Software Fault Localization Using *N*-gram Analysis

Department of Computer Science
The University of Texas at Dallas
{skn051000,maa056000,ewong,lkhan,yxq014100}@utdallas.edu

Abstract. A major portion of software development effort is spent in testing and debugging. Execution sequence collected in the testing phase can be a rich source of information for locating the fault in the program, but the exact execution sequence of a program, i.e., the actual order of execution of the statements in the program, is seldom used due to the huge volume. In this study, we apply data mining techniques on this data to reduce the debugging time by narrowing down the possible location of the fault. Our method applies N-gram analysis to rank the executable statements of a software by level of suspicion. We conducted three case studies to demonstrate the effectiveness of our proposed method. We also present comparison with other approaches, and illustrate the potential of our method.

1 Introduction

Software fault localization is a long standing and very important problem in software engineering. Due to the human involvement in the software development process, it is virtually impossible to develop software free from any kind of fault (a.k.a. *bug* or *defect*). Once a fault is in a software, it is a tedious, time-consuming and difficult process to find its location in the source code as the developer may have to go through the entire code to find the fault. For this reason, research in automated fault localization techniques to indicate or point to possible fault locations is extremely valuable. A lot of research effort has gone into automating the process of discovering the fault location, or *Software fault localization* [1,2,3,4,5,6,7].

Usually fault localization utilizes test cases – sets of inputs with known expected outputs. If the actual output does not match the expected output, the test case has failed. Various information can be collected during the execution of the test cases for later analysis. This information may include statement coverage (the set of statements that were executed at least once during the execution), and exact execution sequence (the actual order in which the statements were executed during the test case executions). Since we will be working only with the exact execution sequence in this paper, we refer to it as *trace*. Usually, the usefulness of trace data is limited by the sheer volume. Data mining traditionally deals with large volumes of data, and in this research, we apply data mining techniques to process this trace data for fault localization.

From trace data, we generate N-grams, i.e., subsequences of length N. From these, we choose N-grams that appear more than a certain number of times in the failing traces. For these N-grams, we calculate the confidence – the conditional probability that a test case fails given that the N-gram appears in that test case's trace. We sort the N-grams in descending order of confidence and report the statements in the program in the order of their first occurrence in the sorted list. We have tested our method on the Siemens suite, the Space program and grep [8]. Our implementation have produced better results on these three suites than the most standard method Tarantula [1].

This paper is organized as follows. In Section 2, we discuss the related terminologies and ideas. In Section 3, we present the complete algorithm. In Section 4, we discuss the reults of applying our algorithm on Siemens suite, Space and grep. In Section 5, we discuss other relevant research. Finally, in Section 6, we present the conclusions from this study and discuss future directions of research.

2 Background

In this section, we discuss the concepts, ideas and definitions related to our method of solving the problem namely *execution sequences*, *N-gram analysis*, *linear execution blocks* and *association rule mining.*

2.1 Execution Sequence

Let P be a program with n lines of source code, labeled as $L = \{l_1, l_2, \ldots, l_n\}$. For example, in the sample program **mid** from [1] in Fig. 1(a), $L = \{4, 5, 6, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 24\}$ after excluding comments, blank lines and structural constructs like '}'. A *test case* is a set of input with known outputs. Let $T = \{t_1, t_2, \ldots, t_n\}$ be the n test cases for program P. Each test case $t_i = \langle I_i, X_i \rangle$ has the input I_i and expected output X_i. When program P is executed with input I_i, it produces actual output A_i. If $A_i = X_i$, then we say t_i is a passing test case, and if $A_i \neq X_i$ then we say t_i is a failing test case. For example, the 6 test cases for the program **mid** in [1], $T = \{t_1, t_2, \ldots, t_6\}$, are shown in Table 1. Let $Y = \langle y_1, y_2, \ldots, y_k \rangle, y_i \in L$ be the trace of program P when running test case T. Then, for **mid** the trace for the test case t_1 is $Y_1 = \langle 4, 4, 5, 10, 11, 12, 14, 15, 24, 6 \rangle$. We define two sets based on the outcome of the test cases – *passing traces* which is $Y_P = \{Y_i | t_i \text{ is a passing test case}\}$ and *failing traces* which is $Y_F = \{Y_i | t_i \text{ is a failing test case}\}$.

