Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

TPVis: A visual analytics system for exploring test case prioritization methods

José Arthur Silveira[a,*], Leandro Vieira[b], Nivan Ferreira[a]

[a]*Centro de Informática (CIn), Universidade Federal de Pernambuco (UFPE), Av. Jornalista Aníbal Fernandes, s/n - Cidade Universitária (Campus Recife) 50.740-560, Recife, PE, Brazil*
[b]*Motorola Mobility, Campinas, São Paulo, Brazil*

ARTICLE INFO

Article history:
Received August 28, 2024

Keywords: Software Testing, Visual Analytics, Test Case Prioritization.

ABSTRACT

Software testing is a vital tool to ensure the quality and trustworthiness of the pieces of software produced. Test suites are often large, which makes the process of testing software a costly and time-consuming process. In this context, test case prioritization (TCP) methods play an important role by ranking test cases in order to enable early fault detection and, hence, enable quicker problem fixes. The evaluation of such methods is a difficult problem, due to the variety of the methods and objectives. To address this issue, we present TPVis, a visual analytics framework that enables the evaluation and comparison of TCP methods designed in collaboration with experts in software testing. Our solution is an open-source web application that provides a variety of analytical tools to assist in the exploration of test suites and prioritization algorithms. Furthermore, TPVis also provides dashboard presets, that were validated with our domain collaborators, that support common analysis goals. We illustrate the usefulness of TPVis through a series of use cases that illustrate our system's flexibility in addressing different problems in analyzing TCP methods. Finally, we also report on feedback received from the domain experts that indicate the effectiveness of TPVis. TPVis is available at `https://github.com/vixe-cin-ufpe/TPVis`.

© 2024 Elsevier B.V. All rights reserved.

1. Introduction

As software complexity is continuously growing, so is the need to ensure that it behaves as expected. Software testing is a vital tool to ensure the quality and trustworthiness of the pieces of software produced. However, test suites are often large, which makes the process of testing software throughout its updates a time-consuming process. This is especially true in the scenario of regression testing when continuously changing software is retested to ensure that new additions do not introduce new bugs. In fact, it has been reported that regression testing can represent 80% of the testing budget and also up to 50% of the cost of software maintenance [1]. With the increasing demand for software development and the adoption of agile methodologies, regression testing has become essential. As software grows in complexity, test suites tend to grow larger. As a result, running these test suites becomes an (possibly prohibitively) expensive process, both in terms of time and also with respect to human and computational resources. It is important to notice that this is not a new problem; back in 2003, some test suites needed up to 7 weeks to execute fully [2].

To solve the problem of executing large test suites, there are

*Corresponding author.
e-mail: `jaobs@cin.ufpe.br` (J.A. Silveira), `leandvie@motorola.com` (L. Vieira), `nivan@cin.ufpe.br` (N. Ferreira)

four main approaches: (I) Test case reduction: Remove redundant or irrelevant tests from the test execution set; (II) Test case selection: Execute only tests that test the software code changes in the current version; (III) Test case prioritization: Execute all tests, but prioritize them in a matter to achieve some goal (e.g., Improve the fault detection rate or improve coverage faster); (IV) Hybrid approach: Mixing the different methods and creating a reduced and prioritized test suite [3, 4].

The test case prioritization (TCP) problem has received a lot of attention in the research community and also attained practical success. For example, Google has reported that applying test case prioritization can significantly reduce the time to find failing tests [5]. There have been many algorithms for test case prioritization. However, generally, they differ in data input and approach. These approaches range from simpler heuristics (e.g., Ordering the tests by source code coverage) to more advanced matters such as machine learning models (e.g., Learn to rank) [6, 7, 8, 9]. Each algorithm may perform differently depending on the software project or data input, so picking a suitable TCP algorithm becomes crucial for obtaining satisfactory results [10].

To address the challenge, we propose TPVis, a visual analysis toolkit intended to support test engineers and test managers in inspecting test case metadata and comparing TCP methods. TPVis was developed in collaboration with software testing professionals and researchers and consists of an interactive system with 12 analytical tools to facilitate the visualization and comparison of prioritization algorithms. To the best of our knowledge, this is the first visual analytics system that targets the goal of supporting the comparison of TCP algorithms. These tools can cover a wide range of analysis tasks and can be used to build dashboards in a wide range of ways. We demonstrate the effectiveness of our tool in two use cases and also report on feedback received by domain experts. Finally, although we demonstrate the tool with a specific dataset, it is essential to note that a generic dataset can be passed to the application, enabling it to be used within different software projects and integrated with Continuous Integration and Continuous Delivery (CI/CD) pipelines. To summarize, our contributions are:

- We propose TPVis (Test Prioritization Visualization): A web-based toolkit that allows software engineers, test engineers, and other software-related stakeholders to explore and inspect how different TCP approaches may perform and behave in their software. To the best of our knowledge, TPVis is the first visual analytics system that targets the comparative analysis of test case prioritization algorithms.
- We introduce a number of tools in TPVis that aid in inspecting the TCP algorithm's behavior and performance. The tools cover different use cases, from test metadata analysis to prioritization comparison and source code coverage evolution.
- We demonstrate the usefulness of TPVis by examining two case studies regarding defects4j projects [11].

2. Related works

Test case prioritization. TCP is an active research field with a lot of work done around the topic. The simplest techniques are the greedy algorithms [12, 13, 14], which find the next best test to add to the prioritized test set, according to some criteria. These algorithms are, in general, easy to implement and cheap in terms of computational cost. Another approach uses genetic algorithms, which try to iterate generations intending to improve a fitness function (which models how good a prioritization is in some constraint, like time to execute or failure detection rate) [15]. Machine learning based methods have also been proposed to perform TCP [16, 17, 9].

Visualization of rankings. Since TCP algorithms are in fact different ways to rank test cases, we also surveyed different approaches for ranking visualization. Rankexplorer considers ranking changes over time and proposes a visualization to track the change and different attributes [18]. LineUp [19], allows the visualization of multiple rankings side by side while displaying multiple heterogeneous attributes. Mylavarapu et al. [20] compared seven visualizations through a crowd-sourced study with 180 participants, concluding that each method has different advantages and disadvantages, highlighting that more than a single visualization may be necessary for this problem.

Visualization of test data. Strandberg et al. [21] have shown that visualizing test results data can aid in stakeholders' decision-making processes. Stranberg et al. [22] proposed Tim, a system designed for test result data visualization. While TPVis characteristics align with those of Tim, our system focuses on inspecting TCP rather than focusing solely on test results. Test suites can also be used to gain better insight into the system under test. In the work of Cornelisse et al. [23], visualizations were based on JUnit tests and UML sequence diagrams, which proved effective in better understanding the project's inner workings. Hammad et al. [24] proposed another visualization system to assist in understanding failed test cases. It can perform analysis at multiple levels (system, package, and class) and has been validated through a user study. Some works visualize test suites at a source code level [25, 26]. Despite all this prior work related to visualizing test suites, we did not find literature related to test case prioritization visualization.

3. Background

Regression testing. As software ages, the cost of maintaining it becomes higher than the cost of building it. Regression testing emerges as a solution to validate that new changes do not break prior use cases. A good regression testing process should help to reduce maintenance costs and improve software reliability [27]. There are two types of regression testing: (I) Corrective, where the software's specification does not change, and (II) Progressive, when the software specification changes to accommodate a new feature or enhancement [28]. Mainly due to the corrective tests, the test suite usually increases in size during the software's life cycle.

Test case prioritization. Test case prioritization methods rank test cases based on specific goals. One common goal is to detect faulty tests as soon as possible. Other possibilities include

Table 1: APFD calculation example in a hypothetical test suite. An X on a cell indicates that the test case (row) detects the corresponding fault (column).

Test	Fault				
	F1	F2	F3	F4	F5
A	X				X
B	X			X	X
C	X	X	X	X	X
D					X
E					

cost reduction (where tests are prioritized based on their execution cost) or maximizing code coverage. The type of tests to be prioritized can vary and be automated (unit tests) or manual. In the case of automated tests, the prioritization algorithms usually have access to source code information as input. For this reason, they are classified as *white-box* techniques. Some examples of white-box prioritization methods are the greedy total and greedy additional (GA) algorithms [12], which give more priority to tests that cover larger portions of the software. When dealing with manual tests, the source code is usually not available or is not feasible for test engineers to use. These cases rely on black-box techniques, which can take previous test results as input or solely the test metadata (e.g., steps and description). As examples in this class of TCP algorithms, we can mention I-TSD [29] and STR [30]. I-TSD uses normalized compression distance to measure the difference between test sets. STR, on the other hand, uses only test strings as input and picks the most distant test from those already selected. One important observation is that some algorithms can be used in a black-box or white-box manner, depending on the data type it processes.

