PROGRAM VISUALIZATION: AN EXPLORATION OF GRAPH BASED VISUALIZATIONS TO ASSIST IN STUDENT LEARNING AND PROGRAMMATIC EVALUATION

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Taylor Woods

June 2015
© 2015
Taylor Woods
ALL RIGHTS RESERVED
COMMITTEE MEMBERSHIP

TITLE: Program Visualization: an Exploration of Graph Based Visualizations to Assist in Student Learning and Programmatic Evaluation

AUTHOR: Taylor Woods

DATE SUBMITTED: June 2015

COMMITTEE CHAIR: Zoë Wood, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: John Clements, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Aaron Keen, Ph.D.
Professor of Computer Science
ABSTRACT

Program Visualization: an Exploration of Graph Based Visualizations to Assist in Student Learning and Programmatic Evaluation

Taylor Woods

As computer science students develop more complex programs at the end of their first year of course work, comprehending the complex and varied interactions of program execution, potential control flow and data relationships become more and more difficult. Additionally, for instructors when evaluating student’s programs, a simplified view of more complex (longer) programs is desirable. This thesis explores algorithms to create a tool for students that provides a simplified view of these concepts via visualization. The tool created for this thesis provides interactive visual representations of student programs. This allows for a simplified representation of the entire program along with depth exploration options to examine potential control flow and data access/mutations.

The following is an exploration of program visualization, with a focus on usability in an educational setting. Two main approaches will be discussed. The first attempts to visualize the call graph of a running program by showing what methods call what other methods, and the frequency in which they are invoked. The second shows all potential paths through a non-running program, by viewing the program on a per-method level. This approach also includes information about how methods interact with data.

As a test case this thesis focuses on a spell check program which builds a binary search tree dictionary then searches it for input strings and provides correction suggestions if the input is not found in the dictionary. We present an evaluation of our tool via creating visualizations of four different student implementations of this program. These visualizations are then analyzed by computer
science faculty to identify common threads throughout all submissions, as well as areas where individual students struggled or excelled. Additionally visualizations are used as a tool in a lecture instructing students about binary search trees. The students provide feedback as to the effectiveness of the visualizations and their comprehension of the material. We conclude that program visualization is a difficult task, especially when students are unused to visualizing control flow. Results indicate potential for use as both a student and instructor tool, though further research is required to identify optimal usage.
ACKNOWLEDGMENTS

I’d like to thank my family for pushing me to be the best I can be and supporting me always. A big thanks to Dr. Zoë Wood for being a wonderful professor, adviser, boss, and mentor. I would not have been able to do this without your help and encouragement. Finally a word of thanks to the entire Cal Poly CSC faculty; your passion inspires us and your knowledge guides us. You are the reason that we are successful.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>ix</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Program Visualization</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Comparative Design</td>
<td>2</td>
</tr>
<tr>
<td>2 RELATED WORK</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Program Visualization for Education</td>
<td>4</td>
</tr>
<tr>
<td>2.1.1 Learning</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Virtual World</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Visual IDEs</td>
<td>5</td>
</tr>
<tr>
<td>2.4 Graph Visualization</td>
<td>9</td>
</tr>
<tr>
<td>2.5 Graph Layout</td>
<td>10</td>
</tr>
<tr>
<td>3 SYSTEM OVERVIEW</td>
<td>13</td>
</tr>
<tr>
<td>4 PROGRAM ANNOTATION AND EXECUTION</td>
<td>15</td>
</tr>
<tr>
<td>4.1 File Parsing</td>
<td>15</td>
</tr>
<tr>
<td>4.2 File Markup</td>
<td>16</td>
</tr>
<tr>
<td>4.3 Creating the Program Flow</td>
<td>20</td>
</tr>
<tr>
<td>5 PROGRAM VISUALIZATION AND ANIMATION</td>
<td>21</td>
</tr>
<tr>
<td>5.1 Global Program Visualization</td>
<td>21</td>
</tr>
<tr>
<td>5.2 Method Graph Generation</td>
<td>23</td>
</tr>
<tr>
<td>5.2.1 Curved Edges</td>
<td>27</td>
</tr>
<tr>
<td>5.2.2 Interaction with the Visualization</td>
<td>27</td>
</tr>
<tr>
<td>5.3 Stack Storage</td>
<td>28</td>
</tr>
<tr>
<td>5.4 Data Display</td>
<td>29</td>
</tr>
<tr>
<td>5.5 Animation</td>
<td>31</td>
</tr>
<tr>
<td>5.5.1 Creating the Path</td>
<td>31</td>
</tr>
<tr>
<td>5.5.2 Displaying the Animation</td>
<td>33</td>
</tr>
<tr>
<td>5.6 Various Views</td>
<td>35</td>
</tr>
</tbody>
</table>
5.7 System Shortcomings ........................................... 37

6 VERIFICATION ...................................................... 39
6.1 Student Evaluation ............................................. 39
   6.1.1 Experiment .................................................. 39
   6.1.2 Analysis ..................................................... 43
   6.1.3 Additional Study .......................................... 44
   6.1.4 Improvement Opportunities ............................ 45
6.2 Instructor Evaluation .......................................... 46
   6.2.1 Experiment .................................................. 46
   6.2.2 Analysis ..................................................... 46

7 CONCLUSION ......................................................... 49

BIBLIOGRAPHY ......................................................... 51

APPENDICES
   A 102 Animation Data ........................................... 57
   B 102 Static Data .................................................. 68
   C 103 Data .......................................................... 79
   D Instructor Data ................................................ 82
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Program Visualization as a Virtual World</td>
<td>5</td>
</tr>
<tr>
<td>2.2 ViLLE Visualization</td>
<td>7</td>
</tr>
<tr>
<td>2.3 VIP Visualization</td>
<td>8</td>
</tr>
<tr>
<td>2.4 VISP Visualization</td>
<td>10</td>
</tr>
<tr>
<td>2.5 Non-overlapping Force Directed Graph</td>
<td>11</td>
</tr>
<tr>
<td>2.6 Directed Acyclic Graph</td>
<td>12</td>
</tr>
<tr>
<td>3.1 Full Visualization Screen Including The Method Call Graph, Data (In Purple) and The Stack (In Green)</td>
<td>14</td>
</tr>
<tr>
<td>4.1 Model Definitions of For and IfThenElse</td>
<td>16</td>
</tr>
<tr>
<td>4.2 Branch Numbering</td>
<td>19</td>
</tr>
<tr>
<td>5.1 Original Graph Design</td>
<td>22</td>
</tr>
<tr>
<td>5.2 Visualization of a Nested Loop</td>
<td>24</td>
</tr>
<tr>
<td>5.3 Graph Branching Example</td>
<td>26</td>
</tr>
<tr>
<td>5.4 Status of the Stack a few Levels Deep in Program Execution</td>
<td>29</td>
</tr>
<tr>
<td>5.5 Data Connections to Methods</td>
<td>30</td>
</tr>
<tr>
<td>5.6 Visualization of Animation in Progress</td>
<td>32</td>
</tr>
<tr>
<td>5.7 Animation Performed in Step Into Mode (Top) vs Step Over Mode (Bottom)</td>
<td>34</td>
</tr>
<tr>
<td>5.8 Linear Layout</td>
<td>35</td>
</tr>
<tr>
<td>5.9 Circular Layout</td>
<td>36</td>
</tr>
<tr>
<td>6.1 Average Student Grades</td>
<td>41</td>
</tr>
<tr>
<td>6.2 Average Student Ranking of Visualization Effectiveness. Higher Scores Indicate Greater Perceived Effectiveness</td>
<td>42</td>
</tr>
<tr>
<td>6.3 Categorization of Student Feedback Based on Provided Comments on Visualization Effectiveness</td>
<td>42</td>
</tr>
<tr>
<td>6.4 BST Data Representation</td>
<td>44</td>
</tr>
<tr>
<td>6.5 SpellCheck Implementation Showing Redundancy</td>
<td>48</td>
</tr>
</tbody>
</table>
Learning computer science is hard. Failure and drop out rates in introductory CS courses are incredibly high. Estimates place failure and dropout rates in introductory CS courses in the range of 15-30\% [14]. A number of recent studies have been focused on identifying and correcting this issue. An excellent analysis of such studies is presented by Sheard et al. [25]. In addition McCracken et al. find that even students who complete and pass introductory courses display sub-par programming skills [20]. This can partially be attributed to the way programs are graded; generally the student’s program is tested against the instructor’s implementation for correctness under various inputs. What this fails to capture is how the student is approaching the problem, or designing their solution. This thesis proposes a method for visualizing student programs which can help both students and professors identify and correct potential issues. It has been found that the usage of similar tools in introductory programming courses can greatly improve student learning [18].

