Mathematical Problems of Computer Science 42, 54–62, 2014.

Scalable and Accurate Clones Detection Based on Metrics for Dependence Graph [*]

Sevak S. Sargsyan, Shamil F. Kurmangaleev, Artiom V. Baloian and Hayk K. Aslanyan

Laboratory of System Programming
IT Educational and Research Centre
Yerevan State University
e-mail: sevaksargsyan@ispras.ru, kursh@ispras.ru,
artyom.baloyan@ysu.am, hayk@ispras.ru

Abstract

The article describes a new method of code clones detection for C/C++ programming languages. The method is based on metrics for program dependence graph. For every node of program dependence graph a characteristic vector is constructed, which contains information about neighbors. These characteristic vectors are represented as sixty four bit integer numbers, which allows determining similarity between two nodes in amortized constant time. Due to this it is possible to analyze effectively projects with million lines of source code. The high accuracy of the determined clones was achieved by checking locations of source code for corresponding nodes. The paper also describes new approach for dependency graphs generation, which allows building them much faster than any of the existing methods. This method was compared with several widely used tools. It performs better both execution time and accuracy.

Keywords: Metrics, Dependency graph, Scalable, Code clones, Compiler, Bit vector.

1. Introduction

Software developer can reuse the same piece of code many times. It can be done with direct copy-paste or copy-past and small modifications. Reused code can lead to many semantic errors. For example, software developer can forget to rename some variable after copy-paste. The software, which has many clones probably will have many mistakes and low quality. According to different studies [1,2] up to 20 percent of source code can be a clone in software. Clone detection tools are widely used during software development to avoid mistakes and improve code quality. Numbers of approaches were provided [3] for clones detection, but they have some restrictions. Some of them cannot detect all clone types (see 2.). Others have high computational complexity, which makes them unusable for large scale projects analysis. The goal of this paper is to introduce a metrics-based scalable and accurate algorithm of code clones detection. The metrics are defined for Program Dependence Graph (PDG). Besides the algorithm, a new approach is presented for dependency graphs generation based

[*]The paper is supported by RFBR grant 15-07-07541

on LLVM compiler [4]. It allows generating these graphs effectively, without double analyses of source code. Flexible definition of metrics and effective generation of PDGs allow us to create a scalable and accurate tool of code clones detection.

2. Background

Clone Types: There are three basic types of clones. The first type is identical to code fragments except the variations in whitespace (may be also variations in layout) and comments **(T1)**. The second type is structurally/syntactically identical to code fragments except the variations in identifiers, literals, types, layout and comments **(T2)**. The third type is copied fragments of code with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout and comments **(T3)**[5].

Code clone detection approaches: There are five basic approaches for code clone detection.

1. Methods based on textual approach consider the source code as text and try to find equal substrings [6]. These substrings are clones. When all clones are found, the clones which are located nearby can be combined into one. Basically **(T1)** type of clones is determined.
2. In case of lexical approach the source code is parsed to sequence of tokens. Then the longest common subsequence is determined. There are a few effective algorithms based on the parameterized suffix tree for clone detection [7]. One more interesting method transforms Java code to some intermediate representation and compares them instead of original source [8]. These types of algorithms can find basically **(T1)** and **(T2)** clone types.
3. The next is the syntactic approach. The algorithm works on Abstract Syntax Tree (AST). In this case the clones matched subtrees of AST. Some algorithms directly compare two ASTs to find common subtrees [9]. Another algorithm constructs vectors for AST subtrees and compares them [10]. Algorithms based on this approach find all three types of clones.
4. Metrics-based algorithms are widely used for clone detection. Algorithms based on this method, compute number of metrics for code fragment and compare them. Basically these metrics are computed for AST and Program Dependence Graph (PDG) [11]. Another method clusters computed metrics by using neural networks [12]. Metrics-based algorithms have better performance than AST or PDG comparison algorithms, but with low accuracy.
5. The last is the semantic approach. The source code is parsed to PDG. Nodes of PDG are instructions of program. Edges of PDG are dependences between the instructions. Algorithms based on PDG try to find maximal isomorphic subgraphs for pair of PDGs [13,14]. All algorithms are approximate because the maximal isomorphic subgraphs detection problem is NP hard. PDG-based methods have high accuracy but low performance.

