Graph Mining for Software Fault Localization: An Edge Ranking based Approach

Marwa Gaber Abd El-Wahab[1], Amal Elsayed Aboutabl, Wessam M.H. EL Behaidy

***Abstract*— Fault localization is considered one of the most challenging activities in the software debugging process. It is vital to guarantee software reliability. Hence, there has been a great demand for automated methods that can pinpoint faults for software developers. Various fault localization techniques that are based on graph mining have been proposed in the literature. These techniques rely on detecting discriminative sub-graphs between failing and passing traces. However, these approaches may not be applicable when the fault does not appear in a discriminative pattern. On the other hand, many approaches focus on selecting potentially faulty program components (statements or predicates) and then ranking these components according to their degree of suspiciousness. One of the difficulties encountered by such approaches is to understand the context of fault occurrence. To address these issues, this paper introduces an approach that helps in analyzing the context of execution traces based on control flow graphs. The proposed approach uses the edge-ranking of basic blocks in software programs using Dstar that proved to be more effective than many fault localization techniques. The proposed method helps in detecting some types of faults that could not be previously detected by many other approaches. Using Siemens benchmark, experiments show the effectiveness of the proposed technique compared to some well-known approaches such as Dstar, Tarantula, SOBER, Cause Transition and Liblit05. The percentage of localized faulty versions versus the percentage of code examined is taken as a measure. For instance, when the percentage of examined code is 30%, the proposed technique can localize nearly 81% of the faulty versions, which outperforms the other four techniques.**

***Index Terms*—Bug localization, basic block, control flow graph, edge – ranking.**

I. Introduction

Manual software debugging is not only a time consuming, tedious and costly task but also error-prone [1]-[2]. This process is crucial yet resource-intensive in software engineering [3], with 50% to 80% of software maintenance costs attributed to fix [4]. An intuitive way to localize faults[2] in software programs is to analyze the memory dump of the faulty program. Another way is to insert a "print" statement around a suspicious region. However, all these manual solutions proved to be inefficient in locating bugs. Various approaches have been introduced in literature to automate the process of locating faults in efficient ways. Slice-based methods have been introduced to reduce the domain of debugging search via slicing. If a test case fails due to an incorrect value of a variable at a statement, then as fault should be found in the slice associated with that variable-statement pair [6]. Static slicing and all extended approaches based on slicing [7]-[8] do not make use of the input values that discover the fault. Moreover, dynamic slicing methods, which determine program slices to further reduce the debugging search domain for possible locations of a fault, may consume excessive time and space. Renieris *et al* [10] have proposed an approach based on nearest neighbour queries where the distance between program execution abstractions is determined. This approach assumes that there is one faulty run and a number of successful runs. The successful run that is most similar to the faulty run is determined based on a distance criterion, and then the difference between both runs is used to locate the fault.

Amongst the most effective diagnostic techniques is Spectra Based Fault Localization (SBFL) [10]-[13], also known as code coverage techniques. SBFL, which has recently shown much popularity to be a very efficient and simple technique in software debugging, focuses on identifying, then assigning a suspicious value for each software component (statement, predicate, function, etc.) and finally ranking them according to how likely they are regarded as fault-relevant [15].

However, SBFL suffers from some drawbacks that cause incorrect results. One of these problems is that SBFL cannot provide enough information to understand the context of faults, because of assessing the suspiciousness of statements or predicates individually [16]. Furthermore, in some approaches as in [17]-[21] predicates are considered in isolation from each other, which may lead to reducing the ability to detect certain faults that are generated from a specific transition from one

Manuscript received September 11, 2017; revised October 28, 2017. Date of publication December 21, 2017.

Authors are with the Faculty of Computers and Information, Helwan University, Egypt (e-mails: [1]marwagaber.24@yahoo.com, {amal.aboutabl, w_behaidy}@fci.helwan.edu.eg).

Digital Object Identifier (DOI): 10.24138/jcomss.v13i4.402

[1]Corresponding author.

[2]The words fault and bug are used interchangeably in this paper.

point (predicate) to another[22]. Many approaches use graph mining for software bug localization. These approaches rely mostly on extracting discriminative sub-graphs as suspicious areas in programs [21] -[23]. In fact, these discriminative sub-graphs do not provide information, which effectively helps in finding the difference between correct and incorrect executions paths.

In this paper, a context-aware bug localization technique that uses control flow graphs is proposed. The execution behavior for failing and passing paths is represented by control flow graphs that are used later in analyzing the executed test cases. The proposed method provides the context of bug to facilitate identifying, understanding and fixing the bug.

The main contributions of this paper include three aspects:

1. Using a lightweight approach that is fully automatic and broadly applicable based on execution runs.
2. Combining control flow edge coverage with block coverage and calculating their suspiciousness to rank the basic blocks in descending order of their suspiciousness value.
3. The proposed method produces promising results especially with some type of faults that cannot be localized using some SBFL methods such as missing statements.

This paper is organized as follows: Section 2 presents related work. Section 3 gives a detailed description of the proposed technique, followed by an illustrative example and experimental evaluation. Finally, Section 4 concludes this paper and presents the future work.

II. Related Work

Many approaches have been proposed for automating fault localization and improving the rate of identifying faults[24]. Graph mining based approaches as in [22], [24]-[25] use the graph structure to demonstrate the execution behavior of software programs. In these approaches, graph nodes represent code units such as predicates, functions or basic blocks and graph edges represent the relations among these code units. Recently, many studies have utilized graph-mining techniques in software fault localization. The behavior of a software program can be represented as a call graph or a subset of a control flow graph. These techniques may mine the dynamic execution graphs, which are labelled as correct or incorrect according to the termination state of each execution. The termination state is determined as correct or incorrect based on whether the expected results are met.

Di Fatta *et al.* [25]and Liu *et al. [*26] propose approaches that rely on applying closed frequent pattern mining and using these patterns as features for training a classifier. However, in large-scale software programs, the number of frequent sub-graphs becomes large therefore increasing mining complexity. Moreover, these approaches do not consider the weight of each transition in their analysis. Due to the expensive computation of applying closed frequent pattern mining to fault localization, discriminative pattern mining approaches have been proposed by many studies [27]-[29].

