View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by KITopen

Mining Edge-Weighted Call Graphs to Localise Software Bugs

Frank Eichinger, Klemens Böhm, and Matthias Huber

Institute for Program Structures and Data Organisation (IPD),
Universität Karlsruhe (TH), Germany, `{eichinger, boehm, huberm}@ipd.uka.de`

Abstract An important problem in software engineering is the automated discovery of noncrashing occasional bugs. In this work we address this problem and show that mining of weighted call graphs of program executions is a promising technique. We mine weighted graphs with a combination of structural and numerical techniques. More specifically, we propose a novel reduction technique for call graphs which introduces edge weights. Then we present an analysis technique for such weighted call graphs based on graph mining and on traditional feature selection schemes. The technique generalises previous graph mining approaches as it allows for an analysis of weights. Our evaluation shows that our approach finds bugs which previous approaches cannot detect so far. Our technique also doubles the precision of finding bugs which existing techniques can already localise in principle.

1 Introduction

Software quality is a big concern in industry. Almost any software displays at least some minor bugs after being released. Such bugs incur significant costs. A class of bugs which is particularly hard to handle is *noncrashing occasional bugs*, i.e., failures which lead to faulty results with some but not with any input data. Noncrashing bugs in general are already hard to find. This is because no stack trace of the failure is available. With occasional bugs, the situation is even more difficult, as they are harder to reproduce. Developers usually try to find and fix bugs by doing an in-depth code review along with testing and classical debugging. Since such reviews are very expensive, there is a need for tools which localise pieces of code that are more likely to contain a bug.

Research in the field of software reliability has been extensive, and various techniques have been developed for locating bugs. Static techniques require a large bug and version history database, which is not always available. Dynamic techniques using instrumentation often have a poor runtime behaviour. Another dynamic technique is the analysis of *call graphs*. Such a graph reflects the invocation structure of a particular program execution. Without any further treatment, a call graph is a *rooted ordered tree*. The `main()` method[1] of a program usually is its root, and all methods invoked directly are its children. Figure 1(a) is an

[1] In this paper, we use *method* interchangeably with *function*.

abstract example of such a call graph. Recent work [1, 2] deploys *graph mining* techniques on call graphs for bug localisation. [2] then derives a ranking of methods which are most probable to contain a bug. Generating such a ranking is not trivial. For instance, follow up bugs need to be identified. [2] does not identify follow up bugs at all, and [1] only generates a backtrace-like structure which helps the programmer to find follow up bugs.

Graph mining is a relatively new discipline in data mining, and innovative algorithms have been developed in recent years [3, 4, 5, 6]. Various algorithms deal with the problem of *mining frequent subgraphs*, i.e., discovering all subgraphs which are frequent in a set of graphs. A difficulty with graph mining on raw call graphs is that the algorithms do not scale. Therefore, reduction techniques are developed and applied first. Such techniques are not obvious: They involve a trade-off between loss of information and the size of the resulting graphs. An important piece of information included in the raw call graphs is the call frequency of all methods. The reduction techniques in [1, 2] loose this information. But it eases detection of bugs which affect the number of invocations of a method, *call frequency affecting bugs*. Various reasons for such bugs exist, e.g., wrongly specified conditions. Note that it is not only loop conditions which cause these bugs, but any wrongly specified condition leading to method calls within in a loop. This is because branches taken within a loop or not affect the frequency of a certain method call. As iterations are elementary in programming languages, a wide range of bugs is call frequency affecting. To find such bugs, we take call frequencies into account and analyse their differences in correct and failing executions.

Graph mining research has focused on structural and categorical techniques, and the various graph miners available target at different kinds of graphs. Almost all algorithms handle categorical data in node and edge labels. However, little attention has gone into the analysis of quantitative information, and no algorithm is available for mining *weighted graphs*. Since we want to analyse call graphs where weighted edges represent call frequencies, we must come up with a solution. Further, finding a suitable combination of call graph reduction, graph mining and the analysis of call frequencies is challenging.

In this work, we use conventional mining techniques for unweighted graphs in conjunction with feature selection algorithms to analyse numerical edge weights. More specifically, we first trace program executions and classify them as correct or failing using a test oracle. We for our part do this by comparing execution results to a fault-free reference. These correct results are typically available, as test suites providing such information are widely used in quality assurance [7]. We represent the program traces as call graphs and reduce these graphs by deleting multiple method calls caused by iterations and introducing edge weights representing call frequencies. We then mine the reduced graphs before taking the edge weights into account: By applying an entropy based feature selection algorithm to the weights of the different edges, we calculate the likelihood of both every method invocation as well as of every method containing a bug. For a final ranking, we combine these likelihoods with another score based on structural

properties of the graph mining results. The rationale is that this ranking is given to a software developer who can do a code review of the suspicious methods.

In other words, solving the problems mentioned so far requires innovations at different levels of analysis. Our contributions are as follows:

Reduction of software call graphs. We propose using a new variant of reduced call graphs and present a technique to accomplish this reduction. The reduced graphs keep much information, while they are relatively small. For example, [2] would reduce one call graph from our evaluation from 9,946 to 50 edges. With our technique, there are just 31 edges, while keeping more information and, ultimately, giving way to better results.

