LogicVis Project Proposal

CSE 403 Software Engineering
Winter 2019

Team LogV

Candice Miao
Leo Gao
Jed Chen
Glenn Zhang
Andrew Liu

Motivations and Objectives

Coding can be difficult to learn. As students, we struggled with many of the fundamental coding concepts. Looking back at these experiences, we wish to build a tool that could make learning how to code much easier as our project. An effective method we found put code into the form of a control structure diagram, which contains graphs with the flows that represent logic. This method helps students understand code by connecting to the students' prior knowledge of aspiring students well[1]. Though graphics tools exist, we noticed a distinct trend amongst all the tools we checked, like JavaVisualizer, and InfoVis Toolkit--they do not handle recursive cases well ([9, 10]). We found this to be quite odd because we remembered that recursion was one of the most difficult topics to master, yet it is left out in most tools that are built to help beginners. Through research, we found that the major difficulties students have with recursion are ([2, 3]):

1. Students not utilizing functional abstractions

The purpose of abstractions is so that we can work with something without knowing how it works. For example, there are two forms of knowledge associated with learning how to drive a car: (1) knowing how to operate the car (2) knowing how the car operates. Most drivers do not actually know in detail how cars work, but they can still utilize cars in those ways. In the context of recursion, students have similar troubles: they focus on *how* the recursive step works instead of *what* the recursive step does. If we can help them accept that the recursive step just does what the function itself does, the difficulty of understanding the recursive part of the function diminishes.

2. Lack of a proper methodology to represent a recursive solution

Many students that are new to recursion do not understand how to formulate a solution that uses recursion. They begin to write recursive solutions when prompted but often cannot imagine the solution they drafted in a concrete way. For example, they will think of including a base case and a recursive call, but cannot to connect these components together to form a coherent solution.

The goal of our project is to build a tool for Java that can help new programmers learn recursion by visualizing the flow behind programs using arrows to connect subsections of the code. This tool can translate a chunk of recursive code to a flowchart so that it is easier for people to grasp the recursive algorithm and build their own recursive programs.

User Study

```java
public static void mystery(int x) {
    mystery(x, 2);
}

public static void mystery(int x, int n) {
    if (n == x) {
        System.out.println(x);
    } else if (x % n == 0) {
        System.out.print(n + " ");
        mystery(x / n, n);
    } else {
        mystery(x, n + 1);
    }
}
```

We conducted a user study to confirm that our product will assist with learning recursion. The code for user study is to the left (prints the prime factorization of x). The code is recursive and somewhat complicated. We have developed a paper-prototype for the code here and used it to test the

effectiveness of our ideas (See User Manual for UI reference).

Choice of Method: The tasks given for the users are to compute the results of functions. We split the study into two similar parts: one part without our tool and one part with it. In each part, we will ask similar questions about the program's output and gauge the participants' understanding of the given piece of code. This way, we hope to find whether our tool helps students follow recursion better. We pick this method to mimic methods often used in beginner programming classes while being practical while using a paper prototype.

Set-up: We performed our study on a college student with a background of AP Computer Science, which means they have learned recursion, but not extensively and not recently. We let the subject look at two methods, hiding both their names (i.e. we labeled them as mystery). The reason we hid the names was to reduce the chances that the participant used their math knowledge to understand the code rather than their knowledge about recursion. The first method we showed returned the nth number of the Fibonacci sequence. The method printed out a number's prime factorization and included a corresponding paper prototype on the expected product. For each method, we asked the following questions:
-   What does factors(20)/fibonacci(5) output? (mystery(20)/mystery(5), to the participant)
-   What does this method do?
-   How does this method work?

Performance: The subject was able to identify the output and functionality for both programs without outside assistance. However, the subject spent significantly longer on the second prompt, completely missing the base case (went to fibonacci(-1)), and drew a representation of the method calls while computing it. For factors(20), the subject was able to quickly find the output but had trouble explaining why n was incremented by 1 in the second recursive case.

Feedback: The user responded that, in comparison, our charts did not help understand a specific line of code, but the flowcharts make the execution flow much easier to follow: the flow chart indicates how each recursive step progresses to the next one and helps the user see where each step executes. Furthermore, the user found that there was too much displayed at once; having a reduced flow chart based on which statements execute may make our chart easier to follow.

