Using Weighted Attributes to Improve Cluster Test Selection

Yabin Wang, Zhenyu Chen, Yang Feng, Bin Luo, Yijie Yang
State Key Laboratory for Novel Software Technology, Nanjing University
Software Institute, Nanjing University
Email: zychen@software.nju.edu.cn

Abstract—Cluster Test Selection (CTS) is widely-used in observation-based testing and regression testing. CTS selects a small subset of tests to fulfill the original testing task by clustering execution profiles. In observation-based testing, CTS saves human efforts for result inspection by reducing the number of tests and finding failures as many as possible. This paper proposes a novel strategy, namely WAS (Weighted Attribute based Strategy), to improve CTS. WAS is inspired by the idea of fault localization, which ranks the program entities to find possible faulty entities. The ranking of entity is considered as a weight of attribute in WAS. And then it helps build up a more suitable distance space for CTS. As a result, a more accurate clustering is obtained to improve CTS. We conducted an experiment on three open-source programs: flex, grep and gzip. The experimental results show that WAS can outperform all existing CTS techniques in observation-based testing.

Index Terms—Iteratively clustering, Attribute weight, Fault localization, Test selection, Jaccard

I. INTRODUCTION

In the past decades, people have made huge achievements in software testing. A lot of automatic techniques have been applied in software testing, including test data generation, test data execution, test report, etc. However, test results could not be verified automatically if test oracles are not available or even test oracles are available. In these cases, testers have to inspect test outputs manually to verify them [8][9][10][11][12]. A lot of human efforts need to be taken if the quantity of tests is large. Hence, test selection techniques are introduced to reduce the number of tests, as well as human costs.

Some researchers introduced clustering technique, called cluster filtering [8][9], into test selection. Their motivation is that failures caused by a same bug often have same behaviors [16]. The clustering method will produce clusters, such that items with high similarity are in same clusters [12]. The failing tests are expected to be grouped into some clusters, which tests in a same cluster can detect the same bugs. In order to reveal these bugs, we could take some strategies to sample some tests from each cluster [19]. Some researchers [21] [10] extend cluster filtering techniques for regression testing. All of these techniques are called Cluster Test Selection (CTS) [6].

The clustering result is a key base for CTS. Most of existing CTS techniques consider that each attribute in the profiles has same effects on bugs. That is, the weight of each attribute is equal in clustering. This simple treatment may result in some mixed clusters, which could not represent bugs in software. These mixed clusters raise challenges to select failing testing for observation-based testing, even for the sophisticated sampling strategy [19]. A main reason of the mixed clusters is that the distance space for clustering does not represent the behaviors of software. Hence, Chen et al. introduce semi-supervised learning to re-shape the distance space, such that the effectiveness of CTS are improved [4].

A common sense is that the attributes should have different weights for for clustering, because the attributes locate a bug are more important for the corresponding cluster. A natural idea is to assign high weights for attributes related to bugs and low weights for other attributes. A challenge of this task is how to assign weights for attributes suitable for software behaviors. We propose a novel approach inspired by the intuitions of existing fault localization techniques. There are many fault localization techniques to rank statements with scores in a program [17]. The statements with the highest scores are considered the most likely to be buggy. Hence, it is natural to use the scores as the weights of attributes.

This paper proposes a novel strategy, namely WAS (Weighted Attribute based Strategy), to improve CTS. WAS calculates the possibility of every tests to be failure. The tests with the maximum failure possibility in each cluster are selected iteratively. Among all of the ranking metrics described above, WAS uses Jaccard ranking metric to calculate weights for attributes in profiles. Clustering with weighted attributes could produce more accurate result, such that CTS is improved. We conducted an experiment on three open-source programs: flex, grep and gzip. We compared WAS with other CTS techniques. The experimental results show that WAS can outperform all existing CTS techniques in observation-based testing.

In this paper, we make the following contributions:

(1) To the best of our knowledge, it is the first time to use weighted attributes of profiles to improve clustering, as well as CTS, in observation-based testing.
(2) We use iterative clustering to improve CTS along with the weights of attributes are generated.
(3) We conducted an experiment with three open-source programs to compare all existing CTS techniques.

