Integrating a Graph Builder into Python Tutor

Diogo Soares
University of Minho, Braga Portugal

Maria João Varanda Pereira[1]
Research Centre in Digitalization and Intelligent Robotics,
Polythechnic Insitute of Bragança, Portugal

Pedro Rangel Henriques
University of Minho, Braga, Portugal

Abstract

Analysing unknown source code to comprehend it is quite hard and expensive task. Therefore, the Program Comprehension (PC) subject has always been an area of interest as it helps to realize how a program works by identifying the code that implements each functionality. This means being able to map the problem domain with the program domain. PC is a complex area, but its importance for programmers is so high that many approaches and tools were proposed along the last two decades. Program Animation is one of those approaches requiring specialized techniques.

For each programming language, there are already tools that enable us to execute a program step by step, visualize its execution path, observe the effect of each instruction on its data structures, and inspect the value of its variables at any point.

In the present context, we sustain the idea that PC techniques and tools can also be of great value for students taking the first steps in programming using a specific language. To this end, we aim to improve Python Tutor, a well-known program visualization tool, with graph-based representations of source code such as Control Flow Graph (CFG), Data Flow Graph (DFG), Function Call Graph (FCG) and System Control Graph (SCG).

This helps novice programmers to understand the source code analyzing not only the variable contents but also a set of automatically generated graph-based visualizations, that were not included in Python Tutor so far. This will allow the students to be focused on certain aspects of the program (depending on the graph), abstracting others such as details of its syntax.

2012 ACM Subject Classification Human-centered computing → Visualization systems and tools; Social and professional topics → Computer science education; Software and its engineering → Source code generation

Keywords and phrases Program Visualization, Python Tutor, Data Flow Graphs, Control Flow Graphs

Digital Object Identifier 10.4230/OASIcs.ICPEC.2021.6

Supplementary Material *Software (Graph Builder)*: `https://graph-builder.di.uminho.pt/visualize.html`

Funding This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the Projects Scopes: UIDB/05757/2020 and UIDB/00319/2020.

1 Introduction

There's along the web many tutorials on programming and several learning methodologies but the majority of them uses the same formula [4]: start with the explanation of a set of examples and propose some similar exercises that train the students to write similar code to solve them. To fully understand a programming language, it is not enough to read the code

[1] to mark corresponding author

of programs written in that language, it is also necessary to comprehend how the computer behaves when executing the programs code. For each piece of code, the student shall be able to build a full map between the program domain and the problem domain.

Usually it is hard to understand his own code, even to a software engineer that has to maintain third-part code he never saw before, the task is much harder. The average developer spends about 60% to 90% of his time understanding code previously developed; these figures are not changing over time [19].

With this in mind, this paper describes and discusses the integration into Python Tutor of a new tool for automatic generation of graph-based visualizations to turn ease the task of program comprehension. As said in [7], the best way to understand programs is by *giving programs another aspect than that of their source code.* This new feature works as Python Tutor plugin and it analyses the input code and produces several types of complementary graphs depending on the user desires. As the program comprehension task is also very relevant for the beginners wishing to understand how a given program works and how it solves a given problem, we believe that Python Tutor improved with our plug-in will be very useful to help programming students.

The paper has 5 sections after that. The state of the art on program animation appears in Section 2. Section 3 discusses how graph-based visualizations can be included in an Animator adding actual value to it. Section 4 exhibits the architecture to build the new tool and discusses how it can be developed and integrated into the host tool. Section 5 illustrates the final system built and describes a experiment conducted to assess the value added by the visualizations provided. Section 6 closes the paper and points out some directions for future research.

2 Program Visualization and Animation

Think-aloud protocol [22] is a method where a person verbalize his thoughts and it can be used not only to improve self—understanding and reasoning but also to explain the cognitive process to others. That's why is so important to produce documentation when developing software. The use of this method also allows realizing that the thoughts of a developer differ according to the familiarity of a program's domains [17]. With this in mind, it was derived two concepts of comprehension models, the top-down comprehension model and the bottom-up comprehension models. The top-down approach is used when the programmer is familiar with the program he's trying to understand. He starts by creating a general hypotheses about the program goal and how it is achieved by using his previous knowledge about the program. This is called the *expectation-based* comprehension, where the programmer has pre-generated expectations of the code's meaning [13]. After creating his hypotheses, the developer starts searching in the code for algorithms/structures that he can link to his theory and refines his hypotheses as he does that [6]. This is the *inference-based comprehension* [13]. If the programmer has no previous knowledge about a program, he has to analyze line by line to create a hypothesis about the program purpose. In this method, the developer starts aggregating functions by their goals. What usually happens is that developers end by using an integrated model [11] where they use top-down approach when possible and bottom-up when necessary.

Although the way of coding varies from person to person, follow a guide of good practices in programming (i.e. appropriate variable names and indentation rules [18]) can lead to an uniformed structure that makes program comprehension easier. A different form of coding can really slow down the comprehension of a program for the most experienced programmer [20].

Nowadays, there's even more factors that can affect program comprehension. For example, a simple color schema, available in syntax-oriented text editor/IDE, helps the programmers comprehending a program [16].

