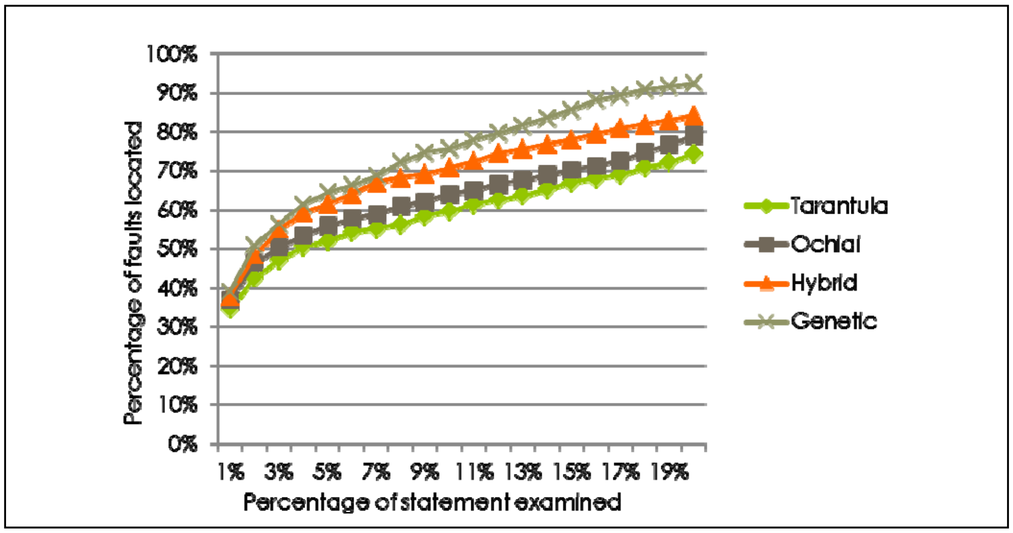


Learning efficient software fault localization via genetic programming

Master of Science Thesis

Secure and dependable computer systems programme

GUOJIAN CHEN

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, March 2012

Learning efficient fault localization via genetic programming

GUOJIAN CHEN

Examiner: Johan Karlsson

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover: Diagram showing the improvement in fault localization effectiveness achieved by a ranking function derived by genetic programming compared to other ranking functions (page 28).

Department of Computer Science and Engineering
Göteborg, Sweden February 2012

Abstract

The high cost associated with debugging of computer software has motivated development of semi-automatic fault localization techniques. Such techniques assist developers in locating faulty code by ranking program statements according to their likelihood of being faulty. The ranking is done by automated analysis of test coverage or execution profile data. A variety of fault localization techniques utilizing different types of ranking functions have been proposed in the past. In this paper, we present a new fault localization technique where we have used genetic programming to find a highly effective ranking function. First, we divide frequently appearing fault types into four subsets. We then identify potentially useful execution profiles and use genetic programming to search for a new improved ranking function for each fault type individually. Finally, we merge the ranking lists provided by the four ranking functions into a final aggregated ranking list, which is used by the developer to search for faulty code. We evaluated the efficiency of our technique using execution profile data from two programs, GCC and SPACE, and compared it to the efficiency of two existing fault localization techniques. The result shows that our approach is highly effective, as we can locate more than 90% of the faults by examining the top 20% of the statements in the ranking list. The improvement in efficiency is about 10% compared to Lightweight fault localization and 20% compared the Tarantula technique.

Acknowledgements

I am heartily thankful to my supervisor Professor Shing-Chi Cheung at Hong Kong University of Science and Technology. He gave me guidance and support throughout the whole period of the thesis. I am grateful to my co-worker Dr Xingming Wang for his great help and support. Also, I would like to thank my examiner Professor Johan Karlsson at Chalmers University of Technology for spending much time on my revision of thesis report. Finally, I offer my regards and blessings to all of those who supported and helped me during this thesis report.

Table of Contents

1. Introduction

Debugging is a time consuming and expensive activity in software development. Locating faulty code within a program has been recognized as the most challenging step in debugging [14]. For this reason, extensive research efforts have been devoted to the development of techniques for software fault localization. The aim of such techniques is to guide developers in the process of finding faulty code.

Generally, software fault localization can be divided into two activities. The first involves using an automated analysis of program properties for providing a list of program entities (statements, methods, branches, etc.) that are deemed suspicious of being faulty. In the second activity, the developer manually checks the correctness of the suspicious program entities according to a priority scheme, or suspiciousness ranking, produced by the automated analysis.

A variety of techniques for automated fault localization have been investigated in the past. Some techniques reduce the debugging search domain by dividing the program into segments. These techniques are in some papers called slice-based techniques (e.g., [18]). Other techniques are using instrumentation and evaluation of predicates in the program to generate a ranking list of suspicious predicates that developers use to find the fault. These techniques are called statistic-based techniques [9].

An important family of fault localization techniques is those that use execution profiles to rank the suspiciousness of program statements. These techniques are in some papers called *spectrum-based fault localization* (e.g., [3] and [19]). An execution profile, a.k.a. a program spectrum, characterizes the behavior of a program for a given set of inputs or test cases [16]. Examples of execution profiles that are used for fault localization include statement coverage [2, 3, 4, 5], branch coverage [2, 3], du-pair coverage [2] and intra-procedural path coverage [7].

A well-known execution profile-based technique is Tarantula [4]. It uses pass/fail information for each test case, and information about executed program entities (statements, branches and methods) as inputs to the ranking function.

In general, execution profile-based fault localization has two main components. An execution profile and a ranking function. The execution profile consists of coverage information collected for a set of test cases. In addition, it also contain information about the outcome of each test case, i.e., whether the program failed or passed the test. The ranking function uses the execution profile information to calculate values of suspiciousness for statements or code segments in the program. Examples of existing ranking functions include the Tarantula function [4], the Ochiai coefficient [2], and the CBI function used in statistical debugging [6].

The possible choices of execution profiles and ranking functions constitute a design space for constructing execution profile-based fault localization techniques. The exploration of this design space for finding more accurate techniques is a key research issue, since also a small improvement in the fault localization accuracy is of direct benefit to a developer. For example, if we can improve the ranking position of a faulty statement with 1%, such as from rank 3000 to rank 2000, for a program with 100K statements, then the developer needs to investigate 1000 fewer statements before the fault is found.

In the literature, each of the existing execution profile-based fault localization techniques contributes with one or a few points in the design space by exploring one or several execution profile type and a ranking method. Till now, existing work in this area has only covered a tiny portion of the design space. Many potentially useful types of execution profiles, such as program state coverage, have not yet been explored. A recent study by Santelices et al [2] suggests that the effectiveness of an execution profiles depends on the nature of the faults. This means that an execution profile that is highly effective in locating certain fault types could do a poor job in locating other fault types. For example, a predicate profile might be good at locating faults that are related to wrong branch condition, while an information flow profile might be good at locating faults that are related to wrong assignments.

