Scalable Graph Analyzing Approach for Software Fault-Localization

Zaynab Mousavian  
Faculty of Computer engineering  
Iran University of Science and Technology  
Tehran, Iran  
zmousavian@comp.iust.ac.ir

Mojtaba Vahidi-Asl  
Faculty of Computer engineering  
Iran University of Science and Technology  
Tehran, Iran  
m_vahidi_asl@iust.ac.ir

Saeed Parsa  
Faculty of Computer engineering  
Iran University of Science and Technology  
Tehran, Iran  
parsa@iust.ac.ir

ABSTRACT
In this paper, a new approach for analyzing program behavioral graphs to detect fault relevant paths is presented. The existing graph mining approaches for bug localization merely detect discriminative sub-graphs between failing and passing runs. However, they are not applicable when the context of a failure is not appeared in a discriminative pattern. In our proposed method, the suspicious transitions are identified by contrasting nearest neighbor failing and passing dynamic behavioral graphs. For finding similar failing and passing graphs, we first convert the graphs into adequate vectors. Then, a combination of Jaccard-Cosine similarity measures is applied to identify the nearest graphs. The new scoring formula takes advantage of null hypothesis testing for ranking weighted transitions. The main advantage of the proposed technique is its scalability which makes it work on large and complex programs with huge number of predicates. Another main capability of our approach is providing the faulty paths constructed from fault suspicious transitions. Considering the weighted execution graphs in the analysis enables us to find those types of bugs which reveal themselves in specific number of transitions between two particular predicates. The experimental results on Siemens test suite and Space program manifest the effectiveness of the proposed method on weighted execution graphs for locating bugs in comparison with other methods.

Categories and Subject Descriptors
D.2 [Software]: Software Engineering; D.2.5 [Software Engineering]: Testing and Debugging

General Terms
Reliability, Experimentation

Keywords
Graph Mining, Weighted Graphs, Software Fault Localization, Null Hypothesis Testing

1. INTRODUCTION
Manual software debugging is an arduous task which requires great time and effort of debuggers. It also does not seem always to be accurate and practical for many complex and large size programs. This has motivated many researchers during past few years to design and develop promising techniques for automating software debugging process [1]. Among automated software bug localization techniques, statistical debugging methods have achieved great success and progress in recent years.

Statistical debugging techniques collect runtime behavior of programs in different failing and passing runs and contrast them to identify the location of bugs. The runtime data are mostly collected from predicates which are designed according to the structure of the program (i.e. branch and loop statements, function calls, etc.). However, many of them consider predicates isolated from each other which reduces their ability to detect particular type of bugs [2]. In some cases, a specific transition from one certain point to another point of a program may cause the program to crash or generate unexpected results [3]. Furthermore, in many situations the context of bug is more needed than just a simple point in the program [4]. By providing the context of bugs, a debugger could easily identify and understand bugs, resulting in better fixing them.

Therefore, it seems to be a good idea to analyze all dynamic transitions among different program points in both failing and passing runs in order to identify the suspicious ones. The result might be faulty sub-graphs as the contexts of bug(s). To achieve this, each program execution trace could be converted to an appropriate dynamic graph. The idea of using graph mining approaches for software debugging has been introduced in recent years [5][6] and a little work has been done in this concept. The main problem with existing graph mining approaches for fault localization is their scalability. They may lose their power for large size programs with a great number of predicates. For such programs the size of the adjacency matrix would be too huge. Although they take advantage of adequate pruning process, finding the best discriminative sub-graph might be too expensive in terms of time and space. Meanwhile, the particular structure of a program is such that, a sub-graph which discriminate failing from passing graphs, is not necessarily the context of bug. Experiments show that in some situations due to nondeterministic nature of faults, the bug signature is not really discriminative which makes the problem more challenging. It is worth noting that, in majority of graph mining approaches, the extracted graphs are assumed to be non-weighted. Since, in some cases, a fault may occur in particular iteration of a transition, it seems important to consider the weight of edges in analyzing execution graphs to find fault relevant transitions.