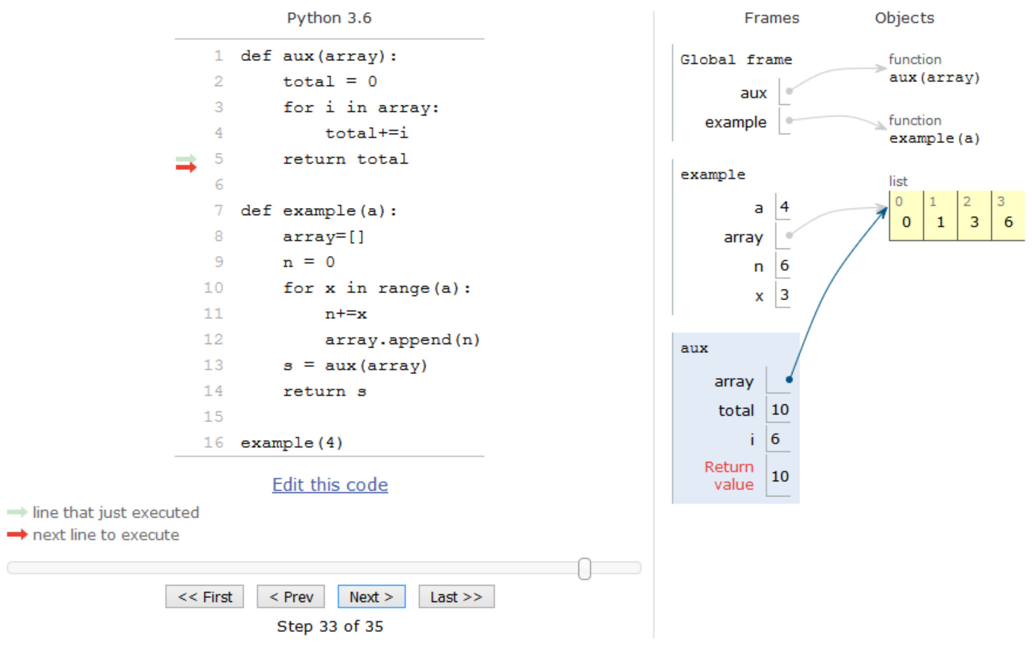
Newer studies have their focus not only on source code comprehension, but also on the comprehension of structures, hierarchies and architecture of the program as well on the relations between components [19].

Software visualization is the process of giving a visual representation of a program. It takes the information that is presented in the source code and converts it to a graphical representation with the goal of improving the comprehension of that program. Although a visual image can help to understand a program, it's not that simple since a basic program can have multiple representations. The best one will depend on the task to be performed and what the developer wants to know. This also means that each representation has to be thought for specific tasks and a good portray of the system is imperative for it to be helpful. *Simply repackaging massive textual information into a massive graphical representation is not helpful*, so a linear translation is not ideal since *experts want to see software visualised in context – not just what the code does, but what it means* [14]. The visual representation of a program can be focused in some aspects of the program and it can have different levels of abstraction. This can also solve scalability problems because the visualization of big programs at once it will not be a good help. So, the visualization tool must allow an high level of interactivity to go one step at a time, filter the information, to establish the data that is wanted, to define the level of abstraction and zoom facilities [8]. Graphically these representations can be based on diagrams, maps, icons, graphs and specific drawings. Some examples are the Nassi-Shneiderman diagram used to represent a control-flow of a program [12]; Space-filling (like tree maps or sunburts) allows to visualize in a very compressed way code metrics and statistics [5]; Graphs are the most common representation and consists of nodes and arcs where the nodes represent blocks of code and the arcs its flow [10].

In terms of functionality, there are two types of visualizations: static and dynamic [10]. A static representation can be seen as a simple image of the program, it doesn't change over time and it's based on static information. Dynamic representations shows the information that changes as the program is executed. A good example of an animated visualization tool is Python Tutor. Python Tutor is an animation web tool that intends to help new programmers to understand programs and currently supports some of the most popular languages at the moment like Python, Java, C, C++, JavaScript, TypeScript and Ruby [15].

An interactive step by step presentation that shows what is really happening behind every line of code during an execution instance (dynamic visualization) is the main feature of Python Tutor. This allows the user to keep up with the modifications of values that each variable suffers as each line of code is executed. The user has total control of this presentation as he can choose if he wants to go to the next or previous step. Global variables are kept in a global frame and then local variables just appear during the function execution.

Tools that produce static visualizations of source code are more usually found. AgileJ StructuredViews [1] is a plugin for eclipse that generates UML class diagrams from the source code. Since UML is a language used to structure software projects that is easy to understand, it can be used to explain to the developers and even to the client. Moreover it creates an easy to read and updated representation of the code. Sourcetrail [3] is a tool that can be connected to an IDE or a text editor that displays an interactive dependency graph and a search bar to search functions, classes or variables. Its interactivity allows the user to see both an overview of the program and the dependencies between classes, methods and variables. CodeSonar [2] offers a visualization software that shows an interactive call



```
def aux(array):
    total = 0
    for i in array:
        total+=i
    return total

def example(a):
    array=[]
    n = 0
    for x in range(a):
        n+=x
        array.append(n)
    s = aux(array)
    return s

example(4)
```

