BugHint: A Visual Debugger Based on Graph Mining

Jennifer L. Leopold
Missouri University of Science and Technology
Department of Computer Science
Rolla, MO, USA
leopoldj@mst.edu

Nathan W. Eloe
Northwest Missouri State University
School of Computer Science and Information Systems
Maryville, MO, USA
nathane@nwmissouri.edu

Patrick Taylor
Missouri University of Science and Technology
Department of Computer Science
Rolla, MO, USA
taylor@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 graph mining to identify which lines in the program are most likely the source of a 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. Experimental results from a group of CS 1 students show that practice with this method: (1) makes students faster at finding bugs, (2) improves the way students test a program, and (3) improves program comments by students.*

1 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 graph mining to identify which lines in the C++ program are most likely causing the erroneous results. 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 in the identified section of the program, thereby finishing the debugging process on his/her own. Experimental results with a group of CS1 students have shown that practice with this tool not only makes students find bugs faster after training, but also improves the way students test their programs and comment their programs. 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, including the graph mining analysis and the graphic user interface. The experimental design and results are presented in Section IV. Future work is discussed in Section V. A summary and conclusions are given in Section VI.

2 RELATED WORK

2.1 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 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.

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.

2.2 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 [5]), a control flow graph (e.g. KDevelop [6] and Dr. Garbage [7]), or UML diagrams (e.g., Eclipse ObjectAid [8]). 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 [9] 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 [10] 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 [11] 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.

3 IMPLEMENTATION

3.1 Discriminative Graph Mining

Our tool, BugHint, was motivated by the work presented in [12] 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 B_j if program execution can flow from B_i to B_j. For more information on control flow graphs and determination of basic blocks, see [13]. 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.

```
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] == )                            // line 6
    {
      a[i] = z;                              // line 7
    }                                        // line 8
  }                                          // line 9
  // code to output a[ ]...
  return 0;                                 // line 10
}
```

Figure 1: Example C++ program

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.

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.

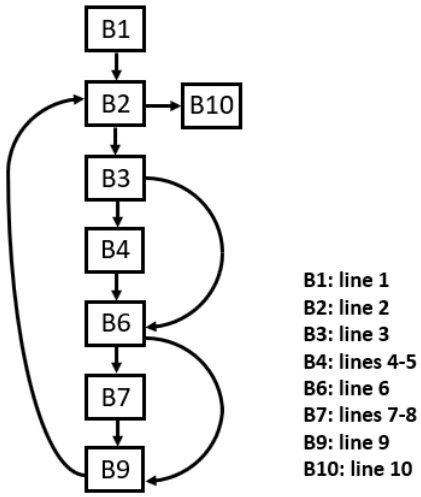

Figure 2: Control flow graph for the example program

Table 1: Sample Test Cases for the Example Program

Sample Case Num	a	x	y	z	Result	
1	{1, 2}	1	7	0	{0, 2}	correct
2	{1, 2}	7	1	0	{0, 2}	correct
3	{1, 7}	1	7	0	{0, 0}	incorrect
4	{1, 7}	7	1	0	{0, 0}	incorrect

Table 2: Code Traces for the Example Program

Sample Case Num	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

The collection of graphs for the sample cases are 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.

