PAPER Special Section on Software Reliability Engineering

Application of Markov Chain Monte Carlo Random Testing to Test Case Prioritization in Regression Testing

Bo ZHOU[†], ***Nonmember***, **Hiroyuki OKAMURA**[††a)], ***and*** **Tadashi DOHI**[††], ***Members***

SUMMARY This paper proposes the test case prioritization in regression testing. The large size of a test suite to be executed in regression testing often causes large amount of testing cost. It is important to reduce the size of test cases according to prioritized test sequence. In this paper, we apply the Markov chain Monte Carlo random testing (MCMC-RT) scheme, which is a promising approach to effectively generate test cases in the framework of random testing. To apply MCMC-RT to the test case prioritization, we consider the coverage-based distance and develop the algorithm of the MCMC-RT test case prioritization using the coverage-based distance. Furthermore, the MCMC-RT test case prioritization technique is consistently comparable to coverage-based adaptive random testing (ART) prioritization techniques and involves much less time cost.

key words: *regression testing, test case prioritization, random testing, Markov chain Monte Carlo*

1. Introduction

Effective software testing schemes are strongly required to make highly reliable software system. In general, software testing schemes depend on how to make test cases to find as-yet-discovered failures. On the other hand, after finding software bugs in the testing, we carefully fix them without the risk of adversely affecting system functionality. That is, in some cases, we make new software faults when removing the bugs. The regression testing is one of the effective techniques to reduce such risk [1]. The regression testing executes a test suite after fixing software bugs. For instance, the retest-all strategy executes all available test cases [2], [3].

To design the effective regression testing, there are major four problems [1]: the regression test selection problem, the coverage identification problem, the test suite execution problem and the test suite maintenance problem. The regression test selection problem is to select a subset of all the test cases to test modified functionality. The coverage identification problem is to check if additional testing is required or not. The test suite execution problem is how to execute the test cases and to check the test results effectively. Finally, the test suite maintenance is the problem of updating and storing test information. This paper focuses on the regression test selection problem, and it strongly affects the testing cost incurred by the regression testing. Researchers have proposed various methods for improving the cost effectiveness of regression testing [4], [5]. Regression test selection techniques reduce testing costs by selecting a subset of test cases from all the test cases to execute on a specific functions of the program. These techniques reduce costs by reducing testing time, but unless they are safe [5], they can omit test cases that would otherwise have detected faults. This can raise the costs of software.

Test case prioritization is one of the techniques to reduce the test cases to be executed [6], [7], which is to give priorities to all the cases in a test suite. The test case prioritization also offers an alternative solution to improve regression testing cost effectiveness.

A variety of prioritization techniques have been proposed. Most techniques are based on code coverage information to give the priority for test cases. Greedy algorithms [8] are a class of coverage-based test case prioritization techniques that have been widely studied in the public literature, which include the total-statement coverage technique and the additional-statement coverage technique [9]. Also, some of dynamic testing strategies were applied to test case prioritization in the regression testing. The dynamic testing strategy is a technique that changes the probability distributions to generate test cases at every time when a test outcome is observed. Although the dynamic testing strategy is used for test case generation in the ordinary software testing like unit testing, Jiang et al. [10] and Zhou [11] discussed the applicability of the adaptive random testing (ART), which is a kind of dynamic testing strategies, to the test case prioritization even in the regression testing. Empirical results in [10], [11] have shown that these prioritization techniques were useful and could improve the performance of random ordering drastically.

This paper discusses the applicability of Markov chain Monte Carlo random testing (MCMC-RT) to the test case prioritization, which is an alternative random-coverage-based algorithm. Zhou et al. [12], [13] proposed the concept of MCMC-RT in the framework of random testing for test case generation. The basic idea of MCMC-RT is to estimate the distribution of fault location from the past outcomes of software test execution, and, as a result, it leads to test cases as evenly as possible across the input domain. MCMC-RT improves the fault-detection capability of RT and ART in terms of using fewer test cases to detect the first fault (the F-measure) [13]. In particular, this paper proposes an MCMC-RT scheme based on the distance between

Manuscript received December 25, 2011.
Manuscript revised April 20, 2012.

[†]The author is with the Department of Computer Science and Engineering, University of California Riverside, Riverside, CA 92521.
[††]The authors are with the Department of Information Engineering, Graduate School of Engineering, Hiroshima University, Higashihiroshima-shi, 739–8527 Japan.
a) E-mail: okamu@rel.hiroshima-u.ac.jp

DOI: 10.1587/transinf.E95.D.2219

test cases measured by coverage information and conducts empirical evaluation to investigate the effectiveness of the proposed method.

The paper is organized as follows. Section 2 presents our MCMC-RT scheme. Section 3 describes the test case prioritization and coverage-based distance between test cases. Furthermore we explain how to use MCMC-RT to the test prioritization. Section 4 presents our empirical study. Some remarks and future studies are presented in Sect. 5.

2. MCMC-RT Algorithm

In this section, we introduce the MCMC-RT algorithm used for test case generation proposed in [12], [13]. The MCMC-RT is a random testing scheme to generate test cases that can find latent failures effectively from the information on input domain and outcomes of the executed test cases. As mentioned before, the idea behind MCMC-RT is to estimate the distribution of fault location based on the Bayes statistics.