In [21], the proposed approach is based on LEAP algorithm [28] to extract program behavior graphs in two levels of granularity: basic blocks and function calls. The extracted discriminative sub-graphs can separate incorrect executions from correct ones, then, the informative signature of faults is determined. Top-K LEAP is an entropy-based algorithm, which identifies the top k ranked discriminate sub-graphs. For some non-deterministic faults, for which the corresponding signatures are not highly discriminative, discriminative pattern mining methods might be inefficient. These approaches may also have problems in scalability.

Some approaches apply data mining algorithms on weighted call graphs[30]. According to these approaches, edge weight plays an important role in finding faulty method calls in faulty executions. However, the main problem is large granularity since using only method call graphs is insufficient to find all types of faults. Variants similarity coefficients are used in ranking program components. Tarantula[11]is one of the early techniques introduced for SBFL using statement-hit spectra. It is based on using the coverage statistics to assign a suspiciousness value to each statement in the program. Tarantula is based on the idea that statements, which are executed by faulty executions are more likely to be suspicious than those which are executed in successful executions.

In [31]-[32], Ochiai and Jaccard similarity coefficients have also been used for fault localization. They are based on the same heuristic as Tarantula except that they use a different formula to compute suspiciousness. According to Abreu *et al.* [32], Ochiai formula is more effective in many experimental studies than Tarantula. Naish *et al.* [14]have introduced another formula called Op2, which proved to be effective in programs that have a single fault. In [17], the proposed statistical-based technique is based on separating effects of different faults and identifying predictors that are associated with individual faults. These predictors uncover the circumstances under which faults happen and reveal the frequencies of modes of failure, which consequently facilitates prioritizing debugging efforts to detect bugs.

Liu *et al.[*1] have proposed an approach called SOBER which compares evaluation patterns of predicates of both failed and successful executions to isolate fault-relevant predicates. Causes transition [33] statistical-based approach, which is abbreviated as CT, is an extension of the previous work of Zeller *et al.* [34], which is based on using the Delta Debugging algorithm to narrow down the state difference between failing and successful executions according to their memory graphs. CT extends this idea by adding the capability of searching in time to searching in space. Searching in time seeks moments when faulty variables start to cause failures in the program.

Despite the effectiveness of SBFL and statistical-based techniques, they ignore dependencies in a program where each component in a program is considered isolated from each other and regarded as independent, which may reduce their ability in detecting faults. However, the coverage run-time of program components is calculated individually. Many fault localization studies that use statement or predicate hit spectra neglect the dependency relationship between program entities, which may locate irrelative entities [35].

In this paper, the proposed approach preserves the consistency of program entities in coverage and it considers the context of faults via path analysis.

III. Preliminaries and The Proposed Fault-Localization Technique

A. Preliminaries

Definition 1. A software program ∏ is formed by a sequence of N statements.

Definition 2. A test suite T = {t_1, t_2… t_m} is a collection of test cases that are intended to test whether the program works as expected or not, where m=|T| is the number of test cases.

Definition 3. A test case is a (I, O) tuple, where I is a collection of input variables for determining whether the program being tested works as expected. O is the expected result. If ∏ (I) =O, the test case is said to be successful and faulty otherwise.

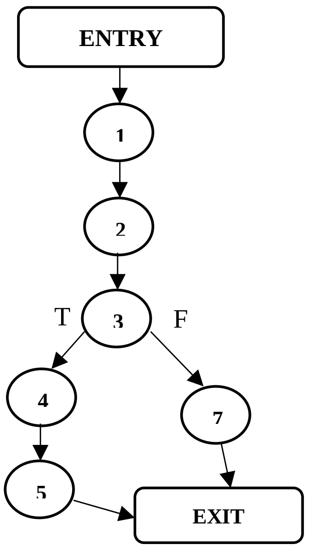
Definition 4. A basic block B in the software program is a sequence of statements or expressions that do not include the transfer of control such that if one of these statements is executed, all other statements are also executed.

Definition 5. Control flow graph CFG= {N, E, P} is used to represent execution paths where N= {N_1, N_2, ⋯, N_m} is the set of nodes which represent basic blocks in the program, E= {E_1, E_2, ⋯, E_n} is the set of graph edges representing transitions from one basic block to another and P= {P_1, P_2, ⋯, P_k} is the set of execution paths

Fig. 1 illustrates the control flow graph for a procedure called max and its control flow graph (CFG).

```
  procedure Max () {
1 int a =read_int();
2 int b =read_int();
3 if (a > b ) {
4 print (a);
5 return ;
6 }
7 print(b);
8 }
```

(a) Procedure Max.



(b) CFG of Max.

Fig. 1.Function max with its control flow

B. The proposed fault localization technique

Given a faulty program to be debugged and a set of test cases for this program, the proposed method performs software-debugging going through four main phases:

1. Constructing control flow graphs (CGF) for all test cases.
2. Ranking graph nodes.
3. Ranking graph edges.
4. Constructing node suspiciousness list.

Step 1: Constructing control flow graphs (CGF) for all test cases.

Before building a CFG, the program being tested is instrumented by inserting a "print" statement in each basic block of the program. In this phase, the block-hit spectrum is used to collect execution traces of a program for each test case. The main idea behind tracing basic blocks is to cover different types of faults such as "missing statements" and perform context-aware fault detection.

Each program is executed with different failing and passing runs. Each execution path is labelled as failed or passed according to the termination state of the program. Then, the CFG is constructed based on the sequence of blocks covered by the execution path. If *i* and *j* are two consecutive basic blocks in an execution path where *i* appears first, then these two blocks are represented by nodes in the CFG and there is a directed edge from *i* to *j.*

Step 2: Ranking graph nodes

In this phase, the suspiciousness score of each node (basic block) is calculated. A suspiciousness metric is a binary similarity metric between the block coverage vector and the result vector (Fig. 2). Various suspiciousness metrics exist as Jaccard metric, Tarantula metric, Ochiai metric, and D* (D-star) metric [36]. In the proposed method, D* (D-star) is used with * = 2 to compute the suspiciousness score of each node [37]. The D^2 is defined as follows:

$$D^2 = \frac{{N_{cf}}^2}{N_{uf}+N_{cs}} \tag{1}$$

where N_{cf} is the number of failed test cases that cover a program entity, N_{uf} is the number of failed test cases that do not cover a program entity and N_{cs} is the number of passed test cases that cover a program entity.

