Enhancing Program Dependency Graph Based Clone Detection using Approximate Subgraph Matching

C. M. Kamalpriya
Bombardier Transportation India Pvt. Limited
Vadodara, India
E-mail: kamalpriya.cm@gmail.com

Paramvir Singh
National Institute of Technology
Jalandhar, India
E-mail: singhpv@nitj.ac.in

***Abstract*—Software code clone detection techniques and tools play a major role in improving the software quality as well as saving maintenance cost and effort. Program Dependency Graph (PDG) based clone detection techniques have a key advantage over other techniques as they are capable of detecting non-contiguous code clones in addition to contiguous clones. We propose further enhancement to current state of the art PDG-based detection to identify all possible (exact and approximate) clone relations from the obtained clone pair (PDG-based) results using Approximate Subgraph Matching (ASM). We obtain clone results of our proposed technique on three subject software systems, and validate the results on eclipse-ant from Bellon's benchmark. We also present a new ASM-based distance measure to represent the similarity between software code clones.**

Keywords—Software Clone Detection; Clone Relations; Approximate Clones; Subsumed Clones; Program Dependency Graph; Software Maintenance

I. INTRODUCTION

Software maintenance is one of the paramount activities in software engineering. It accounts for over 60 percent of the total effort and cost expended in overall software engineering process. During software development process, programmers often reuse existing code with or without modifications, to build new code. This replication of code fragments, famously known as code cloning, facilitates quick software development and modification as per change requests and minimizes the overhead of procedural calls. However, it also results in higher maintenance cost and effort, increased probability of bug propagation, lesser system understanding, larger system size, etc. Experiments performed by Baker suggest that 20-30% of large software systems are cloned code [1]. It is hence vital to detect and eliminate code clones in order to enable cost-cum-effort effective and high quality software development.

Code clones are categorized in a number of ways such as contiguous/non-contiguous (based on the contiguity of matching program elements), exact/approximate (based on the amount and kind of replication), maximal/subsumed (based on the size of the detected clone pair), etc. Type 3 clones occur when a code fragment is copy-pasted and then modifications such as statement insertions and/or deletions are performed. Type 1 clones are exact clones, whereas Type 2, Type 3 and Type 4 clones are approximate clones.

In graph-based clone detection, Program Dependency Graph (PDG) is used as an intermediate source code representation, from which similar subgraphs are identified in order to detect code clones. In contrast to other clone detection techniques, PDG-based techniques are capable of detecting non-contiguous code clones because they use a program slicing based approach for detecting clones. Recent enhancements proposed by Higo and Kusumoto [2] have also improved the performance of PDG-based detection for contiguous code clones. They have implemented their proposed technique in the form of Scorpio tool. So, the current state of the art PDG-based detection identifies both contiguous and non-contiguous code clones.

The motivation of this research work is to address the scope of further improvement in existing PDG-based clone detection technique. It is motivated towards introducing a novel method to detect any new clones from the results of existing PDG-based detection technique. For biomedical text mining, various algorithms and techniques based on exact and approximate subgraph matching have been developed to extract relations and events from biomedical data [8, 9 and 10]. It is also possible to apply these algorithms and techniques to enhance the results of PDG-based code clone detection. In this paper, we propose a method to obtain both maximal and subsumed clone relations along with new approximate clone relations from PDG-based detection results by using Approximate Subgraph Matching (ASM) technique.

The contributions of this research work are: 1) we propose an algorithm which constructs the set of all possible code fragment combinations from the contiguous and non-contiguous clone pairs obtained by PDG-based detection and applies ASM on each combination of the generated set to detect new approximate clone relations, and 2) we present an approach to enumerate all possible node-to-node mappings between the code fragments of each detected clone relation. We also propose a novel ASM-based distance measure to quantify the similarity between the code fragments of each detected clone relation.

II. PRELIMINARIES

Our proposed algorithm uses Approximate Subgraph Matching (ASM) to identify new clone relations from the results of PDG-based detection. In this section, we briefly discuss PDG-based clone detection and ASM algorithm [10].

A. PDG based Clone Detection

In graph-based clone detection, PDG is used as an intermediate source code representation. PDG is a directed attributed graph that represents dependencies between program elements of the source code. A node in a PDG

represents a program element of the source code and an edge represents a dependency between two nodes. In conventional PDGs, there are two types of dependencies between nodes:

1) Control Dependency: It exists from program statements s_1 to s_2 iff: a) s_1 is a conditional predicate (if/switch statement), and; b) the execution of s_2 depends upon the execution of s_1.

2) Data Dependency: It exists from program statement s_1 to s_2 iff: a) s_1 defines *v*; *b)* s_2 references *v,* and; *c)* there exists atleast one execution path from s_1 to s_2 without any redefinition of *v*.

The Scorpio tool [1] also introduced execution next-link dependency to conventional PDGs.

