IJHSR
ijhighschoolresearch.org
■ RESEARCH ARTICLE

Automated Debugging System: Implementing Program Spectrum Analysis and Information Retrieval on Fault Localization

Yin-Siang Hao

Taipei Wego Private High School, No.50, Zhuhai Rd., Beitou Dist., Taipei City, 11254, Taiwan (R.O.C.); samyhao0507@gmail.com
Mentors: Yi-Ting Liao, Sun-Yuan Hsiehs

ABSTRACT: Debugging is often the most time-consuming phase during program development, lengthening the development time and lowering efficiency. Eventhough there are currently existing tools that help raise debugging efficiency, their functions are largely limited, as they only present a more comprehensive analysis behind the compiling and running processes to save complicated steps in debugging; however, developers are still required to test and verify the reasonability of every line of code. Unlike the current manual debuggers, this research aims to build a system that automatically detects bugs[a] in the programs through program spectrum analysis and information retrieval. In the section of program spectrum analysis, the system will statistically analyze every code block's probability of containing bugs in the input source code file according to the provided test cases, later forming an initial suspiciousness ranking based on previous calculations. Afterward, the system retrieves historical files that resemble the current source code and uses their suspiciousness rankings to modify the initial suspiciousness ranking, generating the final suspiciousness ranking as the system's output. This research integrates the two techniques and optimizes the formula of forming the suspiciousness values in program spectrum analysis and the comparison mechanisms in information retrieval, reaching performances of higher comprehensiveness and precision.

a. The word "bugs" here refers to logic errors that cause wrong answers or runtime errors, not syntax errors that lead to compile errors.

KEYWORDS: Systems Software; Algorithms; Program Analysis, Natural Language Processing, and information retrieval.

■ Introduction

Background:

In recent years, computer science and programming have become popular fields due to their infinite potential for innovations as well as their significant contributions to the planet. However, while developing projects ranging from single-file codes to cross-platform software, most programmers are struggling to debug – a process to find the errors that occurred in the codes and resolve them – and it increases inefficiency and time consumption during the process.

Computer programming can be divided into four phases: identifying problems, finding solutions, coding them, and debugging.[1] Relative to the first three phases, debugging often makes the least number of changes, yet it usually requires the longest time length. According to the CVP survey (Figure 1), debugging has accounted for 50% (312 billion US Dollars) of the global software developer wages, equivalent to the wages for designing and building programs.[2]

Currently, multiple tools can assist developers to find bugs, such as debuggers and reversible debugging software. By setting breakpoints and limitations in the debugger mode, users can stop at lines that they think are suspicious, track the execution routes, and monitor the changes of variables and memory allocations line by line.[3] On the other hand, reversible debugging software records all the memory access, computations, modifications to variables, as well as calls to the operating sys

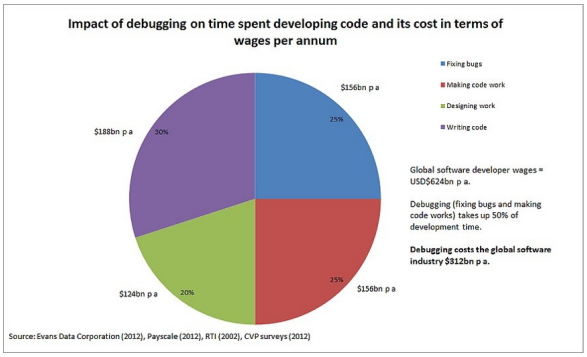

Figure 1: Impact of debugging on time spent developing code and its cost in terms of wages per annum.[4]

tems. By moving forward and backward among lines, the users can inspect the reasonability of the current program states and identify the errors in the codes.[4]

Motivation:

Although there are techniques that help developers to detect bugs more effectively, they still require developers to evaluate the correctness of each program state manually. Currently, although there is research discussing the validity of fault localization via spectrum analysis and information retrieval, the accuracy and effectiveness are not as significant as expected, and no tools are developed to automatically indicate potential bugs for users.

DOI: 10.36838/v5i4.17

Purpose:

Due to the lack of automatic debugging tools, this research aims to develop an automated debugging system that automatically locates potential bugs in given programs. The following are three primary objectives of this research:

1) Save the effort of the developers on assessing and locating bugs

2) Develop algorithms that can implement the functions of this system

3) Enhance the accuracy of the automated debugging outcome

System Architecture:

In the program spectrum analysis section, the buggy source code file is first inputted and processed. A new file is created by integrating the original codes and a line-tracing mechanism. Next, it is compiled and executed with input and output files provided by the users, and the system generates a coverage matrix consisting of m block-hit spectra, each recording whether the code blocks are traversed in the execution or not. The error log records the result of every execution. By comparing the similarity between the coverage matrix of each code block and error log, the system measures the suspiciousness values, or the probability of containing bugs, of the code block – the higher they resemble each other, the more possible that the code block is the reason that leads to failures and contains bugs. Finally, an initial ranked list of code blocks is generated based on the suspiciousness values.

