

# 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.

```
1 def adjacentNode(self, nodeid):
2     adjacencyList = []
3
4     if self.belongNode(nodeid):
5         for edge in self.edges:
6             if edge.get('source') == nodeid:
7                 adjacencyList.append((edge.get('source'), edge.get('target'),
8                                     'SinNombre', 'NULL'))
9
10                for data in edge:
11                    if data.get('key') == self.keys['name']:
12                        lst = list(adjacencyList[-1])
13                        lst[2] = data.text
14                        adjacencyList[-1] = tuple(lst)
15                    elif data.get('key') == self.keys['length']:
16                        lst = list(adjacencyList[-1])
17                        lst[3] = data.text
18                        adjacencyList[-1] = tuple(lst)
19                else:
20                    adjacencyList = '[ERROR] Node does not exist on the node'
21
22            return adjacencyList
23
24 Keys Dictionary
25 for key in self.graph.findall('default:key', self.ns): # Store edge keys
26     if key.get('for') == 'edge':
27         self.keys[key.get('attr.name')] = key.get('id')
```

Code 1.1 Method *adjacentNode*

```

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

```

Code 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:

```

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

```

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

```

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

```

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

```

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

```

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

## 1.2 Timing

```

1 Graph file loaded in: 0.009773681000005752 seconds
2 Adjacency list loaded in: 0.0003832729999970752 seconds

```

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

```

1 Graph file loaded in: 0.071646454999999978 seconds
2 Adjacency list loaded in: 0.0023671510000013996 seconds

```

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

```

1 Graph file loaded in: 0.0075189059999999603 seconds
2 Adjacency list loaded in: 0.0008977220000012665 seconds

```

Timing 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.

```

1 def __init__(self, parent, state, pathcost, action, d):
2     self.parent = parent
3     self.state = state
4     self.pathcost = pathcost
5     self.action = action
6     self.d = d
7     self.f = random.randint(1, 10000)

```

Code 2.1 *TreeNode* implementation

2. **State.** In this class we simply create objects with the required attributes: *current position*, *nodes remaining* and its *md5* identifier.

```

1 def __init__(self, currentPosition, nodesRemaining):
2     self.currentPosition = currentPosition
3     self.nodesRemaining = nodesRemaining
4     self.nodesRemaining.sort()
5     self.md5checksum = hashlib.md5((str(self.currentPosition) + ",".join(
        str(self.nodesRemaining))).encode()).hexdigest()

```

Code 2.2 *State* implementation

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.

```

1 def successors(self, state):
2     successorsList = []
3
4     if self.belongNode(state):
5         adjacencyList = self.graph.adjacentNode(state.currentPosition)
6
7         for node in adjacencyList:
8             newNodesRemaining = state.nodesRemaining.copy()
9
10            try:
11                newNodesRemaining.remove(node[1])
12            except ValueError:
13                pass
14
15            newState = S.State(node[1], newNodesRemaining)
16            acci = "I'm at " + state.currentPosition + " and I go to " +
newState.currentPosition
17            costActi = node[3]
18            successorsList.append((acci, newState, costActi))
19        else:
20            print(state + " does not belong to the graph\n")
21
22    return successorsList

```

Code 2.3 *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.

```

1 def __init__(self, jsonPath):
2     try:
3         with open(jsonPath, "r") as rststate:

```

```

4         rdata = json.load(rstate)
5     except FileNotFoundError as fnf_error:
6         print(fnf_error)
7         raise SystemExit
8     except json.decoder.JSONDecodeError:
9         print(jsonPath + " is not a .json file")
10        raise SystemExit
11
12    self.StateSpace = SP.StateSpace(rdata["graphmlfile"])
13    self.InitState = S.State(rdata["IntSt"]["node"], rdata["IntSt"]["listNodes"])

```

Code 2.4 *.json* file reading

```

1 def isGoal(self, state):
2     if len(state.nodesRemaining) == 1:
3         if state.nodesRemaining[0] == self.InitState.currentPosition:
4             return True
5
6     if not state.nodesRemaining:
7         return True
8     else:
9         return False

```

Code 2.5 *isGoal* implementation

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<sup>1</sup> and removals<sup>2</sup> timing

$$O(\log n) \quad (1)$$

```

1 def __init__(self):
2     self.frontier = blist([])

```

Code 2.6 *Frontier* initialization

```

1 def insert(self, treeNode):
2     if psutil.swap_memory()[3] >= 80:
3         print("Not enough space in memory")
4         raise MemoryError
5     else:
6         self.frontier.append(treeNode)

```

Code 2.7 *insert* implementation

```

1 def remove(self):
2     try:
3         low = 0
4         for i in range(len(self.frontier)):
5             if abs(self.frontier[i].f) < abs(self.frontier[low].f):
6                 low = i
7         return self.frontier.pop(low)
8     except IndexError as index_error:
9         print(index_error)
10        raise SystemExit

