INTELLIGENT SYSTEMS – GROUP A3

LAB TASK1

UNIVERSIDAD DE CASTILLA~LA MANCHA

INDEX

GITHUB REPOSITORY
https://github.com/jupcan/osm-gps

COMPONENTES DEL GRUPO
Juan Perea Campos
Samuel González Linde
Javier Abengózar Palop

1. TASK 1 DOCUMENTATION

1.1 APLICATION REQUIREMENTS

The main goal of the laboratory assignment is to implement a search algorithm in order to obtain an optimal route for a vehicle that circulates through a set of places of a town. The route must pass through several concrete places.

We are going to talk about NODES and ARCS. The application will be fed of the essential information by XML files generated from OpenStreetMap (OSM) and converted to graphml format. So, the first requirement Task 1 is the programming of classes that allows us to represent a map with the information of an XML file. The class named '*graph*' will contain three methods called '*belongNode'*, '*positionNode*' and '*adjacentNodes*'.

1.2 PROGRAMMING LANGUAGE

Since we could choose whatever language to work on, several decisions had to take place before starting to work. At first, we decided to use java language as it has been the familiar one, but we changed opinion when we found that other languages could improve the way of working with data structures, so the final choice was **python**. Also, the simply way the aforementioned language provides to do basic programming methodology will allow us to focus on the main issues of the code related with the artificial intelligent field.

1.3 STRUCTURES OF CREATED ARTIFACTS

We will use three important libraries for this task that will allow us to build the proper structure, manage files information and print data structures with a desired format, essential to solve the problem. The first one is '***lxml***' which contains the element etree, enabling the XML file reading and its analysis to build a tree that we will be using as base to create and store all the data in our chosen data structure. To solve the main goal of the laboratory part, this data structure is essential, so we decided to import a library able to build this kind of abstraction. The second imported library is just one we decided to use in order to print the file name we are using each time in a simple way, it is called '***os.path***'. And the third library is '***pprint***' that will help us to print in a pretty way data structures.

```
from lxml import etree
from os.path import basename
from pprint import pprint
```

GRAPH CLASS

To achieve the objective we have created a class named graph(). It contains the needed variables and methods to read XML files, check if a node belongs to a graph and its position and the adjacent edges of an specific node.

```
class graph():
    _path = ""
    def __init__(self, path):
        self._path = path
        self._keys, self._nodes, self._edges = self._readFile()
```

This class has as attributes:

'_path' : path of the XML file containing the map information.
'_keys' : a dictionary containing all keys information which appear in the XML files and help us as a guide for all they types of keys that exists.
'_nodes' : a dictionary containing all nodes that conform the graph.
'_edges' : a dictionary containing all edges that conform the graph.

The path will be the parameter use in the constructor to create each graph.

This class has four methods:

1. **readFile()**

This method takes care for the reading of the XML file containing the essential information needed to create the tree. The previous mentioned element from the *lxml* library etree can build a tree with *.parse(path)* receiving as a parameter the path of the file. With the method *.getroot()* we get the tree's root (and later its nodes). The method is defined as a way to represent the keys, nodes and edges in a structure as it can be a dictionary in order to access to a specific element in an easy way. It collects the keys from the XML files and store them in the dictionaries. This method returns three dictionaries containing keys, nodes and edges respectively.

```python
    def _readFile(self):
        data = etree.parse(self._path)
        root = data.getroot()
        ns = {'n': 'http://graphml.graphdrawing.org/xmlns'}

        keys, nodes, edges = {}, {}, {}
        print(basename(self._path)) #print file name
        for key in root: #get desired key values for each file
            keys[(key.get('attr.name'), key.get('for'))] = (key.get('id'))
        key_y = keys[('y', 'node')]
        key_x = keys[('x', 'node')]
        key_name = keys[('name', 'edge')]
        key_length = keys[('length', 'edge')]

        for node in root.findall('n:graph/n:node', ns):
            data = dict((d.get('key'), d.text) for d in node)
            values = (data.get(key_y), data[key_x])
            nodes[node.get('id')] = values

        for edge in root.findall('n:graph/n:edge', ns):
            data = dict((d.get('key'), d.text) for d in edge)
            values = (data.get(key_name, 'sinNombre'), data[key_length])
            edges[(edge.get('source'), edge.get('target'))] = values
        return keys, nodes, edges
```

