Spring 2014

Analyzing Automatically Assessed Programming Assignments in CS1/2

Kiruthika Sivaraman
San Jose State University

Analyzing Automatically Assessed Programming Assignments in CS1/2

Writing Project
Presented to
The Faculty of the Department of Computer Science
San José State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by
Kiruthika Sivaraman
May 2014

The Designated Committee Approves the Project Titled

Analyzing Automatically Assessed Programming Assignments in CS1/2

by
Kiruthika Sivaraman

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE
SAN JOSÉ STATE UNIVERSITY
May 2014

Dr. Cay Horstmann — Date
Department of Computer Science

Dr. Mark Stamp — Date
Department of Computer Science

Dr. Chris Tseng — Date
Department of Computer Science

ABSTRACT

Analyzing Automatically Assessed Programming Assignments in CS1/2

by
Kiruthika Sivaraman

This project will focus on two main objectives. The first objective is to analyze Java programming solutions submitted by students and cluster or group them based on their similarities. This will help instructors in finding repeated and unique solutions.

The second objective is to analyze the assignment submission pattern of students. The metrics such as the number of times a student submits an assignment online before deadline and the amount of time a student spends on an assignment are analyzed. This information could be useful to instructors in determining the complexity of assignments and student time management.

ACKNOWLEDGEMENTS

I would like to thank my project advisor, Dr. Cay Horstmann, for his continuous support and encouragement throughout the course of this project. I would also like to thank my committee members Dr. Mark Stamp and Dr. Chris Tseng for their time and suggestions.

Table of Contents

List of Figures

List of Tables

1. Introduction

The project will focus on two main objectives. The first objective is to analyze Java programming solutions, which are submitted by students online. Based on the similarities of Java programming solutions between each other, they are clustered or grouped together. The second objective is to analyze the assignments submission metrics of students to extract some information that could be useful to instructors.

Firstly, to analyze and cluster students' solutions, the following approach is discussed in the paper. The approach is to build an abstract syntax tree (AST) for a set of Java programs that are being evaluated. All the ASTs generated above are compared with each other to find the edit-distances. The programs are then clustered based on the edit-distances between each other.

Secondly, a view of some metrics such as the number of times a student submits an assignment online and the amount of time a student spends on an assignment could be very useful information to instructors. Instructors will be able to know when students start working on a particular assignment. Instructors could relate the amount of time students spend on an assignment to the complexity of that assignment.

One challenge that I encountered in doing both of the above analyses is that in UNIX file system, the performance of search queries, which are used to retrieve students' assignments, is very slow with large amounts of data. If the data is stored in a database, the performance of queries will be much better. The performance of search queries on a sample data set has been tested in a NOSQL database (MongoDB) and a RDBMS database (MySQL). The performance gain of database-based search queries is also discussed in this paper.

1.1. Existing Techniques

Below are some existing techniques that are used to cluster programs based on their similarities and dissimilarities.

1.1.1. Bag of Words Approach

The paper titled “Modeling How Students Learn to Program” [1] explains a technique called “Bag of Words” to cluster students’ programming assignments based on their similarities. In this approach, for each of the programs, a histogram is built using the keywords used in that program. Then the “Euclidean Distance” formula is used to measure the distance between two histograms. The “Euclidean distance” between two points A and B is defined as the length of the line connecting those two points. This distance represents the dissimilarity between the programs. This is the same technique used to measure distance between texts or strings in information retrieval system.

1.1.2. Application Program Interface (API) call dissimilarity

The paper titled “Modeling How Students Learn to Program” [1] explains another technique called “Application Program Interface calls” to cluster students’ programming assignments. In this approach, each program is run with standard inputs and the resulting sequences of API calls are recorded. Then “Needleman-Wunsch global DNA alignment” technique is used to compare the sequences of API calls between programs. Two programs with same sequence of API calls are clustered together as similar programs. API calls work more efficiently for Karel programs due to the fact that there are no variables or expressions in Karel. Only predefined control structures and procedures are used in Karel. Hence the complete functionality of a program can be accurately represented by sequence of API calls. This technique does not work well for programming languages like Java. In Java, there are variables or parameters associated with each API call, making it difficult to capture an API call in a single token.

The paper [1] compares the accuracy of both the above mentioned techniques. From testing both the methods on 90 pairs of programs, it was concluded that the “API call method” has better accuracy than the “Bags of Words” approach.

1.1.3. Building Inverted Index Approach

The paper titled “PDE4Java: Plagiarism Detection Engine for Java source code” [2] explains a technique to cluster similar Java programs. In this method, the first step is to convert a Java program into a stream of tokens. A parser is required to convert the program into tokens. The process of converting a program into tokens will remove all comments and blank lines from the code. Then the stream of tokens is broken into an N-gram representation (An N-gram is a substring of size N from a given string). If a 4-gram representation is used, then each program would be a sequence of 4-gram tokens. Each new token will consist of four tokens of the original sequence. To convert the sequence of tokens into N-gram representation, the sliding window technique is used. The main advantage of this representation is that a change in code will only affect a small area of tokens. Hence reordering of code and adding few dummy lines of code will not impact the final result much. Then an inverted index is constructed for the N-gram representation. An Inverted index is the data structure used by search engines to map search-words to their locations in a set of documents. The two parts of the inverted index are a lexicon word and a series of inverted lists. Each inverted list contains a set of documents, which are mapped to a lexicon word. Then the “Jaccard coefficient” is used to calculate the distance between two programs. The formula to calculate the Jaccard coefficient is given below.

$$\texttt{Jaccard coefficient} = \frac{f_{11}}{f_{01} + f_{10} + f_{11}}$$

Formula adapted from [2]

In the above formula, f_{11} represents the total number of N-grams in both programs A and B. f_{01} represents the total number of N-gram blocks that appeared in program B and not in A. f_{10} represents the total number of N-gram blocks that appeared in program A and not in program B. This is the distance that is used to cluster similar Java programs.

2. Clustering Java Programs

The “Bag of Words” approach has been proven to be not very accurate from the experiment explained in paper [1]. The “API call” method is accurate only for Karel programs. Also it is difficult to implement this method for Java programs as mentioned in [1]. The main purpose of the last technique discussed is to detect plagiarism and not clustering programs as mentioned in [2]. Therefore in this project a different approach is used to cluster Java programs.

2.1. Clustering Approach

This project uses a method in which the Java programs that need to be clustered are converted into Abstract Syntax Trees (AST) by using the ANTLR tool (section 2.2.1). Once ASTs are generated for each program, edit-distances between ASTs are calculated. To find the edit-distance between ASTs, I evaluated Levenshtein’s Distance Algorithm and Robust Tree edit distance Algorithm (section 2.2.2). Finally, based on the edit-distances between ASTs, the Java programs are then clustered together based on their similarity scores (section 2.2.3). Figure 2.1 gives an overall picture of the approach.

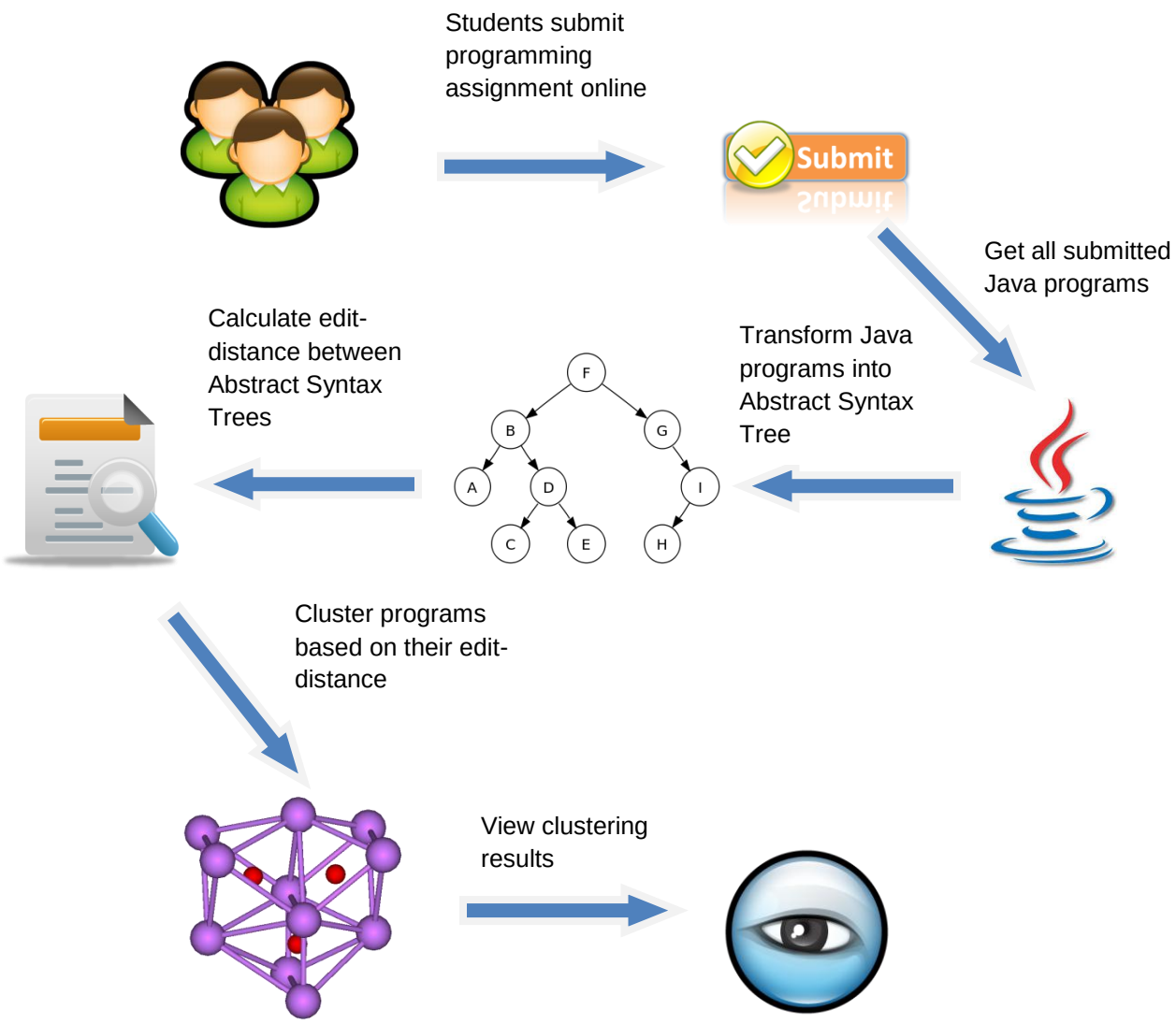


Figure 2.1- Design for Clustering Programs Based on their Similarity

2.2. Generation of Abstract Syntax Tree (AST)

The ANTLR tutorial [3] provides a description on generating abstract syntax trees (AST) for a Java program. The consolidated steps of generating AST as explained in the tutorial are as follows.

To generate AST, the first step is to get the grammar file for the programming language being used. In this case it is Java. A grammar file is a file that explains the lexical and syntactic structures of a program. The Java grammar file can be obtained from the ANTLR website [4]. The ANTLR jar file is then executed by passing the grammar file as input. This process generates the lexer and parser for Java. The final step is to write a wrapper class that takes the Java program as input and produces the AST as output.

