Statistical Fault Localization with Reduced Program Runs

Lina Hong, Rong Chen[1]

College of Informational Science and Technology,
Dalian Maritime University, 1 Linghai Road, Dalian 116026, P.R. China
{tsmc.dmu, rchen.cs}@gmail.com

Abstract. A typical approach to software fault location is to pinpoint buggy statements by comparing the failing program runs with some successful runs. Most of the research works in this line require a large amount of failing runs and successful runs. Those required execution data inevitably contain a large number of redundant or noisy execution paths, and thus leads to a lower efficiency and accuracy of pinpointing. In this paper, we present an improved fault localization method by statistical analysis of difference between reduced program runs. To do so, we first use a clustering method to eliminate the redundancy in execution paths, next calculate the statistics of difference between the reduced failing runs and successful runs, and then rank the buggy statements in a generated bug report. The experimental results show that our algorithm works many times faster than Wang's, and performs better than competitors in terms of accuracy.

Keywords: Software Fault Localization, Path Redundancy, Statistical Method, Clustering

1 Introduction

A typical thinking in fault localization is to compare successful runs and failing runs [2, 3, 4, 5, 6, 7]. There are different ways of comparison, and can be divided into distance measures-based methods [2, 3, 4] and characteristic statistics-based methods [5, 6, 7]. But both methods require a lot of failing runs and successful runs. Our analysis found that many characteristics of the runs are same.

Figure 1.1 shows a program example with a bug in the assignment statement on line 11, which should be "max = x;" In order to locate this buggy statement by using Wang [3] method, we run manual design of 12 test cases, and obtain 8 successful runs and 4 failing runs, in which there exists exactly same paths, such as max (4, 2, 3) and max (8, 5, 7). To be fair, for sorting methods students write, we execute randomly generated test cases, and statistics showed that the proportion of redundant paths is so much as 20% ~50%.

Redundancy leads to many problems: (1) Redundancy contributes nothing to the fault location calculations. If two failing runs are with the same path, it means that

[1] The corresponding Author. This work was supported by National Natural Science Foundation of China (60775028), Dalian Science &Technology Program (2007A14GX042), IT Industry Development of Jilin Province.

they contributions same to fault localization, so there is no need to separately calculate the difference between each of them and successful runs, the same applies two successful runs with two paths the same. (2) Computational efficiency of the system is reduced. Because the system may spend a lot of time doing pointless things. (3) The calculation of too much redundant data may lead to biased results. Thus removing the path redundancy is necessary.

Measures-based method returns executing differences as bug report, however the statements in differences are not ordered by their importance in literature [3]. Techniques based on characteristics statistics rank the statements or predicates in order of suspicious, but it is so many statements and predicates, even including those are not predictive of anything which is usually a majority, statistical work on them are time-consuming and effortless.

To overcome these shortcomings, in this paper, we first propose a clustering method to eliminate the path redundancy, and then through statistical analysis of differences between classes of runs to get and rank suspicious statements. Experimental comparison with Wang's algorithm shows that our algorithm makes a greatly improved efficiency and accuracy of locating. In addition, our algorithm can not only diagnose the reasons for each failing run, it can also get bug report of the whole program through statistical analysis of bug reports of all the failing runs.

```
package function;
public class ThreeNum{
  /**
 * Calculate the maximum of three numbers
 */
public int max(int x, int y, int z)
{
        int max;
        if (x > y)
      {
                 max = y; //should be max=x;
        }
        else
        {
                  max = y;
          }
          if (z > max)
          {
                  max = z;
            }
            return max;
      }
  }
```

Fig.1.A small program

Section 2 summarizes the related work; Section 3 introduces our fault localization method based on statistical differences between reduced runs; Section 4 shows comparative analysis of experimental results. Section 5 and gives the next step.

