 Generating the Abstract Syntax Tree

If a language is a set of strings, then a string is a finite sequence of symbols. The symbols themselves are taken from a finite alphabet. Therefore, in the lexical analysis phase of a compiler the source code is converted to regular expressions, and then these regular expressions are converted to deterministic finite automata (DFA). The reason we use DFA versus non-deterministic finite automata (NFA) is because no two edges leading from the same state are labeled with the same symbol, and DFA are easy to implement by a computer language. Figure 3 is an example of how lexical analyzer works.

For parsing, we can view the string as a source program, the symbols as lexical tokens, and the alphabet as the set of token types returned by the lexical analyzer. The parse stage analyzes the phrase structure of the program. It uses a context-free grammar

to describe the programming language.

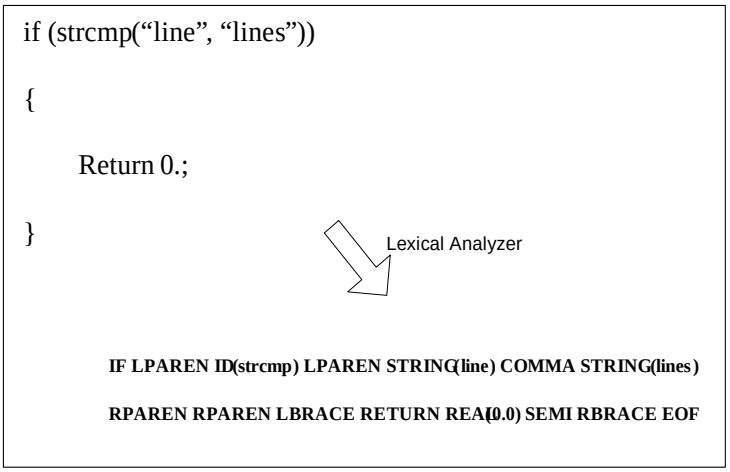

Figure 3. Example of lexical analyzer

After the parse tree is built, the compiler will construct a syntax tree representation of the input program. It indicates its relationship to the actual syntax and parse tree. Since the compilation process is driven by the syntactic structure of a source program, in this tree, the compiler needs to do semantic processing. For the code “i = i + 1”, the tree will look as in Figure 4,

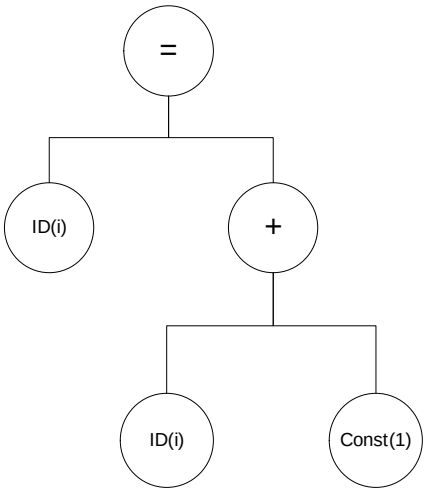

Figure 4. Example of AST

After the Semantic Actions phase, the abstract syntax tree is built, which can be output to a text file.

3.2.4 GNU Compiler Collection (GCC)

GNU Compiler Collection (GCC) is a free compiler which is developing by the GCC team. The first beta version was released in March 1987 and the latest stable version is 3.4.3 which was released in November 2004.

GCC contains two parts which are the “front-end” and the “back-end”. The front-end will convert the all the general concepts that can be found in all high level languages to a stack-based assembly language–RTL. The back-end will optimize the RTL and convert it into the machine language which is recognized by CPU.

There are many different kinds of programming language compilers. The reason we chose GCC is because it can compile several popular programming languages such as C, C++, Objective-C, Fortran, Java, and Ada and it shares the same back-end, which means if GCC want to include a new language, it only need a new front-end for the new language. We can use GCC to generate AST’s for all of the languages that GCC supports.

3.3 Abstract Syntax Tree to SubdueGL Converter

The format of an AST which is generated byg++ is different from the input format required by SubdueGL. So, we need a converter to transform the format. Also, there is some unnecessary (noise) information in the original AST format that we have to filter during the transformation process.

3.3.1 AST Format

There are several options in g++ which generate abstract syntax trees. These options are –fdump-tree-original, –fdump-tree-optimized, and –fdump-translation-unit.

We chose the –fdump-tree-original option because we wanted to compare the AST’s

without any modification. The usage is “g++ –fdump-tree-original filename.c”. It will output the AST tree to filename.c.original as shown in Figure 5.

Each subprogram has its own AST. The first two lines of each section are to describe the function of this tree. The “@” symbol stands for a node. For example, @1 is node 1. The name following the node number is the node name. Following the node name are the edge name and destination. For example, node 1 in the first tree above is named function_decl; it has 3 edges which link it to node 2, node 3, and node 4.

```
;; Function int main() (main)
;; enabled by -dump-tree-original
@1      function_decl    name: @2       type: @3       srcp: round.c:9
                 C            extern       body    : @4
@2      identifier_node  strg: main     lngt: 4
@3      function_type    size: @5       algn: 64       retn: @6
                 prms   : @7
@4      compound_stmt    line: 20       body: @8       next: @9
@5      integer_cst      type: @10      low : 64
…
@132    integer_cst      type: @107     low : 33
@133    pointer_type     size: @61      algn: 32       ptd : @38
@134    fix_trunc_expr   type: @6       op 0: @39
;; Function int round(double) (_Z5roundd)
;; enabled by -dump-tree-original
@1      function_decl    name: @2       mngl: @3       type: @4
                 srcp   : round.c:23            args: @5
                 extern       body   : @6
@2      identifier_node  strg: round    lngt: 5
@3      identifier_node  strg: _Z5roundd            lngt : 9
…
@33     fix_trunc_expr   type: @8       op 0: @34
@34     plus_expr        type: @11      op 0: @5       op 1: @35
@35     real_cst         type: @11
```

Figure 5. AST dumping results from g++

3.3.2 Convertion to SubdueGL Format

The SubdueGL system input format consists of two parts, vertices and edges. Vertices used in edge definitions must always be defined before the edge that uses them. In SubdueGL’s input format, v stands for vertex followed by the vertex number and the vertex’s name. e stands for edge followed by the source vertex number, the target vertex

number, and the edge name. Figure 6 shows an example of SubdueGL input format.

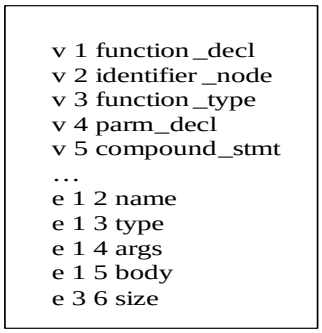

Figure 6. Input graph format of SubdueGL

The way I convert the AST format to SubdueGL format is as follows. For each line the converter reads, if this line contains ";;" symbol, then omit this line. If this line contains "@" symbol, then analyze the first two tokens, and convert the "@" symbol to v, then store this part into a file named V. The converter also needs to store the vertex number at the same time. This number will be used as the source number of the edge. All other vertices part will append to this file. The next step is to analyze the rest of the line. If a token contains the "@" symbol, then convert the "@" symbol to e and store this part include the vertex number that we stored it earlier into a file name E. During the edge part conversion, the converter will also filter some unnecessary information. This kind of information will be discussed in next 3.4.3. This process will continue until it reaches the next ";;" symbol. After each process, the converter has to combine the V file and the E file together, and delete the V and E file; this is for a function. After the whole file has been converted, the converter will combine all the functions together. Figure 7 is the converter flowchart.

We have also created the alternate conversion way which to provide us another view of conversion. We substituted all the variable names to a string–`identifier_node`.

The reason we did this is because we wanted to know whether the variable name affects the results.

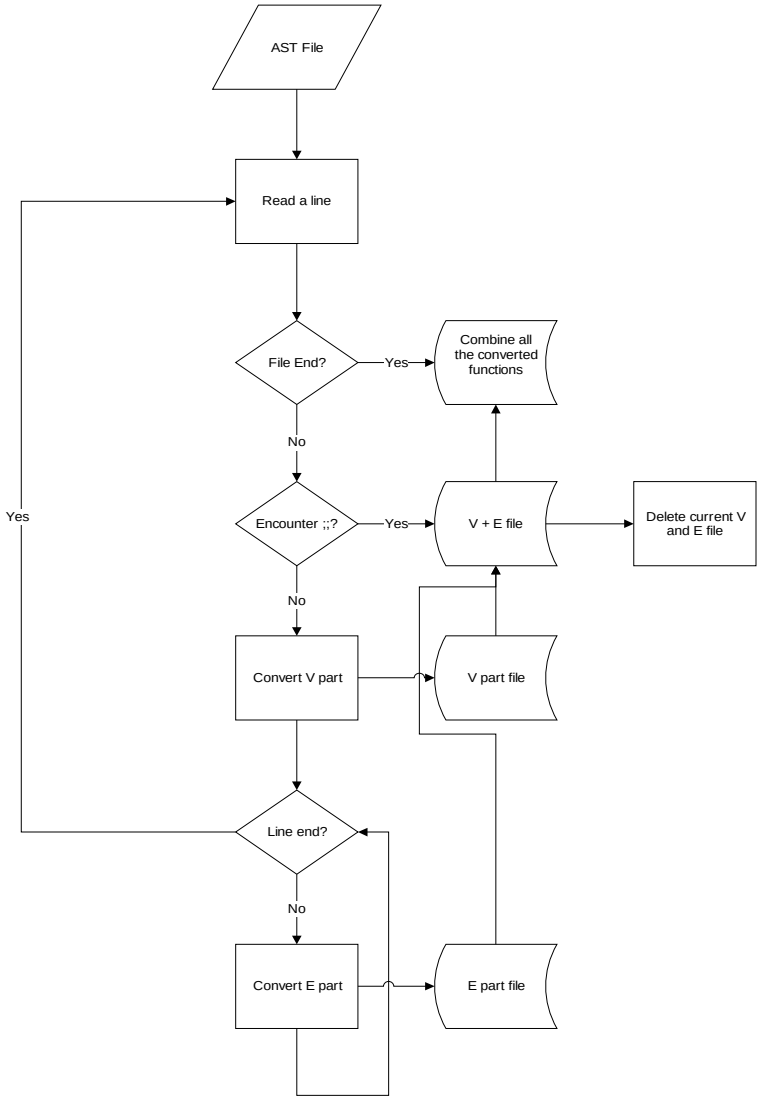

Figure 7. Converter flowchart

3.3.3 Noise cancellation

In the original AST format, there is some information which is useful for the compiler but not needed in the SubdueGL system. The noise data in Figure 8 are underlined. For example, the lines beginning with ";;" are comments and `srcp: round.c:9` in @1 is to tell the compiler where is this function begins in the

