Intelligent Systems

- Milestone 4 -

Juan Garrido Arcos
juan.garrido3@alu.uclm.es

Pedro-Manuel Gómez-Portillo López
pedromanuel.gomezportillo@alu.uclm.es

Index

1.- Abstract

This is a cumulative paper holding the documentation and tests developed for all the different milestones of the laboratory assessments on Intelligent System subject.

On first milestone it was needed reading an OSM file and storing it into a hash table, including in each of our map node the information about its identifier, its geographical coordinates and a list of its adjacent nodes (a tuple holding the identifier of this node, the street through we can reach it and the distance).

Second milestone was aimed mainly to implement a fringe by means of the data structure we considered more appropriate. For doing so, two data structure we studied and the results may be seen on section 2. Also we were asked to fill it with with all the tree nodes it was able to hold, starting by the node of the initial state and continuing obtaining child nodes indefinitely.

Third milestone was related to developing a basic versions of a search algorithm following 3 different strategies; breath-first search, depth-first search (depth-limited search and iterative deeping search) and uniform cost.

On this forth milestone we were asked to implement a prune on the tree in oder to avoid exploring states in the case their values are worse. Furthermore it was needed to generate the output of the program on a GPX file and informing about the basic statistics of the program, as time taken to reach the solution (temporally complexity), number of generated tree nodes (spatial complexity) and final cost of the solution.

It was programmed on Python 2.7, using emacs 24 as IDE and Git+Bitbucket as code control version. The result of the comparative between the studied data structures have been obtained under an Intel CoreDuo Ubuntu 14.04 with 3,8 GB of RAM.

2.- Comparative between data structures

We were asked to choose an appropriate data structure for storing the fringe of our problem. For so, we studied two different alternatives,

· basic Python list managed by **bisect**, a library named so because it uses a basic bisection algorithm for orderly inserting elements into a list
API - https://docs.python.org/2/library/bisect.html

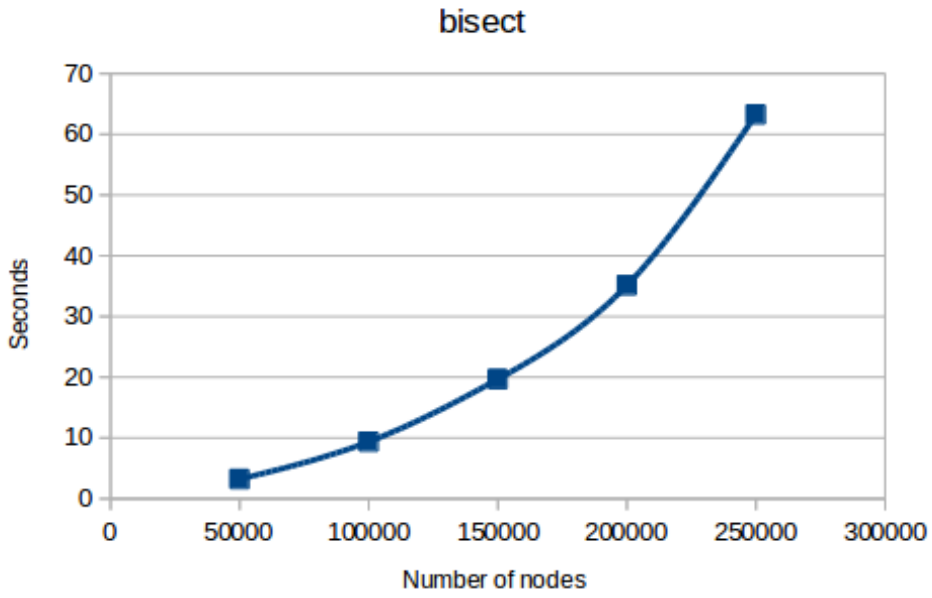
· **heapq**, a library which implements a priority queue by means of a binary tree
API - https://docs.python.org/2/library/heapq.html

2.1.- bisect

By using **bisect** for managing our fringe, we obtained the following results after one minute of execution,

Number of nodes in the fringe	Seconds to accomplish so
50,000	3.218
100,000	9.379
150,000	19.672
200,000	35.112
250,000	63.254

By using the GNU/Linux tool *System Monitor* we were able to compute how many MB of RAM the program generated, and it was not even a 15% of our PC capacity, just around 400MB.



It can be seen that the plot of this function is following an exponential growth.