2 Related Work

Program runs statistical method is in essence a statistical intelligent method, how to make statistics and compare successful runs and failing runs attracted interest of many researchers [2, 3, 4, 5, 6, 7]. According to different type of comparison techniques, these methods can be divided into distance measures-based method and feature-based statistical approach.

Distance measures-based method is to find a successful run which is the most similar to the failing run through a certain distance measure technique, and then calculate the difference between it and failing run for fault location. A typical work of these methods is that Renieris [2] et al proposed the "Nearest Neighbor" and Wang [3] et al proposed calculating difference of two runs through alignment based-on control flow information.

In contrast, feature-based statistical approach locates fault-relevant statements (or faulty statements directly) by comparing the statistical information of program elements in these two kinds of run. Such program elements can be statements [5] or predicates [6, 9, 7]. Tarantula [5] statistics the frequency of every statement occurs in failing runs and successful runs, and by analyzing them to get and rank the suspicious statements. Predicate-based statistical techniques, such as CBI [6, 9] and SOBER [7], locate the program predicates related to faults. CBI [6, 9] measures the increase from the probability of a predicate to be evaluated to be true in all failed runs to that in all the runs, This increase is used as the ranking score, which indicates how much the predicate is related to a fault. SOBER [7] defines evaluation bias to estimate the chance that a predicate is evaluated to be true in each run. In brief, CBI and SOBER use similar kinds of statistical mean comparison.

3 Fault Localization With Reduced Run

3.1 Clustering Execution Paths

Next we define execution paths and their clustering.

Definition 1. (Event) An event denotes an execution of a statement of a given program. As [6] we use the line number *i* of a statement to label the event e_i associated with the statement, and the statement is thus denoted as $stmt(e_i)$. A test case that executes a given program provides the values of input variables and the expected values of output variables.

Definition 2. (Run) A run π of a program is a sequence of events $<e_0, e_1, e_2,\ldots, e_{n-1}>$ that are sequentially associated with statements executed by a given test case. Moreover, we denote the ith event of a run π as e_i^π and number of events of π as $|\pi|$.

Table 1. Test cases and execution runs of program in fig.1

output	Succ0	Succ1	Succ2	Succ3	Fail0	Fail1	Succ4	Succ5	Succ6	Succ7	Fail2	Fail3
TCinputs	2,3,4	2,4,3	3,2,4	3,4,2	4,2,3	4,3,2	5,7,8	5,8,7	7,5,8	7,8,5	8,5,7	8,7,5
Program runs	6_1	6_1	6_1	6_1	6_1	6_1	6_1	6_1	6_1	6_1	6_1	6_1
	8_2	8_2	8_2	8_2	8_2	8_2	8_2	8_2	8_2	8_2	8_2	8_2
	9_3	9_3	9_3	9_3	9_3	9_3	9_3	9_3	9_3	9_3	9_3	9_3
			11_4		11_4	11_4			11_4		11_4	11_4
	15_4	15_4		15_4			15_4	15_4		15_4		
	17_5	17_5	17_5	17_5	17_5	17_5	17_5	17_5	17_5	17_5	17_5	17_5
	19_6		19_6		19_6		19_6		19_6		19_6	
	21_7	21_6	21_7	21_6	21_7	21_6	21_7	21_6	21_7	21_6	21_7	21_6

Consider the program in Fig.1, we can get successful and failing runs showed in Table 1 by executing a set of test cases, where the *TC inputs* row sequentially put down the input values of variables x, y, and z within test cases (e.g., "2,3,4" means a test case having inputs {x=2, y=3, z=4}), while the *output* row describes that fact whether the TC inputs leads to a successful run or a failing run (The indexed Succ for successful runs, and Fail for failing runs), and program runs are put sequentially in columns below the corresponding test cases. For example, the TC inputs “2,3,4” yields a run $<6_1, 8_2, 9_3, 15_4, 17_5, 19_6, 21_7>$, where the subscripts denotes the event indices (e.g., 8_2 means that the statement on line 8 is executed once and is the second event of this run). Note that some cells in program runs are empty because we align statements for highlighting the difference in program runs.

