IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 13, 2019, accepted April 3, 2019, date of publication April 11, 2019, date of current version April 25, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2910327

A Community-Based Fault Isolation Approach for Effective Simultaneous Localization of Faults

ABUBAKAR ZAKARI[1,2], SAI PECK LEE[1], AND IBRAHIM ABAKER TARGIO HASHEM[3]

[1]Faculty of Computer Science and Information Technology, University of Malaya, Kuala Lumpur 50603, Malaysia
[2]Department of Computer Science, Kano University of Science and Technology, Wudil, P.M.B 3244, Kano, Nigeria
[3]School of Computing and IT, Taylor's University, Subang Jaya 47500, Malaysia

Corresponding authors: Abubakar Zakari (abubakar.zakari@yahoo.com), Ibrahim Abaker Targio Hashem (ibrahimabaker.targiohashem@taylors.edu.my), and Sai Peck Lee (saipeck@um.edu.my)

This work was supported by the research project FP001-2016 under the Fundamental Research Grant Scheme provided by the Ministry of Higher Education, Malaysia.

ABSTRACT During program testing, software programs may be discovered to contain multiple faults. Multiple faults in a program may reduce the effectiveness of the existing fault localization techniques due to the complex relationship between faults and failures in the presence of multiple faults. In an ideal case, faults are isolated into fault-focused clusters, each targeting a single fault for developers to localize them simultaneously in parallel. However, the relationship between faults and failures is not easily identified and depends solely on the accuracy of clustering, as such, existing clustering algorithms are not able to isolate failed tests to their causative faults effectively which hinder localization effectiveness. This paper proposes a new approach that makes use of a divisive network community clustering algorithm to isolate faults into separate fault-focused communities that target a single fault each. A community weighting and selection mechanism that aids in prioritizing highly important fault-focused communities to the available developers to debug the faults simultaneously in parallel is also proposed. The approach is evaluated on eight subject programs ranging from medium-sized to large-sized programs (tcas, replace, gzip, sed, flex, grep, make, and ant). Overall, 540 multiple-fault versions of these programs were generated with 2–5 faulty versions. The experimental results have demonstrated that the proposed approach performs significantly better in terms of localization effectiveness in comparison with two other parallel debugging approaches for locating multiple faults in parallel.

INDEX TERMS Complex network, multiple faults, fault localization, fault isolation, program debugging, parallel debugging.

I. INTRODUCTION

Software debugging is a costly and time-consuming activity. A software may often contain faults after deployment regardless of the effort put into its development. The bigger and more sophisticated a software, the higher the likelihood of the software containing faults [1]. When a failure is observed during program testing, the program is said to have contained one or more faults. Therefore, software developers are obliged to locate and fix the faults by identifying the exact location of the faults that caused the failure. This process is generally known as fault localization, which is a tedious and expensive process [2], [3]. Many studies made a single-fault assumption when a program fails which may not hold in practice, because program failures are or might be triggered by many faults. However, identifying tests executions that failed due to the same fault is difficult. Hence, understanding the *due-to* relationship between failures and their corresponding faults is quite difficult [4].

Various studies have utilized the parallel debugging approach to isolate faults into different clusters for more effective localization of multiple faults [5]–[8]. Parallel debugging approach utilizes clustering algorithms to create *fault-focused* clusters that may target single fault each. However, isolating different faults into separate clusters is a very challenging task [1]. In order to isolate faults into different *fault-focused* clusters in parallel debugging approach, tests execution are mostly clustered based on their execution profile similarity [5]–[7]. However, as suggested by Liu *et al.* [9] and Gao and Wong [1], this representation is problematic because faults can be triggered in many different ways. Hence, a cluster may contain more than one

The associate editor coordinating the review of this manuscript and approving it for publication was Basit Shahzad.

fault caused by different failures which can reduce the localization effectiveness of a fault localization technique [1], [5], [6], [8], [10], [11].

Moreover, some obvious challenges during clustering include the know-how to accurately depict a failed test case or to properly measure the variations between two failed test cases based on the distance metric used, and also to conduct clustering based on a suitable clustering algorithm. Therefore, choosing the right clustering algorithm is vital because various solutions to these challenges have different effect to clustering results (accuracy) which will directly affect fault localization effectiveness [6], [12], [13]. In addition, in view of the problem of clustering accuracy on fault isolation in parallel debugging approach when localizing multiple faults simultaneously, a new approach to isolate faults is needed to improve the creation of viable *fault-focused* clusters that target single faults to improve localization effectiveness on multiple-fault programs.

In this paper, we propose a new approach that makes use of a divisive network community clustering algorithm to perform clustering (Section III.D). Although the clustering algorithm used in this paper is a well-known clustering algorithm and has been utilized in the studies of graph theory [14], [15], to the best of our knowledge, this algorithm has not been used in the field of software fault localization. Based on this algorithm, a developer is tasked to find the least connected statements in a program modeled network and then remove the edges between them (connection between statements). Hence, communities are created by continuously removing the edges from the program network based on statements edge-betweenness distance. For any statement in a network, edge-betweenness is defined as the number of shortest paths between pairs of statements that run along it. This process is done repeatedly until the program network is naturally divided into smaller and smaller components composed of densely connected statements that form communities. We name these communities as *fault-focused* communities.

We further introduce a community weighting and selection mechanism to aid in prioritizing highly important *fault-focused* communities to the available developers to debug the faults simultaneously in parallel. Lastly, this work builds on top of our previous work [16], where we proposed a fault localization technique based on complex network theory. Hence, we utilize the same technique in this study for fault localization. Community clustering algorithms have proven to be useful in understanding complex data relationships for community detection in the domain of software engineering [17], physics [18], and social networks [14], [15]. Hence, based on the existing knowledge in the literature that the more densely connected nodes are in a network, the more prone they are to be related to the same variable or information [17], [18]. In extension to this study, we postulated that the more densely connected program statements are in the program network, the more prone they are to be related to the same failure or fault.

The proposed approach is evaluated on eight subject programs ranging from medium-sized to large-sized programs (*tcas, replace, gzip, sed, flex, grep, make,* and *ant*). Overall, 540 multiple-fault versions of these programs were generated with 2, 3, 4, and 5 faulty versions (Section IV), respectively. Furthermore, a cross-comparison between the proposed approach and two other parallel debugging approaches is conducted in Section V [1], [5]. The experimental results strongly demonstrate that the proposed approach outperforms the two parallel debugging approaches in terms of localization effectiveness. In this paper, we apply these terminologies interchangeably, where program statements are nodes; executions or connections between program statements are represented as edges, while clusters are represented as communities.

The major contribution of this paper is as follows:

- A new approach is proposed that makes use of a divisive network community algorithm for clustering to aid in the isolation of faulty program statements caused by different faults into separate *fault-focused* communities (clusters), each targeting a single fault.
- A novel community weighting and selection mechanism is introduced to aid in identifying and prioritizing highly important *fault-focused* communities to available developers to debug the faults simultaneously in parallel.
- An empirical study to evaluate the proposed approach, showing the effectiveness of the approach using eight subject programs ranging from medium-sized to large-sized programs (*tcas, replace, gzip, sed, flex, grep, make,* and *ant*) with 2, 3, 4, and 5 faulty versions. The empirical result shows that the proposed approach performs better in terms of localization effectiveness in comparison with the two other parallel debugging approaches.

The remaining parts of the paper are organized into different sections. Section II highlights the related work. Section III presents the proposed approach in detail. Section IV gives the experimental setup. Section V discusses the result and discussion. Section VI presents the threat to validity. Finally, the study is concluded in Section VII.

II. RELATED WORK

In this section, some of the related work on parallel debugging are highlighted.

In the last two decades, researchers in the software fault localization domain have proposed various approaches and techniques to aid in isolating faults to support in debugging multiple faults in parallel. In an earlier study by Podgurski *et al.* [19], the idea of clustering failed tests executions based on a behavioral model on multiple fault subjects was explored. The authors grouped tests execution profiles to correlate failure causes and to isolate failures that are caused by different faults. In another study by Dickinson *et al.* [20], a control flow-based clustering technique was utilized to cluster tests executions based on their

failure causes. Liu and Han [21] grouped together executions due to the same fault using R-proximity. This proximity regards two failed executions as identical if the two executions suggest the same fault location. The work by Huang *et al.* [22] performed an empirical study on the performance of two different types of clustering techniques, namely k-means clustering and hierarchical clustering on six fault localization techniques to analyze their effectiveness. The result showed that k-means clustering had the best performance in isolating faults. On the same note, few other studies have utilized k-means clustering for the same purpose [23], [24].

A study by Briand *et al.* [25] used the C4.5 decision tree algorithm which is a machine learning technique to classify test cases into different partitions. Failed test cases in the same partition are regarded to have a high probability of failing due to the same fault. Hence, each partition is handled by developers to simultaneously localize the faults effectively. In the study by Liblit *et al.* [26], the authors proposed an algorithm to isolate faults in programs with multiple faults. The algorithm identifies predicates that are correlated with specific singular faults and isolate them in order to prioritize debugging effort and localize faults simultaneously. An approach named Hierarchy-Debug [27] was proposed to localize latent faults. The approach uses a hierarchical clustering algorithm to cluster predicates to support in localizing multiple faults. The result shows that the approach can aid developers in grouping predicates caused by multiple faults to localize them simultaneously. Jones *et al.* [5] presented two techniques for clustering failures (clustering based on execution profile and clustering based on fault localization results). The clustering techniques are aimed to partition failed tests executions into separate clusters that are targeting a different fault each. Hence, each cluster will be given to a single developer to locate the faults simultaneously in parallel. However, the authors postulated that the identified number of clusters predicts the number of faults in a program. In another study [23], a technique to calculate the complexity of identified clusters before distributing the work to the respective debuggers was proposed. After clustering, the technique calculates the weight of each cluster to determine the effort needed to debug the cluster for a more effective debugging task distribution. In the study by Wei and Han [28], a parameter-based combination approach (PBC) was proposed to aid in localizing multiple faults simultaneously. A bisection method was utilized for clustering failed tests executions to formulate *fault-focused* clusters while a crosstab-based fault localization technique was used for localization. Furthermore, a machine-learning fault localization technique based on RBF (radial basis function) neural network was proposed by Wong *et al.* [7]. This technique isolates faults using a clustering technique based on a behavioral model to cluster tests executions. The generated clusters are presumed to target a single fault each. These clusters are given to developers to debug the faults simultaneously in parallel. A recent study in [1] proposed a novel technique for localizing multiple faults in parallel. The authors proposed an improved k-medoids clustering algorithm to aid in the effective identification of the relationship between failed test cases and their corresponding faults. The study concluded that their proposed technique performs better in terms of efficiency and effectiveness in comparison to other techniques.