2. **belongNode()**
This method checks if a node belongs to the graph, with the argument ‘id’ as a parameter representing it the node to be checked.

```python
    def belongNode(self, id):
        #input: osm node id, output: true/false
        if id in self._nodes:
            return True
        else:
            return False
```

3. **positionNode()**
The method checks if a node belongs to the graph by using the previous created method and if so it returns its coordinates, latitude and longitude (x,y). It also receives the ‘id’ of a node as a parameter.

```python
    def positionNode(self, id):
        #input: osm node id, output: latitude&longitude[(y,x)]
        try:
            node_exists = self.belongNode(id)
            if node_exists:
                print([self._nodes[id]])
            else:
                raise ValueError
        except ValueError:
            print("Error. The node does not exist.")
```

4. adjacentNode()
Receives the ‘id’ of a node as a parameter, checks if it belongs to the graph we are using and, if so it prints a list of the adjacent arcs for the given node. The output is also a dictionary in which the key for each street of the map is made up of the tuple (origin, target) since this must be always *UNIQUE*. A given node id could be repeated for several arcs.

```
    def adjacentNode(self, id):
        #input: osm node id, output: list of adjacent arcs
        try:
            node_exists = self.belongNode(id)
            streets = {}
            if node_exists:
                adjacents = [key for key in self._edges.keys() if id in key[0]]
                for data in adjacents:
                    streets[data] = tuple(self._edges[data])
                pprint(streets)
            else:
                raise ValueError
        except ValueError:
            print("Error. The node does not exist.")
```

MAIN CLASS

In the main class we only create an object of the class '*graph*' and call its methods to give us the results, we also check the access time to any of the dictionaries created. We have also added a way to read from the input and write one more node at a time by separating it by comas, to check the correctness of the results in a faster way.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
from graph import graph
import time

def main():
    try:
        filename = input('file: ')
        if filename.isdigit():
            raise ValueError
        town1 = graph('data/%s.graphml.xml' % filename)
        nodes = [i for i in input('nodes: ').split(',')]
        for node in nodes:
            start_time = time.time()
            print(town1.belongNode(node))
            town1.positionNode(node)
            town1.adjacentNode(node)
            print("%s seconds" % (time.time() - start_time))

    except ValueError:
        print ("Error. Not a valid input.")

