LogicVis Project Proposal

CSE 403 Software Engineering
Winter 2019

Team LogV
Candice Miao
Leo Gao
Jed Chen
Glenn Zhang
Andrew Liu

Motivation

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. The method we found most effective is to be able to visualize code in charts since it connects with the prior knowledge of aspiring students well [0]. In accordance with that, many general purpose graphic tools that visualize of code already exist. On the other hand, we looked for flaws in such tools to see what more we could contribute. In this process, we noticed a distinct trend amongst all the tools we checked--they do not handle recursive cases. 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. Seeing this inspired us to build a graphical tool that can better help students learn programming, a tool that represents recursion as a more intuitive concept.

Objective

The goal of our project is to build a tool for Java that can help programmers visualize the logic behind recursion. To be more specific, we want a tool that can translate a chunk of recursion code to a logic graph so that it is easier for people to understand the recursive algorithm. If this tool is successfully built, programming students who have trouble with learning the concept of recursion will have an easier time. Furthermore, the tool ideally can also provide variable and flow tracking, which greatly benefits the debugging and testing process.

Current Work

The way many people go about understanding a piece of code is by reading line by line. However, when the function is large, reading it line by line is extremely time-consuming and frustrating. Many software solves this by abstracting from the actual code so that people will understand the code without actually having to look at the code. One such tool was developed to generate documentation by executing the program. [1](Sulír, Matúš & Porubän, Jaroslav. 2017) Their approach comes in handy for people to understand people's code as a whole by looking at the documentation. But the logic behind the code is still not understandable unless the actual code has been read, and this will be difficult for people who need to modify the code for other purposes like adding features and debugging. In terms of code-to-graph visualization, there are many helpful tools that take a snippet of code and generate a flowchart from it. For example, the Eclipse CFG Generator[2] was designed as an Eclipse plugin to do just that. However, it

and many other tools[3][4] falls short in that it does not handle recursion any more than simply noting that the method was called. In other words, though it helps users understand a program's flow, it does not give much assistance in understanding recursion. Another tool, VisuAlgo[5], runs a program, such as GCD, and demonstrates an example of the method call flow as a recursive tree. However, it stays simple (only works with integers, limiting the number of possible recursive algorithms) and does not go into detail on what happens within each call. Though it would help a user see the general flow of the program, it does not give enough information to understand why the values were returned. We have looked for many tools that track more complicated recursive calls, but no visualizations exist.

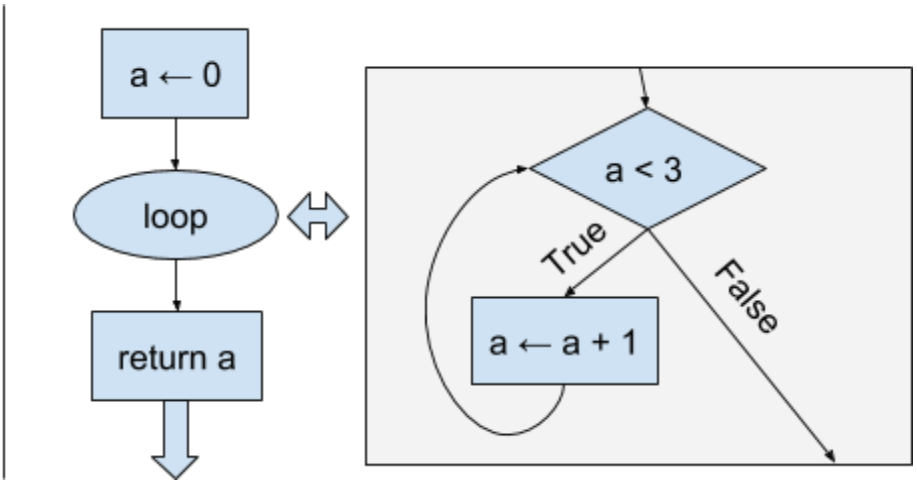
Our Approach

LogicVis is a program that takes an input of a piece of Java recursive function code and outputs a logic graph. 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. We will first implement a method that takes in a piece of Java code containing a recursive function and outputs a built graph object that contains the logic visualization with simple variable value tracking for the function as our Minimum Viable Product. 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 MVP will be designed such that a strict number of recursion iterations will be enforced. It will have a clear and easy to use Graphic User Interface as well. 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.