Looking at these studies, failed test cases are often clustered to generate *fault-focused* clusters. However, taking into account all tests execution for clustering failure is something worth considering. Due to the fact that in the presence of multiple faults, faulty statements are often executed by passed tests execution [29], [30]. In this study, a program network N is modeled using both passed and failed tests execution to capture the entire program execution behavior. Using this information, we compute the network community clustering algorithm as described in Section III.D to identify natural community divisions. The readers should be reminded that the program networks in this study are not real-world networks, rather they are artificial networks depicting the overall tests execution behavior. However, based on a previous study [15], the utilized community detection algorithm works well on networks of both types. Each of these communities will be given to a separate developer to debug the faults in parallel. This is based on our postulation that each community contains a single fault. In the next section, our approach will be highlighted in detail.

III. THE APPROACH

In this section, we first revisit our previous work in Section III.A. Then, the tests representations in the existing works and our work are described in Section III.B and Section III.C. The divisive network community clustering algorithm is outlined in Section III.D. A discussion on how to calculate the shortest-path betweenness of a statement is outline in Section III.D.1. We further describe in detail the community (cluster) weighting and selection process which will help in selecting and prioritizing identified *fault-focused* communities for effective simultaneous localization in Section III.D.2. The general framework of our approach is highlighted in Section III.E. Lastly, a running example is given to illustrate the proposed approach in Section III.F.

A. REVISITING OUR PREVIOUS WORK

In order to fully appreciate the work in this paper, we need to revisit our previous work in [16]. We proposed a new fault localization technique based on complex network theory named FLCN. The technique aids in the effective localization of faults in both single-fault and multiple-fault programs. For a given program spectra of passed and failed tests execution for a software program under test, an undirected/unweighted network N is modeled to capture the entire program execution behavior with statements represented as nodes and the execution between the statements as edges irrespective of whether they are executed by passed or failed test cases. The technique takes into account two factors which are: statements behavioral abnormalities and the distance between program

statements to quantify the suspiciousness value of a given statement. The result of our work has shown a significant improvement in terms of effectiveness in the localization of faults simultaneously in a single diagnosis rank list.

However, we observed that despite this success, in some cases, more debugging iterations are needed to locate all the faults in a multiple-fault program which will further reduce the efficacy of the technique and also increase the time-to-delivery of the software. Therefore we have two options: 1). Resort to using the one-bug-at-a-time debugging approach (OBA). 2). Use the parallel debugging approach. We understand that using the OBA approach will further increase the time-to-delivery of the software product and will not accommodate the simultaneous localization of the faults. This is because, OBA debugging approach is the process whereby a developer finds the first fault, fix it, and then re-test the program iteratively to see if the program is failure-free or not. Hence, for each iteration, one fault is neutralized. That is the reason why the parallel debugging approach is considered. In this work, we use the latter to extend our work by parallelizing the debugging task to generate *fault-focused* communities using the clustering algorithm in Section III.D to aid in the simultaneous localization of faults and also to facilitate the process in producing a failure-free program.

B. TEST REPRESENTATIONS IN EXISTING WORKS

To localize multiple faults simultaneously in parallel, program spectra is primarily used in this process. Program spectra refers to the runtime profile of program statements in correspondence to the test cases that execute them. In other words, the test results (passed/failed) and test coverage information form the program spectra. In the existing works, failed test cases are often used to generate the *fault-focused* clusters that target a single fault each [1], [5], [6]. Each of these *fault-focused* clusters of failed test cases will be combined with all passed test cases to localize the faults simultaneously in parallel. Hence, for the given failed tests, to understand the *due-to* relationship or the know-how (as explained in Section 1) between them, various representations are used. One of the most prominent is the grouping of failed tests execution based on the similarity of their execution profiles which is used in some multiple fault localization studies [9], [22], [23], [31]. Other representations used a more advanced technique in clustering failed tests due to the same faults. The study by Gao and Wong [1] uses a distance metric named Kendall tau distance in its revised form to measure the *due-to* relationship between failed tests with an approach that simultaneously estimates the number of clusters. The latter representation has shown to be better than the former. In this paper, we use neither of these representations (Section III.C).

C. TEST REPRESENTATION IN OUR WORK

In contrast with the test representations highlighted in Section III.B, the program spectra in our work is transformed to a network N as discussed in Section III.A. In almost all the existing studies, failed tests execution are specifically isolated and clustered to generate the *fault-focused* clusters. However, in this work, the network N is the representation of all the program execution profile of both passed and failed tests execution. Hence, our approach does not perform clustering on tests execution rather it performs clustering on program statements in the network that is modeled based on the tests execution profile. Therefore, the *due-to* relationship between program statements is measured to create *fault-focused* communities instead of between failed test cases in the existing works (Section III.B). As further discussed in Section III.D, the program statements that are more densely connected together are considered to be more likely to be in the same community (cluster), in extension, more likely to be caused by the same faults. Therefore, in computing the *due-to* relationship between the program statements, edge-betweenness distance is calculated between program statements where edge is the connection between program statements. The higher the edge-betweenness value between statements, the less likely they are to be in the same community.

D. COMMUNITY CLUSTERING ALGORITHM

To cluster failures in the fault localization domain, various clustering algorithms were utilized in recent years [1], [5], [6], [13]. These studies mainly use program tests execution profile to isolate faults into distinct clusters with tests execution profile similarity often used to justify failure groupings. Hence, distance metrics such as Euclidean distance, Jaccard distance, and Hamming distance are often used to compute the test-to-test distance to determine the cluster a given test will fall into [5], [7], [22]. However, recent studies have shown that the use of this representation is problematic and is not an effective way to isolate failed tests based on their causative faults [1], [9]. In contrast, for our approach (following the tests representation in Section III.C), instead of using the traditional distance metrics (Euclidean distance, Jaccard distance, Hamming distance) which is somewhat less effective for measuring the *due-to* relationship between tests, edge-betweenness based distance is used by our divisive network community clustering algorithm to measure the distance between program statements (nodes) executions. Network community clustering algorithms fall into two classes which are agglomerative and divisive [32]. Algorithms are classified based on whether they concentrate on addition or removal of edges to a network or out of a network.

In this paper, we use the latter (divisive) clustering method to isolate faults into different communities. Based on this method, a developer is tasked to find the least connected statements in a faulty program network and then remove the edges between them. If this process is done repeatedly, the program network will naturally be divided into smaller and smaller groups composed of densely connected statements. These smaller groups can be considered as the network communities when the process is halted. This is based on the knowledge that the more densely connected nodes are in a network, the more prone they are to be related to the same variable or information [17], [18]. In extension to this study,

we postulated that the more densely connected program statements are in the program network, the more prone they are to be related to the same failure or fault. Our approach to community identification in this paper basically follows these lines. We use the algorithm proposed by Girvan and Newman for community detection [14], [15].

We generate communities by continuously removing edges (connection between program statements) from the modeled program network based on their betweenness centrality score value. Program statement betweenness is defined as a measure of statements centrality and influence in a network [33]. The betweenness centrality measures statement influence based on information flow from a statement to other statements in a network, particularly where the information flow in the network follows the shortest available paths. Hence, to identify the statements in a network that are mostly between other statements, the betweenness centrality is generalized to network edges as edge-betweenness. Edge-betweenness of an edge is defined as the number of the shortest paths between statements that run along it. For statements that have more than one shortest paths, all the paths will be given equal weight so that the total weight of all paths is unified. In a scenario where a network has communities or clusters that are loosely connected by intergroup edges as shown in Figure 2, the shortest paths between these communities must move along these few edges. Therefore, edges connecting communities will normally have high edge-betweenness score value. The community structure of the network can be revealed in distinct groups by removing these edges.

The algorithm used in this study for community detection in its general form is stated in detail as follows:

Algorithm 1 Girvan Newman Algorithm

Input: Undirected and unweighted program network $N(n, e)$

Output: List of identified communities

1: Calculate the betweenness of all edges in the network

2: Identify and remove the edges that has the highest betweenness score

3: Recalculate betweenness for all the remaining edges affected by the removal

4: Repeat from step 2 until no edges remain

5: End

We calculate the betweenness of the networks using Newman fast algorithm [34]. This algorithm calculates the betweenness for all edges e in a network of n nodes in best-case time O(en) as claimed in [15]. The betweenness of edges that are affected by the removal of the edge in step 2 will be recalculated. Therefore, this algorithm calculates the betweenness score of each edge starting from statement in N until the betweenness score is computed from each and every statement in N. Henceforth, the betweenness scores of the edges from each and every statement will be added up and divided by 2 to get the final edge-betweenness scores of all the edges in N. To get a better result, the recalculation step of the algorithm is the most vital step. Therefore, the recalculation step is very crucial in detecting good communities in the program network N.

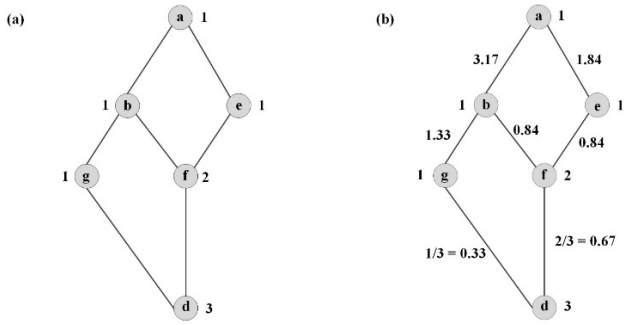

FIGURE 1. Calculation of shortest-path betweenness.

1) SHORTEST-PATH BETWEENNESS

In this study, to calculate the shortest path between program statements in a program modeled network N, it can be done using breadth-first search in time O(e) with O(n^2) statements pairs [34], [35]. Breadth-first search can find the shortest paths from a statement m_i to all other statements in time O(e) where e represents the edge between statements. Figure 1 shows an example of a shortest path "tree" for a simple network. Figure 1a shows a simple network that illustrate how breadth-first search find the shortest paths between statements in time O(e). The number of shortest paths from statement m_a to itself is weight $w_a = 1$ as an initial condition. For any other statements that are directly next to m_a, in this case, m_b and m_e, they will be given equal weight as m_a. In the case of m_f, the number of shortest paths from m_a to m_f is the number of shortest path from m_a to m_b plus the number of shortest path from m_a to m_e. Therefore, m_f will carry the weight of 2 as $w_f = 2$. In other words, because m_a has multiple paths to m_f and each path holds a weight of 1. Therefore, the weight of m_f will be the weight of m_a to m_b plus the weight of m_a to m_e which equals to a total weight of $w_f = 2$ for m_f.