The D* metric computes the suspiciousness score at the level of program statements which prove its effectiveness over 38 other metrics used for evaluation. However, the main problem with using a statement as the program entity is that statements are considered in isolation from each other and regarded as independent units that may reduce the ability to detect all types of faults. In the proposed method, the use of D* is adapted such that the program entity of concern here is the basic block not the statement. The basic block coverage matrix represents the coverage information for each test case where (X) at basic block B_i and test case t_j indicates that B_i is covered by t_j. Each entry in the result vector indicates if that test case is faulty or successful.

After calculating the suspicious score for all nodes of the graph that represent basic blocks of the faulty program, all nodes are sorted in a descending order to form the ranking list L_N of suspicious basic blocks.

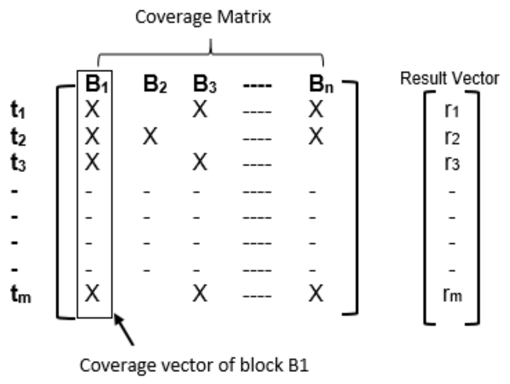


Fig. 2. Coverage information for each test and basic blocks

Step 3: Ranking graph edges

Because of dependencies, a suspicious node may affect all execution paths that cover this node and triggers the fault. Therefore, ranking nodes suspiciousness scores individually might report incorrect results. To incorporate dependencies between nodes and pinpoint the block that is the root cause of a fault, suspiciousness score is calculated for each edge in the graph based on the same similarity metric, D*, adopted for ranking nodes. The definition of the D*, as (1) where * is adapted to calculate a suspiciousness score for each edge *e*.

In analogy to the block coverage matrix, an edge coverage matrix and a result vector are constructed (Fig. 3).

Each column in the edge column matrix represents an edge coverage vector that is used to compute the edge suspiciousness score. An (X) for edge E_i and test case t_j indicates that E_i is covered by t_j.

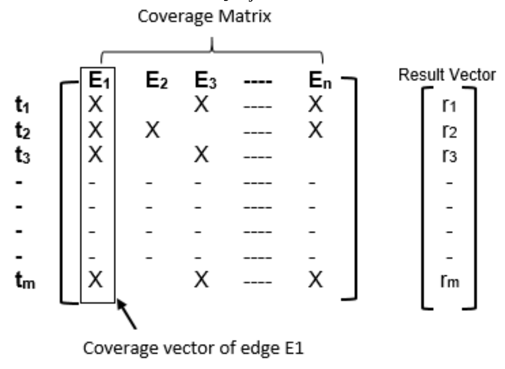


Fig. 3. Edge Coverage Matrix and Result Vector

Step 4: Constructing node suspiciousness list

Many studies provide a list of ranked basic blocks only after calculating their suspiciousness value using one of the similarity coefficients. In this work, edge ranking is incorporated into the suspiciousness value of each node. After ranking edges that appear in the CFG of a faulty execution, the suspiciousness score of each edge is assigned to both of its incident nodes. For example, if an edge *e* is incident to nodes x and y, the suspiciousness value of *e* is assigned to both x and y. If node x has a set of edges E_x to which it is incident, the maximum edge suspiciousness value of all edges in E_x is assigned to x. Then, all nodes are sorted in a descending order of their suspiciousness values to form the ranking list L_E of suspicious basic blocks according to the suspiciousness value of their edges.

If more than one node has the same suspiciousness value, these nodes are ranked according to their suspiciousness value in the L_N list formed in step 2. Finally, the node that has the highest suspiciousness value should be inspected first. If it does not contain the fault, the node with the next highest suspiciousness value is inspected and so on until the fault is found.

IV. ILLUSTRATIVE EXAMPLE

The previous steps are illustrated through one of the Siemens benchmark programs, namely, *schedule v4*. Siemens benchmark [38] programs are used in this work to test the effectiveness of the proposed technique as will be explained in later sections. Fig. 4 shows a code fragment of *schedule v4*, which is a priority scheduler. This fragment contains only one fault at line 23.

As step 1, The CFG of the chosen code fragment is generated. It contains 33 statements and 9 basic blocks. Nodes represent basic blocks of the faulty program and each edge in the CFG represents a control flow transition between one basic block to another.

As shown in Table 1, nodes and edges that appear in CFG in Fig. 4

TABLE 1
NODES AND EDGES OF CFG OF FIG. 4

Nodes	Edges
B1	B1 B3
B2	B1 B2
B3	B3 B4
B4	B3 B9
B5	B4 B5
B6	B6 B7
B7	B7 B5
B8	B8 B1
B9	

In step 2, Table 2 shows the ranking of nodes in the fragment code using the D^2 similarity metric as in (1). According to Table 2, B7 is the first block to be checked for the fault, then B3 and so on. The fault will be detected in the second basic block B3. Using suspicious basic blocks ranking individually may not report accurate results. Because of dependencies, a suspicious node may affect all execution paths that cover this node.

