International Journal of Innovative Technology and Exploring Engineering (IJITEE)
ISSN: 2278-3075 (Online), Volume-9 Issue-4, February 2020

Detection of Syntax Similarity of Source Code using a Graph based Hybrid Technique

Babita Pathik, Meena Sharma

Abstract*: Software evolves inherently due to change requirement. The change request applied with intent to achieve the appropriate functionality of the software. This change inside the code makes some differences in previous code. Changes in somewhere in existing code may also affect some other part of the code. Our focus is on finding similarity of two codes to draw static call graph and program dependency graph which shows the dependencies and data flow among various part of the code then apply a distance metrics to find the percentage of similarity between two codes. This paper presents a dependency graph based hybrid technique (DGHT) for detection of similarity of two variations of python code. This method also includes a Machine learning technique which analyzes syntactic structure of object oriented software system. The objective is to apply the outcomes of this work on change impact analysis. The results of the framework will help to estimate actual impact set to optimize testing efforts.

***Keywords*: Actual impact set, call graph, change impact analysis, dependency graph, software evolution,**

I. INTRODUCTION

Software code change impact analysis is a necessary to minimize the efforts in software evolution. Changes in software systems are the reasons for removing bugs reported, software maintenance, updating system, enhancing functionality of system, adding some new features, enhancing the current implementation and many more. The change in any part of software can affect some other part of the source code which termed as ripple effect [1]. The basic change impact analysis (CIA) is categorized in two ways; first one is static CIA and another one is dynamic CIA. This paper presents static CIA on source code artifacts. A software system is simulated with many artifacts. In this paper we targeted source code artifact. Changes in any system software affect the code and code structure upto some extent. This effected part of the code need to be identified and analyzed for software maintenance, reducing testing efforts. Impact analysis of software code is always been a thrust area developers and researchers. So many approaches to achieve this task have been proposed by researchers on their literature for performing IA [3], [5], [6].

Revised Manuscript Received on February 28, 2020.
***** Correspondence Author
Babita Pathik*, Information Technology department, Institute of Engineering and Technology, DAVV, Indore. Email: babitapathik@gmail.com
Meena Sharma, Computer Sc. and Engg., Institute of Engineering and Technology, DAVV, Indore. Email: meena@myself.com

The approaches opted are categorized into some traditional and modern techniques. Traditional approach involves static and dynamic analysis techniques whereas new method involves Mining Software Repositories (MSR), Information Retrieval (IR), and Machine Learning (ML) etc. In our hybrid technique we first generate call and dependency graph to identify the dependencies among artifacts. A call graph also known as control flow graph is a representation of relationships among functions, methods or subroutines in a source code [5]. Dependency graph represents the dependencies among different objects. It is a kind of directed graph [4]. The dependence graph demonstrates two dependencies one is Control dependency and another is Data dependency. These relationships are statically evaluated and interpreted to through matrix. Although call graph based technique for impact analysis is not sufficient to produce result for impact set from two variation of program. So we further extend the process toward finding the similar code and identify the difference between two using Information Retrieval (IR) approach for the same. This approach is used to find the similar code and indentify the changed code that will help to analyze the impacted code.

The remaining part of the paper is organized as mentioned. In section 2 we mentioned the works related with CIA approaches. In section 3 proposed approaches is prescribed. Section 4 empirical evaluation is described. In section 5 we concluded our work.

II. RELATED WORK

Giovanni Acampora et al [7] proposed a fuzzy-based approach for plagiarism detection of source code. Their idea is independent of programming language. This novel approach is based on Fuzzy C-Means and the Adaptive-Neuro Fuzzy Inference System (ANFIS).

A novel approach for the similarity of code fragments is given by Meital Zilberstein et al [8]. They first obtained textual descriptions of code fragments and then used natural language processing techniques to set up similarity between textual descriptions and between their corresponding code fragments. Aspect oriented program is considered as an object by Syarbaini Ahmad et al [4].They have developed a technique to analyze the target code. Aspect oriented dependence flow graph is used as intermediate representation model with call graph and dependence graph for program analysis. Malcom Gethers et al [12] proposed a new approach to carry out impact analysis (IA) for the change requirement placed for source code.

