Prioritizing Natural Language Test Cases Based on Highly-Used Game Features

Markos Viggiato[1], Dale Paas[2], Cor-Paul Bezemer[1]
[1]Department of Electrical and Computer Engineering, University of Alberta, Edmonton, Canada
[2]Prodigy Education, Toronto, Canada
viggiato@ualberta.ca,dale.paas@prodigygame.com,bezemer@ualberta.ca

ABSTRACT

Software testing is still a manual activity in many industries, such as the gaming industry. But manually executing tests becomes impractical as the system grows and resources are restricted, mainly in a scenario with short release cycles. Test case prioritization is a commonly used technique to optimize the test execution. However, most prioritization approaches do not work for manual test cases as they require source code information or test execution history, which is often not available in a manual testing scenario. In this paper, we propose a prioritization approach for manual test cases written in natural language based on the tested application features (in particular, highly-used application features). Our approach consists of (1) identifying the tested features from natural language test cases (with zero-shot classification techniques) and (2) prioritizing test cases based on the features that they test. We leveraged the NSGA-II genetic algorithm for the multi-objective optimization of the test case ordering to maximize the coverage of highly-used features while minimizing the cumulative execution time. Our findings show that we can successfully identify the application features covered by test cases using an ensemble of pre-trained models with strong zero-shot capabilities (an F-score of 76.1%). Also, our prioritization approaches can find test case orderings that cover highly-used application features early in the test execution while keeping the time required to execute test cases short. QA engineers can use our approach to focus the test execution on test cases that cover features that are relevant to users.

CCS CONCEPTS

• **Computing methodologies** → **Natural language processing**;
• **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Test case prioritization, Multi-objective genetic algorithm, Zero-shot classification, Feature usage

1 INTRODUCTION

Software testing is an essential, yet costly, quality assurance activity during the software development life cycle [4, 17, 19, 21]. Despite the recent advances in test automation techniques [31, 41, 42, 52], manual tests are still widely performed across different industries [19, 41, 42, 53, 59]. In the gaming industry, for example, game developers face several challenges to automate tests and, consequently, manual testing is a predominant practice [41, 42, 53–55].

Manually executing tests is a tedious activity and requires a large amount of human effort as testers need to perform several steps to achieve the testing goal [54, 55]. As systems grow and the number of test cases increases, it becomes impractical to execute all manual test cases, mainly in a scenario with a short release cycle [19, 21, 22, 27].

Prior work proposed several approaches to optimize the execution of test cases when resources are restricted, such as prioritizing test cases during regression testing [15, 16, 21, 34, 39, 40, 43, 49, 57, 65]. However, most proposed approaches do not work for manual test cases as (1) they depend on test case source code, which does not exist for manual test cases and (2) the execution history of test cases, which could be out-of-date or difficult to be accessed [62] or is generally not meaningful for manual test cases as they tend to be specified at a higher level. Because of these two limitations of manual test cases, it is difficult to automatically prioritize their execution based on a meaningful metric.

In this paper, we propose an approach for prioritizing manual test cases that are written in natural language based on the application feature(s) that they test. In particular, we prioritize test cases that test highly-used application features, to ensure that the limited testing resources are used to test features in which bugs will affect the largest group of users. Our approach performs a multi-objective optimization with the widely-used NSGA-II genetic algorithm [12] to find optimal orderings of test cases according to two objectives: (a) highly-used feature coverage and (b) test case execution time. For objective (a), we need to identify the link between the test cases and the application features to identify which features are covered by test cases. We then collect the feature usage data for each feature. To identify this link, we leverage the strong zero-shot capabilities of several pre-trained language models.

We evaluated and optimized our approach for the data of a game from our industry partner (*Prodigy Education*).[1] Our experiments were performed with the test cases in the test suite of *Prodigy Education*.

The main contributions of our work are as follows:

- We build an automated mapping between natural language test cases and the game feature(s) that they cover.
- We propose a novel prioritization technique for natural language test cases based on the game feature(s) that they cover, in particular, the highly-used game features.

The source code of our experiments is available online.[2] The remainder of our paper is structured as follows. Section 2 describes our industrial case subject and Section 3 gives an overview of our approach for test case prioritization. Section 4 presents the experiments and results to build our zero-shot ensemble model. Sections 5 and 6 present and discuss the prioritization experiments and results. We discuss practical aspects of our approach in Section 7, related

[1]https://www.prodigygame.com/main-en/
[2]https://github.com/asgaardlab/natural-language-test-prioritization

work in Section 8 and the threats to validity in Section 9. Finally, Section 10 concludes the paper.

2 INDUSTRIAL CASE STUDY SUBJECT

In this paper, we applied our approach to the *Prodigy Math game* (from Prodigy Education), which is a proprietary, online, RPG-style educational math game with more than 100 million users around the world. The game contains over 50,000 math questions spanning Grade 1-8. The players play the role of a character (a wizard) in the Prodigy world and can go to several world zones available in the game. As the players answer math questions, their wizards can evolve, learn new spells, and acquire new equipment and in-game items. We use the test cases designed by Prodigy Education developers, the usage data generated by the players and the features of the *Prodigy Math game* as input for our approach.

Dataset characteristics. Our case study subject has 1,146 test cases that are written in natural language. Each test case contains the following fields:

- a test case name.
- an objective with the main goal of the test case.
- the time required to execute the test case, as provided by developers and QA engineers.
- one or more steps that the tester must perform.

The total combined execution time of the test cases is 133 hours. In total, the test cases cover 110 features of the *Prodigy Math game*. Every test case covers at least one feature, and a feature may be covered by more than one test case. For example, the "login" feature is covered by 27 test cases. In our data, a test case covers a median of 2 game features.

3 OVERVIEW OF OUR APPROACH FOR TEST CASE PRIORITIZATION

Our approach for prioritizing natural language test cases consists of two steps: (1) automatically identifying the tested game feature(s) from natural language test case descriptions and (2) prioritizing test cases based on the highly-used game features covered by test cases. Our approach finds the orderings of test cases that maximize the number of highly-used features covered early in the test execution and minimize the execution time. Figure 1 presents an overview of our approach.

3.1 Input

Our approach takes as inputs (1) manual test cases specified in natural language, (2) a pre-defined set of features of the application under test, and (3) the data generated from the interaction of users with the system (e.g., an execution log).

3.2 Extracting test case information

Our approach starts by extracting the execution time and textual descriptions from test cases. We use the concatenation of the test case name and objective as the textual description of the test case. We then use several techniques with strong zero-shot capabilities to identify the game features tested by test cases as we do not have a mapping between test cases and the game feature(s) that they cover. Since a test case might cover more than one feature, our approach performs a multi-label classification of test cases with those techniques. We chose the zero-shot approach because we do not have labeled data to train a classifier from scratch or even to fine-tune pre-trained models, as they require large amounts of data. Manually labeling data to train a classifier is not feasible because we have more than a hundred labels. Also, a manual classification of all the data is error-prone and infeasible due to the large number of test cases.

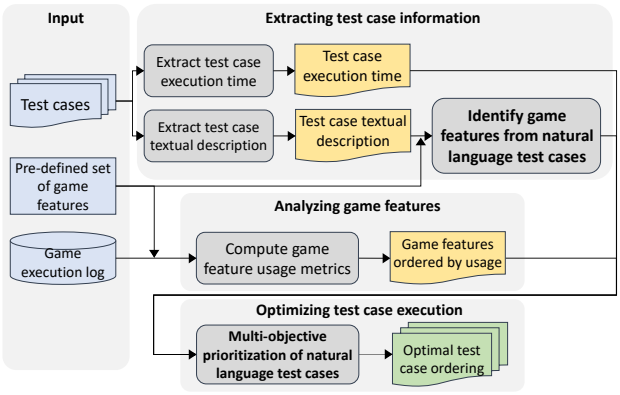


Figure 1: Overview of our approach for prioritizing natural language test cases.