Inspired by the work of Santelices et al, we believe that the combination of multiple execution profiles is a promising way toward finding more efficient and effective fault localization techniques. Designing such a technique is however a challenging task as the choices of execution profiles and the way to combine them in the ranking method are many. In this thesis, we use genetic programming to automate the search for effective ranking functions.

Genetic programming is a general technique for finding efficient or optimal solutions to a given problem. It simulates the process of problem solving in an evolutionary way. This means that the solution to a given problem will evolve until the user accepts it. Based on this, we formulate a search space, which can be described as a large number of execution based fault localization techniques. Our aim is to try to search and evaluate a large number of possible solutions to find the best one. In this case, we can get a new execution profile-based fault localization technique at the end and know that this technique is better than most of the technique in the search space.

The remainder of this thesis is organized as follows. We describe related work in execution profile-based fault localization in Section 2. In Section 3, we provide a short introduction to genetic programming and discuss how we use it to search for effective ranking functions. Section 4 describes three techniques for combining data from different execution profiles for fault localization. In Section 5 we describe how our experiments were designed and implemented. Section 6 presents our experimental results. Conclusions and suggestions for future work are given in Section 7.

2. Related work

In this section, we give an introduction to execution profile-based fault localization and describe two existing fault localization techniques, Tarantula [4] and Lightweight fault localization [2]. In Section 2.1, we describe twelve different execution profiles that can be used for fault localization. Overviews of Tarantula and Lightweight fault localization are given in Section 2.2 and Section 2.3, respectively.

2.1. Execution profiles

Execution profile-based fault localization uses information collected during program execution to rank the suspiciousness of each statement in a program, subprogram or a block of code. The execution profile data is collected while the program is subjected to a set of test cases. This data reflects different aspects of the program execution such as which statements, branches, blocks and paths that are executed during execution of the test cases. The ranking function uses this information to produce a ranking list of suspicious program statements.

We use several types of execution profiles as input to genetic programming to find a method to combine different execution profiles and to generate efficient ranking functions. There are four execution profiles that are popular among fault localization researchers: statement coverage [2, 3, 4, 5], branch coverage [2, 3], du-pair coverage [2] and path coverage [7]. We use these four and another eight execution profiles when we search for ranking functions via genetic programming.

Statement coverage consists of execution trace information in terms of how each test case covers the executable statements and the corresponding pass/fail execution result.

Branch coverage measures which possible branches are followed during the execution of a test case. Sometimes faults are triggered only when a specific branch is taken. Santelices et al [2] combines branch coverage, statement coverage and DU-pair coverage in their approach call Lightweight fault localization. Their results show that the use of branch coverage made it easier to locate faults such as a missing branch statement or a wrong branch assignment, as well as those that are triggered by a specific branch.

Path coverage counts the number of times each acyclic path within a program or block of code is executed. Path coverage is computational expensive to measure, but provides more information about a program's dynamic behavior than statement coverage, basic block coverage or edge coverage, which are simpler to measure. A technique for efficient path profiling is presented by Ball and Larus in [15]. This technique profiles each procedure separately and is therefore known as an intra-procedural path profiling technique. Examples of fault localization techniques that use path coverage are described in [7] and [16].

DU-pair coverage. A DU-pair is a pair consisting of the definition and the use of a program variable. Using DU-pair coverage can simplify identification of faults that are related to erroneous handling of variables. Missing assignments or assigning the wrong value to a variable are examples of such faults.

Predicate coverage measures the outcome of instrumented predicates inserted at selected program points. A variety of predicates can be used for fault localization. In

[20], the authors use predicates related to branches, function return values and scalar pairs for statistical bug isolation.

Function coverage tracks function calls and records their execution information.

Block coverage is similar to statement coverage, but uses basic blocks rather than individual statements as the measured entity.

Supersede coverage is a special version of statement coverage. Let X_y denote the number of *failed test cases* that execute statement *y*. The supersede coverage for statement *i* is defined as the cardinality of the set of statements $S = \{ s \mid X_s > X_i \}$. That is, the supercede coverage for statement *i* is equal to the number statements which has a value of X that is greater than X_i.

Surpass coverage for statement *i* is the number of statements having a value of *X* that is less than X_i. That is, the supersede coverage for statement *i* is defined as the cardinality of the set of statements $S = \{ s \mid X_s < X_i \}$.

Equivalent coverage. For a statement *i*, the equivalent coverage measures the number of statements that have been executed by the same number of pass and fail test cases as statement *i*.

Total Coverage is equal to total number of test cases (fail or pass) that executes a given statement.

The supersede, surpass, total, and equivalent coverage can be viewed as special execution profiles since they are represented by single variables in the genetic programming approach.

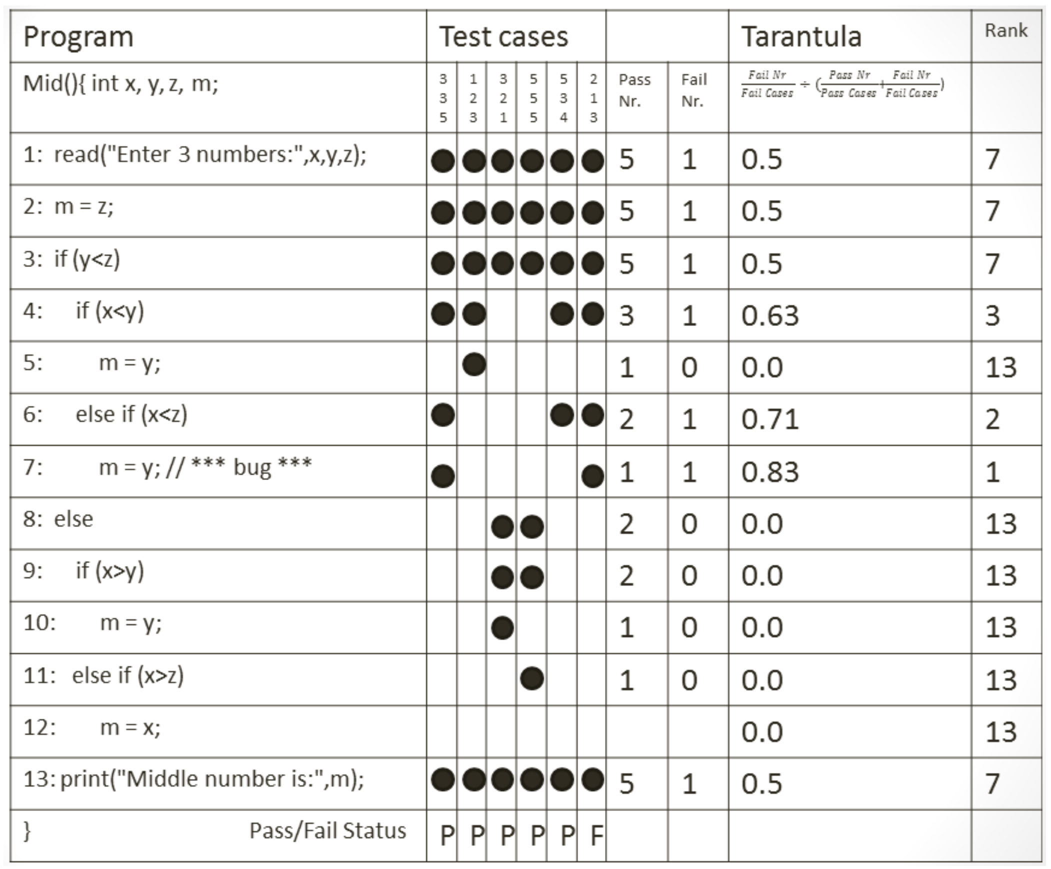
2.2. Tarantula

In the Tarantula technique [4], the execution profile is statement coverage and the ranking method is the heuristic formula shown in Figure 1. Failed Nr(S) is the number of failed test cases that execute statement(S). Passed Nr(S) is the number of passed test cases that execute statement(S). The variable Total Failed Cases is the total number of failed test cases. Total Passed Cases is the total number of passed test cases.