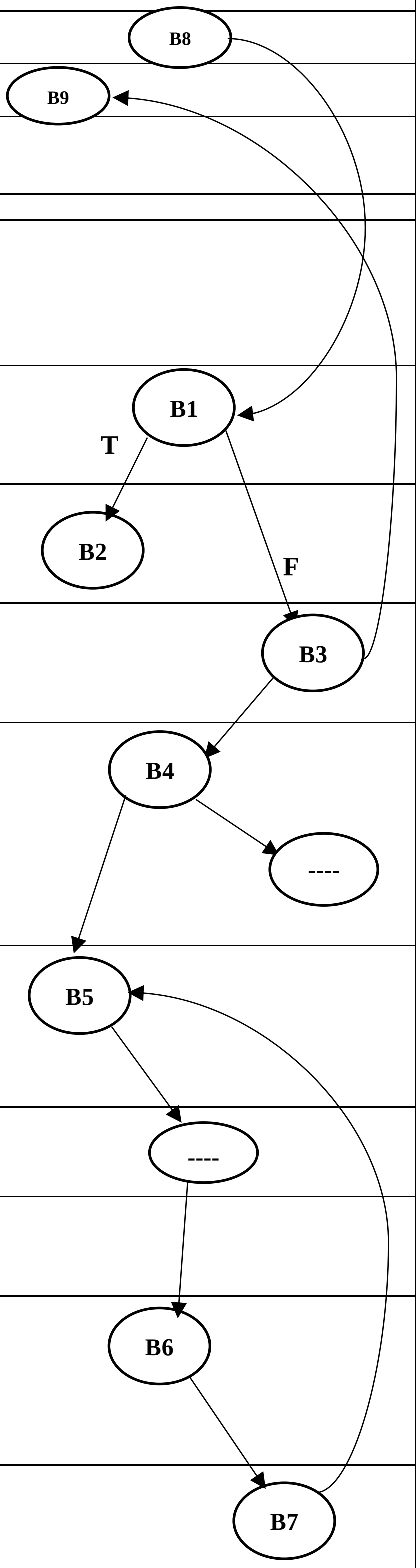
Lines	Stat.	Control Flow Graph	Code
1	1	 B8 B9 B1 T B2 F B3 B4 ---- B5 ---- B6 B7	`Else`
2	-		`{//B8`
3	2		`upgrade_process_prio(prio, ratio);`
4	-		`}//B9`
5	3		`break;`
6	4		`case NEW_JOB:`
			`------` `------`
7	-		`}`
8	5		`void upgrade_process_prio(prio, ratio)`
9	6		`int prio;`
10	7		`float ratio;`
11	8		`int count;`
12	-		`{//B1:` **`the following statements are in one block called B1`**
13	9		`int n;`
14	10		`Ele *proc;`
15	11		`List *src_queue, *dest_queue;`
16	12		`if (prio >= MAXPRIO)`
17	-		`{//B2`
18	13		`return;`
19	-		`}//B3`
20	14		`src_queue = prio_queue[prio];`
21	15		`dest_queue = prio_queue[prio+1];`
22	16		`count = src_queue->mem_count;`
23	17		`if (count > 1)//bug, it should be  if (count>0)`
24	-		`{//B4`
25	18		`n = (int) (count*ratio + 1);`
26	19		`proc = find_nth(src_queue, n);`
			`-----` `-----` `-----`
27	20		`Ele *find_nth(f_list, n) List *f_list; int n;`
28	-		`{//B5`
29	21		`Ele *f_ele;`
30	22		`int i;`
31	23		`if (!f_list) {`
			`------` `------` `------`
32	24		`void unblock_process(ratio)`
33	25		`float ratio;`
34	-		`{//B6`
35	26		`int count;`
36	27		`int n;`
37	28		`Ele *proc;`
38	29		`int prio;`
39	30		`if (block_queue)`
40	-		`{//B7`
41	31		`count = block_queue->mem_count;`
42	32		`n = (int) (count*ratio + 1);`
43	33		`proc = find_nth(block_queue, n);`

Fig. 4. Fragment of schedule v4 program (Siemens Test Suite)

TABLE 2
FRAGMENT CODE BASIC BLOCKS RANKING USING D^2 BASED ON BLOCKS SUSPICIOUSNESS VALUES ONLY.

Rank	Basic blocks
58.615	B7
49.629	B6
58.363	B3
56.828	B9
56.828	B8
56.828	B1
56.081	B5
44.645	B4
0.426	B2

In step 3, the suspiciousness score is computed for each edge in the CFGs of all execution traces using D^2. Edges are, then, ranked in descending order as in Table 3.

As shown in Table 3, (B3, B9) edge is ranked first which means that this transition has the greatest suspiciousness score and it should be inspected first, (B6, B7) is second and so on.

In step 4, nodes are ranked according to the suspiciousness score of their edges. For example, the suspiciousness score of edge (B3, B9) is 64.8918 so the suspiciousness score of
B3 = 64:8918 and B9 = 64:8918.

TABLE 3
FRAGMENT CODE EDGE RANKING USING D^2.

Rank	Edges
64.891	B3 B9
58.615	B6 B7
58.615	B7 B5
58.363	B1 B3
56.828	B8 B1
44.645	B3 B4
44.645	B4 B5
0.4269	B1 B2

In case node B1 has a set of edges E_x = {(B1, B2), (B1; B3)} to which it is incident. The suspiciousness score of (B1, B2) and (B1, B3) is 0.4269 and 58.363 respectively. Therefore, node B1 is assigned the maximum edge suspiciousness value in E_x which is 58.363. With the same procedure, node B3 is assigned the suspiciousness value 64.891.

In Table 4, a fragment of a ranking list of nodes according to suspicious values of their edges. This list is noted L_E.

TABLE 4
FRAGMENT OF RANKING VALUES OF NODES ACCORDING TO EDGES RANKING

Rank	Basic Blocks
64.891	B3
64.891	B9
58.615	B6
58.615	B7
58.615	B5
58.363	B1
56.828	B8
44.645	B4
0.4269 B2	B2

In case, there are more than one node has the same suspiciousness value such as B3 and B9, step 2 is used to rank these nodes according to their position in L_N. The final ranking (See Table 5) orders basic blocks in the order they should be inspected by developers to find the fault. After incorporating edge ranking, the faulty basic block is the first on the list instead of the second.

TABLE 5
FINAL BASIC BLOCKS RANKING LIST AFTER INCORPORATING EDGE RANKING

Rank	Basic blocks
1	B3
2	B9
3	B7
4	B5
5	B6
6	B1
7	B8
8	B4
9	B2

V. Experimental Study

In this paper, the Siemens test suite [38] is used as a benchmark to compare the proposed method to other well-known approaches for fault localization. The experiments are conducted on Ubuntu-15.04 platform using GCC 4.9.2 compiler.

