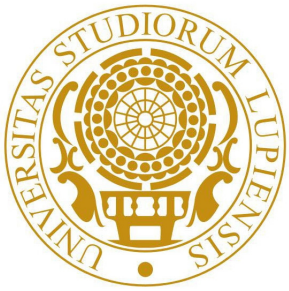
UNIVERSITÀ DEL SALENTO



Facoltà di Ingegneria

Corso di Laurea Magistrale in Computer Engineering

Advanced Control Techniques Project

Undirected and Directed Graphs with NetworkX

Professor: **Gianfranco Parlangeli**

Student:
Salvatore Corvaglia

Academic year 2019/2020

Undirected and Directed Graphs with NetworkX

June 29, 2020

1 Introduction

The linked structures that we have studied so far, such as linked lists and binary trees, consist of nodes that are connected to one another in a strictly specified pattern. But more general structures, consisting of nodes that can be linked in arbitrary ways, are often useful.

These structures are called **graphs**. A graph consists of a set of nodes and a set of links that connect them. In the context of graphs, though, the nodes are called **vertices** and the links are called **edges**. An edge connects a pair of vertices. Graphs come in two main varieties: directed and undirected.

1.0.1 Undirected Graph

In an undirected graph all connections are bidirectional. Undirected graphs can be represented as directed graphs, because if (u, v) is an edge in an undirected graph, it would be the same as having a directed graph with the edges (u, v) and (v, u).

The degree of a node $deg(v)$ is the number of nodes connected to that node. In a directed graph we have both outdegree (the number of edges pointing away from the node) and indegree (the number of edges pointing towards the node). A cycle is a path in a graph that goes from a node to itself.

An undirected graph, G, consists of a set, V , of vertices and a set, E, of edges, where each edge is a set of two vertices. If v_1v_2 is an edge, then the vertices v_1 and v_2 are called the endpoints of the edge.

NB: an edge in an undirected graph must have two distinct endpoints. That is, we do not allow an edge in an undirected graph to connect a vertex to itself.

1.0.2 Directed Graph

In a directed graph the connections between vertices are one-way and can be visualized as arrows linking pairs of vertices.

A directed graph, G, consists of a set, V , of vertices and a set, E, of edges, where each edge is an ordered pair of vertices.

We write $G = (V, E)$ to indicate that V is the set of vertices in G and E is the set of edges in G. If v_1v_2 is an edge in a directed graph, then the vertex v_1 is called the tail and v_2 is called the head of the edge.

We say that a directed graph is **strongly connected** if for any pair of vertices u and v, there is both a path from u to v and a path from v to u in the graph.

NB: it is possible for an edge in a directed graph to lead from a vertex back to the same vertex. That is, the head and the tail of the edge can be the same vertex.

1.0.3 Some properties

There are many basic definitions related to graphs. A **path** in a graph (directed or undirected) is a sequence of vertices $v_0v_1...v_n$ such that for each $i = 1, 2, ..., n$, there is an edge in the graph from v_{i-1} to v_i. The **length** of this path is n. That is, the length of a path is the number of *edges* in the path. A path is said to be **closed** if the starting and ending vertex of the path are the same.

The path $v_0v_1...v_n$ is said to be **simple** if there are no repeated vertices in the path, except possibly that the starting vertex v_0 can be equal to the ending vertex v_n. A simple closed path is called a **cycle**. A graph that contains no cycles is said to be **acyclic**. A directed acyclic graph, that is, a directed graph that contains no cycles, is referred to as a **dag**. If u and v are vertices in a graph, then v is said to be reachable from u if either $u = v$ or there is a path in the graph from u to v. An undirected graph is said to be **connected** if any vertex in the graph is **reachable** from any other vertex.

We say that a directed graph is **strongly connected** if for any pair of vertices u and v, there is both a path from u to v and a path from v to u in the graph. Finally, we consider one more variation on graphs. A **weighted graph** is a graph in which each edge is labeled with a number. The number associated with an edge is called the **weight** of that edge. In a weighted graph the length of a path refers to the sum of the weights of all the edges in that path.