1.1 Program Visualization

Program visualizations are a subset of computer visualizations in general that are focused on showing the execution of a program. It is important to keep this separate from algorithm visualizations. Algorithm visualizations focus on displaying the functionality of a specific, or class of, algorithm. For example, it is very common to see visualizations of sorting algorithms [24]. Overall, algorithm visualizations are targeted visualizations of a specific algorithmic process, while
Program visualizations are general and focus on creating a visualization for any implementation of an algorithm or problem. The goal of program visualization, and this thesis work, is to generate an accurate image for any input program.

There are many potential ways to create program visualizations [26], but a common technique, and the one employed by this thesis, is through a graph. Within this graph, methods are represented by nodes and potential execution paths are represented by edges.

Program visualizations tend to show either program execution, or a view of data access and manipulation. This thesis presents a visualization of program execution, but it should be noted that many program visualization tools focus on the display of data instead [13, 15, 23].

The execution graph created for this thesis is constructed by running the program in question and keeping track of the order in which it executes and returns from functions. From this ordering, subsections of the graph for each method can be dynamically built echoing the execution state of the program. The benefits of this are that the subsections of the graph corresponding to different methods can be constructed on an “as needed” basis, i.e. only when the user desires to investigate a method, or during animation. Also this ordering can be used to step both forward and backwards through program execution.

1.2 Comparative Design

This thesis describes a system which constructs interactive program visualizations for Java programs. This system is expected to be used as a tool for beginners to better understand how their program is running as well as a comparative tool for instructors to evaluate student work. It shows promise in the area of comparing alternate implementations, for example student against instructor.
By allowing students to see, at a macro level, how their program is operating and comparing it to the operation of their instructors program, students can reach a better understanding of proper program design. It is important to note that this comparison will only expose the code structure, while the implementation details are still left to the student to determine.

In order to allow students to better understand proper program design, we must have some idea what that represents. It is difficult to say exactly what constitutes an ideal program design. There have been a number of studies attempting to define software quality metrics [1, 3, 6] which address the quality of a program. These metrics however, tend to focus on large scale software development projects, and as such are not very applicable to introductory programs.

This thesis will focus on Object Oriented (OO) languages, namely Java. In OO languages we look for encapsulation of functionality within classes and limited dependence on other classes save those within a class’ inheritance tree. Additionally within this setting we have a reference solution to the problem which we may compare to. Although the instructor’s implementation may not always be “ideal” this thesis will treat it as such. One would expect the graph built from this implementation to be nicely laid out with easily identifiable call paths running through it.

Our goals for this thesis are to generate a visualization of a program which:

- Shows execution paths through methods
- Allows for visual identification of redundancy within methods
- Allows comparison across implementations
Chapter 2

RELATED WORK

This section details related work in two main areas of focus: program visualization and graph layout.

2.1 Program Visualization for Education

An extensive review of program visualization systems by Sorva et al. [26] shows roughly three different approaches taken when designing program visualization systems. First and most common is to provide a visual IDE (Interactive Development Environment). In these systems the source code is displayed on part of the screen while another part is reserved for the visualization. These visualizations are typically pretty simple often consisting of only variable names and values. The second type of system is essentially a video game in which program statements are mapped to actions or interactions within a virtual world. The final type of system is a graph construction as done in this thesis.

2.1.1 Learning

There is a large body of work focused on student learning in general. An excellent overview and analysis of which is provided by Sheard et al. [25]. This thesis is not focused on evaluating student learning, and instead is focused on automatically creating a graph layout of a program to be used for educational purposes. This layout allows comparison between programs and aims to assist in basic understanding of the program execution. Future studies focused on exact learning outcomes are beyond the scope of the current project and are left for
2.2 Virtual World

The virtual world system of program visualizations will be discussed first, as it holds the least relevance to this thesis. While still a viable method of program visualization it operates a much higher level of abstraction than the other formats of program visualization. An example of this type of system is presented by Jimenez et al. [17]. It produces a visualization as shown in Figure 2.1. This system ViRPlay3D2, demonstrates the interactions between objects in an OO environment. Students control avatars, representing objects, whose actions represent interactions within the system.

2.3 Visual IDEs

The next type of visualizations to be discussed are visual IDEs, where part of the screen contains the code being visualized, and the other part contains some visual representation of that code. There are a wide variety of these systems for a number of languages.
As mentioned in Section 1, students struggle to grasp basic computer science principles. Kaila et al. [18] show that program visualizations are an effective teaching tool. They do this by comparing final exam scores for two sections of the same class. One section uses a program visualization tool ViLLE for one lab at the beginning of the course. The other section uses ViLLE on all assignments throughout the course. They find that the section which used ViLLE throughout the course got significantly better scores on the final exam. This shows that at least this specific visualization tool provides a teaching benefit.

ViLLE is first introduced by Rajala et al. in [23]. ViLLE is a generalized program visualization tool which supports Java, C++ and user defined pseudo-languages. It provides animation support as well as stepping through programs forwards and backwards. It is displayed as a visual IDE with code on the left and the visualization on the right as shown in Figure 2.2. It primarily differs from the system presented in this thesis due to its IDE based approach instead of a graph based approach, as well as its visualization focusing on data instead of control flow. Additional similar tools, both for Python, include Jype [15] and Python Tutor [13].

Another study examining the usefulness of program visualization was conducted by Lahtinen et al. [19]. They allowed the use of a program visualization tool, VIP, on all assignments. VIP again uses the concept of a visual IDE, as shown in Figure 2.3. The use of this tool was completely voluntary. They found that half the students tried the tool at least once, and a quarter of the students used the tool for all their assignments. The students which used VIP for all their assignments were found to be more likely to continue turning in assignments and less likely to drop out than the group which never used the tool. Additionally it was found that more students chose to use the tool for more difficult assignments. This shows that students think visualizations can be helpful in learning difficult
Figure 2.2: ViLLE Visualization
Figure 2.3: VIP Visualization
concepts.

The effectiveness of program visualizations as a teaching tool is further explored by Urquiza-Fuentes et al. [29]. They find that PAV’s (Program/Algorithm Visualization) are a viable tool for teaching computer science. They break learning into three major categories: knowledge acquisition, attitude, and skill. As may seem reasonable they find that PAV use has the largest effect on knowledge acquisition.

Additional research on the effectiveness of program visualization, specifically the role of animation, has been studied by Nevalainen et al. [21]. They focused on the short term effects of a program visualization, e.g. what does a user focus on during the visualization and what do they take away immediately afterwards. In the study they provide the source code on the left with active lines highlighted, and an animation of variables and their values on the right. The results show that participants actually spent most time viewing the source code, however, as execution progressed more focus was paid to the animation. These results are rather open ended, it could mean that having an animation is truly not useful, however, it could also be that they have a poor animation or that it is actually distracting to display the source code alongside the animation. They also bring up a concept which may explain where student’s difficulties come from. They state, “In programming education, programming knowledge is usually introduced through program knowledge: students are given example programs and they are supposed to construct the general programming principles by themselves.”

2.4 Graph Visualization

Callaghan et al. [4] describe a system, VISP shown in Figure 2.4, for visualizing Java programs which is the most similar to the system described in this thesis.
This system is capable of visualizing small Java programs in 3D, showing both class definitions and instantiated objects as well as the program flow between them. A major downside to this system is that it requires manual effort to convert programs into the format recognized by the visualization. The biggest differences between VISP and this thesis is VISP’s use of three dimensions, and VISP requiring a special input format whereas this thesis uses standard Java. Reducing the drawing to two dimensions greatly simplifies what a user is viewing, however, it also means less space to work with. To that end the graph layout algorithm becomes a very important factor in the usability of the final product.

