A Strategy to Determine When to Stop Using Automatic Bug Localization

Zhendong Shi[1,2], Jacky Keung[2], Kwabena Ebo Bennin[2], Nachai Limsettho[3], Qinbao Song[1]

[1]Department of Computer Science and Technology
Xi'an Jiaotong University
Xi'an, China
zdshi0@stu.xjtu.edu.cn;
qbsong@mail.xjtu.edu.cn

[2]Department of Computer Science
City University of Hong Kong
Hong Kong, China
Jacky.Keung@cityu.edu.hk;
kebennin2-c@my.cityu.edu.hk

[3]Graduate School of Information Science
Nara Institute of Science and Technology
Nara, Japan
nachai.limsettho.nz2@is.naist.jp

***Abstract*— Information retrieval based automatic bug localization techniques provide developers a ranked list of suspicious buggy source entities to aid locate the ones needed to be modified and to fix the bug. However, it is unavoidable that some buggy entities are ranked low in the result list using these automatic techniques. We assume a bug localization process to address this challenge. Each time a source code entity in the ranked list is examined; the developers will have the option as to whether to continue examining the automatic bug localization result, or simply switch to using a conventional localization approach. We propose a new evaluation metric called ETC (Expected Time Cost) in the localization process, which includes the time cost of using the conventional approach. Under our assumptions, we derived simple criteria to minimize ETC. We compared the time cost of a state-of-art automatic localization method, BugLocator, with and without using our strategy in two projects. The result shows that using our proposed strategy combining both automatic localization technique together with conventional approach performs better than using only either the automatic localization technique or the conventional approach.**

Keywords-bug localization; information retrieval; bug reports; fault localization; feature location; evaluation metric

I. INTRODUCTION

Bug localization refers to the process of finding the relevant source code entities (methods, classes, etc.) that need to be modified to fix the bug according to a bug report [1]. Information Retrieval (IR) techniques [2–8] have been widely applied within the domain of bug localization using textual similarity between bug report and source code entities, which produces a ranked list of source code entities for developers to examine.

However, a problem with the ranked list based technique is that buggy entities which are ranked in the bottom of the result list will thus require a lot of time to identify. Conventional bug localization approaches whereby automatic methods are not employed in finding bugs are known to be more efficient than using the ranked list when the rank of relevant entity is low. We assume using the conventional technology costs a fixed time for a specific bug. The previous automatic techniques [2–8] usually do not consider the situation of using conventional localization approaches, while in reality, developers have the option to use the conventional approach besides only using the automatic localization result.

To address the issue of combining the conventional bug localization approach into automatic techniques, we assume a bug localization process, in which each time a source code entity in the ranked list has been examined, the developers will have an option whether to continue examining the automatic bug localization result or change to using the conventional localization approach. We propose a new evaluation metric ETC (Expected Time Cost) which indicates the expected total time cost including the conventional localization approaches. The value of this metric will be in direct proportion to the expected time cost in the real world. The simulated process of bug localization is to minimize the ETC value for each bug. When using the ETC as the evaluation metric for different localization techniques, the developers can approximately find out how much time it takes for a bug to be localized.

The contributions of this paper are as follows:

1. We created a hybrid bug localization simulation process that includes the use of the conventional approach.
2. We proposed a new evaluation metric in the simulated bug localization process, ETC, which is the expected time cost of localizing a bug. It includes the time expended in using the both automatic and conventional approach.
3. We proposed a strategy and derived simple criteria to minimize ETC. This strategy and criteria could be used as a recommendation to developers as to when to stop using the automatic bug localization results and when to switch over to a conventional approach.
4. We compared the performance using an automatic bug localization technique, BugLocator [9], with and without our strategy. The results show that the strategy is very effective in comparison to using only the automatic localization results or using only the conventional approaches in two real projects.

The background is presented in the next Section. The research assumptions are introduced in Section III. The probability of an entity being buggy is discussed in Section IV. The simulation process and ETC calculating are introduced in Section V. The discussion, related work, and conclusion are presented in Sections VI, VII, and VIII.