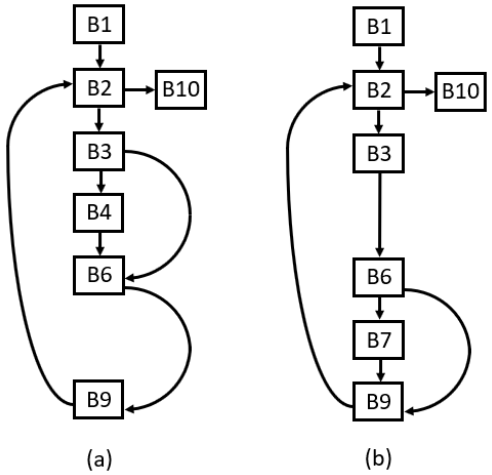

Figure 3: Control flow graphs for (a) trace 1 and (b) trace 2 from Table 2

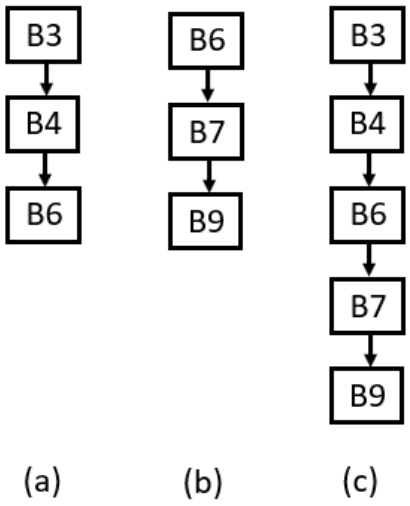

Figure 4: Control flow graphs with non-discriminative edges removed for (a) trace 1, (b) trace 2, and (c) traces 3 and 4 from Table 2

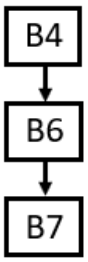

Figure 5: Discriminative control flow graph for the example program

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 [12] and [14] for discriminative graph mining. Let C+ and C- represent the sets of control flow graphs for the sample test cases producing correct and incorrect results, respectively; there must be at least one graph in each such set. The function FindDiscriminativeGraph (Alg. 1) first removes non-discriminative edges from the graphs in both sets. It then calls CreateDiscriminativeGraph (Alg. 2) 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 RelaxedCreateDiscriminativeGraph (Alg. 3) 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 RelaxedCreateDiscriminativeGraph 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.

Algorithm 1 FindDiscriminativeGraph(C+, C-, α, β)

Require: C+: set of control flow graphs for inputs producing correct output
Require: C-: set of control flow graphs for inputs producing incorrect output
Require: α: percentage of graphs that discriminative subgraph need not be present in C+ when relaxing conditions
Require: 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: $G = RelaxedCreateDiscriminativeGraph(C-, C+, |C+| * \alpha)$
5: **if** G is empty **then**
6: $G = CreateDiscriminativeGraph(C+, C-)$;
7: **if** G is empty **then**
8: $G = RelaxedCreateDiscriminativeGraph(C+, C-, |C-| * \beta)$
9: **end if**
10: **end if**
11: **end if**
12: G' = smallest connected component in G;
13: output G'

Algorithm 2 CreateDiscriminativeGraph(S1, S2)

Require: S1: set of control flow graphs
Require: S2: set of control flow graphs
1: FreqSG = queue of 1-edge subgraphs in S1
2: **while** F **do**reqSG 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)

Algorithm 3 RelaxedCreateDiscriminativeGraph(S1, S2, γ)

Require: S1: set of control flow graphs
Require: S2: set of control flow graphs
Require: γ: threshold for number of graphs discriminative subgraph must be present in

```
FreqSG = queue of 1-edge subgraphs in S1
while FregSG is not empty do
    G = FreqSG.dequeue()
    if G is in ¡ γ graphs in S2 then return (G)
    end if
    NewGraphs = Augment(G);
    for each graph G' in NewGraphs do
        FreqSG.enqueue(G')
    end for
end while
return (empty graph)
```

It should be noted that it is possible that our algorithm will not find any graph that meets the discriminative conditions; this is largely dependent upon the specified test cases. If the resulting discriminative graph is empty, the user will be told that no hint can be provided and that specification of additional test cases (that produce both correct and incorrect output) might help.

3.2 Graphic User Interface

Some of the tools used to generate the information needed for the GUI are not easily installable on all platforms (specifically, clang/LLVM). To make BugHunt available to a broad range of novice programmers, a primary concern was making the tool platform-independent. The GUI for BugHunt is a web application written in Node.JS, using vis.js for the visualization of the control flow graphs. The web application is integrated with various utilities written in Python, and makes the necessary system calls to add code to display basic block information, compile, and run the various traces, as well as to analyze the correct and incorrect execution traces to provide the bug hint. Hence the tool can be run from any type of computer but can be deployed using emerging technologies such as Docker or the Linux Subsystem for Windows.