To resolve the difficulties, in this paper a new graph analyzing technique for bug localization is presented. Instead of finding frequent or discriminative sub-graphs at once using traditional data mining approaches (e.g. entropy based techniques), we first assign scores to fault suspicious edges and then try to find appropriate sub-graphs which contain those edges. In other words,
we work in reverse order of existing graph mining techniques in the field of fault localization. An advantage of this mechanism is its capability to rank sub-graphs according to the overall score of including edges.

To achieve this, the nearest failing and passing execution graphs are contrasted to identify fault relevant transitions which will be used later to construct faulty paths in the program. To this end, first, a directed weighted graph, so called execution graph, is extracted from program execution trace, like our previous work [7]. The execution graph integrates transitions between methods and basic blocks and therefore it is capable to consider the dynamic behavior of program execution in terms of transitions among basic blocks and method calls. The proposed approach is inspired from the nearest neighbor method presented in [8] and finds a predetermined number of similar passing execution graphs with each failing one. The proposed metric for finding similar graphs is based on Cosine and Jacard similarity measures. To use such similarity measures, we convert each execution graph to an appropriate vector. The edges of graphs included in near passing and failing graphs are then ranked according to their presence in failing execution graphs and differences in weights using a null hypothesis testing. Since the edges are analyzed in isolation, the size of program may not harm the effectiveness of the approach. Another advantage of the proposed technique is providing the fault relevant sub-graph as a faulty sub-path, which may be highly informative for the programmer to find the cause of bug and fix it.

In summary, these main contributions are made in this paper:

1- A weighted execution graph is constructed from different failing and passing executions of the program.
2- Providing faulty paths and ranking them according to the scores of including edges helps the programmer to understand the context of the fault(s) easier.
3- The execution graphs are converted to vectors and a combination of Cosine and Jacard similarity measures are used to find similar passing and failing graphs. Therefore, the main novelty is extending the idea of NN method [8] on analyzing graph structure for finding bugs.
4- A novel ranking method is introduced in this paper which considers the behavior of a transition in failing contrast to passing runs using a null hypothesis testing.

The remaining parts of this paper are organized as follows. In section two, the related works in the concept of using graphs for software fault localization are briefly reviewed. A fully description of the proposed graph analyzing technique is explained in section 3. Section 4, contains the results of applying the technique on Siemens test suite and Space program. Section 5 presents the threats to validity. Finally, we portray the concluding remarks and future works in section 6.

2. RELATED WORKS

As mentioned earlier, programmers may prefer to be informed about the context of bugs rather than merely pinpointed distinct predicates. RAPID is a well known work on bug signature identification [4]. It has found a number of bugs which have not been reported in previous works. It scores program statements using Tarantula technique [9] to identify fault suspicious statements. It then converts each execution trace to a sequence and tries to find the longest subsequence which is common in all failing runs containing the identified statements. As a result, the high ranked signatures are reported to the programmer. Although the technique provides contextual information, because of using sequence construct, it does not actually take advantage of graph structure.

In order to provide the context of faults with the aim of finding faulty paths, Jiang and Su [10] proposed a context-aware technique. It first identifies discriminative predicates and then groups them using a clustering method based on their presence across multiple executions. To identify the bug signature, correlated predicates belonging to the same group are connected using control flow graph of the program and the path is generated heuristically. Hence, the faulty paths are approximated heuristically though they may not actually happen in failing executions.

In recent years, many researchers have focused on software bug localization problem by using graph mining techniques [5][6]. Behavior of a program can be summarized as a subset of control flow graph or a call graph. The fault localization technique mines the extracted graphs labeling with correct or incorrect according to the termination state of each corresponding execution.

Finding frequent sub-graphs could be an important phase in some graph mining techniques. In [5][6], the proposed approaches try to find bugs by applying closed frequent pattern mining and use these patterns as features for training a classification framework. But in large scale programs, the number of frequent sub-graphs is large and the mining complexity on this huge set could be high and very time consuming. Furthermore, the majority of these works do not consider the weight of transitions in their analysis.