To evaluate the proposed technique, the *EXAM* score [39] is used as a measure. The *EXAM* score indicates the percentage of program elements (number of examined basic blocks) that needs to be examined before the first fault is reached. The lower the *EXAM* score, the better the performance. *EXAM* score is computed as follows:

$$EXAM \text{ score} = \frac{\text{Rank of the first faulty program element}}{\text{Total number of executable program elements}} \times 100\% \quad (2)$$

The *EXAM* score is used to evaluate the result based on the generated ranking lists in case using D* as * = 2 with basic blocks and incorporating edge ranking in basic blocks ranking. Furthermore, it is used for comparing the proposed technique to some well-known approaches that use SBFL.

A. Siemens Test Suite

This suite was originally prepared by Siemens Corporation Research with the aim of studying test adequacy criteria [40]. Many software fault localization studies as in [1], [9], [37], [41]-[44] used the Siemens test suite to evaluate their performance. The suite contains seven programs with different types of injected faults. Every version contains only one fault.

The seven programs in this suite perform a variety of tasks: *print-tokens* and *print-tokens2* are lexical analyzers, *schedule1* and *schedule2* are priority schedulers, *replace* performs pattern matching and substitution, *tot-info* computes statistics given input data and *tcas* is an aircraft collision avoidance system. Some versions have been excluded from the experiments in the same manner as several previous studies did as in [12], [31]-[35] due to the absence of faulty test cases in some versions or the absence of syntactic differences from correct versions of the program. Table 6 provides some statistics of the programs and test cases in the Siemens test suite.

The vertical axis represents the percentage of identified faults that are located by examining an amount of code less than or equal to the corresponding value on the horizontal axis.

In Fig. 5(a), it can be noted that by examining 5% of the code, the proposed technique can locate all faults in the print-tokens program while using D* on basic blocks, only 80% of the faults can be located. Performing edge ranking considerably improves the results. For all seven programs, edge ranking provides more information than pure basic block ranking, thus more faults can be located by examining less code.

TABLE 6
BRIEF DESCRIPTION OF SIEMENS TEST SUITE

Siemens Programs	No.of faulty versions	Executabl e LOC	No.of test cases	Description
Print_toke ns	5	341-342	4130	Lexical analyzer
Print_toke n2	9	350–354	4115	Lexical analyzer
Replace	27	508–515	5542	Pattern Replacement
Schedule	5	291-294	2650	Priority scheduler
Schedule2	9	261-263	2710	Priority scheduler
Tcas	37	133-137	1608	Altitude separation
Tot_info	20	272-274	1052	Information measure

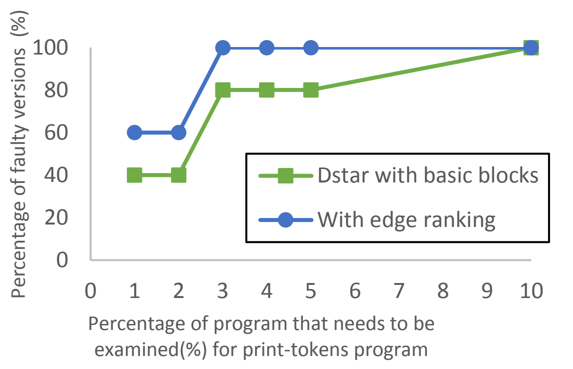
B. Comparison with well-known techniques

In [24], empirical studies have shown that Dstar [37] is more effective than Ochiai [32], which is in turn more effective than O and OP [14], RBF [44], Crosstab-based [39], H3b and H3c [41]. Hence, according to the previous section, the proposed approach is suggested to be more effective than those techniques. Therefore, it will be more convenient to compare the proposed approach with some well-known techniques that use different similarity coefficients to locate bugs.

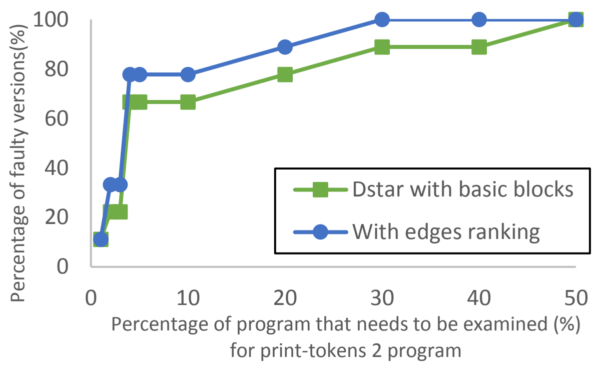
The proposed technique is compared to Tarantula [12], SOBER [1], Cause Transition [33] and Liblit05[17].These well-known methods are often chosen for comparison in previous studies. To evaluate the effectiveness of a fault localization approach, the number of detected faults and the amount of code that needs to be examined manually to find the root cause of the error has to be considered.

Fig. 6 shows the percentage of code that should be examined manually in the source code to find the main cause of failure compared with some well-known fault localization techniques. For example, by examining 30% of program code, 82.1% of faults (faulty versions) can be detected.

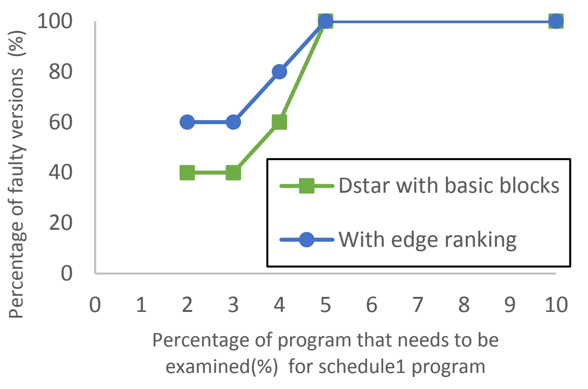
Although the proposed technique does not perform best in all cases, it is very promising in bug localization. When the percentage of examined code is approximately above 30%, the proposed technique can localize more faulty versions than the other techniques.