2.5 Graph Layout

There are a huge number of algorithms for drawing graphs, a bibliography of such algorithms is presented in [9]. Two different approaches are applied in this thesis. The first being force-directed, wherein sibling nodes repel each other, and edges are set to a fixed length. This causes nodes to fan out around their parent. One problem with this approach is that it can lead to node overlapping in large,
highly connected graphs. This can be solved using the algorithm proposed by Dobkin et al [8]. It produces graphs as shown in Figure 2.5.

The second layout algorithm used in this thesis is a specialized approach for drawing a flow graph which is most similar to that produced by Gansner et al [12]. Their system draws directed acyclic graphs, but in simple cases can produce an output like that shown in Figure 2.6. Due to the inherent mapping and constraints acquired from structured source code, the layout system presented in Section 5.2 is substantially simpler.
Figure 2.6: Directed Acyclic Graph
The system presented in this thesis creates an interactive visualization of an input Java program. This visualization is made up of multiple components, first and foremost the methods called within the program. Then additionally the visualization consists of, visual representations for: the stack, data that is used in methods from parameters, and finally an animation which traces a particular run through the program. See Figure 3.1 for an example of the complete system. Creating a visualization of a program requires a number of steps, those taken by this thesis are listed below.

1. Parse files
2. Markup files
3. Run program
4. Construct Graph
5. Visualize the program

Of these, the first three are what is known in graphics as a preprocess, anything which happens before the screen appears. Preprocesses do not impact the graphical performance of the application once it is running.

Steps 1 - 3 will be discussed in Chapter 4. Steps 4 - 5 are discussed in Chapter 5. Note that steps two and three above are required solely for the purpose of animation. Creating a static, but still interactive, visualization requires only parsing the files, then the graph can immediately be created. Visualizations generated look something the layout shown in Figure 3.1
Figure 3.1: Full Visualization Screen Including The Method Call Graph, Data (In Purple) and The Stack (In Green)
Chapter 4

PROGRAM ANNOTATION AND EXECUTION

As the goal of this project is to create visualizations of Java programs, the first step must be parsing the program into a usable format. Afterwards, in order to create an animation, we will annotate the input source code in order to track the path taken when the program is run. This chapter details the processes used in program parsing and annotation.

4.1 File Parsing

Program parsing is done using the PYLJ\textsuperscript{1} package. PLYJ is a Lex Yacc parser meaning that its functionality is split into two distinct components. Lex is a table of regular expressions which are to be found within an input stream. Yacc, short for Yet Another Compiler-Compiler, is a mapping of keywords to methods. The combination of these two tools allows source code of a given language, in this case Java, to be represented as data structures in a separate language, in this case Python. PLYJ is split across two files, the first holds the logic for parsing a Java file, the second holds the python models which will be created. The parsing code has been slightly augmented to include line numbers when creating models. The model code has been heavily modified, mainly the addition of helper methods for traversing the models and extracting pertinent data. As an example, the constructors of the For and IfThenElse models are shown in Figure 4.1. The most important fields in these models are body in For and if_true and if_false in IfThenElse. These fields hold references to additional models, for example, body is an array of all statements contained within the for loop. This leads to a nested

\textsuperscript{1}https://github.com/musiKk/plyj
4.2 File Markup

There are a couple of potential strategies for tracking the execution of a program. One option is to use a debugger for the language in question and inspect the program state at each step. The approach taken in this thesis however, is to insert print statements as markers into the code at strategic locations. First, we will insert markers as the first and last line of every method, as well as directly before any return statement. These markers will inform us as to the name of the method, the name of the class, and the current thread of execution. Additionally we will place markers at locations in the code where a branch occurs, for example the first line of an if statement. These markers include a branch number which is used to map a location in code to a location within the graph. Then when the program is run, we receive a list of the methods which the program entered, in...
the order in which they were executed, as well as which specific branches were taken to travel through the method. We can then trace through this list to follow the path which the program took.

A motivating example for the necessity of tracking these code branches is shown below.

```java
public void branchEx() {
    if (One()) {
        Two(true);
    } else {
        Two(false);
    }
    return;
}
```

If we only track the current method of execution, we will see the program enter branchEx(), then One(), then Two(), but we have no way to identify which instance of Two() was called. A branch numbering system is developed which allows us to track where within a method a program is executing, and map that location to the proper node in the graph. The numbering system is simple, starting at zero the branch is incremented by one when a new branch is entered or when branches merge. This would lead the above example to be numbered as follows.

```java
public void branchEx() {
    // branch 0
    if (One()) {
        // branch 1
        Two(true);
    }
    return;
}
```
Branch numbering is reset at the beginning of each method, such that every method starts at branch zero. The syntactic restrictions imposed by Java limit the locations we can insert print statements. Take the following example.

```java
public void branchEx() {
    // branch 0
    if (Zero()) {
        // branch 1
        One();
    }
    // branch 2
    else if (Two()) {
        // branch 3
        Three();
    }
    // branch 4
    return;
}
```

Java will not allow us to place a print at the location of branch 2. This means we need to identify the location of all branches as well as which ones we are able
to mark and which ones we are not. Any model which results in a branch, if and
any loop (for, foreach, and while), has a branch_line_num field which is computed
in that models constructor. This field holds an array of the line numbers of the
locations where branch markers should be placed in the source code. Entries
of negative one are used to represent InvisibleNode locations. InvisibleNodes
are necessary for correct branching graph layout and are explained in Section
5.2. These line numbers are calculated with the same system which builds the
branched array used in Section 5.2. This assures that the branch numbers as
marked in the source code match up with the branch numbers in the graph.
These branches are numbered in a depth first fashion, going upwards as shown
in Figure 4.2.

This numbering scheme is important to the ability to determine the proper
path taken during animation when transitioning from one node to another. Notice
that any path through the method will result in continually increasing branch
numbers, with one exception. When repeating within a loop we will encounter
a branch number which is less than or equal to our current branch. This can
guide the path calculation to ensure it loops back properly. The other important
constant is that, with the exception of loops, InvisibleNodes will always be on
the bottom branch if present. This means that when determining paths, if we are
unable to find the node we are searching for in either of the branches attached to
the current node, we will take the bottom branch.

4.3 Creating the Program Flow

With source files marked up we now need to run the program in order to find the
specific path taken. Once a file has been marked up it can be compiled. As files
are parsed we track which file contains main. When all files have been marked
up and compiled, we then run the compiled file which holds main. We track all
output of the program as it runs. We filter out any statements which were part
of the original program, and relay them directly to the user through standard
out. The remaining statements, those which we added to the program, are stored
in a list referred to as the program flow. This program flow will then enable
animation as discussed in Section 5.5.
To visualize a graph of the program execution, Pyglet is used for graphical output and interaction. Pyglet\textsuperscript{1} is a high level graphics API for Python which provides OpenGL bindings as well as a windowing system and additional multimedia support for things such as audio. An benefit of Pyglet is that it has no additional dependencies simplifying distribution and installation. The visualization of this thesis is done entirely in 2D using the fixed function graphics pipeline (no custom shaders).

The system presented in this thesis allows user interaction through the mouse and keyboard. With the mouse users can navigate through methods as well as explore data connections. The keyboard allows camera control with both movement and zoom, and the ability to control animations.

5.1 Global Program Visualization

Before jumping into the components of the final system, we present the original design of the visualization. The initial design of the system was focused on a similar but different visualization. This visualization attempted to show all of the program call graph in one screen.

In this visualization the graph is built up step by step through the program adding new nodes when a method is called for the first time. Edges are created between methods representing one method calling another. The thickness of the edge shows the frequency of that call. See Figure \textsuperscript{5.1} for an example.

\footnote{http://www.pyglet.org}
Figure 5.1: Original Graph Design
Ultimately this design was abandoned due to complications as the input program grew in size. Because the graph had to be constructed one step at a time it made animation through construction virtually impossible. The final version of the graph could still be constructed in semi-reasonable time. Additionally it was determined that this full call graph ultimately does not represent how programmers tend to think about systems, as it shows the entire problem space instead of the decomposition of a problem into methods.

