The Journal of Systems and Software 135 (2018) 107–125

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Test case prioritization for object-oriented software: An adaptive random sequence approach based on clustering☆

Jinfu Chen[a,*], Lili Zhu[a], Tsong Yueh Chen[b], Dave Towey[c], Fei-Ching Kuo[b], Rubing Huang[a], Yuchi Guo[a]

[a] School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, 202000, China
[b] Department of Computer Science and Software Engineering, Swinburne University of Technology, Hawthorn, 3122, Australia
[c] School of Computer Science, University of Nottingham Ningbo China, Ningbo, 315100, China

ARTICLE INFO

Article history:
Received 7 March 2017
Revised 21 September 2017
Accepted 30 September 2017
Available online 7 October 2017

Keywords:
Object-oriented software
Adaptive random sequence
Test cases prioritization
Cluster analysis
Test cases selection

ABSTRACT

Test case prioritization (TCP) attempts to improve fault detection effectiveness by scheduling the important test cases to be executed earlier, where the importance is determined by some criteria or strategies. Adaptive random sequences (ARSs) can be used to improve the effectiveness of TCP based on white-box information (such as code coverage information) or black-box information (such as test input information). To improve the testing effectiveness for object-oriented software in regression testing, in this paper, we present an ARS approach based on clustering techniques using black-box information. We use two clustering methods: (1) clustering test cases according to the number of objects and methods, using the K-means and K-medoids clustering algorithms; and (2) clustered based on an object and method invocation sequence similarity metric using the K-medoids clustering algorithm. Our approach can construct ARSs that attempt to make their neighboring test cases as diverse as possible. Experimental studies were also conducted to verify the proposed approach, with the results showing both enhanced probability of earlier fault detection, and higher effectiveness than random prioritization and method coverage TCP technique.

1. Introduction

Software testing is an important approach for ensuring the quality and reliability of software. Since the development of object-oriented (OO) technology, object-oriented software (OOS) has become widely used. However, testers may face challenges when attempting to apply traditional software testing approaches to OOS testing, due to some special characteristics of OO languages such as encapsulation, inheritance and polymorphism (Binder, 1996; Pezze and Young, 2004; Chen et al., 1998). Many OOS testing approaches have been studied, including random testing (RT) (Anand et al., 2013), state-based testing (Holt et al., 2014), and sequence-based testing (Hui et al., 2005). Among these approaches, RT has often been used in industry, partly due to its simplicity (Hamlet, 2002; Chen et al., 2015). Other testing approaches generally require more professional testing skills, and often focus on some specific kinds of software. A problem with the evolution of OOS is that test suites generated by these OOS testing approaches often include very large numbers of test cases, and hence execution of all of them can incur a very high cost (Nie et al., 2015; Ledru et al., 2012; Ciupa et al., 2008).

In order to improve the testing efficiency of OOS in regression testing, we need to prioritize test cases to find faults as quickly as possible. Generally speaking, since only some test inputs can detect faults, if these particular inputs could be prioritized for early execution, then the testing efficiency could be greatly improved. This kind of test case prioritization (TCP) should make it possible to detect faults earlier (Elbaum et al., 2002).

Current TCP techniques are developed based on white-box or black-box information (Zhang et al., 2016). The white-box information often includes program source code coverage, a program model and fault detection history; and black-box information usually includes test input information. In regression

☆ A preliminary version of this paper was presented at the 7th IEEE International Workshop on Program Debugging (IWPD 2016) Chen et al. (2016).

* Corresponding author.
E-mail addresses: jinfuchen@ujs.edu.cn, cjfnet@126.com (J. Chen), lilizhu@ujs.edu.cn (L. Zhu), tychen@swin.edu.au (T.Y. Chen), dave.towey@nottingham.edu.cn (D. Towey), rbhuang@ujs.edu.cn (R. Huang), greta0928@163.com (Y. Guo).

https://doi.org/10.1016/j.jss.2017.09.031
0164-1212/

testing the white-box information is usually based on previous program versions, but the testing is done on the current version (Luo et al., 2016). TCP techniques using black-box information do not have this problem.

Random sampling is a black-box prioritization technique, and is usually used as a benchmark for effectiveness evaluation of other prioritization techniques. In order to improve the effectiveness of random sequences and present a better prioritization benchmark in regression testing, research has resulted in a prioritization technique using Adaptive Random Sequences (ARSs) (Zhang et al., 2016, 2014). ARSs can be regarded as an alternative random sequence, in which test cases are evenly spread in the input domain with the purpose of improving the performance of the random sequence. ARSs originated from the concept of Adaptive Random Testing (ART) (Chen et al., 2004, 2010, 2013; Barus et al., 2016), which is an enhanced version of RT that attempts to improve RT's failure-detection effectiveness by evenly spreading test inputs throughout the entire input domain. Adaptive random sampling can generate ARSs to make the selection of the ordered test cases as diverse across the input domain as possible (Liu et al., 2014).

ARSs have been applied to TCP for process-oriented software, based on ART techniques (Zhang et al., 2014; Chen et al., 2010; Barus et al., 2016). We first used the ARS technique on complex OO programs (Chen et al., 2016), and proposed an ARS approach for OOS test case prioritization. In this paper, we extend the previous work and use the notion of clustering to generate ARSs for OOS, with test cases of similar properties grouped into the same cluster, and test cases in the same cluster being different from those in other clusters. Intuitively speaking, test cases in the same cluster may have similar fault detection capability (Aqilburney and Tariq, 2014). Thus test cases extracted from different clusters should have different properties, and hence should be able to detect different failures. Based on this intuition, we used cluster analysis technology to generate ARSs from different clusters, aiming to achieve an even spread of the prioritized adaptive sequence test cases across the input domain.

In this paper, we report on using method object clustering (MOClustering) and dissimilarity metric clustering (DMClustering) to generate ARSs. MOClustering forms clusters according to the number of objects and the length of method invocation sequences, using the K-means and K-medoids clustering algorithms. DMClustering uses K-medoids clustering algorithm and the structure information of test inputs to form clusters according to the Object and Method Invocation Sequence Similarity (OMISS) metric (Chen et al., 2017), which is a dissimilarity measurement for the test inputs of OO programs (based on calculation of the dissimilarity between two series of objects and between two sequences of method invocations). Additionally, a sampling strategy called MSampling (maximum sampling) is used to construct the ARSs within the MOClustering and the DMClustering frameworks. Because the proposed approach uses three clustering algorithms, three ARSs are constructed. We conducted empirical studies using seven open source subject programs, with the results showing that the proposed approaches can effectively prioritize the test cases and enhance the failure detection effectiveness. In particular, DMClustering outperforms other methods in testing large scale programs with complex structure.

The remainder of this paper is organized as follows. The research background is given in Section 2. The three clustering algorithms are explained in Section 3. The ARS generation algorithm is presented in Section 4. The results of our empirical studies and experimental analysis are reported in Section 5. Some related work is discussed in Section 6. And the conclusion and future work are presented in Section 7.

2. Background

2.1. Regression testing

Regression testing is important for ensuring software quality and reliability. The purpose of regression testing is to ensure that the modified program still confirms to the software requirements (Legunsen et al., 2016). Regression testing techniques usually involve test case reduction and test case prioritization (Ledru et al., 2012; Gonzalez-Sanchez et al., 2011). Test case reduction selects a subset of a given test suite, and aims to reduce regression testing time by only re-running the test cases affected by code changes. Test case prioritization techniques aim to reorder test executions so as to maximize some objectives, such as detecting faults earlier or reducing the testing cost. Compared to test case reduction, test case prioritization may be a more conservative approach, because it does not discard test cases and only prioritizes them (Ledru et al., 2012).

2.2. Cluster analysis

Cluster analysis can be used to improve software testing effectiveness, using the basic idea that test cases with similar properties be grouped into the same cluster: test cases in the same cluster are similar to each another but different from test cases in other clusters. In general, most clustering methods can be classified into one of the following five categories (Suganya and Devi, 2010): (1) partition methods; (2) hierarchical methods; (3) density-based methods; (4) grid-based methods; and (5) model-based methods.

2.3. Test case prioritization

The purpose of test case prioritization (TCP) is to increase the test suite's rate of fault detection by scheduling test cases with higher priority to be executed earlier, according to some criteria. TCP can identify a permutation of a test suite, from the set of all possible permutations, that maximizes the value of a fitness function — where the function reflects a given testing goal, such as the number of detected faults. Elbaum et al. (2002) and Rothermel et al. (2002) proposed the weighted average percentage of faults detected (APFD) as a metric to measure prioritization performance. If T represents an ordered test suite containing n test cases, and F represents a set of m failures detected by T, then TF_i represents the number of test cases executed in T' before detecting fault i. The formula of APFD is defined as follows, with APFD values ranging from 0 to 1, and higher values indicating better fault detection rates.

$$APFD = 1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{nm} + \frac{1}{2n} \quad (1)$$

Existing TCP techniques are classified as either white-box or black-box (Legunsen et al., 2016; Henard et al., 2016). Most white-box TCP techniques are based on the coverage information of the test suite for previous program versions. The white-box approaches use a selected test coverage criterion to prioritize the test suites. Test coverage criteria mainly include statement coverage, branch coverage, path coverage, method coverage and class coverage. Black-box TCP techniques usually prioritize the test suites using information associated with the test input and output information. Black-box TCP techniques mainly include combinatorial interaction testing, input model diversity and input (output) test set diameter.

2.4. Adaptive random sequence

Chen et al. proposed Adaptive Random Testing (ART) as an enhancement to RT (Chen et al., 2004, 2010). ART attempts to

improve on RT's failure-detection effectiveness by evenly spreading test inputs throughout the entire input domain, using a similarity/dissimilarity metric (Chen et al., 2010, 2013). ART can be used not only to generate its own sequence of test cases, but also to order a given test suite to improve its chance of detecting failures earlier, with such an ordered sequence being called an Adaptive Random Sequence (ARS). Similar to ART, an ARS is also based on the idea of even spreading across the input domain — a concept that has been shown to effectively reveal failures faster. ARSs can be applied to regression testing, and may be a simple, effective, and relatively low-overhead alternate to random sequences (RSs), which are commonly used in regression testing. Thus, we can use ARSs to prioritize test suites, and to enhance the performance of regression testing for OOS.

2.5. Test case generation

In integration and system testing of OOS, a test case t can consist of two parts: *t.OBJ* and *t.MINV*, where *t.OBJ* is a list of objects and *t.MINV* is an ordered list of methods (representing a sequence of method invocations) in the test case. Before ordering the test cases, the test suites for regression testing must first be generated.The test suites are randomly generated in our approach. Since test cases are generated based on the class information of the program under test, it is necessary to first obtain and analyze the class diagram. Visual Studio (Microsoft visual studio, 2013) was used to obtain the detailed class information of the subject programs, and the class diagrams.

The test suites were randomly generated, and the generation steps are as follows. First, the class diagram of the program under test is obtained. Based on this, the second step is to create a random number of objects, with random values assigned to each member object. Next, a random number of methods are generated as the length of method sequence, and the method sequence is verified. Finally, values are assigned to the method parameters by calling a random value generator for the corresponding data type. As a result, a test case is generated. The above steps were repeated until sufficiently many test cases were generated.

3. Clustering algorithms

In this study, we used three methods to cluster test cases: MOClustering_means (method object clustering with K-means), MOClustering_medoids (method object clustering with K-medoids), and DMClustering (dissimilarity metric clustering with K-medoids). MOClustering_means and MOClustering_medoids used the Euclidean distance to calculate the dissimilarity between test cases, while DMClustering employed the OMISS metric to calculate the dissimilarity. In DMClustering, because the OOS test inputs involved objects and methods rather than numerical data, the K-means could not be calculated. Hence, only the K-medoids clustering algorithm was used in DMClustering.

3.1. Framework overview

Fig. 1 shows the framework for our approaches. Before generating the test suites, the class diagram of the program under test is first obtained and analyzed. Then the test suites are generated, with each test case consisting of objects and the methods called by these objects.

Next, three methods are applied to cluster test cases of the constructed test suites.In MOClustering (method object clustering), test cases are represented in the form of vectors, and K-means and K-medoids clustering algorithms are applied, using Euclidean distance, to cluster the test cases — MOClustering with K-medoids clustering algorithm is referred to as MOClustering_medoids; and

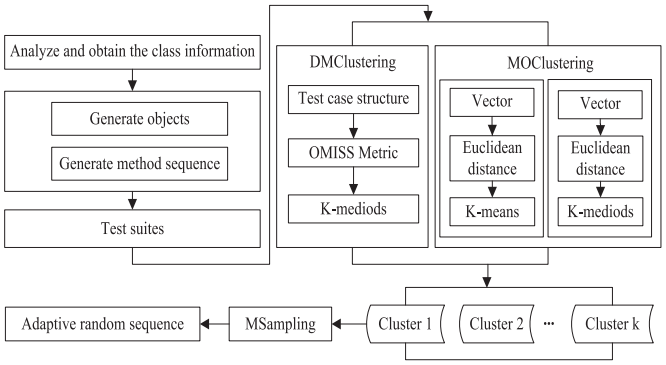

Fig. 1. TCP Framework.

MOClustering algorithm with K-means is referred to as MOClustering_means. In DMClustering, OMISS is used to calculate the dissimilarity between test cases, and the K-medoids clustering algorithm groups test cases into clusters. Finally, the adaptive random test sequences are generated using the MSampling (maximum sampling) strategy.

3.2. MOClustering

3.2.1. Object method vector

When conducting OOS integration and system testing, typically, a test case t will consist of a set of objects and an ordered list of methods. We therefore use an object method vector to represent a test case, defined as follows.

Definition 1. (object method vector, omv): An object method vector of a test case is defined as an ordered pair of the number of its objects and the total number of methods called by all of its objects, denoted omv= < On, Mn >, where On is the number of objects in the test input, and Mn is the total number of methods called by all objects that are in the test input.

For example, the object method vector for a test case t_1 with three objects and five methods called by all objects is represented as < 3, 5 >.