$$ranking(S) = \frac{\frac{Failed\ Nr(S)}{Total\ Failed\ Cases}}{\frac{Failed\ Nr(S)}{Total\ Failed\ Cases} + \frac{Passed\ Nr(S)}{Total\ Passed\ Cases}}$$

Figure 1- The Tarantula ranking function

Figure 2 illustrates how Tarantula works with a simple example. This example is taken from [4]. The black dots in the column denoted Test cases indicate which statements that are executed for a given test case. For example, the test case, x=3, y=3, z=5, executes statement 1, 2, 3, 4, 6, 7 and 13. We can see that statement 7 is assigned the highest value of suspiciousness. Consequently, Tarantula successfully pinpoints the faulty statement for this example.



Program	Test cases								Tarantula	Rank
Mid(){ int x, y, z, m;	3 3 5	1 2 3	3 2 1	5 5 5	5 3 4	2 1 3	Pass Nr.	Fail Nr.	$\frac{Fail\ Nr}{Fail\ Cases} \div (\frac{Pass\ Nr}{Pass\ Cases} + \frac{Fail\ Nr}{Fail\ Cases})$	
1: read("Enter 3 numbers:",x,y,z);	●	●	●	●	●	●	5	1	0.5	7
2: m = z;	●	●	●	●	●	●	5	1	0.5	7
3: if (y<z)	●	●	●	●	●	●	5	1	0.5	7
4: if (x<y)	●	●			●	●	3	1	0.63	3
5: m = y;		●					1	0	0.0	13
6: else if (x<z)	●				●	●	2	1	0.71	2
7: m = y; // *** bug ***	●					●	1	1	0.83	1
8: else			●	●			2	0	0.0	13
9: if (x>y)			●	●			2	0	0.0	13
10: m = y;			●				1	0	0.0	13
11: else if (x>z)				●			1	0	0.0	13
12: m = x;									0.0	13
13: print("Middle number is:",m);	●	●	●	●	●	●	5	1	0.5	7
} Pass/Fail Status	P	P	P	P	P	F				

Figure 2 – Tarantula example

2.3. Lightweight fault-localization using multiple coverage types

Figure 3 shows the result of applying three different execution profile-based fault localization techniques on 43 real faults, as reported in [2]. These three techniques apply the Tarantula ranking formula (shown in Figure 1) on statement coverage profile, branch coverage profile, and Du-Pair coverage profile, respectively. Their results are shown as points of three different colors/shapes. In the diagram, the y-axis represents the effectiveness of a fault localization technique, measured by percentage of code that developers need to examine before they find the fault when they follow the ranking result produced by a certain technique. The x-axis presents the index of the fault.

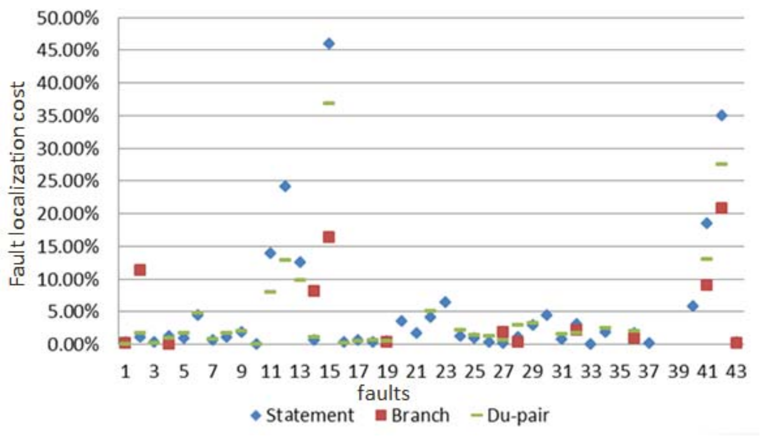


Figure 3 – Fault Localization using Statement Branch Du-pair reported in [2].

From Figure 3, we can observe that different execution profiles are effective at locating different types of faults. For example, fault 12 is best located with the statement coverage profile, fault 13 is best located with the du-pair coverage profile, and fault 15 is best located with the branch coverage profile. Therefore, none of the three techniques is ideal. In [2], the authors suggest to combine the three execution profiles toward a way to meet an ideal method, which is illustrated by the black curve in Figure 4.

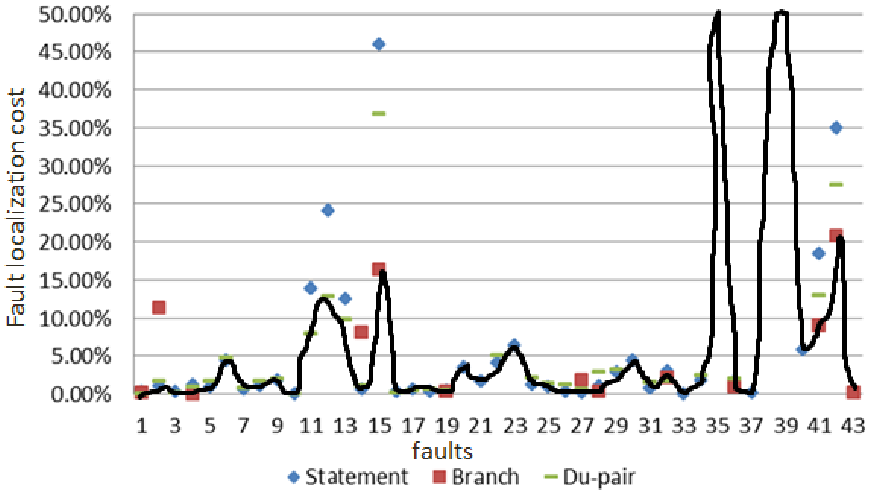


Figure 4 - Ideal method in [2]

Note that fault #35 still requires the developer to examine more than 50% of statements, i.e., half the total source codes, to locate the fault. This observation suggests that this curve still not represents the result of an ideal fault localization technique.

Table 1- Example from Santelices et al [2].

