Learning to Prioritize GUI Test Cases for Android Laboratory Programs

Cheng-Zen Yang, Yuan-Fu Luo, Yu-Jen Chien, Hsiang-Lin Wen
Department of Computer Science and Engineering
Yuan Ze University, Chungli, Taiwan, 320
czyang@syslab.cse.yzu.edu.tw, {s981542,s981522}@mail.yzu.edu.tw,
hlwen13@syslab.cse.yzu.edu.tw

ABSTRACT
Software GUI testing for laboratory programs is a cumbersome and time-consuming task. In the past studies, a systematic approach for the efficiency issue of testing the laboratory work has not been discussed. In this paper, we propose a learning framework based on the RankBoost learning-to-rank approach to facilitate the verification task by learning to prioritize GUI test cases. In the experiments, we collected Android laboratory programs to investigate the effectiveness of the proposed learning framework. The experimental results show that the proposed learning framework can effectively improve the recall performance based on pairwise priority relations.

CCS Concepts
• Software and its engineering → Software testing and debugging; • Computing methodologies → Learning to rank.

Keywords
GUI testing; test case prioritization; learning to rank; RankBoost.

1. INTRODUCTION
As the emergence of smartphones, laboratory courses in embedded software programming are populated worldwide recently. To understand whether students are familiar with the programming techniques for mobile devices, program assessment plays an important role in modern learner-centric software engineering courses [1]. However, verification of laboratory programs is a cumbersome and time-consuming task [2]. One major reason is that the number of enrolled students is rapidly increased and a class may have many students. For example, Muppala has mentioned that the enrollment of an embedded software course in the Hong Kong University of Science and Technology was increased by about 30% when Android was introduced in the course [3]. In addition, students may use different GUI components for their laboratory programs such that the verification process becomes more complicated. These facts lead to a similar conclusion as shown in [4] that having suitable testing techniques and tools for mobile applications is a crucial issue for mobile software development.

ICAIR and CACRE '16, July 13-15, 2016, Kitakyushu, Japan

DOI: http://dx.doi.org/10.1145/2952744.2952755

In the past, research studies on automating GUI testing for smart mobile devices have been conducted to reduce the cost of the software verification work, such as the approaches for Android programs proposed in [5, 6, 7, 8, 9]. However, to the best of our survey, traditional research approaches on automating GUI testing do not consider the efficiency issue of testing laboratory work in an educational environment. On the other hand, research studies on test case prioritization for an individual software application focus on reducing the cost of regression testing, such as the approaches in [10, 11, 12]. These prioritization schemes mainly address the problem of reusing test suites for an individual software application. They do not consider the verification task for a collection of laboratory programs. The laboratory programs under test in a programming course usually satisfy the same laboratory requirements, but many differences may exist because of the diversity of student programming skills. Moreover, some types of common errors may regularly appear in these student programs not only for novice programmers [13, 14, 15] but also for intermediate programmers [16]. The occurrence model of these common errors can therefore be leveraged to speed up the overall verification process by adjusting the sequence of the test cases.

In this paper, we propose a learning framework based on a learning-to-rank approach to facilitate the verification task by prioritizing GUI test cases from the past verification experiences. The framework employs the RankBoost algorithm [17] to dynamically adjust the testing sequence according to the designated testing metrics and the past testing experiences. RankBoost is a statistical learning approach that can rank a collection of objects based on their pairwise relationships. In addition, RankBoost can incorporate the preference data in the learning process to take the benefits of considering the expertise domain knowledge in practice. In the past, RankBoost has been employed to prioritize test cases for individual software programs [12]. In this work, we extend its prioritization scope to a group of student programs.

To verify the effectiveness of the proposed learning framework, we have collected GUI programs of two laboratory assignments in an Android course to conduct experiments. In the experiments, test cases were designed based on the eight error types studied in [5]. The experimental results show that the proposed learning framework can get higher recall performance than a baseline scheme that uses a random testing scheme.

