Laboratory practice No. 5: Binary Trees

Juan S. Cárdenas Rodríguez
Universidad EAFIT
Medellín, Colombia
jscardenar@eafit.edu.co

David Plazas Escudero
Universidad EAFIT
Medellín, Colombia
dplazas@eafit.edu.co

November 5, 2017

1) Code for delivery on GitHub

1.a. Graphs implementation

The code for both implementations is inside the `DataStructures.py` file in the codigo folder.

1.b. Most Children

The code can be found inside the `Code.py` file in the codigo folder.

1.c. Test with Dijkstra's alorithm

The tests and the implementation for Dijkstra's algorithm can be found inside the `Code.py` file inside the codigo folder.

1.d. City-Based Graph

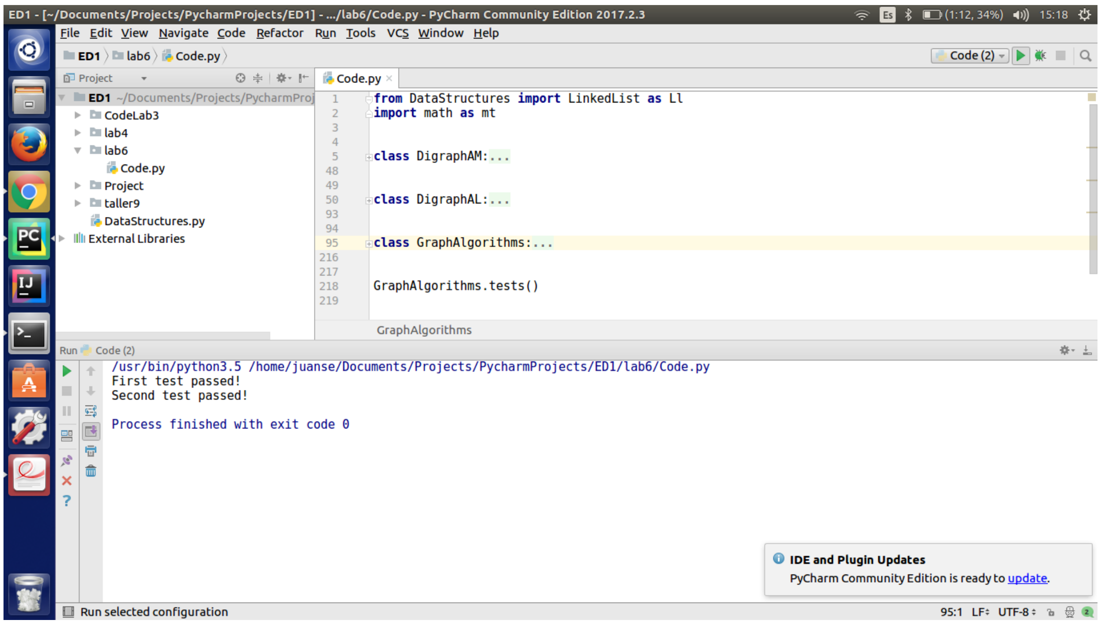
The code can be found in the `City.py` file in the codigo folder.

2) Online Exercises

The solution to the problem can be found in the `Code.py` file in the codigo folder.

3) Other questions

3.a. Proof for 1.3


ED1 - [~/Documents/Projects/PycharmProjects/ED1] - .../lab6/Code.py - PyCharm Community Edition 2017.2.3
File Edit View Navigate Code Refactor Run Tools VCS Window Help
ED1 lab6 Code.py
Code (2)
Project
ED1 ~/Documents/Projects/PycharmProj
CodeLab3
lab4
lab6
Code.py
Project
taller9
DataStructures.py
External Libraries
Code.py
GraphAlgorithms
Run Code (2)
Run selected configuration
95:1 LF: UTF-8:

```python
from DataStructures import LinkedList as Ll
import math as mt


class DigraphAM:...


class DigraphAL:...


class GraphAlgorithms:...


GraphAlgorithms.tests()
```

```
/usr/bin/python3.5 /home/juanse/Documents/Projects/PycharmProjects/ED1/lab6/Code.py
First test passed!
Second test passed!

Process finished with exit code 0
```

IDE and Plugin Updates
PyCharm Community Edition is ready to update.