Figure 2.2 gives a sample AST for a Java program. A sample main class to generate AST using the lexer and parser generated from grammar file is given below.

```
public class Main
{
   public static void main(String[] args) throws Exception
   {
      String filename="";
      if(args.length == 1)
      {
          filename = args[0];
          JavaLexer lexer = new JavaLexer(new ANTLRFileStream(filename));
          JavaParser parser = new JavaParser(new CommonTokenStream(lexer));
          CommonTree tree = (CommonTree)parser.JavaSource().getTree();
          DOTTreeGenerator gen = new DOTTreeGenerator();
          StringTemplate st = gen.toDOT(tree);
          System.out.println(st);
      }
      else
      {
          System.out.println("Please enter the filename to generate AST");
      }
   }
}
```

```
public class Test {

    String s;
    int j = 1 + 2;

    Test(String s)
    {
        this.s = s;
    }

}
```

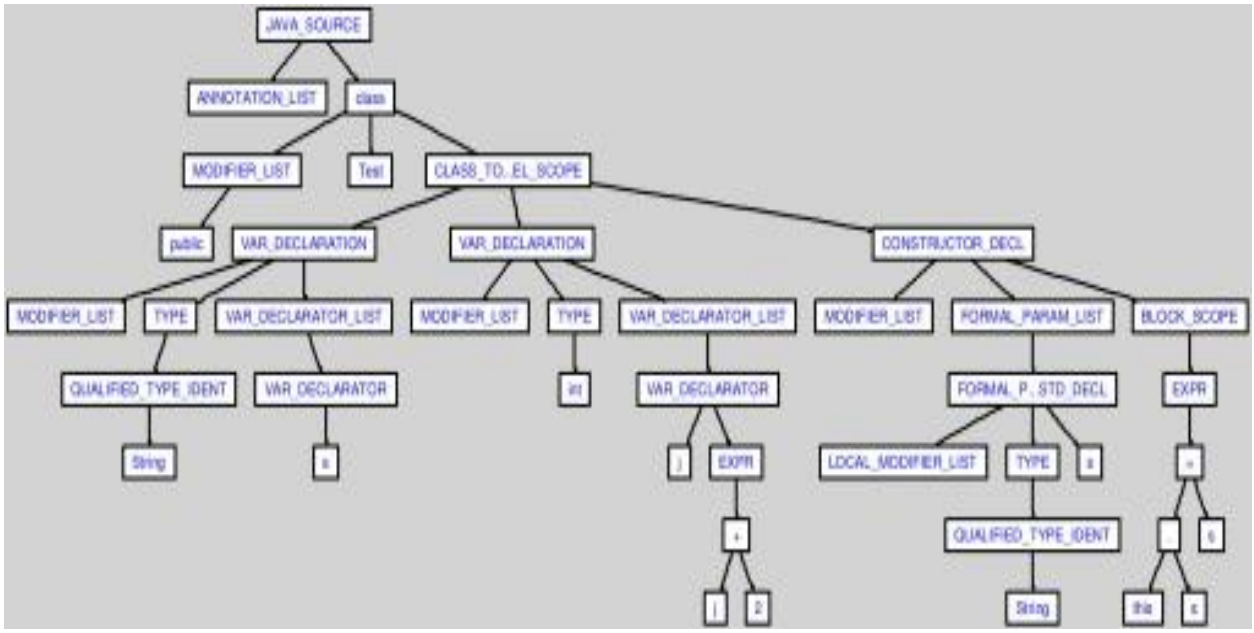


Figure 2.2 – Sample Abstract Syntax Tree for a Java program

2.3. Calculating the Edit-Distance between ASTs

In general, the edit-distance between two trees can be calculated by determining the number of edit-operations required to transform one AST tree into another. Edit-operations that can be performed on a tree are inserting a node, deleting a node and renaming or relabeling of a node. Figure 2.3 gives an example of the above mentioned tree edit operations.

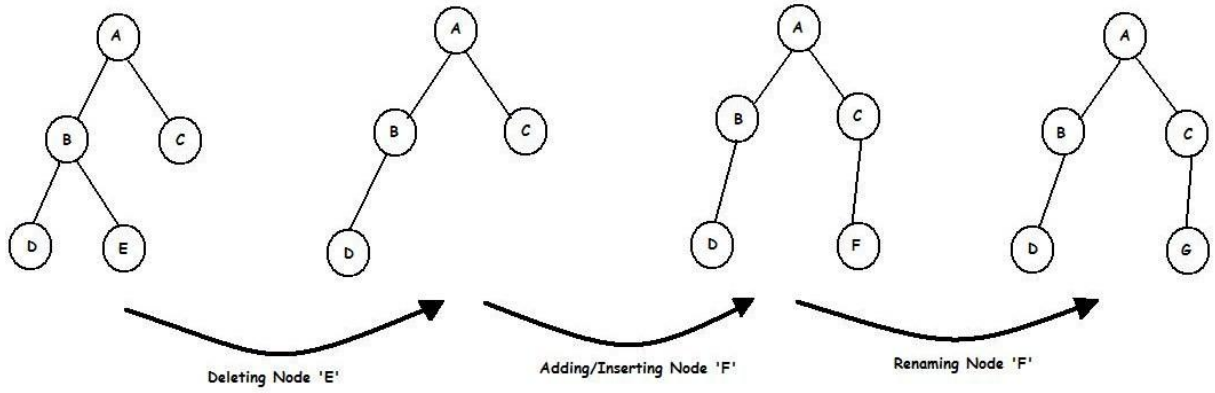


Figure 2.3 – Tree Edit-operations

There are many algorithms available to find the edit-distance between ASTs. Two widely used ones are discussed below.

2.3.1. Algorithm 1: Levenshtein Distance Algorithm

The paper titled “Learning String Edit-Distance” [5] explains this algorithm to calculate the edit-distance. Once ASTs of Java programs are generated using ANTLR, ASTs are read in post-order fashion to form a string. The Levenshtein distance formula is then used to calculate the string edit distance between two ASTs. The formula is given below.

$$= \min \begin{cases} D(i-1,j-1)+m & //substitute \\ D(i-1,j)+1 & //insert \\ D(i,j-1)+1 & //delete \end{cases}$$

Formula adapted from [6]

Suppose the two trees being compared are tree A and tree B. The values of i and j are the indexes or positions of the nodes of tree A and tree B respectively. The values of i and j are integer values ranging from zero to the maximum number of nodes in the tree. If A(i) = B(j), then m = 0. Else m = 1. D(i, j) is the distance between node i of tree A and node j of tree B.

For example, consider the trees given in Figure 2.4.

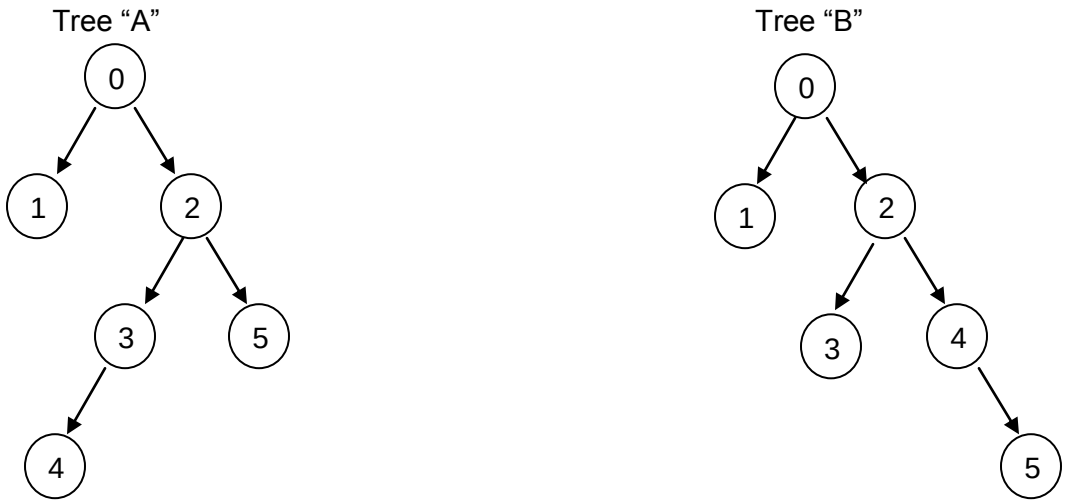


Figure 2.4 – Sample Trees to Calculate Edit-Distance

Traversing through the trees in post-order fashion, results in the following two strings.

Tree A (String)	=>	1	4	3	5	2	0
Tree B (String)	=>	1	3	5	4	2	0

The Levenshtein distance formula is then applied to strings tree A string and tree B String to calculate the edit-distance between them. The calculation is given below.

In Table 2.1, the second row represents tree A string and the second column represents tree B string. D(i, 0) and D (0, j) are initialized to same values as i and j because to convert string “1” to “”, one delete operation is needed, to convert “14” to “” two delete operations are needed and so on. The same principle is applied to the third column or tree B too.

i / j		0	1	2	3	4	5	6
			1	**4**	**3**	**5**	**2**	**0**
0		0	1	2	3	4	5	6
1	**1**	1	**?**					
2	**3**	2						
3	**5**	3						
4	**4**	4						
5	**2**	5						
6	**0**	6						

Table 2.1– Initialization of Levenshtein distance table

To calculate the value of D (1, 1), the above formula is applied. The calculation is given below.

Given i = 1 and j = 1,

(1) D (0, 0) + m = 0 + 0 = 0 (m = 0 since Tree A(1) = Tree B(1) = 1)

(2) D (0, 1) + 1 = 1 + 1 = 2

(3) D (1, 0) + 1 = 1 + 1 =2

The minimum of above three values is 0. Hence the value 0 is filled in the cell D (1, 1) as given in Table 2.2.

i / j		0	1	2	3	4	5	6
			1	**4**	**3**	**5**	**2**	**0**
0		0	1	2	3	4	5	6
1	**1**	1	**0**					
2	**3**	2						
3	**5**	3						
4	**4**	4						
5	**2**	5						
6	**0**	6						

Table 2.2 - Levenshtein distance table – step 1

The same technique is used to complete the whole table. The final cell D (6, 6) represents the edit-distance between the two string representations of the trees. The completed table is given below in Table 2.3.

i / j		0	1	2	3	4	5	6
			1	**4**	**3**	**5**	**2**	**0**
0		0	1	2	3	4	5	6
1	**1**	1	0	1	2	3	4	5
2	**3**	2	1	1	1	2	3	4
3	**5**	3	2	2	2	1	2	3
4	**4**	4	3	2	3	2	2	3
5	**2**	5	4	3	3	3	2	3
6	**0**	6	5	4	4	4	3	**2**

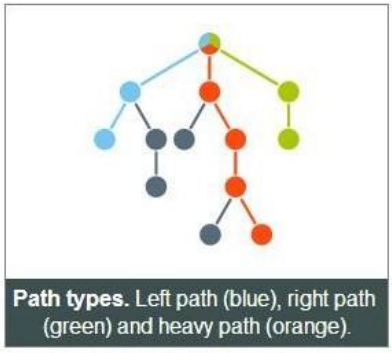
Table 2.3 - Levenshtein distance formula applied to Tree A and Tree B

2.3.2. Algorithm 2: Robust Algorithm for Tree Edit Distance (RTED)

The paper titled “RTED: A Robust Algorithm for the Tree Edit Distance” [7] explains a different algorithm to compute edit distance between two trees. The basic idea behind this algorithm is recursively decomposing the input trees into subforests by removing nodes. At each recursive call, either the leftmost or the rightmost node must be removed. The runtime complexity of this algorithm depends on the choice, made at each step. Unlike other algorithms, an optimal choice is made dynamically every time. In RTED’s algorithm, a recursive cost formula is used to find the optimal strategy. The time it takes to compute the cost of optimal strategy is very minimal compared to the overall runtime. The relevant subforests of a tree are derived by removing nodes in the below order.

- Root nodes are deleted till there are no nodes left.
- Leftmost root nodes are deleted till it is the only left leaf node.
- Rightmost root nodes are deleted till it is the only right leaf node.
- The above process is repeated recursively for the resulting sub-trees.