Definition 3. (Dependence): Given a program P, let a variable v_1 be defined in statement S_1 and is used in statement S_2 defining value of a variable v_2. v_2 is dependent on v_1 if changes to v_1 in the execution will is likely to influence the definition and value of v_2. Equivalently, we also say statement S_2 is dependent on S_1. Moreover, v_2 is *control dependent* on v_1, if v_1 occurs in a condition the truth of which controls the execution from S_1 to S_2; otherwise, we say that variable v_2 is *data dependent* on v_1. Equivalently, we raise control/data dependence to the level of statements [8].

Definition 4. (Dynamic Control Dependence) Given a program run π, an event e_i^π is dynamically control dependent on another event e_j^π if e_j^π is the nearest event coming before e_i^π in π such that $stmt(e_i^\pi)$ is control dependent on $stmt(e_j^\pi)$. Moreover, we use $dep(e_i^\pi, \pi)$ to denote the events on which e_i^π is dynamically control dependent in a run π.

Definition 5. (Alignment [3]) For any pair of events e and e' (e in run π and e′ in run π'), we say e and e' are aligned, denoted as $align(e, e')=true$, iff (1) $stmt(e)=stmt(e')$, and (2) either e and e′ are the first events appearing in π and π' respectively, or $align(dep(e, \pi), dep(e', \pi'))=true$.

Definition 6. (Alignment Vector) Given two program runs π' and π, a alignment vector, denoted as $\Delta(\pi, \pi')$, is an array of marks for all events in π' and π, obtained as follows: for each event e of π, mark is 1 if there exists e' of π' which can aligned to e, otherwise mark is 0.

By definition 6, the following corollary is obvious:

Corollary 1. Given two program runs π' and π, $\Delta(\pi, \pi')= \Delta(\pi', \pi)$ holds。

We use Algorithm I to compute alignment vectors, where predicate $alignExist(i, \pi_1, \pi_2)$ is used to check whether there exists an event of π_2 that can be aligned to some e_i of π_1, and function $alignIndex(i, \pi_1, \pi_2)$ is used to get the index of such event of π_2. Moreover, function $\Delta.add(i)$ adds a mark i in the alignment vector Δ.

```
Algorithm I: delta(π1, π2)
input: two program runs π1 and π2
output: an alignment vector Δ(π1, π2)
t=1; j=0; //temporarily store the index
          //of events of π2
Δ={}; //the return result
outer:
for (i=1; i<=|π1|; i++){
    if alignExist(i, π1, π2) {
        j=alignIndex(i, π1, π2);
        k=j – temp2 – 1;
        for (n=0; n<k; n++) Δ.add(0);
        Δ.add(1);   //add a "1"
    } else {//otherwise add a "0"
        Δ.add(0); }
  t=j;
continue outer;
}
return Δ;
```

Definition 7. (Alignment Matrix) An alignment matrix of a program run π is a matrix composed of all alignment vectors $\Delta(\pi, \pi')$ such that π' is any run distinguish from π, and 0s are appended for padding if the length of alignment vectors are not constant.

By Table I and definition 6, we have (1) Δ(**fail0**,**fail0**)= <1,1,1,1,1,1,1>, (2) Δ(**fail0**,**fail1**)= <1,1,1,1,1,0,1>, (3) Δ(**fail0**, **fail2**)= <1,1,1,1,1,1,1>, and (4)

Δ(**fail0**,**fail3**)= <1,1,1,1,1,0,1>. With such an alignment matrix, we cluster the matrix by column and thus four failing runs in Table I can be divided into two classes:

F_I: fail$_0$(4,2,3), fail$_2$(8,5,7); F_{II}: fail$_1$(4,3,2), fail$_3$(8,7,5)

where fail$_i$(4,2,3) is abbreviated for the ith failing run with a TC inputs (4,2,3)). In the same manner, eight successful runs are divided into three classes:

S_I:succ$_0$(2,3,4), succ$_4$(5,7,8); S_{II}:succ$_1$(2,4,3), succ$_3$(3,4,2), succ$_5$(5,8,7) and succ$_7$(7,8,5);

S_{III}: succ$_2$(3,2,4), succ$_6$(7,5,8).

```
Algorithm II: diff($\pi_1$, $\pi_2$)
input: two program runs $\pi_1$ and $\pi_2$
output: a set Γof event indices representing
        difference between $\pi_1$ and $\pi_2$
Γ={};
Δ=delta($\pi_1$, $\pi_2$);
$n$=0; //to store the total number
       //of 0s coming before 1
for ($i$=0; $i$ <|Δ| − 1; $i$++) {
    if (Δ.get($i$)=0) $n$++;
    if (Δ.get($i$)=1∧Δ.get($i$+1)=0) {
//if the ith event can be aligned, while
//the next event can't be aligned
        Γ.add($i$+1−$n$+|Γ|); }
}
```

3.2 Difference Metric

Definition 8. (Difference Metric [3]) Given two program runs π and π'. The difference between π and π', denoted as $diff(\pi,\pi')$, is defined as $diff(\pi,\pi')$=<e_{i1}^{π}, e_{i2}^{π},…, e_{ik}^{π}>, such that (1) each event e in $diff(\pi,\pi')$ is a branch event occurrence drawn from run π, (2) the events in $diff(\pi,\pi')$ appear in the same order as in π, that is, for all $1 \leqslant j < k$, $i_j < i_j+1$ (event e_{ij}^{π} appears before event e_{ij+1}^{π} in π), (3) for each e in $diff(\pi,\pi')$, there exists another branch occurrence e' in run π' such that $align(e,e')=true$ (i.e. e and e' can be aligned). Furthermore, the outcome of e in π is different from the outcome of e' in π'. (4) All events in π satisfying (1) and (2) are included in $diff(\pi,\pi')$. (5) As a special case, if execution runs π and π' have the same control flow, then we define $diff(\pi,\pi')$=<e_0^{π}>.

By Definition 8, the statements in the difference are all branching statements from which two runs separately go to different branches. So if there are two adjacent values in the vector such that the former is a "1" and the latter a "0", then the event that "1" represents is added to the difference. In Algorithm II, Note that Δ.*get*(i) returns the ith bit of the alignment vector Δ, while Γ.*add*(i) adds a mark i in the alignment matrix Γ. Regarding our running example, differences are summarized in Table2 and Table 3:

Table 2. Difference between each class of failing runs and successful runs

	S_I	S_{II}	S_{III}
F_I	9_3	9_3,17_5	null
F_{II}	9_3,17_5	9_3	17_5

Table 3. Difference between each class of failing runs

	F_I	F_{II}
F_I	null	17_5
F_{II}	17_5	null

3.3 Fault Localization

For each predicate p in a program P, LIBLIT [6] estimates two conditional probabilities: （1）Pr1=Pr(P fails| p is ever observed); （2）Pr2=Pr(P fails| p is ever observed as true). LIBIT then thinks the difference Pr2−Pr1 as an indicator of how relevant p is to the fault. So we conclude that difference between failing runs and successful runs indicates that faults are more likely to appear in these positions, while difference between failing runs and failing runs indicates that the probability that fault appears in these position is very small, that is, the predicate being true can not increase the possibility of failure. Let $P(e,\pi,S)$ be the probability that event e of run π appears in the difference between π and successful runs, and $P(e,\pi,F)$ between π and failing runs. Then for any event e in a failing run π, we define the following $Score(e,\pi)$ to indicate the suspiciousness of statements $stmt(e)$. the formulas are as follows:

$$P(e,\pi,S)=\frac{\textit{times that e appear in difference between } \pi \textit{ and successful runs}}{\textit{the total number of events in difference between } \pi \textit{ and successful runs}} \quad (2)$$

$$P(e,\pi,F)=\frac{\textit{times that e appear in difference between } \pi \textit{ and failing runs}}{\textit{the total number of events in difference between } \pi \textit{ and failing runs}} \quad (3)$$

$$Score(e,\pi)=\begin{cases} P(e,\pi,S)\big/P(e,\pi,F), & P(e,\pi,F)\neq 0 \\ \infty, & otherwise \end{cases} \quad (4)$$

