FaST: A linear time stack trace alignment heuristic for crash report deduplication

Irving Muller Rodrigues
Polytechnique Montreal
Montreal, Canada
irving.muller-rodrigues@polymtl.ca

Daniel Aloise
Polytechnique Montreal
Montreal, Canada
daniel.aloise@polymtl.ca

Eraldo Rezende Fernandes
Leuphana University of Lüneburg
Lüneburg, Germany
eraldo.fernandes@leuphana.de

ABSTRACT

In software projects, applications are often monitored by systems that automatically identify crashes, collect their information into reports, and submit them to developers. Especially in popular applications, such systems tend to generate a large number of crash reports in which a significant portion of them are duplicate. Due to this high submission volume, in practice, the crash report deduplication is supported by devising automatic systems whose efficiency is a critical constraint. In this paper, we focus on improving deduplication system throughput by speeding up the stack trace comparison. In contrast to the state-of-the-art techniques, we propose FaST, a novel sequence alignment method that computes the similarity score between two stack traces in linear time. Our method independently aligns identical frames in two stack traces by means of a simple alignment heuristic. We evaluate FaST and five competing methods on four datasets from open-source projects using ranking and binary metrics. Despite its simplicity, FaST consistently achieves state-of-the-art performance regarding all metrics considered. Moreover, our experiments confirm that FaST is substantially more efficient than methods based on optimal sequence alignment.

CCS CONCEPTS

• **Software and its engineering** → *Software creation and management*; *Software maintenance tools*; • **Applied computing**;

KEYWORDS

Duplicate Crash Report, Crash Report Deduplication, Duplicate Crash Report Detection, Automatic Crash Reporting, Stack Trace Similarity

ACM Reference Format:
Irving Muller Rodrigues, Daniel Aloise, and Eraldo Rezende Fernandes. 2022. *FaST*: A linear time stack trace alignment heuristic for crash report deduplication. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524842.3527951

MSR '22, May 23–24, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9303-4/22/05…$15.00
https://doi.org/10.1145/3524842.3527951

```
1  Crashed when I opened a website and clicked on 'flash version'.
2
3  Architecture: amd64
4  Date: Tue Jun 19 20:16:04 2007
5  DistroRelease: Ubuntu 7.10
6  ExecutablePath: /usr/bin/gnash
7  NonfreeKernelModules: vmnet vmmon cdrom
8  Package: gnash 0.8.0~cvs20070611.1016-1ubuntu2
9  PackageArchitecture: amd64
10 ProcCwd: /home/martin
11 SourcePackage: gnash
12 UserGroups: adm admin audio cdrom dialout dip floppy video
13 Title: gnash crashed with SIGSEGV in std::_Rb_tree::erase()
14
15 #0  0x027540aea in std::_Rb_tree::erase() at /usr/include/c++/4.1/bits/stl_tree.h:692
16 #1  0x027540b78 in std::_Rb_tree::erase() at /usr/include/c++/4.1/bits/stl_tree.h:1215
17 #2  0x02753f0c5 in movie_root::remove_key_listener() at /usr/include/c++/4.1/bits/stl_set.h:387
18 #3  0x02758100d in ~button_character_instance() at button_character_instance.cpp:280
19 #4  0x02753f0d3 in movie_root::remove_key_listener() at /usr/include/boost/intrusive_ptr.hpp:83
20 #5  0x02758100d in ~button_character_instance() at button_character_instance.cpp:280
21 #6  0x02753f0d3 in movie_root::remove_key_listener() at /usr/include/boost/intrusive_ptr.hpp:83
22 #7  0x02758100d in ~button_character_instance() at button_character_instance.cpp:280
```

Figure 1: Crash report example.

1 INTRODUCTION

To reduce user dependence in bug reporting and collect more data about errors, many software projects use *automated crash reporting systems* to monitor application executions. When target systems crash, such tools are invoked to gather relevant information about the failures and send it to backend systems [7].

The submitted information about a software error is grouped in a document called the *crash report*. A shortcoming of automated crash reporting systems is that they tend to rapidly increase the number of duplicate crash reports, that is, reports associated with the same failure. Therefore, it becomes vital to automate deduplication when such tools are employed [15]. In the literature, such a task is denoted *crash report deduplication*, being also referred to as *duplicate crash report detection* or *crash report bucketing* [5].

In Figure 1, we depict an example of a crash report. Such documents may include the failure descriptions provided by users (**lines 1 and 13**) and environment information (**lines 5–12**). Additionally, crash reports contain stack traces (**lines 15–22**), one of the most valuable information source for bug fixing [25]. A stack trace is a sequence of *frames* in which the first frame corresponds to the topmost element in the application's call stack at the moment a crash occurs. The subsequent frames represent subroutines waiting for the execution of the previous frames near to the top. As shown in Figure 1, stack traces can contain multiple information about the frames (e.g., the source file name). Inspired by previous works [4, 7, 23, 24], this paper focuses on the positions and subroutine names of the frames. Moreover, to compare whether two frames are identical or not, we consider subroutine names as frame identifiers (shortly, *frame ids*).

In the literature, a prevalent assumption is that crash reports are more likely to be duplicate when their stack traces are similar. Thus, the majority of techniques address crash report deduplication

by comparing stack traces. For instance, *TraceSim* [23], one of the current state-of-the-art (SOTA) methods in crash report deduplication, measures the similarity of two stack traces by computing a weighted version of the optimal global alignment score [21] between them. In real environments, hundreds or even thousands of crash reports are submitted every day [5]. Each one of them must be analyzed by deduplication systems to identify whether they are duplicate in very large repositories of submitted reports. Hence, due to this high volume of data, deduplication systems must be implemented observing feasible throughput. A simple strategy to improve deduplication system performances is to speed up the similarity measurement of stack traces. In the literature of crash report deduplication, some methods can efficiently compare two stack traces in $O(m+n)$, where m and n are their lengths. However, such methods are significantly less effective than TraceSim, that computes the stack trace similarity score in $O(nm)$.

The inefficiency of methods based on optimal sequence alignment, including TraceSim, are mainly caused by their search for *optimal* alignments. In order to guarantee optimality, these methods iteratively compute a dynamic programming matrix using recursive functions. Furthermore, the found alignment must preserve the sequence order which makes challenging to independently compare subsets of subroutines in stack traces. Leveraging the removal of the optimality requirement and the order constraint, one can develop efficient heuristic algorithms that find near-optimal alignments. It is worthy to mention here that the final task objective does not consist in finding the optimal sequence alignment, but rather computing similarity scores that are effective to group duplicate reports.

Inspired by this idea, we propose *FaST*, a Fast Stack Trace alignment method for crash report deduplication. In *FaST*, the sequence alignment is produced by individually aligning the frames of each unique identifier in the stack traces. Since stack traces of duplicate reports are expected to contain subroutines in similar *absolute* positions [23], we argue that similarity scores can be fairly captured by directly comparing overlaps or missing frames of each individual subroutine. Instead of optimally aligning frames, we employ a simple alignment heuristic: given the frames of each distinct indentifier, *FaST* iteratively matches the two closest ones to the top positions. Such heuristic is based on the rationale that frames near the topmost position should be prioritized for alignment over those in the bottom, since they are usually more relevant for the deduplication [7, 23, 25]. In cases where frames of an identifier are only available in one of the stack traces, *FaST* aligns such remaining frames to special structures, called *gaps*. After finding the alignment, the similarity score is computed considering two important pieces of information regarding a subroutine: its position and its global frequency [23]. Due to its simplified alignment algorithm, *FaST* can compare stack traces in $O(n+m)$, i.e., linear time on the length of the two sequences.

