IS Lab Project |A1-06

Pablo Alcázar Morales Güray Meriç Diego Pedregal Hidalgo

https://github.com/thmasker/A1-06

1 *Task1*

For this task, we found a simple, effective and easy to use library on Python, called *ElementTree*, so after seeing other options like *minidom*, we decided to use it. After that, we created the required *Graph* class, which constructor parses the chosen *.graphml* file, and stores both nodes and edges in two independent variables. Both *belongNode* and *positionNode* methods are really easy to understand, so we won't go on further explanations.

The most complex of the three methods created is *adjacentNode* (see *Figure 1.1*). This method goes through all the node's edges, and appends to the adjacency list those which source node is the *nodeid* we are looking for. Firstly, we append each edge to the adjacency list with their *name* and *length* fields as *SinNombre* and *NULL* respectively. After that we search both fields on the inner data of each edge using the keys dictionary created on the constructor, and we refill both attributes with the real ones. If anyone of them is missing, we'll know that, as it will appear as *SinNombre* or *NULL* in the adjacency list returned.

```
def adjacentNode(self, nodeid):
        adjacencyList = []

        if self.belongNode(nodeid):
            for edge in self.edges:
                if edge.get('source') == nodeid:
                    adjacencyList.append((edge.get('source'), edge.get('target'),
    'SinNombre', 'NULL'))

                    for data in edge:
                        if data.get('key') == self.keys['name']:
                            lst = list(adjacencyList[-1])
                            lst[2] = data.text
                            adjacencyList[-1] = tuple(lst)
                        elif data.get('key') == self.keys['length']:
                            lst = list(adjacencyList[-1])
                            lst[3] = data.text
                            adjacencyList[-1] = tuple(lst)
        else:
            adjacencyList = '[ERROR] Node does not exist on the node'

        return adjacencyList

Keys Dictionary

for key in self.graph.findall('default:key', self.ns):      # Store edge keys
            if key.get('for') == 'edge':
                self.keys[key.get('attr.name')] = key.get('id')
```

Figure 1.1 Method *adjacentNode*

```
for key in self.graph.findall('default:key', self.ns):      # Store edge keys
            if key.get('for') == 'edge':
                self.keys[key.get('attr.name')] = key.get('id')
```

Figure 1.2 Keys dictionary

1.1 Some results

We coded a script (*tests.py*) to be able to test the results given by the program. We show some of them next:

```
Is this node on the graph? True
Position (latitude and longitude): ('-4.3544889', '38.8808616')
Adjacency list: [('1851273807', '4587612273', 'Plaza del Pilar', '23.656'),
    ('1851273807', '1851273984', 'Plaza del Pilar', '21.923'), ('1851273807',
    '1851274092', 'Carretera de los Pozuelos', '98.36399999999999')]
```

Figure 1.1.1 Result for *nodeid* 1851273807 from *Abenójar.graphml*

```
Is this node on the graph? True
Position (latitude and longitude): ('-3.9227797', '38.9936719')
Adjacency list: [('764039207', '2711270344', 'Calle Pozo Santa Catalina',
    '135.996'), ('764039207', '764039210', 'Calle Extramuros de Calatrava',
    '50.307')]
```

Figure 1.1.2 Result for *nodeid* 764039207 from *Ciudad Real.graphml*

```
Is this node on the graph? True
Position (latitude and longitude): ('-4.8550238', '39.5363048')
Adjacency list: [('2012263661', '2012263377', 'Calle Carmen', '51.09'),
    ('2012263661', '2012263373', 'Calle Carmen', '62.747'), ('2012263661',
    '2012263518', 'Calle Carmen', '119.99600000000001')]
```

Figure 1.1.3 Result for *nodeid* 2012263661 from *Anchuras.graphml*

1.2 Timing

```
Graph file loaded in:  0.009773681000005752  seconds
Adjacency list loaded in:  0.0003832729999970752  seconds
```

Figure 1.2.1 Result for *nodeid* 4753226234 from *Abenójar.graphml*

```
Graph file loaded in:  0.07164645499999978  seconds
Adjacency list loaded in:  0.0023671510000013996  seconds
```

Figure 1.2.2 Result for *nodeid* 764039166 from *Ciudad Real.graphml*

```
Graph file loaded in:  0.007518905999999603  seconds
Adjacency list loaded in:  0.0008977220000012665  seconds
```

Figure 1.2.3 Result for *nodeid* 4928063639 from *Anchuras.graphml*

2 *Task2*

On this task, we implemented five different classes:

1. *TreeNode.* This class has not anything special, but the only required attributes: *parent, state, cost of the path, action, current depth* and *f*, which is a random number between 1 and 10 000.