Analysis and Future Testing: From the fact that the subject made a separate representation for the first method, we found that the prototype we created successfully served as a separate way of understanding in itself. In addition to being able to characterize the program faster, the subject also had no trouble following the execution of code, likely because all the steps are expanded out for the subject. On the other hand, our prototype struggled to give meaning to why each and every line of the code existed.

Further testing about user debugging and developing will be necessary to examine other aspects of this project.

Our Approach

LogicVis is a program that takes an input of a Java recursive function code and its parameters and outputs a logic graph of control flow. It reads in the Java code line by line and turns each line into a node representation with the shape indicating the type of command. LogicVis graphs contain different representations for basic Computer Science concepts, including if/else, loops, recursion, etc. This tool will output the logic graph with input and output parameters tracking for each iteration. As previously mentioned, students generally have two problems with recursion, and we seek to help students by solving these problems. The below is the our graph representation of a simple recursive program.
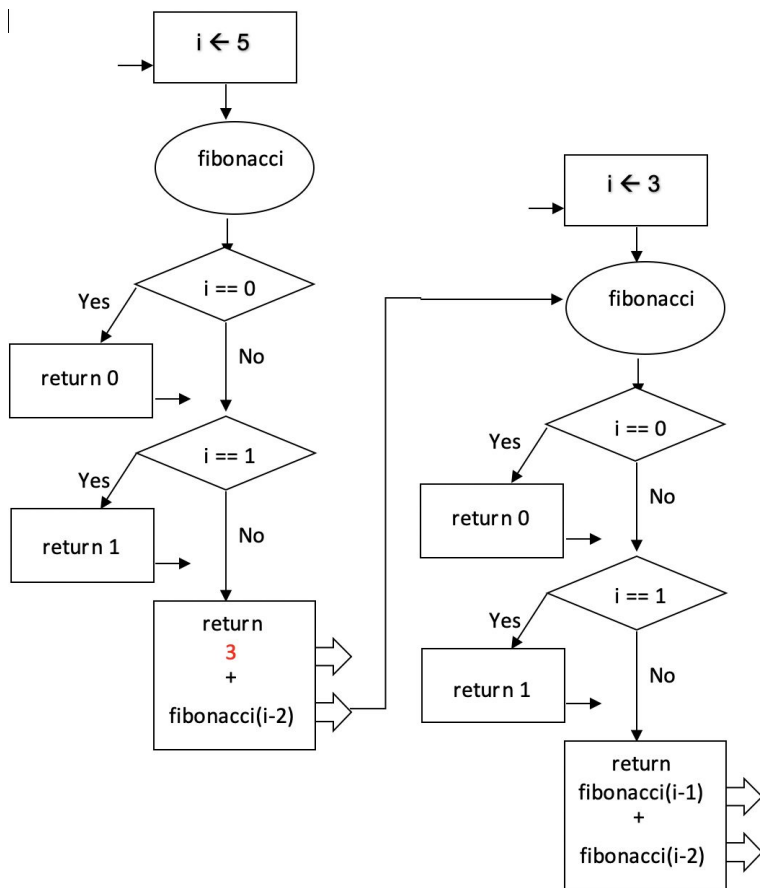
1. Not utilizing functional abstractions: Our approach to this problem is to attempt to unpack the abstractions that is packed in the recursive function. Here we are building our graph to represent those recursive steps concretely. Our take on the visualization of a recursive function will exhaustively list iterations of recursive calls. If the students can see the results of the code by expanding the graphs, they have the potential to acknowledge and trust that the recursive call will perform its intended function. This does not directly teach students the concept and usage of functional abstractions, but it makes recursion more easily recognizable by the students, and by that familiarize student with abstractions.

2. Lack of a proper methodology to represent a recursive solution

This graph also aims to be the alternate methodology for students to represent the codes that they write. Tracing the steps in a recursive program provides students with a mechanical means to follow the recursive algorithms. In our implementation, we will have made it clear that another function frame has been made every time we descend down the stack. Another utility this graph seeks to focus is on helping to debug their own code. Since students are unfamiliar with drafting solutions to a program, intended use of this program is to represent what the students write themselves into concrete graphs. This way, students will understand the effects

of every line they write, as opposed to putting down lines they imagine that are supposed to be in there.