if __name__ == '__main__':
    main()
```

When executing in the command line, an input example could be the following one:

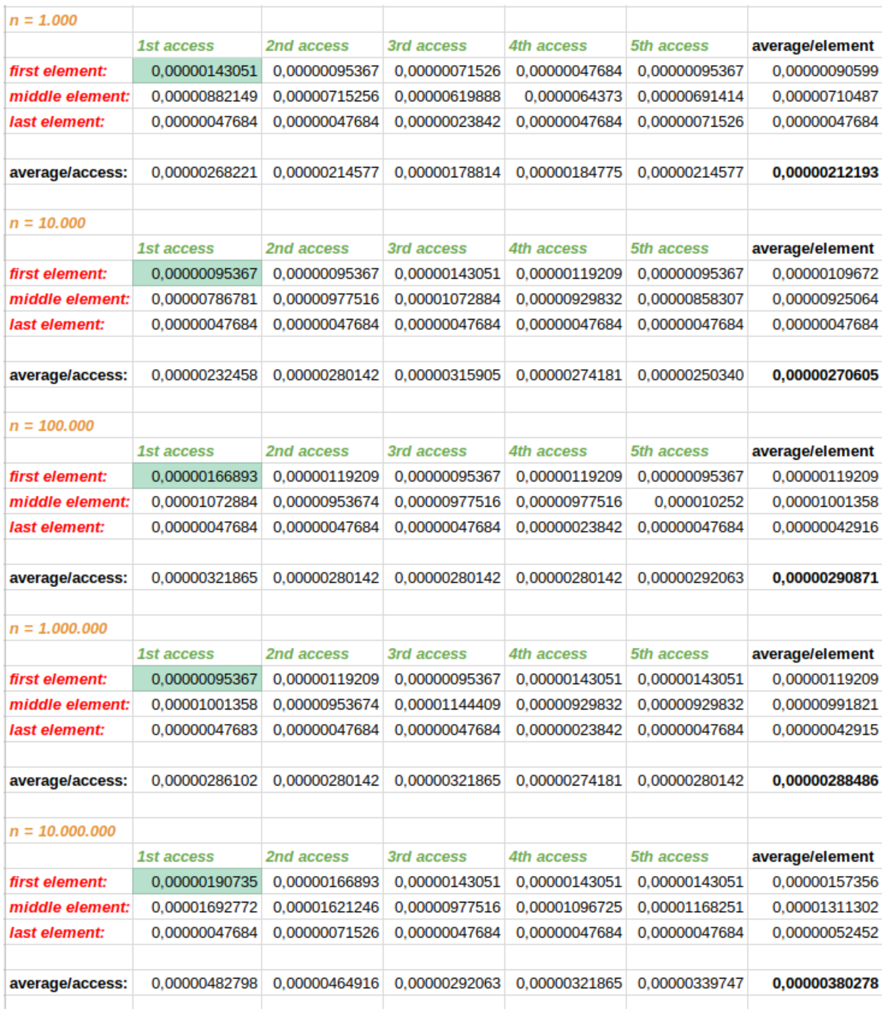
```
file: ciudad real.graphml.xml
nodes: 796725819,765309509,827212358
```

1.4 JUSTIFICATION OF USED ARTIFACTS - ACCESSING TIMES

- Number of elements (n): 1.000, 10.000, 100.000, 1.000.000 and 10.000.000:
- Checking of the access time for n element sized list
- 5 access tests for each n, mean of access
- Average list access time

Working with the first element of the list, mid element of the list and final element of the list in order to take the values of the accessing time in different positions. The same properties described before will be used to test the accessing times of the dictionaries.

LIST TIMES



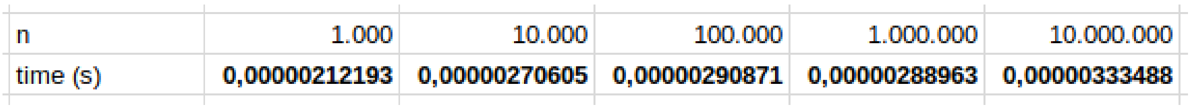
n = 1.000	1st access	2nd access	3rd access	4th access	5th access	average/element
first element:	0,00000143051	0,00000095367	0,00000071526	0,00000047684	0,00000095367	0,00000090599
middle element:	0,00000882149	0,00000715256	0,00000619888	0,0000064373	0,00000691414	0,00000710487
last element:	0,00000047684	0,00000047684	0,00000023842	0,00000047684	0,00000071526	0,00000047684
average/access:	0,00000268221	0,00000214577	0,00000178814	0,00000184775	0,00000214577	**0,00000212193**

n = 10.000	1st access	2nd access	3rd access	4th access	5th access	average/element
first element:	0,00000095367	0,00000095367	0,00000143051	0,00000119209	0,00000095367	0,00000109672
middle element:	0,00000786781	0,00000977516	0,00001072884	0,00000929832	0,00000858307	0,00000925064
