QTEP: Quality-Aware Test Case Prioritization

Song Wang, Jaechang Nam, Lin Tan
{song.wang,jc.nam,lintan}@uwaterloo.ca
Electrical and Computer Engineering, University of Waterloo
Waterloo, ON, Canada

ABSTRACT

Test case prioritization (TCP) is a practical activity in software testing for exposing faults earlier. Researchers have proposed many TCP techniques to reorder test cases. Among them, coverage-based TCPs have been widely investigated. Specifically, coverage-based TCP approaches leverage coverage information between source code and test cases, i.e., static code coverage and dynamic code coverage, to schedule test cases. Existing coverage-based TCP techniques mainly focus on maximizing coverage while often do not consider the likely distribution of faults in source code. However, software faults are not often equally distributed in source code, e.g., around 80% faults are located in about 20% source code. Intuitively, test cases that cover the faulty source code should have higher priorities, since they are more likely to find faults.

In this paper, we present a quality-aware test case prioritization technique, QTEP, to address the limitation of existing coverage-based TCP algorithms. In QTEP, we leverage code inspection techniques, i.e., a typical statistic defect prediction model and a typical static bug finder, to detect fault-prone source code and then adapt existing coverage-based TCP algorithms by considering the weighted source code in terms of fault-proneness. Our evaluation with 16 variant QTEP techniques on 33 different versions of 7 open source Java projects shows that QTEP could improve existing coverage-based TCP techniques for both regression and new test cases. Specifically, the improvement of the best variant of QTEP for regression test cases could be up to 15.0% and on average 7.6%, and for all test cases (both regression and new test cases), the improvement could be up to 10.0% and on average 5.0%.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Automated static analysis**;

KEYWORDS

Test case prioritization, defect prediction, static bug finder

ACM Reference format:
Song Wang, Jaechang Nam, Lin Tan. 2017. QTEP: Quality-Aware Test Case Prioritization. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 12 pages.
https://doi.org/10.1145/3106237.3106258

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106258

1 INTRODUCTION

Modern software constantly evolves as developers make source code changes such as fixing bugs, adding new features, refactoring existing code, etc. To ensure that the changes do not introduce new bugs, regression testing is commonly conducted against existing functionalities.

However, regression testing can be expensive. Especially for large projects, the regression testing could consume 80% of the overall testing budgets and require weeks to run all test suites [14, 24, 64]. Intuitively, test cases that could reveal bugs should be run earlier so that the developers could have more time to fix the revealed bugs and speed up the system delivery.

Along this line, test case prioritization (TCP) has been proposed and intensively studied for regression testing [23, 44, 50, 52, 53, 64, 66, 75, 82, 85]. TCP techniques reorder test cases to maximize a certain objective function, typically exploring faults earlier [64]. TCPs also have been widely adopted in industry. For instance, Microsoft has deployed systems that support TCP such as Echelon [70] and Microsoft Dynamics AX [13]. Researchers have applied TCP techniques on projects from Google [24, 84] and Salesforce.com [12] and have shown that TCPs could significantly improve the efficiency of regression testing.

Many TCP techniques have been proposed such as coverage-based TCPs [17, 36, 37, 44, 50, 64, 82, 88], requirement-based TCPs [7, 33], and change-based TCPs [35, 66], etc. Coverage-based TCPs have been shown to outperform other TCPs in terms of revealing faults [32, 47, 48]. A typical coverage-based TCP technique leverages coverage information between source code and test cases, i.e., static code coverage (static call graph from current version) and dynamic code coverage (dynamic call graph from the last execution), to schedule test cases by maximizing code coverage with different coverage criteria. Coverage-based TCPs assign higher priorities to test cases that have higher dynamic or static code coverages.

Existing TCP techniques often do not take the likely distribution of faults in source code into consideration. In other words, they assume that faults in program source code are equally distributed. However, as reported in existing work [60, 61, 72], the fault distribution in source code is often unbalanced, i.e., around 80% faults are located in about 20% source code [60]. Intuitively, test cases that cover the more fault-prone code are more likely to reveal bugs so that they should be run with a higher priority.

The goal of this study is to propose a quality-aware TCP technique, QTEP, that addresses the above limitation of existing TCPs. We evaluate the quality of source code in terms of fault-proneness

and then we further use the quality information to prioritize test cases.

To achieve the goal, QTEP gives more weight to fault-prone source code so test cases that cover the fault-prone code have a higher priority to be executed. We identify fault-prone source code in a software project by using two code inspection approaches, i.e., static bug finders and defect prediction models, which are two widely studied approaches in code inspection research to help developers find bugs [9, 42, 62]. In this study, we leverage these two code inspection techniques to improve existing coverage-based TCPs.

In addition, we apply QTEP to both regression and new test cases. Most existing TCP techniques only focus on prioritizing regression test cases [17, 36, 37, 44, 50, 64, 82, 88]. However, in real-world testing practice, the test suite for a modified software system often consists of: (1) existing test cases, i.e., regression test cases, which are designed to verify whether the existing functionalities still perform correctly after changed, and (2) new test cases, which are added to test the modification. During software evolution, these two types of test cases are essential for testing the modified software and detecting bugs [47]. Thus, we consider both the two types of test cases in this study.

This paper makes the following contributions:

- We propose a novel quality-aware TCP technique, QTEP, which leverages two dominant code inspection techniques, i.e., a typical statistic defect prediction model and a typical static bug finder, to weight source code in terms of fault-proneness. Then, QTEP adapts existing coverage-based TCPs by considering the fault-proneness of source code.
- We conduct an extensive study to compare the performance of QTEP with existing coverage-based TCPs for both regression test cases and new test cases at the class- and method-level granularities.
- We present a rigorous empirical evaluation using 33 versions of 7 open source Java projects and explore 16 different variants of the proposed QTEP. Results show that QTEP could improve existing coverage-based TCP techniques for both regression and new test cases. Specifically, the improvement of the best variant of QTEP for regression test cases could be up to 15.0% and on average 7.6%, and for all test cases (both regression and new test cases), the improvement could be up to 10.0% and on average 5.0%.

The rest of this paper are organized as follows. Section 2 describes the basic background. Section 3 shows the design of QTEP. Section 4 shows the setup of our experiments. Section 5 presents the results of our research questions. Section 6 discusses our results and the threats to the validity of this work. Section 7 surveys the related work. Finally, we summarize this paper in Section 8.

2 BACKGROUND

2.1 Test Case Prioritization

A typical TCP technique reorders the execution sequence of test cases based on a certain objective, e.g., fault-detection rate [64]. Specifically, TCP can be formally defined as follows: given a test suite T and the set of its all possible permutations PT, TCP techniques aim to find a permutation $P' \in PT$ that $(\forall P'')\ (P'' \in PT)\ (P'' \neq P'),\ f(P') \geq f(P'')$, where f is the objective function.

Most existing TCPs leverage coverage information, e.g., dynamic code coverage (dynamic call graph) from the last run of test cases [37, 65], static code coverage (static call graph) from static code analysis [17, 36, 50, 69, 74]. The commonly used coverage criteria include statement, method, and branch coverages. In this work, we choose to examine statement and method coverages, since previous work has shown that statement and method coverages are more effective than other coverage criteria [47, 48, 66].

For coverage-based TCP techniques, there are two widely used prioritization strategies, i.e., total strategy and additional strategy [44, 64, 82]. The total coverage strategy schedules the execution order of test cases based on the total number of statements or methods covered by these test cases. Whereas, the additional coverage strategy reorders the execution sequence of test cases based on the number of statements or methods that are not covered by already ordered test cases but covered by the unordered test cases.

In this study, we validate whether performances of the coverage-based TCP techniques could be improved by considering the results of code inspection techniques. The reason we focus on the coverage-based TCP techniques is that they have been widely explored and outperformed most of the other TCP techniques [32, 47, 48]. In addition, we evaluate the state-of-the-art coverage-based TCP techniques for both regression and new test cases.

2.2 Code Inspection Techniques

Software static code analysis techniques and software defect prediction models are two lines of work to help detect software bugs. Static bug finders, e.g., FindBugs [34], PMD, and Jlint, leverage well-defined bug patterns and report likely bug locations in source code. Defect prediction models build machine learning classifiers based on various metrics [18, 29, 30, 49, 78] and predict where defects are likely to occur in the source code [42]. Both techniques are widely used for detecting fault-prone code. Rahman et al. [62] found that these two techniques complement each other to find different bugs.

In this work, we examine whether the two code inspection techniques could help improve existing TCP techniques.

2.2.1 Static Bug Finders. Static bug finders (BF) detect specific kinds of coding errors such as buffer overflow, race conditions, and memory allocation errors, via programming patterns [15, 25, 26, 39, 45, 58, 62, 71, 77, 79]. Static bug finders mainly include informal heuristic pattern-matching approaches and formal verification techniques. In practice, formal verification approaches are not scalable, so most static bug finders are pattern-based.