source code. We have to filter this kind of information because it does not have vertex or edge information.

```
;; Function int main() (main)
;; enabled by-dump-tree-original
@1     function_decl   name: @2      type: @3      srcp: round.c:9
                C          extern      body: @4
@2     identifier_node  strg: main    lngt: 4
@3     function_type   size: @5      algn: 64      retn: @6
                prms  : @7
@4     compound_stmt   line: 20      body: @8      next: @9
@5     integer_cst     type: @10     low : 64
…
@132   integer_cst     type: @107    low : 33
@133   pointer_type    size: @61     algn: 32      ptd : @38
@134   fix_trunc_expr  type: @6      op 0: @39
```

Figure 8. Example of noise data

3.4 Subdue

The Subdue knowledge discovery system [2, 4, 5, 6, 8] is the tool that analyzes the abstract syntax tree in our system. In this section, I introduce the Subdue system. Section 3.4.1 will introduce the fundamental ideas; Section 3.4.2 will introduce the SubdueGL algorithm which is an extension of Subdue.

3.4.1 The Subdue knowledge discovery system

The Subdue knowledge discovery system is created at University of Texas Arlington [2, 4, 5, 6, 8]. It finds repetitive subgraphs (substructures) in the input data, which is a labeled, directed graph. The search is driven by the minimum description length principle (MDL) which was introduced by Rissanen (1989). The MDL principle has been used for wide area, such as image processing, decision tree induction, concept learning for relational data, and learning models of non-homogeneous engineering domains. The MDL heuristic can calculate the value of a substructure using the formula

$$Value(s) = DL(S) + DL(G \mid S) \qquad \text{(eq. 1)}$$

where $DL(S)$ is the description length of the substructure, G is the input graph, and $DL(G|S)$ is the description length of the input graph compressed by the substructure. The Figure 9 is the MDL example graph, and I will use it to illustrate how MDL works.

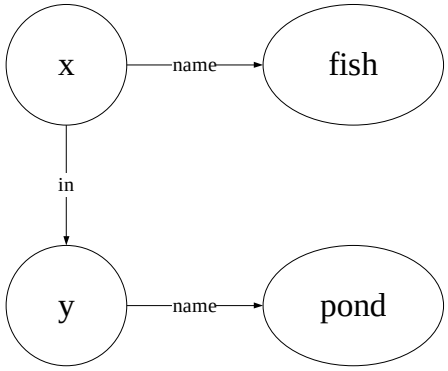

Figure 9. MDL example graph

First, we define the minimum description length of a graph to be the number if bits which is necessary to completely describe the graph. The bits include three parts–vbits, rbits, and ebits. The vbits is the number of bits which needs to encode the vertex labels of the graph. The rbits is the number of bits which needs to encode the row of the adjacency matrix A. The adjacency matrix A represents the graph connectivity. If A[i,j]=1, then there is a connection between vertex i and vertex j. If A[i,j]=0, then there is no connection between vertex i and vertex j. The ebits is the number of bits which needs to encode the edges represented by the entries A[i,j]=1 of the adjacency matrix A. In this example, the adjacency matrix is shown below.

$$\begin{array}{c} \\ x \\ fish \\ y \\ pond \end{array} \begin{bmatrix} \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Then MDL uses the following three equations to calculate the bits.

$$vbits = \lg v + v \lg l_u \quad \text{(eq. 2)}$$

$$rbits = (v+1)\lg(b+1) \sum_{i=1}^{v} \lg \binom{v}{k_i} \quad \text{(eq. 3)}$$

$$ebits = e(1 + \lg l_u) + (K+1)\lg m \quad \text{(eq. 4)}$$

where v is number of vertices in the graph, l_u is number of unique labels in the graph, k_i is the number of 1 in i^{th} row of adjacency matrix, b is the maximum number of k_i, e is the number of edges in the graph, K is the number of 1s in the adjacency matrix A, and m is the maximum number of edges between vertex i and vertex j.

For the example in Figure 9, v = 4, l_u = 6, b = 2, e = 3, K =3, and m =1. Therefore

$$vbits = \lg 4 + 4 \lg 6 = 12.34,$$

$$rbits = 5\lg(3) + \lg\binom{4}{2} + \lg\binom{4}{0} + \lg\binom{4}{1} + \lg\binom{4}{0} = 12.51, \text{ and}$$

$$ebits = 3(1 + \lg 6) + (3+1)\lg 1 = 10.75.$$

The total encoding of graph needs $12.34 + 12.51 + 10.75 = 35.6$ bits.

The Subdue algorithm works as follows. First, it begins by collecting all single-vertex subgraphs, each of which may have many instances. The algorithm finds the subgraphs that are deemed the best by the MDL heuristic. Then, it expands the best substructures by all neighboring edges, one at a time, creating new substructures. After a substructure is discovered, each instance of the substructure in the input graph will be replaced by a single vertex. The best structure will be saved in a list. After all the possible substructures have been evaluated or the computation exceeds a user-defined limit, the algorithm returns the best structures. Figure 10 shows the pseudo code of the Subdue

algorithm.

```
Subdue ( graph G, int Beam, int Limit )
  queue Q = { v | v has a unique label in G }
  bestSub = first substructure in Q
  repeat
    newQ = {}
    for each S in Q
      newSubs = S extended by an adjacent edge from G
                in all possible ways
      newQ = newQ U newSubs
      Limit = Limit - 1
    evaluate substructures in newQ by compression of G
    Q = substructures in newQ with top Beam compression
        scores
    if best substructure in Q better than bestSub
    then bestSub = best substructure in Q
  until Q is empty or Limit <= 0
  return bestSub
```

Figure 10. Pseudo code of Subdue

3.4.2 SubdueGL

Based on the Subdue approach, an extended algorithm called Subdue Grammar Learner, or SubdueGL [2, 4, 5, 6] was created. SubdueGL is a bottom-up graph grammar learning algorithm and to discover the common substructures in graphs. The graph grammar is a set of grammar production rules that describe a graph-based database. In SubdueGL, if a grammar production is found, it will be replaced by a non-terminal graph. Like Subdue, SubdueGL is driven by the minimum description length (MDL) principle as well. SubdueGL keeps the substructures when it is the best of MDL heuristic evaluation. SubdueGL iterates until the whole input graph is replaced by a single non-terminal graph, or a user-defined stopping condition is reached. In other words, SubdueGL will generate

all the possible grammar rules for the input graph.

There are two features in SubdueGL which are not found in the original Subdue: detecting recursion and variables. If an instance of substructure is connected to any other instances by an edge, it is possible to have a recursive production. In Figure 12, the square-looking subgraph is an example of a recursive production. Variable-detection is for substructures in which all instances have the same structure, as in a regular substructure, but some of the labels differ in the same isomorphic position. These vertices can be substituted by variables. Figure 14 is an example of variable or alternative production.

Figure 11 to Figure 16 are the example of how SubdueGL works. Figure 11 is the input graph; it contains two triangle shape structures, two square shape structures, a vertex, and several edges. The letters in the vertices are the name of the vertex. The letters on the edges are the name of the edge.

