Review On Accelerating Source Code Analysis At Massive Scale

[1]Kalyani N. Suryawanshi, [2]Ashok N. Kamthane
[1]PG Scholar, Department of CSE, Matoshri Pratishthan Group of Institutions (M.P.G.I.) Nanded.
[2]Associate Professor, Department of CSE, Matoshri Pratishthan Group of Institutions (M.P.G.I.) Nanded.

Abstract : Ultra-large scale mining has been shown to be useful for a number of Software Engineering tasks Such as Mining Specification, Defect Prediction. Extant techniques have dedicated on leveraging distributed computing to solve this problem, but with a concomitant increase in computational resource needs. This work proposes a technique that reduces the amount of calculation performed by the ultra-large-scale source code mining task, especially those that make use of control and data flow analyses. Our key idea is to analyze the mining task to identify and remove the irrelevant portions of the source code, prior to running the mining task. We show a recognition of our insight for mining and analyzing massive collections of control flow graphs of source codes. Our evaluation using 16 established control-/data-flow analyses that are typical components of mining tasks and 7 Million CFGs shows that our technique can achieve on average a 40% decline in the task calculation time. Our case studies demonstrates the applicability of our technique to massive scale source code mining tasks.

IndexTerms - Source code analysis, Mining Software Repositories, Ultra-large-scale Mining, Data-driven Software Engineering.

I. INTRODUCTION

The primary goal of software development is to deliver high quality software in the least amount of time[3]. To achieve these goals, Software Engineers are progressively applying data mining algorithms to various software engineering tasks [1] to improve software efficiency and quality.There has recently been significant interest and success in analyzing large corpora of source code repositories to solve a broad range of software engineering problems including but not limited to defect prediction [1], discovering programming patterns [2], [3], suggesting bug fixes[4],[5], requirement implication [6], [7], [8], [9].This is because in case of syntactic duplicates, the amount of hurrying is limited by the amount of copy-and-paste code and in case of semantic duplicates, only a subset of analysis can accelerate, for instance control flow only analyses[10]. In this work, we propose a balancing technique that accelerates ultra-large-scale mining tasks without difficult additional computational resources[5].

Given a mining task and an ultra-large dataset of programs on which the mining task needs to be run, our technique first analyzes the mining task to extract information about parts of the input programs that will be relevant for the mining task, and then it uses this information to perform a pre-analysis that removes the irrelevant parts from the input programs prior to running the mining task on them. For example, if a mining task is about extracting the conditions that are checked before calling a certain API method, the relevant statements are those that contains API method invocations and the conditional statements immediate the API method invocations [4]. Our technique automatically extracts this information about the relevant statements by analyzing the mining task source code and removes all irrelevant statements from the input programs prior to running the mining task [10]. Source code mining can be performed either on the source code text or on the intermediate depictions like abstract syntax trees (ASTs) and control flow graphs (CFGs) [6],[16].

In this work, we target source code mining tasks that perform control and data flow analysis on loads of CFGs[7]. Given the source code of the mining task, we first perform a static analysis to extract a set of rules that can help to identify the relevant nodes in the CFGs. Using these rules, we perform a lightweight pre-analysis that identifies and annotates the relevant nodes in the CFGs. We then perform a reduction of the CFGs to remove irrelevant nodes[4]. Finally, we run the mining task on the compacted CFGs.Running the mining task on compacted CFGs is definite to produce the same result as running the mining task on the original CFGs, but with significantly less computational cost. Our intuition behind hurrying is that, for source code mining tasks that iterates through the source code parts several times can save the unnecessary iterations and computations on the irrelevant statements, if the target source code is optimized to contain only the relevant statements for the given mining task[5]. Source code mining can be performed either on the source code text or on the intermediate representations like abstract syntax trees (ASTs) and control flow graphs (CFGs). In this work, we target source code mining tasks that perform control and data flow analysis on millions of CFGs[11].

Given the source code of the mining task, we first perform a static analysis to extract a set of rules that can help to identify the relevant nodes in the CFGs[5]. Using these rules, we perform a lightweight pre-analysis that identifies and interprets the relevant nodes in the CFGs. We then perform a reduction of the CFGs to remove irrelevant nodes. Finally, we run the mining task on the compacted CFGs[8],[4]. Running the mining task on compressed CFGs is guaranteed to produce the same result as running the mining task on the original CFGs, but with significantly less computational cost[5],[16]. Our intuition behind acceleration is that, for source code mining tasks that iterates through the source code parts several times can save the unnecessary iterations and computations on the irrelevant statements, if the target source code is optimized to contain only the relevant statements for the given mining task[6],[12].

