INTELLIGENT SYSTEMS – LAB GROUP A3

OSM-GPS

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 APPLICATION 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 our most used programming language, but the decision changed 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 a 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 the type 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.

It also 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 to represent the keys, nodes and edges in a dictionary 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.

```
    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.

```
    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.

```
    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)	**0,00212192600**	**0,00270605050**	**0,00290870650**	**0,00288486350**	**0,00380277650**

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	0,00000149042	0,00000357628	0,00000286102	0,00000268227
last element:	0,00000071526	0,00000047684	0,00000095367	0,00000071526	0,00000071526	0,00000071526
average/access:	0,00000095368	0,00000113249	0,00000078984	0,00000137091	0,00000107289	**0,00000106396**
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:	0,00006480217	0,00000166893	0,00000143051	0,00001049042	0,00000190735	0,00001605988
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:	0,00004349947	0,00000190735	0,00000184774	0,00001287460	0,00000172854	**0,00001237154**

n	1.000	10.000	100.000	1.000.000	10.000.000
time (s)	**0,00106395900**	**0,00133514350**	**0,00165700800**	**0,00184774250**	**0,01237154000**

GRAPHICAL RESULTS

Figure 1

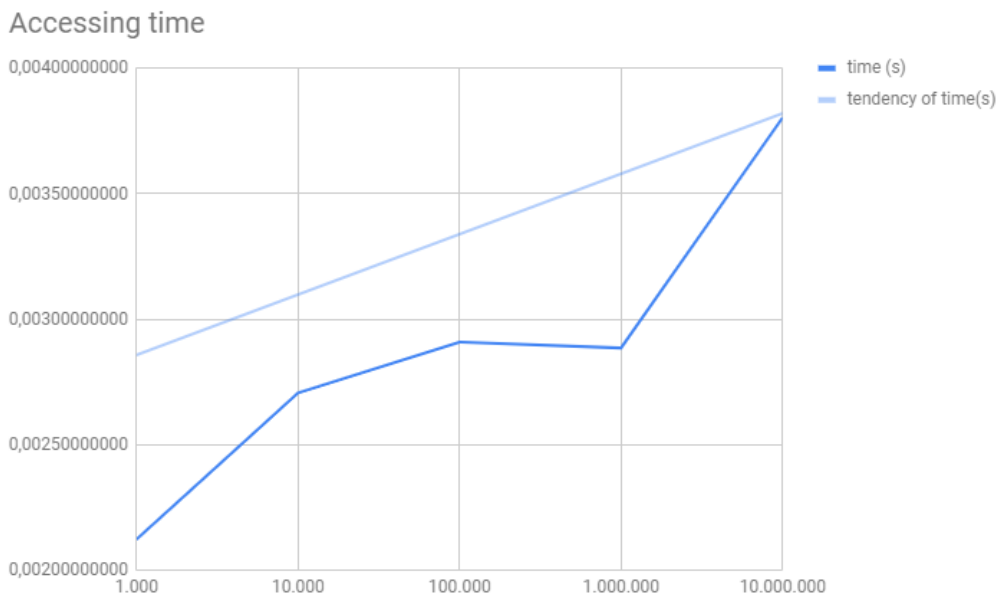
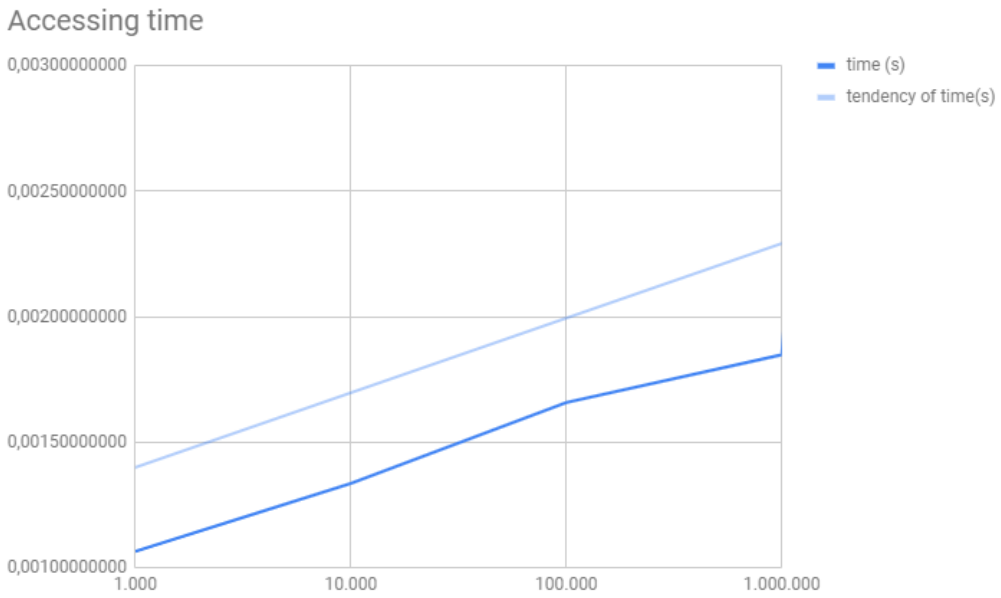


Figure 2

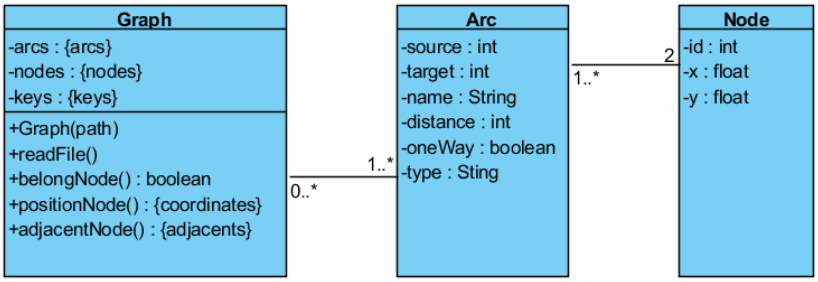


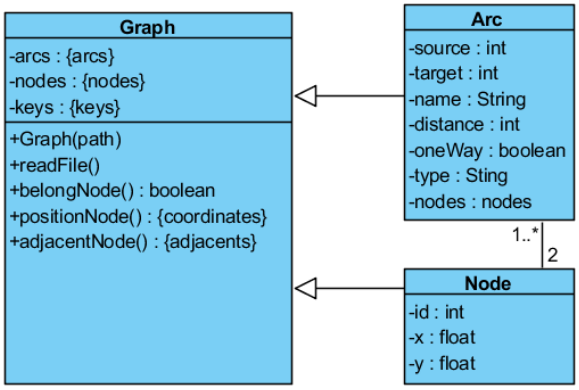