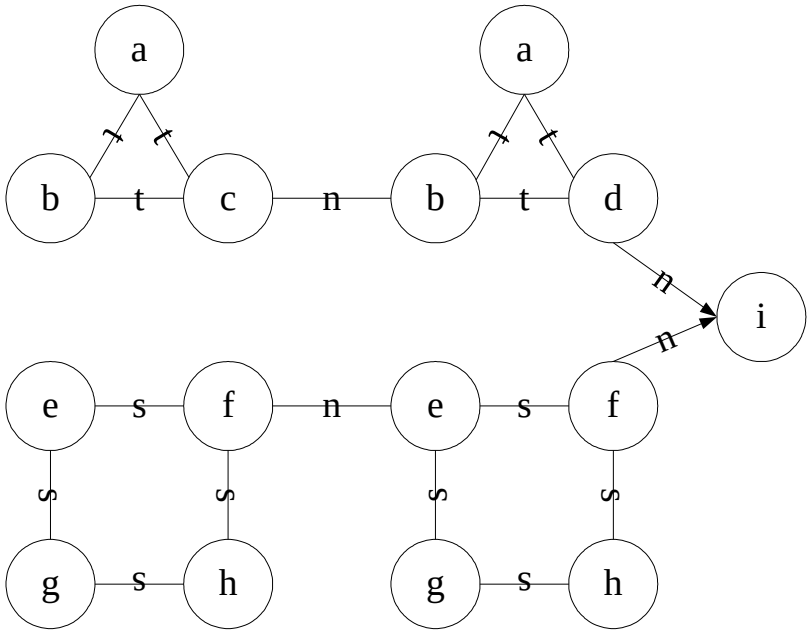

Figure 11. Example of input graph of SubdueGL

Let us follow vertex “e” and keep in mind that all the expansions are working in parallel. So, start from the vertex “e”, it generates some 2-vertex substructures–(e, s, f), (e, s, g), and (f, n, e). Since the first two substructures have two instances and the last one has one instance, we choose the compress the first two substructures. After several expansions, the substructure will have vertices {e, f, g, h}, it is the first biggest and common substructure. Also, the recursive production found that this {e, f, g, h} substructure is a recursive grammar rule. It can be replaced by a single vertex shown in Figure 12. Figure 13 is the result of extension of vertex “e”.

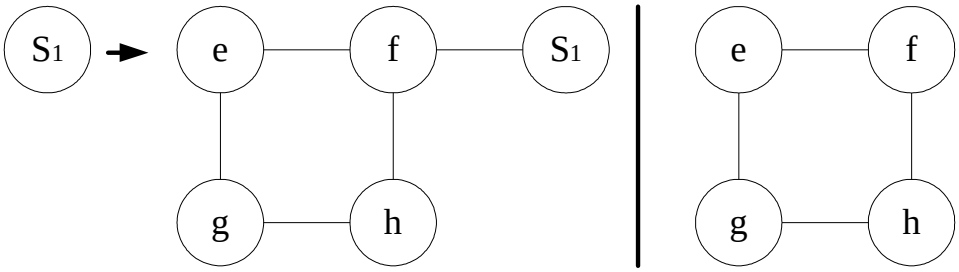

Figure 12. First production

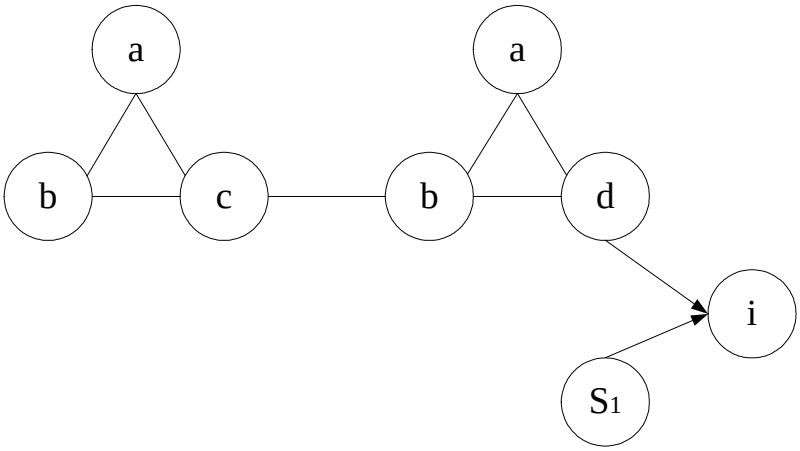

Figure 13. Input graph after the first production

In the next iteration we use the graph after the first production as input graph. In these two triangle shape graphs, a vertex {a, b} and edge (a, t, b) can be found as a

substructure and have two instances. However, if the extension goes further, by an edge, it will encounter different label name–vertex “c” and vertex “d”. Although the shape is the same, SubdueGL cannot compress this substructure directly. SubdueGL will use variable-detection to substitute those label names. In this example, SubdueGL will replace it with a non-terminal variable S3, shown in Figure 14. The second production is shown in Figure 15. Figure 16 shows the fully parsed graph.

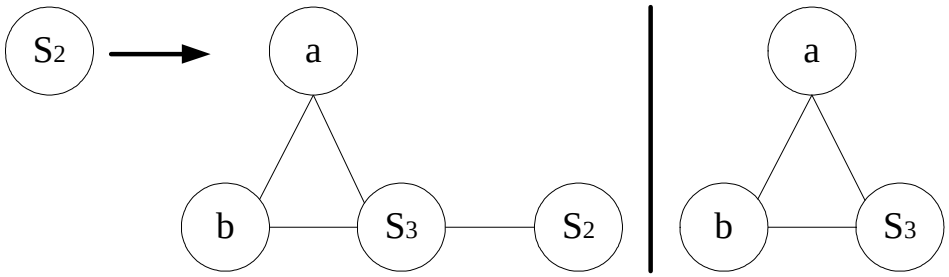

Figure 14. Second production

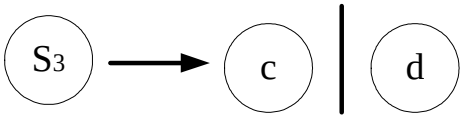

Figure 15. Third production

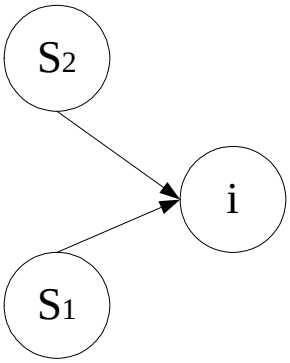

Figure 16. Final production returned by SubdueGL

To operate SubdueGL, we need to know the options of SubdueGL. There are about 37 options of SubdueGL. I will introduce some options that are commonly used. For

example, `-save` is for saving the result in predefined substructure file format under the name <input graph file>.s. `-exhaust` can be used with `-cluster` to exhaustively analyze the graph. That is, even if there is no compression, but there are original vertices to classify, clustering will continue. `-gg` enables discovery of graph grammars. It also turns on recursion, variables, and relationship finding. `-norecursion` disables the discovery of recursive substructures, when turned on by `-gg`. `-novariables` disables the discovery of variables, when turned on by `-gg`.

3.5 Computing the Similarity

The last stage of our system is graph match. SubdueGL will generate a fingerprint, which consists of grammar rules. These are the significant substructures of the AST. We use a program called Submatch to compare the grammars which are generated by SubdueGL. Submatch will take two grammar files and output the similarity in percentage between these two files.

3.5.1 Submatch

Submatch is a program which compares two grammar files from the SubdueGL system. It loads two grammar files and compares each substructure in the first output file (G1) to the every substructure in the second output file (G2). Once G1 finds a match in G2, the substructure in G2 will be removed. In Figure 17, the best structure 1 of program 1 compares all the best structures of program 2, and found a match in best structure 3 of program 2. Best structure 3 of program 2 is removed. Then the best structure 2 of program 1 compares all the best structures of program 2, and found a match in best structure 2 of program 2. Best structure 2 of program 2 is removed. Then the best

structure 3 of program 1 compares all the best structures of program 2, and found a match in best structure 1 of program 2. Best structure 3 of program 1 is removed and Submatch is terminated

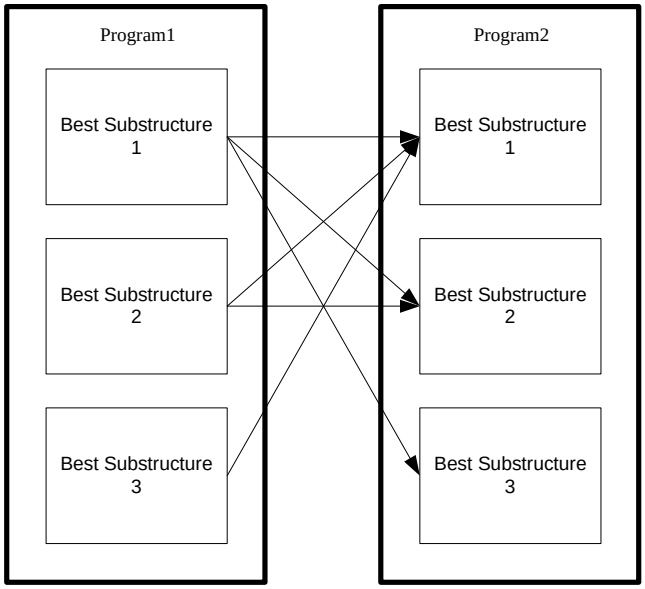

Figure 17. Example comparison performed by Submatch

If a substructure in the first output file is similar to a substructure in the second output file, the transformation cost is equal to the edge difference between G1 and G2. The transformation cost is estimated by an “Inexact graph match” algorithm which is also used in SubdueGL system. Inexact graph match was designed to deal with those graphs with slightly differences by Bunke and Allermann (1983). In this algorithm, every transformation such as insertion, deletion, and substitution of vertices and edges will be assign a cost. SubdueGL system uses branch-and-bound search to extend Bunke’s algorithm to get better performance. Branch-and-bound search can guarantee an optimized solution, the search ends as soon as the first complete mapping is found.