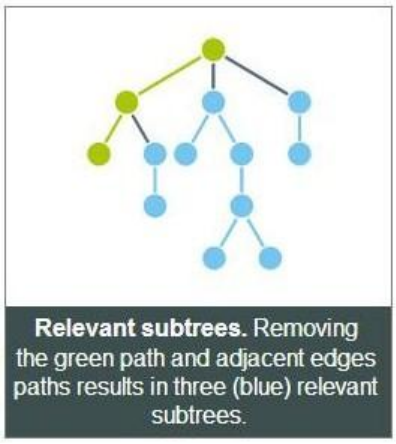
In general, this decomposition of trees into subtrees is done by choosing one of the paths – left path, right path or heavy path. The left path is the path that connects the root node of the tree with its left most child node. The right path is the path that connects the root node of the tree to its right most child node. Finally, the heavy path is the path that connects the root node of a tree to the node which forms the largest sub-tree. The chosen path is deleted from the tree to create the sub-trees. Figure 2.5 highlights the left, right and heavy paths for a tree.



Path types. Left path (blue), right path (green) and heavy path (orange).

Figure 2.5 – Left, Right and Heavy paths of a tree. Image adapted from [8]

The tree decomposition differs based on which path is chosen. From Figure 2.5, if the left path is chosen for tree decomposition, then it decomposes to three sub-trees. If the right path is chosen for decomposition, then it decomposes to two sub-trees. And if heavy path is chosen, then it decomposes to four sub-trees. Figure 2.6 shows the three sub-trees obtained if left path is selected.



Relevant subtrees. Removing the green path and adjacent edges paths results in three (blue) relevant subtrees.

Figure 2.6 – Sub-trees if left path is chosen for tree decomposition. Image adapted from [8]

Once the tree is decomposed choosing a right path, the next step is to recursively decompose the tree into sub-forests. At each step in the recursion, there are two ways in which a tree can be decomposed - either the left root node is deleted or the right root node is deleted. The following strategy is used to find which root node should be used for decomposition. Initially all tree nodes are colored gray. When root nodes of both trees are gray, then the path chosen for decomposition is colored green in the left tree. Then two solutions are obtained by deleting the root node (left solution) or by deleting all the root nodes (right solution). Then in the next step if

a forest in the left solution consists of a green root node and also if it's left most root node is gray, then the right solution is used. Otherwise, the left solution is used to decompose further. Figure 2.7 explains the choice of recursive solution for two trees.

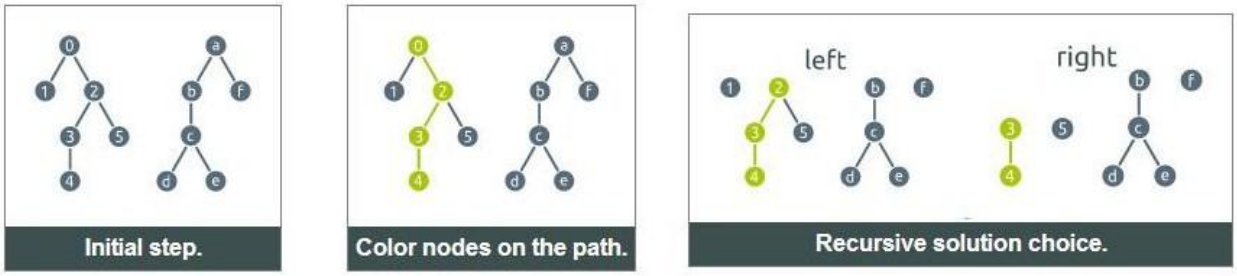


Figure 2.7 – Choosing from Recursive Solution. . Image adapted from [8]

A single-path function computes the distance between two relevant sub-trees according to the path chosen. For any given pair of trees A and B, it computes the distance between each sub-tree in tree A and each sub-tree in tree B.

General tree edit-distance algorithm is, for any two given trees A and B, the edit-distance is computed in two steps using the single-path function. The first step is to compute the distance between relevant sub-forests of tree A and all the sub-forests of tree B. These distances are then stored for later use. Then the second step is to calculate the distance between relevant sub-forests of tree A and all the sub-forests of tree B in bottom up manner. The distances that have been already computed in the first step are not computed again in step two.

The robust tree edit-distance algorithm applies an optimal strategy to the general tree edit distance algorithm. The optimal strategy is to perform an exhaustive search of all possible sub-trees considering left, right and heavy path beforehand to choose an optimal path for the tree decomposition at each step.

2.3.3. Algorithm 1 vs. Algorithm 2

For a given pair of ASTs, both Levenshtein's algorithm and RTED's algorithm produce more or less the same edit-distance. Due to the exhaustive nature of RTED's algorithm, Levenshtein's algorithm is much faster than RTED's algorithm. To compute the edit-distances of hundreds of ASTs of simple Java programs, Levenshtein's algorithm takes approximately three minutes, whereas RTED's algorithm takes around three hours. Similarly, to compute edit-distances of

hundreds of ASTs of complex Java programs, Levenshtein's algorithm takes approximately 15 minutes, whereas RTED's algorithm takes more than 6 hours. Due to this reason, this project uses a program based on Levenshtein's algorithm.

2.4. Clustering based on the Edit-Distance

2.4.1. Algorithm 1: Clustering based on Centroid

Once the edit-distances between all the ASTs are calculated, the final step is to cluster programs based on their edit-distances. The below algorithm is used for clustering.

1. Assume a random threshold value (TV).
2. Get the edit-distance between the programs to be analyzed in tabular format.
3. Add all the nodes, where each node is a program in consideration, to the unprocessed list.
4. While the unprocessed list is not empty, repeat the below steps.
    4.1. Get the centroid of all the nodes in the unprocessed list. The centroid is calculated by using the formula: min(sum (edit-distance between a node and all the other nodes in unprocessed list) / total number of nodes in the unprocessed list).
    4.2. For the centroid node, get all the nodes whose edit-distance is less than or equal to the threshold value assumed in step 1. Cluster these nodes together including the centroid node.
    4.3. Check the size of the cluster generated in the previous step. If the cluster size is equal to 1, then add the node in the cluster to outliers list, which is a list of unique programs. Otherwise, add it as a new cluster.
    4.4. Mark all nodes, which are clustered above, as processed and delete them from the unprocessed list.

When the unprocessed list is empty, it means that all the programs will be clustered. The unique programs will be in the outliers list.

Below are some examples, which explain the clustering results for programs with different level of complexities. The degree of complexity of a program is determined by the number of statements, loops and conditions used in that program.

Clustering for a simple program

The program that is considered to test simple programs is called `BankAccountTester`. The main functionality of this program is to create an object of `BankAccount`, call a method to add interest and to display the new account balance. This program is considered simple because of its functionality and, on average, the number of lines in the solutions of this problem is five. For a threshold value of 15, the method described above generates one cluster and seven outliers. Outliers are the clusters of size one or unique solutions. The clustering result and one of the unique solutions are shown in the Table 2.4.

Cluster # / Size	**Characteristic of cluster**	**Sample Program**

1 / 122	1. Constructor to initialize a BankAccount.
	
	2. A method call to add interest.
	
	3. Statement to print the balance.
		`public class BankAccountTester`
		`{`
		`  public static void main(String[] args)`
		`  {`
		`  BankAccount checking = new`
		`  BankAccount(1000);`
		`  checking.addInterest(10, 5);`
		`  System.out.println(`
		`  checking.getBalance());`
		`  System.out.println("Expected: 1500");`
		`  }`
		`}`

Outliers / 7	1. Another method `deposit()` is used instead of using a constructor to initialize the BankAccount.
	
	2. Variables are used to pass values to the method `addInterest()`.
		`public class BankAccountTester`
		`{`
		`  public static void main (String [] args)`
		`  {`
		`  int year = 10;`
		`  double percent = 5.0;`
		`  BankAccount balance = new`
		`  BankAccount();`
		`  balance.deposit (1000);`
		`  balance.addInterest (year, percent);`
		`  System.out.println(`
		`  balance.getBalance());`
		`  System.out.println ("Expected: " +`
		`  (1000 + (10 * 5 * 1000 / 100)) );`
		`  }`
		`}`

Table 2.4 – Clustering results for a simple program

Clustering of a moderately complex program

The program that is considered to test moderately complicated programs is `Mail`. The main functionality of this program is to construct a message to send an email. This method takes the name of a person as input and adds salutation to the input name. This program is considered as moderately complicated because of its functionality and, on average, the number of lines in the solutions of this problem is ten. For a threshold value of 20, the method described above generates 13 clusters and 23 outliers. Some of the clustering results are explained in Table 2.5.

Cluster # / Size	**Characteristic of cluster**	**Sample Program**

1 / 22	Instance variables are defined.	`public class Mail`
		`{`
		`  private String name;`
		`  private String assignment;`
		`  private String warning;`
		
		`  public Mail(String greeting, String body, String professor)`
		`  {  }`
		
		`  public String createMessage (String name, String assignment)`
		`  {`
		`  return "Dear ".concat(name);`
		`  }`
		`}`

2 / 47	No instance variables are defined.	`public class Mail`
		`{`
		`  public Mail (String name, String warning, String assignment)`
		`  {  }`
		
		`  public String createMessage (String name, String assignment)`
		`  {`
		`  name = "Dear ".concat(name);`
		`  return name;`
		`  }`
		`}`

3 / 9	Instance variables are defined.
	Instance variables are initialized in the constructor.
		`public class Mail`
		`{`
		`  private String professor;`
		`  private String body;`
		`  private String greeting;`
		
		`  public Mail(String greet, String bod,`
		`  String pro)`
		`  {`
		`  professor= pro;`
		`  body= bod;`
		`  greeting= greet;`
		`  }`
		
		`  public String createMessage(String`
		`  name,String assignment)`
		`  {`
		`  String Dear = "Dear ".concat (name);`
		`  return Dear;`
		`  }`
		`}`

4 / 6	Instance variables are defined. Plus sign is used for string concatenation rather than the `concat()` method as used in the other programs.	`public class Mail`
		`{`
		`  private String body;`
		`  private String goodbye;`
		`  private String name;`
		
		`  public Mail(String intro, String body,`
		`  String goodbye)`
		`  {  }`
		
		`  public String createMessage(String`
		`  name, String body)`
		`  {`
		`  String message = "Dear "+name+ ",";`
		`  return message;`
		`  }`
		`}`

Table 2.5 – Clustering results of a moderately complex program

Clustering of a very complex program

The program that is considered to test more complex programs is `RectangularGrid`. The main functionality of this program is to get and set a location, to find valid neighbors for a given location and to find occupied locations in a rectangular grid. This program is considered complex because of its functionality and also, on average, the number of lines in the solutions of this problem is more than 20. For a threshold value of 400, the method described above

generates nine cluster and six outliers. Some of the clustering results are explained in Table 2.6.

