Escuela Superior de Informática

Inteligent Systems

---

osm-gps

Members:
Juan Perea Campos
Samuel González Linde
Javier Abengózar Palop

Repository:
`https://github.com/jupcan/osm-gps`

December 8, 2018

Contents

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
```

1.3.1 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 a specific element in an easy way[6]. It collects the keys from the XML files and store them in the dictionaries returning three dictionaries with 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 = {}, {}, {}
    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, output: true/false if in nodes
    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, output: latitude&longitude(y,x) of current node
    try:
        if self.belongNode(id):
            return self._nodes[id]
        else:
            raise ValueError
    except ValueError:
        print("error. the node does not exist"); sys.exit(1)
```

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.")
```

1.3.2 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:

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

1.4 Justification of Used Artifacts - Accesing times

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

Working with the first, mid and final elements 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.

1.4.1 List Times

n	**1.000**					
element	**1st acc**	**2nd acc**	**3rd acc**	**4th acc**	**5th acc**	**avrg/ele**
first	1,43051E-06	9,5367E-07	7,1526E-07	4,7684E-07	9,5367E-07	9,0599E-07
middle	8,82149E-06	7,15256E-06	6,19888E-06	6,4373E-06	6,91414E-06	7,10487E-06
last	4,7684E-07	4,7684E-07	2,3842E-07	4,7684E-07	7,1526E-07	4,7684E-07
avrg/acc:	2,68221E-06	2,14577E-06	1,78814E-06	1,84775E-06	2,14577E-06	2,12193E-06
n	**10.000**					
first	9,5367E-07	9,5367E-07	1,43051E-06	1,19209E-06	9,5367E-07	1,09672E-06
middle	7,86781E-06	9,77516E-06	1,072884E-05	9,29832E-06	8,58307E-06	9,25064E-06
last	4,7684E-07	4,7684E-07	4,7684E-07	4,7684E-07	4,7684E-07	4,7684E-07
avrg/acc:	2,32458E-06	2,80142E-06	3,15905E-06	2,74181E-06	2,5034E-06	2,70605E-06
n	**100.000**					
first	1,66893E-06	1,19209E-06	9,5367E-07	1,19209E-06	9,5367E-07	1,19209E-06