3.2.2. Distance measure

Because Euclidean distance is a natural measurement for distance between numerical data, it is used to measure the distance between pairs of *omv*. If X is the *omv* of t_1, and Y is the *omv* of t_2, with $X = < x_1, x_2 >$ and $Y = < y_1, y_2 >$, then the distance between X and Y is defined as:

$$d(X,Y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2} \quad (2)$$

For example, if X is < 3, 5 > and Y is < 3, 4 >, then the Euclidean distance between X and Y is equal to 1, because $d(X, Y) = \sqrt{(3-3)^2 + (5-4)^2} = \sqrt{1^2} = 1$.

3.2.3. MOClustering_Means algorithm

In MOClustering_means, test cases are clustered according to the numbers of objects and methods in each test case. The K-means clustering algorithm is efficient and scalable for large data sets, and was therefore used in MOClustering. The algorithm first selects K test cases as the initial data for each cluster. Each remaining test case is allocated to the closest cluster, defined by the lowest distance to the mean value of the cluster. The mean value of each cluster is then updated. This process is repeated until objects in each cluster no longer change or the sum of square error converges. After clustering, the test cases in the same cluster

are expected to be similar each another, and different to those in other clusters.

MOClustering_means is shown in Algorithm 1, and has three

Algorithm 1 MOClustering_means (*testcasepool, K, TNum*).

```
1: Construct OriginalTCase = {} to store the selected test cases;
2: Construct OMV = {} to store the set of object method vectors;
3: Construct Clustering = {} to store the generated clusters;
4: Construct meanValue = {} to store the mean value of each cluster;
5: Choose TNum test cases from testcasepool randomly and add them to OriginalTCase;
6: for (i=1 to TNum)
7:    On = |OriginalTCase[i].Objects|; //|OriginalTCase[i].Objects| is equal to the number of objects of OriginalTCase[i].
8:    Mn = |OriginalTCase[i].Methods|; //|OriginalTCase[i].Methods| is equal to the number of methods of OriginalTCase[i].
9:    OMV[i] =< On, Mn >; // The ith element of OMV is denoted by OMV[i].
10: end for
11: Choose K elements from OMV and add the corresponding test cases to Clustering as the initial cluster center;
12: Set change = true;
13: while (change == true)
14:    Update meanValue for each cluster;
15:    for (i=1 to TNum)
16:       for (j=1 to K)
17:          calculate d(OMV[i], meanValue[j]) according to Formula 2;
18:       end for
19:       Put the corresponding test case of OMV[i] to the nearest cluster;
20:    end for
21:    if (each cluster keep invariant)
22:       Set change = false;
23:    else
24:       Set change = true;
25:    end if
26: end while
27: return Clustering
```

input parameters: *testcasepool* (the simulated input domain), *TNum* (the number of test cases to be selected from *testcasepool* to form a test suite) and *K* (the number of clusters to be generated). The algorithm will generate *K* clusters for *TNum* test cases selected from *testcasepool*. In MOClustering_means, *TNum* test cases are first randomly selected to form a test suite that is to be prioritized; and the number of objects and methods is extracted from each chosen test case to construct the object method vectors set *OMV* for the *TNum* test cases, i.e., we construct the corresponding relationship between *OMV* and *TNum* test cases, and thus the test cases are grouped based on the corresponding clustering operation of the elements in *OMV*. Next, the first *K* test cases are selected as the initial cluster center of each cluster, and the mean value of each cluster updated according to Formula 3. Then, the Euclidean distance between each element of *OMV* and the mean value of each cluster are calculated, and the corresponding test case of each object method vector is assigned to the closest cluster. This is repeated until test cases in each cluster no longer change, or the sum of square error (Formula 4) converges. At this point, *K* clusters would have been generated and stored in the data set *clustering*.

Let *OMV(c)* be the set of object method vectors corresponding to cluster *c*. Suppose $OMV(c) = \{omv_1, omv_2, \ldots, omv_n\}$, where $omv_i =< On_i, Mn_i >, i = 1, 2, \ldots, n$, where On_i is the number of objects of the test input t_i in *c*, and Mn_i is the sum of the number

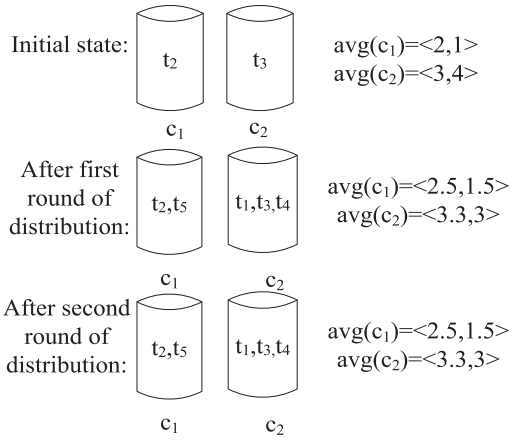

Fig. 2. Illustration of MOClustering_means clustering process.

of methods called by each object of the test input t_i in *c*. Let *avg(c)* denote the mean of cluster *c* which is defined as a vector of two mean values shown below:

$$avg(c) =< \frac{\sum_{i=1}^{n} On_i}{n}, \frac{\sum_{i=1}^{n} Mn_i}{n} > \quad (3)$$

Suppose that *C* is a cluster set, with $C = \{c_1, c_2, \ldots, c_K\}$, and *OMV(C)* (or $OMV(c_i)$) is the set of object method vectors corresponding to *C* (or c_i), with $OMV(C) = \bigcup_{i=1}^{K} OMV(c_i)$, where $OMV(c_i) = \{omv_{i1}, omv_{i2}, \ldots, omv_{ih}\}$. The mean value of cluster c_i — $avg(c_i)$ — is calculated according to Formula 3. Let *ES* denote the sum of square error among cluster set *C*, which is defined as:

$$ES = \sum_{i=1}^{K} \sum_{j=1}^{h} d(omv_{ij}, avg(c_i))^2 \quad (4)$$

For example, suppose that the test suites have five test cases, and we extract the number of objects and methods from each test case to construct *OMV*: $omv_1 =< 4, 3 >, omv_2 =< 2, 1 >, omv_3 =< 3, 4 >, omv_4 =< 1, 5 >$ *and* $omv_5 =< 3, 2 >$. Also assume that *K* is set to 2, and that test cases t_2 and t_3 are somehow chosen as the initial cluster centers. We first calculate the distance between $omv_i (i = 1, 4, 5)$ and omv_2, and the distance between $omv_i (i = 1, 4, 5)$ and omv_3, then put each test case into its nearest cluster. For example, since the distance between omv_1 and omv_2 is 2.83, and the distance between omv_1 and omv_3 is 1.41, then t_1 should be put into cluster c_2. After the first round distribution, cluster c_1 has two test cases t_2 and t_5, and cluster c_2 has three t_1, t_3 and t_4. The mean value of the new clusters should next be updated. After the second round distribution, cluster c_1 still has t_2 and t_5, and cluster c_2 still has three t_1, t_3 and t_4. Because the clusters are the same as in the previous round, the process of clustering is completed. Fig. 2 summarizes the three rounds of distribution for the above example.

3.2.4. MOClustering_Medoids algorithm

The K-medoids clustering algorithm randomly selects *K* test cases as the center points (also referred to as the representative test cases) of *K* clusters, and whenever the clusters are changed, the algorithm iteratively uses non-representative test cases (non-center points) to replace the representative test case, if necessary. The representative test case *O* is defined as follow (Kaufmann and Rousseeuw, 1987; Park and Jun, 2009).

Definition 2. A Representative Test Case,*O*, of a cluster is the test case that has the minimum absolute error value in the cluster.

The absolute error value (E) of the representative test case O is calculated by either Formula 5 or Formula 9 according to which distance metric is being used. In MOClustering_medoids, test cases are clustered according to their *Object Method Vectors*. Although K-means is efficient, it is also sensitive to outliers. Thus, when a test case with extreme values appears, the data distribution may be significantly distorted. The K-medoids algorithm can reduce the sensitivity to outliers by selecting a test case to represent the cluster without using the mean value. Thus, K-medoids was used in the MOClustering method to compare with MOClustering_means. The algorithm first selects K test cases to set up the initial K clusters. Each remaining test case is then allocated to the closest cluster, defined by the lowest distance to the representative test case of the cluster. The representative test case of each cluster is then updated. This process is repeated until test cases in each cluster no longer change. After clustering, the test cases in one cluster are close to the representative test case of that cluster, and far away from other clusters.

MOClustering_medoids is shown in Algorithm 2, and has three input parameters: *testcasepool* (the simulated input domain), *TNum* (the number of test cases to be selected from the simulated input domain to form a test suite on which prioritization is to be conducted) and K (the number of clusters to be generated). That is, the algorithm will generate K clusters for *TNum* test cases selected from *testcasepool*. In MOClustering_medoids, *TNum* test cases are first randomly selected from the simulated input domain as the initial data; and the number of objects and methods is extracted from each chosen test case to construct a data set *OMV* for these *TNum* test cases, i.e., we construct the corresponding relationship between *OMV* and *TNum* test cases, and the test cases are grouped based on the corresponding clustering operation on the elements of *OMV*. Next, the first K test cases corresponding to the first K elements from *OMV* are selected as the initial representative test cases for the K clusters and the selected representative test cases are stored in *RepreTCase*. Then, the Euclidean distance between each element of *OMV* and the *omv* of the representative test case O (of each cluster) is calculated, and the corresponding test case of each object method vector is assigned to the closest cluster. Finally, for every cluster, we consider each of its non-representative test cases, denoted by O', and calculate the absolute error value E' of O' using Formula 5 – if E' is less than E which is the absolute value of O, then O is replaced with O'. This is repeated until all clusters become steady, that is, there are no changes in any clusters after an updating process. By then, K clusters would have been generated and stored in the data set *clustering*.

Algorithm 2 MOClustering_medoids (*testcasepool, K, TNum*).

```
1: Construct OriginalTCase = {} to store the selected test cases;
2: Construct OMV = {} to store the set of object method vectors;
3: Construct Clustering = {} to store the generated clusters;
4: Construct RepreTCase= {} to store representative test cases of
   each cluster;
5: Choose TNum test cases from testcasepool randomly and add
   them to OriginalTCase;
6: for (i=1 to TNum)
7:    On = |OriginalTCase[i].Objects|; //|OriginalTCase[i].Objects| is
      equal to the number of objects of OriginalTCase[i].
8:    Mn = |OriginalTCase[i].Methods|; //|OriginalTCase[i].Methods|
      is equal to the number of methods of OriginalTCase[i].
9:    OMV[i] =< On, Mn >; // The element of OMV is denoted by
      OMV[i].
10: end for
11: Choose K items from OMV and add the corresponding test
    cases to RepreTCase as the initial representative test case;
12: Set change = true;
13: while (change == true)
14:    for (i=1 to TNum)
15:       for (j=1 to K)
16:          Calculate d(OMV[i], RepreTCase[j]) according to For-
             mula 2 ;
17:       end for
18:       Put the corresponding test case of OMV[i] to the nearest
          cluster;
19:       Update the cluster that OMV[i] corresponds to in
          Clustering;
20:    end for
21:    for (i=1 to K)
22:       for (each non-representative test case O' in the cluster)
23:          Compute its absolute error value E'; // Formula 5
24:          if (E' < E)
25:             RepreTCase[i] = O';
26:          end if
27:       end for
28:    end for
29:    if (each RepreTCase[i] keep invariant)
30:       Set change = false;
31:    else
32:       Set change = true;
33:    end if
34: end while
35: return Clustering
```

Suppose that $OMV(c)$ is the set of object method vectors corresponding to c, and $OMV(c) = \{omv_1, omv_2, \ldots, omv_n\}$. Let E denote the absolute error value of a test case O in cluster c, and $omv(O)$ be the element of $OMV(c)$ corresponding to O. In MOClustering_medoids, the absolute error value of the test case O is defined as:

$$E = \sum_{i=1}^{n} d(omv_i, omv(O)) \tag{5}$$

For example, suppose that the constructed test suite has five test cases (that is, *TNum* is 5) and their respective *OMV*: $omv_1 =< 4, 3 >$, $omv_2 =< 2, 1 >$, $omv_3 =< 3, 4 >$, $omv_4 =< 1, 5 >$ and $omv_5 =< 3, 2 >$. Also assume that K is set to 2, i.e., there are two clusters, c_1 and c_2. The calculation process of the earlier stage is the same as in Algorithm 2. Suppose we somehow choose two test cases as the initial representative test cases: t_2 for c_1 and t_3 for c_2. We first calculate the distance between $omv_i (i = 1, 4, 5)$ and omv_2, and the distance between $omv_i (i = 1, 4, 5)$ and omv_3, then put each test case into the nearest cluster. For example, as the distance between omv_1 and omv_2 is 2.83, and the distance between omv_1 and omv_3 is 1.41, then omv_1 should be put into cluster c_2. After the end of the first round distribution based on the similar operations, cluster c_1 has two test cases (t_2 and t_5), and cluster c_2 has three test cases (t_1, t_3 and t_4). Then we need to see whether the representative test case of each cluster needs to be updated or not. For example, in c_2, consider t_1. Calculate its absolute error value E_1 according to Formula 5. If E_1 is smaller than E_3 (which is t_3's E), then t_1 replaces t_3 to become the new representative test case. Other test cases in c_2 are also examined. If no change is observed for representative test cases of any cluster, then the clustering process is completed. Fig. 3 summarizes the three rounds of distribution for the above example.

3.3. DMClustering

3.3.1. OMISS metric

The OOS test input structure may be very complex because it may include different combinations of objects and methods, including multiple classes, multiple objects, inherited elements, reference objects, self-defined methods, and method invocation

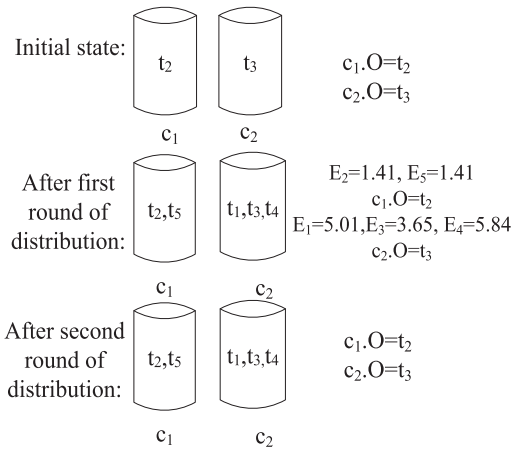

Fig. 3. Illustration of MOClustering_medoids clustering process.

sequences. To investigate the impact of using different distance metrics on test case prioritization, we use our recently developed OMISS metric to calculate the distance between test cases in the clustering process.

