Time-Aware Test-Case Prioritization using Integer Linear Programming

Lu Zhang[1,2], Shan-Shan Hou[1,2], Chao Guo[1,2], Tao Xie[3], Hong Mei[1,2,*]

[1]Institute of Software, School of Electronics Engineering and Computer Science
[2]Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
Peking University, Beijing, 100871, China
{zhanglu,houss,guochao09,meih}@sei.pku.edu.cn
[3]Department of Computer Science, North Carolina State University, Raleigh, NC 27695
xie@csc.ncsu.edu

ABSTRACT

Techniques for test-case prioritization re-order test cases to increase their rate of fault detection. When there is a fixed time budget that does not allow the execution of all the test cases, time-aware techniques for test-case prioritization may achieve a better rate of fault detection than traditional techniques for test-case prioritization. In this paper, we propose a novel approach to time-aware test-case prioritization using integer linear programming. To evaluate our approach, we performed experiments on two subject programs involving four techniques for our approach, two techniques for an approach to time-aware test-case prioritization based on genetic algorithms, and four traditional techniques for test-case prioritization. The empirical results indicate that two of our techniques outperform all the other techniques for the two subjects under the scenarios of both general and version-specific prioritization. The empirical results also indicate that some traditional techniques with lower analysis time cost for test-case prioritization may still perform competitively when the time budget is not quite tight.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Measurement, Performance, Experimentation

Keywords

Test-case prioritization, Integer linear programming

1. INTRODUCTION

In the software industry, developers usually rely on regression testing to confirm that changes to the software achieve their intentions and do not introduce unexpected side effects. Typically, regression testing involves executing a large number of test cases and thus is very time-consuming. For instance, the industrial collaborator of Elbaum et al. [11, 13] reported that it costs seven weeks to execute the entire test suite of one of their products.

To cope with the preceding situation, researchers have proposed various techniques for test-case prioritization [32, 11, 33, 13] to re-order the test cases for regression testing to meet the goal of detecting faults as early as possible. However, these techniques do not explicitly consider the time budget (which does not allow the execution of the entire test suite) and the difference in execution time of each test case. To address this problem, Walcott et al. [35] and Alspaugh et al. [1] studied a time-constrained situation for test-case prioritization (i.e., time-aware test-case prioritization). In particular, they formalized the problem of time-aware test-case prioritization as a 0/1 Knapsack problem and proposed various techniques for solving this problem. Walcott et al. [35] studied the use of a genetic algorithm (GA) and empirically compared the proposed approach with the initial ordering, the reverse ordering, random prioritization, and fault-aware prioritization. Alspaugh et al. [1] studied the use of traditional Knapsack solvers for time-aware test-case prioritization and empirically compared seven Knapsack solvers with and without scaling.

As the time budget does not allow the execution of the entire test suite, time-aware test-case prioritization implies the selection of a subset from the test suite for prioritization. In this paper, we propose to explicitly separate test-case selection and prioritization for time-aware test-case prioritization. The rationales for this separation are as follows. First, previous research [39] demonstrates that integer linear programming (ILP) [36] is more effective than GA for selecting a subset of test cases in the context of test-suite reduction. Second, previous research [21] also demonstrates that traditional techniques can outperform GA-based techniques for prioritizing test cases without considering the time budget. In this paper, we propose to use ILP for the test-case selection, and traditional techniques of test-case prioritization for prioritizing the selected test cases.

Previous studies on time-aware techniques for test-case prioritization [35, 1] do not evaluate the effectiveness of traditional techniques for test-case prioritization in the time-constrained situation and a recent study [8] on traditional techniques for test-case prioritization in the time-constrained situation does not consider time-aware techniques. Therefore, it is unknown to what extent the traditional techniques

[*]Corresponding Author

for test-case prioritization can still perform as competitively as time-aware techniques. Note that traditional techniques for test-case prioritization are typically less time-consuming to conduct than techniques for time-aware test-case prioritization, which may take much longer analysis time than the execution time of the entire test suite [35]. In this paper, we empirically compare four of our ILP-based techniques with two GA-based techniques [35] and four traditional techniques for test-case prioritization under the scenarios of both general and version-specific prioritization. The experimental results indicate that our additional techniques are superior to other techniques especially when the time budget is tight; and that the traditional techniques perform competitively when the time budget is not quite tight.

In summary, this paper makes the following main contributions:

- The first attempt to separate test-case selection and prioritization in time-aware test-case prioritization, and the first attempt to use ILP for test-case selection in time-aware test-case prioritization.
- Empirical evaluation of the proposed approach in comparison with existing approaches, including the first empirical comparison of time-aware techniques and traditional techniques in the context of time-aware test-case prioritization.

The rest of this paper is organized as follows. Section 2 presents the formal definition of time-aware test-case prioritization. Section 3 presents the details of our approach. Section 4 presents empirical results of our approach, the existing GA-based approach, and the traditional approach. Section 5 discusses related research. Section 6 concludes.

2. TIME-AWARE TEST-CASE PRIORITIZATION

According to Rothermel et al. [32, 33] and Walcott et al. [35], we formally define the problem of time-aware test-case prioritization as follows.

The Problem of Time-Aware Test-Case Prioritization:
Given: T, a test suite; PT, the set of permutations of all subsets of T; f and $time$, two functions from PT to the real numbers; $time_{max}$, the time budget.

Problem: Find $T' \in PT$ and $time(T') \le time_{max}$, such that $(\forall T'')(T'' \in PT)(T'' \neq T')(time(T'') \le time_{max})[f(T') \ge f(T'')]$.

In the preceding definition, PT is the set of all possible orderings of T and all possible orderings of any subset of T, $time$ is a function that measures the execution time of each such ordering, and f is a function that measures the award value of each such ordering. For test case t, we use $time(t)$ to denote the execution time of t for simplicity. Thus, with the $time$ function, we can decide whether ordering T' satisfies the time budget (i.e., $time_{max}$) through evaluating the inequality $time(T') \le time_{max}$. With the f function, the problem requires us to seek for an ordering that satisfies the time budget and has the highest award value.

3. OUR APPROACH

Similar as previous research [35], our approach also requires that test cases in the original test suite (denoted as T) be independent of each other. This requirement has the following two properties. First, any ordering of T or any subset of T is a feasible ordering. That is to say, any such ordering can be used for the actual testing. Second, let $T' = \{t_1, t_2, t_3, ...t_k\}$ be T or a subset of T, and T'' be a permutation of T', $time(T'') = \sum_{i=1}^{k} time(t_i)$. That is to say, the execution time of any test case remains the same, no matter where it appears in an ordering. These two properties enable us to divide time-aware test-case prioritization into two steps. In the first step, we use ILP to select a subset of test cases that satisfy the time budget. In the second step, we adopt traditional techniques to prioritize the test cases selected in the first step.