Figure 1: Proof for exercise 1.3

3.b. Explanation for 1.1

Both are digraphs, since the edges have direction. Both have a data structure to save the edges' information; the first one has a matrix A_{nxn} that inserts a GraphObj in the position i, j if the node i is connected with the j one. The GraphObj has value, weight and a Python dictionary for further attributes of the edge. On the other hand, the second implementation has a linked list with all the children (nodes that are connected to the node we are on). The insertion works similarly in both cases, the user has to insert the parent and the child, making an edge between the parent and the child. Check the `DataStructures.py` file in the codigo folder for the source code.

3.c. On which graphs is better to use the matrix implementation and on which cases is better to use the linked list one?

The matrix representation is more useful when the important things are the connections, not the weights; so, in a simple graph the matrix representation is a fairly good approach to represent it better. But, the priority is to use "greedy algorithms" or a really big sample of data de linked list representation is much better.

3.d. Which implementation is better for a city-based graph?

As we know that it is a graph related to a city, one can note that it is not possible (or HIGHLY unlikely) that all nodes are connected to each other. Therefore, if a matrix representation is used, there will be a lot of empty spaces in the matrix. So it is definitely better to implement the liked list representation for a city graph.

3.e. Which implementation is better for a facebook-based network?

It's better to use the linked list representation because, of the big size of the data (so we can optimize memory) and this type of data is a priority to search for smallest path and more.

3.f. Which implementation is better for a routing table?

The linked list implementation will be more practical since one will be interested in the nodes that are connected; therefore, if this implementation is used, you will know immediately the nodes that are connected, you won't have to check the whole row in the matrix for the nodes connected. Additionally, the matrix one consumes a lot of memory, which is never good.

3.g. Complexity of exercise 2.1

```
@staticmethod
def bi():
    n = int(input())
    graphs = []
    while n != 0:
        graph = DigraphAM(n)
        graphs.append(graph)
        connections = int(input())
        for i in range(connections):
            rel = input().split(" ")
            graph.insert(int(rel[0]), int(rel[1]))
            graph.insert(int(rel[1]), int(rel[0]))
        n = int(input())

    def bi_color(graph):
        color = [0] * graph.n  # c1
        visit = [False] * graph.n # c2

        def dfs(node, active):  # O(n)
            children = graph.search(node)
            color[node] = active
            visit[node] = True
            for child in children:
                if not visit[child]:
                    if active == 1:
                        dfs(child, 0)
                    else:
```

```
                    dfs(child, 1)

        dfs(0, 1)  # O(n)

        for i in range(graph.n):  # c3 * n
            children = graph.search(i)  # Matrix: O(n)
                                        # Linked List: O(m)
            for child in children:  # c4 * n * m
                if color[child] == color[i]: # c3 n * m
                    return False

        return True #

    for graph in graphs:
        if bi_color(graph):
            print("Is bi colorable")
        else:
            print("Is not bi colorable")
```

Then, exercise 2.1 is $O(mn)$, where n is the number of vertex and m is the maximum amount of children that a vertex has.

4) Exam simulation

i.

	0	1	2	3	4	5	6	7
0				1	1			
1	1		1			1		
2		1			1		1	
3								1
4			1					
5								
6			1					
7								

ii.

```
0 -> 3 4
1 -> 0 2 5
2 -> 1 4 6
3 -> 7
4 -> 2
5 ->
6 -> 2
7 ->
```

Vertex	DFS	BFS
0	3742156	3472165
1	0372465	0253467
2	1037546	1460537
3	7	7
4	2103756	2160537
5		
6	2103754	2140537
7		

Table 1: DFS and BFS for each vertex.

iii.

iv. a) $O(n)$

v.

```
def aPath(graph,p,q):
  seen = [False]*graph.size
  return aPathAux(graph,p,q,[],seen)

def aPathAux(graph,p,q,myList,seen):
  if p == q:
    if len(myList) == 0:
      myList.append(q)
    return myList

  children = graph.getSuccesors(p)
  for child in children:
    if not seen[child]:
      seen[child] = True
      myList.append(child)
      return aPathAux(graph,child,q,myList,seen)
  return
```