According to the OMISS metric (Chen et al., 2017), a test input t consists of an object set (*OBJ*) and a method invocation sequence (*MINV*), i.e, $t = \{t.OBJ, t.MINV\}$. The distance between test inputs (*TestcaseDistance*) is defined as the sum of the distance of object sets (*TCobjectDistance*) and the distance of method invocation sequences (*TCmSeqDist*), as shown in Formula 6. In Formula 6, $t_1.OBJ$ and $t_2.OBJ$ refer to the objects sets in *testcase1* and *testcase2*, respectively; and $t_1.MINV$ and $t_2.MINV$ represent the method invocation sets of *testcase1* and *testcase2*, respectively.

$$\begin{aligned} TestcaseDistance(t_1, t_2) = {} & TCobjectDistance(t_1.OBJ, t_2.OBJ) \\ & + TCmSeqDist(t_1.MINV, t_2.MINV) \end{aligned} \tag{6}$$

The distance between two object sets (*TCobjectDistance*) is calculated by comparing each pair of objects in the two sets, and is defined as the minimum sum of distances amongst all possible objects pairing between $t_1.OBJ$ and $t_2.OBJ$. An object can be divided into two parts: the attribute section and behavior section. The attribute section includes self-defined attributes (the attributes are defined by the current class), inherited attributes, and reference attributes. The behavior section includes self-defined methods and inherited methods. Hence, the distance between objects (*ObjectDistance*) is determined by the attribute section (*AttributeDistance*) and the behavior section (*BehaviorDistance*) of the object. The distance between objects is defined in Formula 7, where *p.A* refers to the attribute section of object *p*, *q.A* refers to the attribute section of object *q*, *p.B* means the behavior section of object *p*, and *q.B* means the behavior section of object *q*.

$$\begin{aligned} ObjectDistance(p, q) = {} & AttributeDistance(p.A, q.A) \\ & + BehaviorDistance(p.B, q.B) \end{aligned} \tag{7}$$

The distance between the two method invocation sequences, which is defined in Formula 8, includes the length difference, the set difference and the sequence difference. The sequence difference is calculated by *SequenceDissimilarity*($t_1.MINV$, $t_2.MINV$) in Formula 8 based on the ordered lists, and is equal to the number of common methods in the same position divided by the number of methods in the shorter sequence. For example, if there are two method invocation sequences, $t_1.MINV = \{m_3, m_2, m_1\}$, which has three methods, and $t_2.MINV = \{m_4, m_2, m_1, m_3, m_5\}$, which has five methods, then the length difference is 2; the set difference is 0.4 (1-3/5), because $t_1.MINV$ and $t_2.MINV$ have three common methods (m_1, m_2 and m_3) and five different methods (m_1, m_2, m_3, m_4, and m_5); the sequence difference is 0.667 (=2/3), because the second and third methods of $t_1.MINV$ are equal to the second and third methods of $t_2.MINV$; and $t_1.MINV$ is the shorter sequence, with a total of three methods. Therefore the distance between $t_1.MINV$ and $t_2.MINV$ is 3.067 (2+0.4+0.667).

$$\begin{aligned} & TCmSeqDist(t_1.MINV, t_2.MINV) \\ & = |length(t_1.MINV) - length(t_2.MINV)| \\ & \quad + \left(1 - \left|\frac{t_1.MINV \cap t_2.MINV}{t_1.MINV \cup t_2.MINV}\right|\right) \\ & \quad + SequenceDisssimilarity(t_1.MINV, t_2.MINV) \end{aligned} \tag{8}$$

The detailed explanations and examples with regard to Formulas 6,7, and 8 can be found in Chen et al. (2017).

3.3.2. DMClustering Algorithm

Because DMClustering applies to objects and methods, which are not numerical data, the K-means algorithm could not be used. Hence, only the K-medoids clustering algorithm was used in DMClustering.

DMClustering is shown in Algorithm 3, and has three input parameters: *testcasepool* (the simulated input domain), *TNum* (the number of test cases to be selected from *testcasepool* to form a test suite) and *K* (the number of clusters to be generated). The algorithm generates *K* clusters for *TNum* test cases selected from *testcasepool*. In DMClustering, *TNum* test cases are first randomly selected from the *testcasepool* as the initial data, and are then added to *OriginalTCase*. Next, *K* items from *OriginalTCase* are selected as the initial representative test case *O* of each cluster,

Algorithm 3 DMClustering (*testcasepool, K, TNum*).

```
1: Construct OriginalTCase = {} to store the selected test cases;
2: Construct Clustering = {} to store the generated clusters;
3: Construct RepreTCase= {} to store representative test cases of
   each cluster;
4: Choose TNum test cases from testcasepool randomly and add
   them to OriginalTCase;
5: Choose K items from OriginalTCase and add them to RepreTCase
   as the initial representative test case;
6: Set change = true;
7: while (change == true)
8:    for (i=1 to TNum)
9:       for (j=1 to K)
10:         Calculate TestcaseDistance(OriginalTCase[i], RepreTCase
            [j]); // Formula 6
11:      end for
12:      Put OriginalTCase[i] to the nearest cluster;
13:      Update the cluster of OriginalTCase[i] in Clustering;
14:   end for
15:   for (i=1 to K)
16:      for (each non-representative test case O' in the cluster)
17:         Compute its absolute error value E'; // Formula 9
18:         if (E' < E)
19:            RepreTCase[i] = O';
20:         end if
21:      end for
22:   end for
23:   if (each RepreTCase[i] keep invariant)
24:      Set change = false;
25:   else
26:      Set change = true;
27:   end if
28: end while
29: return Clustering
```

and the generated representative test case is stored in *RepreTCase*. Then, the difference between each remaining test case and each representative test case O (of each cluster) are calculated with the OMISS metric (Formulas 6, 7, and 8), and each test case is assigned to the nearest cluster. Finally, for each non-representative test case O', its absolute error value (E') is calculated using Formula 9 – if E' is less than E (the absolute value of O), then O is replaced with O' and the clusters are updated. This is repeated until items in each cluster no longer change, at which point, K clusters will have been generated and stored in *clustering*.

Suppose that T is the set of test cases for cluster c, $T = \{t_1, t_2, \ldots, t_n\}$. Let E denote the absolute error value of the representative test case O in cluster c. In DMClustering, the absolute error value of the test case O is defined as:

$$E = \sum_{i=1}^{n} OMISS(t_i, O) \tag{9}$$

For example, assume that a cluster c_1 has three test cases, t_1, t_2, and t_3. First, calculate the sum of the distances $OMISS(t_2, t_1)$ and $OMISS(t_3, t_1)$, denoted E_1 (the E for t_1). Then, calculate the sum of $OMISS(t_1, t_2)$ and $OMISS(t_3, t_2)$, denoted E_2 (the E for t_2), and the sum of $OMISS(t_1, t_3)$ and $OMISS(t_2, t_3)$, denoted E_3 (the E for t_3). If E_3 is smaller than E_1 and E_2, then t_3 is the representative test case of cluster c_1.

4. Adaptive random sequence generation

After all test cases have been clustered, a sampling strategy is needed to choose test cases from the clusters. The traditional random sampling strategy selects n test cases randomly from the entire pool of test cases. Some of these n test cases may be from the same cluster, which may have similar properties, including the ability to detect the same fault. Such a test case sequence may lead to a poor fault detection rate. The same problem occurs if random sampling is applied to choose a cluster, from which a test case is then selected. To maintain the diversity in test cases, we use a new MSampling (maximum) sampling mechanism.

MSampling is explained in Algorithm 4. It has three input parameters: K (the number of generated clusters), n (the specified number of test cases to be prioritized), and *clustering* (the K clusters generated by MOClustering and DMClustering), where n is less than or equal to the number of test cases in all K clusters. The specific steps of MSampling are: (1) Randomly choose an initial cluster. (2) When these clusters are generated by MOClustering, the distances between the selected cluster and the unselected clusters are calculated using Formulas 10 and 11. When the clusters are generated using DMClustering, the distance is calculated with Formula 12. (3) The most distant cluster is selected next. (4) Steps 2 and 3 are repeated until all clusters are selected, and an ordered sequence of clusters is generated. (5) According to the order of clusters in the sequence, randomly choose a unique test case from each cluster, in sequence. (6) Repeat Step 5 until the specified number (n) of prioritized test cases has been obtained. If the number of test cases selected in the current cluster is equal to the length of this cluster, we should jump to the next cluster. The prioritized test case sequence is stored in the data set *GTCases*.

When these clusters are generated by MOClustering_means, then, if C is a cluster set, and $C = \{c_1, c_2, \ldots, c_K\}$, AVG is the mean value set, and $AVG = \{avg_1, avg_2, \ldots, avg_K\}$, where avg_i is the mean value of c_i. Let $DMO\_M(c_i, c_j)$ be the distance between clusters c_i and c_j $(i, j = 1, 2, \ldots, K)$. The distance between any two clusters is defined as:

$$DMO\_M(c_i, c_j) = d(avg_i, avg_j) \tag{10}$$

Similarly, when the clusters are generated by MOClustering_medoids, if C is a cluster set, and $C = \{c_1, c_2, \ldots, c_K\}$, OS is a representative test cases set, and $OS = \{o_1, o_2, \ldots, o_K\}$, with o_i being the representative test case of the corresponding c_i, $(i = 1, 2, \ldots, K)$. Let $DMO\_K(c_i, c_j)$ be the distance between clusters c_i and c_j, $(i, j = 1, 2, \ldots, K)$. The distance between clusters is defined as:

$$DMO\_K(c_i, c_j) = d(omv(o_i), omv(o_j)) \tag{11}$$

With DMClustering, if C is a cluster set, and $C = \{c_1, c_2, \ldots, c_K\}$, OS is a representative test cases set, and $OS = \{o_1, o_2, \ldots, o_K\}$, with o_i being the representative test case of the corresponding c_i. Let $DDM(c_i, c_j)$ stand for the distance between clusters c_i and c_j, which is defined as:

$$DDM(c_i, c_j) = OMISS(o_i, o_j) \tag{12}$$

For example, if we have three clusters, $c_1 = \{t_{11}, t_{12}\}$, $c_2 = \{t_{21}, t_{22}\}$, and $c_3 = \{t_{31}, t_{32}\}$, then suppose c_2 is chosen as the first cluster, and the distances between c_1 and c_2 and between c_3 and c_2 are calculated. If the distance between c_1 and c_2 is less than that between c_3 and c_2, then the order of clusters is c_2, c_3 and c_1. Based on Algorithm 4, test cases are selected from c_2, c_3 and c_1 in sequence. Suppose $GTCases = (t_{21}, t_{31}, t_{11})$ after the first round of selection, then the next round of selection is conducted. At the end of the sampling strategy, a final adaptive random sequence for the prioritized test cases is generated: $GTCases = (t_{21}, t_{31}, t_{11}, t_{22}, t_{32}, t_{12})$. Test cases in the sequence are expected to be evenly spread in the input domain and are executed in this order in the testing framework.

Algorithm 4 MSampling(*K, n, clustering*).

```
1: Construct a set to store K clusters OC = {c_1, c_2, ..., c_i, ..., c_K};
2: Construct C = () to store the chosen cluster;
3: Construct GTCases =() to store the prioritized test case sequence;
4: Randomly choose a cluster c;
5: Add c to C;
6: while !(all clusters are added to C)
7:    for (i = 1 to K)
8:       if (OC[i] is not added to C)
9:          Calculate the distance between C and OC[i];
10:      end if
11:   end for
12:   Update c = the cluster that has the farthest distance with C;
13:   Add c to C;
14: end while
15: while !(the number of test cases in GTCases is up to n)
16:   for (each c in C (in their order in C and assume C is circular))
17:      if (the number of test cases selected in c < the length of c)
18:         Take a test case t from each cluster in turn;
19:         Append t to GTCases;
20:      else
21:         Jump to the next cluster;
22:      end if
23:   end for
24: end while
25: return GTCases;
```

5. Empirical studies and analysis

5.1. Setup of the empirical studies

Mutant programs are often used in empirical studies to investigate the fault detection effectiveness of different testing methods. Given the same test inputs, if the outputs produced by a mutant

Table 1
Subject programs.

ID	Name	Lines of code	Num. of public classes	Num. of public methods	Num. of faults	Description
1	CCoinBox (Codeforge-free open source codes forge and sharing, 2013)	120	1	7	4	C++ library that simulates a vending machine
2	WindShieldWiper (Codeforge-free open source codes forge and sharing, 2013)	233	1	13	4	C++ library that simulates a windshield wiper
3	SATM (Codeforge-free open source codes forge and sharing, 2013)	197	1	9	4	C++ library that simulates an Automatic Teller Machine
4	RabbitsAndFoxes (Sourceforge-download, develop and publish free open source software, 2013)	770	6	33	9	C# program that simulates a predator-prey model
5	WaveletLibrary (Codeplex-open source project hosting, 2013)	2406	12	84	15	C# library for wavelet algorithms
6	IceChat (Codeplex-open source project hosting, 2013)	571000	101	271	24	C# program that implements an IRC (Internet Relay Chat) Client
7	CSPspEmu (Github, where software is built, 2015)	406808	443	1433	26	C# program for a PSP (PlayStation Portable) emulator

version are different from the outputs produced by the original program, then these test inputs can be regarded as failure-causing inputs. Our study also used mutation programs to evaluate how quickly a test case prioritization method could find failure-causing inputs.