We define our problem as: *given program P with executable statements L, test cases T and actual outputs A, the problem is to rank the statements in L according to their probability of containing the fault.* To compare our method with other methods like [1], we report our results in terms of statements, but it can also work at function level.

Given an ordered list, an *N-gram* is any sub-list of N consecutive elements in the list. The elements of the N-gram must be in the same order as they were in the original list, and they must be consecutive. Given an execution trace Y,

Table 1. Test cases for program **mid** [1]

Test Case, t_i	Input I_i	Expected Output, X_i	Actual Output, A_i	Test case type	Trace
t_1	3, 3, 5	3	3	Passing	4,4,5,10,11,12,14,15,24,6
t_2	1, 2, 3	2	2	Passing	4,4,5,10,11,12,13,24,6
t_3	3, 2, 1	2	2	Passing	4,4,5,10,11,18,19,24,6
t_4	5, 5, 5	5	5	Passing	4,4,5,10,11,18,20,24,6
t_5	5, 3, 4	4	4	Passing	4,4,5,10,11,12,14,24,6
t_6	2, 1, 3	2	1	Failing	4,4,5,10,11,12,14,15,24,6

an N-gram $G_{Y,N,\alpha}$ is a contiguous subsequence $\langle y_\alpha, y_{\alpha+1}, y_{\alpha+2}, \ldots, y_{\alpha+N-1} \rangle$ of length N starting at position α. For a trace Y, the set of all line N-grams is $G_{Y,N} = \{G_{Y,N,1}, G_{Y,N,2}, \ldots, G_{Y,N,K-N+1}\}$.

2.2 Linear Execution Blocks

From the set of all traces, we identify the execution blocks, i.e., the code segments with a single point of entry and a single point of exit. For this, we construct the *Execution Sequence Graph* $XSG(P) = (V, E)$ where the set of vertices is $V \subseteq L$ such that for each $v_i \in V$, $v_i \in Y_k$ for some k. E is the set of edges such that for each edge $\langle v_i, v_j \rangle \in E$, we have $v_i, v_j \in Y_k$ for some k and that v_i and v_j are consecutive in Y_k. This is similar to a *Control Flow Graph*, but the vertices in an XSG represent statements rather than blocks. In this graph, there is an edge between two vertices only if they were executed in succession in at least one of

```
 1  #include <stdio.h>
 2  int main(){
 3      int x, y, z, m;
 4      scanf("%d␣%d␣%d,␣"&x, &y, &z);
 5      m = mid(x, y, z);
 6      printf("%d",m);
 7  }
 8  int mid(int x, int y, int z){
 9      int m;
10      m = z;
11      if (y<z){
12          if (x<y){
13              m = y;
14          }else if (x<z){
15              m = y;
16          }
17      }else{
18          if (x>y){
19              m = y;
20          }else if (x>z){
21              m = x;
22          }
23      }
24      return m;
25  }
```

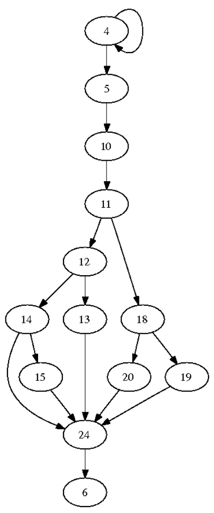


Fig. 1. (a) Sample source code: mid.c, (b) execution sequence graph for program **mid**

the execution traces. The XSG for **mid** is given in Fig. 1(b), where we can see that the blocks of **mid** are $\{b_1, b_2, \ldots, b_{10}\} = \{ \langle 4\rangle, \langle 5, 10, 11\rangle, \langle 12\rangle, \langle 18\rangle, \langle 20\rangle, \langle 19\rangle, \langle 24, 6\rangle, \langle 14\rangle, \langle 13\rangle, \langle 15\rangle \}$. Thus, trace of test case t_1 can be converted to block level trace by $\langle b_1, b_2, b_3, b_8, b_{10}, b_7\rangle$.