In [11], some traditional data mining algorithms are applied on weighted call graphs. The frequencies of a method A calling another method B, forms the weight of edge connecting node A to node B. It suggests that the weights of edges in faulty executions play an important role in finding bug suspicious method calls. However, the large granularity is the main problem and in some situations using only method call graphs is not helpful for finding all type of bugs.

According to the fact that frequent pattern mining might be computationally expensive [5][6], discriminative pattern mining technique is suggested by researchers [12][13][14]. Cheng et. al [3], based on LEAP algorithm [14], extracts program behavior graphs in two levels of granularity: basic blocks and method calls. Top-K LEAP is an entropy based algorithm which identifies top k ranked discriminate sub-graphs. The extracted discriminative patterns can separate failing runs from correct ones and provide an informative signature of faults. However, this technique may also suffer from scalability problems. Furthermore, for some non-deterministic bugs which the corresponding signatures are not highly discriminative, the family of discriminative pattern mining techniques might have difficulties.

In previous work [7], we have proposed an entropy based technique, applicable on weighted graphs. But, the edges in each discriminative sub-graph are not ranked and user should examine all the edges to find the actual cause(s) of fault. In addition to non-scalability of [7], the order of executed predicates in faulty sub-graph is not specified. Furthermore, all mentioned graph based bug localization techniques may have problems when the number of failing test cases is much less than the number of passing runs. Therefore, in this paper, instead of using traditional graph mining techniques, we have developed a new ranking model on program execution graphs which focuses on specific aspects of fault localization problem.
3. THE METHOD OVERVIEW

The proposed graph analyzing technique, converts each execution graph to a graph vector. The execution graphs could be presented as adjacency matrices. Since for large size programs the extracted adjacency matrices would be huge and sparse, converting the matrix to an appropriate vector may drastically reduce the size of problem space. Furthermore, the problem of finding similar graphs is converted into finding similar vectors which could be solved in polynomial time.

The proposed technique has four main phases: 1) building program execution graphs, 2) finding similar passing and failing graphs, 3) ranking edges of graphs, 4) constructing the best fault relevant path. Each main phase is described in detail in the following sub-sections.

3.1 Building Program Execution Graphs

This phase is done in three steps: program instrumentation, collecting execution data and building execution graphs. In order to instrument a program, any point in the program which determines an execution path, is considered as a predicate and a probe is inserted before the point. This enables us to know whether that specific predicate has been executed in the program in each particular run. The points we chose as predicates include explicit branch statements such as conditional and loop instructions. We also consider the calling point of a function and its return values as informative locations for collecting data. It is important to know, which function has been called, where it is called and what has it returned. Therefore, we have integrated both method calls and basic block transitions into a graph structure so called graph execution graph.

In the second step of this phase, the program is executed with different failing and passing test cases. In each execution, a sequence of predicates evaluated as True in that particular run is gathered and these sequences are used in the later step to form the adjacency matrix (i.e. correspond to the execution graph) for each specific execution. Each execution graph is labeled failing/passing according to the termination state of the program. To achieve this, for each two predicates \( P_i \) and \( P_j \) presented in the sequence, their corresponding edge is represented in execution graph and the weight of this edge is equal to the number of transitions from predicates \( P_i \) to \( P_j \) in the corresponding sequence. Table 1 shows some information about the size of the matrix constructed for Siemens suite and Space program. As shown in the table, the size of the matrix is a great concern in graph debugging techniques.