Stmt Nr.	Branch Coverage		DU-pair Coverage		Avg.Score	Avg.Rank
	Score	Rank	Score	Rank	Score	Rank
1	0	3	0.71	3	0.35	3
6	0.71	2	0.6	4	0.65	2
7(Faulty)	0.71	2	0.71	3	0.71	1
13	0	3	0.71	3	0.35	3

Table 1 shows how the lightweight fault localization technique combines different execution profiles into a single ranking function. If we only use branch coverage, the faulty statement is ranked as the second most suspicious statement. If we use du-pair coverage instead, the faulty statement is ranked as the third most suspicious statement. Now suppose we combine the two execution profiles, then the faulty statement is ranked as the most suspicious statement. The method to combine two execution profiles is simply to calculate the average score as the final ranking score for each statement. The reason the combination of execution profile leads to better result is that statement 6 and 7 are in the same branch, so the pass/fail data are the same, which means that the ranking values also are the same. However, statement 6 and 7 are in different du-pairs, which causes statement 7 to rank higher than statement 6 in the du-pair coverage profile. Hence, if we use the average score as the ranking value, the faulty statement 7 is given and highest rank.

3. Genetic programming

In this section, we describe what genetic programming is and how genetic programming can be used to search for effective ranking functions.

Genetic programming is an evolutionary algorithm-based methodology to produce computer programs (or algorithms) that can solve a given problem.

A genetic algorithm encodes the solution to a given problem as a string representing an expression like the one shown in Figure 5. Each solution is called an individual. The entire set of individuals is called a population. We evaluate the fitness of each individual according to the usefulness of solving the given problem. To create the next generation, we need a set of genetic operations to create new individuals from the previous generation. An individual which has a higher fitness value has a higher chance of being chosen to propagate to the next generation. This means that next generation contains many individuals which contain one or more component from good individuals of the previous generation. The ranking functions in the first generation are generated randomly and are therefore not likely to achieve high fitness values. However, the solutions improve rapidly within a few generations.

The procedure of genetic programming can be divided into 4 steps:

1. Randomly generate a set of possible solutions (individuals) to the problem.
2. Test each possible solution against the problem using a fitness function.
3. Keep the best solutions, and use them to generate new possible solutions.
4. Repeat the previous three steps until either an acceptable solution is found, or the algorithm has iterated through a given number of cycles (generations)

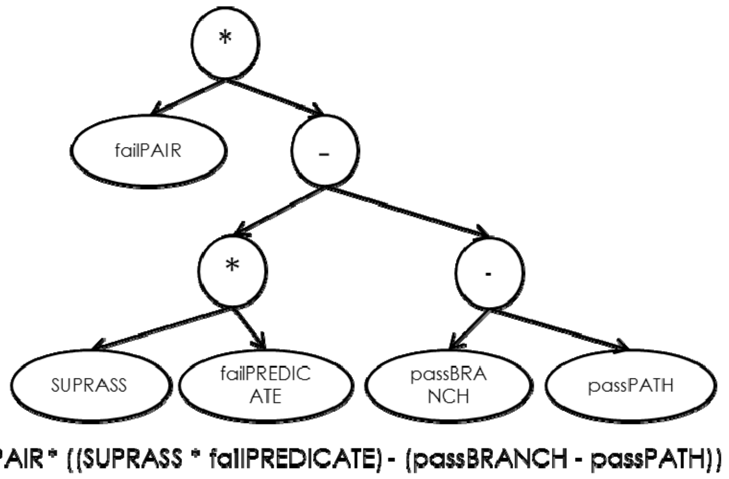

*
failPAIR
-
*
-
SUPRASS
failPREDIC ATE
passBRA NCH
passPATH
failPAIR * ((SUPRASS * failPREDICATE) - (passBRANCH - passPATH))

Figure 5 - Tree representation of a ranking function

3.1. Representation of expressions

In our case, individuals are represented by a tree structure containing a set of math operators, and a set of variables from different execution profiles.

Variables of each execution profile

As we collect data for a given execution profile, we get four types of values: 1) the number of failed test cases, 2) number of passed test cases, 3) number of failed test cases that executed the profile entity, and 4) number of passed test cases that executed the profile entity. Here profile entity corresponds to the profile type, e.g., statement coverage, path coverage, du-pair coverage. When programmers check a program for faults, they directly focus on statements. So each profile entity should be determined for each individual statement.

Hence, each execution profile can be represented by four variables.

For execution profile X:

- *Totalfail*, represent the total number of failed test cases.
- *Totalpass*, represent the total number of passed test cases.
- *FailX(S)*, represent the number of failed test cases that executed profile X’s entity related to statement S.
- *PassX(S)*, represent the number of passed test cases that executed profile X’s entity related to statement S.

Thus, an execution profile consists of a table with at least four entries for each statement. In some execution profiles, there can be multiple entries for one statement. An example of this is given in Figure 6, where the table containing the path coverage data contains multiple entries for statement 1 and statement 3. The multiple entries appear because one statement can be included in several paths.

Mathematical Operators

We use the following mathematical operators in the search for effective ranking functions: Addition, Subtraction, Multiplication, Division, Logrithm and Exponent. We limit ourselves to these mathematical operators to ensure that the formulas do not get too complicated.

3.2. Genetic operators

Genetic operators are used for generating a new generation of individuals. They are a key element in genetic programming, as they control the evolution process.

Replication

To guarantee that the good individuals survive the evolution process, the best individuals are copied to the next generation. This set of individuals is about 10-50% of whole population. Normally, the default percentage is 20% in genetic programming.

Mutation

Mutation can be divided into two sub-operators: node mutation and sub-tree mutation. In node mutation, we randomly choose a node, and replace it with another node of the same type. If the chosen node holds an execution profile variable,it is replaced with a node representing another execution profile variable. If the node holds a mathematical operator, it is replaced with node holding a different mathematical operator.

In sub-tree mutation, a sub-tree is randomly selected and replaced with a new randomly generated sub-tree.

Crossover

Crossover operator is used to exchange two nodes which are randomly selected in two individuals. This creates two new individuals in the next generation.

4. Definition of ranking functions

Our approach is to use genetic programming to find efficient ranking functions which use data from many execution profiles. Our goal is to find a ranking function that is more efficient than those previously published in the literature. To this end, we have developed a Java program that uses en open source library for genetic programming called JGAP.

The input to this program is a set mathematical operators, Add, Subtract, Multiply, Divide, Log, Exp, and the names of variables from the 12 execution profiles.

After we get a ranking function from genetic programming, the evaluation of this ranking function is an important part too. If we get one ranking function, which contains component variables from several execution profiles, we need to design how to pick the passed/failed values in different execution profiles, since some execution profiles have multiple entries for one statement, just like path coverage in Figure 6. If we get several ranking functions we need to design how to combine them .