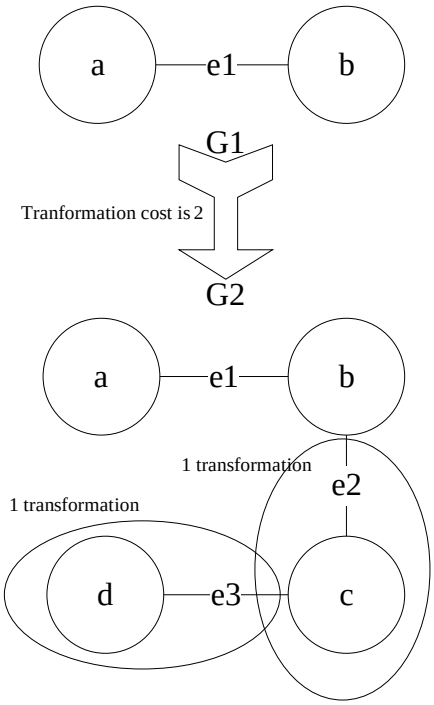

Figure 18. Example of inexact graph match

In figure 18, vertex a, vertex b and edge e1 are the same in both graphs. So the transformation cost between G1 and G2 is 2 by adding 2 sets of vertices and edges. Submatch will use the total value of transformation cost to determine the similarity by using equation 4.

$$Similarity = \frac{TotalSubstructureG1 - TotalTransformationCost}{TotalSubstructureG1} * 100\% \qquad \text{(eq. 4)}$$

Figure 19 is an example that how the Submatch compares two result files of SubdueGL. Grammar 1 contains three best substructures. Grammar 2 contains two best substructures. Figure 20 is the graph representation of Grammar 1 and Grammar 2. From Figure 20 we can observe the first rule in Grammar 1 matches the second rule in Grammar 2. Since they are matched perfectly, no transformation cost is needed and the second rule in Grammar 2 is removed. The second rule in Grammar 1 matches the first rule in Grammar 2. Again, there is no transformation cost is needed because they are

matched perfectly and the second rule in Grammar 2 is removed. Now, there are no more rules in Grammar 2 but one more left in Grammar 1. Therefore, the third rule in Grammar 1 needs three transformation cost because there are one vertex and two sets of vertices and edges need to be added to an empty graph to match the third rule in Grammar 1. The total substructure of G1 is 24, so the similarity is equal to (24 - 3) / 24 * 100% = 87.5%.

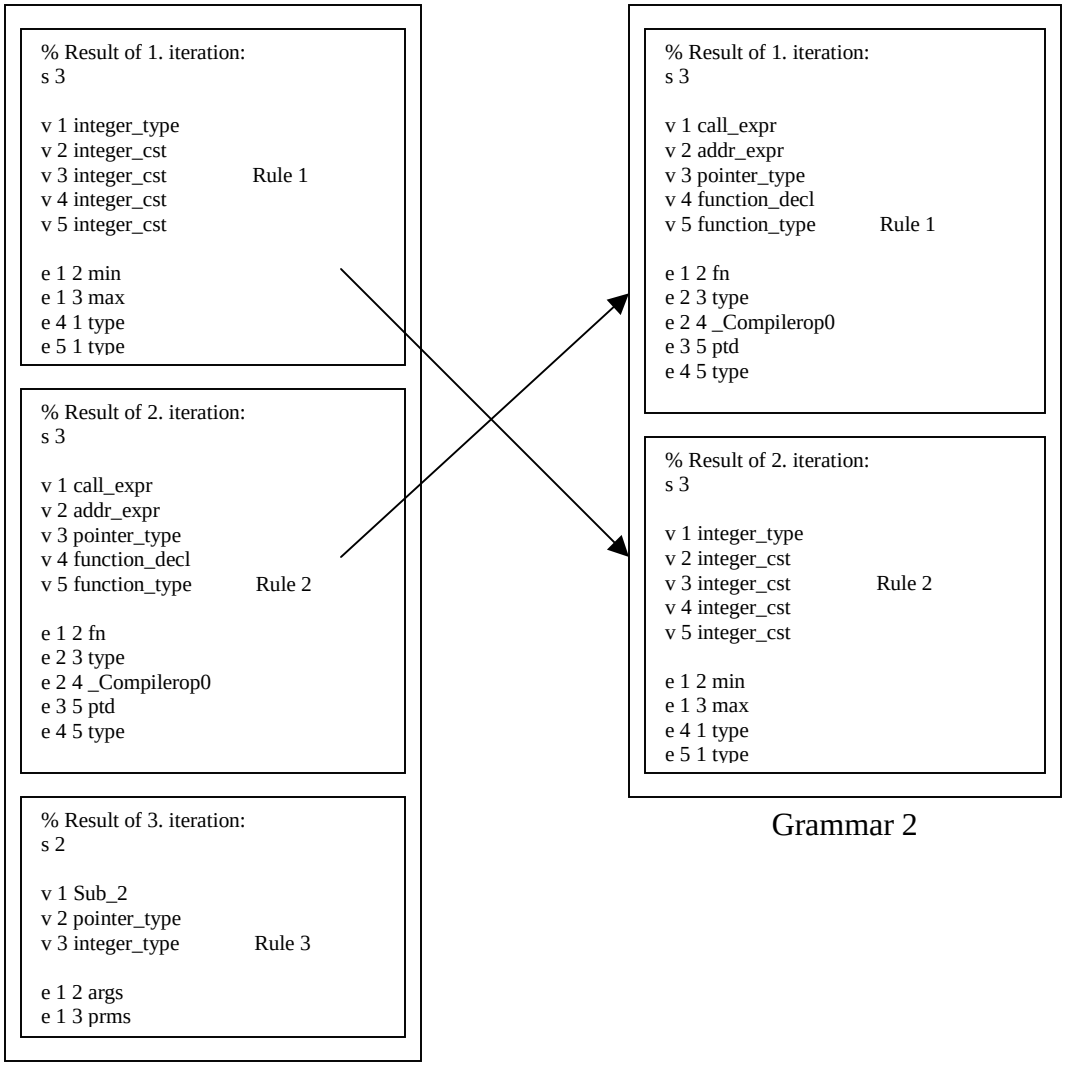

Figure 19. Example of comparing two results of SubdueGL

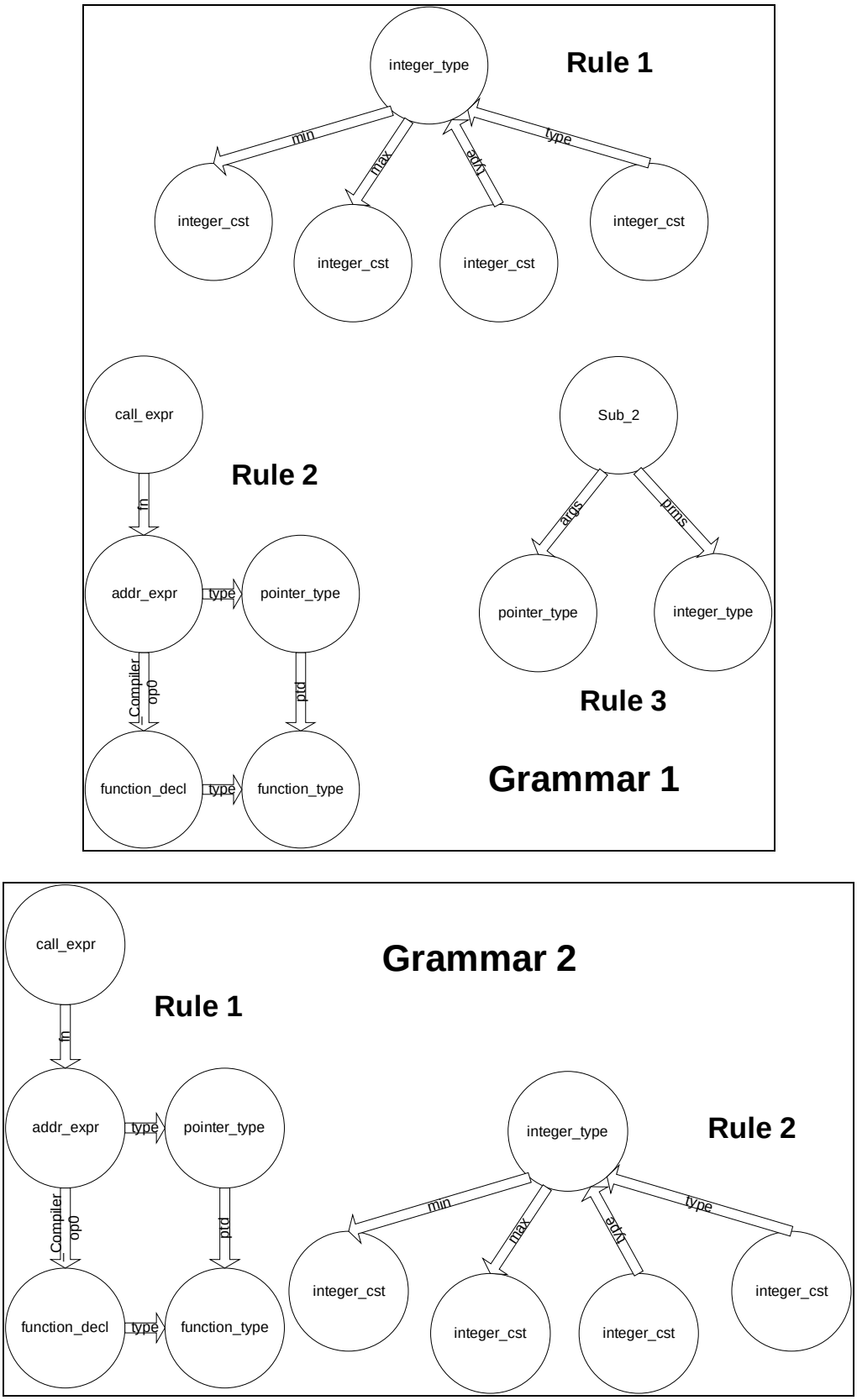

Figure 20. Graph representation of Figure 19

CHAPTER 4

EXPERIMENTS

In this research, we performed three experiments. Section 4.1 describes an artificial domain. In this experiment, we use a source code written in C to test our concept. Section 4.2 describes experiments using real world data. We arbitrarily chose 19 students' source codes from Assignment 1 and 20 students' source codes from Assignment 2. We also used MOSS, DFGGI, and JPlag to compare the data in section 4.1 and 4.2 for comparison.