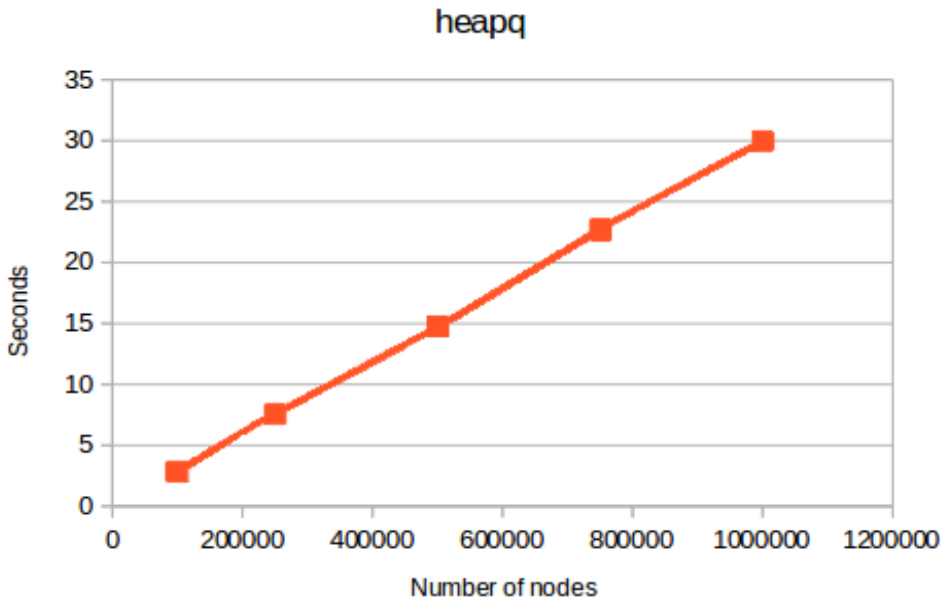
2.2.- heapq

After studying **heapq** we collected results shown bellow,

Number of nodes in the fringe	Seconds to accomplish so
100,000	2.808
250,000	7.561
500,000	14.723
750,000	22.696
1,000,000	29.949

Passed 30 seconds we had to kill the process as the computer in which we were testing it ran out of RAM; a total of 70% of 3,8 GB of RAM were occupied just before killing the process.

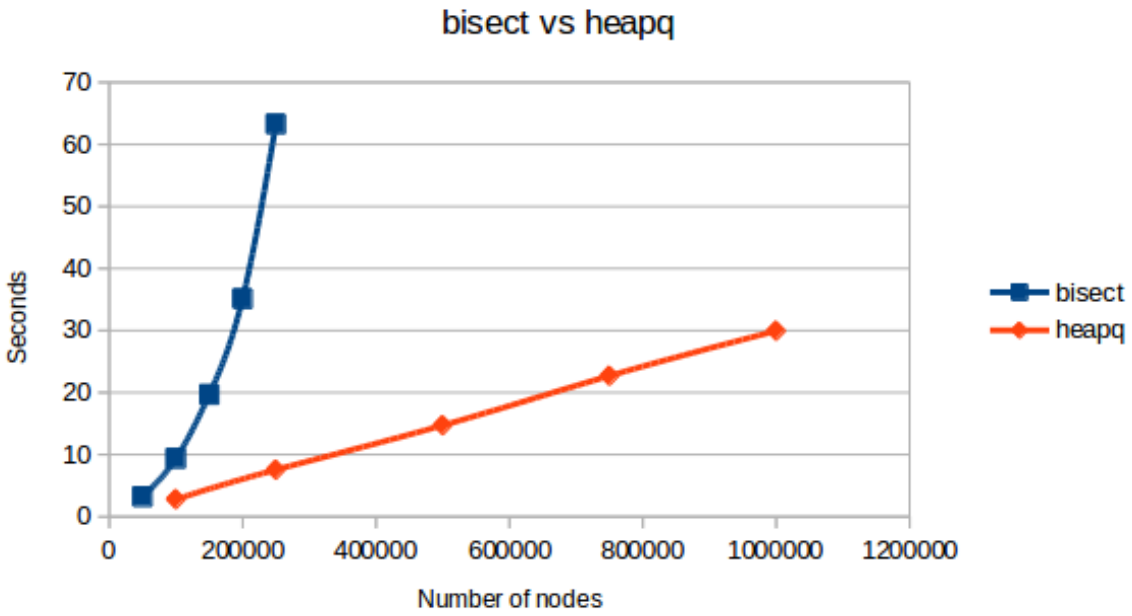
Several times it was needed hard-reseting the PC in order to stop it.



As it is shown, the function follows a linear growth.

2.3.- Conclusions

The comparative among the experimental result of both data structures is displayed in the following graph,



It is clearly visible that bisect behaves considerably worse. In this abridged example bisect has been executed for more than a minute, while heapq just for 30 seconds, and the second one has managed to insert four times the number of nodes bisect has been able to. Furthermore, by the graph we can deduce that gap in efficiency will heighten very quickly as time goes on.

In conclusion, it is clear that we will be **implementing our fringe by means of the heapq** library, as it is much faster than the bisect library.

3.- Equivalence between pseudocode and Python code

We have ensured that our code is equivalent to the pseudocode described in the requirement but working on Python 2.7.