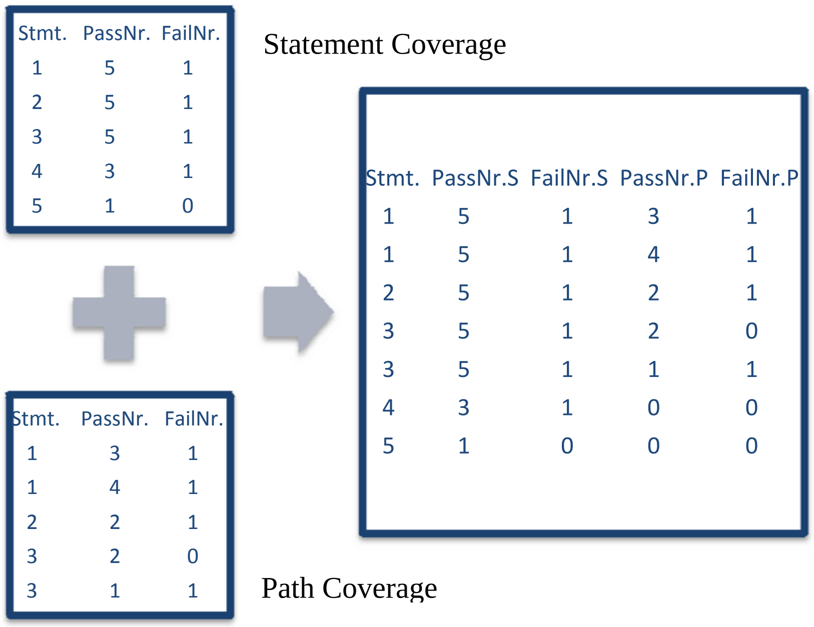
To evaluate a ranking function, we need to pick the values from the matrix of execution profiles and apply them to ranking function to calculate the ranking value for each statement. Finally, each statement has a ranking value; we sort the program according to these values and check where the faulty statement is. Then we know how well this ranking function is.

We have investigated three methods for combining execution profiles.

Combination Method 1:

In this method we try to learn a single ranking function which contains different execution profile variables. In order to solve the multiple entries problems, we need to associate different execution profiles one by one. That means we try to calculate all the possible ranking values for a statement when we pick the values from different execution profiles.

Figure 6 shows that in path coverage, statement 1 and 3 are executed by two paths separately. Therefore, statements 1 and 3 both have 2 entries from different paths. When we calculate the ranking value for statement 1, there is one possible value to pick for statement 1 in statement coverage and two possible values to pick in path coverage. Hence, the merge table should have 2 rows data for statement 1. Similarly, if a ranking function contains variables from branch coverage, path coverage and du-pair coverage and there are 2 entries in branch coverage 3 entries in path coverage and 4 entries in du-pair coverage for one statement. The possible number of ranking values for this statement is 2*3*4=24, which means that there are 24 rows data for this statement in the merge table. Among the 24 possible ranking values, we pick the highest as this statement's final ranking value.



Stmt.	PassNr.	FailNr.
1	5	1
2	5	1
3	5	1
4	3	1
5	1	0

Stmt.	PassNr.	FailNr.
1	3	1
1	4	1
2	2	1
3	2	0
3	1	1

Stmt.	PassNr.S	FailNr.S	PassNr.P	FailNr.P
1	5	1	3	1
1	5	1	4	1
2	5	1	2	1
3	5	1	2	0
3	5	1	1	1
4	3	1	0	0
5	1	0	0	0

Figure 6 – How to combine statement coverage and path coverage.

Unfortunately, it took us more than 12 hours to produce one generation with this combination technique. The reason was that the execution time increases rapidly with the number of profile variables that are associated with each statement. In our experiment, there are 5000 individuals in one population and 400 faults are used to evaluate the ranking function. Each fault is represented by data for 6000 statements. If we assume that there are 50 possible combinations for each statement, then the number of calculations required to produce one generation is 6000*5000*50*400 = 6000 0000 0000. This number does not include the sorting and compare operation. Due to this high computational complexity we had to give up this combination method.

Combination Method 2:

In method 2, we learn the best ranking function for each execution profile individually. This means that we obtain one ranking list for each execution profile. For profiles affected by the multiple entries problem, we pick the highest ranking value of a statement as its ranking value. The different ranking lists are merged into a single ranking list by assigning to each statement a ranking value that is equal to the sum of the statement's ranking values for each execution profile.

While the execution time for this method was acceptable, the results were poor. For this method, we encounter situations where we use ranking values based on branch coverage for locating bugs which are not associated with branches, which is ineffective. We therefore believe that this method is unlikely to produce ranking functions that more accurate than those used in existing fault localization techniques.

Combination Method 3:

In method 3, we learn four ranking functions. Each ranking function is associated with a specific fault type. The fault types are: (1) missing conditional statement, missing condition, (2) missing variable assignment, missing function call, (3) wrong variable assignment, wrong function parameter, wrong function call, and (4) wrong branch condition, missing term, missing branch. We merge the four ranking list into a single ranking list by selecting for each statement the best ranking value (lowest number) as the statement’s ranking value. This procedure is illustrated in Table 2 with three ranking functions.

Table 2 - Combination of several ranking functions based on best value

Stmt. No.	Formula1(Ranking position)	Formula2(Ranking position)	Formula3(Ranking position)	Final ranking position
1	15	827	278	15
2	369	27	47	27
3	274	381	763	274
4	32	748	62	32
…				

One of the design goals for this combination method is to improve localization of faults that are difficult to locate using a single ranking function. Table 3 shows the ranking positions assigned to three real faults from the GCC program by method 3 and by the Tarantula ranking function. The table shows that all three faults are difficult to locate. Tarantula assigns them ranking values between 59 and 97, while our method assigns them with ranking values between 15 and 61. Hence, for these faults combination method 3 shows a significant improvement over the Tarantula technique, even though the rankings are far from the perfect result (rank No.1).

Table 3- Example of Method 3 from GCC program

Faults	Ranking function1(Ranking position)	Ranking function2(Ranking position)	Ranking function3(Ranking position)	Total(Ranking position)	Tarantula(Ranking position)
Fault 1	215	457	12	24	71
Fault 2	26	890	1290	61	97
Fault 3	142	6	713	15	59

Moreover, we introduce a new method to calculate the ranking value and solving the multiple entries problem, which can improve the speed of learning. First we analyze the formula and determine which variable is using the highest/lowest value. We then use these values as input to calculate the ranking value. As an example, consider the following simple ranking function: failPATH(S) – passPATH(S). To ensure that this ranking function always get the highest value for each statement, failPATH(S) should use the maximum value in the MinMax table and passPATH(S) should use the minimum value. The format of a MinMax table shows in Table 4.

Table 4 - MinMax Table example

StmtNr.	Path Pass Max	Path Pass Min	Path Fail Max	Path Fail Min	Du-pair Pass Max	Du-pair Pass Min	Du-pair Fail Max	Du-pair Fail Min	…
1	63	32	4	1	47	24	4	2	
2	36	15	3	3	64	45	3	2	
3	73	23	2	0	31	23	4	1	
…									

5. Work process and experimental setup

In this section, we describe our work process, the fault data and relevant details about the genetic programming configuration.

5.1. Work process

Our work can be divided into nine activities as shown in Figure 7. In activity 1, we identified fault types which appear frequently in two real world programs: GCC and SPACE. To extend the fault set, we generated a set of mutation faults for GCC and SPACE in activity 2. For both programs, the number of real faults is about 200 and the number of mutation faults is about 5000.