The impact set is estimated using Latent Semantic Indexing (LSI). Single version of software system was taken to estimate the impact set.

The author used a dynamic analysis with information retrieval and history-based mining technique for mining software repositories techniques (MSR).

Amir M. Saeidi et al [13] used topic modeling approach for source code analysis on the basis of the information gathered from end-users, developers and architects. They called this web-based toolkit as ITMViz which supports to interpret the topic models. Stephen W. Thomas et al [15] focused on topic models suitability towards analysis of software evolution. They have identified topics and compute various metrics. In software evolution, to analyze the changes in the source code they used topic models. Erik Linstead et al [16] developed and applied an unsupervised statistical topic models LDA. The model focuses on functional parts of the source code and goes through the evolution of multiple versions of software. The technique they developed integrates feature, applies on project management and program understanding. A machine learning based approach is proposed by Lal H et al [17] to review the software system at code level. For faster code reviews they used a supervised machine learning technique with a version control system and a bug tracking system. Eijirou Kitsu et al [19] detecting the changes in program by using the history of the edit operation of source code helps to maintainers to maintain the system.

III. PROPOSED METHODOLOGY

This section describes the DGHA process for finding the similarity to static change impact analysis we are going through following steps:

1. Choose the source code artifact for analysis of two versions of program,
   Input: python code pyV.1, and pyV.2
2. An object oriented program is considered at the method level granularity then go for syntax level analysis.
   Class -> Method -> Syntax
3. To extract the relationship among the component of selected source code we use call graph technique. Call graph indicates the dependencies between two objects of code if it exits.
   Callgraph(pyV.1) and Callgraph(pyV.2)
4. If any change affects some object (dependent artifacts), then it is very much possible that other object that dependent on them may also get affected. The change impact analysis observes these relationships dependent artifacts in source code.

4.1 Direct Impact:

D(c1) → D(c2) : Entity c2 depends on entity c1, a change in c1 propagates in c2

4.2 Indirect Impact:

If D(c1) → D(c2) & D(c2) → D(c3) : then

D (c1) → D (c3)

Entity c1 can directly impact c2, entity c2 can directly impact c3 then c1 indirectly impact the entity c3

5. This process is applied on both versions of software to analyze the similar part except these changes. The elements in the graph may be affected by the change.
6. The actual impact set apart from similarity is the part of code which needs to be tested as new test case suite.

Figure1 describing the overall process of our proposed approach.

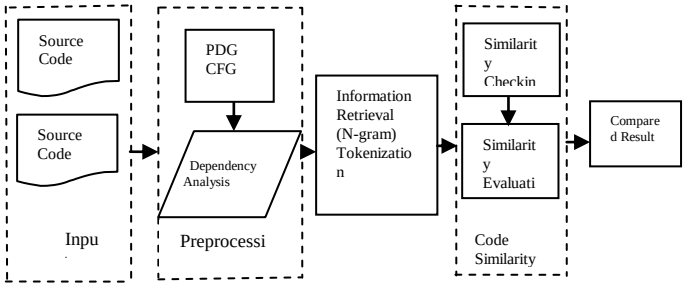

Fig.1. A flow of proposed approach

A. Call Graph

Call graphs are a medium to show the structural background of a program that can be used for human understanding. Call graphs shows the flow of execution of program and it also helps to find procedures that are never called. A static call graph is supposed to represent every possible flow of the program. Nodes of a graph are a representation of methods or functions where edge (a, b) shows that function a calling procedure b. A precise graph closely approximates the behavior of the real program if it will be interpreted properly.

B. Dependency Graph

For vertical tracing dependency graph is a suitable representation. Identifying the effect of changes by analyzing syntactic dependencies, this dependency based impact analysis techniques is useful because syntactic dependencies may cause semantic dependencies.

C. Syntax Processing

We apply n-gram approach to tokenize the syntax of code. N-gram is combination of n words which appeared in a sequence as per their occurrence. We are assuming for unigram in which each line of syntax is tokenize into words then this words are stored with frequency count using word bag of word (BoW) approach.