4.1 Artificial Domain

In this experiment, we used a C source code named "AD.c". AD contains seven functions, and the length of AD is 788 lines. We also created some test programs which were slightly modified from the original AD program. There were five test programs which were renamed three to fifteen variables from the original AD program, and five test programs which were rearranged the function sequence.

SubdueGL discovers recursion and variables. These options were discussed in Chapter 3.3.2. We would like to know how these two options affect our system, so we performed experiments to see what options give us the best results. In other words, we would like to know which options generate the fingerprints that work best for plagiarism detection. Results are shown in Table 1.

From the results, we observed that the recursions and variables options only make

slight differences for variable renamed data; however it makes a big difference for the function rearranged data. So, we chose to turn off those two options to get better results in both rearranged and renamed data.

Description	variables & recursions	variables & norecursions	novariables & recursions	novariables & norecursions
Itself	100.00%	100.00%	100.00%	100.00%
Renamed 3 Variables	100.00%	100.00%	100.00%	100.00%
Renamed 6 Variables	100.00%	100.00%	100.00%	100.00%
Renamed 9 Variables	99.86%	99.87%	100.00%	100.00%
Renamed 12 Variables	99.70%	99.77%	100.00%	100.00%
Renamed 15 Variables	99.64%	99.68%	100.00%	100.00%
Rearranged 1 time	21.57%	20.44%	97.07%	95.43%
Rearranged 2 times	35.46%	28.92%	95.00%	98.22%
Rearranged 3 times	28.12%	25.72%	68.68%	98.22%
Rearranged 4 times	33.18%	31.27%	68.68%	98.22%
Rearranged 5 times	21.93%	23.31%	97.87%	100.00%

Table 1. Results of experiment of choosing SubdueGL options

Next, we had to decide which conversion should be used in this research. In Chapter 3.4.2, I introduced two methods of conversion, the original which I used to perform the previous experiment, and the variable substitution. We performed experiments to help us to determine which is the best way.

In Table 2, we observed that variable substitution does affect the results. Using

"identifier_node" to substitute the original variable names only affects the results of rearranged data. However, we still chose "with variables" to be our future conversion method. The reason is both methods can provide us the high percentage results, but with variables method provides more variable information to us. This information is important especially for our "Partial Test" method which will be introduced later in this Chapter.

Description	With Variables	Without Variables
Itself	100.00%	100.00%
Renamed 3 Variables	100.00%	100.00%
Renamed 6 Variables	100.00%	100.00%
Renamed 9 Variables	100.00%	100.00%
Renamed 12 Variables	100.00%	100.00%
Renamed 15 Variables	100.00%	100.00%
Rearranged 1 time	95.43%	100.00%
Rearranged 2 times	98.22%	100.00%
Rearranged 3 times	98.22%	100.00%
Rearranged 4 times	98.22%	100.00%
Rearranged 5 times	100.00%	100.00%

Table 2. Results of experiment of choosing conversion method

There is a function of our software plagiarism detector that no other systems currently do. In this approach, we can compare each function of the program separately, because each program has its own AST and they are independent. The benefit of doing this is if the plagiarism happened only in one or two functions, the similarity between the two programs is low but will be high in individual functions.

During the conversion process, each AST was converted and combined to SubdueGL format file. However, those before-combination individual files still exist and are in SubdueGL format. We can compare these individual files one by one to find out which function is plagiarized. This is the reason that we keep the variables during the conversion process, because it can help us to identify which function shows sign of plagiarism. The italic numbers in Table 3 show the plagiarized functions. No matter how

the variables or the sequence changed, we observed that our system can find the functions plagiarized.

	AD-0	AD-1	AD-2	AD-3	AD-4	AD-5	AD-6
AD-0	***100.00%***	2.07%	8.88%	12.73%	37.50%	0.35%	27.42%
AD-1	2.07%	***100.00%***	15.29%	4.55%	4.13%	10.25%	4.55%
AD-2	8.88%	15.29%	***100.00%***	7.69%	5.33%	14.49%	0.00%
AD-3	12.73%	4.55%	7.69%	***100.00%***	6.94%	0.00%	25.81%
AD-4	37.50%	4.13%	5.33%	6.94%	***100.00%***	4.24%	19.44%
AD-5	0.35%	10.25%	14.49%	0.00%	4.24%	***100.00%***	0.00%
AD-6	27.42%	4.55%	0.00%	25.81%	19.44%	0.00%	***100.00%***
AD_R1-0	2.07%	***100.00%***	15.29%	4.55%	4.13%	10.25%	4.55%
AD_R1-1	8.88%	15.29%	***100.00%***	7.69%	5.33%	14.49%	0.00%
AD_R1-2	12.73%	4.55%	7.69%	***100.00%***	6.94%	0.00%	25.81%
AD_R1-3	37.50%	4.13%	5.33%	6.94%	***100.00%***	4.24%	19.44%
AD_R1-4	0.35%	10.25%	14.49%	0.00%	4.24%	***100.00%***	0.00%
AD_R1-5	27.42%	4.55%	0.00%	25.81%	19.44%	0.00%	***100.00%***
AD_R1-6	***100.00%***	2.07%	8.88%	12.73%	37.50%	0.35%	27.42%
AD_R2-0	2.07%	***100.00%***	15.29%	4.55%	4.13%	10.25%	4.55%
AD_R2-1	8.88%	15.29%	***100.00%***	7.69%	5.33%	14.49%	0.00%
AD_R2-2	12.73%	4.55%	7.69%	***100.00%***	6.94%	0.00%	25.81%
AD_R2-3	0.35%	10.25%	14.49%	0.00%	4.24%	***100.00%***	0.00%
AD_R2-4	27.42%	4.55%	0.00%	25.81%	19.44%	0.00%	***100.00%***
AD_R2-5	***100.00%***	2.07%	8.88%	12.73%	37.50%	0.35%	27.42%
AD_R2-6	37.50%	4.13%	5.33%	6.94%	***100.00%***	4.24%	19.44%
AD_R3-0	8.88%	15.29%	***100.00%***	7.69%	5.33%	14.49%	0.00%
AD_R3-1	12.73%	4.55%	7.69%	***100.00%***	6.94%	0.00%	25.81%
AD_R3-2	0.35%	10.25%	14.49%	0.00%	4.24%	***100.00%***	0.00%
AD_R3-3	2.07%	***100.00%***	15.29%	4.55%	4.13%	10.25%	4.55%
AD_R3-4	27.42%	4.55%	0.00%	25.81%	19.44%	0.00%	***100.00%***
AD_R3-5	***100.00%***	2.07%	8.88%	12.73%	37.50%	0.35%	27.42%
AD_R3-6	37.50%	4.13%	5.33%	6.94%	***100.00%***	4.24%	19.44%
AD_R4-0	8.88%	15.29%	***100.00%***	7.69%	5.33%	14.49%	0.00%
AD_R4-1	27.42%	4.55%	0.00%	25.81%	19.44%	0.00%	***100.00%***
AD_R4-2	0.35%	10.25%	14.49%	0.00%	4.24%	***100.00%***	0.00%
AD_R4-3	2.07%	***100.00%***	15.29%	4.55%	4.13%	10.25%	4.55%
AD_R4-4	***100.00%***	2.07%	8.88%	12.73%	37.50%	0.35%	27.42%
AD_R4-5	37.50%	4.13%	5.33%	6.94%	***100.00%***	4.24%	19.44%
AD_R4-6	12.73%	4.55%	7.69%	***100.00%***	6.94%	0.00%	25.81%
AD_R5-0	27.42%	4.55%	0.00%	25.81%	19.44%	0.00%	***100.00%***
AD_R5-1	8.88%	15.29%	***100.00%***	7.69%	5.33%	14.49%	0.00%
AD_R5-2	0.35%	10.25%	14.49%	0.00%	4.24%	***100.00%***	0.00%
AD_R5-3	2.07%	***100.00%***	15.29%	4.55%	4.13%	10.25%	4.55%
AD_R5-4	***100.00%***	2.07%	8.88%	12.73%	37.50%	0.35%	27.42%
AD_R5-5	37.50%	4.13%	5.33%	6.94%	***100.00%***	4.24%	19.44%
AD_R5-6	12.73%	4.55%	7.69%	***100.00%***	6.94%	0.00%	25.81%
AD_V1-0	***100.00%***	2.07%	8.88%	12.73%	37.50%	0.35%	27.42%
AD_V1-1	2.07%	***100.00%***	15.29%	4.55%	4.13%	10.25%	4.55%
AD_V1-2	8.88%	15.29%	***100.00%***	7.69%	5.33%	14.49%	0.00%
AD_V1-3	12.73%	4.55%	7.69%	***100.00%***	6.94%	0.00%	25.81%
AD_V1-4	37.50%	4.13%	5.33%	6.94%	***100.00%***	4.24%	19.44%