Typical pattern-based bug finders include FindBugs [34], PMD, and Jlint, which have been widely used to detect real-world software bugs. Previous work [62] has shown that FindBugs is more effective than PMD and Jlint in terms of detection precision. Since one focus of this study is to explore the feasibility of leveraging the results of static bug finders to improve the efficiency of existing TCP techniques. Thus, we only use the representative static bug finder, i.e., FindBugs, in our experiments.

2.2.2 Software Defect Prediction. Software defect prediction techniques (DP) leverage various software metrics to build machine learning models to predict unknown defects in the source code [31, 38, 51, 56, 57, 72, 78, 80, 87, 91]. Based on the prediction results, software quality assurance teams can focus on the most defective parts of their projects in advance.

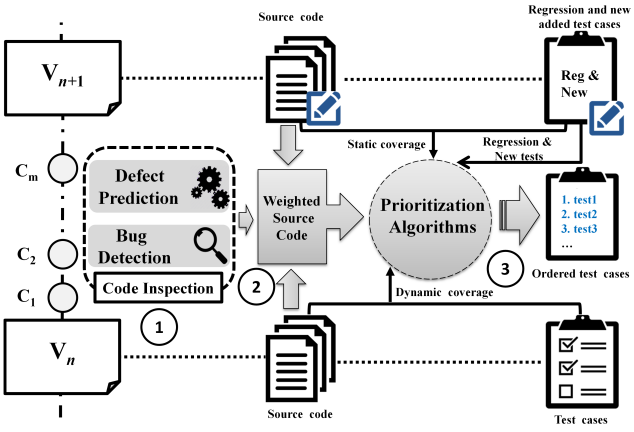


Figure 1: Overview of the proposed QTEP. V_n and V_{n+1} are two versions. C_1 to C_m are consecutive commits between these two versions that introduce changes to the source code and test cases.

Typically, defect prediction models could be categorized as supervised or unsupervised models. Most of existing defect prediction models are supervised [31, 38, 51, 56, 72, 78, 80, 91]. These models leverage past defects from software historical data to build machine learning classifiers and then use the classifiers to predict future bugs. However, not all projects have enough defect data to build a defect prediction model, so unsupervised models [57, 87] are also proposed based on the characteristics of defect prediction metrics.

To directly compare QTEP to existing TCP techniques, we reuse 33 versions from 7 open source Java projects from previous TCP studies [21, 50, 66]. Some of these projects do not have well-maintained past defects. This means we do not have enough defect data to build supervised models. Thus, we only examine unsupervised defect prediction models in our experiments.

Note that our goal of this study is not to find the best defect prediction metrics, so we only build defect prediction models with 21 widely used code metrics, e.g., lines of code [46], code complexity metrics [31, 46, 51], and object-oriented metrics [22], etc. We use the *Understand* [3] to collect these code metrics. The work from Zhang et al. [86] contains the full list of these metrics, which are described in their Table III.

3 APPROACH

Figure 1 illustrates that our approach consists of three steps: (1) leveraging code inspection techniques to detect fault-prone source code (Section 3.1), (2) weighting source code based on results from different code inspection approaches (Section 3.2), and (3) adapting existing TCP techniques and evaluating the results (Section 3.3).

3.1 Fault-prone Code Inspection

Code Inspection with FindBugs. Given a subject project, we directly perform FindBugs on its source code (without test code) to detect potential bugs. Since FindBugs only outputs detection results at line level, we aggregate the results into method level and class level to meet different TCP granularities. Moreover, we consider all detection results grouped by various categories. FindBugs groups detected fault-prone code instances into categories such as correctness, vulnerability, malicious code, security, multi-threaded correctness, performance, bad practice, and dodgy code. The first six types are likely to be real bugs, the last two types are refactoring issues. Since bad practice and dodgy code could also fail test cases, in this work, we use all these reported fault-prone code instances as seeds to weight source code.

Code Inspection with Defect Prediction Model. Similar to static bug finders, defect prediction models also predict potential faults in the source code snapshot. As we described in Section 2.2.2, we build unsupervised defect prediction models in this work. For the unsupervised defect prediction models, we use the state-of-the-art technique, i.e., CLAMI [57], which achieves comparable performance to supervised models and has been open-sourced. To consider different test case prioritization strategies and scenarios, we build CLAMI models at both method and class levels and CLAMI directly outputs the lists of predicted bugs at both levels.

In this study, we use all the reported warnings from FindBugs (or CLAMI) to initialize QTEP without filtering out any of them.

Algorithm 1 Weighting source code units

Input: Inspection results at class level C_{Buggy} and at method level M_{Buggy}. The sets of all methods M and all statements S to be weighted. Parameters $weight\_base$, $weight\_c$, and $weight\_m$.

Output: Weighted method set $MWeighted$ and statement set $SWeighted$.

```
1: //Initialization, set default values for examined code units.
2: for each unweighted method MWeighted_i in MWeighted do
3:     MWeighted_i = weight_base;
4: end for
5: for each unweighted statement SWeighted_i in SWeighted do
6:     SWeighted_i = weight_base;
7: end for
8: //Weight methods
9: for each unweighted method M_i in M do
10:    if M_i in M_Buggy then
11:        MWeighted_i += weight_m;
12:    end if
13:    if C_Buggy contains M_i then
14:        MWeighted_i += weight_c;
15:    end if
16: end for
17: //Weight statements
18: for each unweighted statement S_i in S do
19:    if M_Buggy contains S_i then
20:        SWeighted_i += weight_m;
21:    end if
22:    if C_Buggy contains S_i then
23:        SWeighted_i += weight_c;
24:    end if
25: end for
```

3.2 Weighting Source Code Units

We leverage the detection results from the two code inspection approaches to weight the fault-prone source code units. A code unit could be a statement, a branch, a method, or a class, which depends on different test case prioritization strategies. In this work, we focus on weighting statement-level and method-level code units since we use statement-level and method-level coverage criteria to examine coverage-based TCPs (Section 2.1).

Algorithm 1 shows how to weight statement-level and method-level code units by using detection results from the code inspection approaches. Note that, we use the detected buggy classes and methods by the two code inspection approaches. Initially, the algorithm assigns a default weight to all code units (i.e., all statements and methods). Then, given a code unit, if the class or the method that contains this code unit is detected as buggy, the weight of this code unit will be calculated by accumulating the weights of the buggy

class or the buggy method. Otherwise, if the class or the method that contains this code unit is identified as clean, its weight will not be updated. Code units that are not covered by any buggy class or method will be assigned the default weight.

Parameters: In the above algorithm, there are three parameters, i.e., *weight_base*, *weight_c*, and *weight_m* that could affect the effectiveness of the proposed QTEP. We describe the setup, tuning, and impact of these three parameters in Section 4.6.

- *weight_base* is the base weight for all code units, i.e., the default weight for initializing the weights of all code units.
- *weight_c* is the weight for detected buggy classes by code inspection techniques.
- *weight_m* is the weight for detected buggy methods by code inspection techniques.

For CLAMI, we use the class-level prediction results as seeds to weight code units using *weight_c*, and use the method-level prediction results as seeds to weight code units using *weight_m*. FindBugs outputs detection results at line level, with the line-level detection results, one can assign the reported buggy lines a different weight, and further accumulate the lines to weight the involved statements and methods. While, to make the calculation of FindBugs consistent with defect prediction models, in this work, similar to CLAMI, we use the classes and methods that contain the detected buggy lines as seeds to weight code units using *weight_c* and *weight_m* respectively.

3.3 Quality-Aware Test Case Prioritization

After weighting all the source code units of a project, we then adapt existing coverage-based TCP techniques using these weighted code units. Comparing to existing coverage-based TCPs, QTEP leverages the quality-aware coverage information of test cases. In this section, we show how to calculate the quality-aware statement and method coverages of a test case.

A project P has m method-level code units, i.e., $\{mc_1, mc_2, ..., mc_m\}$, and s statement-level code units, i.e., $\{sc_1, sc_2, ..., sc_s\}$. Its test suite T consists of n test cases, i.e., $\{t_1, t_2, ..., t_n\}$. $MWeighted$ ($\{mw_1, mw_2, ..., mw_m\}$) and $SWeighted$ ($\{sw_1, sw_2, ..., sw_s\}$) are the weight sets for the method-level and the statement-level code units respectively.

Given a test case t_i ($1 \leq i \leq n$), we use $QMCoverage[t_i]$ and $QSCoverage[t_i]$ to denote its quality-aware method coverage and statement coverage respectively.

$$QMCoverage[t_i] = \sum_{j=1}^{m} cover(t_i, mc_j) * mw_j \quad (1)$$

$$QSCoverage[t_i] = \sum_{j=1}^{s} cover(t_i, sc_j) * sw_j \quad (2)$$

where, $cover(t_i, mc_j)$ or $cover(t_i, sc_j)$ is 1, if test case t_i covers code unit mc_j or sc_j, otherwise 0. mw_j and sw_j are the weights for method-level code unit mc_j and statement-level code unit sc_j respectively.