1.0.4 Representations of Graphs

When graphs are used in computer programs, they must be represented by data structures. The simplest representation for a graph is an **adjacency matrix**. A matrix is simply a two-dimensional array. An adjacency matrix for a graph is a two-dimensional array of boolean values in which the number of rows and the number of columns are both equal to the number of vertices in the graph.

Note that the vertices of a graph are not listed in any particular order as far as the graph itself goes. However, in order to represent the graph as an adjacency matrix, we must assign some order to the vertices. Suppose that the graph is $G = (V, E)$. Let's say that there are n vertices and that the set of vertices is $V = v_0, v_1, ..., v_{n-1}$. Then we define the adjacency matrix representation of G to be an n-by-n array, A, of boolean values such that $A[i][j]$ is true if and only if there is an edge in E from v_i to v_j. We will write the boolean values in adjacency matrices as 0 and 1 rather than as *false* and *true*.

Note that we can use adjacency matrices to represent both directed and undirected graphs. The adjacency matrix for an undirected graph is symmetric; that is, $A[i][j] = A[j][i]$ for all i and j. Furthermore, the adjacency matrix for an undirected graph satisfies the condition that $A[i][i] = 0$ for all i, since an undirected graph cannot contain an edge from a vertex to itself. The adjacency matrix for a weighted graph is a two-dimensional array of numbers. Suppose A is such a matrix for a weighted graph and that there is an edge from vertex v_i to vertex v_j in that graph. Then $A[i][j]$ is the weight of the edge from v_i to v_j. In the case where there is no edge from v_i to v_j, we take the value of $A[i][j]$ to be ∞, even though we might not be able to represent an infinite value in a program.

The adjacency matrix representation is easy to use, but it is not appropriate for **sparse** graphs. A graph is said to be sparse if the number of edges in the graph is a small fraction of the maximum

possible number of edges. In practice, sparse graphs are very common, and it is worthwhile to have a representation designed especially for them. An alternative representation for graphs is the **edge list** representation.

1.0.5 Introduction to NetworkX

Graph theory deals with various properties and algorithms concerned with Graphs. Although it is very easy to implement a Graph ADT (Abstract Data Type) in Python, we will use NetworkX library for Graph Analysis as it has inbuilt support for visualizing graphs.

NetworkX provides data structures for graphs (or networks) along with graph algorithms, generators and drawing tools.

The structure of NetworkX can be seen by the organization of its source code. The package provides classes for graph objects, generators to create standard graphs, IO routines for reading in existing datasets, algorithms to analyse the resulting networks and some basic drawing tools.

1.1 Undirected Graph

First lets import some useful packages:

[1]:
```
import sys
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
import scipy
```

Initialize a basic empty graph:

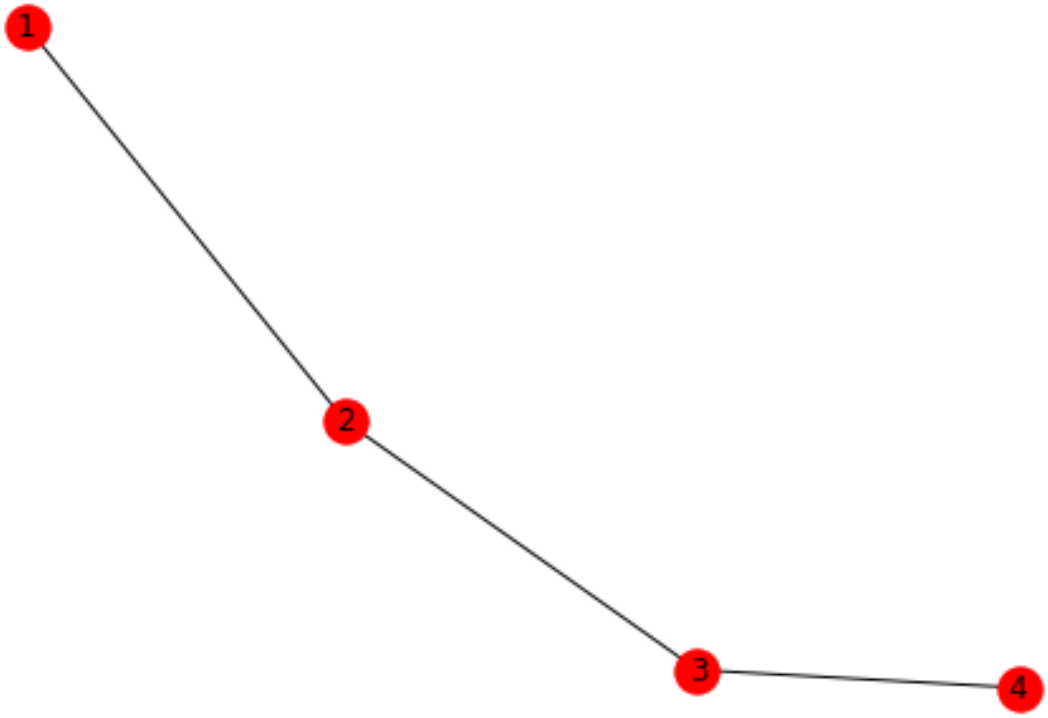
[2]:
```
Graph = nx.Graph()
```

Add edges to the graph note (no nodes have been specified but it automatically creates nodes and edges for the graph):

[3]:
```
Graph.add_edge(1,2)
Graph.add_edge(2,3)
Graph.add_edge(3,4)
```

Draw the graph and plot it:

[4]:
```
nx.draw(Graph, with_labels= True, node_color='red')
plt.show()
```



Print the basic information of the graph:

[5]:
```
print(nx.info(Graph))
```

```
Name:
Type: Graph
Number of nodes: 4
Number of edges: 3
Average degree:   1.5000
```

Define a Laplacian Matrix. The graph Laplacian is the matrix $L = D - A$, where A is the adjacency matrix and D is the diagonal matrix of node degrees.

[6]:
```
L = nx.laplacian_matrix(Graph)
print(L)
```

```
  (0, 0)        1
  (0, 1)        -1
  (1, 0)        -1
  (1, 1)        2
  (1, 2)        -1
  (2, 1)        -1
  (2, 2)        2
  (2, 3)        -1
```

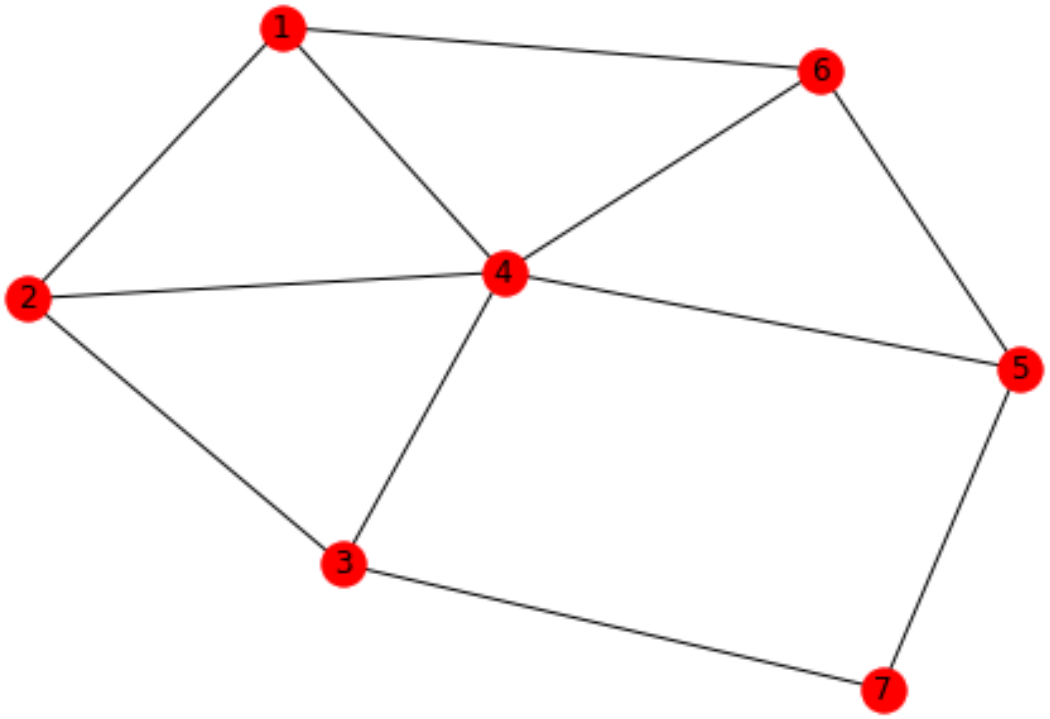
```
(3, 2)        -1
(3, 3)        1
```