In this paper, we will refer to an instance of a TCP algorithm for a particular data type input as a *TCP Method*, which would be a possible approach that could be chosen by the user for a given use case. For example, *GA-Line*, *GA-Function*, and *GA-Branch* methods refer to the greedy additional algorithm with line, function, and branch-level coverage data inputs, respectively.

Evaluation of TCP methods. The most common metric used to evaluate test prioritizations is the *average percentage fault detection rate* (APFD). The computation of the APFD for a prioritized test suite containing n tests and covering m faults is given in Formula 1, where TF_j represents the rank in the prioritization of the first test to reveal the fault j.

$$\text{APFD} = 1 - \frac{\sum_{j=1}^{m} TF_j}{nm} + \frac{1}{2n} \tag{1}$$

APFD values range from 0 to 1, the higher the value the faster a prioritized test suite finds failures. To better illustrate this concept, we demonstrate the APFD computation on a hypothetical test suite containing 5 tests and which cover 5 software faults, shown in Table 1. Given the stipulated sequence of test cases ABCDE, we can compute the Average Percentage of Fault Detection (APFD) metric by considering the first test to reveal each fault (TF_j). Specifically, TF_1 is assigned a value of 1, since the initial test case uncovers fault F1. Subsequently, TF_2 is assigned a value of 3, indicating the position of the test case that identifies the second fault, and so forth. Considering that n and m would be equal to 5, our APFD value would be 0.7.

While the APFD metric is commonly used, it has several limitations. For instance, it does not account for the severity of faults, treating them equally regardless of their impact. In real-world scenarios, detecting critical bugs early is often more crucial than finding minor ones, but APFD fails to consider this aspect. Moreover, APFD also disregards the test case execution cost, which may vary due to different resource consumption and running time. Finally, APFD only focuses on fault detection and does not consider other quality factors, such as fault diagnosis. Multiple metrics try to address these issues, such as APFDc [31] (which factors in the cost of test execution), APXC [14, 32] (which takes into account how code coverage increases in a given prioritization), and WGFD [33], that allows for a generic concept of "fastness" depending on the use case (e.g, detecting severe faults faster).

Given the size and complexity of test suites, as well as the variety of prioritization methods, we argue that more than the sole use of metrics is needed for stakeholders to make informed decisions on the appropriate approach to their use cases. The goal of TPVis is to make this process more efficient, enabling the evaluation of different aspects of TCP methods.

Continuous integration and delivery. The acceleration of software delivery while simultaneously meeting the dynamic expectations of end-users presents a challenge. Continuous Integration and Continuous Delivery (CI/CD) methodologies emerge as key in addressing it, facilitating the cooperation of development and operations teams through an automated process. CI/CD ensures that every change in the code is subjected to an automated process of testing, building, and deployment. Such methodology expedites the transition from development to production and guarantees that the software releases consistently and reliably [34], avoiding manual (error-prone) work.

4. The TPVis system

This section describes the design process of TPVis, the final design of our system, and details its implementation.

4.1. System requirements

We followed an iterative methodology, during which we had multiple meetings with three domain experts (one software engineering professional and two researchers) who had experience in software testing. During these meetings, we were able to elicitate the following system requirements:

[R1] Enable effective assessment of prioritization performance. The system should support different visual summaries to enable the evaluation of each prioritization technique performance from multiple perspectives, including the test metadata;

[R2] Enable historical evaluation of prioritization methods. Software projects are evolving entities. Some TCP methods may become more efficient as new versions of a project are created and tested. Therefore, it is important

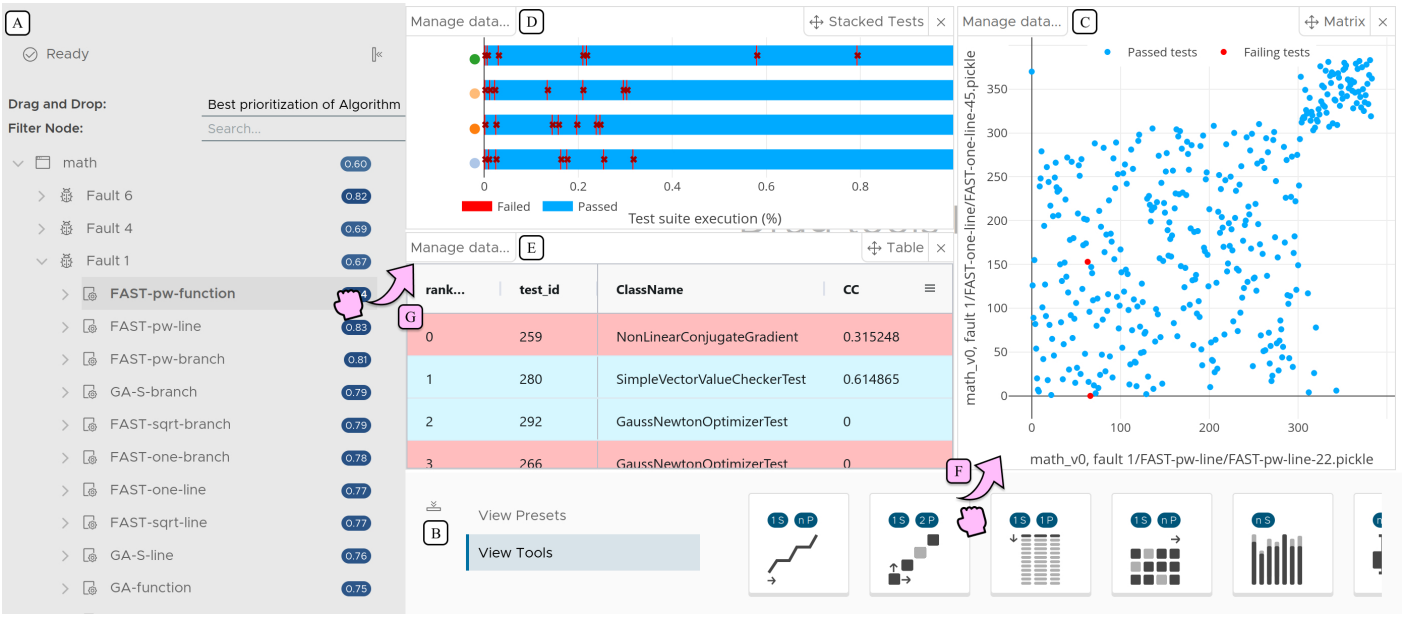


Fig. 1: An overview of the TPVis interface. The system is organized in multiple views, including the (A) *Prioritization tree*, the (B) *Tools pane*. On the dashboard, it is possible to instantiate the different analytical tools showcased in (B) by dragging and dropping (F) them into the dashboard. In this example, the (C) *Matrix*, (D) *Stacked Tests*, and (E) *Table* tools are instantiated. Datasets can be loaded into the different tools by dragging and dropping (G) the corresponding data to the *Manage data* button in each view. The accompanying video demonstrates the user interface and its interactions.

to analyze how different TCP methods perform over different software versions;

[R3] Enable effective comparison of different TCP methods. The visualizations and interactions in the system should be flexible to effectively enable the comparison of multiple prioritization methods on a given software project;

[R4] Enable the analysis of how different TCP methods cover the source code. Source coverage is an important aspect of software testing. For this reason, it is important to understand the source coverage when executing the tests following the order given by different TCP methods. Furthermore, a software project is often organized in a hierarchical structure, and it is important to be able to understand how the prioritizations cover different of this hierarchy, such as classes and packages;

[R5] Support different analytical strategies and goals. The system's stakeholders include professional software testers as well as software engineering researchers. Different users have different goals and may use a diverse set of analytical sequences in the data exploration. Therefore, the system should adapt to the user's analytical flow.

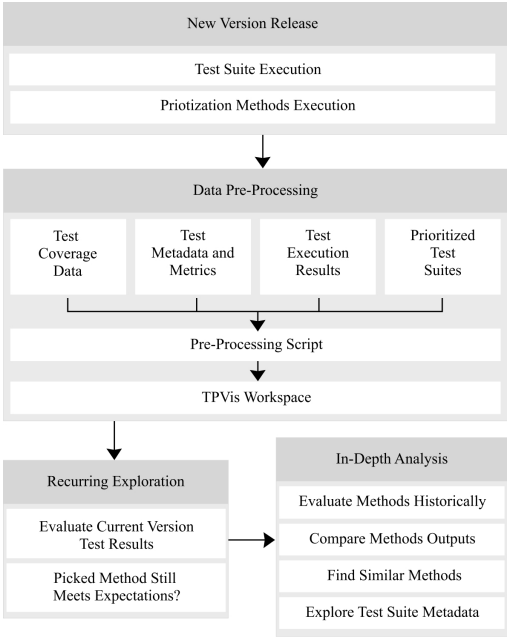


Fig. 2: An overview of the TPVis system and example of analyses it supports.

