Wrocław University of Science and Technology
Faculty of Information and Communication Technology

Field of study:	**Applied Computer Science (IST)**
Speciality:	**Software Engineering (IO)**

MASTER THESIS

Test case prioritization

Tomasz Chojnacki

Supervisor

dr hab. inż. Lech Madeyski, prof. uczelni

Keywords: test case prioritization, continuous integration, ensemble methods, code representation

WROCŁAW 2025

Abstract

Test case prioritization

Context: Test case prioritization (TCP) is a technique widely used by software development organizations to accelerate regression testing. *Objectives:* The thesis aims to systematize existing TCP knowledge and to propose and empirically evaluate a new TCP approach. *Methods:* We conduct a systematic review (SR) on TCP, implement a comprehensive platform for TCP research (TCPFramework), analyze existing evaluation metrics and propose two new ones (rAPFD$_C$ and ATR), investigate the use of code representation for TCP, and develop a family of ensemble TCP methods called approach combinators. *Results:* The SR helped identify 324 studies related to TCP. The proposed techniques were evaluated on the RTPTorrent dataset, achieving performance comparable to the current state of the art (in terms of rAPFD$_C$, NTR, and ATR), while using a distinct, novel approach. *Conclusions:* The proposed methods can be used efficiently for TCP, reducing the time spent on regression testing by up to 2.7%. Approach combinators offer significant potential for improvements in future TCP research, due to their composability.

Streszczenie

Priorytetyzacja przypadków testowych

Kontekst: Priorytetyzacja przypadków testowych (PPT) jest techniką szeroko używaną przez podmioty tworzące oprogramowanie w celu przyspieszenia procesu testowania regresyjnego. *Cele:* Praca ma na celu usystematyzowanie istniejącej wiedzy na temat PPT oraz wytworzenie i empiryczną ocenę nowej metody priorytetyzacji przypadków testowych. *Metody:* Przeprowadzono systematyczny przegląd literatury (SPL) dotyczącej PPT, zaimplementowano uniwersalną platformę badawczą do PPT (TCPFramework), przeanalizowano metryki używane do oceny podejść oraz zaproponowano dwie nowe (rAPFD$_C$ i ATR), zbadano wykorzystanie reprezentacji kodu do PPT oraz skonstruowano rodzinę zespołowych metod PPT zwaną kombinatorami podejść. *Wyniki:* Wykonany SPL pomógł zidentyfikować 324 badania dotyczące PPT. Zaproponowane techniki zostały ocenione na zbiorze RTPTorrent, osiągając wyniki porównywalne z obecnym stanem sztuki (pod względem rAPFD$_C$, NTR i ATR), jednocześnie stosując odmienne, nowe podejście. *Wnioski:* Zaproponowane w pracy podejścia mogą zostać skutecznie wykorzystane do PPT, zmniejszając czas testowania regresyjnego o nawet 2.7%. Kombinatory podejść oferują duży potencjał rozwoju z uwagi na możliwość łączenia istniejących metod.

Contents

Introduction

The introduction describes the motivation and the fundamentals behind the use of test case prioritization (TCP) in regression testing and outlines the aim and scope of the thesis.

Background

Software bugs can frustrate users and reduce their satisfaction [1] or even cause health and safety risks in certain areas, such as avionics or medicine. However, bugs can occur in all software systems [2]. To ensure that a program behaves as expected, software testing is performed. In particular, regression testing (RT) is commonly used to ensure that changes to the system did not introduce any unintended behavior. Regression testing can be performed at any test level (unit, integration, etc.) and can verify compliance with functional and non-functional requirements [3].

Due to the volume of tests executed, RT can be costly, both in terms of time and money, with estimates stating that it may comprise between 33% and 50% of software expenses [3–5]. There exist software systems that contain several thousands of test cases in which the execution of the entire test suite takes hours or days [6–8]. Regression testing is at the same time an important part of Agile and continuous integration (CI), where automated RT suites are run on each build [3, 6, 7, 9]. However, CI processes often require rapid turnaround, to avoid delays in build cycles, and to provide software developers with the necessary feedback [8, 9].

Common approaches to speed up the RT process include test suite minimization, test case selection, and test case prioritization [3], as systematized by Yoo and Harman [10] and presented below. *Test suite minimization* (TSM), also known as test suite reduction (TSR), reduces the size of the test suite by permanently removing redundant or low-quality test cases. However, certain test cases might be irrelevant to the currently evaluated build, but important for a different code change. Following this insight, *test case selection* (TCS), sometimes referred to as regression test selection (RTS), chooses a subset of the test suite for execution, which contains test cases relevant to the change being made. This is often achieved through a white-box analysis of modified code; for this reason, we say that TCS is modification-aware. The main difference between TSM and TCS is that TSM is a permanent one-time process, whereas TCS is run on each CI build and might select different test cases each time.

Finally, *test case prioritization* (TCP), also called regression test prioritization (RTP), aims to reorder test cases, to produce the permutation which would detect possible faults as early as possible. It allows all test cases to run in the worst case, but also permits early termination, for example, when the first failing case is encountered [10]. TCP is safer than previously discussed methods, as an excluded test case might still have identified a fault if it were not omitted [11]. The usage of TCP in the context of RT is shown in Figure 1. This technique is the main focus of the thesis.

As further discussed in Chapter 1, over the years hundreds of TCP methods have been researched. The technique is also used by practitioners, finding success in large software development entities such as Google [12] and Microsoft [13]. However, there are still opportunities for improvement.

The aim of the thesis

Consequently, the thesis has two main objectives. The first is to perform a systematic literature review in the field of TCP. The purpose of this literature study is to summarize the current state of the art (in terms of datasets and approaches) and to identify any gaps in existing studies. The second, larger, aim is to propose, implement, and empirically evaluate a new technique for the prioritization of test cases.

The scope of the thesis

The thesis is structured as follows: Firstly, after the introduction, the overview of the literature review and the related TCP work are described in Chapter 1, providing a necessary context for further research. Then, the proposed methods and their implementations are presented during Chapter 2. The solutions are subsequently analyzed and evaluated in Chapter 3. Finally, the approaches are discussed alongside further work suggestions in Chapter 4, followed by the conclusion. Appendix A contains the systematic literature review protocol constructed during Chapter 1 and Appendix B provides the reproduction instructions for the research.

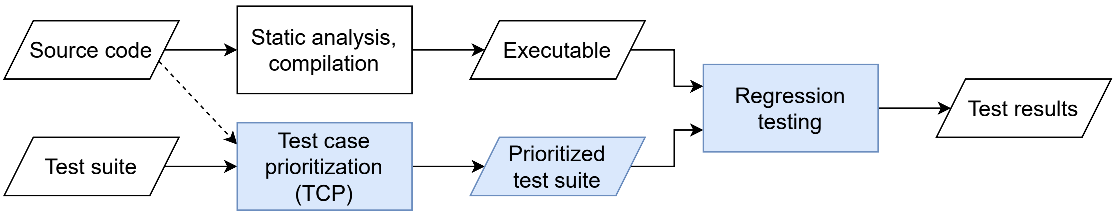


Fig. 1. Test case prioritization in the regression testing process (own work, based on [14]).

Highlights

The main contributions of the thesis consist of:

- A new systematic literature review on the topic of test case prioritization that follows modern review guidelines and is among the largest reviews of the TCP literature in terms of the count of identified works.
- Implementation of an open-source, unified TCP approach development and evaluation infrastructure (TCPFramework) that can be used to test various prioritization methods on the state-of-the-art RTPTorrent dataset.
- Review and analysis of existing TCP evaluation metrics, including recent proposals, such as rAPFD, RPA, NRPA, and NTR.
- Proposition of two new TCP evaluation metrics, $r\text{APFD}_C$ (to assess the fault detection capability of prioritized suites) and ATR (to rate the time efficiency of approaches).
- Investigation of the use of code representation distances to prioritize test cases and resolve ordering ties, including a comparison of achieved performance depending on different model parameter configurations.
- Introduction of the mental model of TCP approach combinators, consisting of three method families (mixers, interpolators, and tiebreakers) requiring no prior training, along with suggestions of sample models built with this strategy.
- Empirical evaluation of proposed approach combinator models, which achieve results comparable to current state of the art, and almost three times better than the random approach, reducing the time spent on regression testing by up to 2.7%.

The research carried out during the creation of this thesis will be additionally submitted to the Journal of Systems and Software under the name *Unifying Test Case Prioritization with Approach Combinators*.

Reproducible research (RR) can address problems with the validity of findings in software engineering research [15]. To support RR and good scientific practices, the thesis is backed by project files in the form of code, scripts, and systematic review artifacts. The data is available at `https://github.com/LechMadeyski/MSc25TomaszChojnacki`.

Acknowledgments

I express my gratitude to everyone who helped in any way with the writing of this thesis. Firstly, special thanks go to the thesis supervisor, Professor Lech Madeyski, for assistance with the development of the thesis, as well as for the help throughout the entire course of my studies. This endeavor would not have been possible without my friends and family, who supported me throughout the research process. In particular, I would like to acknowledge Przemysław, Jakub, and especially Kamila for their help.

1. Related work

The aim of this chapter is to describe the process of collecting works related to the research area and to summarize the available TCP datasets and algorithms. Firstly, the motivation and the process of performing the systematic literature review are discussed. Then, various steps of the review are described in detail, after which selected papers from various research sub-areas are covered.

1.1. Systematic literature review

In order to summarize the current state of the art of TCP for use in the latter section of this chapter, and to locate opportunities for further improvement to subsequently implement in Chapter 2, a systematic literature review was undertaken.

Systematic literature review, also known as a systematic review (SR), is a technique for discovering and evaluating scientific research related to a specific topic. Compared to traditional unsystematic reviews, SRs are conducted formally, using a set of methodological steps and approaches listed in a review protocol [16, 17]. This type of survey was first popularized at the beginning of the 20th century in medicine and then introduced to the field of software engineering (SE) nearly 100 years later. In 2004 Kitchenham [18] adapted the evidence-based approach to SE, later producing guidelines [17] for conducting systematic reviews in software engineering [16], hereinafter called GSRSE. There exist multiple refinements to the guidelines, with the latest development, called Software Engineering Guidelines for REporting Secondary Studies (SEGRESS), published in 2023 by Kitchenham et al. [19]. The newest installment does not discourage the use of GSRSE, but we focus on SEGRESS regardless, as it is more modern.

Unfortunately, since SR guidelines aid in conducting standalone SRs by large groups of researchers, they are not perfectly fit for a master's thesis, where the literature survey only supplements a proposed solution, and the research is carried out by a single author. For example, the checklist components stating the requirements for the title and abstract of the document do not apply. The GSRSE addresses the issue, mentioning that single researchers might want to use a lighter version of the guidelines and focus only on the most important points. Such steps were taken, which means that some recommendations of the guidelines were not considered.

1.1.1. Systematic review planning phase

The systematic review usually begins with the planning phase, where the research questions are listed and the research protocol is written [16, 17]. Expanding on the original motivation for performing the SR, the following research questions were stated:

- SRRQ1: *What are the available datasets and subject programs for TCP?*
- SRRQ2: *What are the state-of-the-art TCP algorithms?*

To avoid confusion with the research questions related to the effectiveness of the solution proposed in Chapter 3, the objectives defined above will be discriminated using a prefix and referred to as *systematic review research questions* (SRRQs).

It is worth noting that the presented SRRQs are very high-level. Research objectives are usually much narrower, focusing on a comparison of different methods or evaluating the quality of a technique [17]. However, in this SR, the author instead focused on compiling catalogs of TCP algorithms and datasets to aid in further research, which in turn produced such wide SRRQs.

Upon formulating the research objectives, work on the previously mentioned survey protocol began. According to the available recommendations, the protocol should, among others, describe [16, 17]:

- the motivation for the SR,
- the previously defined research questions,
- search methods (including study selection criteria),
- data extraction and analysis strategies.

The goals of the protocol are to reduce the likelihood of researcher bias, make the survey easier to reproduce, and aid in the latter parts of the SR. The protocol should be evaluated to ensure its consistency and correctness [16]. Addressing this need, Kitchenham et al. [17] note that in the case of PhD research, the review of the protocol by the supervisor may be sufficient. The same process was used in this thesis.

The GSRSE provides its own protocol structure template, while SEGRESS instead recommends using the checklist found in the PRISMA-P statement [20]. The SR protocol was planned using the GSRSE template and then adjusted to fit the PRISMA-P rules, with a focus on SEGRESS recommendations. This resulted in the following protocol structure:

1. Introduction (rationale for the review and the list of research questions).
2. Search process (search method and sources, eligibility criteria, collected data).
3. Data analysis (produced tabulations and charts, record availability).

The protocol was then constructed, before continuing with the search process. It should be noted that the protocol was written in its entirety before the subsequent SR steps were carried out to reduce bias. The finished protocol is available in Appendix A.

1.1.2. Search method description

The search process used in the SR follows the snowballing approach recommended by Wohlin [21]. The iterative nature of this technique provides stable results, and the usage of Google Scholar reduces publisher bias [21].

The snowballing procedure takes a collection of papers that describe a common topic, called the *start set* and applies alternating iterations of *backward snowballing* and *forward snowballing* to expand the list of relevant works. The process stops once no new articles are discovered in a given iteration [21].

Although there is no perfect way to identify a good start set, the author suggests using Google Scholar to construct it. Backward snowballing refers to the process of using the reference list of existing papers to identify new ones. In other words, it looks at papers published before the examined publication, which are related to it. Forward snowballing consists of finding papers citing the current one, which means the works published later than it. Google Scholar is used for this purpose to avoid publisher bias. Each encountered article is evaluated using the inclusion and exclusion criteria noted in the review protocol. After the iterative process, the guidelines recommend directly contacting the featured researchers. This process was not explicitly named by Wohlin, but will be further referred to as *author search*. Data extraction is also mentioned as a separate stage. However, the guidelines later suggest conducting it in parallel with the other stages, whenever a new article is identified [21]. The original snowballing procedure is shown on Figure 1.1.

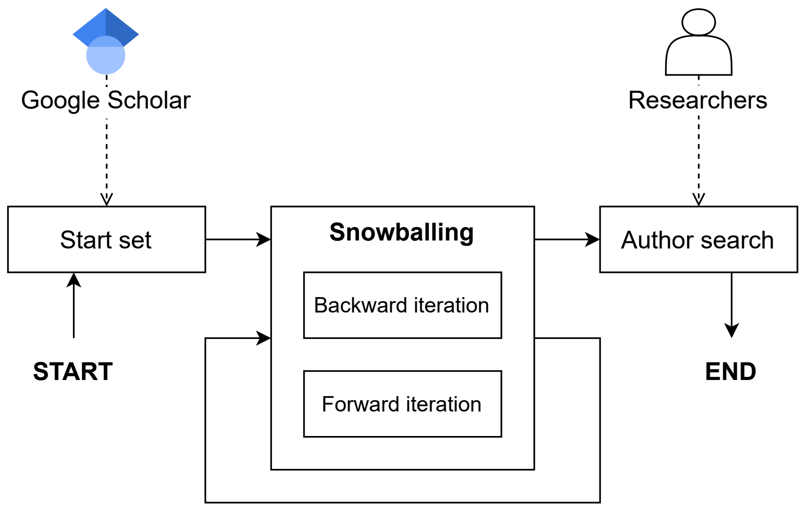


Fig. 1.1. Stages of the snowballing procedure (own work, based on [21]).

In order to adapt the search process to the circumstances of a master's thesis, a few modifications, justified below, were introduced. It should be noted that these changes always influence only the recommendations, never breaking any hard rules established by the guidelines.

First, since a list of unmethodically selected publications closely related to the studied research area was included in the original submission of the thesis topic, it is used instead of Google Scholar to form the snowballing start set. This list will hereinafter be referred to as *initial papers*, and includes contributions from Yang et al. [22], Marijan [9], Yaraghi et al. [14], and Bagherzadeh et al. [8]. Keywords extracted from these articles are used to construct a Scopus search query, yielding the start set. This modification aims to reduce bias through the use of a sophisticated database query string instead of an ad hoc search with Google Scholar.

Secondly, the author search consisting of directly contacting the researchers associated with identified papers is infeasible to conduct in the timeframes of a master's thesis. Alternatively, all works by the authors appearing at least five times in the final list are to be analyzed manually without inquiring them. Unfortunately, this increases the possibility of oversight of a relevant paper, but since the author search acts only supplementally to the main snowballing procedure, the risk should not be significant.

Initially, since TCP is a highly practical topic, there was a plan to introduce an additional search process stage dedicated to the survey of industrial solutions published informally, in the form of blogs or other types of gray literature. However, it was quickly discovered that such solutions are commonly biased in favor of the company or published alongside actual research papers (for example [12, 23]).

The proposed modifications either aim to improve the reliability of the SR or are caused by practical limitations. The new search process is shown on Figure 1.2. Compared to Figure 1.1, the modified components are colored red, while the newly introduced component is colored green.

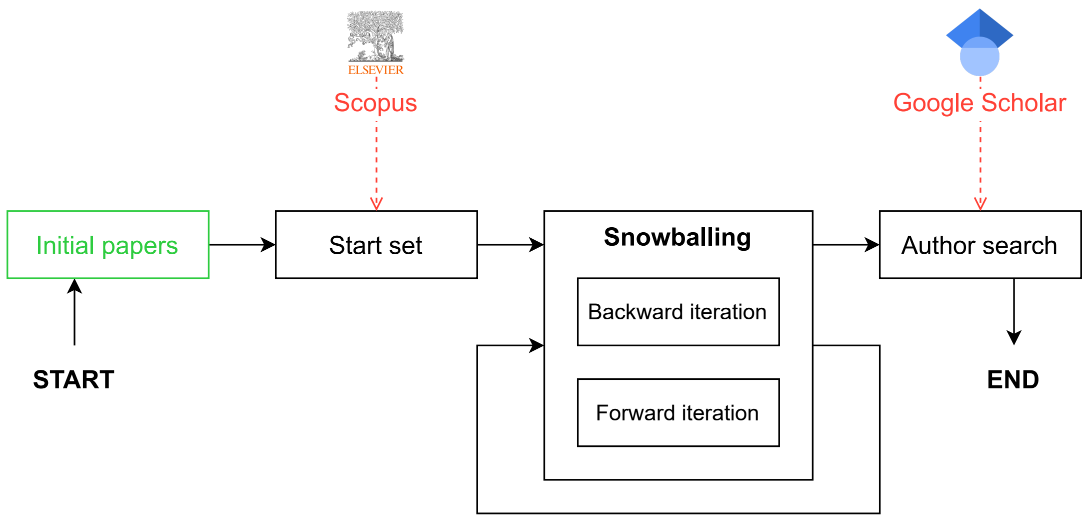


Fig. 1.2. Stages of the altered search procedure (own work).

1.1.3. Course of the literature study

As mentioned in the previous sections, the study started with a set of initial papers, which were thoroughly analyzed. Their keywords, outlined in Table 1.1, were used to derive the Scopus search string. Excluding keywords describing a specific method of performing TCP (e.g., *reinforcement learning*) and keywords providing metadata about a certain paper (e.g., *empirical study*), we can conclude that the most common keywords are: *test case prioritization*, *test prioritization*, and *continuous integration* (CI). After early experimentation, it was observed that including CI in the database query almost doubled the count of articles to review, while simultaneously resulting in a lower proportion of relevant papers among found documents.

Table 1.1. Keywords within the initial paper set.

Initial paper	**Keywords**
Yang et al. [22]	*test case prioritization*, *code representation*, *information retrieval*, *empirical study*
Marijan [9]	*machine learning*, *neural networks*, *support vector regression*, *gradient boosting*, *learning to rank*, *continuous integration*, *software testing*, *regression testing*, *test prioritization*, *test selection*, *test optimization*
Yaraghi et al. [14]	*machine learning*, *software testing*, *test case prioritization*, *test case selection*, *continuous integration*
Bagherzadeh et al. [8]	*continuous integration*, *CI*, *reinforcement learning*, *test prioritization*

Based on the requirements stated in the SR protocol (with respect to the subject area, publication stage, language and year) and the extracted keywords, the Scopus search text found in Listing 1.1 was constructed. The search returned a total of 440 documents. Most notably, all initial papers were also returned by the query. This ensures that the filters were not too restrictive.

```
( TITLE-ABS-KEY ( "test case prioritization" )
    OR TITLE-ABS-KEY ( "test prioritization" ) )
AND ( LIMIT-TO ( SUBJAREA , "COMP" ) )
AND ( LIMIT-TO ( PUBSTAGE , "final" ) )
AND ( LIMIT-TO ( LANGUAGE , "English" ) )
AND PUBYEAR > 2019
```

Listing 1.1. The Scopus search string used to find the snowballing start set.

The result list was then manually traversed, using inclusion and exclusion criteria, to identify works that could act as the start set. This choice could possibly be biased, since it involves a manual selection done by a single author. However, it is not likely to be a major concern, since the leftover articles should be picked up by the snowballing process either way. The guidelines of Wohlin [21] state that a good start set should include papers from different communities, publishers, years, and authors. These recommendations were taken into account, and the author attempted to cluster the found papers and avoid picking multiple papers from a given group.

The following articles were selected as the start set: Bagherzadeh et al. [8] (STS1), Huang et al. [24] (STS2), Vescan et al. [25] (STS3), Mukherjee and Patnaik [26] (STS4). It should be noted that STS1 was previously included in the initial paper set, which is permitted by the SR protocol. Thus, while the start set includes four papers, only three new papers were introduced in this stage. The constructed set spans multiple years, from 2020 (STS2) to 2024 (STS3). There are no common authors between any two papers in the list, and the identified authors come from different communities. The works were all published in distinct venues. Additionally, the entries focus on separate areas: STS1 covers reinforcement learning in TCP, STS2 leverages code coverage, STS3 attempts to unify existing TCP approaches into a single framework, and STS4 is a literature review. The selected works contain a total of 274 references and 222 citations (duplicates included), which should provide sufficient resources for the first iteration of snowballing.

We introduce the notation of `n/m`, which has the following meaning: *`n` newly identified, unique papers were included after evaluating a total of `m` references and citations (duplicates included).* Subsequently, the first iteration of snowballing was conducted. The backward snowballing yielded 25/78 papers for STS1, 20/60 papers for STS2, 11/35 papers for STS3, and 48/101 papers for STS4. Next, forward snowballing helped identify 29/99 papers for STS1, 12/36 papers for STS2, 0/0 papers for STS3 (meaning work of Vescan et al. [25] had no citations), and 19/87 for STS4. The decreasing number of papers included in the latter start set entries is partly due to the fact that some of their references might have been included from previous papers. In total, backward and forward snowballing yielded 104/274 and 60/222 papers appropriately for a total of 168 articles analyzed in the first iteration. During this stage, it was decided to diverge from the protocol and add an exclusion criterion, stating that articles retracted from publication should be excluded. Furthermore, quantum algorithms were excluded as there would be no way to reproduce or improve them within the bounds of the thesis.

In the second backward iteration, 111 papers were included from a total of 6 734 references. The works of Khatibsyarbini et al. [5] and Singh et al. [27] provided the largest amount of new papers, with, respectively, 22/100 and 23/320 of their references being included. That is, these two publications alone yielded more than a third of all papers identified in this stage. Interestingly, both are systematic reviews of the literature. During this iteration, another divergence from the protocol occurred. It was decided to exclude replication studies to avoid duplicate counting of the same algorithm or dataset. In addition, studies adapting a previously identified technique to a particular programming language or testing framework were excluded, as they do not directly assist in answering SRRQ2.

During the second forward iteration, 29 new articles were included from a total of 21 393 analyzed citations. It must be emphasized that the vast majority of these citations were immediately skipped because they had already been investigated before. The largest amount of new inclusions, namely 10/1 817, came from the seminal paper by Rothermel et al. [28]. This fact should not be surprising, since it is one of the oldest publications on TCP. Similarly to the corresponding backward search, this stage used all works included in the first iteration, so it can be split into two parts: the articles obtained previously from backward and forward snowballing. Clear differences were observed between these two parts. The first, which consisted of 104 articles with a median citation count greater than 100, resulted in 28 inclusions. In contrast, the forward analysis of the initial forward iteration, conducted on 60 papers with a much lower median citation count of only three, produced just one new entry. This is reasonable since the forward snowballing is naturally bound by the date of the search.

