# Towards Scalable Bug Localization using the Edit Distance of Call Traces

Themistoklis Diamantopoulos and Andreas Symeonidis
Information Technologies Institute, Centre for Research and Technology Hellas
Electrical and Computer Engineering Dept., Aristotle University of Thessaloniki
Thessaloniki, Greece
{thdiaman,asymeon}@iti.gr

***Abstract*—Locating software bugs is a difficult task, especially if they do not lead to crashes. Current research on automating non-crashing bug detection dictates collecting function call traces and representing them as graphs, and reducing the graphs before applying a subgraph mining algorithm. A ranking of potentially buggy functions is derived using frequency statistics for each node (function) in the correct and incorrect set of traces. Although most existing techniques are effective, they do not achieve scalability. To address this issue, this paper suggests reducing the graph dataset in order to isolate the graphs that are significant in localizing bugs. To this end, we propose the use of tree edit distance algorithms to identify the traces that are closer to each other, while belonging to different sets. The scalability of two proposed algorithms, an exact and a faster approximate one, is evaluated using a dataset derived from a real-world application. Finally, although the main scope of this work lies in scalability, the results indicate that there is no compromise in effectiveness.**

***Keywords*—automated debugging; dynamic bug detection; frequent subgraph mining; tree edit distance**

## I. INTRODUCTION

Software reliability has grown to be a major concern for both academia and the industry. Software bugs lead to faulty software and dissatisfied customers, as testing and debugging are quite costly even compared to the development phase. As software grows more and more complex, though, identifying and eliminating software bugs has become a challenging task.

There are two types of bugs: *crashing* and *non-crashing* ones. The former lead to program crashes, thus they are easier to locate by tracing the call stack at the time of the crash. The latter are logic errors that do not lead to crashes and thus do not produce stack traces. Since dynamic analysis is performed to detect such bugs, the field is known as *dynamic bug detection*. The techniques may be classified according to the granularity of the source code instrumentation approach. Highly granular approaches involve inserting checks in different source code positions, either in the form of counters [1] or boolean predicates [2] while others involve inserting checks at block level [3], where blocks are fragments between branches. Counter-level and block-level approaches are quite precise in localizing bugs. However, since the rise of *Object Oriented Programming* and *Functional Programming* has led to preference for small comprehensive functions, instrumenting functions is effective, as long as proper programming paradigms are employed. Function-level approaches apply *Graph Mining* techniques to call traces to identify which subgraphs are more frequent in incorrect than in correct runs [4]–[6].

The steps used to localize bugs are common. The generated call traces constitute a dataset that has to be mined in order to detect bugs; and this is where the problems start. Even at function-level, datasets are usually huge. For a small application, with, e.g., 150 functions, there may be couples of thousands of transitions among them. In this context, creating an effective, yet also scalable, solution is a challenging problem. And, though it has been broadly studied, most literature approaches focus on reducing the size of each trace, without reducing the number of traces in the dataset.

In this paper, we present a novel approach towards highly scalable Graph Mining solutions for function-level traces. The main contribution lies in the problem formulation, the reduction of the call trace dataset size through different alternatives, and the construction of a realistic dataset to test upon. Dataset size reduction is confronted using tree edit distance algorithms, while the potential benefits and drawbacks with respect to different solutions are discussed. Furthermore, the applicability of several function-level dynamic bug detection techniques in real applications is discussed and the efficiency and effectiveness of our variations are evaluated against them.

Section II of the paper reviews current literature on function-level dynamic bug detection, illustrating the general procedure followed to mine the traces and identify the Graph Mining problems. Section III provides an overview of alternative solutions to known scalability issues. The construction of a realistic dataset that illustrates our contribution is explained in section IV. Finally, our implementation is evaluated in terms of efficiency and effectiveness in section V, while section VI concludes the paper and provides insight for further research.

## II. FUNCTION-LEVEL DYNAMIC BUG DETECTION

In this section, we discuss the steps of constructing a graph dataset, reducing it, and applying Graph Mining techniques to provide the ranking of possibly buggy functions.

### A. Graph Dataset Construction

Given a set of tests, program functions are instrumented and the tests are run to produce a set of *call traces* $S$. A call trace is initially a rooted ordered tree, with the `main` function as its root. Two more sets, $S_{correct}$ and $S_{incorrect}$ are defined, corresponding to correct and incorrect executions, where correctness is determined by an oracle. Thus, the tree (or graph, since all trees are graphs) dataset is constructed.

### B. Graph Reduction

Since graphs are large, with hundreds of nodes, applying any mining algorithm is inefficient. Thus, *graph reduction* is performed to reduce the size of each graph while keeping useful information. Figure 1 depicts reduction techniques.