B. Example for ASM

We consider a pair of graphs G_1 and G_2 shown in Fig. 1 to be tested for ASM matching. The inputs provided to ASM are: {G_1, G_2, *distanceThreshold*, *distanceWeights*}, where *distanceThreshold* is the threshold of subgraph matching distance and *distanceWeights* is an array consisting of three values: *structureWeight* (weight of the structural subgraph distance) denoted as w_s, *labelWeight* (weight of the label subgraph distance) denoted as w_l, and *directionalityWeight* (weight of the directionality subgraph distance) denoted as w_d, which specify the allowed approximation in matching [10]. There are two approximate matching occurrences of G_1 in G_2. The first instance which is an exact match, is marked by dark solid connecting edges. The second instance is an approximate match and is marked by dashed connecting edges. It has variations with respect to G_1 in the edge directionality of its constituent edge labelled C. We have used number notation to label graph vertices and alphabetic notation to label edges. G_1 is the graph whose approximate subgraph matching instances are to be identified from G_2. Vertex 1 is assigned to be the start vertex of G_1 to perform ASM matching. Start vertex is not assigned for G_2, since more than one matching instance of G_1 could be present in G_2. ASM performs the following steps:

- It computes shortest distance paths and corresponding shortest distance values for each pair of vertices in the subgraph (G_1) by applying Dijkstra Algorithm. The shortest distance information is used to compute differences in the structure, vertex labels and edge directionality between the input graphs [10].
- As shown in Table I, a map called *Injective Matches* is constructed which maps each vertex in G_1 to all possible matching vertices in G_2.
- All combinations of node mappings are obtained from *Injective Matches*, from which a list of matching vertex maps called *Candidate Matchings* is constructed as shown in Table II.
- For each map in Candidate Matchings list, subgraph matching distance function - subgraphDistance (G_1,G_2) is computed according to (1). If the computed distance value satisfies (2), then the map is said to be an approximate matching occurrence of G_1 in G_2.

$$subgraphDistance(G_1,G_2) = w_s * structDist(G_1,G_2) + w_l * labelDist(G_1,G_2) + w_d * directionalityDist(G_1,G_2) \quad (1)$$

$$0 \leq \text{subgraphDistance}(G_1,G_2) \leq distanceThreshold \quad (2)$$

[1] https://github.com/YoshikiHigo/TinyPDG

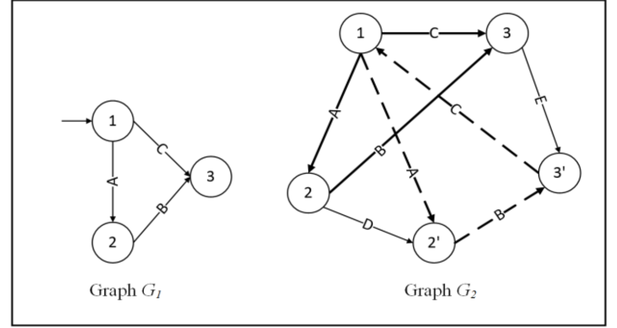

Fig. 1. Example Representation - Approximate Subgraph Matching(G_1, G_2)

TABLE I. INJECTIVE MATCHES (G_1,G_2)

G_1 Vertex	Matching G_2 Vertices
1	1
2	2, 2'
3	3, 3'

TABLE II. CANDIDATE MATCHINGS (G_1,G_2)

Map #	G_1 Vertex	Matching G_2 Vertex	Map #	G_1 Vertex	Matching G_2 Vertex
1)	1	1	3)	1	1
	2	2		2	2'
	3	3		3	3'
2)	1	1	4)	1	1
	2	2'		2	2
	3	3		3	3'

- Similarly, all matching occurrences and their corresponding distance values are identified.

III. Related Work

Komondoor and Horwitz [3] proposed the use of both forward and backward slicing on PDGs for procedures in a software system to identify isomorphic subgraphs that represent code clones. The method can detect non-contiguous clones, intertwined clones and clones in which matching code statements have been reordered. Krinke [4] proposed the use of fine-grained PDGs as intermediate representation of source code. Fine-grained PDG is a hybrid of abstract syntax tree (AST) and traditional PDG. This method identifies maximal similar k-limited subgraphs which represent code clones.

Gabel et al. [6] designed an algorithm for scalable detection of semantic clones (Type 4). This algorithm uses Deckard tool [13] to compute characteristic vectors for ASTs of the source code. For each procedure, it identifies similar PDG subgraphs which represent semantic clone candidates.

Roy and Cordy [7] proposed a method for accurate detection of near-miss clones. It identifies code clones by the use of agile parsing and TXL extract function. Flexible pretty printing, code normalization and code filtering remove insignificant differences in code to enable maximum detection of clones. Jia et al. [5] used a hybrid approach to detect both contiguous and non-contiguous code clones. They applied modified string suffix algorithm to detect Basic Clone Pairs (BCPs) which are enlarged to obtain Type 1 and Type 2 clone pairs. The BCPs are extended using lexical and program dependence analysis to obtain Type 3 clones.