middle	1,072884E-05	9,53674E-06	9,77516E-06	9,77516E-06	1,0252E-05	1,001358E-05
last	4,7684E-07	4,7684E-07	4,7684E-07	2,3842E-07	4,7684E-07	4,2916E-07
avrg/acc:	3,21865E-06	2,80142E-06	2,80142E-06	2,80142E-06	2,92063E-06	2,90871E-06
n	**1.000.000**					
first	9,5367E-07	1,19209E-06	9,5367E-07	1,43051E-06	1,43051E-06	1,19209E-06
middle	1,001358E-05	9,53674E-06	1,144409E-05	9,29832E-06	9,29832E-06	9,91821E-06
last	4,7683E-07	4,7684E-07	4,7684E-07	2,3842E-07	4,7684E-07	4,2915E-07
avrg/acc:	2,86102E-06	2,80142E-06	3,21865E-06	2,74181E-06	2,80142E-06	2,88486E-06
n	**10.000.000**					
first	1,90735E-06	1,66893E-06	1,43051E-06	1,43051E-06	1,43051E-06	1,57356E-06
middle	1,692772E-05	1,621246E-05	9,77516E-06	1,096725E-05	1,168251E-05	1,311302E-05
last	4,7684E-07	7,1526E-07	4,7684E-07	4,7684E-07	4,7684E-07	5,2452E-07
avrg/acc:	4,82798E-06	4,64916E-06	2,92063E-06	3,21865E-06	3,39747E-06	3,80278E-06

Table 1: List Single Element Accesing Times

n	1.000	10.000	100.000	1.000.000	10.000.000
time(s)	0,002121926	0,0027060505	0,0029087065	0,0028848635	0,0038027765

Table 2: List All Elements Average Accesing Times

1.4.2 Dictionary Times

n	**1.000**					
element	**1st acc**	**2nd acc**	**3rd acc**	**4th acc**	**5th acc**	**avrg/ele**
first	7,1526E-07	9,5367E-07	7,1526E-07	1,19209E-06	7,1526E-07	8,5831E-07
middle	2,38419E-06	3,09944E-06	1,49042E-06	3,57628E-06	2,86102E-06	2,68227E-06
last	7,1526E-07	4,7684E-07	9,5367E-07	7,1526E-07	7,1526E-07	7,1526E-07
avrg/acc:	9,5368E-07	1,13249E-06	7,8984E-07	1,37091E-06	1,07289E-06	1,06396E-06
n	**10.000**					
first	9,5367E-07	9,5367E-07	1,19209E-06	9,5367E-07	9,5367E-07	1,00135E-06
middle	3,33786E-06	2,86102E-06	3,57628E-06	4,52995E-06	3,8147E-06	3,62396E-06
last	7,1526E-07	4,7684E-07	7,1526E-07	9,5367E-07	7,1526E-07	7,1526E-07
avrg/acc:	1,2517E-06	1,07288E-06	1,37091E-06	1,60932E-06	1,37091E-06	1,33514E-06
n	**100.000**					
first	1,19209E-06	1,19209E-06	1,66893E-06	1,19209E-06	1,66893E-06	1,38283E-06
middle	3,8147E-06	4,52995E-06	4,52995E-06	3,8147E-06	4,29153E-06	4,19617E-06
last	1,19209E-06	7,1526E-07	9,5367E-07	1,43051E-06	9,5367E-07	1,04904E-06
avrg/acc:	1,54972E-06	1,60933E-06	1,78814E-06	1,60933E-06	1,72853E-06	1,65701E-06
n	**1.000.000**					
first	1,43051E-06	1,19209E-06	1,66893E-06	1,43051E-06	1,66893E-06	1,47819E-06