Name	Co-op: Joining a team
Objective	Verify the functionality of joining another player's team
Test steps	1. Log into the game 2. Try to join the team of another player 3. Verify that the student joined the other player's team
Exec. time	1 (minute)
Cov. features	co-op, co-op join

Table 1: Test case example with the covered features.

Table 1 shows an example test case with the corresponding covered features as identified by our zero-shot classification techniques. The test case named "Co-op: Joining a team" verifies whether players can join another player's team during a cooperative battle, and therefore, covers the cooperative battles feature (named "co-op") and, more specifically, the feature that allows players to join another player's team in a cooperative battle (named "co-op join"). We store the features identified by the zero-shot techniques in a *feature coverage vector*, which is *[co-op, co-op join]* in our example. And after this stage, every test case has a corresponding *feature coverage vector*.

3.3 Analyzing game features

Our approach uses game feature usage data to prioritize test cases that test highly-used features. We collect the *total number of uses*

for each feature of the game for a specific period of time from the execution logs (in our case, the events that are stored in Prodigy's data warehouse). As the feature usage metric in our experiments, we used the average number of feature uses per week for an entire school year (September 2021 to June 2022).

3.4 Optimizing test case execution

Finally, our approach performs a multi-objective optimization with the test case descriptions (with the corresponding *feature coverage vector*), the feature usage metric (*total number of uses*), and the test case execution time. Our approach optimizes the test case order based on a maximization of the number of highly-used game features covered by test cases and a minimization of the cumulative test execution time.

4 IDENTIFYING GAME FEATURES FROM NATURAL LANGUAGE TEST CASES

In our work, we leverage techniques with strong zero-shot capabilities to identify the link between the manual test cases and the features that they cover. Recently proposed pre-trained language models (such as BART [28]) have strong zero-shot capabilities, which means that their knowledge (obtained from very large amounts of data used during pre-training) can be transferred to a new domain which has no labeled data [7, 14, 28]. As a result, we can apply these pre-trained models to new data and classes. Prior work has demonstrated the success of zero-shot learning in different fields, such as computer vision, speech, and natural language processing [9, 10, 13, 44, 52, 61].

4.1 Experiment setup

We did experiments to evaluate the performance of each individual zero-shot classification technique in our dataset. In addition, to have a more robust zero-shot classification, we experimented with different ensembles of the individual zero-shot techniques, as we explain below. For all the experiments, we used the 1,146 test cases of the *Prodigy Math game* and a list of 110 features that was defined by the game developers.

Techniques for zero-shot classification. We used three techniques that have strong zero-shot capabilities as demonstrated by prior work [13, 60, 61, 64]:

4.1.1 `BartLargeMNLI`. facebook/bart-large-mnli [28] is a model trained on the Multi-Genre Natural Language Inference (MNLI) dataset which has been shown to have strong zero-shot capabilities for text classification [64].

4.1.2 `CrossEncoderNLI`. cross-encoder/nli-distilroberta-base[3] is a model trained with a cross-encoder architecture to learn sentence embeddings [45] using the MNLI and the Stanford Natural Language Inference (SNLI) datasets, which also has zero-shot capabilities for text classification. For both `BartLargeMNLI` and `CrossEncoderNLI` models, we provide the textual description of a test case and a list of all the game features of the *Prodigy Math game*. The models output the game features sorted by their probability of being related to the test case.

[3]https://huggingface.co/cross-encoder/nli-distilroberta-base

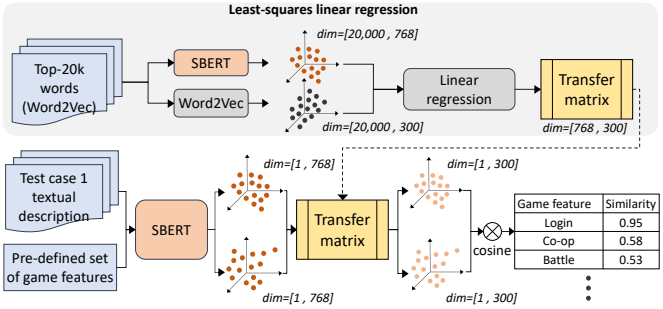


Figure 2: Overview of our `LatentEmb` technique for test case 1.

4.1.3 `LatentEmb`. This is an unsupervised, similarity-based technique that uses text embedding methods to embed sentences (to be classified) and the candidate labels and uses a similarity metric (e.g., cosine) to find the labels that are similar to the sentence [13, 60, 61]. The sentence is then classified into the most similar labels (i.e., labels that are close to the sentence in the embedding space). We need to use a sentence embedding model to embed sentences and a word embedding models to embed labels (which are usually single words). In our work, we use the popular Sentence-BERT (SBERT) model [45] to embed test case textual descriptions (i.e., sentences), with the *sentence-t5-large* pre-trained checkpoint, and the Word2Vec embedding model [37] to embed the game features (i.e., labels). However, we cannot compute the cosine similarity directly between the embedding vectors from SBERT and Word2Vec since they have different scales (SBERT vectors are 768-dimensional, while Word2Vec vectors are 300-dimensional). To compare the embeddings, we need to have the embeddings from test case description and game features in the same space. Therefore, we performed a least-squares linear regression to learn a mapping between the SBERT and the Word2Vec spaces.[4] In practice, the mapping is a "transfer" matrix that can be used to transfer embeddings from one space to the other. We can then embed test case descriptions and game features with SBERT, use the matrix to transfer all embeddings to the Word2Vec space, and compute the consine similarity in the Word2Vec space. Figure 2 shows how we used `LatentEmb` to identify the game features covered by a test case example (test case 1).

To build the mapping, we need to embed the same set of words with both SBERT and Word2Vec and then perform a linear regression with those embedding vectors. We used the top-20k most frequent words from Word2Vec for the linear regression. With the computed matrix, we can embed the description of a test case and the game features with the SBERT model and used the matrix to transfer the embeddings to the 300-dimensional Word2Vec embedding space, where we can compute the cosine metric between the test case embedding and the game feature embeddings. We performed a preliminary analysis to evaluate other word embedding models (*Glove* and *Fasttext*), but using Word2Vec with the top-20k words achieved the best performance. We also used the preliminary analysis to determine the optimal classification threshold to be used for the cosine similarity in the `LatentEmb` approach and for the two pre-trained models (as their outputs contain the game features

[4]https://joeddav.github.io/blog/2020/05/29/ZSL.html

Test case name	True feat. cov. vector	Predicted feat. cov. vector	Ground truth binary vector [battle, login, co-op]	Predicted binary vector [battle, login, co-op]
Login on mobile device	login	co-op	[0, 1, 0]	[0, 0, 1]
Start co-op battle	battle, co-op	login, co-op	[1, 0, 1]	[0, 1, 1]
Check animations in battle	battle	battle	[1, 0, 0]	[1, 0, 0]

Table 2: Example of multi-label classification of test cases. Binary vectors for the "battle" feature are highlighted in green (ground truth) and orange (predicted).

along with their probabilities). We used the best thresholds found in our analysis: 0.9 for `BartLargeMNLI`, 0.6 `CrossEncoderNLI`, and 0.2 for `LatentEmb` (we give more details on how we could evaluate the models in Section 4.2 below).

Ensembles of techniques for zero-shot classification. We also experimented with different ways of aggregating the classifications from each individual zero-shot technique to build an ensemble. Below, we explain the different aggregation methods that we explored.

4.1.1 Ensemble with majority voting (`EnsMajorVoting`*).* Our initial idea is to use a majority voting approach to obtain the final classifications. This ensemble uses the sets of labels obtained from each individual zero-shot model and selects the labels provided by at least two models.

4.1.2 Ensemble with full intersection (`EnsFullInters`*).* Aiming at having more robust and high-confidence classifications, this ensemble uses only the labels that were provided by all the three models.