We experimentally evaluate the efficiency and effectiveness of *FaST* by means of the methodology proposed by Rodrigues et al. [23]. We compare *FaST* with SOTA systems and strong baselines on four different datasets from the following open-source projects: Ubuntu, Eclipse, Netbeans, and Gnome. In our experiments, *FaST* consistently achieves similar or significantly superior performance in terms of effectiveness when compared with its competing methods. Moreover, as expected, we observe that *FaST* is considerable faster than optimal sequence alignment methods. It is important to highlight that we provide the source code of the evaluation framework and methods online[1]. The main contributions of this paper are summarized as follows:

(1) We propose a novel stack trace alignment method with a linear time complexity for crash stack deduplication.
(2) We show that a simple alignment heuristic can be as effective for deduplication as techniques that find optimal global alignments.
(3) Our proposed method achieves state-of-the-art performances on all the considered datasets despite its simplicity.

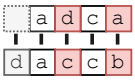


Figure 2: Example of a global alignment between the stack traces adca and daccb. Matches, mismatches and gaps are represented by white, red, and gray, respectively.

2 FAST STACK TRACE ALIGNMENT METHOD FOR CRASH REPORT DEDUPLICATION

As mentioned in the introduction, studies in the literature have addressed crash report deduplication by measuring stack trace similarity based on *optimal global alignment*. In the context of such problem, the term *global* means that frames are lined along the entire lengths of the compared stack traces.

In Figure 2, we depict an example of global alignment between two toy stack traces adca and daccb. As shown in the figure, there are three types of possible alignments: *match*, *mismatch*, and *gap*. A match occurs when two *identical* frames are aligned (e.g., the alignment between two frames a). Conversely, a mismatch arises when two *distinct* frames are aligned (e.g., frames a and b highlighted in red). The third type of alignment corresponds to lining up a frame to a *gap* (e.g., frames d and the cubes with dashed border). A gap represents an insertion/deletion operation performed in a sequence. A global alignment is only valid if the original sequence can be restored by removing the gaps, i.e., the sequence order cannot be altered.

There are many distinct ways to align two stack traces end-to-end. In order to find the best global alignment, a scoring scheme is defined to assign a value to each element alignment. Following such scheme, the score of the entire sequence alignment is equal to the sum of matches values subtracted by the values of mismatch and gap alignments. The *optimal global alignment* consists in finding an alignment between two sequences for which the score alignment is maximum. Given two sequences with lengths n and m, the optimal alignment can be found in $O(nm)$ time with the Needleman–Wunsch (NW) algorithm [21].

In this section, we present *FaST*, a novel sequence alignment method that effectively aligns two stack traces in $O(n+m)$ time. To achieve such complexity, *FaST* relaxes the optimal global alignment problem, allowing the computation of sub-optimal and non-ordered alignments.

[1] https://github.com/irving-muller/FaST

2.1 Similarity Algorithm

Similar to the optimal global alignment problem, *FaST* compares two stack traces based on the overlaps (represented by matches) and differences (captured by mismatches and gaps) between them. First, matches are performed by successively lining up the topmost unaligned frames with the same id. Each match increases the similarity score based on two factors: frame importance and the position discrepancy between the matched frames. The former depends on position and subroutine global frequency (two key frame features) and the latter alleviates the impact of poor matches performed by the heuristic alignment. After performing match alignments, all unmatched frames are aligned to gaps due to empirical evidences that gaps are more adequate than mismatches for crash report deduplication [23]. Each gap alignment penalizes the similarity score also based on the frame importance. Following previous works [7, 19, 23], the similarity score is normalized to be in range [-1.0, 1.0].

In Algorithm 1, we present the pseudo-code of *FaST* algorithm to compute the similarity score for two stack traces. The functions match$(\cdot)$ and gap$(\cdot)$ compute the values of a match and gap alignments, respectively, while w$(\cdot)$ returns a real number, called *frame weight*, that indicates the frame importance for the deduplication. Further details about such functions are provided in Section 2.2. In Figure 3, we depict an example of this algorithm execution on two toy stack traces.

As input, the similarity algorithm receives two lists sorted by frame id and position in ascending order. After sorting, each frame in Q and C are represented as f_p where f is its id and p is its position in the original stack trace. Basically, the sorting operation is analogous to split the frames in the stack trace and group them by their identifiers. It is worthy mentioning that such sort is executed only once right after stack trace creation. Thus, we consider that its time complexity is amortized given: (i) the relative short lengths of stack traces[2]; and (ii) the amount of comparisons performed to a given stack trace in real applications of crash report deduplication is much larger than $O(n \log n)$.

During initialization, the algorithm creates two pointers that refer to the beginning of each list. Each pointer represents the next available frame in a list for the alignment. In the example presented in Figure 3b, the pointers are illustrated by i and j and they point to the first frames of the sorted list Q and C, respectively. As first step, *FaST* compares the frame ids pointed by i and j, i.e., the first elements within Q and C. Since the identifiers are the same, a_1 and a_2 are matched. Then, i and j are moved to the next elements in the lists – a_4 and b_5, respectively. The result of this procedure is depicted in Figure 3c. In the next step, i and j refers to different subroutines. Since frames are sorted by the subroutine identifiers and a is smaller than b, this means there is no other available frame with id = a in the list C. Therefore, as illustrated in Figure 3d, a_4 is aligned to a gap and i is jumped to the subsequent available frame in Q. The algorithm proceeds by sequentially comparing the frames pointed by i and j. Match alignment is performed when frames share the same ids, otherwise the frame with the smallest identifier is aligned to a gap. The pointers are moved to the next available element in the sorted lists when the pointed frames are aligned. If the algorithm reaches the end of a list, then the remaining frames within the other one are aligned to gaps. Regarding our example, we depict the final alignment between Q and C found by such algorithm in Figure 3e.

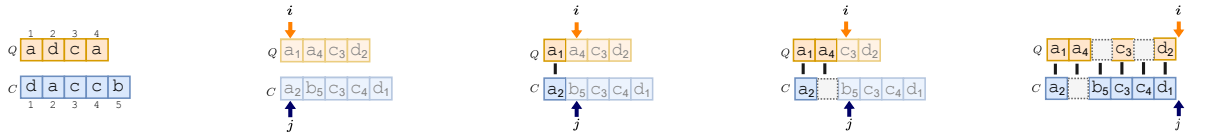
A limitation of *FaST*'s algorithm is that it does not directly penalize order inversions. As depicted in Figure 3e, even though the two first frames in Q and C are in inverse order, *FaST* still performs two matches between these frames. On the other hand, as shown in Figure 2, the NW algorithm penalizes this inversion by only matching the frames `a` and performing a mismatch and a gap alignment regarding the frames `d`. Nevertheless, in *FaST*, the higher is the position difference between two matched frames, the lower is their match value (further details in Section 2.2). Thus, if two frames have their relative order inverted, at least one of them will be in different positions in the two sequences, and this will be penalized

[2]In our datasets, 98% of the stack traces are shorter than 126 subroutines.

Algorithm 1: *FaST* pseudo-code

Input: Q and C: lists of frames of two stack traces that are sorted by frame ids and positions.
Output: Normalized similarity between Q and C.

```
1  sim ← 0.0
2  i ← 1
3  j ← 1
4  while i < length(Q) and j < length(C) do
       // q and u are the id and position of Q[i]
5      q_u ← Q[i]
       // c and v are the id and position of C[j]
6      c_v ← C[j]
7      if q == c then
           // Match alignment.
8          sim += match(q_u, c_v)
9          i += 1
10         j += 1
11     else if q < c then
           // Q[i] is aligned to a gap.
12         sim −= gap(q_u)
13         i += 1
14     else
           // C[j] is aligned to a gap.
15         sim −= gap(c_v)
16         j += 1
   // Align remaining frames in Q or C to gaps
17 while i < length(Q) do
18     sim −= gap(Q[i])
19     i += 1
20 while j < length(C) do
21     sim −= gap(C[j])
22     j += 1
   // Normalize the similarity score
23 return sim / (Σ_{q_u∈Q} w(q_u) + Σ_{c_v∈C} w(c_v))
```



(a) Original stack traces. (b) Input and initialization. (c) End of step 1. (d) End of step 2. (e) Final alignment.

Figure 3: An example of *FaST*'s alignment algorithm.

by our method. Furthermore, order inversions are not usual in stack traces, since they imply indirect recursions.

