Localizing Software Bugs using the Edit Distance of Call Traces

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

***Abstract*—Automating the localization of software bugs that do not lead to crashes is a difficult task that has drawn the attention of several researchers. Several popular methods follow the same approach; function call traces are collected and represented as graphs, which are subsequently mined using subgraph mining algorithms in order to provide a ranking of potentially buggy functions-nodes. Recent work has indicated that the scalability of state-of-the-art methods can be improved by reducing the graph dataset using tree edit distance algorithms. The call traces that are closer to each other, but belong to different sets, are the ones that are most significant in localizing bugs. In this work, we further explore the task of selecting the most significant traces, by proposing different call trace selection techniques, based on the Stable Marriage problem, and testing their effectiveness against current solutions. Upon evaluating our methods on a real-world dataset, we prove that our methodology is scalable and effective enough to be applied on dynamic bug detection scenarios.**

Keywords-automated debugging, dynamic bug detection, frequent subgraph mining, tree edit distance, Stable Marriage problem.

I. INTRODUCTION

During the latest few decades, software reliability has grown to be a major concern for both academia and the industry. Software bugs can lead to faulty software and dissatisfied customers, since testing and debugging are quite costly even compared to the software 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 software bugs: *crashing* bugs and *non-crashing* bugs. The former, as their name implies, lead to program crashes, thus they are easier to locate by tracing the call stack at the time of the crash. The latter, however, are logic errors that do not lead to crashes. The problem of locating non-crashing bugs is quite difficult, since no stack trace of the failure is available. Thus, finding a bug would usually involve careful examination of the source code, thorough testing, even pure intuition as to where the bug might reside. An interesting line of research aims towards automating the bug locating procedure by applying *Data Mining (DM)* techniques on call traces of *correct* and *incorrect* program executions. Hence, since dynamic analysis is performed to detect such bugs, the field is known as *dynamic bug detection*.

As noted in [1], dynamic bug detection techniques may be broadly 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 or boolean predicates. Indicatively, Liblit et al. [2] heuristically eliminate counters and apply logistic regression (or statistics as in [3]) to identify the attributes that affect the class (bug vs no bug). On the other hand, Liu et al. [4] employ boolean predicate statistics on correct and incorrect executions to localize bugs.

A more coarse-grained approach concerns inserting checks at block level, where blocks are fragments of code between branches. Renieris and Reiss [5] follow this approach and perform nearest neighbor queries to identify incorrect execution traces that are close to correct ones and compare these couples to detect potentially buggy blocks of code.

Counter-level and block-level approaches are quite precise in localizing bugs. However, the rise of *Object Oriented* and *Functional Programming* has led to preference for small comprehensive functions, indicating that instrumenting functions can also be effective. Mining traces at function level, and thus identifying potentially buggy functions is generally fine-grained enough for localizing a bug, as long as proper programming paradigms are employed. Approaches in this category employ *Graph Mining* techniques to call traces to identify which subgraphs are more frequent in incorrect than in correct runs [6]–[8]. Current approaches include also efforts towards improving the Graph Mining procedure [9], or using different representations such as N-grams (subsequences of length N) [10], or even reformulating the problem as a search/optimization problem [11]. Although these approaches are effective, their scalability is arguable.

The procedure of bug localization is similar for all approaches. 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.

Previous work [1] on reducing the size of the dataset has indicated considerable improvement on both scalability and effectiveness. We extend this work by reformulating the problem and providing different methodologies that focus on achieving effectiveness without compromising scalability. As in [1], the methodology involves performing Graph Mining

techniques to a subset of the dataset, thus the main focus lies on determining that subset. The similarity between any two call traces of the dataset can be defined as their edit distance, i.e., the cost of turning the one trace to the other. Given the distances between all combinations of traces, the problem is reduced to designing a call trace selector algorithm that successfully determines the most "useful" traces, i.e., the ones that successfully isolate the bug. In this paper, we further explore the effect of the call trace selector algorithm, by proposing two new algorithms, and comparing the different methodologies with respect to scalability and effectiveness. Furthermore, we benchmark our methods against known function-level dynamic bug detection techniques in order to discuss their applicability in real applications and evaluate their effectiveness against the current state-of-the-art.

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 insight for reducing the call trace dataset and explores the tasks of comparing call traces and selecting the most "useful" subset of the call trace dataset. 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 basic steps followed in function-level bug localization techniques, while denoting the different approaches. The following subsections correspond to the main phases of constructing a graph dataset, reducing the graphs to improve scalability, and applying Graph Mining techniques to provide the final ranking of possibly buggy functions.

A. Graph Dataset Construction

Assuming there is a set of test cases, program functions are instrumented and the test cases are executed to produce a set of *call traces* (or *scenarios*) 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 program executions. The correctness of an execution can be determined by the developer. In generated datasets such as the one used in this paper, one can also determine it automatically by comparing the output of the erroneous versions of the program with the correct version. Thus, upon collecting the execution traces, the graph dataset[1] is constructed.

B. Graph Reduction

At this point, the graphs of the dataset are quite large, with thousands of nodes and respective edges. Applying a Graph Mining algorithm to such a dataset would be highly inefficient. Thus, *graph reduction* is performed on each graph in order to keep only useful information while discarding all redundant data, to reduce its size. Figure 1 depicts several state-of-the-art graph reduction techniques.

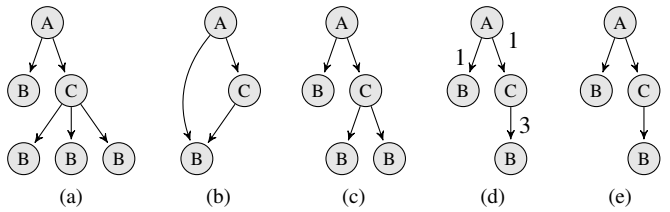

Figure 1. An example call graph (a) and four different reduced graphs with respect to the reduction techniques, including (b) total reduction, (c) one-two-many reduction, (d) subtree reduction and (e) simple tree reduction.

The first technique, known as *total reduction*, is presented by Liu et al. [6]. 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, shown in Figure 1b, is the most efficient reduction method since it actually preserves minimum information.

However, total reduction fails to capture the structure of the call graph, thus 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. [7]; the authors perform *one-two-many* reduction, preserving tree structure by keeping exactly two child nodes whenever the children of a node are more than two (see Figure 1c).

Eichinger et al. [8] 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).

As one might observe, the reduction techniques are based on a compromise between information loss and scalability. Although subtree reduction maintains most information, it is quite inefficient since it immediately adds a weight parameter to the graph. Since the scope of this work lies in scalability, we decided to use a reduction technique called *simple tree reduction*, shown in Figure 1e, which was originally introduced in [1]. Reducing a graph using simple tree reduction involves traversing all 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 (functions) 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

[1] Any tree is obviously also a graph. The terms are used interchangeably concerning the class of techniques that may be applied to the dataset.