Fig. 6 shows a screenshot of the BugHunt GUI with the example program from Fig. 1 and the test cases from Table 1. 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.

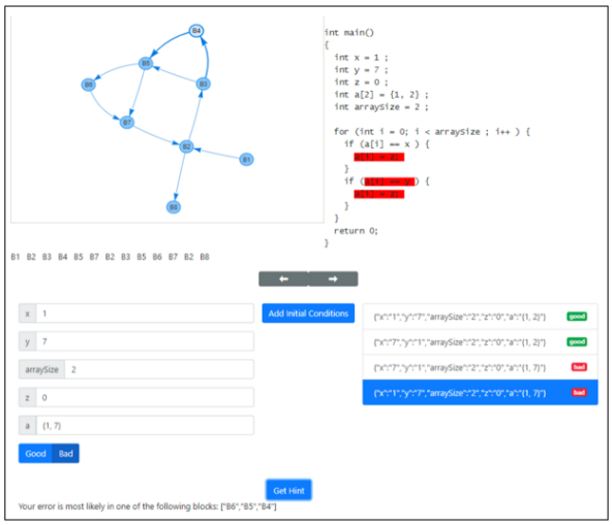

Figure 6: Control flow graphs for (a) trace 1 and (b) trace 2 from Table 2

4 EXPERIMENTS

To test the hypothesis that experience with BugHint improves debugging skill, we tested training with the software using students who had just recently completed CS1. Notably, we did not merely test whether BugHint made it faster to find the bug in a program where a hint was given, but instead whether experience with BugHint would actually improve debugging skill such that students would be better at debugging new programs without BugHint.

4.1 Experimental Design

We employed a between-subjects design, as this is usually less confounded by the design itself, and stronger, when n is sufficiently large to accommodate two groups. Two groups of subjects experienced different pre-training, with identical post-training testing. The human subjects were tested in four sections of a CS2 laboratory class during the second week of the semester (immediately following their recent completion of CS1); two sections were the treatment condition, and two were the control. A total of 163 students participated in this study. All experiments were performed in accord with human subjects and institutional review board guidelines. The two groups of subjects are categorized as follows (Fig. 7):

1. Treatment pre-training - This group was introduced to BugHint and instructed how to use it. They completed a pre-training exercise using BugHint on three small programs that contained bugs.

2. Control pre-training - This group was given instructions and tips on manual debugging. They completed a pre-training exercise using standard manual debugging methods (e.g., printing out values in for-loops and binary search through code with print statements) on the same three small programs that contained bugs that were given as a practice exercise to the treatment group.

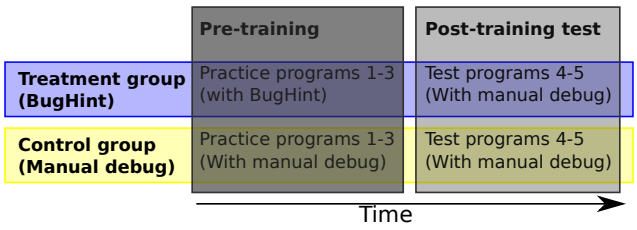

Figure 7: Experimental design. Performance on the Post-training test is the key indicator of training success.

These two groups of subjects completed identical post-training debugging testing activities on three more small programs that contained bugs; neither group received BugHint help. In addition to being required to fix and document a bug, subjects were given the following questions for subjective evaluation:

– What time did you start this problem?: __AM / PM (circle one)
– What time did you fix the bug? __AM / PM (circle one)
– How confident are you that your program works correctly now? 1-5 Likert
– How difficult was the problem? 1-5 Likert
– How much would comments in the code have helped you debug it? 1-5 Likert
– How well-written was the code? 1-5 Likert
– What would make informative test cases to test the buggie program and why? Free response
– What would make have been helpful comments to have added to the buggie program to have helped find the bug? Free response

Importantly, during data entry, all answers were scored and entered by a 'blind' grader who did not know the full study design, intended results, or purpose of the study. All data processing was entirely automated using the same procedures for each measure (barring different statistical tests for binomial versus numerical data).