last element:	0,00000047684	0,00000047684	0,00000047684	0,00000047684	0,00000047684	0,00000047684
average/access:	0,00000232458	0,00000280142	0,00000315905	0,00000274181	0,00000250340	**0,00000270605**

n = 100.000	1st access	2nd access	3rd access	4th access	5th access	average/element
first element:	0,00000166893	0,00000119209	0,00000095367	0,00000119209	0,00000095367	0,00000119209
middle element:	0,00001072884	0,00000953674	0,00000977516	0,00000977516	0,000010252	0,00001001358
last element:	0,00000047684	0,00000047684	0,00000047684	0,00000023842	0,00000047684	0,00000042916
average/access:	0,00000321865	0,00000280142	0,00000280142	0,00000280142	0,00000292063	**0,00000290871**

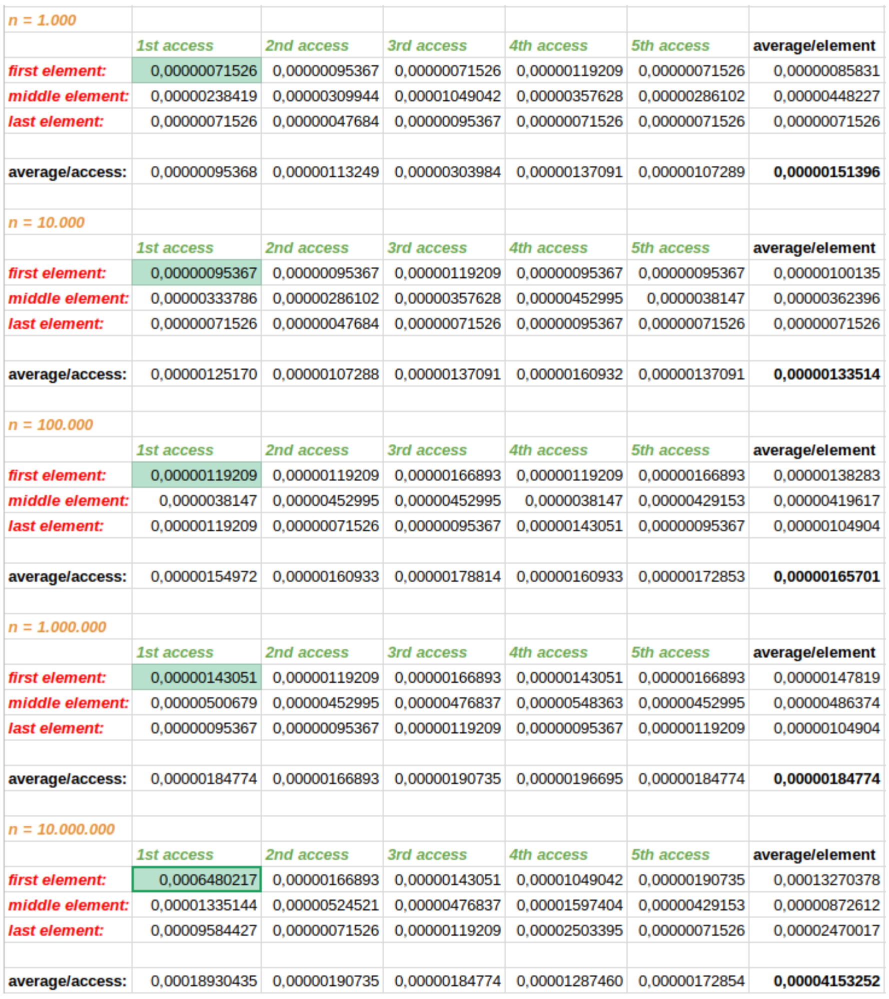
n = 1.000.000	1st access	2nd access	3rd access	4th access	5th access	average/element
first element:	0,00000095367	0,00000119209	0,00000095367	0,00000143051	0,00000143051	0,00000119209
middle element:	0,00001001358	0,00000953674	0,00001144409	0,00000929832	0,00000929832	0,00000991821
last element:	0,00000047683	0,00000047684	0,00000047684	0,00000023842	0,00000047684	0,00000042915
average/access:	0,00000286102	0,00000280142	0,00000321865	0,00000274181	0,00000280142	**0,00000288486**

n = 10.000.000	1st access	2nd access	3rd access	4th access	5th access	average/element
first element:	0,00000190735	0,00000166893	0,00000143051	0,00000143051	0,00000143051	0,00000157356
middle element:	0,00001692772	0,00001621246	0,00000977516	0,00001096725	0,00001168251	0,00001311302