4.1.3 Ensemble with back-off using top-2 models (`EnsBackOffTwo`*).* The ensemble above (`EnsFullInters`) might be too strict sometimes, so we evaluated an ensemble that first obtained the labels that were provided by all the three models and, if that results in an empty set, this ensemble backs-off to the intersection of the two best individual models. Note that this is different from majority voting, which uses the labels provided by a minimum of any two models (not only the top-2 best models).

4.1.4 Ensemble with back-off using all models (`EnsBackOffComplete`*).* If the intersection of all three models is empty, our approach backs-off and inspects the intersection of the two best individual models. Then, if the top-2 intersection is still an empty set, the approach backs-off again and inspect the intersection of the best and third best model. At last, if that also results in an empty set, we use the intersection between the second and third best models. Note that for all the ensembles, if the final result set is empty, we do not assign any labels to the test case.

Baseline. To have a "sanity check", we use a keyword search approach as baseline. We search the feature name in the test description to find if that feature is covered by that test case.

4.2 Evaluation

To evaluate our proposed approaches, we manually labeled a subset of the test cases in the test suite of Prodigy Education. Please note that we only labeled the data to be able to evaluate the zero-shot and ensemble models. To use our approach in practice, no manual data labeling is necessary. To label the test cases, the first author, who has an extensive knowledge of the *Prodigy Math game*, randomly selected test cases until there was at least one labeled example for each label. In total, we labeled 211 test cases and there are, on average, 3 examples for each label. Using the labeled data, we computed the precision, recall, and F-score for all the evaluated approaches. As the F-score metric penalizes both the false positives and false negatives, we focus the discussions on that metric. To compute the evaluation metrics for our multi-label classification task, we used the `scikit-learn` package, which computes the metrics for each individual label and obtains the average. We used a weighted average because our labels are imbalanced (i.e., the number of labeled examples for each label is different). To clarify how we computed the F-score for our multi-label classification, Table 2 shows examples of three test cases, their true feature coverage vector, and the feature coverage vector predicted by a model. We used the *MultiLabelBinarizer* from `scikit-learn` to obtain the binary vectors from the feature coverage vectors. The binary vectors follow a fixed order of the features, such as [battle, login, co-op] in our example, and contain '1' in case that feature is present and '0' otherwise. We then compute the evaluation metrics (e.g., F-score) for each label individually (i.e., for each game feature) and average the per-label metrics. For example, for the "battle" feature, we use the binary elements that correspond to the position of "battle" in the ground truth binary vectors (i.e., the first elements, which are highlighted in green), which gives [0, 1, 1]. We do the same for the predicted binary vectors (elements highlighted in orange) and obtain [0,0,1]. We then compute the F-score between those two vectors, which gives an F-score of 0.67. We do the same procedure for all game features and compute their average weighted by the number of times each game features appears in the ground truth (e.g., "battle" appears two times, in the second and third test cases, while "login" appears once, in the first test case).

4.3 Results

Table 3 presents the results of our experiments with the zero-shot models. All the ensemble approaches perform better than the individual models and the baseline. The *EnsBackOffComplete* approach has the best performance, with an F-score of 76.1, followed closely by the *EnsBackOffTwo* approach, with an F-score of 76.0. The best individual model is the *LatentEmb*, with an F-score of 72.3, while the *BartLargeMNLI* and the *CrossEncoderNLI* achieved F-scores of 70.5 and 69.9, respectively. Based on these results, we used the *EnsBackOffComplete* approach to classify all the test cases in our dataset.

Zero-shot approach	F-score	Precision	Recall
Baseline (keyword search)	59.8	65.5	60.0
BartLargeMNLI	70.5	69.9	79.9
CrossEncoderNLI	69.9	73.5	75.3
LatentEmb	72.3	71.5	80.9
EnsMajorVoting	74.1	71.5	84.4
EnsFullInters	74.7	74.2	83.1
EnsBackOffTwo	76.0	78.3	78.9
EnsBackOffComplete	**76.1**	78.0	79.2

Table 3: Results of experiments with the zero-shot models.

5 MULTI-OBJECTIVE PRIORITIZATION OF NATURAL LANGUAGE TEST CASES

To optimize the execution of manual test cases, our approach performs a multi-objective optimization using a genetic algorithm. Below, we explain how we applied the non-dominated sorted genetic algorithm (NSGA-II) genetic algorithm [12] to test case prioritization, the performed experiments and the obtained results.

5.1 Multi-Objective Genetic Algorithms

A genetic algorithm is a search-based heuristic that uses the concept of natural evolution to find the best solutions from a large number of possible solutions [11]. In our case, a possible solution is a specific test case ordering that is searched among all possible test case orderings (i.e., all the permutations of orderings). We apply the NSGA-II algorithm [12] because it has been widely used for multi-objective optimization for different purposes in the software engineering field [29, 48, 50, 57]. At each iteration, the algorithm uses an objective function (i.e., fitness function) to evaluate the candidate solutions that we generated. Differently from a single-objective optimization, in which the candidates are evaluated using a single objective, in a multi-objective scenario, there is a trade-off between the multiple objectives. NSGA-II uses the concept of dominance [11, 26] to determine the best solutions. A solution s1 dominates solution s2 ($s1 \preceq s2$) if s1 is no worse than s2 for all the objectives and s1 is strictly better than s2 for at least one objective. In the end, the algorithm outputs a set of non-dominated solutions, which are called the *Pareto front* [26].

5.2 Test Case Prioritization Using NSGA-II

For our work, a *Pareto front* consists of a set of test case orderings with the optimal trade-off between the objectives. To use NSGA-II, we need to define the *solution encoding* (i.e., how a solution is represented). In our case, as a solution corresponds to a specific test case ordering, we assign a unique integer to identify each test case. Therefore, a solution is represented by an ordered sequence of integers $[1, 2, \cdots, n]$, where n is the total number of test cases in our test suite. We initialize NSGA-II by randomly sampling a subset of all the possible test case orderings. For the required genetic operators, we use the default operators provided by the Python package that we used (`pymoo` [5]) for permutation problems: binary tournament for the *selection* operator, order-based crossover for the *crossover* operator (order-based is a crossover operator proper for permutation encoded chromosomes, such as our case), and inversion mutation for the *mutation* operator.

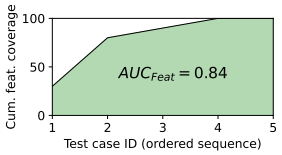
5.3 Objective functions for NSGA-II

We defined three objective functions that are used during the optimization process in our experiments. Similarly to prior work [49, 50], we use normalized metrics as the objective functions to avoid bias of the model towards functions with larger values. In addition, normalized objective functions are inherently more interpretable.

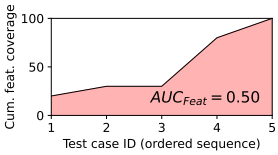
As we mentioned in Section 3, the goal of our prioritization approach is to search for test case orderings such that (1) highly-used features are covered early in the test execution and (2) test cases with shorter durations are executed early in the testing. During our experiments, for comparison purposes, as we explain in Section 5.5, we also performed an optimization such that (3) a large number of features (not necessarily the highly-used features) are covered early in the test execution and (2) test cases with shorter durations are executed early in the testing. To capture these three criteria during the test case prioritization, we defined the following objective functions.

5.3.1 Feature ranking similarity (featRankSim). This metric measures the similarity between two rankings: the feature usage ranking (in which the features are sorted by their total number of uses) and the feature testing ranking (in which the features are sorted according to the order in which they are covered when executing the ordered test cases). Ideally, the feature testing ranking is the same as the feature usage ranking. To measure the ranking similarity, we used the normalized discounted cumulative gain (NDCG) metric [25, 58]. NDCG is commonly used to compute ranking quality in Information Retrieval-based systems [36] and uses a graded-scale relevance for documents, where the usefulness of a document is measured based on its position in the ranking (highly-relevant documents should be at the top of the ranking). The cumulative gain score is computed as we move from top to bottom in the ranking. The lower the position of a document in the ranking, the lower the gain that it provides to the final score. Because of this, NDCG gives greater importance to documents in the top of the ranking. For example, differences in the top of the ranking have a larger impact on the score than differences in the bottom of the ranking. We used the `scikit-learn` implementation of NDCG, which lies in the range *[0,1]*, with 1 indicating a perfect match between the obtained ranking and the ideal ranking. In our case, a document is a game feature and we use the total number of uses of the feature as its relevance score. The ideal ranking is obtained by sorting the features by their usage (feature usage ranking). Finally, we want to maximize the *featRankSim* objective to have the feature testing ranking as similar to the feature usage ranking as possible (which means that highly-used features are tested early in the test case ordering).