As mentioned earlier, the score of the final alignment is computed based on the chosen scoring scheme – functions match(·), gap(·), and w(·) – and it is normalized to be in a fixed interval – **line 23** in Algorithm 1. These algorithm aspects are described in details in the remainder of this section.

2.2 Scoring Scheme

In *FaST*, values of each match and gap alignments are computed using a scoring scheme similar to the one proposed by Rodrigues et al. [23]. In this scoring scheme, weights are assigned to each frame in the stack traces. A weight captures the importance of a frame f_p for the deduplication and it is computed as follows:

$$\mathrm{w}(f_p) = \frac{1}{p^{\alpha}} \times e^{-\beta \frac{\mathrm{df}(f)}{|S|}}, \tag{1}$$

where $|S|$ is the total number of stack traces in a repository S, and $\mathrm{df}(\cdot)$ is the number of stack traces in S that contain at least a subroutine identifier equal to f. The first component of (1) assigns higher values to positions closer to the top since such positions tend to be more related to the failure. The second one depends on the rarity of an id among the stack traces in the dataset. Frequent subroutines are usually ordinary operations in a system (e.g, logging and error-handling) and, thus, they are likely unrelated to the crash cause. Therefore, the more frequent the id of a frame is, the lower its weight should be. In Equation 1, similar to a logical *AND*, these two components are multiplied to consider a frame irrelevant for the deduplication when either its position is close to the bottom or its subroutine is frequent. Finally, $\alpha \in \mathbb{R}_{>0}$ and $\beta \in \mathbb{R}_{>0}$ are parameters that control the impact of the frame position and subroutine rarity on the weight value, respectively.

Following the original scheme, the gap alignment value is equal to the weight of a frame f_p aligned to a gap:

$$\mathrm{gap}(f_p) = \mathrm{w}(f_p). \tag{2}$$

However, unlike Rodrigues et al. [23] that employ the maximum weight between two matched frames q_u and c_v, we calculate the match value by means of the sum of these weights:

$$\mathrm{match}(q_u, c_v) = (\mathrm{w}(q_u) + \mathrm{w}(c_v)) \times \mathrm{diff}(u, v), \tag{3}$$

where the function $\mathrm{diff}(\cdot)$ is defined as:

$$\mathrm{diff}(u, v) = e^{-\gamma |u-v|}.$$

The parameter $\gamma \in \mathbb{R}_{>0}$ regulates the impact of the position difference on the function output. Since a common assumption is that same subroutines appear in closer positions in stack traces of duplicate crash reports, $\mathrm{diff}(\cdot)$ reduces the match value based on the position discrepancy of the matched frames.

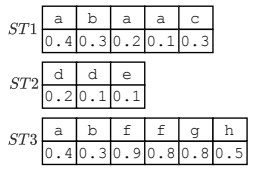


Figure 4: Normalization example.

2.3 Normalization

After aligning the stack traces, the alignment score is computed as the sum of the match values minus the sum of each gap alignment value. However, such score is not directly used for the deduplication since it can degrade the method effectiveness [23]. For instance, three stack traces $ST1$, $ST2$, and $ST3$ are depicted in Figure 4. Frame weights are represented by the real numbers below each subroutine identifier. In this example, the alignment scores are -1.7 and -2.2 when $ST1$ is compared with $ST2$ and $ST3$, respectively. However, this is unreasonable because $ST1$ is completely different of $ST2$ while the two topmost frames of $ST1$ and $ST3$ are overlapped. Such contradictory scores occurs because the alignment score is highly dependent on the frame weight values. Therefore, in order to mitigate such issue, the similarity scores are normalized based on the frame weights [7, 23].

Considering the definitions of gap and match (Equations 2 and 3), we can normalize the similarity score to be within the interval $[-1.0, 1.0]$ by simply dividing the alignment score by the sum of frame weights in the two stack traces (**line 23** in Algorithm 1). For instance, the sum of weights are 1.3, 0.4, and 3.7 in $ST1$, $ST2$, and $ST3$, respectively. Thus, the similarity score by comparing $S1$ with $S2$ and $S3$ is $\frac{-1.7}{1.3+0.4} = -1.0$ and $\frac{-2.2}{1.3+3.7} = -0.44$, respectively.

3 RELATED WORKS

In this section, we focus on studies that address crash report deduplication by means of stack trace similarity.

Modani et al. [19] proposed a *prefix match algorithm* in which the similarity is proportional to the length of the longest common prefix between two stack traces.

Methods based on the popular TF-IDF approach (Term Frequency – Inverse Document Frequency) [5, 16, 24] have also been applied to crash report deduplication. Lerch and Mezini [16] and Campbell et al. [5] employed the TF-IDF-based score function from Lucene

library[3] to measure the stack trace similarity. Sabor et al. [24] proposed the *DURFEX* technique, which uses only the package name of the subroutines, to compare two stack traces using the cosine similarity of their vector representations. One important drawback of these techniques is that they ignore a valuable piece of information: the position of the frames within the stack trace.

Some studies [4, 7, 23] proposed variantions of the NW algorithm to measure the similarity between two stack traces. In the technique designed by Brodie et al. [4], while mismatch and gap alignment values are constant, the match values are computed based on the rarity and position of the matched subroutines. On the other hand, Dang et al. [7] proposed a method, called PDM, in which match values depend only on the frame positions and the alignment score is not penalized by mismatches and gap alignments. Moreover, PDM contains parameters that regulate the impact of position information on the optimal solution. More recently, Rodrigues et al. [23] proposed TraceSim, a method for crash report deduplication that have outperformed previous methods from the literature. TraceSim computes match, mismatch, and gap values based on both the position and the global frequency (considering a large database) of a subroutine. To improve method flexibility over different data distributions, parameters control the weight of these two factors on the final similarity.

In order to improve efficiency without degrading the effectiveness of methods based on NW algorithm, Moroo et al. [20] proposed a reraking model, called PartyCrasher, that combines information retrieval techniques and PDM. First, PartyCrasher selects the top-k most similar candidates to a query by means of the score function designed by Lucene. Then, PDM is employed to compute the similarity between the selected candidates and the query. Finally, the final similarity score is a weighted harmonic mean of the similarities measured by Lucene's score function and PDM.

Edit distance is equivalent to optimal global alignment [26] and have been employed by two studies for crash report deduplication. Bartz et al. [1] proposed a logistic regression model based on the edit distance between two stack traces and some categorical features within the crash reports. To compute the edit distance, Bartz et al. [1] use the following information regarding a frame in the stack trace: the subroutine, its offset, and its module. Dhaliwal et al. [9] proposed to group the crash reports by the subroutine in the topmost position. Each group is then reorganized in subgroups based on the edit distance between its stack traces. These two techniques have the same efficiency issues that are present in techniques based on global alignment.

Khvorov et al. [15] proposed a siamese deep learning model, called S3M, for comparing two stack traces. A Long Short-Term Memory (LSTM) independently encodes the stack traces as fixed-length vectors. Then, a multilayer perceptron (MLP) computes the similarity between two stack traces based on their vector representations.

In Table 1, we present the time complexities of techniques to compute the similarity of two stack traces. Prefix match and methods based on TF-IDF run in linear time. However, they are less effective than more computationally expensive methods. Prefix match is highly affected by negligible differences in subroutine positions, while TF-IDF techniques ignore positional information altogether. Regarding S3M, considering that representations of stack traces are computed once and stored in a database, the amortized time complexity of such model is $O(dh)$, where d and h are the sizes of the input and hidden layer, respectively. In practice, d and h are comparable or even larger than the stack trace lengths, e.g., $i = 600$ and $d = 300$ in S3M while we found that, in our experimental setup, 98% of the stack trace contains less than 130 frames. Besides its quadratic complexity, other limitation of S3M is that it requires a considerable volume of labeled data for training. However, such type of data is not always available in industry projects. Unlike S3M, the parameters of *FaST* can be manually set by a specialist.

Table 1: Time complexity of crash report deduplication methods. The lengths of two stack traces is denoted n and m.