Example:

```
int a = 0;
while (a < 3) {
      a++;
}
return a;
```



Limits and Risks

Since we decided to focus on recursion, we must be able to understand all the reasons that beginners face problems with recursion. This may be different from our initial perceptions, so research into what helps and what doesn't will be important. For example, we have chosen to read the program through static analysis rather than dynamic analysis since we believe that generalizing the recursive call will be more helpful than showing an example recursive call. This could very well backfire on us, however; most of the recursive tools we found relied on the ability to show the recursion step-by-step in an example, which is much easier to do dynamically.

We have also 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.

Week 4 goal:
- Project proposal and Planning
- Write the initial specification
- Design visual components of graphs

Week 5 goal:
- Finalize the implementation directions and formal proposal document
- Assign roles and methods of communication
- Function parsing: Identify the different type of lines in Java

Week 6 goal:
- Design the GUI
- Recursion parsing: recognize keywords and represent the flow

Week 7 goal:
- 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 goal:
- Implement improvements on the graphs if found necessary by testing
- Build the frame of the GUI

Week 9 goal:
- Finish the GUI
- Research on prefered GUI components using testers

Week 10 goal:
- Handle multiple functions
- Link relations between existing functions

Week 11 goal:
- Final testing
- Prepare for presentation

Stretch goal:
- Parse class structures
- Handle multiple classes
- Make it a plugin for Eclipse
- Support 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.

Testing

As described above, we would be testing constantly as we start implementing each part of the project. These tests mostly check that the program behavior matches what we expected. They include verifying outputs, memory usage, time consumed, and how well it contributes to our overall purposes. Though the first three parts can be concretely measured with our written tests, the last part requires extensive experimentation on other people's opinions on our program. Our basic purpose to help programmers understand the code better through the use of graph visualization. To validate that our project achieves its goal, we will perform the following experiments:

1. Code vs. Graph: Put a regular 20 line-long program with recursion in code form and graph form. Then ask customers to compute the output of such programs giving them only 1 form versus giving them both forms. We would measure the correctness of the answers and the time taken by either side to check whether the graphs actually helps with understanding. The following is a sample graph. The points scored will be a system based on a linear combination of correctness and timing, which will be determined after we find out what the performances look like.

2. Graph vs. Graph: Put a variable length recursive program in code form and different graph forms. Then ask customers to compute the output of such

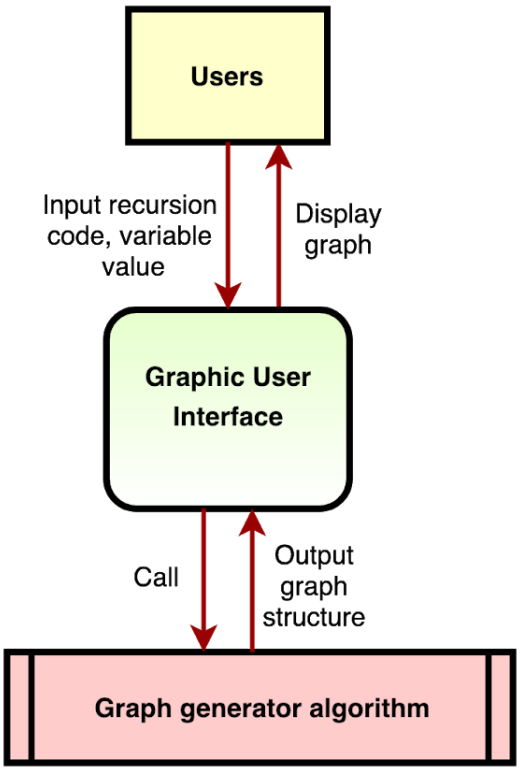
programs giving them the code and one of the graphs. We would measure how the customers performed using correctness and time spent to check which interface of our program is more intuitive and effective to use. The following is the other graph, where we will use a point system similar to the one used above.

Using these kinds of experiments, we can evaluate our approach and determine whether changes are needed. If these experiments show a significant result, we can then declare our project as a success.