middle	5,00679E-06	4,52995E-06	4,76837E-06	5,48363E-06	4,52995E-06	4,86374E-06
last	9,5367E-07	9,5367E-07	1,19209E-06	9,5367E-07	1,19209E-06	1,04904E-06
avrg/acc:	1,84774E-06	1,66893E-06	1,90735E-06	1,96695E-06	1,84774E-06	1,84774E-06
n	**10.000.000**					
first	6,480217E-05	1,66893E-06	1,43051E-06	1,049042E-05	1,90735E-06	1,605988E-05
middle	1,335144E-05	5,24521E-06	4,76837E-06	1,597404E-05	4,29153E-06	8,72612E-06
last	9,584427E-05	7,1526E-07	1,19209E-06	2,503395E-05	7,1526E-07	2,470017E-05
avrg/acc:	4,349947E-05	1,90735E-06	1,84774E-06	1,28746E-05	1,72854E-06	1,237154E-05

Table 3: Dictionary Single Element Accesing Times

n	1.000	10.000	100.000	1.000.000	10.000.000
time(s)	0,001063959	0,0013351435	0,001657008	0,0018477425	0,01237154

Table 4: Dictionary All Elements Average Accesing Times

1.4.3 Graphical Results

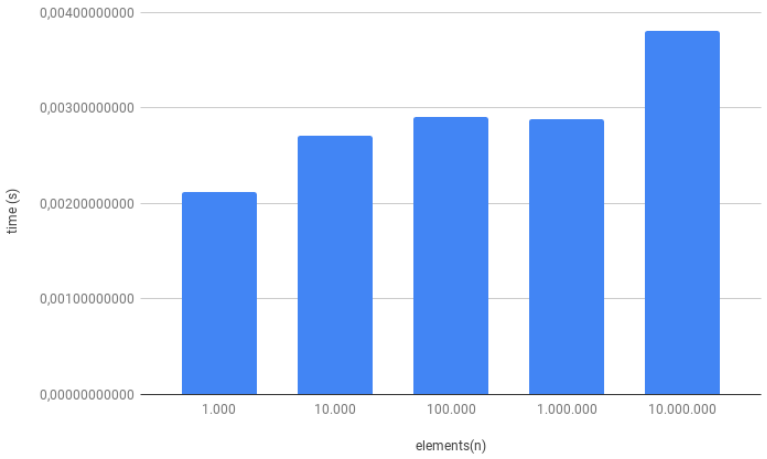

Figure 1: Time vs Elements using Lists

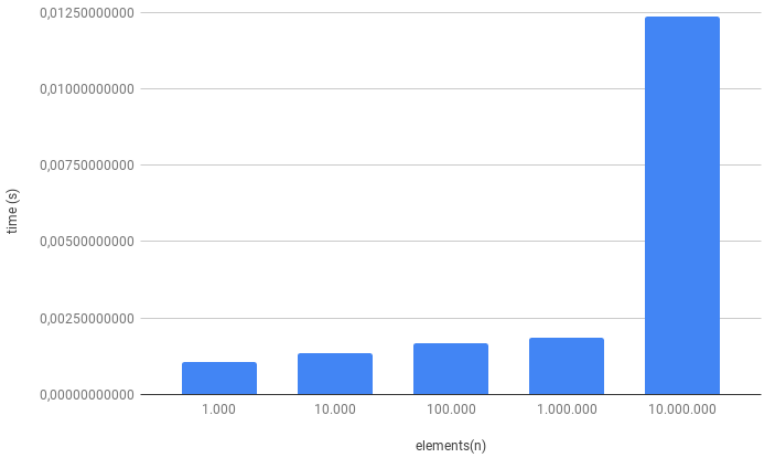

Figure 2: Time vs Elements using Dictionaries

As we can see, comparing both figures being Figure1 for list data structures and Figure2 for dictionaries, last mentioned ones have a lower accessing times (Table3) 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 lists**. At first, we used lists 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 directly access the target number in the set, rather than iterate through every item in the list and compare them.

1.5 UML Sketches

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

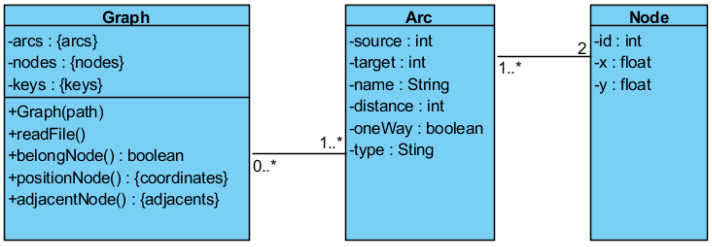

Figure 3: Task1 UML Representation

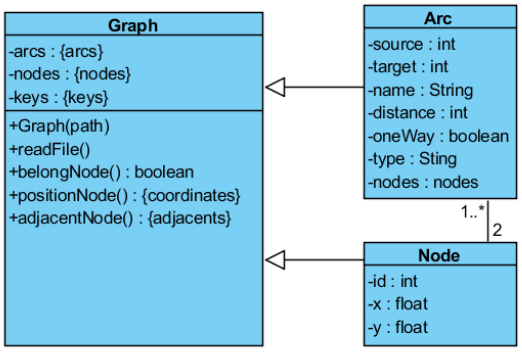

Figure 4: Inheritance Representation

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 used to generate the MD5 representation of each of the states.

2.2.1 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, 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. 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 a specific element in an easy way[6]. It collects the keys from the XML files and store them in the dictionaries returning three dictionaries with keys, nodes and edges respectively.
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.")
```

