FEDERAL UNIVERSITY OF ITAJUBÁ - UNIFEI
INSTITUTE OF SYSTEMS ENGINEERING
AND INFORMATION TECHNOLOGY

Python Graphs with Graphical User Interface

Ana Carolina de Campos - 34199
Geowana Marcella Siqueira - 34191
Pedro Palhari Santos - 33547

Itajubá, nov/2019

1 Introduction

1.1 Technologies and resources

1.1.1 Tkinter – Canvas

Tkinter is one of Python's tools to aid in the development of graphical interfaces. To better understand this tool, you need to know some Tkinter terms.

One is the widget, which refers to any component of the graphic user interface (GUI). Another term is the 'event', which can refer to any user action, such as a mouse click or key press. And, event handlers are responsible for handling these events.

Canvas is a Tkinter widget capable of creating other widgets, making drawings and simulations. Some of its main objects are the arc, a line, an ellipse, or the text. When an object is created, Canvas automatically receives an ID, which is the integer.

However, on a Canvas with dozens of objects, some may have too much relevance to be identified with just a number, so you can name them using tags. For example, to assign the tag "firstBall"to a circle, you must:

self.name_canvas.create_oval(coord., options, tag = "firstBall")

1.1.2 Python Dictionary Concept

This project used data structures that implement mappings. As mapping is a collection of associations between pairs of values, the first element of the parent is called content. In a way, a mapping is a generalization of the idea of accessing data by indexes, except that in a mapping the indexes (or keys) can be of any immutable type.

To create a dictionary, you create a dictionary constant (EXAMPLE). This variable can be indexed in the usual way, i.e. using square brackets. The content associated with a key can be changed by assigning it to that position in the dictionary. New values can be added to a dictionary by assigning to a key not yet defined. And finally, it is not a defined order between key/content pairs in a dictionary.

Example

dic={"joao":100,"maria":150}

dic["joao"]

100

dic["maria"]

150

dic["pedro"]=10

dic{'pedro': 10, 'joao':100, 'maria':150 }

dic{'joao':100, 'maria':150, 'pedro':10 }

dic{'pedro': 10, 'joao':100, 'maria':150 }

The dictionary keys are not stored in any specific order. In fact, dictionaries are implemented by scattering (Hash Tables). The lack of order is purposely, unlike lists, assign to an element of a dictionary does not require the position to exist previously.

1.1.3 Breadth First Search or BFS for a Graph

This project chose to use BFS as a methodology to solve the graph problem. The chosen algorithm aims to systematically explore all vertices and represents the simplest research methodology that exists among others. [1]

The algorithm works as follows: first visit all nodes near the root of the search (by levels) before visiting the furthest. The spawning tree is a shallow (larger) tree with many children for each node. In addition, it can be implemented using a queue. An example of the simulation is shown below. [1]

