A Visual Debugging Aid Based on Discriminative Graph Mining

Jennifer L. Leopold[1], Nathan W. Eloe[2], Jeff Gould[1], and Eric Willard[1]

[1]Missouri University of Science & Technology
Department of Computer Science
Rolla, MO, USA

[2]Northwest Missouri State University
School of Computer Science and
Information Systems
Maryville, MO, USA

leopoldj@mst.edu, nathane@nwmissouri.edu, jg7f9@mst.edu, emwwwc@mst.edu

***Abstract*—*Why doesn't my code work?* Instructors for introductory programming courses frequently are asked that question. Often students understand the problem they are trying to solve well enough to specify a variety of input and output scenarios. However, they lack the ability to identify where the bug is occurring in their code. Mastering the use of a full-feature debugger can be difficult at this stage in their computer science education. But simply providing a hint as to where the problem lies may be sufficient to guide the student to add print statements or do a hand-trace focusing on a certain section of the code. Herein we present a software tool which, given a C++ program, some sample inputs, and respective expected outputs, uses discriminative graph mining to identify which lines in the program are most likely the source of a bug. Additionally, the particular operators (relational, logical, and arithmetic) that are used in the code may be considered in recommending where the bug may be. The tool includes a visual display of the control flow graph for each test case, allowing the user to step through the statements executed.**

Keywords-debugging; graph; data mining; visualization

I. INTRODUCTION

As discussed in [1], instructors and teaching assistants for introductory programming courses frequently are asked by their students: why doesn't my program work? Often the students understand the problem they are trying to solve well enough to articulate a variety of input and output scenarios. For example, if they are trying to find the sum of all even values in a list of numbers, they know that the input list {1, 2, 3, 4, 5} should produce a result of 6, and the input list {1, 3, 5, 7} should produce a result of 0. However, they frequently lack the ability to identify, or even narrow down, where a bug is occurring in their code when it does not produce the correct results. The recommendation to add print statements, although easy for experienced programmers, can require some skill and practice to master, and the use of a full-feature debugger can be cumbersome and intimidating to a novice programmer.

Herein we present *BugHint*, a software tool which, given a C++ program, some sample inputs, and respective outputs, uses discriminative graph mining to identify which lines in the C++ program are most likely causing the erroneous results. Additionally, the particular relational, logical, and arithmetic operators that are used in the program may be considered since beginning programmers tend to make more semantic errors with certain operators and in expressions that utilize multiple operators. The tool includes a visual display of the control flow graph for each test case (i.e., sample input), allowing the user to step through the statements as they are executed. The goal is that the student will take the bug hint and subsequently scrutinize the logic and code in the identified section of the program, thereby finishing the debugging process on his/her own.

The organization of this paper is as follows. Section II provides a brief overview of related work in debugging experiences with beginning programmers and the use of visualization in debugging. Section III discusses the foundation for and implementation of our software tool in terms of the graph mining analysis algorithms. Section IV presents the graphic user interface. A summary and conclusions are given in Section V. Future work is discussed in Section VI.

II. RELATED WORK

A. Debugging Experiences with Beginning Programmers

Several studies (e.g., [1], [2], and [3]) have identified problems that students experience with coding in introductory computer science courses, resulting in a proliferation of program bugs. Debugging strategies such as strategically placed print statements can be difficult to teach [1]. There are full-feature debugging tools such as GDB, which allow one to set breakpoints in the code and/or watch the values of variables change during execution of the program. However, for some novice programmers these tools can be too cumbersome and/or intimidating to use. After years of study, there is no consensus as to whether beginning programmers should be exposed to a full-feature debugger.

There have been studies that have successfully integrated the teaching of programming and a debugger at the introductory level. In [2] the authors used a debugger to demonstrate construction of Java objects and function calls in addition to using the debugger to find bugs in programs. Similarly, the authors of [4] used debugging exercises and simple debugger

DOI Reference Number: 10.18293/VLSS2018-029

functions to reinforce programming concepts (e.g., loops) that they were teaching.

However, full-feature debugger tools are not without criticism. In addition to the complaint that they may further confound the debugging experience for novice programmers who are already dealing with learning about an editor, operating system commands, compiler error messages, and programming language syntax, there is the issue that debuggers can potentially introduce additional bugs. A heisenbug is a software bug that is introduced when one attempts to study or analyze a program. Running a program in a debugger can actually modify the original code, changing memory addresses of variables and the timing of the execution. Debuggers often provide watches or other user interfaces that cause additional code to be executed, which, in turn, modify the state of the program. Time also can be a factor in heisenbugs, because race conditions may not occur when the program is slowed down by single-stepping through lines of code with the debugger.

Many visual debugging tools (such as DDD [5], Nemiver [6], or those debugging tools built into IDEs) provide a more user-friendly interface to command line debugging tools. Command line debugging tools suffer from the limitations of the interface; viewing where execution has stopped or paused requires programmer intervention, determining where breakpoints are to be placed (or have been placed) can be difficult (often requiring the programmer to figure out the exact line number at which they want to break), and determining the path that the execution followed can be difficult if breakpoints are not set appropriately. Additionally, if the programmer wants to figure out how their code behaves with multiple inputs, they will need to change the code, recompile, and run the debugging tool again. For veteran programmers this task is routine (and often more tedious than difficult); for novice programmers the complexity and power of these tools can be daunting and difficult to grasp.