However, the number of shortest paths from m_a and m_b is the same as the number of shortest paths from m_a to m_g, because we have to go through the predecessor m_b in a single direction. Next, to know the number of shortest paths from m_a to m_d (the lowest statement) we will sum up the number of shortest paths from m_a to m_g and the number of shortest paths from m_a and m_f. So, m_d will carry the weight of 3 ($w_d = 3$). We can now use the shortest path of the statements to calculate the betweenness for each edge in the network. Figure 1b shows the betweenness score of each edge in the network. We first start with the edges that are farthest from the initial statement (m_a) which is the lowest edge. Then, we work upwards assigning a score to each edge that is 1 plus the sum of the scores on the edge or edges immediately below it. When we have go through all edges in the network,

the resulting scores of the edges are the betweenness score for the paths from m_a(the betweenness scores of the edges are to be calculated from all the remaining statements as well). This process will be repeated for all statements, and the betweenness scores of all edge will be added up and divided by 2. With this, the final edge-betweenness scores for the shortest paths between all statements will be obtained.

The breadth-first search and the process of calculating the betweenness of all edges in the network both takes worst-case time O(e). With n statements total, therefore the whole calculation is done in best-case time O(en) [15]. In an earlier definition of node betweenness [33], if node has multiple shortest paths (i.e. m_f in Figure 1), all paths leading to the node will be given equal weights summing to 1. For instance, if there are two shortest paths, each paths will be given the weight of $\frac{1}{2}$.

Looking at Figure 1a, to conduct a breadth-first search starting from m_a, the following steps will be performed:

1) The initial Statement m_a will be assigned a distance $d_a = 0$ and a weight $w_a = 1$.
2) For any statement m_i that is next to m_a, a distance $d_i = d_a + 1 = 1$, and a weight $w_i = w_a = 1$ will be given to it.
3) Next, for each statement m_j next to the statement m_i, the following three things are conducted:
   - If m_j is not assigned any distance, a distance $d_j = d_i + 1$ will be assigned and a weight $w_j = w_i$ will be assigned.
   - Moreover, if a distance has already been assigned to m_j as $d_j = d_i + 1$, then, the weight of the statement will be rise by w_i which is $w_j \leftarrow w_j + w_i$.
   - If m_j has already been assigned a distance and $d_j < d_i + 1$, nothing will be done.
4) Lastly, repeat from step 3 until no statements left that have assigned distances but whose neighbors do not have assigned distance.

Furthermore, the weight of a statement m_i indicates the number of different paths from the root statement m_a to m_i. Therefore, these weights are exactly what we want to calculate the edge-betweenness where for two statements m_i and m_j that are connected, with m_j farther than m_i from the root statement m_a, then the shortest paths from m_j through m_i to the root statement m_a will be given by w_i/w_j. To further calculate the edge-betweenness from all the shortest paths starting from m_a as shown in Figure 1b, the following steps will be carried out:

1) Calculate the shortest paths starting from statement m_a, to every other statement using the breath-first search in time O(e).
2) Find the lowest statement m_j (i.e. statement at the bottom of the program network N)
3) For each given statement m_i next to statement m_j, assign a score to the edge from m_i to m_j of w_i/w_j.
4) Next, start from the edges that are far away from the root statement m_a (statements edges that are at the bottom of the diagram in Figure 1) moving upward. For the edge of m_i to m_j, with m_j being far away from m_a than m_i, a score that adds 1 to the sum of the neighboring edges immediately below it will be assign (i.e. edges below it that share common statements). At last, the score will be add up by the weight w_i/w_j.
5) Lastly, repeat step 3 and step 4 until m_a is reached.

Using these steps, we are able to calculate the edge-betweenness of all edges in the program network in best-case time O(en). This calculation will be repeated for each edge removed from the program network N. Given we have e amount of edges where $e = \{e_1, e_2, e_3, \ldots, e_n\}$. The whole community structure algorithm based on the shortest path betweenness will run in worst-case time O(e^2n) or O(n)^3. We experience that, unlike networks with stronger community structures in other research domains [15], [36], a network modeled based on tests execution profile (program spectra) have less community density (mostly sparse). Therefore, the time O(en) it takes to generate *fault-focused* communities is relatively longer but have less effect in the quality of communities created for fault localization.

2) COMMUNITY WEIGHTING AND SELECTION

In practice, when a program fails, a developer does not normally know the number of faults that caused the failure. In extension, when identifying network communities (clusters), the developer probably do not know how many communities the algorithm is going to generate depending solely on network to network modularity [15]. Therefore, there is no reason for the identified communities to be roughly of the same size. For the above practical scenario where the developer does not know the exact number of faults and the exact number of communities or their sizes, localizing those faults might be a bit tricky. Therefore, we introduce a community weighting and selection mechanism to aid in prioritizing highly important communities to the available developers to debug the faults simultaneously in parallel. In this work, we name these communities as *fault-focused* communities which target a single fault each.

For a given community C in a network N, a weight will be assigned to each community based on the total number of statements n a given community contains. Equation 1 will calculate the weight of a single community in network N.

$$C = \sum_{j=1}^{n} m_j \tag{1}$$

where C represents a community, n represents the total number of statements in that community, and m represents a statement. Therefore, the number of communities can be represented as $C = \{c_1, c_2, c_3, \ldots, c_n\}$ in a given program network N.

Furthermore, the weight of all the communities in N will be computed. Suppose there is a network N with three identified communities as shown in Figure 2 where the first community has 7 statements, the second community has 6 statements,

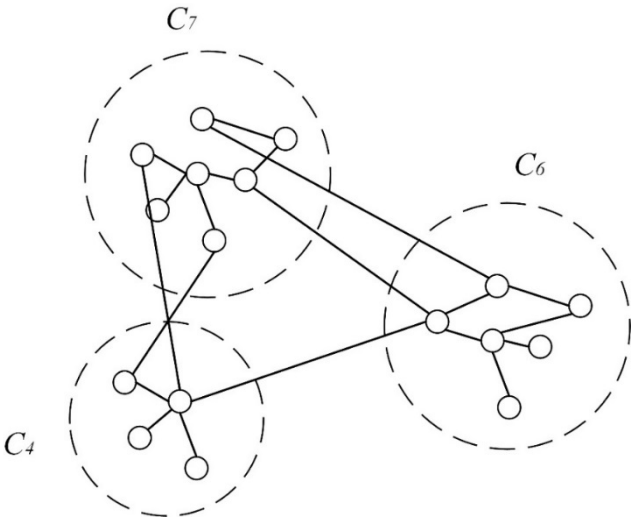

FIGURE 2. Network with group of communities and their weights.

and the third community has 4 statements. Therefore, if a given community has a higher weight with greater number of statements in comparison with other communities in N, that community will be ranked at the top of the community ranking list followed by subsequent communities. All communities will be generated in descending order based on the weights they carried and will be given to developers to debug the faults simultaneously in parallel starting with the community with the highest weight. The main reason why developers start with the community with the highest weight is that, we have postulated that communities with a high number of statements to be more susceptible to contain the faulty program statements.

E. THE COMMUNITY-BASED FAULT ISOLATION APPROACH

In this section, we present the detail steps of the proposed community-based fault isolation approach. Figure 3 demonstrates the overall process of the approach, with a much detail of the process given as follows:

Step 1: *Program tests execution profile*: The first phase of the approach focuses on program execution and collecting the tests execution profile. The faulty program P under test will be executed with its corresponding available test cases t to generate the tests execution profile (program spectra). Test case executions are classified into passed and failed categories depending on whether the output deviates from the expected output or not. Therefore, if a test case produces an expected output, the test case has passed. Otherwise, the test case has failed.

Step 2: *Network modeling*: In this phase, the network N is modeled. The execution profile (program spectra) of both passed and failed test cases obtained from the initial phase will be used as an input to generate the network N that captures the entire program executable behavior [16], [37].

TABLE 1. A program with tests execution.

	mid ( ) { input x, y, z, m;	t_1	t_2	t_3	t_4	t_5
m_1	if (y < z & x< y){	1	1	1	1	1
m_2	m = z; //fault 1 m = y	1	1	1	1	1
m_3	else	1	1	1	1	1
m_4	if (x > z)	1	0	1	1	1
m_5	m = y; //fault 2 m = x	0	0	1	1	1
m_6	}	1	0	0	0	1
m_7	print (“middle number is:”, m);	0	0	0	0	0
m_8	}	0	1	1	1	0

Step 3: *Community (cluster) detection*: The network community clustering algorithm in Section III.D will be computed at this stage. Statements that are densely connected with each other will be isolated into distinct *fault-focused* communities by taking into account the statements edge-betweenness distance as discussed in Section III.D.1. Therefore, the number of existing communities in a given network will be known by the developer.

Step 4: *Community (cluster) weighting and selection*: After knowing the number of communities in N, a developer needs to know where to start the debugging task which can be tricky especially if there are many available communities. Furthermore, not all the communities might contain faulty program statements. Therefore, the number of communities can possibly be larger than the number of faults or vice versa. In this stage, the community weighting and selection mechanism in Section III.D.2 will be used to identify and rank the most fault-prone communities with high possibility of containing faulty statements. The *fault-focused* communities will be generated in descending order based on the weights they hold for developers to work with and localize the faults simultaneously in parallel.

Step 5: *Fault localization*: The fault localization technique based on complex network theory will be used for fault localization [16]. Ultimately, faults will be found and neutralized by each developer. The program will be retested again to see if the debugging is successful.

F. A RUNNING EXAMPLE

Consider a running program sample in Table 1 for discussing how our approach will work for fault localization. The program has eight statements and is executed with five test cases. Statements m_2 and m_5 are both faulty with two failed test cases (t_2 and t_3) and three passed test cases (t_1, t_4, and t_5), respectively.

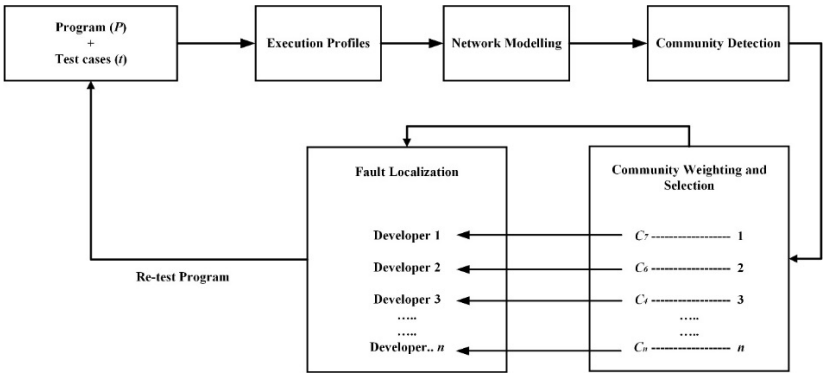

FIGURE 3. General framework of the proposed approach.

TABLE 2. Final edge-betweenness score for each edge in the network.