5.3.2 Cumulative execution time (cumExecTime). This metric captures how the cumulative execution time of test cases changes as test cases are executed in a specific order. For each executed test case, its execution time is added to the partial cumulative execution time. Since we want *cumExecTime* to increase as slow as possible as we execute the ordered test cases, we use the AUC obtained as



(a) Ordering w/ high AUC_{Feat}



(b) Ordering w/ low AUC_{Feat}

Figure 3: Examples to demonstrate our objective function.

we move along the sequence of ordered test cases as the objective function (AUC_{Time}). We normalize AUC_{Time} with regard to the maximum area (which is the test case ordering in which the first test case has an execution time that corresponds to the total execution time of the test suite).

5.3.3 Cumulative feature coverage (cumFeatCov). This metric captures how the number of covered features increases as test cases are executed in a specific order. Since one feature might be tested in multiple test cases, we need to define a minimum number of test cases necessary to consider that a feature has been covered. We defined a threshold for the percentage of test cases that is sufficient to consider that a feature was indeed tested and can be counted as covered (which we call *per-feature coverage threshold*). For example, if the *per-feature coverage threshold* is 0.8, we only consider feature "A" covered after executing 4 out of 5 test cases that cover that feature. To obtain the *cumFeatCov* metric, we get the set of features associated with the test cases as they are executed one at a time and compute how many features are covered. A feature is considered covered if its *per-feature coverage threshold* is met. Since we want *cumFeatCov* to increase as quick as possible as we execute the ordered test cases, we use the area under the curve (AUC) obtained as we move along the sequence of ordered test cases as the objective function (AUC_{Feat}). We normalize AUC_{Feat} similarly as we do for AUC_{Time}

Figure 3 presents examples of different test case orderings that achieve different AUC_{Feat} and helps to clarify our goal of maximizing AUC_{Feat}. Figure 3a shows a test case ordering in which a large number of features is covered early in the sequence (which yields a large AUC_{Feat} of 0.84), while Figure 3b shows that the number of covered features increases slower than in Figure 3a (which yields a smaller AUC_{Feat} of 0.50). Since we want the number of covered features to increase as quick as possible, the ordering with the larger AUC_{Feat} is preferable.

5.4 Stopping Criteria for NSGA-II

Lastly, we need to define the stopping criteria for NSGA-II so that the algorithm can be stopped when no progress is made in the search for the optimal solutions. Similarly to prior work [50], and to have a systematic way of deciding when to stop the algorithm execution, we defined two stopping criteria that we used in our experiments.

T-test: for the non-dominated solutions s_i of each new generation g_i during NSGA-II execution, we run a t-test [56] for each objective to compare the non-dominated solutions of the new generation with the solutions s_{i-1} of the previous generation g_{i-1}. For example, if generation g_i has 10 solutions, there are 10 non-dominated test case orderings, i.e., 10 values for each objective: AUC_{Feat}, AUC_{Time}, and *featRankSim*. When the t-tests for all the objectives show that the difference between the two generations is insignificant (p-value ¡ 0.05) for five consecutive generations, the algorithm execution is stopped.

Mutual Dominance Rate (MDR): we also use the set of non-dominated solutions for two consecutive generations g_{i-1} and g_i to compute the mutual dominance rate (MDR) indicator [18, 35]. Consider a function $\Delta(g_{i-1}, g_i)$ that returns the set of solutions in g_{i-1} that are dominated by at least one solution in g_i. We can then formulate the MDR as:

$$MDR = \frac{|\Delta(g_{i-1}, g_i)|}{|g_{i-1}|} - \frac{|\Delta(g_i, g_{i-1})|}{|g_i|}$$

where $|g|$ is the number of elements in g.

The MDR indicator ranges from -1 to +1, in which an MDR of -1 indicates that the solutions of the current generation are not better than the solutions of the previous generation, while an MDR of +1 indicates that the current solutions are completely better than the previous solutions. An MDR of zero means that no significant progress has been made [35]. Since the MDR can have alternated signs due to the randomness of genetic algorithms, we consider that the algorithm can be stopped when MDR lies within a pre-defined range *[-a, a]* (as done in prior work [50]) for five consecutive generations (which is stricter than prior work [50]). We experimented with different MDR ranges, as we explain below.

5.5 Experiment setup

In this section, we describe the experiments that we performed to assess how our approach works in different scenarios and with different parameters. For all experiments, we used our dataset of 1,146 test cases, with a total execution time of 133 hours and 110 game features covered by test cases. Similarly to prior work [1, 20, 49, 50], we used random-based search approaches as the baselines with which we compare our approaches. We randomly selected 50 test case orderings, named $Random_{50}$, and 100 test case orderings, named $Random_{100}$. The random orderings were selected without replacement from the entire population of test case orderings. Also, following the literature guidelines [2, 49], we used a population of 100 in all our experiments. We executed NSGA-II 50 times during the experiments to mitigate the randomness involved in genetic algorithms and we report the results from all 50 runs.

Experiment 1: *number of covered game features* versus *test execution time* (without feature usage). In this experiment, we performed a bi-objective optimization for different combinations of *per-feature coverage threshold* and stopping criteria. We performed the test case prioritization only with the AUC_{Feat} and AUC_{Time} objective functions. Our goal is to understand the trade-off between game feature coverage and execution time when no feature usage information is included. We evaluated four *per-feature coverage thresholds*: 50%, 75%, 90%, and 100%. We consider that 50% is the minimum acceptable threshold to consider that a feature is covered. For each *per-feature coverage threshold*, we evaluated three approaches with different intervals for MDR in the stopping criteria: $Stop_{0.25}$, $Stop_{0.10}$, and $Stop_{0.05}$, with the following ranges: *[-0.25, 0.25]*, *[-0.10, 0.10]*, and *[-0.05, 0.05]*. For the stopping criteria, both

the t-test (p-value ¡ 0.05) and the MDR criteria must be satisfied for five consecutive generations.

Experiment 2: *number of covered highly-used game features* **versus** *test execution time* **(with feature usage).** In this experiment, we used the game feature usage in the optimization through the *featRankSim* objective function instead of only the number of covered game features. Our goal is to find test case orderings that test highly-used features early in the test execution in the shortest amount of time. We evaluated the same stopping criteria as in experiment 1, i.e., the *[-0.25, 0.25]*, *[-0.10, 0.10]*, and *[-0.05, 0.05]* MDR ranges together with the t-test. For experiment 2, as we included feature usage, we named the approaches as follows: $Stop_{0.25\_usage}$, $Stop_{0.10\_usage}$, and $Stop_{0.05\_usage}$.

5.6 Evaluation of test case prioritization approaches

For each approach, we report the number of non-dominated solutions obtained, the number of fitness evaluations of NSGA-II, and the execution time until the algorithm was stopped. In all cases, we report the median obtained from the 50 runs. The number of fitness evaluations corresponds to the number of test case orderings that were inspected during the optimization and represents the speed with which our approaches converge and their practical applicability. We also report the median of the AUC_{Feat} and AUC_{Time} objective functions for experiment 1, and of the *featRankSim* and AUC_{Time} objective functions for experiment 2. Following the literature recommendations [1], we used the Mann-Whitney U-test [33] and Cliff's delta d effect size [30, 46] to statistically compare our approaches. We adopt the thresholds for d as provided by Hess and Kromrey [23]: *negligible* if $|d| \leq 0.147$, *small* if $0.147 \ddot{Y} |d| \leq 0.33$, *medium* if $0.33 \ddot{Y} |d| \leq 0.474$, and *large* if $0.474 \ddot{Y} |d| \leq 1$.

