Code Clone Detection using Graphs and Adjacency Structures

Mukesh Kumar\textsuperscript{a}, Lalit Kumar Sagar\textsuperscript{b}, Saurabh Kumar\textsuperscript{c}
\textsuperscript{a, b, c}\ Assistant Professor, Dr. K.N.M.I.E.T, India

Abstract: Code clone detection is the common aspect of reuse activity. Copying code fragments and then reuse with or without modifications are common activities in software development. This type of reuse approach of existing code is called code cloning and the pasted code fragment is called a clone of the original \cite{1}. Existing approaches does not make use of adjacency structures and their properties. In this paper we present an efficient way of finding code clones by using adjacency structure. We developed a directed graph of the source code and parsed the information into adjacency structure. By using the properties of adjacency structure we can find the in-degree and out-degree of a particular directed graph. Our insight is to deduce the flow on a particular node of the directed graphs to find the similarity between the nodes of the directed graph. We implemented this algorithm practically by using the tool named as control flow graph factory.

Keywords: Control flow graphs; Adjacency structure; In-degree; Out-degree.

I. INTRODUCTION

1.1 Code Cloning

Code duplication or copying a code fragment and then reuse by pasting with or without any modifications is a well known code smell in software maintenance. Several studies show that about 5\% to 20\% of software systems can contain duplicated code, which are basically the results of copying existing code fragments and using then by pasting with or without minor modifications \cite{1}.

A code fragment CF1, which is a sequence of code line is clone to another code fragment CF2, if they have similar properties i.e. F (CF1) = F(CF2), where “F” is a similar function. Two fragments that have similar properties are referred as clone pair (CF1, CF2) and when many fragment are similar then they form clone class or clone group \cite{2}.

1.2 Type of clones:-

There are two main kinds of similarity between code fragments. Fragments can be similar based on the similarity of their textual similarity \cite{3} \cite{4} \cite{5}, or they can be similar based on their functionality (independent of their text) \cite{6} \cite{7} \cite{8} \cite{9}. The first kind of clone is often the result of copying a code fragment and pasting into another location. In the following we provide the types of clones based on both the textual (Types 1 to 3) \cite{10}, and functional (Type 4) \cite{11} similarities.

Type-1: Identical code fragments except for variations in whitespace, layout and comments.

Type-2: Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.

Type-3: Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.

Type-4: Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

II. AUTHOR ARTWORK

A tool “Control Flow Graph Factory” is used to generate control flow graphs. This tool generates different types of graphs like Byte code graphs, Basic Block Graph. A basic introduction of Control Flow Graph Factory is given below.

2.1 Control Flow Graph Factory

Control Flow Graph Factory is an Eclipse plug-in which generates control flow graphs from java code, edit them and export to GraphXML, DOT or several image formats.

- Features
  - Automatic generation of several types of control flow graphs from Java byte code like:
    - Byte Code Graphs
    - Source Code Graphs
    - Basic Block Graphs
  - Editing of control flow graphs
    - Move, create, delete, rename, ... nodes
  - Multiple algorithms for automatic layout (serial, hierarchical)
There are several steps to generate a control flow graph. These steps are explained below.

1. To generate the graph for the method “main” select the method in “package explorer” and open the context menu “Create Control Flow Graph”. Select submenu “Source code graph” to generate a source code graph for this method.

2. Generate a byte code or a basic code graph in the same way. For that use the context menu in the package explorer "Create Control Flow Graph/Byte code graph" or "Create Control Flow Graph/Basic block graph.

3. For export the graph in DOT, GraphXML format or to an image use the export functions provided by the Control flow graph Factory. For finding the geometry information of the graphs (may be basic block, source code, byte code) export the graph by export geometric info. This geometry information, gives the information about all the vertices and edges which are connected with each other. For example:-

```java
Package test1;

Public class test1 {
    Public static void main(String[] args) {
        int i=0;
        while(i<10)
        {
            System.out.print("hello");
        }
    }
}
```

Take a second java code that print hello using “For” loop.

```java
Package test2;

Public class test2 {
    Public static void main(String[] args) {
        int i=0;
        for (i=0;i<10; i++)
        {
            System.out.print("hello");
        }
    }
}
```

The Basic block Graph of “While” loop code is:

![Basic Block Graph of While loop]

The exported information of the graph is:-

```plaintext
2 [label="B1"]
3 [label="B2"]
4 [label="B3"]
5 [label="B4"]
6 [label="EXIT"]
7 [label="START"]
```

Similarly the Basic Block Graph of “For” loop is:-

![Basic Block Graph of For loop]
The adjacency matrix for the exported information is in the below matrix (Fig. 3).