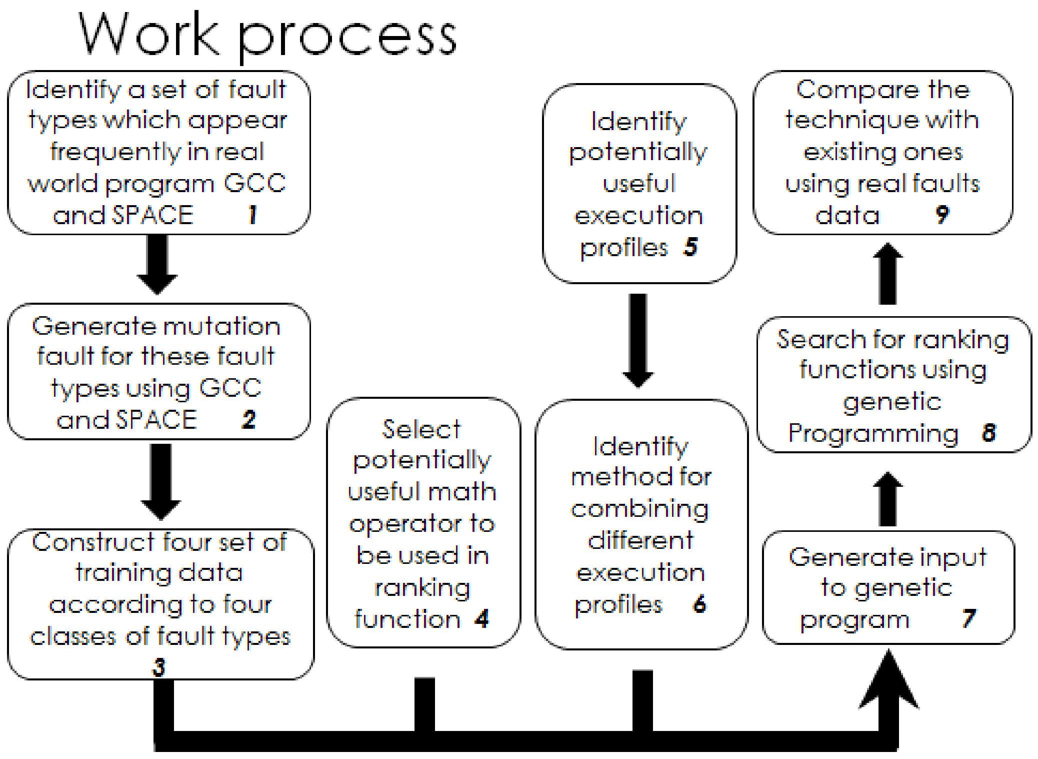

Work process
Identify a set of fault types which appear frequently in real world program GCC and SPACE 1
Generate mutation fault for these fault types using GCC and SPACE 2
Construct four set of training data according to four classes of fault types 3
Select potentially useful math operator to be used in ranking function 4
Identify potentially useful execution profiles 5
Identify method for combining different execution profiles 6
Generate input to genetic program 7
Search for ranking functions using genetic Programming 8
Compare the technique with existing ones using real faults data 9

Figure 7- Work process

We used these faults to construct four sets of training data corresponding to the four fault types used in combination method 3. In activity 4, we selected the potentially useful math operators which were used in the genetic research of ranking functions. In activities 5 and 6, we identified potentially useful execution profiles and investigated the three methods for combining data from different execution profiles.

We generated the input data for the genetic programming search in activity 7. The input data can be divided into two parts. One is the initial population of ranking functions. Here, each individual is generated by random selection of a set of execution profiles variables and a set of math operator. The other input is the matrixes of execution profile data for each fault. These are used in the fitness tests of the ranking functions.

In activity 8 we used genetic programming to search for ranking functions and use a fitness function to evaluate them. We use the Java open source library JGAP, which is a well-known library for implementation of genetic algorithm and genetic programming. Finally in activity 9, we compared the ranking function obtained in activity 8 and with four existing fault localization techniques.

5.2. Fault data

In our experiment, we need a lot of faults as input to the genetic program so that the program can learn the ranking functions precisely. As mentioned previously, the fault data are from GCC and SPACE. These two data sets are often used for evaluation of faults localization techniques since they consist of real faults from two widely used programs.

Each program is divided into several sub-programs. The average size of the subprograms is 4000 statements in SPACE and 10000 statements in GCC. These sub-programs can be divided to four subsets corresponding to the four fault types used by combination method 3. These fault types are not complete, but we believe that they are among the most common fault types.

Table 5 – The four subsets of fault types used by combination method 3

1	Missing conditional statement	Missing condition	
2	Missing variable assignment	Missing function call	
3	Wrong variable assignment	Wrong function parameter	Wrong function call
4	Wrong branch condition	Missing term	Missing branch

We need fault data not only for searching for new ranking functions but also for evaluating the final ranking function. The faults that are used for learning can clearly not be the same as the ones we use for evaluation, because we need to make sure the final ranking function is good for locating also other faults than those we use in the learning process. Therefore, we divide the fault data into two sets, one for learning and one for evaluating the final ranking function. There are about 1600 faults in the leaning set, 400 for each fault type. Since we have more than 5000 faults in total, we use more than 3400 faults for the evaluation of the final ranking function.

5.3. Mapping execution profiles into genetic programming

In order to make use of all execution profile variables in the genetic search, we need to map them to genetic programming variables. As we mention above, each execution profile consist of 4 variables. 4 new profiles, which are supersede, surpass, totalcoverage and equivalent, consist of only one variable. Each variable maps to one variable in JGAP. In fact, our experiment use the same test cases when it collects different execution profiles' data. So Totalfail and Totalpass can be considered as two constants. Hence, 12 execution profiles are represented by 8*2 + 1+1+1+1 + 2 = 22 variables in JGAP.

Operator set and terminal set are the variables that we need to input to genetic programming so that genetic programming can use them to generate ranking functions. Operator set consist of {+, -, *, /, log, Exp}. The terminal set is the variables that we map from the execution profiles: *passStat*, *failStat*, *failBRAN*, *passBRAN*, *passPATH*, *failPATH*, etc. (In total 22 variables). ranking.

5.4. Implementation of combination profiles

The MinMax value analysis function is defined as follows: first it creates a state table which contains one row for each profile variable that is used by the target ranking function. Table 6 shows an example of such a state table. This table has two columns, one is the name of the variable, for example *FailSTAT(S)*, and the other is the state of this variable. Initially the state of a variable is set to 0. After the analysis, it is set to 1, 2 or 3. A “1” means the variable should be assigned the highest value in the execution profile variable in each statement. A “2” means that the variable should be assigned the lowest value in the execution profile. A “3” means that the variable appears several times in the ranking function, and that at least one instance of the variable need the maximum value while least one other instances needs the minimum value. If a ranking function contains a variable with state 3, we discard that ranking function since we want to have only monotonic functions.