Architecture

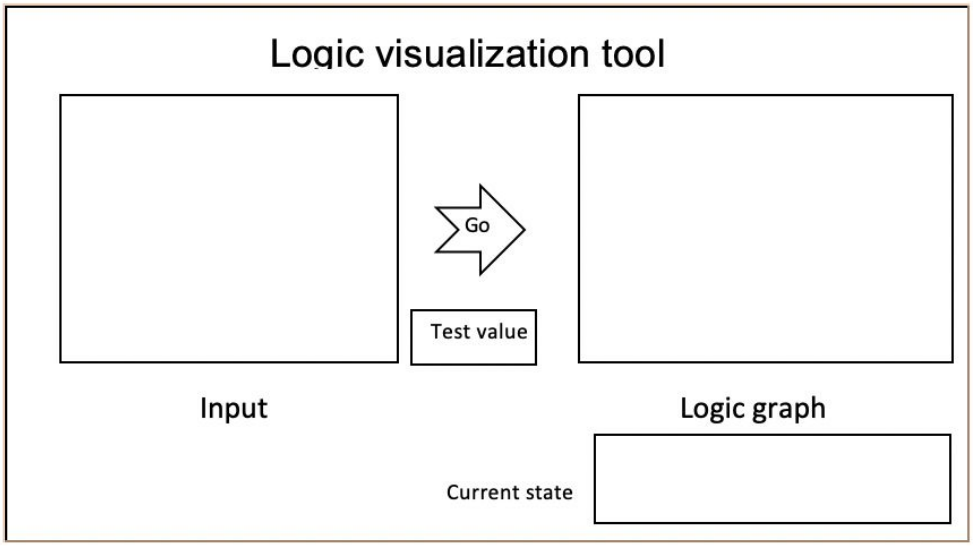
For this project, we decided to implement with three layers: the Users Layer, the Graphic User Interface Layer, and the Graph Generator Algorithm Layer. The Graphic User Interface is first displayed to the user with the option of inputting code, as the users passing in the code to be visualized and the values they want to test out, the Graphic User Interface maps each input to the corresponding functions and calls the Graph Generator Algorithm. The Graph Generator Algorithm will return a graph structure of the corresponding input and the Graphic User Interface will take the returned data and display this to the user in the output box.

When the users click on elements of the graph on the Graphic User Interface, the Graphic User Interface takes in the action and displays corresponding changes on the interface.



Interfaces

Graphical User Interface prototype:



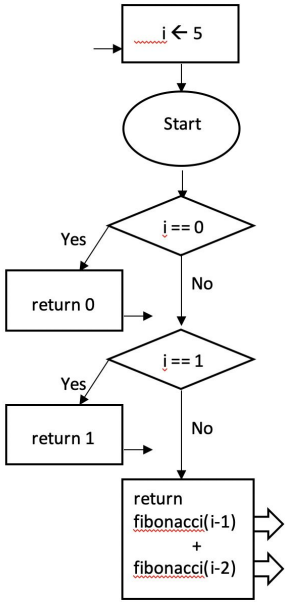
It consists of an input box in which users can type the recursive function that they want to translate, a Go button which will do the translation, a “Test value” button which will change to the test mode, a current state box which will show the variable values in the test mode and output box which will show the graph.

In the input box, users are able to type the input code. For example:

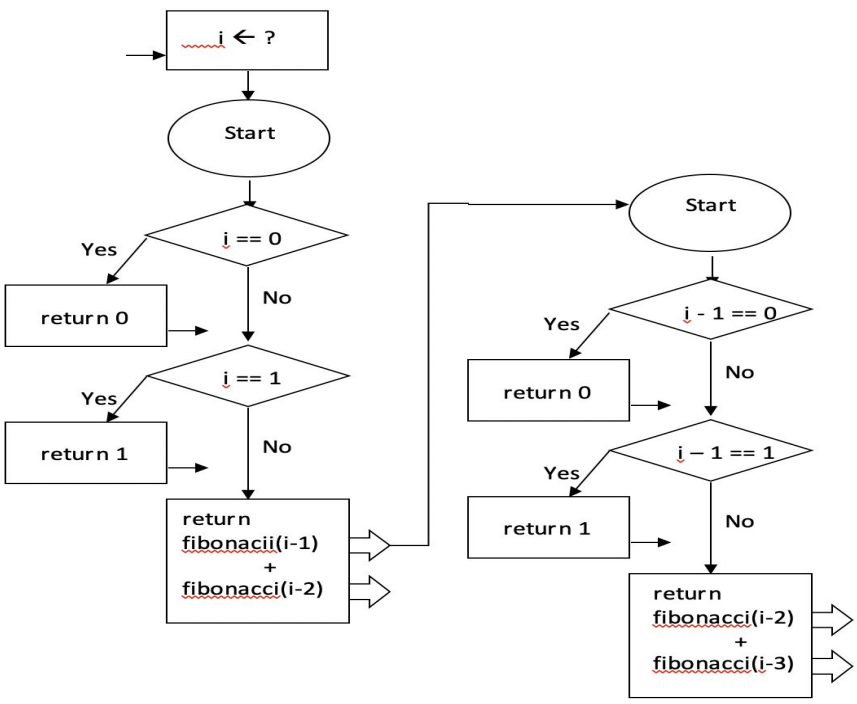
```
public int fibonacci (int i) {
        if (i == 0) return 0;
        if (i == 1) return 1;
        return fibonacci(i - 1) + fibonacci(i - 2);
}
```