 ISBN: 978-1-61208-304-9 

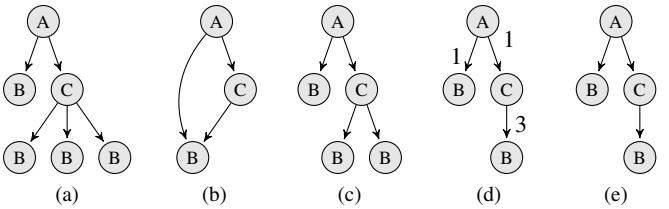



Fig. 1. An example call graph (a) and four different reduced graphs with respect to the reduction techniques, including (b) total reduction, (c) one-two-many reduction, (d) subtree reduction and (e) simple tree reduction.

The first technique, known as *total reduction*, is presented by Liu et al. [4]. The authors create a graph using each edge of the initial call graph once and discard any structural information (i.e., tree levels). Total reduction is the most efficient reduction method since it actually preserves minimum information.

However, since total reduction fails to capture the structure of the call graph, different alternatives have been applied to preserve more information, while keeping the graph as small as possible. A straightforward solution is the one proposed by Di Fatta et al. [5]; the authors perform *one-two-many* reduction, preserving tree structure by keeping two child nodes whenever the children of a node are more than two (see Figure 1c).

Eichinger et al. [6] claim that total reduction and one-two-many reduction are not sufficient, since they discard call frequency information. According to the authors, the number of times (i.e., frequency) that a function calls another function is crucial since it can capture bugs that may occur in, e.g., the third or fourth time the function is called. Thus, they propose *subtree reduction*, a technique that preserves both the structure of the tree and the frequency of function calls (see Figure 1d).

Reduction techniques are based on a compromise between information loss and scalability. Although subtree reduction maintains most information, it is quite inefficient since it adds a weight parameter to the graph. Since the scope of this work lies in scalability, we propose using a reduction technique, which we call *simple tree reduction*, shown in Figure 1e. Reducing a graph using simple tree reduction involves traversing the nodes once and deleting any duplicates as long as they are on the same level. The reduced graph is a satisfactory representation of the original one since large part of its structure is preserved.

### C. Graph Mining

Upon reduction, the problem lies in determining the nodes that are frequent in the incorrect set $S_{incorrect}$ and infrequent in the correct set $S_{correct}$. Intuitively, if a function is called every time the result is incorrect, it is highly possible to have a bug. However, having more than one function with the same frequency is also possible. Thus, the Graph Mining algorithm should find the closed frequent subgraphs, i.e., the subgraphs for which no supergraph has greater support in $S_{incorrect}$.

Finding frequent subgraphs in a graph dataset, known as *Frequent Subgraph Mining (FSM)*, is a well-known problem. State-of-the-art algorithms include, e.g., gSpan [7]. Furthermore, since these graphs are actually trees, several *Frequent Subtree Mining (FTM)* algorithms, such as FreeTreeMiner [8], may be used as well. Although those algorithms are applicable to the problem, there is strong preference for *CloseGraph* [9], an algorithm that is highly scalable since it prunes unnecessary input and outputs only closed frequent subgraphs.

### D. Ranking

The output of CloseGraph is a set of frequent subgraphs, along with their support in the correct and the incorrect set. Hence, the question is how can a ranking of possibly buggy functions be created by such a set. It is typical to use DM techniques based on *support* and *confidence* to determine the interesting subgraphs. For instance, Di Fatta et al. [5] suggest ranking the functions according to their support in the failing set. According to Eichinger et al. [6], this type of ranking can be called *structural* and for each function $f$ is defined as:

$$P_s(f) = support(f, S_{incorrect}) \tag{1}$$

The support of each function in the failing set $S_{incorrect}$ provides a fairly effective ranking. However, the scoring is not sufficient, since it does not take confidence into account. Furthermore, finding the support only on incorrect executions yields skewed results, since a function with large support in both $S_{correct}$ and $S_{incorrect}$ would be ranked high, even though it may be insignificant with respect to the bug.

Several variations of the structural ranking have emerged in order to overcome the aforementioned issues [2][5]. In this paper, we use an entropy-based ranking technique proposed by Eichinger et al. [6] since it is proven to outperform the other techniques. The main intuition behind this ranking technique is to identify the edges that are most significant to discriminate between correct and incorrect call traces. A table is created with columns corresponding to subgraph edges and rows corresponding to graphs. The table holds the support of each edge in every graph. Consider the example of Table I:

TABLE I. ENTROPY-BASED RANKING EXAMPLE

| $Graph$ | $f_1 \rightarrow f_2$ | $f_1 \rightarrow f_3$ | $f_2 \rightarrow f_4$ | $\ldots$ | $Class$ |
|---|---|---|---|---|---|
| $G_1$ | 4 | 7 | 2 | $\ldots$ | $correct$ |
| $G_2$ | 9 | 5 | 8 | $\ldots$ | $incorrect$ |
| $G_3$ | 6 | 3 | 1 | $\ldots$ | $correct$ |
| $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ |

where $F = f_1, f_2, \ldots$ is the set of functions and $G = G_1, G_2, \ldots$ is the set of graphs. Supposing subgraph $SG_1$ appears 4 times in graph $G_1$ and edge $f_1 \rightarrow f_2 \in SG_1$, the support of the edge in graph $G_1$ is 4. As one might observe, the problem is actually a *feature selection* problem, i.e., defining the features (edges) that discriminate between the values of the class feature ($correct$, $incorrect$). Thus, any feature selection algorithm may be used to determine the most significant features. Eichinger et al. [6] calculate the information gain for each feature, and interpret the result for each feature (ranging from 0 to 1) as the probability of it being responsible for a bug. The respective probability $P_e(f)$ for a node (function) is determined by the maximum probability of all the edges it is connected to.

The structural ranking $P_s$ and the entropy-based ranking $P_e$ are used to compute the combined ranking as follows:

$$P(f) = \frac{P_e(f)}{2 \max\limits_{f \in F} P_e(f)} + \frac{P_s(f)}{2 \max\limits_{f \in F} P_s(f)} \tag{2}$$

where the maximum values at the denominator are used in order to normalize the weighting of each ranking.

 ISBN: 978-1-61208-304-9 

## III. REDUCING THE GRAPH DATASET

The steps given in Section II are common for all function-level bug detection algorithms. Several researchers have indicated the need for scalability, which is generally accomplished by reducing the graphs (see subsection II-B). Ideally, the useful information of the graph is retained while its size is minimized. However, even upon reduction, the number of graphs in the dataset is large, thus making the mining step quite inefficient.

Although a dataset of several graphs is given, not all of them are equally useful in locating the bug. Consider a scenario for the `grep` program. Assume the program has a bug that results in faulty executions when the `?` character is used in a *Regular Expression (RE)*, such that the appropriate words are not returned, if the preceding element appears 0 times. Normally, if a symbol is succeeded by the `?` character, then it may be found 0 or 1 times exactly. Consider running the `grep` program for one word at a time for the following phrase:

```
there once was a cat that ate a rat
  and then it sat on a yellow mat
```

In this text, the RE `[a-z]*c?at` should match the words in the set $S_{matched} = \{$`cat`, `that`, `rat`, `sat`, `mat`$\}$, i.e., all words having any letter from `a` to `z` 0 or more times, followed by the letter `c` 0 or 1 times exactly, and followed by letters `a` and `t`. Instead it only matches the word `cat`. Consider also the set of words that are not matched $S_{unmatched} = \{$`there`, `once`, `was`, $\texttt{a}_{(1)}$, `ate`, $\texttt{a}_{(2)}$, `and`, `then`, `it`, `on`, $\texttt{a}_{(3)}$, `yellow`$\}$. Assuming that all the possible traces are created, several of them, such as the ones created from the $S_{matched}$ set, are actually much more significant in identifying the bug, since it actually resides only on the $S_{matched}$ set. Thus, traces of `cat` and `rat` should be more *similar* than traces of `cat` and `yellow`. In fact, when executing the `cat` and `rat` scenarios, many function calls coincide. This is however also true for traces of `was` and `it`. Intuitively, determining which traces are highly indicative of the bug can be based on the similarity between them as well as whether they are correct or incorrect. Thus, correct executions that are *similar* to the incorrect ones (e.g., `rat` may be close to `cat`) should isolate more easily the buggy functions. On the other hand, when two correct (or incorrect) executions are quite close to each other (e.g., the traces from `was` and `it` could be quite similar), then one of them should provide all necessary information.

The example is formed such that it is easy to understand. One could ask why not select test cases by hand, so that they are discriminating. However, this is usually impossible since real scenarios are much more complex, e.g., for the `grep` case there may be passages instead of words. In addition, certain executions may seem similar, yet be significantly different with respect to the call traces. Thus, there is the need for a similarity metric between two traces. Having such a metric, one can apply the call trace selector algorithm shown in Figure 2.

As shown in this figure, the algorithm requires as input the correct and incorrect sets, $S_{correct}$ and $S_{incorrect}$, along with parameter $n$, which controls how many graphs are going to be retained per set. Initially, the set $D$, which contains all correct-incorrect pairs of graphs, is sorted according to the similarity of each pair. The set $S'_{correct}$ contains the first $n$ correct unique graphs that are found in the sorted set $D$, i.e., the $n$ correct graphs that belong to the most similar pairs $d$ of $D$. The set $S'_{incorrect}$ contains the first $n$ incorrect unique graphs that are found in the sorted set $D$. For example, given $n = 2$ and $D = \{d_1, d_2, d_3\} = \{(g_1, g_3), (g_1, g_4), (g_2, g_5)\}$ so that the similarity of pair $d_1$ is larger than that of $d_2$ and the similarity of $d_2$ is larger than that of $d_3$, the sets $S'_{correct}$ and $S'_{incorrect}$ are $\{g_1, g_2\}$ and $\{g_3, g_4\}$ respectively. Function `sort` sorts the set according to the `key` and `index` provides the index of an element. Thus, the issue is how to determine similarity between two graphs, i.e., how to implement the function `similarity`.

```
Input: n, S_correct, S_incorrect
Output: S'_correct, S'_incorrect
D = {(g1, g2) ∀g1 ∈ S_correct, g2 ∈ S_incorrect}
sort(D, key=similarity(g1, g2))
S'_correct =First(n, {g1 : g1 ∈ d ∈ D})
S'_incorrect =First(n, {g2 : g2 ∈ d ∈ D})
```

