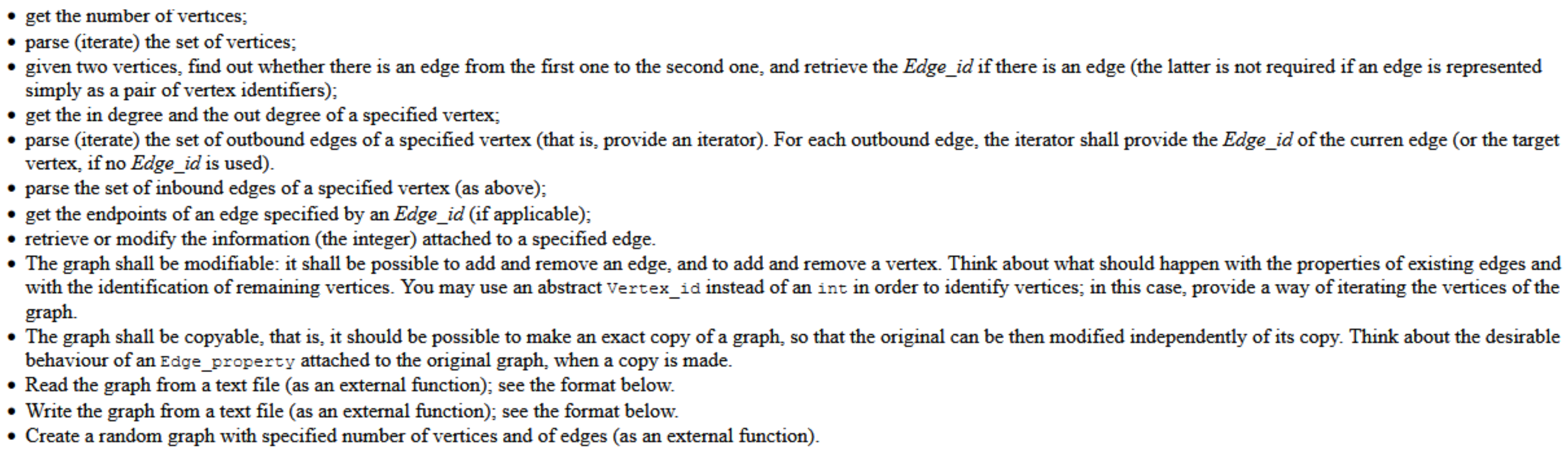
Timofte Razvan Mihai

Assignment 1 – 31.03.2022 - Week 6

The program has the following required operations:



- get the number of vertices;
- parse (iterate) the set of vertices;
- given two vertices, find out whether there is an edge from the first one to the second one, and retrieve the *Edge_id* if there is an edge (the latter is not required if an edge is represented simply as a pair of vertex identifiers);
- get the in degree and the out degree of a specified vertex;
- parse (iterate) the set of outbound edges of a specified vertex (that is, provide an iterator). For each outbound edge, the iterator shall provide the *Edge_id* of the curren edge (or the target vertex, if no *Edge_id* is used).
- parse the set of inbound edges of a specified vertex (as above);
- get the endpoints of an edge specified by an *Edge_id* (if applicable);
- retrieve or modify the information (the integer) attached to a specified edge.
- The graph shall be modifiable: it shall be possible to add and remove an edge, and to add and remove a vertex. Think about what should happen with the properties of existing edges and with the identification of remaining vertices. You may use an abstract `Vertex_id` instead of an `int` in order to identify vertices; in this case, provide a way of iterating the vertices of the graph.
- The graph shall be copyable, that is, it should be possible to make an exact copy of a graph, so that the original can be then modified independently of its copy. Think about the desirable behaviour of an `Edge_property` attached to the original graph, when a copy is made.
- Read the graph from a text file (as an external function); see the format below.
- Write the graph from a text file (as an external function); see the format below.
- Create a random graph with specified number of vertices and of edges (as an external function).

As such all we need is a Graph module where we define the Graph class & an exception class for exception-handling, GraphError

Each function has a specification already, but they will also be mentioned here:

<u>**MAIN** module</u>

# Build function from a file, takes values from a file, splits them and then builds the graph using them

def build_graph_file(filename):

# Displays all the edges

def display_edges(graph):

# Displays the menu

def display_menu():

# Writes the graph to a file

def write_graph_file(graph, filename):

# Builds a random graph using the random library

def build_random_graph(n, m):

# The main function. Infinite loop to make sure the file name given exists // Infinite loop to keep going through the

# options given // Every option is numeric & self-explanatory

def main():

GRAPH module

# Basic exception class taught in FP

class GraphError(Exception):

# Creates the GraphError exception handler

def __init__(self, message=""):

# Graph class

class Graph:

# The graph is created, the in / out vertices being dictionaries of lists

def __init__(self, n=0):

PARSING

# Parses the vertices, shown during seminar

def parse_vertices(self):

# Parses the edges of the graph

def parse_edges(self):

# Parses the out vertices of the given vertex

def parse_out(self, x):

# Parses the in vertices of the given vertex

def parse_in(self, x):

CHECKS

# Checks if the given value is actually a vertex

def is_vertex(self, x):

# Checks if the given values are actually an edge

def is_edge(self, x, y):

# Returns the in degree of the given vertex

def in_degree(self, x):

# Returns the out degree of the given vertex

def out_degree(self, x):

ADD/DELETE

```
# Basic add function that adds an edge with a cost
def add_edge(self, x, y, cost = 0):
# Basic delete function that deletes the edge after it finds it
def delete_edge(self, x, y):
# Basic add function that adds a vertex
def add_vertex(self, x):
# Basic delete function that deletes a vertex
def delete_vertex(self, x):
```

GETTERS/SETTERS

```
# Returns the cost of an edge
def get_cost(self, x, y):
# Sets the cost of an edge
def set_cost(self, x, y, new_cost):
# Returns the number of vertices
def get_vertices(self):
# Returns the number of edges
def get_edges(self):
# Copies the graph
def __copy__(self):
```

Implementation, Summary:

In essence, a graph object is built, which uses dictionaries to handle the vertices and edges, with the keys being the vertices. Each edge is a tuple. From there, every function is quite direct & non-complex, with each having few lines of code. To sum it up, the functions were similar to the basics we were taught during FP, OOP and even during the GraphAlgo seminaries.

The complexity is also as required.

Credit: For this assignment, I looked over and learned from our professor's notes during the Seminary, during which he had programmed a basic graph class. I altered the representation, the way some functions work and also added new ones per the requirement, but the base is quite similar. I also made use of past projects during Fundamentals of Programming.