Herein we do not seek to answer the question of whether the use of a full-feature debugger should be integrated into an introductory programming course. Rather, it is our intention to present a simple tool which the student can use as a debugging aid and training tool. Our aim is similar to the function of the instructor or teaching assistant who provides a hint as to where in the student's code the bug might be occurring. It is still up to the student to add print statements, do a hand-trace focusing on those particular statements, or use other techniques to try to fix the problem on his/her own, considering various input-output test cases.

B. Visualization in Debugging

Many contemporary debugging tools provide some type of visual representation of the source code in addition to displaying the program as text. This visual representation could be in the form of a flow chart (e.g., Visustin [7]), a control flow graph (e.g. KDevelop [8] and Dr. Garbage [9]), or UML diagrams (e.g., Eclipse ObjectAid [10]). The objective of the visualization is to facilitate understanding of some properties of the program such as the logic and/or the interactions between code blocks. To this end, animation (not just a static representation) of program execution has long been found to be useful.

Just as UML diagrams were deemed to be particularly helpful for object-oriented programming languages like Java and C++, control flow graphs have been found to be useful in debuggers for various programming paradigms. The authors of [11] presented GRASP, a graphical environment for analyzing Prolog (i.e., logic) programs; the tool dynamically animates the executed sequence of Prolog subgoals as a control flow graph and allows the user to inspect instantiation of variables as s/he steps through the execution. In [12] the authors introduced a debugging tool for MPI (i.e., parallel) programs that displays a message-passing graph of the execution of an MPI application; parts of the graph are hidden or highlighted based on the sequence of MPI calls that occur during a particular execution. Mochi [13] was created as a visual debugging tool for Hadoop (i.e., distributed programs); it displays the control flow of the workloads of each processor as a graph, allowing the user to observe the map and shuffle processing that takes place, and possibly identify erroneous sequencing and/or data partitioning.

III. IMPLEMENTATION

A. Discriminative Graph Mining

Our tool, *BugHint*, was motivated by the work presented in [14] for identifying bug signatures using discriminative graph mining. The basic idea is to first produce a control flow graph for a program written in a procedural programming language (in our case, this is C++). In brief, a control flow graph is a directed graph made up of nodes representing basic blocks. Each basic block contains one or more statements from the program. There is an edge from basic block Bi to basic block Bj if program execution can flow from Bi to Bj. For more information on control flow graphs and determination of basic blocks, see [15]. For C and C++ programs, a control flow diagram can be generated by compiling the program with *clang* and *opt* (we specify no optimization), and then creating the graph as a dot graph description language file using *dot*.

As an example, consider the C++ program shown in Fig. 1 which is supposed to replace only the first occurrence of either *x* or *y* in an array *a* with the value of *z*. This program does not perform that task correctly; it contains a bug. For simplicity, the code to output the final values of the array is commented out in this program since it is not where the bug occurs.

An example of a control flow graph for this program is shown in Fig. 2. In this graph there are eight blocks; the figure shows which lines of code are contained in each block.

After constructing a control flow graph for the program to be analyzed, our tool needs to consider test cases. These need to be specified in terms of sample input and expected output. The test cases should be as representative as possible of all boundary conditions for the program. However, a novice programmer may be unfamiliar with that notion. At the very least, the user must specify at least one input sample that is known to produce correct output and at least one input sample that is known to produce incorrect output; the user must distinguish these as 'correct' and 'incorrect.' In Table 1 we list some sample test cases for the example program shown in Fig. 1.

```
int main( )                          // line 1
{
  // inputs to the program
  int x = 1;
  int y = 7;
  int z = 0;
  int a[2] = {1, 2};
  int arraySize = 2;

  for (int i = 0; i < arraySize; i++)  // line 2
  {
    if (a[i] == x)                   // line 3
    {
      a[i] = z;                      // line 4
    }                                // line 5
    if (a[i] == y)                   // line 6
    {
      a[i] = z;                      // line 7
    }                                // line 8
  }                                  // line 9
  // code to output a[ ] …
  return(0);                         // line 10
}
```

Figure 1. Example C++ program

For each sample case, our tool produces a code trace in terms of the lines executed for the specified input. The code traces for the four sample cases shown in Table 1 are listed in Table 2. It should be noted that if there is an infinite loop (which is a common bug) during execution of one of the sample input cases, the output from the code trace should be sufficient to identify the line(s) where the bug is occurring and no further analysis should be necessary. From each code trace, we also generate a control flow graph for that sequence. The control flow graphs for code traces 1 and 2 from Table 2 are shown in Fig. 3; the control flow graphs for code traces 3 and 4 are the same as the graph shown in Fig. 2.

The collection of graphs for the sample cases is next analyzed to identify non-discriminative edges. A non-discriminative edge is an edge that appears in every graph that is in the collection of execution graphs. Such edges are removed from each graph in the collection since they are the same in each execution, and, as such, are not informative in distinguishing where the bug occurs. The collection of control flow graphs with non-discriminative edges removed for our running example is shown in Fig. 4. Finally, the collection of graphs is analyzed to determine what subgraph is common to the faulty (i.e., incorrect output) execution graphs, but not common to the correct execution graphs. This corresponds to the section of code where the bug likely occurs. For our running example, such a discriminative control flow graph is shown in Fig. 5. It tells us that the bug involves blocks B4, B6, and B7, which correspond to lines 4-8 in the program. The hope is that the student will use this information to realize that, after changing the value to *z* in line 4, the program should not proceed to lines 6-8 since the specifications of the problem were to change either, not both, the occurrence of *x* or *y* to *z*.