II. INFORMATION RETRIEVAL BASED BUG LOCALIZATION

When software users encounter a fault, they usually submit a bug report, which records the description of the fault, and how to reproduce the bug and other related information. The goal of bug localization is to find which of the source code entities need to be modified in order to fix the bug. Intuitively, the text description of bug contains the semantic information about the functionalities where the bug surfaced, and the source code identifiers are also named by their functionalities.

0730-3157/16 $31.00 © 2016 IEEE
DOI 10.1109/COMPSAC.2016.39

In addition, the source code comments may have related semantic information about the bug. Text information in the buggy source code entity tends to be similar with the bug report, and information retrieval techniques can provide different types of text similarity measures to aid locate the bugs.

The IR-based bug localization is similar to a web search engine: The bug report is the text query, and the source code entities are similar to that of the webpages which needs to be retrieved. The bug localization provides a ranked list of source code entities to developers according to the textual bug report. The higher the rank of the entity, the more suspicious or the higher the chance of the entity being considered as buggy. Developers are therefore tasked to evaluate the entity list from the top until the first relevant (buggy) entity is found.

III. ASSUMPTION

A. Common Assumptions

To create a model simulating the behaviors of localizing bugs in real life, we first introduce the 4 common assumptions which are widely used in bug localization area. Past study may not directly address these assumptions, while these assumptions have been considered as default rules.

Assumption 1. *The bug localization goal is to find one relevant source code entity. We assume the developers can easily find the other buggy entities after they have evaluated the first found buggy entity.*

The source code *entities* indicate the program elements such as class, file, method, etc. [10]. Recent IR-based bug localization methods usually use *file* as their granularity level, which means these techniques finally produce a ranked list of source code files [3], [4], [8], [9]. We use the word *entity* to refer to the file or method which needs to be localized, so the description can be used in different granularity levels.

Assumption 2. *Atomicity: the developer reads the given entity from the beginning to the end, and then determines whether the entity is buggy. The developer will not focus attention to other entities until he/she is done examining the whole content of the given source code entity.*

This assumption is to guarantee that each entity is an individual unit, and entities are processed one by one. The metrics that are widely used in evaluating bug localization techniques such as Top-N, MAP and MRR [11] all follow this assumption.

Assumption 3*. Each source code entity in the same project has equal time cost to inspect for developers to determine whether it is buggy or not. The time cost of checking one entity is to set to T for the whole project.*

Top-N, MAP and MRR are all based on this assumption. They treat a source code entity as a single unit. No matter how large an entity is, it is considered to cost the same time for developers to inspect. Although the lengths of source code entities vary from each other in real software developing scenario, they are treated as the same in evaluation for simplicity.

Assumption 4*. Ideal developers: The developers will precisely determine whether a source code entity is buggy or not, after they examine all contents of the entity.*

This assumption guarantees that developers will not skip the buggy entities when they check it, and also will not give wrong buggy labels to source code entities.

B. Specific Assumptions

The following assumptions form the main basis of our model, and we are the first to propose them in the domain of bug localization. We call them specific assumptions in our strategy. The last assumption is introduced in Section IV.A.

A developer has his own localization approach to find the buggy file without automatic techniques being applied, which is referred as the conventional bug localization approach. It includes examining the stack trace in bug reports, checking the source code based on bug description, reproducing the bug in execution, and so on.

Assumption 5*. When a developer checks the source code entities in the ranked list, he may then give up using the ranked result in the checking process if he does not find any buggy entities, and return to the conventional approach to localize the bug. The developer will not switch back to use the provided ranked list to localize the bug again, once he starts using a conventional approach.*

This assumption is used to simplify the behavior of bug localization. In reality, developers may switch back and forth between the entities from ranked list and other entities using conventional approaches [12]. We assume developers giving up on the ranked list mean they lose confidence in the automatic produced result, so they will not return to use it again after they have given up on it.

