

Graph-Based Fault Localization

Béla Vancsics(✉)

Department of Software Engineering, University of Szeged,
Dugonics tér 13, Szeged 6720, Hungary
vancsics@inf.u-szeged.hu

Abstract. The subject of fault localization (FL) is a much-researched area, it has large literature. There are plenty of algorithms that try to identify the location of the bugs using different approaches. Debugging has a large resource requirement, therefore the bug's location's reflective identifiicaton greatly helps developers and testers to maintain the quality and reliability of the software.

Our goal is to implement a graph-based Fl *GFL* approach that effectively finds the location of the bugs in the source code. In our research we performed an empirical evaluation using the Defects4J and the results were compared with six other algorithms accepted by the literature. The results show that our method finds the errors more effectively than the other presented procedures, thus speeding up the bug fixes.

Keywords: Fault localization · Program spectra · Software testing · Program debugging

1 Introduction

Identifying bugs in the software is not a trivial task, so several methods have been developed to automatically support the fault localization process. That is to detect the error in the code.

We need tests to know if there are bugs in the code. The number of tests, number of methods and code coverage ratio greatly influence the quality and reliability of our program [9,21], also the effectiveness of fault localization methods.

There are a lot of methods that try to predict the location of bugs by some approach ([10,16,32]), but it is not possible to have 100% accuracy in each case. There are different "levels of identification", such as statement, branch or function levels and these can influence the effectiveness of the algorithms. These products a sequence of suspicions between the code elements and the developer checks the source code according to this order.

Examining and validating the effectiveness of the methods is not a trivial issue. There are two types of benchmarks to "compete" with algorithms:

1. the code contains injected errors: Developers intentionally corrupt the code manually or by using some kind of (e.g.: mutation) tool [1,3,31]. Using this

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11622, pp. 372–387, 2019.
https://doi.org/10.1007/978-3-030-24305-0_28

method (or benchmark), the researcher can verify the results of fault localization algorithms on unknown software, because it can be known exactly where the bug is, as it is caused (by the researcher or the tool) In addition, it is an advantage that some "fault groups" (e.g.: incorrect *if* condition) can be produced and analyzed in large quantities.
2. the code contains real bugs: bugs, that are mined from the version tracking (and issue system), are used for evaluation [5,12,17]. The exact location of the patch can be traced back and usually there is a bug fixing test. These types of errors provide a more realistic picture of the effectiveness of the methods.

Studies have shown that there are large discrepancies between result of algorithms if they analyze real errors or injected errors.

In our research we present a new, graph-based FL method that was compared to 6 other metrics [15] using real-bugs [11].

2 Related Work

The fault localization is a much-researched area with extensive literature [17,24,25]. There are a lot of algorithms, all of which are intended to determine the exact location of the bugs. These can be grouped based on their "nature" and the data used [27]:

- Static, Dynamic and Execution Slice-Based Methods (eg. [33])
- Program Spectrum-based Methods (eg. [10,20])
- Statistics-based Methods (eg. [13])
- Program State-based Methods (eg. [22])
- Machine Learning-based Methods (eg. [6,26])
- Model-based Methods (eg. [14])
- Data Mining-based Methods (eg. [7])

One of the largest categories is the Spectrum-Based FL (SBFL). The essence of the method is that the collected execution data can be deduced from the behavior of the program and thus on the possible location of the bug.

One of the most popular SPFL methods is Tarantula [10,34]. It prioritizes the methods using the coverage information and the test results. The algorithm orders the methods based on the ratio of the executed failed tests and the not executed failed tests, as well as the executed passed and do not executed tests.

Renieris and Reiss [18] presented a method which is based on the differences between the demeanor of pass and failed test. The method selects the correct execution according to the distance criterion, which is the most similar to the incorrect run, compares the spectra of the two runs and sets the order based on the "suspicious program elements".

The first failed test does not necessarily have the same information as the second or third and so on. The same applies to pass tests. Wong et al. [28] introduced an algorithm, which gave better results to the SIR[1] C program benchmark than many earlier (eg Tarantula, Liblit05 and SOBER) metrics.

[1] https://sir.csc.ncsu.edu/portal/index.php.