Fig. 2. The call trace selector algorithm that receives the two sets of graphs as input (correct and incorrect) and its output is two new subsets of them.

A metric widely used to represent the similarity between two strings is the *String Edit Distance (SED)*. SED is defined as the number of edit operations required to transform one string to the other. SED operations usually contain insertion or deletion of characters. Concerning trees, such as the ones of our dataset, *Tree Edit Distance (TED)* algorithms can be used to calculate the distance between two of them. The following subsections provide a definition of the TED problem and two well known algorithms of current literature in finding TED.

### A. The Tree Edit Distance Problem

The TED problem was originally posed by Tai [10] in 1979. The possible *edit operations* are defined in Figure 3.

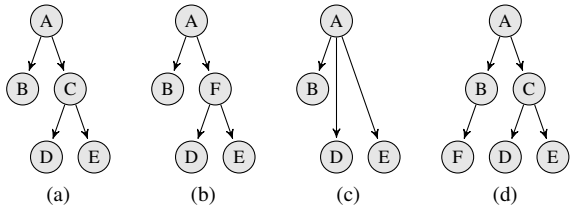



Fig. 3. An example tree (a) and three different edit operations: (b) node relabeling, (c) node deletion, and (d) node insertion.

*Node relabeling* concerns simply changing the label of a node (see Figure 3b). *Node deletion* is performed by deleting a node of the tree and reassigning any children it had so that they become children of the deleted node's parent. For example in Figure 3c, the children of deleted node `C` are reassigned to `C`'s parent `A`. Finally, *node insertion* concerns inserting a new node in a position in the tree, such as inserting node `F` in Figure 3d. Assuming a *cost function* is defined for each operation, an *edit script* between two trees $T_1$, $T_2$ is a sequence of operations required to turn $T_1$ into $T_2$, and its cost is the aggregate cost of them. Thus, the TED problem is defined as determining the *optimal edit script*, i.e., the one with the minimum cost.

### B. Zhang-Shasha Algorithm

Let $\delta(T_1, T_2)$ be the edit distance between trees $T_1$ and $T_2$, and $\gamma(l_1 \rightarrow l_2)$ be the cost of the edit operation from $l_1$ to $l_2$.

A simple algorithm for computing TED is defined as follows:

$$\delta(\theta, \theta) = 0 \quad (3)$$

$$\delta(T_1, \theta) = \delta(T_1 - u, \theta) + \gamma(u \to \lambda) \quad (4)$$

$$\delta(\theta, T_2) = \delta(\theta, T_2 - v) + \gamma(\lambda \to v) \quad (5)$$

$$\delta(T_1, T_2) = min \begin{cases} \delta(T_1 - u, T_2) + \gamma(u \to \lambda) \\ \delta(T_1, T_2 - v) + \gamma(\lambda \to v) \\ \delta(T_1(u), T_2(v)) + \delta(T_1 - T_1(u), \\ \qquad T_2 - T_2(v)) + \gamma(\lambda \to v) \end{cases} \quad (6)$$

where $T - u$ denotes tree $T$ without node $u$ and $T - T(u)$ denotes tree $T$ without $u$ or any of each children. Parameter $\lambda$ is the performed edit operation. The *Zhang-Shasha* algorithm, which was named after its authors, K. Zhang and D. Shasha [11], uses *Dynamic Programming (DP)* in order to compute the TED. The *keyroots* of a tree $T$ are defined as:

$$keyroots(T) = \{root(T)\} \cup \{u \in T : \text{u has left siblings}\} \quad (7)$$

Given (7), the *relevant subtrees* of $T$ are defined as:

$$relevant\_subtrees(T) = \bigcup_u \{T(u)\}, \forall u \in keyroots(T) \quad (8)$$

Thus, the algorithm recursively computes the TED by finding the relevant subtrees and applying equations (3)–(6).

### C. pq-Grams Algorithm

Several algorithms solve the TED problem. However, even the most efficient ones lack scalability, since the polynomial order of the problem is high. A promising way of reducing complexity is by approximating the TED instead of computing its exact value. Approximate TED algorithms can generally be effective enough when results do not need to be exact. In the call trace scenario, the TED is a value denoting the similarity of two trees, thus, even if it is approximate, it shall provide with the appropriate $n$ most significant graphs as in Figure 2.