3. Approval

Textual and lexical approaches are not very effective for detecting clones of **(T3)** type. Tree-based methods are more effective for detecting clones of **(T1)** and **(T2)** types; **(T3)** type of clones are detected with low accuracy, because the added or deleted instructions strictly change the structure of AST. Algorithms based on semantic analysis have high computational complexity, which makes them unusable for large software systems analysis. Metrics-based approaches have low accuracy. For qualitative analysis of software systems, **(T3)** as well as the other clones should be detected. So, there are two scenarios for getting all clones with high accuracy. One of them is metrics-based algorithms accuracy improvement (note: metrics should be defined for PDG). And the second one is the reduction of computational complexity of the semantic approach [15]. This work describes an effective method for PDGs generation and accurate algorithm of code clones detection based on metrics.

Widely used algorithms are based on metrics work for AST [9] or for some specialized PDG [16]. As was discussed, AST-based methods detect **(T3)** clones with low accuracy. Other methods modify PDGs to achieve high performance or make defined metrics stable for variations in code fragments. It does not solve the problem of accurate detection for **(T3)** clones. Some of them [17] bring graph isomorphism problem to tree similarity task, which can the decrease accuracy of the detected clones. Our method computes metrics for nodes of PDG, and compares them. Due to the flexible definition of the metric, the added or removed instructions have a small impact on nodes, which allows to detect all three types of clones with high accuracy.

Generation of PDGs has high computational complexity which requires much time. To reduce its cost we have purposed LLVM-based model of these graphs generation. In this model PDGs are constructed during the compilation of the project. Other methods use their own parser for PDG construction. It has a number of disadvantages. Source code should be analyzed and parsed separately. Dependences between the compilation modules should be properly processed. The large projects are not possible without the use of *Makefile*. LLVM built-in generator of PDGs allows using *Makefile* of the project and analyzing it only once, during its compilation. Comparing with other scalable PDG-based tools [17] our method allows to build these graphs much faster.

4. Dependency Graphs Generation

PDGs for the project are generated by a separate pass of LLVM (see Fig.1).

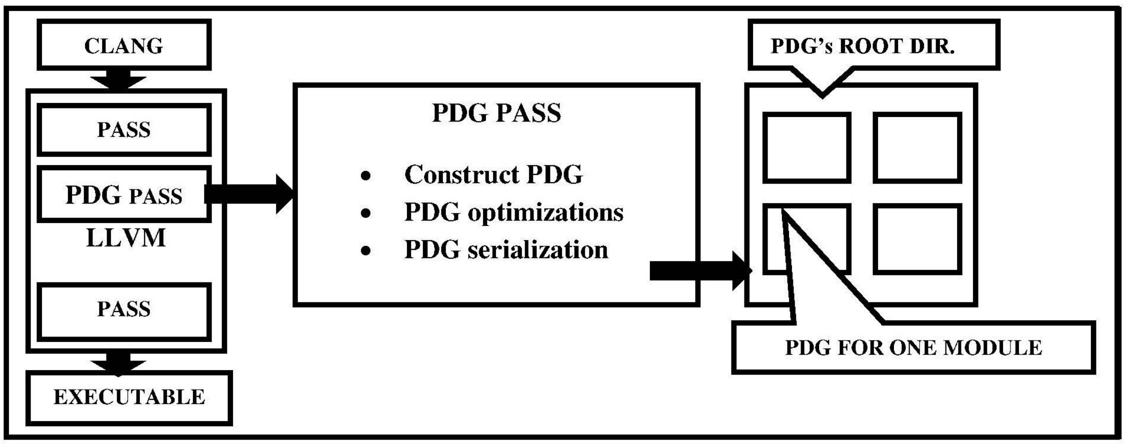

Fig. 1. LLVM-based PDG generation.

It has several advantages. Graphs are generated during the compilation time of the project. It allows to effectively construct graphs for large scale projects (up to million lines of source code). Vertices of PDG graph are LLVM bitcode [4] instructions. Edges are obtained based on LLVM use-def [4], alias and control flow analyses. Those vertices which have no edges are removed. Then the optimized PDGs are stored to a file. The stored graphs are loaded from files to memory before running the clone detection algorithm.