Abreu et al. has used the Ochiai method in their studies ([2–4]). This metric was adapted from molecular biology. It was shown that Ochiai produces better results than Tarantula using the Siemens and the SIR (see Footnote 1) bug dataset.

Wong et al. [23] presented the DStar technique, which was evaluated in 24 programs and compared the results of the algorithm with 38 different techniques. Single-fault and multi-fault programs are used for assessment. Empirical evaluation has shown that DStar is better than all other methods.

There are many comparative studies [3,15,23] that compare the results of different algorithms. To sum up, these studies came to the following conclusions: (1) there is difference inefficiency injected and real bugs (2) Ochiai performed better than Tarantula (3) DStar was better than Ochiai.

3 Overview and Goal

The aim is to create a new graph-based FL method that uses collected per-test, method level coverage information and test result data. In addition, the research includes comparing results with other FL algorithms.

3.1 Method

The process consists of four components (Fig. 1). These steps are necessary to evaluate the effectiveness of the methods.

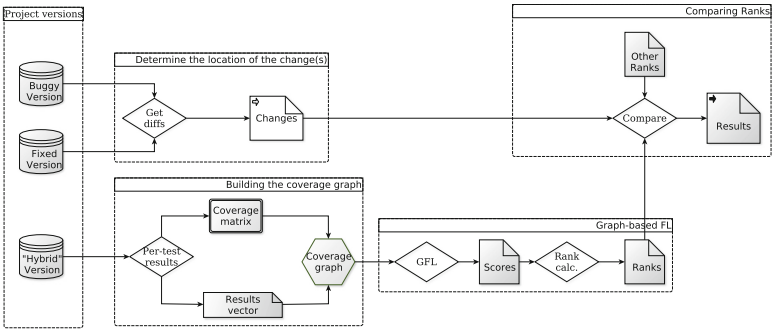

Fig. 1. Overview

(a) **Determination of change set**: We need to know which methods have been changed by the bug-fix. These methods are considered to be faulty, these will be the location of error. This can be extracted by the difference between the buggy and fixed version of the source code.

(b) **Building the graph**: A graph is built from the coverage matrix, where the nodes are the tests (T) and the methods (M). There is an edge between $m \in M$ and $t \in T$, if m is covered by t.

(c) **GFL algorithm**: On the basis of the neighbor-conditions in the graph and the test results, we determine the order of suspicion of the methods. The more efficient the algorithm is, the higher the faulty methods are in the order.
(d) **Comparison**: We examine the different methods about how effectively detect the location of bugs. There are several options: average ranks (project-level or in all bugs) and relationship of metrics to each other.

The differences in efficiency between algorithms are quantified from the results after the fourth step and can be used to infer the effectiveness of the methods.

3.2 Subjects

For evaluation, we need validated, identifiable and fixed bugs. *Defects4J* was used in our research. It is a framework that has collected bug data for real Java projects. High-level's interface provides options to achieve different states of a bug (for example: buggy or fixed version) and it allows to run test cases. The bug DB contains relevant, reproducible and isolated errors [11].

(a) "The bug is related to source code: a developer explicitly labelled the commit of fixed version (V_{fix}) as a bug fixing commit, and the bug fix applies to the source code — bug fixes within the build system, configuration files, documentation, or tests are not included."
(b) "The bug is reproducible: V_{fix} is accompanied by at least one test that passes on V_{fix} but fails on Vbug, and the bug is reproducible using the project's build system and an up-to-date JVM (Java 7)"
(c) "The bug is isolated: the bug fix (i.e., the diff between V_{bug} and V_{fix}) does not include unrelated changes such as features or refactorings."

3.3 Research Questions

Three research questions are formulated about FL-efficiency. These evaluate algorithms in two ways: based on the rankings of the bugs and the relations between the ranks.

- ***RQ1:*** Does *GFL* perform better ranks than other metrics?
  We examine the average rankings of the new FL method. (Does it find the bugs "sooner" than the other methods or not?)
- ***RQ2:*** Does *GFL* give several times better (and less often worse) values than the other metrics?
  We examine how many cases give *GFL* better results than other metrics.
- ***RQ3:*** Does *GFL* result in a relevant improvement?
  We examine how many cases result the *GFL* algorithm in a "quality improvement"

The answers to the questions give an objective picture of the effectiveness of the algorithms.