Method	Time complexity
Prefix Match [19]	$O(max(n, m))$
Lerch and Mezini [16]	$O(n + m)$
Campbell et al. [5]	$O(n + m)$
DURFEX [24]	$O(n + m)$
Brodie et al. [4]	$O(nm)$
PDM [7]	$O(nm)$
TraceSim [23]	$O(nm)$
Bartz et al. [1]	$O(nm)$
S3M [15][4]	$O(dh)$

In order to effectively address crash report deduplication, *FaST* leverages the empirical findings observed in TraceSim's study [23], e.g., frame position, subroutine global frequency, normalization and function diff$(\cdot)$ are crucial for this task. On the other hand, *FaST* finds sub-optimal alignments in linear time complexity that are as effective as the optimal ones found by TraceSim in quadratic time (more details in Section 5). Moreover, our method uses a different function match$(\cdot)$ that is based on the sum of the frame weights instead of the maximum value between them. Such function allows to simplify the normalization: whereas TraceSim's ones is inspired by the weighted Jaccard index, *FaST* 's normalization divides the alignment score by the sum of all frame weights in the stack traces.

Regarding the literature of sequence alignment works, the majority of them come from bioinformatics field. Thus, several heuristics for this problem make use of specific aspects of this domain to speed up algorithms [2]. Although, few optimal sequence alignment techniques were proposed besides the NW algorithm, they still run in $O(nm)$, being their superiority restricted to bioinformatics instances [6]. Overall, due to the particular characteristics of bioinformatics field, the proposed methods and their findings are not applied to the crash report deduplication task.

4 EXPERIMENTAL SETUP

In this section, we present the main components of our experimental setup: datasets, evaluation methodology, evaluation metrics, and competing methods. The developed code – including the evaluation

[3] https://lucene.apache.org/

[4] This corresponds to the multilayer perceptron complexity time. Such component contains one hidden layer of size h and receives an input of size d.

framework and implementation of *FaST* and competing methods – are available online[5].

4.1 Datasets

The datasets published by Rodrigues et al. [23] are employed in our experiments[6]. Due to scarcity of publicly labeled data, a common practice in the literature is to generate crash reports by extracting stack traces from bug reports. Thus, such datasets were created by parsing bug reports from bug tracking systems (BTS) of four open source projects: Ubuntu [17], Eclipse [10], Netbeans [12], and Gnome [11]. Ubuntu's and Gnome's repositories are composed of issues from different applications for Ubuntu Linux distribution and Gnome desktop environment, respectively. Most of these applications are developed in C/C++. Eclipse and Netbeans are two popular Integrated Development Environments (IDEs) implemented in Java. Statistics of these datasets are presented in Table 2.

Table 2: Statistics of datasets.

Dataset	Period	# Duplicates	# Reports	# Buckets
Ubuntu	25/05/07 - 18/10/15	11,468	15,293	3,825
Eclipse	11/10/01 - 31/12/18	8,332	55,968	47,636
Netbeans	25/09/98 - 31/12/16	13,703	65,417	51,714
Gnome	02/01/98 - 31/12/11	117,216	218,160	100,944

We perform two extra preprocessing steps in addition to the ones applied in [23]. In the provided datasets, a crash report can contain multiple stack traces. Rodrigues et al. [23] decided to include all identified stack traces found in the description and attached files of the original bug reports due to different reasons, e.g., the difficulty to determine which subroutine caused the failure, parsing limitations, among others. However, we observed that a significant portion of the stack traces in crash reports are, in fact, the top-k frames of other stack traces in the same reports. In order to improve the readability, testers and developers may only provide the first frames of a stack trace in the description of the bug report. The full content is attached to the report as a file. Thus, to remove this duplicate data, we identify the longest stack trace ST^l in a crash report r and, then, the remaining stack traces in r are filtered when they are a prefix of ST^l. Moreover, specifically for Gnome, we applied the same procedure used in the BTS to identify the "interesting stack traces" of multi-thread systems[7]. In a nutshell, such procedure consists in keeping or removing stack traces based on a list of relevant subroutine names (e.g., `signal` and `segv`). In Table 3, we present the number of crash reports with more than one stack trace found in the datasets before and after our preprocessing.

4.2 Evaluation Methodology

In this work, in order to assess different methods, we employ the comprehensive evaluation methodology proposed in [23]. This methodology uses a dataset D composed of crash reports sorted by their creation date. Then, a *query set* Q is generated by randomly selecting a sequence of consecutive reports in D. In order to assess a similarity-based deduplication method, each query report $q \in Q$ is considered as a newly submitted report, and the method is used to compute the similarity between q and older reports in D (reports submitted before q).

Table 3: Percentage of reports with multiple stack traces in each dataset before and after the preprocessing.

BTS	Before	After
Ubuntu	0.03%	0.03%
Eclipse	24.37%	23.75%
Netbeans	61.37%	28.93%
Gnome	80.14%	9.52%

As mentioned before, crash reports in the dataset are grouped into buckets. A bucket is the set of all reports associated to the same software bug and is denoted as B_r, where r is the first submitted (oldest) report in B_r. In the used evaluation methodology, when a query report q is considered, buckets for reports submitted before q are known. In that way, the evaluation is based on similarities between the query report q and the buckets, instead of individual reports. The similarity between q and a bucket B is defined as:

$$\text{sim}'(q, B) = \max_{c \in B} \text{sim}(q, c),$$

where $\text{sim}(q, c)$ is the similarity between the query report q and a candidate report c of B calculated by the system being evaluated.

Since crash report datasets can be very large, in order to improve system's efficiency, the deduplication of a query report q is restricted to a subset of *candidate buckets* denoted as $C^B(q)$. This set comprises only buckets that include at least one report submitted within a time window of two years before q. All reports in such selected buckets are considered as candidates, including those submitted outside of the time window. Therefore, when deduplicating q, buckets are *unreachable* if all theirs reports are submitted more than two years before q. Reports created after q are always ignored, as mentioned above.

For a more detailed and comprehensive explanation of the methodology, we refer the reader to Rodrigues et al. [23].

4.3 Evaluation Metrics

Considering a query set Q, the methodology evaluates a method by means of three metrics: Mean of Average Precision (MAP), Recall Rate@k (RR@k), and Area Under the ROC Curve (AUC). MAP and RR@k are ranking metrics, i.e., they assess the quality of ranked lists generated for each query based on the similarity technique. In this methodology, a ranked list, denoted as $L(q)$, consists of the candidate buckets for a query report q sorted by their similarity to q in ascending order. Moreover, the ranking metrics are not measured for queries related to crashes that have never been reported before. Such non-duplicate reports, called singletons, are ignored in this ranking evaluation since their respective ranked lists do not contain their correct bucket (the relevant candidate). We denote $Q^d \subset Q$ the subset of non-singleton queries.

[5] https://github.com/irving-muller/FaST

[6] The dataset is available on https://zenodo.org/record/5746044#.YeDFCNtyZH5

[7] The original code can be found in the function `interesting_threads` in the following file: https://bazaar.launchpad.net/~bgo-maintainers/bugzilla-traceparser/3.4/view/head:/lib/TraceParser/Trace.pm

Considering $p_{L(q)}$ as the position of the correct bucket of a query q within a ranked list $L(q)$, RR@kk is computed as follows:

$$\text{RR@}k = \frac{\sum_{q \in Q^d} \mathbb{1}[p_{L(q)} \leq k]}{|Q^d|},$$

where $k \in \mathbb{N}^+$ and $\mathbb{1}[p_L \leq k]$ returns 1 if the position of the correct bucket is in the top-k positions of a ranked list, and 0 otherwise. In summary, RR@kk is the fraction of queries in Q^d whose correct buckets appear in the first k positions of the ranked lists. Particularly, in realistic scenarios where reports are automatically assigned to the most similar buckets, RR@k1 represents the system accuracy for assigning duplicate reports to buckets.

Unlike RR@kk, MAP can summarize the ranked list quality by means of a single real value. In this methodology setting, since there is one relevant item within a ranked list, MAP can be simplified as:

$$\text{MAP} = \frac{1}{|Q^d|} \sum_{q \in Q^d} \frac{1}{p_{L(q)}}.$$

MAP values range in the interval $[0, 1]$ where MAP = 1 when the correct bucket is among the first elements in all ranked lists.