Cluster # / Size	Characteristic of cluster	Sample Program

1 / 54	Two `for` loops are used in this cluster. Location is validated by comparing their coordinates against the grid's boundaries.	`public ArrayList<Location> validNeighbors(Location loc)`
		`{`
		`ArrayList<Location> valid = new ArrayList<Location>();`
		`int row = loc.getRow();`
		`int column = loc.getColumn();`
		
		`for(int i = row - 1; i <= row + 1; i++)`
		`{`
		`for(int j = column -1;j <= column + 1; j++)`
		`{`
		`if(!(i == row && j == column) && i >= 0 && i < grid.length && j >= 0 && j < grid[0].length)`
		`{`
		`valid.add(new Location(i, j));`
		`}`
		`}`
		`}`
		`return valid;`
		`}`

2 / 32	Two `for` loops are used in this cluster. Two location objects are compared to check if the location is valid.	`public ArrayList<Location> validNeighbors(Location loc)`
		`{`
		`ArrayList<Location> neigh = new ArrayList<>();`
		`if(isValid(loc))`
		`{`
		`for (int dr = -1; dr <= 1; dr++)`
		`{`
		`for (int dc = -1; dc <= 1; dc++)`
		`{`
		`Location check = new Location(loc.getRow() + dr, loc.getColumn() + dc);`
		
		`if ( !loc.equals(check) && isValid(check))`
		`{`
		`neigh.add(check);`
		`}`
		`}`
		`}`
		`}`
		`return neigh;`
		`}`

3 / 3	All locations are added to a list without checking their validity. Then all invalid locations are removed from the list altogether.	

```
public ArrayList<Location> validNeighbors(Location
   loc)
{
   ArrayList<Location> locs=new
      ArrayList<Location>();

   locs.add(new Location(loc.getColumn()-1,
      loc.getRow()-1));

   locs.add(new Location(loc.getColumn()-1,
      loc.getRow()));

   locs.add(new Location(loc.getColumn()-1,
      loc.getRow()+1));

   locs.add(new Location(loc.getColumn(),
      loc.getRow()+1));

   locs.add(new Location(loc.getColumn(),
      loc.getRow()-1));

   locs.add(new Location(loc.getColumn()+1,
      loc.getRow()-1));

   locs.add(new Location(loc.getColumn()+1,
      loc.getRow()));

   locs.add(new Location(loc.getColumn()+1,
      loc.getRow()+1));

   for(int i=0;i<locs.size();i++)
   {
      Location fromList=locs.get(i);
      int row=fromList.getRow();
      int column=fromList.getColumn();

      if(!(row>=0 && row < rectGrid.length &&
         column >=0 && column <
            rectGrid[0].length))
      {
         locs.remove(locs.get(i));
      }

      if(fromList.getColumn()<0)
      {
         locs.remove(locs.get(i));

         if(fromList.getRow()<0)

            locs.remove(locs.get(i));
      }
   }

   return locs;
}
```

4 / 3	Many if conditions are used in this cluster.	public ArrayList<Location> validNeighbors(Location loc)
		{
		int r = loc.getRow();
		int c = loc.getColumn();
		
		ArrayList<Location> neighbors = new ArrayList<>();
		
		Location valid = new Location(r - 1, c - 1);
		Location valid2 = new Location(r - 1, c);
		Location valid3 = new Location(r, c - 1);
		Location valid4 = new Location(r - 1, c + 1);
		Location valid5 = new Location(r + 1, c - 1);
		Location valid6 = new Location(r, c + 1);
		
		if (valid != null && valid.getRow() >= 0 && valid.getColumn() >= 0)
		{
		neighbors.add(valid);
		}
		if (valid2 != null && valid2.getRow() >= 0 && valid2.getColumn() >= 0)
		{
		neighbors.add(valid2);
		}
		if (valid3 != null && valid3.getRow() >= 0 && valid3.getColumn() >= 0)
		{
		neighbors.add(valid3);
		}
		if (valid4 != null && valid4.getRow() >= 0 && valid4.getColumn() >= 0)
		{
		neighbors.add(valid4);
		}
		if (valid5 != null && valid5.getRow() >= 0 && valid5.getColumn() >= 0)
		{
		neighbors.add(valid5);
		}
		if (valid6 != null && valid6.getRow() >= 0 && valid6.getColumn() >= 0)
		{
		neighbors.add(valid6);
		}
		
		return neighbors;
		}

Table 2.6 – Clustering results for a very complex Program

Choosing the right threshold value

The threshold value must be chosen properly to get meaningful clustering results. For instance, the maximum edit-distance between two simple programs would be around 200 whereas the maximum edit-distance between two very complex programs would be around 2000. Therefore, a common threshold value will not cluster programs properly. In the former case of simple programs, the right threshold value would be around 20 whereas for the latter it would be around 500. Hence the main challenge here is to choose the right threshold value. In order to get an idea about the range of threshold values, which will provide meaningful clustering results, I developed a tool that plots a graph with the edit-distance as x-axis and the number of clusters and outliers as y-axis. Figures 2.8 and 2.9 show the graphs for a moderately complex program and complex program.

Graph of a moderately complex program

The graph in Figure 2.8 is generated for the moderately complex `Mail` program. From the graph, it can be observed that, for threshold values greater than 80, there are only 1 or 2 clusters and, for threshold values less than 20, there are a large number of outliers. Hence the right threshold value would be between 20 and 80 since for these values the number of clusters is more or less the same as the number of outliers.

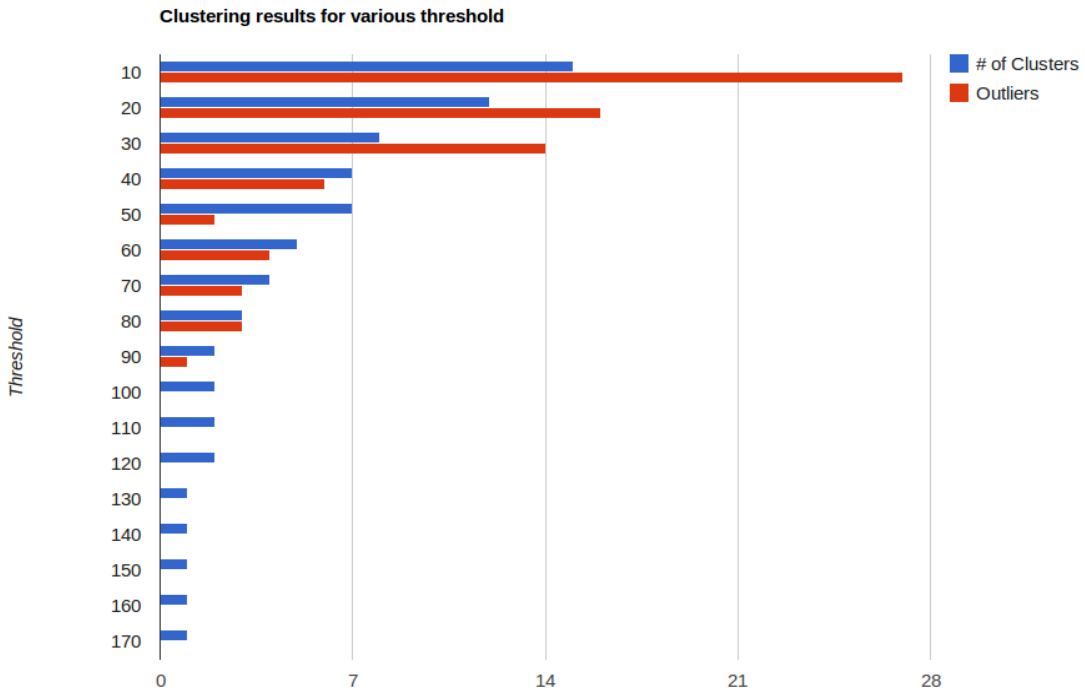


Figure 2.8 – Threshold vs. Number of Clusters and Outliers for a moderately complex program

Graph for a complex program

The graph in Figure 2.9 is generated for the `RectangularGrid` program. From the graph, it can be noted that, for threshold value greater than 700, there is only 1 cluster in the result. For threshold values less than 300, there are a huge number of outliers. Hence the right threshold value would somewhere be between 300 and 700.

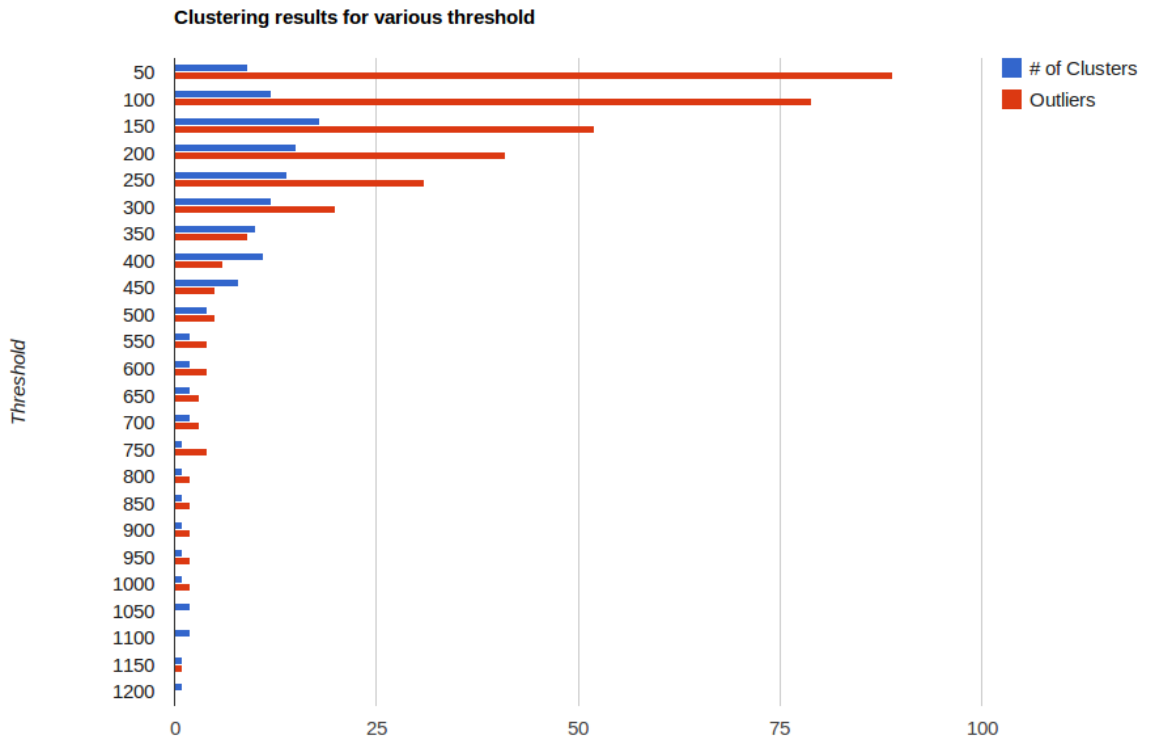


Figure 2.9 - Threshold vs. Number of Clusters and Outliers Graph for a complex program

I have developed a tool, which accepts a zip file from canvas and the path of the output directory as inputs. Given the above inputs, the tool extracts the zip file and separates the Java files from the other files. The Java files are then grouped based on their file names. A graph similar to the one shown in Figure 2.9 is plotted for an ideal range of threshold values. Then an optimal threshold value is derived from that range. The tool then clusters the input program files using the optimal threshold value. Finally, the clustering results are provided in the given output directory along with the HTML graph file. This tool also provides an option to re-run the clustering results for the separated Java files with a different threshold value. Also another option in the tool is that if an instructor knows number of unique solutions to be expected for a problem, then number of clusters can be passed as input. In that case, the tool will find the right threshold value to generate number of clusters passed in the input and will cluster based on that threshold value. Figure 2.10 shows a sample clustering tree structure for a program.