Table 6 - Example of state table produced by analysis function

FailSTAT(S) * PassSTAT(S) – (FailBRAN(S) + FailSTAT(S))	
FailSTAT(S)	3 (Mix)
PassSTAT(S)	1 (Max)
FailBRAN(S)	2 (Min)

5.5. Generation of ranking functions

Figure 8 shows the process of generating ranking functions. The first generation of ranking functions is selected totally random. We evaluate the fitness of each individual by computing a metric that reflects how well the individual can locate faults.

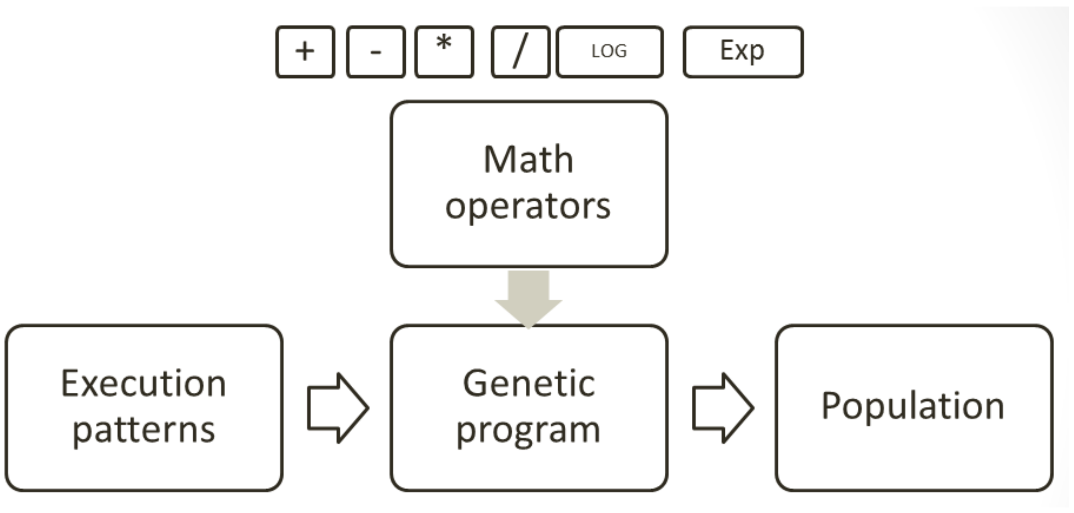


Figure 8- Process of Generating Ranking Function

Each new generation of ranking functions are created from the previous generation by applying the genetic operators described above. An individual which has a high fitness value have a higher probability of being chosen as the subject of a crossover or mutation operation.

The internal of crossover and mutant are defined by JGAP, we use it just set some of configuration. There are two constraints we need to consider:

- The length of each individual (ranking function) need to be limited
- Individuals need to be monotonic

Concerning constraint one, we found that short individuals worked better than long ones. Long individuals were able to locate fault in the learning set, but did not perform well in locating faults that did not belong to the learning set. In order to address this problem, we limited the depth of the tree structure to 6.

For constraint two, we find that all ranking functions of existing techniques are monotonic. We therefore decided to throw away all non-monotonic ranking functions. We used the MinMax value analysis function to identify non-monotonic individuals.

While these constraints limit our search space, we believe it is unlikely that we have missed ranking functions that are significantly more accurate than the ones we found.

5.6. Fitness function

The fitness function is used to evaluate the effectiveness of the individuals in each generation. The calculation of the fitness value is divided into 3 steps:

1. First each variable in the ranking function is analyzed in order to create the state table described in Section 5.4.
2. Program assigns the corresponding value to all variables to compute the ranking value of each statement according to the state table.
3. Using the ranking values for each fault to evaluate the fitness of the target ranking function and then return the fitness value back to genetic programming.

In step 1, method to analysis the ranking function is focus on tree structure. This method just can analyze the operator we define, such as multiply, add, subtract, divide, Log, Exp. Then output the state table we need.

In step 2, we produce a list to store the ranking values of all statements for each fault. After we compute the ranking value of all the statements, we use the ranking list to start step 3.

In step 3, using the ranking value of faulty statement as sample to check how many statements’ ranking value higher than sample. Then, we collect all these numbers for each fault and sum up them as target ranking function’s fitness value in genetic programming.

After genetic programming use fitness function to evaluate all the individuals, genetic programming starts to sort the population according to the fitness value. The individual which has highest fitness value will set as a sample to mutant and crossover for generating next generation.

5.7. Execution of the genetic program

Since the size of the learning set is large, we cannot run the genetic program within one computer. We designed the program to run in a distributed environment with a distributed file system. The program was divided into several parts, one master and many slaves. The master program handles the core data of genetic programming while the slaves evaluate the ranking functions within each generation concurrently.

Hence, after the population of a new generation has been generated, the master program put each individual into a temporary directory which is shared by the other computers. We then start the slaves to evaluate the individuals and write the result back to the temporary directory. After this, the master program searches the result of each individual in the temporary directory. The master program was run in a computer with a dual-core CPU while the slave programs were run on eight computers with 16 cores and on 10 computers with 4 cores.

Due to limitations in computing power, our experiments were terminated after approximately 200 generations. The population size was set to 5000, with a mutation rate of 0.05, and crossover rate of 0.1.

6. Result and analysis

As previously described, we used genetic programming to search for four ranking functions, one for each of the fault types described in Table 5. The four ranking functions are shown Table 7.

Table 7 - Formulas on four subsets fault types

Fault type no.	Ranking functions	Symbol
1	((SUPRASS * failCALLSTACK) + (passFUNC - passPAIR)) + (failSTAT^2 * SUPRASS)	(1)
2	(SUPRASS - passPAIR) * ((EQUIVALENT * passPATH-PAIR) + (SUPRASS * failPATH-CALLSTACK))	(2)
3	((Exp(failPAIR)) - passPAIR) * (SUPRASS * failCALLSTACK)	(3)
4	failPAIR * ((SUPRASS * failPREDICATE) - (passBRANCH - passPATH))	(4)

Table 8 shows the effectiveness of the ranking functions for their respective fault types. The table shows for each fault type the number faults having a fault localization effort below a certain percentage value. The fault localization effort for a given fault is determined by its ranking position and is expressed as the percent of statements that the developer must examine before he or she examines the faulty statement. (It is assumed that developer examines the statements in the order suggested by the ranking function.) As an example, we see that 120 out of 1731 faults of type 1 could be located by examining less than 0.1% of statements. We can conclude that a majority of the faults (76.8% for type 1, 79.3% for type 2, 86.9% for type 3, and 96.7% for type 4) can be located by examining no more than 10% of the statements..

Table 8 – Fault localization effort (percent of statements examined) for each fault type.

	Fault type 1.	Fault type 2	Fault type 3	Fault type 4.
0.1%statements examined	120	141	54	190
1% statements examined	648	582	528	696
5% statements examined	1216	1072	815	1095
10% statements examined	1329 (76.8%)	1360 (79.3%)	967 (86.9%)	1201 (96.7%)
20% statements examined	1645	1592	1027	1230
Total number of faults of this type in evaluation set..	1731	1716	1113	1242
Average % examined to locate all faults	5.198%	6.598%	9.811%	3.156%

However, when we debug a real program we do not know the type of the fault that we are looking for. We therefore combine the four ranking functions into a single ranking function, as described in Section 4 to locate the fault. Table 9 shows the result for the combined ranking function.

