Introduction to python-igraph

October 3, 2017

1 Introduction to python-igraph

In this tutorial, we will present the igraph module in Python in introductory level. igraph is a collection of tools for graph and network analysis. At first, we need to import the module this module:

```
In [22]: import igraph as ig
         print ig.__version__
```

```
0.7.1
```

1.1 Undirected Graphs

Let's start with a very simple part of igraph. By *Graph*() method, we can create a simple undirected graph.

```
In [23]: g = ig.Graph()
         print g
```

```
IGRAPH U--- 0 0 --
```

By **print** *g*, we can get some general information about g. As you see in the above, graph *g* is undirected with no vertex and no edge.

Then, by "*add_vertices*()" method, we can request to add vertices to the graph. In the following example, 6 vertices are added to the graph *g*.

```
In [24]: g.add_vertices(6)
         print g
```

```
IGRAPH U--- 6 0 --
```

As you see, right now *g* has 6 vertices and 0 edge. now, it is time to add some edges to *g*. By *add_edge*, we just add one edge to the graph. By *add_edges*, we add a list of edges to the graph. Edges are defined by their end-points.

```
In [25]: g.add_edge(0,1)
         g.add_edges([(1,2),(0,2),(0,3),(1,4),(4,5),(0,5)])
         print g

IGRAPH U--- 6 7 --
+ edges:
0--1 1--2 0--2 0--3 1--4 4--5 0--5
```

Right now, we introduce another function called *summary*(). As you see in the following, the difference between **print** *g.summary*() is that in the *summary*() the list of edges are not shown anymore. This method can be useful when the graph is very big with a lot of vertices and edges.

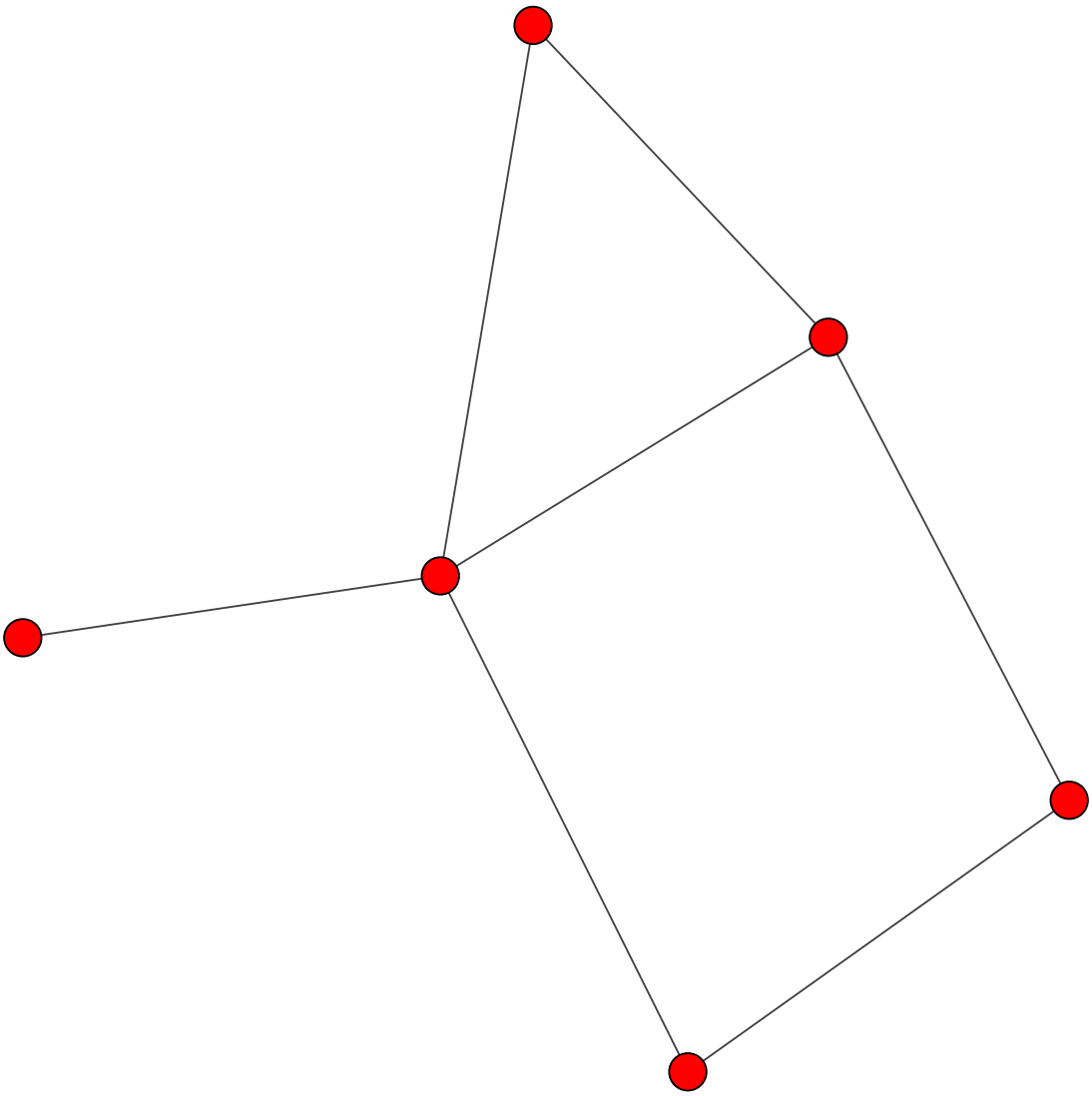
```
In [26]: print g.summary()

IGRAPH U--- 6 7 --
```