The rest of this paper is organized as follows. Section 2 provides a brief overview of the recent related work on GUI testing for smart mobile devices and laboratory work. Section 3 describes the design of the proposed learning framework. In Section 4, we

describe the experiments of comparing the testing performance of the learning framework with the performance of a random testing scheme. Section 5 concludes the paper and presents the future work.

2. RELATED WORK

In the past, many studies have been conducted for testing laboratory embedded software. For example, Legourski, Trodhandl, and Weiss proposed a test system to verify the functionality of laboratory work of embedded programs in 2005 [2]. Hu and Neamtiu proposed a testing framework to perform GUI testing for Android in 2011 [5]. Amalfitano et al. proposed a toolset to perform GUI testing for Android using a ripping approach to explore the GUI structure and accordingly generate the test cases [6, 7]. However, these studies focus on the testing process for a single program under test. They do not consider the problem of prioritizing test cases for a group of laboratory programs.

The research on test case prioritization has been conducted for several decades. The main purpose of prioritizing test cases is to reduce the cost of repetitively executing test cases in software regression testing [11]. For example, Chen and Lau proposed two dividing strategies to minimize the test suite in 1996 [18]. Binkley also proposed a test case selection scheme to reduce the testing cost based on the language semantics of program code [19]. In 2001, Rothermel el al. formally defined the test case prioritization problem and empirically evaluated six prioritization heuristics compared with three basic prioritization configurations (random, no prioritization, and optimal ordering) on eight C programs [10]. They find that these prioritization heuristics can all have improvements. In 2002, Elbaum et al. extended the work of [10] to conduct more comprehensive empirical studies with more techniques, additional measures, and larger programs [11]. They found that most techniques could have improvements, but some heuristics might not outperform the random ordering. In 2006, Tonella, Avesani, and Susi proposed a machine learning approach called Case-Base Ranking (CBR) [12] to prioritize test cases using the RankBoost algorithm [17]. With a case study program, CBR shows its effectiveness in performance improvements. However, these past studies are mainly for regression testing. Although they have demonstrated the promising progress of prioritization techniques, the issue of prioritizing test cases for a group of laboratory programs has not been discussed.

3. DESIGN OF THE LEARNING FRAMEWORK

In this study, a learning framework is proposed to prioritize a set of GUI test cases for a group of laboratory programs. In the proposed framework, we adopt the RankBoost algorithm [17] to prioritize test cases according to the historical testing experiences. RankBoost is a pairwise learning-to-rank algorithm that can rank a collection of objects by learning the ordering preferences from their historical pairwise relations. Therefore, given a group of n laboratory programs under test $L = \{l_1, l_2, \ldots, l_n\}$, a set of m test cases $T = \{t_1, t_2, \ldots, t_m\}$, a set of p prioritization indexes $F = \{f_1, f_2, \ldots, f_p\}$, and a set of the pairwise priority relations $\Phi = \{(t_i, t_j) \in T \times T \mid \phi(t_i, t_j) \neq 0\}$, RankBoost is employed to decide the priorities of the GUI testing cases by learning the historical testing experiences.

In RankBoost, the function ϕ describes the priority relation of two test cases based on the prioritization criteria. For two test cases t_i and t_j, if t_j will be executed before t_i ($t_j \prec t_i$), $\phi(t_i, t_j) = 1$. If $t_i \prec t_j$, $\phi(t_i, t_j) = -1$. If there is no testing preference for t_i and t_j, $\phi(t_i, t_j) = 0$. The pairwise priority relations Φ and the prioritization indexes F are used in a feedback function for adjusting the rank of the test cases and getting new pairwise priority relations.