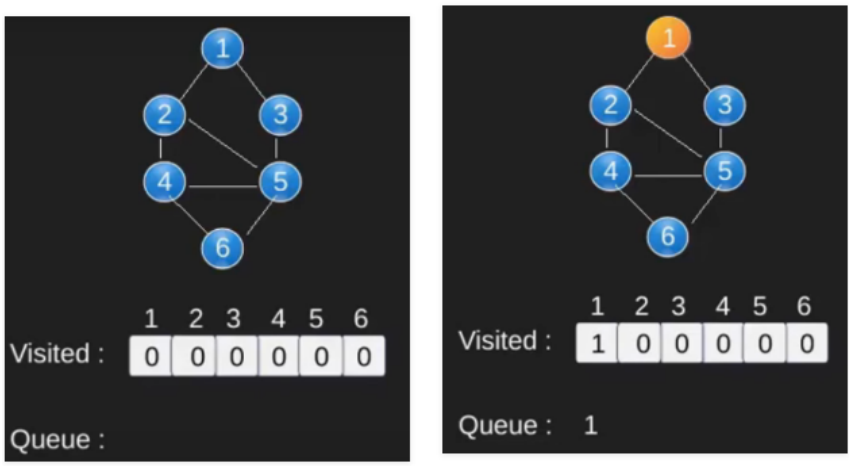

Figura 1

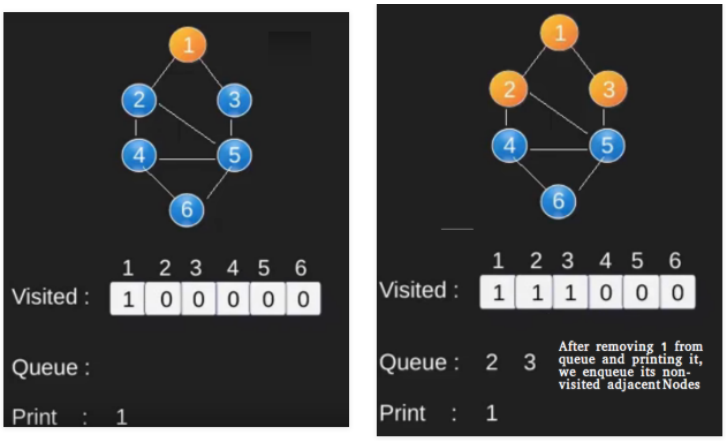

Figura 2

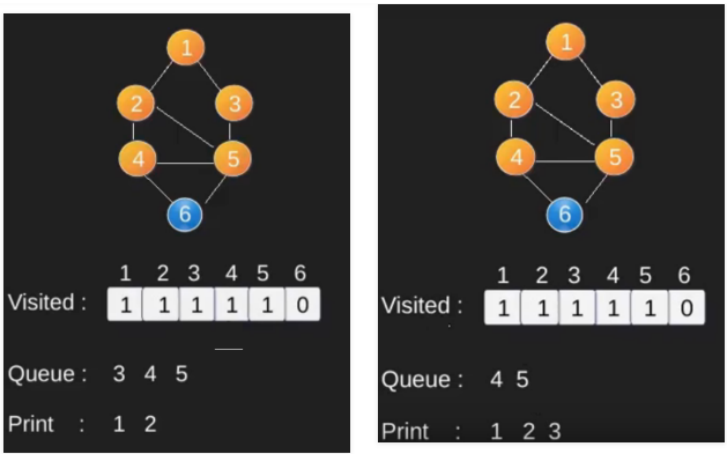

Figura 3

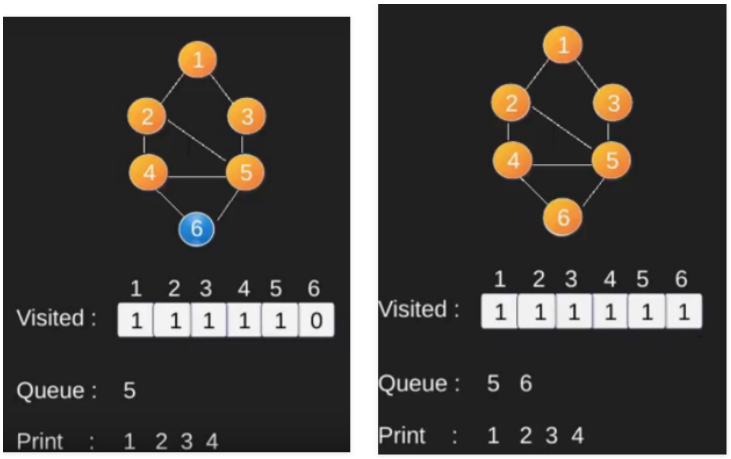

Figura 4

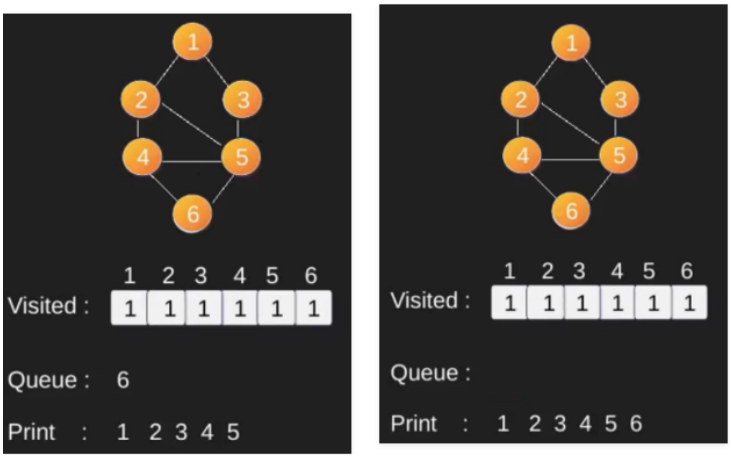

Figura 5

Then, having given the idea of simulation it is possible to develop the algorithm.