### Table 1: The size of execution graphs for Siemens suite and Space program

<table>
<thead>
<tr>
<th>Program</th>
<th>Number of predicates</th>
<th>The size of execution matrix</th>
<th>The size of execution vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcas</td>
<td>28</td>
<td>28×28</td>
<td>55</td>
</tr>
<tr>
<td>totinfo</td>
<td>61</td>
<td>61×61</td>
<td>96</td>
</tr>
<tr>
<td>schedule2</td>
<td>63</td>
<td>63×63</td>
<td>107</td>
</tr>
<tr>
<td>replace</td>
<td>83</td>
<td>83×83</td>
<td>200</td>
</tr>
<tr>
<td>schedule</td>
<td>97</td>
<td>97×97</td>
<td>142</td>
</tr>
<tr>
<td>printtokens</td>
<td>128</td>
<td>128×128</td>
<td>196</td>
</tr>
<tr>
<td>printtokens2</td>
<td>149</td>
<td>149×149</td>
<td>230</td>
</tr>
<tr>
<td>space</td>
<td>1198</td>
<td>1198×1198</td>
<td>1680</td>
</tr>
</tbody>
</table>

3.2 Finding Similar Graphs

After constructing the directed weighted graph for each specific run, it is required to find similar passing and failing graphs. For each failing graph, we compute at least \( k \) similar passing graphs and the aim is to contrast the edges in two classes of failing and passing graphs. It is apparent that if an individual edge is presented in a failing graph, but not in a close passing graph, it is more likely that the edge has relevancy to the fault(s). On the other hand, in cases that an edge has significant weight in a failing comparing with a passing graph, it is more probable that the transition is bug suspicious.

To this end, each graph is represented as a vector where each feature of the vector stands for a specific edge in the graph. First of all, all observed transitions in execution graphs are identified and then each feature of the vector is assigned to a transition which is observed at least in one execution graph. For each edge of a given graph, the corresponding feature in the vector contains the weight of the edge. At this point, the problem of finding similar graphs is converted to finding similar corresponding vectors considering the main aspects of execution graphs.

The similarity between two graphs is defined based on amount of common edges considering their corresponding weights. The Euclidian distance is a traditional measure for computing the difference value between vectors but it only considers unshared edges for estimating amount of distance. Thus, it is not applicable for our application. Another measures like Jacard-similarity and Cosine-similarity, consider both shared and unshared features in computing vector similarity. However, each of them alone might have drawbacks in finding similar graphs and therefore we have applied a combination of Cosine and Jacard measures to construct an adequate metric for our application. Jacard similarity measure, computes how much two given vectors contain shared elements. Assume that \( M_{ij} \) denotes the number of elements which have the value of ‘1’ in two vectors and \( M_{10} \) and \( M_{01} \) represents the number of elements with different values in two vectors, the Jacard measure for two vectors \( A \) and \( B \) is computed as below:

\[
\text{Jacard Index} (A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{M_{11}}{M_{01} + M_{10} + M_{11}}
\]

(1)

In our application, the Cosine similarity measure between each failing vector and all passing vectors is computed and then the passing vector with high amount of similarity is selected if their Jacard similarity is more than a predefined threshold. Jacard Similarity computes whether the number of common edges between failing and passing graphs is more than 50 percent of all edges. For each failing graph \( f \), we compute at least \( k \) nearest passing graphs and the aim is to contrast the edges including in these similar dynamic graphs to identify fault relevant edges.

The cosine similarities for vectors \( A \) and \( B \) is computed as follows:

\[
\text{cosine similarity} (A,B) = \cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}
\]

(2)

Algorithm 1 shows the details of finding similar graphs.

### Algorithm 1 Finding \( k \) similar passing run for each failing run

**Input:**
- Set of failing execution graphs in vector space: failVecs
- Set of passing execution graphs in vector space: passVecs

**Output:**
- Set of neighbor passing graphs : neighborset

1. for each failVec in failVecs
2. #close-graph = 0;
3. index=1;
4. for each passVec in passVecs
3.3 Ranking the Edges of Graph

As mentioned earlier, the concentration of the proposed approach in this paper is to rank the edges of the graphs based on their relevancy to the software fault. After this, in this paper, when we talk about ranking edges, we mean the edges involving in similar passing and failing dynamic graphs. The aim is to give an appropriate score for each edge in passing and passing graphs. At first, a preprocessing step is accomplished for which the edges with higher observation percentage in passing graphs in contrast with failing graphs, are considered as unsuspicious edges and are not considered in the ranking process.