Algorithm 1 illustrates the pseudocode of the RankBoost learning algorithm. The weak learner presented in [17] is used to generate a *weak ranking* $H : T \to \Re$. In Algorithm 1, a normalization factor Z_k is calculated using Eq. (1) such that D_{k+1} is a distribution.

$$Z_k = \sum_{\forall m} D_k(t_i, t_j) \exp(\alpha_k (h_k(t_i) - h_k(t_j))) \quad (1)$$

In Eq. (1), the parameter α_k is used as the weight for the linear combination of the weak learners $h_k(t_i)$ and $h_k(t_j)$. Considering

Algorithm 1 The learning algorithm based on Rankboost

Input:

L: the set of laboratory programs under test

T : the set of test cases

F : the set of prioritization indexes

Initial distribution D = initialize(Φ)

Output:

The final rank: $H(t) = \sum_{k=1}^{K} \alpha_k h_k(t)$

1: Initialize $D_1 = D$

2: **for** k=1 to K **do**

3: Train the weak learner using D_k, T, and F

4: Get a weak ranking $h_k : T \to \Re$

5: Choose $\alpha_k \in \Re$

6: Update D_{k+1}: $D_{k+1} = \frac{D_k(t_i, t_j) \exp(\alpha_k (h_k(t_i) - h_k(t_j)))}{Z_k}$

7: **end for**

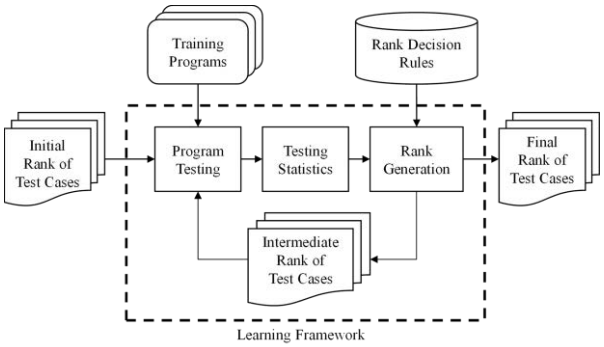


Figure 1. The learning framework for prioritizing GUI test cases.

Table 1. Bug category studied in the experiments

Type	Description
Activity Error	Incorrect implementation of the activity life cycle
Event Error	Wrong action for a received event
API Error	API version incompatibilities
Concurrency error	Errors from the concurrent threads or processes
Dynamic type error	Runtime type exceptions
I/O error	Errors from I/O interaction
Unhandled exception	Exceptions uncatched in program code
Other error	Errors in the program logic

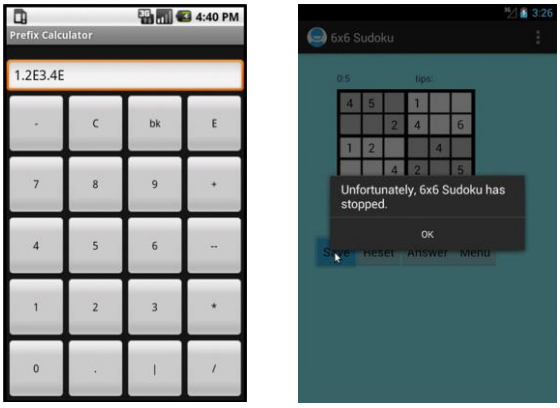
the minimization of the ranking loss $rloss_D(H) \le \prod_{k=1}^{K} Z_k$, the value of α_k can be determined by minimizing Z_k.

Figure 1 illustrates the proposed learning framework for prioritizing the GUI testing cases. The proposed framework iteratively adjusts the rank of test cases. The RankBoost weak learner dynamically decides the weak ranking *H* according to the testing results of the training programs. The final rank is used for the following GUI testing procedure to improve the testing efficiency.

4. EXPERIMENTS

To study the effectiveness of the proposed learning framework for prioritizing GUI test cases, we collected laboratory programs from an Android programming course in Yuan Ze University. Two types of laboratory programs were used for experiments: a prefix calculator and a specialized 6×6 Sudoku game. Figure 2 shows two example screenshots of these two applications. The screenshots also show that these two programs have errors. In Figure 2(a), the calculator app has a *dynamic type error* because it does not check the format of the input "1.2E3.4E" in the program. A *NumberFormatException* fault occurs when this input is passed to a string-to-number conversion method. In Figure 2(b), the Sudoku app has an *I/O error* because it tries to save a game record file which has not been actually created. A *NullPointerException* fault occurs when the record-saving button is pressed.