AD_V1-5	0.35%	10.25%	14.49%	0.00%	4.24%	***100.00%***	0.00%
AD_V1-6	27.42%	4.55%	0.00%	25.81%	19.44%	0.00%	***100.00%***
AD_V2-0	***100.00%***	2.07%	8.88%	12.73%	37.50%	0.35%	27.42%
AD_V2-1	2.07%	***100.00%***	15.29%	4.55%	4.13%	10.25%	4.55%
AD_V2-2	8.88%	15.29%	***100.00%***	7.69%	5.33%	14.49%	0.00%
AD_V2-3	12.73%	4.55%	7.69%	***100.00%***	6.94%	0.00%	25.81%
AD_V2-4	37.50%	4.13%	5.33%	6.94%	***100.00%***	4.24%	19.44%
AD_V2-5	0.35%	10.25%	14.49%	0.00%	4.24%	***100.00%***	0.00%
AD_V2-6	27.42%	4.55%	0.00%	25.81%	19.44%	0.00%	***100.00%***
AD_V3-0	***100.00%***	2.07%	8.88%	12.73%	37.50%	0.35%	27.42%
AD_V3-1	2.07%	***100.00%***	15.29%	4.55%	4.13%	10.25%	4.55%
AD_V3-2	8.88%	15.29%	***100.00%***	7.69%	5.33%	14.49%	0.00%
AD_V3-3	12.73%	4.55%	7.69%	***100.00%***	6.94%	0.00%	25.81%
AD_V3-4	37.50%	4.13%	5.33%	6.94%	***100.00%***	4.24%	19.44%
AD_V3-5	0.35%	10.25%	14.49%	0.00%	4.24%	***100.00%***	0.00%
AD_V3-6	27.42%	4.55%	0.00%	25.81%	19.44%	0.00%	***100.00%***
AD_V4-0	***100.00%***	2.07%	8.88%	12.73%	37.50%	0.35%	27.42%
AD_V4-1	2.07%	***100.00%***	15.29%	4.55%	4.13%	10.25%	4.55%
AD_V4-2	8.88%	15.29%	***100.00%***	7.69%	5.33%	14.49%	0.00%
AD_V4-3	12.73%	4.55%	7.69%	***100.00%***	6.94%	0.00%	25.81%
AD_V4-4	37.50%	4.13%	5.33%	6.94%	***100.00%***	4.24%	19.44%
AD_V4-5	0.35%	10.25%	14.49%	0.00%	4.24%	***100.00%***	0.00%
AD_V4-6	27.42%	4.55%	0.00%	25.81%	19.44%	0.00%	***100.00%***
AD_V5-0	***100.00%***	2.07%	8.88%	12.73%	37.50%	0.35%	27.42%
AD_V5-1	2.07%	***100.00%***	15.29%	4.55%	4.13%	10.25%	4.55%
AD_V5-2	8.88%	15.29%	***100.00%***	7.69%	5.33%	14.49%	0.00%
AD_V5-3	12.73%	4.55%	7.69%	***100.00%***	6.94%	0.00%	25.81%
AD_V5-4	37.50%	4.13%	5.33%	6.94%	***100.00%***	4.24%	19.44%
AD_V5-5	0.35%	10.25%	14.49%	0.00%	4.24%	***100.00%***	0.00%
AD_V5-6	27.42%	4.55%	0.00%	25.81%	19.44%	0.00%	***100.00%***

Table 3. Partial program comparison

The last experiment of this artificial domain compares MOSS, DFGGI, and JPlag. MOSS and JPlag are mature system and DFGGI uses graph grammar induction like our system does. We would like to know how our system compares to MOSS, DFGGI, and JPlag system deal with our artificial domain. Table 4 shows the results of comparing our system to MOSS, DFGGI, and JPlag system. As we can see, all systems can find plagiarism in the artificial domain.

The above experiments tell us that our software plagiarism detector is working. It also helps us to determine which options are appropriate for use in future experiments.

Description	**SOFTWARE PLAGIARISM DETECTOR**	**DFGGI**	**JPlag**	**MOSS**

Itself	100.00%	100.0%	100.0%	99%
Renamed 3 Variables	100.00%	99.3%	100.0%	99%
Renamed 6 Variables	100.00%	97.5%	100.0%	99%
Renamed 9 Variables	100.00%	98.0%	100.0%	99%
Renamed 12 Variables	100.00%	97.6%	100.0%	99%
Renamed 15 Variables	100.00%	96.7%	100.0%	99%
Rearranged 1 time	96.50%	44.4%	100.0%	99%
Rearranged 2 times	96.50%	37.2%	99.3%	99%
Rearranged 3 times	96.50%	40.6%	99.7%	99%
Rearranged 4 times	96.50%	47.2%	99.3%	97%
Rearranged 5 times	96.75%	43.1%	99.7%	97%

Table 4. Software plagiarism detector versus MOSS

4.2 Real World Experiments

In this chapter, I will use the software plagiarism detector to deal with real world data. The data we chose are actual student programming assignments. There are two assignments, both written in C or C++. We performed three experiments on these source codes.

In Assignment 1, the comprehensive test detected two sets of programs (4,9) and (8,9) have over 60% similarities in Table 5.

Then we used MOSS on Assignment 1. The results are extracted in Table 6. It also indicated the 8th and the 9th programs are the most similar programs in the data set 1. But MOSS only got 36% on the program 4 and 8. Essentially, MOSS system also detected higher similarity among the programs 4, 8, 9, and 19, these four source codes are also the highest percentage in our system.

We also applied DFGGI and JPlag to Assignment 1. Table 7 shows the results of DFGGI. It indicates a 65% similarity between the programs 8 and 9 program. Table 8 shows the results of JPlag, and it indicates a 67% similarity between the program 8 and 9.

1	100																		
2	23	100																	
3	26	26	100																
4	41	34	25	100															
5	26	24	21	33	100														
6	29	38	21	34	23	100													
7	32	19	19	28	19	17	100												
8	41	33	32	50	39	33	30	100											
9	43	32	24	***63***	30	31	29	***67***	100										
10	33	21	22	29	18	26	24	27	25	100									
11	31	27	14	34	16	50	20	29	28	23	100								
12	43	27	22	36	27	31	19	39	37	30	29	100							
13	45	27	19	45	24	38	17	36	43	31	33	46	100						
14	20	37	18	22	18	35	17	23	23	20	31	27	27	100					
15	21	40	20	20	15	29	21	22	20	21	32	22	27	35	100				
16	23	23	15	31	25	26	20	26	26	24	14	26	24	21	25	100			
17	16	23	12	17	8	23	12	16	14	13	21	21	20	19	19	19	100		
18	30	26	24	40	30	26	28	39	38	27	26	34	26	15	20	34	16	100	
19	32	28	38	39	23	23	23	48	49	28	20	28	27	18	18	16	10	25	100
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	***19***

Table 5. Comprehensive test on Assignment 1

File 1	File 2	Lines Matched
./p1/8.c (59%)	./p1/9.c (58%)	231
./p1/4.c (36%)	./p1/8.c (35%)	133
./p1/19.c (35%)	./p1/9.c (32%)	117
./p1/19.c (35%)	./p1/8.c (32%)	110
./p1/4.c (32%)	./p1/9.c (31%)	100
./p1/19.c (32%)	./p1/4.c (30%)	105
./p1/15.c (18%)	./p1/7.c (15%)	70
./p1/1.c (17%)	./p1/7.c (14%)	70
./p1/16.c (14%)	./p1/7.c (13%)	58
./p1/2.c (16%)	./p1/7.c (12%)	57
./p1/7.c (12%)	./p1/8.c (15%)	45

Table 6. MOSS on Assignment 1

In Assignment 2, the comprehensive test detected two sets of programs (4, 9, and 10). Table 9 shows the results of the comprehensive test on Assignment 2.

2	28																
3	34	40															
4	30	45	46														
5	27	41	47	44													
6	27	36	23	36	33												
7	37	50	48	51	59	33											
8	35	46	41	57	52	30	48										
9	34	49	44	52	52	28	48	**65**									
10	49	51	43	44	44	26	46	46	50								
11	24	47	41	45	44	33	43	47	50	34							
12	18	28	13	25	29	37	25	28	30	31	34						
13	51	47	45	47	44	27	50	51	51	55	50	33					
14	48	46	40	41	41	25	45	44	46	48	43	30	50				
15	33	45	44	44	51	33	43	43	48	43	44	34	42	38			
16	35	49	53	45	54	36	50	50	52	41	49	36	42	40	48		
17	32	44	35	39	40	27	36	40	41	42	48	31	38	37	36	38	
18	31	37	41	42	41	40	41	45	46	39	41	42	42	34	37	38	43
19	19	33	40	45	47	34	36	43	43	25	38	34	29	21	40	40	35
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**

Table 7. DFGGI test on Assignment 1

1	100																	
2	11	100																
3	-	-	100															
4	-	-	-	100														
5	-	-	32	-	100													
6	-	-	-	-	-	100												
7	24	11	14	-	14	-	100											
8	-	11	-	54	17	-	-	100										
9	-	-	-	36	-	-	-	***67***	100									
10	-	-	-	-	-	-	-	-	-	100								
11	-	-	-	-	-	-	-	10	-	-	100							
12	-	-	-	-	-	-	-	-	-	-	-	100						
13	-	-	-	-	-	-	-	-	-	-	-	-	100					
14	13	-	-	-	-	-	10	-	-	-	-	-	11	100				
15	16	-	-	-	12	-	14	-	-	-	-	-	-	-	100			
16	-	-	-	-	-	-	-	-	17	-	-	-	-	-	-	100		
17	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	100	
18	-	12	-	-	-	-	-	11	11	-	-	-	-	-	-	15	-	100
19	-	-	-	30	-	-	-	42	37	-	-	-	-	-	-	17	-	11
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**

- : Similarity < 10

Table 8. JPlag test on Assignment 1

We applied MOSS to Assignment 2, and the results were extracted in Table 10. It provided similar results, except that, according to their system, the most similar source codes are the 4th and the 9th programs, while according to our system the most similar source codes are the 9th and the 10th programs. But both systems caught these three plagiarism programs.

