Towards Summarizing Program Statements in Source Code Search

Victor J. Marin
Rochester Institute of Technology
vxm4964@rit.edu

Iti Bansal
Rochester Institute of Technology
ib6355@rit.edu

Carlos R. Rivero
Rochester Institute of Technology
crr@cs.rit.edu

ABSTRACT

A common practice among programmers is to find pieces of source code using search engines. Programs retrieved by these engines are typically semantically but not necessarily syntactically similar. As a result, ranking methods are exploited to present relevant programs to users. However, due to implementation variability, users need to understand such programs. In this paper, we propose a method to group statements into clusters from a set of programs retrieved by a source code search engine. Each cluster comprises a number of program statements that have similar but not exact semantics and are pervasive. Our hypothesis is that such clusters help understand at a glance a set of semantically-related programs. We use approximate graph alignment to find correspondences among statements in two program dependence graphs that are similar with respect to their control and data flows, as well as operations they perform. We then build a graph with pairwise comparisons of program dependence graphs, and cast the problem of clustering statements as finding communities of statements that consistently align. Our evaluation using programs collected by BigCloneBench shows that clusters of statements discovered by our approach help discern implementation variations.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; **Search-based software engineering**;

KEYWORDS

Source Code Search, Program Dependence Graph, Approximate Graph Alignment, Community Detection

ACM Reference Format:
Victor J. Marin, Iti Bansal, and Carlos R. Rivero. 2020. Towards Summarizing Program Statements in, Source Code Search. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3341105.3374055

1 INTRODUCTION

With the proliferation of web-based source code repositories and question-and-answer websites, programmers typically search online for source code to perform their daily tasks [9]. Despite of the fact that this is a very popular practice, we still lack a good understanding on how these searches are usually conducted [7]. Programmers tend to search for programs that can be directly incorporated into software projects, or for reference implementations to be consulted for certain information like API calls [7, 9].

Current source code search engines allow users to retrieve relevant programs based on keywords, regular expressions, input/output samples, or code snippets [10]. These engines exploit ranking algorithms to present relevant results to the user; however, ranking source code is challenging because of the variety of retrieved programs [6]. Ranking programs that are semantically similar but syntactically diverse is usually not enough, since users still need to understand the retrieved programs in order to reutilize them [8].

We observe that there are certain program statements in the programs retrieved as a result of a given query that have similar semantics and are pervasive. These individual statements can be clustered together since they roughly have the same semantic meaning, for instance, finding a pivot in a binary search, e.g., `mid = (low+high)/2` or `mid = low+(high-low)/2`.

In this paper, we propose an automatic approach to cluster individual statements of a set of semantically-related programs [5]. We translate programs into program dependence graphs [2]. We approximately compare program dependence graphs side by side and find a correspondence between their statements (nodes). We compose a graph including all pairwise correspondences and find communities of statements that have been consistently aligned.

Our evaluation results show that clusters of statements are useful to understand a set of semantically-related programs. We rely on programs collected from real-world projects by BigCloneBench [11], which provides pairwise comparisons of programs based on token similarities. Using such information as ground truth, we show that our approach achieves high accuracy when discerning implementation variability using certain user-defined parameters. Furthermore, manually inspected clusters of statements provide a high-level overview of a set of programs using a reduced number of clusters. Finally, we train a model using discovered clusters of statements to classify other statements belonging to test programs.

Section 2 presents an overview of our approach, Section 3 describes our evaluation, and Section 4 summarizes our conclusions.

2 OUR APPROACH

Given a set of programs, our goal is the automatic detection of clusters of individual statements. Each cluster contains a number of individual program statements that perform similar (but not exact) semantics and are pervasive. We assume that the set of programs are semantically related but not necessarily syntactically, which is the case when performing a source code search. For instance, BigCloneBench [11] uses regular expressions over programs to split them into functionalities that range from well (e.g., programs

SAC '20, March 30-April 3, 2020, Brno, Czech Republic

ACM ISBN 978-1-4503-6866-7/20/03.
https://doi.org/10.1145/3341105.3374055

implementing a binary search) to loosely defined (e.g., programs that connect to a database using the JDBC driver). Instead of regular expressions, the search can be performed by other methods [10].

Clustering semantically-related statements in a set of programs is a challenging task since it involves proving semantic properties of programs, which is undecidable in general [12]. We identify two major challenges when addressing our goal as follows:

- Context. When determining whether two statements in different programs are performing the same task, we cannot simply compare them in isolation. For example, an `i++` statement in isolation has the meaning of post-incrementing `i`. However, we can leverage context to see a bigger picture of its mission, e.g., updating an index in a loop to access an array, or updating a counter every time a condition holds.
- Pervasiveness. From the many statements in a set of programs, we are interested in the prevalent ones. There may be few statements with exact same semantics that can be grouped together if their semantics are relaxed, thus increasing their pervasiveness. For example, an array of numbers can be iterated using `nextInt` or `nextLong` depending on their types, but both statements have similar semantics.