In the information retrieval section, the input source code is vectorized into a term vector through the process of TF-IDF. It is then compared with the TF-IDF vector of every dataset, a collection of similar source code files, in the historical file database. Of all datasets, one dataset with the highest relevance score is chosen. In the dataset, all historical source code files are vectorized into term vectors through TF-IDF and compared with the term vectors of the input source code file, where a relevance score is generated. The final buggy code block ranking of each historical source code file and then alters the initial ranked list of the input source code file to the extent proportional to the relevance score between the historical and input files. Finally, a final ranking of the current code file is generated and outputted, indicating the suspiciousness values of each code block. (Figure 2)

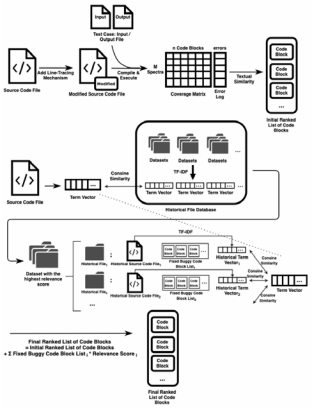

Figure 2: The algorithmic process of the implemented automated debugging system

Related Works

Since this research is based on a variety of algorithms and theories, the essential concept of each should be briefly explained to avoid further ambiguity. Multiple papers in support of this research are cited in this section for higher thoroughness and authority.

Program Spectra:

The program spectrum represents different perspectives toward a program and focuses on different features during program executions.[5] The two types of spectra are hit and count, of which the former only records true or false, and the latter records the number of times the spectrum is executed. Branch spectra only record the steps regarding conditional statements, such as "if", "for", and "while." Complete-path spectra track the complete routes of execution, including conditional branches, loops, and statements. Different from the complete-path spectra, path spectra only record partial paths based on an acyclic control flow graph, exclusive of any loops. Different from the normal control flow graph, the acyclic eliminates the back edges that form loops, becoming loop-free. Data-dependence spectra record definition-use pairs, each of which has the form (d, u, v), respectively meaning the definition statement of a variable, statements using the variable, and variable name. Output spectra save the output of the execution. Similar to complete-path spectra, execution-trace spectra record the entire route; yet, the main difference between them is that execution trace includes real codes, whereas complete-path spectra only contain line numbers.[6] Block spectra form program blocks that compound statements.[7] For example, statements under if or else are included in the same block because they are always run together under an execution. Table 1 illustrates the spectra, including its profiled code lines, and execution records based on hit and count, for example, the program Number of n Digits in Figure 3.

```
1   int n;
2   cin >> n;
3   int count = 0, output = 0;
4   while(count =≠ n) {
5       int num;
6       cin >> num;
7       output = output * 10 + num;
8       count++;
9   }
10  cout << output << endl;
```

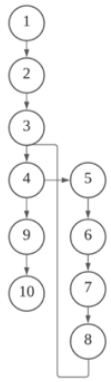

Figure 3: Example code *Number of n Digits* and its control flow graph

Table 1: Spectra for *Number of n Digits* of Figure 3

Spectrum	Profiled Entities	Executions (Hit / Count)	
		Execution 1 (Input: 0)	Execution 2 (Input: 2 5 7)
Branch	(2, 3)	T/ 1	T/ 1
	(4, 5)	F/ 0	T/ 2
	(4, 9)	T/ 1	T/ 1
Complete-Path	(1, 2, 3, 4, 9, 10)	T/ NA	F/ NA
	(1, 2, 3, 4, (5, 6, 7, 8, 4)^2, 9, 10)	F/ NA	T/ NA
Path	(1, 2, 3, 4), (9, 10)	T/ 1	T/ 1
	(1, 2, 3, 4, 9, 10)	T/ 1	F/ 0
	(1, 2, 3, 4, 5, 6, 7, 8, 9, 10), (5, 6, 7, 8)	F/ 0	T/ 1

Data-Dependence	(1, (2, 4), n), (3, 10, output)	T/ 1	T/ 1
	(3, (7, 10), output)	F/ 0	T/ 1
	(5, (6, 7), num)	F/ 0	T/ 2
Output	output = 0	T/ 1	F/ 0
	output = 57	F/ 0	T/ 1
Execution Trace	(int n, cin ≫ n, int count = 0, output = 0, while (count != n), }, cout ≪ output ≪ endl)	T/ 1	T/ 1
	(int num, cin ≫ num)	F/ 0	T/ 1
Block	(1, 2, 3, 4)	T/ 1	T/ 1
	(5, 6, 7, 8)	F/ 0	T/ 2
	(9, 10)	T/ 1	T/ 1

Spectrum-Based Fault Localization:

Spectrum-based fault localization, or SBFL, evaluates the suspiciousness of every program block. This technique measures the frequency at which each block is executed during failed executions, and this number of frequencies is seen as the suspiciousness value of this program block. There have been different program spectra proposed for this technique, and the most commonly used is block-hit, due to the high availability of its result and the low cost of collecting them.[8] In a process of spectrum-based fault localization, provided test cases are used for the execution of the source code program. During every execution, program blocks are marked with dots if they are executed, as shown in Figure 4. This forms the coverage matrix, which has a row number equal to the number of test cases and a column number equal to the number of program blocks, as shown in Figure 5. After the execution, the system will get a list of outcomes, each representing the success or failure of execution, known as the error vector. In Figure 4, the *error vector* is in the last row labeled "Execution results."[5]

The process can be shown in another form with only matrices, presented in Figure 5. M spectra indicate the number of runs of the program, and N stands for the number of code blocks. The M runs generate M results, each recorded either with 0 for successful (no errors) or 1 for failed (with errors), recorded in the error vector.[7]