5. Algorithm

Bit vector (BV) of PDG's node is a vector the length of which is equal to $2 * N$, where N is the number of all possible different types of nodes. Assume that each type of node is labeled with a number from 1 to N (these labels are instruction types). The bit vector for node is initialized in the following way. For every $i = 1, ..., N$, i_{-th} position of vector is assigned to 1 if the corresponding node has an incoming edge from the node, which has the label i. Analogically, for every $j = N + 1, ..., 2 * N$, j_{-th} position of vector is assigned to 1 if the corresponding node has an outgoing edge to some node labeled as $j - N$ (see Fig. 2). Other positions are assigned to 0.

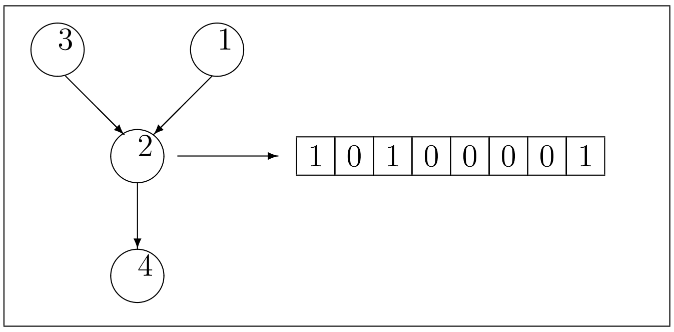

Fig. 2. Bit vector's representation.

Definition 1: *For two vectors $V1$ and $V2$ which have a length N, $V1\&V2$ is a new vector V with a length N where $V[i] = V1[i]\&V2[i]$ ($1\&1 = 1$, otherwise 0), $1 \leq i \leq N$.*

Definition 2: *For two vectors $V1$ and $V2$ which have a length N, $V1|V2$ is a new vector V with a length N where $V[i] = V1[i]||V2[i]$ ($0|0 = 0$, otherwise 1), $1 \leq i \leq N$.*

Definition 3: *For two vectors $V1$ and $V2$ with equal lengths, and with elements 0 or 1, $andC(V1, V2)$ is a number of 1 in $V1\&V2$ vector.*

Definition 4: *For two vectors $V1$ and $V2$ with equal lengths, and with elements 0 or 1, $orC(V1, V2)$ is a number of 1 in $V1|V2$ vector.*

Definition 5: *The similarity for two vectors $V1$ and $V2$ with equal lengths, and with elements 0 or 1 is a number $1 - \frac{(orC(V1,V2) - andC(V1,V2))}{(1+orC(V1,V2))}$.*

Definition 6: *Density for set of nodes of PDG is $\frac{|S|}{(max(S) - min(S))}$ where S is a set of nodes of PDG sorted by the corresponding source code lines of these nodes, $max(S)$ is a corresponding source code line for maximal node of S, $min(S)$ is a corresponding source code line for minimal node of S.*

The first stage of algorithm constructs two sets of similar nodes based on BV(bit vector) for the corresponding graphs. Every PDG node has information about the source code line from which it was constructed. The second stage of algorithm eliminates the nodes from the constructed sets. Node is removed if its corresponding source code line is located far from the other nodes source code lines.

Description of the Metrics-Based Code Clone Detector (MBCCD) algorithm:

1. Input: $G1$ and $G2$ PDG graphs, S similarity level and CL minimal length of clone.
2. Construct $C1$ and $C2$ multi maps (key is BV, value is PDG node) of BV (bit vector) for $G1$ and $G2$ nodes.
3. Any element n_1 from $C1$ is removed, if there is no such n_2 from $C2$, that similarity of the corresponding BVs of n_1 and n_2 is higher than S. The same is done for $C2$.
4. Construct $S1$ and $S2$ sets of PDG's nodes for the corresponding $C1$ and $C2$. Sort the sets by the corresponding lines numbers of source code for nodes.
5. Consider the first element is fe from $S1$ and the last element is le from $S1$. If $Density(S1 \setminus \{fe\}) > Density(S1 \setminus \{le\})$ then remove le from $S1$, otherwise remove fe. Repeat the process until $Density(S1)$ becomes higher than S. The same is done for $S2$.
6. If $S1$ and $S2$ have more elements than CL, this pair of sets as clones are printed.