After the preprocessing step, the ranking step starts from an individual failing execution graph. To be more precisely, when a failing graph, $G_{f}$, is contrasted with the nearest neighbor passing graph, $G_{p}$, all the edges in two graphs are compared one by one. There might be three different situations when contrasting $G_{f}$ with $G_{p}$:

1) If there is an edge in the failing graph, but not in passing one, the score of that edge is incremented by one.

2) If the passing graph contains an edge which does not exist in the corresponding failing graph, the score does not change.

3) If an edge exists in both failing and passing graphs with different weights, a null hypothesis testing is applied which is described below.

In some situations, the weight of an edge in failing execution graph is more than the corresponding edge in the passing graph. Since, we only consider a few number of passing execution graphs (i.e. the passing graphs near to failing ones), we cannot confidentially decide whether the higher weighted edge in failing graph is fault suspicious. Therefore, it is better to consider the weight distribution of a given edge in all passing and failing graphs. If we conclude that a given edge has similar weights in both failing and passing graphs, the rank of suspiciousness will not change. Otherwise, according to the difference, the score of the edge will be changed. To achieve this, we have applied a null hypothesis testing on the median of failing and passing samples.

In statistics [15], hypothesis testing is to verify an assertion about a distribution of a random variable or other statistical properties. The null hypothesis testing could be used when we want to verify whether a property in the statistical data is true or not. Therefore, we may have two symbols: $H_0$ for the null hypothesis that we want to test and $H_1$ for the alternative hypothesis. Based on the application, we have chosen null hypothesis test concerning means of the two populations: passing and failing graphs. Here, we want to test the overall behavior of a given edge in both passing and failing runs. Therefore, the null hypothesis is regarded against the alternative as shown below:

\[ H_0 : \mu_p = \mu_f \]
\[ H_1 : \mu_p \neq \mu_f \] (3)

Where $\mu_p$ and $\mu_f$ are the mean of the given edge weight in passing and failing runs, respectively.

In other words, we want to study, whether the behavior of an edge is similar in both populations of runs (i.e. passing and failing). If not, based on the amount of difference, which is measured with the $Z$ statistic, described later, the score of the edge will be increased. After some experiments on passing executions with almost 2000 runs in average, we found the population acting normal and since the failing executions are more than 30 with finite variance for many experimented programs, we applied central limit theorem to justify using the test of normal populations. In order to accept or reject the assertion, the $Z$ normalized static is used:

\[ Z = \frac{\bar{X} - \mu_p}{\sigma / \sqrt{n}} \] (4)

Where $\bar{X}$ is a random variable representing the weight of the edge and $\mu_p$ is the mean of $X$ in failing runs. The parameter $\sigma$ is the standard deviation of $X$'s in passing executions and $n$ is the number of failing graphs (i.e. runs). As shown in Figure 1 the computed $Z$-value is compared with the $Z_{\alpha/2}$: the boundaries of the critical regions. If it is in the range of $[-Z_{\alpha/2}, Z_{\alpha/2}]$, the $H_0$ assertion is satisfied and hence there is no considerable difference in behavior of an edge in passing and failing runs. Otherwise, the transition could be suspicious and the rank of edge will be changed as follows:

\[ \text{score} = \text{score} + \frac{W_{\text{fail}} - W_{\text{pass}}}{W_{\text{fail}}} \] (5)

In (5), $W_{\text{fail}}$ and $W_{\text{pass}}$ represent the weight of corresponding edge in failing and passing graphs respectively.