5.7 Results

In this section we present the results of the two experiments that we performed. We report the results from all the 50 executions of the NSGA-II algorithm.

Experiment 1: *number of covered game features* **versus** *test execution time* **(without feature usage).** Figure 4 shows the non-dominated solutions found by 50 runs of NSGA-II for our approaches across different *per-feature coverage thresholds*. The $Stop_{0.05}$ approach found a median of 56.5 non-dominated solutions across all *per-feature coverage thresholds*, while the $Stop_{0.10}$ and $Stop_{0.25}$ approaches found a median of 34.0 and 18.0 non-dominated solutions. In terms of fitness evaluations (i.e., the number of test case orderings that were inspected until the algorithm was stopped), the $Stop_{0.05}$ approach had a median of 37,950 fitness evaluations across all *per-feature coverage thresholds*, while $Stop_{0.10}$ and $Stop_{0.25}$ had a median of 15,300 and 3,600 evaluations, respectively. As expected, the number of fitness evaluations for the $Stop_{0.05}$ approach was higher since the stopping criteria is stricter. The $Stop_{0.05}$ approach took a median of 85.98 seconds to execute, while $Stop_{0.10}$ and $Stop_{0.25}$ took a median of 32.11 seconds and 8.06 seconds, respectively.

Figure 4 shows that all our proposed approaches achieve better solutions than random search for all *per-feature coverage threshold* values. The $Stop_{0.25}$, $Stop_{0.10}$, and $Stop_{0.05}$ approaches found solutions with a better trade-off between the objective functions, i.e., with lower AUC_{Time} and larger AUC_{Feat}. For a *per-feature coverage threshold* of 50%, presented in Figure 4a, the $Stop_{0.05}$ approach has a median AUC_{Time} of 0.29, while the $Stop_{0.10}$ and $Stop_{0.25}$ approaches have larger median AUC_{Time}: 0.32 and 0.38, respectively. The $Random_{50}$ and $Random_{100}$ approaches have a median AUC_{Time} of 0.49 and 0.50, respectively. Regarding the AUC_{Feat}, $Stop_{0.05}$ has a median of 0.80, while $Stop_{0.10}$ and $Stop_{0.25}$ have medians of 0.77 and 0.69. The $Random_{50}$ and $Random_{100}$ approaches have a median AUC_{Feat} of 0.58 and 0.57, respectively. Among our proposed approaches, $Stop_{0.05}$ found the best solutions since it has the best trade-off between the AUC_{Time} and AUC_{Feat}. All the proposed approaches are significantly better than both random search approaches (p-value less than 0.05) and the Cliff's delta shows large effect sizes for both objective functions. Pairwise comparisons between the three approaches also show statistically significant differences with large effect sizes.

A similar behavior is observed for the *per-feature coverage thresholds* of 75%, 90%, and 100%, in Figures 4b, 4c, and 4d. In all these cases, the $Stop_{0.05}$ found the best solutions. However, the range in which the AUC_{Feat} lies gets smaller as we increase the *per-feature coverage threshold*. This happens because a higher threshold means that we need to execute more test cases to consider a feature as covered, so the number of covered features increases more slowly as we execute the ordered test cases, which yields a smaller AUC_{Feat} (as we explained in Section 5.3, in Figure 3).

Experiment 2: *number of covered highly-used game features* **versus** *test execution time* **(with feature usage).** Figure 5 shows the non-dominated solutions found by 50 runs of NSGA-II for our approaches. For this experiment, we did not use the *per-feature coverage threshold* since we did not use the AUC_{Feat} objective function. The $Stop_{0.05\_usage}$ approach found a median of 42.5 non-dominated solutions for all 50 runs, while the $Stop_{0.10\_usage}$ and $Stop_{0.25\_usage}$ approaches found a median of 29.0 and 18.0 non-dominated solutions. In terms of fitness evaluations, the $Stop_{0.05\_usage}$ approach had a median of 47,850 fitness evaluations, while $Stop_{0.10\_usage}$ and $Stop_{0.25\_usage}$ had a median of 12,800 and 3,650 evaluations, respectively. As expected, the number of fitness evaluations for the $Stop_{0.05\_usage}$ approach was higher since the stopping criteria is stricter. Also as expected, the $Stop_{0.05\_usage}$ approach took longer to execute until the stopping criteria were satisfied, with a median of 100.16 seconds. The $Stop_{0.10\_usage}$ and $Stop_{0.25\_usage}$ approaches took a median of 26.36 seconds and 7.47 seconds, respectively. We can see that our best approach ($Stop_{0.05\_usage}$) is feasible to be used in practice as it can find the best solutions in less than 2 minutes.

Figure 5 shows that all our proposed approaches achieve better solutions than random search since our approaches present better trade-offs between the objective functions (i.e., lower AUC_{Time} and larger *featRankSim*). The $Stop_{0.05\_usage}$ approach has a median AUC_{Time} of 0.29, while the $Stop_{0.10\_usage}$ and $Stop_{0.25\_usage}$ approaches have larger median AUC_{Time}: 0.33 and 0.38, respectively. The $Random_{50}$ and $Random_{100}$ approaches have a median AUC_{Time} of 0.49 and 0.50, respectively. Regarding *featRankSim*, $Stop_{0.05\_usage}$ has the largest median, with a value of 0.96, while $Stop_{0.10\_usage}$