Given Table 1, Table 2 and (2), (3), and (4), we have:
$P(9_3,F_I,S)=2/3$, $P(9_3,F_I,F)=0$, and $Score(9_3,F_I)=\infty$;
$P(17_5,F_I,S)=1/3$, $P(17_5,F_I,F)=1$, and $Score(17_5,_{FI})=1/3$;
$P(9_3,F_{II},S)=1/2$, $P(9_3,F_{II},F)=0$, and $Score(9_3,F_{II})=\infty$;
$P(17_5,F_{II},S)=1/2$, $P(17_5,F_{II},F)=1$, and $Score(17_5,F_{II})=1/2$.

Then we get the rank score of suspiciousness that each event in the differences is the real cause of each failure. We summarize them in Table 4, where ∞(2/3) and the like denote the case of $P(e,\pi,F)=0$, $P(e,\pi,S)=2/3$, in which we compare suspiciousness by $P(e,\pi,S)$. If the rank score of two events are equal, we consider event that appears later has a larger suspiciousness score and will be ranked in the top [3].Assume a certain statement is responsible for a failure; it may be executed several times and appears several times in the differences. So we compute ranking scores of each statement for each failure as (5). So by (5), we get Table 5 of suspicious statements for each class of failures in our running example.

$$Score(stmt,\pi)=\sum Score(e,\pi) \quad (e\in\pi \textit{ and } stmt(e)=stmt) \qquad (5)$$

Table 4. Score of suspicious events for failures

Score of suspicious events of F_I		
	9_3	17_5
F_I	∞(2/3)	1/3
Score of suspicious events of F_{II}		
F_I	9_3	17_5
F_{II}	∞(1/2)	1/2

Table.5. Score of suspicious statements for failures

Statements	**9**	**17**
F_I	∞(2/3)	1/3
F_{II}	∞(1/2)	1/2

We can now locate faults through the analysis of Table 5. Each row of the table represents the ranking scores of each statement for a certain failing run, if there are some statements the value of which is significantly larger than others, the statements are the real cause of such failures. Similarly each column represents the ranking scores of a certain statement for each failing run,, the relevant statements with significantly larger values are the cause of these failures. As for our example, statement 9 has a large effect on both F_I and F_{II}, indicating that statement 9 leads to the failure with respect to these two classes of failing runs. So we use the following equation to compute the suspiciousness of any statement.

$$Score_p(stmt)=\sum_{\pi\in F} Score(stmt,\pi)(F \textit{ is the set of all failing runs}) \quad (6)$$

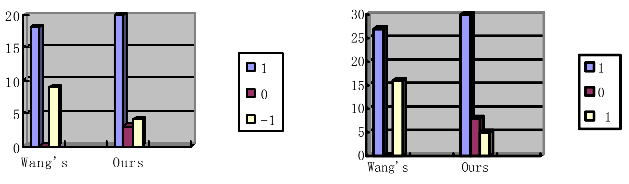
4 Experiment Results

We chose four middle-sized programs with branches, by manually injecting different errors (some programs injected with an error and some with two errors). In this way, we get 43 buggy programs and 79 failing runs. Table 6 shows the description of buggy programs generated from the original four programs.