3.1.- Search

This method is devoted to explore incrementally the depth of the map until a solution is obtained or until it exceed its maximum allowed depth.

```
def search(p, strategy, max_depth=10000, incr_depth=1):
    current_depth = incr_depth
    solution = None
    while not solution and current_depth <= max_depth:
        solution = bounded_search(p, strategy, current_depth)
        current_depth = current_depth + incr_depth

    return solution
```

```
Busqueda(Prob,estrategia,Prof_Max,Inc_Prof):Solución o Nada.
  Prof_Actual<-Inc_Prof
  Solución<- Ninguna
  Mientras (not Solución) and (Prof_Act=< Prof_Max):
     Solución= Busqueda_Acotada(Prob,estrategia,Prof_Actual)
     Prof_Actual <- Prof_Actual + Inc_Prof
  Fin_Mientras
  devolver Solución
```

3.2.- Bounded search

This method is in charge of keep inserting nodes in the frontier depending on their value and retrieving the one with the minimum value until a goal state is achieved or the fringe gets empty.

```
def bounded_search(p, strategy, max_depth=10000):


    frontier = []
    initial_node = Node_Tree(p.initial_state)

    heapq.heappush(frontier, initial_node)
    solution = False

    while not solution and len(frontier) <> 0:
        current_node = heapq.heappop(frontier)
        if p.isGoal(current_node.state):
            solution = True
        else:
            successors_list = p.state_space.getSuccessors
            (current_node.state,  p.hash_table)
            successor_nodes = create_nodes(successors_list,
                                current_node, max_depth, strategy)
             for node in successor_nodes:
                heapq.heappush(frontier, node)


    if solution:
        return create_solution(current_node)
    else:
        return None
```

```
Búsqueda_Acotada (Prob,estrategia,Prof_Max): Solución o
NO_Solución

  frontera <- crear_frontera_vacia()
  n_inicial<- crea_nodo(padre<-Nada,estado<-
  Prob.EstadoInicial(),prof<-0,costo<-0,valor<-0)
  frontera.insertar(n_inicial)
  solución<- Falso

  Mientras No solución y No frontera.esVacia() hacer
     n_actual<-selecciona_nodo(frontera)
     si Prob.Estado_Meta(n_actual.Estado) entonces
        solución<-True
     si_no
        LS<-Prob.Sucesores(n_actual.Estado)

        LN<-CreaListaNodosArbol (LS,nodo_actual,
                                 Prof_Max,estrategia)
        frontera.insertaLista(LN)
    fin_si
  Fin_Mientras

  si solución entonces
      devolver CreaSolución(n_actual)
  si_no
      devolver NO_Solución
  fin_si
Fin_Búsqueda
```

4.- Interesting part of code

In this section, most interesting part of the code will be discussed.

4.1.- Get successors

getSuccessors() is a function located onto the State_Space class. This method provides all the successors of a given state, which are tuples *action*, *state*, *cost*.

On **line 3** the hash table previously built is used to acquire the map node with the same identifier stored on this data structure, and then the list with all its adjacent nodes is obtainer.

On **line 5**, the for-loop iterates over each element in the previous list and builds the wished tuple; *action* and *cost* are directly obtained (**lines 6 and 7**) , but a *state* has a set of objective nodes which inherits from its parent. Thus, that set is obtained and they will be stored onto the child's objetive nodes those ones different to that child.

On **line 14**, still into the loop, each generated tuple will be stored and finally, on **line 16**, it will return all the tuples.

```
1   def getSuccessors(self, prev_state, hash_table):
2       successors_list = []
3       adj_list = hash_table[prev_state.node_map.key].ady_list[:]
4
5       for adjacent_node in adj_list:
6           action = adjacent_node.street_name
7           cost = adjacent_node.distance
8
8           current_state = State(hash_table[adyacent_node.key])
10          for objective_node in prev_state.objective_nodes:
11              if objective_node <> current_state.node_map.key:
12                  current_state.objective _nodes.append(objective_node)
13
14          successors_list.append((action, current_state, cost))
15
16      return successors_list
```

4.2.- Create solution

This method is devoted to keep retrieving the parent nodes from the final node until it finds the initial node (**line 5**, *while parent(node) <> null*) and storing them onto a LIFO queue. Thereby, the search's resulting path will be generated.

Afterwards, on **line 10**, the obtained path is written onto a .gpx file.

```
1   def create_solution(final_node):
2
3     current_node = final_node
4     stack = [current_node]
5
6     while current_node.parent is not None:
7        stack.append(current_node.parent)
8        current_node = current_node.parent
9
10    generateGPX(stack)
11
12
13  def generateGPX(stack):
14    output_file = 'output/path.gpx'
15    print "Path generated on " + output_file
16
17    with open(output_file, 'w+') as f:
18       f.write('<?xml version="1.0" encoding="UTF-8"?>\n<gpx
      version="1.1">\n\t<name>Finalroute</name>\n\t\t<trk><name>
      Route</name><number>1</number><trkseg>\n')
19
20       while len(stack)>0:
21           f.write(stack.pop().toGPX())
22
23       f.write('\t</trkseg></trk>\n</gpx>')


24   class Node_Tree:
25      def toGPX(self):
26          return '\t\t<trkpt lat="' +str(self.state.node_map.latitud) +
          '" lon="' + str(self.state.node_map.longitud) +
          '"><ele>628</ele><time>2007-10-14T10:09:57</time></trkpt>\n'
```

4.3.- Creating and pruning nodes

This method is the one really in charge of creating the nodes which will be inserted on the frontier. In here, a node is created from the different variables of its parent (**lines 7-12**) and then grant it with a value according to the search algorithm strategy we are currently using (**lines 14-31**).

