Dynamic Test Case Prioritization for Functional Testing

Yue Wu[a], Chunhai Zhang[b]
School of Ocean University of China, Qingdao 266100, China
[a]wuyue9898@163.com, [b]1531393510@qq.com

Keywords: functionnal test, prioritization, dynamic adjustment.

Abstract. In order to improve the efficiency of the functional test, the dynamic test tase prioritizing algorithm is designed, which is used to optimize the execution sequence of the test cases. In this algorithm, 4 factors which are related to the software function are used to calculate the priority of the use case, and The highest priority algorithm instead of the traditional test case sorting algorithm was designed to get the current time has the highest priority test cases rather than all the test cases to sort, and to shorten the test time. In the execution of the use case, the remaining functional coverage is used to achieve dynamic adjustment of the priority which can gradually optimize the execution order of the test case. Finally, it is applied to the actual project testing work, and the results show that the algorithm can improve the functional test efficiency.

1. Introduction

The test case has a very high status in software testing, and the implementation of the test case determines the efficiency of the test, With the expansion of software scale, the number of test cases is increasing rapidly, however different test cases have different contribution to the completion of test objectives, in order to complete more and more important tasks as soon as possible, we need to execute test cases in a certain order, test case prioritization is designed to solve this problem, it was originally designed for optimization of regression tests, with the large number of research is carried out, the prioritization of regression test is already relatively perfect, however, the study of other tests has just begun, such as test case prioritization based on requirement[1], advanced test case prioritization for black box testing[2], multi-objective optimization based on-line adjustment strategy of test case prioritization[3], dynamic test case priority ranking method based on demand and feedback[4] , etc.. There is no the method of test case prioritization for functional testing, but we find that the above method are more or less applicable to functional testing, but we also find that they have some shortcomings. First, Factors that are referenced by the algorithm are less, so the priority of each use case that is calculated according to the algorithm is not accurate. Second, the dynamic adjustment of the prioritization is based on the feedback information of the execution, so we need to keep collecting information during the execution, the complexity of its implementation is significantly higher.

Aiming at the above problems, this paper proposes a dynamic test tase prioritizing algorithm. The algorithm can not only be very suitable for functional testing, improve the efficiency of functional testing, but also has the following advantages: on the one hand, the algorithm covers four influencing factors, so that the calculation results of test case priority is as accurate as possible, on the other hand, the dynamic adjustment is implemented by the use of remaining functional coverage, Instead of using the execution feedback information of the use case, which not only ensures the timeliness of the test case priority, but also reduces the execution complexity and shortens the test time.

2. Algorithm Introduction

Before the test begins, we need to calculate the initial priority of each test case based on the actual project by calculating the initial value of each priority factor.

2.1 Priority Influencing Factors and Their Initial Value Calculation Method

This paper introduces the four factors of functional coverage, functional importance, functional modifying rate and implementation complexity by analyzing and summarizing the results of predecessors, and designes the corresponding calculation method of initial value .Here we introduce them one by one:

2.1.1 Functional Coverage

It is used to measure how much of the test case covers the function. We can shorten the test time by calculating the functional coverage for the test case and preferentially executing the test case with a higher function coverage. Frist, we can get the coverage relation of test case U and function F by analyzing the design information of the test case, and then construct it as a functional coverage matrix Δ(U, F), the matrix element is defined as $\delta(u_i, f_j)=1$, the use case u_i covers the function f_j; $\delta(u_i, f_j)=0$, the use case u_i does not cover the function f_j, next, we need to count the total number of functions, that is denoted as $FC_{u_i|t_0}$, covered by each test case u_i, where t_0 indicates the time when the test has not yet started.The functional coverage matrix is very important and will be used to the calculation of functional importance and functional modifying rate and implementation complexity of use case. Finally, the calculation formula of functional coverage $PFC_{u_i|t_0}$ is given as follows:

$$PFC_{u_i|t_0}=\frac{FC_{u_i|t_0}}{COUNT(F_n)} \tag{1}$$

Where $COUNT(F_n)$ represents the total number of initial test requirements, $COUNT(F_n)=F_1+F_2+\ldots\ldots+F_n$.