1	100																		
2	24	100																	
3	28	20	100																
4	31	42	13	100															
5	30	41	20	41	100														
6	24	15	36	19	22	100													
7	37	18	29	23	27	21	100												
8	24	16	25	21	22	25	33	100											
9	27	51	14	***57***	52	17	19	21	100										
10	25	45	19	***61***	53	17	16	22	***82***	100									
11	34	45	34	18	25	27	40	27	24	20	100								
12	27	45	24	38	39	11	31	24	43	40	27	100							
13	34	26	22	24	34	23	36	27	26	24	22	32	100						
14	18	10	20	9	11	16	17	14	9	10	16	11	17	100					
15	25	31	20	19	42	25	25	18	26	28	24	24	25	9	100				
16	35	19	38	18	25	35	34	29	18	17	32	27	32	22	14	100			
17	28	14	24	14	25	16	34	19	16	18	32	23	25	22	21	29	100		
18	30	43	18	53	42	19	32	24	47	45	22	51	28	12	28	23	15	100	
19	34	21	27	19	29	22	35	26	23	23	50	25	28	17	19	21	22	22	100
20	18	28	8	26	28	14	16	23	28	27	24	30	35	14	29	19	15	27	16	100
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**

Table 9. Comprehensive test on Assignment 2

Table 11 shows the results of applying DFGGI on Assignment 2. The DFGGI system provided similar results of our system this time. It indicates the 4th, the 9th, and the 10th programs are the most similar programs of Assignment 2.

File 1	File 2	Lines Matched
./p3/4.c (78%)	./p3/9.c (77%)	238
./p3/10.c (39%)	./p3/9.c (43%)	129
./p3/10.c (36%)	./p3/4.c (40%)	121
./p3/11.c (38%)	./p3/16.c (21%)	88
./p3/20.c (6%)	./p3/7.c (7%)	17
./p3/5.c (5%)	./p3/9.c (7%)	20
./p3/4.c (7%)	./p3/5.c (5%)	20
./p3/10.c (6%)	./p3/5.c (5%)	20
./p3/15.c (6%)	./p3/7.c (5%)	9
./p3/18.c (5%)	./p3/8.c (4%)	9
./p3/1.c (5%)	./p3/7.c (5%)	9

Table 10. MOSS on Assignment 2

Table 12 shows the results of applying JPlag on Assignment 2. JPlag found the programs 4, 9, and 10 are the most similar programs of Assignment, except it does not indicates high similarity between the programs 4 and 10.

2	25																	
3	39	40																
4	33	47	44															
5	52	38	42	44														
6	31	37	38	45	33													
7	32	44	34	40	23	24												
8	33	40	42	32	30	40	33											
9	27	46	39	**69**	27	36	42	18										
10	31	45	29	**60**	32	29	49	18	**60**									
11	44	44	35	41	42	35	41	22	45	46								
12	54	36	33	42	54	19	46	13	41	46	36							
13	50	35	20	41	53	23	46	15	35	41	34	48						
14	46	39	16	43	44	21	49	13	40	45	34	42	42					
15	42	45	42	48	35	28	49	18	46	50	37	34	35	37				
16	35	37	41	31	33	36	33	26	30	35	16	32	34	40	32			
17	40	42	29	41	40	44	41	42	41	41	39	39	39	40	40	41		
18	49	37	33	42	50	20	44	15	39	45	38	50	47	45	47	27	34	
19	40	42	29	41	40	44	41	42	41	41	39	39	39	40	40	41	26	39
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**

Table 11. DFGGI test on Assignment 2

1	100																	
2	-	100																
3	-	6	100															
4	6	-	-	100														
5	-	-	-	13	100													
6	-	-	-	-	-	100												
7	-	-	-	-	-	5	100											
8	-	-	-	-	-	-	8	100										
9	6	-	-	***82***	9	-	-	-	100									
10	6	-	-	35	9	-	-	-	***52***	100								
11	5	-	-	-	-	-	-	-	-	-	100							
12	-	-	-	-	-	-	-	-	-	-	-	100						
13	-	-	-	-	-	-	-	-	-	-	-	-	100					
14	-	-	-	-	-	-	-	-	-	-	-	-	-	100				
15	-	-	-	-	-	5	-	-	-	-	-	-	-	-	100			
16	-	-	-	-	-	-	-	-	-	-	37	-	-	-	-	100		
17	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	100	
18	7	-	-	14	9	9	6	7	12	-	-	-	-	-	-	-	-	100
19	5	-	-	14	5	6	-	-	-	-	7	-	-	-	-	6	-	-
20	-	-	-	-	-	-	-	8	-	-	-	-	-	-	-	-	-	-
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**

19	**19**	100	
20	**20**	7	100
		19	**20**

- : Similarity < 5

Table 12. JPlag on Assignment 2

The last experiment of real world data is to test our partial plagiarism detection concept. In this experiment, I chose two source codes from Assignment 1, and named it A and B. These two source codes, without cheating behavior, were written in C++ language. Then I copied two functions in A to B, and made sure it could pass the compiler. The comprehensive test shows a similarity of 38.08%, and MOSS system shows 21%. However, in our partial plagiarism detection idea, it is easy to find out which functions are the same. In Table 13, we can see that the first function in A and the sixth function in

B are the same; the second function in A and the seventh function in B are the same as well. There are also some 100.00% functions in Table 13. The reason of this situation is because they use the same library. I will discuss more in Chapter 5.

0	***100***	2	1	9	0	2	4	2	18	9	0	2	0	7		
1	4	4	5	9	2	3	***100***	2	5	8	0	24	13	36		
2	2	2	14	7	33	5	2	***100***	6	10	0	12	0	16		
3	0	13	3	9	0	24	1	0	10	10	0	2	8	2		
4	6	4	0	3	13	4	33	16	5	0	8	56	23	94		
5	9	28	23	9	1	32	0	0	33	14	2	1	0	2		
6	6	4	16	11	41	14	5	54	12	14	2	14	0	20		
7	8	6	9	8	9	7	55	16	9	8	0	26	33	39		
8	0	5	5	5	65	8	4	63	9	0	11	20	0	22		
9	0	0	2	2	0	0	5	12	3	4	0	14	0	21		
10	0	7	9	7	7	6	61	18	5	8	0	25	15	41		
11	0	0	1	6	36	0	5	25	9	0	13	23	0	15		
12	0	4	10	5	13	10	0	25	4	0	6	17	2	17		
13	0	4	10	5	13	10	0	25	4	0	6	17	2	17		
14	0	12	18	26	4	17	2	6	25	20	0	2	1	1		
15	0	0	0	0	9	0	0	0	0	0	***100***	21	0	4		
16	2	0	0	2	16	2	24	12	3	0	21	***100***	17	50		
17	0	0	0	4	0	6	13	0	3	0	0	17	***100***	21	38	21
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**Comprehensive**	**MOSS**

Table 13. Partial test on real world data

CHAPTER 5

DISCUSSION

In this chapter, I would like to discuss the results we got from the experiments. The results provide some useful information to help us find out how this approach works. It also raises some new questions which we did not think about at the beginning.

During the real world data experiments, we found an edge information that will affect the result. This edge is called `scpe` which stands for scope. Scope is the variable visibility of the compiler. If the variable is global, then its scope is the whole program. If the variable is local, then its scope is that local function. In other word, local variables only can be used in the local function. Global variables can be used by the whole program and local function cannot declare the variable with the same name. We are considering deleting this specific edge information is because in the experiments we found the same function but in different place of the program will have one difference in AST which is the scope edge. This edge will make the whole AST structure contain different repetitive structure and the SubdueGL system will generate different grammars. There are not only one scope edge in the AST tree, the specific one we want to filter is the one which does not link to the original (@1) point. Table 14 is the results of scope cancellation experiment on the artificial domain. The results of experiment with scope from the AST are slight better than the without one which matches our expectation. These results raise another question–why this scope edge only affects the real world data? From the results in Table 14, we can observe that cancel the scope edge will not affect the

result too much. Our guess is the artificial domain we used did not generate the repetitive substructures like the real world data did. In other words, we think it is a coincidence that the artificial domain was not affected by the scope edge. Therefore, we want to take this scope edge out. Again, we are not focus on the actual percentage of similarity, as long as they both are high. Cancel the scope edge can provide us more consistent experiments, because we do not want this kind of unexpected information which is generated by the compiler affect the AST structure.

In our fundamental assumption, we should compare the AST information without any modification. However, this scope edge affects the results too much, we have no choice but to remove this edge. We do not know whether this scope edge will affect the results in other languages. It needs further research to proof.