Assumption 6*. The time cost of using conventional localizing approach to find the first buggy entity is set to T_b for a bug report b.*

It is hard to define the time cost of a bug being localized in conventional approaches. We set it to a constant value T_b to represent the cost which developers expect in using experience for each bug. Different bugs have different expected time to be localized and fixed. For different developers, T_b also varies.

Assumption 7*. There is no intersection between human checked entities in the ranked list and conventional localization checking route.*

This assumption is to simplify the model, to ensure the value of T_b will not be influenced by entities that have been checked using the ranked list. Since both the entities checked in the ranked list and the entities checked in the conventional localization technique only occupy a small part of the whole source code, there is little chance the two types of entities will intersect with each other.

IV. PROBABILITY FOR BEING BUGGY

In this section, we first introduce the definition of the probability of the first entity in the ranking list being buggy, and then using real data to illustrate the monotonically decreasing attribute of the probability.

A. Definition

Definition: The probability of the k^{th} ranked entity of the result list being buggy in the examining process is defined as P_k. Since the localization task is to find the first buggy entity,

when we calculate the probability P_k, the entities with higher rank than k are supposed to have been checked with developers not finding any bugs.

The probability of the k^{th} ranked entity not being buggy in the examining process is defined as P_k':

$$P'_k = 1 - P_k \quad (1)$$

We use the (1) to simplify the formulas in the bug localization simulation process. Since there is no guarantee that the buggy entities will be or not be found in the rank k, we have the following assumption:

$$\forall k, 0 < P_k < 1, \quad and\ 0 < P'_k < 1 \quad (2)$$

The localizing result in Section IV.B shows, P_k in high rank is larger than that in low rank overall. Especially P_1 has the largest value because Top-1 metric always occupy the main portion. So we have the following assumption.

Assumption 8. *Monotonicity: The probability of entities in the rank k being buggy in the examining process, P_k, is monotonically decreasing. And* $\forall k, 0 < P_k < 1, \ and\ 0 < P'_k < 1$.

The monotonicity will be proven in the real data shown in next sub section. And this assumption is an important part in the strategy derivation.

B. The Real Data

BugLocator dataset [9] have been used in several bug localization approaches [4], [6], [8], and authors provide tools and data. We analyzed the results of BugLocator algorithm including SWT [1] and Eclipse [2] projects, and counted the numbers of buggy entities for each rank in the result list. For each bug report, we only count the entity with highest rank, as the developers only need to find the first buggy entity (Assumption 1). The numbers of buggy entities for each rank in the result list are shown in Fig. 1 and Fig. 2 in the two projects. The figures record the count of buggy source code entities across all bug reports in a project. To avoid undefined number (logarithm of zero) in logarithm scale in the figures, all number of buggy entities are added by 1.

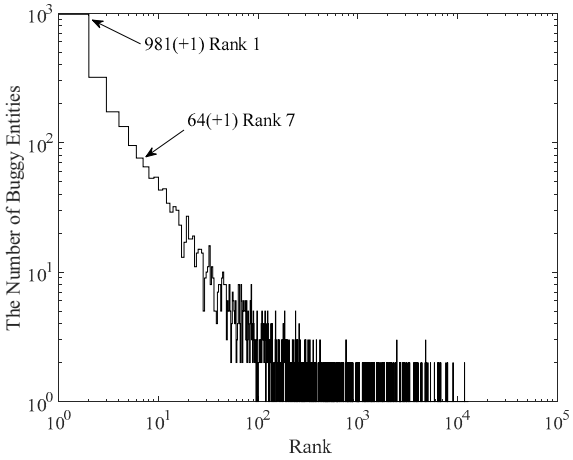

Figure 1. The number of buggy entities in the ranked list – Eclipse

(The value has been added by 1 to avoid log(0))