Let $\mathcal{D}$ denote an input domain that can be regarded as a metric space with d degrees of dimension. Then $\boldsymbol{x} = (x_1, \ldots, x_d) \in \mathcal{D}$ represents a vector of a test case. For example, $\boldsymbol{x}$ becomes parameters for a function to be tested.

Define the function representing testing outcomes with respect to the input domain $\mathcal{D}$:

$$T(\boldsymbol{x}) = \begin{cases} 0, & \boldsymbol{x} \text{ is not a fault-detecting test case,} \\ 1, & \boldsymbol{x} \text{ is a fault-detecting test case.} \end{cases} \tag{1}$$

Then the software testing activity can be described as the search of fault-detecting test cases $\mathcal{D}_f = \{\boldsymbol{x} \in \mathcal{D}; T(\boldsymbol{x}) = 1\}$.

In the MCMC-RT scheme, we consider the posterior probability that a test case $\boldsymbol{x}$ has a failure after obtaining m test outcomes. Let $\tau_1 = T(\boldsymbol{x}_1), \ldots, \tau_m = T(\boldsymbol{x}_m)$ be test outcomes for already executed m test cases. According to Bayes rule, we have

$$\begin{aligned} &P(T(\boldsymbol{x}) = 1 | \tau_1, \ldots, \tau_m) \\ &\quad = \frac{p(\tau_1, \ldots, \tau_m | T(\boldsymbol{x}) = 1) P(T(\boldsymbol{x}) = 1)}{Z}, \end{aligned} \tag{2}$$

where Z is a normalizing constant. Assuming $\tau_1, \ldots, \tau_m$ are conditional independence for a test outcome of the test case $\boldsymbol{x}$, the numerator of Eq. (2) is given by

$$p(\tau_1, \ldots, \tau_m | T(\boldsymbol{x}) = 1) = \prod_{i=1}^{m} p(\tau_i | T(\boldsymbol{x}) = 1). \tag{3}$$

The normalizing constant is given by

$$\begin{aligned} Z = &\prod_{i=1}^{m} p(\tau_i | T(\boldsymbol{x}) = 0) P(T(\boldsymbol{x}) = 0) \\ &+ \prod_{i=1}^{m} p(\tau_i | T(\boldsymbol{x}) = 1) P(T(\boldsymbol{x}) = 1). \end{aligned} \tag{4}$$

In the above equation, the conditional probabilities $p(\tau_i | T(\boldsymbol{x}) = 0)$ and $p(\tau_i | T(\boldsymbol{x}) = 0)$ are important to estimate the posterior distribution of fault location. However, it is impossible to find the true conditional probabilities, because they strongly depend on source codes and program logic of the software to be tested. In [13], Zhou et al. provided a rational and useful assumption to the conditional probabilities. Concretely, the conditional probabilities $p(\tau_i | T(\boldsymbol{x}) = 0)$ and $p(\tau_i | T(\boldsymbol{x}) = 1)$ are given as the functions of a metric (distance) on input space, i.e.,

$$p(T(\boldsymbol{x}') = 0 | T(\boldsymbol{x}) = 0) = \exp\left(-\frac{D(\boldsymbol{x}, \boldsymbol{x}')}{\beta_0}\right), \tag{5}$$

$$p(T(\boldsymbol{x}') = 1 | T(\boldsymbol{x}) = 1) = \exp\left(-\frac{D(\boldsymbol{x}, \boldsymbol{x}')}{\beta_1}\right), \tag{6}$$

where $D(\boldsymbol{x}, \boldsymbol{x}')$ is the distance between two test cases $\boldsymbol{x}$ and $\boldsymbol{x}'$. Equations (5) and (6) imply that the correlation between outcomes of two test cases exponentially decreases as the distance of the test cases increases. The parameters β_0 and β_1 correspond to the strengths of test case correlations. In [13], we assumed $\beta = \beta_0 = \beta_1$. Additionally β was determined by the maximum distance on the input domain, and also $P(T(\boldsymbol{x}) = 0)$ and $P(T(\boldsymbol{x}) = 1)$ were given by flat distributions, i.e., constants. From Eq. (2), the fault location is given by the following distribution

$$f_{fail}(\boldsymbol{x}) = \frac{\prod_{i=1}^{m} p(\tau_i | T(\boldsymbol{x}) = 1) P(T(\boldsymbol{x}) = 1)}{\int \prod_{i=1}^{m} p(\tau_i | T(\boldsymbol{x}) = 1) P(T(\boldsymbol{x}) = 1) d\boldsymbol{x}}. \tag{7}$$

The test case generation of MCMC-RT is to make a sample drawn from $f_{fail}(\boldsymbol{x})$ and is realized by MCMC algorithm. Concretely, the procedure of MCMC-RT until the first failure is discovered is presented as follows[†]:

- **Step 1:** Set an executed set to be empty set; $\mathcal{S}_E \leftarrow \phi$.
- **Step 2:** Generate an initial test case which is randomly selected from the input domain and execute it. If no fault is detected, add this executed test case to $\mathcal{S}_E$. Otherwise, stop the procedure.
- **Step 3:** Generate a new test case $\boldsymbol{x}$ based on the set of executed test cases $\mathcal{S}_E$ according to the following steps:
  - **Step 3-1:** Generate an initial test case $\boldsymbol{x}$ which is a uniform random variable on the input domain.
  - **Step 3-2:** Generate a new candidate $\boldsymbol{x}'$ according to the uniform distribution.
  - **Step 3-3:** If an uniform random number U becomes less than or equal to the acceptance probability P_α, i.e.,

$$U \leq P_\alpha = \min\left(\frac{f_{fail}(\boldsymbol{x}')}{f_{fail}(\boldsymbol{x})}, 1\right), \tag{8}$$

    $\boldsymbol{x}'$ is accepted as a new test case; $\boldsymbol{x} \leftarrow \boldsymbol{x}'$. Otherwise, if U becomes more than the acceptance probability, then the test case is not updated.
  - **Step 3-4:** Execute Step 3-2 through Step 3-3 for

[†]The original paper proposed two algorithms. This paper focuses only on MCMC-RT1 in [13] because it was superior to MCMC-RT2 in terms of the failure-finding ability.

the fixed number of iterations, and return $\boldsymbol{x}$ as a generated test case.

- **Step 4:** Execute the test case $\boldsymbol{x}$ generated in Step 3. If no fault is detected, add $\boldsymbol{x}$ to $\mathcal{S}_E$ and go to Step 3. Otherwise, if a fault is detected, stop the procedure.

3. Test Case Prioritization Using MCMC-RT

3.1 Test Case Prioritization Using Random Strategy

The test case prioritization is to give the priorities of test cases which are in a test suite, and is effective to reduce testing cost especially incurred by the regression testing. Most of test case prioritization techniques are based on the information of coverages such as code and branch coverages. Generally speaking, these methods require high computation cost to select the test cases from the test suite, since they are essentially same as the optimization problem.

On the other hand, the simplest and cost-effective approach to the test case prioritization is a class of random strategies, which is to choose a subset of the test cases from a test suite randomly. The weakness of such the random selection is clearly to choose the useless test cases that are not related to the modified functions in the regression testing. In other words, we should consider the coverage information even in the test case selection according to random strategy.

Jiang et al. [10] and Zhou [11] presented the test case prioritization with such coverage information based on the adaptive random testing (ART). ART was originally proposed to generate test cases in the ordinary software testing. However, the idea behind ART can directly be used for the random strategy of test case prioritization. In their schemes, test cases are randomly generated until one of the samples increases a coverage, and one of the test cases is selected such that it maximizes a distance function with the already selected test cases. This distance function can be either the minimum distance with all executed tests, the maximum distance, or the average distance. The ART with such distance function is called distance-based ART (DART). Empirical results have shown that DART prioritization techniques were useful and could improve the performance of random ordering drastically. This paper examines the applicability of MCMC-RT scheme to the test case prioritization under the similar manner of ART-based test case prioritization.

3.2 Distance Measure

While applying the MCMC-RT algorithm to the test case prioritization, we should decide the distance measure between two test cases based on the coverage information. The distance used in [13] is based on the metric on input domain. Since it does not include the information on the coverage, it cannot be used in the test case prioritization. There are many ways to measure the distance between two test cases. This paper uses *Coverage Manhattan Distance* (CMD) to measure the difference between any two test cases [11].

Let $\boldsymbol{x} = (x_1, \ldots, x_n)$ and $\boldsymbol{y} = (y_1, \ldots, y_n)$ be coverage vectors for elements of two programs. In general, an element can be defined as a node or an edge in the program control flow graph, data flow graph and other types of graphs. Examples of elements are statements, branches and conditions. The coverage vector is defined as a vector whose element takes 0 or 1. In the vector, 0 means that the corresponding element has not been covered yet, 1 means that the element has already been covered. Then the CMD can be obtained as the following equation:

$$CMD(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{n} |x_i - y_i|. \tag{9}$$

In particular, this paper focuses on the CMD using the branch coverage.

3.3 Algorithm

Suppose that branch coverage of each test case in a test suite is measured and that the corresponding branch coverage vectors are given. Then the algorithm for test case prioritization using MCMC-RT in the regression testing is written as follows. Although the MCMC-RT with test case generation [13] is to generate a new test case, the following algorithm is to select a test case from a test suite.