![Figure 1: The critical regions for accepting or rejecting the null hypothesis](image)

After analyzing all involving edges in three mentioned cases described earlier and calculating their scores, the overall score, final-score, for each edge is computed by (6).

\[ \text{final-score}_{(m,n)} = [\text{Freq}_f(m,n) - \text{Freq}_p(m,n)] \times 100 + \text{score} \] (6)

The formula in (6) computes the final-score for each including edge $(m, n)$. It first computes the percentage that specific edge, $(m, n)$, is observed in failing execution graphs (i.e. $\text{Freq}_f(m,n)$). A similar computation is performed for the same edge in passing
execution graphs (i.e., $Freq\ p(m,n)$). Next, a difference between two computed values is calculated which represents the suspiciousness of an edge for being fault relevant. According to the preprocessing step which has been done at first stage, the frequency of an edge in failing runs is more than its presence in passing executions and the result would be always positive. The resulting difference is added to the score, computed earlier based on three mentioned situations. The edges are then ranked based on their corresponding final-score. An edge with the highest score is the most fault suspicious edge and vice versa. The corresponding ranking method is shown in Algorithm 2.

### 3.4 Constructing the Fault Relevant Paths

As mentioned before, a programmer prefers to know about the context of a bug rather than a single location. In this phase of the proposed technique, the top $\Phi$ number of high ranked edges are identified and used for constructing fault relevant paths. At this point, we may have one or more paths with edges labeled by their corresponding scores. A whole score of a path is specified by computing the sum score value of all its edges, namely path-score.

Hence, if there exist more than one fault relevant path, the paths are ranked based on their corresponding path-score value. The algorithm is presented in Algorithm 3.

### Algorithm 2: Ranking graph edges

**Input:**
- Representation of transitions in vector space: vector
- Percentage of observing edges in failing graphs: $freq_{fail}$
- Percentage of observing edges in passing graphs: $freq_{pass}$
- Set of failing execution graphs in vector space: $failVec_{set}$
- Set of passing execution graphs in vector space: $passVec_{set}$

**Output:**
- Array of computed scores for suspicious edges: $Score$

1. for each feature in vector
2. $curr_{edge}$=edge corresponding to feature;
3. if ($freq_{fail}(curr_{edge}) < freq_{pass}(curr_{edge})$)
   4. continue;
5. $Z(curr_{edge}) = Compute\ Z\ statistic\ for\ curr_{edge}$;
6. for each $failVec\ in\ failVec_{set}$
7. for each $closeVec\ in\ neighborhood(failVec)$
8. if ($failVec(feature) != 0$ and $closeVec(feature) = 0$)
   9. $Score(curr_{edge}) += Z(curr_{edge})$;
10. else if ($closeVec(feature) != 0$ and $failVec(feature) > closeVec(feature)$)
   11. if ($Z(curr_{edge}) > \alpha/2$ or $Z(curr_{edge}) < -\alpha/2$)
      12. $Score(curr_{edge}) = Score(curr_{edge}) + [(closeVec(feature)-failVec(feature))/failVec(feature)]$;
13. end for
14. end for
15. $Score(curr_{edge}) += Score(curr_{edge}) + (freq_{fail}(curr_{edge})-freq_{pass}(curr_{edge})) * 100$;
16. end for
17. return $Score$;

### Algorithm 3: Constructing fault relevant paths

**Input:**
- Set of k suspicious edges with high Scores: $suspect$
- Structure of graph adjacency matrix: $ExeMat_{set}$

**Output:**
- Array of computed scores for fault relevant paths: $Score$

1. Initialize $ExeMat$ to 0;
2. for each edge in $suspect$
3. $ExeMat[start_{edge}, end_{edge}] = Score(edge)$;
4. building a directed weighted graph based on $ExeMat$;
5. $path_{set} \leftarrow all\ paths\ in\ graph$;
6. for each $path\ in\ path_{set}$
7. $Score(path) = \sum_{edge\ in\ path} Score(edge)$
8. return $Score$

### 4. EXPERIMENTAL RESULTS

In this section, the evaluation results of our approach on Siemens test suite and Space program are compared with some well known proposed techniques for software debugging. Siemens test suite contains seven standard programs. A wide variety of faults is seeded in this suite which makes it special for evaluating fault localization techniques [16].