The rest of this paper is structured as follows. In section II we discuss related work on this paper, section III presents our approach WAS in detail. In section IV the experiment
to investigate CTS techniques are described. In section V we make conclusions and future work.

II. RELATED WORK

As we described in the introduction part, in an ideal case, only one test needs to be selected randomly from every cluster for CTS. This sampling strategy is called one-per-cluster sampling [5]. Unfortunately, the ideal case is hard or even impossible to reach in practice. A cluster usually contains passing and failing tests and the failing tests in a cluster are often caused by different bugs. On the other hand, one failing test is not sufficient for debugging or other related tasks. We need more failures to debug and maintain software. Therefore, it is better to reveal failures as many as possible in some cases. The n-per-cluster sampling [5] is proposed for CTS. The difference between these two strategies is that the n-per-cluster sampling randomly selects n tests from each cluster. It finds more failures than one-per-cluster sampling by selecting more tests. When using these sampling strategies, testers have no information to guide their test selection. Both of the two strategies are completely random.

CTS techniques group similar executions into same clusters. William et al. [8], [9] propose cluster filtering techniques and conclude that it is more effective than random test selection. Furthermore, they proposed adaptive sampling strategy to select all tests in a cluster contains an inspected failed test. They compared n-per-cluster sampling, one-per-cluster sampling and adaptive sampling techniques. Their studies show that adaptive sampling can reveal more failures than the other two sampling techniques for selecting the same number of tests [8].

All these sampling strategies use the observation result of selected tests to guide test selection without collecting other information from program. They do not use software execution spectra analysis technique to improve clustering. On the other hand, there are many studies on fault localization using execution spectra information. Spectra information has also been used in software diagnosis by many researchers. The Tarantula system [15] is the first system to use spectra information for ranking statements in software diagnosis. The AMPLE system [7] was used to diagnose object-oriented program. Jaccard was used in the web diagnosis system [3]. Ochiai, Jaccard, Ample and Tarantula metrics all have been used the Siemens test suite to evaluate their effectiveness for diagnosing software [1]. In [18] several metrics were proposed and evaluated by Wong et al.

Yan et al. firstly use spectra information, which is inspired by the the intuitions of fault localization to improve sampling strategies [19]. Their dynamic sampling strategy ESBS could outperform the adaptive sampling strategy. However, the dynamic sampling strategy should also be based on a well-established a clustering result. Chen et al. firstly re-shape the distance space of profiles, such that the clustering result is more consistent with software behaviors. They introduce a well-known semi-supervised learning SSDR to improve clustering, as well as CTS. They use novel machine learning techniques to improve clustering and we use program spectra information to improve clustering.

There are other techniques to reduce the human effort in software testing: test suite reduction and test prioritization. Some of these studies are also related to fault localization. Yu et al. [20] compared different test reduction techniques and observed their influence on fault localization. Jiang et al. [14] did experiment to research the influence of test prioritization on fault localization. Their research showed that test prioritization and test reduction could make fault localization more effective. Our study in turn shows that fault localization can improve the effectiveness of test selection.

In this paper, we analyze the programs and rank the functions by giving larger weights to functions possibly contain bugs. We use the information to change the distance of execution spectrum in order to get more accurate clustering result, as well as CTS.

III. OUR APPROACH

This section describes our approach. We will give a framework of our approach and then describe the detailed information about our approach.

A. Framework

The framework of WAS mainly contains three parts.

1. Running Tests. All tests are executed and their execution profiles are collected. For each test the subject program execution profile are collected. In the profiles, each attribute is a function of program. The profiles, function call profiles used in our approach, provide the information of whether each function is called in an execution. In order to measure the dissimilarity between profiles, the function call profiles are represented as a (0, 1) vector. In a vector, 1 represents that a function is called and 0 represents that a function is not called in a run.

2. Cluster Analysis. The function call profiles are used for cluster analysis. The clustering algorithm used in WAS is K-means, which is one of the most popular clustering algorithms. Clustering techniques group tests with similar profiles into same clusters. In our experiment, we introduce the Euclidean distance as the dissimilarity metric between profiles. If $T$ and $T'$ represent 2 tests $t: <e_1, e_2, \ldots, e_n>$ and $t': <e'_1, e'_2, \ldots, e'_n>$. The distance between $t$ and $t'$ is $\sqrt{\sum_{i=1}^{n} (e_i - e'_i)^2}$. The $e_i$ represents whether function $f_i$ is executed in test $t$, $e'_i$ represents whether function $f_i$ is executed in test $t'$.