Note that, to calculate the quality-aware coverages for test cases, one could leverage different coverage information (i.e., dynamic coverage and static coverage information) and different code inspection techniques (i.e., FindBugs and CLAMI). With the quality-aware coverages (i.e., *QMCoverage* and *QSCoverage*) of each test case, QTEP further prioritizes test cases with different prioritization strategies (i.e., total and additional).

Table 1: Experimental subject programs. VPair denotes a version pair. RTC, RTM, and RF are the number of regression test classes, regression test methods, and regression faults respectively. NTC, NTM, and NF are the number of new test classes, new test methods, and mutation faults for new test cases respectively.

No.	Project	VPair	#RTC	#RTM	#RF	#NTC	#NTM	#NF
P_1	Time&Money	3.0-4.0	15	143	1	7	32	1*100
P_2	Time&Money	4.0-5.0	16	159	1	8	24	1*100
P_3	Mime4J	0.50-0.60	24	120	3	21	139	3*100
P_4	Mime4J	0.61-0.68	57	348	4	6	72	4*100
P_5	Jaxen	1.0b7-1.0b9	12	24	3	0	0	-
P_6	Jaxen	1.1b6-1.1b7	41	243	1	28	250	1*100
P_7	Jaxen	1.1b9-1.1b11	69	645	1	7	29	1*100
P_8	Xml-Security	1.0-1.1	15	91	2	3	29	2*100
P_9	XStream	1.20-1.21	115	637	1	8	38	1*100
P_{10}	XStream	1.21-1.22	124	698	2	7	58	2*100
P_{11}	XStream	1.22-1.30	133	768	11	19	134	11*100
P_{12}	XStream	1.30-1.31	150	885	3	9	76	3*100
P_{13}	XStream	1.31-1.40	140	924	7	18	180	7*100
P_{14}	XStream	1.41-1.42	157	1,200	5	3	23	5*100
P_{15}	Commons-Lang	3.02-3.03	83	1,698	1	7	122	1*100
P_{16}	Commons-Lang	3.03-3.04	83	1,703	2	13	119	2*100
P_{17}	Joda-Time	0.90-0.95	10	219	2	1	43	2*100
P_{18}	Joda-Time	0.98-0.99	71	1,932	2	9	211	2*100
P_{19}	Joda-Time	1.10-1.20	90	2,420	1	3	415	1*100
P_{20}	Joda-Time	1.20-1.30	93	2,516	3	11	532	3*100

4 EXPERIMENTAL SETUP

4.1 Research Questions

We answer the following research questions to evaluate the performance of the proposed QTEP.

RQ1. Is QTEP more effective than the state-of-the-art coverage-based TCPs for regression test cases only?

RQ2. Is QTEP more effective than the state-of-the-art coverage-based TCPs for all test cases (both regression and new test cases)?

RQ3. How effective are the variants of QTEP combined with static bug finders versus defect prediction models in terms of improving existing TCP techniques?

RQ4. How effective are static bug finders versus defect prediction models in identifying buggy code units defined by test cases for TCP?

In RQ1 and RQ2, we explore the effectiveness of QTEP for regression test cases and all test cases respectively. In RQ3, we aim to understand the performance of the two different types (BF-based and DP-based) of QTEP variants. In RQ4, we investigate the performance of static bug finders and defect prediction models in terms of identifying buggy code units defined by test cases for TCP.

4.2 Supporting Tools

In this study, we focus on coverage-based TCPs, which require both dynamic and static code coverage information of test cases. To collect dynamic code coverage, following existing work [47, 48, 66], we use the ASM bytecode manipulation and analysis framework under FaultTracer tool [89] to collect the dynamic code coverage information for test cases. To collect static code coverage, following existing work [48, 50], we use the WALA framework [4] to collect the static call graphs for the test cases, and traverse the call graphs to obtain the involved methods and statements for each test method and test class.

Table 2: The experimental scenarios for TCPs in this work.

Test Granularities	Test Case Types
Method (M)	Regression test cases (R)
Class (C)	Regression+ New test cases (RN)

All the test prioritization techniques have been implemented in Java and all the experiments were carried out on a 4.0GHz i5-2400 desktop with 6GB of memory.

4.3 Subject Systems, Test Cases, and Faults

To facilitate the replication and verification of our experiments, we choose 33 versions from 7 open source Java projects, which are widely utilized as benchmarks to address real-world test case prioritization problem [21, 50, 66]. Table 1 lists all the projects and the detailed statistical information. The sizes of these systems vary from 5.7K LOC (Time&Money) to 114.1K LOC (Joda-Time).

For regression test cases, following existing work [50, 66], for each listed version-pair, we use the real-world regression faults for regression test cases. Each version-pair has at least one real-world regression fault, which will crash at least one regression test case on the later version. For example, there are 11 regression faults (#RF) in the project P_{11} in Table 1.

Since not all benchmark projects have faults for new test cases, following existing work [5, 20, 32, 40, 47], we use mutation faults when considering the new test cases. We generate mutation faults using a set of carefully selected mutation operators [6], e.g., logical, arithmetic, statement deletion, etc. Specifically, we use the *Major* mutation tool [1] to generate these mutation faults for new test cases. Note that not all generated mutation faults can be revealed by test cases, thus we use a subset of detected faults obtained by further running *Major* with all test cases. For each project, we randomly select m mutation faults killed by new test cases only. We set m to be equal to the number of regression faults to simulate the real-world testing scenario. To mitigate the randomness, we repeat this process 100 times. For instance, we randomly select 11 mutation faults and repeat this 100 times as 11*100 (#NF of P_{11} in Table 1). Thus, we have 100 fault version-pairs for each of experimental subjects when considering new test cases.

4.4 Evaluation Measure

We use the Average Percentage Fault Detected (APFD) [65], a widely used metric for evaluating the performance of TCP techniques. APFD measures the average percentage of faults detected over the life of a test suite, and is defined by the following formula:

$$APFD = 1 - \frac{\sum_{i=1}^{num_f} TF_i}{num_t \times num_f} + \frac{1}{2 \times num_t} \quad (3)$$

where, num_t denotes the total number of test cases, num_f denotes the total number of detected faults, and TF_i ($1 \leq i \leq num_f$) denotes the smallest number of test cases in sequence that need to be run in order to expose the fault i. APFD values range from 0 to 1. For any given test suite, its num_t and num_f are fixed. The higher APFD value signals that the average value of TF_i is lower and thus represents a higher fault-detection rate.

4.5 Experimental Scenarios

Table 2 shows the TCP scenario options in our experiments. By combining these options, **four different TCP scenarios** can be defined. We first conduct TCP at two different granularity levels, i.e., method (M) and class (C). In addition, following existing work [47], we also conduct TCP for (1) regression test cases only (R), and (2) all test cases (regression and newly added test cases, RN). Running regression test cases only or running all the test cases are two practical testing activities during software evolution [55]. Based on the combinations of these settings, the four scenarios are defined as follows: *regression test method* (M-R), *regression test class* (C-R), *all test method* (*regression and newly added*) (M-RN), and *all test class* (*regression and newly added*) (C-RN). In Section 5, we report APFD values for these four scenarios.

Table 3: The experimental independent variables of QTEP.

IV1: Coverage Techniques	IV2: Coverage Criteria	IV3: Prioritization Strategies	IV4: Code Inspection Techniques
Dynamic (D)	Method (M)	Total (T)	Bug finders (BF)
Static-JUPTA (J)	Statement (S)	Additional (A)	Defect prediction (DP)

Table 3 shows the four independent variables (IVs) used in our experiments that could affect TCP performance in terms of APFD:

IV1: Coverage Techniques. For examining the TCP performance of QTEP, we use two representative coverage techniques from the existing coverage-based TCP techniques.

- **Dynamic-coverage-based TCP** is based on the information of the dynamic call graph from the latest run of a subject project. We use test coverage information based on the dynamic call graph to prioritize test cases.
- **Static-coverage-based TCP** ranks test cases based on the information from static call graph. JUPTA is the state-of-the-art static-coverage-based TCP technique [50]. We use JUPTA as a representative static-coverage-based TCP technique for the experiments.

IV2: Coverage Criteria. Since all the studied techniques rely on code coverage information, we also investigate the influence of coverage criteria. We study two widely used coverage criteria: (1) *Method* coverage, (2) *Statement* coverage.

IV3: Prioritization Strategies. As we described in Section 2.1, the *Total* strategy and *Additional* strategy are widely used in most existing studies to schedule the execution order of test cases [7, 47, 50, 65, 66]. Thus, we also investigate the influence of these two different prioritization strategies.

