Code Similarity Using Program Dependence Graphs
Tobin Pereira(ttp5232@rit.edu)
Advisor: Dr. Carlos Rivero(crr@cs.rit.edu)

Abstract—Propose a system which uses Program Dependence Graphs as an intermediate representation of codes to perform approximate sub graph isomorphism using graph alignment techniques for finding similar code. The document aims at defining all of the concepts used to build the system along with the results being compared with the current state of the art, JPLAG. The proposed system only works for Java codes for now.

I. INTRODUCTION

Finding things which are exactly the same in every aspect is fairly a straight forward job for both humans and computers alike. For instance given two image side by side which are the same in every aspect the human brain will barely take any efforts to come to a conclusion that they are the same, or if they are different it would reach to the conclusion that it is different in no time and not with much efforts either. Even for computers the algorithm will be as straightforward as comparing every pixel with the pixel value at the corresponding position of the other image and then reach to a conclusion that the images at the same. Now replacing the two same images in question with two texts which are exactly the same, will not change things too much for either the human brain or for the computer. Now consider changing the requirement from that of finding if the two things in questions are exactly the same to that of finding things if they are similar. The human brain is an excellent pattern detector be it with images or even with text, the job here is again fairly straight forward for the human brain. Here is where it starts getting tricky for a computer to do this. It is not really that straightforward for it to figure out similar things, be it in the form of text or images.

Let us now move from texts and images to pieces of code. A piece of code is essentially is piece of text. Now figuring out if two pieces of code are exactly the same can be converted into the simple problem of figuring out if two pieces of text are the same. Thus finding two pieces of code being exactly the same is a very simple problem to solve. Moving to find if two pieces of code are similar is a different ball game altogether. To illustrate this consider following pieces of code which are trying to count the number of odd numbers that exist within a given integer array.

```java
public int countOdd(int[] input) {
    int count = 0;
    for (int i = 0; i < input.length; i++) {
        if (input[i] % 2 != 0) {
            count++;
        }
    }
    return count;
}
```

Both the codes do exactly the same thing and are essentially the same. By just changing the variable names, a normal text based comparison automatically is thrown off. Consider the piece of code which is shown below.

```java
global int countOdd(int[] input) {
    global int countToReturn = 0;
    for (int forLoopVariable = 0; forLoopVariable < input.length; forLoopVariable++) {
        if (input[forLoopVariable] % 2 != 0) {
            countToReturn++;
        }
    }
    return countToReturn;
}
```

A normal text based comparisons fails with worse results in the above situation. To make matters even worse consider the following pieces of code which counts not only the number of odd numbers about also the number of even numbers.

```java
public int countOddEven(int[] input) {
    int countOdd = 0;
    for (int i = 0; i < input.length; i++) {
        if (input[i] % 2 != 0) {
            countOdd++;
        }
    }
    System.out.println("Odd count: "+countOdd+"and Even Count:"+countEven);
}
```

```java
global int countOddEven(int[] input) {
    global int countEven = 0;
    for (int i = 0; i < input.length; i++) {
        if (input[i] % 2 == 0) {
            countEven++;
        }
    }
    System.out.println("Odd count:"+countOdd+"and Even Count:"+countEven);
}
```
System.out.println("Odd Count: "+countOdd+" and Even Count: "+countEven);
}

Re-arranging the code will throw the text based approach off even without changing the variable name. From the above mentioned cases it should be easy to conclude that finding similar code is not an easy task because there are a few straightforward masks which can throw off a text or token based approach. The aim of this project is to address this problem of finding an effective and efficient way to find if two pieces of codes are similar.

II. OBJECTIVE

As explained above finding similar code is no easy task. The goal of this project to have a system to find similar code which is immune to the various obvious masks that throws off the normal tradition approach. To break it down further once identifying that the traditional approaches fail, the goal of this project is to represent code in a way that the representation provides the freedom required to perform matching on it. Once the representation is ready it should fairly be straightforward from there. The representation chosen to represent code is called Program Dependence Graph[1]. As the name does give it away, it is a type of graph data structure that has been chosen to represent the code. The problem of finding code that is similar then can be thought of a problem of subgraph isomorphism. Performing sub-graph isomorphism on the two Program dependence graphs in questions can give us a direct relation as to if the two codes from which the program dependence graphs were generated are similar or not.