4 Spectrum-Based Fault Localization

Several FL methods are available in the literature, which can be classified into different groups by their characteristics [25]. For example: Program State-based, Machine Learning-Based, or Program Spectrum-Based Methods.

One of the most popular algorithm-group is Program Spectrum-Based Methods (PSBM), which capture program execution data (such as executable information), and thus monitor the behavior of the program. The list of suspicious elements are determined based on dynamic information provided by tests, however, not only faulty test cases can provide useful information [20], but also pass tests [28]. Studies have shown that algorithms that take into account both successful and unsuccessful tests are more effective than those focusing on unsuccessful tests.

4.1 Coverage and Result Data

One way to store this dynamic information is the coverage matrix whose rows represent the tests (T), and columns show the (source) code elements (for example: method - M). The matrix has 1 value in a given $x_{i,j}$ position if the m_j code element is covered by t_i test, otherwise this value is 0.

With another data structure (with the result vector) the results of the tests are stored (where if the t_i test is fail then it gets 0, otherwise 1): In our research, we worked with method-level granularity, code elements designated methods. These data provide the information they need to build the graph.

$$COV = \begin{bmatrix} x_{1,1} & \dots & x_{1,l} \\ \vdots & \ddots & \vdots \\ x_{k,1} & \dots & x_{k,l} \end{bmatrix}, \; RES = \begin{bmatrix} y_1 \\ \vdots \\ y_k \end{bmatrix} \quad \begin{array}{c} x_{a,b} \in \{0,1\} : \; a \in T \; b \in M \\ x_{i,j} = 1 \text{ if } m_j \text{ is covered by } t_i \\ y_a \in \{0,1\} : \; a \in T \\ y_a = 1 \text{ if } t_i \text{ is pass, else } 0 \end{array}$$

4.2 Metrics

A special case for PSBM is the Executable Statement Hit Spectrum (ESHS) [18]. It collects which methods (or statements or branches) have been executed by the tests. We focused on the method level in this paper.

For a m method, it assigns four values based on the coverage and the result information

(a) m_{ep}: number of passing tests covered by m
(b) m_{fp}: number of failing tests covered by m
(c) m_{np}: number of passing tests not covered by m
(d) m_{nf}: number of failing tests not covered by m

These four values are used by many formula [19,30], thus trying to achieve the best results. In this paper, the GFL algorithm's results were compared with 6 popular metrics.

Barinel [3]: $1 - \frac{m_{ep}}{m_{ep} + m_{ef}}$, DStar [24] : $\frac{m_{ef}^2}{m_{ep} + (m_{ef} + m_{nf}) - m_{ef}}$

Jaccard [8]: $\frac{m_{ef}}{m_{ef} + m_{nf} + m_{ep}}$, Ochiai [2]: $\frac{m_{ef}}{\sqrt{(m_{ef} + m_{nf}) \cdot (m_{ef} + m_{ep})}}$

Op2 [15]: $m_{ef} - \frac{m_{ep}}{m_{ep} + m_{np} + 1}$, Tarantula [10]: $\frac{\frac{m_{ef}}{m_{ef}+m_{nf}}}{\frac{m_{ef}}{m_{ef}+m_{nf}} + \frac{m_{ep}}{m_{ep}+m_{np}}}$

5 Graph-Based FL (GFL)

Our neighbors-based fault localization method differs somewhat from the methods described above. The input data required for the method is the coverage matrix and the result vector, but *GFL* cannot be written with only the four numbers used.

An example can be seen in the Fig. 2 that contains 7 methods and 8 tests. Two of the tests are failed ($t1$ and $t2$), the others are pass.

$$\mathbf{C} = \begin{array}{c} \\ t1 \\ t2 \\ t3 \\ t4 \\ t5 \\ t6 \\ t7 \\ t8 \end{array} \begin{array}{c} \begin{array}{ccccccc} m1 & m2 & m3 & m4 & m5 & m6 & m7 \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \quad \mathbf{R} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

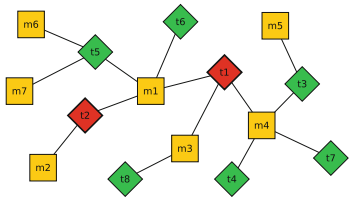

Fig. 2. Coverage matrix, result vector and coverage graph

5.1 Algorithm

As the first step, we build a graph from the coverage matrix. Tests and the methods will be the nodes, and $t_i \mapsto m_j$ edge exists if an m_j ($\in M$) is covered by the t_i ($\in T$), otherwise it does not.

