Ensemble Approaches for Test Case Prioritization in UI Testing

Tri Cao[1,2,3], Tuan Ngoc Vu[2,3], Huyen Thao Le[2,3], Vu Nguyen[1,2,3,*]
[1]*Katalon LLC*
[2]*Faculty of Information Technology, University of Science, Ho Chi Minh City, Vietnam*
[3]*Vietnam National University, Ho Chi Minh City, Vietnam*
{*cttri19, vntuan19, lthuyen19*}*@apcs.fitus.edu.vn, nvu@fit.hcmus.edu.vn*

***Abstract*—Test case prioritization, which focuses on ranking test cases, is an important activity in software engineering given a large number of test cases to be executed within a short period of time. Recent approaches use test execution history and test coverage as the key information for ranking prediction while reinforcement learning has the potential for improving the accuracy of prioritization. Still, each approach has its own advantages and limitations. This paper proposes a ensemble method to take advantages of several existing models by combining different them into a single one. We evaluate our ensemble models on the data sets, including sixteen projects. The results show that one of our proposed models outperforms all single models on 12 over 16 data sets.**

***Index Terms*—test case prioritization, UI testing, ensemble method**

I. Introduction

Test case prioritization (TCP) is an important activity to reduce testing effort in software projects given the lack of time for testing and delivering software releases. For user interface (UI) testing, TCP is even more important as UI testing requires more time to execute than do other types of testing [1]. TCP helps determine a subset of tests to run instead of running all available tests while aiming to satisfy a certain objective. One common objective is to maximize the chance of detecting faults.

A number of approaches have been introduced for TCP using this objective [2]–[5]. Among them, history-based and coverage-based approaches have been shown to be effective in prioritizing tests [2], [4], [5]. The history-based method uses the test verdicts from the execution history of test cases while the coverage-based focuses on optimizing the coverage of test cases using features such as length of test cases, functions, and code changes [2], [3], [6]. Bryce et al. [6] calculated the coverage of test cases based on defined parameter-value pairs. Nguyen and Le [4] proposed an approach combining test execution history and test coverage together with reinforcement learning to prioritize UI test cases. Each approach has its own strengths and limitations. For example, history-based and coverage-based approaches do not take into account the addition of test cases in regression testing since the added test cases have less information of execution history and coverage. Whereas the reinforcement learning method shows its advantage in such scenarios as it facilitates the interaction between the agent and the environment (i.e., the test cases added).

In this paper, we present an ensemble method to build test prioritization models by combining different approaches with the hope of taking advantages of these approaches. We introduce three ensemble models, including History-based ensemble by combining two single models that use execution history as the primary information, Coverage-based ensemble by using two single models that rely on coverage information, and History Coverage-based ensemble by combining History-based and Coverage-based models. The ensemble method proposed falls into the parallel category, which means each model is run independently, and the final ranking is the average of results received from those models. Furthermore, as our method focuses on UI testing, it can take advantage of the information of test steps in each test case. By that, the method uses the test step verdict to calculate the weight of each test case. To the best of our knowledge, though ensemble methods are fairly common in machine learning, their application in UI test case prioritization has not been explored before.

We perform experiments evaluating the proposed and other state-of-the-art individual approaches using 16 data sets. The results obtained from our experiments demonstrate that our best ensemble model outperforms base models on 12 over 16 data sets. This finding suggests that the ensemble method can take advantage of each individual method to produce better test prioritization performance.

The rest of our paper is organized as follows. In section II, we describe related studies on test case prioritization in the context of UI testing. Section III presents in detail our approach. Section IV describes the experimental setup, including research questions, evaluation metrics, and data sets. The experimental results are shown in section V and discussed in section VI. Section VII is our conclusion and future work discussion.

II. Related Work

Research on test suite optimization can be grouped into three main problems: test case selection, test set minimization, and test case prioritization [7]. Since our work focuses on test

* Corresponding: Vu Nguyen (nvu@fit.hcmus.edu.vn)
DOI reference number: 10.18293/SEKE2022-148

case prioritization for UI testing, this section presents related works for this specific task.

