Documentation

By defining a class Graph, we aim to represent a directed graph with a number of edges and vertices. In order to properly do that, we are using three dictionaries that represent (I) the neighbours of each vertex (totalling to the out-degree of the given vertex), (II) the transposed list containing each vertex (totalling to the in-degree of the given vertex), and (III) the costs for each edge. Additionally, we define our own exception classes (VertexError and EdgeError) in order to properly handle the errors in manipulating the graph.

The class Graph contains the following methods:

- `__init__(self, n = 0, m = 0)`

Constructor for class Graph, where n represents the numuber of vertices and m the number of edges, both of which are by default zero;

- `vertices_iterator(self)`

Returns an iterator over the set of vertices;

- `neighbours_iterator(self, vertex)`

Returns an iterator to the set of (outbound) neighbours of a vertex;

- `transposed_iterator(self, vertex)`

Returns an iterator to the set of (inbound) neighbours of a vertex;

- `edges_iterator(self)`

Returns an iterator to the set of edges;

- `is_vertex(self, vertex)`

Returns True if vertex belongs to the graph;

- `is_edge(self, vertex1, vertex2)`

Returns True if the edge from the first vertex to the second vertex belongs to the graph;

- `count_vertices(self)`

Returns the number of vertices in the graph;

- `count_edges(self)`

Returns the number of edges in the graph;

- `in_degree(self, vertex)`

Returns the number of edges with the endpoint vertex, taking as input a given (user-input) vertex;

- `out_degree(self, vertex)`

Returns the number of edges with the start point vertex, taking as input a given (user-input) vertex;

- `get_edge_cost(self, vertex1, vertex2)`

Returns the cost of an edge if it exists.

- `set_edge_cost(self, vertex1, vertex2, new_cost)`

Sets the cost of an edge in the graph if it exists;

- `add_vertex(self, vertex)`

Adds a vertex to the graph;

- `add_edge(self, vertex1, vertex2, edge_cost=0)`

Adds an edge to the graph, with a given cost; if no cost is given, the default is 0. The method raises an exception if the edge already exists, and it works by adding the edge to the set of neighbours of the first vertex;

- `remove_edge(self, vertex1, vertex2)`

Removes an edge from the graph, if it exists. The method raises an exception if the edge does not exist. It works by removing the cost of the edge, and removing the second vertex from the set of neighbours of the first vertex. Then, it removes the first vertex from the set of transposed neighbours of the second vertex;

- `remove_vertex(self, vertex)`

Removes a vertex from the graph, if it exists. It works by using an auxiliary list to store the neighbours of the vertex, and then to remove the edges between the vertex and its neighbours. It then goes through the list of transposed neighbours, and removes the edges from the transposed neighbours to the vertex. Finally, it removes the entry from the dictionary of neighbours and transposed neighbours, and finally removes the vertex from the set of vertices;

- `copy(self)`

Returns a deep copy of the graph.

Implementation

In order to implement the Graph class, we used the Python-specific dynamically allocated lists, an unordered set for vertices, as well as dictionaries to retain the the “in” and “out” neighbours, and the cost of each edge.