3. Assigning Weights. We iteratively select tests from clusters and use the spectra information to assign/change the weights of functions iteratively. In the iterative procedure, we use a kind of function called ranking metric [17] which computes a value for every function in the profiles to change the weights of functions. A ranking metric is a function which will calculate a score for every function of the execution spectra. The higher a score is the more the corresponding function to be more
possible buggy. That is, the function with the highest weight are considered the most likely to contain bugs.

In a ranking metric, a notation \(< a_{up}, a_{uf}, a_{ep}, a_{ef} > [2]\) is given to every function. The first part of the subscript indicates whether the attribute was executed (e) or not (n) and the second indicates whether the passing test (p) or failing test (f) [17]. For example, \(a_{ep}\) of a function is the number of passing tests and called the function. The metric maps the four values in the annotation to a single value. We expect that functions which contain bugs have high \(a_{ef}\) value and low \(a_{ep}\) value. When \(a_{ef}\) value reaches its maximum value and \(a_{ep}\) reaches its minimum value, the corresponding function is called by all the failed tests while no passing test calling it. Then the function has a high possibility to be buggy.

We use a metric called Jaccard [13] to calculate weights for functions. The intuition behind Jaccard is to measure the similarity of two sets. In our situation, the two sets are the set of tests that fail and the set of tests for which this function is called. The larger the similarity is the more this function likely to be buggy. The Jaccard metric is as following:

\[
FunctionWeight = \frac{a_{ef}}{a_{ef} + a_{nf} + a_{ep}} \tag{1}
\]

\(a_{ef} + a_{ep}\) indicates the number of tests which call the function during execution. \(a_{ef} + a_{nf}\) indicates the number of failing tests. \(a_{ef}\) indicates the number of the elements in the intersection of the two sets. \(a_{ef} + a_{nf} + a_{ep}\) indicates the number of elements in the union of the two sets. The larger weight of function in equation (1) indicates that more failing tests call the particular function and at the same time less passing tests call the function. In other words the function is more related to the failing tests than passing tests. Hence, it is expected that the function may contain bugs.

Function weights are assigned after cluster analysis of original execution profiles. We start our procedures from a cluster randomly selected. Figure 1 shows the detailed procedure of assigning/changing function weights. We induce the function weight changing to 5 procedures:

1. **Test selection**: select a test which has the highest suspiciousness. If more than one test have the highest suspiciousness we randomly select one.

2. **Function confidence calculation** first Test result inspection is actualized. The result of the test is inspected to see whether it passes or fail. The inspection result will guide us to analyze the program to see which parts of the program will affect the failed or success execution. Second implement the Confidence of functions calculation. The confidence of functions are set according to whether the selected tests pass. When the selected test passes, the confidence of each function it calls will be increased by 1, otherwise, the confidence of these functions will be decreased by 1. This kind of quantitative analysis can show more accurate information about which function may affect the failure or success execution.

3. **< \(a_{up}, a_{uf}, a_{ep}, a_{ef} >\) calculation**: calculate the **< \(a_{up}, a_{uf}, a_{ep}, a_{ef} >\)** which is used to measure the weights of functions. If the selected test passes, the \(a_{ep}\) value of every called function is increased by 1 and the \(a_{up}\) value of every uncalled function is increased by 1. If the selected test fails, the \(a_{ef}\) value of every called function is increased by 1 and the \(a_{uf}\) value of every uncalled function is increased by 1. Then we see whether test counts reach \(TL\). If the selected test counts reach \(TL\), Function weight calculation will be done. Otherwise Suspicous test identification will be implemented. In our experiment we set the value to 5% of the total number of tests.

4. **Function weight calculation**: calculate the Jaccard function for every function to give each function a weight. The execution profile is traversed. If a function is called during the execution its original value in the profile is replaced with its weight. Then by deleting the information of tests selected, a new profile is produced. We group the
profiles generated and rerun our approach from step 1).