4.2. System overview

Figure 2 illustrates the general steps followed to use TPVis in a software testing scenario, for example in a CI/CD pipeline. After each candidate release, the user must gather the data required by TPVis, by executing the intended test suite as well as by running the TCP methods of interest. Following this, the user should execute a pre-processing script to gather all necessary information and build the TPVis workspace (dataset). We will provide detailed information about this pre-processing step in Section 4.3. Once the dataset is built, one can load it in TPVis. As described in Sections 4.4 and 5, TPVis' functionalities allow the user to perform multiple analytical tasks including the exploration of test suite metadata, test case results, evaluation of a single TCP method (which we call recurring explorations, since they are executed more often) as well as a comparative evaluation of different TCP methods (which we call in-depth analysis).

4.3. TPVis back-end

A TPVis dataset is called a workspace. Each workspace consists of a tree structure composed of nodes of three different types: folder, subject, and prioritization. Folders group nodes into meaningful categories; Subjects represent a software version and hold the test set (with the corresponding metadata and filed test set) for that specific software version; Prioritizations hold a list of ranked test IDs. One important usage of folder nodes is to group different prioritizations obtained by the same technique. This is useful for methods that employ a degree of randomness. Upon initialization, the TPVis back-end calculates performance metrics (e.g., APFD) for every prioritization in the workspace. These metrics are used in the user interface as guiding indicators for the analysis. The workspace is stored as a folder, with a *workspace.json* file in its root. This file holds the workspace metadata and the hierarchical tree structure itself. There are also folders for the test sets, prioritizations, and test coverage data. This data is referenced from the tree and lazily loaded by the application when needed. This structure is created by a script that gathers all the necessary data. This script can be executed in a CI/CD pipeline. The resulting workspace is then uploaded to a TPVis application server.

4.4. User interface

The TPVis interface (see Figure 1) is comprised of multiple widgets, offering a range of analytical tools designed to meet the design requirements listed in Section 4.1. The *prioritization tree* (Figure 1 (A)) enables navigating through the current workspace and filter nodes using regular expressions. The dropdown menu on top allows the customization of the drag-and-drop behavior when nodes containing multiple prioritizations are used: either select the best prioritization from the selected nodes or all the prioritization at once. A numerical summary (e.g., APFD) of the quality of the prioritizations in the node is presented on the right side of the pane. This helps guiding the user towards potentially interesting prioritizations. The *tools pane* (Figure 1 (B)) displays all the analytical tools and dashboard presets (pre-configured dashboards) available in our system. These are accessed via the two tabs (View Presets and View Tools on the left). The user can instantiate a tool by dragging the tool icon (Figure 1 (F)) to the dashboard (Figure 1 (C, D and E)), which is the main canvas of the system. The tool instances are positioned in a grid layout and can be resized by the user. Furthermore, these instances can also be repositioned and removed in the dashboard via the icons close to the tool name (top right corner of the tool viewport). The data and parameters for each tool can be accessed via the *data management overlay*, which can be opened by clicking in the **Manage data** button on the top left corner of each tool's viewport (Figure 1 (G)).

4.5. Analytical tools

This section describes the analytical tools implemented in TPVis. Table 2 summarizes each tool's expected inputs and its relationship to the project requirements.

Test Curve. This tool allows the user to select a faulty version and a set of prioritizations as input. It consists of a line chart that displays a line for each of the input prioritizations. Its x-axis represents the percentage of the test suite executed, and the y-axis represents the percentage of test failures detected. The more points the curve has located to the top left of this plot, the sooner failures are found in this prioritization. On the right of Figure 3, we can see an example of this tool being used to analyze a set of prioritizations. The orange and gray curves represent the prioritizations that find more failures sooner.

Table 2: Overview of available analytical tools in *TPVis* and how many faulty versions and prioritizations each can consider simultaneously.

Tool	Data Input		Fullfils
	Faults	Pri.	
Test Curve	1	n	**R1, R3**
Failure History	n	Ø	**R5**
Test Stack	1	Ø	**R1, R3**
Metric Box Plot	Ø	n	**R1, R2, R3**
Test Pos. Parallel Coordinates	1	n	**R1, R3**
Test Curve TSNE	1	n	**R3**
Metric Evolution Line	n	Ø	**R1, R2, R3**
Metric Evolution Heatmap	n	Ø	**R1, R2, R3**
Prioritization Matrix	1	2	**R1, R3**
Coverage Evolution	1	2	**R1, R3, R4**
Test Age Failure Analysis	n	Ø	**R5**
Table	1	1	**R1, R3, R4**

Failure History. This tool summarizes a set of software versions and the number of failed and passed tests in each of them as a bar chart. Figure 10 (b) presents an example where most faulty versions have few test failures.

Test Stack. This tool arranges the tests in a grid format, with prioritizations on the y-axis and tests, represented as thin rectangles, stacked side by side on the x-axis. Blue rectangles represent tests that passed, and the red ones represent tests that failed. This allows the identification of the failed test positions across the prioritizations without the issue of curve overlaps found in the Test Curve tool (while losing the sense of cumulativeness of failure detection). An example can be seen in Figure 1 (d).

Metric Box Plot. This tool consists of a box plot of a chosen metric (commonly the APFD) for the evaluation of test case prioritizations. Figure 11 shows an example. This tool allows the user to define how to group the prioritizations in each box. For example, it is possible to plot the APFD distribution of a single method across multiple software versions or plot multiple TCP methods in a single software version.

Test Position Parallel Coordinates. This tool consists of a parallel coordinates plot to compare test ranks across multiple prioritizations, similar to the approach presented by Gratzl et al. [19]. Each axis corresponds to a prioritization, and each line corresponds to a test. The line connects the rank of the corresponding test on the different prioritizations. The user can select one of the display options: show all tests, show only tests that change positions, or show only failures. To minimize visual clutter, we use the optimal leaf order algorithm [35], which sorts a set of elements (in this case, each prioritization) such that similar elements are placed next to each other. This is done via

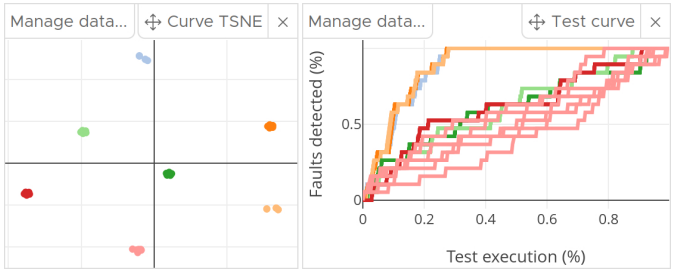


Fig. 3: Test Curve and Curve TSNE tools loaded with 6 prioritizations for 7 different methods. Note that the Curve TSNE tool clustered each method together.

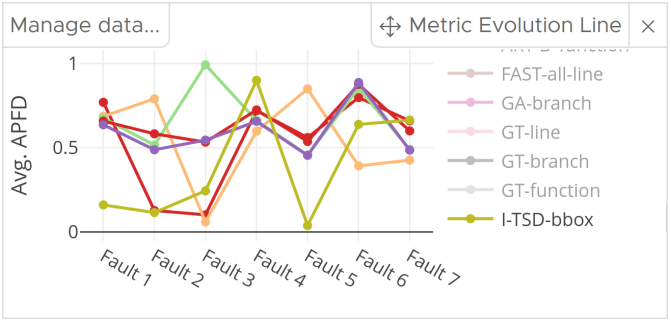


Fig. 4: Metric Evolution Line, filtered to display only 5 methods.

a binary hierarchical clustering of the elements and re-ordering the leaves of the corresponding dendrogram so that the sum of similarities of neighboring elements should be maximized.

Figure 8 (b) shows an example where groups of tests are permuted across different prioritizations. This suggests that the method has ranked these tests similarly. Notice also the presence of a prioritization that ranks a test 16(59) very differently from all the others. Some tests are missing from the plot because the tool is configured to display only tests that fail or change positions.

Test Curve TSNE. This tool complements the Test Curve. It consists of a scatterplot that depicts the similarity of the rankings corresponding to the test curves. Each point corresponds to one prioritization, and the closer the points, the more similar are the rankings in the prioritization. The points are positioned using the TSNE [36], a widely used dimensionality reduction technique. To measure the similarity between prioritizations, we used the Kendall tau distance [37] and used TSNE to position the corresponding points on the 2D plane according to this similarity. This allows for the identification of prioritizations that, despite different, may produce similar results in a specific software version (similar test curves). Identifying prioritizations with similar outcomes is essential, enabling stakeholders to choose the option that may be less costly to compute or faster to include in the project processes. Figure 3 highlights how this tool can be used to identify clusters. Additionally, users can select points via brushing, which filters the data in all the other tools instantiated.