ID	Program block	T_1	T_2	T_3	T_4	N_{CF}	N_{CS}	N_S	N_F	Suspiciousness	Ranking
b_1	int count n; Ele *proc; List *src_queue, *dest_queue; if (prio > = MAXPRIO) { /*MAXPRIO = 3*/	•	•	•	•	1	3	3	1	0.5	2
b_2	return;	•				0	1	3	1	0	3
	}										

b_3	src_queue = prio_queue[prio];
	dest_queue = prio_queue[prio + 1];
	count = src_queue- > mem_count;
	if (count > 1) {
			•	•	•	1	2	3	1	0.6	1

b_4	/* **BUG: It should be if (count** > =1) */
	n = (int) (count*ratio + 1);
	proc = find_nth(src_queue,n);
			•	•		0	2	3	1	0	3

b_5	if (proc) {
	src_queue = del_ele(src_queue,proc);
	proc- > priority = prio;
	dest_queue = append_ele(dest_queue,proc);
	}
			•	•		0	2	3	1	0	3

	}										
Execution results		S	S	S	F						

Figure 4: An example of spectrum-based fault localization 5

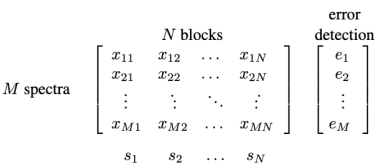

Figure 5: The Coverage Matrix and Error Vector 7

The purpose of calculating the suspiciousness value of every code block is to find to what extent a block reflects the error vector in the M runs. The more closely they resemble each other, the more probable of the block being the bug. This deduction is based on the fact that if the block is involved in executions that turn out to be the failed ones, it may be the factor that leads to the program's failure, thus being where the bug is located.[9] Measures for the similarity between the vector of $x_{1,J}$ to $x_{M,J}$ and the error vector are called *similarity coefficients*. There are four kinds of similarity coefficients, each calculated through a different formula, namely Jaccard, Tarantula, AMPLE, and Ochiai.[10]

Jaccard: $s_j = \frac{a_{11}}{a_{11}+a_{01}+a_{10}}$

Tarantula: $s_t = \frac{\frac{a_{11}}{a_{11}+a_{01}}}{\frac{a_{11}}{a_{11}+a_{01}}+\frac{a_{10}}{a_{10}+a_{00}}}$

AMPLE: $s_a = \left|\frac{a_{11}}{a_{11}+a_{01}} - \frac{a_{10}}{a_{10}+a_{00}}\right|$

Ochiai: $s_o = \frac{a_{11}}{\sqrt{(a_{11}+a_{01})\times(a_{11}+a_{10})}}$

$a_{00} = |\{i|x_{ij} = 0 \wedge e_i = 0\}|$, *the number of successful runs in which block j is not involved.*

$a_{00} = |\{i|x_{ij} = 0 \wedge e_i = 1\}|$, *the number of failed runs in which block j is not involved.*

$a_{10} = |\{i|x_{ij} = 1 \wedge e_i = 0\}|$, *the number of successful runs in which block j is involved.*

$a_{11} = |\{i|x_{ij} = 1 \wedge e_i = 1\}|$, *the number of failed runs in which block j is involved.*

TF-IDF:

TF-IDF, which stands for "term frequency-inverse document frequency," evaluates the importance of a word in a document based on its occurring frequency in a document and the corpus. As seen in the mathematical definition below, it is the product of term frequency and inverse document frequency.[11]

$$tfidf(t,d,D) = tf(t,d) \cdot idf(t,D)$$

$$t = term, d = document, D = set\ of\ document$$

Term frequency represents the frequency of a word in a document. Mathematically, it is the number of a word's occurrence in the document over the word count of the document.[11]

$$tf(t,d) = \frac{f_{t,d}\ (frequency\ of\ t\ in\ d)}{\sum_{t' \in d} f_{t',d}\ (total\ number\ of\ words\ in\ d)}$$

Inverse document frequency indicates the universality of a word in the corpus. Mathematically, it is calculated by dividing the total number of documents in the corpus by the number of documents with the term t included. The lower this number is, the more common t is, and vice versa.[11]

$$idf(t,D) = log\ log\ \frac{|D|\ (total\ number\ of\ documents\ in\ the\ corpus)}{|\{d \in D: t \in d\}|\ (documents\ in\ the\ corpus\ that\ include\ term\ t)}$$

Cosine Similarity:

Cosine Similarity is an approach that calculates the similarity between two documents. The documents are presented in the form of vectors, with each value representing the term frequency of a word. Then, the cosine formula of vectors is applied to the measurement of the distance between two vectors.[12]

$$sim(a,b) = \frac{a \cdot b}{\|a\|\|b\|}$$

In the formula, the similarity is calculated by dividing the product of vectors a and b by the product of the two vectors' lengths. The length of a vector is measured using the Euclidean norm. It is defined as the square root of the sum of the square of every vector component.[13]

$$A\ vector\ \vec{V}\ of\ i\ components, \|V\| = \sqrt{V_1^2 + V_2^2 + V_3^2 + \cdots + V_i^2}$$

DOI: 10.36838/v5i4.17

Information Retrieval- Based Fault Localization:

Information Retrieval-Based Fault Localization, or IRFL, aims to find out a ranked list of program elements based on their probability to be bugs. Throughout the process, it uses bug reports, documents that contain specific information about the failure of a program, to generate textual similarities with each program element, such as "for," "if," and "while," and rank them using these relevance scores. The technique that most system uses to calculate relevance scores is a combination of TF-IDF and cosine similarity. In the TF-IDF section, the compared documents (bug report and program element files) are changed into a vector of numbers, each representing the importance of every word. Then, the cosine similarity formula will be applied to calculate the distance (in this case, the similarity) between vectors.