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 is a well-known problem, defined as *Frequent Subgraph Mining (FSM)*. State-of-the-art algorithms include gSpan [12] and Gaston [13]. Furthermore, since these graphs are actually trees, several *Frequent Subtree Mining (FTM)* algorithms, such as FreeTreeMiner [14], may be used as well. Although those algorithms are applicable to the problem, there is strong preference for *CloseGraph* [15], an algorithm that is highly scalable since it prunes unnecessary input and outputs only closed frequent subgraphs.

D. Ranking

The output of the CloseGraph algorithm 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 which subgraphs are actually interesting. For instance, Di Fatta et al. [7] suggest ranking the functions according to their support in the failing set. According to Eichinger et al. [8], this type of ranking can be called *structural*. The structural ranking 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 defined in equation (1) 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 [4][7]. In this paper, we use an entropy-based ranking technique proposed by Eichinger et al. [8] 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

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

In Table I, $F = f_1, f_2, \ldots$ is the set of functions and $G = G_1, G_2, \ldots$ is the set of graphs. The table is constructed given the support of each subgraph in the graphs. Thus, 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. If, e.g, both SG_2 and SG_3 contain the edge $f_1 \rightarrow f_2$ and they appear 5 and 4 times respectively in graph G_2, then the support of this edge in graph G_1 is 9. 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. [8] 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.

Finally, one can calculate the combined ranking for a function f by averaging over its structural ranking $P_s(f)$ and its entropy-based ranking $P_e(f)$. However, since $P_s(f)$ and $P_e(f)$ may have different maximum values, it is necessary that their values are normalized by dividing each ranking by its maximum value. Thus, the combined ranking $P(f)$ is calculated as follows:

$$P(f) = \frac{1}{2} \cdot \left(\frac{P_e(f)}{\max\limits_{f \in F} P_e(f)} + \frac{P_s(f)}{\max\limits_{f \in F} P_s(f)} \right) \tag{2}$$

where the maximum values at the denominators are used in order to normalize the weighting of each ranking. Finally,$P(f)$ provides the probability that a function f contains a bug in the range $[0, 1]$.

III. REDUCING THE GRAPH DATASET

The steps defined in Section II are common for all function-level bug detection algorithms. Several researchers have indicated the need for scalable solutions, which is generally accomplished by reducing the graphs (see Subsection II-B). Ideally, graph reduction captures the most important information of the graph while minimizing its size. However, even upon reduction, the number of graphs in the dataset is quite large, thus making the mining step quite inefficient.

When a dataset of several graphs is given, not all of them are equally useful in locating the bug. Consider a simple 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, a$_{(1)}$, ate, a$_{(2)}$, and, then, it, on, 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. However, this is also true for the 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 above 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, at first, there is the need for a similarity metric between two traces. Having such a metric, one can apply an algorithm to select the most discriminating call traces based on the aforementioned intuitive remarks. Similarly to [1], the metric used to compare the similarity between two trees is the edit distance between them. Subsection III-A provides different alternatives for computing this metric, while Subsection III-B illustrates our proposed algorithms for using it to reduce the size of the dataset.

A. The Tree Edit Distance Problem

A metric widely used to represent the similarity between two strings is the *String Edit Distance (SED)* between them. 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 paragraphs provide a definition of the TED problem as well as two well known algorithms of current literature in finding TED.

The TED problem was originally posed by Tai [16] in 1979. The possible *edit operations* are defined in Figure 2.

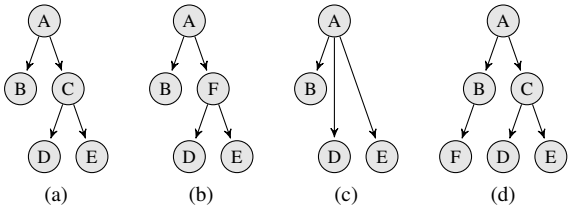

Figure 2. An example tree (a) and three different edit operations: (b) node relabeling, (c) node deletion, and (d) node insertion.

The first operation, *node relabeling*, concerns simply changing the label of a node (see Figure 2b). *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 2c, 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 2d. Assuming a *cost function* is defined for each edit 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 these operations. Thus, the TED problem is defined as determining the *optimal edit script*, i.e., the one with the minimum cost.

1) Zhang-Shasha Algorithm: One of the most well known TED algorithms is the *Zhang-Shasha* algorithm, which was named after its authors, K. Zhang and D. Shasha [17]. 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 recursive algorithm for computing TED is defined using the following equations:

$$\delta(\theta, \theta) = 0 \tag{3}$$

$$\delta(T_1, \theta) = \delta(T_1 - u, \theta) + \gamma(u \rightarrow \lambda) \tag{4}$$

$$\delta(\theta, T_2) = \delta(\theta, T_2 - v) + \gamma(\lambda \rightarrow v) \tag{5}$$

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

where $T - u$ denotes tree T without node u and $T - T(u)$ denotes tree T without u or any of each children. Parameter λ is the performed edit operation. The *Zhang-Shasha* algorithm uses *Dynamic Programming (DP)* in order to compute the TED. The *keyroots* of a tree T are defined as:

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

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

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

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

2) pq-Grams Algorithm: Several algorithms solve the TED problem effectively. However, even the most efficient ones lack scalability, since the polynomial order of the problem is high. A promising way of reducing the complexity of the problem and improving efficiency 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 be sufficient for the call trace selector algorithms of Subsection III-B.

Such an approximate TED algorithm is the *pq-Grams* based algorithm proposed by Augsten et al. [18]. 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 3.

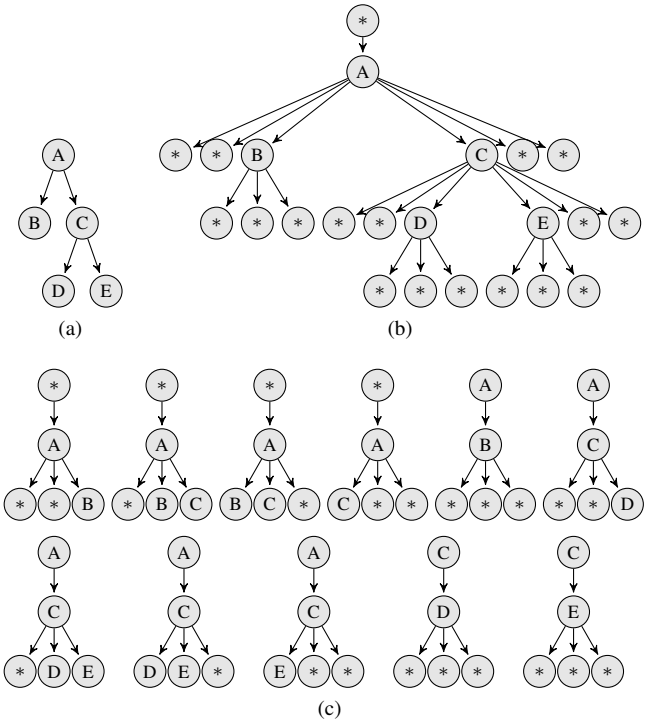

Figure 3. A pq-Grams example for $p = 2$ and $q = 3$, containing (a) an example tree, (b) its extended form for $p = 2$ and $q = 3$, and (c) its pq-Grams.