last element:	0,00000047684	0,00000071526	0,00000047684	0,00000047684	0,00000047684	0,00000052452
average/access:	0,00000482798	0,00000464916	0,00000292063	0,00000321865	0,00000339747	**0,00000380278**

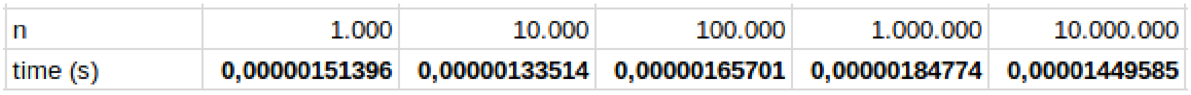


n	1.000	10.000	100.000	1.000.000	10.000.000
time (s)	**0,00000212193**	**0,00000270605**	**0,00000290871**	**0,00000288963**	**0,00000333488**

DICTIONARIES TIMES



n = 1.000						
	1st access	*2nd access*	*3rd access*	*4th access*	*5th access*	**average/element**
first element:	0,00000071526	0,00000095367	0,00000071526	0,00000119209	0,00000071526	0,00000085831
middle element:	0,00000238419	0,00000309944	0,00001049042	0,00000357628	0,00000286102	0,00000448227
last element:	0,00000071526	0,00000047684	0,00000095367	0,00000071526	0,00000071526	0,00000071526
average/access:	0,00000095368	0,00000113249	0,00000303984	0,00000137091	0,00000107289	**0,00000151396**
n = 10.000						
	1st access	*2nd access*	*3rd access*	*4th access*	*5th access*	**average/element**
first element:	0,00000095367	0,00000095367	0,00000119209	0,00000095367	0,00000095367	0,00000100135
middle element:	0,00000333786	0,00000286102	0,00000357628	0,00000452995	0,0000038147	0,00000362396
last element:	0,00000071526	0,00000047684	0,00000071526	0,00000095367	0,00000071526	0,00000071526
average/access:	0,00000125170	0,00000107288	0,00000137091	0,00000160932	0,00000137091	**0,00000133514**
n = 100.000						
	1st access	*2nd access*	*3rd access*	*4th access*	*5th access*	**average/element**