After pressing the “Go” button the Logic graph box will show the logic graph where the parameter i is an indefinite value:

In this graph, both arrows in the last box can be clicked because they are calling itself in recursion.

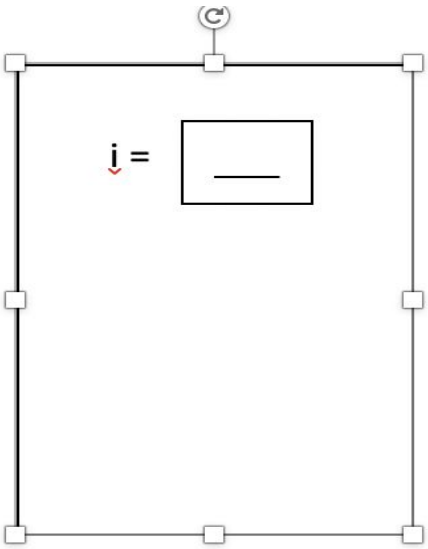


After clicking the arrow for fibonacci(i - 1) we get:

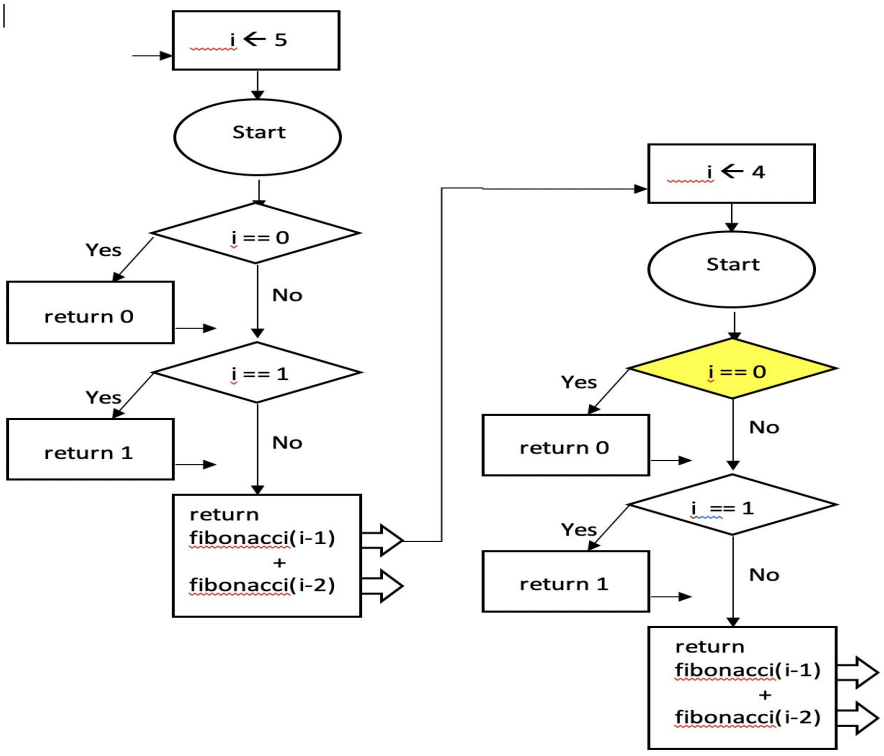


This shows an extended graph with a recursive function and in the recursive function, i will become i - 1.

In the main panel, we have a test value button. After clicking it, it will ask for the input value for i.