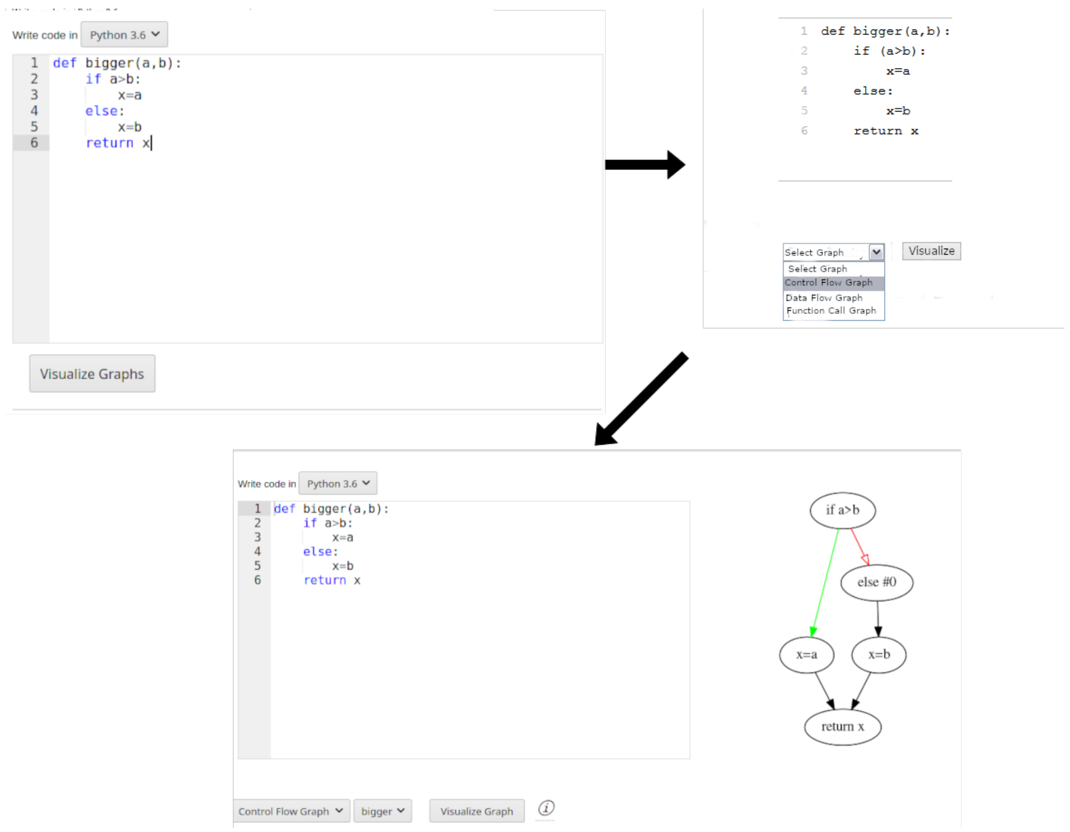
Figure 1 Python Tutor visualization.

graph where it is possible to see the directories, files and functions rearranged hierarchically. It allows the user to choose which metrics he wants to be displayed in the graph as well compare functions/files based on that metrics.

3 Improving Python Tutor with graph-based visualizations

The main idea is to improve the platform Python Tutor by incorporating new features to it. The objective is an interactive tool where the user can select what kind of graph he wants to see in order of being able to analyze visually the flow of code or data dependencies in a part of his code which execution is being animated. Three types of graphs are proposed:

- Control Flow Graph (CFG): It is a representation of all paths (sequences of instructions) that might be traversed during the execution of a given program; each node represents a basic block of code (a sequence of instructions without alternatives, with just one input and one output). By observing a control flow graph we can see to where the values go and from where they came. It is a very useful graph to realize the dependencies that exist in the program and which statements are influenced by others. For instance, a statement B is control dependent on a statement A if B execution depends of the A outcome [23].
- Data Flow Graph (DFG): In this graph there are two types of nodes where one represents the values/variables and other the operators [21]. This graph allows the user to observe all operations that any variable suffers in its lifetime and in this way understand the existing data dependencies. For instance, it is said that B is data dependent on A when A value is used to compute B. This class of graphs is usually more difficult to understand because they tend to represent a lot of information. Besides that, it is not so linear to read as a CFG for example.
- Function Call Graph (FCG): This class of graphs displays the relationships between a function and the functions it calls [9]. On one hand, FCG is useful to understand the functions necessary to execute a function, and this is crucial to local errors and change points. On the other hand, it allows the user to identify the most used functions and



```
def bigger(a,b):
    if a>b:
        x=a
    else:
        x=b
    return x
```

```
def bigger(a,b):
    if (a>b):
        x=a
    else:
        x=b
    return x
```

```
def bigger(a,b):
    if a>b:
        x=a
    else:
        x=b
    return x
```

Figure 2 System Workflow.

the ones that are not being used at all. This information is extracted from the program static analysis and knowing which functions are most called can be useful to improve the program performance, since improving these functions will affect a big part of the program. It is, usually, a very easy to read graph.

These new features were incorporated on Python Tutor adding a new field with a scroll bar to select the graph type and a new button to generate the desired graph. Figure 2 shows a diagram of the desired flow.

The source program to be visualized through the dependency graphs is inserted (typed, or edited) at the home page of Python Tutor in order to reuse as much as possible the existing functionalities. So the input text is sent to the back-end of Python Tutor to be compiled. If the input program is correct and ready to be ran and animated, the front-end will receive the execution trace. Once the input is validated, the system will open a new web page created only to visualize the desired graphs. After the user choose the graph that he wants to see, the input is passed through a new route to the graph generator. The graph generator will create an image with the chosen graph and return it to the front-end. The front-end will display the image on that new page so the user can visualize it and then remove the image in order to avoid an overcrowding of files.

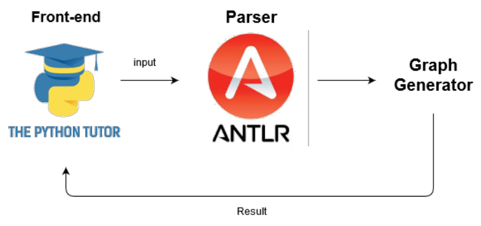


Figure 3 System architecture.

4 Integration Architecture and Developments

The Graph Builder was developed in Python using AnTLR to generate the parser and static analyser for the source programs. It is incorporated into Python Tutor to increase its functionality. The user uses Python Tutor editor to write the input program and then it will be parsed by the syntactic analyser generated by AnTLR to identify the control and data dependencies among program elements and represent them as graphs.