2.1.2 Functional Importance

It is used to measure the importance of the function. Experience shows that, It is effective to improve the test efficiency that Preferentially testing and verifying the relatively important function for the user. First, we need to confirm the functional importance through the user, and quantify them through the use of 1, 3, 5, 7, 9 and other 10 or less integer values, the correspondence between the nth functional importance FI_n and the value is defined as follows:

(1) $FI_n=9$, the function is extremely important.
(2) $FI_n=7$, the function is very important.
(3) $FI_n=5$, the function is clearly important.
(4) $FI_n=3$, the function is slightly important.
(5) $FI_n=1$, the function is less important.

If the function importance is between the above judgments, can also be used 2, 4, 6, 8 said, the greater the value, the greater the function importance.

The function importance is defined as the ratio of the value of the function importance to the sum of all values, for a test case, its value of the function importance, that is recorded as $FI_{u_i|t_0}$, should be the sum of the value of all function importance it covers. We need to get all the functions covered by the test case u_i according to Δ(U, F), and obtain values corresponding to these functions, and then $FI_{u_i|t_0}$ will be got by accumulating these values. Marking the total priority of the test case coverage requirement as $PFI_{u_i|t_0}$, It is calculated as:

$$PFI_{u_i|t_0}=\frac{FI_{u_i|t_0}}{COUNT(FI_n)} \tag{2}$$

$COUNT(FI_n)=FI_1+FI_2+\ldots\ldots+FI_n$, The greater the $PFI_{u_i|t_0}$, the greater the priority of the test case.

2.1.3 Functional Modifying Rate

It is used to measure the number of functional modification. Before the software is completed, changes in requirement are common, then modifying the software function has become very common, experience shows, modification may produce more defects, therefore, we can achieve the purpose of detecting more defects in a shorter time by preferentially executing a test case with a large functional modifying rate.

The functional modifying rate is defined as the ratio of the number of modification to this function and the sum of the total number. The changing number of the nth function is defined as $FM_{n|t_0}$, $FM_{n|t_0}$ can be obtained in the demand document, for a test case, the number of modifications should be the sum of the modifing number of all functions it covers, which can be denoted $FM_{u_i|t_0}$.we can also get all the functions covered by the test case u_i according to $\Delta(U, F)$, and then accumulate $RM_{u_i|t_0}$.Marking the functional modifying rate of the test case as $PFM_{u_i|t_0}$, It is calculated as:

$$PFM_{u_i|t_0}=\frac{FM_{u_i|t_0}}{COUNT(FM_n)} \tag{3}$$

$COUNT(FM_n)=FM_1+FM_2+\ldots\ldots+FM_n$, The greater the $PFM_{u_i|t_0}$, the greater the priority of the test case.

2.1.4 Implementation complexity

It is used to measure the complexity of the functional implementation. Experience shows, the more complex parts of the software may have more defects. Implementation complexity is given by the developer, and also quantify them through the use of 1, 3, 5, 7, 9 and other 10 or less integer values, the correspondence between the implementation complexity of the nth function IC_n and the value is defined as follows:

(1) $IC_n=9$, the function is extremely difficult to implement.
(2) $IC_n=7$, the function is very difficult to implement.
(3) $IC_n=5$, the function is clearly difficult to implement.
(4) $IC_n=3$, the function is slightly difficult to implement.
(5) $IC_n=1$, the function is easy to implement.

If the implementation complexity of the function is between the above judgments, can also be used 2, 4, 6, 8 said.

For a test case, its implementation complexity value should be the sum of the implementation complexity values for all functions it covers, which can be denoted as $IC_{u_i|t_0}$, getting all the functions covered by the test case u_i according to $\Delta(U, F)$, and obtaining the implementation complexity values corresponding to these functions, and then accumulating these values to get $IC_{u_i|t_0}$.Marking the total implementation complexity of the test case as $PIC_{u_i|t_0}$, It is calculated as:

$$PIC_{u_i|t_0}=\frac{IC_{u_i|t_0}}{COUNT(IC_n)} \tag{4}$$

$COUNT(IC_n)=IC_1+IC_2+\ldots\ldots+IC_n$.The greater the $PIC_{u_i|t_0}$, the greater the priority of the test case.