The recursion will be handled by adding options to expand recursive calls when they appear in the original function graph. They will be generated on demand and allowed a complete expansion in concrete cases. Our minimum viable product will be designed such that a strict number of recursion iterations will be enforced with a working GUI. For the final product, we are aiming for removing the strict limit on the number of recursion iterations but rather utilize the lazy evaluation approach based on scrolling in the UI.

If we have time in the end, we want to potentially look at implement this product as a plugin to Eclipse or supporting modification on the graph generated being reflected in the code. Since we are not planning on basing our product off of any existing products or purchasing outside resource, our cost plan, in this case, is 0. We have allocated 8 weeks based on our designed functionality of the final product. However, we did also account for estimation errors by designing a Minimum Viable Product that will be implemented first and a stretch goal in case we have time.

Interfaces
(Refer to the user manual)

Implementation

Architecture
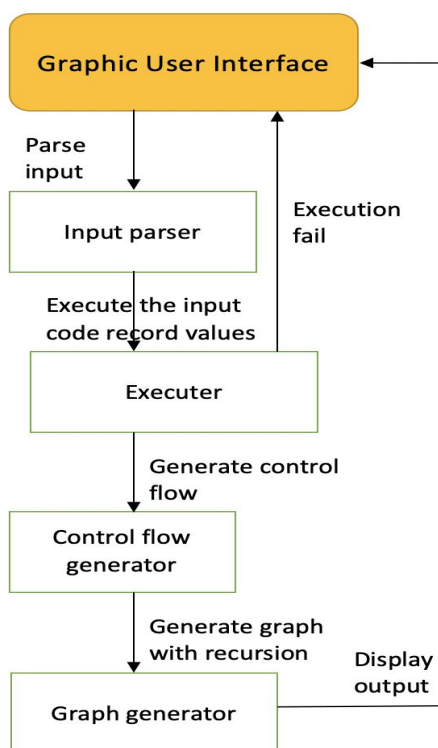
The architecture is represented by the diagram to the left. The boxes represent components of the architecture and the arrows represent the flow of the data from one component the another.

1. Graphic User Interface

This layer is the front end part of our project. It will let user type input code and get the parameter value and display the graph.

2. Input parser

This layer will read the input from the front end software, get the input value, preprocess it and send the code to the executor.

3. Executor

This layer will execute the input code. If the execution failed, send an error message to front end and display error message. If succeeded, the executor will send the code along with the

information such as recursion depth and parameter value of each depth to the control flow generator.

4. Control flow generator

This layer will generate the basic control flow code of each depth of recursion call with corresponding input value and send the control flow graph information to the graph generator. This layer does not generate any graphs, it only generates the data how the graph is supposed to be constructed.

5. Graph generator

This layer will visualize the graph and implement the clickable features to the graph, then send it back to GUI and display the graph.

Technologies
We have chosen to develop this software in Java since we are most familiar with the Java language and its libraries.

Graphic User Interface:
We plan to use the JavaFX library, which provides a clean graphical UI that works as a standalone. The older libraries such as Swing and AWT do not provide as much functionality whereas others like Pivot are used as RIA.

Graph Generator Algorithm:
We plan to build this from scratch. We want to be able to make a parsing and graph generation software that fits our Graphics User Interface, so although we may reference existing flowchart generators, we would have to rewrite most of it if we wanted to use one.

Limits and Risks

We run risks that our approach is not received well by the intended audience. The reasons for this may be having unintuitive navigation, unclear information, or distracting factors in the final product. To mitigate the harmful impact of such risks, we plan on conducting user research after the product has been built as well as having researched using paper-prototyping already.