As we can see, comparing both figures being the first one for list data structures and the second one for dictionaries, last mentioned ones have a lower accessing time to data that will be key for us since when our program starts to be more complex and doing searches, the time could grow exponentially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





2. TASK 2 DOCUMENTATION

2.1 APPLICATION REQUIREMENTS

The second task is about the definition and implementation of several elements needed to solve main problem. These elements are *Frontier*, *State*, *Space State*, *Problem* and *TreeNode*. Now, the 'graph' class is named "stateSpace' and 'adjacentNodes' method has been replaced by 'successors' method.

2.2 STRUCTURES OF CREATED ARTIFACTS

To the ones imported in the task1 we are adding mostly calls to other classes so that we can use their content. Also mention an array bisection algorithm **'bisect'** that provides support for maintaining a list in sorted order without having to sort the list after each insertion in the class 'frontier'. The library **'json'** is imported in class 'problem' to read json files which represent the statement of the problem. In class *state*, **'haslib'** is imported to generate the MD5 representation of each of the states.

STATE SPACE

To achieve the objective, we have created a class 'stateSpace' that receives as a parameter the file's path containing the map information. It contains the needed variables and methods to read XML files and create the State Space, check if a node belongs to a graph and its position and generation of a specific state's successors list.

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 the type 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.

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 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.

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.

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.

4. **successors() :** Receives the 'id' of a node as a parameter, checks if it belongs to the created graph. The output is a list of state successors of the given node.