Parameters p and q 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 3c is $\{*, \texttt{A}\}$ and its base is $\{*, *, \texttt{B}\}$. Since the pq-Grams for the tree of Figure 3a cannot be directly created, an intermediate step of extending the tree with dummy nodes is shown in Figure 3b. Finally, The *pq-Gram profile* is the set of all pq-Grams of a tree (see Figure 3c), while the *pq-Gram index* of the tree is defined as the bag of all label tuples for the tree. For example, the pq-Gram index for the tree of Figure 3 is defined as:

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

According to Augsten et al. [18], 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.

B. The Call Trace Selection Problem

In the previous subsection, we provided two different methods for defining and computing the similarity between two call traces. Assuming that the similarity between all correct-incorrect pairs of the dataset is computed, the problem lies in using this information to determine which call traces can successfully isolate the bug. Formally, assuming that our input consists of the correct and incorrect sets, $S_{correct}$ and $S_{incorrect}$, we must design an algorithm that shall output two new sets $S'_{correct}$ and $S'_{incorrect}$. Let n be the size of each of the new sets, where set $S'_{correct}$ contains the n most important correct graphs and set $S'_{incorrect}$ contains the n most important incorrect graphs. The following paragraphs describe three different call trace selector algorithms that we implemented to solve the problem.

1) The SimpleSelector algorithm: The first algorithm implemented is the *SimpleSelector* algorithm, which was originally described in [1]. The application of the algorithm is shown in Figure 4.

`Input:` n, $S_{correct}$, $S_{incorrect}$
`Output:` $S'_{correct}$, $S'_{incorrect}$
$D = \{(g_c, g_i) \ \forall g_c \in S_{correct}, g_i \in S_{incorrect}\}$
`sort(`D`, key=similarity(`g_c, g_i`))`
$S'_{correct}$ =`First(`$n, \{g_c : \ g_c \in d \in D\}$`)`
$S'_{incorrect}$ =`First(`$n, \{g_i : \ g_i \in d \in D\}$`)`

Figure 4. The SimpleSelector algorithm that sorts the pairs of (correct, incorrect) traces and outputs the first n correct and the first n incorrect traces.

As shown in the 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`. Finally, function `similarity` can be easily determined using either of the two methods presented in Subsection III-A.

Previous work [1] has indicated that the SimpleSelector algorithm is adequate in terms of effectiveness. Intuitively, since the algorithm selects the most similar pairs of traces, it certainly captures some of the most important traces. However, the algorithm does not account for similar graphs of the same set. In specific, given the above example, graphs g_3 and g_4 could potentially be quite similar to each other. Assuming g_3 and g_4 have identical similarity metrics with g_1, keeping both of them on the final set could produce redundant information. Thus, the second call trace selector algorithm was implemented in order to eliminate this redundancy.

2) The StableMarriage algorithm: The second algorithm implemented is based on the *stable marriage* (or *stable matching*) problem. The problem, first introduced by D. Gale and L. S. Shapley [19] in 1962, has many variants (see [20] for an

extensive survey) since it has numerous applications to real-life problems. According to its original definition, there are N men and N women, and every person ranks the members of the opposite sex in a strict order of preference, i.e., with no equal preference for any member of the opposite sex. The problem is to find a matching between men and women so that there are no two people of opposite sex who would both rather be matched to each other than their current partners. If there are no such people, the matching (marriage) is considered to be *stable*.

The call trace selection problem can be formulated as a stable marriage problem. In this case, the two sexes are the traces of the correct and the traces of the incorrect set. For any trace, given its similarity with the traces of the opposite set, a ranked preference list is formed. Given that the probability two pairs having the same similarity value is too low, we can safely assume that the lists are strictly ordered. Upon forming all the preference lists, the *Gale-Shapley* algorithm [19] can be applied to our problem. The application of the algorithm is shown in Figure 5.

```
Input: n, S_correct, S_incorrect
Output: S'_correct, S'_incorrect
Find preference list ∀g ∈ {S_correct ∪ S_incorrect}
Set all g_c ∈ S_correct and g_i ∈ S_incorrect to unmatched
while ∃ free g_c that has a g_i to try to match
  g_i = g_c's highest ranked incorrect trace
            that g_c has not yet tried to match
  if g_i is unmatched
    (g_c, g_i) are matched together
  else there is a matching (g'_c, g_i)
    if g_i prefers g_c to g'_c
      (g_c, g_i) are matched together
      g'_c becomes unmatched
D = {all pairs (g_c, g_i) of stable matching}
sort(D, key=similarity(g_c, g_i))
S'_correct =First(n, {g_c : g_c ∈ d ∈ D})
S'_incorrect =First(n, {g_i : g_i ∈ d ∈ D})
```

Figure 5. The StableMarriage algorithm that finds the stable matching among the traces of the correct and the incorrect sets, and outputs the traces of the first n (correct, incorrect) pairs.

Functions `sort` and `similarity` work similarly to the SimpleSelector algorithm. The algorithm of Figure 5 initially creates the preference lists for each graph of the sets $S_{correct}$ and $S_{incorrect}$. Note that this step can be reduced to minimum complexity using suitable data structures. At the beginning of the algorithm, all graphs of both sets are unmatched. The algorithm iterates over all correct graphs and tries to match each correct graph g_c to its most preferred incorrect graph g_i for which there was not yet an attempt to match. Any incorrect graph accepts a proposal to match if it is unmatched or if the proposed matching is preferable to its current matching. Given sets $S_{correct} = \{g_1, g_2, g_3\}$ and $S_{incorrect} = \{g_4, g_5, g_6\}$ and the similarity between all correct-incorrect pairs one can construct the ranked preference lists of Figure 6.

$g_1 : g_5 > g_4 > g_6$ $\quad$ $g_4 : g_2 > g_1 > g_3$
$g_2 : g_5 > g_6 > g_4$ $\quad$ $g_5 : g_1 > g_2 > g_3$
$g_3 : g_6 > g_4 > g_5$ $\quad$ $g_6 : g_2 > g_3 > g_1$

(a) (b)

Figure 6. Example preference lists for the graphs of (a) the correct set $S_{correct} = \{g_1, g_2, g_3\}$ and (b) the incorrect set $S_{incorrect} = \{g_4, g_5, g_6\}$.

Iterating over the correct graphs, g_1 initially is matched to g_5 since it is in the top of its preference list. After that, g_2 tries also to match with g_5, and since the g_5 preference list indicates preference for g_2 over g_1, the new pair is (g_2, g_5) and g_1 becomes unmatched. After that, g_3 gets matched with g_6. Since there is still an unmatched graph g_1, and it already tried to match with its first choice it tries to match with its second choice g_4, which is accepted since g_4 is unmatched. Thus, the final pairs that form the set D are sorted according to their similarity, for example resulting in $D = \{(g_1, g_4), (g_3, g_6), (g_2, g_5)\}$. Given, e.g., that the number of retained graphs per set is $n = 2$, the sets $S'_{correct}$ and $S'_{incorrect}$ are $\{g_1, g_3\}$ and $\{g_4, g_6\}$ respectively.