Mining weighted graphs. We present an approach which combines numerical analysis of edge weights with conventional graph mining. Our approach distinguishes between occurrences of edges which appear more than once within a subgraph. This allows for a detailed analysis. To our knowledge, a technique which analyses weights in the postprocessing of graph mining has not been described before. This particular contribution is not limited to call graph analysis, but can be transferred to any domain where weighted graphs are present.

Combination of numerical and structural techniques. There exist frequent subgraphs which occur both in call graphs of correct and of failing executions, as well as frequent subgraphs occurring in failing executions only. We analyse the edge weights for subgraphs from the first class, whereas we generate purely structural evidence from subgraphs from the second class. Our approach then combines these different kinds of evidence, and we demonstrate its usefulness.

Evaluation. We show that our approach is particularly well suited to discover call frequency affecting bugs. Unlike previous techniques, it also detects follow up bugs. With regard to bugs existing techniques can already localise, our approach doubles the precision of finding them. Furthermore, it finds bugs on the granularity of method invocations instead of the level of methods.

In this work, we do not reduce recursive calls and concentrate on iterations. The reduction of recursions is not obvious and is beyond the scope of this study.

Paper outline: Section 2 reviews related work. Section 3 presents an overview of graph mining with call graphs. Section 4 discusses graph reduction techniques. Section 5 describes how to calculate probabilities of containing bugs based on reduced graphs. Section 6 features an evaluation. Section 7 concludes.

2 Related Work

A lot of research has been done in the field of software reliability. Approaches range from static code analysis and mining of software repositories and bug databases [8, 9, 10] to dynamic program verification. The latter focus on the data flow [11, 12] or, like all call graph based techniques, on the control flow [13, 14]. In the following, we will first discuss the application of data mining techniques in this context – bug localisation is just one application. Then we concentrate on two graph mining based approaches [1, 2] which are most related to our work. Finally, we describe some related work in the area of mining weighted structures.

2.1 Mining software metrics and invariants

[8] maps post-release failures from a bug database to defects in static source code. Using standard complexity measures from software engineering, the source code is mined with regression models, which can then predict post-release failures for new software entities. A similar study uses decision trees to predict failure probabilities [9]. [10] uses regression techniques to predict the likelihood of bugs based on static usage relationships between software components. All approaches mentioned require a large collection of bugs and version history.

Dynamic program slicing [11] gives hints which parts of a program might have contributed to a faulty execution, without ranking the locations in question. This is done by discovering all statements that actually affect the variables involved. Advanced techniques like [12] perform dynamic data flow focused analysis by instrumenting the source code to gain program invariants. These are used as features of correct and failing executions which are analysed with regression techniques. This leads to potentially faulty pieces of code. A similar approach, but with a focus on the control flow, is [13]. It instruments condition statements and calculates a ranking based on its evaluation frequencies. The instrumentation based approaches mentioned either suffer from poor runtime behaviour or miss bugs if only sampled parts of the software are instrumented.

[14] is a technique which uses tracing and visualisation. It relies on a simple ranking of program components based on the information which components are executed more often in failing program executions. This ranking serves as a basis for more sophisticated rankings in [2] as well as in our approach.

2.2 Call Graph based Fault Detection

The approach from Liu et al. [1]: This first study which applies graph mining techniques to dynamic call graphs considers so called *software behaviour graphs*. These are reduced call graphs, augmented with some temporal information. Section 4 will provide more information on these graphs. The behaviour graphs representing correct and failing program executions are mined with a variant of the *CloseGraph* algorithm [5]. This step results in frequent subgraphs which are used as binary features characterising a program execution: A boolean feature vector represents every execution. In this vector, every element indicates if a certain subgraph is included in the corresponding behaviour graph. Using those feature vectors, a support vector machine is learned which decides if a program execution is correct or failing. More precisely, for every method, two classifiers are learned: one based on behaviour graphs including the respective method, one based on graphs without it. If the precision rises significantly when adding graphs containing a certain method, this method is deemed more likely to contain a bug. Experiments with five out of 130 bugs from the Siemens Programs [15] demonstrate good classification performance, but do not evaluate the precision of the bug localisation. Furthermore, the authors do not generate a ranking of methods suspected to contain a bug. As we do so, our approach can not be compared directly. However, in Sect. 6, we compare the reduction techniques.

The approach from Di Fatta et al. [2]: In this work, a reduction technique is again applied to the raw call trees first (see Sect. 4 for details). The next step is similar to the one described before: A collection of reduced call graphs representing correct and failing program executions is analysed with graph mining. The authors use the tree miner *FREQT* [16] to find all frequent subtrees. The call trees analysed are large and lead to scalability problems of the algorithm. Hence the authors limit the size of the subtrees searched to values up to 4. Then the authors identify which resulting subgraphs are frequent in the set of failing program executions, but not frequent in the set of correct ones. This set of subgraphs is called *specific neighbourhood*. For all methods invoked within subgraphs which are part of the specific neighbourhood, a probability of containing a bug is calculated based on support values. Like [1], [2] does not put attention on call frequencies.