There are three useful methods for checking is the graph is directed, weighted and connected. The result is a boolean value. Besides by *degree*() function, we can have the degrees of vertices.

```
In [27]: print g.degree()
         print g.is_directed()
         print g.is_weighted()
         print g.is_connected()

[4, 3, 2, 1, 2, 2]
False
False
True
```

Now, by *plot*() method, we can have a graphical presentation of the graph:

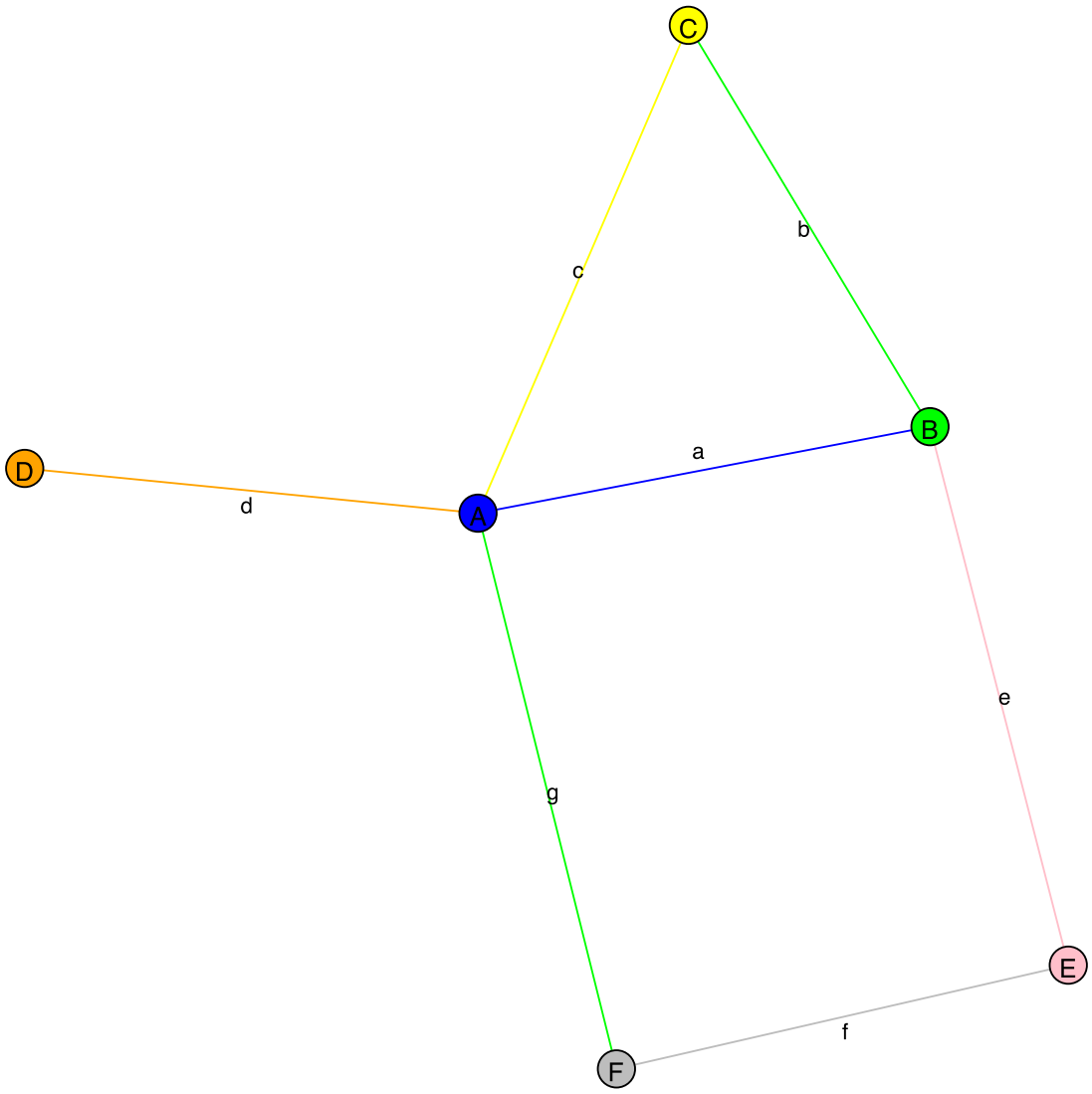
```
In [28]: ig.plot(g)

   Out[28]:
```