GFL works as though the failed tests would infect the methods that cover them. If a method has many connections with failed tests and the proportion of these tests is high compared to the pass tests it is suspicious. In addition, we examine how the passed-failed test pairs resemble each other and we use it to create the rank.

The process consists of six steps, the result of which will be the scores assigned to the method.

Every method has a start value (score = 1.0). Then we continue the algorithm with the following steps:

1. "Specificity of tests" (Fig. 3): Number of methods and number of covered methods (for $t_i in T$) are devided. If this value is high, it indicates that the failed t_i test only covers a small part of the methods, ie the tests are "method specific". We propagandize these values on the $t_i \mapsto m_j$, it will be the weight of the $t_i \mapsto m_j$ edge.

$$t_i \mapsto m_j = \frac{\|M\|}{\|CM_t\|}$$
$$t_i \in T_F,\ m_j \in M$$
$$T_F = \{t \in T : R_t = 0\}$$
$$CM_t = \{m \in M : C_{t,m} = 1\}$$

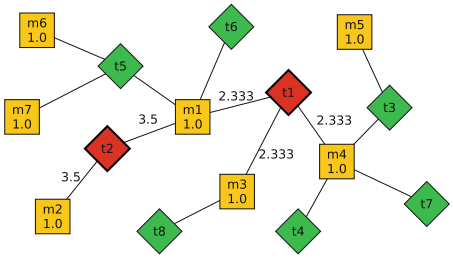

Fig. 3. "Specificity of tests"

2. "Projection" (Fig. 4): These values are projected to the method-edges. It determines the average value of $t_i \mapsto m_j$ edge weight.

$$t_i \mapsto m_j = \frac{t_i \mapsto m_j}{|CPT_{m_i} \cup CFT_{m_i}|}$$
$$T_P = \{t \in T : R_t = 1\}$$
$$CFT_{m_j} = \{t \in T_F : C_{t,m_j} = 1\}$$
$$CPT_{m_j} = \{t \in T_P : C_{t,m_j} = 1\}$$

3. "Aggregation" (Fig. 5): The $t_i \mapsto m_j$ edge weight will be aggregated into the scores of methods.

$$score_{m_j} = \sum_{t \in CFT_{m_j}} t \mapsto m_j$$
$$CFT_{m_j} = \{t \in T_F : C_{t,m_j} = 1\}$$

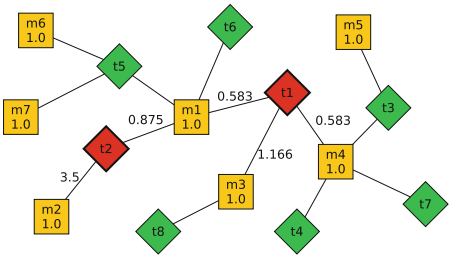

Fig. 4. "Projection"

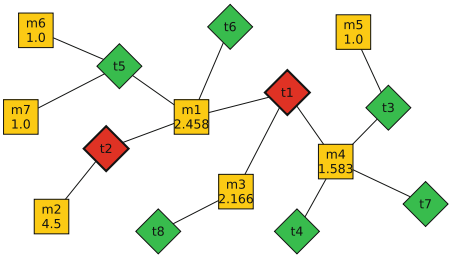

Fig. 5. "Aggregation"

4. "Responsibility" (Fig. 6): We calculate the ratio of the number of failed edge (from m) and the number of total failed edge.

$$score_{m_j} = score_{m_j} \cdot \frac{\left|CT_{m_j} \cap T_F\right|}{|FE|}$$

$$CT_{m_j} = \{t \in T : C_{t,m_j} = 1\}$$

$$FE = \{(t, m) : t \in T_F, m \in M,\ C_{m,t} = 1\}$$

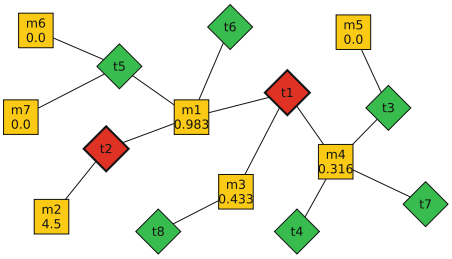

Fig. 6. "Responsibility"