Two major relevance functions carry out document comparisons, respectively direct and indirect relevance functions. In the direct relevance function, the relevance score between the current bug report and each program element file is calculated, creating an initial ranking of program element files. In the indirect relevance function, the current bug report is first compared with every history bug report related to the current case, with its relevance score calculated. Then, the system finds the program elements fixed in every history bug report, and multiplies the relevance score between the history bug report and fixed elements to the previous relevance scores. This final score turns out to be the indirect relevance score between the current bug report and those fixed elements.[14]

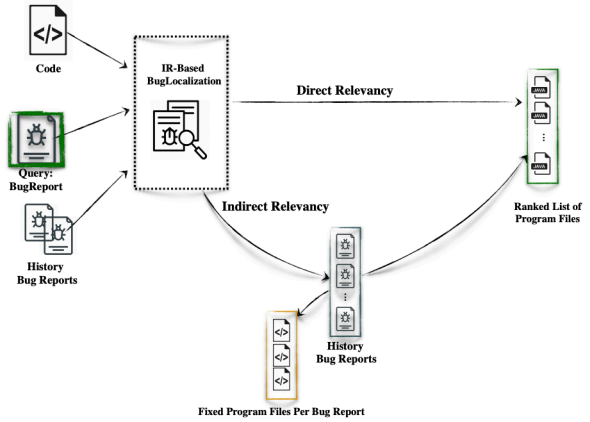

Figure 6: The process of direct and indirect relevance functions in IRFL[10]

Combining the results of the two relevance functions, the system will generate an ultimate program element ranking that indicates their ranked suspiciousness to be bugs, as shown in Figure 6.

Methods

This research mainly focuses on the debugging of the C++ programming language and uses C++ as the language for the implementation of the automated debugging system.

Facilities:

The hardware used in this research includes a laptop (CPU: 2 GHz Quad-Core Intel Core i5 with Intel Iris Plus Graphics, Memory: 16 GB, SSD: 512GB) for researching, developing the automated debugging system, conducting and analyzing experiments, as well as a notebook for recording the experimental data. The development environment for the automated debugging system is Visual Studio Code, and the programming language for development is C and C++

Program spectrum analysis:

In this section, the system aims to generate a suspiciousness sequence that indicates the suspiciousness of code blocks to contain bugs. The most important components in the measurement of suspiciousness values are the execution paths, which constitute the coverage matrices, and the execution result, which constitutes the error vectors. The process can be divided into three main steps: Modifying the source code file to make it fit the following operations, executing the source code file, and analyzing the data collected in the previous executions.

Since the initial source code files don't have the functions of collecting execution path and result, a new file is created by the system, including the initial codes along with additional mechanisms, as shown in Figure 7.

Figure 7: The contrast between an input source code file (left) and its modified source code file (right)

__LINE__ is added to the end of every available line. It is a preprocessor macro that provides the line number of the current statement.[15] The variable bracket is declared to record the level of curly braces the current statement is wrapped in. A line is available to add *__LINE__* when bracket > 0, or the current statement is wrapped in at least one curly braces pair, and when the statement is not ended with a closing curly brace. During every execution, the value of *__LINE__* is recorded in route.txt, which will contain a complete execution path when the execution finishes.

In the program, the modified source code file is executed when pairs of input and expected output files are provided. To compile and execute with the code file, the function *system*() is needed. It invokes the command-line interface to execute commands given as the function's parameters.[16] After every execution, the output file is checked by comparing it with the expected output file provided by the users, after which the result is generated (Figure 8).

Figure 8: Compilation and Execution of source codes with input and output files

When all executions end, the system generates a set of route files and an error log, indicating whether an execution fails. Using these data, the system calculates the numbers of successful and failed execution that each line of code is involved in and the total number of successful and failed executions. An optimized Tarantula Formula is used as the program spectrum formula to identify every line of code's suspiciousness value:

$$Sus_i = \left(\frac{\frac{n_{i,F,1}}{n_{i,F,0}+n_{i,F,1}}}{\frac{n_{i,F,1}}{n_{i,F,0}+n_{i,F,1}}+\frac{n_{i,S,1}}{n_{i,S,0}+n_{i,S,1}}}, \frac{1}{u+1} \times \frac{\frac{n_{i,S,1}}{n_{i,S,0}+n_{i,S,1}}-\frac{n_{i,F,1}}{n_{i,F,0}+n_{i,F,1}}}{\frac{n_{i,F,1}}{n_{i,F,0}+n_{i,F,1}}+\frac{n_{i,S,1}}{n_{i,S,0}+n_{i,S,1}}} \right)$$

$$, \{n_{i,F,1} = |\{j|R_j \exists i, E_j = 1\}| \; n_{i,F,0} = |\{j|R_j \nexists i, E_j = 1\}| \; n_{i,S,1} = |\{j|R_j \exists i, E_j = 0\}| \; n_{i,S,0} = |\{j|R_j \nexists i, E_j = 0\}|$$

Where u equals the number of times the latter parameter of max function is the maximum. Besides the original Tarantula formula, a new formula calculating the probability of another circumstance is generated and compared with the original one. It considers the possibility that bugs come from "not passing correct code blocks," which differs from the original formula examining bugs from executions "passing certain buggy code blocks."

However, it is risky to expect all code blocks to be passed to get successful outcomes since some are open to restricted conditions and designed not to pass in these failed test cases. Thus, 1/(u+1) is applied as a coefficient to rationalize the probabilities.

After the suspiciousness value calculations of every line, adjacent lines with the same suspiciousness values are combined into blocks of code *codeBlocks*[*i*] and sorted so that their suspiciousness values, *codeBlocks*[*i*].*sus*, are arranged in descending order, becoming a ranked list of code blocks.