The Gale-Shapley algorithm is proven to converge to a *male-optimal* solution [19], indicating in our case that there is no better matching for the correct graphs of the set. Furthermore, since the final list of the algorithm has no duplicates, any redundant data, i.e., graphs that belong to the same set, are discarded. Although the StableMarriage algorithm seems well adapted to the problem, the algorithm disregards the fact that the pairs are not only ranked but also weighted. Thus, StableMarriage can actually provide a sub-optimal solution to the problem. As a result, we have implemented another algorithm that accounts for the weights and is better suited to the problem at hand.

3) The MaxLinkMax algorithm: The third algorithm that we implemented was initially proposed as a solution to an extension of the stable marriage problem, based on the concept of defining a different form of *stability* for the problem [21]. One can define different notions of stability for a stable matching problem with weighted preferences (see [21] for an extensive definition of several alternatives). In our case, we used the *link-max stability* as a criterion denoting whether the matching is stable. Returning to the marriage metaphor, given a man and a woman, their *link-max strength* is the maximum between the preference value that the man gives to the woman and the preference value that the woman gives to the man. Given a stable marriage problem with weights, a matching is said to be *link-max stable* if there are no two people of the opposite sex of which the marriage would have larger link-max strength than either of the current matchings that each partner has. Formally, let $lm(m, w)$ be the link-max strength of a man m and a woman w, a link-max stable marriage does not contain any pair (m, w) such that:

$$lm(m, w) > lm(m', w) \tag{11}$$

$$lm(m, w) > lm(m, w') \tag{12}$$

where m' is the current partner of w and w' is the current partner of m.

The call trace selection problem can be formulated as a link-max stable matching problem, where the weighted preferences are the similarity values that were determined in Subsection III-A. Thus, one can easily calculate the link-max strength for any pair of (correct, incorrect) graphs, in our case defined as the similarity of the pair. After that, the problem can be solved similarly to the *Max-link-max* algorithm, which was introduced in [21] as a solution to link-max stable marriage problems. The application of the algorithm is shown in Figure 7.

```
Input: n, S_correct, S_incorrect
Output: S'_correct, S'_incorrect
Find preference list ∀g ∈ {S_correct ∪ S_incorrect}
D = ∅
M = {(g_c, g_i) ∀g_c ∈ S_correct, g_i ∈ S_incorrect}
while M ≠ ∅
   (g_c, g_i) = arg max_{(g_c,g_i)∈M} lm((g_c, g_i))
   D = D ∪ (g_c, g_i)
   M = M \ {(g_c, g'_i), (g'_c, g_i) ∀g'_c ∈ S_correct, g'_i ∈ S_incorrect}
S'_correct = First(n, {g_c : g_c ∈ d ∈ D})
S'_incorrect = First(n, {g_i : g_i ∈ d ∈ D})
```

Figure 7. The MaxLinkMax algorithm that finds the link-max stable matching among the traces of the correct and the incorrect sets, and outputs the traces of the first n (correct, incorrect) pairs.

The first step of the algorithm shown in Figure 7 is to create a weighted preference list for each graph in the dataset. After that, two sets are defined: the initial set M, which contains all possible correct-incorrect pairs of graphs and the set D, which is initially empty. The algorithm iterates until the set M is empty. For each iteration, the correct-incorrect pair with the maximum link-max strength is found and added to set D. Let (g_c, g_i) be the selected pair, all the pairs of set M that contain either g_c or g_i are removed from graph M. After all the iterations are over, i.e., the set M is empty, the set D contains the final matching. By contrast with the other algorithms, the pairs are already sorted, thus the final sets $S'_{correct}$ and $S'_{incorrect}$ are immediately defined as the first n correct graphs and the first n incorrect graphs that belong to the first n pairs of D.

The MaxLinkMax algorithm is rather more complex than the other two, since it requires not only a ranking but also the weighted preference for each pair of graphs in the dataset. In line with the remarks of [21], one can find the pair with the maximum link-max strength by saving only the maximum weight for every correct and for every incorrect graph in the dataset. This indicates that the complexity of the algorithm is no higher than that of the two previous algorithms that we implemented. For example, given the graphs of Figure 6, and the weights of the pairs (g_1, g_5), (g_2, g_5), (g_3, g_6), (g_4, g_2), (g_5, g_1) (which is equal to (g_1, g_5)), and (g_6, g_2), one requires to compare only these six values in order to find the pair with the maximum link-max strength. Assuming this pair is the (g_1, g_5), the new pairs are immediately reduced to (g_2, g_6), (g_3, g_6), (g_4, g_2), and (g_6, g_2) (which is equal to (g_2, g_6)). Let $lm(g_2, g_6) > lm(g_3, g_6) > lm(g_4, g_2)$, the pairs are reduced to the minimum (g_3, g_4) and (g_4, g_3) which are equal, directly providing with the final sorted set $D = \{(g_1, g_5), (g_2, g_6), (g_3, g_4)\}$.

In terms of the computational complexity of the algorithms, one could argue that it is quite satisfactory. However, since the expected number of call traces is usually small (e.g., usually less than a hundred), the main scope of these approaches lies in improving effectiveness rather than performance.

IV. Dataset

The bug detection techniques analyzed in Section II are quite effective for bug localization in small applications. For example, Eichinger et al. [8] evaluate their method against two known literature bug localization techniques ([6] and [7]) using a small dataset, generated using a `diff` implementation in `Java`. Although the effectiveness of the techniques is irrefutable, their 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 call graphs of roughly 20 nodes after the reduction step.

A much more realistic dataset was used in [1]. The dataset was generated using the source code of `daisydiff` [22], a `Java` application that compares `html` files. `daisydiff` provides a more suitable benchmark since it has almost 70 files with 9500 lines of code. Thus, concerning scalability in a real application, that dataset is certainly sufficient. In this work, we have further extended the `daisydiff` dataset to test more thoroughly the scalability and the effectiveness of our methodology. We have planted 6 different bugs in the code of the 1.2 version of `daisydiff`, which are shown in Table II.

TABLE II. Planted Bugs

Bugs	Description	# Functions
1	Wrong limit conditions (Forgot +1)	637
2	Missing AND condition (Forgot a < check)	737
3	Wrong condition (> instead of <)	777
4	Missing OR condition (Forgot a != check)	723
5	Missing 1 of 3 AND conditions (Forgot a == check)	756
6	Missing 1 of 2 AND conditions (Forgot a == check)	638

The table contains the description of each bug as well as the average number of unique functions that are called in the respective runs. The dataset covers common types of bugs, such as missing boolean conditions, wrong conditions and wrong limit conditions. As mentioned above, the aim of these bugs is twofold. Experiments on different bug cases shall provide insight on the effectiveness of our algorithms, while the scalability of our methodology will be evaluated on different scenarios. The initial bug-free version of the program and the six buggy versions were all run 100 times given different inputs. The dataset can be found online in [23].

V. Evaluation

This section presents the results of applying the algorithms to the dataset described in Section IV. Upon creating the traces, the executions were crosschecked against the correct versions of the traces to provide the correct and incorrect sets.