To overcome the limitations of PDG-based techniques such as lower performance in the detection of contiguous code clones and large running time, Higo and Kusumoto [2]

proposed PDG specializations and heuristics to enhance the clone detection process. Higo et al. also proposed an incremental clone detection technique inspired by similar technique in text and token based detection [14].

Ammar Hammid [16] performed changes to the technique proposed by Komondoor and Horwitz [3]. The experiment indicates that accuracy improves, given the clones are detected only within the reachable procedures. Also, clone detection should be performed by applying only backward slicing. Forward slicing should be used only to define the refactoring strategy. Murakami et al. [15] proposed a method for the detection of gapped code clones. The method inspired by Smith-Waterman algorithm used for biological sequences, is effective in detecting partial clones. Moreover, it is cost-optimized unlike AST or PDG-based techniques.

Liu et al. proposed the Exact Subgraph Matching (ESM) algorithm for dependency graphs using backtracking approach [8]. It has been successfully applied for the extraction of relations and events from biomedical data such as detection of protein-residue associations and protein-protein interactions. Tian et al. developed Substructure index-based Approximate Graph Alignment (SAGA) algorithm that identifies approximate graph matching by relaxing graph differences such as node gaps, node mismatches and structural differences [9]. Liu et al. later proposed ASM algorithm for dependency graphs [10]. It identifies all approximate subgraph matching instances for a given graph pair based on the inputs - subgraph distance threshold and weight values which define the allowable difference in the graphs in terms of labels, structure and directionality.

IV. Proposed Approach and Algorithm

We propose an algorithm to identify new approximate clone relations from the clone pairs obtained using PDG-based detection. The algorithm can be applied to the clone pair results obtained by using any PDG-based detection tool. For a given subject system, the algorithm obtains all code fragment combinations from the clone pairs detected by PDG-based detection. For each combination, it constructs a pair of directed graphs and applies Approximate Subgraph Matching (ASM) [10] to identify approximate subgraph matching instances in the generated graph pair. On the basis of ASM results, the algorithm detects new approximate clone relations.

In our experimental work, the initial clone pair results for a subject system were obtained by applying the Scorpio tool. Each clone pair is a pair of code fragments represented as left code fragment (*lcf*) and right code fragment (*rcf*) which contain matching program elements (and matching PDG nodes). We have implemented the algorithm in Java and obtained the results for each subject system. A brief description of the algorithm is as follows:

In lines 1 – 6 of the algorithm, all input clone pairs are put into a list *Combinations*. This has been done because the clone pairs already identified by PDG-based detection need not be tested for approximate subgraph matching. In lines 7 – 48 of the algorithm, following Steps *i* – *viii* are performed on each code fragment combination.

i. Create an array list of PDG nodes *nodesG1* for the first code fragment in the combination.

ii. If there exists any method entry type node in *nodesG1*, set *flag1=1* and store its label in *startNodeTextG1*.

iii. Create an array list of PDG edges *edgesG1* which connect the nodes in list *nodesG1*.

iv. Perform following steps to construct directed graph *G1*:

- For each node, compute hash value from its core type (program element type) using the same method as in Scorpio [2]. Corresponding to the node, add new vertex into *G1*.
- For each edge, identify source and destination vertices in *G1*. Compute hash value from edge type. Add new edge into *G1*.

v. Repeat steps *i* - *iv* for second code fragment.

vi. Find if the two code fragments are identical. Next, check if the combination has already been tested. In any such case, skip the combination. Else, add it to Combinations list.

vii. If the combination is added to *Combinations* list and the vertex counts (VC) of graphs *G1*, *G2* satisfy (3) then proceed further, else skip to next combination i.e. step *i* above.

$$|VC(G_1) - VC(G_2)| \leq \lfloor (p/100) * \max(VC(G_1), VC(G_2)) \rfloor \quad (3)$$

viii. If vertex count of *G1* is less than or equal to that of *G2* (*G1* is to be considered as subgraph and its matching instances are to be found in *G2*), then perform following steps on (*G1*,*G2*), else on (*G2*,*G1*):

- As per values of *flag1* and *flag2*, assign *startVertex* of *G1* and *startVertices* of *G2*.
- Perform ASM matching on (*G1*, *G2*). Identify all node-to-node mappings between the graphs and store their distance values in distanceList. Also, find the minimum distance value.
- If atleast one mapping is found, then the combination has an ASM relation. If minimum distance mapping occurs at 0.0 value, then the ASM relation is a clone relation.

Line 49 of the algorithm outputs ASM and clone relations.