Still this type of visualization may serve as some use in analysing common paths through systems. Much of the code and techniques developed for this visualization was re-purposed for use in the final system, most notably the print system for tracking movement through a program, as described in Section 4.2.

5.2 Method Graph Generation

A natural structure to represent a program’s execution is a graph. Commonly referred to as the call graph, this is a structure where graph nodes represent the methods of a program, and edges represent a call from one method to another. We visualize the complete call graph at the method level. That is, within a given method we show all possible paths the program can take.

We refer to this graph as the method flow graph, and it represents the primary component of this thesis. The graph is constructed using a recursive algorithm which assumes worst case conditions. This allows it to guarantee that nodes will never overlap, and compute their layout rather quickly. The downside is that the node placement may become sub-optimal for graphs which are particularly wide.

The algorithm works by placing nodes in a line until it reaches a branch. Once it reaches a branch it finds the total width of that branch and uses that to determine the vertical position of the node. Similarly when merging branches together
it finds the longest of any of the branches to determine the horizontal position of the node. These positions are then stored by the nodes for drawing. Additionally before being drawn a camera transform is applied, allowing movement around the graph without updating node positions.

Nodes in this graph represent methods in the input program’s source code. With additional nodes added for Start(), Return(), Break(), LoopStart(), LoopEnd(), and Repeat(). Nodes are drawn in light orange if their method definitions come from the input program’s source code, these nodes will be referred to as user nodes. These are nodes which the user is able to navigate into. Other nodes are drawn in dark orange, which represent the special nodes listed above, as well as library methods, these nodes will be referred to as non-user nodes. Examples of this coloring scheme can be found in Figure 5.2.

Individual branches are represented in an array. The full method structure is created by nesting these branches together. When traversing this nested array encountering the start of a new array represents the start of a new branch. As an example if given the following code.

```java
if (One()) {
```
Two();
if (true){
    Three();
}
else if (Four()){
    Five();
}
The following array would be created.

```plaintext
[ 'Start', 'One',
  [ 'Two', 'InvisibleNode',
    [ 'Three'],
    [ 'Four',
      [ 'Five'], [ 'InvisibleNode']
    ]
  ],
  [ 'InvisibleNode'],
  'Return'
]
```

Which creates the graph shown in Figure 5.3.

The graph may require branching that is not inherently obvious from the source code. For example in the code shown above we need to display the path
which the code would take if `One()` returned false. This leads to the path going from `One()` to `Return()` in Figure 5.3. To handle these cases we introduce the use of InvisibleNodes. An InvisibleNode is a special node which is used to create paths which may not be immediately apparent from, or present in, the given source code. For example an if without an else should generate two branches even though the code only appears to have one. These nodes are not drawn, but their position is still tracked to allow the creation of paths without any methods.

The graph layout produced by this system is drawn entirely in 2D, meaning that the order in which things are drawn is important to the correctness of the visualization. Dealing with this restriction requires two passes through the nodes to achieve a correct render. In the first pass only the edges of the graph are drawn, the second pass draws the nodes, which includes their text labels. This ensures that all nodes are drawn on top of all edges.

5.2.1 Curved Edges

The curved edges connecting nodes are generated using a quadratic spline system. A control point is placed by finding the midpoint between start and end nodes then traveling along the vector perpendicular to the direction between the start and end nodes. With these three points (start, control, and end) a quadratic equation is generated by solving a matrix inversion. Small line strips are then placed along this equation between start and end, in order to generate our final curve.

5.2.2 Interaction with the Visualization

Interaction with this graph is made possible with a brute force collision detection testing the cursors position against the regions of all nodes. It utilizes a trans-
lation from camera space to node space in order to handle collisions properly as
the camera moves and zooms. If a method is found to be clicked its parsed model
is then found and a new graph is generated from it.

5.3 Stack Storage

In order to accurately represent the execution of a program we need a way to
track and visualize the stack. Naturally this is represented internally using a
stack data structure. The internal stack is made up of Frame’s which track four
important things; branch number, branch index, camera location, and the node
which holds the parsed model for a given function. The data stored within a
Frame is the minimally sufficient information to recreate the visualization in its
prior state when returning from a method. This stack uses the traditional push
and pop operations, while also implementing a pop_to method. Pop_to takes in
a node and pops down the stack until that node is found, this allows us to jump
back multiple levels of the stack in a single click.

The stack is visualized in the bottom right corner of the screen growing ver-
tically upwards. Each Frame is drawn by its node, the same class used for the
graph layout. Since the stack is drawn as a static element, it is drawn without
any camera transformation applied. This allows the stack to remain in the same
location on the screen regardless of camera position or zoom level. An example of
the stack is shown in Figure 5.4. The stack provides interaction through mouse
clicks, which invoke pop_to when a Frame is clicked. The top element of the stack
always represents the method currently being visualized.
5.4 Data Display

In order to give an indication of how a method interacts with data we include a visualization of method parameters and their usage throughout a function. We attempted to include this same visualization for all data (primitives and objects) created within a method, but found that for many methods the data space was too large, and cluttered the visualization. As a result only method parameters are shown, which represents a reasonable balance between information and understandability.

Data is visualized in the bottom left corner growing right. Each argument is contained within its own box with a text description. If the data is a primitive, the box will include its type and name. If the data is an object defined by the input program, the box will include its type, name, and the types and names of
all of its fields. The data visualization presents two types of interaction to tie it into the graph. First, hovering the mouse over the data will draw a connection from that data to every method in the graph which uses it. Second, hovering the mouse over any method in the graph will highlight the data used by that method. Objects can be highlighted on a per-field basis, or the entire object. An example of the data, when being hovered is shown in Figure 5.5.

Relevant data for a method is determined upon entering that method by looking at its parameters. If the parameter is a primitive we can just create its data view. If it is an object however we must determine all fields of that object. When the Java program is first parsed all class definitions are identified and stored. At this point we look through those stored class definitions until we find
the type matching that which we are searching for. We can then search through
the class definitions to find all fields, a minor optimization could be made here
to store this information so as to not need to search the definition every time.
These fields are then passed back to the data view to include.

A similar approach is taken to enable the interaction described above. When
it is detected that the mouse is hovering over a node on the graph, the data used
by that method is determined and then the data visualization is scanned for any
matches. Similarly, if a hover over a data node is detected the entire graph is
scanned for usages of that data. Any methods which do use that data create an
edge between itself and an InvisibleNode placed behind the data view.

5.5 Animation

At this point we have constructed a visualization of all possible paths through
a program. We now tackle a method for tracking a specific path taken through
a program. This will be done via an animation following the program flow as
constructed by the file markup. With this animation we can see the exact path
taken through a program for a given input.

Adding animation into this graph creates a couple of interesting problems.
First is tracking the branches taken by a specific run of a program. Second is the
visualization of this data, mainly due to the 2D rendering system described at
the end of Section 5.2

5.5.1 Creating the Path

The path to animate is stored in animation_path, a list of nodes which the an-
imation needs to visit in order. An animation path is computed for every new
method entered, which moves the animation from its current position to that method. This path is made up of nodes which the animation needs to visit in order to properly trace the path. Note that this can be an arbitrary number of nodes due to non-user nodes, but must end in either a user node or a Return(). An example of an animation in progress is shown in Figure 5.6. The remainder of the animation path is drawn in blue, showing that this animation will continue until reaching the ItemRecord() method at which point a new path will be calculated.

Construction of the animation path is done using the program flow. This list is stepped forward from a current location until the next method push or pop is found. During the stepping process many branch prints may be encountered; these are saved to a new list to be followed before reaching the next method. Iterating through all of these branches we create a path from our current branch to the target branch by adding all methods remaining in the current branch to the animation path, then determine if we can follow the current branch to the target branch. If we can reach the target branch, we move to it. Otherwise we follow the bottom branch out, so long as the bottom branch doesn’t cause us to overshoot the target branch. Once the target branch is found we add everything
up to and including the target method. If we were searching for a pop rather than a push, the target method simply becomes “Return()”.