Coverage approaches [2], [6] try to order test cases to cover the target items (which can be branches, functions, objects, code changes, etc.). These approaches are straightforward and simple. However, code coverage tools which are used to track code items during execution are not always available.

History-based approaches [8], [9] use the verdicts from the execution history of test cases in different ways to prioritize test cases. Hemmati et al. grouped test cases with similar fault percentages and used diversity or random algorithms to prioritize test cases in each group [5]. Other studies proposed functions that calculate a historical value from the cost and fault severity of each test case, then order these test cases using their values [10], [11]. Noor and Hemmati calculate the similarity between the code of each test case using Hamming distance, edit distance, basic counting and combine the similarity with historical verdicts to prioritize the test cases. Wu et al. [12] consider the time window before the execution of each test case, calculate the percentage of failure for each time window, then sort the test cases based on their likelihood of failure.

Alongside new methods to represent and extract data, machine learning algorithms have been used in TCP recently because they can learn the rules automatically and therefore becomes more compatible for each project compared to traditional methods. Learning to rank algorithms, which were originally used to rank searching results or prioritize content on websites, was applied to TCP in [13]. In [4], researchers proposed a coverage graph that can utilize both the historical and coverage information of the test case. The graph can update itself after each cycle so that cycles would affect the order of test cases. Sharma and Agrawal built an information graph from UML and user story, then fed that graph into metaheuristic algorithms [14]. Kaur et al. extracted the elements of UI in each test step, then used this data as input for traditional machine learning algorithms (SVM, decision trees, naive Bayes, etc.) [15]. For the first time, ensemble methods are applied with traditional machine learning for a general case of TCP in [16].

III. Our Approach

In this section, we describe the proposed ensemble method for UI test case prioritization.

A. Ensemble method for UI test case prioritization

Ensemble is a common method in machine learning where models need to make predictions. Combining the predictions of two or more models often gives better results than the performance of a single model [17]. This is because by joining multiple models together, weaknesses of one model are expected to be improved by other models and vice versa. In UI test case prioritization, there are different approaches with their own strengths and limitations which are able to complement each other. Hence, we apply this idea of the ensemble method for prioritizing UI test cases.

Ensemble methods in machine learning have two main paradigms: sequential ensemble methods and parallel ensemble methods [18]. In sequential ensemble methods, the dependence between base models is exploited by successively applying those base models one after another. On the contrary, parallel ensemble methods focus on the independence between base models by running them in parallel and combining the results later on. In this paper, we apply parallel methods since we do not observe any considerable dependencies between the chosen UI test case prioritization approaches.

Fig. 1 provides an overview of our model. Our ensemble method is divided into two phases: the first is to obtain predictive ranking from different models, and the second phase is to combine results from the first phase to one final ranking using a voting policy. The test suite to be prioritized contains *N* test cases that are passed through each individual model. The output of every single model is the order of the test cases in the test suite. That is, each test case has an index representing its position corresponding to each model. Thus, each test case in the test suite will have *M* position values from *M* single models, where *M* is the number of models participating in the ensemble model. We calculate the final weight of a certain test case by adding all the weights of that test case from *M* models. Specifically, in formula (1), a_i is the final weight of the test case i^{th} in the original order, and it is the sum of w_{ji} where w_{ji} represents the weight (position) of the ith test case in the test suite given by the jth model. The test cases are then sorted according to their final weights.

B. Strategies to choose the method for ensemble

In the context of UI testing, many TCP approaches have been proposed. Two common approaches are history-based and coverage-based. Both aim to increase the efficiency of fault detection, yet they use different information and are based on different hypotheses. Moreover, each method carries different advantages. We implement ensemble models to combine those advantages in the hope that they can help increase the overall performance. In detail, we propose three ensemble models, which are History-based, Coverage-based, and History Coverage-based models.