Algorithm:

Input: *clonepairArray* is the array of clone pairs detected by PDG-based detection tool from input subject system, *N* is the length of clonepairArray, *distanceThreshold*, *distanceWeights* is an array consisting of three values: *structureWeight*, *labelWeight*, *directionalityWeight* (as defined in Section II.B), *p* is the percentage parameter that defines which clone pairs are processed for ASM matching

Output:

- *ASMRelations* is a list of detected ASM relations
- *CloneRelations* is a list of detected clone relations

Steps:

1. //building *Combinations*[ ] list from *clonepairArray*
2. for index *i* from *0* to (*N-1*) do
3. A_i = getLeftCodeFragment(*clonepairArray*[*i*])
4. B_i = getRightCodeFragment(*clonepairArray*[*i*])
5. *C1* = CreateCombination(A_i, B_i)
6. Add *C1* to *Combinations*
7. //constructing directed graphs and applying *ASM*
8. for index *j* from *0* to (*N* – *2*) repeat step 9
9. for index *k* from (*j*+*1*) to (*N* – *1*) repeat steps 10 – 48
10. A_j = getLeftCodeFragment(*clonepairArray*[*j*])
11. A_k = getLeftCodeFragment(*clonepairArray*[*k*])
12. // Perform steps 13 – 23 to construct directed graph for A_j
13. *nodesG1* = getLeftNodes(*clonepairArray*[*j*])
14. Initialize *flag1 = 0*
15. if any node $n \in$ *nodesG1* is of type *PDGMethodEnterNode,* then set:
flag1 = 1,
startNodeTextG1 = getText(getPDGMethodEnterNode(*nodesG1*))
16. *edgesG1* = getAllConnectingEdges(*nodesG1*)
17. *G1* = createDirectedGraph( )

```
18.        for each node n in nodesG1
              nhash = computeNodeHash(core(n))
              v = createVertex(nhash, getText(n))
              Add vertex v to G1
19.        for each edge e in edgesG1
20.           // identify src and dest vertices of e
21.           ehash = computeEdgeHash(type(e))
22.           e = createEdge(src, dest, ehash)
23.           Add edge e between vertices src and dest of G1
24.        Repeat steps 13 – 23 on A_k to generate
           nodesG2, startNodeTextG2, flag2, edgesG2, G2
25.        // Perform steps 26 – 44 on (G1, flag1,
           startNodeTextG1, G2, flag2, startNodeTextG2)
26.        C2 = createCombination (A_j, A_k)
27.        Initialize toTest = 1
28.        if A_j and A_k are identical code fragments, then set toTest = 0
29.        Initialize exists = 0
30.        If C2 already exists in Combinations, set exists = 1
31.        if (toTest = = 1 and exists = = 0) then add C2 to Combinations
32.        count1 = getVertexCount(G1)
33.        count2 = getVertexCount(G2)
34.        if ( |count1 - count2| ≤ (( p / 100 ) * max( count1, count2 )) and
           (toTest = = 1) and (exists = = 0)) then perform steps 35 - 44
35.           if (count1 ≤ count2), perform steps 36 – 38 on (G1, G2), else
              on (G2, G1)
36.              if (flag1 = = 0 or flag2 = = 0) then
                    startVertex = getRandomVertex(getVertices(G1))
                    startVertices = getVertices(G2)
37.              else
                    if (getLabel(v) = = startNodeTextG1) for any vertex
                    v ϵ getVertices(G1), set startVertex = v;
                    for each vertex v ϵ getVertices(G2)
                        if (getLabel(v) = = startNodeTextG2) then
                        startVertices = startVertices ∪ v
38.           asm1 = createASM(G1, startVertex, G2, startVertices,
              distanceThreshold, distanceWeights)
39.           // storing all mappings between G1, G2 and
              distance values of the mappings
40.           matching1 = performMatching(asm1)
41.           distanceList = getDistances(matching1)
42.           minDist = findMinimum(distanceList)
43.           if (matching1 is non-empty) then
                 add C2 to ASMRelations
44.           if ((matching1 is non-empty) and
              (minDist = = 0.0)) then
                 add C2 to CloneRelations
45.        B_j = getRightCodeFragment(clonepairArray[j])
46.        B_k = getRightCodeFragment(clonepairArray[k])
47.        Repeat steps 13 – 23 for B_j, B_k to generate graphs G3, G4
48.        Repeat steps 26 – 44 for graph pairs (G3, G4), (G1, G4), (G2, G3)
49.     Output ASMRelations and CloneRelations
```

The algorithm's input parameters, *distanceThreshold* and *distanceWeights,* are user-defined. For biomedical text mining, approximate subgraph matching instances for a given graph pair are found at *distanceThreshold* value as high as 10. The *distanceWeights* = {structureWeight, labelWeight, directionalityWeight} array is set as per application requirement [10]. But for clone detection purpose, we use a smaller *distanceThreshold* to allow only minimal variations within a subgraph-graph pair that is tested for ASM matching. Also, it is preferable to assign small, uniform weight values to elements in *distanceWeights*. We use *distanceThreshold* = 1 and *distanceWeights* = {1,1,1}. This selection identifies clone pairs with small differences in the corresponding subgraph-graph pair and small, uniform variations in terms of structure, node labels and edge directionality.