To include contextual information of statements and capture their scope more faithfully, we exploit graph representations of programs that are well-suited to capture local relationships among their entities [2]. We rely on program dependence graphs combining control and data flows, which have proved to be less sensitive to program variability and more resilient to statement reordering or control replacement than tokens or abstract syntax trees [3]. Determining whether two statements in different programs are performing the same semantic task is an undecidable problem in general. Consequently, our approximation consists of finding a correspondence from statements in one program to statements in another program so as to maximize a similarity function, which takes into account both operation labels and contextual information. On one hand, operation labels encode operations a given statement is performing, such as division, array access or API call, which are compared using a set similarity function like Jaccard. On the other hand, contextual information is captured by node embeddings (vectors) that encode the control and data flows surrounding a given statement, and are compared by means of a similarity function.

Using the previous similarity function, we find a correspondence (alignment) between the statements in a given pair of programs. This correspondence represents the statements that are most similar to each other with respect to both operation labels and contextual information. We perform the same operation for every pair of programs of interest, creating thus an alignment graph that contains all program statements and their correspondences to other statements. We cast the problem of finding pervasive statements with similar semantics as finding communities of program statements that have been consistently aligned. In the literature, there exist many graph-based structures with the goal of detecting communities of nodes in graphs. Such structures include quasi-cliques, n-cliques, n-clans, k-plexes, k-cores, f-groups, n-clubs and lambda sets; however, finding such structures in large graphs is still computationally expensive with the notable exception of k-cores [1]. We thus deem k-cores a suitable model for clustering semantically-related program statements. A k-core is a maximal subgraph of a graph in which all nodes have at least k neighbors, i.e., degree of k. While enforcing nodes in a community to strongly interact, k-cores allow for a certain relaxation compared to cliques, i.e., subsets of nodes where every two distinct nodes are adjacent.

Finally, individual statements that consistently appear across the programs under evaluation fulfilling approximately similar semantics form a cluster. Individual statements that form part of the same cluster are labeled with the same cluster number.

Table 1: Results obtained for the functionalities identified by BigCloneBench using $\alpha = 0.5$, $k_0 = 1$, $p = 0.15$ and $t = 0.5$

Id	$\lvert\mathbb{G}\rvert$	$\lvert V_N\rvert$	$\lvert E_N\rvert$	$\lvert ran\ c\rvert$	$\lvert\mathbb{S}_f\rvert$	$u(\mathbb{S}_f)$	F1
f_3	1,112	30,100	9,559,476	19	795	0.95	0.93
f_7	129	3,710	120,732	19	94	0.99	0.94
f_{10}	525	29,396	4,073,870	25	315	1.00	0.83
f_{14}	336	6,791	764,550	20	194	0.99	0.92
f_{27}	345	15,519	1,309,058	26	61	1.00	0.94

3 EVALUATION

We implemented a prototype of our approach using Java 8 and SourceDG, a publicly-available framework to build program dependence graphs directly from Java source code [4]. We relied on the programs collected by BigCloneBench from real-world projects [11]. We computed a ground truth as follows: BigCloneBench stores pairwise comparisons of Java programs as well as both token and line similarities between the pairs of programs suspected of being clones, i.e., copied-and-pasted pieces of source code with some possible modifications that are the result of reusing code in software projects. We relied on the information provided by BigCloneBench to group programs based on their similarities. BigCloneBench splits these programs into several functionalities using regular expressions, which we consider as queries to a source code search engine. We selected the following five functionalities:

- f_3: Generate secure hash.
- f_7: Sort array using bubble sort.
- f_{10}: Execute database update and (conditionally) roll back.
- f_{14}: Binary search.
- f_{27}: Call method using reflection.

Our approach relies on four parameters: α to balance between contextual information and operation labels, k_0 to discard node pairs in alignments that are very dissimilar, p as the minimum support for forming clusters of statements, and t to discard clusters with low quality, i.e., clusters that contain many statements from the same programs, which can detriment our pervasiveness requirement. Intuitively, we expect that, to effectively discover clusters of semantically-related statements, we should have a proper balance between context and labels, discard node pairs that are not necessarily very dissimilar, allow a relatively high minimum support, and consider high quality clusters only. We thus use the following parameter values: $\alpha = 0.5$, $k_0 = 1$, $p = 0.15$ and $t = 0.5$.

Table 1 presents the results for each of the functionalities analyzed, where $\lvert\mathbb{G}\rvert$ represents the number of programs in a given

Table 2: Clusters of statements detected for the binary search functionality in BigCloneBench

Id	Cluster semantics	Sample statements		
c_1	Loop	`while (low <= high)`	`while (x1 < x2)`	`while (ret == -1 && l <= r)`
c_2	Compute pivot	`mid = (low + high) / 2`	`i = x1 + (x2 - x1) / 2`	
c_3	Compare using pivot	`id = elem(mid).getId()`	`cmp = j.compareTo(thresh.get(mid))`	
c_4	Less than using pivot	`if (key < a[mid])`	`if (current < key)`	`if (cmp < 0)`
c_5	Greater than using pivot	`if (initIndex > endIndex)`	`if (idx[mid] > virtIndex)`	`if (cmp > 0)`
c_6	Equals using pivot	`if (target == values[mid])`	`if (cmpValue == 0)`	`if (midValue == key)`
c_7	Update low pointer	`low = middle + 1`		
c_8	Update high pointer	`max = currentIndex - 1`		
c_9	Element found	`return midVal`	`return get(mid)`	
c_{10}	Element not found	`return -1`	`return -(lowIdx + 1)`	`return -1 * (i + 1)`