We evaluated our technique using a collection of 16 demonstrative control and data flow analyses that are often used in the source code mining tasks and compilers[4].We also present four case studies using the source code mining tasks drawn from prior works to show the applicability of our technique. For certain mining tasks, our technique was able to reduce the task calculation time by as much as 90%, while on average a 40% reduction is seen across our collection of 16 analyses[10].

This paper makes the following contributions:

- We present that evaluates the mining task to repeatedly extract the information about parts of the source code that are related for the mining task.
- Our valuation shows that, for few mining tasks, our technique can reduce the task calculation time by as much as 90%, while on average a 40% reduction

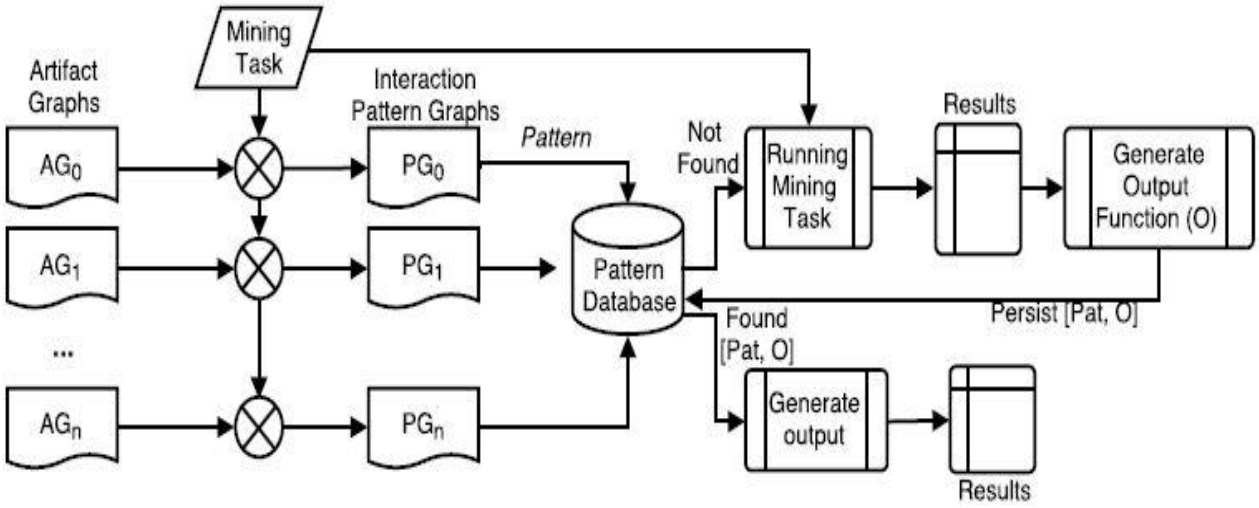

Fig. 1. Overview of the approach: accelerating ultra-large scale mining using the interaction pattern between the mining task and the artifacts.

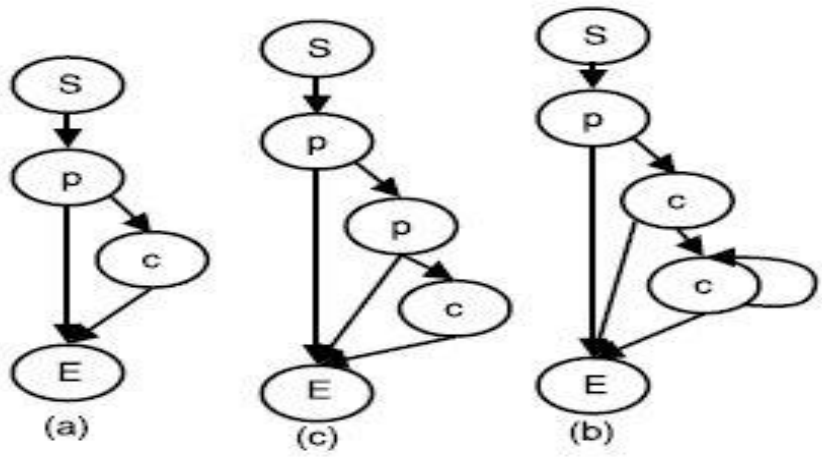

Fig. 2. Top three interaction pattern graphs for the API Precondition Mining Task. Here, S and E are start and end nodes, p is the node predicate expression, and c is the node that calls an API method.