The third snowballing iteration produced much fewer articles, with only 5/4 730 and 8/4 437 identified papers for the backward and forward search, respectively. The included references and citations were spread in terms of sources, with every prior paper giving at most one new identification. The snowballing process was concluded after the fourth iteration, during which no new works were identified, after analyzing 425 references and 173 citations. During these stages, no divergences were made from the research protocol.

Subsequently, the author search was conducted in a form that complies with the requirements of the review protocol. This means that the list of authors that were attributed to at least five publications out of more than 300 currently identified was collected. The list contains 20 researchers, with the most prominent being Lu Zhang, co-authoring 13 articles, as well as Gregg Rothermel and Dan Hao, both contributing to 12 works. The authors' Google Scholar pages (and, where possible, their ORCID pages and personal websites) were traversed in search of new papers that conformed to the inclusion and exclusion criteria. In total, the records of 3 769 articles were checked, which did not result in any new inclusions. Unfortunately, the mentioned count is highly inflated, since many of the entries were not written in English or were wrongly classified by Scholar as scientific publications. Despite not producing new additions, many previously included articles were seen in this stage, indicating the effectiveness of the snowballing process.

This step concluded the entire search process. The detailed results of the survey are presented in the following section. In general, only minor divergences were made from the research protocol, all of which were documented above.

1.1.4. Results of the systematic review

During the entire snowballing process, a total of 324 articles were identified that comply with the previously specified inclusion and exclusion criteria (and their additional addenda), with 292 primary studies and 32 secondary studies. This was achieved through an analysis of more than 42 thousand references and citations, the vast majority of which were duplicates. The count of papers identified at each stage of the SR is presented in Table 1.2, along with the number of entries (references and citations) evaluated.

Table 1.2. The total number of new papers included in each search process stage.

Review stage	**Evaluated entries** (duplicates included)	**Identified papers** (new and unique)
Initial papers	—	4
Start set	440	* 3
Iteration #1: Backward	274	104
Iteration #1: Forward	222	60
Iteration #2: Backward	6 734	111
Iteration #2: Forward	21 393	29
Iteration #3: Backward	4 730	5
Iteration #3: Forward	4 437	8
Iteration #4: Backward	425	0
Iteration #4: Forward	173	0
Author search	3 769	0

* – while there is a total of four start set entries, one is also an initial paper

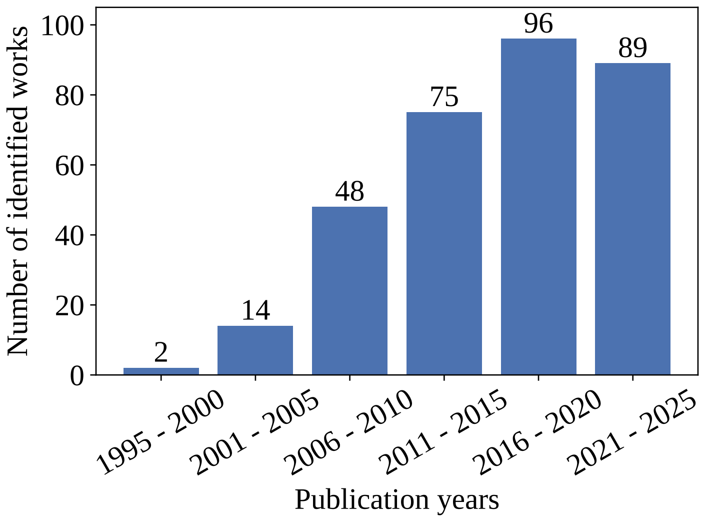


Fig. 1.3. Number of identified works in different publication years (own work).

The identified works were spread over multiple decades, with the first published in 1999 by Rothermel et al. [29] and the last during 2025. The methods used for the SR should ensure that it is unbiased in terms of publication years. The total number of articles from different years is shown in Figure 1.3. Since the earliest paper comes from 1999, the first bucket actually includes only the years 1999 and 2000 (with one publication each). Similarly, January 2025 constitutes the end date of the SR, and thus the last bucket includes only one month from that year. We can observe that TCP is a growing research field with an increasing number of works over the years.

As mentioned in the protocol, each entry was assigned to one or more of the following paper kinds: `SRRQ1` – articles presenting new TCP datasets; `SRRQ2` – articles describing new TCP algorithms; `SR` – systematic reviews; `CMP` – publications comparing available TCP solutions. Since the sets of these types overlap, in Figure 1.4, a Venn diagram was used to present the counts of entries with given paper kind combinations. A total of 54 articles introduced new datasets, 267 articles presented novel algorithms, 32 articles were secondary studies, and 28 articles were dedicated to empirically comparing existing solutions. The combination of `SRRQ1` and `SRRQ2` kinds was the most common, with 46 papers describing both algorithms and datasets. Some articles combined `CMP` with either `SRRQ1` or `SRRQ2`, and only two works focused on datasets alone. Notably, the `SR` set did not overlap with other paper kinds.

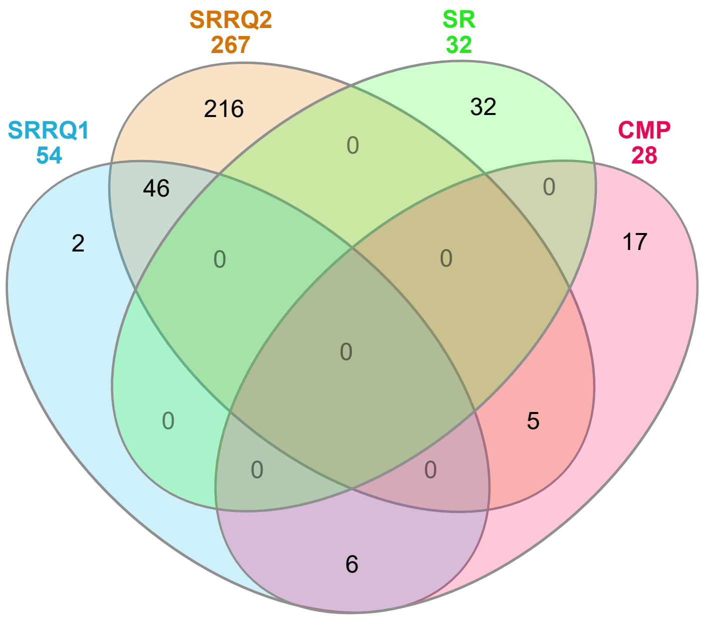


Fig. 1.4. Venn diagram of identified paper kind sets (own work).

During the survey, a total of 32 previous reviews were identified. Although these studies were not particularly useful for answering the SRRQs, they greatly boosted the snowballing process, since they contained a lot of references. Some of the identified surveys focus on a narrow part of the research area, for instance, covering only genetic algorithms, or the application of TCP within software product lines. On the other hand, some of the reviews report on a wider area, additionally including test case selection and test suite minimization. Numerous articles had a scope similar to the proposed review, but did not disclose the number of primary studies analyzed. Since the discussed works are not suitable for a comparison with the proposed SR, of the 32 identified articles, only eight were handpicked and chronologically presented in Table 1.3. In order to provide a fairer assessment, since the proposed search included other SRs, only the number of primary studies identified was noted. Among the newest secondary studies, the works of Prado Lima and Vergilio [7] and Mukherjee and Patnaik [26] described other reviews they came across. All of them were also found during the snowballing process, further indicating the effectiveness of the proposed SR.

Table 1.3. Overview of other surveys encountered during the systematic review.

Year	Study reference	Years covered	Identified works
2012	Singh et al. [30]	1997 – 2011	65
2013	Catal and Mishra [31]	2001 – 2011	120
2017	de S. Campos Junior et al. [32]	1999 – 2016	90
2018	Khatibsyarbini et al. [5]	1999 – 2016	80
2020	de Castro-Cabrera et al. [33]	2017 – 2019	320
2020	Prado Lima and Vergilio [7]	2009 – 2019	35
2021	Mukherjee and Patnaik [26]	2001 – 2018	90
2023	Abubakar et al. [34]	2002 – 2021	250
2025	*proposed*	1999 – 2025	292

In general, the thoroughness of the proposed review is difficult to compare with other studies, as they contained diverse research questions and years covered, and it is possible that some other reviews were missed. However, it can be said that the conducted study is among the largest SRs of TCP in terms of the number of identified publications.

There were a number of papers that had to be excluded due to various reasons (mainly because they did not conform to any of the inclusion criteria), but which are particularly interesting in the wider context of conducted research. These were noted during the review and, among others, include:

- Memon et al. [12] and Czerwonka et al. [13], which describe the practical usage of TCP in large software development entities (Google and Microsoft, respectively).
- Do and Rothermel [35] and Luo et al. [36], which investigate the usage of synthetic defects, called mutants, for the evaluation of TCP techniques.
- Mendoza et al. [37], which applies data balancing techniques on datasets to improve the effectiveness of machine learning-based TCP solutions.

SRRQ1: *What are the available datasets and subject programs for TCP?*

The accumulated data was used to answer SRRQ1. Since TCP can be applied to almost any software product, where testing (not even necessarily autonomous) is used, nearly any open-source software repository can be adapted to act as a dataset. Some solutions followed this approach and adapted the test suites of publicly accessible projects, with the most common choice being Software-artifact Infrastructure Repository projects[1] (SIR): Unix utilities (i.e., bash, grep, sed), Siemens programs, JMeter, JTopas, Ant [7, 30–32]. Other identified subject programs include compilers (GCC, javac, Jikes), JDepend, Gradebook, Tomcat, Camel, and Drupal.

Some solutions instead opt for processed test suite datasets, such as the Google Shared Dataset of Test Suite Results[2] (GSDTSR), Defects4j[3], and Paint Control[4] [7, 33]. The RTPTorrent dataset is the only identified data source supported by a dedicated research paper. It is based on test results from more than 9 years of build runs within 20 open-source Java programs and provides reproducible baselines for other researchers [38].

In less complicated problem statements, TCP can be applied to simple *fault matrices*, which state whether a given test case can detect a given fault, for every case in the suite, and are presented as a matrix. An example of such a dataset is the BigFaultMatrix[5]. On the other hand, complex TCP problem statements might, for example, require system models as inputs. In these cases, the datasets have to be enriched with additional data.

Finally, synthetically introduced faults (mutants) might be a viable evaluation method for TCP algorithms. Do and Rothermel [35] conclude that mutants can be used as a low-cost dataset construction method. However, Luo et al. [36] highlight that TCP algorithms that perform the best on sets of mutants might perform worse on real-world data.

In general, the answer to SRRQ1 is that there are many TCP evaluation methods available and none of them are universally used. The choice of the dataset should be influenced by the exact problem statement and other factors.

SRRQ2: *What are the state-of-the-art TCP algorithms?*

Subsequently, the results of the study were used to answer SRRQ2. Here, since TCP has many subdomains, each with its own limitations with regard to data availability, time constraints, and more, it is difficult to directly compare two TCP algorithms. The problem is worsened by the fact that many articles only compare their solutions with a baseline, disregarding previous state-of-the-art approaches.

Singh et al. [30] noted that more empirical comparisons of existing techniques should be performed. However, this is difficult, since to directly compare two methods, they must work on the same definitions, metrics, and datasets. They also remark that some

[1] `https://sir.csc.ncsu.edu`

[2] `https://code.google.com/archive/p/google-shared-dataset-of-test-suite-results`

[3] `https://github.com/rjust/defects4j`

[4] `https://bitbucket.org/HelgeS/retecs/src/master/DATA`

[5] `https://github.com/dathpo/test-case-prioritisation-ga`

studies even fail to report the achieved performance. Catal and Mishra [31] offer similar recommendations. Only 13% of the studies identified by them compare the performance of different TCP approaches. The findings are confirmed by the currently proposed SR, with such studies comprising 9% of the publications identified. Furthermore, the authors state the need for more empirical evaluations of the proposed techniques, noting that a quarter of the identified TCP methods did not use any evaluation metrics to assess effectiveness. Although the cited works are not recent, their findings unfortunately still apply.

Multiple studies attempted to compare existing TCP methods. In 2018, Haghighatkhah et al. [6] analyzed diversity-based and history-based test prioritization applied to six subject programs. They found that both methods can be used effectively but can also be combined to achieve even better results. Luo et al. [39] instead compared static and dynamic test case prioritization techniques. They discovered that different techniques perform best depending on subject program choice and test granularity (whether entire test classes or individual test methods are prioritized). Marijan [9] compared ML approaches with heuristic-based techniques, finding that LambdaRank with neural network and the ROCKET [40] algorithm are the most effective among the tested. The ROCKET technique is also the second most widely used baseline in the evaluation of TCP, after random test case ordering, according to Prado Lima and Vergilio [7].

In summary, unfortunately, many existing TCP studies do not empirically evaluate the solutions against other approaches. This is partially caused by the fact that input data requirements, method limitations, and metric and dataset choices must all be matched to fairly compare two different TCP methods. As a baseline, ROCKET and random test prioritization may be used, despite not necessarily being the best performing state-of-the-art solutions. Proposed TCP solutions should also be evaluated against any comparable and well-performing techniques.

1.2. Previous works in test case prioritization

Within regression testing, the main techniques to speed up the execution of the test suite include test suite minimization (TSM), test case selection (TCS), and test case prioritization (TCP) [3, 10], described in the introduction.

The test case prioritization problem was previously formally defined in the work of Rothermel et al. [28], and the definition can be seen in Equation (1.1):

$$\begin{aligned} \textbf{Given: } & T\text{, a test suite; } PT\text{, the set of permutations of } T\text{;} \\ & f\text{, a function from } PT \text{ to the real numbers.} \\ \textbf{Problem: } & \text{Find } T' \in PT \text{ such that } (\forall T'')\,(T'' \in PT) \\ & (T'' \neq T')\,[f\,(T') \geq f\,(T'')]\,. \end{aligned} \tag{1.1}$$

In the problem statement, f is used to evaluate the quality of the ordering (higher values are better). In simpler words, our aim is to find the best permutation of test cases in terms of a given metric. Viable choices for such scoring functions are outlined near the end of this chapter.

1.2.1. Foundational works and TCP approach taxonomy

To the best knowledge of the author, the first mention of TCP occurs in the work of Wong et al. [41], where it is used as an additional step after test case selection. The first comprehensive look at TCP was then provided by Rothermel et al. [29], where a wide range of prioritization methods were investigated.

The following basic prioritization approaches were listed by Rothermel et al. [29]: *no prioritization* – an approach in which test cases are run as if no TCP was applied; *random prioritization* – random ordering of test cases in the suite; *optimal prioritization* – the best theoretical ordering of cases, which is highly impractical, as it first requires a full prior execution of the suite. They then define four techniques based on code coverage and two additional approaches based on fault-exposing-potential (FEP). The work of Rothermel et al. [29] also introduced a metric, indicating the quality of a given permutation in the form of *average of the percentage of faults detected* (APFD) [42].

Over the years multiple TCP approach categorizations have been proposed. Examples include the taxonomies used by Khatibsyarbini et al. [5], Singh et al. [30], and Mukherjee and Patnaik [26]. For the purpose of the study, the classification introduced by Yoo and Harman [10] and later used by Catal and Mishra [31] and Prado Lima and Vergilio [7] was selected. It includes the following categories of TCP approaches [10]: coverage-based, distribution-based, human-based, probabilistic, history-based, requirements-based, model-based, cost-aware and other approaches. Compared with the other categorizations, using insights collected during the SR, it can be observed that machine learning solutions are underrepresented in the list of approaches. Pan et al. [43] provided a separate taxonomy for just ML-based approaches, with the following categories: supervised learning, unsupervised learning, reinforcement learning, and natural language processing. It should be noted that the mentioned sets are not disjoint, and many hybrid approaches were introduced. Both described classifications are presented side by side in Figure 1.5.

In addition to the taxonomies in Figure 1.5, the TCP approaches are characterized by their properties. Most existing TCP methods can be classified as either static or dynamic [39]. *Static* techniques operate only on the source and test code, or sometimes even without them, using only static program analysis. On the other hand, *dynamic* techniques utilize rich run-time semantics in addition to the statically available information. This enables the algorithm to reorder the cases as the tests are executed. The techniques

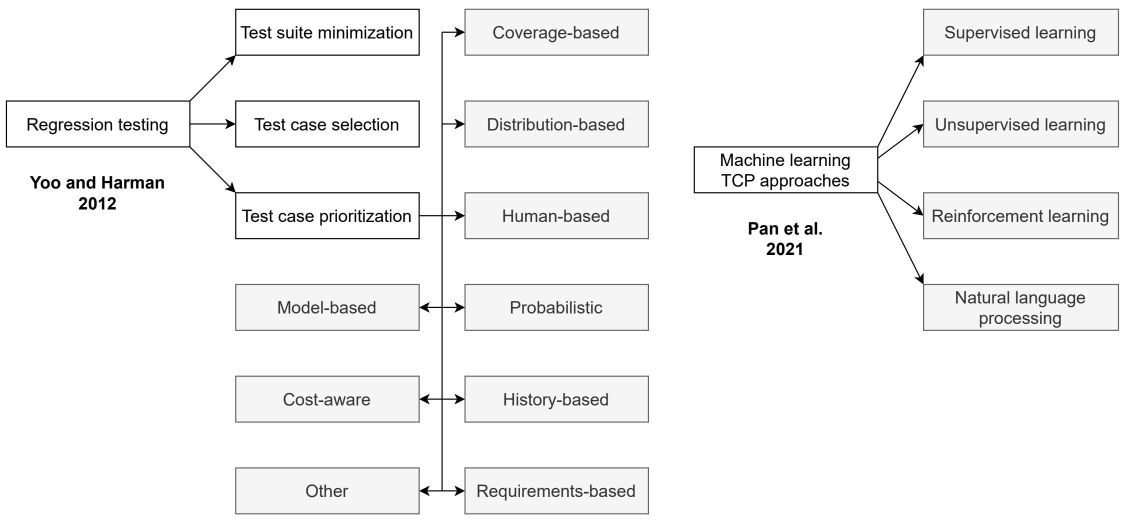


Fig. 1.5. Taxonomies of TCP [10] and ML-based TCP [43] approaches (own work).

can also be divided into black-box and white-box types [44]. *Black-box* approaches do not require access to the source code of the system under test, in contrast, *white-box* approaches can use the source code information to improve the prioritization performance. The listed characteristics of the prioritization methods have previously been compared by Henard et al. [44] (black-box and white-box) and Luo et al. [39] (static and dynamic).

1.2.2. Coverage-based prioritization

Coverage-based prioritization is among the most popular TCP methods [5, 26, 30]. It uses code coverage metrics of test cases to rank them [10]. Intuitively, rapid early widening of structural coverage facilitates earlier fault detection. Coverage-based approaches can generally be divided into total and additional strategies [45]. *Total* strategy maximizes the count of sections covered by the test (regardless of whether the section was already visited before), while the *additional* strategy instead focuses on the number of sections, which have not yet been covered. The strategies are also characterized by the granularity of the structural coverage: line, statement, branch, method, and class.

Coverage-based approaches are also one of the oldest, being used as early as 1999 by Rothermel et al. [29]. They introduced four coverage-based techniques, namely *total branch*, *additional branch*, *total statement* and *additional statement* coverage strategies. It is worth mentioning that additional strategies may achieve full structural coverage before all cases are ranked. An alternative approach should be used in such circumstances, with Rothermel et al. [29] falling back to the total strategy. They concluded that all coverage methods perform better than random prioritization, with the total strategy achieving higher results than the additional variant. Elbaum et al. [46] investigated coverage-based prioritization further, checking how test case granularity affected fault detection of the approaches. They statistically verified that coarser granularity (e.g., function level) decreases performance.

Later, Zhang et al. [45] presented a way to unify the total and additional strategies into a single approach. They introduced a parameter called p. When $p = 0$, the proposed approach mirrors the total strategy, and when $p = 1$, it is equivalent to the additional strategy. Other values of p blend the approaches, producing a strategy combining total and additional coverage. They found that differentiated p values can perform better than the basic techniques. Di Nardo et al. [47] instead evaluated TSM, TCS, and TCP in a real industrial system. In their prioritization experiments, techniques based on additional structural coverage applied to finer element granularity perform best.

1.2.3. History-based prioritization

Kim and Porter [48] were among the first to use historical data for TCP [10, 30]. *History-based* prioritization takes data from past test executions, such as previous failures, and change information to order cases [33]. Intuitively, if a test case commonly failed in the past, it is also highly probable that it will fail during the current test cycle. Kim and Porter [48] found out that information about previous test cycles could be used to reduce costs and increase the efficacy of regression tests. They also demonstrated the necessity of different approaches to TCP in constrained and nonconstrained environments.

Further research on history-based TCP was conducted by Park et al. [49], who introduced the historical value-based approach [30]. They gathered past fault severities and testing costs and used them to evaluate new test cases. Their experiments verified the applicability of the proposed approach, which can also be fused with other TCP methods.

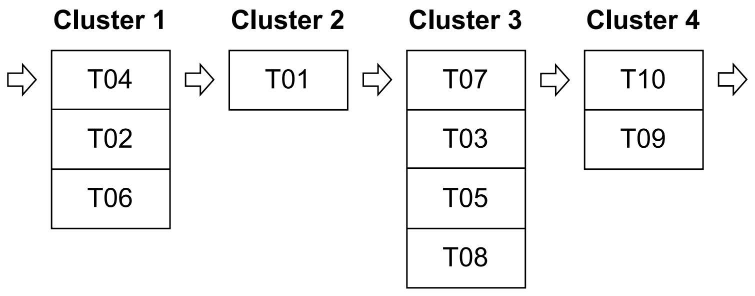
History-based prioritization can be combined with other TCP paradigms to create various hybrid methods. For example, the ROCKET approach from Marijan et al. [40] incorporated specialized heuristics along with historical data to increase the fault detection rate. Methods based on reinforcement learning, such as the work of Bagherzadeh et al. [8] can also be seen as a form of history-based TCP, due to the extensive usage of the results of previous test executions.

1.2.4. Distribution-based prioritization

Distribution-based prioritization aims to order test cases using the distribution of their characteristics in multidimensional spaces [10]. Intuitively, test cases that are very similar should reveal similar faults. In the extreme, if two test cases are fully equivalent, one of them is redundant and can be run last in the suite (or omitted in case of TCS and TSM). Thus, we should try to increase the diversity of cases ran early, to cover as many eventual faults as possible. Various metrics are used to approximate the dissimilarity between two test case profiles [10], explaining why adjacent techniques are also referred to as *similarity-* or *dissimilarity-based*. The survey from de Castro-Cabrera et al. [33] noted a high increase in similarity-based prioritization research in the second half of the 2010s.

The distribution of profiles was used for the first time by Leon and Podgurski [50] to filter and prioritize test cases [10]. They produced hybrid techniques that introduced distribution information into coverage-based prioritization. The methods first ordered test cases until full total structural coverage was achieved, and then selected cases with divergent characteristics. Ledru et al. [51] instead compared test case inputs using string distances. They first selected the input farthest away from all others and then greedily built the entire list, choosing the input that was most distant from already prioritized cases. Different distance metrics were evaluated.

A common technique is to use clustering algorithms to construct sets of similar test cases [50, 52–54]. Then, cases from different clusters should be picked to maximize profile diversity. For TCP usage, two questions arise: how should the clusters be ordered, and how should the cases within each cluster be sequenced? Various data can be used for clustering, including execution history, code coverage, test inputs, and even test code. An example of test prioritization using clusters is shown in Figure 1.6.



Prioritized test cases:

T04, T01, T07, T10, T02, T03, T09, T06, T05, T08

Fig. 1.6. Example of TCP using clustering (own work, based on [52]).

Carlson et al. [52] evaluated clustering algorithms using code coverage, complexity, and fault data in a large industrial project, namely Microsoft Dynamics Ax. They concluded that clustering might improve the results achieved by TCP. Arafeen and Do [53] instead focused on grouping test cases together using related requirements. They first clustered software requirements using textual similarity, then mapped each requirement to its associated test cases, producing test clusters, and finally established the orders of clusters and cases within clusters. Their results suggest that incorporating requirement data into similarity-based TCP can provide benefits. Finally, Zhou et al. [54] recently used code representation for clustering. They grouped the generated tests according to their GraphCodeBERT embedding and then sorted the groups according to their fault-revealing scores.

Although vastly different from the original distribution-based method presented by Leon and Podgurski [50], information retrieval-based TCP can also use various similarity metrics. Yang et al. [22] demonstrated a method that uses code representation similarity to associate test cases with tested methods. Previous information retrieval-aided TCP techniques instead utilized textual similarity.