2.2 Priority Calculation Method

Due to the impact of these four factors on the priorities of the test case is not same, we need to assign them the corresponding weighting factors: α, β, γ, δ, the weighting factor can be determined according to the specific items and the environment (need to meet $\alpha + \beta + \gamma + \delta= 1$).The initial priority of the test case, is denoted as $P_{u_i|t_0}$, can be calculated by weighting them, the formula is as follows:

$$P_{u_i|t_0}=PFC_{u_i|t_0}*\alpha+PFI_{u_i|t_0}*\beta+PFM_{u_i|t_0}*\gamma+PIC_{u_i|t_0}*\delta \tag{5}$$

Through the introduction of the above factors, we know that the larger the value of $P_{u_i|t_0}$, the greater the priority of the test case.

2.3 Priority Ranking Algorithm Design

Taking into account, we will make frequent dynamic adjustments to the order in which the use cases are executed, the highest priority algorithm instead of the traditional test case sorting algorithm was designed to get the current time has the highest priority test cases rather than all the test cases to sort. And then we continue to adjust the priority of test cases by the dynamic adjustment algorithm In the course of execution, and cycle the above steps until the test is completed, which can shorten the test time and improve test efficiency. The priority ranking algorithm designed in this paper includes two aspects: the highest priority algorithm and the dynamic adjustment algorithm, they are described below.

2.3.1 The highest priority algorithm

It is used to obtain the highest priority test case after calculating the priority of each use case.

```
Function HighestP($P_{u_i|t_i}$)()
Input:$P_{u_i|t_i}$ //input the priority of all test cases at time $t_i$
Output:$u_i$ //output the highest priority test case at time $t_i$
Begin
hig <- $P_{u_1|t_0}$ //assign the first test case priority to hig
For i=1 to n { //traverse the priority of all test cases
    If $P_{u_{i+1}}$>$P_{u_i}$ Then { //determine the priority of a test case and the previous one
        hig <- $P_{u_{i+1}}$ //If the latter is large, it will be assigned to Maxr, otherwise hig will not change
        UI[ ] <- i //Store i into an array
    /}
/}
Return UI[n-1] //output the last element of the array, the corresponding test case is the use case with the highest priority
End
```

2.3.2 Dynamic Adjustment Algorithm

Choosing the remainder functional coverage as the basis for the dynamic adjustment algorithm. We achieve the goal of detecting more features at an earlier time through the use of the test cases that cover more of remainder functions, rather than the inherent functional coverage.

The remainder functional coverage is used to measure the number that the test case covered the function that have not yet been covered. After the i-th test cases is executed(recorded as t_j time), the remainder function can be got through subtracting the function that has been covered with the total function, the matrix Δ (U_{t_j}, R_{t_j}) of the remainder test cases and the remainder functions is constructed, the definition of the matrix element is the same as the functional coverage matrix, except that the rows of the matrix are changed from all use cases to the remainder cases at this time; The columns are replaced by all the functions for the remainder functions at this tim, After each test case is executed, the rows and columns of the matrix are transformed always. We can apply the calculation of the functional coverage to calculate the remainder functional coverage. The specific formula is as follows:

$$PFC_{u_i|t_j}=\frac{FC_{u_i|t_j}}{COUNT(F_n)} \tag{6}$$

The functional coverage, the functional importance, the functional modifying rate and the implementation complexity of the test case are calculated on the basis of the functional coverage matrix, while the matrix is constant changing, which causes their values to follow constant changes. In general, when a test case is executed, the priority of the remainder test cases should be recalculated, in other words, the priority of the test cases is dynamically adjusted in real time to ensure that the test cases, are executed each time, are most suitable for the current test environment. The priority calculation formula for the remainder test cases is as follows:

$$P_{u_i|t_i}=PFC_{u_i|t_i}*\alpha+PFI_{u_i|t_i}*\beta+PFM_{u_i|t_i}*\gamma+PIC_{u_i|t_i}*\delta \tag{7}$$

The dynamic adjustment algorithm is as follows:

```
Function Adjp($u_i$)()
Input:$u_i$ //enter all the test cases at time $t_i$
Begin
For i=1 to n {
    Calculate $P_{u_i|t_0}$ //Calculates the initial priority of all test cases
/}
$U_{t_0}$ <- HighestP($P_{u_i|t_0}$)() //call the highest priority algorithm to get the first test case with the
```

```
highest priority $U_{t_0}$
    Run($U_{t_0}$) //run test cases $U_{t0}$
    For j=1 to n {
        Update $\Delta(U_{t_j}, P_{t_j})$ //update the remaining demand coverage matrix
        For i=1 to n {
            Calculate $P_{u_i|t_j}$ //calculate the priority of the remaining test cases at time $t_i$
        /}
        U <- HighestP($P_{u_i|t_j}$)() //call the highest priority algorithm to get the highest priority test
    case at this time U
        Run(U) //run test cases U
    /}
    End
```