Method Metric Evolution tools (Line and Heatmap). These tools enable the observation of average method metric evolution across software versions (also uses the buildNumber attribute of subjects). We introduced two tools for this purpose. One uses a line plot, and the other uses a matrix (where rows are ordered using the optimal leaf order algorithm). When the user's dataset has fewer methods, the line chart is preferred since perceiving position is more effortless than differentiating color. Figure 4 displays an example of the Metric Evolution Line, while Figure 10 (a) shows an example of the Metric Evolution Heatmap. Note that the heatmap version has the added benefit of grouping similarly-performing (regarding average metric) methods.

Prioritizations Matrix. The matrix tool allows the comparison of two distinct prioritizations. It consists of a scatter plot where the axes are the positions of the prioritizations. A dot represents each test, and the coordinates consist of the test position in both prioritizations. In Figure 1 (c), we can see an instance of the matrix tool demonstrating a case where both methods disagree greatly on the position of the tests but reach an agreement that there are two groups of tests that mostly should be together.

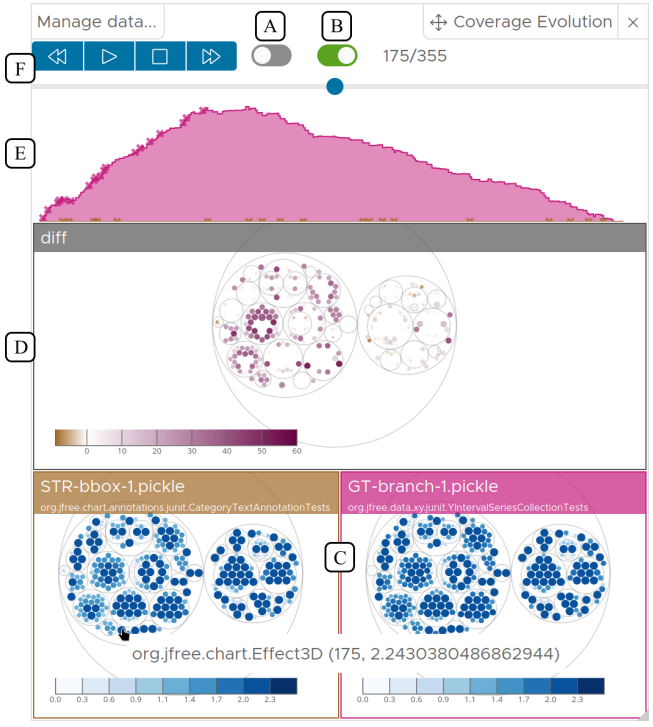


Fig. 5: Coverage Evolution tool with two loaded prioritizations. (A) Toggle between class/package mode; (B) Toggle between absolute or incremental mode; (C) Individual coverage inspection; (D) Difference view; (E) Prioritization coverage summary; (F) Playback controls.

Coverage Evolution. Code coverage plays a crucial role in numerous white-box test case prioritization strategies. This specialized tool was developed to assess how coverage evolves during the execution of prioritized test suites. The Coverage Evolution visualization can analyze a prioritization pair, utilizing a circle pack layout that mirrors the package signature associated with each class.

Figure 5 indicates the sections of the Coverage Evolution tool. The user can configure the tool to display the data at a class or package level (a). At the package level, the underlying classes are all aggregated together, while each class is displayed individually at the class level. It is also possible to change between absolute or incremental modes (b); the absolute mode

paints a class in one of the prioritizations if it was ever interacted by an executed test. In incremental mode, the color depends on how often the class was interacted with.

At the widget's lower section, highlighted as (c) in the figure, the user is presented with visual representations of the loaded prioritizations, including an indicator for the test currently in progress. Meanwhile, the upper circle pack section, indicated by (d), highlights the absolute difference in coverage between the prioritizations at the selected time interval. The visualizations at the bottom use a logarithmic scale to enhance readability. The class size and position inside the parent package reflect the count of interactions it has had with the tests at the end of the test suite execution.

Indicated as (e) is a line chart representing the sum of differences, helping the user know prioritization covers more of the software in different time frames. When using the absolute mode, the lines are displayed cumulatively and represent how much of the software was called by the tests executed. The user can also advance or rewind using the playback controls (f), allowing them to see how the software is being covered along the prioritization execution.

Additionally, one may map classes to software features, which can offer more semantic clarity by grouping related classes under a single component. To achieve this, the Data Management Overlay in this tool offers class alias functionality, allowing users to map specific classes to strings representing features. This feature also provides the flexibility to apply aliases in bulk or filter the class list. With this mapping, classes can be represented as features such as "users.create" and "users.onboarding".

The core purpose of this tool is to identify which parts of the software are being tested in which order in different prioritizations. Figure 6 compares a black-box and a white-box methods, STR and GT-Branch, respectively. Knowing that the greedy total algorithm picks tests based on their coverage of the software as a whole, a package more heavily prioritized by the STR algorithm indicates that the tests that test the package classes are more specific to those and have less coverage. Also, Figure 5 displays the same prioritizations, and, as expected, the GT-Branch method covers the software faster, as indicated by the line plot (Figure 5 (e)).

Test Age Failure Analysis. This tool does not directly relate to prioritizations and takes only software versions as inputs. It consists of a scatter plot (Figure 7 (a)), where the X-axis consists of the test creation date (attribute creation time in test metadata), and the Y-axis consists of the number of times the test failed. The test age failure analysis can help in the identification of old tests that fail frequently or keep failing in recent software versions. One may argue that tests that fit this criterion should be prioritized to the detriment of others.

Table. The Table tool (Figure 1 (e)) allows the users to inspect a test set and its test metadata. This metadata must be defined in the pre-processing step and is displayed in a dynamic table. Test IDs and ranks (given the prioritization selected in the *data management overlay*) are presented in the two first (fixed) columns. This tool is particularly useful for inspecting the metadata of prioritized tests, aiming to find patterns in the methods. Furthermore, the tool allows for sorting, filtering, and reordering columns, bringing flexibility to any analysis. Additionally, it can aid stakeholders in finding out that a method is heavily prioritizing a particular feature of the program or is substantially dependent on specific test metadata (e.g. code coverage).

4.6. View presets

Composing dashboards may become a repetitive task. With that in mind, three visualization presets, supporting three different tasks proposed by our domain collaborators, are available and can be accessed via the tool pane by clicking on "View Presets" (Figure 1 (b)). Selecting a preset will instantiate the corresponding tools in the dashboard. These three view presets are: (I) Historical view, intended to analyze how prioritizations have performed historically; (II) Algorithm comparison, to assist in analyzing how different algorithms compare in different aspects; and (III) Algorithm Inspectiong, to inspect how a specific prioritization method may behave in different scenarios. The content of each dashboard is detailed in Table 3.

Table 3: Tool composition of the view presets available.

Preset	Tools
Historical View	Metric Evolution Heatmap
	Test Age Failure Analysis
	Failure History
Algorithm Comparison	Curve TSNE
	Test Curve
	Box Plot
Algorithm Inspection	Curve TSNE
	Test Curve
	Metric Box Plot
	Table Tool

4.7. Implementation

TPVis was implemented following the frontend backend pattern. The backend was built in Java, using the Quarkus framework [38], which consists of endpoints to serve workspace data. The frontend was implemented in Javascript using Angular 16 [39], Clarity Design System [40] and Gridster.js [41]. The visualizations were all implemented using Plotly JS [42], D3 [43], and AG Grid [44]. Furthermore, Pyodide [45], a web-based Python interpreter, is also used for some functionalities. For the optimal leaf ordering algorithm, we used the Reorder.js's [46] implementation. To implement the Test Curve TSNE tool we used the Scipy [47] implementation of the Kendall tau distance and the Scikit Learn's [48] implementation of TSNE. For TSNE, we used all default parameters except for perplexity, which we used 3 (determined experimentally for the datasets used in our use cases). One of the main design goals was to structure the implementation to make it modular and, therefore, easy to extend. The implementation of TPVis is available at GitHub [1].