III. PROGRAM DEPENDENCE GRAPH

It was easy to conclude from the examples in the introduction section that comparing two pieces of code in their raw form as texts will pose a lot of problems and will not be effective. Thus an intermediate representation of the code was required. The essence of a program or code can be explained effectively by tracking the flow of data and also the flow of the control in the program[1]. Considering the first example in the introduction sections where the aim of the program is count the number of odd numbers in the input array. Here if we try to take model the flow of data and that of control, it can be explained as follows: there is a count variable which has been initialized to zero. The count variable has been changed (incremented) with in a control for loop with in a control condition both of whose condition are based on the data in the input array. This statement over here gives a good idea about what the program intends to do while keeping a layer of abstraction thus not making it too specific. This representation reeks of all of the properties of a popular data structures in computer science, the one that tries to map objects (vertices) and if they are connected to one another (edges), that is a graph. A graph will be an excellent data structure which can server the purpose of being an intermediate representation of the code in question which maps the control as well as the data flow of the program [1]. Program dependence graphs was introduced in [1]. It was intended to map the data and control relationship of a program and hence the name Program Dependence Graph[1].

As other graph, Program Dependence Graph is made up of a set of vertices and a set of edges. Every statement in the code is converted into one or more Program Dependence Graph vertices. The Program Dependence Graph vertices store the actual content of the statement into something called as labels. The vertices also belong to a set number of vertex types available. They as listed as below

- Declaration Nodes.
  Declaration nodes as the name suggests are nodes that correspond to statements such as int i; or ArrayList<Integer> array= new ArrayList<Integer>;

- Assignment Nodes.
  Assignment Nodes are something like i = 0; or i++. There is not much difference between declaration nodes and assignments node. Declaration nodes can be thought of assignment nodes with out the explicit assignment of values.

- Control Nodes.
  Control nodes on the other hand are nodes that can be either defined as if statements or loop statements. Loop statements can essentially be of following types
    - for
    - while
    - do-while

The thing common across them, the loops and if statements, is that they control the execution of a statement or a set of statements which are defined within their scope. And the execution of this or these statements depends on a conditional statement which essentially gives a boolean value of either true or false. This control part of these control structures are what are encompassed by the control nodes. Thus the entire if statement itself is a control node. On the other hand for loops like for or while, it is just the statement which defines the control and gives the boolean value of true or false. For instance,

```java
for(int i=0; i<a.length; i++) {
    ...
    ...
}
```

The above for loop the control is essentially the statement in bold which is i < a.length. The other two statements are one declaration node and one assignment nodes. i=0 is a declaration node which are not very different from assignment node as explained above and i++ is also an assignment node in which is assigning a value of i+1 to i. The control for loop structures is broken down into two assignment nodes and one single control node. Such are the cases where a single statement in the program are converted into more than one vertex in the Program Dependence Graph. And a control if statement is only broken down into a single control node as explained above. Now these control nodes can be one of the eight following sub types:
- Greater than.
  A control node which has “>” operator defining the control statement.
- Less than.
  A control node which has “<” operator defining the control statement.
- Equality.
  A control node which has “==” operator defining the control statement.
- Inequality.
  A control node which has “!=“ operator defining the control statement.
- Modulo equality.
  A control node which has “%” and “==” operator defining the control statement.
- Modulo inequality.
  A control node which has “%” and “!=“ operator defining the control statement.
- Combination.
  Use of “&&” and/or “||” to use a combination of the above mentioned.

• Call Nodes.
  Statement such as System.out.println(), return(); are the statements that get converted into a call vertex or node.

The diagram below shows the taxonomy of the types of nodes or vertices of a Program Dependence Graph can be.

![Taxonomy of the types of Vertices in Program Dependence Graphs](image)

Now that the vertices and their types have been dealt with, let us consider how the edges are formed between these vertices. The edges are the things that are going to model the flow of the program both, data and the control. Thus without the edges the Program Dependence Graph will have no value. But the way these edges are connected is very important so that it correctly models the flow that is required. The edges are connected by traversing the code top down. The Program Dependence Graph are created line by line when the code is being parsed. The vertices are created line by line. Now the data flow can be represented by creating an edge between the last time (or the last line compared to the current line) the variable was assigned a new value. Thus data flow edges are going to be made only in the cases when there is an assignment node, and they can exist only between assignment nodes and any other node types. Because the source of the data has to occur either at an assignment or a declaration node. These are the only nodes types through which data flow might occur. The control of the program is usually linear thus there is no point making control edges between successive statements. The only place where the control of a program changes in when there is loop or an if statement. Thus all the statements that exist within a loop or all the ones that exist within an if statement will have control edges with the corresponding loop or if node. Consider the follow example Fig.2 of a piece of code and its corresponding program dependence graph.

```java
public int countEven(int input[]) {
    int count = 0;
    for(int i=0; i<input.length; i++) {
        if(input[i]%2==0) {
            count++;
        }
    }
    return count;
}
```