Information Retrieval:

The goal of applying information retrieval is to utilize previous debugging experiences to help optimize the accuracy of the current suspiciousness value of each code block. The historical file database provides information that contributes to the optimizations during the process.

The database contains folders indicating highly relevant sets of historical debugging analysis. In every historical debugging analysis, all information about the code blocks of the historical source code file is recorded, including every code block's starting, ending line, suspiciousness value, and description.

Throughout this section, the TF-IDF vectorization function is used to help evaluate the relevance among documents and suspiciousness sequences. The input text is first preprocessed through a series of text preprocessing methods. Non-alphabetical and non-numerical characters are removed, all characters are converted to lowercase, and all words in the text are tokenized into string vector *vocabList*. Finally, the tokens are stemmed, or to remove their inflections to simpler forms of words, using OleanderStemmingLibrary.[17]

Two maps record the token's term frequency (TF) and inverse term frequency (IDF), respectively. The former counts the occurrences of every term in *vocabList* and divides them by the total number of tokens in the text. The latter uses Code Description Corpus (Figure 9) to measure the document frequency. The corpus is read and outputted as token list *allVocabList*[*i*] referencing document i in the corpus. During the calculation of every term's inverse document frequency, the system iterates over *allVocabList*[*i*] and identifies whether the document contains the term through *binary_search*.

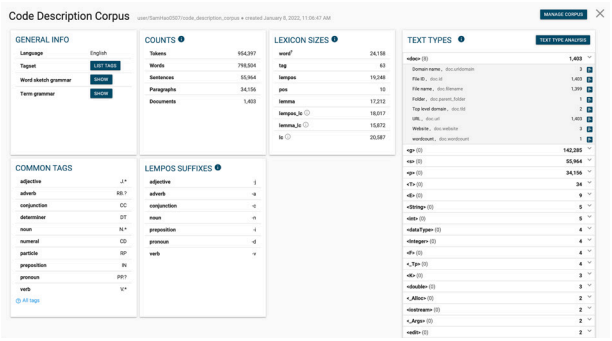

Figure 9: Code Description Corpus

First, the system vectorizes the general description of the current source code provided by the users into the TF-IDF vector, calculates the relevance score between the vector and all folders' description vectors using the cosine similarity formula, and finds the most relevant folder among the database, which is the primary reference for information retrieval in the following processes.

The system uses cosine similarity to measure the relevance between the current source code file and historical debugging analysis to determine how the analysis can affect the current suspiciousness ranking. The relevance score is a multiplication of three sub-relevance scores, namely *fileRelevanceScore*, *bugRelevanceScore*, and code block suspicion relevance, *cbMatchSus*[*i*] of the *i*th code block.

Based on the provided code block descriptions of the current and historical source code file, the system pairs *codeBlocks*[*i*] with the most relevant code block of the historical file. *cbMatch*[*i*], where *i* is the code block's index of the current source code file, indicates the index of *codeBlocks*[*i*]'s corresponding code block of the historical source code file. For every *odeBlocks*[*i*], the system calculates the cosine similarity between code block description *codeBlocksDesc*[*i*] and *cmpCodeBlocksDesc*[*i*], and finds the historical code block, *cmpCodeBlocks*[*cbMatch*[*i*]], whose description has the maximum cosine similarity with *codeBlocksDesc*[*i*].

With the matches of code blocks, the system vectorizes the string form of the combination of all code block descriptions codeBlocksDescStr and the corresponding historical code block descriptions *cmpCodeBlocksDescStr*, and generate the relevance score *fileRelevanceScore* using cosine similarity.

While *fileRelevanceScore* represents the similarity of content between the current and historical source code file, *bugRelevanceScore* indicates the consistency of bug conditions between the two files. It is the cosine similarity between the ranked suspiciousness sequence and the corresponding suspiciousness sequence of the historical source code file.

Besides *fileRelevanceScore* and *bugRelevanceScore*, *cbMatchSus*[*i*] indicates the consistency of bug conditions between the ith pair of code blocks. It is calculated through restricted growth formula to extremize the values at both ends (0% and 100%):

DOI: 10.36838/v5i4.17

$$cbMatchSus[i] = e^{-k\Delta sus}$$

Where Δsus is the difference of suspiciousness value between the pair of code blocks. Logically, a difference of 0.5 in suspiciousness value indicates a 50% suspiciousness relevance of the two code blocks since the case is placed under an ambiguous circumstance where the possibility of being completely irrelevant equals that of being completely relevant. Therefore, by substituting Δsus with 0.5 and $cbMatchSus[i]$ with 50%, $-k=ln\ln$ 0.5/50% ≈ -1.38629436112. Therefore, the formula presented is:

$$cbMatchSus[i] = e^{(-1.38629436112)\times \Delta sus}$$

Multiplying the three relevance scores gets the final relevance score for updating the current suspiciousness ranking.

After retrieving the historical file's result, $buggyCodeBlocks[i]$ records the index of the final fixed buggy code blocks in the historical file. The system refers back to the corresponding code block of the current source code file using *cbMatch*, and adds the relevance score to *updateWeight*, which measures the final weight of updating the suspiciousness sequence:

$$updateWeight[i] = \frac{\sum_j \square\, (relevance\ score_{file_{j,i}})^2}{\sum_j \square\, relevance\ score_{file\ j,i}}$$

Where $file_{j,i}$ is the file with one of its fixed buggy code blocks index equal to $cbMatch[i]$. The final suspiciousness value $finalSus[i]$ is updated according to $updateWeight[i]$:

$$finalSus[i] = codeBlocks[i].sus + (1 - codeBlocks[i].sus) \times updateWeight[i]$$

After the process, the code block information of current source code files is written to a new debugging analysis and submitted to the database.