- **Step 1:** Set an executed set to be empty set; $\mathcal{S}_E \leftarrow \phi$.
- **Step 2:** Select an initial test case from the test suite randomly and execute it. If no fault is detected, add this executed test case to $\mathcal{S}_E$. Otherwise, stop the procedure.
- **Step 3:** Select a new test case $\boldsymbol{x}$ from the test suite based on the set of executed test cases $\mathcal{S}_E$:
  - **Step 3-1:** Select a new test case $\boldsymbol{x}$ from the test suite according to the uniform distribution.
  - **Step 3-2:** Select a new candidate (test case) $\boldsymbol{x}'$ from the test suite according to the uniform distribution.
  - **Step 3-3:** If an uniform random number U becomes less than or equal to the acceptance probability P_α, i.e.,

$$U \leq P_\alpha = \min\left(\frac{f_{fail}(\boldsymbol{x}')}{f_{fail}(\boldsymbol{x})}, 1\right), \tag{10}$$

    $\boldsymbol{x}'$ is accepted as a new test case; $\boldsymbol{x} \leftarrow \boldsymbol{x}'$. Otherwise, if U becomes more than the acceptance probability, then the test case is not updated.
  - **Step 3-4:** Execute Step 3-2 through Step 3-3 for the fixed number of iterations, and return $\boldsymbol{x}$ as a selected test case.
- **Step 4:** Execute the test case $\boldsymbol{x}$ selected in Step 3. If no fault is detected, add $\boldsymbol{x}$ to $\mathcal{S}_E$ and go to Step 3. Otherwise, if a fault is detected, stop the procedure.

In the calculation of $f_{fail}(\boldsymbol{x})$, the CMD of branch coverage

is used. The advantage of this scheme is to choose less test cases that cover all the branches than in the case of random strategy. In the MCMC-RT scheme, if one test case is chosen and is executed without a failure, the probability that one or more faults are located around the already executed test cases is expected to be small. Then the next selection is a test case that is far from the previous one in terms of the branch coverage distance.

4. Empirical Study

4.1 Subject Programs

To assess the fault-detection capability of the proposed method, five well-known subject programs were used to conduct empirical experiments. Table 1 provides the information about the programs used in our experiments. In the table, LOC indicates the lines-of-code of non-faulty version.

The first four programs in Table 1 are the Siemens suite of programs downloaded from the Software-artifact Infrastructure Repository (http://sir.unl.edu). These programs, with faulty versions and test cases, were assembled by researchers at Siemens Corporate Research for a study of the fault detection capabilities of control-flow and data-flow coverage criteria [2]. The Siemens programs perform various tasks: *replace* performs pattern matching and substitution, *print_tokens2* is lexical analyzer, *schedule2* is a priority scheduler and *tot_info* computes statistics.

Each version of these programs contains a single fault. In this context, the use of single-fault versions is an important experiment design that allows experimenters to precisely determine whether a test case reveals a particular fault simply by determining whether the version containing that fault fails. In the absence of this methodology, it may be difficult or impossible to associate test cases with particular faults.

The fifth program *space* is a program developed for the European Space Agency [14], downloaded from the Software-artifact Infrastructure Repository (http://sir.unl.edu). *space* consists of 5905 lines of code, and functions as an interpreter for an array definition language (ADL). The program reads a file that contains several ADL statements, and checks the contents of the file for adherence to the ADL grammar and to specific consistency rules. If the ADL file is correct, *space* outputs an array data file containing a list of array elements, positions, and excitations; otherwise the program outputs error messages. *space* has 38 associated versions, each containing a single fault that had been discovered during the program's development.

Table 1 Subject programs.

Program Name	Faulty Versions	LOC	Number of Test Cases
replace	32	513	5542
print_tokens2	10	355	4115
schedule2	10	266	2710
tot_info	23	297	1052
space	38	5905	13585

4.2 Experiment Procedure

As introduced above, each subject program package includes a base version (non-faulty version), associated faulty versions, and a suite of test cases. For each subject program, the experiment was conducted as follows: we first run the base version using all the provided test cases. During each test case execution of the base version, the Linux utility *gcov* (a standard test coverage tool in concert with *gcc*) was used to collect branch coverage data. Outputs of the base version were also recorded as the test oracle. By comparing the test oracle with outputs of the faulty versions, we detect the error occurrence.

Table 2 Results of F-measure.

	RT	DART	MCMCRT	
			$\beta = M/2$	$\beta = M/4$
replace	249.61	133.29	141.48	144.39
print_tokens2	64.08	14.70	10.64	16.85
schedule2	115.93	54.55	48.43	56.15
tot_info	25.38	11.60	12.63	15.06
space	172.75	46.52	36.33	37.64
average	125.55	52.13	49.90	54.02

Table 3 Results of observed mean F-measure with program *replace*.

	RT	DART	MCMCRT	
			$\beta = M/2$	$\beta = M/4$
v1	82.25	38.41	27.30	60.26
v2	173.92	80.23	31.17	100.13
v3	42.40	12.34	14.28	36.41
v4	39.45	11.43	14.01	34.64
v5	19.04	14.56	13.18	18.93
v6	56.62	52.55	73.66	49.09
v7	72.24	24.59	24.13	54.10
v8	108.82	44.08	23.48	78.88
v9	159.33	169.93	360.41	137.82
v10	177.00	154.43	216.41	151.94
v11	159.33	169.93	360.41	137.82
v12	17.21	16.64	18.36	16.91
v14	33.40	18.84	21.12	35.97
v15	95.38	81.35	21.96	69.28
v16	72.24	24.59	24.13	54.10
v17	252.04	222.65	84.76	141.55
v18	24.19	17.39	31.00	24.78
v19	1476.68	589.12	262.42	763.25
v20	286.09	244.70	87.09	151.86
v21	1607.62	696.07	1322.68	831.57
v22	275.51	251.39	654.62	175.67
v24	29.66	13.81	11.68	27.05
v25	1535.30	710.18	177.23	733.00
v27	19.71	9.52	9.09	18.17
v28	42.80	12.23	13.47	32.87
v29	85.75	26.11	23.04	66.63
v30	21.03	7.69	9.27	15.37
v31	24.19	17.39	31.00	24.78
average	249.61	133.29	141.48	144.39

Table 4 Results of observed mean F-measure with program *print_tokens2*.