We collected 40 calculator programs and 37 Sudoku programs, and selected 24 calculator programs and 20 Sudoku programs among them in the experiments. These programs were selected



(a) Calculator (b) 6×6 Sudoku

Figure 2. The screenshots of the calculator and Sudoku apps.

because they were executable and could provide the full functions for the program assignments. In addition, most of these programs had one to three errors. Following the bug category studied in [5], we designed 10 test cases for each program separately. Table 1 illustrates these types of errors studied in the experiments. There are totally eight error types. We designed test cases to detect these program errors. For example, the following test case was used to find an unhandled exception fault in the Sudoku app: (<*Enter one of the grids with an alphabet or number greater than 6*>, *Expect program does not crash*). Table 2 shows the numbers of programs containing errors in these two categories of the laboratory programs.

In the experiments, we also implemented a baseline testing method using a random ranking scheme. To evaluate the effectiveness of the proposed prioritization scheme, 5-fold cross

Table 2. The number of programs having errors

Error #	Calculator	Sudoku
0	4	0
1	11	9
2	8	11
3	1	0
Total	24	20

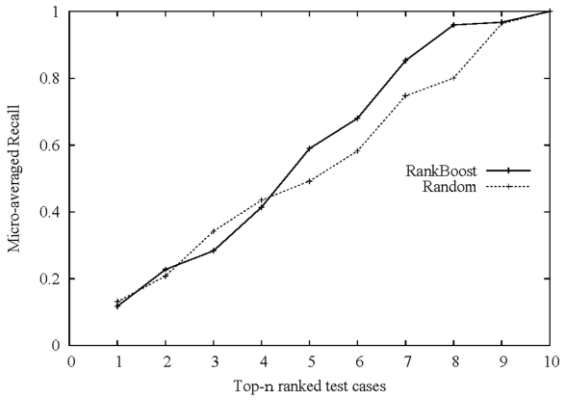
validation was used in the experiments. All programs under test were first equally divided into five subsets (folds). Among these five folds, the programs in four folds were used to train the prioritization model and the rest one fold was used for testing. The above testing process was repeated five times and a different fold was used as the testing set each time. In the experiments, we used the number of the found errors as the prioritization index in the RankBoost algorithm.

In this study, the top-*n* *micro-averaged recall* (MiR) rates and the top-*n* *macro-averaged recall* (MaR) rates of both testing scheme were used to measure the performance. The top-*n* MiR is defined as the number of correctly found errors to the total number of errors in Eq. (2).

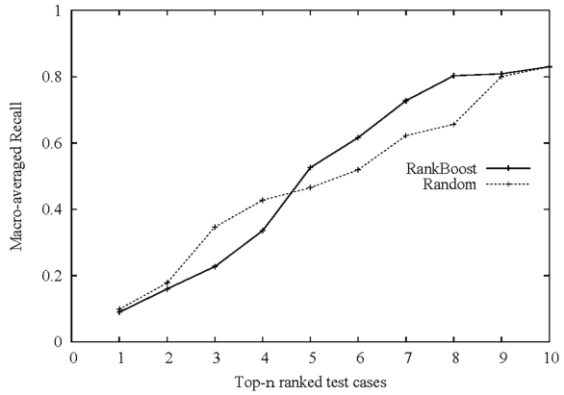
$$MiR@n = \frac{NC_{all}@n}{NT_{all}} \tag{2}$$