```
def successors(self, id):
    #input: problem state, output: list of adjacent nodes + extra info
    try:
        successors = []
        if self.belongNode(id):
            adjacents = [key for key in self._edges.keys() if id._current in key[0]]
            for data in adjacents:
                acc = "I'm in %s and I go to %s" % (data[0].zfill(10), data[1].zfill(10))
                aux = state(data[1], id.visited(data[1], id._nodes)) #creates new ._md5
                cost = self._edges[data][1]
                successors.append((acc, aux, cost))
            return successors
        else:
            raise ValueError
    except ValueError:
        print("Error. The node does not exist.")
```

PROBLEM

In order to represent the statement of the task, we've created the class 'problem'. This class is composed of the next elements:

- **file**: *.json* file.
- **init_state**: the initial state of the problem
- **state_space**: the generation of the graph from a xml file in the class 'stateSpace'.

Methods:

1. **readJson():** this method load the json_data received as a parameter in the constructor.
2. **isGoal():** receives as input an specific state and checks if it meets the requirements to be a goal state otherwise it returns false.

STATE

The description of the current state in the state space is represented by the class ‘state’. This class is composed of the next elements:

- **current**: the current position represented as a node.
- **nodes**: ordered list of nodes to visit.
- **md5**: MD5 representation of the state.

Methods:

1. **createCode():** its input parameters are the current node and a list of nodes (that is a state) and generates a representation of them in form of md5.
2. **visited():** receives a list of nodes and a specific node’s id. The method checks if the node belongs to the list. If it does, it is removed.

FRONTIER

The representation of the element frontier necessary to solve the search problem is collected in the class **‘frontier’**. Its attributes are:

- **frontier**: ordered list containing tree nodes in ascending order depending on the ‘f’ attribute of each node that it’s a randomly generated attribute.

Methods:

1. **createFrontier()**: creates an empty one.
2. **insert()**: adds a node to the frontier.
3. **remove()**: removes a node from the frontier
4. **isEmpty()** : true if the frontier has items, false otherwise.

TREENODE

This class represent the node inside a tree, element needed to solve the tree search problem. Its attributes are:

- **parent**: current state parent node.
- **state**: current state node.
- **cost**: from the initial node to the current one
- **action**: from the parent to reach the current state.
- **d**: depth of the node.
- **f**: random value that determines the insertion order in the frontier.

MAIN

We have created main class so as to work with this previous mentioned classes and generate tests which helps us to complete the task. Contains a method called **'stressTest'** that receives a frontier and a problem and test the maximum memory addressing capability.