It should be noted that our definition of blocks is different than the traditional blocks [9]. Since we identify blocks from traces, our blocks may include function or procedure entry points. For example, $\langle 5, 10, 11\rangle$ will not be a single block by the traditional definition since it has a function started at line 10. Due to this difference, we name our blocks *Linear Execution Blocks*, defined as follows: *A* Linear Execution Block $B = \langle v_i, v_{i+1}, \ldots, v_j\rangle$ *is a directed path in* XSG *such that the indegree of each vertex* $v_k \in B$ *is 0 or 1.* Advantages of using block traces are: (a) it reduces the size of the traces, and, (b) in a block trace, each sequence of two blocks indicate one possible branch. Therefore, in N-gram analysis on block traces, each block N-gram represents $N - 1$ branches. This helps the choice of N for N-gram analysis, discussed in Section 3.1.

2.3 Association Rule Mining

Association Rule Mining searches for interesting relationships among items in a given data set [10]. It has the following two parts:

Frequent Itemset Generation. Search for sets of items occurring together frequently, called a *Frequent Itemset*, whose frequency in the data set, called *Support*, exceeds a predefined threshold, called *Minimum Support.*

Association Rule Generation. Look for association rules like $A \Rightarrow B$ among the elements of the frequent itemsets, meaning that the appearance of A in a set implies the appearance of B in the same set. The conditional probability $P(B|A)$ is called *Confidence*, which must be greater than a predefined *Minimum Confidence* for a rule to be considered. More details can be found in [10].

In our research, we model the blocks as items and the block traces as the transactions. For example, $Y_1 = \langle b_1, b_2, b_3, b_8, b_{10}, b_7\rangle$ is a transaction for **mid** corresponding to the first test case, T_1. We generate frequent itemsets from the transactions with the additional constraint that the items in an itemset must be consecutive in the original transaction. To do this, we generate N-grams from the block traces, and from them, we choose the ones with at least the minimum support. For a block N-gram $G_{Y_i,N,p}$, support is the number of failing traces containing $G_{Y_i,N,p}$:

$$Support(G_{Y_i,N,p}) = |\{Y_j | G_{Y_i,N,p} \in Y_j \text{ and } Y_j \in Y_F\}| \tag{1}$$

For example, for **mid**, the support for $\langle b_2, b_3, b_8\rangle$ is 1 since it occurs in one failing trace. We add the test case type to the itemset. For example, after adding the test case type to the itemset $\langle b_2, b_3, b_8\rangle$, the itemset becomes $\langle b_2, b_3, b_8, passing\rangle$. Then, we try to discover association rules of the form $A \Rightarrow failing$ from these itemsets where the antecedent is a block N-gram and the consequent is *failing.* Therefore, the block N-grams that appear as antecedents in the association rules are most likely to have caused the failure of the test case. We sort these block N-grams in descending order of confidence. For a block N-gram $G_{Y_i,N,p}$, confidence is the conditional probability that the test case outcome is failure given that $G_{Y_i,N,p}$ appears in the trace of that test case. That is,

$$Confidence(G_{Y_i,N,p}) = \frac{Prob(G_{Y_i,N,p} \in Y_j \text{ and } t_j \text{ is a failing test case})}{Prob(G_{Y_i,N,p} \in Y_j)} \quad (2)$$

For example, the confidence the rule $\langle b_2, b_3, b_8 \rangle \Rightarrow failing$ has confidence 0.33. After sorting the block N-grams, we convert the blocks back to line numbers and report this sequence of lines to investigate to find the fault location.

3 Methodology

In this section, we present our methodology for localizing faults. As input we use the source code, the test case types and the traces for all the test cases, and produce as output an ordered list of statements, sorted in order of probability of containing the fault. We first convert the traces to block traces, and then apply N-gram analysis on these block traces to generate all possible unique N-grams for a given range of N. For each N-gram, we count its frequency in passing and failing traces. The set of N-grams and their frequencies are analyzed using the association rule mining technique described in Section 2.3.

The execution of the faulty statement may not always cause failure of the test case. There might be quite a number of test cases in which the faulty statement was executed but it did not cause a failure. In most cases, the failure is dependent on the sequence of execution. A specific sequence or path of execution will cause the program to fail, and this sequence will be very common in the failing traces but not so common in the passing traces. Therefore we can find these subsequences that are most likely to contain the fault by analyzing the traces during passing and failing test cases.