The discriminative graph, and hence the bug in the program, may not consist of lines that are executed in the incorrect cases, but not executed in the correct cases (as was the situation in this example program); it could be the reverse situation. Or it could be the case that we cannot find a subgraph that is common to *all* faulty (or correct) execution graphs, but not common to the correct (or faulty) execution graphs. The algorithms we utilize for identifying the "best" discriminative graph are explained next. These differ slightly from those proposed in [14] and [16] for discriminative graph mining.

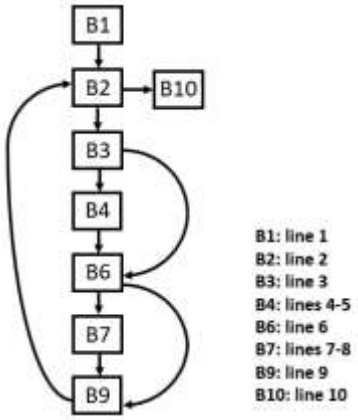

Figure 2. Control flow graph for the example program

TABLE 1. SAMPLE TEST CASES FOR THE EXAMPLE PROGRAM (INCORRECT CASES HIGHLIGHTED

Sample No.	a	x	y	z	Result
1	{1, 2}	1	7	0	{0, 2}
2	{1, 2}	7	1	0	{0, 2}
3	{1, 7}	1	7	0	{0, 0}
4	{1, 7}	7	1	0	{0, 0}

TABLE 2. CODE TRACES FOR THE EXAMPLE PROGRAM

	Trace Line Numbers	Trace Block Numbers
1	1 2 3 4 5 6 9 2 3 6 9 2 10	B1 B2 B3 B4 B6 B9 B2 B3 B6 B9 B2 B10
2	1 2 3 6 7 8 9 2 3 6 9 2 10	B1 B2 B3 B6 B7 B9 B2 B3 B6 B9 B2 B10
3	1 2 3 4 5 6 9 2 3 6 7 8 9 2 10	B1 B2 B3 B4 B6 B9 B2 B3 B6 B7 B9 B2 B10
4	1 2 3 6 7 8 9 2 3 4 5 6 9 2 10	B1 B2 B3 B6 B7 B9 B2 B3 B4 B6 B9 B2 B10

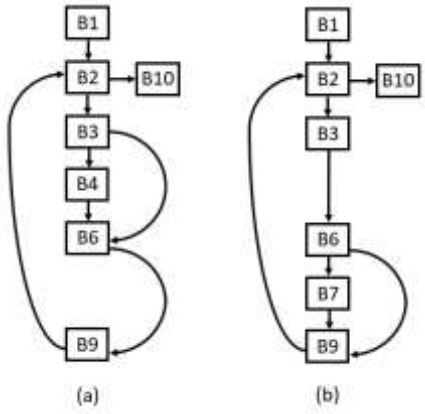

Figure 3. Control flow graphs for (a) trace 1 and (b) trace 2 from Table 2

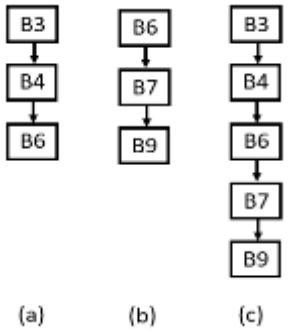

Figure 4. Control flow graphs with non-discriminative edges removed for (a) trace 1, (b) trace 2, and (c) traces 3 and 4 from Table 2

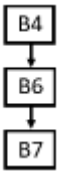

Figure 5. Discriminative control flow graph for the example program

Let C^+ and C^- represent the sets of control flow graphs for the sample test cases producing correct and incorrect results, respectively; we require that there be at least one graph in each such set. The function *FindDiscriminativeGraph* (Fig. 6) first removes non-discriminative edges from the graphs in both sets. It then calls *CreateDiscriminativeGraph* (Fig. 7) to try to find a subgraph that is common to all faulty execution graphs, but not common to all the correct execution graphs. If we are unable to find such a graph, then the function *RelaxedCreate-DiscriminativeGraph* (Fig. 8) is called, which relaxes the requirement that the subgraph we seek not be present in all of the correct execution graphs; instead the subgraph only has to not be present in $\alpha * |C^+|$ of the correct execution graphs, where α is a user-specified parameter (our default is $\alpha = 0.5$).

FindDiscriminativeGraph and *CreateDiscriminativeGraph* use a function called *Augment*; this function takes the subgraph G and adds to it an edge (and possibly a node) such that the source vertex exists in G, and the edge (and destination node) exists in all graphs in S1. In this way, a subgraph with an additional edge that exists in all elements of S1 is created and considered by the algorithm.

If we still fail to find a discriminative subgraph, then the bug likely does not involve code that is executed in all faulty cases and not in correct cases, but rather involves code that is executed in correct cases and not in faulty cases. Thus, we again call *CreateDiscriminativeGraph*, but reverse the order of the parameters (C^+ and C^-) from our previous call. If we still fail to find a discriminative subgraph, we again call *Relaxed-CreateDiscriminativeGraph* and look for a subgraph that only has to not be present in $\beta * |C^+|$ of the correct execution graphs, where β is a user-specified parameter (our default is $\beta = 0.5$).