Table 6. Description of experiment data

Programs	**nested level of branches**	**Buggy Versions**			
		having one bug	***Kinds of failing runs***	***having two bugs***	***Kinds of failing runs***
P1	0	5[a]	14	3	11
P2	1	4	4	4	7
P3	2	8	9	8	16
P4	3	5	5	7	13

To evaluate our algorithm and compared it with Wang's, we consider three cases for each failing run: (1) case "1": for ours the largest suspiciousness statement or for Wang's the whole bug report exactly indicates the actual fault position, (2) case "-1": for two methods the actual faulty location is missed in the bug report, and (3) case "0": for ours the most suspicious statement is not the actual fault position, but the actual fault location is under suspicion. Statistical results obtained from experiments are shown in Fig.2, where the vertical axis is the number of failing runs.



(a) single (b) multiple

Figure.2. Comparison of the bug report of two methods

Wang's algorithm returns the smallest difference as the bug report, ignoring that the position of the fault may not be in this minimal difference but in other differences. In contrast, we use statistical methods taking into account all the circumstances, and also rank the suspicious statements according to suspiciousness. The faulty statements are bound to occur in our report unless they do not appear in any of the differences.

For programs with multiple bugs, we use the evaluation criteria in [3] to calculate the *pgm_score(P)*:instead of using each failing run and successful run, we use each class of failing runs and each class of successful runs. The *pgm_score*(*P*) measures the percentage of code that can be ignored for debugging, the algorithm calculates the score of the most suspicious statement and statement with its suspicious in the second place, and finally calculate *pgm_score*(*P*), as shown in Table 7.

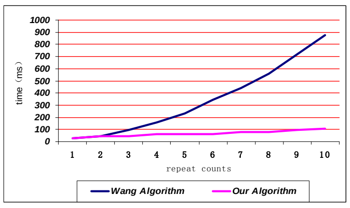
According to table 7, when there are two errors in the program, the most suspicious statement in our bug report has higher score than that by Wang's algorithm. The second suspicious statement also has a higher score, indicating that it is also likely to be the actual fault position, that is, there are more likely two errors in the program

It can be seen from Fig.3, the time Wang spending grows exponentially as the redundant data increases since it spends much on calculating differences between each failing run and each successful run. While our algorithm eliminates redundancy by clustering all the runs, and the next difference computation is less time-consuming, so the time grows slightly. Moreover, we not only consider the suspiciousness for each failing run, but also that for the entire program through statistical analysis of all failing runs and rank statements in bug report based on their overall suspiciousness.

Table 7. Comparison of two methods when there are two errors in the program

Score	**Wang algorithm**	**The first place of our algorithm**	**The second place of our algorithm**
0.8-1.0	13.3	33.3	46.7
0.7-0.79	60.0	26.7	6.7
0.6-0.69	6.7	6.7	20.0
0.5-0.59	13.3	26.7	6.7
0-0.49	6.7	6.7	20.0

Figure.3 Comparison of run time of two methods



5 Conclusion

In this paper, we present an improved fault localization method using a clustering method to eliminate the path redundancy first and then by statistical analysis of differences between classes of runs to get and rank suspicious statements. Experimental results show the great improvement in terms of efficiency and accuracy in fault localization. The next step we will consider to rank all suspicious statements and do further experimental study of our techniques running against large software.

References

[1] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In Proceedings of the 24th International Conference on Software Engineering, pages 467--477, 2002.

[2] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries, 2003.

[3] Liang Guo, Abhik Roychoudhury, and Tao Wang. Accurately choosing execution runs for software fault localization. In CC, pages 80--95, 2006.

[4] Tao Wang and Abhik Roychoudhury. Automated path generation for software fault localization. In ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pages 347--351, New York, NY, USA, 2005. ACM.

[5] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pages 273--282, New York, NY, USA, 2005. ACM.

[6] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pages 15--26, New York, NY, USA, 2005. ACM.

[7] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. Sober: statistical model-based bug localization. SIGSOFT Softw. Eng. Notes, 30(5):286--295, 2005.

[8] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems, 9:319--349, 1987.

[9] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, pages 141--154, New York, NY, USA, 2003. ACM.