This analyser built by AnTLR from the Python grammar is able to recognize Python programs. Once the information about the input program is gathered the next step is to generate the graphs according to that information. Initially the graphs were built in Python with the help of some libraries like '`igraph`'. In a second phase, it was used PyGraphViz to build better quality graphs with zoom features and more appealing for users.

Figure 3 shows a diagram depicting the architecture of the developed system.

As said above, to implement the referred parser it was required to define a Python grammar. In this case, it was used the Python grammar available on AnTLR Github. Depending on the information needed to extract, it was required to make some minor modifications on the original grammar. For instance, splitting one grammar rule into three different ones in order to treat the data more specifically. After defining the grammar, semantic actions were specified in order to collect information and generate graphs. The semantic actions were associated to each grammar rule and were constantly adjusted during the development of the graph builder tool. Every time some piece of input code allowed to detect an unhandled case, semantic actions were updated (added or changed) in one or more grammar rules. The way the information was processed had to be as accurate as possible to faithfully build graphs that deal with a large amount of data. So, the structures had to be constantly updated with new information.

The three graphs presented use, by definition, a level of granularity where each node corresponds to an instruction, which can be difficult to visualize for large programs. On the other hand, Python Tutor itself has a limit of lines of code for the input program and even so, for larger programs, it is possible to zoom in and zoom out of the graphs.

In the next section, the final structures chosen to represent each graph will be presented and explained as well the information they hold.

4.1 Function Call Graph

There are two types of functions that can be called in a program: functions defined by the developer or built-in functions (i.e., functions that are available from Python libraries). A function is recognized by an identifier (its name) followed by parenthesis or, in case of being a function written by the programmer, by the keyword `def` followed by its definition. All the function calls found in a program are gathered in a Python dictionary where the key

```
def exemplo(listapontos):
    listadist=[]
    for (a,b) in listapontos:
        dist = calcDist(a,b)
        listadist.append(dist)
    imprimeDist(listadist)

def calcDist(a,b):
    dist = sqrt(b**2 + a**2)
    return round(dist)

Def imprimeDist(lista):
    print(lista[0])
    if len(lista)>1:
        imprimeDist(lista[1:])
```

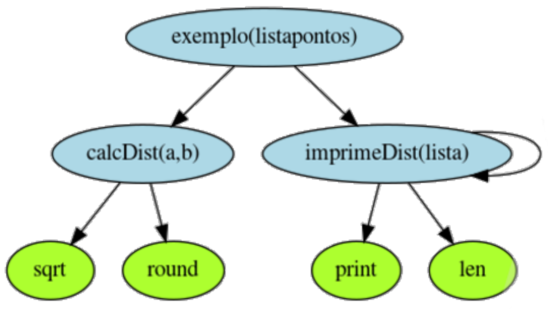


Figure 4 FCG construction.

is the name of the *caller* (the function that calls) and the value is the *callee* (the function called). In Figure 4 it can be seen how the information required to create a Function Call Graph is stored. On the final graph the two types of functions are distinguished by color: blue for the functions created by the user, and green for Python built-in functions.

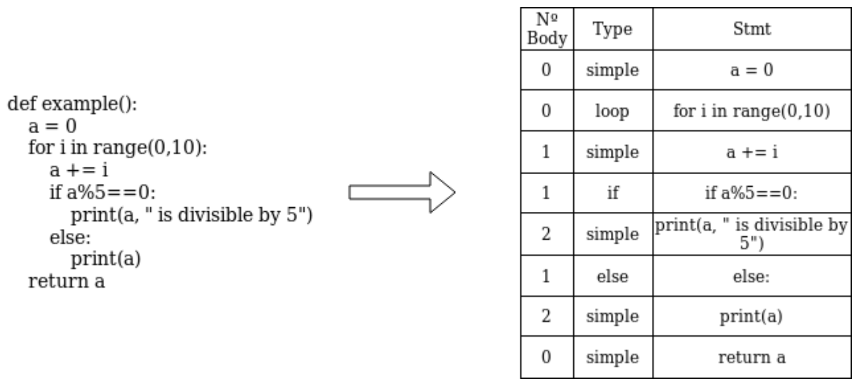
4.2 Control Flow Graph

The data for the construction of a CFG is stored in a similar way as the one explained for the previous graph, that is represented in a dictionary with an entry for each defined function. As the construction of this graph is not so linear as the previous one, it is required to save more information in the dictionary. So, instead of saving just the code statement, it also saves the type of that statement and the block number where it appears.

Analysing each statement, one can notice that some do not affect the execution flow and others like loop (`for` or `while`) or conditional instructions (`if`,`elif`, `else`) have a strong influence on the flow. Each node of the graph can have one or more edges coming out depending on which type of statement it represents. For example, a node of *loop* or *conditional* type will have two edges coming out of it, one in case of the condition being true and the other in case of being false. A *loop* node will also have at least one more edge coming in from the last statement inside the loop. An *else* node will always be preceded by one node of a non-simple type.

The block number represents the statement level: the first level corresponds to the main, other levels mean that is inside a (possibly nested) block depending on a condition. The body (group of statements that depends on one statement) of non-simple statements will always have a bigger depth level. When the depth level changes from one statement to another, it allows the software to recognize that it will start a new block or that one just ended.

There are other statements that affect the flow of a program like for example the reserved words `break` or `continue`. As can be observed in Figure 6, the CFG also contains coloured arrows in order to be more intuitive for the user to identify which path is followed if the condition present in the node is true (green arrow) or false (red arrow). Besides the color difference, the true path arrowhead is filled unlike the false one that is not. These kind of features can help the user to better read the graph.



```
def example():
    a = 0
    for i in range(0,10):
        a += i
        if a%5==0:
            print(a, " is divisible by 5")
        else:
            print(a)
    return a
```

№ Body	Type	Stmt
0	simple	a = 0
0	loop	for i in range(0,10)
1	simple	a += i
1	if	if a%5==0:
2	simple	print(a, " is divisible by 5")
1	else	else:
2	simple	print(a)
0	simple	return a

Figure 5 CFG construction.