In the next step, we are going to give labels to the vertices and edges and then plot the graph again. We are going to color the vertices and edges as well.

```
In [29]: g.vs["label"]=["A","B","C","D","E","F"]
         g.es["label"]=["a","b","c","d","e","f","g"]
         ig.plot(g,vertex_color=["blue","green","yellow","orange","pink","gray"],
                 edge_color=["blue","green","yellow","orange","pink","gray","green"])
```

Out[29]:



1.2 Directed Graphs

In this section, we will see how we can build a directed graph and do whatever we have just done on undirected graphs. At first, we need to build a directed graph:

```
In [30]: D = ig.Graph(directed=True)
         print D

IGRAPH D--- 0 0 --
```

Now, by *add_vertices*(), *add_edge*() and *add_edges*(), we can add vertices and edges to the graph. The important difference here is that when we add edges, the order of vertices are important.

```
In [31]: D.add_vertices(6)
         D.add_edge(0,1)
         D.add_edges([(1,2),(0,2),(0,3),(1,4),(4,5),(0,5)])
```

Like before, we assign labels to the vertices and edges of graph:

```
In [32]: D.vs["label"]=["A","B","C","D","E","F"]
         D.es["label"]=["a","b","c","d","e","f","g"]
```

Now, we check if the graph D is directed, weighted and connected:

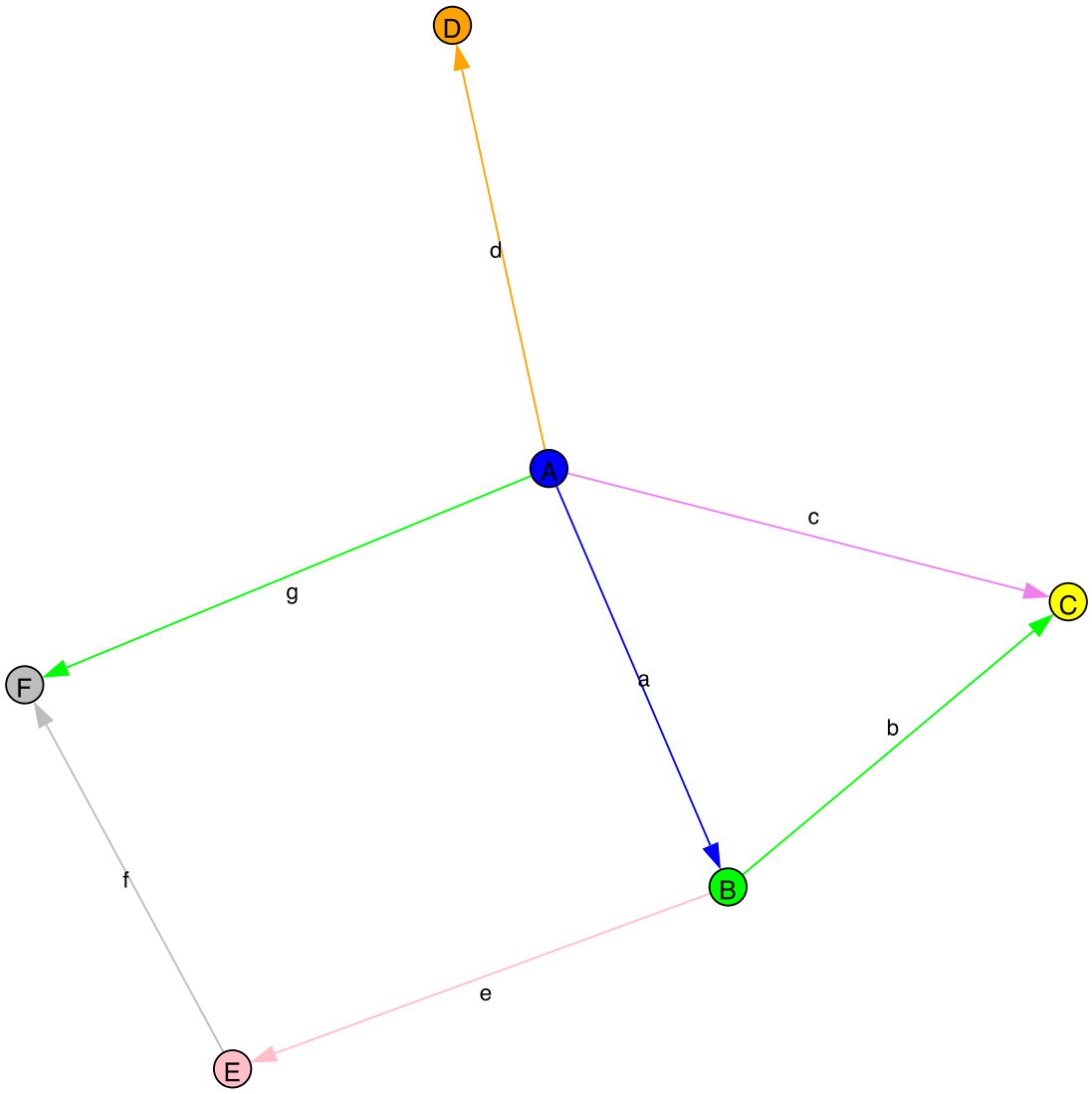
```
In [33]: print D.is_directed()
         print D.is_weighted()
         print D.is_connected()
```

```
True
False
False
```