Such an approximate TED algorithm is the *pq-Grams* based algorithm proposed by Augsten et al. [12]. The authors define $pq$-Grams as a port of known string $q$-grams to trees. An example tree and its $pq$-Grams are shown in Figure 4. The $p$ and $q$ parameters define the *stem* and the *base* of the $pq$-Gram, respectively. Let $p = 2$ and $q = 3$, the stem of the first $pq$-Gram of Figure 4c is $\{*, \texttt{A}\}$ and its base is $\{*, *, \texttt{B}\}$. Since the $pq$-Grams for the tree of Figure 4a cannot be directly created, an intermediate step of extending the tree with dummy nodes is shown in Figure 4b. The *pq-Gram profile* is the set of all $pq$-Grams of a tree (see Figure 4c), while the *pq-Gram index* of the tree is defined as the bag of all label tuples for the tree. The $pq$-Gram index for the tree of Figure 4 is:

$$I(T) = \{*A{*}{*}B, *A{*}BC, *ABC{*}, *AC{*}{*}, AB{*}{*}{*}, AC{*}{*}D, AC{*}DE, ACDE{*}, ACE{*}{*}, CD{*}{*}{*}, CE{*}{*}{*}\} \quad (9)$$

According to Augsten et al. [12], the TED between two trees is effectively approximated by the distance between their $pq$-Gram indexes. Let $I(T)$ be the $pq$-Gram index of tree $T$, the $pq$-Gram distance between trees $T_1$ and $T_2$ is defined as:

$$\delta(T_1, T_2) = |I(T_1) \cup I(T_2)| - 2|I(T_1) \cap I(T_2)| \quad (10)$$

Equation (10) provides a fast way of approximating the TED between any pair of trees of the dataset. Thus, the $pq$-Gram distance function can be used in place of the `similarity` function which is required by the algorithm shown in Figure 2.

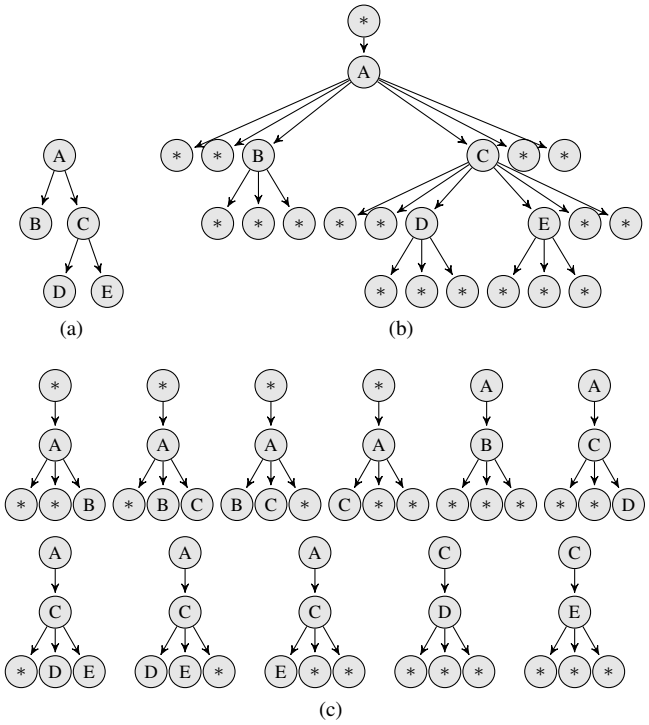



Fig. 4. A $pq$-Grams example for $p = 2$ and $q = 3$, containing (a) an example tree, (b) its extended form for $p = 2$ and $q = 3$, and (c) its $pq$-Grams.

## IV. DATASET

The techniques of section II are effective for bug localization in small applications. For example, Eichinger et al. [6] evaluate their method against two known literature bug localization techniques ([4] and [5]) using a small dataset. Although effectiveness is irrefutable, efficiency is not thoroughly tested since the dataset is too small to resemble a real application. Indicatively, the size of the program is almost 2 pages of code, leading to graphs of roughly 20 nodes after the reduction step.

Since the main scope of this paper lies in achieving scalability in order to locate bugs of real applications, a larger dataset has to be used. The dataset was generated using the source code of `daisydiff` [13], a `Java` application that compares `html` files. We used the 1.2 version of `daisydiff` and planted 3 types of bugs in the code, as shown in Table II.

TABLE II. PLANTED BUGS

| Bugs | Description | Function Calls |
|---|---|---|
| 1 | Wrong limit conditions (Forgot `+1`) | 17509 |
| 2 | Missing condition (Forgot a $<$ check) | 54137 |
| 3 | Wrong condition ($>$ instead of $<$) | 78837 |

These bugs do not aim to cover possible bug classes, as in [6], rather to test algorithm efficiency. Three scenarios with different number of function calls are created to demonstrate our proof of concept. The bug-free and the three buggy versions were run 100 times given different inputs. The application has almost 70 files with 9500 lines, leading to graphs of almost 750 nodes after reduction. The dataset is given online in [14].