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

	pq-Grams							NoTED	ZhangShasha						
	5	10	15	20	25	30	35		5	10	15	20	25	30	35
Graph Parsing	16.71	6.57	6.45	6.38	6.35	6.34	6.86	8.94	6.86	6.39	6.40	6.46	6.42	6.48	6.41
Graph Reduction	3.43	3.20	3.21	3.18	3.19	3.17	3.18	4.15	3.18	3.18	3.14	3.29	3.15	3.24	3.23
Dataset Reduction	70.84	70.49	70.53	70.03	69.72	69.64	69.74	0.00	162.50	162.31	161.54	162.38	162.00	162.78	160.75
Subgraph Mining	19.24	45.64	178.63	538.78	1391.44	1675.24	1973.72	24296.94	22.95	85.56	94.87	447.33	1319.32	714.84	1974.36
Ranking Calculation	0.51	1.75	6.87	17.75	42.61	75.08	97.47	2003.99	0.48	2.08	7.79	23.37	50.05	71.84	113.04
Total	110.73	127.65	265.69	636.12	1513.31	1829.47	2150.97	26314.02	195.97	259.52	273.74	642.83	1540.94	959.18	2257.79

TABLE IV. ELAPSED TIME (IN SECONDS) FOR THE DIFFERENT PHASES OF THE ALGORITHMS FOR THE STABLEMARRIAGE APPROACH

	pq-Grams							NoTED	ZhangShasha						
	5	10	15	20	25	30	35		5	10	15	20	25	30	35
Graph Parsing	6.78	6.40	6.33	6.28	6.27	6.38	6.29	8.94	6.38	6.46	6.32	6.45	6.36	6.38	6.39
Graph Reduction	3.19	3.23	3.17	3.14	3.17	3.18	3.24	4.15	3.14	3.19	3.17	3.16	3.15	3.15	3.21
Dataset Reduction	69.33	70.05	69.88	69.93	69.65	69.27	69.62	0.00	161.10	161.22	161.28	160.78	161.15	160.74	163.28
Subgraph Mining	35.51	111.15	334.84	322.36	884.47	998.60	827.78	24296.94	21.81	154.33	376.75	630.86	822.89	678.92	917.49
Ranking Calculation	0.57	3.63	12.07	21.77	32.34	38.72	42.68	2003.99	0.60	4.05	15.16	28.29	40.79	45.99	54.72
Total	115.38	194.46	426.29	423.48	995.90	1116.15	949.61	26314.02	193.03	329.25	562.68	829.54	1034.34	895.18	1145.09

TABLE V. ELAPSED TIME (IN SECONDS) FOR THE DIFFERENT PHASES OF THE ALGORITHMS FOR THE MAXLINKMAX APPROACH

	pq-Grams							NoTED	ZhangShasha						
	5	10	15	20	25	30	35		5	10	15	20	25	30	35
Graph Parsing	6.64	6.45	6.46	6.38	6.60	6.48	6.86	8.94	6.86	6.49	6.44	6.55	6.48	7.51	7.15
Graph Reduction	3.21	3.18	3.21	3.18	3.35	3.29	3.39	4.15	3.28	3.27	3.28	3.27	3.32	3.30	3.37
Dataset Reduction	70.79	70.66	71.66	70.85	74.41	74.35	74.79	0.00	169.19	168.89	168.29	168.24	168.29	168.56	168.53
Subgraph Mining	15.43	42.54	162.40	386.40	514.70	742.17	798.19	24296.94	19.13	226.84	379.05	412.76	844.61	1345.54	1439.16
Ranking Calculation	0.53	2.24	9.66	31.33	42.29	51.66	50.89	2003.99	0.54	3.07	12.95	28.67	55.16	75.43	74.60
Total	96.60	125.07	253.39	498.14	641.35	877.95	934.12	26314.02	199.00	408.56	570.01	619.49	1077.86	1600.34	1692.81

A. Experimental Setup

Several methods were implemented in order to test the validity of our dataset reduction hypothesis. An initial test was performed by applying the algorithm by Eichinger et al. [8] as discussed in Section II. However, due to performance limitations, subtree reduction (see Subsection II-B) could not be applied in such a large dataset. Thus, simple tree reduction is used in its place (refer to [1] for more details). In the conducted tests, the graph mining step is performed through the Parallel and Sequential Mining Suite (ParSeMiS) [24] implementation of CloseGraph, while the InfoGain algorithm of the ranking step was implemented using the Waikato Environment for Knowledge Analysis (WEKA) [25].

We implemented six more algorithms, accounting for all possible combinations of similarity metrics: ZhangShasha and pq-Grams implementations combined with all three call trace selection algorithms (SimpleSelector, StableMarriage and MaxLinkMax). For comparison reasons, all algorithms were implemented using the same libraries stated above. The experiments were conducted using 7 different values for the n parameter (5, 10, 15, 20, 25, 30, and 35) in order to explore its effect on performance and effectiveness. Concerning efficiency, the selection of these values for n is reasonable; the algorithms would be inefficient if they took into account more than 35 traces per set, i.e., 70 traces overall when the total number of traces is 100. In contrast, smaller values of n would compromise effectiveness. When few traces are retained per set, the bugs may not be distinguishable between these traces.

All experiments were performed using an 8-core i7 processor with 8 GBs 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. The call trace selection algorithms were executed sequentially. 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 20% support threshold.

The performance results for all edit distance methods are shown in Tables III, IV and V for the SimpleSelector, StableMarriage and MaxLinkMax approaches respectively. These tables contain the average measurements for all six bugs of the dataset. Although the elapsed time required to localize a bug can be significantly different for two different bugs (e.g., bug 3 required 5 times more time than bug 1), the trend for all bugs is similar. Each table contains measurements for the