1) History-based ensemble: History-based approach is based on the hypotheses that test cases with errors in the past have a high probability of continuing to detect errors. We choose two methods to include in the history-based ensemble. The first method prioritizes test cases according to the number of failures of each test case in the past. This approach, which is called HBRL hereafter, was proposed by Hemmati et al. [5]. The second is RLTCP [4], which also uses execution history information and a combination of test coverage and reinforcement learning. Although RLTCP is generally more efficient, the first method prevails in the first cycles of the application under test (AUT) because RLTCP needs to go through several cycles to be effective. Combining an ML-based and a traditional method is expected to provide a stable use of the model across stages in software development.

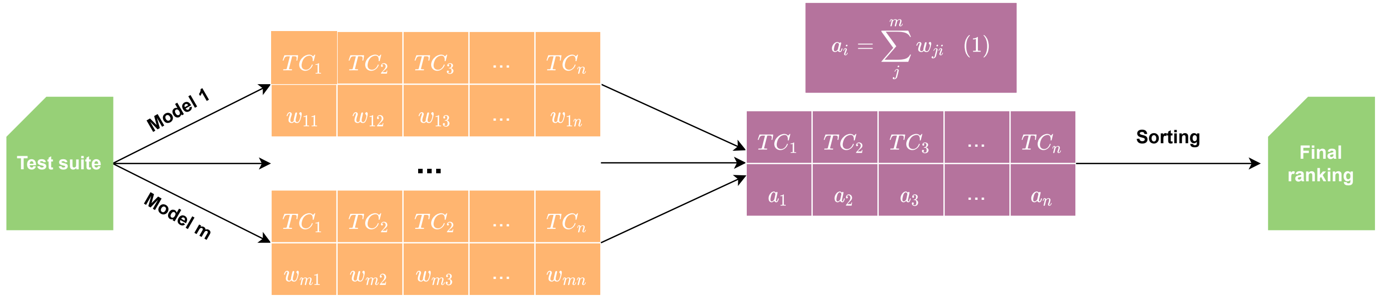

Fig. 1: The basic scheme of the two-step data fusion approach. The first one is to obtain predictive results from different models. The second one is to combine these results to one final result with voting policy

2) Coverage-based ensemble: Coverage-based is based on the coverage information of a test case for one or several components in the AUT. We choose two methods for the coverage-based ensemble. The first method is implemented by sorting based on the number of test steps of a test case, i.e., the length-based method. The test case with the most number of test steps will be ranked first. The second method, called StepGreedy, tries to cover all test steps as quickly as possible using a greedy algorithm. The former considers test cases to be independent of each other, meaning that though test cases include the same test steps, they are considered different. In other words, if two test cases share the same large number of test steps, they are both ranked high even though we just need to execute one of them. Whereas the latter takes into account the relation between test cases, which means that two test steps doing the same action are considered as one test step. After one test case is ranked, its test steps are marked satisfied and removed from further consideration. We select a next test case that maximizes the number of test steps not covered in previously selected tests. In the context of UI testing, certain test cases depend on each other, but there are also test cases entirely independent of each other. Therefore, the expectation of combining these two methods is to increase the efficiency of the model.

3) History Coverage-based approach: We choose RLTCP and StepGreedy from the coverage-based approach for the history coverage-based ensemble. This combination aims to create an approach that uses both execution history and coverage information of test cases.

IV. Experimental design

A. Overview

We conduct experiments based on how TCP approaches can be used in the software development process. At each iteration or cycle, testers use the models to predict the order of test cases to be executed for the release corresponding to the iteration. They then use the prediction to execute tests that are highly ranked given their time limit.

B. Research Questions

We design the experiments to answer the following research questions:

RQ1: How do the ensemble models perform in comparison with individual models in UI test case prioritization?

We compare the performance of ensemble models with single models to see how effective they are. Single models are the four that we discussed in section III. From that, we find out which ensemble method has the best performance for optimizing early error detection.

RQ2: How does the performance of the ensemble models change over test iterations?

The number of test cases changes after each iteration in software development. Test cases that fail in previous iterations may no longer fail in the current iteration or vice versa. We perform an experiment to represent the performance of ensemble models to compare with the single methods across test execution iterations.