[1] https://github.com/vixe-cin-ufpe/TPVis

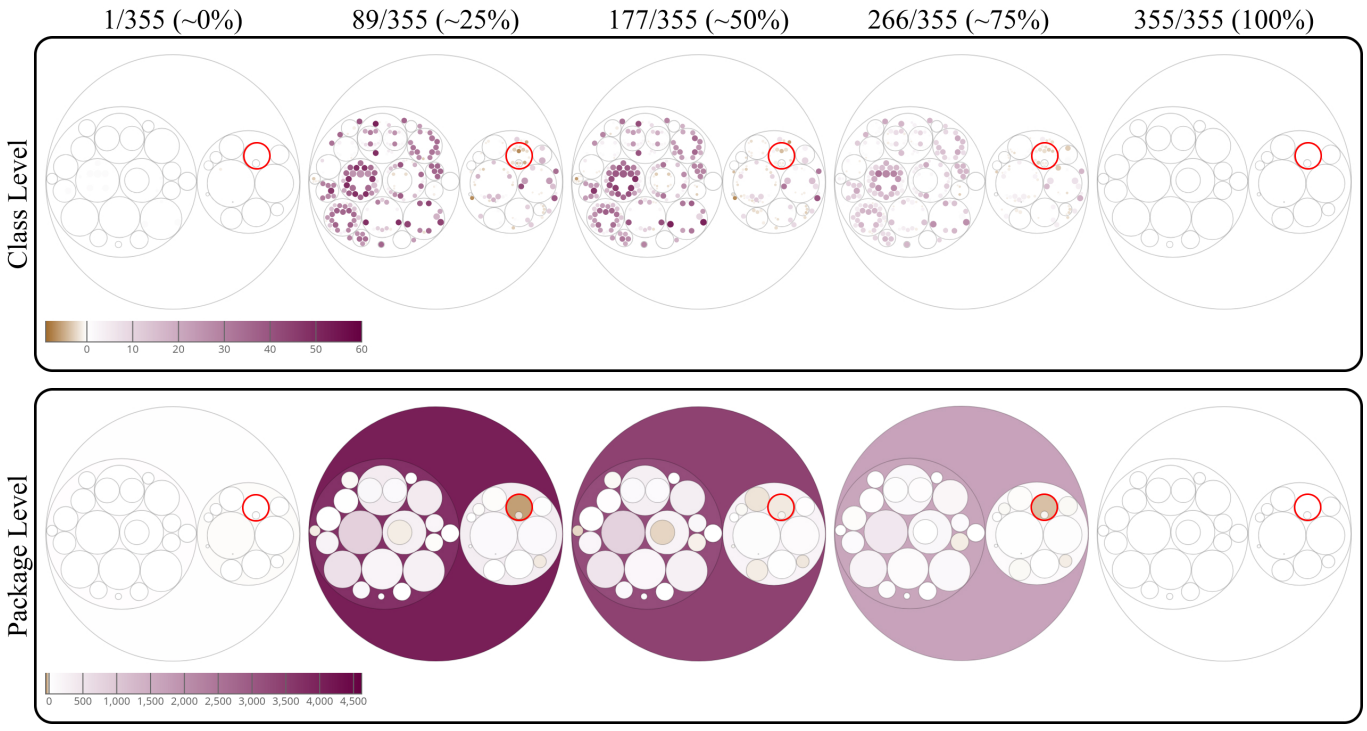


Fig. 6: Coverage Evolution tool with data from STR-BBOX and GT-Branch methods loaded. The package where STR-BBOX prioritizes most (despite GT-Branch leading in whole prioritization) is indicated in red.

5. Use cases

This section presents two use cases that showcase the main TPVis' capabilities. For this, we use the dataset provided by Miranda et al. [49], which we call the FAST dataset. This dataset consists of a series of prioritization results for different TCP methods applied to multiple test suites, including the ones in Defects4j [11]. Defects4j is a comprehensive database of reproducible bugs and their corresponding fixes of, as of the time of the submission of this paper, 17 Java projects. This framework simplifies access to the faulty versions and their solutions by leveraging version control and isolating the flawed versions and their fixes. To evaluate the system in a real-world context, we exclusively used the projects from Defects4j in the FAST dataset. Our final dataset comprises data from 37 distinct prioritization methods, each executed 50 times to account for their non-deterministic nature. In the use cases, we will analyze the results obtained by applying 7 different prioritization methods, being I-TSD, GA-Function, and all 5 FAST black-box variants (all, sqrt, log, one, and pw) to the Joda Time and Apache Commons Lang projects.

5.1. Assessing historically problematic tests

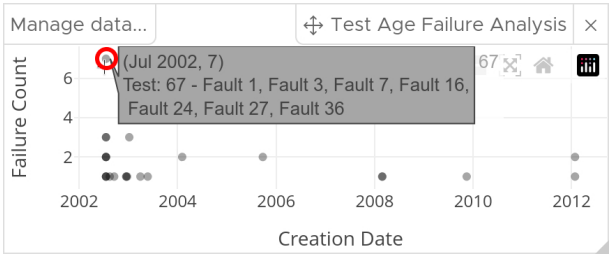
Apache commons-lang is a prevalent Java library that has gained wide adoption due to its range of utility functions, particularly for tasks such as string and number manipulation. The library was developed to address some shortcomings of the Java API, which did not provide adequate utilities to interact with its classes in the view of many developers at the time.

The Test Age Failure Analysis tool can be used to identify historically problematic tests. Figure 7 (a) shows that the test with ID 67 failed noticeably more than other tests despite being almost a decade old. If a test fails with a higher frequency, executing it first or finding a method that prioritizes it may be essential. Utilizing the Table tool filtering feature (Figure 7 (b)), we can identify that the problematic test is the NumberUtilsTest, which covers 34% of the software. This test validates the NumberUtils class, an essential library component that contains numerous utility methods for working with number-related classes in the Java API.

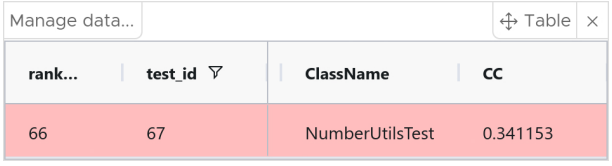
By adding only the faults that were caused by the test to the Metric Evolution Heatmap (Figure 7 (c)), we found that all the methods had the same APFD average for these versions. This can be expected because, as confirmed by the Failure History tool, they all share the same single failed test.

Both the I-TSD-bbox and GA-function methods perform similarly, with APFD values of 0.746 and 0.751, respectively. However, By utilizing the Metric Box Plot (Figure 8 (a)), we can observe that while the I-TSD algorithm can achieve better results than the GA-Function in some cases, It also has a higher variance. This means that the failing test had many different positions in the 50 iterations produced by the method. Additionally, the Position Parallel Coordinates (Figure 8 (b)) also reveal that the historically failing-only test varies between positions 26 and 28 in the GA-Function results.

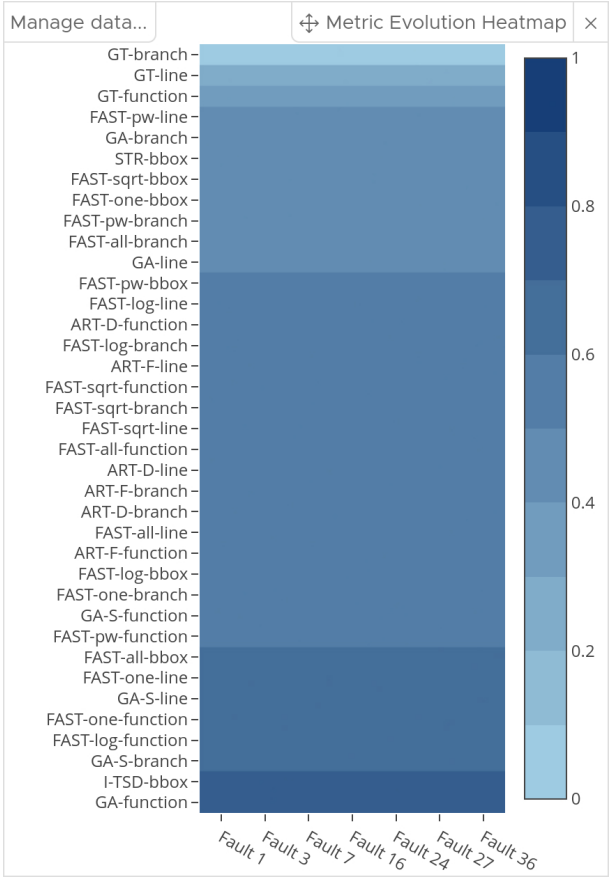
Another advantage of the GA algorithm is that it, due to its nature, achieves software coverage faster than I-TSD. This can be validated using the line plot in the Coverage Evolution tool (Figure 9). Finally, Considering the higher cost of executing the I-TSD algorithm and all the points discussed, the choice for GA-Function becomes evident.



(a) Test Age Failure Analysis indicating a single test failing in 7 different faulty versions

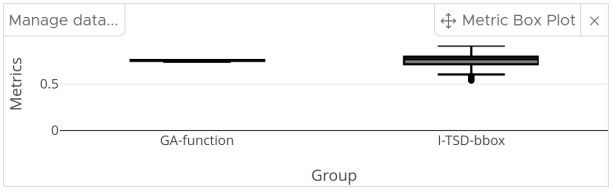


(b) Table tool used to identify the offending test.