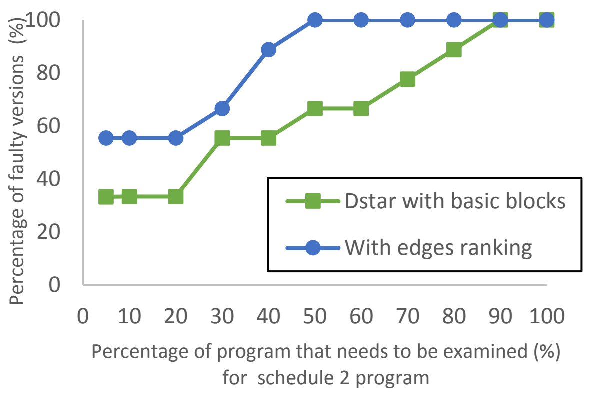
(a)



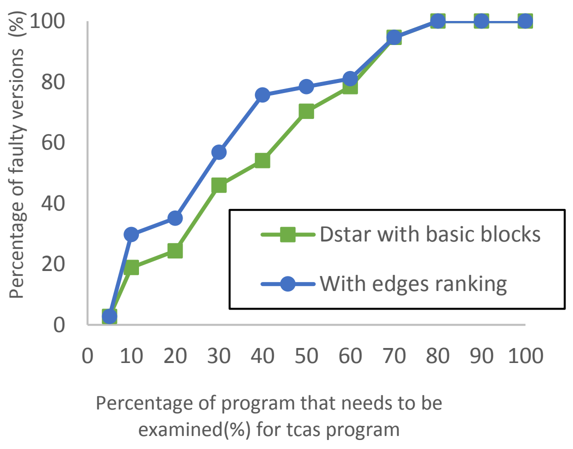
(b)



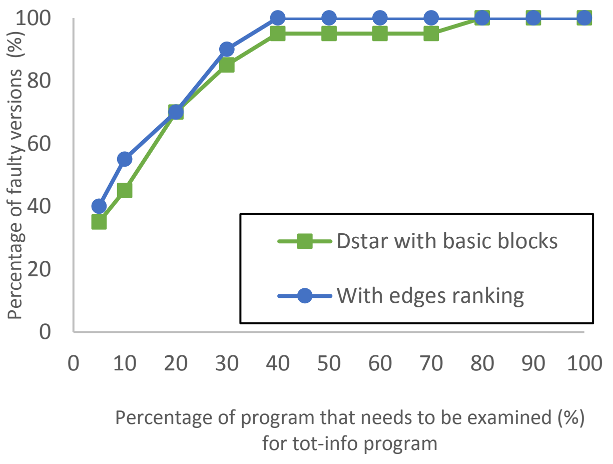
(c)



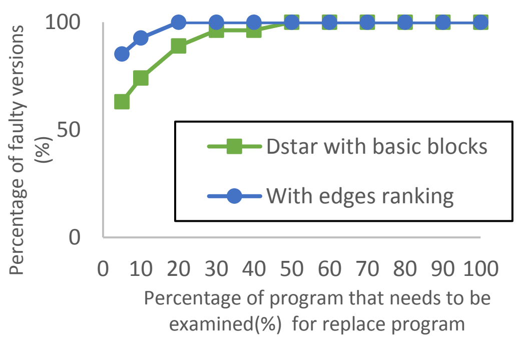
(d)



(e)



(f)



(g)

Fig. 5. Evaluation of the proposed approach results using Dstar with basic blocks and edge ranking for the seven Siemens programs

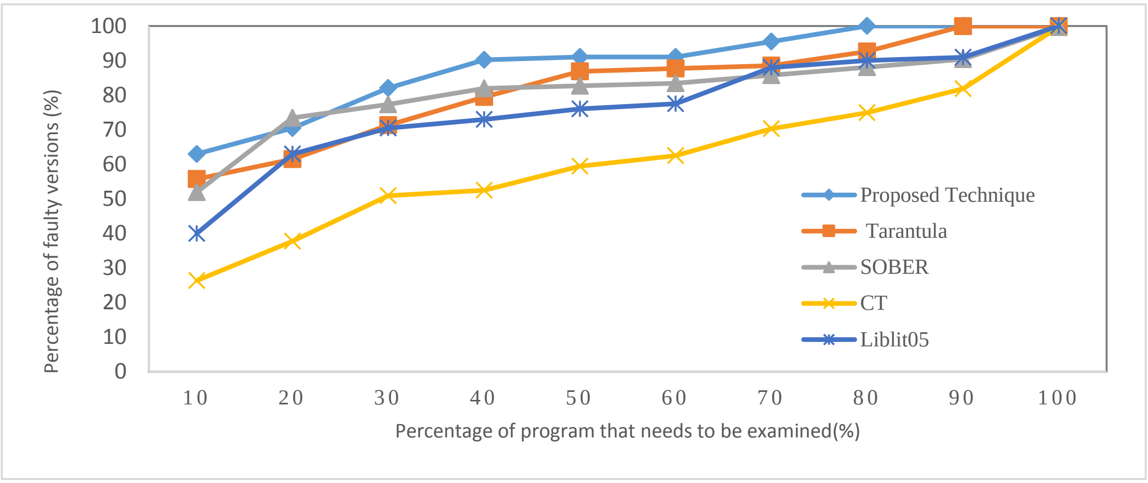


Fig. 5. The effectiveness of the proposed technique

VI. Conclusion and Future Work

An edge-ranking approach is introduced in this paper to localize faults in faulty software programs. The proposed approach provides a context-aware understanding for located faults using control flow graphs. It combines control flow graphs with block coverage to calculate the suspicious score for each basic block for successful and failed execution paths. Then, all suspicious basic blocks are ranked in descending order based on their suspiciousness. The final ranked list should be inspected by developers to check the suspicious blocks.

By following the ranked list of suspicious basic blocks that are constructed using edges ranking, the number of inspected basic blocks to find the fault (i.e., the search space), is reduced.

Experiments are conducted to compare the effectiveness of the proposed technique with existing representative techniques using Siemens benchmark. Results of these experiments are promising. The percentage of localized faulty versions is measured against the percentage of code examined. In most cases, the proposed technique outperforms Tarantula, SOBER, CT and Liblit05. For instance, when the percentage of examined code is 30%, the proposed technique can localize nearly 81% of the faulty versions, outperforming the other four techniques.