2.3 Subsumption

Frequent subgraph mining is a generalisation of previous structural knowledge discovery techniques such as mining of itemsets, sequences and trees [17]. Early work [18] in the area of itemsets has introduced the problem of *weighted structure mining*. Itemsets can be seen as graphs consisting of nodes only. In [18], these nodes are weighted and are discretised during preprocessing. Corresponding techniques are applied to graphs in transportation networks [19] and image analysis [20]. Such discretisation leads to a loss of information, as we will discuss in Sect. 5.2. In [21], we have already analysed tuples of weights of sequences as a postprocessing step of sequence mining. This allows for a more detailed analysis of weights in different structural contexts. This current work is in the field of weighted structure mining as well – it analyses edge weights subsequent to a graph mining step.

[22] is a preliminary and much shorter version of this paper. It directly combines its results with those of [2] in order to find a wider range of bugs. It requires two costly graph mining steps. This current work avoids this.

3 Call Graph Mining Overview

Before we focus on reduction and ranking techniques in Sections 4 and 5, we now give an overview of the procedure of localising bugs with graph mining. Note that [2] follows this general procedure as well. Algorithm 1 first assigns a class (*correct*, *failing*) to every program trace (Line 4), using a test oracle. Then every trace is reduced (Line 5), which leads to smaller call graphs. Techniques to do so are discussed in Sect. 4. Now frequent subgraphs are mined (Line 7). For this step, several algorithms, e.g., tree mining or graph mining in different variants, can be used. The last step is calculating a likelihood of containing a bug. This can either be fine granular for every method invocation or, more coarse grained, for every method (as shown in Line 8). The calculation of the likelihood is based on the frequent subgraphs mined and facilitates a ranking of the methods, which can then be given to the software developer.

Algorithm 1 Generic graph mining based bug localisation procedure.

1: **Input:** a collection of program traces $t \in T$
2: $G = \emptyset$ // initialise a collection of reduced graphs
3: **for all** traces $t \in T$ **do**
4: assign a $class \in \{correct, failing\}$ to t
5: $G = G \cup \{reduce(t)\}$
6: **end for**
7: $SG = frequent_subgraph_mining(G)$
8: calculate $P(m)$ for all methods m; based on SG

4 Call Graph Reduction

In the related work on call graph mining (see Sect. 2.2), two different approaches exist which lead to reduced call graphs (as specified in Line 5 of Algorithm 1). The reduction technique used in [1] projects every node representing the same method in the call graph to a single node in the reduced graph. We call this technique *total reduction*. Note that this may give way to the existence of loops and limits the size of the graph (in terms of nodes) to the total number of methods in the program analysed. As an example, the raw ordered call tree in Fig. 1(a) would result in the reduced graph displayed in Fig. 1(b). In addition to the reduction, so called *temporal edges* are inserted between all methods which are executed consecutively and are invoked by the same method. This technique integrates the temporal order from the raw ordered call trees into the graph representations. Technically, temporal edges are directed edges having another label, e.g., "temporal", compared to other edges which are labelled, say, "call". Figure 1(c) serves as an example of a graph using the reduction technique and temporal edges (dotted), called *software behaviour graph*. This reduction is rather severe, e.g., from several millions of nodes to several hundreds. It allows graph mining based bug localisation even with larger software projects. In contrast, much information about the program execution is lost. This concerns frequencies of the execution of methods as well as information on different structural patterns within the graphs. In particular, the information in which context a certain substructure is executed is lost (see Sect. 5.2). Furthermore, the tempo-

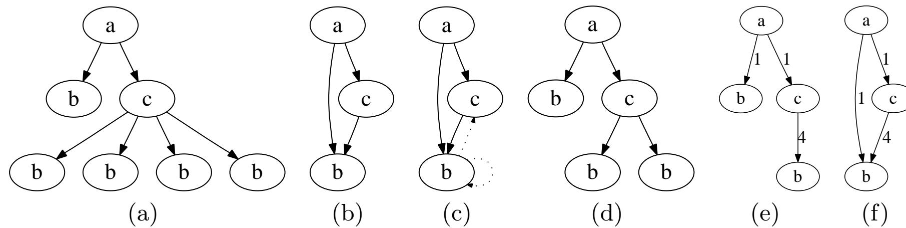

Figure 1. Variants of reduced call graphs.

ral edges increase the size of the graphs significantly. An increased precision of fault detection by using temporal edges has not been evaluated.

The approach in [2] keeps more information. It omits substructures of subsequent executions, which are invoked more than twice in a row from the same node. See Fig. 1(d) for an example. This reduction ensures that many equal substructures called within a loop do not lead to call graphs of an extreme size. In contrast, the information that some substructure is executed several times is still encoded in the graph structure, but without exact numbers. Compared to [1], much more information about a program execution is kept, compromised by a call graph which is generally much larger. For example, graphs are reduced from several millions of nodes to several ten thousand nodes.