TABLE VI. 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	SS	7 (1.1%)	9 (1.4%)	31 (4.9%)	9 (1.4%)	8 (1.3%)	8 (1.3%)	8 (1.3%)	**5 (0.8%)**	7 (1.1%)	8 (1.3%)	9 (1.4%)	8 (1.3%)	9 (1.4%)	8 (1.3%)	8 (1.3%)
	SM	6 (0.9%)	8 (1.3%)	10 (1.6%)	8 (1.3%)	8 (1.3%)	8 (1.3%)	8 (1.3%)	**5 (0.8%)**	6 (0.9%)	8 (1.3%)	13 (2.0%)	12 (1.9%)	12 (1.9%)	12 (1.9%)	12 (1.9%)
	MLM	6 (0.9%)	8 (1.3%)	9 (1.4%)	8 (1.3%)	8 (1.3%)	8 (1.3%)	8 (1.3%)	**5 (0.8%)**	6 (0.9%)	8 (1.3%)	10 (1.6%)	8 (1.3%)	8 (1.3%)	8 (1.3%)	8 (1.3%)
Bug 2	SS	**5 (0.7%)**	**5 (0.7%)**	9 (1.2%)	9 (1.2%)	9 (1.2%)	9 (1.2%)	9 (1.2%)	9 (1.2%)	**5 (0.7%)**	**5 (0.7%)**	9 (1.2%)	9 (1.2%)	9 (1.2%)	9 (1.2%)	9 (1.2%)
	SM	**5 (0.7%)**	**5 (0.7%)**	9 (1.2%)	9 (1.2%)	9 (1.2%)	9 (1.2%)	9 (1.2%)	9 (1.2%)	**5 (0.7%)**	**5 (0.7%)**	9 (1.2%)	9 (1.2%)	9 (1.2%)	9 (1.2%)	9 (1.2%)
	MLM	**5 (0.7%)**	**5 (0.7%)**	9 (1.2%)	9 (1.2%)	9 (1.2%)	9 (1.2%)	9 (1.2%)	9 (1.2%)	**5 (0.7%)**	**5 (0.7%)**	9 (1.2%)	9 (1.2%)	9 (1.2%)	9 (1.2%)	9 (1.2%)
Bug 3	SS	254 (32%)	252 (32%)	342 (44%)	27 (3.5%)	3 (0.4%)	**1 (0.1%)**	16 (2.1%)	17 (2.2%)	259 (33%)	253 (32%)	346 (44%)	355 (45%)	**1 (0.1%)**	**1 (0.1%)**	2 (0.3%)
	SM	168 (21%)	336 (43%)	341 (43%)	215 (27%)	3 (0.4%)	**2 (0.3%)**	16 (2.1%)	17 (2.2%)	246 (31%)	261 (33%)	353 (45%)	360 (46%)	9 (1.2%)	**2 (0.3%)**	4 (0.5%)
	MLM	256 (32%)	336 (43%)	6 (0.8%)	2 (0.3%)	**1 (0.1%)**	**1 (0.1%)**	**1 (0.1%)**	17 (2.2%)	256 (32%)	251 (32%)	3 (0.4%)	2 (0.3%)	2 (0.3%)	**1 (0.1%)**	**1 (0.1%)**
Bug 4	SS	270 (37%)	371 (51%)	**18 (2.5%)**	24 (3.3%)	29 (4.0%)	23 (3.2%)	51 (7.1%)	65 (9.0%)	268 (37%)	363 (50%)	27 (3.7%)	28 (3.9%)	35 (4.8%)	43 (5.9%)	29 (4.0%)
	SM	223 (30%)	391 (54%)	23 (3.2%)	23 (3.2%)	25 (3.5%)	24 (3.3%)	41 (5.7%)	65 (9.0%)	47 (6.5%)	399 (55%)	41 (5.7%)	**17 (2.4%)**	30 (4.1%)	35 (4.8%)	35 (4.8%)
	MLM	270 (37%)	35 (4.8%)	25 (3.5%)	25 (3.5%)	25 (3.5%)	25 (3.5%)	25 (3.5%)	65 (9.0%)	269 (37%)	33 (4.6%)	**20 (2.8%)**	**20 (2.8%)**	**20 (2.8%)**	**20 (2.8%)**	**20 (2.8%)**
Bug 5	SS	12 (1.6%)	6 (0.8%)	6 (0.8%)	6 (0.8%)	**5 (0.7%)**	**5 (0.7%)**	**5 (0.7%)**	**5 (0.7%)**	12 (1.6%)	6 (0.8%)	6 (0.8%)	6 (0.8%)	6 (0.8%)	6 (0.8%)	**5 (0.7%)**
	SM	19 (2.5%)	7 (0.9%)	6 (0.8%)	6 (0.8%)	6 (0.8%)	6 (0.8%)	6 (0.8%)	**5 (0.7%)**	19 (2.5%)	13 (1.7%)	7 (0.9%)	6 (0.8%)	6 (0.8%)	6 (0.8%)	6 (0.8%)
	MLM	12 (1.6%)	6 (0.8%)	6 (0.8%)	**5 (0.7%)**	**5 (0.7%)**	**5 (0.7%)**	**5 (0.7%)**	**5 (0.7%)**	12 (1.6%)	6 (0.8%)	6 (0.8%)	**5 (0.7%)**	**5 (0.7%)**	**5 (0.7%)**	**5 (0.7%)**
Bug 6	SS	6 (0.9%)	**3 (0.5%)**	4 (0.6%)	11 (1.7%)	9 (1.4%)	18 (2.8%)	15 (2.4%)	15 (2.4%)	22 (3.4%)	**3 (0.5%)**	**3 (0.5%)**	18 (2.8%)	9 (1.4%)	17 (2.7%)	15 (2.4%)
	SM	**3 (0.5%)**	**3 (0.5%)**	17 (2.7%)	18 (2.8%)	18 (2.8%)	18 (2.8%)	18 (2.8%)	15 (2.4%)	17 (2.7%)	**3 (0.5%)**	17 (2.7%)	15 (2.4%)	15 (2.4%)	15 (2.4%)	15 (2.4%)
	MLM	**3 (0.5%)**	**3 (0.5%)**	**3 (0.5%)**	18 (2.8%)	18 (2.8%)	18 (2.8%)	18 (2.8%)	15 (2.4%)	17 (2.7%)	17 (2.7%)	**3 (0.5%)**	15 (2.4%)	15 (2.4%)	15 (2.4%)	15 (2.4%)

*SS: SimpleSelector, SM: StableMarriage, MLM: MaxLinkMax

different values of n for each of the two similarity algorithms, pq-Grams and ZhangShasha, as well as the NoTED approach, which is the one not using any TED algorithm to reduce the size of the dataset.

In terms of total execution time, the proposed implementations clearly outperform the NoTED approach. In particular, even when n equals 35, the pq-Grams and ZhangShasha approaches require roughly 30 minutes using the SimpleSelector call trace selection algorithm. For the StableMarriage and MaxLinkMax algorithms, the respective value is even below 25 minutes. By contrast, the NoTED approach requires more than 7 hours in order to provide the final ranked list of functions. In relative terms, the proposed approaches are approximately 15 times faster than the NoTED approach.

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. The difference between the execution times of the tree edit distance algorithms pq-Grams and ZhangShasha is quite significant since the former is twice as faster as the latter. However, their contribution to the total execution time is rather insignificant with respect to the mining step. Using either of the proposed TED techniques, the choice of a call trace selection algorithm seems also irrelevant to the performance of the different implementations. Note, however, that since graph mining algorithms depend highly on the graphs that are given to them, dataset reduction could affect the measurement. Finally, although graph reduction techniques deviate from the scope of this paper, note that subtree reduction required many hours to reduce the graphs.

Table VI provides effectiveness measurements for locating the six bugs, for all different algorithms. The value inside each each cell 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 is shown in Table II for each bug, the percentage of the program's functions that should be examined to locate the bug is also provided. This is given inside a parenthesis in the value of each cell.

Our approaches seem to perform not only closely, but also even more effectively than the NoTED approach. In particular, our approaches provide a better ranking for bugs 2, 3, 4, and 6. The effectiveness metrics for bugs 3, 4, and 6 are very promising; these bugs seem to be the most "difficult" to locate. They are localized ineffectively by the NoTED approach, requiring 17, 65, and 15 functions respectively to be examined in order to find them. Our approaches outperform these results for bugs 3 and 4, as long as n is large enough. Concerning the sixth bug, the localization is even more satisfactory; our algorithms manage to localize the bug even when the values of n are small. This is also true for bugs 1 and 2, where the bug is localized either almost as good as the NoTED approach (for bug 1) or even better (for bug 2). Finally, the results for the fifth bug are also quite encouraging since our algorithms perform no worse than the NoTED one.