Here, D is not connected, because D is directed and in D there are some vertices with only incoming edges. Therefore, from these vertices there is no path to other vertices. In order to check this out graphically, we plot D.

```
In [34]: ig.plot(D,vertex_color=["blue","green","yellow","orange","pink","gray"],
                  edge_color=["blue","green","violet","orange","pink","gray","green"])
```

Out[34]:



The vertices C, D and F, have only incoming edges. Now, by *degree*() method, we can see the degree of all vertices, but for a directed graph, it is better to have incoming and outgoing degrees of each vertex. Here, we calculate all these 3 cases:

```
In [35]: print(D.degree())
         print(D.degree(mode="in"))
         print(D.degree(mode="out"))
```

```
[4, 3, 2, 1, 2, 2]
[0, 1, 2, 1, 1, 2]
[4, 2, 0, 0, 1, 0]
```

As you see, for every vertex v, $d(v) = d_{in}(v) + d_{out}(v)$.

1.3 Some More Methods

In this section, we just introduce some more methods. The first two are *vcount*() and *ecount*() which count the number of vertices and edges, respectively in a graph. In the following we see the number of vertices and edges in both graphs g and D:

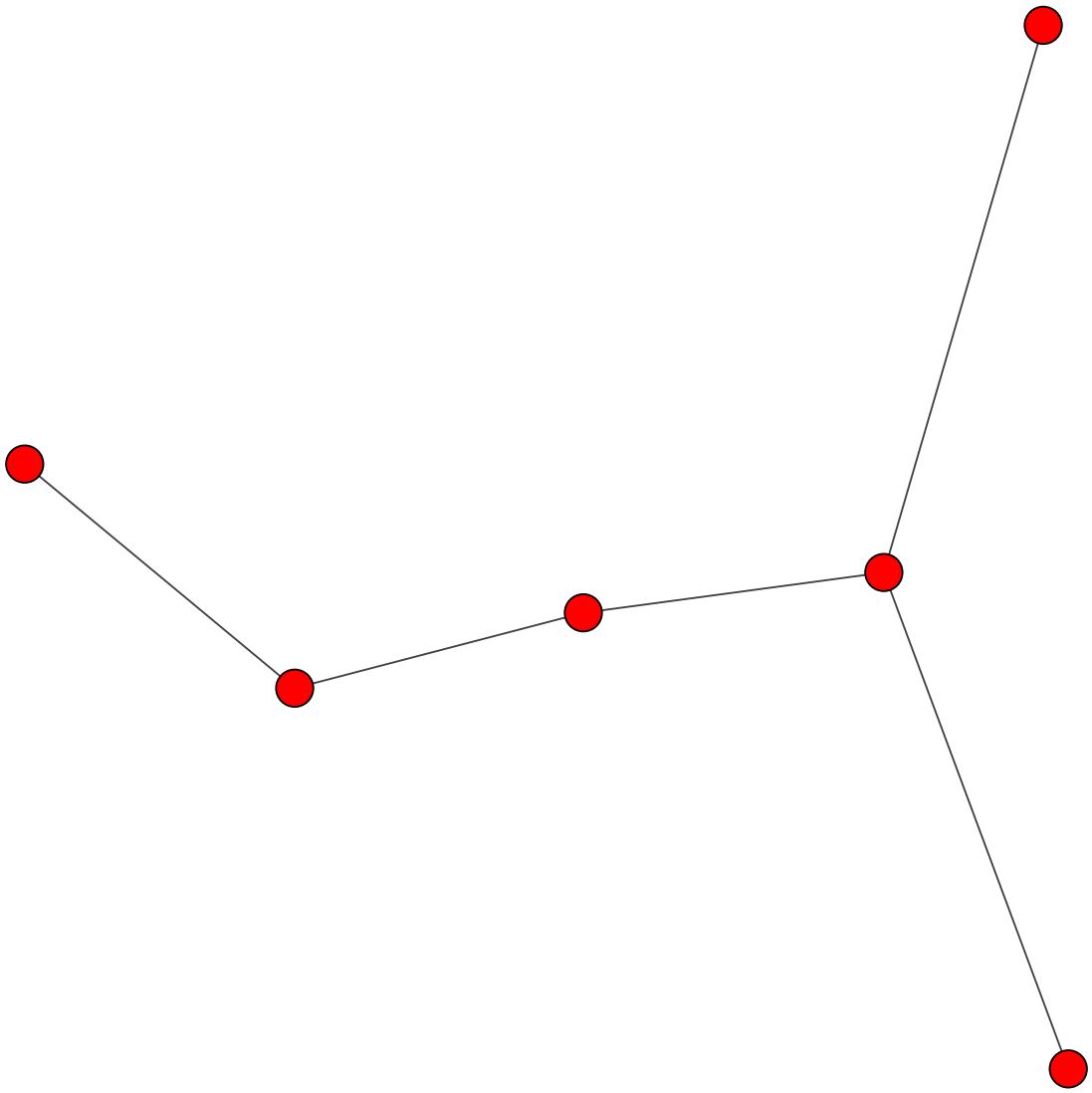
```
In [36]: print "Number of vertices and edges in g are ",g.vcount(), " and ", g.ecount()
         print "Number of vertices and edges in D are ",D.vcount(), " and ", D.ecount()