In real systems, it is important to distinguish a duplicate report from a singleton since a significant portion of the queries are non-duplicate reports. Thus, in order to consider such aspect of the deduplication task, this methodology also cast this task as a binary classification. In this case, a query is classified as duplicate when the highest similarity score between the query and its candidate buckets is greater than a given threshold t. For an evaluation that is independent of the threshold value, the classification performance is measured by the well-known area under the ROC curve (AUC) metric [13]. The ROC (receiving operating characteristic) curve is the plot of true positive rate versus the false positive rate for all possible values of t. The AUC metric derives a single real number from the ROC curve.

4.4 Parameter Tuning and Model Validation

Following [23], evaluation is performed using two subsequent, but disjoint, query sets: a tuning set T and a validation set V. The query set V is composed of reports submitted during a (randomly selected) period of one year, and the query set T comprises the last 250 reports submitted immediately before V. Additionally, P is denoted as the set of reports submitted earlier than T. In the experiments, the parameters are first tuned on T by means of a Tree-structured Parzen Estimator (TPE) [3]. Given a maximum number of iterations[8], such optimizer tries to search for parameter values that maximize the sum of MAP and AUC scores on the tuning set. Finally, using the best parameters found, we evaluate the method effectiveness on the corresponding validation set V.

Since data distribution tends to significantly change during the repository lifetime, the performance of the same method can highly vary depending on the data period used in the evaluation [22]. In order to better capture the method effectiveness along the whole repository, 50 validation sets (periods of one year) are randomly selected in each dataset. Thus, the tuning sets are generated based on each sampled validation set. In Figure 5, we illustrate an example where three random validation sets are sampled.

[8]TPE is run in 100 iterations.

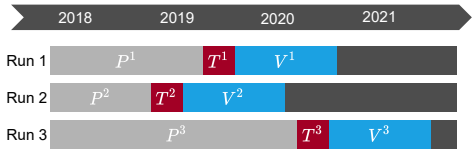


Figure 5: Three validation sets (along with the corresponding tuning sets) sampled from a dataset. The validation set, tuning set, and P in run k is represented as V^k, T^k, and P^k, respectively.

4.5 Competing Methods

In order to empirically demonstrate the effectiveness and efficiency of the proposed alignment heuristic, we compare *FaST* to two optimal sequence alignment methods: *TraceSim* and *PDM*. TraceSim significantly outperformed sequence matching and information retrieval methods for the majority of metrics (AUC, MAP, and RR@k) and datasets. On the other hand, PDM was the only method to surpass TraceSim's performance in one specific scenario and it can be better optimized than others NW algorithm variants [4, 23]. Finally, our method is also compared to a modified version of TraceSim, called *TSM*, whose match$(\cdot)$ and normalization are equivalent to *FaST*'s ones. Our objective is to investigate whether the proposed match$(\cdot)$ and normalization significantly impact the method effectiveness.

Additionally, *FaST* is compared to Prefix Match, TF-IDF, and DURFEX due to their relatively low time complexity. For simplicity, the first method is abbreviated to *PrefixM*. To guarantee a fair comparison in the experiments, we implement TF-IDF in our evaluation framework following Lucene's implementation and we only consider the subroutine names and positions within stack traces for the crash report deduplication. The subroutine names are the fully qualified method names in Java's stack traces while function names in C++'s ones. Finally, we only evaluate DURFEX on Eclipse and Netbeans datasets, since it was designed for the Java language.

5 EXPERIMENTAL RESULTS

In this section, we study the effectiveness and efficiency of *FaST* in comparison with four competing methods on Ubuntu, Eclipse, Netbeans, and Gnome datasets. For each method, we report throughput values (queries/second) and their distributions over the 50 validation sets by means of box plots combined with scatter plots. Moreover, we measure the speedup between *FaST* and a competing technique in terms of throughput on each validation set. Then, we depict the distribution of the obtained speedup values over the validation sets using box plots. It is worthy to mention here that speedup between two methods in dataset depends only how fast each method compares two stack traces. Thus, other variables related to the dataset (e.g., the number of reports) do not affect such measurement.

Furthermore, violin plots [14] are used to present the distribution values of AUC, MAP, and RR@1 achieved by each method over the validation sets. Such plots are generated by means of seaborn library and they contains three dashed lines to represent the 25th, the 50th, and the 75th percentiles. Additionally, we calculate the performance

differences between *FaST* and each competitor regarding AUC, MAP, and RR@1 in each validation set. The differences of such metrics are denoted ΔAUC, ΔMAP, and ΔRR@1, respectively. These differences are positive whenever *FaST* outperforms its competitor. We report the distributions of ΔAUC, ΔMAP, and ΔRR@1 in the 50 validation sets using box plots. Throughout this section, the mean values are depicted as white circles in the plots.

In order to assess whether a method superiority is statistically significant, we apply the Wilcoxon signed-rank test to ΔAUC, ΔMAP, and ΔRR@1 as follows:

H_0: The two methods yield the same performance.

H_1: The two methods yield different performances.

We accept the alternative hypothesis (H_1), consequently rejecting the null hypothesis (H_0), when $p < 0.01$. In the plots, the symbol ★ next to the name of a method indicates that the performance difference to *FaST* is statistically significant.

In Figure 6 (left), we depict the throughput of *FaST* and competitive methods on Ubuntu, Eclipse, Netbeans, and Gnome. At the right of this figure, we report the distribution of speedups between *FaST* and its competing methods on the tested datasets. Additionally, in Figure 7, we depict the differences ΔAUC, ΔMAP, and ΔRR@1 between FaST and competitors on Ubuntu, Eclipse, Netbeans, and Gnome. Complementary, for each dataset, the distributions of the absolute metric values are depicted in Figure 8.

Overall, *FaST* is not only more efficient than the optimal sequence alignment methods, but it is also at least as effective as them. As shown in Figure 6, considering the speedup average, *FaST* is two to four times faster than PDM while its speedup ranges between 4x - 8x regarding TraceSim. In addition to this superior efficiency, our method significantly surpasses PDM and TraceSim in nine and six of the twelve possible evaluation scenarios, respectively. In the remaining ones, the performance of *FaST* and such techniques are considered as comparable since the differences in their results are not statistical significant.

As expected, the efficiency superiority of *FaST* over TSM is similar to the one observed in the previous TraceSim's analysis. However, in terms of effectiveness, we do not find statistical significance in their performances in the evaluation scenarios, except on Ubuntu regarding ΔMAP which *FaST* is superior. Despite this finding, we cannot conclude whether the normalization and function match$(\cdot)$ used in *FaST* are more effective than the ones proposed in TraceSim. In additional significance tests, we found that TSM's and TraceSim's performances are comparable in all scenario with the exception to Netbeans regarding RR@1.

In our experiments, PrefixM is the most efficient technique. In comparison to our method, on average, it is 2.19, 4.30, 6.35, and 2.53 times faster than *FaST* in Ubuntu, Eclipse, Netbeans, and Gnome, respectively. Such high efficiency is due to the fact that the similarity is computed only considering the topmost shared frames between two stack traces, i.e., it can easily filter frames that do not affect the comparison. However, this negatively affects the method effectiveness. As reported in Figure 7, PrefixM is significantly outperformed by *FaST* regarding AUC, MAP, and RR@1 on all datasets. For instance, the lowest average of ΔAUC, ΔMAP, and ΔRR@1 between these two methods are +1.57%, +2.31%, and +1.61% in Netbeans. However, we observe higher performance differences in datasets with C++ stack traces, e.g., *FaST* largely outperforms PrefixM by 4.20%, 8.76%, and 7.67% regarding ΔAUC, ΔMAP, and ΔRR@1 in Ubuntu.