Table 1 presents the seven subject programs investigated in the experiment. The programs were all written in the C++ or C# language and are from some open sources websites (Codeforge-free open source codes forge and sharing, 2013; Sourceforge-download, develop and publish free open source software, 2013; Codeplex-open source project hosting, 2013; Github, where software is built, 2015). Faults were manually seeded into the subject program methods based on common mutation operators. In this study, we used the following 13 operators (Jia and Harman, 2010), which generate some typical program faults.

(1) arithmetic operators replacement (AOR);
(2) logical operators replacement (LOR);
(3) relational operators replacement (ROR);
(4) constant for scalar variable replacement (CSR);
(5) scalar variable for scalar variable replacement (SVR);
(6) scalar variable for constant replacement (SCR);
(7) array reference for constant replacement (ACR);
(8) new method invocation with child class type (NMI);
(9) argument order change (AOC);
(10) accessor method change (AMeC);
(11) access modifier change (AMoC);
(12) hiding variable deletion (HVD);
(13) property replacement with member field (PRM).

Of these 13 mutation operators, the last six are OO-specific, and are used to generate OO-specific faults. Table 2 shows the type of mutation operators and the number of faults seeded for each program. The machine used to conduct the testing has an Intel dual core i3-2120 3.3 GHz processor, 4 GB of RAM, and runs under the Windows 7 operating system.

Table 2
Mutation operators and the number of faults seeded.

ID	Num. of faults	Mutation operators (number)
1	4	AOR(1), LOR(2), ROR(1)
2	4	AOR(1), LOR(1), ROR(1), ACR(1)
3	4	AOR(1), LOR(1), ROR(1), SCR(1)
4	9	AOR(1), LOR(1), SVR(1), NMI(1), AOC(1), AMeC(1), AMoC(1), HVD(1), PRM(1)
5	15	LOR(1), SVR(1),CSR(2), SCR(1), ACR(2), NMI(1), AOC(1), AMeC(1), AMoC(2), HVD(2), PRM(1)
6	24	AOR(2), LOR(1), ROR(1), SVR(2),CSR(2), SCR(1), ACR(2), NMI(2), AOC(3), AMeC(2), AMoC(2), HVD(2), PRM(2)
7	26	AOR(2), LOR(1), ROR(1),SVR(2),CSR(1), SCR(1), ACR(2), NMI(2), AOC(3), AMeC(3), AMoC(3), HVD(3), PRM(2)

5.2. Effectiveness measure criteria

In our study, we used three measures to compare the TCP approaches: F_m (F-measure) – the number of the test cases executed before finding the first fault; E – the total number of distinct faults detected by a specific number of test cases; and *APFD* – the weighted average percentage of faults detected. A testing approach is considered effective if it has a low F-measure, a high E, and a high *APFD* value (Chen and Merkel, 2008). In this study, we compared MOClustering_means, MOClustering_medoids, DMClustering, and RT-ms (RT with method sequence — a random sequence generation approach for OOS test cases with method invocation sequence), and Method_Coverage (a method coverage TCP technique).

In order to properly assess the statistical significance of the differences between our methods and other methods, we conducted a statistical analysis based on the p-values and effect size (set at a 5% level of significance) using the unpaired two-tailed Wilcoxon-Mann-Whitney test and the non-parametric Vargha and Delaney effect size measure (Arcuri and Briand, 2014; Vargha and Delaney, 2000; Harman et al., 2012). The p-value is used to show the statistical significance of difference. If the p-value (probability value) is less than 0.05, which means that there is significant difference between the two compared methods, otherwise not (Arcuri and Briand, 2014). Additionally, we used the non-parametric effect size (ES) measure to show the probability that one method is better than another (Vargha and Delaney, 2000). That is, when we get the ES for any two methods A and B, a higher ES value indicates higher probability showing A is better than B. In this study, we used R language (Team, 2009) to obtain the p-value and ES value for the pair-wise TCP techniques.

Table 3
The value of K for each subject programs.

ID	Name	K	Percentage of the total number of test cases
1	CCoinBox	500	10%
2	WindShieldWiper	500	10%
3	SATM	500	10%
4	RabbitsAndFoxes	750	15%
5	WaveletLibrary	750	15%
6	IceChat	750	15%
7	CSPspEmu	750	15%

5.3. Experimental parameters

For both the *K*-means and the *K*-medoids clustering algorithms, *K* is the main input parameter. If the value of *K* is not suitable, low quality clusters may be generated: if test cases are clustered into too many clusters, then some similar test cases may be put into different clusters; if they are clustered into too few, then dissimilar test cases may be put into the same cluster. Both of these situations may lead to poor failure detection performance.

In this study, in order to find its most suitable value, *K* was set to 2%, 5%, 10%, 15%, 20%, 25%, and 30% of the total number of test cases (5000 test cases). Based on the overall experimental results, appropriate values of *K* for each subject program were determined, as shown in Table 3.

In addition, in all experiments (F_m, *E* and *APFD*), *testcasepool* in Algorithms 1–3 simulated the input domain, and *TNum* in Algorithms 1–3 was the total number of test cases (5000). The value of *n* in Algorithm 4 was set to 100, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000 and 5000.

5.4. Experiments

To evaluate the effectiveness of our approaches, we attempted to answer the following three research questions:

RQ1: Do cluster TCP techniques perform better than prioritization with random sequences or method coverage, in terms of F_m?

RQ2: Do cluster TCP techniques perform better than prioritization with random sequences or method coverage, in terms of *E*?

RQ3: Do cluster TCP techniques perform better than prioritization with random sequences or method coverage, in terms of APFD?

5.4.1. Results and discussion

1)Do MOClustering_means, MOClustering_medoids and DMClustering perform better than prioritization with random sequences and Method_Coverage, in terms of F_m ?

Table 4 summarizes the F_m results for the five different methods. All results in the table were averaged over 100 runs of tests for each subject program, each time with a different seed.

Table 4 shows that, for the CCoinBox program, MOClustering_means used the least number of test cases to detect the first failure, followed by MOClustering_medoids, Method_Coverage, DMClustering and RT-ms. For programs WindShieldWiper, MOClustering_medoids found the first fault with the least number of test cases, followed by DMClustering, MOClustering_means, Method_Coverage and RT-ms. For programs SATM, MOClustering_medoids found the first fault with the least number of test cases, followed by DMClustering, Method_Coverage, MOClustering_means and RT-ms. For the RabbitsAndFoxes program, the number of test cases used by MOClustering_means and DMClustering to detect the first failure was similar, and less than that for Method_Coverage, MOClustering_medoids and RT-ms. For the WaveletLibrary, IceChat, and CSPspEmu programs, DMClustering used the least number of test cases to find the first failure, and RT-ms used the most. For the programs IceChat and CSPspEmu, MOClustering_medoids performed better than MOClustering_means and RT-ms, but for the program WaveletLibrary, MOClustering_means performed better than Method_Coverage, MOClustering_medoids and RT-ms. Therefore, in terms of the F_m, on average, DMClustering performed best, especially for the large-scale programs, followed by MOClustering_means, MOClustering_medoids, Method_Coverage and RT-ms. Compared with RT-ms, DMClustering achieved an average of 18.72% improvement; MOClustering_means achieved an average of 17.07%; and MOClustering_medoids achieved an average of 15.62% improvement. Compared with Method_Coverage, DMClustering achieved an average of 9.55% improvement; MOClustering_means achieved an average of 7.71%; and MOClustering_medoids achieved an average of 6.09% improvement. Hence, the proposed cluster TCP techniques always performed better than prioritization with random sequences and method coverage prioritization, in terms of F_m.

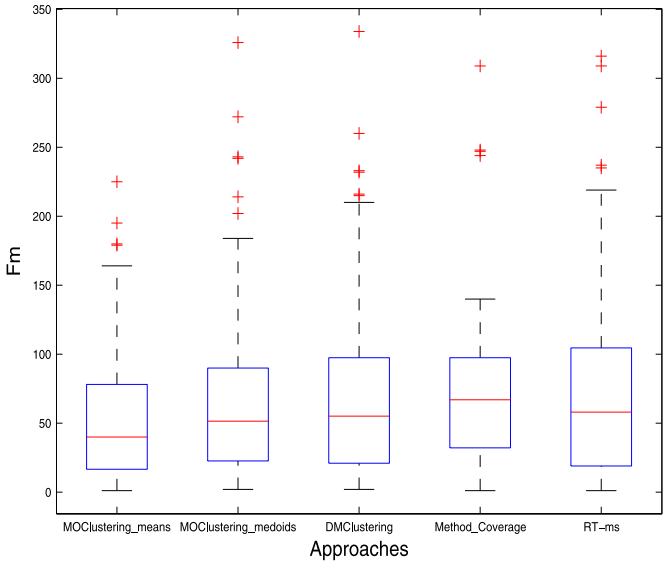

Fig. 4. F_m experimental results for CcoinBox.

In order to further analyze the F_m of each testing method for each subject program, Tables 4 and 5 also summarize the main statistical measures including *sDev* (standard deviation) for the 7 subject programs. The standard deviation for RT-ms is the biggest (58.20), which indicates that its data points are spread out over a wider range than other TCP techniques.

Fig. 4–10 are box-plots diagrams showing the F_m results for the seven subject programs, with the data in each box-plot being the F_m results over 100 runs for each subject program with different seeds.

As can be observed from Fig. 4, both the outlying and the maximum observed values of MOClustering_means are far smaller than corresponding values of RT-ms. Fig. 5 shows that the performances of the five methods are similar, but the medians of MOClustering_means, MOClustering_medoids and DMClustering are much smaller than the medians of Method_Coverage and RT-ms. This implies that in most cases, Method_Coverage and RT-ms required more test cases to find the first fault. As shown by Fig. 6, three cluster TCP techniques outperform other methods with smaller medians. As observed from Fig. 7, the performances of MOClustering_medoids, MOClustering_means, DMClustering, and Method_Coverage are similar, while they outperform RT-ms with larger outlying point values and maximum values. As seen from Figs. 8 and 10, DMClustering has the shorter IQR (interquartile range) and smaller medians than Method_Coverage and RT-ms,

Table 4
F_M of various TCP methods.

ID	F_m				
	MOClustering_means	*MOClustering_medoids*	*DMClustering*	*Method_Coverage*	*RT-ms*
1	53.94	70.37	72.15	71.92	74.87
2	63.88	58.54	58.85	68.84	75.54
3	49.93	46.67	47.17	49.05	52.90
4	21.88	27.08	22.91	24.02	28.45
5	6.96	8.58	6.75	8.44	8.66
6	37.58	36.54	33.14	40.19	55.00
7	85.14	77.14	71.98	83.53	89.67
mean	45.62	46.42	44.71	49.43	55.01
sDev	48.30	51.47	49.82	41.29	58.20

Table 5
Statistical result of F_M for 7 subject programs.

ID		MOClustering_means	MOClustering_medoids	DMClustering	Method_Coverage	RT-ms
1	mean	53.94	70.37	72.15	71.92	74.87
	sDev	48.93	64.43	64.39	53.93	70.01
2	mean	63.88	58.54	58.85	68.84	75.54
	sDev	61.27	66.24	57.75	48.59	71.40
3	mean	49.93	46.67	47.17	49.05	52.90
	sDev	43.13	38.56	40.43	36.79	51.90
4	mean	21.88	27.08	22.91	24.02	28.45
	sDev	17.37	24.50	20.80	16.42	29.32
5	mean	6.96	8.58	6.75	8.44	8.66
	sDev	6.22	6.88	5.39	5.50	7.67
6	mean	37.58	36.54	33.14	40.19	55.00
	sDev	34.09	35.88	34.57	34.13	39.36
7	mean	85.14	77.14	71.98	83.53	89.67
	sDev	53.06	56.03	54.40	60.42	61.91

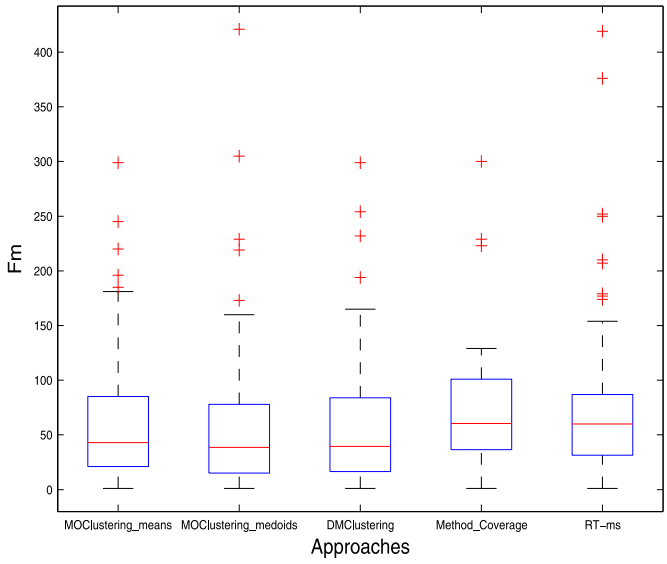

Fig. 5. F_m experimental results for WindShieldWiper.

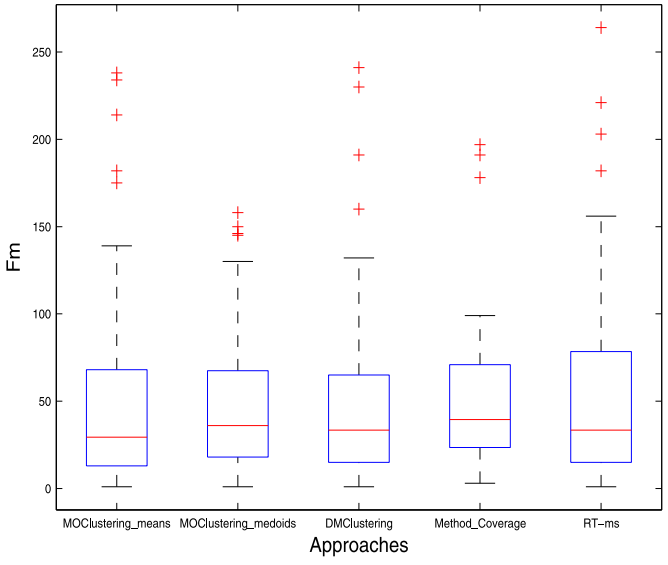

Fig. 6. F_m experimental results for SATM.

which means that its performance is more stable than that of these two methods for the two larger programs. Fig. 9 shows that all three cluster TCP techniques have smaller medians than Method_Coverage and RT-ms, which implies that their performances are better than those of these two methods for the program on average. From Figs. 4–10, we can find that the cluster TCP techniques have shorter IQRs and smaller medians than the other methods. Hence, they have more stable performance, especially for the larger programs.