If option **prune** has **not** been selected (**line 32**) the generated node will be directly inserted on the list of successor nodes (**lines 39-42**). Otherwise, it will be checked if the the current node have been previously visited and if so if the value already stored is greater than the one of the current node (**line 33**).

If this last condition is satisfied that value will be updated (**line 34**) and the node will be stored on the frontier (**line 36**). The previous node is not eliminated due to two main reasons; it is not known whether it is still on the frontier or not, so it will be very inefficient traveling all the frontier looking for it, and even if it is in the fringe the current node will be stored before it, so that worse value may not be ever used.

```
1  def create_nodes(successors_list, parent_node, max_depth, strategy):
2
3      successor_nodes = []
4
5      for succ in successors_list:
6
7          state = succ[1]
8          cost = parent_node.cost + succ[2]
9          street = succ[0]
10         depth = parent_node.depth + 1
11         parent = parent_node
12         value = 0
13
14         if strategy == Searching_Strategies.BFS:
15             value = depth
16
17         elif strategy == Searching_Strategies.DFS:
18             value = -depth
19
20         elif strategy == Searching_Strategies.DLS:
21              value = -depth
22
23         elif strategy == Searching_Strategies.IDS:
24             value = -depth
25
26         elif strategy == Searching_Strategies.UC:
27             value = cost
28
29         elif strategy == Searching_Strategies.AStar:
27             if len(state.objetive_nodes):
28                 value = cost +  distance_on_unit_sphere(state.node_map,
                                problem.hash_table[state.objetive_nodes[0]])
29             else:
30                 value = cost
31
```

```
32      if prune == 'Y':
33           if value < problem.hash_table[state.node_map.key].BestValue:
34                problem.hash_table[state.node_map.key].BestValue = value
35                current_succesor = Node_Tree(state, cost, street,
                                                depth, parent, value)
36                successor_nodes.append(current_succesor)
37
38       else:
39           current_successor = Node_Tree(state, cost, street,
                                           depth, parent, value)
40           successor_nodes.append(current_successor)
41
42     successor_nodes.append(current_successor)
43
44     return successor_nodes
```

5.- Search tests

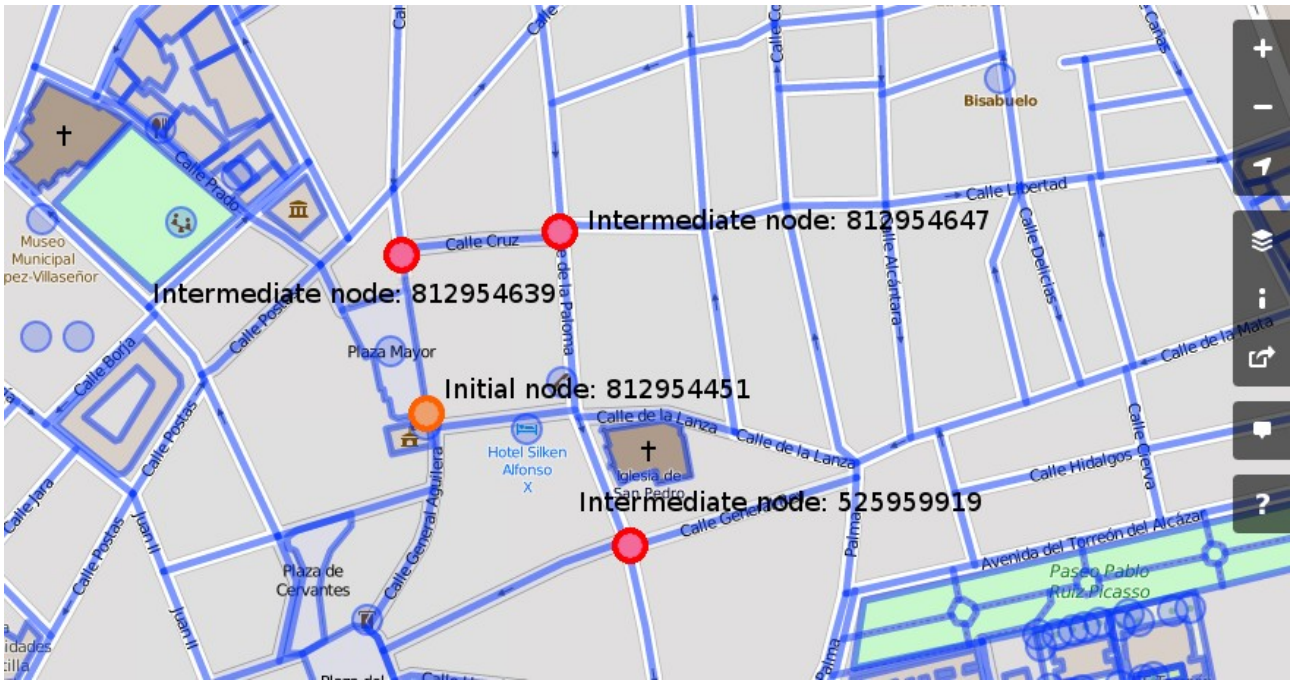
In this section several test relating all the different search algorithm implemented will be shown in detail.