5.5.2 Displaying the Animation

The constructed animation path is then fed to the AnimationDot item by item. AnimationDot is in charge of calculating the path from its current position to the position of the next node and animating along that path, as well as notifying the graph of two important events. First the graph is notified when the AnimationDot completes its animation, at which point the next item from the animation path will be given to it, if one exists, otherwise the next path will be calculated. Second the graph is notified when the animation is halfway complete. The reason for the halfway complete callback is tied into the 2D rendering technique mentioned earlier. Since the visualization is drawn with no depth information, the draw order is important. In order to get the animation desired, it needs to be drawn after the node it is currently on top of, but before that node’s label, so as not to obstruct it. Since nodes and their labels are drawn one after the other, the AnimationDot needs to be injected in the middle of one specific nodes draw call. This introduces the concept of the active node, which tracks which node the AnimationDot is currently on top of. The active node is passed the AnimationDot and will draw the dot after it has drawn itself, but before it renders its own text. Thus notifying the graph when the animation is halfway complete allows the active node to be updated so that the dot stays on top of nodes but below labels at all times. Additionally by updating this value halfway through the animation we ensure that the dot is currently on the path spanning two nodes, meaning that no overlap between the dot and a node will occur.

We allow the user some control over the animation through modes analogous
to debugging’s step over, and step into. The difference between these is shown in Figure 5.7. In this figure the top three frames represent a progression through an animation while using step into mode. Notice that in the third frame, the animation entered a new method insert(), and started executing. The bottom panels show an animation starting from the same position but using step over mode. Here instead of entering the insert() method, the animation steps over it and continues playing within the current method.

The user can toggle between step over and step into modes at will and are shown as different colors for the dot. This mode is used after the calculation of the animation path to determine how to update the flow index. When in step into mode, the index will simply be incremented. When in step over mode we need to seek forward through the flow until we find the point at which the method we are animating towards gets popped. Additionally this mode is used once the animation is complete, before a new path is calculated, to determine if we should enter into the method that was just reached. As well as when the end of a method is reached; if in step over mode, the animation will just stop, in step into mode the visualization will exit the method and continue the animation.
5.6 Various Views

The graphs presented thus far were generated using the branch layout. In addition to this, two other layouts are explored. Both these layouts make use of all prior described systems, changing only final node placement. First we see a linear layout which places all nodes at a set Y value as shown in Figure 5.8. This layout loses the ability to easily distinguish between branches, but clearly shows methods where multiple paths converge, and may prove useful as a comparison tool for viewing one graph next to another.

The second additional layout explored is the circular layout shown in Figure 5.9. This layout places nodes around a circle whose radius is determined based on the number of nodes in the method. This graph shows the same method as Figure 5.8. Note the location of large gaps between nodes is because InvisibleNodes are still given their place on the circle. Again this layout loses the ability to directly see branches, but has the upside that many more nodes can be fit on one screen. Because of this feature, the circular layout seems ideal for comparison of particularly large methods.
5.7 System Shortcomings

The overall system has a few limitations which need to be addressed here. First is the use of a simple string name comparison in determining which model is associated with a given node. This leads to an issue with method overloading both in terms of method parameters and access level. This problem is solvable by replacing all simple string comparisons with a comparison which takes the full method signature into account.

Next is that Java implements boolean logic short circuiting which this system has no way to detect. This can lead to a situation where a method call will be visualized as entering from the wrong branch. Take the following example.

```java
if (One() && Two()){
    //Something
} else if (Two() && Three()){
    //Something else
}
```

If the call to One() returns false, then Java will not attempt to run Two() within the first if check. It will instead run Two() within the second if check. The visualization system however does not detect this and thus will show entering method Two() from the first branch. This however does not cause any functional issues with the original program nor will it cause the system to crash. A fix to this issue would be creating a print function within the Java code, say branchPrint(x) which takes in a number and adds a branch print of that number then returns true. This method could then be added as the first term of every if, anded to the remainder of the expression. This would ensure a branch printout before any method call within an if check. Again this would not change the functionality of
the program, though it would simplify path finding and the markup logic.

A further consideration is that if a user clicks to enter a method, we need to update the flow index so that the animation is able to properly start at the beginning of that method. The approach taken by this thesis is to seek the flow index to the first occurrence of that method. This could cause confusion in that if a user clicks on a method near the end of main, but that method is also called earlier, when returning from the method the animation will jump to the earlier call of the method.

Finally, programs which operate on a large amount of data can take a substantial amount of time to run, due to the IO backlog incurred from print markers. A solution to this could be a minor modification of the run system to send the program output directly to a file which we read in. In the meantime our recommendation in such cases is to adapt the program to run on a subset of the full data, and create a visualization from that smaller input.
Chapter 6

VERIFICATION

This thesis presents a program visualization system which displays an interactive visualization of possible paths through a program at the method level, including an animation of a specific execution of the program. This is displayed as a 2D graph where nodes are methods and edges are possible paths. This system has a dual purpose as a student success tool in that it should be accessible to students to visualize their own or instructor code, as well as to instructors to be used to assess the quality of student submissions. As such we present two different experiments designed to measure this system’s usefulness to each of its desired audiences.

6.1 Student Evaluation

6.1.1 Experiment

This experiment focuses on the usefulness of this system as a tool for teaching students programming techniques. We hypothesize that this visualization is a useful tool to assist students in learning programming. Secondly we believe that the inclusion of animation will further enhance the usefulness of the visualization. To verify these hypotheses we present a lecture on a new topic to students, augmented with visualization, and then quiz them on understanding as well as their perceived effectiveness of the visualization.

The lecture was presented to four different CSC 102 (the third introductory programming course) classes during Spring quarter 2015, taught by two different professors. The students were given a lecture on binary search trees (BST), and
their potential usage in a spell check application. The students in these classes had not yet been introduced to the tree as a data structure. For each professor, one section was given a presentation involving only static visualizations, while the other section was presented visualizations including animation. Two animations were included. The first shows three items being inserted into an initially empty BST, after adding the root, one item is inserted to the right and the other to the left. The second shows searching for an item within a BST, which is not found.

After listening to the presentation they were given a quiz containing three questions judging the extent to which they absorbed the material. The questions included were as follows, all were multiple choice.

- What are the benefits of using a BST for spell checking as opposed to an unsorted array?
- When finding a word in the BST how do you choose where to look?
- If we have a balanced tree with n nodes, what is the maximum number of nodes we would have to check before finding a word?

Both professors whose classes were tested concluded that their students would be unable to answer any of the questions prior to the presentation. The results from these questions after receiving the lecture are shown in Figure 6.1. The grades from the quizzes have been normalized out of 10.

Additionally the quiz had an open ended question on if they felt the visualization helped them in understanding BSTs. As part of this question students were asked to rank the usefulness of the visualization on a scale of 1-10, 10 being the best. Unfortunately details of the ranking were only described verbally, not detailed on their quizzes. As a result a number of students failed to include a ranking. Out of the 111 total data points collected, only 65 of them included a
Figure 6.1: Average Student Grades

ranking. These 65 data points are shown in Figure 6.2. The labels above each bar indicate the number of data points in each section.

In order to make some use of the additional data which did not include rankings we group all results into three categories based on the students written feedback. Students who did originally include a rank were binned as follows ranks 7 - 10 were considered positive, ranks 4 - 6 were considered neutral, and ranks 1 - 3 were considered negative. Students who did not originally include a ranking were binned based on their comments about the system. Comments along the lines of “yes”, and “it helped” were considered positive. Comments including “sort of”, “yes and no”, or “a little bit” were considered neutral. Comments claiming “no” and “not at all” were considered negative. Results from this classification are shown in Figure 6.3.
Figure 6.2: Average Student Ranking of Visualization Effectiveness. Higher Scores Indicate Greater Perceived Effectiveness

Figure 6.3: Categorization of Student Feedback Based on Provided Comments on Visualization Effectiveness
6.1.2 Analysis

Viewing this data it is very difficult to conclude that the visualization affects comprehension, positively or negatively. We see an average grade across all groups of 75.4%. This average is a decent number considering the advanced nature of the material relative to the students knowledge level, and the time constraints of the presentation (20 minutes). A regression analysis of the relation between students grade and their ranking of visualization effectiveness finds no statistical correlation. For sections receiving static visualization the p value is 0.28, while the p value for sections receiving animation is .95.