In our approach, we try to overcome the shortcomings of both approaches and try to keep most of the information available. We reduce subtrees executed iteratively by deleting all but the first one and inserting the call frequencies as edge weights. This makes the graphs relatively small and keeps a lot of information. An example is given in Fig. 1(e). The introduction of edge weights allows for a detailed analysis. If, e.g., a bug is hidden in a loop condition, this might lead to hundreds of iterations of the loop, compared to just a few in correct program executions. Note that, with both previous graph representations, the graph of the correct and of the failing execution is reduced to exactly the same structure in this case. In our approach, the edge weights would be significantly different. Analysis techniques can then discover this.

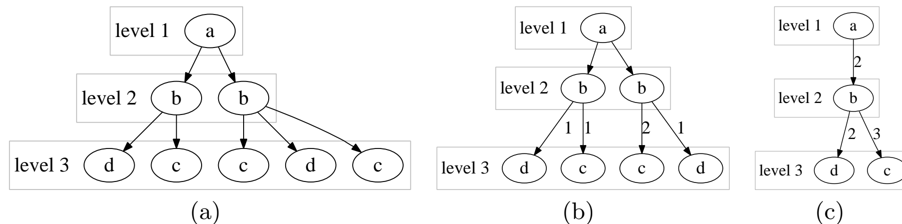

Figure 2. A raw call tree, its first and second transformation step.

For our graph reduction approach, which implements the functionality specified in Line 5 of Algorithm 1, we organise the call tree into n horizontal levels. The root node is in *level* 1, all other nodes are in subsequent levels, increasing with the distance to the root. See Fig. 2(a). A naïve approach to reduce our example call tree in Fig. 2(a) would be to start at *level* 1 with Node a. There, one would find two child subtrees having a different structure – one could not merge anything. Therefore, we work level by level, starting from *level* $n-1$, as described in Algorithm 2. In our example in Fig. 2(a), we have to start in *level* 2. The left node b has two different children. Thus, nothing can be merged there. In the right b, the two children c are merged by adding the edge weights of the merged edges, yielding the tree in Fig. 2(b). In the next level, *level* 1, we process

the root node a. Here, the structure of the two succeeding subtrees is the same. We merge them, resulting in the tree in Fig. 2(c).

Algorithm 2 Subtree reduction algorithm.

1: **Input:** a call tree organised in n levels
2: **for** $level = n - 1$ **to** 1 **do**
3:     **for each** *node* **in** *level* **do**
4:         merge all identical child-subtrees of *node*, sum up corresponding edge weights
5:     **end for**
6: **end for**

Our reduction technique obviously is not lossless. The size of the resulting graphs would be prohibitive. With large pieces of software, graph mining may not scale with the size of the call graphs, even if the reduction technique applied is effective. To deal with this problem, it seems promising to use graphs of coarser granularity: Instead of using methods as nodes in call graphs, it is possible to, say, use classes as a coarser abstraction. Such call graphs would have classes as nodes and inter class method calls as edges. Obviously, such a coarser abstraction would lead to less detailed bug localisation as well.

5 The Ranking Framework

So far, we have discussed how to reduce call graphs. Now we will describe our framework for deriving a ranking of potentially buggy method invocations (edges) and methods (nodes) from such graphs. Before we focus on the individual components in the following subsections, we give an overview of our framework. At first, we apply frequent subgraph mining to the reduced call graphs, without considering the weights for now (Sect. 5.1). This corresponds to Line 7 in Algorithm 1. We then partition the set of frequent subgraphs just mined and consider two sets separately: (1) the set of subgraphs which occur in both correct and failing executions SG_{cf} and (2) the set of subgraphs which only occur in failing executions SG_f[2]. We use SG_{cf} to build a ranking based on the differences in edge weights in correct and failing executions (Sect. 5.2). This ranking can be based on method invocations or on individual methods. As subgraphs in SG_f are never contained in correct executions, it is not possible to analyse differences based on SG_f. In contrast, SG_f provides crucial information about the graph structures in failing executions. Therefore, we derive a score based on the support in SG_f (Sect. 5.3) and combine it with the edge weight based one mentioned above (Sect. 5.4). This combination is in Line 8 in Algorithm 1.

[2] In preliminary experiments, we have also evaluated the influence of subgraphs which occur in correct executions only. It has turned out that such graphs do not help to localise bugs.

5.1 Graph Mining Step

After having reduced the call graphs gained from correct and failing program executions, we search for frequent closed subgraphs SG in the graph dataset G using the *CloseGraph* algorithm [5]. For this step, we employ the *ParSeMiS* graph mining suite[3]. Closed mining reduces the number of graphs in the result set significantly and increases the performance of the mining algorithm (studying its effects on result quality is beyond the scope of this article). Furthermore, the usage of a general graph mining algorithm instead of a tree miner allows for comparative experiments with other graph reduction techniques (see Sect. 6). After the graph mining step, we partition SG and derive the set of subgraphs which occur in correct and failing executions SG_{cf} and the set of subgraphs which occur in failing executions only SG_f.

