Pi: A simple unsupervised approach to locating crash-inducing changes

Abstract In the process of large software projects evolution, there may be various unexpected errors because of their complexity. One of these severe faults is crash fault that would occur in the usage process by users. Crash faults often occur on these large projects may cause serious consequences. To find the root cause of one crash, developers need to check lots of source codes to locate it, which is a time-consuming and tedious work confusing developers all the time. One feasible solution to the problem is to investigate software changes inducing a crash bug, called crash-inducing changes [5]. By reviewing the these changes, developers can lock the primary cause where the crash was induced more easily. We aim to investigate the most suspicious changes which may be crash-inducing ones to give some cues in solving bugs. In this paper, we propose Pi, a simple method to locate crash-inducing changes without supervised labels. We evaluated Pi with six release versions of Netbeans project. The result show that the recall rate of Pi are close to ChangeLocator while the scores of MAP and MRR are higher than previous one.

Keywords Crash-inducing, Bug localization, Unsupervised approach

1 Introduction

Various new features and functions would be integrated together during the long-term evolution of a software. Because of the complexity of application logic and the expansion of software structure, developers would inevitably introduce some unknown errors. The more serious of these bugs are known as crash faults, which would crash in unexpected use procedure. These faults can have a great impact on the user of the software (e.g., property loss, privacy leakage, and even threat to life). Thus, locating the hidden mistakes as soon as possible is very important. Many projects have designed and deployed crash report systems such as Windows Error Reporting [1], Apple Crash Reporter, Mozilla Crash Reports and Netbeans Exception Reports for this purpose. These systems will automatically collect the corresponding crash information from user end when one crash occurs and will allocate them into different buckets to distinguish diverse bugs. A bug is considered to be a serious one that must be fixed when the amount of the crash in a bucket reaches a certain threshold. Developers usually need to check a great number of code to locate the position of the crash occured and then fix it, which is generally a time-consuming and tedious work bothered develop all the time.

A great deal of bug localization techniques(BugLocator [6], BLUiR [2], BRTracer [4]) based on bug report has been proposed to facilitate software repair process. These methods locate the bug at the source file level, which refer to a bug report and source files as a query and a set of docuemnts. They calculate the suspicious score according to the similarity between the query and the documents, and then send the developer a list of descending order files according to these scores. The files in front of the list are more likely to contain the specified bug. From the

experimental results, these methods get some satisfied results in the case where the quality of source codes and reports are both good.

Another direction of research in bug localization is to review the bug-inducing changes. Wen et al. [3] found that to understand and fix a certain bug, developers often refer to the bug inducing changes in the discussion of bug reports. Although some approaches(e.g., Locus [3], ChangeLocator [5]) have been proposed, we found that the existing approaches still exists some issues well, leading to an undesirable effect.

In this paper, We design a simple unsupervised method, called Pi, to locate crash-inducing changes which is at commit level given buckets of crash reports by computing the position of changes and the relativity with components the bug exists. To evaluate Pi, we conduct an experimental study using data from six release versions of NetBeans project. The result show that the recall rate of Pi are close to ChangeLocator while the scores of MAP and MRR are higher than previous one.

The remainder of this paper is organized as follows. First, we briefly introduce the existing approach in Section 2. Section 3 presents our proposed Pi approach in details. We then design our experiment in Section 4 and show the evaluation results in Section 5. In Section 6 and Section 7, we discuss issues involved in our approach and the threats to validity, respectively. We illustrate the related work in Section 8 and conclude this paper in Section 9.

2 Existing approach

Various bug localization approaches has been proposed before to improve the effect of localization, such as the traditional IR based approach using VSM model and its different variants(e.g., rVSM with similar bug reports [6], rVSM with structured information [2], rVSM with stack trace [4]), bug-inducing changes based methods(e.g., Locus [3], ChangeLocator [5]), etc. The rest of this section is the details introduction.

2.1 IR based approach

In early years, many researchers have porposed information retrieval(IR) techniques to locate relevant source files based on given bug report automatically [6]. IR based methods consider a bug report as a query and rank the source files by computing their relevance to the query. The most popular model used for IR methods is called VSM(e.g., Vector Space Model), which is a matrix model that each row denotes a query and each column as a source code document, and each element of which is represented by *tf-idf*. These IR based approaches don't need program execution information so that their analysis is simple and easy to implementation.

