**Module** **23.** **Graphs (Graph Theory)**

**Learning Objectives**

* Understand the fundamental principles of graph modeling, implementation, and practical applications using Python libraries.
* Define the key terms; participants should recognize that a graph comprises nodes (or vertices) connected by edges, forming structures that can represent various real-world scenarios.
* Explain the different types of graphs such as directed, undirected, and weighted graphs is crucial for understanding how to model various relationships.
* Implement graphs in Python effectively. Python does not have a built-in graph data type, so student need to familiarize themselves with alternative representations, such as using dictionaries or custom classes. This includes creating, modifying, and accessing graph structures through practical coding exercises, which strengthens programming skills alongside theoretical knowledge.

1. **Introduction**

A "graph" in mathematics and computer science consists of "nodes", also known as "vertices". Nodes may or may not be connected with one another. In our illustration, Figure 1, - which is a pictorial representation of a graph,

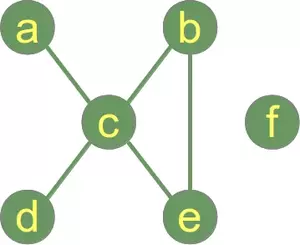


Figure 1. Graphs and its components.

* the node "a" is connected with the node "c", but "a" is not connected with "b". The connecting line between two nodes is called an edge. If the edges between the nodes are undirected, the graph is called an undirected graph. If an edge is directed from one vertex (node) to another, a graph is called a directed graph. An directed edge is called an arc. Though graphs may look very theoretical, many practical problems can be represented by graphs. They are often used to model problems or situations in physics, biology, psychology and above all in computer science. In computer science, graphs are used to represent networks of communication, data organization, computational devices, the flow of computation, In the latter case, the are used to represent the data organization, like the file system of an operating system, or communication networks. The link structure of websites can be seen as a graph as well, i.e. a directed graph, because a link is a directed edge or an arc. Python has no built-in data type or class for graphs, but it is easy to implement them in Python. One data type is ideal for representing graphs in Python, i.e. dictionaries. The graph in our illustration can be implemented in the following way:
* graph = { "a" : {"c"},
* "b" : {"c", "e"},
* "c" : {"a", "b", "d", "e"},
* "d" : {"c"},
* "e" : {"c", "b"},
* "f" : {}
* }

The keys of the dictionary above are the nodes of our graph. The corresponding values are sets with the nodes, which are connected by an edge. A set is better than a list or a tuple, because this way, we can have only one edge between two nodes. There is no simpler and more elegant way to represent a graph.

An edge can also be ideally implemented as a set with two elements, i.e. the end nodes. This is ideal for undirected graphs. For directed graphs we would prefer lists or tuples to implement edges.

Function to generate the list of all edges:

graph = { "a" : {"c"},

"b" : {"c", "e"},

"c" : {"a", "b", "d", "e"},

"d" : {"c"},

"e" : {"c", "b"},

"f" : {}

}

def generate\_edges(graph):

edges = []

for node in graph:

for neighbour in graph[node]:

edges.append({node, neighbour})

return edges

print(generate\_edges(graph))

Output



As we can see, there is no edge containing the node "f". "f" is an isolated node of our graph. The following Python function calculates the isolated nodes of a given graph:

graph = { "a" : {"c"},

"b" : {"c", "e"},

"c" : {"a", "b", "d", "e"},

"d" : {"c"},

"e" : {"c", "b"},

"f" : {}

}

def generate\_edges(graph):

edges = []

for node in graph:

for neighbour in graph[node]:

edges.append({node, neighbour})

return edges

print(generate\_edges(graph))

def find\_isolated\_nodes(graph):

""" returns a set of isolated nodes. """

isolated = set()

for node in graph:

if not graph[node]:

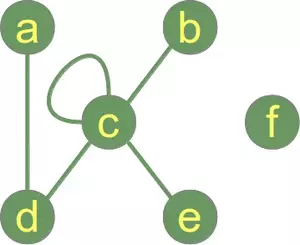
isolated.add(node)

return isolated

Output



1. **Graphs as a Python Class**



Before we go on with writing functions for graphs, we have a first go at a Python graph class implementation.

If you look at the following listing of our class, you can see in the init-method that we use a dictionary "self.\_graph\_dict" for storing the vertices and their corresponding adjacent vertices.

""" A Python Class

A simple Python graph class, demonstrating the essential

facts and functionalities of graphs.

"""

class Graph(object):

def \_\_init\_\_(self, graph\_dict=None):

""" initializes a graph object

If no dictionary or None is given,

an empty dictionary will be used

"""

if graph\_dict == None:

graph\_dict = {}

self.\_graph\_dict = graph\_dict

def edges(self, vertice):

""" returns a list of all the edges of a vertice"""

return self.\_graph\_dict[vertice]

def all\_vertices(self):

""" returns the vertices of a graph as a set """

return set(self.\_graph\_dict.keys())

def all\_edges(self):

""" returns the edges of a graph """

return self.\_\_generate\_edges()

def add\_vertex(self, vertex):

""" If the vertex "vertex" is not in

self.\_graph\_dict, a key "vertex" with an empty

list as a value is added to the dictionary.