Number of vertices and edges in g are  6  and  7
Number of vertices and edges in D are  6  and  7
```

Meanwhile, there are some special graphs that we can construct them directly, for example complete graph which is built by *Full*() method, and tree which is built by *Tree*() method. As an example, consider the two following graphs:

```
In [37]: F=ig.Graph.Full(4)
         T=ig.Graph.Tree(6,2)
```

Here F is a K_4 graph, while T is a tree with 6 vertices and each vertex, except leaves, has two children. Let's plot T:

```
In [38]: ig.plot(T)

   Out[38]:
```



2 Practical Example

One of the most important applications of graph theory is finding the shortest path between two points.

```
In [39]: g=ig.Graph(16)
         g.add_edges([(0,1),(1,2),(2,3),(0,4),(1,5),(2,6),(3,7)
                      ,(4,5),(5,6),(6,7),(4,8),(5,9),(6,10),(7,11)
                      ,(8,9),(9,10),(10,11),(8,12),(9,13),(10,14),(11,15)
                      ,(12,13),(13,14),(14,15)])
         g.es["weight"]=[1, 4, 2, 6, 1, 8, 3, 1, 9, 8, 4, 7, 5, 8, 5, 4, 6, 9, 1, 6, 6, 2, 7, 3]
```

```
In [40]: shortetPath=g.get_all_shortest_paths(0, to=15,weights=g.es["weight"], mode=3)
         print(shortetPath)
```

```
[[0, 1, 5, 9, 13, 14, 15]]
```

```
In [41]: shortestLength=g.shortest_paths_dijkstra(source=0, target=15, weights=g.es["weight"], m
         print(shortestLength)
```

```
[[20.0]]
```