Graph Implementation

This lesson will cover the implementation of a directed graph via adjacency list in Python.

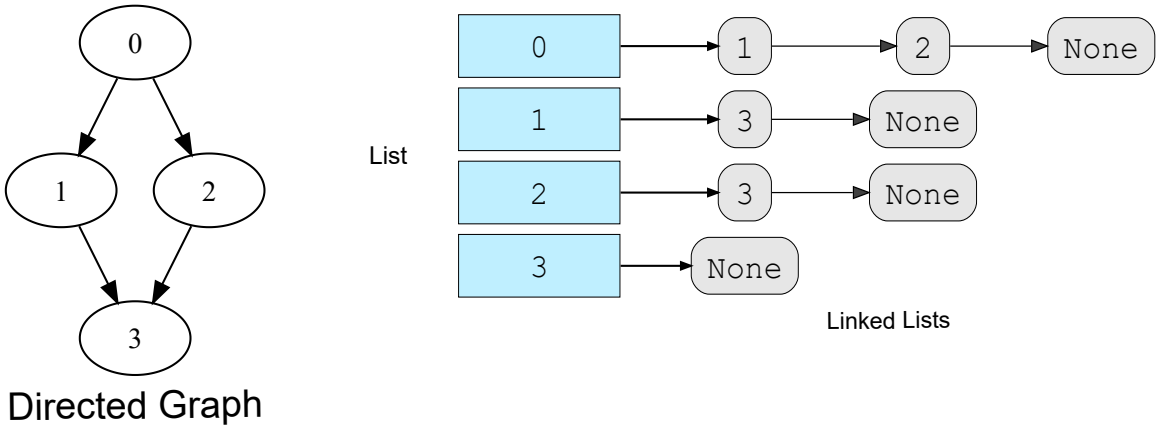
We'll cover the following

Introduction

At this point, we've understood the theoretical concepts of graphs. In this lesson, we will use this knowledge to implement the graph data structure in Python. Our graph will be directed and have no bidirectional edges.

The implementation will be based on the **adjacency list** model. The linked list class we created earlier will be used to represent adjacent vertices.

As a refresher, here is the illustration of the graph we'll be producing using an adjacency list:



The Graph Class

Graph class consists of two data members:

- The total number of vertices in the graph
- A list of linked lists to store adjacent vertices

So let's get down to the implementation!

Graph.py

```
class Graph:
    def __init__(self, vertices):
        # Total number of vertices
        self.vertices = vertices
        # Defining a list which can hold multiple LinkedLists
        # equal to the number of vertices in the graph
        self.array = []
        # Creating a new LinkedList for each vertex/index of the list
        for i in range(vertices):
            temp = LinkedList()
            self.array.append(temp)

```

We've laid down the foundation of our `Graph` class. The variable `vertices` contains an integer specifying the total number of vertices.

The second component is `array`, which will act as our adjacency list. We simply have to run a loop and create a linked list for each vertex.

Additional Functionality

Now, we'll add two methods to make this class functional:

1. `print_graph()` - Prints the content of the graph
2. `add_edge()` - Connects a source with a destination

main.py

Graph.py

LinkedList.py

Node.py

```
from LinkedList import LinkedList


class Graph:
    def __init__(self, vertices):
        # Total number of vertices
        self.vertices = vertices
        # defininng a list which can hold multiple LinkedLists
        # equal to the number of vertices in the graph
        self.array = []
        # Creating a new Linked List for each vertex/index of the li
        for i in range(vertices):
            temp = LinkedList()
            self.array.append(temp)

    # Function to add an edge from source to destination
    def add_edge(self, source, destination):
        if (source < self.vertices and destination < self.vertices):
        # As we are implementing a directed graph, (1,0) is not equa
            self.array[source].insert_at_head(destination)
            # Uncomment the following line for undirected graph
            # self.array[destination].insert_at_head(source)


        # If we were to implement an Undirected Graph i.e (1,0) == (
        # We would create an edge from destination towards source as
        # i.e self.list[destination].insertAtHead(source)

    def print_graph(self):
        print(">>Adjacency List of Directed Graph<<")
        print
        for i in range(self.vertices):
            print("|", i, end=" | => ")
```

```
34              temp = self.array[i].get_head()
35              while(temp is not None):
36                  print("[", temp.data, end=" ] -> ")
37                  temp = temp.next_element
38              print("None")
39
```

Let's break down the two new functions that we've implemented.

add_edge (self, source, destination)

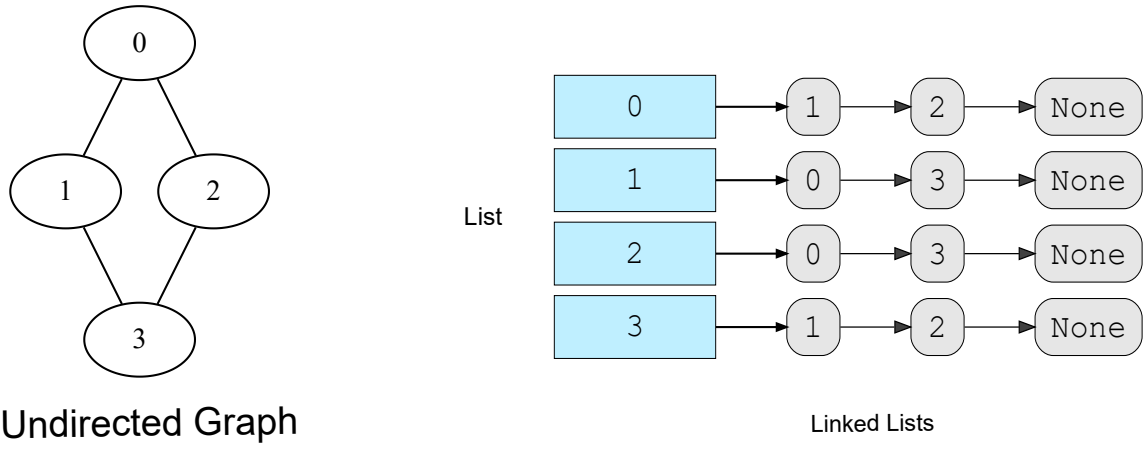
Thanks to the graph constructor, `source` and `destination` are already stored as indices of our list. This function simply inserts a destination vertex into the adjacency linked list of the source vertex by running the following line of code:

```
array[source].insert_at_head(destination)
```

One important thing to note is that we are implementing a directed graph, so `add_edge(0, 1)` is not equal to `add_edge(1, 0)`. In the case of an undirected graph, we will have to create an edge from the source to the destination and from the destination to the source, making it a bidirectional edge:

```
array[source].insert_at_head(destination)
array[destination].insert_at_head(source)
```

The figure below illustrates the corresponding undirected graph with bidirectional edges.



`addEdge()` will not work if `source` is less than zero and greater than or equal to the number of vertices. Likewise, `destination` also has to be greater than or equal to 0 and less than the number of vertices. In the actual production code, you need to cover the error handling of these edge cases.

print_graph(self)

This function uses a simple nested loop to iterate through the adjacency list. Each linked list is

being traversed here.

We've seen the `add_edge` and `print_graph` methods. What do you think is the time complexity of these functions? The next lesson will answer this question.

Back

Representation of Graphs

Next

Complexities of Graph Operations

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/graph-implementation__introduction-to-graphs__data-structures-for-coding-interviews-in-python)