first element:	0,00000119209	0,00000119209	0,00000166893	0,00000119209	0,00000166893	0,00000138283
middle element:	0,0000038147	0,00000452995	0,00000452995	0,0000038147	0,00000429153	0,00000419617
last element:	0,00000119209	0,00000071526	0,00000095367	0,00000143051	0,00000095367	0,00000104904
average/access:	0,00000154972	0,00000160933	0,00000178814	0,00000160933	0,00000172853	**0,00000165701**
n = 1.000.000						
	1st access	*2nd access*	*3rd access*	*4th access*	*5th access*	**average/element**
first element:	0,00000143051	0,00000119209	0,00000166893	0,00000143051	0,00000166893	0,00000147819
middle element:	0,00000500679	0,00000452995	0,00000476837	0,00000548363	0,00000452995	0,00000486374
last element:	0,00000095367	0,00000095367	0,00000119209	0,00000095367	0,00000119209	0,00000104904
average/access:	0,00000184774	0,00000166893	0,00000190735	0,00000196695	0,00000184774	**0,00000184774**
n = 10.000.000						
	1st access	*2nd access*	*3rd access*	*4th access*	*5th access*	**average/element**
first element:	0,0006480217	0,00000166893	0,00000143051	0,00001049042	0,00000190735	0,00013270378
middle element:	0,00001335144	0,00000524521	0,00000476837	0,00001597404	0,00000429153	0,00000872612
last element:	0,00009584427	0,00000071526	0,00000119209	0,00002503395	0,00000071526	0,00002470017
average/access:	0,00018930435	0,00000190735	0,00000184774	0,00001287460	0,00000172854	**0,00004153252**



n	1.000	10.000	100.000	1.000.000	10.000.000
time (s)	**0,00000151396**	**0,00000133514**	**0,00000165701**	**0,00000184774**	**0,00001449585**