3.1 Parameters of Algorithm

There are two major parameters in the algorithm - the first one is $MinSup$, the minimum support for selecting the N-grams, and the second is N_{MAX}, the maximum value of N for generating the N-grams. Taking a low value of minimum support will result in the inclusion of irrelevant N-grams in consideration. Therefore, we should take minimum support at a high value. Our experience suggests that 90% is a good choice. However, choice of an appropriate N_{MAX} is harder. Two execution paths can differ because of conditional branches. Such differences can be detected by 2-grams. Again, the same function can be called from different functions, which can also be detected with 2-grams. Since we are using execution blocks, an N-gram can capture $(N-1)$ branches, and a choice of 2 or 3 for N_{MAX} should give good results in most cases. If we use higher N-grams, the algorithm will still be able to find the fault, but due to larger N-grams, we will have to examine more lines to find the fault.

Algorithm 1. Fault Localization using N-gram Analysis

```
 1: procedure LocalizeFaults($Y, Y_F, K, MINSUP$)
 2:     for all $Y_i \in Y$ do
 3:         Convert $Y_i$ to block trace
 4:     end for
 5:     $NG \leftarrow \phi$
 6:     for $N = 1$ to $N_{MAX}$ do
 7:         $NG \leftarrow NG \cup GenerateNGrams(Y, N)$
 8:     end for
 9:     $L_{rel} \leftarrow \{n|n \in NG$ and $|n| = 1\}$
10:     for all $n \in L_{rel}$ do
11:         if $Support(n) \neq |Y_F|$ then
12:             Remove $n$ from $NG$ and $L_{rel}$
13:         end if
14:     end for
15:     $NG_1 \leftarrow \{n|n \in NG$ and for all $s \in L_{rel}, s \notin n\}$
16:     $NG \leftarrow NG - NG_1$
17:     for all $n \in NG$ do
18:         if $Support(n) < MINSUP$ then
19:             Remove $n$ from $NG$
20:         end if
21:     end for
22:     for all $n \in NG$ do
23:         $NF \leftarrow |\{Y_k|Y_k \in Y_F$ and $n \in Y_k\}|$
24:         $NT \leftarrow |\{Y_k|Y_k \in Y$ and $n \in Y_k\}|$
25:         $n.confidence \leftarrow NF \div NT$
26:     end for
27:     Sort $NG$ in descending order of confidence
28:     Convert the block numbers in the $N$-grams in $NG$ to line numbers
29:     Report the line numbers in the order of their first appearance in $NG$
30: end procedure
```

3.2 Algorithm

In this section, the complete algorithm is presented in Algorithm 1. Following is a description of the steps in the algorithm.

L2B: Convert exact execution sequences to block traces. From the line level traces, we create the Execution Sequence Graph (XSG) as described in Section 2.1. From the XSG, we find the Linear Execution Blocks (LEB). Then we convert the traces into block traces in lines 2 to 4 of Algorithm 1.

GNG: Generate N-grams. In this step, we first generate all possible N-grams of lengths 1 to N_{MAX} from the block traces. The generation of all N-grams from a set of block traces for a given N is done in lines 1 to 7, and the generation and combination of all the N-grams are done in lines 5 to 8. Then, we find out how many passing and failing traces each N-gram occurs in.

Algorithm 2. N-gram generation

```
1: function GenerateNGrams($Y, N$)
2:     $G \leftarrow \phi$
3:     for $Y_i \in Y$ do
4:         $G \leftarrow G \cup G_{Y_i,N}$
5:     end for
6:     return $G$
7: end function
```

FRB: Find Relevant Blocks. From 1-gram, we construct a set of relevant blocks, B_{rel} that contains only those blocks that have appeared in each of the failing traces in lines 10 to 14.

EIN: Eliminate Irrelevant N-grams. In lines 15 to 16, we discard those N-grams that do not contain any block from the relevant block set, B_{rel}.

FFN: Find Frequent N-grams. In lines 17 to 21, we eliminate N-grams with support less than the minimum support as described in Section 2.3.

RNC: Rank N-grams by Confidence. For each surviving N-gram, we compute its confidence using Eqn. 2. This is done in lines 22 to 26. Then we order the N-grams in order of confidence in line 27.

B2L: Convert Blocks in N-grams to Line Numbers. We convert each block in the N-grams back to line numbers using the XSG in line 28.

RLS: Rank Lines According to Suspicion. We traverse the ordered list of N-grams, and report the line numbers in the order of their first appearance in the list. This is done in line 29.