If we typed i = 5, the graph will be changed to test mode. In the test mode, just like a debugging tool, it can ask users to step forward, the block with yellow color will show the current step:



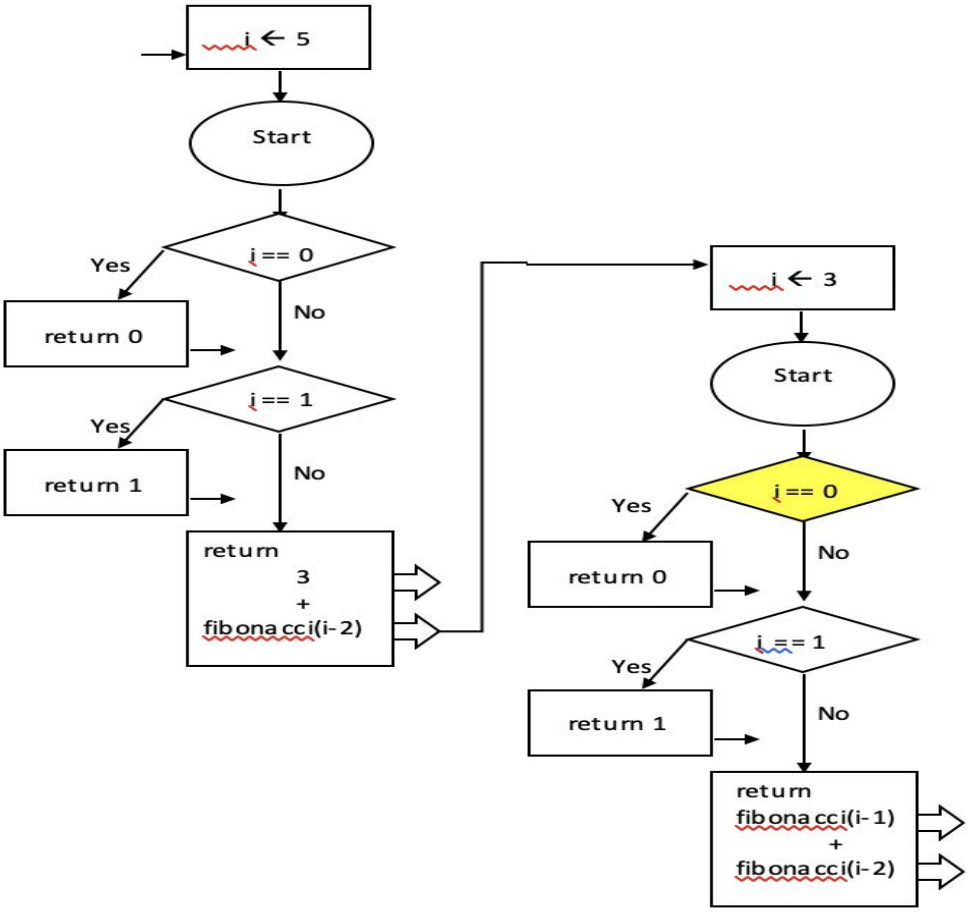
Other several differences between the test mode and indefinite value mode are:

(1) The recursive function call will have a value for i before the start block

(2) Instead of showing i - 1 like the previous graph, it will show i instead with i has value 4.

(3) In the main panel, the current state block will show the variable values for the current state. In the example above, when we are at the yellow step, the current state block will show i = 4

We move forward to the second recursive function:



(4) After finishing the first recursive function fibonacci(i-1), it will change the function to a returned value 3. In the example above, the current state block will show i = 3.

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 Graphic User Interface, so although we may reference existing flowchart generators, we would have to rewrite most of it if we wanted to use one.

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

Citations

[0] Rachel Gillett (2014). Why We're More Likely To Remember Content With Images and Videos (Infographic)

[1] Sulír, Matúš & Porubän, Jaroslav. (2017). Source Code Documentation Generation Using Program Execution. Information. 8. 148. 10.3390/info8040148.

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

[3] Grammatech's tool is not free and focuses on understanding programs at a larger scale (i.e. multiple classes and function calls between them).
Rfleming. (2015, December 30). Code Visualization. Retrieved January 29, 2019, from https://www.grammatech.com/products/code-visualization

[4] Code2Flow is an online code visualization tool that suffers from the same issue as the Eclipse CFG Generator in that it does not handle recursion in a clear way.
The simplest way to describe your flows. (n.d.). Retrieved February 3, 2019, from https://code2flow.com

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