![Example code and PDG](image)

Fig. 2: Example code and PDG.

The corresponding table I of the vertices are as shown below:

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Vertex Type</th>
<th>Vertex Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>Declaration</td>
<td>int input[]</td>
</tr>
<tr>
<td>V2</td>
<td>Assignment</td>
<td>int count = 0</td>
</tr>
<tr>
<td>V3</td>
<td>Assignment</td>
<td></td>
</tr>
<tr>
<td>V4</td>
<td>Control LESS THAN</td>
<td>i&lt;input.length</td>
</tr>
<tr>
<td>V5</td>
<td>Assignment</td>
<td>i++</td>
</tr>
<tr>
<td>V6</td>
<td>Control MOD EQUALITY</td>
<td>input[i]%2==0</td>
</tr>
<tr>
<td>V7</td>
<td>Assignment</td>
<td>count++</td>
</tr>
<tr>
<td>V8</td>
<td>Call</td>
<td>return count</td>
</tr>
</tbody>
</table>

TABLE I: Vertices their types and contents

Considering the above code and its corresponding Program Dependence Graph, it is easy to see how program dependence graph captures the data and control flow. For instance if we notice there are two data edges coming out of v1 and going into v4 and v6. From the table above it can be seen that v1 is the vertex which corresponds to the statement of declaring the input array. This value is being read in v4 and v6 which are both control nodes, v4 reads its length value and v6 on the other hand read the actual values at the ith location. Thus the data edges do give a good idea about how the data is flowing through the program dependence graph. Consider the vertices v2, v7 and v8. All three of them deal with the variable count. Vertex v2 is where is gets assigned for the first time, v7 is where it gets incremented and v8 is where itâ€™s value gets returned. Thus there is an edge going from v2 to v7 and v7 to v8 as well which show the flow if the data in variable count. Moving on to control edges, control edges only go out of â€œifâ€ and loop statements. For instance in the example we are dealing with the two control edges that we have are v4 and v6. v4 is formed from the for loop statement for(int i=-; i<input.length; i++). The fore mentioned for loop is transformed into three vertices, one control v4 and two assignments v3 and v5. Control vertices as discussed earlier are a single vertex that have an associated condition along with it, a condition that essentially returns a boolean value.
Coming back to the example at hand statements,

```java
if (i < input.length) {
    count++;
}
```

are all bound within the for loop. Thus there is a control edge going from v4 to v6 and v7 too, indicating that these two vertices that are bound by the condition in vertex v4. Similar logic and be applied to vertices v6 and v7, that v7 is bound by the condition in v6. 
To sum it all up program dependence graph is an intermediate representation of a code that basically tries to map the data along with the control flow [1].

IV. HYPOTHESIS

The aim of this project is try to find similar code. We have already defined an intermediate representation of the code using which we are going to go about the business of doing this. This representation being a graph, this problem can be directly thought of as finding subgraph isomorphism of the two Program Dependence Graph of the code which are to be figured out if they are similar.
The only problem is that subgraph isomorphism is known to be a NP-hard problem, in other words there is no efficient way to solve this problem. To scale this problem into something that can be solved in a more efficient and elegant manner, graph alignment techniques [2], are going to be employed.

V. GRAPH ALIGNMENT TECHNIQUE

This technique was first put forward in 2004 in the paper [1]. It tries to find mapping between two graphs using the structural information of every vertex in the two graphs which is addressed as signature of vertex [1] in the technique. This signature of vertex is a vector of size 73 and the pseudo code of the entire technique is shown in Algorithm 1. For in-depth explanation of the technique please refer to [1] and the independent study report.