6. Complexity

A few optimizations were applied to achieve high performance and make this algorithm scalable for analyzed million lines of source code. Programming languages have some limited set of operations. Usually it is less than 60. It allows representing the above described BV as a 64-bit integer number. Consequently, the complexity of two BV comparisons is constant.

For every BV of the first PDG, MBCCD algorithm examines all BVs of the second PDG, which requires $N1 * N2$ steps, where $N1$ is a number of nodes for the first graph and $N2$ is a number of nodes for the second graph. Complexity of the MBCCD algorithm is $O(N1 * N2)$.

7. Results

Comparison: The described method was compared with two widely used tools. The first one is MOSS [18]. It has been developed for detecting a plagiarism in programming classes (Stanford University). The second one is CloneDR [19]. It was developed by Semantic Designs Company, which provides different tools for software design and analysis. Test suites are described in the Table 1. The first test (Original Code) was modified in different ways to obtain all three types of clones. Article [5] contains more details for all tests. Theoretically all files are clones, because they were obtained by modification of one test. Clone detection tool with high accuracy should determine as much clones as possible.

Table 1. Test suites.

copy00.cpp: Original Code.	**copy01.cpp:** spaces are changed with tabs in line 8 and 9.
	copy02.cpp: comments are added in line 6 and 9.
1: void foo(float sum, float prod) {	**copy03.cpp:** variables ”sum” and ”prod” are renamed to ”s” and ”p”.
2: float result = sum + prod;	**copy04.cpp:** arguments of ”foo” are exchanged, line 10.
3: }	**copy05.cpp:** type of ”sum” and ”prod” are changed into int, line 5 and 6.
4: void sumProd(int n) {	**copy06.cpp:** i is exchanged into (i*i), line 8 and 9.
5: float sum = 0.0; //C1	**copy07.cpp:** lines 5 and 6 are exchanged.
6: float prod = 1.0;	**copy08.cpp:** lines 8 and 9 are exchanged.
7: for (int i = 1; i<=n; i++) {	**copy09.cpp:** lines 9 and 10 are exchanged.
8: sum = sum + i;	**copy10.cpp:** ”for” replace with ”while”.
9: prod = prod * i;	**copy11.cpp:** condition (if(i%2)) is added after line 7.
10: foo(sum, prod);	**copy12.cpp:** instruction in line 9 is deleted.
11: }	**copy13.cpp:** condition (if(i%2)) is added after line 9.
12: }	**copy14.cpp:** default value is added for the second argument for ”foo” function.
	copy15.cpp: extra argument is added for ”foo” function.

Table 2 shows results of comparison for MOSS, CloneDR and MBCCD algorithms.

Table 2. The results of comparison: ”yes” - clone is found, ”no” - clone is not found.

Test Name	MOSS	CloneDR	MBCCD
copy00.cpp	yes	yes	yes
copy01.cpp	yes	yes	yes
copy02.cpp	yes	yes	yes
copy03.cpp	yes	yes	yes
copy04.cpp	yes	yes	yes
copy05.cpp	yes	yes	yes
copy06.cpp	yes	yes	yes
copy07.cpp	yes	yes	yes
copy08.cpp	no	no	yes
copy09.cpp	no	yes	yes
copy10.cpp	no	yes	yes
copy11.cpp	no	no	yes
copy12.cpp	yes	yes	no
copy13.cpp	yes	yes	yes
copy14.cpp	yes	yes	yes
copy15.cpp	yes	yes	yes

Run time: The MBCCD was applied to a number of widely used libraries and software systems. Tests were run on Intel Core i3 CPU 540, 8GB RAM. Table 3 shows basic results of clone detection with minimal length equal to fifteen and similarity higher than 85%.

Table 3. Test results of libraries: openssl, llvm/clang and firefox.