2. *State*. As we should not consider that the list of nodes given will be already ordered, we implemented the typical *mergeSort* algorithm to order the elements in an optimized way, which running time in asymptotic notation is:

$$O(n \log n) \tag{1}$$

```
def mergesortNodes(self, nodesList, leftIndex, rightIndex):
  if leftIndex < rightIndex:
    middleIndex = (leftIndex + rightIndex) // 2
    self.mergesortNodes(nodesList, leftIndex, middleIndex)
    self.mergesortNodes(nodesList, middleIndex+1, rightIndex)
    self.mergeNodes(nodesList, leftIndex, middleIndex, rightIndex)

def mergeNodes(self, nodesList, leftIndex, middleIndex, rightIndex):
  k = leftIndex
  tempLeft, tempRight = [0] * (middleIndex - leftIndex + 1), [0] * (
    rightIndex - middleIndex)

  for i in range(0, middleIndex - leftIndex + 1):
    tempLeft[i] = nodesList[leftIndex + i]

  for j in range(0, rightIndex - middleIndex):
    tempRight[j] = nodesList[middleIndex + 1 + j]

  i, j = 0, 0

  while (i < len(tempLeft)) and (j < len(tempRight)):
    if tempLeft[i] <= tempRight[j]:
      nodesList[k] = tempLeft[i]
      i += 1
    else:
      nodesList[k] = tempRight[j]
      j += 1

    k += 1

  while i < len(tempLeft):
    nodesList[k] = tempLeft[i]
    i += 1
    k += 1

  while j < len(tempRight):
    nodesList[k] = tempRight[j]
    j += 1
    k += 1
```

Figure 2.1 *mergeSort* implementations

3. *StateSpace*. This class basically generates the successors for every specific *State* on the problem. Obviously, the main functionality is the function *successors*, which inserts in a list the action to do, its cost, and the new *State* caused for each adjacent node of the current state.

```
def successors(self, state):
        successorsList = []

        if self.belongNode(state.currentPosition):
            adjacencyList = self.graph.adjacentNode(state.
    currentPosition)
            for node in adjacencyList:
```

```
7                   try:
8                       newState = S.State(node[1], state.nodesRemaining.
    remove(node[1]))
9                   except ValueError:
10                      print("Node " + newState.currentPosition + " is
    not in nodesRemaining\n")
11                      return
12                  acci = "I'm at " + state.currentPosition + "and I go
    to " + newState.currentPosition
13                  costActi = node[3]
14                  successorsList.append((acci, newState, costActi))
15          else:
16              print(state + " does not belong to the graph\n")
17
18          return successorsList
```

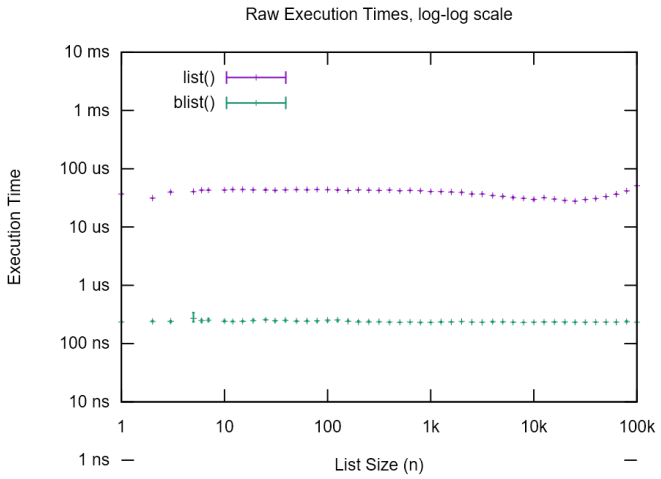
Figure 2.2 *successors* implementation

4. *Problem.* In this case, there is not much difficulty, as we only read the initial state of the problem from a *.json* file (thanks to *json* library). We also created the method *isGoal*, which returns *True* if the goal has been reached and *False* otherwise.

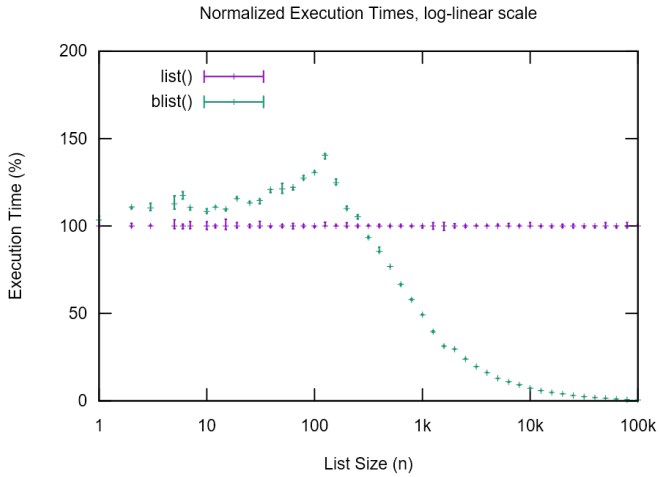
5. *Frontier.* The key in this class is the implementation of the frontier. To do that, we have used the *blist* library as it is a powerful tool to deal with python lists as it makes insertions[1] and removals[2] timing

$$O(\log n) \tag{2}$$

After performing a test available on the library (*speed_test.py*) we obtained some charts about the execution time for large amounts of data.



[1] *blist* insertions documentation *here*
[2] *blist* removals documentation *here*



In addition, after performing the stress test, we decided to stop the program after reaching the 80% of the memory utilization using the *psutil* library.

```
if psutil.swap_memory()[3] >= 80:
  print("Not enough space in memory")
  raise MemoryError
```

Figure 2.3 Memory utilization control