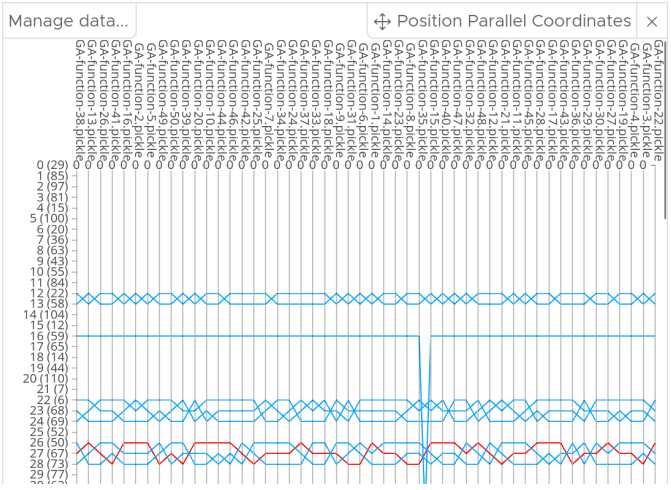


(c) Metric Evolution Heatmap comparing how the dataset methods perform in the affected versions.

Fig. 7: Identifying recurring failing test in Apache commons-lang project and suitable TCP methods to prioritize it.



(a) Metric Box Plot with all prioritizations for the analyzed methods.



(b) Position Parallel Coordinates for all prioritizations produced by GA-Function, showing that the failing test varies at most for 3 positions between each execution.

Fig. 8: Selecting between GA-Function and I-TSD methods to prioritize a recurring failing test in Apache Commons Lang.

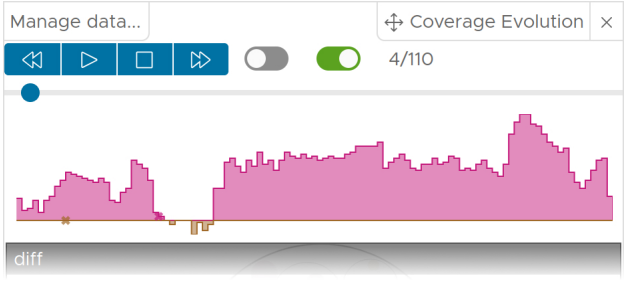


Fig. 9: A quick way to identify which prioritization covers a software faster is by utilizing the line plot available in the Coverage Evolution tool. In this case, GA (represented in pink) outperforms I-TSD, maintaining a higher coverage across most of the prioritization.

5.2. *Identifying an appropriate TCP method for Joda-Time*

Joda Time is a Java library that facilitates date and time manipulation. However, following the advent of the new java.time API (JSR-310), many developers have deemed the library obsolete. Nevertheless, the library is still utilized in numerous legacy systems and, therefore, remains maintained to this day.

Figure 10 shows the failure history preset loaded with all the failures in the Joda Time project. The data is loaded using the "drop globally" feature, which is presented when the user drags a node from the prioritization tree. The Failure History tool provides an overview of the test suite size and how frequently test failures occur. In this case, the project generally has approximately two failures, implying that methods have less chance of prioritizing a test that reveals a faulty version.

By inspecting the Metric Evolution Heatmap (Figure 10 (a)) we can see that some faults have slightly higher APFD metrics than others in most methods. These are the ones that have more failed tests (10, 12, and 22, as seen in Figure 10 (b)). Also, a closer inspection indicates that the Fast BBOX methods have outperformed the others in the latest versions.

To decide between these methods, we use the Metric Box Plot tool to visualize how effectively these methods perform across iterations since they are non-deterministic (Figure 11). Plotting the metrics for the methods in the latest 4 versions (F27, F26, F25, F24) shows that the fast-pw has the lowest met-

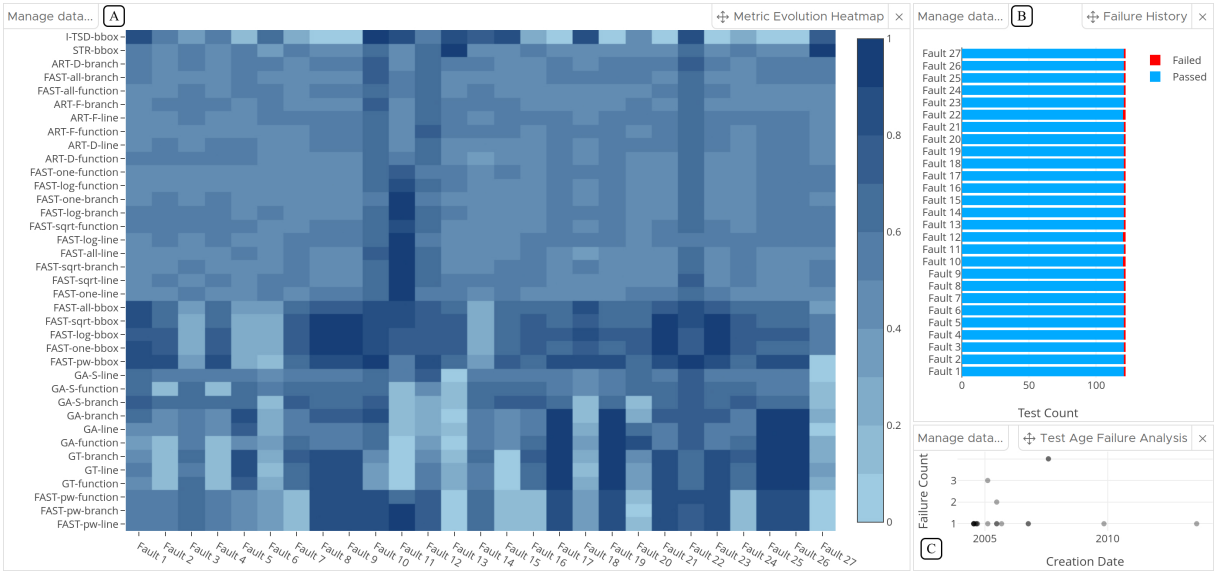
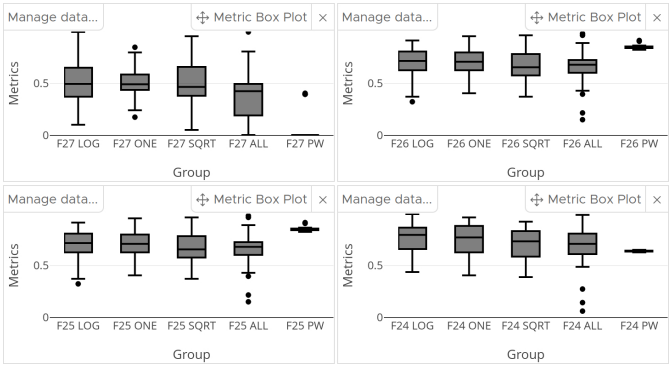
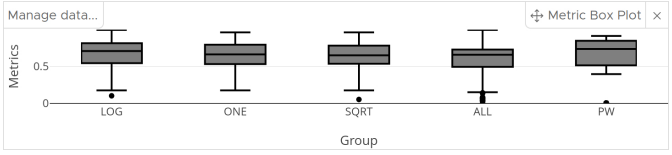


Fig. 10: The Historical Analysis Preset with Joda-Time project loaded. (A) Metric Evolution Heatmap with all project faults loaded; (B) Failure History tool, where, by hovering, it's possible to identify that faults 22, 12, and 10 had two failed tests while the rest had one; (C) Test Age Failure Analysis tool.



(a) Last 4 faults of the best-performing methods for the Joda-Time project



(b) Aggregate APFD metric for the best-performing methods for the Joda-Time project.

Fig. 11: Analyzing metrics of the best-performing methods for Joda-Time.

ric variance and fewest outliers. However, it also demonstrated consistent letdown in fault 27. Furthermore, by plotting these methods in those latter versions together, resulting in each box having 200 prioritizations instead of 50, it is observed that the fast-pw-bbox produces the best results with lower variance.

6. Discussion

Domain experts feedback. During the development process of TPVis, several meetings were held with specialists in software testing. The first specialist raised concerns regarding the core usability of the system. Specifically, he mentioned that in many cases the analysts would be required to re-create dashboards intended to perform specific analyses (since an earlier version was presented). To address this, we created the view presets. Another concern was regarding the ability to compare different methods easily without needing to drag and drop a single prioritization from each one. To address this, we implemented the "best prioritization for method" drag-and-drop behavior.