2.2.2 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**: 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.

2.2.3 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.

2.2.4 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.

2.2.5 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.

2.2.6 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('n 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:

```
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

2.3.1 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.

2.3.2 List

Once we tried using dictionaries and realized we went in the wrong way, we tried using lists plus the ***bisect module***[2] 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.

2.3.3 Priority Queue

We tried the priority queues by using 2 different modules:

- **heapq**[3]: 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.

  ```
  bisect.insort(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**[4]: 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 Table5, 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

Table 5: Modules Times Comparison

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 as first draft think about our goals, found it interesting to also show it over here.

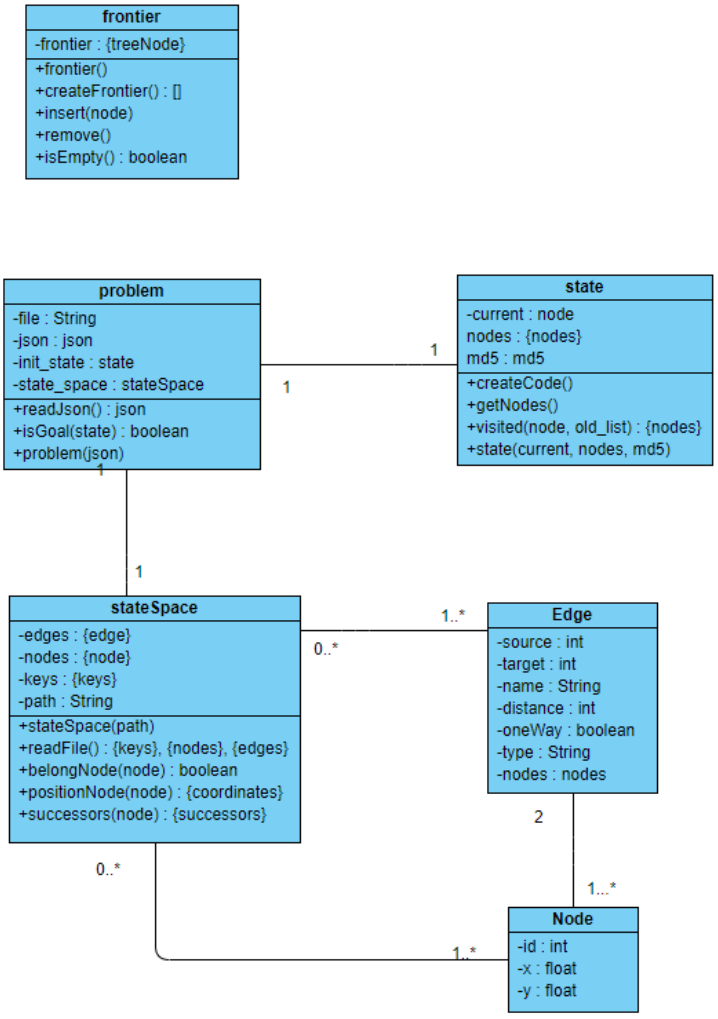

Figure 5: Task2 UML Representation

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 **sorted-containers**[1] 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 as it was in task2.

3.2.1 Problem

New attribute added:

- **visitedList**: 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:

1. **createTreeNodes()**: creates the tree representation to be able to search.

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

3.2.2 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:

1. **createFrontier()**: creates an empty one using *SortedKeyList* from **sorted-containers** 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.

3.2.3 TreeNode

To this class we have added a *getter* although in python they aren't necessary, we were needing[1] 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 our *getter* is the one that gives us the value we want the elements in the frontier to be ordered by.

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

[1]Something similar to Java's compareTo() method[7] but in Python.

3.2.4 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.

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: 'greedy search', 6: 'a*
            search'}
        print("\n".join("{}: {}".format(k, v) for k, v in switch.items()))

        strategy = int(input('strategy: '))
        if isinstance(strategy, str) or strategy > 6 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; stat = switch[strategy] + ' w/ pruning'
        print(stat)
        elif pruning in no: pruning = False; stat = switch[strategy] + ' w/o pruning'
        print(stat)
        else: raise ValueError

        depthl = int(input('depth: '))-1
        if isinstance(depthl, str): raise ValueError
        return filename, strategy, depthl, pruning, stat
    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()
    num_f = 0
    initial = treeNode(problem._init_state, strategy)
    f.insert(initial); num_f += 1
    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); num_f += 1
                        problem._visitedList[node._state._md5] = node._f
                    elif abs(node._f) < abs(problem._visitedList[node._state._md5]):
                        f.insert(node); num_f += 1
                        problem._visitedList[node._state._md5] = node._f
            else:
                for node in ln: f.insert(node)
    if(sol): return act, num_f
    else: return None
```

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

```
def search(problem, strategy, depthl, depthi, pruning):
    depthact = depthi
    sol = None
    while(not sol and depthact <= depthl+1):
        sol = limSearch(problem, strategy, depthact, pruning)
        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 and write it in **out.txt** file.

```
def createSolution(sol, itime, etime, num_f):
    txt = open('solu/out.txt','w')
    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: %.3f, depth: %d, spatialcxty: %d, temporalcxty:
        %fs\ncheck out.txt for more info' % (sol._cost, sol._d, num_f, etime-itime))

        line1 = 'cost: %.3f, depth: %d, spatialcxty: %d, temporalcxty: %fs\n'
        % (sol._cost, sol._d, num_f, etime-itime)
        line2 = 'goal node: %s - %s\n' % (sol, sol._state._current)
        line3 = time.strftime('time and date: %H:%M:%S-%d/%m/%Y\n\n')
        line4 = pformat(list)
        txt.writelines([line1, line2, line3, line4])
    else:
        print('no solution found for the given depth limit')
        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.

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.

3.4 Text Solution Example

For a given example json file like the following:

```
{
   "graphlmfile": "Ciudad Real/data/anchuras.graphml",
   "IntSt": {
      "node": "4331489739",
      "listNodes":["4331489528","4331489668","4331489711","4762868815","4928063625"],
      "id": "f4b616551965fb586e608397c308bf0f"
   }
}
```

And using the following command line inputs:

```
0: breath-first search
1: depth-first search
2: depth-limited search
3: iterative deepening search
4: uniform cost search
strategy: 0
pruning(y/n): y
breath-first search w/ pruning
depth: 999
data/anchuras.graphml.xml
```

The generated **out.txt** containing the solution will be:

```
cost: 10891.904, depth: 36, spatialcxty: 2783, temporalcxty: 0.210517s
goal node: <treeNode.treeNode object at 0x7f2898b68d68> − 4928063625
time and date: 19:29:22−23/11/2018