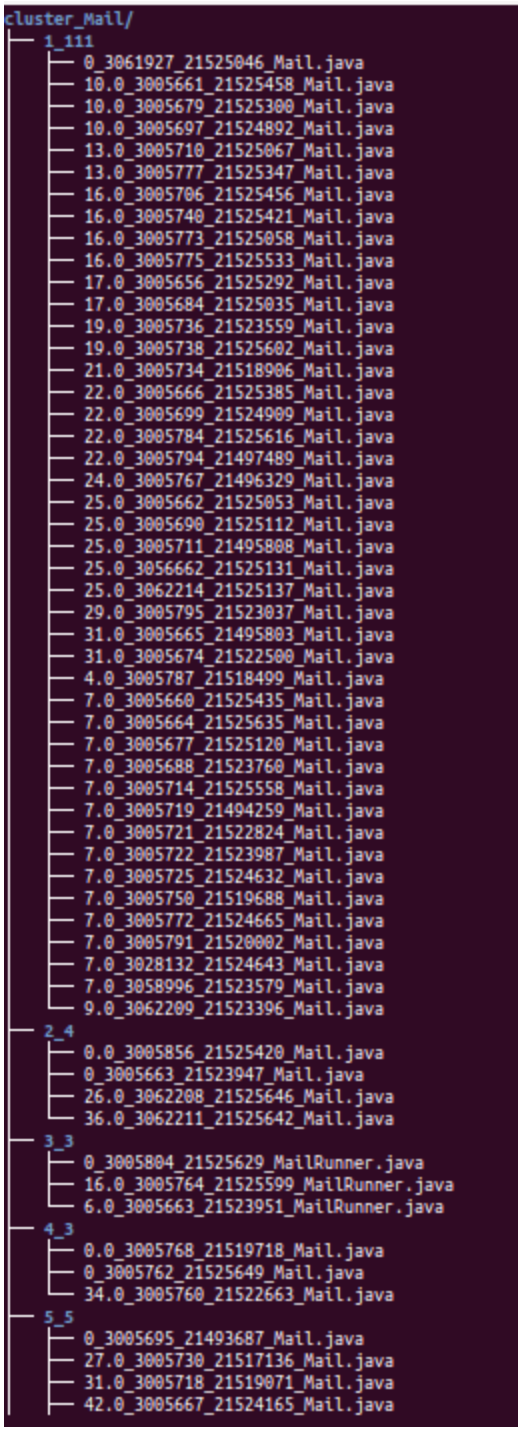


Figure 2.10 – Sample clustering tree structure

Comparison of clustering results with MOSS

MOSS [9] is a plagiarism detection engine for source code written in languages like Java, C, C++, Python etc. It detects code plagiarism by using data mining techniques. When MOSS is provided with a set of programs, it returns an output which shows the similarity scores between each other. In order to validate the clustering results of this project, the results are compared with MOSS results for a given set of programs. The program that is considered to test this case is called *Mail*. It has been observed from this comparison that the programs, for which MOSS reports more than 89% similarity, are also grouped together in the same cluster in the clustering results of this project. This verifies the method that is suggested in this paper to cluster programs together. Table 2.7 specifies similarity percentage reported by MOSS and also specifies if both the programs are grouped in the same cluster or not in the clustering results of this project.

Program Files	MOSS Results	Are the files grouped in the same cluster?
Student1_Mail.Java Student2_Mail.Java	81 % 80 %	No
Student3_Mail.Java Student4_Mail.Java	94 % 94 %	Yes
Student5_Mail.Java Student6_Mail.Java	97 % 97 %	Yes
Student7_Mail.Java Student8_Mail.Java	90 % 90 %	Yes
Student9_Mail.Java Student10_Mail.Java	89 % 81 %	No
Student11_Mail.Java Student12_Mail.Java	89 % 81 %	No

Table 2.7 – Comparison of MOSS results with clustering results

2.4.2. Algorithm 2: K-Mean Algorithm for Clustering

K-Mean is a flat clustering algorithm [10], which clusters n documents into k clusters. Given the number of clusters to start with, each document is added to its nearest cluster based on its mean edit-distance. The steps of K-Mean algorithm are given below.

1. Get the number of clusters k, as input.
2. For i = 1 to k do,
    3.1 Pick a random document n (initial seed), and add it to cluster i.
    3.2 Set document n as processed.
3. While the centroids of clusters are different from their values in the previous iteration
    3.1 For i = 1 to total number of documents
        3.1.1. Calculate mean edit distance of document i with the documents in each cluster.
        3.1.2. Add document i to the nearest cluster based on the mean calculated in step 3.1.1.
        3.1.3. Re-compute the centroid of that cluster.

Choosing the right initial seeds

The silhouette coefficient is used to validate the clustering [11]. The idea behind this is to calculate the mean of the edit-distance between document di in cluster ci and all other documents in the same cluster. Let the value computed be a. Then the mean of edit-distances between document di and all other documents in cluster cj is calculated, where ci ≠ cj. Let b be the shortest mean of all the means between di and cj. Then the silhouette coefficient is calculated using the formula given below.

$$s = 1 - a/b \quad \text{if } a < b$$

Formula adapted from [11].

Usually a is less than b, as the mean of the edit distances between document di and documents in its same cluster ci will be less than the mean of edit-distance between document di and all the documents in other cluster cj. The closer the value of s is to 1, the better is the clustering.

The value of s is not always 1 when all the initial seeds are selected randomly. Also it takes more iteration to converge. On the other hand, if the first seed is selected randomly and the farthest document from the first seed is selected as the second seed, and the farthest document from the second seed is selected as the third seed and so on, then it is found that it converges faster and also the value of s is always 1.

2.4.3. Algorithm 1 vs. Algorithm 2

For K-Mean algorithm, clustering results depend on the number of initial clusters provided as input. The drawback here is that one may not know how many unique solutions exist among the programs submitted, at the time of input. On the other hand, in centroid-based clustering approach suggested in this paper, without having to provide any input, clustering results are generated for an automatically estimated threshold value, along with a graph of threshold values vs. number of clusters and outliers. Based on this graphical analysis, one can easily know the optimal range of threshold values for which clustering results stay almost the same. Overall, apart from the above mentioned points, it was observed for the same set of Java programming files that the clustering results are more or less the same for both the above algorithms.

3. Data Mining of Automatically Assessed programs

There is around 15 GB of data that came from automatically assessed Java programming assignments submitted by the students of SJSU in the past. The old data as such is incomplete as there was no information to track or connect each program to a student. Since January 2014, cookie information has been embedded into each file that students submit. The following analyses have been done to extract useful information out of this data.

3.1. Tracking the number of submissions by a student per assignment

For each assignment, there are more than 1000 submissions from students. Initially it was difficult to track the number of submissions per student as all the submissions were anonymous. Not all the students used to add proper author tags in their programs. Out of 1400 submissions, it was found that only 300 of them had author tags. 150 of that 300 were empty author tags. Hence cookies were added to each student's programming assignment file to track the number of submissions each student makes per assignment. I found that for an assignment some students make more than 100 submissions, whereas some make less than 10 submissions. An excerpt from the output of the analysis of submissions made for an assignment is shown below. The first column represents the number of submissions and the second column is the cookie information. From the graph in Figure 3.1, it can be observed that one student has made more than 300 submissions whereas some have made less than 10 submissions.

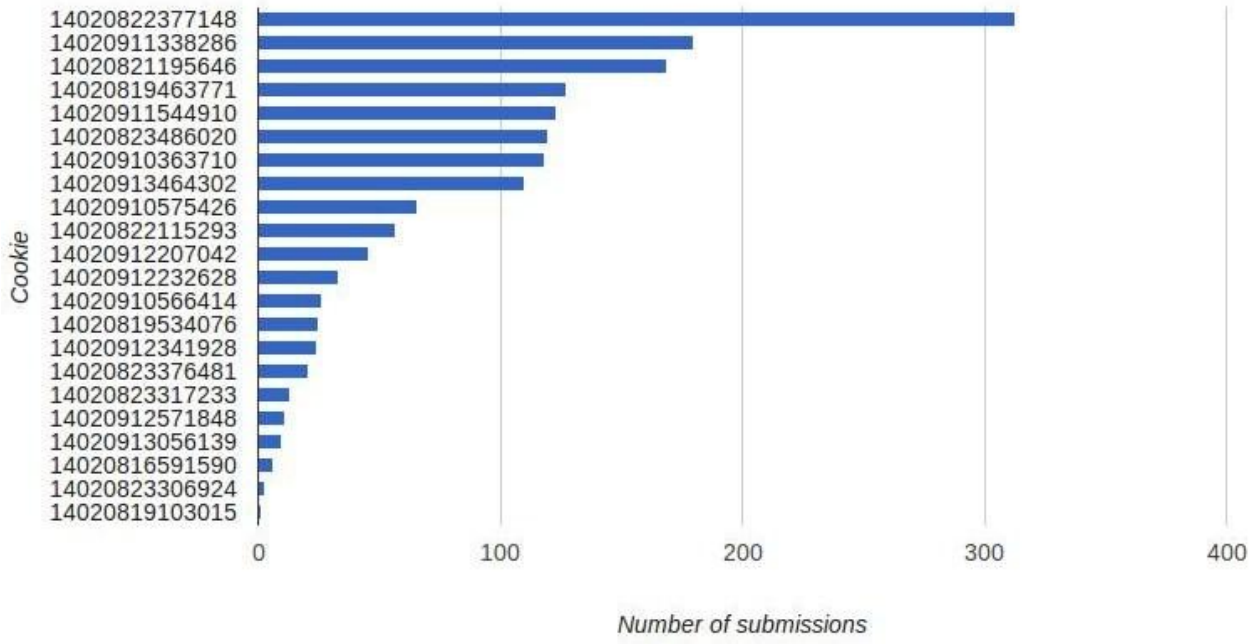


Figure 3.1 – Graph for tracking number of submissions per cookie

3.2. Analyze when students start and finish assignments

Since cookie information is now embedded in assignment files, it is easy to track when the first submission was made by a cookie for an assignment and when the last submission was made by the cookie for the same assignment. These metrics were gathered for each cookie or student for a simple programming assignment and a complex programming assignment, with respect to draft submission deadline and final submission deadline. Basically students are supposed to make a little progress or write only the code necessary to solve a subset of the main problem by draft due date. By the final due date, students are supposed to submit a version of the program that solves the entire problem. The analysis of these metrics is presented below.

3.2.1. Submission Analysis for a simple program

The program considered for this test is called `DoubleInvestment`. The function of the program is to keep prompting the user to enter an interest value 20 times for next 20 years. The interest will be added to the user's account balance each time. The program will stop if the balance is doubled from the initial value at any point.

The draft submission was due on March 12, 2014 and the final submission was due on March 16, 2014. The metrics discussed in previous section were collected for this assignment. It was first sorted based first submission dates. A graph is plotted for the same. From the graph in Figure 3.2, I observed that most of the students made their first submission for this assignment on March 11, 2014, approximately, a day before the draft submission.