Table 1: Coverage and execution time

	st_1	st_2	st_3	st_4	st_5	st_6	Time (Second)
t_1	X	X		X	X		9
t_2	X						2
t_3		X	X	X			6
t_4	X				X		4
t_5						X	5
t_6		X		X			5

In the research of Rothermel et al. [32, 33] and Elbaum et al. [11, 13], the total strategy and the additional strategy are two main strategies for test-case prioritization. These two strategies are applicable for any coverage criterion (such as statement coverage or method coverage[1]). Let us use statement coverage as an example. Total statement-coverage prioritization just sorts the test cases in the descending order of the number of statements covered by each test case. Additional statement-coverage prioritization always orders the test case covering the most statements not yet covered by previously executed test cases before any other previously unexecuted test cases. Similarly, for other coverage criteria, there are also total and additional strategies.

Unlike the work of Walcott et al. [35], which does not explicitly distinguish the total and the additional strategies, our approach supports both total and additional strategies for each coverage criterion. For the ease of presentation, we present our total and additional strategies for time-aware test-case prioritization only in terms of statement coverage. Analogously, we also have total and additional strategies for other coverage criteria (e.g., method coverage) in time-aware test-case prioritization.

3.1 Example

Before we present our approach, we present an illustrative example to facilitate the presentation of our approach.

There are six test cases in a regression test suite (denoted as $T = \{t_1, t_2, t_3, t_4, t_5, t_6\}$) and there are six statements (denoted as $ST = \{st_1, st_2, st_3, st_4, st_5, st_6\}$). We depict the coverage information and the execution time of the six test cases in Table 1. Supposing that the time budget is 19 seconds, we need to find an ordering (denoted as T') of a subset of T under the condition that the execution time of T' is no more than 19 seconds. Furthermore, we also expect that the ordering can be effective in detecting faults as early as possible.

3.2 Total Strategy

In time-aware total statement-coverage prioritization, we select a subset (denoted as T') of the original test suite (denoted as T) such that T' satisfies both the time budget and the following condition. For each test case t in T', if we

[1]Method coverage is similar to function coverage studied in previous research. As the subjects in our experiments are written in Java, we use the term “method” instead of “function”.

denote the number of statements covered by t as $StN(t)$, we require to maximize $\sum_{t \in T'} StN(t)$. Then we prioritize the selected test cases (i.e., T') using traditional total statement-coverage prioritization [32, 11, 33, 13]. In particular, we formalize test-case selection in our time-aware total statement-coverage prioritization as an ILP model consisting of decision variables, an objective function, and a constraint system described below followed by an illustration with the example in Section 3.1.

3.2.1 *Decision Variables*

For each test case, we use a Boolean decision variable to represent whether the test case is selected. Thus, for test suite $T = \{t_1, t_2, ...t_n\}$, we use n Boolean decision variables (denoted as x_i, where $1 \leq i \leq n$). Formally, we define x_i in Formula 1 as follows.

$$x_i = \begin{cases} 1, & \text{if } t_i \ (1 \leq i \leq n) \text{ is selected} \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

3.2.2 *Objective Function*

To achieve the goal of test-case selection in our total strategy, we define our objective function as Formula 2.

$$\max \sum_{i=1}^{n} StN(t_i) * x_i \quad (2)$$

In the objective function, the coefficient of variable x_i is $StN(t_i)$, which represents the number of statements covered by test case t_i. Thus, as x_i is 0 for an unselected test case t_i, only statements covered by selected test cases are counted in the objective function.

3.2.3 *Constraint System*

To ensure that the selected test cases satisfy the time budget, we define the constraint system of this model as the following inequality (i.e., Formula 3), which indicates that the sum of execution time of all the selected test cases is no more than the time budget.

$$\sum_{i=1}^{n} time(t_i) * x_i \leq time_{max} \quad (3)$$

In the constraint system, $\sum_{i=1}^{n} time(t_i) * x_i$ represents the sum of execution time of selected test cases, because if test case t_i is not selected, x_i is 0 and its execution time is not counted in the sum in Formula 3.

3.2.4 *Illustration with the Example*

For the example in Section 3.1, our total statement-coverage strategy yields the following ILP model. The objective function and the constraint system are depicted in Formula 4 and Formula 5, respectively.

$$\max \ (4x_1 + x_2 + 3x_3 + 2x_4 + x_5 + 2x_6) \quad (4)$$

$$9x_1 + 2x_2 + 6x_3 + 4x_4 + 5x_5 + 5x_6 \leq 19 \quad (5)$$

Thus, our total statement-coverage strategy in our approach selects t_1 (covering four statements in nine seconds), t_3 (covering three statements in six seconds), and t_4 (covering two statements in four seconds); and the selected test cases in total cover the six statements nine times in 19 seconds. The ordering is t_1, t_3, and t_4.

3.3 Additional Strategy

Similar to our total strategy, our additional strategy selects a subset (denoted as T') of T such that T' satisfies the time budget, and prioritizes the selected test cases (i.e., T') using traditional additional statement-coverage prioritization [32, 11, 33, 13]. However, the technique for test-case selection in our additional strategy is different from that in our total strategy. In our additional strategy, we select test cases to maximize the number of covered statements, where we count each covered statement just once although more than one selected test case may cover the statement. In particular, we formalize test-case selection in our time-aware additional statement-coverage prioritization as an ILP model described below.

3.3.1 *Decision Variables*

The same as the ILP model in our total strategy, the ILP model in our additional strategy also has a group of Boolean decision variables (denoted as x_i) to represent whether each test case is selected. Furthermore, to facilitate the counting of the number of covered statements, we use another group of Boolean decision variables to represent whether each statement is covered by one or more test cases. Formally, for a set of statements (denoted as $ST = \{st_1, st_2, ...st_m\}$), we define y_j $(1 \leq j \leq m)$ in Formula 6.

$$y_j = \begin{cases} 1, & \text{if one or more selected test cases cover } st_j \\ 0, & \text{otherwise} \end{cases} \quad (6)$$

With the help of y_j $(1 \leq j \leq m)$, we have the following objective function.

3.3.2 *Objective Function*

As the goal of test-case selection in our additional strategy is to maximize the number of covered statements, we define our objective function in Formula 7.

$$\max \sum_{j=1}^{m} y_j \quad (7)$$

In the objective function, if any selected test case does not cover st_j, the value of y_j is 0. Thus, Formula 7 ensures to count each covered statement just once.

3.3.3 *Constraint System*

To ensure that the selected test cases satisfy the time budget, the constraint system for the ILP model in our additional strategy also includes the inequality denoted in Formula 3.

Furthermore, we need a group of inequalities to ensure that, if y_j $(1 \leq j \leq m)$ is 1, at least one test case covering st_j is selected. To define these inequalities, we need the coverage information of the test cases. In particular, for a test suite (denoted as $T = \{t_1, t_2, ...t_n\}$) and a set of statements (denoted as $ST = \{st_1, st_2, ...st_m\}$), we use Formula 8 to represent whether a test case covers a statement.

$$c_{ij} = \begin{cases} 1, & \text{if } t_i \text{ covers } st_j \\ 0, & \text{otherwise} \end{cases} \quad (8)$$

Using the coverage information, we define these inequalities in Formula 9.

$$\sum_{i=1}^{n} c_{ij} * x_i \geq y_j \ (1 \leq j \leq m) \quad (9)$$

If y_j is 1, Formula 9 ensures that at least one test case covering st_j is selected. Otherwise, if no such test case is selected, for any test case t_i $(1 \leq i \leq n)$, either c_{ij} or x_i is 0, and thus the value of $\sum_{i=1}^{n} c_{ij} * x_i$ is 0 for any j $(1 \leq j \leq m)$, not satisfying Formula 9.