Description	With SCOPE	Without SCOPE
Itself	100.00%	100.00%
Renamed 3 Variables	100.00%	100.00%
Renamed 6 Variables	100.00%	100.00%
Renamed 9 Variables	100.00%	100.00%
Renamed 12 Variables	100.00%	100.00%
Renamed 15 Variables	100.00%	100.00%
Rearranged 1 time	100.00%	96.50%
Rearranged 2 times	100.00%	96.50%
Rearranged 3 times	100.00%	96.50%
Rearranged 4 times	100.00%	96.50%
Rearranged 5 times	100.00%	96.75%

Table 14. Experiment results of canceling scope

In the rest of this Chapter, I would like to discuss the advantage and the disadvantages of our software plagiarism detection system. The advantage of our system is partial function comparison. The current structure-metrics system cannot find if the program contains some plagiarism functions, especially when the functions are relatively small in the whole program. This problem is because their systems have to work with the entire source code; but in this research, the compiler separates the functions for us. So we

can compare those separated functions to find whether plagiarism occurs in those functions or not.

There are four disadvantages. The first one is the abstract syntax tree generating problem. In the hypothesis, we assume the GCC can provide us AST for all kinds of languages that it supports; however, we can only use the dump option for g++ which means we only can compare C and C++ languages. I tried to modify the source code of GCC and extracted the AST information for JAVA successfully; however, that information is used for real-time compiler processing. It is impossible to modify it into the AST information we want, because it is too large and complex. If this problem is solved, then this system can be used in those popular languages.

The second problem is the compiler problem. This is the major problem of this research, because the compiler is the source of this research. It is in charge of generating the abstract syntax tree. However, it generates some information that we did not expect. For example, there are only two functions regarding to plagiarism in Table 13; however, we have seen more than two 100% similarities in the table. After we dug into those functions, we found the compiler generates the AST not only for the functions, but also for some specific library or declaration ways. For example, both source codes use “namespace” at the beginning of the code, so the AST of the first functions are for the namespace. It will cause two problems; one is that we don’t know how the compiler generates the AST, we cannot prevent it. The second one is, if one of two similar programs contains several “namespace”, the similarity of comprehensive test will be lower. This problem can be solved by using the partial test approach. Table 13 is an example.

The third problem is about a program can be compiled or not will affect the result. We found that if a source code cannot pass the compiler, the compiler will only generate a partial AST. This partial generation occurs because the stages of the compiler are working in parallel. The AST is building when the compiler is parsing the rest of the code. So, if an error occurs, the compiler process will be terminated disallowing the AST to fully generate. We can compare partial ASTs; however, the problem that with two similar programs, one can pass the compiler and the other one cannot, the similarity of the comprehensive and the partial tests will be lower, even though the source codes are very alike.

The last problem is the performance problem. Our system requires graph-based data mining technique, it takes time especially when the graph is large. During the experiment, there was an AST file which contained 177 functions; it took about five hours to run SubdueGL on a Pentium 4 computer to generate a grammar file. In contrast with MOSS gives us feedback in seconds and JPlag gives us feedback in a minute.

CHAPTER 6

CONCLUSIONS

Software plagiarism is widely seen on student assignments and commercial software. Because of the complexity of source codes we would like to automate the discovery of cases of plagiarism. We use graph-based data mining to examine the source code structure–abstract syntax tree. Basically, we obtain the AST from the compiler during the compile process. Then we use a graph-based data mining tool–SubdueGL to find the significant structures in the AST. Then we use a program called Submatch to compare the grammar file that SubdueGL generated. There are some existing solutions to help people to catch it automatically, such like MOSS and YAP. They all focus on the text pattern. Thus, we would like to use the other way to find a solution.

We performed experiments on two kinds of data–artificial and real world data. The experiments of the artificial data helped us to discover the optimize option of SubdueGL and also proved that this concept is feasible. The experiments of the real-world data indicated our system can deal with real plagiarism behaviors. In some experiments, our system was more sensitive than the MOSS and the DFGGI system. And in the partial test comparison, our system provided a new method to catch the plagiarism. From the experiments we have performed, we can prove our software plagiarism detector opens another door to detect software plagiarism. Although this system is slow and depends on a compiler, I think some of these problems can be overcome in the future.

The major problem of this system is the compiler which is the source of this system.

If the source is not stable, then the rest of system cannot be performed. We will find a way to extract AST for every language in the future. We will make the AST data more reliable; but we cannot overcome the “Compiled or Not” problem.

In future work, more noise cancellation work might be considered to add to our system. Since the compiler generates AST for those specific libraries and functions and that extra AST information is for the compiler, it is not necessary to compare them. In other words, this system wants to compare the AST only as it exists in the source code. I think this idea can make the AST more reliable. It also can solve the “Compiler Problem.”

REFERENCES

[1] I. Jonyer, P. Apiratikul, and J. Thomas, “Source Code Fingerprinting Using Graph Grammar Induction,” *Proceedings of the Eighteenth Annual Florida AI Research Society*, May 2005.

[2] I. Jonyer and L. B. Holder, and D. J. Cook “MDL-Based Context-Free Graph Grammar Induction and Applications,". *International Journal of Artificial Intelligence Tools*, Volume 13, 2004.

[3] X. Chen, B. Francia, M. Li, B. Mckinnon, A. Seker, “Shared Information and Program Plagiarism Detection,” *IEEE Transactions on Information Theory*, 2004.

[4] I. Jonyer, L. Holder, and D. J. Cook, “MDL-Based Context-Free Graph Grammar Induction,” *Proceedings of the Sixteenth International Conference of the Florida AI Research Society*, 2003.

[5] I. Jonyer, L. B. Holder, and D. J. Cook, “Concept Formation Using Graph Grammars,” *Proceedings of the KDD Workshop on Multi-Relational Data Mining*, 2002.

[6] I. Jonyer, L.B. Holder, and D.J. Cook, “Graph-Based Hierarchical Conceptual Clustering,” *Journal of Machine Learning Research*, 2001.

[7] M. J. Wise, “YAP3: Improved Detection of Similarities in Computer Program and Other Texts,” *SIGCSE Bulletin*, 1996

[8] D. J. Cook and L. B. Holder, “Substructure Discovery Using Minimum Description Length and Background Knowledge,” *Journal of Artificial Intelligence Research*, Volume 1, pages 231-255, 1994.

[9] K. J. Ottenstein, “An algorithmic approach to the detection and prevention of plagiarism**,**” *SIGCSE Bulletin*, 1977.

[10] A. Aiken, Measure of Software Similarity. [Online] [Cited 16 October 2004] Available: <http://www.cs.berkeley.edu/~aiken/moss.html>

[11] G. Whale, “Identification of program similarity in large populations,” *The Computer Journal*, 1990.

[12] M. H. Halstead, “Natural laws controlling algorithm structure?,” *ACM SIGPLAN Notices*, 1972.

[13] L. Prechelt, G. Malpohl, M. Philippsen, “Finding plagiarisms among a set of programs with JPlag,” [Online] [Cited 8 March 2005] Available: < http://page.mi.fu-berlin.de/~prechelt/Biblio/jplag_jucs2001.pdf>

[14] J. L. Donaldson, A. Lancaster, P. H. Sposato, “A plagiarism detection system,” *ACM SIGSCE Bulletin, 1981.*

[15] S. Grier, “A tool that detects plagiarism in Pascal programs,” *ACM SIGSCE Bulletin, 1981.*

VITA

Hsi-Yue Sean Hsiao

Candidate for the Degree of

Master of Science

Thesis: SOFTWARE PLAGIARISM DETECTION USING ABSTRACT SYNTAX TREE AND GRAPH-BASED DATA MINING

Major Field: Computer Science

Biographical:

Personal Data: Born in Taipei, Taiwan, February 17th, 1974, the son of Hsien-Chi Hsiao and Ching-Hua Pai. He, his wife, Hui-Chen Jane Chou live in Stillwater, OK currently.

Education: Received Bachelor of Science degree in Computer Science from Oklahoma State University in December 2002. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in May 2005.

Professional Experience: Network System Administrator and Project Team Leader, Exsior Data & Information Technology Inc. 1997~1998

Name: Hsi-Yue Sean Hsiao Date of Degree: May, 2005

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: SOFTWARE PLAGIARISM DETECTION USING ABSTRACT SYNTAX TREES AND GRAPH-BASED DATA MINING

Pages in Study: 44 Candidate for the Degree of Master of Science

Major Field: Computer Science

Scope and Method of Study: This study is using a graph-based data mining technique to discover cases of software plagiarism. We hypothesize that repetitive patterns found in the abstract syntax tree (AST) representation of source code will only match such patterns of other source code if the author of both are the same. A graph-based data mining technique was used for analyzing the AST and extracting the patterns. The results from the data miner were compared using a graph matching algorithm, which provided the measure of similarity. We used artificial test sets and actual student assignments for evaluation.

Findings and Conclusions: The experiments identified plagiarism behaviors in both artificial and real-world data. These findings proved the system to be feasible. This system can be applied to every kind of programming language that use abstract syntax trees for compilation, and these ASTs can easily be extracted using the compiler. An advantage of this system over other plagiarism detectors is that it can deal with partial source code plagiarism behavior, which others do not currently do. Disadvantages of our approach include slow speed because of the graph-based data mining system used, and dependence on compilers to provide the AST. Also, if a source code cannot be compiled, the compiler will not provide a full AST, and the results will be inaccurate.

Advisor's Approval: Dr. Istvan Jonyer