graph.Graph Class Reference

Public Member Functions

def **__init__** (self, n=0, m=0, nodes=None, dict_in=None, dict_out=None, dict_cost=None)
def **copy_graph** (self)
def **parse_vertices** (self)
def **is_edge** (self, node_in, node_out)
def **in_degree** (self, node)
def **out_degree** (self, node)
def **parse_outbound_edges** (self, vertex)
def **parse_inbound_edges** (self, vertex)
def **modify_cost** (self, node_in, node_out, new_cost)
def **add_edge** (self, node_in, node_out, cost)
def **remove_edge** (self, node_in, node_out)
def **add_node** (self, vertex)
def **remove_node** (self, vertex)
def **read_from_file** (self, file_name)
def **write_to_file** (self, file_name)
def **get_cost** (self, n1, n2)
def **get_nr_of_vertices** (self)
def **get_nr_of_edges** (self)
def **set_nr_of_vertices** (self, n)
def **set_nr_of_edges** (self, m)
def **get_dict_in** (self)
def **get_dict_out** (self)
def **get_dict_cost** (self)
def **__str__** (self)

Static Public Member Functions

def **random_graph** (n, m)

Detailed Description

```
Class for the bidirectional graph
```

Member Function Documentation

◆ add_edge()

def graph.Graph.add_edge (self,
node_in,
node_out,
cost
)

```
Function that adds an edge to the graph
Complexity: Theta(1)
:param node_in: the first vertex
:param node_out: the second vertex
:param cost: the cost of the new edge (int)
:raise Exception: if the edge already exists, or if the nodes don't exist
:return: -
```

◆ add_node()

def graph.Graph.add_node (self,
vertex
)

```
Function that adds a node to the graph
Complexity: Theta(1)
:param vertex: the node to be added
:raise Exception: the node is invalid (it already exists)
:return: -
```

◆ copy_graph()

def graph.Graph.copy_graph (self)

```
Function that returns a copy of the graph

:return (Graph) an exact deepcopy of this graph
```

◆ in_degree()

def graph.Graph.in_degree (self,
node
)

```
Function that returns the in-degree of the given vertex
Complexity: Theta(1)
:param node: the node to compute the in-degree of
:raise Exception: if the node doesn't exist
:return: the in-degree of the given node (int)
```

◆ is_edge()

```
def graph.Graph.is_edge ( self,
                          node_in,
                          node_out
                        )
```

```
Function that checks if (node_in, node_out) is an edge
Complexity: Theta(1)
:param node_in: the first vertex
:param node_out: the second vertex
:return true if (node_1, node_2) is a vertex, false otherwise
```

◆ modify_cost()

```
def graph.Graph.modify_cost ( self,
                              node_in,
                              node_out,
                              new_cost
                            )
```

```
Function that modifies the cost of the given edge
Complexity: Theta(1)
:param node_in: the first vertex
:param node_out: the second vertex
:param new_cost: the new cost of the edge (int)
:raise Exception: if the edge doesn't exist
:return: -
```

◆ out_degree()

```
def graph.Graph.out_degree ( self,
                             node
                           )
```

```
Function that returns the out-degree of the given vertex
Complexity: Theta(1)
:param node: the node to compute the out-degree of
:raise Exception: if the node doesn't exist
:return: the out-degree of the given node (int)
```

◆ parse_inbound_edges()

```
def graph.Graph.parse_inbound_edges ( self,
                                      vertex
                                    )
```

```
Function that returns an iterator through the inbound edges of the vertex
Complexity: Theta(1)
:param vertex: the node to parse through
:raise Exception: if the node doesn't exist
:return: an iterator through the list of vertices that are inbound connected to the vertex
```

◆ parse_outbound_edges()

def graph.Graph.parse_outbound_edges (self,
vertex
)

```
Function that returns an iterator through the outbound edges of the vertex
Complexity: Theta(1)
:param vertex: the node to parse through
:raise Exception: if the node doesn't exist
:return: an iterator through the list of vertices that are outbound connected to the vertex
```

◆ parse_vertices()

def graph.Graph.parse_vertices (self)

```
Function that returns an iterator to the vertices list

:return an iterator through the list of vertices
```

◆ random_graph()

def graph.Graph.random_graph (n,
m
)

static

```
Function that creates a random graph
:param n: the number of vertices
:param m: the number of edges
:return: a graph
```

◆ read_from_file()

def graph.Graph.read_from_file (self,
file_name
)

```
Function that reads a graph from a file
Complexity: O(n)
:param file_name: the file to read from
:return: -
```

◆ remove_edge()

def graph.Graph.remove_edge (self,
node_in,
node_out
)

```
Function that removes an edge from the graph
Complexity: Theta(1)
:param node_in: the first vertex
:param node_out: the second vertex
:raise Exception: if the edge doesn't exist
:return: -
```

◆ remove_node()

```
def graph.Graph.remove_node ( self,
                              vertex
                            )
```

```
Function that removes a node from the graph
Complexity: O(n)
:param vertex: the node to be removed
:raise Exception: if the vertex doesn't exist
:return: -
```

◆ write_to_file()

```
def graph.Graph.write_to_file ( self,
                                file_name
                              )
```

```
Function that writes to a file the graph
Complexity: O(n)
:param file_name: the file to write into
:return: -
```

The documentation for this class was generated from the following file:

- graph.py