2022 年臺灣國際科學展覽會
優勝作品專輯

作品編號　**190013**

參展科別　電腦科學與資訊工程

作品名稱　**Automated Debugging System – Implementing Program Spectrum Analysis and Information Retrieval on Fault Localization**

得獎獎項　大會獎 四等獎

就讀學校　臺北市私立薇閣高級中學

指導教師　謝孫源、廖翊廷

作者姓名　郝胤翔

關鍵詞　**Automated Debugging、Program Spectrum Analysis、Information Retrieval**

作者簡介

我是郝胤翔，一位熱愛電腦科學的高中生。

平時，我很喜歡研讀資訊工程和程式設計領域的新知，也非常榮幸能夠參加此次國際科展，讓我有機會把自己的想法實作並展示出來，也感謝我的指導老師時常對我研究上的提點與指教，更要感謝爸媽的全力支持與鼓勵，希望未來能夠繼續朝引領科技創新的道路前行。

Abstract

在程式專案開發中，偵測錯誤常為最耗時的環節，進而影響整個專案的開發時長。而現今雖有些許輔助開發者提升偵錯效能的工具，但這些工具也只能藉由提供編譯執行中的資訊讓開發者省去偵錯時的繁瑣步驟，仍須開發者自行評估每段程式碼的正確性。此研究透過程式段落分析與資訊檢索實現自動錯誤定位，在每個程式段落標記其成為臭蟲（bug）的可能性。在程式段落分析中，執行使用者之原始碼，並透過歸納最終結果為正確及錯誤之執行路徑差異分析出每個程式段落的可疑性。接著運用資訊檢索技術於資料庫中找尋相似之原始碼，並參考其偵錯結果優化現有之可疑性，形成最終之可疑性排名。此研究不只結合了上述兩種技術，更優化可疑程度之計算方法以及資訊檢索中的相似度比對機制，達到更完善的錯誤定位。（此指「臭蟲」非語法錯誤（Syntax Errors），而為邏輯錯誤（Logic Errors）。）

Debugging is often the most time-consuming phase during program development, lengthening the development time and lowering efficiency. Even though 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 in order to save complicated steps in debugging; however, developers are still required to evaluate the reasonability and suspiciousness of every line of code. Unlike the current manual debuggers, this research aims to build a system that automatically detects bugs* in the programs through program spectrum analysis and information retrieval. In the section of program spectrum analysis, the system will statistically analyze the suspiciousness of every code block in the input source code file according to the provided test cases, later formulating 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 suspiciousness in program spectrum analysis and the comparison mechanisms in information retrieval, reaching performances of higher comprehensiveness and preciseness.

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

1. Introduction

1.1 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 globe. 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 turns out to increase 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 cost 50% (312 billion US Dollars) of the global software developer wages, equivalent to the wages for designing and building programs [2].