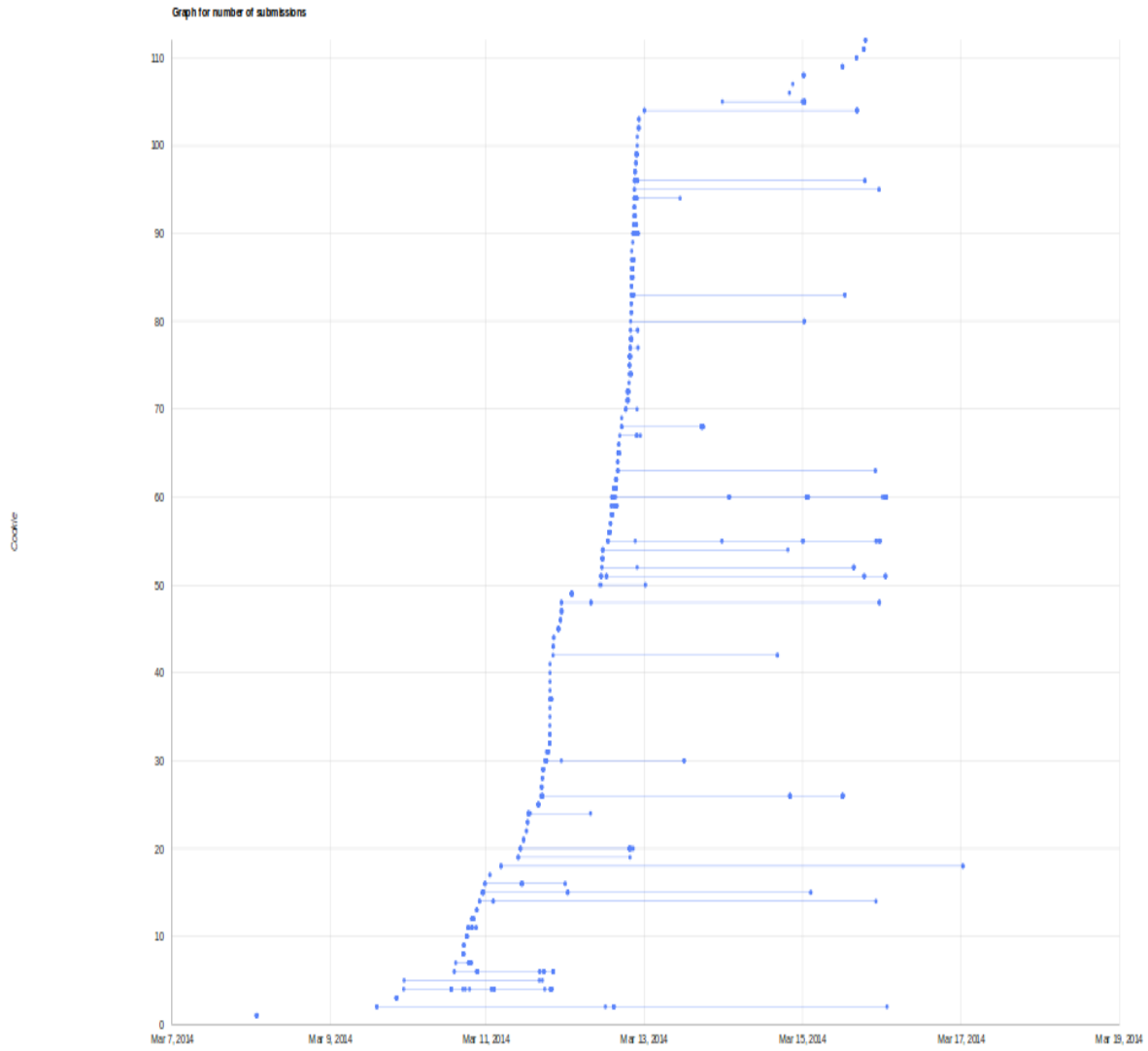


Figure 3.2– Graph for DoubleInvestment program Sorted based on Start Date

Then the data gathered was sorted based on when the last submissions were made for the assignment. A graph was plotted for the same. From the graph in Figure 3.3, I observed that most of the students made their final submission at the draft itself. Only very few students did make submissions after the draft. It means that, given a simple assignment, most students tend to finish it by the draft due date.

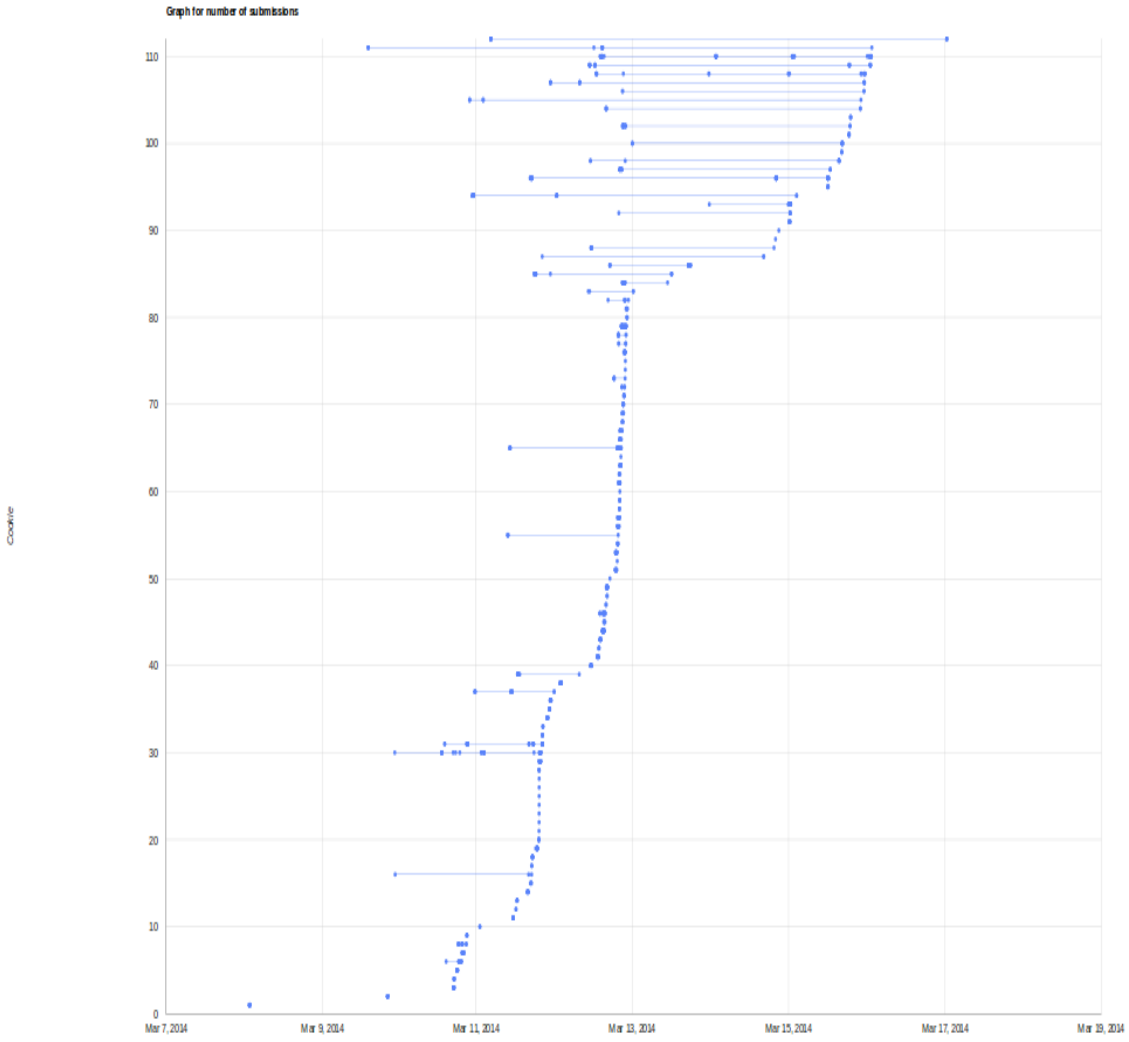


Figure 3.3 – Graph for DoubleInvestment program Sorted based on End Date

3.2.2. Submission Analysis for a complex program

The complex program that was considered for this analysis is called `UniqueCharacters`. The program takes a string as input. Then it finds all the letters that occurs only once in the string. For this assignment, the draft submission was due on March 19, 2014 and the final submission was due on March 21, 2014.

The same metrics discussed above were collected for this assignment too. Then it was first sorted based on the first submission dates. From the graph in Figure 3.4, I observed that half of the students made their first submission of this assignment on March 18th or before, whereas the other half did that only on March 19th.

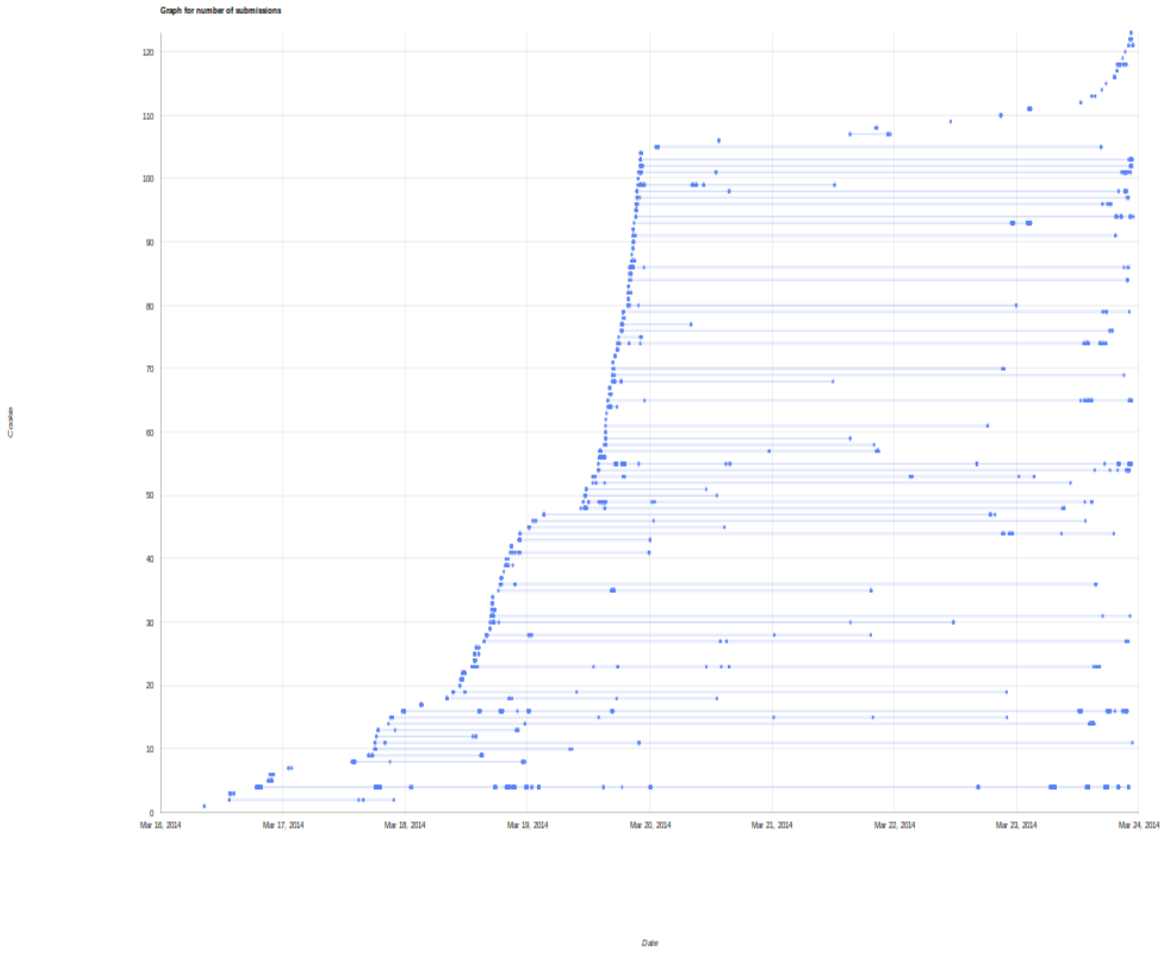


Figure 3.4 - Graph for UniqueCharacters program Sorted based on Start Date

The metrics gathered was then sorted based on the last submission dates for the same assignment. From the graph in Figure 3.4, it was observed that only few students turned in their final version of the programs by draft submission date, whereas most of the students were able to finish only by the final submission date.