5. "Paired comparison" (Fig. 7): We examine all passed-failed test pairs and increase the pScore of those methods (pScore initially 0.0), which were only covered by the failed test. (The figure shows only two test-pairs)

$$pScore_{m_j} = \sum_{\substack{f_{m_j} \in CFT_{m_j} \\ p_{m_j} \in NCPT_{m_j}}} \frac{\left| CM_{f_{m_j}} \setminus (CM_{p_{m_j}} \cap CM_{f_{m_j}}) \right|}{\left| CM_{f_{m_j}} \right|}$$

$$NCPT_{m_j} = \{t \in T_P : C_{m_j,t} = 0\}$$

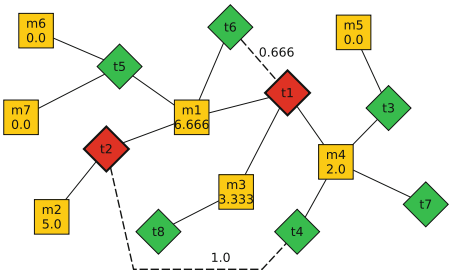

Fig. 7. "Paired comparison"

6. "Averaging" (Fig. 8): The value associated with the method is divided by the number of covered failed tests.

$$pScore_{m_j} = \frac{pScore_{m_j}}{\left| CFT_{m_j} \right|}$$

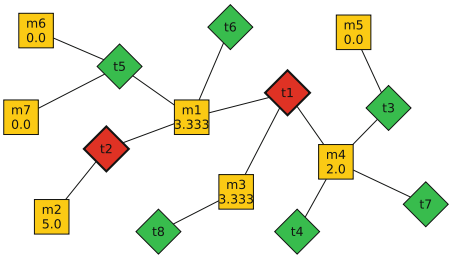

Fig. 8. "Averaging"

The six steps described above can be summarized in the following formula:

$$GFL_{m_j} = \frac{\sum\limits_{t \in CFT_{m_j}} \frac{\frac{|M|}{|CM_t|}}{|CPT_{m_j} \cup CFT_{m_j}|} \left|CT_{m_j} \cap T_F\right|}{|FE|} \sum_{\substack{f_{m_j} \in CFT_{m_j} \\ p_{m_j} \in NCPT_{m_j}}} \frac{\left|CM_{f_{m_j}} \setminus (CM_{p_{m_j}} \cap CM_{f_{m_j}})\right|}{\left|CM_{f_{m_j}}\right| |CFT_{m_j}|}$$

The values thus obtained will be the basis of rank-order.

6 Evaluation

Defects4J was used to quantify the fault localization algorithms. With the help of the bytecode instrumentation tool, which is based on *Javassist*, the necessary information about the programs have been collected.

The minimum requirement for *Javassist* is Java 5, so we had to exclude older programs and those where the instrument was unable to perform a successful results (Table 1) [11].

Table 1. Subject programs

Project	KLOC	Tests	Bugs
Commons Lang	22	2 245	52
Commons Math	85	3 602	82
JFreeChart	96	2 205	24
Joda-Time	28	4 130	26
Total	231	12 182	184

Ranks were used as comparison basis, because the scores of metrics for the methods cannot be compared, for example interpretation range of *GFL* is not limited to $[0.0, 1.0]$ intervals. Bug-rank is the position of changed (buggy) method in suspicious order, which calculated based on FL scores. If there are more methods with the same score, they will get their average rank. Fixing the bug can affect more methods, in this case, the lowest value of the method ranks is taken into account. For example: if $m2$ and $m4$ are modified methods then the rank for this bug is 1, because $m2$ has lower rank (rank:1), than $m4$ (rank:4).

Comparisons have been made for each project based on ranking. Tables 2 and 3 show the results. It can be seen that in four cases (Commons Lang: 3,981, Joda-Time: 12,769, JFreeChart: 7,583 and all bugs: 7,416) the GFL gave the best average rank value, and it gets the 2th best result on Commons Math.