3.3.4 Further Test-Case Selection

Using the preceding ILP model, we can select a subset of test cases that can satisfy the time budget and maximize the number of statements covered by the selected test cases. However, there may be some unselected test cases that, if further selected, cannot exceed the time budget. Of course, further selecting such test cases cannot increase the number of covered statements, but these test cases may still be helpful for revealing faults.

To further select these test cases, we use another ILP model, in which we adopt a strategy similar to our total strategy in Section 3.2. We present the decision variables, the objective function, and the constraint system of the ILP model as follows.

Let us use $TC = \{tc_1, tc_2, ...tc_l\}$ to denote the set of unselected test cases and $time_{left}$ to denote the time left from the time budget. The ILP model for further test-case selection includes l Boolean decision variables (denoted as z_k $(1 \leq k \leq l)$), defined in Formula 10.

$$z_k = \begin{cases} 1, & \text{if } tc_k \ (1 \leq k \leq l) \text{ is further selected} \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

We define the objective function for the ILP model in Formula 11.

$$\max \sum_{k=1}^{l} StN(tc_k) * z_k \tag{11}$$

We define the constraint system for the ILP model as the inequality in Formula 12.

$$\sum_{k=1}^{l} time(tc_k) * z_k \leq time_{left} \tag{12}$$

3.3.5 Illustration with the Example

For the example in Section 3.1, our additional statement-coverage strategy yields the ILP model with the objective function and the constraint system depicted in Formula 13 and Formula 14, respectively.

$$\max \ (y_1 + y_2 + y_3 + y_4 + y_5 + y_6) \tag{13}$$

$$\begin{cases} 9x_1 + 2x_2 + 6x_3 + 4x_4 + 5x_5 + 5x_6 \leq 19 \\ x_1 + x_2 + x_4 \geq y_1 \\ x_1 + x_3 + x_6 \geq y_2 \\ x_3 \geq y_3 \\ x_1 + x_3 + x_6 \geq y_4 \\ x_1 + x_4 \geq y_5 \\ x_5 \geq y_6 \end{cases} \tag{14}$$

According to the preceding ILP model, selecting t_3, t_4, and t_5 can cover all the six statements, and the total execution time of the three test cases is 15 seconds. As the time budget is 19 seconds and there are still 4 seconds left, our time-aware additional statement-coverage strategy uses the following ILP model to select more test cases. The objective function and the constraint system are depicted in Formula 15 and Formula 16, respectively.

$$\max \ (4x_1 + x_2 + 2x_6) \tag{15}$$

$$9x_1 + 2x_2 + 5x_6 \leq 4 \tag{16}$$

By solving this ILP model, we further select t_2, and we prioritize the selected test cases in the order of t_3, t_4, t_5, and t_2.

4. EXPERIMENTS

To evaluate the techniques proposed in Section 3, we experimentally compare our techniques and existing techniques for both traditional and time-aware test case prioritization. To evaluate the effectiveness of the techniques considered in our experiments, we adopt the APFD[2] metric defined by Walcott et al. [35] for measuring the effectiveness of time-aware test-case prioritization. Furthermore, we also compare the analysis time efficiency of the considered techniques in our experiments.

In previous research, Rothermel et al. [32, 33] and Elbaum et al. [11, 13] evaluated techniques for test-case prioritization in two scenarios: general prioritization (in which, the effectiveness of a technique is measured in terms of the rate for detecting faults over a succession of subsequent versions) and version-specific prioritization[3] (in which, the effectiveness of a technique is measured in terms of the rate for detecting faults in a particular version).

In general, we investigate the following research questions in our experiments:

- **RQ1**: How do our techniques compare with existing techniques under the scenario of general prioritization in terms of APFD?
- **RQ2**: How do our techniques compare with existing techniques under the scenario of version-specific prioritization in terms of APFD?
- **RQ3**: How do our techniques compare with existing techniques in terms of analysis time efficiency?

4.1 Experimental Setup

We conducted our experiments on a PC with a 3GHz Intel Pentium 4 CPU and 1GB memory running the Windows XP operating system. In the rest of this subsection, we present detailed information of the techniques considered in our experiments (Section 4.1.1), subject programs, versions, faults, and test suites used in our experiments (Section 4.1.2), and how we collected the information of coverage and execution time in our experiments (Section 4.1.3).

4.1.1 Considered Techniques

Table 2 lists the 11 techniques considered in our experiments. First, as the control technique in our experiments, we considered random prioritization (abbreviated as rand in Table 2), in which we iteratively select one test case randomly from the test suite until the selected test cases reach the time budget. We did not consider optimal prioritization as a control technique, as it is not straightforward to obtain the optimal ordering for time-aware test-case prioritization even if the faults are known.

Second, for our approach, we considered both the total and the additional strategies at both the statement and the method levels. In particular, we considered four techniques for our approach: time-aware total statement-coverage prioritization via ILP (TA-ILP-st-total), time-aware additional statement-coverage prioritization via ILP (TA-ILP-st-addtl),

[2]APFD is the abbreviation of *Average of the Percentage of Faults Detected*, which was first introduced by Rothermel et al. [32] for traditional test-case prioritization. Walcott et al. [35] adapted the APFD metric for time-aware test-case prioritization.

[3]Researchers (such as Srivastava and Thiagarajan [34]) have proposed techniques especially for version-specific prioritization. However, previous research [32, 11, 33, 13] also evaluated techniques for general prioritization under the scenario of version-specific prioritization.

Table 2: Techniques being experimented

No.	Abbreviation	Description
T1	rand	random prioritization
T2	trdtl-st-total	traditional total statement-coverage prioritization
T3	trdtl-st-addtl	traditional additional statement-coverage prioritization
T4	trdtl-md-total	traditional total method-coverage prioritization
T5	trdtl-md-addtl	traditional additional method-coverage prioritization
T6	TA-GA-st	time-aware statement-coverage prioritization via GA
T7	TA-GA-md	time-aware method-coverage prioritization via GA
T8	TA-ILP-st-total	time-aware total statement-coverage prioritization via ILP
T9	TA-ILP-st-addtl	time-aware additional statement-coverage prioritization via ILP
T10	TA-ILP-md-total	time-aware total method-coverage prioritization via ILP
T11	TA-ILP-md-addtl	time-aware additional method-coverage prioritization via ILP

time-aware total method-coverage prioritization via ILP (TA-ILP-md-total), and time-aware additional method-coverage prioritization via ILP (TA-ILP-md-addtl). As our approach models test-case selection in the four techniques as ILP models, we employed IBM's SYMPHONY [29] for solving the ILP models.