5.2 Entropy Based Scoring

We now focus on frequent subgraphs which occur in both correct and failing executions (SG_{cf}). Our goal is to develop an approach which discovers which edge weights of call graphs from a program are most significant to discriminate between *correct* and *failing*. To do so, one possibility is to consider different *edge types*, e.g., edges having the same calling method m_s (start) and the same method called m_e (end). However, edges of one type can appear more than once within one subgraph and, of course, in several different subgraphs. Therefore, we analyse every edge in every such location, which we refer to as a *context*. To specify the exact location of an edge in its context within a certain subgraph, we do not use the method names, as they may occur more than once. Instead, we use a unique id for the calling node (id_s) and another one for the method called (id_e). All ids are valid within its subgraph. To sum up, we reference an edge in its context in a certain subgraph sg with the following tuple: (sg, id_s, id_e). A certain bug does not affect all method calls (edges) of the same type, but method calls of the same type in the same context. Therefore, we assemble a feature table with every edge in every context as columns and all program executions (represented by their reduced call graphs) as rows. The table cells contain the respective edge weights. The following table serves as an example:

	$a \rightarrow b$ (sg_1, id_1, id_2)	$a \rightarrow b$ (sg_1, id_1, id_3)	$a \rightarrow c$ (sg_2, id_1, id_2)	$\cdots$	Class
g_1	445	21	7	$\cdots$	*failing*
g_2	0	0	4	$\cdots$	*correct*
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

The first column corresponds to the first subgraph (sg_1) and the edge from id_1 (method a) to id_2 (method b). The second column corresponds to the same subgraph (sg_1) but to the edge from id_1 (method a) to id_3 (method b). The third column represents an edge from id_1 to id_2 in the second subgraph (sg_2). Note

[3] http://www2.informatik.uni-erlangen.de/Forschung/Projekte/ParSeMiS/

that method b occurs twice in sg_1 and that ids have different meanings in sg_1 and sg_2. The last column contains the class *correct* or *failing*. The rows correspond to all reduced call graphs $g_1, ..., g_n \in G$ available. If a certain subgraph is not contained in a call graph, the corresponding cells have value 0, like g_2 which does not contain sg_1. Graphs (rows) can contain a certain subgraph not just once, but several times at different locations. In this case, we use aggregates in the corresponding cells of the table. As minimum values would ignore bugs resulting in increased numbers and maximum values would ignore bugs leading to reduced numbers, we use the average. In the example, sg_2 is embedded at two locations in g_1. In one location the edge from id_1 to id_2 has the weight 6, in the other one weight 8.

The table structure described allows for a detailed analysis of edge weights in different contexts within a subgraph. All following steps in this subsection are described in Algorithm 3. After assembling the table, we employ a standard feature selection algorithm to score the columns of the table and thus the different edges. We use an entropy based algorithm from the *Weka* data mining suite [23] which calculates the information gain *InfoGain* [24] (with respect to the class of the executions, *correct* or *failing*) for every column (Line 2 in Algorithm 3). The information gain is a value between 0 and 1 which we interpret as a likelihood of being responsible for bugs. Columns with an information gain of 0, e.g., the edges always have the same weights in both classes, are discarded immediately (Line 3 in Algorithm 3).

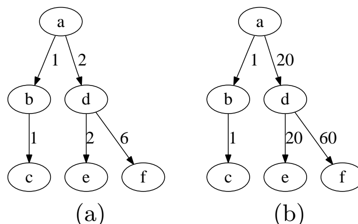

Figure 3. Follow up bugs.

Call graphs of failing executions frequently contain bug-like patterns which are caused by a preceding bug. We call such patterns *follow up bugs* and remove them from our ranked list of features. Figure 3 illustrates a follow up bug: (a) represents a bug free version, (b) contains a bug in method a where it calls method d. Here, this method is called 20 times instead of twice. Following our reduction technique, this leads to the same (or a proportional) increase in the number of calls in method d. In our entropy based ranking, the edges $d \rightarrow e$ and $d \rightarrow f$ inherit the score from $a \rightarrow d$ if the scaling of the weights is proportional. Thus, we interpret these two edges as follow up bugs and remove them from our ranking. More formally, we remove edges if the edge leading to its direct parent within the same subgraph has the same entropy score (Line 4 in Algorithm 3).

In case of more than one bug in a program, this way of follow up bug detection might not find all such bugs, but preliminary experiments have shown that it does detect common cases efficiently. We leave aside the pathological case that this technique classifies a real bug as follow up bug. This is acceptable, since the probability of a certain entropy value is the same for every bug. Therefore, it is very unlikely that two unrelated bugs lead to exactly the same entropy value, which would lead to a 'false positive' classification.

Algorithm 3 Procedure to calculate $P_e(m_s, m_e)$ and $P_e(m)$.