The adjacency matrix for the exported information is in the below matrix (Fig. 3).

```
<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

This means 6 nodes are presented in the graph and
- Node 7 is connected with node 2.
- Node 2 is connected with node 3.
- Node 4 is connected with node 3.
- Node 3 is connected with node 4.
- Node 3 is connected with node 5.
- Node 5 is connected with node 6.

So the exported information is used in the adjacency matrix to find the “out-degree” of a graph.
- An adjacency matrix is created on the basis of exported information.
- Take the transpose of adjacency matrix.
- Multiplication of adjacency matrix with its transpose provides the information about its out-degree and of the directed graph.

Again the exported information is used in the adjacency matrix to find the “in-degree” of a graph.
- An adjacency matrix is created on the basis of exported information.
- Take transpose of adjacency matrix.
- But at that time we multiply transpose of matrix and adjacency matrix. The multiplication of transpose of matrix and adjacency matrix gives in-degree of the graph.

The above process gives the “in-degree” and “out-degree” of a graph. “In-degree” and “out-degree” of two graphs is compared according to “in-degree” of one graph with “in-degree” of second graph, and similarly for “out-degree”.

The above algorithm gives the information about which nodes are similar in two graphs, and how the information flows from one node to another node, and how many “in-degree” and “out-degree” each node have. Then comparison of these two graphs is done on the basis of their in-degree and out-degree.

Here is an example to understand the following procedure:-

Example 1 includes an algorithm to find the out-degree of a graph using following steps:

1. Take a directed graph with 3 nodes.
2. Draw an adjacency matrix for that graph.
3. Take the transpose of the adjacency matrix which is obtained from the graph.
5. Find a new matrix, the new matrix diagonal element gives the out-degree of that graph.
Example 2 uses an algorithm to find the in-degree of a graph:

1. Take a directed graph with 3 nodes.
2. Draw an adjacency matrix for that graph.
3. Take the transpose of the matrix which is obtained from the graph.
4. Multiply transpose of adjacency matrix and adjacency matrix. \( (A^T \cdot A) \).
5. Find a new matrix, the new matrix diagonal element gives the in-degree of that graph.

When in-degree and out-degree of a graph is available, compare it with two graphs.

The above algorithm finds in-degree and out-degree of any graph, so based on in-degree and out-degree algorithm can compares two graphs, and find out which node is similar to each other.

This is the mathematical approach to find out the clones with similar nodes from a source code.

Psuedo code

Input :- 2d array of strings
output :- matrix which contains the outdegree in 1st row and indegree in 2nd row of a graph.

```plaintext
for aa←1 to count_of_lines do
    If text[aa].contains(">")
        adj[0] = Source_of_edge
        adj[1] = target_of_edge
        l++
    end if
end for

//Calculating minimum and maximum
for aa←1 to l
    for j←0 to 1
        if adj[aa][j] > max
            max = adj[aa][j]
        endif
        if (adj[aa][j] < min)
            min = adj[aa][j]
        endif
    end for
```

The algorithm to find in-degree and out-degree of a graph is given below:

```
fig.4 Directed Graph

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

fig.5 Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

fig.6 Transpose of adjacency matrix

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

fig.7 (A^T.A)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

fig.8 Directed Graph

fig.9 Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

fig.10 Transpose of Adjacency matrix

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

fig.11 (A^T.A)
```plaintext
min = adj[j][k]
endif
endfor
endfor
//Forming Adjacency
for j←i-1 to 0
    adjacency[adj[j][0] - min][adj[j][1] - min] = 1
endfor
//Forming transpose
for j←0 to max
    for k←0 to max
        if adjacency[j][k]==1
            adjacencyT[k][j] =1
        endif
    end for
end for
//Multiplying Adjacency and its transpose for outdegree
for i1←0 to max-min
    for j←0 to max-min
        for k←0 to max-min
            outdegree[i1][j] += adjacency[i1][k] * adjacencyT[k][j]
        end for
    end for
end for
//Multiplying transpose and Adjacency for indegree
for i1←0 to max-min
    for j←0 to max-min
        for k←0 to max-min
            indegree[i1][j] += adjacencyT[i1][k] * adjacency[k][j]
        end for
    end for
end for
// forming a club-up matrix which contains the outdegree in 1st row and indegree in 2nd row
for i1← 0 to max - min
    clubup[0][i1] = outdegree[i1][i1]
    clubup[1][i1] = indegree[i1][i1]
endfor

REFERENCES
```