Edges	m_1 - m_2	m_2 - m_3	m_3 - m_4	m_4 - m_5	m_3 - m_8	m_4 - m_6	m_8 - m_5	m_5 - m_6
Edge-betweenness score	5.99	9.99	6.82	3.32	6.32	2.99	5.16	5.08

In the next step, the network N will be modeled so as to capture the entire program executable behavior on both passed and failed tests execution [16]. For the first test case t_1, there is an edge from m_1 to m_2, m_2 to m_3, m_3 to m_4, m_4 to m_6. For the second test case t_2, we add additional edge from m_3 to m_8. For the third test case t_3, an edge is added from m_4 to m_5, and from m_5 to m_8. Test case t_4 will be ignored, because it has the same execution profile with t_3. Therefore, all the resulting edges will be redundant. For t_5, an edge from m_5 to m_6, is added. Therefore, step 1 and step 2 are completed.

Moving to step 3, the divisive network community detection algorithm in Section III.D will be computed to identify *fault-focused* communities that target single fault each by taking into account the statements' edge-betweenness distances as discussed in Section III.D.1. Moreover, Figure 4 show the calculation of shortest path betweenness of all edges from all the program statements in the program network N of the program in Table 1. The figure also shows the process and results of calculating the breadth-first search and the process of calculating the betweenness scores of all edges from all the seven executable statements that the network contains. As stated earlier in Section III.D and III.D.1, all these calculations will be repeated from each and every statement in N. For clarification, looking at Figure 4, the program network is divided into seven sub-networks, each network shows the betweenness score calculation for each edge from each statement in N. Therefore, the betweenness scores of all edges from all the statements in the respective networks will be added up and divided by 2 to get the final edge-betweenness score of an edge in N. From Figure 4, the betweenness scores of the edge from statement m_3 to m_4 in all calculations of each statement in N can be added up as $2.16 + 2.16 + 2.16 + 3.33 + 1.5 + 1.5 + 0.83 = 13.64$. Therefore, the total score will be divided by 2 to get the final edge-betweenness score as (6.82) of that edge. The same is applied to all the remaining edges.

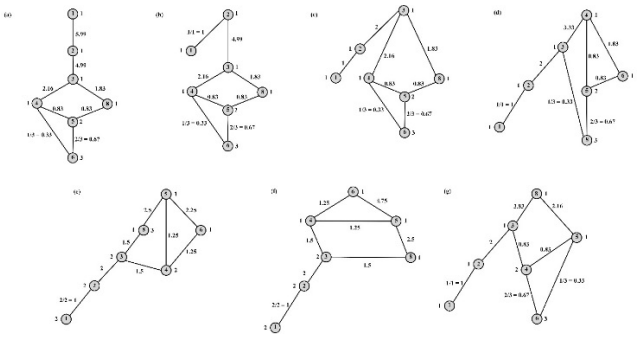

FIGURE 4. Calculation of shortest-path betweenness.

Table 2 highlights the final edge-betweenness score for all edges in the Figure 4 by adding up the betweenness score of individual edges from all statements in N and dividing the total score by 2. We identified that edge of m_2 to m_3 has the highest edge-betweenness score as $4.99+4.99+2+2+2+2+2 = 19.98$. Hence, the final edge-betweenness score of the edge will be $(19.98/2 = 9.99)$. Therefore, the edge will be removed to create two *fault-focused* communities which are $C_1 = \{m_1 \text{ and } m_2\}$ and $C_2 = \{m_3, m_4, m_5, m_6, \text{and } m_8\}$.

Next, based on our community weighting and selected process in step 4, the second community C_2 will be ranked at the top of the community ranking list because it has the highest number of statements. Therefore, each of these communities will contains single fault where community one (C_1) has faulty statement m_2 and community two (C_2) has faulty statement m_5. Lastly, the fault localization technique based on complex network theory in [16] will be used for localizing the faults in each *fault-focused* community.

TABLE 3. Experimental subject programs.

Program	Description	LOC	Faulty versions	Test cases
tcas	Altitude separation	173	41	1608
replace	Pattern replacement	564	32	5542
gzip	Data compression	6573	5	211
sed	Textual manipulator	12062	7	360
flex	Lexical analyzer	13,892	22	525
grep	Pattern searcher	12,653	7	470
make	Code manager	20,014	29	793
ant	Files builder	75,333	23	871

IV. EXPERIMENTAL SETUP

In this section, we highlight the subject programs and data collection process for our experimental process in Section IV.A. The evaluation metrics utilized are also presented in Section IV.B. Additionally, the approaches used for cross-comparison are introduced in Section IV.C.

A. SUBJECT PROGRAMS AND DATA COLLECTION

In our experiments, eight subject programs are used to evaluate the effectiveness of the proposed approach, which are *tcas, replace, gzip, sed, flex, grep, make,* and *ant* as highlighted in Table 3. These programs were also used in previous studies for fault localization [1], [38]–[41]. The programs are written in C (*tcas*, *replace*, *gzip*, *sed*, *flex*, *grep*, and *make*), and Java (*ant*) languages. All the programs were downloaded from the software infrastructure repository (SIR) site (https://sir.csc.ncsu.edu/content/sir.php) [42]. In addition to the existing faulty versions of the programs in Table 3, more faulty versions were created using mutation-based fault injection technique [1], [29], [43]. Existing studies have shown that mutation-based faults can be useful to represent real faults and provide reliable results in program debugging experiments [44], [45].

In this study, more faults are generated using two classes of mutant which are, 1). Arithmetic replacement, increment and decrement of data variables or assignment operator by another operator from the same class, and rational/logical error. 2). Decision negation in an *if* or *while* statement. Therefore, multiple-fault versions were generated with 2, 3, 4, and 5 faults for each program. This method of multiple faults generation has been utilized by various studies [1], [6], [7], [22], [46]. Therefore, 25 distinct faulty versions with 2, 3, 4, and 5 faults for the programs were created. Overall, 540 multiple-fault versions were used for this experiment.

All the executions were done on a PC with 2.13 GH_z Intel Core 2 Duo CPU and 8 GB physical memory. To compile the programs and obtain the code coverage information for each of the test cases, we used GCC compiler for the former and Gcov for the latter. Test execution output for each faulty version was compared with the output of its corresponding fault-free version provided by SIR [42]. However, if the output of the faulty version differs from its corresponding fault-free version, then the test case is labeled as a failed test. It will be labeled as passed test otherwise. The approach as well as the fault localization process has been implemented and run in the Cystoscope platform [16].

B. EVALUATION METRIC

The effectiveness of a software fault localization technique is normally evaluated by the percentage of statements that need to be examined (or not examined) to find the fault. This means that a developer will examine all the program executable statements from top to bottom according to the ranking list generated by the fault localization technique. However, to evaluate the localization of multiple faults simultaneously using the parallel debugging approach, the following metrics are used in this study.

1) AVERAGE NUMBER OF STATEMENTS TO BE EXAMINED

Using this metric, the average number of statements that developers need to examine to find all the faults in a subject program with multiple faults will be computed [47]. Suppose we have a program with n faulty versions where $X(i)$ and $Y(i)$ are the numbers of statements that need to be examined to locate all the faults in the i th multiple-faulty version by two debugging approaches X and Y, respectively. Hence, approach X is more effective than approach Y if approach X requires a developer to examine less amount of statements than approach Y to find all faults in the faulty versions as shown in Equation 2.

$$\frac{\sum_{i=1}^{n} X(i)}{n} < \frac{\sum_{i=1}^{n} Y(i)}{n} \tag{2}$$

2) TOTAL DEVELOPER EXPENSE (TDE)

To evaluate the localization of multiple faults using parallel debugging approach, we use a metric named EXAM Score. EXAM Score is defined as the percentage of program statements a developer must examine to find a fault. This metric helps in knowing the effort a single developer would spend on finding a single program fault. This metric is computed by the following equation:

$$\text{EXAMScore} = \frac{\text{rank of fault}}{\textit{Number of executable statements}} \times 100\% \tag{3}$$

Therefore, for each *fault-focused* community given to a developer, the effort a developer spent in finding the fault can be measured. The original EXAM Score as presented in Equation 3 is extended to a metric named total developer expense (TDE) to evaluate the localization of multiple faults using the parallel debugging approach [5]. Therefore, the effort a developer spent in finding a fault for each *fault-focused* community can be measured. Hence, to calculate the TDE to find all faults in a program using parallel debugging, Equation 4 will be utilized.

$$TDE = \sum_{i=1}^{p} \sum_{j=1}^{q} \text{EXAM Score}(i, j) \tag{4}$$

where p be the number of debugging iterations, q be the number of *fault-focused* communities (clusters) generated in each debugging iteration, and EXAM Score (i, j) is the percentage of program statements that is needed to be examined to locate the faults for jth *fault-focused* community in ith debugging iteration. For example, suppose we have a program version with four faults and 200 program statements. Let's say that the proposed approach needs two debugging iterations to locate all the faults. For the first debugging iteration, two *fault-focused* communities are generated for developers to check for the faults. For the first cluster, 5 statements need to be examined to locate the fault (which will be $5/200{*}100 = 2.5$), while 12 statements have to be examined in the second cluster to find another fault (which will be $12/200{*}100 = 6$). In the second debugging iteration, two more *fault-focused* communities are generated. For the first cluster, the developer needs to examine 6 statements to locate the faults (which will be $6/200{*}100 = 3$), while 7 statements have to be examined in the second cluster (which will be $7/200{*}100 = 3.5$). In totality, the TDE score to locate all the five faults will be $2.5 + 6 + 3 + 3.5 = 15$. For a fault localization technique utilizing the parallel debugging approach on multiple-fault subject programs, its effectiveness can be computed using TDE. If technique X has a lesser TDE score than technique Y, then technique X is considered more effective than technique Y.

3) WILCOXON SIGNED-RANK TEST

Wilcoxon signed-rank test which is also known as Mann-Whitney U test is an alternative option to other existing hypothesis tests such as z-test and paired student's t-test particularly when a normal distribution of a given population cannot be assumed [47], [48]. Wilcoxon signed-rank test is also utilized to give a comparison with a solid statistical basis between different techniques in terms of effectiveness. After computing the number of statements that a developer needs to examine on all approaches, an evaluation will be conducted on the one-tailed alternative hypothesis that the baseline approaches used for cross-comparison require the examination of an equal or greater number of statements than the proposed approach.

Hence, the null hypothesis, in this case, specifies that the baseline approaches require to examine fewer statements than the proposed approach. The null hypothesis is stated as follows:

H_0: The number of statements examined by the baseline approaches to locate all faults in a multiple-fault program $\leq$ the number of statements examined by the proposed approach.

Therefore, if H_0 is rejected, the alternative hypothesis is accepted. The alternative hypothesis implies that the proposed approach will require the examination of fewer statements than the baseline approaches which indicates that the proposed approach is more effective.

4) EFFICIENCY

Measuring the efficiency of a given parallel debugging approach particularly in terms of the number of debugging iterations a given approach has to utilize to neutralize all faults is very important. In this study, we evaluate the approaches based on the average number of debugging iterations that are needed to locate all the faults in a given faulty program [1]. Therefore, for two debugging approaches X and Y, if X required fewer debugging iterations to locate all the faults than Y, then one can say that X debugging approach is more efficient than Y debugging approach.