It is possible that the resulting discriminative graph will be disconnected. We output the smallest connected component in that graph using the assumption that a novice programmer will want to focus on a single, sequential section of his/her program for scrutinizing the bug, rather that examining multiple, "fragmented" sections of code.

It should be noted that it is possible that our algorithm will not find any graph that meets the discriminative conditions. This could be because the specified test cases do not adequately exercise all paths through the control flow graph or it could be that the path through the control flow graph will be the same regardless of the input. Additionally, it could be the case that multiple subgraphs could be viable candidates to be the source of the bug. These last two situations are addressed in the next section.

Algorithm: *FindDiscriminativeGraph(C^+, C^-, α, β)*
C^+: set of control flow graphs for inputs producing correct output
C^-: set of control flow graphs for inputs producing incorrect output
α: percentage of graphs that discriminative subgraph need not be present in C^+ when relaxing conditions
β: percentage of graphs that discriminative subgraph need not be present in C^- when relaxing conditions

```
1.  remove non-discriminative edges from graphs in C+ and C-;
2.  G = CreateDiscriminativeGraph(C-, C+);
3.  if G is empty then
4.    G = RelaxedCreateDiscriminativeGraph(C-, C+,
                                           |C+| * α);
5.    if G is empty then
6.      G = CreateDiscriminativeGraph(C+, C-);
7.      if G is empty then
8.        G = RelaxedCreateDiscriminativeGraph(C+, C-,
                                               |C-| * β);
9.      end-if;
10.   end-if;
11. end-if;
12. G' =  smallest connected component in G;
13. output G'
```

Figure 6. Algorithm for *FindDiscriminativeGraph*

Algorithm: *CreateDiscriminativeGraph(S1, S2)*
S1: set of control flow graphs
S2: set of control flow graphs

```
1.  FreqSG = queue of 1-edge subgraphs in S1;
2.  while FreqSG is not empty do
3.    G = FreqSG.dequeue();
4.    if G is not in any graph in S2 then
5.      return(G);
6.    end-if;
7.    NewGraphs = Augment(G);
8.    for each graph G' in NewGraphs do
9.      FreqSG.enqueue(G');
10.   end-for;
11. end-while;
12. return(empty graph)
```

Figure 7. Algorithm for *CreateDiscriminativeGraph*

Algorithm: *RelaxedCreateDiscriminativeGraph(S1, S2, γ)*
S1: set of control flow graphs
S2: set of control flow graphs
γ: threshold for number of graphs discriminative subgraph must be present in

```
1.  FreqSG = queue of 1-edge subgraphs in S1;
2.  while FregSG is not empty do
3.     G = FreqSG.dequeue();
4.     if G is in < γ graphs in S2 then
5.        return(G);
6.     end-if;
7.     NewGraphs = Augment(G);
8.     for each graph G' in NewGraphs do
9.        FreqSG.enqueue(G');
10.    end-for;
11. end-while;
12. return(empty graph)
```

Figure 8. Algorithm for *RelaxedCreateDiscriminativeGraph*

B. Operator Complexity

Earlier versions of *BugHint* [21] focused solely on the discriminative graphs. This approach is effective when the erroneous code causes the execution to follow a different code path in the CFG. However, it is often the case with novice programmers that, in an attempt to be clever, they introduce some relatively complex calculation in the condition of a loop or an if-statement that effectively short-circuits the branch and forces the code to always execute the same branches. In our testing of *BugHint*, we identified cases where no basic blocks were identified as problematic despite the presence of a bug. As such, the decision was made to augment the CFG analysis with more information based on additional scrutinization.

Semantic errors frequently occur in lines of code that are more complex in structure. Unless the errors encountered are in output formatting, a simple C printf statement or C++ cout statement is unlikely to introduce a semantic error into a program. Instead, bugs are often introduced in lines of code that compare or change the values of variables in the program. The more complex a line (or basic block) is in terms of relational, logical, and arithmetic operators, the more likely it is that a sematic error will occur. Additionally, some operators like && or ||, while familiar to seasoned programmers, seem foreign in their functionality to new coders. *BugHint* takes this into account by looking not only at the CFG analysis of the program (with good and bad traces) but at the complexity of each basic block encountered in the program.

Since there is no published formal study analyzing the frequency of semantic errors for various relational, logical, and arithmetic operators by beginning programmers, the complexity of various C++ operators was determined by over 40 years of collective teaching and tutoring experience of the authors for *BugHint* consideration. Several frequently used operators were sorted into ranks which were used to create a complexity score for each operator. Table 3 shows the ranks of these operators as well as the (informal) rationale for their relative ranking. A lower ranking indicates that the operator is less likely to cause a semantic error when used in an expression (on its own). For example, +, *, and – correspond to familiar arithmetic concepts for beginning programmers; students have used those operations since grade school. It should be noted that division (/) is not grouped with addition, multiplication, and, subtraction because integer division (e.g., 3 / 4 evaluates to 0 in C++) is the source of many semantic errors for novice programmers. The comparison operator == also is a frequent source of bugs since many students confuse it with the assignment operator =. In contrast to multiplication, subtraction, and addition, the logical operators && and || are new concepts to most beginning programmers and require higher order thinking than addition, multiplication, and subtraction; thus, they are ranked much higher. If not explicitly listed in the table, an operator defaults to a rank of 0.