5. **Suspicious test identification** first we do **Suspicious functions selection**. A parameter Confidence Threshold (CT) [19] is introduced. If the confidence of a function is smaller than CT, the function is a suspicious function. A suspicious function is a function which we think could contain faults. The intuition behind it is that if a function is called by more failing tests than passing tests, we think it may contain faults. Then we do **test suspiciousness calculation**. We use the number of all the suspicious functions to compute the suspiciousness of every unselected test in the cluster. The suspiciousness of a test equals to the number of suspicious functions it calls during the execution. A test is suspicious if its suspiciousness is larger than 0. After calculation, a collection of suspicious test is produced, if the Collection is empty, another cluster will be chosen and our approach will be run from the beginning. Otherwise the 5 procedures above will iterate.

Our approach terminates when all the clusters have no suspicious tests. Every time before dealing with a cluster, the confidence of every function is set to 0. In the original state, every function is a suspicious function.

<table>
<thead>
<tr>
<th>TABLE I</th>
<th>Example with Five Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>t1</td>
</tr>
<tr>
<td>t1</td>
<td>1</td>
</tr>
<tr>
<td>t2</td>
<td>0</td>
</tr>
<tr>
<td>t3</td>
<td>1</td>
</tr>
<tr>
<td>t4</td>
<td>1</td>
</tr>
<tr>
<td>t5</td>
<td>1</td>
</tr>
<tr>
<td>test</td>
<td>fail</td>
</tr>
</tbody>
</table>

In the following contents we will give an example to illustrate our way of selecting tests and changing the weight of functions. In this example we set the CT value to 1. As shown in Table I column one shows the five functions f1, f2, f3, f4, f5 which the tests may call. Column two to six shows five tests, their profiles and their inspection result. **Table II** shows the way of selecting tests. The suspiciousness of tests from t1 to t5 are calculated in column 5 and the suspiciousness of selected test is represented by *. The way of calculating $< a_{np}, a_{nf}, a_{ep}, a_{ef} >$, changing function weight and producing new profile will be shown in Table III.

The explanations of Table II is as follows. First the confidences of all the functions are set to 0. Then all the functions are suspicious functions. According to the calling functions of the five tests, we can calculate the suspiciousness of the tests. t1 is selected as t1 has the maximum suspiciousness. As t1 fails and it calls f1, f3, f4 and f5, the confidence of the functions is decreased by 1. Now all the confidences of functions are still less than the CT value, so all the functions are suspicious functions. Now the suspiciousness of tests from t2 to t5 are 2,3,1,3 by recalculating.

Second we find that the largest suspiciousness is 3 and both of t3 and t5 hold the value. So we random select one from them. Assuming that t5 is selected the confidence of f1, f2 and f3 is increased by 1 as t5 passes and it calls f1, f2 and f3. The f1, f3, f4 and f5 are suspicious functions. Among the not selected tests t3 has the maximum value of suspiciousness as it calls two of the suspicious functions while others calling one.

Third t3 is selected. After selecting the confidences of f2, f4 and f5 are increased to 2, 0 and 0 as t3 passes and calls f2, f4 and f5. Now all the functions except f2 are suspicious functions. According to the calling functions of tests the suspiciousness of remaining tests t2 and t4 are 1.

Then t2 is randomly selected. It passes and calls f2 and f3 so the confidence of f2 and f3 are increased to 3 and 1 while others are 0. So the suspicious functions are f1, f4 and f5. t4 calls one of them so its suspiciousness is 1. At last t4 is selected which is not shown Table II.

Table III shows the way we change function weight and change the original profile shown in Table I to a new profile using the function weight we calculate. The tests selected order is in column one and two. The functions $< a_{np}, a_{nf}, a_{ep}, a_{ef} >$ value calculated each time after a test are shown in column three. We calculate the weight of functions after the five tests have been selected and the result is in column four. The new profile in column five is generated by using each function weight to substitute the value in original profile in Table I. The procedure is explained as follows:

At the beginning, each function’s $< a_{np}, a_{nf}, a_{ep}, a_{ef} >$ is $< 0, 0, 0, 0 >$. First t1 is selected. It fails and it calls function f1, f3, f4, f5. So the $a_{ef}$ value of these three functions is increased by 1. The $a_{np}$ value of uncalled function f4, f5 is increased by 1. Second t5 is selected. t5 passes and it calls f1, f2, f3. So the $a_{ep}$ value of these functions is increased by 1 and the $a_{np}$ values of uncalled functions f4, f5 are increased by 1. Third t3 is selected and it passes. The $a_{ep}$ values of called function f2, f4, f5 are increased by 1 and the $a_{np}$ values of uncalled f1, f3 are increased by 1. Then t2 is selected and the result of it is passed. The $a_{ep}$ values of f2, f3 are increased by 1 and the $a_{np}$ values of the uncalled f1, f4, f5 are increased by 1. Last t4 is selected. It passes and the $a_{ep}$ value of f1 is increased by 1 and the other functions $a_{np}$ values are increased by 1. The weight of all the functions are computed according to the last calculated $< a_{np}, a_{nf}, a_{ep}, a_{ef} >$ of every function.

Table II shows one iteration of function weight calculation. Each time a certain amount of tests are selected, we realize the function weight calculation using the data of $< a_{np}, a_{nf}, a_{ep}, a_{ef} >$ collected. After several iterations are actualized a function call profile with the weight of functions which can show how likely a function to be buggy is generated. With the profiles, we can obtain a better cluster result.

IV. Experiment

In [19], ESBS has been shown to be better than one-per-cluster sampling, 3-per-cluster sampling and adaptive sampling in both test detection ratio and failure detection ratio. To the
best of our knowledge, ESBS is the best of all the existing sampling strategies. We conducted experiments to evaluate the failure found efficiency of our approach, ESBS, one-per-cluster sampling and adaptive sampling. We compare the WAS and the other sampling strategies on both the test selection ratio and failure detection ratio. The purposes are to investigate the question that whether our approach is better than the ESBS strategies and other sampling strategies.

A. Subject Program

We experiment with three open-source programs fio, grep and gzip. The information of programs are shown in Table IV.

C. Experimental Step

1. Identify failures. In order to calculate the failure detection ratio, which tests fail must be known in advance. We run the same test on both the correct version and incorrect version of our subject program and collect the outputs respectively. If the two outputs are different we treat the test as a failing one.

2. Create execution profile. A tool named geov is used to record the different kinds of coverage information during the executions of tests. The information is analyzed to create the execution profile of each test. In the execution profile each attribute is a function in our experiment. The profile gives the information of whether a function is called while running a test. In the experiment, we denote the profile as a 0-1 vector, in which, 1 presents the function is called, 0 otherwise.

3. Cluster analysis. We used the data mining tool Weka to cluster the function call profiles. The clustering algorithm we use is K-means and the distance to measure the

### TABLE II

**Test Selection**

<table>
<thead>
<tr>
<th>Selected Order</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>Suspicious functions</th>
<th>Not selected test</th>
<th>Suspiciousness(t1,t2,t3,t4,t5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidence</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11,12,13,14,15</td>
<td>4,2,3,1,3</td>
</tr>
<tr>
<td>t1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>11,12,13,14,15</td>
<td>12,13,14</td>
<td>*1,2,1,3</td>
</tr>
<tr>
<td>t5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>11,12,13,14,15</td>
<td>12,14</td>
<td>*1,2,1,3</td>
</tr>
<tr>
<td>t3</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11,12,13,14,15</td>
<td>12,14</td>
<td>*1,2,1,3</td>
</tr>
<tr>
<td>t2</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>11,14,15</td>
<td>14</td>
<td>*1,1,1,1</td>
</tr>
</tbody>
</table>