1.2.5. Requirement-based and model-based prioritization

Produced code artifacts are often linked to different natural language documents and diagrams, such as requirement specifications and system models. The goal of the two types of prioritization approaches described below is to use different forms of these documents to increase TCP performance.

Requirement-based prioritization uses requirement information (such as their priority, complexity, and text content) to order test cases [7, 31]. It was first proposed by Srikanth et al. [55], who mapped requirements to test cases and used requirement properties for prioritization [10, 30]. Their approach, called PORT, used a total of four input variables: complexity, volatility, priority, and fault proneness. PORT was evaluated using graduate student projects and was shown to improve fault detection, with the priority factor contributing the most [55]. Arafeen and Do [53] combined clustering with requirement-based prioritization, as described in Section 1.2.4. Later, in 2016, Srikanth et al. [56] expanded the PORT approach, evaluating it in an enterprise application. They focused on priority and fault proneness and found that these indicators can be used to improve TCP processes.

Model-based prioritization instead uses system models, such as UML (Unified Modeling Language) sequence and activity diagrams [7, 31]. This category of prioritization approaches was first introduced by Korel et al. [57], and initially simply performed classification on cases dividing them into two groups according to priority and then randomly shuffling each group [10]. The priority of a given test case was based on whether it was relevant to the changes in the model elements. Korel and Koutsogiannakis [58] experimentally compared model-based prioritization with code-aware methods. To achieve this, they evaluated two heuristic TCP methods on common datasets. Their results indicate that model-based ordering can perform better than a code-based approach. Unfortunately, the study focused on only two heuristics, limiting its ability to generalize to all approaches with these categories. One of the more recent advances in model-based prioritization is the work of Huang et al. [59], who suggested a method incorporating a random forest on top of existing heuristical approaches to construct a robust learn-to-rank model. They compared their approach with random prioritization and seven other approaches, showing that their approach can be applied to TCP.

1.2.6. Search-based and nature-inspired prioritization

Search-based prioritization algorithms of various types have been widely used to order test cases [5, 30, 51, 60–62]. Li et al. [60] noticed that prior to their study, most TCP approaches used greedy algorithms, which could produce suboptimal results. They evaluated various greedy, heuristical, and search-based algorithms on six datasets. Among others, hill climbing and genetic algorithms were tested. Their results suggest that more robust search techniques outperform greedy search. In 2010, Li et al. [61] experimented with five search-based TCP methods. In contrast to the previous study, they used simulated data instead of real-world projects. The results and conclusions of the two described studies overlap most of the time, with a notable difference in the performance of the genetic algorithms [30, 60, 61]. Di Nucci et al. [62] proposed the use of the hypervolume metric to improve search-based TCP. Their genetic algorithm achieves a shorter execution time and greater cost-effectiveness.

Many prioritization solutions use nature-inspired techniques. There are more than 20 primary studies on the use of genetic algorithms alone [11]. They vary by algorithm type, encoding scheme, population and generation sizes, as well as used genetic operators. Researchers also frequently employ swarm-based algorithms. The systematic review performed identified the following approaches in the literature, ordered alphabetically: ant colony, bat, cat swarm, cuckoo, dwarf mongoose, firefly, fish school, honey bee, particle swarm, rat swarm, and whale.

Singh et al. [63] proposed an ant colony optimization (ACO) solution for TCP. They presented a theoretical technique, and its implementation, for test case ordering in time-sensitive environments. In terms of the average percentage of faults detected, the ACO approach performed the best of the compared algorithms. Gao et al. [64] used ACO on three input variables: detected fault count, fault severity, and test duration. Their method produced results close to the optimal order and verified the applicability of ACO to test case prioritization. Öztürk [65] evaluated a bat-inspired algorithm, called BITCP. They mapped the test duration to the distance-from-pray parameter and the fault count to the loudness parameter. They tested the solution in five .NET projects, achieving the highest performance metric value. Hla et al. [66] applied particle swarm optimization to TCP for embedded systems. The proposed algorithm prioritized tests executed on the basis of modified software components. They found that particle swarm optimization can produce high-quality test suite permutations quickly.

As mentioned above, there are numerous other research studies on nature-inspired TCP solutions, with several dozens identified in the systematic review. Since the thesis is not focused on this prioritization paradigm, they will not be described further unless necessary. The results of the literature survey also suggest the need for an in-depth study and classification of nature-inspired TCP approaches. However, this is outside of the scope of this thesis.

1.2.7. Reinforcement learning for TCP

Reinforcement learning (RL) is a machine learning technique, in which the system learns an effective task solution based on feedback [3]. In the context of TCP, the problem is represented as a sequence of interactions between the continuous integration (CI) environment and an RL agent [8]. This technique of reordering test suites is rather new, with 10 works identified, all between 2017 and 2024. In addition, only one identified article was published before 2020 [67]. Figure 1.7 presents the general mechanism of reinforcement learning-based TCP.

Spieker et al. [67] was the first to propose RL for test case prioritization [68]. They presented the RETECS method, which is CI-oriented and aims to reduce the time between code contribution and testing feedback. The performance of RETECS is comparable to that of simple deterministic approaches, such as classification based on recent outcomes and a weighted sum of features. Bertolino et al. [68] introduced a partition of machine learning TCP methods. In contrast to the taxonomy introduced by Pan et al. [43], only two categories were introduced. *Learning-to-rank* (LTR) approaches use supervised learning to predict test case ranks based on the features of the test suite. On the other hand, *ranking-to-learn* (RTL) denotes the use of RL algorithms for TCP [68]. A comparative analysis proved that the RL algorithms perform worse than the state-of-the-art supervised models [8]. Later, Bagherzadeh et al. [8] was able to improve the RTL methods, providing a significant increase in performance over previous works. They concluded that RL can be effectively used for TCP.

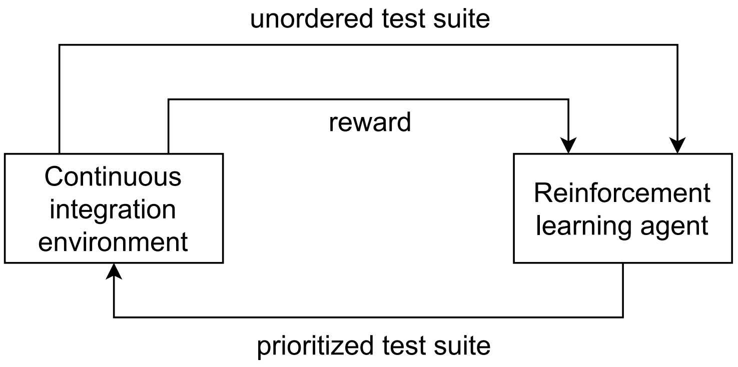

unordered test suite
reward
Continuous integration environment
Reinforcement learning agent
prioritized test suite

Fig. 1.7. Overview of RL-based TCP in CI environments (own work, based on [67]).

1.2.8. Additional TCP constraints

When the theoretical test case ordering problem gets applied in the industry, limitations may apply. For example, time and resource constraints decrease the applicability of many regression testing optimization methods [48]. If the prioritization of the suite uses a computationally expensive model, which takes much more time to predict than the actual execution time of the suite, prioritization might be undesirable. Kim and Porter [48] noticed this issue and experimentally proved that different regression testing approaches should be used in constrained and nonconstrained environments.

Another difficulty of TCP is its applicability in highly configurable software. Some systems contain hundreds of parameters, some of which might be mutually exclusive, and testing all possible combinations of them might be infeasible [69]. Some prioritization approaches focus on such systems with the aim of improving the time effectiveness of regression testing. Marijan et al. [69] introduced the TITAN test suite optimizer and evaluated it during an industrial collaboration with Cisco Systems. TITAN achieved better results than prior industry practice, both for the prioritization and selection of test cases.

1.2.9. Unification of TCP

The undertaken systematic review shows a large fragmentation in the field of TCP, which is supported by the findings of other researchers. Singh et al. [30] highlighted that no standards emerged in terms of tools, datasets, and evaluation techniques used, leading to the incomparability of the performance of different approaches. Catal and Mishra [31] suggested the need for a wider use of public datasets and stricter evaluation practices. Prado Lima and Vergilio [7] note that only a small percentage of studies use statistical tests to validate the performance of the solution.

Zhang et al. [45] proposed one of the first attempts to unify the fragmented TCP ecosystem. They create a model which merges previously separate total and additional coverage-based prioritization methods. Mattis et al. [38] contributed the RTPTorrent dataset collection, which is the only publication focused solely on producing high-quality open TCP datasets, identified in the systematic review. Their solution addresses threats to validity caused by existing TCP datasets with the following choices: high variety of selected projects (in terms of maturity, system size and contributor count), openness and reproducibility, as well as the inclusion of nontrivial comparison baselines.

Vescan et al. [25] noticed that there are many TCP methods available, but a framework (neither mental nor technical) was not developed that unifies all of them. They attempted to address this gap by developing a view of TCP based on three pillars. The first characteristic, called *traceability* describes the relationships between requirements, source code, and test cases. The second, referred to as *contexts* outlines the type of tests performed (unit, integration, end-to-end, etc.), granularity, and test case design. Finally, the *information* pillar lists available factors. It consists of the following input data: test cases, faults, execution times, requirements (optionally with change information), source code (optionally with change information), and resource constraints.

1.2.10. Test case prioritization evaluation metrics

Rothermel et al. [29] introduced the *average of the percentage of faults detected* (APFD) metric in 1999 [42, 70]. Although it exhibits multiple limitations, it is still the most widely used TCP metric in the literature [5, 26, 30–32, 34]. APFD measures how quickly an ordered test case list detects faults. The metric ranges from 0 to 1 or 0 to 100 in percentages, with higher values of the metric representing better orderings [26, 31, 42, 46, 70]. It can be calculated using the formula from Equation (1.2) [42, 70, 71]:

$$\text{APFD}(s) = 1 - \frac{\sum_{j=1}^{m} \text{TF}_j}{nm} + \frac{1}{2n}, \tag{1.2}$$

where s is the suite, n is the number of its test cases, m is the number of identified faults, and TF_j is the rank of the first test in the ordered suite, which reveals the j-th fault. Although different symbols are used in the literature, all the definitions found were equivalent. APFD can only be calculated for CI cycles that fail [70]. Since fault information is rarely tracked in real-world use, each test failure is often treated as a separate fault [38].

The *cost-cognizant weighted average percentage of faults detected* (APFD_C) introduced by Elbaum et al. [72] extends APFD to incorporate differences in fault severities and test case costs (for example, their execution times). It is also referred to as *APFD with cost consideration* [5, 7], *units-of-fault-severity-detected-per-unit-of-test-cost* [26, 72] or *APFD per cost* [26] and abbreviated as APFD_{CW} [71] or APFD(c) [31]. The APFD_C metric is very popular and is sometimes considered the second most widely used TCP metric [30, 32]. It can be seen as an extension of APFD, where test case costs and fault severities are known, and can be calculated as shown in Equation (1.3) [42, 71]:

$$\text{APFD}_C\,(s) = \frac{\sum_{j=1}^{m} \left(f_j \cdot \left(\sum_{i=\text{TF}_j}^{n} t_i - \frac{1}{2} t_{\text{TF}j} \right) \right)}{\sum_{j=1}^{n} t_j \cdot \sum_{j=1}^{m} f_j}, \tag{1.3}$$

where s, n, m, TF_j have the same meaning as in Equation (1.2) and f_i and t_j describe the severity of i-th fault and execution cost of j-th test case, respectively. It has the same value range as APFD [71].

In real-world studies, fault severities are rarely known. The APFD_C formula works also when all severities are assumed to be equal [42, 71, 72]. Equation (1.3) can then be simplified down to the form shown in Equation (1.4) [42, 71]:

$$\text{APFD}_C\,(s) = \frac{\sum_{j=1}^{m} \left(\sum_{i=\text{TF}_j}^{n} t_i - \frac{1}{2} t_{\text{TF}_j} \right)}{m \cdot \sum_{j=1}^{n} t_j}, \quad \text{when } \forall_{1 \le j \le m},\ f_j = 1. \tag{1.4}$$

The formula from Equation (1.4) is sometimes referred to as *average percentage of faults detected, time-aware* (APFD_{TA}) [71] or *simplified APFD_C* [42]. When all costs are equal and all severities are equal, APFD_C becomes APFD [71, 72].

If the severities of the faults are known and the approach is requirement-based, *average severity of faults detected* (ASFD) and the auxiliary *total severity of faults detected* (TSFD) metrics could be used instead [26, 31].

Whenever we introduce constraints to TCP, some faults might not get detected, as some test cases are not run. Qu et al. [73] introduced *normalized APFD* (NAPFD) in 2007, including an additional p factor that represents the fraction of identified faults in all identifiable faults [34, 42, 70, 71]. This metric is not useful for unconstrained TCP, since it is equal to APFD when $p = 1$ [71]. Similarly to other metrics in the APFD family, NAPFD can only be calculated for failed cycles [70]. The formula for NAPFD is given in Equation (1.5), where TF_j is set to zero if a fault is never detected [42, 71]:

$$\text{NAPFD}\left(s\right) = p - \frac{\sum_{j=1}^{m} \text{TF}_j}{nm} + \frac{p}{2n}. \tag{1.5}$$

Zhao et al. [70] identified weaknesses in existing metrics from the APFD family, namely that APFD has different value ranges depending on fault and test case counts. This also means that although it returns results bounded by 0 and 1, the optimal solution does not have an APFD of 1, and neither does the worst solution have an APFD of 0. They addressed this weakness, min-max mapping the measure from the $[\text{APFD}_{\text{min}}\left(s\right),\ \text{APFD}_{\text{max}}\left(s\right)]$ interval to $[0,\ 1]$, introducing the *rectified APFD* (rAPFD) metric, and ensuring that the optimal solution has $r\text{APFD} = 1$. The value of rAPFD for a given test suite is shown in Equation (1.6):

$$r\text{APFD}\left(s\right) = \frac{\text{APFD}\left(s\right) - \text{APFD}_{\text{min}}\left(s\right)}{\text{APFD}_{\text{max}}\left(s\right) - \text{APFD}_{\text{min}}\left(s\right)}. \tag{1.6}$$

The rAPFD metric should not be confused with the *relative average percent of faults detected* (RAPFD) series of metrics introduced by Wang et al. [71]. It should also be noted that NAPFD and APFD_C are subject to the same weaknesses as APFD. Despite this, we did not find rectified versions for NAPFD or APFD_C, neither in the systematic review nor a separate ad hoc forward snowballing on the work of Zhao et al. [70].

Bertolino et al. [68] introduced the *rank percentile average* (RPA) and its normalization called *normalized-rank-percentile-average* (NRPA) to evaluate the order of the test suite independently of the algorithm and the choice of criteria [8, 70]. They indicate how similar a proposed ranking is to the optimal one. Unlike APFD series metrics, NRPA can be calculated for nonfailing cycles. However, these measures have their own limitations and have been criticized before [8, 70]. Bagherzadeh et al. [8] note that NRPA treats successful and failed test cases equally, while failures are generally more important. The metric should also not be used for cycles with only a few cases, since they inflate the results (in the worst case, suites with just one test case always return $\text{NRPA} = 1$). Zhao et al. [70] support the limitations described previously and additionally report that sometimes worse sequences report higher NRPA values.

Many metrics for coverage-based TCP exist. For example, the AP*X*C family measures the average percentage of some structural coverage [42]. They are calculated similarly to APFD and can theoretically be calculated without suite execution. Metrics from this series include *average percentage block coverage* (APBC), *average percentage decision coverage* (APDC), *average percentage statement coverage* (APSC) [42]. Alternatively, *coverage effectiveness* (CE) is used to combine the costs and coverages of test cases, producing a value between 0 and 1. The metric is normalized by dividing the coverage area of the current suite by the coverage area of the optimal solution [5, 26, 31]. Finally, *expense* is a measure stating the percentage of structural coverage of the system required to discover a given fault [7].

The metrics previously listed are used to assess the effectiveness of TCP approaches. However, TCP applicability is also evaluated in the literature. If a method has high time or resource cost, it might be really effective but not applicable in practical contexts. Various time-related metrics are most commonly used for this goal [5, 7]. There is no consensus on the names and abbreviations of these measures in the literature; for the purpose of this thesis, they will be named as follows: *training time*, *build time*, *prioritization time* (or *prediction time* for machine learning-based solutions), *execution time*, *time to detect the first fault*, *cycle time* (sometimes called *end-to-end time*). The measures are presented in the context of CI in Figure 1.8. They can be totaled or averaged, and various ratios between them can be computed. The diagram assumes that the prioritization process does not require build information, and thus build and prioritization can run in parallel (and either of them can finish first). In such cases, if the TCP approach can predict the order faster than the system takes to compile and analyze, the ordering does not increase the cycle time. Training time is not included in Figure 1.8. It refers to the time it takes to train an algorithm on historical data and is performed offline. Thus, it is much less important than cycle time [70].

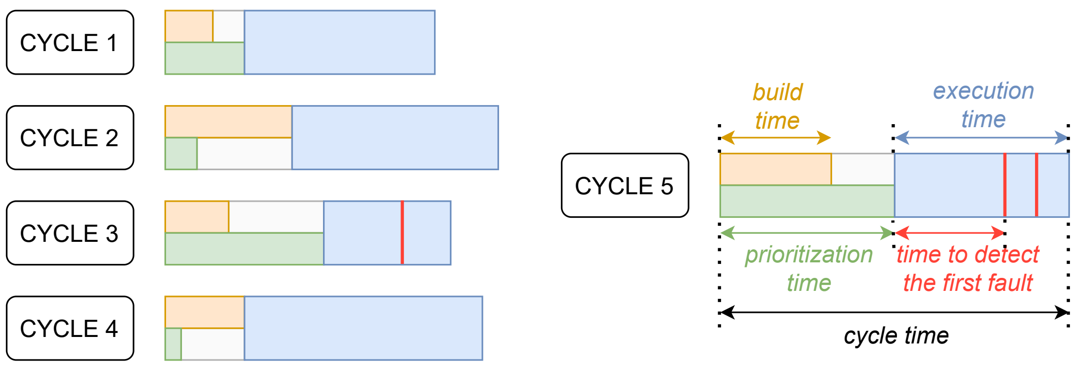


Fig. 1.8. Diagram of static TCP time-based evaluation measures in various cycles (own work).

Prado Lima and Vergilio [74] defined the *normalized time reduction* (NTR) metric to measure time savings gained if the cycle stopped at the first detected fault. It is only calculated for failing cycles and can be determined using Equation (1.7) [70, 74]:

$$\mathrm{NTR} = \frac{\sum_{i=1}^{\mathrm{CI}^{\mathrm{fail}}} (\hat{r}_i - r_i)}{\sum_{i=1}^{\mathrm{CI}^{\mathrm{fail}}} \hat{r}_i}, \tag{1.7}$$

where $\mathrm{CI}^{\mathrm{fail}}$ is the number of failed cycles, $\hat{r}_i$ is the execution time of i-th failing cycle, and r_i is the time to first fault of i-th failing cycle. The NTR measure was later adopted by Zhao et al. [70] to evaluate the applicability of TCP models.

All previously described measures, with the exception of severity-specific metrics (ASFD, TSFD) and indicators for coverage-based prioritization (APBC, APDC, APSC, CE, expense) are compared in Table 1.4.

Table 1.4. Comparison of described TCP effectiveness metrics.

Measure	Computable	Normalized *	Awareness			
			Failure	Cost	Severity	Constraint
APFD [29]	failed cycles	✗	✓	✗	✗	✗
APFD$_C$ [72]	failed cycles	✗	✓	✓	✓	✗
APFD$_C$ (simple)	failed cycles	✗	✓	✓	✗	✗
NAPFD [73]	failed cycles	✗	✓	✗	✗	✓
r**APFD** [70]	failed cycles	✓	✓	✗	✗	✗
RPA [68]	all cycles	✗	✗	✗	✗	✗
NRPA [68]	all cycles	✓	✗	✗	✗	✗
NTR [74]	entire history	✓	✓	✓	✗	✗

* – whether for a metric M, there occurs $M_{\max} = 1$

Various statistical tests have been used to validate significant differences between the performance of different TCP methods. According to Prado Lima and Vergilio [7], the most common tests applied to TCP are ANOVA and Wilcoxon Mann-Whitney. However, many primary studies lack such statistical verification.

1.2.11. Approach combinators

After an analysis of the titles and full abstracts of all publications identified during the undertaken systematic literature review, only two works adjacent to the methods proposed in the thesis were identified.

Huang et al. [59] attempted to combine the results of multiple heuristical TCP methods using ML algorithms. They focused on model-based prioritization and collected six simple estimators rating test cases based on state transitions. Examples of measures include the count of marked transitions and their frequencies. The outputs from particular heuristics are then accumulated using a random forest model to produce the final prediction. Their research differs meaningfully from the proposed approach, with the following differences: it focuses only on model-based TCP, instead of supporting all prioritization paradigms; it works only on top of certain methods, instead of treating sub-approaches as black boxes; finally, since it is a supervised machine learning algorithm, it requires prior training.

Hassan et al. [75] investigated the use of random sorting to resolve the ties in the ranking of the test cases. They concluded that this method does not yield optimal results and should only be used as a final resort. The approach proposed in this thesis supports random tie-breaking, but it also offers more sophisticated strategies.

An additional ad hoc search was performed, after the solutions from this thesis had already been implemented, in an attempt to find similar works in the literature. It was carried out in full using Scopus and used the *test case prioritization* term along strings such as `combinat*`, `aggregat*`, `ensemble`, `mix*`, `interpolat*`, and `tie*`. Two additional articles, described below, were identified.

The research of Cao et al. [76] from 2022 used a simple voting scheme to combine results from multiple models. Each model assigned votes in a decreasing sequence to the next cases from its output. The final score was calculated as the sum of the votes from all the models and used for sorting to determine the final result. Their research differs from the proposed model in the following ways: they focused only on user interface testing, instead of proposing a method for general usage in TCP; the combination used the simplest voting scheme available; moreover, no method to assign different weights to models was presented, while some sub-approaches might be more or less important.

The work of Mondal and Nasre [77], who introduced the Hansie framework for TCP, is most adjacent to the thesis. They introduced the concept of *priority-blind aggregation* or *consensus*, which assumes that the internal mechanisms of heuristics are unknown. Then, Borda count is used for result combination, similarly to one of the proposed algorithms. Despite resemblances, we believe that the approach shown by Mondal and Nasre [77], which as noted before was found after conceptualizing approach combinators, is not equivalent. First, the Borda count approach constitutes only a minor part of the research conducted in both publications. Secondly, Hansie uses parallel prioritization to resolve ties, which is not always feasible [78], while the proposal introduces different methods to deal with ties.

2. Proposed solution and its implementation

The proposed solution consists of both a mental framework and a concrete Python implementation of the described methods. The described algorithms are available for inspection and replication, using the instructions in Appendix B.

This chapter starts with the description of the framework upon which all subsequent approaches get built. Then, different non-combinator models are outlined, slowly ramping up in complexity. Finally, the approach combinator strategy is introduced, and concrete examples of models built using combinators are presented.

2.1. Research infrastructure (TCPFramework)

Although many works in the field of TCP publish associated software packages to aid reproducible research [39, 44, 68, 70, 77], most of them are ad hoc scripts used to collect results for research questions. We were unable to find a comprehensive and unified framework for the development and evaluation of TCP approaches in the existing literature.

To aid further research, both for the scope of this thesis and for other researchers, we propose an extensible platform for the development of test case prioritization methods, called TCPFramework.

The system is built as a Python module and allows for extension through subclassing and interface implementation. At the core of the framework, the `Approach` abstract base class is used as a template for all TCP methods to implement. For the simplest algorithms, the overriding of the `prioritize` method is sufficient. The method receives a set of test cases, enclosed in a convenient `RunContext` wrapper class. This type additionally ensures safety guarantees, such as:

- No test case can be marked as prioritized twice.
- All input test cases must be present in the output.
- Test cases not present in the cycle cannot be prioritized.
- The case information (execution time, verdict) is not extractable before execution.

For example, the optimal solution is, correctly, not implementable as a subclass of `Approach`, since it requires a posteriori cycle information to work. More advanced approaches may implement additional methods on the abstract class, such as: `get_dry_ordering`, used to compute an order without submitting it for execution;

on_static_feedback, used to dynamically adapt the internal algorithm state based on cycle results; or reset, to clear this internal state.