1: **Input:** a set of edges $e \in E$ representing edges in their context, $e = (sg, id_s, id_e)$
2: assign every $e \in E$ its information gain *InfoGain*
3: $E = E \setminus \{e \mid e.InfoGain = 0\}$
4: // remove follow up bugs:
$E = E \setminus \{e \mid \exists p : p \in E, p.sg = e.sg, p.id_e = e.id_s, p.InfoGain = e.InfoGain\}$
5: $E_{(m_s,m_e)} = \{e \mid e \in E \wedge e.id_s.label = m_s \wedge e.id_e.label = m_e\}$
6: $P_e(m_s, m_e) = \max_{e \in E_{(m_s,m_e)}} (e.InfoGain)$
7: $E_m = \{e \mid e \in E \wedge e.id_s.label = m\}$
8: $P_e(m) = \max_{e \in E_m} (e.InfoGain)$

Now we calculate likelihoods of containing a bug for every method invocation (described by a calling method m_s and a method called m_e). We call this score $P_e(m_s, m_e)$ as it is based on entropy. To do so, we first determine sets $E_{(m_s,m_e)}$ of edges $e \in E$ for every method invocation in Line 5 of Algorithm 3. In Line 6, we use the max() function to calculate $P_e(m_s, m_e)$, the maximum of all edges (method invocations) in $E_{(m_s,m_e)}$. This is necessary, as in general there are many edges in E with the same method invocation. This is because an invocation can occur in different contexts. With the max() function, we assign every invocation the score from the context ranked highest. Lower scores for the same invocation are less important, and we ignore them.

At this point, the ranking does not only provide the score for a method invocation, but also the subgraphs where it occurs and the locations within it. This information might be important for a software developer. We report this information additionally. As we also want to compare our results to those of [2] which does not provide information on the invocation level, we also calculate $P_e(m)$ for every calling method m in Lines 7 and 8 of Algorithm 3. The explanation is analogous to the one of the calculation of $P_e(m_s, m_e)$ in Lines 5 and 6.

This subsection has presented a technique relying on the analysis of edge weights in different contexts. As we will see in the evaluation (Sect. 6), the consideration of different contexts is key for good results. As contexts are defined based on the subgraphs mined, such a differentiated analysis is only possible subsequent to graph mining, but not during preprocessing (see Sect. 2.3).

5.3 Structural Scoring

Our entropy based scoring (Sect. 5.2) cannot detect bugs which are not call frequency affecting, as it analyses call frequencies only. At the same time, it does not consider subgraphs which occur in failing executions only (SG_f). As some bugs result in such subgraphs, these are essential to detect bugs as well. Therefore, we calculate the score $P_s(m)$ for individual methods based on the support in SG_f. This score is another likelihood of containing a bug, as it refers to the frequency of method invocations in failing executions.

$$P_s(m) = \mathit{supp}(m, SG_f) \tag{1}$$

where $\mathit{supp}(m, SG_f)$ is the fraction of graphs in SG_f containing a node m.

5.4 Combination

Now we calculate the overall likelihood $P(m)$ of containing a bug for every method m, based on the average of the normalised values for $P_e(m)$ (see Sect. 5.2) and $P_s(m)$ (see Sect. 5.3). Normalisation is necessary: While both values are in the [0, 1]-range, their maximum can be very different. Normalisation keeps us from overemphasising one of the two rankings. $P(m)$ is the basis for the ranking of all methods m, which is used to locate bugs:

$$P(m) = \frac{P_e(m)}{2 \max\limits_{n \in t \in T}(P_e(n))} + \frac{P_s(m)}{2 \max\limits_{n \in t \in T}(P_s(n))} \tag{2}$$

where n is a method in a program trace t in the collection of all traces T.

6 Evaluation

To evaluate bug localisation techniques, the *Siemens Programs* [15] are often used [1, 2, 13] as a reference suite of C programs artificially instrumented with different bugs. More specifically, it usually is just a small subset of this benchmark which is used. For example, [2] just considers three of the seven Siemens Programs, [1] only five different bugs out of 130 available in total. As most bug detection techniques are limited to certain classes of bugs, these techniques cannot find every element of a standard suite of bugs. Our approach, as well as the two most related ones [1, 2], focus on noncrashing occasional bugs.

As we rely on Java software, we use a well known Java diff tool (taken from [25]) and instrument it with fourteen different bugs[4]. To do so, we have examined the Siemens Programs and have identified five types of bugs which are most frequent within them. Our programs contain these bugs and also the kinds of bugs used in [1]. Most of these bugs are call frequency affecting. The Siemens

[4] We provide the software versions containing the bugs used at `http://www.ipd.uka.de/~eichi/papers/eichinger08mining/`

Programs mostly contain bugs in single lines and just a few programs with more than one bug. To mimic the Siemens Programs as close as possible, we have instrumented only two out of fourteen versions (Bugs 7 and 8) with more than one bug. We give an overview of the kinds of bugs used in the following table:

Version	Description
Bug 1, Bug 10	Wrong variable used
Bug 2, Bug 11	Additional or-condition
Bug 3	`>=` instead of `!=`
Bug 4, Bug 12	`i+1` instead of `i` in array access
Bug 5, Bug 13	`>=` instead of `>`
Bug 6	`>` instead of `<`
Bug 7	A combination of Bug 2 and Bug 4 (in the same line)
Bug 8	`i+1` instead of `i` in array access + additional or condition
Bug 9, Bug 14	Missing condition

Every version has been executed exactly 100 times with different input data. The results have been classified as correct or failing executions with a test oracle based on a bug free reference version. Based on this data, we carry out four experiments:

1. The *conventional method ranking*, following [2], including its graph reduction technique, using the same method scoring and the same mining parameters for support and maximum subgraph size.
2. The *total reduction method ranking*, the total reduction from [1] with edge weights representing call frequencies (see Fig. 1(f)) together with the combined scoring (Sect. 5.4).
3. The *entropy-based method ranking*, our reduction technique (Sect. 4) together with the entropy based scoring (Sect. 5.2) but without the combination.
4. The *combined method ranking* (Sect. 5.4), our reduction technique (Sect. 4) together with the entropy based scoring (Sect. 5.2) and the structural scoring (Sect. 5.3).