C. APPROACHES FOR CROSS-COMPARISON

We compare the proposed approach with two other parallel debugging approaches. The first approach uses the same parallel debugging process as used by Jones *et al.* [5]. Henceforth, we refer to the first approach as P1. P1 applies a k-means clustering algorithm to cluster failed tests executions, it then uses the Euclidian distance metric to measure the distance between failed tests as use in [22]. Additionally, P1 cluster failed tests based on their execution profile similarities which have been highlighted to be problematic [9].

On the other hand, the second approach used for cross-comparison is MSeer [1]. MSeer uses an improved k-medoids clustering algorithm to perform tests clustering and a revised Kendall tau distance to measure the distance between failed tests. Both approaches use the test representations discussed in Section III.B, while for the proposed approach, we use the representation in Section III.C. For P1, suspiciousness rankings are generated using Ochiai similarity coefficient-based technique, a fault localization technique that has shown to be very effective [49], while MSeer uses Crosstab fault localization technique [50]. Some of the major differences between the proposed approach and the approaches used for comparison are:

- P1 uses a k-means clustering algorithm to isolate tests based on their causative faults and MSeer uses an advanced approach that simultaneously estimates the number of clusters. MSeer uses an improved k-medoids clustering algorithm for failed tests clustering. In this paper, we proposed the use of a divisive network community clustering algorithm to isolate statements based

TABLE 4. Average number of statements examined (best case).

		tcas	replace	gzip	sed	flex	grep	make	ant
2-fault	Our Approach	5.97	12.03	10.87	50.21	9.08	333.99	190.60	14.11
	P1	10.09	19.14	40.03	81.48	35.23	370.01	402.19	20.64
3-fault	Our Approach	11.04	20.18	25.66	101.18	30.18	450.39	612.77	29.40
	P1	22.07	24.05	80.63	393.89	110.03	512.15	783.16	54.97
4-fault	Our Approach	18.11	39.06	60.00	280.38	60.07	525.33	581.11	41.16
	P1	33.33	60.04	161.04	500.10	190.00	601.60	850.58	70.60
5-fault	Our Approach	28.14	85.11	77.18	549.35	95.83	549.35	1050.89	50.67
	P1	49.04	98.91	260.03	679.74	230.06	621.34	1153.87	89.11

TABLE 5. Average number of statements examined (worst case).

		tcas	replace	gzip	sed	flex	grep	make	ant
2-fault	Our Approach	13.03	27.89	80.89	200.13	62.18	621.76	621.11	50.12
	P1	17.49	46.41	194.33	292.09	161.08	1310.01	830.74	110.64
3-fault	Our Approach	22.33	38.03	150.14	450.03	101.04	888.89	1090.16	87.78
	P1	28.49	55.19	254.19	601.75	243.63	1901.77	1190.58	204.18
4-fault	Our Approach	40.99	75.99	231.05	701.14	150.23	1340.70	1120.56	130.17
	P1	42.89	100.89	499.01	889.05	294.00	2300.03	1260.83	383.19
5-fault	Our Approach	46.11	120.83	310.11	1125.55	186.99	1599.14	1809.22	159.49
	P1	56.02	155.97	570.35	1501.08	322.15	2609.10	2203.14	503.77

on their causative faults into separate communities (Section III.D).

- P1 uses Euclidian distance metric to measuring failed tests distance, while MSeer uses a revised Kendall tau distance metric. The proposed approach used an edge-betweenness distance to measure statement-to-statement distance.

V. RESULTS AND DISCUSSION

In this section, the results and discussion are presented. The results of our cross-comparison with other parallel debugging approaches (as highlighted in Section IV.C) are also detailed and discussed. The evaluation is based on the average number of statements examined, the TDE score, Wilcoxon Signed-rank test, and Efficiency.

A. CROSS-COMPARISON WITH P1

For all our experiments, we presume that for the best case effectiveness, the faulty statement is at the very top of the suspicious statements ranking list; and for the worst case effectiveness, the faulty statement is at the very end of the ranking list. Table 4 and Table 5 present the average number of statements that need to be examined by both the proposed approach and P1 with respect to the best and worst cases. The average number of statements examined by the approaches are based on 25 versions of a given program each containing x amount of faults ($x = 2, 3, 4$, and 5). We observed that the average number of statements examined by the proposed approach on 2-fault faulty versions of *flex* is 9.08 in the best case, and 62.18 in the worst case. On the other hand, for P1, the best case is 35.23, and the worst case is 161.08. For the

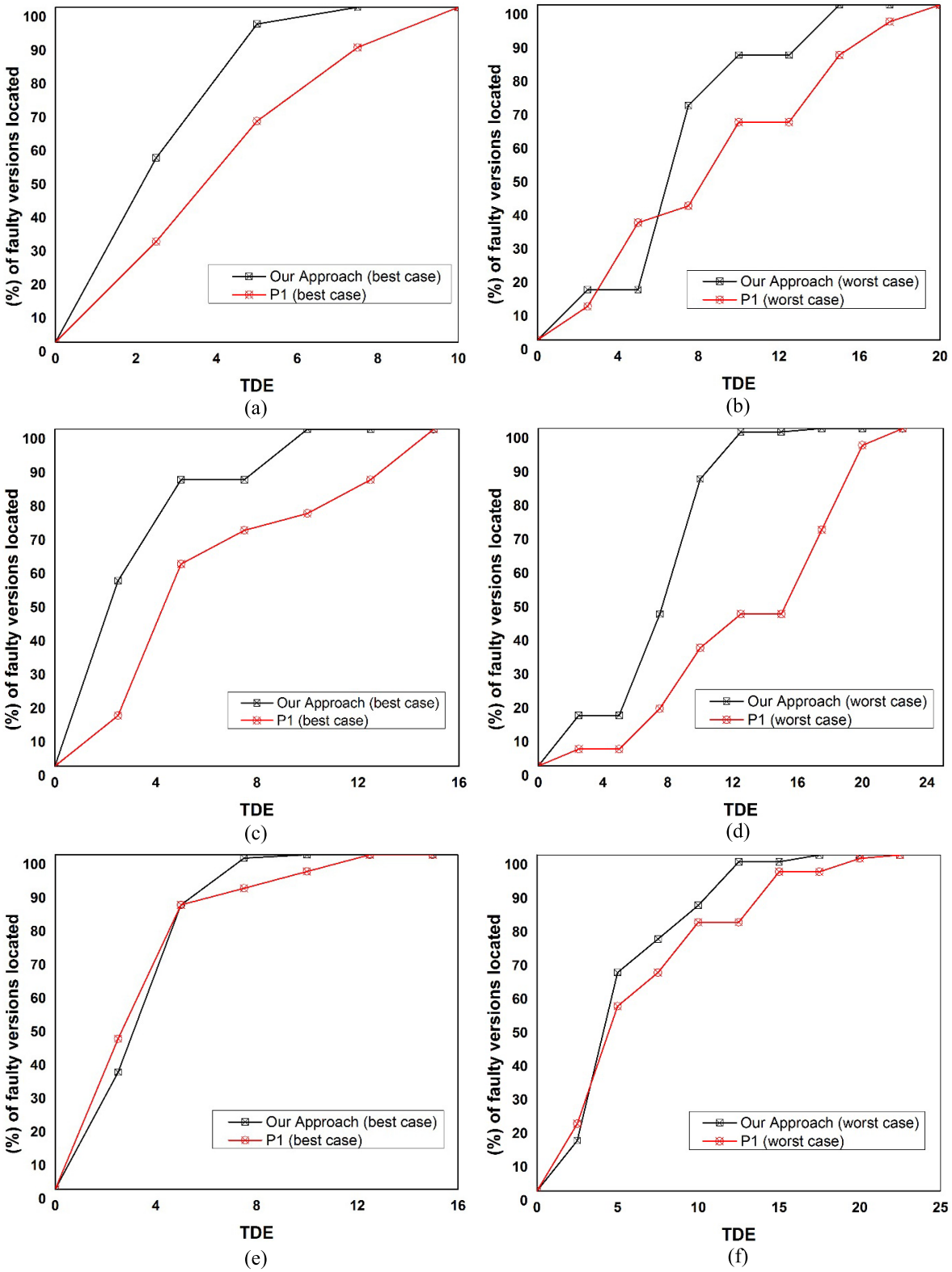

FIGURE 5. TDE score-based comparison between the proposed approach and P1 on gzip, sed, and flex (2-fault versions). (a) best case of it gzip 2-fault versions. (b) worst case of it gzip 2-fault versions. (c) best case of it sed 2-fault versions. (d) worst case of it sed 2-fault versions. (e) best case of it flex 2-fault versions. (f) worst case of it flex 2-fault versions.

3-fault faulty versions of *tcas*, the average number of statements to be examined by the proposed approach is 11.04 in the best case and 22.33 in the worst case. With respect to P1, the average number of statements to be examined in the same faulty versions (*tcas*, 3-fault) to locate all faults is 22.07 in the best case, and 28.49 in the worst case. From both tables

TABLE 6. The confidence with which it can be claimed that the proposed approach is more effective than P1 (best case).

		tcas	replace	gzip	sed	flex	grep	make	ant
2-fault	P1	75.73%	85.94%	96.58%	96.81%	96.18%	97.23%	99.53%	84.69%
3-fault	P1	90.94%	74.17%	98.19%	99.66%	98.75%	98.39%	99.41%	96.09%
4-fault	P1	93.43%	95.24%	99.02%	99.55%	99.24%	98.69%	99.63%	96.61%
5-fault	P1	95.22%	92.76%	99.46%	99.24%	99.26%	98.62%	99.03%	97.40%

TABLE 7. The confidence with which it can be claimed that the proposed approach is more effective than P1 (worst case).

		tcas	replace	gzip	sed	flex	grep	make	ant
2-fault	P1	77.58%	94.61%	99.12%	98.92%	98.99%	99.86%	99.52%	98.35%
3-fault	P1	83.77%	94.18%	99.04%	99.35%	99.30%	99.91%	99.01%	99.14%
4-fault	P1	47.37%	95.99%	99.63%	99.47%	99.31%	99.90%	99.29%	99.61%
5-fault	P1	89.91%	97.16%	99.62%	99.74%	99.27%	99.91%	99.75%	99.71%

(Table 4 and Table 5), we observed that, regardless of whether the best case or worst case considered, the proposed approach is always the most effective in comparison with P1. The proposed approach has shown to be generally more effective than P1, however, it is not surprising due to P1 obvious limitations where it cluster failed tests execution based on their execution profile similarities. With tests representation and the distance metric that the approach utilized (P1), less effective localization inferencing is expected. But it is worth re-emphasizing that our approach is by far the most effective in comparison with P1.