It should be noted that the implementators of the prioritize method should not return the test case order, instead it should be established through consecutive calls of the execute method of RunContext. This pattern provides certain desirable characteristics. Firstly, approaches that dynamically adjust the order based on the execution results of previous test cases are implementable. Secondly, a convenient way to express ties in the results is available. For example, an approach ordering the cases based on the total number of prior failures can treat two cases with the same number of failures as equal, instead of randomly preferring one of them.

The Dataset class is another form of abstraction offered by the TCPFramework. Its default implementation extracts the necessary information from the RTPTorrent dataset, which is described in further detail in Chapter 3. Other subject programs with compatible data formats can also be loaded and the Dataset class can be further derived to adapt other data sources. Input data can be cached between runs to speed up the evaluation process, and various dataset statistics can be calculated, such as cycle and failure counts or the number of invalid cycles.

Furthermore, TCPFramework provides the tools necessary for the calculation of evaluation metrics, in the form of the MetricCalc class. It contains methods which calculate all the metrics from the literature presented in Chapter 1, as well as the measures proposed in this thesis and described in Chapter 3, meaning: APFD, rAPFD, APFD$_C$, rAPFD$_C$, NAPFD, RPA, NRPA, NTR, and ATR. Performance indicators can be computed using standalone static methods and incrementally recalculated during dataset evaluation.

The entire solution is supported by data-centric classes representing different domain concerns, such as Cycle, TestCase, or TestResult. Inspection of the source code of a test case is supported. Finally, the standalone evaluate function loosely couples all extensible parts of the framework, namely, approaches, the dataset, and the metrics.

The structure of the platform is presented using Unified Modeling Language (UML) diagrams, rendered using PlantUML[1]. Figure 2.1 shows the UML class diagram of the solution. For clarity, the types in the method signatures are omitted, as well as private items, and the debug parameters are hidden. Figure 2.2 presents the UML package diagram of TCPFramework. The package structure is very simple, since most of the key classes reside directly in the tcp_framework module. Furthermore, the public interface of the Approach class described above is presented in Listing 2.1.

More information on the research infrastructure, including repository structure, as well as code execution and extension instructions, is shown in Appendix B. We believe that the usage of TCPFramework may be beneficial to future researchers.

[1] https://plantuml.com

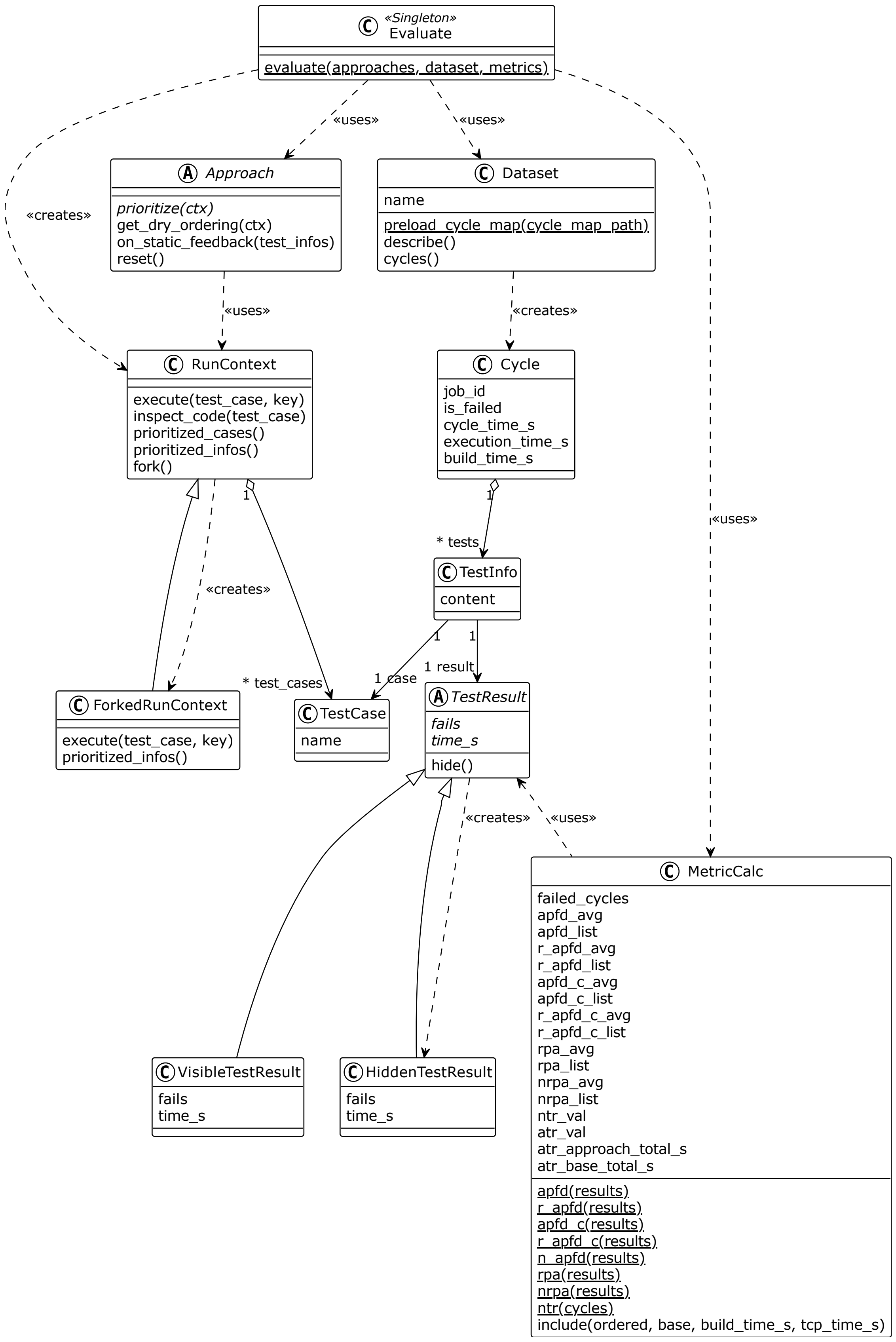


Fig. 2.1. The UML class diagram representation of TCPFramework (own work).

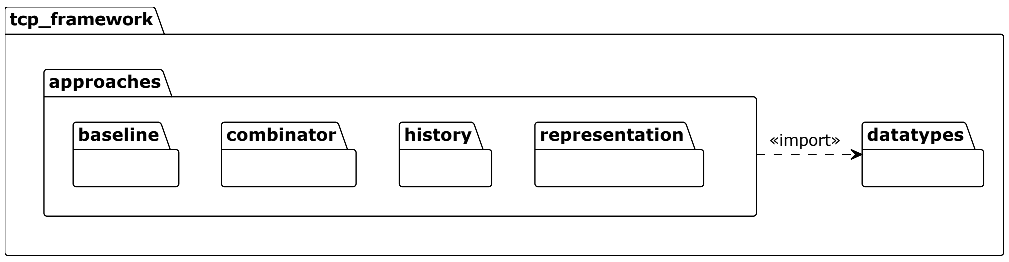


Fig. 2.2. The UML package diagram representation of TCPFramework (own work).

```python
class Approach(ABC):
    @abstractmethod
    def prioritize(self, ctx: RunContext) -> None: ...

    def get_dry_ordering(self, ctx: RunContext) -> Ordering: ...

    def on_static_feedback(self, test_infos: Sequence[TestInfo]) -> None: pass

    def reset(self) -> None: pass
```

Listing 2.1. Public interface of the `Approach` abstract base class.

2.2. Simple prioritization approaches

To confirm the validity of TCPFramework and to provide simple baselines for further evaluations, basic approaches based on foundational publications in TCP and straightforward intuitions were developed. These are as follows:

- *BaseOrder*: The base order approach executes the test cases from the suite in their original arrangement [29]. This method can be used as a no-prioritization-performed baseline to evaluate the gains of applying TCP to a certain software project. Since the original test case order is arbitrary, it might incidentally even be the optimal solution.
- *RandomOrder*: The random approach unmethodically shuffles the test suite and executes test cases in the resulting order [29]. Similarly to the base-order method, due to randomness, this technique can produce both great and terrible results, unless averaged over multiple random seeds. To aid in reproducibility, all TCP approaches that incorporate randomness in TCPFramework use constant random seeds.
- *TestLocOrder*: The test case lines of code (LOC) method ranks test cases in descending order based on the LOC in their implementation. Intuitively, longer test cases have more opportunities to detect faults (or even to be faulty). Optionally, only non-blank LOC are counted.

- *RecentnessOrder*: The recentness order rates test cases on the basis of how recently they were introduced to the project. A counter is kept for each case, accumulating the times it was seen in a cycle. The test cases that appeared fewer times in previous regression testing cycles are executed first. This reflects the intuition that verdicts of new test cases are less predictable (since less historical information can be used to estimate their outputs).

The remaining simple methods are mostly history-based and rate test cases using aggregations of their previous performances. The factors include execution time and failure count, and the aggregation ranges from simple sums to various moving averages. *Exponential smoothing* was previously used to aggregate historical results for use in test case prioritization [38, 48]. It is defined using the formula from Equation (2.1):

$$P(k) = \begin{cases} 0, & k = 0 \\ \alpha \cdot f(k) + (1 - \alpha) \cdot P(k-1), & k \geq 1 \end{cases} \tag{2.1}$$

where $f(k)$ is equal to the value of some property of the execution result for the test case in cycle k and α is a constant used to indicate the importance of recent results. Techniques using history-based aggregations are outlined below:

- *FoldFailsOrder*: The failure folding approach applies an aggregation function (folder) to failure counts of particular test cases from each subsequent cycle. The most notable folders include `total` (adding the current value to the already accumulated value) and `dfe` (which applies exponential smoothing). The test cases are then ordered either by random choice (where the accumulator value indicates the weight of choosing the given case) or by absolute value of the accumulator. *Demonstrated fault effectiveness* (DFE), previously used by Kim and Porter [48] is a special case of this approach.
- *ExeTimeOrder*: The execution time order applies exponential smoothing to subsequent execution times of a given test case achieved in prior cycles. This indicator is unaware of verdicts, which means that it does not discriminate between failed and passed tests. Regardless of this fact, it is useful when developing cost-aware prioritization methods.
- *FailDensityOrder*: The failure density approach combines two previous techniques by applying exponential smoothing separately to both the failure count and the execution time to then prioritize test cases according to their quotient. Since suites ordered by exact values of fail densities tend to achieve extremely high results in terms of multiple metrics, their approximation should ideally also provide sufficient outcomes.

In early experimentation, approaches from this family achieved significantly higher results than the random baseline. They can consequently be used as non-trivial reference points for newly developed methods, as done by Mattis et al. [38].

2.3. Code representation approach (CodeDistOrder)

The next developed model, *CodeDistOrder*, builds on the work of Yang et al. [22] and Zhou et al. [54], who used code representation to improve test case prioritization. Code representation aims to embed the semantic and syntactic information present in the source of a program as a numeric vector [22, 79]. This is often achieved through large neural networks. While the work of Yang et al. [22] addressed information retrieval-based test prioritization and Zhou et al. [54] used code representation to cluster test cases, the approach presented in this thesis instead focuses on the distances between code representations.

Since semantically similar pieces of code end up in the same areas of the multidimensional code representation space, the likeness of two test cases can be estimated as some distance measure between two vectors. These include the Manhattan distance, the Euclidean distance, and the cosine similarity [79].

Inspired by the string-distance-based approach shown by Ledru et al. [51], we present the following intuition: Test cases that are placed far apart in the code representation space are diverse and probably detect different code faults. Similarly, test cases close to each other should detect familiar faults. Consequently, if the suite contains multiple independent faults, it would be beneficial to explore dissimilar test cases next.

The goal of the proposed optimizer is to find the longest path, in terms of the source code representation distance, going through all test cases. Finding an exact solution to such a problem is computationally difficult [51] and would make the method unapplicable to real world problems. Instead, we employ a greedy algorithm that selects the highest possible distance at each step. The proposed method goes through the following steps:

1. The code representation is calculated for each test case using a selected vectorizer.
2. Representation distances are calculated between all unique pairs of cases.
3. The starting point is selected as the test case that has the highest aggregated distance (min, average, max) from all other in the suite. The prioritized set is initialized to contain only the starting point, while the remaining set includes all test cases except the starting point.
4. Until the remaining set is not empty: The test case with the highest aggregated distance (min, average, max) from the prioritized set is selected and executed. The test case is then removed from the remaining set and inserted into the prioritized set.

As an optimization, the distances between test case representations are computed lazily, only when necessary, and cached for further access. As alluded above, the approach can be tuned using numerous parameters. These include the vectorizer, the distance measure, and the distance aggregator. In terms of vectorization, TCPFramework supports the usage of any SentenceTransformers [80] model.

Other parameters are outlined in detail below:

- *Normalization*: The produced vectors contain information on both semantics and syntax of the source code [22]. We argue that the syntactic style does not affect the fault-detecting capabilities of cases, and its influence should be minimized with normalization. Tree-sitter[2] is used to tokenize the source code, and all unnecessary whitespace is omitted before passing to the vectorizer (the `formatting` mode). Alternatively, there is an option to extract only the identifiers (the `identifiers` mode), focusing even further on the semantic aspect of test case implementations.
- *Slicing*: In early experimentation, it was observed that limiting the vectorizer input to the first n characters of the test case code produced comparable results while significantly shortening the embedding duration. The slicing parameter is therefore the count of the leading n characters extracted from the source of the case.
- *Distance*: The cosine similarity, Euclidean distance, and Manhattan distance are available as distance measures. Note that while the higher values of cosine similarity indicate more similar vectors, the opposite occurs for Manhattan and Euclidean distances. The values of cosine distance are, therefore, subtracted from one for compatibility.
- *Aggregation*: Although the distance between two vectors is well defined, the distance between a vector and a set of vectors is ambiguous. In the literature, minimal, average, and maximal distances are used [51], which apply the mentioned aggregation functions to the distances from the target vector to each element of the set. All these functions are available in TCPFramework.
- *Failure adaptation*: We predict that once a fault is located, it is more beneficial to focus on test cases similar to it rather than continuing the diversity-focused search. The failure adaptation mechanism switches the optimizer from maximizing the distance to minimizing for the next few searches if the previous test case failed. Unfortunately, approaches using dynamic execution information have limited applicability. Their predictions cannot be performed in parallel with the build process and cannot be combined with all other methods.

The work of da Silva and Vilain [81] proposed an alternative test case similarity measure named LCCSS. It uses lengths of common token sequences in two source codes to find similar test cases. Due to opting against using large neural network models, LCCSS is much faster than the solution using SentenceTransformer.

An approach using the same pseudocode, yet employing LCCSS instead of vectorization and distance measurement, is also implemented in TCPFramework. Unfortunately, due to the underwhelming performance in the preliminary experiments, it was decided not to evaluate it in Chapter 3.

[2] `https://tree-sitter.github.io`

2.4. Approach combinators

The main focus of this thesis is the development of the mental model of approach combinators for test case prioritization. The existing literature often employs hybrid approaches that merge multiple factors. Only a few approaches mapped by Prado Lima and Vergilio [7] consider just one source of information. Despite this, the methods found during the systematic review are characterized by tight coupling. In response to calls to unify the field of TCP [25], we observe a need for a universal approach to the merging and enriching of existing TCP techniques.

To facilitate the combination of different TCP approaches, regardless of their complexity and implementation details, we believe that they should be treated as black boxes that take the set of test cases and produce its permutation using opaque mechanisms.

This guideline is respected by all the techniques in this section, collectively called *approach combinators*. They are a class of TCP approaches that take other approaches, use them to prioritize test suites without the knowledge of their internal algorithms, and then merge and enrich them using different methods. Three kinds of approach combinators are presented: *mixers*, which take multiple prioritization algorithms and mix their weighted results into the final sequence; *interpolators*, which shift the relative importance of different techniques as cycles pass; and *tiebreakers*, that aim to resolve ties produced by an approach, using a different approach or some heuristic. It should be noted that mixers are the most important primitives of the proposed model and that both interpolators and tiebreakers can be further implemented using them.

All developed strategies require no prior training, in contrast to many of the recently proposed ML-based approaches. This makes them applicable even in smaller projects, where there is insufficient historical information. Additionally, smaller development entities can use the methods as they are suitable for low-specification hardware.

2.4.1. Mixers

Mixers take multiple prioritization algorithms and mix their weighted results, returning the final test case sequence. The primary intuition is that various test case properties should be incorporated in an algorithm to produce high-quality predictions. For example, for cost-aware TCP, both test case verdicts and execution times should be taken into account. So, a combination of FoldFailsOrder and ExeTimeOrder from the previous section could be used. Moreover, failures are more crucial than execution time, so one prioritizer should be treated as more important than the other.

In summary, combinators from the mixer family take in multiple approaches and their corresponding weights (equal by default). As a result, a new approach is produced, which for each cycle prioritizes the test suite independently using each sub-approach and then merges the (possibly tied) results, called *queues*, into one final list using some selected algorithm. No internal information leaks from any sub-algorithm and, in particular, adaptive prioritization is impossible as the test cases are actually executed only after each sub-approach finished its calculations. The process is shown in Figure 2.3.

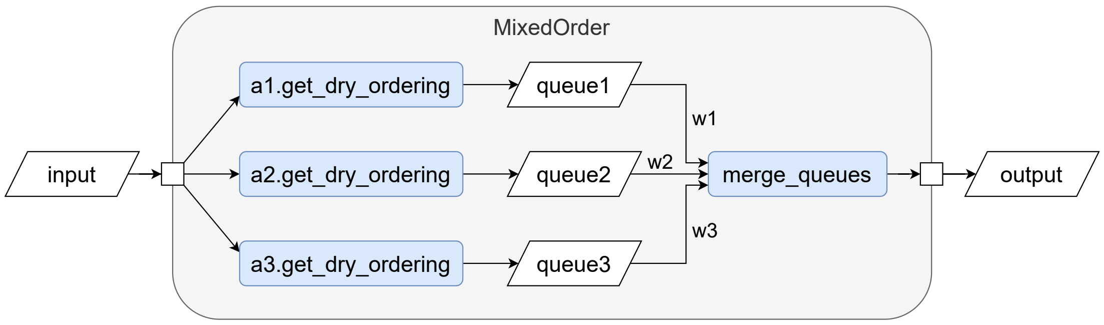


Fig. 2.3. Summary of the general workflow of a mixer (own work).

The only remaining detail is the way the results are merged. The simplest proposed approach, further called *RandomMixedOrder* takes the first test case from the i-th queue, where i was chosen randomly according to the specified weights. Once a test case is executed, it is removed from all the queues, and the method ignores empty queues. For instance, for two sub-approaches with weights set to 0.25 and 0.75, respectively, the mixer would take the test cases from the second queue roughly three times more commonly.

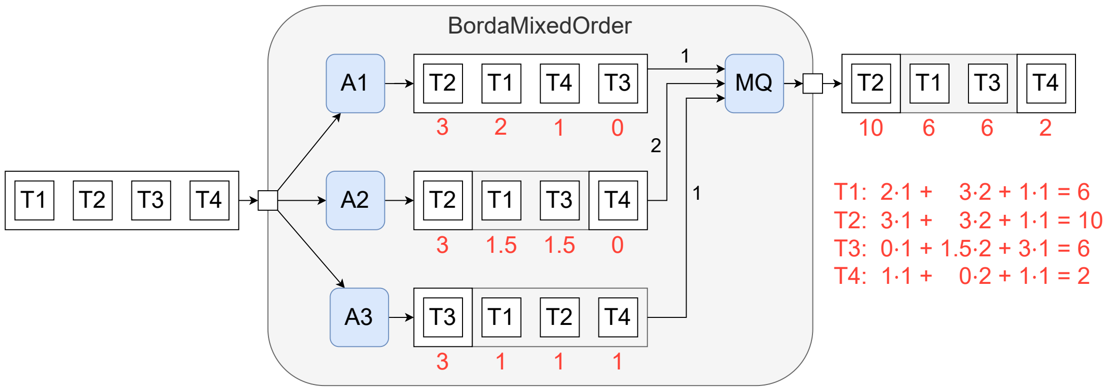


Fig. 2.4. An example of prioritization using `BordaMixedOrder` (own work).

Other proposed merging schemes come directly from electoral system theory. The first approach called *BordaMixedOrder* uses the Borda method for positional voting to arrange the suite. Firstly, for each sub-approach outcome, we assign points to test cases based on their position. The first case in a given queue gets $n - 1$ points (where n is the size of the suite), the second gets $n - 2$, etc. until the penultimate and last test cases get 1 and 0 points, respectively. If two test cases are tied, their points are averaged and the whole mixing can also result in new ties. The points are then multiplied by the weight of a given sub-method and summed up over all queues for each test case. In the end, the test cases are sorted by their total points in descending order and executed [82]. Whenever all weights are equal, a simpler version of this method, used by Cao et al. [76], emerges that ranks test cases according to their average rank in all queues. An example of merging through BordaMixedOrder is shown in Figure 2.4.

The Schulze method is also implemented, as *SchulzeMixedOrder*, in the proposed solution. It is much more robust than the Borda voting system, providing certain desirable qualities, such as the *majority winner* rule – if a test case has more than 50% points, it must win [82]. The entire implementation of the Schulze method is not described in the thesis as it is not a vital part of the solution, and its source code is available in the replication package. However, it should be noted that as a variant of the Floyd-Warshall algorithm is used, the computational complexity of this combinator reaches $O(n^3)$ [82], making it unsuitable for larger collections of test cases. We predict that despite its robustness, the efficiency of this mixer will be sub-par, since early experimentation has shown that a bit of randomness might be actually beneficial for the performance of a method.

In TCPFramework, all mixers implement the `MixedOrder` abstract base class, which has its programming interface presented in Listing 2.2. As an optimization, if a certain sub-approach has the weight of zero, its order is not computed, as all merging schemes should ignore queues produced by these approaches. Prioritization feedback and the reset action are further propagated to all target methods.

```python
class MixedOrder(Approach):
    def __init__(self, targets: Sequence[Approach],
                 weights: list[float] | None) -> None: ...

    @abstractmethod
    def merge_queues(self, queues: list[Ordering]) -> Ordering: ...
```

Listing 2.2. Public interface of the `MixedOrder` abstract base class.

2.4.2. Interpolators

Interpolators work with different approaches and shift their relative importance values as cycles (failed or any) pass. The main motivation in this case is that numerous simple approaches are history-based and thus perform poorly in the early iterations. Examples of such techniques include FoldFailsOrder, ExeTimeOrder, and FailDensityOrder. However, once they collect enough historical information, they can become fairly performant. We consequently want to use these methods once they gather enough information, but replace them with others in the early cycles.

The approach works as follows: two approaches (called *before* and *after*) and a certain cycle count goal (called *cutoff*) are manually set, and the two approaches are mixed according to the current progress towards this objective. So in the first cycle, only the *before* method is executed, and in every cycle after the cutoff, only the *after* technique is run. It should be noted that both approaches can react to execution feedback, regardless of which one was used, meaning that the *after* prioritizer can train itself in earlier cycles. This strategy, called *InterpolatedOrder*, can be conveniently implemented in terms of mixers, as shown in Listing 2.3.

```
if self._cycle >= self._cutoff:
    self._after.prioritize(ctx)
    return
w = self._cycle / self._cutoff
mixer = BordaMixedOrder([self._before, self._after], weights=[1.0 - w, w])
mixer.prioritize(ctx)
```

Listing 2.3. Implementation of an interpolator in terms of a mixer.

2.4.3. Tiebreakers

Tiebreakers resolve ties from an input ordering using another prioritizer or some heuristic. Its construction is motivated by the fact that some approaches naturally produce clustered results. For example, FoldFailsOrder using the total strategy partitions the test suite by the count of previous failures. For early cycles, almost all the cases have a priority value of zero and are treated equally by the prioritizer. Thus, a need to break the ties in the resulting orders emerges.

We propose two tiebreakers. Firstly, the *GenericBrokenOrder* is an universal tiebreaker that runs a first sub-approach producing a clustered test case list, and then executes a second sub-approach to prioritize each group, producing the final order. If the second sub-method returns ties, they are also present in the final output. Similarly to mixers and interpolators, execution feedback and reset signal is propagated to both sub-approaches. The prioritization logic for GenericBrokenOrder is shown in Listing 2.4.

```
for ci, cluster in enumerate(self._target.get_dry_ordering(ctx)):
    for ti, tcs in enumerate(self._breaker.get_dry_ordering(ctx.fork(cluster))):
        for tc in tcs:
            ctx.execute(tc, key=f"{ci}:{ti}")
```

Listing 2.4. The prioritization logic for the GenericBrokenOrder tiebreaker.