D. Similarity Detection

The main task of this work is to calculate similarity between two variations of the source code in terms of total percentage. For this purpose we are using Jaccard similarity metric. The Jaccard similarity used to measure the similarity between finite sample sets of object and is defined as the cardinality of the intersection of sets divided by the cardinality of the union of the sample sets. Assuming that there are two set of object A and B we want to find Jaccard similarity between these two sets A and B it is the ration of cardinality of intersection A and B (A ∩ B) and union A and B (A ∪ B).

Jaccard Similarity J(A, B) = | (A ∩ B)| / | (A ∪ B) |

Here the sets A and B are the set of words which we extract through n-gram approach.

IV. EVALUATION AND RESULTS

For empirical evaluation we have taken two version python as an example name pyV.1 and pyV.2. The first version of program is calculating salary with DA percentage shown in figure 2.

pyV.1

```
class Employee:
   empCount = 0
   def __init__(self, name, age):
      self.name = name
      self.age = age
      Employee.empCount +=1;
   def salary(self,sal):
      self.sal = sal
   def display(self):
      print("total employee", Employee.empCount)
   def displayEmp(self):
      print("Name:" ,self.name, "Age:", self.age, "Salary:",
self.sal)
class Pension(Employee):
   def __init__(self, name, age, DAper):
      Employee.__init__(self,name, age)
      self.DAper = DAper
   def salar(self):
      if self.age > 60 :
         self.salr = self.sal + self.sal*self.DAper/100
         print("pention",self.salr)
      else:
         print("not pentionable")
   def displayp(self):
      print("name of employee:",self.name , "Age is: ",
self.age ,"Per month", "DA:", self.DAper,"%")
      Pension.salar(self)
emp1 = Employee("Raj", 30)
emp2 = Employee("Kabir", 62)
e1 = Pension("Raj", 30, 40)
e2 = Pension("Kabir", 62, 40)
e1.salary(40000)
e2.salary(50000)
e1.displayEmp()
e2.displayEmp()
e1.displayp()
e2.displayp()
```

Fig. 2. Source Code for version V.1

The second version program is calculating salary with DA and house rent. The program is shown in figure 3. The part of program has modified and changed according to requirement. The correlation among the modules is interpreted through syntactic structure evaluation.

The call graph and dependency graph generation is required to check module level dependency. Figure 4 and figure 5 showing call graphs for two python code respectively Dependency graph for example code for both versions are shown in figure 6. In this graph bold line is interpreted as data dependency whereas dashed line shows the control dependency.

pyV.2

```
class Employee:
   empCount = 0
   def __init__(self, name, age):
      self.name = name
      self.age = age
      Employee.empCount +=1;
   def salary(self,sal):
      self.sal = sal
   def homea(self,hr): // changed
      self.housr = hr // changed
   def display(self):
      print("total employee", Employee.empCount)
   def displayEmp(self):
      print("Name:" ,self.name, "Age:", self.age, "Salary:",
self.sal)
class Pension(Employee):
   def __init__(self, name, age, DAper):
      Employee.__init__(self,name, age)
      self.DAper = DAper
   def salar(self):
      if self.age > 60 :
         self.salr = self.sal + self.sal*self.DAper/100
         self.tot = self.salr + self.housr  // changed
         print("Pension",self.tot) // changed
      else:
         print("not Pensionable")
   def displayp(self):
      print("name of employee:",self.name , "Age is: ",
self.age ,"Per month", "DA:", self.DAper,"%")
      Pension.salar(self)
emp1 = Employee("Raj", 30)
emp2 = Employee("Kabir", 62)
e1 = Pension("Raj", 30, 40)
e2 = Pension("Kabir", 62, 40)
e1.salary(40000)
e2.salary(50000)
e1.homea(2000)
e2.homea(3000)
e1.displayEmp()
e2.displayEmp()
e1.displayp()
e2.displayp()
```

Fig. 3. Source Code for version V.2

The statistics of unigram is given in table 1. First we apply unigram then tokenize through 2-grarm. On the same code unigram is applied and we got pair of words as token, then comparative analysis is done on basis of the outcome of 2-gram.