If there are multiple N-grams with the same confidence as the N-gram containing the faulty statement, the best case will be the ordering in which the faulty statement appears in the earliest possible position in the group, and the worst case will be the ordering in which the faulty statement appears in the latest possible position.

4 Case Study

We define the number of lines a programmer needs to examine to find out the fault location as the rank of the program. For example if we have to check α lines to find the fault location of a program, then we say α is the rank of that program. When we are comparing two methods, the method that gives smaller rank is the more effective method. For example, for a program P if method M_1 gives the rank α and method M_2 gives rank β and if $\alpha < \beta$ then it is said that M_1 performs better than M_2. For a program with multiple versions, if methodology M_1 gives smaller ranking for more faulty versions than M_2 then we say M_1 is better than M_2 for that program. Section 4.1 describes the test suites and programs downloaded from [8] used in this study.

4.1 Test Suites

The *Siemens suite* contains 7 programs. The number of faulty versions range from 7 to 41, number of executable statements range from 55 to 216, and number of test cases range from 1052 to 5542. Of the 132 faulty versions, three were not used in our study because one did not have any failing test case and two had faults in header files. The *Space* program has 6218 lines of executable code, 38 faulty versions and 13585 test cases. We did not use 3 faulty versions in our study because there were no failing tests for these versions. The *grep* program has 3306 executable statements, 470 test cases and 18 versions. Compared to [11] that failed to detect any of these faults, we could detect 4 faults in our environment. So we used these 4 versions and also used 2 faults injected by [11], and followed a similar approach to inject 13 more bugs, for a total 19 faulty versions. Manually injected faults are designed to mimic realistic bugs, as described in [11].

4.2 Running the Tests

We conducted our experiments on a Sun Microsystems with 64 bit Intel CPU, 1GB physical memory running Solaris 5.10. We used GCC 3.4.3 and GDB 6.6. For each program, we generated the expected outputs by running the correct program for each test case. Then, we executed the program with the test cases through GDB using a java program to collect the traces. The advantage of using GDB to collect traces is that unlike other studies [6,2] no instrumentation is needed, and we can collect the complete data even if there is a segmentation fault. After data collection, we compared the output of each run with the corresponding expected output and labeled accordingly as passing or failing.

4.3 Applying and Evaluating Our Method

We applied our method on the data collected in Sect. 4.2. For each version, we ran our method for $N = 1, 2, \ldots, 6$ and minimum support of $30\%, 40\%, \ldots, 90\%$ and determined the best case and worst case ranks of the line containig the fault in the source code as described in Section 3.2. From these ranks, we calculated the percantage of code that needs to be examined to find a fault in the best and worst case. From this experiment we found that the best result is obtained when $N = 3$, Minimum Support = 90%, validating our analysis in Section 3.1.

To evaluate our method we compared our results with results from Tarantula [1]. To make the comparison fair we had to collect the data and run Tarantula again because [1] excluded 10 faulty versions and used slightly different number of test cases. We collected the coverage data using a revised version of χSuds [12]. The comparisons of the results are discussed in the following sections.

The Siemens Suite. Fig. 2 shows the comparison between the Tarantula [1] and our N-gram method. The horizontal axis represents the cumulative percentage of code to be examined and the vertical axis represents the total number of faulty versions for which bug can be detected by examining this percentage of

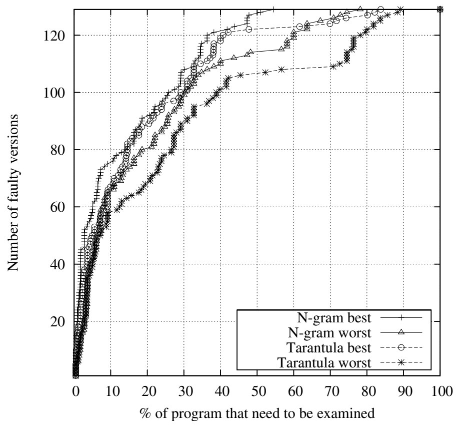
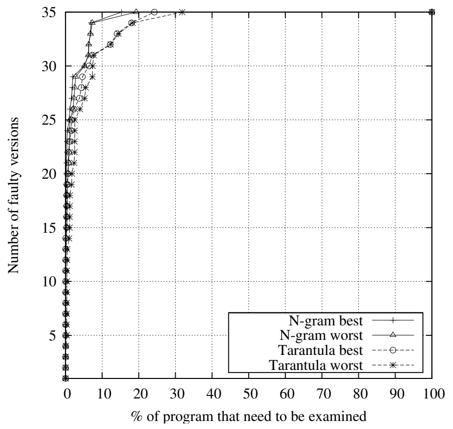