It can be seen this technique produces a mapping which is solely based on the structural information of the vertices in the graph and also because it does not have any form of thresholding or any filtering, this method will generate a mapping no matter how different the graph. Thus solely using this technique, the mappings that it will produce cannot be taken into account without some kind of processing. This is because as the above mentioned method will give mappings no matter how different two graphs are, This will lead to mappings which might be totally wrong and which do not makes sense. This is the case where there is no similar code what so ever in the two given input codes. The mappings that graph alignment generates will be something that does not make sense. Hence the need for further processing.

VI. PROGRAM DEPENDENCE ALIGNMENT

The graph alignment technique is going to generate the best possible mapping using only the structural information of the vertices of the graph thus essentially using the structural information of the graph itself [1]. The problem here is that it will find the best fitting sub graph from the two input graphs, even if the best that is found not supposed to be found. Thus as discussed earlier given two pieces of code if we get some random mappings the questions is how are we supposed to validate it. As mentioned earlier till this point only the structural information of the Program dependence graph has been used, this is when we start making use of the different types of labels that were defined to validate these mappings that were found. Just as a recap there are 4 main types of labels that were defined:
1) Declaration
2) Assignment
3) Control
4) Call

The idea here in program dependence alignment is involving some kind of penalty if the pair that is mapped do not have the same label. If the labels are the same this indicates that the mapping was on par with what was needed, thus there is no need for any penalty. Now coming back to if the labels not matching, let us consider the situation where the two vertices mapped are having labels declaration and assignment. It should be noted here that declaration and assignment vertices that are very similar to one another. Declaration vertex can be thought of a special type of assignment vertex and things should still make sense in the program dependence graph. Thus when the two differing labels are that of assignment and declaration there is no penalty involved. Apart from declaration and assignment any vertex pair having different labels the penalty involved is the maximum, meaning that pair is removed from the mappings that was produced. There is one additional scenario that needs to be considered is when two the vertices are control. The problem here is that the control labels are further divided into 7 sub types. It they belong to the same subtype no penalty is involved. But when they do not belong to the same subtype a penalty of
0.02 is imposed on the similarity score of the two vertices. When one of the vertices is a combination control vertex for every control label that is missing in the other vertex a penalty of 0.02 is imposed. But after these penalties are imposed the mapped pairs that have similarity score of greater than 1.2 is considered to be bad and are removed from the final mappings.

Algorithm 2 shows the pseudo code for imposing penalty on the mappings found and to get the final mappings: Consider

**Data:** Query Graph U, Target Graph V, mappings

**Result:** final mappings after PDG alignment

for currentPair(u,v) in mappings do
  if similarity(currentPair) > 1.2 then
    mappings.remove(currentPair);
    continue;
  end
  typeOne = getVertexType(u);
  typeTwo = getVertexType(v);
  if typeOne == Declaration then
    typeOne = Assignment;
  end
  if typeTwo == Declaration then
    typeTwo = Assignment;
  end
  if typeOne == Control_combination or typeTwo == Control_combination then
    count = getCountOfMissingIndividualControls(typeOne, typeTwo);
    similarity[currentPair] = similarity[currentPair] + (count*0.2);
  end
  if similarity(currentPair) > 1.2 then
    mappings.remove(currentPair);
    continue;
  end
  if typeOne == typeTwo then
    continue;
  end
  mappings.remove(currentPair);
end

**Algorithm 2:** Algorithm to generate the final PDG mappings

the following pieces of codes, and the entire flow of how the mappings take place. Apart from the final mappings, the signature of each node, the similarity of the mapped nodes and the final mapping after considering the labels of the vertices are also shown below. The example taken is same the ones in the introduction.

**Query Code**

```java
public int countOdd(int input[]) {
    int count = 0;
    for (int i=0; i<input.length; i++) {
        if (input[i] % 2 != 0) {
            count++;
        }
    }
    return count;
}
```

**Target Code**

```java
public void oddEvenCounts(int input[]) {
    int countOdd = 0;
    for (int i=0; i<input.length; i++) {
        if (input[i] % 2 != 0) {
            countOdd++;
        }
    }
    int countEven = 0;
    for (int i=0; i<input.length; i++) {
        if (input[i] % 2 == 0) {
            countEven++;
        }
    }
    System.out.println(Odd Count +countOdd+ Even Count +countEven);
}
```

The corresponding Program Dependence Graphs are as shown below:

**Query Program Dependence Graph:**

```
Fig. 3: Program Dependence Graph of Target Code
```

**Target Program Dependence Graph:**

```
Fig. 4: Program Dependence Graph of Target Code
```

**Summary Program Dependence Table of Query:** Summary
TABLE II: Vertex Type and Content of Query Code

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Vertex Type</th>
<th>Vertex Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>Declaration</td>
<td>int input[]</td>
</tr>
<tr>
<td>V2</td>
<td>Assignment</td>
<td>int count = 0</td>
</tr>
<tr>
<td>V3</td>
<td>Assignment</td>
<td>int rait</td>
</tr>
<tr>
<td>V4</td>
<td>Control LESS THAN</td>
<td>i&lt;input.length</td>
</tr>
<tr>
<td>V5</td>
<td>Assignment</td>
<td>i++</td>
</tr>
<tr>
<td>V6</td>
<td>Control MOD EQUALITY</td>
<td>input[i]%2==0</td>
</tr>
<tr>
<td>V7</td>
<td>Assignment</td>
<td>countEven++</td>
</tr>
<tr>
<td>V8</td>
<td>Call</td>
<td>return count</td>
</tr>
</tbody>
</table>

Program Dependence Table of Target:

TABLE III: Vertex Type and Content of Target Code

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Vertex Type</th>
<th>Vertex Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>u1</td>
<td>Declaration</td>
<td>int input[]</td>
</tr>
<tr>
<td>u2</td>
<td>Assignment</td>
<td>int countOdd = 0</td>
</tr>
<tr>
<td>u3</td>
<td>Assignment</td>
<td>int i=0</td>
</tr>
<tr>
<td>u4</td>
<td>Control LESS THAN</td>
<td>i&lt;input.length</td>
</tr>
<tr>
<td>u5</td>
<td>Assignment</td>
<td>i++</td>
</tr>
<tr>
<td>u6</td>
<td>Control MOD INEQUALITY</td>
<td>input[i]%2!=0</td>
</tr>
<tr>
<td>u7</td>
<td>Assignment</td>
<td>countOdd++</td>
</tr>
<tr>
<td>u8</td>
<td>Assignment</td>
<td>int countEven = 0</td>
</tr>
<tr>
<td>u9</td>
<td>Assignment</td>
<td>int i=0</td>
</tr>
<tr>
<td>u10</td>
<td>Control LESS THAN</td>
<td>i&lt;input.length</td>
</tr>
<tr>
<td>u11</td>
<td>Assignment</td>
<td>i++</td>
</tr>
<tr>
<td>u12</td>
<td>Control MOD INEQUALITY</td>
<td>input[i]%2!=0</td>
</tr>
<tr>
<td>u13</td>
<td>Assignment</td>
<td>countEven++</td>
</tr>
<tr>
<td>u14</td>
<td>Call</td>
<td>System.out.print((\d+))</td>
</tr>
</tbody>
</table>

Signature Table of Query graph can be seen as shown in table IV The corresponding Signature Table of Target graph can be seen as show in table V

The Alignment Cost of vertex pairs without labels are shown in table VII.

The Alignment Cost of vertex pairs with labels are shown in table VIII. The ones that are highlighted in blue are the ones that have the lowest cost and one that are highlighted in orange are the ones whose cost suffered a penalty because of the labels. And the X is marked when the labels are incompatible. Now from this table it can be inferred that there are multiple mappings that can be generated if we end up choosing the minimum costs. To get around this the initial seed point is selected by taking into account the minimum cost that occurs in the through the table. And if there are multiple pairs that have the same minimum cost then ties are broken by choosing the pair whoZs query vertex has the maximum degree(a sum of the in and out degree), there the pair then turns out to be (v6, u6). This becomes our seed pair now. And now the mapping is expanded based on the neighbors of the query vertices. And again the ties are now broken as per the nearest neighbor of the target vertex. For instance the immediate neighbors of v6 are v5, v4, v1, v7. Assume we choose v5. Now the table shows that there are two vertices that have the same minimum cost to align with v5 which are u3 and u11. To break this tie, the immediate neighbors of u6 is taken into consideration which are u5, u4, u1 and u7. Thus u11 is not a part of this set. And thus u3 is chosen to be the target vertex which gets mapped to v5. This is how ties are broken. If there are situations when ties still do not break they are broken randomly then. Thus the final mappings produced are:

TABLE VI: Final Mappings that are generated

<table>
<thead>
<tr>
<th>Query Vertex</th>
<th>Target Vertex</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>u1</td>
<td>1.2</td>
</tr>
<tr>
<td>v2</td>
<td>u2</td>
<td>0.0</td>
</tr>
<tr>
<td>v3</td>
<td>u3</td>
<td>0.0</td>
</tr>
<tr>
<td>v4</td>
<td>u4</td>
<td>0.0</td>
</tr>
<tr>
<td>v5</td>
<td>u5</td>
<td>0.0</td>
</tr>
<tr>
<td>v6</td>
<td>u6</td>
<td>0.0</td>
</tr>
<tr>
<td>v7</td>
<td>u7</td>
<td>0.0</td>
</tr>
<tr>
<td>v8</td>
<td>u14</td>
<td>0.0</td>
</tr>
</tbody>
</table>

VII. RESULTS

Before getting into how well the proposed algorithm performs as compared to the current state of the art, we need to figure out a dataset which helps us perform this comparison. Initial research did not lead to any place which had public access to a dataset of code which had annotation as to which were similar to one another and which werenZt. This posed a problem for the revaluation of the system at hand. One path ahead was creating our own dataset, but this would a very slow process and might lead to a dataset which is biased towards the method suggested in this paper. Thus there was a need for a dataset which was completely neutral and which also had scope for similar code or plagiarized code. The obvious place to look was a place where multiple people try to solve the same problem and all of the submissions need to be public which serves two purposes, first one was it needed to be public; the need being make it easy to store and save the submissions, and the second not so obvious one is as the submissions are public there is a good chance of users checking out code submitted by other users and then using the same code to make their own submission. One such platform is codechef [3]. This a platform where a common problem is posted regularly and the users on the platform take a stab at it. All the submissions made by users are visible to everyone. This seemed like a good place to start building a dataset. The submissions on this website could be made in a variety of different programming languages, namely java, python, C and CPP. This project is mainly dealing with Java for now. The logic can be extended for any of the above mentioned languages as well. Roughly given a fairly old problem posted on [3], about a couple of week old, will have at least 600 odd submissions just in java. This thus seemed like a good place to build the dataset. To scrape this website given a problem and to generate the dataset to be worked with along with all the details, a web crawler was implemented as well. This web crawler as implemented in Python making using of Mechanize and the beautiful soup library. The purpose of this web crawler was given a problem name from [3], this crawler would crawl through [3] and retrieve all the java submissions that was available for the given problem. Apart from the code itself it also extract the use who submitted it as well as if it complied or not, what kind of error did it throw if it did throw any and if the submission was correct or if it produced the wrong answer. These details were stored in a details.csv file that the crawler generates. The crawler generate the following directory structures and the corresponding java files in them:
TABLE IV: Signature of all the Query Vertices

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Orbit Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>u1</td>
<td>1</td>
</tr>
<tr>
<td>u2</td>
<td>2</td>
</tr>
<tr>
<td>u3</td>
<td>3</td>
</tr>
<tr>
<td>u4</td>
<td>4</td>
</tr>
<tr>
<td>u5</td>
<td>5</td>
</tr>
<tr>
<td>u6</td>
<td>6</td>
</tr>
<tr>
<td>u7</td>
<td>7</td>
</tr>
<tr>
<td>u8</td>
<td>8</td>
</tr>
<tr>
<td>u9</td>
<td>9</td>
</tr>
<tr>
<td>u10</td>
<td>10</td>
</tr>
<tr>
<td>u11</td>
<td>11</td>
</tr>
<tr>
<td>u12</td>
<td>12</td>
</tr>
<tr>
<td>u13</td>
<td>13</td>
</tr>
<tr>
<td>u14</td>
<td>14</td>
</tr>
</tbody>
</table>

TABLE V: Signature of all the Target Vertices

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Orbit Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>1</td>
</tr>
<tr>
<td>v2</td>
<td>2</td>
</tr>
<tr>
<td>v3</td>
<td>3</td>
</tr>
<tr>
<td>v4</td>
<td>4</td>
</tr>
<tr>
<td>v5</td>
<td>5</td>
</tr>
<tr>
<td>v6</td>
<td>6</td>
</tr>
<tr>
<td>v7</td>
<td>7</td>
</tr>
<tr>
<td>v8</td>
<td>8</td>
</tr>
</tbody>
</table>