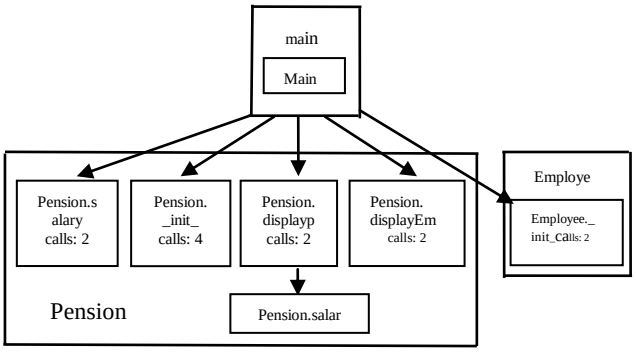

Fig. 4. Call graph representation of given Python code V.1

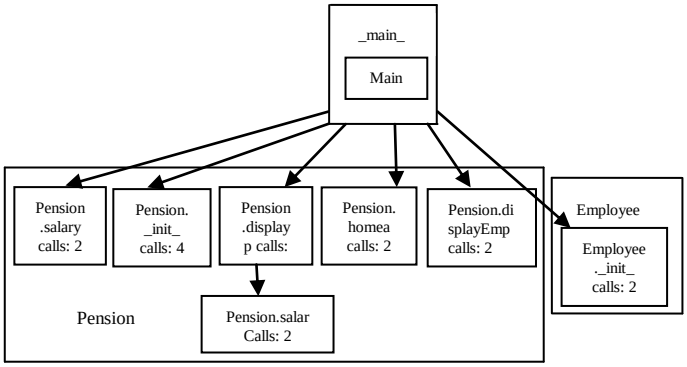

Fig. 5. Call graph representation of given Python code V.1

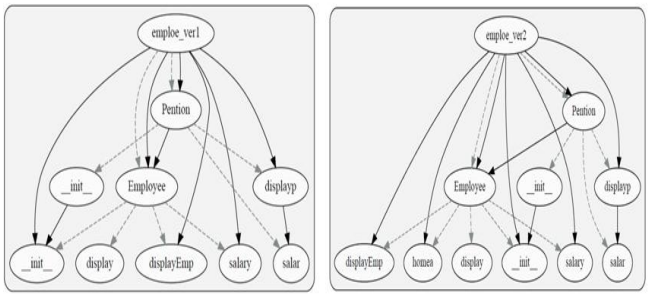

Fig. 6. Dependency graph for (a) V.1 and (b) V.2

On the same code 2-gram is applied and we got pair of words as token, then comparative analysis is done on basis of the outcome of 2-gram.

class Employee	1	class Employee	1
Employee empCount	1	Employee empCount	1

Table 1. Unigram statistics of python program (a) pyV.1 (b) pyV.2

Token	Count	Token	Count
Class	2	Class	2
Employee	4	Employee	4
empCount	1	empCount	1
=	11	=	13
Def	7	Def	8
init	2	_init_	2

In next step by applying any distance metric we can find the similarity between two source codes. For finding the similarity between set of words we have used Jaccard similarity metric.

Definition: The Jaccard similarity used to measure the similarity between finite sample sets of object and is defined as the cardinality of the intersection of sets divided by the cardinality of the union of the sample sets. Assuming that there are two set of object A and B we want to find Jaccard similarity between these two sets A and B it is the ration of cardinality of intersection A and B ($A \cap B$) and union A and B ($A \cup B$).

Jaccard Similarity $J(A, B) = |(A \cap B)| / |(A \cup B)|$

Here the sets A and B are the set of words which we extract through n-gram approach.

Overall similarity of two programs is calculated and shown in the table 2. It is assumed that 1.89 % of code is different and that is the change done in new version of the program.

Table 2: Overall similarity score of V.1 and V.2

Old Version	New Version	Similarity in %
Python code version V.1	Python code version V.2	98.11%

V. CONCLUSION

In this paper we have taken two versions of python program. A hybrid technique has been proposed. First we generate call graph and dependency graph for these two programs then we applied text processing to extract words as token. These token are then extended into bigram or 2-gram. Similarity between two set of words, one from old version of program and second from new version of program. The outcome of this metric is matched statistics. This work can be further extended by applying various version of program at different level of granularities.