Due to the current implementation (a simplification state ≈ node), if prune is chosen a node will only be traveled once, so no backtracking will be possible. Due to that, on tests where it is needed to go back over the traveled path to find the solution (as in **test3**) prune will guard against obtaining a result.

5.1.- Depth-first search

This test corresponds to **make test5 without prune.**

In this case a search with the initial node 812954451 and looking for the nodes [812954647, 525959919, 812954639] was executed, but in the case of DFS the program did not find any solution path.



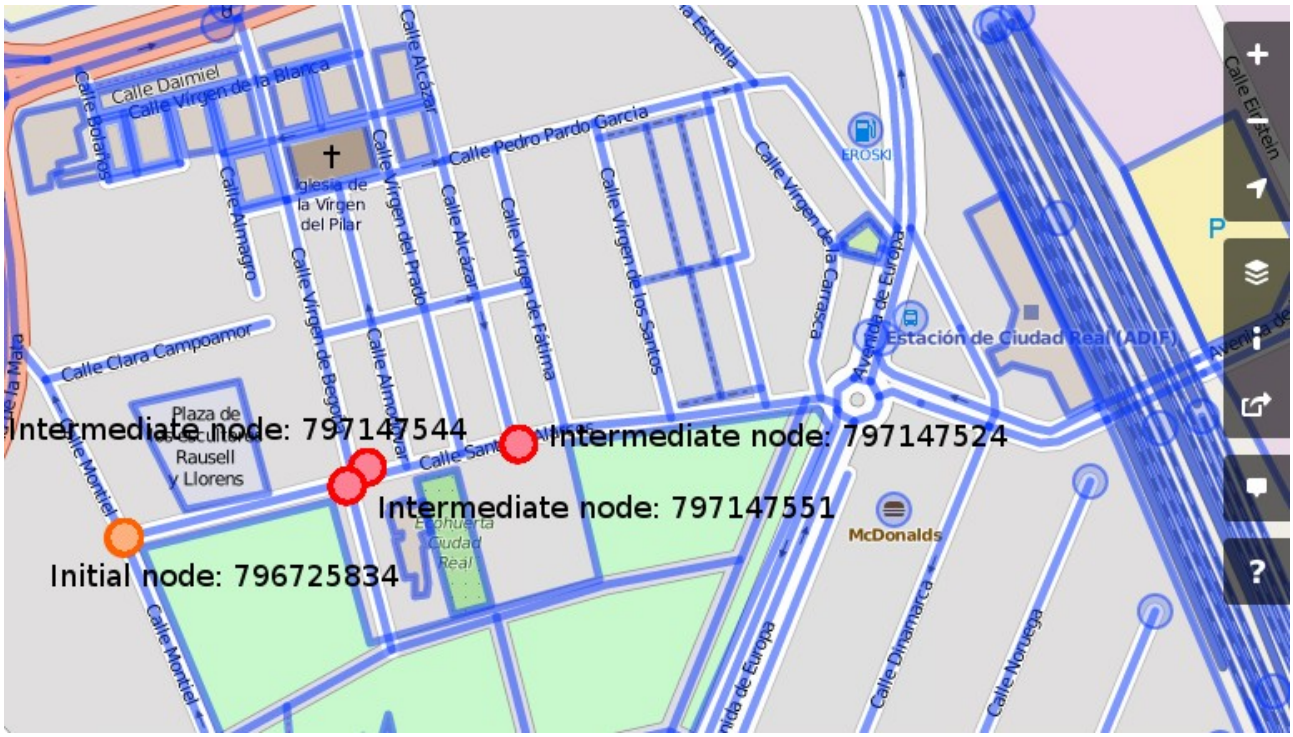
Nonetheless, the program found a solution both for BFS and for UC search algorithms.

Thus, a different input was tried, resulting into the following test.

This test corresponds to **make test6 with prune**.

In this case a search with the initial node 796725834 and looking for the nodes [797147544, 797147524, 797147551] was executed, but in the case of DFS the program did not find any solution path.

Node #796725834, value: 0, street: None
Node #797147551, value: -1, street: Calle Santa María de Alarcos
Node #797147544, value: -2, street: Calle Santa María de Alarcos
Node #797147608, value: -3, street: Calle Santa María de Alarcos
Node #797147538, value: -4, street: Calle Santa María de Alarcos
Node #796725833, value: -5, street: Calle Santa María de Alarcos
Node #797147524, value: -6, street: Calle Santa María de Alarcos