Tiebreakers can also perform a role similar to that of interpolators. In early cycles, techniques such as FoldFailsOrder with the total strategy return a single cluster containing all test cases. In such a case, when using a tiebreaker, the entire prioritization is delegated to the second sub-approach.

The second proposed tiebreaker, *CodeDistBrokenOrder*, uses code representation distances to break ties within groups of equal priority. Instead of simply running the CodeDistOrder prioritizer within each cluster, we keep shared sets of prioritized and remaining test cases, ensuring that the chosen candidates are placed as far as possible within the code representation space of the entire test suite.

2.5. Example complete models

Since all techniques described in the previous section are actually entire families of approaches, some concrete, usable examples are given below. We first present a comparison of base methods before combining them with approach combinators. Table 2.1 shows the input data awareness of various base TCP approaches. To construct a high-performing prioritizer, we intuitively want to combine multiple information factors.

Table 2.1. Information factors used by the base TCP approaches.

Approach	**Awareness**			
	History	**Failure**	**Cost**	**Code**
BaseOrder	✗	✗	✗	✗
RandomOrder	✗	✗	✗	✗
TestLocOrder	✗	✗	✗	✓
CodeDistOrder	✗	✗	✗	✓
RecentnessOrder	✓	✗	✗	✗
FoldFailsOrder	✓	✓	✗	✗
ExeTimeOrder	✓	✗	✓	✗
FailDensityOrder	✓	✓	✓	✗

Subsequently, the following sample combined approaches are defined:

- *P1* – a mixer that blends FoldFailsOrder (in the seedless DFE mode, with $\alpha = 0.8$), RecentnessOrder (considering only the latest cases) and ExeTimeOrder (with $\alpha = 0.4$). Available in all mixer variants, that is, random (P1.1), Borda (P1.2), and Schulze (P1.3). As we believe execution time to be less impactful, its priority is set to be twice as low.
- *P2* – an interpolator, which uses the FailDensityOrder after five failed cycles and an equal mix of ExeTimeOrder (with $\alpha = 0.4$) and RecentnessOrder before. This also demonstrates the ability to nest approach combinators within each other, as a Borda mixer is used inside an interpolator.
- *P3* – a tiebreaker, which resolves ties in the FoldFailsOrder prioritizer (with the seedless total strategy), using ExeTimeOrder (P3.1) or CodeDistOrder (P3.2).

The exact definitions of all the sample approach combinator prioritizers mentioned above are shown in Listing 2.5. Both P1 and P3 offer different variants, which will be evaluated against each other in Chapter 3.

```
P11 = RandomMixedOrder([FoldFailsOrder(seed=None),
↪  RecentnessOrder(latest_only=True), ExeTimeOrder()], [1, 1, 0.5])
P12 = BordaMixedOrder([FoldFailsOrder(seed=None),
↪  RecentnessOrder(latest_only=True), ExeTimeOrder()], [1, 1, 0.5])
P13 = SchulzeMixedOrder([FoldFailsOrder(seed=None),
↪  RecentnessOrder(latest_only=True), ExeTimeOrder()], [1, 1, 0.5])

P2 = InterpolatedOrder(BordaMixedOrder([ExeTimeOrder(), RecentnessOrder()]), 5,
↪  FailDensityOrder(), mode="failed")

P31 = GenericBrokenOrder(target=FoldFailsOrder("total", seed=None),
↪  breaker=ExeTimeOrder())
P32 = CodeDistBrokenOrder(target=FoldFailsOrder("total", seed=None))
```

Listing 2.5. Exact definitions of all proposed approach combinator examples.

The following motivations guide each outlined proposal. For P1, we predict that combining the failure and cost information factors, similar to FailDensityOrder, will result in better performance than using either factor individually. We additionally use the latest-only RecentnessOrder approach to increase the priority of new test cases. In terms of weights, failures are generally more important than execution time, according to early experimentation, so we have to reduce the weight of test durations. The weight of recentness prioritizer is less important, as it produces large tied clusters (at most two per suite), which will then get broken by the two other approaches.

For P2, we observe that FailDensityOrder performs well but is extremely unstable in the first iterations. To perform early prioritization, ExeTimeOrder could be used; however, it also needs a few cycles to stabilize its internal state. We therefore use its mix with a generally stable RecentnessOrder until a couple failing cycles are encountered.

Finally, for P3, the total strategy of FoldFailsOrder naturally produces rather large clusters, especially in the early cycles. We predict that breaking these tied groups would significantly boost the effectiveness of the approach. In the described scenario, the tiebreaker simultaneously acts similarly to an interpolator, as in the early cycles the main prioritizer would return just one cluster, with all the cases tied at zero fails.

3. Empirical evaluation of the proposed solution

This chapter describes all aspects of the empirical evaluation of the proposed algorithms. First, selected subject programs used for all research questions are described. The considered evaluation metrics are then outlined, including new metrics proposed as part of this thesis. Research questions are stated, including evaluation metrics and comparison baselines. Finally, we perform an analysis of the achieved results.

3.1. Selected subject programs (RTPTorrent)

The RTPTorrent dataset [38] is used for all evaluations. It comes from the only publication identified in the systematic review of the literature that focused solely on sourcing a high-quality TCP dataset. RTPTorrent consists of multiple mined open-source software projects. The solution is characterized by the following desired qualities [38]:

- It uses data from real software systems instead of synthetically generating them.
- It includes a diverse range of Java repositories in terms of maturity, contributor count, test case count, and codebase size.
- It provides a non-trivial baseline based on a microstudy, in form of the previously mentioned DFE method, which is a better reference point than base or random order.
- It is publicly available and uses only open-source projects from GitHub, which supports the reproducibility of research results.
- It is compatible with the GHTorrent and TravisTorrent [83] datasets.

Arguably, the biggest issue with the dataset is that it only includes open-source Java repositories. This makes the results obtained on RTPTorrent poorly generalizable to projects written in other programming languages and to proprietary systems. Additionally, the granularity of the included test cases is low – test classes are used instead of test methods [38]. Regardless, we believe that RTPTorrent is extremely suitable for TCP studies and beats multiple other datasets.

Particular projects from the dataset can be conveniently loaded through TCPFramework. The cycle test result information from RTPTorrent is enriched by the cycle duration data from TravisTorrent, another compatible open-source dataset [83]. Unfortunately, not all projects could be used for evaluation. The following subject programs were excluded for various reasons: `Achilles`, `buck`, and `sonarqube` – due to the use of too much memory for the evaluation environment; `deeplearning4j`, `graylog2-server`, `jOOQ`, `jcabi-github`, and `jetty.project` – due to a high number of missing cycles, commits, or files; `sling` – due to missing repository data. This resulted in eleven remaining projects of various sizes, which are described in Table 3.1. We do not report the number of lines of code or the number of implementation classes since the projects are evaluated at different points in their lifetimes. This means that in the early cycles they can be tiny, yet grow massively in subsequent cycles. Additional information on the included systems is given in the original publication [38].

Table 3.1. The description of included RTPTorrent repositories.

Subject program	**Count of CI cycles**	**Failure percentage**	**Average number of test cases**	**Total time spent in CI**
`LittleProxy`	427	6.6%	25.6	59 h
`HikariCP`	1 577	5.3%	14.1	76 h
`jade4j`	931	9.1%	38.5	52 h
`wicket-bootstrap`	904	2.7%	42.9	53 h
`titan`	941	10.4%	41.7	1 215 h
`dynjs`	935	1.7%	73.3	229 h
`jsprit`	1 061	4.1%	86.5	137 h
`DSpace`	1 863	1.3%	62.5	326 h
`optiq`	1 306	1.8%	42.2	1 011 h
`cloudify`	4 968	2.6%	55.1	1 447 h
`okhttp`	5 425	6.5%	42.8	868 h

The included repositories should be sufficiently diverse, with cycle counts ranging from 427 to almost 5 000, failed cycle percentages from 1.3% to 10.4%, and average test cases per cycle from 14.1 to almost 90. Finally, the total time spent during continuous integration jobs varies from a few dozen hours to a few months.

The RTPTorrent dataset normally contains a list of CI jobs and test cases executed during them (including verdicts, execution times, and their order). The base project data lack full cycle durations (including non-testing tasks) and the source codes of test classes.

To enrich the dataset with the job execution time, the previously mentioned TravisTorrent dataset was joined using the compatible job identifier. Evaluations in TCPFramework can be performed without joining the TravisTorrent table. In such a case, the cycle duration is assumed to be the sum of time spent on the execution of every test case, and an appropriate warning is presented.

To include the source codes, the full version control system (VCS) repositories of the evaluated systems are downloaded, and the appropriate files are retrieved according to the commit identifiers given by RTPTorrent. This step resulted in the largest count of subject program exclusions, performed due to missing VCS repositories, missing commits, and missing files. The total number of errors identified within the remaining projects with respect to the exclusion reason is shown in Table 3.2 below. We believe that the number of mismatches in selected projects is negligible and does not pose a threat to the validity of the empirical study.

Table 3.2. Errors encountered during the enrichment of RTPTorrent.

Subject program	Invalid commits	Invalid files
`LittleProxy`	0	4
`HikariCP`	0	0
`jade4j`	0	0
`wicket-bootstrap`	3	0
`titan`	0	0
`dynjs`	0	0
`jsprit`	0	0
`DSpace`	0	66
`optiq`	0	0
`cloudify`	0	5
`okhttp`	11	61

3.2. Considered evaluation metrics

To gather objective, measurable, and replicable results, proper evaluation metrics should be selected. All metrics presented in Chapter 1 (that is, APFD, APFD$_C$ in both versions, NAPFD, rAPFD, RPA, NRPA and NTR) were analyzed to select the right indicators for this study.

Firstly, NAPFD and the severity-aware version of APFD$_C$ were ruled out. The NAPFD metric is used to judge prioritization under various resource constraints [73]. Since the type of TCP performed in this work is not constrained by time or the environment, this metric is simply not suitable. Similarly, the full formula for APFD$_C$ uses severity information [72], which is rarely known in real systems [71] and is not available in the RTPTorrent dataset.

Then, RPA and NRPA were excluded. These metrics were previously criticized [8, 70], because they treat all cycles equally (even though failed cycles are more generally important), and since sometimes well-performing permutations may receive low RPA/NRPA scores.

To rate the effectiveness of cost-unaware approaches, we prefer the usage of rAPFD over APFD. The former has a consistent value range of exactly zero to one, where zero is achieved by the worst solution and one is achieved by the optimal ordering [70]. Since rAPFD has the same monotonicity as APFD, which means that if an ordering is better than another in terms of rAPFD it is always also better in terms of APFD, the rectified metric can be safely used. It should be noted that the range of values of plain APFD also depends on the characteristics of the measured suite (in particular, its size). This problem is not present for rAPFD. The only real issue with the rectified metric is that it makes it harder to compare study results with the much more widespread [5] base variant. However, the problem is insignificant, since the direct comparison of the APFD values is already extremely difficult due to the divergences in definitions, metric application, and selected subject programs [30]. Unfortunately, no rectified version of APFD$_C$ is presented in the analyzed literature.

The NTR metric is suitable for evaluating the applicability of a TCP method within CI environments. However, despite being a useful measure, its interpretation might be misleading. Normalized time reduction indicates the percentage time savings assuming that the CI cycle is stopped after the first failure. That said, these savings occur only for failed cycles, which, per Table 3.1, can be as low as 1.3% of all cycles. Meanwhile, the overhead of prioritization must occur for all cycles, since the status of a CI job cannot be known a priori. Thus, a metric showing the actual time saved is needed, including the unfavorable TCP overhead.

In summary, to rate the effectiveness of cost-unaware TCP methods, rAPFD is used. To evaluate the applicability of the techniques, NTR is used. The analysis has shown the need for a rectified version of APFD$_C$ and an applicability metric with better interpretability.

3.2.1. rAPFD$_C$

To evaluate the effectiveness of cost-aware (but severity-unaware) TCP methods, we propose a new metric called the *rectified time-aware average percentage of faults detected* (rAPFD$_C$). Similarly to the rAPFD metric introduced by Zhao et al. [70], rAPFD$_C$ is defined as a min-max normalization of a previously used indicator. The formula for rAPFD$_C$ is shown in Equation (3.1):

$$r\text{APFD}_C\left(s\right) = \frac{\text{APFD}_C\left(s\right) - \text{APFD}_{C\ \text{min}}\left(s\right)}{\text{APFD}_{C\ \text{max}}\left(s\right) - \text{APFD}_{C\ \text{min}}\left(s\right)}. \tag{3.1}$$

The proposed metric inhibits the following desirable qualities:

- The range of values of rAPFD$_C$ is bounded by 0 and 1 ($0 \leq r\text{APFD}_C(s) \leq 1$).
- The optimal solution always has rAPFD$_C$ of 1 ($r\text{APFD}_{C\ \text{max}}(s) = 1$).
- The worst solution always has rAPFD$_C$ of 0 ($r\text{APFD}_{C\ \text{min}}(s) = 0$).
- The metric has the same monotonicity as APFD$_C$ – if an ordering is better than another in terms of rAPFD$_C$, it is also better in terms of APFD$_C$.

Furthermore, although not formally or rigorously proved, we observed that the values of rAPFD$_C$ for the random order tend to approach 0.5, when aggregated over multiple random state seeds. Following an analogous argument to the one regarding rAPFD, it is safe to evaluate solutions using rAPFD$_C$ instead of APFD$_C$.

3.2.2. ATR

As stated previously, a new TCP CI applicability metric is needed for use alongside NTR. The introduced metric should show the total time savings (or costs) of using a given TCP method in a given project, rather than being measured only on the failing cycles.

An excerpt from a diagram presented previously is shown in Figure 3.1. It presents the breakdown of different time durations occurring within a CI cycle which uses TCP. As long as the TCP method does not use information from static analysis, priotization can be performed in parallel with the build process (see Figure 1). It can take either longer or shorter time to complete. If prioritization is faster than compilation and static analysis, there is no time cost to performing TCP; if it is slower, there exists an overhead. Similarly to NTR, we assume that the testing process is concluded after the first failure occurs.

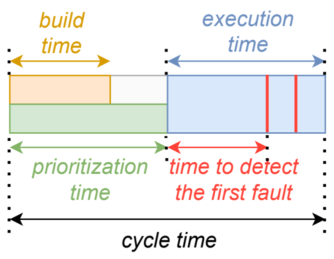


Fig. 3.1. Diagram of static TCP cycle time measures (own work, excerpt from Figure 1.8).

Thus, we introduce *actual time reduction* (ATR), which represents the genuine percentage of time savings from using a given TCP method in a given repository, as an alternative to the normalized time reduction (NTR) metric. The proposed ATR metric sums up the total *testing time* (TT) spent in CI jobs. The testing time is defined using the formula from Equation (3.2):

$$TT = \max\left(PT - BT,\, 0\right) + \begin{cases} TF & \text{if } TF \neq null \\ EF & \text{if } TF = null, \end{cases} \tag{3.2}$$

where: testing time TT – total time spent on regression testing-related activities in a given cycle; prioritization time PT – time spent performing TCP (or zero if TCP is not used); build time BT – time spent on compilation and static analysis independently of testing; time to detect the first fault TF – duration until the end of the first failing test case in the sequence; execution time EF – time spent on executing all test cases in the suite. Consequently, we define the actual time reduction (ATR) for approach $\mathcal{A}$ as in Equation (3.3):

$$\text{ATR} = 1 - \frac{\sum_{i=1}^{\text{CI}} TT_{\mathcal{A}}}{\sum_{i=1}^{\text{CI}} TT_{\text{base}}}. \tag{3.3}$$

All durations should be measured using common units. Although the choice of the unit influences the value of TT, it does not affect ATR. The intuitive interpretation of ATR is as follows: a value of 0.01 indicates that approach $\mathcal{A}$ reduces the time spent on all testing activities by 1%. If an approach has ATR $= -0.01$, it increases the testing time by 1%, making the use of such a method counterproductive. Finally, an ATR of zero indicates that the approach does not influence the time spent in regression testing activities at all.

The approach in which the test cases are executed in the original order is an example of a technique with ATR $= 0$. For presentation purposes, since we expect the values of ATR to be close to zero, they may be scaled up by a constant multiplier. Regardless of scaling, negative ATR indicates TCP methods that would be harmful to apply in practice.

Although $r\text{APFD}_C$ can be used safely instead of APFD_C, the newly introduced ATR metric is vastly different from NTR in terms of construction and interpretation. Thus, to avoid threats to the construct validity, questions regarding the CI applicability of approaches will be answered using both NTR and ATR.

3.3. Research questions

We aim to obtain answers to three high-level questions about the proposed work. First, what choices of parameters of the code representation distance approach perform the best. Second, whether approach combinators manage to beat the sub-approaches they combine. Third, is the performance of proposed solutions comparable with the current state of the art, and whether they can be used to perform TCP in real-world scenarios.

Consequently, the first research question (RQ) is stated as follows, RQ1: *What is the impact of the parameters of CodeDistOrder on its effectiveness?* This question is split into three sub-questions (RQ1.1 through RQ1.3), each examining a different model parameter, that is, normalization type, vector distance, and aggregation function. The explanation of these variables is available in Chapter 2. We test the following values for each parameter: normalization of `formatting` (removal of unnecessary whitespace), `identifiers` (removal of all tokens other than identifiers and type identifiers), or none applied; vector distance of Manhattan or Euclidean distance; and aggregation function of min, average, or max. Cosine similarity is not evaluated since for the selected vectorizer it is equivalent to the Euclidean distance. It should be noted that all parameters are tested separately; as they may interdepend, the combination of the best results from each sub-question might not necessarily yield the best prioritizer. The approach using `identifiers` normalization, Euclidean distance, and min aggregation is used as the baseline. Since CodeDistOrder is not aware of costs, rAPFD is used for evaluation.

The second research question focuses on approach combinators and has the following form, RQ2: *Can approach combinators beat their sub-approaches in terms of effectiveness?* Similarly to RQ1, it was divided into multiple sub-questions. Mixers are covered in RQ2.1, interpolators in RQ2.2, and tiebreakers in RQ2.3. This time, all approaches are compared in terms of $r\text{APFD}_C$ as cost-aware methods are also evaluated. Since approach combinators form entire families of approaches, certain concrete examples have to be selected for comparison. We use the sample models from Chapter 2 for this purpose and compare them against all the sub-approaches that they employ. Consequently, P1 (a mixer of test case failures, recentness, and execution time) is used for RQ2.1; P2 (an interpolator that goes from a mixture of execution time and recentness to failure density) is used for RQ2.2; and P3 (a tiebreaker that breaks total fails with execution time or code representation distance) is used for RQ2.3. The detailed definitions of P1, P2, and P3 are given in Chapter 2.

The last research question investigates the performance of the proposed approach combinator examples (P1, P2, and P3) against solutions from the state of the art and has the following form, RQ3: *What is the performance of the proposed approaches?* The comparison includes two aspects, general effectiveness (measured using rAPFD$_C$) and applicability within CI (in terms of NTR and ATR). We want to evaluate both the theoretical quality of the ordering, as in many other works on TCP, and the real-world practicality. A solution that yields near-perfect test case permutations but increases the total CI time should not be viewed as effective.

The main difficulty related to RQ3 is the selection of representative comparison reference points, both in the form of trivial baselines and state-of-the-art approaches. We first gather all appropriate approaches and evaluate them in terms of rAPFD$_C$, and then select a few of them as baselines to answer RQ3. From the previously described methods, we selected BaseOrder, RandomOrder, FoldFailsOrder (with the DFE strategy), and FailDensityOrder. Two steps were used to find more comparison points. We checked the most popular TCP evaluation baselines from Prado Lima and Vergilio [7], and then collected all prioritization approaches, published since 2020, found in the SR described in Chapter 1. A total of 72 possible techniques were reviewed, with the vast majority ending up excluded. There were four duplicates and two irreproducible studies. Eleven studies had incompatible problem statements, for example, they performed TCP in resource-constrained environments, focused on manual prioritization, or used a different level of test case granularity. Sixteen approaches used different information factors, such as structural coverage, requirement data, or bug reports. Finally, the largest part, consisting of 36 solutions, was discarded due to requiring training. A large upside of the proposed solutions is that they require no training, making them usable in real-world projects even in the first CI runs. At the same time, it makes the comparison with pre-trained ML models biased. In the end, three new techniques were included as baselines, ROCKET by Marijan et al. [40], DBP by Zhou et al. [84], and an unnamed approach by Fazlalizadeh et al. [85].

We believe that the resulting set of seven potential baselines is sufficient and plan to select 2-4 of them for the final evaluation of RQ3. We justify the suitability of some of the selected baselines below:

- Random prioritization is the most common reference point for TCP evaluation, according to Prado Lima and Vergilio [7] and the base order holds second place ex aequo. They are used in a multitude of related studies [9, 28, 29, 40, 67, 74, 84, 85] and can express performance that would be achieved if no prioritization were performed.
- DFE [48] is used as a non-trivial baseline in RTPTorrent [38], which was introduced by a highly cited publication. It represents the use of simple TCP approaches.

- ROCKET [40] is a robust approach, tied as the second most widely used TCP evaluation baseline, per Prado Lima and Vergilio [7]. It uses information factors similar to the proposed methods and, despite being developed in 2013, was used as a reference point as recently as 2023 [9]. Most importantly, it has been shown to perform better than many ML methods (such as support-vector machines, artificial neural networks, and gradient-boosting decision trees) [9]. It represents the state-of-the-art heuristical TCP.

All stated RQs are described along with the used metrics in Table 3.3 below.

Number	**Question**	**Metrics**
RQ1:	*What is the impact of the parameters of CodeDistOrder on its effectiveness?*	—
• RQ1.1:	*How does the choice of* ***normalization type*** *affect CodeDistOrder's effectiveness?*	rAPFD
• RQ1.2:	*How does the choice of* ***vector distance*** *affect CodeDistOrder's effectiveness?*	rAPFD
• RQ1.3:	*How does the choice of* ***aggregation function*** *affect CodeDistOrder's effectiveness?*	rAPFD
RQ2:	*Can approach combinators beat their sub-approaches in terms of effectiveness?*	—
• RQ2.1:	*Can* ***mixers*** *beat their sub-approaches in terms of effectiveness?*	$r\text{APFD}_C$
• RQ2.2:	*Can* ***interpolators*** *beat their sub-approaches in terms of effectiveness?*	$r\text{APFD}_C$
• RQ2.3:	*Can* ***tiebreakers*** *beat their base approaches in terms of effectiveness?*	$r\text{APFD}_C$
RQ3:	*What is the performance of the proposed approaches?*	—
• RQ3.1:	*What is the* ***effectiveness*** *of the proposed approaches?*	$r\text{APFD}_C$
• RQ3.2:	*What is the* ***CI applicability*** *of the proposed approaches?*	NTR, ATR

Table 3.3. The summary of stated research questions.

3.4. Results and analysis

All experiments were conducted on a 4-core 2.42 GHz CPU from 2022 and 16 GB of RAM. These hardware specifications can only affect the ATR metric, with no influence on effectiveness-related measures. To avoid misleading evaluation results, we count the APFD family metrics only for failed cycles, with no data imputation for others, and include only cycles with at least 6 test cases in the suite.

Two main forms of presentation of the results are used. Firstly, the average metric values are organized into tables, with different subject programs in subsequent rows, and various approaches in columns. In the footer, the means and medians across all datasets are presented as we want to find methods which perform the best universally. Since all calculated metrics are normalized, it is safe to aggregate them in this way. For each row, the best result is bolded. Sometimes multiple sub-questions or different aspects are presented in a single table with gaps between each approach groups. In these cases, each group is emphasized separately. Secondly, the results are visualized as box plots, showing the metric value distributions across all datasets. There, boxes show the range between the first and third quartiles, whiskers extend from the box to 1.5x the inter-quartile range, and outliers are shown as dots (○). The median is presented as a red horizontal line (—) and the mean is marked by a small red triangle (▲). The boxes are colored green (🟩) for the baseline methods and yellow (🟨) for the proposed approaches. The vertical axis is automatically scaled to highlight differences between the compared approaches. It should be noted that for rAPFD and rAPFD$_C$ the value range goes from zero to one.

3.4.1. Impact of the CodeDistOrder parameters (RQ1)

The results for RQ1 are collected in Table 3.4 that contains three sections for RQ1.1, RQ1.2, and RQ1.3 organized from left to right. They are also visualized in Figure 3.2 which shows box plots for three sub-questions, RQ1.1 on the top left, RQ1.2 on the top right and RQ1.3 on the bottom.