Results and Discussion

In order to test the system's accuracy of correctly indicating bugs in the source codes, multiple pairs of buggy and fixed source code files written in c++ are used to examine the consistency of the output suspiciousness rankings and the buggy lines fixed in the correct source code file.

Data and Preprocessing:

In order to exhibit the validity of the information retrieval section, the testing focuses on one coding problem *Wanna Go Back Home* from *AtCoder Grant Contest 003*.[18] Throughout the testing, 15 pairs of buggy and fixed source code files are randomly chosen from the submission page of the problem (Figure 10). Each pair of buggy and correct source code files are written by the same user in AtCoder. The buggy source codes are labeled "WA (Wrong Answer)" in the online judge, and the correct one with "AC (Accepted)" (Figure 10).

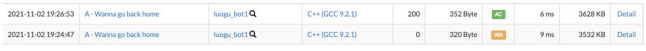

Figure 10: One of the pairs of buggy and correct source code files for Wanna Go Back Home

Figure 11: Example Buggy Source Code File (Left) and Correct Source Code File (Right) of the source code files pair from AtCoder Beginner Contest 216 Task A – Signed Difficulty

Source code and code block descriptions are generated for every pair of buggy and fixed source code files. Source code descriptions are the paraphrased form of the problem statement and are unique from other source code descriptions, while code block descriptions present the content of code blocks in the form of plain words, as close to the codes as possible.

Complete input and output test case folders are downloaded from its official folder atcoder_testcases on Dropbox.[19] All of the test cases are involved in the execution of the buggy source code files.

Preprocessing:

1. Removal of Comments

In order to condense the code length and unnecessary runtime, additional comments are removed.

2. Addition of curly braces for single-line loop or conditional statements

Since the line "route ≪ to_String(__LINE__) ≪ "";"is added to the end of a line, single-line loop or conditional statements disable the mechanism to detect whether the statements within are passed. Besides, if additional lines are added after an *if* that is followed by an else if or else, the syntax will be incorrect and lead to compile errors. Therefore, curly braces are added to the statements' end to wrap the content in curly braces.

Result

Among the 20 pairs of buggy and correct source code files, 48 code blocks are buggy and fixed, each receiving a final suspiciousness value and rank.

The suspiciousness values of the buggy code blocks are concentrated between 70% to 89%, illustrating that the system has highlighted most of the buggy code blocks as highly suspicious, as shown in Figure 12.

Also, 70.83% of the buggy code blocks are ranked above the 70th percentile (Figure 13), indicating that these code blocks are also considered as being most likely to contain bugs compared to other code blocks in the source code.

Overall, the suspiciousness values of the buggy code blocks are updated by an average of 22.08% from program spectrum analysis to the information retrieval section (Figure 14). The significant improvement of the suspiciousness values demonstrates the effectiveness of information retrieval on relevant

historical files and increases its accuracy in identifying buggy code blocks.

Besides the rank prominence, the calculated suspiciousness values of buggy code blocks also have outstanding suspiciousness values compared to other correct code blocks. The overall average of the deviation score of the buggy code block's suspiciousness values is 0.8, and 75% of them have deviation scores above 0.8 (Figure 15), signifying that the suspiciousness values of the buggy code blocks are not only ranked top but distant to that of other code blocks in the suspiciousness rankings (nearly one standard deviation away from the average).

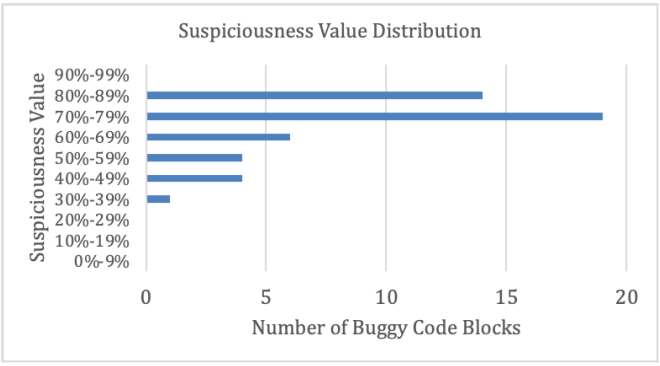

Figure 12: Suspiciousness Value Distribution of Buggy Code Blocks

Table 2: Mean (± Standard Deviation) of the suspiciousness value, rank, and absolute accuracy of the fixed code blocks

Average Suspiciousness Value of Buggy Code Blocks	Average Ranking of Buggy Code Blocks	Average Percentile Rank of Buggy Code Blocks	Absolute Accuracy (Buggy Code Blocks ranked No. 1)
70.94% ± 13.32%	No. 2.63 ± 2.10 / 11.08	76.60% ± 14.69%	43.75%

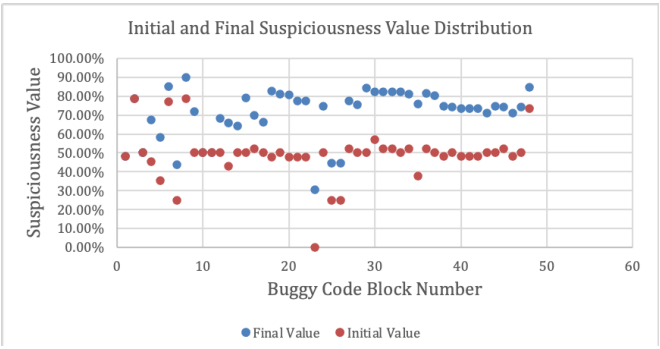

Figure 14: Initial and Final Suspiciousness Value Distribution of Buggy Code Blocks