For applying ASM in our implementation, ASM[2] source code was modified as follows:

i. The ASM input schema was modified to [*subgraph*, *subgraphStartVertex*, *graph*, *graphStartVertices*, *subgraphDistanceThreshold*, *subgraphWeights*].

ii. A function *performMatching*() was added. It is used in step 40 of the algorithm. It performs the following steps:

- Calls function *computeSubgraphPairwiseShortestDistanceAndPaths*( ) on the subgraph parameter.
- Calls function *getApproximateSubgraphMatchingMatches*( ) to find ASM matching between *subgraph* and *graph.*

V. Experimental Study Design

This section discusses details of the subject systems on which the proposed approach was tested, tools used, and the generic methodology followed in this study.

A. Dataset

We selected three subject systems for our experimental work and analysis, the details of which are shown in Table III. We included the test system[3] of Scorpio tool as one of the subject systems for our experimental work because we used the clone pair results of Scorpio detection as the input for our proposed algorithm. So, we found it useful to test our algorithm on this system first. EIRC (Eight IRC)[4] (also used as a subject system in [11]), is the Eteria Internet Relay Chat software for Windows operating system. Eclipse-ant[5] is a widely used Java tool for software and web development. It is a constituent subject system of Bellon's benchmark [12]. This selection of subject systems enabled us to test our algorithm on systems of different sizes.

B. Tools Used

In this work, we have applied Scorpio tool to obtain initial clone pair results from each subject system. Table IV shows for each subject system, the number of clone references (clones that should be detected as per Bellon et al [12]) and the number of clone pairs detected by Scorpio. The number of clone references is available only for eclipse system because – it is a part of Bellon's benchmark [12]. Scorpio detects clone

TABLE III. Subject Systems

S. No.	Software	Short Name	Lines of Code
I	Test System of Scorpio	-	5836
II	Eight IRC – 1.0.3	EIRC	14360
II	Eclipse-ant	Eclipse	34744

TABLE IV. Number of Clone References and Number of Clones Detected by scorpio

Software	Number of Clone References	Number of Clones Detected by Scorpio
Test System of Scorpio	-	141
EIRC	-	301
Eclipse	30	653

[2] https://sourceforge.net/projects/asmalgorithm/files/asm/1.0/
[3] https://github.com/YoshikiHigo/TinyPDG/tree/master/TinyPDG/test
[4] https://sourceforge.net/projects/eirc/
[5] http://www.eclipse.org/eclipse/ant/

```
243 int getSize() {                          260 void put(final ByteVector out) {
244   int size = 0;                          261   int n = 0;
245   AnnotationWriter aw = this;            262   int size = 2;
246   while (aw != null) {                   263   AnnotationWriter aw = this;
247       size += aw.bv.length;              264   AnnotationWriter last = null;
248       aw = aw.next;                      265   while (aw != null) {
249   }                                      266       ++n;
250   return size;                           267       size += aw.bv.length;
251 }                                                  …
                                             271       aw = aw.next;
```

a) Cloned Code Fragment 1 b) Cloned Code Fragment 2

Fig. 2. Clone Pair Detected by Scorpio from its Test System

pairs in the format: f_1, s_1, e_1, f_2, s_2, e_2, g_1, g_2 as defined in [2]. An example of cloned code fragments identified by the tool from its test system is shown in Fig. 2. The clone pair detected by Scorpio for the clone code fragment pair is as follows:

{test/test017/AnnotationWriter.java, 244, 248, test/test017/AnnotationWriter.java, 261, 271, ─, {264, 266, 268, 269,270}}

According to the clone pair format used by Scorpio, the code fragment 244 – 248 in file AnnotationWriter is a clone of the code fragment 261 – 271 in the same file. There are no clone gaps in the former code fragment, whereas lines 264, 266, 268 - 270 of the latter code fragment indicate clone gap, as observable from Fig. 2.

C. Generic Methodology

The methodology framed for the experimental work is as follows. First, the Scorpio tool was applied on a particular subject system. Then, our proposed algorithm was applied on the clone pairs detected by Scorpio. The algorithm generates all possible code fragment combinations. It applies ASM on each code fragment combination. All ASM relations are detected from the code fragment combinations. The ASM relations are filtered to find those relations which have 0.0 distance node-to-node mapping. These ASM relations are the clone relations detected by our approach.

VI. Results and Discussion

A. Clone Detection Results

Table V shows the clone detection results of Scorpio tool and that of our proposed algorithm for each subject system. This subsection discusses the results obtained by the algorithm. The comparison of these results with the results of Scorpio will be discussed later in Section V.C.

The p% specified in Table V is a user-defined input parameter. It is a measure that defines the type of subgraph-graph pairs we intend to find clone relations from. For a given subject system, if we keep p% small e.g. 10%, then only the code fragment combinations for which subgraph-graph pair have comparable sizes (size difference ≤ 10%) are tested for ASM matching whereas the combinations for which subgraph - graph pairs have larger differences in vertex counts are excluded. As we increase p%, the combinations with higher variation in graph sizes are also tested for ASM matching and hence, the number of ASM relations increase. When p% is as high as 100%, we test all combinations for ASM matching, so maximum number of ASM relations is obtained. We performed our experiment for varying p% values - 10, 30, 40, 50 and 100%.