Analyzing the presented data, we first notice that the group which liked the visualization the best, professor A static, had the worst test scores, by a significant margin. Next, comparing static to animation, we note that professor A’s classes preferred static to animated, but professor B’s was the other way around. Looking at just the animation data we see very similar results in terms of visualization effectiveness, but a noticeable difference in comprehension.

The results of this experiment certainly conclude negatively, or at least inconclusively, on hypothesis two, inclusion of animation will further enhance the usefulness of the visualization. Outlook on hypothesis one fairs similarly. The average rank across all groups is 6.55. If we take 5 to be rank at which students could either "take it or leave it”, then we can say that 6.55 is at least providing some benefit. Additionally when effectiveness is categorized into positive, negative and neutral, as shown in Figure 6.3, the majority of reviews claimed the visualization was helpful. Unfortunately we were unable to prove any correlation between students perceived effectiveness and their comprehension of the material. From these results we are unable to make any claims about hypothesis one, this visualization is a useful tool to help students learn programming. We
suggest further experimentation to identify better ways to use the tool to help student learning, or to identify what the visualization lacks that would make it more useful.

An additional observation may be made to the effectiveness of data visualization. Included in the slides, along with visualizations created by this thesis, were more traditional data visualizations of BSTs, like the one shown in Figure 6.4. Many of the students pointed specifically to images like this one being more useful in their understanding of the material than the execution based visualizations as produced by this thesis. This suggests that data visualization, at least in the specific case of BST may be an easier way to understand the problem.

6.1.3 Additional Study

An additional study was conducted on students in the next course in the introductory series (CSC 103) as an attempt to see if students who already knew BSTs could get more out of the visualization. These students were given a simi-
lar presentation to the prior animation presentation with slides explaining BSTs removed. They were quizzed in a slightly different manner as well; being asked to answer questions directly related to the animations they were shown. Unfortunately there was a misunderstanding of knowledge level of the students at the time of the presentation. The students had not yet been introduced to BSTs and as such with the explaining slides removed the presentation went completely over their heads. For completeness this data is still included with the Appendix, but no reasonable conclusions can be reached from it.

6.1.4 Improvement Opportunities

Student results to the open ended question about effectiveness prove a good place for improvement opportunities. By far the most common complaints among negative reviews were, “it’s too fast”, “it’s too hard to see”, and “I don’t understand what it’s doing”. The first two are likely artifacts of trying to use the visualization as part of a slide show. The last suggests a deeper problem that it’s not immediately obvious what the visualization is trying to show. This hints at the fact that the visualization is not particularly well suited for display within a slide show. Rather what may be more effective is allowing students to get their hands on the system and use it to visualize their own code throughout a course. This allows them to acclimate to what the visualization is showing starting with, presumably, simple examples. Also, as they would be in control, they can step through at any speed and resolution they desire. This research is left for future work.
6.2 Instructor Evaluation

6.2.1 Experiment

This experiment attempts to identify if professors are able to use the visualization to quantify the design of student programs. To do so, three professors were given this tool, along with four different implementations of a BST spell check program. Three of these implementations were taken from prior student work, while one was written by the authors of this thesis and designed to be “ideal”. The professors were asked to run the visualization on each of the four programs and answer the following questions.

- Are you able to identify the sections relating to inserting into a tree, searching a tree, and “spell checking”?  

- Do you find the structure of any of the methods in the program surprising? Which method? Why?

- Approximately how long did it take you to analyze each program? Do you expect that this is faster or slower than if you were looking at the source code?

- Do you feel you got enough information about the program to make a judgement on its quality? If not what was missing?

Finally they were asked to rank the four programs in order from best to worst.

6.2.2 Analysis

Results from this test show that all instructors were able to pick out the “ideal” design (s2). One professor gave no ordering for the remaining programs, but the
other two have a close remaining ordering of s4, s3, s1, and s4, s1, s3 respectively. Both s3 and s1 were reasonably poor designs and could easily be argued which was worse, though the expected order was s2, s4, s3, s1. These results show that the system creates a visualization which can represent the design of programs. As well as that professors are able to pick up on the differences between implementations.

Instructors spent an average of about six minutes per program, which is roughly the same amount of time they estimated it would take looking through the source code to answer the same questions. The question of if the visualization provided sufficient detail to make a judgement on quality received more mixed reviews. One instructor found it sufficient, while another wanted to be able to view the code side by side with the visualization. They were specifically informed not to look at the source code for the purposes of this experiment. The last suggested that they wanted the inclusion of a method purpose statement in order to more fully understand what methods are designed to do, as an addition to their name and parameters.

When it came to using the tool to identify surprising method structures two of the professors were more or less on the same page. They identified the main of one program and the spellcheck of another, both which were expected to be found surprising. The other surprising methods, though not consistent between professors, are all methods which the authors agree showcase interesting method structures.

The ability for instructors to identify specific portions of the code proved more difficult than expected. In general it seems that the ability to identify the functionality depended almost entirely on method names. This led to insert and search being identifiable, but spell check could often not be found due to the fact that two of the four programs did not implement a function directly called
Figure 6.5: SpellCheck Implementation Showing Redundancy

spellcheck. That functionality was instead buried inside of other methods, and/or poorly named.

Although the system did not show functionality well, it showed unexpected potential in identification of redundancy. This is best illustrated in Figure 6.5. This image shows an overly complex implementation of a function which does auto-correct functionality of misspelled words through simple string manipulation. In it you can clearly see repetition of structure across different branches, showing opportunities for refactoring.

Viewing the data as a whole we can answer our original question in the positive. Professors are able to use the visualization to quantify the design of student programs. This analysis can be completed in roughly equal time to viewing the source code directly. It is clear however that the tool is not well suited for identifying the functionality of individual methods beyond that which their name implies.
Chapter 7

CONCLUSION

This thesis presents a program visualization system which creates a graph of all possible paths through a Java program. It is able to show both the static structure present in the code as well as the dynamic nature of a specific execution. This visualization is intended for use by instructors and students in introductory programming courses.

Initial evaluation of this system has shown potential for usage as a tool for student learning. We find that presented within a slide show students claim the visualization is helpful, though we fail to find any correlation between students perceived effectiveness of the tool and their comprehension of the material. We identify that a more useful approach may be teaching students to use the tool throughout their learning, and allowing them hands-on interaction. Further research is still required to determine the extent of usefulness, and the optimal usages. Although in general professors were able to come to a decision on the quality of a program they did provide feedback as to ways to convey additional information. It was suggested that viewing the visualizations alongside the source code, including method purpose statements, and the ability to compare graphs against each other would improve the usability of the tool. It seems that comparison of different implementations of the same functionality may be one of the greatest strengths of this system.

This system contains many opportunities for future work in terms of both experimentation and development. Further areas of research have already been discussed in Section 6.1 so this discussion will focus on further development. The system in its current state supports a sub-set of the full Java language. Full lan-
guage support would be ideal, and is achievable by adding proper helper methods to all models. Based on the conclusion above that this system excels at comparing implementations, it would be ideal to have a way to feed multiple programs in and visualize them concurrently. Additionally the visualization portion of the thesis is language independent, meaning that support for additional languages could be added through additional parser modules. These new parsers would have to implement a similar class structure to that provided by PLYJ however, or a common interface could be developed. Finally improvements could be made to the visual aesthetic of the visualization. One of the best places to start would be to develop a smooth transition when entering and exiting methods during animation. Many student reviews commented that the jumpiness of the animation was jarring and made it difficult to follow.


APPENDICES
### Appendix A

#### 102 ANIMATION DATA

Data gathered from 102 students when presented with visualizations including animation. Data from students who included a ranking.

<table>
<thead>
<tr>
<th>BST Benefits</th>
<th>BST Search</th>
<th>Efficiency</th>
<th>Rank</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>Yes. Some of the animations went too fast for narration to really be effective. That tree at the end with 190,000 nodes was really impressive and cool to see</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>It would have been more helpful if it was slower and each box was explained a little bit more before hand, potential for 10/10</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>It did but it went a bit fast to be able to trace it</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>The animation was unclear to me. I understood that a search was happening but could not relate it to how the spell check would work</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>They were a bit hard to see on the projector</td>
</tr>
</tbody>
</table>
I thought the video could use a little more explanation as to why it kept jumping back to start (or else a clearer resolution, I couldn’t read what the boxes said). It took me a minute to figure out what it was doing, but after it made my understanding clearer.