We now present the evaluation of our approach with respect to TDE score. In Figure 5, the 2-fault versions of *gzip*, *sed*, and *flex* in best and worst cases are presented. The *y*-axis represents the percentage (%) of faulty versions located by a developer while the *x*-axis represents the percentage of code (total expense) examined. Looking at part (a) and (b) of Figure 5, we find out that on the *gzip* program, by examining less than 7.5% of the program code, the proposed approach can locate all the faulty versions in the best case, and 70% in the worst case. In contrast, by examining the same amount of code, P1 can only locate 88% of faults in the best case, and 40% in the worst case. In part (c) and (d), the effectiveness score of *sed* 2-fault faulty versions are presented. The curves show that by examining less than 10% of the program code, the developer can locate all the faulty versions in the best case with the proposed approach, while P1 can locate 75% of the faulty versions. In the worst case, by examining the same amount of code, 85% of the faulty versions can be located using the proposed approach, and 35% with P1.

Consistently, looking at part (e) and (f) of Figure 5 (*flex* program), the proposed approach is still the most effective. However, it is worth highlighting that in some cases, the TDE score differences between the two techniques is not much. For example, with respect to *flex* program part (e) in the best case, the TDE score differences is 2.5% whereby using the proposed approach all the faulty versions can be located by examining less than 10% of the code and using P1, all the faulty versions can be located by examining 12.5% of the program code. Looking at Figure 5, the story is the same as in Table 4 and Table 5 where the proposed approach consistently surpasses P1 in locating faults effectively. For the rest of the program faulty versions (3-fault, 4-fault, 5-fault), even though their curves are not visually represented, the conclusion applies to the versions as well.

Having looked at the results in terms of the average number of statements examined and TDE score, Table 6 and Table 7 give the effectiveness comparisons of the best and worst cases using the Wilcoxon signed-rank test. The tables give the confidence of which the alternative hypothesis can be accepted (that the proposed approach requires the examination of fewer statements than the compared baseline approach). For example, it can be said with 99.46% (best case) and 99.62% (worst case) confidence that the proposed approach is more effective than P1 on 5-fault versions of *gzip*. For *gzip, sed, flex, make,* and *grep* programs, the confidence to accept the alternative hypothesis is higher than 96% in both best and worst cases across all faulty versions (2-fault, 3-fault, 4-fault, and 5-fault).

However, for *tcas* and replace programs across all the faulty versions on the best and worst cases scenarios, the alternative hypothesis is accepted with a confidence level of higher than 90% in most cases with a few exceptions having confidence level as low as 47.37% (in the worst case of *tcas* program 4-fault versions). The scenarios with low confidence level are where the difference between the proposed approach and P1 in terms of statements examined to locate all the faulty versions by each approach is very small. In totality, the result shows that the proposed approach performs better than P1 in both best and worst cases.

B. CROSS-COMPARISON WITH MSeer AND P1

With respect to 25 versions of five programs (*gzip, flex, grep, make,* and *ant*) containing x amount of faults ($x = 3$), Table 8 gives the average number of statements examined by the proposed approach, MSeer, and P1 to produce a

TABLE 8. Average number of statements examined using the proposed approach, MSeer, and P1 (3-fault versions).

	Our Approach		MSeer		P1	
	Best case	Worst case	Best case	Worst case	Best case	Worst case
gzip	25.66	150.14	41.77	174.57	80.63	254.19
flex	30.18	101.40	23.30	111.40	110.03	243.63
grep	450.39	888.89	492.93	948.47	512.15	1901.77
make	612.77	1090.16	640.70	1130.47	783.16	1190.58
ant	29.40	87.78	35.13	111.87	54.97	204.18

failure-free program. For instance, the average number of statements examined by the proposed approach on *grep* is 450.39 in the best case, and 888.89 in the worst case. On the other hand, MSeer is 492.93 in the best case, 948.47 in the worst case. For P1, the best case is 512.15, and the worst case is 1901.77. Hence, with respect to the result on the 3-fault faulty versions of these programs, we made a significant observation. We observed that in all but the best case of *flex,* the proposed approach is more effective than MSeer in all programs faulty versions.

In all, the proposed approach is more effective because it examines fewer statements than MSeer and P1 in most cases. The difference between the proposed approach and MSeer is highly significant in some cases, for example, in the best case of *gzip*. In all cases, P1 is less effective, and the difference in comparison with other approaches is very significant. For instance, we observe that the average number of statements to be examined in the best case of *flex* is 30.18 for the proposed approach, and 110.03 for P1 in the best case. This clearly indicates that the proposed approach is more effective than P1. Another significant point worth noting is that in some cases, the effectiveness differences between the proposed approach and MSeer is insignificant. For example, in the worst case of *flex* where the proposed approach on average examined 101.40 statements in the worst case, and MSeer examined 111.40 in the worst case. Another significant observation is that in some cases, MSeer performs better than the proposed approach. For example, in the best case of *flex* where the proposed approach on average examined 30.18 statements in the best case, and MSeer examined 23.30 in the best case. Therefore, by examining lesser number of statements of *flex* than the proposed approach in the best case scenario, MSeer is the most effective in the scenario.

Next, we present the result of the cross comparison between the proposed approach, MSeer, and P1 with respect to TDE score of all faulty versions. Figure 6 presents the 3-fault versions of *gzip, grep,* and *make* in the best case and worst case. For instance, in part (a) and part (b), by examining less than 10% of the program code, the proposed approach can locate 100% of the faulty versions in the best case and 60% in the worst case. MSeer (the second best approach) can only locate 92% in the best case and 42% in the worst case when examining the same program code. For P1 (the third best), by examining the same amount of program code (less than 10%), is 80% (best case), and 35% (worst case). In part (c) and (d) (*grep* 3-fault versions), the curves show that by examining less than 20% of the code, 85% of the faulty versions are located by the proposed approach in the best case and 35% in the worst case. For MSeer and P1, these percentages are 90% and 70% in the best case, and 20% and 10% in the worst case, respectively. Moreover, the curves in part (e) and (f) shows that by examining less than 10% of the code, 55% of the faulty versions are located by the proposed approach in the best case and 15% in the worst case, respectively. For MSeer and P1 is 45% and 30% in the best case, and 5% and 10% in the worst case, respectively. The results in Figure 5 and Figure 6 suggest that the proposed approach is convincingly the most effective in comparison to the best and worst cases of the comparative approaches (MSeer and P1).

Based on the results in Table 8 and Figure 6, the proposed approach emerges to be the most effective in comparison to MSeer and P1. Hence, with respect to the third evaluation metric, Table 9 gives the effectiveness comparison of the proposed approach in terms of the Wilcoxon signed-rank test on the 3-fault versions of *gzip, flex, grep, make,*and *ant* programs. If the null hypothesis is accepted, the confidence level will be given as 00.00%. The cell with a black background in Table 9 is the cell where the null hypothesis is accepted.

For instance, for the best and worst cases of P1 on all programs, the confidence level to accept the alternative hypothesis is higher than 98%. Furthermore, for MSeer, the confidence to accept the alternative hypothesis is greater than 90% in most cases. However, for the best case of *flex* on MSeer, the null hypothesis is accepted, meaning that MSeer examined a fewer number of statements than the proposed approach which makes it more effective in this scenario. Therefore, because MSeer examined fewer statements than the proposed approach in the best case of *flex*, the null

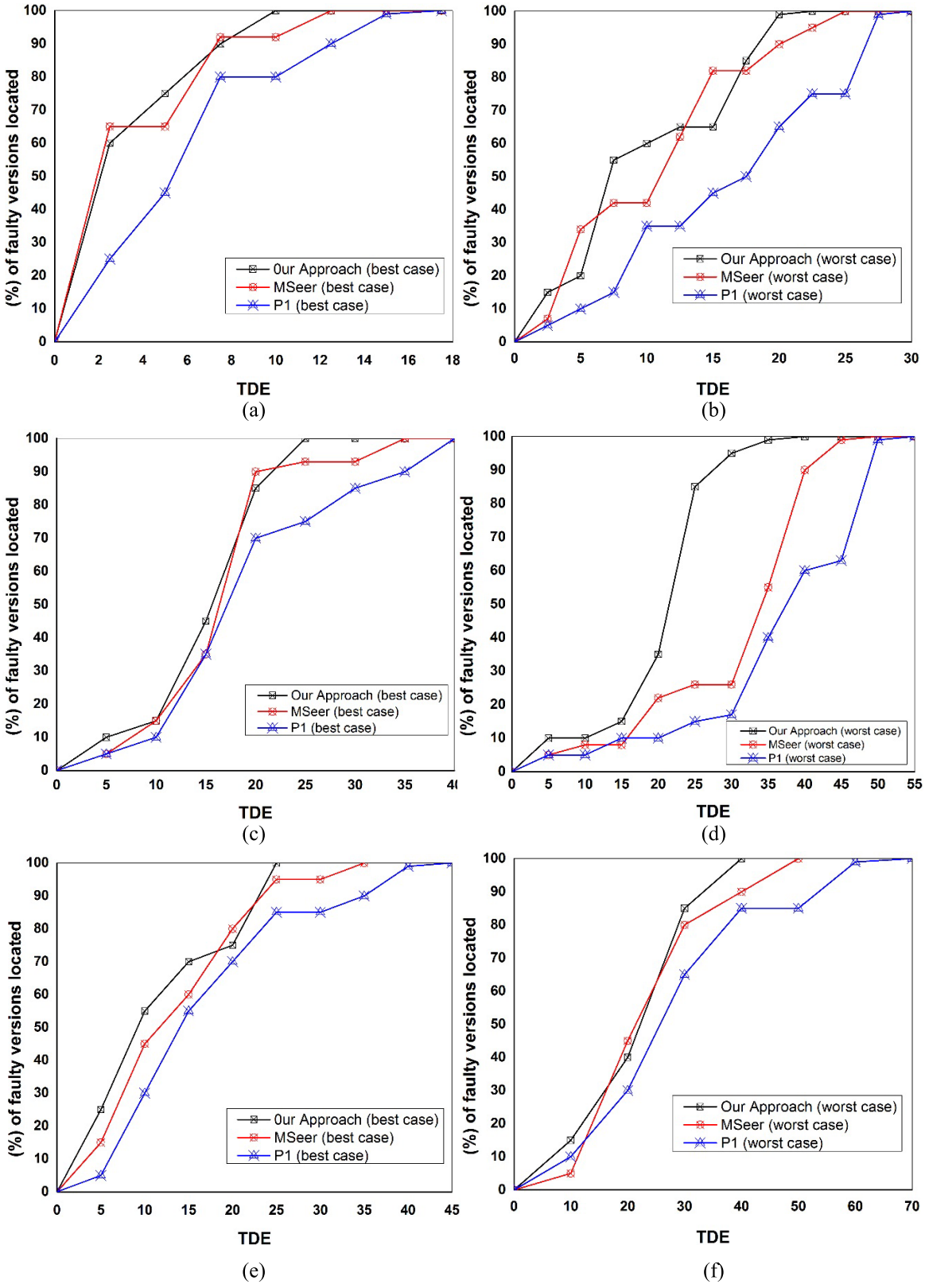

FIGURE 6. TDE score-based comparison between the proposed approach with P1 and MSeer on gzip, grep, and make (3-fault versions). (a) best case of it gzip 3-fault versions. (b) worst case of it gzip 3-fault versions. (c) best case of it grep 3-fault versions. (d) worst case of it grep 3-fault versions. (e) best case of it make 3-fault versions. (f) worst case of it make 3-fault versions.