```
def stressTest(f,p):
    elements = 0
    avg = 0
    maxim = 0
    minim = 9999
    try:
        while True:
            try:
                newState = state(str(int(p._init_state._current)+elements),
p._init_state._nodes)
                if(elements == 0):
                    node = treeNode(p._init_state, 0, "", elements)
                else:
                    node = treeNode(newState, 0, "", elements)
                start = time.time()
                f.insert(node)
                end = time.time()
                elements+=1
                timer = end - start
                avg += timer
                if(timer > maxim):
                    maxim = timer
                if(timer < minim):
                    minim = timer
            except MemoryError:
                print("full memory")
                print('nº elements: '+elements)
                print('min time: %.11f' % minim)
                print('max time: %.11f' %  maxim)
                print('avg time: %.11f' % (avg/elements))
                break
    except KeyboardInterrupt:
            print('avg time: %.11f' % (avg/elements))
            print('min time: %.11f' % minim)
            print('max time: %.11f' %  maxim)
            return elements
    return elements
```

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

```
json file: example
example.json
data/anchuras.graphml.xml
[(127.1360111900787, <treeNode.treeNode object at 0x7f33cc1c8fd0>),
(591.7378136374725, <treeNode.treeNode object at 0x7f33cc1d9e80>),
(612.2393770939552, <treeNode.treeNode object at 0x7f33cc1d9e10>)]
[(591.7378136374725, <treeNode.treeNode object at 0x7f33cc1d9e80>),
(612.2393770939552, <treeNode.treeNode object at 0x7f33cc1d9e10>)]
True
4331489738 state md5: 331b41c249f9d392954ce4a36df65ca7
4331489740 state md5: 229da16d1c640e1e81747a47bdd716fa
4331489763 state md5: 6d7014c2db65e030d3ae0af84286edfe
[("I'm in 4331489739 and I go to 4331489738", "(4331489738, ['4331489528',
'4331489668', '4331489711', '4762868815', '4928063625'])", '48.137'), ("I'm in
4331489739 and I go to 4331489740", "(4331489740, ['4331489528', '4331489668',
'4331489711', '4762868815', '4928063625'])", '108.841'), ("I'm in 4331489739 and
I go to 4331489763", "(4331489763, ['4331489528', '4331489668', '4331489711',
'4762868815', '4928063625'])", '63.11')]
```

2.3 JUSTIFICATION OF USED ARTIFACTS – FRONTIER DATA STRUCTURE

DICTIONARY

Our **first idea** was to use dictionaries, as we did before, to be the data structure to represent the frontier. Once we had implemented the stress test, we had to deal with the data order, the frontier wasn't ordered and we wanted to sort it in an ascendant order in some way, so we used the function **"dict"** which takes the dictionary and order it by a key; that decision causes us to have a very bad timing on inserting *TreeNode* elements in the frontier as the bigger the number of nodes in the frontier, the bigger the time it takes to reorder them in an ascendant way.

That's because every time an element was inserted in the frontier the sorted function was called. So first thousand elements weren't a problem but when it reached ten thousand elements the insertion time was affected because of the way of sorting.

So, one night of stress test results only into 256 thousand elements inserted. That way of sorting was the worst choice as it had an exponential growth.

LIST

Once we tried using dictionaries and realized we went in the wrong way, we tried using lists plus the **bisect module** to be able to make an ordered insert and not calling a sort method after an element is inserted or at the end when the list is completed. As mentioned in the documentation for the module: "*provides support for maintaining a list in sorted order without having to sort the list after each insertion*" so exactly what we were looking for.

We get the index where the element should be placed in real time with bisect and then call *insort(frontier, node)* to insert the required values in the previously give index that corresponds to the position so that the order remains correct.

```
bisect.insort(self._frontier, (node._f, node))
```

The complexity of this algorithm is O(n log n), little bit worse than linear but enough for what we need it for or at least that's what we think up to date, if not we will improve it.

PRIORITY QUEUE

We tried the priority queues by using 2 different modules:

- **heapq**: the speed of the insertion of nodes in the frontier drastically increased. In a minute a few millions of nodes could have been introduced in the frontier.

  ```
  heapq.heappush(self._frontier, (node._f, node))
  ```

  The reason for that was that this structure does not order the whole bunch of elements instead of that it just get the element with the lower "*f*" and then puts it on the front of the queue. Even when the first element (the lower one) was removed, the speed of retaking the next lower element was quite good.

- **queue**: as we wanted the frontier to be all ordered we considered this one.

  ```
  frontier = queue.PriorityQueue()
  ```

  By using that module, the frontier is defined as a Priority Queue object.
  This time, the whole frontier would be ordered by "f" **but not the objects inside**.