The complexity score of an operator is defined as its rank /5.0, or more generally:

$$complexity = rank / number\_of\_ranks$$

This results in a complexity score in the range [0, 1) that increases as the complexity of an operator increases. The generalization of the calculation of the complexity is such that the ranks and relative complexities can be fine-tuned based on empirical testing and observation.

TABLE 3. OPERATOR COMPLEXITY RANKS

Rank	Ops	Rationale
0	+ * -	Most familiar
1	< > <= >= ! += -= *=	Easy concepts, but more foreign than rank 0
2	== /	== confused with =, / is integer division
3	++ -- %	New concepts, pre/post increment/decrement can give different results
4	&& ||	Requires higher order thinking

The complexity of a basic block is calculated as the sum of the number of operators in the basic block (a non-negative integer) and the average complexity of an operator in the basic block (a real value less than 1). This ensure that basic blocks with more operators are indicated as more complex, but, between basic blocks of the same number of operations, those with higher average operator complexity are classified as more complex; these are indications of two common Code Smells [19]. If the basic block has relatively few lines of code but a high number of operations, this may indicate an excessively long line of code (or a God Line); if the basic block has a large number of lines, it may hint at the presence of the Long Method code smell. While these code smells are not an indication that a bug is present, they are frequent indicators that something is wrong, or that a refactoring may result in code that is easier to debug and maintain.

BugHint augments the CFG analysis of source code with the complexity analysis. After identifying basic blocks that could be problematic using the discriminative graph mining, the basic blocks that were identified as potentially problematic are sorted by complexity. The top N (a user-specified parameter that

defaults to 2) are reported to the user as the most likely to contain the bug.

This augmentation to the CFG analysis has some benefits. Consider the following main function (where here the value of *answer* is hard-coded on the second line instead of being entered by the user to allow for *BugHint* to operate on the source):

```
// block 1
bool validEntry;
char answer = 'n';
cout << "Is this yes or no?" << endl;
validEntry = (answer == 'y' && answer == 'Y') ||
             (answer == 'n' && answer == 'N');
// block 2
if (validEntry)
   cout << "Valid entry!" << endl;
// block 3
else
   cout << "Invalid Entry" << endl;
// block 4
return 0;
```

Veteran programmers will quickly notice that the condition in line 4 is incorrect and will always return false regardless of the value of *answer*. However, CFG analysis will not indicate erroneous blocks because all traces will contain blocks 1, 3, and 4 in that order. Examination of the complexity of the basic blocks will show that blocks 2 through 4 have a complexity of 0, but block 1 has a complexity of 3 + (0.8 * 3) / 3 = 3.8. *BugHint* would augment the CFG analysis (which would be unable to find a discriminative subgraph) with the information from the complexity analysis and indicate to the user that the problem exists in block one.

IV. User Interface

Some of the tools used to generate the information needed to display to the user are not easily installable/usable on all platforms (specifically, clang/LLVM and some bash tools are not easily installable on Windows). Making the *BugHunt* platform independent was a top priority so that it could be readily available to as broad a range of novice programmers as possible. *BugHunt* has been developed as a web application written in Node.js and using vis.js for the visualization of the control flow graphs. The web application backend interfaces with various utilities written in Python, which make the appropriate system calls to clean, format, instrument, and run the source code with various starting conditions. Additionally, the backend analyzes the traces and basic blocks of the provided source code.

Web applications inherently have a large number of security concerns, and a system designed specifically to compile and run arbitrary C/C++ code exhibits an exceptionally large attack surface area. This application was not designed to run as a globally hosted web application, but rather was intended to be deployed to individuals using emerging technologies such as Docker (for platform-independent deployments of the web application that run locally) or the Linux Subsystem for Windows (which would allow a more native desktop interface to be quickly developed using Electron Shell [20]).

Initially, the user is presented with a single toolbar with a button that allows them to upload a single C++ source file. When the system receives the file, it generates the CFG using clang's analysis tools (specifically DumpCFG), calculates the complexity of each basic block, maps lines of code to basic blocks, and instruments the code by adding output statements at the beginning of each basic block (see Fig. 9). The system then runs the instrumented program and sends the resulting traces back to the UI. A student can choose individual starting configurations and walk forwards and backwards through the execution (with lines of code highlighted) using arrow buttons on the UI.

```
#include <iostream>
using namespace std;
int main()
{
    int a = 0;
    double d = 1.1;
    for (int i=0; i<10; i++)
    {
        cout << "hi" << endl;
    }
    return 0;
}
```

Figure 9a) Original C++ source

```
#include <iostream>
#include <sstream>
#include <unistd.h>
void __bbinstr(const char bbid[]) {
 std::cerr << bbid << std::endl;
 return;
}
#include <iostream>

using namespace std;
/*:B6:*/ int main() /*:/B6:*/
{
 /*:B5:*/ __bbinstr("B5");
 int a = /*%a%*/ 0 /*%~a%*/; /*:/B5:*/
 /*:B5:*/ double d = /*%d%*/ 1.1 /*%~d%*/;
/*:/B5:*/
 for (int i = 0; __bbinstr("B4"), i < 10;
__bbinstr("B2"), i++) {
   /*:B3:*/ __bbinstr("B3");
   cout << "hi" << endl; /*:/B3:*/
 }
 /*:B1:*/ __bbinstr("B1");
 return 0; /*:/B1:*/
}
```

Figure 9b) Instrumented C++ source (formatted for more readability)

The system assumes that the initial configuration that was uploaded by the student produced a good trace. However, the user can add additional starting conditions (by changing the starting values of the variables at initialization) and specify whether or not it produces a good trace (i.e., correct results) or a bad trace (i.e., incorrect results). At any point, the student may

select "Get Hint." The UI will then send the traces to the backend, which will analyze the CFG, the traces, and the complexity of the basic blocks. The resulting list of basic blocks is sent to the GUI for display.

