Comparison of Different Multi Objective Evolutionary Algorithms for Bug Localization

Maria Tamoor [1], Saira Osama [2], Dr Irfan Younas [3], Saara Asif [4]
1 NUCES and FCC University Lahore, Pakistan. 2,3 NUCES, Lahore, Pakistan
4 FCC University, Lahore

Abstract—Bug localization is a crucial and time consuming process in software debugging and testing phase. Many effective techniques have been proposed to address the problem. In this paper, we have introduced a novel approach by mapping the problem as a multi objective optimization problem to minimize the number of relevant code fragments (classes) and maximizing the similarity between the bug report and the retrieved code fragment. The solution is based on finding the cosine similarity of a bug report with different classes. We performed our experiments on three different open source java projects using releases prior to bug fixes for respective bug reports. Experiment results showed promising results that outperforms our approach in making correct recommendations of relevant code fragments. Precisely our approach truly locates top 15 ranked buggy classes for more than 85% bug reports.

Keywords: Multi objective optimization method; bug reports; software debugging and testing, cosine similarity.

1. Introduction

A bug in software is an error in the software code which makes the program to produce incorrect and unexpected results. Irrespective of the effort which a developer spends on developing a program, it still contains bugs. Basically probability of finding bugs is directly proportional to the complexity of the program [7]. Number of the bug reports can be very large e.g. MOZILLA has more than 420,000 buggy reports [9]. During software debugging and testing phase, bug localization is the main concern in which identification of the exact bug locations in the source files of the program are determined. This process is a very expensive in terms of time i.e. time required by the developer to exactly locate and remove the bug. In addition, bug reports with less information can make the process more tedious and time consuming. This time also depends on the effectiveness of the technique used for bug localization. Effectiveness is usually measured by using the metrics MAP (Mean Average Precision) and top N Relevant Ranked classes.

An efficient and effective methodology would lead the developer to perform their tasks in a minimum possible time in a more accurately. The method should acquire less effort and time of the developers.

In recent research it is determined that statistical bug localization is more effective as compared to the dynamic one, because they can be useful at any phase of software development. For this a BugCatcher tool is created [20]. The main objective of the tool is to re-rank the documents to increase the localization precision. Finding the bug location can also be done using both the semantic as well as structural information [21]. Detailed work on the addressed problem is given in the literature review section. Most of the studies so far rely on the lexical matching measures in the bug reports and code elements. Consequently, all these approaches have limited efficiency as natural languages and programming languages differ and their similarity is needs to be contemplated.

In our work, we have used the cosine similarity in the bug report and source file after performing some preprocessing of bug reports and the source code files. Using API documentation of code fragments, we can generate a better similarity than the source code itself [11]. Classes recently ranked relevant using the bug reports are likely to be bug prone for the current bug reports of the same type. Similarly, recently ranked relevant classes still may contain more bugs in comparison to the previous classes which are ranked long time ago. Lastly, inspection of large number of ranked classes may be more time consuming in finding the bug location. A multi-objective optimization algorithm NSGA-II has already been used for bug localization considering these observations. In this research, we suggested an approach that uses bug reports written in English language and finding the relevance of bug reports to the source code. We recommend the use of another multi-objective optimization algorithm SPEA II given in Algorithm 1. Research shows that SPEA-II outperforms NSGA-II in the early generations [10]. We tried to minimize the number of relevant suggested classes and maximizing lexical similarity of bug reports with the code files. We formulate the problem (figure 1) as a search of
optimal solution in terms of our two above specified conflicting objectives as shown in the Table 1.
To evaluate our proposed approach, we performed a number of experiments using 3 open source versions of java projects prior to bug fixes. We used Eclipse, SWT and AspectJ and took more than 10,000 reports which contain errors from the existing benchmark. Precisely our approach truly locates top 15 ranked buggy classes for more than 85% bug reports. Bug report Sample of SWT (ID: 227638) is given in figure 2, a unique id is associated with each bug report and summary and description is added with respect to each bug report.

2. Related Work

Bunescu R. and Liu C. used rank technique, in which they trained weights using previous bugs’ reports and used them in narrowing down the search to most relevant bug reports [1]. In this ranking model they computed a matching score for every bug report and used this as training model for any new bug report to determine the relevance. As evaluation measures they used accuracy, MAP and MRR (mean reciprocal rank) and they found very promising results over the testing data set.