["I'm in 4331489739 and I go to 4331489738 c/Barrio Nuevo",
"I'm in 4331489738 and I go to 4331489532 c/Barrio Nuevo",
"I'm in 4331489532 and I go to 0946409139 c/sinNombre",
"I'm in 0946409139 and I go to 4331489528 c/sinNombre",
"I'm in 4331489528 and I go to 4331489718 c/Calle Arroyo",
"I'm in 4331489718 and I go to 4331489716 c/Calle Arroyo",
"I'm in 4331489716 and I go to 4331489709 c/Plaza Espana",
"I'm in 4331489709 and I go to 4331489708 c/Calle de la Iglesia",
"I'm in 4331489708 and I go to 4331489711 c/Calle Cervantes",
"I'm in 4331489711 and I go to 4331489710 c/Calle Cuevas",
"I'm in 4331489710 and I go to 4762868812 c/Calle Periodista Antonio Herrero",
"I'm in 4762868812 and I go to 4331489692 c/Calle Periodista Antonio Herrero",
"I'm in 4331489692 and I go to 4331489662 c/Calle Periodista Antonio Herrero",
"I'm in 4331489662 and I go to 4331489668 c/Calle Periodista Antonio Herrero",
"I'm in 4331489668 and I go to 4331489662 c/Calle Periodista Antonio Herrero",
"I'm in 4331489662 and I go to 4331489695 c/Calle Gibraltar Espanol",
"I'm in 4331489695 and I go to 4331489665 c/Calle Gibraltar Espanol",
"I'm in 4331489665 and I go to 4331489667 c/Calle Gibraltar Espanol",
"I'm in 4331489667 and I go to 4331489656 c/Calle Gibraltar Espanol",
"I'm in 4331489656 and I go to 4331489564 c/Calle Gibraltar Espanol",
"I'm in 4331489564 and I go to 4331489573 c/Calle Carretera",
"I'm in 4331489573 and I go to 4331489570 c/Calle Carretera",
"I'm in 4331489570 and I go to 4331489627 c/Calle Carretera",
"I'm in 4331489627 and I go to 4331489577 c/Calle Carretera",
"I'm in 4331489577 and I go to 4762868815 c/Calle Carretera",
"I'm in 4762868815 and I go to 4331489577 c/Calle Carretera",
"I'm in 4331489577 and I go to 4331489627 c/Calle Carretera",
"I'm in 4331489627 and I go to 0946409159 c/sinNombre",
"I'm in 0946409159 and I go to 0982621562 c/sinNombre",
"I'm in 0982621562 and I go to 4331489516 c/sinNombre",
"I'm in 4331489516 and I go to 0982621804 c/sinNombre",
"I'm in 0982621804 and I go to 4928063639 c/Calle Manzanillo",
"I'm in 4928063639 and I go to 4928063630 c/Calle Horcajo",
"I'm in 4928063630 and I go to 4928063624 c/Calle Horcajo",
"I'm in 4928063624 and I go to 4928063625 c/Calle Horcajo"]
```

4 Task 4 Documentation

4.1 Application Requirements

This fourth and last assignment goal is to add two new strategies and generate a file with the solution, its cost, the spatial complexity and the temporal one.

- A* using as heuristic for a concrete state: **h(state)=minimum distance** between the OSM node of the current state and any node in the list of nodes to be traveled.
- Greedy search

The distance must be represented in meters and is computed by means of a function provided by our teacher, we are showing it in the structures of created artifacts section. As an added feature it is possible to generate a sequence of images representing the solution, we have decided to implement this functionality by means of a *.gpx* file generation.

4.2 Structures of Created Artifacts

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 as it was in task3.

4.2.1 Problem

Added the method we were talking about before to compute the straight-line distance between two openstreetmap nodes that will give us the heuristic, it is called in *createTreeNodes* method (page 35) so as to pass the value to the *treeNode* class.

```
def distance(self, node1, node2):
    (lng1, lat1) = self.positionNode(node1)
    (lng2, lat2) = self.positionNode(node2)
    earth_radius = 6371009

    phi1 = math.radians(float(lat1)); phi2 = math.radians(float(lat2))
    theta1 = math.radians(float(lng1)); theta2 = math.radians(float(lng2))
    d_phi = phi2 - phi1; d_theta = theta2 - theta1
    h = math.sin(d_phi/2)**2 +math.cos(phi1)*math.cos(phi2)*\
    math.sin(d_theta/2)**2
    h = min(1.0, h) #protect against floating point errors
    arc = 2*math.asin(math.sqrt(h))

    #return distance in units of earth_radius
    dist = arc*earth_radius
    return dist
```

4.2.2 TreeNode

Included the two new search strategies and added the heuristics to the constructor to be able to use it when needed, in the method (page 35) *createTreeNodes* from problem we create objects of the class passing all the arguments plus the new **heu**.