Table 9 - Result of total

	Total
0.1% statements examined	188 (3.235%)
1% statements examined	2056 (35.381%)
5% statements examined	3612 (62.157%)
10% statements examined	4364 (75.098%)
20% statements examined	5337 (91.843%)
Total faults number	5811
Average % locate all faults	6.440%

Table 9 shows that we are able to locate 62.157% faults within 5% statements examined and 91.843% faults within 20% statements examined. Total number of faults that we use for evaluation is 5811. Also, we compare our result to three other ranking functions, namely, Tarantula, Ochiai and Hybrid. The results are shown in Figure 9. We get about 5% improvement with 10% statements examined and 10% improvement with 20% statements examined comparing to the second best technique, which is the hybrid [2]. Tarantula [4] locates about 50% faults within 5% statements examined and 60% faults within 10% statements examined for our fault set.

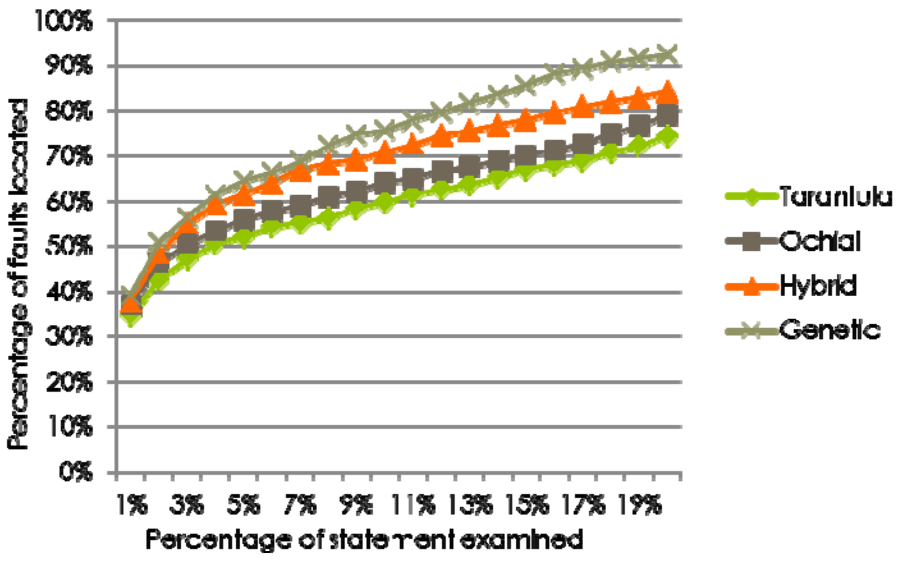


Figure 9 - Result of different techniques

7. Conclusion and future work

In Section 4 we introduce three methods to combine different execution profiles. We found that Combination Method 1 was infeasible due to its computational complexity and that Combination Method 2 gave bad results. Hence, the result part just contains data from Combination Method 3.

Our results show that the ranking functions derived through genetic programming performs better than the Hybrid, Tarantula and Ochiai ranking functions. Our technique was able to locate almost 92% of the faults by examining 20% of the statements. This is an improvement of more than 10% compared to the Hybrid technique, which was the second best technique. But this is not enough for real world program, since if the program has 1000K statements then 1% statements is 10K statements which is still a big time-consuming work.

Hence, I think there is existing one method which can improve much more than I found. Maybe there is one execution profile or a combination method that have not yet been explored.

Meanwhile, our experiment just focuses on single fault. But in real world, most of the program has multi-faults problems. So this problem is another big issue in future works.

Finally, computing power is the main issue in our experiment. Since if we have very powerful computing facilities we can use Combination Method 1 to make experiment much more precisely. Also, we can try more execution profiles in genetic programming. So we can try to re-run the experiment again if we can get powerful computer in future.

References

[1] Software Fault Localization, W. E. Wong and V. Debroy, (Section Authors), IEEE Transactions on Reliability, Volume 59, Issue 3, pp. 473-475, September 2010

[2] Raul Santelices, James A. Jones, Yanbing Yu, and Mary Jean Harrold. 2009. Lightweight fault-localization using multiple coverage types. In *Proceedings of the 31st International Conference on Software Engineering* (ICSE '09). IEEE Computer Society, Washington, DC, USA, 56-66.

[3] Rui Abreu, Peter Zoeteweij, Arjan J.C. van Gemund, "On the Accuracy of Spectrum-based Fault Localization," Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, pp. 89-98, Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007), 2007

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

[5] Xinming Wang, S. C. Cheung, W. K. Chan, and Zhenyu Zhang. 2009. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *Proceedings of the 31st International Conference on Software Engineering* (ICSE '09). IEEE Computer Society, Washington, DC, USA, 45-55.

[6] Tai-Yi Huang, Pin-Chuan Chou, Cheng-Han Tsai, and Hsin-An Chen. 2007. Automated fault localization with statistically suspicious program states. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems* (LCTES '07). ACM, New York, NY, USA, 11-20.

[7] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. 2009. HOLMES: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering* (ICSE '09). IEEE Computer Society, Washington, DC, USA, 34-44.

[8] Trotman, A. (2005). Learning to rank. *Information Retrieval*, 8(3), 359-381.

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

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

[11] Joel Emer and Nikolas Gloy. 1997. A language for describing predictors and its application to automatic synthesis. *SIGARCH Comput. Archit. News* 25, 2 (May 1997), 304-314.

[12] Maggie Hamill, Katerina Goseva-Popstojanova, "Common Trends in Software Fault and Failure Data," IEEE Transactions on Software Engineering, pp. 484-496, July/August, 2009

[13] George K. Baah, Andy Podgurski, and Mary Jean Harrold. 2010. Causal inference for statistical fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis* (ISSTA '10). ACM, New York, NY, USA, 73-84.

[14] Jones, J. A., Harrold, M.J., and Stasko, J., Fault Localization Using Visualization of Test Information, In Proc. of the 26th ICSE, Washington, USA, May, 2002

[15] Thomas Ball and James R. Larus. 1996. Efficient path profiling. In Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture (MICRO 29). IEEE Computer Society, Washington, DC, USA, 46-57.

[16]Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. 1998. An empirical investigation of program spectra. SIGPLAN Not. 33, 7 (July 1998), 83-90. DOI=10.1145/277633.277647

[17] T. Reps, T. Ball, M. Das, and .J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. ACM Software Engineering Notes, 22(6):432-439, Nov. 1997.

[18] M. Weiser, “Programmers use slices when debugging,” Communications of the ACM, 25(7):446-452, July 1982

[19] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. 2009. A practical evaluation of spectrum-based fault localization. J. Syst. Softw. 82, 11 (November 2009), 1780-1792. DOI=10.1016/j.jss.2009.06.035 http://dx.doi.org/10.1016/j.jss.2009.06.035

[20] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan,”Scalable statistical bug isolation,” in Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 15-26, Chicago, Illinois, June 2005