Third, as there is no reported study on evaluating traditional techniques for test-case prioritization in the time-constrained situation, it should be interesting to consider the comparison between our techniques and traditional techniques. In particular, we considered four traditional techniques for test-case prioritization [32, 11, 33, 13] with some straightforward adaptation: traditional total statement-coverage prioritization (trdtl-st-total), traditional additional statement-coverage prioritization (trdtl-st-addtl), traditional total method-coverage prioritization (trdtl-md-total), and traditional additional method-coverage prioritization (trdtl-md-addtl). In our adaptation, for each such technique, we use the technique to produce an ordering of the entire test suite, and in this ordering, we take the longest prefix that still satisfies the time budget as the ordering for time-aware prioritization. For example, if the ordering of the entire test suite is $< t_1, t_2, t_3, t_4, t_5 >$, $< t_1, t_2, t_3 >$ satisfies the time budget, and $< t_1, t_2, t_3, t_4 >$ does not satisfy the time budget, we use $< t_1, t_2, t_3 >$ as the ordering for time-aware prioritization.

Finally, as Walcott et al. [35] also studied time-aware test-case prioritization, we compared our approach with their approach at both the statement and the method levels in our experiments. In particular, we considered two techniques for their GA-based approach: time-aware statement-coverage prioritization via GA (TA-GA-st) and time-aware method-coverage prioritization via GA (TA-GA-md). The settings of the parameters of our implementation of the GA-based approach are the same as those provided by Walcott et al. [35]. In particular, the number of initial orderings of test cases (denoted as NO) and the number of iterations for the genetic algorithm (denoted as NI) are two main parameters of the GA-based approach. Like Walcott et al. [35], our experiments considered three combinations of the two parameters: ($NO = 60$, $NI = 25$), ($NO = 30$, $NI = 50$), and ($NO = 15$, $NI = 75$). As the GA-based approach produces initial orderings of test cases randomly, for each combination, we ran the GA-based approach five times and used the average results of the five runs as the results of the GA-based approach for that combination.

Table 3: Statistics of subjects

	JDepend	*JTopas*
Classes/Interfaces	22	44
Methods	305	555
Non-Comment Source Statements	1808	5361
Test Cases	53	209
Faults	40	40

4.1.2 Subject Programs, Versions, Faults, and Test Suites

In our experiments, we used two programs written in Java as subjects: *JDepend* (Release 2.9), which is downloadable from its website[4], and *JTopas* (Release 0.6), which is downloadable from the Software-artifact Infrastructure Repository (SIR)[5]. Table 3 lists the statistics about the two subjects.

JDepend is a tool that measures the design quality of Java software. Walcott et al. [35] also used *JDepend* to evaluate their GA-based approach to time-aware test-case prioritization. The developers of *JDepend* distribute 53 test cases with the software. We seeded 40 faults[6] in the original program using Jester [25] based on mutation operators [26] such as wrong constants and variables, and faults in arithmetic, logical, and relational operators. The 53 test cases can detect all the 40 faults. In fact, we repeatedly seeded one new fault until the total number of seeded faults that the 53 test cases can detect reached 40.

JTopas is a tool that facilitates users to tokenize and parse arbitrary text data. Do and Rothermel [9] used *JTopas* in their experiments on test-case prioritization. From SIR, there are 209 test cases distributed with the *JTopas* subject. Furthermore, we seeded 40 faults in *JTopas* using the same procedure for *JDepend* and the 209 test cases can detect all the 40 faults.

In the scenario of version-specific prioritization, for each subject, we required a series of faulty versions. To create these faulty versions, we adopted the same methodology used by Elbaum et al. [11, 13]. First, we created 40 single-fault versions for *JDepend* and 40 single-fault versions for *JTopas*. Second, for *JDepend*, we created 10 multiple-fault versions, each being the combination of some single-fault versions; and for *JTopas*, we created 10 multiple-fault versions in the same way. In the scenario of version-specific prioritization, these 20 multiple-fault versions served as the subject versions.

4.1.3 Information Collected for Coverage and Execution Time

In our experiments, we used Emma[7] to collect the cover-

[4]`http://www.clarkware.com/software/JDepend.html`, Accessed in Jan. 2009.

[5]`http://sir.unl.edu/portal/index.html`, Accessed in Jan. 2009.

[6]To produce results comparable with Walcott et al. [35], we seeded the same number of faults.

[7]`http://emma.sourceforge.net/index.html`, Accessed in Jan. 2009.