In order to compute the percentage of code inspection, each reported faulty path with the highest score is treated as a sequence containing the scored edges (i.e., the scores given to the edges of the path according to their fault relevance). Therefore each transition is considered individually based on its score. For each transition $(p,q)$ we simply start from the predecessor predicate of $p$ in control flow graph and the statements located between predecessor predicate of $p$, so called $p'$, and the successor predicate of $q$, so called $q'$, are examined until we reach the cause of failure or all the statements in $(p',q')$ are examined. In the latter case, we go to the next high scored transition. With this mechanism, the number of examined statements is computed over whole program statements to find the percentage of examined code.

In Table 4, the number of localized bugs in terms of manually code inspection for different debugging methods is presented. As shown in the table, with 1% amount of code inspection, Context-Aware method has detected 38 faults in compared with our method which has detected 22 faults. However, with 20% amount of code scrutinizing our method with 95 located faults has outperformed Context-Aware method which has only found 73 faults. Furthermore, in overall the proposed approach has outperformed the mentioned methods with less than 30% code inspection.

Each reported faulty path is considered in computing the precision rate. For each existing fault the reported faulty path may or may not include the faulty statement. Therefore, in computing the precision rate we consider the number of paths which actually include the fault to all reported faulty paths as below:

$$precision = \frac{(reported\ paths\ containing\ fault)}{(reported\ paths\ containing\ fault) + (reported\ paths\ without\ fault)}$$ (7)

The recall rate is computed as (8)

$$Recall = \frac{the\ number\ of\ detected\ faults}{the\ number\ of\ considered\ faulty\ versions}$$ (8)

According to the precision and recall rates, the proposed approach is compared with RAPID [4] and Top-K LEAP [3] methods in Table 3. As shown in the table, in average the precision of RAPID and Top-K LEAP are 72.5 and 74.3, respectively. In the proposed approach, the precision has been increased to 79.68 which depicts the lower number of error reports in comparison with two other techniques. In total, the number of identified faults in terms of recall rate in RAPID and Top-K LEAP methods are 73.7 and 91, respectively; this value in the proposed approach is 92.42.

In order to show the scalability of our approach, the proposed method is applied to a standard program, called Space [16], containing 6218 lines of code. By applying our method on 24 faulty versions of Space program, 19 bugs are identified...
effectively by examining less than 1% of code and 21 bugs are located by 10% manually code inspection. In contrast, the Tarantula method can identify 12 out of 30 faulty considered versions of space program with 1% code inspection and 26 bugs are identified by examining 10% of code.

We also have compared our method with HOLMES on a limited test suite which is shown in Table 2. According to the reported results in [17], HOLMES by performing path profiling can identify only 25 out of 45 bugs with 10% manually code inspection, but our method has detected 40 out of 45 bugs with maximum 10% code examination.

Table 2: The test benchmark in HOLMES [17]

<table>
<thead>
<tr>
<th>program</th>
<th>versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>printhunks</td>
<td>V1-V10</td>
</tr>
<tr>
<td>printhunks2</td>
<td>V1-V10</td>
</tr>
<tr>
<td>schedule</td>
<td>V1-V10</td>
</tr>
<tr>
<td>space</td>
<td>V1-V10</td>
</tr>
<tr>
<td>totoinfo</td>
<td>V1-V10</td>
</tr>
<tr>
<td>replace</td>
<td>V1-V10</td>
</tr>
</tbody>
</table>

5. THREATS TO VALIDITY
In order to address some of the threats to validity, we may discuss the adequacy of Siemens test suite as a benchmark for the proposed technique. Siemens suite contains middle size programs which cannot justify our claim about the scalability of the technique. Although, the results achieved using the Siemens suite cannot be generalized to arbitrary programs, we believe that on larger programs with greater separation of concerns, the technique may work better. To support this belief we apply our technique on Space program which is a larger program in compare with Siemens programs. The results on Space reveal that with examining 24 versions, we have found 19 faults with less amount of 1% code inspection which outperforms the Tarantula method.