We also have taken risks in deciding the scope of our project. We have chosen to make this software specifically for Java, which means that building a graph for any other language is not possible. We based this on the idea that most students that learn computer science start with Java. Another risk that we will be taking is our decision to not handle Java built-in library functions. Displaying these would introduce a whole new set of problems, such as additional clutter and having to implement it in the first place. So, we currently have chosen to treat imported functions like a generic line, but we may change this depending on the feedback we receive when doing research. Overall, we hope to build our software for a large enough scope of statements to help the user while not also overwhelming them. If we have successfully

chosen the correct scope for our product, we will have made the version of our software that will help most people.

Schedule

We plan to split major sections of the projects into work done for each week. For each week, we will begin planning the specifics of what each goal entails. After reaching the goal of implementation for every week, we start testing the programs to ensure its correctness. This means that testing will be involved in every week, so we chose to omit it in the schedule below.

Time	Goals
Week 4	- Project proposal and Planning - Write the initial specification - Design visual components of graphs
Week 5	- Finalize the implementation directions and formal proposal document - Assign roles and methods of communication
Week 6	- Design the GUI - Write the user manual - Implement function parsing: Identify the different type of lines in Java - Implement recursion parsing: recognize keywords and represent the flow
Week 7	- Build a data structure to represent the code - Visualize the data structure into a readable graph - Evaluate the performance of the graph using testers
Week 8	- Implement improvements on the graphs if found necessary by testing - Complete the UI - Build the frame of the GUI
Week 9	- Finish the GUI - Research on preferred GUI components using user testing
Week 10	- Handle multiple functions - Link relations between existing functions
Week 11	- Final testing - Prepare for presentation

If we have time left over for extra development, we can consider implementing:
- Parsing class structures

- Handling a class with recursive methods
- Making it a plugin for Eclipse
- Supporting modification on generated graphs being reflected in the code

A good "midterm exam" for this project would be checking to ensure that the base visualization of the graph we are building matches our expectations at week 7. A good "final exam" for this project would be looking at the final product to see whether we are handling multiple functions correctly at the end of week 11. At that point, we would also test everything else to see that they are robust and optimal rather than just its correctness according to the specification.

Current Work

As we mentioned before, there are many visualization tools online, but none of them handle recursion to the amount of detail we hoped for. For example, the Eclipse CFG Generator[4] was designed as an Eclipse plugin to translate a piece of code into a flowchart. The following graph is an example of how it works. However, the visualization it produces neither shows what each line of code is doing (each line of code are simplified to the category of the code or the name of the variable, like "expression" and "plaintext") nor it shows a clear logic flow within the code (it only visualizes the if/else condition and ignores all other method calls) which makes it impossible to visualize recursion. Also, the CFG Generator and many other tools[5][6] fall short in that they do not handle recursion any more than simply noting that the method was called. For example, Eclipse CFG Generator simplifies the function call to "expression" which tells no information about this function call. In other words, though it helps users understand a program's flow in a low level, users will still be unable to understand a recursive piece of code, as it provides no extra help on the abstraction of recursion under the call. In comparison, our approach gives visual representations of every single line of code within the recursive function and we track the parameters and return values of the recursive calls, giving the users of our program, especially students, a better sense of the current iteration of the recursion.
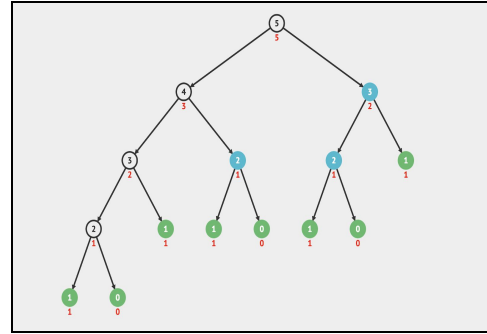