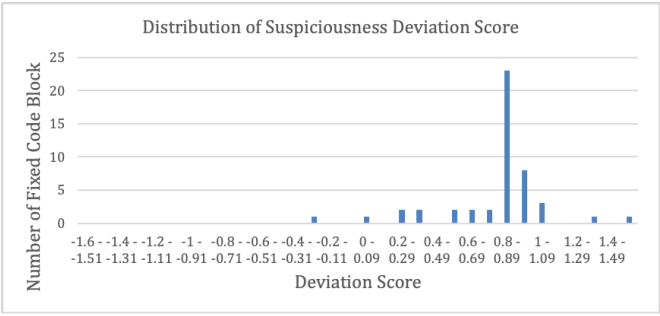

Figure 15: Distribution of Buggy Code Block's Suspiciousness Value Deviation Scores

Discussion

After testing with the source code files, multiple conclusions are drawn from the statistical results.

Problems:

1. Weakened effectiveness under all-WA situations

Based on the program spectrum formula:

$$\left(\frac{\frac{n_{i,F,1}}{n_{i,F,0}+n_{i,F,1}}}{\frac{n_{i,F,1}}{n_{i,F,0}+n_{i,F,1}}+\frac{n_{i,S,1}}{n_{i,S,0}+n_{i,S,1}}}, \frac{1}{u+1}\times\frac{\frac{n_{i,S,1}}{n_{i,S,0}+n_{i,S,1}}-\frac{n_{i,F,1}}{n_{i,F,0}+n_{i,F,1}}}{\frac{n_{i,F,1}}{n_{i,F,0}+n_{i,F,1}}+\frac{n_{i,S,1}}{n_{i,S,0}+n_{i,S,1}}}\right),$$

when there are only failed cases, $n_{i,S,0}$ and $n_{i,S,1}$ both equals 0, making the suspiciousness value 100%, which represents definite bugginess of the code block and is therefore inaccurate.

2. High Time Complexity

The time complexity of this system is $O(n_f \times n_c^2 \times n_{term} \times n_{doc} \times \log \log n_{token})$ where n_f is the number of historical files in the most relevant folder, n_c is the number of code blocks in a source code file, n_{term} is the number of terms in an input text file, n_{doc} is the number of documents in the corpus, and n_{token} is the number of tokens in each document of the corpus. This system becomes time-consuming when n_c is large.

3. Inaccurate Code block Matchings

Occasionally, the current source code blocks are matched with irrelevant historical code blocks, making the result inaccurate.

4. Noise Problem

Since the system uses relevance scores as the updating weights, files and code blocks with little relevance still affect the final suspiciousness ranking, making it less accurate.

Solution:

To solve the problem that occurred under all-WA conditions, the design of the program spectrum formula is changed by adding special cases to make the suspiciousness value be 0.5 under the condition that $n_{i,S,1}/n_{i,S,0}+n_{i,S,1}$ equals 0 and $n_{i,F,1}/n_{i,F,0}+n_{i,F,1}$ doesn't. The database then updates the suspiciousness sequence according to the retrieval of the historical debugging analysis, therefore enhancing the accuracy of the result. An example is shown in Figure 16, where initially the suspiciousness values of all the potentially buggy code blocks are 100%, yet through the optimization mentioned above, the more reasonable result of the program spectrum section provides space for the information retrieval section to update the values according to relevant historical files.

	Suspiciousness Value (Before)	Suspiciousness Value (After)	Buggy Code Blocks
Line 19 ~ 22	100.00%	74.79%	✓
Line 6 ~ 16	100.00%	73.78%	
Line 24 ~ 24	100.00%	69.08%	

Figure 15: Buggy Source Code of AGC003_12.cpp (Left), Comparison of its suspiciousness values before and after the optimization is implemented (Right)

DOI: 10.36838/v5i4.17

Solution Proposal:

1. After calculating the IDF value of terms, the system can save the results onto the database and directly retrieves them when there are identical input terms afterward. This approach is especially helpful for common terms.
2. To solve the inaccuracy of code block matching and noise problem, changing the code block matching algorithm from textual similarity to control flow graph comparison is more accurate regarding the similarity of the functions of the code block pairs.
3. Increase the weight of relevance score in the update weight formula (i.e., change the numerator from $\sum_j \left(relevance\ score_{file_{j,i}}\right)^2$ to $\sum_j \left(relevance\ score_{file_{j,i}}\right)^3$). Simultaneously, broaden the number of historical files and folders to provide ample and more likely valuable references. Once the system can retrieve sufficient relevant files, the irrelevant ones will have little impact on the final suspiciousness rankings.

Conclusion

Summary of Findings:

By implementing program spectrum analysis and information retrieval in the system, it is demonstrated that the system can detect and rank most of the buggy code blocks as highly suspicious, reflected in the final code block ranking. To further improve the system's accuracy and efficiency in localizing bugs, calculations should be saved and used when the next similar request is made. Also, the code block matching algorithm can be changed to control flow graph comparison to keep the matchings consistent with the relevance of code blocks' content. At the same time, the database should constantly expand to handle a wider range of source code files.

