ErikGraph Documentation

Release .2

Erik Gafni

December 06, 2013

Contents

CHAPTER 1

Introduction

A company asked me to write a class that implements the following specification:

> A class Graph representing an undirected graph structure with weighted edges (i.e. a set of vertices with undirected edges connecting pairs of vertices, where each edge has a nonnegative weight). In addition to methods for adding and removing vertices, class Graph should define (at minimum) the following instance methods

```
def neighbor_vertices(self, a):
"""
Return a sequence of vertices that are neighbors of vertex a (e.g. are joined by a single edge). Rais
an exception if a is not in the graph.
"""
pass

def neighbors(self, a, b):
    """
    Return True if vertices a and b are joined by an edge, and False otherwise. Raise an exception i
    a or b are not in the graph.
    """
    pass

def minimum_weight_path(self, a, b):
    """
    Return a 2-tuple comprising the minimum-weight path connecting vertices a and b, and the associat
    Return None if no such path exists. Raise an exception if a or b are not in the graph.
    """
    pass

def minimum_edge_path(self, a, b):
    """
    Return a 2-tuple comprising the minimum-edge path connecting vertices a and b, and the associate
    of edges (e.g. a path comprising 3 edges is shorter than a path comprising 4 edges, regardless o
    Return None if no such path exists. Raise an exception if a or b are not in the graph.
    """
    pass
```

Note: This is a classic graph problem, and can be solved using Dijkstra's algorithm. My implementation runs in O(|V| + |E|) as all vertices and edges might have to be traversed.

The code is available here: https://github.com/egafni/ErikGraph.

CHAPTER 2

Install

```
$ pip install erikgraph
```

CHAPTER 3

Run Doctests

From the command line, type:

```
$ python -m doctest /path/to/erikgraph/__init__.py -v
```

or from an interactive session

```
import doctest
import erikgraph
doctest.testmod(erikgraph,verbose=True)

Trying:
G = Graph(['a'],[('a','b',2),('b','c',1)])
Expecting nothing
ok
Trying:
    'b' in G.data
Expecting:
    True
ok
...
...
10 items passed all tests:
   4 tests in __init__.Graph.__init__
   3 tests in __init__.Graph.add_edge
   4 tests in __init__.Graph.add_vertex
   4 tests in __init__.Graph.delete_edge
   5 tests in __init__.Graph.delete_vertex
   3 tests in __init__.Graph.get_edge_weight
   9 tests in __init__.Graph.minimum_edge_path
   9 tests in __init__.Graph.minimum_weight_path
   4 tests in __init__.Graph.neighbor_vertices
   4 tests in __init__.Graph.neighbors
49 tests in 15 items.
49 passed and 0 failed.
Test passed.
```

CHAPTER 4

Using erikgraph

4.1 Shortest Path

To find the shortest path and its distance, use `minimum_weight_path()`.

`minimum_edge_path()` will ignore edge weight values, and find the path between two nodes that utilizes the fewest number of total edges.

```
from erikgraph import Graph
G = Graph(edges=[
        ('a','c',10),('b','c',2),('c','f',20),
        ('a','d',1),('d','e',5),('e','f',16),
        ('e','h',1),('h','f',4),('d','h',7),
        ('d','g',2)
    ]
)
print G.minimum_weight_path('a','f')
(11, ['a', 'd', 'e', 'h', 'f'])
print G.minimum_edge_path('a','f')
(2, ['a', 'c', 'f'])
```

See the method's API for details, which has plenty of examples.

4.2 API

class `erikgraph.`**`Graph`**(*vertices*=[ ], *edges*=[ ])

A miniature graph class. Implements the following specification:

A class Graph representing an undirected graph structure with weighted edges (i.e. a set of vertices with undirected edges connecting pairs of vertices, where each edge has a nonnegative weight). In addition to methods for adding and removing vertices, class Graph should define (at minimum) the following instance methods

Property data A dictionary who's keys are vertices and values are another dictionary who's keys are neighboring nodes and values are the weight connecting them

`add_edge`(*v1*, *v2*, *weight*)

Parameters

- **v1** – One of the vertices the edge is connecting to.
- **v2** – The other vertex.
- **weight** – The weight associated with the edge.

```
>>> G = Graph()
>>> G.add_edge('a','b',1)
>>> G.data['b']['a'] == 1
True
```

add_vertex(*v*)

Parameters **v** – A vertex

```
>>> G = Graph()
>>> G.add_vertex('a')
>>> G.data['a']
{}
>>> G.data['b']
Traceback (most recent call last):
    ...
KeyError: 'b'
```

delete_edge(*v1*, *v2*)