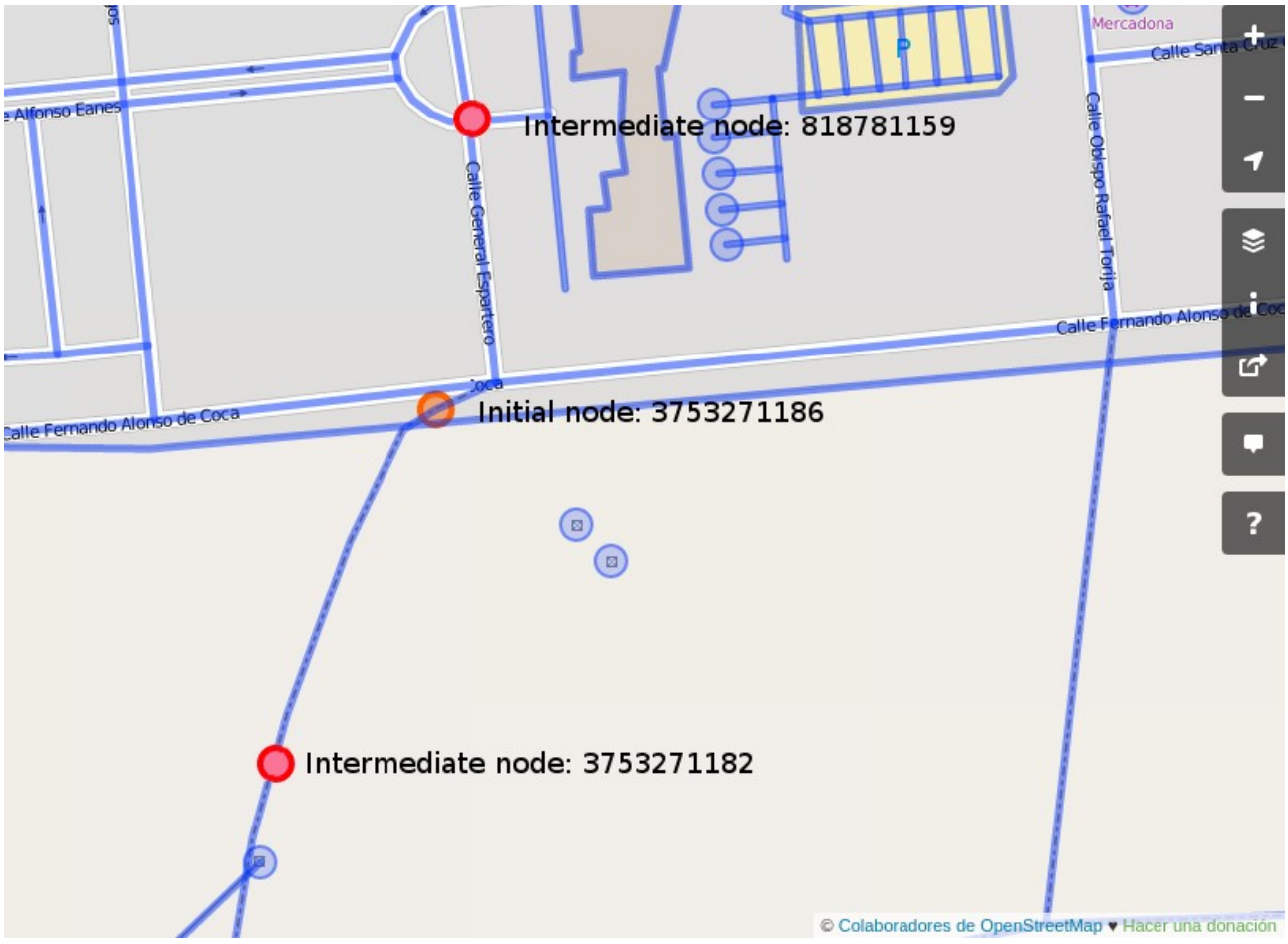
Several tests were performed until a valid solution was found; DFS turned out to be very susceptible to endless loops.

5.2.- Breath-first search

This test corresponds to **make test3 without prune**.

This time a search with the initial node 3753271186 and looking for the nodes [3753271182, 818781159] was executed, generating the next result.

Node #3753271186, value: 0, street: None
Node #818781140, value: 1, street: Autovía Extremadura - Comunidad Valenciana
Node #818781159, value: 2, street: Calle General Espartero
Node #818781140, value: 3, street: Calle General Espartero
Node #3753271186, value: 4, street: Autovía Extremadura - Comunidad Valenciana
Node #3753271185, value: 5, street: Autovía Extremadura - Comunidad Valenciana
Node #3753271184, value: 6, street: Autovía Extremadura - Comunidad Valenciana
Node #3753271183, value: 7, street: Autovía Extremadura - Comunidad Valenciana
Node #3753271182, value: 8, street: Autovía Extremadura - Comunidad Valenciana



Should we try to execute this this pruning the fringe, no solution will be found.