Fig. 2. Comparison between N-gram and Tarantula Method for Siemens suite

code. In worst case N-gram method can discover 80 out of 129 faults by examining only 20% of code while Tarantula can discover only 68 faults from the same percentage of code. Also, in most cases the best case result for N-gram method is better than the best case result for Tarantula, and also worst case result for N-gram method is always better than the worst case result for Tarantula. Also we can see that in worst case N-gram method can discover all 129 faults by examining 78% code while Tarantula has to examine 89% code to discover all faults. Also, we can see from Table 2 that in best case N-gram method performs better than Tarantula method in 120 versions and in worst case our method performs better than Tarantula method in 92 versions.

The Space Program. Fig. 3(a) gives the comparison for Space program between Tarantula [1] and our N-gram method. The axes are same as Fig. 2. In worst case N-gram method can discover 24 faults out of 35 by examining only 1% of code while Tarantula can discover 22 faults by examining that much code. The best and worst case results for N-gram method is always better than the best and worst case results for Tarantula respectively. Also, in worst case N-gram method can discover all faults by examining 20% code while Tarantula needs 32% code for this. For 12 faulty versions out of 35 our *worst* case result is better than Tarantula's *best* case result. Table 2 shows that in best case our method performs better than Tarantula in 21 versions and in worst case our method performs better than Tarantula in 31 versions out of 35.

The grep Program. Fig. 3(b) shows that in worst case N-gram method can discover all faults by examining only 5% of code while Tarantula can discover only 11 faults by examinig that percentage of code. The graph also shows that the *worst* case result for N-gram method is always better than the *best* case result for Tarantula. Table 2 shows that in best case our method performs better than Tarantula method in 17 versions and in worst case our method peforms better than Tarantula method in all versions.



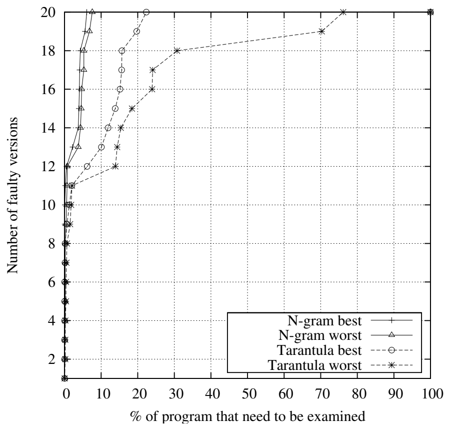


Fig. 3. (a) Comparison between N-gram and Tarantula Method for Space, (b) Comparison between N-gram and Tarantula Method for grep

Table 2. Pairwise comparison between N-gram and Tarantula Method

	Siemens Suite	Space	Grep
Ngram better than Tarantula in best case	120	21	17
Tarantula better than Ngram in best case	4	8	2
Ngram equal to Tarantula in best case	5	6	0
Ngram better than Tarantula in worst case	92	32	19
Tarantula better than Ngram in worst case	28	1	0
Tarantula equal to Ngram in worst case	9	2	0

From the above results we can say that N-gram method outperforms than Tarantula [1] in all of the programs. We also observe from this result that our method perform very well for larger programs and it proves that our method can handle large volume of data than the traditional method.

5 Related Works

In the last few years, a lot of research has been done in this area. In [3], Guo et. al. selected a single passing execution most similiar to a failing trace tried to identify the fault location based on their differences. In [13], Renieris et. al. also find the most similar passing traces but they use nearest neighbor method to measure similarity. Liblit et. al., in [5], described how to collect program execution traces at run time by deploying assertions in the program. They collected only predicate level trace and gave their results at function level. Jones et. al., in their work [1], present a visualization technique using the coverage matrix of the program execution to identify suspicious statements. Denmat et. al. shows in [14] that Association Rule Mining can be applied on coverage matrix. Liu et. al., in their work [2], took each logical expressions as features and tried to detect features that behave differently in passing and failing runs. They also used clustering to detect multiple bugs in [6]. Their method give result in function level. In their research in [4] on software behavior graphs, they used SVM classification to detect suspicious subgraphs, producing a back trace for the fault location. Other works on software behavior graph mining include [15], where Fatta et. al. present their work on finding discriminative patterns based on the failing and passing program execution. Besides test case analysis, researchers also analyze the source code to detect the defect in the source code which may cause software failure, for example, [16,17,18,19,20].