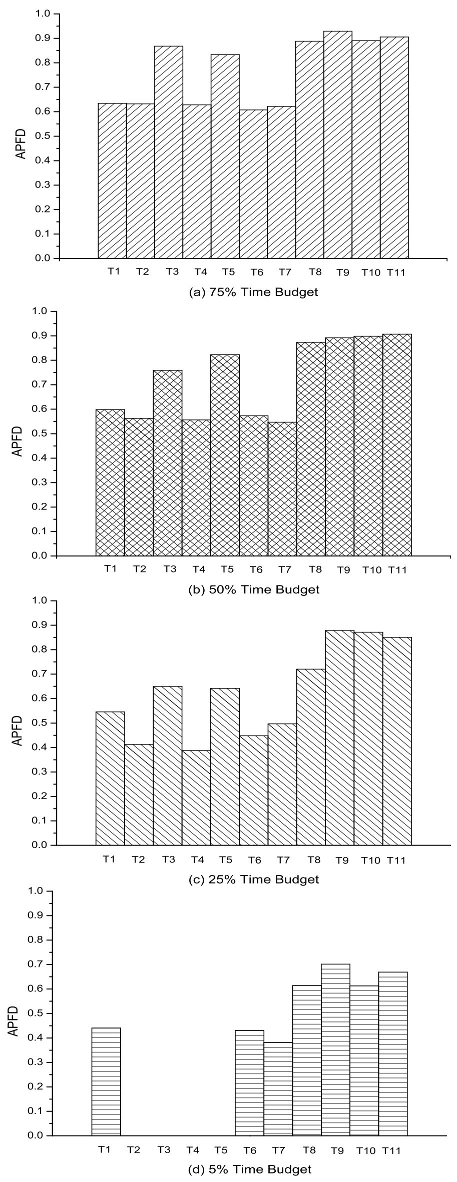

Figure 1: General prioritization for *JDepend*

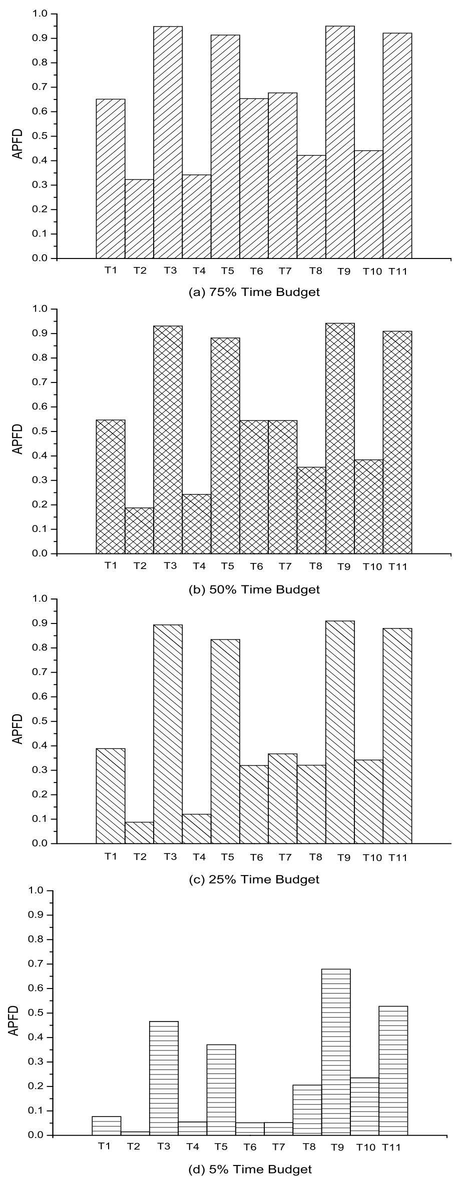

Figure 2: General prioritization for *JTopas*

age information. As Emma provides the coverage information of only an entire test suite, for either subject, we created a series of test suites, each containing just one test case. That is to say, we created 53 test suites for *JDepend* and 209 test suites for *JTopas*. Using these test suites, we collected the coverage information for each test case, and thus our implementation of the considered techniques used the collected coverage information instead of interacting with Emma directly. As interaction with Emma is time-consuming, our implementation ensured that all the techniques considered in our experiments execute efficiently. As both *JDepend* and *JTopas* are in Java, we executed each test case and used the facility provided by JDK to record its execution time. In total, the execution time of the 53 test cases for *JDepend* is 2.141 seconds[8] and that of the 209 test case for *JTopas* is 5.859 seconds.

[8]Due to the difference in hardware and software configurations, the total execution time for *JDepend* in our experiments is different from that in previous research [35].

4.1.4 Time Budgets

Following Walcott et al. [35], we used their three time budgets for each subject program: 25%, 50%, and 75% of the execution time of the entire test suite. We also used 5% of the total execution time to represent a very tight time budget for each subject program.

4.2 Experimental Results

4.2.1 RQ1: Effectiveness for General Prioritization

The first research question is concerned with comparing the effectiveness achieved by the 11 techniques under the scenario of general prioritization. Figures 1 and 2 depict the results on *JDepend* and *JTopas*, respectively. The X-axis shows the 11 techniques and the Y-axis shows the APFD values. Concerning the comparison of our techniques with other techniques, we have the following observations.

First, almost for both subjects and all the four time budgets, the four techniques of our ILP-based approach (i.e., $T8$, $T9$, $T10$, and $T11$) outperform the two techniques of the GA-based approach (i.e., $T6$ and $T7$). The only exception

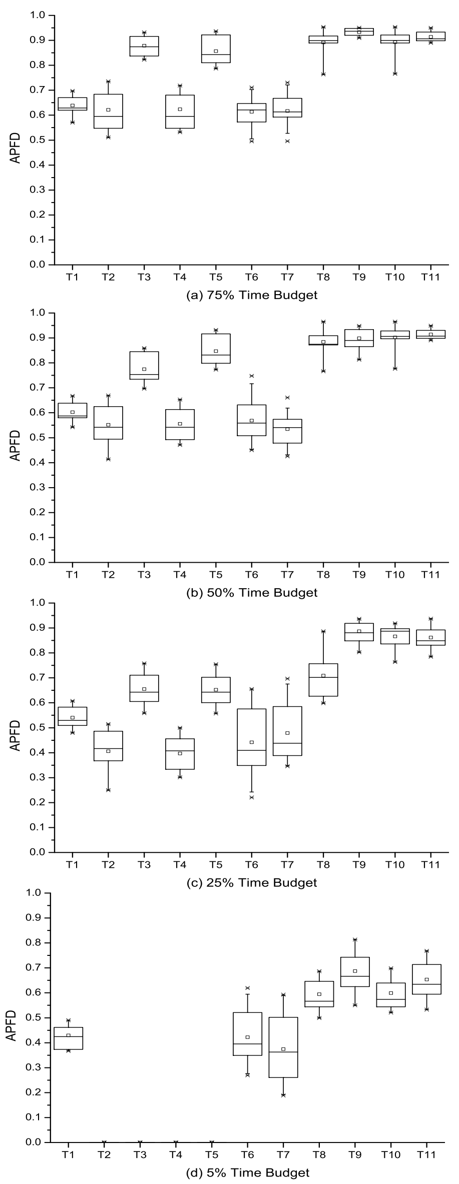

Figure 3: Version-specific prioritization for *JDepend*

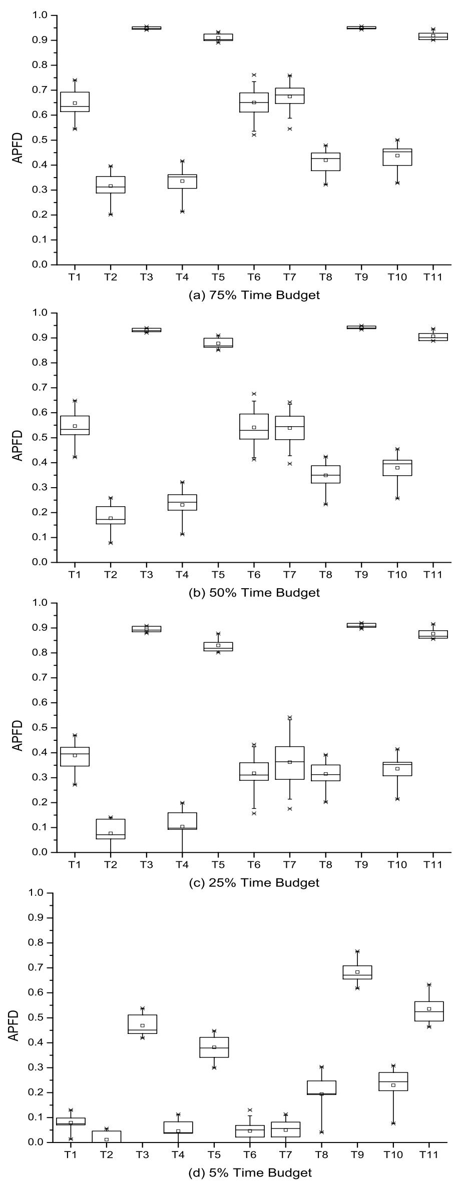

Figure 4: Version-specific prioritization for *JTopas*

is that, for *JTopas*, both GA-based techniques outperform our two total techniques for 75%, 50%, and 25% time budgets. Note that, for the 5% time budget for *JDepend*, the execution time of the first test case in each of the traditional total and additional techniques exceeds the time budget and thus produces no APFD value in Figure 1. It should also be noted that, for *JDepend*, the results of the GA-based approach are a little different from the results reported by Walcott et al. [35]. We suspect the reason to be that the seeded faults in our experiments are different from theirs.