```
class treeNode():
    def __init__(self, state, strategy, parent=None, cost=0, action=None, h=0, d
        =1):
        self._state = state #current state of the problem
        self._strategy = strategy
        self._parent = parent
        if parent is not None: self._cost=parent._cost+cost; self._d=parent._d+1
        else: self._cost = cost; self._d = d
        self._action = action
        switch = {0: d, 1: -d, 2: -d, 3: -d, 4: self._cost, 5: h, 6:h+self._cost}
        self._f = switch[strategy] #factory pattern

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

4.2.3 Main

Added the new searches to be able to execute them and to generate a graphical representation of the solution, we have included a new method *createGpx* that, as the *createSolution(4)* one does, goes through it and generates two different lists, one with all the steps required as before and a new one with the points we have to go through specified in the json file (list of nodes of problems' initial state).

As stated in the gpx osm wiki[8], the file can contain routes, **tracks** and **waypoints**, we are using the last two representing the solution as a *track⟨trkseg⟩* will all the *steps⟨trkpt⟩* that will be shown over the map and, as *waypoints⟨wpt⟩*, the nodes the track must go through adding to them as a mouse hover animation, the osm node id and its coordinates(x,y).

In other words, we are creating kind of a template with all the elements following the format needed and printing the statistics of the program in a box. Just mention that we are also generating a *svg* image of the solution by using a subprocess call to **gpx2svg**[5] open source script that converts a given gpx file into a svg image.

```
call(['solu/gpx2svg', '−i', 'solu/out.gpx', '−o', 'solu/out.svg'])
```

All solution files are stored in solu project folder.

1. **createGpx()**: creates graphical representation of the solution.

```
def createGpx(problem, sol, itime, etime, num_f, stat):
    if(sol is not None):
        list, points = [], []
        act = sol
        list.append(problem._state_space.positionNode(act._state._current))
        while(act._parent is not None and act._parent._action is not None):
            list.append(problem._state_space.positionNode(act._parent._state._current))
            act = act._parent
        list.reverse()
        points.append(problem._init_state._current)
        for i in problem._init_state._nodes: points.append(i)

    gpx = open('solu/out.gpx','wb')
    root = etree.Element('gpx', version='1.0'); root.text = '\n'
    for n in points:
        wpt = etree.SubElement(root, 'wpt', lat=problem._state_space.positionNode(n)[0],
            lon=problem._state_space.positionNode(n)[1]); wpt.tail = '\n'
        w_name = etree.SubElement(wpt, 'name'); w_name.text = n
        w_desc = etree.SubElement(wpt, 'desc'); w_desc.text = problem._state_space.
            positionNode(n)[0] + ', ' + problem._state_space.positionNode(n)[1]
    trk = etree.SubElement(root, 'trk')
    t_name = etree.SubElement(trk, 'name'); t_name.text = 'out.gpx'; t_name.tail = '\n'
    desc = etree.SubElement(trk, 'desc'); desc.text = '%s, cost: %.3f, depth: %d, scxty: %d
        , tcxty: %fs' % (stat, sol._cost, sol._d, num_f, etime-itime); desc.tail = '\n'
    link = etree.SubElement(trk, 'link', href='http://www.uclm.es')
    text = link = etree.SubElement(link, 'text'); text.text = 'uclm project'
    trkseg = etree.SubElement(trk, 'trkseg'); trkseg.text = '\n'
    for n in list:
        trkpt = etree.SubElement(trkseg, 'trkpt', lat=n[0].zfill(10), lon=n[1].zfill(10)); trkpt.
            tail = '\n'
        date = etree.SubElement(trkpt, 'time'); date.text = time.strftime('%Y-%m%dT%
            H:%M:%S')
    doc = etree.ElementTree(root)
    doc.write(gpx, xml_declaration=True, encoding='utf-8')
    gpx.close()
```