functionality, $|V_N|$ and $|E_N|$ entail the total number of nodes and edges in the alignment graph, respectively, $|ran\ c|$ and $|\mathbb{S}_f|$ stand for the number of statement clusters discovered and clusters in the ground truth, respectively, and $u(\mathbb{S}_f)$ represents the uniqueness score that evaluates the clusters generated by our approach based on the ground truth computed from BigCloneBench. A score of 1, $u(\mathbb{S}_f) = 1$, implies that the cluster numbers found are unique, i.e., given two program groups, its cluster numbers differ at least in one. We wish to compute clusters that are unique for the different implementations in a given functionality.

We obtain a uniqueness score greater or equal than 0.95 for the five functionalities under evaluation. We manually inspected f_{14} that corresponds to programs performing binary search, and provided a semantic meaning to some of the clusters of statements discovered by our approach, which are presented in Table 2. The table also provides examples of various program statements grouped in each of the clusters. As it can be observed, our statement clusters cover different aspects of a binary search, including computing pivots, comparing with pivots, updating pointer and returning found (or not) elements. Note that the statements clustered together are diverse but have similar semantics, which is our main goal.

We investigated whether individual statements of a given input program can be automatically classified. We trained a classifier for each functionality that labels each statement in a program as belonging to a particular cluster or not belonging to any cluster. Therefore, the possible output classes equal the number of clusters in a functionality ($|\mathbb{S}_f|$) plus one rejection class accounting for "uninteresting" statements not belonging to any cluster. Column F1 in Table 1 reports the average of the F1 scores for each output class. We observe that, for closed functionalities like sorting an array using bubble sort (f_7), the respective F1 scores are notably high (0.94). Conversely, for open functionalities that appear in diverse contexts like executing database updates (f_{10}), the respective F1 scores drop (0.80). Therefore, our clusters of statements can be used to automatically classify statements in other programs implementing variations of the same closed functionality.

4 CONCLUSIONS

We presented an approach to cluster statements in a set of programs that share similar semantics and are pervasive. Our approach for automatically discovering statement clusters relies on detecting communities in the graph resulting from the pairwise alignments of program dependence graphs. Communities are mined using k-cores to provide certain guarantees that statements belonging to a cluster are semantically related, and a minimum support (cluster size) is required for a cluster to ensure pervasiveness throughout the set of programs. Our evaluation suggests that statement clusters can indeed be a useful tool for tasks involving the analysis of implementation variations, and that clusters of statements discovered in a set of programs can be successfully extrapolated to other programs, i.e., statements in test programs can be categorized as part of discovered clusters in a training set. Therefore, we argue that statement clusters are valuable for applications like functionality summarization or promoting diversity in ranking code search results.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1915404.

REFERENCES

[1] James Cheng, Yiping Ke, Shumo Chu, and M. Tamer Özsu. 2011. Efficient core decomposition in massive networks. In *ICDE*. 51–62.
[2] Susan Horwitz and Thomas W. Reps. 1992. The Use of Program Dependence Graphs in Software Engineering. In *ICSE*. 392–411.
[3] Victor J. Marin, Tobin Pereira, Srinivas Sridharan, and Carlos R. Rivero. 2017. Automated Personalized Feedback in Introductory Java Programming MOOCs. In *ICDE*. 1259–1270.
[4] Victor J. Marin and Carlos R. Rivero. 2018. Towards a framework for generating program dependence graphs from source code. In *SWAN*. 30–36.
[5] Victor J. Marin and Carlos R. Rivero. 2019. Clustering Recurrent and Semantically Cohesive Program Statements in Introductory Programming Assignments. In *CIKM*. 911–920.
[6] Lee Martie and André van der Hoek. 2015. Sameness: An Experiment in Code Search. In *MSR*. 76–87.
[7] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian G. Elbaum. 2015. How developers search for code: a case study. In *ESEC/FSE*. 191–201.
[8] Huascar Sanchez, Jim Whitehead, and Martin Schäf. 2016. Multistaging to understand: Distilling the essence of Java code examples. In *ICPC*. 1–10.
[9] Susan Elliott Sim, Medha Umarji, Sukanya Ratanotayanon, and Cristina V. Lopes. 2011. How Well Do Search Engines Support Code Retrieval on the Web? *TOSEM* 21, 1 (2011), 4:1–4:25.
[10] Kathryn T. Stolee, Sebastian G. Elbaum, and Daniel Dobos. 2014. Solving the Search for Source Code. *TOSEM* 23, 3 (2014), 26:1–26:45.
[11] Jeffrey Svajlenko and Chanchal K. Roy. 2015. Evaluating clone detection tools with BigCloneBench. In *ICSME*. 131–140.
[12] Ahmad Taherkhani. 2010. Recognizing Sorting Algorithms with the C4.5 Decision Tree Classifier. In *ICPC*. 72–75.