**Iaguta Alen Mihael**

**Gr. 913**

The project contains the implementation of the following functions:

* Add, remove, print vertices, print no. of vertices, inbound/outbound edges, parse vertices, print degrees
* Add, remove, print edges, print costs, change costs, get costs, get edges, is edge
* Load and write, copy a graph and create a random graph

Specification:

Main class: class SetGraph:

def \_\_init\_\_(self, vertices):

self.inbound\_dict = {}

self.outbound\_dict = {}

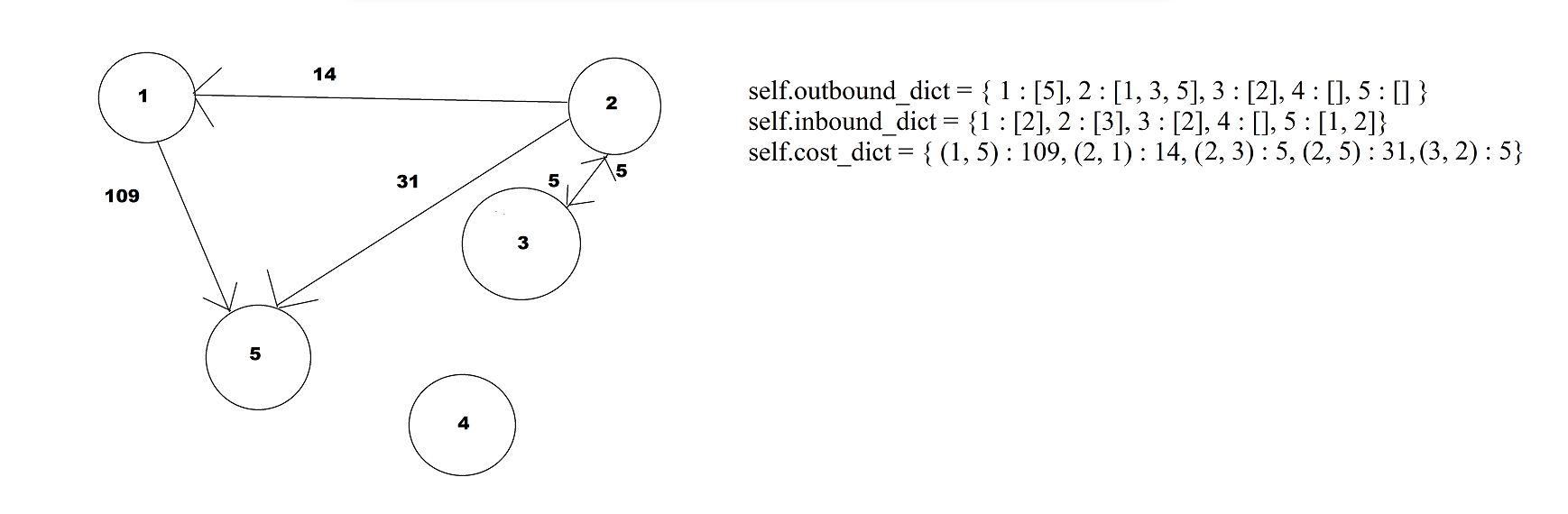
self.cost\_dict = {}

for i in range(vertices):

self.inbound\_dict[i] = []

self.outbound\_dict[i] = []

def get\_all\_edges(self):  
 edges = []  
 for edge in self.cost\_dict:  
 edges.append(edge)  
 return edges[:]  
  
def is\_edge(self, start\_vertex, end\_vertex):  
 try:  
 return end\_vertex in self.outbound\_dict[start\_vertex]  
 except KeyError:  
 raise GraphException(f"({start\_vertex}, {end\_vertex}) is not an edge")  
  
def add\_edge(self, start\_vertex, end\_vertex, cost\_of\_edge):  
 err = ""  
 if self.is\_edge(start\_vertex, end\_vertex):  
 err = err + "Already an edge"  
 if len(err):  
 raise GraphException(err)  
 self.outbound\_dict[start\_vertex].append(end\_vertex)  
 self.inbound\_dict[end\_vertex].append(start\_vertex)  
 self.cost\_dict[(start\_vertex, end\_vertex)] = cost\_of\_edge  
  
def remove\_edge(self, start\_vertex, end\_vertex):  
 if not self.is\_edge(start\_vertex, end\_vertex):  
 raise GraphException("Edge doesn't exist")  
 del self.cost\_dict[(start\_vertex, end\_vertex)]  
 self.outbound\_dict[start\_vertex].remove(end\_vertex)  
 self.inbound\_dict[end\_vertex].remove(start\_vertex)  
  