As shown in Figure 6, the second most efficient technique is TF-IDF. Regarding *FaST*, such method speeds the experiments, on average, by 1.22x, 1.19x, 1.49x, and 2.09x in Ubuntu, Eclipse, Netbeans, and Gnome, respectively. Such speedups are considerable lower than the ones found in PrefixM. But, similar to PrefixM, *FaST* significantly outperforms TF-IDF in all evaluation scenarios. For example, the lowest average value of ΔAUC, ΔMAP, and ΔRR@1 between FaST and TF-IDF are +4.23%, +2.23%, and +2.54%, respectively. However, such performances occurs in datasets that contain stack traces from C++ applications. Considering only Netbeans and Eclipse, those values increase to 9.27%, 5.66%, and 6.23%, respectively. Finally, although we were not able to observe a conclusive efficiency difference between *FaST* and DURFEX, the results show that our method is statistically more effective than the latter regarding ΔAUC, ΔMAP, and ΔRR@1 in all datasets.

6 THREATS TO VALIDITY

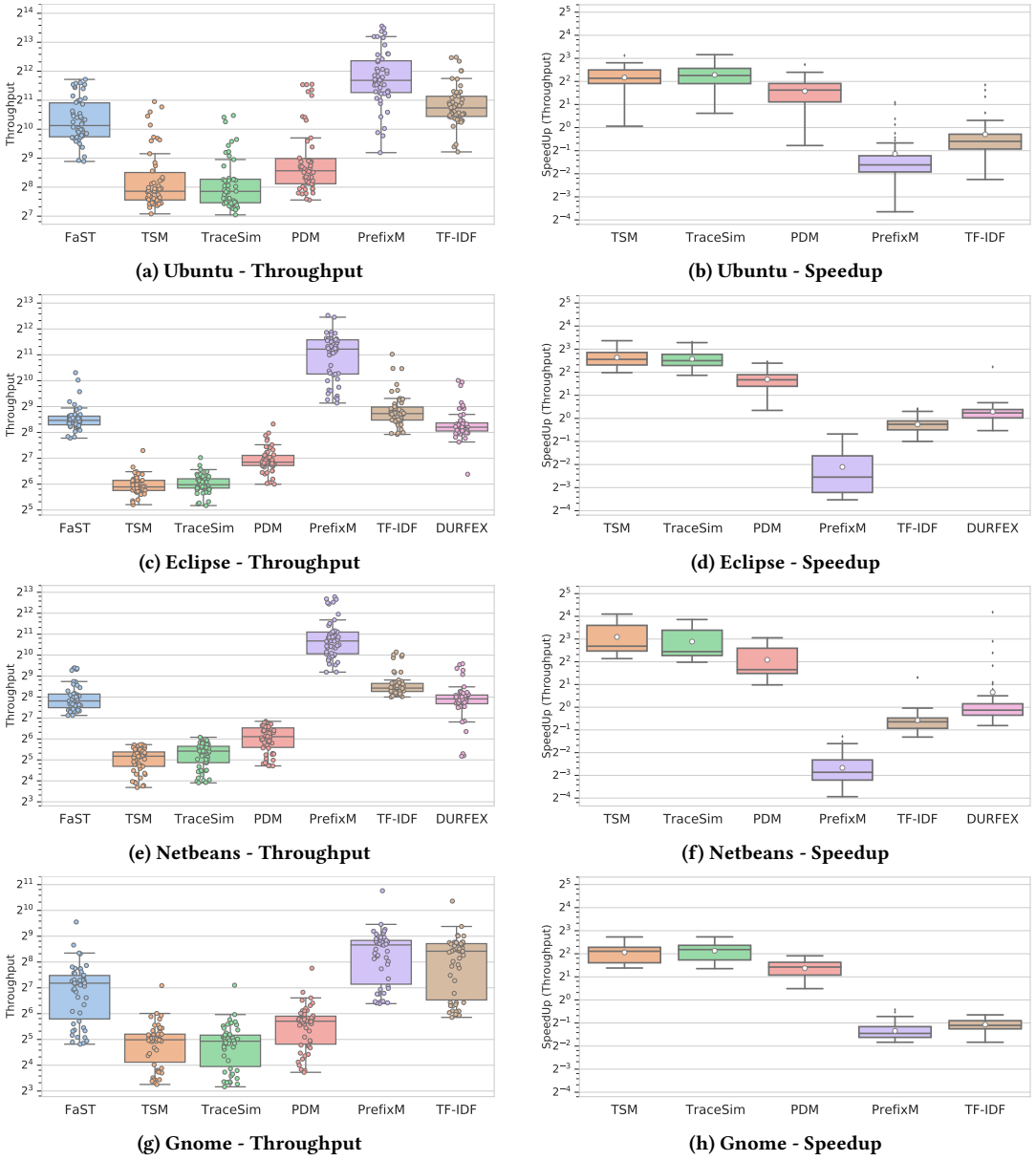
In this section, the threats to validity of our study are presented as follows.

Data quality In this study, the experimental evaluation is based on manual labeled data provided in BTSs. However, due to the complexity associated to the deduplication task, reports might be assigned to incorrect buckets or considered as singletons by triagers. Moreover, stack traces are mostly extracted from textual data, i.e., from files and report descriptions. However, due to this unstructured data nature, data extraction is not trivial and, therefore, portion of text might be incorrectly identified as a stack trace, and vice-versa. To mitigate these problems, we employ datasets that have been used in previous studies. Additionally, regarding mislabeled data, we used data from popular open source projects that contain mature triage processes. Finally, Rodrigues et al. [23] mitigate the problem related to stack trace extraction by using parsers already employed in real environments or well-known studies.

Subject selection bias. In this paper, we perform an empirical study to compare the effectiveness and efficiency of distinct methods. Thus, due to different domain characteristics, our findings might not be observed in other software projects. To mitigate such threat, our experimental setup includes data from different projects and two distinct programming languages. Moreover, the experimental methodology and framework are developed to replicate real environments as much as possible.

7 CONCLUSIONS

In this study, we proposed *FaST*, a novel alignment method heuristic for crash report deduplication. In contrast to previous methods based on optimal sequence alignment, *FaST* heuristically computes the similarity of stack traces in linear time. We experimentally evaluated *FaST* and its competing methods by means of the methodology proposed in [23]. Our results revealed that *FaST* is consistently faster than previous SOTA methods while being at least as effective – it was more effective in many of the considered scenarios. In fact, our experimental results indicate that sub-optimal alignments can be as effective as optimal ones for crash report deduplication.



(a) Ubuntu - Throughput **(b) Ubuntu - Speedup**

(c) Eclipse - Throughput **(d) Eclipse - Speedup**

(e) Netbeans - Throughput **(f) Netbeans - Speedup**

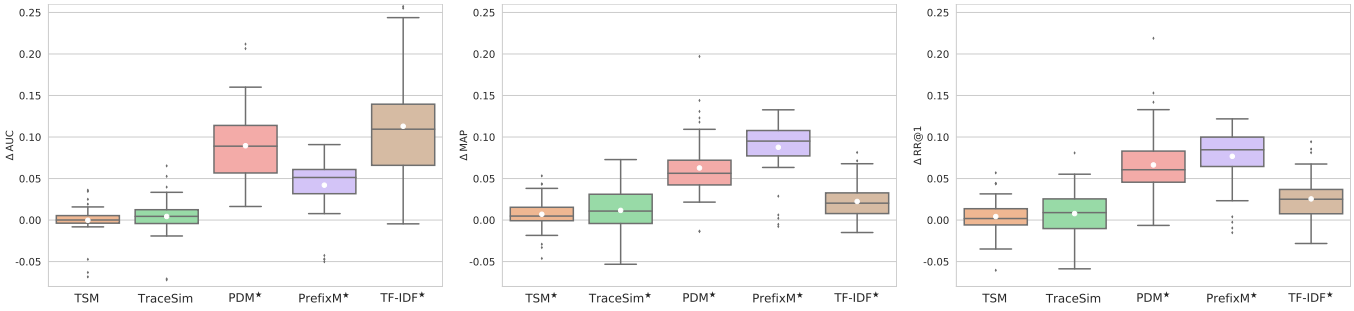
(g) Gnome - Throughput **(h) Gnome - Speedup**

Figure 6: At the left, throughput (queries per second) of methods in all validation sets of Ubuntu, Eclipse, Netbeans, and Gnome. At the right, the speedup between *FaST* and the competing methods regarding throughput.