. II. APPROACH

An overview of our approach is shown in Figure 1. Given a mining task and a large collection of artifact graphs1, alight-weight traversal is performed on each artifact graph that identifies the parts relevant for the mining task and removes the irrelevant parts to produce an interaction pattern graph2[3],[16],[17]. Upon generating the interaction pattern graph, we check if the pattern graph is already seen before while mining other piece graphs, if not then we run the mining task on the original piece graph to generate the output[16]. An output function that provides expressions to generate the output is erected.

A simple output function is the mining task itself (as we see later in the realization of this model, we use the mining task as the output function)[6],[14]. We persist the pattern along with its output function, such that next time when a match happens, we extract and apply the output function to generate the result, instead of running the mining task[16]. Many source code mining tasks require performing control and data flow analysis over CFGs of a large collection of programs.

API precondition mining [4], API usage pattern mining source code search discovering vulnerabilities to name a few. These source code mining tasks can be expensive[17]. For example, consider the API precondition mining that analyzes the control flow graphs of millions of methods to extract conditions that are checked before invoking the API methods and uses these conditions to construct specifications for API methods[10].

Let us outline our approach for mining the preconditions for API methods.

Figure 2 gives an overview, which can be summarized as:
1. The input is the set of all API methods under analysis and client projects to mine.
2. For each method in the corpus that calls an API, we build the control dependence relation between each method call and the predicates in the method (from the control-flow graph)and identify all precondition of API calls.
3. Next, we normalize the preconditions to identify and combine the equivalent ones.
4. We then analyze the preconditions to infer additional ones which are not directly present in the client code.
5. Finally we filter out non-frequent preconditions[4],[6],[16].

2.1 Mining API Preconditions

In this case study we use Mining API Preconditions, described in xII-A, that mines API preconditions of a given API method using the client methods that call the API method[16]. Here, the API preconditions are the predicate expressions that guard the API method calls in the client methods. This mining task traverses the CFG of each client method, identifies nodes that have API method calls and collect the predicate expressions on which the API method call node is control dependent using a dominator analysis[10],[16].

The mining task outputs the normalized predicate as preconditions for a given API method call. Figure 2 shows top three interaction pattern graphs that appeared in the client methods that calls the substring(int; int) API method. We argue that candidates in each cluster shows similar behaviors with respect to the mining task. Studying the candidates in each cluster may itself be a new research direction for exploring and answering mining task related questions.

For instance, we found that the interaction pattern graph shown in Figure 2(a), almost all the time provides predicate expressions that are generic to the API method and not specific to the client method that calls the API method[5],[16]. This API method invocation is guarded by the condition colon >=0 at line 6, hence the condition is considered as precondition. To extract this precondition, the mining task first builds the CFG of the method as shown in Figure 2 (CFG node numbers corresponds to the line numbers in Figure 1). The task performs several traversals of the CFG.

In the first traversal, it identifies the API method call nodes as well as the conditional nodes that provide conditions[10]. In the next traversal, the mining task performs a dominator analysis to compute a set of dominator statements for every statement in the program (A statement x dominates another statement y, if every execution of statement y includes the execution of statement x).In the final traversal, the mining task uses the results from the first two traversals to extract the conditions of all the dominating nodes of the API method invocation nodes. For this analysis the relevant nodes are:
i) the nodes that contain substring(int,int) API method invocations and ii) the conditional nodes. In the client method shown in Figure 1, only line 6 & 7 are relevant (node 6 &7 in the corresponding CFG shown in Figure 2). All other nodes are irrelevant and any computation performed on these nodes is not going to affect the mining results.

In the absence of our technique, the API requirement. mining task would traverse all the nodes in the CFG in all the three traversals, where the traversal and the computation of the irrelevant nodes (computing dominators and extracting conditions of the dominators of the API invocation nodes) can be avoided to save the unnecessary calculations. For instance, if the mining task is run on a reduced CFG as shown in Figure 2 that contains only relevant nodes, the mining task can be accelerated substantially. As we show in our case study (x5.1), for the API precondition mining task, we were able to reduce the overall mining task time by 50% [16].