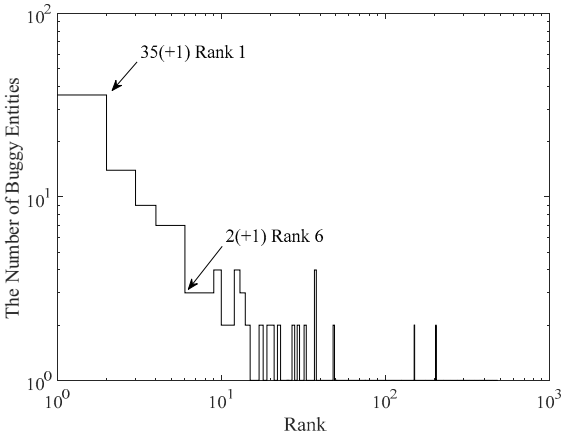

Figure 2. The number of buggy entities in the ranked list – SWT

(The value has been added by 1 to avoid log(0))

From Fig. 1 and Fig. 2, we can see in both projects, the count of buggy entities are monotonically decreasing with the rank of the entities at the result list, where the count of buggy entities can be easily transformed to the probability being buggy in the next sub section.

C. Classical Probablity Model

A simple way to use the localized bug data to calculate the probability P_k is using the classical probability model as follows:

$$P_k = \begin{cases} \dfrac{N_1}{N}, & k = 1 \\ \dfrac{N_k}{N - \sum_{i=1}^{k-1} N_i}, & k > 1 \end{cases} \quad (3)$$

where N is the total number of buggy entities, N_i is the number of buggy entities in the rank i in the result list across all bugs shown in Fig.1 & 2. $(N - \sum_{i=1}^{k-1} N_i)$ is the number of candidate entities being buggy, since the previous $\sum_{i=1}^{k-1} N_i$ entities have been examined and labeled to be buggy entities in higher rank. N=3075 for eclipse, and N=98 for SWT.

This is a rough deviation method for P_k. Especially when k is large, most number of buggy N_k at low rank will be 0 as shown in Fig. 1 and Fig. 2. It is not realistic to use this method to derive P_k in the very low ranks. But in our strategy, we only need to calculate P_k in the top ranks. The calculation of both projects P_k values using the model in (3) is shown in Fig. 3 and Fig. 4, respectively. All P_k values in Fig. 3 and Fig. 4 are added by 0.001 to avoid the logarithm of zero, and this procedure will not affect the original form of the curve but make the figure clearer.

Fig. 3 and Fig. 4 also provide the data support to Assumption 8. According to (3), when $N_k = 0$, $P_k = 0$. The P_k values 0.001 (has been added by 0.001 to avoid log(0)) in the Fig. 3 and Fig. 4 indicate the corresponding number of buggy entities N_k is zero. We also know that the last $P_k = 1$, because there is only one candidate entity to choose, which is shown as the right-most peak in the figures. When the rank is high, it is obvious P_k is monotonically decreasing with k. When the

[1] http://www.eclipse.org/swt/

[2] http://www.eclipse.org/

rank is low, since the denominator of (3) is getting lower, if the numerator of (3) $N_i > 0$, Pi will be quite large using (3). But actually, most numerator of (3) N_i in low ranks are zero, thus, using a smoothing technique will make the decreasing curve clear. Moreover in our strategy discussed in Section V, we will ignore the low ranks part, since the critical condition of the strategy will not happen in low ranks. (See Section V.B)

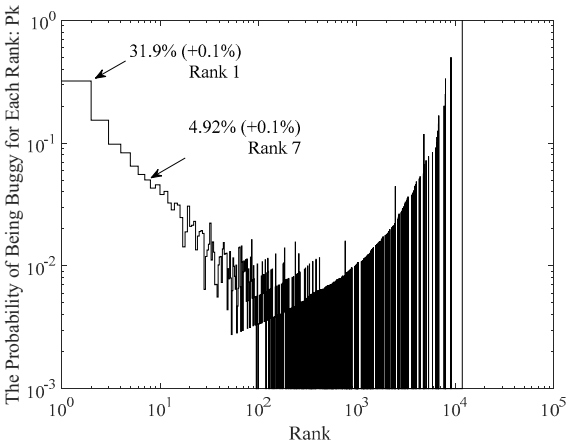

Figure 3. P_k for each rank – Eclipse
(The value has been added by 0.001 to avoid log(0))

