Graph Algorithms – Practical Work no. 1

Vlad Bogdan-Tudor, Computer Science, 1st Year, Group 917

We will define a class named **TripleDictGraph**, which will represent a directed graph. As its internal representation, **TripleDictGraph** uses 3 dictionaries, each with the following purpose:

- **dict_in** – keeps track of the "out" relationships, i.e., the vertices of the graph will be the keys and the outbound neighbours (in a list) of the corresponding vertex will be the values.
- **dict_out**– keeps track of the "in" relationships, i.e., the vertices of the graph will be the keys and the inbound neighbours (in a list) of the corresponding vertex will be the value.
- **cost** – memorizes the edge costs in the graph. The vertices are tuples representing edges, while the keys will be the corresponding cost of each edge in the graph.

Python implementation

The class TripleDictGraph will provide the following methods:

get_no_vertices(self):

Returns the number of vertices in the graph.

get_no_edges(self):

Returns the number of edges in the graph

get_all_vertices(self):

Returns all the vertices from the graph using an iterator

get_all_edges(self):

Using a generator, returns all the edges in the graph as triples in the following format (_from, _to, cost):

:param _from: - starting vertex,
:param _to: - ending index
:param cost: - the weight of the edge.

get_cost_of_edge(self, _from, _to):

Returns the weight of a given edge (given by the starting and ending vertices). If the given edge does not
exist an exception is thrown (GraphException).
:param _from: The starting vertex of the edge; integer
:param _to: The ending vertex of the edge; integer
:return: The cost of the edge <_from> -> <_to>
:preconditions: The edge <_from> -> <_to> is in the graph

is_edge_in_graph(self, _from, _to):

Returns True if there exists an edge between the 2 given vertices in the graph; False otherwise. If one of
the given vertices does not exist in the graph an exception is thrown (GraphException).
:param _from: The starting vertex of the edge; integer
:param _to: The ending vertex of the edge; integer
:return: True if the given edge exists; False otherwise

is_vertex_in_graph(self, vertex):

Checks if the given vertex exists in the graph or not
:param vertex: The vertex we want to check; integer
:return: True if the vertex is in the graph; False otherwise

get_in_degree(self, vertex):

Returns the in degree of a given vertex. If the given vertex does not exist in the graph an exception is
thrown (GraphException).
:param vertex: The vertex whose in-degree we want; integer
:return: The in-degree of <vertex>
:preconditions: The vertex exists in the graph

get_out_degree(self, vertex):

Returns the out degree of a given vertex. If the given vertex does not exist in the graph an exception is
thrown (GraphException).
:param vertex: The vertex whose out-degree we want; integer
:return: The out-degree of <vertex>
:preconditions: The vertex exists in the graph

get_outbound_neighbours(self, vertex):

Returns the outbound neighbours of a given vertex. If the given vertex does not exist in the graph an
exception is thrown (GraphException).
:param vertex: The vertex whose outbound neighbours we want; integer
:return: A generator with the outbound neighbours of the given vertex
:preconditions: The vertex exists in the graph

get_inbound_neighbours(self, vertex):

Returns the inbound neighbours of a given vertex. If the given vertex does not exist in the graph an
exception is thrown (GraphException).
:param vertex: The vertex whose inbound neighbours we want; integer
:return: A generator with the inbound neighbours of the given vertex
:preconditions: The vertex exists in the graph

get_outbound_neighbours_with_cost(self, vertex):

Using a generator, returns all of the outbound neighbours of a vertex, along with the cost of the edge from
the given vertex to its outbound neighbour.
:param vertex: The vertex whose outbound neighbours we want; integer
:return: A generator with (neighbour, cost) type pairs

get_inbound_neighbours_with_cost(self, vertex):

Using a generator, returns all of the inbound neighbours of a vertex, along with the cost of the edge from
the outbound neighbour to the given vertex.
:param vertex: The vertex whose inbound neighbours we want; integer
:return: A generator with (neighbour, cost) type pairs

change_edge_cost(self, _from, _to, new_cost):

Changes the cost of an edge given by its starting and ending vertices. If the edge does not exist in the graph an exception is thrown (GraphException).
:param _from: The starting vertex of the edge; integer
:param _to: The ending vertex of the edge; integer
:param new_cost: The new cost of the edge; integer
:preconditions: Both vertices are in the graph and there exists an edge between these 2 vertices.

add_edge(self, _from, _to, cost):

Adds an edge between 2 given vertices. If there already exists an edge between those 2 vertices in the graph
or one of the 2 given vertices is not present in the graph an exception is thrown (GraphException).
:param _from: The starting vertex of the edge; integer
:param _to: The ending vertex of the edge; integer
:param cost: The cost of the vertex; integer
:return: -
:preconditions: The edge does not already exist in the graph and both vertices are in the graph.

remove_edge(self, _from, _to):

Removes the edge between the 2 given vertices in the graph. If an edge does not exist between these 2 vertices
or one of the 2 given vertices is not present in the graph an exception is thrown (GraphException).
:param _from: The starting vertex of the edge; integer
:param _to: The ending vertex of the edge; integer
:return: -
:preconditions: The edge exists in the graph

add_vertex(self, vertex):

Adds a vertex in the graph. If the vertex is already present in the graph an exception is thrown.
:param vertex: The number of the vertex we want to add; integer
:return: -
:preconditions: The vertex does not already exist in the graph

remove_vertex(self, vertex):

Removes a vertex from the graph. If the given vertex is not present in the graph an exception is thrown.
:param vertex: The number of the vertex we want to remove; integer
:return: -
:preconditions: The vertex exists in the graph

get_copy_of_graph(self):

Returns a copy of the graph.

The following are external functions (outside of the TripleDictGraph class):

read_graph(file_name):

Reads a graph from a given file, builds this graph and returns it.
:param file_name: The name of the file where the graph is stored (should be given with extension i.e., 'txt')
:return: An instance of TripleDictGraph; the randomly generated graph

write_graph(graph, file_name):

Writes the given graph in a file.
:param graph: The graph we want to save; an instance of TripleDictGraph
:param file_name: The name of the file where we want to save the graph (should be given with extension i.e., 'txt')
:return: -

create_random_graph(no_vertices, no_edges):

Creates a random graph with <no_vertices> vertices and <no_edges> edges.
:param no_vertices: The number of vertices the graph should have; integer
:param no_edges: The number of edges the graph should have; integer
:return: An instance of TripleDictGraph; the randomly generated graph