TABLE 9. The Confidence with which it can be claimed that the proposed approach is more effective than MSeer and P1 (best & worst case) (3-fault versions).

	MSeer		P1	
	Best case	Worst case	Best case	Worst case
gzip	93.80%	95.91%	98.19%	99.04%
flex	0.00%	90.00%	98.75%	99.30%
grep	97.65%	98.33%	98.39%	99.91%
make	96.42%	97.52%	99.41%	99.01%
ant	82.55%	95.85%	96.09%	99.145

hypothesis is accepted and the confidence level is given as 00.00% in Table 9.

C. DISTANCE METRICS

Distance metrics play a critical and important role to achieve a good tests clustering result, which measures the distance between failed tests or program statements (in the proposed approach context). In this section, the three distance metrics used by the proposed approach and the remaining two approaches (MSeer and P1) were investigated. For the proposed approach, the edge-betweenness distance is used, MSeer used the revised Kendall tau distance, and P1 used the Euclidian distance metric. The effectiveness of these approaches using these distance metrics were compared. In Table 10, the average number of statements examined on 25 versions with 5-fault for *gzip, grep, flex, make,* and *ant* are highlighted. Using the proposed approach, the average number of statements examined for *flex* is 95.83 (best case) and 186.99 (worst case). For MSeer and P1, they are 104.67 and 230.06 (best cases), and 195.80 and 322.15 (worst cases), respectively. However, it was observed that in some cases, MSeer examined fewer statements. For example, in the best case of *grep* is 612.11 for the proposed approach and 598.80 for MSeer. Even though the difference is slightly small in some case between the proposed approach and MSeer, the result clearly shows that the edge-betweenness distance used by the proposed approach is more effective.

D. EFFICIENCY OF THE DEBUGGING APPROACHES BASED ON THE NUMBER OF DEBUGGING ITERATIONS

Table 11 presents the average number of debugging iterations needed by the proposed approach, MSeer, and P1 to neutralize all faults in their respective faulty versions. The result in the table is with respect to 25 versions of *gzip, flex, grep, make,* and *ant* programs containing x amount of faults (x = 2, 3, 4, and 5), respectively.

From the table (Table 11), we observed that the average number of debugging iterations required by the proposed approach is generally smaller than that required by MSeer and P1 except for few exceptions. For example, in *flex* 2 fault versions, *grep* 5 fault versions, and *ant* 5 fault versions. In these scenario, the average number of debugging iterations for MSeer is smaller in comparison to the proposed approach. Generally, the result in terms of efficiency shows that the proposed approach is more efficient in comparison with MSeer and P1, hence, more effective.

We observed that, in most cases, the proposed approach only need one debugging iteration to neutralize all faults in a given faulty version due to the nature of the approach. Therefore, in some scenarios where a developer checked 50% of the *fault-focused* communities generated for a single debugging iteration in our proposed approach without neutralizing all the know faults, the program has to be retested to the second debugging iteration to neutralize the remaining faults. This can be curtailed with the introduction of community estimation step in the network community clustering algorithm so as to be able to neutralize all the faults in a single debugging iteration.

E. RESULT SUMMARY

In totality, the following observations were made:

- Overall, based on the average number of statements examined by both the proposed approach and P1 in Table 4 and Table 5, the proposed approach is the most effective where in most cases, fewer statements were examined to locate all the faulty versions than P1. For instance, it was observed that the average number of statements examined by the proposed approach on 2-fault faulty versions of *flex* is 9.08 (best case), and 62.18 (worst case). On the other hand, for P-Ochiai, it is 35.23 (best case), and 161.08 (worst case). Additionally, the proposed approach is still the most effective in terms of TDE score. However, it is worth highlighting that in some cases, the TDE score difference between the two approaches is not much. For example, with respect to *flex* program part (e) in the best case (Figure 5), using the proposed approach, all the faulty versions can be located by examining less than 10% of the code, and using P1, all the faulty versions can be located by examining less than 12.5% of the program code.
- Furthermore, with respect to the 3-fault faulty versions of *flex*, we observed that, in all but the best case of *flex*, the proposed approach is more effective, however, MSeer on average can locate all the faulty versions by examining only 23.30 statements (best case), and 30.18 (best case) using the proposed approach. Another significant point worth noting is that in some cases, the effectiveness difference between the proposed approach and MSeer is insignificant. For example, in the worst case of *flex* where the proposed approach on average examined 101.40 statements (worst case), and MSeer examined 111.40 (worst case).
- In all program faulty versions of *gzip* and *grep* (Figure 6), the proposed approach is convincingly the most effective in the best and worst cases in comparison with MSeer and P1 approaches. For instance, in part

TABLE 10. Average number of statements examined using the proposed approach, MSeer, and P1 (5-Fault versions).

	Our Approach		MSeer		P1	
	Best case	Worst case	Best case	Worst case	Best case	Worst case
gzip	77.18	310.11	80.00	318.27	260.03	570.35
grep	612.11	1599.14	598.80	1644.77	621.34	2609.10
flex	95.83	186.99	104.67	195.80	230.06	322.15
make	1050.89	1809.22	1142.97	1959.37	1153.87	2203.14
ant	50.67	159.49	53.40	178.47	89.11	503.77

TABLE 11. Average number of debugging iterations.

		gzip	flex	grep	make	ant
2-fault	Our Approach	1.08	1.20	1.20	1.32	0.92
	MSeer	1.17	1.13	1.23	1.47	1.10
	P1	1.64	1.68	1.80	1.56	1.48
3-fault	Our Approach	1.20	1.44	1.48	1.60	1.12
	MSeer	1.43	1.53	1.53	1.83	1.27
	P1	1.52	1.80	1.96	1.88	1.52
4-fault	Our Approach	1.72	1.84	1.96	2.08	1.84
	MSeer	1.93	1.93	1.80	2.27	1.73
	P1	2.04	2.44	2.36	2.32	2.20
5-fault	Our Approach	1.96	2.04	1.92	2.20	1.88
	MSeer	2.40	2.37	2.27	2.53	2.03
	P1	2.48	2.64	2.76	2.96	2.88

(a) and part (b) of *gzip*, by examining less than 10% of the program code, the proposed approach can locate all the faulty versions in the best case and 60% (worst case). MSeer (the second best approach) can only locate 92% (best case) and 42% (worst case). For P1 (the third best), it is 80% (best case) and 35% (worst case) to locate the faulty versions. Therefore, the proposed approach performs better than MSeer and P1.

- With respect to the evaluation in terms of efficiency, we observed that the average number of debugging iterations required by the proposed approach is generally smaller than that required by MSeer and P1 (Table 11). For instance, in *flex* 3 fault versions, the average debugging iterations needed for MSeer and P1 to locate all faults are 1.53 and 1.80, respectively. However, for the proposed approach is 1.44 which makes it more efficient in this scenario. In some few exceptions such as in *flex* 2 fault versions, *grep* 5 fault versions, and *ant* 5 fault versions, the average number of debugging iterations for MSeer is smaller in comparison to the proposed approach. Generally, the result in terms of efficiency shows that the proposed approach is more efficient in comparison with MSeer and P1, hence, more effective.

VI. THREAT TO VALIDITY

There are a number of threats to validity of the proposed approach and its associated results which include but are not limited to the following.

The main threat to external validity is the generalizability of the experimental results. It is generally known that *tcas* and *replace* programs are relatively smaller in size. Even though the UNIX utility programs used (*gzip*, *sed*, *flex*, *make* and *grep*) are considerably larger programs, they are still not very large due to the current diversity of existing software (in size and complexity). Therefore, more diverse (in terms of language) larger-sized industrial programs should be further used to validate the proposed approach. Henceforth, we cannot generalize these results to other subject programs. Furthermore, mutation-based fault injection techniques were used for multiple-fault versions generations. Even though this technique was ascertained in simulating real faults behavior, yet more experiments would be needed on multiple faults subject programs containing real faults [3].

Another threat to validity may be caused by the proposed community weighting and selection mechanism. Even though the proposed community weighting and selection mechanism aids in ranking the highly suspicious communities in most cases, some communities that contain faults often reside in the lowest community rank list which may or will cause a long time to produce a failure-free program. Hence, this threat needs to be mitigated. The threat to construct validity refers to whether or not the experiment measures what it was intended to measure. Moreover, the basic threat to construct validity is the utilization of four metrics namely, the average number of statements examined, total developer expense (TDE), Wilcoxon Signed-rank test, and efficiency. However, these metrics have been used in previous studies adopting parallel debugging approach [1], [5], [6], so the threat is reasonably mitigated.

VII. CONCLUSION

In this paper, we have proposed an approach that makes used of a divisive network community clustering algorithm to isolate faults into *fault-focused* communities targeting a single fault each. Based on this algorithm, a developer is tasked to find the least similar connected statements in a faulty program modeled network and then remove the edges between them (connection between statements). Hence, communities are created by continuously removing the edges from the modeled program network based on the statements edge-betweenness distance. This process is done repeatedly until the program network is naturally divided into smaller and smaller groups composed of densely connected statements that form communities. A novel community weighting and selection mechanism was further introduced to aid in prioritizing highly important *fault-focused* communities to the available developers to debug the faults simultaneously in parallel.

The approach was evaluated on eight subject programs ranging from medium-sized to large-sized programs (*tcas, replace, gzip, sed, flex, grep, make,*and *ant*). Overall, 540 multiple-fault versions of these programs were generated with 2, 3, 4, and 5 faulty versions. The result of the experiments showed that the proposed approach outperforms two other parallel debugging approaches in terms of localization effectiveness. For future work, to help solidify and generalize our results, further work on more diverse (in terms of language) and larger-sized industrial programs with a varying number of real faults will be conducted. Hence, more work will be done to improve the divisive network community clustering algorithm by adding a community estimation step to limit the number of communities produced which will improve accuracy to reduce the time to produce a failure-free program.

APPENDIX

Additional information can be found at this link (https://bit.ly/2IcYgBm)

Acknowledgment

This work was supported by the research project FP001-2016 under the Fundamental Research Grant Scheme provided by the Ministry of Higher Education, Malaysia.