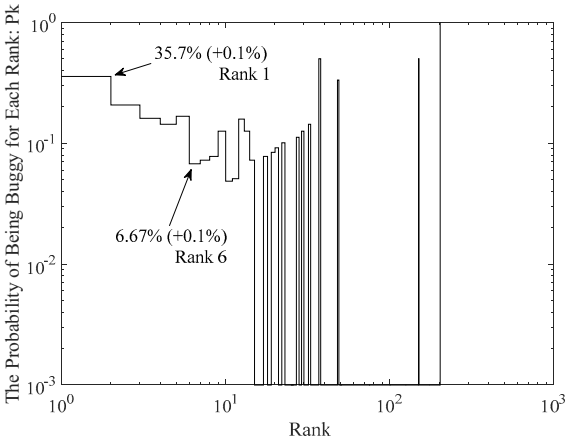

Figure 4. P_k for each rank – SWT
(The value has been added by 0.001 to avoid log(0))

V. Localization Process Simulation

In this section, we simulate the process of bug localization including the conventional localization approaches, and derived a criteria condition about when to stop using the automatic localization result.

A. Process Simulation

The cost of checking a source code entity at rank k of the list is defined as $Cost_{Find}(k;X)$, where X means: For a bug localization process for bug b, after X entities are examined in the ranked list and no buggy entities are found, the developer decides to stop using the ranked list and switch to conventional approaches. The goal of this model is to optimize the X value. If X value is set too small, the developer may switch to conventional approach too early to efficiently use the ranked list. If X value is set too large, the developer may spend too much time in checking the ranked list when the bug localization technique is not effective for this report.

The expected time cost when first buggy entity is found at rank k of the ranked list is defined as:

$$Cost_{Find}(k;X) = \begin{cases} TP_1 \ , & k = 1 \\ kTP_k \prod_{i=1}^{k-1} P_i' \ , & 1 < k \le X \end{cases} \quad (4)$$

If $k = 1$, the time cost equals T; the probability of finding the buggy entity at the first rank is P_1. If $1 < k \le X$, the time cost equals kT; Since there are k entities examined, the probability of finding buggy entity at the k rank is $P_k \prod_{i=1}^{k-1} P_i'$, since developers do not find buggy entity at the top i-1 entities, and finally find a buggy entity at rank k.

The expected time cost after the developer has examined X ($X \ge 0$) entities and not found any buggy entities is as follows:

$$Cost_{NotFind}(X) = \begin{cases} T_b \ , & X = 0 \\ (XT + T_b) \prod_{i=1}^{X} P_i' \ , & X > 0 \end{cases} \quad (5)$$

The situation $X = 0$ indicates the developers do not use the IR technique result and directly switch to a conventional approach, so the time cost is T_b. The probability $\prod_{i=1}^{n} P_i'$ indicates the probability of previous n entities not being buggy, and the $(XT + T_b)$ is time cost of examining X entities and using conventional methods to localize bugs, and $\prod_{i=1}^{X} P_i'$ is the probability of not finding buggy entities in the top X ranked entities.

The expected time cost for bug report b with determination that stops using the ranked list after checking X entities in the list without finding buggy entities, is the sum of expectation of finding buggy entities at all the top X in the ranked list and the conventional localizing cost if no buggy entities is found in the top X entities:

$$Cost_{Total}(X) = \begin{cases} Cost_{NotFind}(X) \ , & X = 0 \\ \sum_{k=1}^{X} Cost_{Find}(k;X) + Cost_{NotFind}(X), & X > 0 \end{cases} \quad (6)$$

The average total cost is the evaluation metric ETC (Expected Time cost), we want to minimize. We use the $Cost_{Total}(X)$ to indicate the ETC for a specific bug, which can be distinguished from $Cost_{Find}(X)$ and $Cost_{NotFind}(X)$.