Since the nature of each bug is complex, no safe assumption can be made concerning the effectiveness of any algorithm with respect to the type of each bug or the number of function calls. In other words, the effectiveness is highly dependent on the selected dataset. Removing or switching certain boolean conditions can lead to bugs that are very difficult to locate, such as bugs 3, 4, and 6, or to easier cases, such as bugs 2 and 5. This is expected since the structure of the call traces can be considerably altered by any bug. Notice, for instance, how bugs 5 and 6, though similar, result to traces with 756 and 638 (unique) function calls respectively. In any case, the planted bugs are quite indicative of those arising in realistic scenarios.

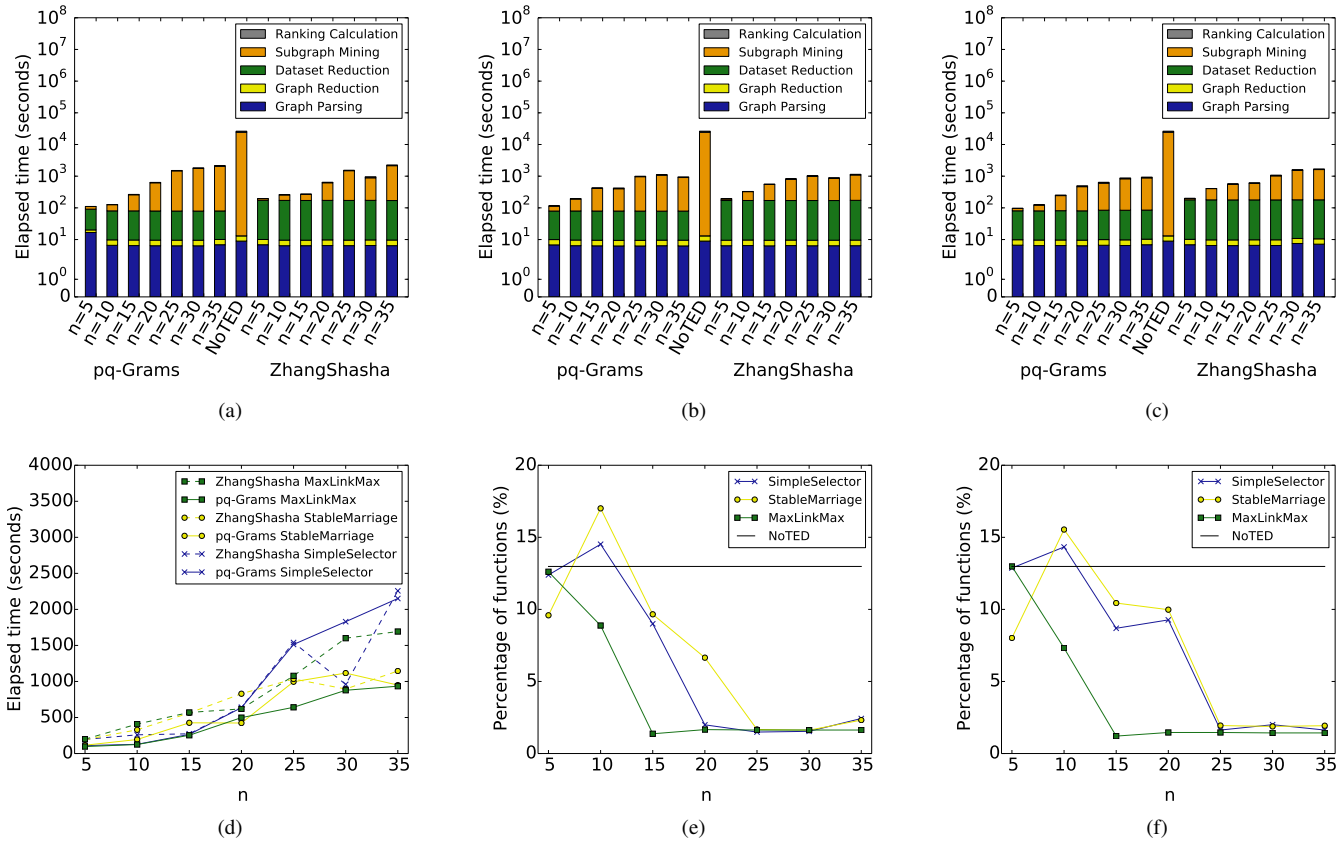

Figure 8. Average performance and effectiveness diagrams for the bugs of the dataset. Diagrams (a), (b) and (c) illustrate the performance for each phase of the algorithms in logarithmic scale versus the value of n (which denotes the number of traces retained from each of the two sets, correct and incorrect). The three diagrams correspond to the SimpleSelector, StableMarriage and MaxLinkMax algorithms. Diagram (d) depicts the total elapsed time of all combinations of the TED approaches, pq-Grams and ZhangShasha, with the call trace selection approaches, SimpleSelector, StableMarriage and MaxLinkMax. Diagrams (e) and (f) illustrate the percentage of functions to be examined in order to detect the bug versus n, for the pq-Grams and ZhangShasha approaches respectively.

Our conclusions regarding both effectiveness and performance are also confirmed by plotting the results, as in Figure 8. Figures 8a, 8b, and 8c illustrate the performance of our methodology for the SimpleSelector, StableMarriage and MaxLinkMax approaches respectively. Note that the vertical axis in these figures 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 the number of graphs is better depicted in Figure 8d; the execution time of all approaches is high-order-polynomial with respect to consecutive values of n. This is expected since subgraph mining algorithms, such as CloseGraph, are largely affected by the size of the graphs and the size of the dataset.

Further analyzing Figure 8d, pq-Grams seems to execute faster than ZhangShasha for most values of n, regardless of which call trace selection algorithm is used. Peaks such as the one of the ZhangShasha with SimpleSelector approach are 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, useful conclusions can also be drawn for the performance of the three different call trace selection approaches. MaxLinkMax is both efficient and stable, indicating that it is robust and fits the problem better than the other two algorithms.

Concerning effectiveness, the impact of n is illustrated in Figures 8e and 8f, which depict the percentage of functions required to be examined versus n for the three call trace selection algorithms, and the NoTED approach. These figures correspond to the pq-Grams and the ZhangShasha approaches respectively. As shown in this figures, the effectiveness of our algorithms is indeed significant for large enough values of n. In specific, these average values indicate that our algorithms outperform the NoTED approach as long as n is larger than or equal to 15, while the MaxLinkMax approach outperforms it even for values larger than 10.

Given that n is the number of traces retained from the two sets (correct and incorrect), its impact on effectiveness is rather expected. In specific, when few traces are kept from each set (e.g., for n values lower than 15), the algorithms may not effectively isolate the bug since it may not be clearly distinguishable between these few traces. On the other hand, large values of n ensure that at least some of the retained traces will be highly relevant for isolating the bug. However,

since larger n values result also in larger execution times, it is preferable to select values near 25 or 30 where the algorithms exhibit high effectiveness while also being efficient.