  ```
  frontier.put((node._f, node))
  ```

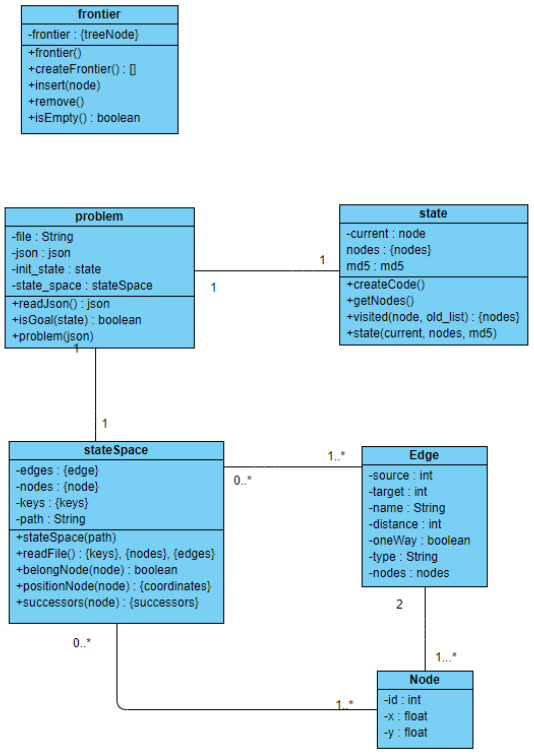
In the following table, we represent some of the values we got depending of the previous modules mentioned before that we have tried. These values were obtained be executing the stress test a total time of 1 minute each.

module used	avg time (s)	min time (s)	max time (s)	# elements
bisect	0,00006409883	0,00000143051	0,00312209129	784.241
heapq	0,00000186862	0,00000119209	0,01135897636	4.570.800
queue	0,00000377094	0,00000286102	0,02315831184	5.454.733

As we can see the most optimal ones may be **heapq** or **queue** but they only sort the first value of the frontier which is the key of priority queues data structure, if we only need to have it ordered cause of the elimination of nodes (the one with the lower f should be removed) and do not take into account how the rest of nodes are ordered, then the way to go must be one of those modules instead of the bisection one we are using at the moment.

2.4 UML SKETCHES

This does not clearly represent the implementation of our code but was used to represent the general idea and thinking about our goal, so we found it interesting to also show it over here.



3. TASK 3 DOCUMENTATION

3.1 APPLICATION REQUIREMENTS

The third assignment main goal is based on the construction of a basic version of the search algorithm needed to solve the global task. This search algorithm considers three strategies which are the followings:

- Breath-first search
- Depth-first search, Depth-limited search, Iterative Deepening search
- Uniform cost search

An important aspect appears in this task that is the pruning, so the program allows the user to do the search with or without pruning.

3.2 STRUCTURES OF CREATED ARTIFACTS

To the ones imported in task1 and task2 an additional library called **'sortedcontainers'** has been called in order to use the data structure *SortedKeyList*, an element that helps us to control the priority inside the frontier.

Also, we need to create a new list of elements called 'visitedList' where we store the nodes that have been removed from the frontier, the ones that has been visited. For that job we choose a simple list as the data structure since no order is needed.

Talking about each class, we are listing below the ones in which we have made changes in this new task, if one is not mentioned then it remains at I was in task2.

PROBLEM

New attribute added:

- **visitedList**: normal list to store the nodes we have visited and be able to prune.

A new method have also been added to create the nodes of the tree:

```
def createTreeNodes(self, ls, node, depthl, strategy):
      tree = []
        if(depthl >= node._d):
            for (action, result, cost) in ls:
                s = treeNode(result, strategy, node, float(cost), action)
                tree.append(s)
        return tree
```

FRONTIER

The representation of the frontier is no longer an ordered dictionary, so we have changed the following method to make use of the new data structure we are using now:

1. **createFrontier()**: creates an empty one using *SortedKeyList* from 'sortedcontainers' in order to achieve and ordered insert where duplicates are allowed.

```
def _createFrontier(self):
            frontier = SortedKeyList(key=treeNode.getF)
            return frontier
```

To the constructor of the data structure we pass a desired *key* that will be the insertion value considered when making an ordered insert, in this case the attribute 'f' of the class treeNode since we are inserting objects of that class.

TREENODE

To this class we have just added a 'getter' although in python they are not necessary, we were needing it to pass it as the key in *SortedKeyList* data structure since that key can not be an attribute, it must be a function so therefore out 'getter' is the function that's give us the value we want the elements in the frontier to be ordered by.

```
def getF(self):
      return self._f
```

MAIN

This is the class were most changes have been done since is the one containing the search algorithms which is the main goal of this task.

This class contains the following methods:

1. **askInfo() :** the method is created to manage the initial information introduced by the user needed to solve the problem.

```
def askInfo():
    try:
        filename = input('json file: ')
        if filename.isdigit():
            raise ValueError
        print(filename + ".json") #print json file name
        switch = {
        0: 'breath-first search', 1: 'depth-first search', 2: 'depth-limited search',\
        3: 'iterative deepening search', 4: 'uniform cost search', 5: 'a* search'}
        print("\n".join("{}: {}".format(k, v) for k, v in switch.items()))
        strategy = int(input('strategy: '))
        if isinstance(strategy, str) or strategy > 5 or strategy < 0: raise ValueError
        yes = {'y','yes','yay'}; no = {'n','no','nay'}
        pruning = input('pruning(y/n): ').lower()
        if pruning in yes: pruning = True; print(switch[strategy] + ' w/ pruning')
        elif pruning in no: pruning = False; print(switch[strategy] + ' w/o pruning')
        else: raise ValueError
        return filename, strategy, pruning
    except ValueError:
        print("Error. Not a valid input.")
        sys.exit(1)
```

2. **limSearch():** method containing the limited search algorithm with and without pruning.