After substituting (1), (4) and (5) into (6), we derive:

When $X = 0$,

$$Cost_{Total}(X) = T_b \quad (7)$$

When $X = 1$,

$$Cost_{Total}(X) = TP_1 + (T + T_b)P_1' = T + T_bP_1' \quad (8)$$

When $X > 1$,

$$Cost_{Total}(X) = TP_1 + T\sum_{k=2}^{X}(kP_k\prod_{i=1}^{k-1}P_i') + (XT + T_b)\prod_{i=1}^{X}P_i' = T\left(1 + \sum_{k=2}^{X}\prod_{i=1}^{k-1}P_i'\right) + T_b\prod_{i=1}^{X}P_i' \quad (9)$$

B. Optimization of ETC

The goal of this model is to find out the optimized X value X_{ideal} which makes the $Cost_{Total}(X)$ minimum:

$$X_{ideal} = \arg\min_{X} Cost_{Total}(X) \quad (10)$$

To optimize the total time cost, we use the difference:

$$\Delta Cost_{Total}(X) = Cost_{Total}(X+1) - Cost_{Total}(X) = \begin{cases} T - T_b P_1' , & X = 0 \\ (T - T_b P_{X+1}) \prod_{i=1}^{X} P_i' , & X > 0 \end{cases} \quad (11)$$

We set $\Delta Cost_{Total}(X) = 0$ in (11), and get the critical value of P_{X+1} with the condition $\forall i, P_i' > 0$, under $X \geq 0$:

$$P_{X+1} = \frac{T}{T_b} \quad (12)$$

With (11) and (12), we can derive:

$$If\ P_{X+1} < \frac{T}{T_b}, \Delta Cost_{Total}(X) > 0; \quad (13)$$

$$If\ P_{X+1} > \frac{T}{T_b}, \Delta Cost_{Total}(X) < 0 \quad (14)$$

According to Assumption 8, P_{X+1} is monotonically decreasing in terms of X when the rank is high, so with (11), $\Delta Cost_{Total}(X)$ is monotonically increasing in terms of X. In the localizing results of previous study, usually a large proportion of buggy source code entities are ranked at the top of the list provided by the IR-based approaches, and with increasing X, P_X is approximately 0. This means inequality $P_1 > T/T_b$ is most likely going to hold true, and $P_N < T/T_b$ will also be true in the overwhelming majority of cases, where N is the number of all source code entities.

Then $\Delta Cost_{Total}(X) < 0$, when $X = 0$. With increasing X, P_X is decreasing, and $\Delta Cost_{Total}(X)$ is getting larger. When P_X is decreasing and finally smaller than T/T_b, then $\Delta Cost_{Total}(X) > 0$ while $\Delta Cost_{Total}(X-1) < 0$, and $Cost_{Total}(X)$ get the extreme minimum value. The bug localization goal is minimize the cost of localization task. So the X_{ideal} is the minimum X, which makes $P_{X+1} < T/T_b$ true:

$$X_{ideal} = \min\{X \mid P_{X+1} < \frac{T}{T_b}, X \geq 0, X \in Z\} \quad (15)$$

If $P_1 < T/T_b$, then $\forall X, P_X < T/T_b$, because P_X is assumed to be monotonically decreasing. So according to (13), $\forall X$, $\Delta Cost_{Total}(X) > 0$, $Cost_{Total}(X)$ is monotonically increasing. In this condition, Equation (15) is still true, and $X_{ideal} = 0$. In other words, if $P_1 < T/T_b$, the performance of automatic localization technique is so bad that developers need not to examine the ranked list, but use conventional approaches instead.