RQ1.1: *How does the choice of normalization type affect CodeDistOrder's effectiveness?*

We observe that the use of normalization improves the performance achieved by CodeDistOrder. The disabled normalization approach was superior only in the `DSpace` system. The `formatting` (unnecessary whitespace omitted) and `identifiers` (all code other than identifiers stripped) modes both achieved the highest rAPFD on 5 out of 11 subject programs. The `identifiers` normalization had the highest mean and median rAPFD values. We suspect that these dependencies might be related to the slicing parameter introduced in Chapter 2. Both normalization modes can help fit more semantic information in the narrow slicing window, boosting the effectiveness. Regardless, the performance for all settings is unsatisfactory.

RQ1.2: *How does the choice of vector distance affect CodeDistOrder's effectiveness?*

The Manhattan distance showed superior performance in 5 out of 11 subject programs, while the Euclidean distance was better in the remaining six. The Euclidean measure simultaneously had a higher mean and median rAPFD, and we generally find it to be the better technique. However, as in RQ1.1, performance was subpar no matter which distance measure was used.

Table 3.4. Average rAPFD per dataset for RQ1.1, RQ1.2 and RQ1.3.

Subject program	**RQ1.1: Normalization**			**RQ1.2: Distance**		**RQ1.3: Aggregation**		
	None	**Formatting**	**Identifiers**	**Manhattan**	**Euclidean**	**Min**	**Average**	**Max**
`LittleProxy`	.481	.524	**.556**	**.567**	.556	.556	.500	**.642**
`HikariCP`	.289	.334	**.384**	.339	**.384**	.384	.376	**.429**
`jade4j`	.452	**.671**	.498	.476	**.498**	**.498**	.477	.415
`wicket-bootstrap`	.725	**.731**	.532	.529	**.532**	.532	**.534**	.380
`titan`	.111	.170	**.328**	**.339**	.328	.328	.390	**.719**
`dynjs`	.481	.415	**.532**	**.534**	.532	.532	**.536**	.328
`jsprit`	.539	**.556**	.533	**.548**	.533	.533	**.584**	.512
`DSpace`	**.433**	.345	.406	.302	**.406**	**.406**	.282	.362
`optiq`	.444	.419	**.522**	.492	**.522**	**.522**	.416	.455
`cloudify`	.447	**.453**	.413	**.420**	.413	**.413**	.340	.252
`okhttp`	.315	**.316**	.312	.295	**.312**	.312	.437	**.469**
Mean	.429	.449	**.456**	.440	**.456**	**.456**	.443	.451
Median	.447	.419	**.498**	.476	**.498**	**.498**	.437	.429

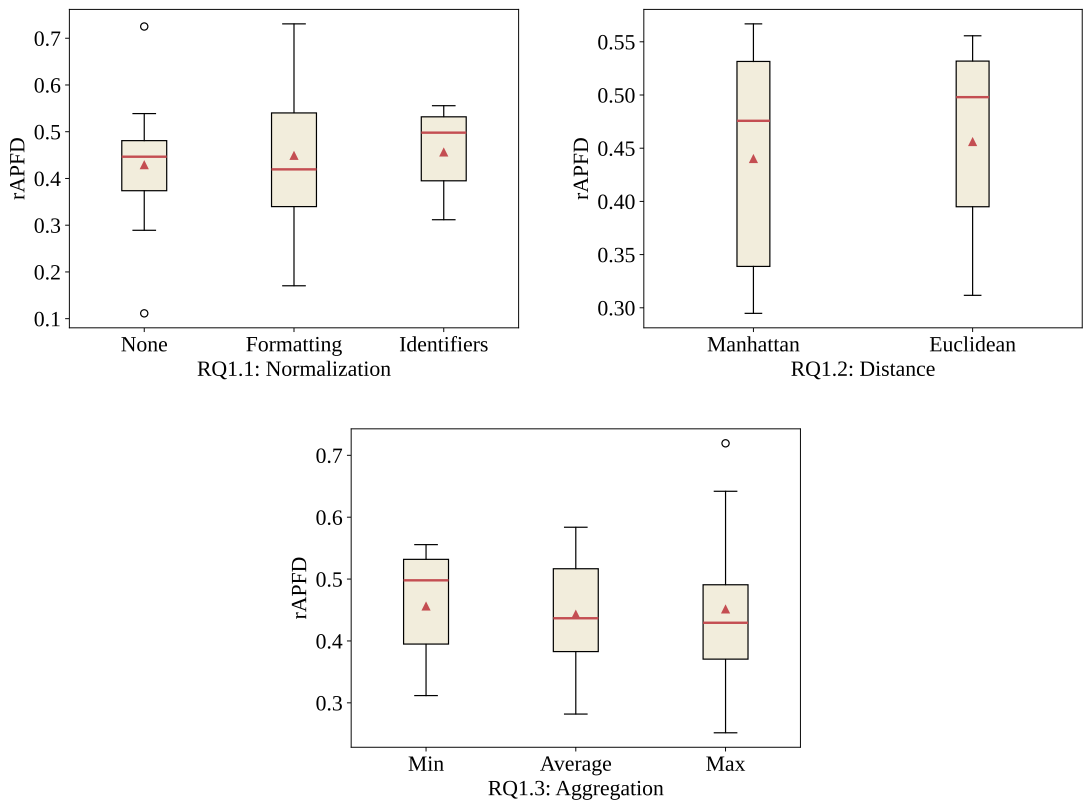


Fig. 3.2. Boxplot of average rAPFD values for RQ1.1, RQ1.2 and RQ1.3 (own work).

RQ1.3: *How does the choice of aggregation function affect CodeDistOrder's effectiveness?*

The best results in terms of the aggregation function are almost evenly split, with four highest values for min, three for average, and four for max. The min technique has both the highest mean and the median, but it is hard to clearly declare it a winner. Once again, all the collected results are unsatisfactory.

RQ1: *What is the impact of the parameters of CodeDistOrder on its effectiveness?*

In terms of mean and median rAPFD, the `identifiers` normalization, Euclidean distance, and min aggregation perform the best in isolation. Concerning normalization, there is definitely some gain in using normalization, regardless of the mode. We recommend using the Euclidean distance even though the results for RQ1.2 were less conclusive. Finally, after a detailed analysis, we find that the aggregation results are highly dependent on the characteristics of the subject program with every setting applicable to some repositories. It would be best to select the appropriate function depending on the system, but if that is not possible, min aggregation can be used instead.

The parameters of CodeDistOrder definitely impact its effectiveness. However, regardless of the selected settings, each benchmark produced poor results. It is understandable since the approach uses only one information factor (see Table 2.1), that said, it is expected that its performance will be higher when combined with other approaches.

3.4.2. Approach combinators compared to base methods (RQ2)

The evaluation metric values for RQ2.1 and RQ2.2 are presented in Table 3.5 and Figure 3.3, while the results for RQ2.3 are shown in Table 3.6 and Figure 3.4. We start this subsection with the results related to mixers and interpolators and then analyze two tiebreaking approaches.

RQ2.1: *Can mixers beat their sub-approaches in terms of effectiveness?*

In RQ2.1, we compare all variants of the P1 approach (P1.1 – random, P1.2 – Borda, and P1.3 – Schulze) against their sub-methods. We can observe the Borda approach achieving the highest results on 6 out of 11 subject programs (with three joint first places tied with Schulze). However, one of the sub-approaches, FoldFailsOrder, achieved best effectiveness in the five remaining systems. The Borda mixer (P1.2) also has the highest mean and median rAPFD$_C$ values. The Schulze mixer (P1.3) was slightly worse, with its performance comparable to that of FoldFailsOrder. Finally, the random mixer (P1.1) had the lowest effectiveness and was beaten by its best sub-approach on 8 out of 11 subject programs. In the conducted experiment, the Borda mixer was definitely the best technique and beat its sub-approaches for the majority of subject programs and also in terms of mean and median rAPFD$_C$.

RQ2.2: *Can interpolators beat their sub-approaches in terms of effectiveness?*

The results of RQ2.2, which compares the proposed P2 interpolator against its sub-approaches, are easier to interpret. The interpolator achieves the highest rAPFD$_C$ metric value on 10 out of 11 datasets, with its best sub-method, FailDensityOrder, being better only for `dynjs`. It also has the highest mean and median rAPFD$_C$. The gains are not large, as FailDensityOrder takes over all prioritization in the late cycles, but it can be safely said that P2 beats its sub-approaches in terms of effectiveness.

Table 3.5. Average rAPFD$_C$ per dataset for RQ2.1 and RQ2.2.

Subject program	**RQ2.1: Mixers vs. sub-methods**						**RQ2.2: Interpolator vs. sub-methods**			
	FoldFails	**Recent**	**ExeTime**	**P1.1**	**P1.2**	**P1.3**	**ExeTime**	**Recent**	**FailDensity**	**P2**
`LittleProxy`	**.697**	.375	.501	.584	.639	.628	.501	.375	.694	**.707**
`HikariCP`	.749	.530	.666	.719	**.846**	.831	.666	.530	**.775**	**.775**
`jade4j`	.661	.805	.657	.652	**.857**	.681	.657	.805	**.887**	**.887**
`wicket-bootstrap`	.748	.727	.495	.756	**.833**	**.833**	.495	.727	.747	**.785**
`titan`	**.896**	.431	.287	.844	.856	.856	.287	.431	.908	**.930**
`dynjs`	.840	.533	.824	.857	**.888**	.885	.824	.533	**.846**	.831
`jsprit`	.872	.623	.601	.917	**.975**	**.975**	.601	.623	.872	**.896**
`DSpace`	**.738**	.482	.676	.619	.719	.719	.676	.482	.738	**.793**
`optiq`	.870	.216	.692	.814	**.889**	**.889**	.692	.216	.870	**.914**
`cloudify`	**.856**	.639	.628	.794	.823	.822	.628	.639	.855	**.868**
`okhttp`	**.905**	.456	.560	.715	.834	.835	.560	.456	.903	**.908**
Mean	.803	.529	.599	.752	**.833**	.814	.599	.529	.827	**.845**
Median	.840	.530	.628	.756	**.846**	.833	.628	.530	.855	**.868**

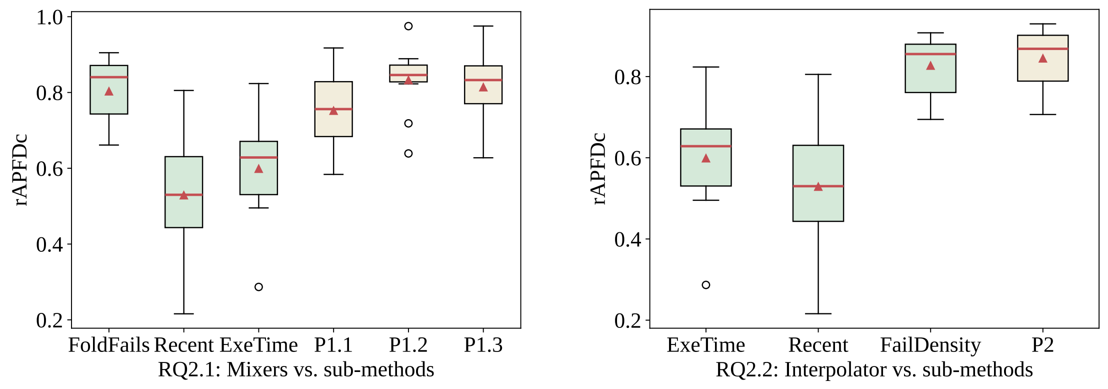


Fig. 3.3. Boxplot of average rAPFD$_C$ values for RQ2.1 and RQ2.2 (own work).

RQ2.3: *Can tiebreakers beat their base approaches in terms of effectiveness?*

As stated previously, two tiebreaker experiments were conducted and their results are shown in Table 3.6 and Figure 3.4. The left half of both the table and the figure shows the evaluation of P3.1 (which breaks ties in TotalFailsOrder using ExeTimeOrder), and the right half presents P3.2 (which works on the same prioritizer but uses CodeDistOrder instead). To answer RQ2.3, we use results from both experiments.

For the execution time tiebreaking, P3.1 achieved the highest results for all subject programs except `HikariCP` and `jade4j`, where surprisingly the generally worse sub-method, ExeTimeOrder, had the best performance. The mean and median rAPFD$_C$ of the tiebreaker was also much higher than that of its sub-approaches.

Similarly, for code representation distance tiebreaking, P3.2 outperformed all other approaches in 10 out of 11 repositories, with the only exception being `HikariCP`, where TotalFailsOrder performed better. It also had the highest mean and median rAPFD$_C$, compared to its sub-methods. However, it performed worse than the execution time tiebreaker. In summary, it can be said that tiebreakers are able to improve the performance of the base approaches.

Table 3.6. Average $r\text{APFD}_C$ per dataset for RQ2.3.

Subject program	**RQ2.3: ExeTime breaking**			**RQ2.3: CodeDist breaking**		
	TotalFails	**ExeTime**	**P3.1**	**TotalFails**	**CodeDist**	**P3.2**
`LittleProxy`	.678	.501	**.716**	.678	.614	**.679**
`HikariCP`	.583	**.666**	.596	**.583**	.450	.579
`jade4j`	.606	**.657**	.617	.606	.379	**.615**
`wicket-bootstrap`	.744	.495	**.898**	.744	.557	**.811**
`titan`	.888	.287	**.899**	.888	.505	**.915**
`dynjs`	.835	.824	**.947**	.835	.397	**.920**
`jsprit`	.819	.601	**.898**	.819	.604	**.847**
`DSpace`	.739	.676	**.822**	.739	.427	**.818**
`optiq`	.852	.692	**.943**	.852	.703	**.915**
`cloudify`	.767	.628	**.809**	.767	.358	**.804**
`okhttp`	.865	.560	**.884**	.865	.368	**.878**
Mean	.761	.599	**.821**	.761	.488	**.798**
Median	.767	.628	**.884**	.767	.450	**.818**

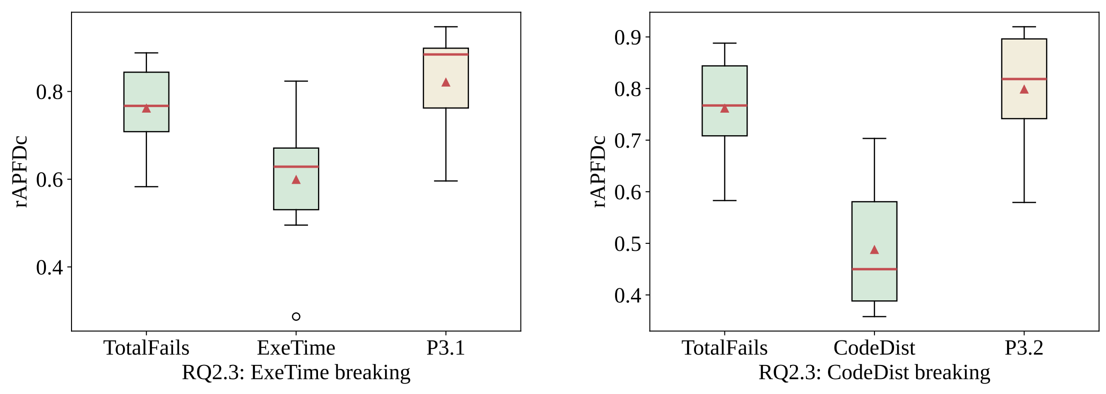


Fig. 3.4. Boxplot of average $r\text{APFD}_C$ values for RQ2.3 (own work).

RQ2: *Can approach combinators beat their sub-approaches in terms of effectiveness?*

To summarize the entire research question, approach combinator-based models can effectively beat their sub-approaches in terms of $r\text{APFD}_C$. Without any fine-tuning, we were able to construct sample approach combinator techniques (P1, P2 and P3), which consistently outperform their base methods across the majority of subject programs. Between all tested mixers, the one that used Borda count (P1.2) had the best effectiveness, and in terms of RQ2.3, the execution time tiebreaker (P3.1) was better than its alternative. Thus, for the comparison in RQ3, the P1.2, P2, and P3.1 prioritizers will be used.

It should be noted that since approach combinators form entire families of approaches, we cannot prove that all constructed combinators outperform their sub-methods. The observed performance relationships apply only to the specific sample prioritizers drawn from these families, rather than to all possible approach combinators.

3.4.3. Performance of the proposed approaches (RQ3)

The performance of the proposed approaches is compared with both trivial baselines and state-of-the-art solutions, in terms of both effectiveness (rAPFD$_C$; Table 3.4 and Figure 3.5) and applicability within CI (NTR and ATR; Table 3.8 and Figure 3.6).

First, to select suitable reference points for further comparison, a preliminary analysis of rAPFD$_C$ of all baselines is performed. We compare: BaseOrder [29] (B1), DBP [84] (B2), RandomOrder [29] (B3), unnamed approach by Fazlalizadeh et al. [85] (B4), DFE [48] (B5), FailDensityOrder (B6), and ROCKET [40] with $m = 100$ (B7) and $m = 1000$ (B8).

The results of this microstudy are shown in the left section of Table 3.7 and Figure 3.5. We observe a clique of low-performance approaches consisting of B1, B2, and B3. Of these, we select B3, which is the random ordering of the test cases, as the lowest baseline. From B4 through B6, we select DFE, as it performed better than the unnamed approach proposed by Fazlalizadeh et al. [85], and was also previously used as a reference point by Mattis et al. [38]. Finally, we select B7 as the ROCKET variant that performs better. In summary, for further evaluation of the proposed techniques in RQ3.1 and RQ3.2, we use the following baselines: random prioritization (B3), DFE (B5), and ROCKET (B7).

RQ3.1: *What is the effectiveness of the proposed approaches?*

We collect the rAPFD$_C$ results for B3, B5, B7, P1.2, P2, and P3.1, which are shown in Table 3.7 and Figure 3.5. The best approaches vary by dataset, with P2 having the highest mean and P3.1 the highest median rAPFD$_C$. Random prioritization and DFE do not achieve the highest result for any dataset. It seems that all proposed approaches can compete with ROCKET in terms of effectiveness. ROCKET also outperforms P1.2 in terms of mean rAPFD$_C$ and P3.1 in terms of average. We perform a statistical test for a more objective analysis later in this chapter.

Table 3.7. Average rAPFD$_C$ per dataset for baselines and RQ3.1.

Subject program	**RQ3: Baselines**								**RQ3.1: Approaches**					
	B1	**B2**	**B3**	**B4**	**B5**	**B6**	**B7**	**B8**	**B3**	**B5**	**B7**	**P1.2**	**P2**	**P3.1**
LittleProxy	.453	.424	.558	.470	.659	**.694**	.584	.426	.558	.659	.584	.639	.707	**.716**
HikariCP	.570	.462	.446	.527	.744	.775	**.793**	.535	.446	.744	.793	**.846**	.775	.596
jade4j	.226	.485	.566	.769	.661	**.887**	.570	.842	.566	.661	.570	.857	**.887**	.617
wicket-bootstrap	.009	.529	.514	.741	**.892**	.747	.855	.783	.514	.892	.855	.833	.785	**.898**
titan	.492	.488	.520	.524	.916	.908	**.930**	.719	.520	.916	**.930**	.856	**.930**	.899
dynjs	.398	.514	.494	.791	.871	.846	**.951**	.948	.494	.871	**.951**	.888	.831	.947
jsprit	.433	.373	.573	.758	.913	.872	**.936**	.724	.573	.913	.936	**.975**	.896	.898
DSpace	.303	.616	.592	.702	.658	.738	.801	**.851**	.592	.658	.801	.719	.793	**.822**
optiq	.205	.508	.457	.429	.836	**.870**	.858	.840	.457	.836	.858	.889	.914	**.943**
cloudify	.333	.492	.443	.758	.849	.855	**.902**	.898	.443	.849	**.902**	.823	.868	.809
okhttp	.333	.459	.512	.544	.771	**.903**	.869	.687	.512	.771	.869	.834	**.908**	.884
Mean	.341	.486	.516	.638	.797	**.827**	.823	.750	.516	.797	.823	.833	**.845**	.821
Median	.333	.488	.514	.702	.836	.855	**.858**	.783	.514	.836	.858	.846	.868	**.884**

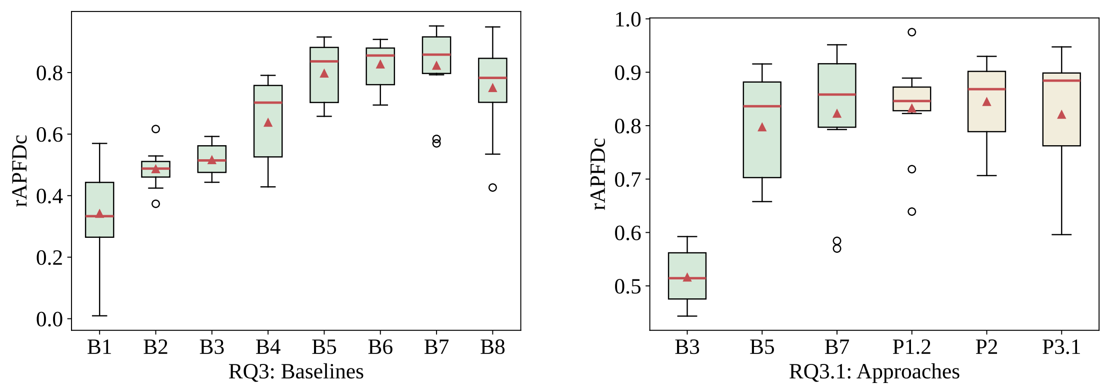


Fig. 3.5. Boxplot of average $r\text{APFD}_C$ values for baselines and RQ3.1 (own work).

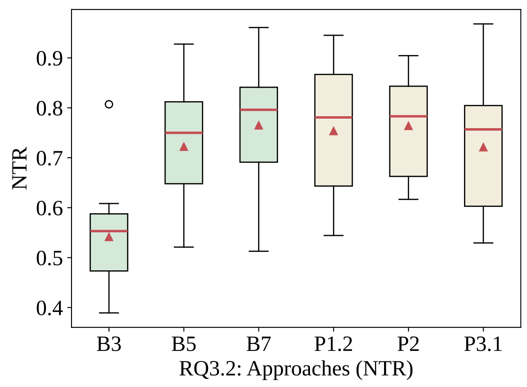
RQ3.2: *What is the CI applicability of the proposed approaches?*

As mentioned previously, we evaluate the applicability within CI in terms of both NTR and ATR, to reduce the risks caused by the use of a new evaluation metric. The results in terms of these two metrics are shown in Table 3.8 and Figure 3.6. While NTR has the standard value range of zero to one, ATR is technically not bounded from either side and can take both positive and negative values. Whenever the ATR is lower than zero, the actual testing time in CI increases when a given approach is applied.

In terms of NTR, the ROCKET approach has the highest mean and median metric value; however, its results are close to those of the proposed approaches. In particular, for P2, which performed the best, the mean NTR is lower by less than one percent, and the median NTR is lower by only 1.6%. Although the best-performing methods vary between subject programs, neither random prioritization nor DFE achieves the highest results for any repository. It should be noted that, despite intuitions that NTR of the base order would be equal to zero, or exactly 0.5, it can actually take any value from the range.

Table 3.8. Average NTR and ATR per dataset for RQ3.2.