Second, when comparing our techniques with the traditional techniques (i.e., $T2$, $T3$, $T4$, and $T5$), each of our four techniques (i.e., $T8$, $T9$, $T10$, and $T11$) outperforms its corresponding traditional technique for both subjects and all the four time budgets, although for some cases the differences are very small. Furthermore, the difference between our techniques and the traditional techniques becomes larger when the time budget becomes tighter. For *JDepend*, our additional techniques slightly outperform the traditional additional techniques for the 75% time budgets and the differences become more significant for 50%, 25%, and 5% time budgets. For *JTopas*, the traditional additional techniques perform very competitively for 75%, 50%, and 25% time budgets but our additional techniques significantly outperform the corresponding traditional additional techniques for the 5% time budget. This result also indicates that, when the time budget is not quite tight, using the traditional techniques based on additional coverage would be sufficient.

Third, except for the two total techniques in our approach for *JTopas* for 75%, 50%, and 25% time budgets, our techniques also outperform the random prioritization (i.e., $T1$) for both subjects and all the four time budgets. It should be noted that, for some cases, the random prioritization even outperforms the GA-based techniques and some of the traditional techniques.

From the preceding three observations, we conclude that our techniques (especially the additional techniques) outperform the other techniques for both *JDepend* and *JTopas*. Beside the three observations concerning the comparison of our techniques and other techniques, we also have the following observations from Figures 1 and 2.

First, with the increase of the time budget, the effectiveness of each technique also increases. This result is natural, because when the time budget increases, the constraint imposed by the time budget loosens and thus it becomes easier for each technique to select a good ordering.

Second, techniques based on additional coverage (i.e., $T3$, $T5$, $T9$, and $T11$) typically outperform techniques based on total coverage (i.e., $T2$, $T4$, $T8$, and $T10$). Note that, for $JTopas$, both our total techniques and the traditional total techniques achieve surprisingly unsatisfactory performance compared with their corresponding additional techniques. We checked the source code and the test cases of $JTopas$. We found that the 209 test cases can be classified into a few groups, and each group contains test cases with very similar coverage and fault-detection capability. Thus, the total strategy typically orders test cases from the same group next to each other, and thus produces unsatisfactory APFD values.

Third, for both subjects, a technique at the statement level does not significantly outperform the corresponding technique at the method level. For some cases, the method-level technique may even outperform the statement-level technique. One possible explanation is that methods in both subjects are typically very small and thus techniques based on statement coverage do not actually differ much with techniques based on method coverage. In $JDepend$, one method contains only 5.93 non-comment source statements on average; in $JTopas$, one method contains 9.66 non-comment source statements on average. In fact, the preceding information is in accordance with the trend that a statement-level technique is more likely to outperform its corresponding method-level technique for $JTopas$ than for $JDepend$.

Fifth, when comparing any two techniques, the difference between them becomes larger when the time budget becomes tighter. That is to say, with a tighter time budget, the need to seek for a good prioritization becomes more urgent.

4.2.2 RQ2: Effectiveness for Version-Specific Prioritization

The second research question is concerned with comparing the effectiveness of the 11 techniques under the scenario of version-specific prioritization. The box plots in Figures 3 and 4 depict the results of the comparison for $JDepend$ and $JTopas$, respectively. From the two figures, we have the following observations.

First, under the scenario of version-specific prioritization, all the 11 techniques perform similarly compared with the scenario of general prioritization for both subjects. That is to say, our techniques also outperform the other techniques in version-specific prioritization, except for the two total techniques for $JTopas$. Note that, similar to Figure 1, the four traditional techniques in Figure 3 have no APFD value for the 5% time budget for $JDepend$. Furthermore, the traditional techniques based on additional coverage achieve effectiveness somehow close to our techniques when the time budget is not quite tight.

Second, beside random prioritization, all the other 10 techniques become less stable when the time budget becomes tighter. We suspect the main reasons as follows. When the time budget is tight, a particular technique cannot guarantee that the executed test cases can detect all the faults in all the versions. Thus, for a particular version, the number of faults not detected by the executed test cases seriously affects the effectiveness of the technique. That is to say, the instability comes from the difference among versions. For the random prioritization, whose instability mainly comes from the randomness of prioritization, the difference among versions does not further increase its intrinsic instability.

Third, concerning the stability of each technique, our techniques, especially the two additional techniques (i.e., $T9$ and $T11$), perform more stably than the other techniques for both subjects and for all the four time budgets. That is to say, no matter what the version is and what the time budget is, our additional techniques always achieve the expected effectiveness.

4.2.3 RQ3: Analysis Time Efficiency

The third research question is concerned with comparing the analysis time efficiency of the 11 techniques. As random prioritization serves as the control technique and does not require much analysis time, we compare only the other 10 techniques. Given a subject and a time budget, the analysis time cost for each GA-based technique is the average analysis time cost under different parameters and different initial orderings. Table 4 lists the analysis time costs of the 10 techniques for both $JDepend$ and $JTopas$. From this table, we have the following observations.

First, the four traditional techniques are much more time efficient than the other techniques. For the two traditional total techniques, due to the precision limit, we even record zeros as their time costs in the table.

Second, our four techniques are more time efficient than the two GA-based techniques in most cases. The only exceptions are our additional statement-coverage technique for both $JDepend$ and $JTopas$ and our additional statement-coverage technique for $JTopas$. Within our approach, the two additional techniques require more analysis time than the two total techniques; and the two statement-level techniques require more time than the two method-level techniques. Furthermore, when the time budget becomes very tight, the analysis time of our additional techniques also increases.

Third, the GA-based approach seems to be the least time efficient in most cases. Within the GA-based approach, the statement-level technique requires more analysis time than the method-level technique. Furthermore, unlike our approach, the analysis time of the GA-based approach seems to increase when the time budget becomes less tight. Note that the analysis time costs of the GA-based approach are much smaller than those reported by Walcott et al. [35]. One possible explanation is that we used pre-collected coverage information in the experiments.

4.2.4 Summary

In summary, we have the following main findings from the preceding experimental results:

- Concerning the rate of fault detection, our ILP-based approach (especially the two additional techniques) outperforms all the other approaches considered in the experiments for both general and version-specific prioritization.

- Concerning the analysis time costs, our ILP-based approach outperforms the GA-based approach, but requires more analysis time than the traditional techniques for test-case prioritization.

Table 4: Analysis time costs for the subjects (Sec.)

Subject	*JDepend*				*JTopas*			
Time	75%	50%	25%	5%	75%	50%	25%	5%
T2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T3	0.02	0.02	0.02	0.00	0.63	0.63	0.63	0.67
T4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
T5	0.02	0.02	0.02	0.00	0.14	0.14	0.14	0.16
T6	22.7	21.8	17.1	15.8	3498	1498	335	11.3
T7	5.88	5.22	4.48	3.94	708	310	75.6	2.98
T8	0.69	0.39	0.31	0.20	2.39	2.91	2.13	1.64
T9	1.27	1.13	2.42	35.71	5.80	15.06	7.17	109
T10	0.22	0.20	0.23	0.19	2.09	0.81	1.08	1.63
T11	0.67	0.52	0.84	3.52	1.25	2.50	1.81	4.66

- When the time budget is not quite tight, traditional techniques based on additional coverage can be as competitive as our approach for both general and version-specific prioritization.
- With a tight time budget, it is preferable to choose an advanced technique (like ours) for prioritization, as the tight time budget may magnify the differences between techniques.