4.2 Experimental Results

We analyzed the proportion of students who found each bug in post-testing, comparing the group with BugHint pre-training (the treatment group) to the group with normal debugging pre-training (the control group). As shown in Fig. 8, the left panel represents the results for one test program, and the right panel represents the results for another test program. Z-scores for a between-group test for binomial data indicate the treatment group found more bugs during post-testing; success or failure finding the bug was encoded as a boolean, and thus a binomial test was required. Further, when turned into proportions, t-tests were also significant, though are not the most appropriate test. The t-statistic is the difference of means between compared groups, divided by the standard error of the mean, $(u_1–u_2)/SEM$. Generally with reasonable n, it is necessarily the case that when SEM error bars are not overlapping, the t-test would be be significant, with α about $p < 0.03$ on a 1-tailed test in the expected direction.

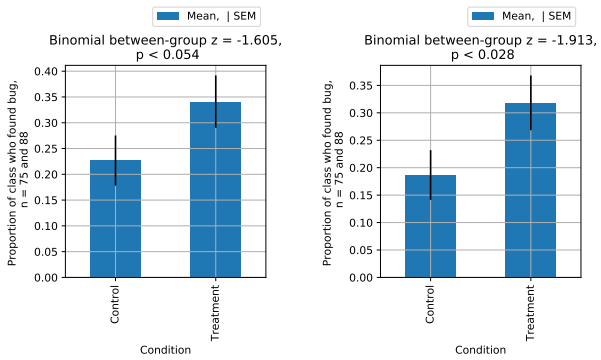

Figure 8: Proportion of students who found each bug for two test programs

Additionally, we examined how much time the students spent before they thought they had found a bug. Students in the treatment group reported finishing searching for the bugs with non-significantly less time than those in the control group. These data were not filtered on students actually having correctly found the bug, and so data merely represent the duration of time until they found what they thought was the solution. The comparisons between the treatment and control groups for two test programs are shown in Fig. 9.

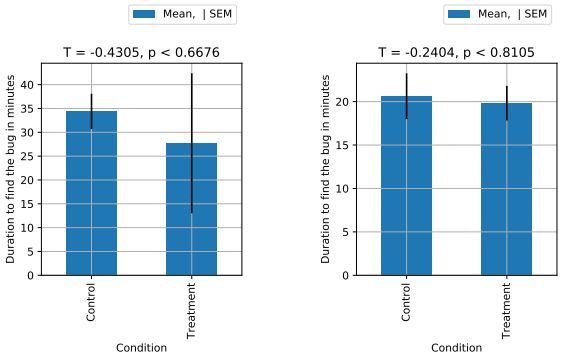

Figure 9: Duration of time spent before students thought they had found the bugs for two test programs

The students were given a few sample test cases (in terms of input and output) for each program that they were asked to debug. They were then asked to come up with additional informative test cases, and their reasoning for such test cases. These responses were blindly graded and given a point value from 1-5, with 5 being correct and good reasoning. Students who completed the BugHint pre-training demonstrated superior ability to come up with and reason about test cases for debugging. As shown in Fig. 10, t-tests and SEM bars indicate "significance", and support the conclusion that the BugHint group demonstrated better post-test performance.

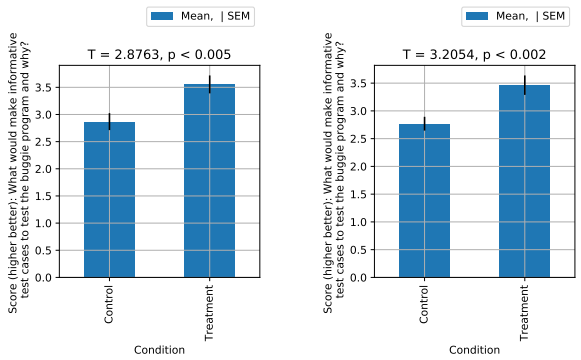

Figure 10: How well students came up with informative test cases for two test programs

In addition to assessing the importance of test cases, we sought to evaluate the impact of commenting in relation to debugging. The students were asked what would have been helpful comments to have included in each program they debugged. These comments were manually graded and evaluated by the blind grader, and given a score from 1-5, with 5 being best. Again, t-tests and SEM bars tend toward "significance." As shown in Fig. 11, we found that the BugHint group appeared to demonstrate better post-test performance.