To keep the comparison focused, we leave aside the temporal order inside the call graphs. We have found graphs with temporal edges as used in [1] too large (in terms of edges) to be mined efficiently. Nevertheless, our study is fair since we leave aside that temporal information with all alternatives.

All experiments produce ordered lists of methods. A software developer doing a code review would start with the top ranked method in such a list. The maximum number of methods which have to be checked to find the bug is, therefore, the line number of the faulty method in the ranked list. Sometimes two or more subsequent lines have the same score. As the intuition is to count the maximum number of methods which have to be checked, all lines with the same score have the number of the last line with this score.

In order to evaluate the accuracy of the results, the line of the ranking where the first instrumented bug is found needs to be identified. If the first instrumented bug is, e.g., reported in the third line, this is a fairly good result. A software

developer only has to do a code review of maximally three out of 25 methods from our target program. Furthermore, in the entropy-based and the combined method ranking, there usually is more information available where a bug is located within a method and in which context it appears. Thus, our comparison is conservative, i.e., it does not demonstrate the full capabilities of our approach.

We present the results (the number of the first line in which a bug is found) of the four experiments for all fourteen bugs in the following table. For versions with two bugs, we concentrate on finding the first of the two bugs contained. Value 25 refers to a bug which is not discovered with the respective approach.

Experiment \ Bug Version Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Conventional method ranking (1)	3	25	1	4	6	4	3	3	1	6	4	4	25	4
Total reduction method ranking (2)	1	5	1	4	3	5	5	2	25	2	5	4	6	3
Entropy-based method ranking (3)	3	3	1	1	1	3	3	1	25	2	3	3	3	3
Combined method ranking (4)	3	2	1	1	1	2	2	1	8	2	2	3	3	3

Comparing the results from (1) and (3), the entropy-based approach almost always performs as good or better than the conventional one. This shows that analysing numerical call frequencies is adequate to locate bugs. Bugs 2, 9 and 13 illustrate that both approaches alone cannot find certain bugs. Bug 9 can not be found by comparing call frequencies (3), as a condition has been modified which now always leads to the invocation of a certain method. Therefore, the call frequency is always the same (not call frequency affecting). Bugs 2 and 13 are not found with the purely structural conventional approach (1). Both are typical call frequency affecting bugs: Bug 2 is in a condition inside a loop and leads to more invocations of a certain method. In Bug 13, a modified for-condition slightly changes the call frequency of a method inside the loop. With the reduction technique used in (1), this leads in Bug 2 and Bug 13 to the same graph structure both with correct and with failing executions. Thus, it is not possible to identify structural differences.

To ease presentation, we first describe the combined approach (4) before we discuss (2). Experiment (4) is intended to take important structural information into account as well and therefore to improve the results from (3). We do achieve this goal: We retain the already good results from (3) in nine cases and improve them in five. In particular, (4) finds Bug 9, which is not possible with (3) alone. Therefore, the combination of numerical and structural techniques turns out to be superior. When calculating averages of the improvement of certain approaches in the following, we leave aside Bugs 2, 9 and 13. This is because high values would have a too strong influence on the results. Note that this lets our approach look somewhat worse, as it does find all three bugs. Bugs 7 and 8 illustrate that our approach analyses versions with more than one bug successfully as well. In (4), both bugs are ranked at Position 1 and 2. This is not worse than versions with one bug only.

Experiment (2) is intended to evaluate the graph reduction technique in [1]. Except for the graph reduction technique, the approach is identical to the one in (4). In almost all cases, (2) performs worse than (4). This confirms that our graph

reduction technique is reasonable and that it is worth to keep more structural information than the total reduction does.

Summing up, our experiments show that weighted graph mining on call graphs reduced with our method is appropriate for precise software bug localisation. Even if previous approaches are able to detect call frequency affecting bugs, our approach can detect them with a much higher degree of precision. This is because it explicitly analyses call frequencies. Furthermore, our approach finds bugs in settings with more than one bug and doubles the precision of [2] on average.

7 Conclusions

In this work we have addressed the problem of localising noncrashing occasional software bugs. This localisation is important as such bugs are hard to detect manually and cause significant costs. Our approach is dynamic and control flow centred as it relies on call graphs of program executions. We have presented a novel technique to reduce such graphs. It keeps the size of the resulting graphs relatively small while keeping more important information. In particular, it introduces edge weights representing call frequencies. As none of the recently developed graph mining algorithms analyse weighted graphs, we have developed a combined approach which does so. It consists of conventional frequent subgraph mining and subsequently scoring of numerical edge weights using an entropy based algorithm. Our experiments do not just show a doubled precision of bug localisation. They also show that our approach detects bugs which previous approaches can not find in principle. We demonstrate that the numerical information kept with our call graph reduction technique is important for good results. We have shown that our combination of structural and numerical mining techniques is key for precise localisations.