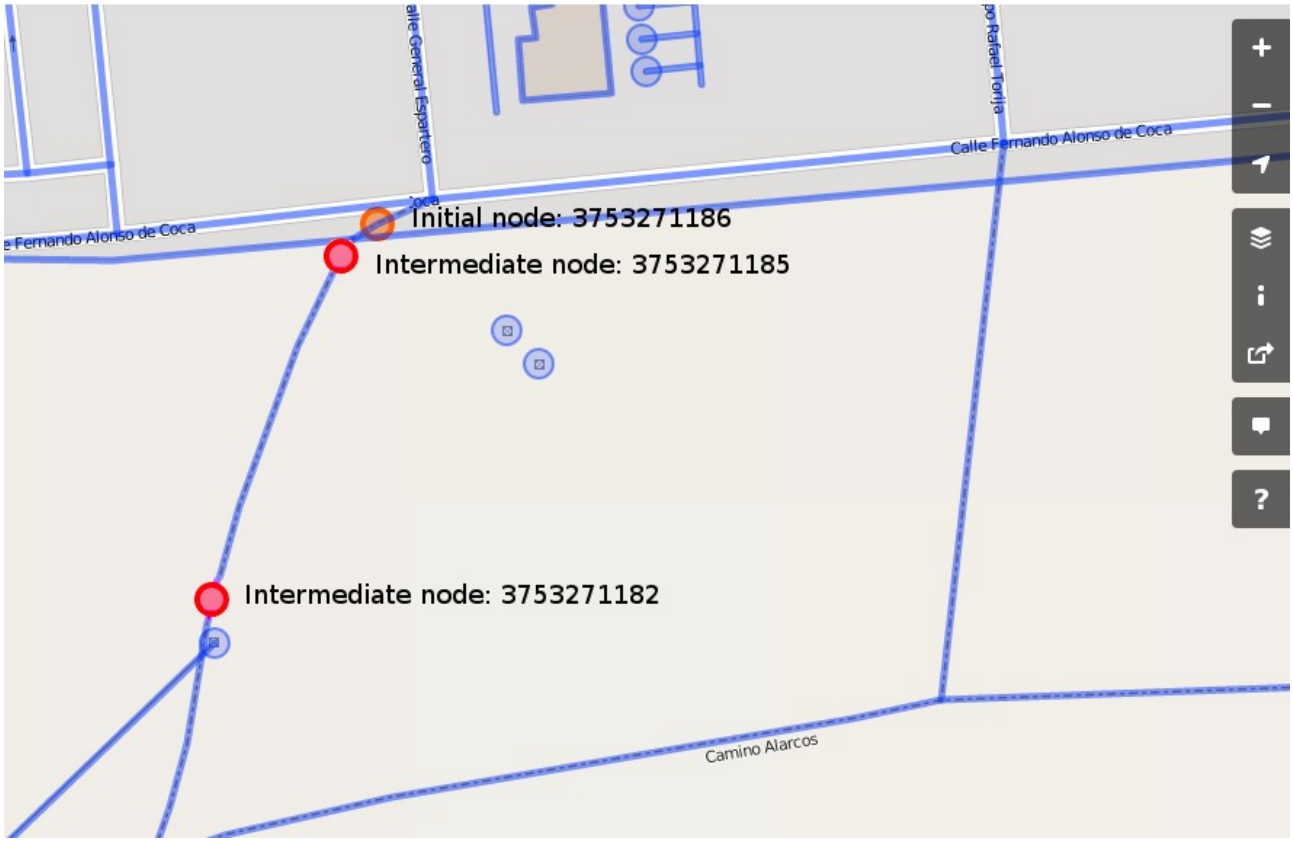
If this input happened to be tested with a Uniform Cost or with an Depth-first search algorithm, it would never end executing.

5.3.- Uniform Cost

This test corresponds to **make test2 with prune**.

This kind of search algorithm select the node closer to the initial node. This time a search with the initial node 3753271186 and looking for the nodes [3753271185, 3753271182] was executed, generating the following resulting path.

Node #3753271186, value: 0, street: None
Node #3753271185, value: 22.7959399741, street: Autovía Extremadura - Comunidad Valenciana
Node #3753271184, value: 95.611870499, street: Autovía Extremadura - Comunidad Valenciana
Node #3753271183, value: 204.038579428, street: Autovía Extremadura - Comunidad Valenciana
Node #3753271182, value: 280.641717818, street: Autovía Extremadura - Comunidad Valenciana