Parameters

- **v1** – a vertex
- **v2** – a neighbor of v1

Raises VertexDoesNotExist if a or b are not in the graph

```
>>> G = Graph(edges=[('a','b',2)])
>>> G.neighbors('a','b')
True
>>> G.delete_edge('b','a')
>>> G.neighbors('a','b')
False
```

delete_vertex(*v*)

Parameters **v** – the vertex to delete

```
>>> G = Graph(['x'],[('a','b',1)])
>>> 'b' in G.data
True
>>> G.delete_vertex('b')
>>> 'b' in G.data
False
>>> 'b' in G.neighbor_vertices('a')
False
```

get_edge_weight(*a*, *b*)

Returns The weight of the edge connecting a to b

Raises VertexDoesNotExist if a or b are not in the graph

```
>>> G = Graph(edges=[('a','b',2)])
>>> G.get_edge_weight('a','b')
2
>>> G.get_edge_weight('a','x')
```

```
Traceback (most recent call last):
...
VertexDoesNotExist
```

minimum_edge_path(*a*, *b*)

Returns a 2-tuple comprising the fewest number of edges required to get from vertex a to vertex b, and the path used. Returns None of there is no path from at to b.

Raises VertexDoesNotExist if a or b are not in the graph.

```
>>> G = Graph(vertices=['x','y'],edges=[('x','y',3),('b','c',1),('a','c',2),('c','d',6),('d'
>>> G.minimum_edge_path('a','d')
(2, ['a', 'c', 'd'])
>>> G.minimum_edge_path('a','e')
(2, ['a', 'c', 'e'])
>>> G = Graph(vertices=['x','y'],edges=[('x','y',3),('b','c',1),('a','c',2),('c','d',4),('d'
>>> G.minimum_edge_path('a','e')
(2, ['a', 'c', 'e'])
>>> G.minimum_edge_path('a','x')

>>> G.minimum_edge_path('a','foo')
Traceback (most recent call last):
...
VertexDoesNotExist
>>> G = Graph(edges=[('a','c',10),('b','c',2),('c','f',20),('a','d',1),('d','e',5),('e','f',
>>> G.minimum_edge_path('a','f')
(2, ['a', 'c', 'f'])
```

minimum_weight_path(*a*, *b*)

Implementation of Single Source Shortest Path using Dijkstra's

Returns a 2-tuple comprising the minimum weight used by the shortest path from a to b, and the path used. Return None if no such path exists.

Raises VertexDoesNotExist if a or b are not in the graph.

```
>>> G = Graph(vertices=['x','y'],edges=[('x','y',3),('b','c',1),('a','c',2),('c','d',6),('d'
>>> G.minimum_weight_path('a','d')
(8, ['a', 'c', 'd'])
>>> G.minimum_weight_path('a','e')
(10, ['a', 'c', 'd', 'e'])
>>> G = Graph(vertices=['x','y'],edges=[('x','y',3),('b','c',1),('a','c',2),('c','d',4),('d'
>>> G.minimum_weight_path('a','e')
(3, ['a', 'c', 'e'])
>>> G.minimum_weight_path('a','x')

>>> G.minimum_weight_path('a','foo')
Traceback (most recent call last):
...
VertexDoesNotExist
>>> G = Graph(edges=[('a','c',10),('b','c',2),('c','f',20),('a','d',1),('d','e',5),('e','f',
>>> G.minimum_weight_path('a','f')
(11, ['a', 'd', 'e', 'h', 'f'])
```

neighbor_vertices(*a*)

Returns a sequence of vertices that are neighbors of vertex a (e.g. are joined by a single edge).

Raises VertexDoesNotExist if a is not in the graph.

```
>>> G = Graph(vertices=['a'],edges=[('b','c',1),('d','c',2)])
>>> ns = G.neighbor_vertices('c')
>>> 'b' in ns and 'd' in ns and 'a' not in ns
True
>>> G.neighbor_vertices('x')
Traceback (most recent call last):
...
VertexDoesNotExist: The vertex <x> does not exist in this graph
```

neighbors(*a*, *b*)

Returns True if vertices a and b are joined by an edge, and False otherwise.

Raises VertexDoesNotExist if a or b are not in the graph.

```
>>> G = Graph(vertices=['a'],edges=[('b','c',1)])
>>> G.neighbors('c','b')
True
>>> G.neighbors('a','c')
False
>>> G.neighbors('a','x')
Traceback (most recent call last):
...
VertexDoesNotExist: The vertex <x> does not exist in this graph
```

single_source_shortest_path(*a*, *b*, *use_weights=True*)

Implements Dijstra's algorithm to determine the shortest path between vertices a and b

Returns a 2-tuple comprising the minimum weight used by the shortest path from a to b, and the path used. Return None if no such path exists.

Raises VertexDoesNotExist if a or b are not in the graph.

vertices

A list of vertices in the graph

exception erikgraph.**VertexDoesNotExist**

CHAPTER 5

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index

e