Although the proposed edge-ranking based fault localization approach helps in locating many types of bugs and can provide a context-aware understanding for these bugs, there are many aspects that should be considered as future work. The proposed edge-ranking based approach may be enhanced by detecting multiple faults instead of pinpointing just the first bug in the program. In addition, enhancing the proposed method by detecting faults in huge software and not only locating bugs in medium and small programs. Finally, we suggest attempting to enhance the proposed method by combining some other types of graphs (such as data-flow and program dependence graphs) with spectrum-fault localization approaches.

References

[1] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, “Statistical debugging: A hypothesis testing-based approach, ”IEEE Trans. Software Eng., vol.32, no. 10, pp. 831–848, Oct. 2006.

[2] Goel, A. L., “Software Reliability Models: Assumptions, Limitations, and Applicability”. IEEE Transactions on Software Engineering, vol. SE-11, no. 12, pp.1411-1423 1985, DOI. 10.1109/TSE.1985.232177

[3] Xie, M., Yang, B.,” A study of the effect of imperfect debugging on software development cost”. IEEE Transactions on Software Engineering, vol. 29, no. 5, pp. 471-473, 2003, DOI: 10.1109/TSE.2003.1199075

[4] Agrawal, H., Horgan, J., London, S., Wong, W.,” Fault localization using execution slices and Data flow tests. Proceedings of Sixth International Symposium on Software Reliability Engineering”, vol.ISSRE’95, pp143-151,1995,DOI:10.1109/ISSRE.1995.497652

[5] Collofello, J. S., Woodfield, S. N.,” Evaluating the effectiveness of reliability-assurance techniques”. The Journal of Systems and Software, vol. 9, no.3, pp.191-195, 1989, DOI: 10.1016/0164-1212(89)90039-3.

[6] Weiser, M., “Programmers use slices when debugging”. Communications of the ACM, vol.25, no.7, pp.446-452, 1982 DOI:10.1145/358557.358577.

[7] Agrawal, H., Demillo, R. A., Spafford, E. H., “Debugging with dynamic slicing and backtracking”. Software: Practice and Experience, vol. 23, no.6, pp.589-616, 1993, DOI: 10.1002/spe.4380230603.

[8] Zhang, X., He, H., Gupta, N., Gupta, R., “Experimental evaluation of using dynamic slices for fault location". International Workshop on Automated and Algorithmic Debugging (AADEBUG 2005), pp.33-42, 2005, DOI: 10.1145/1085130.1085135

[9] Sterling, C. D., Olsson, R. A.,”Automated bug isolation via program chipping. Software” – Practice and Experience, vol.37, no.10, pp. 1061-1086, 2007, DOI: 10.1002/spe.798

[10] Renieres, M., Reiss, S. P., “Fault Localization with Nearest Neighbour Queries”. Automated Software Engineering, pp.30-39, 2003, DOI:10.1109/ASE.2003.1240292

[11] Xie, X., Chen, T. Y., Kuo, F.-C., Xu, B., “A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization”. ACM Transactions on Software Engineering and Methodology, vol. 22 ,no.4, 1-40,2013, DOI : 10.1145/2522920.2522924

[12] Jones, J. A., Harrold, M. J., Stasko, J., “Visualization of test information

to assist fault localization". Proceedings of the 24th international conference on Software engineering – ICSE '02, pp.467,2002, DOI:10.1145/581339.581397

[13] Gemund, A. J. C. V., "An Evaluation of Similarity Coefficients for Software Fault Localization". Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing-PRDC '06, pp. 39-46,2006, DOI:10.1109/PRDC.2006.18

[14] Naish, L., Lee, H. J., Ramamohanarao, K., "A model for spectra-based software diagnosis". ACM Transactions on Software Engineering and Methodology, vol. 20, no.3, pp.1-32,2011, DOI:10.1145/2000791.2000795

[15] Perez, A., Abreu, R., Wong, W. E., "A Survey on Fault Localization Techniques", 2004.

[16] Hsu, H. Y., Jones, J. A., Orso, A., "Rapid: Identifying bug signatures to support debugging activities". ASE 2008 -23rd IEEE/ACM International Conference on Automated Software Engineering, Proceedings, pp. 439-442,2008, DOI:10.1109/ASE.2008.68

[17] Liblit, B., Naik, M., Zheng, A. X., Aiken, A., Jordan, M. I., "Scalable statistical bug isolation". ACM SIGPLAN Notices, vol. 40, no. 6, pp. 15, 2005, DOI:10.1145/1064978.1065014

[18] Arumuga Nainar, P., Chen, T., Rosin, J., Liblit, B.,"Statistical debugging using compound Boolean predicates". Proceedings of the 2007 international symposium on Software testing and analysis - ISSTA '07, 2007 DOI:10.1145/1273463.1273467

[19] Baah, G. K., Podgurski, A., Harrold, M. J., "The probabilistic program dependence graph and its application to fault diagnosis". IEEE Transactions on Software Engineering, vol.36, no.4, pp. 528-545, 2010, DOI: 10.1109/TSE.2009.87

[20] Yu, K., Lin, M., Gao, Q., Zhang, H., Zhang, X., "Locating faults using multiple spectra-specific models". Proceedings of the 2011 ACM Symposium on Applied Computing - SAC '11, pp.1404-1410, 2011, DOI: 10.1145/1982185.1982490

[21] Tiantian, W., Xiaohong, S., Peijun, M., Kechao, W., "Comprehension oriented software fault location". Proceedings of 2011 International Conference on Computer Science and Network Technology, pp. 340 - 343, 2011, DOI: 10.1109/ICCSNT.2011.6181971

[22] Cheng, H., Lo, D., Zhou, Y., Wang, X., Yan, X., "Identifying bug signatures using discriminative graph mining". Proceedings of the eighteenth international symposium on Software testing and analysis-ISSTA '09, pp. 141, 2009, DOI:10.1145/1572272.1572290

[23] Ernst, M. D., Cockrell, J., Griswold, W. G., Notkin, D., "Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering, vol.27, no. 2, 99-123, 2001, DOI:10.1109/32.908957

[24] Wong, W. E., Gao, R., Li, Y., Abreu, R., Wotawa, F.,"A survey on software fault localization". IEEE Transactions on Software Engineering, vol.42, no. 8, pp. 707-740, 2016, DOI: 10.1109/TSE.2016.2521368