C. Evaluation metrics

We use a standard metric in the Test case selection and prioritization problem called Average Percentage of Fault Detected (APFD). This metric is proposed by Rothermel et al. [19], and it measures the proportion of errors identified at each percentage of test suite execution and then calculates the average to evaluate the effectiveness of a test case prioritizing approach. APFD is calculated according to equation (1). For each fault, the metric determines the first test case index that detects that fault and computes the summation of those indexes of all faults, then divides it by the product of m total faults and n total test cases in the test suite. TFi denotes the number of tests needed to execute before discovering the fault i. That is, if all test cases that expose undetected errors are prioritized and run before test cases that fail to detect new errors, the APFD value for that order will be the highest.

$$APFD = 1 - \frac{TF_1 + TF_2 + TF_3 + ... + TF_m}{mn} + \frac{1}{2n} \quad (1)$$

We use a paired-sample non-parametric Mann-Whitney U test with a confidence level of 0.05 to confirm if the APFD difference between the proposed and other methods is statistically significant. The null hypothesis states that the two methods have no statistically significant difference in APFD. If a paired-sample Mann–Whitney U test has a p-value of less than 0.05, the null hypothesis is rejected, which means

that the difference between two methods in terms of APFD is statistically significant.

D. Data sets

In our experiment, we use four web applications, including Mattermost[1], Moodle[2], Spectrum[3], and Elementary Web[4] as the AUT. The applications are quite popular, adequately mature, and accumulate a considerable number of software iterations.

To create automated UI test cases and run tests, we utilize the test automation tool Katalon Studio[5]. After executing the test suite, the results of test execution will be stored in reports. The report contains the name of test cases, the test steps that each test case includes, the verdict of each test case, and the step that causes failure for test cases. We use these reports as the data sets for validation.

We establish a test suite and tweak its test cases and the AUT's source code to produce mutations across numerous iterations to replicate a realistic software development situation. The specific is described as follows:

- Iteration 1: Initialize the original suite by creating UI test cases based on the original AUT version. The number of test cases in the original suite is at least 20, and some test cases may fail.
- Iteration 2: Test suite will be added at least 10 test cases. Newly added test cases may fail. Test cases that failed in the previous iteration may have been fixed.
- Iteration 3: The suite contains at least 40 test cases. Some components can be added, removed or modified to the AUT to simulate real software development. Some test cases may fail due to these modifications.
- Iteration 4 and after: The number of test cases in each test suite varies, but no test suite has less than 20. The AUT continues to be tweaked with each iteration to simulate software development. Test cases that fail in an iteration can be debugged to ensure that a test case does not repeatedly fail in successive iterations. The last iteration may contain no failing test case.

The details for each data set is represented in Table I. No.test, No.step, No.fail are the number of total test cases, test steps in all test cases, and failing tests respectively that are accumulated across all test suites.

In general, the number of iterations of data sets ranges from 10 to 15. Sixteen groups of senior computer science students correspond to 16 data sets. Each of the four student groups worked independently of one another. This procedure reflects the nature of software development, where functionality and test cases may be added or updated after each release.

TABLE I: Number of iterations, test cases, test steps and failing tests in sixteen data sets.

Datasets	No.iteration	No.fail	No.step	No.test
Elementary01	10	104	10,740	690
Spectrum01	10	46	2,041	240
Spectrum02	14	316	3,295	638
Spectrum03	15	80	1,948	431
Moodle01	15	84	9,623	690
Moodle02	15	79	11,796	690
Moodle03	15	96	9,996	690
Moodle04	15	88	7,739	690
Moodle05	15	81	13,368	690
Moodle06	15	92	5,225	675
Moodle07	15	216	11,276	684
Moodle08	15	98	13,995	690
Moodle09	14	161	8,574	640
Mattermost01	11	198	4,631	451
Mattermost02	10	61	8,336	440
Mattermost03	14	183	6,708	637

Fig. 2: APFD of the methods over each iteration for some datasets

V. Results

Table II describes our experimental results of the ensemble methods and single methods across all data sets. Each cell in the table represents the average APFD value of a method on the corresponding data set. The two last rows are the standard deviation and mean values of the APFD scores for each method over 16 data sets.