Whereas, the effect of method purely using VSM model is not just as one wishes, thus, researchers proposed many different revised variants to boost the rank performance. Zhou et al. [6] take the length of source file into consideration and argue that the information about similar bug reports that have been fixed before is useful to locate a bug. According to their research, larger source code files tend to have higher probability of containing a bug and the assumption about similar bugs is that similar bugs tend to fix similar files.

2.2 Bug-inducing changes based approach

According to Wu et al.'s [5] investigation, bug-inducing changes is a good indicator to fix a bug. They summarize that developers can obtain four primary benefits by checking and understanding bug-inducing changes. First, a developer who committed specific bug-inducing changes is familiar with the corrsponding buggy source code that need be fixed. Second, the contextual clues underlying in a bug-inducing change is more easily to find so that develop can

get more helpful information about how to fix the bug. A bug, moreover, may be fixed very conveniently just by reverting bug-inducing changes and 1069 bugs in NetBeans project was fixed following the opeartor. They also found that *automatic program repair techniques* can greatly benefited from the ability to locate the bug inducing changes. Based on these researches of bug-inducing changes, they implemented their technique as a tool call ChangeLocator.

The research results of Wu et al's empirical study also suggest that one can narrow down the candidate set of the crash-inducing changes based on crash reports. Futhermore, crash-inducing changes exhibit certain characteristics. Crash-inducing changes usuallly appear closer to the crash point that is the crash occured and a change is more likely to be a inducing one if the change modified the lines near to the crashing lines. In addition, Crash-inducing changes for a bucket appear frequently in the crash reports of that bucket, and those changes that appear in multiple buckets are less likely to be crash-inducing changes. And that, The change affects the source files in the component where the bucket of crash reports happened is more likely to be a crash-inducing change for that bucket. Finally, the committed time of crash-inducing changes is closer to the reporting time of the crashes.

3 Pi approach

As mentioned before, we select two heuristic metrics IADCP and CC in ChangeLocator and compose them to *Pi*, a new metric to measure the possibility of one change is a crash-inducing one. Then, We get the ranked list by ordering corrsponding Pi score.

Inverse Average Distance to Crashing Point(IADCP) Wu et al.'s empirical study indicate that, if a revision appears in a crash frame which is closer to the crash point, it is more likely to be the crash-inducing change. Therefore, the feature *Inverse Average Distance to Crash Point(IADCP)* that measures how close a revision r is to the crash point of crash reports in a crash bucket B. The fromal definition is as follows.

$$IADCP(r, B) = \frac{1}{1 + \sum_{j=1}^{n} DCP_j(r)/n} \tag{1}$$

where n is the total number of crash reports within the bucket B which contains revision r, and $DCP_j(r)$ represents the minimum distance between the revision r and the crash point in the j^{th} crash report of the bucket B.

Crash Component(CC) As mentioned in Section 2.2, if source files of one component that the crash bucket belongs to was affected by a revision, it is more likely to be the crash-inducing changes. Therefore, **CC** metric was designed to revise the IADCP measure when the score of IADCP of an unrelated revision.

$$CC(r, B) = \begin{cases} 1 \text{ if } r \text{ affects source files in the component that } B \text{ belongs to} \\ 0 \text{ otherwise} \end{cases} \tag{2}$$

Pi(Pos revised by isComponent) It is intuitively plausible that if a revision does not affect the component that the crash bucket belongs to, the revision is more unlikely to the crash inducing one. *Pi* is suitable for removing the revision discussed above.

$$Pi(r, B) = IADCP * CC \tag{3}$$

For example, given two revision $r_a = 0.9$, $r_b = 0.8$ and one bucket B, r_a would rank in front of r_b if we only use $IADCP$ measure. The result may be correct in the case that the relativity

is identical. Whereas, the result is unreliable if r_b has affected portion that containing crash components while r_a over the left. By using *Pi*, the value of r_a turns into 0 so that r_b becomes the most suspicious revision.

4 Experimental design

This section describes our experimental design for evaluating Pi. We first introduce the experimental setup. Then, we describe the metrics. After that, we present the research questions.

4.1 Experimental Setup

We choose the large-scale projects, namely NetBeans[1], as our evaluation subjects. The details of the dataset are listed in Table 1. To compare the performance of *Pi* and existing approaches, we choose two mentioned above state-of-art approaches, *Locus* and *ChangeLocator* as our cross reference.