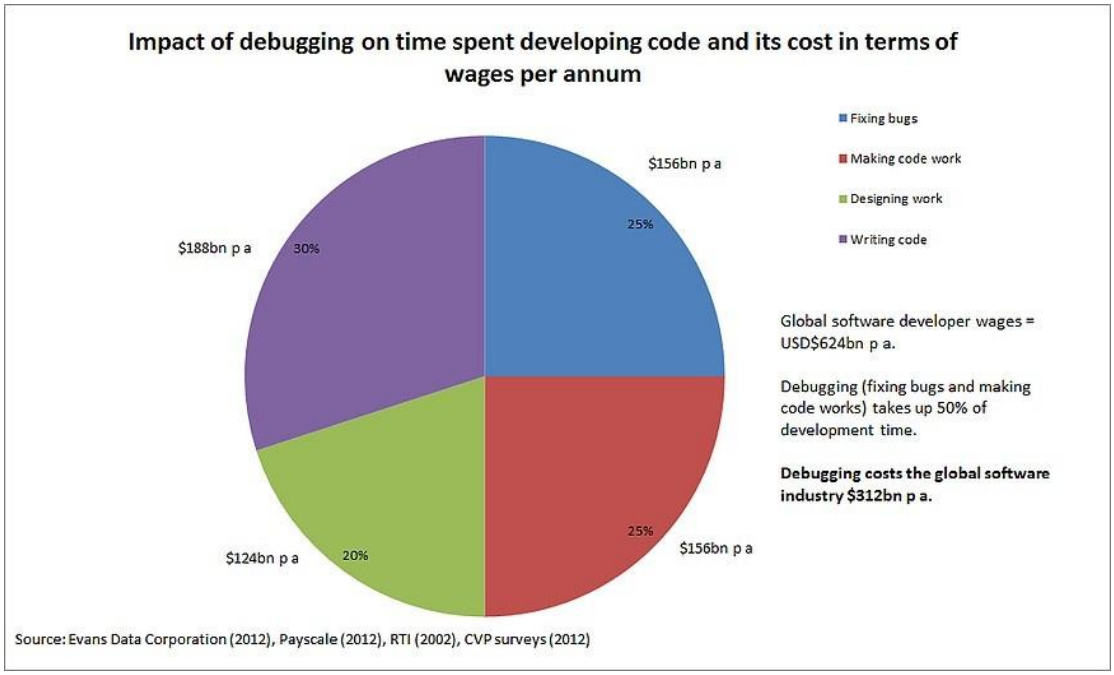


Figure 1: Impact of debugging on time spent developing code and its cost in terms of wages per annum.

Currently, multiple tools are able to 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 systems. 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].

1.2 Motivation

Although there are currently techniques that help developers to detect bugs more effectively, they still require developers to evaluate the validity of each program state and determine the final location. Fundamentally, they can't solve the problem of manual evaluations and high time consumption.