While EnCov gives a second-worst result at both standard deviation and APFD score with the value of 9.1 and 63.9, respectively. The other two ensemble methods have the highest APFD score and a relatively small variance when compared to individual methods.

Among the based models, StepGreedy is the most stable one with a standard deviation of 4.3. LengthBased has the lowest APFD score of 56.0 and the biggest variance when its standard deviation is significantly higher than other methods (9.1 while the others vary from 4.3 to 6.9). RLTCP often has the highest APFD score among individual methods, and its results are sometimes comparable or even bigger than EnHis

[1]https://github.com/mattermost/mattermost-webapp
[2]https://github.com/moodle/moodle
[3]https://github.com/withspectrum/spectrum
[4]https://github.com/vector-im/element-web
[5]https://katalon.com

TABLE II: Average APFD over iterations of each method on considered dataset

Datasets	StepGreedy	LengthBased	RLTCP	HBRL	EnCov	EnHis	EnCovHis
Elementary01	69.3	43.2	68.1	66.1	51.7	73.1	**74.4**
Spectrum01	66.8	49.6	67.2	67.7	57.8	70.8	**71.2**
Spectrum02	67.4	44.5	70.2	62.0	55.0	70.3	**74.5**
Spectrum03	63.8	65.0	77.6	70.6	68.0	**77.7**	76.1
Moodle01	63.8	58.5	59.9	54.4	61.5	61.3	**64.4**
Moodle02	64.2	53.5	63.3	54.1	60.2	63.1	**68.5**
Moodle03	64.3	56.7	66.4	65.8	64.4	70.3	**70.5**
Moodle04	64.9	58.9	**75.2**	59.0	67.6	70.9	74.1
Moodle05	66.2	59.9	79.7	57.7	71.3	74.2	**80.5**
Moodle06	66.6	45.2	67.2	60.8	51.2	70.4	**72.4**
Moodle07	67.7	68.3	73.4	65.0	72.2	75.8	**77.5**
Moodle08	68.8	58.5	73.1	62.9	66.6	75.6	**75.8**
Moodle09	73.0	51.0	73.0	61.9	62.8	74.2	**77.6**
Mattermost01	73.6	68.2	74.0	70.6	65.6	74.4	**77.2**
Mattermost02	79.7	70.5	85.3	76.7	74.4	87.6	**88.0**
Mattermost03	68.9	45.3	72.5	66.8	62.5	71.8	**75.0**
Stdev	**4.3**	9.1	6.3	6.1	6.9	5.9	5.3
Avarage	68.1	56.0	71.6	63.9	63.3	72.6	**74.9**

and EnCovHis.

Table III is the p-value of the Mann–Whitney U test mentioned in section IV when comparing EnCovHis with other methods being experimented with within this paper. The bold value indicates the statistically significant difference in APFD between the EnCovHis and the others (p-value $\leq$ 0.05). The table suggests rejecting the null hypothesis for most of the dataset. The results show that EnCovHis is better than RLTCP on twelve datasets. Even in Elementary01, Moodle07, Mattermost01, their p-values are less than 0.01. On Spectrum01, Spectrum03, Moodle04, Moodle05, p-values are greater than 0.05, although only two of them have average APFDs smaller.

Figure.2 illustrates the APFD score of each algorithm throughout cycles for six data sets. It can be seen that EnCovHis and EnHis may perform badly during some first iterations because of lacking historical information, which is crucial for RLTCP and HBRL to give a good prioritize order, but they get better and become more stable in the latter cycles. Meanwhile, since StepGreedy does not use the execution history, it maintains a small variance during iterations of the data sets.

VI. Discussion

This section focuses on answering the research questions in Section IV using the experimental results from Section V.

RQ1: How do the ensemble models perform in comparison with individual models in UI test case prioritization?

As mentioned above, it can be seen that EnCov does not have good results. This may be because the performance of the LengthBased method is too low compared to StepGreedy. Therefore, instead of combining with each other to get a higher result, LengthBased drags the StepGreedy method down, so the APFD scores of EnCov usually lie somewhere between these two methods.