	RT	DART	MCMCRT	
			$\beta = M/2$	$\beta = M/4$
v1	33.51	10.06	6.71	8.40
v2	32.08	9.66	6.49	8.24
v3	149.09	23.72	18.37	38.16
v4	38.52	10.42	7.29	9.73
v5	82.67	14.40	12.75	21.89
v6	12.35	4.46	4.41	5.01
v7	33.51	16.97	12.31	18.47
v8	29.52	8.45	6.93	8.66
v9	146.92	34.50	18.37	28.08
v10	82.67	14.40	12.75	21.89
average	64.08	14.70	10.64	16.85

Table 5 Results of observed mean F-measure with program *schedule2*.

	RT	DART	MCMCRT	
			$\beta = M/2$	$\beta = M/4$
v1	40.55	44.33	24.95	38.27
v2	90.68	71.35	73.40	80.74
v3	69.71	89.48	74.21	50.97
v5	78.04	59.40	71.71	52.68
v6	436.43	70.94	47.42	97.40
v7	90.68	71.35	73.40	80.74
v8	60.68	14.32	11.08	24.13
v10	60.68	15.19	11.29	24.23
average	115.93	54.55	48.43	56.15

Table 6 Results of observed mean F-measure with program *tot_info*.

	RT	DART	MCMCRT	
			$\beta = M/2$	$\beta = M/4$
v1	7.26	4.47	4.57	5.56
v2	96.18	27.79	33.93	43.33
v4	26.89	17.12	19.23	17.59
v5	34.55	14.90	17.57	21.75
v6	23.04	18.48	19.50	19.76
v7	8.85	4.62	6.10	7.71
v8	5.91	3.58	3.94	4.14
v9	28.31	13.01	15.77	18.23
v10	121.39	33.90	27.88	50.77
v11	5.91	3.58	3.94	4.14
v12	36.81	17.70	21.66	18.89
v13	8.52	4.47	5.91	7.42
v15	5.91	3.58	3.94	4.14
v16	6.37	3.98	4.98	5.98
v17	20.23	15.55	17.07	16.22
v18	9.84	5.03	5.73	6.93
v19	11.64	12.36	11.23	13.36
v20	14.53	6.70	7.84	9.67
v21	11.66	6.37	8.36	9.84
v22	37.79	18.73	17.30	20.07
v23	11.48	7.69	8.77	10.75
average	25.38	11.60	12.63	15.06

Then, for each faulty version, we consider three test case sequences (permutations) of all the test cases. The three test case sequences are generated by the following random strategies: the ordinary random testing (RT), the distance-based ART (DART) and MCMC-RT. The last two strategies use the branch coverage Manhattan distance as the difference measure between test cases. Note that the coverage data were collected from the base version. The F-measures under the test case sequences of RT, DART and MCMC-RT were recorded, where the F-measure is defined as the number of executed test cases until an error is detected. For each faulty version, this process was repeated 100 times. Also, according to the experiment in [13], we set the parameter β in MCMC-RT as $\beta = M/2$ and $\beta = M/4$, where M is the maximum distance. In our experiment, M can be given by the number of branches (length of coverage vectors). The numbers of branches for *replace*, *print_tokens2*, *schedule2*, *tot_info* and *space* are 180, 162, 88, 88 and 1,190, respectively.

We carry out the experiment on a Dell Power Edge T605 server serving a CentOS 5.3 Linux OS. The server is equipped with AMD Opteron 6276 (2.3 GHz, 8 core) processor with 16 GB physical memory.

Table 7 Results of observed mean F-measure with program *space*.

	RT	DART	MCMCRT	
			$\beta = M/2$	$\beta = M/4$
v3	19.53	25.31	23.10	21.02
v4	1.16	1.23	1.23	1.24
v5	3.28	3.54	4.07	4.26
v6	1.13	1.19	1.18	1.18
v7	76.93	22.21	16.76	19.83
v8	109.71	46.30	49.87	53.48
v9	3.40	2.97	3.22	3.22
v10	9.99	7.49	8.27	8.44
v11	13.49	9.94	11.13	10.35
v12	928.16	204.44	134.47	130.36
v13	17.87	8.18	9.96	10.63
v14	7.31	4.61	5.38	5.74
v15	4.51	3.05	3.79	3.94
v16	30.69	8.30	8.39	9.77
v17	70.54	78.40	49.93	43.73
v18	928.16	204.44	134.47	130.36
v19	9.57	9.71	10.14	10.13
v20	66.50	21.82	22.72	23.78
v21	66.50	21.82	22.72	23.78
v22	365.55	61.82	34.61	41.13
v23	42.51	16.28	13.74	14.83
v24	20.19	6.62	7.48	8.29
v25	3.42	2.95	3.27	3.28
v26	8.75	5.10	5.67	5.74
v27	680.74	117.55	36.77	49.13
v28	2.17	2.09	2.25	2.26
v29	18.69	10.87	12.04	12.47
v30	1.31	1.45	1.51	1.52
v31	8.63	5.93	6.52	6.77
v33	1692.08	399.67	104.28	135.25
v35	65.01	19.50	69.79	68.71
v36	131.15	135.22	34.98	36.17
v37	112.57	38.49	24.26	28.38
v38	352.21	73.21	357.24	350.43
average	172.75	46.52	36.33	37.64