In order to further study the significance of the differences in F_m, we report in Table 6 the p-value and effective size (ES) (Arcuri and Briand, 2014) for pairwise comparisons between the representative techniques from two different groups, from which we can find that the difference of F_m between our methods and RT-ms is significant (because the p-value is less than 0.05), and the difference in F_m between DMClustering and Method_Coverage is also significant. But the difference among our methods is not significant (because the p-value is larger than 0.05). Through a further analysis on the ES values for different pairwise comparisons, we can find that these values between our methods and other methods including RT-ms and Method_Coverage are larger than 0.5, which indicates that our methods perform better

Table 6
Comparison between various pairs of methods using p-value and effective size methods on F_M.

Pair of methods	MOClustering_means and RT-ms	MOClusterig_medoids and RT-ms	DMClusterig and RT-ms	MOClustering_means and Method_Coverage	MOClustering_medoids and Method_Coverage	DMClustering and Method_Coverage	MOClustering_means and DMClustering	MOClustering_medoids and DMClustering	MOClustering_means and MOClustering_medoids
P-value	0.007193	0.006183	0.000192	0.000943	0.000692	1.64E−05	0.261891	0.360047	0.844637
ES	0.5434847	0.5422582	0.557556	0.551042	0.5523612	0.566514	0.4826837	0.485873	0.503026
Better method	MOClustering_means	MOClustering_medoids	DMClustering	MOClustering_means	MOClustering_medoids	DMClustering	DMClustering	DMClustering	MOClustering_means

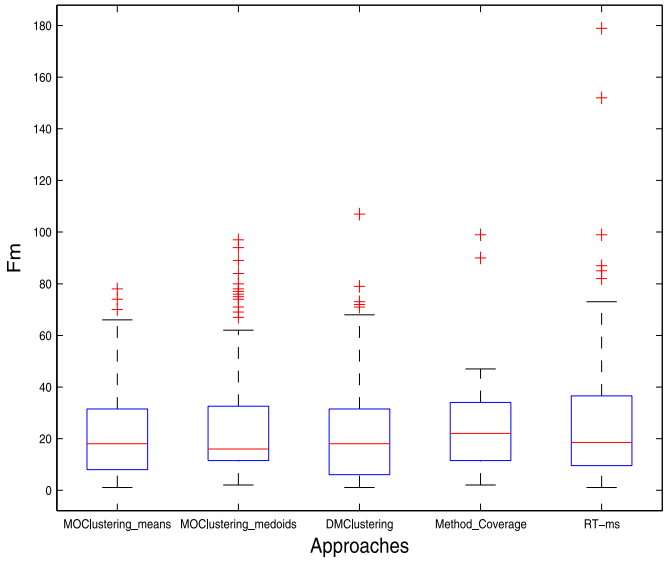

Fig. 7. F_m experimental results for RabbitsAndFoxes.

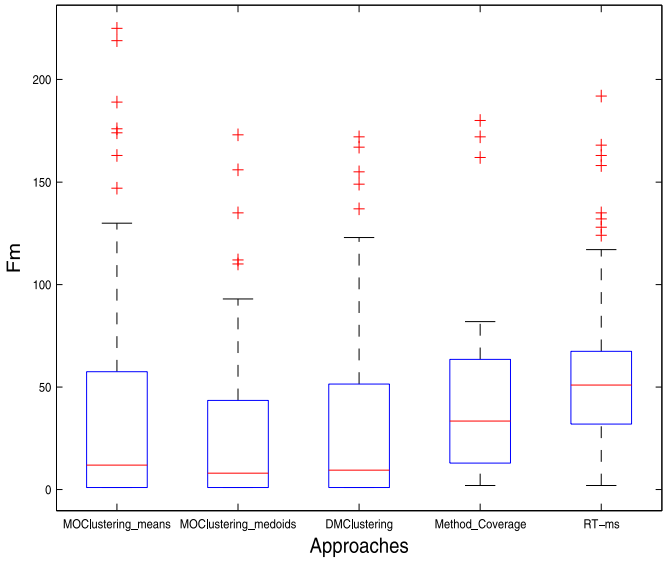

Fig. 9. F_m experimental results for IceChat.

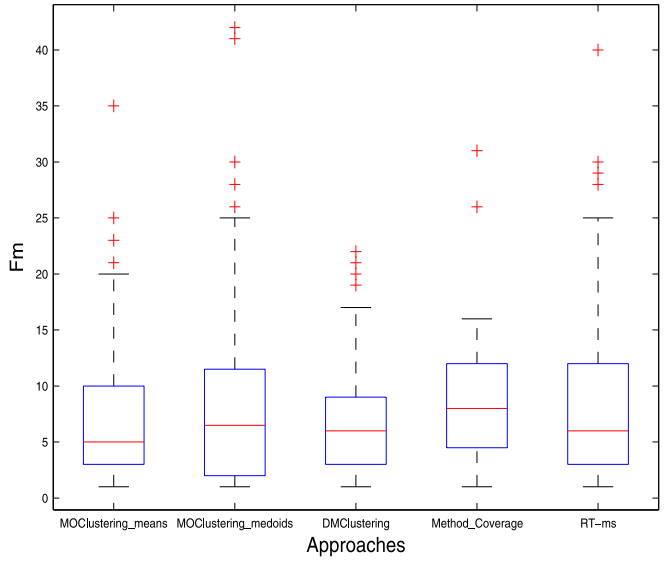

Fig. 8. F_m experimental results for WaveletLibrary.

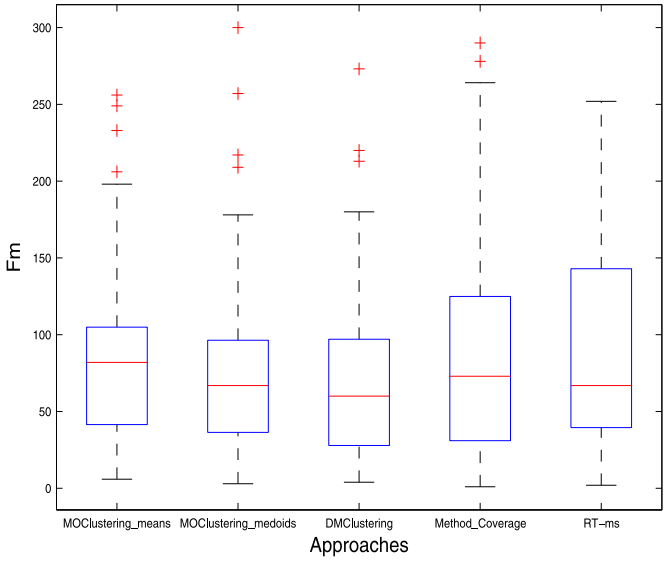

Fig. 10. F_m experimental results for CSPspEmu.

than RT-ms and Method_Coverage. Column "Better Method" of Table 6 presents the better method of the relevant pair. In three cluster TCP techniques, DMClustering performs best, followed by MOClustering_means and MOClustering_medoids on average.

We also analyzed the time taken to detect the first failure (F_m-*time*) for the different methods for the seven subject programs. Table 7 shows the F_m-*time* results for the five different methods. RT-ms required the least amount of time to detect the first failure. The testing time depends on the specific program under test, and the testing time generally includes both test case generation and execution time, which is usually the main cost in real testing activities. The testing times for MOClustering_means, MOClustering_medoids and DMClustering were not more than twice that of RT-ms on average.

RT-ms performs better than MOClustering_means, MOClustering_medoids, DMClustering and Method_Coverage in terms of

Table 7
F_M-Time of various TCP methods.

ID	F_m-time (Seconds)				
	MOClustering_means	*MOClustering_medoids*	*DMClustering*	*Method_Coverage*	*RT-ms*
1	0.71	0.86	1.13	0.67	0.64
2	1.15	1.37	1.85	1.05	0.96
3	0.79	0.83	1.17	0.71	0.68
4	0.83	0.92	1.22	0.73	0.67
5	0.74	0.82	1.06	0.69	0.62
6	1.28	1.64	2.13	1.14	0.97
7	1.84	2.36	3.04	1.46	1.35
Mean	1.05	1.31	1.66	0.92	0.84

Table 8
The sum of fault detected for all seven subject programs with different numbers of test cases.

Number of Test Cases	*E*				
	MOClustering_means	*MOClustering_medoids*	*DMClustering*	*Method_Coverage*	*RT-ms*
100	16.10	16.10	17.01	15.82	13.79
500	38.92	37.80	39.20	38.50	36.12
1000	46.55	45.71	46.97	44.10	42.77
1500	50.47	49.70	50.89	48.58	46.55
2000	53.41	52.64	54.11	51.52	49.14
2500	55.86	54.74	56.42	52.92	51.52
3000	57.75	56.77	58.52	56.14	53.55
3500	59.29	58.38	59.78	57.40	55.30
4000	60.76	60.06	61.25	58.80	57.19
5000	62.97	62.91	63.33	62.83	62.56

F_m-*time*, but it has low effectiveness in terms of F_m. DMClustering outperforms RT-ms in terms of F_m. Due to the complex structure of OOS test inputs in the subject programs under test, OMISS requires more time to calculate the distance between test inputs. Hence, DMClustering improves the fault detection effectiveness, but at the expense of more time for computing the OMISS metric.

On the other hand, MOClustering (especially MOClustering_means) outperforms RT-ms in terms of F_m, and outperforms DMClustering in terms of $F_m - time$. Since the distance between test inputs in MOClustering is calculated using the Euclidean distance which is much simpler than OMISS metric used in DMClustring, the $F_m - time$ of MOClustering was less than that of DMClustering on average. Therefore, according to the different testing requirements, we have a trade-off for employing different methods. In other words, when we know the approximate execution time for specific subject programs, we may be able to determine which method should be used based on F_m performance. For example, if the test case execution time is less than or equal to the test case generation time, then we may consider the influence of the generation time; but, if the generation time is much less than the execution time, then we may ignore its impact.

2) Do MOClustering_means, MOClustering_medoids and DMClustering perform better than prioritization with random sequences and Method_Coverage, in terms of E?

Table 8 shows the total number of distinct faults detected for seven subject programs using ten different test suite sizes – 100, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000 and 5000. All results were again obtained over 100 runs, with different seeds for each run.

Table 8 shows, as expected, that as the number of test cases used increases, the sum of distinct faults detected also increases. Furthermore, DMClustering has the best performance among the testing methods, followed by MOClustering_means, MOClustering_medoids, Method_Coverage and RT-ms.

Fig. 11 shows the total number of distinct faults detected by a number (n) of test inputs generated by each testing method, across all subject programs. We found that DMClustering outperformed all other methods, followed by MOClustering_means, MOClustering_medoids, Method_Coverage and RT-ms, regardless of the value of n.

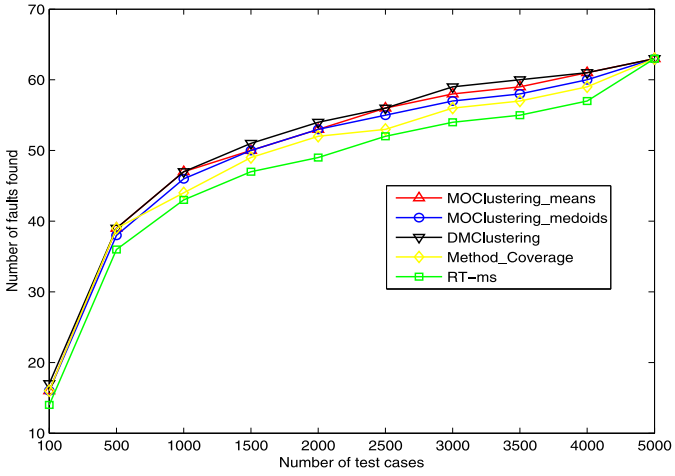

Fig. 11. Relationship between average number of distinct faults found and number of test cases used for all seven subject programs.

In order to further analyze the difference between different methods for each program as the number of test cases increases, Fig. 12–18 show the number of detected faults in ten stages – 100, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000 and 5000. Since different test suites may detect different numbers of faults in the 100 runs, the results were averaged over 100 runs, each time with a different seed and different test suites.

Looking at Figs. 12, 13, and 15, it appears that MOClustering_means has the best performance; MOClustering_medoids performs best in Fig. 14; and in Fig. 16–18, it is DMClustering that finds the most faults, regardless of the number of test cases used. This is because the distance metric used in DMClustering is more effective when applied to large-scale programs, but MOClustering_means and MOClustering_medoids are more effective in relatively small-scale programs. In Figs. 14 and 16, we can observe

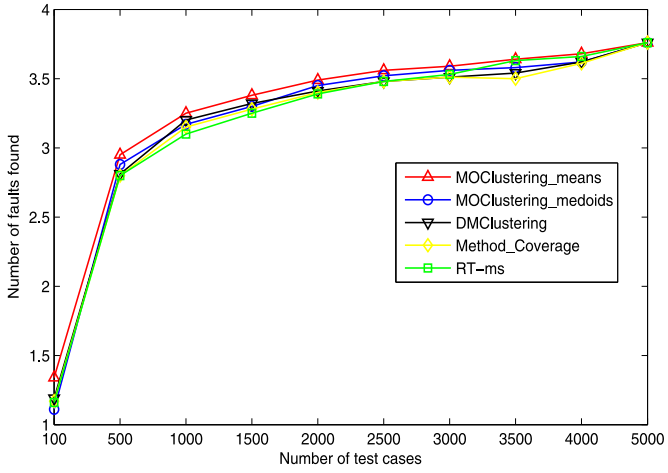

Fig. 12. Relationship between the average number of faults found and the number of test cases used for CcoinBox.