I didn’t understand what it had to do with spell check, but it helped clarify what was going on in the BST.

Wasn’t all that helpful was more confusing at times, but with more explanation the visualization could be very helpful.

Pretty good, just that I could’ve used even more visuals.

Yes it helped us understand how recursion plays a big factor in BSTs.
<table>
<thead>
<tr>
<th>Score</th>
<th>Rating</th>
<th>Type</th>
<th>Feedback</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Ummm...the visualization was not the most clear for me. It was never pointed out that the bottom right showed the stack/level of recursion we were at which would have been helpful I think.

The visualization was a bit hard to follow, maybe having the tree it was building on the side would help demonstrate the thought processes behind the decision making; else it just looks like it’s taking random paths.

I originally found it hard to follow but after being exposed to it a few times it was a very helpful tool that made BST’s easier to understand.
The videos were a little bit hard to follow; maybe if you included it side by side with going through an image of a tree.

It did help but could be clarified a little. The video was very jumpy and hard to follow.

Yes, I felt as though the visualization helped explain the concept of recursion very well.

It did, I didn’t know the benefits and structure of trees and how they can be useful.

Yes, but it was a bit confusing on what each function was doing.

The animation seemed like it would be helpful but it went very fast. In this presentation I did not see at all how it worked.
It allowed us to see the control flow and logical flow of the algorithm in a simple manner.

Yes, due to the structured nature of the tree and the use of recursion, graphics is very helpful.

Data from students who did not enter a ranking. The ranking was determined based on their comments.

<table>
<thead>
<tr>
<th>BST Benefits</th>
<th>BST Search</th>
<th>Efficiency</th>
<th>Rank</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>The visualization did not really help me, I thought it was kind of simple and not really that informative. The person explaining trees with words were more helpful</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>Yes, given that I had no clue what these were until now, it helped</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>The tree diagrams helped more than the animation illustrating recursion</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>Yes, easier to visualize</td>
</tr>
</tbody>
</table>
Not really, would have been more helpful to pause the visualization and explain exactly what it was doing as well as make the font bigger to see the tree.

Not really, visualization went too fast and was too small to see.

Yes it did. It helped me visualize the logic behind it.

Yes It was very helpful and allowed me to follow along.

Yes it was helpful as it provided a clear image of how tree works.

Yes the visuals helped to make the concept easier to understand.

Yes, helped a lot as a visual learner.

The videos made it confusing because they were rushed through and not explained very well.
<table>
<thead>
<tr>
<th>Rating</th>
<th>Grade</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
The visualization were great for recursive functions, however it might be beneficial to step through each of the program steps individually, rather than a video which does it automatically.

It helped but it went a little fast.

Yes it helped reinforce the concept.

Yes, because it shows the logic in a simple manner.

Yes having a visual helps especially since we are new to recursion it helps to see the efficiency instead of just using an nmo??

Yes. The video moved quickly, but the presenters comments as the video played helped. It also helped demonstrate the many test cases.
The animation went too fast. Try explaining functionality with a more step by step approach.

Yes the tree demo was pretty helpful and easy to understand.

Yes seeing a project is always helpful.

Yes it helped me to see how a BST can be used and how the tree would be traversed.

Yes it allowed me to see what the code could have been used for.

Yes it helped illustrate how the concept worked as was very helpful.

Yes they helped but not as much as the example with numbers in trees.

Yes it’s easier to conceptually grasp BST’s with the visualization.
Yes, very much so. It’s difficult to conceptualize such a strange concept, but clear visuals makes it possible.
Appendix B

102 STATIC DATA

Data gathered from 102 students when presented with visualizations without animation. Data from students who included a ranking.

<table>
<thead>
<tr>
<th>BST Benefits</th>
<th>BST Search</th>
<th>Efficiency</th>
<th>Rank</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>Yes, it was very helpful to see a visual. I saw the logic in play</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>The illustration helps explain the tree so you don’t get lost</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>It did, in terms of computer science though you need something to go with visualization, like the code, but in a sense that could also be somewhat visualization. In computer science it can definitely be very helpful but you also need to be more careful.</td>
</tr>
</tbody>
</table>
Seeing the visualization helped understand the material b/c it was the first time learning it and it made it easier to comprehend.

I liked how you went back to the visualization after the code but it wasn't the most helpful thing in the world, it seemed kind of basic.

Yes it did help me understand BST's better. It was easy to learn the concept and grasp it better than I would have if the information was just spoken to me.

Make the visualization more clear with arrows to make the flow of the program more obvious.

Yes, helps show what is happening and the different paths that can be taken. Didn't really understand why there were multiple compares on the same path.
<table>
<thead>
<tr>
<th>Score</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Yes helped immensely. I’ve taken discrete structures so BST’s weren’t new to me, but helped in understanding</td>
</tr>
<tr>
<td>6.5</td>
<td>Yes allowed to see program flow. A little confusing with code explanation</td>
</tr>
<tr>
<td>7</td>
<td>A little, I still wasn’t sure at the end</td>
</tr>
<tr>
<td>8</td>
<td>Yes the tree diagram with nodes helped visualize how BST worked</td>
</tr>
<tr>
<td>7</td>
<td>Yes it gives a better representation of traversal through a tree</td>
</tr>
<tr>
<td>3.75</td>
<td>The actual trees were helpful, but I thought the visualization with start(); insert(); was confusing</td>
</tr>
<tr>
<td>10</td>
<td>Yes, before I didn’t understand what a tree was, but the visualization depicted it well and how to sort through it</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
It helped, but I may have been able to understand without it.

Yes helped retain key points in my head. Also gave me a better understanding of the topic.

Yes, seeing how much of the tree can be cutoff w/ one comparison helped show its usefulness. Visualization is quite helpful.

Yes, interesting and engaging helped map concept.

The visualization helps me understand what the methods to create a BST do.

Yes, it showed the process behind the BST.

Yes, the branches and possible paths laid out visually made it much clearer.

Yes, I can actually see what it looks like.

The spell check program was a 6. I feel there could be a more practical example.
It was definitely more helpful to see a visualization than not having it, but the ones specifically in this mini-lecture were a little harder to understand at first.

Frankly, you really wanted it to help but I don’t feel like it was explained well enough. The lecture overall was helpful and I learned about BST but the visualization did not make any sense to me. It felt like it was just thrown into the presentation hoping it would do something magical. I honestly think that visualization may help, but explain them better. Walk us through them like you did the tree. Maybe even construct them with us as we add the logic.

Yes it makes it intuitively obvious.
The visualizations helped a lot when they were expanded upon with the code. For example the insert visualization when you lecture on insert you went into a bit more detail as to how the program runs through the visualization. The second visualization wasn't as helpful because the lecture didn't back it up. For me the helpfulness of a visualization is proportional to how the lecture interacts with the visualization.

It certainly was helpful, but not invaluable. It gave a good "birds eye view" of the code we were subsequently shown allowing the code to be understood more easily. But the code itself was the most useful element.
Yes the diagonol format is a bit harder to read but might be easier on eyes than a linear if the structure.

I found that the visualization of the spell check helped me to understand how to organize and order data better.

It helped in a few hases; however it seemed only useful for highlight the outliers.

Yes, for me trees are a concept that should be explained using a visual format. Also it made everything easier to trace.

Visualization helped seeing possible paths from start to finish helps visualize what the options for the program are.

Cool to see branches of the function.
The visualization of the spell check program was fairly confusing. I need some form of direction (arrows, etc).

I like the idea of the visuals but they can be a little improved. Perhaps clearer with sidenotes or something.

Data from students who did not enter a ranking. The ranking was determined based on their comments.

<table>
<thead>
<tr>
<th>BST Benefits</th>
<th>BST Search</th>
<th>Efficiency</th>
<th>Rank</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>Yes the examples you showed us showing the BST in action helped, but the images of the code and the picture with black background and 4 lines did not really help me.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>Yes very much so for the tree explanation. The path diagram(with 4 route) wasn’t as clear to me, I didn’t quite understand what each method did exactly</td>
</tr>
</tbody>
</table>
Yes because it allowed me to see the flow of where the info is being carried to in each function.