(Eclipse CFG Generator)
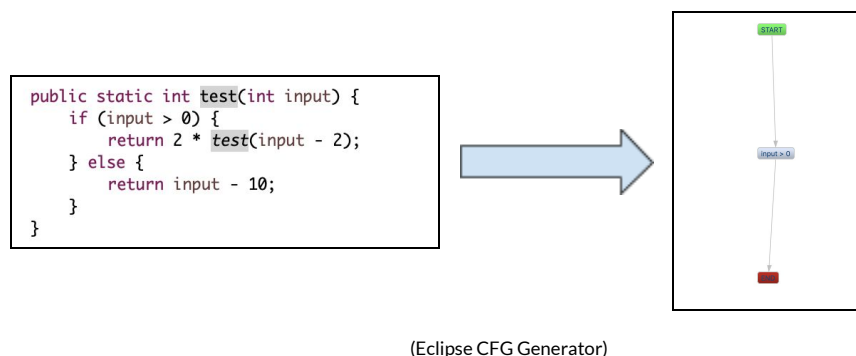
The best recursion trackers we found online were those like VisuAlgo[7]. VisuAlgo runs a program, such as GCD, and demonstrates a live example of the method call flow as a recursive tree. It lets you step one recursive call at a time and shows the parameter values and return values of each call. The following graph is an example.
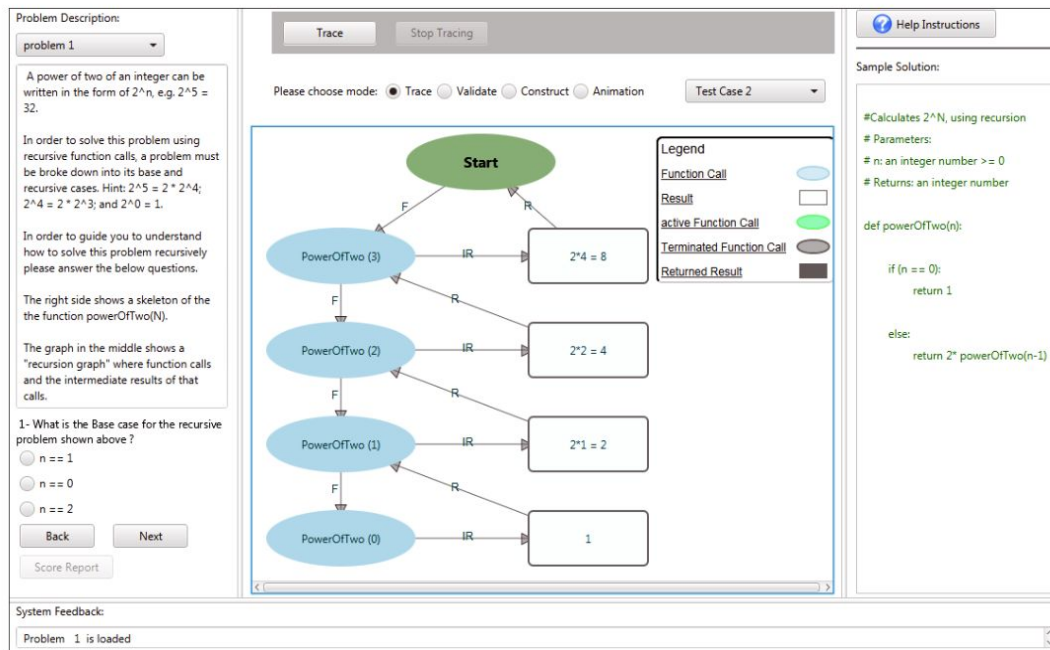
(VisuAlgo)

We are using a similar approach, stepping one recursive call at a time with the parameter and return value visualized. The difference is that VisuAlgo represents each recursive calls as a single node with no information on what happens inside the function. Students struggle to understand the recursive program because they have little to no idea about how the result was obtained. We have looked for many tools that track complicated recursive calls, but no such graphing tools exist. Our approach takes into account every line of code, making the user completely aware of the program execution without high level uses of abstractions. In addition, when students use LogicVis to examine their own code, they can contrast our outputs to grasp the impact of each specific line, which VisuAlgo cannot do.

Another approach on helping student learning recursion is ChiQat-Tutor System, a system that helps visualize certain predefined recursive functions[8]. It is helpful to get students to understand the given recursive function cases by giving giving students questions and tasks to do about given recursive functions, but it is similar to the previous examples in that they do not generate the details about the recursive call. Since this tool is not able to process non-preset programs, students may also have difficulties with reasoning about their own recursive code during the debugging process. Our approach plans to be able to generalize on all the recursive functions to provide specific graphs on recursive programs for students to compare.