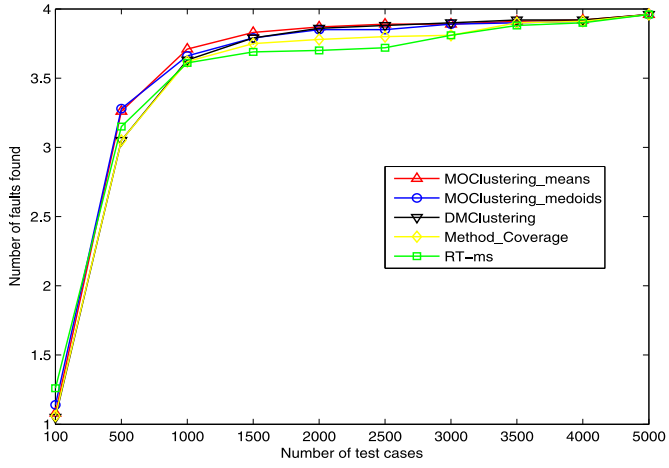

Fig. 13. Relationship between the average number of faults found and the number of test cases used for WindShieldWiper.

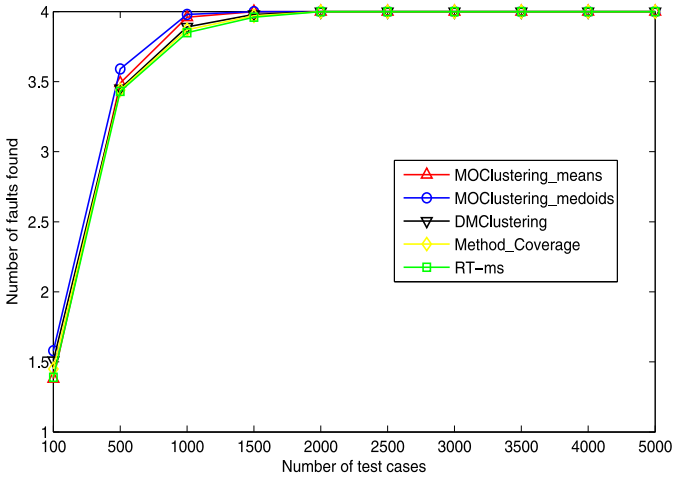

Fig. 14. Relationship between the average number of faults found and the number of test cases used for SATM.

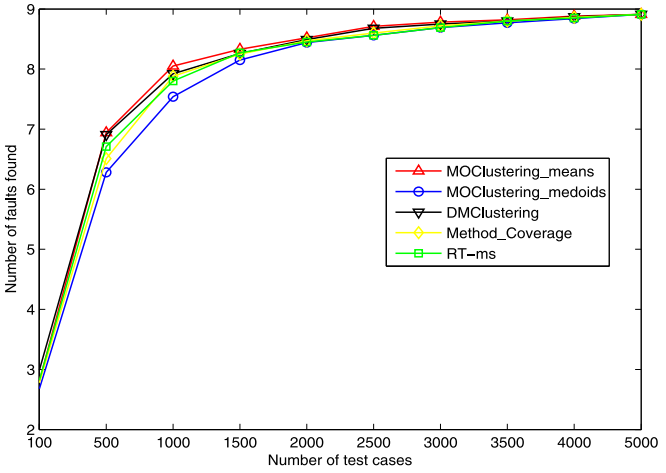

Fig. 15. Relationship between the average number of faults found and the number of test cases used for RabbitsAndFoxes.

that the lines of MOClustering_means, MOClustering_medoid, DMClustering, Method_Coverage and RT-ms are almost coincident when the number of test cases reaches 2000. The most appropriate explanation is that the rates of fault detection for SATM and WaveletLibrary are very high, and the faults are easily found. In Figs. 17 and 18, all methods display a trend of finding more faults as the number of test cases use increases. Through a further analysis, we can observe that some seeded faults in program IceChat and CSPspEmu are very difficult to be detected by random test cases, because they are associated with very lower failure rates. Thus, 5000 test suite is not large enough to detect all faults for these two programs, but large enough to detect all faults for the other programs.

From Table 8 and Figs. 12–18, we have the following observations: the number of faults detected increases as n increases; based on the 5000 test inputs (*TNum* in Algorithms 1–3), DMClustering outperforms other methods, followed by MOClustering_means, MOClustering_medoids, Method_Coverage and RT-ms (regardless of the value of n).

In order to further analyze the significance of the difference in E with different test cases, we report in Table 9 the p-value and effective size (ES) for pairwise comparisons between the representative techniques from two different groups. We find that the difference between our methods and RT-ms is significant (because the p-value is less than 0.05), and the difference between our methods and Method_Coverage is also significant, in most cases with different number of test cases. Through a further analysis of the ES values for different pairwise comparisons, we found that the ES values between our methods and RT-ms and Method_Coverage are larger than 0.5, which indicates that our methods perform better than RT-ms and Method_Coverage. Amongst the three cluster TCP techniques, DMClustering performs best, followed by MOClustering_means and MOClustering_medoids. In addition, when the number of test cases is 5000, all methods have similar results, which can be seen based on the values of p-value and ES. The reason for this is that 5000 test cases can find most of the faults in most of the subject programs.

3) Do MOClustering_means, MOClustering_medoids and DMClustering perform better than prioritization with random sequences and Method_Coverage, in terms of APFD?

Table 10 shows the average APFD values of the seven subject programs. All results were averaged over 100 runs, each time with a different seed.

As Table 10 shows, for program CCoinBox, MOClustering_means performs best, followed by MOClustering_medoids, DMClustering, Method_Coverage and RT-ms. For program RabbitsAndFoxes, MOClustering_means also performs best, and DMClustering outperforms MOClustering_medoids, Method_Coverage and RT-ms. For program SATM, the APFD values of three proposed methods are the same, and are much better than Method_Coverage and

Table 9
Comparison between various pairs of methods using p-value and effective size methods on E with different numbers of test cases.

Number of Test Cases	DMClustering and RT-ms		MOClustering_means and RT-ms		MOClustering_medoids and RT-ms		DMClustering and Method_Coverage		MOClustering_means and Method_Coverage		MOClustering_medoids and Method_Coverage	
	P-value	ES	P-value	ES	P-value	ES	P-value	ES	P-value	ES	P-value	ES
100	0.000017	0.642921	0.008701	0.574365	0.005234	0.566491	0.003482	0.531962	0.008672	0.523585	0.009823	0.523491
500	0.000236	0.623832	0.007124	0.553474	0.004352	0.537582	0.007573	0.542852	0.010583	0.524774	0.198732	0.512827
1000	0.000648	0.620743	0.009833	0.544585	0.006763	0.528643	0.006462	0.533946	0.008472	0.535668	0.007410	0.526918
1500	0.000092	0.615612	0.007721	0.555694	0.005542	0.539532	0.007573	0.524739	0.006384	0.526754	0.009321	0.527826
2000	0.000025	0.609758	0.005643	0.576783	0.003631	0.558443	0.005682	0.535648	0.005493	0.537863	0.008432	0.528935
2500	0.000138	0.607869	0.004532	0.567892	0.005742	0.549556	0.006894	0.536757	0.007502	0.528974	0.009543	0.520624
3000	0.000246	0.598750	0.008621	0.558763	0.006853	0.530447	0.008013	0.547868	0.009611	0.538083	0.004432	0.528530
3500	0.000571	0.579862	0.006732	0.559854	0.003962	0.551536	0.005124	0.538757	0.006520	0.529195	0.007541	0.539423
4000	0.000304	0.558751	0.003510	0.548743	0.004851	0.532425	0.006251	0.529847	0.008651	0.525206	0.008657	0.528934
5000	0.846479	0.509167	0.899845	0.508654	0.913564	0.501065	0.935468	0.500957	0.957851	0.500672	0.965874	0.500478

Table 10
APFD of various TCP methods.

ID	APFD				
	MOClustering_means	*MOClustering_medoids*	*DMClustering*	*Method_Coverage*	*RT-ms*
1	0.90	0.89	0.88	0.88	0.87
2	0.93	0.93	0.94	0.93	0.92
3	0.96	0.96	0.96	0.95	0.94
4	0.92	0.90	0.91	0.90	0.90
5	0.96	0.95	0.96	0.95	0.94
6	0.70	0.70	0.72	0.70	0.69
7	0.69	0.68	0.76	0.68	0.67
Mean	0.87	0.86	0.88	0.86	0.85
sDev	0.11	0.11	0.10	0.12	0.12

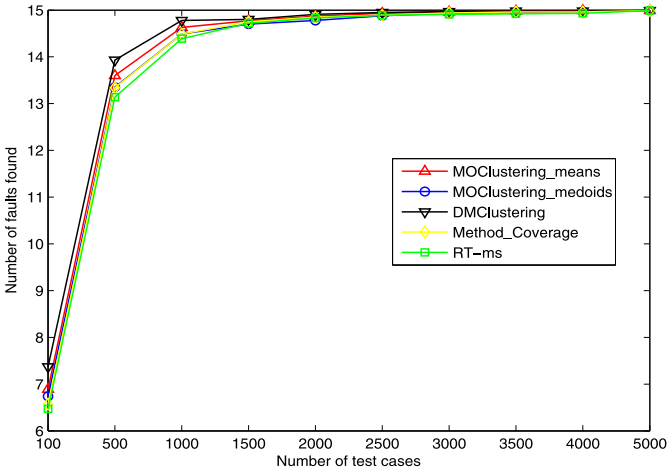

Fig. 16. Relationship between the average number of faults found and the number of test cases used for WaveletLibrary.

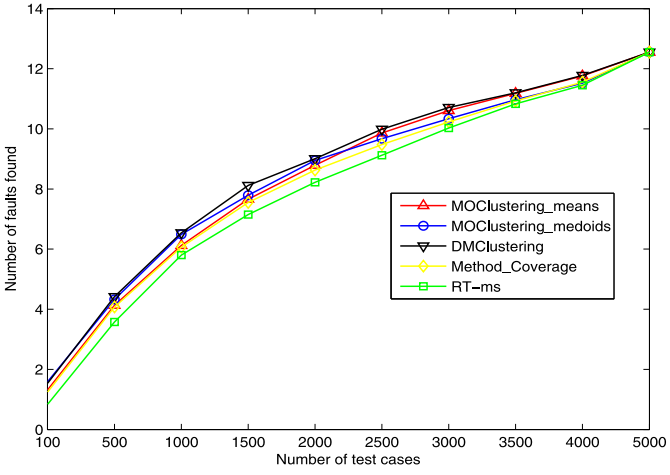

Fig. 17. Relationship between the average number of faults found and the number of test cases used for IceChat.

RT-ms. In programs WindShieldWiper, WaveletLibrary, IceChat and CSPspEmu, DMClustering performs best, followed by MOClustering_means, MOClustering_medoids, Method_Coverage and RT-ms in programs WindShieldWiper and WaveletLibrary, and followed by Method_Coverage, MOClustering_means, MOClustering_medoids and RT-ms in programs IceChat and CSPspEmu. On average, DMClustering performs best, followed by MOClustering_means, MOClustering_medoids, Method_Coverage and RT-ms.

In order to further analyze the APFD of each testing method for each subject program, Table 11 summarizes the major statistical measures including *sDev* (standard deviation) for the 7 subject programs. The standard deviation for RT-ms is the biggest (0.12), which indicates that the data points are spread out over a wider range of values than other TCP techniques.

Fig. 19–25 show the APFD box-plots for the seven subject programs. In the figures, the x-axis has the prioritization methods and the y-axis gives the APFD values for each method.

In Figs. 19–21, the upper quartile and median values of three cluster TCP techniques are all higher than those of Method_Coverage and RT-ms, implying a better performance, with respect to APFD values. In Fig. 22, the lower quartile, median and upper quartile values for MOClustering_means are all higher than those of the other methods. In Figs. 23–25, the lower quartile and median values for DMClustering are all higher than those of the other methods.

In order to further analyze the significance of the difference in APFD, we report in Table 12 the p-value and effective size (ES) for pairwise comparisons between the representative techniques from two different groups. We find that the difference between

Table 11
Statistical result of APFD for 7 subject programs.

ID		MOClustering_means	MOClustering_medoids	DMClustering	Method_Coverage	RT-ms
1	mean	0.90	0.89	0.88	0.88	0.87
	sDev	0.06	0.05	0.06	0.04	0.07
2	mean	0.93	0.93	0.94	0.93	0.92
	sDev	0.03	0.03	0.02	0.03	0.03
3	mean	0.96	0.96	0.96	0.95	0.94
	sDev	0.02	0.02	0.02	0.03	0.02
4	mean	0.92	0.90	0.91	0.90	0.90
	sDev	0.03	0.02	0.03	0.05	0.03
5	mean	0.96	0.95	0.96	0.95	0.94
	sDev	0.01	0.02	0.01	0.03	0.02
6	mean	0.70	0.70	0.72	0.70	0.69
	sDev	0.05	0.05	0.07	0.11	0.09
7	mean	0.69	0.68	0.76	0.68	0.67
	sDev	0.06	0.03	0.05	0.07	0.07

Table 12
Comparison between various pairs of methods using p-value and effective size methods on APFD.

Pair of methods	MOClustering_ means and RT-ms	MOClusterig_ medoids and RT-ms	DMClusterig and RT-ms	MOClustering_ means and Method_ Coverage	MOClustering_ medoids and Method_ Coverage	DMClustering and Method_ Coverage	MOClustering_ means and DMClustering	MOClustering_ medoids and DMClustering	MOClustering_ means and MOClustering_ medoids
P-value	8.58E−08	0.002017	2.59E−09	0.003581	0.004118	0.001594	0.202008	0.197287	0.604694
ES	0.582653	0.547663	0.591930	0.544963	0.535095	0.548731	0.480305	0.480291	0.503989
Better method	MOClustering_ means	MOClustering_ medoids	DMClustering	MOClustering_ means	MOClustering_ medoids	DMClustering	DMClustering	DMClustering	MOClustering_ means

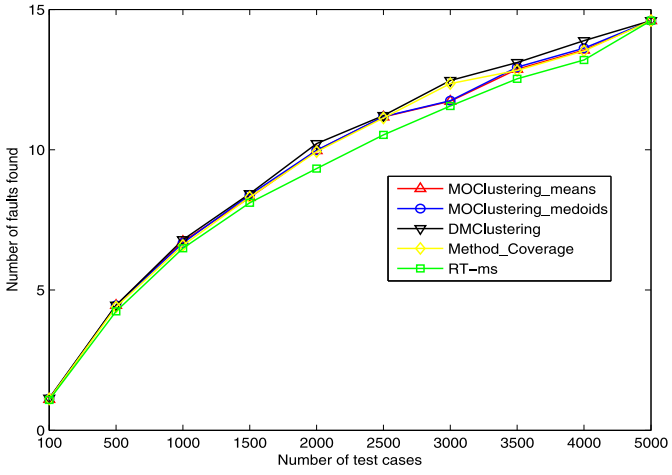

Fig. 18. Relationship between the average number of faults found and the number of test cases used for CSPspEmu.