 ISBN: 978-1-61208-304-9 

TABLE III. AVERAGE ELAPSED TIME (IN SECONDS) FOR THE DIFFERENT PHASES OF THE ALGORITHMS

| | pq-Grams | | | | | | | NoTED | ZhangShasha | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 5 | 10 | 15 | 20 | 25 | 30 | 35 | - | 5 | 10 | 15 | 20 | 25 | 30 | 35 |
| Graph Parsing | 7.81 | 7.15 | 7.11 | 7.15 | 7.13 | 7.12 | 7.13 | 7.14 | 8.71 | 7.05 | 7.06 | 7.02 | 7.03 | 7.02 | 7.02 |
| Graph Reduction | 4.03 | 3.97 | 4.00 | 4.04 | 4.03 | 3.96 | 3.97 | 3.93 | 4.07 | 3.98 | 3.97 | 3.94 | 3.99 | 3.92 | 3.94 |
| Dataset Reduction | 84.22 | 84.10 | 84.91 | 83.99 | 83.94 | 83.86 | 84.07 | 0.00 | 188.23 | 187.90 | 187.37 | 187.35 | 187.81 | 187.41 | 187.28 |
| Subgraph Mining | 7.55 | 27.10 | 54.69 | 131.63 | 440.51 | 412.46 | 450.87 | 4712.54 | 5.87 | 40.05 | 69.91 | 149.03 | 389.21 | 436.68 | 611.15 |
| Ranking Calculation | 0.56 | 2.25 | 6.51 | 16.62 | 34.82 | 36.47 | 45.86 | 533.59 | 0.58 | 2.77 | 7.23 | 16.33 | 33.87 | 38.31 | 55.12 |
| Total | 104.17 | 124.57 | 157.22 | 243.43 | 570.43 | 543.87 | 591.90 | 5257.20 | 207.46 | 241.75 | 275.54 | 363.67 | 621.91 | 673.34 | 864.51 |

TABLE IV. RANKING POSITION AND PERCENTAGE OF FUNCTIONS TO BE EXAMINED TO FIND THE BUGS

| | | pq-Grams | | | | | | | NoTED | ZhangShasha | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 5 | 10 | 15 | 20 | 25 | 30 | 35 | - | 5 | 10 | 15 | 20 | 25 | 30 | 35 |
| Bug 1 | Position | 7 | 7 | 31 | 9 | 8 | 8 | 8 | 8 | 7 | 6 | 9 | 8 | 9 | 8 | 8 |
| | Percentage | 1.10% | 1.10% | 4.87% | 1.41% | 1.26% | 1.26% | 1.26% | 1.26% | 1.10% | 0.94% | 1.41% | 1.26% | 1.41% | 1.26% | 1.26% |
| Bug 2 | Position | 5 | 5 | 9 | 9 | 9 | 10 | 9 | 9 | 5 | 5 | 9 | 9 | 9 | 9 | 9 |
| | Percentage | 0.68% | 0.68% | 1.22% | 1.22% | 1.22% | 1.36% | 1.22% | 1.22% | 0.68% | 0.68% | 1.22% | 1.22% | 1.22% | 1.22% | 1.22% |
| Bug 3 | Position | 105 | 105 | 341 | 27 | 3 | 1 | 15 | 17 | 105 | 337 | 342 | 352 | 1 | 1 | 1 |
| | Percentage | 13.51% | 13.51% | 43.89% | 3.47% | 0.39% | 0.13% | 1.93% | 2.19% | 13.51% | 43.37% | 44.02% | 45.30% | 0.13% | 0.13% | 0.13% |

## V. EVALUATION

This section presents the results of applying three different algorithms to the dataset described in section IV.

### A. Experimental Setup

We implemented three algorithms to test the validity of our dataset reduction hypothesis. The first is the algorithm by Eichinger et al. [6] as explained in section II. Due to performance issues, subtree reduction (see subsection II-B) could not be applied in such a large dataset. Thus, simple tree reduction is used in its place. The mining step is performed through the ParSeMiS [15] implementation of CloseGraph, while InfoGain (ranking step) was implemented using WEKA [16].

The two other algorithms (ZhangShasha and $pq$-Grams) were implemented similarly, inserting a dataset reduction step before the graph mining step. Both implementations use the call trace selector of Figure 2, while different values of the $n$ parameter are tested. The first implementation realizes the ZhangShasha algorithm and the second implementation the $pq$-Grams algorithm in order to reduce the size of the dataset.

All experiments were performed using an 8-core processor with 8 GB of memory. The graph reduction, dataset reduction and subgraph mining steps were performed in parallel. Graph reduction was performed on 8 threads, where each thread performed simple tree reduction to a fragment of the dataset. The TED algorithms were applied in parallel using 4 threads (using more threads was impossible due to memory limitations) that calculated the TED for each correct-incorrect pair of the dataset. Finally, CloseGraph was executed using 8 threads, while the trace parsing and ranking steps were sequential.