GRAPHICAL RESULTS

Figure 1

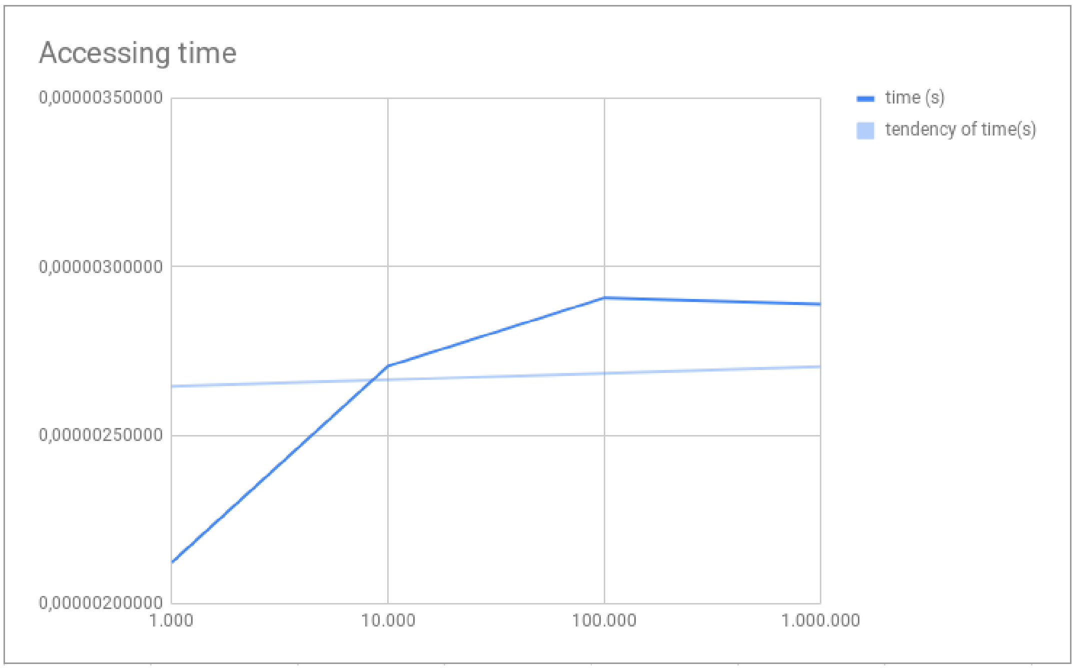
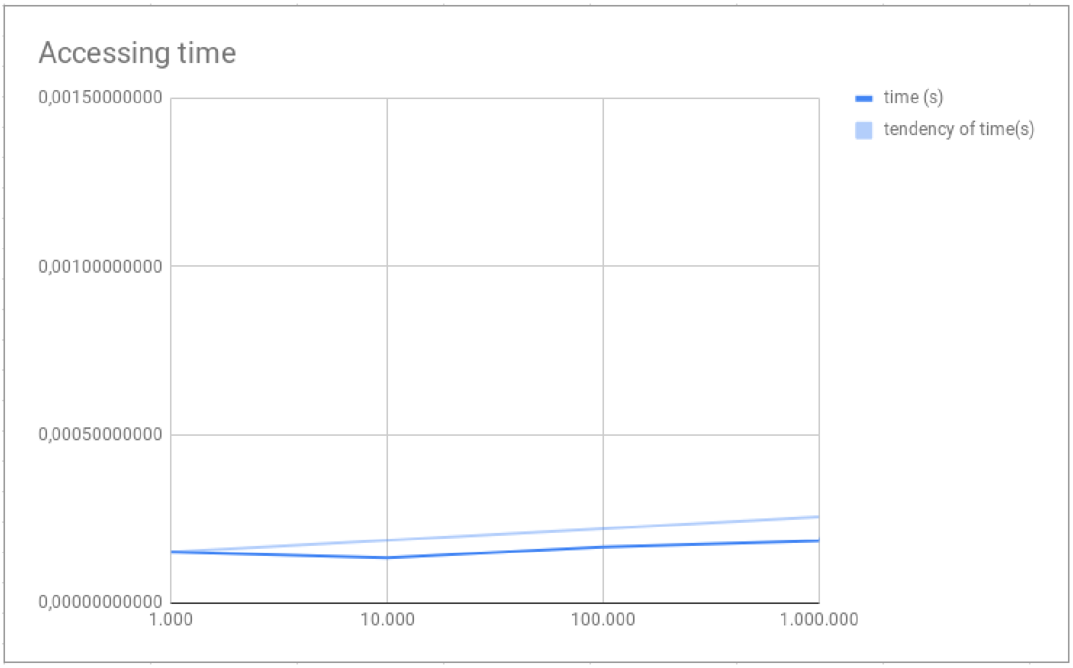


Figure 2

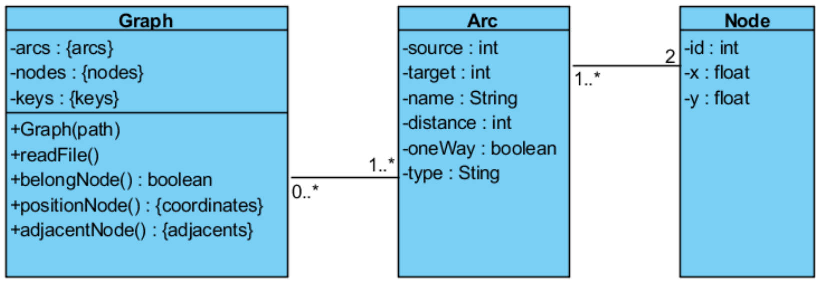


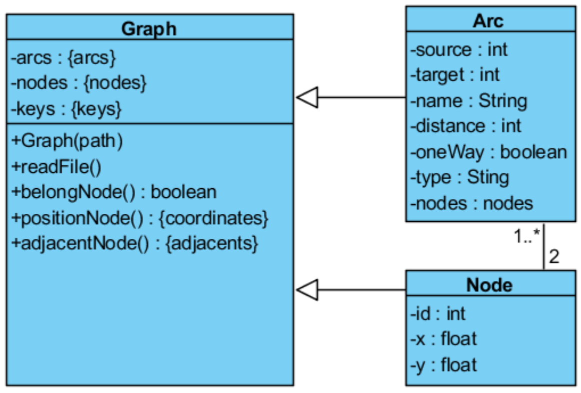
As we can see, comparing both figures being the first one for list data structures and the second one for dictionaries, last mentioned ones have a lower accessing time to data that will be key for us since when our program starts to be more complex and doing searches, the time could grow exponentially causing lots of memory problems.

That's the **main reason** has led us to **use dictionaries instead of list**. At first, we used list and look for the desired data by means of a sequential access which was not efficient enough, in python we can also access a list member by checking its index but the language behind the scenes, iterates through the list until it finds the number or reaches the end of the list so it's the same as going across all elements in a '*for*', however in a dictionary python can attempt to directly access the target number in the set, rather than iterate through every item in the list and compare every item to the target number.

1.5 UML SKETCHES

This does not clearly represent the implementation of our code but was used as first draft to start thinking about our goals so we found it interesting to also show it over here.





1.6 BIBLIOGRAPHY

- https://docs.quantifiedcode.com/python-anti-patterns/performance/using_key_in_list_to_check_if_key_is_contained_in_a_list.html
- https://stackoverflow.com/