2.2 *A Control Flow Graph*

Given a mining task that contains one or more traversals, our static analysis analyzes each traversal by constructing the control flow graph representation of the traversal and enumerating all acyclic paths in it **(CFG)** of a program is defined as G = (G = (N,E,T, _I) where G is a directed graph with a set of nodes N representing program statements and a set of edges E representing the control flow between statements. > and ? denote the entry and exit nodes of the CFG.2

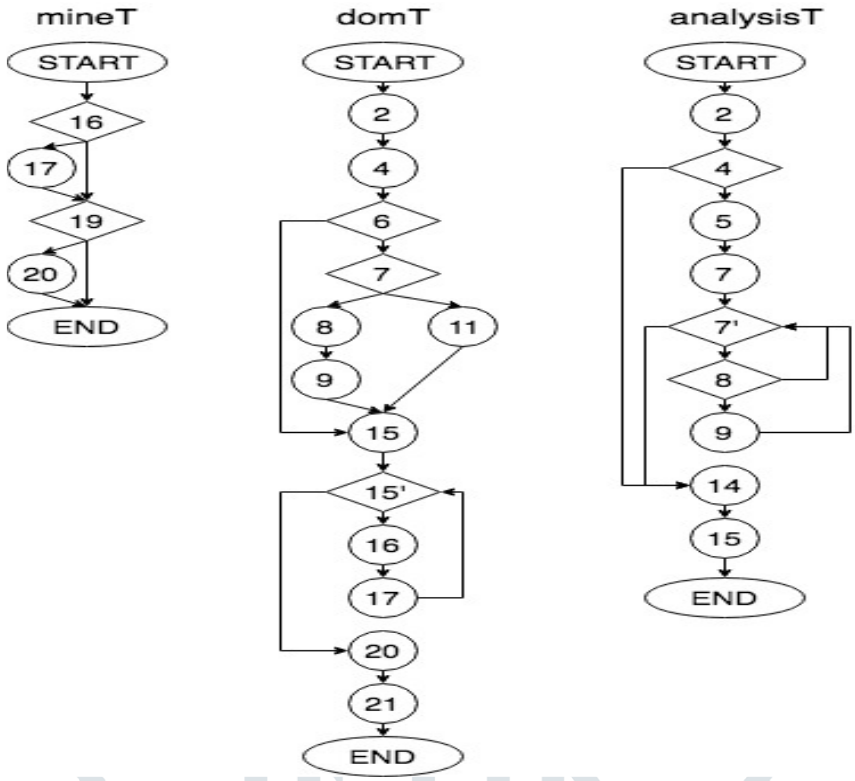

Fig. 3: Control flow graphs of the three traversals

The key idea of our static analysis is to select a subset of all acyclic paths based on the two conditions: 1) the path contains statements that provide predicates on the input variable (a CFG node), and 2) the path contains statements that contributes to the analysis output. An input variable of a traversal (or an analysis function) is always a CFG node as described in the traversal definition. For example, in the mineT traversal definition, function can be one of the three kinds:

1) the variables returned by the traversals as outputs, for instance, doms in case of domT traversal, 2) the global variables, for instance, api Call Nodes and conditions AtNodes in case of mineT traversal, or 3) the variables that are written to console as outputs, for instance, preconditions in case of analysisT traversal.

node is the input variable. Output variables of an analysis

Every selected path produces a rule that is a path condition. 3 For example, consider the API precondition mining task shown in Figure 3. This mining task contains three

traversals (mineT, domT, and analysisT), where mineT, domT, and analysisT contains 4, 6, and 4 acyclic paths respectively as shown in Figure 3. Our static analysis analyzes each of these paths to select a subset of paths satisfying the two conditions described above. Given a set of acyclic paths of a traversal (or an analysis function), and input/output variables of the traversal. A path condition is a conjunction of all node conditions and it represents a condition that must be true for the path to be taken at runtime.

III. Accelerating Software Analysis:

Kulkarni et al. [9] accelerates program analysis in Data log by running the analysis offline on a corpus of training programs to learn analysis facts over shared code and then reuses the learnt facts to accelerate the analysis of other programs that share code with the training corpus [17].

When compared to their approach, our approach does not require programs or artifacts to share code or other pieces. Recycling analysis results to accelerate interprocedural analysis by calculating partial or complete procedure summaries [10] is also studied [10]. However, to best to our awareness there is no technique that can benefit analysis crosswise programs and cluster programs specific to analysis [4],[10],[16], [17].