$NC_{all}@n$ is the number of all correctly found errors by using top-*n* test cases, and NT_{all} is the total number of errors of all types. Eq. (3) defines the top-*n* MaR:

$$MaR@n = \frac{1}{n}\sum_{i=1}^{8}\frac{NC_i@n}{NT_i} \tag{3}$$

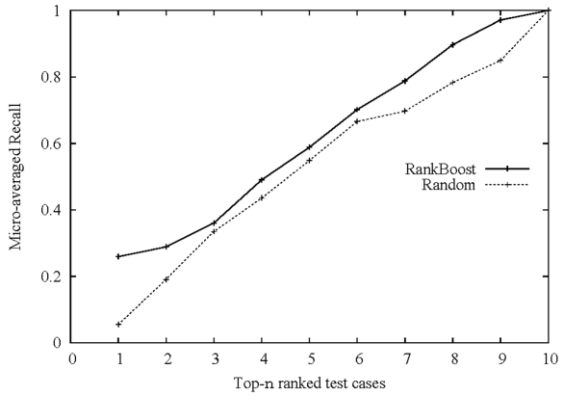


(a) Micro-average recall

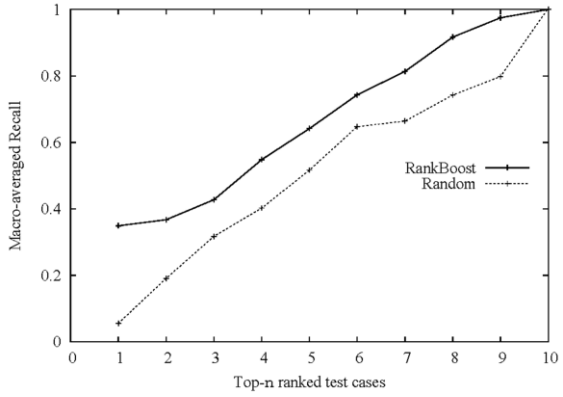


(b) Macro-average recall

Figure 3. Micro-averaged and macro-averaged recalls for the calculator app.



(a) Micro-average recall



(b) Macro-average recall

Figure 4. Micro-averaged and macro-averaged recalls for the Sudoku app.

$NC_i@n$ is the number of correctly found errors of error type i by using top-n test cases, and NT_i is the total number of errors of error type i.

Figure 3 illustrates the top-n MiR and top-n MaR performance for the prefix calculator programs. The experimental results show that the MiR and MaR performance of the RankBoost-based learning framework is close to the performance of the random testing baseline if the number of the test cases is smaller than five. However, if more than four test cases are used, the RankBoost-based learning framework has higher MiR and MaR rates. This is mainly because a part of calculator programs do not have any errors. Therefore, the weak learner of RankBoost cannot effectively adjust the testing sequence with few pairwise priority relations.

Figure 4 illustrates the top-n MiR and top-n MaR performance for the Sudoku programs. The experimental results show that the RankBoost-based learning framework outperforms the random testing baseline. This is mainly because every Sudoku laboratory program has at least one error. The RankBoost weak learner can effectively learn the pairwise priority relations from the training test cases.

5. CONCLUSION AND FUTURE WORK

In the past, many research studies have been conducted for GUI testing. However, the problem of laboratory program testing is rarely discussed. In this paper, we propose a learning framework based on the RankBoost learning-to-rank approach to prioritize GUI test cases. Given the testing metrics as the prioritization indexes, the proposed framework can effectively adjust the testing sequence by learning the past testing experiences.

To verify the effectiveness of the proposed learning framework, we collected two collections of Android laboratory programs from an Android programming course and conducted experiments. The experimental results show that the proposed learning framework can effectively rank the test cases if there are enough pairwise priority relations in the training test cases.

More investigations are still needed for the proposed learning approach. First, other priority indexes, such as the error numbers of different error types, may further improve the testing performance. Therefore, more priority indexes will be discussed in our future work. Second, the current approach cannot obtain performance improvements when the number of the existing pairwise priority relations in the training test cases is small. Other learning-to-rank techniques may need to be considered to explore more significant prioritization information for performance improvements. In the future, we also plan to discuss the generalization of this learning-to-rank approach on other programming environments.