Future Prospects and Applications:

Currently, this system only supports single-file code projects written in c++. In the future, it is expected to expand the types of supported programming languages and give more users access to the system.

The system can also be integrated into code editors and IDE (Integrated Development Environment) and combined with debuggers. By utilizing the large quantities of source codes developed on the platforms and the comprehensive functions of debuggers, the system can expand the database and implement higher quality fault localization on the input source codes.

By saving the developers' development time, this system can ultimately increase the production of software programs and even fasten technological growth.

Acknowledgments

I want to acknowledge and give my warmest thanks to my two advisors, Teacher Yi-Ting Liao and Dr. Sun-Yuan Hsieh. Without their guidance on the content, overall structure, and format of my research, I wouldn't be able to achieve the current quality.

References

1. Valenzuela, Jorge. "Computer Programming in 4 Steps." ISTE, 20 Mar. 2018, https://www.iste.org/explore/Computer-Science/Computer-programming-in-4-steps.
2. Britton, Tom, *et al*. Reversible Debugging Software "*Quantify the Time and Cost Saved Using Reversible Debuggers*" Nov. 2020, https://www.researchgate.net/publication/345843594.
3. Sanjeev Kumar Aggarwal; M. Sarath Kumar (2003). "Debuggers for Programming Languages". In Y.N. Srikant; Priti Shankar (eds.). *The Compiler Design Handbook: Optimizations and Machine Code Generation*. Boca Raton, Florida; CRC Press. pp. 295–327. ISBN 978-0-8493-1240-3.
4. "What Is Reverse Debugging, and Why Do We Need It?" *Undo.io*, https://undo.io/resources/reverse-debugging-whitepaper/.
5. Arrieta, Aitor, *et al*. "Spectrum-Based Fault Localization in Software Product Lines." *Information and Soft ware Technology*, vol. 100, 2018, pp. 18–31., https://doi. org/10.1016/j.infsof.2018.03.008.
6. Harrold, Mary Jean, et al. "An Empirical Investigation of the Relationship between Spectra Differences and Regression Faults." *Software Testing, Verification and Reliability*,vol. 10, no. 3, Sept. 2000, pp. 171–194., https://doi.org/10.1002/1099-1689(200009)10:3<171::aid-st vr209>3.0.co;2-j.
7. Abreu, Rui, *et al*. (2007). "On the Accuracy of Spectrum-based Fault Localization." *Proceedings - Testing: Academic and Industrial Conference Practice and Research Techniques*, TAIC PART-Mutation 2007. 89-98. 10.1109/ TAIC.PART.2007.13.
8. LE, Tien-Duy B., *et al*. (2015). "Should I follow this fault localization tool's output? Automated prediction of fault localization effectiveness." *Empirical Software Engineering*. 20, (5), 1237-1274. Research Collection School Of Computing and Information Systems.
9. Abreu, Rui, *et al*."A Practical Evaluation of Spectrum-Based Fault Localization." *Journal of Systems and Software*, vol.82, no. 11, 2009, pp. 1780–1792., https://doi.org/10.1016/j. jss 2009.06.035.
10. Abreu, Rui & Zoeteweij, Peter & Gemund, Arjan. (2007). An Evaluation of Similarity Coefficients for Software Fault Localization. Proceedings - 12th Pacific Rim International Symposium on Dependable Computing, PRDC 2006. 39 - 46. 10.1109/PRDC.2006.18.
11. M, Darla. "How TF-IDF Works." *Medium*, Towards Data Science, 17 Feb. 2021, https://towardsdatascience.com/how-tf-idf-works-3dbf35e568f0.
12. "Cosine Similarity." *Cosine Similarity - an Overview* | ScienceDirect Topics, https://www.sciencedirect.com/topics/computer-science/cosine-similarity.
13. "Euclidean Norm." *Euclidean Norm - an Overview* | ScienceDirect Topics, https://www.sciencedirect.com/topics.engineering/euclidean-norm.
14. Miryeganeh, Nima, et al. "GloBug: Using Global Data in Fault Localization." *Journal of Systems and Software*, vol.177, 14 Jan. 2021, p. 110961., https://doi.org/10.1016/j.jss.2021.110961.
15. "3.7.1 Standard Predefined Macros." *Standard Predefined Macros (the C Preprocessor)*, GNU Compiler Collections, https://gcc.gnu.org/onlinedocs/cpp/ Standard-Predefined-Macros.html.
16. "System." Cplusplus.com, https://www.cplusplus.com/reference/cstdlib/system/.
17. OleanderSoftware (2020). OleanderStemmingLibrary (Version 2019) [Computer software]. https://github.com/OleanderSoftware/OleanderStemmingLibrary
18. Inc., AtCoder. "A - Wanna Go Back Home." *AtCoder*, 21 Aug. 2016, https://atcoder.jp/contests/agc003/tasks/agc003_a.
19. "A." Dropbox, https://www.dropbox.com/sh/nx3tnilzqz7d

f8a/AADDSrewzVyidpzV8jmnaf_pa/AGC003/ A?dl=0&subfolder_nav_tracking=1.

Appendix

The source code of the automated debugging system developed in this research is publicized on Github. Link: https://github.com/Sammyhao/Automated-Debugging-System.

Authors

Sam is a senior at Wego High School who loves programming and is always eager to explore the newest findings in computer and information systems. Besides doing research, he is currently developing web applications and leading the informatics club at school. He intends to extend her interest in coding by majoring in computer science in college.

DOI: 10.36838/v5i4.17