Otherwise nothing has to be done.

"""

if vertex not in self.\_graph\_dict:

self.\_graph\_dict[vertex] = []

def add\_edge(self, edge):

""" assumes that edge is of type set, tuple or list;

between two vertices can be multiple edges!

"""

edge = set(edge)

vertex1, vertex2 = tuple(edge)

for x, y in [(vertex1, vertex2), (vertex2, vertex1)]:

if x in self.\_graph\_dict:

self.\_graph\_dict[x].add(y)

else:

self.\_graph\_dict[x] = [y]

def \_\_generate\_edges(self):

""" A static method generating the edges of the

graph "graph". Edges are represented as sets

with one (a loop back to the vertex) or two

vertices

"""

edges = []

for vertex in self.\_graph\_dict:

for neighbour in self.\_graph\_dict[vertex]:

if {neighbour, vertex} not in edges:

edges.append({vertex, neighbour})

return edges

def \_\_iter\_\_(self):

self.\_iter\_obj = iter(self.\_graph\_dict)

return self.\_iter\_obj

def \_\_next\_\_(self):

""" allows us to iterate over the vertices """

return next(self.\_iter\_obj)

def \_\_str\_\_(self):

res = "vertices: "

for k in self.\_graph\_dict:

res += str(k) + " "

res += "\nedges: "

for edge in self.\_\_generate\_edges():

res += str(edge) + " "

return res

g = { "a" : {"d"},

"b" : {"c"},

"c" : {"b", "c", "d", "e"},

"d" : {"a", "c"},

"e" : {"c"},

"f" : {}

}

graph = Graph(g)

for vertice in graph:

print(f"Edges of vertice {vertice}: ", graph.edges(vertice))

Output

A screen shot of a computer code

Description automatically generated

Let's calculate the list of all the vertices and the list of all the edges of our graph:

graph.add\_edge({"ab", "fg"})

graph.add\_edge({"xyz", "bla"})

print("")

print("Vertices of graph:")

print(graph.all\_vertices())

print("Edges of graph:")

print(graph.all\_edges())

Output

A black background with white text

Description automatically generated

We add a vertex and and edge to the graph:

print("Add vertex:")

graph.add\_vertex("z")

print("Add an edge:")

graph.add\_edge({"a", "d"})

print("Vertices of graph:")

print(graph.all\_vertices())

print("Edges of graph:")

print(graph.all\_edges())

Output

A black background with white text

Description automatically generated

print('Adding an edge {"x","y"} with new vertices:')

graph.add\_edge({"x","y"})

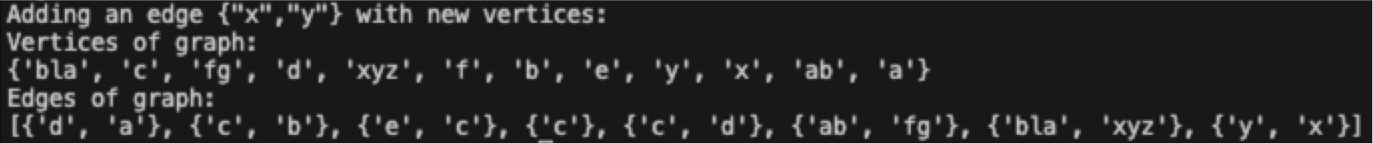
print("Vertices of graph:")

print(graph.all\_vertices())

print("Edges of graph:")

print(graph.all\_edges())

Output



1. **Paths in Graphs**

**Adjacent vertices:**

Two vertices are adjacent when they are both incident to a common edge.

**Path in an undirected Graph:**

A path in an undirected graph is a sequence of vertices such that is adjacent to for 1 ≤ i < n. Such a path P is called a path of length n from to .

**Simple Path:**

A path with no repeated vertices is called a simple path.

Example:

(a, c, e) is a simple path in our graph, as well as (a,c,e,b). (a,c,e,b,c,d) is a path but not a simple path, because the node c appears twice.

We add a method find\_path to our class Graph. It tries to find a path from a start vertex to an end vertex. We also add a method find\_all\_paths, which finds all the paths from a start vertex to an end vertex:

""" A Python Class

A simple Python graph class, demonstrating the essential

facts and functionalities of graphs.

"""

class Graph(object):

def \_\_init\_\_(self, graph\_dict=None):

""" initializes a graph object

If no dictionary or None is given,

an empty dictionary will be used

"""

if graph\_dict == None:

graph\_dict = {}

self.\_graph\_dict = graph\_dict

def edges(self, vertice):

""" returns a list of all the edges of a vertice"""

return self.\_graph\_dict[vertice]

def all\_vertices(self):

""" returns the vertices of a graph as a set """

return set(self.\_graph\_dict.keys())

def all\_edges(self):

""" returns the edges of a graph """

return self.\_\_generate\_edges()

def add\_vertex(self, vertex):

""" If the vertex "vertex" is not in

self.\_graph\_dict, a key "vertex" with an empty

list as a value is added to the dictionary.

Otherwise nothing has to be done.