While EvCovHis and EnHis both have promísing results, EnCovHis has a slightly better performance with a more significant average APFD score and a smaller standard deviation. The difference between standard deviations can be explained when examining the standard deviations of every single model. EnHis, which uses two methods with similar standard deviation values, gives a smaller deviation compared to every single method. However, because EnCovHis has StepGreedy, which is the most stable method acting as one of its base models, it has a better standard deviation than that of EnHis.

TABLE III: p-value results of Mann-Whitney U tests between EnCovHis and four single methods

Dataset	StepGreedy	LengthBased	RLTCP	HBRL
Elementary01	**0.004**	**0.001**	**0.001**	**0.005**
Spectrum01	**0.016**	**0.004**	0.131	**0.049**
Spectrum02	**0.007**	**0.001**	**0.015**	**0.002**
Spectrum03	**0.029**	**0.054**	0.736	**0.021**
Moodle01	0.472	0.242	**0.046**	**0.025**
Moodle02	0.145	**0.035**	**0.032**	**0.013**
Moodle03	**0.047**	**0.006**	**0.040**	0.117
Moodle04	**0.007**	**0.009**	0.712	**0.001**
Moodle05	**0.001**	**0.003**	0.484	**0.001**
Moodle06	0.076	**0.002**	**0.030**	**0.009**
Moodle07	**0.002**	**0.021**	**0.005**	**0.003**
Moodle08	**0.013**	**0.003**	**0.040**	**0.001**
Moodle09	**0.033**	**0.001**	**0.033**	**0.002**
Mattermost01	0.120	**0.007**	**0.002**	**0.034**
Mattermost02	**0.038**	**0.038**	**0.038**	**0.021**
Mattermost03	**0.001**	**0.001**	**0.035**	**0.007**

The Mann-Whitney U test results from Table III suggest that EnCovHis outperforms the four considered existing methods in most of the dataset with the confidence of 95%. However, there are some exceptions, such as Moodle01, Moodle02 where the performance of history-based methods is extremely unstable. The variance of ensemble methods for these datasets is therefore affected, while the Coverage-based method can still keep a good standard deviation, so there is not enough evidence to reject the null hypothesis. RLTCP has a higher overall performance when compared to StepGreedy, LengthBase, and HBRL. Therefore, in some datasets, the APFD score of the ensemble methods only varies around RLTCP's score, and the improvements are not significant.

Examining more closely at the Table II in each dataset,

we can see that the ensemble model will be more likely to have higher results than each of its base models if these base models have a similar performance. Therefore, EnCovHis may perform better than EnHis since its two single models have a closer average APFD score than the two base methods of EnHis.

To sum up, the ensemble method can be used in prioritizing UI test cases to get a higher result if the base models are suitable. The experiment result suggests using methods with similar performances and small variances in order to get higher ensemble performance. The APFD score of the ensemble method throughout cycles depends on its based methods.

RQ2: How does the performance of the ensemble models change over test iterations?

The Fig.2 shows the APFD score of each method for all iterations of a dataset. For most of the cases, the pattern would look similar to that in Spectrum03 and Elementary01. The historical data in the first cycles are not adequate for history-based approaches to make a good decision, so they perform badly at the beginning, then become more stable and exceed coverage-based approaches. Meanwhile, since Step-Greedy does not use the execution history, it maintains a small variance during iterations of the datasets but cannot improve the result after each iteration. EnCovHis and EnHis can neutralize the pros and cons of both History-based and Coverage-based approaches. They suffer less from the lacking of execution data in the first iterations, improve faster for the next cycles while giving a stable and high APFD score in the latter ones.

VII. Conclusion

In this paper, we proposed a test case prioritization technique in regression UI testing by ensembling multiple single models into a single one. We design three ensemble models, which are history-based, coverage-based, and history coverage-based ensembles. These three models were evaluated using 16 data sets with the source code of four different AUT, including Elementary, Spectrum, Moodle, and Mattermost. The evaluating result shows that the history coverage-based model is the best one which achieves the average APFD of 74.9%. This indicates that with suitable based models, the ensemble method can outperform its own base algorithms.