However, when we talk about scalability, we address debugging techniques using graph mining methods. With graph presentation the problem space may be enlarged (see Table 1) and we may face challenges such as sparsity. Therefore, for a program with edge, this number would be in our application. Nevertheless, we expect to see even better results on even larger programs that have an even greater separation of concerns. Although the Siemens suite is the standard suite of programs on which several researchers have evaluated their work, programs of larger size and number of faults may provide greater ability to generalize the results. We need more realistic programs with higher interactions among predicates for which the fault(s) occurs in specific transitions among predicates rather than the predicate itself.

The metric we have used to measure the similarity between graphs could also be another threat to the validity. Since this metric does not necessarily consider the structure of the given graphs it might face difficulties for some type of execution graphs.

Another limitation to the experiment is that we did not implement the entire graph debugging techniques in order to report the differences in timing results. However, we believe that due to the nature of the technique we could obtain adequate timing results compared with similar techniques.

6. CONCLUSIONS AND FUTURE WORK
In this paper, a new fault localization approach for analyzing program execution graphs extracted from multiple executions has been introduced. An execution graph is capable to summarize an overall dynamic behavior of a single program run into an adequate and informative structure.

A novel ranking method introduced in the paper, assigns scores to all suspicious edges and sorts them based on their fault relevancies. In the next step, based on the structure of program and existing test cases, one or more faulty paths are formed and reported to the programmer. By generating fault relevant paths, the context of bug is provided for programmer which helps her/him understand the context of bug more easily. The experiments on Siemens test suite and Space program manifest the capability of the proposed approach in contrast with some outstanding methods like Context-Aware [10], Tarantula [9], SOBER [18], NN [8], Liblit05 [19], CT [20] and Argus [21] according to the percentage of uncovered faults in terms of manual code inspection. Also in contrast with HOLMES, the proposed approach can identify more bugs. Furthermore, it is observed that our method performs better than RAPID [4] and Top-K LEAP [3] based on the computed precision and recall rates.

We believe that a reason for effectiveness of the approach is considering the weight of transitions in forming program execution graphs. Another privilege of the proposed technique is its scalability. Since the edges are analyzed separately, the size of program might not harm the effectiveness of the ranking algorithm; whereas in entropy based graph mining techniques, the size of graphs has considerable effect in decreasing the efficiency.

For future work we may consider the structure of execution graphs in converting graph matrices into vectors. The graph kernels might be helpful in this context. An important goal in this area is finding multiple bugs in program which will be studied in future works. We may also employ sampling techniques, to reduce the overhead of our heavy weight instrumentation framework. Evaluating the approach on larger and more realistic programs with high interactions among predicates will also be considered in future. We may also seek to improve our similarity metric for measuring the similarity between execution graphs.

7. ACKNOWLEDGMENTS
Our thanks to Xiaoyin Wang for his valuable comments in comparing our work with [3]. We would also like to appreciate reviewers for their insightful questions, comments and suggestions which helped us to improve the paper.
Table 4: The comparison on the number of localized bugs between our method and other well-known methods in terms of manual code inspection

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22</td>
<td>38</td>
<td>17</td>
<td>11</td>
<td>10</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>82</td>
<td>67</td>
<td>68</td>
<td>68</td>
<td>52</td>
<td>34</td>
<td>74</td>
<td>18</td>
</tr>
<tr>
<td>20</td>
<td>95</td>
<td>73</td>
<td>75</td>
<td>96</td>
<td>85</td>
<td>49</td>
<td>98</td>
<td>28</td>
</tr>
<tr>
<td>30</td>
<td>101</td>
<td>-</td>
<td>87</td>
<td>102</td>
<td>91</td>
<td>67</td>
<td>107</td>
<td>41</td>
</tr>
</tbody>
</table>

8. REFERENCES