IV4: Code Inspection Techniques. We consider two types of code inspection techniques for detecting fault-prone source code and weighting source code units. They are static bug finder (i.e., FindBugs) and statistical defect prediction model (i.e., CLAMI).

We can form 8 combinations from the first three IVs (IV1 to IV3) from the existing TCP techniques as baselines. The IV1, IV2, and IV3 in Table 3 represent technical options that we can select from the existing coverage-based TCP techniques. Based on acronyms for IV options in Table 3, we can list the 8 combinations as follows: DMT (i.e., **d**ynamic **m**ethod coverage with **t**otal strategy), DMA, DST, DSA, JMT, JMA, JST, and JSA.

We also use the random TCP as a baseline. The random TCP runs all the test cases randomly, therefore the performance of the random TCP might vary across different runs. To mitigate the randomness, we run the random TCP 500 times on each subject and obtain the average performance in APFD. Following existing work [8], we denote the random TCP technique as RT.

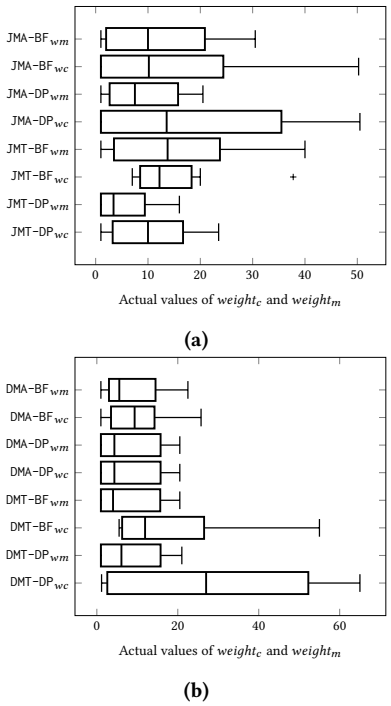


Figure 2: The distribution of the best $weight\_c$ (wc) and $weight\_m$ (wm) for the variants of QTEP that are adapted from static-cover-based TCPs (a) and dynamic-coverage-based TCPs (b).

The IV4 is a technical option in QTEP for detecting buggy code and further weighting source code units. By combining all IVs including IV4, we can define 16 variants of QTEP. Based on acronyms for IV options in Table 3, the **16 variants of QTEP** are DMT-BF, DMA-BF, DST-BF, DSA-BF, JMT-BF, JMA-BF, JST-BF, JSA-BF, DMT-DP, DMA-DP, DST-DP, DSA-DP, JMT-DP, JMA-DP, JST-DP, and JSA-DP.

To investigate the TCP performance of QTEP, we compare the 8 combinations from IV1–IV3 and RT to the 16 variants in Section 5.

4.6 Parameter Setting

As presented in Section 3.2, our algorithm for weighting source code has three parameters, i.e., *weight_base* (default weight for all code units), *weight_c* (weight for detected buggy classes), and *weight_m* (weight for detected buggy methods). Different weights of these parameters could significantly affect the performance of QTEP. In this section, we study the impact of the three parameters of QTEP on the performance of prioritizing both regression and all test cases (both regression and new test cases). Specifically, for regression test cases, we select the first version-pair from each project listed in Table 1 as experimental subjects. When considering both regression and new test cases, we randomly selected 20 faulty versions from the first version-pair of each project as experimental subjects.

We then tune the parameters for each project using each of the 16 variants in QTEP (described in 4.5) and evaluate the specific values of the parameters by the average APFD scores at the class and the method levels (with or without new test cases).

For simplifying the tuning process, we set *weight_base* equal to 1. Then, we set *weight_c* equal to $c \times$ *weight_base* and *weight_m* equal to $m \times$ *weight_base*, we experiment c and m with a range from 1 to 100. We use all the combinations of the three weights in the tuning process, which includes $100 \times 100 \times 20$ (#project) $\times 16$ (#variants of QTEP) experiments for regression test cases, and $100 \times 100 \times 20$ (#project) $\times 20$ (#mutation fault version) $\times 16$ (#variants of QTEP) experiments for all test cases (regression and new test cases).

Figure 2 (a) and Figure 2 (b) show the distribution of the best values of parameters *weight_c* and *weight_m* for all the 16 variants of QTEP on the 20 version-pairs. We could see that the best values of *weight_c* and *weight_m* vary dramatically for different projects. On average, for variants of QTEP that are adapted from static-coverage-based TCPs, *weight_c* is 16.7 times of *weight_base* and *weight_m* is 13.1 times of *weight_base*. For variants of QTEP that are adapted from dynamic-coverage-based TCPs, *weight_c* is 18.7 times of *weight_base* and *weight_m* is 11.6 times of *weight_base*.

In this work, we use the best values of *weight_c* and *weight_m* that are obtained from the first version-pair of a project as default parameters for all the left version-pairs of this project. Note that since project Xml-Security only has one available version-pair from the existing benchmark dataset, thus for this project, we tune parameters and report the performance on the same version-pair.

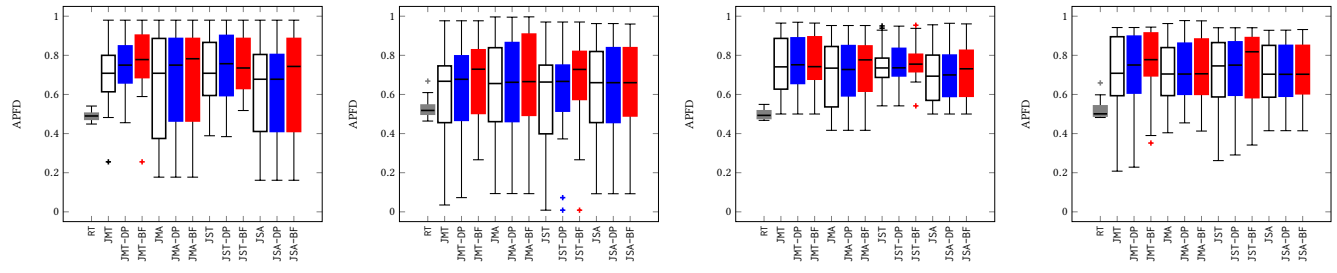
5 RESULTS

5.1 RQ1 & RQ2: Performance of QTEP for Regression and All Test Cases

Figure 3 and Figure 4 show the comparison results in the four scenarios on each of the 20 version-pairs. Specifically, they show the boxplots of the APFD values for the 16 variants of QTEP, the eight variants of coverage-based TCPs, and the random baseline RT in the four scenarios. Each sub-figure presents the detailed APFD results of one type of QTEP (i.e., static-coverage-based or dynamic-coverage-based variants of QTEP) and the corresponding baseline TCPs on a specific scenario. For example, Figure 3 (a) shows the C-R (regression test class) scenario of static-coverage-based techniques, RT, and static-coverage-based variants of QTEP, while Figure 4 (a) shows the C-R scenario of dynamic-coverage-based techniques, RT, and dynamic-coverage-based variants of QTEP. Each boxplot presents the APFD distribution (median and upper/lower quartiles) of prioritization results of one variant of QTEP on the 20 version-pairs. We use gray (●), white (○), yellow (●), blue (●), and red (●) boxes to represent the random, static-coverage-based, dynamic-coverage-based, DP-based QTEP, and BF-based QTEP techniques respectively.

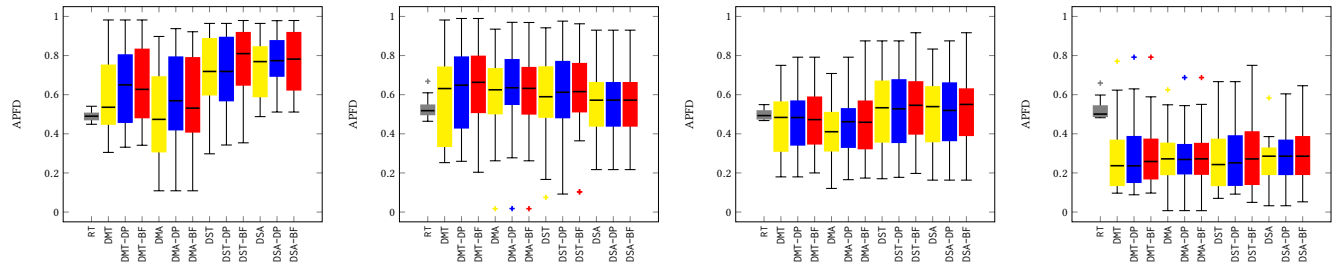
The figures show that overall both DP-based (●) and BF-based (●) variants of QTEP could outperform corresponding traditional coverage-based TCPs (○ and ●) and RT (●) for both regression test cases and new test cases at both method-level and class-level TCPs. In addition, static-coverage-based variants of QTEP techniques are overall more effective than dynamic-coverage-based variants of QTEP. Specifically, for C-R, among all examined TCP techniques, JMT-BF produces the best APFD with a median value of 0.79, which is almost 10% higher than the best traditional coverage-based technique, i.e., JMT. For M-R, JMT-BF outperforms all other examined TCPs. While considering new test cases, JMA-BF and JST-BF produce the best performance for C-RN and M-RN respectively.