The CFG analysis requires at least one "good" trace. While the student might not understand why their code is producing the correct answer in some cases, it must be assumed that the novice programmer has managed to get correct output in at least one case. As use of this tool requires knowledge of inputs and expected outputs (and thus what inputs result in correct output), it can be reasonably assumed that the student can identify at least one scenario where their code produces a correct result.

Fig. 10 shows a screenshot of the main view in the *BugHunt* GUI with the example program from Fig. 1 and the test cases from Table 1. The hints have been augmented using the complexity information. The arrow buttons in the GUI allow the user to step forwards and backwards through a selected execution case; both the corresponding nodes and (text) lines in the program subsequently will be highlighted. Any particular block in an execution sequence (listed below the graph display) also can be selected (i.e., clicked-on) with the mouse.

BugHint also supports user-defined functions. When loading a single cpp source file containing multiple functions, the system splits the information so that the CFG for each function and the code for each function displays in a tab. When walking through the traces, the tabbed view switches automatically to the function which is currently being executed in the trace. Fig. 11, 12, and 13 show this tabbed view for an implementation of a function that finds the maximum of two integer values; the function contains a bug (Fig. 13). When giving hints, *BugHunt* will highlight all lines of code that are suspected of being in the vicinity of the bug (i.e., are in the detected block).

V. SUMMARY AND CONCLUSIONS

Herein we have presented a simple debugging tool, which, given a C++ program that has a logic error just serious enough to occasionally produce erroneous output while sometimes producing correct output, and some sample inputs with corresponding outputs, uses discriminative graph mining to identify which lines in the program are most likely the source of the bug. Additionally, it may examine the particular relational, logical, and arithmetic operators that are used in the code to determine what lines in the code are probably causing the bug. The tool includes a visual display of the control flow graph for each test case, allowing the user to step through the statements executed.

In a previous study [21], we found that students who completed pre-training using *BugHint* did better on post-testing than students who completed pre-training without *BugHint*, even though all groups had no help for post-training. During a post-training exercise where both groups completed the exact same activity of debugging three practice programs, the treatment group found more of the bugs, self-generated more informative test cases and reasoning regarding those test cases, and self-generated more helpful comments to add to the code itself. Hence, it appears that the extra-formalized method of using *BugHint* may improve the way students think about the debugging practice.

We look forward to utilizing *BugHint* in our introductory programming courses, and performing additional usefulness and usability studies to guide further refinement of this tool, and reduce some of the time that novice students spend debugging their programs.

VI. FUTURE WORK

As with any visual programming environment, we expect that there will be a challenge in accommodating the additional visual complexity in the graphical user interface that will result from larger programs. We intend to perform usefulness and usability studies with novice programmers to find ways of implementing visual representation and navigation of a fairly large number of modules of the control flow graph in a manner that they (the students) can best understand. By fairly large number we mean a number commensurate with the size of programs that beginning programmers write. From our own teaching experience that has been approximately 50 lines of code (with no user-defined functions) at the beginning of the CS1 semester, and 1500 or more lines of code (with 20 or more user-defined functions) by the end of the semester.

We also intend to compare our system to the hints that would be produced by utilizing other existing algorithms for finding discriminative subgraphs (e.g., [17] and [18]) and/or adding other options (in addition to our current α and β parameters) to our algorithm in order to find the best discriminative subgraph, and hence provide the best suggestion for the bug hint.

Additionally, the system currently has rather strict rules on the formatting of the files it can handle due to the nature of the parser. Years of introductory computer science education experience has demonstrated that these rules are frequently not followed by students despite the insistence of course instructors. The backend will be made more robust to reduce these restrictions.

REFERENCES

[1] C. Lewis and C. Gregg, "How Do You Teach Debugging?: Resources and Strategies for Better Student Debugging", Proceedings of the 47th ACM Technical Symposium on Computing Science Education, Memphis, TN, Mar. 2-5, 2016, p. 706.

[2] R.C. Bryce, A. Cooley, A. Hansen, and N. Hayrapetyan, "A One Year Empirical Study of Student Programming Bugs", Frontiers in Education Conference, Washington, DC, Oct. 27-30, 2010, pp. 1-7.

[3] J.H.I.I. Cross, T.D. Hendrix, and L.A. Barowski, "Using the Debugger as an Integral Part of Teaching CS1", "Frontiers in Education, Boston, MA, Nov. 6-9, 2002. pp. 1-6.

[4] G.C. Lee and J.C. Wu, "Debug It: A Debugging Practicing System", Computers & Education, 32, 1999, pp. 165-179.

[5] DataDisplayDebugger, https://www.gnu.org/software/ddd/

[6] Nemiver, https://wiki.gnome.org/Apps/Nemiver

[7] Visustin, http://www.aivosto.com/visustin.html

[8] KDevelop, https://liveblue.wordpress.com/2009/07/21/3-visualize-your-code-in-kdevelop/

[9] Dr. Garbage, https://sourceforge.net/projects/drgarbagetools/files/

[10] Eclipse ObjectAid, http://www.objectaid.com/sequence-diagram

[11] H. Shinomi, “Graphical Representation and Execution Animation for Prolog Programs”, International Workshop on Industrial Applications of Machine Intelligence and Vision (MIV-89), Tokyo, Apr. 10-12, 1989, pp. 181-186.