Saha et al. [2] used static technique of informational retrieval to avoid expensiveness and time consuming approach of dynamic methods of bug localization. They used a localization tool for the automation of bugs which is basically based on the model of structured information retrieval. They effectively used paradigms such as bug summaries, names of classes and methods which are present in several bug reports which can be used for more accurate localization of bugs. Though they observed that structured retrieval is more expensive as compared to normal text retrieval but it gives promising results due to its better accuracy and overall saves developer’s time.

Methods for model inferencing, which are used for documents are normally in natural language, are proved to be generally useful to various software related documents, Zhang et al [3]. In their approach, they adapted an inference method which is event-based model and used for generating additional tests using genetic search-based algorithms. This type of inference has been extensively used in software testing. In the paper, they aimed to perform model inference from bug reports which are generated by users and not using traditional system logs. They used NSGA-II and GA [4] for model inference from bug reports. To determine the performance of both algorithms on proposed idea they used hypervolume [5] for analyses and got sound results.
To handle large number of reports, researchers have inspected bug reports based on fault localization approach. Localization refers to finding a small subset from the entire code. But large files may contain noise and contribute less in finding relevant code fragment with respect to the bug reports. The issue can be resolved as proposed by Chu-Pan Wong et al. segmenting the source code file and use the part with the highest similarity [12]. As a result, the noise that may be present in other segments of the file was reduced. The other technique used in the paper is Stacktrace analysis in which ranking of the files covered by the stack trace that can be more susceptible to having bugs, was increased. This work improved the performance of BugLocator [13].

Another contribution is by Xin Ye et al. who have used API documentations to shrink the lexical gap present in the bug reports and the source codes; training examples consist of already fixed bug reports for the suggested ranking model in combination with learning to rank technique [14]. The evaluation results presented show the outperformance compared with the previous benchmarks for over 70% of the bug reports.

As system softwares are complex due to their wide domains, they often contain bugs and errors that might be the cause of loss of billions of dollars in software industry [15]. So, bugs fixation is a vital process in software maintenance phase [16]. One of the major contributions in this regard is the use of discriminative models for IR to spot duplicate bug reports as the same bug might be reported twice or more by one or different users [17]. For this purpose, extensive feature extraction is performed to calculate the similarity between two reports. Use of this technique improved the accuracy up to 43% for state of the art automated bug detection systems.

Another major addition to the bug localization process is the study of effectiveness of techniques used for bug localizations for C language [18] unlike earlier work focused on Java. In this paper, a benchmark dataset which contained more than 7,500 bug reports from five famous C projects was created and evaluated it using their own bug localization tool BLUir. Their results indicate that the localization in C language at the source file level is generally as effective as in Java language but the use of program structure information for bug localization is tilted towards Java than C. Also they established the relationship in the use of English language words in code and the success of bug localization. For C and Java code, bug localization mostly depend on similar information such as method names and identifiers’ names. Bug localization through C filenames provides less information than Java classes.

Kochhar et al. [19] performed a study by surveying 386 software professionals from more than 30 different
countries about their anticipations of research in fault localization and compiled their results which were found useful to direct the research in this regard.

A paper [20] focuses on bug localization and introduced a new multilevel re-ranking approach. It got promising scores for the top N documents, MAP and MRR for the benchmark data sets used in the previous paper of bug localization tools. They created a BugCatcher tool whose main objective is to re-rank the documents to improve the localization precision. The results showed that the BugLocator defeats the previous BLUir and BugCatcher. Davendra et al. [21] proposed to find the bug location using both semantic as well as structural information. They used the merger of LDA (Latent Dirichlet Allocation) and Call Graph denoted as CG and named the technique LDACG approach. Results show that LDACG is better than LDA in both of the metrics.

Qianqian et al. [22] focused on the evaluation of IR techniques for bug localization. They did the study in both perspectives i.e., Analytical and user centric. It is proved in the paper that apart from being popular, the IR techniques have some limitations also reducing their performance. The performance of the techniques is measured in ranked document, MAP, MRR and BugLocator are used to compute it. The paper results show that the effectiveness of techniques can vary with varying bug reports. Reports written with thorough detail and information contribute more to the performance as compared to poorly written or less informative content reports. The user centric evaluation was performed on group of developers doing debugging with or without the IR techniques. The conclusion is that IR techniques can only be successful practically if only correctly ranked documents are retrieved and also with the information needed by the developer to locate and correct the bugs.