### B. Experimental Results

The algorithms are evaluated both in terms of effectiveness and performance. Concerning certain parameters, $p$ and $q$ of the $pq$-Grams approach were given the values 2 and 3 respectively, having little impact on performance and effectiveness, and CloseGraph was run with a $10\%$ support threshold.

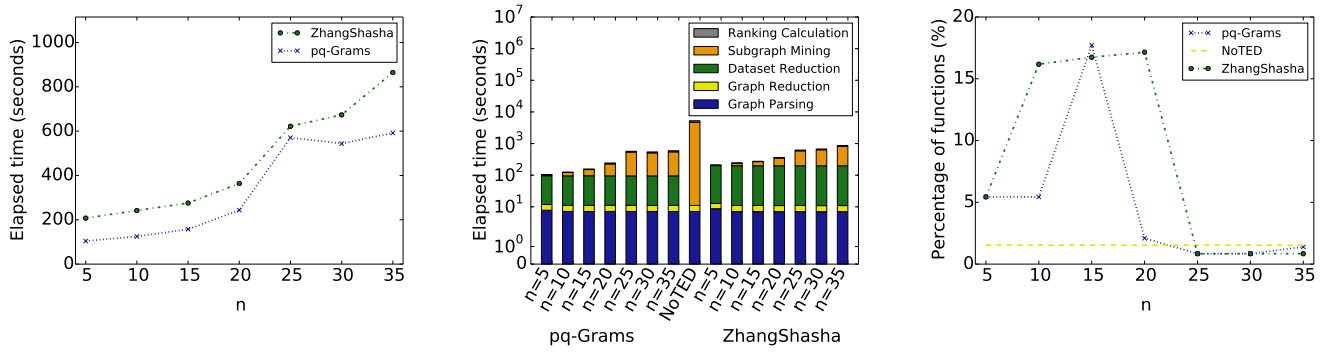
The performance results are shown in Table III, where the NoTED approach is the one not using any TED algorithm to reduce the size of the dataset. Due to space limitations in paper length, the average measurements are shown for all three bugs, instead of separate ones for each bug. In terms of performance, both proposed implementations ($pq$-Grams and ZhangShasha) clearly outperform the NoTED approach. In particular, even when $n$ equals $35$, the $pq$-Grams algorithm requires no more than $10$ minutes, whereas the NoTED approach requires almost $90$ minutes. The ZhangShasha algorithm is also quite compelling requiring less than $15$ minutes to run. Thus, the $pq$-Grams and ZhangShasha approaches are approximately $9.5$ and $6.5$ times faster than the NoTED approach, respectively.

Concerning all approaches, the mining step is indeed the most inefficient. Although ranking might also seem inefficient, its elapsed time depends mainly on the output of the mining step. Concerning the graph reduction step, simple tree reduction performs quite efficiently. Although graph reduction techniques deviate from the scope of this paper, note that subtree reduction required many hours to reduce the graphs.

Performance results are also shown in Figure 5b, where the vertical axis is in logarithmic scale in order to sufficiently illustrate the steps of the algorithms. As expected, performance is largely affected by the number of graphs taken into account, i.e., the $n$ parameter. The impact of $n$ is depicted in Figure 5a; the execution time of both approaches is high-order-polynomial with respect to consecutive values of $n$. This is expected since subgraph mining algorithms, such as CloseGraph, are affected by the size of the graphs and the size of the dataset. Further analyzing Figure 5a, the peak at $n = 25$ is not totally unexpected since the performance of subgraph mining algorithms may be affected by numerous properties, such as the structure of the graph. In any case, concerning the proposed algorithms, $pq$-Grams executes faster than ZhangShasha for all values of $n$, while NoTED is certainly less efficient.

Table IV provides effectiveness measurements for locating the three bugs, for all different algorithms. The "Position" attribute of the table indicates how many functions should the developer examine in order to locate the bug. This metric is created using the final ranking of the functions and identifying the position of the "buggy" function. Using the total number of functions, which for bugs 1, 2, and 3 is 637, 737, and 777 respectively, the percentage of the program's functions that should be examined to locate the bug is also provided.

Our approaches seem to perform not only closely, but also even more effectively than the NoTED approach, as long as




(a) (b) (c)

Fig. 5. Average performance and effectiveness diagrams for the bugs of the dataset. Diagrams (a) and (b) provide the elapsed time for each run of the algorithm. Diagram (a) depicts the total elapsed time of the $pq$-Grams and ZhangShasha approaches versus the value of parameter $n$ (which denotes the number of traces retained from each of the two sets, correct and incorrect), while diagram (b) illustrates the performance for each phase of the algorithms in logarithmic scale. Diagram (c) illusrates the percentage of functions to be examined in order to detect the bug, versus $n$.