[12] B. Schaeli, A. Al-Shabibi, and R.D. Hersch, “Visual Debugging of MPI Applications”, in Recent Advances in Parallel Virtual Machine and Message Passing Interface, A. Lastovetsky, T. Kechadi, J. Dongarra (eds)., EuroPVM/MPI, Lecture Notes in Computer Science, vol. 5205, Springer, Berlin, Heidelberg, 2008, pp. 239-247.

[13] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, “Mochi: Visual Log-Analysis Based Tools for Debugging Hadoop”, CMU-PDL-09-103, Parallel Data Laboratory, Carnegie Mellon University, Pittsburg, PA, May 2009.

[14] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, “Identifying Bug Signatures Using Discriminative Graph Mining”, ISSTA, Chicago, IL, Jul. 19-23, 2009, pp. 141-151.

[15] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman, Compilers: Principles, Techniques, and Tools, Addison Wesley, 2nd edition, 2006.

[16] X. Yan, H. Cheng, J. Han, and P.S. Yu, “Mining Significant Graph Patterns by Leap Search”, SIGMOD 2008, Jun. 9-12, 2008, Vancouver, BC, Canada, pp. 433-444.

[17] N. Jin and W. Wei, “LTS: Discriminative Subgraph Mining by Learning from Search History”, IEEE 27th International Conference on Data Engineering (ICDE), 2011, pp. 207-218.

[18] M.G.A. El-Wahab, A.E. Aboutabl, and W.M.H. El Behaidy, “Graph Mining for Software Fault Localization: An Edge Ranking Based Approach”, Journal of Communications Software and Systems, Vol. 13. No. 4, Dec. 2017, pp. 178-188.

[19] Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D., 1999. *Refactoring: improving the design of existing code*. Addison-Wesley Professional. pp. 76-78.

[20] Electron, https://electronjs.org/

[21] J.L. Leopold, N.W. Eloe, and P. Taylor, “BugHint: A Visual Debugger Based on Graph Mining”, Proceedings of the 24th International Conference on Visualization and Visual Languages, San Francisco, CA, June 29-30, 2018, pp. 109-118

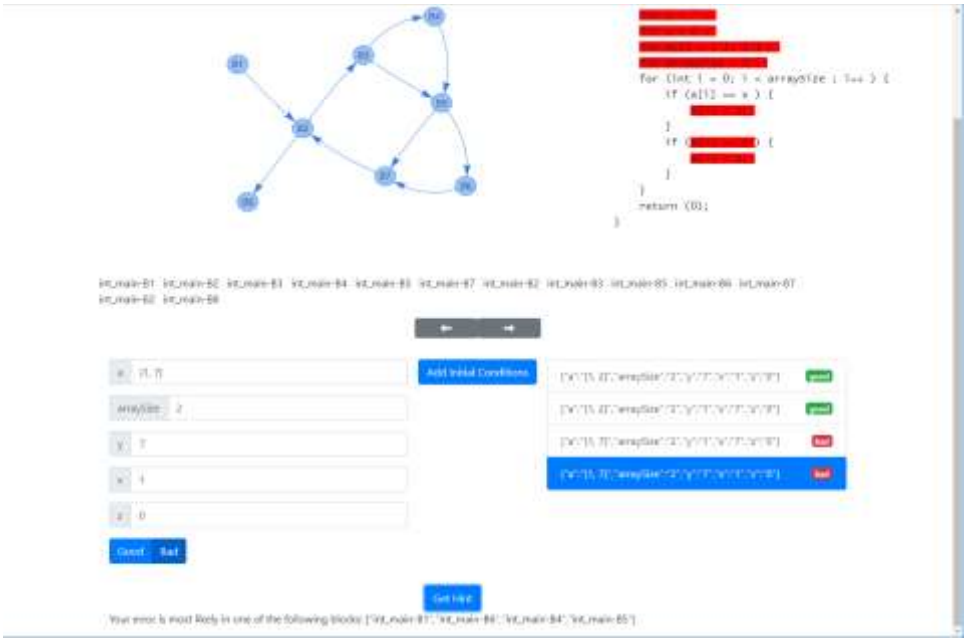

Figure 10. *BugHunt* GUI, showing the possible erroneous lines as determined by control flow graph trace analysis and complexity analysis