The proposed modifications on TraceSim's scoring scheme allows our method to compute the similarity score by exclusively considering the shared frames between stack traces. For that, the assumption is that the sums of frame weights of the stack traces are known before the algorithm execution. This method capability combined to frame independence makes our method more appropriate for effectively speeding up the deduplication by means of inverted index data structure [18] or MapReduce [8]. Thus, as a possible avenue for future works, we intend to evaluate the method speedup achieved when inverted index data structure is used and the method is implemented based on MapReduce. Moreover, analogous to prefix match, we intend to investigate different strategies to compare stack traces only considering a small portion of the frames.

ACKNOWLEDGMENTS

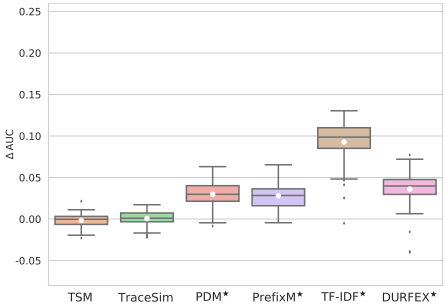
We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson, Ciena, and EffciOS for funding this project. Moreover, this research was enabled in part by the support provided by WestGrid (https://www.westgrid.ca/) and Compute Canada (www.computecanada.ca).