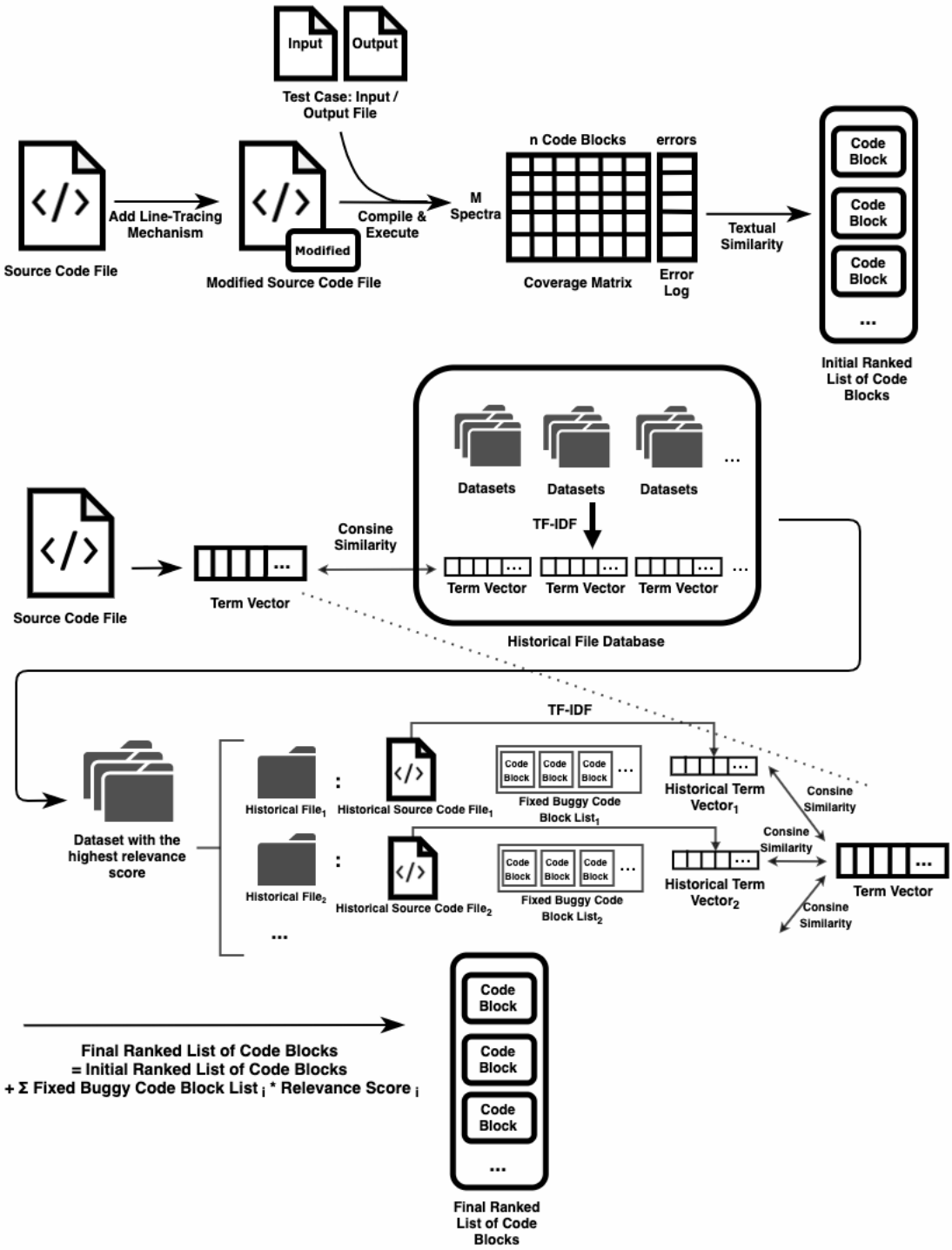
1.3 Purpose

This research aims to develop an automated debugging system that analyzes the most possible location of the bugs using test cases given by the users along with previously

fixed source code files that are comparable to current ones. The following are three primary objectives of this research:

1) Develop algorithms that automatically localize potential bugs
2) Reduce or eliminate the time that the developers spend on debugging
3) Save the effort of the developers on assessing and locating bugs

1.4 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 to contain bugs, of the code block – the higher they resemble, 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.

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 consisting 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. The fixed buggy code blocks list regarding each historical source code file then alters the initial ranked list of code blocks to the extent based on the relevance score between the historical and input source code files. Finally, a final ranked list of code blocks is generated and outputted, indicating the suspiciousness values of each code block.

2. 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.

2.1 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 records partial path 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, execution records based on hit and count, for example the program Number of n Digits in Figure 2.

```
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;
```

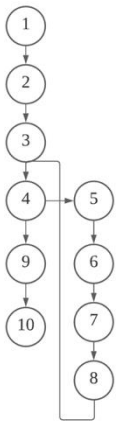


Figure 2: Example code **Number of n Digits** and its control flow graph

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

Table 1: Spectra for **Number of n Digits** of Figure 2

2.2 Spectrum-Based Fault Localization

Spectrum-based fault localization, or SBFL, evaluates the suspiciousness of every program block. This technique measures the frequency each program 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 3. 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 4. 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 3, 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 4. 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) {
	/* **BUG: It should be if (count > =1)** */
			•	•	•	1	2	3	1	0.6	1

b_4	n= (int) (count*ratio + 1);
	proc = find_nth(src_queue,n);
	if (proc) {
			•	•		0	2	3	1	0	3

b_5	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 3: An example of spectrum-based fault localization [5]

$$
\begin{array}{c} \\ M \text{ spectra} \\ \\ \end{array}
\begin{array}{c}
\overset{N \text{ blocks}}{\overbrace{\phantom{x_{11}\ x_{12}\ \dots\ x_{1N}}}} \\
\begin{bmatrix}
x_{11} & x_{12} & \dots & x_{1N} \\
x_{21} & x_{22} & \dots & x_{2N} \\
\vdots & \vdots & \ddots & \vdots \\
x_{M1} & x_{M2} & \dots & x_{MN}
\end{bmatrix} \\
\begin{array}{cccc} s_1 & s_2 & \dots & s_N \end{array}
\end{array}
\quad
\begin{array}{c}
\text{error} \\
\text{detection} \\
\begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix}
\end{array}
$$

Figure 4: 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, 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 locates [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: $s_o = \frac{a_{11}}{\sqrt{(a_{11}+a_{01})\times(a_{11}+a_{10})}}$

$$\begin{cases} a_{00} = \left|\{i|x_{ij} = 0 \wedge e_i = 0\}\right| \\ , the\ number\ of\ successful\ runs\ in\ which\ block\ j\ is\ not\ involved. \\ a_{00} = \left|\{i|x_{ij} = 0 \wedge e_i = 1\}\right| \\ , the\ number\ of\ failed\ runs\ in\ which\ block\ j\ is\ not\ involved. \\ a_{10} = \left|\{i|x_{ij} = 1 \wedge e_i = 0\}\right| \\ , the\ number\ of\ successful\ runs\ in\ which\ block\ j\ is\ involved. \\ a_{11} = \left|\{i|x_{ij} = 1 \wedge e_i = 1\}\right| \\ , the\ number\ of\ failed\ runs\ in\ which\ block\ j\ is\ involved. \end{cases}$$

2.3 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].