Figures 8e and 8f illustrate the relative effectiveness of the call trace selection techniques. In particular, the MaxLinkMax approach outperforms the other techniques in terms of both effectiveness and stability. The algorithm seems to converge to satisfactory values faster than its opposing techniques. In Figure 8e, the percentage of functions to be examined for the MaxLinkMax approach drops below 2% for all n values that are lower than or equal to 15. The SimpleSelector and StableMarriage approaches require values lower than or equal to 20 and 25, respectively, to exhibit similar effectiveness. The results for the ZhangShasha algorithm are even more characteristic, with the SimpleSelector and StableMarriage requiring both at least 25 call traces per set in order to approach the effectiveness achieved by MaxLinkMax when the number of kept call traces is at least 15.

As a result of the above analysis, the effect of the call trace selection algorithm on localizing the bugs is quite significant. The MaxLinkMax algorithm was actually expected to prevail since it is the most well adapted algorithm to the problem at hand; intuitively, determining the unique graph pairs with the maximum pair weights should provide satisfactory results. Allowing duplicates and disregarding weights, as done by SimpleSelector and StableMarriage respectively, can be seen as drawbacks, therefore, leading to suboptimal solutions. As expected, however, these algorithms perform satisfactorily for large n values, since useful trace information is then retained.

The relative effectiveness among the approaches using pq-Grams and the three ones using ZhangShasha may seem slightly surprising. The approximate pq-Grams implementations seem more stable and more effective than their exact ZhangShasha counterparts. However, the difference between the MaxLinkMax approaches is not significant, and the effectiveness for the other two call trace selection algorithms differs only for small values of n. This indicates that both pq-Grams and ZhangShasha provide satisfactory results, as long as the call trace selection algorithm properly utilizes the weight values provided by them.

VI. Conclusion and Future Work

Current approaches in the field of dynamic bug detection suffer from scalability issues. In this paper, we have expanded on previous work [1], further testing the effect of reducing the size of the call trace dataset on both performance and effectiveness. With support from the experimental results of Subsection V-B, we argue that our approaches exhibit considerable improvement on the current state-of-the-art, even more so since they are tested on a realistic dataset.

Concerning the dataset reduction step, both TED algorithms are effective in terms of finding relative edit distances between graphs. Thus, the main scope of this work focused on exploring different possibilities for selecting the most useful call traces. We provided three call trace selection algorithms and explored how they affect the effectiveness of our methodology. The third algorithm we proposed, MaxLinkMax, proved to be the most stable and effective, since it outperformed all other implementations, while also being highly efficient.

Although the field of locating non-crashing bugs is far from exhausted, we argue that our methodology provides an interesting perspective on the problem. In specific, this work indicates that analyzing the call trace dataset to isolate useful traces yields quite promising results. Hence, future research includes further exploring the applicability and effectiveness of our techniques on different datasets. Furthermore, the parameters of our techniques (mainly the number of retained traces) could be optimized or automatically derived for each dataset. Finally, further analysis of the newly defined problem of call trace selection with respect also to the subgraph mining step could lead to more effective solutions.

References

[1] T. Diamantopoulos and A. Symeonidis, "Towards Scalable Bug Localization using the Edit Distance of Call Traces," in Proceedings of the Eighth International Conference on Software Engineering Advances (ICSEA), Oct 2013, pp. 45–50.

[2] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug Isolation via Remote Program Sampling," SIGPLAN, vol. 38, no. 5, 2003, pp. 141–154.

[3] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable Statistical Bug Isolation," SIGPLAN, vol. 40, no. 6, 2005, pp. 15–26.

[4] 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, Sep. 2005, pp. 286–295.

[5] M. Renieris and S. Reiss, "Fault Localization with Nearest Neighbor Queries," in Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE), 2003, pp. 30–39.

[6] C. Liu, X. Yan, H. Yu, J. Han, and P. S. Yu, "Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs," in Proceedings of the 2005 SIAM International Conference on Data Mining, 2005, pp. 286–297.

[7] G. Di Fatta, S. Leue, and E. Stegantova, "Discriminative Pattern Mining in Software Fault Detection," in Proceedings of the 3rd International Workshop on Software quality assurance (SOQUA), 2006, pp. 62–69.

[8] F. Eichinger, K. Böhm, and M. Huber, "Mining Edge-Weighted Call Graphs to Localise Software Bugs," in Proceedings of the 2008 European Conference on Machine Learning and Knowledge Discovery in Databases - Part I, 2008, pp. 333–348.

[9] F. Eichinger, C. Oßner, and K. Böhm, "Scalable Software-Defect Localisation by Hierarchical Mining of Dynamic Call Graphs," in Proceedings of the 2011 SIAM International Conference on Data Mining, 2011, pp. 723–734.

[10] S. Nessa, M. Abedin, W. E. Wong, L. Khan, and Y. Qi, "Software Fault Localization Using N-gram Analysis," in Proceedings of the Third International Conference on Wireless Algorithms, Systems, and Applications. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 548–559.

[11] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying Bug Signatures Using Discriminative Graph Mining," in Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. New York, NY, USA: ACM, 2009, pp. 141–152.

[12] 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.

[13] S. Nijssen and J. N. Kok, "A Quickstart in Frequent Structure Mining Can Make a Difference," in Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. New York, NY, USA: ACM, 2004, pp. 647–652.

[14] Y. Chi, Y. Yang, and R. R. Muntz, "Indexing and Mining Free Trees," in Proceedings of the Third IEEE International Conference on Data Mining (ICDM), 2003, pp. 509–512.

[15] X. Yan and J. Han, "CloseGraph: Mining Closed Frequent Graph Patterns," in Proc. of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2003, pp. 286–295.

[16] K.-C. Tai, "The Tree-to-Tree Correction Problem," Journal of the ACM, vol. 26, no. 3, Jul. 1979, pp. 422–433.

[17] 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.

[18] 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, 2005, pp. 301–312.

[19] D. Gale and L. S. Shapley, "College Admissions and the Stability of Marriage," American Math. Monthly, vol. 69, no. 1, 1962, pp. 9–15.

[20] D. Gusfield and R. W. Irving, The Stable Marriage Problem: Structure and Algorithms. Cambridge, MA, USA: MIT Press, 1989.

[21] M. S. Pini, F. Rossi, K. B. Venable, and T. Walsh, "Stability, Optimality and Manipulation in Matching Problems with Weighted Preferences," Algorithms, vol. 6, no. 4, 2013, pp. 782–804.

[22] "daisydiff: A Java Library to Compare HTML Files," [retrieved May, 2014]. [Online]. Available: http://code.google.com/p/daisydiff/

[23] "Software & Algorithms, ISSEL," [retrieved May, 2014]. [Online]. Available: http://issel.ee.auth.gr/software-algorithms/

[24] "ParSeMiS: The Parallel and Sequential Graph Mining Suite," [retrieved May, 2014]. [Online]. Available: https://www2.cs.fau.de/EN/research/zold/ParSeMiS/

[25] 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.