3. Experiment Analysis

The dynamic test case prioritization for functional testing is applied to the testing work of the DianBaoWang, that is a website for customers to assessment and pawn their own items (cars, digital, diamonds, others, etc.).we chooses the enterprise module to carry on the experiment, the module has a total of 108 test requirements, designs a total of 131 test cases, and found a total of 95 bugs. We perform the test cases in two execution order, first, initial creation order and second, the order of The dynamic test case prioritization for functional testing, APFD metrics were used to verify the test efficiency of each sort.

The comparison of the execution of two orders is as follows:

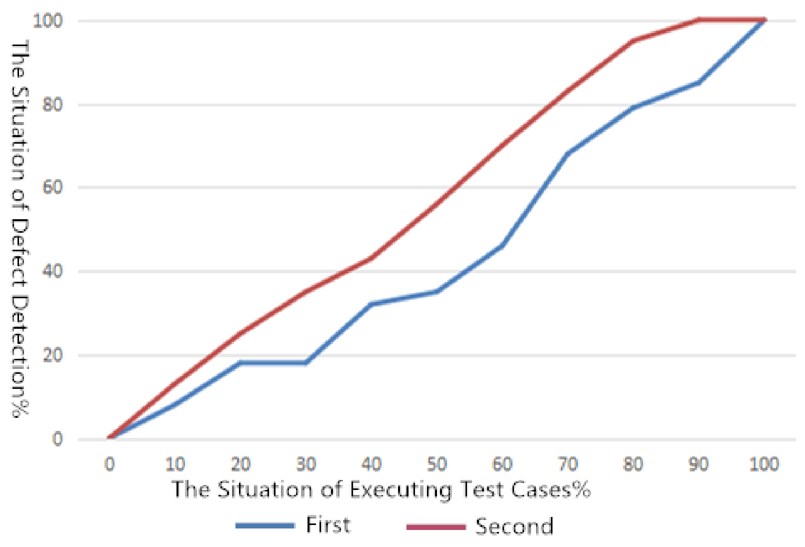

Fig. 1 The figure of comparison

As can be seen from Fig. 1, the second defect detection is significantly better than the first two, the rate of the detection is relatively stable, and the number of defects, is detected in the pre-test case, is higher than that of the first. the time of that defects are completely detected is before the first, the APFD value of the first is 58.4%, and two is 65.7% that higher than the first 7.3%.Experimental results show: The dynamic test case prioritization for functional testing has a higher defect detection capability, and can significantly improve the test efficiency.

4. Conclusion

The dynamic test case prioritization for functional test not only makes up the gaps of the prioritization in the functional test, which helps to improve the efficiency of functional testing, but also takes into account the various factors that affect the priority of the test cases, during the execution of the test case, the execution order of the test cases is continually optimized by the dynamic adjustment strategy, which helps to have a better test effect. The next step, our work mainly for the following, (1) We will continue to seek priority factors to further improve the accuracy of the test case priority, (2) optimizing the calculation of each influencing factor, (3) optimizing dynamic adjustment strategies, (4) applying it to more projects, and improving its usefulness.

References

[1] Yang GuangHua, Bao Yang, Li DongHong, et al. test case prioritization based on requirement [J]. Computer Engineering and Design, 2011, 32(8): 2724-2728.
[2] Dai Ruxin, Gu Chunhua. Advanced test case prioritization for black box testing [J]. Computer Engineering and Design, 2010, 31(20): 4343-4346.
[3] Zhang Na, Yao Lan, Bao Xiaoan, et al. multi-objective optimization based on-line adjustment strategy of test case prioritization [J]. Journal of Software, 2015, 26(10): 2451-2464.
[4] Xie Xiaozhu, Xiao Lei, Cui Jianfeng, et al. dynamic test case priority ranking method based on demand and feedback [J]. Journal of Xiamen Institute of Technology, 2015(5): 70-74.