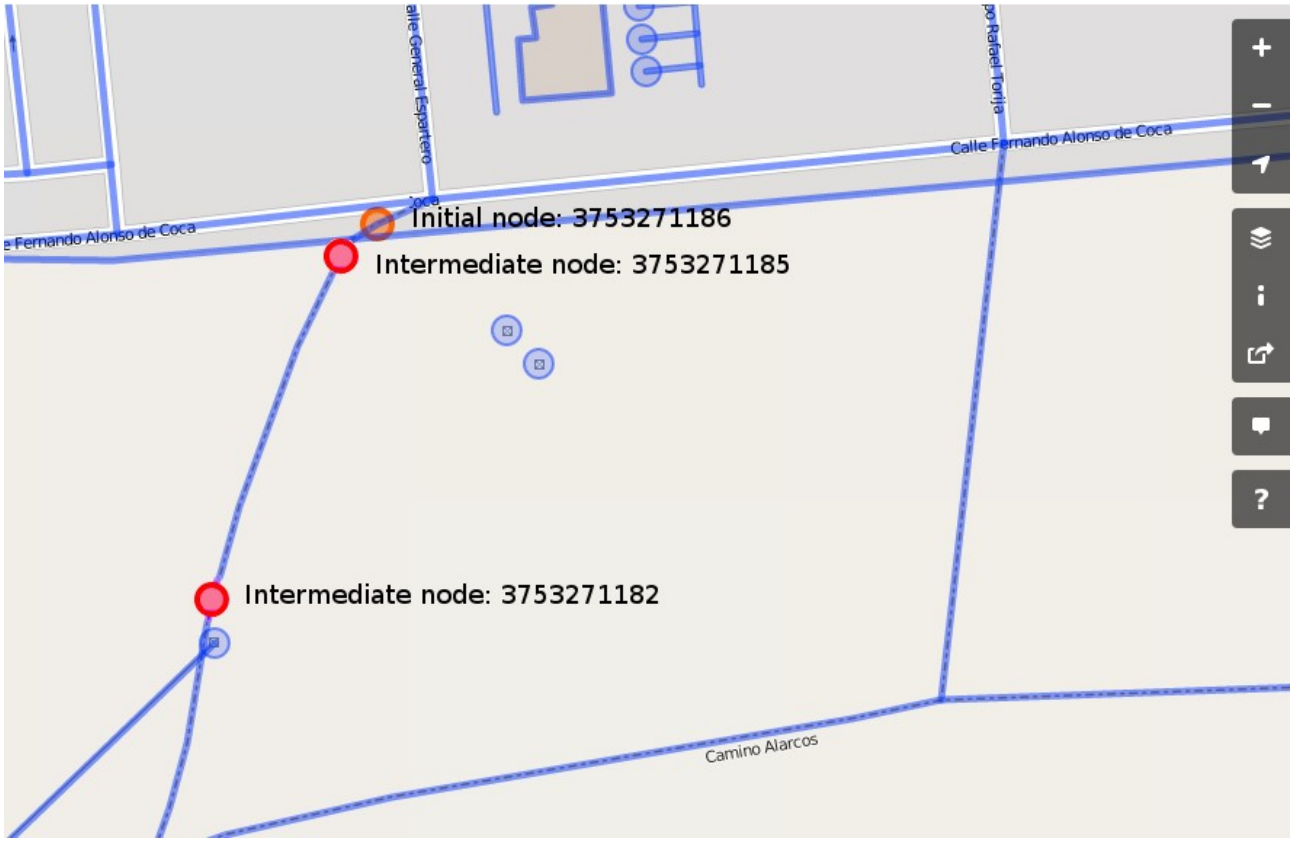
This input generates a path for DFS and BFS as well.

5.4.- A*

This test corresponds to **make test2 without prune**.

This search algorithm select the node closer to the initial node. This time a search with the initial node 3753271186 and looking for the nodes [3753271185, 3753271182] was executed, generating the following resulting path.

Node #3753271186, value: 0, street: None
Node #3753271185, value: 22.7959399741, street: Autovía Extremadura - Comunidad Valenciana
Node #3753271184, value: 95.611870499, street: Autovía Extremadura - Comunidad Valenciana
Node #3753271183, value: 204.038579428, street: Autovía Extremadura - Comunidad Valenciana
Node #3753271182, value: 280.641717818, street: Autovía Extremadura - Comunidad Valenciana



6.- Basic manual for using the program

Should you wish to execute this program, a Makefile has been written for this purpose.

6.1.- Makefile

Being at the root directory of this project, execute $make:

- build: to download the .osm file of the map of Ciudad Real (~3MB) and apply the appropriate permissions to the executable files
- test<n>: this command will execute the program with different right nodes
- test_error: this option will execute the program with an invalid node, thus any search will ever end
- clean: this option will remove the generated binaries and all the temporally files.
- remove_all: for removing the .osm file you have previously downloaded

Should you want to execute a custom search, you need to type on a console:

python src/main.py <InitialNode> <latMin> <lonMin> <latMax><lonMax> <objetiveNode1>… <objetiveNodeN>

6.2.- Example of use:

```
python src/main.py 3753271186 -3.9719000 38.9650000 -3.8847000 39.0062000
3753271182 3753271186

Search algorithm? (BFS = 0, DFS = 1, DLS = 2, IDS = 3, UC = 4, AStar = 5)
0
Execute prune? (Y/N)
Y
Looking for nodes ['3753271182', '3753271186']
Analysis of the .osm file: lines: 39524, nodes: 10702, connections: 12023
Path generated on output/path.gpx
Path generated on output/path.txt

******** STATISTICS *********
Final cost: 249.630459376
Time taken to accomplish the search: 0:00:00.016651
Number of generated nodes: 1086
```

7.- Bibliography

https://docs.python.org/2/library/bisect.html

https://docs.python.org/2/library/heapq.html

https://docs.python.org/2/library/stdtypes.html

https://docs.python.org/2/library/operator.html

https://docs.python.org/2/tutorial/datastructures.html

http://stackoverflow.com/questions/19979518/what-is-pythons-heapq-module

http://stackoverflow.com/questions/4713088/how-to-use-git-bisect

http://extensions.libreoffice.org/extension-center/code-colorizer-formatter

http://tohtml.com/python/