Looking at the results, it can be seen that the DStar produces rather extreme ranks. In many cases it gives very good value, but there are also extremely bad results and these have a negative effect on the average rank values. It is interesting that Ochiai gets the second best in complete bug data set, but it

Table 2. Average ranks

	Barinel	DStar	Jaccard	Ochiai	Op2	Tarantula	GFL
Commons Math	7,92	200,90	7,82	8,16	8,83	7,92	7,85
Commons Lang	4,84	168,24	4,37	4,26	4,20	4,84	3,98
Joda-Time	17,90	13,77	16,89	13,89	18,00	17,90	12,77
JFreeChart	12,19	280,5	8,73	8,23	35,65	12,19	7,58
The average of all	9,02	175,61	8,24	7,88	12,32	9,02	7,42

Table 3. Ranks of the average ranks

Project	Barinel	DStar	Jaccard	Ochiai	Op2	Tarantula	GFL
Commons Math	3,5	7	1	5	6	3,5	2
Commons Lang	5,5	7	4	3	2	5,5	1
Joda-Time	5,5	2	4	3	7	5,5	1
JFreeChart	4,5	7	3	2	6	4,5	1
Ranks based on all	4,5	7	3	2	6	4,5	1

produced once the second best result for a single project (and in other cases it was worse).

Another comparative option is to examine the rankings in pairs. There are three "options" for each pairing comparison: M_a is lower than M_b, M_a is equal to M_b, M_a is higher than M_b. M_a is better than M_b, if its rank is lower. For example: Op2 28 times gave smaller rank than Barinel and 109 times gave equal than DStar metric (Tables 4 and 5).

Table 6 shows the results per project received by comparison. It can be seen that for the DStar the number of draws is the lowest and there are more cases when a metric loses more than it wins (for example: DStar - Commons Lang or Jaccard - Joda-Time).

Table 4. Comparison of metric-pairs (number of non-equal ranks)

<	Barinel	DStar	Jaccard	Ochiai	Op2	Tarantula	GFL	**SUM**
Barinel		31	7	8	19	0	11	**76**
DStar	44		41	32	37	44	26	**224**
Jaccard	20	28		4	18	20	7	**97**
Ochiai	26	27	16		17	26	4	**116**
Op2	28	38	25	22		28	18	**159**
Tarantula	0	31	7	8	19		11	**76**
GFL	47	54	42	36	31	47		**257**
SUM	165	209	138	110	141	165	77	

Table 5. Comparison of metric-pairs (number of equal ranks)

=	Barinel	DStar	Jaccard	Ochiai	Op2	Tarantula	GFL	**SUM**
Barinel		109	157	150	137	184	126	**863**
DStar	109		115	125	109	109	104	**671**
Jaccard	157	115		164	141	157	135	**869**
Ochiai	150	125	164		145	150	144	**878**
Op2	137	109	141	145		137	135	**804**
Tarantula	184	109	157	150	137		126	**863**
GFL	126	104	135	144	135	126		**770**

Table 6. Comparison of metric-pairs: number of "action" (rank)

Project	"Actions"	Barinel	DStar	Jaccard	Ochiai	Op2	Tarantula	GFL
Common Math	Win	27	90	37	33	44	29	103
	Draw	412	323	411	419	385	412	352
	Defeat	51	79	44	40	63	51	37
Commons Lang	Win	20	55	26	36	50	20	67
	Draw	244	186	248	248	236	244	230
	Defeat	48	71	38	28	26	48	15
Joda-Time	Win	6	44	13	23	42	6	42
	Draw	111	96	109	112	98	111	103
	Defeat	39	16	34	21	16	39	11
JFreeChart	Win	21	35	21	24	23	21	45
	Draw	96	66	101	99	85	96	85
	Defeat	27	43	22	21	36	27	14
All	Win	76	224	97	116	159	76	257
	Draw	863	671	869	878	804	863	770
	Defeat	165	209	138	110	141	165	77

Overall, GFL won the most often (257) and lost the least (77). Interestingly, DStar was the second (224) and lost (209) the most often. It can be stated based on Tables 2 and 6 that DStar behaves rather extremely: often wins but also often produces extremely bad rank values, which greatly affects the rank values. The results in the tables clearly show that the graph-based algorithm is fairly balanced and it produces the best ranks.