[25] Di Fatta, G., Leue, S., Stegantova, E., "Discriminative pattern mining in software fault detection". Proceedings of the 3rd international workshop on Software quality assurance, pp. 62-69, 2006, DOI: 10.1145/1188895.1188910

[26] Liu, C., Yan, X., Yu, H., Han, J., Yu, P. S., " Mining Behaviour Graphs for "Backtrace" of Noncrashing Bugs", Proceedings of the 2005 SIAM International Conference on Data Mining, pp. 286-297, 2005, DOI: 10.1137/1.9781611972757.26

[27] Lo, D., Han, J., "Classification of Software Behaviors for Failure Detection: A Discriminative Pattern Mining Approach". Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '09, pp. 557-565, 2009, DOI: 10.1145/1557019.1557083

[28] Cheng, H., Yan, X., Han, J., Yu, P. S., "Direct Discriminative Pattern Mining Classification Effective", IEEE 24th International Conference on Data Engineering, pp. 169-178, 2008, DOI: 10.1109/ICDE.2008.4497425

[29] Yan, X., Cheng, H., Han, J., Yu, P. S., "Mining significant graph patterns by leap search". Proceedings of the 2008 ACM SIGMOD Int. conf. on Management of Data, pp. 433-444, 2008, DOI:10.1145/1376616.1376662

[30] Eichinger, F., Böhm, K., Huber, M., "Mining edge-weighted call graphs to localise software bugs". Lecture Notes in Computer Science (LNCS) 5211 LNAI (PART 1), pp. 333-348, 2008, DOI: 10.1007/978-3-540-87479-9_40

[31] Yu, Y., Jones, J. A., Harrold, M. J., "An empirical study of the effects of test-suite reduction on fault localization". Proceedings of the 13th international conference on Software engineering - ICSE '08, 201, 2008, DOI:10.1145/1368088.1368116

[32] Abreu, R., Zoeteweij, P., Golsteijn, R., van Gemund, A. J. C., "A practical evaluation of spectrum based fault localization". Journal of Systems and Software", vol. 82, no.11, pp. 1780-1792, 2009. DOI: 10.1016/j.jss.2009.06.035

[33] Cleve, H., Zeller, A., "Locating causes of program failures. Software Engineering", ICSE 2005.Proceedings. 27th International Conference, pp. 342-351, 2005, DOI: 10.1109/ICSE.2005.1553577

[34] Zeller, A.," Isolating cause-effect chains from computer programs". Proceedings of the tenth ACM SIGSOFT symposium on Foundations of software engineering - SIGSOFT '02/FSE 10, pp.1.2002, DOI:10.1145/587051.587053

[35] Wong, W. E., Qi, Y., "Effective program debugging based on execution slices and inter-block data dependency". Journal of Systems and Software, vol.79, no.7, pp. 891-903, 2006, DOI:10.1016/j.jss.2005.06.045

[36] Seung-Seok, C., Sung-Hyuk, C., Tappert, C. C., 2010. A "Survey of Binary Similarity and Distance Measures". Journal of Systemics, Cybernetics & Informatics, vol.8, no.1, pp. 43-48, 2010.

[37] Wong, W. E., Debroy, V., Gao, R., Li, Y., "The DStar method for effective software fault localization. IEEE Transactions on Reliability, vol. 63, no.1, 290-308,2014, DOI:10.1109/TR.2013.2285319

[38] The Software Infrastructure Repository, [Online].Available: https://sir.unl.edu/portal/index.html

[39] Wong, W. E., Debroy, V., Xu, D., "Towards better fault localization: A crosstab-based statistical approach". IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews, vol. 42, no. 3, pp. 378-396, 2012, DOI:10.1109/TSMCC.2011.2118751

[40] Do, H., Elbaum, S., Rothermel, G., "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact". Empirical Software Engineering, vol. 10, no. 4, pp. 405-435, 2005, DOI: 10.1007/s10664-005-3861-2

[41] Eric Wong, W., Debroy, V., Choi, B., "A family of code coverage-based heuristics for effective fault localization." Journal of Systems and Software. vol.83, no. 2, pp. 188-208,2010,DOI:10.1016/j.jss.2009.09.037

[42] Bookstein, A., Kulyukin, V. a., Raita, T, "Generalized Hamming Distance". Inf. Retr.,vol.5,no.4, pp.353-375, 2002, DOI: 10.1023/a:1020499411651

[43] Jones, J. J. a., Harrold, M. J. M., "Empirical evaluation of the tarantula automatic fault localization technique". Automated Software Engineering, pp. 282-292,2005, DOI:10.1145/1101908.1101949

[44] W. E. Wong, V. Debroy, R. Golden, X. Xu and B. Thuraisingham, "Effective Software Fault Localization using an RBF Neural Network," IEEE Transactions on Reliability, Volume 61, Number 1, pp. 149-169, March 2012

Marwa Gaber Abd El-Wahab is currently a Teaching Assistant at Computer Science Department, Future Academy, Cairo, Egypt. She received her B.Sc. in Computer Science department, Faculty of Computers and Information, Helwan University, Cairo, Egypt. She is currently a M.Sc. student at Computer Science department, Faculty of Computers and Information, Helwan University.

Amal Elsayed Aboutabl is currently an Associate Professor at the Computer Science Department, Faculty of Computers and Information, Helwan University, Cairo, Egypt. She received her B.Sc. in Computer Science from the American University in Cairo and both of her M.Sc. and Ph.D. in Computer Science from Cairo University. She worked for IBM and ICL in Egypt for seven years. She was also a Fulbright Scholar at the Department of Computer Science, University of Virginia, USA. Her current research interests include software engineering and natural language processing.

Wessam M.H. EL Behaidy is currently a Teacher Assistant in faculty of Computers and Information, Helwan University, Cairo, Egypt. She received her B.Sc. is in Computer Science, from faculty of Computers and Information, Helwan University, Cairo, Egypt in 2000. Also, she earned her M.Sc. and Ph.D. in Computer Science, Helwan University in 2004 and 2012 respectively. Her research interests also include structural bioinformatics, protein structure prediction, machine learning, and image processing. She published 6 papers in international conferences and Springer.