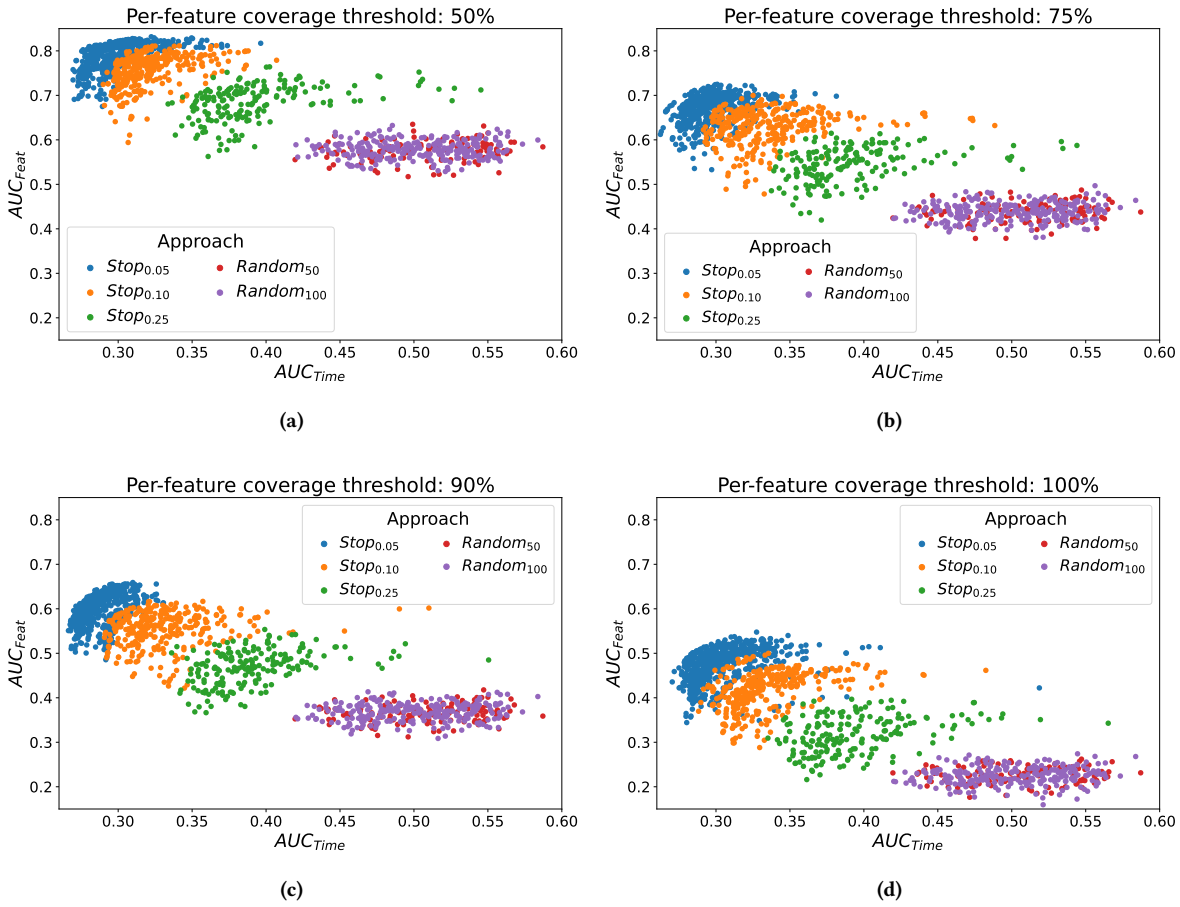


Figure 4: Experiment 1: Trade-off between AUC_{Time} and AUC_{Feat} for different *per-feature coverage thresholds* across our different approaches (without feature usage).

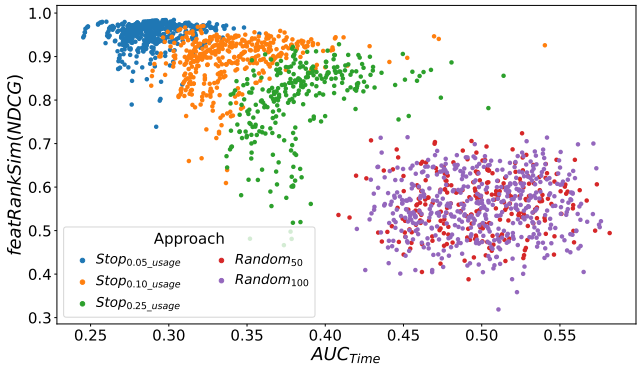


Figure 5: Experiment 2: Trade-off between AUC_{Time} and *featRankSim* across our approaches (with feature usage).

and $Stop_{0.25\_usage}$ have medians of 0.90 and 0.82. The $Random_{50}$ and $Random_{100}$ approaches have a median *featRankSim* of 0.56 and 0.55, respectively. Among our proposed approaches, $Stop_{0.05\_usage}$ found the best solutions since it has the best trade-off between the AUC_{Time} and *featRankSim*. This demonstrates that our $Stop_{0.05\_usage}$ approach can find test case orderings that cover highly-used game features early in the test execution (with a high *featRankSim* of 0.96) while keeping the cumulative test execution time small. All the proposed approaches are significantly better than the random search approaches (p-value less than 0.05) and the Cliff's delta shows large effect sizes for both objective functions. Pairwise comparisons between the three approaches also show statistically significant differences with large effect sizes.

6 DISCUSSION

In this section, we compare the results obtained with our experiments for prioritization with and without feature usage. Figure 6 shows the distribution of the *featRankSim* objective for our random approaches and for the non-dominated solutions obtained with our proposed approaches. In experiment 1, without feature usage, all the approaches (using a *per-feature coverage threshold* of 50%) achieve a similar median *featRankSim*: 0.55, 0.52, and 0.53 for $Stop_{0.25}$, $Stop_{0.10}$, and $Stop_{0.05}$, respectively. The random approaches achieve similar median values of *featRankSim*: 0.56 and 0.55 for $Random_{50}$ and $Random_{100}$, respectively. In contrast, the

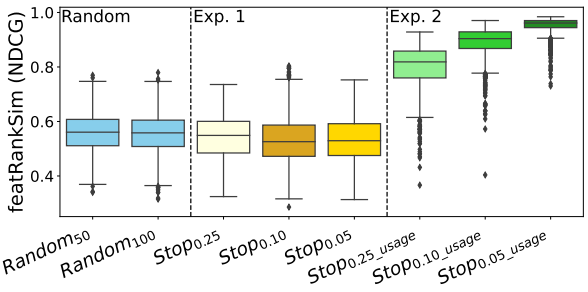


Figure 6: Distributions of *featRankSim* (NDCG) for our different approaches.

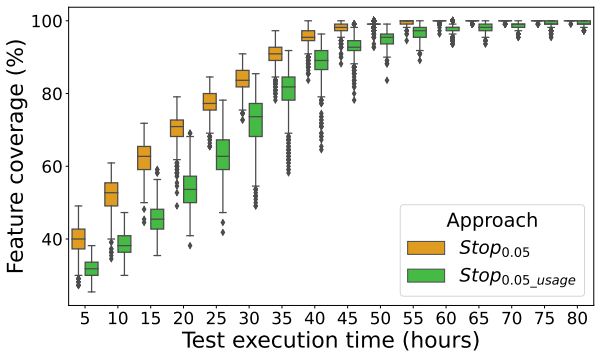


Figure 7: Comparison of game feature coverage for our best approaches in experiments 1 and 2.

approaches in experiment 2 achieve larger median *featRankSim* values: 0.82, 0.90, and 0.96 for $Stop_{0.25\_usage}$, $Stop_{0.10\_usage}$, and $Stop_{0.05\_usage}$, respectively. The larger *featRankSim* obtained by our approaches in experiment 2 shows that those approaches, in particular $Stop_{0.05\_usage}$, can successfully obtain test case orderings that cover highly-used game features early in the test execution.

Often during regression testing, the main constraint is the time available to execute test cases. Therefore, we discuss below how (1) the percentage of covered game features and (2) the percentage of coverage of the top-k most used game features change for different test execution times. For this analysis, we used the solutions found by our best approaches in experiments 1 and 2 ($Stop_{0.05}$, with a *per-feature coverage threshold* of 50%, and $Stop_{0.05\_usage}$).

Figure 7 shows how the percentage of covered game features changes for different execution times. A large amount of testing time is necessary to achieve large game feature coverage. For example, to achieve 100% coverage, a median of 55 hours for the $Stop_{0.05}$ approach and 70 hours for the $Stop_{0.05\_usage}$ approach are necessary. Even for lower coverage, a large amount of time is necessary. For example, to achieve 90% of coverage, $Stop_{0.05}$ requires 35 hours and $Stop_{0.05\_usage}$ requires 40 hours. We also observe that the last 10% of feat coverage requires an extremely large amount of time (approximately 30 hours additional testing time).

Achieving a high percentage of game feature coverage is not feasible in practice due to the large amount of time necessary, even for the $Stop_{0.05}$ approach that was optimized to achieve a large game feature coverage in the shortest time possible. If we analyze a more feasible scenario, with an available testing time of 5 hours, for example, $Stop_{0.05}$ covers a median of 40% of game features, while $Stop_{0.05\_usage}$ covers a median of 32% of game features. However, despite achieving a slightly smaller coverage for the same amount of time available, the test case orderings obtained with $Stop_{0.05\_usage}$ cover highly-used features earlier in the test execution compared to the solutions obtained with $Stop_{0.05}$. For example, if we analyze the coverage of the top-20% of the most used features (which gives the top-16 most used features) in 5 hours, $Stop_{0.05}$ covers only 68% of those features, while $Stop_{0.05\_usage}$ covers 93%. With one additional hour, $Stop_{0.05\_usage}$ covers all the top-20% most used features, while $Stop_{0.05}$ covers 75%. Therefore, with a feature usage-based test prioritization, we can find test case orderings that cover most of the highly-used features early in the test execution, which helps to avoid bugs that would affect a large number of users. Finally, QA engineers can achieve a higher coverage of game features and highly-used game features by parallelizing the test execution. For example, two QA engineers can execute independent test cases in parallel to achieve higher coverages within 5 hours.