6. ACKNOWLEDGMENTS

This work was supported in part by Ministry of Science and Technology, Taiwan under grant MOST 104-2221-E-155-004. The authors would also like to express their sincere thanks to anonymous reviewers for their precious comments.

7. REFERENCES

[1] T. G. Wang, D. Schwartz, and R. Lingard, "Assessing Student Learning in Software Engineering," *Journal of Computing Sciences in Colleges*, vol. 23, no. 6, pp. 239–248, Jun. 2008.

[2] V. Legourski, C. Tr¨odhandl, and B. Weiss, "A System for Automatic Testing of Embedded Software in Undergraduate Study Exercises," *ACM SIGBED Review*, vol. 2, no. 4, pp. 48–55, Oct. 2005.

[3] J. K. Muppala, "Teaching Embedded Software Concepts using Android," in *Proceedings of the 6th Workshop on Embedded Systems Education* (WESE '11), 2011, pp. 32–37.

[4] A. I. Wasserman, "Software Engineering Issues for Mobile Application Development," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research* (FoSER '10), 2010, pp. 397–400.

[5] C. Hu and I. Neamtiu, "Automating GUI Testing for Android Applications," in *Proceedings of the 6th International Workshop on Automation of Software Test* (AST '11), 2011, pp. 77–83.

[6] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI Ripping for Automated Testing of Android Applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (ASE 2012), 2012, pp. 258–261.

[7] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and G. Imparato, "A Toolset for GUI Testing of Android Applications," in *Proceedings of the 28th IEEE International Conference on Software Maintenance* (ICSM 2012), 2012, pp. 650–653.

[8] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "SmartDroid: an Automatic System for Revealing UI-based Trigger Conditions in Android Applications," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (SPSM '12), 2012, pp. 93–104.

[9] H.-L. Wen, C.-H. Lin, T.-H. Hsieh, and C.-Z. Yang, "PATS: A Parallel GUI Testing Framework for Android Applications," in *Proceedings of the 2015 IEEE 39th Annual Computer Software and Applications Conference* (COMPSAC 2015), 2015, pp. 210–215.

[10] G. Rothermel, R. J. Untch, C. Chu, and M. J. Harrold, "Prioritizing Test Cases For Regression Testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, Oct. 2001.

[11] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test Case Prioritization: A Family of Empirical Studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, Feb. 2002.

[12] P. Tonella, P. Avesani, and A. Susi, "Using the Case-Based Ranking Methodology for Test Case Prioritization," in *Proceedings of the 22nd IEEE International Conference on Software Maintenance* (ICSM '06), 2006, pp. 123–133.

[13] J. C. Spohrer and E. Soloway, "Novice Mistakes: Are the Folk Wisdoms Correct?" *Communications of the ACM*, vol. 29, no. 7, pp. 624–632, Jul. 1986.

[14] R. Lischner, "Explorations: Structured Labs for First-Time Programmers," *ACM SIGCSE Bulletin*, vol. 33, no. 1, pp. 154–158, Feb. 2001.

[15] G. Yarmish and D. Kopec, "Revisiting Novice Programmer Errors," *ACM SIGCSE Bulletin*, vol. 39, no. 2, pp. 131–137, Jun. 2007.

[16] D. Kopec, G. Yarmish, and P. Cheung, "A Description and Study of Intermediate Student Programmer Errors," *ACM SIGCSE Bulletin*, vol. 39, no. 2, pp. 146–156, Jun. 2007.

[17] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer, "An Efficient Boosting Algorithm for Combining Preferences," *Journal of Machine Learning Research*, vol. 4, pp. 933–969, Dec. 2003.

[18] T. Y. Chen and M. F. Lau, "Dividing Strategies for the Optimization of a Test Suite," *Information Processing Letters*, vol. 60, no. 3, pp. 135–141, Nov. 1996.

[19] D. Binkley, "Semantics Guided Regression Test Cost Reduction," *IEEE Transactions on Software Engineering*, vol. 23, no. 8, pp. 498–516, Aug. 1997.