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

2.4 Cosine Similarity

Cosine Similarity is an approach that calculates the similarity of 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].

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

2.5 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 contains specific information about the failure of a program, to generate textual similarities with each program element, such as "for," "if," "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 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 their relevance score is calculated. Then, the system finds the program elements fixed in every history bug report, and multiply 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].

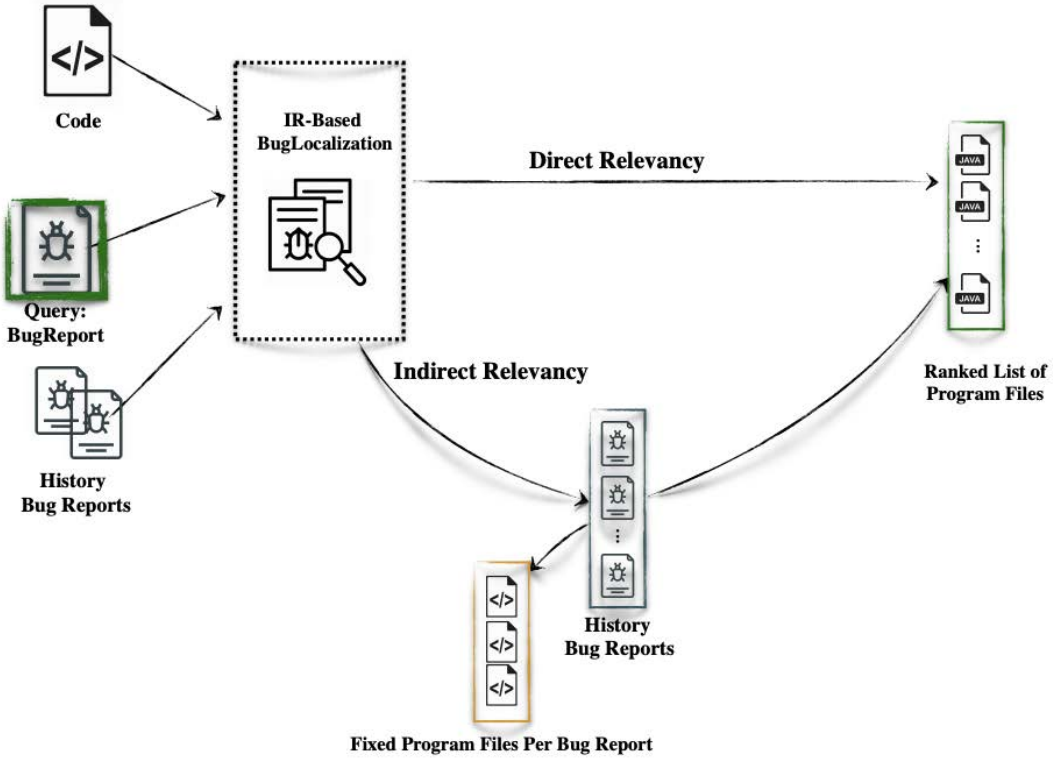


Figure 5: The process of direct and indirect relevance functions in IRFL [10]

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 5.

3. Methodology

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.

3.1 Facilities

Hardware:

1. Laptop (CPU: 2 GHz Quad-Core Intel Core i5 with Intel Iris Plus Graphics, Memory: 16 GB, SSD: 512GB)
2. Notebook

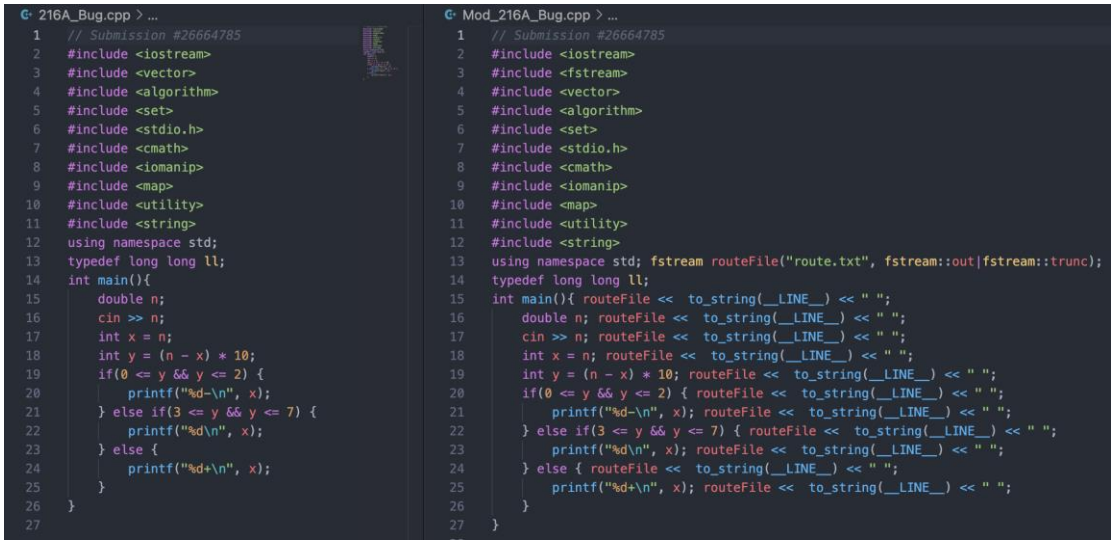
Developing environment: Visual Studio Code

Programming language: C, C++

3.2 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 6.



216A_Bug.cpp

```cpp
// Submission #26664785
#include <iostream>
#include <vector>
#include <algorithm>
#include <set>
#include <stdio.h>
#include <cmath>
#include <iomanip>
#include <map>
#include <utility>
#include <string>
using namespace std;
typedef long long ll;
int main(){
    double n;
    cin >> n;
    int x = n;
    int y = (n - x) * 10;
    if(0 <= y && y <= 2) {
        printf("%d-\n", x);
    } else if(3 <= y && y <= 7) {
        printf("%d\n", x);
    } else {
        printf("%d+\n", x);
    }
}

```

Mod_216A_Bug.cpp

```cpp
// Submission #26664785
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <set>
#include <stdio.h>
#include <cmath>
#include <iomanip>
#include <map>
#include <utility>
#include <string>
using namespace std; fstream routeFile("route.txt", fstream::out|fstream::trunc);
typedef long long ll;
int main(){ routeFile <<  to_string(__LINE__) << " ";
    double n; routeFile <<  to_string(__LINE__) << " ";
    cin >> n; routeFile <<  to_string(__LINE__) << " ";
    int x = n; routeFile <<  to_string(__LINE__) << " ";
    int y = (n - x) * 10; routeFile <<  to_string(__LINE__) << " ";
    if(0 <= y && y <= 2) { routeFile <<  to_string(__LINE__) << " ";
        printf("%d-\n", x); routeFile <<  to_string(__LINE__) << " ";
    } else if(3 <= y && y <= 7) { routeFile <<  to_string(__LINE__) << " ";
        printf("%d\n", x); routeFile <<  to_string(__LINE__) << " ";
    } else { routeFile <<  to_string(__LINE__) << " ";
        printf("%d+\n", x); routeFile <<  to_string(__LINE__) << " ";
    }
}
```

Figure 6: 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 with the expected output file provided by the users, after which the result is generated (Figure 7).

```cpp
bool LocalJudge::ExecuteCodeFile(std::string inputFileName, std::string answerFileName) {
    std::string s = "g++ Mod_" + codeFileName; // command to compile source code file
    const char* compCommand = s.c_str();
    system(compCommand); // compile
    std::fstream outputFile("output.txt", std::fstream::out); // generate "output.txt"
    outputFile.close();
    s = "./a.out <" + inputFileName + " >output.txt"; // command to execute with input and ouptut file
    const char* exeCommand = s.c_str();
    system(exeCommand);// execute with command
    routeFile.open("route.txt", std::fstream::out | std::fstream::app);
    routeFile << "\n"; // add an endline
    routeFile.close();
    return cmpFiles(answerFileName, "output.txt"); // compare answerFile & outputFile and return result
}
```

Figure 7: Compilation and Execution of source codes with input and output files

When all executions end, the system generates a set of route file R and an error log, E, which indicates whether an execution fails. Using these data, the system calculates the numbers of successful and failed execution that each line of code is involved, as well as 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:

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

$$, \begin{cases} n_{i,F,1} = \left|\{j|R_j \exists i, E_j = 1\}\right| \\ n_{i,F,0} = \left|\{j|R_j \nexists i, E_j = 1\}\right| \\ n_{i,S,1} = \left|\{j|R_j \exists i, E_j = 0\}\right| \\ n_{i,S,0} = \left|\{j|R_j \nexists i, E_j = 0\}\right| \end{cases}$$

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 consider all code blocks as code blocks that should be passed in order to get successful outcomes, since there are code blocks opening to restricted conditions, and designed not to pass in these failed test cases. Thus, $\frac{1}{u+1}$ is applied as a coefficient to rationalize the probabilities.

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

3.3 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.

During the process, the historical file database provides information that contributes to the optimizations.

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, TF-IDF vectorization function is used to help evaluate the relevance among documents and suspiciousness sequences. The input text are 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 records respectively the token's term frequency (TF) and inverse term frequency (IDF). 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 8) to measure the document frequency. The corpus is read and outputted as token list $allVocabList[i]$ referencing document i in the corpus. During calculation of every term's inverse document frequency, the system iterates over $allVocabList[i]$ and identify whether the document contains the term through $binary\_search$.

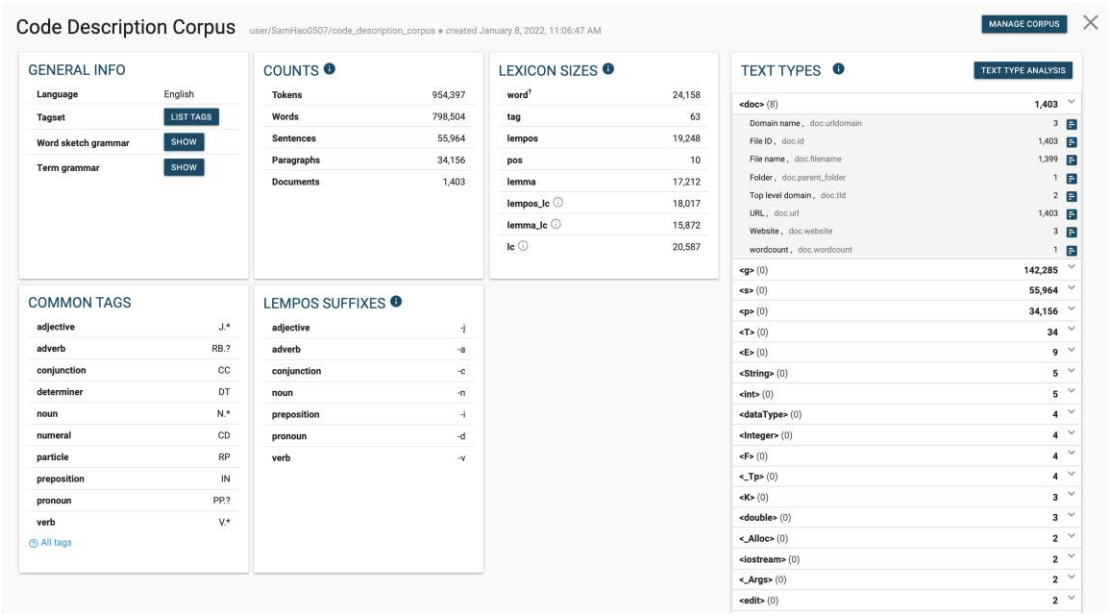


Figure 8: Code Description Corpus

Firstly, the system vectorizes the general description of the current source code provided by the users into TF-IDF vector, calculate the relevance score between the vector and all folders' description vectors using cosine similarity formula, and find 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 current source code file and historical debugging analysis to determine to what extent 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 suspiciousness relevance, $cbMatchSus[i]$ of the ith 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 $codeBlocks[i]$, 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 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 $bugRelevanceScore$,$codeBlockBugRelevance\text{-}Score$ indicates the consistency of bug conditions between the pair of code blocks. It is calculated through restricted growth formula:

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

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%, $-k = \frac{\ln 0.5}{50\%} \approx -1.38629436112$. 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 finally 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:

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

Where $file_{j,i}$ is the file that has 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.