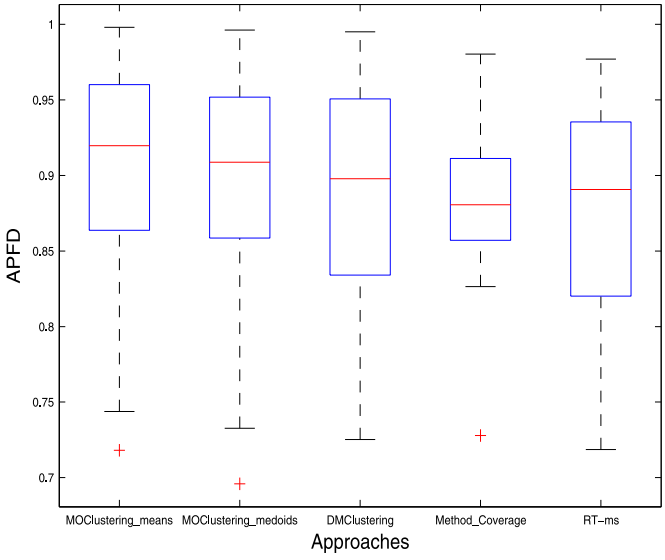

Fig. 19. APFD values for CcoinBox.

our approaches and RT-ms and Method_Coverage is significant (because the p-value is less than 0.05). But the difference among our proposed clustering methods is not significant (because the p-value is larger than 0.05). Through a further analysis on the ES values for different pairwise comparisons, we can find that these values between our methods and RT-ms and Method_Coverage are larger than 0.5, which indicates that our methods perform better than RT-ms and Method_Coverage. Column "Result" of Table 12 gives the better method of the relevant pair. Amongst three cluster TCP techniques, DMClustering performs best, followed by MOClustering_means and MOClustering_medoids on average.

In summary, based on Table 4–12, and Fig. 4–25, we have the following observations: (1) DMClustering performs better than other methods for larger programs (WaveletLibrary, IceChat and CSPspEmu); (2) MOClustering_means and MOClustering_medoids have good performance with smaller programs, and MOClustering_means is relatively more effective than MOClustering_medoids in terms of F_m, E and APFD; (3) generally speaking, DMClustering performs best in terms of F_m, E and APFD on average; (4) the three cluster TCP techniques (DMClustering, MOClustering_means and MOClustering_medoids) outperform Method_Coverage (method coverage prioritization) in terms of F_m, E and APFD; and (5) the four TCP techniques (DMClustering, MOClustering_means, MOClustering_medoids and Method_Coverage) perform better than RT-ms (prioritization with random sequences) in terms of F_m, E and APFD. All the above outperformance is statistically significant.

5.4.2. Threats to validity

Although we believe that the experiment was well-designed and implemented, the study may still face some threats to its validity, as explained in the following. In the clustering algorithms, the number of clusters K is generally required to be known in

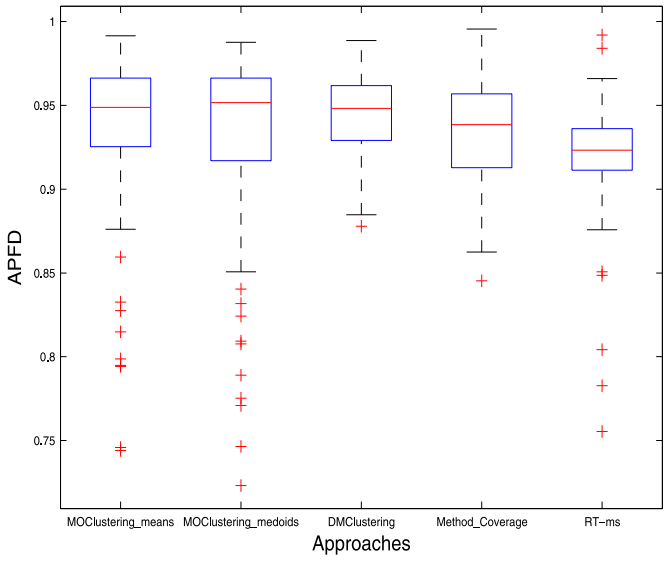

Fig. 20. APFD values for WindShieldWiper.

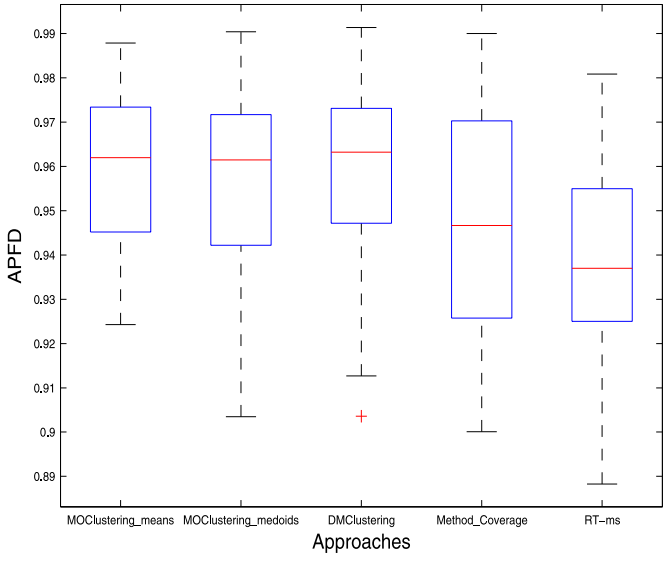

Fig. 21. APFD values for SATM.

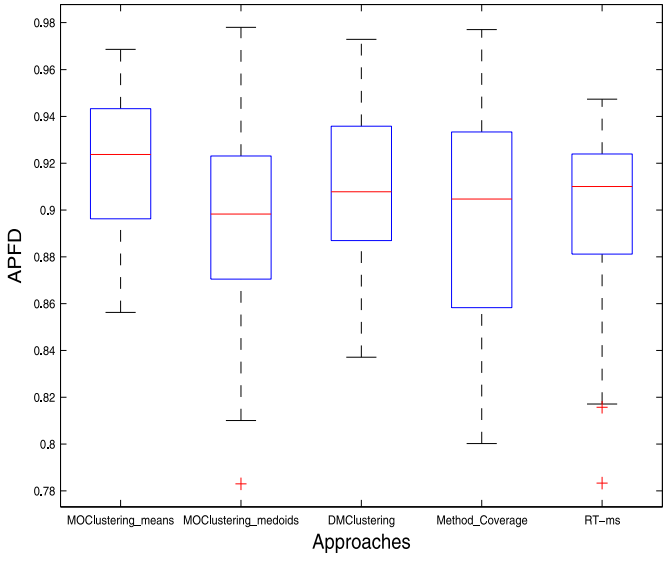

Fig. 22. APFD values for RabbitsAndFoxes.

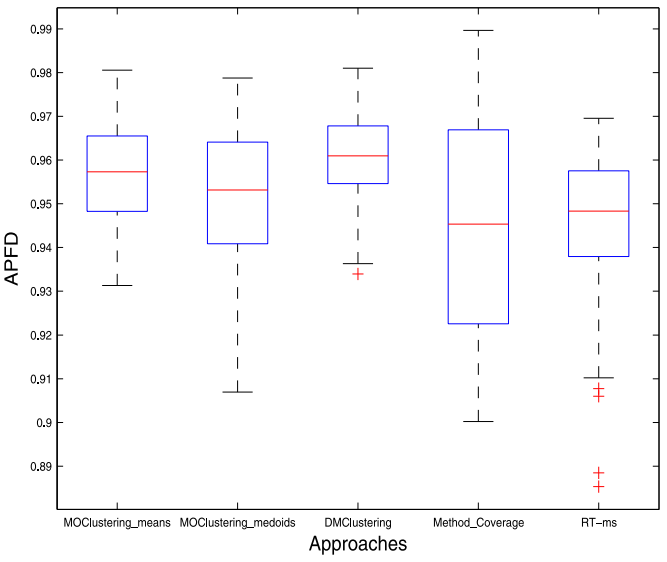

Fig. 23. APFD values for WaveletLibrary.

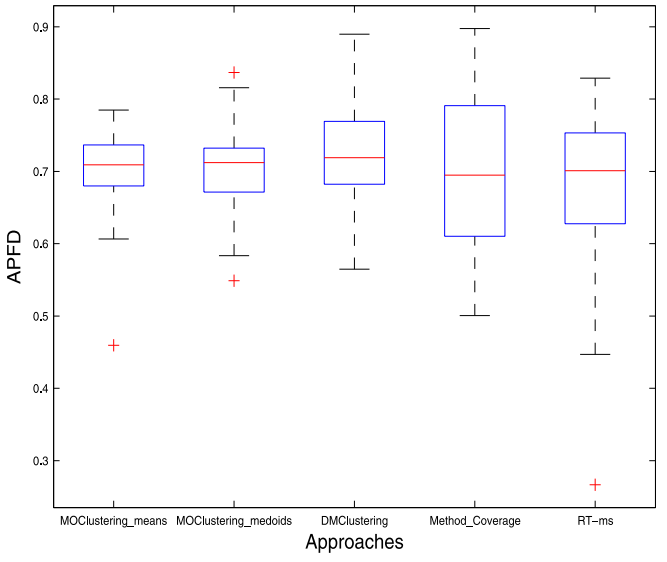

Fig. 24. APFD values for IceChat.

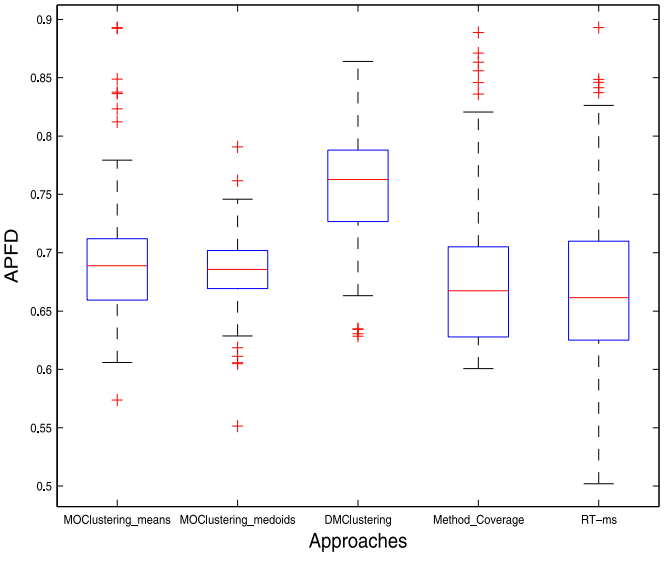

Fig. 25. APFD values for CSPspEmu.

advance. Obviously, the value of K has a significant influence on the clustering quality. Hence, if K is not correctly chosen, then the clustering analysis algorithm may produce low quality results. In this study, the value for K was determined experimentally, but in some other studies, it was determined according to the gap statistic algorithm (Tibshirani et al., 2001) and the distribution characteristics of the test cases. In addition, the subject programs were downloaded from some open source websites, but these subject programs may not be associated with any test cases. Although we tried our best to find some OO programs (in C# or C++), with real test cases for OO integration testing in some famous software repositories such as the Software Infrastructure Repository (SIR) (Software-artifact infrastructure repository, 2016), unfortunately, we did not find suitable ones. Hence, we developed a tool which randomly generates test cases for these subject programs. In the absence of real test suites, we believe that random test suites are fair and reasonable solutions.

In this study, the mutants in the seven subject programs were generated by hand, due to a lack of good automatic mutation tools for both C++ and C# programs. However, the location and type of seeded faults were selected using a random number generator, thus making the process semirandom and semiautomatic. Additionally, in order to reduce the threats, we manually filter as many subsumed mutants (Papadakis et al., 2016) as possible.

6. Related work

Chen et al. (2010) first suggested how to use ART in test case prioritization, calling such an approach an adaptive random sequence (ARS), and explaining how it could be a cost-effective alternative to random sequences. Rothermel et al. (2002) proposed several code coverage based TCP approaches, including total statement coverage prioritization, additional statement coverage prioritization, total branch coverage prioritization, and total fault-exposing potential prioritization. Their experimental results show that these methods can improve the fault detection rates of test suites.

Cluster analysis has drawn a lot of attention in the TCP community. Dickinson et al. (2001) proposed a clustering based test case filtering technique that improves on the efficiency of random sampling by using an agglomerative hierarchical clustering algorithm, which is a bottom-up approach, where each test case is used as a cluster, and the clusters with minimal dissimilarity are merged into larger clusters until a predefined number of clusters remain. They studied several dissimilarity metrics, including binary metric, proportional metric, SD (standard deviation) metric, histogram metric, linear regression metric, count-binary metric, and proportional-binary metric. The inputs to the cluster analysis are function call profiles. In the profile, each pair of methods is represented as an entry showing the frequency of the executed methods. Although this approach can reflect the dynamic behavior of test cases, only the methods' execution time (including whether or not the method is executed) is used.