To estimate the value of X_{ideal}, the probability distribution of P_X should be revealed. An estimation of P_X in historical bug data has been introduced in Section IV.C. Since the rank X will be high in this strategy (see Section VI.A), P_X can directly be derived using (3) in high ranks without worrying about the unstable value of (3) in low ranks.

VI. DISCUSSION

A. Comparision of Using and not Using the Strategy

In this section, we compare the performance of a state-of-art technique BugLocator [9] with using the strategy introduced in this paper and without using the strategy. We use expected time cost ETC as the evaluation metric. The comparison is done using the BugLocator algorithm on Eclipse and SWT projects introduced in the previous Section.

The time cost of the conventional approach needs to be estimated by developers. For simplicity, we set the time cost of conventional approach T_b in this comparison study to $20T$ for all bugs in Eclipse project. According to the critical condition (12), $P(X+1)$=1/20. It means when the probability fall below 1/20 in the examination process from the top of ranked list, developers are suggested to stop using the ranked list. We use the P_X calculated from the classical probability model (3). For Eclipse project, X_{ideal} = 6, since the $P(6+1)$=4.92%, (see Fig. 3) which is the first time $P(X+1)$ becomes less than 1/20. The number of examined source code entities (time cost) in real case for different X values are shown in Fig. 5. The value of X is the rank to stop using automatic methods and switch to a conventional approach.

In the real case of Eclipse, X_{ideal} is also equal to 6, when examining the 12.17 source entities to localize a bug on average. If using only a conventional approach, then the average ETC is $T_b = 20T$. If we only use the bug localization result without using our strategy, we will need to examine 141.8 entities on average.

In real case of SWT, we set $T_b = 10T$, critical condition is $P(X+1) = 1/10$. $P(5+1) = 6.67$ (see Fig.4) since that is the first time $P(X+1)$ becomes less than 1/10. The real number which needs to be examined to fix the bug is shown in Fig. 6. $X_{ideal} = 5$ for both theory and data, and developers only need to examine 6.01 source code entities on average to localize a bug in SWT. When only using automatic methods, developers need to examine 10.03 entities to localize a bug on average. Only using conventional approaches will cost developers more to examine 10 entities for each bug.

We can conclude the performance comparison in two projects: Using our strategy (ETC=12.17T, 6.01T) combining both automatic localization technique together with conventional approach performs better than using only either the automatic localization method (141.8T, 10.03T) or the conventional approach (20T, 10T).

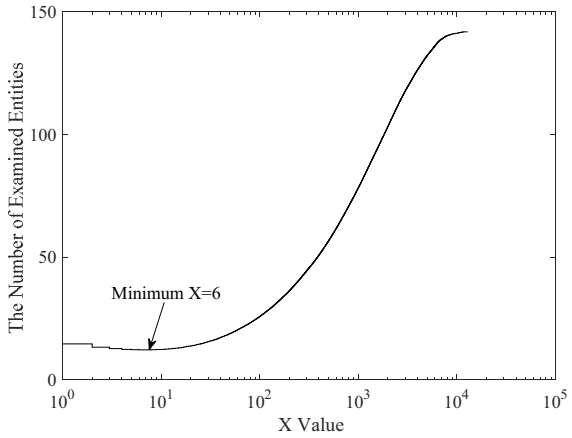

Figure 5. The number of entities need to be exmined to fix a bug on average with different X value – Eclipse (T_b=20T)

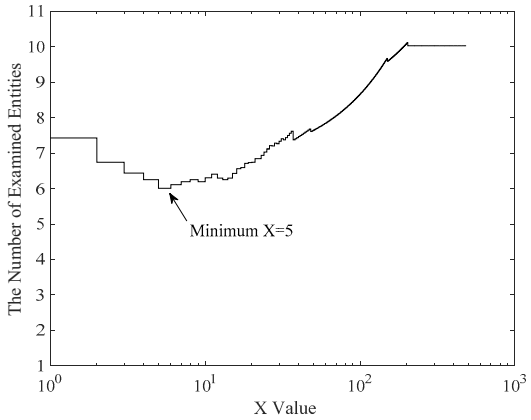

Figure 6. The number of entities need to be exmined to fix a bug on average with different X value – SWT (T_b=10T)