TABLE VII: Alignment Costs of each of the query vertices with the corresponding target vertices

<table>
<thead>
<tr>
<th>Query Vertex</th>
<th>Target Vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>u1</td>
<td>1.5</td>
</tr>
<tr>
<td>u2</td>
<td>2.0</td>
</tr>
<tr>
<td>u3</td>
<td>2.0</td>
</tr>
<tr>
<td>u4</td>
<td>2.0</td>
</tr>
<tr>
<td>u5</td>
<td>2.0</td>
</tr>
<tr>
<td>u6</td>
<td>2.0</td>
</tr>
<tr>
<td>u7</td>
<td>2.0</td>
</tr>
<tr>
<td>u8</td>
<td>2.0</td>
</tr>
<tr>
<td>u9</td>
<td>2.0</td>
</tr>
<tr>
<td>u10</td>
<td>2.0</td>
</tr>
<tr>
<td>u11</td>
<td>2.0</td>
</tr>
<tr>
<td>u12</td>
<td>2.0</td>
</tr>
<tr>
<td>u13</td>
<td>2.0</td>
</tr>
<tr>
<td>u14</td>
<td>2.0</td>
</tr>
</tbody>
</table>

TABLE VIII: Alignment Costs of each of the query vertices with the corresponding target vertices considering the labels as well

<table>
<thead>
<tr>
<th>Query Vertex</th>
<th>Target Vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>u1</td>
<td>1.5</td>
</tr>
<tr>
<td>u2</td>
<td>2.0</td>
</tr>
<tr>
<td>u3</td>
<td>X</td>
</tr>
<tr>
<td>u4</td>
<td>2.0</td>
</tr>
<tr>
<td>u5</td>
<td>2.0</td>
</tr>
<tr>
<td>u6</td>
<td>2.0</td>
</tr>
<tr>
<td>u7</td>
<td>2.0</td>
</tr>
<tr>
<td>u8</td>
<td>2.0</td>
</tr>
<tr>
<td>u9</td>
<td>2.0</td>
</tr>
<tr>
<td>u10</td>
<td>2.0</td>
</tr>
<tr>
<td>u11</td>
<td>2.0</td>
</tr>
<tr>
<td>u12</td>
<td>2.0</td>
</tr>
<tr>
<td>u13</td>
<td>2.0</td>
</tr>
</tbody>
</table>

1) **Problem Name**

The name of the problem that the crawler scraped codechef.

a) **Submissions**

The folder which contains the java files in its subdirectories.

i) **all**

This folder contains all of the java submissions for the problem.

ii) **compilation Error**

Contains all the java submissions that have compilation errors.

iii) **correct**

Contains all the java submissions that produced correct results.

iv) **run time error**

Contains all the java submissions with java run time errors.

v) **wrong**

Contains all the java submissions that produced wrong results.

b) **Users**

Contains subdirectories of each user and their submissions.

c) **details.csv**

Contains all the details of each of the java submissions found.

Now that the dataset is formed and ready, let us move on to evaluations. The performance of the proposed system is going to be compared against the current state of the art which the JPLAG. JPLAG generates clusters of files which are similar to one another. The output of the proposed system has been tuned.
to produce clusters of file which are similar too, thus a direct comparison can be made with between the clusters produced. The system was run against the three assignments picked from [3]. Each of them having roughly 600 submissions. The details of each of them are given in Table IX

Each of these assignments were fed in JPLAG and the proposed system. As mentioned earlier both the systems end up generating cluster of similar codes. Thus codes belonging to the same clusters are intended to be similar. There are three graphs that were generated of each of the problems. The distribution of each of them is on par with the expectations. The first graph shows the percentage of overlap between the clusters generated by JPLAG and proposed system. This essentially means counting of the number of files common in the two clusters and calculating the percentage with respect to counts in the JPLAG clusters. The X axis of this graph are the cluster numbers corresponding to the clusters generated by JPLAG. And Y axis is the percentage overlap with the corresponding cluster generated by the system. The X axis is sorted as per the highest percent of overlap. This graph give us a good indication as to how well the proposed system was able to perform, in other terms it gives a good indication if the system could identify all the cases that JPLAG could. The second graph counts the number of files that were missing from JPLAG clusters that were found in the clusters that were found. Here the X axis is again the cluster numbers representing the cluster number generated by JPLAG, they maintain the same order as in the first graph to there can be direct comparison that can be made. The Y axis is count of the missing files. The second graph gives further understanding as to how many files were missed by the clusters produced by the system. The third graph the count of files that are additionally to how many files were missed by the clusters produced by the system. The third graph the count of files that are additionally present in the clusters produced by the system. This graph doesn’t really provide as much value as that of the above too, but still it give a good indication about how many additional files were there in each of the clusters generated by the system. The proposed system aims at identifying all of the similarities are found by JPLAG and at-least in theory more than the one JPLAG does because of the kind of labels associated for vertices of the graph which will turn out to be very specific.