REFERENCES

1. Bohner SA.: Software Change Impacts an Evolving Perspective. In Software Maintenance, IEEE Computer Society, 263-272 (2002)
2. Binkley D, Lawrie D.: Information retrieval applications in software maintenance and evolution. Encyclopedia of Software Engineering, 454-463 (2010)
3. Tanveer B. Guidelines for utilizing change impact analysis when estimating effort in agile software development. In Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering (pp. 252-257). ACM. (2017)
4. Ahmad S, Ghani AA, Sani FM. Dependence flow graph for analysis of aspect-oriented programs. International Journal of Software Engineering & Applications.;5(6):125. (2014)
5. Wang W, He Y, Li T, Zhu J, Liu J. An Integrated Model for Information Retrieval Based Change Impact Analysis. Scientific Programming. (2018)
6. Parashar P, Bhatia R, Kalia A. Change impact analysis: A tool for effective regression testing. In International Conference on Information Intelligence, Systems, Technology and Management (pp. 160-169). Springer, Berlin, Heidelberg. (2011)
7. Acampora G, Cosma G. A Fuzzy-based approach to programming language independent source-code plagiarism detection. In2015 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE) 2015 Aug 2 (pp. 1-8). IEEE.

8. Zilberstein M, Yahav E. Leveraging a corpus of natural language descriptions for program similarity. InProceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software 2016 Oct 20 (pp. 197-211). ACM.
9. Gethers, M., Kagdi, H., Dit, B., Poshyvanyk, D.: An Adaptive Approach to Impact Analysis from Change Requests to Source Code. In Proceedings of the 26th IEEE/ACM international conference on automated software engineering 540-543 (2011)
10. Zimmermann, T., Diehl, S., Zeller, A.: Mining Version Histories to Guide Software Changes. IEEE Transactions on Software Engineering, vol. 31, no. 6, 429-445 (2005)
11. Corley CS, Kashuda KL, May DS, Kraft NA. Modeling change set topics. In 4th Workshop on Mining Unstructured Data (MUD), (pp. 6-10). IEEE. (2014)
12. Gethers M, Dit B, Kagdi H, Poshyvanyk D. Integrated impact analysis for managing software changes. In 34th International Conference on Software Engineering (ICSE), (pp. 430-440). IEEE. (2012)
13. Saeidi AM, Hage J, Khadka R, Jansen S. ITMViz: interactive topic modeling for source code analysis. In Proceedings of the 23rd International Conference on Program Comprehension (pp. 295-298). IEEE Press. (2015)
14. Lukins SK, Kraft NA, Etzkorn LH. Source Code Retrieval for Bug Localization using Latent Dirichlet Allocation. In 15th Working Conference on Reverse Engineering (pp. 155-164). IEEE. (2008)
15. Thomas SW, Adams B, Hassan AE, Blostein D. Validating the use of topic models for software evolution. In 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM), (pp. 55-64). IEEE. (2010)
16. Linstead E, Lopes C, Baldi P. An application of latent Dirichlet allocation to analyzing software evolution. In Seventh International Conference on Machine Learning and Applications. ICMLA'08 (pp. 813-818). IEEE (2008)
17. Lal H, Pahwa G. Code review analysis of software system using machine learning techniques. In 11th International Conference on Intelligent Systems and Control (ISCO), (pp. 8-13). IEEE. (2017)
18. Gethers M, Oliveto R, Poshyvanyk D, De Lucia A. On integrating orthogonal information retrieval methods to improve traceability recovery. In 27th IEEE International Conference on Software Maintenance (ICSM), (pp. 133-142). IEEE. (2011)
19. Kitsu Eijirou, Takayuki Omori and Katsuhisa Maruyama. Detecting Program Changes from Edit History of Source Code. 20th Asia-Pacific Software Engineering Conference (APSEC) 1 : 299-306.IEEE (2013)

AUTHORS PROFILE

Babita Pathik is a PhD scholar in Information Technology from Institute of Engineering and Technology, DAVV, Indore (MP).

Meena Sharma is Professor in Computer Science and Engineering Department at Institute of Engineering and Technology, DAVV, Indore (MP), She has done PhD in year 2012. She has published so many research papers in reputed journal.