### TABLE III

**Function Weight**

<table>
<thead>
<tr>
<th>Selected Order</th>
<th>Selected test</th>
<th>&lt; nexp, naf, nexp, naf &gt;</th>
<th>Weight</th>
<th>New profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>t1</td>
<td>f1&lt; 0,0,0,1,1 &gt; f2&lt; 0,1,0,0,0 &gt; f3&lt; 0,0,0,1,1 &gt; f4&lt; 0,0,1,0,1 &gt; f5&lt; 0,0,0,1,1 &gt;</td>
<td>f1(2/5) f2(0) f3(1/4) f4(1/3) f5(1/3)</td>
<td>2/5, 0, 1/4, 1/3</td>
</tr>
<tr>
<td>2</td>
<td>t5</td>
<td>f1&lt; 0,1,1,1 &gt; f2&lt; 0,1,1,1 &gt; f3&lt; 0,1,1,1 &gt; f4&lt; 1,0,1,1 &gt; f5&lt; 1,0,0,1 &gt;</td>
<td>0, 0, 1/4, 1/3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>t3</td>
<td>f1&lt; 0,1,1,1 &gt; f2&lt; 0,1,1,1 &gt; f3&lt; 0,1,1,1 &gt; f4&lt; 1,0,1,1 &gt; f5&lt; 1,0,0,1 &gt;</td>
<td>0, 0, 1/4, 1/3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>t2</td>
<td>f1&lt; 1,0,1,2 &gt; f2&lt; 1,0,1,3 &gt; f3&lt; 1,0,1,1 &gt; f4&lt; 1,0,1,1 &gt; f5&lt; 1,0,0,1 &gt;</td>
<td>2/5, 0, 1/4, 0, 0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>t4</td>
<td>f1&lt; 0,1,1,2 &gt; f2&lt; 0,1,1,3 &gt; f3&lt; 1,0,1,1 &gt; f4&lt; 1,0,1,1 &gt; f5&lt; 1,1,1,1 &gt;</td>
<td>2/5, 0, 1/4, 0, 0</td>
<td></td>
</tr>
</tbody>
</table>

### TABLE IV

**Subject Programs**

<table>
<thead>
<tr>
<th>Program name</th>
<th>Number of functions</th>
<th>Test suite size</th>
<th>Failure counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>fio</td>
<td>162</td>
<td>567</td>
<td>237</td>
</tr>
<tr>
<td>grep</td>
<td>87</td>
<td>214</td>
<td>12</td>
</tr>
<tr>
<td>gzip</td>
<td>82</td>
<td>214</td>
<td>12</td>
</tr>
</tbody>
</table>

They are all middle sized programs. The execution profiles of tests on small programs tends to be similar with each other. It is difficult to find their dissimilarities. Hence cluster analysis is limited when dealing with small size programs. These middle size programs could provide some evidences of our approach.

B. Evaluation Model

To evaluate ESBS and WAS we use two metrics, one is to calculate the ability of detecting failures, and the other is to calculate human efforts each method cost.

- **Failure-detection-ratio.** We use failure-detection-ratio to indicate the failure detection ability. If there are totally F failures, and a method finds F1 of them, then the failure-detection-ratio is:

  \[
  \text{failure-detection-ratio} = \frac{F1}{F} \times 100\%
  \]

- **Test-selection-ratio** We use test-selection-ratio to indicate the human efforts we cost during the experiment. If there are totally T tests, and a method finds T1 of them, then the test-detection-ratio is:

  \[
  \text{test-selection-ratio} = \frac{T1}{T} \times 100\%
  \]
dissimilarity between execution profiles is Euclidean distance. The clusters count we use for ESBS our approach is 15% of the test size.

4. WAS. We wrote programs to implement WAS and ESBS. We compare our approach with ESBS in both test-selection-ratio and failure-detection-ratio. For the two methods we both ran them 10 times and calculate the average test-detection-ratio and failure-detection-ratio. In this paper we set the TL value to 5% of number of tests.

D. Experiment Result

As Figure 2, Figure 3 and Figure 4 shows, our approach is better than ESBS, one-per cluster sampling, 3-per cluster sampling and adaptive sampling. WAS can find more failures while only need to select the same or less tests. Changing the CT value can make ESBS more efficient in failure detection. But our approach is always more efficient than ESBS even though the CT changes. For ESBS we set the CT value to 3, 4, 5 and found that WAS always can detect more failures while inspecting less tests.

From Figure 4, we can see that our approach performs better on gzip and grep than on flex. Especially for grep WAS can find 100% of failures when only selected 12% of the tests. It shows that WAS can reduce the human efforts enormously. For grep, in the situation of 50% of all the failures has been detected ESBS needs to inspect at least more than 12% of the tests while WAS only needs to inspect 7% of the tests. For gzip, when 80% of failures have been found 15% of tests have to be inspected in ESBS but only 9% of tests needed to be inspected in WAS.