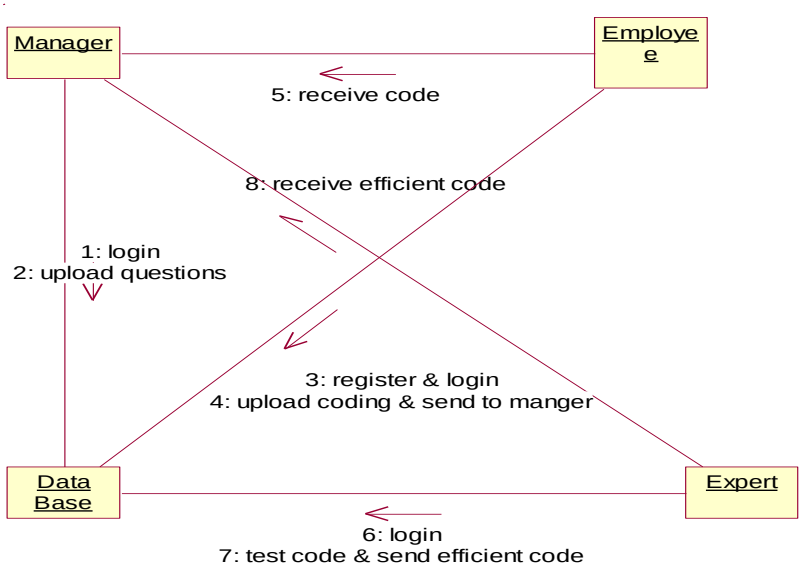

Fig.4 Collaboration Diagram

A collaboration diagram describes interactions among objects in terms of sequenced messages. Collaboration diagrams represent a combination of information taken from class, sequence, and use case diagrams describing both the static structure and dynamic behavior of a system. In this diagram first cloud user login into cloud. Here uploading his constraints to cloud provider. If a cloud want to access another cloud through Agent he need to get access from cloud admin after getting the access the Cloud can be able to view product support. Through user feedback cloud admin can monito the cloud Providers and how Agent take Place in the Interaction.

IV. CONCLUSION

Data-driven software engineering demands mining and analyzing source code repositories at massive scale, and this activity can be expensive. Extant techniques have focused on leveraging distributed computing techniques to solve this problem, but with a concomitant increase in the computational resource needs. This work proposes a complementary technique that reduces the amount of computation performed by the ultra-large-scale source code mining tasks without compromising the accuracy of the results.The key idea is to analyze the mining task to identify and remove parts of the source code that are irrelevant to the mining task prior to running the mining task. We have described a realization of our insights for mining tasks that performs control and data flow analysis at massive scale. Our evaluation using 16 classical control and data flow analyses has demonstrated substantial reduction in the mining task time. Our case studies demonstrated the applicability of our technique to massive-scale source code mining tasks.

V. REFERENCES

[1] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," Empirical Softw. Engg., vol. 17, no. 4-5, pp. 531–577, Aug. 2012.
[2] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 35–44.
[3] S. Thummalapenta and T. Xie, "Alattin: Mining alternative patterns for detecting neglected conditions," in Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 283–294.
[4] B. Livshits and T. Zimmermann, "Dynamine: Finding common error patterns by mining software revision histories," in Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 296–305.
[5] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," IEEE Trans. Softw. Eng., vol. 32, no. 3, pp. 176–192, Mar. 2006.
[6] Georgios Gousios. The GHTorrent dataset and tool suite. In MSR '13.
[7] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. Sci. Comput. Program. 2009.
[8] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan, "Mining preconditions of apis in large-scale code corpus," in Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 166–177.
[9] Sulekha Kulkarni, Ravi Mangal, Xin Zhang, and Mayur Naik. Accelerating Program Analyses by Cross-program Training. In OOPSLA'16.

[10] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, “Boa: A language and infrastructure for analyzing ultra-large-scale software repositories,” in Proceedings of the 2013 International Conference on Software Engineering, ser. ICSE ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 422–431.
[11]. S. Bajracharya, J. Ossher, and C. Lopes, “Sourcerer: An infrastructure for large-scale collection and analysis of opensource code,” Sci. Comput. Program., vol. 79, pp. 241–259, Jan. 2014.
[12] G. Gousios, “The ghtorent dataset and tool suite,” in Proceedings of the 10th Working Conference on Mining Software Repositories, ser. MSR ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 233–236.
[13] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, “Mining billions of ast nodes to study actual and potential usage of java language features,” in Proceedings of the 36th International Conference on Software Engineering, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 779–790.
[14] M. Weiser, “Program slicing,” in Proceedings of the 5th International Conference on Software Engineering, ser. ICSE ’81. Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449.
[15] John Demme and Simha Sethuma dhavan. Approximate Graph Clustering for Program Characterization. TACO’12.
[16] Ganesha Upadhyaya, Hridesh Rajan."On Accelerating Source Code Analysis at Massive Scale", IEEE Transactions on Software Engineering, 2018
[17] Ganesha Upadhyaya, Hridesh Rajan."On Accelerating ultra-large-scale mining",IEEE Transactions on Software Engineering, 2018.

.