REFERENCES

[1] R. Gao and W. E. Wong, "MSeer—An advanced technique for locating multiple bugs in parallel," *IEEE Trans. Softw. Eng.*, vol. 45, no. 3, pp. 301–318, Mar. 2019.

[2] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proc. 27th Int. Conf. Softw. Eng. (ICSE)*, 2005, pp. 342–351.

[3] S. Pearson *et al.*, "Evaluating and improving fault localization," in *Proc. 39th Int. Conf. Softw. Eng.*, May 2017, pp. 609–620.

[4] N. DiGiuseppe and J. A. Jones, "Software behavior and failure clustering: An empirical study of fault causality," in *Proc. IEEE 5th Int. Conf. Softw. Test., Verification Validation*, Apr. 2012, pp. 191–200.

[5] J. A. Jones, J. F. Bowring, and M. J. Harrold, "Debugging in parallel," in *Proc. Int. Symp. Softw. Test. Anal.*, 2007, pp. 16–26.

[6] W. Högerle, F. Steimann, and M. Frenkel, "More debugging in parallel," in *Proc. IEEE 25th Int. Symp. Softw. Rel. Eng.*, Nov. 2014, pp. 133–143.

[7] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham, "Effective software fault localization using an RBF neural network," *IEEE Trans. Rel.*, vol. 61, no. 1, pp. 149–169, Mar. 2012.

[8] D. Jeffrey, N. Gupta, and R. Gupta, "Effective and efficient localization of multiple faults using value replacement," in *Proc. IEEE Int. Conf. Softw. Maintenance (ICSM)*, Sep. 2009, pp. 221–230.

[9] C. Liu, X. Zhang, and J. Han, "A systematic study of failure proximity," *IEEE Trans. Softw. Eng.*, vol. 34, no. 6, pp. 826–843, Nov. 2008.

[10] B. Liu, S. Nejati, L. Braind, and T. Bruckmann, "Localizing multiple faults in simulink models," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, Mar. 2016, pp. 141–156.

[11] X. Sun, X. Peng, B. Li, B. Li, and W. Wen, "IPSETFUL: An iterative process of selecting test cases for effective fault localization by exploring concept lattice of program spectra," *Frontiers Comput. Sci.*, vol. 10, no. 5, pp. 812–831, 2016.

[12] N. DiGiuseppe and J. A. Jones, "Concept-based failure clustering," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, pp. 1–29.

[13] Y. Wang, R. Gao, Z. Chen, W. E. Wong, and B. Luo, "WAS: A weighted attribute-based strategy for cluster test selection," *J. Syst. Softw.*, vol. 98, pp. 44–58, Dec. 2014.

[14] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proc. Nat. Acad. Sci. USA*, vol. 99, no. 12, pp. 7821–7826, Apr. 2002.

[15] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 69, Feb. 2004, Art. no. 026113.

[16] A. Zakari, S. P. Lee, and C. Y. Chong, "Simultaneous localization of software faults based on complex network theory," *IEEE Access*, vol. 6, pp. 23990–24002, 2018.
[17] L. Šubelj and M. Bajec, "Community structure of complex software systems: Analysis and applications," *Phys. A, Stat. Mech. Appl.*, vol. 390, no. 16, pp. 2968–2975, 2011.
[18] V. D. Blondel, J. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *J. Stat. Mech., Theory Exp.*, vol. 2008, no. 10, 2008, Art. no. P10008.
[19] A. Podgurski *et al.*, "Automated support for classifying software failure reports," in *Proc. 25th Int. Conf. Softw. Eng.*, May 2003, pp. 465–475.
[20] W. Dickinson, D. Leon, and A. Podgurski, "Finding failures by cluster analysis of execution profiles," in *Proc. 23rd Int. Conf. Softw. Eng.*, 2001, pp. 339–348.
[21] C. Liu and J. Han, "Failure proximity: A fault localization-based approach," in *Proc. 14th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2006, pp. 46–56.
[22] Y. Huang, J. Wu, Y. Feng, Z. Chen, and Z. Zhao, "An empirical study on clustering for isolating bugs in fault localization," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Nov. 2013, pp. 138–143.
[23] M. Srivastav, Y. Singh, C. Gupta, and D. S. Chauhan, "Complexity estimation approach for debugging in parallel," in *Proc. 2nd Int. Conf. Comput. Res. Develop.*, May 2010, pp. 223–227.
[24] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: Simultaneous identification of multiple bugs," in *Proc. 23rd Int. Conf. Mach. Learn.*, 2006, pp. 1105–1112.
[25] L. C. Briand, Y. Labiche, and X. Liu, "Using machine learning to support debugging with tarantula," in *Proc. 18th IEEE Int. Symp. Softw. Rel. (ISSRE)*, Nov. 2007, pp. 137–146.
[26] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 15–26, 2005.
[27] S. Parsa, M. Vahidi-Asl, and M. Asadi-Aghbolaghi, "Hierarchy-debug: A scalable statistical technique for fault localization," *Softw. Qual. J.*, vol. 22, no. 3, pp. 427–466, 2014.
[28] Z. Wei and B. Han, "Multiple-bug oriented fault localization: A parameter-based combination approach," in *Proc. IEEE 7th Int. Conf. Softw. Secur. Rel.-Companion (SERE-C)*, Jun. 2013, pp. 125–130.
[29] N. DiGiuseppe and J. A. Jones, "Fault density, fault types, and spectra-based fault localization," *Empirical Softw. Eng.*, vol. 20, no. 4, pp. 928–967, 2015.
[30] X. Xue and A. S. Namin, "How significant is the effect of fault interactions on coverage-based fault localizations?" in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, Oct. 2013, pp. 113–122.
[31] J. Lee, J. Kim, and E. Lee, "Enhanced fault localization by weighting test cases with multiple faults," in *Proc. Int. Conf. Softw. Eng. Res. Pract. (SERP)*, 2016, pp. 1–116.
[32] J. Scott, *Social Network Analysis: A Handbook*, 2nd ed. London, U.K.: Sage, 2000.
[33] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, no. 1, pp. 35–41, 1977.
[34] M. E. Newman, "Scientific collaboration networks. II. Shortest paths, weighted networks, and centrality," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 64, no. 1, 2001, Art. no. 016132.
[35] C. E. Leiserson, R. C. Rivest, C. Stein, and T. H. Cormen, *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press, 2001.
[36] S. N. Dorogovtsev and J. F. F. Mendes, *Evolution of Networks: From Biological Nets to the Internet and WWW*. Oxford, U.K.: Oxford Univ. Press, 2003.
[37] L.-Z. Zhu, B.-B. Yin, and K.-Y. Cai, "Software fault localization based on centrality measures," in *Proc. IEEE 35th Annu. Comput. Softw. Appl. Conf. Workshops (COMPSACW)*, Jul. 2011, pp. 37–42.
[38] N. DiGiuseppe and J. A. Jones, "On the influence of multiple faults on coverage-based fault localization," in *Proc. Int. Symp. Softw. Test. Anal.*, 2011, pp. 210–220.
[39] Z. You, Z. Qin, and Z. Zheng, "Statistical fault localization using execution sequence," in *Proc. Int. Conf. Mach. Learn. Cybern. (ICMLC)*, Jul. 2012, pp. 889–905.
[40] T.-D. B. Le, F. Thung, and D. Lo, "Theory and practice, do they match? A case with spectrum-based fault localization," in *Proc. 29th IEEE Int. Conf. Softw. Maintenance (ICSM)*, Sep. 2013, pp. 380–383.
[41] X. Xia, L. Gong, T. Le, D. Lo, L. Jiang, and H. Zhang, "Diversity maximization speedup for localizing faults in single-fault and multi-fault programs," *Autom. Softw. Eng.*, vol. 23, no. 1, pp. 43–75, 2016.
[42] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Softw. Eng.*, vol. 10, no. 4, pp. 405–435, 2005.
[43] Y. Zheng, Z. Wang, X. Fan, X. Chen, and Z. Yang, "Localizing multiple software faults based on evolution algorithm," *J. Syst. Softw.*, vol. 139, pp. 107–123, May 2018.
[44] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. 27th Int. Conf. Softw. Eng.*, 2005, pp. 402–411.
[45] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, Aug. 2006.
[46] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "Simultaneous debugging of software faults," *J. Syst. Softw.*, vol. 84, no. 4, pp. 573–586, 2011.
[47] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The DStar Method for Effective Software Fault Localization," *IEEE Trans. Rel.*, vol. 63, no. 1, pp. 290–308, Mar. 2014.
[48] R. L. Ott and M. T. Longnecker, *An Introduction to Statistical Methods and Data Analysis*. Toronto, ON, Canada: Nelson Education, 2015.
[49] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Proc. Test., Acad. Ind. Conf. Pract. Res. Techn. (MUTATION)*, Sep. 2007, pp. 89–98.
[50] W. E. Wong, V. Debroy, and D. Xu, "Towards better fault localization: A crosstab-based statistical approach," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 42, no. 3, pp. 378–396, May 2012.

ABUBAKAR ZAKARI received the master's degree in computer networks from Middlesex University, London, in 2014. He is currently pursuing the Ph.D. degree with the Software Engineering Department, University of Malaya, Malaysia. He is currently a Lecturer with the Kano University of Science and Technology, Wudil, Nigeria. His current research interests include software testing, software fault localization, dynamic software update, and graph theory.

SAI PECK LEE received the master's degree from the University of Malaya, the Diplôme d'Études Approfondies (D.E.A.) degree from the Université Pierre et Marie Curie (Paris VI), and the Ph.D. degree from Université Panthéon-Sorbonne (Paris I), all in computer science. She is currently a Professor with the Faculty of Computer Science and Information Technology, University of Malaya. She has published an academic book, a few book chapters, and more than 100 papers in various local and international conferences and journals. Her current research interests include software engineering include object-oriented techniques and CASE tools, software reuse, software fault localization, requirements engineering, application and persistence frameworks, software traceability, and clustering. She has been an active Member in the reviewer committees and programme committees of several local and international conferences. She is currently in several Experts Referee Panels, both locally and internationally.

IBRAHIM ABAKER TARGIO HASHEM received the master's degree in computer science from the University of Wales, Newport, and the Doctor of Philosophy (Ph.D.) degree in computer science from the University of Malaya. He received professional certificates from CISCO (CCNP, CCNA, and CCNA Security) and APMG Group (PRINCE2 Foundation, ITIL v3 Foundation, and OBASHI Foundation). He is currently a Lecturer with the Department of Computing and IT, Taylor's University, Selangor, Malaysia. He has published a number of research articles in refereed international journals and magazines. His numerous research articles are very famous and among the most downloaded in top journals. His research interests include big data, cloud computing, distributed computing, software debagging, and machine learning. He is an active Member of the Mobile Cloud Computing Center, Malaysia.