4.3 Threats to Validity

4.3.1 Internal Validity

Threats to internal validity are concerned with the uncontrolled factors that may also be responsible for the results. In our experiments, the main threat to internal validity is the possible faults in our implementation of the techniques considered in our experiments and the tool for calculating the APFD values. To reduce this threat, we reused some existing robust components, such as IBM SYMPHONY. Furthermore, we reviewed all the code that we produced for our experiments before conducting the experiments.

4.3.2 External Validity

Threats to external validity are concerned with whether the results in our experiments are generalizable for other situations. The first threat lies in the representativeness of the subjects. To reduce this threat, we chose two subjects (including one used in previous research [35] for evaluating an approach to time-aware test-case prioritization) with medium sizes. The second threat lies in the faults in the subjects. To reduce this threat, we used a widely accepted procedure[9] to produce the faults for the two subjects. The third threat lies in the test cases used in our experiments. To reduce this threat, we used the test cases distributed by the developers for both subjects. Conducting more experiments using more subjects of larger sizes with more real-world test cases and faults is one way to further reduce these threats.

4.3.3 Construct Validity

Threats to construct validity are concerned with whether the setup and the measurement in our experiments reflect real-world situations. The main threat to construct validity is the APFD metric. To reduce this threat, we used the same APFD metric proposed by Walcott et al. [35] for measuring the effectiveness of time-aware test-case prioritization. To our knowledge, APFD metrics are widely accepted metrics for various kinds of test-case prioritization, and the APFD metric used in our experiments is the only APFD metric designed particularly for time-aware test-case prioritization. However, as pointed out by Walcott et al. [35], this APFD metric also has its limitations. Further reduction of this threat requires the use of better metrics to assess the effectiveness of techniques for time-aware prioritization. One possible metric is the APFD metric proposed by Qu et al. [28], which considers the situation where the budget does not allow executing all the test cases or detecting all the faults.

[9]Andrew et al. [2] and Do and Rothermel [9] reported empirical evidence that faults seeded via mutation operators achieve similar effects as real faults in testing experiments, especially in those for test-case prioritization.

5. RELATED WORK

Test-case prioritization [37, 32] is an intensively studied research topic in regression testing. Techniques for test-case prioritization aim to improve the rate of fault detection through re-ordering test cases for execution. In the literature, there are several lines of research on test-case prioritization.

The first line of research is to study techniques for test-case prioritization based on different coverage criteria. Rothermel et al. [32, 33] developed a family of techniques for test-case prioritization based on several coverage criteria at the statement level, such as statement coverage, branch coverage, and the probability of exposing known faults. Elbaum et al. [11, 13] further considered the coverage criterion at the function level. Do et al. [10] considered the coverage criteria at the block level and the method level for Java software. Note that there is a subtle difference between function coverage and method coverage: function coverage implies procedural code and method coverage implies object-oriented code. Jones and Harrold [16] considered a coverage criterion at a very fine granularity: the modified condition/decision coverage. Korel et al. [19, 18] even considered the coverage of the system model. Typically, for each coverage criterion, researchers used two greedy strategies for test-case prioritization (i.e., the total and the additional strategies).

The second line of research is to study techniques for test-case prioritization under different usage scenarios. That is to say, techniques for test-case prioritization can be used for the scenario of both general prioritization and version-specific prioritization. In general prioritization, a technique is expected to be effective over a succession of subsequent versions of the software; in version-specific prioritization, a technique is expected to be effective for a particular version. Most of the research (e.g., Rothermel et al. [32, 33]) on test-case prioritization focused on general prioritization. Elbaum et al. [11, 13] studied the effectiveness of various techniques for test-case prioritization under the scenario of version-specific prioritization, but the studied techniques are not designed especially for version-specific prioritization. Srivastava and Thiagarajan [34] proposed a technique especially for version-specific prioritization. Unlike previous techniques, Srivastava and Thiagarajan's technique further considers the changes between a version and its previous version, and uses the coverage of the code impacted by the changes to guide the prioritization process.

The third line of research is to study different strategies and practical complications in test-case prioritization. The total and the additional strategies are two widely-investigated greedy strategies for test-case prioritization. However, Rothermel et al. [33] also pointed out that the greedy strategies may not always produce the optimal ordering of test cases.

Li et al. [21] further studied another greedy strategy (i.e., the 2-optimal strategy based on the k-optimal greedy algorithm [22]) and two meta-heuristic search strategies [30] (i.e., the hill-climbing strategy and the strategy using a genetic algorithm). Elbaum et al. [12, 23] and Park et al. [27] studied the impacts of test costs and fault severities on test-case prioritization. Elbaum et al. [12, 23] proposed a new APFD metric named $APFD_C$ to consider test costs and fault severities in the evaluation of test-case prioritization. They also proposed techniques for test-case prioritization based on the weighted function coverage, where estimated fault proneness or severity serves as the weight. Park et al. [27] proposed a technique based on estimation of test costs and fault severities using historical information. In our previous research [15], we studied the constraint imposed by request quotas of Web services when prioritizing test cases for regression testing of software composed of Web services. We divided the testing process into a series of time slots and adopted integer linear programming for the test-case prioritization under the quota constraint for each time slot.

The research most similar to our research in this paper is the research by Kim and Porter [17], by Walcott et al. [35], by Alspaugh et al. [1], and by Do et al. [8]. Kim and Porter [17] studied the situation when the resource constraint does not allow the execution of the entire test suite, and they proposed a technique based on the performance of each test case in prior testing using exponential smoothing. Given a percentage number (denoted as n), Kim and Porter's technique selects and prioritizes $n\%$ test cases in the entire test suite. Walcott et al. [35] studied the problem of time-aware test-case prioritization, which considers an explicit time budget and the difference in execution time for each test case. Walcott et al. proposed an approach based on a genetic algorithm and empirically compared the proposed approach with the initial ordering, the reverse ordering, and two control techniques (i.e., random prioritization and fault-aware prioritization). Walcott et al. also defined a new APFD metric for evaluating the effectiveness of prioritization in the time-constrained situation. Alspaugh et al. [1] further studied the problem of time-aware test-case prioritization, and empirically compared seven Knapsack solvers (i.e., the random, the greedy by ratio, the greedy by value, the greedy by weight, dynamic programming, generalized tabular, and the core) with and without scaling. Do et al. [8] evaluated traditional techniques for test-case prioritization in the context of time-aware test-case prioritization.

In this paper, we also study the problem of time-aware test-case prioritization. Our research differs from previous research on test-case prioritization as follows. First, to our knowledge, this paper presents the first study on the use of integer linear programming for time-aware test-case prioritization. Second, to our knowledge, this paper presents the first empirical comparison of techniques specific to time-aware test-case prioritization and traditional techniques for test-case prioritization in the context of time-constrained prioritization.