6 Conclusions and Future Works

We have developed a new fault localization algorithm by analyzing the statement sequences of faulty versions. Applying N-gram analysis to fault localization has a very promising future as the results presented in this paper indicate. Using our method, worst case average number of lines to check is 18 in the Siemens suite, 78 for Space and 231 for grep. These results are much better than those achieved by the most standard method Tarantula [1], whose corresponding results are 26 lines, 146 lines and 1488 lines respectively. In all cases, our worst case result is better than Tarantula's worst case result. This shows that our method is both practical and produces better results. Speciallly for larger programs our method produces much better results than [1]. In this study our method only works on single fault, but it can be extended to multiple faults by grouping the failing cases which are caused by same fault and applying our method on these groups.

Research using exact execution sequences, as well as applying data mining to fault localization, is still in beginning phase and there are a lot of avenues to explore and places for improving the results. We are investigating augmenting the execution traces with data flows in order to pinpoint data-driven faults. With software sizes growing with time, processing the huge data collected from test cases will eventually only be possible with data mining methods. Even then, we need to improve our methods to reduce execution time and space. Scalability has to be studied to ensure that it can be applied to large scale software. Also, since it is very common in real life, we need to develop methods to handle multiple faults. It is our belief that research in these directions can help significantly reduce the efforts required to produce fault-free software.

References

1. Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering (2002)
2. Liu, C., Yan, X., Han, J.: Mining control flow abnormality for logic error isolation. In: Proceedings of 2006 SIAM International Conference on Data Mining (2006)
3. Guo, L., Roychoudhury, A., Wang, T.: Accurately choosing execution runs for software fault localization. In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 80–95. Springer, Heidelberg (2006)

4. Liu, C., Yan, X., Yu, H., Han, J., Yu, P.S.: Mining behavior graphs for backtrace of noncrashing bugs. In: Proc. 2005 SIAM Int. Conf. on Data Mining (2005)
5. Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (2003)
6. Liu, C., Han, J.: Failure proximity: a fault localization-based approach. In: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering (2006)
7. Liu, C., Lian, Z., Han, J.: How bayesians debug. In: IEEE International Conference on Data Mining (2006)
8. Do, H., Elbaum, S.G., Rothermel, G.: Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering: An International Journal (2005)
9. Agrawal, H.: Dominators, super blocks, and program coverage. In: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (1994)
10. Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers, San Francisco (2001)
11. Liu, M.C., Fei, M.L., Yan, M.X., Han, S.M.J., Midkiff, M.S.P.: Statistical debugging: A hypothesis testing-based approach. IEEE Trans. Softw. Eng. (2006)
12. Li, J.J., Horgan, J.R.: χsuds-sdl: A tool for testing software architecture specifications. Software Quality Journal (2000)
13. Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: Proceedings of 18th IEEE International Conference on Automated Software Engineering (2003)
14. Denmat, T., Ducass, M., Ridoux, O.: Data mining and cross-checking of execution traces: a re-interpretation of jones, harrold and stasko test information. In: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (2005)
15. Fatta, G.D., Leue, S., Stegantova, E.: Discriminative pattern mining in software fault detection. In: Proceedings of the 3rd international workshop on Software quality assurance (2006)
16. Engler, D., Chen, D.Y., Hallem, S., Chou, A., Chelf, B.: Bugs as deviant behavior: a general approach to inferring errors in systems code. SIGOPS Oper. Syst. Rev (2001)
17. Li, Z., Zhou, Y.: Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. SIGSOFT Softw. Eng. Notes (2005)
18. Chang, R.Y., Podgurski, A., Yang, J.: Finding what's not there: a new approach to revealing neglected conditions in software. In: Proceedings of the 2007 international symposium on Software testing and analysis (2007)
19. Ramanathan, M.K., Grama, A., Jagannathan, S.: Path-sensitive inference of function precedence protocols. In: Proceedings of the 29th international conference on Software Engineering (2007)
20. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: Cp-miner: a tool for finding copy-paste and related bugs in operating system code. In: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation (2004)