TABLE V. Clone Detection Results of Scorpio and Proposed Algorithm

Subject System	No. of Clones Detected by Scorpio	Results of Proposed Algorithm									
		No. of ASM Relations					*No. of Clone Relations*				
p%		*10*	*30*	*40*	*50*	*100*	*10*	*30*	*40*	*50*	*100*
Test System of Scorpio	141	50	158	164	284	296	43	126	130	246	252
EIRC	301	69	251	324	413	617	25	117	152	199	263
Eclipse-ant	653	156	790	1063	1216	1421	103	440	591	688	762

The proposed algorithm identifies ASM-based clone relations from the code fragment combinations whose corresponding graph pairs satisfy p% condition defined in (3). ASM matching is performed for every such combination. If any matching occurrence exists between the graphs constructed for the combination, then ASM generates all possible node-to-node mappings between the constructed graphs and calculates the subgraph distance function [10] for each mapping. The subgraph distance value for each mapping satisfies the condition specified in (2). Since, *distanceThreshold* = 1, we observe that in our results each node-to-node mapping generated for a clone relation has a distance value in the range [0.0 – 1.0]. For every code fragment combination having ASM matching occurrence, the algorithm identifies the node-to-node mapping at minimum distance. The combinations having minimum distance mapping at 0.0 value, are the new approximate clone relations detected by our approach.

For any given subject system, increasing p% automatically increases the number of detected subsumed clone relations, and hence increases the total no. of clone relations identified. So, it can be noted that detection results at a higher p% are inclusive of the results obtained at a lower p%. We recommend the selection of an optimum p% value of 30%, so that meaningful clone relations are identified i.e. the code fragment combinations for which there is large difference in vertex counts of the constructed graphs are not processed.

From the results of the test system of Scorpio, it can be observed that for any given p% value, a large percentage of ASM relations yield clone relations. Since *distanceThreshold* = 1, mainly those ASM relations are detected for which subgraph-graph pair has limited variations in structure, vertex labels and edge directionality. Such ASM relations generally have atleast one 0.0 distance mapping and are therefore, detected as clone relations. Further, it can be observed that the number of ASM relations and the number of clone relations both increase with increase in p%. But, as p% is increased beyond 50%, only a small increase occurs in both the values. This is because clone occurrences are more probable for code fragments with comparable vertex counts than otherwise.

It is also noticeable that as we increase p% from 30 to 40%, there is only a small increase in the number of ASM relations and the number of clone relations. This is because the characteristics of the clone pair dataset obtained from Scorpio's test system by Scorpio tool are such that, as we increase p% from 30 to 40%, there is only a small increase in the code fragment combinations tested for ASM matching. In contrast to this, a steep increase is observed in both the number of ASM relations and clone relations as we increase p% from 40 to 50%, again due to the nature of input dataset. Such observations vary from one subject system to another.

The results for subject software systems EIRC-1.0.3 and eclipse-ant are also shown in Table V. These systems are large in size (large number of lines of code). The clone pairs detected for each of them by Scorpio tool are also large in number. Hence, for both of these subject systems, the input dataset for our algorithm is large. At p = 10%, we obtain less number of ASM relations and clone relations in accordance with (3). For p = 30%, both the values show a large increase compared to p = 10%. When p is increased from 30 to 40%, both values increase at a normal rate. Similar trend is observed when p is changed from 40 to 50%. At p = 100%, the values are at maximum level because all possible new approximate clone relations are identified.

It can be observed from Table V, that given a particular p%, both the number of ASM relations and the number of clone relations generally grow as the sizes of subject systems increase (exception: number of clone relations at p = 10, 30 and 50% for EIRC system is lesser than the number of clone relations at respective p% values for Scorpio test system). This is because, as the system size increases, the probability of clone occurrence also increases. But, this observation is not always true because clone occurrences largely depend on the methods and techniques adopted during system development.

When we identify new clone relations from PDG-based results by applying our proposed approach, we obtain larger clone sets based on approximate matching and by identifying transitive clone relations. The purpose of identifying these larger and more informative clone sets is to better analyze clones present in a given subject system and enhance clone refactoring process. Though we also get false positives and expect a lower precision value, it is important to note that we are able to extract some new approximate clone relations that could not be identified by conventional PDG-based detection.