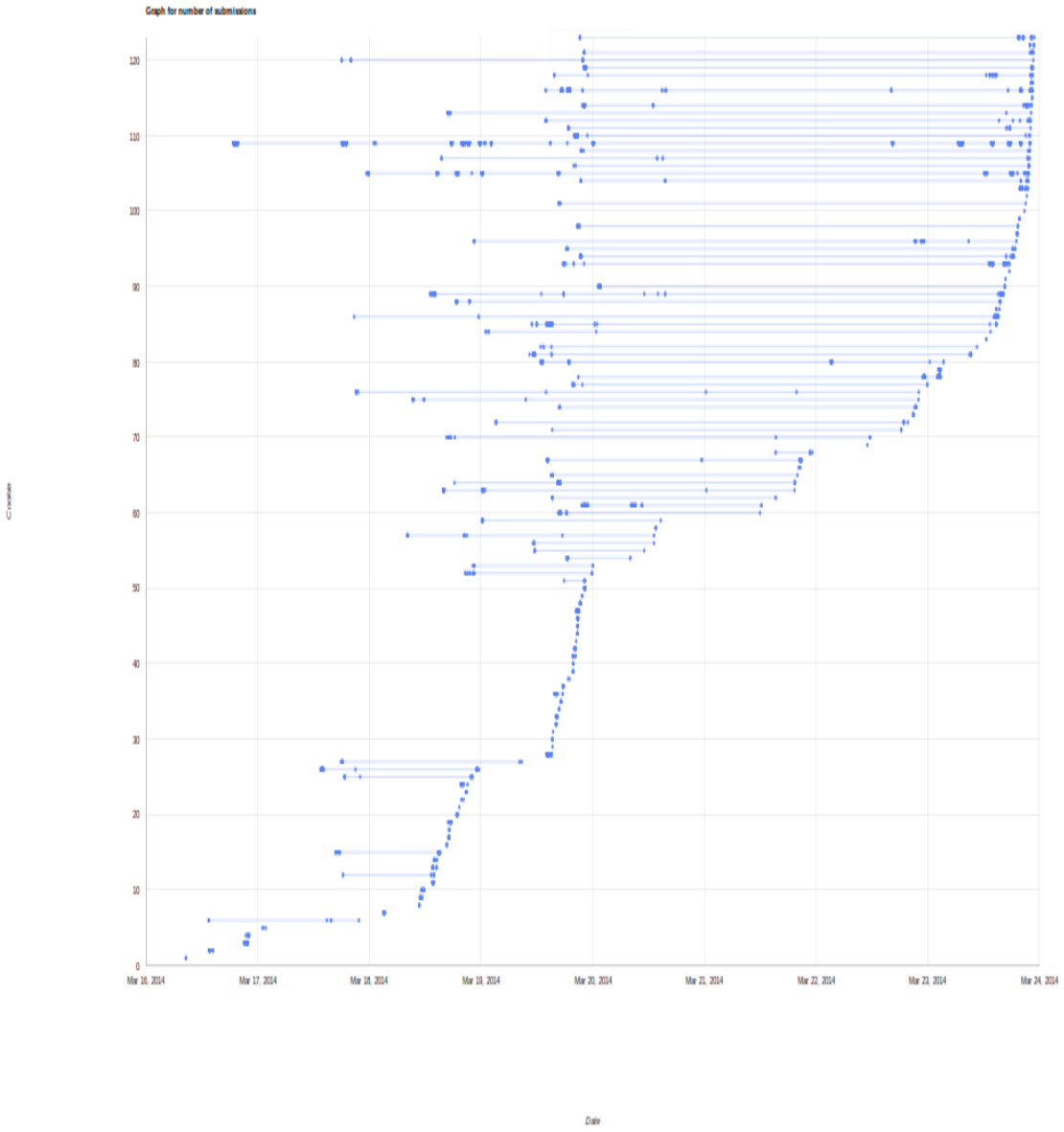


Figure 3.5 - Graph for UniqueCharacters program Sorted based on End Date

From this analysis, it was understood that, when the assignments are complicated, students submit the final versions of their assignments only by the final submission due date. Otherwise most of the students complete their final version of the assignment by the draft submission due date. Also it was noted that in both of the cases students make first submission of assignments only on the day before the draft due date. While graphing data based on first submission dates does not provide much insight, graphing data based on last submission dates would help reassure instructors about the complexity level of assignments.

4. Performance of Search Queries in RDBMS and NoSQL databases

When I started this project, I received a directory with approximately 1 million submissions and approximately 10GB of total data. Searching for the simplest facts took several hours. Therefore, I decided to test search performance with the MySQL (RDBMS) database and the MongoDB (NoSQL) database. I also integrated elasticsearch, a popular search engine, with MongoDB to check whether the performance could be improved any further. The metrics from all the above cases were compared against UNIX file system-based searches. The results of this comparison are reported in this section. The performance tests were conducted on a machine with 6 GB memory and Intel Quad core i5 CPU running at 1.70 GHz. The machine was running the Ubuntu operating system. The implementation of the above scenarios on Ubuntu OS is explained in this section in detail.

4.1. MongoDB with Elasticsearch

MongoDB is a NoSQL database. Elasticsearch is an open-source search engine that is based on Java and the Lucene search server. Elasticsearch can be used to search for data that is stored in the Mongo database. In this case, the student submissions were loaded to the database using “mongofiles” utility of MongoDB. Once the above steps are done, one can perform searches on Elasticsearch using various keywords including file names, regular expression of a string and so on.

MongoDB was first installed on the machine described above. Installation is clearly explained in the tutorial “Install MongoDB on Ubuntu” [12]. The MongoDB version used for this test is 2.4.5. Once installation is successful, a mongod instance is started with a replica set name by executing the below command from a terminal. Replica sets are read-only copies of the primary Mongo database, for which the data is kept in synchronization with primary database in real time as per your requirements.

Syntax:

```
sudo /usr/bin/mongod --replSet <replica-set-name> --port <portnum> -
fork --quiet --dbpath <path-to-data-dir> --logpath <path-to-log-file>
```

Example:

```
sudo /usr/bin/mongod --replSet FSsearch --port 27017 -fork --quiet --
dbpath /data/mongodata/r0 --logpath /data/log/mongodb/mongodb0.log
```

The above Mongo instance will be the primary database. To add a secondary database or a backup database, another Mongo instance needs to be started on a different port, with different data directory but with the same replica set name.

Example:

```
sudo /usr/bin/mongod --replSet FSsearch --port 27018 -fork --quiet --
dbpath /data/mongodata/r1 --logpath /data/log/mongodb/mongodb1.log
```

Once both the Mongo instances are started successfully, login into any of the Mongo instance using “mongo <server-name>:<port>” command from the terminal. Then from the mongo prompt to execute the below command to start the replica set.

```
> rsconf = {
   _id: "FSsearch",
   members: [
   {
      _id: 0,
      host: "localhost:27017"
   }
   ]
}

> rs.initiate( rsconf )
```

The above command specifies that the Mongo instance running on port 27017 will act as a primary database and configures or initiates the replica set. To check if the initiation is successful, “rs.status()” command can be used. Once the initiation of the primary database in the replica set is done, the secondary database can be added by using the below command.

Syntax:

```
rs.add("<server-name>:<port>")
```

Example:

```
rs.add("localhost:27018")
```

After the Mongo database configuration is done, Elasticsearch needs to be installed. The article titled “Setting up ElasticSearch with MongoDB” [13] explains how to install and integrate Elasticsearch with MongoDB. Elasticsearch can be downloaded from [14]. The Elasticsearch version used for this testing is 0.90.2. Also some plugins need to be installed for Elasticsearch to enable us to have a web front-end to interact with Elasticsearch and to enable indexing for document formats like Text, Word Documents, HTML files etc. The downloaded tar file was first extracted and placed under /usr/local/share directory.

Elasticsearch’s plugin “head” was installed by executing the below command from a terminal.

```
sudo /usr/local/share/elasticsearch/bin/plugin -install
mobz/elasticsearch-head
```

Then the plugin elasticsearch-mapper-attachments was installed by executing the below command from the terminal. The version installed is 1.7.0.

```
sudo /usr/local/share/elasticsearch/bin/plugin -install
elasticsearch/elasticsearch-mapper-attachments/1.7.0
```

The next plugin that is required is elasticsearch-river-mongodb. This plugin is used to index the Java programming files that are stored in MongoDB. The plugin version installed for this test was 1.6.11. This version can be downloaded from the Maven repository [15]. Also as Elasticsearch is implemented in Java, a mongo-Java-driver is also needed to read and index the files from MongoDB. The version of jar used for this purpose was 2.11.2. This version also can be downloaded from maven repository [16]. Once both the jars are downloaded, they were moved to the path `$ES_HOME/plugins/river-mongodb`, where `$ES_HOME` is set to the home directory of Elasticsearch where it was installed.

After installing all the plugins, Elasticsearch can be started by executing the below command from a terminal.

```
sudo service elasticsearch start
```

To check if Elasticsearch has been started properly, the below query can be used.

```
curl http://localhost:9200
```

The above query will return “OK” only if Elasticsearch has been started properly. The next step is to create an index in Elasticsearch and map it to a collection in MongoDB, from where the files will be retrieved. The below command needs to be executed to create the index..

```
curl -XPUT "localhost:9200/_river/subfs/_meta" -d'
{
 "type": "mongodb",
   "mongodb": {
      "db": "submission",
      "collection": "fs",
      "gridfs": true
   },
   "index": {
      "name": "sub_idx",
      "type": "files"
   }
}'
```

The above query returns “OK” if the index is created properly. Otherwise, an error message is returned. The name of the index created is “sub_idx”. This index is mapped to the collection “fs” of the “submission” database in MongoDB. When programming files are loaded into the database, MongoDB creates a default collection called “fs” to store those files. Also the parameter gridfs is set to true, because MongoDB uses grid file system to store the files.

After setting up the Elasticsearch index, files can be loaded into MongoDB using the “mongofiles” utility. It is done by executing the below command from a terminal.

```
/usr/bin/mongofiles --host localhost:27017 --db submission --
collection fs --type text/plain put ‘Tester.Java’
```

Once the files are uploaded to MongoDB, it will be automatically indexed by Elasticsearch since an index already exists, which is mapped to the “submission” database. To check if the files are indexed by Elasticsearch, execute the below command from a terminal.

```
curl -XGET "localhost:9200/sub_idx/files/_count?pretty"
```

This query gives the count of the documents indexed by Elasticsearch. This count will be equal to the number of documents which were inserted into MongoDB. Once all the documents are

inserted into MongoDB and Elasticsearch indexing is done, the below query can be used to retrieve the files for which the word “Mail” is a part of the name.

```
curl -XGET
"localhost:9200/sub_idx/files/_search?fields=filename&size=10000&prett
y=1" -d '{"query" : {
"field" : {
     "filename" : "*Mail*"
     }
}
}'
```

MongoDB automatically creates a unique ID for every file inserted into MongoDB. In order to query a file based on this unique ID, the below query can be executed from a command line window.

```
curl -XGET
"localhost:9200/sub_idx/files/521ac6932ac32d0505640c1a?pretty=true"
```

In the above query, “521ac6932ac32d0505640c1a” is the unique id of the file. In order to query the files, which contain the word “Mail” anywhere inside the files or their file names, then the below query can be used.

```
curl -XGET "localhost:9200/sub_idx/files/_search?q=Mail&pretty=true"
```

In order to delete an existing Elasticsearch index, the below command can be used.

```
curl -XDELETE 'http://localhost:9200/sub_idx'
```

The above command will return “true” if the index is deleted successfully. To list all the indexes that are created in Elasticsearch, the below command can be used.

```
curl -XGET 'http://localhost:9200/_aliases'
```

The most commonly used queries to create/delete/query an index of Elasticsearch are shown in this section. Also, the versions of plug-ins and software installed above are important and need to be followed as mentioned above. Any deviation in software or plugin versions could result in MongoDB and Elasticsearch integration issues.

4.1.1. Indexing large data sets using Elasticsearch

When a large number of documents have to be indexed using Elasticsearch it is noticed that there may be some glitches. Sometimes Elasticsearch might error out after indexing about 150 thousand records as it may not able to scale to insert a huge set of data at a time. Therefore create an Elasticsearch index first and then insert files into MongoDB to avoid this problem. The files will get indexed simultaneously while files are being inserted into MongoDB.

4.1.2. Changing Heap Size of Elasticsearch

To be able to retrieve a huge result set, the heap size of Elasticsearch must to be changed. The steps to change heap size are given below.

First, Elasticsearch must be stopped using this command.

```
sudo service elasticsearch stop
```

Once the service is stopped, the parameter `ES_HEAP_SIZE` value needs to be changed to the new heap size value and then Elasticsearch is restarted. To change other configuration parameters, refer to the tutorial “Elasticsearch - Running as a service on linux” [17].