We further take a closer look at each individual program. To save space, we only show the detailed comparison between the results of static-coverage-based variants of QTEP and the results



(a) C-R (b) M-R (c) C-RN (d) M-RN

Figure 3: Results of static-coverage-based variants of QTEP and static-coverage-based TCPs (i.e., random ●, static coverage-based TCPs ○, defect-prediction-based variants of QTEP ●, and bug-finder-based variants of QTEP ●)



(a) C-R (b) M-R (c) C-RN (d) M-RN

Figure 4: Results of dynamic-coverage-based variants of QTEP and dynamic-coverage-based TCPs. (i.e., random ●, dynamic-coverage-based TCPs ●, defect-prediction-based variants of QTEP ●, and bug-finder-based variants of QTEP ●)

of the corresponding coverage-based TCPs, since they are overall more effective than dynamic-coverage-based variants of QTEP. The comparison between the dynamic-coverage-based variants of QTEP and the corresponding coverage-based TCPs is also available online [2].

Table 4 shows the average APFD values of all static-coverage-based variants of QTEP and the corresponding coverage-based TCPs on each project. Numbers in brackets are the improvements of DP-based (●) and BF-based (●) variants of QTEP compared to corresponding coverage-based TCPs. We can see that BF-based (●) variants of QTEP improve the APFD values for all the projects. However, the improvement varies on different projects. For example, on project Time&Money, JSA-BF achieves the best APFD for C-R (i.e., 0.91), which is 14 percentage points higher than the corresponding JSA (i.e., 0.77). While on XStream, the improvement is only 1 percentage point. In the worst case, e.g., JSA-DP, QTEP does not improve traditional coverage-based TCPs. The same phenomenon is also observed in dynamic-coverage-based variants of QTEP.

The variations of improvements depending on different projects might be because the performance of fault detection varies on different projects. To explore this, we further compute the Spearman correlation between the false positive rates and the improvements of DP-based and BF-based QTEP on all projects. Results show that the Spearman correlation values for the false positive rates and the improvements of DP-based and BF-based QTEP are -0.50 and -0.52 respectively. This indicates that the performance of QTEP on each project is negatively correlated with the false positive rate of the investigated code inspection technique on this project.

In addition, figures (i.e., Figure 3 and Figure 4) show that all static-coverage-based variants of QTEP generate better results than RT and the improvement ranges from 9 to 30 percentage points. However, we also note that the performance of dynamic-coverage-based variants of QTEP has a dramatically decline when considering M-RN and C-RN compared to static-coverage-based variants of QTEP, and cannot even outperform RT. For example, the APFD of DSA-DP in M-RN is only 0.28, which is 26 percentage points lower than RT. This is because the dynamic coverage information comes from the last execution of the test suite, which does not contain the new test cases. Thus, the faults that can be revealed only by the new test cases are ignored since the coverage information of new test cases is always unavailable in the dynamic-coverage-based TCPs [47, 48]. While, static coverage information of both regression and new test cases could be obtained by static code analysis. Thus, the performances of static-coverage-based variants of QTEP are similar between with and without the new test cases.

For statistical tests, we also conduct the Wilcoxon signed-rank test ($p < 0.05$) to compare the performance of QTEP and existing TCPs. Specifically, we compare each variant of QTEP with its corresponding coverage-based TCP technique on all projects for both regression and new test cases. Results show that eight of the 16 variants of QTEP could achieve significantly better performance than the corresponding coverage-based TCPs (i.e., JMT-DP, JMT-BF, JST-BF, JSA-BF, DMT-BF, DST-BF, DSA-DP, and DSA-BF). For the other eight variants, their performances are slightly better or equal to the corresponding coverage-based TCPs.

In summary, QTEP is overall more effective than the corresponding coverage-based TCP techniques. While dynamic-coverage-based QTEP variants exhibit significantly better performance on regression test cases than on all test cases, static-coverage-based QTEP variants produce similarly good performance on both regression test cases and all test cases (both regression and new test cases).

Table 4: Comparison between the static-coverage-based variants of QTEP and the corresponding coverage-based TCPs for each project

Subject	Scenario	JMT	JMT-DP	JMT-BF	JMA	JMA-DP	JMA-BF	JST	JST-DP	JST-BF	JSA	JSA-DP	JSA-BF
Time&Money	C-R	0.65	0.71(+0.06)	0.73(+0.08)	0.59	0.82(+0.23)	0.85(+0.26)	0.82	0.82	0.87(+0.05)	0.77	0.77	0.91(+0.14)
	M-R	0.24	0.27(+0.03)	0.38(+0.14)	0.49	0.50(+0.01)	0.57(+0.08)	0.12	0.29(+0.17)	0.30(+0.18)	0.47	0.49(+0.02)	0.56(+0.09)
	C-RN	0.61	0.62(+0.01)	0.64(+0.03)	0.63	0.64(+0.01)	0.70(+0.07)	0.64	0.64	0.69(+0.05)	0.69	0.69	0.73(+0.04)
	M-RN	0.36	0.39(+0.03)	0.46(+0.10)	0.60	0.59	0.66(+0.06)	0.34	0.43(+0.09)	0.40(+0.06)	0.57	0.57	0.62(+0.05)
Mime4J	C-R	0.65	0.71(+0.06)	0.76(+0.11)	0.61	0.62(+0.01)	0.61	0.73	0.86(+0.13)	0.78(+0.05)	0.59	0.59	0.59
	M-R	0.71	0.75(+0.04)	0.80(+0.09)	0.55	0.55	0.55	0.69	0.69	0.76(+0.07)	0.55	0.55	0.55
	C-RN	0.75	0.76(+0.01)	0.75	0.65	0.71(+0.06)	0.73(+0.08)	0.70	0.71(+0.01)	0.75(+0.05)	0.62	0.62	0.62
	M-RN	0.68	0.70(+0.02)	0.73(+0.05)	0.48	0.48	0.48	0.63	0.63	0.65(+0.02)	0.48	0.48	0.48
Jaxen	C-R	0.94	0.95(+0.01)	0.95(+0.01)	0.90	0.90	0.90	0.90	0.90	0.90	0.85	0.85	0.85
	M-R	0.67	0.69(+0.02)	0.71(+0.04)	0.77	0.77	0.78(+0.01)	0.73	0.73	0.76(+0.03)	0.78	0.78	0.78
	C-RN	0.80	0.80	0.81(+0.01)	0.74	0.75(+0.01)	0.75(+0.01)	0.80	0.81(+0.01)	0.83(+0.03)	0.65	0.65	0.7(+0.05)
	M-RN	0.65	0.68(+0.03)	0.74(+0.09)	0.66	0.66	0.66	0.68	0.71(+0.03)	0.68	0.68	0.68	0.68
Xml-Security	C-R	0.71	0.73(+0.02)	0.74(+0.03)	0.38	0.38	0.38	0.71	0.73(+0.02)	0.74(+0.03)	0.38	0.38	0.38
	M-R	0.97	0.97	0.97	0.84	0.84	0.84	0.97	0.97	0.97	0.84	0.84	0.84
	C-RN	0.50	0.50	0.50	0.42	0.42	0.42	0.54	0.54	0.54	0.50	0.50	0.50
	M-RN	0.91	0.91	0.91	0.83	0.83	0.83	0.94	0.94	0.94	0.85	0.85	0.85
Xstream	C-R	0.72	0.76(+0.04)	0.75(+0.03)	0.64	0.65(+0.01)	0.65(+0.01)	0.73	0.73	0.76(+0.03)	0.66	0.67(+0.01)	0.66
	M-R	0.66	0.67(+0.01)	0.70(+0.04)	0.72	0.73(+0.01)	0.72	0.66	0.68(+0.02)	0.73(+0.07)	0.73	0.73	0.73
	C-RN	0.88	0.89(+0.01)	0.89(+0.01)	0.82	0.82	0.82	0.86	0.87(+0.01)	0.86	0.82	0.82	0.83(+0.01)
	M-RN	0.76	0.76	0.77(+0.01)	0.81	0.82(+0.01)	0.81	0.76	0.77(+0.01)	0.80(+0.04)	0.81	0.81	0.81
Commons-Lang	C-R	0.67	0.71(+0.04)	0.73(+0.06)	0.75	0.76(+0.01)	0.76(+0.01)	0.52	0.55(+0.03)	0.60(+0.08)	0.70	0.70	0.71(+0.01)
	M-R	0.36	0.36	0.51(+0.15)	0.37	0.38(+0.01)	0.38(+0.01)	0.20	0.24(+0.04)	0.29(+0.09)	0.35	0.35	0.37(+0.02)
	C-RN	0.67	0.67	0.69(+0.02)	0.69	0.69	0.69	0.70	0.70	0.71(+0.01)	0.72	0.72	0.73(+0.01)
	M-RN	0.67	0.67	0.74(+0.07)	0.61	0.62(+0.01)	0.64(+0.03)	0.54	0.57(+0.03)	0.60(+0.06)	0.62	0.62	0.63(+0.01)
Joda-Time	C-R	0.65	0.67(+0.02)	0.74(+0.09)	0.61	0.62(+0.01)	0.62(+0.01)	0.62	0.62	0.62	0.47	0.50(+0.03)	0.49(+0.02)
	M-R	0.70	0.70	0.77(+0.07)	0.76	0.78(+0.02)	0.78(+0.02)	0.70	0.70	0.78(+0.08)	0.66	0.73(+0.07)	0.67(+0.01)
	C-RN	0.71	0.71	0.72(+0.01)	0.66	0.67(+0.01)	0.68(+0.02)	0.72	0.72	0.74(+0.02)	0.70	0.70	0.73(+0.03)
	M-RN	0.83	0.83	0.86(+0.03)	0.80	0.80	0.82(+0.02)	0.85	0.85	0.88(+0.03)	0.78	0.78	0.79(+0.01)