Subject program	**RQ3.2: Approaches (NTR)**						**RQ3.2: Approaches (ATR)**					
	B3	**B5**	**B7**	**P1.2**	**P2**	**P3.1**	**B3**	**B5**	**B7**	**P1.2**	**P2**	**P3.1**
`LittleProxy`	.440	.521	.513	.564	**.617**	.570	+.000	+.009	+.008	+.014	**+.020**	+.015
`HikariCP`	.608	.634	.796	**.835**	.635	.590	-.006	-.004	+.011	**+.014**	-.004	-.008
`jade4j`	.807	.750	.863	.900	**.904**	.630	+.019	+.015	+.024	**+.027**	**+.027**	+.005
`wicket-bootstrap`	.472	.807	.768	.747	.746	**.824**	+.012	**+.021**	+.020	+.020	+.020	**+.022**
`titan`	.558	.779	**.825**	.703	.816	.757	+.006	+.022	**+.025**	+.016	**+.024**	+.020
`dynjs`	.590	.928	.961	.945	.870	**.968**	+.001	**+.005**	**+.006**	**+.006**	+.005	**+.006**
`jsprit`	.585	.824	.857	**.899**	.783	.785	+.009	+.021	+.023	**+.025**	+.019	+.019
`DSpace`	.553	.556	.631	.544	**.658**	.616	+.003	+.003	**+.004**	+.003	**+.004**	**+.004**
`optiq`	.474	.817	.693	.781	**.890**	.883	+.003	**+.008**	+.006	+.007	**+.009**	**+.009**
`cloudify`	.389	.662	**.690**	.583	.667	.529	+.002	**+.005**	**+.005**	**+.004**	**+.005**	+.003
`okhttp`	.479	.666	**.818**	.789	.817	.781	+.005	+.013	**+.019**	+.018	**+.019**	+.018
Mean	.541	.722	**.765**	.754	.764	.721	+.005	+.011	**+.014**	**+.014**	**+.013**	+.010
Median	.553	.750	**.796**	.781	.783	.757	+.003	+.009	+.011	+.014	**+.019**	+.009



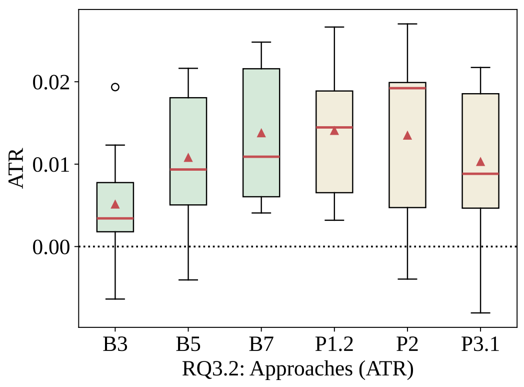


Fig. 3.6. Boxplot of average NTR and ATR values for RQ3.2 (own work).

For ATR, P2 has the highest median value, while the mean value is tied between joint winners: ROCKET, P1.2, and P2. We observe higher ATR values for more sophisticated, yet lightweight methods, which supports the construct validity of the metric. The value of ATR for the base order is also interpretable, as it would be equal to exactly zero in all datasets. It should be noted that ATR is the only investigated metric that could actually vary based on hardware specification, as it includes the measured prioritization time.

One subject program, `HikariCP`, had negative ATR values for the majority of prioritization approaches. It seems to be related to its low total testing time and high cycle count. If a repository has rapid testing turnaround, it might not be useful to perform TCP there.

When interpreted as percentage values, ATR is actually low in all areas, and the best approaches reduce the prioritization time by only up to 2.7%. However, since TCP cannot reduce the actual execution time of particular test cases and also incurs overhead of its own, the reduction is actually quite large. In total terms, the proposed approaches saved, respectively 7.5 h, 11.1 h, and 9.2 h for `titan`; 5.7 h, 6.8 h, and 6.8 h for `optiq`; and 2.9 h, 3.1 h, and 2.7 h for `okhttp`. These time gains may translate into substantial financial savings for these projects.

RQ3: *What is the performance of the proposed approaches?*

Based on the results from RQ3.1 and RQ3.2, we state that the proposed approach combinator prioritizers have a performance comparable to the current heuristical state of the art methods while simultaneously using a novel technique.

To support this claim, we performed a statistical test on the results of RQ3.1 ($r\text{APFD}_C$ values of the proposed algorithms and baselines). The Friedman test, previously used in the field of TCP [9, 70], is employed. It is a non-parametric test, resistant to non-normality and differing variances [86]. The test calculates the average ranks of the treatments (approaches) across all outcomes (subject programs), to check whether the performance differences are statistically significant. If the Friedman test succeeds at a given significance level α, we continue with the post hoc Wilcoxon test, to determine which pairs bear the significant difference [86]. As multiple hypotheses are tested, the procedure must be adjusted, in this case with Holm's method.

For data visualization, we use critical difference (CD) diagrams, originally proposed by Demšar [86]. The CD plot shows the mean positions of the techniques, where the lower results are better and are displayed more to the right. Treatments are connected by a horizontal line if there are no statistically significant differences between their results. In contrast, they are disconnected if significant differences occur.

The results of the test are visualized on a CD diagram in Figure 3.7 below. We can see that in terms of rAPFD$_C$ performance, ROCKET wins but is closely followed by the proposed approaches P2 and P3.1. Then, the P1.2 and DFE approaches follow, and the random approach is the last with a mean rank of 6. However, there are no statistically significant differences (at $\alpha = 0.05$) between the performance of all the state-of-the-art and proposed methods, as marked by the horizontal line. In contrast, a statistically significant difference occurs between random ordering and any other technique in the list.

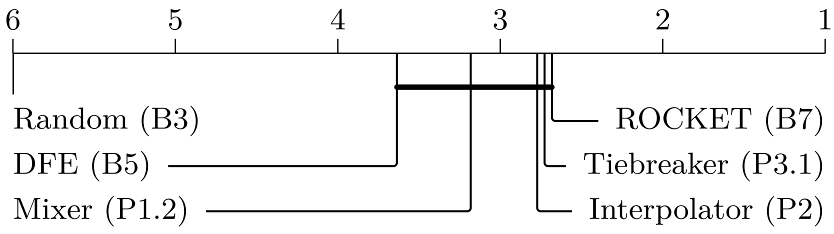


Fig. 3.7. Critical difference plot for RQ3.1 rAPFD$_C$ of different approaches (own work).

In conclusion, the proposed approaches demonstrate effectiveness that is slightly lower than, yet not statistically significantly different from, the current state of the art. The solutions are directly comparable to previous methods, while using a novel technique. In terms of applicability, all tested approaches consistently reduced the time spent testing in all but one subject systems. Their use in CI is thus beneficial and applicable.

3.4.4. Key findings of the empirical study

The key insights from the answers to the research questions are as follows:

- RQ1: The use of varying parameters influences the effectiveness of CodeDistOrder; however, the performance of this method is unsatisfactory and it is not recommended for use without combining it with other approaches.
- RQ2: Prioritizers based on approach combinators can consistently outperform their sub-approaches, even for simple and untuned mixers, interpolators, and tiebreakers.
- RQ3: The proposed methods based on approach combinators achieve performance comparable to the current state of the art, showing no statistically significant differences in terms of relative rAPFD$_C$ ranking, while using a distinct technique.

4. Discussion and further work

The discussion chapter contains general insights formulated during the entire research process. Threats to the validity of the results of the study are elaborately listed. Then, recommendations for other researchers and further work directions are outlined. Finally, since TCP is a highly practical field with widespread industry usage, recommendations for practitioners are described.

4.1. Threats to validity

Since the thesis aims to address two objectives, namely, to summarize the current state of the art of TCP, and to propose a new prioritization approach, the threats to validity sections was split in half to reflect that. Firstly, the concerns regarding the study outlined in Chapter 1 are described. Then, threats related to the proposed solution from Chapter 2 and its evaluation from Chapter 3 are discussed jointly.

4.1.1. Validity of the systematic review

The following threats to the validity of the systematic review are identified:

Internal validity: Concerns about the internal validity of the SR undermine its soundness and coherence. The fact that only one researcher conducted the SR poses the primary threat to its internal validity. To address this threat, state-of-the-art guidelines [19–21] for conducting literature reviews were used. A systematic review protocol was written, which describes, among others, the inclusion and exclusion criteria used to reduce selection bias. The protocol was then reviewed and approved by the supervisor. The use of the snowballing process should also reduce internal validity threats as it reduces the negative effects of constructing an unsound database search query.

External validity: The external validity describes how well the results of the systematic review can be generalized beyond the publications found. It is insignificantly concerned with the choice to select only peer-reviewed studies written in the English language. At the time of the SR, 98.7% of Scopus-searchable papers containing the *test case prioritization* term (within title, abstract, or keywords) were written in English. The second most popular language is Chinese, which would be infeasible for the author to analyze regardless. In terms of peer review, initially gray literature was considered for the SR. However, significant industrial research on TCP [12, 13] was quickly found to also be often published in journals and conferences. Furthermore, due to the use of Google Scholar, the external validity threats resulting from publisher and database bias are greatly reduced [21]. Finally, time frame limitations do not occur, as the years of the included articles were not bounded, and the foundational work on TCP [28, 29, 41, 46, 72] was reached during the search.

4.1.2. Validity of the solution and its evaluation

The following threats to the validity of the empirical study are identified:

Internal validity: The threats to internal validity influence how confidently the observed improvements can be attributed to the design of the proposed techniques. The main threat to internal validity concerns the correctness of the implementation of the proposed solution. Multiple tactics were used to reduce this risk. The entire contributed project uses mypy typechecking and Ruff linting, with all rules enabled (the strictest setting) by default. The proposed framework has correctness precautions built in. For example, results of test cases cannot be seen before they are executed, each case in the suite has to be ordered, and no case can be prioritized twice. The output of the code was compared with the theoretically correct values, especially for the metric calculation. Finally, the research artifacts are published for external inspection. Another smaller threat to internal validity occurs due to mismatches found during the dataset assembly. To address this issue, RTPTorrent repositories with a high count of import errors were excluded from the study, and issue counts within retained projects were listed.

External validity: For the undertaken empirical research, external validity refers mainly to generalizability of the results obtained. The main risk to external validity comes from the fact that only subject programs written in the Java language were tested. The reached conclusions might not generalize to projects written in other languages, especially if they are dissimilar. To reduce this risk, no processing specific to the language of test cases was used in the approaches. The external validity is also threatened by the exclusive use of open-source repositories. However, the usage of proprietary projects would significantly harm the reproducibility of the results. Generally, we believe that despite the reported threats, the use of RTPTorrent was the correct choice for the study. The dataset contains real-world data instead of synthetic test suites, it includes projects with diverse maturities, contributor counts, test case counts, codebase sizes, and failure percentages. All of these characteristics support the generalizability of the research.

Construct validity: The validity of constructs describes how well the gathered measurements reflect the actual quality of solutions. The primary risks in this area refer to the use of less popular rAPFD instead of APFD, and the use of newly proposed rAPFD$_C$ and ATR metrics. The threat arising from the usage of rAPFD and rAPFD$_C$ is considered minimal. Most importantly, these metrics have the same monotonicity as APFD and APFD$_C$, respectively. If a technique performs better than another in terms of rectified metrics, it also achieves proportionally higher values of traditional metrics. Thus, it does not harm the relative comparisons of different algorithms. In terms of absolute evaluation metric values, the results obtained with APFD and APFD$_C$ would be incomparable to prior studies regardless, due to possible divergences in used datasets and methodology. Instead, the rectified metrics have much more convenient semantics, since the scores of 0 and 1 are always achieved for the worst and optimal solutions, respectively. In terms of the newly introduced ATR metric, although it has a logical interpretation in the real world, its construction is very different from that of the NTR metric. To eliminate risks associated with the use of this measure, all applicability results are reported in terms of both NTR and ATR. Regarding the general use of evaluation metrics, we calculate them only for cycles with more than five test cases and never impute missing measurements to address concerns raised in the literature [8, 70].

4.2. Recommendations for researchers

During the research, multiple opportunities for further work were discovered:

Validity of TCP research: We raise some concerns about the validity of the TCP research ecosystem. Firstly, following Mattis et al. [38], we observe that TCP research is often done on synthetic datasets, proprietary data, or projects that do not reflect real-world software development practices. Secondly, not all studies are reproducible, and some do not publish their results at all [31]. Finally, incorrect use of metrics occurs in the literature [8, 70]. We recommend adhering to the subsequent guidelines. Studies should use open-source datasets with diverse subject programs, to support reproducible research and increase ecological validity; for example, the RTPTorrent [38] dataset may be used. Their results should be compared with baselines better than random prioritization. Lastly, researchers should be careful when calculating evaluation metrics. APFD and APFD$_C$ are unsuitable for many studies due to their misleading value range. We instead propose using rAPFD and rAPFD$_C$. Regardless, these metrics should only be averaged over failing cycles with at least six test cases, without data imputation for other jobs [8].

Unification of TCP: We observe a lack of unification within the TCP research. We recommend that future researchers focus on contributing universal datasets, tools, and platforms for TCP evaluation. TCPFramework introduced in this thesis is an attempt at such a contribution. Meanwhile, we suggest using the RTPTorrent [38] dataset for evaluation. Furthermore, we recommend the creation of a new comprehensive TCP approach taxonomy, considering the wide use of hybrid methods. Finally, to improve the adoption of TCP techniques in practice, programming interfaces for priorization should be present in popular testing frameworks.

Approach combinators: There are numerous areas for improvement of the approach combinator methodology. Firstly, additional combinators, or even types of combinators, may be proposed. Although they are applicable to almost all TCP methods, in this thesis only combinators using heuristical algorithms were constructed. There is an opportunity to evaluate the combinator technique on machine learning-based approaches, both supervised and reinforcement, in the future. Finally, we see an option to use supervised ML strategies on some combinators; namely, the weights of mixers may be automatically learned.

4.3. Recommendations for practitioners

To sum up all the insights from both the systematic review and empirical experimentation, the primary guidance for practitioners is that prioritization can provide measurable gains for CI processes. If implemented correctly, TCP usage should reveal regression test failures earlier, and if testing is stopped after the first fault is identified, TCP can provide noticeable time and resource savings.

Simultaneously understanding the difficulties of integrating heavy ML-based TCP solutions in CI pipelines, we recommend the use of lightweight prioritization methods such as ones constructed with the approach combinator framework, which do not require training or historical testing data. Our methods can reduce the testing time by up to 2.7%, which for the evaluated systems corresponds to up to 11 hours of testing time saved. These results can be achieved with only a few dozen lines of code and negligible incurred overhead.

Furthermore, we recommend using TCP methods that incorporate different information factors, such as combining previous failure history, average execution times, or code metrics, since such techniques achieved higher performance in our empirical evaluations. The use of approach combinators allows to mix these factors, without implementing specialized hybrid algorithms manually.

Conclusion

All the stated thesis objectives were fully achieved and even exceeded, and the entire scope of the work was fully realized. The systematic literature review was carried out, resulting in the identification of 292 primary studies and 32 secondary studies. Furthermore, numerous sub-areas of TCP were described and two SR research questions were answered. For the empirical part of the thesis, an entire family of TCP methods consisting of three categories was proposed, implemented, and empirically evaluated using the state-of-the-art TCP evaluation methodology. The proposed solutions achieved performance comparable with the current heuristical state of the art, despite using a novel approach. Furthermore, numerous additional contributions exceeding the original scope of the thesis were achieved, as outlined below:

- Implementation of an open-source, unified TCP approach development and evaluation infrastructure (TCPFramework) that can be used to test various prioritization methods on the state-of-the-art RTPTorrent dataset.
- Review and analysis of existing TCP evaluation metrics, including recent proposals, such as rAPFD, RPA, NRPA, and NTR.
- Proposition of two new TCP evaluation metrics: $r\text{APFD}_C$ (to assess the quality of prioritized suites) and ATR (to rate the time effectiveness of approaches).
- Investigation of the use of code representation distances to prioritize test cases and resolve ordering ties, including a comparison of achieved performance based on different model parameters.

The key findings of the literature survey are reiterated as follows:

- SRRQ1: There are many TCP evaluation tools available and none of them are universally used. The choice of the dataset should be influenced by the exact problem statement and other factors. For general use, we recommend the RTPTorrent [38] dataset, since it is publicly available and includes diverse subject programs.
- SRRQ2: It is hard to establish the state-of-the-art TCP algorithms as studies from this area are mostly incomparable due to divergences in datasets, metrics, tools, information factors, and problem statements used. Some studies do not empirically evaluate their proposed approaches at all. The most widely used evaluation reference points are base order, random order, and the ROCKET [40] algorithm.

Similarly, the main highlights of the empirical study are shown below:

- RQ1: The use of varying parameters influences the effectiveness of code representation distance-based prioritization; however, the performance of this method is insufficient in isolation, and it is not recommended for use without combining it with other approaches.
- RQ2: Prioritizers based on the approach combinators framework can consistently outperform their sub-approaches, even for simple and untuned mixers, interpolators, and tiebreakers. We recommend using approach combinators to merge the insights from prioritizers facilitating different test suite information factors.
- RQ3: The proposed methods based on approach combinators achieve performance comparable to the current state of the art, showing no statistically significant differences in terms of relative $r\text{APFD}_C$ ranking, while using a distinct technique. In terms of applicability, all proposed approaches consistently reduced the time spent testing in all but one subject system.

The results of the study were presented in conjunction with a replication package to support reproducible research. Furthermore, appendices are attached, containing the systematic review protocol and reproduction instructions.

The research carried out during the creation of this thesis will be additionally submitted to the Journal of Systems and Software under the name *Unifying Test Case Prioritization with Approach Combinators*.

Although many further work opportunities remain, this thesis marks a step forward in addressing the problem of constructing universal, efficient, and applicable test case prioritization approaches, to reduce the costs of software testing.

Bibliography

[1] R. K. Saha, S. Khurshid, D. E. Perry, An empirical study of long lived bugs, in: 2014 Software Evolution Week – IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), 2014, pp. 144–153. doi:10.1109/CSMR-WCRE.2014.6747164.

[2] S. Ajorloo, A. Jamarani, M. Kashfi, M. Haghi Kashani, A. Najafizadeh, A systematic review of machine learning methods in software testing, Applied Software Computing 162 (2024) 111805. doi:10.1016/j.asoc.2024.111805.

[3] H. Washizaki, Guide to the Software Engineering Body of Knowledge (SWEBOK Guide), IEEE Computer Society, 2024. URL: https://www.swebok.org, version 4.0, retrieved March 8, 2025.

[4] G. J. Myers, C. Sandler, T. Badgett, The Art of Software Testing, John Wiley & Sons, Inc., 2011. 3rd edition.

[5] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, R. Tumeng, Test case prioritization approaches in regression testing: A systematic literature review, Information and Software Technology 93 (2018) 74–93. doi:10.1016/j.infsof.2017.08.014.

[6] A. Haghighatkhah, M. Mäntylä, M. Oivo, P. Kuvaja, Test prioritization in continuous integration environments, Journal of Systems and Software 146 (2018) 80–98. doi:10.1016/j.jss.2018.08.061.

[7] J. A. Prado Lima, S. R. Vergilio, Test case prioritization in continuous integration environments: A systematic mapping study, Information and Software Technology 121 (2020) 106268. doi:10.1016/j.infsof.2020.106268.

[8] M. Bagherzadeh, N. Kahani, L. Briand, Reinforcement learning for test case prioritization, IEEE Transactions on Software Engineering 48 (2022) 2836–2856. doi:10.1109/TSE.2021.3070549.

[9] D. Marijan, Comparative study of machine learning test case prioritization for continuous integration testing, Software Quality Journal 31 (2023) 1415–1438. doi:10.1007/s11219-023-09646-0.

[10] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: a survey, Software Testing, Verification & Reliability 22 (2012) 67–120. doi:10.1002/stv.430.

[11] A. Bajaj, O. P. Sangwan, A systematic literature review of test case prioritization using genetic algorithms, IEEE Access 7 (2019) 126355–126375. doi:10.1109/ACCESS.2019.2938260.

[12] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, J. Micco, Taming Google-scale continuous testing, in: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), 2017, pp. 233–242. doi:10.1109/ICSE-SEIP.2017.16.

[13] J. Czerwonka, R. Das, N. Nagappan, A. Tarvo, A. Teterev, CRANE: Failure prediction, change analysis and test prioritization in practice – experiences from Windows, in: 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, 2011, pp. 357–366. doi:10.1109/ICST.2011.24.

[14] A. S. Yaraghi, M. Bagherzadeh, N. Kahani, L. C. Briand, Scalable and accurate test case prioritization in continuous integration contexts, IEEE Transactions on Software Engineering 49 (2023) 1615–1639. doi:10.1109/TSE.2022.3184842.

[15] L. Madeyski, B. Kitchenham, Would wider adoption of reproducible research be beneficial for empirical software engineering research?, Journal of Intelligent & Fuzzy Systems 32 (2017) 1509–1521. doi:10.3233/JIFS-169146.

[16] J. Biolchini, P. Gomes Mian, A. Candida Cruz Natali, G. Horta Travassos, Systematic Review in Software Engineering, Technical Report, COPPE / UFRJ, 2005. URL: https://www.cos.ufrj.br/uploadfile/es67905.pdf, retrieved February 22, 2025.

[17] B. Kitchenham, S. Charters, D. Budgen, P. Brereton, M. Turner, S. Linkman, M. Jørgensen, E. Mendes, G. Visaggio, Guidelines for performing systematic literature reviews in software engineering, Technical Report, Keele University, 2007. URL: https://madeyski.e-informatyka.pl/download/Kitchenham07.pdf, version 2.3, retrieved February 22, 2025.

[18] B. Kitchenham, Procedures for Performing Systematic Reviews, Technical Report, Keele University, 2004. URL: https://madeyski.e-informatyka.pl/download/Kitchenham04a.pdf, retrieved February 22, 2025.

[19] B. Kitchenham, L. Madeyski, D. Budgen, SEGRESS: Software Engineering Guidelines for REporting Secondary Studies, IEEE Transactions on Software Engineering 49 (2023) 1273–1298. doi:10.1109/TSE.2022.3174092.

[20] D. Moher, L. Shamseer, M. Clarke, D. Ghersi, A. Liberati, M. Petticrew, P. Shekelle, L. A. Stewart, PRISMA-P Group, Preferred reporting items for systematic review and meta-analysis protocols (PRISMA-P) 2015 statement, Systematic Reviews 4 (2015) 1. doi:10.1186/2046-4053-4-1.

[21] C. Wohlin, Guidelines for snowballing in systematic literature studies and a replication in software engineering, in: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14, Association for Computing Machinery, 2014, pp. 1–10. doi:10.1145/2601248.2601268.

[22] L. Yang, J. Chen, H. You, J. Han, J. Jiang, Z. Sun, X. Lin, F. Liang, Y. Kang, Can code representation boost IR-based test case prioritization?, in: 2023 IEEE 34th

International Symposium on Software Reliability Engineering (ISSRE), 2023, pp. 240–251. doi:10.1109/ISSRE59848.2023.00077.

[23] G. Guizzo, J. Petke, F. Sarro, M. Harman, Enhancing genetic improvement of software with regression test selection, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 1323–1333. doi:10.1109/ICSE43902.2021.00120.

[24] R. Huang, Q. Zhang, D. Towey, W. Sun, J. Chen, Regression test case prioritization by code combinations coverage, Journal of Systems and Software 169 (2020) 110712. doi:10.1016/j.jss.2020.110712.

[25] A. Vescan, R. Găceanu, A. Szederjesi-Dragomir, Embracing unification: A comprehensive approach to modern test case prioritization, in: Proceedings of the 19th International Conference on Evaluation of Novel Approaches to Software Engineering – ENASE, INSTICC, SciTePress, 2024, pp. 396–405. doi:10.5220/0012631000003687.

[26] R. Mukherjee, K. S. Patnaik, A survey on different approaches for software test case prioritization, Journal of King Saud University – Computer and Information Sciences 33 (2021) 1041–1054. doi:10.1016/j.jksuci.2018.09.005.

[27] A. Singh, A. Singhrova, R. Bhatia, D. Rattan, A Systematic Literature Review on Test Case Prioritization Techniques, John Wiley & Sons, Ltd, 2023, pp. 101–159. doi:10.1002/9781119896838.ch7.

[28] G. Rothermel, R. Untch, C. Chu, M. Harrold, Prioritizing test cases for regression testing, IEEE Transactions on Software Engineering 27 (2001) 929–948. doi:10.1109/32.962562.

[29] G. Rothermel, R. Untch, C. Chu, M. 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), 1999, pp. 179–188. doi:10.1109/ICSM.1999.792604.

[30] Y. Singh, A. Kaur, B. Suri, S. Singhal, Systematic literature review on regression test prioritization techniques, Informatica (Slovenia) 36 (2012) 379–408. URL: https://www.informatica.si/index.php/informatica/article/view/420/424, retrieved March 2, 2025.

[31] C. Catal, D. Mishra, Test case prioritization: a systematic mapping study, Software Quality Journal 21 (2013) 445–478. doi:10.1007/s11219-012-9181-z.

[32] H. de S. Campos Junior, M. A. P. Araújo, J. M. N. David, R. Braga, F. Campos, V. Ströele, Test case prioritization: a systematic review and mapping of the literature, in: Proceedings of the XXXI Brazilian Symposium on Software Engineering, SBES '17, Association for Computing Machinery, 2017, p. 34–43. doi:10.1145/3131151.3131170.