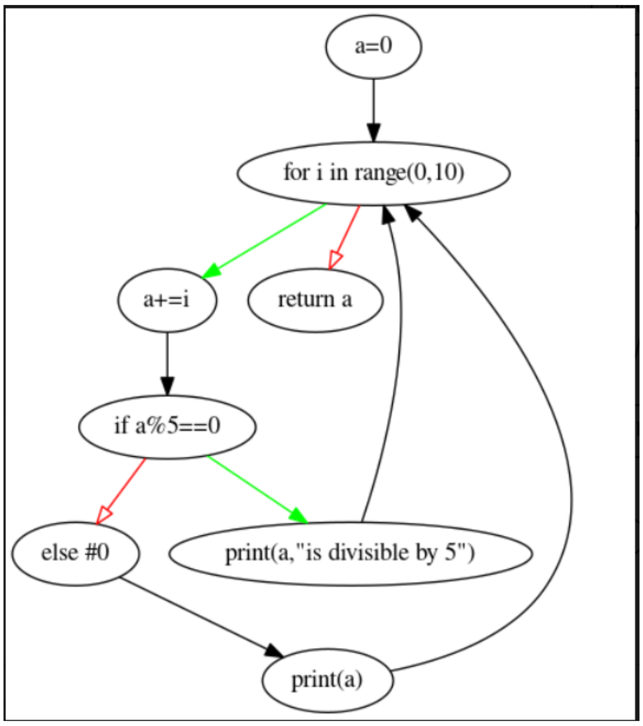


Figure 6 CFG Example.

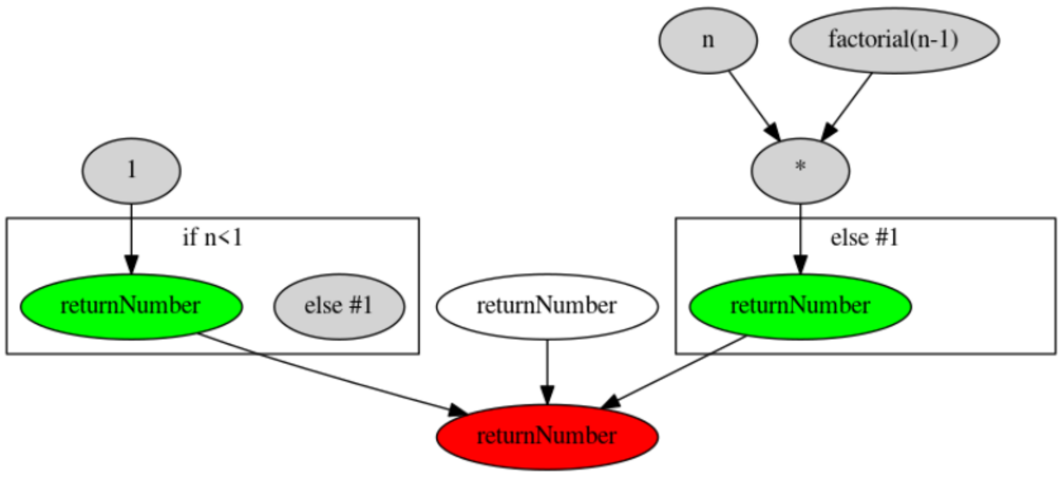


Figure 7 DFG construction.

4.3 Data Flow Graph

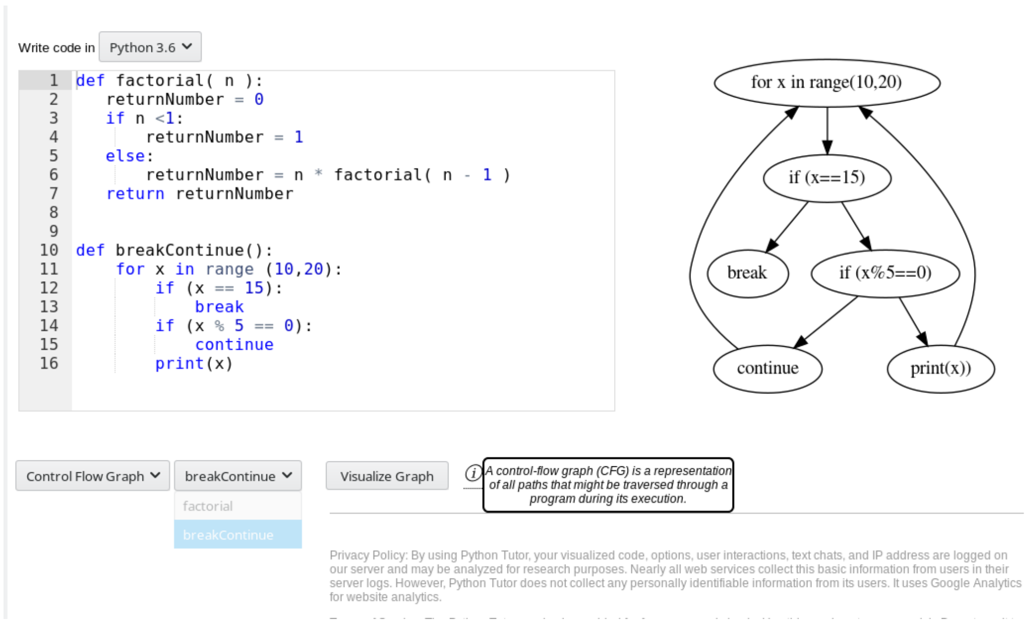
This sort of graph is not as linear as the previous ones because one variable depends not only on the other variables or function results but also on loops or conditional statements. There are also variables that only exist on a specific scope. Despite these constraints, the information necessary to build DFG is gathered in the same way as the previous ones. In this case, there are three types of tuples in the construction of this graph, one that represents a change of value of a variable (variable name, block number, new value and operations that produced the new value), one that serves to control the loops and conditionals dependencies (depth level, statement, alternative path), and a third one to represent the case when a variable can have multiple values depending on the execution path (all the possible variable values).