4. Result 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.

4.1 Data and Preprocessing

In order to exhibit the validity of 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 9). Each pair of buggy and correct source code files are written by the same user in AtCoder. The buggy source codes are be labeled "WA (Wrong Answer)" in the online judge, and the correct one with "AC (Accepted)" (Figure 9).

2021-11-02 19:26:53	A - Wanna go back home	luogu_bot1	C++ (GCC 9.2.1)	200	352 Byte	AC	6 ms	3628 KB	Detail
2021-11-02 19:24:47	A - Wanna go back home	luogu_bot1	C++ (GCC 9.2.1)	0	320 Byte	WA	9 ms	3532 KB	Detail

Figure 9: One of the pairs of buggy and correct source code files for *Wanna Go Back Home*

216A > 216A_Bug.cpp > ...

```cpp
// Submission #26664785
#include <iostream>
#include <vector>
#include <algorithm>
#include <set>
#include <stdio.h>
#include <cmath>
#include <iomanip>
#include <map>
#include <utility>
#include <string>
using namespace std;
typedef long long ll;
int main(){
    double n;
    cin >> n;
    int x = n;
    int y = (n - x) * 10;
    if(0 <= y && y <= 2) {
        printf("%d-\n", x);
    } else if(3 <= y && y <= 7) {
        printf("%d\n", x);
    } else {
        printf("%d+\n", x);
    }
}
```

216A > 216A.cpp > ...

```cpp
// Submission #26664789
#include <iostream>
#include <vector>
#include <algorithm>
#include <set>
#include <stdio.h>
#include <cmath>
#include <iomanip>
#include <map>
#include <utility>
#include <string>
using namespace std;
typedef long long ll;
int main(){
    double n;
    cin >> n;
    int x = n;
    int y = (n - x) * 10;
    if(0 <= y && y <= 2) {
        printf("%d-\n", x);
    } else if(3 <= y && y <= 6) {
        printf("%d\n", x);
    } else {
        printf("%d+\n", x);
    }
}
```

Figure 10: 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

For every pair of buggy and fixed source code files, source code and code block descriptions are generated. Source code descriptions are the paraphrased form of the problem statement and unique from other source code descriptions, while code block descriptions are the translation of code blocks into 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 there are the additional line is 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 end of the statements to make the content wrapped in curly braces.

4.2 Result

Among the 15 pairs of buggy and correct source code files, 29 code blocks turned out to be fixed, each receiving a final suspiciousness value and rank, as shown in Figure 11 and Table 2. Through the distribution and mean of the suspiciousness value and rank, it can be concluded that the system has identified and highlighted most of the fixed buggy code blocks, but the suspicious value is mostly concentrated around 40% to 60%, which is not obvious enough to indicate the high suspicion of the fixed code blocks.