Table 1: Six released NetBeans subject for evaluation

Released Version	#Files	KLOC	#Revs	#Buckets
Netbeans 6.5	25,558	5,524	110,887	42
Netbeans 6.7	48,044	10,331	139,189	61
Netbeans 6.8	499,18	10,717	158,936	52
Netbeans 6.9	53,264	11,632	176,495	40
Netbeans 7.0	46,189	9,745	196,886	38
Netbeans 7.1	38,900	8,429	217,206	41
Netbeans 7.2	39,674	8,666	235,526	39

4.2 Evaluation Metrics

Just like ChangeLocator, Pi will produce a ranked list of all candidate changes according to their suspicious score. Developers can check the changes from the top of list and find out the location where crash occured. In order to evaluate the effectiveness of Pi, we adopt the following three metrics, which are widely used to evaluate the performance of ranking relevant tecchniques.

- **Recall@K** This metric measures the percentage of crash-inducing changes for which we make at least one correct recommendation in the top k ranked results. Since we are only considering the top few ranks, and only requires finding one of the bug-inducing for per bug, the **K** usually is assigned as 1, 5 and 10. The higher thees three metric values, the better the crash localization performance. This metric emphasizes early precision over total recall.
- **MRR** (Mean Reciprocal Rank) This metric is used to evaluate the ability to locate the first relevant bug-inducing change for a bug. The reciprocal rank for a crash is the inverse rank of the first relevant bug-inducing change found. MRR is the reciprocal rank averaged over all inducing changes.

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{1}$$

[1]https://netbeans.org/

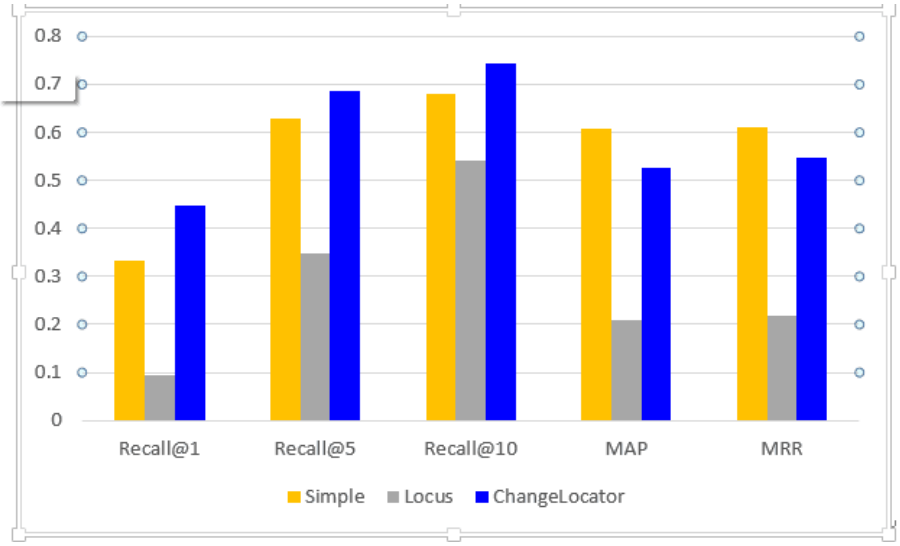
- **MAP** The metric provides a single-figure measure of quality across recall levels. This metric is used to evaluate the ability of approaches to locate all the buggy entities of a bug.

$$MAP = \sum_{q=1}^{|Q|} \frac{AvgP(q)}{|Q|}, AvgP = \sum_{k \in K} \frac{Prec@k}{|K|} \tag{2}$$

$$Prec@k = \frac{\#\ of\ relevant\ docs\ in\ top\ k}{k} \tag{3}$$

4.3 Research Questions

5 Experimental result



6 Discussions

7 Threats to validity

8 Related work

9 Conclusions

10 Acknowledgments

We would like to thank Wu et al. [5] , who collect the NetBeans project dataset.

References

[1] Kinshuman Kinshumann, Kirk Glerum, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (very) large: ten years of implementation and experience. *Communications of the Acm*, 54(7):111–116, 2011.

[2] Ripon Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne Perry. Improving bug localization using structured information retrieval. pages 345–355, 11 2013.

[3] Ming Wen, Rongxin Wu, and Shing Chi Cheung. Locus: Locating bugs from software changes. pages 262–273, 2016.

[4] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. pages 181–190, 01 2014.

[5] Rongxin wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. Changelocator: locate crash-inducing changes based on crash reports. pages 1–35, 11 2017.

[6] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *International Conference on Software Engineering*, pages 14–24, 2012.