As can be seen in Figure 7, it is fundamental to use colors for the nodes in order to get more intuitive drawings. Therefore it is used the color green for the nodes that represent the variables which value changed, and in red the nodes that aggregate all possible values of that variable at a given moment. In addition to use different colors different arrow types are also generated as can be seen in figure 6. The other nodes depict values or operations. This graph also includes boxes surrounding sub-graphs that represent loop blocks (cycles) or conditional blocks aiming at producing clearer and more informative drawings.

5 Python Tutor with Graph Builder

To illustrate the project final outcome, Figure 8 shows a screenshot of Python Tutor integrated with the Graph Builder. In this case it was generated a Control Flow Graph (showed in the right side window) for the input program written in the left side window.

In order to assess the users' satisfaction and the usability of the Graph Builder plugin for Python Tutor, a specific experiment with Python Programmers as testers was designed and conducted. For that purpose, it was created a survey that contained instructions for the testers to follow in order to provide their opinion, answering some questions about the new features. The testers should be able to write Python functions and analyse the resulting graphs. This survey was completed by thirty one participants. The majority of the participants were students or ex-students of a Master's in Informatics Engineering, thus fulfilling the necessary requirements. These participants already have programming knowledge; that fact allows us to get the insights of people familiarized with the world of



```
def factorial( n ):
    returnNumber = 0
    if n <1:
        returnNumber = 1
    else:
        returnNumber = n * factorial( n - 1 )
    return returnNumber


def breakContinue():
    for x in range (10,20):
        if (x == 15):
            break
        if (x % 5 == 0):
            continue
        print(x)
```

A control-flow graph (CFG) is a representation of all paths that might be traversed through a program during its execution.

Privacy Policy: By using Python Tutor, your visualized code, options, user interactions, text chats, and IP address are logged on our server and may be analyzed for research purposes. Nearly all web services collect this basic information from users in their server logs. However, Python Tutor does not collect any personally identifiable information from its users. It uses Google Analytics for website analytics.

Figure 8 Python Tutor with Graph Builder.

programming. The results so far obtained were very positive and increased the confidence on this Python Tutor plugin (the Graph Builder); they also provided a good insight on its relevance in the area of program comprehension. The survey also was helpful to identify some minor bugs in the construction of the graphs; after being fixed, the accuracy and the intuitiveness of the graphs were increased.

5.1 Survey Result Analysis

First, it will be presented the results related to the technical side, like the accuracy of the tool and the complexity level of the functions used to test the feature. Then, it will be discussed the opinion of the testers about the features offered by the new Python Tutor plugin. These results, unlike the first ones, will be analysed separately for each graph so that we can identify clearly those that can be more helpful for a deeper program comprehension.

5.1.1 Code Complexity

The first question of the survey was related to the complexity of the code the tester wrote. In order to test a major diversity of programs each tester was free to chose the program to use and this question was included to relate the correctness of the resulting graph with its complexity. As it is observable on the pie-chart of Figure 9, the majority of the functions written to test were of low complexity. This is not a problem for the desired assessment because Python Tutor is directed to new programmers and, under these circumstances, the expected inputs are of low to medium complexity. It is important to note that the complexity of the program is subjective to whoever wrote it.

5.1.2 Graphs Correctness

The three graphics in Figure 10 allow to analyse the accuracy for each graph type.

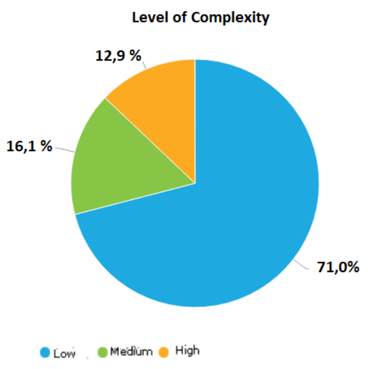
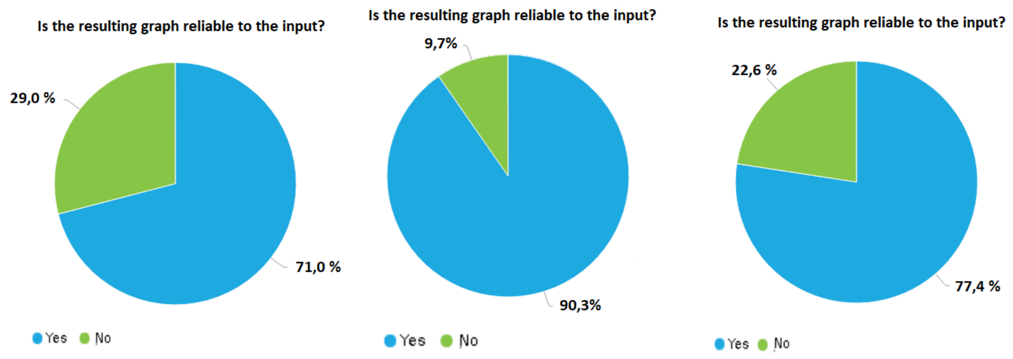


Figure 9 Complexity level of functions used to test the feature.