7 USING OUR PRIORITIZATION APPROACH IN PRACTICE

We implemented an internal web application prototype of our approach to collect initial feedback on the outcome of our approach. We are now integrating the application into our industry partner's cloud infrastructure. Our approach can be used by QA engineers in an environment where resources (e.g., time) are restricted to obtain a set of test case orderings with the best trade-off between multiple objectives. Because of how we performed the optimization, in a situation of an early-stop of test execution, our approach ensures that the highest number of highly-used features are covered with the shortest time possible. The tester may also choose one particular test case ordering among the optimal orderings that maximizes one specific objective of interest to the detriment of the other. For example, an ordering that maximizes only the highly-used features may not have the shortest cumulative test execution time. Another practical aspect concerns a small subset of features that are critical to the *Prodigy Math game* of Prodigy Education (such as the "game membership purchase" feature). These features must be frequently tested during regression regardless of their usage. Therefore, we allow the users of our application to identify the critical features and retrieve the associated test cases before executing the optimization. Those test cases are then removed from the set of test cases that we use in the optimization (as they will always be executed before the optimized ordering).

8 RELATED WORK

Optimizing the execution of test cases in a manual testing scenario is of extreme importance [19, 21, 22]. However, only a few works investigated prioritization techniques for manual test cases which are described only in natural language (i.e., no source code is associated with them) [21, 27]. Hemmati et al. [21] investigated approaches to prioritize manual test cases using test execution history and Latent Dirichlet Allocation (LDA) [6] to find the topics related to test cases.

Our approach uses a pre-defined list of the application features to find the features being tested and leverages zero-shot models for that purpose, which does not require any manual analysis to further understand which features are covered by test cases (as LDA requires). Furthermore, our approach does not require the test execution history, which could be difficult to be accessed or not meaningful for manual test cases [62]. Lachmann et al. [27] investigated a supervised approach for prioritization of manual test cases using textual descriptions, test execution history, and the link between test cases and requirements. Their approach requires an expert to manually label test cases as (un)important to build the training set. In contrast, our approach does not require any manual data labeling nor the test execution history. Furthermore, none of the above mentioned works take into consideration the impact that bugs might have on users. We include the coverage of highly-used features in our approach, which helps to test those features more often and avoid impacting a large number of users.

Several works proposed prioritization techniques for test cases with associated source code [3, 8, 24, 32, 34, 38–40, 47, 51, 57, 63]. For instance, Marchetto et al. [34] performed test case prioritization with NSGA-II. However, differently from our work, they used code coverage and the link between requirements and source code in their approach. Instead, we do not have source code test cases and we used the link between natural language test cases and the covered features. We also include the coverage of highly-used features in our approach. Wang et al. [57] proposed to use multi-objective search algorithms for a resource-aware test case prioritization using four objectives. Their goal was to achieve a test case ordering for a limited time budget while maximizing the usage of the available test resources. In contrast, we focus on prioritizing manual test cases to maximize the coverage of the game features and the coverage of highly-used features.

9 THREATS TO VALIDITY

A threat to the **external validity** concerns the generalizability of our zero-shot methods and prioritization techniques. Using applications from other domains might yield different results. Another threat regards the used techniques. Using different classification models and optimization algorithms might achieve different results. Future studies should investigate if our approaches can be improved with other techniques.

A threat to the **internal validity** concerns the percentage of test cases that we consider sufficient to count a feature as covered. To mitigate this threat, we experimented with different percentages (from 50% up to 100%), but using other values will achieve different results. Also, companies that already have the link between test cases and covered features might use a different percentage. Another threat is related to the feature usage metric that we use (*total number of uses*). Other metrics can also capture feature usage, such as using the *number of unique users* who used a feature, which might achieve different orderings of features based on usage. Finally, using different conditions to stop the optimization algorithm (e.g., other p-value thresholds or other MDR ranges) might result in different non-dominated test case orderings.

10 CONCLUSION

In this paper, we propose a novel approach to prioritize natural language test cases. Our approach leverages zero-shot classification techniques to identify the features covered by the test cases of a game and uses this information to optimize the execution of test cases. In particular, we prioritize test cases that cover highly-used game features, in which bugs would affect a large group of players. Our findings show that we can successfully identify the game features covered by test cases with an ensemble of zero-shot models (an F-score of 76.1%). Also, our prioritization approaches can find test case orderings that cover highly-used game features early in the test execution while keeping the time required to execute test cases short. In practice, QA engineers and developers can use our approach to focus the test execution on test cases that cover game features that are relevant to players.

REFERENCES

[1] Andrea Arcuri and Lionel Briand. 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250.

[2] Andrea Arcuri and Gordon Fraser. 2011. On parameter tuning in search based software engineering. In *International Symposium on Search Based Software Engineering*. Springer, 33–47.

[3] Mojtaba Bagherzadeh, Nafiseh Kahani, and Lionel Briand. 2021. Reinforcement learning for test case prioritization. *IEEE Transactions on Software Engineering* (2021).

[4] Antonia Bertolino. 2007. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE'07)*. IEEE, 85–103.

[5] J. Blank and K. Deb. 2020. pymoo: Multi-Objective Optimization in Python. *IEEE Access* 8 (2020), 89497–89509.

[6] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *Journal of machine Learning research* 3, Jan (2003), 993–1022.

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[8] Benjamin Busjaeger and Tao Xie. 2016. Learning for test prioritization: an industrial case study. In *Proceedings of the 2016 24th ACM SIGSOFT International symposium on foundations of software engineering*. 975–980.

[9] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*. PMLR, 1597–1607.

[10] Nan Cui, Yuze Jiang, Xiaodong Gu, and Beijun Shen. 2022. Zero-Shot Program Representation Learning. In *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. 60–70. https://doi.org/10.1145/3524610.3527888

[11] Kalyanmoy Deb. 2011. Multi-objective optimisation using evolutionary algorithms: an introduction. In *Multi-objective evolutionary optimisation for product design and manufacturing*. Springer, 3–34.

[12] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197.

[13] Peter Devine and Kelly Blincoe. 2022. Unsupervised Extreme Multi Label Classification of Stack Overflow Posts. (2022).

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[15] Jackson Antonio do Prado Lima and Silvia Regina Vergilio. 2020. A multi-armed bandit approach for test case prioritization in continuous integration environments. *IEEE Transactions on Software Engineering* (2020).

[16] Michael G Epitropakis, Shin Yoo, Mark Harman, and Edmund K Burke. 2015. Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 234–245.

[17] Vahid Garousi and Junji Zhi. 2013. A survey of software testing practices in Canada. *Journal of Systems and Software* 86, 5 (2013), 1354–1376.

[18] José L Guerrero, Jesús García, Luis Martí, José Manuel Molina, and Antonio Berlanga. 2009. A stopping criterion based on Kalman estimation techniques with several progress indicators. In *Proceedings of the 11th Annual conference on Genetic and Evolutionary Computation*. 587–594.

[19] Roman Haas, Daniel Elsner, Elmar Juergens, Alexander Pretschner, and Sven Apel. 2021. How can manual testing processes be optimized? Developer survey,

optimization guidelines, and case studies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1281–1291.

[20] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 1–61.

[21] Hadi Hemmati, Zhihan Fang, and Mika V Mantyla. 2015. Prioritizing manual test cases in traditional and rapid release environments. In *Proceedings of the 8th Int'l Conference on Software Testing, Verification and Validation (ICST)*. 1–10.

[22] Hadi Hemmati and Fatemeh Sharifi. 2018. Investigating NLP-based approaches for predicting manual test case failure. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 309–319.

[23] Melinda R Hess and Jeffrey D Kromrey. 2004. Robust confidence intervals for effect sizes: A comparative study of Cohen's d and Cliff's delta under non-normality and heterogeneous variances. In *annual meeting of the American Educational Research Association*, Vol. 1. Citeseer.

[24] Rubing Huang, Dave Towey, Yinyin Xu, Yunan Zhou, and Ning Yang. 2022. Dissimilarity-based test case prioritization through data fusion. *Software: Practice and Experience* 52, 6 (2022), 1352–1377.