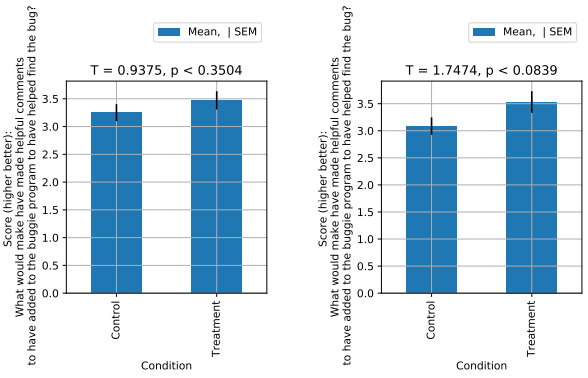

Figure 11: Helpfulness of comments that students added for two test programs

Finally, we asked two subjective questions regarding the students' overall experience. The first question was whether they felt that debugging was easier with BugHint. We found that a majority of students reported that finding bugs was easier after having done the pre-training with the BugHint methods than the manual debugging post-test. This is to be expected since hints (if correct and specific enough) should make finding bugs easier. The results are shown in Fig. 12 (left panel). Single-group binomial data do not have SEM, and a single-group binomial test-statistic with expected proportions of 50/50 was performed (as shown on the plot).

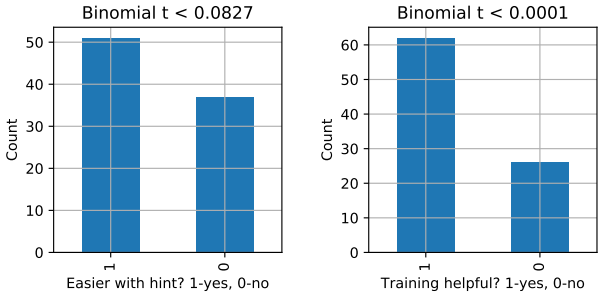

Figure 12: Left: Was debugging subjectively easier with BugHint? Right: Did training with BugHint help you learn to debug even without it?

The second question was whether the students thought that training with BugHint would help them learn to debug without BugHint later on. Students in the treatment pre-training condition felt that they benefited from the pre-training, specifically in regard to their ability to debug later without its help. Single-group binomial data do not have SEM, and a single-group binomial test-statistic with expected proportions of 50/50 was performed as shown on the plot in Fig. 12 (right panel).

5 FUTURE WORK

The current implementation of BugHint is restricted to programs that do not contain user-defined functions. Our algorithm for finding a discriminative subgraph, which is based on analysis of the control flow graphs for correct and incorrect test cases, will be easily extendible to additional block structures. The larger challenge will be accommodating this additional visual complexity in the graphical user interface. We intend to perform usefulness and usability studies with novice programmers to find ways of implementing visual representation and navigation of the different modules of the control flow graph in a manner that they (the students) can best understand.

We also intend to consider using other existing algorithms for finding discriminative subgraphs (e.g., [15] and [16]) 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. For example, we may prioritize subgraphs containing statements with multiple operators and/or particular operators (e.g., && and ||), which are the source of common (logic) mistakes for novice programmers.

6 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. The tool includes a visual display of the control flow graph for each test case, allowing the user to step through the statements executed. 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 more 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. In general, it appears that the extra-formalized method of using BugHint may improve the way students think about the debugging practice. We speculate that this may be due to one requirement of the BugHint method, which is focused on the specification of some test cases that generate correct output and some test cases which produce incorrect output. This exercise itself could be performed to further test that hypothesis.

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

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

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.