According to some studies [29], developers examine only top-5 or top-10 suspicious elements during bug fixes. Therefore 3 groups were created: (1) errors' location is 5th or lower in ranking (2) errors' location is between 5th and 10th place (3) errors' location is worse than the 10th place. Table 7 shows what kind of movements happened, that is, the rank of a bug was transferred to another

group by the GFL algorithm. It seems that the result of GFL often puts the bug into a "better group" (a total of 58) and rarely makes it into a "worse group" (a total of 10).

Table 7. Rank improvement

	$[\infty, 10) \rightarrow [10, 5)$	$[\infty, 10) \rightarrow [1, 5]$	$[10, 5) \rightarrow [1, 5]$	$\sum$
$Barinel \rightarrow GFL$	2	1	4	7
$DStar \rightarrow GFL$	2	20	4	26
$Jaccard \rightarrow GFL$	1	1	3	5
$Ochia \rightarrow GFL$	2	0	2	4
$Op2 \rightarrow GFL$	1	8	0	9
$Tarantula \rightarrow GFL$	2	1	4	7
	$[10, 5) \rightarrow [\infty, 10)$	$[1, 5] \rightarrow [\infty, 10)$	$[1, 5] \rightarrow [10, 5)$	$\sum$
$Barinel \rightarrow GFL$	0	1	1	2
$DStar \rightarrow GFL$	0	0	4	4
$Jaccard \rightarrow GFL$	0	0	0	0
$Ochia \rightarrow GFL$	0	0	0	0
$Op2 \rightarrow GFL$	0	1	1	2
$Tarantula \rightarrow GFL$	0	1	1	2

Answers to previous questions are based on the results presented above:

RQ1: Results showed that four out of five cases the *GFL* gave the best results. It produced the lowest rank for three projects and aggregate data (Tables 2 and Table 3).

RQ2: The *GFL* won the most often (it was the second in only one project), it lost the least (in every case), it achieved the best result based on all and aggregated results (Table 4-Table 6).

RQ3: In any case, the *GFL* had larger improvements than it had losses. It has resulted in a total of 58 "quality improvements" and it was only 10 times worse than the "original quality group" (Table 7).

Acknowledgment. This study was supported by the project "Integrated program for training new generation of scientists in the fields of computer science", no EFOP-3.6.3-VEKOP-16-2017-0002. The project was supported by the European Union and co-funded by the European Social Fund.

References

1. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007), pp. 89–98 (2007)
2. Abreu, R., Zoeteweij, P., c. Van Gemund, A.J.: An evaluation of similarity coefficients for software fault localization. In: 2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC 2006), pp. 39–46, December 2006
3. Abreu, R., Zoeteweij, P., Gemund, A.J.C.v.: Spectrum-based multiple fault localization. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE 2009, pp. 88–99. IEEE Computer Society (2009)
4. Abreu, R., Zoeteweij, P., Golsteijn, R., van Gemund, A.J.: A practical evaluation of spectrum-based fault localization. J. Syst. Softw. **82**(11), 1780–1792 (2009). sI: TAIC PART 2007 and MUTATION 2007
5. B Le, T.D., Lo, D., Le Goues, C., Grunske, L.: A learning-to-rank based fault localization approach using likely invariants. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, pp. 177–188 (2016)
6. Brun, Y., Ernst, M.D.: Finding latent code errors via machine learning over program executions. In: Proceedings. 26th International Conference on Software Engineering, pp. 480–490, May 2004
7. Cellier, P., Ducassé, M., Ferré, S., Ridoux, O.: Formal concept analysis enhances fault localization in software. In: Medina, R., Obiedkov, S. (eds.) ICFCA 2008. LNCS (LNAI), vol. 4933, pp. 273–288. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78137-0_20
8. Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E.: Pinpoint: problem determination in large, dynamic internet services. In: Proceedings of the 2002 International Conference on Dependable Systems and Networks, pp. 595–604. IEEE Computer Society, Washington (2002)
9. Horváth, F., et al.: Test suite evaluation using code coverage based metrics. In: 14th Symposium on Programming Languages and Software Tools, SPLST 2015, CEUR-WS (2015)
10. Jones, J.A., Harrold, M.J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005, pp. 273–282. ACM, New York (2005)
11. Just, R., Jalali, D., Ernst, M.D.: Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, pp. 437–440. ACM, New York (2014)
12. Laghari, G., Murgia, A., Demeyer, S.: Fine-tuning spectrum based fault localisation with frequent method item sets. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, pp. 274–285. ACM, New York (2016)
13. Liu, C., Fei, L., Yan, X., Han, J., Midkiff, S.P.: Statistical debugging: a hypothesis testing-based approach. IEEE Trans. Softw. Eng. **32**(10), 831–848 (2006)
14. Mateis, C., Stumptner, M., Wotawa, F.: Modeling java programs for diagnosis. In: Proceedings of the 14th European Conference on Artificial Intelligence, pp. 171–175. IOS Press (2000)