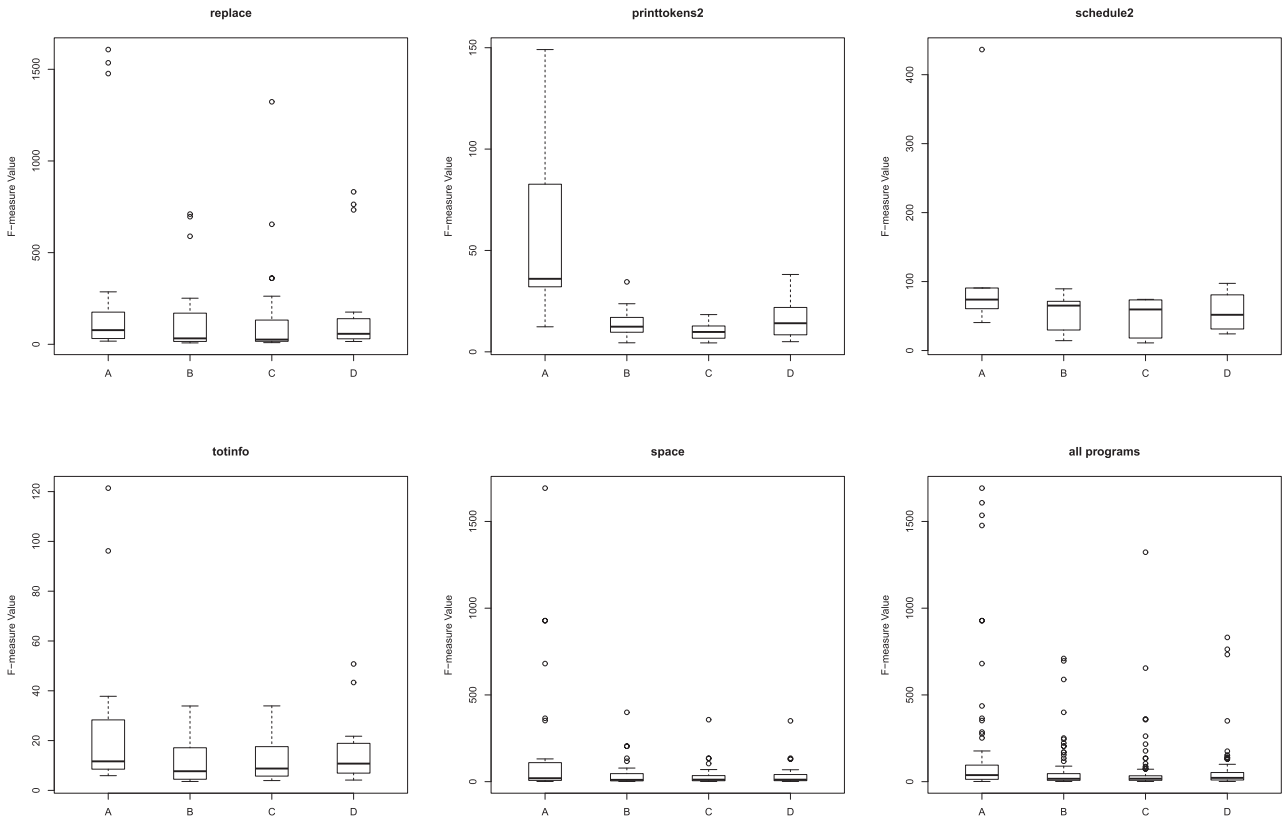


Fig. 1 F-measure distributions for each program and all the programs. A, B, C, D represent RT, DART, MCMC-RT($\beta = M/2$) and MCMC-RT($\beta = M/4$) respectively.

4.3 Results and Discussion

Results of all experiments are shown in Table 2. Table 2 shows that MCMC-RT ($\beta = M/2$ or $\beta = M/4$) outperformed RT and DART in the case of *print_token2*, *schedule2*, *space*. The highest saving occurs for program *print_tokens2*, for which the ratios of MCMC-RT($\beta = M/2$)/RT and MCMC-RT($\beta = M/2$)/DART are 16.6% and 72.4%, which means that MCMC-RT used about 83.4% and 27.6% fewer test cases than RT and DART to detect the first fault, respectively. Furthermore, in the average of five programs, MCMC-RT with $\beta = M/2$ is superior to the other strategies.