Various methods are proposed e.g., IR or spectrum based techniques for finding fault location in softwares to make the developer’s life easier. Program Spectra defines set of code components corresponding to different test cases. Still all the work uses any one kind of data either the textual or the spectra. Tien-Duy B. Le, et al. recommended a multimodal approach using the combination of both procedures and achieved promising results [23]. In comparison to the previous benchmarks it outperforms well in terms of localizing the bug up to 47.62% and improves the MAP by 28.80%.

3. Multi Objective Approach

First we present the idea of multi-objective method to identify the relevant code segment which is related to bug reports (classes with relevant bug reports). Then detailed information has been given for the multi objective method and formulation which we will use for getting the relevance of bug reports.

3.1 Methodology Overview

Our method determines the relevant classes and give a description about the bug report. During exploration process of search space, it is analyzed that not only by the total possible class combination numbers to be recommended, but also by the relevance of classes to each bug report. Also every bug report may require the review of many classes to locate and fix the bugs.

We have two conflicting objective functions, one is to maximize the lexical similarity based relevance in the bug reports and source codes. Second one is to minimize the relevant classes to be recommended for one bug report. Therefore, we are solving this bug localization problem as multi objective problem by using strength Pareto evolutionary algorithm II (SPEA II). This algorithm will be covering a large search and objective space.

The basic idea and organization of our proposed approach has been outlined in Figure 1. The input to algorithm is JAVA source code of the program, description of the bug reports and specific bug report for which we want to calculate the relevant classes. Also we have testing dataset containing bug reports for which we already know the relevant classes. Output of the system will be a nearly optimal solution that will provide a series of ranked classes which will be maximizing the relevance of these ranked classes to the bug reports and also minimizing the recommended classes for a bug report. In the next part, we describe the general idea of SPEA II algorithm, solution construction, a formal mathematical formulation of previously defined two objectives which will help in getting the optimal solution.
3.2 SPEA II

SPEA II is one of the commonly used multi Objective algorithm [24]. This is a recent method for approximating and finding the Pareto optimal front for multi-objective optimization problems. Research on SPEA II has shown very promising results in comparison to the other multi-objective evolutionary algorithms. Results showed that performance values produced by SPEA II were ahead of NSGA-II for m = 2 [24]. Therefore, the main idea of SPEA II is to create a population using last child population as well as an archive which is containing pareto set optimized solutions to resolve a multi objective problem for optimization. Non dominated solution represented in the pareto set is one which provides an appropriate compromise among all objectives without decreasing performance of any one of them. Basic steps are given in Algorithm 1. As defined in the given Algorithm for SPEA II, the first step is to create a random population P₀ along with an empty archive using a specific presentation (Step 1). Then Compute fitness values for every individual in population P₀ and archive set A₀, (Step 2). Determine non-dominated solutions and add them in archive set, and next after reaching terminating times execution it will give us set of non-dominated set. Apply mating pool using binary selection on A₁ (Step 5), after that apply recombination and mutation to mating pool. Random population will be generated for each bug report which we want to inspect and will get the bug report as the input. The following subsections refer to specifically our use of SPEA II to the bug localization problem.

<table>
<thead>
<tr>
<th>Algorithm 1. Basic Pseudocode for SPEA II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1: Generate initial population p₀ along with an Archive A₀. Also Set t = 0.</td>
</tr>
<tr>
<td>Step 2: Compute fitness values for every individual in P₀ and A₀.</td>
</tr>
<tr>
<td>Step 3: A₁ = non dominated individuals in P₀ and A₀. Check if size of A₁ &gt; N, reduce A₁, otherwise fill A₁ with dominated solutions in P₀ and A₀.</td>
</tr>
<tr>
<td>Step 4: If t &gt; terminating value then output the non-dominated set which is A₁+1 and Stop</td>
</tr>
<tr>
<td>Step 5: Fill mating pool using tournament selection (binary) with replacement on A₁+1</td>
</tr>
<tr>
<td>Step 6: Use recombination operator and mutation to the mating pool and assign P₁ to the resultant population. Assign t=t+1 and then goto Step 2.</td>
</tr>
</tbody>
</table>

Solution approach

3.3 Solution representation

For an individual candidate solution representation for one bug report, a complete generation is generated for one bug report for the one we want to determine the relevant classes. Length of every solution in population is the total number of classes in the repository. Hence, one solution is designed as a series of classes for recommendation to be inspected by the programmer for bug localization. Number of relevant classes are dependent on the bug report as bug can be present and related to many java classes. Also our goal is to minimize number of the recommended classes alongside maximizing relevance of bug report to the classes which is the second objective.