Test Name	Lines	PDGs	PDG gen. time	Time	clones	False
openssl	280.000	1800	43	3	16	0
llvm/clang	1.300.000	19946	396	1876	94	4
firefox	3.800.000	61643	465	2489	18	0

The second column contains the number of source code lines written in C/C++ programming languages. The third column contains the number of PDGs for the project. The fourth column contains PDGs generation time. The fifth column contains the detection time in seconds. The sixth column contains the number of detected clones. The last column presents the results of manual analysis. We have tried to run MOSS and CloneDR on the same tests, but these tests were analyzed partially (MOSS and CloneDR are not able to process dependences between the compilation modules properly). Even for these partially analyzed pieces of code MBCCD is faster and more accurate. Table 4. illustrates one of the detected clones for the openssl library.

Table 4. Illustration of clones.

openssl-1.0.1g/crypto/cast/c_enc.c	**openssl-1.0.1g/crypto/bf/bf_enc.c**
141: for (l-=8; l>=0; l-=8)	237: for (l-=8; l>=0; l-=8)
142: {	238: {
143: n2l(in,tin0);	239: n2l(in,tin0);
144: n2l(in,tin1);	240: n2l(in,tin1);
145: tin0^=tout0;	241: tin0^=tout0;
146: tin1^=tout1;	242: tin1^=tout1;
147: tin[0]=tin0;	243: tin[0]=tin0;
148: tin[1]=tin1;	244: tin[1]=tin1;
149: CAST_encrypt(tin,ks);	245: BF_encrypt(tin,schedule);
150: tout0=tin[0];	246: tout0=tin[0];
151: tout1=tin[1];	247: tout1=tin[1];
152: l2n(tout0,out);	248: l2n(tout0,out);
153: l2n(tout1,out);	249: l2n(tout1,out);
154: }	250: }

8. Conclusion

We have proposed a metrics-based method of code clones detection, which is capable to analyze million lines of source code. **(T3)** as well as other types of clones are detected. The method is used as built-in instrument for LLVM. LLVM bitcode-based construction of PDGs allows building them much faster than any existed method. This tool can analyze and compare source code quality. Semantic mistakes arising during the software development process can be detected by the compiler in early stages. It can also be used for automatic refactoring.

References

[1] B. Baker, “On finding duplication and near-duplication in large software systems”, *Proceedings of the 2nd Working Conference on Reverse Engineering*, WCRE 1995, pp. 86-95, 1995.

[2] C. K. Roy and J. R. Cordy, "An empirical study of function clones in open source software systems", *Proceedings of the 15th Working Conference on Reverse Engineering*, WCRE 2008, pp. 81-90, 2008.

[3] D. Rattana, R. Bhatiab and M. Singhc, "Software clone detection", *A systematic review, Information and Software Technology*, vol. 55, no. 7, pp. 1165-1199, 2013.

[4] [Online]. Available: http://llvm.org

[5] Ch. K. Roya , J. R. Cordya and R. Koschkeb, "Comparison and evaluation of code clone detection techniques and tools", *A qualitative approach, Science of Computer Programming*, vol.74, no. 7, pp. 470-495, 2009.

[6] S. Ducasse, M. Rieger and S. Demeyer, "A language independent approach for detecting duplicated code", *Proceedings of the 15th International Conference on Software Maintenance*, (ICSM'99), Oxford, England, UK, pp. 109-119, 1999.

[7] T.Kamiya, S.Kusumoto and K.Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code", *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654-670, 2002.

[8] R. Kaur and S. Singh, "Clone detection in software source code using operational similarity of statements", *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 3, pp. 1-5, 2014.

[9] I. Baxter, A. Yahin, L. Moura and M. Anna, "Clone detection using abstract syntax trees", *Proceedings of the 14th IEEE International Conference on Software Maintenance, IEEE Computer Society*, pp. 368-377, 1998.

[10] L.Jiang, G.Misherghi, Z.Su and S.Glondu, "DECKARD : Scalable and accurate tree-based detection of code clones", *Proceedings of the 29th International Conference on Software Engineering, (ICSE07), IEEE Computer Society*, pp. 96-105, 2007.

[11] J. Mayrand, C. Leblanc and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics", *Proceedings of the 12th International Conference on Software Maintenance, (ICSM96)*, Monterey, CA, USA, pp. 244-253, 1996.