```

Code 2.8 *remove* Implementation

---

<sup>1</sup>*blist* insertions documentation [here](#)

<sup>2</sup>*blist* removals documentation [here](#)

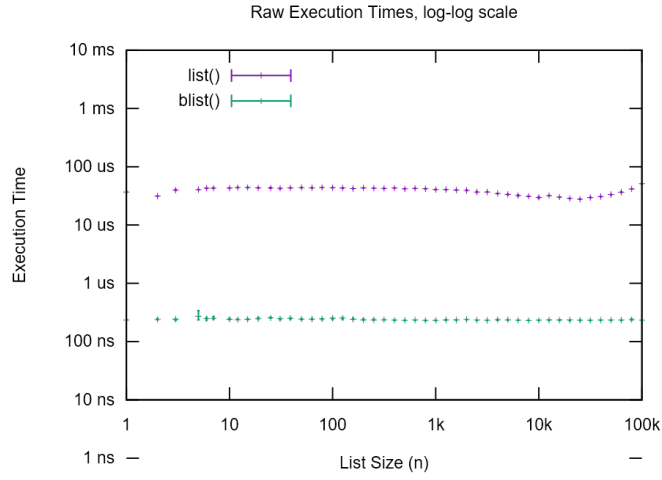


Figure 1: Execution time comparison between python list and *blist*

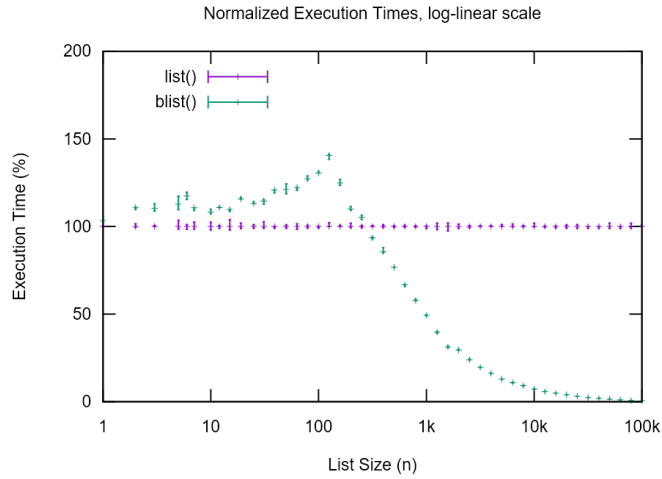


Figure 2: Relative execution time of *blist* compared with python lists

```

1 def isEmpty(self):
2     if not self.frontier:
3         return True
4     else:
5         return False

```

Code 2.9 *isEmpty* Implementation

After performing a test available on the library (*speed\_test.py*) we obtained some charts about the execution time for large amounts of data (see *Figures 1* and *2*).

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.

```

1 if psutil.swap_memory()[3] >= 80:

```

```

2 print("Not enough space in memory")
3 raise MemoryError

```

Code 2.10 Memory utilization control

### 3 Task3

This time we have implemented the search algorithms, following the pseudocode that was given to us. Thus, there is a class called *Search*, in which we organize the search in three main parts:

1. **Search creation.** During this first part, we create a *Search* object and called the function *search*, which will be responsible for starting the search strictly speaking.

```

1 def __init__(self, jsonPath, strategy, max_depth, inc_depth, pruning):
2     self.problem = P.Problem(jsonPath)
3     self.solution = self.search(self.problem, strategy, max_depth, inc_depth,
        pruning)

```

Code 3.1 *Search* constructor

```

1 def search(self, problem, strategy, max_depth, inc_depth, pruning):
2     current_depth = inc_depth
3     solution = None
4
5     while not solution and (current_depth <= max_depth):
6         if strategy == 'ids':
7             solution = self.fenced_search(problem, strategy, current_depth, pruning
            )
8             current_depth += inc_depth
9         else:
10            solution = self.fenced_search(problem, strategy, max_depth, pruning)
11            current_depth += max_depth
12
13     return solution

```

Code 3.2 *search* implementation

As you can appreciate, we differentiate between the execution of *Iterative Deepening Search* and the rest, because depending on this we set the increment in depth as the one entered by the user or as the maximum depth chosen, to make the algorithm iterate only once.