Now create a new graph G:

[7]:
```
G = nx.Graph()
G.add_edges_from([(1,2),(1,4),(1,6),(2,4),(2,3),
                  (3,4),(3,7),(4,5),(4,6),(5,6),
                  (5,7)])
```

Draw the graph including its labels:

[8]:
```
nx.draw(G,with_labels= True, node_color='red')
plt.show()
```



Get some basic information of this graph:

[9]:
```
print(nx.info(G))
```

```
Name:
Type: Graph
Number of nodes: 7
Number of edges: 11
Average degree:   3.1429
```

Write a for loop to print the names and degrees of all the nodes in the graph G:

[10]:
```
for i in range(1,len(G.nodes)+1):
  print("Node " + str(i) + "'s degree is: ", G.degree(i))
```

```
Node 1's degree is:  3
Node 2's degree is:  3
Node 3's degree is:  3
Node 4's degree is:  5
Node 5's degree is:  3
Node 6's degree is:  3
Node 7's degree is:  2
```

Define a general function to iterate over and print the names and degrees of all the nodes in any graph:

[11]:
```
def NodeDegreeCalc(Graph):
  print("Degrees of nodes in the graph are:")
  for i in range(1,len(Graph.nodes)+1):
    print("Node " + str(i) + "'s degree is: ", Graph.degree(i))
NodeDegreeCalc(G)
```

```
Degrees of nodes in the graph are:
Node 1's degree is:  3
Node 2's degree is:  3
Node 3's degree is:  3
Node 4's degree is:  5
Node 5's degree is:  3
Node 6's degree is:  3
Node 7's degree is:  2
```

[12]:
```
def NodeDegreeCalcList(Graph):
  ListOfNodes = [None]*(len(Graph.nodes))

  for i in range(1,len(Graph.nodes)+1):
    ListOfNodes[i-1] = Graph.degree(i)

  Average_Degrees = sum(ListOfNodes)/len(Graph.nodes)
  print("The average degrees in the entire graph is: " + str(Average_Degrees))
NodeDegreeCalcList(Graph)
```

```
The average degrees in the entire graph is: 1.5
```

The normalized graph Laplacian is the matrix:

$N = D^{-1/2}LD^{-1/2}$

where L is the graph Laplacian and D is the diagonal matrix of node degrees.

[13]:
```
L = nx.normalized_laplacian_matrix(G)
e = np.round(np.linalg.eigvals(L.A),1)
print("Eigenvalues:", e)
print("Largest eigenvalue:", max(e))
print("Smallest eigenvalue:", min(e))
```

```
Eigenvalues: [0.  1.8 0.6 1.5 1.2 0.7 1.3]
Largest eigenvalue: 1.8
Smallest eigenvalue: 0.0
```

1.2 Directed Graph

This section is focused on the creation and manipulation of Directed Grap.

[14]:
```
DG = nx.DiGraph()
```

Add the edges to the graph:

[15]:
```
DG.add_edges_from([(1,2),(1,4),(2,4),(3,1),(4,1),(4,5)])
```

Calculate the Adjacency Matrix:

[16]:
```
Adj = nx.adjacency_matrix(DG)
```

Note that *Adj* is now of the form (edges) 1 all zero edges are excluded:

[17]:
```
print(Adj)
```

```
  (0, 1)        1
  (0, 2)        1
  (1, 2)        1
  (2, 0)        1
  (2, 4)        1
  (3, 0)        1
```