5.2 RQ3: Comparison between the Two Categories of QTEP's Variants

In order to answer RQ3, we first weight all source code units with the detection results from both defect prediction models and FindBugs using Algorithm 1. Note that, in Algorithm 1, we use the tuned best values of *weight_c* and *weight_m* (details are in Section 4.6) for each project under different code inspection techniques. We then run all variants of QTEP on all version-pairs (P_1-P_{20}). As we stated in Section 1, different from most of the existing TCP related studies [44, 50, 64, 66, 82] (mainly focused on regression test cases), in this work, we extensively explore the performance of QTEP on both regression test cases and all test cases. Thus, for each version-pair, we perform experiments on four different scenarios: M-R, C-R, M-RN, and C-RN (details are in Section 4.5).

Table 5 and Table 6 present the average APFDs of all the 16 variants of QTEP on the four different scenarios of the 20 version-pairs. We conduct the Wilcoxon signed-rank test ($p < 0.05$) to compare the performance of BF-based and DP-based variants of QTEP. Between these two variants, better APFD values with statistical significance are shown with an asterisk (*) in Table 5 and Table 6. Results show that overall BF-based variants of QTEP are significantly better than DP-based variants. The improvement could be up to 6 percentage points (JMT-BF: 0.69 vs JMT-DP: 0.63 in M-R of Table 5). Specifically, JMT-BF outperforms all the other variants of QTEP in terms of the average APFD at M-R (i.e., 0.69), C-R (i.e., 0.79), M-RN (i.e., 0.77), and C-RN (i.e., 0.78). One possible reason is BF has a lower false positive rate than DP (details are in Section 5.3).

We further investigate whether the APFDs of different variants of QTEP vary with the false positive rates of the two code inspection techniques. Specifically, we compute the Spearman correlation between the false positive rates and the APFD values of these two

Table 5: The average APFDs of the variants of QTEP that are adapted from static-coverage-based TCPs.

Scenario	JMT-DP	JMT-BF	JMA-DP	JMA-BF	JST-DP	JST-BF	JSA-DP	JSA-BF
C-R	0.75	**0.79***	0.69	0.69	0.74	**0.75***	0.65	**0.67***
M-R	0.63	**0.69***	0.67	**0.68***	0.62	**0.67***	0.66	0.66
C-RN	0.78	0.78	0.74	**0.76***	0.77	**0.78***	0.72	**0.74***
M-RN	0.73	**0.77***	0.73	0.73	0.72	**0.74***	0.72	0.72

Table 6: The average APFDs of the variants of QTEP that are adapted from dynamic-coverage-based TCPs.

Scenario	DMT-DP	DMT-BF	DMA-DP	DMA-BF	DST-DP	DST-BF	DSA-DP	DSA-BF
C-R	0.63	**0.65***	0.61	0.60	0.75	**0.76***	0.78	0.77
M-R	0.64	**0.67***	0.65	0.63	0.61	**0.63***	0.56	0.56
C-RN	0.47	0.47	0.45	**0.46***	0.54	**0.55***	0.54	**0.55***
M-RN	0.30	**0.31***	0.29	0.29	0.30	**0.31***	0.28	**0.29***

categories of QTEP, i.e., DP and BF-based variants. The high correlation value (-0.52) indicates that the accuracy of different variants of QTEP has a negative correlation with the false positive rates of the investigated code inspection techniques. This explains why BF-based variants of QTEP perform slightly better than DP-based variants.

Overall, BF-based variants of QTEP are more effective than DP-based variants for both regression test cases and all test cases (both regression and new test cases).

5.3 RQ4: Effectiveness of FindBugs and Defect Prediction Models

To answer RQ4, we first collect the failure trace of each failed test case on each version-pair and then we parse the failure traces to label all involved source classes and methods as buggy. Source classes and methods that are not involved are labeled as clean.

We then use defect prediction models and FindBugs to identify buggy code units on the later version of each of the 20 version-pairs.

Table 7: The performance of CLAMI and FindBugs on revealing buggy code units. P is precision and R is recall.

Granularity	CLAMI		FindBugs	
	P	R	P	R
Class	0.250	0.496	0.276	0.178
Method	0.115	0.193	0.138	0.027

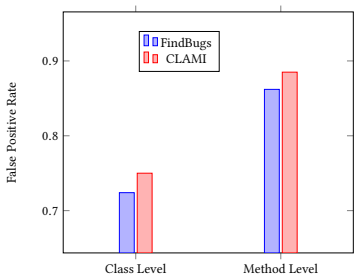


Figure 5: False positive rates of the two code inspection techniques

Note that, we label buggy source classes and methods using the failure traces of failed test cases instead of posted bug reports [57]. This might make the ratio of buggy classes and methods quite small. For instance, when labeling version-pair 3.02-3.03 of Commons-Lang, only 1 out of 196 classes and 2 out of 1,552 methods are labeled as buggy. That is because there is only one failed test case among all regression test cases. Such a limited number of labeled buggy classes and methods could make the different performances between our study and the previous studies on defect prediction [57] and bug finders [62, 77].

To measure the performance of the two code inspection techniques, we use *Precision* and *Recall*, which are widely used to evaluate defect prediction models [31, 51, 56, 80, 91] and bug finders [77]. Precision is the percentage of correctly identified buggy code units in all the code units which are identified buggy, and recall is the percentage of correctly identified buggy code units in all the real buggy code units. To show the overall performance, we use the weighted average precision and recall following existing work [72].

Table 7 summarizes the weighted average performance of CLAMI and FindBugs on the 20 version-pairs listed in Table 1. Specifically, we could see that CLAMI has better recall values than FindBugs at both class and method levels. However, CLAMI has smaller precision values than FindBugs. This implies CLAMI may generate more false positives. We further show the false positive rates in Figure 5, from which we could see that at both class level and method level, CLAMI has a 3.0% higher false positive rate.

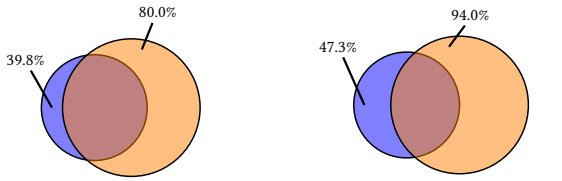
In summary, defect prediction models identify more buggy code units, while FindBugs has a lower false positive rate.

5.4 Time and Memory Overhead

Comparing with traditional coverage-based TCPs, the extra running overhead of QTEP depends on the applied code inspection techniques, i.e., defect prediction models and bug finders, and our source-code-weight algorithm. To understand the overhead of QTEP, we collect the time and space costs for all experiments, and details are presented in Table 8. We can see that the total execution time for weighting source code, running defect prediction models and FindBugs varies from 7.0 to 75.2 seconds. The largest project (in terms of the number of test cases), Joda-Time, uses 96.5MB of memory. As shown in the table, code inspection techniques spent more time than weighting source code units. The total time cost on each project is less than 2 minutes. Overall, the results demonstrate QTEP's practical aspect.

Table 8: The average time cost of QTEP on each subject.

Subject	Time (s)		
	Defect prediction	FindBugs	Weighting code
Time&Money	0.4	6.5	< 0.1
Mime4J	0.4	27.2	< 0.1
Jaxen	0.6	15.3	< 0.1
Xml-Security	0.2	22.3	< 0.1
Xstream	0.9	35.4	< 0.1
Commons-Lang	1.2	69.0	< 0.1
Joda-Time	1.6	73.5	< 0.1



(a) Overlap at class level. **(b) Overlap at method level.**

Figure 6: The overlaps between FindBugs (●) and CLAMI (●).