(ChiQat Tutor)

There is one tool that helps code visualization extremely well, which is code2flow.[6] It takes text input and generates control flow based on it. It also visualizes function calls with details by having a box containing details of the function outside the main graph. The following graph is an example.
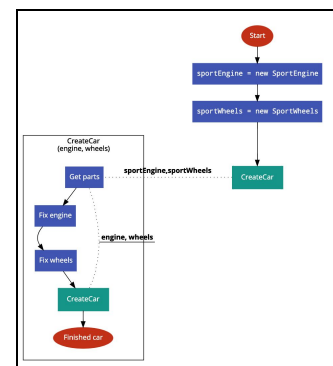


```
1  function CreateCar(engine, wheels) {
2      Get parts;
3      Fix engine;
4      Fix wheels;
5      call CreateCar(engine, wheels);
6      return Finished car;
7  }
8
9  Start;
10 `sportEngine = new SportEngine`;
11 `sportWheels = new SportWheels`;
12 call CreateCar(sportEngine
       ,sportWheels);
```

(code2flow)

Our approach plans to generate a control flow graph similar to this because it makes each step of the execution clear and easier to understand. However, this tool only gives a static graph with no parameters and return values indicated in the output. In the case of understanding recursion, parameters and return values helps students a lot on understanding the current fields of the execution and the result of each iteration. Our approach enables users to formulate their own code by demonstrating the details of their code based on customized inputs and stepping each iterations of the recursive call to have a better awareness of the overall iterations of the program execution.

Roles

Implementation Team: in charge of programming the software so that it runs.
- Andrew Liu
- Candice Miao
- Leo Gao

Evaluation Team: in charge of designing the product and testing its effectiveness.
- Glenn Zhang
- Jed Chen

Feedback

Rashmi: Figure 1 feels repetitive and doesn't convey any information. I would get rid of it and use Figure 2 instead. (About the User Manual)

Response: We think it is useful to convey to the users what they are supposed to see when the User Interface was just booted up. This helps keeping the users on the same page.

Michael: A way to stage your work could be to show the complete execution tree as phase 1 of your work; to enable clicking to show or hide parts of the execution tree as phase 2; and for phase 3 graying out any control flow graph nodes that are not executed.

Response: This is a good idea. After careful consideration, we decided that we will be following phase 1 and 2 and group them together. However, for phase 3, the implementation of it would be difficult for us due to our previous decisions in implementation, so we will not be following it.

Rashmi brought up that we should include a detailed description of the UI.

Response: Regarding this issue, Michael responded that we need not include it if the section is identical as in the manual, so we followed Michael's advice.

Citations

[1] Cross II, J.H. & Sheppard, S.V.. (1988). The control structure diagram. 274 - 278. Computers and Communications, 1988. Conference Proceedings., Seventh Annual International Phoenix Conference on. 10.1109/PCCC.1988.10084.

[2] Raja Sooriamurthi. Problems in Comprehending Recursion and Suggested Solutions. University of West Florida, 2001. Proceedings of the 6th annual conference on Innovation and technology in computer science education Pages 25-28.

[3] Tamarisk Scholtz, Ian Sanders. Mental Models of Recursion: Investigating Students' Understanding of Recursion. University of the Witwatersrand, 2010. Proceedings of the fifteenth annual conference on Innovation and technology in computer science education Pages 103-107.

[4] Alimucaj, A. (2009). Eclipse Control Flow Graph Generator. Retrieved January 31, 2019, from http://eclipsefcg.sourceforge.net/

[5] Code Visualization.
https://www.grammatech.com/products/code-visualization

[6] Code2Flow.
https://code2flow.com

[7] Halim, S. (2011). VisuAlgo.net/en. Retrieved February 4, 2019, from https://visualgo.net/en

[8] AlZoubi, Omar & Fossati, Davide & Di Eugenio, Barbara & Green, Nick & Alizadeh, Mehrdad & Harsley, Rachel. (2015). A Hybrid Model for Teaching Recursion. The 16th ACM Annual Conference on Information Technology Education, Chicago, USA. 10.1145/2808006.2808030.

[9] JavaVisualizer.
https://cscircles.cemc.uwaterloo.ca/java_visualize/

[10] InfoVis Toolkit.
http://ivtk.sourceforge.net/