Figure 2 represents a bug report to find all possible nearly optimal relevant classes. This bug report is taken from Eclipse project with bug ID: 227638 and specifies an error related to SWT.VIRTUAL which is not working properly. It also includes a detailed description to get a better solution for fixing the bug.

**Bug ID:** 227638, **Summary:** Bug 227638 Tree with SWT.VIRTUAL not working

**Description:** The WPF TreeView widget does not virtualize its items like ListView can/does. We use OnTriggerEnter to send our SetData event that tells the user to populate the SWT.TreeItem

**Reported date:** 2008-04-17 15:51:09

![Figure 2. Bug report (ID 227638)](image)

The solution for this specified bug report will be most relevant classes which are determined using lexical based similarity between bug report and all java classes. Respective Java code segment from the java SWT project is specified in figure 3 which is found relevant to the bug report given above.

```java
int C = 1000;

final String [] itemString = new String [COUNT];

for (int j = 0; j < C; j++) {
    itemString [i] = "items " + j;
}

final Table tab = new Table(parent, SWT.BORDER | SWT.VIRTUAL);

table.addListener(SWT.SetData, new Listener() {
    public void handleEvent(Event event) {
        TableItem item1 = (TableItem)event.item;

        int index1 = event.index;

        item1.setText(itemStrings [index1]);
```
A population of size N will be generated for each bug report to find relevant classes. For each bug report a binary chromosome of length n (n is total number of classes) will be created randomly.

Classes

\[
\begin{array}{cccccccccc}
1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & \ldots & 1
\end{array}
\]

Bug report

Figure 4. Binary of length n (no. of classes)

Where each bit represents one class and one binary chromosome represents lexical comparison of a bug report and a class. Bit is 1 if there is some similarity found, otherwise it will be 0.

4. Mathematical Formulation

We represent bug reports with set I and I ∈ {1, 2, m} where m is the total number of bug reports which are taken from SWT java prior to fix versions of bug reports. Similarly we used Java open source SWT to get all classes for bug localization. All classes in repository are represented with set J where J ∈ {1, 2, …, n} classes. For i ∈ I, j ∈ J, we define S(i, j) as the similarity of each class j to bug report i. The similarity matrix represents lexical similarity of each bug report with every class in the repository. Cosine similarity calculation is used for populating matrix S(i, j). Using cosine similarity, it is determined between two factors: i and j is determined as follows:

\[
S(i, j) = \cos(\vec{i}, \vec{j}) = \frac{\vec{i} \cdot \vec{j}}{\|\vec{i}\| \cdot \|\vec{j}\|}
\]

where \( \vec{i} = (w_{i,1}, w_{i,2}, \ldots, w_{i,n}) \) and \( \vec{j} = (w_{1,1}, w_{1,2}, \ldots, w_{1,n}) \) are term vector corresponding to the actor i and j respectively. Weights w_{ij} are calculated using IR (information retrieval) methods which is the Term Frequency and Inverse Term Frequency (TF-IDF) technique. The first lexical similarity function value is defined as sum of the cosine similarity values of the source codes of each of the recommended classes with description of a bug report which is divided by total number of suggested classes.

As given in Figures 2 and 3, we can see that the description of the report example comprises many interrelated words with one of the suggested classes to inspect. Consequently, the function of cosine similarity applied between the description of the bug report and the java code of the relevant class. This value helps out in achieving the objective function I.

Fitness function

Objective function I

We have described two objective functions to get the desired nearly optimal solutions as given in Table 1. We applied SPEA II which is a multi-objective algorithm and performs better than NSGA II [24] for objective functions m=2 to find out the optimal solutions using these two objective functions. We are maximizing objective function I i.e. to get maximum relevance of a bug report with java classes. Objective function II will be minimized i.e. to get smallest number of relevant classes in order to find out the optimal solution.

<table>
<thead>
<tr>
<th>Objective 1: Maximize relevance of recommended classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective 2: Minimize number of recommended classes.</td>
</tr>
</tbody>
</table>

Table 1. Multi-objectives used for SPEA II

Given below is the fitness function to implement objective function I: Maximize relevance of recommended classes,

\[
(\text{Maximize}) \ f_{1(k)} = \sum_{j=1}^{n} S(i, j) \cdot bval_{kj}
\]

where bval_{kj} represents the value for class to a bug report i (I=1,2,…m) in solution k (k= 1 … N).