Though ensemble learning is a popular machine learning technique, this is the first time it is applied in UI test case prioritization. Thus, it is a promising direction and can be further explored in the future. Not only in UI testing, but the ensemble method can also be applied in other testing problems. Furthermore, due to the limitation of our data sets, we only conduct experiments based on the idea of the parallel ensemble method, i.e., bagging ensemble. However, other ensemble approaches are still applicable and worth investing in when having larger data and suitable base models.

ACKNOWLEDGEMENTS

This research is funded by Katalon LLC. We would also like to thank students at the University of Science, Vietnam National University, Ho Chi Minh city for participating in our experiments.

References

[1] H. Vocke. (2018) The practical test pyramid. [Online]. Available: https://martinfowler.com/articles/practical-test-pyramid.html?fbclid=IwAR31q-GxAE7fx-8rLO8ayUmeFXsSy6fs1vcqylLmZVtnL2VuWVKWR0I4dboUiTests

[2] R. C. Bryce and A. M. Memon, "Test suite prioritization by interaction coverage," in *Workshop on Domain specific approaches to software test automation: in conjunction with the 6th ESEC/FSE joint meeting*, 2007, pp. 1–7.

[3] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system," *Software Testing, Verification and Reliability*, vol. 25, no. 4, pp. 371–396, 2015.

[4] V. Nguyen and B. Le, "Rltcp: A reinforcement learning approach to prioritizing automated user interface tests," *Information and Software Technology*, vol. 136, p. 106574, 2021.

[5] H. Hemmati, Z. Fang, M. V. Mäntylä, and B. Adams, "Prioritizing manual test cases in rapid release environments," *Software Testing, Verification and Reliability*, vol. 27, no. 6, p. e1609, 2017.

[6] R. C. Bryce, S. Sampath, J. B. Pedersen, and S. Manchester, "Test suite prioritization by cost-based combinatorial interaction coverage," *International Journal of System Assurance Engineering and Management*, vol. 2, no. 2, pp. 126–134, 2011.

[7] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[8] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24th international conference on software engineering*, 2002, pp. 119–129.

[9] A. Khalilian, M. A. Azgomi, and Y. Fazlalizadeh, "An improved method for test case prioritization by incorporating historical test case data," *Science of Computer Programming*, vol. 78, no. 1, pp. 93–116, 2012.

[10] 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 *2008 Second International Conference on Secure System Integration and Reliability Improvement*. IEEE, 2008, pp. 39–46.

[11] D. Marijan, A. Gotlieb, and S. Sen, "Test case prioritization for continuous regression testing: An industrial case study," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 540–543.

[12] Z. Wu, Y. Yang, Z. Li, and R. Zhao, "A time window based reinforcement learning reward for test case prioritization in continuous integration," in *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, 2019, pp. 1–6.

[13] Y. Huang, T. Shu, and Z. Ding, "A learn-to-rank method for model-based regression test case prioritization," *IEEE Access*, vol. 9, pp. 16 365–16 382, 2021.

[14] M. M. Sharma and A. Agrawal, "Test case design and test case prioritization using machine learning," *International Journal of Engineering and Advanced Technology*, vol. 9, no. 1, pp. 2742–2748, 2019.

[15] P. Kaur, P. Bansal, and R. Sibal, "Prioritization of test scenarios derived from uml activity diagram using path complexity," in *Proceedings of the CUBE International Information Technology Conference*, 2012, pp. 355–359.

[16] R. Lachmann, "Machine learning-driven test case prioritization approaches for black-box software testing," in *The European Test and Telemetry Conference, Nuremberg, Germany*, 2018.

[17] R. Polikar, "Ensemble based systems in decision making," *IEEE Circuits and systems magazine*, vol. 6, no. 3, pp. 21–45, 2006.

[18] P. Bühlmann, "Bagging, boosting and ensemble methods," in *Handbook of computational statistics*. Springer, 2012, pp. 985–1022.

[19] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 1999, pp. 179–188.