B. Threats to Validity

The critical condition is based on the expected time cost of conventional approach T_b. The conventional localization time cost T_b varies a lot under different conditions including developers, type of bugs, projects, etc. In this paper, we assume the value of T_b is estimated by developers using their experience. Precise estimation of T_b will induce the strategy to be more effective. We leave this as a future work. We assume examining each source code entity costs the same time. Previous approaches using Top-N, MAP and MRR have the same assumption. The time cost to examine an entity is also different, since length of an entity varies a lot.

VII. Related Work

In the early stage of bug location, researchers usually attempt different IR techniques in bug localization for example VSM (Vector space model) [3], LDA (Latent Dirichlet Allocation) [2], SUM (Smoothed Unigram Model) [3], and LSI (Latent Semantic Indexing) [13]. Recent research in bug localization usually focused on finding some additional features like version history [8], dynamic analysis [14] etc. These methods found other attributes besides textual similarity which can help in localizing bugs.

VIII. Conclusion

This paper introduces a simulated process of bug localization, which considers the situation of using a conventional localization approach if the automatic method is not effective in locating bugs, while previous automatic IR-based bug localization methods do not consider this situation. We assume developers are able decide when to stop using automatic localization result and switch to a conventional approach. In the simulated process, we proposed a novel evaluation metric ETC to indicate the expected total time cost to localize a bug, which also contains the time cost of using a conventional localization approach. The goal of this simulated process is to minimize the ETC for each bug. The criteria to minimize ETC is derived in quite simple form, which is in terms of the probability to localize the bug in the next entity of ranked list, and the expected time cost of a conventional approach to localize the bug. We compared the time cost of a state-of-art automatic localization method, BugLocator, with and without using our strategy in two projects. The result shows that using our proposed strategy combining both automatic localization technique together with conventional approach performs better than using only either the automatic localization technique or the conventional approach.

Acknowledgment

This research is supported by the City University of Hong Kong research funds (Project No. 7200354, 7004222, 7004474).

References

[1] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[2] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent Dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.

[3] S. Rao and A. Kak, "Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 43–52.

[4] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, 2013, pp. 345–355.

[5] P. Singh and S. Batra, "A Novel Technique for Call Graph Reduction for Bug Localization," *International Journal of Computer Applications*, vol. 47, no. 15, pp. 1–5, 2012.

[6] C. Tantithamthavorn, A. Ihara, and K.-I. Matsumoto, "Using Co-change Histories to Improve Bug Localization Performance," in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2013 14th ACIS International Conference on*, 2013, pp. 543–548.

[7] S. Wang, F. Khomh, and Y. Zou, "Improving bug localization using correlations in crash reports," in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, 2013, pp. 247–256.

[8] S. Wang and D. Lo, "Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization," in *Proceedings of the 22Nd International Conference on Program Comprehension*, 2014, pp. 53–63.

[9] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports," in *Software Engineering (ICSE), 2012 34th International Conference on*, 2012, pp. 14–24.

[10] S. W. Thomas, M. Nagappan, D. Blostein, and A. E. Hassan, "The Impact of Classifier Configuration and Classifier Combination on Bug Localization," *Software Engineering, IEEE Transactions on*, vol. 39, no. 10, pp. 1427–1443, 2013.

[11] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge university press Cambridge, 2008.

[12] Q. Wang, C. Parnin, and A. Orso, "Evaluating the Usefulness of IR-based Fault Localization Techniques," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 1–11.

[13] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification," in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, 2006, pp. 137–148.

[14] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval," *Software Engineering, IEEE Transactions on*, vol. 33, no. 6, pp. 420–432, 2007.