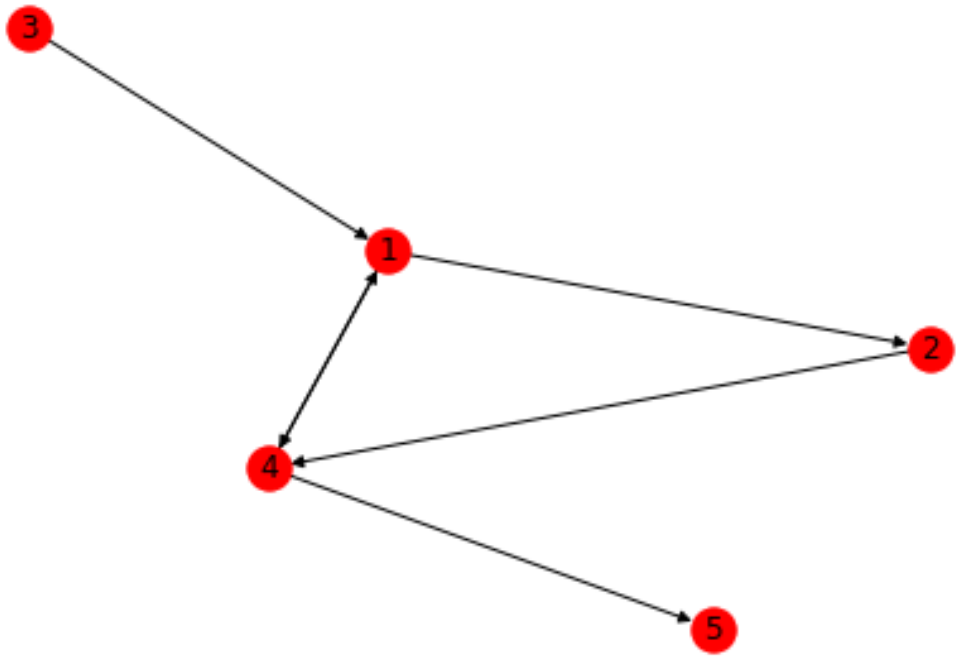
Convert Adjacency Matrix to a density matrix using the *.todense*():

[18]:
```
print(Adj.todense())
```

```
[[0 1 1 0 0]
 [0 0 1 0 0]
 [1 0 0 0 1]
 [1 0 0 0 0]
 [0 0 0 0 0]]
```

Render and output the graph:

[19]:
```
nx.draw(DG,with_labels= True, node_color='red')
```



The graph directed Laplacian is the matrix:

$L = I - (\Phi^{1/2} P \Phi^{-1/2} + \Phi^{-1/2} P^T \Phi^{1/2})/2$

where I is the identity matrix, P is the transition matrix of the graph, and Φ a matrix with the Perron vector of P in the diagonal and zeros elsewhere.

[20]:
```
L_3 = nx.directed_laplacian_matrix(DG)
print(L_3)
```

```
[[ 0.99       -0.30041709 -0.48932563 -0.22313518 -0.09581722]
 [-0.30041709  0.99       -0.35071726 -0.011969   -0.1148495 ]
 [-0.48932563 -0.35071726  0.99       -0.01487968 -0.38596282]
 [-0.22313518 -0.011969   -0.01487968  0.99       -0.20703795]
 [-0.09581722 -0.1148495  -0.38596282 -0.20703795  0.8       ]]
```

[21]:
```
e_3 = np.round(np.linalg.eigvals(L_3.A),0)
print("Eigenvalues:", e_3)
print("Largest eigenvalue:", max(e_3))
print("Smallest eigenvalue:", min(e_3))
```

```
Eigenvalues: [0. 2. 1. 1. 1.]
Largest eigenvalue: 2.0
Smallest eigenvalue: 0.0
```

Get the degree centrality of your graph's nodes.The degree centrality for a node v is the fraction of nodes it is connected to:

[22]:
```
DegreeCentralityDict = nx.degree_centrality(DG)
print(DegreeCentralityDict)
```

```
{1: 1.0, 2: 0.5, 4: 1.0, 3: 0.25, 5: 0.25}
```