4.3 Graphical Solution Example

For the same example (3.4) used in task3 documentation, its associated *.gpx* file and therefore its graphical representation as svg and shown in openstreetmap are:

```
<?xml version='1.0' encoding='UTF−8'?>
<gpx version="1.0">
<wpt lat="39.4811805" lon="−4.8343247"><name>4331489739</name><desc>39.4811805, −
4.8343247</desc></wpt>
<wpt lat="39.4806398" lon="−4.8353269"><name>4331489528</name><desc>39.4806398, −
4.8353269</desc></wpt>
<wpt lat="39.4824022" lon="−4.8398297"><name>4331489668</name><desc>39.4824022, −
4.8398297</desc></wpt>
<wpt lat="39.4799802" lon="−4.8366674"><name>4331489711</name><desc>39.4799802, −
4.8366674</desc></wpt>
<wpt lat="39.4799865" lon="−4.8426079"><name>4762868815</name><desc>39.4799865, −
4.8426079</desc></wpt>
<wpt lat="39.4413282" lon="−4.8032811"><name>4928063625</name><desc>39.4413282, −
4.8032811</desc></wpt>
<trk><name>out.gpx</name>
<desc>breath−first search w/ pruning, cost: 10891.904, depth: 36, scxty: 2783, tcxty:
0.210517s</desc>
<link href="http://www.uclm.es"><text>uclm project</text></link><trkseg>
<trkpt lat="39.4808935" lon="−4.8347446"><time>2018−11−23T19:29:22</time></trkpt>
<trkpt lat="39.4810593" lon="−4.8350192"><time>2018−11−23T19:29:22</time></trkpt>
<trkpt lat="039.480927" lon="−004.83529"><time>2018−11−23T19:29:22</time></trkpt>
<trkpt lat="39.4806398" lon="−4.8353269"><time>2018−11−23T19:29:22</time></trkpt>
<trkpt lat="039.480455" lon="−4.8356325"><time>2018−11−23T19:29:22</time></trkpt>
<trkpt lat="039.480176" lon="−4.8359648"><time>2018−11−23T19:29:22</time></trkpt>
<trkpt lat="39.4801017" lon="−4.8361395"><time>2018−11−23T19:29:22</time></trkpt>
<trkpt lat="39.4799142" lon="−4.8366188"><time>2018−11−23T19:29:22</time></trkpt>
<trkpt lat="39.4799802" lon="−4.8366674"><time>2018−11−23T19:29:22</time></trkpt>
<trkpt lat="39.4798372" lon="−4.8371945"><time>2018−11−23T19:29:22</time></trkpt>
<trkpt lat="39.4800834" lon="−4.8371603"><time>2018−11−23T19:29:22</time></trkpt>
.
.
.
<trkpt lat="39.4776182" lon="−4.8368194"><time>2018−11−23T19:29:22</time></trkpt>
<trkpt lat="39.4427755" lon="−04.810121"><time>2018−11−23T19:29:22</time></trkpt>
<trkpt lat="39.4410817" lon="−4.8028064"><time>2018−11−23T19:29:22</time></trkpt>
<trkpt lat="39.4413551" lon="−4.8029432"><time>2018−11−23T19:29:22</time></trkpt>
<trkpt lat="39.4413993" lon="−4.8029679"><time>2018−11−23T19:29:22</time></trkpt>
<trkpt lat="39.4413282" lon="−4.8032811"><time>2018−11−23T19:29:22</time></trkpt>
</trkseg></trk></gpx>
```