Detailed results of experiments for each program are shown in Tables 3–7. In these tables, we give the average F-measure value for each version of each program after 100 times execution.

The *replace* package includes 32 faulty versions. Version 32 is excluded from the experiments because it generated identical outputs as the base version on all the 5,542 test cases. Furthermore, versions 13, 23, and 26 are not stable as each of them produced different outputs (hence different sets of fault-detecting test cases) when run at different times or under different environments. We therefore also excluded these versions from the experiments. For program *print_tokens2* shown in Table 4, we use all the 10 faulty versions. Also, we excluded the version 4 of program *schedule2* given in Table 5 and versions 3 and 14 of program *tot_info* given by Table 6 from the simulation since they cannot execute properly under our experiment environment. Furthermore, version 9 of program *schedule2* is excluded from the experiments since it produced the same output as the base version. For the *space* program shown in Table 7, versions 1, 2, 32, and 34 are excluded from the experiments because they produced identical outputs as the base version.

According to Tables 3–7, we find that our MCMC-RT scheme can provide better fault detection performance than both RT and DART in many cases. Only for program *tot_info* and some versions of *replace* and *space*, MCMC-RT is worse performed than RT or DART, but the difference between MCMC-RT and DART in this program is marginal. The reason is that the fault detection rate in such program is too high and it causes the decrease of fault detection capability.

Figure 1 shows the box-whisker plots[†] of F-measure distributions for each and overall programs. We observe that MCMC-RT prioritization performs better than both random ordering and comparable with the ART techniques.

[†]Boxplots provide a concise display of a distribution. The central line in each box marks the median value. The edges of the box mark the first and third quartiles. The whiskers extend from the quartiles to the farthest observation lying within 1.5 times the distance between the quartiles. Individual markers beyond the whiskers are outliers.

Table 8 Comparing difference between F-measures.

Comparison	replace	print_tokens2	schedule2	tot_info	space	all programs
RT: DART	<	<	=	<	<	<
RT: MCMCRT($\beta = M/2$)	=	<	=	<	<	<
RT: MCMCRT($\beta = M/4$)	<	<	<	<	<	<
DART: MCMCRT($\beta = M/2$)	=	=	=	>	=	=
DART: MCMCRT($\beta = M/4$)	=	<	=	>	=	=
MCMCRT($\beta = M/4$): MCMCRT($\beta = M/2$)	=	<	=	<	<	<

Table 9 Results of computation time (in seconds).

	RT	DART	MCMCRT	
			$\beta = M/2$	$\beta = M/4$
replace	0.028	6.43	20.56	15.56
print_tokens2	0.005	0.39	2.23	1.29
schedule2	0.005	0.25	0.86	1.70
tot_info	0.005	0.059	0.61	1.06
space	0.026	5.46	12.24	14.34

Next we conduct signed tests to compare the performance of RT, DART and MCMC-RT. The signed test is one of the non-parametric statistical tests to reveal whether there is a difference between two experimental outcomes. Table 8 presents the results of signed tests. We perform the signed tests to two experimental outcomes. For example, the first row indicates the results for comparison of F-measures of RT and DART. If the null hypothesis; F-measures of RT and DART are same, under the alternative hypothesis; F-measures of DART are less than that of RT, is rejected, we put "<" to the corresponding result. On the other hand, If the null hypothesis under the alternative hypothesis; F-measures of RT are less than that of DART, is rejected, we put ">" to the corresponding result. Also, "=" indicates that the null hypothesis is not rejected. In our experiment, the significant level is set as 0.01. Additionally, to avoid the statistical problem that the rejection of null hypothesis occurs optimistically in the case of multiple comparisons, we used the Bonferroni method which divides the significant level by the number of statistical tests.

From Table 8, it is clear that MCMC-RT and DART always perform better than or equal to random testing. Also, we can find that for all the five programs and the overall results, the performance of our MCMC-RT and DART are generally comparable to each other.

As a matter of fact, we further analyze the time cost of MCMC-RT prioritization techniques and compare them with RT and DART techniques to help guide practical use. Table 9 presents the time cost (user time in seconds with a single thread) of prioritization techniques in five subject programs. We calculate the mean prioritization time across all techniques for every subject program. The computation time is a sum of generating and executing 100 test cases, which excludes the time for measuring the coverage. We observe that the MCMC-RT spends more time cost than RT and DART.

5. Conclusion

In this paper, we have proposed an MCMC-RT algorithm for the test case prioritization. In particular, we have discussed the distance measure based on the coverage and how to incorporate it into the MCMC-RT scheme. Empirical studies were further conducted for the branch coverage Manhattan distance measure. The experimental results show that, the proposed method improves the fault-detection capability of random testing significantly.

The proposed method can directly be used to prioritize regression test cases. Future research should study the usefulness of other types of coverage information for MCMC-RT, including coverage of elements in function call graphs and in other types of graphs/models used in modeling systems, such as state diagrams and other unified modeling language (UML) diagrams. Furthermore, we plan to apply our MCMC-RT technique to test suite augmentation [15] based on program dependence graph (PDG) [16].