6 DISCUSSION

6.1 Does the Combination of BF and DP Achieve Better Performance for TCP?

In this work, we have explored the feasibility of leveraging the results of both static bug finders and statistical defect prediction models to improve testing efficiency. As shown in Section 5.2, both BF-based and DP-based variants of QTEP could improve coverage-based TCPs. In this section, we further examine whether combining the detection results of BF and DP could achieve better results. Specifically, following the described process of QTEP in Section 3, we use the combination of detection results of the two code inspection techniques to tune and weight source code units. And then we run the new eight (adapted from both dynamic- and static-coverage-based TCPs) variants of QTEP on the 20 version-pairs. Note that, we experiment with two combinations, i.e., union and intersection.

Overall, the intersection combination produces better performance than the union combination. When comparing with BF-based and DP-based variants of QTEP, results show that the union-combination-based QTEP has a similar performance to DP-based QTEP, we further perform one-way ANOVA analysis [81] at the significance level of 0.05 to investigate whether there is a significant difference between the performance of these two approaches. The high p (0.978) indicates that there is no significant difference between the union-combination-based and DP-based QTEP. In addition, the intersection-combination-based and BF-based QTEP perform similar, also our ANOVA test shows there is no significant difference between them.

The above results suggest that both the union-combination-based and the intersection-combination-based QTEP cannot outperform BF-based QTEP.

To explore this issue, we further show the overlaps between detection results of BF and DP in Figure 6. We could see that at the class level, over 60% detected buggy classes by FindBugs are overlapped

with the buggy classes detected by CLAMI, while the overlaps account for only 20% of the detected buggy classes in CLAMI. At the method level, about 52% predicted buggy methods are overlapped with FindBugs, however the overlaps only account for up to 6% of the detected buggy methods in CLAMI. Consequently, the detection result of the union set from FindBugs and CLAMI is quite similar to the result of CLAMI. While, the detection result of the intersection set is similar to the result of FindBugs. This could help explain why the union-combination-based QTEP and DP-based QTEP produce similar results and the intersection-combination-based QTEP has similar performance to BF-based QTEP.

6.2 Practical Guidelines for TCP

Our study reveals several interesting findings that can serve as the practical guidelines for improving test efficiency.

For different scenarios. Overall, both dynamic- and static-coverage-based variants of QTEP are applicable for improving prioritization of regression test cases. While for prioritizing all test cases (both regression and new test cases), we only recommend static-coverage-based QTEP, since dynamic-coverage-based QTEP cannot handle new test cases well.

For different projects. As we described in Section 5.1, the performance of QTEP on a project is negatively correlated with the false positive rate of the used code inspection technique on this project. Thus, users could choose either DP-based or BF-based QTEP by considering their false positive rates on the history data of this project. While for new projects (no history data are available), we recommend BF-based QTEP, since our experiment results show that DP tends to have a higher false positive rate than BF.

6.3 Threats to Validity

Internal Validity. The proposed QTEP leverages two code inspection techniques to help regression testing. An assumption behind this combination is that the bugs detected by each of them are (largely) different. Our manual inspection confirms this assumption, e.g., all the bugs in the dataset cannot be detected by FindBugs. Some bugs are project-specific bugs, that could be revealed by regression testing only. Note that, our work also suggests that using each technique alone could complement each other to detect more bugs. The success of the proposed approach may also depend on the effectiveness of the static bug finder, i.e., FindBugs and the defect prediction model, i.e., CLAMI. With different static bug finders or different defect prediction models, the performance of QTEP may vary. To evaluate the quality of prioritization, we choose APFD, which has been extensively used in the field of TCP. However, APFD can not reflect time and space costs or the severity of faults. We plan to use more metrics, e.g., APFDc [23], to reduce this threat.

External Validity. In this work, all the experiment subjects are open source projects and written in Java with JUnit test cases. Although they are popular projects and widely used in TCP research, our findings may not be generalizable to commercial projects or projects in other languages. To mitigate this threat, we plan to explore the effectiveness of QTEP on C/C++ projects in the future.

7 RELATED WORK

Many regression testing techniques have been proposed for improving test efficiency by test case prioritization [16, 17, 28, 36, 37, 41, 44, 50, 64–66, 73, 74, 82, 83, 88], test case selection [10, 63, 68], and test case reduction [11, 67, 90]. In terms of TCP techniques, Rothermel et al. [64] first presented a family of prioritization techniques including both the total and additional test prioritization strategies using various coverage information.

A majority of existing TCPs leverage code coverage information to rank test cases. Dynamic code coverage from the last execution is widely used in existing TCP techniques [37, 44, 65, 82]. Another widely used code coverage is static code coverage, which is estimated from static analysis. Mei et al. [50] are the first to prioritize test cases with static coverage information.

Some other TCP techniques [19, 35, 59, 66] leveraged similarity between test cases and source code to prioritize test cases. Specifically, Saha et al. [66] proposed an information retrieval approach for regression testing prioritization based on program changes. Noor et al. [59] proposed a similarity-based approach for test case prioritization using historical failure data.

Instead of using the coverage or similarity information between source code and test cases, some approaches use other software process information as the proxy to rank test cases [7, 27, 43, 52–54, 75, 76, 85, 88]. Arafeen et al. [7] used software requirements to group and rank test cases. Engstrom et al. [27] selected a small set of test cases for regression testing selection based on previously fixed faults. Their work required previously revealed bugs within a given period. For projects that do not have well-maintained bugs or new projects (no past bugs are available), their approach cannot work. While QTEP does not have such limitation, since it focuses on potentially unrevealed faults. Laali et al. [43] proposed to utilize the locations of revealed faults of the executed test cases to rank the remaining test cases. Different from QTEP, they used injected faults, and the performance of their approach on real-world faults is unknown. Yu et al. [85] proposed the fault-based prioritization of test cases that is designed by using the fault-based test case generation models. Different from QTEP, their approach assumed the fault-detecting ability of each test case is available. Miranda et al. [53] proposed scope-aided TCP for testing the reused code by using possible constraints delimiting the new input domain scope. On the contrary, QTEP is not limited to testing the reused code.