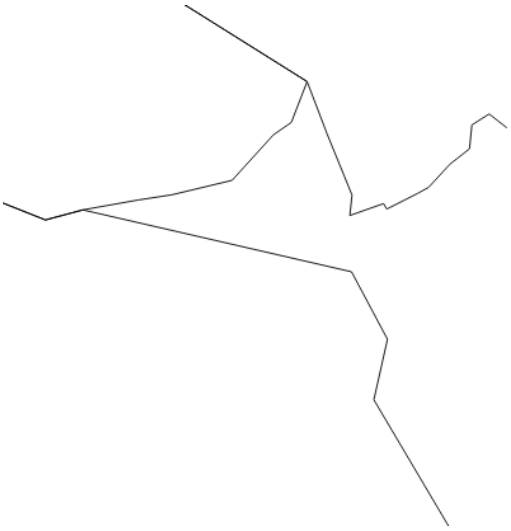

Figure 6: Zoomed in *solu/out.svg* image

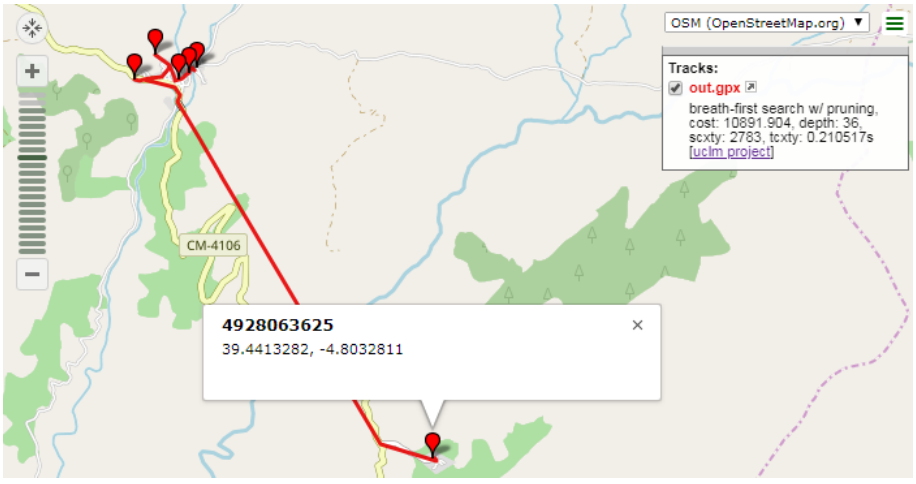

Figure 7: Solution shown in OpenStreetMap[2]

[2]More examples and complete *gpx* files available in the project repository.

5 Task 5 Documentation

5.1 Application Requirements

This last task consists in the revision of the code to find faults, make its corrections and thus complete the lab project of the subject.

5.2 Structures of Created Artifacts

As in the other tasks, we have below each class in which we have made some changes in order to accomplish this last task, if one is not mentioned then it remains as it was in task4.

5.2.1 Problem

We have implemented the use of a new heuristic (h0) better than the one we have had until now (h1). Given a state **(x, [a, b, c])** we can define them as:

- h1 = **min_distance**[3]((x, a),(x, b),(x, c))
- h0 = h1 + **min_distance**((a, b), (a, c))

By adding another minimum distance with current node the first one to be visited, we obtain a bigger heuristic closer to the real cost of the path thus a better one to consider, we can appreciate a spacial complexity smaller when running the program.

[3]Method *distance* explained in section 4.2.1

1. **createTreeNodes()**: included the new heuristic to the method.

```
def createTreeNodes(self, ls, node, depthl, strategy, heu):
    nodes = []; h = 0
    if(depthl >= node._d):
        for (action, result, cost) in ls:
            if result._nodes:
                dmin = min([self.distance(result._current, n) for n in result.
                    _nodes])
                if heu == 'h1':
                    h = dmin
                elif heu == 'h0':
                    for a, b in itertools.combinations(result._nodes, 2):
                        h = dmin+min([self.distance(a, b)])
            s = treeNode(result, strategy, node, float(cost), action, h)
            nodes.append(s)
    return nodes
```

5.2.2 Main

When performing an iterative deepening search (IDS), we were not able to, by giving it an increase in depth, find the solution unless that increase was exactly the depth at which the solution was, so something was not working as expected.

The solution to this error has been added in the *limSearch* method, we need to empty the *visitedList* every time we execute a search because if not, the second time we called the method in an iterative way, they visited list must remain as it was when the first iteration finished, the nodes will all be visited and therefore no solution is going to be found unless that first iteration is at the exact same depth where the solution is found, which in most cases using *IDS*, is not.

1. **limSearch()**: note the called in the second line to empty the *visitedList*.

```
def limSearch(problem, strategy, depthl, pruning):
    f = frontier(); problem._visitedList = {}
```

5.3 Project Conclusion

We have learned a lot by the making of such a nice lab project like this one is, which have allowed us to put all programming subjects knowledges adquired in previous years together in one and see a real world example like a gps is where search algorithms are used *day-to-day* in lots of devices.

We would like to thank our teacher for the help provided in the making, been kindly attended in his office whenever we have needed it.

References

[1] Python Sorted Containers. Sortedkeylist. Website, 2014.

[2] Python Software Foundation. bisect - array bisection algorithm. Website, 2018.

[3] Python Software Foundation. heapq - heap queue algorithm. Website, 2018.

[4] Python Software Foundation. queue - a synchronized queue class. Website, 2018.

[5] Tobias Leupold. gpx2svg. Website, 2005.

[6] Quantifiedcode. Python anti-patterns. Website, 2018.

[7] Stackoverflow. Python equivalent of java's compareto(). Website, 2012.

[8] OpenStreetMap Wiki. Gpx. Website, 2018.