As our approach involves the selection of a subset from the original test suite, our approach is also related to the research on test-suite reduction [14, 6, 24, 5, 38, 39] and regression test selection [20, 7, 31, 4, 3] in a broad sense. Test-suite reduction, which is also referred to as test-suite minimization, aims to select the minimal subset from the original test suite while maintaining the same capability of coverage. Like test-case prioritization, test-suite reduction is applicable for different overage criteria, and there are different strategies for test-suite reduction (e.g., greedy strategies [14, 6], simulated annealing and genetic algorithms [24, 38], and integer linear programming [5]). Regression test selection aims to select a subset from the original test suite to test a specific modified version. Typically, such a technique analyzes the changes between a version and its previous version to select test cases that are most likely to be affected by the changes. Based on whether the technique can guarantee that the selected test cases have the same capability to detect faults as the original test suite, techniques for regression test selection can be divided into safe techniques [7, 31, 3] and non-safe techniques [20, 4]. It should also be noted that some research considered test-suite reduction [16] or regression test selection [37] together with test-case prioritization. The technique for test-case selection proposed in this paper fundamentally differs from previous techniques for test-case selection in both test-suite reduction and regression test selection, as the objective and constraints of our test-case selection differ from those in previous techniques for test-case selection. For example, although both our approach and the approach by Black et al. [5] use ILP for test-case selection, the ILP models formulated by our approach are significantly different from those by Black et al. [5]. The reason is that our ILP models aim to maximize total and additional coverage under the constraint of the time budget while Black et al.'s ILP models aim to minimize the number of selected test cases and to maximize fault detection under the constraint of maintaining coverage.

6. CONCLUSION AND FUTURE WORK

In this paper, we propose a novel approach to time-aware test-case prioritization using integer linear programming. This paper also reports empirical results of comparing four techniques for our approach with two techniques for the GA-based approach and four traditional techniques for test-case prioritization. The empirical results indicate that our additional techniques are superior to other techniques, especially when the time budget is tight, and the traditional additional techniques with lower analysis time cost can perform competitively when the time budget is not quite tight.

In future work, we plan to study the following issues. First, we plan to further improve our approach in the following ways: using additional coverage rather than resorting to total coverage when the full coverage is reached, and considering the execution time when prioritizing the selected test cases. Second, we plan to investigate other ways of test-case selection for time-aware test-case prioritization. Finally, we plan to conduct more experiments on larger subjects to further investigate the concern of time cost for our approach.

Acknowledgments

The authors from Peking University are supported by the 973 Program of China No. 2009CB320703, the 863 Program of China (863) No. 2007AA010301, the Science Fund for Creative Research Groups of China No. 60821003, and the National Science Foundation of China No. 90718016. Tao Xie's work is supported in part by NSF grant CCF-0725190.

7. REFERENCES

[1] S. Alspaugh, K. R. Walcott, M. Belanich, G. M. Kapfhammer, and M. L. Soffa. Efficient time-aware prioritization with knapsack solvers. In *Proc. WEASELTech*, pages 17–31, 2007.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. ICSE*, pages 402–411, 2005.

[3] T. Ball. On the limit of control flow analysis for regression test selection. In *Proc. ISSTA*, pages 134–142, 1998.

[4] D. Binkley. Semantics guided regression test cost reduction. *IEEE TSE*, 23(8):498–516, 1998.

[5] J. Black, E. Melachrinoudis, and D. Kaeli. Bi-criteria models for all-uses test suite reduction. In *Proc. ICSE*, pages 106–115, 2004.

[6] T. Y. Chen and M. F. Lau. A new heuristic for test suite reduction. *IST*, 40(5-6):347–354, 1998.

[7] Y. Chen, D. Rosenblum, and K. Vo. TestTube: A system for selective regression testing. In *Proc. ICSE*, pages 113–124, 1994.

[8] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. An empirical study of the effect of time constraints on the cost-benefits of regression testing. In *Proc. FSE*, pages 71–82, 2008.

[9] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE TSE*, 32(9):733–752, 2006.

[10] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *Proc. ISSRE*, pages 113–124, 2004.

[11] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proc. ISSTA*, pages 102–112, 2000.

[12] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proc. ICSE*, pages 329–338, 2001.

[13] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE TSE*, 28(2):159–182, 2002.

[14] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM TOSEM*, 2(3):270–285, 1993.

[15] S. Hou, L. Zhang, T. Xie, and J. Sun. Quota-constrained test-case prioritization for regression testing of service-centric systems. In *Proc. ICSM*, pages 257–266, 2008.

[16] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE TSE*, 29(3):195–209, 2003.

[17] J. M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proc. ICSE*, pages 119–129, 2002.

[18] B. Korel, G. Koutsogiannakis, and L. Tahat. Application of system models in regression test suite prioritization. In *Proc. ICSM*, pages 247–256, 2008.

[19] B. Korel, L. Tahat, and M. Harman. Test prioritization using system models. In *Proc. ICSM*, pages 559–568, 2005.

[20] H. Leung and L. White. A study of integration testing and software regression at the integration level. In *Proc. ICSM*, pages 290–300, 1990.

[21] Z. Li, M. Harman, and R. Hierons. Search algorithms for regression test case prioritisation. *IEEE TSE*, 33(4):225–237, 2007.

[22] S. Lin. Computer solutions of the travelling salesman problem. *Bell System Technical Journal*, 44(5):2245–2269, 1965.

[23] A. Malishevsky, J. R. Ruthru, G. Rothermel, and S. Elbaum. Cost-cognizant test case prioritization. Technical report, Department Computer Science and Engineering of University of Nebraska, 2006.

[24] N. Mansour and K. El-Fakin. Simulated annealing and genetic algorithms for optimal regression testing. *Journal of Software Maintenance: Research and Practice*, 11(1):19–34, 1999.

[25] I. Moore. Jester-a JUnit test tester. In *Proc. International Conference on Extreme Programming and Flexible Processes*, pages 84–87, 2001.

[26] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *Proc. ICSE*, pages 100–107, 1993.

[27] H. Park, H. Ryu, and J. Baik. Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing. In *Proc. SSIRI*, pages 39–46, 2008.

[28] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Proc. ISSTA*, pages 75–86, 2008.

[29] T. Ralphs and M. Guzelsoy. The SYMPHONY callable library for mixed integer programming. In *Proc. INFORMS Computing Society Conference*, pages 61–73, 2005.

[30] C. R. Reeves. *Modern Heuristic technqiues for combinatorial problems*. John Wiley & Sons, 1993.

[31] G. Rothermel and M. Harrold. A safe, efficient regression test selection technique. *ACM TOSEM*, 6(2):173–210, 1997.

[32] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *Proc. ICSM*, pages 179–188, 1999.

[33] G. Rothermel, R. J. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE TSE*, 27(10):929–948, 2001.

[34] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proc. ISSTA*, pages 97–106, 2002.

[35] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Time aware test suite prioritization. In *Proc. ISSTA*, pages 1–11, 2006.

[36] H. Williams. *Model Building in Mathematical Programming*. John Wiley, New York, 1993.

[37] W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proc. ISSRE*, pages 230–238, 1997.

[38] S. Yoo and M. Harman. Pareto efficient multi objective test case selection. In *Proc. ISSTA*, pages 140–150, 2007.

[39] H. Zhong, L. Zhang, and H. Mei. An experimental study of four typical test suite reduction techniques. *IST*, 50(6):534–546, 2008.