8 CONCLUSION

In this paper, to address the limitations of existing TCP algorithms, we present a quality-aware TCP technique named QTEP. Specifically, we leverage code inspection techniques, i.e., statistical defect prediction models and static bug detection techniques, to detect fault-prone source code and then adapt existing coverage-based TCP algorithms by considering the weighted defectiveness of source code. Our evaluation shows that QTEP could improve existing coverage-based TCP techniques for both regression test cases and all test cases. As future work, we plan to explore the impact of fault-revealing capability of test suites and the severities of different bugs on the performance of QTEP. We also plan to investigate the different aggregations of code inspection results in QTEP.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their feedback which helped improve this paper. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] 2017. Major. (2017). http://mutation-testing.org/
[2] 2017. QTEP. (2017). http://asset.uwaterloo.ca/qtep/
[3] 2017. Understand. (2017). https://scitools.com
[4] 2017. WALA. (2017). https://github.com/wala/WALA
[5] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments?. In *ICSE'05*. 402–411.
[6] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *TSE'06* (2006), 608–624.
[7] Md Junaid Arafeen and Hyunsook Do. Test case prioritization using requirements-based clustering. In *ICST'13*. 312–321.
[8] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE'11*. 1–10.
[9] Nathaniel Ayewah, David Hovemeyer, J David Morgenthaler, John Penix, and William Pugh. 2008. Using static analysis to find bugs. *Software, IEEE* 25, 5 (2008), 22–29.
[10] Thomas Ball. 1998. On the limit of control flow analysis for regression test selection. *ACM SIGSOFT Software Engineering Notes* 23, 2 (1998), 134–142.
[11] Jennifer Black, Emanuel Melachrinoudis, and David Kaeli. Bi-criteria models for all-uses test suite reduction. In *ICSE'04*. 106–115.
[12] Benjamin Busjaeger and Tao Xie. Learning for test prioritization: an industrial case study. In *FSE'16*. 975–980.
[13] Ryan Carlson, Hyunsook Do, and Anne Denton. A clustering approach to improving test case prioritization: An industrial case study. In *ICSM'11*. 382–391.
[14] Kaner Cem. Improving the Maintainability of Automated Test Suites. In *QW'97*.
[15] Ray-Yaung Chang, Andy Podgurski, and Jiong Yang. Finding what's not there: a new approach to revealing neglected conditions in software. In *ISSTA'07*. 163–173.
[16] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. Learning to prioritize test programs for compiler testing. In *ICSE'17*. 700–711.
[17] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. Test case prioritization for compilers: A text-vector based approach. In *ICST'16*. 266–277.
[18] Shyam R Chidamber and Chris F Kemerer. 1994. A Metrics Suite for Object Oriented Design. *TSE'94* 20, 6 (1994), 476–493.
[19] Jacek Czerwonka, Rajiv Das, Nachiappan Nagappan, Alex Tarvo, and Alex Teterev. Crane: Failure prediction, change analysis and test prioritization in practice–experiences from windows. In *ICST'11*. 357–366.
[20] Hyunsook Do and Gregg Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *ICSM'05*. 411–420.
[21] Hyunsook Do, Gregg Rothermel, and Alex Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *ISSRE'04*. 113–124.
[22] Fernando Brito e Abreu and Rogério Carapuça. 1994. Candidate metrics for object-oriented software within a taxonomy framework. *JSS'94* 26, 1 (1994), 87–96.
[23] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *ICSE'01*. 329–338.
[24] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *FSE'14*. 235–245.
[25] Dawson Engler and Ken Ashcraft. 2003. RacerX: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*. 237–252.
[26] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. *Bugs as deviant behavior: A general approach to inferring errors in systems code*. 57–72 pages.
[27] Emelie Engstrom, Per Runeson, and Greger Wikstrand. An empirical evaluation of regression testing based on fix-cache recommendations. In *ICST'10*. 75–78.
[28] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *ISSTA'15*. 211–222.
[29] Maurice H Halstead. 1977. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc.
[30] Rachel Harrison, Steve J Counsell, and Reuben V Nithi. 1998. An evaluation of the MOOD set of object-oriented software metrics. *TSE'98* 24, 6 (1998), 491–496.
[31] Ahmed E Hassan. Predicting faults using the complexity of code changes. In *ICSE'09*. 78–88.
[32] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. Comparing white-box and black-box test prioritization. In *ICSE'16*. 523–534.
[33] Charitha Hettiarachchi, Hyunsook Do, and Byoungju Choi. Effective regression testing using requirements and risks. In *SERE'14*. 157–166.
[34] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. *ACM Sigplan Notices* 39, 12 (2004), 92–106.
[35] Vilas Jagannath, Qingzhou Luo, and Darko Marinov. Change-aware preemption prioritization. In *ISSTA'11*. 133–143.
[36] Bo Jiang and WK Chan. Bypassing code coverage approximation limitations via effective input-based randomized test case prioritization. In *COMPSAC'13*. 190–199.
[37] Bo Jiang, Zhenyu Zhang, Wing Kwong Chan, and TH Tse. Adaptive random test case prioritization. In *ASE'09*. 233–244.
[38] Tian Jiang, Lin Tan, and Sunghun Kim. Personalized defect prediction. In *ASE'13*. 279–289.
[39] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *S&P'06*. 6.
[40] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing?. In *FSE'14*. 654–665.
[41] Mijung Kim, Jaechang Nam, Jaehyuk Yeon, Soonhwang Choi, and Sunghun Kim. REMI: defect prediction for efficient API testing. In *FSE'15*. 990–993.
[42] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. Predicting faults from cached history. In *ICSE'07*. 489–498.
[43] Mohsen Laali, Huai Liu, Margaret Hamilton, Maria Spichkova, and Heinz W Schmidt. 2016. Test Case Prioritization Using Online Fault Detection Information. In *21th Ada-Europe International Conference on Reliable Software Technologies*. 78–93.
[44] Zheng Li, Mark Harman, and Robert M Hierons. 2007. Search algorithms for regression test case prioritization. *TSE'07* 33, 4 (2007), 225–237.
[45] Zhenmin Li and Yuanyuan Zhou. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *FSE'05*. 306–315.
[46] Myron Lipow. 1982. Number of faults per line of code. *TSE'82* 4 (1982), 437–439.
[47] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. How does regression test prioritization perform in real-world software evolution?. In *ICSE'16*. 535–546.
[48] Qi Luo, Kevin Moran, and Denys Poshyvanyk. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In *FSE'16*. 559–570.
[49] Thomas J McCabe. 1976. A complexity measure. *TSE'76* 4 (1976), 308–320.
[50] Hong Mei, Dan Hao, Lingming Zhang, Lu Zhang, Ji Zhou, and Gregg Rothermel. 2012. A static approach to prioritizing junit test cases. *TSE'12* 38, 6 (2012), 1258–1275.
[51] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. 2010. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering* 17, 4 (2010), 375–407.
[52] Breno Miranda and Antonia Bertolino. Does code coverage provide a good stopping rule for operational profile based testing?. In *AST'16*. 22–28.
[53] Breno Miranda and Antonia Bertolino. 2016. Scope-aided test prioritization, selection and minimization for software reuse. *JSS'16* (2016), 1–22.
[54] S Mirarab and L Tahvildari. 2007. A Prioritization Approach for Software Test Cases on Bayesian Networks. *FASE'07* (2007), 4422–0276.
[55] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The art of software testing*.
[56] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *ICSE'06*. 452–461.
[57] Jaechang Nam and Sunghun Kim. CLAMI: Defect Prediction on Unlabeled Datasets. In *ASE'15*. 452–463.
[58] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H Pham, Jafar M Al-Kofahi, and Tien N Nguyen. Graph-based mining of multiple object usage patterns. In *FSE'09*. 383–392.
[59] Tanzeem Bin Noor and Hadi Hemmati. A similarity-based approach for test case prioritization using historical failure data. In *ISSRE'15*. 58–68.
[60] Thomas J Ostrand and Elaine J Weyuker. The distribution of faults in a large industrial software system. In *FSE'07*. 55–64.
[61] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. 2005. Predicting the location and number of faults in large software systems. *TSE'05* 31, 4 (2005), 340–355.
[62] Foyzur Rahman, Sameer Khatri, Earl T Barr, and Premkumar Devanbu. Comparing static bug finders and statistical prediction. In *ICSE'14*. 424–434.
[63] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *TOSEM* 6, 2 (1997), 173–210.
[64] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: An empirical study. In *ICSM'99*. 179–188.
[65] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. 2011. Prioritizing test cases for regression testing. *TSE'11* 27, 10 (2011), 929–948.
[66] Ripon K Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E Perry. An information retrieval approach for regression test prioritization based on program changes. In *ICSE'15*. 268–279.
[67] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. Balancing trade-offs in test-suite reduction. In *FSE'14*. 246–256.
[68] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. Comparing and combining test-suite reduction and regression test selection. In *FSE'15*. 237–247.
[69] Md Saeed Siddik and Kazi Sakib. RDCC: An effective test case prioritization framework using software requirements, design and source code collaboration. In *ICCIT'14*. 75–80.
[70] Amitabh Srivastava and Jay Thiagarajan. 2002. Effectively prioritizing tests in development environment. In *ACM SIGSOFT Software Engineering Notes*. 97–106.

[71] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* iComment: Bugs or bad comments?*. In *SOSP'07*. 145–158.
[72] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *ICSE'15*. 99–108.
[73] Xinye Tang, Song Wang, and Ke Mao. Will This Bug-Fixing Change Break Regression Testing?. In *ESEM'15*. 1–10.
[74] Stephen W Thomas, Hadi Hemmati, Ahmed E Hassan, and Dorothea Blostein. 2014. Static test case prioritization using topic models. *EMSE'14* 19, 1 (2014), 182–212.
[75] Paolo Tonella, Paolo Avesani, and Angelo Susi. Using the case-based ranking methodology for test case prioritization. In *ICSM'06*. 123–133.
[76] Sujata Varun Kumar and Mohit Kumar. 2010. Test Case Prioritization Using Fault Severity. *IJCST'10* 1, 1 (2010).
[77] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. Bugram: Bug detection with n-gram language models. In *ASE'16*. 708–719.
[78] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *ICSE'16*. 297–308.
[79] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *FSE'07*. 35–44.
[80] Elaine J Weyuker, Thomas J Ostrand, and Robert M Bell. Using developer information as a factor for fault prediction. In *PROMISE'07*. 8.
[81] Thomas H Wonnacott and Ronald J Wonnacott. 1972. *Introductory statistics*. Vol. 19690.
[82] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *STVR'12* 22, 2 (2012), 67–120.
[83] Shin Yoo, Mark Harman, and David Clark. 2013. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *TOSEM'13* 22, 3 (2013), 19.
[84] Shin Yoo, Robert Nilsson, and Mark Harman. Faster fault finding at Google using multi objective regression test optimisation. In *FSE'11*.
[85] Yuen Tak Yu and Man Fai Lau. 2012. Fault-based test suite prioritization for specification-based testing. *IST'12* 54, 2 (2012), 179–202.
[86] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. Towards building a universal defect prediction model. In *MSR'14*. 182–191.
[87] Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *ICSE'16*. 309–320.
[88] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *ICSE'13*. 192–201.
[89] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. FaultTracer: a change impact and regression fault analysis tool for evolving Java programs. In *FSE'12*. 40.
[90] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. An empirical study of junit test-suite reduction. In *ISSRE'11*. 170–179.
[91] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *PROMISE'07*. 9–9.