2. **Search itself.** This operation starts whenever the method *search* calls *fenced\_search*. In this case, we create the initial node of the tree from the initial state given by the problem, setting its *f* value as 0. After that, we check that the algorithm has not reached the goal state by calling the function in *Code 2.5*. Then, while we have no solution and the frontier is not empty, we keep searching for the solution, by generating the successor nodes of the current node (see *Code 2.3*) and their corresponding tree nodes (see *Code 3.5*), which will set the *f* value of each node depending on the algorithm executed (depth for *BFS*; -depth for *DFS*, *DLS* and *IDS*; and the cost of the path for *UCS*). Once we have all the new successor nodes, for every one of them we check if it is already in the frontier (see *Code 3.3*):

```

1 if pruning:
2     if node.state.md5checksum not in visitedList:
3         found = False

```

```

4     for i in range(len(frontier.frontier)):
5         if frontier.frontier[i].state.md5checksum == node.state.md5checksum:
6             found = True
7             break
8
9     if not found:
10        frontier.insert(node)
11    else:
12        if abs(frontier.frontier[i].f) > abs(node.f):
13            frontier.frontier.pop(i)
14            frontier.insert(node)
15 else:
16    frontier.insert(node)

```

Code 3.3 Frontier check

- a. If the node is not in the frontier we simply add it.
- b. If it is already in the frontier, we only add it if its  $f$  value is better (lower) than the one in the frontier, substituting the worst one.

```

1 initial_node = TN.TreeNode(None, self.problem.InitState, 0, None, 0)
2 initial_node.f = 0

```

Code 3.4 *initialnode* creation

```

1 def createTreeNodes(self, successorsList, current_node, max_depth, strategy):
2     treeNodesList = blist([])
3
4     if current_node.d < max_depth:
5         for successor in successorsList:
6             node = TN.TreeNode(current_node, successor[1], current_node.pathcost +
7                                 float(successor[2]), successor[0],
8                                 current_node.d + 1)
9
10            if strategy == 'bfs':
11                node.f = node.d
12            elif (strategy == 'dfs') or (strategy == 'dls') or (strategy == 'ids'):
13                node.f = -node.d
14            elif strategy == 'ucs':
15                node.f = node.pathcost
16
17            treeNodesList.append(node)
18
19    return treeNodesList

```

Code 3.5 *createTreeNodes* implementation

3. **Generate solution.** Once we have a solution, we have to go through the last node to the first one to generate the proper solution (see *Code 3.6*). We simply obtain the parent of each solution node and add it to the solution list until we reach the root node, when we stop and return the reversed list.

```

1 def createSolution(self, current_node):
2     solution = blist([])
3
4     if current_node.parent is None:
5         solution.append(current_node)
6
7     while current_node.parent is not None:

```

```

8     solution.append(current_node)
9     current_node = current_node.parent
10
11     solution.reverse()
12
13     return solution

```

Code 3.6 *createSolution* implementation

The last operation we must do is to create the solution file, which consists on writing the actions to follow in a file, along with the cost of the solution and the depth reached (see *Code 3.7*).

```

1 file = open("solution.txt", "w+")
2
3 for node in searching.solution:
4     if node.parent is None:
5         file.write("You already are in the goal node!!!\n")
6     else:
7         file.write(node.action + "\n")
8
9 file.write("\nThe cost of the path is " + str(searching.solution[-1].pathcost
10 ) + "\n")
11 file.write("The solution was found at depth " + str(searching.solution[-1].d
12 )
13
14 file.close()
15
16 print("\nSolution found!! You can see it at " + os.path.abspath("../solution.
17 txt"))

```

Code 3.7 Solution file generation from *main.py*

An example of a solution file generated would be like this:

```

1 I'm at 3 and I go to 8
2 I'm at 8 and I go to 11
3 I'm at 11 and I go to 14
4 I'm at 14 and I go to 13
5 I'm at 13 and I go to 14
6 I'm at 14 and I go to 15
7 I'm at 15 and I go to 18
8 I'm at 18 and I go to 19
9 I'm at 19 and I go to 20
10
11 The cost of the path is 1036.0
12 The solution was found at depth 9

```

### 3.1 Timing

We have taken time measurements for every algorithm with and without pruning, so we can appreciate correctly the important effect pruning has in searching algorithms. The tests included here were done with the file *Tests.graphml* which follows the *RomaniaMap* seen in class.

```

1 Max depth: 10
2
3 Without pruning:
4     BFS:
5         Execution time: 72.3842867 seconds
6     DFS:

```

```

7      Execution time: 74.02109329999999 seconds
8      DLS:
9      Execution time: 78.46946439999999 seconds
10     IDS:
11         Increment of depth: 2
12     Execution time: 78.1188335 seconds
13     UCS:
14         Execution time: 37.7549191 seconds
15
16 With pruning:
17     BFS:
18         Execution time: 0.0055617999999999996 seconds
19     DFS:
20         Execution time: 0.0054572 seconds
21     DLS:
22         Execution time: 0.0057617 seconds
23     IDS:
24         Increment of depth: 2
25     Execution time: 0.0139816 seconds
26     UCS:
27         Execution time: 0.0052702999999999995 seconds

```

Time measurements for initial node 3 (*Arad*), and the nodes remaining were 14 (*Bucharest*), 13 (*Glurglu* and 20 (*Neamt*))