Both the first and second specialists identified a lack of historical visualization tools in the system. As a result, we implemented Metric Evolution (Line and Heatmap), Test Age Failure Analysis, and Failure History tools to address these concerns. This second specialist, who was from the TCP field, additionally provided feedback on the presentation of the workspace in the prioritization tree and how it wasn't trivial to know what each node was (requiring a mouse hover). We resolved this feedback by adding a "markAs" attribute to nodes to enable easier identification of different node types. This enables, for instance, a faulty version to be displayed with a bug icon instead of a generic folder icon. This specialist also suggested creating a tool to assess how prioritization covers different software parts. To address this, the Coverage Evolution tool was implemented. Once these recommendations were fulfilled, all domain experts involved could see the benefits of using such a tool in a production environment. Furthermore, all the experts agreed that the interface seemed intuitive and easy to use.

Computational Performance and Scalability. TPVis has been tested with a fairly large dataset consisting of 5 projects, 198 faulty versions, 37 prioritization methods, and 366.300 individual prioritizations. The largest test suite in our dataset is from the "math" project, comprising 384 tests (as a comparison, Luo et al. [50] used suites up to 122 tests in size in their empirical evaluation). Thanks to the lazy-loading approach adopted, there were no problems regarding the amount of prioritizations or the amount of methods in the workspace. However, when loading many prioritizations in a tool, it may take a while for them to be fetched from the Quarkus backend (which could be improved by parallelizing the requests). In our current implementation, it

takes around 2 seconds to load 100 prioritizations.

Some of the analytical tools, like the Coverage Evolution, perform a fair amount of processing while being used, and some, like the Test Position Parallel Coordinates, perform some heavy lifting to render initially. Web workers could be used to offload the processing from the main thread. The Curve TSNE tool has its processing written in Python (executed with Pyodide) and runs in a web worker. This validates that the web worker approach and the application is responsive when computing even hundreds of prioritizations in this tool.

Algorithms like TSNE and optimal leaf ordering may require expensive computational efforts for large datasets. For example, in our experiments, we observed a time of around 3 seconds to lay out 100 prioritizations with TSNE. While this was deemed acceptable by our domain collaborators, it is well-known that such delays could harm the user data analysis process [51]. We defer the optimization of our implementation to reduce these computational times to future work.

Visual Scalability. While our system is capable of tackling many real-world situations, some of the tools in TPVis are inherently limited with respect to their visual scalability. In some cases, these limitations can be overcome by the use of multiple views. For example, as shown in Figure 3, the Curve TSNE tool naturally complements the Test Curve in case visual clutter occurs. In our system, we use colors to represent the different TCP methods. While this clearly limits the number of methods that can be effectively loaded in our system at the same time (otherwise they would look the same), it is rarely the case that they all would need to be considered at the same time. We could remove this limitation by enabling the different methods to be hierarchically grouped in classes and, following the InfoVis mantra [52], enabling the user to inspect "details on demand". Finally, our Coverage Evolution tool can only monitor two prioritizations at a time. While the ability to monitor coverage evolution for multiple prioritizations within a single tool is desired, we believe the solution to this problem to be non-trivial and we believe it to be an interesting direction to future research.

Requirement of a pre-processing script. In order to implement TPVis in any project, a pre-processing script is necessary, although it is a straightforward task. However, the development process could be made much smoother with the development of an API. An API could make the representation of various node types in the TPVis workspace tree easier, streamlining the construction of the dataset file structure. To take it even further, plugins could be developed for widely used CI/CD pipeline tools, like Jenkins and Github Actions, as well as testing frameworks, which would eliminate the requirement for a pre-processing script in most scenarios.

7. Conclusion

We introduced TPVis, a comprehensive toolkit comprising 12 different tools designed to analyze test case prioritization data across multiple software versions and TCP methods. Our paper delves into the unique features of each tool and showcases two use cases that demonstrate how they can provide valuable insights intuitively. Furthermore, we demonstrate how TPVis can be seamlessly integrated into real-world scenarios and present the results of our analysis using actual prioritization methods and test data. In conclusion, TPVis offers a flexible and all-encompassing solution for prioritizing, analyzing, and inspecting test cases in varying software environments.

In the future, we aim to improve TPVis by investigating how to address some limitations of the current analytical tools. In particular, we intend to include different techniques to aggregate the larger number of faults and prioritizations loaded. Furthermore, we also intend to improve the system's responsiveness when dealing with large datasets. Finally, we are currently planning a formal user study in a production environment in collaboration with a software testing team. In this user study, we want to perform a comprehensive evaluation of TPVis overall usability, learnability and also to assess the long-term impact of the use of our system in a production environment. Finally, we also want to more closely evaluate and extend the Coverage Evolution tool (to support the comparison of more than two methods at a time) in terms of its effectiveness on the general problem of analyzing code coverage over the execution of a test suite.

8. Acknowledgments

We would like to thank the reviewers for their constructive comments and feedback. This work was supported by CNPq (311425/2023-2), CAPES (88887.683727/2022-00) and the research cooperation project between Motorola Mobility (a Lenovo Company) and CIn-UFPE.

References

[1] Harrold, MJ. Reduce, reuse, recycle, recover: Techniques for improved regression testing. In: 2009 IEEE International Conference on Software Maintenance. 2009, p. 5–5. doi:10.1109/ICSM.2009.5306347.

[2] Rothermel, G, Elbaum, S. Putting your best tests forward. IEEE Software 2003;20(5):74–77. URL: https://ieeexplore.ieee.org/document/1231157. doi:10.1109/MS.2003.1231157.

[3] Duggal, G, Suri, B. Understanding regression testing techniques. In: Proceedings of 2nd National Conference on Challenges and Opportunities in Information Technology; vol. 1. 2008, p. 8.

[4] Lou, Y, Chen, J, Zhang, L, Hao, D. Chapter one - a survey on regression test-case prioritization. vol. 113 of *Advances in Computers*. Elsevier; 2019, p. 1–46. URL: https://www.sciencedirect.com/science/article/pii/S0065245818300615. doi:https://doi.org/10.1016/bs.adcom.2018.10.001.

[5] Elbaum, S, Rothermel, G, Penix, J. Techniques for improving regression testing in continuous integration development environments. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2014; New York, NY, USA: Association for Computing Machinery. ISBN 9781450330565; 2014, p. 235–245. URL: https://doi.org/10.1145/2635868.2635910. doi:10.1145/2635868.2635910.

[6] Zhai, K, Jiang, B, Chan, W. Prioritizing test cases for regression testing of location-based services: Metrics, techniques, and case study. IEEE Transactions on Services Computing 2014;7(1):54–67. URL: https://ieeexplore.ieee.org/document/6375700. doi:10.1109/TSC.2012.40.

[7] Mukherjee, R, Patnaik, KS. A survey on different approaches for software test case prioritization. Journal of King Saud University - Computer and Information Sciences 2021;33(9):1041–1054. URL: https://www.sciencedirect.com/science/article/pii/S1319157818303616. doi:https://doi.org/10.1016/j.jksuci.2018.09.005.

[8] Khatibsyarbini, M, Isa, MA, Jawawi, DN, Tumeng, R. Test case prioritization approaches in regression testing: A systematic literature review. Information and Software Technology 2018;93:74–93. URL: https://www.sciencedirect.com/science/article/pii/S0950584916304888. doi:https://doi.org/10.1016/j.infsof.2017.08.014.

[9] Omri, S, Sinz, C. Learning to rank for test case prioritization. In: Proceedings of the 15th Workshop on Search-Based Software Testing. SBST '22; New York, NY, USA: Association for Computing Machinery. ISBN 9781450393188; 2023, p. 16–24. URL: https://doi.org/10.1145/3526072.3527525. doi:10.1145/3526072.3527525.

[10] Elbaum, S, Rothermel, G, Kanduri, S, Malishevsky, AG. Selecting a cost-effective test case prioritization technique. Software Quality Journal 2004;12(3):185–210. URL: https://doi.org/10.1023/B:SQJO.0000034708.84524.22. doi:10.1023/B:SQJO.0000034708.84524.22.

[11] Just, R, Jalali, D, Ernst, MD. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. ISSTA 2014; New York, NY, USA: Association for Computing Machinery. ISBN 9781450326452; 2014, p. 437–440. URL: https://doi.org/10.1145/2610384.2628055. doi:10.1145/2610384.2628055.

[12] Rothermel, G, Untch, R, Chu, C, Harrold, M. Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering 2001;27(10):929–948. URL: https://ieeexplore.ieee.org/document/962562. doi:10.1109/32.962562.