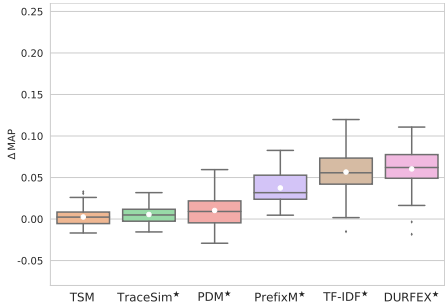
(a) Ubuntu - ΔAUC

(b) Ubuntu - ΔMAP

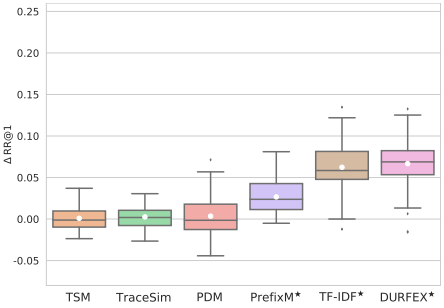
(c) Ubuntu - ΔRR@1



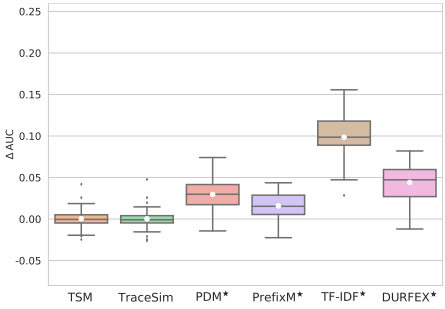
(d) Eclipse - ΔAUC



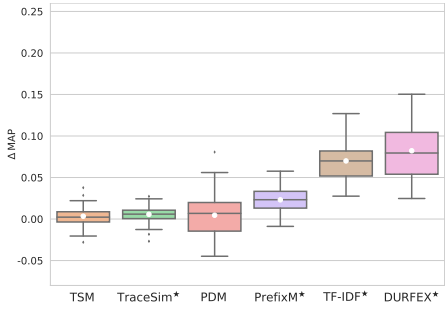
(e) Eclipse - ΔMAP



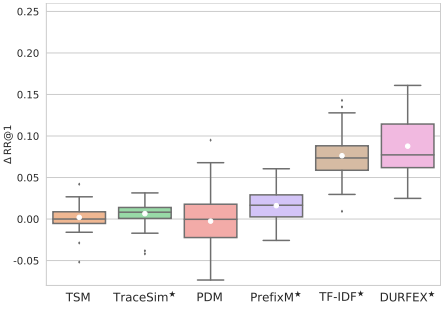
(f) Eclipse - ΔRR@1



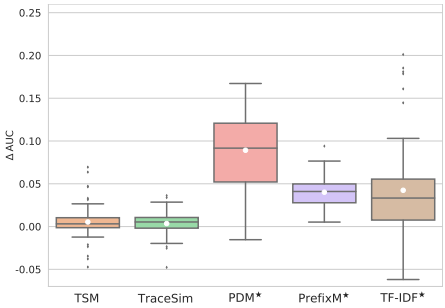
(g) Netbeans - ΔAUC



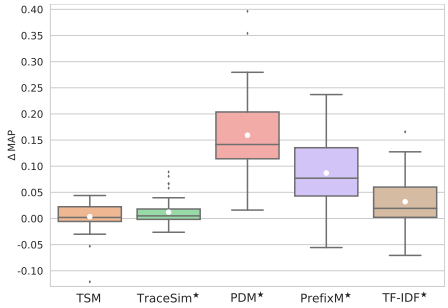
(h) Netbeans - ΔMAP



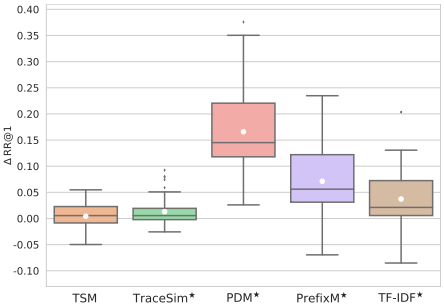
(i) Netbeans - ΔRR@1



(j) Gnome - ΔAUC



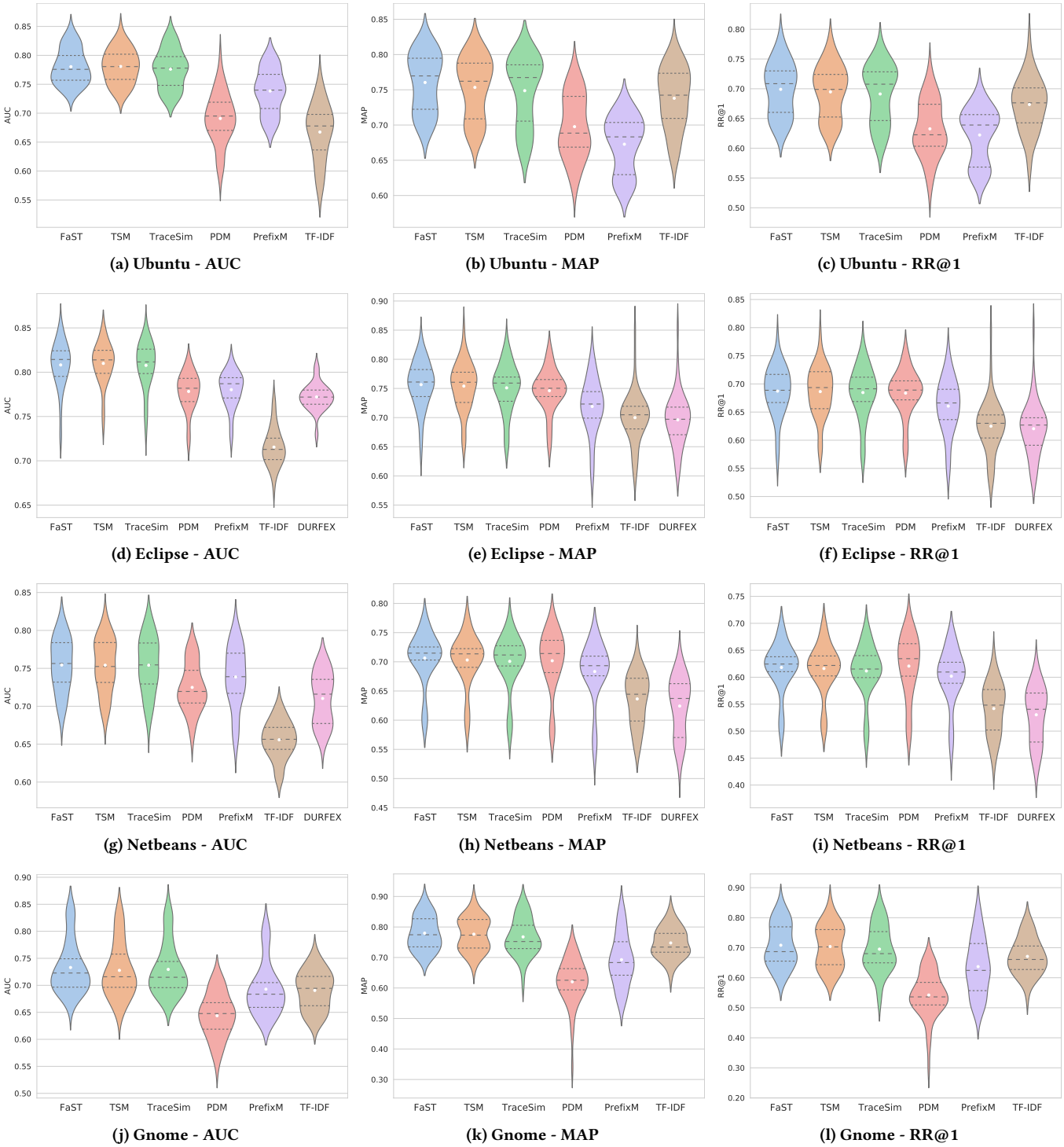
(k) Gnome - ΔMAP



(l) Gnome - ΔRR@1

★ $p < 0.01$

Figure 7: The distribution of ΔAUC, ΔMAP, and ΔRR@1 between *FaST* and each competing method in all validation sets of each dataset.



(a) Ubuntu - AUC (b) Ubuntu - MAP (c) Ubuntu - RR@1

(d) Eclipse - AUC (e) Eclipse - MAP (f) Eclipse - RR@1

(g) Netbeans - AUC (h) Netbeans - MAP (i) Netbeans - RR@1

(j) Gnome - AUC (k) Gnome - MAP (l) Gnome - RR@1

Figure 8: The distributions of AUC, MAP, and RR@1 achieved by *FaST* and competitors in all validation sets of each dataset.

REFERENCES

[1] Kevin Bartz, Jack W. Stokes, John C. Platt, Ryan Kivett, David Grant, Silviu Calinoiu, and Gretchen Loihle. 2008. Finding Similar Failures Using Callstack Similarity. In *Proceedings of the Third Conference on Tackling Computer Systems Problems with Machine Learning Techniques* (San Diego, California) *(SysML'08)*. USENIX Association, Berkeley, CA, USA, 1–1. http://dl.acm.org/citation.cfm?id=1855895.1855896

[2] Serafim Batzoglou. 2005. The many faces of sequence alignment. *Briefings in bioinformatics* 6, 1 (2005), 6–22.

[3] J. Bergstra, D. Yamins, and D. D. Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28* (Atlanta, GA, USA) *(ICML'13)*. JMLR.org, I–115–I–123.

[4] M. Brodie, Sheng Ma, G. Lohman, L. Mignet, N. Modani, M. Wilding, J. Champlin, and P. Sohn. 2005. Quickly Finding Known Software Problems via Automated Symptom Matching. In *Second International Conference on Autonomic Computing (ICAC'05)*. 101–110. https://doi.org/10.1109/ICAC.2005.49

[5] Joshua Charles Campbell, Eddie Antonio Santos, and Abram Hindle. 2016. The Unreasonable Effectiveness of Traditional Information Retrieval in Crash Report Deduplication. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. ACM, New York, NY, USA, 269–280. https://doi.org/10.1145/2901739.2901766

[6] Angana Chakraborty and Sanghamitra Bandyopadhyay. 2013. FOGSAA: Fast optimal global sequence alignment algorithm. *Scientific reports* 3, 1 (2013), 1–9.

[7] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. 2012. ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 1084–1093. http://dl.acm.org/citation.cfm?id=2337223.2337364

[8] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (jan 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[9] Tejinder Dhaliwal, Foutse Khomh, and Ying Zou. 2011. Classifying Field Crash Reports for Fixing Bugs: A Case Study of Mozilla Firefox. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance (ICSM '11)*. IEEE Computer Society, Washington, DC, USA, 333–342. https://doi.org/10.1109/ICSM.2011.6080800

[10] Eclipse Foundation. 2021. *Eclipse BTS*. https://bugs.eclipse.org/bugs/

[11] The Apache Software Foundation. 2013. *Netbeans BTS*. https://bugzilla.gnome.org/

[12] The Apache Software Foundation. 2016. *Netbeans BTS*. https://bz.apache.org/netbeans/

[13] James A Hanley and Barbara J McNeil. 1982. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology* 143, 1 (1982), 29–36.

[14] Peter Kampstra. 2008. Beanplot: A Boxplot Alternative for Visual Comparison of Distributions. *Journal of Statistical Software, Code Snippets* 28, 1 (2008), 1–9. https://doi.org/10.18637/jss.v028.c01

[15] Aleksandr Khvorov, Roman Vasiliev, George Chernishev, Irving Muller Rodrigues, Dmitrij Koznov, and Nikita Povarov. 2021. S3M: Siamese Stack (Trace) Similarity Measure. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 266–270. https://doi.org/10.1109/MSR52588.2021.00038

[16] Johannes Lerch and Mira Mezini. 2013. Finding Duplicates of Your Yet Unwritten Bug Report. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering (CSMR '13)*. IEEE Computer Society, Washington, DC, USA, 69–78. https://doi.org/10.1109/CSMR.2013.17

[17] Canonical Ltd. 2021. *Ubuntu BTS*. https://bugs.launchpad.net/

[18] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, UK. http://nlp.stanford.edu/IR-book/information-retrieval-book.html

[19] Natwar Modani, Rajeev Gupta, Guy Lohman, Tanveer Syeda-Mahmood, and Laurent Mignet. 2007. Automatically Identifying Known Software Problems. In *Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop (ICDEW '07)*. IEEE Computer Society, Washington, DC, USA, 433–441. https://doi.org/10.1109/ICDEW.2007.4401026

[20] A. Moroo, A. Aizawa, and T. Hamamoto. 2017. Reranking-based Crash Report Deduplication. In *SEKE '17*, X. He (Ed.). 507–510. https://doi.org/10.18293/SEKE2017-135

[21] S.B. Needleman and C.D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* 48, 3 (1970), 443–453. https://doi.org/10.1016/0022-2836(70)90057-4

[22] M. S. Rakha, C. Bezemer, and A. E. Hassan. 2018. Revisiting the Performance Evaluation of Automated Approaches for the Retrieval of Duplicate Issue Reports. *IEEE Transactions on Software Engineering* 44, 12 (2018), 1245–1268. https://doi.org/10.1109/TSE.2017.2755005

[23] Irving Muller Rodrigues, Aleksandr Khvorov, Daniel Aloise, Roman Vasiliev, Dmitrij Koznov, Eraldo Rezende Fernandes, George Chernishev, Dmitry Luciv, and Nikita Povarov. 2022. TraceSim: An Alignment Method for Computing Stack Trace Similarity. *Empirical Software Engineering* 27, 2 (01 Mar 2022), 53. https://doi.org/10.1007/s10664-021-10070-w

[24] Korosh Koochekian Sabor, Abdelwahab Hamou-Lhadj, and Alf Larsson. 2017. DURFEX: A Feature Extraction Technique for Efficient Detection of Duplicate Bug Reports. In *2017 IEEE International Conference on Software Quality, Reliability and Security, QRS 2017, Prague, Czech Republic, July 25-29, 2017*. IEEE, 240–250. https://doi.org/10.1109/QRS.2017.35

[25] Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. 2010. Do stack traces help developers fix bugs?. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 118–121.

[26] Peter H. Sellers. 1974. On the Theory and Computation of Evolutionary Distances. *SIAM J. Appl. Math.* 26, 4 (1974), 787–793.