15. Naish, L., Lee, H.J., Ramamohanarao, K.: A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol. **20**(3) (2011)
16. Nessa, S., Abedin, M., Wong, W.E., Khan, L., Qi, Y.: Software fault localization using N-gram analysis. In: Li, Y., Huynh, D.T., Das, S.K., Du, D.-Z. (eds.) WASA 2008. LNCS, vol. 5258, pp. 548–559. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88582-5_51
17. Pearson, S., et al.: Evaluating and improving fault localization. In: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, pp. 609–620. IEEE Press, Piscataway (2017)
18. Renieres, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: 18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings, pp. 30–39, October 2003
19. Sohn, J., Yoo, S.: Fluccs: Using code and change metrics to improve fault localization. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017, pp. 273–283. ACM, New York (2017)
20. Taha, A.B., Thebaut, S.M., Liu, S.S.: An approach to software fault localization and revalidation based on incremental data flow analysis. In: Proceedings of the Thirteenth Annual International Computer Software Applications Conference, pp. 527–534 (1989)
21. Tengeri, D., et al.: Relating code coverage, mutation score and test suite reducibility to defect density. In: 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 174–179 (2016)
22. Wang, T., Roychoudhury, A.: Automated path generation for software fault localization. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005, pp. 347–351. ACM, New York (2005)
23. Wong, W.E., Debroy, V., Gao, R., Li, Y.: The DStar method for effective software fault localization. IEEE Trans. Reliab. **63**(1), 290–308 (2014)
24. Wong, W.E., Debroy, V., Li, Y., Gao, R.: Software fault localization using DStar (D*). In: 2012 IEEE Sixth International Conference on Software Security and Reliability, pp. 21–30 (2012)
25. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Trans. Softw. Eng. **42**(8), 707–740 (2016)
26. Wong, W.E., Shi, Y., Qi, Y., Golden, R.: Using an RBF neural network to locate program bugs. In: 2008 19th International Symposium on Software Reliability Engineering (ISSRE), pp. 27–36 (2008)
27. Wong, W.E., Debroy, V.: A survey of software fault localization. Department of Computer Science, University of Texas at Dallas, Technical report UTDCS-45 9 (2009)
28. Wong, W.E., Debroy, V., Choi, B.: A family of code coverage-based heuristics for effective fault localization. J. Syst. Softw. **83**(2), 188–208 (2010). Computer Software and Applications
29. Xia, X., Bao, L., Lo, D., Li, S.: "Automated debugging considered harmful" considered harmful: a user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 267–278. IEEE, October 2016
30. Xie, X., Chen, T.Y., Kuo, F.C., Xu, B.: A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans. Softw. Eng. Methodol. **22**(4), 31:1–31:40 (2013)

31. Xuan, J., Monperrus, M.: Test case purification for improving fault localization. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pp. 52–63. ACM, New York (2014)
32. Zeller, A.: Isolating cause-effect chains from computer programs. SIGSOFT Softw. Eng. Notes **27**(6), 1–10 (2002)
33. Zhang, X., He, H., Gupta, N., Gupta, R.: Experimental evaluation of using dynamic slices for fault location. In: Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging, AADEBUG 2005, pp. 33–42. ACM (2005)
34. Zou, D., Liang, J., Xiong, Y., Ernst, M.D., Zhang, L.: An empirical study of fault localization families and their combinations. IEEE Trans. Softw. Eng. 1–1 (2019). https://doi.org/10.1109/TSE.2019.2892102