VIII. CONCLUSION

It can be clearly concluded from the graphs in the results that the clusters generated by the system were more general as compared to the ones generated by JPLAG. This can be concluded by just looking at the first and second graphs of each of the assignments, which map the percentage overlap and the number of files that are missing. The cluster numbers are the same in both the graphs, thus a direct comparison can be made. The cluster number are ordered in the order of the highest percent of overlap in all the graphs for easier comparison. As it can be seen that all the clusters that have overlap of greater than 50% almost have no files that are missing from them. As the percentage drops down so does the count of the missing files, indicating that those missing files are included in the cluster that was defined before. This inference can be notes also by looking at the distribution of the graph of additional files in clusters, which shows most of the clusters generated by the system is bigger than the one generated by JPLAG. Thus the system can compare with the current state of the art.

From the results it can be easily concluded that the Program Dependence Graph representation of code is an excellent way to represent code to perform similarity matching and performance measurement as well[1]. It is also worth mentioning that the graph alignment techniques that were used to perform approximate subgraph matching were on point and did help speed up the entire process without giving a drop in accuracy.

IX. FUTURE WORK

The system at hand can detect similar code only for java as for now. The individual components of the system are designed to work with Program Dependence Graphs, thus the immediate future work involved will be to generate parses for other programming language like python, c, cpp, c# and others. Thus having the parser implemented and using the exact same framework for generating the Program Dependence Graphs which system can be ported for other programming languages as well.

It is also interesting to point out the use case of using Program Dependence Graphs for providing large scale automated feedback of student assignments. The idea here is using the techniques after to identify patterns that are repeating through the submissions and presenting them as patterns to the grader and the grader now need to provide feedback for only the identified repeating patterns which will be applied to all of the submissions with the repeating pattern. This makes it large scale and personalized as well because feedback will be associated for vertices of the graph which will turn out to be very specific.

REFERENCES

### TABLE IX: Information about the Assignments

<table>
<thead>
<tr>
<th>Assignment Name</th>
<th>Number of Submissions</th>
<th>Number of Users</th>
<th>Link to the description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGAME</td>
<td>623</td>
<td>148</td>
<td><a href="https://www.codechef.com/problems/RGAME">https://www.codechef.com/problems/RGAME</a></td>
</tr>
<tr>
<td>Life, Universe and Everything</td>
<td>625</td>
<td>197</td>
<td><a href="https://www.codechef.com/problems/TEST">https://www.codechef.com/problems/TEST</a></td>
</tr>
<tr>
<td>Ciel and Receipt</td>
<td>625</td>
<td>298</td>
<td><a href="https://www.codechef.com/problems/CIELRCPT">https://www.codechef.com/problems/CIELRCPT</a></td>
</tr>
</tbody>
</table>

(a) Percentage overlap between the JPLAG clusters and the clusters generated by the system
(b) Count of missing files from the clusters generated by the system as compared to the one’s by JPLAG
(c) Count of Additional files in the clusters generated by the system as compared to the one’s by JPLAG

Fig. 5: Graphs showing results for Assignment RGAME

(a) Percentage overlap between the JPLAG clusters and the clusters generated by the system
(b) Count of missing files from the clusters generated by the system as compared to the one’s by JPLAG
(c) Count of Additional files in the clusters generated by the system as compared to the one’s by JPLAG

Fig. 6: Graphs showing results for Assignment Life, Universe and Everything

(a) Percentage overlap between the JPLAG clusters and the clusters generated by the system
(b) Count of missing files from the clusters generated by the system as compared to the one’s by JPLAG
(c) Count of Additional files in the clusters generated by the system as compared to the one’s by JPLAG

Fig. 7: Graphs showing results for Assignment Ciel and Receipt