[12] N. Davey, P. Barson, S. Field and R. Frank, "The development of a software clone detector", *International Journal of Applied Software Technology*, vol. 1, no. 3/4, pp. 219-236, 1995.

[13] R.Komondoor and S.Horwitz, "Using slicing to identify duplication in source code", *Proceedings of the 8th International Symposium on Static Analysis*, pp. 40-56, 2001.

[14] J. Krinke, "Identifying similar code with program dependence graphs", *Proceedings of the 8th Working Conference on Reverse Engineering, (WCRE 2001)*, pp. 301-309, 2001.

[15] S. Gupta and P. C. Gupta, "Literature Survey of Clone Detection Techniques", *International Journal of Computer Applications*, vol. 99, no. 3, pp. 41-44, 2014.

[16] Y. Higo and S. Kusumoto, "Code clone detection on specialized PDGs with heuristics", *Proceedings of the 15th European Conference on Software Maintenance and Reengineering* (CSMR11), Oldenburg, Germany, pp.75-84, 2011.

[17] M. Gabel, L. Jiang and Z. Su, "Scalable detection of semantic clones", *Proceedings of 30th International Conference on Software Engineering (ICSE08)*, Leipzig, Germany, pp. 321-330, 2008.

[18] [Online]. Available: http://theory.stanford.edu/∼ aiken/moss/

[19] [Online]. Available: http://www.semdesigns.com/products/clone/

Submitted 06.09.2014, accepted 27.11.2014.

Ծրագրի կախվածության գրաֆի վրա սահմանված մետրիկաներով կոդի կլոնների ընդլայնելի և ճշգրիտ որոնում

Ս. Սարգսյան, Շ. Կուրմանգալեև, Ա. Բալոյան և Հ. Ասլանյան

Ամփոփում

Հոդվածում նկարագրված է կոդի կլոնների որոնման նոր մեթոդ C/C++ ծրագրավորման լեզուների համար: Այն հիմնված է ծրագրի կախվածության գրաֆի համար սահմանված մետրիկաների վրա: Կախվածության գրաֆի ամեն մի գագաթի համար սահմանվում է բնութագրիչ վեկտոր, որը պարունակում է տվյալներ՝ հարևան գագաթների մասին: Շնորհիվ նրան, որ բնութագրիչ վեկտորները ներկայացվում են որպես վաթսունչորս բիթ պարունակող ամբողջատիպ թվեր, հնարավոր է երկու գագաթների նմանությունը պարզել հաստատուն ժամանակում: Դա թույլ է տալիս արդյունավետորեն հետազոտել միլիոնավոր տողեր պարունակող ծրագրեր: Գտնված կլոնների ճշգրտությունը ապահովվում է համապատասխան գագաթների տողերի դասավորվածության ստուգմամբ: Հոդվածում նկարագրված է նաև կախվածության գրաֆի ստացման նոր մոտեցում, որը թույլ է տալիս ստանալ այդ գրաֆները շատ ավելի արագ, քան գոյություն ունեցող մեթոդները: Նկարագրված մեթոդը համեմատվել է մի շարք լայն տարածում գտած այլ մեթոդների հետ: Արդյունքները ցույց են տալիս, որ այս մեթոդը աշխատում է ավելի արագ և ճշգրիտ:

Масштабируемый и точный инструмент поиска клонов кода на основе метрик для графа зависимостей программы

С. Саргсян, Ш. Курмангалеев, А. Балоян и А. Асланян

Аннотация

Статья описывает новый метод поиска клонов кода для языков C/C++. Он основан на метриках для графа зависимостей программы. Для каждой вершины графа строится характеристический вектор, который содержит информацию о соседних вершинах. Эти векторы представлены в виде шестидесяти четырех битных целых чисел, и это позволяет определить сходство между двумя вершинами в постоянное время. Это позволило эффективно анализировать миллионы строк исходного кода. Высокая точность найденных клонов была достигнута путем проверки местоположения исходного кода соответствующих вершин. В статье также предлагается новый подход для генерации графов зависимостей, что позволяет получить их гораздо быстрее, чем любой существующий метод. Предложенный метод был сравнен с несколькими широко используемыми методами. Результаты показали, что этот метод работает быстрее и точнее.