Acknowledgements

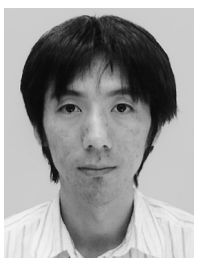
This research was supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research (C), Grant No. 21510167 (2009–2011), Grant No. 23500047 (2011–2013) and Grant No. .23510171 (2011–2013).

References

[1] A.K. Onoma, W.T. Tsai, M. Poonawala, and H. Suganuma, "Regression testing in an industrial environment," Commun. ACM, vol.41, pp.81–86, May 1998.
[2] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," Proc. 16th International Conference on Software Engineering, ICSE '94, pp.191–200, Los Alamitos, CA, 1994.
[3] M.K. Ramanathan, M. Koyuturk, A. Grama, and S. Jagannathan, "Phalanx: A graph-theoretic framework for test case prioritization," Proc. 2008 ACM Symposium on Applied Computing, SAC '08, New York, NY, pp.667–673, 2008.
[4] G. Rothermel and M.J. Harrold, "Analyzing regression test selection techniques," IEEE Trans. Softw. Eng., vol.22, no.8, pp.529–551, Aug. 1996.
[5] G. Rothermel and M.J. Harrold, "A safe, efficient regression test selection technique," ACM Trans. Softw. Eng. Methodol., vol.6, no.2, pp.173–210, April 1997.
[6] G. Rothermel, R.J. Untch, and C. Chu, "Prioritizing test cases for regression testing," IEEE Trans. Softw. Eng., vol.27, no.10, pp.929–948, Oct. 2001.
[7] S. Elbaum, G. Rothermel, S. Kanduri, and A.G. Malishevsky, "Selecting a cost-effective test case prioritization technique," Software Quality Control, vol.12, pp.185–210, Sept. 2004.

[8] Z. Li, M. Harman, and R.M. Hierons, "Search algorithms for regression test case prioritization," IEEE Trans. Softw. Eng., vol.33, no.4, pp.225–237, April 2007.
[9] S. Elbaum, A.G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," IEEE Trans. Softw. Eng., vol.28, no.2, pp.159–182, Feb. 2002.
[10] B. Jiang, Z. Zhang, W.K. Chan, and T.H. Tse, "Adaptive random test case prioritization," Proc. 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09, pp.233–244, Washington, DC, 2009.
[11] Z.Q. Zhou, "Using coverage information to guide test case selection in adaptive random testing," Computer Software and Applications Conference Workshops, pp.208–213, Los Alamitos, CA, 2010.
[12] B. Zhou, H. Okamura, and T. Dohi, "Markov chain Monte Carlo random testing," Proc. 2010 International Conference on Advances in Computer Science and Information Technology, AST/UCMA/ISA/ACN'10, pp.447–456, Berlin, Heidelberg, 2010.
[13] B. Zhou, H. Okamura, and T. Dohi, "Enhancing performance of random testing through Markov chain Monte Carlo methods," IEEE Trans. Comput. (accepted).
[14] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," Empirical Softw. Eng., vol.10, pp.405–435, Oct. 2005.
[15] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M.B. Cohen, "Directed test suite augmentation: techniques and tradeoffs," Proc. Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, pp.257–266, New York, NY, ACM, 2010.
[16] J. Ferrante, K.J. Ottenstein, and J.D. Warren, "The program dependence graph and its use in optimization," ACM Trans. Program. Lang. Syst., vol.9, pp.319–349, July 1987.

Bo Zhou received the M.S. degree of engineering from Hiroshima University, Hiroshima, Japan, in 2011. He received the B.S. degree of engineering from University of Electronic Science and Technology of China, Chengdu, China, in 2007. He is now a Ph.D. student at the Department of Computer Science and Engineering, University of California Riverside. His research interests include software testing and debugging, software engineering, program language and compilers.



Hiroyuki Okamura received the B.S.E., M.S. and Dr. of Engineering degrees from Hiroshima University, Japan, in 1995, 1997 and 2001, respectively. In 1998 he joined the Hiroshima University as an Assistant Professor, and is now working as an Associate Professor in the Department of Information Engineering, Graduate School of Engineering from 2003. His research areas include Performance Evaluation, Dependable Computing and Applied Statistics. He is a regular member of ORSJ, IPSJ and IEEE.

Tadashi Dohi received the B.S.E., M.S. and Dr. of Eng. degrees from Hiroshima University, Japan, in 1989, 1991 and 1995, respectively. Since 2002, he has been working as a full Professor in the Department of Information Engineering, Graduate School of Engineering, Hiroshima University. In 1992 and 2000, he was a Visiting Researcher in the Faculty of Commerce and Business Administration, University of British Columbia, Canada, and Hudson School of Engineering, Duke University, USA, respectively, on the leave absent from Hiroshima University. His research areas include Reliability Engineering, Software Reliability, Dependable Computing, Performance Evaluation and High Assurance Systems Design. He is a regular member of ORSJ, JAMS, ISCIE, SICE and IEEE.