"""

if vertex not in self.\_graph\_dict:

self.\_graph\_dict[vertex] = []

def add\_edge(self, edge):

""" assumes that edge is of type set, tuple or list;

between two vertices can be multiple edges!

"""

edge = set(edge)

vertex1, vertex2 = tuple(edge)

for x, y in [(vertex1, vertex2), (vertex2, vertex1)]:

if x in self.\_graph\_dict:

self.\_graph\_dict[x].append(y)

else:

self.\_graph\_dict[x] = [y]

def \_\_generate\_edges(self):

""" A static method generating the edges of the

graph "graph". Edges are represented as sets

with one (a loop back to the vertex) or two

vertices

"""

edges = []

for vertex in self.\_graph\_dict:

for neighbour in self.\_graph\_dict[vertex]:

if {neighbour, vertex} not in edges:

edges.append({vertex, neighbour})

return edges

def \_\_iter\_\_(self):

self.\_iter\_obj = iter(self.\_graph\_dict)

return self.\_iter\_obj

def \_\_next\_\_(self):

""" allows us to iterate over the vertices """

return next(self.\_iter\_obj)

def \_\_str\_\_(self):

res = "vertices: "

for k in self.\_graph\_dict:

res += str(k) + " "

res += "\nedges: "

for edge in self.\_\_generate\_edges():

res += str(edge) + " "

return res

def find\_path(self, start\_vertex, end\_vertex, path=None):

""" find a path from start\_vertex to end\_vertex

in graph """

if path == None:

path = []

graph = self.\_graph\_dict

path = path + [start\_vertex]

if start\_vertex == end\_vertex:

return path

if start\_vertex not in graph:

return None

for vertex in graph[start\_vertex]:

if vertex not in path:

extended\_path = self.find\_path(vertex,

end\_vertex,

path)

if extended\_path:

return extended\_path

return None

def find\_all\_paths(self, start\_vertex, end\_vertex, path=[]):

""" find all paths from start\_vertex to

end\_vertex in graph """

graph = self.\_graph\_dict

path = path + [start\_vertex]

if start\_vertex == end\_vertex:

return [path]

if start\_vertex not in graph:

return []

paths = []

for vertex in graph[start\_vertex]:

if vertex not in path:

extended\_paths = self.find\_all\_paths(vertex,

end\_vertex,

path)

for p in extended\_paths:

paths.append(p)

return paths

We check in the following the way of working of our find\_path and find\_all\_paths methods.

g = { "a" : {"d"},

"b" : {"c"},

"c" : {"b", "c", "d", "e"},

"d" : {"a", "c"},

"e" : {"c"},

"f" : {}

}

graph = Graph(g)

print("Vertices of graph:")

print(graph.all\_vertices())

print("Edges of graph:")

print(graph.all\_edges())

print('The path from vertex "a" to vertex "b":')

path = graph.find\_path("a", "b")

print(path)

print('The path from vertex "a" to vertex "f":')

path = graph.find\_path("a", "f")

print(path)

print('The path from vertex "c" to vertex "c":')

path = graph.find\_path("c", "c")

print(path)

A computer screen with white text

Description automatically generated

1. **Degree and Degree Sequence**

The degree of a vertex is defined as the number of edges connected to it. ​The degree sequence of a graph is a list of vertex degrees sorted in non-increasing order.​ This information is crucial for many graph-theoretic algorithms, including those related to network connectivity and flow.

* Graph Class: Contains methods to add vertices and edges and to compute the degrees.
* add\_vertex(vertex): Adds a new vertex to the graph if it does not already exist.
* add\_edge(vertex1, vertex2): Creates an undirected edge by adding both vertices to each other's adjacency list.
* degree(vertex): Calculates the degree by returning the length of the adjacency list for the specified vertex.
* degree\_sequence(): Compiles the degree of all vertices, sorts them in non-increasing order, and returns the degree sequence.

class Graph:

def \_\_init\_\_(self):

"""Initialize an empty graph."""

self.graph\_dict = {}

def add\_vertex(self, vertex):

"""Add a vertex to the graph."""

if vertex not in self.graph\_dict:

self.graph\_dict[vertex] = []

def add\_edge(self, vertex1, vertex2):

"""Add an edge between two vertices."""

if vertex1 not in self.graph\_dict:

self.add\_vertex(vertex1)

if vertex2 not in self.graph\_dict:

self.add\_vertex(vertex2)

self.graph\_dict[vertex1].append(vertex2)

self.graph\_dict[vertex2].append(vertex1) # For undirected graph

def degree(self, vertex):

"""Return the degree of a vertex."""

if vertex in self.graph\_dict:

return len(self.graph\_dict[vertex])

return 0

def degree\_sequence(self):

"""Return the degree sequence of the graph."""

return sorted([self.degree(vertex) for vertex in self.graph\_dict], reverse=True)

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

g = Graph()

g.add\_edge("A", "B")

g.add\_edge("A", "C")

g.add\_edge("B", "C")

g.add\_edge("C", "D")

print("Degree Sequence:", g.degree\_sequence())

Output: show the degree sequence of the constructed graph based on the vertices and edges added.