```
def limSearch(problem, strategy, depthl, pruning):
    f = frontier()
    initial = treeNode(problem._init_state, strategy)
    f.insert(initial)
    problem._visitedList[initial._state._md5] = initial._f
    sol = False
    while(not sol and not f.isEmpty()):
        act = f.remove()
        if(problem.isGoal(act._state)): sol = True
        else:
            ls = problem._state_space.successors(act._state)
            ln = problem.createTreeNodes(ls, act, depthl, strategy)
            if pruning:
                for node in ln:
                    if node._state._md5 not in problem._visitedList:
                        f.insert(node)
                        problem._visitedList[node._state._md5] = node._f
                    elif abs(node._f) < abs(problem._visitedList[node._state._md5]):
                        f.insert(node)
                        problem._visitedList[node._state._md5] = node._f
            else:
                for node in ln: f.insert(node)
    if(sol): return act
    else: return None
```

3. **search():** includes the algorithm used in the iterative deepening search strategy, it calls recursively the previous explained *limSearch* method.

```
def search(problem, strategy, depthl, depthi, pruning):
    depthact = depthi
    sol = None
    while(not sol and depthact <= depthl+1):
        print(depthact)
        sol = limSearch(problem, strategy, depthact, pruning)
        print(sol)
        depthact += depthi
    return sol
```

4. **createSolution():** once a solution is reached by executing one of the previous search algorithms, this methods process it by creating a new list and appending to it each method string action ("I'm in node1 and I go to node2") until we reach the problem root or saying it in a programming way until parent is None. Then we reverse the list to have it ordered in a more comprehensive way for the user.

```
def createSolution(sol, itime, etime):
    if(sol is not None):
        list = []
        act = sol
        list.append(act._action)
        while(act._parent is not None and act._parent._action is not None):
            list.append(act._parent._action)
            act = act._parent
        list.reverse()
        print('cost: %f, depth: %d, elapsed time: %fs\ncheck out.txt for more
info' % (sol._cost, sol._d, etime-itime))
        pprint(list)
        writeSolution(sol, itime, etime, list)
    else:
        print('no solution found for the given depth limit')
```

5. **writeSolution():** it is called in the previous method and is where we manage the output file (.txt) that is generated including the solution to the problem.

```
def writeSolution(sol, itime, etime, list):
    txt = open('out.txt','w')
    if(sol is not None):
        line1 = 'cost: %f, depth: %d, elapsed time: %fs\n' % (sol._cost,
sol._d, etime-itime)
        line2 = 'goal node: %s\n' % str(sol)
        line3 = time.strftime('time and date: %H:%M:%S-%d/%m/%Y\n\n')
        line4 = pformat(list)
        txt.writelines([line1, line2, line3, line4])
    else:
        txt.write('no solution found for the given depth limit')
    txt.close()
```

3.3 JUSTIFICATION OF USED ARTIFACTS – FRONTIER DATA STRUCTURE

At first, when developing the code for the third task, we were using ordered dictionaries as data structure for the frontier, but we ran into a problem. As soon as we start executing the search algorithms, we found that if the solution wasn't in the branch the algorithm was searching it won't find any solution which the expected result was not.

Our problem was that because of using dictionaries, keys could not be duplicated so if we execute a uniform cost search, the algorithm would work since cost are most of them different one from another but when executing a breath or depth first searches, it wouldn't give a solution; the are lots of nodes at same depth values so instead of searching for all of them, just one was taken into account and the other ones were overwritten.

We needed a data structure that allowed us to make ordered insertion in real time but allowing duplicates, so we decided to use *SortedKeyList* as we have mentioned before. The stress test was done using it, realizing that the insertion time had drastically decreased comparing to the data structures used in task2, just a few seconds were enough to insert millions of elements in the frontier, every one sorted by its "f" value and allowing duplicates.

4. BIBLIOGRAPHY

All used sources in the realization of the project documentation for all the different tasks is listed below, our intelligent systems teacher notes have also been used.

- 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/
- https://docs.python.org/2/library/bisect.html
- https://docs.python.org/3/library/queue.html
- https://docs.python.org/2/library/heapq.html
- https://stackoverflow.com/questions/11215851/python-equivalent-of-javas-compareto
- http://www.grantjenks.com/docs/sortedcontainers/