Objective function II

As given in Table 1 we want to minimize the number of relevant classes which are determined for every bug report using SPEA II algorithm. Given below is the fitness function for the objective function II: Minimize the number of recommended classes,

\[
(\text{Minimize}) \ f_{2(k)} = \sum_{j=1}^{n} bval_{kj}
\]

Figure 3 code fragment of SWT project

```java
12 }();
14 tab.setItemCount(C);
15 Button b1 = new Button(parent, SWT.PUSH);
16 b1.setText("Change item on index 5");
17 b1.addListener(SWT.Selection, new Listener()
18 {
19   public void handleEvent(Event event)
20   {
21     itemStrings [5] = "item " +
22     System.currentTimeMillis();
23   }
24 });
```

where \( bval_{ij} \) represents the value for class to a bug report \( i (I=1,2,...m) \) in solution \( k (k= 1 .... N) \). We get solutions in Pareto front after using SPEA II, all solutions are non-dominated using both objective functions.

The LS (lexical based similarity) using cosine similarity is calculated between each bug report and every java source code classes. The vocabulary was built from the names of classes, variables, parameters, methods, types, etc. For tokenization we used the Camel Case Splitter for preprocessing the variables and all identifiers [25]. During this process of tokenization, a standard information retrieval technique has been used for removing the stop words to exclude irrelevant information from the code and bug reports such as numbers, punctuation etc. In addition, all vocabulary words are reduced to their stem form, based on the famous Porter stemmer for stemming of words. This operation helps out in better matching of words to the same stem design. After performing all these pre-processing steps, the cosine similarity calculation is implemented to analyze the lexical similarity in the Java source code and description of the bug report.

5. Evaluation

To evaluate our method for relevant classes’ recommendation for one bug report we performed a number of experiments using different parameters. Each experiment on one sample has been performed for different generations and different number of population size. To check the results of our experiments we have used both GA based indicating metrics for analyzing the performance of our approach as well as IR based metrics like N ranked documents, Precision and Recall. We conducted our experiments of using NSGA II and SPEA II algorithms and compared the results.

6. Performance

We validated our proposed multi-objective approach using open source java project SWT. To evaluate the accuracy of the recommended classes using NSGA II and SPEA II we used following evaluation measures:

**Precision@N**: It determines the total number of correctly recommended classes in the top N of recommended files for the bug localization. This number is divided by total minimum number of the classes to check, in the ranked recommendations list

\[
\text{Precision@N} = \frac{\text{Total number of correct recommended classes in top N files recommended}}{\text{Minimum number of classes to inspect in ranked recommendation list}}
\]

**Recall@N**: It determines the total number of correctly recommended classes in the top N classes recommended by the solution. This number is divided by the total number of the expected classes which are relevant.

7. Results

Our results obtained after several emperimentation are given in table 2 and 3.

Table 2. Recall and Precision @ N

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>N-Ranked document</th>
<th>Recall@N</th>
<th>Precision@N</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSGA II</td>
<td>22 (12)</td>
<td>54.4545%</td>
<td>66.65%</td>
</tr>
<tr>
<td>SPEA II</td>
<td>22 (13)</td>
<td>59.0909%</td>
<td>69.250%</td>
</tr>
</tbody>
</table>

Table 3. For small Max. Evaluations

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time used for execution before preprocessing</th>
<th>Time used for execution after preprocessing</th>
<th>Max evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSGA II</td>
<td>35 mins</td>
<td>20 mins</td>
<td>1000</td>
</tr>
<tr>
<td>SPEA II</td>
<td>29 mins</td>
<td>18 mins</td>
<td>1000</td>
</tr>
</tbody>
</table>

8. Conclusion and Future Work

After conducting a number of experiments on different bug reports we observed that for early generations SPEA II runs efficiently in terms of time but in increasing number of generations to nearly 1 million NSGAII runs faster. Similarly, in hyper volume calculation SPEI shows slightly better results as compared to NSGAII so it does mean that Pareto front obtained using SPEA II is diverse and converging as compared to NSGA II algorithm.

As a future work we can improve the similarities between bug report and source code using SPEA II by including API specifications of the methods which are present in the identified relevant classes. It will improve the recall and precision of evolutionary algorithms for bug localization process. History based similarity can be used along with lexical based similarity with SPEA II. Other than NSGA II and SPEA II some more multi-objective evolutionary algorithms can be used for bug localization.

REFERENCES


[12] Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. Microsoft Research, Beijing, China 2014


[14] Learning to Rank Relevant Files for Bug Reports using Domain Knowledge Xin Ye, Razvan Bunescu, and Chang Liu. SIGSOFT/FSE ‘14, November 16 – 22, 2014, Hong Kong, China