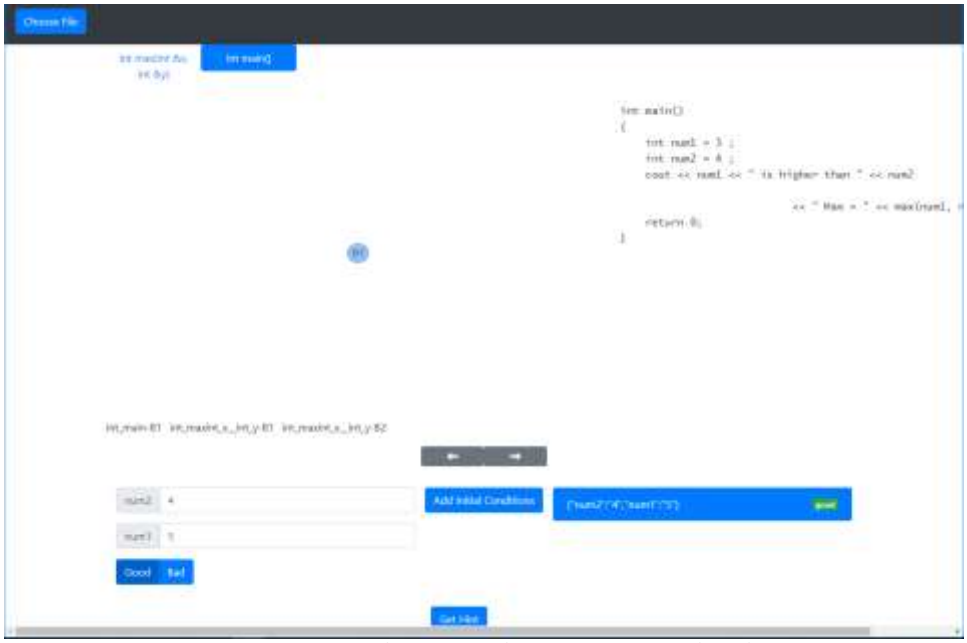

Figure 11. *BugHunt* GUI after loading a file that calls a user-defined function

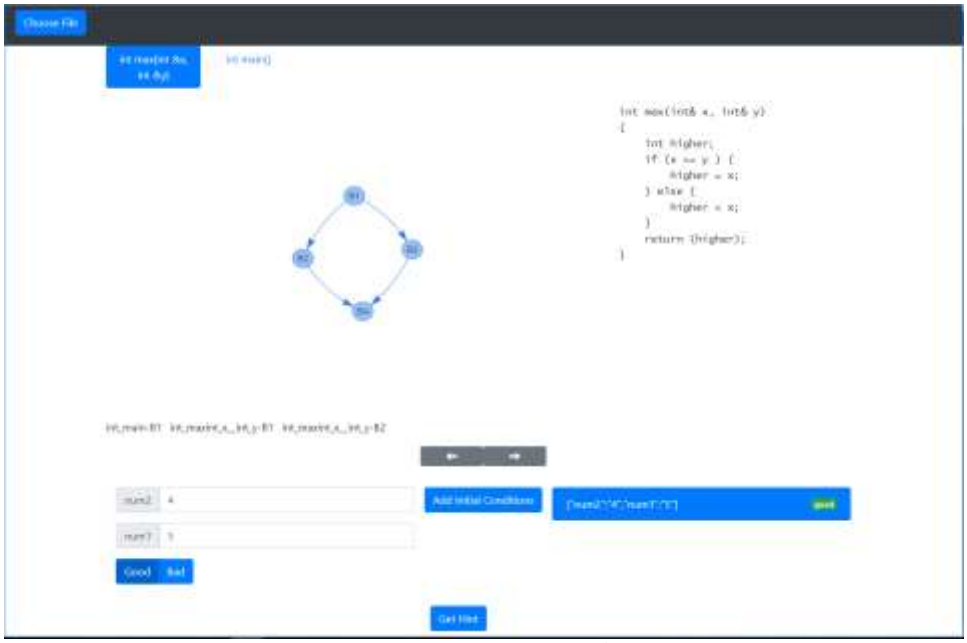

Figure 12. *BugHunt* GUI when switching function views

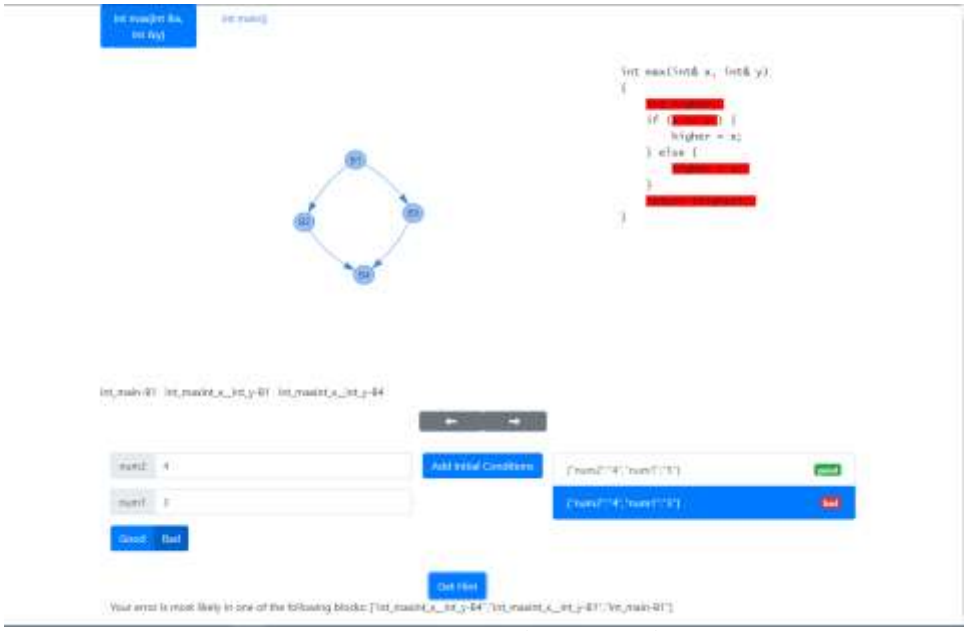

Figure 13. Hint to the location of a bug in max function