In order to study the effect of CT value to our experiment, we fix the number of clusters to 10% of the total number of tests and change the CT value of WAS in order to see the changing of experiment result. Figure 5, Figure 6 and Figure 7 shows the result. As it shows, for all of the three subject programs, when the CT value changed from 1 to 5 the lines became closer and closer to x axis. It conveys the information that the smaller CT value is the higher the failure detection capability WAS has. Especially for grep and gzip, the failure detection ratio of WAS is dramatically large when CT value is set to 1.

Different CT values represent different standards of defining suspicious functions. The larger CT value is the more functions will be seen as suspicious functions. Then the suspiciousness of tests in a cluster will increase. According to the definition of suspicious tests in this paper more tests seen as suspicious will be selected. So as the Figure 5, 6, 7 shows, when the CT value became larger more tests are selected and a little more failures was found.

In Figure 5, for flex 50% of all the tests are selected while
80% of all the failures are detected. When the CT value increased to 5, the failure detection ratio can get a modest increase to 82% while 70% of tests needed to be selected. In Figure 6 for grep adding the CT value can no longer increase the failure detection capability. In Figure 7, for gzip when the CT value is set to 1 WAS can find 90% of the tests only needing to select 10% of tests. So setting small CT values will take less effort while not reducing the failure detection capability of WAS.

Fig. 5. the effect of different CT values to WAS on flex

Fig. 6. the effect of different CT values to WAS on grep

Fig. 7. the effect of different CT values to WAS on gzip

In order to study the effect of the number of clusters to our to WAS, we set the CT value to 1 and change the number of clusters when running WAS. Figure 8, Figure 9 and Figure 10 shows the running result on flex, grep and gzip. From the result we can see that WAS’s failure detection capability increased with the number of clusters increasing from 0.5% to 15% of the number of tests. But after that WAS’s failure detection capability stop increasing even though the number of clusters continued to increase. More clusters can give us more detailed information about the similarity and dissimilarity of tests and can guide test selection more accurately. But too much number of clusters can no longer strengthen this kind of guide.

WAS improves the clustering precision by changing the distance space of tests. As the figures shown, the improved clustering precision can reduce the number of tests needed to select and increase the number of failures found. When using WAS, it is better to set small CT value and specific number of clusters which is no more than 15% of the number of tests.

Fig. 8. the effect of different numbers of cluster to WAS on flex
Fig. 9. the effect of different numbers of cluster to WAS on grep

Fig. 10. the effect of different numbers of cluster to WAS on gzip

E. Threats to validity

Our experiments may have some limitations. The limitations include the use of only a small set of subject programs and tests. In practice the situations may be more complex. But we believe that our subject programs are still representative. They are all middle size programs and they have been used in many researches. Another threat is that the types of errors are limited in the subject programs. In practical, there may be some more other different and complex bugs in programs, but in the experiment, all bugs in the programs are all real-life examples, which are really exist in the old versions of the program. And the last threat which is taken into our consideration is that all the subject programs are all C programs and have all of the features of C programs, we should take notice for pointers, memory allocation, varieties of arithmetic, complex flow control and so on.

V. CONCLUSION AND FUTURE WORK

In this paper, we proposed WAS and conducted an experiment to verify it. Inspired by fault localization, we introduce weighted attributes to improve CTS. Our experimental results show that WAS can get lower test selection ratio and higher failure detection ratio than ESBS. Please note that WAS has a great room for improvement. We just use one kind of fault localization technique, the Jaccard ranking metric, which is simple. There may be other ranking metrics which are more effective than Jaccard. We will use other metrics in the future. We will conduct more experiments to verify which metric fits our approach better. It is interesting in combining machine learning techniques [4] and program spectra information to improve CTS. We will evaluate the cost-effectiveness of combination techniques. For the programs in real world, there are always multi-bugs rather than single bugs. We would implement WAS based on multi-label learning technique [11] to achieve more effectiveness of CTS in the future.

ACKNOWLEDGMENT

The work described in this article was partially supported by the National Natural Science Foundation of China (90818027, 61003024, 61170067).

REFERENCES

[12] Jiawei Han and Micheline Kamber. Data mining concepts and techniques, second edition. 2010.