[13] Zhang, L, Hao, D, Zhang, L, Rothermel, G, Mei, H. Bridging the gap between the total and additional test-case prioritization strategies. In: 2013 35th International Conference on Software Engineering (ICSE). 2013, p. 192–201. URL: https://ieeexplore.ieee.org/document/65194. doi:10.1109/ICSE.2013.6606565.

[14] Li, Z, Harman, M, Hierons, RM. Search algorithms for regression test case prioritization. IEEE Transactions on Software Engineering 2007;33(4):225–237. URL: https://ieeexplore.ieee.org/document/4123325. doi:10.1109/TSE.2007.38.

[15] Bajaj, A, Sangwan, OP. A systematic literature review of test case prioritization using genetic algorithms. IEEE Access 2019;7:126355–126375. URL: https://ieeexplore.ieee.org/document/8819910. doi:10.1109/ACCESS.2019.2938260.

[16] Tonella, P, Avesani, P, Susi, A. Using the case-based ranking methodology for test case prioritization. In: 2006 22nd IEEE International Conference on Software Maintenance. 2006, p. 123–133. URL: https://ieeexplore.ieee.org/document/4021329. doi:10.1109/ICSM.2006.74.

[17] Busjaeger, B, Xie, T. Learning for test prioritization: an industrial case study. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2016; New York, NY, USA: Association for Computing Machinery. ISBN 9781450342186; 2016, p. 975–980. URL: https://doi.org/10.1145/2950290.2983954. doi:10.1145/2950290.2983954.

[18] Shi, C, Cui, W, Liu, S, Xu, P, Chen, W, Qu, H. Rankexplorer: Visualization of ranking changes in large time series data. IEEE Transactions on Visualization and Computer Graphics 2012;18(12):2669–2678. URL: https://ieeexplore.ieee.org/document/6327273. doi:10.1109/TVCG.2012.253.

[19] Gratzl, S, Lex, A, Gehlenborg, N, Pfister, H, Streit, M. Lineup: Visual analysis of multi-attribute rankings. IEEE Transactions on Visualization and Computer Graphics 2013;19(12):2277–2286. URL: https://ieeexplore.ieee.org/document/6634146. doi:10.1109/TVCG.2013.173.

[20] Mylavarapu, P, Yalcin, A, Gregg, X, Elmqvist, N. Ranked-list visualization: A graphical perception study. In: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems. CHI '19; New York, NY, USA: Association for Computing Machinery. ISBN 9781450359702; 2019, p. 1–12. URL: https://doi.org/10.1145/3290605.3300422. doi:10.1145/3290605.3300422.

[21] Strandberg, PE, Afzal, W, Sundmark, D. Decision making and visualizations based on test results. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM '18; New York, NY, USA: Association for Computing Machinery. ISBN 9781450358231; 2018,URL: https://doi.org/10.1145/3239235.3268921. doi:10.1145/3239235.3268921.

[22] Strandberg, PE, Afzal, W, Sundmark, D. Software test results exploration and visualization with continuous integration and nightly testing. International Journal on Software Tools for Technology Transfer 2022;24(2):261–285. URL: https://doi.org/10.1007/s10009-022-00647-1. doi:10.1007/s10009-022-00647-1.

[23] Cornelissen, B, van Deursen, A, Moonen, L, Zaidman, A. Visualizing testsuites to aid in software understanding. In: 11th European Conference on Software Maintenance and Reengineering (CSMR'07). 2007, p. 213–222. URL: https://ieeexplore.ieee.org/document/4145039. doi:10.1109/CSMR.2007.54.

[24] Hammad, M, Otoom, AF, Hammad, M, Al-Jawabreh, N, Abu Seini, R. Multiview visualization of software testing results. International Journal of Computing and Digital Systems 2020;9(1). URL: https://pdfs.semanticscholar.org/511d/9fb55dcbc971e6fbd5037eea9cbf0ea69e0f.pdf. doi:10.12785/ijcds/090105.

[25] Jones, JA, Harrold, MJ, Stasko, J. Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering. ICSE '02; New York, NY, USA: Association for Computing Machinery. ISBN 158113472X; 2002, p. 467–477. URL: https://doi.org/10.1145/581339.581397. doi:10.1145/581339.581397.

[26] Brandt, C, Zaidman, A. How does this new developer test fit in? a visualization to understand amplified test cases. In: 2022 Working Conference on Software Visualization (VISSOFT). 2022, p. 17–28. doi:10.1109/VISSOFT55257.2022.00011.

[27] Wahl, NJ. An overview of regression testing. SIGSOFT Softw Eng Notes 1999;24(1):69–73. URL: https://doi.org/10.1145/308769.308790. doi:10.1145/308769.308790.

[28] Leung, H, White, L. Insights into regression testing (software testing). In: Proceedings. Conference on Software Maintenance - 1989. 1989, p. 60–69. URL: https://ieeexplore.ieee.org/document/65194. doi:10.1109/ICSM.1989.65194.

[29] Feldt, R, Poulding, S, Clark, D, Yoo, S. Test set diameter: Quantifying the diversity of sets of test cases. In: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST). 2016, p. 223–233. doi:10.1109/ICST.2016.33.

[30] Ledru, Y, Petrenko, A, Boroday, S, Mandran, N. Prioritizing test cases with string distances. Automated Software Engineering 2012;19:65–95.

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

[32] Hao, D, Zhang, L, Zang, L, Wang, Y, Wu, X, Xie, T. To be optimal or not in test-case prioritization. IEEE Transactions on Software Engineering 2016;42(5):490–505. URL: https://ieeexplore.ieee.org/document/7314957. doi:10.1109/TSE.2015.2496939.

[33] Lv, J, Yin, B, Cai, KY. On the gain of measuring test case prioritization. In: 2013 IEEE 37th Annual Computer Software and Applications Conference. 2013, p. 627–632. URL: https://ieeexplore.ieee.org/document/6649891. doi:10.1109/COMPSAC.2013.101.

[34] Shahin, M, Ali Babar, M, Zhu, L. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. IEEE Access 2017;5:3909–3943. doi:10.1109/ACCESS.2017.2685629.

[35] Bar-Joseph, Z, Gifford, DK, Jaakkola, TS. Fast optimal leaf ordering for hierarchical clustering . Bioinformatics 2001;17(suppl_1):S22–S29. URL: https://doi.org/10.1093/bioinformatics/17.suppl_1.S22. doi:10.1093/bioinformatics/17.suppl_1.S22.

[36] Van der Maaten, L, Hinton, G. Visualizing data using t-sne. Journal of machine learning research 2008;9(11).

[37] Kendall, MG. A new measure of rank correlation. Biometrika 1938;30(1-2):81–93.

[38] Quarkus framework. 2024. URL: https://quarkus.io.

[39] Angular framework. 2024. URL: https://angular.io.

[40] Clarity design system. 2024. URL: https://clarity.design.

[41] Gridster.js. 2024. URL: https://dsmorse.github.io/gridster.js.

[42] Plotly js. 2024. URL: https://plotly.com/javascript.

[43] D3. 2024. URL: https://d3js.org.

[44] Ag grid. 2024. URL: https://ag-grid.com/angular-data-grid.

[45] Pyodide. 2024. URL: https://pyodide.org.

[46] Reorder.js. 2024. URL: https://github.com/jdfekete/reorder.js/.

[47] Virtanen, P, Gommers, R, Oliphant, TE, Haberland, M, Reddy, T, Cournapeau, D, et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods 2020;17:261–272. doi:10.1038/s41592-019-0686-2.

[48] Pedregosa, F, Varoquaux, G, Gramfort, A, Michel, V, Thirion, B, Grisel, O, et al. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 2011;12:2825–2830.

[49] Miranda, B, Cruciani, E, Verdecchia, R, Bertolino, A. Fast approaches to scalable similarity-based test case prioritization. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). 2018, p. 222–232. URL: https://ieeexplore.ieee.org/document/8453081. doi:10.1145/3180155.3180210.

[50] Luo, Q, Moran, K, Poshyvanyk, D. 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; New York, NY, USA: Association for Computing Machinery. ISBN 9781450342186; 2016, p. 559–570. URL: https://doi.org/10.1145/2950290.2950344. doi:10.1145/2950290.2950344.

[51] Liu, Z, Heer, J. The effects of interactive latency on exploratory visual analysis. IEEE transactions on visualization and computer graphics 2014;20(12):2122–2131.

[52] Shneiderman, B. The eyes have it: A task by data type taxonomy for information visualizations. In: BEDERSON, BB, SHNEIDERMAN, B, editors. The Craft of Information Visualization. Interactive Technologies; San Francisco: Morgan Kaufmann. ISBN 978-1-55860-915-0; 2003, p. 364–371. URL: https://www.sciencedirect.com/science/article/pii/B9781558609150500469. doi:https://doi.org/10.1016/B978-155860915-0/50046-9.