Programming fundamentals

Graphs with OOP

Pepe García

2020-11-30

Introduction

In this session we'll create a OOP version of a graph using adjacency lists.

Project

There will be three modules in our project:

Project

There will be three modules in our project:

- session-21.py

Project

There will be three modules in our project:

- session-21.py
- graph.py

Project

There will be three modules in our project:

- session-21.py
- graph.py
- vertex.py

Creating the Vertex class

In the `vertex.py` file.

Vertex

Create a Vertex class with two attributes: value and edges

Creating the Vertex class

In the `vertex.py` file.

Vertex

Create a Vertex class with two attributes: value and edges

get_edges

Create a method in the Vertex class to return the edges as a list

Creating the Vertex class

```
class Vertex:

    def __init__(self, value):
        self.value = value
        self.edges = {}

    def get_edges(self):
        return list(self.edges.keys())
```

Adding edges to a vertex

In the `vertex.py` file.

add_edge

Create a method add_edge in the Vertex class. The edge should be represented as a True value in the dictionary.

Adding edges to a vertex

```
class Vertex:

    def __init__(self, value):
        self.value = value
        self.edges = {}

    def add_edge(self, vertex):
        self.edges[vertex] = True

    def get_edges(self):
        return list(self.edges.keys())
```

Creating the Graph

Now, in `graph.py`, let's create a `Graph` class. This `Graph` class should receive a boolean in the constructor method that indicates if the graph is directed or not. It should initialize two attributes:

- `directed`
- `graph_dict`

Creating the Graph

Now, in `graph.py`, let's create a `Graph` class. This `Graph` class should receive a boolean in the constructor method that indicates if the graph is directed or not. It should initialize two attributes:

- `directed`
- `graph_dict`

```
class Graph:

    def __init__(self, directed = False):
        self.directed = directed
        self.graph_dict = {}
```

Creating the Graph

After creating the `Graph` class, let's create a method for adding a new vertex to the graph. Adding a new vertex to the graph means adding a key to the `graph_dict` that represents the value of the vertex, and the `Vertex` itself as value.

Creating the Graph

After creating the `Graph` class, let's create a method for adding a new vertex to the graph. Adding a new vertex to the graph means adding a key to the `graph_dict` that represents the value of the vertex, and the `Vertex` itself as value.

```
class Graph:

    # def __init__(self, directed = False): ...

    def add_vertex(self, vertex):
        self.graph_dict[vertex.value] = vertex
```

Adding edges in a graph

Now we will need to create a method to add edges between vertices in a graph.

inside the `Graph` class, create an `add_edge` method, that will take two vertices. If the graph is directed it should create edge one single edge, if it's undirected, it should create both.

Adding edges in a graph

```
class Graph:
    # def __init__(self, directed = False): ...
    # def add_vertex(self, vertex): ...

    def add_edge(self, from, to):
        self.graph_dict[from.value].add_edge(to.value)
```

Adding edges in a graph

```
class Graph:
    # def __init__(self, directed = False): ...
    # def add_vertex(self, vertex): ...

    def add_edge(self, from, to):
        self.graph_dict[from.value].add_edge(to.value)

        if not self.directed:
            self.graph_dict[to.value].add_edge(from.value)
```

Building a graph

Now, in `session21.py`, let's create a couple of vertex and add them to a graph!

Building a graph

```
from vertex import Vertex
from graph import Graph

plaza_de_castilla = Vertex("Plaza de Castilla")
chamartin = Vertex("Chamartin")
cuzco = Vertex("Cuzco")

metro = Graph(directed = False)

metro.add_vertex(plaza_de_castilla)
metro.add_vertex(chamartin)
metro.add_vertex(cuzco)

metro.add_edge(plaza_de_castilla, chamartin)
metro.add_edge(chamartin, cuzco)
```

Pathfinding

In order to implement pathfinding, we will need to apply the same BFS algorithm we applied for *dictionary-based* graphs.

Pathfinding

Remember the BFS implementation for graphs:

```
def find_all_paths(graph, start):
    queue = [start]
    paths = {start: [start]}

    while queue:
        current = queue.pop(0)
        for neighbor in graph[current]:
            if neighbor not in paths:
                paths[neighbor] = paths[current] + [neighbor]
                queue.append(neighbor)
    return paths
```

Pathfinding

```
class Graph

    def find_path(self, start_vertex, end_vertex):
        queue = [start_vertex.value]
        paths = {start_vertex.value: [start_vertex.value]}
        while len(queue) > 0:
            current_vertex = queue.pop(0)

            for neighbor in self.graph_dict[current_vertex].get_edges():
                if neighbor not in paths:
                    paths[neighbor] = paths[current_vertex] + [neighbor]
                    queue.append(neighbor)
        if end_vertex.value in paths:
            return paths[end_vertex.value]
        else:
            return None
```