Yoo et al. (2009) proposed a cluster-based TCP technique that significantly reduces the required number of pair-wise comparisons. Their clustering method partitions test cases into different subsets based on their dynamic runtime behavior, with test cases in each group having common properties. The clustering approach uses binary strings to represent test inputs and whether or not a statement is executed: If the source code statement has been executed, the digit of the corresponding bit in the binary string is set to 1; otherwise, it is set to 0. Zhang et al. (2014) proposed online and offline ARS-based TCP techniques using black-box information based on the string distances of the input data, without referring to the execution history and code coverage information. The offline TCP algorithm selects new test cases farthest from all prioritized ones; and the online algorithm uses feedback information such that the next prioritized test case depends on the existing execution results.

7. Conclusion and future work

Software testing is an important aspect of examining the quality and reliability of object-oriented software (OOS). Because OOS test cases may be very complex, traditional software testing approaches may not be appropriate for testing OOS. Although studies have been carried out to enhance OOS testing, OOS test case prioritization (TCP) has not yet been fully explored. TCP can increase fault detection rates by optimizing test case execution sequences such that more important test cases are executed earlier - based on some criteria. Cluster analysis has recently been applied to improving TCP effectiveness.

In this paper, in order to improve the effectiveness of TCP for OOS, we have proposed an ARS approach based on clustering techniques. We used three clustering methods to define our ARS methods: MOClustering_means, MOClustering_medoids and DMClustering. In MOClustering_means and MOClustering_medoids, test cases are clustered according to the number of objects and methods, using K-means and K-medoids clustering algorithms. In these two methods, the Object Method Vector is constructed to calculate the distance between test cases using the Euclidean distance formula. In DMClustering, test cases are clustered based on an object and method invocation sequence similarity (OMISS) metric with the K-medoids clustering algorithm. Furthermore, a sampling strategy MSampling is used to construct the ARSs. The final prioritized test case sequence is generated from the K clusters. The experimental results show that the three proposed cluster methods outperform Method_Coverage and RT-ms in terms of F_m, E and APFD; and DMClustering performs best overall, and is therefore a good choice for test case prioritization, especially for large scale OOS testing. Furthermore, all the better performances are statistically significant.

Based on the observations from our experimentations, we recommend that for large programs, it is better to set K to around 15% of the total number of test cases, and for small programs, it is better to set it to around 10%.

In future, we will conduct further investigations into OOS test case features, and add other important information to the Object Method Vector and OMISS metric to enhance the probability of the selected test cases for OOS to be more evenly spread across the input domain. We also will improve the sampling strategy to better optimize the TCP test cases.

Acknowledgement

This work is partly supported by National Natural Science Foundation of China (NSFC grant numbers: 61202110 and 61502205), the Postdoctoral Science Foundation of China (Grant numbers: 2015M571687 and 2015M581739), and the Natural Science Foundation of the Jiangsu Higher Education Institutions of China (Grant number: 15KJB520007). Dave Towey acknowledges the financial support from the Artificial Intelligence and Optimisation Research Group of the University of Nottingham Ningbo China; the International Doctoral Innovation Centre; the Ningbo Education Bureau; the Ningbo Science and Technology Bureau; and the University of Nottingham.

It is with deep regret and sadness that we report the passing of the fifth author Fei-Ching Kuo on October 6, 2017.

References

Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., Mcminn, P., Bertolino, A., 2013. An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. 86 (8), 1978–2001.

Arcuri, A., Briand, L., 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw. Test. Verification Reliab. 24 (3), 219–250.
Barus, A.C., Chen, T.Y., Kuo, F.C., Liu, H., Merkel, R., Rothermel, G., 2016. A cost-effective random testing method for programs with non-numeric inputs. IEEE Trans. Comput. 65 (12), 3509–3523.
Binder, R.V., 1996. Testing object-oriented software: a survey. Softw. Test. Verification Reliab. 6 (3), 125–252.
Chen, H.Y., Tse, T.H., Chan, F.T., Chen, T.Y., 1998. In black and white: an integrated approach to class-level testing of object-oriented programs. ACM Trans. Software Eng. Method. 7 (3), 250–295.
Chen, J., Kuo, F.C., Towey, T.Y., Su, C., Huang, R., 2017. A similarity metric for the inputs of OO programs and its application in adaptive random testing. IEEE Trans. Reliab. 66 (2), 373–402.
Chen, J., Zhu, L., Chen, T.Y., Huang, R., Towey, D., Kuo, F.C., Guo, Y., 2016. An adaptive sequence approach for oos test case prioritization. In: IEEE International Symposium on Software Reliability Engineering Workshops(ISSRE-IWPD 2016), IEEE, Canada, pp. 205–212.
Chen, T.Y., Fei-Ching, K., Dave, T., Quan, Z.Z., 2015. A revisit of three studies related to random testing. Sci. China Inf. Sci. 58 (5), 1–9.
Chen, T.Y., Kuo, F.C., Liu, H., Wong, W.E., 2013. Code coverage of adaptive random testing. IEEE Trans. Reliab. 62 (1), 226–237.
Chen, T.Y., Kuo, F.C., Merkel, R.G., Tse, T.H., 2010. Adaptive random testing: the art of test case diversity. J. Syst. Softw. 83 (1), 60–66.
Chen, T.Y., Leung, H., Mak, I.K., 2004. Adaptive random testing. In: Proceedings of the 9th Asian Computing Science Conference (ASIAN 2004), Thailand. Springer LNCS, pp. 320–329.
Chen, T.Y., Merkel, R., 2008. An upper bound on software testing effectiveness. ACM Trans. Software Eng. Method. 17 (3), 1–27.
Ciupa, I., Leitner, A., Oriol, M., Meyer, B., 2008. ARTOO:adaptive random testing for object-oriented software. In: ACM/IEEE 30th International Conference on Software Engineering (ICSE 2008), IEEE, New York, USA, pp. 71–80.
Codeforge-free open source codes forge and sharing, 2013. http://www.codeforge.com.
Codeplex-open source project hosting, 2013. http://www.codeplex.com.
Dickinson, W., Leon, D., Fodgurski, A., 2001. Finding failures by cluster analysis of execution profiles. In: 23rd IEEE International Conference on Software Engineering Proceedings (ICSE 2001), IEEE, Ontario, Canada, pp. 339–348.
Elbaum, S., Malishevsky, A.G., Rothermel, G., 2002. Test case prioritization: a family of empirical studies. IEEE Trans. Softw. Eng. 28 (2), 159–182.
Gonzalez-Sanchez, A., Piel, E., Abreu, R., Gross, H.G., Van Gemund, A.J.C., 2011. Prioritizing tests for software fault diagnosis. Softw. Pract. Exp. 41 (10), 1105–1129.
Github, where software is built, 2015. https://www.github.com.
Hamlet, R., 2002. Random testing. Encyclopedia of Software Engineering. John Wiley and Sons.
Harman, M., McMinn, P., Souza, J., Yoo, S., 2012. Search based software engineering: techniques, taxonomy, tutorial. In: Empirical Software Engineering and Verification, pp. 1–59.
Henard, C., Papadakis, M., Harman, M., Jia, Y., Traon, Y.L., 2016. Comparing white-box and black-box test prioritization. In: International Conference on Software Engineering, pp. 523–534.
Holt, N.E., Briand, L.C., Torkar, R., 2014. Empirical evaluations on the cost-effectiveness of state-based testing: an industrial case study. Inf. Softw. Technol. 56 (8), 890–910.
Hui, S.U., Zhang, Y., Yao, H., Fei, R., 2005. Object-oriented software cluster-level testing based on uml sequence diagram. Comput. Eng. 31 (24), 78–80.
Jia, Y., Harman, M., 2010. An analysis and survey of the development of mutation testing. IEEE Trans. Software Eng. 37 (5), 649–678.
Kaufmann, L., Rousseeuw, P.J., 1987. Clustering by means of medoids. In: Statistical Data Analysis Based on the L1-norm and Related Methods, North-Holland, pp. 405–416.
Ledru, Y., Petrenko, A., Boroday, S., Mandran, N., 2012. Prioritizing test cases with string distances. Automated Softw. Eng. 19 (1), 65–95.
Legunsen, O., Hariri, F., Shi, A., Lu, Y., Zhang, L., Marinov, D., 2016. An extensive study of static regression test selection in modern software evolution. In: ACM Sigsoft International Symposium on Foundations of Software Engineering(FSE 2016), ACM, Washington, USA, pp. 583–594.
Liu, H., Kuo, F.C., Towey, D., Chen, T.Y., 2014. How effectively does metamorphic testing alleviate the oracle problem? IEEE Trans. Software Eng. 40 (1), 4–22.
Luo, Q., Moran, K., Poshyvanyk, D., 2016. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In: ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2016), ACM, Washington, USA, pp. 559–570.
M. Aqilburney, S., Tariq, H., 2014. K-Means cluster analysis for image segmentation. Int. J. Comput. Appl. 96 (4), 1–8.
Microsoft visual studio, 2013, https://www.visualstudio.com.
Nie, C., Wu, H., Niu, X., Kuo, F.C., Leung, H., Colbourn, C.J., 2015. Combinatorial testing, random testing, and adaptive random testing for detecting interaction triggered failures. Inf. Softw. Technol. 62, 198–213.
Papadakis, M., Henard, C., Harman, M., Jia, Y., Traon, Y.L., 2016. Threats to the validity of mutation-based test assessment. In: International Symposium on Software Testing and Analysis, pp. 354–365.
Park, H.S., Jun, C.H., 2009. A simple and fast algorithm for k-medoids clustering. Expert Syst. Appl. 36 (2), 3336–3341.
Pezze, M., Young, M., 2004. Testing object-oriented software. In: 26th International Conference on Software Engineering Proceedings (ICSE 2004),United Kingdom. IEEE, pp. 739–740.
Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J., 2002. Prioritizing test cases for regression testing. IEEE Trans. Software Eng. 27 (10), 929–948.
Suganya, S.R., Devi, G.S., 2010. Data mining: concepts and techniques. Data Mining and Knowledge Engineering 929–948.
Software-artifact infrastructure repository, 2016. http://www.sir.unl.edu/portal/index.php.
Software-artifact infrastructure repository Sourceforge-download, develop and publish free open source software, 2013. http://www.sourceforge.net.
Team, R.D.C., 2009. Development core team, R: A language and environment for statistical computing.
Tibshirani, R., Walther, G., Hastie, T., 2001. Estimating the number of clusters in a data set via the gap statistic. J. R. Stat. Soc. 63 (2), 411–423.
Vargha, A., Delaney, H.D., 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J. Edu. Behav. Stat. 25 (2), 101–132.
Yoo, S., Harman, M., Tonella, P., Susi, A., 2009. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In: Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, USA, July, pp. 201–212.
Zhang, X., Chen, T.Y., Liu, H., 2014. An application of adaptive random sequence in test case prioritization. In: International Conference on Software Engineering and Knowledge Engineering (SEKE 2014), IEEE, Canada, pp. 126–131.
Zhang, X., Xie, X., Chen, T.Y., 2016. Test case prioritization using adaptive random sequence with category-partition-based distance. In: IEEE International Conference on Software Quality, Reliability and Security (QRS 2016), IEEE, Washington, USA, pp. 374–385.

Jinfu Chen received the B.E. degree from Nanchang Hangkong University, Nanchang, China, in 2004, and the Ph.D. degree from Huazhong University of Science and Technology, Wuhan, China, in 2009, both in computer science. He is an Associate Professor in the School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, China. His major research interests include software testing, software analysis, and trusted software.

Lili Zhu received the B.E. degree in computer science in 2014 from Jiangsu University, Zhenjiang, China, where she is currently working toward the Masters degree in the School of Computer Science and Communication Engineering. Her research interests include software testing and software analysis.

Tsong Yueh Chen received the B.Sc. and M.Phil. degrees from the University of Hong Kong, China, the M.Sc. degree and DIC from the Imperial College of Science and Technology, London, U.K., and the Ph.D. degree from the University of Melbourne, Australia. He is currently a Professor of Software Engineering in the Department of Computer Science and Software Engineering, Swinburne University of Technology, Australia. He is the originator of metamorphic testing and adaptive random testing. His current research interests include software testing, debugging, and program repair.

Dave Towey received the B.A. and M.A. degrees in computer science, linguistics, and languages from Trinity College, the University of Dublin, Ireland; the M.Ed. degree in education leadership from the University of Bristol, U.K.; and the Ph.D. degree in computer science from the University of Hong Kong, China. He is currently an Associate Professor, the Director of Teaching and Learning, and the Deputy Head of the School of Computer Science. He also serves as Deputy Director of the International Doctoral Innovation Centre (IDIC). His current research interests include software testing and technology enhanced teaching and learning. In 2016, he co-founded the ICSE International Workshop on Metamorphic Testing (ICSE MET'16).

Fei-Ching Kuo passed away of illness on October 6, 2017. She received the Bachelors of Science (Hons.) degree in computer science and the Ph.D. degree in software engineering, both from Swinburne University of Technology, Australia. She was a Senior Lecturer in the Department of Computer Science and Software Engineering, Swinburne University of Technology, Australia. She was also previously a Lecturer at the University of Wollongong, Australia. She was the Program Committee Chair for the 10th International Conference on Quality Software 2010, and was the Guest Editor of a special issue of the Journal of Systems and Software, a special issue of Software: Practice and Experience, and a special issue of the International Journal of Software Engineering and Knowledge Engineering. Her research interests included software analysis, testing, and debugging.

Rubing Huang is an Associate Professor at the Department of Software Engineering, School of Computer Science and Communication Engineering, Jiangsu University, China. He received the Ph.D. degree in Computer Science and Technology from Huazhong University of Science and Technology, China, in 2013. His current research interests include software testing and software maintenance, especially combinatorial testing, random testing, adaptive random testing, and test case prioritization. He has more than 30 publications in journals and proceedings including ICSE, IEEE Transactions on Reliability, JSS, IST, IJSEKE, SCN, COMPSAC, SEKE, and SAC. He has served as the program committee member of SEKE2014, SEKE2015, SEKE2016, SEKE2017, SAC2017, CTA2017, SAC2018, and SEKE2018.

Yuchi Guo received the B.E. degree in software engineering in 2013 from Huaiyin Institute of Technology, Huaian, China. She is currently working toward the Masters degree in the School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, China. Her research interests include software testing and software analysis.