(a) FCG Accuracy. **(b)** CFG Accuracy. **(c)** DFG Accuracy.

Figure 10 Accuracy of each type of graph.

The numbers shown in Figure 10 indicate a good percentage of accuracy for each graph. The most surprisingly result being the FCG (see 10a); it was expected to have the highest accuracy but ended up being the one with the least. This may have happened because as this is the simplest graph, the tests performed on this graph were not as intense as the others, not allowing to catch as many exceptions like happened with the other graphs.

On the other hand, it was encouraging to realise how accurate was the CFG (see 10b), and that the DFG got also a good rating (see 10c) mainly keeping in mind the amount of dependencies necessary to show. These results were very satisfying mainly because the bugs reported were corrected which means that this accuracy is now increased.

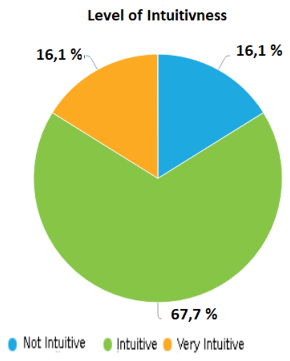
5.1.3 Graphs and Program Comprehension

In this subsection we will see whether graphs available can help people on comprehending a program. The survey prepared for the experiment under discussion includes two questions for each graph.

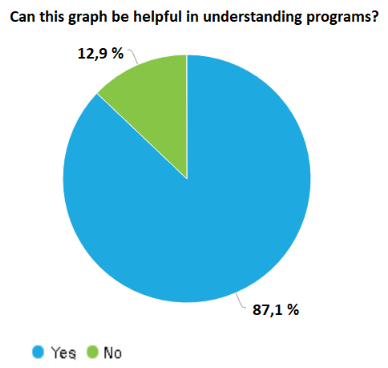
The first question aims at evaluating the intuitiveness of the respective graph, or in other words, perceive whether the graphs are easy to read or not.

The second question aims at understanding if the tester thinks the graphs can help the comprehension of the program being visualized.

According to the graphic in Figure 11b, 87,1% of the programmers who tested the tool think that FCG can improve a program comprehension what follows the Figure 11a where 83.8% find it intuitive. This is because it is a very simple graph providing a generic overview of the system behavior as it only shows the functions who are called by other functions.



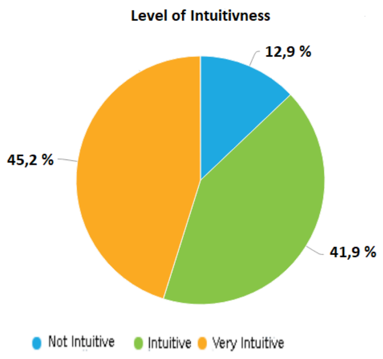
(a) FCG Level of Intuitiveness.



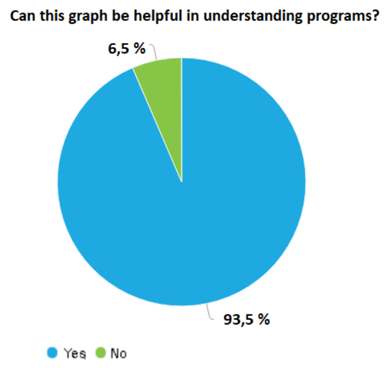
(b) FCG On Program Comprehension.

Figure 11 Perceptions on FCG.

The results shown in the pie graphics of Figure 12 reinforce the preconceived idea that the CFG will be the strength of the new feature. These positive results boost the confidence that this graph can really be helpful on program comprehension.



(a) CFG Level of Intuitiveness.

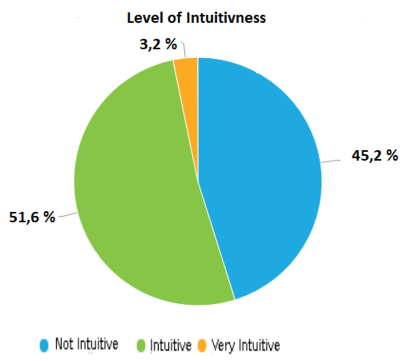


(b) CFG On Program Comprehension.

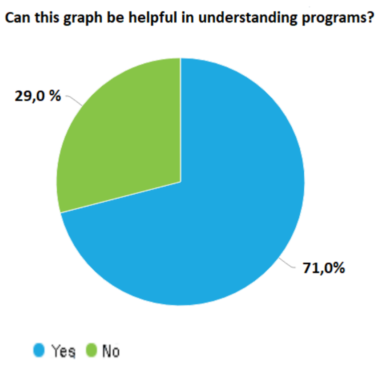
Figure 12 Perceptions on CFG.

As expected, DFG was considered the least intuitive graph (see Figure 13a for details). As shown in Figure 13b, although the lack of intuitiveness of this graph, a considerable amount of people still thinks that it can help on the comprehension of a program, which means that working out on the improvement of this component is a priority in the near future.

At the end, these results were very positive and increased the confidence on this Python Tutor plugin (Graph Builder) and how it can aid on program comprehension. This survey let us identify some minor bugs made along the construction of the graphs that after being fixed increased their accuracy and intuitiveness.



(a) DFG Level of Intuitiveness.



(b) DFG On Program Comprehension.

Figure 13 Perceptions on DFG.

6 Conclusion

Control Flow Graph (CFG), Data Flow Graph (DFG) and Function Call Graph (FCG) are examples of formal instruments that help to visualize statically some program perspectives that depict the program behavior. Those instruments are considered important for professional programmers to better comprehend the software systems they have to deal with. Taking this statement as proved in area of Program Comprehension, we argued in this paper that they can also be valuable artifacts to help beginner programmers to learn that topic. Moreover, if we can combine the visualization of the referred graphs with program animation tools, we believe that we can motivate and engaged students.