The computation of clone detection parameters such as recall for such large subject systems as selected in this work requires separate study and research for manually identifying correctly detected clones with respect to actual clones for e.g. as per Bellon et al. [12]. Hence, we manually validated and verified our identified clone results on the eclipse-ant system. We present an example of a clone relation detected by our method from eclipse-ant for all p% values – 10, 30, 40, 50 and 100%. The output format used is:

Output format for *ASMRelations* or *CloneRelations* entry:
{*path1*, *firstLine1*, *lastLine1*, *path2*, *firstLine2*, *lastLine2*} where:

- *path1*, *path2*: are the absolute paths of source files which contain ASM or Clone Relations
- *firstLine1*, *lastLine1*: represent start and end line no. of the related code fragment in file path1
- *firstLine2*, *lastLine2*: represent start and end line no. of the related code fragment in file path2

Clone Relation as per the above output format is:
{eclipse-ant/src/ant/taskdefs/Execute.java, 339, 350, eclipse-ant/src/ant/util/SourceFileScanner.java, 102, 143}

The cloned code fragments C_1, C_2 identified in this clone relation are shown in Figs. 3a and 3b respectively. Similar program statements have been highlighted to indicate code similarity in the detected fragments. For this clone relation, four node-to-node mappings have been found. The distance values computed corresponding to the four mappings are 0.0, 0.3201, 0.3922, 0.5044. Since, minimum distance mapping is at 0.0, this ASM relation is identified as clone relation. Fig. 4

```
339   for (int i = 0; i < env.length; i++) {
340       int pos = env[i].indexOf('=');
          ...
344       for (int j = 0; j < size; j++) {
              ...
347           }
348       osEnv.addElement(env[i]); }
349   String[] result = new String[osEnv.size()];
350   osEnv.copyInto(result);
```

a) Cloned Code Fragment C_1

```
102 for (int i=0; i< files.length; i++) {
103     String[] targets = mapper.mapFileName(files[i]);
        ...
117         targetList.setLength(0);
118     for (int j=0; !added && j<targets.length; j++) {
            ...
137     }
        ...
141}
142 String[] result = new String[v.size()];
143 v.copyInto(result);
```

b) Cloned Code Fragment C_2

Fig. 3. Cloned Code Fragments C_1, C_2 Detected from Eclipse-ant System

```
Mapping at Distance 0.0:
i++ <339> -> i++ <102>
int i = 0 <339> -> int i = 0 <102>
j++ <344> -> j++ <118>
osEnv.copyInto(result); <350> -> v.copyInto(result); <143>
Enter <333...351> -> Enter <78...144>
String[] result = new String[][]; <349> -> String[] result = new String[][]; <142>
i < env.length <339> -> i < files.length <102>
int j = 0 <344> -> int j = 0 <118>
```

Fig. 4. Node-to-Node Mapping obtained for Cloned Code Fragments C_1, C_2

shows the node-to-node mapping obtained at 0.0 distance value. The format used for representing a pair of matching nodes in a mapping is: $nl_1 <id_1> \rightarrow nl_2 <id_2>$ where nl_1, nl_2 represent node labels of the matching PDG nodes and id_1, id_2 represent the program element type identification numbers (ids) of the respective nodes (in Scorpio's PDG construction).

B. Distance Measure for Detected ASM and Clone Relations

The proposed approach also presents a new ASM-based distance measure to quantify the similarity in the detected cloned code fragments. For each ASM relation detected by the algorithm, all possible node-to-node mappings and their ASM distance values are computed. Fig. 5 is snapshot of the distance values computed for the ASM relations and clone relations detected from the test system of Scorpio tool.

C. Comparison of Results with Scorpio

The experimental results of the proposed algorithm have been compared with the results of Scorpio tool. Table V also shows this comparative analysis. The comparison has been done for results obtained from each subject system for varying p% values. It can be observed from Table V that for each subject system, a large number of ASM relations and clone relations have been identified from the clone pairs detected by Scorpio. The clone relations identified by the proposed method include some maximal clones detected by Scorpio, some subsumed clone relations derived from Scorpio's results and some new clone relations found on the basis of approximate subgraph matching.

```
k: 0    1 matchings generated  min distance: 0.0
k: 1    1 matchings generated  min distance: 0.2571428571428572
…
k: 8    2 matchings generated  min distance: 0.0
k: 9    2 matchings generated  min distance: 0.18660287081339713
...
k: 48  2 matchings generated  min distance: 0.0
k: 49  2 matchings generated  min distance: 0.0
Total no. of ASM relations: 50
Total no. of min=0.0 occurrences: 43    // Total no. of clone relation
```

Fig. 5. Distance Measure for the Detected ASM and Clone Relations

The number of clone relations detected for a given subject system is higher than the number of clones detected by Scorpio for p% more than or equal to 50%. There is an exception in case of EIRC system. But, the number of ASM relations identified is still higher compared to Scorpio. This means that a higher p% value will yield more clone relations for EIRC.