"""  
Functions for COST  
"""  
  
def get\_cost(self, start\_vertex, end\_vertex):  
 if self.is\_edge(start\_vertex, end\_vertex):  
 return self.cost\_dict[(start\_vertex, end\_vertex)]  
  
def get\_all\_costs(self):  
 costs = []  
 for cost in self.cost\_dict:  
 costs.append(self.cost\_dict[cost])  
 return costs[:]  
  
def change\_cost(self, start\_vertex, end\_vertex, new\_cost):  
 if not self.is\_edge(start\_vertex, end\_vertex):  
 raise GraphException("Edge doesn't exist")  
 self.cost\_dict[(start\_vertex, end\_vertex)] = new\_cost  
"""  
Functions for VERTICES  
"""  
def get\_vertices(self):  
 return list(self.outbound\_dict.keys())  
  
def get\_number\_of\_vertices(self):  
 return len(self.get\_vertices())  
  
def get\_isolated\_vertices(self):  
 vertices = []  
 for vertex in self.get\_vertices():  
 if self.inbound\_dict[vertex]==[] and self.outbound\_dict[vertex]==[]:  
 vertices.append(vertex)  
 return vertices[:]  
  
def get\_degrees(self, vertex):  
 try:  
 return len(self.inbound\_dict[vertex]),len(self.outbound\_dict[vertex])  
 except KeyError:  
 GraphException("Vertex doesn't exist")  
  
def add\_vertex(self, vertex):  
 if vertex in self.get\_vertices():  
 raise GraphException("Vertex already exists")  
 self.outbound\_dict[vertex] = []  
 self.inbound\_dict[vertex] = []  
  
def remove\_vertex(self, vertex):  
 if vertex not in self.get\_vertices():  
 raise GraphException("Vertex doesn't exist")  
 for i in self.outbound\_dict[vertex]:  
 self.inbound\_dict[i].remove(vertex)  
 del self.cost\_dict[(vertex, i)]  
 del self.outbound\_dict[vertex]  
 for i in self.inbound\_dict[vertex]:  
 self.outbound\_dict[i].remove(vertex)  
 del self.cost\_dict[(i, vertex)]  
 del self.inbound\_dict[vertex]  
  
def copy\_graph(self):  
 new\_graph = SetGraph(len(self.get\_vertices()))  
 new\_graph.inbound\_dict = copy.deepcopy(self.inbound\_dict)  
 new\_graph.outbound\_dict = copy.deepcopy(self.outbound\_dict)  
 new\_graph.cost\_dict = copy.deepcopy(self.cost\_dict)  
 return new\_graph



RandomGraph(vertices, edges) generates a random directed graph.

class RandomGraph:  
  
 def \_\_init\_\_(self, vertices, edges):  
 self.graph = SetGraph(vertices)  
 self.generate(vertices, edges)  
  
 def generate(self, vertices, edges):  
 list\_of\_vertices = [i for i in range(vertices)]  
 costs = [0]  
 for i in range(1, 101):  
 costs.append(i)  
 costs.append(-i)  
 index = 0  
 while index < edges:  
 start = random.choice(list\_of\_vertices)  
 end = random.choice(list\_of\_vertices)  
 cost = random.choice(costs)  
 try:  
 self.graph.add\_edge(start, end, cost)  
 index = index + 1  
 except GraphException:  
 pass  
 return self.graph  
  
 def print\_random\_graph(self):  
 for edge in self.graph.get\_all\_edges():  
 print(f"{edge} with cost {self.graph.get\_cost(edge[0],edge[1])}")

The UI module controls the ui and the menu.