[25] Kalervo Järvelin and Jaana Kekäläinen. 2002. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems (TOIS)* 20, 4 (2002), 422–446.

[26] Joshua D Knowles and David W Corne. 2000. Approximating the nondominated front using the Pareto archived evolution strategy. *Evolutionary computation* 8, 2 (2000), 149–172.

[27] Remo Lachmann, Sandro Schulze, Manuel Nieke, Christoph Seidl, and Ina Schaefer. 2016. System-level test case prioritization using machine learning. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 361–368.

[28] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461* (2019).

[29] Zheng Li, Yi Bian, Ruilian Zhao, and Jun Cheng. 2013. A fine-grained parallel multi-objective test case prioritization on GPU. In *International Symposium on Search Based Software Engineering*. Springer, 111–125.

[30] Jeffrey D Long, Du Feng, and Norman Cliff. 2003. Ordinal analysis of behavioral data. (2003).

[31] Finlay Macklon, Mohammad Reza Taesiri, Markos Viggiato, Stefan Antoszko, Natalia Romanova, Dale Paas, and Cor-Paul Bezemer. 2022. Automatically Detecting Visual Bugs in HTML5 Canvas Games. In *2022 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[32] Mostafa Mahdieh, Seyed-Hassan Mirian-Hosseinabadi, and Mohsen Mahdieh. 2022. Test case prioritization using test case diversification and fault-proneness estimations. *Automated Software Engineering* 29, 2 (2022), 1–43.

[33] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[34] Alessandro Marchetto, Md Mahfuzul Islam, Waseem Asghar, Angelo Susi, and Giuseppe Scanniello. 2015. A multi-objective technique to prioritize test cases. *IEEE Transactions on Software Engineering* 42, 10 (2015), 918–940.

[35] Luis Martí, Jesús García, Antonio Berlanga, and José M Molina. 2009. An approach to stopping criteria for multi-objective optimization evolutionary algorithms: The MGBM criterion. In *2009 IEEE congress on evolutionary computation*. IEEE, 1263–1270.

[36] Frank McSherry and Marc Najork. 2008. Computing information retrieval performance measures efficiently in the presence of tied scores. In *European conference on information retrieval*. Springer, 414–421.

[37] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* 26 (2013).

[38] Breno Miranda, Emilio Cruciani, Roberto Verdecchia, and Antonia Bertolino. 2018. FAST approaches to scalable similarity-based test case prioritization. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 222–232.

[39] Tanzeem Bin Noor and Hadi Hemmati. 2015. A similarity-based approach for test case prioritization using historical failure data. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 58–68.

[40] Safa Omri and Carsten Sinz. 2022. Learning to Rank for Test Case Prioritization. In *2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 16–24.

[41] Luca Pascarella, Fabio Palomba, Massimiliano Di Penta, and Alberto Bacchelli. 2018. How is video game development different from software development in open source?. In *Proceedings of the 15th Int'l Conference on Mining Software Repositories (MSR)*. 392–402.

[42] Cristiano Politowski, Fabio Petrillo, and Yann-Gäel Guéhéneuc. 2021. A Survey of Video Game Testing. *arXiv preprint arXiv:2103.06431* (2021).

[43] Dipesh Pradhan, Shuai Wang, Shaukat Ali, Tao Yue, and Marius Liaaen. 2018. CBGA-ES+: A cluster-based genetic algorithm with non-dominated elitist selection for supporting multi-objective test optimization. *IEEE Transactions on Software Engineering* 47, 1 (2018), 86–107.

[44] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*. PMLR, 8748–8763.

[45] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. https://arxiv.org/abs/1908.10084

[46] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, Jeff Skowronek, and Linda Devine. 2006. Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen's d indices the most appropriate choices. In *annual meeting of the Southern Association for Institutional Research*. Citeseer, 1–51.

[47] Ripon K Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E Perry. 2015. An information retrieval approach for regression test prioritization based on program changes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 268–279.

[48] Abdel Salam Sayyad, Tim Menzies, and Hany Ammar. 2013. On the value of user preferences in search-based software engineering: A case study in software product lines. In *2013 35Th international conference on software engineering (ICSE)*. IEEE, 492–501.

[49] Seung Yeob Shin, Shiva Nejati, Mehrdad Sabetzadeh, Lionel C Briand, and Frank Zimmer. 2018. Test case prioritization for acceptance testing of cyber physical systems: a multi-objective search-based approach. In *Proceedings of the 27th acm sigsoft international symposium on software testing and analysis*. 49–60.

[50] Ravjot Singh, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E Hassan. 2016. Optimizing the performance-related configurations of object-relational mapping frameworks using a multi-objective genetic algorithm. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. 309–320.

[51] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 12–22.

[52] Mohammad Reza Taesiri, Finlay Macklon, and Cor-Paul Bezemer. 2022. CLIP meets GamePhysics: Towards bug identification in gameplay videos using zero-shot transfer learning. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE, 46–57.

[53] Simon Varvaressos, Kim Lavoie, Sébastien Gaboury, and Sylvain Hallé. 2017. Automated bug finding in video games: A case study for runtime monitoring. *Computers in Entertainment (CIE)* 15, 1 (2017), 1–28.

[54] Markos Viggiato, Dale Paas, Chris Buzon, and Cor-Paul Bezemer. 2022. Identifying similar test cases that are specified in natural language. *IEEE Transactions on Software Engineering* (2022).

[55] Markos Viggiato, Dale Paas, Chris Buzon, and Cor-Paul Bezemer. 2022. Using natural language processing techniques to improve manual test case descriptions. In *International Conference on Software Engineering-Software Engineering in Practice (ICSE-SEIP) Track.(May 8, 2022)*.

[56] Tobias Wagner, Heike Trautmann, and Luis Martí. 2011. A taxonomy of online stopping criteria for multi-objective evolutionary algorithms. In *International Conference on Evolutionary Multi-Criterion Optimization*. Springer, 16–30.

[57] Shuai Wang, Shaukat Ali, Tao Yue, Øyvind Bakkeli, and Marius Liaaen. 2016. Enhancing test case prioritization in an industrial setting with resource awareness and multi-objective search. In *Proceedings of the 38th International Conference on Software Engineering Companion*. 182–191.

[58] Yining Wang, Liwei Wang, Yuanzhi Li, Di He, Wei Chen, and Tie-Yan Liu. 2013. A theoretical analysis of NDCG ranking measures. In *Proceedings of the 26th annual conference on learning theory (COLT 2013)*, Vol. 8. 6.

[59] Kristian Wiklund, Sigrid Eldh, Daniel Sundmark, and Kristina Lundqvist. 2017. Impediments for software test automation: A systematic literature review. *Software Testing, Verification and Reliability* 27, 8 (2017), e1639.

[60] Yongqin Xian, Zeynep Akata, Gaurav Sharma, Quynh Nguyen, Matthias Hein, and Bernt Schiele. 2016. Latent embeddings for zero-shot classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 69–77.

[61] Huang Xie and Tuomas Virtanen. 2021. Zero-shot audio classification via semantic embeddings. *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 29 (2021), 1233–1242.

[62] Yilin Yang, Xinhai Huang, Xuefei Hao, Zicong Liu, and Zhenyu Chen. 2017. An industrial study of natural language processing based test case prioritization. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 548–549.

[63] Ahmadreza Saboor Yaraghi, Mojtaba Bagherzadeh, Nafiseh Kahani, and Lionel Briand. 2022. Scalable and Accurate Test Case Prioritization in Continuous Integration Contexts. *IEEE Transactions on Software Engineering* (2022).

[64] Wenpeng Yin, Jamaal Hay, and Dan Roth. 2019. Benchmarking zero-shot text classification: Datasets, evaluation and entailment approach. *arXiv preprint*

arXiv:1909.00161 (2019).

[65] Zhi Quan Zhou, Chen Liu, Tsong Yueh Chen, TH Tse, and Willy Susilo. 2020. Beating random test case prioritization. *IEEE Transactions on Reliability* 70, 2 (2020), 654–675.