1.1.4 Factory Method

The Factory Method was best known with the book 'Design Patterns: Elements of Reusable Oriented Object Oriented' in 1994, written by Erich Gamma, John Vlissides, Ralph Johnson and Richard Helm, also known as Gang of Four or GoF.

This method is widely used in Java, but can also be used in other programming languages such as Python.

According to the GoF this method, "defines an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses". [1]

When a client class is initialized, the responsibility lies with the factory class, as if it were a virtual builder. This factory class provides the objects, which can be accessed through a common interface.

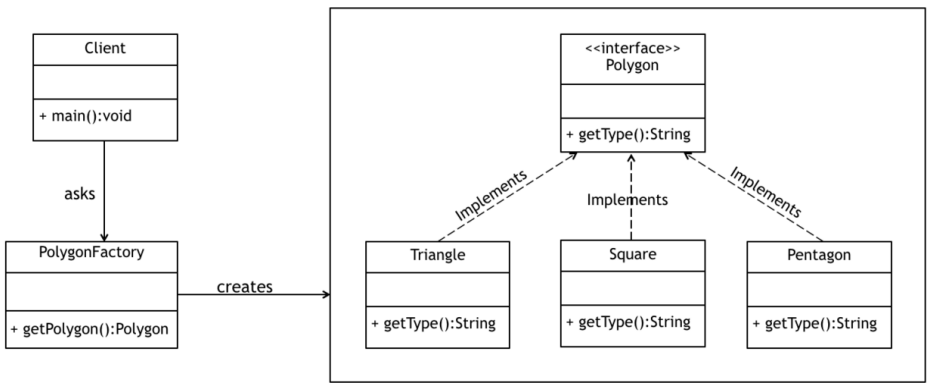

Figura 6 – Diagram of a factory method

Some advantages of using the Factory Method is that object creation is decoupled from knowledge of the concrete object type. It also connects parallel class hierarchies and facilitates extensibility.

2 Development

Brief explaining of the structs, with coding, used in the app and it's purpose. It presents a lot of coding and then the final flux of runtime execution, so it's advised that the reader skim lightly through the content on the first time to take in the defined structure, and then reach the end and understand the runtime flux assuming it's connections based on the structure.

2.1 Using the factory method with injected global props

The code is separated in 2 different layouts. The *Main.py* that defines the initializes the entities and the *ComponentHandler.py* that initializes each of the Components (Button, Vertex).

An example of creating a button in *Main.py*:

```
import tkinter as tk
import ComponentsHandler

root = tk.Tk()
canv = tk.Canvas(root, width=640, height=480)

ComponentManager = ComponentsHandler.initComponents(canv, GLOBAL_OBJ)

ComponentManager.createButton(
    470, 420, 150, 40, text="BFS!", callbackFunc=doBFS)

canv.pack()
root.mainloop()
```

The component handler is defined as:

```
from types import SimpleNamespace

import Button

def initComponents(canvas, GLOBAL_OBJ):
    createButton = Button.initButton(canvas).createButton

    returnMap = {
```

```
        "createButton": createButton,
    }

    return SimpleNamespace(**returnMap)
```

A single button component is defined as:

```
def initButton(canvas):
    def createButton(x, y, width, height, text, callbackFunc):
        nonlocal canvas

        # Use the canvas reference to draw a button
        button = canvas.create_rectangle(
            x, y, x + width, y + height, width=5, fill="black")

        def clickFeedback(event):
            nonlocal button

            #Do things with the button reference...

        canvas.tag_bind(button, '<ButtonPress-1>', clickFeedback)

    returnMap = {
        "createButton": createButton
    }

    return SimpleNamespace(**returnMap)
```

Explaining to some key definitions that helps us understand better the model and its purposes:

- *nonlocal*: a python keyword that allows us to use clojures. In that way, when I'm using **createButton(...)** the canvas reference is the same as the one passed on the **initButton(...)** function in the *ComponengHandler.py*. The same applies to *nonlocalbutton*. So whenever a button is clicked, it's actions take effect on itself (for example, the button flashes when I hover it).
- *SimpleNamespace*: a python implementation that allows for someone to call a dictionary with the dot notation, for example:

```
    function initialize():
        def log():
```

```
            print("I'm being logged!")

        #a dictionary

        returnMap = {
            "log": log
        }

    return SimpleNamespace(**returnMap)


    myComponent = initialize()

    myComponent.log() #calls the function

    #without SimpleNamespace, we'd need to do myComponent['log'], t
```

2.2 Connecting a graph to the canvas

When creating a canvas object, it returns an id that represents uniquely that object in the canvas, allowing us to modify it.