$n$ is large enough. In fact, the $pq$-Grams and ZhangShasha approaches provide a better ranking for the third bug if $n$ is greater than or equal to 25. Effectiveness is also satisfactory for the first two bugs. The diversity of the results for the three bugs is rather expected since the size of the traces is different for each bug (see Table II). Thus, the third bug produces a much more difficult test case than the other two.

The impact of $n$ on effectiveness is illustrated in Figure 5c, which depicts the percentage of functions required to be examined versus $n$ for the three implementations. The effectiveness of our algorithms is indeed significant for large enough values of $n$. Although small $n$ values result in less satisfactory results, this is rather expected since useful trace information is lost. However, selecting an appropriate $n$ value not only reaches but also surpasses the effectiveness of the NoTED algorithm.

## VI. Conclusion and Future Work

Although there are several approaches for locating non-crashing bugs in source code, many of them suffer from scalability issues. With support from the experimental results of subsection V-B, we argue that our approaches achieve scalability without compromising effectiveness. According to our findings, reducing also the size of the dataset, as opposed to reducing only the graphs, yields quite promising results.

Concerning the dataset reduction step, both TED algorithms are very efficient. Although the performance of Zhang-Shasha is satisfactory, using $pq$-Grams provided faster runs and better function rankings. Conclusively, when only the relative edit distance of tree pairs is important, approximate TED algorithms, such as $pq$-Grams, perform similarly to exact ones.

The field of dynamic bug detection is far from exhausted when it concerns creating a scalable and effective algorithm. We argue, however, that our algorithms are a step in the right direction. Future research includes further testing to explore their efficiency in different datasets. In addition, further analysis of TED algorithms could lead to more effective solutions. Finally, the dataset reduction and subgraph mining steps can also be improved by designing new approaches. In any case, dataset reduction should definitely be taken into account.

## References

[1] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, “Bug isolation via remote program sampling,” SIGPLAN Not., vol. 38, no. 5, May 2003, pp. 141–154.

[2] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, “Sober: statistical model-based bug localization,” SIGSOFT Softw. Eng. Notes, vol. 30, no. 5, Sept. 2005, pp. 286–295.

[3] M. Renieris and S. Reiss, “Fault localization with nearest neighbor queries,” in Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on, 2003, pp. 30–39.

[4] C. Liu, X. Yan, H. Yu, J. Han, and P. S. Yu, “Mining Behavior Graphs for ”Backtrace” of Noncrashing Bugs,” in SDM, 2005.

[5] G. Di Fatta, S. Leue, and E. Stegantova, “Discriminative pattern mining in software fault detection,” in Proc. of the 3rd international workshop on Software quality assurance (SOQUA), 2006, pp. 62–69.

[6] F. Eichinger, K. Böhm, and M. Huber, “Mining edge-weighted call graphs to localise software bugs,” in European Conference on Machine Learning and Knowledge Discovery in Databases, 2008, pp. 333–348.

[7] X. Yan and J. Han, “gspan: Graph-based substructure pattern mining,” in Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM), 2002, pp. 721–724.

[8] Y. Chi, Y. Yang, and R. R. Muntz, “Indexing and mining free trees,” in Proc. of the Third IEEE International Conference on Data Mining, ser. ICDM ’03, 2003, pp. 509–512.

[9] X. Yan and J. Han, “Closegraph: mining closed frequent graph patterns,” in Proc. of the 9th ACM international conference on Knowledge Discovery and Data Mining, ser. KDD ’03, 2003, pp. 286–295.

[10] K.-C. Tai, “The tree-to-tree correction problem,” J. ACM, vol. 26, no. 3, July 1979, pp. 422–433.

[11] K. Zhang and D. Shasha, “Simple fast algorithms for the editing distance between trees and related problems,” SIAM J. Comput., vol. 18, no. 6, Dec. 1989, pp. 1245–1262.

[12] N. Augsten, M. Böhlen, and J. Gamper, “Approximate matching of hierarchical data using pq-grams,” in Proceedings of the 31st international conference on Very large data bases (VLDB), 2005, pp. 301–312.

[13] “daisydiff: A java library to compare html files,” [retrieved August, 2013]. [Online]. Available: http://code.google.com/p/daisydiff/

[14] “Software & algorithms, ISSEL,” [retrieved August, 2013]. [Online]. Available: http://issel.ee.auth.gr/software-algorithms/

[15] M. Philippsen, M. Wörlein, A. Dreweke, and T. Werth, “Parsemis: The parallel and sequential mining suite,” [retrieved August, 2013]. [Online]. Available: www2.informatik.uni-erlangen.de/EN/research/ParSeMiS/

[16] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, “The weka data mining software: an update,” SIGKDD Explor. Newsl., vol. 11, no. 1, Nov. 2009, pp. 10–18.

 ISBN: 978-1-61208-304-9 