D. Advantages of PDG-Based Clone Detection using ASM

Firstly, the proposed technique can be applied to large software projects and even combinations of multiple projects to detect approximate clone relations. By observing similarity in the type of detected clone relations and identifying occurrence of transitive relations, clone sets can be constructed. Such clone sets will be larger and more informative on clones present in the system than clone sets identified by earlier clone detection techniques, because of the inclusion of approximate clone relations. Secondly, the detected clone sets can be used for data mining the information regarding most commonly occurring clone types. Thirdly, for each identified clone relation, our technique finds different node-to-node mappings and their approximate subgraph matching distances. This is useful to analyze how the clones were originally created i.e. what type of modifications would have been made after a code fragment was copy-pasted. Fourthly, the ASM-based distance measure can be used to sort clones on a priority basis for identifying clones that need to be refactored or handled on higher priority.

VII. Threats to Validity

We have tested the accuracy and performance of our proposed algorithm for subject systems of sizes upto 34 KLOC (approximately). The experiment was performed on a personal computer having Intel Core i3 processor (processing speed of 1.80 GHz) and RAM of 4.00 GB. So, we have not tested the performance for very large software systems. We have not computed precision or recall for our approach. This is because their evaluation requires manual identification of correctly detected clones of the large subject systems chosen in this study, with respect to actual clones as per Bellon et al. [12] etc. and this calls for a separate study and research. So, we have analyzed and validated our clone detection results for eclipse-ant system as illustrated in Section VI.

VIII. Conclusions and Future Work

In this research work, we have proposed a novel algorithm to detect new approximate clone relations from the clone pair results of PDG-based detection. Our algorithm uses Approximate Subgraph Matching (ASM) technique to detect new clone relations. We manually validated the obtained results on Eclipse-ant system. We have also compared our results with the results of Scorpio tool. For each clone relation, our approach identifies all possible node-to-node mappings and their ASM-based distance values. Also, a new ASM-based distance measure has been proposed to quantify similarity between the detected cloned code fragments.

We aim to research further by testing the implementation of our proposed algorithm using other approximate subgraph matching techniques and algorithms. We would perform statistical analysis and comparison of the performance of our proposed algorithm for all the selected approximate matching techniques. Also, the clone distance calculation method proposed as part of our algorithm, should be compared to other clone identification and distance calculation measures such as Euclidean similarity, Cosine similarity, Jaccard distance etc. We would evaluate the actual cost benefit gained by applying the proposed clone detection method. Also, we propose to test the algorithm on combinations of software systems to detect and analyze approximate clone relations for enhanced software development and maintenance.

References

[1] B. S. Baker, "On finding duplication and near-duplication in large software systems," *Proc. Working Conf. on Rev. Eng.*, pp. 86-95, 1995.

[2] Y. Higo, and S. Kusumoto, "Code clone detection on specialized PDGs with heuristics," *Europ. Conf. Softw. Maint. and Reeng.*, pp. 75-84, 2011.

[3] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," *Proc. Int'l Static Analysis Symposium*, pp. 40-56, 2001.

[4] J. Krinke, "Identifying similar code with program dependence graphs," *Proc. Working Conference on Reverse Engineering*, pp. 301-309, 2001.

[5] Y. Jia, D. Binkley, M. Harman, J. Krinke and M. Matsushita, "KClone: a proposed approach to fast precise code clone detection," *Proceedings 3rd International Workshop on Detection of Software Clones (IWSC)*, 2009.

[6] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," *International Conference on Software Engineering*, pp. 321-330, 2008.

[7] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," *International Conference on Program Comprehension*, pp. 172-181, 2008.

[8] H. Liu, V. Keselj, C. Blouin and K. Verspoor, "Subgraph matching-based literature mining for biomedical relations and events," *2012 AAAI Fall Symposium Series*, 2012.

[9] Y. Tian, R. C. Mceachin, C. Santos and J. M. Patel, "SAGA: a subgraph matching tool for biological graphs," *Bioinformatics*, vol. 23, no. 2, pp. 232-239, 2007.

[10] H. Liu, L. Hunter, V. Kešelj and K. Verspoor, "Approximate subgraph matching-based literature mining for biomedical events and relations," *PLoS ONE*, vol. 8, no. 4, 2013.

[11] E. Kodhai and S. Kanmani, "Method-level code clone detection through LWH (Light Weight Hybrid) approach," *Journal of Software Engineering Research and Development*, vol. 2, no. 1, 2014.

[12] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577-591, 2007.

[13] L. Jiang, G. Misherghi, Z. Su and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," *Proc. International Conference on Software Engineering*, pp. 96-105, 2007.

[14] Y. Higo, U. Yasushi, M. Nishino and S. Kusumoto, "Incremental code clone detection: A pdg-based approach," *Proceedings of The18th Working Conference on Reverse Engineering*, pp. 3-12, 2011.

[15] H. Murakami, K. Hotta, Y. Higo, H. Igaki and S. Kusumoto, "Gapped Code Clone Detection with Lightweight Source Code Analysis," *Proceedings of the 21st IEEE International Conference on Program Comprehension (ICPC)*, pp. 93–102, 2013.

[16] A. Hamid, "Detecting Refactorable Clones Using PDG and Program Slicing," *Master's Thesis*, Universiteit van Amsterdam, August 2014.