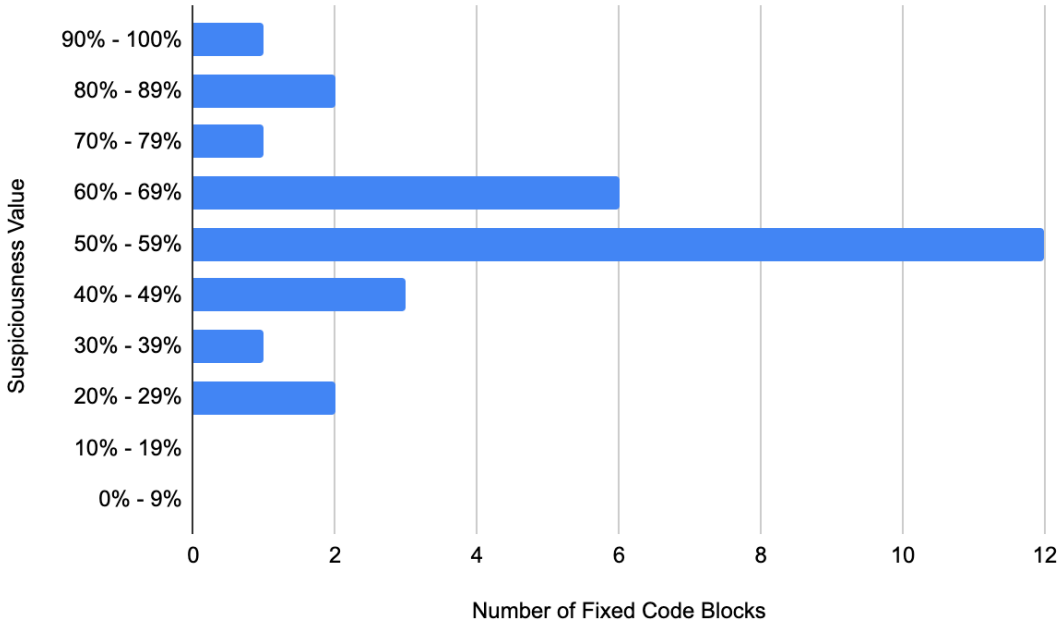


Figure 11: Statistics of the suspiciousness value of fixed code blocks

Average suspiciousness probability of the fixed code blocks	Average ranking of the fixed code blocks	Absolute Accuracy (Fixed code blocks ranked No. 1)
$58.58\% \pm 15.29\%$	No. 3.19 ± 2.11 / 10.93	46.66%

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

4.3 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 :

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

when there are only failed cases, $n_{i,S,0}$ and $n_{i,S,1}$ both equals 0, leading the suspiciousness value to be definitely 1, making the result inaccurate.

2. High Time Complexity

   The time complexity of this system is $O(n_f \times {n_c}^2 \times n_{term} \times n_{doc} \times \log n_{token})$ 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 document 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, code blocks of the current source code file are matched with irrelevant historical code blocks, making the result inaccurate.

4. Noise Problem

   Due to the fact that the system uses relevance scores as weight for updating weights, files and code blocks with little relevance still affect the final suspiciousness ranking, making it less accurate.

Solution:

To solve the problem occurred under all-WA conditions, I change the design of the program spectrum formula to make the suspiciousness value be 0.5 under the condition that $\frac{n_{i,S,1}}{n_{i,S,0}+n_{i,S,1}}$ equals 0 and $\frac{n_{i,F,1}}{n_{i,F,0}+n_{i,F,1}}$ doesn't. The database then optimizes the suspiciousness sequence according to the retrieval of the historical debugging analysis. An example is shown in Figure 12, which 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 spaces for the information retrieval section to update the values according to relevant historical files.

	Suspiciousness Value (Before)	Suspiciousness Ranking (After)	Fixed Code Blocks
Line 19 ~ 22	100.00%	63.03%	✓
Line 6 ~ 16	100.00 %	58.97%	
Line 24 ~ 24	100.00%	51.55%	

Figure 12: Comparison of the suspiciousness value before and after the optimization is implemented

Solution Proposal:

1. After calculations of the IDF value of terms, the system can save the results onto the database, and directly use them when there are identical input terms afterwards. This approach is especially helpful for common terms.

2. To solve the inaccuracy of code blocks 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. Broaden the number of historical files and folders to provide ample and more likely useful references. Once the system can retrieve sufficient relevant files, the irrelevant ones will have little impact on the formation of the final suspiciousness rankings.

5. Conclusion

5.1 Summary of Findings

By implementing program spectrum analysis and information retrieval in the system, it is demonstrated that the result is stable and moderately accurate, but they are still ambiguous. To improve the system's accuracy and efficiency on localizing bugs, calculations should be saved and used when next similar request is made. Also, code block matching algorithm can be changed to control flow graph comparison to keep the matchings consistent with the relevance of code blocks' content, while database should be constantly expanding to handle wider range of source code files.

5.2 Future Prospects and Applications

Currently, this system only support single-file code project written in c++. In the future, I expected to expand the programming language support in the future and let more users to access 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, the system can expand the database and implement higher quality of fault localization on the input source codes. By saving the developers' development time lengths, this system can ultimately increase the productions of software programs and even fasten technological growth.

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 Software 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-stvr209>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/nx3tnilzqz7df8a/AADDSrewzVyidpzV8jmnaf_pa/AGC003/A?dl=0&subfolder_nav_tracking=1.

【評語】190013

這個研究能減少 program debugging 所需的時間因此相當有價值，但程式的 bugs 有很多種類，此作品需要清楚說明其適用的 bug 種類。另外，此作品宣稱能找出會造成 logic error 的 bugs，但一個程式所想達到的 logic 此系統並不會知道，因此此系統宣稱能偵測到 logic bugs 這個能力有待驗證。另外，此作品應提供一些具體的例子來支持其宣稱能找出 logic bugs 的能力。