So, if the graph structure is defined as:

dict {<VERTEX_ID>: [<VERTEX_NEIGHBOR>, <VERTEX_NEIGHBOR>, <VERTEX_NEIGHBOR>, ...]}

We could use an algorithm that traverses the Graph (such as BFS) and when traversing indicate on the Canvas that we are processing that part of the graph, for example, changing its color with Tk functions as:

```
canvas.itemconfig(vertex_id, fill="red", outline="black")
```

The only thing the BFS function would need alongside it's natural execution is a canvas reference, easily injected after initialized.

Another global object used to keep references is the lines that connects the vertexes, defined as:

dict {<LINE_ID>: (<OUT_VERTEX_ID>, <IN_VERTEX_ID>)}

To end, we also keep two references that stores the ID of the two clicked vertex, so we can connect them with a line.

All of these properties are grouped in an object called **GLOBAL_OBJ** that is injected, alongside the canvas, to the objects and graph algorithms.

2.3 Setup and runtime flux

2.3.1 Setup

The setup flux is:

1. Initialize the GLOBAL_OBJ empty.
2. Initialize the canvas.
3. Initialize the ComponentHandler, passing GLOBAL_OBJ and canvas, injecting them as *nonlocals* in the components.
4. Create the buttons and bind the graph functions to them.
5. Bind the *createVertex* and *deleteVertex* functions to canvas clicks.
6. Run the GUI loop.

2.3.2 Creating a vertex

When creating a vertex:

1. Draw the vertex and text in the clicked position using the canvas injected
2. Create *onClick* functions for the vertex and bind it to them, create also *onHover* functions for visual feedback

2.3.3 Clicking on a vertex

When clicking on a vertex:

1. Check if the *GLOBAL_OBJ.out* has a value
   - if it doesn't has, set this VERTEX_ID to it.
   - else,
     a) add this connection to the global graph (*GLOBAL_OBJ.graph*) object as a neighbor of the *GLOBAL_OBJ.out* vertex
     b) clear *GLOBAL_OBJ.out*
     c) get both vertex positions using a canvas helper similar to coloring it and draw a line using these positions indicating both vertexes are connected
     d) add a line reference to *GLOBAL_OBJ.lineSet*

2.3.4 Deleting a vertex

When deleting a vertex:

1. delete all its graphical references using a canvas helper function
2. delete all lines that are in some way related to this vertex from the *GLOBAL_OBJ.lineSet* and also graphically
3. remove its reference as a neighbor from all the other vertexes in *GLOBAL_OBJ.graph*

2.3.5 Running BFS

When running BFS:

1. get the graph from *GLOBAL_OBJ.graph* and start the algorithm
2. when adding to the visited array, also color it using a canvas helper function

2.3.6 Running tree

When running tree:

1. the same as bfs, but keeping the lines that represents the vertex being added to visited
2. deleting the vertex that are out, graphically and on the *GLOBAL_OBJ.graph*, sanitizing it after to remove them as neighbors to other vertexes
3. deleting the lines that didn't get included and its references

2.4 Bonus: drawing the line from the vertex border

Just because it took a long time, and a lot of brain melted in the effort:

```
canvas.create_line(
    fromCoords[0] + r/2 + math.cos(angle) * r/2,
    fromCoords[1] + r/2 + math.sin(angle) * r/2,
    toCoords[0] + r/2 - math.cos(angle) * r/2,
    toCoords[1] + r/2 - math.sin(angle) * r/2,
    width=5, arrow="last")
```

2.5 Github repo

To view the source code of this work, go to: ECOE41_Canvas.

The repository includes instructions on using the application alongside a *gif* of it in action.

Referências

1 GEEKSFORGEEKS. *Breadth First Search oh BFS for a Graph*. 2019. Disponível em: <https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>. Acesso em: 22 nov 2019.