Future work will address recursive method invocations. Another direction is mining of call graphs with constraint based and approximative techniques.

References

[1] Liu, C., Yan, X., Yu, H., Han, J., Yu, P.S.: Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs. In: Proc. of the 5th Int. Conf. on Data Mining (SDM). (2005)

[2] Di Fatta, G., Leue, S., Stegantova, E.: Discriminative Pattern Mining in Software Fault Detection. In: Proc. of the 3rd Int. Workshop on Software Quality Assurance (SOQUA). (2006)

[3] Borgelt, C., Berthold, M.R.: Mining Molecular Fragments: Finding Relevant Substructures of Molecules. In: Proc. of the 2nd Int. Conf. on Data Mining (ICDM). (2002)

[4] Yan, X., Han, J.: gSpan: Graph-Based Substructure Pattern Mining. In: Proc. of the 2nd Int. Conf. on Data Mining (ICDM). (2002)

[5] Yan, X., Han, J.: CloseGraph: Mining Closed Frequent Graph Patterns. In: Proc. of the 9th Int. Conf. on Knowledge Discovery and Data Mining (KDD). (2003)

[6] Nijssen, S., Kok, J.N.: A Quickstart in Frequent Structure Mining Can Make a Difference. In: Proc. of the 10th Int. Conf. on Knowledge Discovery and Data Mining (KDD). (2004)
[7] Harrold, M.J., Gupta, R., Soffa, M.L.: A Methodology for Controlling the Size of a Test Suite. ACM Transactions on Software Engineering and Methodology (TOSEM) **2**(3) (1993) 270–285
[8] Nagappan, N., Ball, T., Zeller, A.: Mining Metrics to Predict Component Failures. In: Proc. of the 28th Int. Conf. on Software Engineering (ICSE). (2006)
[9] Knab, P., Pinzger, M., Bernstein, A.: Predicting Defect Densities in Source Code Files with Decision Tree Learners. In: Proc. of the Int. Workshop on Mining Software Repositories (MSR) at ICSE. (2006)
[10] Schröter, A., Zimmermann, T., Zeller, A.: Predicting Component Failures at Design Time. In: Proc. of the 5th Int. Symposium on Empirical Software Engineering. (2006)
[11] Korel, B., Laski, J.: Dynamic Program Slicing. Information Processing Letters **29**(3) (1988) 155–163
[12] Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug Isolation via Remote Program Sampling. ACM SIGPLAN Notices **38**(5) (2003) 141–154
[13] Liu, C., Yan, X., Han, J.: Mining Control Flow Abnormality for Logic Error Isolation. In: Proc. of the 6th Int. Conf. on Data Mining (SDM). (2006)
[14] Eagan, J., Harrold, M.J., Jones, J.A., Stasko, J.: Technical Note: Visually Encoding Program Test Information to Find Faults in Software. In: Proc. of the Symposium on Information Visualization (INFOVIS). (2001)
[15] Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria. In: Proc. of the 16th Int. Conf. on Software Engineering (ICSE). (1994)
[16] Asai, T., Abe, K., Kawasoe, S., Arimura, H., Sakamoto, H., Arikawa, S.: Efficient Substructure Discovery from Large Semi-structured Data. In: Proc. of the 2nd Int. Conf. on Data Mining (SDM). (2002)
[17] Han, J., Cheng, H., Xin, D., Yan, X.: Frequent Pattern Mining: Current Status and Future Directions. Data Mining and Knowledge Discovery **15**(1) (2007) 55–86
[18] Srikant, R., Agrawal, R.: Mining Quantitative Association Rules in Large Relational Tables. In: Proc. of the Int. Conf. on Management of Data (SIGMOD). (1996)
[19] Jiang, W., Vaidya, J., Balaporia, Z., Clifton, C., Banich, B.: Knowledge Discovery from Transportation Network Data. In: Proc. of the 21st Int. Conf. on Data Engineering (ICDE). (2005)
[20] Nowozin, S., Tsuda, K., Uno, T., Kudo, T., Bakir, G.: Weighted Substructure Mining for Image Analysis. In: Proc. of the Conf. on Computer Vision and Pattern Recognition (CVPR). (2007)
[21] Eichinger, F., Nauck, D.D., Klawonn, F.: Sequence Mining for Customer Behaviour Predictions in Telecommunications. In: Proc. of the Workshop on Practical Data Mining at ECML/PKDD. (2006)
[22] Eichinger, F., Böhm, K., Huber, M.: Improved Software Fault Detection with Graph Mining. In: Proceedings of the 6th Int. Workshop on Mining and Learning with Graphs (MLG) at ICML. (2008)
[23] Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. Morgan Kaufmann (2005)
[24] Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann (1993)
[25] Darwin, I.F.: Java Cookbook. O'Reilly (2004)