4.2. Stand-alone MongoDB

MongoDB installation is straightforward. The installation is clearly explained in the tutorial “Install MongoDB on Ubuntu” [12]. Once the MongoDB is installed successfully, the database instance can be started using the below command.

Syntax:

```
sudo /usr/bin/mongod --port <portnum> -fork --quiet --dbpath <path-to-data-dir> --logpath <path-to-log-file>
```

Example:

```
sudo /usr/bin/mongod --port 27017 -fork --quiet --dbpath /data/mongodata/r0 --logpath /data/log/mongodb/mongodb0.log
```

Once the mongo instance is started, the below code can be used to insert student submission files into MongoDB. The mongofiles utility does not provide a way to search files based on its contents. Hence, the files are inserted as strings into MongoDB. Each record has three columns – a filename, last modified date of the file and the content of the file. Class declaration and variable declarations are not provided here.

```
MongoConnect mc = new MongoConnect();

String contents = mc.readFile(filename, StandardCharsets.UTF_8);

MongoClient mongoClient = new MongoClient( "localhost" , 27017 );
DB db = mongoClient.getDB(dbStr);

DBCollection coll = db.getCollection(collStr);

mongoClient.setWriteConcern(WriteConcern.NORMAL);

BasicDBObject doc = new BasicDBObject("filename", filename).
   append("date", date).append("contents", contents);

coll.insert(doc);
```

A mongo-Java-driver jar is required to insert the file contents into MongoDB using the above Java program. The version of jar used is 2.11.2. This version also can be downloaded from the Maven repository [16].

4.3. MySQL

The query performance was tested in MySQL to check if performance differs between RDBMS and noSQL databases. MySQL was installed by following the steps provided in “Ubuntu Documentation - MySQL” [18]. The jar “mysql.jar”, which is needed to connect to MySQL using a Java program, is provided along with the installation. The code below is used to insert the files into MySQL. The class declaration and variable declarations are not provided here.

```
Class.forName(dbClassName);

Properties p = new Properties();

p.put("user","root");
p.put("password","");

Connection conn = DriverManager.getConnection(CONNECTION,p);
Statement stmt = conn.createStatement();

String sql = "insert into ALL_SUBMISSIONS values(" + id + ", '" + fname
   + "', '" + content + "', '" + date + "')";

stmt.executeUpdate(sql);
conn.close();
```

4.4. Comparison of performance of queries

Queries were executed to retrieve 1000, 10000 and 100000 records from MySQL, Unix file system, MongoDB and MongoDB with Elasticsearch. The response times of the queries were recorded which is shown in Table 4.1 and a graph is plotted for the same as shown in Figure 4.1.

No. of records Retrieved	Time taken by the Query to execute			
	UNIX File System	**MongoDB with Elasticsearch**	**MongoDB**	**MySQL**
1000	15 mins	10 secs	18 secs	20 secs
10,000	29 mins	18 secs	29 secs	27 secs
100,000	Query timed out	22 secs	57 secs	51 secs

Table 4.1 – Comparison of performance of queries among MongoDB, MySQL, MongoDB with Elasticsearch and UNIX file system

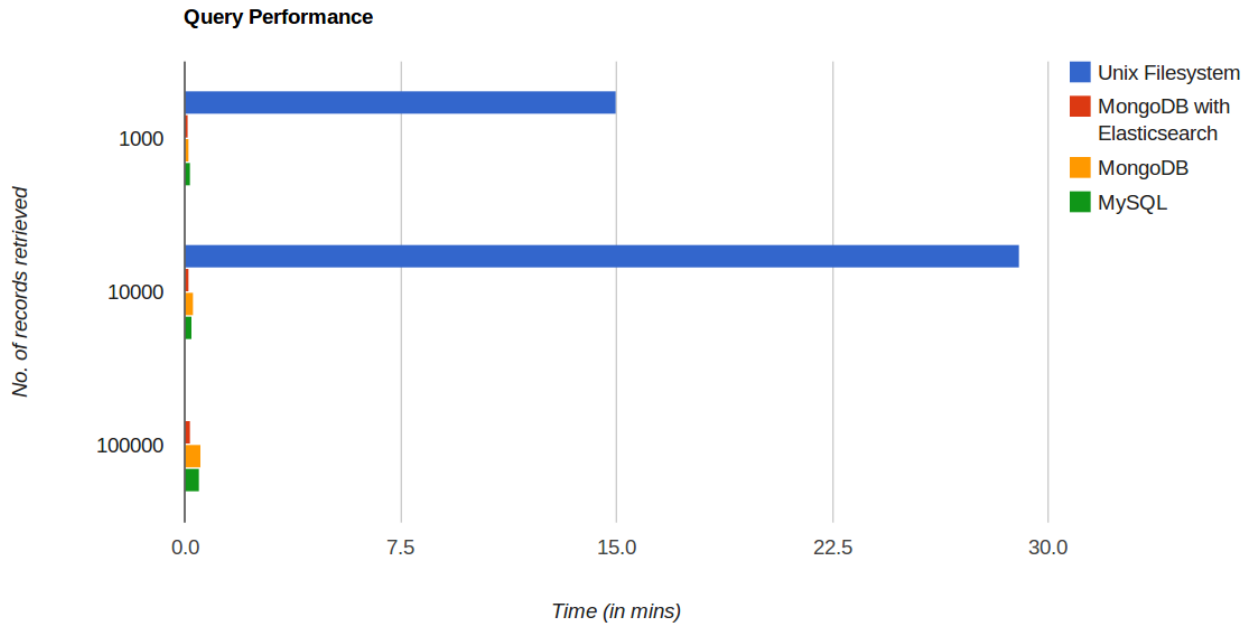


Figure 4.1 - Comparison of Query Performance in MongoDB, MySQL, MongoDB with Elasticsearch and UNIX file system

In the performance test results presented above, it clearly shows that Elasticsearch and MongoDB combination provides better performance than the other options which were considered. But the major challenge with Elasticsearch is that it needs at least 4GB heap space to display 9000 records in the output. If the heap space is not sufficient, then Elasticsearch is likely to crash with OutOfMemory error. The heap space must be set to an appropriate value with respect to the size of the result set of the queries. If there is a need to run queries that return a few thousand or more records, and if memory is limited, then Elasticsearch with MongoDB is not the best choice. For search and data manipulation queries, MongoDB by itself provides more or less the same performance as the combination of MongoDB and Elasticsearch. Since data can be easily loaded to the database as JSON objects, irrespective of the nature of the data, it is a better choice than MySQL. In MySQL, due to the existence of multiple special characters, such as single quotes, semi colons etc., in programming files and the data needs to be inserted as a string, the formation of insert commands with correct syntaxes was quite challenging. UNIX file systems exhibit poor performance as expected. Therefore, MongoDB by itself is the best option as per the current requirements.

5. Conclusion

I have been able to achieve all the objectives - clustering unique solutions, finding useful information, analyzing and improving the performance of the existing system - of this project successfully. Firstly, I utilized ANTLR to convert automatically assessed Java programming solutions or files to Abstract Syntax Trees. Secondly, I implemented Levenshteins algorithm in order to find edit-distances between the files, which also provided a simple similarity comparison metric. Thirdly, I was able to create simple Java programs to cluster unique solutions, using a Centroid-based approach and K-Mean algorithm. From the above implementations, I observed that the clustering results are more or less the same for both of the methods. The main challenge I encountered while using Centroid-based approach was choosing the right threshold value. On the other hand, for K-Mean algorithm, I had to choose the right initial seeds and the right number of clusters, which was more challenging. When I selected these values correctly, both approaches generated meaningful clustering results. Finally, I validated the clustering results from both of these methods against the results provided by MOSS website for a subset of Java programming files from our system.

I plotted simple graphs of the last submission timestamps of students, using which I was able to validate the complexity of the assignments just by seeing when students submit assignments. I analyzed a few RDBMS and NoSQL solutions to find a feasible solution to improve performance of the current system. From testing the performance of each of these solutions, I found that MongoDB alone provides a much better performance in comparison with current UNIX-based file system.

The current scope of this project is clustering Java-based programming assignments. The same approach that was followed in this project can be applied for programming assignments based on other languages like C, C++, and Python.

References

[1] C. Piech, M. Sahami, D. Koller, S. Cooper and P. Blikstein, "Modeling How Students Learn to Program," *Proceedings of the 43rd ACM technical symposium on Computer Science Education,* pp. 153-160, 29 2 2012.

[2] A. Jadalla and A. Elnagar, "PDE4Java: Plagiarism Detection Engine for Java source code: a clustering approach," *International Journal of Business Intelligence and Data Mining,* vol. 3, no. 2, pp. 121-135, 2008.

[3] T. Parr, "ANTLR 4 Documentation," [Online]. Available: https://theantlrguy.atlassian.net/wiki/display/ANTLR4/ANTLR+4+Documentation. [Accessed 15 September 2013].

[4] "Directory Listing for Grammar," 2012. [Online]. Available: http://www.antlr3.org/grammar/1207932239307/Java1_5Grammars/. [Accessed 10 October 2013].

[5] E. S. Ristad and P. N. Yianilos, "Learning string-edit distance," *Pattern Analysis and Machine Intelligence, IEEE Transactions,* vol. 20, no. 5, pp. 522 - 532, 1998.

[6] A. McCallum, "String Edit Distance," [Online]. Available: https://people.cs.umass.edu/~mccallum/courses/inlp2007/lect4-stredit.ppt.pdf. [Accessed 15 March 2014].

[7] M. Pawlik and N. Augsten, "RTED: A Robust Algorithm for the Tree Edit Distance," *Proceedings of the VLDB Endowment,* vol. 5, no. 4, pp. 334-345, December 2011.

[8] M. Pawlik and N. Augsten, "Introduction to the Tree Edit Distance," [Online]. Available: http://www.inf.unibz.it/dis/projects/tree-edit-distance/tree-edit-distance.php. [Accessed 22 November 2013].

[9] A. Aiken, "MOSS - A System for Detecting Software Plagiarism," [Online]. Available: http://theory.stanford.edu/~aiken/moss/. [Accessed 14 February 2014].

[10] "K-Means," Cambridge University Press, [Online]. Available: http://nlp.stanford.edu/IR-book/html/htmledition/k-means-1.html. [Accessed 23 April 2014].

[11] "Clustering," [Online]. Available: cs-people.bu.edu/evimaria/cs565/lect8.ppt. [Accessed 25 April 2014].

[12] "Install MongoDB on Ubuntu," MongoDB, Inc, [Online]. Available:

http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/. [Accessed 1 November 2013].

[13] J. West, "Setting up ElasticSearch with MongoDB," [Online]. Available: https://coderwall.com/p/sy1qcw. [Accessed 2 December 2013].

[14] "Elasticsearch Past Releases & Notes," Elasticsearch, [Online]. Available: http://www.elasticsearch.org/downloads/page/2/. [Accessed 2 December 2013].

[15] "MvnRepository - elasticsearch-river-mongodb," MvnRepository, [Online]. Available: http://mvnrepository.com/artifact/com.github.richardwilly98.elasticsearch/elasticsearch-river-mongodb/1.6.11. [Accessed 3 December 2013].

[16] "MvnRepository - MongoDB Java Driver," MvnRepository, [Online]. Available: http://mvnrepository.com/artifact/org.mongodb/mongo-java-driver/2.11.2. [Accessed 2 December 2013].

[17] "Running as a service on Linux," Elasticsearch, [Online]. Available: http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/setup-service.html. [Accessed 11 November 2013].

[18] "Ubuntu Documentation - MySQL," [Online]. Available: https://help.ubuntu.com/12.04/serverguide/mysql.html. [Accessed 23 January 2014].