Yes a little bit. The idea of seeing the different paths was nice but without labels on the junctions I couldn't tell what it did until I saw the code.

Yes it helped me understand how the program would run step-by-step. It also helped me understand the different paths the program could take.

Yes it shows different paths the program follows, like when it can’t find the word, the wrong path, and find the correct words.

The tree of all the 190,000 was amazing! Yes, honestly, was so excited by it.
Yes and no, it definitely helped to visualize the different paths through each method and the action resulting from that path.

I guess they were not explained very well I thought.

Yes it helped me understand how to sort through a BST.

I found myself pathing more attention to the slides where as listening first and trying to understand then comparing my visualization to the actual image.

Yes, recursion is a lot easier to understand when explained with this tree diagram.

Somewhat, I understood what should happen but felt that the other examples were clarifying enough.

I was a little confused by the visualization with the black background, but I liked all the other ones.
It was better than just code, but make visualization more explicit, not just function names. Tree diagram/flow helped significantly.

Yes it demonstrated the possible paths of the code (through if statements, for example).

Yes it list the steps in an organized manner.

The steps weren’t clearly explained enough, the explanation on actual trees were more useful to understand the concepts.

Yes, didn’t understand the node part in linked list.

It was helpful, but the visualization could have been better explained.
Data gathered from 103 students when presented with visualizations including animation.

<table>
<thead>
<tr>
<th>Insert</th>
<th>Find</th>
<th>Rank</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>On the right track but small blurry boxes won't make sense to people. Some people will make the connection to the structure you showed before the video. The people who don't understand BST's will just be lost</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>6</td>
<td>Yes and no, since the presentation was really quick and the video was not explained enough</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>5</td>
<td>No</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>3</td>
<td>I couldn't read the small words</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>6</td>
<td>It helped a little I was just a little confused about what was going on with the stack part</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>I have enough trouble conceptualizing the structure of a BST. Now you are overlaying a completely different kind of diagram that I have to try to understand - what do the shapes mean, what do the lines mean, what do the colors mean, what do the animated depicts represent? Then I have to try to relate that to BST - Oops, it isn't about BST, it's about the code for a BST which is yet another abstraction. So trying to relate the code, the BST and the visualization is just too overwhelming</td>
</tr>
</tbody>
</table>
Yes it helped. The insert tree was a bit more confusing than the search. On insert I wasn’t clear on what each path meant. On search you clarified what each path meant so maybe do that for insert.

To some degree. It was easy to get lost (ie if you miss a second the rest doesn’t make sense). Perhaps this would be mitigated if the parameters / stack were large enough to see.

No, The visualization did not make sense to me at all. Looking at the nodes of a tree is more clear.

I felt like it would’ve been more helpful if the video had been a bit larger, but it seemed useful.

The visualization was hard to follow but I’m pretty slow about most things. If I had more time with it, I’d give it an 8. But I didn’t get it so I give it a 5.

It was too abstract to understand how to do (the insert question) or I wasn’t aware enough. But if I had a better introduction I think it would.

Can’t read what’s on the board.

I think it’s a good idea but your system of visualization isn’t very clear. It may just need better labeling so the viewer understands what each path means (because it’s difficult to remember).
Cool looking, but not really sure what it represented completely. The videos were honestly a little hard to follow because there were a lot of different actions happening. I like the idea but maybe the video could explain better what each action meant.

The visualization was a little confusion but it did help me understand BST's a bit.

I felt as though the videos were hard to follow. Perhaps a little more instruction on how they work (such as a worked out example before we try) would be better. At the moment I was unable to understand them enough to do (the question on insert).

I didn't understand the first video because before starting it I didn't know the pathways. But after explaining it better I understood it for the second one.
Appendix D

INSTRUCTOR DATA

Data gathered from instructors reviewing 4 BST programs. Instructor one, ranking s2, s4, s3, s1.

<table>
<thead>
<tr>
<th>Identification</th>
<th>Surprising Methods</th>
<th>time</th>
<th>enough info?</th>
</tr>
</thead>
<tbody>
<tr>
<td>I think so - my confidence level is 75 percent with the amount of time I am spending</td>
<td>Why is s1’s main’s so branch-y and long?</td>
<td>I think 5 minutes – I think that is slower, but this was the first one I looked at</td>
<td>I want to be able to compare programs against one another!!</td>
</tr>
<tr>
<td>Mostly - I like this one better, but I didn’t find the substring comparisons?? confused about that part - not I am only spending a limited time, so I am sure I could figure it out...</td>
<td>It takes a bit of an adjustment to read parts like this (recursive branching search) ie, what is the difference in those “compare()” calls - it makes me wish I had more information.</td>
<td>about 4 min to look and not write anything down - I still want to look at code...</td>
<td>Much cleaner main but I am still finding myself confused and wanting to read the code (but that is my ‘norm’ so of course that is what I want - I wish there was a side screen with the code in it!!)</td>
</tr>
<tr>
<td>sort of - confidence 60percent with the amount of time I am spending</td>
<td>spell check is insane</td>
<td>8 min</td>
<td>I do not like this one that much - but is it your layout or their code??</td>
</tr>
</tbody>
</table>

82
yes - most confident about this one (but it is also the last one I looked at) 90 percent with the amount of time I am spending

These kinds of layouts are really hard to read. I am wondering if curves will help but as I said above, I think coloring will also help trace branches.

5 minutes

Instructor 2, ranking s2, s4, s1, s3.

<table>
<thead>
<tr>
<th>Identification</th>
<th>Surprising Methods</th>
<th>time</th>
<th>enough info?</th>
</tr>
</thead>
<tbody>
<tr>
<td>I think so, but quite possibly only because they are in an ordering matching this question.</td>
<td>insert apparently loops. main is too complicated.</td>
<td>It did not take long to analyze this program (5 minutes?). I suspect it is about the same amount of time, but being new to this too, I am not sure that I understand the program as well as if I were looking at the source.</td>
<td>I feel there is enough information to make qualitative judgment.</td>
</tr>
</tbody>
</table>
Inserting and searching are clear from the names. The spell-checking aspect itself is not obvious.

No. A couple of minutes. For this program using the tool may be a bit faster than reading the source, because the structure is as expected. I expect the code is relatively clear though.

Really only from the method names spellCheck is horrific. Many levels of nested loops?

Yes, I feel like I had enough information. Just a couple of minutes.

Not as well as some of the others. This issue stems from the source, not the tool. p-updateTallies was initially surprising in terms of the visualization, but a bit less so after viewing the graph for a bit. This one took longer because of p-updateTallies. I suspect it would have been quicker to look at the code.
Instructor 3, ranking no idea. Maybe the second one was the best?

<table>
<thead>
<tr>
<th>Identification</th>
<th>Surprising Methods</th>
<th>time</th>
<th>enough info?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not sure, no</td>
<td>Surprising? Well, I have a hard time understanding any of them some methods (e.g. <code>getWord</code>)</td>
<td>Well, this is just the first I was missing program, but the purpose just have the trivial picture (<code>Start()</code>, <code>Return()</code>). Can't tell what these do. Lots more branches from <code>Start()</code>.</td>
<td>No I feel like Im giving up statement that after 10 minutes, and I feel told me what like I haven't each method really analyzed did. The naked the program at name of the all. This seems method is not much slower enough for me.</td>
</tr>
</tbody>
</table>
A little better than the first. Probably most of this is that I've now seen two, and I'm picking up on the similarities and differences. Generally speaking, though, no, I can't.

I still don't really feel surprised, because I don't really understand any of them.

No. See earlier comments.

This one has a call named spellcheck, which makes things clearer. In general, though, I'm starting to think that I would probably hate the source code for these submissions; so, in some sense, the things that would make the pictures more useful are also things that would make the source code more readable.

Ther's still tons of branching that I don't understand. Why does toCharArray() branch? No. In general, though, it seems clear to me that Java at least the way we're teaching it to students is way, way too low-level.

No, still no all of them are surprising? Cant make sense of any of them.

Spent about five minutes on this one.