[33] M. d. C. de Castro-Cabrera, A. García-Dominguez, I. Medina-Bulo, Trends in

prioritization of test cases: 2017-2019, in: Proceedings of the 35th Annual ACM Symposium on Applied Computing, SAC '20, Association for Computing Machinery, 2020, p. 2005–2011. doi:10.1145/3341105.3374036.

[34] H. Abubakar, F. Zambuk, U. Ahmed, A. Gital, A review on the new trend in regression test case prioritization, ATBU Journal of Science, Technology and Education 11 (2023) 426–436. URL: https://www.atbuftejoste.com.ng/index.php/joste/article/view/1837/pdf_1229, retrieved March 2, 2025.

[35] H. Do, G. Rothermel, On the use of mutation faults in empirical assessments of test case prioritization techniques, IEEE Transactions on Software Engineering 32 (2006) 733–752. doi:10.1109/TSE.2006.92.

[36] Q. Luo, K. Moran, D. Poshyvanyk, M. Di Penta, Assessing test case prioritization on real faults and mutants, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 240–251. doi:10.1109/ICSME.2018.00033.

[37] J. Mendoza, J. Mycroft, L. Milbury, N. Kahani, J. Jaskolka, On the effectiveness of data balancing techniques in the context of ML-based test case prioritization, in: Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2022, Association for Computing Machinery, 2022, p. 72–81. doi:10.1145/3558489.3559073.

[38] T. Mattis, P. Rein, F. Dürsch, R. Hirschfeld, RTPTorrent: An open-source dataset for evaluating regression test prioritization, in: Proceedings of the 17th International Conference on Mining Software Repositories, MSR '20, Association for Computing Machinery, 2020, p. 385–396. doi:10.1145/3379597.3387458.

[39] Q. Luo, K. Moran, D. Poshyvanyk, A large-scale empirical comparison of static and dynamic test case prioritization techniques, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Association for Computing Machinery, 2016, p. 559–570. doi:10.1145/2950290.2950344.

[40] D. Marijan, A. Gotlieb, S. Sen, Test case prioritization for continuous regression testing: An industrial case study, in: 2013 IEEE International Conference on Software Maintenance, 2013, pp. 540–543. doi:10.1109/ICSM.2013.91.

[41] W. Wong, J. Horgan, S. London, H. Agrawal, A study of effective regression testing in practice, in: Proceedings The Eighth International Symposium on Software Reliability Engineering, 1997, pp. 264–274. doi:10.1109/ISSRE.1997.630875.

[42] Y. Lou, J. Chen, L. Zhang, D. Hao, Chapter one – a survey on regression test-case prioritization, Advances in Computers 113 (2019) 1–46. doi:10.1016/bs.adcom.2018.10.001.

[43] R. Pan, M. Bagherzadeh, T. A. Ghaleb, L. Briand, Test case selection and prioritization using machine learning: a systematic literature review, Empirical Software

Engineering 27 (2021) 29. doi:10.1007/s10664-021-10066-6.

[44] C. Henard, M. Papadakis, M. Harman, Y. Jia, Y. Le Traon, Comparing white-box and black-box test prioritization, in: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, Association for Computing Machinery, 2016, p. 523–534. doi:10.1145/2884781.2884791.

[45] L. Zhang, D. Hao, L. Zhang, G. Rothermel, H. Mei, Bridging the gap between the total and additional test-case prioritization strategies, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, 2013, p. 192–201. doi:10.1109/ICSE.2013.6606565.

[46] S. Elbaum, A. G. Malishevsky, G. Rothermel, Prioritizing test cases for regression testing, SIGSOFT Software Engineering Notes 25 (2000) 102–112. doi:10.1145/347636.348910.

[47] D. Di Nardo, N. Alshahwan, L. Briand, Y. Labiche, Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system, Software Testing, Verification and Reliability 25 (2015) 371–396. doi:10.1002/stvr.1572.

[48] J.-M. Kim, A. A. Porter, A history-based test prioritization technique for regression testing in resource constrained environments, Proceedings of the 24th International Conference on Software Engineering. ICSE 2002 (2002) 119–129. doi:10.1109/ICSE.2002.1007961.

[49] H. Park, H. Ryu, 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, 2008, pp. 39–46. doi:10.1109/SSIRI.2008.52.

[50] D. Leon, A. Podgurski, A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases, in: Proceedings of the 14th International Symposium on Software Reliability Engineering, ISSRE '03, IEEE Computer Society, 2003, p. 442. doi:10.1109/issre.2003.1251065.

[51] Y. Ledru, A. Petrenko, S. Boroday, N. Mandran, Prioritizing test cases with string distances, Automated Software Engineering 19 (2012) 65–95. doi:10.1007/s10515-011-0093-0.

[52] R. Carlson, H. Do, A. Denton, A clustering approach to improving test case prioritization: An industrial case study, in: 2011 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp. 382–391. doi:10.1109/ICSM.2011.6080805.

[53] J. Arafeen, H. Do, Test case prioritization using requirements-based clustering, in: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, 2013, pp. 312–321. doi:10.1109/ICST.2013.12.

[54] W. Zhou, X. Jiang, C. Qin, C-CORE: Clustering by code representation to prioritize

test cases in compiler testing, CMES – Computer Modeling in Engineering and Sciences 139 (2024) 2069–2093. doi:10.32604/cmes.2023.043248.

[55] H. Srikanth, L. Williams, J. Osborne, System test case prioritization of new and regression test cases, in: 2005 International Symposium on Empirical Software Engineering, 2005, p. 10. doi:10.1109/ISESE.2005.1541815.

[56] H. Srikanth, C. Hettiarachchi, H. Do, Requirements based test prioritization using risk factors, Information and Software Technology 69 (2016) 71–83. doi:10.1016/j.infsof.2015.09.002.

[57] B. Korel, L. H. Tahat, M. Harman, Test prioritization using system models, in: 21st IEEE International Conference on Software Maintenance (ICSM'05), 2005, pp. 559–568. doi:10.1109/ICSM.2005.87.

[58] B. Korel, G. Koutsogiannakis, Experimental comparison of code-based and model-based test prioritization, in: 2009 International Conference on Software Testing, Verification, and Validation Workshops, 2009, pp. 77–84. doi:10.1109/ICSTW.2009.45.

[59] Y. Huang, T. Shu, Z. Ding, A learn-to-rank method for model-based regression test case prioritization, IEEE Access 9 (2021) 16365–16382. doi:10.1109/ACCESS.2021.3053163.

[60] Z. Li, M. Harman, R. M. Hierons, Search algorithms for regression test case prioritization, IEEE Transactions on Software Engineering 33 (2007) 225–237. doi:10.1109/TSE.2007.38.

[61] S. Li, N. Bian, Z. Chen, D. You, Y. He, A simulation study on some search algorithms for regression test case prioritization, in: 2010 10th International Conference on Quality Software, 2010, pp. 72–81. doi:10.1109/QSIC.2010.15.

[62] D. Di Nucci, A. Panichella, A. Zaidman, A. De Lucia, Hypervolume-based search for test case prioritization, in: M. Barros, Y. Labiche (Eds.), Search-Based Software Engineering, Springer International Publishing, 2015, pp. 157–172. doi:10.1007/978-3-319-22183-0_11.

[63] Y. Singh, A. Kaur, B. Suri, Test case prioritization using ant colony optimization, SIGSOFT Software Engineering Notes 35 (2010) 1–7. doi:10.1145/1811226.1811238.

[64] D. Gao, X. Guo, L. Zhao, Test case prioritization for regression testing based on ant colony optimization, in: 2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS), 2015, pp. 275–279. doi:10.1109/ICSESS.2015.7339054.

[65] M. M. Öztürk, A bat-inspired algorithm for prioritizing test cases, Vietnam Journal of Computer Science 5 (2018) 45–57. doi:10.1007/s40595-017-0100-x.

[66] K. H. S. Hla, Y. Choi, J. S. Park, Applying particle swarm optimization to prioritizing test cases for embedded real time software retesting, in: 2008 IEEE 8th International

Conference on Computer and Information Technology Workshops, 2008, pp. 527–532. doi:10.1109/CIT.2008.Workshops.104.

[67] H. Spieker, A. Gotlieb, D. Marijan, M. Mossige, 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, ISSTA 2017, Association for Computing Machinery, 2017, p. 12–22. doi:10.1145/3092703.3092709.

[68] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono, S. Russo, Learning-to-rank vs ranking-to-learn: strategies for regression testing in continuous integration, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20, Association for Computing Machinery, 2020, p. 1–12. doi:10.1145/3377811.3380369.

[69] D. Marijan, M. Liaaen, A. Gotlieb, S. Sen, C. Ieva, TITAN: Test suite optimization for highly configurable software, in: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2017, pp. 524–531. doi:10.1109/ICST.2017.60.

[70] Y. Zhao, D. Hao, L. Zhang, Revisiting machine learning based test case prioritization for continuous integration, in: 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE Computer Society, 2023, pp. 232–244. doi:10.1109/ICSME58846.2023.00032.

[71] Z. Wang, C. Fang, L. Chen, Z. Zhang, A revisit of metrics for test case prioritization problems, International Journal of Software Engineering and Knowledge Engineering 30 (2020) 1139–1167. doi:10.1142/S0218194020500291.

[72] S. Elbaum, A. Malishevsky, G. Rothermel, Incorporating varying test costs and fault severities into test case prioritization, in: Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001, 2001, pp. 329–338. doi:10.1109/ICSE.2001.919106.

[73] X. Qu, M. B. Cohen, K. M. Woolf, Combinatorial interaction regression testing: A study of test case generation and prioritization, in: 2007 IEEE International Conference on Software Maintenance, 2007, pp. 255–264. doi:10.1109/ICSM.2007.4362638.

[74] J. A. Prado Lima, S. R. Vergilio, A multi-armed bandit approach for test case prioritization in continuous integration environments, IEEE Transactions on Software Engineering 48 (2022) 453–465. doi:10.1109/TSE.2020.2992428.

[75] S. M. J. Hassan, D. Jawawi, J. Ahmad, An exploratory study of history-based test case prioritization techniques on different datasets, Baghdad Science Journal 21 (2024) 609–621. doi:10.21123/bsj.2024.9604.

[76] T. Cao, T. Vu, H. Le, V. Nguyen, Ensemble approaches for test case prioritization in ui testing, in: The 34th International Conference on Software Engineering and

Knowledge Engineering, 2022, pp. 231–236. doi:10.18293/SEKE2022-148.

[77] S. Mondal, R. Nasre, Hansie: Hybrid and consensus regression test prioritization, Journal of Systems and Software 172 (2021) 110850. doi:10.1016/j.jss.2020.110850.

[78] M. Parsa, A. Ashraf, D. Truscan, I. Porres Paltor, On optimization of test parallelization with constraints, in: Software Engineering Workshops 2016, CEUR-WS.org, 2016, p. 164–171. URL: https://ceur-ws.org/Vol-1559/paper21.pdf, retrieved May 21, 2025.

[79] P. Hrkút, M. Ďuračík, Š. Toth, M. Meško, Current trends in the search for similarities in source codes with an application in the field of plagiarism and clone detection, in: 2023 33rd Conference of Open Innovations Association (FRUCT), 2023, pp. 77–84. doi:10.23919/FRUCT58615.2023.10143064.

[80] N. Reimers, I. Gurevych, 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, 2019, pp. 3982–3992. doi:10.18653/v1/D19-1410.

[81] L. P. da Silva, P. Vilain, LCCSS: A similarity metric for identifying similar test code, in: Proceedings of the 14th Brazilian Symposium on Software Components, Architectures, and Reuse, SBCARS ’20, Association for Computing Machinery, 2020, p. 91–100. doi:10.1145/3425269.3425283.

[82] F. Brandt, V. Conitzer, U. Endriss, J. Lang, A. D. Procaccia (Eds.), Handbook of Computational Social Choice, Cambridge University Press, 2016. doi:10.1017/CBO9781107446984.

[83] M. Beller, G. Gousios, A. Zaidman, TravisTorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration, in: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), 2017, pp. 447–450. doi:10.1109/MSR.2017.24.

[84] Z. Q. Zhou, C. Liu, T. Y. Chen, T. H. Tse, W. Susilo, Beating random test case prioritization, IEEE Transactions on Reliability 70 (2021) 654–675. doi:10.1109/TR.2020.2979815.

[85] Y. Fazlalizadeh, A. Khalilian, M. Abdollahi Azgomi, S. Parsa, Incorporating historical test case performance data and resource constraints into test case prioritization, in: C. Dubois (Ed.), Tests and Proofs, Springer Berlin Heidelberg, 2009, pp. 43–57. doi:10.1007/978-3-642-02949-3_5.

[86] J. Demšar, Statistical comparisons of classifiers over multiple data sets, The Journal of Machine Learning Research 7 (2006) 1–30. URL: https://jmlr.org/papers/volume7/demsar06a/demsar06a.pdf, retrieved June 5, 2025.

List of Figures

List of Listings

List of Tables

Appendix

A. Systematic review protocol

Tomasz Chojnacki

The following review protocol describes the methods used to carry out a systematic review (SR) on the topic of test case prioritization (TCP). The protocol is structured according to the guidelines provided by Kitchenham et al. [17] (the simplified version recommended for Ph.D. students) and the PRISMA-P statement [20] (as recommended by SEGRESS [19]).

Additional information is provided in Chapter 1 of the main work.

Introduction

The purpose of the study is to find relevant TCP datasets and algorithms (with a slight emphasis on machine learning solutions; however, other works should also be included). The reasons for conducting the review are:

- To locate opportunities for further improvement in existing studies.
- To summarize the foundational studies and the current state-of-the-art of TCP research for use in the related work description of the thesis.

Based on the research purpose described above, the following systematic review research questions (SRRQs) were formulated:

- SRRQ1: *What are the available datasets and subject programs for TCP?*
- SRRQ2: *What are the state-of-the-art TCP algorithms?*

Search process

The search process follows the guidelines of Kitchenham et al. [17] and especially SEGRESS [19]. Furthermore, a slightly altered version of the snowballing method proposed by Wohlin [21] and later acknowledged by Kitchenham et al. [19] is used. The theory of the snowballing method is described in detail in Chapter 1, together with the alterations. The requirements and guidelines for different stages of the search process are listed below:

Initial papers: The snowballing approach requires a *start set* of papers to begin the iterative process. Initially, a list of four papers, provided by the thesis supervisor in the original submission of the thesis topic, is available, namely the works of Yang et al. [22], Marijan [9], Yaraghi et al. [14] and Bagherzadeh et al. [8]. Since these papers were unmethodically cherry-picked, they cannot be directly used as the start set. However, since they are all recent and unrelated publications on the process of TCP, they can provide insight into which terms and keywords are used in the field.

Start set: The extracted list of keywords along with the SRRQs and eligibility criteria described below will be used to construct a search query for the Scopus[1] database. From the search results, a start set of a few unrelated papers published in the last 5 years will be selected. Original submission papers may be included in the start set if deemed appropriate.

Snowballing: Subsequently, for backward snowballing, the reference list located in the paper text should be traversed, and for forward snowballing, Google Scholar[2] is used. Each encountered paper is validated regarding the inclusion and exclusion criteria and gets included in the list if it fulfills all requirements.

Author search: The snowballing guidelines recommend contacting the authors of the articles after the iterative process is concluded [21]. However, this is infeasible to perform in the time allocated for master's thesis completion. Thus, instead of this procedure, if certain authors appear at least five times in the paper collection, their list of publications will be analyzed separately (through Scopus, Google Scholar, ORCID[3], and personal websites) in search of omitted works, without direct contact.

Kitchenham et al. [17] suggests that in the case of a single researcher, techniques used to reduce bias should be noted. To mitigate the issue, all stages of the systematic review will be presented to the thesis supervisor.

Inclusion and exclusion criteria

According to the guidelines of Wohlin [21], the screening is done by examining the title and abstract, the place of the citation, and only if these are insufficient, the full text of the article. All articles should be analyzed at least to the level of abstract inspection, which means that a given article should not be included or excluded solely on the basis of title.

Articles must meet at least one of the following **inclusion criteria** to be included:

1. The article presents a new TCP dataset (SRRQ1).
2. The article describes a new TCP algorithm (SRRQ2).
3. The article is a secondary study on the topic of TCP.
4. The article compares available TCP solutions.

The *paper kind* is defined for included works. A paper has a kind of `SRRQ1` if it fulfills the first condition in the list, `SRRQ2` if it fulfills the second, `SR` if it fulfills the third, and `CMP` if it fulfills the last. A given paper can have multiple kinds (for example, if it introduces both a dataset and an algorithm), and it must belong to at least one kind.

For the purpose of the review, the use of an open-source program as a dataset does not grant the article the kind of `SRRQ2`. However, creating an entirely new dataset, either from existing data or synthetically, modifying an existing dataset, or gathering own data internally, all count towards this label.

Articles that violate any of the following **exclusion criteria** will be excluded:

[1] `https://www.scopus.com/search/form.uri`
[2] `https://scholar.google.com`
[3] `https://orcid.org`

1. The paper places the main emphasis on test case generation, selection, or minimization.
2. The paper is unrelated to the Computer Science research area.
3. The paper was not peer-reviewed or has not yet been published.
4. The paper consists of less than 5 pages.
5. The paper was not written in the English language.

The third exclusion criterion implicitly disallows the inclusion of blogs, vlogs, or other types of gray literature. Although engineering blogs may provide interesting solutions, they are often available in the form of a research paper or are biased.

Furthermore, while not an explicitly formulated exclusion criterion, papers published after January 27th, 2025 (the commencement date of the SR search process) will not be included. These articles would not have been available during the search.

Data collection

The following information will be collected for each article:

- Digital Object Identifier (DOI) of the paper, if available.
- Title of the paper.
- Authors of the article.
- Year of publication (the lowest date if it was published multiple times).
- Search stage (e.g., *Iteration #1: Backward*) in which the paper was found.
- Paper kind of the article (as described in the previous section).
- Short notes regarding the paper (mainly names of datasets or algorithms).

It is sometimes recommended to maintain the list of excluded papers, similarly to the list of inclusions [17]. However, because of the huge amount of papers evaluated during snowballing, some of which are obviously unrelated to the studied topic, it was decided not to keep a list of excluded papers. Individual excluded papers might get highlighted if the circumstances of their exclusion are particularly interesting.

Data analysis

The resulting list of included papers alongside the collected data, ordered by the inclusion order, will be kept. All records will be stored in a spreadsheet file using the XLSX format and published in the reproduction package.

The following aggregations of results will be analyzed:

- total number of new papers included in each search process stage,
- total number of papers per year,
- total number of papers per kind.

All identified SRs covering the entire research area will be listed chronologically, along with relevant information: target years, limitations, and analyzed paper counts. Although other SRs are of little interest in the context of this study, they greatly aid in the snowballing process, since they include many references.

The most relevant papers, namely those that describe foundational aspects of TCP, introduce new datasets, or produce the state-of-the-art results, will be further analyzed as part of the thesis research. Different subareas of the topic will be summarized, as well as key definitions and metrics.

The results of the study will be published as part of the author's master's thesis. The detailed search process advances along with any divergences from the protocol will be described in the relevant chapter of the main work.

B. Research reproduction instructions

The following appendix describes the research reproduction instructions, including the project architecture, required dependencies, useful commands for replicating the research questions, and, finally, guidelines for the extension of TCPFramework. The source code can be accessed at `https://github.com/LechMadeyski/MSc25TomaszChojnacki`.

Repository structure

The structure of the repository is as follows:

- `experiments/` – early experimentation artifacts;
- `figures/` – code used to generate figures for this thesis;
- `tcp-framework/` – root directory of TCPFramework:
  - `datasets/` – RTPTorrent and TravisTorrent files;
  - `tcp_framework/` – source directory of TCPFramework:
    - `approaches/` – implementation of approaches from Chapter 2;
    - `datatypes/` – domain-specific classes of TCPFramework;
  - `check.sh` – script for formatting, linting and typechecking at once;
  - `pyproject.toml` – definition of Python version and dependencies;
  - `rq1.py` – script gathering results for RQ1;
  - `rq2.py` – script gathering results for RQ2;
  - `rq3.py` – script gathering results for RQ3;
- `literature-review.xlsx` – results of the SR from Chapter 1.

Running the code

The officially supported way to set up the repository is to use uv[1] – a popular Python package and project manager written in Rust. The uv executable can be installed through standalone installers available for download from the project website or through pip.

Although not recommended, it should be possible to run the project without uv, by manually installing the dependencies listed in `pyproject.toml` in Python 3.13 or newer. These include `gitpython`, `matplotlib`, `pandas`, `sentence-transformers`, `tqdm`, `tree-sitter`, and `tree-sitter-java`.

[1] `https://docs.astral.sh/uv`

The project uses Ruff[2], a formatter and linter from the authors of uv. The code can be formatted and linted with `uv run ruff format` and `uv run ruff check`, respectively. The repository also uses mypy[3] for typechecking. The framework can be checked using the `uv run mypy .` command.

To run the project as is, selected repositories of the RTPTorrent [38] dataset are required, which are not included in the replication package. Firstly, the ZIP file from the RTPTorrent Zenodo package[4] should be downloaded. Inside, the `rtp-torrent` directory contains subfolders for different source programs. The main CSV file from each project directory should be extracted into the `tcp-framework/datasets` directory and optionally renamed. Additionally, the `tr_all_built_commits.csv` file should be moved to the same directory (alternatively, the `travistorrent_8_2_2017.csv` from the TravisTorrent [83] dataset can be used to achieve more reliable applicability results). Finally, the Git repositories for the selected projects should be cloned into the `datasets` directory, under the same names as their corresponding CSV files. Hyperlinks to the repositories used in this study are presented in Table B.1. Sample setup instructions for Unix-based operating systems are presented in Listing B.1.

Table B.1. Hyperlinks to the project repositories used in the study.

System	Repository
LittleProxy	github.com/adamfisk/LittleProxy
HikariCP	github.com/brettwooldridge/HikariCP
jade4j	github.com/neuland/jade4j
wicket-bootstrap	github.com/martin-g/wicket-bootstrap
titan	github.com/thinkaurelius/titan
dynjs	github.com/dynjs/dynjs
jsprit	github.com/graphhopper/jsprit
DSpace	github.com/DSpace/DSpace
optiq	github.com/julianhyde/optiq
cloudify	github.com/CloudifySource/cloudify
okhttp	github.com/square/okhttp

```
wget https://zenodo.org/records/4046180/files/rtp-torrent-v11.zip
unzip rtp-torrent-v11.zip
mv rtp-torrent/brettwooldridge@HikariCP/brettwooldridge@HikariCP.csv
↪  tcp-framework/datasets/HikariCP.csv
mv rtp-torrent/tr_all_built_commits.csv tcp-framework/datasets
git clone https://github.com/brettwooldridge/HikariCP
↪  tcp-framework/datasets/HikariCP
```

Listing B.1. Example dataset extraction commands for TCPFramework.

Scripts used for research questions can be evaluated using the `uv run` subcommand, that is, `uv run rq1.py`, `uv run rq2.py`, `uv run rq3.py`, and other scripts using TCPFramework can be executed analogously.

[2] https://docs.astral.sh/ruff
[3] https://www.mypy-lang.org
[4] https://zenodo.org/records/4046180

Extending the project

The solution is highly expandable. New approaches should be implemented as subclasses of the `Approach` abstract base class. For simple prioritizers, only the `prioritize` method has to be implemented. It should call the `ctx.execute` function on subsequent test cases extracted from `ctx.test_cases` property. More advanced techniques may optionally override the `get_dry_ordering`, `on_static_feedback`, and `reset` methods. Their usage examples can be seen by inspecting the approaches packed in the `approaches` module. A simple implementation of a custom algorithm that orders test cases in random order can be seen in Listing B.2.

```python
class RandomOrder(Approach):
    def __init__(self, seed: int = 0) -> None:
        self._seed = seed
        self._rng = Random(seed)

    @override
    def prioritize(self, ctx: RunContext) -> None:
        cases = ctx.test_cases[:]
        self._rng.shuffle(cases)
        for tc in cases:
            ctx.execute(tc)

    @override
    def reset(self) -> None:
        self._rng.seed(self._seed)
```

Listing B.2. Example implementation of a simple custom prioritization approach.

Custom datasets can be included by subclassing the `Dataset` class. Most importantly, the `cycles` method should be overriden. The `MetricCalc` class is not designed with expandability in mind; the easiest way to calculate new metrics is to modify its code. On the other hand, the approaches related to code representation and approach combinators are highly modular and should be easy to extend.