In this context, the paper reported a project aiming at building a tool that transforms a function into its corresponding control and data flow graphs and to integrate it with the program animator tool Python Tutor. At the end, it was proven that the usage of program animation features complemented with dependency graphs visualization tools is feasible and facilitates the comprehension of programs.

The ambition was to make this tool a general one that could help all computer programmers, not only beginners having to learn problem solving and deep their knowledge on a specific programming language, but also professionals carrying out their software maintenance tasks. Even recognizing that scaling up this feature to cope with large, real size, programs is not so effective – because the corresponding graphs are big and confusing, and they require more resources than those provided Python Tutor – we think that the experience succeeded as a support to programming courses since it is covered the statements common to the most used imperative languages.

The process of building this tool encountered obviously some obstacles. The biggest difficulty faced was related to the Python Grammar used that does not cover all the real program situations causing many compiling exceptions that frequently arose. Other challenge was the integration with Python Tutor as this platform resorted to some outdated libraries. However, all these troubles have been overcome ending up with a functional tool to help understanding the algorithm and data structures implemented by small programs that students have to study following an easy, attractive and fast learning process.

The design and conduction of new and more complete experiments involving a larger number of students with different ages and in different programming courses, is the most urgent task that needs to be performed to complete the project. It is important to show that it is an effective way to help programming students. The developed tool (Graph-Builder) is available at `https://graph-builder.di.uminho.pt/visualize.html`

As future work, the first proposal is to implement the same control-flow and data-flow graphs for the other languages presented in the Python Tutor like Java and C. It is expected to be easier to implement it now as the new languages will only need the construction of a grammar for each language as the graph generator can be reused and the integration with Python Tutor would only need a few adaptations. The other direction for the future of this Python Tutor add-on is to improve the aspect of the DFG as the testers inquired said that it is not very intuitive and easy to comprehend.

References

1 Agilej structureviews. `http://www.agilej.com/`.

2 Codesonar. `https://www.grammatech.com/products/source-code-analysis`.

3 Sourcetrail. `https://www.sourcetrail.com/`.

4 Luís Alves, Dušan Gajic, Pedro Rangel Henriques, Vladimir Ivancevic, Vladimir Ivkovic, Maksim Lalic, Ivan Lukovic, Maria João Varanda Pereira, Srdan Popov, and Paula Correia Tavares. C Tutor usage in relation to student achievement and progress: A study of introductory programming courses in Portugal and Serbia. *Computer Applications in Engineering Education*, n/a(n/a), 2020. `doi:10.1002/cae.22278`.

5 Marla Baker and Stephen Eick. Space-filling software visualization. *Journal of Visual Languages & Computing*, 6:119–133, June 1995. `doi:10.1006/jvlc.1995.1007`.

6 Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983. `doi:10.1016/S0020-7373(83)80031-5`.

7 Luis M. Gómez-Henríquez. Software visualization: An overview, 2001.

8 Martin Hadley. 3 benefits of interactive visualization, January 2018.

9 Ben Holland. Call graph construction algorithms explained, March 2016.

10 François Lemieux and Martin Salois. Visualization techniques for program comprehension - a literature review. In *SoMeT*, 2006.

11 A. Mayrhauser and A. Marie Vans. From program comprehension to tool requirements for an industrial environment. In *[1993] IEEE Second Workshop on Program Comprehension*, pages 78–86, August 1993. `doi:10.1109/WPC.1993.263903`.

12 Ike Nassi and B. Shneiderman. Flowchart techniques for structured programming. *ACM SIGPLAN Notices*, 8:12–26, August 1973. `doi:10.1145/953349.953350`.

13 Michael O'Brien, Jim Buckley, and Teresa Shaft. Expectation-based, inference-based, and bottom-up software comprehension. *Journal of Software Maintenance*, 16:427–447, November 2004. `doi:10.1002/smr.307`.

14 Marian Petre. Mental imagery, visualisation tools and team work. In *Proceedings of the Second Program Visualisation Workshop*, pages 3–14, January 2002.

15 Ben Putano. A look at 5 of the most popular programming languages of 2019. `https://stackify.com/popular-programming-languages-2018/`. Accessed: 26-9-2019.

16 Gerard Rambally. The influence of color on program readability and comprehensibility. *ACM Sigcse Bulletin*, 18(1):173–181, February 1986. `doi:10.1145/5600.5702`.

17 Teresa M. Shaft and Iris Vessey. The relevance of application domain knowledge: Characterizing the computer program comprehension process. *J. Manage. Inf. Syst.*, 15(1):51–78, 1998. `doi:10.1080/07421222.1998.11518196`.

18 Ben Shneiderman, Richard Mayer, Don McKay, and Peter Heller. Experimental investigations of the utility of detailed flowcharts in programming. *Commun. ACM*, 20:373–381, June 1977. `doi:10.1145/359605.359610`.

19 J. Siegmund. Program comprehension: Past, present, and future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 13–20, March 2016. `doi:10.1109/SANER.2016.35`.

20 E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, September 1984. `doi:10.1109/TSE.1984.5010283`.

21 Marilyn Wolf. *Computers As Components, Third Edition: Principles of Embedded Computing System Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2012.

22 Mami Yoshida. Think-aloud protocols and type of reading task: The issue of reactivity in l2 reading research. In *Selected Proceedings of the 2007 Second Language Research Forum*, January 2008.

23 Andreas Zeller. Chapter 7 - deducing errors. In Andreas Zeller, editor, *Why Programs Fail (Second Edition)*, pages 147–173. Morgan Kaufmann, Boston, second edition edition, 2009. doi:10.1016/B978-0-12-374515-6.00007-1.